



1장. 파이썬 기반의 머신러닝과 생태계 이해

EURON 6기 초급세션 1주차 예습과제
2198015 김수연

01. 머신러닝의 개념



머신러닝이란, 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법을 통칭

- 지도 학습 : 분류, 회귀, 추천 시스템, 시각/음성 감지/인지, 텍스트분석, NLP
- 비지도 학습 : 클러스터링, 차원 축소, 강화학습
- 강화학습

02. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

- 머신러닝 패키지: 사이킷런, 텐서플로, 케라스
- 행렬/선형대수/통계 패키지: 넘파이, 사이파이
- 데이터 핸들링: 판다스
- 시각화: 맷플롯립, 시본
- 서드파티 라이브러리
- 주피터 노트북

03. 넘파이



넘파이란, 선형대수 기반의 프로그램을 쉽게 만들 수 있도록 지원하는 대표적인 패키지

- 루프를 사용하지 않고 대량 데이터의 배열 연산을 가능하게 하므로 빠른 배열 연산 속도를 보장
- C/C++과 같은 저수준 언어 기반의 호환 API를 제공하여 타프로그램과 데이터를 주고받거나 API를 호출해 쉽게 통합할 수 있는 기능 제공, 파이썬 언어 자체의 수행 성능 제약으로 수행 성능이 중요한 부분을 C/C++ 기반의 코드로 작성하고 이를 넘파이에서 호출하는 방식으로 통합
- 데이터 핸들링 기능 제공, 그러나 행렬 기반의 데이터 처리는 판다스가 더 특화되어 있다.

```
import numpy as np
```

- `as np` 를 추가해 약어로 모듈을 표현해주는 것이 관례

```
array1 = np.array([1, 2, 3])
print('array1 type: ', type(array1))
print('array1 array 형태: ', array1.shape)

array2 = np.array([[1, 3, 3], [2, 3, 4]])
print('array2 type: ', type(array2))
print('array2 array 형태: ', array2.shape)

array3 = np.array([[1, 2, 3]])
print('array3 type: ', type(array3))
print('array3 array 형태: ', array3.shape)
```

```
'''
array1 type: <class 'numpy.ndarray'>
array1 array 형태: (3,)
array2 type: <class 'numpy.ndarray'>
array2 array 형태: (2, 3)
array3 type: <class 'numpy.ndarray'>
array3 array 형태: (1, 3)
'''
```

- ndarray로 변환을 원하는 객체를 인자로 입력하면 ndarray를 반환
- `ndarray.shape` 는 ndarray의 차원과 크기를 (row, column)의 튜플(tuple) 형태로 나타낸다.
- array1과 array3은 동일한 데이터 건수를 가지고 있지만, array1은 명확하게 1차원임을 표현

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

- 데이터 값은 숫자 값, 문자열 값, 불 값 모두 가능
- 숫자형의 경우 int형, unsigned int형, float형, complex 타입 제공
- `reshape()` 함수를 통해 차원의 차수 변경 가능

ndarray의 데이터 타입

```
list1 = [1, 2, 3]
print(type(list1))
array1 = np.array(list1)
print(type(array1))
print(array1, array1.dtype)

'''
<class 'list'>
<class 'numpy.ndarray'>
[1 2 3] int64
'''
```

- 리스트 자료형을 ndarray로 변환

```
list2 = [1, 2, 'test']
array2 = np.array(list2)
print(array2, array2.dtype)
list3 = [1, 2, 3.0]
array3 = np.array(list3)
print(array3, array3.dtype)

'''
['1' '2' 'test'] <U21
[1. 2. 3.] float64
'''
```

- 서로 다른 데이터 타입을 가질 수 있는 리스트와는 다르게 ndarray 내의 데이터 타입은 그 연산의 특성상 같은 데이터 타입만 가능, 만약 다른 데이터 유형이 섞여 있는 리스트를 ndarray로 변경하면 데이터 크기가 더 큰 데이터 타입으로 형 변환을 일괄 적용

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

array_int1 = array_float.astype('int32')
```

```
print(array_int1, array_int1.dtype)

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.] float64
[1 2 3] int32
[1 2 3] int32
'''
```

- `astype()` 메서드를 통해 ndarray 내 데이터값 타입 변경함으로써 메모리를 절약할 수 있다.

ndarray를 편리하게 생성하기 - `arange`, `zeros`, `ones`

```
sequence_array = np.arange(10)
print(sequence_array)
print(sequence_array.dtype, sequence_array.shape)

'''
[0 1 2 3 4 5 6 7 8 9]
int64 (10,)
'''
```

- ndarray를 연속값이나 0 또는 1로 초기화해 쉽게 생성해야 할 필요가 있는 경우, `arange()`, `zeros()`, `ones()`를 이용해 쉽게 ndarray를 생성할 수 있다. 주로 테스트용 데이터를 만들거나 대규모의 데이터를 일괄적으로 초기화하기 위해 사용.

📌 `arange()` 는 `range()`와 유사한 기능, array를 `range()`로 표현하는 것

- default 함수 인자는 stop 값, 0부터 함수 인자 -1까지의 값을 순차적으로 ndarray의 데이터값으로 변환
- `range`와 유사하게 start 값을 부여할 수도 있다.

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

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

'''
[[0 0]
 [0 0]
 [0 0]]
int32 (3, 2)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
float64 (3, 2)
'''
```

📌 `zeros()` 는 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape을 가진 ndarray를 반환

📌 `ones()` 도 마찬가지로 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 1로 채운 해당 shape을 가진 ndarray를 반환

- 함수 인자로 dtype을 정해주지 않으면 default로 float64 형의 데이터로 ndarray를 채운다.

📌 ndarray의 차원과 크기를 변경하는 `reshape()`

```
array1 = np.arange(10)
print('array1:\n', array1)
```

```
array2 = array1.reshape(2, 5)
print('array2:\n', array2)
```

```
array3 = array1.reshape(5, 2)
print('array3:/n', array3)
```

```
'''
array1:
[0 1 2 3 4 5 6 7 8 9]
array2:
[[0 1 2 3 4]
 [5 6 7 8 9]]
array3:/n [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
'''
```

- ndarray를 특정 차원 및 크기로 변환, 변환을 원하는 크기를 함수 인자로 부여
- 지정된 사이즈로 변경이 불가능하면 오류가 발생

```
array1 = np.arange(10)
print(array1)
array2 = array1.reshape(-1, 5)
print('array2 shape: ', array2.shape)
array3 = array1.reshape(5, -1)
print('array3 shape: ', array3.shape)
```

```
'''
[0 1 2 3 4 5 6 7 8 9]
array2 shape: (2, 5)
array3 shape: (5, 2)
'''
```

- 인자로 -1을 적용하면 원래 ndarray와 호환되는 새로운 shape로 변환해준다.
- 그러나 -1을 사용하더라도 호환될 수 없는 형태의 경우 에러를 발생시킨다.

```
array1 = np.arange(8)
array3d = array1.reshape((2, 2, 2))
print('array3d:\n', array3d.tolist())
```

```
# 3차원 ndarray를 2차원 ndarray로 변환
array5 = array3d.reshape(-1, 1)
print('array5:\n', array5.tolist())
print('array5 shape: ', array5.shape)
```

```
#1차원 ndarray를 2차원 ndarray로 변환
array6 = array1.reshape(-1, 1)
print('array6:\n', array6.tolist())
print('array6 shape: ', array6.shape)
```

```
'''
array3d:
[[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
array5:
[[0], [1], [2], [3], [4], [5], [6], [7]]
array5 shape: (8, 1)
array6:
[[0], [1], [2], [3], [4], [5], [6], [7]]
'''
```

```
array6 shape: (8, 1)
'''
```

- `reshape(-1, 1)`는 원본 ndarray가 어떤 형태라도 2차원이고, 여러 개의 row를 가지되 반드시 1개의 column을 가진 ndarray로 변환
- 여러 개의 넘파이 ndarray는 `stack`이나 `concat`으로 결합할 때 각각의 ndarray의 형태를 통일해 유용하게 사용된다.
- 위의 예제는 `reshape(-1, 1)`을 사용해 3차원을 2차원으로, 1차원을 2차원으로 변경한다.

넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

- **특정한 데이터만 추출**: 원하는 위치의 인덱스 값을 지정하면 해당 위치의 데이터가 반환
- **슬라이싱**: 연속된 인덱스상의 ndarray를 추출. ':' 기호 사이에 시작 인덱스와 종료 인덱스를 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의 ndarray를 반환
- **팬시 인덱싱**: 일정한 인덱스 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray를 반환
- **불린 인덱싱**: 특정 조건에 해당하는지 여부인 True/False 값 인덱스 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray를 반환

📌 단일 값 추출

- 단일 인덱스를 이용해 ndarray 내의 데이터값도 간단히 수정 가능
- axis 0은 row 방향, axis 1은 column 방향의 축을 의미

```
# 1부터 9까지의 1차원 ndarray 생성
array1 = np.arange(start=1, stop=10)
print('array1: ', array1)
# index는 0부터 시작하므로 array1[2]는 3번째 index 위치의 데이터값을 의미
value = array1[2]
print('value: ', value)
print(type(value))

'''
array1:  [1 2 3 4 5 6 7 8 9]
value:  3
<class 'numpy.int64'>
'''
```

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

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

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

print('(row=1, 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])

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

```
[7 8 9]]
(row=1, 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
'''
```

📌 슬라이싱

- ':' 기호를 이용해 연속한 데이터를 슬라이싱해서 추출
- 단일 데이터값 추출을 제외하고 슬라이싱, 팬시 인덱싱, 불린 인덱싱으로 추출된 데이터 세트는 모두 ndarray 타입
- 기호 앞/뒤에 시작/종료 인덱스 생략하면 자동으로 맨/처음 마지막 인덱스로 간주

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

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

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

array5 = array1[3:]
print(array5)

array6 = array1[:]
print(array6)

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

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

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[:, 1:]\n', array2d[:, 1:])
print('array2d[:, 0]\n', array2d[:, 0])

'''
array2d:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
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]
'''
```

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

- 2차원 ndarray에서 뒤에 오는 인덱스를 없애면 1차원 ndarray를 반환

📌 팬시 인덱싱

- 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] # (0,2)와 (1,2)로 반영
print('array2d[[0, 1], 2] => ', array3.tolist())

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

array5 = array2d[[0, 1]] # ((0,:), (1,:))로 반영
print('array2d[[0, 1]] => ', array5.tolist())

'''
array2d[[0, 1], 2] => [3, 6]
array2d[[0, 1], 0:2] => [[1, 2], [4, 5]]
array2d[[0, 1]] => [[1, 2, 3], [4, 5, 6]]
'''
```

📌 불린 인덱싱

- 조건 필터링과 검색을 동시에 할 수 있기 때문에 매우 자주 사용
- for loop/if else 문보다 훨씬 간단하게 구현 가능
- ndarray의 인덱스를 지정하는 `[]` 내에 조건문을 기재
- True 값에 해당하는 인덱스 값을 저장, 저장된 인덱스 데이터 세트에 ndarray 조회

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

행렬의 정렬 - sort()와 argsort()

행렬 정렬

- `np.sort()` 는 원 행렬을 그대로 유지한 채 원 행렬의 정렬된 행렬 반환
- `ndarray.sort()` 는 원 행렬 자체를 정렬한 형태로 변환하며 반환 값은 None

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

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)

sort_array2 = org_array.sort()
print('org_array.sort() 호출 후 반환된 행렬: ', sort_array2)
print('ort_array.sort() 호출 후 원본 행렬: ', org_array)

'''
원본 행렬:  [3 1 9 5]
np.sort() 호출 후 반환된 정렬 행렬:  [1 3 5 9]
np.sort() 호출 후 원본 행렬:  [3 1 9 5]
org_array.sort() 호출 후 반환된 행렬:  None
ort_array.sort() 호출 후 원본 행렬:  [1 3 5 9]
'''
```

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

- 기본적으로 오름차순으로 행렬 내 원소를 정렬
- 내림차순으로 정렬하기 위해서는 `[::-1]` 을 적용

```
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('칼럼 방향으로 정렬:\n', sort_array2d_axis1)

'''
로우 방향으로 정렬:
[[ 7  1]
 [ 8 12]]
칼럼 방향으로 정렬:
[[ 8 12]
 [ 1  7]]
'''
```

- 행렬이 2차원 이상일 경우 axis 축 값을 통해 로우 방향, 또는 칼럼 방향으로 정렬을 수행

정렬된 행렬의 인덱스를 반환하기


- `np.argsort()` 는 원본 행렬이 정렬된 후 기존 원본 행렬의 원소에 대한 인덱스가 필요할 때 사용

- 정렬 행렬의 원본 행렬 인덱스를 ndarray 형으로 반환
- 내림차순으로 정렬하기 위해서는 마찬가지로 `[::-1]` 을 사용
- 넘파이의 데이터 추출에서 유용하게 사용된다.


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

선형대수 연산 - 행렬 내적과 전치 행렬 구하기

행렬 내적(행렬 곱)

 `np.dot()` 을 이용해 두 행렬 A와 B의 내적 계산




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

→ 행렬 내적의 특성으로 왼쪽 행렬의 열 개수와 오른쪽 행렬의 행 개수가 동일해야 연산 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

 `np.transpose()` 을 이용해 행렬 A의 전치 행렬 계산



원 행렬에서 행과 열 위치를 교환한 원소로 구성된 행렬

```
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. 판다스

- 파이썬의 리스트, 컬렉션, 넘파이 등의 내부 데이터뿐만 아니라 CSV 등의 파일을 쉽게 DataFrame으로 변경해 데이터의 가공/분석을 편리하게 수행할 수 있게 한다.
- **DataFrame**은 여러 개의 행과 열로 이뤄진 2차원 데이터를 담는 데이터 구조체
- **Index**는 개별 데이터를 고유하게 식별하는 key 값
- Series는 칼럼이 하나뿐인 데이터 구조체이고, DataFrame은 칼럼이 여러 개인 데이터 구조체. DataFrame은 여러 개의 Series로 이루어져 있다.
- csv, tab과 같은 다양한 유형의 분리 문자로 칼럼을 분리한 파일을 손쉽게 DataFrame으로 로딩할 수 있다.

```
import pandas as pd
```

📌 `read_csv()` 는 CSV(칼럼을 ';'로 구분한 파일 포맷) 파일 포맷 변환을 위한 API

- CSV뿐만 아니라 어떤 필드 구분 문자 기반의 파일 포맷도 DataFrame으로 변환 가능
- 인자인 `sep`에 해당 구분 문자를 입력, 예를 들어 `read_csv('파일명', sep='\t')`
- `sep` 인자를 생략하면 자동으로 콤마로 할당
- 가장 중요한 인자는 `filepath`, 로드하려는 데이터 파일의 경로를 포함한 파일명 입력

📌 `read_table()` 의 디폴트 필드 구분 문자는 탭 문자

📌 `read_fwf()` 는 Fixed Width, 즉 고정 길이 기반의 칼럼 포맷을 DataFrame으로 로딩하기 위한 API

```
titanic_df = pd.read_csv('titanic_train.csv')
print('titanic 변수 type: ', type(titanic_df))
titanic_df # default는 5개
titanic_df.head(3) # 맨 앞의 3개 로우를 반환
```

- `pd.read_csv()`는 호출 시 파일명 인자로 들어온 파일을 로딩해 DataFrame 객체로 반환
- 별다른 파라미터 지정이 없으면 파일의 맨 처음 로우를 칼럼명으로 인지하고 칼럼으로 변환
- 판다스의 Index 객체 값을 맨 왼쪽에 순차적으로 표시

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

📌 `shape` 변수는 DataFrame의 행과 열의 크기를 튜플 형태로 반환

```
titanic_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

1 RangeIndex는 DataFrame index의 범위를 나타내므로 전체 row 수를 알 수 있습니다. 전체 데이터는 891개 row입니다. 그리고 Column 수는 12개입니다.

2 칼럼별 데이터 타입을 나타냅니다. 가령 Pclass 칼럼의 데이터 타입은 int64형입니다. Name 칼럼의 데이터 타입은 object 인데, 이는 문자열 타입으로 생각해도 무방합니다.

3 몇 개의 데이터가 non-null(Null 값이 아님)인지 나타냅니다. 가령 Age 칼럼의 714 non-null의 의미는 Age 칼럼 891개 데이터 중 714개가 Null이 아니며 177개는 Null이라는 의미입니다.

4 전체 12개의 칼럼들의 타입을 요약한 것입니다. 2개 칼럼이 float64, 5개 칼럼이 int64, 5개 칼럼이 object 타입입니다.

📌 `info()` 메서드를 통해 총 데이터 건수와 데이터 타입, Null 건수를 알 수 있다.

```
titanic_df.describe()
```

【Output】

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

📌 `describe()` 메서드를 통해 칼럼별 숫자형 데이터값의 n-percentile 분포도, 평균값, 최댓값, 최솟값을 나타낸다. 오직 숫자형(int, float 등) 칼럼의 분포도만 조사한다.

→ 데이터의 분포도를 아는 것은 머신러닝 알고리즘의 성능을 향상시키는 중요한 요소

→ 계량적인 수준의 분포도를 확인할 수 있어 유용

- 카테고리 칼럼은 특정 범주에 속하는 값을 코드화한 칼럼, 예를 들어 '남'을 1, '여'를 2와 같이 표현
- 이러한 카테고리 칼럼을 숫자로 표시할 수 있는데 `describe()`를 통해 확인 가능
- `Survived`의 경우 min 0, 25~75%도 0, max도 1이므로 0과 1로 이뤄진 숫자형 카테고리 칼럼
- `Pclass`의 경우도 min이 1, 25~75%가 2와 3, max가 3이므로 1, 2, 3으로 이뤄진 숫자형 카테고리 칼럼

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

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

- DataFrame의 [] 연산자 내부에 칼럼명을 입력하면 Series 형태로 특정 칼럼 데이터 세트가 반환
- Series는 Index와 단 하나의 칼럼으로 구성된 데이터 세트

📌 이렇게 반환된 Series 객체에 `value_counts()` 메서드를 호출하면 해당 칼럼값의 유형과 건수 확인

- Series 객체에서만 정의, DataFrame에서 사용 불가!
- 데이터의 분포도를 확인하는 데 매우 유용한 함수!

```
value_counts = titanic_df['Pclass'].value_counts()
print(type(value_counts))
print(value_counts)
'''
<class 'pandas.core.series.Series'>
Pclass
3    491
```

```
1    216
2    184
Name: count, dtype: int64
...
```

- `value_counts()` 가 반환하는 데이터 타입 역시 Series 객체
- `value_counts()` 는 칼럼 값별 데이터 건수를 반환하므로 고유 칼럼 값을 식별자로 사용할 수 있다.



즉, 인덱스는 순차 값과 같은 의미 없는 식별자만 할당하는 것이 아니라 고유성이 보장된다면 의미 있는 데이터값 할당도 가능

- DataFrame, Series가 만들어진 후에도 변경 가능
- 고유성이 보장된다는 가정 하에, 숫자 뿐만 아니라 문자열도 가능

DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

- DataFrame은 파이썬의 리스트, 딕셔너리 그리고 넘파이 ndarray 등 다양한 데이터로부터 생성될 수 있다. 또한 DataFrame은 반대로 파이썬의 리스트, 딕셔너리 그리고 넘파이 ndarray 등으로 변환될 수 있다.
- DataFrame과 넘파이 ndarray 상호 간의 변환은 매우 빈번하게 발생

넘파이 ndarray, 리스트, 딕셔너리를 DataFrame으로 변환하기

- DataFrame은 리스트와 넘파이 ndarray와 다르게 칼럼명을 가지고 있어 상대적으로 편하게 데이터 핸들링 가능
- DataFrame으로 변환 시 칼럼명 지정 (지정하지 않으면 자동으로 칼럼명 할당)
- 판다스 DataFrame 객체의 생성 인자 data는 리스트나 딕셔너리 또는 넘파이 ndarray를 입력받고, 생성 인자 columns는 칼럼명 리스트를 입력 받아서 DataFrame 생성
- DataFrame은 행과 열을 가지는 2차원 데이터이기 때문에 2차원 이하의 데이터들만 DataFrame으로 변환 가능

```
import numpy as np

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
...
```

- 1차원 형태의 데이터틀 기반으로 DataFrame을 생성하므로 컬럼명이 한 개만 필요하다는 사실에 주의

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

# 2행 x 3열 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
'''
```

- 2행 3열 2차원 형태의 리스트와 ndarray를 기반으로 DataFrame을 생성하므로 컬럼명은 3개 필요

```
# Key는 문자열 컬럼명으로 매핑, Value는 리스트 형(또는 ndarray) 컬럼 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
'''
```

- 딕셔너리를 DataFrame으로 변환 시에는 딕셔너리의 키(Key)는 컬럼명으로, 값(Value)은 키에 해당하는 컬럼 데이터로 변환
- 키의 경우는 문자열, 값의 경우 리스트(또는 ndarray) 형태로 딕셔너리를 구성

DataFrame을 넘파이 ndarray, 리스트, 딕셔너리로 변환하기

- 많은 머신러닝 패키지가 기본 데이터 형으로 넘파이 ndarray를 사용, 따라서 데이터 핸들링은 DataFrame을 이용하더라도 머신러닝 패키지의 입력 인자 등에 적용하기 위해 ndarray로 변환하는 경우 빈번히 발생
- DataFrame을 넘파이 ndarray로 변환하는 것은 DataFrame 객체의 values를 이용해 쉽게 할 수 있다.

📌 DataFrame을 ndarray로 변환: `.values`

```
array3 = df_dict.values
print('df_dict.values 타입: ', type(array3), 'df_dict.values shape: ', array3.shape)
print(array3)

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

📌 DataFrame을 리스트로 변환: `.values.tolist()`

values로 얻은 ndarray에 `tolist()`를 호출

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

📌 DataFrame을 딕셔너리로 변환: `.to_dict()`

이때 인자로 'list'를 입력하면 딕셔너리의 값이 리스트형으로 반환

```
dict3 = df_dict.to_dict('list')
print('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

📌 DataFrame의 칼럼 데이터 세트 생성과 수정 역시 `[]` 연산자를 이용

```
titanic_df['Age_0'] = 0
titanic_df.head(3)
```

- DataFrame[] 내에 새로운 칼럼명을 입력하고 값을 할당

```
titanic_df['Age_by_10'] = titanic_df['Age']*10
titanic_df['Family_No'] = titanic_df['SibSp'] + titanic_df['Parch'] + 1
titanic_df.head(3)
```

- 기존 칼럼 Series를 가공해 새로운 칼럼 Series 추가

```
titanic_df['Age_by_10'] = titanic_df['Age_by_10'] + 100
titanic_df.head(3)
```

- 기존 칼럼 값 일괄 업데이트, 원하는 칼럼 Series를 DataFrame[]내에 칼럼 명으로 입력한 뒤 값을 할당

DataFrame 데이터 삭제

📌 DataFrame에서 데이터의 삭제는 `.drop()` 메서드를 이용

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

📌 `drop()` 메서드에 `axis=1`을 입력하면 칼럼 축 방향으로 드롭을 수행하므로 칼럼을 드롭

- labels에 원하는 칼럼 명을 입력하고 `axis=1`을 입력하면 지정된 칼럼을 드롭
- 기존 칼럼 값을 가공해 새로운 칼럼을 만들고 삭제

📌 `drop()` 메서드에 `axis=0`을 입력하면 로우 축 방향으로 드롭을 수행하므로 특정 로우를 드롭

- DataFrame의 특정 로우를 가리키는 것은 인덱스, 따라서 자동으로 labels에 오는 값을 인덱스로 간주
- 이상치 데이터 삭제

```
titanic_drop_df = titanic_df.drop('Age_0', axis=1)
titanic_drop_df.head(3)
titanic_df.head(3)
```

📌 **inplace=False** (inplace는 디폴트 값이 False이므로 파라미터를 기재하지 않으면 자동으로 False)이면 자기 자신의 DataFrame의 데이터는 삭제하지 않으며, 삭제된 결과 DataFrame을 반환

📌 **inplace=True** 이면 자신의 DataFrame 데이터를 삭제

여러 개의 칼럼을 삭제하고 싶으면 리스트 형태로 삭제하고자 하는 칼럼 명을 입력해 labels 파라미터로 입력

```
drop_result = titanic_df.drop(['Age_0', 'Age_by_10', 'Family_No'], axis=1, inplace=True)
print('inplace=True로 drop 후 반환된 값: ', drop_result)
titanic_df.head(3)
# inplace=True로 drop 후 반환된 값: None
```

→ 반환 값이 None임을 주의! 반환 값을 다시 자신의 DataFrame 객체로 할당하지 않도록 주의!

```
pd.set_option('display.width', 1000)
pd.set_option('display.max_colwidth', 15)
print('#### before axis 0 drop ####')
print(titanic_df.head(3))

titanic_df.drop([0, 1, 2], axis=0, inplace=True)
print('#### after axis 0 drop ####')
print(titanic_df.head(3))
```

Index 객체

판다스의 Index 객체는 DataFrame, 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)
# Index 객체를 실제 값 array로 변환
print('Index 객체 array값:\n', indexes.values)
'''
RangeIndex(start=0, stop=891, step=1)
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]
'''
```

- 반환된 Index객체의 실제 값은 넘파이 1차원 ndarray로 볼 수 있다.
- Index 객체의 values 속성으로 ndarray 값을 알 수 있다.

```
print(type(indexes.values))
print(indexes.values.shape)
print(indexes[:5].values)
print(indexes.values[:5])
print(indexes[6])
'''
<class 'numpy.ndarray'>
```


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

- ndarray와 유사하게 단일 값 반환 및 슬라이싱도 가능
- 하지만 한 번 만들어진 DataFrame 및 Series의 Index 객체는 함부로 변경할 수 없다!

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

- Series 객체는 Index 객체를 포함하지만 Series 객체에 연산 함수를 적용할 때 Index는 연산에서 제외

```
titanic_reset_df = titanic_df.reset_index(inplace=False)
titanic_reset_df.head(3)
```

 `reset_index()` 메서드를 수행하면 새롭게 인덱스를 연속 숫자형으로 할당하며 기존 인덱스는 'index'라는 새로운 칼럼 명으로 추가

```
print('### before reset_index ###')
value_counts = titanic_df['Pclass'].value_counts()
print(value_counts)
print('value_counts 객체 변수 타입: ', type(value_counts))
new_value_counts = value_counts.reset_index(inplace=False)
print('### After reset_index ###')
print(new_value_counts)
print('new_value_counts 객체 변수 타입: ', type(new_value_counts))
'''
### before reset_index ###
Pclass
3    491
1    216
2    184
Name: count, dtype: int64
value_counts 객체 변수 타입: <class 'pandas.core.series.Series'>
### After reset_index ###
   Pclass  count
0       3    491
1       1    216
2       2    184
new_value_counts 객체 변수 타입: <class 'pandas.core.frame.DataFrame'>
'''
```

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

- 이때 Series가 아닌 DataFrame이 반환되는 것에 주의
- `reset_index()`의 파라미터 중 `drop=True`로 설정하면 기존 인덱스는 새로운 칼럼으로 추가되지 않고 삭제

데이터 선택 및 필터링

넘파이의 경우 `[]` 연산자 내 단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱을 통해 데이터를 추출

판다스의 경우 `ix[], iloc[], loc[]` 연산자를 통해 동일한 작업 수행

DataFrame의 [] 연산자

• 넘파이에서 `[]` 연산자는 행의 위치, 열의 위치, 슬라이싱 범위 등을 지정해 데이터를 가져올 수 있다.

• 하지만 DataFrame 바로 뒤에 있는 `[]` 안에 들어갈 수 있는 것은 칼럼 명 문자(또는 칼럼 명의 리스트 객체), 또는 인덱스로 변환 가능한 표현식

DataFrame 뒤에 있는 `[]`는 칼럼만 지정할 수 있는 '칼럼 지정 연산자'로 이해

```
print('단일 칼럼 데이터 추출:\n', titanic_df['Pclass'].head(3))
print('\n여러 칼럼의 데이터 추출:\n', titanic_df[['Survived', 'Pclass']].head(3))
print('[] 안에 숫자 index는 KeyError 오류 발생:\n', titanic_df[0])
'''
```

단일 칼럼 데이터 추출:

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

여러 칼럼의 데이터 추출:

```
Survived  Pclass
0         0      3
1         1      1
2         1      3
```

```
-----
KeyError                                Traceback (most recent call last)
...
KeyError: 0
'''
```

- DataFrame에 ['칼럼명']으로 칼럼명에 해당하는 칼럼 데이터의 일부만 추출
- 여러 개의 칼럼에서 데이터를 추출하려면 리스트 객체를 이용

```
titanic_df[0:2]
titanic_df[titanic_df['Pclass'] == 3].head(3)
```


- 숫자 값을 입력할 경우 오류가 발생하지만, 판다스의 인덱스 형태로 변환 가능한 표현식은 입력 가능
- 불린 인덱싱 표현도 가능, 원하는 데이터를 편리하게 추출해주므로 자주 사용된다.

DataFrame ix[] 연산자

• `ix[]`는 넘파이 ndarray의 `[]` 연산자와 유사한 기능을 DataFrame에 제공

- `ix[0, 'Pclass']`와 같이 행 위치 지정으로 인덱스값 0, 열 위치 지정으로 칼럼 명인 'Pclass'를 입력해 원하는 위치의 데이터 추출
 - 행 위치 지정은 DataFrame의 인덱스값
 - 열 위치 지정은 칼럼 명뿐만 아니라 `ix[0, 2]`와 같이 칼럼의 위치 값 지정도 가능
- DataFrame의 인덱스 값은 명칭 기반 인덱싱
- 명칭 기반과 위치 기반, 두 가지 방식의 혼돈으로 인해 사라졌다!


DataFrame.iloc[] 연산자

 `iloc[]` : 칼럼 위치 기반 인덱싱

0을 출발점으로 하는 가로축, 세로축 좌표 기반의 행과 열 위치를 기반으로 데이터 지정
따라서 행과 열 값으로 integer, integer 형의 슬라이싱, 팬시 리스트의 값을 입력
명확한 위치 기반 인덱싱이 사용되어야 하는 제약으로 인해 불린 인덱싱은 제공하지 않는다.

```
data_df.iloc[0, 0]
data_df_reset.iloc[0, 1]
```

DataFrame.loc[] 연산자

 `loc[]` : 칼럼 명칭 기반 인덱싱

행 위치에는 DataFrame index 값을, 열 위치에는 칼럼 명을 입력

```
data_df.loc['one', 'Name']
data_df_reset.loc[1, 'Name'] # 문자열이 아니어도 가능
```

```
print('위치 기반 iloc slicing\n', data_df.iloc[0:1, 0], '\n')
print('명칭 기반 loc slicing\n', data_df.loc['one':'two', 'Name'])

'''
위치 기반 iloc slicing
   one    Chulmin
Name: Name, dtype: object

명칭 기반 loc slicing
   one    Chulmin
   two    Eunkyoung
Name: Name, dtype: object
'''
```

- 슬라이싱 기호를 적용하면 종료 값-1이 아니라 종료 값까지 포함하는 것을 의미

```
print(data_df_reset.loc[1:2, 'Name'])
'''
1    Chulmin
2    Eunkyoung
Name: Name, dtype: object
'''
```

- 특히 DataFrame의 인덱스가 정수형일 때 주의

불린 인덱싱

- 매우 편리한 데이터 필터링 방식
- [], ix[], loc[]에서 공통으로 지원
- iloc[]는 정수 값이 아닌 불린 값에 대해서는 지원하지 않기 때문에 지원되지 않는다.

```
titanic_df = pd.read_csv('titanic_train.csv')
titanic_boolean = titanic_df[titanic_df['Age']>60]
print(type(titanic_boolean))
```

```
titanic_boolean
# <class 'pandas.core.frame.DataFrame'>
```

- 반환된 객체 타입은 DataFrame

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

- [] 내에 불린 인덱싱을 적용하면 반환되는 객체가 DataFrame이므로 원하는 칼럼 명만 별도로 추출 가능

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

- loc[]을 이용해도 동일하게 적용 가능
- ['Name', 'Age']는 칼럼 위치에 놓여야 한다.

```
titanic_df[(titanic_df['Age']>60) & (titanic_df['Pclass']==1) & (titanic_df['Sex']=='female')]
```

```
cond1 = titanic_df['Age'] > 60
cond2 = titanic_df['Pclass'] == 1
cond3 = titanic_df['Sex'] == 'female'
titanic_df[cond1&cond2&cond3]
```

- 여러 개의 복합 조건도 결합해 적용 가능

정렬, Aggregation 함수, GroupBy 적용

DataFrame, Series의 정렬 - sort_values()

📌 `sort_values()` : DataFrame과 Series를 정렬

- `by` 로 특정 칼럼을 입력하면 해당 칼럼으로 정렬을 수행
- `ascending = True` (디폴트)로 설정하면 오름차순, `ascending = False` 로 설정하면 내림차순으로 정렬
- `inplace = False` (디폴트)로 설정하면 `sort_values()`를 호출한 DataFrame은 그대로 유지하되 정렬된 DataFrame을 결과로 반환, `inplace = True` 로 설정하면 호출한 DataFrame의 정렬 결과를 그대로 적용

```
titanic_sorted = titanic_df.sort_values(by=['Name'])
titanic_sorted.head(3)
```

```
titanic_sorted = titanic_df.sort_values(by=['Pclass', 'Name'], ascending = False)
titanic_sorted.head(3)
```

Aggregation 함수 적용

📌 DataFrame에서 `min()`, `max()`, `sum()`, `count()` 와 같은 aggregation 함수의 적용은 RDBMS SQL의 aggregation 함수 적용과 유사

```
titanic_df.count()
```

- DataFrame에서 바로 aggregation을 호출할 경우 모든 칼럼에 해당 aggregation을 적용

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

- 특정 칼럼에 aggregation 함수를 적용하기 위해서는 대상 칼럼들만 추출

groupby 적용

📌 DataFrame의 `groupby()` 사용 시 입력 파라미터 `by`에 칼럼을 입력하면 대상 칼럼으로 `groupby()`

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

DataFrameGroupBy라는 또 다른 형태의 DataFrame을 반환

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

`groupby()`를 호출해 반환된 결과에 aggregation 함수를 호출하면 `groupby()` 대상 칼럼을 제외한 모든 칼럼에 해당 aggregation 함수 적용

```
titanic_groupby = titanic_df.groupby('Pclass')[['PassengerId', 'Survived']].count()
titanic_groupby
```

DataFrame의 `groupby()`에 특정 칼럼만 aggregation 함수를 적용하려면 `groupby`로 반환된 `DataFrameGroupBy` 객체에 해당 칼럼을 필터링한 뒤 aggregation 함수를 적용

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

서로 다른 aggregation 함수를 적용할 경우 여러 개의 함수명을 `DataFrameGroupBy` 객체의 `agg()` 내에 인자로 입력

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

DataFrame의 `groupby()`를 이용해 API 기반으로 처리하다보니 유연성이 떨어진다.

여러 개의 칼럼이 서로 다른 aggregation 함수를 `groupby`에서 호출하려면 복잡한 처리가 필요하다.

결손 데이터 처리하기

결손 데이터란 칼럼에 값이 없는, NULL인 경우를 의미, 이를 넘파이의 NaN으로 표시

머신러닝 알고리즘은 NaN 값을 처리하지 않으므로 이 값을 다른 값으로 대체해야 한다. 또한 NaN 값은 평균, 총합 등의 함수 연산 시 제외된다.

isna()로 결손 데이터 여부 확인

📌 `isna()` : NaN의 여부를 확인 (True나 False값 반환)

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

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

- 결손 데이터의 개수는 `isna()` 결과에 `sum()` 함수를 추가해 구할 수 있다.
- `sum()`을 호출 시 True는 내부적으로 숫자 1로, False는 숫자 0으로 변환된다.

fillna()로 결손 데이터 대체하기

📌 `fillna()` : NaN 값을 다른 값으로 대체

```
titanic_df['Cabin'] = titanic_df['Cabin'].fillna('C000')
titanic_df.head(3)
```

```
titanic_df['Age'] = titanic_df['Age'].fillna(titanic_df['Age'].mean())
titanic_df['Embarked'] = titanic_df['Embarked'].fillna('S')
```

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

- fillna()를 이용해 반환 값을 다시 받거나(위의 경우) inplace=True 파라미터를 fillna()에 추가해야 실제 데이터 세트 값이 변경되는 것에 주의

apply lambda 식으로 데이터 가공

apply 함수에 lambda 식을 결합해 DataFrame이나 Series의 레코드별로 데이터를 가공

칼럼에 일괄적으로 데이터 가공을 하는 것이 속도 면에서 더 빠르나 복잡한 데이터 가공이 필요할 경우 사용



lambda란, 함수의 선언과 함수 내의 처리를 한 줄의 식으로 변환하는 식

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

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

여러 개의 값을 입력 인자로 사용해야 할 경우 `map()` 함수를 결합

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

```
titanic_df['Child_Adult'] = titanic_df['Age'].apply(lambda x : 'Child' if x <= 15 else 'Adult')
titanic_df[['Age', 'Child_Adult']].head(8)
```

- if 절의 경우 if 식보다 반환 값을 먼저 기술

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

- else if를 이용하기 위해서는 else 절을 ()로 내포해 () 내에서 다시 if, else를 적용하여 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 지정
# get_category(x)는 입력값으로 'Age' 칼럼 값을 받아서 해당하는 cat 반환
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : get_category(x))
titanic_df[['Age', 'Age_cat']].head()
```

- else if가 많이 나와야 하는 경우나 switch case 문의 경우 별도의 함수를 만들어 사용