EURON 1주차 과제_1장

Numpy (Numerical Python)

- 빠른 배열 연산 속도
- 높은 호환성(C/C++)
- Pandas보다 편리성이 떨어진다는 한계

넘파이 ndarray 개요

```
import numpy as np
```

- as np 추가해 약어로 모듈 표현
- 기반 데이터 타입 ndarray → 다차원 배열을 쉽게 생성하고 다양한 연산 수행 가능

```
array1 = np.array([1, 2, 3])
print('array1 type:', type(array1))
print('array2 array 형태:', array1.shape)

array2 = np.array([[1,2,3,],
                   [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)
```

<Output>

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

- array1은 1차원 array, 3개의 데이터
- array2는 2차원 array, 2 row and 3 column
- array3는 2차원 array, 1 row and 3 column

→ array1과 array3는 같은 데이터값은 같으나 차원이 다름

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

<Output>

```
array1: 1차원, array2: 2차원, array3: 2차원
```

- array()의 함수의 인자로 파이썬의 리스트 객체 주로 사용
- 리스트 []는 1차원, 리스트의 리스트 [[]]는 2차원 으로 나타낼 수 있음

ndarray의 데이터 타입

- ndarray내의 데이터값은 숫자, 문자열, 불 값 모두 가능
- 숫자형의 경우 int형, unsigned int형, float형, complex형 모두 가능
- 그러나 한 ndarray 객체 내에서는 같은 타입의 데이터 타입만 가능 (ex. int와 float는 함께 있을 수 없음)
- 데이터 타입 dtype 속성으로 확인 가능

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

<Output>

```
<class 'list'>
<class 'numpy.ndarray'>
[1 2 3] int32
```

- list1은 리스트 자료형이며 숫자 1, 2, 3을 값으로 가짐
- ndarray로 쉽게 변경 가능, 데이터값은 int32형

만약 다른 데이터 유형이 섞여있는 리스트를 ndarray로 변경하면 데이터 크기가 더 큰 데이터 타입으로 형 변형을 일괄 적용

```
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)
```

<Output>

```
['1' '2' 'test'] <U11
[1. 2. 3.] float64
```

- list2에서 숫자형 값 1, 2가 문자열 값으로 변환
- list3에서 int 1, 2가 float64형으로 변환

ndarray 내 데이터값의 타입 변경은 astype() 메서드를 통해 가능. 메모리를 더 절약해야할 때 보통 이용

```
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_int.dtype)
```

<Output>

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

- int32형 데이터를 float64로 변환
- 다시 float64를 int32로 변경 (float를 int형으로 변경할 때 소수점 이하는 모두 없어짐)

ndarray를 편리하게 생성하기 - arrange, zeors, ones

arrange()

- range()와 유사한 기능
- array를 range()로 표현
- 0부터 함수 인자 값 -1까지의 값을 순차적으로 ndarray의 데이터값으로 변환

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

<Output>

```
[0 1 2 3 4 5 6 7 8 9]
int32 (10,)
```

- default 함수 인자는 stop 값
- 0부터 stop 값인 10에서 -1을 더한 9까지의 연속 숫자 값으로 구성된 1차원 ndarray를 만들어줌
- range와 유사하게 start 값도 부여해 0이 아닌 다른 값부터 시작한 연속 값을 부여할 수 있음

zeros()

- 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray를 반환함

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

<Output>

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

ones()

- 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 1로 채운 해당 shape를 가진 ndarray를 반환함
- 함수 인자로 dtype을 정해주지 않으면 default로 float64 형의 데이터로 ndarray를 채움

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

<Output>

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

ndarray의 차원과 크기를 변경하는 reshape()

- reshape() 메서드는 ndarray를 특정 차원 및 크기로 변환
- 변환을 원하는 크기를 함수 인자로 부여

```
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)
```

<Output>

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

- reshape()는 지정된 사이즈로 변경이 불가능하면 오류 발생

```
array1.reshape(4, 3)
```

<Output>

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 1
----> 1 array1.reshape(4, 3)

ValueError: cannot reshape array of size 10 into shape (4,3)
```

- 인자로 -1 적용 시 원래 ndarray와 호환되는 새로운 shape로 변환해줌

```
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)
```

<Output>

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

- array1은 1차원 ndarray로 0~9까지의 데이터, 총 10개의 데이터를 가짐
- 이를 5개의 column으로 나타내려면 row는 2개가 되어야 함 → array2의 shape
- 5개의 row로 나타내려면 column은 2개가 되어야 함 → array3의 shape

```
array1 = np.arange(10)
array4 = array1.reshape(-1,4)
```

<Output>

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 2
      1 array1 = np.arange(10)
----> 2 array4 = array1.reshape(-1,4)

ValueError: cannot reshape array of size 10 into shape (4)
```

- 그러나 이 경우, 10개의 데이터값은 4개의 column을 가진 row로 변경할 수 없기 때문에 에러가 발생함

주로 reshape() 에서 -1 인자는 reshape(-1, 1)와 같은 형태로 자주 사용

- 원본 ndarray가 어떤 형태라도 2차원이고, 여러 개의 row를 가지되 반드시 1개의 column을 가진 ndarray로 변환됨을 보장
- 여러개의 넘파이 ndarray를 stack이나 concat으로 결합할 때 각각의 ndarray의 형태를 통일하는데 사용될 수 있음

```
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())

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

<Output>

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

- reshape(-1,1)을 이용해 3차원을 2차원으로, 1차원을 2차원으로 변경
- ndarray는 tolist() 메서드를 이용해 리스트 자료형으로 변환 가능

넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

1. 특정한 데이터만 추출: 원하는 위치의 인덱스 값을 지정하면 해당 위치의 데이터가 반환됩니다.
2. 슬라이싱(Slicing): 슬라이싱은 연속된 인덱스상의 ndarray를 추출하는 방식입니다. ‘:’ 기호 사이에 시작 인덱스와 종료 인덱스를 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의 ndarray를 반환합니다. 예를 들어 1:5라고 하면 시작 인덱스 1과 종료 인덱스 4까지에 해당하는 ndarray를 반환합니다.
3. 팬시 인덱싱(Fancy Indexing): 일정한 인덱싱 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray를 반환합니다.
4. 불린 인덱싱(Boolean Indexing): 특정 조건에 해당하는지 여부인 Ture/False 값 인덱싱 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray를 반환합니다.

단일 값 추출

```
#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))
```

<Output>

```
array1: [1 2 3 4 5 6 7 8 9]
value: 3
<class 'numpy.int32'>
```

- 인덱스는 0부터 시작하므로 array1[2]는 3번째 인덱스 위치의 데이터값 → 3
- array1[2]의 타입은 ndarray 타입이 아닌 데이터값

인덱스에 마이너스 기호 이용 시 맨 뒤에서부터 데이터값 추출

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

<Output>

```
맨 뒤의 값 9 맨 뒤에서 두번째 값: 8
```

단일 인덱스 이용해 ndarray 내의 데이터값 수정 시

```
array1[0] = 9
array1[8] = 0
print('array1:', array1)
```

<Output>

```
array1: [9 2 3 4 5 6 7 8 0]
```

다차원 ndarray에서 단일 값 추출

- 1차원과의 차이로 2차원의 경우, 콤마(,)로 분리된 로우와 칼럼 위치의 인덱스를 통해 접근

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

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])
```

<Output>

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

- 실제 넘파이에서는 row와 column을 사용하지 않음
- row = axis 0, column = axis 1
- ex) [row=0, col=1] 은 다음이 옳은 표현임 [axis 0=0, axis 1=1]
- 3차원의 경우 axis 0, axis 1, axis 2로 구분

슬라이싱

- ':' 기호를 이용해 연속한 데이터 슬라이싱해 추출
- 단일 데이터값 추출의 데이터 타입 : 데이터 값 / 슬라이싱, 불린, 팬시 인덱싱의 데이터 타입 : ndarray
- ':' 사이에 시작 인덱스와 종료 인덱스 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의 ndarray 변환

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

<Output>

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

슬라이싱 기호인 ':' 사이의 시작, 종료 인덱스는 생략 가능

1. ':' 기호 앞에 시작 인덱스 생략시 맨 처음 인덱스인 0으로 간주
2. ':' 기호 뒤에 종료 인덱스 생략시 맨 마지막 인덱스로 간주
3. ':' 기호 앞/뒤에 시작/종료 인덱스 생략시 맨 처음/마지막 인덱스로 간주

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

array5 = array1[3:]
print(array5)

array6 = array1[:]
print(array6)
```

<Output>

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

2차원 ndarray의 경우 콤마(,)로 로우와 칼럼 인덱스를 지칭

```
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[:2, 1:]\n', array2d[:2, 1:])
print('array2d[:2, 0]\n', array2d[:2, 0])
```

<Output>

```
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]
```

팬시 인덱싱

팬시 인덱싱(Fancy Indexing)은 리스트나 ndarray로 인덱스 집합을 지정하면 해당 위치의 인덱스에 해당하는 ndarray를 반환하는 인덱싱 방식

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

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

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

array5 = array2d[[0, 1]]
print('array2d[[0,1]]=>', array5.tolist())
```

<Output>

```
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

- 불린 인덱싱(Boolean indexing)은 조건 필터링과 검색을 동시에 할 수 있기 때문에 매우 자주 사용되는 인덱싱 방식
- 불린 인덱싱 이용 시 for loops/if else 문보다 훨씬 간단하게 이를 구현할 수 있음

```
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)
```

<Output>

```
array1d > 5 불린 인덱싱 결과 값 : [6 7 8 9]
```

위와 동일한 불린 ndarray를 만들고 이를 array1d[]내에 인덱스로 입력하면 동일한 데이터 세트가 반환됨

```
boolean_indexs = np.array([False, False, False, False, False, True, True, True, True ])
array3 = array1d[boolean_indexs]
print('불린 인덱스로 필터링 결과 :', array3)
```

<Output>

```
불린 인덱스로 필터링 결과 : [6 7 8 9]
```

다음과 같이 직접 인덱스 집합을 만들어 대입한 것과 동일

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

<Output>

```
일반 인덱스로 필터링 결과 : [6 7 8 9]
```

행렬의 정렬 - sort()와 argsort()

행렬 정렬

- np.sort() : 넘파이에서 sort()를 호출하는 방식 → 원행렬은 유지한 채 원 행렬의 정렬된 행렬 반환
- ndarray.sort() : 행렬 자체에서 sort()를 호출 → 행렬 자체를 정렬한 형태로 반환, 반환값은 None

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

<Output>

```
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
org_array.sort( ) 호출 후 원본 행렬: [1 3 5 9]
```

기본적으로 오름차순으로 행렬 내 원소 정렬

내림차순의 경우 [::-1] 적용

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

<Output>

```
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)
```

<Output>

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

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

정렬된 행렬의 인덱스를 반환하기

- np.argsort()는 정렬 행렬의 원본 행렬 인덱스를 ndarray 형으로 반환

```
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)
```

<Output>

```
<class 'numpy.ndarray'>
행렬 정렬 시 원본 행렬의 인덱스: [1 0 3 2]
```

내림차순은 마찬가지로 np.argsort()[::-1] 적용

```
org_array = np.array([ 3, 1, 9, 5])
sort_indices_desc = np.argsort(org_array)[::-1]
print('행렬 내림차순 정렬 시 원본 행렬의 인덱스:', sort_indices_desc)
```

<Output>

```
행렬 내림차순 정렬 시 원본 행렬의 인덱스: [2 3 0 1]
```

예제

```
import numpy as np

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

sort_indices_asc = np.argsort(score_array)
print('성적 오름차순 정렬 시 score_array의 인덱스:', sort_indices_asc)
print('성적 오름차순으로 name_array의 이름 출력:', name_array[sort_indices_asc])
```

<Output>

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

선형대수 연산 - 행렬 내적과 전치 행렬 구하기

행렬 내적(행렬 곱)

- np.dot()을 이용해 계산 가능
- 왼쪽 행렬의 열 개수와 오른족 행렬의 행 개수가 동일해야 내적 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)
```

<Output>

```
행렬 내적 결과:
 [[ 58  64]
 [139 154]]
```

전치 행렬

- 원 행렬에서 행과 열 위치를 교환한 원소로 구성한 행렬
- 행렬 A의 전치 행렬은 A^T 와 같이 표시

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

<Output>

```
A의 전치 행렬:
 [[1 3]
 [2 4]]
```

판다스

- 파이썬에서 데이터 처리를 위해 존재하는 가장 인기 있는 라이브러리
- 2차원 데이터를 효율적으로 가공/처리할 수 있는 다양하고 훌륭한 기능 제공
- DataFrame이 핵심 객체
- DataFrame : 여러 개의 행과 열로 이뤄진 2차원 데이터를 담는 데이터 구조체

- Index : RDBMS의 PK처럼 개별 데이터를 고유하게 식별하는 key 값
- Series : Index를 key 값으로 가짐.
- Series는 칼럼이 하나뿐인 데이터 구조체인데 반해 DataFrame은 칼럼이 여러 개인 구조체

  →DataFrame은 여러 개의 Series로 이뤄져 있음

판다스 시작 - 파일은 DataFrame으로 로딩, 기본 API

```
import pands as pd
```

판다스는 다양한 포맷으로 된 파일을 DataFrame으로 로딩할 수 있는 편리한 API를 제공

대표적으로 read_csv(), read_table(), read_fwf가 있음

▼ read_csv() : CSV(칼럼을 ‘,’로 구분한 파일 포맷) 파일 포맷 변환을 위한 API

- read_csv()는 CSV뿐만 아니라 어떤 필드 구분 문자 기반의 파일 포맷도 DataFrame으로 변환 가능
- 인자인 sep에 해당 구분 문자를 입력하면 됨

ex) read_csv(’파일명’, sep=’\t’), 생략시 콤마로 할당(sep=’,’)

- read_csv(filepath_or_buffer, sep=’,’,…)함수에서 가장 중요한 인자는 filepath, 나머지 인자는 지정하지 않으면 디폴트 값으로 할당
- filepath에는 로드하려는 데이터 파일의 경로를 포함한 파일명 입력

- read_table() : 필드 구분 문자가 탭(’\t’)
- read_fwf() : Fixed Width, 고정 길이 기반의 칼럼 포맷을 DataFrame으로 로딩하기 위한 API

객체의 shape 변수를 이용하여 행과 열의 크기를 알아볼 수 있음(각각 튜플 형태로 반환)

DataFrame은 데이터뿐만 아니라 칼럼의 타입, Null데이터 개수, 데이터 분포도 등의 메타 데이터 등도 조회 가능

- info() : 총 데이터 건수와 데이터 타입, Null 건수를 알 수 있음
- describe() : 칼럼별 숫자형 데이터값의 n-percentile 분포도, 평균값, 최댓값, 최솟값 나타냄
- describe() : 오직 숫자형(int, float 등) 칼럼의 분포도만 조사. 자동으로 object 타입의 칼럼은 출력에서 제외

데이터의 분포도를 아는 것은 머신러닝 알고리즘의 성능을 향상시키는 중요한 요소

describe() 메서드를 통해 개략적인 수준의 분포도 확인 가능

- count : Not Null인 데이터 건수
- mean : 전체 데이터의 평균값

- std : 표준편차
- min/max : 최솟값/최댓값

describe() 메서드를 통해 해당 숫자 칼럼이 숫자형 카테고리 칼럼인지 판단 가능

- 카테고리 칼럼 : 특정 범주에 속하는 값을 코드화한 칼럼

DataFrame['칼럼명'] → Series 형태로 특정 칼럼 데이터 세트가 반환

value_counts() → 해당 칼럼값의 유형과 건수 확인 가능

- Null 값을 포함하여 개별 데이터 값의 건수를 계산할지를 dropna 인자로 판단
- dropna의 기본값은 True, Null 값을 무시하고 개별 데이터 값의 건수 계산
- Null 값 포함 원하면 dropna의 인자값을 False로 입력

DataFrame과 리스트, 딕셔너리, 넘파이 ndarray 상호 변환

DataFrame은 파이썬 리스트, 딕셔너리, 넘파이 ndarray 등으로 상호 변환 가능

사이킷런에서 DataFrame과 넘파이 ndarray 상호 간의 변환 빈번하게 발생

넘파이 ndarray, 리스트, 딕셔너리를 DateFrame으로 변환하기

- DateFrame은 칼럼명을 가짐(일반적으로 DataFrame으로 변환 시 칼럼명 지정, 안하면 자동으로 할당)
- 생성 인자 data는 리스트나 딕셔너리 또는 넘파이 ndarray 입력
- 생성 인자 columns는 칼럼명 리스트 입력
- DataFrame은 행과 열을 가지는 2차원 데이터이기 때문에 2차원 이하의 데이터들만 DataFrame으로 변환 가능

DataFrame을 넘파이 ndarray, 리스트, 딕셔너리로 변환하기

- 많은 머신러닝 패키지가 기본 데이터 형으로 넘파이를 사용
- 머신러닝 패키지의 입력 인자 등을 적용하기 위해 다시 넘파이 ndarray로 변환하는 경우가 빈번
- DataFrame을 넘파이 ndarray로 변환하는 것은 DataFrame 객체의 values를 이용해 가능
- DataFrame을 리스트로 변환하는 것은 values로 얻은 ndarray에 tolist()를 호출
- DataFrame을 딕셔너리로 변환하는 것은 DataFrame 객체의 to_dict() 메서드 호출, 인자로 'list'입력 시 딕셔너리 값이 리스트형으로 반환

DataFrame의 칼럼 데이터 세트 생성과 수정

DataFrame의 칼럼 데이터 세트 생성 과 수정 역시 [] 연산자를 이용해 가능

DataFrame 데이터 삭제

DataFrame에서 데이터 삭제는 drop() 메서드 이용

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

- axis 값에 따라서 특정 칼럼 또는 특정 행을 드롭 (axis 0은 로우 방향 축, axis 1은 칼럼 방향 축)
- 대부분의 경우 칼럼을 드롭하기 때문에 axis=1을 씀
- 이상치 데이터를 삭제하는 경우 axis=0을 씀

Inplace=False

- 자신의 데이터프레임 데이터는 삭제하지 않으며, 삭제된 결과 데이터프레임을 반환

Inplace=True

- 자신의 DataFrame의 데이터 삭제
- 반환 값이 None이 됨 → 반환 값을 다시 자신의 DataFrame 객체로 할당하면 안 됨

Index 객체

- DataFrame, Series의 레코드를 고유하게 식별하는 객체
- DataFrame.index, Series.index 속성을 통해 객체 추출 가능
- 한 번 만들어진 DataFrame 및 Series의 Index 객체는 변경 불가
- Series 객체에 연산 함수를 적용할 때 Index를 연산에서 제외, 오직 식별용

reset_index()

- 인덱스를 연속 int 숫자형 데이터로 만들때 사용
- Series에 적용 시 DataFrame 반환
- parameter 중 drop=True로 설정 시 기존 인덱스 추가되지 않고 삭제됨 → Series로 유지

데이터 셀렉션 및 필터링

DataFrame의 [] 연산자

- [] : 칼럼 지정 연산자, 연산자 안에 칼럼명 문자 또는 인덱스로 변환 가능한 표현식 들어갈 수 있음
- 불린 인덱싱 표현도 가능
- DataFrame 바로 뒤의 [] 연산자는 넘파이의 []나 Series의 []과 다름
- 슬라이싱 연산으로 데이터 추출하는 방법은 사용하지 않는 게 좋음

DataFrame iloc[] 연산자

- iloc[] : 위치 기반 인덱싱 방식으로 동작
- 행과 열 위치를 세로축, 가로축 좌표 정숫값으로 지정
- 좌표 위치에 정숫값 또는 정수형의 슬라이싱, 팬시 리스트 값 입력
- 좌표 위치에 인덱스 값이나 칼럼명 입력 시 오류 발생
- 열 위치에 -1 입력 시 마지막 열 데이터 가져올 수 있음
- 불린 인덱싱 제공X

DataFrame. loc[] 연산자

- loc[] : 명칭기반 인덱싱 방식으로 동작
- 인덱스 값으로 행 위치, 칼럼 명칭으로 열 위치 지정
- 슬라이싱 기호 ':' 적용 시 종료값 포함

불린 인덱싱

- 매우 편리한 데이터 필터링 방식 : 처음부터 가져올 값을 조건으로 []내에 입력하면 자동으로 원하는 값 필터링
- [], loc[]에서 공통으로 지원, iloc[]는 정수형 값이 아닌 불린 값에 대해서는 지원하지 않아 불린 인덱싱 지원되지 않음

정렬, Aggregation 함수, GroupBy 적용

DataFrame, Series의 정렬 - sort_values()

- sort_values : RDBMS SQL의 order by 키워드와 유사
- 주요 입력 파라미터 : by, ascending, inplace
- by : 특정 칼럼 입력 시 해당 칼럽으로 정렬 수행
- ascending =True로 설정 시 오름차순 정렬, =False로 설정 시 내림차순 정렬
- inplace : =False로 설정 시 sort_values() 호출한 DataFrame은 유지하고 정렬된 DataFrame 결과로 반환, =True로 설정 시 호출한 정렬 결과 그대로 적용

Aggregation 함수 적용

- min(), max(), sum(), count() 등
- RDBMS SQL의 aggregation 함수 적용과 유사 → 모든 칼럼에 aggregation 적용한다는 차이
- count()는 Null 값을 반영하지 않은 결과를 반환

groupby() 적용

- 입력 파라미터 by에 칼럼 입력 시 대상 칼럼으로 groupby됨
- DataFrame에 groupby() 호출 시 DataFrameGroupBy 라는 또 다른 형태의 DataFrame 반환
- groupby()를 호출해 반환된 결과에 aggregation 함수를 호출하면 groupby() 대상 칼럼을 제외한 모든 칼럼에 해당 aggregation 함수 적용
- 서로 다른 aggregation 함수를 적용할 경우 적용하려는 여러 개의 aggregation 함수명을 DataFrameGroupBy 객체의 agg() 내에 인자로 입력해 사용

결손 데이터 처리

- 결손 데이터는 칼럼에 값이 없는, 즉 NULL인 경우를 의미, 넘파이의 NaN으로 표시
- 머신러닝 알고리즘은 NaN 값을 처리하지 않음 , 다른 값으로 대체해야 함
- NaN 값은 평균, 총합 등의 함수 연산 시 제외

insa()로 결손 데이터 여부 확인

- DataFrame에 insa() 수행 시 모든 칼럼의 값이 NaN인지 아닌지를 True나 False로 알려줌
- 결손 데이터의 개수는 insa() 결과에 sum() 함수를 추가해 구할 수 있음

fillna()로 결손 데이터 대체하기

- filln()를 이용해 반환 값을 다시 받거나 inplace=True 파라미터를 fillna()에 추가해야 실제 데이터 세트 값이 변경 됨

apply lambda 식으로 데이터 가공

- apply 함수에 lambda 식을 결합해 DataFrame이나 Series의 레코드별로 데이터를 가공
- 일괄적으로 데이터 가공하는 것이 속도 면에서 더 빠르나 복잡한 데이터 가공이 필요한 경우 이용
- lambda x : x ** 2에서 ‘:’로 입력 인자와 반환될 입력 인자의 계산식을 분리
- ‘:’ 왼쪽에 있는 x는 입력 인자, 오른쪽은 입력 인자의 계산식 → 오른쪽 계산식은 반환 값을 의미
- 여러 개의 입력 인자로 사용해야 할 경우, map() 합수를 결합해 사용
- if else 지원 : if 절의 경우 if 식보다 반환 값을 먼저 기술해야함