

벡터 & 행렬 & 배열

벡터, 행렬, 배열

➤ 벡터 만들기

- numpy로 1차원 배열 만듭니다.
- numpy 배열은 ndarray 클래스의 객체입니다.
- numpy.ndarray()
- numpy.asarray() #배열로 생성

라이브러리를 임포트합니다.

```
import numpy as np
```

행이 하나인 벡터를 만듭니다.

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

열이 하나인 벡터를 만듭니다.

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

벡터, 행렬, 배열

➤ 행렬 만들기

- numpy의 2차원 배열을 사용해 행렬(matrix)를 만듭니다.
- numpy.empty()는 초기값 대신 크기만 지정하여 임의의 값이 채워진 배열을 만듭니다.
- numpy.zeros()는 0으로 채운 배열을 만들고, ones()는 1로 채운 배열을 만듭니다.
- numpy.full(배열 크기, 채울 값) 는 특정 값으로 채운 배열을 만듭니다.

```
import numpy as np

matrix = np.array([[1, 2],
                  [1, 2],
                  [1, 2]])
empty_matrix = np.empty((3, 2))
zero_matrix = np.zeros((3, 2))
one_matrix = np.ones((3, 2))
seven_matrix = np.zeros((3, 2)) + 7
seven_matrix = np.full((3, 2), 7)
```

벡터, 행렬, 배열

➤ 희소 행렬

- 데이터에 0이 아닌 값이 매우 적을 때 이를 효율적으로 표현
- 0이 아닌 원소만 저장
- 다른 모든 원소는 0이라 가정하므로 계산 비용이 크게 절감
- CSR(Compressed Sparse Row) 행렬

라이브러리를 임포트합니다.

```
import numpy as np
from scipy import sparse
```

행렬을 만듭니다.

```
matrix = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])
```

CSR 행렬을 만듭니다.

```
matrix_sparse = sparse.csr_matrix(matrix)
```

희소 행렬을 출력합니다.

```
print(matrix_sparse)
```

```
(1, 1) 1
(2, 0) 3
```

벡터, 행렬, 배열

➤ 희소 행렬

```
# (data, (row_index, col_index))로 구성된 튜플을 전달합니다.  
# shape 매개변수에서 0을 포함한 행렬의 전체 크기를 지정합니다.  
matrix_sparse_2 = sparse.csr_matrix(([1, 3], ([1, 2], [1, 0])), shape=(3, 10))  
  
print(matrix_sparse_2)
```

```
(1, 1) 1  
(2, 0) 3
```

```
print(matrix_sparse_2.toarray())
```

```
[[0 0 0 0 0 0 0 0 0 0]  
 [0 1 0 0 0 0 0 0 0 0]  
 [3 0 0 0 0 0 0 0 0 0]]
```

```
matrix_sparse_2.todense()
```

```
matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
        [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int64)
```

벡터, 행렬, 배열

➤ 원소 선택

- numpy 배열의 인덱스는 0부터 시작합니다.
- numpy 배열에서 원소나 원소 그룹을 선택하는 방법으로 인덱싱과 슬라이싱을 제공합니다.
- 불리언 마스크 배열을 만들어 원소를 선택할 수도 있습니다.
- 행과 열의 인덱스 리스트를 전달하여 배열의 원소를 선택할 수 있습니다. - 팬시 인덱싱(fancy indexing)
- 불리언 마스크 배열을 만들어 원소를 선택할 수도 있습니다.

```
import numpy as np
vector = np.array([1, 2, 3, 4, 5, 6])
matrix = np.array([ [ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9] ])
vector[2]
matrix[1, 1]
vector[ :] #벡터에 있는 모든 원소를 선택합니다.
vector[ : 3] #세 번째 원소를 포함하여 모든 원소를 선택합니다.
vector[3: ] #세 번째 이후의 모든 원소를 선택합니다.
vector[-1]
matrix[:2, :]
matrix[:, 1:2]
matrix[[0, 2]]
matrix[[0,2], [1,0]]
mask = matrix > 5
matrix[mask]
```

벡터, 행렬, 배열

➤ 행렬 정보 확인

- shape - 행렬의 크기 확인
- size - 행렬의 원소 개수 확인(행*열)
- ndim -차원 수 확인
- dtype - 원소의 데이터 타입이나 바이트 크기를 확인

```
import numpy as np
matrix = np.array( [ [ 1, 2, 3, 4], [ 5, 6, 7, 8], [9, 10, 11, 12 ] ] )

matrix.shape
matrix.size
matrix.ndim
matrix.dtype #원소의 데이터 타입 확인
matrix.itemsize #원소 하나가 차지하는 바이트 크기
matrix.nbytes # 배열 전체가 차지하는 바이트 크기
```

벡터, 행렬, 배열

➤ 벡터화 연산

- numpy의 vectorize 클래스는 배열의 일부나 전체에 적용하도록 함수를 변환시킵니다.
- 넘파이 배열은 차원이 달라도 배열 간의 연산을 수행할 수 있습니다. (브로드캐스팅)
- 브로드캐스팅은 배열에 차원을 추가하거나 반복해서 배열 크기를 맞춥니다.

```
import numpy as np
```

```
matrix = np.array( [ [ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9] ])
```

```
add_100 = lambda i: i+100
```

```
vectorized_add_100 = np.vectorize(add_100)
```

```
vectorized_add_100(matrix)
```

```
matrix + 100
```

```
matrix + [100, 100, 100]
```

```
matrix + [[100], [100], [100]]
```

```
np.max(matrix)
```

```
np.min(matrix)
```

```
#axis 매개변수를 사용하여 특정 축을 따라 연산을 수행할 수 있습니다.
```

```
np.max(matrix, axis = 0) #각 열에서 최대값을 찾습니다.
```

```
np.min(matrix, axis = 1) #각 행에서 최대값을 찾습니다.
```


벡터, 행렬, 배열

➤ 벡터화 연산

- keepdims 매개변수를 True로 지정하면 원본 배열의 차원과 동일한 결과를 만듭니다.

```
vector_column = np.max(matrix, axis=1, keepdims=True)
vector_column
np.mean(matrix) #평균을 반환
np.var(matrix) #분산을 반환
np.std(matrix) #표준편차를 반환
```

벡터, 행렬, 배열

➤ 배열 크기 바꾸기

- 데이터를 동일하게 유지하면서 배열 크기(행과 열의 수, 구조) 변경
- 새로 생성된 행렬은 원래 행렬과 원소 개수가 같음
- reshape에 사용할 수 있는 매개변수 -1은 가능한 많이라는 뜻으로 유용하게 사용됨
- reshape(1, -1)은 행 하나에 열은 가능한 많게 라는 의미
- reshape에 정수 하나를 입력하면 그 길이의 1차원 배열을 반환
- reshape에 -1을 입력하면 1차원 배열을 반환

```
import numpy as np

matrix = np.array( [ [ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9] , [10, 11, 12]])
matrix.size
matrix.reshape(2, 6)
matrix.size
matrix.reshape(1, -1)
matrix.reshape(12)
matrix.reshape(-1)
matrix.ravel()
```

벡터, 행렬, 배열

➤ 벡터나 행렬 전치

- 전치는 선형대수학에서 자주 사용하는 연산으로 각 원소의 행과 열의 인덱스를 바꿉니다.
- `transpose()`는 튜플로 바꿀 차원을 직접 지정할 수 있다

```
import numpy as np

matrix = np.array( [ [ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9] ] )
matrix.T
np.array([1, 2, 3, 4, 5, 6] ).T  #벡터는 값의 모음이기 때문에 전치 할 수 없음
np.array([ [ 1, 2, 3, 4, 5, 6] ] ).T  #행 벡터를 열 벡터로 전치
matrix.transpose()
np.array([ [ [1, 2] ,[ 3, 4 ], [5, 6] ],
           [ [7, 8 ], [9, 10 ], [11, 12] ] ] ) #2x3x2 행렬
matrix.transpose((0 2, 1)) #2x2x3 행렬로 변환
```

벡터, 행렬, 배열

➤ 행렬 펼치기

- flatten() - 행렬을 1차원 배열로 펼치기
- reshape()는 numpy 배열의 뷰를 반환하지만 flatten()는 새로운 배열을 만듭니다.

```
import numpy as np

matrix = np.array( [ [ 1, 2, 3], [ 4, 5, 6], [ 7, 8, 9] ] )
matrix.flatten()
matrix.reshape(1, -1)
vector_resaped = matrix.reshape( -1)
vector_flattened = matrix.flatten()
matrix[0][0] = -1
vector_resaped
vector_flattened
```

벡터, 행렬, 배열

➤ 행렬의 랭크

- 행렬의 랭크는 행이나 열이 만든 벡터 공간의 차원
- numpy의 선형대수 메서드 matrix_rank를 사용
- 행렬의 랭크(계수)는 선형 독립적인 행 또는 열 개수
- matrix_rank()는 특잇값 분해 방식으로 랭크를 계산

```
import numpy as np
```

```
matrix = np.array( [ [ 1, 1, 1], [ 1, 1, 10], [ 1, 1, 15] ] )
```

```
np.linalg.matrix_rank(matrix) #행렬은 첫번째 열과 두번째 열이 동일하기 때문에 독립적인 열이 2개
```

```
s = np.linalg.svd(matrix, compute_uv = False) #svd 함수로 특잇값만 계산
```

```
np.sum(s > 1e-10) #오차를 고려하여 0에 가까운 아주 작은 값을 지정
```

벡터, 행렬, 배열

➤ 행렬식 계산

- 행렬식은 정방행렬에 의한 선형 변환이 특징을 나타내는 스칼라값입니다.
- (2,2) 크기의 행렬 $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ 가 있다면 행렬식은 $\det(A) = ad - bc$ 와 같이 계산
- 더 큰 행렬은 라플라스 전개를 사용해 재귀적으로 작은 행렬식으로 표현

```
import numpy as np
```

```
matrix = np.array( [ [ 1, 2, 3], [ 2, 4, 6], [ 3, 8, 9] ] )  
np.linalg.det(matrix) #행렬의 행렬식 반환
```

벡터, 행렬, 배열

➤ 행렬의 대각원소 추출

- numpy의 diagonal()를 사용하여 대각원소를 얻을 수 있습니다.
- offset매개변수를 사용하여 주 대각선에서 벗어난 대각원소를 얻을 수 있습니다
- diagonal()는 원본 배열의 뷰를 반환합니다.
- diag()도 대각원소를 추출

```
import numpy as np

matrix = np.array( [[ 1, 2, 3], [ 2, 4, 6], [ 3, 8, 9] ])
matrix.diagonal()
matrix.diagonal(offset=1) #주 대각선 하나 위의 대각원소를 반환
matrix.diagonal(offset=-1) #주 대각선 하나 아래의 대각원소를 반환
a = matrix.diagonal().copy() #반환된 배열을 변경하려면 복사
a = np.diag(matrix)
```

벡터, 행렬, 배열

➤ 행렬의 대각합 계산

- 행렬의 대각합은 대각원소의 합으로 머신러닝 알고리즘 내부에서 종종 사용됩니다.
- 넘파이 다차원 배열의 대각합은 `trace()`를 사용하여 간단하게 계산
- `offset` 매개변수를 지원

```
import numpy as np

matrix = np.array( [[ 1, 2, 3], [ 2, 4, 6], [ 3, 8, 9] ])
matrix.trace()
sum(matrix.diagonal())
matrix.trace(offset=1)
matrix.trace(offset=-1)
```


벡터, 행렬, 배열

➤ 고유값과 고유벡터 찾기

- 행렬 A로 표현되는 선형 변환을 적용할 때 고유벡터는 스케일(scale)만 바뀌는 벡터
- 넘파이 선형대수 모듈에서 eig 함수는 정방행렬의 고유값과 고유벡터를 계산
- 대칭행렬일 경우 linalg.eigh 함수를 고유값과 고유벡터를 더 빠르게 계산

```
import numpy as np
```

```
matrix = np.array( [ [ 1, -1, 3], [ 1, 1, 6], [ 3, 8, 9] ])
```

```
eigenvalues, eigenvectors = np.linalg.eig(matrix)
```

```
eigenvalues
```

```
#대칭행렬
```

```
matrix = np.array([ [ 1, -1, 3], [-1, 1, 6], [3, 6, 9]])
```

```
eigenvalues, eigenvectors = np.linalg.eig(matrix)
```

벡터, 행렬, 배열

➤ 행렬 점곱 계산

- 넘파이 dot함수를 사용하여 점곱을 계산
- np.matmul 함수는 np.dot함수와 달리 넘파이 스칼라 배열에 적용되지 않습니다.

```
import numpy as np

vector_a = np.array( [ 1, 2, 3] )
vector_b = np.array( [ 4, 5, 6 ] )

np.dot(vector_a, vector_b)
vector_a @ vector_b

scalar_a = np.array(1)
scalar_b = np.array(2)
np.dot(scalar_a, scalar_b)

scakar_a @ scalar_b  #ValueError)
```

벡터, 행렬, 배열

➤ 행렬의 덧셈, 뺄셈, 곱셈 계산

- dot(), @ : 두 행렬의 곱
- * 연산자 : 원소별 곱셈을 수행
- np.dot()는 다차원 배열에도 적용 가능 (첫 번째 배열의 마지막 차원과 두 번째 배열의 끝에서 두 번째 차원이 동일해야 합니다.)

```
import numpy as np
```

```
matrix_a = np.array( [ [ 1, 1, 1], [ 1, 1, 1], [ 1, 1, 2] ] )  
matrix_b = np.array( [ [ 1, 3, 1], [ 1, 3, 1], [ 1, 3, 8] ] )  
np.add(matrix_a , matrix_b)  
np.subtract(matrix_a , matrix_b)  
matrix_a + matrix_b
```

```
matrix_a = np.array( [ [ 1, 1 ], [ 1, 2 ] ] )  
matrix_b = np.array( [ [ 1, 3], [ 1, 2 ] ] )  
np.dot(matrix_a, matrix_b)  
matrix_a @ matrix_b
```

벡터, 행렬, 배열

➤ 역행렬

- `pinv()`는 정방행렬이 아닌 행렬의 역행렬을 계산

```
import numpy as np

matrix = np.array( [ [ 1, 4 ], [ 2, 5 ] ])
np.linalg.inv(matrix)
matrix @ np.linalg.inv(matrix)
```

벡터, 행렬, 배열

➤ 난수 생성

- random 모듈 사용

```
import numpy as np
np.random.seed(0)
np.random.random(3) #세개의 실수 난수 생성
np.random.randint(0, 11, 3) #1~10사이의 세 개이 정수 난수 생성
np.random.normal(0.0, 1.0, 3) # 평균이 0.0이고 표준편차가 1.0인 정규분포에서 세 개의 난수 생성
np.random.logistic(0.0, 1.0, 3) # 평균이 0.0이고 스케일이 1.0인 로지스틱 분포에서 세 개의 난수 생성
np.random.uniform(1.0, 2.0, 3) # 1.0보다 크거나 같고 2.0보다 작은 세 개의 난수 생성
```

DataSet Load

데이터셋 로드

➤ 샘플 데이터 셋 로드

- sklearn.datasets 모듈 아래에 있는 함수들은 파이썬 딕셔너리와 유사한 Bunch 클래스 객체를 반환
- `load_boston`
- `load_iris`
- `load_digits`

```
from sklearn import datasets
#숫자 데이터셋을 적재
digits = datasets.load_digits()

features = digits.data # 특성 행렬을 만듭니다.
target = digits.target #타깃 벡터를 만듭니다.
features[0]
digits.keys()
digits['DESCR'][:70]

import numpy as np
#매개변수 return_X_y를 True로 설정하면 특성 X와 타깃 y 배열을 반환
#load_digits 함수는 필요한 숫자 개수를 지정할 수 있는 n_class 매개변수를 제공
x, y = datasets.load_digits(n_class=5, return_x_y=True)
np.unique(y)
```

데이터셋 로드

➤ 모의 데이터 생성

- `make_regression()` - 선형 회귀에 사용할 데이터 셋 생성 (선형으로 분산된 데이터를 생성, 가우스 노이즈의 표준 편차, 원하는 피처의 수 지정 가능)
- `make_classification()` - 분류에 필요한 모의 데이터 셋 생성
- `make_blobs()` - 군집 알고리즘에 적용할 데이터셋 생성 (n개의 무작위 데이터 클러스터를 생성)
- `make_circles()` - 두개의 차원에 작은 원을 포함하는 큰 원이 포함된 임의의 데이터셋을 생성, SVM(Support Vector Machines)과 같은 알고리즘을 사용하여 분류를 수행할 때 유용

```
from sklearn.datasets import make_regression
#특성 행렬, 타깃 벡터, 정답계수를 생성
features, target, coefficients = make_regression(n_samples = 100,
n_features = 3, n_informative = 3, n_targets = 1, noise=0.0, coef=True, random_state = 1)
print('특성 행렬\n', features[:3])
print('타깃 벡터\n', target[:3])
features, target = make_classification(n_samples = 100,
n_features = 3, n_informative = 3, n_redundant = 1, n_classes = 2, weight=[.25, .75], random_state = 1)
print('특성 행렬\n', features[:3])
print('타깃 벡터\n', target[:3])
features, target = make_blobs(n_samples = 100,
n_features = 2, centers =3, cluster_std = 0.5, shuffle=True, random_state = 1)
print('특성 행렬\n', features[:3])
print('타깃 벡터\n', target[:3])
```


데이터셋 로드

➤ 모의 데이터 생성

- make_regression은 실수 특성 행렬과 실수 타깃 벡터를 반환
- make_classification과 make_blobs는 실수 특성 행렬과 클래스의 소속을 나타내는 정수 타깃 벡터를 반환
- make_regression과 make_classification의 n_informative 는 타깃 벡터를 생성하는 데 사용할 특성 수를 결정
- n_features 는 전체 특성 수
- make_classification의 weight 매개변수를 사용해 불균형한 클래스를 가진 모의 데이터셋 생성
- make_blobs의 centers 매개변수는 생성될 클러스터의 수를 결정

```
from sklearn.datasets import make_regression

features, target = make_blobs(n_samples = 100, n_features = 2, centers = 3, cluster_std = 0.5,
                             shuffle=True, random_state = 1)
print('특성 행렬\n', features[:3])
print('타깃 벡터\n', target[:3])

import matplotlib.pyplot as plt
plt.scatter(features[:, 0], features[:, 1], c=target)
plt.show()
```

데이터셋 로드

➤ CSV 데이터 로드

- read_csv : 로컬 혹은 원격 CSV파일 적재 (seq, header, skiprows, nrows, ...)

```
import pandas as pd

url = 'https://tinyurl.com/simulated-data'
dataframe = pd.read_csv(url)
dataframe.head(2)

dataframe = pd.read_csv(url, skiprows=range(1, 11), nrows=1)
dataframe
```

데이터셋 로드

➤ Excel 데이터 로드

- read_excel : 엑셀 스프레드시트를 적재 (na_filter, skip_nrows, keep-default_na, na_values 등 매개변수 지원)
- sheet_name 매개변수는 시트 이름 문자열이나 시트의 위치를 나타내는 정수(0부터 시작되는 인덱스)를 모두 받을 수 있습니다.
- read_excel 함수를 사용하려면 xlrd 패키지를 설치해야 합니다.

```
import pandas as pd

url = 'https://tinyurl.com/simulated-excel'
dataframe = pd.read_excel(url, sheet_name=0, header=1)
dataframe.head(2)
```

데이터셋 로드

➤ JSON 데이터 로드

- orient 매개변수 - JSON 파일이 어떻게 구성되었는지 지정
- json_normalize() : 구조화가 덜 된 JSON 데이터를 판다스 데이터프레임으로 변환하는 도구
- split, records, index, columns, values 매개변수.

```
import pandas as pd

url = "https://tinyurl.com/simulated-json"
dataframe = pd.read_json(url, orient='columns')
dataframe.head(2)
```

데이터셋 로드

➤ 데이터베이스로부터 데이터 로드

- read_sql_query()를 사용하여 데이터베이스에 SQL 쿼리를 던져 데이터를 적재
- SQLite 데이터베이스 엔진으로 연결하기 위해 create_engine 함수를 사용합니다.

```
import pandas as pd
from sqlalchemy import create_engine

database_connection = create_engine('sqlite://sample.db')

dataframe = pd.read_sql_query(' select * from data', database_connection)

dataframe.head(2)

dataframe = pd.read_sql_table('data', database_connection)
dataframe.head(2)
```

데이터 랭글링(data wrangling)

데이터 랭글링(data wrangling)

➤ 데이터 랭글링(data wrangling)

- 광범위한 의미의 원본 데이터를 정제하고 사용 가능한 형태로 구성하기 위한 변환 과정
- 데이터 랭글링에 사용되는 가장 일반적인 데이터 구조 - 데이터프레임.

```
import pandas as pd
```

```
url = 'https://tinyurl.com/titanic-csv'  
dataframe = pd.read_csv(url)  
dataframe.head(5)
```

```
import pandas as pd
```

```
dataframe = pd.DataFrame()  
dataframe['Name'] = ['Jacky Jackson', 'Steven Stevenson']  
dataframe['Age'] = [38, 25]  
dataframe['Driver'] = [True, False]  
Dataframe  
new_person = pd.Series(['Molly Mooney', 40, True],  
                        index = ['Name', 'Age', 'Driver' ]) #열 생성  
dataframe.append(new_person, ignore_index = True) #열 추가  
dataframe
```

데이터 랭글링(data wrangling)

➤ 데이터프레임 생성

- 열 이름은 columns 매개변수에 지정
- 원본 리스트를 전달하여 데이터프레임 생성
- 열 이름과 데이터를 매핑한 딕셔너리를 사용해 데이터프레임 생성

```
import pandas as pd

data = [['Jacky Jackson', 38, True], ['Steven Stevenson', 25, False] ]
matrix = np.array(data)
pd.DataFrame(matrix, columns=['Name', 'Age', 'Driver'])
pd.DataFrame(data, columns=['Name', 'Age', 'Driver'])

data = {'Name' : ['Jacky Jackson' , 'Steven Stevenson'],
        'Age' : [38, 25],
        'Driver' : [True, False]}
pd.dataFrame(data)

data = [ {'Name': 'Jacky Jackson', 'Age' : 38, 'Driver' : True},
        {'Name': 'Steven Stevenson', 'Age' : 25, 'Driver' : False} ]
pd.DataFrame(data, index=['row1', 'row2'])
```


데이터 랭글링(data wrangling)

➤ 데이터프레임 구조 이해

- head() - 데이터의 처음 몇 개의 행을 확인
- tail() - 데이터의 마지막 몇 개의 행을 확인
- shape() - 데이터프레임의 행과 열의 수를 확인
- describe() - 수치형 열의 기본 통계를 확인
- loc(), iloc()를 사용하여 하나 이상의 행이나 값을 선택합니다.
- 콜론(:)을 사용하여 원하는 행의 슬라이스를 선택할 수 있습니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)
dataframe.head(2)
dataframe.shape
dataframe.describe()
dataframe.iloc[0]
dataframe.iloc[1:4]
dataframe.iloc[:4]
```

데이터 랭글링(data wrangling)

➤ 데이터프레임 행 선택

-

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe[dataframe['Sex'] == 'female'].head(2)  #조건문으로 행 선택
dataframe[(dataframe['Sex'] == 'female') & (dataframe['Age'] >=65)]  #여러 조건으로 행 선택 가능
```

데이터 랭글링(data wrangling)

➤ 값 치환

- `replace()` : 하나의 열 또는 전체 DataFrame 객체에서 값을 찾아 바꿀 수 있습니다
정규 표현식을 이용하여 값을 찾아 바꿀 수 있습니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)
dataframe['Sex'].replace("female", "Woman").head(2)
dataframe['Sex'].replace(["female", "male"], [ "Woman", "Man"] ).head(5)
dataframe.replace(r"1st", "First", regex=True).head(2)
dataframe.replace(["female", "male", "person").head(3)
dataframe.replace({"female" : 1, "male": 0 }).head(3)
```

데이터 랭글링(data wrangling)

➤ 열 이름 변경

- rename() - 여러 개의 열 이름을 변경할 경우 columns 매개변수에 딕셔너리를 전달
index 매개변수를 사용하여 인덱스 변경

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.rename(columns={'PClass' : 'Passenger Class'}).head(2)
dataframe.rename(columns={'PClass' : 'Passenger Class', 'Sex' : 'Gender' }).head(2)

import collections
column_names = collections.defaultdict(str)
for name in dataframe.columns :      #키를 만듭니다.
    column_names[name]

column_names

dataframe.rename(index={0:-1}).head(2)
dataframe.rename(str.lower, axis='columns').head(2)
```

데이터 랭글링(data wrangling)

➤ 최소값, 최대값, 합, 평균 계산

- std() : 표준편차
- kurt() : 첨도, 확률분포의 뾰족한 정도, 첨도가 3에 가까우면 정규분포와 비슷하며 3보다 작을 경우에는 정규분포보다 더 납작하고 3보다 크면 더 뾰족합니다.
- skew() : 확률분포의 비대칭도. 음수일 경우 정규분포보다 오른쪽으로 치우쳐 있고 양수일 경우 반대인 왼쪽으로 치우쳐 있습니다.
- sem() : 표준오차, 샘플링된 표본의 평균에 대한 표준편차입니다
- mode() : 최빈값
- median() : 중간값
- corr() : 상관계수
- cov() : 공분산

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

print('최대값 : ', dataframe['Age'].max())
print('최소값 : ', dataframe['Age'].min())
print('평균 : ', dataframe['Age'].mean())
print('합 : ', dataframe['Age'].sum())
print('카운트 : ', dataframe['Age'].count())
dataframe.corr()
dataframe.cov()
```

데이터 랭글링(data wrangling)

➤ 고유값 찾기

- unique와 value_unique는 범주형 열을 탐색하거나 조작할 때 유용
- value_counts() : 고유값과 등장 횟수를 출력
- nunique() : 고유값의 개수 반환

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe['Sex'].unique()
dataframe['Sex'].value_counts()

dataframe['PClass'].value_counts()
dataframe['PClass'].nunique()
dataframe.nunique()
```

데이터 랭글링(data wrangling)

➤ 누락된 값 처리

- 판다스는 넘파이의 NaN("Not A Number")를 사용하여 누락된 값을 표시합니다.
- isnull() : 불리언 값 반환
- notnull() : 불리언 값 반환

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe[dataframe['Age'].isnull()].head(2)

import numpy as np
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)

#데이터를 로드하고 누락된 값을 설정
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

데이터 랭글링(data wrangling)

➤ 누락된 값 처리

- keep_default_na 매개변수를 False로 지정하면 판다스가 기본적으로 NaN으로 인식하는 문자열을 NaN으로 인식하지 않습니다.
- na_filter를 False로 설정하면 NaN 변환을 하지 않습니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe = pd.read_csv(url, na_values=['female'], keep_default_na=False)
dataframe[12:14]

dataframe = pd.read_csv(url, na_filter=False)
dataframe[12:14]
```


데이터 랭글링(data wrangling)

➤ 열 삭제

- `drop()` : `axis=1` 매개변수 사용.
- 여러 열을 삭제해야 하는 경우 첫번째 매개변수로 열 이름을 리스트로 전달
- `dataframe.columns`에 열 인덱스를 지정하여 삭제할 수 있습니다
- `inplace` 매개변수를 `True`로 설정하면 데이터프레임을 수정 가능한 객체처럼 다루므로 데이터 처리 파이프라인을 더욱 복잡하게 만듭니다.

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.drop('Age', axis=1).head(2)
dataframe.drop(['Age', 'Sex'], axis=1).head(2)

dataframe.drop(dataframe.columns[1], axis=1).head(2)

del dataframe['Age'] #판단스 내부의 호출 방식 때문에 권장하지 않음

dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

데이터 랭글링(data wrangling)

➤ 행 삭제

- 불리언 조건을 사용하여 삭제하고 싶은 행을 제외한 새로운 데이터프레임을 만듭니다.
- 조건을 사용하면 행 하나 또는 여러 개를 동시에 삭제할 수 있습니다.
- `drop_duplicates()` 중복된 행 삭제
- `drop_duplicates`는 기본적으로 모든 열이 완벽히 동일한 행만 삭제
- 일부 열만 대상으로 중복된 행을 검사하여 삭제할 경우 `subset` 매개변수를 사용
- `keep` 매개변수는 중복된 행의 첫 행을 유지할지 마지막 행을 유지할지 선택할 수 있습니다.
- `duplicated()` : 행이 중복되었는지 여부를 알려주는 불리언 시리즈를 반환

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe[dataframe['Sex'] != 'male'].head(2)
dataframe[dataframe.index != 0].head(2))

dataframe.drop_duplicates().head(2)
print("원본 데이터프레임 행의 수 :", len(dataframe))
print("중복 삭제 후 행의 수 :", len(dataframe.drop_duplicates()))
dataframe.drop_duplicates(subset=['Sex'])
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

데이터 랭글링(data wrangling)

➤ 값에 따라 행을 그룹핑

- `groupby()` : 통계적 계산과 같이 각 그룹에 적용할 연산이 필요
- `resample()` - 시간 간격에 따라 행을 그룹핑, `datetime` 형태의 인덱스를 사용하므로 시간 간격(오프셋)을 넓혀서 행을 그룹핑

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.groupby('Survived')['Name'].count()
dataframe.groupby(['Sex', 'Survived'])['Age'].mean()
```

```
import numpy as np

time_index = pd.date_range('06/06/2017', periods=10000, freq='305')

dataframe = pd.DataFrame(index=time_index)
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)
dataframe.resample('W').sum() #주 단위로 행을 그룹핑한 다음 합 계산
dataframe.resample('2W').mean() #2주 단위로 그룹핑하고 평균을 계산
dataframe.resample('M').count() #한달 간격으로 그룹핑하고 행을 카운트
```

데이터 랭글링(data wrangling)

- 열 원소 순회, 모든 열 원소에 함수 적용
 - 반복문 외에 리스트 컴프리헨션을 사용할 수 있습니다.
 - `apply()` : 열의 모든 원소에 내장 함수나 사용자 정의 함수를 적용합니다
 - `map()` : `apply`와 유사, 딕셔너리를 입력으로 넣을 수 있음

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

for name in dataframe['Name'][0:2]:
    print(name.upper())

[name.upper() for name in dataframe['Name'][0:2]]

def uppercase(x):
    return x.upper()

dataframe['Name'].apply(uppercase)[0:2]
dataframe['Survived'].map({1: 'Live', 0: 'Dead'})[:5]
dataframe['Age'].apply(lambda x, age : x < age, age=30)[:5]
```

데이터 랭글링(data wrangling)

- 열 원소 순회, 모든 열 원소에 함수 적용
 - apply()와 applymap() : 데이터프레임 전체에 적용
 - apply() : 데이터프레임 열 전체에 적용
 - applymap() : 열의 각 원소에 적용

```
import pandas as pd

url = 'https://tinyurl.com/titanic-csv'
dataframe = pd.read_csv(url)

dataframe.apply(lambda x: max(x)) #각 열에서 큰 값을 뽑음
def truncate_string(x):
    if type(x) == str:
        return x[:20]
    return x

dataframe.applymap(truncate_string)[5] #문자열의 길이를 최대 20자로 줄입니다.

dataframe.groupby('Sex').apply(lambda x: x.count())
```

데이터 랭글링(data wrangling)

➤ 데이터 프레임 연결하기

- concat (axis=0) : 행의 축에 따라 연결
- append() : 데이터프레임에 새로운 행을 추가

```
import pandas as pd

data_a = {'id':['1', '2', '3'],
          'first' : ['Alex', 'Amy', 'Allen'],
          'last' : ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns=['id', 'first', 'last'])

data_b = {'id':['4', '5', '6'],
          'first' : ['Billy', 'Brian', 'Bran'],
          'last' : ['Bonder', 'Black', 'Balwner']}
dataframe_a = pd.DataFrame(data_a, columns=['id', 'first', 'last'])
pd.concat([dataframe_a, dataframe_b], axis=0)
pd.concat([dataframe_a, dataframe_b], axis=1)

row = pd.Series(10, 'Chris', 'Chillon' ], index=['id', 'first', 'last'])
dataframe_a.append(row, ignore_index=True)
```

데이터 랭글링(data wrangling)

➤ 데이터 프레임 병합

- merge() - 내부 조인 수행을 위해 on 매개변수에 병합 열을 지정
동일한 매개변수로 왼쪽 조인과 오른쪽 조인을 지정할 수 있습니다.
- 각 데이터프레임에서 병합하기 위한 열 이름을 지정할 수 있습니다
- 각 데이터프레임의 인덱스를 기준으로 병합하려면 left_on과 right_on 매개변수를 right_index=True와 left_index=True로 바꿉니다.

```
import pandas as pd

employee_data = {'employee_id' : ['1', '2', '3', '4'],
                 'name' : ['Amy Jones', 'Allen Keys', 'Alice Bees', 'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns = ['employee_id', 'name'])

sales_data = {'employee_id' : ['3','4', '5','6'],
              , 'total_sales' : [23456, 2512, 2345, 1455] }
dataframe_sales = pd.DataFrame(sales_data, columns=['employee_id', 'total_sales'])
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='left')
pd.merge(dataframe_employees, dataframe_sales, left_on='employee_id', right_on='employee_id', )
```

데이터 랭글링(data wrangling)

➤ 데이터 프레임 병합

- how매개변수 inner : 두 데이터프레임에 모두 존재하는 행만 반환
- how매개변수 outer : 두 데이터프레임의 모든 행이 반환, 행이 한쪽 데이터프레임에만 존재한다면 누락된 값은 NaN으로 채워짐
- how매개변수 left : 왼쪽 데이터프레임의 모든 행이 반환, 오른쪽 데이터프레임은 왼쪽의 데이터프레임과 매칭되는 행만 반환
- how매개변수 right : 오른쪽 데이터프레임의 모든 행이 반환, 왼쪽 데이터프레임은 오른쪽의 데이터프레임과 매칭되는 행만 반환

수치형 데이터 처리

데이터 스케일링이란 데이터 전처리 과정의 하나입니다.

데이터 스케일링을 해주는 이유는 데이터의 값이 너무 크거나 혹은 작은 경우에 모델 알고리즘 학습과정에서 0으로 수렴하거나 무한으로 발산해버릴 수 있기 때문입니다.

따라서, scaling은 데이터 전처리 과정에서 굉장히 중요한 과정입니다.