# 03. 유용한 라이브러리 : 판다스와 넘파이

#### 학습 목표

판다스와 넘파이는 파이썬에서 데이터를 다루기 위해 사용되는 가장 대표적인 라이브러리입니다. 파이썬 머신러닝에서는 일반적으로 판다스와 넘파이를 이용하여 데이터를 다룹니다. 이 장에서는 판다스와 넘파이에 대한 기초적인 개념을 학습하고 머신러닝을 배우면서 더 다양한 내용을 학습하게 됩니다. 판다스와 넘파이에 익숙한 분은 이 장을 건너뛰어 4장으로 이동해주세요.

#### 판다스 소개

데이터 분석에서 가장 많이 쓰이는 라이브러리인 판다스는 데이터프레임과 시리즈라는 두 가지 자료구조를 제공합니다. 열이 하나면 시리즈, 둘 이상이면 데이터프레임입니다. 그래서 행이 10개 있는 데이터프레임에서 첫 번째 행만 불러오면 기본적으로 판다스 시리즈로 불러옵니다.

#### 넘파이 소개

판다스가 사람이 읽기 쉬운 형태의 자료구조를 제공한다면, 넘파이는 컴퓨터가 계산하기 좋은 형태로 제공합니다. 그래서 판다스에 비해 사람이 눈으로 읽기에는 다소 불편하지만 메모리가 덜 들고 계산도 더 빠릅니다.

#### 판다스와 넘파이 비교

| **판다스** | **구분** | **넘파이** |
| --- | --- | --- |
| * 하나 이상의 자료형을 원소로 가질 수 있음 * 테이블 형식의 작업(SQL과 같은 쿼리나 조인) 가능 * 2차원 이하 배열의 데이터 | 특징 | * 같은 자료형만 원소로 가질 수 있음 * 행렬 및 벡터 연산 기반 * 3차원 이상의 배열도 가능 |
|
|
|
|
| * 상대적으로 더 많이 필요 | 메모리 | * 상대적으로 더 조금 필요 |
| * 느림 | 속도 | * 빠름 |
| * SQL, 엑셀 파일, CSV 파일, 데이터베이스에서 데이터를 읽어들이고, 반대로 데이터를 파일 형식으로 만들거나 데이터베이스에 올릴 수 있음 | 입출력 | * npy, npz와 같은 확장자 혹은 텍스트 파일로 입출력 |

## 3.1 판다스

판다스는 pandas 라이브러리를 임포트해야 사용할 수 있습니다. 다음과 같이 import를 이용해 불러옵시다.

| import pandas |
| --- |

불러온 라이브러리는 해당 이름을 사용합니다. 여기에서는 pandas라는 이름을 써서 판다스 라이브러리를 사용하는데, 불러오는 시점에서 as를 사용하면 불러온 라이브러리의 이름을 별도로 지정할 수 있습니다. 판다스는 보통 pd라는 약자로 불러와서 사용합니다.

| import pandas as pd |
| --- |

판다스는 데이터프레임과 시리즈라는 두 가지 자료구조를 제공하는데, 데이터프레임의 한 열만 떼어내면 시리즈가 됩니다. 반대로 시리즈를 하나 이상 병합한 형태가 데이터프레임입니다.



위 그림에서 보시는 것처럼 한 행만 있는 데이터(apple행과 orange행 각각)는 시리즈 형태이고, 이 둘이 합쳐진 우측 형태는 데이터프레임입니다. 필요하다면 시리즈 형태를 데이터프레임으로 강제변환도 가능하지만, 기본적으로 판다스에서 한 행으로 된 데이터는 시리즈 형태로 읽어져옵니다.

3.1.1 판다스로 csv 파일 불러오기

판다스를 사용해서 sample.csv 파일을 불러오겠습니다. 해당 파일의 URL을 file\_url이라는 이름으로 저장하고, 이를 read\_csv() 함수 안에 써주면 됩니다. 그러면 판다스 데이터프레임 형태로 데이터를 불러옵니다. 이를 원하는 객체에 저장하면 해당 객체를 이용해 데이터를 사용할 수 있습니다. 여기서는 sample이라는 객체로 저장하겠습니다.

| file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/sample.csv'  sample = pd.read\_csv(file\_url) |
| --- |

이제 sample을 실행하면 저장된 데이터의 출력물을 볼 수 있습니다.

| sample |
| --- |

|  | Var\_1 | Var\_2 |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 3 | 4 |
| 2 | 2 | 5 |
| 3 | 5 | 6 |
| 4 | 3 | 2 |
| 5 | 1 | 1 |
| 6 | 2 | 6 |
| 7 | 5 | 7 |
| 8 | 6 | 8 |
| 9 | 7 | 4 |
| 10 | 7 | 7 |
| 11 | 8 | 3 |
| 12 | 9 | 7 |
| 13 | 3 | 9 |
| 14 | 2 | 2 |
| 15 | 1 | 3 |
| 16 | 2 | 6 |
| 17 | 2 | 4 |
| 18 | 3 | 3 |

19행 2열 데이터가 출력되었습니다. 가장 왼쪽의 숫자는 인덱스라고하며 행의 이름입니다. 기본적으로 0부터 시작하는 숫자가 부여되며, 변경도 가능합니다.

3.1.2 데이터프레임을 살펴보는 다양한 함수

데이터프레임을 살펴보는 다양한 함수를 알아봅시다. 앞에서는 총 19행이 전부 출력되었는데, 데이터의 앞부분 혹은 뒷부분만 간략하게 살펴보는 함수가 있습니다. 바로 head()와 tail()입니다. 이 두 함수는 기본적으로 5줄만 출력합니다. 더 많거나 적은 줄을 출력하려면 괄호 안에 원하는 숫자를 넣어주면 됩니다. 사용 예제를 몇 가지 보여드리겠습니다.

| sample.head() |
| --- |

|  | Var\_1 | Var\_2 |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 3 | 4 |
| 2 | 2 | 5 |
| 3 | 5 | 6 |
| 4 | 3 | 2 |

| sample.head(3) |
| --- |

|  | Var\_1 | Var\_2 |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 3 | 4 |
| 2 | 2 | 5 |

| sample.tail() |
| --- |

|  | Var\_1 | Var\_2 |
| --- | --- | --- |
| 14 | 2 | 2 |
| 15 | 1 | 3 |
| 16 | 2 | 6 |
| 17 | 2 | 4 |
| 18 | 3 | 3 |

| sample.tail(10) |
| --- |

|  | Var\_1 | Var\_2 |
| --- | --- | --- |
| 9 | 7 | 4 |
| 10 | 7 | 7 |
| 11 | 8 | 3 |
| 12 | 9 | 7 |
| 13 | 3 | 9 |
| 14 | 2 | 2 |
| 15 | 1 | 3 |
| 16 | 2 | 6 |
| 17 | 2 | 4 |
| 18 | 3 | 3 |

다음은 info() 함수를 사용해 데이터를 확인해봅시다. info() 함수는 데이터에 대한 요약 정보를 출력합니다.

| sample.info() |
| --- |



❶ 데이터 행 수(19)와 인덱스(0 to18)를 보여줍니다.

❷ 몇 개의 열이 있는지 보여줍니다.

❸ 데이터가 가지고 있는 변수명을 보여줍니다.

❹ Non-Null Count는 Null이 아닌 데이터 수입니다. 여기서 Null이란 데이터가 비어 있다는 뜻입니다. 이 데이터에는 비어 있는 값이 없으므로 Var\_1과 Var\_2 모두에서 19를 출력합니다.

❺ 자료형을 의미합니다. Var\_1과 Var\_2 모두 int64라는 자료형입니다.

판다스에서 볼 수 있는 주요 자료형은 다음과 같습니다.

▽ 판다스 자료형

| **자료형** | **설명** |
| --- | --- |
| object | 텍스트와 같은 문자 형태입니다. ' ' 혹은 " "으로 감싸졌습니다. 숫자도 따옴표로 감싸면 object 타입으로 인식됩니다. |
| int64 | 소수점이 없는 숫자입니다. |
| float64 | 소수점이 있는 숫자입니다. |
| bool | 불리언이라고 하며, True/False 두 가지 형태의 값을 지닌 자료형입니다. |
| datetime64 | 날짜/시간을 다루는 자료형입니다. |

마지막으로 데이터의 통계 정보를 출력하는 describe() 함수를 사용해봅시다.

| sample.describe() |
| --- |

|  | Var\_1 | Var\_2 |
| --- | --- | --- |
| count | 19.000000 | 19.000000 |
| mean | 3.789474 | 4.684211 |
| std | 2.529360 | 2.310667 |
| min | 1.000000 | 1.000000 |
| 25% | 2.000000 | 3.000000 |
| 50% | 3.000000 | 4.000000 |
| 75% | 5.500000 | 6.500000 |
| max | 9.000000 | 9.000000 |

이 함수는 각 변수(행)별로 통계적 정보를 요약해 보여줍니다. 보여주는 통계적 정보는 다음과 같습니다.

▽ 통계적 정보

| **용어** | **설명** |
| --- | --- |
| count | 데이터 개수를 보여줍니다. |
| mean | 각 변수에 속하는 데이터의 평균값입니다. |
| std | 각 변수에 속하는 데이터의 표준편차입니다. |
| min, max | 각 변수에 속하는 최솟값과 최댓값입니다. |
| 25%, 50%, 75% | 각 변수의 사분위수입니다. 사분위수란 데이터를 오름차순으로 정리해 25%, 50%, 75% 되는 지점을 의미합니다. |

3.1.3 데이터프레임 직접 만들기

이 책에서는 대부분 csv 파일을 데이터프레임 형태로 불러옵니다. 하지만 분석을 하다 보면 종종 직접 데이터프레임을 만들어야 하는 상황도 있습니다. 이번에는 직접 데이터프레임을 만드는 법을 알아보겠습니다.

우선 데이터프레임에 들어갈 데이터가 필요합니다. 위에서 배운 딕셔너리 형태로 정의하면 쉽게 데이터프레임을 만들 수 있습니다. 다음과 같이 sample\_dic이라는 이름의 딕셔너리를 만들고, 판다스의 DataFrame() 함수 안에 넣어줍니다(대소문자 구분해주세요).

| sample\_dic = {'name': ['John', 'Ann','Kevin'], 'age': [23, 22, 21]}  pd.DataFrame(sample\_dic) |
| --- |

|  | name | age |
| --- | --- | --- |
| 0 | John | 23 |
| 1 | Ann | 22 |
| 2 | Kevin | 21 |

데이터프레임 형태로 변환되었습니다. 이번에는 변수명과 인덱스명을 별도로 지정하기 위해, 리스트를 이용해 데이터프레임을 만들어봅시다. 리스트는 한 줄이고 데이터프레임은 행과 열이 있기 때문에, 리스트 안에 리스트를 넣는 방법으로 데이터를 만들겠습니다.

| [[1, 2],  [3, 4],  [5, 6],  [7, 8]] |
| --- |

[[1, 2], [3, 4], [5, 6], [7, 8]]

4개의 리스트 [1,2],[3,4],[5,6],[7,8]를 또 하나의 [ ]가 감싸고 있습니다. 리스트 안에 리스트가 포함된 형태로, 데이터프레임으로 불러오면 4행 2열의 형태로 인식됩니다. 위의 예시는 이해하기 쉽게 네 줄로 나누어 썼으나 한 줄로 써도 같은 결과를 보입니다. 그럼 이 리스트를 사용해 데이터프레임을 만들겠습니다.

| pd.DataFrame([[1, 2],[3, 4],[5, 6],[7, 8]]) |
| --- |

|  | 0 | 1 |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 3 | 4 |
| 2 | 5 | 6 |
| 3 | 7 | 8 |

기대한 대로 데이터프레임이 만들어졌습니다. 이번에는 columns와 index 매개변수를 추가해, 변수명을 var\_1, var\_2로, 인덱스를 a,b,c,d로 설정해 데이터프레임을 만들겠습니다.

| pd.DataFrame([[1, 2],[3, 4],[5, 6],[7, 8]], columns = ['var\_1', 'var\_2'], index=['a', 'b', 'c', 'd']) |
| --- |

|  | var\_1 | var\_2 |
| --- | --- | --- |
| a | 1 | 2 |
| b | 3 | 4 |
| c | 5 | 6 |
| d | 7 | 8 |

위와 같이 columns와 index 매개변수에 원하는 이름을 리스트 형태로 지정해주면 됩니다.

3.1.4 데이터프레임 인덱싱

데이터프레임의 특정 행과 열에서 데이터의 일부를 선택하는 것을 인덱싱이라고 합니다. sample\_df.csv를 불러와서 인덱싱을 하겠습니다.

이번에는 csv 파일을 불러올 때 index\_col이라는 매개변수를 추가합니다. index\_col은 특정 행을 인덱스로 지정하는 역할을 합니다. 이 데이터에서는 0번째 행을 인덱스로 사용합니다.

| file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/sample\_df.csv'  sample\_df = pd.read\_csv(file\_url, index\_col=0) |
| --- |

sample\_df를 확인하면 다음과 같은 형태입니다.

| sample\_df |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| a | 2 | 2 | 1 | 4 | 3 |
| b | 4 | 3 | 3 | 7 | 1 |
| c | 5 | 4 | 6 | 3 | 5 |
| d | 1 | 4 | 5 | 6 | 7 |
| e | 4 | 5 | 7 | 8 | 3 |
| f | 5 | 4 | 8 | 9 | 4 |
| g | 7 | 5 | 2 | 0 | 6 |
| h | 8 | 8 | 1 | 7 | 8 |
| i | 2 | 3 | 5 | 2 | 1 |
| j | 9 | 3 | 7 | 6 | 5 |

#### 컬럼 기준으로 인덱싱하기

우선 컬럼 기준으로 인덱싱하겠습니다. 변수를 인덱싱하려면 [ ] 안에 인덱싱할 변수명을 써서 붙여주면 됩니다. var\_1을 인덱싱하는 코드는 다음과 같습니다.

| sample\_df['var\_1'] |
| --- |

a 2

b 4

c 5

d 1

e 4

f 5

g 7

h 8

i 2

j 9

Name: var\_1, dtype: int64

var\_1에 해당하는 한 줄의 데이터가 나왔습니다. 한 줄짜리 데이터이다보니 판다스 시리즈 형태로 출력됩니다. 시리즈는 데이터프레임과는 다르게 위에 컬럼명이 표기되지 않습니다.

이번에는 var\_1과 var\_2를 함께 인덱싱해보겠습니다. 우선 다음과 같이 시도하겠습니다.

| sample\_df['var\_1', 'var\_2'] |
| --- |

<출력 결과/>

**KeyError** Traceback (most recent call last)

**~\anaconda3\lib\site-packages\pandas\core\indexes\base.py** in get\_loc**(self, key, method, tolerance)**

3079 **try:**

**-> 3080 return** self**.**\_engine**.**get\_loc**(**casted\_key**)**

3081 **except** KeyError **as** err**:**

... 중략 ...

**KeyError**: ('var\_1', 'var\_2')

</>

KeyError가 발생했습니다. 이 코드가 작동하지 않는 이유는 컬럼을 인덱싱할 때 [ ] 안에는 단 하나의 값만 넣을 수 있기 때문입니다. var\_1과 var\_2는 두 개이므로 이를 하나의 값으로 합쳐주어야 합니다. 리스트, 즉 [ ] 기호를 사용해서 [var\_1, var\_2]와 같이 만들면 하나의 값으로 인식되게 만들 수 있습니다.

| sample\_df[['var\_1', 'var\_2']] |
| --- |

|  | var\_1 | var\_2 |
| --- | --- | --- |
| a | 2 | 2 |
| b | 4 | 3 |
| c | 5 | 4 |
| d | 1 | 4 |
| e | 4 | 5 |
| f | 5 | 4 |
| g | 7 | 5 |
| h | 8 | 8 |
| i | 2 | 3 |
| j | 9 | 3 |

원하는 대로 인덱싱되었습니다. 여기서 [[ ]]의 형태를 다시 한번 설명드리면, 바깥쪽의 [ ]는 인덱싱을 위한 것이고, 안쪽의 [ ]는 변수명들을 하나의 값으로 합칠 리스트를 정의하는 용도입니다. 변수 개수가 3개든 4개든 리스트 하나로 묶어주면 됩니다.

#### 행 기준으로 인덱싱하기

다음은 행(인덱스) 기준으로 인덱싱하겠습니다. 이 데이터에서는 행에도 a,b,c,d,e와 같은 이름이 붙어 있으므로 loc[ ]을 사용하면 이름을 기준으로 인덱싱할 수 있습니다. a행을 인덱싱하겠습니다.

| sample\_df.loc['a'] |
| --- |

var\_1 2

var\_2 2

var\_3 1

var\_4 4

var\_5 3

Name: a, dtype: int64

a행의 데이터가 시리즈 형태로 출력되며, 시리즈 형태이기 때문에 기존의 가로 배열이 아닌 세로 배열로 출력되었습니다. 이번에는 a,b,c 행을 인덱싱해봅시다. 컬럼 때와 마찬가지로 a,b,c를 리스트로 묶어주어야 합니다.

| sample\_df.loc[['a','b','c']] |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| a | 2 | 2 | 1 | 4 | 3 |
| b | 4 | 3 | 3 | 7 | 1 |
| c | 5 | 4 | 6 | 3 | 5 |

조금 다른 형식으로 “a부터 c까지”라는 의미인 ‘a’ : ‘c’를 이용할 수도 있습니다. ‘a’ : ‘c’는 자체가 하나의 값으로 인식되므로 리스트를 사용할 필요가 없습니다.

| sample\_df.loc['a':'c'] |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| a | 2 | 2 | 1 | 4 | 3 |
| b | 4 | 3 | 3 | 7 | 1 |
| c | 5 | 4 | 6 | 3 | 5 |

같은 결과가 출력되었습니다. iloc[ ]를 이용하면 인덱스 이름이 아닌 행 위치를 기준으로 인덱싱할 수도 있습니다. 파이썬에서 행은 1이 아닌 0번째부터 시작하므로 0,1,2행을 인덱싱하겠습니다.

| sample\_df.iloc[[0,1,2]] |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| a | 2 | 2 | 1 | 4 | 3 |
| b | 4 | 3 | 3 | 7 | 1 |
| c | 5 | 4 | 6 | 3 | 5 |

이번에는 행 순서를 하나씩 입력하지 말고 0 : 2를 이용해봅시다.

| sample\_df.iloc[0:2] |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| a | 2 | 2 | 1 | 4 | 3 |
| b | 4 | 3 | 3 | 7 | 1 |

예상과 다르게 단 두 줄만 출력되었습니다. [시작 : 끝] 형태로 인덱싱할 때 끝 숫자 이전 데이터까지 인덱싱합니다. 숫자 형태로 지정할 때 해당하는 것이기 때문에 위에서 사용한 [‘a’ : ‘c’]에서는 적용되지 않습니다. 0 ~ 2행을 인덱싱하려면 [0 : 3]으로 정의해야 합니다.

| sample\_df.iloc[0:3] |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| a | 2 | 2 | 1 | 4 | 3 |
| b | 4 | 3 | 3 | 7 | 1 |
| c | 5 | 4 | 6 | 3 | 5 |

iloc에서는 행 뿐만 아니라 열까지 동시에 인덱싱할 수 있습니다. 인덱싱할 행 뒤에 ,를 사용해 인덱싱할 열을 정의하면 됩니다. 그럼 0~2행, 2~3열을 인덱싱해봅시다.

| sample\_df.iloc[0:3, 2:4] |
| --- |

|  | var\_3 | var\_4 |
| --- | --- | --- |
| a | 1 | 4 |
| b | 3 | 7 |
| c | 6 | 3 |

#### 컬럼 제거하기

만약 총 5개 변수 중 4개 변수를 인덱싱한다면, 오히려 선택하지 않은 한 개 변수를 제거하는 것이 편합니다. drop() 함수를 사용해 var\_1 변수를 제거하겠습니다. 변수를 제거하려면 반드시 axis 매개변수에 1을 지정해야 합니다.

| sample\_df.drop('var\_1', axis=1) |
| --- |

|  | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- |
| a | 2 | 1 | 4 | 3 |
| b | 3 | 3 | 7 | 1 |
| c | 4 | 6 | 3 | 5 |
| d | 4 | 5 | 6 | 7 |
| e | 5 | 7 | 8 | 3 |
| f | 4 | 8 | 9 | 4 |
| g | 5 | 2 | 0 | 6 |
| h | 8 | 1 | 7 | 8 |
| i | 3 | 5 | 2 | 1 |
| j | 3 | 7 | 6 | 5 |

두 개 이상의 변수를 제거하려면 여기에서도 리스트 형식으로 변수명을 묶어주어야 합니다. var\_1과 var\_2를 제거하겠습니다.

| sample\_df.drop(['var\_1','var\_2'], axis=1) |
| --- |

|  | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- |
| a | 1 | 4 | 3 |
| b | 3 | 7 | 1 |
| c | 6 | 3 | 5 |
| d | 5 | 6 | 7 |
| e | 7 | 8 | 3 |
| f | 8 | 9 | 4 |
| g | 2 | 0 | 6 |
| h | 1 | 7 | 8 |
| i | 5 | 2 | 1 |
| j | 7 | 6 | 5 |

여기에서 axis 매개변수에 1을 지정하는 이유는 drop() 함수가 기본적으로 **행**을 제거하도록 설정되어 있기 때문입니다(axis=0). 즉, 아무것도 써주지 않으면 axis=0으로 반영되어 행 단위에서 해당 이름을 찾아 제거하고, axis=1을 쓰면 열 단위에서 해당 이름을 찾아서 제거하게 됩니다. 위의 경우는 열에서 var\_1과 var\_2를 찾아서 지우는 것이기 때문에 axis=1이 꼭 들어가야 합니다. 그럼 이번에는 a,b,c 행을 제거하는 코드를 작성하겠습니다.

| sample\_df.drop(['a','b','c']) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| d | 1 | 4 | 5 | 6 | 7 |
| e | 4 | 5 | 7 | 8 | 3 |
| f | 5 | 4 | 8 | 9 | 4 |
| g | 7 | 5 | 2 | 0 | 6 |
| h | 8 | 8 | 1 | 7 | 8 |
| i | 2 | 3 | 5 | 2 | 1 |
| j | 9 | 3 | 7 | 6 | 5 |

### 3.1.5 데이터프레임의 인덱스 변경

데이터프레임에 있는 인덱스를 특정 변수로 대체할 수도 있고, 인덱스를 별도의 변수로 빼내올 수 있습니다.

앞에서 다룬 sample\_df에서 인덱스를 별도의 변수로 빼내는 방법부터 살펴보겠습니다. reset\_index() 함수를 사용하면 됩니다.

| sample\_df.reset\_index() |
| --- |

|  | index | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | a | 2 | 2 | 1 | 4 | 3 |
| 1 | b | 4 | 3 | 3 | 7 | 1 |
| 2 | c | 5 | 4 | 6 | 3 | 5 |
| 3 | d | 1 | 4 | 5 | 6 | 7 |
| 4 | e | 4 | 5 | 7 | 8 | 3 |
| 5 | f | 5 | 4 | 8 | 9 | 4 |
| 6 | g | 7 | 5 | 2 | 0 | 6 |
| 7 | h | 8 | 8 | 1 | 7 | 8 |
| 8 | i | 2 | 3 | 5 | 2 | 1 |
| 9 | j | 9 | 3 | 7 | 6 | 5 |

index라는 새로운 변수가 추가되었고, 인덱스에는 기본값인 숫자로 대체되었습니다. 만약 기존 인덱스를 제거하되 새로운 변수로 추가하고 싶지 않다면, reset\_index()에 drop이라는 매개변수를 사용하면 됩니다.

| sample\_df.reset\_index(drop=True) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| 0 | 2 | 2 | 1 | 4 | 3 |
| 1 | 4 | 3 | 3 | 7 | 1 |
| 2 | 5 | 4 | 6 | 3 | 5 |
| 3 | 1 | 4 | 5 | 6 | 7 |
| 4 | 4 | 5 | 7 | 8 | 3 |
| 5 | 5 | 4 | 8 | 9 | 4 |
| 6 | 7 | 5 | 2 | 0 | 6 |
| 7 | 8 | 8 | 1 | 7 | 8 |
| 8 | 2 | 3 | 5 | 2 | 1 |
| 9 | 9 | 3 | 7 | 6 | 5 |

기존 인덱스값인 a,b,c,d,e가 숫자로 바뀌었고 새로운 변수는 추가되지 않았습니다.

이번에는 var\_1을 인덱스로 집어넣는 방법을 소개합니다. set\_index() 함수를 사용하며 괄호 안에 변수명을 넣어주면 됩니다.

| sample\_df.set\_index('var\_1') |
| --- |

|  | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- |
| var\_1 |  | | | |
| 2 | 2 | 1 | 4 | 3 |
| 4 | 3 | 3 | 7 | 1 |
| 5 | 4 | 6 | 3 | 5 |
| 1 | 4 | 5 | 6 | 7 |
| 4 | 5 | 7 | 8 | 3 |
| 5 | 4 | 8 | 9 | 4 |
| 7 | 5 | 2 | 0 | 6 |
| 8 | 8 | 1 | 7 | 8 |
| 2 | 3 | 5 | 2 | 1 |
| 9 | 3 | 7 | 6 | 5 |

### 3.1.6 데이터프레임의 변수별 계산

데이터프레임에서는 간단하게 변수별 합, 평균, 표준편차 등을 계산할 수 있습니다. 데이터프레임 이름 뒤에 sum() 함수를 추가하면 sample\_df의 변수별 합을 구하겠습니다.

| sample\_df.sum() |
| --- |

var\_1 47

var\_2 41

var\_3 45

var\_4 52

var\_5 43

dtype: int64

데이터프레임의 변수별 계산에 사용하는 주요 함수는 다음과 같습니다.

▽ 데이터프레임이 제공하는 주요 통계 함수

| 데이터 개수 | count() |
| --- | --- |
| 합 | sum() |
| 평균 | mean() |
| 중위값 | median() |
| 분산 | var() |
| 표준편차 | std() |

변수별 합과 평균을 함께 볼 수 있는 방법도 있습니다. aggregate() 함수의 괄호 안에 합과 평균을 뜻하는 sum과 mean을 입력하면 됩니다. 단, 하나의 값으로 입력되도록 sum과 mean을 리스트로 묶어주어야 합니다.

| sample\_df.aggregate(['sum','mean']) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| sum | 47.0 | 41.0 | 45.0 | 52.0 | 43.0 |
| mean | 4.7 | 4.1 | 4.5 | 5.2 | 4.3 |

### 3.1.7 그룹별 계산

판다스에서는 특정 변수를 기준으로 그룹을 만들어 계산할 수 있습니다. 우선 예제 데이터 iris.csv를 불러오겠습니다. 이 파일에는 붓꽃iris의 종마다 꽃받침과 꽃잎의 크기가 어떻게 다른지를 보여주는 데이터가 들어 있습니다.

| file\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/iris.csv' iris = pd.read\_csv(file\_url) |
| --- |

head() 함수를 사용해 데이터의 전반적인 모습을 살펴봅시다.

| iris.head() |
| --- |

|  | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | class |
| --- | --- | --- | --- | --- | --- |
| 0.0 | 6.1 | 3.0 | 4.6 | 1.4 | versicolor |
| 1 | 7.2 | 3.0 | 5.8 | 1.6 | virginica |
| 2 | 7 | 3 | 4 | 1 | versicolor |
| 3 | 6 | 3 | 4 | 1 | versicolor |
| 4 | 5 | 3 | 1 | 0 | setosa |

가장 우측의 변수 class는 붓꽃(iris)의 종류로 ‘versicolor’, ‘virginica’, ‘setosa’가 있습니다. 나머지 4개의 변수는 꽃받침(sepal)과 꽃잎(petal)의 길이와 너비입니다. 이 데이터에서 붓꽃의 종류별 꽃받침 길이와 너비, 꽃잎 길이와 너비 평균값을 간단히 계산할 수 있습니다. groupby() 함수를 사용해 그룹을 만들 변수를 지정하고, 이어서 평균을 내주는 함수 mean()을 붙여주면 됩니다.

| iris.groupby('class').mean() |
| --- |

| class | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) |
| --- | --- | --- | --- | --- |
| setosa | 5.006 | 3.428 | 1.462 | 0.246 |
| versicolor | 5.936 | 2.770 | 4.260 | 1.326 |
| virginica | 6.588 | 2.974 | 5.552 | 2.026 |

그룹별 합이나 표준편차 등을 구할 수도 있습니다. 또한 groupby() 뒤에 agg() 함수를 사용하면 여기서도 여러 가지 계산을 한 번에 할 수 있습니다. 붓꽃의 종류별로 전체 데이터 개수 (count)와, 평균(mean)을 동시에 확인하겠습니다.

| iris.groupby('class').agg(['count','mean']) |
| --- |

|  | sepal length (cm) | | sepal width (cm) | | petal length (cm) | | petal width (cm) | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| class | count | mean | count | mean | count | mean | count | mean |
| setosa | 50 | 5.006 | 50 | 3.428 | 50 | 1.462 | 50 | 0.246 |
| versicolor | 50 | 5.936 | 50 | 2.770 | 50 | 4.260 | 50 | 1.326 |
| virginica | 50 | 6.588 | 50 | 2.974 | 50 | 5.552 | 50 | 2.026 |

### 3.1.8 변수 내 고윳값 확인하기

위의 iris 데이터에서 class 변수에 ‘versicolor’, ‘virginica’, ‘setosa’ 3가지 종류가 있다고 설명드렸습니다. 하지만 누군가 이렇게 알려주지 않는다면 해당 변수에 어떤 종류가 있는지 어떻게 알 수 있을까요? unique()를 사용하면 데이터를 일일히 확인하지 않고 고유한 값들만 볼 수 있습니다.

| iris['class'].unique() |
| --- |

array(['versicolor', 'virginica', 'setosa'], dtype=object)

class에 등장하는 종류가 잘 정리되었네요.

nunique()를 이용하면 간단하게 몇 가지 종류인지 숫자로 확인할 수도 있습니다.

| iris['class'].nunique() |
| --- |

3

예상한 대로 3가지 종류라는 결과를 출력했네요.

마지막으로 각 종류별로 몇 건의 데이터가 있는지도 value\_counts()를 통해 확인할 수 있습니다.

| iris['class'].value\_counts() |
| --- |

setosa 50

virginica 50

versicolor 50

Name: class, dtype: int64

종류별로 각각 50개의 데이터가 있습니다. 이 함수들은 일종의 카테고리를 확인하는 역할을 하기 때문에 주로 object 타입, 즉 텍스트 형태로 된 변수에 사용합니다. int와 float 형태 같은 숫자형 데이터에도 사용할 수 있지만, 고윳값 종류가 너무 많다면 의미있는 정보를 얻기 힘듭니다.

### 3.1.9 데이터프레임 합치기

이번에는 다양한 방법으로 데이터프레임을 합쳐보겠습니다. 데이터를 결합하는 방법으로 크게 내부 조인(INNER JOIN), 전체 조인(FULL JOIN), 왼쪽 조인(LEFT JOIN), 오른쪽 조인(RIGHT JOIN)이 있습니다.



아직 명확히 이해가 안 됐다 해도 괜찮습니다. 이제부터 실습하면 금방 이해할 수 있을 겁니다. 실습에 사용할 left.csv와 right.csv 파일을 불러옵니다.

| left\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/left.csv' right\_url = 'https://media.githubusercontent.com/media/musthave-ML10/data\_source/main/right.csv'  left = pd.read\_csv(left\_url) right = pd.read\_csv(right\_url) |
| --- |

left, 오른쪽 테이블을 각각 살펴보겠습니다.

| left |
| --- |

|  | key | var\_1 | var\_2 |
| --- | --- | --- | --- |
| 0 | a | 1 | 1 |
| 1 | b | 3 | 2 |
| 2 | c | 4 | 4 |
| 3 | d | 2 | 3 |
| 4 | e | 1 | 0 |

| right |
| --- |

|  | key | var\_3 | var\_4 |
| --- | --- | --- | --- |
| 0 | b | 4 | 3 |
| 1 | c | 6 | 5 |
| 2 | e | 3 | 8 |
| 3 | f | 2 | 7 |
| 4 | g | 3 | 4 |

여기서 key에 주목하세요. 앞으로 이 key값을 기준으로 데이터를 합쳐줄 겁니다. left는 a,b,c,d,e, right는 b,c,e,f,g라는 key를 가지고 있습니다. 이 둘의 겹치는 부분은 b,c,e입니다.



#### merge() 함수로 결합하기

처음에는 merge() 함수를 사용해 두 데이터를 조인하겠습니다. merge() 함수는 괄호 안에 합치려는 데이터 이름을 넣어주면 됩니다.

| left.merge(right) |
| --- |

|  | key | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- | --- |
| 0 | b | 3 | 2 | 4 | 3 |
| 1 | c | 4 | 4 | 6 | 5 |
| 2 | e | 1 | 0 | 3 | 8 |

merge() 함수는 기본적으로 내부 조인을 수행합니다. 따라서 두 테이블에 공통된 key인 b,c,e만 결합되고, 나머지 데이터는 버려졌습니다.

또한 merge() 함수는 자동적으로 두 테이블에 공통으로 존재하는 변수명을 찾아 키값으로 활용합니다. 만약 특정 변수를 지정해 키값으로 활용하시고 싶으시면 on이라는 매개변수에 변수명을 지정해주면 됩니다.

| left.merge(right, on = {'원하는 변수'}) |
| --- |

이번에는 전체 조인을 해볼 텐데, 매개변수 how에 결합 방법을 ‘outer’를 지정해주면 됩니다.

| left.merge(right, how = 'outer') |
| --- |

|  | key | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- | --- |
| 0 | a | 1.0 | 1.0 | NaN | NaN |
| 1 | b | 3.0 | 2.0 | 4.0 | 3.0 |
| 2 | c | 4.0 | 4.0 | 6.0 | 5.0 |
| 3 | d | 2.0 | 3.0 | NaN | NaN |
| 4 | e | 1.0 | 0.0 | 3.0 | 8.0 |
| 5 | f | NaN | NaN | 2.0 | 7.0 |
| 6 | g | NaN | NaN | 3.0 | 4.0 |

양쪽 테이블의 모든 키값 a~g에 대해서 데이터가 조인되었으며, 키값에 해당하는 데이터가 없으면 NaN(빈 값)으로 표시했습니다.

왼쪽 조인과 오른쪽 조인도 마찬가지로, 특정 테이블에 조인에 반영된 키값이 없으면 NaN으로 표시합니다. 왼쪽 조인 예시만 살펴보겠습니다.

| left.merge(right, how = 'left') |
| --- |

|  | key | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- | --- |
| 0 | a | 1 | 1 | NaN | NaN |
| 1 | b | 3 | 2 | 4.0 | 3.0 |
| 2 | c | 4 | 4 | 6.0 | 5.0 |
| 3 | d | 2 | 3 | NaN | NaN |
| 4 | e | 1 | 0 | 3.0 | 8.0 |

키값은 왼쪽 테이블에 있는 키값들로만 되어 있고, 그중 오른쪽 테이블에 없는 키값에 대해서는 var\_3, var\_4가 NaN으로 처리되었습니다.

#### join() 함수로 조인하기

join() 함수는 인덱스를 key로 삼아서 테이블을 결합합니다. 그러나 아래와 같이 left와 right에 join을 사용하면 에러가 발생합니다.

| left.join(right) |
| --- |

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

[<ipython-input-121-894480a0fafb>](https://localhost:8080/#) in <module>()

----> 1 left.join(right)

4 frames

[/usr/local/lib/python3.7/dist-packages/pandas/core/reshape/merge.py](https://localhost:8080/#) in \_items\_overlap\_with\_suffix(left, right, suffixes)

**2312**

**2313** if not lsuffix and not rsuffix:

-> 2314 raise ValueError(f"columns overlap but no suffix specified: {to\_rename}")

**2315**

**2316** def renamer(x, suffix):

ValueError: columns overlap but no suffix specified: Index(['key'], dtype='object')

에러 메시지를 살펴보면 ['key']라는 이름의 컬럼이 겹쳐서 안 된다고 말하고 있습니다. 즉, join에서는 인덱스를 기준으로 합치기 때문에 left와 right에 있는 key 컬럼은 key 역할이 아닌 그저 변수의 역할로 합쳐지게 되는데, 이 이름이 겹치기 때문에 합칠 수 없다는 에러가 발생한 겁니다. 이 에러를 피하기 위하여 각 데이터에서 key 컬럼을 제외하고 합쳐보겠습니다.

| left.drop('key', axis=1).join(right.drop('key', axis=1)) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 4 | 3 |
| 1 | 3 | 2 | 6 | 5 |
| 2 | 4 | 4 | 3 | 8 |
| 3 | 2 | 3 | 2 | 7 |

이번에는 key 컬럼을 기준으로 합쳐보겠습니다. key 컬럼이 진짜 key 역할을 하려면 인덱스로 들어가줘야 합니다. 아래와 같이 set\_index로 key 컬럼을 인덱스로 설정하겠습니다.

| left = left.set\_index('key') right = right.set\_index('key') |
| --- |

join() 함수는 기본적으로 왼쪽 조인을 실행합니다. 사용 방법은 merge() 함수와 거의 유사합니다.

| left.join(right) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- |
| key |  |  |  |  |
| a | 1 | 1 | NaN | NaN |
| b | 3 | 2 | 4.0 | 3.0 |
| c | 4 | 4 | 6.0 | 5.0 |
| d | 2 | 3 | NaN | NaN |
| e | 1 | 0 | 3.0 | 8.0 |

인덱스에 key가 들어있는 점을 제외하면 merge()를 사용해 왼쪽 조인을 수행했을 때와 같습니다. join() 함수에도 on과 how 매개변수가 존재하기 때문에, 원하면 특정 변수명을 기준으로 결합할 수도 있고, 왼쪽 조인 외에 다른 형태도 가능합니다. 연습 삼아 내부 조인을 만들겠습니다.

| left.join(right, how='inner') |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- |
| key |  |  |  |  |
| b | 3 | 2 | 4 | 3 |
| c | 4 | 4 | 6 | 5 |
| e | 1 | 0 | 3 | 8 |

#### concat() 함수로 결합하기

concat() 함수 괄호 안에 리스트 형태로 결합할 데이터들을 넣어주면 되고, 행 기준으로 결합됩니다. merge()나 join()과는 다르게 데이터 이름 뒤에 붙여서 쓰지 않고, 판다스(pd) 뒤에 붙여서 사용합니다.

| pd.concat([left,right]) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- |
| key |  |  |  |  |
| a | 1.0 | 1.0 | NaN | NaN |
| b | 3.0 | 2.0 | NaN | NaN |
| c | 4.0 | 4.0 | NaN | NaN |
| d | 2.0 | 3.0 | NaN | NaN |
| e | 1.0 | 0.0 | NaN | NaN |
| b | NaN | NaN | 4.0 | 3.0 |
| c | NaN | NaN | 6.0 | 5.0 |
| e | NaN | NaN | 3.0 | 8.0 |
| f | NaN | NaN | 2.0 | 7.0 |
| g | NaN | NaN | 3.0 | 4.0 |

왼쪽 테이블이 위에, 오른쪽 테이블이 아래에 있는 모습이 보이나요? 두 테이블이 가지고 있는 변수명이 서로 달라서 수많은 NaN가 보입니다. 이렇게 concat()은 기본적으로 행 기준으로 결합시키는데, axis 매개변수를 1로 두면 열 기준으로도 결합할 수 있습니다.

| pd.concat([left,right], axis=1) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 |
| --- | --- | --- | --- | --- |
| key |  |  |  |  |
| a | 1 | 1 | NaN | NaN |
| b | 3 | 2 | 4 | 3 |
| c | 4 | 4 | 6 | 5 |
| d | 2 | 3 | NaN | NaN |
| e | 1 | 0 | 3 | 8 |
| f | NaN | NaN | 2 | 7 |
| g | NaN | NaN | 3 | 4 |

보시는 것처럼 인덱스를 기준으로 전체 조인이 되었습니다.

merge(), join(), concat()은 이처럼 조금씩 다르면서도, 어떻게 사용하느냐에 따라 같은 결과를 만들어낼 수도 있습니다.

▽ 판다스에서 조인을 수행하는 3가지 함수

| **함수명** | **설명** | **조인 예시** |
| --- | --- | --- |
| merge() | 특정 컬럼을 기준으로 데이터를 합침 | left.merge(right, how='left') |
| join() | 인덱스를 기준으로 데이터를 합침 | left.join(right) |
| concat() | 기본적으로 행을 기준으로 합치게 되며, axis=1를 사용하여 열 기준으로 합칠 수도 있음. 왼쪽/오른쪽 조인은 지원하지 않고, 내부/외부 조인만 가능 | pd.concat([left, right], axis=1) |

## 3.2 넘파이

넘파이Numpy도 판다스와 마찬가지로 라이브러리를 임포트해야 합니다. 보통 np라는 이름으로 줄여서 사용합니다.

| import numpy as np |
| --- |

넘파이를 사용하려면 배열을 알아야 합니다.



1차원 배열은 리스트나 튜플처럼 한 줄로 이루어진 형태의 배열입니다. 다음과 같은 방식으로 정의할 수 있습니다.

| np.array([1,2,3]) |
| --- |

array([1, 2, 3])

2차원 배열은 행과 열이 있는 엑셀 스프레드시트와 같은 형태의 배열이라고 보시면 됩니다. 다음과 같이 각각의 행을 하나로 묶어서 정의해주고, 이들 전체를 리스트로 한번 더 감싸주면 됩니다.

| np.array([[1,2,3],  [4,5,6],  [7,8,9]]) |
| --- |

array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

### 3.2.1 배열 생성

아주 간단한 넘파이 배열부터 만들겠습니다. np.array() 함수에 리스트를 넣어주면, 해당 리스트를 넘파이 배열로 변환할 수 있습니다.

| np.array([1,2,3,4,5]) |
| --- |

array([1, 2, 3, 4, 5])

판다스 형태 같은 2차원 데이터도 넘파이 배열로 가능합니다. 판다스 설명에서 썼던 sample\_df를 넘파이 배열로 변환해봅시다.

| np.array(sample\_df) |
| --- |

array([[2, 2, 1, 4, 3],

[4, 3, 3, 7, 1],

[5, 4, 6, 3, 5],

[1, 4, 5, 6, 7],

[4, 5, 7, 8, 3],

[5, 4, 8, 9, 4],

[7, 5, 2, 0, 6],

[8, 8, 1, 7, 8],

[2, 3, 5, 2, 1],

[9, 3, 7, 6, 5]])

판다스와 달리 인덱스명이나 변수명이 없습니다. 이처럼 판다스 데이터가 넘파이 배열로 변환될 수 있고, 반대로 넘파이 배열을 판다스 형식으로 변환할 수도 있습니다. 넘파이 배열에서는 인덱스나 변수명이 없기 때문에, 판다스 형식으로 변환하면 임의의 변수명으로 만들어집니다.

그럼 위의 넘파이 배열을 sample\_np로 저장한 뒤, 데이터프레임으로 변경하겠습니다.

| sample\_np = np.array(sample\_df)  pd.DataFrame(sample\_np) |
| --- |

|  | 0 | 1 | 2 | 3 | 4 |
| --- | --- | --- | --- | --- | --- |
| 0 | 2 | 2 | 1 | 4 | 3 |
| 1 | 4 | 3 | 3 | 7 | 1 |
| 2 | 5 | 4 | 6 | 3 | 5 |
| 3 | 1 | 4 | 5 | 6 | 7 |
| 4 | 4 | 5 | 7 | 8 | 3 |
| 5 | 5 | 4 | 8 | 9 | 4 |
| 6 | 7 | 5 | 2 | 0 | 6 |
| 7 | 8 | 8 | 1 | 7 | 8 |
| 8 | 2 | 3 | 5 | 2 | 1 |
| 9 | 9 | 3 | 7 | 6 | 5 |

변수명이 0,1,2,3,4로 되었습니다. 그럼 변수명을 추가해볼까요? 일단 기존의 sample\_df 뒤에 columns를 붙이면 변수명을 받아올 수 있습니다.

| sample\_df.columns |
| --- |

Index(['var\_1', 'var\_2', 'var\_3', 'var\_4', 'var\_5'], dtype='object')

이 값을 DataFrame의 columns 매개변수에 넣어줍니다.

| pd.DataFrame(sample\_np, columns = sample\_df.columns) |
| --- |

|  | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 |
| --- | --- | --- | --- | --- | --- |
| 0 | 2 | 2 | 1 | 4 | 3 |
| 1 | 4 | 3 | 3 | 7 | 1 |
| 2 | 5 | 4 | 6 | 3 | 5 |
| 3 | 1 | 4 | 5 | 6 | 7 |
| 4 | 4 | 5 | 7 | 8 | 3 |
| 5 | 5 | 4 | 8 | 9 | 4 |
| 6 | 7 | 5 | 2 | 0 | 6 |
| 7 | 8 | 8 | 1 | 7 | 8 |
| 8 | 2 | 3 | 5 | 2 | 1 |
| 9 | 9 | 3 | 7 | 6 | 5 |

이제 기존의 데이터프레임과 같은 형태로 돌아왔습니다. 이렇게 데이터프레임과 넘파이 배열은 상호 변환이 가능합니다. 함수 혹은 목적에 따라 다른 형태의 데이터가 필요하니 숙지해두는 것이 좋습니다.

### 3.2.2 배열 탐색(인덱싱)

넘파이 배열에서도 [ ]을 사용해 인덱싱할 수 있습니다. 바로 앞에서 사용한 sample\_np를 가지고 인덱싱 연습을 해봅시다. 우선 sample\_np가 어떻게 생겼는지 한 번 더 확인하겠습니다.

| sample\_np |
| --- |

array([[2, 2, 1, 4, 3],

[4, 3, 3, 7, 1],

[5, 4, 6, 3, 5],

[1, 4, 5, 6, 7],

[4, 5, 7, 8, 3],

[5, 4, 8, 9, 4],

[7, 5, 2, 0, 6],

[8, 8, 1, 7, 8],

[2, 3, 5, 2, 1],

[9, 3, 7, 6, 5]])

여기서 가장 첫 번째 행인 [2,2,1,4,3]을 인덱싱하겠습니다. [ ] 안에 행 숫자를 넣으면 간단하게 해당 행만 출력할 수 있습니다.

| sample\_np[0] |
| --- |

### array([2, 2, 1, 4, 3])

이번에는 위의 결과에서 세 번째 숫자에 해당하는 1을 인덱싱하겠습니다. 즉 첫 번째 행의 세 번째 숫자를 인덱싱하는 것으로, 행과 열을 동시에 인덱싱하는 방법입니다. 동일하게 [ ]를 사용하되, [행번호, 열번호] 순으로 기입해줍니다(파이썬에서는 1이 아닌 0부터 시작하는 점 유의하세요).

| sample\_np[0,2] |
| --- |

1

왜 1인지는 다음 그림을 살펴보면 금방 알 수 있습니다.



하나의 행, 하나의 열이 아닌 다수의 행과 열도 인덱싱할 수 있습니다. [ ]에 :를 사용해 시작번호와 끝번호를 입력해주시되, 끝번호는 인덱싱에 포함이 안 되는 점 주의해야 합니다.

즉 행번호를 0:3으로 인덱싱하면 3은 포함이 안 되는 0행, 1행, 2행이 인덱싱됩니다. 그럼 0~2행과 2~3열을 인덱싱하겠습니다.

| sample\_np[0:3,2:4] |
| --- |

array([[1, 4],

[3, 7],

[6, 3]])

특정 열만 인덱싱하려면 어떻게 할까요? 위의 방법에서 : 앞뒤를 빈값으로 넣어주면 모든 행/열이 출력됩니다. 이 방법으로 모든 행에 대한 3번째 열을 인덱싱하겠습니다.

| sample\_np[:,2] |
| --- |

array([1, 3, 6, 5, 7, 8, 2, 1, 5, 7])

### 3.2.3 배열의 연산

### 넘파이 배열에서의 기본적인 사칙연산은 아주 간단합니다. 우선 실습을 위해 다음과 같이 np\_a라는 이름의 2✕2 행렬의 배열을 만들고, 결과 출력까지 하겠습니다.

| np\_a = np.array([[1,3], [0,-2]])  np\_a |
| --- |

array([[ 1, 3],

[ 0, -2]])

그리고 여기에 일괄적으로 10을 더하겠습니다.

| np\_a + 10 |
| --- |

### array([[11, 13],

### [10, 8]])

### 각 원소에 10이 더해진 모습입니다. 나머지 사칙연산도 마찬가지입니다. 이부분은 매우 직관적이니 임의의 숫자를 넣어 테스트해보고 빠르게 넘어가겠습니다.

| np\_a - 5 |
| --- |

array([[-4, -2],

[-5, -7]])

| np\_a \* 2 |
| --- |

array([[ 2, 6],

[ 0, -4]])

| np\_a+10 / 3 |
| --- |

array([[4.33333333, 6.33333333],

[3.33333333, 1.33333333]])

이번에는 넘파이 배열끼리를 계산을 해보겠습니다. 우선 실습에 사용할 2✕2 넘파이 배열을 하나 더 만들어봅시다.

| np\_b = np.array([[1,0], [0,1]]) np\_b |
| --- |

array([[1, 0],

[0, 1]])

그럼 이제 np\_a와 np\_b에 대한 덧셈과 뺄셈을 하겠습니다.

| np\_a + np\_b |
| --- |

array([[ 2, 3],

[ 0, -1]])

| np\_a - np\_b |
| --- |

array([[ 0, 3],

[ 0, -3]])

덧셈과 뺄셈에 대해서는 직관적으로 이해가 가능합니다. 각 넘파이 배열에서 같은 자리에 있는 원소들끼리 덧셈/뺄셈이 이루어졌습니다. 그렇다면 곱셈은 어떨까요?

| np\_a \* np\_b |
| --- |

array([[ 1, 0],

[ 0, -2]])

이번에도 마찬가지로 같은 위치에 있는 원소끼리 계산되었습니다. 그런데 우리가 고등학교 수학 과정에서 배운 행렬의 곱셈은 이와는 조금 다릅니다. 행렬의 곱셈은 다음과 같은 방식으로 계산되어야 합니다.

[[a, b], [[e, f], [[a\*e + b\*g, a\*f + b\*h],

[c, d]] \* [g, h]] = [c\*e + d\*g, c\*f + d\*h]]

파이썬에서 이렇게 계산하려면 \* 가 아닌 @을 사용해 곱해야 합니다.

| np\_a @ np\_b |
| --- |

array([[ 1, 3],

[ 0, -2]])

### 3.2.4 임의의 숫자를 뽑는 random.XXX() 함수

이번에는 넘파이에서 랜덤 셀렉션과 관련된 함수들을 소개하겠습니다. 우선 지정된 범위 안에서 임의의 숫자를 뽑기 위해 random.randint()를 사용할 수 있습니다. random.randint() 안에 숫자 하나(시드값이라고 합니다)를 지정해주면 **0부터 지정된 숫자 직전**까지의 숫자 중 임의로 한 개를 불러옵니다. 랜덤하게 진행되기 때문에 실행할 때마다 매번 다른 숫자를 얻게 됩니다. 0~10 사이에서 임의의 숫자를 불러와보겠습니다.

| np.random.randint(11) |
| --- |

9

임의로 9가 선택되었습니다. 콤마를 사용해 숫자 2개를 지정해주면 시작과 끝지점을 지정할 수 있습니다. 역시 끝지점의 숫자는 포함이 안 되고 그 직전까지 반영됩니다. 그럼 50~70 사이의 숫자 중 하나를 랜덤으로 불러보겠습니다.

| np.random.randint(50, 71) |
| --- |

58

여기에 콤마를 하나 더해 여러 개의 숫자를 선택할 수 있습니다. 50~70 사이의 숫자 중 5개를 랜덤하게 선택하겠습니다.

| np.random.randint(50, 71, 5) |
| --- |

array([68, 64, 59, 53, 62])

이번에는 주어진 목록 중 랜덤으로 선택하는 방법입니다. random.choice() 함수를 쓰면 되며, 선택될 대상 리스트와 크기를 매개변수로 입력합니다. 예를 들어 [‘red’, ‘green’, ‘white’, ‘black’, ‘blue’]에서 임의의 3개를 뽑아보겠습니다.

| np.random.choice(['red', 'green','white','black','blue'],size=3) |
| --- |

array(['black', 'black', 'blue'], dtype='<U5')

black, black, blue가 뽑혔습니다. 이 함수는 복원 추출이기 때문에 같은 값들이 여러 번 등장할 수 있습니다. 만약 같은 값이 중복 추출되는 것을 원하지 않는다면 replace 매개변수를 이용하셔서 비복원 추출을 하면 됩니다.

| np.random.choice(['red', 'green','white','black','blue'],size=3, replace=False) |
| --- |

array(['black', 'white', 'red'], dtype='<U5')

### 3.2.5 그외 유용한 함수

넘파이에서 제공하는 두 가지 유용한 함수를 살펴보겠습니다.

#### 일련의 숫자를 만드는 arange() 함수

첫 번째로 arange()를 사용하면 쉽게 일련의 숫자를 만들 수 있습니다. 예를 들어 1부터 10까지의 숫자를 arange()로 만들겠습니다. 괄호 안에 시작점과 끝점을 콤마로 구분해 써주면 되고, 인덱싱에서 배운 [시작 : 끝]과 같이, 끝 지점은 포함이 안 되고 그 직전까지 불러옵니다. 즉, 10까지 숫자를 불러오려면 11을 입력해주세요.

| np.arange(1,11) |
| --- |

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

그리고 1부터 10까지 숫자 중 1,3,5,7,9와 같이 2칸씩 건너뛴 값을 얻고 싶으면, 콤마를 하나 추가해 얼만큼 건너뛸지 숫자를 입력하면 됩니다.

| np.arange(1,11,2) |
| --- |

array([1, 3, 5, 7, 9])

#### 시작점과 끝점을 균등한 간격으로 나누는 linspace() 함수

linspace() 함수는 시작점과 끝점을 균등한 간격으로 나눈 지점들을 보여줍니다. arange()와 달리 끝점에 입력되는 숫자가 포함되니 주의해주세요. 그럼 1부터 10까지의 숫자에서 같은 간격으로 나눈 4개의 지점을 찾아봅시다.

| np.linspace(1,10,4) |
| --- |

array([ 1., 4., 7., 10.])

## 학습 마무리

판다스와 넘파이는 데이터 분석에 가장 많이 쓰이는 라이브러리입니다. 판다스는 사람에 친화적인, 넘파이는 컴퓨터 계산에 더 친화적인 자료구조를 제공합니다. 앞으로 머신러닝 알고리즘을 다루면서 각 라이브러리에서 제공하는 다양한 함수를 사용해보겠습니다.

#### 핵심 요약

1. 판다스는 데이터프레임과 시리즈를 자료구조로 제공합니다. 데이터프레임과 시리즈에는 인덱스와 컬럼명이 있습니다. 데이터프레임은 여러 시리즈를 합친 형태입니다.
2. 넘파이는 배열을 자료구조로 제공합니다. 배열을 이용하면 빠른 수치 계산이 가능합니다. 배열에는 인덱스와 컬럼명이 없습니다.