

파이썬 데이터 전처리 모듈 소개

(2) Pandas

2019. 09. 16.

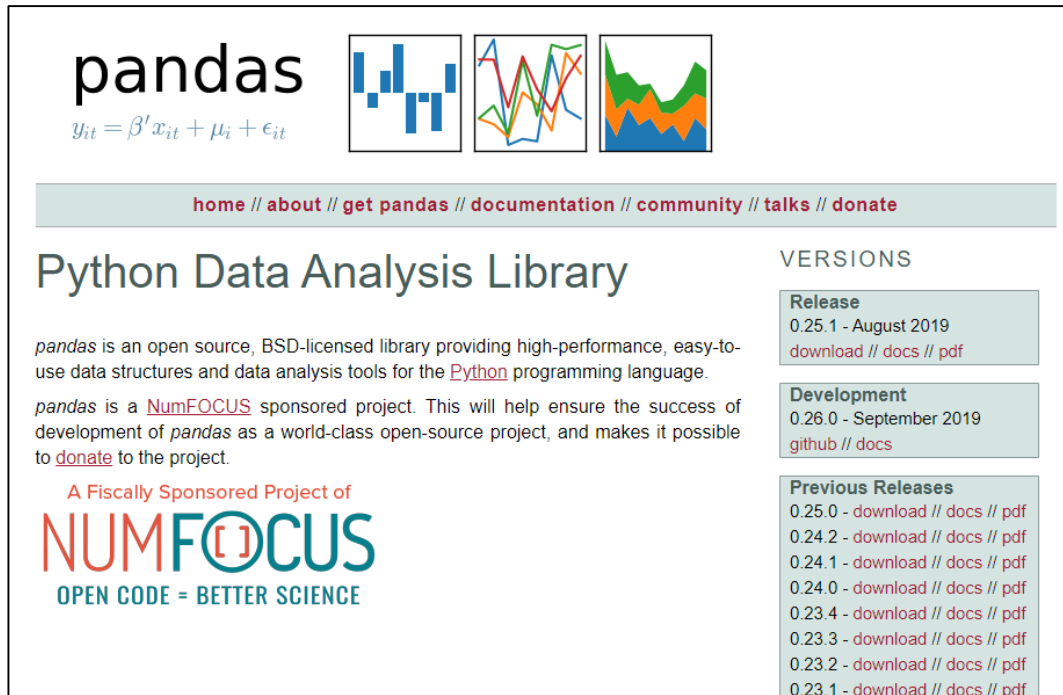
한양대학교 산업경영공학과

목차

1. Pandas 모듈 소개
2. Pandas의 객체: series, index, dataframe
3. 데이터 인덱싱과 선택
4. 결측치 처리
5. 데이터 결합

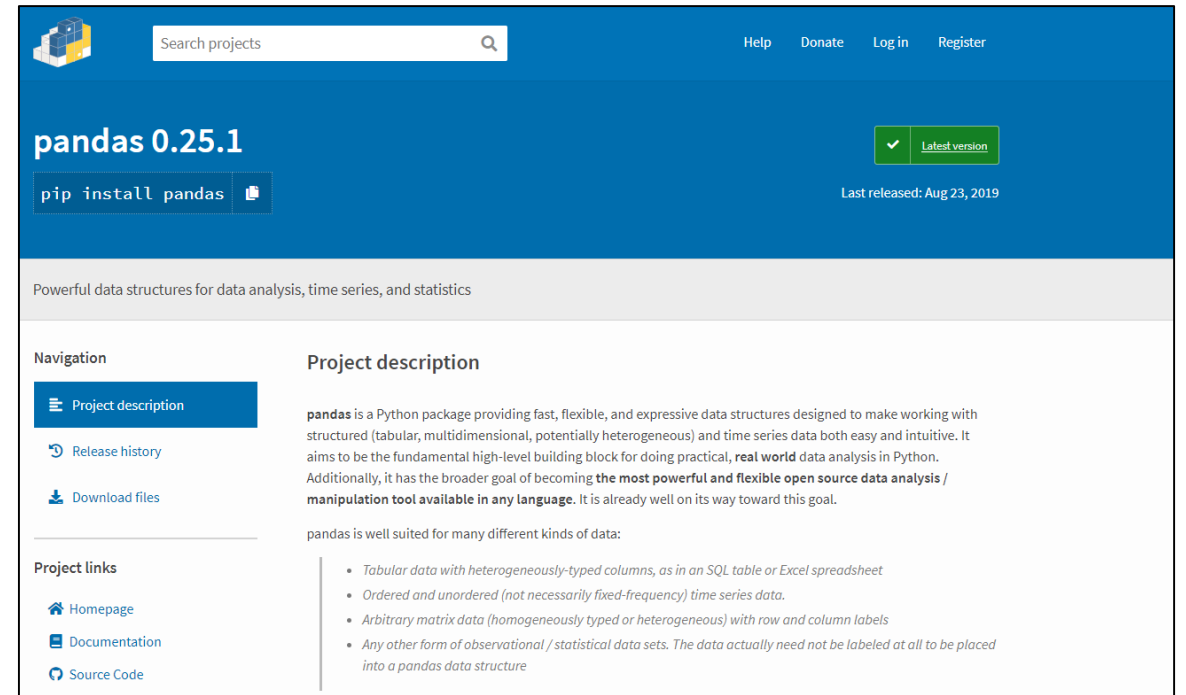
1. Pandas 모듈 소개

- 데이터 전처리를 포함한 분석 과정에서 가장 많이 활용되는 모듈



The screenshot shows the official Pandas website. At the top, the word "pandas" is displayed in a large, lowercase font, with the mathematical formula $y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$ underneath it. To the right of the text are three small icons: a bar chart, a line graph, and a stacked area chart. Below the header is a navigation bar with links: "home // about // get pandas // documentation // community // talks // donate". The main heading is "Python Data Analysis Library". Below this, there is a paragraph describing Pandas as an open source, BSD-licensed library. To the right, under the heading "VERSIONS", there are three sections: "Release" (0.25.1 - August 2019), "Development" (0.26.0 - September 2019), and "Previous Releases" (listing versions from 0.25.0 down to 0.23.1). At the bottom left, there is a logo for "NUMFOCUS" with the tagline "OPEN CODE = BETTER SCIENCE".

Pandas 홈페이지
(<http://pandas.pydata.org>)



The screenshot shows the Pandas project page on the Python Package Index (PyPI). The header is blue with a search bar and links for "Help", "Donate", "Log in", and "Register". The main heading is "pandas 0.25.1", with a green button labeled "Latest version" and a note "Last released: Aug 23, 2019". Below the heading is a button "pip install pandas". The page is divided into two main sections: "Navigation" on the left and "Project description" on the right. The "Navigation" section includes links for "Project description", "Release history", and "Download files". The "Project description" section contains a paragraph about Pandas and a list of bullet points describing the types of data it can handle: "Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet", "Ordered and unordered (not necessarily fixed-frequency) time series data", "Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels", and "Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure".

"Pypi"에 등록된 pandas 모듈
(<https://pypi.org/project/pandas/>)

1. Pandas 모듈 소개

- Pandas 모듈로 주로 하는 작업
 - 데이터 읽기(csv, excel, etc.)
 - 데이터 구성(series는 '벡터', dataframe은 '행렬')
 - 데이터 인덱싱
 - 결측치 처리
 - 데이터 결합(merge, concat)

2. Pandas의 객체: series, index, dataframe

- Pandas의 객체: Series

➤ Series는 일련의 값과 인덱스를 모두 감싸고 있으며, index 속성으로 접근 가능함

```
## 객체: series, index, dataframe
import pandas as pd
# 일반화된 numpy 배열로서의 series
# index를 주지 않으면 0, 1, 2, ...로 설정됨 (default)
data1 = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data2 = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])

# dictionary로서의 series (key가 index 역할을 함)
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
```

```
In [157]: data1
Out[157]:
a      0.25
b      0.50
c      0.75
d      1.00
dtype: float64
```

```
In [158]: data2
Out[158]:
2      0.25
5      0.50
3      0.75
7      1.00
dtype: float64
```

```
In [159]: population
Out[159]:
California      38332521
Florida         19552860
Illinois        12882135
New York        19651127
Texas           26448193
dtype: int64
```

2. Pandas의 객체: series, index, dataframe

- Pandas의 객체: DataFrame (대소문자 구분)
 - Series가 유연한 인덱스를 가지는 1차원 배열이라면, DataFrame은 유연한 행 인덱스와 열 이름을 가지는 2차원 배열이라고 볼 수 있음

```
# DataFrame 생성
area_dict = {'California': 423967, 'Texas': 695662, 'New York':
            141297, 'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
# 여러 series (population, area)의 결합으로 만드는 dataframe (states)
states = pd.DataFrame({'population': population, 'area': area})
```

In [165]: states

Out[165]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

- DataFrame은 머신러닝 모델을 학습하는데 필요한 가장 기본적인 데이터 구조임

2. Pandas의 객체: series, index, dataframe

Series와 DataFrame의 머신러닝에서의 역할

- 시리즈: 1차원 배열로, 하나의 열과 대응되는 데이터 구조

Index	Data
1	'A'
2	'B'
3	'C'
4	'D'
5	'E'

- 데이터 프레임: 다차원 배열로, 하나의 엑셀 시트와 대응되는 데이터 구조

	gdp pc 2011 ppp	1800	1801	1802	1803
0	Afghanistan	634.400014	634.400014	634.400014	634.400014
1	Albania	793.136557	793.960291	794.784880	795.610326
2	Algeria	1520.025973	1519.988511	1519.951050	1519.913589
3	Angola	650.000000	NaN	NaN	NaN
4	Antigua and Barbuda	771.878735	771.878735	771.878735	771.878735

- 시리즈는 벡터와 데이터 프레임은 행렬과 대응됨
- 따라서 시리즈 관련 연산은 벡터 연산과 대응되며, 데이터 프레임 관련 연산은 행렬 연산과 대응됨

2. Pandas의 객체: series, index, dataframe

- (Tip) 지도 학습을 위한 이상적인 데이터 구조
 - 지도 학습을 위한 이상적인 구조는 하나의 데이터 프레임(행렬)과 시리즈(열 벡터)가 결합한 형태로, ID를 제외한 모든 성분이 수치형(numerical)인 것이 바람직함

ID	x_1	x_2	x_3	Label (y)
1	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	y_1
2	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	y_2
3	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	y_3
4	$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	y_4
5	$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	y_5
6	$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	y_6

2. Pandas의 객체: series, index, dataframe

- DataFrame 객체를 구성하는 방법
 - `pd.DataFrame(Series, columns)`
 - `pd.DataFrame(dictionary list)`
 - `pd.DataFrame(numpy array, columns, index)`

```
pd.DataFrame(population, columns=['population'])
```

```
data = [{'a': i, 'b': 2 * i} for i in range(3)]  
# data = [{'a':0, 'b':0}, {'a':1, 'b':2}, {'a':2, 'b':4}]  
pd.DataFrame(data)
```

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

```
pd.DataFrame(np.random.rand(3, 2),  
             columns=['foo', 'bar'], # 열 이름  
             index=['a', 'b', 'c']) #행 이름
```

```
In [171]: pd.DataFrame(population, columns=['population'])
```

```
Out[171]:
```

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

```
In [174]: pd.DataFrame(data)
```

```
Out[174]:
```

	a	b
0	0	0
1	1	2
2	2	4

```
In [180]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

```
Out[180]:
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

<Dictionary list to DataFrame>
columns = key들의 합 집합

```
In [181]: pd.DataFrame(np.random.rand(3, 2),  
                       ...,  
                       columns=['foo', 'bar'],  
                       ...,  
                       index=['a', 'b', 'c'])
```

```
Out[181]:
```

	foo	bar
a	0.764338	0.520490
b	0.796853	0.307196
c	0.630002	0.500035

2. Pandas의 객체: series, index, dataframe

- Pandas의 객체: Index

➤ Index 객체는 불변의 배열이나 정렬된 집합과 같이 작동함

```
# Index 객체
ind = pd.Index([2, 3, 5, 7, 11])
ind[1] # 배열에서 제공하는 인덱싱이 가능
ind[::2] # 배열에서 제공하는 슬라이싱 역시 가능
ind[1] = 0 # Type error 발생 (요소 변경 불가)
```

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
indA&indB # 집합 연산이 가능함
indA|indB
```

```
In [184]: ind[1] # 배열에서 제공하는 인덱싱이 가능
Out[184]: 3
```

```
In [185]: ind[::2] # 배열에서 제공하는 슬라이싱 역시 가능
Out[185]: Int64Index([2, 5, 11], dtype='int64')
```

```
In [186]: ind[1] = 0 # Type error 발생
Traceback (most recent call last):
```

```
File "<ipython-input-186-b54356fec5c8>", line 1, in <module>
    ind[1] = 0 # Type error 발생
```

```
In [188]: indA&indB # 집합 연산이 가능함
Out[188]: Int64Index([3, 5, 7], dtype='int64')
```

```
In [189]: indA|indB
Out[189]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

3. 데이터 인덱싱과 선택

- Key를 이용한 Series에서 데이터 인덱싱
 - Series 객체는 딕셔너리와 마찬가지로 키의 집합을 값의 집합에 매핑함

```
## 데이터 인덱싱과 선택
```

```
import pandas as pd
```

```
# Key를 이용한 Series에서 데이터 선택
```

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                 index=['a', 'b', 'c', 'd'])
```

```
data['b'] # index 'b'의 값을 가져오기
```

```
'a' in data # 'a'가 data에 있는지 확인
```

```
data.keys() # data의 key (index) 가져오기
```

```
list(data.items()) # data의 key - value 가져오기
```

```
data['e'] = 1.25 # 값 치환
```

```
data['d'] = 9.9999
```

```
In [192]: data['b']
```

```
Out[192]: 0.5
```

```
In [193]: 'a' in data
```

```
Out[193]: True
```

```
In [194]: data.keys()
```

```
Out[194]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [195]: data.items()
```

```
Out[195]: <zip at 0x23c00273b48>
```

```
In [196]: list(data.items())
```

```
Out[196]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

```
In [197]: data['e'] = 1.25
```

```
In [198]: data['d'] = 9.9999
```

```
In [199]: data
```

```
Out[199]:
```

```
a    0.2500
```

```
b    0.5000
```

```
c    0.7500
```

```
d    9.9999
```

```
e    1.2500
```

```
dtype: float64
```

3. 데이터 인덱싱과 선택

- Key를 이용한 Series에서 데이터 슬라이싱

```
data['a':'c'] # explicit한 인덱스 사용  
data[0:2] # implicit한 인덱스 사용  
data[(data > 0.3) & (data < 0.8)]  
data[['a', 'e']] # 팬시 인덱싱  
data[[3, 2]]
```

```
In [201]: data['a':'c']  
Out[201]:  
a    0.25  
b    0.50  
c    0.75  
Name: 0, dtype: float64  
  
In [202]: data[0:2]  
Out[202]:  
a    0.25  
b    0.50  
Name: 0, dtype: float64  
  
In [203]: data[(data > 0.3) & (data < 0.8)]  
Out[203]:  
b    0.50  
c    0.75  
Name: 0, dtype: float64  
  
In [204]: data[['a', 'e']]  
Out[204]:  
a    0.25  
e    1.25  
Name: 0, dtype: float64  
  
In [205]: data[[3, 2]]  
Out[205]:  
d    9.9999  
c    0.7500  
Name: 0, dtype: float64
```

3. 데이터 인덱싱과 선택

loc 인덱서와 iloc 인덱서

- `data = pd.Series(['a', 'b', 'c'], index = [1, 3, 5])`라고 정의된 Series
 - `data[1]`은 'b'를 나타내는 것일까 (파이썬 암묵적 인덱스)? 아니면 'a'를 나타내는 것일까 (명시적인 인덱스)
 - 정답은 'a'이지만 (명시적인 인덱스 우선), 혼동이 충분히 일어날 수 있음
 - 다양한 인덱서가 필요한 이유: 명시적인 정수 인덱스가 포함되면 혼동을 일으킬 수 있음
- loc 인덱서: **명시적인 인덱스**를 참조하는 인덱싱과 슬라이싱을 가능하게 함
- iloc 인덱서: **암묵적인 인덱스**를 참조하는 인덱싱과 슬라이싱을 가능하게 함

#인덱서: loc, iloc

`data = pd.Series(['a', 'b', 'c'], index = [1, 3, 5])`

`data.iloc[1]` # 위치가 1인 값 ('b')

`data.iloc[1:3]` # 위치가 1부터 3까지인 값 ('b', 'c')

`data.loc[3]` # 인덱스가 3인 값

`data.loc[3:5]` # 인덱스가 3부터 5까지인 값

```
In [208]: data.iloc[1]
Out[208]: 'b'

In [209]: data.iloc[1:3]
Out[209]:
3      b
5      c
dtype: object

In [210]: data.loc[3]
Out[210]: 'b'

In [211]: data.loc[3:5]
Out[211]:
3      b
5      c
dtype: object
```

3. 데이터 인덱싱과 선택

DataFrame에서 데이터 선택

- DataFrame은 키로 접근도 가능하고, 속성으로도 접근 가능함
 - `data['column']`: data를 딕셔너리로 보고, 'area'라는 키를 사용하는 경우
 - `data.column`: area를 data의 속성으로 보는 경우
- (주의) data 속성으로 보는 경우, 속성이 다른 메서드와 이름이 겹치는 경우에는 사용할 수 없음
 - (예시) data에 pop이라는 열이 포함되더라도, `data.pop`은 pop 열이 아니라 `pop()` 메서드를 호출하게 됨
- 위 접근 방법은 객체를 변경할 때도 사용할 수 있음

DataFrame에서 데이터 선택

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data['density'] = data['pop'] / data['area'] #새로운 열 생성
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

3. 데이터 인덱싱과 선택

DataFrame의 인덱싱

- 인덱싱은 열을 참조하지만, 슬라이싱은 행을 참조한다!
- 마스킹 연산은 열 단위가 아닌 행 단위로 해석된다.

중요 팁! 데이터 프레임의 참조

data['Florida':'Illinois'] # 행 단위 슬라이싱 연산

data['Florida'] # 인덱싱 연산 - 오류 발생!

data['area':'density'] # 슬라이싱 연산 - 이상한 결과

data['area'] # 열 단위 인덱싱

data[data.density>100] # density 열이 100보다 큰 값만 가져오기

```
In [230]: data['Florida':'Illinois']
```

```
Out[230]:
```

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

```
In [231]: data['area']
```

```
Out[231]:
```

California	423967
Florida	170312
Illinois	149995
New York	141297
Texas	695662

```
Name: area, dtype: int64
```

```
In [232]: data[data.density>100]
```

```
Out[232]:
```

	area	pop	density
Florida	170312	19552860	114.806121
New York	141297	19651127	139.076746

4. 결측치 처리

Pandas 모듈에서 정의한 결측치의 종류

- NaN(Not a Number)은 표현 불가능한 수치형 값이라는 뜻이지만, 대부분의 결측값은 NaN으로 처리됨
 - Pandas에서는 NaN을 default missing value marker로 쓰고 있음
- None: 함수의 출력 값(return)이 정의되지 않은 경우에 None이 출력됨

4. 결측치 처리

- NaN 연산

- NaN은 객체 배열과는 달리 연산을 지원하며, 부동 소수점 값으로 처리됨
- 단, NaN과 연산한 결과는 다시 NaN이 되어버림

```
# NaN: 누락된 숫자 데이터
vals = np.array([1, np.nan, 3, 4])
sum(vals)
1 + np.nan
0 * np.nan

np.nansum(vals)
np.nanmin(vals)
```

```
In [243]: sum(vals)
```

```
Out[243]: nan
```

```
In [244]: 1 + np.nan
```

```
Out[244]: nan
```

```
In [245]: 0 * np.nan
```

```
Out[245]: nan
```

```
In [246]: np.nansum(vals)
```

```
Out[246]: 8.0
```

```
In [247]: np.nanmin(vals)
```

```
Out[247]: 1.0
```

4. 결측치 처리

- NaN과 None

- None값이 배열에 포함된 경우
 - 자동으로 NaN 값으로 변환됨(NaN의 데이터 타입은 'float64')

```
# Pandas에서 NaN과 None
```

```
pd.Series([1, np.nan, 2, None])
```

```
x = pd.Series([0, 1], dtype=int)
```

```
x.dtype
```

```
x[0] = None
```

```
x.dtype
```

```
In [255]: pd.Series([1, np.nan, 2, None])
```

```
Out[255]:
```

```
0      1.0
```

```
1      NaN
```

```
2      2.0
```

```
3      NaN
```

```
dtype: float64
```

```
In [256]: x.dtype
```

```
Out[256]: dtype('int32')
```

```
In [257]: x[0] = None
```

```
In [258]: x.dtype
```

```
Out[258]: dtype('float64')
```

4. 결측치 처리

Null값 연산하기

- Pandas 데이터 구조의 널 값을 감지하고 삭제하고 대체하는 메서드
 - isnull(): 누락 값을 가리키는 부울 마스크 생성
 - notnull(): isnull의 반대인 부울 마스크 생성
 - dropna(): 누락 값을 제거한 데이터를 반환
 - fillna(x): 누락 값을 x로 채운 데이터 사본을 반환 (보통 어떤 값을 채울까?)

Null 값 연산 - Series

```
data = pd.Series([1, np.nan, 'hello', None])  
data.isnull()  
data.notnull()  
data.dropna()  
data.fillna(0) # 누락 값을 0으로 채우기
```

```
In [271]: data.isnull()  
Out[271]:  
0    False  
1     True  
2    False  
3     True  
dtype: bool
```

```
In [272]: data.notnull()  
Out[272]:  
0     True  
1    False  
2     True  
3    False  
dtype: bool
```

```
In [273]: data.dropna()  
Out[273]:  
0      1  
2  hello  
dtype: object
```

```
In [274]: data.fillna(0)  
Out[274]:  
0      1  
1      0  
2  hello  
3      0  
dtype: object
```

4. 결측치 처리

Null값 연산하기

Null 값 연산 - DataFrame

```
df = pd.DataFrame([[np.nan, np.nan, np.nan],  
[2, 3, 5], [np.nan, 4, 6]])
```

```
df.isnull()
```

```
df.notnull()
```

```
df.dropna(axis = 0) # nan을 포함하는 행을 삭제
```

```
df.dropna(axis = 1) # nan을 포함하는 열을 삭제
```

```
df.dropna(axis = 0, how = 'all') # 모든 요소가 nan인
```

행 삭제

```
df.fillna(0) # nan값을 0으로 채우기
```

```
df.fillna(method = 'ffill') # 이전값으로 채우기
```

```
df.fillna(method = 'bfill') # 다음값으로 채우기
```

```
In [283]: df.isnull()  
Out[283]:
```

	0	1	2
0	True	True	True
1	False	False	False
2	True	False	False

```
In [284]: df.notnull()  
Out[284]:
```

	0	1	2
0	False	False	False
1	True	True	True
2	False	True	True

```
In [285]: df.dropna(axis = 0)  
Out[285]:
```

	0	1	2
1	2.0	3.0	5.0

```
In [286]: df.dropna(axis = 1)  
Out[286]:  
Empty DataFrame  
Columns: []  
Index: [0, 1, 2]
```

```
In [287]: df.dropna(axis = 0, how = 'all')  
Out[287]:
```

	0	1	2
1	2.0	3.0	5.0
2	NaN	4.0	6.0

```
In [288]: df.fillna(0)  
Out[288]:
```

	0	1	2
0	0.0	0.0	0.0
1	2.0	3.0	5.0
2	0.0	4.0	6.0

```
In [289]: df.fillna(method = 'ffill') # 이전값으로 채우기  
Out[289]:
```

	0	1	2
0	NaN	NaN	NaN
1	2.0	3.0	5.0
2	2.0	4.0	6.0

```
In [290]: df.fillna(method = 'bfill') # 다음값으로 채우기  
Out[290]:
```

	0	1	2
0	2.0	3.0	5.0
1	2.0	3.0	5.0
2	NaN	4.0	6.0

5. 데이터 결합

pd.concat 함수

- `concat([series1, series2, ...])`, `concat([df1, df2, ...])`: series(df)를 결합하는데 사용
- 기본적으로는 행 단위 결합이 일어남 (default 설정)

데이터 결합

Concat 개요

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
```

```
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
```

```
pd.concat([ser1, ser2]) # 시리즈 결합
```

```
df1 = pd.DataFrame({'A': ['A1', 'A2'], 'B': ['B1', 'B2']}, index = [0, 1])
```

```
df2 = pd.DataFrame({'A': ['A1', 'A2'], 'B': ['B1', 'B2']}, index = [2, 3])
```

```
pd.concat([df1, df2]) # 데이터 프레임 결합 (수직 스택)
```

```
In [293]: pd.concat([ser1, ser2])
```

```
Out[293]:
```

```
1      A
```

```
2      B
```

```
3      C
```

```
4      D
```

```
5      E
```

```
6      F
```

```
In [304]: pd.concat([df1, df2])
```

```
Out[304]:
```

```
      A      B
```

```
0  A1    B1
```

```
1  A2    B2
```

```
2  A1    B1
```

```
3  A2    B2
```

5. 데이터 결합

pd.concat – 인덱스 복제

- np.concatenate와 pd.concat의 중요한 차이는 Pandas에서의 연결은 그 결과가 복제된 인덱스를 가지더라도 인덱스를 유지한다는데 있음

```
df2.index = df1.index  
# df2의 index를 df1의 index로 바꾸기  
pd.concat([df1, df2])
```

```
In [308]: pd.concat([df1, df2])
```

```
Out[308]:
```

	A	B
0	A1	B1
1	A2	B2
0	A1	B1
1	A2	B2

인덱스가 유지되어 있음

- 해결 방법
(1) 인덱스 무시: pd.concat([df1, df2], ignore_index = True)

	A	B
0	A1	B1
1	A2	B2
2	A1	B1
3	A2	B2

- (2) 다중인덱스 추가: pd.concat([df1, df2], keys = ['df1', 'df2'])

		A	B
df1	0	A1	B1
	1	A2	B2
df2	0	A1	B1
	1	A2	B2

5. 데이터 결합

pd.merge 함수

데이터베이스 조인을 위한 merge 함수

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],  
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
```

```
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],  
                    'hire_date': [2004, 2008, 2012, 2014]})
```

```
df3 = pd.merge(df1, df2)
```

In [316]: df1

Out[316]:

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

In [317]: df2

Out[317]:

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014



In [315]: df3

Out[315]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

5. 데이터 결합

pd.merge - 병합 키 지정

- `pd.merge()`는 두 개의 데이터 집합 사이에 일치하는 하나 이상의 열 이름을 찾아 그것을 키로 사용함.
그러나 열 이름이 일치하지 않는 경우는 혼치 않거나, 키로 사용하면 안되는 열이 존재할 수 도 있음
- 따라서 `on` 키워드 혹은 `index` 키워드를 사용함
 - 키 열 이름이 같은 경우: `on` 키워드 사용
 - 키 열 이름이 다른 경우: `left_on` & `right_on` 키워드 사용
 - 인덱스 이름이 같은 경우: `left_index` & `right_index` 키워드 사용

5. 데이터 결합

키 열 이름이 같은 경우: one 키워드 사용

- `pd.merge(df1, df2, on = [key list])`

on 키워드의 사용

`pd.merge(df1, df2, on = ['employee'])` # employee 열을 기준으로 결합

In [332]: df1

Out[332]:

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

In [333]: df2

Out[333]:

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

In [334]: `pd.merge(df1, df2, on = 'employee')`

Out[334]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

5. 데이터 결합

키 열 이름이 다른 경우: left on & right on 키워드 사용

- `pd.merge(df1, df2, left_on = [key list], right_on = [key list])`

left_on & right_on 키워드의 사용

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],  
                    'salary': [70000, 80000, 120000, 90000]})
```

```
pd.merge(df1, df3, left_on = "employee", right_on = "name") # employee (=name)으로 결합
```

In [338]: df1

Out[338]:

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

In [339]: df3

Out[339]:

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

In [340]: `pd.merge(df1, df3, left_on = "employee", right_on = "name")`

Out[340]:

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

일반적으로는
둘 중 하나를
drop해줘야 함

`df.drop(columns =
['name'])`

5. 데이터 결합

인덱스 이름이 같은 경우: left index & right index 키워드 사용

- `pd.merge(df1, df2, left_index = True, right_index = True)`

left_index & right_index 키워드의 사용

```
pd.merge(df1, df3, left_index = True, right_index = True)
```

In [338]: df1

Out[338]:

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

In [339]: df3

Out[339]:

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

In [340]: `pd.merge(df1, df3, left_index = True, right_index = True)`

Out[340]:

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

6. 데이터 프레임 객체 구성하기

- [예제1](#). np.array 포맷의 데이터를 데이터 프레임 객체로 정의하기
- [예제2](#). csv 파일로 저장된 데이터를 데이터 프레임 객체로 정의하기