

DANDYRILLA

# 판다스(pandas) 기본 사용법 익히기

12 AUG 2017

python pandas 10min

본 글은 판다스(pandas)의 기본 사용법을 소개해 놓은 [10 Minutes to pandas](#) 을 번역한 내용입니다. 이에 덧붙여 직접 실습을 해 보면서 조금 더 자세한 설명이 필요한 부분을 추가하였습니다. 그러다 보니 원글의 제목과 달리 이를 10분만에 읽어 보기는 쉽지는 않지만, 차근차근 실습을 해 보면서 pandas 의 기본 사용법을 익히시려는 분들께 많은 도움이 되었으면 좋겠습니다.

## 목차

- 시작하기에 앞서
- 1. 데이터 오브젝트 생성하기
- 2. 데이터 확인하기 (Viewing Data)
- 3. 데이터 선택하기 (Selection)
- 4. 결측치 (Missing Data)
- 5. 연산 (Operations)
- 6. 합치기 (Merging)
- 7. 묶기 (Grouping)
- 8. 변형하기 (Reshaping)
- 9. 시계열 데이터 다루기 (Time Series)
- 10. 범주형 데이터 다루기 (Categoricals)
- 11. 그래프로 표현하기 (Plotting)
- 12. 데이터 입/출력 (Getting Data In/Out)

## 시작하기에 앞서

pandas 를 사용하기 위해서 다음과 같이 모듈을 임포트(import) 합니다. 임포트를 할 때에는 pandas 라는 네임스페이스를 그대로 사용해도 되지만 간결성을 위해 pd 라는 축약된 이름을 관례적으로 많이 사용합니다. 본 실습을 진행하기 위해 pandas 외에도 배열 구조나 랜덤 값 생성 등의 기능을 활용하기 위한 numpy 와 그래프를 그리기 위한 matplotlib 패키지도 함께 import 해줍니다.

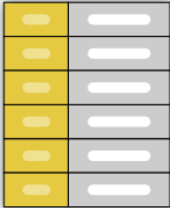
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

## 1. 데이터 오브젝트 생성하기

데이터 오브젝트는 ‘데이터를 담고 있는 그릇’이라고 생각하시면 쉬운데요. 여러분이 pandas 에서 자주 사용하시게 될 데이터 오브젝트는 Series 와 DataFrame 이 있습니다. 이 두 종류의 데이터 오브젝트를 잘 이해하고 사용하는 것이 pandas 의 전 부라고 해도 과언이 아닐 정도로 중요합니다. 그렇다면 이 두 종류의 ‘그릇’의 차이점은 무엇일까요? 바로 데이터를 담는 그릇의 ‘형태’가 다른데요. 쉽게 말하자면 Series 는 1차원 배열로, DataFrame 은 2차원 배열로 데이터를 담고 있다고 생각하시면 됩니다. 이번 섹션에서는 Series 와 DataFrame 이라는 데이터 오브젝트를 만들어 보는 실습을 해 보겠습니다.

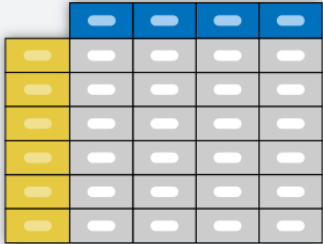
### Pandas 에서 사용되는 대표적인 데이터 오브젝트

#### 시리즈 (Series)



Series 는 1차원 배열의 형태를 갖는다. 인덱스(노란색)라는 한 가지 기준에 의하여 데이터가 저장된다.

#### 데이터프레임 (DataFrame)



DataFrame 은 2차원 배열의 형태를 갖는다. 인덱스(노란색)와 컬럼(파란색)이라는 두 가지 기준에 의하여 표 형태처럼 데이터가 저장된다.

dandyrilla.github.io

Pandas 의 중요한 데이터 오브젝트 중 하나인 **Series**는 기본적으로 아래와 같이 값의 리스트를 넘겨주어 만들 수 있습니다. 또한 값이 위치하고 있는 정보인 인덱스(index)가 Series 에 같이 저장되게 되는데요. 따로 전달해주지 않는 한 기본적으로 0 부터 시작하여 1씩 증가하는 정수 인덱스가 사용됨을 알 수 있습니다.

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

```
# 0    1.0
# 1    3.0
# 2    5.0
# 3    NaN
# 4    6.0
# 5    8.0
# dtype: float64
```

컬럼명을 이용하여 데이터 선택하기 (컬럼 선택)

df.A 또는 df['A']

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

특정 ‘행 범위’를 가져오고 싶다면 다음과 같이 리스트를 슬라이싱 할 때와 같이 `[]` 를 이용할 수 있습니다. `df[0:3]` 라고 하면 0, 1, 2번째 행을 가져옵니다(데이터프레임의 첫번째 행을 0번째 행이라고 가정). `[0:3]` 이라고 입력했지만 3번째 행을 가져오지 않음에 유의합니다. 또 다른 방법으로 `df['20130102':'20130104']` 인덱스명을 직접 넣어서 해당하는 ‘행 범위’를 가져올 수도 있습니다. 이 때에는 숫자를 이용하여 슬라이싱 할 때와 달리 처음과 끝의 행이 모두 포함된 결과를 가져옵니다.

```
## 맨 처음 3개의 행을 가져옵니다.
df[0:3]
#           A           B           C           D
# 2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
# 2013-01-02  1.212112 -0.173215  0.119209 -1.044236
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804

## 인덱스명에 해당하는 값들을 가져옵니다.
df['20130102':'20130104']
#           A           B           C           D
# 2013-01-02  1.212112 -0.173215  0.119209 -1.044236
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
# 2013-01-04  0.721555 -0.706771 -1.039575  0.271860
```

특정 행 범위의 데이터 선택하기 (행 선택)

df[0:3]

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

df['2013-01-02':'2013-01-04']

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

여기서 제가 왜 특정 ‘행’이 아니라 ‘행 범위’라고 강조하였는지를 설명드리겠습니다. 만약 특정 행 하나를 가져오고 싶은 경우에 `df['20130102']` 라고 하면 `KeyError` 가 발생합니다. 왜일까요? 이 때에는 ‘20130102’라는 이름의 ‘인덱스’가 아니라 ‘컬럼’을 갖고 있는지 찾게 됩니다. 따라서 현재 데이터프레임에는 없으므로 키 값이 없다는 에러를 출력하게 되는 것입니다. 특정 ‘행 하나’를 선택하고 싶을 때에는 `df['20130102':'20130102']` 와 같이 입력하면 됩니다. 다시 정리하자면, 데이터프레임 자체가 갖고 있는 슬라이싱은 `df[컬럼명]`, `df[시작인덱스:끝인덱스+1]`, `df[시작인덱스명:끝인덱스명]` 의 형태로 사용할 수 있습니다.

이름을 이용하여 선택하기: `.loc`

라벨의 이름을 이용하여 선택할 수 있는 `.loc` 를 이용할 수도 있습니다.

첫 번째 인덱스의 값인 ‘2013-01-01’에 해당하는 모든 컬럼의 값 가져오기. `df.loc[dates[0]]` 외에도 `df.loc['20130101']` 또는 `df.loc['2013-01-01']` 처럼 날짜를 직접 입력해도 잘 작동합니다.

```
df.loc[dates[0]]
# A      0.469112
# B     -0.282863
# C     -1.509059
# D     -1.135632
# Name: 2013-01-01 00:00:00, dtype: float64
```

행 이름을 이용하여 데이터 선택하기 (행 선택)

df.loc['20130101']

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

컬럼 'A'와 컬럼 'B'에 대한 모든 값 가져오기.

```
df.loc[:,['A','B']]
#           A           B
# 2013-01-01  0.469112 -0.282863
# 2013-01-02  1.212112 -0.173215
# 2013-01-03 -0.861849 -2.104569
# 2013-01-04  0.721555 -0.706771
# 2013-01-05 -0.424972  0.567020
# 2013-01-06 -0.673690  0.113648
```

여러 컬럼명을 이용하여 데이터 선택하기 (컬럼 선택)

df.loc[:, ['A', 'B']]

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

인덱스 '2013-01-02'부터 '2013-01-04'까지의 컬럼 'A'와 컬럼 'B'의 값 가져오기.

```
df.loc['20130102':'20130104',['A','B']]
#           A           B
# 2013-01-02  1.212112 -0.173215
# 2013-01-03 -0.861849 -2.104569
# 2013-01-04  0.721555 -0.706771
```

행과 컬럼 조건에 맞는 데이터 선택하기 (1)

df.loc['20130102':'20130104', ['A', 'B']]

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

특정 인덱스 값의 컬럼 'A', 'B' 값을 가져오기.

```
df.loc[dates[0], ['A','B']]
# A    1.212112
# B   -0.173215
# Name: 2013-01-02 00:00:00, dtype: float64
```

행과 컬럼 조건에 맞는 데이터 선택하기 (2)

df.loc['20130101', ['A', 'B']]

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

특정 인덱스 값과 특정 컬럼에 있는 값 가져오기. 이는 .at 을 이용할 수도 있습니다.

```
df.loc[dates[0], 'A']
# 0.46911229990718628

df.at[dates[0], 'A']
# 0.46911229990718628
```

특정 행과 특정 컬럼에 있는 데이터 선택하기

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

df.loc['20130101', 'A']  
또는 df.at['20130101', 'A']

dandyrilla.github.io

위치를 이용하여 선택하기: .iloc

다음과 같이 위치를 나타내는 인덱스 번호를 이용하여 데이터를 선택할 수 있습니다. 여기서 인덱스 번호는 python 에서 사용하는 인덱스와 같은 개념으로 이해하시면 됩니다. 인덱스 번호는 0 부터 시작하므로, 첫 번째 데이터는 인덱스 번호가 0 이고, 두 번째 데이터는 인덱스 번호가 1 이라는 뜻입니다. 아래는 인덱스 번호 3 (네 번째 행)을 선택하는 예제입니다.

```
df.iloc[3]
# A      0.721555
# B     -0.706771
# C     -1.039575
# D      0.271860
# Name: 2013-01-04 00:00:00, dtype: float64
```

위치를 기준으로 데이터 선택하기 (1)

행 번호		A	B	C	D
0	2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
1	2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2	2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
3	2013-01-04	0.721555	-0.706771	-1.039575	0.271860
4	2013-01-05	-0.424972	0.567020	0.276232	-1.087401
5	2013-01-06	-0.673690	0.113648	-1.478427	0.524988

df.iloc[3]

dandyrilla.github.io

인덱스 번호로 행 뿐만 아니라 열도 선택할 수 있습니다. 또한 numpy 나 python 의 슬라이싱 기능과 비슷하게 사용할 수 있습니다. 아래는 행과 열의 인덱스를 기준으로 이용하여 데이터를 선택하는 예제입니다. 행의 인덱스는 3:5 로 네 번째 행과 다섯 번째 행을 선택하며, 열의 인덱스는 0:2 로 첫 번째 열과 두 번째 열을 선택합니다.

```
df.iloc[3:5,0:2]
#              A      B
# 2013-01-04  0.721555 -0.706771
# 2013-01-05 -0.424972  0.567020
```

위치를 기준으로 데이터 선택하기 (2)

열 번호	0	1	2	3	
행 번호	A	B	C	D	
0	2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
1	2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2	2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
3	2013-01-04	0.721555	-0.706771	-1.039575	0.271860
4	2013-01-05	-0.424972	0.567020	0.276232	-1.087401
5	2013-01-06	-0.673690	0.113648	-1.478427	0.524988

df.iloc[3:5,0:2]

dandyrilla.github.io

또한 행과 열의 인덱스를 리스트로 넘겨줄 수도 있습니다. 다음은 두 번째, 세 번째, 다섯 번째 행과, 첫 번째와 세 번째 열을 선택하는 예제입니다.

```
df.iloc[[1,2,4],[0,2]]
#              A      C
# 2013-01-02  1.212112  0.119209
# 2013-01-03 -0.861849 -0.494929
# 2013-01-05 -0.424972  0.276232
```

위치를 기준으로 데이터 선택하기 (3)

열 번호	0	1	2	3	
행 번호	A	B	C	D	
0	2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
1	2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2	2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
3	2013-01-04	0.721555	-0.706771	-1.039575	0.271860
4	2013-01-05	-0.424972	0.567020	0.276232	-1.087401
5	2013-01-06	-0.673690	0.113648	-1.478427	0.524988

df.iloc[[1,2,4],[0,2]]

dandyrilla.github.io

명시적으로 행이나 열 선택 인자에 : 슬라이스를 전달하면 다음과 같이 행 또는 열 전체를 가져올 수도 있습니다.

```
df.iloc[1:3,: ]
#              A      B      C      D
# 2013-01-02  1.212112 -0.173215  0.119209 -1.044236
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804

df.iloc[:,1:3]
#              B      C
# 2013-01-01 -0.282863 -1.509059
# 2013-01-02 -0.173215  0.119209
# 2013-01-03 -2.104569 -0.494929
# 2013-01-04 -0.706771 -1.039575
```

```
# 2013-01-05    0.567020    0.276232
# 2013-01-06    0.113648   -1.478427
```

위치를 기준으로 데이터 선택하기 (4)

열 번호

0

1

2

3

행 번호

A

B

C

D

0

2013-01-01

0.469112

-0.282863

-1.509059

-1.135632

1

2013-01-02

1.212112

-0.173215

0.119209

-1.044236

2

2013-01-03

-0.861849

-2.104569

-0.494929

1.071804

3

2013-01-04

0.721555

df.iloc[1:3,:]

0.271860

4

2013-01-05

-0.424972

0.567020

0.276232

-1.087401

5

2013-01-06

-0.673690

0.113648

-1.478427

0.524988

열 번호

0

1

2

3

행 번호

A

B

C

D

0

2013-01-01

0.469112

-0.282863

-1.509059

-1.135632

1

2013-01-02

1.212112

-0.173215

0.119209

-1.044236

2

2013-01-03

-0.861849

-2.104569

-0.494929

1.071804

3

2013-01-04

0.721555

-0.706771

-1.039575

0.271860

4

2013-01-05

-0.424972

0.567020

0.276232

-1.087401

5

2013-01-06

-0.673690

0.113648

-1.478427

0.524988

df.iloc[:, 1:3]

dandyrilla.github.io

값 하나를 선택하기 위해서는 특정 행과 열을 지정하는 방식으로 하면 됩니다. 아래의 두 방법 모두 동일한 방법입니다.

```
df.iloc[1,1]
# -0.17321464905330858
df.iat[1,1]
# -0.17321464905330858
```

위치를 기준으로 데이터 선택하기 (5)

		열 번호	0	1	2	3
행 번호						
		A	B	C	D	
0	2013-01-01	0.469112	-0.282863	-1.509059	-1.135632	
1	2013-01-02	1.212112	-0.173215	0.119209	-1.044236	
2	2013-01-03	-0.861849	df.iloc[1,1] 또는 df.iat[1,1]		1.071804	
3	2013-01-04	0.721555	-0.706771	-1.039575	0.271860	
4	2013-01-05	-0.424972	0.567020	0.276232	-1.087401	
5	2013-01-06	-0.673690	0.113648	-1.478427	0.524988	

dandyrilla.github.io

이 부분에 대해 더 많은 사용법을 원하신다면 [Indexing and selecting data](#) 을 참고하시면 됩니다.

조건을 이용하여 선택하기

특정한 열의 값들을 기준으로 조건을 만들어 해당 조건에 만족하는 행들만 선택할 수 있는 방법이 있습니다. 다음은 A라는 열에 들어있는 값이 양수인 경우에 해당하는 행들을 선택하는 예제입니다.

```
df[df.A > 0]
#
# 2013-01-01    0.469112   -0.282863   -1.509059   -1.135632
# 2013-01-02    1.212112   -0.173215    0.119209   -1.044236
# 2013-01-04    0.721555   -0.706771   -1.039575    0.271860
```

조건을 이용하여 데이터 선택하기 (행 선택)

df[df.A > 0]

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

dandyrilla.github.io

또한 각 값을 기준으로 조건을 만들 수도 있습니다. 이 때에는 행이 선택되는 것이 아니라 데이터 프레임의 전체 모양은 유지된 채로 조건에 맞는 값들만 그대로 보여지고 나머지 값들은 추후에 배울 결측치(missing value)로 나타나게 됩니다. 다음은 값이 양수인 것들만 보여지고 나머지 값들(0 혹은 음수)은 NaN 으로 보여지는 예제입니다.

```
df[df > 0]
#
# 2013-01-01    0.469112      NaN      NaN      NaN
# 2013-01-02    1.212112      NaN    0.119209      NaN
# 2013-01-03      NaN      NaN      NaN    1.071804
# 2013-01-04    0.721555      NaN      NaN    0.271860
# 2013-01-05      NaN    0.567020    0.276232      NaN
# 2013-01-06      NaN    0.113648      NaN    0.524988
```

조건을 이용하여 데이터 선택하기 (개별 선택)

df[df > 0]

	A	B	C	D
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632
2013-01-02	1.212112	-0.173215	0.119209	-1.044236
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804
2013-01-04	0.721555	-0.706771	-1.039575	0.271860
2013-01-05	-0.424972	0.567020	0.276232	-1.087401
2013-01-06	-0.673690	0.113648	-1.478427	0.524988

\* 선택되지 않은 값들은 결측치(NaN)로 표현됩니다.

dandyrilla.github.io

또한 필터링을 해야 하는 경우에 사용할 수 있는 `isin()` 이라는 메소드도 제공합니다. 다음과 같이 새로운 열 하나를 추가한 후 새롭게 추가된 열에 들어있는 값을 기준으로 행을 선택할 수 있습니다.

```
df2 = df.copy()
df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']
```

df2

#	A	B	C	D	E
# 2013-01-01	0.469112	-0.282863	-1.509059	-1.135632	one
# 2013-01-02	1.212112	-0.173215	0.119209	-1.044236	one
# 2013-01-03	-0.861849	-2.104569	-0.494929	1.071804	two
# 2013-01-04	0.721555	-0.706771	-1.039575	0.271860	three
# 2013-01-05	-0.424972	0.567020	0.276232	-1.087401	four
# 2013-01-06	-0.673690	0.113648	-1.478427	0.524988	three

```
df2[df2['E'].isin(['two', 'four'])]
#           A           B           C           D           E
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804      two
# 2013-01-05 -0.424972  0.567020  0.276232 -1.087401      four
```

조건을 이용하여 데이터 선택하기 (isin 메소드로 행 선택)

df2[df2['E'].isin(['two', 'four'])]

	A	B	C	D	E
2013-01-01	0.469112	-0.282863	-1.509059	-1.135632	one
2013-01-02	1.212112	-0.173215	0.119209	-1.044236	one
2013-01-03	-0.861849	-2.104569	-0.494929	1.071804	two
2013-01-04	0.721555	-0.706771	-1.039575	0.271860	three
2013-01-05	-0.424972	0.567020	0.276232	-1.087401	four
2013-01-06	-0.673690	0.113648	-1.478427	0.524988	three

dandyrilla.github.io

데이터 변경하기

우리가 선택했던 데이터 프레임의 특정 값들을 다른 값으로 변경할 수 있습니다. 이에 대한 방법을 알아봅니다.

기존 데이터 프레임에 새로운 열을 추가하고 싶을 때는 다음과 같이 같은 인덱스를 가진 시리즈 하나를 데이터 프레임의 열 하나를 지정하여 넣어 줍니다.

```
s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range('20130102', periods=6))
```

```
s1
# 2013-01-02    1
# 2013-01-03    2
# 2013-01-04    3
# 2013-01-05    4
# 2013-01-06    5
# 2013-01-07    6
# Freq: D, dtype: int64
```

```
df['F'] = s1
```

데이터 프레임의 특정 값 하나를 선택하여 다른 값으로 바꿀 수 있습니다.

```
df.at[dates[0], 'A'] = 0
```

앞서 배운 값의 위치(인덱스 번호)를 이용한 변경도 가능합니다.

```
df.iat[0,1] = 0
```

여러 값을 한꺼번에 바꾸고 싶을 때는 데이터의 크기만 잘 맞춰 주면 됩니다. 다음은 NumPy array를 이용한 방법입니다.

```
df.loc[:, 'D'] = np.array([5] * len(df))
```

앞에서 바꾼 데이터들을 모두 적용하여 데이터 프레임의 값들을 한번 살펴보겠습니다.

```
df
#           A           B           C           D           F
# 2013-01-01  0.000000  0.000000 -1.509059    5  NaN
# 2013-01-02  1.212112 -0.173215  0.119209    5  1.0
# 2013-01-03 -0.861849 -2.104569 -0.494929    5  2.0
# 2013-01-04  0.721555 -0.706771 -1.039575    5  3.0
# 2013-01-05 -0.424972  0.567020  0.276232    5  4.0
# 2013-01-06 -0.673690  0.113648 -1.478427    5  5.0
```

데이터 변경하기

df.at[dates[0], 'A'] = 0

df.iat[0, 1] = 0

df.loc[:, 'D'] = np.array([5] \* len(df))

dff['F'] = s1

	A	B	C	D	F
2013-01-01	0	0	-1.509059	5	NaN
2013-01-02	1.212112	-0.173215	0.119209	5	1
2013-01-03	-0.861849	-2.104569	-0.494929	5	2
2013-01-04	0.721555	-0.706771	-1.039575	5	3
2013-01-05	-0.424972	0.567020	0.276232	5	4
2013-01-06	-0.673690	0.113648	-1.478427	5	5

Series에 존재하지 않는 인덱스의 값은 NaN으로 채워집니다.

2013-01-02	1
2013-01-03	2
2013-01-04	3
2013-01-05	4
2013-01-06	5
2013-01-07	6

s1

Dataframe에 존재하지 않는 인덱스의 값은 제외됩니다.

dandyrilla.github.io

앞서 배운 조건을 이용한 데이터 선택 방법을 이용하여 다음과 같이 특정 조건에 만족하는 값들만 변경할 수도 있습니다. 다음 은 양수의 값을 가지는 값들에 한해서 음수로 바꿔주는 예제입니다. 결국에는 0 또는 음수만을 가지는 데이터 프레임들 만들 수 있습니다.

```
df2 = df.copy()
df2[df2 > 0] = -df2
```

df2		A	B	C	D	F
#						
#	2013-01-01	0.000000	0.000000	-1.509059	-5	NaN
#	2013-01-02	-1.212112	-0.173215	-0.119209	-5	-1.0
#	2013-01-03	-0.861849	-2.104569	-0.494929	-5	-2.0
#	2013-01-04	-0.721555	-0.706771	-1.039575	-5	-3.0
#	2013-01-05	-0.424972	-0.567020	-0.276232	-5	-4.0
#	2013-01-06	-0.673690	-0.113648	-1.478427	-5	-5.0

## 4. 결측치 (Missing Data)

여러가지 이유로 우리는 데이터를 전부 다 측정하지 못하는 경우가 종종 발생합니다. 이처럼 측정되지 못하여 비어있는 데이터를 ‘결측치’라고 합니다. pandas 에서는 결측치를 np.nan 으로 나타냅니다. pandas 에서는 결측치를 기본적으로 연산에서 제외시키고 있습니다. Working with missing data 항목을 참고하기 바랍니다.

재인덱싱(reindex)은 해당 축에 대하여 인덱스를 변경/추가/삭제를 하게됩니다. 이는 복사된 데이터프레임을 반환합니다.

```
df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])
df1.loc[dates[0]:dates[1], 'E'] = 1
```

#		A	B	C	D	F	E
#	2013-01-01	0.000000	0.000000	-1.509059	5	NaN	1.0
#	2013-01-02	1.212112	-0.173215	0.119209	5	1.0	1.0
#	2013-01-03	-0.861849	-2.104569	-0.494929	5	2.0	NaN
#	2013-01-04	0.721555	-0.706771	-1.039575	5	3.0	NaN

https://dandyrilla.github.io/2017-08-12/pandas-10min/

19/35

2020. 9. 13.

판다스(pandas) 기본 사용법 익히기

결측치가 하나라도 존재하는 행들을 버리고 싶을 때는 dropna() 메소드를 이용합니다. 결과적으로 결측치가 하나도 없는 두 번째 행만 남고 나머지 행들은 사라졌습니다.

```
df1.dropna(how='any')
```

#		A	B	C	D	F	E
#	2013-01-02	1.212112	-0.173215	0.119209	5	1.0	1.0

만약 결측치가 있는 부분을 다른 값으로 채우고 싶다면 fillna() 메소드를 이용하세요.

```
df1.fillna(value=5)
```

#		A	B	C	D	F	E
#	2013-01-01	0.000000	0.000000	-1.509059	5	5.0	1.0
#	2013-01-02	1.212112	-0.173215	0.119209	5	1.0	1.0
#	2013-01-03	-0.861849	-2.104569	-0.494929	5	2.0	5.0
#	2013-01-04	0.721555	-0.706771	-1.039575	5	3.0	5.0

그리고 해당 값이 결측치인지 아닌지의 여부를 알고싶다면 isna() 메소드를 이용하면 됩니다. 결측치이면 True, 값이 있다면 False 로 나타냅니다.

```
pd.isna(df1)
```

#		A	B	C	D	F	E
#	2013-01-01	False	False	False	False	True	False
#	2013-01-02	False	False	False	False	False	False
#	2013-01-03	False	False	False	False	False	True
#	2013-01-04	False	False	False	False	False	True

## 5. 연산 (Operations)

사용자 가이드의 바이너리 연산자 를 참고하세요.

## 통계적 지표들 (Stats)

평균 구하기. 일반적으로 결측치는 제외하고 연산을 합니다.

```
df.mean()
```

# A	-0.004474
# B	-0.383981
# C	-0.687758
# D	5.000000
# F	3.000000
# dtype:	float64

다른 축에 대해서 평균 구하기. mean() 함수의 인자로 1을 주게 되면 컬럼이 아닌 인덱스를 기준으로 연산을 합니다.

```
df.mean(1)
```

# 2013-01-01	0.872735
# 2013-01-02	1.431621
# 2013-01-03	0.707731
# 2013-01-04	1.395042
# 2013-01-05	1.883656
# 2013-01-06	1.592306
# Freq: D, dtype: float64	

https://dandyrilla.github.io/2017-08-12/pandas-10min/

20/35

서로 차원이 달라 인덱스를 맞추어야 하는 두 오브젝트 간의 연산의 예제입니다. pandas 는 맞추어야 할 축만 지정해 준다면 자동으로 해당 축을 기준으로 맞추어 연산을 수행합니다. 아래는 인덱스를 기준으로 연산이 수행되고 있습니다. 기존 데이터 프레임의 인덱스가 2013-01-03, 04, 05 인 모든 컬럼에 해당하는 값에 각각 1.0, 3.0, 5.0 를 빼준 값이 결과로 나옵니다. 또한 결측치가 존재하는 경우에는 계산이 불가능 하므로 NaN 으로 표시된다는 것도 알 수 있습니다.

```
s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)
# 2013-01-01      NaN
# 2013-01-02      NaN
# 2013-01-03      1.0
# 2013-01-04      3.0
# 2013-01-05      5.0
# 2013-01-06      NaN
# Freq: D, dtype: float64
```

```
df.sub(s, axis='index')
#               A          B          C          D          F
# 2013-01-01      NaN      NaN      NaN      NaN      NaN
# 2013-01-02      NaN      NaN      NaN      NaN      NaN
# 2013-01-03 -1.861849 -3.104569 -1.494929  4.0    1.0
# 2013-01-04 -2.278445 -3.706771 -4.039575  2.0    0.0
# 2013-01-05 -5.424972 -4.432980 -4.723768  0.0   -1.0
# 2013-01-06      NaN      NaN      NaN      NaN      NaN
```

함수 적용하기 (Apply)

데이터프레임에 함수를 적용할 수 있습니다. 기존에 존재하는 함수를 사용하거나 사용자가 정의한 람다 함수를 사용할 수도 있습니다.

```
df.apply(np.cumsum)
#               A          B          C          D          F
# 2013-01-01  0.000000  0.000000 -1.509059    5      NaN
# 2013-01-02  1.212112 -0.173215 -1.389850   10    1.0
# 2013-01-03  0.350263 -2.277784 -1.884779   15    3.0
# 2013-01-04  1.071818 -2.984555 -2.924354   20    6.0
# 2013-01-05  0.646846 -2.417535 -2.648122   25   10.0
# 2013-01-06 -0.026844 -2.303886 -4.126549   30   15.0
```

```
df.apply(lambda x: x.max() - x.min())
# A      2.073961
# B      2.671590
# C      1.785291
# D      0.000000
# F      4.000000
# dtype: float64
```

히스토그램 구하기 (Histogramming)

데이터의 값들의 빈도를 조사하여 히스토그램을 만들 수 있습니다. [Histogramming and Discretization](#)에서 더 많은 정보를 찾아보세요.

```
s = pd.Series(np.random.randint(0, 7, size=10))
# 0      4
# 1      2
# 2      1
# 3      2
```

```
# 4      6
# 5      4
# 6      4
# 7      6
# 8      4
# 9      4
# dtype: int64
```

```
s.value_counts()
# 4      5
# 6      2
# 2      2
# 1      1
# dtype: int64
```

문자열 관련 메소드들 (String methods)

아래의 예제처럼 시리즈(Series)는 배열의 각 요소에 쉽게 적용이 가능하도록 `str` 이라는 속성에 문자열을 처리할 수 있는 여러가지의 메소드들을 갖추고 있습니다. 문자열 내에서의 패턴을 찾기 위한 작업들은 일반적으로 기본적으로 정규표현식을 사용하는 것에 유의합니다. (몇몇의 경우에는 항상 정규표현식을 사용합니다.) 더 많은 정보는 [Vectorized String Methods](#) 에서 찾아보세요.

```
s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
```

```
s.str.lower()
# 0      a
# 1      b
# 2      c
# 3    aaba
# 4    baca
# 5     NaN
# 6    caba
# 7    dog
# 8    cat
# dtype: object
```

6. 합치기 (Merging)

다양한 정보를 담은 자료들이 있을 때 이들을 합쳐 새로운 자료를 만들어야 할 때가 있습니다. 이번에는 시리즈(Series) 또는 데이터프레임(DataFrame)을 어떻게 합치는지를 알아볼 것입니다. 같은 형태의 자료들을 이어 하나로 만들어주는 `concat` , 다른 형태의 자료들을 한 컬럼을 기준으로 합치는 `merge` , 기존 데이터 프레임에 하나의 행을 추가하는 `append` 의 사용법에 대해 알아봅시다.

Concat

아래는 concat 을 이용하여 pandas 오브젝트들을 일렬로 잇는 예제입니다. 임의의 수를 담고있는 10 x 4 형태의 데이터 프레임의 만든 후 세 부분으로 쪼개었다가 pandas 에 있는 concat 메소드를 이용하여 원래대로 다시 합칠 수 있다는 것을 보여 줍니다.

```
df = pd.DataFrame(np.random.randn(10, 4))
#               0          1          2          3
# 0 -0.548702  1.467327 -1.015962 -0.483075
```



```
# 1  1.637550 -1.217659 -0.291519 -1.745505
# 2 -0.263952  0.991460 -0.919069  0.266046
# 3 -0.709661  1.669052  1.037882 -1.705775
# 4 -0.919854 -0.042379  1.247642 -0.009920
# 5  0.290213  0.495767  0.362949  1.548106
# 6 -1.131345 -0.089329  0.337863 -0.945867
# 7 -0.932132  1.956030  0.017587 -0.016692
# 8 -0.575247  0.254161 -1.143704  0.215897
# 9  1.193555 -0.077118 -0.408530 -0.862495
```

```
# break it into pieces
pieces = [df[:3], df[3:7], df[7:]]
```

```
# concatenate again
pd.concat(pieces)
#           0           1           2           3
# 0 -0.548702  1.467327 -1.015962 -0.483075
# 1  1.637550 -1.217659 -0.291519 -1.745505
# 2 -0.263952  0.991460 -0.919069  0.266046
# 3 -0.709661  1.669052  1.037882 -1.705775
# 4 -0.919854 -0.042379  1.247642 -0.009920
# 5  0.290213  0.495767  0.362949  1.548106
# 6 -1.131345 -0.089329  0.337863 -0.945867
# 7 -0.932132  1.956030  0.017587 -0.016692
# 8 -0.575247  0.254161 -1.143704  0.215897
# 9  1.193555 -0.077118 -0.408530 -0.862495
```

concat 메소드에 대한 설명과 예제가 더 필요하신 분은 pandas 사용자 가이드 중 [concatenating objects](#) 을 참고하시기 바랍니다.

## Join

데이터베이스에서 사용하는 SQL 스타일의 합치기 기능입니다. merge 메소드를 통해 이루어집니다.

```
left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
#   key  lval
# 0  foo     1
# 1  foo     2

right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})
#   key  rval
# 0  foo     4
# 1  foo     5

merged = pd.merge(left, right, on='key')
#   key  lval  rval
# 0  foo     1     4
# 1  foo     1     5
# 2  foo     2     4
# 3  foo     2     5
```

또 다른 예제로는 이런 것이 있을 수 있습니다.

```
left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
#   key  lval
# 0  foo     1
# 1  bar     2
```

```
right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})
#   key  rval
# 0  foo     4
# 1  bar     5

merged = pd.merge(left, right, on='key')
#   key  lval  rval
# 0  foo     1     4
# 1  bar     2     5
```

위의 예제와 아래의 예제는 key 값을 중복으로 가질 때와 그렇지 않을 때의 merge 메소드의 작동방식을 설명해줍니다. 위의 예제에서는 key 값으로 모두 ‘foo’ 라는 문자열을 가지고 있고, 아래의 예제에서는 key 값으로 ‘foo’ 또는 ‘bar’ 를 가지고 있습니다. 보통 key 로 사용하는 값은 중복될 경우가 잘 없지만, 만약에 중복된 값이 있을 때에는 모든 경우의 수를 만들어내는 작동방식을 보여주고 있습니다.

## Append

데이터프레임의 맨 뒤에 행을 추가합니다. 아래의 예제는 4번째 행을 기존의 데이터프레임의 맨 뒤에 한번 더 추가하는 방법을 보여주고 있습니다.

```
df = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])
#           A           B           C           D
# 0  1.346061  1.511763  1.627081 -0.990582
# 1 -0.441652  1.211526  0.268520  0.024580
# 2 -1.577585  0.396823 -0.105381 -0.532532
# 3  1.453749  1.208843 -0.080952 -0.264610
# 4 -0.727965 -0.589346  0.339969 -0.693205
# 5 -0.339355  0.593616  0.884345  1.591431
# 6  0.141809  0.220390  0.435589  0.192451
# 7 -0.096701  0.803351  1.715071 -0.708758
```

```
s = df.iloc[3]
df.append(s, ignore_index=True)
#           A           B           C           D
# 0  1.346061  1.511763  1.627081 -0.990582
# 1 -0.441652  1.211526  0.268520  0.024580
# 2 -1.577585  0.396823 -0.105381 -0.532532
# 3  1.453749  1.208843 -0.080952 -0.264610
# 4 -0.727965 -0.589346  0.339969 -0.693205
# 5 -0.339355  0.593616  0.884345  1.591431
# 6  0.141809  0.220390  0.435589  0.192451
# 7 -0.096701  0.803351  1.715071 -0.708758
# 8  1.453749  1.208843 -0.080952 -0.264610
```

Append 메소드에 대한 더 자세한 설명은 [Appending to dataframe](#) 섹션을 참고해주세요.

## 7. 묶기 (Grouping)

‘그룹화 (group by)’는 다음과 같은 처리를 하는 과정들을 지칭합니다.

- 어떠한 기준을 바탕으로 데이터를 나누는 일 (splitting)
- 각 그룹에 어떤 함수를 독립적으로 적용시키는 일 (applying)
- 적용되어 나온 결과들을 통합하는 일 (combining)

자세한 사항은 [grouping](#) 섹션을 참고하기 바랍니다.

```
df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                        'foo', 'bar', 'foo', 'foo'],
                  'B': ['one', 'one', 'two', 'three',
                        'two', 'two', 'one', 'three'],
                  'C': np.random.randn(8),
                  'D': np.random.randn(8)})
```

#	A	B	C	D
# 0	foo	one	-1.202872	-0.055224
# 1	bar	one	-1.814470	2.395985
# 2	foo	two	1.018601	1.552825
# 3	bar	three	-0.595447	0.166599
# 4	foo	two	1.395433	0.047609
# 5	bar	two	-0.392670	-0.136473
# 6	foo	one	0.007207	-0.561757
# 7	foo	three	1.928123	-1.623033

A 컬럼의 값을 기준으로 그룹을 묶고 각 그룹에 합계를 구하는 `sum()` 함수를 적용해 봅시다. 인덱스로는 A 컬럼이 되고, 합계를 구할 수 있는 C 와 D 컬럼에 있는 숫자들의 합계가 구해진 데이터프레임이 만들어집니다.

```
df.groupby('A').sum()
#           C           D
# A
# bar -2.802588  2.42611
# foo  3.146492 -0.63958
```

그룹을 묶을 때 여러 컬럼을 기준으로 이용할 수도 있습니다. 다음은 A와 B 컬럼을 기준으로 묶어 계층 구조의 인덱스를 형성하고, 앞의 예제와 마찬가지로 합계를 다시 구해봅시다.

```
df.groupby(['A', 'B']).sum()
#           C           D
# A  B
# bar one   -1.814470  2.395985
#     three -0.595447  0.166599
#     two   -0.392670 -0.136473
# foo one   -1.195665 -0.616981
#     three  1.928123 -1.623033
#     two    2.410434  1.600434
```

## 8. 변형하기 (Reshaping)

데이터 프레임은 다른 형태로 변형하는 방법들에 대해 알아봅시다. 자세한 내용은 [Hierarchical Indexing](#) 과 [Reshaping](#) 을 참고 바랍니다.

### Stack

stack 메소드는 데이터 프레임의 컬럼들을 인덱스의 레벨로 만듭니다. 이를 ‘압축’ 한다고 표현합니다. 아래 예제를 보시면 `df2` 라는 데이터프레임은 A 와 B 컬럼을 갖고 있었지만 stack 메소드를 통해 A 와 B 라는 값을 가지는 인덱스 레벨이 하나 더 추가된 형태로 변형되었습니다.

```
tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                    'foo', 'foo', 'qux', 'qux'],
                    ['one', 'two', 'one', 'two',
                    'one', 'two', 'one', 'two']]))
index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])
df2 = df[:4]
```

#		A	B
# first second			
# bar one		0.029399	-0.542108
# two		0.282696	-0.087302
# baz one		-1.575170	1.771208
# two		0.816482	1.100230

```
stacked = df2.stack()
# first second
# bar one A 0.029399
# B -0.542108
# two A 0.282696
# B -0.087302
# baz one A -1.575170
# B 1.771208
# two A 0.816482
# B 1.100230
# dtype: float64
```

stack 메소드를 통해 압축된 수준을 갖는 데이터프레임은 다시 unstack 메소드를 통해 원래대로 돌아올 수 있습니다. 여러번 unstack 메소드를 적용할 수 있지만 기본적으로 unstack 메소드는 stack 메소드를 통해 압축되었던 마지막 수준부터 풀어주는 기능을 갖습니다.

```
stacked.unstack()
#           A           B
# first second
# bar one 0.029399 -0.542108
# two 0.282696 -0.087302
# baz one -1.575170 1.771208
# two 0.816482 1.100230
```

그리고 아래와 같이 해제할 수준을 지정해 줄 수 있습니다. stack() 메소드의 인수로 0 을 입력하면 첫 번째 수준을 해제하므로 인덱스에서 first 수준이 해제되어 bar 와 baz 라는 컬럼이 생기게 되고, 1 을 입력하면 두 번째 수준인 second 를 해제하므로 one 과 two 라는 컬럼이 만들어진 데이터프레임을 얻을 수 있게 됩니다.

```
stacked.unstack(0)
# first bar baz
# second
# one A 0.029399 -1.575170
# B -0.542108 1.771208
# two A 0.282696 0.816482
# B -0.087302 1.100230
```

```
stacked.unstack(1)
# second one two
# first
# bar A 0.029399 0.282696
# B -0.542108 -0.087302
# baz A -1.575170 0.816482
# B 1.771208 1.100230
```

# Pivot Tables

Pivot Tables 을 참고하세요.

```
df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3,
                   'B': ['A', 'B', 'C'] * 4,
                   'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
                   'D': np.random.randn(12),
                   'E': np.random.randn(12)})
```

#		A	B	C	D	E
# 0	one	A	foo	1.418757	-0.179666	
# 1	one	B	foo	-1.879024	1.291836	
# 2	two	C	foo	0.536826	-0.009614	
# 3	three	A	bar	1.006160	0.392149	
# 4	one	B	bar	-0.029716	0.264599	
# 5	one	C	bar	-1.146178	-0.057409	
# 6	two	A	foo	0.100900	-1.425638	
# 7	three	B	foo	-1.035018	1.024098	
# 8	one	C	foo	0.314665	-0.106062	
# 9	one	A	bar	-0.773723	1.824375	
# 10	two	B	bar	-1.170653	0.595974	
# 11	three	C	bar	0.648740	1.167115	

위와 같은 데이터 프레임의 형식을 피벗 테이블 기능을 이용하여 아래와 같이 쉽게 변형할 수 있습니다. 찾지 못한 값은 NaN 으로 표시됩니다.

```
pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
# C          bar      foo
# A          B
# one  A  -0.773723  1.418757
#      B  -0.029716 -1.879024
#      C  -1.146178  0.314665
# three A   1.006160      NaN
#      B      NaN -1.035018
#      C   0.648740      NaN
# two  A      NaN  0.100900
#      B  -1.170653      NaN
#      C      NaN  0.536826
```

## 9. 시계열 데이터 다루기 (Time Series)

시계열 데이터에서 1초 마다 측정된 데이터를 5분 마다 측정된 데이터의 형태로 바꾸고 싶을 땐 어떻게 할까요? Pandas 는 이렇게 시계열 단위의 주기(frequency)를 다시 샘플링 할 수 있는 단순하고, 강력하며, 효과적인 기능을 가지고 있습니다. 이 는 특히 금융 데이터를 다룰 때 매우 흔히 하는 연산입니다. (그렇다고 꼭 금융 데이터에 한정되어 있다는 뜻은 아닙니다.)

Time series / date functionality 을 참고하세요.

```
rng = pd.date_range('1/1/2012', periods=100, freq='S')

ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

ts.resample('5Min').sum()
```

```
# 2012-01-01      25083
# Freq: 5T, dtype: int64
```

타임존 표현:

```
rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')
```

```
ts = pd.Series(np.random.randn(len(rng)), rng)
```

```
ts
# 2012-03-06      0.464000
# 2012-03-07      0.227371
# 2012-03-08     -0.496922
# 2012-03-09      0.306389
# 2012-03-10     -2.290613
# Freq: D, dtype: float64
```

```
ts_utc = ts.tz_localize('UTC')
```

```
ts_utc
# 2012-03-06 00:00:00+00:00      0.464000
# 2012-03-07 00:00:00+00:00      0.227371
# 2012-03-08 00:00:00+00:00     -0.496922
# 2012-03-09 00:00:00+00:00      0.306389
# 2012-03-10 00:00:00+00:00     -2.290613
# Freq: D, dtype: float64
```

다른 타임존으로 변경하기:

```
ts_utc.tz_convert('US/Eastern')
# 2012-03-05 19:00:00-05:00      0.464000
# 2012-03-06 19:00:00-05:00      0.227371
# 2012-03-07 19:00:00-05:00     -0.496922
# 2012-03-08 19:00:00-05:00      0.306389
# 2012-03-09 19:00:00-05:00     -2.290613
# Freq: D, dtype: float64
```

시간 표현법으로 변경하기:

```
rng = pd.date_range('1/1/2012', periods=5, freq='M')
```

```
ts = pd.Series(np.random.randn(len(rng)), index=rng)
```

```
ts
# 2012-01-31     -1.134623
# 2012-02-29     -1.561819
# 2012-03-31     -0.260838
# 2012-04-30      0.281957
# 2012-05-31      1.523962
# Freq: M, dtype: float64
```

```
ps = ts.to_period()
```

```
ps
# 2012-01     -1.134623
# 2012-02     -1.561819
# 2012-03     -0.260838
# 2012-04      0.281957
```

```
# 2012-05      1.523962
# Freq: M, dtype: float64

ps.to_timestamp()
# 2012-01-01    -1.134623
# 2012-02-01    -1.561819
# 2012-03-01    -0.260838
# 2012-04-01     0.281957
# 2012-05-01     1.523962
# Freq: MS, dtype: float64
```

기간과 특정시간 사이의 변환에 편리한 산술적 기능들을 사용할 수 있습니다. 뒤이어 나오는 예제는 11월을 끝으로 하는 4분기 체계에서 각 분기의 마지막 달에 9시간을 더한 시각을 시작으로 하는 체계로 바꾸는 것을 보여줍니다.

```
prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

ts = pd.Series(np.random.randn(len(prng)), prng)

ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9
```

```
ts.head()
# 1990-03-01 09:00    -0.902937
# 1990-06-01 09:00     0.068159
# 1990-09-01 09:00    -0.057873
# 1990-12-01 09:00    -0.368204
# 1991-03-01 09:00    -1.144073
# Freq: H, dtype: float64
```

## 10. 범주형 데이터 다루기 (Categoricals)

Pandas는 데이터프레임(DataFrame)에 범주형 데이터도 포함시킬 수 있습니다. 더 자세한 정보는 [categorical introduction](#) 과 [API documentation](#) 을 참고하세요.

```
df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
                   "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})
```

단순한 문자로 되어있는 raw grade 컬럼을 범주형으로 바꿀 수 있습니다.

```
df["grade"] = df["raw_grade"].astype("category")

df["grade"]
# 0    a
# 1    b
# 2    b
# 3    a
# 4    a
# 5    e
# Name: grade, dtype: category
# Categories (3, object): [a, b, e]
```

범주들의 이름을 더욱 의미있는 것으로 바꾸어 줄 수 있습니다. (곧바로 `Series.cat.categories` 에 이름들을 할당하면 됩니다.)

```
df["grade"].cat.categories = ["very good", "good", "very bad"]
```

범주의 순서를 재정렬하는 동시에, 현재 갖고있지 않는 범주도 추가 가능합니다. ( `Series.cat` 아래의 메소드들은 기본적으로 새로운 시리즈를 반환합니다.)

```
df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium",
                                             "good", "very good"])
```

```
df["grade"]
# 0    very good
# 1         good
# 2         good
# 3    very good
# 4    very good
# 5    very bad
# Name: grade, dtype: category
# Categories (5, object): [very bad, bad, medium, good, very good]
```

정렬은 범주 이름의 어휘적 순서가 아닌, 범주에 이미 매겨진 값의 순서대로 이루어집니다. (즉, 범주형 자료를 만들거나 범주들을 재정의할 때 이루어진 순서가 범주에 매겨진 값입니다.)

```
df.sort_values(by="grade")
#   id raw_grade  grade
# 5   6         e  very bad
# 1   2         b    good
# 2   3         b    good
# 0   1         a  very good
# 3   4         a  very good
# 4   5         a  very good
```

범주형 자료를 담고있는 컬럼을 그룹으로 묶고 각 범주에 해당하는 값의 빈도수를 출력합니다. 이렇게 하면 비어있는 범주가 무엇인지도 알 수 있습니다.

```
df.groupby("grade").size()
# grade
# very bad    1
# bad         0
# medium      0
# good        2
# very good   3
# dtype: int64
```

## 11. 그래프로 표현하기 (Plotting)

다음과 같은 시계열 데이터가 있을 때, 그래프 그리기는 다음과 같이 `plot()` 메소드 하나만으로 완성할 수 있다.

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
ts = ts.cumsum()
ts.plot()
```

## 12. 데이터 입/출력 (Getting Data In/Out)

데이터를 다양한 형식의 파일에 읽고 쓰는 방법을 알아봅니다.

### CSV

데이터 프레임을 CSV 형식으로 저장하기.

```
df.to_csv('foo.csv')
```

CSV 형식으로 된 파일로부터 데이터 프레임의 형식으로 읽어오기. CSV 형식으로 부터 읽어올 때 주의할 점은 기존 행 인덱스를 인식하지 못하고 행 인덱스를 가지는 새로운 열이 추가로 잡힌다는 것입니다. 따라서 저장할 당시에는 4개였던 열의 개수가 5개가 되어있는 것을 확인할 수 있습니다.

```
pd.read_csv('foo.csv')
#      Unnamed: 0      A      B      C      D
# 0  2000-01-01  0.266457 -0.399641 -0.219582  1.186860
# 1  2000-01-02 -1.170732 -0.345873  1.653061 -0.282953
# 2  2000-01-03 -1.734933  0.530468  2.060811 -0.515536
# 3  2000-01-04 -1.555121  1.452620  0.239859 -1.156896
# 4  2000-01-05  0.578117  0.511371  0.103552 -2.428202
# 5  2000-01-06  0.478344  0.449933 -0.741620 -1.962409
# 6  2000-01-07  1.235339 -0.091757 -1.543861 -1.084753
# ...      ...      ...      ...      ...
# 993 2002-09-20 -10.628548 -9.153563 -7.883146  28.313940
# 994 2002-09-21 -10.390377 -8.727491 -6.399645  30.914107
# 995 2002-09-22 -8.985362 -8.485624 -4.669462  31.367740
# 996 2002-09-23 -9.558560 -8.781216 -4.499815  30.518439
# 997 2002-09-24 -9.902058 -9.340490 -4.386639  30.105593
# 998 2002-09-25 -10.216020 -9.480682 -3.933802  29.758560
# 999 2002-09-26 -11.856774 -10.671012 -3.216025  29.369368
#
# [1000 rows x 5 columns]
```

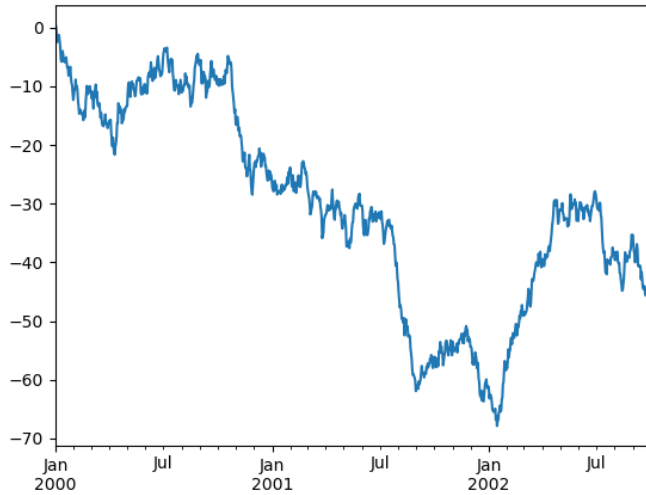
### HDF5

HDF5 형식으로 저장하기.

```
df.to_hdf('foo.h5', 'df')
```

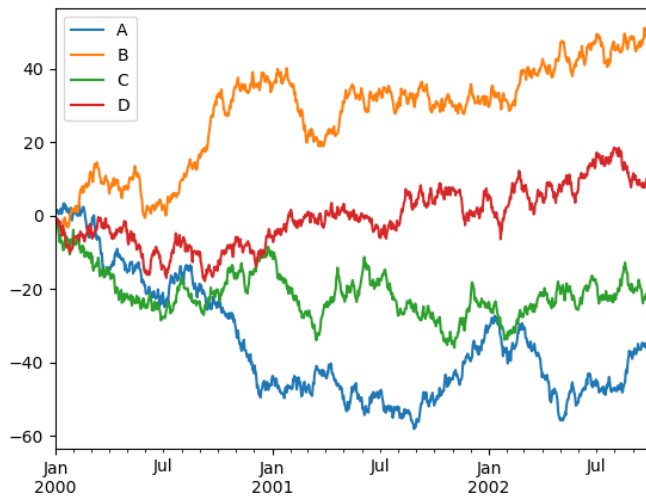
HDF5 형식으로부터 읽어오기.

```
pd.read_hdf('foo.h5', 'df')
#      A      B      C      D
# 2000-01-01  0.266457 -0.399641 -0.219582  1.186860
# 2000-01-02 -1.170732 -0.345873  1.653061 -0.282953
# 2000-01-03 -1.734933  0.530468  2.060811 -0.515536
# 2000-01-04 -1.555121  1.452620  0.239859 -1.156896
# 2000-01-05  0.578117  0.511371  0.103552 -2.428202
# 2000-01-06  0.478344  0.449933 -0.741620 -1.962409
# 2000-01-07  1.235339 -0.091757 -1.543861 -1.084753
# ...      ...      ...      ...
# 2002-09-20 -10.628548 -9.153563 -7.883146  28.313940
```



`plot()` 메소드는 여러 개의 열을 한 번에 그릴 수 있는 편리함도 제공하고 있다. 다음과 같이 A, B, C, D의 4개의 열에 해당하는 데이터를 `legend` 와 함께 표시할 수 있다.

```
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=['A', 'B', 'C', 'D'])
df = df.cumsum()
plt.figure(); df.plot(); plt.legend(loc='best')
```



그래프를 그리는 더욱 자세한 방법은 [visualization](#) 항목을 참고하시기 바랍니다.

Google에 의해 종료된 광고입니다.

또 다른 데이터 오브젝트인 **DataFrame**은 여러 형태의 데이터를 받아 생성할 수 있는데요. 그 중 한 방법으로 아래와 같이 numpy array를 받아 생성이 가능합니다. 앞서 설명드린 것처럼 DataFrame은 2차원 배열의 형태를 띄고 있습니다. 따라서 우리가 자주 보는 표 형태와 같이 두 가지의 기준에 따라 데이터를 담고 있습니다. 아래의 예제에서는 첫번째 기준은 날짜, 두 번째 기준은 장소(A, B, C, D라는 네 곳의 위치) **1**에 따라 측정된 어떤 값들이 담겨 있다고 생각하면 쉬울 것 같습니다. DataFrame을 만들기 위해서는 `pd.DataFrame()`라는 클래스 생성자를 사용하며, 행에 해당하는 기준(첫번째 기준)인 인덱스를 `index`라는 인수로 전달하며, 열에 해당하는 기준(두번째 기준)인 컬럼을 `columns`이라는 인수로 전달합니다. 여기에서는 인덱스로 `pd.date_range()`를 사용하여 날짜 값들을 만들어 전달해 주었고, 컬럼의 이름은 A, B, C, D라는 이름이 담긴 리스트로 넣어보았습니다.

```
dates = pd.date_range('20130101', periods=6)
# DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
#                '2013-01-05', '2013-01-06'],
#               dtype='datetime64[ns]', freq='D')

df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
#           A          B          C          D
# 2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
# 2013-01-02  1.212112 -0.173215  0.119209 -1.044236
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
# 2013-01-04  0.721555 -0.706771 -1.039575  0.271860
# 2013-01-05 -0.424972  0.567020  0.276232 -1.087401
# 2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

AISchool

TensorFlow

딥러닝-비전 코스

프로젝트 실습을 통해  
TensorFlow 2.0을 이용한  
딥러닝 알고리즘 구현방법을  
학습합니다.

DataFrame을 생성하는 또 다른 방법으로 아래와 같이 여러 종류의 자료들이 담긴 딕셔너리(dict)를 넣어주어 만들 수 있습니다. 이 때에는 dict의 key 값이 열을 정의하는 컬럼이 되며, 행을 정의하는 인덱스는 자동으로 0부터 시작하여 1씩 증가하는 정수 인덱스가 사용됩니다.

```
df2 = pd.DataFrame({'A': 1.,
                    'B': pd.Timestamp('20130102'),
                    'C': pd.Series(1, index=list(range(4)), dtype='float32'),
                    'D': np.array([3]*4, dtype='int32'),
                    'E': pd.Categorical(['test', 'train', 'test', 'train']),
                    'F': 'foo'})

#           A          B          C          D          E          F
# 0  1.0 2013-01-02  1.0 3  test  foo
# 1  1.0 2013-01-02  1.0 3  train  foo
# 2  1.0 2013-01-02  1.0 3  test  foo
# 3  1.0 2013-01-02  1.0 3  train  foo
```

Google에 의해 종료된 광고입니다.

DataFrame의 컬럼들은 각기 특별한 자료형을 갖고 있을 수 있습니다. 이는 DataFrame내에 있는 dtypes라는 속성을 통해 확인 가능합니다. 파이썬의 기본적인 소수점은 float64로 잡히고, 기본적인 문자열은 str이 아니라 object라는 자료형으로 나타납니다.

```
df2.dtypes
# A          float64
# B  datetime64[ns]
# C          float32
# D          int32
# E          category
# F          object
# dtype: object
```

Google에 의해 종료된 광고입니다.

Jupyter를 사용하시는 분이라면 `df2.<TAB>` ('df2.'까지 입력하고 탭을 누름)을 통해 다음과 같이 dtypes 외에도 다른 속성들이 무엇이 있는지 확인할 수 있습니다.

```
In [13]: df2.<TAB>
df2.A          df2.bool
```

df2.abs	df2.boxplot
df2.add	df2.C
df2.add_prefix	df2.clip
df2.add_suffix	df2.clip_lower
df2.align	df2.clip_upper
df2.all	df2.columns
df2.any	df2.combine
df2.append	df2.combine_first
df2.apply	df2.compound
df2.applymap	df2.consolidate
df2.as_blocks	df2.convert_objects
df2.asfreq	df2.copy
df2.as_matrix	df2.corr
df2.astype	df2.corrwith
df2.at	df2.count
df2.at_time	df2.cov
df2.axes	df2.cummax
df2.B	df2.cummin
df2.between_time	df2.cumprod
df2.bfill	df2.cumsum
df2.blocks	df2.D

보다시피 컬럼 A, B, C, D 가 자동적으로 생성되어 나타나는 것을 확인할 수 있습니다. 나머지 속성들은 간결성을 위해 생략하였기 때문에 E 도 뒤에 있을 것입니다.

Google에 의해 종료된 광고입니다.

Ipython 을 사용하지 않는 분이라면, python 의 빌트인 함수 `dir` 을 통해 다음과 같이 오브젝트가 갖고 있는 속성 및 메소드들을 모두 확인 가능합니다. (약 400 개가 넘는 항목입니다. 엄청 많습니다)

```
dir(df2)
# ['A', 'B', 'C', ... , 'values', 'var', 'where', 'xs']
```

이 외에도 pandas 에서 제공하는 자료 구조들이 무엇이 있는지 알아보시려면 pandas 공식 문서에 있는 [Intro to data structures](#) 을 참고하시면 됩니다.

## 2. 데이터 확인하기 (Viewing Data)

이 부분에 대해 더 자세히 알고 싶으시면 [Essential basic functionality](#) 을 참고해주세요.

DataFrame 에 들어있는 자료들을 확인하기 위해 맨 앞이나 뒤의 자료들 몇 개를 알아보고 싶다면 다음과 같이 `.head()` 와 `.tail()` 메소드를 사용하면 됩니다. 기본적으로 상위 또는 하위 5 개의 자료를 보여주는데, 더 적게 혹은 많이 보고 싶다면 메소드의 인자로 보고싶은 데이터의 개수를 숫자를 넣어주면 됩니다.

```
## 첫 5개 행의 데이터를 보여줍니다.
df.head()
#           A           B           C           D
# 2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
# 2013-01-02  1.212112 -0.173215  0.119209 -1.044236
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
# 2013-01-04  0.721555 -0.706771 -1.039575  0.271860
# 2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

```
## 마지막 3개 행의 데이터를 보여줍니다.
df.tail(3)
#           A           B           C           D
# 2013-01-04  0.721555 -0.706771 -1.039575  0.271860
# 2013-01-05 -0.424972  0.567020  0.276232 -1.087401
# 2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Google에 의해 종료된 광고입니다.

DataFrame의 인덱스를 보려면 `.index` 속성을, 컬럼을 보려면 `.columns` 속성을, 안에 들어있는 numpy 데이터를 보려면 `.values` 속성을 통해 확인하면 됩니다.

```
df.index
# DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
#               '2013-01-05', '2013-01-06'],
#               dtype='datetime64[ns]', freq='D')
```

```
df.columns
# Index(['A', 'B', 'C', 'D'], dtype='object')
```

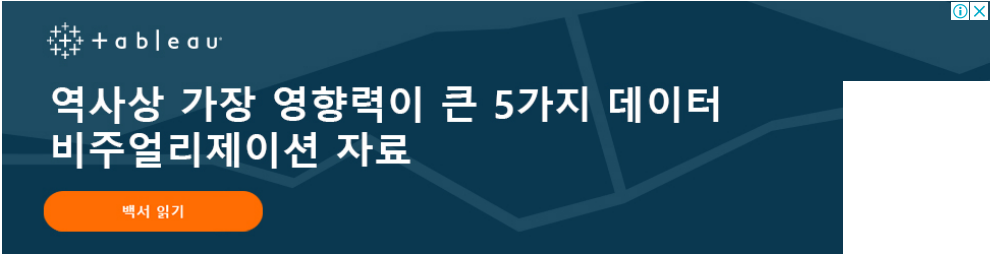
```
df.values
# [[ 0.4691, -0.2829, -1.5091, -1.1356],
#   [ 1.2121, -0.1732,  0.1192, -1.0442],
#   [-0.8618, -2.1046, -0.4949,  1.0718],
#   [ 0.7216, -0.7068, -1.0396,  0.2719],
#   [-0.425 ,  0.567 ,  0.2762, -1.0874],
#   [-0.6737,  0.1136, -1.4784,  0.525 ]]
```

`.describe()` 메소드는 생성했던 DataFrame 의 간단한 통계 정보를 보여줍니다. 컬럼별로 데이터의 개수(count), 데이터의 평균값(mean), 표준 편차(std), 최솟값(min), 4분위수(25%, 50%, 75%), 그리고 최댓값(max)들의 정보를 알 수 있습니다.

```
df.describe()
#           A           B           C           D
# count  6.000000  6.000000  6.000000  6.000000
# mean    0.073711 -0.431125 -0.687758 -0.233103
# std     0.843157  0.922818  0.779887  0.973118
# min    -0.861849 -2.104569 -1.509059 -1.135632
```

```
# 25%    -0.611510  -0.600794  -1.368714  -1.076610
# 50%     0.022070  -0.228039  -0.767252  -0.386188
# 75%     0.658444   0.041933  -0.034326   0.461706
# max      1.212112   0.567020   0.276232   1.071804
```

.T 속성은 DataFrame 에서 index와 column 을 바꾼 형태의 DataFrame 입니다. pandas.DataFrame.T 에는 .T 를 ‘Transpose index and columns’와 같이 설명해 놓고 있어서 index와 column 을 바꾼 후 리턴값으로 돌려주는 메소드로 착각할 수 있습니다. 따라서 .T() 로 호출하는 경우가 있으실 텐데, 그렇게 해보니 에러가 나는군요. 메소드가 아니라 미리 계산되어 저장되어 있는 ‘속성’이라는 점을 다시 강조합니다.



```
## 열과 행을 바꾼 형태의 데이터프레임입니다.
df.T
#      2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
# A      0.469112    1.212112   -0.861849    0.721555   -0.424972   -0.673690
# B     -0.282863   -0.173215   -2.104569   -0.706771    0.567020    0.113648
# C     -1.509059    0.119209   -0.494929   -1.039575    0.276232   -1.478427
# D     -1.135632   -1.044236    1.071804    0.271860   -1.087401    0.524988
```

```
## .T는 속성임을 알아두세요. 다음과 같이 메소드로 호출한다면 에러를 냅니다.
df.T()
# Traceback (most recent call last):
#   File "./main.py", line 5, in __main__
#     dFT = df.T()
# TypeError: 'DataFrame' object is not callable
```

그리고 .sort\_index() 라는 메소드로 행과 열 이름을 정렬하여 나타낼 수도 있습니다. 정렬할 대상 축을 결정할 때에는 axis 를 이용합니다. axis=0 라고 쓰주면 인덱스를 기준으로 정렬하며(기본값), axis=1 라고 쓰주면 컬럼을 기준으로 정렬합니다. 정렬의 방향에 대한 파라미터는 ascending 를 이용합니다. ascending=True 는 오름차순 정렬을 하겠다는 것이고(기본값), ascending=False 는 내림차순 정렬을 하겠다는 의미입니다. 다음은 컬럼에 대하여 내림차순 정렬을 하는 예제입니다.

```
df.sort_index(axis=1, ascending=False)
#      D      C      B      A
# 2013-01-01 -1.135632 -1.509059 -0.282863  0.469112
# 2013-01-02 -1.044236  0.119209 -0.173215  1.212112
# 2013-01-03  1.071804 -0.494929 -2.104569 -0.861849
# 2013-01-04  0.271860 -1.039575 -0.706771  0.721555
# 2013-01-05 -1.087401  0.276232  0.567020 -0.424972
# 2013-01-06  0.524988 -1.478427  0.113648 -0.673690
```

또한 DataFrame 내부에 있는 값으로 정렬할 수도 있습니다. 다음은 B 컬럼에 대해 정렬한 결과를 보여줍니다.

```
df.sort_values(by='B')
#      A      B      C      D
# 2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
# 2013-01-04  0.721555 -0.706771 -1.039575  0.271860
# 2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
# 2013-01-02  1.212112 -0.173215  0.119209 -1.044236
# 2013-01-06 -0.673690  0.113648 -1.478427  0.524988
# 2013-01-05 -0.424972  0.567020  0.276232 -1.087401
```

### 3. 데이터 선택하기 (Selection)

데이터프레임 자체가 갖고 있는 [] 슬라이싱 기능을 이용하는 방법입니다. 특정 ‘컬럼’의 값들만 가져오고 싶다면 df['A'] 와 같은 형태로 입력합니다. 이는 df.A 와 동일합니다. 리턴되는 값은 Series 의 자료구조를 갖고 있습니다.

```
## A라는 이름을 가진 컬럼의 데이터만 갖고옵니다.
df['A']
# 2013-01-01    0.469112
# 2013-01-02    1.212112
# 2013-01-03   -0.861849
# 2013-01-04    0.721555
# 2013-01-05   -0.424972
# 2013-01-06   -0.673690
# req: D, Name: A, dtype: float64
```

```
type(df['A'])
# <class 'pandas.core.series.Series'>
```



여러분의 이해를 돕기 위해 그림으로 다시 나타내면 다음과 같습니다. (앞으로 나오는 예제들도 그림으로 한번 더 설명하겠습니다.)



```
# 2002-09-21 -10.390377 -8.727491 -6.399645 30.914107
# 2002-09-22 -8.985362 -8.485624 -4.669462 31.367740
# 2002-09-23 -9.558560 -8.781216 -4.499815 30.518439
# 2002-09-24 -9.902058 -9.340490 -4.386639 30.105593
# 2002-09-25 -10.216020 -9.480682 -3.933802 29.758560
# 2002-09-26 -11.856774 -10.671012 -3.216025 29.369368
#
# [1000 rows x 4 columns]
```

Excel

데이터 프레임을 엑셀 파일로 저장하기.

```
df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

엑셀 파일로부터 데이터 프레임 읽어오기.

```
pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
#           A           B           C           D
# 2000-01-01  0.266457 -0.399641 -0.219582  1.186860
# 2000-01-02 -1.170732 -0.345873  1.653061 -0.282953
# 2000-01-03 -1.734933  0.530468  2.060811 -0.515536
# 2000-01-04 -1.555121  1.452620  0.239859 -1.156896
# 2000-01-05  0.578117  0.511371  0.103552 -2.428202
# 2000-01-06  0.478344  0.449933 -0.741620 -1.962409
# 2000-01-07  1.235339 -0.091757 -1.543861 -1.084753
# ...      ...      ...      ...      ...
# 2002-09-20 -10.628548 -9.153563 -7.883146 28.313940
# 2002-09-21 -10.390377 -8.727491 -6.399645 30.914107
# 2002-09-22 -8.985362 -8.485624 -4.669462 31.367740
# 2002-09-23 -9.558560 -8.781216 -4.499815 30.518439
# 2002-09-24 -9.902058 -9.340490 -4.386639 30.105593
# 2002-09-25 -10.216020 -9.480682 -3.933802 29.758560
# 2002-09-26 -11.856774 -10.671012 -3.216025 29.369368
#
# [1000 rows x 4 columns]
```

1. 예제 코드만으로는 A, B, C, D가 장소임을 알 수 없지만, 이해를 돕기 위하여 장소를 가리키는 명칭으로 해석하여 설명하였습니다. ↗

댓글 커뮤니티 개인정보 보호정책

♡ 추천 9 Tweet f 공유 인기순 ▼

토론 참여하기

정말 깔끔하게 정리 잘 되어있어 답글로라도 감사 표시 합니다.  
1 ^ | v · 답글 · 공유

Sukjun Kim 관리자 → kyle jang · 4달 전  
보시고 이렇게 답글 남겨주시니 정말 큰 힘이 됩니다.  
감사합니다!  
^ | v · 답글 · 공유

Kenneth · 일년 전  
맥키니 선생님이 올고갈만큼 멋지게 정리된 포스트네요! 자주  
들르겠습니다 ☺  
1 ^ | v · 답글 · 공유

Sukjun Kim 관리자 → Kenneth · 일년 전  
감사합니다. :-)  
^ | v · 답글 · 공유

상훈송 · 일년 전  
글이 고급집니다.  
즐거찾기 하고 수시 방문해야되는 페이지네요...  
잘보았습니다.  
1 ^ | v · 답글 · 공유

Sukjun Kim 관리자 → 상훈송 · 일년 전  
네 ㅎㅎ 판다스 사용시 자주 도움이 될 수 있는 페이지  
가 되었으면 좋겠습니다. 감사합니다!  
^ | v · 답글 · 공유

조광훈 · 2년 전  
잘 보았습니다.  
초보인데 많은 도움이 되었습니다.

df.to\_excel 에서 궁금한점 몇개 문의 합니다.

- 1.글씨의 크기 및 컬러 조절
- 2.셀 배경색
- 3. 데이터 형식을 yyyy-mm-dd hh:mm:ss 형식으로 하려면.

위 내용이 궁금하네요.


참고할 만한 사이트라도  
1 ^ | v · 답글 · 공유

Sukjun Kim 관리자 → 조광훈 · 2년 전  
광훈님, 안녕하세요.  
시간이 날 때마다 작성중이라 아직 일부만 완성되어 있  
는 포스팅인데, 도움이 되셨다니 감사합니다.

저도 to\_excel 메소드는 잘 써본 적이 없어서 자세히는  
모르는데요.  
몇 가지 키워드로 검색해 보니 pd.ExcelWriter 의 객체  
를 받아 원하시는 작업들을 하실 수 있을 것 같습니다.


아래 사이트에서 참고하시면 좋을 것 같습니다.

https://pandas.pydata.org/p...  
https://xlsxwriter.readthed...  
^ | v · 답글 · 공유



예시와 함께 정리를 너무 잘 해주셨네요! 감사합니다 다른 항목들도 업데이트 되었으면 좋겠어요 ^^


1 ^ | ^ . 답글 . 공유 >



**Sukjun Kim** 관리자 → Nack S • 2년 전

감사합니다!


^ | ^ . 답글 . 공유 >



**Gee Ji** • 16일 전

지금까지 본 판다스 정리 중 최고네요. 특히 데이터프레임마다 색깔로 보기 좋게 구분이 돼있어서 설명을 이해하기 좋았어요. 애쓰신 흔적이 보입니다. 덕분에 잘 배우고 갑니다. 즐겨찾기 추가했어요. 감사해요.


^ | ^ . 답글 . 공유 >



**Sukjun Kim** 관리자 → Gee Ji • 6일 전

우와... 글을 보는 안목이 정확하신데요 :) 어쩜 이렇게 확실하게 글의 특징을 짚어주셨나요! ㅎㅎㅎ 대단히 감사합니다.

^ | ^ . 답글 . 공유 >




**Gee Ji** → Sukjun Kim • 5일 전

안목은요~~ㅎㅎ. 잘 정리된 게 자연스럽게 눈에 들어옵니다. 마치 pandas cheat sheet처럼 말이죠!

요즘 판다스로 불러온 데이터프레임을 활용해서 다양한 코드가 있더군요. 자연어처리, word2vec 등등 구체적인 응용예시가 있으면 좋겠어요...기 대해 봅니다.

^ | ^ . 답글 . 공유 >



**Ilkoo Ahn** • 23일 전

감사합니다

^ | ^ . 답글 . 공유 >



**Sukjun Kim** 관리자 → Ilkoo Ahn • 23일 전

감사합니다!

^ | ^ . 답글 . 공유 >