Chapter 1. Pandas Basic

1.1 Introduction

데이터를 원하는 형식으로 항상 사용할 수 있는 것은 아니기 때문에 데이터 처리는 데이터 분석에서 중요한 부분입니다. Cleaning, Restructuring, Merging 등 데이터를 분석하기 전에 다양한 처리가 필요합니다. Numpy, Scipy, Cython, Pandas는 파이썬에서 사용할 수 있는 도구로서 데이터를 빠르게 처리할 수 있습니다.

Pandas는 다양한 유형의 데이터를 처리할 수 있는 풍부한 기능을 제공합니다. 또한 Pandas를 이용하면, 다른 도구보다 빠르고 쉽고 표현력이 뛰어납니다. Pandas는 스프레드시트 및 관계형 데이터베이스와 같은 유연한 데이터 조작 기술과 함께 Numpy처럼 빠른 데이터 처리 기능을 제공합니다. 마지막으로, Pandas는 Matplotlib 라이브러리와 잘 통합되어 데이터를 분석하는 데 매우 유용한 도구입니다.

● Note

- 1 장에서 두 가지 중요한 데이터 구조. 시리즈와 데이터 프레임에 대해 설명합니다.
- 2 장에서는 Pandas의 자주 사용되는 특징을 예로 보여줍니다. 그리고 이후의 챕터에는 Pandas에 대한 다양한 정보가 포함되어 있습니다.

1.1 Data structures

Pandas는 데이터를 처리하는 데 매우 유용한 두 가지 데이터 구조를 제공합니다. Series와 DataFrame을 이번 절에서 설명합니다.

1.2.1 Series

Series는 혼합 데이터 유형을 포함하여 다양한 데이터 유형을 저장할 수 있는 1차원 배열입니다. Series의 행 레이블을 인덱스라고 합니다. 모든 목록, 튜플 및 딕셔너리는 아래와 같이 'series' 메소드를 사용하여 시리즈로 변환할 수 있습니다.

```
>>> import pandas as pd

>>> # converting tuple to Series
>>> h = ('AA', '2012-02-01', 100, 10.2)
>>> s = pd.Series(h)

>>> type(s)
<class 'pandas.core.series.Series'>
```

```
>>> print(s)
0           AA
1   2012-02-01
2          100
3         10.2
dtype: object

>>> # converting dict to Series
>>> d = {'name' : 'IBM', 'date' : '2010-09-08', 'shares' : 100, 'price' : 10.2}
>>> ds = pd.Series(d)

>>> type(ds)
<class 'pandas.core.series.Series'>

>>> print(ds)
date      2010-09-08
name             IBM
price           10.2
shares           100
dtype: object
```

튜플 변환에서 인덱스는 '0, 1, 2 및 3'으로 설정됩니다. 다음과 같이 사용자 지정 인덱스 이름을 제공할 수 있습니다.

```
>>> f = ['FB', '2001-08-02', 90, 3.2]
>>> f = pd.Series(f, index = ['name', 'date', 'shares', 'price'])

>>> print(f)
name              FB
date      2001-08-02
shares            90
price            3.2
dtype: object

>>> f['shares']
90
>>> f[0]
'FB'
>>>
```

시리즈의 요소는 인덱스 이름을 사용하여 액세스할 수 있습니다. 아래 코드에서 f['shares'] 또는 f[0]. 또한 목록에 색인을 제공하여 특정 요소를 선택할 수 있습니다.

```
>>> f[['shares', 'price']]
shares     90
price     3.2
dtype: object
```

1.2.2 DataFrame

DataFrame은 Pandas의 널리 사용되는 데이터 구조입니다. Series는 1차원 배열과 함께 작동하는 데 사용되는 반면 DataFrame은 2차원 배열과 함께 사용할 수 있습니다. DataFrame에는 열 인덱스와 행 인덱스의 두 가지 인덱스가 있습니다.

DataFrame을 생성하는 가장 일반적인 방법은 아래와 같이 동일한 길이의 목록 딕셔너리를 사용하는 것입니다. 또한 모든 스프레드시트와 텍스트 파일은 DataFrame으로 읽혀지기 때문에 pandas의 매우 중요한 데이터 구조입니다.

```
>>> data = { 'name' : ['AA', 'IBM', 'GOOG'],
...         'date' : ['2001-12-01', '2012-02-10', '2010-04-09'],
...         'shares' : [100, 30, 90],
...         'price' : [12.3, 10.3, 32.2]
... }

>>> df = pd.DataFrame(data)
>>> type(df)
<class 'pandas.core.frame.DataFrame'>

>>> df
         date  name  price  shares
0  2001-12-01    AA   12.3     100
1  2012-02-10   IBM   10.3      30
2  2010-04-09  GOOG   32.2      90
```

아래와 같이 DataFrame을 정의한 후 열을 추가할 수 있습니다.

```
>>> df['owner'] = 'Unknown'
>>> df
         date  name  price  shares    owner
0  2001-12-01    AA   12.3     100  Unknown
1  2012-02-10   IBM   10.3      30  Unknown
2  2010-04-09  GOOG   32.2      90  Unknown
```

현재 행 인덱스는 0, 1, 2로 설정되어 있습니다. 이는 'index' 속성을 사용하여 아래와 같이 변경할 수 있습니다.

```
>>> df.index = ['one', 'two', 'three']
>>> df
             date  name  price  shares    owner
one    2001-12-01    AA   12.3     100  Unknown
two    2012-02-10   IBM   10.3      30  Unknown
three  2010-04-09  GOOG   32.2      90  Unknown
```

또한 DataFrame의 모든 컬럼은 아래와 같이 'set_index()' 속성을 사용하여 인덱스로 설정할 수 있습니다.

```
>>> df = df.set_index(['name'])
>>> df
            date  price  shares    owner
name
AA    2001-12-01   12.3     100  Unknown
IBM   2012-02-10   10.3      30  Unknown
GOOG  2010-04-09   32.2      90  Unknown
```

데이터는 행 및 열 인덱스를 사용하는 두 가지 방법으로 액세스할 수 있습니다.

```
>>> # access data using column-index
>>> df['shares']
name
AA      100
IBM      30
GOOG     90
Name: shares, dtype: int64

>>> # access data by row-index
>>> df.ix['AA']
date      2001-12-01
price           12.3
shares           100
owner        Unknown
Name: AA, dtype: object
```

```
>>> # access all rows for a column
>>> df.ix[:, 'name']
0      AA
1     IBM
2    GOOG
Name: name, dtype: object

>>> # access specific element from the DataFrame,
>>> df.ix[0, 'shares']
100
```

모든 열은 'del' 또는 'drop' 명령을 사용하여 삭제할 수 있습니다.

```
>>> del df['owner']
>>> df
            date  price  shares
name
AA    2001-12-01   12.3     100
IBM   2012-02-10   10.3      30
GOOG  2010-04-09   32.2      90

>>> df.drop('shares', axis = 1)
            date  price
name
AA    2001-12-01   12.3
IBM   2012-02-10   10.3
GOOG  2010-04-09   32.2
```

Chapter 2. Overview

이 장에서는 Pandas의 다양한 기능을 예제와 함께 보여주며, 이에 대해서는 이후 장에서도 설명합니다.

● Note

• CSV 파일은 아래 링크에서 다운로드할 수 있습니다.

• https://bitbucket.org/pythondsp/pandasguide/downloads/

2.1 Reading files

이 섹션에서는 'titles.csv'와 'cast.csv'라는 두 개의 데이터 파일이 사용됩니다. 'titles.csv' 파일에는 개봉연도의 영화 목록이 포함되어 있습니다. 반면 'cast.csv' 파일에는 아래와 같이 영화 제목, 개봉연도, 출연배우, 유형(배우/배우), 캐릭터 및 배우 등급을 저장하는 5개의 열이 있습니다.

```
>>> import pandas as pd

>>> casts = pd.read_csv('cast.csv', index_col=None)
>>> casts.head()
                 title  year      name   type                character     n
0       Closet Monster  2015  Buffy #1  actor                  Buffy 4  31.0
1      Suuri illusioni  1985    Homo $  actor                   Guests  22.0
2   Battle of the Sexes  2017   $hutter  actor           Bobby Riggs Fan  10.0
3  Secret in Their Eyes  2015   $hutter  actor           2002 Dodger Fan   NaN
4            Steve Jobs  2015   $hutter  actor  1988 Opera House Patron   NaN

>>> titles = pd.read_csv('titles.csv', index_col =None)
>>> titles.tail()
                     title  year
49995                Rebel  1970
49996              Suzanne  1996
49997                Bomba  2013
49998  Aao Jao Ghar Tumhara  1984
49999            Mrs. Munck  1995
```

• read_csv : csv 파일에서 데이터를 읽습니다.

• index_col = None : 인덱스가 없습니다. 즉, 첫 번째 열이 데이터입니다.

• head() : DataFrame의 처음 5개 요소만 표시

• tail() : DataFrame의 마지막 5개 요소만 표시

인코딩으로 인해 파일을 읽는 동안 오류가 발생하면 다음 옵션도 시도하십시오.

```
titles = pd.read_csv('titles.csv', index_col=None, encoding='utf-8')
```

DataFrame의 이름을 간단히 입력하면(즉, 아래 코드에 cast) 전체 열 목록과 함께 파일의 처음 30개 행과 마지막 20개 행이 표시됩니다. 아래와 같이 'set_options'를 사용하여 제한할 수 있습니다. 또한 테이블의 끝에 행과 열의 총 수가 표시됩니다.

```
>>> pd.set_option('max_rows', 10, 'max_columns', 10)
>>> titles
                        title  year
0                The Rising Son  1990
1      The Thousand Plane Raid  1969
2             Crucea de piatra  1993
3                      Country  2000
4                   Gaiking II  2011
...                        ...   ...
49995                   Rebel  1970
49996                 Suzanne  1996
49997                   Bomba  2013
49998     Aao Jao Ghar Tumhara  1984
49999                Mrs. Munck  1995

[50000 rows x 2 columns]
```

• len : 'len' 명령을 사용하여 파일의 총 행 수를 볼 수 있습니다.

```
>>> len(titles)
50000
```

● Note

head() 및 tail() 명령을 사용하여 파일의 헤더와 내용을 요약시킬 수 있습니다. 이 두 명령은 각각 파일의 처음 5줄과 마지막 5줄을 표시합니다. 또한 이러한 명령으로 표시할 총 줄 수를 변경할 수 있습니다.

```
>>> titles.head(3)
                    title  year
0          The Rising Son  1990
1  The Thousand Plane Raid  1969
2         Crucea de piatra  1993
```

2.2 Data operations

이 섹션에서는 DataFrame에 대한 다양한 유용한 데이터 작업을 보여줍니다.

2.2.1 Row and column selection

DataFrame의 모든 행이나 열은 열 또는 행의 이름을 전달하여 선택할 수 있습니다. DataFrame에서 하나를 선택하면 1차원이 되므로 Series로 간주합니다.

• ix : 'ix' 명령어를 사용하여 DataFrame에서 행을 선택합니다.

```
>>> t = titles['title']

>>> type(t)
<class 'pandas.core.series.Series'>

>>> t.head()
0             The Rising Son
1    The Thousand Plane Raid
2           Crucea de piatra
3                    Country
4                 Gaiking II
Name: title, dtype: object
>>>

>>> titles.ix[0]
title    The Rising Son
year               1990
Name: 0, dtype: object
>>>
```

2.2.2 Filter Data

DataFrame에서 일부 부울 표현식을 제공하여 데이터를 필터링할 수 있습니다. 예를 들어 아래 코드에서 1985년 이후에 출시된 영화는 DataFrame '제목'에서 필터링되어 새 DataFrame, 즉 after85에 저장됩니다.

```
>>> # movies after 1985
>>> after85 = titles[titles['year'] > 1985]
>>> after85.head()
               title  year
 0    The Rising Son  1990
 2  Crucea de piatra  1993
 3           Country  2000
 4        Gaiking II  2011
 5       Medusa (IV)  2015
>>>
```

참고: 부울 결과를 DataFrame에 전달하면 위 코드와 같이 panda가 True(True 및 False를 표시하는 대신)에 해당하는 모든 결과를 표시합니다. 또한 '&' 및 '|'는 아래와 같이 두 조건을 결합하는 데 사용할 수 있습니다.

아래 코드에서는 1990년(1900-1999년)의 모든 영화가 선택되었습니다. 또한 't = titles'은 단순성을 위해서만 사용됩니다.

```
>>> # display movie in years 1990 - 1999
>>> t = titles
>>> movies90 = t[ (t['year']>=1990) & (t['year']<2000) ]
>>> movies90.head()
                      title  year
0             The Rising Son  1990
2           Crucea de piatra  1993
12  Poka Makorer Ghar Bosoti  1996
19          Maa Durga Shakti  1999
24      Conflict of Interest  1993
>>>
```

2.2.3 Sorting

정렬은 'sort_index' 또는 'sort_values' 키워드를 사용하여 수행할 수 있으며,

```
>>> # find all movies named as 'Macbeth'
>>> t = titles
>>> macbeth = t[ t['title'] == 'Macbeth']
>>> macbeth.head()
        title  year
4226   Macbeth  1913
9322   Macbeth  2006
11722  Macbeth  2013
17166  Macbeth  1997
25847  Macbeth  1998
```

위의 필터링 작업에서 데이터는 인덱스별로 정렬됩니다. 즉, 기본적으로 'sort_index' 작업이 아래와 같이 사용됩니다.

```
>>> # by default, sort by index i.e. row header
>>> macbeth = t[ t['title'] == 'Macbeth'].sort_index()
>>> macbeth.head()
        title  year
4226   Macbeth  1913
9322   Macbeth  2006
11722  Macbeth  2013
17166  Macbeth  1997
25847  Macbeth  1998
>>>
```

데이터를 값으로 정렬하려면 'sort_value' 옵션을 사용할 수 있습니다. 아래 코드에서 데이터는 현재 연도별로 정렬됩니다.

```
>>> # sort by year
>>> macbeth = t[ t['title'] == 'Macbeth'].sort_values('year')
>>> macbeth.head()
        title  year
4226   Macbeth  1913
17166  Macbeth  1997
25847  Macbeth  1998
9322   Macbeth  2006
11722  Macbeth  2013
>>>
```

2.2.4 Null values

다양한 열에는 일반적으로 NaN으로 채워지는 값이 없을 수 있습니다. 예를 들어, cast의 3-4 행은 아래와 같이 NaN입니다.

```
>>> casts.ix[3:4]
                 title  year     name   type                character   n
3  Secret in Their Eyes  2015  $hutter  actor          2002 Dodger Fan NaN
4           Steve Jobs  2015  $hutter  actor  1988 Opera House Patron NaN
```

이러한 null 값은 쉽게 선택하거나 선택을 취소하거나 내용을 다른 값으로 대체할 수 있습니다. 빈 문자열 또는 0 등 이 섹션에는 null 값의 다양한 예가 나와 있습니다.

• 'isnull' 명령은 행에 null 값이 있는 경우 true 값을 반환합니다. 따라서 3-4행은 NaN 값을 가지므로 True로 표시됩니다.

```
>>> c = casts
>>> c['n'].isnull().head()
0    False
1    False
2    False
3     True
4     True
Name: n, dtype: bool
```

• 'notnull'은 isnull의 반대이며 null이 아닌 값에 대해 true를 반환합니다.

```
>>> c['n'].notnull().head()
0     True
1     True
2     True
3    False
4    False
Name: n, dtype: bool
```

• null 값이 있는 행을 표시하려면 DataFrame에서 조건을 전달해야 합니다.

```
>>> c[c['n'].isnull()].head(3)
                    title  year     name   type                 character   n
3    Secret in Their Eyes  2015  $hutter  actor           2002 Dodger Fan NaN
4              Steve Jobs  2015  $hutter  actor   1988 Opera House Patron NaN
5   Straight Outta Compton 2015  $hutter  actor               Club Patron NaN
>>>
```

• NaN 값은 fillna, ffill(forward fill), bfill(backward fill) 등을 사용하여 채울 수 있습니다. 아래 코드에서 'NaN' 값은 NA로 대체됩니다. 또한 ffill 및 bfill의 예는 튜토리얼의 후반부에 나와 있습니다.

```
>>> c_fill = c[c['n'].isnull()].fillna('NA')
>>> c_fill.head(2)
                 title  year     name   type                character   n
3  Secret in Their Eyes  2015  $hutter  actor          2002 Dodger Fan  NA
4           Steve Jobs  2015  $hutter  actor  1988 Opera House Patron  NA
```

2.2.5 String operations

'.str.' 옵션을 사용하여 다양한 문자열 연산을 수행할 수 있습니다. 먼저 영화 'Maa'를 검색해 보겠습니다.

```
>>> t = titles
>>> t[t['title'] == 'Maa']
      title  year
38880   Maa  1968
>>>
```

목록에 단 하나의 영화가 있습니다. 이제 'Maa'로 시작하는 모든 영화를 검색하려고 합니다. 아래와 같은 쿼리에는 '.str.' 옵션이 필요합니다.

```
>>> t[t['title'].str.startswith("Maa ")].head(3)
                 title  year
19     Maa Durga Shakti  1999
3046      Maa Aur Mamta  1970
7470  Maa Vaibhav Laxmi  1989
>>>
```

2.2.6 Count Values

총 발생 횟수는 'value_counts()' 옵션을 사용하여 계산할 수 있습니다. 다음 코드에서는 연도를 기준으로 총 영화 수를 표시합니다.

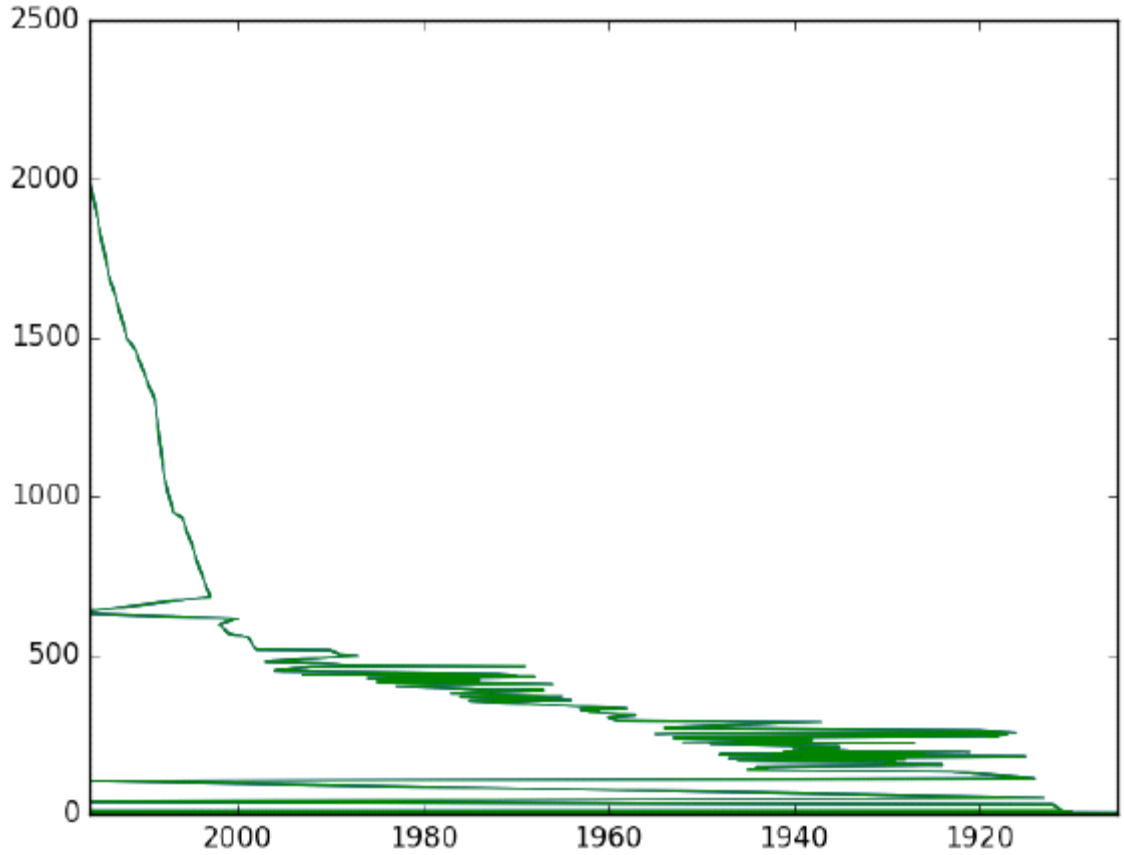
```
>>> t['year'].value_counts().head()
2016    2363
2017    2138
2015    1849
2014    1701
2013    1609
Name: year, dtype: int64
```

2.2.7 Plots

Pandas는 matplotlib 라이브러리를 지원하며 데이터를 그리는 데에도 사용할 수 있습니다. 이전 섹션에서 연간 영화의 총 수는 DataFrame에서 필터링 되었습니다. 아래 코드에서 해당 값은 새 DataFrame에 저장되고 팬더를 사용하여 플롯됩니다.

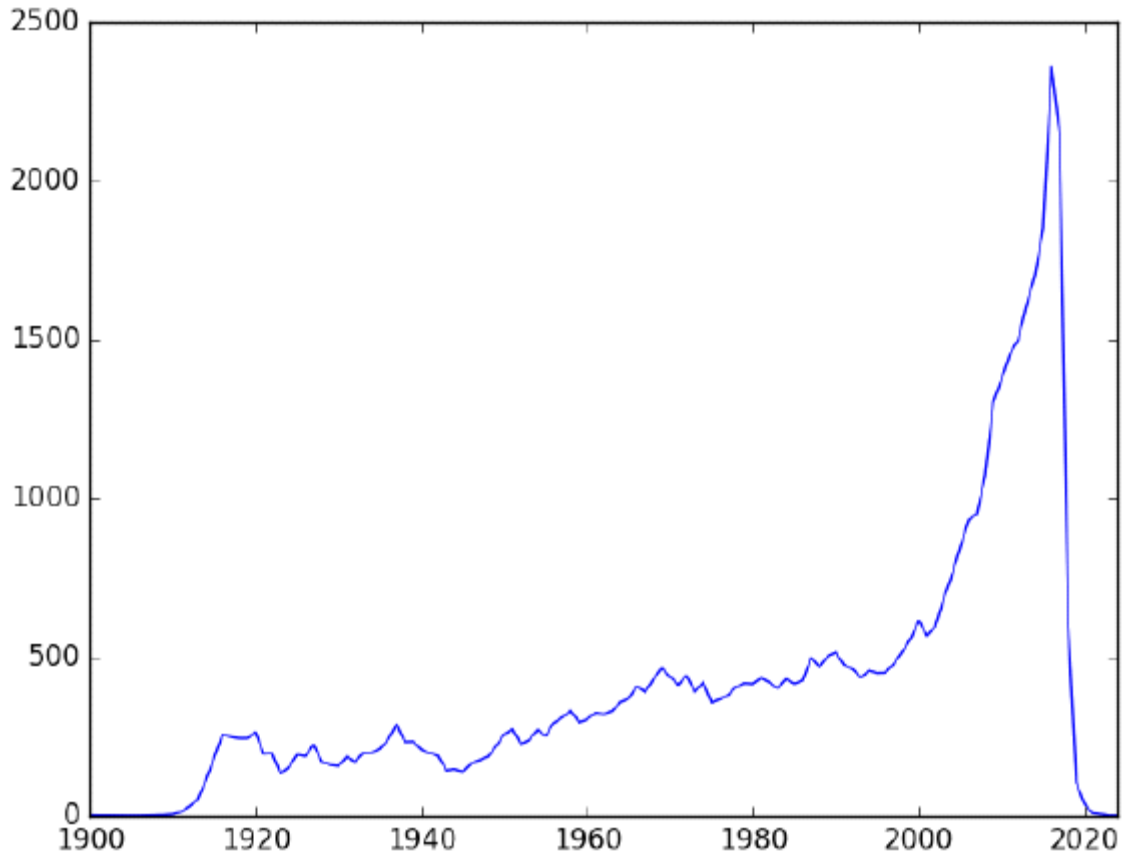
```
>>> import matplotlib.pyplot as plt
>>> t = titles
>>> p = t['year'].value_counts()
>>> p.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xaf18df6c>
>>> plt.show()
```

다음 플롯은 유용한 정보를 제공하지 않는 위의 코드에서 생성됩니다.



먼저 연도(즉, 인덱스)를 정렬한 다음 데이터를 아래와 같이 플롯하는 것이 좋습니다. 여기서 줄거리는 영화의 수가 매년 증가하고 있음을 보여줍니다.

```
>>> p.sort_index().plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xa9cd134c>
>>> plt.show()
```



이제 그래프는 몇 가지 유용한 정보를 제공합니다. 즉, 영화 수가 매년 증가하고 있습니다.

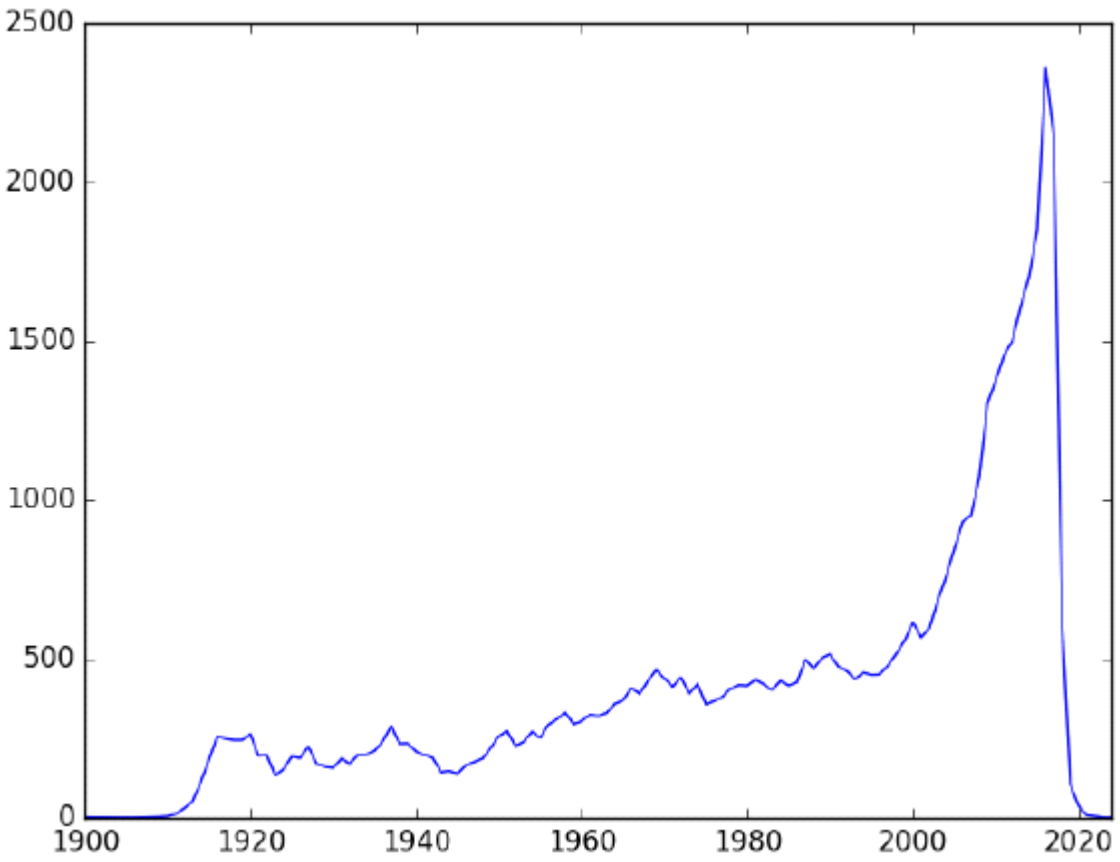
2.3 Groupby

데이터는 열의 headers로 그룹화할 수 있습니다. 또한 사용자 지정 형식을 정의하여 DataFrame의 다양한 요소를 그룹화할 수 있습니다.

2.3.1 Groupby with column-names

Count Values에서는 'count_values()' 메서드를 사용하여 영화/연도 값을 계산했습니다. 'groupby' 방식으로도 동일하게 달성할 수 있습니다. 'groupby' 명령은 개체를 반환하고 일부 결과를 얻으려면 개체에 추가 기능이 필요합니다. 예를 들어 아래 코드에서는 데이터를 'year'로 그룹화한 후 size() 명령을 사용합니다. size() 옵션은 각 연도의 총 행 수를 계산합니다. 따라서 아래 코드의 결과는 'count_values()' 명령과 동일합니다.

```
>>> cg = c.groupby(['year']).size()
>>> cg.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xa9f14b4c>
>>> plt.show()
>>>
```



• 또한 groupby 옵션은 그룹화를 위해 여러 매개변수를 사용할 수 있습니다. 예를 들어 연도를 기준으로 배우 'Aaron Abrams'의 영화를 그룹화하려고 합니다.

```
>>> c = casts
>>> cf = c[c['name'] == 'Aaron Abrams']
>>> cf.groupby(['year']).size().head()
year
2003    2
2004    2
2005    2
2006    1
2007    2
dtype: int64
>>>
```

위 목록은 2003년이 'Aaron Abrams'라는 이름 항목과 함께 두 행에서 발견되었음을 보여줍니다. 즉, 그는 2003년에 2편의 영화를 찍었다.

• 다음으로 영화 목록도 보고 싶은 경우 아래와 같이 목록에 두 개의 매개변수를 전달할 수 있습니다.

```
>>> cf.groupby(['year', 'title']).size().head()
year  title
2003  The In-Laws                              1
      The Visual Bible: The Gospel of John     1
2004  Resident Evil: Apocalypse                1
      Siblings                                 1
2005  Cinderella Man                           1
dtype: int64
>>>
```

위의 코드에서 groupby 연산은 먼저 'year'에 수행된 다음 'title'에 수행됩니다. 즉, 먼저 모든 영화가 연도별로 그룹화됩니다. 그 후, 이 groupby의 결과는 제목을 기준으로 다시 그룹화됩니다. 첫 번째 그룹 명령은 2003년, 2004년, 2005년 등의 순서로 연도를 정렬했습니다. 그런 다음 그룹 명령은 제목을 알파벳 순서로 정렬했습니다.

• 다음으로 연도별 최대 등급을 기준으로 그룹화를 수행하려고 합니다. 즉, 항목을 연도별로 그룹화하고 해당 연도의 최대 등급을 보고자 합니다.

```
>>> c.groupby(['year']).n.max().head()
year
1912     6.0
1913    14.0
1914    39.0
1915    14.0
1916    35.0
Name: n, dtype: float64
```

위의 결과는 1912년의 최대 등급이 Aaron Abrams에 대해 6임을 보여줍니다.
• 마찬가지로 최소 등급을 확인할 수 있습니다.

```
>>> c.groupby(['year']).n.min().head()
year
1912    6.0
1913    1.0
1914    1.0
1915    1.0
1916    1.0
Name: n, dtype: float64
```

• 마지막으로 매년 평균등급을 확인합니다.

```
>>> c.groupby(['year']).n.mean().head()
year
1912    6.000000
1913    4.142857
1914    7.085106
1915    4.236111
1916    5.037736
Name: n, dtype: float64
```

2.3.2 Groupby with custom field

수십 년을 기준으로 데이터를 그룹화하고 사용자 지정 groupby 필드를 만들어야 한다고 가정합니다.

```
>>> # decade conversion : 1985//10 = 198, 198*10 = 1980
>>> decade = c['year']//10*10
>>> c_dec = c.groupby(decade).n.size()
>>>
>>> c_dec.head()
year
1910     669
1920    1121
1930    3448
1940    3997
1950    3892
dtype: int64
```

위의 결과는 각 10년 동안의 총 영화 수를 보여줍니다.

2.4 Unstack

언스택을 이해하기 전에 cast.csv 파일에서 한 가지 사례를 살펴보겠습니다. 다음 코드에서 데이터는 10년 및 유형(예: actor and actress)별로 그룹화됩니다.

```
>>> c = casts
>>> c.groupby( [c['year']//10*10, 'type'] ).size().head(8)
year  type
1910  actor       384
      actress     285
1920  actor       710
      actress     411
1930  actor      2628
      actress     820
1940  actor      3014
      actress     983
dtype: int64
>>>
```

참고: Unstack은 Unstack 데이터 섹션에서 자세히 설명합니다.

이제 우리는 각 10년의 배우와 여배우의 총 수를 비교하고 플롯하려고 합니다. 이 문제에 대한 한 가지 해결책은 짝수 행과 홀수 행을 별도로 잡고 데이터를 그리는 것입니다. new-actor, new-actress 및 teen-actors 등. 이러한 문제에 대한 간단한 해결책은 아래와 같이 그룹화된 Dataframe을 기반으로 새로운 DataFrame을 생성할 수 있는 'unstack'입니다.

• 배우와 배우를 기반으로 한 플롯을 원하므로 먼저 아래와 같이 '유형'에 따라 데이터를 그룹화해야 합니다.

```
>>> c = casts
>>> c_decade = c.groupby( ['type', c['year']//10*10] ).size()
>>> c_decade
type     year
actor    1910      384
         1920      710
         1930     2628
         [...]
actress  1910      285
         1920      411
         1930      820
         [...]
dtype: int64
>>>
```

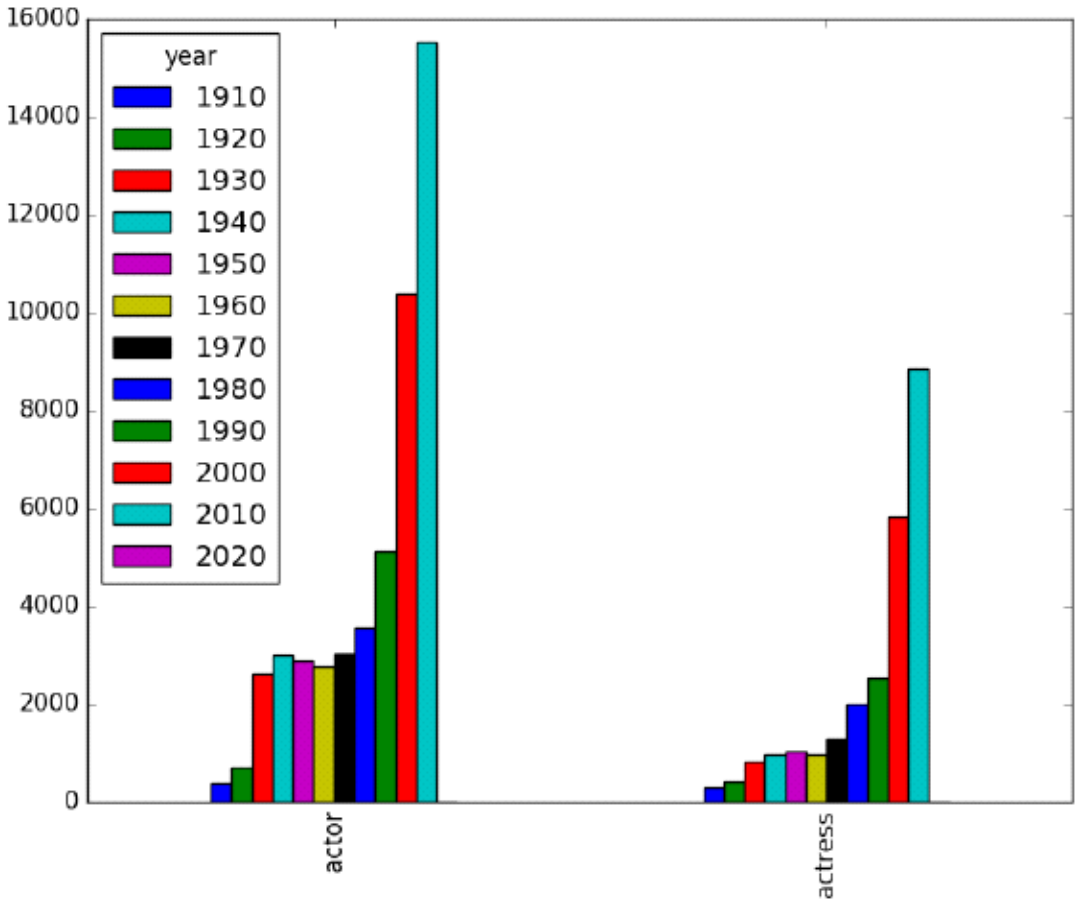
• 이제 'unstack' 명령을 사용하여 새로운 DataFrame을 생성할 수 있습니다. 'unstack' 명령은 인덱스를 기반으로 새로운 DataFrame을 생성합니다.

```
>>> c_decade.unstack()
year     1910  1920  1930  1940  1950  1960  1970  1980  1990  [...]
type
actor     384   710  2628  3014  2877  2775  3044  3565  5108  [...]
actress   285   411   820   983  1015   968  1299  1989  2544  [...]
```

• 다음 명령을 사용하여 위의 데이터를 플롯하고,

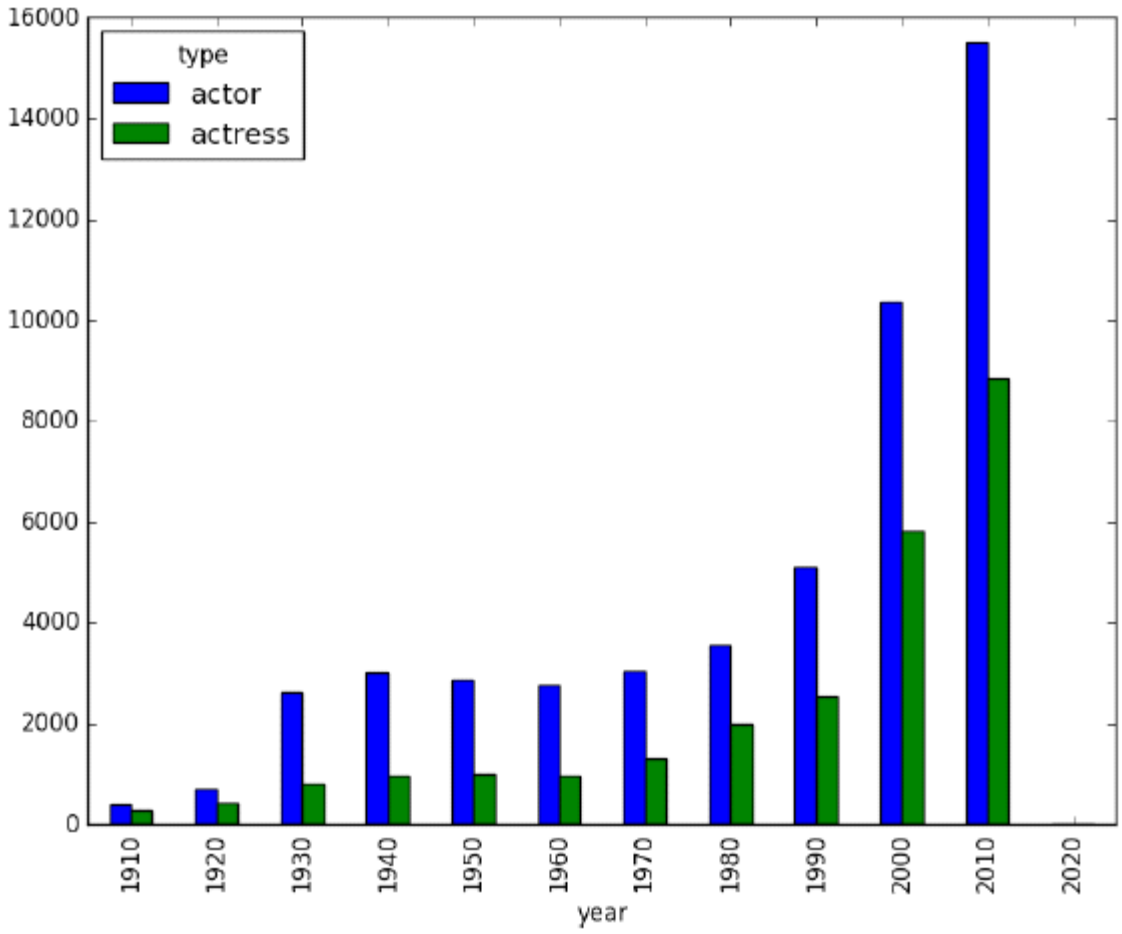
```
>>> c_decade.unstack().plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xb1cec56c>
>>> plt.show()
>>> c_decade.unstack().plot(kind='bar')
<matplotlib.axes._subplots.AxesSubplot object at 0xa8bf778c>
>>> plt.show()
```

위의 명령에서 아래 그림이 생성됩니다. 플롯에서 배우와 여배우는 그룹에서 별도로 플롯됩니다.



• 데이터를 나란히 표시하려면 아래와 같이 unstack(0) 옵션을 사용합니다(기본적으로 unstack(-1)이 사용됨).

```
>>> c_decade.unstack(0)
type  actor  actress
year
1910    384      285
1920    710      411
1930   2628      820
1940   3014      983
1950   2877     1015
1960   2775      968
1970   3044     1299
1980   3565     1989
1990   5108     2544
2000  10368     5831
2010  15523     8853
2020      4        3

>>> c_decade.unstack(0).plot(kind='bar')
<matplotlib.axes._subplots.AxesSubplot object at 0xb1d218cc>
>>> plt.show()
```



2.5 Merge

일반적으로 동일한 프로젝트의 다른 데이터는 다양한 파일에서 사용할 수 있습니다. 이러한 파일에서 유용한 정보를 얻으려면 이러한 파일을 결합해야 합니다. 또한 특정 정보를 얻으려면 동일한 파일의 다른 데이터를 병합해야 합니다. 이 섹션에서는 이 두 가지 병합, 즉 다른 파일과 병합하고 동일한 파일과 병합하는 것을 이해할 것입니다.

2.5.1 Merge with different files

이 섹션에서는 'release_dates.csv'와 'cast.csv'라는 두 테이블의 데이터를 병합합니다. 'release_dates.csv' 파일에는 여러 국가의 영화 개봉일이 포함되어 있습니다.

• 먼저 'cast.csv'에 나열된 일부 영화의 개봉일이 포함된 'release_dates.csv' 파일을 로드합니다. 다음은 'release_dates.csv' 파일의 내용입니다.

```
>>> release = pd.read_csv('release_dates.csv', index_col=None)
>>> release.head()
                    title  year      country        date
0   #73, Shaanthi Nivaasa  2007        India  2007-06-15
1                 #Beings  2015      Romania  2015-01-29
2              #Declimax  2018  Netherlands  2018-01-21
3  #Ewankosau saranghaeyo  2015  Philippines  2015-01-21
4                 #Horror  2015          USA  2015-11-20


>>> casts.head()
                 title  year      name   type                character     n
0       Closet Monster  2015  Buffy #1  actor                  Buffy 4  31.0
1      Suuri illusioni  1985    Homo $  actor                   Guests  22.0
2   Battle of the Sexes  2017   $hutter  actor           Bobby Riggs Fan  10.0
3  Secret in Their Eyes  2015   $hutter  actor           2002 Dodger Fan   NaN
4            Steve Jobs  2015   $hutter  actor  1988 Opera House Patron   NaN
```

• 영화 'Amelia' 개봉일을 알아보자. 이를 위해 먼저 아래와 같이 DataFrame 'cast'에서 Amelia를 필터링합니다. 영화 Amelia에 대한 항목은 두 개뿐입니다.

```
>>> c_amelia = casts[ casts['title'] == 'Amelia']
>>> c_amelia.head()
       title  year            name   type    character     n
5767   Amelia  2009    Aaron Abrams  actor  Slim Gordon   8.0
23319  Amelia  2009  Jeremy Akerman  actor      Sheriff  19.0
>>>
```

• 다음으로 영화 'Amelia'의 개봉일을 DataFrame ‘release’에서 아래와 같이 알아보겠습니다. 아래 결과에서 우리는 영화에 대해 1966년과 2009년이라는 두 개의 다른 출시 연도가 있음을 알 수 있습니다.

```
>>> release [ release['title'] == 'Amelia' ].head()
       title  year    country        date
20543  Amelia  1966     Mexico  1966-03-10
20544  Amelia  2009     Canada  2009-10-23
20545  Amelia  2009        USA  2009-10-23
20546  Amelia  2009  Australia  2009-11-12
20547  Amelia  2009  Singapore  2009-11-12
>>>
```

• cast DataFrame에 Amelia-1966에 대한 항목이 없으므로 merge 명령은 Amelia-1966 release 데이터의 날짜를 병합하지 않습니다. 다음 결과에서 Amelia 2009 release 데이터의 날짜만 DataFrame 캐스트와 병합되었음을 알 수 있습니다.

```
>>> c_amelia.merge(release).head()
    title  year          name   type    character    n    country        date
0  Amelia  2009  Aaron Abrams  actor  Slim Gordon  8.0     Canada  2009-10-23
1  Amelia  2009  Aaron Abrams  actor  Slim Gordon  8.0        USA  2009-10-23
2  Amelia  2009  Aaron Abrams  actor  Slim Gordon  8.0  Australia  2009-11-12
3  Amelia  2009  Aaron Abrams  actor  Slim Gordon  8.0  Singapore  2009-11-12
4  Amelia  2009  Aaron Abrams  actor  Slim Gordon  8.0    Ireland  2009-11-13
```

2.5.2 Merge table with itself

영화의 공동 배우 목록을 보고 싶다고 가정해 보겠습니다. 이를 위해 아래와 같이 제목과 연도를 기준으로 테이블을 자체적으로 병합해야 합니다. 아래 코드에는 배우 'Aaron Abrams'의 공동 주연이 표시되어 있습니다.

• 먼저 'Aaron Abrams'에 대한 결과를 필터링합니다.

```
>>> c = casts[ casts['name']=='Aaron Abrams' ]
>>> c.head(2)
                  title  year          name   type       character    n
5765       #FromJennifer  2017  Aaron Abrams  actor  Ralph Sinclair  NaN
5766  388 Arletta Avenue  2011  Aaron Abrams  actor            Alex  4.0
>>>
```

• 다음으로, 공동 주연을 찾으려면 '제목'과 '연도'를 기준으로 DataFrame을 자체적으로 병합합니다. 즉, 공동 주연이 되려면 영화 이름과 연도가 같아야 합니다.

• 대괄호 안에 c 대신 'cast'가 사용된다는 점에 유의하십시오.

```
c.merge(casts, on=['title', 'year']).head()
```

위의 조인의 문제는 'Aaron Abrams'도 그의 공동 배우로 표시된다는 것입니다(첫 번째 행 참조). 이 문제는 아래와 같이 피할 수 있습니다.

```
c_costar = c.merge (casts, on=['title', 'year'])
c_costar = c_costar[c_costar['name_y'] != 'Aaron Abrams']
c_costar.head()
```

2.6 Index

이전 섹션에서 데이터 정렬 및 플로팅에 인덱스를 사용하는 방법을 살펴보았습니다. 이 섹션에서는 인덱스에 대해 자세히 설명합니다.

인덱스는 pandas에서 매우 중요한 도구입니다. 데이터를 구성하고 데이터에 대한 빠른 액세스를 제공하는 데 사용됩니다. 이 섹션에서는 인덱싱이 있는 데이터와 없는 데이터의 데이터 액세스 시간을 비교합니다. 이 섹션에서는 Jupyter 노트북을 '%%timeit'으로 사용하여 다양한 액세스 작업에 필요한 시간을 비교하는 데 매우 사용하기 쉽습니다.

2.6.1 Creating index

```
import pandas as pd
cast = pd.read_csv('cast.csv', index_col=None)
cast.head()
```

```
%%time

# data access without indexing
cast[cast['title']=='Macbeth']
```

```
CPU times: user 8 ms, sys: 4 ms, total: 12 ms
Wall time: 13.8 ms
```

'%%timeit'은 쉘을 여러 번 실행하고 평균 시간을 표시하므로 보다 정확한 결과를 위해 사용할 수 있습니다. 그러나 쉘의 출력은 표시되지 않습니다.

```
%%timeit

# data access without indexing
cast[cast['title']=='Macbeth']
```

```
100 loops, best of 3: 9.85 ms per loop
```

'set_index'는 데이터에 대한 인덱스를 만드는 데 사용할 수 있습니다. 아래 코드에서 'title'은 인덱스에 설정되어 있으므로 인덱스 번호는 'title'으로 대체됩니다(첫 번째 열 참조).

```
# below line will not work for multiple index
# c = cast.set_index('title')

c = cast.set_index(['title'])
c.head(4)
```

위의 인덱싱을 사용하려면 빠른 작업을 위해 '.loc'을 사용해야 하며,

```
%%time

# data access with indexing
# note that there is minor performance improvement
c.loc['Macbeth']
```

```
CPU times: user 36 ms, sys: 0 ns, total: 36 ms
Wall time: 36.2 ms
```

```
%%timeit

# data access with indexing
# note that there is minor performance improvement
c.loc['Macbeth']
```

```
100 loops, best of 3: 5.64 ms per loop
```

** 인덱스를 정렬하면 속도가 더 빨라지기 때문에 인덱싱을 사용하여 성능이 향상됨(즉, 11ms에서 6ms)을 볼 수 있습니다. **
다음으로 인덱스를 정렬하고 필터 작업을 수행합니다.

```
cs = cast.set_index(['title']).sort_index()
cs.tail(4)
```

```
%%time

# data access with indexing
# note that there is huge performance improvement
cs.loc['Macbeth']
```

```
CPU times: user 36 ms, sys: 0 ns, total: 36 ms
Wall time: 38.8 ms
```

이제 아래 결과와 같이 약 '0.5ms'(4ms가 아닌)에 필터링이 완료됩니다.

```
%%timeit

# data access with indexing
# note that there huge performance improvement
cs.loc['Macbeth']
```

```
1000 loops, best of 3: 480 µs per loop
```

2.6.2 Multiple index

또한 데이터에 여러 인덱스를 가질 수 있습니다.

```
# data with two index i.e. title and n
cm = cast.set_index(['title', 'n']).sort_index()
cm.tail(30)
```

```
>>> cm.loc['Macbeth']
      year                 name     type        character
n
 4.0   1916  Spottiswoode Aitken    actor           Duncan
 6.0   1916           Mary Alden  actress     Lady Macduff
 18.0  1948       William Alland    actor  Second Murderer
 21.0  1997         Stevie Allen    actor         Murderer
NaN    2015       Darren Adamson    actor          Soldier
NaN    1948          Robert Alan    actor   Third Murderer
NaN    2016        John Albasiny    actor           Doctor
NaN    2014          Moyo Akand?  actress            Witch
```

위의 결과에서 'title'이 인덱스 목록에서 제거되어 필터링에 사용할 수 있는 인덱스 수준이 하나 더 있음을 나타냅니다. 두 번째 인덱스로도 데이터를 다시 필터링할 수 있습니다.

```
# show Macbeth with ranking 4-18
cm.loc['Macbeth'].loc[4:18]
```

일치 데이터가 하나만 있는 경우 Series는 DataFrame 대신에 반환합니다.

```
# show Macbeth with ranking 4
cm.loc['Macbeth'].loc[4]
```

```
year                        1916
name         Spottiswoode Aitken
type                       actor
character                 Duncan
Name: 4.0, dtype: object
```

2.6.3 Reset index

인덱스는 'reset_index' 명령을 사용하여 재설정할 수 있습니다. 다시 'cm' DataFrame을 살펴보자.

```
cm.head(2)
```

'cm' DataFrame에는 두 개의 인덱스가 있습니다. 그리고 이들 중 하나, 즉 n은 'reset_index' 명령을 사용하여 제거됩니다.

```
# remove 'n' from index
cm = cm.reset_index('n')
cm.head(2)
```

2.7 Implement using Python-CSV library

위의 모든 논리는 python-csv 라이브러리를 사용하여 구현할 수도 있습니다. 이 섹션에서는 위 섹션의 일부 논리를 python-csv 라이브러리를 사용하여 다시 구현합니다. 다음 예를 보면 다음과 같이 할 수 있습니다.

python-csv 라이브러리와 비교하여 pandas로 작업하는 것이 얼마나 쉬운지 확인하십시오. 그러나 파이썬 내장 라이브러리가 더 재미있습니다.

2.7.1 Read the file

```
import csv
titles = list(csv.DictReader(open('titles.csv')))
titles[0:5]  # display first 5 rows
```

```
[OrderedDict([('title', 'The Rising Son'), ('year', '1990')]),
OrderedDict([('title', 'The Thousand Plane Raid'), ('year', '1969')]),
OrderedDict([('title', 'Crucea de piatra'), ('year', '1993')]),
OrderedDict([('title', 'Country'), ('year', '2000')]),
OrderedDict([('title', 'Gaiking II'), ('year', '2011')])]
```

```
# display last 5 rows
titles[-5:]
```

```
[OrderedDict([('title', 'Rebel'), ('year', '1970')]),
OrderedDict([('title', 'Suzanne'), ('year', '1996')]),
OrderedDict([('title', 'Bomba'), ('year', '2013')]),
OrderedDict([('title', 'Aao Jao Ghar Tumhara'), ('year', '1984')]),
OrderedDict([('title', 'Mrs. Munck'), ('year', '1995')])]
```

• 별도의 행에 제목과 연도 표시합니다.

```
for k, v in titles[0].items():
    print(k, ':',  v)
```

```
title : The Rising Son
year : 1990
```

2.7.2 Display movies according to year

• 1985년의 모든 영화 표시

```
year85 = [a for a in titles if a['year'] == '1985']
year85[:5]
```

```
[OrderedDict([('title', 'Insaaf Main Karoonga'), ('year', '1985')]),
OrderedDict([('title', 'Vivre pour survivre'), ('year', '1985')]),
OrderedDict([('title', 'Water'), ('year', '1985')]),
OrderedDict([('title', 'Doea tanda mata'), ('year', '1985')]),
OrderedDict([('title', 'Koritsia gia tsibima'), ('year', '1985')])]
```

• 1990~1999년 영화 표시

```
# movies from 1990 to 1999
movies90 = [m for m in titles if (int(m['year']) < int('2000')) and (int(m['year']) > int('1989'))]
movies90[:5]
```

```
[OrderedDict([('title', 'The Rising Son'), ('year', '1990')]),
OrderedDict([('title', 'Crucea de piatra'), ('year', '1993')]),
OrderedDict([('title', 'Poka Makorer Ghar Bosoti'), ('year', '1996')]),
OrderedDict([('title', 'Maa Durga Shakti'), ('year', '1999')]),
OrderedDict([('title', 'Conflict of Interest'), ('year', '1993')])]
```

• 모든 영화 'Macbeth' 찾기

```
# find Macbeth movies
macbeth = [m for m in titles if m['title']=='Macbeth']
macbeth[:3]
```

```
[OrderedDict([('title', 'Macbeth'), ('year', '1913')]),
OrderedDict([('title', 'Macbeth'), ('year', '2006')]),
OrderedDict([('title', 'Macbeth'), ('year', '2013')])]
```

2.7.3 operator.iemgetter

• 연도별로 영화 정렬

```
# sort based on year and display 3
from operator import itemgetter
sorted(macbeth, key=itemgetter('year'))[:3]
```

```
[OrderedDict([('title', 'Macbeth'), ('year', '1913')]),
OrderedDict([('title', 'Macbeth'), ('year', '1997')]),
OrderedDict([('title', 'Macbeth'), ('year', '1998')])]
```

2.7.4 Replace empty string with 0

```
casts = list(csv.DictReader(open('cast.csv')))
```

```
casts[3:5]
```

```
[OrderedDict([('title', 'Secret in Their Eyes'),
    ('year', '2015'),
    ('name', '$hutter'),
    ('type', 'actor'),
    ('character', '2002 Dodger Fan'),
    ('n', '')]),
OrderedDict([('title', 'Steve Jobs'),
    ('year', '2015'),
    ('name', '$hutter'),
    ('type', 'actor'),
    ('character', '1988 Opera House Patron'),
    ('n', '')])]
```

```
# replace '' with 0
cast0 = [{**c, 'n':c['n'].replace('', '0')} for c in casts]
cast0[3:5]
```

```
[{'title': 'Secret in Their Eyes',
    'year': '2015', 'name': '$hutter',
    'type': 'actor', 'character': '2002 Dodger Fan',
    'n': '0'},
{'title': 'Steve Jobs',
'year': '2015', 'name': '$hutter',
'type': 'actor', 'character': '1988 Opera House Patron',
'n': '0'}]
```

• 'Maa'로 시작하는 영화

```
# Movies starts with Maa
maa = [m for m in titles if m['title'].startswith('Maa')]
maa[:3]
```

```
[OrderedDict([('title', 'Maa Durga Shakti'), ('year', '1999')]),
OrderedDict([('title', 'Maarek hob'), ('year', '2004')]),
OrderedDict([('title', 'Maa Aur Mamta'), ('year', '1970')])]
```

2.7.5 collections.Counter

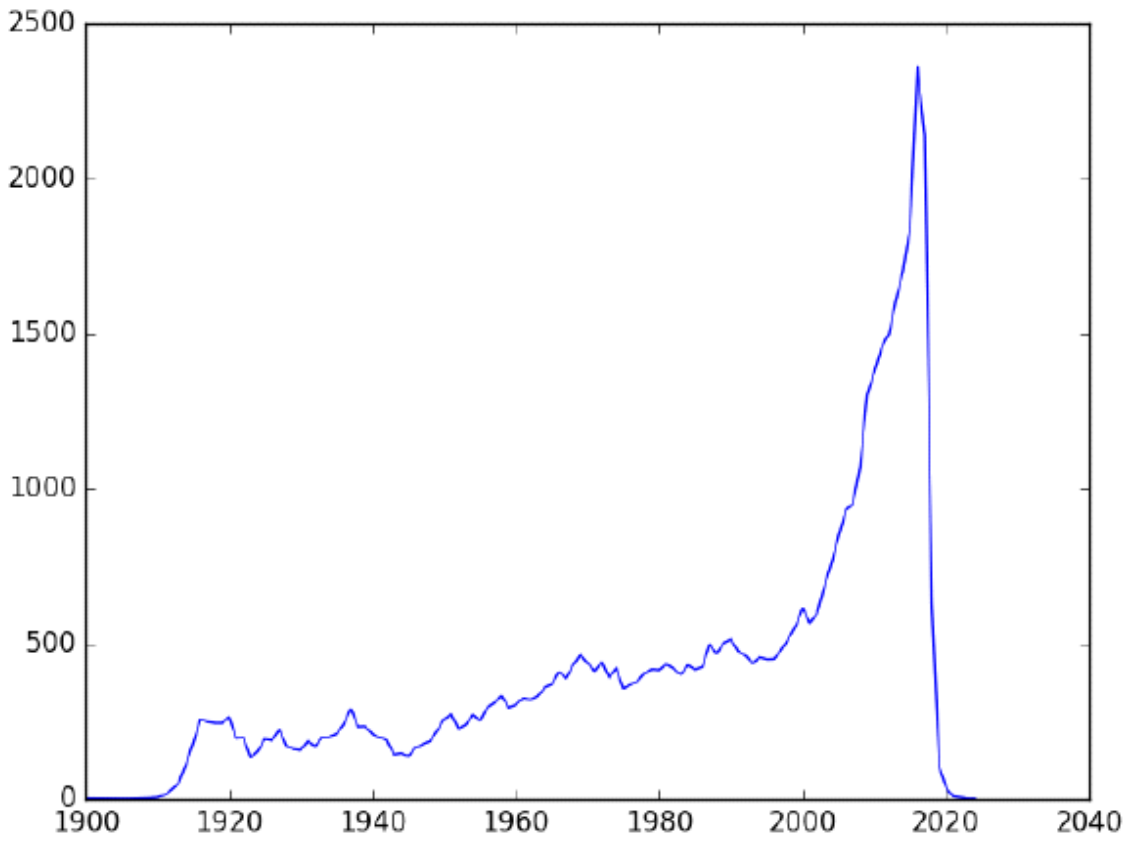
• 연도별로 영화 계산,

```
# Most release movies
from collections import Counter
by_year = Counter(t['year'] for t in titles)
by_year.most_common(3)
# by_year.elements # to see the complete dictionary
```

```
['1990', '1969', '1993', '2000', '2011']
```

• 데이터를 플롯

```
import matplotlib.pyplot as plt
data = by_year.most_common(len(titles))
data = sorted(data)  # sort the data for proper axis
x = [c[0] for c in data]  # extract year
y = [c[1] for c in data]  # extract total number of movies
plt.plot(x, y)
plt.show()
```

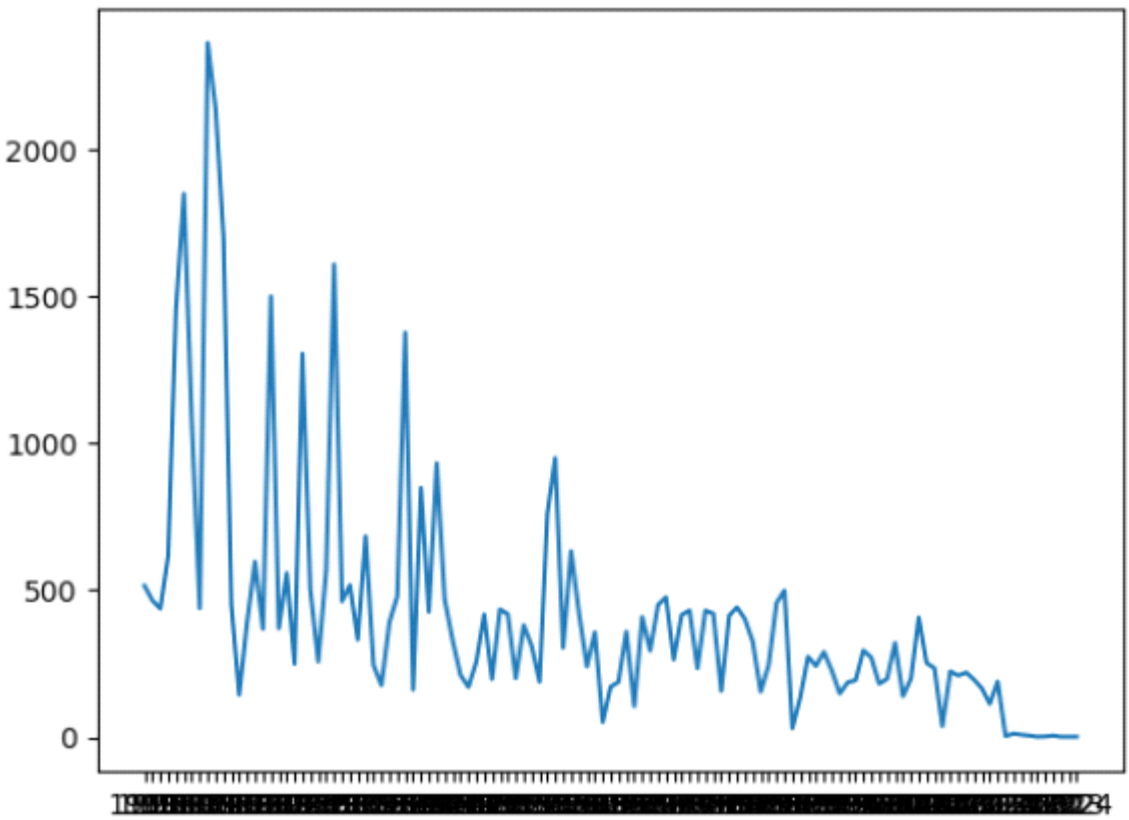


2.7.6 collections.defaultdict

• 연도별로 dictionary에 영화 추가,

```
from collections import defaultdict
```

```
d = defaultdict(list)
for row in titles:
    d[row['year']].append(row['title'])

xx=[]
yy=[]
for k, v in d.items():
    xx.append(k)# = k
    yy.append(len(v))# = len(v)

plt.plot(sorted(xx), yy)
plt.show()
```




```
xx[:5]  # display content of xx
```

```
['1976', '1964', '1914', '1934', '1952']
```

```
yy[:5] # display content of yy
```

```
[515, 465, 437, 616, 1457]
```

• Aaron Abrams의 모든 영화 보기

```
# show all movies of Aaron Abrams
cf = [c for c in casts if c['name']=='Aaron Abrams']
cf[:3]
```

```
[OrderedDict([('title', '#FromJennifer'), ('year', '2017'),
    ('name', 'Aaron Abrams'), ('type', 'actor'),
    ('character', 'Ralph Sinclair'), ('n', '')]),
OrderedDict([('title', '388 Arletta Avenue'), ('year', '2011'),
    ('name', 'Aaron Abrams'), ('type', 'actor'),
    ('character', 'Alex'), ('n', '4')]),
OrderedDict([('title', 'Amelia'), ('year', '2009'),
    ('name', 'Aaron Abrams'), ('type', 'actor'),
    ('character', 'Slim Gordon'), ('n', '8')])]
```

• 연도별로 Aaron Abrams의 모든 영화를 수집

```
# Display movies of Aaron Abrams by year
dcf = defaultdict(list)
for row in cf:
    dcf[row['year']].append(row['title'])

dcf
```

```
defaultdict(<class 'list'>, {
    '2017': ['#FromJennifer', 'The Go-Getters'],
    '2011': ['388 Arletta Avenue', 'Jesus Henry Christ', 'Jesus Henry Christ', 'Take This Waltz', 'The␣
↪Chicago 8'], '2009': ['Amelia', 'At Home by Myself... with You'],
```

(continues on next page)

```
    '2005': ['Cinderella Man', 'Sabah'],
    '2015': ['Closet Monster', 'Regression'],
    '2018': ['Code 8'], '2007': ['Firehouse Dog', 'Young People Fucking'],
    '2008': ['Flash of Genius'], '2013': ['It Was You Charlie'],
    '2004': ['Resident Evil: Apocalypse', 'Siblings'],
    '2003': ['The In-Laws', 'The Visual Bible: The Gospel of John'],
    '2006': ['Zoom']})
```

Chapter 3. Numpy

Numerical Python(Numpy)은 파이썬에서 다양한 수치 계산을 수행하는 데 사용됩니다. Numpy 배열을 사용한 계산은 일반 파이썬 배열보다 빠릅니다. 또한 pandas는 numpy 배열 위에 빌드되므로 파이썬에 대한 더 나은 이해는 pandas를 더 효과적으로 사용하는 데 도움이 될 수 있습니다.

3.1 Creating Arrays

아래 예제와 같이 다차원 배열을 정의하는 것은 numpy에서 매우 쉽습니다.

```
>>> import numpy as np

>>> # 1-D array
>>> d = np.array([1, 2, 3])
>>> type(d)
<class 'numpy.ndarray'>
>>> d
array([1, 2, 3])
>>>

>>> # multi dimensional array
>>> nd = np.array([[1, 2, 3], [3, 4, 5], [10, 11, 12]])
>>> type(nd)
<class 'numpy.ndarray'>
>>> nd
array([[ 1,  2,  3],
       [ 3,  4,  5],
       [10, 11, 12]])
>>> nd.shape  #  shape of array
(3, 3)
>>> nd.dtype  #  data type
dtype('int32')
>>>

>>> #  define zero matrix
>>> np.zeros(3)
array([ 0.,  0.,  0.])
>>> np.zeros([3, 2])
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])

>>> # diagonal matrix
>>> e = np.eye(3)
```

(continued from previous page)

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> # add 2 to e
>>> e2 = e + 2
>>> e2
array([[ 3.,  2.,  2.],
       [ 2.,  3.,  2.],
       [ 2.,  2.,  3.]])


>>> # create matrix with all entries as 1 and size as 'e2'
>>> o = np.ones_like(e2)
>>> o
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

>>> # changing data type
>>> o = np.ones_like(e2)
>>> o.dtype
dtype('float64')
>>> oi = o.astype(np.int32)
>>> oi
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
>>> oi.dtype
dtype('int32')
>>>

>>> # convert string-list to float
>>> a = ['1', '2', '3']
>>> a_arr = np.array(a, dtype=np.string_)  # convert list to ndarray
>>> af = a_arr.astype(float)  # change ndarray type
>>> af
array([ 1.,  2.,  3.])
>>> af.dtype
dtype('float64')
```

3.2 Boolean indexing

부울 인덱싱은 pandas에서 자주 사용되는 numpy의 매우 중요한 기능입니다.

```
>>> # accessing data with boolean indexing
>>> data = np.random.randn(5, 3)
>>> data
array([[ 0.96174001,  1.49352768, -0.31277422],
       [ 0.25044202,  2.35367396,  0.5697222 ],
       [-1.21536074,  0.82088599, -1.85503026],
       [-1.31492648,  1.24546252,  0.27972961],
       [ 0.23487862, -0.20627825,  0.41470205]])
>>> name = np.array(['a', 'b', 'c', 'a', 'b'])
>>> name=='a'
array([ True, False, False,  True, False], dtype=bool)
>>> data[name=='a']
array([[ 0.96174001,  1.49352768, -0.31277422],
       [-1.31492648,  1.24546252,  0.27972961]])

>>> data[name != 'a']
array([[ 0.25044202,  2.35367396,  0.5697222 ],
       [-1.21536074,  0.82088599, -1.85503026],
       [ 0.23487862, -0.20627825,  0.41470205]])
>>> data[(name == 'b')  |  (name=='c')]
array([[ 0.25044202,  2.35367396,  0.5697222 ],
       [-1.21536074,  0.82088599, -1.85503026],
       [ 0.23487862, -0.20627825,  0.41470205]])

>>> data[ (data > 1) & (data < 2) ]
array([ 1.49352768,  1.24546252])
```

3.2 Reshaping arrays

```
>>> a = np.arange(0, 20)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])

>>> # reshape array a
>>> a45 = a.reshape(4, 5)
>>> a45
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])

>>> # select row 2, 0 and 1 from a45 and store in b
>>> b = a45[ [2, 0, 1] ]
>>> b
array([[10, 11, 12, 13, 14],
       [ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]])

>>> # transpose array b
>>> b.T
array([[10,  0,  5],
       [11,  1,  6],
       [12,  2,  7],
       [13,  3,  8],
       [14,  4,  9]])
```

3.4 Concatenating the data

'concatenate' 명령을 사용하여 데이터를 두 개의 배열로 결합할 수 있습니다.

```
>>> arr = np.arange(12).reshape(3,4)
>>> rn = np.random.randn(3, 4)
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> rn
array([[-0.25178434,  0.98443663, -0.99723191, -0.64737102],
       [ 1.29179768, -0.88437251, -1.25608884, -1.60265896],
       [-0.60085171,  0.8569506 ,  0.62657649,  1.43647342]])

>>> # merge data of rn below the arr
>>> np.concatenate([arr, rn])
array([[ 0.        ,  1.        ,  2.        ,  3.        ],
       [ 4.        ,  5.        ,  6.        ,  7.        ],
       [ 8.        ,  9.        , 10.        , 11.        ],
       [-0.25178434,  0.98443663, -0.99723191, -0.64737102],
       [ 1.29179768, -0.88437251, -1.25608884, -1.60265896],
       [-0.60085171,  0.8569506 ,  0.62657649,  1.43647342]])

>>> # merge dataof rn on the right side of the arr
>>> np.concatenate([arr, rn], axis=1)
array([[ 0.        ,  1.        ,  2.        ,  3.        ,
        -0.25178434,  0.98443663, -0.99723191, -0.64737102],
       [ 4.        ,  5.        ,  6.        ,  7.        ,
         1.29179768, -0.88437251, -1.25608884, -1.60265896],
       [ 8.        ,  9.        , 10.        , 11.        ,
        -0.60085171,  0.8569506 ,  0.62657649,  1.43647342]])
>>>
```

Chapter 4. Data processing

데이터 분석 및 모델링에서 대부분의 프로그래밍 작업은 데이터 준비에 사용됩니다. 데이터 로드, 정리 및 재정렬 등. Pandas는 Python 라이브러리와 함께 데이터 처리를 위한 고성능, 유연하고 높은 수준의 환경을 제공합니다.
1장에서 판다의 기초를 보았습니다. 판다에 대한 이해를 돕기 위해 2장에 다양한 예가 나와 있습니다. 3장에서는 numpy의 몇 가지 기본 사항을 제시했습니다. 이 장에서는 데이터를 효과적으로 처리하기 위한 pandas의 기능을 더 살펴보겠습니다.

4.1 Hierarchical indexing

계층적 인덱싱은 우리가 여러 인덱스 수준을 가질 수 있게 해주는 pandas의 중요한 기능입니다. Section Multiple index에서 이미 그 예를 보았습니다. 이 섹션에서는 이러한 인덱싱을 사용하여 인덱싱 및 데이터 액세스에 대해 자세히 알아봅니다.

4.1.1 Creating multiple index

• 다음은 다중 인덱스를 가진 series 예입니다.

```
>>> import pandas as pd
>>> data = pd.Series([10, 20, 30, 40, 15, 25, 35, 25], index = [['a', 'a',
... 'a', 'a', 'b', 'b', 'b', 'b'], ['obj1', 'obj2', 'obj3', 'obj4', 'obj1',
... 'obj2', 'obj3', 'obj4']])
>>> data
a  obj1    10
   obj2    20
   obj3    30
   obj4    40
b  obj1    15
   obj2    25
   obj3    35
   obj4    25
dtype: int64
```

• 여기에는 (a, b) 및 (obj1, ... , obj4)의 두 가지 수준의 인덱스가 있습니다. 인덱스는 아래와 같이 'index' 명령어를 사용하여 볼 수 있습니다.

```
>>> data.index
MultiIndex(levels=[['a', 'b'], ['obj1', 'obj2', 'obj3', 'obj4']],
           labels=[[0, 0, 0, 0, 1, 1, 1, 1], [0, 1, 2, 3, 0, 1, 2, 3]])
```

4.1.2 Partial indexing

계층적 인덱싱에서 특정 인덱스를 선택하는 것을 부분 인덱싱이라고 합니다.

• 아래 코드에서는 데이터에서 인덱스 'b'를 추출하고,

```
>>> data['b']
obj1    15
obj2    25
obj3    35
obj4    25
dtype: int64
```

• 또한 내부 수준(예: 'obj')을 기반으로 데이터를 추출할 수 있습니다. 아래 결과는 series에서 'obj2'에 대해 사용 가능한 두 값을 보여줍니다.

```
>>> data[:, 'obj2']
a    20
b    25
dtype: int64
>>>
```

4.1.3 Unstack the data

Unstack 섹션에서 unstack 작업의 사용을 보았습니다. Unstack은 행 머리글을 열 머리글로 변경합니다. 행 인덱스가 열 인덱스로 변경되었으므로 이 경우 Series가 DataFrame이 됩니다. 다음은 데이터 스택을 해제하는 몇 가지 예입니다.

```
>>> # unstack based on first level i.e. a, b
>>> # note that data row-labels are a and b
>>> data.unstack(0)
       a   b
obj1  10  15
obj2  20  25
obj3  30  35
obj4  40  25

>>> # unstack based on second level i.e. 'obj'
>>> data.unstack(1)
   obj1  obj2  obj3  obj4
a    10    20    30    40
b    15    25    35    25
>>>

>>> # by default innermost level is used for unstacking
>>> d = data.unstack()
>>> d
   obj1  obj2  obj3  obj4
a    10    20    30    40
b    15    25    35    25
```

• 'stack()' 연산은 열 인덱스를 다시 행 인덱스로 변환합니다. 위의 코드에서 DataFrame 'd'는 컬럼 인덱스로 'obj'를 가지고 있으며, 이는 'stack' 연산을 사용하여 행 인덱스로 변환될 수 있습니다.

```
>>> d.stack()
a  obj1    10
   obj2    20
   obj3    30
   obj4    40
b  obj1    15
```

```
   obj2    25
   obj3    35
   obj4    25
dtype: int64
```

4.1.4 Column indexing

열 인덱싱에는 2차원 데이터가 필요하기 때문에 열 인덱싱은 DataFrame에서만 가능합니다(시리즈 제외). 인덱스가 여러 개인 열을 이해하기 위해 아래와 같이 새로운 DataFrame을 생성해 보겠습니다.

```
>>> import numpy as np
>>> df = pd.DataFrame(np.arange(12).reshape(4, 3),
...     index = [['a', 'a', 'b', 'b'], ['one', 'two', 'three', 'four']],
...     columns = [['num1', 'num2', 'num3'], ['red', 'green', 'red']]
... )
>>>
>>> df
        num1  num2 num3
         red green  red
a one      0     1    2
  two      3     4    5
b three    6     7    8
  four     9    10   11
>>>

>>> # display row index
>>> df.index
MultiIndex(levels=[['a', 'b'], ['four', 'one', 'three', 'two']],
           labels=[[0, 0, 1, 1], [1, 3, 2, 0]])

>>> # display column index
>>> df.columns
MultiIndex(levels=[['num1', 'num2', 'num3'], ['green', 'red']],
           labels=[[0, 1, 2], [1, 0, 1]])
```

• 이전 섹션에서는 스택 및 스택 해제 작업에 숫자를 사용했습니다(unstack(0) 등). 아래와 같이 인덱스에 이름을 지정할 수 있습니다.

```
>>> df.index.names=['key1', 'key2']
>>> df.columns.names=['n', 'color']
>>> df
n          num1  num2 num3
color       red green  red
key1 key2
a    one      0     1    2
     two      3     4    5
b    three    6     7    8
     four     9    10   11
```

• 이제 부분 인덱싱 작업을 수행할 수 있습니다. 다음 코드에서는 DataFrame의 데이터에 액세스하는 다양한 방법을 보여줍니다.

```
>>> # accessing the column for num1
>>> df['num1']  #  df.ix[:, 'num1']
color       red
key1 key2
a    one      0
     two      3
b    three    6
     four     9

>>> # accessing the column for a
>>> df.ix['a']
n     num1  num2 num3
color  red green  red
key2
one      0     1    2
two      3     4    5

>>> # access row 0 only
>>> df.ix[0]
n     color
num1  red      0
num2  green    1
num3  red      2
Name: (a, one), dtype: int32
```

4.1.5 Swap and sort level

두 개의 레벨 숫자를 입력으로 사용하는 'swaplevel' 명령을 사용하여 인덱스 레벨을 바꿀 수 있습니다.

```
>>> df.swaplevel('key1', 'key2')
n          num1  num2 num3
color       red green  red
key2  key1
one   a       0     1    2
two   a       3     4    5
three b       6     7    8
four  b       9    10   11
>>>
```

'sort_index' 명령을 사용하여 레벨을 정렬할 수 있습니다. 아래 코드에서 데이터는 'key2' 이름으로 정렬됩니다. 즉, key2는 알파벳순으로 정렬됩니다.

```
>>> df.sort_index(level='key2')
n          num1  num2 num3
color       red green  red
key1 key2
b    four     9    10   11
a    one      0     1    2
b    three    6     7    8
a    two      3     4    5
>>>
```

4.1.6 Summary statistics by level

Groupby 섹션에서 groupby 명령의 예를 보았습니다. Pandas는 아래 표시된 'level'을 사용하여 이러한 작업을 수행하는 몇 가지 더 쉬운 방법을 제공합니다.

```
>>> # add all rows with similar key1 name
>>> df.sum(level = 'key1')
n      num1  num2 num3
color   red green  red
key1
a         3     5    7
b        15    17   19
>>>

>>> # add all the columns based on similar color
>>> df.sum(level= 'color', axis=1)
color       green  red
key1 key2
a    one        1    2
     two        4    8
b    three      7   14
     four      10   20
```

4.2 File operations

이 섹션에서는 파일을 읽고 쓰는 다양한 방법에 대해 설명합니다.

4.2.1. Reading files

Pandas는 다양한 유형의 파일 형식(csv, 텍스트, 엑셀 및 다른 데이터베이스 등)을 지원합니다. 파일은 종종 다른 형식으로도 저장됩니다. 파일에는 머리글, 바닥글 및 주석 등이 포함될 수도 있고 포함되지 않을 수도 있습니다. 따라서 파일의 내용을 처리해야 합니다. Pandas는 파일을 읽는 동안 일반적인 처리 중 일부를 처리할 수 있는 다양한 기능을 제공합니다. 이러한 처리 중 일부가 이 섹션에 나와 있습니다.

• 파일은 아래와 같이 'read_csv', 'read_table' 또는 'DataFrame.from_csv' 옵션을 사용하여 읽을 수 있습니다. 이 모든 메소드의 출력은 동일하지만 파일을 올바르게 읽으려면 다른 매개변수를 제공해야 합니다

다음은 'ex1.csv' 파일의 내용이며,

```
$ cat ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

다음은 다양한 파일 읽기 방법의 출력입니다. 'read_csv'는 파일을 읽기 위한 범용 방법이므로 이 방법은 튜토리얼의 나머지 부분에서 사용됩니다.

```
>>> import pandas as pd

>>> # DataFrame.from_csv
>>> df = pd.DataFrame.from_csv('ex1.csv', index_col=None)
>>> df
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

>>> # read_csv
>>> df = pd.read_csv('ex1.csv')
>>> df
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

>>> # read_table
```

```
>>> df = pd.read_table('ex1.csv', sep=',')
>>> df
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
>>>
```

• 위의 출력에서 헤더는 파일에서 추가됩니다. 그러나 모든 파일에 헤더가 있는 것은 아닙니다. 이 경우 다음과 같이 헤더를 명시적으로 정의해야 합니다.
다음은 'ex2.csv' 파일의 내용입니다.

```
$ cat ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,food
```

헤더가 위의 파일에 없기 때문에 "header" 인수를 명시적으로 제공해야 합니다.

```
>>> import pandas as pd

>>> # set header as none, default values will be used as header
>>> pd.read_csv('ex2.csv', header=None)
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

>>> # specify the header using 'names'
>>> pd.read_csv('ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

>>> # specify the row and column header both
>>> pd.read_csv('ex2.csv', names=['a', 'b', 'c', 'd', 'message'], index_col='message')
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
>>>
```

• 'index_col' 인수에 리스트를 제공하여 계층적 인덱스를 생성할 수 있습니다.
다음은 'csv_mindex.csv' 파일의 내용이며,

```
$ cat csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16
```

계층적 인덱스는 다음과 같이 '키' 값으로 생성할 수 있습니다.

```
>>> pd.read_csv('csv_mindex.csv', index_col=['key1', 'key2'])
             value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

• 일부 파일에는 추가 정보나 설명이 포함될 수 있으므로 데이터 처리를 위해 이러한 정보를 제거해야 합니다. 이것은 'skiprows' 명령을 사용하여 수행할 수 있습니다.
다음은 'ex4.csv' 파일의 내용이며,

```
$ cat ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foodh
```

위 결과에서 줄 0, 2, 3에는 몇 가지 주석이 포함되어 있습니다. 다음과 같이 제거할 수 있습니다.

```
>>> d = pd.read_csv('ex4.csv', skiprows=[0,2,3])
>>> d
    a   b   c   d message
 0  1   2   3   4   hello
 1  5   6   7   8   world
 2  9  10  11  12     foo
```

4.2.2 Writing dadta to a file

'to_csv' 명령은 파일을 저장하는 데 사용됩니다. 다음 코드에서 이전 데이터 'd'는 인덱스가 있는 파일과 인덱스가 없는 d_out.csv 및 d_out2.csv의 두 파일에 저장됩니다.

```
>>> d.to_csv('d_out.csv')

>>> # save without headers
>>> d.to_csv('d_out2.csv', header=False, index=False)
```

위 두 파일의 내용은 아래와 같습니다.

```
$ cat d_out.csv
,a,b,c,d,message
0,1,2,3,4,hello
1,5,6,7,8,world
2,9,10,11,12,foo

$ cat d_out2.csv
0,1,2,3,4,hello
1,5,6,7,8,world
2,9,10,11,12,foo
```

4.3 Merge

병합 또는 조인 작업은 하나 이상의 키를 사용하여 데이터 세트를 결합합니다. '병합' 기능은 데이터에서 이러한 알고리즘을 사용하기 위한 주요 진입점입니다. 다음 예를 통해 이것을 이해합시다.

```
>>> df1 = pd.DataFrame({ 'key' : ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
...                      'data1' : range(7)})

>>> df2 = pd.DataFrame({ 'key' : ['a', 'b', 'd'],
...                      'data2' : range(3)})

>>> df1
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b

>>> df2 = pd.DataFrame({ 'key' : ['a', 'b', 'd', 'b'],
...                      'data2' : range(4)})
>>> df2
   data2 key
0      0   a
1      1   b
2      2   d
3      3   b
>>>
```

4.3.1 Many to one

• '다대일' 병합은 행의 데카르트 곱을 조인합니다. 예: df1과 df2는 각각 'b'의 총 3행과 2행을 가지므로 조인하면 총 6행이 됩니다. 또한 조인을 사용할 때 'on' 키워드를 정의하는 것이 코드의 가독성을 높여줍니다.

```
>>> pd.merge(df1, df2)  # or pd.merge(df1, df2, on='key')
   data1 key  data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      6   b      1
5      6   b      3
6      2   a      0
7      4   a      0
8      5   a      0
>>>
```

• 앞의 경우 두 DataFrame은 동일한 헤더 'key'를 갖습니다. 다음 예제에서는 'left_on' 및 'right_on' 키워드를 사용하여 다른 키를 기반으로 데이터를 결합합니다.

```
>>> # data is same as previous, only 'key' is replaces with 'key1' and 'key2'
>>> df1 = pd.DataFrame({ 'key1' : ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
...                      'data1' : range(7)})
>>> df2 = pd.DataFrame({ 'key2' : ['a', 'b', 'd', 'b'],
...                      'data1' : range(4)})
```

```
>>> pd.merge(df1, df2, left_on='key1', right_on='key2')
   data1_x key1  data1_y key2
0        0    b        1    b
1        0    b        3    b
2        1    b        1    b
3        1    b        3    b
4        6    b        1    b
5        6    b        3    b
6        2    a        0    a
7        4    a        0    a
8        5    a        0    a
>>>
```

4.3.2 Inner and outer join

이전 예에서 DataFrame 'df1' 및 'df2'의 흔하지 않은 항목이 병합에서 누락되었음을 알 수 있습니다. 'd'는 병합된 데이터에 없습니다. 이것은 공통 키만 병합되는 '내부 조인'의 예입니다.

기본적으로 pandas는 내부 조인을 수행합니다. 외부 조인을 수행하려면 'left', 'right' 및 'outer'의 3가지 다른 값을 가질 수 있는 'how' 키워드를 사용해야 합니다. 'left' 옵션은 왼쪽 DataFrame을 취하고 모든 항목을 다른 DataFrame과 병합합니다. 마찬가지로 '오른쪽' 옵션은 오른쪽 DataFrame의 항목을 왼쪽 DataFrame과 병합합니다. 마지막으로 'outer' 옵션은 아래와 같이 두 DataFrame의 모든 항목을 병합합니다. 테이블 조인 후 누락된 항목은 'NaN'으로 표시됩니다.

```
>>> # left join
>>> pd.merge(df1, df2, left_on='key1', right_on='key2', how="left")
   data1_x key1  data1_y key2
0        0    b      1.0    b
1        0    b      3.0    b
2        1    b      1.0    b
3        1    b      3.0    b
4        2    a      0.0    a
5        3    c      NaN  NaN
6        4    a      0.0    a
7        5    a      0.0    a
8        6    b      1.0    b
9        6    b      3.0    b

>>> # right join
>>> pd.merge(df1, df2, left_on='key1', right_on='key2', how="right")
    data1_x key1  data1_y key2
0       0.0    b        1    b
1       1.0    b        1    b
2       6.0    b        1    b
3       0.0    b        3    b
4       1.0    b        3    b
5       6.0    b        3    b
6       2.0    a        0    a
7       4.0    a        0    a
8       5.0    a        0    a
9       NaN  NaN        2    d

>>> # outer join
>>> pd.merge(df1, df2, left_on='key1', right_on='key2', how="outer")
    data1_x key1  data1_y key2
0       0.0    b      1.0    b
1       0.0    b      3.0    b
2       1.0    b      1.0    b
3       1.0    b      3.0    b
4       6.0    b      1.0    b
5       6.0    b      3.0    b
6       2.0    a      0.0    a
7       4.0    a      0.0    a
8       5.0    a      0.0    a
9       3.0    c      NaN  NaN
10      NaN  NaN      2.0    d
```

4.3.3 Concatenating the data

Numpy에서 데이터 연결을 보았습니다. Pandas 연결은 Numpy보다 더 일반화되어 있습니다. 이 섹션에 표시된 대로 그룹화를 시각화하기 위해 레이블링과 함께 데이터의 합집합 또는 교집합을 기반으로 한 연결을 허용합니다.

```
>>> s1 = pd.Series([0, 1], index=['a', 'b'])
>>> s2 = pd.Series([2, 1, 3], index=['c', 'd', 'e'])
>>> s3 = pd.Series([4, 7], index=['a', 'e'])

>>> s1
a    0
b    1
dtype: int64

>>> s2
c    2
d    1
e    3
dtype: int64

>>> s3
a    4
e    7
dtype: int64

>>> # concatenate s1 and s2
>>> pd.concat([s1, s2])
a    0
b    1
c    2
d    1
e    3
dtype: int64

>>> # join on axis 1
>>> pd.concat([s1, s2], axis=1)
     0    1
a  0.0  NaN
b  1.0  NaN
c  NaN  2.0
d  NaN  1.0
e  NaN  3.0
```

• 위의 결과에서 연결 작업의 다른 부분을 식별하기가 어렵습니다. 작업을 식별할 수 있도록 'key'를 제공할 수 있습니다.

```
>>> pd.concat([s1, s2, s3], keys=['one', 'two', 'three'])
one    a    0
       b    1
two    c    2
       d    1
       e    3
three  a    4
       e    7
dtype: int64
```

참고: 위의 연결 작업은 두 데이터 세트의 합집합입니다. 즉, 외부 조인입니다. 데이터 교차에 "join='inner'"를 사용할 수 있습니다.

```
>>> pd.concat([s1, s3], join='inner', axis=1)
   0  1
 a 0  4
```

• DataFrame을 연결하는 방법은 위와 동일합니다. 다음은 DataFrame을 연결한 예입니다. 이 섹션의 시작 부분에 'df1' 및 'df2'가 정의되어 있습니다.

```
>>> pd.concat([df1, df2], join='inner', axis=1, keys=['one', 'two'])
    one        two
  data1 key1 data1 key2
0     0    b     0    a
1     1    b     1    b
2     2    a     2    d
3     3    c     3    b
```

• 연결 작업을 위해 DataFrame도 dictionary로 전달할 수 있습니다. 이 경우 dictionary의 키는 작업을 위한 'key'로 사용됩니다.

```
>>> pd.concat({ 'level1':df1, 'level2':df2}, axis=1, join='inner')
    level1      level2
     data1 key1  data1 key2
  0      0    b      0    a
  1      1    b      1    b
  2      2    a      2    d
  3      3    c      3    b
  >>>
```

4.4 Data transformation

이전 섹션에서 다양한 데이터를 결합하는 다양한 작업을 보았습니다. 다음으로 중요한 단계는 데이터 변환(예: 데이터 정리 및 필터링)입니다. 중복 항목 제거 및 NaN 값 바꾸기 등

4.4.1 Removing duplicates

• 'drop_duplicates' 명령으로 중복 항목을 제거하는 것은 매우 쉽습니다. 또한 'duplicate()' 명령을 사용하여 아래와 같이 중복 항목을 확인할 수 있습니다.

```
>>> # create DataFrame with duplicate entries
>>> df = pd.DataFrame({'k1':['one']*3 + ['two']*4,
...                    'k2':[1,1,2,3,3,4,4]})
>>> df
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4

>>> # see the duplicate entries
>>> df.duplicated()
0    false
1     true
2    false
3    false
4     true
5    false
6     true
dtype: bool

>>> # drop the duplicate entries
>>> df.drop_duplicates()
    k1  k2
0  one   1
2  one   2
3  two   3
5  two   4
```

• 현재 마지막 항목은 drop_duplicates commnad에 의해 제거됩니다. 마지막 항목을 유지하려면 'keep' 키워드를 사용할 수 있습니다.

```
>>> df.drop_duplicates(keep="last")
    k1  k2
1  one   1
2  one   2
4  two   3
6  two   4
>>>
```

• 특정 열을 기반으로 하는 모든 중복 값을 삭제할 수도 있습니다.

```
>>> # drop duplicate entries based on k1 only
>>> df.drop_duplicates(['k1'])
    k1  k2
0  one   1
3  two   3

>>> # drop if k1 and k2 column matched
>>> df.drop_duplicates(['k1', 'k2'])
    k1  k2
0  one   1
2  one   2
3  two   3
5  two   4
>>>
```

4.4.2 Replacing values

아래와 같이 pandas를 사용하여 값을 바꾸는 것은 매우 쉽습니다.

```
>>> # replace 'one' with 'One'
>>> df.replace('one', 'One')
    k1  k2
0  One   1
1  One   1
2  One   2
3  two   3
4  two   3
5  two   4
6  two   4

>>> # replace 'one'->'One'  and 3->30
>>> df.replace(['one', 3], ['One', '30'])
    k1  k2
0  One   1
1  One   1
2  One   2
3  two  30
4  two  30
5  two   4
6  two   4
>>>
```

• 인수도 dictionary 형태로 전달할 수 있습니다.

```
>>> df.replace({'one':'One', 3:30})
    k1  k2
0  One   1
1  One   1
2  One   2
3  two  30
4  two  30
5  two   4
6  two   4
```

4.5 Groupby and data aggregation

4.5.1 Basics

Groupby 섹션에서 다양한 groupby 작업을 보았습니다. 여기에서는 gropby 작업의 몇 가지 추가 기능에 대해 설명합니다.

먼저 DataFrame을 생성해 보겠습니다.

```
>>> df = pd.DataFrame({'k1':['a', 'a', 'b', 'b', 'a'],
...                    'k2':['one', 'two', 'one', 'two', 'one'],
...                    'data1': [2, 3, 3, 2, 4],
...                    'data2': [5, 5, 5, 5, 10]})
>>> df
   data1  data2 k1   k2
0      2      5  a  one
1      3      5  a  two
2      3      5  b  one
3      2      5  b  two
4      4     10  a  one
```

• 이제 'k1'을 기준으로 그룹을 만들고 아래와 같이 평균값을 구합니다. 다음 코드에서 행 (0, 1, 4) 및 (2, 3)은 함께 그룹화됩니다. 따라서 평균값은 3과 2.5입니다.

```
>>> gp1 = df['data1'].groupby(df['k1'])
>>> gp1
<pandas.core.groupby.SeriesGroupBy object at 0xb21f6bcc>
```

```
>>> gp1.mean()
k1
a    3.0
b    2.5
Name: data1, dtype: float64
```

• 그룹화를 위해 여러 매개변수를 전달할 수도 있습니다.

```
>>> gp2 = df['data1'].groupby([df['k1'], df['k2']])
>>> mean = gp2.mean()
>>> mean
k1  k2
a   one    3
    two    3
b   one    3
    two    2
Name: data1, dtype: int64
>>>
```

4.5.2 Iterating over group

• groupby 작업은 group-name과 data라는 두 가지 값으로 튜플을 생성하는 반복을 지원합니다.

```
>>> for name, group in gp1:
...     print(name)
...     print(group)
...
a
0    2
1    3
4    4
Name: data1, dtype: int64
b
2    3
3    2
Name: data1, dtype: int64
```

• groupby 연산이 여러 키를 기반으로 수행되면 키에 대한 튜플도 생성됩니다.

```
>>> for name, group in gp2:
...     print(name)
...     print(group)
...
('a', 'one')
0    2
4    4
Name: data1, dtype: int64
('a', 'two')
1    3
Name: data1, dtype: int64
('b', 'one')
2    3
Name: data1, dtype: int64
('b', 'two')
3    2
Name: data1, dtype: int64
>>> # seperate key values as well
>>> for (k1, k2), group in gp2:
...     print(k1, k2)
...     print(group)
...
a one
0    2
4    4
Name: data1, dtype: int64
a two
1    3
Name: data1, dtype: int64
b one
2    3
Name: data1, dtype: int64
b two
3    2
Name: data1, dtype: int64
>>>
```

4.5.3 Data aggregation

그룹화된 데이터에 대해서도 다양한 집계 작업을 수행할 수 있습니다.

```
>>> gp1.max()
k1
a    4
b    3
Name: data1, dtype: int64
>>> gp2.min()
k1  k2
a   one    2
    two    3
b   one    3
    two    2
Name: data1, dtype: int64
```

Chapter 5. Time series

5.1 Dates and times

5.1.1 Generate series of time

'date_range' 명령을 사용하여 일련의 시간을 생성할 수 있습니다. 아래 코드에서 'periods'는 총 샘플 수입니다. 반면 freq = 'M'은 '월'을 기준으로 series를 생성해야 함을 나타냅니다.

• 기본적으로 pandas는 'M'을 월말로 간주합니다. 월의 시작에는 'MS'를 사용합니다. 마찬가지로 day ('D'), business days ('B'), hours ('H') 등에 대한 다른 옵션도 사용할 수 있습니다.

```
>>> import pandas as pd
>>> import numpy as np
>>> rng = pd.date_range('2011-03-01 10:15', periods = 10, freq = 'M')
>>> rng
DatetimeIndex(['2011-03-31 10:15:00', '2011-04-30 10:15:00',
               '2011-05-31 10:15:00', '2011-06-30 10:15:00',
               '2011-07-31 10:15:00', '2011-08-31 10:15:00',
               '2011-09-30 10:15:00', '2011-10-31 10:15:00',
               '2011-11-30 10:15:00', '2011-12-31 10:15:00'],
              dtype='datetime64[ns]', freq='M')

>>> rng = pd.date_range('2015 Jul 2 10:15', periods = 10, freq = 'M')
>>> rng
DatetimeIndex(['2015-07-31 10:15:00', '2015-08-31 10:15:00',
               '2015-09-30 10:15:00', '2015-10-31 10:15:00',
               '2015-11-30 10:15:00', '2015-12-31 10:15:00',
               '2016-01-31 10:15:00', '2016-02-29 10:15:00',
               '2016-03-31 10:15:00', '2016-04-30 10:15:00'],
              dtype='datetime64[ns]', freq='M')
```

• 마찬가지로 아래와 같이 'start' 및 'end' 매개변수를 사용하여 시계열을 생성할 수 있습니다.

```
>>> rng = pd.date_range(start = '2015 Jul 2 10:15', end = '2015 July 12', freq = '12H')
>>> rng
DatetimeIndex(['2015-07-02 10:15:00', '2015-07-02 22:15:00',
               '2015-07-03 10:15:00', '2015-07-03 22:15:00',
               '2015-07-04 10:15:00', '2015-07-04 22:15:00',
               '2015-07-05 10:15:00', '2015-07-05 22:15:00',
               '2015-07-06 10:15:00', '2015-07-06 22:15:00',
               '2015-07-07 10:15:00', '2015-07-07 22:15:00',
               '2015-07-08 10:15:00', '2015-07-08 22:15:00',
               '2015-07-09 10:15:00', '2015-07-09 22:15:00',
               '2015-07-10 10:15:00', '2015-07-10 22:15:00',
               '2015-07-11 10:15:00', '2015-07-11 22:15:00'],
              dtype='datetime64[ns]', freq='12H')
>>> len(rng)
20
```

• 시리즈 생성을 위해 시간대를 지정할 수 있습니다.

```
>>> rng = pd.date_range(start = '2015 Jul 2 10:15', end = '2015 July 12', freq = '12H', tz=
↪'Asia/Kolkata')
>>> rng
DatetimeIndex(['2015-07-02 10:15:00+05:30', '2015-07-02 22:15:00+05:30',
               '2015-07-03 10:15:00+05:30', '2015-07-03 22:15:00+05:30',
               '2015-07-04 10:15:00+05:30', '2015-07-04 22:15:00+05:30',
               '2015-07-05 10:15:00+05:30', '2015-07-05 22:15:00+05:30',
               '2015-07-06 10:15:00+05:30', '2015-07-06 22:15:00+05:30',
               '2015-07-07 10:15:00+05:30', '2015-07-07 22:15:00+05:30',
               '2015-07-08 10:15:00+05:30', '2015-07-08 22:15:00+05:30',
               '2015-07-09 10:15:00+05:30', '2015-07-09 22:15:00+05:30',
               '2015-07-10 10:15:00+05:30', '2015-07-10 22:15:00+05:30',
               '2015-07-11 10:15:00+05:30', '2015-07-11 22:15:00+05:30'],
              dtype='datetime64[ns, Asia/Kolkata]', freq='12H')
 >>>
```

• 또한 다양한 비교를 위해 데이터의 시간대를 변경할 수 있으며,

```
>>> rng.tz_convert('Australia/Sydney')DatetimeIndex(['2015-07-02 14:45:00+10:00', '2015-07-03␣
↪02:45:00+10:00',
               '2015-07-03 14:45:00+10:00', '2015-07-04 02:45:00+10:00',
               '2015-07-04 14:45:00+10:00', '2015-07-05 02:45:00+10:00',
               '2015-07-05 14:45:00+10:00', '2015-07-06 02:45:00+10:00',
               '2015-07-06 14:45:00+10:00', '2015-07-07 02:45:00+10:00',
               '2015-07-07 14:45:00+10:00', '2015-07-08 02:45:00+10:00',
               '2015-07-08 14:45:00+10:00', '2015-07-09 02:45:00+10:00',
               '2015-07-09 14:45:00+10:00', '2015-07-10 02:45:00+10:00',
               '2015-07-10 14:45:00+10:00', '2015-07-11 02:45:00+10:00',
               '2015-07-11 14:45:00+10:00', '2015-07-12 02:45:00+10:00'],
              dtype='datetime64[ns, Australia/Sydney]', freq='12H')
```

• 이러한 날짜 유형은 Timestamp 클래스입니다.

```
>>> type(rng[0])
<class 'pandas.tslib.Timestamp'>
>>>
```

5.1.2 Convert string to dates

문자열 형식의 날짜는 아래와 같이 'to_datetime' 옵션을 사용하여 타임스탬프로 변환할 수 있습니다.

```
>>> dd = ['07/07/2015', '08/12/2015', '12/04/2015']
>>> dd
['07/07/2015', '08/12/2015', '12/04/2015']
>>> type(dd[0])
<class 'str'>

>>> # American style
>>> list(pd.to_datetime(dd))
[Timestamp('2015-07-07 00:00:00'), Timestamp('2015-08-12 00:00:00'), Timestamp('2015-12-04 00:00:00')]

>>> # European format
>>> d = list(pd.to_datetime(dd, dayfirst=True))
>>> d
[Timestamp('2015-07-07 00:00:00'), Timestamp('2015-12-08 00:00:00'), Timestamp('2015-04-12 00:00:00')]
>>> type(d[0])
<class 'pandas.tslib.Timestamp'>
>>>
```

5.1.3 Periods

기간은 시간 범위를 나타냅니다. 일, 년, 분기 또는 월 등. pandas의 기간 클래스를 사용하면 빈도를 쉽게 변환할 수 있습니다.

5.1.3.1 Generating periods and frequency conversion

다음 코드에서는 주기가 'M'인 'Period' 명령을 사용하여 주기를 생성합니다. 'asfreq' 연산을 'start' 연산과 함께 사용할 때 날짜는 '01'이고 'end' 옵션이 있는 '31'입니다.

```
>>> pr = pd.Period('2012', freq='M')
>>> pr.asfreq('D', 'start')
Period('2012-01-01', 'D')
>>> pr.asfreq('D', 'end')
Period('2012-01-31', 'D')
>>>
```

5.1.3.2 Period arithmetic

기간에 대해 다양한 산술 연산을 수행할 수 있습니다. 모든 작업은 'freq'를 기반으로 수행됩니다.

```
>>> pr = pd.Period('2012', freq='A')  # Annual
>>> pr
Period('2012', 'A-DEC')
>>> pr + 1
Period('2013', 'A-DEC')

>>> # Year to month conversion
>>> prMonth = pr.asfreq('M')
>>> prMonth
Period('2012-12', 'M')
>>> prMonth - 1
Period('2012-11', 'M')
>>>
```

5.1.3.3 Creating period range

기간의 범위는 'period_range' 명령을 사용하여 생성할 수 있습니다.

```
>>> prg = pd.period_range('2010', '2015', freq='A')
>>> prg
PeriodIndex(['2010', '2011', '2012', '2013', '2014', '2015'], dtype='int64', freq='A-DEC')

>>> # create a series with index as 'prg'
>>> data = pd.Series(np.random.rand(len(prg)), index=prg)
>>> data
2010    0.785453
2011    0.606939
2012    0.558619
2013    0.321185
2014    0.224793
2015    0.561374
Freq: A-DEC, dtype: float64
>>>
```

5.1.3.4 Converting string-dates to period

문자열 날짜를 마침표로 변환하는 것은 두 단계 프로세스입니다. 즉, 먼저 문자열을 날짜 형식으로 변환한 다음 아래와 같이 날짜를 마침표로 변환해야 합니다.

```
>>> # dates as string
>>> dates = ['2013-02-02', '2012-02-02', '2013-02-02']

>>> # convert string to date format
>>> d = pd.to_datetime(dates)
>>> d
DatetimeIndex(['2013-02-02', '2012-02-02', '2013-02-02'], dtype='datetime64[ns]', freq=None)

>>> # create PeriodIndex from DatetimeIndex
>>> prd = d.to_period(freq='M')
>>> prd
PeriodIndex(['2013-02', '2012-02', '2013-02'], dtype='int64', freq='M')

>>> # change frequency type
>>> prd.asfreq('D')
PeriodIndex(['2013-02-28', '2012-02-29', '2013-02-28'], dtype='int64', freq='D')
>>> prd.asfreq('Y')
PeriodIndex(['2013', '2012', '2013'], dtype='int64', freq='A-DEC')
```

5.1.3.5 Convert periods to timestamps

기간은 'to_timestamp' 명령을 사용하여 타임스탬프로 다시 변환할 수 있습니다.

```
>>> prd
PeriodIndex(['2013-02', '2012-02', '2013-02'], dtype='int64', freq='M')
>>> prd.to_timestamp()
DatetimeIndex(['2013-02-01', '2012-02-01', '2013-02-01'], dtype='datetime64[ns]', freq=None)
>>> prd.to_timestamp(how='end')
DatetimeIndex(['2013-02-28', '2012-02-29', '2013-02-28'], dtype='datetime64[ns]', freq=None)
>>>
```

5.1.4 Time offsets

시간 간격띄우기는 다음과 같이 정의할 수 있습니다. 또한 더하기, 빼기 등 다양한 작업을 제때 수행할 수 있습니다.

```
>>> # generate time offset
>>> pd.Timedelta('3 days')
Timedelta('3 days 00:00:00')
>>> pd.Timedelta('3M')
Timedelta('0 days 00:03:00')
>>> pd.Timedelta('4 days 3M')
Timedelta('4 days 00:03:00')
>>> # adding Timedelta to time
>>> pd.Timestamp('9 July 2016 12:00') + pd.Timedelta('1 day 3 min')
Timestamp('2016-07-10 12:03:00')
>>>

>>> # add Timedelta to complete rng
>>> rng + pd.Timedelta('1 day')
DatetimeIndex(['2015-07-03 10:15:00+05:30', '2015-07-03 22:15:00+05:30',
               '2015-07-04 10:15:00+05:30', '2015-07-04 22:15:00+05:30',
               '2015-07-05 10:15:00+05:30', '2015-07-05 22:15:00+05:30',
               '2015-07-06 10:15:00+05:30', '2015-07-06 22:15:00+05:30',
               '2015-07-07 10:15:00+05:30', '2015-07-07 22:15:00+05:30',
               '2015-07-08 10:15:00+05:30', '2015-07-08 22:15:00+05:30',
               '2015-07-09 10:15:00+05:30', '2015-07-09 22:15:00+05:30',
               '2015-07-10 10:15:00+05:30', '2015-07-10 22:15:00+05:30',
               '2015-07-11 10:15:00+05:30', '2015-07-11 22:15:00+05:30',
               '2015-07-12 10:15:00+05:30', '2015-07-12 22:15:00+05:30'],
              dtype='datetime64[ns, Asia/Kolkata]', freq='12H')
>>>
```

5.1.5 Index data with time

이 섹션에서 시간은 Series 및 DataFrame에 대한 인덱스로 사용됩니다. 그런 다음 이러한 데이터 구조에 대해 다양한 작업이 수행됩니다.

• 먼저 아래와 같이 'date_range' 옵션을 사용하여 시계열을 생성합니다.

```
>>> dates = pd.date_range('2015-01-12', '2015-06-14', freq = 'M')
>>> dates
DatetimeIndex(['2015-01-31', '2015-02-28', '2015-03-31', '2015-04-30',
               '2015-05-31'],
              dtype='datetime64[ns]', freq='M')
>>> len(dates)
5
```

• 다음으로 날짜와 길이가 같은 일련의 온도를 만들고,

```
>>> atemp = pd.Series([100.2, 98, 93, 98, 100], index=dates)
>>> atemp
2015-01-31    100.2
2015-02-28     98.0
2015-03-31     93.0
2015-04-30     98.0
2015-05-31    100.0
Freq: M, dtype: float64
>>>
```

• 이제 시간 인덱스를 사용하여 아래와 같이 온도에 액세스할 수 있습니다.

```
>>> idx = atemp.index[3]
>>> idx
Timestamp('2015-04-30 00:00:00', offset='M')
>>> atemp[idx]
98.0
>>>
```

• 다음으로 또 다른 온도 계열 'stemp'를 만들고 아래와 같이 'stemp'와 'atemp'를 이용하여 DataFrame을 생성한다.

```
>>> stemp = pd.Series([89, 98, 100, 88, 89], index=dates)
>>> stemp
2015-01-31     89
2015-02-28     98
2015-03-31    100
2015-04-30     88
2015-05-31     89
Freq: M, dtype: int64
>>>

>>> # create DataFrame
>>> temps = pd.DataFrame({'Auckland':atemp, 'Delhi':stemp})
>>> temps
            Auckland  Delhi
2015-01-31     100.2     89
2015-02-28      98.0     98
2015-03-31      93.0    100
2015-04-30      98.0     88
2015-05-31     100.0     89
>>>

>>> # check the temperature of Auckland
>>> temps['Auckland']  # or temps.Auckland
2015-01-31    100.2
2015-02-28     98.0
2015-03-31     93.0
2015-04-30     98.0
2015-05-31    100.0
Freq: M, Name: Auckland, dtype: float64
>>>
```

• 이 두 도시 간의 온도 차이를 보여주는 DataFrame 'temp'에 열을 하나 더 추가할 수 있습니다.

```
>>> temps['Diff'] = temps['Auckland'] - temps['Delhi']
>>> temps
            Auckland  Delhi  Diff
2015-01-31     100.2     89  11.2
2015-02-28      98.0     98   0.0
2015-03-31      93.0    100  -7.0
2015-04-30      98.0     88  10.0
2015-05-31     100.0     89  11.0
>>>

>>> # delete the temp['Diff']
>>> del temps['Diff']
>>> temps
            Auckland  Delhi
2015-01-31     100.2     89
2015-02-28      98.0     98
2015-03-31      93.0    100
2015-04-30      98.0     88
2015-05-31     100.0     89
>>>
```

5.2 Application

이전 섹션에서 시계열의 몇 가지 기본 사항을 살펴보았습니다. 이 섹션에서는 예제를 통해 시계열의 몇 가지 사용법을 배웁니다.

5.2.1 Basics

• 먼저 아래와 같이 stock.csv 파일을 로드하고,

```
>>> import pandas as pd
>>> df = pd.read_csv('stocks.csv')
>>> df.head()
                 date    AA    GE    IBM  MSFT
0  1990-02-01 00:00:00  4.98  2.87  16.79  0.51
1  1990-02-02 00:00:00  5.04  2.87  16.89  0.51
2  1990-02-05 00:00:00  5.07  2.87  17.32  0.51
3  1990-02-06 00:00:00  5.01  2.88  17.56  0.51
4  1990-02-07 00:00:00  5.04  2.91  17.93  0.51
>>>
```

• 'date' 열의 형식을 확인하면 문자열(날짜가 아님)임을 알 수 있습니다.

```
>>> d = df.date[0]
>>> d
'1990-02-01 00:00:00'
>>> type(d)
<class 'str'>
>>>
```

• 'date'를 타임스탬프로 가져오려면 'parse_dates' 옵션을 아래와 같이 사용할 수 있습니다.

```
>>> df = pd.DataFrame.from_csv('stocks.csv', parse_dates=['date'])
>>> d = df.date[0]
>>> d
Timestamp('1990-02-01 00:00:00')
>>> type(d)
<class 'pandas.tslib.Timestamp'>
>>>

>>> df.head()
        date    AA    GE    IBM  MSFT
0 1990-02-01  4.98  2.87  16.79  0.51
1 1990-02-02  5.04  2.87  16.89  0.51
2 1990-02-05  5.07  2.87  17.32  0.51
3 1990-02-06  5.01  2.88  17.56  0.51
4 1990-02-07  5.04  2.91  17.93  0.51
```

• 날짜를 인덱스로 사용하고 싶으므로 인덱스로 로드합니다.

```
>>> df = pd.DataFrame.from_csv('stocks.csv', parse_dates=['date'], index_col='date')
>>> df.head()
            Unnamed: 0    AA    GE    IBM  MSFT
date
1990-02-01           0  4.98  2.87  16.79  0.51
1990-02-02           1  5.04  2.87  16.89  0.51
1990-02-05           2  5.07  2.87  17.32  0.51
1990-02-06           3  5.01  2.88  17.56  0.51
1990-02-07           4  5.04  2.91  17.93  0.51
```

• 'Unnamed: 0'은 유용한 컬럼이 아니므로 아래와 같이 제거할 수 있다.

```
>>> del df['Unnamed: 0']
>>> df.head()
              AA    GE    IBM  MSFT
date
1990-02-01  4.98  2.87  16.79  0.51
1990-02-02  5.04  2.87  16.89  0.51
1990-02-05  5.07  2.87  17.32  0.51
1990-02-06  5.01  2.88  17.56  0.51
1990-02-07  5.04  2.91  17.93  0.51
>>>
```

• 더 진행하기 전에 데이터를 플로팅하면서 다양한 장소에서 사용될 인덱스의 이름을 확인해 보겠습니다. 여기서 인덱스는 자동으로 플롯에 사용됩니다. 참고로 데이터는 'drop' 키워드를 사용하여 인덱스와 함께 컬럼으로 사용됩니다.

```
>>> # check the name of the index
>>> df.index.name
'date'
>>>
```

• 위의 모든 단계를 다른 방법으로 다시 실행해 보겠습니다.

```
>>> # load and display first file line of the file
>>> stocks = pd.DataFrame.from_csv('stocks.csv', parse_dates=['date'])
>>> stocks.head()
        date    AA    GE    IBM  MSFT
0 1990-02-01  4.98  2.87  16.79  0.51
1 1990-02-02  5.04  2.87  16.89  0.51
2 1990-02-05  5.07  2.87  17.32  0.51
3 1990-02-06  5.01  2.88  17.56  0.51
4 1990-02-07  5.04  2.91  17.93  0.51

>>> stocks.index.name  # nothing is set as index
>>> # set date as index but do not remove it from column
>>> stocks = stocks.set_index('date', drop=False)
>>> stocks.index.name
'date'
>>> stocks.head()
                date    AA    GE    IBM  MSFT
date
1990-02-01 1990-02-01  4.98  2.87  16.79  0.51
1990-02-02 1990-02-02  5.04  2.87  16.89  0.51
1990-02-05 1990-02-05  5.07  2.87  17.32  0.51
1990-02-06 1990-02-06  5.01  2.88  17.56  0.51
1990-02-07 1990-02-07  5.04  2.91  17.93  0.51
>>>

>>> # check the type of date
>>> type(stocks.date[0])
<class 'pandas.tslib.Timestamp'>
>>>
```

• 데이터는 아래와 같이 유효한 형식으로 날짜를 제공하여 액세스할 수 있습니다.

```
>>> # all four commands have same results
>>> # stocks.ix['1990, 02, 01']
>>> # stocks.ix['1990-02-01']
>>> # stocks.ix['1990/02/01']
>>> stocks.ix['1990-Feb-01']
date    1990-02-01 00:00:00
AA                     4.98
GE                     2.87
IBM                   16.79
MSFT                   0.51
Name: 1990-02-01 00:00:00, dtype: object
>>>
```

• 슬라이스 작업으로 일부 범위 사이의 결과를 표시(90년 2월 1일부터 90년 2월 6일까지)할 수 있습니다.
슬라이스의 마지막 날짜가 결과에 포함됩니다.

```
>>> stocks.ix['1990-Feb-01':'1990-Feb-06']
                date    AA    GE    IBM  MSFT
date
1990-02-01 1990-02-01  4.98  2.87  16.79  0.51
1990-02-02 1990-02-02  5.04  2.87  16.89  0.51
1990-02-05 1990-02-05  5.07  2.87  17.32  0.51
1990-02-06 1990-02-06  5.01  2.88  17.56  0.51
>>>

>>> # select all from Feb-1990 and display first 5
>>> stocks.ix['1990-Feb'].head()
                date    AA    GE    IBM  MSFT
date
1990-02-01 1990-02-01  4.98  2.87  16.79  0.51
1990-02-02 1990-02-02  5.04  2.87  16.89  0.51
1990-02-05 1990-02-05  5.07  2.87  17.32  0.51
1990-02-06 1990-02-06  5.01  2.88  17.56  0.51
1990-02-07 1990-02-07  5.04  2.91  17.93  0.51
>>>

>>> # use python-timedelta or pandas-offset for defining range
>>> from datetime import datetime, timedelta
>>> start = datetime(1990, 2, 1)
>>> # stocks.ix[start:start+timedelta(days=5)]  # python-timedelta
>>> stocks.ix[start:start+pd.offsets.Day(5)]  # pandas-offset
                date    AA    GE    IBM  MSFT
date
1990-02-01 1990-02-01  4.98  2.87  16.79  0.51
1990-02-02 1990-02-02  5.04  2.87  16.89  0.51
1990-02-05 1990-02-05  5.07  2.87  17.32  0.51
1990-02-06 1990-02-06  5.01  2.88  17.56  0.51
>>>
```

참고: 위의 슬라이스 작업은 날짜가 정렬된 순서인 경우에만 작동합니다. 날짜가 정렬되지 않은 경우 다음을 수행해야 합니다.
sort_index() 명령, 즉 stocks.sort_index()를 사용하여 먼저 정렬하십시오.

5.2.2 Resampling

리샘플링은 시계열을 한 빈도에서 다른 빈도로 변환하는 것입니다. 더 높은 빈도 데이터를 더 낮은 빈도로 변환하면 다운 샘플링이라고 합니다. 반면에 데이터가 낮은 빈도에서 높은 빈도로 변환되면 업샘플링이라고 합니다.

• 매월 말 데이터만 보고 싶다고 가정하고(일일 기준 아님) 다음 리샘플링 코드를 사용할 수 있습니다.

```
>>> stocks.ix[pd.date_range(stocks.index[0], stocks.index[-1], freq='M')].head()
                date    AA    GE    IBM  MSFT
1990-02-28 1990-02-28  5.22  2.89  18.06  0.54
1990-03-31        NaT   NaN   NaN    NaN   NaN   # it is not business day i.e. sat/sun
1990-04-30 1990-04-30  5.07  2.99  18.95  0.63
1990-05-31 1990-05-31  5.39  3.24  21.10  0.80
1990-06-30        NaT   NaN   NaN    NaN   NaN

>>> # 'BM' can be used for 'business month'
>>> stocks.ix[pd.date_range(stocks.index[0], stocks.index[-1], freq='BM')].head()
                date    AA    GE    IBM  MSFT
1990-02-28 1990-02-28  5.22  2.89  18.06  0.54
1990-03-30 1990-03-30  5.26  3.01  18.45  0.60
1990-04-30 1990-04-30  5.07  2.99  18.95  0.63
1990-05-31 1990-05-31  5.39  3.24  21.10  0.80
1990-06-29 1990-06-29  5.21  3.26  20.66  0.83
>>>

>>> # confirm the entry on 1990-03-30
>>> stocks.ix['1990-Mar-30']
date    1990-03-30 00:00:00
AA                     5.26
GE                     3.01
IBM                   18.45
MSFT                    0.6
Name: 1990-03-30 00:00:00, dtype: object
```

Downsampling

다음은 다운샘플링의 예입니다.

```
>>> # resample and find mean of each bin
>>> stocks.resample('BM').mean().head()
                AA        GE        IBM      MSFT
date
1990-02-28  5.043684  2.873158  17.781579  0.523158
1990-03-30  5.362273  2.963636  18.466818  0.595000
1990-04-30  5.141000  3.037500  18.767500  0.638500
1990-05-31  5.278182  3.160000  20.121818  0.731364
1990-06-29  5.399048  3.275714  20.933810  0.821429

>>> # size() : total number of rows in each bin
>>> stocks.resample('BM').size().head(3)
date
1990-02-28    19  # total 19 business days in Feb-90
1990-03-30    22
1990-04-30    20
Freq: BM, dtype: int64

>>> # count total number of rows in each bin for each column
>>> stocks.resample('BM').count().head(3)
            date  AA  GE  IBM  MSFT
date
1990-02-28    19  19  19   19    19
1990-03-30    22  22  22   22    22
1990-04-30    20  20  20   20    20
>>>

>>> # display last resample value from each bin
>>> ds = stocks.resample('BM').asfreq().head()
>>> ds
                date    AA    GE    IBM  MSFT
date
1990-02-28 1990-02-28  5.22  2.89  18.06  0.54
1990-03-30 1990-03-30  5.26  3.01  18.45  0.60
1990-04-30 1990-04-30  5.07  2.99  18.95  0.63
1990-05-31 1990-05-31  5.39  3.24  21.10  0.80
1990-06-29 1990-06-29  5.21  3.26  20.66  0.83
>>>
```

Upsampling

• 데이터를 업샘플링할 때 값은 NaN으로 채워집니다. 따라서 아래와 같이 NaN 값을 다른 값으로 대체하려면 'fillna' 방법을 사용해야 합니다.

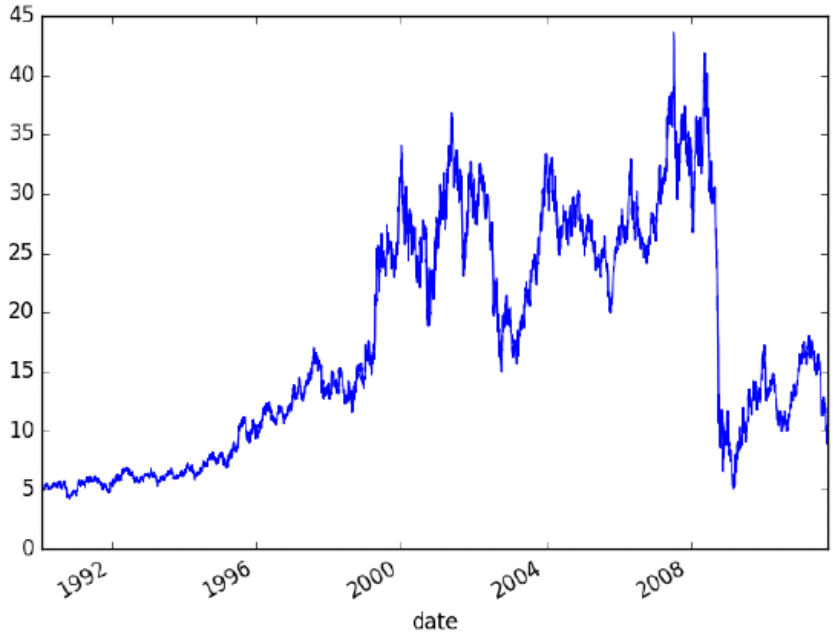
```
>>> # blank places are filled by NaN
>>> rs = ds.resample('B').asfreq()
>>> rs.head()
                date    AA    GE    IBM  MSFT
date
1990-02-28 1990-02-28  5.22  2.89  18.06  0.54
1990-03-01        NaT   NaN   NaN    NaN   NaN
1990-03-02        NaT   NaN   NaN    NaN   NaN
1990-03-05        NaT   NaN   NaN    NaN   NaN
1990-03-06        NaT   NaN   NaN    NaN   NaN


>>> # forward fill the NaN
>>> rs = ds.resample('B').asfreq().fillna(method='ffill')
>>> rs.head()
                date    AA    GE    IBM  MSFT
date
1990-02-28 1990-02-28  5.22  2.89  18.06  0.54
1990-03-01 1990-02-28  5.22  2.89  18.06  0.54
1990-03-02 1990-02-28  5.22  2.89  18.06  0.54
1990-03-05 1990-02-28  5.22  2.89  18.06  0.54
1990-03-06 1990-02-28  5.22  2.89  18.06  0.54
>>>
```

5.2.3 Plotting the data

이 섹션에서는 다양한 시간 범위에 대한 DataFrame '주식'의 다양한 데이터를 플로팅합니다.

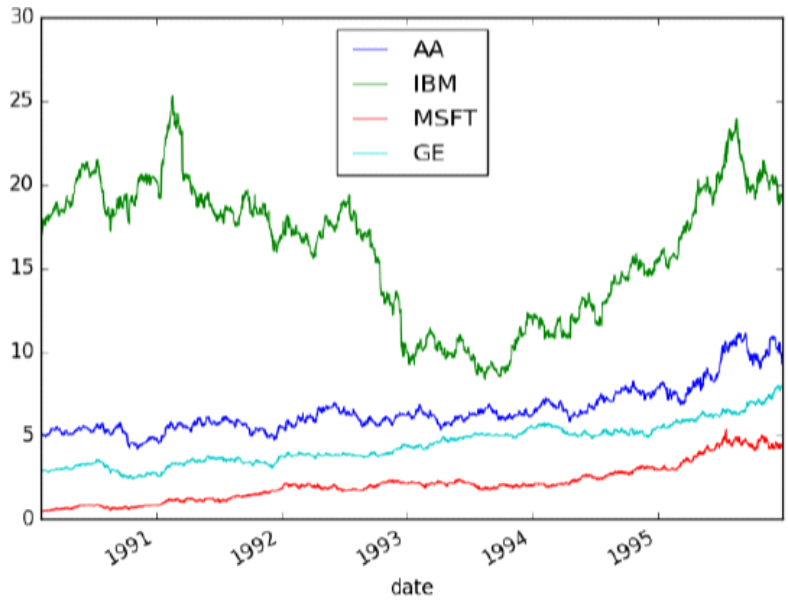
• 먼저 전체 범위에 대한 'AA'의 데이터를 플로팅합니다.

```
>>> import matplotlib.pyplot as plt
>>> stocks.AA.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xa9c3060c>
>>> plt.show()
```



• 'ix'를 사용하여 열을 선택하여 동일한 창에서 다양한 데이터를 플롯할 수 있습니다.

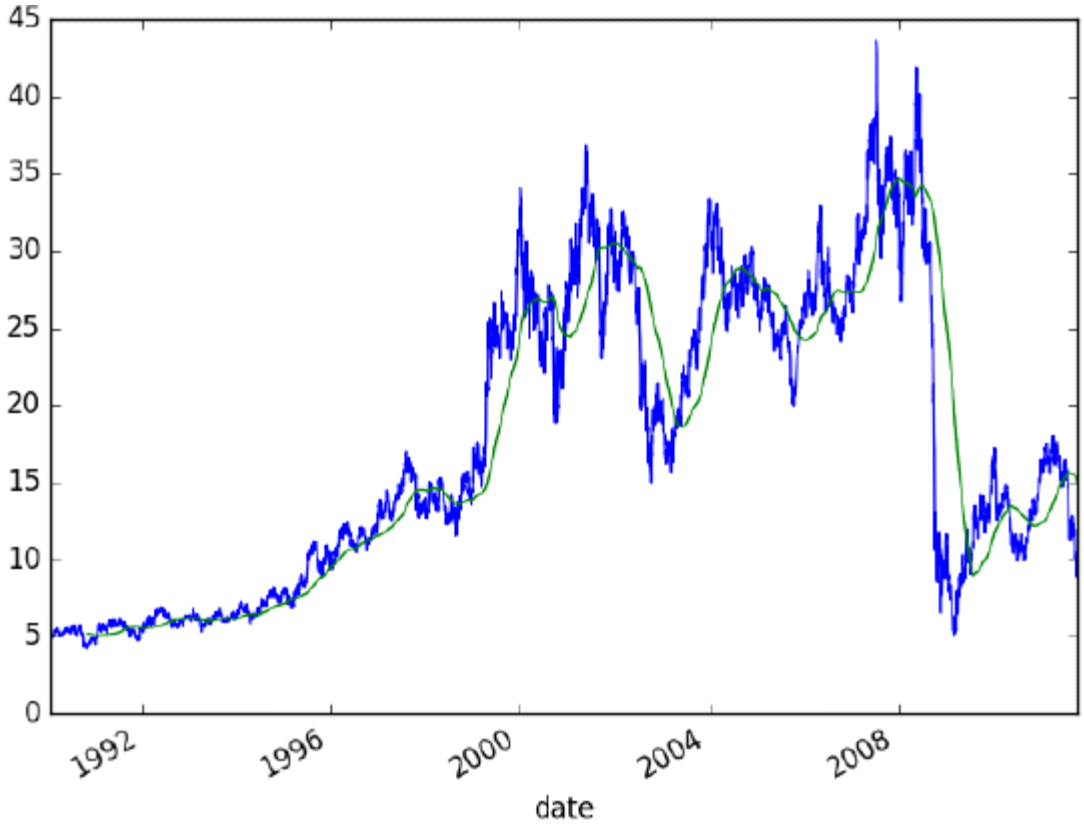
```
>>> stocks.ix['1990':'1995', ['AA', 'IBM', 'MSFT', 'GE']].plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xa9c2d2ac>
>>> plt.show()
>>>
```



5.2.4 Moving windows functions

Pandas는 슬라이딩 윈도우를 통해 데이터를 분석하는 방법을 제공합니다. 아래 코드에서 'AA'의 데이터는 길이가 250인 창에 대한 평균값과 함께 표시됩니다.

```
>>> stocks.AA.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xa9c5f4ec>
>>> stocks.AA.rolling(window=200,center=False).mean().plot()
<matplotlib.axes._subplots.AxesSubplot object at 0xa9c5f4ec>
>>> plt.show()
>>>
```



Chapter 6. Reading multiple files

이전 장에서는 데이터를 읽는 데 하나 또는 두 개의 파일만 사용했습니다. 이 장에서는 여러 파일을 연결하여 데이터를 분석합니다.

6.1 Example: Baby names trend

이 섹션에서는 다양한 텍스트 파일에서 유용한 정보를 수집하기 위해 다양한 작업을 수행합니다. 이 텍스트 파일에는 1880년 이후의 아기 이름 목록이 포함되어 있습니다. 개별 연간 파일의 각 기록은 "이름, 성별, 번호" 형식을 가집니다. 여기서 이름은 2~15자, 성별은 M(남성) 또는 F(여성)입니다. "번호"는 이름의 출현 횟수입니다. 각 파일은 먼저 성별에 따라 정렬된 다음 발생 횟수에 따라 내림차순으로 정렬됩니다. 발생 횟수가 동률인 경우 이름은 가나다 순으로 나열됩니다. 다음은 다양한 작업이 수행될 파일 목록입니다.

```
$ ls
yob1880.txt  yob1882.txt  yob1884.txt  yob1886.txt
yob1881.txt  yob1883.txt  yob1885.txt  yob1887.txt
```

다음은 yob1880.txt 파일의 처음 10줄입니다.

```
$ head -n 10 yob1980.txt
Jennifer,F,58375
Amanda,F,35817
Jessica,F,33914
Melissa,F,31625
Sarah,F,25737
Heather,F,19965
Nicole,F,19910
Amy,F,19832
Elizabeth,F,19523
Michelle,F,19113
```

6.2 Total boys and girls in year 1880

• 먼저 하나의 파일을 읽은 다음 해당 파일의 총 행 수를 확인합니다.

```
>>> import pandas as pd
>>> names1880 = pd.read_csv('yob1880.txt', names=['name', 'gender', 'birthcount'])
>>> # total number of rows in yob1880.txt
>>> len(names1880)
2000
```

```
>>>
>>> names1880.head()
        name gender  birthcount
0       Mary      F        7065
1       Anna      F        2604
2       Emma      F        2003
3  Elizabeth      F        1939
4     Minnie      F        1746
>>>
```

• 파일에는 2000개의 행이 포함되어 있습니다. 각 행에는 성별 정보와 함께 특정 이름을 가진 아기의 이름과 총 수가 포함됩니다. 성별에 따라 '출생 수'를 추가하여 총 남학생과 여학생 수를 계산할 수 있습니다. 즉, 성별에 따라 데이터를 그룹화한 다음 개별 그룹의 출생 수를 추가해야 합니다.

```
>>> # total number of boys and girls in year 1880
>>> names1880.groupby('gender').birthcount.sum()
gender
F     90993
M    110493
Name: birthcount, dtype: int64
>>>
```

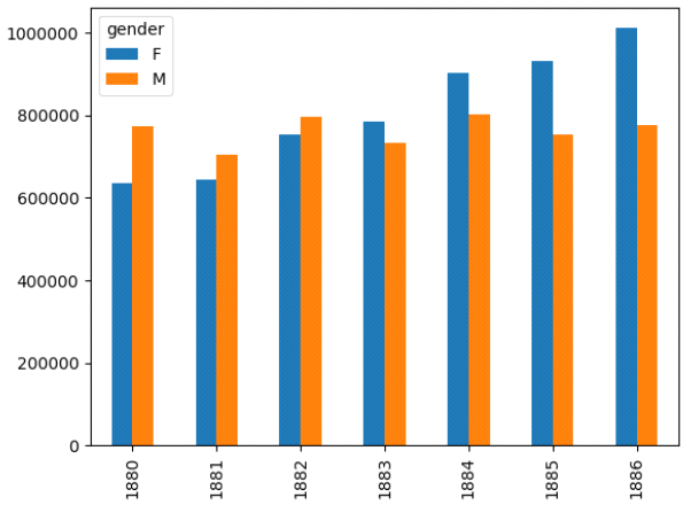
6.3 pivot_table

이전 장에서 groupby 및 unstack 작업의 다양한 예를 보았습니다. 이 두 작업은 단일 작업(예: pivot_table)으로도 수행할 수 있습니다. 이 섹션에서는 pivot_table을 사용하여 1880년에서 1887년 사이의 총 출생 수를 계산합니다. 이를 위해 먼저 올해 파일의 데이터를 병합해야 합니다.

```
>>> years = range(1880, 1887)
>>> pieces = []
>>> columns = ['name', 'gender', 'birthcount']
>>> for year in years:
...     path = 'yob{}.txt'.format(year)
...     columns = ['name', 'gender', 'birthcount']
...     for year in years:
...             path =  'yob{}.txt'.format(year)
...             df = pd.read_csv(path, names=columns)
...             df['year']=year
...             pieces.append(df)
...             allData = pd.concat(pieces, ignore_index=True)
...
>>>
>>> len(allData)
105903
>>>
>>> allData.head(2)
   name gender  birthcount  year
0  Mary      F        7065  1880
1  Anna      F        2604  1880
```

• 총 출생 수는 아래 그림과 같이 pivot_table을 사용하여 계산할 수 있습니다.

```
>>> import matplotlib.pyplot as plt
>>> total_births = allData.pivot_table('birthcount', index=['year'], columns=['gender'], aggfunc=sum)
>>> total_births.head(3)
gender       F       M
year
1880    636951  773451
1881    643685  705236
1882    754957  795809


>>> total_births.plot(kind='bar')
<matplotlib.axes._subplots.AxesSubplot object at 0xa580b44c>
>>> plt.show()
>>>
```



• 아래와 같이 'groupby' 옵션을 사용하여 동일한 작업을 수행할 수 있습니다.

```
>>> allData.groupby(['year', 'gender']).sum().unstack('gender').head(3)
       birthcount
gender          F       M
year
1880       636951  773451
1881       643685  705236
1882       754957  795809
```

• 다음으로, 전체 이름 수에 대한 이름의 비율을 확인하려고 합니다. 이를 위해 배급량을 계산하여 groupby 옵션에 적용하는 함수를 작성할 수 있습니다.

```
>>> # calculate ratio
... def add_prop(group):
...     births = group.birthcount.astype(float)
...     group['prop'] = births/births.sum()  # add column prop
...     return group
...
>>>
>>> names = allData.groupby(['year', 'gender']).apply(add_prop)

>>> names.head()
        name gender  birthcount  year      prop
0       Mary      F        7065  1880  0.011092
1       Anna      F        2604  1880  0.004088
2       Emma      F        2003  1880  0.003145
3  Elizabeth      F        1939  1880  0.003044
4     Minnie      F        1746  1880  0.00274
```