7. 데이터 정제 및 준비

데이터 분석과 모델링 작업에서는 데이터를 불러오고, 정제하고, 변형하고, 재정렬하는 데이터 준비 과정에 많은 시간을 들인다. 이런 작업들은 분석 시간의 80%를 잡아먹기도 한다. 가끔은 파일이나 DB에 저장된 데이터가 애플리케이션에서 사용하기 쉽지 않은 방식으로 저장되어 있기도 하다. 대부분의 사람은 파일이나 DB에 저장된 데이터를 다른 형태로 바꾸기 위해 파이썬이나 Perl, R, Java 혹은 awk 이나 sed 같은 유닉스의 텍스트 처리 유틸리티를 사용하기도 하는데, 파이썬 표준 라이브러리를 pandas와 함께 사용하면 큰 수고 없이 데이터를 원하는 형태로 가공할 수 있다. pandas는 이런 작업을 위해 유연하고 빠른 고수준의 알고리즘과 처리 기능을 제공한다.

혹시 이 책이나 pandas 라이브러리에서 찾을 수 없는 새로운 형태의 데이터 처리 방식을 한다면 파이썬 메일링 리스트나 pandas 깃허브에 올려놓기 바란다. 실제로 pandas는 대부분의 설계와 구현에 실제 애플리케이션 개발 과정 중에 발생한 요구 사항을 고려했다.

이 장에서는 결측치, 중복 데이터, 문자열 처리 그리고 다른 분석적 데이터 변환에 대한 도구들을 다룬다. 다음 장에서는 데이터를 합치고 재배열하는 다양한 방법을 알아보겠다.

7.1 누락된 데이터 처리하기

누락된 데이터를 처리하는 일은 데이터 분석 애플리케이션에서 흔히 발생하는 일이다. pandas의 설계 목표 중 하나는 누락 데이터를 가능한 쉽게 처리할 수 있도록 하는 것이다. 예를 들어 pandas 객체의 모든 기술 통계는 누락된 데이터를 배제하고 처리한다.

pandas 객체에서 누락된 값을 표현하는 방식은 완벽하다고 할 수 없다. 산술 데이터에 한해 pandas는 누락된 데이터를 실숫값인 NaN으로 취급한다. 이는 누락된 값을 쉽게 찾을 수 있도록하는 파수병 역활을 한다.

```
In [278]: string_data = pd.Series(['aardvark','artichoke',np.nan,'avoca
     ...: do'])

In [279]: string_data
Out[279]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [280]: string_data.isnull()
Out[280]:
0    False
1    False
2     True
3    False
dtype: bool
```

pandas에서는 R 프로그래밍 언어에서 결측치를 NA로 취급하는 개념을 차용했다. 분석 애플리케이션에서 NA데이터는 데이터가 존재하지 않거나, 존재하더라도 데이터를 수집하는 과정 등에서 검출되지 않았음을 의미한다. 분석을 위해 데이터를 정제하는 과정에서 결측치 자체를 데이터 수집 과정에서의 실수나 결측치로 인한 잠재적인 편향을 찾아내는 수단으로 인식하는 것은 중요하다.

파이썬의 내장 None 값 또한 NA값으로 취급된다.

```
In [281]: string_data[0] = None

In [282]: string_data.isnull()
Out[282]:
0     True
1    False
2     True
3    False
dtype: bool
```

pandas 프로젝트에서는 결측치를 처리하는 방법을 개선하는 작업이 진행 중이지만 pandas, isnull 같은 사용자 API 함수에서는 성가신 부분을 추상화로 제거했다. 다음 표에서 결측치 처리와 관련된 함수를 정리해봤다.

NA 처리 메서드

인자	설명
dropna	누락된 데이터가 있는 축(로우, 컬럼)을 제외시킨다. 어느 정도의 누락 데이터까지 용인할 것인지 지정할 수 있다.
fillna	누락된 데이터를 대신할 값을 채우거나 'ffill' 이나 'bfill' 같은 보간 메서드를 적용한다,
isnull	누락되거나 NA인 값을 알려주는 불리언값이 저장된 같은 형의 객체를 반환한다.
notnull	isnull과 반대되는 메서드

7.1.1 누락된 데이터 골라내기

누락된 데이터를 골라내는 몇 가지 방법이 있는데, pandas.isnull 이나 불리언 색인을 사용해 직접 손으로 제거하는 것도 한 가지 방법이지만, dropna를 쓰면 매우 요용하게 처리할 수 있다. Series에 dropna 메서드를 적용하면 Null이 아닌 데이터와 색인값만 들어 있는 Series를 반환한다.

```
In [283]: from numpy import nan as NA

In [284]: data = pd.Series([1,NA,3.5,NA,7])

In [285]: data.dropna()
Out[285]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

위 코드는 다음과 동일하다.

```
In [286]: data[data.notnull()]
Out[286]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

DataFrame 객체의 경우에는 조금 복잡한데, 모두 NA값인 로우나 컬럼을 제외시키거나 NA값을 하나라도 포함하고 있는 로우나 컬럼을 제외시킬 수 있다. dropna는 기본적으로 NA값을 하나라도 포함하고 있는 로우를 제외시킨다.

```
In [287]: data = pd.DataFrame([[1.,6.5,3.],[1,NA,NA],[NA,NA,NA]
     ...:          ,[NA,6.5,3]])

In [288]: data
Out[288]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

In [289]: cleaned = data.dropna()

In [290]: cleaned
Out[290]:
     0    1    2
0  1.0  6.5  3.0
```

how = 'all' 옵션을 넘기면 모두 NA 값인 로우만 제외시킨다.

```
In [291]: data.dropna(how='all')
Out[291]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

컬럼을 제외시키는 방법도 동일하게 동작한다. 옵션으로 axis = 1을 넘겨주면 된다.

```
In [292]: data.dropna(how='all', axis = 1)
Out[292]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

컬럼을 제외시키는 방법도 동일하게 동작한다. 옵션으로 axis = 1을 넘겨주면 된다.

```
In [293]: data[4] = NA

In [294]: data
Out[294]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN

In [295]: data.dropna(axis =1,how='all')
Out[295]:
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

DataFrame의 로우를 제외시키는 방법은 사계열 데이터에 주로 사용되는 경향이 있다. 몇 개 이상의 값이 들어 있는 로우만 살펴보고 싶다면 thresh 인자에 원하는 값을 넘기면 된다.

```
In [296]: df = pd.DataFrame(np.random.randn(7,3))

In [297]: df.iloc[:4,1] = NA

In [298]: df.loc[:1,2] = NA

In [299]: df
Out[299]:
          0         1         2
0 -0.073971       NaN       NaN
1 -1.960882       NaN       NaN
2 -2.855304       NaN -1.184459
3 -1.654010       NaN  0.247734
4 -0.177429 -1.398389 -0.438246
5 -1.672870 -0.268012  0.207077
6 -1.604443 -0.317003  1.060488

In [300]: df.dropna()
Out[300]:
          0         1         2
4 -0.177429 -1.398389 -0.438246
5 -1.672870 -0.268012  0.207077
6 -1.604443 -0.317003  1.060488

In [301]: df.dropna(thresh=2)
Out[301]:
          0         1         2
2 -2.855304       NaN -1.184459
3 -1.654010       NaN  0.247734
4 -0.177429 -1.398389 -0.438246
5 -1.672870 -0.268012  0.207077
6 -1.604443 -0.317003  1.060488
```

7.1.2 결측치 채우기

누락된 값을 예외시키지 않고 (잠재적으로 다른 데이터라도 함께 버려질 가능성이 있다) 데이터상의 '구멍'을 어떻게든 메우고 싶은 경우가 있다. 이 경우 fillna 메서드를 활용하면 되는데, fillna 메서드에 채워 넣고 싶은 값을 넘겨주면 된다.

```
In [302]: df.fillna(0)
Out[302]:
          0         1         2
0 -0.073971  0.000000  0.000000
1 -1.960882  0.000000  0.000000
2 -2.855304  0.000000 -1.184459
3 -1.654010  0.000000  0.247734
4 -0.177429 -1.398389 -0.438246
5 -1.672870 -0.268012  0.207077
6 -1.604443 -0.317003  1.060488
```

fillna에 사전값을 넘겨서 각 컬럼마다 다른 값을 채울 수도 있다.

```
In [303]: df.fillna({1:0.5,2:0})
Out[303]:
          0         1         2
0 -0.073971  0.500000  0.000000
1 -1.960882  0.500000  0.000000
2 -2.855304  0.500000 -1.184459
3 -1.654010  0.500000  0.247734
4 -0.177429 -1.398389 -0.438246
5 -1.672870 -0.268012  0.207077
6 -1.604443 -0.317003  1.060488
```

fillna는 새로운 객체를 반환하지만 다음처럼 기존 객체를 변경할 수 도 있다.

```
In [304]: _=df.fillna(0,inplace=True)

In [305]: df
Out[305]:
          0         1         2
0 -0.073971  0.000000  0.000000
1 -1.960882  0.000000  0.000000
2 -2.855304  0.000000 -1.184459
3 -1.654010  0.000000  0.247734
4 -0.177429 -1.398389 -0.438246
5 -1.672870 -0.268012  0.207077
6 -1.604443 -0.317003  1.060488
```

재색인에서 사용 가능한 보간 메서드는 fillna 매서드에서도 사용가능하다.

```
In [306]: df = pd.DataFrame(np.random.randn(6,3))

In [307]: df.iloc[2:,1]=NA

In [308]: df.iloc[4:,2]=NA

In [309]: df
Out[309]:
          0         1         2
0 -0.110138 -0.674279  0.249441
1 -0.286022 -0.272327 -0.582323
2 -0.779379       NaN  0.403444
3  0.147179       NaN -1.070702
4  2.029751       NaN       NaN
```

```
5  0.841747       NaN       NaN

In [310]: df.fillna(method = 'ffill')
Out[310]:
          0         1         2
0 -0.110138 -0.674279  0.249441
1 -0.286022 -0.272327 -0.582323
2 -0.779379 -0.272327  0.403444
3  0.147179 -0.272327 -1.070702
4  2.029751 -0.272327 -1.070702
5  0.841747 -0.272327 -1.070702
```

조금만 창의적으로 생각하면 fillna를 이용해서 매우 다양한 일을 할 수 있는데 예를 들어 Series의 평균값이나 중간값을 전달할 수도 있다.

```
In [311]: data = pd.Series([1.,NA,3.5,NA,7])

In [312]: data.fillna(data.mean())
Out[312]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64

In [313]: data.fillna(data.median())
Out[313]:
0    1.0
1    3.5
2    3.5
3    3.5
4    7.0
dtype: float64
```

fillna에 대한 설명은 다음과 같다.

인자	설명
value	비어 있는 값을 채울 스칼라값이나 사전 형식의 객체
method	보간 방식, 기본적으로 'ffill'을 사용한다.
axis	값을 채워 넣을 축, 기본값은 axis = 0이다.
inplace	복사본을 생성하지 않고 호출한 객체를 변경한다. 기본값은 false이다.
limit	값을 앞 혹은 뒤에서부터 몇 개까지 채울지 지정한다.

7.2 데이터 변형

지금까지 데이터를 재배치하는 방법을 알아봤다. 필터링, 정제 및 다른 변형 역시 중요한 연산이다.

7.2.1 중복 제거하기

여러 가지 이유로 DataFrame에서 중복된 로우를 발견할 수 있다. 예제를 보자 .

```
In [314]: data = pd.DataFrame({'k1':['one','two']*3+['two'],
     ...:                      'k2':[1,1,2,3,3,4,4]})

In [315]: data
Out[315]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

DataFrame의 duplicated 메서드는 각 로우가 중복인지 아닌지 알려주는 불리언 Series를 반환한다.

```
In [316]: data.duplicated()
Out[316]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

drop._duplicates는 duplicated 배열이 False인 DataFrame을 반환한다.

```
In [317]: data.drop_duplicates()
Out[317]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

이 두 메서드는 기본적으로 모든 컬럼에 적용되며 중복을 찾아내기 위한 부분합을 따로 지정해 줄 수도 있다. 새로운 칼럼을 하나 추가하고 'k1' 컬럼에 기반해서 중복을 걸러내려면 다음과 같이 하면 된다.

```
In [318]: data['v1']=range(7)

In [319]: data.drop_duplicates(['k1'])
Out[319]:
    k1  k2  v1
0  one   1   0
1  two   1   1
```

duplicatad와 drop_duplicates는 기본적으로 처음 발견된 값을 유지한다. keep='last' 옵션을 넘기면 마지막으로 발견된 값을 반환한다.

```
In [320]: data.drop_duplicates(['k1','k2'],keep='last')
Out[320]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6

In [321]: data.drop_duplicates(['k1','k2'])
Out[321]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
5  two   4   5
```

7.2.2 함수나 매핑을 이용해서 데이터 변형하기

데이터를 다루다 보면 DataFrame의 칼럼이나 Series, 배열 내의 값을 기반으로 데이터의 형태를 변환하고 싶은 경우가 있다. 가상으로 수집한 육류에 대한 다음 정보를 한번 살펴보자.

```
In [322]: data = pd.DataFrame({'food':['bacon','pulled pork','bacon',
     ...:                               'Pastrami','corned beef','Bacon'
     ...:                               ,'pastrami','honey ham','nova lo
     ...:                              x'],
     ...:                     'ounces':[4,3,12,6,7.5,8,3,5,6] })

In [323]: data
Out[323]:
            food  ounces
0          bacon     4.0
1    pulled pork     3.0
2          bacon    12.0
3       Pastrami     6.0
4    corned beef     7.5
5          Bacon     8.0
6       pastrami     3.0
7     honey ham      5.0
8       nova lox     6.0
```

그리고 해당 육류가 어떤 동물의 고기인지 알려줄 수 있는 칼럼을 하나 추가한다고 가정하고 육류별 동물을 담고 있는 사전 데이터를 아래처럼 작성하자.

```
In [324]: meat_to_animal = {
     ...:                    'bacon':'pig',
     ...:                    'pulled':'pig',
     ...:                    'pastrami':'cow',
     ...:                    'corned beaf':'cow',
     ...:                    'honey jam':'pig',
     ...:                    'nova lox':'salmon'}
```

Series의 map 메서드는 사전류의 객체나 어떤 함수를 받을 수 있는데, 위 데이터에는 육류 이름에 대소문자가 섞여 있는 사소한 문제가 있으므로 str.lower 메서드를 사용해서 모두 소문자로 변경한다.

```
In [345]: lowercased = data['food'].str.lower()

In [346]: lowercased
Out[346]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object

In [347]: data['animal'] = lowercased.map(meat_to_animal)

In [348]: data
Out[348]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

물론 함수를 넘겨서 같은 일을 수행할 수도 있다.

```
In [349]: data['food'].map(lambda x:meat_to_animal[x.lower()])
#   조작하고 싶은 데이터 대상.map(람다 함수 x:구분해줄 기준 데이터 집합[x에 대한 데이터 소문자로 변환])

Out[349]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

map 메서드를 사용하면 데이터의 요소별 변환 및 데이터를 다듬는 작업을 편리하게 수행할 수 있다.

7.2.3 값 치환하기

fillna 메서드를 사용해서 누락된 값을 채우는 일은 일반적인 값 치환 작업이라고 볼 수 있다. 위에서 살펴봤듯이 map 메서드를 한 객체 안에서 값의 부분집합을 변경하는 데 사용했다면 replace 메서드는 같은 작업에 대해 좀 더 간단하고 유연한 방법을 제공한다. 다음 Series 객체를 살펴보자.

```
In [350]: data = pd.Series([1.,-999.,2.,-999.,-1000.,3.])

In [351]: data
Out[351]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

-999는 누락된 데이터를 나타내기 위한 값이다. replace 메서드를 이용하면 이 값을 pandas에서 인식할 수 있는 NA 값으로 치환한 새로운 Series를 생성할 수 있다.(인자로 inplace = Ttue를 넘기지 않았다면)

```
In [352]: data.replace(-999,np.nan)
Out[352]:
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

여러 개의 값을 한 번에 치환하려면 하나의 값 대신 치환하려는 값의 리스트를 넘기면 된다.

```
In [353]: data.replace([-999,-1000],np.nan)
Out[353]:
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

치환하려는 값마다 다른 값으로 치환하려면 누락된 값 대신 새로 지정할 값의 리스트를 사용하면 된다.

```
In [354]: data.replace([-999,-1000],[np.nan,0])
Out[354]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

두 개의 리스트 대신 사전을 이용하는 것도 가능하다.

```
In [355]: data.replace({-999:np.nan,-1000:0})
Out[355]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

NOTE_ data.replace 메서드는 문자열 치환을 항목 단위로 수행하는 data.str.replace와 구별되는데 자세한 내용은 나중에 Series의 문자열 메서드와 함께 알아보겠다.

7.2.4 축 색인 이름 바꾸기

Series의 값들처럼 축 이름 역시 유사한 방식으로 함수나 새롭게 바꿀 값을 이용해서 변환할 수 있다. 새로운 자료구조를 만들지 않고 그 자리에서 바로 축 이름을 변경하는 것이 가능하다.

다음 예제를 살펴보자,

```
In [357]: data = pd.DataFrame(np.arange(12).reshape((3,4)),
     ...:                     index = ['Ohio','Colorado','New York'],
     ...:                     columns = ['one','two','three','four'])
```

Series와 마찬가지로 축 색인에도 map 메서드가 있다.

```
In [363]: transform = lambda x: x[:4].upper()

In [364]: data.index.map(transform)
Out[364]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')

In [365]: data.columns.map(transform)
Out[365]: Index(['ONE', 'TWO', 'THRE', 'FOUR'], dtype='object')
```

대문자로 변경된 축 이름을 DataFrame의 index에 바로 대입할 수 있다.

```
In [366]: data.index = data.index.map(transform)

In [367]: data
Out[367]:
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

원래 객체를 변경하지 않고 새로운 객체를 생성하려면 rename 메서드를 사용한다.

```
In [368]: data.rename(index=str.title,columns=str.upper)
Out[368]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11

In [369]: data.rename(index=str.upper,columns=str.upper)
Out[369]:
      ONE  TWO  THREE  FOUR
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

특히 rename 메서드는 사전 형식의 객체를 이용해서 축 이름 중 일부만 변경하는 것도 가능하다.

```
In [370]: data.rename(index={'OHIO':'INDIANA'},
     ...:             columns={'three':'peekaboo'})
Out[370]:
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

rename 메서드를 사용하면 DataFrame을 직접 복사해서 index와 columns 속성을 갱신할 필요 없이 바로 변경할 수 있다. 원본 데이터를 바로 변경하려면 inplace = True 옵션을 넘겨주면 된다.

```
In [371]: data.rename(index={'OHIO':'INDIANA'},
     ...:             inplace = True)

In [372]: data
Out[372]:
         one  two  three  four
INDIANA    0    1      2     3
COLO       4    5      6     7
NEW        8    9     10    11
```

7.2.5 개별화와 양자화

연속성 데이터는 종종 개별로 분할하거나 아니면 분석을 위해 그룹별로 나누기도 하는데, 수업에 참여하는 학생 그룹 데이터가 있고, 나이대에 따라 분류한다고 가정하자.

```
In [373]: ages = [20,22,25,27,21,23,37,31,61,45,41,32]
```

이 데이터를 pandas의 cut 함수를 이용해서 18~25, 26~35, 35~60, 60이상 그룹으로 나눠보자.

```
In [374]: bins = [18,25,35,60,100]

In [375]: cats = pd.cut(ages,bins)

In [376]: cats
Out[376]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35],
 (60, 100], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

pandas에서 반환하는 객체는 Categorical 이라는 특수한 객체이다. 결과에서 보이는 그룹은 pandas.cut으로 계산된 것이다. 이 객체는 그룹 이름이 담긴 배열이라고 생각하면 된다. 이 Categorical 객체는 codes 속성에 있는 ages 데이터에 대한 카테고리 이름을 categories라는 배열에 내부적으로 담고 있다.

```
In [377]: cats.codes
Out[377]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [378]: cats.categories
Out[378]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')

In [379]: pd.value_counts(cats)
Out[379]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

pd.value_counts(cat)는 pandas.cut 결과에 대한 그룹 수다.

간격을 나타내는 표기법은 중괄호로 시작해서 대괄호로 끝나는데 **중괄호 쪽의 값은 포함하지 않고 대괄호 쪽의 값은 포함**하는 간격을 나타낸다. right = False를 넘겨서 중괄호 대신 대괄호 쪽이 포함되지 않도록 바꿀 수 있다.

```
In [380]: pd.cut(ages,[18,26,36,61,100],right=False)
Out[380]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ...,
[26, 36), [61, 100), [36, 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

labels 옵션으로 그룹의 이름을 직접 넘겨줄 수도 있다.

```
In [381]: group_names = ['Youth','YoungAdult','MiddleAged','Senier']

In [382]: pd.cut(ages, bins, labels=group_names)
Out[382]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult,
Senier, MiddleAged, MiddleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senier]
```

만약 cut 함수에 명시적으로 그룹의 경곗값을 넘기지 않고 그룹의 개수를 넘겨주면 데이터에서 최솟값과 최댓값을 기준으로 균등한 길의의 그룹을 자동으로 계산한다. 어떤 균등분포 내에서 4개의 그룹으로 나누는 경우를 생각해보자.

```
In [384]: data = np.random.rand(20)

In [385]: pd.cut(data,4,precision=2)
Out[385]:
[(0.042, 0.28], (0.28, 0.52], (0.52, 0.75], (0.28, 0.52], (0.75, 0.99],
..., (0.042, 0.28], (0.75, 0.99], (0.52, 0.75], (0.52, 0.75], (0.75, 0.99]]
Length: 20
Categories (4, interval[float64]): [(0.042, 0.28] < (0.28, 0.52]
< (0.52, 0.75] < (0.75, 0.99]]
```

precision = 2 옵션은 소수점 아래 2자리까지로 제한한다.

이를 위한 가장 적절한 함수로 qcut이 있는데, 표본 변위치를 기반으로 데이터를 나눠준다. cut 함수를 사용하면 데이터의 분산에 따라 각각의 그룹마다 데이터 수가 다르게 나뉘는 경우가 많다.

qcut 은 표준 변위치를 사용하기 때문에 적당히 칼은 크기의 그룹으로 나눌 수 있다.

```
In [386]: data = np.random.randn(1000) #정규분포

In [387]: cats = pd.qcut(data,4) #4분위로 분류

In [388]: cats
Out[388]:
[(-0.672, 0.0357], (0.0357, 0.754], (-3.116, -0.672], (-3.116, -0.672],
(-0.672, 0.0357], ..., (-3.116, -0.672], (-0.672, 0.0357], (-0.672, 0.0357],
(0.754, 3.385], (0.0357, 0.754]]
Length: 1000
Categories (4, interval[float64]): [(-3.116, -0.672] < (-0.672, 0.0357] <
                                    (0.0357, 0.754] <
                                    (0.754, 3.385]]

In [389]: pd.value_counts(cats)
Out[389]:
(0.754, 3.385]      250
(0.0357, 0.754]     250
(-0.672, 0.0357]    250
(-3.116, -0.672]    250
dtype: int64
```

cut 함수와 유사하게 변위치를 직접 지정해줄 수 있다(변위치는 0부터 1까지).

```
In [390]: pd.qcut(data, [0,0.1,0.5,0.9,1.])
Out[390]:
[(-1.345, 0.0357], (0.0357, 1.289], (-1.345, 0.0357],
(-3.116, -1.345], (-1.345, 0.0357], ..., (-1.345, 0.0357],
(-1.345, 0.0357], (-1.345, 0.0357], (0.0357, 1.289], (0.0357, 1.289]]
Length: 1000
Categories (4, interval[float64]): [(-3.116, -1.345] < (-1.345, 0.0357]
                                    < (0.0357, 1.289] <  (1.289, 3.385]]
```

그룹 분석과 변위치를 다룰 때는 cut과 qcut 함수 같은 이산함수가 특히 더 유용한데 이 내용은 수집과 그룹 연산을 다루는 10장에서 다시 한 번 살펴보도록 하겠다.

7.2.6 특잇값을 찾고 제외하기

배열 연산을 수행할 때는 특잇값을 제외하거나 적당한 값으로 대체하는 것이 중요하다. 적절히 분산된 값이 담겨 있는 다음 DataFrame을 살펴보자.

```
In [391]: data = pd.DataFrame(np.random.randn(100,4))

In [392]: data.describe()
Out[392]:
                0           1           2           3
count  100.000000  100.000000  100.000000  100.000000
mean    -0.037060   -0.130995    0.118413    0.118915
std      0.921145    0.984549    0.940277    0.995542
min     -2.722456   -3.208492   -2.868804   -3.781210
25%     -0.590850   -0.735375   -0.438463   -0.430893
50%     -0.097533   -0.142177    0.035975    0.022899
75%      0.502266    0.538154    0.690383    0.730492
max      2.118501    2.575787    2.476841    2.888625
```

이 DataFrame의 한 컬럼에서 절댓값이 2.5를 초과하는 값을 찾아내자.

```
In [401]: col = data[1]

In [402]: col[np.abs(col)>3]
Out[402]:
77   -3.208492
Name: 1, dtype: float64
```

절댓값이 3을 초과하는 값이 들어 있는 모든 로우를 선택하려면 불리언 DataFrame에서 any 메서드를 사용하면 된다.

```
In [403]: data[(np.abs(data)>3).any(1)]
Out[403]:
           0         1         2        3
20  0.169532  1.062933 -0.589852 -3.78121
77  0.545588 -3.208492 -0.317684  0.72231
```

이 기준대로 쉽게 값을 선택할 수 있으며, 아래 코드로 -3 이나 3을 초과하는 값을 -3 또는 3으로 지정할 수 있다.

```
In [406]: data[np.abs(data)>3]=np.sign(data)*3

In [407]: data.describe()
Out[407]:
                0           1           2           3
count  100.000000  100.000000  100.000000  100.000000
mean    -0.037060   -0.128910    0.118413    0.126727
std      0.921145    0.978167    0.940277    0.967293
min     -2.722456   -3.000000   -2.868804   -3.000000
25%     -0.590850   -0.735375   -0.438463   -0.430893
50%     -0.097533   -0.142177    0.035975    0.022899
75%      0.502266    0.538154    0.690383    0.730492
max      2.118501    2.575787    2.476841    2.888625
```

np.sign(data)는 data값이 양수인지 음수인지에 따라 1이나 -1이 담긴 배열을 반환한다.

```
In [408]: np.sign(data).head()
Out[408]:
     0    1    2    3
0 -1.0  1.0  1.0 -1.0
1  1.0 -1.0 -1.0 -1.0
2  1.0 -1.0 -1.0 -1.0
3 -1.0 -1.0  1.0  1.0
4 -1.0 -1.0 -1.0  1.0
```

7.2.7 치환과 임의 샘플링

numpy.random.permutation 함수를 이용하면 Series 나 DataFrame의 로우를 쉽게 임의 순서로 재배치할 수 있다. 순서를 바꾸고 싶은 만큼의 길이를 permutation 함수로 넘기면 바뀐 순서가 담긴 정수 배열이 생성된다.

```
In [410]: df = pd.DataFrame(np.arange(5*4).reshape((5,4)))

In [411]: sampler = np.random.permutation(5)

In [412]: sampler
Out[412]: array([2, 4, 1, 0, 3])
```

이 배열은 iloc 기반의 색인이나 take 함수에서 사용가능하다.

```
In [413]: df
Out[413]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [414]: df.take(sampler)
Out[414]:
    0   1   2   3
2   8   9  10  11
4  16  17  18  19
1   4   5   6   7
0   0   1   2   3
3  12  13  14  15
```

치환 없이 일부만 임의로 선택하려면 Series나 DataFrame의 simple 메서드를 사용하면 된다.

```
In [415]: df.sample(n=3)
Out[415]:
    0   1   2   3
3  12  13  14  15
0   0   1   2   3
1   4   5   6   7
```

(반복 선택을 허용하며) 표본 치환을 통해 생성해내려면 sample에 replace = True 옵션을 넘긴다.

```
In [416]: choices = pd.Series([5,7,-1,6,4])
```

```
In [417]: draws = choices.sample(n=10,replace=True)

In [418]: draws
Out[418]:
4    4
4    4
3    6
2   -1
2   -1
0    5
1    7
2   -1
4    4
4    4
dtype: int64
```

7.2.8 표시자/더미 변수 계산하기

통계 모델이나 머신러닝 어플리케이션을 위한 또 다른 데이터 변환은 분류값을 '더미' 나 '표시자' 행렬로 전환하는 것이다. 만약 어떤 DataFrame의 한 컬럼에 K가지의 값이 있다면 K개의 컬럼이 있는 DataFrame이나 행렬을 만들고 값으로는 1과 0을 채워 넣을 것이다. pandas의 get_dummies가 이를 위한 함수 인데 독자 스스로 새로운 방법을 고안해내는 것도 어렵지 않을 것이다. 앞서 살펴본 DataFrame을 다시 살펴보자.

```
In [419]: df = pd.DataFrame({'key':['b','b','a','c','a','b'],
     ...:                    'data1':range(6)})

In [420]: pd.get_dummies(df['key'])
Out[420]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

표시자 DataFrame 안에 있는 컬럼에 접두어를 추가한 후 다른 데이터와 병합하고 싶을 경우가 있다. get_dummies 함수의 prefix 인자를 사용하면 이를 수행할 수 있다.

```
In [4]: dummies = pd.get_dummies(df['key'],prefix = 'key')

In [5]: dummies
Out[5]:
   key_a  key_b  key_c
0      0      1      0
1      0      1      0
2      1      0      0
3      0      0      1
4      1      0      0
5      0      1      0

In [6]: df_with_dummy = df[['data1']].join(dummies)

In [7]: df_with_dummy
```

```
Out[7]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

DataFrame의 한 로우가 여러 카테고리에 속한다면 조금 복잡해지는데, 자세한 내용은 차후에 다룬다.

```
In [14]: movies = pd.read_table('C:/Users/김대현/Desktop/data/파라데/py
    ...: data-book-2nd-edition/datasets/movielens/movies.dat',sep='::',
    ...: header = None, names = mnames)
C:\ProgramData\Anaconda3\Scripts\ipython:1: ParserWarning: Fa
lling back to the 'python' engine because the 'c' engine does not support
regex separators (separators > 1 char and different
from '\s+' are interpreted as regex); you can avoid this warning
by specifying engine='python'.

In [15]: mnames = ['movie_id','title', 'genres']

In [16]: movies[:10]
Out[16]:
   movie_id  ...                         genres
0         1  ...   Animation|Children's|Comedy
1         2  ...  Adventure|Children's|Fantasy
2         3  ...                Comedy|Romance
3         4  ...                  Comedy|Drama
4         5  ...                        Comedy
5         6  ...         Action|Crime|Thriller
6         7  ...                Comedy|Romance
7         8  ...          Adventure|Children's
8         9  ...                        Action
9        10  ...     Action|Adventure|Thriller

[10 rows x 3 columns]
```

각 장르마다 표시자 값을 추가하려면 약가느이 수고를 해야 하는데 먼저 데이터 묶음에서 유일한 장르 목록을 추출해야 한다.

```
In [17]: all_genres = []

In [18]: for x in movies.genres:
    ...:     all_genres.extend(x.split('|'))
    ...:

In [19]: genres = pd.unique(all_genres)
```

이제 장르는 아래와 같은 모양이 된다.

```
In [20]: genres
Out[20]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

이제 표시자 DataFrame을 생성하기 위해 0으로 초기화된 DataFrame을 생성하자.

```
In [22]: zero_matrix = np.zeros((len(movies),len(genres)))

In [23]: dummies = pd.DataFrame(zero_matrix,columns=genres)

    In [24]: zero_matrix
Out[24]:
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])

In [25]: dummies
Out[25]:
      Animation  Children's  Comedy  ...  Mystery  Film-Noir  Western
0           0.0         0.0     0.0  ...      0.0        0.0      0.0
1           0.0         0.0     0.0  ...      0.0        0.0      0.0
2           0.0         0.0     0.0  ...      0.0        0.0      0.0
3           0.0         0.0     0.0  ...      0.0        0.0      0.0
4           0.0         0.0     0.0  ...      0.0        0.0      0.0
...         ...         ...     ...  ...      ...        ...      ...
3878        0.0         0.0     0.0  ...      0.0        0.0      0.0
3879        0.0         0.0     0.0  ...      0.0        0.0      0.0
3880        0.0         0.0     0.0  ...      0.0        0.0      0.0
3881        0.0         0.0     0.0  ...      0.0        0.0      0.0
3882        0.0         0.0     0.0  ...      0.0        0.0      0.0

[3883 rows x 18 columns]
```

각 영화를 순회하면서 dummies의 각 로우 항목을 1로 설정한다. 각 장르의 컬럼 색인을 계산하기 위해 dummies.columns를 사용하자.

```
In [26]: gen = movies.genres[0]

In [27]: gen
Out[27]: "Animation|Children's|Comedy"

In [28]: gen.split('|')
Out[28]: ['Animation', "Children's", 'Comedy']

In [30]: dummies.columns.get_indexer(gen.split('|'))
Out[30]: array([0, 1, 2], dtype=int64)
```

그리고 iloc를 이용해서 색인에 맞게 값을 대입하자.

```
In [52]: for i, gen in enumerate(movies, genres):
    ...:     indices = dummies.columns.get_indexer(gen.split('|'))
    ...:     dummies.iloc[i, indices] = 1
    ...:
```

그리고 앞에서 한 것처럼 movies와 조합하면 된다.

```
In [54]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [59]: movies_windic.iloc[0]
Out[59]:
movie_id                                        1
title                             Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                                 1
Genre_Children's                                1
Genre_Comedy                                    1
Genre_Adventure                                 0
Genre_Fantasy                                   0
Genre_Romance                                   0
Genre_Drama                                     0
Genre_Action                                    0
Genre_Crime                                     0
Genre_Thriller                                  0
Genre_Horror                                    0
Genre_Sci-Fi                                    0
Genre_Documentary                               0
Genre_War                                       0
Genre_Musical                                   0
Genre_Mystery                                   0
Genre_Film-Noir                                 0
Genre_Western                                   0
Name: 0, dtype: object
```

Notes_ 이보다 더 큰 데이터라면 이 방법으로 다중 멤버쉽을 갖는 표시자 변수를 생성하는 것은 그다지 빠른 방법이 아니다. 속도를 높이려면 직접 NumPy 배열에 접근하는 저수준의 함수를 작성해서 DataFrame에 결과를 저장하도록 해야한다.

get.dummies와 cut 같은 이산함수를 잘 조합하면 통계 어플에서 유용하게 사용할 수 있다.

```
In [60]: np.random.seed(12345)

In [63]: values = np.random.rand(10)

In [64]: values
Out[64]:
array([0.92961609, 0.31637555, 0.18391881, 0.20456028, 0.56772503,
       0.5955447 , 0.96451452, 0.6531771 , 0.74890664, 0.65356987])

In [65]: bins = [0,0.2,0.4,0.6,0.8,1]

In [66]: pd.get_dummies(pd.cut(values,bins))
Out[66]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           0           0           1
1           0           1           0           0           0
```

```
2          1          0          0          0          0
3          0          1          0          0          0
4          0          0          1          0          0
5          0          0          1          0          0
6          0          0          0          0          1
7          0          0          0          1          0
8          0          0          0          1          0
9          0          0          0          1          0
```

여기에서는 예제 값이 불편하도록 numpy.random.see 함수를 이용해서 난수 시드값을 지정했다.

pandas.get_dummies 함수는 나중에 다시 살펴볼 것이다.

7.3 문자열 다루기

파이썬은 문자열이나 텍스트 처리의 용이함 덕분에 원시 데이터를 처리하는 인기 있는 언어가 되었다. 대부분의 텍스트 연산은 문자열 객체의 내장 메서드로 간단하게 처리할 수 있다. 좀 더 복잡한 패턴 매칭이나 텍스트 조작은 정규 표현식을 필요로 한다. **pandas는 배열 데이터 전체에 쉽게 정규 표현식을 적용하고 , 누락된 데이터를 편리하게 처리할 수 있는 기능을 포함하고 있다.**

7.3.1 문자열 객체 메서드

문자열을 다뤄야 하는 대부분의 애플리케이션은 내장 문자열 메서드만으로도 충분하다. 예를 들어 쉼표로 구분된 문자열은 split 메서드를 이용해서 분리할 수 있다.

```
In [71]: val = 'a,b,        guido'

In [72]: val.split(',')
Out[72]: ['a', 'b', '        guido']
```

split 메서드는 종종 공백 문자 (줄바꿈 문자 포함)를 제거하는 strip 메서드와 조합해서 사용하기도 한다.

```
In [72]: val.split(',')
Out[72]: ['a', 'b', '        guido']

In [73]: pieces = [x.strip() for x in val.split(',')]

In [74]: pieces
Out[74]: ['a', 'b', 'guido']
```

이렇게 분리된 문자열은 더하기 연산을 사용해서 ::문자열과 합칠 수도 있다.

```
In [75]: first,    second, third = pieces

In [76]: first +'::'+second+'::'+third
Out[76]: 'a::b::guido'
```

하지만 이 방법은 실용적이면서 범용적인 메서드는 아니다. 빠르고 좀 더 파이썬스러운 방법은 리스트나 튜플을 ::문자열의 join 메서드로 전달하는 것이다.

```
In [77]: '::'.join(pieces)
Out[77]: 'a::b::guido'
```

일치하는 부분문자열의 위치를 찾는 방법도 있다. index나 find를 사용하는 것도 가능하지만 파이썬의 in 예약어를 사용하면 일치하는 부분문자열을 쉽게 찾을 수 있다.

```
In [78]: 'guido' in val
Out[78]: True

In [79]: val.index(',')
Out[79]: 1

In [80]: val.find(':')
Out[80]: -1
```

find와 index의 차이점은 index의 경우 문자열을 찾지 못하면 예외를 발생시킨다는 것이다.

find의 경우에는 -1을 반환한다.

```
In [81]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-81-2c016e7367ac> in <module>
----> 1 val.index(':')

ValueError: substring not found
```

count는 특정 부분문자열이 몇 건 발견됐는지 반환한다.

```
In [83]: val.count(',')
Out[83]: 2
```

replace는 찾아낸 패턴을 다른 문자열로 치환한다. 이 메서드는 대체할 문자열로 비어 있는 문자열을 넘겨서 패턴을 삭제하기 위한 방법으로 자주 사용되기도 한다.

```
In [84]: val.replace(',','::')
Out[84]: 'a::b::        guido'
    In [85]: val.replace(',','')
Out[85]: 'ab        guido'
```

아래 표에 파이썬의 문자열 함수를 모아뒀다.

인자	설명
count	문자열에서 겹치지 않은 부분문자열의 개수를 반환한다.
endswith	문자열이 주어진 접미사로 끝날 경우 True를 반환한다.
startwith	문자열이 주어진 접두사로 시작할 경우 True를 반환한다.
join	문자열을 구분자로 하여 다른 문자열을 순서대로 이어붙인다.
index	부분문자열의 첫 번째 글자의 위치를 반환한다. 부분 문자열이 없을 경우 ValueError 예외가 발생한다.
find	**첫 번째** 부분문자열의 첫 번째 글자의 위치를 반환한다. index와 유사하지만 부분문자열이 없을 경우 -1을 반환한다.
rfind	**마지막** 부분문자열의 첫 번째 글자의 위치를 반환한다. 부분문자열이 없을 경우 -1을 반환한다.
replace	문자열을 다른 문자열로 치환한다.
strip,rstrip,lstrip	개행 문자를 포함한 공백 문자를 제거한다. lstrip은 문자열의 시작 부분에 있는 공백 문자만 제거하며, rstrip은 문자열의 마지막 부분에 있는 공백 문자만 제거한다.
split	문자열을 구분자 기준으로, 부분문자열의 리스트로 분리한다.
lower	알파벳 문자를 소문자로 변환한다.
upper	알파벳 문자를 대문자로 변환한다.
casefold	문자를 소문자로 변환한다. 지역 문자들은 그에 상응하는 대체 문자로 교체된다.
ljust, rjust	문자열을 오른쪽 또는 왼쪽으로 정렬하고 주어진 길이에서 문자열의 길이를 제외한 나머지 부분은 공백 문자를 채워 넣는다.

정규 표현식을 이러한 다양한 용도로 사용할 수 있다.

7.3.2 정규 표현식

정규 표현식은 텍스트에서 문자열 패턴을 찾는 유연한 방법을 제공한다. 흔히 regex라 불리는 단일 표현식은 정규 표현 언어로 구성된 문자열이다. 파이썬에는 re 모듈이 내장되어 있어서 문자열에 대한 정규 표현식을 처리한다. 몇 가지 예제로 알아보자.

Note_ 정규 표현식을 작성하는 방법은 그 자체로 하나의 독립된 장으로 구성할 수 있고 따라서 이 책에서 다루는 범위를 벗어난다. 정규 표현식에 관한 많은 튜토리얼과 레퍼런스가 있다.

re모듈함수는 **패턴 매칭, 치환, 분리 세 가지**로 나눌 수 있다. 물론 이 세 가지는 모두 서로 연관되어 있는데, 정규 표현식은 텍스트 내에 존재하는 패턴을 표현하고 이를 여러 가지 다양한 목적으로 사용할 수 있도록 돼 있다. 간단한 예제를 하나 살펴보자. 여러 가지 공백 문자(탭, 스페이스, 개행문자)가 포함된 문자열을 나누고 싶다면 하나 이상의 공백 문자를 의미하는 \s+ 를 사용해서 문자열을 분리한다.

```
In [86]: import re

In [87]: text = "foo bar\t baz \tqux"

In [88]: re.split('\s+',text)
Out[88]: ['foo', 'bar', 'baz', 'qux']
```

re.split('\s+',text)를 사용하면 먼저 정규 표현식이 **컴파일**되고 그 다음에 split 메서드가 실행된다.

그 다음에 split 메서드가 실행된다. re.compile로 직접 정규 표현식을 컴파일하고 그렇게 얻은 정규 표현식 객체를 재사용하는 것도 가능하다.

```
In [89]: regex = re.compile('\s+')

In [90]: regex.split(text)
Out[90]: ['foo', 'bar', 'baz', 'qux']
```

정규 표현식에 매칭되는 모든 패턴의 목록을 얻고 싶다면 findall 메서드를 사용한다.

```
In [91]: regex.findall(text)
Out[91]: [' ', '\t ', ' \t']
```

Note_ 정규 표현식에서 \문자가 이스케이프되는 문제를 피하려면, raw 문자열 표기법을 사용한다. 그러면 \를 이스케이프 문자로 처리하지 않고 일반 문자로 처리하기 때문에 \를 간단하게 표현할 수 있다. 즉, 'C:\\x ' 대신 r'C:\x' 를 사용한다.

같은 정규 표현식을 다른 문자열에도 적용해야 한다면 re.compile을 이용해서 정규 표현식 객체를 만들어 쓰는 방법을 추천한다. 이렇게 하면 CPU 사용량을 아낄 수 있다.

match와 search는 findall 메서드와 관련이 있다.

findall은 문자열에서 일치하는 모든 부분문자열을 찾아주지만,

search 메서드는 패턴과 일치하는 첫 번째 존재를 반환한다.

match 메서드는 이보다 더 엄격해서 문자열의 시작부분에서 일치하는 것만 찾아준다.

약간 복잡한 예제로 이메일 주소를 검사하는 정규 표현식을 한 번 살펴보자.

```
In [92]: text = """Dave dave@google.com
    ...: Steve steve@gmail.com
    ...: Rob rob@gmail.com
    ...: Ryan ryan@yahoo.com
    ...: """
    ...: pattern=r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

#re.IGNORECASE는 정규 표현식이 대소문자를 가리지 않도록 한다.
In [94]: regex=re.compile(pattern,flags=re.IGNORECASE)

In [95]: regex
Out[95]: re.compile(r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}',
                    re.IGNORECASE|re.UNICODE)
```

findall 메서드를 사용해서 이메일 주소의 리스트를 생성하자.

```
In [4]: regex.findall(text)
Out[4]: ['dave@google.com', 'steve@gmail.com',
         'rob@gmail.com', 'ryan@yahoo.com']
```

search는 텍스트에서 첫 번째 이메일 주소만을 찾아준다. 위 정규 표현식에 대한 match 객체는 그 정규 표현 패턴이 문자열 내에서 위치하는 시작점과 끝점만을 알려준다.

```
In [8]: m = regex.search(text)

In [10]: m
Out[10]: <re.Match object; span=(5, 20), match='dave@google.com'>

In [11]: text[m.start():m.end()]
Out[11]: 'dave@google.com'

In [12]: m.start()
Out[12]: 5

In [13]: m.end()
Out[13]: 20
```

regex.match는 None을 반환한다. 그 정규 표현 패턴이 문자열의 시작점에서부터 일치하는지 검사하기 때문이다.

```
In [20]: print(regex.match(text))
None
```

sub 메서드는 찾은 패턴을 주어진 문자열로 치환하여 새로운 문자열을 반환한다.

```
In [21]: print(regex.sub('REDACTED',text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

이메일 주소를 찾아서 동시에 각 이메일 주소를 **사용자 이름, 도메인 이름, 접미사** 세 가지 컴포넌트로 나눠야 한다면 각 패턴을 괄호로 묶어준다.

```
In [39]: pattern = r'([A-ZO-9._%+-]+)@([A-ZO-9.-]+)\.([A-Z]{2,4})'

In [40]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

이렇게 만든 match 객체를 이용하면 groups 메서드로 각 패턴 컴포넌트의 튜플을 얻을 수 있다.

```
In [46]: m = regex.match('wesm@bright.net')

In [47]: m.groups()
Out[47]: ('wesm', 'bright', 'net')
```

패턴에 그룹이 존재한다면 findall 메서드는 튜플의 목록을 반환한다.

```
In [48]: regex.findall(text)
Out[48]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

sub 역시 마찬가지로 \1,\2 같은 특수한 기호를 사용해서 각 패턴 그룹에 접근할 수 있다. \1은 첫 번째로 찾은 그룹을 의미하고, \2는 두 번째로 찾은 그룹을 의미한다.

```
In [51]: print(regex.sub(r'Username:\1,Domain:\2,Suffix:\3',text))
Dave Username:dave,Domain:google,Suffix:com
Steve Username:steve,Domain:gmail,Suffix:com
Rob Username:rob,Domain:gmail,Suffix:com
Ryan Username:ryan,Domain:yahoo,Suffix:com
```

이 밖에도 파이썬에서 할 수 있는 정규 표현식은 많이 있지만 대부분 이 책의 범위를 벗어난다.

다음은 정규 표현식의 메서드를 정리한 것이다.

인자	설명
findall	문자열에서 겹치지 않는 모든 발견된 패턴을 리스트로 반환한다.
finditer	findall과 같지만 발견된 패턴을 이터레이터를 통해 하나씩 반환한다.
match	문자열의 시작점부터 패턴을 찾고 선택적으로 패턴 컴포넌트를 그룹으로 나눈다. 일치하는 패턴이 있다면 match 객체를 반환하고 그렇지 않으면 None을 반환한다.
search	문자열에서 패턴과 일치하는 내용을 검색하고 match 객체를 반환한다. match 매서드와는 다르게 시작부터 일치하는 내용만 찾지 않고 문자열 어디든 일치하는 내용이 있다면 반환한다.
split	문자열에서 패턴과 일치하는 부분을 분리한다.
sub, subn	문자열에서 일치하는 모든 패턴(sub) 혹은 패턴(subn)을 대체 표현으로 치환한다. 대체 표현 문자열은 \1 , \2 , ... 와 같은 기호를 사용해서 매치 그룹의 요소를 참조한다.

7.3.3 pandas의 벡터화된 문자열 함수

뒤죽박죽인 데이터를 분석을 위해 정리하는 일은 문자열을 다듬고 정규화하는 작업을 필요로 한다.

문자열을 담고 있는 컬럼에 누락된 값이 있다면 일을 더 복잡하게 만든다.

```
In [59]: data = {'Dave' : 'dave@google.com',
    ...:         'Steve': 'steve@gmail.com',
    ...:         'Rob':'rob@gmail.com',
    ...:         'Wes':np.nan}

In [60]: data = pd.Series(data)

In [61]: data
Out[61]:
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
```

```
Wes                  NaN
dtype: object

In [62]: data.isnull()
Out[62]:
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

문자열과 정규 표현식 메서드는 data.ma을 사용해서 각 값에 적용(lambda 혹은 다른 함수를 넘겨서) 할 수 있지만 NA값을 만나면 실패한다. 이런 문제에 대처하기 위해 Series에는 NA 값을 건너뛰도록 하는 간결한 문자열 처리 메서드가 있다. 이는 Series의 str 속성을 이용하는데, 예를 들어 각 이메일 주소가 'gmail'을 포함하고 있는지 str.contains를 이용해서 검사할 수 있다.

```
In [64]: data.str.contains('gmail')
Out[64]:
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

정규 표현식을 IGNORECASE 같은 re 옵션과 함께 사용하는 것도 가능하다.

```
In [68]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [69]: pattern
Out[69]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [70]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[70]:
Dave     [(dave, google, com)]
Steve    [(steve, gmail, com)]
Rob        [(rob, gmail, com)]
Wes                        NaN
dtype: object
```

벡터화된 요소를 꺼내오는 몇 가지 방법이 있는데 str.get을 이용하거나 str 속성의 색인을 이용한다.

```
In [71]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [72]: matches
Out[72]:
Dave     True
Steve    True
Rob      True
Wes       NaN
dtype: object
```

내재된 리스트의 원소에 접근하기 위해서는 색인을 넘기면 된다.

```
In [106]: data.str[:5]
Out[106]:
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

벡터화된 문자열 메서드

```
In [109]: data.str.cat()
Out[109]: 'dave@google.comsteve@gmail.comrob@gmail.com'

#cat : 선택적인 구분자와 함께 요소별로 문자열을 이어 붙인다.
#contains : 문자열이 패턴이나 정규 표현식을 포함하는지 나타내는 불리언 배열을 반환한다.

In [113]: data.count()
Out[113]: 3
#count : 일치하는 패선 수를 반환한다.

In [120]: data.str.extract(r'([dave])')
Out[120]:
         0
Dave     d
Steve    e
Rob      a
Wes    NaN
#extract : 문자열이 담긴 Series에서 하나 이상의 문자열을 추출하기 위해
#           정규 표현식을 이용한다. 결과는 각 그룹이 하나의 컬럼이 되는 DataFrame이다.

In [131]: data.str.findall(pattern,flags = re.IGNORECASE)
Out[131]:
Dave     [(dave, google, com)]
Steve    [(steve, gmail, com)]
Rob        [(rob, gmail, com)]
Wes                        NaN
dtype: object
#findall :  각 문자열에 대해 일치하는 패턴/정규 표현식의 전체 목록을 구한다.

In [132]: data.str.get(1)
Out[132]:
Dave       a
Steve      t
Rob        o
Wes      NaN
dtype: object
#get : i번째 요소를 반환한다.
```

7.4 마치며

효율적인 데이터 준비 과정은 분석 준비를 하는 데 드는 시간을 줄이고 실제 분석에 좀 더 많은 시간을 쓸 수 있도록 하여 결과적으로는 생산성을 향상시킨다. 이 장에서 많은 도구를 살펴봤지만 여기서 살펴본 것들이 전부는 아니다. 다음 장에서는 pandas의 데이터 병합과 그룹 기능을 살펴보겠다.