6. 데이터 로딩과 저장, 파일 형식

이 책에서 다루는 대부분의 도구를 사용하는 첫 관문은 데이터에 접근하는 것이다. 다양한 형식의 데이터를 읽고 쓸 수 있는 많은 라이브러리가 있지만 이 책에서는 pandas에 초점을 맞춰 설명한다.

일반적으로 입출력은 몇 가지 작은 범주로 나뉘는데, 텍스트 파일을 이용하는 방법, 데이터베이스를 이용하는 방법, 웹 API를 이용해서 네트워크를 통해 불러오는 방법이 있다.

6.1 텍스트 파일에서 데이터를 읽고 쓰는 법

pandas에는 표 형식의 자료를 DataFrame 객체로 읽어오는 몇 가지 기능을 제공하고 있다. 아마 read_csv와 read_table을 주로 사용하게 될 테지만 다은 함수도 정리해뒀다.

pandas 파일 파싱함수

함수	설명
read_csv	파일, URL 또는 파일과 유사한 객체로부터 구분된 데이터를 읽어온다.
read.table	파일, URL 또는 파일과 유사한 객체로부터 구분된 데이터를 읽어온다. 데이터 구분자는 탭('\t')을 기본으로 한다.
read_fwf	고정폭 컬럼 형식에서 데이터를 읽어온다(구분자가 없는 데이터)
read_clipboard	클립도으에 있는 데이터를 읽어오는 read_table 함수. 웹페이지에서 표를 읽어올 때 유용하다.
read_excel	엑셀 파일(XLS, XLSX)에서 표 형식의 데이터를 읽어온다.
read_hdf	pandas에서 저장한 HDFS 파일에서 데이터를 읽어온다.
read_html	HTML 문서 내의 모든 테이블의 데이터를 읽어온다.
read_json	JSON 문자열에서 데이터를 읽어온다.
read_msgpack	메시지팩 바이너리 포멧으로 인코딩된 pandas 데이터를 읽어온다.
read_pickle	파이썬 피클 포맷으로 저장된 객체를 읽어온다.
read_sas	SAS 시스템의 사용자 정의 저장 포맷으로 저장된 데이터를 읽어온다.
read_sql	SQL 쿼리 결과를 pandas의 DataFrame 형식으로 읽어온다.
read_stata	Stata 파일에서 데이터를 읽어온다.
read_feather	Feather 바이너리 파일 모맷으로부터 데이터를 읽어온다.

위 함수들은 텍스트 데이터를 DataFrame으로 읽어오기 위한 함수인데, 아래와 같은 몇 가지 옵션을 취한다.

- **색인**

반환하는 DataFrame에서 하나 이상의 컬럼을 색인으로 지정할 수 있다. 파일이나 사용자로부터 컬럼 이름을 받거나 아무것도 받지 않을 수 있다.

- **자료형 추론과 데이터 변환**

사용자 정의 값 변환과 비어 있는 값을 위한 사용자 리스트를 포함한다.

- **날짜 분석**

여러 컬럼에 걸쳐 있는 날짜와 시간 정보를 하나의 컬럼에 조합해서 결과에 반영한다.

- **반복**

여러 개의 파일에 걸쳐 있는 자료를 반복적으로 읽어올 수 있다.

- **정제되지 않은 데이터 처리**

로우나 꼬리말, 주석 건너뛰기 또는 천 단위 마다 쉼표로 구분된 숫자 같은 사소한 것들의 처리를 해준다.

실제 데이터는 엉망진창인 상태이므로 데이터를 불러오는 함수(특히 read.csv 같은)는 개발이 계속됨에 따라 복잡도가 급속도로 증가한다. 넘쳐나는 함수 인자들(read.csv의 함수 인자는 현재 50개가 넘는다)을 보고 있으면 두통이 온다. pandas 온라인 문서에는 각 인자들이 어떻게 동작하는지 다양한 예제와 함께 설명하고 있으므로 특정 파일을 읽는 데 어려움을 느낀다면 필요한 인자와 그에 맞는 예제가 도움 될 것이다.

이런 함수들 중 일부, 예를 들어 pandas.read_csv 같은 함수들은 데이터 형식에 자료형이 포함되어 있지 않은 관계로 **타입 추론**을 수행한다. HDF5나 Feather나 msgpack의 경우에는 데이터 형식에 자료형이 포함되어 있다.

날짜나 다른 몇 가지 사용자 정의 자료형을 처리하려면 수고가 조금 필요하다. 쉼표로 구분된 작은 CSV 파일을 한번 살펴보자.

```
In [53]: !type ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Note_ 여기서는 유닉스의 cat 명령어를 사용해서 파일의 내용을 확인했다. 윈도우 사용자라면 cat 대신 type명령어를 사용해서 내용을 확인할 수 있다.

이 파일은 쉼표로 구분되어 있기에 read_csv 를 사용해서 DataFrame으로 읽어올 수 잇다.

```
In [54]: df = pd.read_csv('ex1.csv')

In [55]: df
Out[55]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

read_table에 구분자를 쉼포로 지정해서 읽어올 수도 있다.

```
In [56]: pd.read_table('ex1.csv', sep=',')
Out[56]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

모든 파일에 컬럼 이름이 있는 건 아니다. 다음 파일을 보자.

```
In [57]: !type ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

이 파일을 읽어노는 몇 가지 옵션이 있는데, pandas가 자동으로 컬럼 이름을 생성하도록 하거나 우리가 직접 컬럼 이름을 지정한다.

```
In [58]: pd.read_csv('ex2.csv',header = None)
Out[58]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [59]: pd.read_csv('ex2.csv',names = ['a','b','c','d','message'])
Out[59]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

message 컬럼을 색인으로 하는 DataFrame을 반환하려면 index_col 인자에 4번째 'message'이름을 가진 컬럼을 지정해서 색인으로 만들 수 있다.

```
In [60]: names = ['a', 'b', 'c', 'd', 'message']

In [62]: pd.read_csv('ex2.csv', names=names, index_col='message')
Out[62]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

계층적 색인을 지정하고 싶다면 컬럼 번호나 이름의 리스트를 넘기면 된다.

```
In [64]: !type 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

In [65]: parsed = pd.read_csv('csv_mindex.csv',index_col = ['key1', 'key2'])

In [66]: parsed
Out[66]:
           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
```

가끔 고정된 구분자 없이 공백이나 다른 패턴으로 필드를 구분해놓은 경우가 있다. 다음과 같은 파일이 있다고 하자.

```
In [67]: list(open('ex3.txt'))
Out[67]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

직접 파일을 고쳐도 되지만, 이 파일은 필드가 여러 개의 공백 문자로 구분되어 있으므로 이를 표현할 수 있는 정규 표현식 \s+를 사용해서 처리할 수도 있다.

```
In [68]: result = pd.read_table('ex3.txt',sep='\s+')

In [69]: result
Out[69]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

이 경우 첫 번째 로우는 다른 로우보다 컬럼이 하나 적기 때문에 read_table은 첫 번째 컬럼이 DataFrame의 색인이 되어야 한다고 추론한다.

파서 함수는 파일 형식에서 발생할 수 있는 매우 다양한 예외를 잘 처리할 수 있도록 많은 추가 인자를 가지고 있는데, 예를 들면 skiprows를 이용해서 첫 번째, 세 번째, 네 번째 로우를 건너뛸 수 있다.

```
In [70]: !type 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,foo

In [71]: pd.read_csv('ex4.csv',skiprows = [0,2,3])
Out[71]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

누락된 값을 잘 처리하는 것은 파일을 읽는 과정에서 자주 발생하는 일이고 중요한 문제 ㅔ다. 보통 텍스트 파일에서 누락된 값은 표기되지 않거나 (비어 있는 문자열) 구분하기 쉬운 특수한 문자로 표기된다. 기본적으로 pandas는 NA나 NULL처럼 흔히 통용되는 문자를 비어 있는 값으로 사용한다.

```
In [72]: !type ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [73]: result = pd.read_csv('ex5.csv')

In [74]: result
Out[74]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [75]: pd.isnull(result)
Out[75]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

na_values 옵션은 리스트나 문자열 집합을 받아서 누락된 값을 처리한다.

```
In [76]: result = pd.read_csv('ex5.csv',na_values = ['NULL'])

In [77]: result
Out[77]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

컬럼마다 다른 NA문자를 사전값으로 넘겨서 처리할 수도 있다.

```
In [78]: sentinels = {'message':['foo','NA'], 'something':['two']}

In [79]: pd.read_csv('ex5.csv',na_values=sentinels)
Out[79]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

다음 표에 pandas_read.csv와 pandas_read_table에서 자주 사용하는 인자들을 모아두었다.

read_csv와 read_table 함수 인자

인자	설명
path	파일 시슽메에서의 위치, URL, 파일 객체를 나타내는 문자열
sep 또는 delimeter	필드를 구분하기 위해 사용할 연속된 문자나 정규 표현식
header	컬럼 이름으로 사용할 로우 번호, 기본값은 0(첫 번째 로우)이며, 헤더가 없을 경우에는 None으로 지정할 수 있다..
index_col	색인으로 사용할 컬럼 번호나 이름. 계층적 색인을 지정할 경우 리스트를 넘길 수 있다.
names	컬럼 이름으로 사용할 리스트. header = None과 함께 사용한다.
skiprows	파일의 시작부터 무시할 행 수 또는 무시할 로우 번호가 담긴 리스트
na_values	NA값으로 처리할 값들의 목록
comment	주석으로 분류되어 파싱하지 않을 문자 혹은 문자열
parse_dates	날짜를 datetime으로 변환할지 여부. 기본값은 False이며, True일 경우 모든 컬럼에 적용한다. 컬럼의 번호나 이름을 포함한 리스트를 넘겨서 변환할 컬럼을 지정할 수 있는데, [1,2,3]을 넘기면 각각의 컬럼을 datetime으로 변환하며, [[1,3]]을 넘기면 1,3번 컬럼을 조합해서 하나의 datetime으로 변환한다.
keep_date_col	여러 컬럼을 datetime으로 변환했을 경우 원래 컬럼으로 남겨둘지 여부, 기본값은 True
converters	변환 시 컬럼에 적용할 함수를 지정한다. 예를 들어 {'foo' : f}는 'foo' 컬럼에 f 함수를 적용시킨다. 전달하는 사전의 키값은 컬럼 이름이나 번호가 될 수 있다.
dayfirst	모호한 날짜 형식일 경우 국제 형식으로 간주한다. (7/6/2012는 2012년 6월 7일로 간주한다). 기본값은 False
date_parser	날짜 변환 시 사용할 함수
nrows	파일의 첫 일부만 읽어올 때 처음 몇 줄을 읽을 것인지 지정
iteratior	파일을 조금씩 읽을 때 사용하도록 TextParser 객체를 반환하도록 한다. 기본값은 False
chunksize	TextParser 객체에서 사용할 한 번에 읽을 파일의 크기
skip_footer	파일의 끝에서 무시할 라인 수
verbose	파싱 결과에 대한 정보를 출력한다. 숫자가 아닌 값이 들어 있는 칼럼에 누락된 값이 있다면 줄 번호를 출력해준다. 기본값은 False
encoding	유니코드 인코딩 종류를 지정한다. UTF-8로 인코딩된 텍스트일 경우 'utf-8'로 지정한다.
squeeze	만일 컬럼이 하나뿐이라면 Series 객체를 반환한다. 기본값은 False
thousands	숫자를 천 단위로 끊을 때 사용할 ',' 나 '.' 같은 구분자

6.1.1 텍스트 파일 조금씩 읽어오기

매우 큰 파일을 처리할 때 인자를 제대로 주었는지 알아보기 위해 파일의 일부분만 읽어보거나 여러 파일 중에서 몇 개의 파일만 읽어서 확인해보고 싶을 것이다.

큰 파일을 다루기 전에 pandas의 출력 설정을 조금 손보자.

```
In [80]: pd.options.display.max_rows = 10
```

이제 최대 10개의 데이터만 출력한다.

```
In [81]: result = pd.read_csv('ex6.csv')

In [82]: result
Out[82]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0

[10000 rows x 5 columns]
```

파일 전체를 읽는 대신 처음 몇 줄만 읽어보고 싶다면 nrow 옵션을 주면 된다.

```
In [83]: pd.read_csv('ex6.csv',nrows = 5)
Out[83]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

파일을 여러 조각으로 나누어서 읽고 싶다면 chunksize 옵션으로 로우의 개수를 주면 된다.

```
In [84]: chunker = pd.read_csv('ex6.csv',chunksize=1000)

In [85]: chunker
Out[85]: <pandas.io.parsers.TextFileReader at 0x1fed3d337c8>
```

read_csv에서 반환된 TextParser 객체를 이용해서 chunksize에 따라 분리된 파일들을 순회 할 수 있다. 예를 들어 ex 6.csv 파일을 순회하면서 'key' 로우에 있는 값을 세어보려면 다음처럼 하면 된다.

```
In [87]: tot=pd.Series([])
    ...: for piece in chunker:
    ...:     tot=tot.add(piece['key'].value_counts(),fill_value=0)
    ...:
```

```
C:\ProgramData\Anaconda3\Scripts\ipython:1: DeprecationWarning: The default
dtype for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.

In [88]: tot=tot.sort_values(ascending=False)

In [89]: tot[:10]
Out[89]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

TextParser는 임의 크기의 조각을 읽을 수 있는 get_chunk 메서드도 포함하고 있다.

6.1.2 데이터를 텍스트 형식으로 기록하기

읽어오기와 마찬가지로 데이터를 구분자로 구분한 형식으로 내보내는 것도 가능하다. 위에서 읽었던 CSV 파일 중 하나를 다시 보자.

```
In [90]: data = pd.read_csv('ex5.csv')

In [91]: data
Out[91]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

DataFrame의 to_csv 메서드를 이용하면 데이터를 쉼표로 구분된 형식으로 파일에 쓸 수 있다.

```
In [92]: data.to_csv('out.csv')

In [93]: !type out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

물론 다른 구분자도 사용 가능하다(콘솔에서 확인할 수 있도록 실제 파일로 기록하지 않고 sys.stdout에 결과를 기록하도록 했다).

```
In [94]: import sys

In [95]: data.to_csv(sys.stdout,sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

결과에서 누락된 값은 비어 있는 문자열로 나타나는데, 이것 역시 원하는 값으로 지정 가능하다.

```
In [96]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

다른 옵션을 명시하지 않으면 로우와 컬럼 이름이 기록된다. 로우와 컬럼 이름을 포함하지 않으려면 다음과 같이 한다.

```
In [97]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

칼럼의 일부분만 기록할 수도 있으며, 순서를 직접 지정할 수도 있다.

```
In [98]: data.to_csv(sys.stdout,index=False,columns=['a','b','c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series 에도 to_csv 메서드가 존재한다.

```
In [99]: dates = pd.date_range('1/1/2000', periods=7)

In [100]: ts = pd.Series(np.arange(7),index=dates)

In [101]: ts.to_csv('tseries.csv')

In [102]: !type tseries.csv
,0
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

6.1.3 구분자 형식 다루기

pandas_read_tables 함수를 이용해서 디스크에 표 형태로 저장된 대부분의 파일 형식을 불러올 수 있다. 하지만 수동으로 처리해야 하는 경우도 있다. read_table 함수가데이터를 불러오는데 실패하게끔 만드는 잘못된 라인이 포함되어 있는 데이터를 전달받는 경우도 종종 있다. 우선 작은 CSV 파일을 불러오는 과정으로 기본적인 도구 사용법을 익혀보자.

```
In [103]: !type ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

구분자가 한 글자인 파일은 파이썬 내장 csv모듈을 이용해서 처리할 수 있는데, 열려진 파일 객체를 csv.reader 함수에 넘기기만 하면 된다.

```
In [105]: import csv

In [106]: f = open('ex7.csv')

In [107]: reader = csv.reader(f)
```

파일을 읽듯이 reader를 순회하면 둘러싸고 있던 큰따움표가 제거된 튜플을 얻을 수 있다.

```
In [109]: for line in reader:
     ...:     print(line)
     ...:
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

이제 원하는 형태로 데이터를 넣을 수 있게 차근차근 따라 해보자. 먼저 파일을 읽어 줄 단위 리스트로 저장한다.

```
In [110]: with open('ex7.csv') as f:
     ...:     lines = list(csv.reader(f))
     ...:
```

그리고 헤더와 데이터를 구분한다.

```
In [111]: header, values = lines[0],lines[1:]
```

사전 표기법과 로우를 칼럼으로 전치해주는 zip(*values)를 이용해서 데이터 컬럼 사전을 만들어보자.

```
In [112]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [113]: data_dict
Out[113]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV 파일은 다양한 형태로 존재할 수 있다. 다양한 구분자, 문자열을 둘러싸는 방법, 개행 문자 같은 것들은 csv.Dialect를 상속받아 새로운 클래스를 정의해서 해결할 수 있다.

```
In [125]: class my_dialect(csv.Dialect):
     ...:     lineterminator = '\n'
     ...:     delimiter = ';'
     ...:     quotechar = '"'
     ...:     quoting = csv.QUOTE_MINIMAL

In [126]: reader = csv.reader(f,dialect = my_dialect)
```

```
In [136]: for line in reader:
     ...:     print(line)
     ...:
['a,"b","c"']
['1,"2","3"']
['1,"2","3"']
```

서브클래스를 정의하지 않고 csv.reader에 키워드 인자로 각각의 CSV 파일의 특징을 지정해서 전달해도 된다.

```
In [134]: reder = csv.reader(f,delimiter = '|')
```

사용 가능한 옵션(csv.Dialect의 속성)과 어떤 역활을 하는지에 대해서는 다음 표를 살펴보자.

인자	설명
delimiter	필드를 구분하기 위한 한 문자로 된 구분자. 기본값은 ','
lineterminator	파일을 저장할 때 사용할 개행 문자. 기본값은 '\r\n' , 파일을 읽을 때는 이 값을 무시하며, 자동으로 플랫폼별 개행 문자를 인식한다.
quotechar	각 필드에서 값을 둘러싸고 있는 문자.
quoting	값을 읽거나 쓸 때 둘러쌀 문자 컨벤션. csv.QUOTE_ALL(모든 필드에 적용), csv.QUOTE_MINIMAL(구분자 같은 특별한 문자가 포함된 필드만 적용), csv.QUOTE_NONE(값을 둘러싸지 않음) 옵션이 있다. 자세한 내용은 파이썬 문서를 참고해라. 기본값은 QUOTE_MINIMAL이다.
skipinitialspace	구분자 뒤에 공백 문자를 무시할지 여부. 기본값은 False
doublequote	값을 둘러싸는 문자가 필드 내에 존재할 경우 처리 여부. True일 경우 그 문자까지 모두 둘러싼다.
escapechar	quoting이 csv.QUOTE_NONE일 때, 값에 구분자와 같은 문자가 있을 경우 구별할 수 있도록 해주는 이스케이프 문자('\' 같은). 기본값은 None

NOTE_ 더 복잡하거나 한 글자를 초과하는 고정 길이를 가진다면 csv 모듈을 사용할 수 없다. 이 경우 줄을 나누고 문자열의 split 메서드나 정규 표현식 메서드인 re.split 등을 이용해서 가공하는 작업이 필요하다 .

CSV처럼 구분자로 구분된 파일을 **기록**하려면 csv,writer를 이용하면 된다. csv.writer는 이미 열린, 쓰기가 가능한 파일 객체를 받아서 csv.reader와 동일한 옵션으로 파일을 기록한다.

```
In [138]: with open('mydata.csv','w') as f:
     ...:     writer = csv.writer(f,dialect=my_dialect)
     ...:     writer.writerow(('one','two','three'))
     ...:     writer.writerow(('1','2','3'))
     ...:     writer.writerow(('4','5','6'))
     ...:     writer.writerow(('7','8','9'))
     ...:

In [139]: !type mydata.csv
one;two;three
1;2;3
4;5;6
7;8;9
```

6.1.4 JSON 데이터

JSON(Javascript Object Notation)은 웹브라우저와 다른 애플리케이션이 HTTP 요청으로 데이터를 보낼 때 널리 사용하는 표준 파일 형식 중 하나다. JSON은 CSV 같은 표 형식의 텍스트보다 보다 더 유연한 데이터 형식이다. 아래는 JSON데이터의 예이다.

JSON은 널값 NULL과 다른 몇 가지 미묘한 차이(리스트의 마지막에 쉼표가 있으면 안 되는 등)를 제외하면 파이썬 코드와 거의 유사하다. 기본 자료형은 객체(사전), 배열(리스트), 문자열, 숫자, 불리언, 그리고 널이다. 객체의 키는 반드시 문자열이어야 한다. JSON 데이터를 읽고 쓸 수 있는 파이썬 라이브러리가 몇 가지 있는데 여기서는 파이썬 표준 라이브러리인 json을 사용하겠다.

pandas.read_json은 자동으로 JSON 데이터셋을 Series나 DataFrame으로 변환할 수 있다.

```
In [148]: !type example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

별다른 옵션이 주어지지 않았을 경우, pandas.read_json은 JSON 배열에 담긴 각 객체를 테이블의 로우로 간주한다.

```
In [149]: data = pd.read_json('example.json')

In [150]: data
Out[150]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

좀 더 심화된 JSON 데이터를 읽고 다루는 (중첩된 레코드를 포함해서) 예제는 차후에 보기로 한다.

pandas 의 데이터를 JSON으로 저장하는 한 가지 방법은 to_json 함수를 이용하는 것이다.

```
In [154]: print(data.to_json(orient = 'records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]

In [155]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}
```

6.1.5 XML과 HTML : 웹 스크롤링

파이썬에는 lxml, Beautiful Soup 그리고 html5lib 같은 HTML과 XML 형식의 데이터를 읽고 쓸 수 있는 라이브러리가 무척 많다. 그중에서도 lxml은 가장 빠르게 동작하고 깨진 HTML과 XML파일도 잘 처리해 준다.

pandas에는 read_html 이라는 내장 함수가 있다. 이는 lxml이나 Beautiful Soup 같은 라이브러리를 사용해서 자동으로 HTML 파일을 파싱하여 DataFrame으로 변환해준다. 사용법을 알아보기 위해 미연방 예금보험공사에서 부도은행을 보여주는 HTML을 다운로드하자. 우선 read_html을 사용하기 위해 아래 라이브러리들을 설치해야 한다.

```
In [156]: pip install lxml
In [157]: pip install beautifulsoup4 html5lib
```

conda를 사용하지 않는다면 pip install lxml을 입력해도 설치가 가능하다.

pandas.read_html 함수에는 다양한 옵션이 있는데 기본적으로

태그 안에 있는 표 형식의 데이터 파싱을 시도한다. 결과는 DataFrame 객체의 리스트에 저장된다.

```
In [158]: table = pd.read_html('fdic_failed_bank_list.html')

In [159]: len(table)
Out[159]: 1

In [161]: failures = table[0]

In [162]: failures.head()
Out[162]:
                      Bank Name  ...       Updated Date
0                   Allied Bank  ...  November 17, 2016
1  The Woodbury Banking Company  ...  November 17, 2016
2        First CornerStone Bank  ...  September 6, 2016
3            Trust Company Bank  ...  September 6, 2016
4    North Milwaukee State Bank  ...      June 16, 2016

[5 rows x 7 columns]
```

failures 에는 컬럼이 많으므로 pandas는 \문자로 줄을 구분해서 보여준다. 나중에 더 살펴보겠지만 지금부터 데이터 정제와 연도별 부도은행 수 계산 등의 분석을 시작할 수 있다.

```
In [167]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [168]: close_timestamps
Out[168]:
0     2016-09-23
1     2016-08-19
2     2016-05-06
3     2016-04-29
4     2016-03-11
         ...
542   2001-07-27
543   2001-05-03
544   2001-02-02
545   2000-12-14
546   2000-10-13
Name: Closing Date, Length: 547, dtype: datetime64[ns]
```

```
In [169]: close_timestamps.dt.year.value_counts()
Out[169]:
2010    157
2009    140
2011     92
2012     51
2008     25
       ...
2004      4
2001      4
2007      3
2003      3
2000      2
Name: Closing Date, Length: 15, dtype: int64

In [170]: close_timestamps.dt.day.value_counts()
Out[170]:
23    31
30    30
19    30
20    27
17    23
      ..
12    10
9     10
31     8
1      8
3      3
Name: Closing Date, Length: 31, dtype: int64
```

lxml.objectify를 이용해서 XML파싱하기

XML(eXtensible Makeup Language)은 계층적 구조와 메타데이터를 포함하는 중첩된 데이터 구조를 지원하는 또 다른 유명한 데이터 형식이다.

앞에서는 HTML에서 데이터를 파싱하기 위해 내부적으로 lxml 또는 Beautiful Soup를 사용하는 pandas.read_html 함수를 살펴봤다. XML과 HTML은 구조적으로 유사하지만, XML이 좀 더 범용적이다. 여기서는 lxml를 이용해서 XML형식에서 데이터를 파싱하는 방법을 살펴보겠다.

뉴욕 MTA는 버스와 전철 운영에 관한 여러가지 데이터를 공개하고 있다. 그중에서 우리가 살펴볼 것은 여러 XML 파일로 제공되는 실적 자료다. 전철과 버스운영은 매월 다음과 비슷한 내용의 각기 다른 파일로 제공된다.

lxml.objectify를 이용해서 파일을 파싱한 후 getroot 함수를 이용해서 XML 파일의 루트노드에 대한 참조를 얻어오자.

```
In [175]:     path = 'Performance_MNR.xml'
     ...:     parsed = objectify.parse(open(path))
     ...:     root = parsed.getroot()
```

root.INDICATOR을 이용해서 모든 XML 엘리먼트를 끄집어낼 수 있다. 각각의 항목에 대해 몇몇 태그는 제외하고 태그 이름(YTD_ACTUAL 같은)을 키값으로 하는 사전을 만들어 낼 수 있다.

```
In [176]: data = []

In [177]: skip_fields = ['PARENT_SEQ','INDICATOR_SEQ',
     ...:                'DESIRED_CHANGE','DECIAMAL_PLACES',
     ...:                ]
     ...: for elt in root.INDICATOR:
     ...:     el_data = {}
     ...:     for child in elt.getchildren():
     ...:         if child.tag in skip_fields:
     ...:             continue
     ...:         el_data[child.tag] = child.pyval
     ...:     data.append(el_data)
```

이 사전 리스트를 DataFrame으로 변환하자.

```
In [178]: perf = pd.DataFrame(data)

In [179]: perf.head()
Out[179]:
            AGENCY_NAME  ... MONTHLY_ACTUAL
0  Metro-North Railroad  ...           96.9
1  Metro-North Railroad  ...             95
2  Metro-North Railroad  ...           96.9
3  Metro-North Railroad  ...           98.3
4  Metro-North Railroad  ...           95.8

[5 rows x 13 columns]

In [180]: perf
Out[180]:
              AGENCY_NAME  ... MONTHLY_ACTUAL
0    Metro-North Railroad  ...           96.9
1    Metro-North Railroad  ...             95
2    Metro-North Railroad  ...           96.9
3    Metro-North Railroad  ...           98.3
4    Metro-North Railroad  ...           95.8
..                    ...  ...            ...
643  Metro-North Railroad  ...
644  Metro-North Railroad  ...
645  Metro-North Railroad  ...
646  Metro-North Railroad  ...
647  Metro-North Railroad  ...

[648 rows x 13 columns]
```

XML 데이터를 얻으려면 지금 본 예제보다 훨씬 더 복잡한 과정을 거쳐야 한다. 각각의 태그 또한 메타데이터를 가지고 있을 수 있다. 유효한 XML 형식인 HTML의 태그를 생각하면 된다.

```
In [181]: from io import StringIO

In [182]: tag = '<a herf="https://www.google.com/">Google</a>'
     ...: root = objectify.parse(StringIO(tag)).getroot()
```

이제 태그나 링크 이름에서 어떤 필드 (href 같은)라도 접근이 가능하다.

```
In [183]: root
Out[183]: <Element a at 0x1fed0609948>

In [184]: root.get('href')

In [185]: root.get('href')

In [186]: root.text
Out[186]: 'Google'
```

6.2 이진 데이터 형식

데이터를 효율적으로 저장하는 가장 손쉬운 방법은 파이썬에 기본으로 내장되어 있는 pickle **직렬화**

를 사용해서 데이터를 이진 형식으로 저장하는 것이다. 편리하게도 pandas 객체는 모두 pickle을 이용해서 데이터를 저장하는 to_pickle 메서드를 가지고 있다.

```
In [188]: frame = pd.read_csv('ex1.csv')

In [189]: frame
Out[189]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [190]: frame.to_pickle('frame_pickle')
```

pickle로 직렬화된 객체는 내장 함수인 pickle로 직접 불러오거나 아니면 좀 더 편리한 pickle 함수인 pandas.read_pickle 메서드를 이용하여 불러올 수 있다.

```
In [192]: pd.read_pickle('frame_pickle')
Out[192]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

CAUTION_ pickle은 오래 보관할 필요가 없는 데이터일 경우에만 추천한다. 오랜 시간이 지나도 안정적으로 데이터를 저장할 거라고 보장하기 힘든 문제가 있기 때문이다. 최근에 pickle을 이용해서 저장한 데이터는 나중에 라이브러리 버전이 올라갔을 때 다시 읽어오지 못할 가능성이 있기 때문이다. 많은 노력을 기울여 이 문제를 검증해보았는데 pandas에서는 문제가 해결 되지 않았다. 하지만 앞으로 언젠가 pickle로 저장된 데이터를 하나씩 까봐야 하는 일이 생길 수도 있다.

pandas는 HDF5와 Message-Pack, 두 가지 바이너리 포맷을 지원한다. 다음 절에서 HDF5예제를 살펴보겠지만 다양한 파일 형식이 실제 독자들의 분석 작업에 얼마나 더 적절한지 직접 살펴보기 권장한다. 다음과 같은 pandas 또는 NumPy 데이터를 위한 다른 저장 형식도 존재한다.

- Boolz

Blocs 압축 알고리즘에 기반한 압축이 가능한 칼럼지향 바이너리 포맷이다.

- Feather

R 커뮤니티의 해들리 위컴과 내가 함께 설계한 컬럼지향 파일 형식이다. Feather는 아파치 에로우의 메모리 포맷을 사용한다.

6.2.1 HDF5 형식 사용하기

HDF5는 대량의 과학 계산용 배열 데이터를 저장하기 위해 고안된 훌륭한 파일 포맷이다. C 라이브러리로도 존재하며 자바, 줄리아, 매트랩 그리고 파이썬 같은 다양한 다른 언어에서도 사용할 수 있는 인터페이스를 제공한다. HDF는 Hirerarchical Data Format의 약자로 계층적 데이터 형식이라는 뜻이다. 각각의 HDF5 파일은 여러 개의 데이터 셋을 저장하고 부가 정보를 기록할 수 있다. 보다 단순한 형식과 비교하면 HDF5는 다양한 압축 기술을 사용해서 온더플라이(on-the-fly(실시간)) 압축을 지원하며 반복되는 패턴을 가진 데이털르 좀 더 효과적으롤 저장할 수 있다. 메모리에 모두 적재할 수 없는 엄청나게 큰 데이터를 아주 큰 배열에서 필요한 작은 부분들만 효과적으로 읽고 쓸 수 있는 훌륭한 선택이다.

PyTables 나 h5py 라이브러리를 이용해서 직접 HDF5 파일에 접근하는 것도 가능하지만 pandas 는 Series 나 DataFrame 객체로 간단히 저장할 수 있는 고수준의 인터페이스를 제공한다. HDFStore 클래스는 사전처럼 작동하며 세밀한 요구 사항도 잘 처리해준다.

```
In [194]: frame = pd.DataFrame({'a':np.random.randn(100)})

In [195]: frame
Out[195]:
           a
0   1.243393
1  -0.465976
2  -0.761733
3   0.171352
4   1.502075
..       ...
95  0.752463
96  0.256178
97  0.043906
98 -1.434741
99 -0.760151

[100 rows x 1 columns]

In [205]: store = pd.HDFStore('mydata.h5')

In [206]: store['obj1'] = frame

In [207]: store['obj1_col'] = frame['a']

In [208]: store
Out[208]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
```

HDFStore는 'fixed'와 'table' 두 가지 저장 스키마를 지원한다. 'table' 스키마가 일반적으로 더 느리지만 아래와 같은 특별한 문법을 이용해 쿼리 연산을 지원한다.

```
In [210]: store.put('obj2', frame, format='table')

In [211]: store.select('obj2',where = ['index >=10 and index<=15'])
Out[211]:
           a
10  0.353067
11 -0.721632
12  0.564363
13 -0.673285
14  0.696291
15 -1.015128

In [212]: store.close()
```

put은 명시적인 store['obj2'] = frame 메서드지만 저장 스키마를 지정하는 등의 다른 옵션을 제공한다.

pandas,read_hdf 함수는 이런 기능들을 축약해서 사용할 수 있다.

```
In [227]: frame.to_hdf('store','obj3',format = 'table')

In [228]: pd.read_hdf('store','obj3',where=['index < 5'])
Out[228]:
          a
0  1.243393
1 -0.465976
2 -0.761733
3  0.171352
4  1.502075
```

NOTE_ 만일 아마존 S3 이나 HDF5 같은 원격 서버에 저장된 데이터를 처리해야 한다면 아파치 파케이(Parquet) 같은 분산 저장소를 고려하여 설계된 다른 바이너리 형식을 사용하는 편이 좀 더 올바른 선택일 수 있다. 아직 파케이나 다른 저장 형식은 파이썬을 지원하기 위한 개발이 진행중이므로 이 책에서는 따로 설명하지 않겠다.

만약 로컬 스토리지에서 엄청난 양의 데이터를 다뤄야 한다면 PyTable와 h5py를 살펴보고 목적에 맞는지 알아보기를 권장한다. 실제로 대부분의 데이터 분석 문제는 CPU보다는 IO 성능에 으존적이므로 HDF5 같은 도구를 사용하면 애플리케이션의 성능을 어마어마하게 향상시킬 수 있다.

CAUTION_ HDF5는 DB가 아니다. HDF5는 한 번만 기록하고 자주 여러 번 읽어야 하는 데이터에 최적화되어 있다. 데이터는 아무 때나 파일에 추가할 수 있지만 만약 여러 곳에서 동시에 파일에 추가한다면 파일이 깨지는 문제가 발생할 수 있다.

6.2.2 마이크로소프트 엑셀 파일에서 데이터 읽어오기

pandas는 ExcelFile 클래스나 pandas.read_excel 함수를 사용해서 마이크로소프트 엑셀 2003 이후 버전의 데이터를 읽어올 수 있다. 내부적으로 이들 도구는 XLS 파일과 XLSX 파일을 읽기 위해 각각 xlrd와 openpyxl 패키지를 이용하므로 사용하기 전에 pip나 conda 명령을 이용해 두 패키지를 설치해야 한다.

```
In [235]: xlsx = pd.ExcelFile('ex1.xlsx')
```

시트에 있는 데이터는 parse 함수를 이용해서 DataFrame으로 읽어올 수 있다.

```
In [236]: pd.read_excel(xlsx,'Sheet1')
Out[236]:
   Unnamed: 0  a   b   c   d message
0           0  1   2   3   4   hello
1           1  5   6   7   8   world
2           2  9  10  11  12     foo
```

한 파일에서 여러 시트를 읽어오려면 ExcelFile을 생성하면 빠르지만 간단하게는 pandas.read_excel 에 파일 이름만 넘겨도 된다.

```
In [237]: frame = pd.read_excel('ex1.xlsx','Sheet1')

In [238]: frame
Out[238]:
   Unnamed: 0  a   b   c   d message
0           0  1   2   3   4   hello
1           1  5   6   7   8   world
2           2  9  10  11  12     foo
```

pandas 데이터를 엑셀 파일로 저장하고 싶다면 ExcelWrite를 생성해서 데이터를 기록하고 pandas 객체의 to_excel 메서드로 넘기면 된다.

```
In [239]: writer = pd.ExcelWriter('ex2.xlsx')

In [240]: frame.to_excel(writer,'Sheet1')

In [241]: writer.save()
```

ExcelWriter를 사용하지 않고 to_excel 메서드에 파일 경로만 넘겨도 좋다.

```
In [242]: frame.to_excel('ex2.xlsx')
```

6.3 웹 API와 함께 사용하기

데이터 피드를 JSON이나 여타 다른 형식으로 얻을 수 있는 공개 API를 제공하는 웹사이트가 많다.

파이썬으로 이 API를 사용하는 방법은 다양한데, 내가 추천하는 가장 손쉬운 방법은 requests 패키지를 이용하는 것이다.

pandas 깃허브에서 최근 30개의 이슈를 가져오려면 requests 라이브러리를 이용해서 다음과 같은 GET HTTP 요청을 생성하면 된다.

```
In [243]: import requests

In [244]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'
     ...:

In [245]: resp = requests.get(url)

In [246]: resp
Out[246]: <Response [200]>
```

응답 객체의 json 메서드는 JSON의 내용을 파이썬 사전 형태로 변환한 객체를 반환한다.

```
In [247]: data = resp.json()

In [249]: data[0]['title']
Out[249]: 'DEP: bump numpy min version?'
```

data의 각 항목은 깃허브 이슈 페이지(댓글 제외)에서 찾을 수 있는 모든 데이터를 담고 있다.

이 data를 바로 DataFrame으로 생성하고 관심이 있는 필드만 따로 추출할 수 있다.

```
In [250]: issues = pd.DataFrame(data, columns=['number','title','labels
     ...: ','state'])

In [251]: issues
Out[251]:
    number  ... state
0    33718  ...  open
1    33717  ...  open
2    33716  ...  open
3    33715  ...  open
4    33714  ...  open
..     ...  ...   ...
25   33676  ...  open
26   33675  ...  open
27   33674  ...  open
28   33673  ...  open
29   33671  ...  open

[30 rows x 4 columns]
```

조금만 더 수고하면 평범한 웹 API를 위한 고수준의 인터페이스를 만들어서 DataFrame에 저장하고 쉽게 분석 작업을 수행할 수 있다.

6.4 DB와 함께 사용하기

비즈니스 관점에서 대부분의 데이터는 텍스트 파일이나 엑셀 파일로 저장하지 않고 SQL기반의 관계형 DB(SQL서버, PostgreSQL, MySQL)를 많이 사용하는데, 다른 종류의 대안 DB들도 꽤 인기를 끌고 있다. DB는 보통 애플리케이션에서 필요한 성능이나 데이터 무결성 그리고 확장성에 맞춰서 선택하는 것이 일반적이다.

SQL에서 데이터를 읽어와서 DataFrame에 저장하는 것은 꽤 직관적이며 pandas에는 이 과정을 간결하게 해주는 몇 가지 함수가 있다. 한 예로 파이썬 내장 sqlite3 드라이버를 사용해서 SQLite DB를 이용할 수 있다.

```
In [252]: import sqlite3

In [253]: query = """
     ...: CREATE TABLE test
     ...: (a VARCHAR(20),
     ...: b VARCHAR(20),
     ...: c REAL,
     ...: d INTECER);
     ...: """

In [254]: con = sqlite3.connect('mydata.sqlite')
```

```
In [255]: con.execute(query)
Out[255]: <sqlite3.Cursor at 0x1fed788ce30>

In [256]: con.commit()
```

이제 데이터를 몇 개 입력한다.

```
In [257]: data = [('Atlanta','Georgia',1.25,6),]

In [258]: data = [('Tallahassee','Florida',2.6,3),
     ...:         ('Sacramento','California',1.7,5)
     ...: ]

In [259]: stmt = "INSERT INTO test VALUES(?,?,?,?)"

In [260]: con.executemany(stmt,data)
Out[260]: <sqlite3.Cursor at 0x1fed788c570>

In [261]: con.commit()
```

대부분의 파이썬 SQL 드라이버(PyODBC,psycopg2,MySQLdb, pymssql 등)는 테이블에 대해 select 쿼리를 수행하면 튜플 리스트를 반환한다.

```
In [262]: cursor = con.execute('select * from test')

In [263]: rows = cursor.fetchall()

In [264]: rows
Out[264]: [('Tallahassee', 'Florida', 2.6, 3),
           ('Sacramento', 'California', 1.7, 5)]
```

반환된 튜플 리스트를 DataFrame 생성자에 바로 전달해도 되지만, 컬럼 이름을 정해주면 더 편하다.

cursor의 description 속성을 활용하자.

```
In [272]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description]
     ...: )
Out[272]:
             a           b    c  d
0  Tallahassee     Florida  2.6  3
1   Sacramento  California  1.7  5

In [273]: pd.DataFrame(rows, columns=[x[1] for x in cursor.description]
     ...: )
Out[273]:
           NaN         NaN  NaN  NaN
0  Tallahassee     Florida  2.6    3
1   Sacramento  California  1.7    5

In [274]: cursor.description
Out[274]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))
```

DB에 쿼리를 보내기 위해 매번 이렇게 하는 건너무 귀찮은 일이다. 유명한 파이썬 SQL 툴킷인 SQLAlchemy(SQL알케미) 프로젝트는 SQL DB간의 일반적인 차이점을 추상화하여 제공한다. pandas는 real_sql 함수를 제공하여 SQLAlchemy의 일반적인 연결을 이용해 쉽게 데이터를 읽을 수 있게 한다. 다음 SQLAlchemy 를 사용하여 같은 SQLite DB에 접속하고 앞서 만든 테이블에서 데이터를 읽어오는 예이다.

```
In [275]: import sqlalchemy as sqla

In [276]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [277]: pd.read_sql('select * from test',db)
Out[277]:
             a           b    c  d
0  Tallahassee     Florida  2.6  3
1   Sacramento  California  1.7  5
```

6.5 마치며

데이터에 접근하는 것은 데이터 분석 과정의 첫 번째 관문이다. 이 장에서는 이 관문을 통과하는 데 도움이 될 만한 여러 가지 도구를 살펴봤다. 다음 장에서는 데이터 정제, 시각화, 시계열 분석 및 다른 주제를 좀 더 깊이 살펴보겠다.