第 6 章

数据加载、存储与文件格式

读取数据并使数据可访问（称为数据加载）是使用本书所介绍的大部分工具的第一步。还有另外一个术语“解析”，用于描述加载文本数据并将其解释为表格和不同的数据类型。我会着重介绍 pandas 的数据输入与输出，虽然别的库中也有不少用于读写不同格式数据的工具。

数据输入输出通常可以划分为几大类：读取文本文件和其他更有效的磁盘存储格式，加载数据库中的数据，以及与网络资源（比如 Web API）交互。

6.1 读写文本格式的数据

pandas 提供了一些用于将表格型数据读取为 DataFrame 对象的函数。表 6-1 对其中一些进行了总结，其中 `read_csv` 是本书中使用最多的。在 6.2 节中，我们会进一步学习二进制数据格式。

表 6-1：pandas 中的文本和二进制数据的加载函数

函数	说明
`read_csv`	从文件、URL、文件型对象中加载带分隔符的数据，默认分隔符为逗号
`read_fwf`	读取特定宽度列格式的数据（无分隔符）
`read_clipboard`	读取剪贴板中的数据，可以看作 `read_table` 的剪贴板版，在将网页转换为表格时很有用
`read_excel`	从 Excel XLS 或 XLSX 文件中读取表格数据
`read_hdf`	读取 pandas 存储的 HDF5 文件
`read_html`	读取 HTML 文档中的所有表格
`read_json`	读取 JSON 字符串、文件、URL 或文件型对象中的数据
`read_feather`	读取 Feather 二进制文件格式

表 6-1：pandas 中的文本和二进制数据的加载函数（续）

函数	说明
read_orc	读取 Apache ORC 二进制文件格式
read_parquet	读取 Apache Parquet 二进制文件格式
read_pickle	读取 pandas 使用 Python pickle 格式存储的对象
read_sas	读取存储于 SAS 系统的自定义存储格式的 SAS 数据集
read_spss	读取 SPSS 创建的数据文件
read_sql	读取 SQL 查询的结果（使用 SQLAlchemy）
read_sql_table	读取整个 SQL 表（使用 SQLAlchemy），等价于使用 read_sql 读取表中的所有数据
read_stata	读取 Stata 文件格式的数据集
read_xml	读取 XML 文件中的数据表格

我会大致介绍一下这些函数的机制，它们旨在将文本数据转换为 DataFrame。这些函数的选项可以划分为以下几类：

索引

将一列或多列作为返回的 DataFrame，以及是否从文件、参数获取列名。

类型推断和数据转换

包括用户定义的值转换和自定义的缺失值标记列表。

日期和时间解析

包括组合功能，比如将分散在多个列中的日期和时间信息组合成结果中的单个列。

迭代

支持对大文件进行逐块迭代。

不规整数据问题

跳过行、页脚、注释或其他不重要的内容（比如由数千个逗号分隔的数值数据）。

因为实际工作中碰到的数据可能十分混乱，一些数据加载函数（尤其是 pandas.read_csv）的选项逐渐变得复杂。面对不同的参数，感到头痛很正常（pandas.read_csv 的参数超过 50 个）。pandas 线上文档有大量示例展示这些参数是如何工作的，如果你感到读取某个文件很困难，可以通过相似的示例找到正确的参数。

其中一些函数会进行类型推断，因为列的数据类型不属于数据类型。也就是说，无须指定列的类型到底是数值、整数、布尔值还是字符串。其他的数据格式，如 HDF5、ORC 和 Parquet，会在格式中嵌入数据类型。

处理日期和其他自定义类型的处理需要多花点时间。

首先我们来看一个小型的逗号分隔值（CSV）文本文件：

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

这里我使用Unix的cat命令来打印文件的原始内容。如果你用的是Windows，则可以使用type达到同样的效果。

由于该文件以逗号分隔，因此可以使用pandas.read_csv将其读入DataFrame：

```
In [11]: df = pd.read_csv("examples/ex1.csv")

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

并不是所有文件都有表头。看下面这个文件：

```
In [13]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

要读取这个文件，有几个可选项。可以让pandas用默认的列名来赋值，也可以自己指定列名：

```
In [14]: pd.read_csv("examples/ex2.csv", header=None)
Out[14]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [15]: pd.read_csv("examples/ex2.csv", names=["a", "b", "c", "d", "message"])
Out[15]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

假如你想用message列来作为返回的DataFrame的索引，既可以指明索引为4的列，也可以使用index_col参数指明使用"message"：

```
In [16]: names = ["a", "b", "c", "d", "message"]

In [17]: pd.read_csv("examples/ex2.csv", names=names, index_col="message")
Out[17]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

如果希望将多列做成层次化索引（将在 8.1 节中介绍），只需传入由列编号或列名组成的列表即可：

```
In [18]: !cat examples/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 [19]: parsed = pd.read_csv("examples/csv_mindex.csv",
   ....:                      index_col=["key1", "key2"])

In [20]: parsed
Out[20]:
           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
```

在某些情况下，有些表格使用的可能不是固定的分隔符，而是用空白符或其他方式分隔字段。来看下面这个文本文件：

```
In [21]: !cat examples/ex3.txt
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
```

虽然可以手动对数据进行规整，但这里的字段是被不同数量的空白字符间隔开的，手动比较麻烦。这种情况下，可以传入一个正则表达式作为 pandas.read_csv 的分隔符。正

则表达式可以是 \s+，于是有：

```
In [22]: result = pd.read_csv("examples/ex3.txt", sep="\s+")

In [23]: result
Out[23]:
            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
```

本例中，由于列名的数量比数据的行数少一个，所以 pandas.read_csv 推断第一列作为 DataFrame 的索引。

文件解析器函数还有许多参数可以帮助你处理各种各样的特例文件格式（表 6-2 列举了一些）。例如，可以用 skiprows 跳过下面文件的第一行、第三行和第四行：

```
In [24]: !cat examples/ex4.csv
# 嘿!
a,b,c,d,message
# 为你制造一些困难
# 毕竟，还有什么人会用计算机读取 CSV 文件呢?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [25]: pd.read_csv("examples/ex4.csv", skiprows=[0, 2, 3])
Out[25]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

处理缺失值是文件读取过程中重要但烦琐的环节。缺失数据经常是要么丢失（空字符串），要么用某些标记值（占位符）表示。默认情况下，pandas 会使用常见的标识，比如 NA 和 NULL：

```
In [26]: !cat examples/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 [27]: result = pd.read_csv("examples/ex5.csv")

In [28]: result
Out[28]:
  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
```

pandas 会将缺失值输出为 NaN，因此 result 中有两个缺失值：

```
In [29]: pd.isna(result)
Out[29]:
   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 可以接收字符串序列，将其添加到默认的缺失值字符串列表中：

```
In [30]: result = pd.read_csv("examples/ex5.csv", na_values=["NULL"])

In [31]: result
Out[31]:
  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
```

pandas.read_csv 有多种默认的 NA 标记值，可以用选项 keep_default_na 使其失效：

```
In [32]: result2 = pd.read_csv("examples/ex5.csv", keep_default_na=False)

In [33]: result2
Out[33]:
  something  a   b   c   d message
0       one  1   2   3   4      NA
1       two  5   6       8   world
2     three  9  10  11  12     foo

In [34]: result2.isna()
Out[34]:
   something      a      b      c      d  message
0      False  False  False  False  False    False
1      False  False  False  False  False    False
2      False  False  False  False  False    False

In [35]: result3 = pd.read_csv("examples/ex5.csv", keep_default_na=False,
   ....:                       na_values=["NA"])

In [36]: result3
Out[36]:
  something  a   b   c   d message
0       one  1   2   3   4     NaN
1       two  5   6       8   world
2     three  9  10  11  12     foo

In [37]: result3.isna()
Out[37]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False  False  False    False
2      False  False  False  False  False    False
```

对于字典中的各列，可以指定不同的缺失值标识：

```
In [38]: sentinels = {"message": ["foo", "NA"], "something": ["two"]}

In [39]: pd.read_csv("examples/ex5.csv", na_values=sentinels,
   ....:             keep_default_na=False)
Out[39]:
  something  a   b   c   d message
0       one  1   2   3   4     NaN
1       NaN  5   6       8   world
2     three  9  10  11  12     NaN
```

表 6-2 列举了 pandas.read_csv 中常用的选项。

表 6-2：pandas.read_csv 的部分函数参数

参数	说明
path	表示文件系统位置、URL、文件型对象的字符串
sep 或 delimiter	用于对行中各字段进行拆分的字符序列或正则表达式
header	用作列名的行号。默认为 0（第一行），如果没有表头则为 None
index_col	用作结果中行索引的列编号或列名。可以是单个名称或数字，也可以是名称或数字组成的列表，用于层次化索引
names	用于结果的列名列表
skiprows	从文件开始处算起，需要忽略的行数或需要跳过的行数列表（从 0 开始）
na_values	一组用于替换 NA 的值序列。将其添加到默认列表，如果 keep_default_na=False，则不添加
keep_default_na	是否使用默认的 NA 值列表（默认为 True）
comment	用于在行尾分隔注释的字符（一个或多个）
parse_dates	尝试将数据解析为 datetime，默认为 False。如果为 True，则尝试解析所有列。此外，还可以指定需要解析的一组列号或列名的列表。如果列表的元素为列表或元组，就会将多个列组合到一起再进行日期解析（例如，日期或时间分别位于两列中）
keep_date_col	如果连接多列来解析日期，则保留已连接的列。默认为 False
converters	由列号或列名与函数之间的映射关系组成的字典（例如，{"foo": f} 会对 foo 列的所有值应用函数 f）
dayfirst	当解析有歧义的日期时，将其按国际格式处理（例如，7/6/2012 处理为 June 7, 2012）。默认为 False
date_parser	用于解析日期的函数
nrows	从文件开始处算起，需要读取的行数（不包括表头）
iterator	返回一个 TextFileReader，用于逐块读取文件。该对象还可以用于 with 语句
chunksize	用于迭代的文件块的大小
skip footer	文件末尾处需要忽略的行数

表 6-2：pandas.read_csv 的部分函数参数（续）

参数	说明
verbose	打印各种解析器输出信息，比如文件转换每个阶段的用时和内存使用信息
encoding	用于 unicode 的文本编码格式。例如，“utf-8”表示用 UTF-8 编码的文本
squeeze	如果数据经解析后仅含一列，则返回 Series
thousands	千分位分隔符，如“,”或“.”，默认没有分隔符
decimal	小数点分隔符，如“,”或“.”，默认是“.”
engine	用于解析和转换 CSV 文件的引擎，可以是“c”“python”“pyarrow”其中之一。默认引擎是“c”，但“pyarrow”对某些文件的转换速度更快。“python”稍慢，但支持另外两种引擎没有的功能

6.1.1 逐块读取文本文件

在处理大文件或找出正确的参数集以处理大文件时，你可能只想读取文件的一小部分或逐块对文件进行迭代。

在查看大文件之前，我们先设置 pandas，使显示更加紧凑：

```
In [40]: pd.options.display.max_rows = 10
```

然后读取文件：

```
In [41]: result = pd.read_csv("examples/ex6.csv")

In [42]: result
Out[42]:
           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]
```

省略号表示忽略 DataFrame 中间的行。

如果只想读取几行（避免读取整个文件），通过 nrows 进行指定即可：

```
In [43]: pd.read_csv("examples/ex6.csv", nrows=5)
Out[43]:
        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
```

要逐块读取文件，可以指定 chunksize 作为行数：

```
In [44]: chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)

In [45]: type(chunker)
Out[45]: pandas.io.parsers.readers.TextFileReader
```

pandas.read_csv 所返回的 TextFileReader 对象可以根据 chunksize 对文件进行逐块迭代。比如说，我们可以迭代处理 ex6.csv，连接 "key" 列中的值，如下所示：

```
chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)

tot = pd.Series([], dtype='int64')
for piece in chunker:
    tot = tot.add(piece["key"].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

然后得到：

```
In [47]: tot[:10]
Out[47]:
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
```

TextFileReader 还有一个 get_chunk 方法，用于读取任意大小的数据块。

6.1.2 将数据写入文本格式

数据也可以输出为分隔符格式的文本。来看之前读取过的一个 CSV 文件：

```
In [48]: data = pd.read_csv("examples/ex5.csv")

In [49]: data
```

```
Out[49]:
  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
```

利用 DataFrame 的 to_csv 方法，可以将数据导出到逗号分隔的文件中：

```
In [50]: data.to_csv("examples/out.csv")

In [51]: !cat examples/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
```

当然，还可以使用其他分隔符（由于这里直接输出到 sys.stdout，因此打印文本结果）：

```
In [52]: import sys

In [53]: 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
```

缺失值在输出结果中会表示为空字符串。你可能希望将其表示为其他标记值：

```
In [54]: 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
```

如果没有设置其他选项，则会输出行和列的标签。当然，它们也都可以被禁用：

```
In [55]: 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
```

此外，你还可以只输出列的一部分，并以指定的顺序排列：

```
In [56]: data.to_csv(sys.stdout, index=False, columns=["a", "b", "c"])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

6.1.3 使用其他分隔符格式

大部分存储在磁盘上的表格型数据都能用 pandas.read_csv 进行加载。然而，有时还是

需要做一些手工处理。由于接收含有错误行的文件而使 pandas.read_csv 出问题的情况并不少见。为了说明这些基本工具，来看下面这个 CSV 文件：

```
In [57]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

对于任何单字符分隔符文件，可以直接使用 Python 内置的 csv 模块。将任意已打开的文件或文件型对象传给 csv.reader：

```
In [58]: import csv

In [59]: f = open("examples/ex7.csv")

In [60]: reader = csv.reader(f)
```

对这个 reader 进行迭代会产生值列表，并删除所有值的引号：

```
In [61]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']

In [62]: f.close()
```

现在，为了使数据格式合乎要求，你需要对其做一些规整工作。我们一步一步来做。首先，读取文件到一个多行的列表中：

```
In [63]: with open("examples/ex7.csv") as f:
   ....:     lines = list(csv.reader(f))
```

然后，将这些行分为表头行和数据行：

```
In [64]: header, values = lines[0], lines[1:]
```

然后，用字典推导式和 zip(*values) 来创建数据列的字典，后者用于将行转置为列（对于大文件，会占用大量内存）：

```
In [65]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [66]: data_dict
Out[66]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV 文件的形式有很多。只需定义 csv.Dialect 的一个子类即可定义出新格式（如专门的分隔符、字符串引用约定、行终止符等）：

```
class my_dialect(csv.Dialect):
    lineterminator = "\n"
```

```
        delimiter = ";"
        quotechar = '"'
        quoting = csv.QUOTE_MINIMAL
    reader = csv.reader(f, dialect=my_dialect)
```

也可以用关键字的形式将 CSV 格式参数提供给 csv.reader，而无须定义子类：

```
reader = csv.reader(f, delimiter="|")
```

表 6-3 列出了可用的选项（csv.Dialect 的属性）及其功能。

表 6-3：CSV dialect 的选项

参数	说明
delimiter	用于分隔字段的单字符字符串。默认为“,”
lineterminator	用于写操作的行终止符，默认为“\r\n”。读操作将忽略此选项，并能识别跨平台的行终止符
quotechar	用于带有特殊字符（如分隔符）字段的引用符号。默认为“"”
quoting	引用惯例。可选值包括 csv.QUOTE_ALL（引用所有字段）、csv.QUOTE_MINIMAL（只引用带有诸如分隔符之类特殊字符的字段）、csv.QUOTE_NONNUMERIC 以及 csv.QUOTE_NON（不引用）。完整信息请参考 Python 文档。默认为 QUOTE_MINIMAL
skipinitialspace	忽略分隔符后面的空白符。默认为 False
doublequote	如何处理字段内的引用符号。如果为 True，则双写（完整信息请参见在线文档）
escapechar	分隔符，如果设置 quoting 为 csv.QUOTE_NONE，则用于对分隔符进行转义。默认禁用

对于那些使用复杂分隔符或固定多字符分隔符的文件，csv 模块就无能为力了。这种情况下，你就只能使用字符串的 split 方法或正则表达式方法 re.split 来做行拆分和其他清洗工作了。幸好，只要传入必要的选项，pandas.read_csv 就能胜任绝大部分工作，所以基本不需要手动解析文件。

要手工输出分隔符文件，可以使用 csv.writer。它接收一个已打开且可写的文件对象，以及与 csv.reader 相同的语法和格式化选项：

```
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"))
```

6.1.4 JSON 数据

JSON 已经成为通过 HTTP 请求在 Web 浏览器和其他应用程序之间发送数据的标准格式之一。这是一种比表格型文本格式（如 CSV）灵活得多的数据格式。下面是一个例子：

```
obj = """
{"name": "Wes",
 "cities_lived": ["Akron", "Nashville", "New York", "San Francisco"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars", "soccer"]},
              {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}]
}
"""
```

除了空值 null 和一些其他的细微差别（如列表末尾不允许存在多余的逗号）之外，JSON 非常接近于正规的 Python 代码。JSON 的基本类型有对象（字典）、数组（列表）、字符串、数值、布尔值以及空值。对象中所有的键都必须是字符串。存在多个 Python 库可以读写 JSON 数据。这里我将使用 json 包，因为它内置于 Python 标准库。通过 json.loads 即可将 JSON 字符串转换成 Python 形式：

```
In [68]: import json

In [69]: result = json.loads(obj)

In [70]: result
Out[70]:
{'name': 'Wes',
 'cities_lived': ['Akron', 'Nashville', 'New York', 'San Francisco'],
 'pet': None,
 'siblings': [{'name': 'Scott',
   'age': 34,
   'hobbies': ['guitars', 'soccer']},
  {'name': 'Katie', 'age': 42, 'hobbies': ['diving', 'art']}]}
```

json.dumps 将 Python 对象转换成 JSON 格式：

```
In [71]: asjson = json.dumps(result)

In [72]: asjson
Out[72]: '{"name": "Wes", "cities_lived": ["Akron", "Nashville", "New York", "San
 Francisco"], "pet": null, "siblings": [{"name": "Scott", "age": 34, "hobbies": [
"guitars", "soccer"]}, {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}
]}'
```

用户可以自行决定如何将（一个或一组）JSON 对象转换为 DataFrame 或其他便于分析的数据结构。最简单方便的方式是向 DataFrame 构造器传入一个字典的列表（就是上面的 JSON 对象），并选取数据字段的子集：

```
In [73]: siblings = pd.DataFrame(result["siblings"], columns=["name", "age"])

In [74]: siblings
Out[74]:
    name  age
0  Scott   34
1  Katie   42
```

pandas.read_json 可以自动将 JSON 数据集转换为指定形式的 Series 或 DataFrame。例如：

```
In [75]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

pandas.read_json 的默认选项假设 JSON 数组中的每个对象是表格中的一行：

```
In [76]: data = pd.read_json("examples/example.json")

In [77]: data
Out[77]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

如需了解读取和处理 JSON 数据（包括嵌套记录）的拓展示例，请参考第 13 章中关于 USDA 食品数据库的示例。

如果需要将数据从 pandas 输出到 JSON，可以对 Series 和 DataFrame 使用 to_json 方法：

```
In [78]: data.to_json(sys.stdout)
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}
In [79]: data.to_json(sys.stdout, orient="records")
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

6.1.5 XML 和 HTML：网络抓取

Python 有许多可以读写常见的 HTML 和 XML 格式数据的库，包括 lxml、Beautiful Soup 和 html5lib。lxml 的速度相对较快，但其他的库处理有误的 HTML 或 XML 文件更好。

pandas 有一个内置函数 pandas.read_html，可以使用所有这些解析库自动将 HTML 文件中的表格解析为 DataFrame 对象。为了进行展示，我从美国联邦存款保险公司下载了一个 HTML 文件（pandas 文档中也使用过），它记录了银行倒闭的数据[注 1]。首先，你需

注 1：完整列表见 *https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-list/banklist.html*。

要安装 read_html 用到的库：

```
conda install lxml beautifulsoup4 html5lib
```

如果你用的不是 conda，可以使用 pip install lxml。

pandas.read_html 有一些选项，默认条件下它会搜索、尝试解析 <table> 标签内的所有表格数据。结果是 DataFrame 对象的列表：

```
In [80]: tables = pd.read_html("examples/fdic_failed_bank_list.html")

In [81]: len(tables)
Out[81]: 1

In [82]: failures = tables[0]

In [83]: failures.head()
Out[83]:
                      Bank Name             City  ST   CERT  \
0                   Allied Bank         Mulberry  AR     91
1  The Woodbury Banking Company         Woodbury  GA  11297
2        First CornerStone Bank  King of Prussia  PA  35312
3            Trust Company Bank          Memphis  TN   9956
4    North Milwaukee State Bank        Milwaukee  WI  20364
                 Acquiring Institution        Closing Date       Updated Date
0                        Today's Bank  September 23, 2016  November 17, 2016
1                         United Bank     August 19, 2016  November 17, 2016
2  First-Citizens Bank & Trust Company         May 6, 2016  September 6, 2016
3           The Bank of Fayette County      April 29, 2016  September 6, 2016
4  First-Citizens Bank & Trust Company      March 11, 2016      June 16, 2016
```

因为 failures 有许多列，所以 pandas 插入了一个换行符 \。

我们在这里可以做一些数据清洗和分析工作，比如按年份计算倒闭的银行数：

```
In [84]: close_timestamps = pd.to_datetime(failures["Closing Date"])

In [85]: close_timestamps.dt.year.value_counts()
Out[85]:
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
```

利用 lxml.objectify 解析 XML

XML 是另一种常见的支持分层、嵌套数据以及元数据的结构化数据格式。本书实际上就是从一系列大型 XML 文档生成的。

前面，我介绍了 `pandas.read_html` 函数，它可以使用 lxml 或 Beautiful Soup 从 HTML 解析数据。XML 和 HTML 的结构很相似，但 XML 更为通用。这里，我会用一个例子演示如何利用 lxml 从 XML 格式解析数据。

多年以来，纽约大都会运输署（MTA）用 XML 格式发布了大量公交和列车服务方面的数据资料。这里，我们将看看包含在一组 XML 文件中的运行情况数据。每个列车或公交服务都有各自的文件（如 Metro-North Railroad 的文件是 *Performance_MNR.xml*），其中每条 XML 记录就是一条月度数据，如下所示：

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
  systemwide. The availability rate is based on physical observations performed
  the morning of regular business days only. This is a new indicator the agency
  began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

我们使用 `lxml.objectify` 解析该文件，通过 `getroot` 获取对该 XML 文件根节点的引用：

```
In [86]: from lxml import objectify

In [87]: path = "datasets/mta_perf/Performance_MNR.xml"

In [88]: with open(path) as f:
   ....:     parsed = objectify.parse(f)

In [89]: root = parsed.getroot()
```

`root.INDICATOR` 能返回一个用于生成各个 `<INDICATOR>`XML 元素的生成器。对于每条

记录，运行以下代码可以将包含标签名（如 YTD_ACTUAL）的字典转换为数据值（移除几个标签）：

```
data = []

skip_fields = ["PARENT_SEQ", "INDICATOR_SEQ",
               "DESIRED_CHANGE", "DECIMAL_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)
```

最后，将这个字典的列表转换为 DataFrame：

```
In [91]: perf = pd.DataFrame(data)

In [92]: perf.head()
Out[92]:
            AGENCY_NAME                       INDICATOR_NAME  \
0  Metro-North Railroad  On-Time Performance (West of Hudson)
1  Metro-North Railroad  On-Time Performance (West of Hudson)
2  Metro-North Railroad  On-Time Performance (West of Hudson)
3  Metro-North Railroad  On-Time Performance (West of Hudson)
4  Metro-North Railroad  On-Time Performance (West of Hudson)
                                                                DESCRIPTION  \
0  Percent of commuter trains that arrive at their destinations within 5 m...
1  Percent of commuter trains that arrive at their destinations within 5 m...
2  Percent of commuter trains that arrive at their destinations within 5 m...
3  Percent of commuter trains that arrive at their destinations within 5 m...
4  Percent of commuter trains that arrive at their destinations within 5 m...
   PERIOD_YEAR  PERIOD_MONTH           CATEGORY FREQUENCY INDICATOR_UNIT  \
0         2008             1  Service Indicators         M              %
1         2008             2  Service Indicators         M              %
2         2008             3  Service Indicators         M              %
3         2008             4  Service Indicators         M              %
4         2008             5  Service Indicators         M              %
  YTD_TARGET YTD_ACTUAL MONTHLY_TARGET MONTHLY_ACTUAL
0       95.0       96.9           95.0           96.9
1       95.0       96.0           95.0           95.0
2       95.0       96.3           95.0           96.9
3       95.0       96.8           95.0           98.3
4       95.0       96.6           95.0           95.8
```

以上整个过程可以用一行 pandas 的 `pandas.read_xml` 函数表达式完成：

```
In [93]: perf2 = pd.read_xml(path)

In [94]: perf2.head()
Out[94]:
   INDICATOR_SEQ  PARENT_SEQ          AGENCY_NAME  \
```

```
0          28445         NaN  Metro-North Railroad
1          28445         NaN  Metro-North Railroad
2          28445         NaN  Metro-North Railroad
3          28445         NaN  Metro-North Railroad
4          28445         NaN  Metro-North Railroad
                        INDICATOR_NAME  \
0  On-Time Performance (West of Hudson)
1  On-Time Performance (West of Hudson)
2  On-Time Performance (West of Hudson)
3  On-Time Performance (West of Hudson)
4  On-Time Performance (West of Hudson)
                                                                DESCRIPTION  \
0  Percent of commuter trains that arrive at their destinations within 5 m...
1  Percent of commuter trains that arrive at their destinations within 5 m...
2  Percent of commuter trains that arrive at their destinations within 5 m...
3  Percent of commuter trains that arrive at their destinations within 5 m...
4  Percent of commuter trains that arrive at their destinations within 5 m...
   PERIOD_YEAR  PERIOD_MONTH            CATEGORY FREQUENCY DESIRED_CHANGE  \
0         2008             1  Service Indicators         M              U
1         2008             2  Service Indicators         M              U
2         2008             3  Service Indicators         M              U
3         2008             4  Service Indicators         M              U
4         2008             5  Service Indicators         M              U
  INDICATOR_UNIT  DECIMAL_PLACES YTD_TARGET YTD_ACTUAL MONTHLY_TARGET  \
0              %               1      95.00      96.90          95.00
1              %               1      95.00      96.00          95.00
2              %               1      95.00      96.30          95.00
3              %               1      95.00      96.80          95.00
4              %               1      95.00      96.60          95.00
  MONTHLY_ACTUAL
0          96.90
1          95.00
2          96.90
3          98.30
4          95.80
```

对于更为复杂的 XML 文档，可以参考 `pandas.read_xml` 的文档。

6.2 二进制数据格式

使用 Python 内置的 `pickle` 模块，能便捷地将数据存储（或序列化）为二进制格式。所有 pandas 对象都有一个 `to_pickle` 方法，可以将数据以 pickle 格式保存到磁盘上：

```
In [95]: frame = pd.read_csv("examples/ex1.csv")

In [96]: frame
Out[96]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

```
In [97]: frame.to_pickle("examples/frame_pickle")
```

通常，pickle 文件只对 Python 是可读的。通过内置的 `pickle` 可以直接读取序列化数据，或者使用更为便捷的 `pandas.read_pickle`：

```
In [98]: pd.read_pickle("examples/frame_pickle")
Out[98]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

建议将 `pickle` 只用于短期存储格式。其原因是很难保证该格式永远是稳定的。现在序列化的对象可能无法被后续版本的库反序列化出来。虽然 pandas 尽力保证做到向后兼容，但是今后说不定还是得抛弃 pickle 格式。

pandas 内置支持其他的开源二进制数据格式，比如 HDF5、ORC 和 Apache Parquet。例如，如果你安装了 `pyarrow` 包（`conda install pyarrow`），就可以用 `pandas.read_parquet` 读取 Parquet 文件：

```
In [100]: fec = pd.read_parquet('datasets/fec/fec.parquet')
```

在 6.2.2 节中，我会给出若干 HDF5 示例。建议读者探索不同的文件格式，看看不同的文件格式在读取速度上的差异，以及是否适合自己的分析工作。

6.2.1 读取 Microsoft Excel 文件

使用 `pandas.ExcelFile` 类或 `pandas.read_excel` 函数，pandas 还支持读取存储于 Excel 2003（以及更高版本）文件中的表格型数据。在这些工具内部，它们分别使用附加组件包 `xlrd` 和 `openpyxl` 读取旧格式的 XLS 文件和新格式的 XLSX 文件。必须使用 pip 或 conda 安装这两个包：

```
conda install openpyxl xlrd
```

要使用 `pandas.ExcelFile`，传入 `xls` 或 `xlsx` 文件的路径：

```
In [101]: xlsx = pd.ExcelFile("examples/ex1.xlsx")
```

`xlsx` 对象可以用列表的形式展示文件中的工作表名称：

```
In [102]: xlsx.sheet_names
Out[102]: ['Sheet1']
```

利用 `parse` 将工作表中的数据读取到 DataFrame 中：

```
In [103]: xlsx.parse(sheet_name="Sheet1")
Out[103]:
   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
```

这张 Excel 表有一个索引列，可以用参数 index_col 指定索引列：

```
In [104]: xlsx.parse(sheet_name="Sheet1", index_col=0)
Out[104]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

如果要从文件读取多张工作表，这样一次性创建 pandas.ExcelFile 比较快，另外也可以将文件名传给 pandas.read_excel：

```
In [105]: frame = pd.read_excel("examples/ex1.xlsx", sheet_name="Sheet1")

In [106]: frame
Out[106]:
   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
```

如果要将 pandas 数据写入 Excel 格式，必须首先创建 ExcelWriter，然后用 pandas 对象的 to_excel 方法写入：

```
In [107]: writer = pd.ExcelWriter("examples/ex2.xlsx")

In [108]: frame.to_excel(writer, "Sheet1")

In [109]: writer.save()
```

如果不想使用 ExcelWriter，可以将文件路径直接传递给 to_excel：

```
In [110]: frame.to_excel("examples/ex2.xlsx")
```

6.2.2 使用 HDF5 格式

HDF5 是一种备受好评的文件格式，用于存储大规模科学数组数据。它以 C 标准库的形式存在，并带有许多其他语言的接口，包括 Java、Julia、MATLAB 和 Python 等。HDF5 中的“ HDF”指的是层次型数据格式（hierarchical data format）。每个 HDF5 文件能够存储多个数据集并支持元数据。与更简单的格式相比，HDF5 支持多种压缩模式的即时压缩，还能更高效地存储重复模式数据。对于那些大到无法放入内存的数据集，HDF5 是不错的选择，因为它可以高效读写大型数组中的一小部分。

要使用 pandas 读取 HDF5 文件，必须首先通过 conda 安装 pytables 包：

```
conda install pytables
```

注意，在 PyPI 中，PyTables 的包名是“tables”。所以，用 pip 安装的话，命令是 pip install tables。

虽然可以用 PyTables 或 h5py 库直接访问 HDF5 文件，但是 pandas 提供了更为高级的接口，可以简化存储 Series 和 DataFrame 对象。HDFStore 类可以像字典一样处理底层细节：

```
In [113]: frame = pd.DataFrame({"a": np.random.standard_normal(100)})

In [114]: store = pd.HDFStore("examples/mydata.h5")

In [115]: store["obj1"] = frame

In [116]: store["obj1_col"] = frame["a"]

In [117]: store
Out[117]:
<class 'pandas.io.pytables.HDFStore'>
File path: examples/mydata.h5
```

HDF5 文件中的对象可以通过相同的字典型 API 进行获取：

```
In [118]: store["obj1"]
Out[118]:
           a
0  -0.204708
1   0.478943
2  -0.519439
3  -0.555730
4   1.965781
..       ...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

HDFStore 支持两种存储模式："fixed" 和 "table"（"fixed" 是默认模式）。后者通常更慢，但支持使用特殊语法进行查询操作：

```
In [119]: store.put("obj2", frame, format="table")

In [120]: store.select("obj2", where=["index >= 10 and index <= 15"])
```

```
Out[120]:
          a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [121]: store.close()
```

put 是 store["obj2"] = frame 方法的显示版本，允许我们设置其他的选项，比如存储格式。

pandas.read_hdf 函数可以快捷使用这些工具：

```
In [122]: frame.to_hdf("examples/mydata.h5", "obj3", format="table")

In [123]: pd.read_hdf("examples/mydata.h5", "obj3", where=["index < 5"])
Out[123]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

可以按如下所示删除创建的 HDF5 文件：

```
In [124]: import os

In [125]: os.remove("examples/mydata.h5")
```

如果你要处理的数据位于远程服务器，比如 Amazon S3 或 HDFS，使用专门为分布式存储（比如 Apache Parquet）设计的二进制格式也许更加合适。

如果需要在本地处理海量数据，建议读者好好研究一下 PyTables 和 h5py，看看它们是否满足你的需求。由于大量数据分析问题都是 IO 密集型（而不是 CPU 密集型）问题，利用 HDF5 这样的工具能显著提升应用程序的效率。

HDF5 不是数据库。它最适合用作“一次写多次读”的数据集。虽然数据可以在任何时候被添加到文件中，但如果同时发生多个写操作，文件就可能会受到破坏。

6.3 与 Web API 交互

许多网站都有一些通过 JSON 或其他格式提供数据的公共 API。通过 Python 访问这些 API 的办法有不少。推荐使用 `requests` 包（*http://docs.python-requests.org*），可以通过 pip 或 conda 安装：

```
conda install requests
```

为了获取 GitHub 上关于 pandas 的 30 个最新问题，我们可以使用 `requests` 库发一个 HTTP `GET` 请求：

```
In [126]: import requests

In [127]: url = "https://api.github.com/repos/pandas-dev/pandas/issues"

In [128]: resp = requests.get(url)

In [129]: resp.raise_for_status()

In [130]: resp
Out[130]: <Response [200]>
```

每次使用 `requests.get` 之后，最好都调用 `raise_for_status` 检查 HTTP 的错误。

响应对象的 `json` 方法会返回一个 Python 对象，其中包含解析过的 JSON 数据字典或列表（取决于返回的 JSON 是什么）：

```
In [131]: data = resp.json()

In [132]: data[0]["title"]
Out[132]: 'REF: make copy keyword non-stateful'
```

获取到的结果是基于实时数据的，当你运行这段代码时，结果会不一样。

`data` 中的每个元素都是包含 GitHub 问题页上所有数据（不包含评论）的字典。我们可以直接将 `data` 传递给 `pandas.DataFrame`，并提取感兴趣的字段：

```
In [133]: issues = pd.DataFrame(data, columns=["number", "title",
   .....:                                      "labels", "state"])

In [134]: issues
Out[134]:
    number  \
0    48062
1    48061
2    48060
3    48059
4    48058
..     ...
```

```
25   48032
26   48030
27   48028
28   48027
29   48026
                                                                  title  \
0                                        REF: make copy keyword non-stateful
1                                                      STYLE: upgrade flake8
2   DOC: "Creating a Python environment" in "Creating a development environ...
3                                      REGR: Avoid overflow with groupby sum
4   REGR: fix reset_index (Index.insert) regression with custom Index subcl...
..                                                                        ...
25                  BUG: Union of multi index with EA types can lose EA dtype
26                                                     ENH: Add rolling.prod()
27                                     CLN: Refactor groupby's _make_wrapper
28                                        ENH: Support masks in groupby prod
29                                           DEP: Add pip to environment.yml
                                                                  labels  \
0                                                                     []
1   [{'id': 106935113, 'node_id': 'MDU6TGFiZWwxMDY5MzUxMTM=', 'url': 'https...
2   [{'id': 134699, 'node_id': 'MDU6TGFiZWwxMzQ2OTk=', 'url': 'https://api....
3   [{'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
4   [{'id': 32815646, 'node_id': 'MDU6TGFiZWwzMjgxNTY0Ng==', 'url': 'https:...
..                                                                        ...
25  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...
26  [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url': 'https://api.g...
27  [{'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
28  [{'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
29  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...
   state
0   open
1   open
2   open
3   open
4   open
..   ...
25  open
26  open
27  open
28  open
29  open
[30 rows x 4 columns]
```

花费一些精力，你就可以创建一些更高级的接口访问常见的 Web API，这些 API 返回 DataFrame 对象，以便更方便地进行分析。

6.4 与数据库交互

在业务场景下，大多数数据可能不是存储在文本或 Excel 文件中。基于 SQL 的关系型数据库（如 SQL Server、PostgreSQL 和 MySQL 等）使用非常广泛，其他数据库也很流行。数据库的选择通常取决于性能、数据完整性以及应用程序的可伸缩性需求。

对于将 SQL 查询结果加载到 DataFrame，pandas 有一些函数能简化这个流程。例如，我通过 Python 内置的 sqlite3 驱动创建一个 SQLite 数据库：

```
In [135]: import sqlite3

In [136]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
   .....: );"""

In [137]: con = sqlite3.connect("mydata.sqlite")

In [138]: con.execute(query)
Out[138]: <sqlite3.Cursor at 0x7fdfd73b69c0>

In [139]: con.commit()
```

插入几行数据：

```
In [140]: data = [("Atlanta", "Georgia", 1.25, 6),
   .....:         ("Tallahassee", "Florida", 2.6, 3),
   .....:         ("Sacramento", "California", 1.7, 5)]

In [141]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [142]: con.executemany(stmt, data)
Out[142]: <sqlite3.Cursor at 0x7fdfd73a00c0>

In [143]: con.commit()
```

从表中选取数据时，大部分 Python 的 SQL 驱动器都返回一个元组列表：

```
In [144]: cursor = con.execute("SELECT * FROM test")

In [145]: rows = cursor.fetchall()

In [146]: rows
Out[146]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

你可以将这个元组列表传给 DataFrame 构造器，但还需要包含于光标 `description` 属性中的列名。对于 SQLite3，光标的 `description` 属性只是提供了列名（其他属于 Python 数据库 API 说明的字段为 `None`），但其他的数据库驱动提供了更多列信息：

```
In [147]: cursor.description
Out[147]:
(('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))

In [148]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[148]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

这里需要相当多的数据规整，你肯定不想每查一次数据库就重复一次。SQLAlchemy项目（*https://www.sqlalchemy.org/*）是一个流行的Python SQL工具集，抽象地去除了SQL数据库之间的许多常见差异。pandas有一个`read_sql`函数，可以让你从通用的SQLAlchemy连接轻松读取数据。用conda安装SQLAlchemy，如下所示：

```
conda install sqlalchemy
```

现在，我们用SQLAlchemy连接同样的SQLite数据库，并从之前创建的表读取数据：

```
In [149]: import sqlalchemy as sqla

In [150]: db = sqla.create_engine("sqlite:///mydata.sqlite")

In [151]: pd.read_sql("SELECT * FROM test", db)
Out[151]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

6.5 总结

访问数据通常是数据分析流程的第一步。在本章中，我们学习了一些有用的工具，可以帮助你入门。在接下来的章节中，我们将深入研究数据规整、数据可视化、时间序列分析等主题。