# PYTHON

## DATA LOADING, STORAGE, AND FILE FORMATS

- Reading and Writing Data in Text Format
- Binary Data Formats
- Interacting with Web APIs
- Interacting with Databases

## DATA CLEANING AND PREPARATION

- Handling Missing Data
- Data Transformation
- String Manipulation

# I

# DATA LOADING, STORAGE, AND FILE FORMATS

1

# Reading and Writing Data in Text Format

READING DATA

READING DATA

LOADING DATA

WRITING DATA

1

READING DATA

# Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. Table 6-1 summarizes some of them, though read_csv and read_table are likely the ones you'll use the most.

*Table 6-1. Parsing functions in pandas*

| Function | Description |
|---|---|
| `read_csv` | Load delimited data from a file, URL, or file-like object; use comma as default delimiter |
| `read_table` | Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter |
| `read_fwf` | Read data in fixed-width column format (i.e., no delimiters) |
| `read_clipboard` | Version of `read_table` that reads data from the clipboard; useful for converting tables from web pages |
| `read_excel` | Read tabular data from an Excel XLS or XLSX file |
| `read_hdf` | Read HDF5 files written by pandas |
| `read_html` | Read all tables found in the given HTML document |
| `read_json` | Read data from a JSON (JavaScript Object Notation) string representation |
| `read_msgpack` | Read pandas data encoded using the MessagePack binary format |
| `read_pickle` | Read an arbitrary object stored in Python pickle format |
| `read_sas` | Read a SAS dataset stored in one of the SAS system's custom storage formats |
| `read_sql` | Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame |
| `read_stata` | Read a dataset from Stata file format |
| `read_feather` | Read the Feather binary file format |

READING DATA

1

# Reading and Writing Data in Text Format

Handling dates and other custom types can require extra effort. Let's start with a small comma-separated (CSV) text file:

```
In [16]: !D:\D2\1. CMC\OneDrive\4. MCI\2. Program\4.Python\pydata-book-2nd-edition\pydata-book-2nd-edition\examples\ex1.csv
```

```
In [14]: path = "D:D2/1. CMC/OneDrive/4. MCI/2. Program/4.Python\pydata-book-2nd-edition/pydata-book-2nd-edition/examples/"
         file_name = "ex1.csv"
         df = pd.read_csv(path + file_name)
         df
```

Out[14]:

| | 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 |

We could also have used read_table and specified the delimiter:

```
In [4]: pd.read_table(path + file_name, sep=',')
```

Out[4]:

| | a | b | c | d | message |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | hello |
| 1 | 5 | 6 | 7 | 8 | world |
| 2 | 9 | 10 | 11 | 12 | foo |

READING DATA

1

# Reading and Writing Data in Text Format

```
In [5]: file_name2 = "ex2.csv"
```

```
In [12]: from IPython.core.interactiveshell import InteractiveShell
         InteractiveShell.ast_node_interactivity = "all"

         pd.read_csv(path + file_name2, header=None)
         pd.read_csv(path + file_name2, names=['a', 'b', 'c', 'd', 'message'])
```

Remove header

Add column names

Out[12]:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | 1 | 2 | 3 | 4 | hello |
| **1** | 5 | 6 | 7 | 8 | world |
| **2** | 9 | 10 | 11 | 12 | foo |

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 |

```
In [19]: names = ['a', 'b', 'c', 'd', 'message']
         pd.read_csv(path + file_name2, names=names, index_col='message')
```

Out[19]:

| | a | b | c | d |
|---|---|---|---|---|
| **message** | | | | |
| **hello** | 1 | 2 | 3 | 4 |
| **world** | 5 | 6 | 7 | 8 |
| **foo** | 9 | 10 | 11 | 12 |

Name for column index

# 1 Reading and Writing Data in Text Format

READING DATA

In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

```
parsed = pd.read_csv(path + 'csv_mindex.csv',
                     index_col=['key1', 'key2'])
parsed
```

Out[22]:

| key1 | key2 | value1 | value2 |
|---|---|---|---|
| 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 |

# 1 Reading and Writing Data in Text Format

READING DATA

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. Consider a text file that looks like this:

```
In [23]: list(open(path + 'ex3.txt'))

Out[23]: ['            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']
```

In these cases, you can pass a regular expression as a delimiter for read_table. This can be expressed by the regular expression \s+

```
In [26]: # result = pd.read_table(path + 'ex3.txt', sep='\s+')
         result = pd.read_table(path + 'ex3.txt', sep = '\s+')
         result

Out[26]:
```

| | 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 |

1

READING DATA

# Reading and Writing Data in Text Format

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur

In [32]:
```
pd.read_csv(path + 'ex4.csv', skiprows=[0, 2, 3])
```

Out[32]:

| | a | b | c | d | message |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | hello |
| 1 | 5 | 6 | 7 | 8 | world |
| 2 | 9 | 10 | 11 | 12 | foo |

# 1

# Reading and Writing Data in Text Format

READING DATA

Handling missing values is an important and frequently nuanced part of the file pars-
ing process.

Missing data is usually either not present (empty string) or marked by some *sentinel* value.

By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:

```
In [20]: result = pd.read_csv(path + "ex5.csv")
         result
         pd.isnull(result)
```

Out[20]:

| | 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 |

```
In [22]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}
         pd.read_csv(path + 'ex5.csv', na_values=sentinels)
```

Out[22]:

| | 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 |

```
In [21]: result = pd.read_csv(path + 'ex5.csv', na_values=['NULL'])
         result
```

Out[21]:

| | 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 |

# 1 Reading and Writing Data in Text Format

*Table 6-2. Some read_csv/read_table function arguments*

| Argument | Description |
|---|---|
| path | String indicating filesystem location, URL, or file-like object |
| sep or delimiter | Character sequence or regular expression to use to split fields in each row |
| header | Row number to use as column names; defaults to 0 (first row), but should be None if there is no header row |
| index_col | Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index |
| names | List of column names for result, combine with header=None |
| comment | Character(s) to split comments off the end of lines. |
| parse_dates | Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns). |
| keep_date_col | If joining columns to parse date, keep the joined columns; False by default. |
| converters | Dict containing column number of name mapping to functions (e.g., {'foo': f} would apply the function f to all values in the 'foo' column). |

*Table 6-2. Some read_csv/read_table function arguments*

| Argument | Description |
|---|---|
| dayfirst | When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); False by default. |
| date_parser | Function to use to parse dates. |
| nrows | Number of rows to read from beginning of file. |
| iterator | Return a TextParser object for reading file piecemeal. |
| chunksize | For iteration, size of file chunks. |
| skip_footer | Number of lines to ignore at end of file. |
| verbose | Print various parser output information, like the number of missing values placed in non-numeric columns. |
| encoding | Text encoding for Unicode (e.g., 'utf-8' for UTF-8 encoded text). |
| squeeze | If the parsed data only contains one column, return a Series. |
| thousands | Separator for thousands (e.g., ',' or '.'). |

1

READING DATA

# Reading and Writing Data in Text Format

## Reading Text Files in Pieces

In [23]:
```
pd.options.display.max_rows = 10
```

In [24]:
```
result = pd.read_csv(path + 'ex6.csv')
result
```

Out[24]:

| | 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 × 5 columns

If you want to only read a small number of rows (avoiding reading the entire file), specify that with nrows:

In [26]:
```
pd.read_csv(path + 'ex6.csv', nrows=5)
```

Out[26]:

| | 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 |

1

READING DATA

# Reading and Writing Data in Text Format

## Reading Text Files in Pieces

To read a file in pieces, specify a chunksize as a number of rows:

```
In [30]: chunker = pd.read_csv(path + 'ex6.csv', chunksize=1000)
         chunker

Out[30]: <pandas.io.parsers.TextFileReader at 0xc416b50>

In [31]: chunker = pd.read_csv(path + 'ex6.csv', chunksize=1000)

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

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

1

READING DATA

# Reading and Writing Data in Text Format

## Reading Text Files in Pieces

The TextParser object returned by read_csv allows you to iterate over the parts of the file according to the chunksize.

For example, we can iterate over ex6.csv, aggregating the value counts in the 'key' column like so:

```
In [29]: tot[:10]

Out[29]: 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
```

1

WRITTING DATA

# Reading and Writing Data in Text Format

## Writing Data to Text Format¶

Data can also be exported to a delimited format.

```
In [32]: data = pd.read_csv(path + 'ex5.csv')
         data
```

Out[32]:

| | 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 |

1

WRITTING DATA

# Reading and Writing Data in Text Format

### Writing Data to Text Format¶

Other delimiters can be used(writing to sys.stdout so it prints the text result to the console). Also can denote NULL for n/a

```
In [33]: data.to_csv(path + '/out.csv')
```

```
In [34]: import sys
         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
```

```
In [35]: 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
```

1

WRITTING DATA

# Reading and Writing Data in Text Format

## Writing Data to Text Format¶

```
In [36]: 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
```

```
In [37]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
```

```
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

```
In [38]: dates = pd.date_range('1/1/2000', periods=7)
         ts = pd.Series(np.arange(7), index=dates)
         ts.to_csv(path + '/tseries.csv')
```

1

READING DATA

# Reading and Writing Data in Text Format

## Working with Delimited Formats

```
In [ ]: import csv
        f = open(path + 'ex7.csv')

        reader = csv.reader(f)
```

```
In [ ]: for line in reader:
            print(line)
```

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

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

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

# Reading and Writing Data in Text Format

1

READING DATA

## JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications.

It is a much more free-form data format than a tabular text form like CSV

```
In [ ]: obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

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

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

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

```
In [ ]: data = pd.read_json(path + '/example.json')
data
```

```
In [ ]: print(data.to_json())
print(data.to_json(orient='records'))
```

1

# Reading and Writing Data in Text Format

## XML and HTML: Web Scraping

READING DATA

conda install lxml pip install beautifulsoup4 html5lib

In [39]:
```
tables = pd.read_html(path + '/fdic_failed_bank_list.html')
len(tables)
failures = tables[0]
failures.head()
```

Out[39]:

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

# Reading and Writing Data in Text Format

## XML and HTML: Web Scraping

```
In [40]: close_timestamps = pd.to_datetime(failures['Closing Date'])
         close_timestamps.dt.year.value_counts()

Out[40]: 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
```

2

# Binary Data Formats

READING DATA

In [46]:
```
frame = pd.read_csv(path + '/ex1.csv')
frame
frame.to_pickle(path + '/frame_pickle')
```

Out[46]:

| | 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 |

In [44]:
```
pd.read_pickle(path + '/frame_pickle')
```

Out[44]:

| | 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 |

2

# Binary Data Formats

## Using HDF5 Format

```
In [ ]: frame = pd.DataFrame({'a': np.random.randn(100)})
        store = pd.HDFStore('mydata.h5')
        store['obj1'] = frame
        store['obj1_col'] = frame['a']
        store

In [ ]: store['obj1']

In [ ]: store.put('obj2', frame, format='table')
        store.select('obj2', where=['index >= 10 and index <= 15'])
        store.close()

In [ ]: frame.to_hdf('mydata.h5', 'obj3', format='table')
        pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])

In [ ]: os.remove('mydata.h5')
```

2

READING DATA

# Binary Data Formats

## Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the ExcelFile class or pandas.read_excel function.

Internally these tools use the add-on packages xlrd and openpyxl to read XLS and XLSX files, respec-
tively.

You may need to install these manually with pip or conda

```
In [75]: xlsx = pd.ExcelFile(path + '/ex1.xlsx')

In [76]: pd.read_excel(xlsx, 'Sheet1')
Out[76]:
```

| | 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 |

```
In [77]: frame = pd.read_excel(path + '/ex1.xlsx', 'Sheet1')
         frame
Out[77]:
```

| | 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 |

```
In [126]: writer = pd.ExcelWriter(path + '/ex2.xlsx')
          frame.to_excel(writer, 'Sheet1')
          writer.save()

In [131]: frame.to_excel(path + 'ex2.xlsx')
```

# Interacting with Web APIs

```
In [72]: import requests
         url = 'https://api.github.com/repos/pandas-dev/pandas/issues'
         # url = 'https://vnexpress.net'
         resp = requests.get(url)
         resp

Out[72]: <Response [200]>

In [73]: data = resp.json()
         data[0]['title']
         # data[0]

Out[73]: 'BUG:Timezone lost when assigning Datetime via DataFrame.at'

In [74]: issues = pd.DataFrame(data, columns=['number', 'title',
                                              'labels', 'state'])
         issues

Out[74]:
```

| | number | title | labels | state |
|---|---|---|---|---|
| 0 | 33544 | BUG:Timezone lost when assigning Datetime via ... | [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ=... | open |
| 1 | 33543 | Preserving boolean dtype in Series.any/all fun... | [] | open |
| 2 | 33542 | ENH: Use self values in Series groupby | [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg=... | open |
| 3 | 33541 | DownSampling Time Series Data using pandas | [{'id': 1954720290, 'node_id': 'MDU6TGFiZWwxOT... | open |

# 4 Interacting with Databases

LOADING DATA

In a business setting, most data may not be stored in text or Excel files. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative databases have become quite popular.

The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application

```
In [33]: import sqlite3
         query = """
         CREATE TABLE test
         (a VARCHAR(20), b VARCHAR(20),
          c REAL,        d INTEGER
         );"""
         con = sqlite3.connect('mydata.sqlite')
         con.execute(query)
         con.commit()

Out[33]: <sqlite3.Cursor at 0xd9fe20>
```

# Interacting with Databases

4

LOADING DATA

Connect to SQL

```
In [50]: data = [('Tiep', 'Cuong', 1.25, 6),
                 ('Nam', 'Vuong', 2.6, 3),
                 ('Cuong', 'Lan', 1.7, 5)]
         stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
         con.executemany(stmt, data)
         con.commit()

Out[50]: <sqlite3.Cursor at 0xe597a0>
```

```
In [51]: cursor = con.execute('select * from test')
         rows = cursor.fetchall()
         rows

Out[51]: [('Atlanta', 'Georgia', 1.25, 6),
          ('Tallahassee', 'Florida', 2.6, 3),
          ('Sacramento', 'California', 1.7, 5),
          ('Tiep', 'Cuong', 1.25, 6),
          ('Nam', 'Vuong', 2.6, 3),
          ('Cuong', 'Lan', 1.7, 5),
          ('Tiep', 'Cuong', 1.25, 6),
          ('Nam', 'Vuong', 2.6, 3),
          ('Cuong', 'Lan', 1.7, 5)]
```

# Interacting with Databases

4

LOADING DATA

Connect to SQL

In [55]:
```
cursor.description
pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
```

Out[55]:
```
(('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))
```

Out[55]:

| | a | b | c | d |
|---|---|---|---|---|
| **0** | Atlanta | Georgia | 1.25 | 6 |
| **1** | Tallahassee | Florida | 2.60 | 3 |
| **2** | Sacramento | California | 1.70 | 5 |
| **3** | Tiep | Cuong | 1.25 | 6 |
| **4** | Nam | Vuong | 2.60 | 3 |
| **5** | Cuong | Lan | 1.70 | 5 |
| **6** | Tiep | Cuong | 1.25 | 6 |
| **7** | Nam | Vuong | 2.60 | 3 |

4

# Interacting with Databases

LOADING DATA

Connect to SQL

```
In [56]: import sqlalchemy as sqla
         db = sqla.create_engine('sqlite:///mydata.sqlite')
         pd.read_sql('select * from test', db)
```

Out[56]:

| | a | b | c | d |
|---|---|---|---|---|
| 0 | Atlanta | Georgia | 1.25 | 6 |
| 1 | Tallahassee | Florida | 2.60 | 3 |
| 2 | Sacramento | California | 1.70 | 5 |
| 3 | Tiep | Cuong | 1.25 | 6 |
| 4 | Nam | Vuong | 2.60 | 3 |
| 5 | Cuong | Lan | 1.70 | 5 |
| 6 | Tiep | Cuong | 1.25 | 6 |
| 7 | Nam | Vuong | 2.60 | 3 |
| 8 | Cuong | Lan | 1.70 | 5 |

II

# DATA CLEANING AND PREPARATION

# Data Cleaning and Preparation

- During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging.
- Such tasks are often reported to take up **80%** or more of an analyst's time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task.
- Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, like Python, Perl, R, or Java, or Unix text-processing tools like sed or awk.
- Fortunately, pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form

# Handling Missing Data

*sentinel value* can be easily detected with Pandas

```
In [2]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
        string_data
        string_data.isnull()

Out[2]: 0    False
        1    False
        2     True
        3    False
        dtype: bool

In [3]: string_data[0] = None
        string_data.isnull()

Out[3]: 0     True
        1    False
        2     True
        3    False
        dtype: bool
```

*Table 7-1. NA handling methods*

| Argument | Description |
|---|---|
| `dropna` | Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate. |
| `fillna` | Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`. |
| `isnull` | Return boolean values indicating which values are missing/NA. |
| `notnull` | Negation of `isnull`. |

1

DATA CLEANING AND PREPARATION

# Handling Missing Data

## Filtering Out Missing Data

There are a few ways to filter out missing data.

While you always have the option to do it by hand using pandas.isnull and boolean indexing, the dropna can be helpful.

On a Series, it returns the Series with only the non-null data and index values:

```
In [4]: from numpy import nan as NA
        data = pd.Series([1, NA, 3.5, NA, 7])
        data.dropna()

Out[4]: 0    1.0
        2    3.5
        4    7.0
        dtype: float64

In [5]: data[data.notnull()]

Out[5]: 0    1.0
        2    3.5
        4    7.0
        dtype: float64
```

# Handling Missing Data

1

DATA CLEANING AND PREPARATION

How to handle missing data?

In [6]:
```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                     [NA, NA, NA], [NA, 6.5, 3.]])
cleaned = data.dropna()
data
cleaned
```

Out[6]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 6.5 | 3.0 |

In [7]:
```
data.dropna(how='all')
```

Out[7]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 6.5 | 3.0 |
| 1 | 1.0 | NaN | NaN |
| 3 | NaN | 6.5 | 3.0 |

1

# Handling Missing Data

DATA CLEANING AND PREPARATION

In [7]:
```
data.dropna(how='all')
```

Out[7]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1.0 | 6.5 | 3.0 |
| 1 | 1.0 | NaN | NaN |
| 3 | NaN | 6.5 | 3.0 |

In [8]:
```
data[4] = NA
data
data.dropna(axis=1, how='all')
```

Out[8]:

| | 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 [9]:
```
df = pd.DataFrame(np.random.randn(7, 3))
df.iloc[:4, 1] = NA
df.iloc[:2, 2] = NA
df
df.dropna()
df.dropna(thresh=2)
```

Out[9]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 2 | 0.092908 | NaN | 0.769023 |
| 3 | 1.246435 | NaN | -1.296221 |
| 4 | 0.274992 | 0.228913 | 1.352917 |
| 5 | 0.886429 | -2.001637 | -0.371843 |
| 6 | 1.669025 | -0.438570 | -0.539741 |

# Handling Missing Data

## Filling In Missing Data

In [10]:
```
df.fillna(0)
```

Out[10]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | -0.204708 | 0.000000 | 0.000000 |
| 1 | -0.555730 | 0.000000 | 0.000000 |
| 2 | 0.092908 | 0.000000 | 0.769023 |
| 3 | 1.246435 | 0.000000 | -1.296221 |
| 4 | 0.274992 | 0.228913 | 1.352917 |
| 5 | 0.886429 | -2.001637 | -0.371843 |
| 6 | 1.669025 | -0.438570 | -0.539741 |

In [11]:
```
df.fillna({1: 0.5, 2: 0})
```

Out[11]:

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | -0.204708 | 0.500000 | 0.000000 |
| 1 | -0.555730 | 0.500000 | 0.000000 |
| 2 | 0.092908 | 0.500000 | 0.769023 |
| 3 | 1.246435 | 0.500000 | -1.296221 |
| 4 | 0.274992 | 0.228913 | 1.352917 |
| 5 | 0.886429 | -2.001637 | -0.371843 |
| 6 | 1.669025 | -0.438570 | -0.539741 |

1

# Handling Missing Data

DATA CLEANING AND PREPARATION

In [12]:
```
_ = df.fillna(0, inplace=True)
df
```

Out[12]:

| | 0 | 1 | 2 |
|---|---|---|---|
| **0** | -0.204708 | 0.000000 | 0.000000 |
| **1** | -0.555730 | 0.000000 | 0.000000 |
| **2** | 0.092908 | 0.000000 | 0.769023 |
| **3** | 1.246435 | 0.000000 | -1.296221 |
| **4** | 0.274992 | 0.228913 | 1.352917 |
| **5** | 0.886429 | -2.001637 | -0.371843 |
| **6** | 1.669025 | -0.438570 | -0.539741 |

In [13]:
```
df = pd.DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA
df.iloc[4:, 2] = NA
df
df.fillna(method='ffill')
df.fillna(method='ffill', limit=2)
```

Out[13]:

| | 0 | 1 | 2 |
|---|---|---|---|
| **0** | 0.476985 | 3.248944 | -1.021228 |
| **1** | -0.577087 | 0.124121 | 0.302614 |
| **2** | 0.523772 | 0.124121 | 1.343810 |
| **3** | -0.713544 | 0.124121 | -2.370232 |
| **4** | -1.860761 | NaN | -2.370232 |
| **5** | -1.265934 | NaN | -2.370232 |

# 1 Handling Missing Data

DATA CLEANING AND PREPARATION

```
In [14]: data = pd.Series([1., NA, 3.5, NA, 7])
         data.fillna(data.mean())

Out[14]: 0    1.000000
         1    3.833333
         2    3.500000
         3    3.833333
         4    7.000000
         dtype: float64
```

# Data Transformation

## Removing Duplicates

```
In [15]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                              'k2': [1, 1, 2, 3, 3, 4, 4]})
         data

Out[15]:
```

| | k1 | k2 |
|---|---|---|
| 0 | one | 1 |
| 1 | two | 1 |
| 2 | one | 2 |
| 3 | two | 3 |
| 4 | one | 3 |
| 5 | two | 4 |
| 6 | two | 4 |

2

# Data Transformation

## Removing Duplicates

```
In [16]: data.duplicated()

Out[16]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         dtype: bool
```

```
In [17]: data.drop_duplicates()

Out[17]:
```

| | k1 | k2 |
|---|---|---|
| **0** | one | 1 |
| **1** | two | 1 |
| **2** | one | 2 |
| **3** | two | 3 |
| **4** | one | 3 |
| **5** | two | 4 |

# 2 Data Transformation

DATA CLEANING AND PREPARATION

## Removing Duplicates

In [18]:
```
data['v1'] = range(7)
data.drop_duplicates(['k1'])
```

Out[18]:

| | k1 | k2 | v1 |
|---|---|---|---|
| 0 | one | 1 | 0 |
| 1 | two | 1 | 1 |

In [19]:
```
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[19]:

| | 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 |

# Data Transformation

2

DATA CLEANING AND PREPARATION

## Transforming Data Using a Function or Mapping

```
In [20]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                                       'Pastrami', 'corned beef', 'Bacon',
                                       'pastrami', 'honey ham', 'nova lox'],
                              'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
         data
```

Out[20]:

| | 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 |

# Data Transformation

2

DATA CLEANING AND PREPARATION

**Transforming Data Using a Function or Mapping**

```
In [21]: meat_to_animal = {
           'bacon': 'pig',
           'pulled pork': 'pig',
           'pastrami': 'cow',
           'corned beef': 'cow',
           'honey ham': 'pig',
           'nova lox': 'salmon'
         }
```

```
In [22]: lowercased = data['food'].str.lower()
         lowercased
         data['animal'] = lowercased.map(meat_to_animal)
         data
```

```
In [23]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

# Data Transformation

## Replacing Values

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

```
In [ ]: data.replace(-999, np.nan)
```

```
In [ ]: data.replace([-999, -1000], np.nan)
```

```
In [ ]: data.replace([-999, -1000], [np.nan, 0])
```

```
In [ ]: data.replace({-999: np.nan, -1000: 0})
```

# Data Transformation

## Renaming Axis Indexes

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

```
In [ ]: transform = lambda x: x[:4].upper()
        data.index.map(transform)
```

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

```
In [ ]: data.rename(index=str.title, columns=str.upper)
```

```
In [ ]: data.rename(index={'OHIO': 'INDIANA'},
                    columns={'three': 'peekaboo'})
```

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

2

# Data Transformation

DATA CLEANING AND PREPARATION

Applicable for classification & catergorization

### Discretization and Binning

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

```
In [ ]: bins = [18, 25, 35, 60, 100]
        cats = pd.cut(ages, bins)
        cats
```

```
In [ ]: cats.codes
        cats.categories
        pd.value_counts(cats)
```

```
In [ ]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

```
In [ ]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
        pd.cut(ages, bins, labels=group_names)
```

```
In [ ]: data = np.random.rand(20)
        pd.cut(data, 4, precision=2)
```

```
In [ ]: data = np.random.randn(1000)  # Normally distributed
        cats = pd.qcut(data, 4)  # Cut into quartiles
        cats
        pd.value_counts(cats)
```

```
In [ ]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

# Data Transformation

## Detecting and Filtering Outliers

```
In [ ]: data = pd.DataFrame(np.random.randn(1000, 4))
        data.describe()
```

```
In [ ]: col = data[2]
        col[np.abs(col) > 3]
```

```
In [ ]: data[(np.abs(data) > 3).any(1)]
```

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

```
In [ ]: np.sign(data).head()
```

# Data Transformation

## Permutation and Random Sampling

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

```
In [ ]: df
        df.take(sampler)
```

```
In [ ]: df.sample(n=3)
```

```
In [ ]: choices = pd.Series([5, 7, -1, 6, 4])
        draws = choices.sample(n=10, replace=True)
        draws
```

# String Manipulation

## String Object Methods

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

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

In [ ]:
```
first, second, third = pieces
first + '::' + second + '::' + third
```

In [ ]:
```
'::'.join(pieces)
```

In [ ]:
```
'guido' in val
val.index(',')
val.find(':')
```

In [ ]:
```
val.index(':')
```

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

In [ ]:
```
val.replace(',', '::')
val.replace(',', '')
```

*Table 7-3. Python built-in string methods*

| Argument | Description |
|---|---|
| `count` | Return the number of non-overlapping occurrences of substring in the string. |
| `endswith` | Returns `True` if string ends with suffix. |
| `startswith` | Returns `True` if string starts with prefix. |
| `join` | Use string as delimiter for concatenating a sequence of other strings. |
| `index` | Return position of first character in substring if found in the string; raises `ValueError` if not found. |
| `find` | Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found. |
| `rfind` | Return position of first character of *last* occurrence of substring in the string; returns –1 if not found. |
| `replace` | Replace occurrences of string with another string. |
| `strip`, `rstrip`, `lstrip` | Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element. |
| `split` | Break string into list of substrings using passed delimiter. |
| `lower` | Convert alphabet characters to lowercase. |
| `upper` | Convert alphabet characters to uppercase. |
| `casefold` | Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form. |
| `ljust`, `rjust` | Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width. |

3

DATA CLEANING AND PREPARATION

# String Manipulation

## Regular Expressions

In [ ]:
```
import re
text = "foo    bar\t baz  \tqux"
re.split('\s+', text)
```

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

In [ ]:
```
regex.findall(text)
```

In [ ]:
```
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.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

DATA CLEANING AND PREPARATION

3

# String Manipulation

```
In [ ]: regex.findall(text)

In [ ]: m = regex.search(text)
        m
        text[m.start():m.end()]

In [ ]: print(regex.match(text))

In [ ]: print(regex.sub('REDACTED', text))

In [ ]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
        regex = re.compile(pattern, flags=re.IGNORECASE)

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

In [ ]: regex.findall(text)

In [ ]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

# String Manipulation

*Table 7-4. Regular expression methods*

| Argument | Description |
|---|---|
| `findall` | Return all non-overlapping matching patterns in a string as a list |
| `finditer` | Like `findall`, but returns an iterator |
| `match` | Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None |
| `search` | Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning |
| `split` | Break string into pieces at each occurrence of pattern |
| `sub, subn` | Replace all (`sub`) or first `n` occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string |

# String Manipulation

## Vectorized String Functions in pandas

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

In [ ]:
```
data.str.contains('gmail')
```

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

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

In [ ]:
```
matches.str.get(1)
matches.str[0]
```

In [ ]:
```
data.str[:5]
```

In [ ]:
```
pd.options.display.max_rows = PREVIOUS_MAX_ROWS
```

# THANKS FOR LISTENING!!!