PRACTICE SESSION 5

DATA LOADING, STORAGE, AND FILE FORMATS

Reading and Writing Data in Text Format

> pandas features a number of functions for reading tabular data as a DataFrame object.

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

Reading and Writing Data in Text Format

Function	Description
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 and Writing Data in Text Format

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

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

Reading and Writing Data in Text Format

> A file will not always have a header row

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

> To read this file, you have a couple of options

– You can allow pandas to assign default column names

– You can specify names yourself

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo
```

Reading and Writing Data in Text Format

> To read this file, you have a couple of options

– You can allow pandas to assign default column names

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo
```

– You can specify names yourself

```
In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Reading and Writing Data in Text Format

> Suppose you wanted the message column to be the index of the returned DataFrame.

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

Reading and Writing Data in Text Format

> If you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

```
In [17]: !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
```

Reading and Writing Data in Text Format

> If you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

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

In [19]: parsed
Out[19]:
           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
```

Reading and Writing Data in Text Format

> In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields:

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            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']
```

Reading and Writing Data in Text Format

> In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            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
```

`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
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip
`na_values`	na_values Sequence of values to replace with NA.
`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).

`read_csv`/`read_table` function arguments

Argument	Description
`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).
`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 '.').

`read_csv`/`read_table` function arguments

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

> Example: You can skip the first, third, and fourth rows of a file with **`skiprows`**:

```
In [23]: !cat examples/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 [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

`read_csv`/`read_table` function arguments

> Handling missing values is an important and frequently nuanced part of the file parsing process.

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

`read_csv`/`read_table` function arguments

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

```
In [27]: result
Out[27]:
  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 [28]: pd.isnull(result)
Out[28]:
   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
```

`read_csv/read_table` function arguments

> The **`na_values`** option can take either a list or set of strings to consider missing values:

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  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
```

`read_csv/read_table` function arguments

> Different NA sentinels can be specified for each column in a dict:

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  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
```

Reading Text Files in Pieces

> When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        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
```

Make the pandas display settings more compact

```
In [33]: pd.options.display.max_rows = 10

In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           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]
```

Reading Text Files in Pieces

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

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

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

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

```
In [40]: tot[:10]
Out[40]:
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
```

Writing Data to Text Format

> Data can also be exported to a delimited format

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  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
```

Writing Data to Text Format

> Data can also be exported to a delimited format

> Using DataFrame's to_csv method, we can write the data out to a comma-separated file:

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  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 [43]: data.to_csv('examples/out.csv')

In [44]: !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
```

Writing Data to Text Format

> Other delimiters can be used

```
In [45]: import sys

In [46]: 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
```

Writing Data to Text Format

> Missing values appear as empty strings in the output.

> You might want to denote them by some other sentinel value

```
In [47]: 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
```

Writing Data to Text Format

> The row and column labels can be disabled

```
In [48]: 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
```

Writing Data to Text Format

> You can also write only a subset of the columns, and in an order of your choosing

```
In [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Writing Data to Text Format

> Series also has a **`to_csv`** method

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
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
```

Working with Delimited Formats

> It's possible to load most forms of tabular data from disk using functions like **`pandas.read_table`**.

> In some cases, however, some manual processing may be necessary.

> It's not uncommon to receive a file with one or more malformed lines that trip up **`read_table`**.

Working with Delimited Formats

> To illustrate the basic tools, consider a small CSV file:

```
In [54]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

> Iterating through the reader like a file yields tuples of values with any quote characters removed:

```
In [56]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

Working with Delimited Formats

> From there, it's up to you to do the wrangling necessary to put the data in the form that you need it

```
In [57]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

> Split the lines into the header line and the data lines

```
In [58]: header, values = lines[0], lines[1:]
```

Working with Delimited Formats

> Create a dictionary of data columns using a dictionary comprehension and the expression zip(*values), which transposes rows to columns:

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [60]: data_dict
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV Dialect

> To define a new format with a different delimiter, string quoting convention, or line terminator, we define a simple subclass of **`csv.Dialect`**:

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

reader = csv.reader(f, dialect=my_dialect)
```

CSV Dialect

> We can also give individual CSV dialect parameters as keywords to **`csv.reader`** without having to define a subclass:

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

CSV dialect options

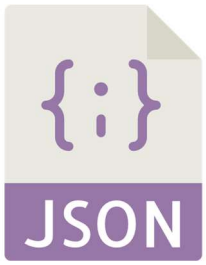
Argument	Description
`delimiter`	One-character string to separate fields; defaults to ','.
`lineterminator`	Line terminator for writing; defaults to '\r\n'. Reader ignores this and recognizes cross-platform line terminators.
`quotechar`	Quote character for fields with special characters (like a delimiter); default is '"'.
`quoting`	Quoting convention. Options include **`csv.QUOTE_ALL`** (quote all fields), **`csv.QUOTE_MINIMAL`** (only fields with special characters like the delimiter), **`csv.QUOTE_NONNUMERIC`**, and **`csv.QUOTE_NONE`** (no quoting). See Python's documentation for full details. Defaults to **`QUOTE_MINIMAL`**.
`skipinitialspace`	Ignore whitespace after each delimiter; default is False.
`doublequote`	How to handle quoting character inside a field; if True, it is doubled (see online documentation for full detail and behavior).
`escapechar`	String to escape the delimiter if quoting is set to **`csv.QUOTE_NONE`**; disabled by default

CSV dialect options

> To *write* delimited files manually, you can use **`csv.writer`**.

> It accepts an open, writable file object and the same dialect and format options as **`csv.reader`**:

```
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'))
```



JSON

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

```
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"]}]
}
"""
```

JSON Data

> JSON is very nearly valid Python code with the exception of its null value null and some other nuances (such as disallowing trailing commas at the end of lists).

> The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls.

> All of the keys in an object must be strings.

JSON Data

> There are several Python libraries for reading and writing JSON data.

> We will use **`json`**

  – It is built into the *Python standard library*.

> To convert a JSON string to Python form, use **`json.loads`**:

```
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

JSON Data

> **`json.dumps`** converts a Python object back to JSON:

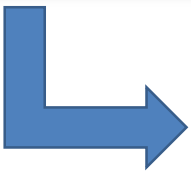
```
In [65]: asjson = json.dumps(result)
```

JSON Data

> The **`pandas.read_json`** can automatically convert JSON datasets in specific arrangements into a Series or DataFrame.

> The default options for pandas.read_json assume that each object in the JSON array is a row in the table:

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```



```
In [69]: data = pd.read_json('examples/example.json')

In [70]: data
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

JSON Data

> If you need to export data from pandas to JSON, one way is to use the **`to_json`** methods on Series and DataFrame:

```
In [71]: 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}}

In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

XML AND HTML: WEB SCRAPING

XML and HTML: Web Scraping

> Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats.

> Examples include lxml, Beautiful Soup, and html5lib.

> While lxml is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

XML and HTML: Web Scraping

> pandas has a built-in function, **`read_html`**

> **`read_html`** uses libraries like lxml and Beautiful Soup to automatically parse tables out of HTML files as DataFrame objects.

> You must install some additional libraries used by **`read_html`**:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

XML and HTML: Web Scraping

> The **`pandas.read_html`** function has a number of options

> By default it searches for and attempts to parse all tabular data contained within **`<table>`** tags

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]:
                      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
```

Parsing XML with **`lxml.objectify`**

> XML (eXtensible Markup Language) is another common structured data format supporting hierarchical, nested data with metadata.

```
<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>
```

Parsing XML with **lxml.objectify**

> Using **lxml.objectify**, we parse the file and get a reference to the root node of the XML file with getroot:

```
from lxml import objectify

path = 'examples/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

Parsing XML with **lxml.objectify**

> **root.INDICATOR** returns a generator yielding each **<INDICATOR>** XML element

```
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)
```

BINARY DATA FORMATS

Binary Data Formats

> One of the easiest ways to store data (also known as *serialization*) efficiently in binary format is using Python's built-in pickle serialization.

> pandas objects all have a **`to_pickle`** method that writes the data to disk in pickle format:

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [89]: frame.to_pickle('examples/frame_pickle')
```

Binary Data Formats

> You can read any “pickled” object stored in a file by using the built-in pickle directly, or even more conveniently using **`pandas.read_pickle`**:

```
In [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Binary Data Formats

> pickle is only recommended as a short-term storage format.

> The problem is that it is hard to guarantee that the format will be stable over time

> An object pickled today may not unpickle with a later version of a library.

> We have tried to maintain backward compatibility when possible, but at some point in the future it may be necessary to “break” the pickle format.

Binary Data Formats

> pandas has built-in support for two more binary data formats: HDF5 and Message-Pack

> Some other storage formats for pandas or NumPy data include:

– *bcolz*

- A compressable column-oriented binary format based on the Blosc compression library.

– *Feather*

- A cross-language column-oriented file format I designed with the R programming community's Hadley Wickham.
- Feather uses the Apache Arrow columnar memory format.

Using HDF5 Format

> HDF5 is a well-regarded file format intended for storing large quantities of scientific array data.

> It is available as a C library, and it has interfaces available in many other languages, including Java, Julia, MATLAB, and Python.

> The "HDF" in HDF5 stands for *hierarchical data format*.

> Each HDF5 file can store multiple datasets and supporting metadata.

> Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compression modes, enabling data with repeated patterns to be stored more efficiently.

> HDF5 can be a good choice for working with very large datasets that don't fit into memory, as you can efficiently read and write small sections of much larger arrays.

Using HDF5 Format

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1                frame        (shape->[100,1])

/obj1_col            series       (shape->[100])

/obj2                frame_table  (typ->appendable,nrows->100,ncols->1,indexers-> [index])

/obj3                frame_table  (typ->appendable,nrows->100,ncols->1,indexers-> [index])
```

Using HDF5 Format

> HDFStore supports two storage schemas, **`'fixed'`** and **`'table'`**.

> **`'table'`** is generally slower, but it supports query operations using a special syntax:

```
In [98]: store.put('obj2', frame, format='table')

In [99]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[99]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [100]: store.close()
```

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, respectively.

> You may need to install these manually with pip or conda.

Reading Microsoft Excel Files

> To use ExcelFile, create an instance by passing a path to an xls or xlsx file:

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

> Data stored in a sheet can then be read into DataFrame with parse:

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Reading Microsoft Excel Files

> You can also simply pass the filename to **`pandas.read_excel`**

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Writing Microsoft Excel Files

> To write pandas data to Excel format, you must first create an **`ExcelWriter`**, then write data to it using pandas objects' **`to_excel`** method:

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [109]: frame.to_excel(writer, 'Sheet1')

In [110]: writer.save()
```

> You can also pass a file path to **`to_excel`** and avoid the **`ExcelWriter`**:

```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

INTERACTING WITH WEB APIS

Interacting with Web APIs

```
In [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

Interacting with Web APIs

```
import time

import requests
import schedule


def call_binance_for_ticker():
  resp = requests.get("https://api.binance.com/api/v3/ticker/price?symbol=BTCUSDT")
  print(resp.json())


schedule.every(1).seconds.do(call_binance_for_ticker)

while 1:
   schedule.run_pending()
   time.sleep(1)
```

```
{'symbol': 'BTCUSDT', 'price': '10344.55000000'}
{'symbol': 'BTCUSDT', 'price': '10344.56000000'}
{'symbol': 'BTCUSDT', 'price': '10344.55000000'}
{'symbol': 'BTCUSDT', 'price': '10345.30000000'}
{'symbol': 'BTCUSDT', 'price': '10344.98000000'}
{'symbol': 'BTCUSDT', 'price': '10344.80000000'}
{'symbol': 'BTCUSDT', 'price': '10344.75000000'}
{'symbol': 'BTCUSDT', 'price': '10342.31000000'}
{'symbol': 'BTCUSDT', 'price': '10342.30000000'}
{'symbol': 'BTCUSDT', 'price': '10342.30000000'}
{'symbol': 'BTCUSDT', 'price': '10341.75000000'}
```

Interacting with Web APIs

```
import asyncio
import json
import websockets as ws

async def consumer_handler(frames):
    async for frame in frames:
        trade = json.loads(frame)
        print(trade)


async def connect():
   async with ws.connect("wss://stream.binance.com:9443/ws/btcusdt@trade") as w:
   await consumer_handler(w)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(connect())
    loop.run_forever()
```

Interacting with Web APIs


```
{'e': 'trade', 'E': 1599880573712, 's': 'BTCUSDT', 't': 412335346, 'p': '10337.58000000', 'q': '0.01324200', 'b': 3190838950, 'a': 3190838817, 'T': 1599880573706, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880573714, 's': 'BTCUSDT', 't': 412335347, 'p': '10337.58000000', 'q': '0.01324200', 'b': 3190838951, 'a': 3190838817, 'T': 1599880573708, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880573715, 's': 'BTCUSDT', 't': 412335348, 'p': '10337.58000000', 'q': '0.01324200', 'b': 3190838952, 'a': 3190838817, 'T': 1599880573708, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880573718, 's': 'BTCUSDT', 't': 412335349, 'p': '10337.58000000', 'q': '0.01324200', 'b': 3190838953, 'a': 3190838817, 'T': 1599880573709, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880573718, 's': 'BTCUSDT', 't': 412335350, 'p': '10337.58000000', 'q': '0.01324200', 'b': 3190838954, 'a': 3190838817, 'T': 1599880573709, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880573742, 's': 'BTCUSDT', 't': 412335351, 'p': '10337.58000000', 'q': '0.01000000', 'b': 3190838956, 'a': 3190838852, 'T': 1599880573741, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880573742, 's': 'BTCUSDT', 't': 412335352, 'p': '10337.73000000', 'q': '0.00137500', 'b': 3190838956, 'a': 3190838749, 'T': 1599880573741, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880574265, 's': 'BTCUSDT', 't': 412335353, 'p': '10337.72000000', 'q': '0.05079000', 'b': 3190838959, 'a': 3190839001, 'T': 1599880574263, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574380, 's': 'BTCUSDT', 't': 412335354, 'p': '10337.65000000', 'q': '0.01698900', 'b': 3190838979, 'a': 3190839015, 'T': 1599880574379, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574380, 's': 'BTCUSDT', 't': 412335355, 'p': '10337.58000000', 'q': '0.03458900', 'b': 3190839014, 'a': 3190839015, 'T': 1599880574379, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574483, 's': 'BTCUSDT', 't': 412335356, 'p': '10337.57000000', 'q': '0.01525500', 'b': 3190838812, 'a': 3190839048, 'T': 1599880574482, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574483, 's': 'BTCUSDT', 't': 412335357, 'p': '10337.57000000', 'q': '0.03316800', 'b': 3190839040, 'a': 3190839048, 'T': 1599880574482, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574528, 's': 'BTCUSDT', 't': 412335358, 'p': '10337.40000000', 'q': '0.01698900', 'b': 3190839056, 'a': 3190839052, 'T': 1599880574527, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880574528, 's': 'BTCUSDT', 't': 412335359, 'p': '10337.58000000', 'q': '0.38301100', 'b': 3190839056, 'a': 3190839028, 'T': 1599880574527, 'm': False, 'M': True}
{'e': 'trade', 'E': 1599880574589, 's': 'BTCUSDT', 't': 412335360, 'p': '10337.40000000', 'q': '0.05048300', 'b': 3190839060, 'a': 3190839072, 'T': 1599880574588, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574703, 's': 'BTCUSDT', 't': 412335361, 'p': '10337.49000000', 'q': '0.01698900', 'b': 3190839076, 'a': 3190839096, 'T': 1599880574702, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574703, 's': 'BTCUSDT', 't': 412335362, 'p': '10337.49000000', 'q': '0.03649500', 'b': 3190839080, 'a': 3190839096, 'T': 1599880574702, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574803, 's': 'BTCUSDT', 't': 412335363, 'p': '10337.49000000', 'q': '0.04930900', 'b': 3190839080, 'a': 3190839101, 'T': 1599880574802, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880574909, 's': 'BTCUSDT', 't': 412335364, 'p': '10337.49000000', 'q': '0.05248300', 'b': 3190839102, 'a': 3190839107, 'T': 1599880574908, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880575869, 's': 'BTCUSDT', 't': 412335365, 'p': '10337.39000000', 'q': '0.00704200', 'b': 3190839053, 'a': 3190839157, 'T': 1599880575868, 'm': True, 'M': True}
{'e': 'trade', 'E': 1599880576587, 's': 'BTCUSDT', 't': 412335366, 'p': '10337.40000000', 'q': '0.11009700', 'b': 3190839186, 'a': 3190839183, 'T': 1599880576585, 'm': False, 'M': True}
```


INTERACTING WITH DATABASES

INTERACTING WITH DATABASES

MYSQL

Install MySQL Driver

> Python needs a MySQL driver to access the MySQL database.

> In this training, we will use the driver "MySQL Connector"

```
pip install mysql-connector-python
```

Test MySQL Connector

> To test if the installation was successful, or if you already have "MySQL Connector" installed, create a Python script

```
import mysql.connector
```

Create Connection

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Secret_123"
)

print(mydb)
```

Creating a Database

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Secret_123"
)
mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE module07")
```

Check if Database Exists

```
mycursor = mydb.cursor()

mycursor.execute("SHOW DATABASES")

for db in mycursor:
  print(db[0])
```

Connecting to a Database

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Secret_123",
  database="module06"
)

print(mydb)
```

Creating a Table

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Secret_123",
  database="module06"
)

mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (id INT AUTO_INCREMENT
PRIMARY KEY, name VARCHAR(255), address VARCHAR(255))")
```

Check if Table Exists

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Secret_123",
  database="module06"
)
mycursor = mydb.cursor()
mycursor.execute("SHOW TABLES")
for table in mycursor:
  print(table[0])
```

Insert Into Table

```
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Secret_123",
  database="module06"
)
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("Jack", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

Insert Multiple Rows

```
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Jack', 'Lowstreet 4'),
  ('Kate', 'Apple st 652'),
  ('James', 'Mountain 21'),
  ('Ben', 'Valley 345'),
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "was inserted.")
```

Get Inserted ID

```
mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("Michelle", "Blue Village")
mycursor.execute(sql, val)

mydb.commit()

print("1 record inserted, ID:", mycursor.lastrowid)
```

Select From a Table

```
mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

for row in myresult:
  print(row)
```

Selecting Columns

```
mycursor = mydb.cursor()

mycursor.execute("SELECT name, address FROM customers")

myresult = mycursor.fetchall()

for row in myresult:
  print(row)
```

Using the **fetchone**() Method

> If you are only interested in one row, you can use the fetchone() method.

> The fetchone() method will return the first row of the result

```
mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchone()

print(myresult)
```

Select With a Filter

> When selecting records from a table, you can filter the selection by using the "WHERE" statement:

```
mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address LIKE '%way%'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

for customer in myresult:
  print(customer)
```

Prevent SQL Injection

> When query values are provided by the user, you should escape the values.

> This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.

> The mysql.connector module has methods to escape query values:

```
mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2" )

mycursor.execute(sql, adr)
myresult = mycursor.fetchall()
for customer in myresult:
  print(customer)
```

Update Table

```
mycursor = mydb.cursor()

sql = "UPDATE customers \
   SET address = 'Canyon 123' \
   WHERE address = 'Valley 345'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

Prevent SQL Injection

```
mycursor = mydb.cursor()

sql = "UPDATE customers \
   SET address = %s \
   WHERE address = %s"
val = ("Valley 345", "Canyon 123")

mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

Delete Record

```
mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Limit the Result

> You can limit the number of records returned from the query, by using the "LIMIT" statement

> Start from position 3, and return 5 records:

```
mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")

myresult = mycursor.fetchall()

for customer in myresult:
  print(customer)
```

Join Two or More Tables

```
mycursor = mydb.cursor()

sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  INNER JOIN products ON users.fav = products.id"

mycursor.execute(sql)
myresult = mycursor.fetchall()

for row in myresult:
```

INTERACTING WITH DATABASES

MONGODB

Terminology

- A *document* is the basic unit of data for MongoDB and is roughly equivalent to a row in a relational database management system (but much more expressive).
- A *collection* can be thought of as a table with a dynamic schema.
- A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections.
- Every document has a special key, "_id", that is unique within a collection.
- MongoDB comes with a simple but powerful JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation.

Key-Value Documents

- An ordered set of keys with associated values.
- The representation of a document varies by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary.
- In JavaScript, for example, documents are represented as objects:

```
{"greeting" : "Hello, world!"}
```

- multiple key/value pairs:

```
{"greeting" : "Hello, world!",
 "foo" : 3}
```

Keys in Documents

- The keys in a document are strings.
- Any UTF-8 character is allowed in a key, with a few notable exceptions:
- Keys must not contain the character **\0** (the null character).
  - This character is used to signify the end of a key.
- The **.** and **$** characters have some special properties and should be used only in certain circumstances.
  - In general, they should be considered reserved, and drivers will complain if they are used inappropriately.

Type-sensitive and Case-sensitive

> MongoDB is type-sensitive and case-sensitive.

> For example, these documents are distinct:

```
{"foo" : 3}
{"foo" : "3"}
```

> as are as these:

```
{"foo" : 3}
{"Foo" : 3}
```

Duplicate Keys

> MongoDB cannot contain duplicate keys.

> For example, the following is not a legal document:

```
{
  "greeting" : "Hello, world!",
  "greeting" : "Hello, MongoDB!"
}
```

Ordered Key-Value Pairs

> Key/value pairs in documents are ordered:

```
{"x" : 1, "y" : 2}
```

> is not the same as

```
{"y" : 2, "x" : 1}
```

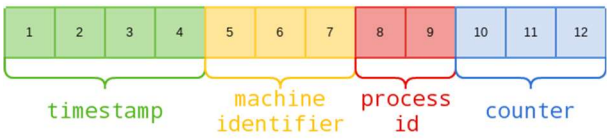
> Field order does not usually matter and you should not design your schema to depend on a certain ordering of fields (MongoDB may reorder them).

"_id"

> Immutable and unique
  - You cannot change after the document is created
  - Two different documents cannot have the same `_id` attribute value

"_id"

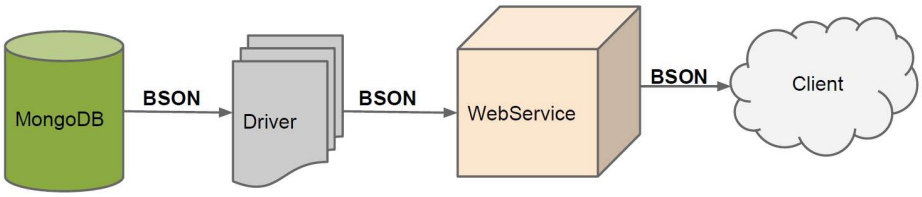
> **ObjectId** (12-Byte) (Big-Endian)
  - Time the data created (4-Byte)
  - Process Id (2-Byte)
  - Machine Id (3-Byte)
  - Incremental number (3-Byte)



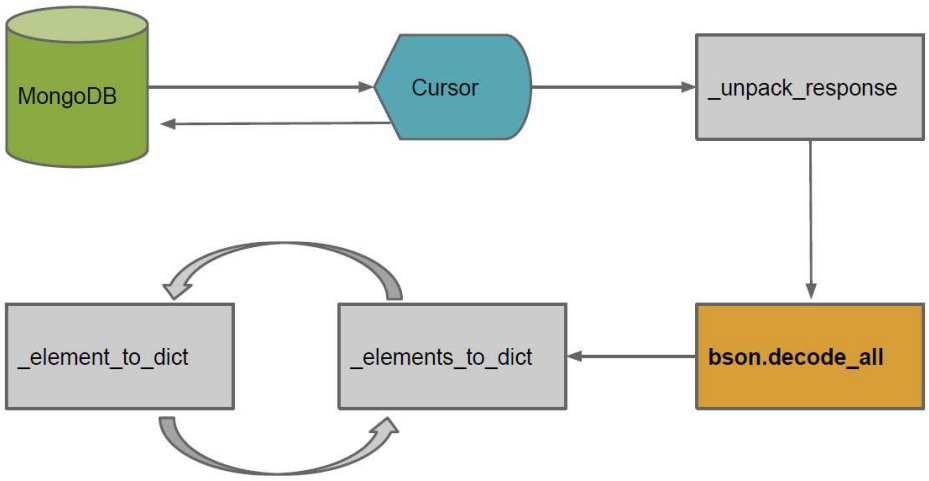
MongoDB Flow

> BSON is the serialization format used by mongodb to talk its clients

> Involves decoding BSON and then re-encoding JSON



The Python Driver



Exercises #1

```
from pymongo import MongoClient
import pprint

client = MongoClient("mongodb://localhost:27017")
db = client["world"]
countries1 = db.countries1

for country in countries1.find():
    pprint.pprint(country)
```

Exercises #2

```
from pymongo import MongoClient
import pprint

client = MongoClient("mongodb://localhost:27017")
db = client["world"]
countries1 = db.countries1

continents = countries1.distinct("continent")
pprint.pprint(continents)
```

Exercises #3

```
from pymongo import MongoClient
import pprint

client = MongoClient("mongodb://localhost:27017")
db = client["imdb"]
movies1 = db.movies1

aggregates = movies1.aggregate([ {"$project": {"genres" : 1} } ,
                                 {"$unwind" : "$genres"}, \
                               {"$group" : { "_id": "$genres.name", \
                                             "total": {"$sum": 1}}} \
                               ])
for agg in aggregates:
    pprint.pprint(agg)
```

Exercises #4

```
import pprint
import pymongo

from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['imdb'];
movies1 = db.movies1

movies = movies1.find({"year" : { "$gte" : 1970, "$lt": 1980 }},\
          {"title": 1, "year":2 , "_id": 0})\
                  .sort([("year", pymongo.DESCENDING),\
                         ("title",pymongo.ASCENDING)])

for movie in movies:
    pprint.pprint(movie)
```

Example #5

```
from pymongo import MongoClient
import pprint
import re

client = MongoClient("mongodb://localhost:27017")
db = client["world"]
countries1 = db.countries1

rge_country_name = re.compile("^.{5}$", re.IGNORECASE)

countries = countries1.find({"name": rge_country_name},\
                            {"name": True, "_id": False})

for country in countries:
    pprint.pprint(country)
```

Example #6

```
import pprint
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['world'];
countries1 = db.countries1

countryCountsByContinent = countries1.group(\
    key={"continent": 1}, condition={}, initial={"count": 0},\
    reduce="function(c,h){ h.count = h.count +1;}")
pprint.pprint(countryCountsByContinent)
```

Example #7

```
import pprint
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['imdb'];
movies1 = db.movies1

movies = movies1.find( \
  { "$where": "this.directors.length > 1" } , \
  {"title": 1, "directors": 1, "_id":0 })

for movie in movies:
    pprint.pprint(movie)
```

Example #8

```
import pprint
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['world'];
countries1 = db.countries1

countriesByContinent = countries1.group(key={"continent": 1},\
                    condition={}, initial={"countries": []},\
           reduce="function(c,h){ h.countries.push(c.name)}")
pprint.pprint(countriesByContinent)
```