DSC 201: Data Analysis & Visualization

Reading Data

Dr. David Koop
Data Frame

- A dictionary of Series (labels for each series)
- A spreadsheet with column headers
- Has an index shared with each series
- Allows easy reference to any cell
- `df = DataFrame({'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada'],
                   'pop': [1.5, 1.7, 3.6, 2.4]})`
- Index is automatically assigned just as with a series but can be passed in as well via index kwarg
- Can reassign column names by passing columns kwarg
Index Objects

• Similar to index for Series
• Immutable
• Can be shared with multiple structures (DataFrames or Series)
• in operator works with: 'Ohio' in df.columns
Indexing

• Same as with NumPy arrays but can use Series's index labels

• Slicing with labels: NumPy is **exclusive**, Pandas is **inclusive**!
  - \( s = \text{Series(np.arange(4))} \)
  \( s[0:2] \) # gives two values like numpy
  - \( s = \text{Series(np.arange(4), index=\['a', 'b', 'c', 'd'\])} \)
  \( s['a':'c'] \) # gives **three** values, not two!

• Obtaining data subsets
  - []: get data by either label or position (can be slow and opaque)
  - \texttt{loc}: get data by label
  - \texttt{iloc}: get data by position (integer index)
  - \texttt{ix}: get data by either label or position (**AVOID!**), checks label first
  - For scalars, also have \texttt{at} and \texttt{iat}
Filtering

- Same as with numpy arrays but allows use of column-based criteria
  - \( \text{data} [\text{data} < 5] = 0 \)
  - \( \text{data} [\text{data} [\text{three}] > 5] \)
  - \( \text{data} < 5 \) creates a boolean data frame that can be used to select specific elements
Data Frame Operations

- Reindexing (.reindex): can fill in missing entries, too (rows or columns)
- Dropping Entries (.drop): drop rows or columns
Arithmetic

- Add, subtract, multiply, and divide are element-wise like numpy
- ...but use labels to align
- ...and missing labels lead to NaN (not a number) values

```
In [28]: obj3
Out[28]:
Ohio     35000
Oregon   16000
Texas     71000
Utah     5000
dtype: int64

In [29]: obj4
Out[29]:
California NaN
Ohio     35000
Oregon   16000
Texas     71000
Utah     5000
dtype: float64

In [30]: obj3 + obj4
Out[30]:
California NaN
Ohio     70000
Oregon   32000
Texas   142000
Utah     NaN
dtype: float64
```

- also have .add, .subtract, ... that allow fill_value argument
- `obj3.add(obj4, fill_value=0)`
Arithmetic between DataFrames and Series

• Broadcasting: e.g. apply single row operation across all rows

• Example:

```python
In [148]: frame
Out[148]:
  b  d  e
Utah 0 1 2
Ohio 3 4 5
Texas 6 7 8
Oregon 9 10 11

In [149]: series
Out[149]:
  b  d  e
  Utah 1
  Ohio 4
  Texas 7
  Oregon 10

In [150]: frame - series
Out[150]:
  b  d  e
  Utah 0 0 0
  Ohio 3 3 3
  Texas 6 6 6
  Oregon 9 9 9
```

• To broadcast over **columns**, use methods (.add, ...)

```python
In [154]: frame
Out[154]:
  b  d  e
Utah 0 1 2
Ohio 3 4 5
Texas 6 7 8
Oregon 9 10 11

In [155]: series3
Out[155]:
  Name: d, dtype: float64

In [156]: frame.sub(series3, axis=0)
Out[156]:
  b  d  e
  Utah -1 0 1
  Ohio -1 0 1
  Texas -1 0 1
  Oregon -1 0 1
```
Assignment 2

- [http://www.cis.umassd.edu/~dkoop/dsc201/assignment2.html](http://www.cis.umassd.edu/~dkoop/dsc201/assignment2.html)
- Analyze data on refugees
- Requires cleaning up the data
- Data stored in dictionaries
- Questions?
Sorting by Index (sort_index)

- **Sort by index (lexicographical):**
  
  In [168]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

  In [169]: obj.sort_index()
  Out[169]:
  a    1
  b    2
  c    3
  d    0
  dtype: int64

- **DataFrame sorting:**

  In [170]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
  columns=['d', 'a', 'b', 'c'])

  In [171]: frame.sort_index()        In [172]: frame.sort_index(axis=1)
  Out[171]:                           Out[172]:
  d  a  b  c                          a  b  c  d
  one    4  5  6  7                   three  1  2  3  0
  three  0  1  2  3                   one    5  6  7  4

- Any missing values are sorted to the end of the Series by default:

  In [176]: obj = Series([4, np.nan, 7, np.nan, -3, 2])
  In [177]: obj.sort_index()
  Out[177]:
  4    -3
  5     2
  0     4

dataframe sorting:

```python
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
    columns=['d', 'a', 'b', 'c'])

In [172]: frame.sort_index(axis=1)
```

- **axis controls sort rows (0) vs. sort columns (1)**
Sorting by Value (sort_values)

• sort_values method on series
  - obj.sort_values()

• Missing values (NaN) are at the end by default (na_position controls, can be first)

• sort_values on DataFrame:
  - df.sort_values(<list-of-columns>)
  - df.sort_values(by=['a', 'b'])
  - Can also use axis=1 to sort by index labels
Ranking

- `rank()` method:
  
  ```python
  In [182]: obj = Series([7, -5, 7, 4, 2, 0, 4])
  In [183]: obj
  Out[183]:
     0    7
     1   -5
     2    7
     3    4
     4    2
     5    0
     6    4
  dtype: int64
  
  In [183]: obj.rank()
  Out[183]:
   0    6.5
   1    1.0
   2    6.5
   3    4.5
   4    3.0
   5    2.0
   6    4.5
  dtype: float64
  ```

- ascending and method arguments:

- Works on data frames, too

```python
In [188]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1], 'c': [-2, 5, 8, -2.5]})
In [187]: frame
Out[187]:
   a    b    c
 0  0  4.3 -2.0
 1  1  7.0  5.0
 2  0 -3.0  8.0
 3  1  2.0 -2.5
```

```python
In [188]: frame.rank(axis=1)
Out[188]:
   a  b  c
 0  2  3  1
 1  1  3  2
 2  2  1  3
 3  2  3  1
```

- Works on data frames, too

```python
In [185]: obj.rank(ascending=False, method='max')
Out[185]:
   0    2
   1    7
   2    2
   3    4
   4    5
   5    6
   6    4
  dtype: float64
```
Statistics

• **sum**: column sums (`axis=1` gives sums over rows)

• missing values are excluded unless the whole slice is NaN

• `idxmax, idxmin` are like `argmax, argmin` (return index)

• `describe`: shortcut for easy stats!

```python
In [204]: df.describe()
Out[204]:
         one       two
count   3.000000  2.000000
mean    3.083333 -2.900000
std     3.493685  2.262742
min     0.750000 -4.500000
25%     1.075000 -3.700000
50%     1.400000 -2.900000
75%     4.250000 -2.100000
max     7.100000 -1.300000
```

Another type of method is neither a reduction nor an accumulation.
`describe` is one such example, producing multiple summary statistics in one shot:

```python
In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [206]: obj.describe()
Out[206]:
count     16
unique     3
top        a
freq       8
dtype: object
```

See Table 5-10 for a full list of summary statistics and related methods.

**Table 5-10. Descriptive and summary statistics**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-NA values</td>
</tr>
<tr>
<td>describe</td>
<td>Compute set of summary statistics for Series or each DataFrame column</td>
</tr>
<tr>
<td>min, max</td>
<td>Compute minimum and maximum values</td>
</tr>
<tr>
<td>argmin, argmax</td>
<td>Compute index locations (integers) at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td>idxmin, idxmax</td>
<td>Compute index values at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td>quantile</td>
<td>Compute sample quantile ranging from 0 to 1</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median (50% quantile) of values</td>
</tr>
<tr>
<td>mad</td>
<td>Mean absolute deviation from mean value</td>
</tr>
<tr>
<td>var</td>
<td>Sample variance of values</td>
</tr>
<tr>
<td>std</td>
<td>Sample standard deviation of values</td>
</tr>
</tbody>
</table>
Another type of method is neither a reduction nor an accumulation. One such example, producing multiple summary statistics in one shot:

```python
In [204]: df.describe()
Out[204]:
   one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

On non-numeric data, `describe` produces alternate summary statistics:

```python
In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [206]: obj.describe()
Out[206]:
   count     16
   unique     3
   top        a
   freq       8
dtype: object
```

See Table 5-10 for a full list of summary statistics and related methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-NA values</td>
</tr>
<tr>
<td>describe</td>
<td>Compute set of summary statistics for Series or each DataFrame column</td>
</tr>
<tr>
<td>min, max</td>
<td>Compute minimum and maximum values</td>
</tr>
<tr>
<td>argmin, argmax</td>
<td>Compute index locations (integers) at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td>idxmin, idxmax</td>
<td>Compute index values at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td>quantile</td>
<td>Compute sample quantile ranging from 0 to 1</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median (50% quantile) of values</td>
</tr>
<tr>
<td>mad</td>
<td>Mean absolute deviation from mean value</td>
</tr>
<tr>
<td>var</td>
<td>Sample variance of values</td>
</tr>
<tr>
<td>std</td>
<td>Sample standard deviation of values</td>
</tr>
<tr>
<td>skew</td>
<td>Sample skewness (3rd moment) of values</td>
</tr>
<tr>
<td>kurt</td>
<td>Sample kurtosis (4th moment) of values</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum of values</td>
</tr>
<tr>
<td>cummin, cummax</td>
<td>Cumulative minimum or maximum of values, respectively</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product of values</td>
</tr>
<tr>
<td>diff</td>
<td>Compute 1st arithmetic difference (useful for time series)</td>
</tr>
<tr>
<td>pct_change</td>
<td>Compute percent changes</td>
</tr>
</tbody>
</table>

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance:

```python
import pandas.io.data as web
all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker)
price = DataFrame({tic: data['Adj Close'] for tic, data in all_data.iteritems()})
volume = DataFrame({tic: data['Volume'] for tic, data in all_data.iteritems()})
```

I now compute percent changes of the prices:

```python
In [208]: returns = price.pct_change()
In [209]: returns.tail()
Out[209]:
   AAPL      GOOG       IBM      MSFT
Date
2014-07-07  0.020632 -0.004241 -0.002599  0.004545
2014-07-08 -0.006460 -0.019167 -0.004361 -0.005001
2014-07-09  0.000420  0.008738  0.006410 -0.002633
2014-07-10 -0.003669 -0.008645 -0.003821  0.000480
2014-07-11  0.001894  0.014148  0.001598  0.009595
```

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

```python
In [210]: returns.MSFT.corr(returns.IBM)
Out[210]: 0.51360438136345077
In [211]: returns.MSFT.cov(returns.IBM)
Out[211]: 8.4825099973219876e-05
```

DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

• Correlation and Covariance

[W. McKinney, Python for Data Analysis]
Unique Values and Value Counts

• unique returns an array with only the unique values (no index)
  
  \[
  s = \text{Series(['c','a','d','a','a','b','b','c','c'])}
  
  s.unique() \# array(['c', 'a', 'd', 'b'])
  
• value_counts returns a Series with index frequencies:
  
  \[
  s.value_counts() \# Series({'c': 3,'a': 3,'b': 2,'d': 1})\]
Handling Missing Data

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropna</td>
<td>Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.</td>
</tr>
<tr>
<td>fillna</td>
<td>Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.</td>
</tr>
<tr>
<td>isnull</td>
<td>Return like-type object containing boolean values indicating which values are missing / NA.</td>
</tr>
<tr>
<td>nonnull</td>
<td>Negation of isnull.</td>
</tr>
</tbody>
</table>
Reading Data in Python

- Use the `open()` method to open a file for reading
- `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate "read"
- Can iterate through the file (think of the file as a collection of lines):
  ```python
  - f = open('huck-finn.txt', 'r')
  for line in f:
    if 'Huckleberry' in line:
      print(line.strip())
  ```
- Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text
- Closing the file: `f.close()`
With Statement: Improved File Handling

• With statement allows "enter" and "exit" handling (kind of like the finally clause):

• In the previous example, we need to remember to call `f.close()`

• Using a with statement, this is done automatically:
  
  ```python
  with open('huck-finn.txt', 'r') as f:
      for line in f:
          if 'Huckleberry' in line:
              print(line.strip())
  ```

• This is more important for writing files!
Comma-separated values (CSV) Format

- Comma is a field separator, newlines denote records
  - a,b,c,d,message
  - 1,2,3,4,hello
  - 5,6,7,8,world
  - 9,10,11,12,foo

- May have a header (a,b,c,d,message), but not required

- No type information: we do not know what the columns are (numbers, strings, floating point, etc.)
  - Default: just keep everything as a string
  - Type inference: Figure out what type to make each column based on what they look like

- What about commas in a value? → double quotes
Delimiter-separated Values

• Comma is a delimiter, specifies boundary between fields
• Could be a tab, pipe (|), or perhaps spaces instead
• All of these follow similar styles to CSV
Fixed-width Format

• Old school
• Each field gets a certain number of spots in the file
• Example:

  - id8141    360.242940    149.910199   11950.7
  id1594    444.953632    166.985655   11788.4
  id1849    364.136849    183.628767   11806.2
  id1230    413.836124    184.375703   11916.8
  id1948    502.953953    173.237159   12468.3

• Specify exact character ranges for each field, e.g. 0-6 is the id
JavaScript Object Notation (JSON)

• A format for web data
• Looks very similar to python dictionaries and lists
• Example:
  ```json
  - {"name": "Wes",
    "places_lived": ["United States", "Spain", "Germany"],
    "pet": null,
    "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
      {"name": "Katie", "age": 33, "pet": "Cisco"}]
  }
  
  • Only contains literals (no variables) but allows null
  • Values: strings, arrays, dictionaries, numbers, booleans, or null
    - Dictionary keys must be strings
    - Quotation marks help differentiate string or numeric values
eXtensible Markup Language (XML)

- Older, self-describing format with nesting
- Each field has tags
- Example:
  - `<INDICATOR>`
  - `<INDICATOR_SEQ>373889</INDICATOR_SEQ>`
  - `<PARENT_SEQ></PARENT_SEQ>`
  - `<AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>`
  - `<INDICATOR_NAME>Escalator Avail.</INDICATOR_NAME>`
  - `<PERIOD_YEAR>2011</PERIOD_YEAR>`
  - `<PERIOD_MONTH>12</PERIOD_MONTH>`
  - `<CATEGORY>Service Indicators</CATEGORY>`
  - `<FREQUENCY>M</FREQUENCY>`
  - `<YTD_TARGET>97.00</YTD_TARGET>`
  - `</INDICATOR>`

- Top element is the root
The tools in this book are of little use if you can't easily import and export data in Python. I'm going to be focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process. NumPy, for example, features low-level but extremely fast binary data loading and storage, including support for memory-mapped array. See Chapter 12 for more on those.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.

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

### Table 6-1. Parsing functions in pandas

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_csv</td>
<td>Load delimited data from a file, URL, or file-like object. Use comma as default delimiter</td>
</tr>
<tr>
<td>read_table</td>
<td>Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiter</td>
</tr>
<tr>
<td>read_fwf</td>
<td>Read data in fixed-width column format (that is, no delimiters)</td>
</tr>
<tr>
<td>read_clipboard</td>
<td>Version of read_table that reads data from the clipboard. Useful for converting tables from web pages</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
read_csv

• Convenient method to read csv files
• Lots of different options to help get data into the desired format
• Basic: `df = pd.read_csv(fname)`
• Parameters:
  - `sep`: the delimiter (',', ' ', '	', '\s+')
  - `header`: if None, no header
  - `names`: list of header names (e.g. if the file has no header)
  - `skiprows`: number of list of lines to skip
Some read_csv/read_tables arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>String indicating filesystem location, URL, or file-like object</td>
</tr>
<tr>
<td>sep or delimiter</td>
<td>Character sequence or regular expression to use to split fields in each row</td>
</tr>
<tr>
<td>header</td>
<td>Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row</td>
</tr>
<tr>
<td>index_col</td>
<td>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</td>
</tr>
<tr>
<td>names</td>
<td>List of column names for result, combine with header=None</td>
</tr>
<tr>
<td>skiprows</td>
<td>Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip</td>
</tr>
<tr>
<td>na_values</td>
<td>Sequence of values to replace with NA</td>
</tr>
<tr>
<td>comment</td>
<td>Character or characters to split comments off the end of lines</td>
</tr>
<tr>
<td>parse_dates</td>
<td>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 (for example if date/time split across two columns)</td>
</tr>
<tr>
<td>keep_date_col</td>
<td>If joining columns to parse date, keep the joined columns. Default False</td>
</tr>
<tr>
<td>converters</td>
<td>Dict containing column number of name mapping to functions. For example { 'foo' : f} would apply the function f to all values in the 'foo' column</td>
</tr>
</tbody>
</table>

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.

```python
In [494]: result = pd.read_csv('ch06/ex6.csv')
In [495]: result
```

[158 | Chapter 6: Data Loading, Storage, and File Formats | D. Koop, DSC 201, Fall 2016 [W. McKinney, Python for Data Analysis]
Python csv module

• Also, can read csv files outside of pandas using csv module

    - import csv
      with open('persons_of_concern.csv', 'r') as f:
          for i in range(3):
              next(f)
          reader = csv.reader(f)
          records = [r for r in reader] # r is a list

• or

    - import csv
      with open('persons_of_concern.csv', 'r') as f:
          for i in range(3):
              next(f)
          reader = csv.DictReader(f)
          records = [r for r in reader] # r is a dict
Writing CSV data

- `df.to_csv(<fname>)`
Reading JSON data

- Python has a built-in `json` module
- Pandas has `read_json`, `to_json` methods