DSC 201: Data Analysis & Visualization

Reading Data

Dr. David Koop
Series

• A one-dimensional array with an **index**
• Index defaults to numbers but can also be text (like a dictionary)
• Allows easier reference to specific items
• Has an associated type just like a NumPy array

```
obj = pd.Series([7,14,-2,1])
```

• Basically two arrays: `obj.values` and `obj.index`
• Can specify the index explicitly and use strings

```
obj2 = pd.Series([4, 7, -5, 3],
                 index=['d', 'b', 'a', 'c'])
```

• Could think of a fixed-length, ordered dictionary
• Can create from a dictionary

```
obj3 = pd.Series({'Ohio': 35000, 'Texas': 71000,
                  'Oregon': 16000, 'Utah': 5000})
```
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

```python
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 = Series(np.arange(4))
    s[0:2] # gives two values like numpy
  - s = 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)
  - loc: get data by label
  - iloc: get data by position (integer index)

- For scalars, also have at and iat
Filtering

• Same as with numpy arrays but allows use of column-based criteria
  - `data[data < 5] = 0`
  - `data[data['three'] > 5]`
  - `data < 5` creates a boolean data frame that can be used to select specific elements
Arithmetic between DataFrames and Series

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

```
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
Utah 0  1
Ohio 3  4
Texas 6  7
Oregon 9 10

In [150]: frame - series
Out[150]:
   b  d  e
Utah  0  0  1
Ohio  3  3  3
Texas  7  6  6
Oregon 10  9  9
```

• To broadcast over **columns**, use methods \( \texttt{.add, ...} \)

```
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]:
   UTah  d
ame: Utah, dtype: float64
Utah  1
Ohio  4
Texas  7
Oregon 10

In [156]: frame.sub(series3, axis=0)
Out[156]:
   b  d  e
Utah -1  0  1
Ohio  3  3  3
Texas -1  0  1
Oregon -1  0  1
```

- If an index value is not found in either the DataFrame’s columns or the Series’s index,
- Operations between DataFrame and Series
- Description
- Method for addition (+)
- Method for subtraction (-)
- Method for multiplication (*)
- Method for division (/)
- Use one of the arithmetic methods. For example:
Other Methods

- **Sorting:**
  - `sort_index`
  - `sort_values`: for DataFrames, pass in the column(s), axis kwarg

- **Ranking:**
  - `rank(ascending=True, method='average')`

- **Describe:**
  - `describe()`: different output for categorical vs. quantitative

- **More:**
  - `count`, `min/max`, `argmin/argmax`, `idxmin`, `idxmax`, `quantile`, `sum`, `mean`, `median`, `mad`, `var`, `std`, `skew`, `kurt`, `cumsum`, `cummin/cummax`, `cumprod`, `diff`, `pct_change`
Unique Values and Value Counts

• unique returns an array with only the unique values (no index)
  - `s = 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})
I do not claim that pandas’s NA representation is optimal, but it is simple and reasonably consistent. It’s the best solution, with good all-around performance characteristics and a simple API, that I could concoct in the absence of a true NA data type or bit pattern in NumPy’s data types. Ongoing development work in NumPy may change this in the future.

Table 5-12. NA handling methods

<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>

Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, dropna can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [233]: from numpy import nan as NA
In [234]: data = Series([1, NA, 3.5, NA, 7])
In [235]: data.dropna()
Out[235]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [236]: data[data.notnull()]
Out[236]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. dropna by default drops any row containing a missing value:
Assignment 3

• http://www.cis.umassd.edu/~dkoop/dsc201-2017fa/assignment3.html

• Create a class that processes refugee data from Assignment 1

• Requires wrangling the data
  - changing column names
  - converting strings

• Call the wrangling functions in the constructor!

• Due Monday
Reading Data in Python

• Use the `open()` method to open a file for reading
  
  \[f = \text{open('huck-finn.txt')}\]

• 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):
  
  \[-f = \text{open('huck-finn.txt', 'r')}\]
  \[\text{for line in } f:\]
  \[\quad \text{if 'Huckleberry' in line:}
  \]
  \[\qquad \text{print(line.strip())}\]

• 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:**
  
<table>
<thead>
<tr>
<th>id</th>
<th>360.242940</th>
<th>149.910199</th>
<th>11950.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>id8141</td>
<td>444.953632</td>
<td>166.985655</td>
<td>11788.4</td>
</tr>
<tr>
<td>id1594</td>
<td>364.136849</td>
<td>183.628767</td>
<td>11806.2</td>
</tr>
<tr>
<td>id1849</td>
<td>413.836124</td>
<td>184.375703</td>
<td>11916.8</td>
</tr>
<tr>
<td>id1230</td>
<td>502.953953</td>
<td>173.237159</td>
<td>12468.3</td>
</tr>
</tbody>
</table>
- Specify exact character ranges for each field, e.g. 0-6 is the id
## Reading Data in Pandas

**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 (i.e., 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>
<tr>
<td>read_excel</td>
<td>Read tabular data from an Excel XLS or XLSX file</td>
</tr>
<tr>
<td>read_hdf</td>
<td>Read HDF5 files written by pandas</td>
</tr>
<tr>
<td>read_html</td>
<td>Read all tables found in the given HTML document</td>
</tr>
<tr>
<td>read_json</td>
<td>Read data from a JSON (JavaScript Object Notation) string representation</td>
</tr>
<tr>
<td>read_msgpack</td>
<td>Read pandas data encoded using the MessagePack binary format</td>
</tr>
<tr>
<td>read_pickle</td>
<td>Read an arbitrary object stored in Python pickle format</td>
</tr>
<tr>
<td>read_sas</td>
<td>Read a SAS dataset stored in one of the SAS system’s custom storage formats</td>
</tr>
<tr>
<td>read_sql</td>
<td>Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame</td>
</tr>
<tr>
<td>read_stata</td>
<td>Read a dataset from Stata file format</td>
</tr>
<tr>
<td>read_feather</td>
<td>Read the Feather binary file format</td>
</tr>
</tbody>
</table>

Because of how messy data in the real world can be, some of the data loading functions (especially read_csv) have grown very complex in their options over time. It's normal to feel overwhelmed by the number of different parameters (read_csv has over 50 as of this writing). The online pandas documentation has many examples about how each of them works, so if you're struggling to read a particular file, there might be a similar enough example to help you find the right parameters.
Types of arguments for readers

• Indexing: choose a column to index the data, get column names from file or user
• Type inference and data conversion: automatic or user-defined
• Datetime parsing: can combine information from multiple columns
• Iterating: deal with very large files
• Unclean Data: skip rows (e.g. comments) or deal with formatted numbers (e.g. 1,000,345)
read_csv

- Convenient method to read csv files
- Lots of different options to help get data into the desired format
- Basic: \texttt{df = pd.read_csv(fname)}

Parameters:
- \texttt{path}: where to read the data from
- \texttt{sep (or delimiter)}: the delimiter (',', ' ', '\t', '\s+')
- \texttt{header}: if None, no header
- \texttt{index\_col}: which column to use as the row index
- \texttt{names}: list of header names (e.g. if the file has no header)
- \texttt{skiprows}: number of list of lines to skip
## More read_csv/read_tables arguments

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

• With very large files, we may not want to read the entire file

• Why?
  - Time
  - Want to understand part of data before processing all of it

• Reading only a few rows:
  - `df = pd.read_csv('example.csv', nrows=5)`

• Reading chunks:
  - Get an iterator that returns the next chunk of the file
  - `chunker = pd.read_csv('example.csv', chunksize=1000)`
  - `for piece in chunker:
      process_data(piece)`
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 with pandas

- **Basic:** `df.to_csv(<fname>)`
- Change delimiter with `sep` kwarg:
  - `df.to_csv('example.dsv', sep='|')`
- Change missing value representation
  - `df.to_csv('example.dsv', na_rep='NULL')`
- Don't write row or column labels:
  - `df.to_csv('example.csv', index=False, header=False)`
- Series may also be written to csv
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
What is the problem with reading this data?

- [{"name": "Wes",
  "places_lived": ["United States", "Spain", "Germany"],
  "pet": null,
  "siblings": [
    {"name": "Scott", "age": 25, "pet": "Zuko"},
    {"name": "Katie", "age": 33, "pet": "Cisco"}
  ]},
  {"name": "Nia",
  "address": {"street": "143 Main",
               "city": "New York",
               "state": "New York"},
  "pet": "Fido",
  "siblings": [
    {"name": "Jacques", "age": 15, "pet": "Fido"}
  ]},
...}
Reading JSON data

• Python has a built-in `json` module
  - `with open('example.json') as f:`
  - `data = json.load(f)`

  - Can also load/dump to strings:
    - `json.loads`, `json.dumps`

• Pandas has `read_json`, `to_json` methods
JSON Orientation

- Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:
  
  - **split**: dict like `{index -> [index],
columns -> [columns],
data -> [values]}`
  - **records**: list like
    ```
    [{column -> value}, ..., {column -> value}]
    ```
  - **columns**: dict like `{column -> {index -> value}}`
  - **values**: just the values array