Advanced Data Management (CSCI 490/680)

Data Wrangling

Dr. David Koop
DataFrame Access and Manipulation

• `df.values` → 2D NumPy array

• Accessing a column:
  - `df["<column>"]`
  - `df.<column>`
  - Both return Series
  - Dot syntax only works when the column is a valid identifier

• Assigning to a column:
  - `df["<column>"] = <scalar> # all cells set to same value`
  - `df["<column>"] = <array> # values set in order`
  - `df["<column>"] = <series> # values set according to match # between df and series indexes`
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 columns by label
  - `loc`: get rows/cols by label
  - `iloc`: get rows/cols by position (integer index)
  - For single cells (scalars), also have `at` and `iat`
Indexing

- `s = Series(np.arange(4.), index=[4,3,2,1])`
- `s[3]`
- `s.loc[3]`
- `s.iloc[3]`
- `s2 = pd.Series(np.arange(4), index=['a','b','c','d'])`
- `s2[3]`
Filtering

• Same as with numpy arrays but allows use of column-based criteria
  - data[data < 5] = 0
  - data[data['three'] > 5]
  - data < 5 → boolean data frame, can be used to select specific elements
Arithmetic

- Add, subtract, multiply, and divide are element-wise like numpy
- …but use labels to align
- …and missing labels lead to \texttt{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         32000
Texas          142000
Utah           NaN
dtype: float64
```

- also have \texttt{.add}, \texttt{.subtract}, ... that allow \texttt{fill_value} argument
- \texttt{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
  0  1  2
  Utah
  3  4  5
  Ohio
  6  7  8
  Texas
  9  10 11
  Oregon
  
  In [149]: series
  Out[149]:
  b  0
  Ohio
  d  1
  Utah
  e  2
  Texas
  
  In [150]: frame - series
  Out[150]:
  b  d  e
  0  1  2
  Utah
  3  4  5
  Ohio
  6  7  8
  Texas
  9  10 11
  Oregon
  ```

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

  ```python
  In [154]: frame
  Out[154]:
  b  d  e
  0  1  2
  Utah
  3  4  5
  Ohio
  6  7  8
  Texas
  9  10 11
  Oregon
  
  In [155]: series3
  Out[155]:
  Name: Utah, dtype: float64

  In [156]: frame - series3
  Out[156]:
  b  d  e
  0  1  2
  Utah
  3  4  5
  Ohio
  6  7  8
  Texas
  9  10 11
  Oregon
  ```
Sorting by Index (sort_index)

• Sort by index (lexicographical):

```python
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:

```python
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
```

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```python
In [173]: frame.sort_index(axis=1, ascending=False)
```

To sort a Series by its values, use its `order` method:

```python
In [174]: obj = Series([4, 7, -3, 2])
In [175]: obj.order()
Out[175]:
2   -3
3    2
0    4
1    7
dtype: int64
```

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

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

• 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
Assignment 2

- Same data as A1, different version of the dataset
- Dealing with the raw data now
- Same questions as A1, but use pandas
- CS680 students + some questions about problems with the data
Ranking

• **rank() method:**

```python
In [182]: obj = Series([7, -5, 7, 4, 2, 0, 4])

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:**

```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
```

• **Works on data frames, too**

```python
In [178]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [179]: frame        Out[179]:
      a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [180]: frame.sort_index(by='b')
Out[180]:
      a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [181]: frame.sort_index(by=['a', 'b'])
Out[181]:
      a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

Ranking is closely related to sorting, assigning ranks from one through the number of valid data points in an array. It is similar to the indirect sort indices produced by `numpy.argsort`, except that ties are broken according to a rule. The `rank` methods for `Series` and `DataFrame` are the place to look; by default `rank` breaks ties by assigning each group the mean rank:

```python
In [182]: obj = Series([7, -5, 7, 4, 2, 0, 4])
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
```

Ranks can also be assigned according to the order they’re observed in the data:

```python
In [184]: obj.rank(method='first')
Out[184]:
0   6
1   1
2   7
3   4
4   3
5   2
6   5
dtype: float64
```

Naturally, you can rank in descending order, 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
```

See Table 5-8 for a list of tie-breaking methods available. `DataFrame` can compute ranks over the rows or the columns:

```python
In [186]: 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

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

### Table 5-8: Tie-breaking methods with `rank`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'average'</td>
<td>Default: assign the average rank to each entry in the equal group.</td>
</tr>
<tr>
<td>'min'</td>
<td>Use the minimum rank for the whole group.</td>
</tr>
<tr>
<td>'max'</td>
<td>Use the maximum rank for the whole group.</td>
</tr>
<tr>
<td>'first'</td>
<td>Assign ranks in the order the values appear in the data.</td>
</tr>
</tbody>
</table>

Up until now all of the examples I’ve showed you have had unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it’s not mandatory. Let’s consider a small Series with duplicate indices:

```python
In [189]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
In [190]: obj
Out[190]:
a    0
a    1
b    2
b    3
c    4
```

Axis indexes with duplicate values

D. Koop, CSCI 680/490, Spring 2021
Statistics

- **sum**: column sums ($\text{axis}=1$ gives sums over rows)
- missing values are excluded unless the whole slice is $\text{NaN}$
- **idxmax, idxmin** are like argmax, argmin (return index)
- **describe**: shortcut for easy stats!

```
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:

```
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:

```
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><strong>describe</strong></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>
One type of method is neither a reduction nor an accumulation. Describe is one such example, producing multiple summary statistics in one shot:

```
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:

```
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:

```
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:

```
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:
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'])
```  
- **Data Frames** use **drop_duplicates**
- **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>

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.

```
140 | Chapter 5: Getting Started with pandas
```

[D. Koop, CSCI 680/490, Spring 2021]
Data Formats
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 the type to make each column based on values

- 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
Reading & Writing Data
• 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):
  - `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 does "enter" and "exit" handling (similar to 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!
  ```python
  with open('output.txt', 'w') as f:
    for k, v in counts.items():
      f.write(k + ': ' + v + '\n')
  ```
• Without `with`, we need `f.close()`
# Reading & Writing Data in Pandas

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Description</th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>CSV</td>
<td>read_csv</td>
<td>to_csv</td>
</tr>
<tr>
<td>text</td>
<td>Fixed-Width Text File</td>
<td>read_fwf</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>JSON</td>
<td>read_json</td>
<td>to_json</td>
</tr>
<tr>
<td>text</td>
<td>HTML</td>
<td>read_html</td>
<td>to_html</td>
</tr>
<tr>
<td>text</td>
<td>Local clipboard</td>
<td>read_clipboard</td>
<td>to_clipboard</td>
</tr>
<tr>
<td></td>
<td>MS Excel</td>
<td>read_excel</td>
<td>to_excel</td>
</tr>
<tr>
<td>binary</td>
<td>OpenDocument</td>
<td>read_excel</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>HDF5 Format</td>
<td>read_hdf</td>
<td>to_hdf</td>
</tr>
<tr>
<td>binary</td>
<td>Feather Format</td>
<td>read_feather</td>
<td>to_feather</td>
</tr>
<tr>
<td>binary</td>
<td>Parquet Format</td>
<td>read_parquet</td>
<td>to_parquet</td>
</tr>
<tr>
<td>binary</td>
<td>ORC Format</td>
<td>read_orc</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>Msgpack</td>
<td>read_msgpack</td>
<td>to_msgpack</td>
</tr>
<tr>
<td>binary</td>
<td>Stata</td>
<td>read_stata</td>
<td>to_stata</td>
</tr>
<tr>
<td>binary</td>
<td>SAS</td>
<td>read_sas</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>SPSS</td>
<td>read_spss</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td>read_pickle</td>
<td>to_pickle</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>read_sql</td>
<td>to_sql</td>
</tr>
<tr>
<td>SQL</td>
<td>Google BigQuery</td>
<td>read_gbq</td>
<td>to_gbq</td>
</tr>
</tbody>
</table>

[https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html]
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:** `df = pd.read_csv(fname)`
- **Parameters:**
  - `path`: where to read the data from
  - `sep` (or `delimiter`): the delimiter (',', '#', '	', '\s+')
  - `header`: if None, no header
  - `index_col`: which column to use as the row index
  - `names`: list of header names (e.g. if the file has no header)
  - `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>
</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:
  - \texttt{df = pd.read\_csv('example.csv', nrows=5)}

• Reading chunks:
  - Get an iterator that returns the next chunk of the file
    - \texttt{chunker = pd.read\_csv('example.csv', chunksize=1000)}
  - for piece in chunker:
    \hspace{1cm} 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
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
No built-in method

Use lxml library (also can use ElementTree)

```python
from lxml import objectify
path = 'datasets/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
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)
perf = pd.DataFrame(data)
```

[W. McKinney, Python for Data Analysis]
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
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}]`

  - **index**: dict like `{index -> {column -> value}}`

  - **columns**: dict like `{column -> {index -> value}}`

  - **values**: just the values array
Binary Formats

- CSV, JSON, and XML are all text formats
- What is a binary format?
- Pickle: Python's built-in serialization
- HDF5: Library for storing large scientific data
  - Hierarchical Data Format, supports compression
  - Interfaces in C, Java, MATLAB, etc.
  - Use `pd.HDFStore` to access
  - Shortcuts: `read_hdf/to_hdf`, need to specify object
- Excel: need to specify sheet when a spreadsheet has multiple sheets
  - `pd.ExcelFile` or `pd.read_excel`
Databases

Dim_Date
- Id
- Date
- Day
- Day_of_Week
- Month
- Month_Name
- Quarter
- Quarter_Name
- Year

Fact_Sales
- Date_Id
- Store_Id
- Product_Id
- Units_Sold

Dim_Store
- Id
- Store_Number
- State_Province
- Country

Dim_Product
- Id
- EAN_Code
- Product_Name
- Brand
- Product_Category

[Wikipedia]
Databases

• Relational databases are similar to multiple data frames but have many more features
  - links between tables via foreign keys
  - SQL to create, store, and query data
• sqlite3 is a simple database with built-in support in python
• Python has a database API which lets you access most database systems through a common API.
Python DBAPI Example

```python
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()
# Insert some data
data = [('Atlanta', 'Georgia', 1.25, 6),
    ('Tallahassee', 'Florida', 2.6, 3),
    ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
con.executemany(stmt, data)
con.commit()
```

[W. McKinney, Python for Data Analysis]
Databases

- Similar syntax from other database systems (MySQL, Microsoft SQL Server, Oracle, etc.)
- SQLAlchemy: Python package that abstracts away differences between different database systems
- SQLAlchemy gives support for reading queries to data frame:
  ```python
  import sqlalchemy as sqla
db = sqla.create_engine('sqlite:///mydata.sqlite')
pd.read_sql('select * from test', db)
  ```
What if data isn't correct/trustworthy/in the right format?
Dirty Data

[Flickr]
Geolocation Errors

• Maxmind helps companies determine where users are located based on IP address

• "How a quiet Kansas home wound up with 600 million IP addresses and a world of trouble" [Washington Post, 2016]
Numeric Outliers

ages of employees (US)

- median 37
- mean 58.52632
- variance 9252.041

[J. Hellerstein via J. Canny et al.]
FINDINGS

we got about the future of the data science, the most salient takeaway was how excited our respondents were about the evolution of the field. They cited things in their own practice, how they saw their jobs getting more interesting and less repetitive, all while expressing a real and broad enthusiasm about the value of the work in their organization.

As data science becomes more commonplace and simultaneously a bit demystified, we expect this trend to continue as well. After all, last year's respondents were just as excited about their work (about 79% were "satisfied" or better).

How a Data Scientist Spends Their Day

Here's where the popular view of data scientists diverges pretty significantly from reality. Generally, we think of data scientists building algorithms, exploring data, and doing predictive analysis. That's actually not what they spend most of their time doing, however.

As you can see from the chart above, 3 out of every 5 data scientists we surveyed actually spend the most time cleaning and organizing data. You may have heard this referred to as "data wrangling" or compared to digital janitor work. Everything from list verification to removing commas to debugging databases— that time adds up and it adds up immensely. Messy data is by far the more time-consuming aspect of the typical data scientist's work flow. And nearly 60% said they simply spent too much time doing it.

[Data scientist job satisfaction chart]

- Building training sets: 3%
- Cleaning and organizing data: 60%
- Collecting data sets: 19%
- Mining data for patterns: 9%
- Refining algorithms: 4%
- Other: 5%

This takes a lot of time!

[CrowdFlower Data Science Report, 2016]
FINDINGS

Why That’s a Problem

Simply put, data wrangling isn’t fun. It takes forever. In fact, a few years back, the New York Times estimated that up to 80% of a data scientist’s time is spent doing this sort of work.

Here, it’s necessary to point out that data cleaning is incredibly important. You can’t do the sort of work data scientists truly enjoy doing with messy data. It needs to be cleaned, labeled, and enriched before you can trust the output.

The problem here is two-fold. One: data scientists simply don’t like doing this kind of work, and, as mentioned, this kind of work takes up most of their time. We asked our respondents what was the least enjoyable part of their job.

They had this to say:

Note how those last two charts mirror each other. The things data scientists do most are the things they enjoy least. Last year, we found that respondents far prefer doing the more creative, interesting parts of their job, things like predictive analysis and mining data for patterns. That’s where the real value comes. But again, you simply can’t do that work unless the data is properly labeled. And nobody likes labeling data.

Do Data Scientists Have What They Need?

With a shortage of data scientists out there in the world, we wanted to find out if they thought they were properly supported in their job. After all, when you need more data scientists, you’ll often find a single person doing the work of several.

What’s the least enjoyable part of data science?

- Building training sets: 10%
- Cleaning and organizing data: 57%
- Collecting data sets: 21%
- Mining data for patterns: 3%
- Refining algorithms: 4%
- Other: 5%

[CrowdFlower Data Science Report, 2016]
Dirty Data: Statistician's View

• Some process produces the data
• Want a model but have non-ideal samples:
  - Distortion: some samples corrupted by a process
  - Selection bias: likelihood of a sample depends on its value
  - Left and right censorship: users come and go from scrutiny
  - Dependence: samples are not independent (e.g. social networks)
• You can add/augment models for different problems, but cannot model everything
• Trade-off between accuracy and simplicity
Dirty Data: Database Expert's View

• Got a dataset
• Some values are missing, corrupted, wrong, duplicated
• Results are absolute (relational model)
• Better answers come from improving the quality of values in the dataset
Dirty Data: Domain Expert's View

- Data doesn't look right
- Answer doesn't look right
- What happened?
- Domain experts carry an implicit model of the data they test against
- You don't always need to be a domain expert to do this
  - Can a person run 50 miles an hour?
  - Can a mountain on Earth be 50,000 feet above sea level?
  - Use common sense

[J. Canny et al.]
Dirty Data: Data Scientist's View

- Combination of the previous three views
- All of the views present problems with the data
- The goal may dictate the solutions:
  - Median value: don't worry too much about crazy outliers
  - Generally, aggregation is less susceptible by numeric errors
  - Be careful, the data may be correct…
Be careful how you detect dirty data

- The appearance of a hole in the earth’s ozone layer over Antarctica, first detected in 1976, was so unexpected that scientists didn’t pay attention to what their instruments were telling them; they thought their instruments were malfunctioning.

  – National Center for Atmospheric Research
Where does dirty data originate?

• Source data is bad, e.g. person entered it incorrectly
• Transformations corrupt the data, e.g. certain values processed incorrectly due to a software bug
• Integration of different datasets causes problems
• Error propagation: one error is magnified
Types of Dirty Data Problems

- **Separator Issues**: e.g. CSV without respecting double quotes
  - 12, 13, "Doe, John", 45
- **Naming Conventions**: NYC vs. New York
- **Missing required fields**, e.g. key
- **Different representations**: 2 vs. two
- **Truncated data**: "Janice Keihanaikukauakahihuliheekahaunaele" becomes "Janice Keihanaikukauakahihuliheek" on Hawaii license
- **Redundant records**: may be exactly the same or have some overlap
- **Formatting issues**: 2017–11–07 vs. 07/11/2017 vs. 11/07/2017

[J. Canny et al.]
Data Wrangling

- Data wrangling: transform raw data to a more meaningful format that can be better analyzed
- Data cleaning: getting rid of inaccurate data
- Data transformations: changing the data from one representation to another
- Data reshaping: reorganizing the data
- Data merging: combining two datasets
Data Cleaning
Wrangler: Interactive Visual Specification of Data Transformation Scripts

S. Kandel, A. Paepcke, J. Hellerstein, J. Heer
Data Wrangler Demo

- [http://vis.stanford.edu/wrangler/app/](http://vis.stanford.edu/wrangler/app/)

### Transform Script

- **Split data repeatedly on newline into rows**
- **Split split repeatedly on ','**
- **Promote row 0 to header**
- **Delete row 7**
- **Delete empty rows**
- **Fill row 7 by copying values from above**

<table>
<thead>
<tr>
<th>Year</th>
<th>Property_crime_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reported crime in Alabama</td>
<td>0</td>
</tr>
<tr>
<td>2004</td>
<td>4029.3</td>
</tr>
<tr>
<td>2005</td>
<td>3900</td>
</tr>
<tr>
<td>2006</td>
<td>3937</td>
</tr>
<tr>
<td>2007</td>
<td>3974.9</td>
</tr>
<tr>
<td>2008</td>
<td>4081.9</td>
</tr>
<tr>
<td>Reported crime in Alaska</td>
<td>8</td>
</tr>
<tr>
<td>2004</td>
<td>3370.9</td>
</tr>
<tr>
<td>2005</td>
<td>3615</td>
</tr>
<tr>
<td>2006</td>
<td>3582</td>
</tr>
</tbody>
</table>
Wrangler

- Data cleaning takes a lot of **time** and **human effort**
- "Tedium is the message"
- Repeating this process on multiple data sets is even worse!
- Solution:
  - interactive interface (mixed-initiative)
  - transformation language with natural language "translations"
  - suggestions + "programming by demonstration"