Advanced Data Management (CSCI 490/680)

Data Wrangling

Dr. David Koop
pandas

- Contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python
- Built on top of NumPy
- Requirements:
  - Data structures with labeled axes (aligning data)
  - Time series data
  - Arithmetic operations that include metadata (labels)
  - Handle missing data
  - Merge and relational operations
Series

- A one-dimensional array (with a type) with an **index**
- Index defaults to numbers but can also be text (like a dictionary)
- Allows easier reference to specific items
- `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'])`
- Kind of like 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

- 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
Indexing

• Same as with NumPy arrays but can use Series's index labels
• Slicing with labels: NumPy is **exclusive**, Pandas is **inclusive**!
  
  - \( s = \text{Series}(\text{np.arange}(4)) \)
    \( s[0:2] \) # gives two values like numpy
  
  - \( s = \text{Series}(\text{np.arange}(4), \text{index}=\text{['a', 'b', 'c', 'd']}) \)
    \( s[\text{a':'c']} \) # 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 Data Frames

• Brackets can be ambiguous:
  - df['Address']
  - df[0:4]

• .loc and .iloc require more code (always row and column), but are clearer
  - df.loc[:,'Address']
  - df.iloc[0:4,:]

• Putting them together:
  - df.iloc[0:4,:].loc[:,'Address']
  - df.loc[df.index[0:4],'Address']
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
Unique Values and Value Counts

• **unique** returns an array with only the unique values (no index)
  
  ```python
  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:
  
  ```python
  s.value_counts() # Series({'c': 3,'a': 3,'b': 2,'d': 1})
  ```
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!

```
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 -2.100000
75%   4.250000 -2.100000
max   7.100000 -1.300000
```

```
In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [206]: obj.describe()
Out[206]:
          count unique  top     freq  dtype
count      16     3      a       8  object
```
Assignment 2

• Similar to Assignment 1, now with pandas

• Part 5:
  - CS 680 → Required
  - CS 490 → Extra Credit

• Due Friday, Feb. 7

<table>
<thead>
<tr>
<th>year</th>
<th>month 1</th>
<th>month 2</th>
<th>month 3</th>
<th>month 4</th>
<th>month 5</th>
<th>month 6</th>
<th>month 7</th>
<th>month 8</th>
<th>month 9</th>
<th>month 10</th>
<th>month 11</th>
<th>month 12</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>1851</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1852</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1853</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1854</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1855</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2015</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2016</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>2017</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>2018</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>All</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>38</td>
<td>125</td>
<td>171</td>
<td>448</td>
<td>623</td>
<td>353</td>
<td>89</td>
<td>14</td>
<td>1873</td>
</tr>
</tbody>
</table>

169 rows x 13 columns
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:**

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>360.242940</th>
<th>149.910199</th>
<th>11950.7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id8141</td>
<td>360.242940</td>
<td>149.910199</td>
<td>11950.7</td>
</tr>
<tr>
<td></td>
<td>id1594</td>
<td>444.953632</td>
<td>166.985655</td>
<td>11788.4</td>
</tr>
<tr>
<td></td>
<td>id1849</td>
<td>364.136849</td>
<td>183.628767</td>
<td>11806.2</td>
</tr>
<tr>
<td></td>
<td>id1230</td>
<td>413.836124</td>
<td>184.375703</td>
<td>11916.8</td>
</tr>
<tr>
<td></td>
<td>id1948</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 & 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><strong>CSV</strong></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><strong>JSON</strong></td>
<td>read_json</td>
<td>to_json</td>
</tr>
<tr>
<td>text</td>
<td><strong>HTML</strong></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>MS Excel</td>
<td></td>
<td>read_excel</td>
<td>to_excel</td>
</tr>
<tr>
<td>binary</td>
<td><strong>OpenDocument</strong></td>
<td>read_excel</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td><strong>HDF5 Format</strong></td>
<td>read_hdf</td>
<td>to_hdf</td>
</tr>
<tr>
<td>binary</td>
<td><strong>Feather Format</strong></td>
<td>read_feather</td>
<td>to_feather</td>
</tr>
<tr>
<td>binary</td>
<td><strong>Parquet Format</strong></td>
<td>read_parquet</td>
<td>to_parquet</td>
</tr>
<tr>
<td>binary</td>
<td><strong>ORC Format</strong></td>
<td>read_orc</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td><strong>Msgpack</strong></td>
<td>read_msgpack</td>
<td>to_msgpack</td>
</tr>
<tr>
<td>binary</td>
<td><strong>Stata</strong></td>
<td>read_stata</td>
<td>to_stata</td>
</tr>
<tr>
<td>binary</td>
<td><strong>SAS</strong></td>
<td>read_sas</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td><strong>SPSS</strong></td>
<td>read_spss</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td><strong>Python Pickle Format</strong></td>
<td>read_pickle</td>
<td>to_pickle</td>
</tr>
<tr>
<td>SQL</td>
<td><strong>SQL</strong></td>
<td>read_sql</td>
<td>to_sql</td>
</tr>
<tr>
<td>SQL</td>
<td><strong>Google BigQuery</strong></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: \(df = \text{pd.read_csv}(\text{fname})\)

- 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>

[W. McKinney, Python for Data Analysis]
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
eXtensible Markup Language (XML)

- Older, self-describing format with nesting; each field has tags
- Example:
  
  ```xml
  <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
XML

• No built-in method
• Use lxml library (also can use ElementTree)
• 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:
  - `{"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?

• 

```json
[{"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
  - Interfaces in C, Java, MATLAB, etc.
  - Supports **compression**
  - 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
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
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 job—seeing 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.

This takes a lot of time!

What data scientists spend the most time doing

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

[CrowdFlower Data Science Report, 2016]
...and it isn't the most fun thing to do

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

[J. Canny et al.]
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 Keihanaikukauakahihiuliheekahaunaele" becomes "Janice Keihanaikukauakahihiuliheek" 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

### Data Table

<table>
<thead>
<tr>
<th>Year</th>
<th>Property_crime_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reported crime in Alabama</td>
</tr>
<tr>
<td>2 2004</td>
<td>4029.3</td>
</tr>
<tr>
<td>3 2005</td>
<td>3900</td>
</tr>
<tr>
<td>4 2006</td>
<td>3937</td>
</tr>
<tr>
<td>5 2007</td>
<td>3974.9</td>
</tr>
<tr>
<td>6 2008</td>
<td>4081.9</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Reported crime in Alaska</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10 2004</td>
<td>3370.9</td>
</tr>
<tr>
<td>11 2005</td>
<td>3615</td>
</tr>
<tr>
<td>12 2006</td>
<td>3582</td>
</tr>
</tbody>
</table>