Advanced Data Management (CSCI 490/680)

Data Wrangling

Dr. David Koop
Tables

Flat
- Data organized by rows & columns
  - row ~ item (usually)
  - column ~ attribute
  - label ~ attribute name
- Key: identifies each item (row)
  - Usually **unique**
  - Allows **join** of data from 2+ tables
  - Compound key: key split among multiple columns, e.g. (state, year) for population

Multidimensional
- Split compound key

[Munzner (ill. Maguire), 2014]
Attribute Types

- **Categorical**
- **Ordered**
  - **Ordinal**
  - **Quantitative**

[Muñzner (ill. Maguire), 2014]
## Categorical, Ordinal, and Quantitative

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Order ID</td>
<td>Order Date</td>
<td>Order Priority</td>
<td>Product Size</td>
<td>Product Base Margin</td>
</tr>
<tr>
<td>3</td>
<td>10/14/06</td>
<td>5-Low</td>
<td>Large Box</td>
<td>0.8</td>
<td>10/21/06</td>
</tr>
<tr>
<td>6</td>
<td>2/21/08</td>
<td>4-Not Specified</td>
<td>Small Pack</td>
<td>0.55</td>
<td>2/22/08</td>
</tr>
<tr>
<td>32</td>
<td>7/16/07</td>
<td>2-High</td>
<td>Small Pack</td>
<td>0.79</td>
<td>7/17/07</td>
</tr>
<tr>
<td>32</td>
<td>7/16/07</td>
<td>2-High</td>
<td>Jumbo Box</td>
<td>0.72</td>
<td>7/17/07</td>
</tr>
<tr>
<td>32</td>
<td>7/16/07</td>
<td>2-High</td>
<td>Medium Box</td>
<td>0.6</td>
<td>7/18/07</td>
</tr>
<tr>
<td>32</td>
<td>7/16/07</td>
<td>2-High</td>
<td>Medium Box</td>
<td>0.65</td>
<td>7/18/07</td>
</tr>
<tr>
<td>35</td>
<td>10/23/07</td>
<td>4-Not Specified</td>
<td>Wrap Bag</td>
<td>0.52</td>
<td>10/24/07</td>
</tr>
<tr>
<td>35</td>
<td>10/23/07</td>
<td>4-Not Specified</td>
<td>Small Box</td>
<td>0.58</td>
<td>10/25/07</td>
</tr>
<tr>
<td>36</td>
<td>11/3/07</td>
<td>1-Urgent</td>
<td>Small Box</td>
<td>0.55</td>
<td>11/3/07</td>
</tr>
<tr>
<td>65</td>
<td>3/18/07</td>
<td>1-Urgent</td>
<td>Small Pack</td>
<td>0.49</td>
<td>3/19/07</td>
</tr>
<tr>
<td>66</td>
<td>1/20/05</td>
<td>5-Low</td>
<td>Wrap Bag</td>
<td>0.56</td>
<td>1/20/05</td>
</tr>
<tr>
<td>69</td>
<td>6/4/05</td>
<td>4-Not Specified</td>
<td>Small Pack</td>
<td>0.44</td>
<td>6/6/05</td>
</tr>
<tr>
<td>69</td>
<td>6/4/05</td>
<td>4-Not Specified</td>
<td>Small Pack</td>
<td>0.6</td>
<td>6/6/05</td>
</tr>
<tr>
<td>70</td>
<td>12/18/06</td>
<td>5-Low</td>
<td>0.59</td>
<td>12/23/06</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>12/18/06</td>
<td>5-Low</td>
<td>0.82</td>
<td>12/23/06</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>4/17/05</td>
<td>2-High</td>
<td>0.38</td>
<td>4/19/05</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>1/29/06</td>
<td>3-Medium</td>
<td>0.38</td>
<td>1/30/06</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>11/19/08</td>
<td>5-Low</td>
<td>0.37</td>
<td>11/28/08</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>5/8/08</td>
<td>2-High</td>
<td>Small Box</td>
<td>0.37</td>
<td>5/9/08</td>
</tr>
<tr>
<td>130</td>
<td>5/8/08</td>
<td>2-High</td>
<td>Medium Box</td>
<td>0.38</td>
<td>5/10/08</td>
</tr>
<tr>
<td>130</td>
<td>5/8/08</td>
<td>2-High</td>
<td>Small Box</td>
<td>0.6</td>
<td>5/11/08</td>
</tr>
<tr>
<td>132</td>
<td>6/11/06</td>
<td>3-Medium</td>
<td>Medium Box</td>
<td>0.6</td>
<td>6/12/06</td>
</tr>
<tr>
<td>132</td>
<td>6/11/06</td>
<td>3-Medium</td>
<td>Jumbo Box</td>
<td>0.69</td>
<td>6/14/06</td>
</tr>
<tr>
<td>134</td>
<td>5/1/08</td>
<td>4-Not Specified</td>
<td>Large Box</td>
<td>0.82</td>
<td>5/3/08</td>
</tr>
<tr>
<td>135</td>
<td>10/21/07</td>
<td>4-Not Specified</td>
<td>Small Pack</td>
<td>0.64</td>
<td>10/23/07</td>
</tr>
<tr>
<td>166</td>
<td>9/12/07</td>
<td>2-High</td>
<td>Small Box</td>
<td>0.55</td>
<td>9/14/07</td>
</tr>
<tr>
<td>193</td>
<td>8/8/06</td>
<td>1-Urgent</td>
<td>Medium Box</td>
<td>0.57</td>
<td>8/10/06</td>
</tr>
<tr>
<td>194</td>
<td>4/5/08</td>
<td>3-Medium</td>
<td>Wrap Bag</td>
<td>0.42</td>
<td>4/7/08</td>
</tr>
</tbody>
</table>
Semantics

- The meaning of the data
- Example: 94023, 90210, 02747, 60115
Semantics

• The meaning of the data
• Example: 94023, 90210, 02747, 60115
  - Attendance at college football games?
Semantics

• The meaning of the data
• Example: 94023, 90210, 02747, 60115
  - Attendance at college football games?
  - Salaries?
Semantics

• The meaning of the data

• Example: 94023, 90210, 02747, 60115
  - Attendance at college football games?
  - Salaries?
  - Zip codes?

• Cannot always infer based on what the data looks like

• Often require semantics to better understand data

• Column names help with semantics

• May also include rules about data: a zip code is part of an address that uniquely identifies a residence

• Useful for asking good questions about the data
Data Model vs. Conceptual Model

- **Data Model:** raw data that has a specific data type (e.g. floats):
  - Temperature Example: \([32.5, 54.0, -17.3]\) (floats)

- **Conceptual Model:** how we think about the data
  - Includes semantics, reasoning
  - Temperature Example:
    - **Quantitative:** \([32.50, 54.00, -17.30]\)

[via A. Lex, 2015]
Data Model vs. Conceptual Model

- **Data Model**: raw data that has a specific data type (e.g. floats):
  - Temperature Example: \([32.5, 54.0, -17.3]\) (floats)

- **Conceptual Model**: how we think about the data
  - Includes semantics, reasoning
  - Temperature Example:
    - Quantitative: \([32.50, 54.00, -17.30]\)
    - Ordered: \([\text{warm}, \text{hot}, \text{cold}]\)
Data Model vs. Conceptual Model

• Data Model: raw data that has a specific data type (e.g. floats):
  - Temperature Example: [32.5, 54.0, -17.3] (floats)

• Conceptual Model: how we think about the data
  - Includes semantics, reasoning
  - Temperature Example:
    • Quantitative: [32.50, 54.00, -17.30]
    • Ordered: [warm, hot, cold]
    • Categorical: [not burned, burned, not burned]
Chicago Food Inspections Exploration

- Based on David Beazley's PyData Chicago talk
- YouTube video: https://www.youtube.com/watch?v=j6VSAsKAj98
- Our in-class exploration:
  - Python can give answers fairly quickly
  - Data analysis questions:
    - What is information is available
    - **Questions** are interesting about this dataset
    - How to decide on good follow-up questions
    - What the computations mean
Assignment 2

• Similar to Assignment 1, now with pandas
• Part 5:
  - CS 680 → Required
  - CS 490 → Extra Credit
• Due Friday, Feb. 7
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
🎉 pandas 1.0.0 🎉
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})`
Series

- **Indexing:** `s[1]` or `s['Oregon']`
- **Can check for missing data:** `pd.isnull(s)` or `pd.notnull(s)`
- **Both index and values can have an associated name:**
  - `s.name = 'population'; s.index.name = 'state'`
- **Addition and NumPy ops work as expected and preserve the index-value link**
- **These operations align:**

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

  [W. McKinney, Python for Data Analysis]
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
DataFrame Constructor Inputs

<table>
<thead>
<tr>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D ndarray</td>
<td>A matrix of data, passing optional row and column labels</td>
</tr>
<tr>
<td>dict of arrays, lists, or tuples</td>
<td>Each sequence becomes a column in the DataFrame. All sequences must be the same length.</td>
</tr>
<tr>
<td>NumPy structured/record array</td>
<td>Treated as the “dict of arrays” case</td>
</tr>
<tr>
<td>dict of Series</td>
<td>Each value becomes a column. Indexes from each Series are unioned together to form the result’s row index if no explicit index is passed.</td>
</tr>
<tr>
<td>dict of dicts</td>
<td>Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.</td>
</tr>
<tr>
<td>list of dicts or Series</td>
<td>Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels</td>
</tr>
<tr>
<td>List of lists or tuples</td>
<td>Treated as the “2D ndarray” case</td>
</tr>
<tr>
<td>Another DataFrame</td>
<td>The DataFrame’s indexes are used unless different ones are passed</td>
</tr>
<tr>
<td>NumPy MaskedArray</td>
<td>Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
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
DataFrame Index

- Similar to index for Series
- Immutable
- Can be shared with multiple structures (DataFrames or Series)
- `in` operator works with: 'Ohio' in df.index
### Index methods and properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Concatenate with additional Index objects, producing a new Index</td>
</tr>
<tr>
<td>diff</td>
<td>Compute set difference as an Index</td>
</tr>
<tr>
<td>intersection</td>
<td>Compute set intersection</td>
</tr>
<tr>
<td>union</td>
<td>Compute set union</td>
</tr>
<tr>
<td>isin</td>
<td>Compute boolean array indicating whether each value is contained in the passed collection</td>
</tr>
<tr>
<td>delete</td>
<td>Compute new Index with element at index $i$ deleted</td>
</tr>
<tr>
<td>drop</td>
<td>Compute new index by deleting passed values</td>
</tr>
<tr>
<td>insert</td>
<td>Compute new Index by inserting element at index $i$</td>
</tr>
<tr>
<td>is_monotonic</td>
<td>Returns True if each element is greater than or equal to the previous element</td>
</tr>
<tr>
<td>is_unique</td>
<td>Returns True if the Index has no duplicate values</td>
</tr>
<tr>
<td>unique</td>
<td>Compute the array of unique values in the Index</td>
</tr>
</tbody>
</table>

---

In [78]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [79]: obj
Out[79]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>4.5</td>
</tr>
<tr>
<td>b</td>
<td>7.2</td>
</tr>
<tr>
<td>a</td>
<td>-5.3</td>
</tr>
<tr>
<td>c</td>
<td>3.6</td>
</tr>
<tr>
<td>dtype: float64</td>
<td></td>
</tr>
</tbody>
</table>

---

In [80]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [81]: obj2
Out[81]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>None</td>
</tr>
<tr>
<td>b</td>
<td>None</td>
</tr>
<tr>
<td>c</td>
<td>None</td>
</tr>
<tr>
<td>d</td>
<td>None</td>
</tr>
<tr>
<td>e</td>
<td>None</td>
</tr>
<tr>
<td>dtype: float64</td>
<td></td>
</tr>
</tbody>
</table>

---

[W. McKinney, Python for Data Analysis]
Reindexing

- `reindex` creates a new object with the data conformed to new index
- `obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])`

- Missing values: handle with kwargs
  - `fill_value`: fill any missing value with a specific value
  - `method='ffill'`: fill values forward
  - `method='bfill'`: fill values backward

- Data Frames:
  - reindex rows as with series
  - reindex columns using columns kwarg
Dropping entries

- Can drop one or more entries
- Series:
  - `new_obj = obj.drop('c')`
  - `new_obj = obj.drop(['d', 'c'])`
- Data Frames:
  - `axis` keyword defines which axis to drop (default 0)
    - `axis==0` → rows, `axis==1` → columns
    - `axis = '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 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

• \( s = \text{Series}(\text{np.arange}(4.), \text{index}=[4, 3, 2, 1]) \)
• \( s[3] \)
• \( \text{s.loc}[3] \)
• \( \text{s.iloc}[3] \)
• \( \text{s2} = \text{pd.Series}(\text{np.arange}(4), \text{index}=['a', 'b', 'c', 'd']) \)
• \( \text{s2}[3] \)
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 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]:
Ohio        35000
Oregon      16000
Texas       71000
Utah        5000
California 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:

  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 0
d 1
e 2

  In [150]: frame - series
  Out[150]:
  Utah    1
  Ohio    2
  Texas   2
  Oregon  9

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

• 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.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.sort_index(by='b')
Out[179]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

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

In [181]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                       'c': [-2, 5, 8, -2.5]})
In [182]: frame.sort_index(axis=1)
Out[182]:
   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
```

**Method Description**

- **average**: Default: assign the average rank to each entry in the equal group.
- **min**: Use the minimum rank for the whole group.
- **max**: Use the maximum rank for the whole group.
- **first**: Assign ranks in the order the values appear in the data.
## 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
```

```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
```

### Method Description
- **count**: Number of non-NA values
- **describe**: Compute set of summary statistics for Series or each DataFrame column
- **min, max**: Compute minimum and maximum values
- **argmin, argmax**: Compute index locations (integers) at which minimum or maximum value obtained, respectively
- **idxmin, idxmax**: Compute index values at which minimum or maximum value obtained, respectively
- **quantile**: Compute sample quantile ranging from 0 to 1
- **sum**: Sum of values
- **mean**: Mean of values
- **median**: Arithmetic median (50% quantile) of values
- **mad**: Mean absolute deviation from mean value
- **var**: Sample variance of values
- **std**: Sample standard deviation of values
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>
<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>

[W. McKinney, Python for Data Analysis]
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>notnull</td>
<td>Negation of isnull.</td>
</tr>
</tbody>
</table>

Argument Description

**dropna**
Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.

**fillna**
Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.

**isnull**
Return like-type object containing boolean values indicating which values are missing / NA.

**notnull**
Negation of isnull.

[W. McKinney, Python for Data Analysis]
Back to the Food Inspections Example
Reading & Writing Data
Reading Data in Python

• Use the `open()` method to open a file for reading
  ```python
  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 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:
  - `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!
  - `with open('output.txt', 'w') as f:
      for k, v in counts.items():
          f.write(k + ': ' + v + '\n')`
• Without `with`, we need `f.close()`