DSC 201: Data Analysis & Visualization

Data Frames

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 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})
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, filtering, and NumPy operations 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
  Utah           NaN
  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]
Assignment 3

- Link
- Hurricane data, take 3
- Redo parts of A1 & A2 using pandas
- Part 5 shows how pandas can connect to altair
- May need to read ahead, but have tried to point to specific documentation for most of the concepts
- Due Thursday, Nov. 1
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
# 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>
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

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

• Same as with numpy arrays but allows use of column-based criteria
  - data[data < 5] = 0
  - data[data['three'] > 5]
  - data < 5 creates a boolean data frame that 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

<table>
<thead>
<tr>
<th>state</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ohio</td>
<td>35000</td>
</tr>
<tr>
<td>Oregon</td>
<td>16000</td>
</tr>
<tr>
<td>Texas</td>
<td>71000</td>
</tr>
<tr>
<td>Utah</td>
<td>5000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>NaN</td>
</tr>
<tr>
<td>Ohio</td>
<td>35000</td>
</tr>
<tr>
<td>Oregon</td>
<td>16000</td>
</tr>
<tr>
<td>Texas</td>
<td>71000</td>
</tr>
<tr>
<td>Utah</td>
<td>5000</td>
</tr>
</tbody>
</table>

dtype: int64

<table>
<thead>
<tr>
<th>state</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>NaN</td>
</tr>
<tr>
<td>Ohio</td>
<td>70000</td>
</tr>
<tr>
<td>Oregon</td>
<td>32000</td>
</tr>
<tr>
<td>Texas</td>
<td>142000</td>
</tr>
<tr>
<td>Utah</td>
<td>NaN</td>
</tr>
</tbody>
</table>

dtype: float64

- also have \( \text{add}, \text{subtract}, \ldots \) that allow \text{fill_value} argument
- \text{obj3.add(obj4, fill_value=0)}
Arithmetic between DataFrames and Series

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

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

In [149]: series
Out[149]:
       b  0
Utah  1
Ohio  4
Texas 7
Oregon 10

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

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

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

In [155]: series3
Out[155]:
       b  0
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
```

By default, arithmetic between DataFrame and Series matches the index of the Series
- In this case we mean to match
- Method for addition (+)
- Method for subtraction (-)
- Method for multiplication (*)
- Method for division (/)
- As with NumPy arrays, arithmetic between DataFrame and Series is well-defined. First,
- Use one of the arithmetic methods. For example:
- Table 5-7. Flexible arithmetic methods
Examples

• See ch05.ipynb