DSC 201: Data Analysis & Visualization

Data Frames

Dr. David Koop
List, Array, or Series?

\[
[[1, 2, 3], [4, 5, 6]]
\]
List, Array, or Series?

\[
\begin{bmatrix}
1, 2, 3, \\
4, 5, 6
\end{bmatrix}
\]
List, Array, or Series?

Which should I use to store values that could be either numbers or strings?
Which should I use to store values that could be either numbers or strings?
List, Array, or Series?

Which should I use to store values for **fast** access and processing?
List, **Array**, or **Series**?

Which should I use to store values for **fast** access and processing?
List, Array, or Series?

Which should I use if I want to associate a label with each data value?
List, Array, or Series?

Which should I use if I want to associate a label with each data value?
List, Array, or Series?

What is \( a \) if I can write \( a[1:3] = 5 \)?
List, Array, or Series?

What is $a$ if I can write $a[1:3] = 5$?
Assignment 2

• [http://www.cis.umassd.edu/~dkoop/dsc201/assignment2.html](http://www.cis.umassd.edu/~dkoop/dsc201/assignment2.html)
• Analyze data on refugees
• Requires cleaning up the data
• Data stored in dictionaries
• Questions?
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 = Series([7,14,-2,1])`
- Basically two arrays: `obj.values` and `obj.index`
- Can specify the index explicitly and use strings
  - `obj2 = Series([4, 7, -5, 3],
  \[\text{index=\['d', 'b', 'a', 'c'\]}\])`
- Could think of a fixed-length, ordered dictionary
- Can create from a dictionary
  - `obj3 = 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
  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]
DataFrame Access and Manipulation

- `df.values` → 2D NumPy array
- `df[<column>]` → Series
- `df[<column>] = <scalar>` # all values set to same value
- `df[<column>] = <array>` # values set in order
- `df[<column>] = <series>` # values set according to matches # between df and series indexes
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
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>

[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
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)
  - ix: get data by either label or position (AVOID!), checks label first
  - For scalars, also have at and iat
Indexing

• `s = Series(np.arange(4.), index=[4,3,2,1])`
• `s[3]`
• `s.ix[3]`
• `s.loc[3]`
• `s.iloc[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

```
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
```

- 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]:
  b   d   e
  Utah 0  0  0
  Ohio 3  3  3
  Texas 6  6  6
  Oregon 9  9  9

- To broadcast over columns, use methods (.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]:
  Name: d, dtype: float64

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

• Sort by index (lexicographical):

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

• DataFrame sorting:

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 [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   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>

Axis indexes with duplicate values

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
```

See Table 5-8 for a list of tie-breaking methods available.
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
```

See Table 5-10 for a full list of summary statistics and related methods.

Table 5-10. Descriptive and summary 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>
</tbody>
</table>
### 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>

See Table 5-10 for a full list of summary statistics and related methods.

Table 5-10. Descriptive and summary 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>

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:

```python
In [208]: returns = price.pct_change()
In [209]: returns.tail()
```

#### Correlation and Covariance

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:

```python
In [208]: returns = price.pct_change()
In [209]: returns.tail()
```

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:

```python
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'])

• value_counts returns a Series with index frequencies:
  - `s.value_counts()`  # Series({'c': 3,'a': 3,'b': 2,'d': 1})
# Handling Missing Data

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>notnull</td>
<td>Negation of isnull.</td>
</tr>
</tbody>
</table>