Programming Principles in Python (CSCI 503)

Data

Dr. David Koop
CPU-Bound vs. I/O-Bound

CPU Processing
- Compute Problem 1
- Compute Problem 2

I/O Waiting
- Request 1
- Request 2
- Request 3

CPU Processing

Time

[J. Anderson]
Threading

• Threading address the I/O waits by letting separate pieces of a program run at the same time
• Threads run in the same process
• Threads share the same memory (and global variables)
• Operating system schedules threads; it can manage when each thread runs, e.g. round-robin scheduling
• When blocking for I/O, other threads can run
Python Threading Speed

- If I/O bound, threads work great because time spent waiting can now be used by other threads
- Threads do not run simultaneously in standard Python, i.e. they cannot take advantage of multiple cores
- Use threads when code is I/O bound, otherwise no real speed-up plus some overhead for using threads
Python and the GIL

• Solution for reference counting (used for garbage collection)
• Could add locking to every value/data structure, but with multiple locks comes possible **deadlock**
• Python instead has a Global Interpreter Lock (GIL) that must be acquired to execute any Python code
• This effectively makes Python single-threaded (faster execution)
• Python requires threads to give up GIL after certain amount of time
• Python 3 improved allocation of GIL to threads by not allowing a single CPU-bound thread to hog it
Multiprocessing

• Multiple processes do not need to share the same memory, interact less
• Python makes the difference between processes and threads minimal in most cases
• Big win: can take advantage of multiple cores!

```python
import multiprocessing
with multiprocessing.Pool() as pool:
    pool.map(printer, range(5))
```

• **Warning**: known issues with running this in the notebook, use in scripts or look for alternate possibilities/library

• Set `__spec__ = None` to use the `%run` command in the notebook with a multiprocessing script
Multiprocessing using concurrent.futures

- `import concurrent.futures`
  `import multiprocessing as mp`
  `import time`

  
  ```python
  def dummy(num):
      time.sleep(5)
      return num ** 2
  
  with concurrent.futures.ProcessPoolExecutor(max_workers=5,
                                            mp_context=mp.get_context('fork')) as executor:
      results = executor.map(dummy, range(10))
  ```

- `mp.get_context('fork')` changes from 'spawn' used by default in MacOS, works in notebook
When to use threading or multiprocessing?

- If your code has a lot of I/O or Network usage:
  - Multithreading is your best bet because of its low overhead
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)
Assignment 7

- Downloading and unarchiving files
- File system manipulation
- Threading
- Basic Data Manipulation
- Due Friday
pandas

- Contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python
- Built on top of NumPy
- Built with the following requirements:
  - Data structures with labeled axes (aligning data)
  - Support time series data
  - Do arithmetic operations that include metadata (labels)
  - Handle missing data
  - Add merge and relational operations
Pandas Code Conventions

• Universal:
  - import pandas as pd

• Also used:
  - from pandas import Series, DataFrame
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**
- **Arithmetic operations align:**

```
In [28]: obj3
Out[28]:
Ohio    35000
Oregon  16000
Texas   71000
Utah    5000
Name: population, dtype: int64

In [29]: obj4
Out[29]:
California  NaN
Ohio        35000
Oregon      16000
Texas       71000
Name: state, dtype: float64

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

[W. McKinney, Python for Data Analysis]
Data Frame

- A dictionary of Series (labels for each series)
- A spreadsheet with row keys (the index) and column headers
- Has an index shared with each series
- Allows easy reference to any cell
- \( \text{df} = \text{DataFrame}({'\text{state}': ['Ohio', 'Ohio', 'Ohio', 'Nevada'], '\text{year}': [2000, 2001, 2002, 2001], '\text{pop}': [1.5, 1.7, 3.6, 2.4]}) \)

- 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
- Can choose new index column(s) with `set_index()`
- `reindex` creates a new object with the data conformed to new index
  - `obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])`
  - can fill in missing values in different ways
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 = 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]`
Indexing in Data Frames

- `df['col1']` # a column
- `df.loc['Ohio']` # a row
- `df.loc['Ohio','col1']` # the cell
- Multiple columns use a list inside the brackets
  - `df[['col1','col2']]`
  - Can nest these in `loc`, too: `df.loc['Ohio',['col1','col2']]`
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

- Multiple criteria, use `&`, `|`, and `~`; remember parentheses!
  - `data[(data['three'] > 5) & (data['two'] < 10)]`
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]:
Utah       5000        Texas         71000
Oregon    16000        Ohio          35000
Ohio      35000        California      NaN
dtype: int64           dtype: float64

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]:
Utah       5000        Texas         71000
Oregon    16000        Ohio          35000
Ohio      35000        California      NaN
dtype: int64           dtype: float64

```

- also have `.add`, `.subtract`, ... that allow `fill_value` argument

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

Out[35]:
Jeff    -5
Steve    7
Bob      4
Name: population, dtype: float64

```

- `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  |
    |-----|----|----|
    | 0  | 1  | 2  |
    | 3  | 4  | 5  |
    | 6  | 7  | 8  |
    | 9  | 10 | 11 |

  - In [149]: series
    Out[149]:
    | b  | d  |
    |-----|----|
    | 0   | 1   |
    | 4   | 7   |

  - In [150]: frame - series
    Out[150]:
    | b   | d   | e   |
    |-----|-----|-----|
    | 0   | 1   | 2   |
    | 3   | 3   | 3   |
    | 6   | 6   | 6   |
    | 9   | 9   | 9   |

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

  - In [154]: frame
    Out[154]:
    | b  | d  | e  |
    |-----|----|----|
    | 0  | 1  | 2  |
    | 3  | 4  | 5  |
    | 6  | 7  | 8  |
    | 9  | 10 | 11 |

  - In [155]: series3
    Out[155]:
    | b  | d  | e  |
    |-----|----|----|
    | Utah | 1  |
    | Texas| 7  |
    | Oregon| 10 |

  - In [156]: frame.sub(series3, axis=0)
    Out[156]:
    | b   | d   | e   |
    |-----|-----|-----|
    | 0   | 1   | 2   |
    | -1  | 0   | 1   |
    | -1  | 0   | 1   |
    | -1  | 0   | 1   |

- Operations between DataFrame and Series are similar:

  - `add` for addition
  - `subtract` for subtraction
  - `mul` for multiplication
  - `div` for division

- By default, arithmetic between DataFrame and Series matches the index of the Series and is explained in more detail in Chapter 12: Getting Started with pandas.
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]:
d  a  b  c
one    4  5  6  7
three  0  1  2  3

Out[172]:
d  c  b  a
three  0  3  2  1
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)
```

```
Out[173]:
d  c  b  a
three  0  3  2  1
one    5  6  7  4
```

• 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
```

D. Koop, CSCI 503, Spring 2021
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:**

  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:**

- **Works on data frames, too**

```
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
```
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
```

Another type of method is neither a reduction nor an accumulation. **describe** is one such example, producing multiple summary statistics in one shot:

```
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><strong>count</strong></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><strong>min, max</strong></td>
<td>Compute minimum and maximum values</td>
</tr>
<tr>
<td><strong>argmin, argmax</strong></td>
<td>Compute index locations (integers) at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td><strong>idxmin, idxmax</strong></td>
<td>Compute index values at which minimum or maximum value obtained, respectively</td>
</tr>
<tr>
<td><strong>quantile</strong></td>
<td>Compute sample quantile ranging from 0 to 1</td>
</tr>
<tr>
<td><strong>sum</strong></td>
<td>Sum of values</td>
</tr>
<tr>
<td><strong>mean</strong></td>
<td>Mean of values</td>
</tr>
<tr>
<td><strong>median</strong></td>
<td>Arithmetic median (50% quantile) of values</td>
</tr>
<tr>
<td><strong>mad</strong></td>
<td>Mean absolute deviation from mean value</td>
</tr>
<tr>
<td><strong>var</strong></td>
<td>Sample variance of values</td>
</tr>
<tr>
<td><strong>std</strong></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>

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]:

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>3.000000</td>
<td>2.000000</td>
</tr>
<tr>
<td>mean</td>
<td>3.083333</td>
<td>-2.900000</td>
</tr>
<tr>
<td>std</td>
<td>3.493685</td>
<td>2.262742</td>
</tr>
<tr>
<td>min</td>
<td>0.750000</td>
<td>-4.500000</td>
</tr>
<tr>
<td>25%</td>
<td>1.075000</td>
<td>-3.700000</td>
</tr>
<tr>
<td>50%</td>
<td>1.400000</td>
<td>-2.900000</td>
</tr>
<tr>
<td>75%</td>
<td>4.250000</td>
<td>-2.100000</td>
</tr>
<tr>
<td>max</td>
<td>7.100000</td>
<td>-1.300000</td>
</tr>
</tbody>
</table>

On non-numeric data, describe produces alternate summary statistics:

In [205]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [206]: obj.describe()
Out[206]:

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>unique</th>
<th>top</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>16</td>
<td>3</td>
<td>a</td>
<td>8</td>
</tr>
<tr>
<td>dtype</td>
<td>object</td>
<td></td>
<td></td>
<td></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>

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:

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

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>GOOG</th>
<th>IBM</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014-07-07</td>
<td>0.020632</td>
<td>-0.004241</td>
<td>-0.002599</td>
<td>0.004545</td>
</tr>
<tr>
<td>2014-07-08</td>
<td>-0.006460</td>
<td>-0.019167</td>
<td>-0.004361</td>
<td>-0.005001</td>
</tr>
<tr>
<td>2014-07-09</td>
<td>0.000420</td>
<td>0.008738</td>
<td>0.006410</td>
<td>-0.002633</td>
</tr>
<tr>
<td>2014-07-10</td>
<td>-0.003669</td>
<td>-0.008645</td>
<td>-0.003821</td>
<td>0.000480</td>
</tr>
<tr>
<td>2014-07-11</td>
<td>0.001894</td>
<td>0.014148</td>
<td>0.001598</td>
<td>0.009595</td>
</tr>
</tbody>
</table>

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'])`

- Also **nunique()** to count number of unique entries

- **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>
## 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]
**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
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
inplace

- Generally, when we modify a data frame, we reassign:
  - `rdf = df.reset_index()`
  - This is usually very **efficient**
  - Allows for method chaining

- There are versions where you can do this "inplace":
  - `df.reset_index(inplace=True)`
  - This means **no reassignment**, but it isn't usually any faster nor better
  - Sometimes still creates a copy
  - Will likely be **deprecated**
Documentation

- pandas documentation is pretty good
- Lots of recipes on stackoverflow for particular data manipulations/queries
Food Inspections Example