Pandas

Open-source library of data analysis tools
Low-level ops implemented in Cython (C+Python=Cython, often faster)
Database-like structures, largely similar to those available in R
Optimized for most common operations
E.g., vectorized operations, operations on rows of a table

From the documentation: pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.
Installing pandas

Anaconda:
conda install pandas

Using pip:
pip install pandas

From binary (not recommended):

http://pypi.python.org/pypi/pandas

**Warning:** a few recent updates to pandas have been API-breaking changes, meaning they changed one or more functions (e.g., changed the number of arguments, their default values, or other behaviors). This shouldn’t be a problem for us, but you may as well check that you have the most recent version installed.
Basic Data Structures

Series: represents a one-dimensional **labeled** array
  Labeled just means that there is an index into the array
  Support vectorized operations

Dataframe: table of rows, with labeled columns
  Like a spreadsheet or an R data frame
  Support **numpy** ufuncs (provided data are numeric)
pandas Series

By default, indices are integers, starting from 0, just like you’re used to.

But we can specify a different set of indices if we so choose.

Can create a pandas Series from any array-like structure (e.g., numpy array, Python list, dict).

Pandas tries to infer this data type automatically.

Warning: providing too few or too many indices is a ValueError.
Can create a series from a dictionary. Keys become indices.

Index ‘cthulu’ doesn’t appear in the dictionary, so pandas assigns it NaN, the standard “missing data” symbol.
Indexing works like you’re used to and supports slices, but **not** negative indexing.

This object has type `np.int64`

This object is another `pandas Series`. 

```python
s = pd.Series([2, 3, 5, 7, 11])
s[0]
s[1:3]
s[-1]
```

```
KeyError
```
Caution: indices need not be unique in `pandas Series`. This will only cause an error if/when you perform an operation that requires unique indices.
Series objects are like `np.ndarray` objects, so they support all the same kinds of slice operations, but note that the indices come along with the slices.

Series objects even support most `numpy` functions that act on arrays.
Series objects are **dict-like**, in that we can access and update entries via their keys.

**Not shown:** Series also support the `in` operator: `x in s` checks if `x` appears as an index of Series `s`. Series also supports the dictionary `get` method.

Like a dictionary, accessing a non-existent key is a `KeyError`.

**Note:** I cropped out a bunch of the error message, but you get the idea.
Entries of a Series can be of (almost) any type, and they may be mixed (e.g., some floats, some ints, some strings, etc), but they **can not** be sequences.

Series support universal functions, so long as all their entries support operations.

Series operations require that keys be shared. Missing values become NaN by default.

To reiterate, Series objects support most numpy ufuncs. For example, `np.sqrt(s)` is valid, so long as all entries are positive.
Series have an optional `name` attribute. After it is set, `name` attribute can be changed with `rename` method.

**Note:** this returns a new Series. It does not change `s.name`.

This will become especially useful when we start talking about DataFrames, because these name attributes will be column names.
Mapping and linking Series values

Series `map` method works analogously to Python’s `map` function. Takes a function and applies it to every entry.
Mapping and linking Series values

```
s = pd.Series(['fruit', 'animal', 'animal', 'fruit', 'fruit'],
              index=['apple', 'cat', 'goat', 'banana', 'kiwi'])

apple          fruit
cat            animal
goat           animal
banana         fruit
kiwi           fruit
dtype: object
```

```
t = pd.Series({'fruit':0, 'animal':1})

s.map(t)
```

Series `map` also allows us to change values based on another Series. Here, we're changing the fruit/animal category labels to binary labels.
**pandas DataFrames**

Fundamental unit of **pandas**
- Analogous to R data frame

2-dimensional structure (i.e., rows and columns)
- Columns, of potentially different types
  - Think: spreadsheet (or, better, database, but we haven’t learned those, yet)

Can be created from many different objects
- Dict of {ndarrays, Python lists, dicts, Series}
- 2-dimensional ndarray
- Series
Creating a DataFrame from a dictionary, the keys become the column names. Values become the rows of the dictionary.

```
1 d = {'A':pd.Series([1,2,3], index=['cat', 'dog', 'bird']),
2    'B': {'cat':3.14, 'dog':2.718, 'bird':1.618, 'goat':0.5772}}
3 df = pd.DataFrame(d)
4 df
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bird</td>
<td>3.0</td>
<td>1.6180</td>
</tr>
<tr>
<td>cat</td>
<td>1.0</td>
<td>3.1400</td>
</tr>
<tr>
<td>dog</td>
<td>2.0</td>
<td>2.7180</td>
</tr>
<tr>
<td>goat</td>
<td>NaN</td>
<td>0.5772</td>
</tr>
</tbody>
</table>

Rows that are unspecified for a given column receive \texttt{NaN}.

Note: in the code above, we specified the two columns differently. One was specified as a Series object, and the other as a dictionary. This is just to make the point that there is flexibility in how you construct your DataFrame. More options: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html
**pandas DataFrames: creating DataFrames**

Dictionary has 4 keys, so 4 columns.

```
import pandas as pd

                            index=['Ford', 'Hoover', 'Wilson', 'Obama']),
     'PhD': {'Wilson': 'Johns Hopkins'},
     'JD': {'Ford': 'Yale', 'Obama': 'Harvard'},
     'Terms': pd.Series([1,1,2,2], index=['Ford', 'Hoover', 'Wilson', 'Obama'])}

presidents = pd.DataFrame(d)
presidents
```

**Note:** Dictionary includes both text and numeric columns

By default, rows and columns are ordered alphabetically.
Row and column names accessible as the `index` and `column` attributes, respectively, of the DataFrame.

Both are returned as `pandas` `Index` objects.
DataFrames: accessing/adding columns

DataFrame acts like a dictionary whose keys are column names, values are Series.

Like a dictionary, we can create new key-value pairs.

Note: technically, this isn’t quite correct, because Ford did not serve a full term.
https://en.wikipedia.org/wiki/Gerald_Ford
Since the row labels are ordered, we can specify a new column directly from a Python list, numpy array, etc. without having to specify indices.

Note: by default, new column are inserted at the end. See the `insert` method to change this behavior: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.insert.html
### pandas DataFrames: accessing/adding columns

Scalars are broadcast across the rows.

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Nobels</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
<td>0</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
<td>0</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>2</td>
<td>Columbia</td>
<td>1</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>2</td>
<td>Princeton</td>
<td>1</td>
</tr>
</tbody>
</table>

```python
1 presidents['Fields Medals'] = 0
2 presidents
```

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Nobels</th>
<th>Years</th>
<th>Fields Medals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>2</td>
<td>Columbia</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>2</td>
<td>Princeton</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
Deleting columns

Delete columns identically to deleting keys from a dictionary. One can use the `del` keyword, or pop a key.

```python
del presidents["Years"]
presidents
```

```
JD  PhD  Terms  Undergrad  Nobels  Years  Fields Medals
Ford Yale NaN  1  UMich  0  4  0
Hoover NaN NaN  1  Stanford  0  4  0
Obama Harvard NaN  2  Columbia  1  8  0
Wilson NaN Johns Hopkins  2  Princeton  1  8  0
```

```python
fields = presidents.pop("Fields Medals")
```
Indexing and selection

`df.loc` selects rows by their labels.

`df.iloc` selects rows by their integer labels (starting from 0).
Indexing and selection

Select columns by their names.
Indexing and selection

Select columns by their numerical indices (again 0-indexed). This supports slices.

Note: one can also select slices with lists of column names, e.g., `presidents[['JD','PhD']].`
Indexing and selection

Select columns by Boolean expression.
Indexing and selection

These expressions return Series objects.

- `presidents['JD']`
- `presidents.loc['Obama']`
- `presidents.iloc[1]`
- `presidents[presidents['Terms']<2]`
Indexing and selection

These expressions return Series objects.

These expressions return DataFrames.

Arithmetic with DataFrames

```python
1 df1 = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])
2 df2 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])
3 df1+df2
```

**pandas** tries to align the DataFrames as best it can, filling in non-alignable entries with NaN.

In this example, rows 0 through 4 and columns A through C exist in both DataFrames, so these entries can be successfully added. All other entries get NaN, because $x + \text{NaN} = \text{NaN}$.
By default, Series are aligned to DataFrames via row-wise broadcasting. `df.iloc[0]` is a Series representing the 0-th row of `df`. When we try to subtract it from `df`, pandas forces dimensions to agree by broadcasting the operation across all rows of `df`.
Scalar addition and multiplication works in the obvious way. DataFrames also support scalar division, exponentiation… Basically every numpy ufunc.

DataFrames also support entrywise Boolean operations.
Arithmetic with DataFrames

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1.331635</td>
<td>-0.500870</td>
</tr>
<tr>
<td>1</td>
<td>1.111157</td>
<td>0.293138</td>
</tr>
<tr>
<td>2</td>
<td>-0.669850</td>
<td>0.456863</td>
</tr>
<tr>
<td>3</td>
<td>0.216643</td>
<td>-0.636942</td>
</tr>
</tbody>
</table>

**pandas** DataFrames support *numpy*-like any and all methods.

Just like *numpy*, direct Boolean operations are not supported.

```python
In [1]: df = pd.DataFrame({
                    'A': [-1.331635, 1.111157, -0.669850, 0.216643],
                    'B': [-0.500870, 0.293138, 0.456863, -0.636942]
                })

In [2]: (df > 0).any()
Out[2]:
A    True
B    True
dtype: bool

In [3]: (df > 0).all()
Out[3]:
A    False
B    False
dtype: bool
```

**ValueError**: The truth value of a DataFrame is ambiguous.

Use `a.empty`, `a.bool()`, `a.item()`, `a.any()` or `a.all()`.
Arithmetic with DataFrames

The `values` attribute stores the entries of the table in a numpy array. This is occasionally useful when you want to stop dragging the extra information around and just work with the numbers in the table.
Arithmetic with DataFrames

DataFrames support entrywise multiplication. The `T` attribute is the transpose of the DataFrame.

DataFrames also support matrix multiplication via the `numpy-like dot` method. The DataFrame dimensions must be conformal, of course.

Note: Series also support a `dot` method, so you can compute inner products.
Removing NaNs

**DataFrame** `dropna` method removes rows or columns that contain NaNs.

how='any' will remove all rows/columns that contain even one NaN. how='all' removes rows/columns that have all entries NaN.

**axis** argument controls whether we act on rows, columns, etc.
### Reading/writing files

Pandas supports read/write for a wide range of different file formats. This flexibility is a major advantage of pandas.

<table>
<thead>
<tr>
<th>Format Type</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>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>binary</td>
<td>MS Excel</td>
<td>read_excel</td>
<td>to_excel</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>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>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 Big Query</td>
<td>read_gbq</td>
<td></td>
</tr>
</tbody>
</table>

# Reading/writing files

Pandas supports read/write for a wide range of different file formats. This flexibility is a major advantage of **pandas**.

<table>
<thead>
<tr>
<th>Format Type</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>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>binary</td>
<td>MS Excel</td>
<td>read_excel</td>
<td>to_excel</td>
</tr>
<tr>
<td>binary</td>
<td>Msgpack</td>
<td>read_hdf</td>
<td>to_hdf</td>
</tr>
<tr>
<td>binary</td>
<td>Stata</td>
<td>read_feather</td>
<td>to_feather</td>
</tr>
<tr>
<td>binary</td>
<td>SAS</td>
<td>read_parquet</td>
<td>to_parquet</td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td>read_msgpack</td>
<td>to_msgpack</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>read_pickle</td>
<td>to_pickle</td>
</tr>
<tr>
<td>SQL</td>
<td>Google Big Query</td>
<td>read_gquery</td>
<td>to_gbq</td>
</tr>
</tbody>
</table>

**pandas** file I/O is largely similar to R `read.table` and similar functions, so I’ll leave it to you to read the **pandas** documentation as needed.

Table credit: [https://pandas.pydata.org/pandas-docs/stable/io.html](https://pandas.pydata.org/pandas-docs/stable/io.html)
Summarizing DataFrames

`pd.read_csv()` reads a comma-separated file into a DataFrame.

`info()` method prints summary data about the DataFrame. Number of rows, column names and their types, etc.

*Note:* there is a separate `to_string()` method that generates a string representing the DataFrame in tabular form, but this usually doesn’t display well if you have many columns.
Summarizing DataFrames

The `head()` method displays just the first few rows of the DataFrame (5 by default; change this by supplying an argument). The `tail()` method displays the last few rows.

| id    | year | stint | team   | lg | g    | ab  | r   | h   | X2b | ... | rbi | sb  | cs  | bb  | so  | ibb | hbp  | sh  | sf  | gidp |
|-------|------|-------|--------|----|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 4     | 1871 | 1     | RC1    | NaN| 25   | 120 | 29  | 39  | 11  | ... | 16.0| 6.0 | 2.0 | 2   | 1.0 | NaN | NaN  | NaN | NaN | NaN  |
| 44    | 1871 | 1     | WS3    | NaN| 32   | 162 | 45  | 45  | 9   | ... | 29.0| 8.0 | 0.0 | 4   | 0.0 | NaN | NaN  | NaN | NaN | NaN  |
| 68    | 1871 | 1     | FW1    | NaN| 19   | 89  | 15  | 24  | 3   | ... | 10.0| 2.0 | 1.0 | 2   | 0.0 | NaN | NaN  | NaN | NaN | NaN  |
| 99    | 1871 | 1     | NY2    | NaN| 33   | 161 | 35  | 58  | 5   | ... | 34.0| 4.0 | 2.0 | 3   | 0.0 | NaN | NaN  | NaN | NaN | NaN  |
| 102   | 1871 | 1     | CL1    | NaN| 29   | 128 | 35  | 45  | 3   | ... | 23.0| 3.0 | 1.0 | 1   | 0.0 | NaN | NaN  | NaN | NaN | NaN  |

**5 rows x 22 columns**

**Note:** R and pandas both supply `head/tail` functions, named after UNIX/Linux commands that displays the first/last lines of a file.
Comparing DataFrames

These two DataFrames *ought* to be equivalent...

...but they aren’t.
Comparing DataFrames

These two DataFrames *ought* to be equivalent...

...but they aren’t.

The problem comes from the fact that NaNs are not equal to one another.

**Solution:** DataFrames have a separate `equals()` method for checking the kind of equality that we meant above.
Comparing DataFrames

There is a solid design principle behind this. If there are NaNs in our data, we want to err on the side of being overly careful about what operations we perform on them. We see similar ideas in numpy and in R.

**Solution:** DataFrames have a separate `equals()` method for checking the kind of equality that we meant above.
Pandas Panels

Pandas has another data type, called a Panel
   Meant for representing panel data
      https://en.wikipedia.org/wiki/Panel_data

Panel is deprecated, so I am not going to teach you about it
   But you should be aware that it exists, because it is mentioned in docs

## Statistical Operations on DataFrames

Getting means of DataFrame rows/columns using numpy is possible, but tedious.

```python
np.nanmean(np.array(df.iloc[1]))
-0.50977640954057268
```

**DataFrame.mean** method is a cleaner way to do the same thing. Argument picks out which axis to take means on: rows (1) or columns (0).
Of course, DataFrames also support a bunch of related functions, that work similarly: `sum`, `min`, `max`, `std`, `var` etc. All of these functions take an optional Boolean argument `skipna`. If True, NaNs are **not included** in the computation. If False, NaNs are included (which can mean either that the computation doesn’t work at all, or changes the value only slightly). More information: https://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics

`DataFrame.mean` method is a cleaner way to do the same thing. Argument picks out which axis to take means on: rows (1) or columns (0).
Summarizing DataFrames

DataFrame.describe() is similar to the R summary() function. Non-numeric data will get statistics like counts, number of unique items, etc. If a DataFrame has mixed types (both numeric and non-numeric), the non-numeric data is excluded by default.

Details and optional arguments:
https://pandas.pydata.org/pandas-docs/stable/basics.html#summarizing-data-describe
Row- and column-wise functions: `apply()`

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.284355</td>
<td>1.073402</td>
<td>0.297575</td>
<td>NaN</td>
</tr>
<tr>
<td>-0.791592</td>
<td>0.841969</td>
<td>0.509262</td>
<td>NaN</td>
</tr>
<tr>
<td>-0.657900</td>
<td>-2.184139</td>
<td>1.635736</td>
<td>NaN</td>
</tr>
<tr>
<td>-1.897574</td>
<td>0.502787</td>
<td>-1.911790</td>
<td>NaN</td>
</tr>
<tr>
<td>0.592821</td>
<td>2.091333</td>
<td>-2.813032</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**DataFrame.apply()** takes a function and applies it to each column of the DataFrame.

```python
df.apply(np.mean)
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.293978</td>
<td>0.465070</td>
<td>-0.456450</td>
<td>NaN</td>
</tr>
</tbody>
</table>

dtype: float64

**Axis** argument is 0 by default (column-wise). Change to 1 for row-wise application.

```python
df.apply(np.mean, axis=1)
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.885111</td>
<td>0.186546</td>
<td>-0.402101</td>
<td>-1.102192</td>
<td>-0.042959</td>
<td>NaN</td>
</tr>
</tbody>
</table>

dtype: float64
Row- and column-wise functions: apply()

Numpy ufuncs take vectors and spit out vectors, so using `df.apply()` to apply a ufunc to every row or column in effect ends up applying the ufunc to every element.
Row- and column-wise functions: `apply()`

We can pass positional and keyword arguments into the function via `df.apply`. `Args` is a tuple of the positional arguments (in order), followed by the keyword arguments.

Note: "`apply()` takes an argument `raw` which is `False` by default, which converts each row or column into a Series before applying the function. When set to `True`, the passed function will instead receive an `ndarray` object, which has positive performance implications if you do not need the indexing functionality." This can be useful if your function is meant to work specifically with Series.
Aggregating data

This command creates time series data, with rows indexed by year-month-day timestamps.

Supplying a list of functions to \texttt{agg} will apply each function to each column of the DataFrame, with each function getting a row in the resulting DataFrame.

\texttt{agg} is an alias for the method \texttt{aggregate}. Both work exactly the same.
Aggregating data

`agg` can, alternatively, take a dictionary whose keys are column names, and values are functions.

Note that the values here are strings, not functions! `pandas` supports dispatch on strings. It recognizes certain strings as referring to functions. `apply` supports similar behavior.
Aggregating data

`df` contains mixed data types.

`agg` (and similarly `apply`) will only try to apply these functions on the columns of types supported by those functions.

**Note:** the `DataFrame transform` method provides generally similar functionality to the `agg` method.

`pandas` doesn’t know how to compute a mean string, so it doesn’t try.
Element-wise function application

This causes an error, because `apply` thinks that its argument should be applied to Series (i.e., columns), not to individual entries.
Element-wise function application

`applymap` works similarly to Python’s `map` function (and the Series `map` method). Applies its argument function to every entry of the DataFrame.
Here we have a function composition applied to a DataFrame. This is perfectly valid code, but pandas supports another approach.
Tablewise Function Application

The DataFrame `pipe` method is built for a pattern called **method chaining**. The `pipe` method has better support for passing additional arguments around than does the function composition to the right. This pattern using `pipe` is also more conducive to functional programming patterns.
Iterating over Series and DataFrames

Iterating over a Series gets an iterator over the values of the Series.

Iterating over a DataFrame gets an iterator over the column names.
Iterating over Series and DataFrames

```
for x in df.iteritems():
    print(x)
```

`iteritems()` method is supported by both Series and DataFrames. Returns an iterator over the key-value pairs. In the case of Series, these are (index, value) pairs. In the case of DataFrames, these are (colname, Series) pairs.
Iterating over Series and DataFrames

```
for x in df.iterrows():
    print(x)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.
Iterating over Series and DataFrames

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.

DataFrames are designed to make certain operations (mainly vectorized operations) fast. This implementation has the disadvantage that iteration over a DataFrame is slow. It is usually best to avoid iterating over the elements of a DataFrame or Series, and instead find a way to compute your quantity of interest using a vectorized operation or a map/reduce operation.