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)
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.

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.
pandas Series

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.
**pandas Series**

```
1 s = pd.Series([2, 3, 5, 7, 11], index=['a', 'a', 'a', 'a', 'a'])
2 s

a  2
a  3
a  5
a  7
a 11
dtype: int64
```

**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.

```
1 s['a']

a  2
a  3
a  5
a  7
a 11
dtype: int64
```
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.

More information on indexing:
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

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 columns of the dictionary.

```python
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>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bird</td>
<td>3.0</td>
</tr>
<tr>
<td>cat</td>
<td>1.0</td>
</tr>
<tr>
<td>dog</td>
<td>2.0</td>
</tr>
<tr>
<td>goat</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Indices that are unspecified for a given column receive NaN.

Each column may have its own indices, but the resulting DataFrame will have a row for every index (i.e., every row name) that appears.

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.

```
                              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']) }
7 presidents = pd.DataFrame(d)
7 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.
**pandas DataFrames: accessing/adding columns**

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

```python
1 presidents['PhD']
Ford    NaN
Hoover  NaN
Obama   NaN
Wilson  Johns Hopkins
Name: PhD, dtype: object

1 type(presidents['PhD'])
pandas.core.series.Series
```

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

```python
1 presidents['Years'] = 4*presidents['Terms']
2 presidents
```

**Note:** technically, this isn’t quite correct, because Ford did not serve a full term. [https://en.wikipedia.org/wiki/Gerald_Ford](https://en.wikipedia.org/wiki/Gerald_Ford)
### pandas DataFrames: accessing/adding columns

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](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.insert.html)

```
1 presidents['Nobels'] = [0,0,1,1]
2 presidents
```

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Years</th>
<th>Nobels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>UMich</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>Stanford</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>Columbia</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>Princeton</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
### pandas DataFrames: accessing/adding columns

<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>

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

Scalars are broadcast across the rows.
Deleting columns

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

```python
1 del presidents['Years']
2 presidents
```

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

```python
1 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 rows 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**

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

```
1. presidents['JD']
```

```
Ford  Yale
Hoover NaN
Obama  Harvard
Wilson NaN
Name: JD, dtype: object
```

```
1. presidents.loc['Obama']
```

```
JD        Harvard
PhD       NaN
Terms     2
Undergrad Columbia
Nobels    1
Name: Obama, dtype: object
```

```
1. presidents.iloc[1]
```

```
JD   NaN
PhD  NaN
Terms 1
Undergrad Stanford
Nobels 0
Name: Hoover, dtype: object
```

```
1. presidents[presidents['Terms']<2]
```

```
JD   PhD  Terms  Undergrad   Nobels
Ford NaN  NaN    1           UMich      0
Hoover NaN  NaN   1           Stanford  0
```

Select columns by Boolean expression.
Indexing and selection

These expressions return Series objects.

```python
presidents['JD']
```

```python
presidents.loc['Obama']
```

```python
presidents.iloc[1]
```

```python
presidents[presidents['Terms']<2]
```
Indexing and selection

These expressions return Series objects.

These expressions return DataFrames.

Arithmetic with DataFrames

```python
import pandas as pd
import numpy as np

# Create two DataFrames
df1 = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

# Add the DataFrames
result = 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 $x + \text{NaN} = \text{NaN}$. 

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.722814</td>
<td>-1.889204</td>
<td>-1.170304</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>1.370720</td>
<td>-1.033425</td>
<td>-0.719628</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>-2.281526</td>
<td>0.899515</td>
<td>-0.298246</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>-4.276271</td>
<td>-2.327304</td>
<td>-0.444528</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>-1.418512</td>
<td>0.463528</td>
<td>0.428446</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>7</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Arithmetic with DataFrames

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`. 

```python
def = pd.DataFrame(np.random.randn(4, 2), columns=['A', 'B'])
df
```

<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>

```python
df - df.iloc[0]
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>1</td>
<td>2.442791</td>
<td>0.794009</td>
</tr>
<tr>
<td>2</td>
<td>0.661785</td>
<td>0.957734</td>
</tr>
<tr>
<td>3</td>
<td>1.548277</td>
<td>-0.136072</td>
</tr>
</tbody>
</table>
Arithmetic with DataFrames

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

```
1 (df > 0).any()
A   True
B   True
dtype: bool
```

```
1 df or df
ValueError
```

```
1 (df > 0).all()
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.

<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>

```python
df.values
```

```
array([[-1.33163456, -0.50087024],
       [ 1.11115689, 0.29313846],
       [-0.66984966, 0.45686335],
       [ 0.21664278, -0.63694229]])
```
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.

axis argument controls whether we act on rows, columns, etc.

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

<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>to_gbq</td>
</tr>
</tbody>
</table>

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

#### Reading/writing files

<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_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>to_pickle</td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td>read_pickle</td>
<td>to_sql</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>read_gbq</td>
<td>to_gbq</td>
</tr>
<tr>
<td>SQL</td>
<td>Google Big Query</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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.
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

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

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 \textit{ought} to be equivalent...

...but they aren’t.

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

\textbf{Solution:} DataFrames have a separate \texttt{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.
Getting means of DataFrame rows/columns using numpy is possible, but tedious.

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

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).
Statistical Operations on DataFrames

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](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()`

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

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

```
def quadratic(x, a, b, c=1):
    return a*x**2 + b*x + c
def.apply(quadratic, args=(1, 2), c=5)
```

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.
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.

```python
df.applymap(lambda s: s.upper())
```
Tablewise Function Application

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 \texttt{pipe} method is built for a pattern called \textbf{method chaining}. The \texttt{pipe} method has better support for passing additional arguments around than does the function composition to the right. This pattern using \texttt{pipe} is also more conducive to functional programming patterns.