1 Data Structures

pandas introduces two new data structures to Python - Series and DataFrame, both of which are built on top of NumPy (this means it's fast).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
pd.set_option('max_columns', 50)
```

1.1 Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call:

```
s = Series(data, index=index)
```

Here, data can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

A Series is a one-dimensional object similar to an array, list, or column in a table. It will assign a labeled index to each item in the Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

```
# create a Series with an arbitrary list
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'])
s
```

```
0                7
1       Heisenberg
2             3.14
```

```
3      -1789710578
4    Happy Eating!
dtype: object
```

Alternatively, you can specify an index to use when creating the Series.

```
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'],
              index=['A', 'Z', 'C', 'Y', 'E'])
s
```

```
A                7
Z       Heisenberg
C             3.14
Y      -1789710578
E    Happy Eating!
dtype: object
```

The Series constructor can convert a dictonary as well, using the keys of the dictionary as its index.

```
d = {'Chicago': 1000, 'New York': 1300, 'Portland': 900, 'San Francisco': 1100,
     'Austin': 450, 'Boston': None}
cities = pd.Series(d)
cities
Out[4]:
Austin            450
Boston            NaN
Chicago          1000
New York         1300
Portland          900
San Francisco    1100
dtype: float64
```

You can use the index to select specific items from the Series ...

```
cities['Chicago']
Out[5]:
1000.0
```

```
cities[['Chicago', 'Portland', 'San Francisco']]
Out[6]:
Chicago          1000
Portland          900
San Francisco    1100
dtype: float64
```

Or you can use boolean indexing for selection.

```
cities[cities < 1000]
Out[7]:
Austin      450
Portland    900
dtype: float64
```

That last one might be a little weird, so let's make it more clear - cities ¡ 1000 returns a Series of True/False values, which we then pass to our Series cities, returning the corresponding True items.

```
less_than_1000 = cities < 1000
print less_than_1000
print '\n'
print cities[less_than_1000]
Austin            True
Boston           False
Chicago          False
New York         False
Portland          True
San Francisco    False
dtype: bool


Austin      450
Portland    900
dtype: float64
```

```
You can also change the values in a Series on the fly.

\begin{framed}
\begin{verbatim}
# changing based on the index
print 'Old value:', cities['Chicago']
cities['Chicago'] = 1400
print 'New value:', cities['Chicago']
Old value: 1000.0
New value: 1400.0

\begin{framed}
\begin{verbatim}
# changing values using boolean logic
print cities[cities < 1000]
print '\n'
cities[cities < 1000] = 750

print cities[cities < 1000]
Austin      450
Portland    900
dtype: float64


Austin      750
Portland    750
dtype: float64
```

What if you aren't sure whether an item is in the Series? You can check using idiomatic Python.

```
print 'Seattle' in cities
print 'San Francisco' in cities
False
True
```

Mathematical operations can be done using scalars and functions.

```
# divide city values by 3
cities / 3
Out[12]:
Austin           250.000000
Boston                  NaN
Chicago          466.666667
New York         433.333333
Portland         250.000000
San Francisco    366.666667
dtype: float64

\begin{framed}
\begin{verbatim}
# square city values
np.square(cities)
Out[13]:
Austin            562500
Boston               NaN
Chicago          1960000
New York         1690000
Portland          562500
San Francisco    1210000
dtype: float64
```

You can add two Series together, which returns a union of the two Series with the addition occurring on the shared index values. Values on either Series that did not have a shared index will produce a NULL/NaN (not a number).

```
print cities[['Chicago', 'New York', 'Portland']]
print'\n'
print cities[['Austin', 'New York']]
print'\n'
print cities[['Chicago', 'New York', 'Portland']] + cities[['Austin', 'New York']]
```

```
Chicago     1400
New York    1300
Portland     750
dtype: float64
```

```
Austin       750
New York    1300
dtype: float64


Austin       NaN
Chicago      NaN
New York    2600
Portland     NaN
dtype: float64
```

Notice that because Austin, Chicago, and Portland were not found in both Series, they were returned with NULL/NaN values.

NULL checking can be performed with isnull and notnull.

```
# returns a boolean series indicating which values aren't NULL
cities.notnull()
Out[15]:
Austin            True
Boston           False
Chicago           True
New York          True
Portland          True
San Francisco     True
dtype: bool
In [16]:
# use boolean logic to grab the NULL cities
print cities.isnull()
print '\n'
print cities[cities.isnull()]
Austin           False
Boston            True
Chicago          False
New York         False
Portland         False
San Francisco    False
dtype: bool


Boston   NaN
```

```
dtype: float64
```

1.2 DataFrame

A DataFrame is a tablular data structure comprised of rows and columns, akin to a spreadsheet, database table, or R's data.frame object. You can also think of a DataFrame as a group of Series objects that share an index (the column names).

For the rest of the tutorial, we'll be primarily working with DataFrames.

1.3 Panel

`Panel` is a somewhat less-used, but still important container for 3-dimensional data. The term panel data is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s. The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data and, in particular, econometric analysis of panel data. However, for the strict purposes of slicing and dicing a collection of DataFrame objects, you may find the axis names slightly arbitrary:

- items: axis 0, each item corresponds to a DataFrame contained inside
- major_axis: axis 1, it is the index (rows) of each of the DataFrames
- minor_axis: axis 2, it is the columns of each of the DataFrames