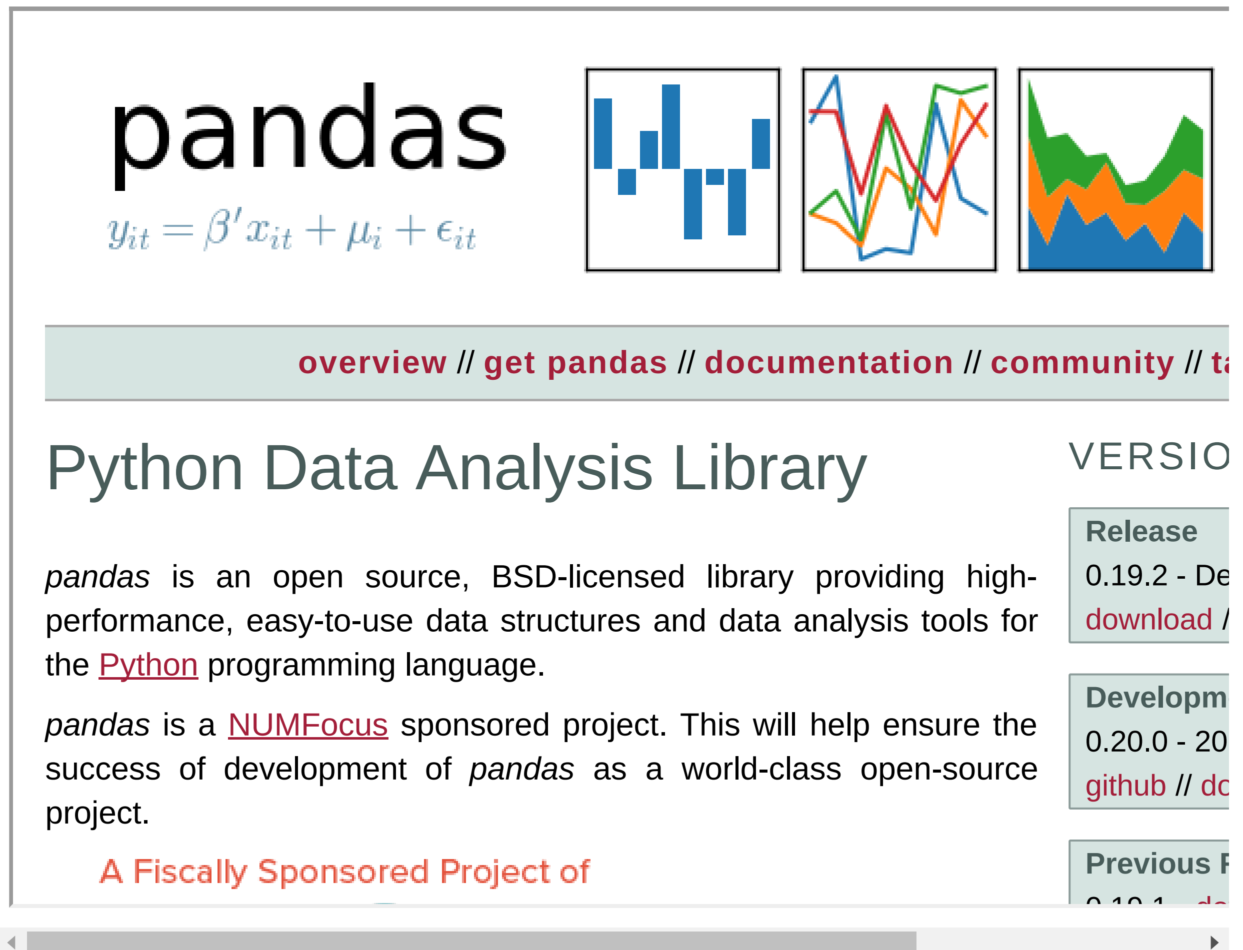
What is pandas?

- **Pandas**: **Pan**el **da**ta **s**ystem
- Python data analysis library, built on top of `numpy`
- Open Sourced by AQR Capital Management, LLC in late 2009
- 30.000 lines of tested Python/Cython code
- More details from PyData.org:
  - (Pandas is a) *high-level building block for doing practical, real world data analysis in Python*
  - *Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language*
  - (Pandas) *provides fast, flexible, and expressive data structures designed to make working with **relational** or **labeled** data both easy and intuitive*

In [1]:
```
from IPython.core.display import HTML
HTML("<iframe src=http://pandas.pydata.org width=800 height=450></iframe
```

Out[1]:



Data structures

Pandas provides two primary data structures:

- `Series` is built for 1-dimensional data series

- `DataFrame` is built for 2-dimensional collections of tabular data

These data structures differ and are related as follows:

- **Series**: 1-dimensional array of **homogeneous** data
- **DataFrame**: 2-dimensional table of **heterogeneous** data, composed of multiple **Series**

note: Pandas has another data structure called Panel which is not commonly used

Data structures index

Series and **DataFrame** also contain an integrated **`index`**:

- **Series** objects include a second array called the **index** that can contain homogeneous values of any type like integers, strings or datetime objects.
- **DataFrame** objects include a column **index**. The **index** objects allow for very sophisticated selection operations using square brackets (`[]`) along with some specialized selection functions.

Fast operations

`pandas` is fast, built on numpy, but goes beyond:

- `Series` and `DataFrames` are built upon `numpy.ndarray`
- `pandas` supports the same `ufunc` operations as in `numpy`, and the same fast vectorized computations.
- goes beyond `numpy` by providing elementwise string and datetime operations on indexing.
- `pandas` uses and supports additional C extensions written in Cython.

Integration with ecosystem

`pandas` is tightly integrated with the rest of the scientific Python ecosystem

- `pandas` is built on `numpy` arrays and `ufuncs`
- `pandas` data structures can be passed into `numpy`, `matplotlib`, and `seaborn` methods
- `pandas` has built-in visualization using `matplotlib`
- `pandas` is a dependency for Python statistics library `statsmodels`

Documentation

`pandas` is an enormous library.

We will concentrate on only a few elementary data analysis tasks using `pandas`, but more extensive descriptions of the motivations and functionality of `pandas` can be found in the resources below:

- pandas.pydata.org (http://pandas.pydata.org/)
  - pandas documentation (http://pandas.pydata.org/pandas-docs/stable/)
  - 10 minutes to Pandas (http://pandas.pydata.org/pandas-docs/stable/10min.html)
  - Cookbook at PyData (http://pandas.pydata.org/pandas-docs/stable/cookbook.html)

How to read a tabular data file into Pandas

In [2]:
```
# conventional way to import pandas
import pandas as pd
```

In [3]:
```
# local data files
!ls -l data/
```

```
total 4012
-rw-r--r-- 1 susana susana  364975 Jan 20 09:39 chipotle.tsv
-rw-r--r-- 1 susana susana    5918 Jan 20 09:39 drinks.csv
-rw-r--r-- 1 susana susana   91499 Jan 20 09:39 imdb_1000.csv
-rw-r--r-- 1 susana susana     300 Jan 20 09:39 mydata.json
-rw-r--r-- 1 susana susana 1410699 Jan 20 09:39 my_file.csv
-rw-r--r-- 1 susana susana 1411589 Jan 20 09:39 num.csv.gz
-rw-r--r-- 1 susana susana    1530 Jan 20 09:39 num.csv.info
-rw-r--r-- 1 susana susana   81845 Jan 20 09:39 output.xlsx
-rw-r--r-- 1 susana susana  668882 Jan 20 09:39 ufo.csv
-rw-r--r-- 1 susana susana   22628 Jan 20 09:39 u.user
```

In [4]:
```
# a new DataFrame reading from a file a dataset of Chipotle orders
orders = pd.read_table('data/chipotle.tsv')
```

In [5]:
```
type(orders)
```

Out[5]:
```
pandas.core.frame.DataFrame
```

Data Inspection

Use the method `DataFrame.head()` to inspect the first few rows of data:

- great way to inspect smaller data sets
- useful for verifying you've read the right file

In [6]:
```
# How to visualize the first five rows of the DataFrame
orders.head()
```

Out[6]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	1	Izze	[Clementine]	$3.39
2	1	1	Nantucket Nectar	[Apple]	$3.39
3	1	1	Chips and Tomatillo-Green Chili Salsa	NaN	$2.39
4	2	2	Chicken Bowl	[Tomatillo-Red Chili Salsa (Hot), [Black Beans...	$16.98

Data Inspection (2)

Use the method `DataFrame.tail()` to inspect the last few rows of data:

In [7]:
```
# How to visualize the last five rows of the DataFrame
orders.tail()
```

Out[7]:

	order_id	quantity	item_name	choice_description	item_price
4617	1833	1	Steak Burrito	[Fresh Tomato Salsa, [Rice, Black Beans, Sour ...	$11.75
4618	1833	1	Steak Burrito	[Fresh Tomato Salsa, [Rice, Sour Cream, Cheese...	$11.75
4619	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Pinto...	$11.25
4620	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Lettu...	$8.75
4621	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Pinto...	$8.75

Data Inspection (3)

In [8]:
```
# How to visualize all DataFrame
orders
```

Data Inspection (4)

In [9]:
```
# How to set the number of rows to be visualized
pd.set_option('max_rows',6)
orders
```

Out[9]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	1	Izze	[Clementine]	$3.39
2	1	1	Nantucket Nectar	[Apple]	$3.39
...	...	...	...	...	...
4619	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Pinto...	$11.25
4620	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Lettu...	$8.75
4621	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Pinto...	$8.75

4622 rows × 5 columns

Data Inspection (5)

Now use the Pandas `DataFrame.info()` method to see a bit more detail

In [10]:
```
# How to get infomations on a DataFrame
orders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4622 entries, 0 to 4621
Data columns (total 5 columns):
order_id              4622 non-null int64
quantity              4622 non-null int64
item_name             4622 non-null object
choice_description    3376 non-null object
item_price            4622 non-null object
dtypes: int64(2), object(3)
memory usage: 180.6+ KB
```

We get the name of each column, the Numpy dtype of each column, how many real values are present and the amount of memory used

side note: Pandas has excellent support for not-a-number (NaN) entries in DataFrames and Series.

Data Statistics

In Pandas we can very easily perform many types of statistical operations

In [11]:
```
pd.set_option('max_rows',8)
# How to calculate summary statistics of a DataFrame
orders.describe()
```

Out[11]:

	order_id	quantity
count	4622.000000	4622.000000
mean	927.254868	1.075725
std	528.890796	0.410186
min	1.000000	1.000000
25%	477.250000	1.000000
50%	926.000000	1.000000
75%	1393.000000	1.000000
max	1834.000000	15.000000

Obviously, only numerics columns have been summarized!

DataFrame attributes

In [12]:
```
# example attribute: number of rows and columns
orders.shape
```

Out[12]:
```
(4622, 5)
```

In [13]:
```
# example attribute: data type of each column
orders.dtypes
```

Out[13]:
```
order_id               int64
quantity               int64
item_name             object
choice_description    object
item_price            object
dtype: object
```

In [14]:
```
# How to get the names of columns
orders.columns
```

Out[14]:
```
Index(['order_id', 'quantity', 'item_name', 'choice_description',
       'item_price'],
      dtype='object')
```

Renaming columns in a DataFrame

In [15]:
```
# rename two of the columns by using the 'rename' method
orders.rename(columns={'order_id':'orderId', 'item_name':'itemName'}, \
              inplace=True)
orders.columns
```

Out[15]:
```
Index(['orderId', 'quantity', 'itemName', 'choice_description', 'item_p
rice'], dtype='object')
```

In [16]:
```
# a list with the modified columns name
myColumns = list(orders.columns)

# replace the col names during the file reading process
# by using 'names' parameter
myOrders = pd.read_table('data/chipotle.tsv', header=0, \
                         names=myColumns)
```

In [17]:
```
myOrders.columns
```

Out[17]:
```
Index(['orderId', 'quantity', 'itemName', 'choice_description', 'item_p
rice'], dtype='object')
```

Removing columns from a DataFrame (1)

In [18]:
```
# remove (temporarily) a single column (axis=1 refers to columns)
myOrders.drop('choice_description', axis=1).head(1)
```

Out[18]:

	orderId	quantity	itemName	item_price
0	1	1	Chips and Fresh Tomato Salsa	$2.39

The last one is a view of `myOrders` DataFrame; the column dropped is still in `myOrders`

In [19]:
```
myOrders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4622 entries, 0 to 4621
Data columns (total 5 columns):
orderId               4622 non-null int64
quantity              4622 non-null int64
itemName              4622 non-null object
choice_description    3376 non-null object
item_price            4622 non-null object
dtypes: int64(2), object(3)
memory usage: 180.6+ KB
```

Removing columns from a DataFrame (2)

To permanently remove a column from the DataFrame, you have to used the `drop` method, with the argument `inplace=True`

In [20]:
```
# To permanently remove the column from the DataFrame
myOrders.drop('choice_description', axis=1, inplace=True)
```

In [21]:
```
myOrders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4622 entries, 0 to 4621
Data columns (total 4 columns):
orderId       4622 non-null int64
quantity      4622 non-null int64
itemName      4622 non-null object
item_price    4622 non-null object
dtypes: int64(2), object(2)
memory usage: 144.5+ KB
```

Removing columns from a DataFrame (3)

In [22]:
```
# how to remove multiple columns at once
myOrders.drop(['orderId', 'item_price'], axis=1, inplace=True)
myOrders.head(3)
```

Out[22]:

	quantity	itemName
0	1	Chips and Fresh Tomato Salsa
1	1	Izze
2	1	Nantucket Nectar

In [23]:
```
myOrders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4622 entries, 0 to 4621
Data columns (total 2 columns):
quantity    4622 non-null int64
itemName    4622 non-null object
dtypes: int64(1), object(1)
memory usage: 72.3+ KB
```

Removing rows from a DataFrame

In [24]:
```
# remove multiple rows at once (axis=0 refers to rows)
myOrders.drop([0, 1], axis=0, inplace=True)
myOrders.head()
```

Out[24]:

	quantity	itemName
2	1	Nantucket Nectar
3	1	Chips and Tomatillo-Green Chili Salsa
4	2	Chicken Bowl
5	1	Chicken Bowl
6	1	Side of Chips

Columns name as DataFrame attributes

In [25]:
```
# Get columns of orders DataFrame
orders.columns
```

Out[25]:
```
Index(['orderId', 'quantity', 'itemName', 'choice_description', 'item_p
rice'], dtype='object')
```

In [26]:
```
# Each column name is an attribute of a DataFrame
orders.orderId
```

Out[26]:
```
0          1
1          1
2          1
3          1
        ...
4618    1833
4619    1834
4620    1834
4621    1834
Name: orderId, dtype: int64
```

- The output of the previous command looks different than the typical DataFrame output! What is the type of `orders.orderId`?

Series

In [27]:
```
# how to get the type of orders.orderId
type(orders.orderId)
```

Out[27]:
```
pandas.core.series.Series
```

- A **Pandas Series** is a single vector of data (like a `numpy` array), with an index that labels each element in the vector
- It supports both integer-based and label-based indexing

How to make a series

In [28]:
```
# Make a Series
S = pd.Series([632, 1638, 569, 115])
S
```

Out[28]:
```
0     632
1    1638
2     569
3     115
dtype: int64
```

Series attributes

In [29]:
```
# How to get the values of a series
S.values
```

Out[29]:
```
array([ 632, 1638,  569,  115])
```

In [30]:
```
# How to get the index of a series
S.index
```
Out[30]:
```
RangeIndex(start=0, stop=4, step=1)
```

- A *numpy array* contains the values of the series
- If an index is not specified, the natural sequence of integers is assigned as index
- The index of a series is a *pandas index object*

pandas.Series.describe()

In [31]:
```
# How to calculate summary statistics of a Series
S.describe()
```
Out[31]:
```
count       4.000000
mean      738.500000
std       642.371907
min       115.000000
25%       455.500000
50%       600.500000
75%       883.500000
max      1638.000000
dtype: float64
```

Name what you see

We can assign meaningful labels to the index, if they are available

In [32]:
```
S1 = pd.Series([632, 1638, 569, 115], index=['A', 'B', 'C', 'D'])
S1
```
Out[32]:
```
A     632
B    1638
C     569
D     115
dtype: int64
```

In [33]:
```
# Now it looks more like a dictionary
S1['C']
```
Out[33]:
```
569
```

Series from a dictionary

In [34]:
```
myDict = {'A': 632, 'B': 1638, 'C': 569, 'D': 115}
pd.Series(myDict)
```

Out[34]:
```
A     632
B    1638
C     569
D     115
dtype: int64
```

Naming index and array of values of a series

In [35]:
```
# We can give both the array of values and the index
# meaningful labels themselves

S1.name = 'counts'
S1.index.name = 'Letter'
S1
```

Out[35]:
```
Letter
A     632
B    1638
C     569
D     115
Name: counts, dtype: int64
```

Numpy and Pandas Series

NumPy's math functions and other operations can be applied to Series without losing the data structure.

In [36]:
```
import numpy as np
np.log(S1)
```

Out[36]:
```
Letter
A    6.448889
B    7.401231
C    6.343880
D    4.744932
Name: counts, dtype: float64
```

The truth about Series

Booleans mask

In [37]:
```
mask = [False, True, True, False]
S1[mask]
```

Out[37]:
```
Letter
B    1638
C     569
Name: counts, dtype: int64
```

Find what is missing

If we pass a custom index to Series

it will select the corresponding values from the dict

In [38]:
```
S2 = pd.Series(myDict, index= ['E','A','B','C'])
S2
```

Out[38]:
```
E       NaN
A     632.0
B    1638.0
C     569.0
dtype: float64
```

Indices without corresponding values are treat as **missing**

Pandas uses the NaN (not a number) type for missing values

In [39]:
```
# How to find what is missing
S2.isnull()
```

Out[39]:
```
E     True
A    False
B    False
C    False
dtype: bool
```

Adding two series

The labels are used to **align** data when used in operations with other Series

In [40]:
```
S1 + S2
```

Out[40]:
```
A    1264.0
B    3276.0
C    1138.0
D       NaN
E       NaN
dtype: float64
```

- We have a different behavior from `numpy`, where arrays of the same length are combined element-wise
- In the resulting series, values are the combination of original values with the same label; *the missing values were propagated* by addition

Back to `DataFrame`

How to make a DataFrame from scratch

- You can make a DataFrame from scratch, using the class `pd.DataFrame`
- DataFrame inputs could be:

  1. a Python dictionary of 1D sequences (e.g. ndarrays, lists, dicts, ..)
  2. a 2-D numpy.ndarray
  3. pandas Series
  4. another DataFrame

In [41]:
```
import numpy as np

# Creating a DataFrame from a dictionary
myDict = {'colA':[1,2,3,4,5,6,7,8,9,10],
          'colB':np.linspace(0,np.pi,10),
          'colC':0.0,
          'colD':["a","b","c","a","b","c","a","b","c","a"]}
myDF = pd.DataFrame(myDict)
myDF
```

Out[41]:

	colA	colB	colC	colD
0	1	0.000000	0.0	a
1	2	0.349066	0.0	b
2	3	0.698132	0.0	c
3	4	1.047198	0.0	a
...	...	...	...	...
6	7	2.094395	0.0	a
7	8	2.443461	0.0	b
8	9	2.792527	0.0	c
9	10	3.141593	0.0	a

10 rows × 4 columns

DataFrame from a 2D `numpy` array

In [42]:
```
myMatrix = np.random.random((1000,4))
myDF = pd.DataFrame(myMatrix, \
       columns=['firstCol','secondCol','thirdCol','fourCol'])
myDF
```

Out[42]:

	firstCol	secondCol	thirdCol	fourCol
0	0.716976	0.028238	0.504655	0.688705
1	0.494818	0.991270	0.213622	0.798711
2	0.525325	0.123154	0.402677	0.973707
3	0.275932	0.807159	0.957203	0.698916
...	...	...	...	...
996	0.853066	0.480446	0.535643	0.556289
997	0.044386	0.654772	0.428477	0.539479
998	0.228202	0.760028	0.981580	0.078365
999	0.990523	0.136896	0.709577	0.616428

1000 rows × 4 columns

Reading a new DataFrame

In [43]:
```
# How to read a DataFrame from the first 3 columns of a .csv.gz file
A = pd.read_csv('data/num.csv.gz', header=None, \
    names=['Elevation', 'Aspect', 'Slope'], usecols=range(0, 3))
A.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 71436 entries, 0 to 71435
Data columns (total 3 columns):
Elevation    71436 non-null int64
Aspect       71436 non-null int64
Slope        71436 non-null int64
dtypes: int64(3)
memory usage: 1.6 MB
```

Selecting and examining columns

In [44]:
```
type(A['Aspect'])
```

Out[44]:
```
pandas.core.series.Series
```

In [45]:
```
# For each value, count number of occurrences
A['Aspect'].value_counts()
```

Out[45]:
```
45     1366
90      931
135     917
0       874
       ...
227      36
224      23
226      19
360       5
Name: Aspect, dtype: int64
```

Aggregating data: the `groupby` method

Problem: for each unique value in column `Aspect` of the our DataFrame, we want to calculate the arithmetic mean of *ALL* other numeric columns

In [46]:
```
A.groupby('Aspect').mean()
```

Out[46]:

	Elevation	Slope
Aspect		
0	2931.376430	8.051487
1	2764.614815	15.296296
2	2793.150000	13.872222
3	2823.753555	14.142180
...	...	...
357	2840.698980	13.474490
358	2792.861272	13.520231
359	2781.500000	16.180851
360	2460.800000	29.800000

361 rows × 2 columns

In [47]:
```
# How to calculate the mean of 'Elevation' values grouped by 'Aspect'
A.groupby('Aspect').Elevation.mean()
```

Out[47]:
```
Aspect
0      2931.376430
1      2764.614815
2      2793.150000
3      2823.753555
          ...
357    2840.698980
358    2792.861272
359    2781.500000
360    2460.800000
Name: Elevation, dtype: float64
```

New columns in a DataFrame

How to add a new column as a function of existing columns

In [48]:
```
A['new_col1'] = A.Elevation * 10
A['new_col2'] = A['Slope'] + A['Aspect'] - 1
# Check it
A.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 71436 entries, 0 to 71435
Data columns (total 5 columns):
Elevation    71436 non-null int64
Aspect       71436 non-null int64
Slope        71436 non-null int64
new_col1     71436 non-null int64
new_col2     71436 non-null int64
dtypes: int64(5)
memory usage: 2.7 MB
```

In [49]:
```
# rename a column
A.rename(columns={'new_col2':'a_sum'}, inplace=True)
A
```

Out[49]:

	Elevation	Aspect	Slope	new_col1	a_sum
0	2596	51	3	25960	53
1	2590	56	2	25900	57
2	2804	139	9	28040	147
3	2785	155	18	27850	172
...	...	...	...	...	...
71432	2912	97	6	29120	102
71433	2911	207	1	29110	207
71434	2912	74	3	29120	76
71435	2910	72	5	29100	76

71436 rows × 5 columns

Recap: attributes and methods

```
A.columns    # column names (which is "an index")
A.dtypes     # data types of each column
A.shape      # number of rows and columns
A.values     # underlying numpy array
```

... and many more: just write `A.` [and press TAB]

Filtering Data

A first simple filter

In [50]:
```
# Filter rows, based on column values
A[A['Aspect'] == 20]
```

Out[50]:

	Elevation	Aspect	Slope	new_col1	a_sum
153	2687	20	11	26870	30
692	2890	20	16	28900	35
1075	3020	20	18	30200	37
1477	3133	20	29	31330	48
...	...	...	...	...	...
70276	2761	20	9	27610	28
70377	3044	20	7	30440	26
70458	2995	20	7	29950	26
70912	2774	20	22	27740	41

270 rows × 5 columns

Advanced logical filtering

In [51]:
```
# use multiple conditions
condition = (A.Elevation < 2400) & (A.Slope == 17)
cols = ['Slope', 'Aspect']
A[condition][cols]
```

Out[51]:

	Slope	Aspect
1820	17	344
1825	17	50
1890	17	262
2234	17	37
...	...	...
13327	17	84
13460	17	48
13555	17	287
15007	17	107

146 rows × 2 columns

Note: `AND` and `OR` do not work inside data frame conditions

Filtering of specific values

In [52]:
```
B = A[A['Aspect'].isin([13,17])]
B
```

Out[52]:

	Elevation	Aspect	Slope	new_col1	a_sum
67	2919	13	13	29190	25
91	2788	13	16	27880	28
119	2837	17	13	28370	29
378	2781	13	3	27810	15
...	...	...	...	...	...
71117	2926	13	5	29260	17
71309	3067	17	11	30670	27
71312	3047	13	13	30470	25
71333	2952	17	12	29520	28

602 rows × 5 columns

In [53]:
```
B.shape
```

Out[53]:
```
(602, 5)
```

Selecting rows and columns from a DataFrame

The **`loc`** (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.loc.html) method is used to select rows and columns by **label**. You can pass it:

- A single label
- A list of labels
- A slice of labels
- A boolean Series
- A colon (which indicates "all labels")

In [54]:
```
# row 0, all columns
A.loc[0]
```

Out[54]:
```
Elevation     2596
Aspect          51
Slope            3
new_col1     25960
a_sum           53
Name: 0, dtype: int64
```

In [55]:
```
# How to get rows 0 and 1 and 2, all columns
A.loc[[0, 1, 2], :]
```

Out[55]:

	Elevation	Aspect	Slope	new_col1	a_sum
0	2596	51	3	25960	53
1	2590	56	2	25900	57
2	2804	139	9	28040	147

In [56]:
```
# How to get rows 1 through 2 (inclusive), all columns
A.loc[1:2, :]
```

Out[56]:

	Elevation	Aspect	Slope	new_col1	a_sum
1	2590	56	2	25900	57
2	2804	139	9	28040	147

In [57]:
```
# also this implies "all columns", but it's better to be explicit!
A.loc[1:2]
```

Out[57]:

	Elevation	Aspect	Slope	new_col1	a_sum
1	2590	56	2	25900	57
2	2804	139	9	28040	147

In [58]:
```
# How to get rows 0 through 2 (inclusive),
# columns 'Aspect' and 'Slope'
A.loc[0:2, ['Aspect', 'Slope']]
```

Out[58]:

	Aspect	Slope
0	51	3
1	56	2
2	139	9

In [59]:
```
# the same selection using double brackets,
# but using 'loc' is more explicit
A[['Aspect', 'Slope']].head(3)
```

Out[59]:

	Aspect	Slope
0	51	3
1	56	2
2	139	9

In [60]:
```
# How to get rows 0 through 2 (inclusive),
# columns 'Elevation' through 'Slope' (inclusive)
A.loc[0:2, 'Elevation':'Slope']
```

Out[60]:

	Elevation	Aspect	Slope
0	2596	51	3
1	2590	56	2
2	2804	139	9

In [61]:
```
# How to get a single element of the DataFrame
A.loc[1,'Slope']
```

Out[61]:
```
2
```

In [62]:
```
# How to get rows 0 through 2 (inclusive), column 'Slope'
A.loc[0:2, 'Slope']
```

Out[62]:
```
0    3
1    2
2    9
Name: Slope, dtype: int64
```

Selecting rows and columns by *position*

The **`iloc`** [(http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iloc.html)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.iloc.html) method is used to select rows and columns by **position**. Purely integer-location based indexing for selection by position

In [63]:
```
A.head(2)
```

Out[63]:

	Elevation	Aspect	Slope	new_col1	a_sum
0	2596	51	3	25960	53
1	2590	56	2	25900	57

In [64]:
```
# How to get the element in row 1 and column 4 ('a_sum')
A.iloc[1,4]
```

Out[64]:
```
57
```

In [65]:
```
# How to get rows 0 through 2 (inclusive), column 4 ('a_sum')
A.iloc[0:3,4]
```

Out[65]:
```
0     53
1     57
2    147
Name: a_sum, dtype: int64
```

Ordering data

In [66]:
```
# How to sort rows by label
A.sort_index()
```

Out[66]:

	Elevation	Aspect	Slope	new_col1	a_sum
0	2596	51	3	25960	53
1	2590	56	2	25900	57
2	2804	139	9	28040	147
3	2785	155	18	27850	172
...	...	...	...	...	...
71432	2912	97	6	29120	102
71433	2911	207	1	29110	207
71434	2912	74	3	29120	76
71435	2910	72	5	29100	76

71436 rows × 5 columns

In [67]:
```
# How to sort rows by a specific column
A.sort_values(by='Aspect')
```

Out[67]:

	Elevation	Aspect	Slope	new_col1	a_sum
65052	2982	0	9	29820	8
35524	3031	0	13	30310	12
11780	3093	0	4	30930	3
5967	2364	0	2	23640	1
...	...	...	...	...	...
22394	2800	360	26	28000	385
15816	2673	360	35	26730	394
2343	2062	360	30	20620	389
12204	2046	360	29	20460	388

71436 rows × 5 columns

In [68]:
```python
# How to sort using descending order instead
A.sort_values(by='Aspect', ascending=False)
```

Out[68]:

	Elevation	Aspect	Slope	new_col1	a_sum
15816	2673	360	35	26730	394
12204	2046	360	29	20460	388
22394	2800	360	26	28000	385
52847	2723	360	29	27230	388
...	...	...	...	...	...
13567	2364	0	4	23640	3
57899	2975	0	5	29750	4
32479	2919	0	9	29190	8
46484	3041	0	9	30410	8

71436 rows × 5 columns

In [69]:
```python
# How to sort by multiple columns
A.sort_values(by=['Aspect', 'Slope'])
```

Out[69]:

	Elevation	Aspect	Slope	new_col1	a_sum
1542	3289	0	0	32890	-1
6521	3084	0	0	30840	-1
8325	3340	0	0	33400	-1
14884	2688	0	0	26880	-1
...	...	...	...	...	...
12204	2046	360	29	20460	388
52847	2723	360	29	27230	388
2343	2062	360	30	20620	389
15816	2673	360	35	26730	394

71436 rows × 5 columns

So far

- Check data before getting started
- Choose columns by label or index
- Filter rows on index and values
- Group by
- Sort

note: just like *databases*

hint: we can do better

Uniqueness of data

How to find what is duplicated?

In [70]:
```
# Series of booleans (True if a row is identical to a previous row)
A.duplicated()
```

Out[70]:
```
0        False
1        False
2        False
3        False
         ...
71432    False
71433    False
71434    False
71435    False
dtype: bool
```

Working with duplicated data

In [71]:
```
# Count duplicates
A.duplicated().sum()
```

Out[71]: 3229

In [72]:
```
# Find the duplicate with minimum elevation value
A[A.duplicated()].sort_values(by='Elevation').head(1)
```

Out[72]:

	Elevation	Aspect	Slope	new_col1	a_sum
3426	2331	185	15	23310	199

In [73]:
```
# Check if it really is a duplicate
A[(A.Elevation==2331) & (A.Aspect==185)]
```

Out[73]:

	Elevation	Aspect	Slope	new_col1	a_sum
3328	2331	185	15	23310	199
3426	2331	185	15	23310	199

Delete duplicated data

In [74]:
```
# Drop duplicate rows
B = A.drop_duplicates()
```

In [75]:
```
# The DataFrame B doesn't have duplicated rows!
B.duplicated().sum()
```

Out[75]: 0

In [76]:
```
# How to delete duplicated rows in A
A.drop_duplicates(inplace=True)
# Now also A doesn't have duplicated rows
A.duplicated().sum()
```

Out[76]: 0

The index

In [77]:
```
# Use a different example dataset
baseball = pd.read_csv("https://raw.githubusercontent.com/fonnesbeck/sta
```

In [78]:
```
pd.set_option('max_columns', 10)
pd.set_option('max_rows', 8)
baseball
```

Out[78]:

	player	year	stint	team	lg	...	ibb	hbp	sh	sf	gidp
id											
88641	womacto01	2006	2	CHN	NL	...	0.0	0.0	3.0	0.0	0.0
88643	schilcu01	2006	1	BOS	AL	...	0.0	0.0	0.0	0.0	0.0
88645	myersmi01	2006	1	NYA	AL	...	0.0	0.0	0.0	0.0	0.0
88649	helliri01	2006	1	MIL	NL	...	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...
89526	benitar01	2007	1	SFN	NL	...	0.0	0.0	0.0	0.0	0.0
89530	ausmubr01	2007	1	HOU	NL	...	3.0	6.0	4.0	1.0	11.0
89533	aloumo01	2007	1	NYN	NL	...	5.0	2.0	0.0	3.0	13.0
89534	alomasa02	2007	1	NYN	NL	...	0.0	0.0	0.0	0.0	0.0

100 rows × 22 columns

Reindexing

In [79]:
```
type(baseball.index)
```

Out[79]:
```
pandas.indexes.numeric.Int64Index
```

Reindexing allows users to manipulate the data labels in a DataFrame

In [80]:
```
# Alter the order of the rows
baseball.reindex(baseball.index[::-1]).head()
```

Out[80]:

	player	year	stint	team	lg	...	ibb	hbp	sh	sf	gidp
id											
89534	alomasa02	2007	1	NYN	NL	...	0.0	0.0	0.0	0.0	0.0
89533	aloumo01	2007	1	NYN	NL	...	5.0	2.0	0.0	3.0	13.0
89530	ausmubr01	2007	1	HOU	NL	...	3.0	6.0	4.0	1.0	11.0
89526	benitar01	2007	1	SFN	NL	...	0.0	0.0	0.0	0.0	0.0
89525	benitar01	2007	2	FLO	NL	...	0.0	0.0	0.0	0.0	0.0

5 rows × 22 columns

In [81]:
```
# Let's try all the possible keys
cols = ['player','year','team']
id_range = range(baseball.index.values.min(),
                 baseball.index.values.max())
baseball[cols].reindex(id_range).head()
```

Out[81]:

	player	year	team
id			
88641	womacto01	2006.0	CHN
88642	NaN	NaN	NaN
88643	schilcu01	2006.0	BOS
88644	NaN	NaN	NaN
88645	myersmi01	2006.0	NYA

In [82]:
```
# Fill the blanks
baseball.reindex(id_range, fill_value='MR Unknown', columns=['player']).
```

Out[82]:

	player
id	
88641	womacto01
88642	MR Unknown
88643	schilcu01
88644	MR Unknown
88645	myersmi01

In [83]:
```
# In different methods
baseball[cols].reindex(id_range, method='ffill',
                       columns=['player','year']).head()
```

Out[83]:

	player	year
id		
88641	womacto01	2006
88642	womacto01	2006
88643	schilcu01	2006
88644	schilcu01	2006
88645	myersmi01	2006

In [85]:
```
baseball.index.is_unique
```

Out[85]: True

Combining DataFrames

the index magic

In [86]:
```
df1 = pd.DataFrame({'A': ['A0','A1'], 'B': ['B0','B1']}, index=[0, 1])
df2 = pd.DataFrame({'A': ['A4','A5','A6'], \
                    'B': ['B4','B5','B6']},index=[4, 5, 6])
df3 = pd.DataFrame({'A': ['A7','A8'],'B': ['B7','B8']}, index=[7, 8])
# Combine these 3 DataFrame
pd.concat([df1,df2,df3])
```

Out[86]:

	A	B
0	A0	B0
1	A1	B1
4	A4	B4
5	A5	B5
6	A6	B6
7	A7	B7
8	A8	B8

Combining DataFrames: an alternative method

In [87]:
```
# Alternative combo
df1.append(df2).append(df3)
```

Out[87]:

	A	B
0	A0	B0
1	A1	B1
4	A4	B4
5	A5	B5
6	A6	B6
7	A7	B7
8	A8	B8

Joining data

`pandas` has full-featured, high performance in-memory join operations, idiomatically very similar to relational databases like SQL.

In [88]:
```
df1 = pd.DataFrame({'A': ['A0','A1'], 'key': ['k1','k2'], \
     'B': ['B0','B1']}, index=[0, 1])
df2 = pd.DataFrame({'A': ['A4','A5','A6','A7'], \
     'key': ['k2','k1','k4','k3'], 'B': ['B4','B0','B6','B7']}, \
     index=[4, 5, 6, 7])
```

In [89]:
```
df1
```
Out[89]:

	A	B	key
0	A0	B0	k1
1	A1	B1	k2

In [90]:
```
df2
```
Out[90]:

	A	B	key
4	A4	B4	k2
5	A5	B0	k1
6	A6	B6	k4
7	A7	B7	k3

The `merge()` method

In [91]:
```
pd.merge(df1,df2, on='key')
```
Out[91]:

	A_x	B_x	key	A_y	B_y
0	A0	B0	k1	A5	B0
1	A1	B1	k2	A4	B4

In [92]:
```
# Join on multiple keys!
pd.merge(df1,df2, on=['key','B'])
```
Out[92]:

	A_x	B	key	A_y
0	A0	B0	k1	A5

pandas joining vs SQL joining

Merge method	SQL Join Name	Description
`left`	`LEFT OUTER JOIN`	Use keys from left frame only
`right`	`RIGHT OUTER JOIN`	Use keys from right frame only
`outer`	`FULL OUTER JOIN`	Use union of keys from both frames
`inner`	`INNER JOIN`	Use intersection of keys from both frames

```
In [39]: result = merge(left, right, how='left', on=['key1', 'key2'])
```

source: [http://pandas.pydata.org/pandas-docs/stable/merging.html (http://pandas.pydata.org/pandas-docs/stable/merging.html)](http://pandas.pydata.org/pandas-docs/stable/merging.html)

pandas I/O

Snapshot of a `DataFrame`

Quickly write the current status of a DataFrame to a CSV file

In [93]:
```
csvfile = 'data/my_file.csv'
```

In [94]:
```
A.to_csv(csvfile)
# Warning: index is used as first column
!head {csvfile}
```

```
,Elevation,Aspect,Slope,new_col1,a_sum
0,2596,51,3,25960,53
1,2590,56,2,25900,57
2,2804,139,9,28040,147
3,2785,155,18,27850,172
4,2595,45,2,25950,46
5,2579,132,6,25790,137
6,2606,45,7,26060,51
7,2605,49,4,26050,52
8,2617,45,9,26170,53
```

In [95]:
```
# Better to ignore the index column!
A.to_csv(csvfile, index=False)
!head {csvfile}
```

```
Elevation,Aspect,Slope,new_col1,a_sum
2596,51,3,25960,53
2590,56,2,25900,57
2804,139,9,28040,147
2785,155,18,27850,172
2595,45,2,25950,46
2579,132,6,25790,137
2606,45,7,26060,51
2605,49,4,26050,52
2617,45,9,26170,53
```

Reading from a remote source

In [96]:
```
# read CSV file directly from a URL and save the results
data = pd.read_csv(\
    'http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv', index_col=0)
data
```

Out[96]:

	TV	Radio	Newspaper	Sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	9.3
4	151.5	41.3	58.5	18.5
...	...	...	...	...
197	94.2	4.9	8.1	9.7
198	177.0	9.3	6.4	12.8
199	283.6	42.0	66.2	25.5
200	232.1	8.6	8.7	13.4

200 rows × 4 columns

pandas I/O in Excel files

In [99]:
```
# Our xls file
xlsfile = 'data/output.xlsx'

# how to write on a xls file from A DataFrame
A[:2500].to_excel(xlsfile,'Sheet1')
```

In [102]:
```
# how to read data from a xls file
B = pd.read_excel(xlsfile, sheetname='Sheet1')
B.head()
```

Out[102]:

	Elevation	Aspect	Slope	new_col1	a_sum
0	2596	51	3	25960	53
1	2590	56	2	25900	57
2	2804	139	9	28040	147
3	2785	155	18	27850	172
4	2595	45	2	25950	46

An uncomfortable DataFrame

In [103]:
```
# A new DataFrame
mydata = pd.DataFrame({0: {'patient': 1, 'phylum': 'Firmicutes',
                           'value': 632},
        1: {'patient': 1, 'phylum': 'Proteobacteria', 'value': 1638},
        2: {'patient': 1, 'phylum': 'Actinobacteria', 'value': 569},
        3: {'patient': 1, 'phylum': 'Bacteroidetes', 'value': 115},
        4: {'patient': 2, 'phylum': 'Firmicutes', 'value': 433},
        5: {'patient': 2, 'phylum': 'Proteobacteria', 'value': 1130},
        6: {'patient': 2, 'phylum': 'Actinobacteria', 'value': 754},
        7: {'patient': 2, 'phylum': 'Bacteroidetes', 'value': 555}})
mydata
```

Out[103]:

	0	**1**	**2**	**3**	**4**	**5**
patient	1	1	1	1	2	2
phylum	Firmicutes	Proteobacteria	Actinobacteria	Bacteroidetes	Firmicutes	Proteobacteria
value	632	1638	569	115	433	1130

Transpose data

In [104]:
```
mydata = mydata.T
mydata
```

Out[104]:

	patient	**phylum**	**value**
0	1	Firmicutes	632
1	1	Proteobacteria	1638
2	1	Actinobacteria	569
3	1	Bacteroidetes	115
4	2	Firmicutes	433
5	2	Proteobacteria	1130
6	2	Actinobacteria	754
7	2	Bacteroidetes	555