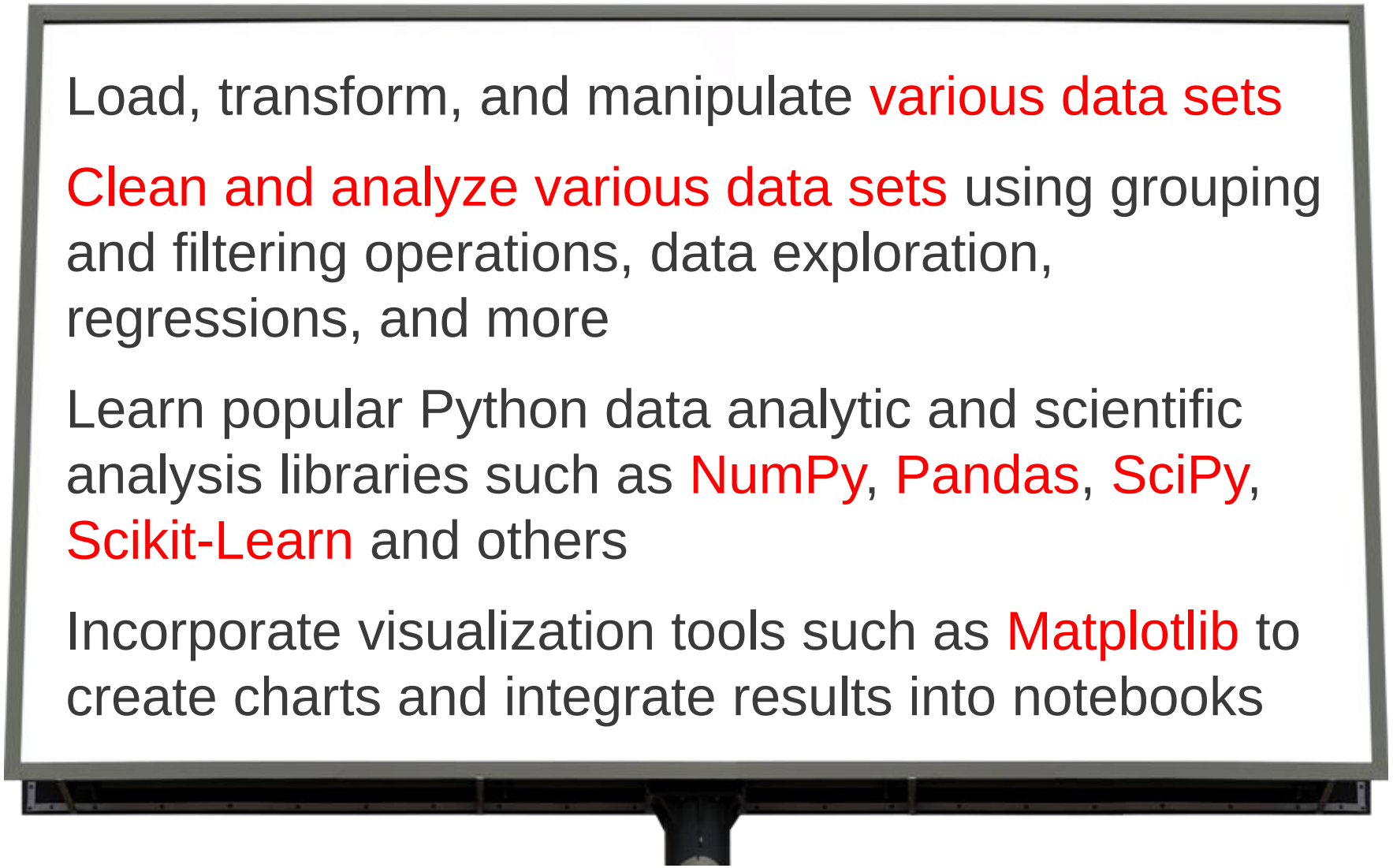
Data Analysis

Using Python

Course Objectives



Course Agenda

Day 1

- Data Analysis Tools and NumPy
- Jupyter Notebook
- More with NumPy
- Introducing Matplotlib

Course Agenda

Day 2

- Working with Pandas
- More with Pandas

Course Agenda

Day 3

- Pandas and the Database
- Seaborn for Exploratory Data Analysis
- Introducing Scikit-learn

Introductions

- Name
- What you work on
- Your background / experience with Python
- Reason for attending

Logistics

Typical Daily Schedule*

9:00	Start Day
10:15	Morning Break
12:00 – 1:00	Lunch
2:15	Afternoon Break 1
3:30	Afternoon Break 2
5:00	End of Day

** Your schedule may vary*

Get the Most from Your Experience

Ask Questions

Chapter 1

Data Analysis

and NumPy

Introducing Data Analysis Tools and Techniques in Python

Overview

The Anaconda Distribution

IPython and Jupyter Notebooks

NumPy

The Anaconda Python Distribution

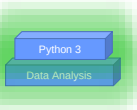
- Anaconda is a Python (and R) distribution used within data science application development
  - Distribution is provided by *Continuum Analytics*
    - Ships with a BSD license for Windows, Linux, OS X
    - Valid for Python 2.7, 3.5, 3.6, 3.7 (as of this writing)
  - Ships with over 100 pre-installed packages

- Anaconda provides its own package installer: *conda*

```
conda install <packagename>
```

Example: `conda install paramiko`

Other conda subcommands include: remove (uninstall), update (upgrade), list, search, info, create



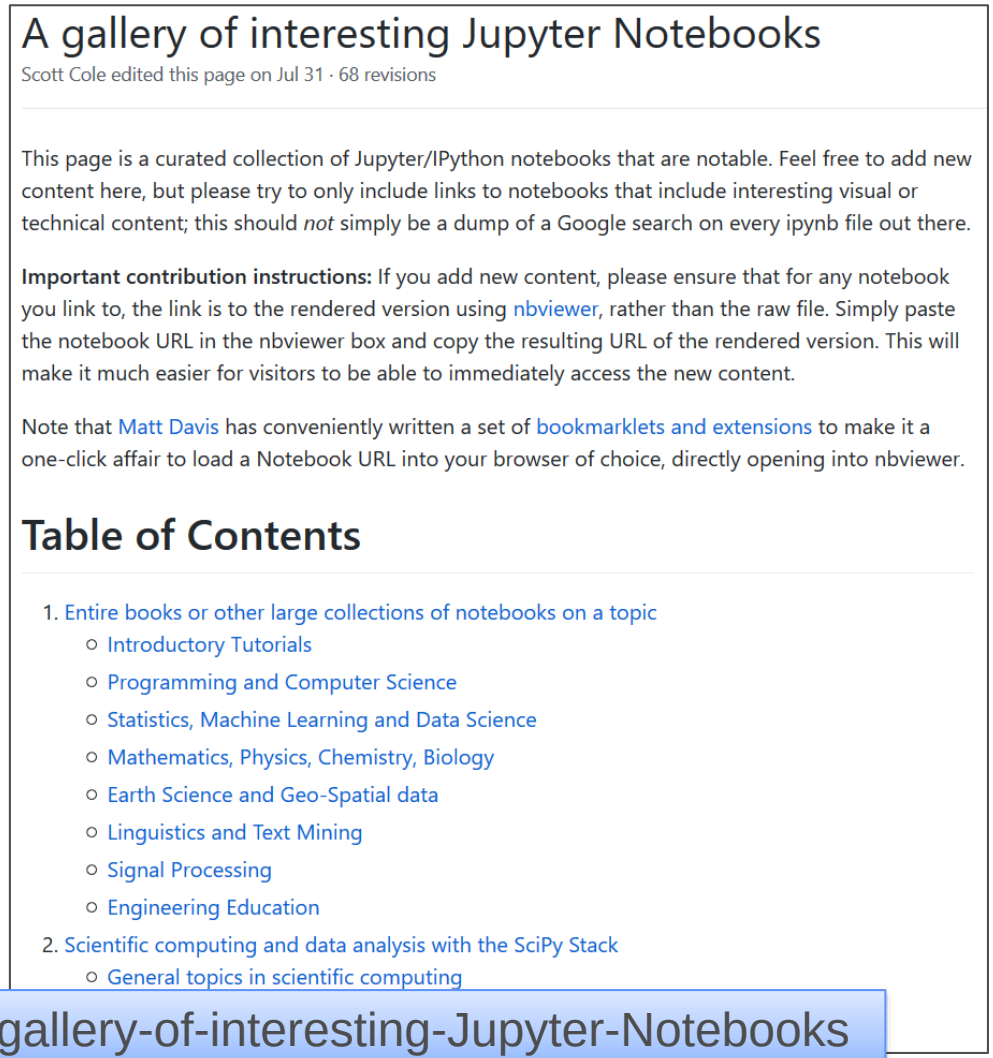
Anaconda Packages

Some of the available packages in the Anaconda distribution

astropy - astronomy
babel - internationalization
beautifulsoup4 - html parsing
bitarray - arrays of Booleans
blaze - big data for numpy, pandas
boto - AWS tools
bottleneck - fast numpy arrays
chest - like shelve
cloudpickle, pickleshare - pickling tools
colorama - colored text
configobj - file reading/writing
decorator - decorators
docutils - tools for doc generation
fastcache - faster cache
flask - web microframework
gevent/greenlets - concurrency
imagesize jpeg, pillow - image manipulation
ipython, jupyter - interactive shell
jinja2 - templating
jmespath - json query language
lxml - xml parser
markupsafe - xml as strings
matplotlib - 2D plots
nose, pytest - testing
notebook - interactive computing
numpy - number & array processing
openpyxl - work with .xls files
pandas - data analysis tools
pathlib2 - file path manipulation
path.py - os.path wrapper
pep8, pyflakes, pylink - style checkers
pexpect - command line manager
pymysql, pyodbc - db tools
pywin32 - Windows extensions
pyyaml - yaml support
pyreadline, readline - read files
redis - key-value database
scipy - scientific tools
six - 2, 3 compatibility
sphinx - documentation generator
spyder - scientific tools
sqlalchemy - Python-database ORM
statsmodels - statistical models
tornado - asynchronous server
unicodecsv - csv unicode strings
xlsxwriter - create xls files
yaml - yaml manipulation/creation
pip, setuptools - package management

IPython and Jupyter

- IPython is a Python-based interactive computing environment
  - Coupled with Jupyter Notebook, it is possible to share, collaborate, validate, or test analyzed data
  - Some publicly shared Jupyter Notebook:



A gallery of interesting Jupyter Notebooks

Scott Cole edited this page on Jul 31 · 68 revisions

This page is a curated collection of Jupyter/IPython notebooks that are notable. Feel free to add new content here, but please try to only include links to notebooks that include interesting visual or technical content; this should *not* simply be a dump of a Google search on every ipynb file out there.

Important contribution instructions: If you add new content, please ensure that for any notebook you link to, the link is to the rendered version using nbviewer, rather than the raw file. Simply paste the notebook URL in the nbviewer box and copy the resulting URL of the rendered version. This will make it much easier for visitors to be able to immediately access the new content.

Note that Matt Davis has conveniently written a set of bookmarklets and extensions to make it a one-click affair to load a Notebook URL into your browser of choice, directly opening into nbviewer.

Table of Contents

1. Entire books or other large collections of notebooks on a topic
   - Introductory Tutorials
   - Programming and Computer Science
   - Statistics, Machine Learning and Data Science
   - Mathematics, Physics, Chemistry, Biology
   - Earth Science and Geo-Spatial data
   - Linguistics and Text Mining
   - Signal Processing
   - Engineering Education
2. Scientific computing and data analysis with the SciPy Stack
   - General topics in scientific computing

https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks

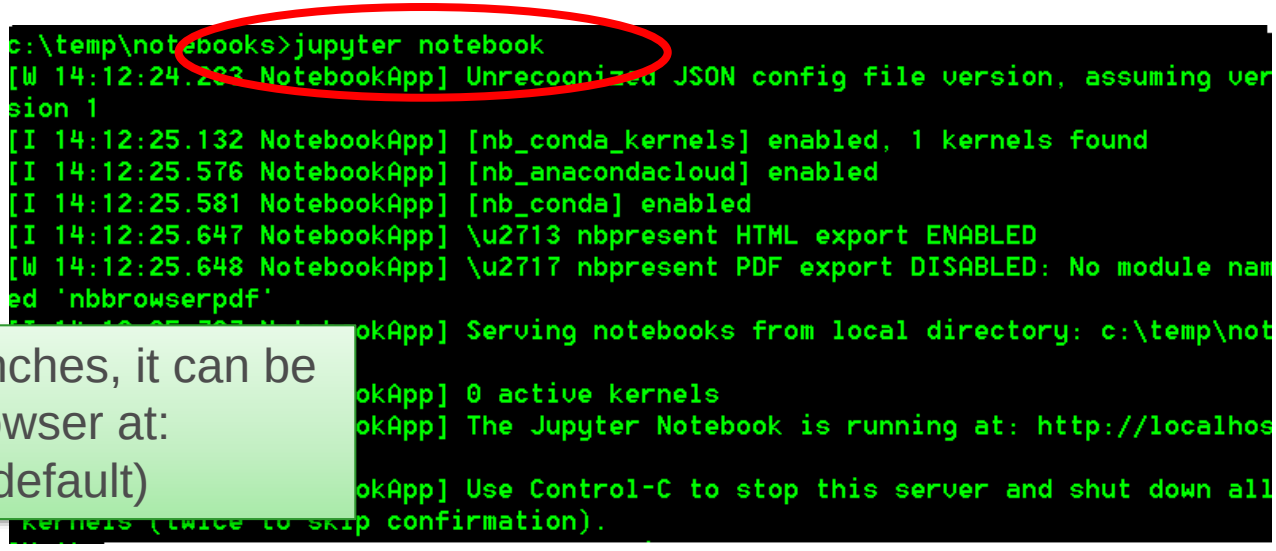
Launching Jupyter

- Anaconda ships with IPython built into it
  - If not using Anaconda, other distributions may install it separately:

```
pip install ipython
pip install jupyter
```

  - Launch a notebook (from a desired home directory):

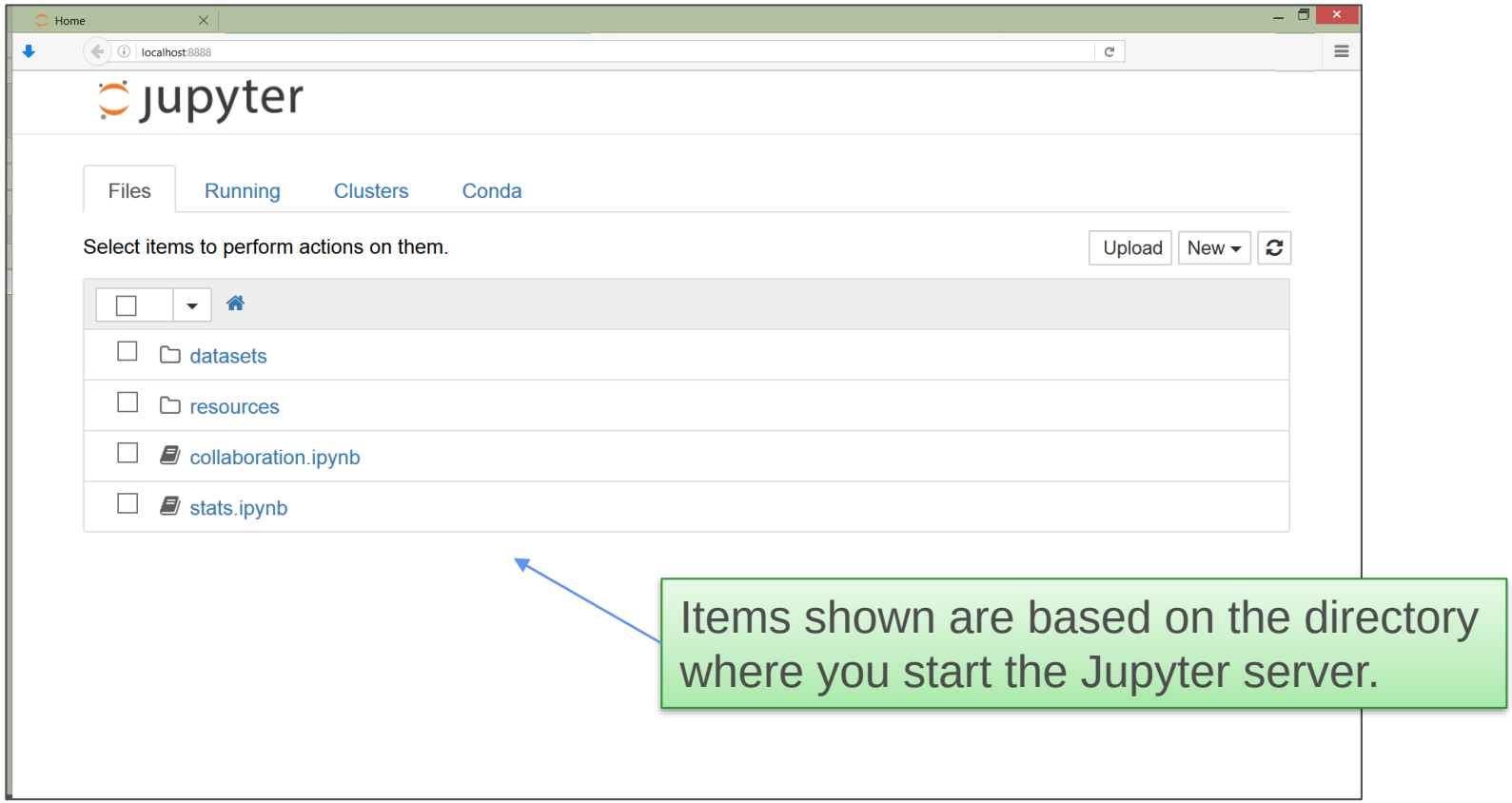
```
jupyter notebook
```



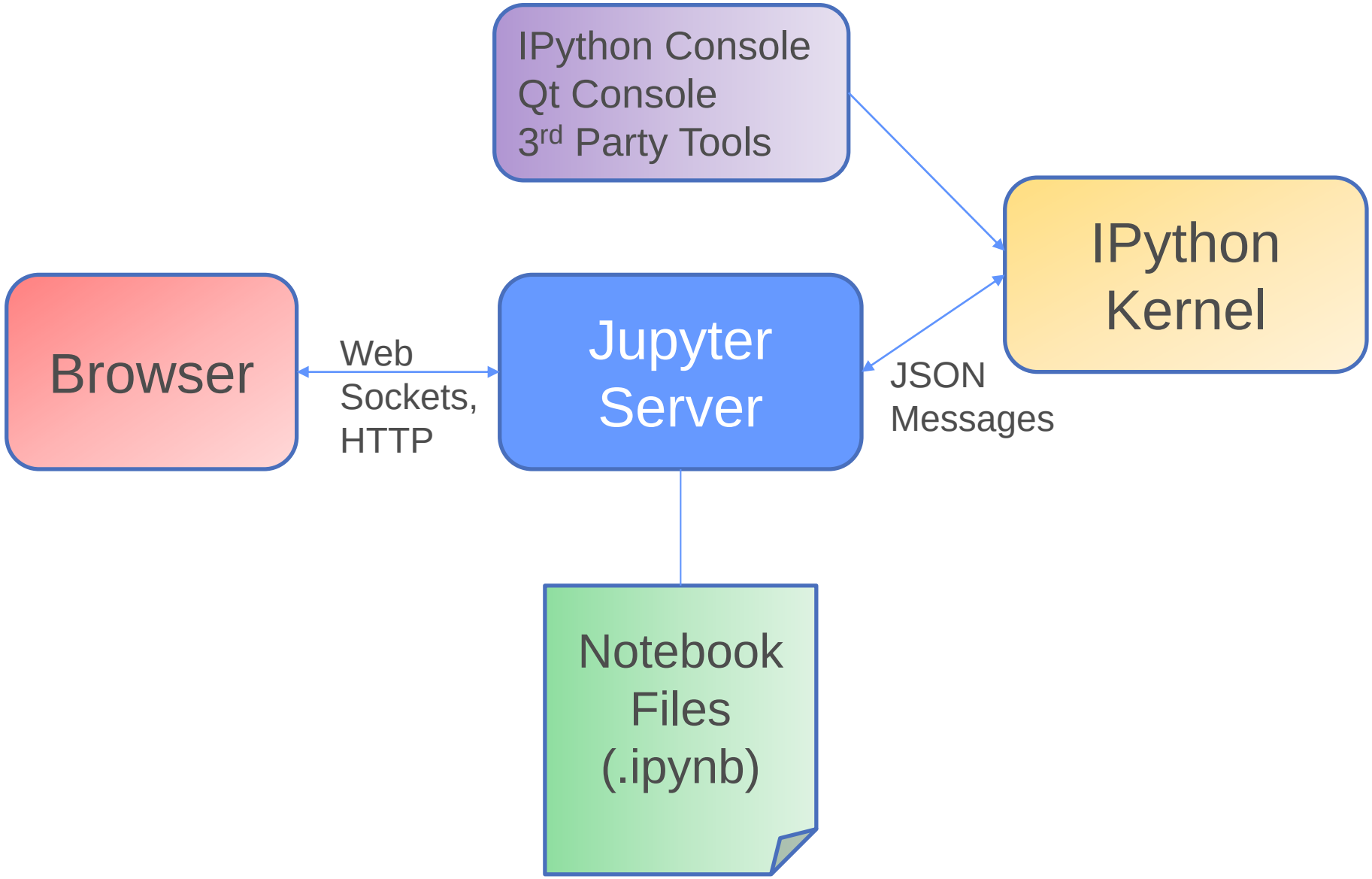
Once the server launches, it can be accessed via the browser at: **localhost:8888** (by default)

Jupyter Dashboard

- The web-based dashboard allows for creating and running code in notebooks



IPython-Jupyter Relationship

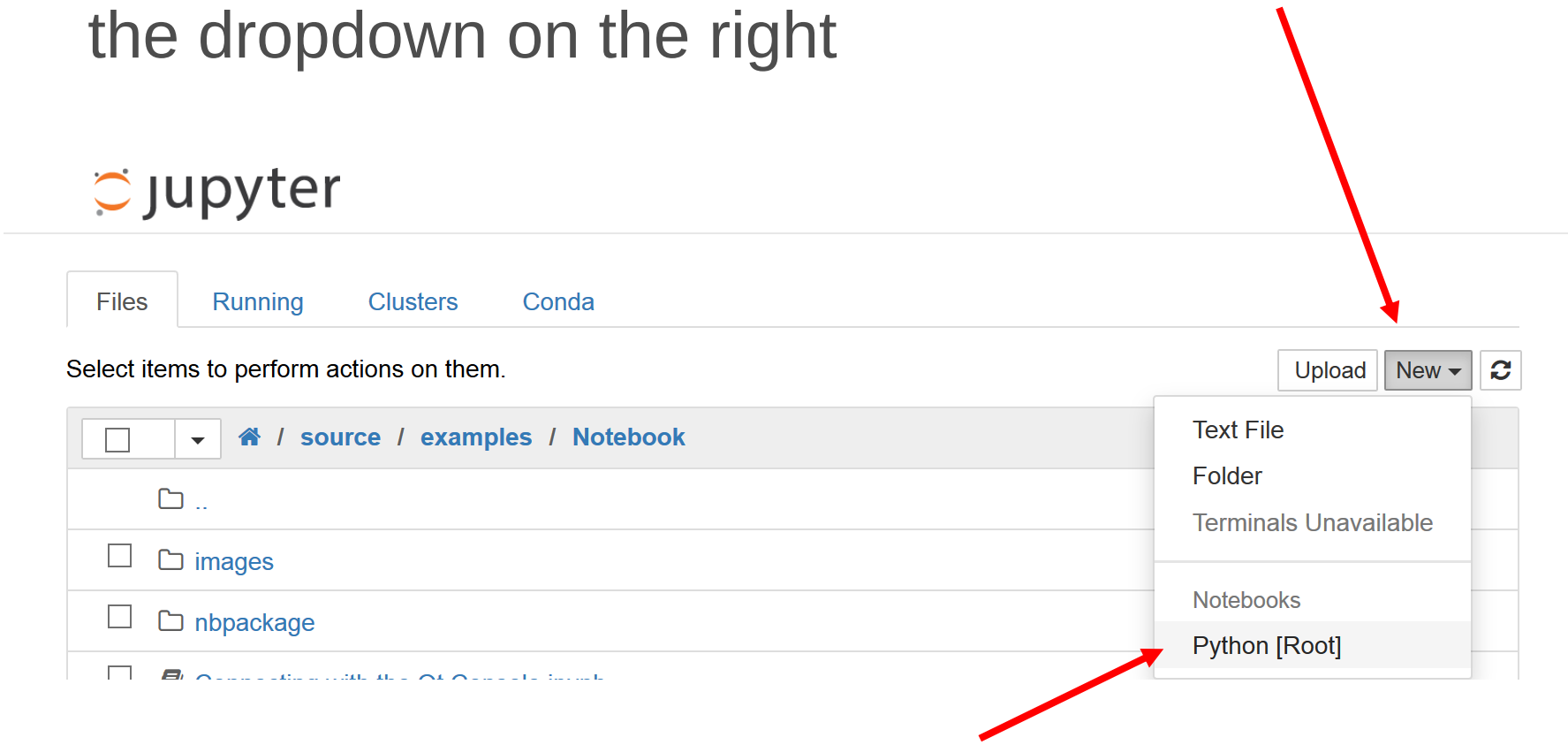


Notebook Components

- Notebooks are documents that contain live code, images, video, text, graphs, widgets and more
  - Notebooks can be run from within the browser using different kernels (programming languages)
  - Create .ipynb files on the local file system
- Notebooks are made up of cells
- Two most common cells:
  - Markdown cells (narrative LaTeX-based text)
  - Code cells (cells containing code from a language)

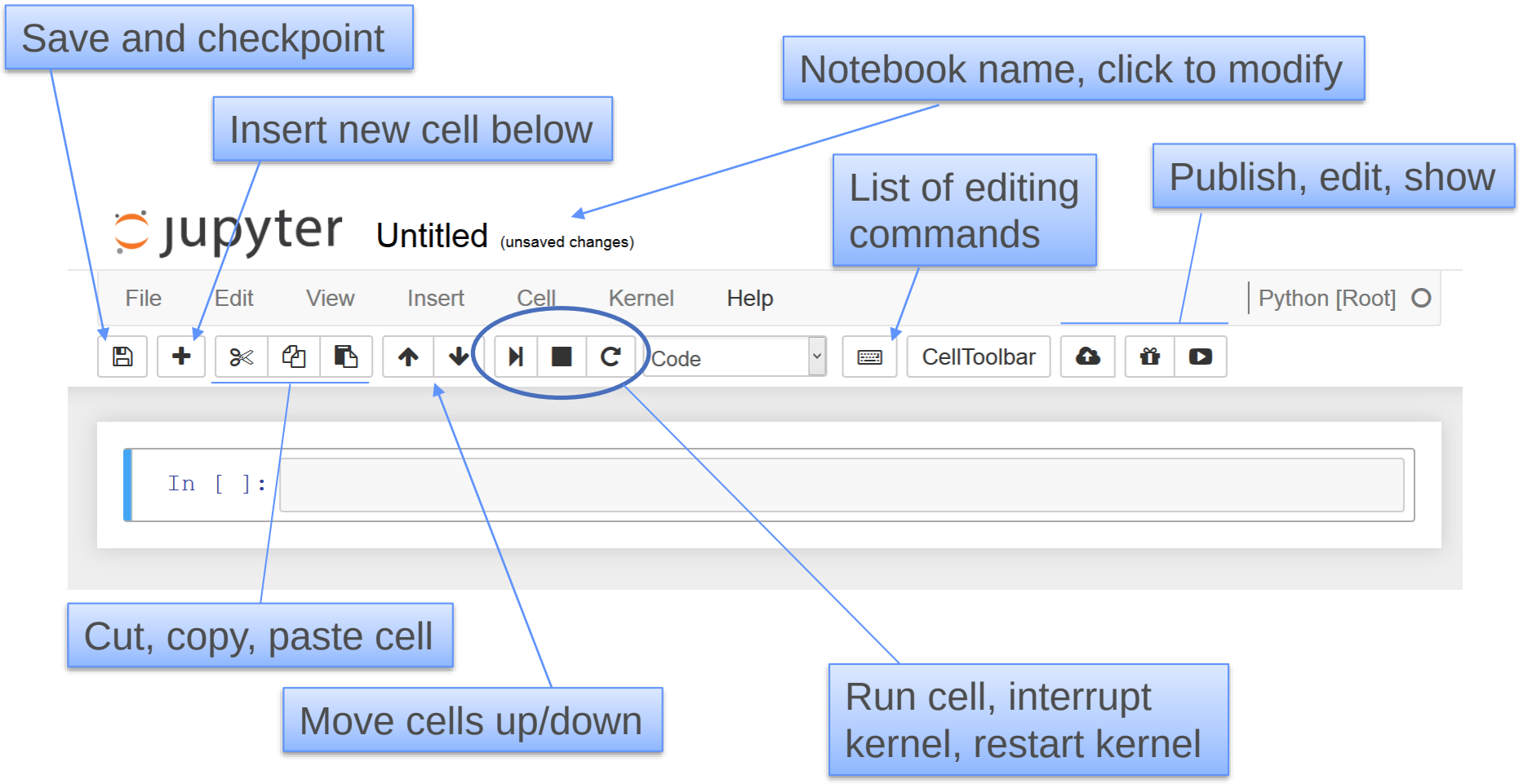
Creating Notebooks

- Create a new notebook by selecting "New" from the dropdown on the right



Notebook Cells Toolbar

- The Notebook menu and toolbar provide the ability to modify, execute, and save Notebooks



Notebook Editor Modes

- **Command mode** `ESC` key on a cell initiates this mode

```
In [ ]:
```

  - In this mode, a cell responds to special commands (see the list of editing commands )

- **Edit mode** `Enter` key on a cell initiates this mode

```
In [ ]: x = 10
```

  - In this mode, a cell can be typed into (like an editor)

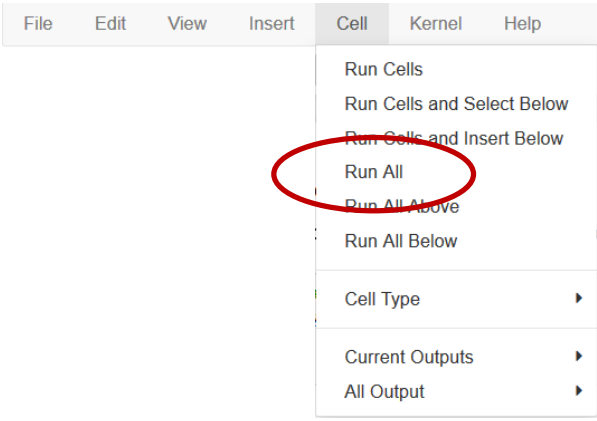
Mouse clicking in the appropriate areas will also cause the cell to go into Edit or Command Mode

Running Notebooks

- Pressing **`Shift-Enter`** or clicking  while the selected cell is in command mode will run the cell

```
In [1]: import sys
        print(sys.version)

        3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.
        1900 64 bit (AMD64)]

In [ ]:
```

- To run all cells, use the menu item:



Markdown Notation

- Markdown cells allow for rendering *text* into notebooks
  - Markdown notation include:

H1	#
H2	##
H3	###
H4	####
H5	#####
H6	#########

Headers

Lists

1. Ordered lists
+ Increments the ordered list

* Unordered lists

Some Italic Text

Italic Text

Links

![Some link](http://url)

Some BoldText

Bold Text

Latex

$inline_expr$
$$expr_on_own_line$$

IPython/Jupyter Magic Commands

- Numerous "special" commands, often called magic commands, can be used within the Jupyter browser interface:

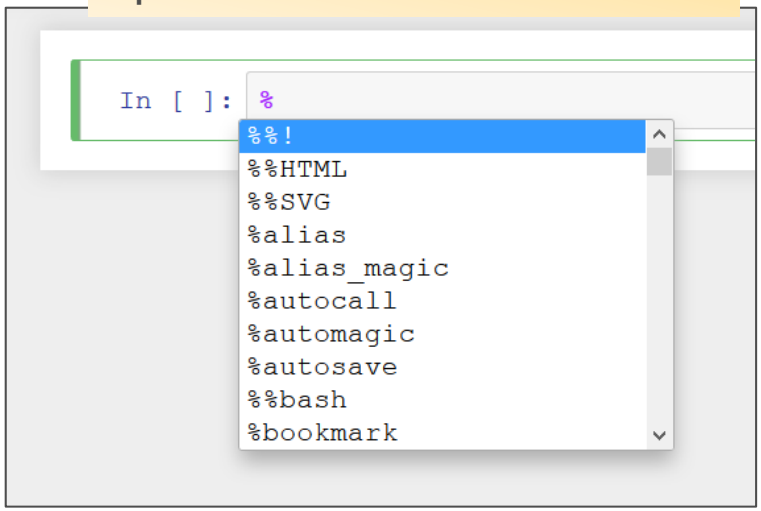
```
%matplotlib inline
```

- These commands often begin with a % or %% syntax

The following URL provides a summary of magic commands:
https://ipython.org/ipython-doc/3/interactive/magics.html

Some Magic Commands

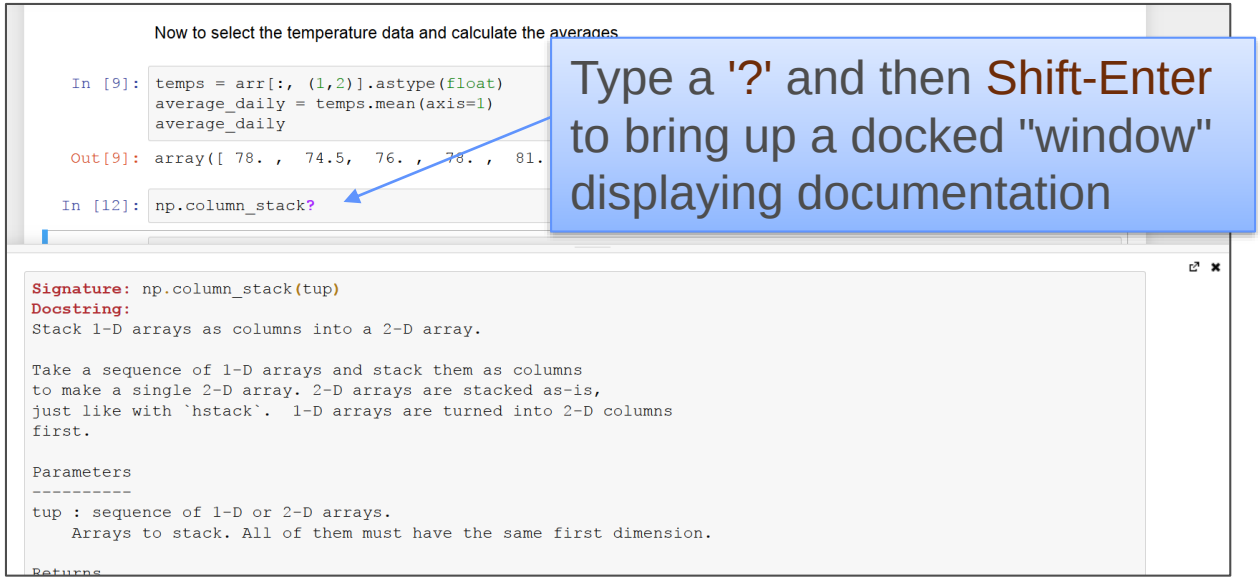
Note: pressing <Tab> brings up a list of commands

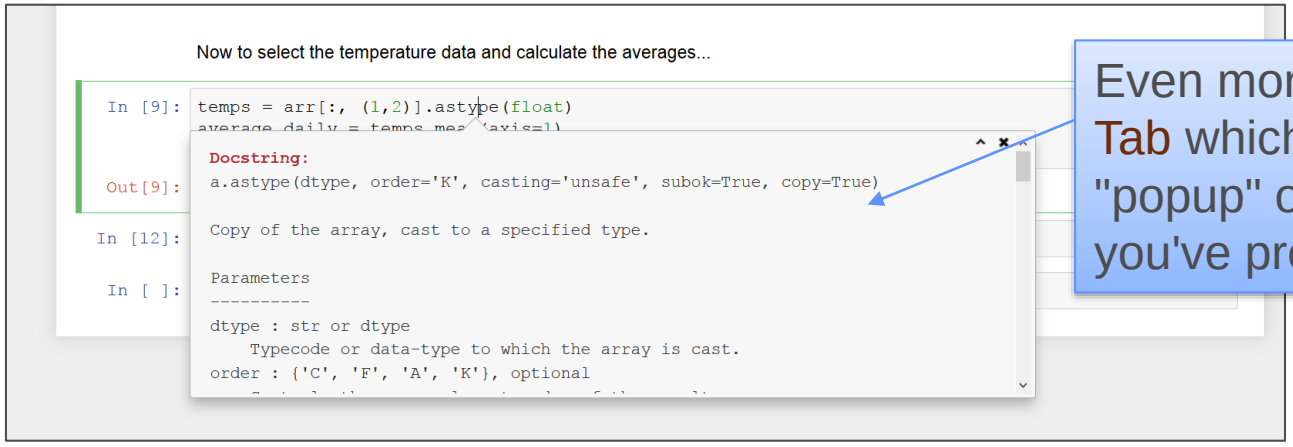


%cd	Change the working directory
%config	Run this to see which objects can be configured
%config <obj>	Run this to see which properties within an obj can be configured
%history	View a history of commands
%load <file>	Loads a file into the cell
%matplotlib	Configure matplotlib features
%pprint	Toggle pretty printing
%precision 3	Sets the output precision
%quickref	Magic Command Quick Reference
%sx dir .	Perform a system command
!!dir .	!! performs a system command also
%timeit <stmt>	Performs a Python timeit execution
%%capture	Captures stdout, stderr
%%writefile [-a] <file>	Write contents of cell to a file

Code Assistance within Jupyter

- Jupyter supports viewing "popup" code assistance:





What Are NumPy and SciPy?

- NumPy is a Python-based framework providing multi-dimensional array creation and manipulation tools

- SciPy is a set of extension frameworks that can support and aid NumPy:
  - SciPy
  - Matplotlib
  - IPython
  - Pandas
  - Sympy
  - Nose

Python Distributions That Include the SciPy Stack	
Anaconda	400+ Math/science packages
Canopy	Free & commercial
Python(x,y)	Free, incl. Spyder, Qt, SciPy Stack
WinPython	Free, Windows only
Pyzo	Free Python alternative to Matlab

The NumPy Array

- NumPy's main object is the array (ndarray type)
  - Array dimensions in NumPy are called axes
  - The number of axes is its rank

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

axes = 1 or rank = 1
Length (of axis 0): 5

```
arr2 = np.array([ [1, 2, 3, 4, 5],
                  [6, 7, 8, 9, 10] ])
```

axes = 2 or rank = 2
Length of axis 0: 2
Length of axis 1: 5

```
arr3 = np.array([ [1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12] ])
```

axes = 2 or rank = 2
Length (of axis 0): 4
Length (of axis 1): 3
Shape: (4, 3)

Array Terms (Definitions)

- An ndarray (NumPy array type) has several attributes:
  - ndarray.ndim number of axes or dimensions or rank
  - ndarray.shape length of array in each dimension
    - For m rows and n columns shape will be (m, n)
  - ndarray.size total number of elements
  - ndarray.strides number of bytes to get to the next element in that dimension

```
arr3 = np.array([ [1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12] ])

print('Shape: {0}, Size:{1}, Axes:{2}, Types:{3}, Strides:{4}'
    .format(arr3.shape, arr3.size, arr3.ndim, arr3.dtype,
            arr3.strides ))
```

Shape: (4, 3), Size: 12, Axes: 2,
Types: int32, Strides: (12, 4)

Array Values and dtype

- NumPy arrays are homogeneous

```
import numpy as np

arr10 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])

arr11 = np.array([[1, 2, 3],
                  [4, 5., 6],
                  [7, 8, 9],
                  [10, 11, 12]])

arr12 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]], dtype=float)

print(arr10.dtype, arr11.dtype,
      arr12.dtype)                          # int32 float64 float64
```

The types within a NumPy array are described (or set) via the dtype attribute

The array value types can be specified using the dtype argument

np.astype()

- astype() can be used to change the array types

```
arr10 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])
```

```
print(arr10.dtype)                          # int32
arr_new = arr10.astype(np.float64)          # new array created
print(arr_new.dtype)                        # float64
```

Creating Arrays

- Several approaches exist for creating NumPy arrays:

```
import numpy as np

arr4 = np.zeros((2,2))
print(arr4)                          # [[ 0. 0.] [ 0. 0.]]


arr5 = np.ones((2,2))
print(arr5)                          # [[ 1. 1.] [ 1. 1.]]


arr6 = np.full((2,2), 6)             # np.full(shape, fill_val)
print(arr6)                          # [[ 6. 6.] [ 6. 6.]]


arr7 = np.eye(2)
print(arr7)                          # 2x2 identity matrix


arr8 = np.empty((2,2))
print(arr8)                          # 2x2 random numbers
```

Other Array Creation Techniques

- linspace() - creates an array with **num_items** number of *evenly spaced* elements

```
np.linspace(start, end, num_items)
```

end IS included!

```
arr14 = np.linspace(0, 20, 5)        # [ 0. 5. 10. 15. 20. ]
```

- arange() - generates array values—similar to Python's range()

```
np.arange(start, end, step)
```

end NOT included!

```
arr15 = np.arange(0, 15, 3)                  # [0 3 6 9 12]
```

- diag() - creates a diagonal array

```
np.diag([vector_values])
```

```
arr16 = np.diag(np.arange(1, 6, 2))   # [[1 0 0] [0 3 0] [0 0 5]]
```

Combining Arrays: Stacking

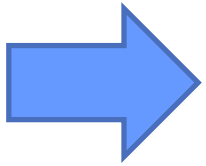
- np.stack([arrs], axis) - joins arrays along a specified axis

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = np.stack([v1, v2])
```

When not specified, default axis is 0

```
[1 2 3]
```
v1

```
[4 5 6]
```
v2

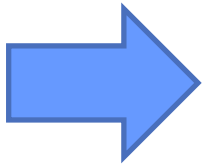


```
[[1 2 3]
 [4 5 6]]
```
result

```
result = np.stack([v1, v2], axis=1)
```

```
[1 2 3]
```
v1

```
[4 5 6]
```
v2



```
[[1 4]
 [2 5]
 [3 6]]
```
result

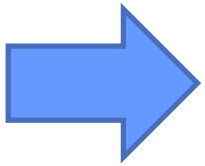
Array Stacking *(continued)*

- np.column_stack([arrs]) - stacks 1D arrays as columns:

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = np.column_stack([v1, v2])
```

```
[1 2 3]
```
v1

```
[4 5 6]
```
v2



```
[[1 4]
 [2 5]
 [3 6]]
```
result

- np.hstack([arrs]) - stacks arrays in seq. horizontally *(column wise)*

```
np.hstack((v1, v2))
```

```
[1 2 3 4 5 6]
```

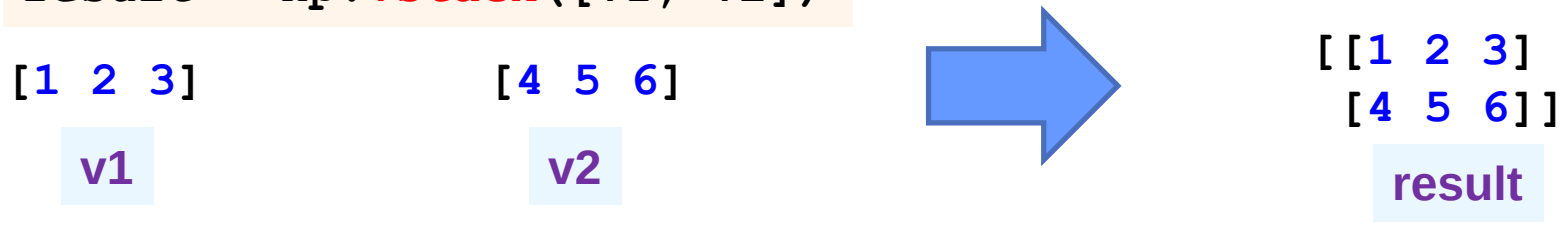
```
np.hstack((result, result))
```

```
[[1 4 1 4]
 [2 5 2 5]
 [3 6 3 6]]
```

Array Stacking *(continued)*

- np.vstack() - stacks arrays in seq. vertically *(row wise)*

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = np.vstack([v1, v2])
```

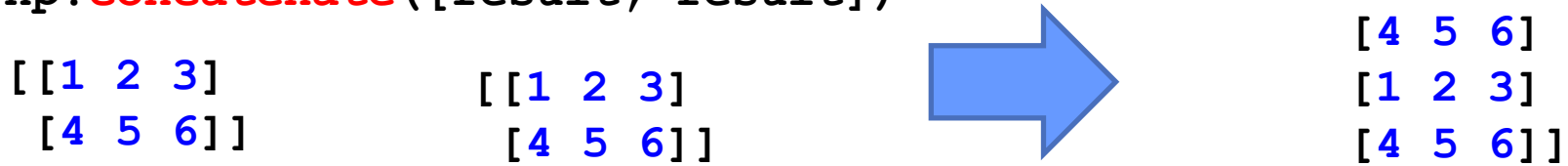


- np.concatenate([arrs], axis)

The previous stacking methods are all convenience methods that wrap concatenate()

```
np.concatenate([v1, v2])
```


```
np.concatenate([result, result])
```



Array Shifting

- np.roll(arr, n, axis) - causes the elements of the array to rotate *n* elements along an axis
  - By default, arr is flattened first if no axis is specified

```
v1 = np.array([1, 2, 3])
```

```
[1 2 3]
```

v1

```
a1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9])
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

a1

```
np.roll(v1, 1)
```

```
[3 1 2]
```

Shifts right one

```
np.roll(v1, -1)
```

```
[2 3 1]
```

Shifts left one

```
np.roll(a1, -1, axis=0)
```

```
[[4 5 6]
 [7 8 9]
 [1 2 3]]
```

Shifts rows up one

```
np.roll(a1, 1, axis=1)
```

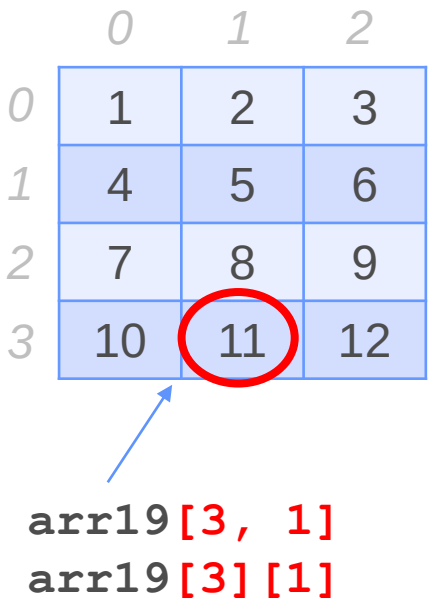
```
[[3 1 2]
 [6 4 5]
 [9 7 8]]
```

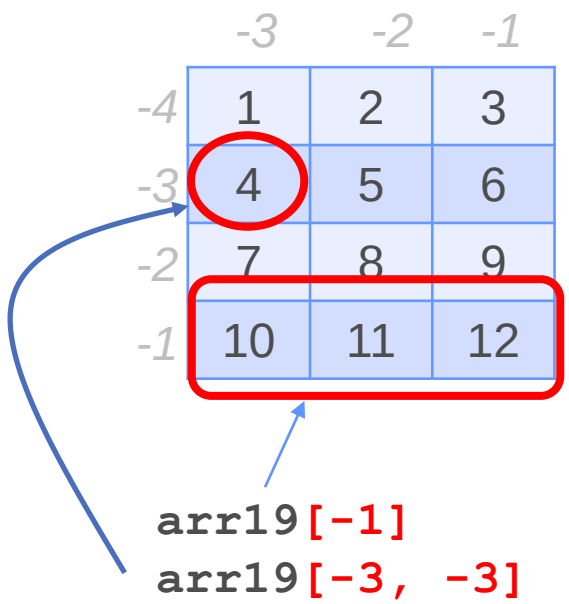
Shifts columns right one

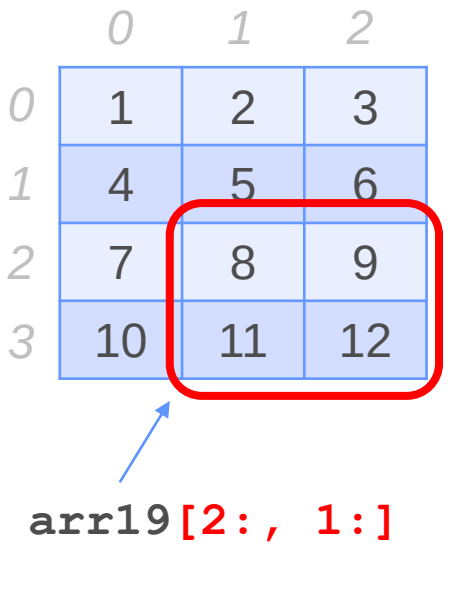
Accessing Values

- Access arrays using subscripts
  - A single pair of square brackets works
  - Use Python's slice notation also

```
arr19 = np.array([ [1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12] ])
```



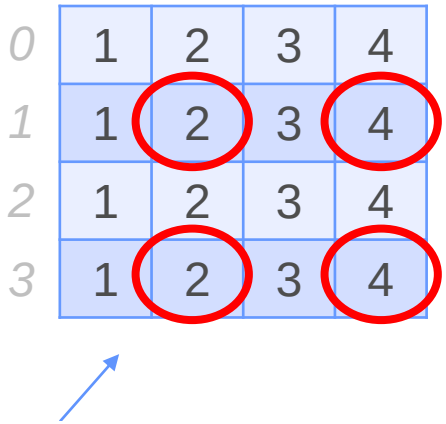




More with Slicing: Stepping

- np.tile(arr, reps) - repeats an array reps times
  - Reps can be an array indicating the number of repeats in each dimension

```
arr20 = np.tile([1, 2, 3, 4], [4, 1])
```



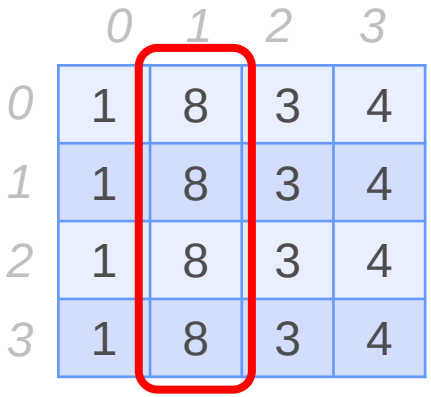

```
arr20[1::2, 1::2]                    # [[2 4] [2 4]]
```

- Slicing operations create a *view* of the original array (a copy of the slice is *not* made)

Selecting, Modifying Columns

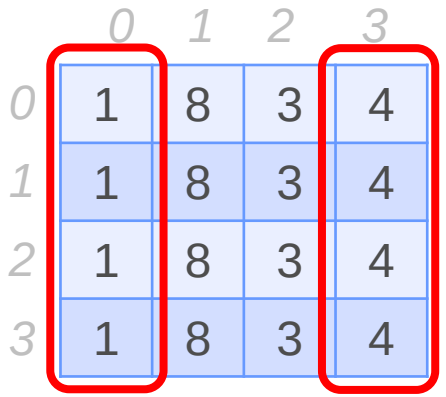
Use slicing to modify multiple values at once

```
arr20[:,1] = 8
```



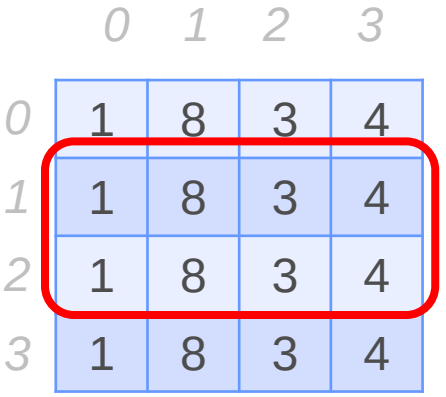
Slicing & selecting specific column indices

```
arr20[:, [0,3]]
```



Selecting specific rows and all columns

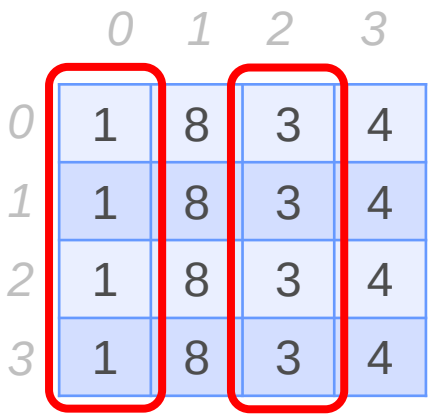
```
arr20[(1, 2), :]
```



Selecting Using Booleans

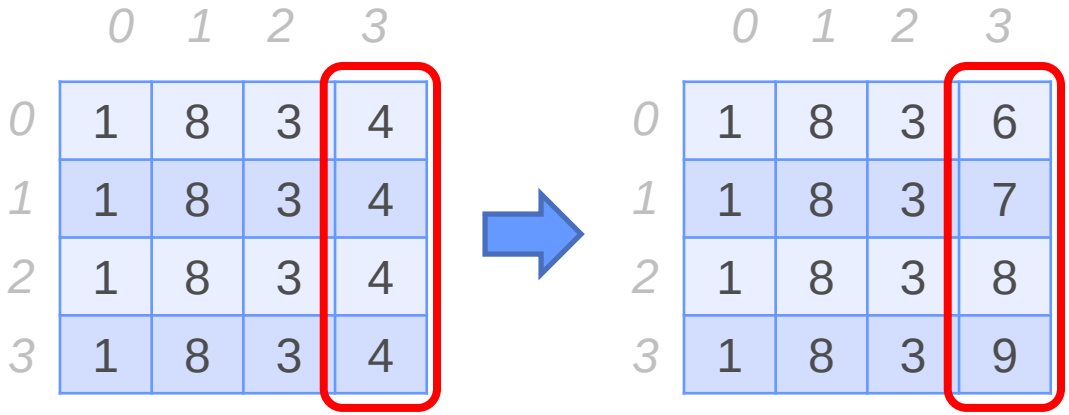
Slicing & selecting via Boolean-valued arrays

```
arr20[:,
  np.array([True, False, True, False])]
```



Slicing & selecting via Boolean-valued arrays

```
arr20[:, np.array([False, False, False, True])]
                                = [[6], [7], [8], [9]]
```



Array Element Math Operations

- NumPy arrays support basic arithmetic operations

```
arr22 = np.array([[1, 2], [3, 4]])
arr23 = np.array([[5, 6], [7, 8]])
```

```
[[ 1  2 ]
 [ 3  4 ]]
```

```
[[ 5  6 ]
 [ 7  8 ]]
```

```
arr24 = arr22 + arr23
```

Same as np.add(arr22, arr23)

```
[[  6   8]
 [ 10 12]]
```

```
arr26 = arr22 - arr23
```

Same as np.subtract(arr22, arr23)

```
[[ -4  -4]
 [ -4  -4]]
```

```
arr27 = arr22 * arr23
```

Same as np.multiply(arr22, arr23)

```
[[  5  12]
 [ 21  32]]
```

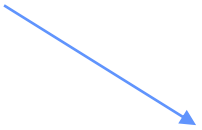
Do arrays have to be the same size?

Transpose

- np.transpose() - transpose an array, also array.T

```
arr35 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])
```

```
print(arr35.shape)                                    # (4, 3)
arr36 = arr35.transpose()
print(arr36.shape)                                    # (3, 4)
print(arr36)
```



```
[[ 1  4  7 10]
 [ 2  5  8 11]
 [ 3  6  9 12]]
```

Summary

- NumPy provides a powerful array type called an ndarray
- Its numerous operations provide the capability to manipulate, transform, and reshape data
- The NumPy array is used as a foundational data structure with other frameworks
  - It integrates nicely with other tools such as Matplotlib and Pandas (each discussed in later chapters)

Your Turn! - Task 1-1

Part 1

- Create a "checkerboard" array (8 x 8) filled with 1's and 0's

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

Part 2

- *(Advanced)*: make it work for an n x n board

	0	*1*	*2*
0	0	1	0
1	1	0	1
2	0	1	0

Hint: you may find **np.logical_not(arr)** useful in flipping 1's to 0's and vice versa

Chapter 2

More with

NumPy

Additional Features Utilizing NumPy

Overview

NumPy Functions

Reshaping

Boolean Indexing

Sorting and Iterating Arrays

Reading Data from Files

NumPy Functions *(1 of 2)*

- NumPy ships with dozens of helpful functions:

abs()
absolute()
accumulate()
add()
all()
allclose()
alltrue()
angle()
any()
append()
arange()
arccos()
arccosh()
arcsin()
arcsinh()
arctan()
arctan2()
arctanh()
argmax()
argmin()
argsort()
array()
arrayrange()
array_split()
asarray()
asanyarray()
asmatrix()
astype()
atleast_1d()
atleast_2d()
atleast_3d()
average()
beta()
binary_repr()
bincount()
binomial()
bitwise_and()
bitwise_or()
bitwise_xor()
bmat()
broadcast()
bytes()
c_[]
cast[]()
ceil()
choose()
clip()
column_stack()
compress()
concatenate()
conj()
conjugate()
copy()
corrcoef()
cos()
cov()
cross()
cumprod()
cumsum()
delete()
det()
diag()
diagflat()
diagonal()
diff()
digitize()
dot()
dsplit()
dstack()
dtype()
empty()
empty_like()
eye()
fft()
fftfreq()
fftshift()
fill()
finfo()
fix()
flat
flatten()
fliplr()
flipud()
floor()
fromarrays()
frombuffer()
fromfile()
fromfunction()
fromiter()
generic
gumbel()
histogram()
hsplit()
hstack()
hypot()
identity()
ifft()
imag
index_exp[]
indices()
inf
inner()
insert()
inv()
iscomplex()
iscomplexobj()
item()
ix_()
lexsort()
linspace()
loadtxt()
logical_and()
logical_not()
logical_or()

NumPy Functions *(2 of 2)*

logical_xor()
logspace()
lstsq()
mat()
matrix()
max()
maximum()
mean()
median()
mgrid[]
min()
minimum()
multiply()
nan
ndenumerate()
ndim
ndindex()
newaxis
nonzero()
ogrid()
ones()
ones_like()
outer()
permutation()
piecewise()
pinv()
poisson()
poly1d()
polyfit()
prod()
ptp()
put()
putmask()
r_[]
rand()
randint()
randn()
random_sample()
ranf()
ravel()
real()
recarray()
reduce()
repeat()
reshape()
resize()
rollaxis()
round()
rot90()
s_[]
sample()
savetxt()
searchsorted()
seed()
select()
shape
shuffle()
slice()
solve()
sometrue()
sort()
split()
squeeze()
std()
sum()
svd()
swapaxes()
T
take()
tensordot()
tile()
tofile()
tolist()
trace()
transpose()
tri()
tril()
trim_zeros()
triu()
typeDict()
uniform()
unique()
unique1d()
vander()
var()
vdot()
vectorize()
view()
vonmises()
vsplit()
vstack()
weibull()
where()
zeros()
zeros_like()

Aggregation Functions

- NumPy provides functions for performing tasks on *all selected elements* within the array

```
arr31 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])
```

```
np.sum(arr31)              # 78
arr31.sum()                # 78
```
Sums all elements

```
np.mean(arr31[:,0])        # 5.5
arr31[:,0].mean()          # 5.5
```
Average of column 0

```
np.var(arr31[1])           # 0.667
arr31[1].var()             # 0.667
```
Variance of row 1

```
np.std(arr31[1])           # 0.816
arr31[1].std()             # 0.816
```
Standard deviation of row 1

```
np.prod(arr31[:2, :2])     # 40
arr31[:2, :2].prod()       # 40
```
Product of upper left 4 nums

```
arr31.prod(axis=1)         # [6 120 504 1320]
```
Product of each row

Your Turn! - Task 2-1

- Use Jupyter Notebook for this task
- Create (manually) a NumPy array from the following data:

Day	Temp (High)	Temp (Low)	Humidity	Winds	Outlook	Red Flag
1	88	68	25	10	Sunny	False
2	84	65	31	5	Cloudy	False
3	86	66	32	5	Light Rain	False
4	89	67	26	5	Rain	False
5	92	70	22	10	Sunny	False
6	95	71	18	20	Sunny	True
7	94	69	27	10	Sunny	False

- Determine the shape, size, strides, and number of dimensions of the array
- Determine the average daily temperature and the average temperature for the week

Reshaping Arrays

- ndarray.reshape(shape) - can reshape an array's dimensions

```
arr32 = np.arange(1, 17).reshape((4,4))
```

New shape

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

arr32

- np.ravel(array) - treats a multidimensional array like a 1-D array

```
np.ravel(arr32)[8]          # 9
```

Reshaping Arrays *(continued)*

- np.squeeze(shape) - removes any additional axes of length 1 from an array:

```
arr33 = np.array([[[[0], [1], [2]], [[3], [4], [5]]]])

arr33.shape                          # (1, 2, 3, 1)
arr34 = arr33.squeeze()              # [[0 1 2] [3 4 5]]
arr34.shape                          # (2, 3)
```

Boolean Indexing (Masking)

- ndarray values can be *Boolean indexed* yielding new "filtered" arrays

```
arr = np.arange(1, 7)
```

[1 2 3 4 5 6]

```
mask = arr <= 3
```

[True True True False False False]

```
new_arr = arr[mask]
```

[1 2 3]

```
arr2 = np.arange(11, 17)
```

[11 12 13 14 15 16]

```
new_arr = arr2[mask]
```

[11 12 13]

Note that ~arr performs a Boolean negation of arr:
`~np.array([True, False])` → `[False, True]`

Sorting Based on Values

- To sort an array based on a specific column:

```
arr = np.roll(np.arange(1, 10).reshape((3, 3)), 2)
```

```
[[8 9 1]
 [2 3 4]
 [5 6 7]]
```

Specifies the first column to sort on

```
sort_by_col0 = arr[arr[:, 0].argsort()]
```

```
[[2 3 4]
 [5 6 7]
 [8 9 1]]
```

Must take the results of the sort and plug it back into the original array to cause the sort to occur

```
sort_by_col2 = arr[arr[:, 2].argsort()]
```

```
[[8 9 1]
 [2 3 4]
 [5 6 7]]
```

Sorts on the third column

Reading Data Using genfromtxt()

- *genfromtxt()* can load data from a source file

```
np.genfromtxt(fname, dtype='float', delimiter=None,
              converters=None, missing_values=None,
              filling_values=None, usecols=None,
              skip_header=0, skip_footer=0)
```

filling_values - default values to use when data is missing
converters - set of functions that can convert data to desired values

- Basic example:

tennis.dat

```
Roger Federer,9.3,45
Rafael Nadal,7.4,25
Maria Sharapova,5.1,22
```

```
result = np.genfromtxt('tennis.dat', delimiter=',',
                       usecols=(1,2), dtype=float )

print(result)
print(result.dtype)
```

```
[[9.3 45.] [7.4 25.] [5.1 22]]
float64
```

Using **dtype=None** causes NumPy to guess column types (here it would read in as structured data)

Reading As Structured or Object Data

- Since the data contains different data types, the following example can be read two ways:

city_data1.dat

```
Colorado Springs,6172,431000.0
Phoenix,1132,1488000.0
Raleigh,437,423000.0
Milwaukee,723,599000.0
Seattle,429,634000.0
```

Indicates a 50 byte string, 4-byte int, and 8-byte float will be read as one structure

  - As *structured data*

```
data = np.genfromtxt('city_data1.dat', delimiter=',', dtype=('|U50', '<i4', '<f8'))
```

data.shape = (5,)

```
[ ('Colorado Springs', 6172,   431000.)
  ('Phoenix', 1132,  1488000.)
  ('Raleigh',  437,   423000.)
  ('Milwaukee',  723,   599000.)
  ('Seattle',  429,   634000.) ]
```

  - As objects

```
data = np.genfromtxt('city_data1.dat', delimiter=',', dtype=object)
```

data.shape = (5, 3), data.dtype=object

NumPy *dtypes*

- Structured data types can be indicated using Python, NumPy, or shorthand values:
  - Legal dtype values in structures include:

Note:
< (little-endian) LSB first
> (big-endian) MSB first
| (vertical bar) byte order doesn't apply

b1, i1, i2, i4, i8, u1, u2, u4, u8, f2, f4, f8, c8, c16

S50 (50-byte string), U50 (unicode)

np.int8, np.int16, np.int32, np.int64, np.int128, np.float16, np.float32, np.float64, np.float128, np.float256, np.uint8, np.uint16, np.uint32, np.uint64, np.uint128

object, int, float, others

```
data = np.genfromtxt('city_data1.dat', delimiter=',',
                     dtype=('U50', np.int32, float))

print(data.dtype)
```

```
[('f0', '<U50'), ('f1', '<i4'), ('f2', '<f8')]
```

Using *names* with Structured Data

- The names param or the dtype labels provide access to the structured data values

city_data1.dat

```
Colorado Springs,6172,431000.0
Phoenix,1132,1488000.0
Raleigh,437,423000.0
Milwaukee,723,599000.0
Seattle,429,634000.0
```

```
data = np.genfromtxt('city_data1.dat', delimiter=',',
                     names=['name', 'elev', 'pop'],
                     dtype=['|U50', '<i4', '<f8'])
```

```
print(data['name'])
```

```
['Colorado Springs' 'Phoenix' 'Raleigh' 'Milwaukee' 'Seattle']
```

```
data = np.genfromtxt('city_data1.dat', delimiter=',',
                     dtype=[('name', '|U50'), ('elev', '<i4'),
                            ('pop', '<f8')])
```

```
print(data['elev'][0])
```

```
6172
```

Handling Missing Values

- This data file has missing values
- It also consists of mixed data types

city_data2.dat

```
Colorado Springs,6172,431000.0
Phoenix,1132,1488000.0
Raleigh,437,423000.0
Milwaukee,,599000.0
Seattle,429,
```

```
data = np.genfromtxt('city_data2.dat', delimiter=',',
                     filling_values=['', -999, -1],
                     dtype=('|U50', '<i4', '<f8'))
```

```
[('Colorado Springs', 6172, 431000.0) ('Phoenix', 1132, 1488000.0)
 ('Raleigh', 437, 423000.0) ('Milwaukee', -999, 599000.0)
 ('Seattle', 429, -1.0)]
```

filling_values defines what to do with values not present when reading from the file.

NaN and Infinity

- NumPy defines NaN (np.nan) and infinity (np.inf)
  - Use np.isnan() or np.isinf() to check for these values

- To remove rows where a NaN or infinity exist:

data

```
[[ 6172.   431000.]
 [ 1132.       inf]
 [  437.   423000.]
 [  nan    599000.]
 [  429.       nan]]
```

We manually generated this value with:
`data[1, 1] = data[1, 1]/0`

```
no_nans = data[~np.isnan(data).any(axis=1)]
```

Removes NaN rows

```
[[ 6172.  431000.]
 [ 1132.      inf]
 [  437.  423000.]]
```

```
no_infs = data[~np.isinf(data).any(axis=1)]
```

Removes inf rows

```
[[ 6172.  431000.]
 [  437.  423000.]
 [   nan  599000.]
 [  429.      nan]]
```

```
neither = data[np.isfinite(data).all(axis=1)]
```

Removes NaN and inf rows

```
[[ 6172.  431000.]
 [  437.  423000.]]
```

Summary

- NumPy relies on fixed data sizes for its array elements in order to process data quickly
- For mixed data types structured data can be used
- Use genfromtxt() to read data from source files

Your Turn! - Task 2-2

- Obtain and *display the top 100 batting averages* from the provided data file
  - Read from <student_files>/resources/baseball/Batting.csv
  - Read columns 6 (atbats) and 8 (hits)
  - Calculate the batting average: $\frac{\text{hits}}{\text{atbats}}$
- Remove rows where the *atbats* is less than 502
  (this is an official MLB minimum needed to count for the batting average title)
- Remove rows where the year is prior to 1957
  (The modern era has a 162-game schedule which began in this year)
  (The second column in the data file defines the year)

Chapter 3

Introducing

Matplotlib

Making Use of Other Tools
Available for Python

Overview

Data Visualization

Introducing Matplotlib

Plotting Graphs

Figures and Axes

Data Visualization and Python

- Numerous frameworks exist for creating 2D and 3D charts:

– Matplotlib	de facto standard for 2D Python visualization
– Pandas	provides integration to work with Matplotlib
– Seaborn	wrapper for Matplotlib providing simpler code and additional charts
– Ggplot	tool that works with Pandas Data-Frames providing layered charts
– Bokeh	Python charting tool allowing for various level of control

Introducing Matplotlib

- Matplotlib is the Python de facto standard API for creating 2D graphs
  - Integrates with NumPy arrays
  - Supports numerous chart types (kinds):

Bar	Polar
Line	Streamplots
Scatter	Ellipses
Pie	Table Demo
Log	...and more...

- To render charts *inline* within Jupyter Notebook, specify the following command:

```
%matplotlib inline
```

Matplotlib Modules

- Common conventions for importing primary modules:

```
import matplotlib as mpl
```

```
import matplotlib.pyplot as plt
```

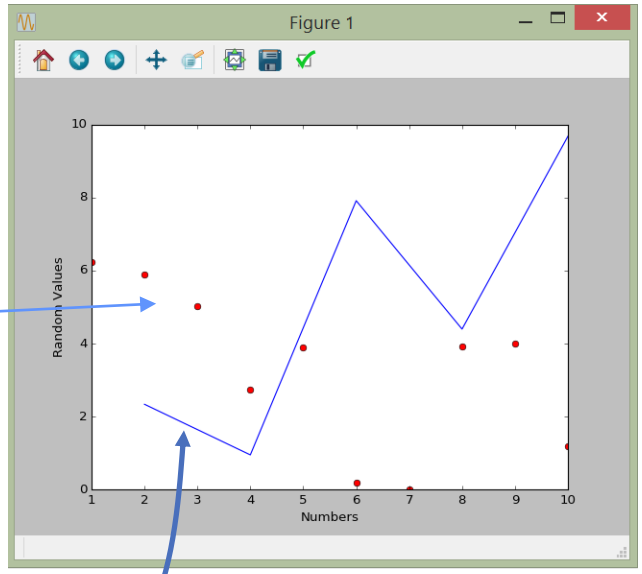
We will refer to this module as plt from now on

The **pyplot** submodule contains the primary functionality for plotting

For reference, the PyPlot API docs can be found here:
http://matplotlib.org/api/pyplot_api.html

A First Simple Plot

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.arange(1, 11)
y1 = 10 * np.random.rand(1, 10).squeeze()
plt.ylabel('Random Values')
plt.xlabel('Numbers')
plt.plot(x1, y1, 'ro')

x2 = np.arange(2, 11, 2)
y2 = 10 * np.random.rand(1, 5).squeeze()
plt.plot(x2, y2)

plt.show()
```



The Plot Method

- The plot() method supports numerous arguments:

```
plt.plot(x, y, style)
```

An array of data on the x-axis

An array of data on the y-axis

The style is a string that controls the color and markers

Chart Styles

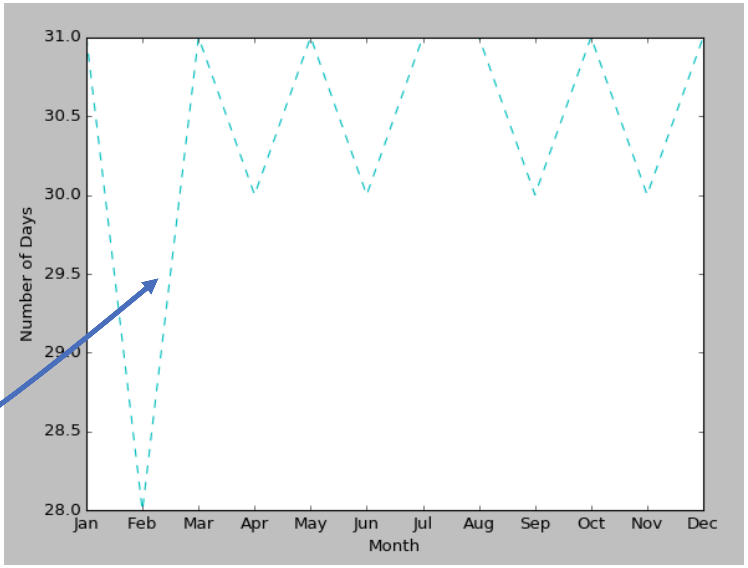
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

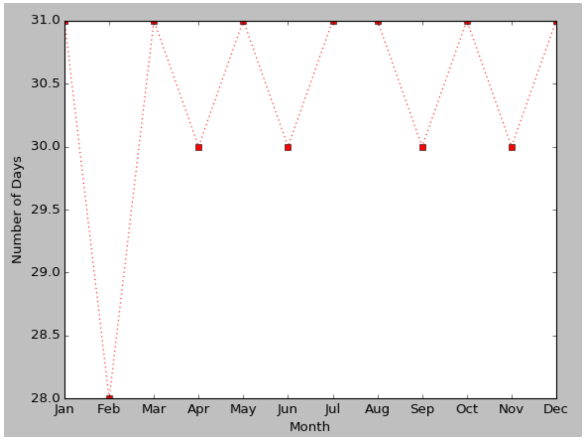
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker

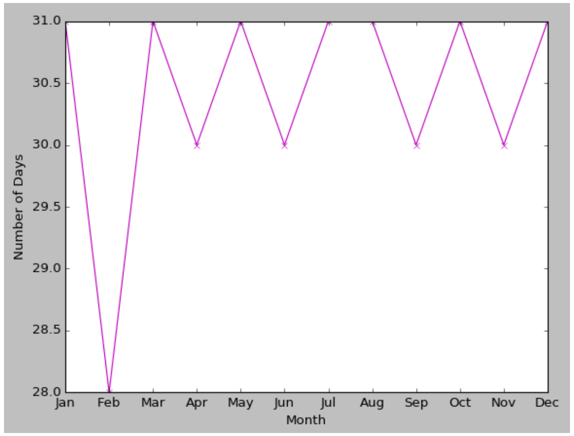
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

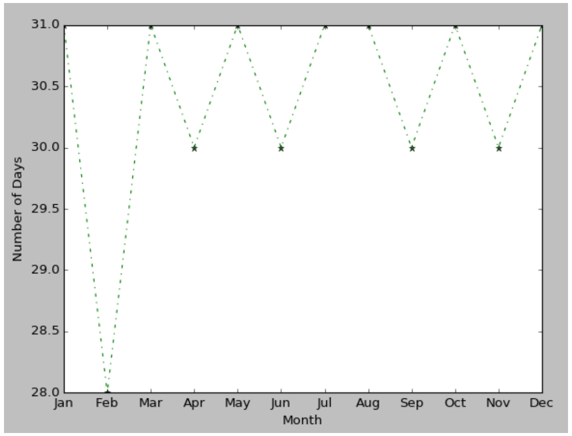
Plot Style Variations

```
x_ticks = ['Jan', 'Feb', ..., 'Dec']
x = np.arange(1, 13)
y = [31, 28, 31, 30, ..., 31]

plt.xlabel('Month')
plt.ylabel('Number of Days')
plt.xticks(x, x_ticks)
plt.plot(x, y, 'c--')
plt.show()
```









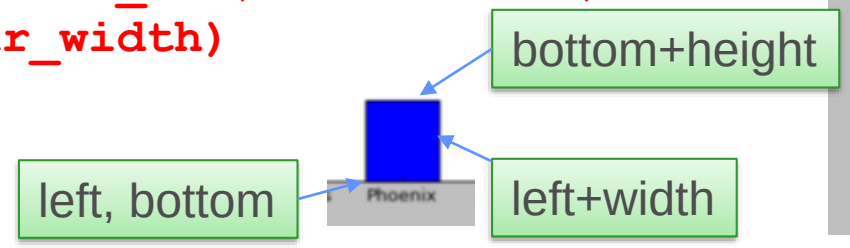

```
plt.plot(x, y, 'rs:')   plt.plot(x, y, 'mx-')   plt.plot(x, y, 'g*-.')
```

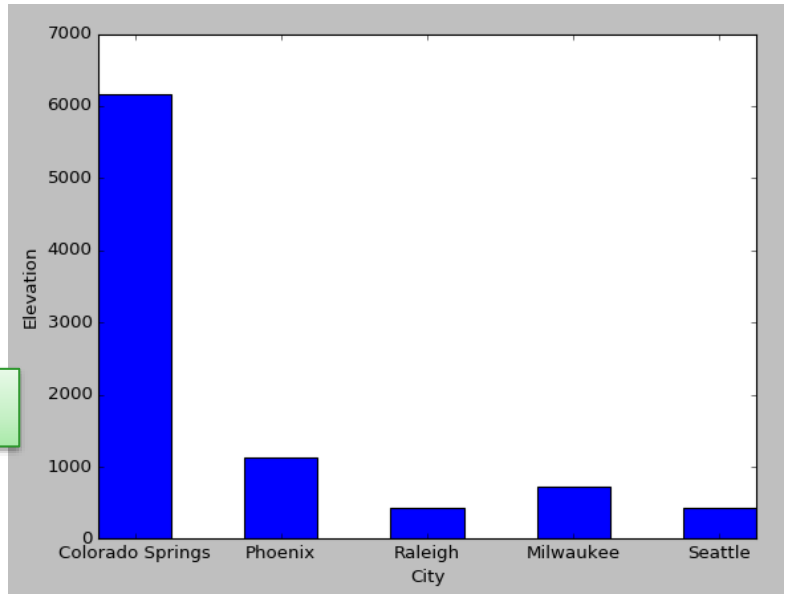
Bar Graphs

- Use the bar() method to create a bar graph:

```
plt.bar(left, height, width, bottom, ...)
```

```
cities = ['Colorado Springs', 'Phoenix',
          'Raleigh', 'Milwaukee', 'Seattle']
cities_idx = np.arange(5)
elevations = [6172, 1132, 437, 723, 429]

plt.xlabel('City')
plt.ylabel('Elevation')

bar_width = 0.5
plt.xticks(cities_idx + bar_width/2,
           cities)
plt.bar(cities_idx, elevations,
        bar_width)

plt.show()
```



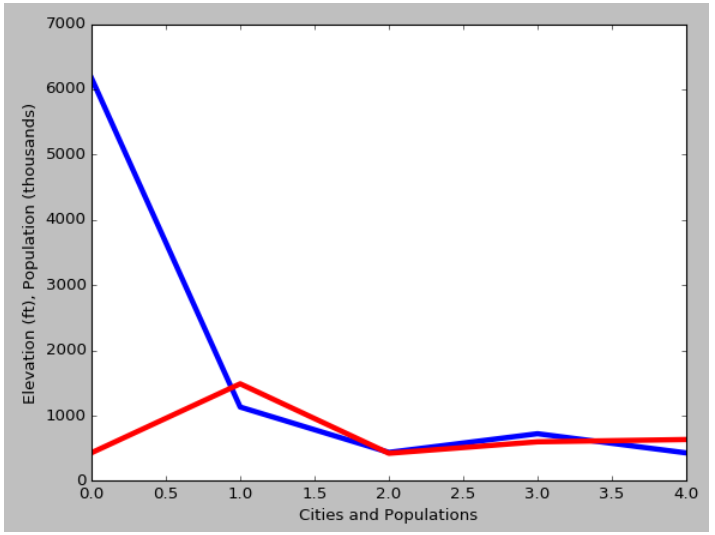


Plotting Multiple Data Sets

- The plot() method supports plotting multiple data sets by specifying additional arguments:

```
plt.plot(x1, y1, style1, x2, y2, style2, ...)
```

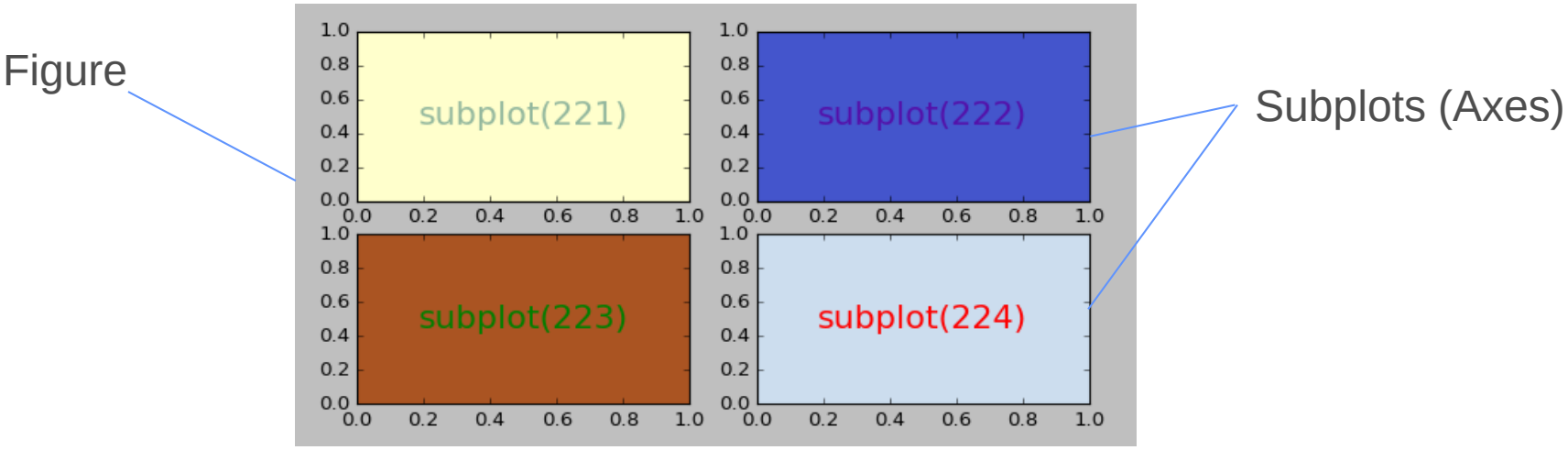
```
cities = ['Colorado Springs', ...]
cities_idx = np.arange(5)
elevations = [6172, 1132, 437, 723, 429]
populations = [431, 1488, 423, 599, 634]
plt.xlabel('Cities and Populations')
plt.ylabel('Elevation (ft), ...')
plt.plot(cities_idx, elevations,  'b-',
         cities_idx, populations, 'r-',
         linewidth=4.0)
plt.show()
```



  - For more control, however, Matplotlib supports rendering multiple plots using Figures and Axes...

Figures and Subplots

- A Matplotlib graph is made up of a Figure instance and one or more Axes instances
  - Figures represent the *entire drawing canvas*



  - Axes represent a *coordinate system for plotting data*
  - Subplots are special cases of (inherit from) Axes
  - Subplots can be placed onto a figure using a layout system: nrows ncols plot_number
    - Use subplots when laying out multiple plots on a grid

Adding Subplots to a Figure

- plt.figure() is a factory for generating figure objects

```
plt.figure(num, figsize, dpi, facecolor, edgecolor)
```

Canvas size in inches. See footnotes for support for pixel sizing.

```
figure = plt.figure(figsize=(8, 4))

sub221 = figure.add_subplot(221, facecolor='#ffffcc')
sub221.text(0.5, 0.5, 'subplot(221)', ha='center',
            va='center', fontsize=20, alpha=.5)

sub222 = figure.add_subplot(222, facecolor ='#4455cc')
sub222.text(0.5, 0.5, 'subplot(222)', ...)

sub223 = figure.add_subplot(223, facecolor ='#33aa22')
sub223.text(0.5, 0.5, 'subplot(223)', ...)

sub224 = figure.add_subplot(224, facecolor ='#ccddee')
sub224.text(0.5, 0.5, 'subplot(224)', ...)

plt.show()
```

Plotting Text

- plt.text() allows any string to be added to a plot

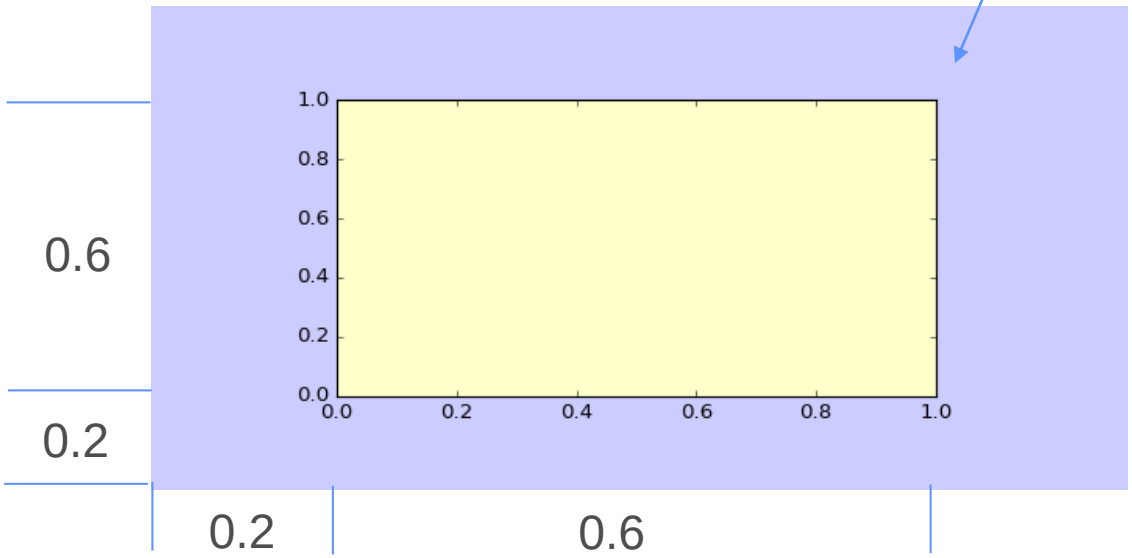
```
plt.text(x, y, s, ...)
```

x, y -	coordinates of the textbox
s -	the string of text to display
horizontalalignment, ha -	'center', 'left', 'right'
verticalalignment, va -	'center', 'top', 'bottom', 'baseline'
color -	'#hex_value' or single letter (e.g. 'b')
backgroundcolor -	same values allowed as color above
alpha -	an alpha-transparency, 0 is transparent, 1 is opaque
fontname, family -	valid font name
fontsize, size -	a pixel size or 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'
rotation -	degrees or 'horizontal' or 'vertical'
style -	'normal', 'italic'

Managing Axes

- Use add_axes() to create a new axis within a Figure

```
figure = plt.figure(figsize=(8, 4), facecolor='#ccccff')
```



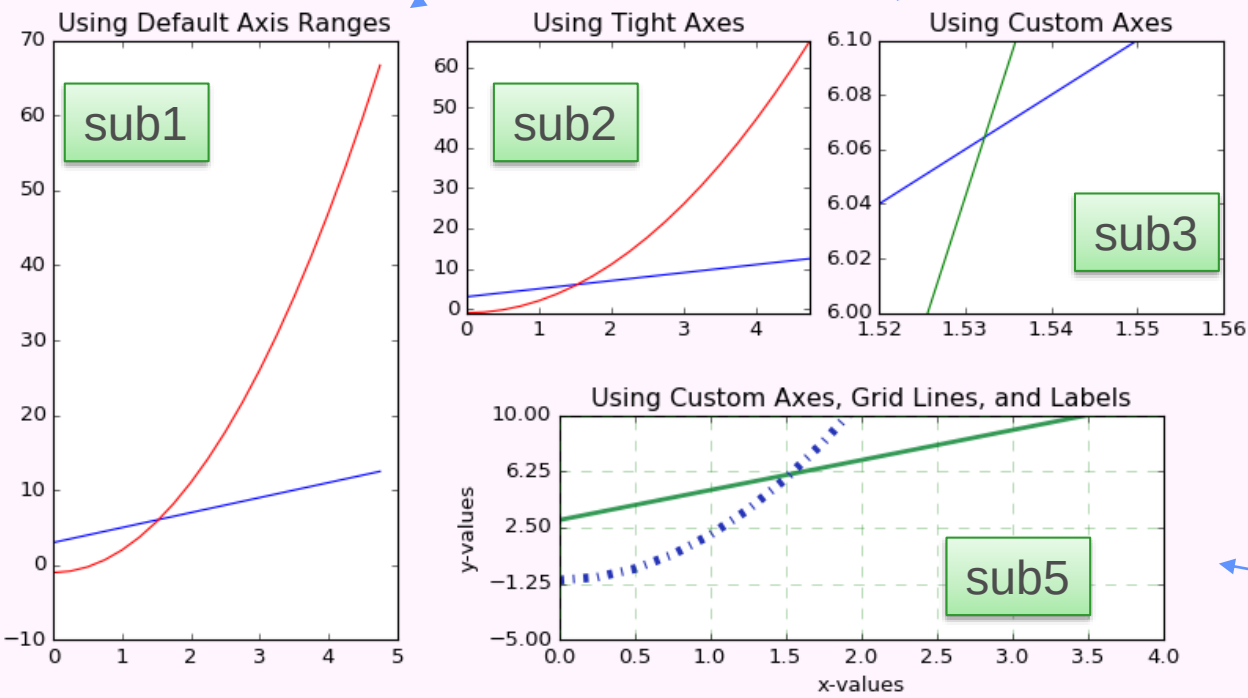

```
figure.add_axes((0.2, 0.2, 0.6, 0.6), facecolor ='#ffffcc')
plt.show()
```

(left, bottom, width, height)
All values are fractions of the figure's total size

Custom Layouts *(1 of 3)*

- Layouts can be created using *add_subplot()* and then specifying 3 scalar values
  - This provides the ability to span multiple columns or rows

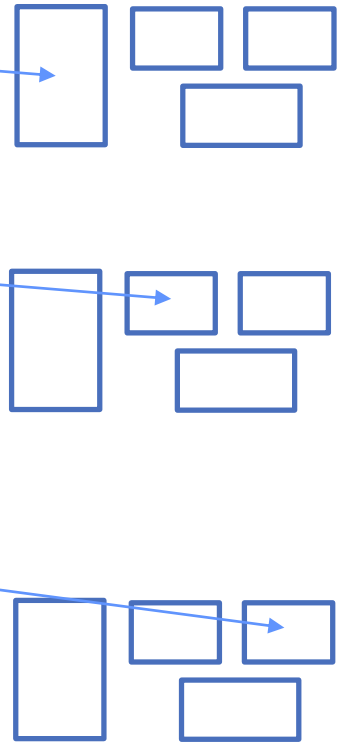
These are laid out using add_subplot() rows and columns



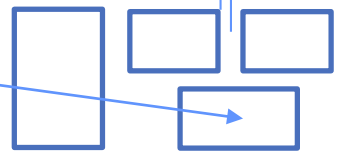
This plot is laid out using axes() and specifying l, b, w, h

Custom Layouts *(2 of 3)*

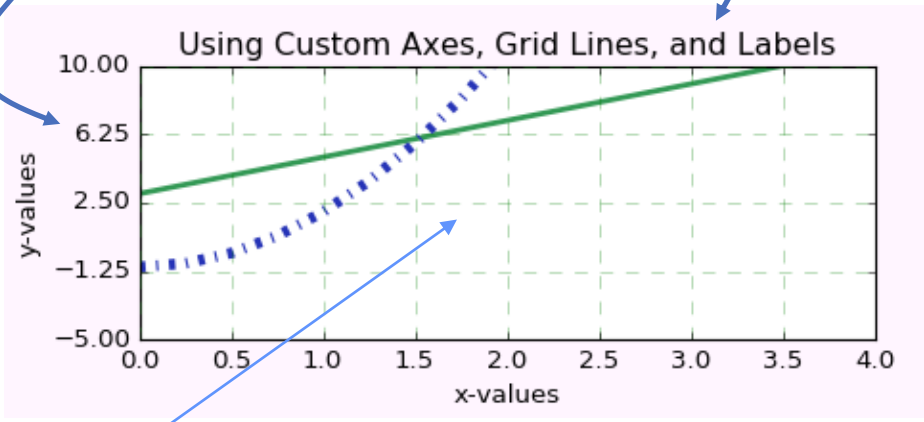
```
x = np.arange(0, 5, 0.25)
y1 = 2*x + 3
y2 = 3*x**2-1

figure = plt.figure(figsize=(12, 6), facecolor='#fff5ff')

sub1 = figure.add_subplot(2, 3, (1, 4))
sub1.plot(x, y1, 'b-', x, y2, 'r-')
sub1.set_title('Using Default Axis Ranges')

sub2 = figure.add_subplot(2, 3, 2)
sub2.plot(x, y1, 'b-', x, y2, 'r-')
sub2.axis('tight')
sub2.set_title('Using Tight Axes')

sub3 = figure.add_subplot(2, 3, 3)
sub3.plot(x, y1, 'b-', x, y2, 'g-')
sub3.set_xticks(np.linspace(1.52, 1.56, 5))
sub3.set_xlim((1.52, 1.56))
sub3.set_ylim((6.0, 6.1))
sub3.set_title('Using Custom Axes')
```



Custom Layouts *(3 of 3)*



```
sub5 = figure.add_axes((0.46, 0.1, 0.4, 0.3))
sub5.plot(x, y1, 'b-', color='#339955', lw=2.5)
sub5.plot(x, y2, 'r-', color='#2533b7', lw=5.0, ls='dashdot')
sub5.set_yticks(np.linspace(-5, 10, 5))
sub5.set_xlim((0, 4))
sub5.set_ylim((-5, 10))
sub5.set_xlabel('x-values')
sub5.set_ylabel('y-values')
```



set_xlim(), set_ylim() set the range for the x and y axes, while set_xticks() and set_yticks() set the frequency and label of the ticks

```
sub5.grid(color='g', alpha=0.5, ls='dashed', lw=0.5)
sub5.set_title('Using Custom Axes, Grid Lines, and Labels')
```

Helpful Line Properties

- The previous slide illustrated a few line properties, here are some valid values that may be used:

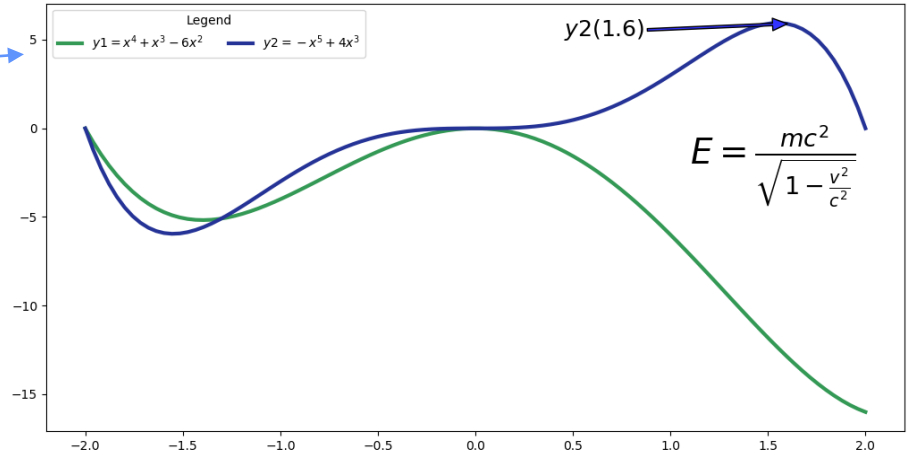
```
plt.plot(xdata, ydata, ...)
```

color, c - any '#hex_value', or single letter ('r'),
linestyle, ls - 'solid', 'dashed', 'dotted', '-', '--', '-.', '.', or other valid symbol
linewidth, lw - float value for the width
alpha - 0.0 (transparent) to 1.0 (opaque)
label - text placed on the line
fillstyle - 'full', 'left', 'right', 'bottom', 'top', 'none'
marker - any marker style described on an earlier slide (e.g. 'v' or 'o')
drawstyle - 'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'
markersize, ms - float value for the size of the markers
markerfacecolor - a color for the markers
markeredgecolor - a color for the edges of the markers
markeredgewidth - width of the line that draws the markers

Legends

- Use plt.legend() to customize a legend

A title, two columns, and optimized positioning creates this legend.

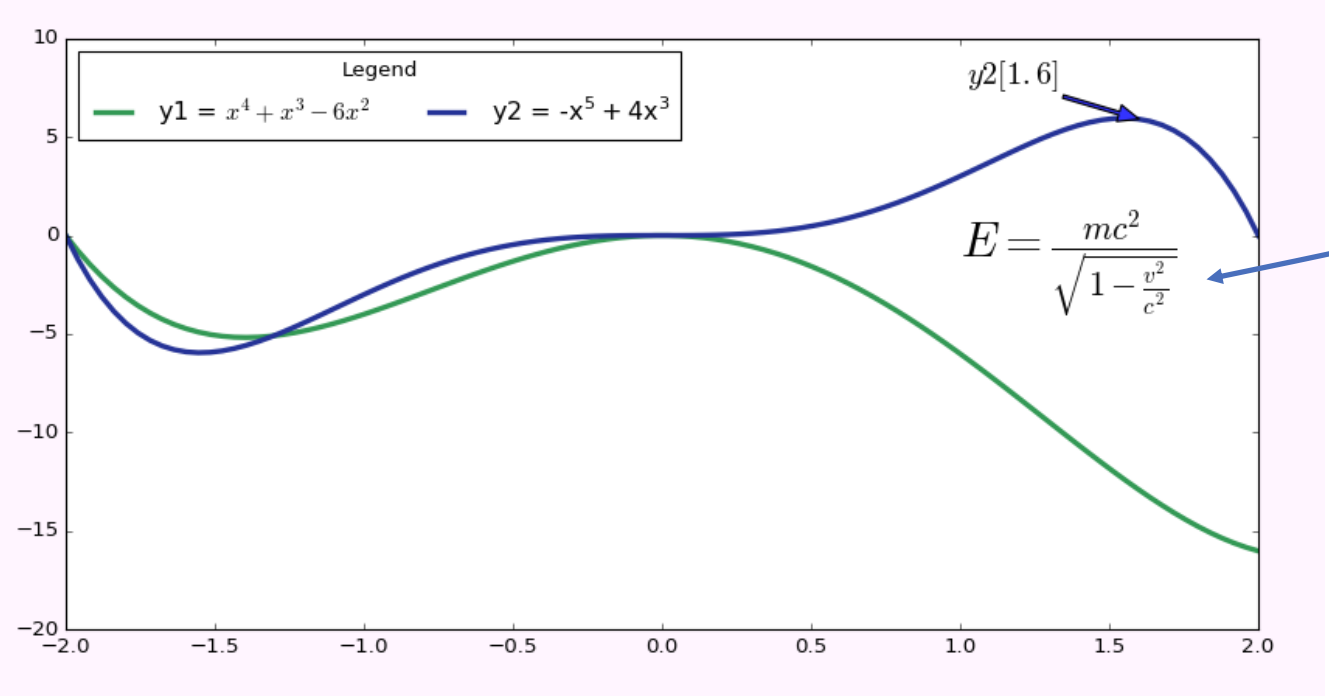



```
x = np.linspace(-2, 2, 100)
y1 = x**4 - x**3 - 6*x**2
y2 = -x**5 + 4*x**3

figure = plt.figure(figsize=(12, 6), facecolor='#fff5ff')

sub1 = figure.add_axes((0.1, 0.1, 0.8, 0.8))
sub1.plot(x, y1, color='#339955', lw=3.0, ls='solid')
sub1.plot(x, y2, color='#253397', lw=3.0, ls='solid')
sub1.legend([r'$y1 = x^4 + x^3 - 6x^2$',
             r'$y2 = -x^5 + 4x^3$'],
            loc='best', title='Legend', ncol=2)
```

LaTeX, a special markup for displaying numerical values.

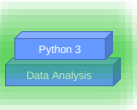
Annotations

```
sub1.annotate(s=r'$E = \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}}$',
              xy=(0.7, 0.6), xycoords='figure fraction',
              fontsize=28)
```



Annotations can be used to give plots additional flare or highlight noteworthy points

```
sub1.annotate(r'$y2(1.6)$',
              xy=(1.6, -1.6**5 + 4*1.6**3), xycoords='data',
              xytext=(-100, +20), textcoords='offset points',
              fontsize=18, arrowprops=dict(facecolor='#3333ff',
                           headwidth=10, frac=0.3, width=2))
```



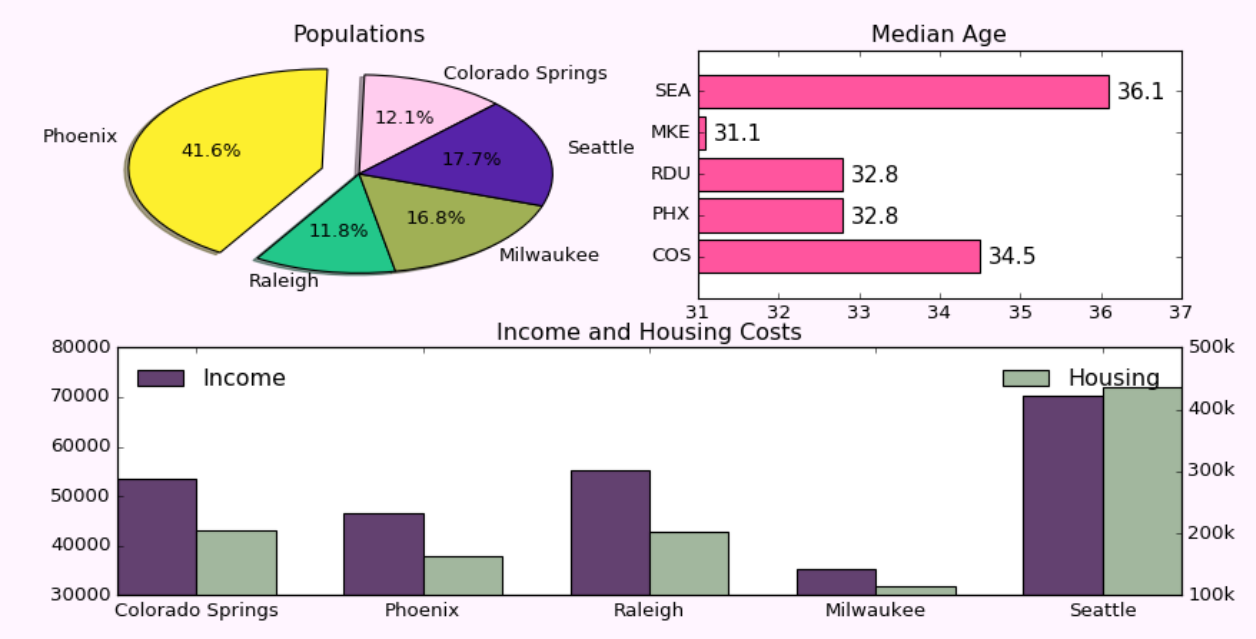
savefig

- Save figures using figure.savefig():

```
figure.savefig(fname, dpi=None, facecolor='w',
               edgecolor='w', format=None
               orientation='portrait')
```

```
figure.savefig('legendary.png')
```

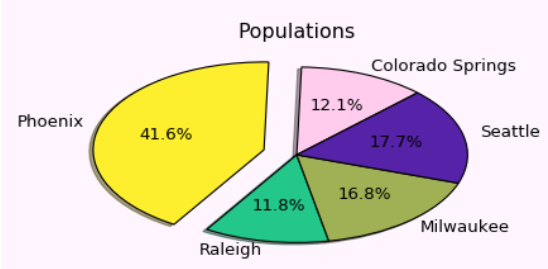
Other Chart Examples *(1 of 3)*



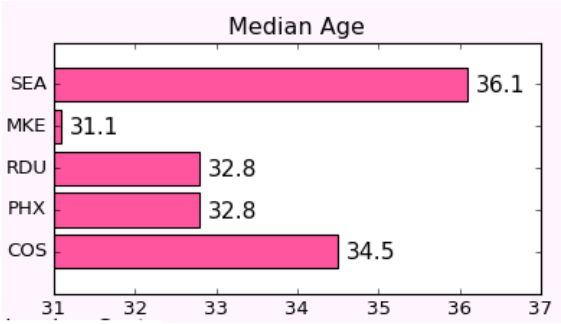

```
figure = plt.figure(figsize=(12, 6), facecolor='#fff5ff')

cities = ['Colorado Springs', 'Phoenix', 'Raleigh',
          'Milwaukee', 'Seattle']
cities_abbr = ['COS', 'PHX', 'RDU', 'MKE', 'SEA']
elevations = [6172, 1132, 437, 723, 429]
populations = [431000, 1488000, 423000, 599000, 634000]
median_age = [34.5, 32.8, 32.8, 31.1, 36.1]
median_income = [53550, 46601, 55170, 35186, 70172]
median_house = [205600, 162300, 202800, 113900, 436600]
```

Other Chart Examples *(2 of 3)*

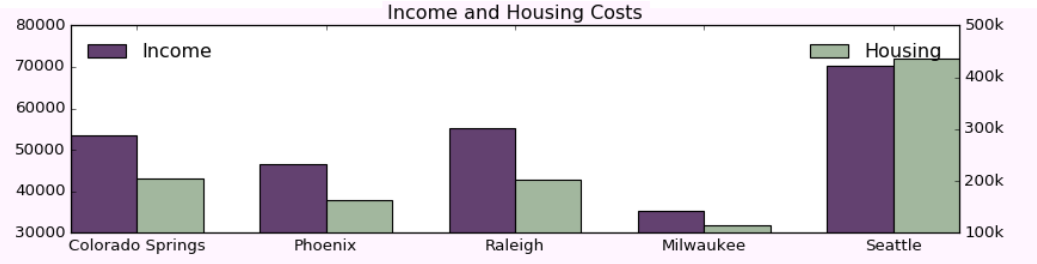
```
sub1 = figure.add_subplot(221)
colors = ['#ffccee', '#fdef2e', '#23c78a', '#a0b055', '#5623a8']
sub1.pie(populations, labels=cities,
         colors=colors,
         explode=[0, 0.2, 0, 0, 0],
         autopct='%1.1f%%',
         shadow=True, startangle=45)
sub1.set_title('Populations')
```




```
sub2 = figure.add_subplot(222)
y_ticks = np.arange(len(cities))
sub2.barh(y_ticks, median_age, height=0.8,
          align='center', color='#ff559e')
sub2.set_yticks(y_ticks)
sub2.set_xlim((31, 37))
sub2.set_yticklabels(cities_abbr)
sub2.set_title('Median Age')
for loc, age in enumerate(median_age):
    sub2.annotate(str(age), xy=(median_age[loc], loc),
                  xycoords='data', xytext=(+5, -5),
                  textcoords='offset points', fontsize=14)
```



Other Chart Examples *(3 of 3)*



Grouped Bar, Twin Axis

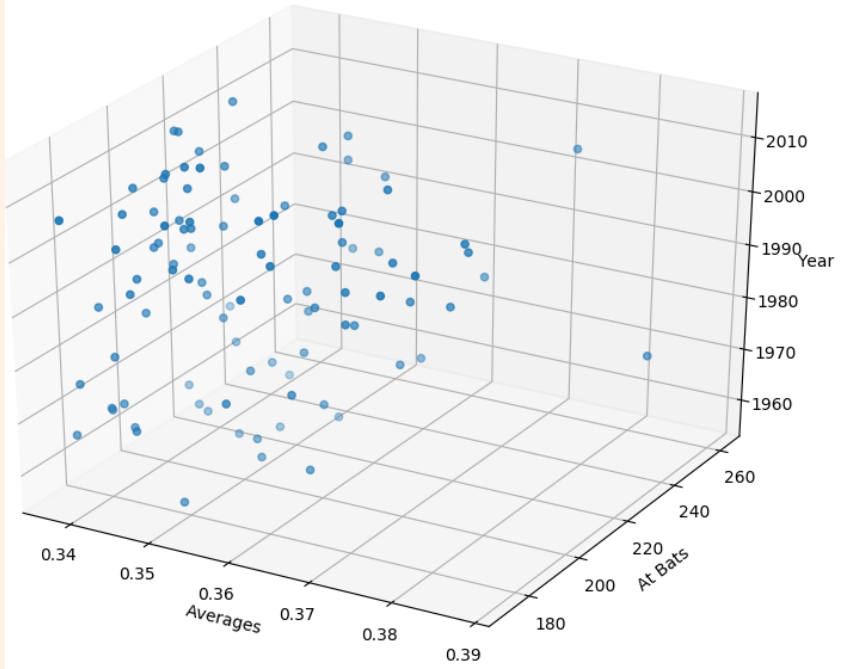
```
sub3 = figure.add_subplot(2, 2, (3, 4))
bar_width = 0.35
x_ticks = np.arange(len(cities))
sub3.bar(x_ticks, median_income, width=bar_width, color='#634170')
sub3.set_xticks(x_ticks + bar_width)
sub3.set_xticklabels(cities)
sub3.set_ylim(30000, 80000)
sub3.set_title('Income and Housing Costs')
sub3.legend(['Income'], loc='upper left', frameon=False)

sub3b = sub3.twinx()
sub3b.bar(x_ticks + bar_width, median_house,
          width=bar_width, color='#a2b79e')
sub3b.set_ylim(100000, 500000)
sub3b.set_yticks(np.arange(100000, 600000, 100000))
sub3b.set_yticklabels(['100k', '200k', '300k', '400k', '500k'])
sub3b.legend(['Housing'], loc='upper right', frameon=False)
```

Creates axis on other side of x

3D Plots

- mpl_toolkits (included within Anaconda) provides additional tools for creating 3D plots
  - Plots include:
    - 3D Scatter
    - 3D Line
    - Surface
    - Wireframe
    - Polygon
    - Contour

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d

...

fig = plt.figure(figsize=(8, 6))
ax = axes3d.Axes3D(fig)
x = data[-1:-100:-1, 3]
y = data[-1:-100:-1, 2]
z = data[-1:-100:-1, 0]
ax.scatter(x, y, z)
ax.set_xlabel('Averages')
ax.set_ylabel('At Bats')
ax.set_zlabel('Year')
plt.show()
```

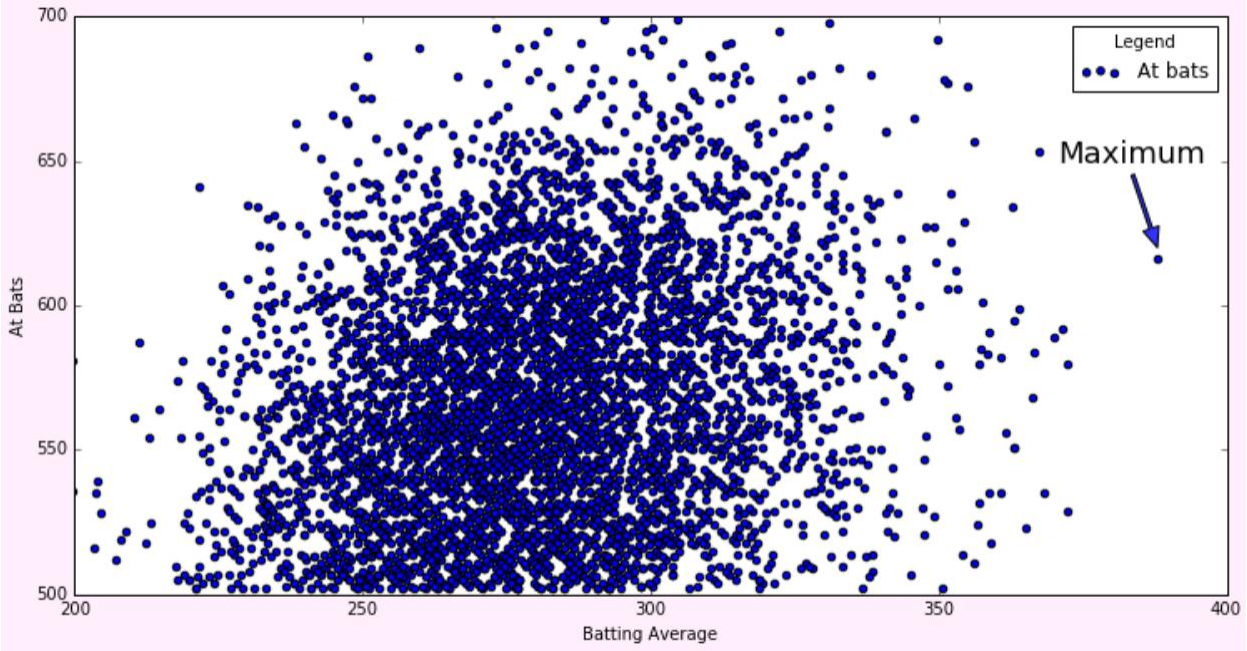


Summary

- Matplotlib is one of the most commonly used visualization libraries
- It supports a huge number of customization features and plot types
- Matplotlib understands and integrates with NumPy

Your Turn! - Task 3-1

- Create a scatter plot that plots batting average (x-axis) against atbats (y-axis)
  - Use plt.scatter(), options are similar to other plots
  - Provide appropriate labels for each axis
  - Create a legend and annotate the maximum batting average



Chapter 4

Pandas

Working with the Python Pandas Tool

Overview

Introducing Pandas

Series

DataFrames

Reading from Files

Introducing Pandas

- Pandas is a data analysis library for Python and part of the SciPy suite

- Features include:
  - APIs for reading data into data structures
  - Data merging, reshaping and pivoting of data
  - Split-apply-combine operations

- Data structures include:
  - Series
  - DataFrame
  - Panel

Panel is deprecated as of version 0.20

Common import syntax
We will refer to this module as pd from now on

```
import pandas as pd
```

Pandas Series

- A Pandas Series is a *one-dimensional ndarray*
  - Series have an index (like a dictionary) called labels
  - Many ndarray methods are overridden to better support Series operations

```
s = pd.Series(data, index, dtype)
```

```
import pandas as pd

ser1 = pd.Series([212, 32, 0, -273])
print(ser1)
```

Output:
```
0    212
1     32
2      0
3   -273
dtype: int64
```

Creating Series

- Additional ways to create series:

```
ser2 = pd.Series(name='City Elevations',
                 index=['Colorado Springs', 'Phoenix',
                        'Raleigh', 'Milwaukee', 'Seattle'],
                 data=[6172, 1132, 437, 723, 429])
```

```
print(ser2)
```

```
Colorado Springs    6172
Phoenix             1132
Raleigh              437
Milwaukee            723
Seattle              429
Name: City Elevations, dtype: int64
```

```
city_elevations = {'Denver': 5883, 'Austin': 632,
       'Philadelphia': 21, 'Billings': 3649, 'Anchorage': 144}
ser3 = pd.Series(city_elevations, name='City Elevations')
```

```
print(ser3)
```

```
Anchorage       144
Austin          632
Billings       3649
Denver         5883
Philadelphia     21
Name: City Elevations, dtype: int64
```

Accessing and Working with Series

- Series allow access to values using methods similar to Python dictionaries:

```
ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                        'New York City', 'Oklahoma City'],
                 name='City Elevations')
```

```
ser4['Mobile']                                  # 218
ser4.get('New York City')                       # 13
ser4['Albuquerque']                             # KeyError
ser4.get('Albuquerque')                         # None
ser4.get('Albuquerque', -1)                     # -1
ser4[['Chicago', 'Oklahoma City']]              # new series
ser4.Chicago                                    # 619

ser4[1:3]                                       # San Diego 15
                                                  Chicago 619
ser4.name                                       # City Elevations
ser4.median()                                   # 218.0
```

Accessing and Working with Series

- *series*.describe() returns a *new summary Series* of the original Series:

```
ser4.describe()
```

```
count          5.000000
mean         432.000000
std          541.983395
min           13.000000
25%           15.000000
50%          218.000000
75%          619.000000
max         1295.000000
Name: City Elevations, dtype: float64
<class 'pandas.core.series.Series'>
```

```
ser4.describe()['mean']
```

```
432.0
```

```
ser4.index
```

```
Index(['Mobile', 'San Diego', 'Chicago', 'New York City', 'Oklahoma City'], dtype='object')
```

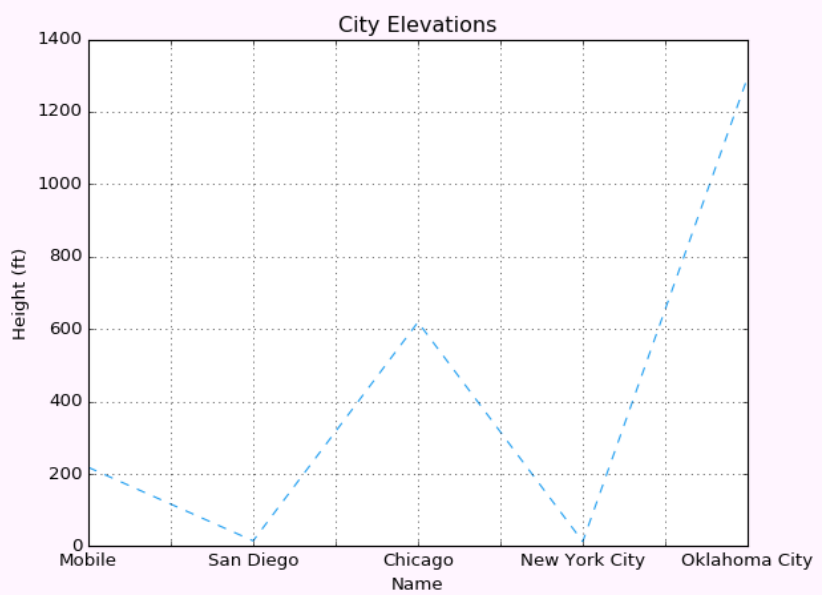
```
ser4.values
```

```
[ 218   15  619   13 1295]
```

Plotting Series

- Pandas data structures support Matplotlib data plotting arguments:

```
import matplotlib.pyplot as plt
import pandas as pd

ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                        'New York City', 'Oklahoma City'],
                 name='City Elevations')
figure = plt.figure()
axis = figure.add_subplot(111)
axis.set_title('City Elevations')
axis.set_xlabel('Name')
axis.set_ylabel('Height (ft)')

# same as ser4.plot.line()
ser4.plot(style='--', kind='line',
          ax=axis, color='#24a1f2',
          grid=True)

plt.show()
```



DataFrames

- Pandas DataFrames are two-dimensional data structures containing labels on each axis

pd.DataFrame(data, index, columns, dtype)

  - Conceptually similar to a spreadsheet
  - DataFrames support an index (rows labels) and columns (column labels)

```
df5 = pd.DataFrame(np.arange(16).reshape(4, 4),
                   index=['row0', 'row1', 'row2', 'row3'],
                   columns=['col0', 'col1', 'col2', 'col3'])
```

index

columns

```
      col0  col1  col2  col3
row0     0     1     2     3
row1     4     5     6     7
row2     8     9    10    11
row3    12    13    14    15
```

values

Creating DataFrames

- DataFrames can be created several ways:

```
df1 = pd.DataFrame([['Colorado Springs', 'Phoenix', 'Raleigh',
                     'Milwaukee', 'Seattle'],
                    [6172, 1132, 437, 723, 429]] )
print(df1)
```

Positional-based index and columns

	0	1	2	3	4
0	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
1	6172	1132	437	723	429

```
df1.index = ['City', 'Elevation']
print(df1)
```

	0	1	2	3	4
City	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
Elevation	6172	1132	437	723	429

Label-based indexing

Creating DataFrames Using a Dict of Lists

```
import pandas as pd

data = {
    'elevation': [6172, 1132, 437, 723, 429],
    'median_age': [34.5, 32.8, 32.8, 31.1, 36.1],
    'median_income': [53550, 46601, 55170, 35186, 70172],
}

df2 = pd.DataFrame(data, index=['Colorado Springs', 'Phoenix',
                                'Raleigh', 'Milwaukee', 'Seattle'])
```

A dict of lists can be used to create a DataFrame

```
print(df2)
```

	elevation	median_age	median_income
Colorado Springs	6172	34.5	53550
Phoenix	1132	32.8	46601
Raleigh	437	32.8	55170
Milwaukee	723	31.1	35186
Seattle	429	36.1	70172

Creating DataFrames Using *ndarrays*

```
data = [
    [6172, 1132, 437, 723, 429],
    [34.5, 32.8, 32.8, 31.1, 36.1],
    [53550, 46601, 55170, 35186, 70172]
]

columns = ['elevation', 'median_age', 'median_income']
index = ['Colorado Springs', 'Phoenix', 'Raleigh',
         'Milwaukee', 'Seattle']
```

A NumPy array can be used to create a DataFrame

```
arr = np.array(data, dtype=int)
df3 = pd.DataFrame(arr.transpose(), index=index, columns=columns)

print(df3)
```

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

Working with DataFrames

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

df4.values

Returns a *NumPy Array* of just the DataFrame data

```
[[ 6172    34 53550]
 [ 1132    32 46601]
 [  437    32 55170]
 [  723    31 35186]
 [  429    36 70172]]
```

df4.T

Returns the transpose (view) of the array

	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
elevation	6172	1132	437	723	429
median_age	34	32	32	31	36
median_income	53550	46601	55170	35186	70172

DataFrame Summary and Metadata

`df4.describe()` Provides summary statistical information

You can transpose this as well!

	elevation	median_age	median_income
count	5.000000	5.0	5.000000
mean	1778.600000	33.0	52135.800000
std	2472.633057	2.0	12791.018555
min	429.000000	31.0	35186.000000
25%	437.000000	32.0	46601.000000
50%	723.000000	32.0	53550.000000
75%	1132.000000	34.0	55170.000000
max	6172.000000	36.0	70172.000000

`df4.info(verbose=True)` Provides metadata info about a DataFrame

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, Colorado Springs to Seattle
Data columns (total 3 columns):
elevation          5 non-null int32
median_age         5 non-null int32
median_income      5 non-null int32
dtypes: int32(3)
memory usage: 100.0+ bytes
```

Accessing Data within DataFrames

- Data from the DataFrame can be accessed numerous ways

Get elevation column, returns a Series

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

```
df4['elevation']
df4.get('elevation')
```

```
Colorado Springs    6172
Phoenix             1132
Raleigh              437
Milwaukee            723
Seattle              429
Name: elevation, dtype: int32
```

```
df4['elevation'].Phoenix
```

```
1132
```

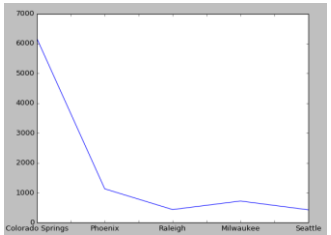
Select column, then row as attribute

```
df4.get('population')
```

```
None
```

Column doesn't exist

```
df4.get('elevation').plot()
plt.show()
```



Accessing DataFrames *(continued)*

Specifying a range of rows

```
df4[1:3]
```

	elevation	median_age	median_income
Phoenix	1132	32	46601
Raleigh	437	32	55170

Specifying multiple columns

```
df4[['elevation', 'median_income']]
```

	elevation	median_income
Colorado Springs	6172	53550
Phoenix	1132	46601
Raleigh	437	55170
Milwaukee	723	35186
Seattle	429	70172

Select a column and then a row

```
df4['elevation']['Colorado Springs']    6172
```

Accessing DataFrames *(continued)*

Specifying a range of rows, and a specific column

```
df4[1:4]['elevation']
```

```
Phoenix       1132
Raleigh        437
Milwaukee      723
Name: elevation, dtype: int32
```

Notes on selecting:

When labels are specified first, columns are selected.

When a range of indices are specified first, rows are selected.

Improving Access with loc[] and iloc[]

- loc[] and iloc[] provide better consistency for selecting data from DataFrames

```
df5 = pd.DataFrame(np.arange(16).reshape(4, 4),
       index=['row0', 'row1', 'row2', 'row3'],
       columns=['col0', 'col1', 'col2', 'col3'])
```

	col0	col1	col2	col3
row0	0	1	2	3
row1	4	5	6	7
row2	8	9	10	11
row3	12	13	14	15

```
df5.iloc[1:3]
```

	col0	col1	col2	col3
row1	4	5	6	7
row2	8	9	10	11

```
df5.loc['row1']
```

```
col0    4
col1    5
col2    6
col3    7
Name: row1, dtype: int32
```

```
df5.iloc[2:, 2:]
```

	col2	col3
row2	10	11
row3	14	15

```
df5.loc['row2':, 'col2':]
```

	col2	col3
row2	10	11
row3	14	15

Labels vs Positions with loc[] and iloc[]

```
ser = pd.Series(1, index=[10, 9, 8, 1, 2, 3, 4, 5])
```

10	1
9	1
8	1
1	1
2	1
3	1
4	1
5	1

dtype: int64

```
ser.iloc[:3]
```

10	1
9	1
8	1

```
ser.loc[:3]
```

10	1
9	1
8	1
1	1
2	1
3	1

```
df = pd.DataFrame(1, columns=[4, 3, 2, 1],
                  index=[10, 9, 8, 1, 2, 3, 4, 5])
```

	4	3	2	1
10	1	1	1	1
9	1	1	1	1
8	1	1	1	1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1

```
df.iloc[:3, :3]
```

	4	3	2
10	1	1	1
9	1	1	1
8	1	1	1

```
df.loc[:3, :3]
```

	4	3
10	1	1
9	1	1
8	1	1
1	1	1
2	1	1
3	1	1

Index (and Column) Sorting

- Data can be "reordered" using pd.sort_index()

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

```
df6 = df4.sort_index()
```

df6

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Milwaukee	723	31	35186
Phoenix	1132	32	46601
Raleigh	437	32	55170
Seattle	429	36	70172

Reorders the index, df4 is unchanged

Use df4.sort_index(inplace=True) to change df4 directly

```
df7 = df4.sort_values(by='median_age')
```

Sorts the DataFrame on the median_age column

Use ascending=False to sort from largest to smallest

DataFrame min() and max()

- df.max(axis=None) - returns the maximum values in a DataFrame
- df.min(axis=None) - returns the minimum values in a DataFrame
- Use df.idxmax() or df.idxmin() to get the associated index for the max or min value

```
df = pd.DataFrame(index=['Colorado Springs', 'Phoenix', 'Raleigh',
                         'Milwaukee', 'Seattle'],
                  data=[6172, 1132, 437, 723, 429])

print(df.max()[0])
print(df.idxmax()[0])
```

```
# 6172
# Colorado Springs
```

df.max() and df.idxmax() each return a Series

Pandas read_csv()

- pd.read_csv() reads data from files into a DataFrame
  - Useful arguments include:
    - delimiter=',' data item separator character
    - skiprows=None how many rows to skip at the beginning
    - encoding='utf-8' file encoding type
    - nrows number of rows to read
    - header which row to use for the column headers
    - usecols select columns, similar to genfromtxt() usage

Doesn't consider the first line as the header now

```
contacts = pd.read_csv('contacts.dat', header=None, dtype='str')
contacts.columns = ['name', 'address', 'state', 'zip',
           'area_code', 'phone', 'email', 'company', 'position']
```

Boolean Indexing (Masking) in Pandas

- Data values that match the given Boolean criterion will be seen in the resulting data structure
  - Example: Find only records where *BOTH* the high and low temps are above 51 degrees:

```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                         index=['Colorado Springs', 'Canon City', 'Pueblo'],
                         columns=['High', 'Low'])
```

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

```
sat_temps.loc[(sat_temps['High'] >= 51) & (sat_temps['Low'] >= 51)]
```

	High	Low
Canon City	82	52
Pueblo	83	53

head() and tail()

- Use df.head(n=5) or df.tail(n=5) to display the first n or last n rows of a DataFrame

```
import pandas as pd
contacts = pd.read_csv('contacts2.dat', header=None, dtype='str')
contacts.columns = ['name', 'address', 'state', 'zip', 'area_code',
                    'phone', 'email', 'company', 'position']

print(contacts.head(3))
```

Displays the first 3 rows of the DataFrame

```
Bob Green, 4517 Elm St. Riverside, NJ,08075,301,356-8921,bob@abc.com, ...
Violet Smith, 220 E. Main Ave Philadelphia, PA,09119,202,421-9008, ...
John Brown, 231 Oak Blvd. Black Hills, SD,82101,719,303-1219, ...
```

Summary

- Pandas *Series* are index-based arrays that behave similar to a Python dictionary
  - They are supported by numerous functions capable of manipulating and transforming data easily
- DataFrames are two-dimensional versions of Series
  - They support similar operations as Series but also support column labels
- Pandas data structures work with both NumPy arrays and Matplotlib

Your Turn! - Task 4-1

- Repeat the Batting Average exercise this time using only Pandas
  - Use pd.read_csv() to read data from the file
  - Keep only records where atbats is 502 or more
  - Keep only records where the year is 1957 or later
  - Sort the records using Pandas sort_values() method
  - Display the top 100 batting averages
- Determine the 50th percentile batting average
- Create a scatter plot from the DataFrame data
  - Use:

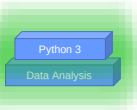
```
df.plot(kind='scatter', x='averages', y='atbats')
```

Work from the task4_1_starter.py, or create a new Jupyter Notebook file

Chapter 5

More Pandas

Advanced Pandas Features



Overview

- Concatenating and Merging Data
- Imputing Values
- Group by:
  - Split
  - Apply
  - Combine
- Pivot Tables
- apply()
- Other read methods

Merging DataFrames

- The correct choice of method (and options) to combine data sets can greatly affect the resulting DataFrame

- concat() - joins DataFrames *along a given axis*
  - Use this approach when "stacking" records

- merge() - joins two DataFrames *by aligning values in a column or index*
  - Best suited when joining by values found in *common columns*, similar to a database style join

concat() Arguments

- Syntax for concat() is:

```
pd.concat(objs, axis=0, join='outer | inner', join_axes=None,
          ignore_index=False, keys=None, levels=None,
          names=None, verify_integrity=False, copy=True)
```

objs -	sequence of Series or DataFrames to join
axis -	concatenate along this axis
join -	‘inner’ or ‘outer’ (def.), outer=union, inner=intersection
ignore_index -	True | False (def.) True=don't use idx values on concat axis
join_axes -	join these specific axes instead of join 'inner' or 'outer'
keys -	sequence, constructs hierarchical index of joined data
levels -	list of sequences, identifies levels of multi-index
names -	list of names for the levels in hierarchy
verify_integrity -	boolean (def. False), check if new concat contains dups.
copy -	boolean (def. True), False=do not copy data

Basic concat() Example

- Use concat() for most DataFrame concatenation requirements:

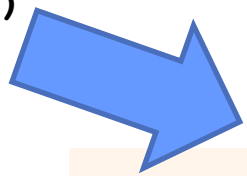
```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                         index=['Colorado Springs', 'Canon City', 'Pueblo'],
                         columns=['High', 'Low'])

sun_temps = pd.DataFrame(data={'High':(77, 81, 84), 'Low':(48, 49, 50)},
                         index=['Colorado Springs', 'Canon City', 'Pueblo'])

merged = pd.concat([sat_temps, sun_temps])
```

sat_temps

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

sun_temps

	High	Low
Colorado Springs	77	48
Canon City	81	49
Pueblo	84	50



	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53
Colorado Springs	77	48
Canon City	81	49
Pueblo	84	50

concat() Using keys, ignore_index

```
merged = pd.concat([sat_temps, sun_temps], keys=['Saturday', 'Sunday'])
```

		High	Low
Saturday	Colorado Springs	78	50
	Canon City	82	52
	Pueblo	83	53
Sunday	Colorado Springs	77	48
	Canon City	81	49
	Pueblo	84	50

```
merged.loc['Saturday', 'Colorado Springs']['High']
```

```
merged = pd.concat([sat_temps, sun_temps], ignore_index=True)
```

keys will be ignored if ignore_index is used

	High	Low
0	78	50
1	82	52
2	83	53
3	77	48
4	81	49
5	84	50

MultiIndex DataFrames

- The previously merged DataFrame uses hierarchical indexing (or MultiIndexing)

merged

		High	Low
Saturday	Colorado Springs	78	50
	Canon City	82	52
	Pueblo	83	53
Sunday	Colorado Springs	77	48
	Canon City	81	49
	Pueblo	84	50

merged.index

Level 0

Level 1

```
MultiIndex(levels=[['Saturday', 'Sunday'], ['Colorado Springs', 'Canon City', 'Pueblo']],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

```
merged.iloc[merged.index.get_level_values(1) == 'Colorado Springs'])
```

		High	Low
Saturday	Colorado Springs	78	50
Sunday	Colorado Springs	77	48

concat() Along an Axis

```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                         index=['Colorado Springs', 'Canon City', 'Pueblo'],
                         columns=['High', 'Low'])

sat_humidity = pd.DataFrame([22, 18, 19, 25],
              index=['Colorado Springs', 'Canon City', 'Pueblo', 'Denver'],
              columns=['Humidity'])

merged = pd.concat([sat_temps, sat_humidity], axis=1)
```

What happens if you change this to axis=0?

sat_temps

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

sat_humidity

	Humidity
Colorado Springs	22
Canon City	18
Pueblo	19
Denver	25

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

concat() Using join=

sat_temps

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

sat_humidity

	Humidity
Colorado Springs	22
Canon City	18
Pueblo	19
Denver	25

```
merged = pd.concat([sat_temps, sat_humidity], axis=1, join='inner')
```

	High	Low	Humidity
Colorado Springs	78	50	22
Canon City	82	52	18
Pueblo	83	53	19

Only common rows are retained. (def. is 'outer' which is a union of the rows)

Merging DataFrames with merge()

- merge() behaves more like SQL

```
merge(left, right, how='inner', on=None, left_on=None,
right_on=None, left_index=False, right_index=False,
sort=False, suffixes=('_x', '_y'), copy=True, indicator=False)
```

left -	a left DataFrame
right -	a right DataFrame
on -	column names to join on. Must be in both left & right
left_on -	columns from the left DataFrame to use as keys
right_on -	columns from the right DataFrame to use as keys
left_index -	if True, use the index (row labels) from the left DataFrame as the keys
right_index -	same as left_index, but for the right DataFrame
how -	type of join operation. Either: 'left', 'right', 'outer', 'inner' (def.)
sort -	sort the result DataFrame by the join keys (def. is True)
suffixes -	tuple of strings to apply to overlapping columns
copy -	always copy data (def. is True) from the passed DataFrame
indicator -	add a column called _merge with info on the source of rows

Using merge()

```
data1 = np.arange(1, 10).reshape(3,3)
df1 = pd.DataFrame(data1,
                   columns=['a', 'b', 'c'])

data2 = np.arange(1, 13).reshape(4,3)
df2 = pd.DataFrame(data2, columns=['d', 'b', 'e'])
```

df1

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

df2

	d	b	e
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

```
merged1 = pd.merge(df1, df2, on='b')
```

merged1

	a	b	c	d	e
0	1	2	3	1	3
1	4	5	6	4	6
2	7	8	9	7	9

```
merged2 = pd.merge(df1, df2, left_on='a',
                   right_on='d', how='outer')
```

merged2

	a	b_x	c	d	b_y	e
0	1.0	2.0	3.0	1	2	3
1	4.0	5.0	6.0	4	5	6
2	7.0	8.0	9.0	7	8	9
3	NaN	NaN	NaN	10	11	12

Using merge() *(continued)*

df1

	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

df2

	d	b	e
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

how='inner' (default)

```
merged3 = pd.merge(df1, df2, left_on='a',
                   right_on='d')
```

merged3

	a	b_x	c	d	b_y	e
0	1	2	3	1	2	3
1	4	5	6	4	5	6
2	7	8	9	7	8	9

```
merged4 = pd.merge(df1, df2, left_index=True,
                   right_index=True)
```

merged4

	a	b_x	c	d	b_y	e
0	1	2	3	1	2	3
1	4	5	6	4	5	6
2	7	8	9	7	8	9

Renaming Columns *(2 ways)*

- The columns attribute to rename columns:

temps_df

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

	High	Low	Pct Hum
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

```
cols = temps_df.columns.values
cols[2] = 'Pct Hum'
temps_df.columns = cols
print(temps_df)
```

This is an ndarray. The 3rd col value is changed and set back to the columns object.

- You can also use the rename() method

From this

To this

```
temps_df.rename(columns={'Pct Hum': '% Hum'},
                inplace=True)
```

	High	Low	% Hum
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

Imputing Missing Values

- Use df.fillna() to deal with missing data values

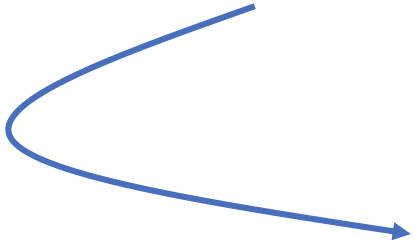
df.fillna(value=None, axis=None, inplace=False)

temps_df

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

Use **temps_df.dropna()** to drop rows that contain NaN values

```
temps_df['High'].fillna(value=80, inplace=True)
temps_df['Low'].fillna(52, inplace=True)
```



temps_df

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	80.0	52.0	25
Pueblo	83.0	53.0	19

Pandas NaN and Infinity

- Pandas uses NumPy values for representing NaN and infinity (np.nan, np.inf):

temps_df

	High	Low	Humidity
Canon City	82.0	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	83.0	53.0	19

```
temps_df.loc[(temps_df['High'] > 80), 'High'] = np.inf
```

Converts any values in the 'High' column > 80 to infinity

temps_df

	High	Low	Humidity
Canon City	inf	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	inf	53.0	19

Removing Infinity Values

- Pandas doesn't have a specific method for removing infinity values
  - Use replace() to first convert the values to NaN, then use dropna() to remove them

temps_df

	High	Low	Humidity
Canon City	inf	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	inf	53.0	19

```
temps_df.replace([np.inf, -np.inf], np.nan, inplace=True)
```

temps_df

	High	Low	Humidity
Canon City	NaN	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	NaN	53.0	19

Dropping Rows Containing NaNs

temps_df

	High	Low	Humidity
Canon City	NaN	52.0	18
Colorado Springs	78.0	50.0	22
Denver	NaN	NaN	25
Pueblo	NaN	53.0	19

```
temps_df.dropna(subset=['Low'], inplace=True)
```

temps_df

	High	Low	Humidity
Canon City	NaN	52.0	18
Colorado Springs	78.0	50.0	22
Pueblo	NaN	53.0	19

Note: use **subset=** to only consider certain columns, leave off to examine all columns

"Group By" Operations

- Much like SQL queries, DataFrame data can be grouped and then operated upon in groups

  – Often this process can be broken down further into discrete steps referred to as:

Split - Apply - Combine

  – Split - groups data according to some criteria
  – Apply - performs operations on the grouped data
  – Combine - places resulting values into a structure

Contacts Sample Data

```
Bob Green,          4517 Elm St. Riverside,                  NJ, 08075,301,356-8921
Violet Smith,       220 E. Main Ave Philadelphia,            PA, 09119,202,421-9008
John Brown,         231 Oak Blvd. Black Hills,               SD, 82101,719,303-1219
Ed Blumenthal,      3012 Briarwood Ln. Denver,               CO, 80101,719,422-8091
Rosey Englund,      1818 Mockingbird Ln. Aurora,             CO, 82101,719,286-1920
Tori Gray,          2218 Masengild Ave.,                     NJ, 08075,301,338-6571
Lisa Black,         89 Prince Dr. Philadelphia,              PA, 09119,202,419-0650
Tom Redford,        2323 Nicholas St. Newark,                NJ, 07101,862,227-8022
Sally White,        3345 Spruce Cir. Harrisburg,             PA, 17105,717,429-1217
Goldy Simpson,      4430 Mountainside Creek Rd Custer,       SD, 57730,605,689-3131
O. Range,           1703 Treeline Dr. Denver,                CO, 80101,719,429-1356
Sil Verna,          557 Pine Ave Aurora                      CO, 82101,719,286-1920
Pinky Tuscadero,    601 Sapling Blvd.,                       NJ, 08501,609,227-6001
Hazel Sanford,      27 Musket Dr. Pittsburg,                 PA, 15201,412,389-7711
```

```
bob@abc.com,              ABC Inc.,                        President
ssmithj@hypex.org,        FakeCo Inc.,                     Janitor
vivoj@wandergem.com,      Wandergem LLC,                   Sr. Analyst
ep20002@gmail.com,        Hanibow & Delite,                Programmer
ke7001@yahoo.com,         Wadlow Inc.,                     Administrative Lead
tjames@acme.com,          Acme Inc.,                       Inventor
victors89@glaser.org,     Glaser Properties LLC.,          Manager
tom.redford@gmail.com,    Illustrative Studio Systems,     Graphics Engineer
swhope@ggworth.com,       Bond Appliances,                 Administrative Specialist
simpson@yahoo.com,        Crater Construction,             Owner
ffnine27@hotmail.com,     n/a,                             Retired
sil@yahoo.com,            Music Enthusiasts,               Salesperson
pinky@freedom.net,        Self-employed,                   Vocal Artist
hazel@outlook.com,        Bourne Legal Associates,         Paralegal
```

Grouping Contacts By State

- To segment (split) data into groups, use the groupby() method:

```
df.groupby(col)
df.groupby([col1, col2])
df.groupby(col, axis=1)
```

```
contacts = pd.read_csv('contacts2.dat', header=None,
                       names=['name', 'address', 'state', 'zip', 'area_code',
                              'phone', 'email', 'company', 'position'],
                       converters={'state': lambda txt: txt.strip()})
```

Used to strip whitespace from specified column

```
bystate = contacts.groupby('state')
print(bystate.groups)
```

```
{'NJ': [0, 5, 7, 12], 'SD': [2, 9],
 'CO': [3, 4, 10, 11], 'PA': [1, 6, 8, 13]}
```

"Grouped" Operations

- Once data is "split" (grouped), it can be operated upon using a number of methods:

group.agg()
group.boxplot()
group.cumin()
group.describe()
group.filter()
group.get_group()
group.median()
group.ngroups
group.plot()
group.rank()
group.std()
group.transform()
group.aggregate()
group.count()
group.cumprod()
group.first() / last()
group.groups
group.hist()
group.max() / min()
group.idxmax() / idxmin()
group.nth(n)
group.prod()
group.resample()
group.sum()
group.var()
group.apply()
group.cummax()
group.cumsum()
group.fillna()
group.head() / tail()
group.indices
group.mean()
group.name
group.size()

Operations on "Grouped" Items

```
bystate.size()
```

Returns a new Series

CO	4
NJ	4
PA	4
SD	2

```
bystate.first()
```

Returns a DataFrame

state	name	...	position
		...	
CO	Ed Blumenthal	...	Programmer
NJ	Bob Green	...	President
PA	Violet Smith	...	Janitor
SD	John Brown	...	Sr. Analyst

```
bystate.nth(1)
```

state	address	...	zip
		...	
CO	1818 Mockingbird Ln. Aurora	...	82101
NJ	2218 Masengild Ave.	...	08075
PA	89 Prince Dr. Philadelphia	...	09119
SD	4430 Mountainside Creek Rd Custer	...	57730

```
bystate.nth(1)['name']
```

Returns a Series

state	
CO	Rosey Englund
NJ	Tori Gray
PA	Lisa Black
SD	Goldy Simpson

Selecting a Group

- Use get_group() to select just a single group to operate on:

```
bystate.get_group('CO')
```

	address	area_code	...	position	zip
3	3012 Briarwood Ln. Denver	719	...	Programmer	80101
4	1818 Mockingbird Ln. Aurora	719	...	Administrative Lead	82101
10	1703 Treeline Dr. Denver	719	...	Retired	80101
11	557 Pine Ave Aurora	719	...	Salesperson	82101

```
colorado = bystate.get_group('CO')
colorado[colorado['zip'] == 82101]
```

	address	area_code	...	position	zip
4	1818 Mockingbird Ln. Aurora	719	...	Administrative Lead	82101
11	557 Pine Ave Aurora	719	...	Salesperson	82101

Pivot Tables *(1 of 2)*

- Pivot tables are summary tables created from an original table
  - Usually perform grouping operations on columns
  - Usually perform averages, sums, sorts on other columns
- Example:

Day	Highs	Lows	Humidity	Wind Speed	Outlook	Red Flag
1	88	68	25	10	Sunny	False
2	84	65	31	5	Cloudy	False
3	86	66	32	5	Light Rain	False
4	89	67	26	5	Rain	False
5	92	70	22	10	Sunny	False
6	95	71	18	20	Sunny	True
7	94	69	27	10	Sunny	False
8	93	72	25	10	Rain	False
9	98	76	16	5	Cloudy	True
10	94	72	22	10	Sunny	False

```
df = pd.DataFrame(data, columns=['Day', 'Highs', 'Lows', 'Humidity',
                                 'Wind Speed', 'Outlook', 'Red Flag'])
df.set_index('Day', inplace=True)
```

Pivot Tables *(2 of 2)*

```
df['Outlook'] = df['Outlook'].astype('category')
df['Outlook'].cat.set_categories(
                  ['Sunny', 'Rain', 'Light Rain', 'Cloudy'],inplace=True)
```

Set the 'Outlook' column to **category** and limit its values

```
pivot = pd.pivot_table(data=df, index=['Outlook'],
                       values=['Highs', 'Lows'], aggfunc=[np.mean, len] )
```

	mean		len	
	Highs	Lows	Highs	Lows
Outlook				
Sunny	92.6	70.0	5	5
Rain	91.0	69.5	2	2
Light Rain	86.0	66.0	1	1
Cloudy	91.0	70.5	2	2

index = column(s) we want to get summary info on
values = the column(s) we want to apply aggregation functions to
aggfunc = the functions to apply to the values columns

```
pivot.query('Outlook == ["Sunny"]')
```

	mean		len	
	Highs	Lows	Highs	Lows
Outlook				
Sunny	92.6	70.0	5	5

Working with Log Files

Note: for the complete solution, refer to: 10_reading_a_log_file.ipynb

- Our data is a typical log file *not a CSV*

resources/new_access.log

```
109.169.248.247 - - [12/Dec/2015:18:25:11 +0100] "GET /administrator/ HTTP/1.1" 200 4263 ...
109.169.248.247 - - [12/Dec/2015:18:25:11 +0100] "POST /administrator/index.php HTTP/1.1" 200 4494 ...
46.72.177.4 - - [12/Dec/2015:18:31:08 +0100] "GET /administrator/ HTTP/1.1" 200 4263 ...
46.72.177.4 - - [12/Dec/2015:18:31:08 +0100] "POST /administrator/index.php HTTP/1.1" 200 4494 ...
83.167.113.100 - - [12/Dec/2015:18:31:25 +0100] "GET /administrator/ HTTP/1.1" 200 4263 ...
83.167.113.100 - - [12/Dec/2015:18:31:25 +0100] "POST /administrator/index.php HTTP/1.1" 200 4494 ...
```

```
log = pd.read_csv('../resources/new_access.log', sep='\s+',
                  usecols=(0, 3, 5, 6, 7, 9),
                  names=['addr', 'req_date', 'request', 'status',
                         'size', 'browser'],
                  error_bad_lines=False)
print(log.shape)
print(log.head())
```

(463915, 6)

	addr	req_date	request	status	size	browser
0	109.169.248.247	[12/Dec/2015:18:25:11	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
1	109.169.248.247	[12/Dec/2015:18:25:11	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2	46.72.177.4	[12/Dec/2015:18:31:08	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
3	46.72.177.4	[12/Dec/2015:18:31:08	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...

Log Files Info & Converting the Date

- Learn about columns using info():

```
log.info()
```

This will happen *after* the code below executes!

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 463915 entries, 0 to 463914
Data columns (total 6 columns):
addr            463915 non-null object
req_date        463915 non-null datetime64[ns]
request         463915 non-null object
status          463915 non-null int64
size            463914 non-null object
browser         463860 non-null object
dtypes: datetime64[ns](1), int64(1), object(4)
memory usage: 21.2+ MB
```

- Convert the req_date column into a date type:

```
log.req_date = pd.to_datetime(log.req_date, format='[%d/%b/%Y:%H:%M:%S')
```

Log Files: Grouping (by Address)

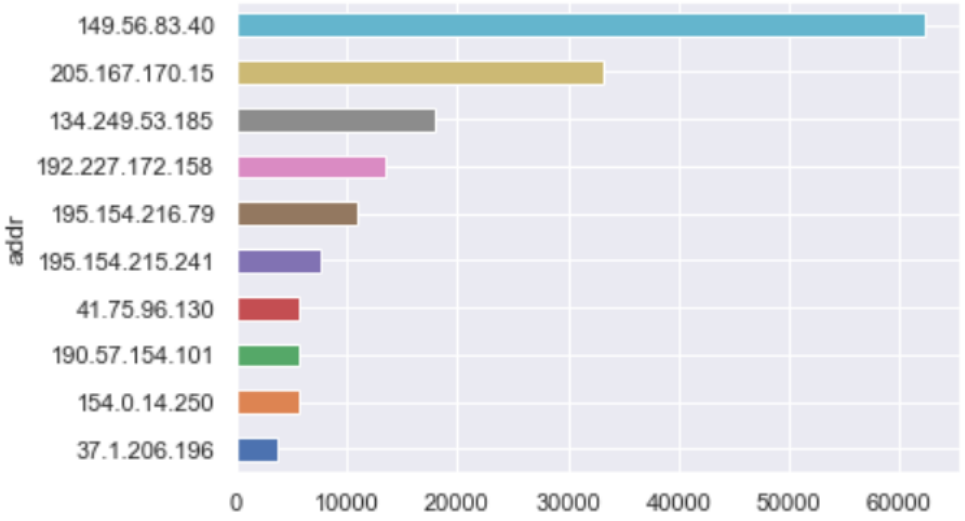
- You can group results in many ways:

```
top10_addr = log.groupby('addr').size().sort_values(ascending=False).head(10)
print(top10_addr)
```

```
addr
149.56.83.40      62178
205.167.170.15    33302
134.249.53.185    17904
192.227.172.158   13474
195.154.216.79    10996
195.154.215.241    7705
41.75.96.130       5664
190.57.154.101     5662
154.0.14.250       5659
37.1.206.196       3780
dtype: int64
```

```
top10_addr[::-1].plot(kind='barh')
```

Allows largest to appear on top



Log Files: Grouping (by Browser)

```
log.groupby('browser').size()
                .sort_values(ascending=False).head(10)[::-1].plot(kind='barh')
```

Custom Column Operations

- Use apply() to execute a custom function on cells
  - Here, the GET and POST values are extracted from the request column and placed into a new http-method column

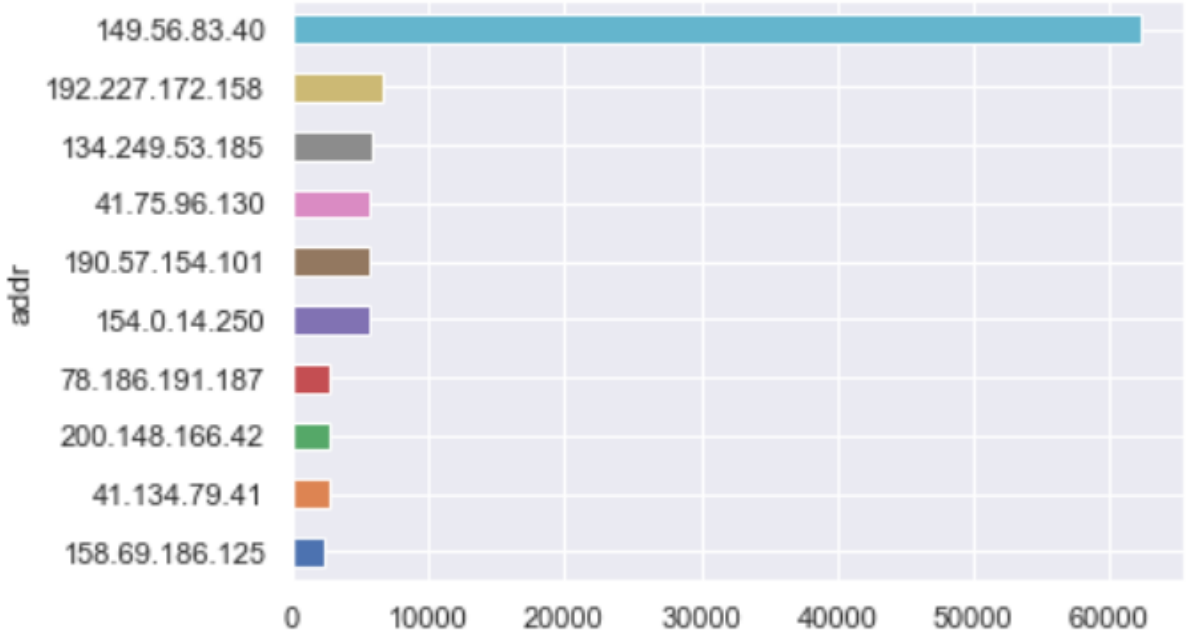
```
log['http-method'] = log.request.apply(lambda s: s.split()[0])
print(log.head())
```

	addr	req_date	request	status	size	browser	http-method
0	109.169.248.247	2015-12-12 18:25:11	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	GET
1	109.169.248.247	2015-12-12 18:25:11	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	POST
2	46.72.177.4	2015-12-12 18:31:08	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	GET
3	46.72.177.4	2015-12-12 18:31:08	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	POST
4	83.167.113.100	2015-12-12 18:31:25	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	GET

Grouping Selected Records

```
top10_POSTS = log[log['http-method'] == 'POST'].groupby('addr').size()
                                                .sort_values(ascending=False).head(10)

top10_POSTS[::-1].plot(kind='barh')
```

Only top 10 POST requests are viewed



Constructing New Data

```
import pygeoip
GEOIP = pygeoip.GeoIP('../resources/GeoLiteCity.dat')
GEOIP.record_by_addr('149.56.83.40')
```


{'postal_code': 'H3A',
 'country_code': 'CA',
 'country_code3': 'CAN',
 'country_name': 'Canada',
 'continent': 'NA',
 'region_code': 'QC',
 'city': 'Montréal',
 'latitude': 45.50399999999999,
 'longitude': -73.5747,
 'time_zone': 'America/Montreal'}


```
def get_location(addr):
    results = ['', '', 0, 0]
    try:
        info = GEOIP.record_by_addr(addr)
        if info:
            results = [info.get('country_name'), info.get('city'),
                       info.get('latitude'),  info.get('longitude')]
    except pygeoip.GeoIPError:
        pass

    return results
```

map() is similar to apply() except for Series

```
results = log.addr.map(get_location)

new_df = pd.DataFrame(results.values.tolist(),
             columns=['country', 'city', 'latitude', 'longitude'])
print(new_df.head())
```

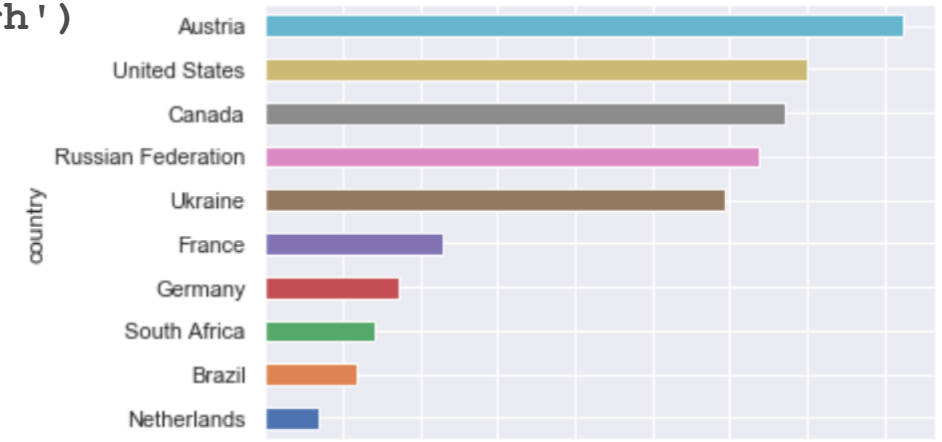
	country	city	latitude	longitude
0	Russian Federation	Chapaevsk	52.9781	49.7197
1	Russian Federation	Chapaevsk	52.9781	49.7197
2	Russian Federation	Kursk	51.8830	36.2659
3	Russian Federation	Kursk	51.8830	36.2659
4	Russian Federation	Moscow	55.7522	37.6156

Merging the Two DataFrames

```
merged = pd.concat([log, new_df])
print(merged.head())
```

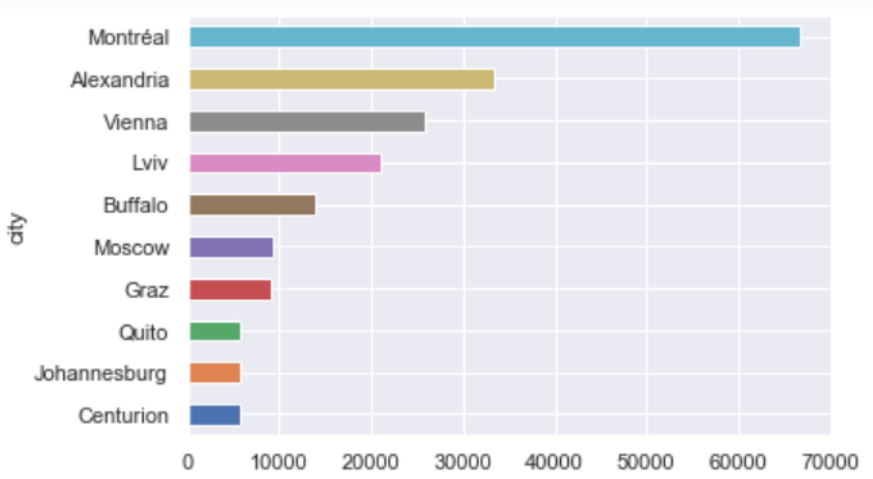
	addr	browser	city	country	http-method	latitude	longitude	req_date	request	size	status
0	109.169.248.247	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	GET	NaN	NaN	2015-12-12 18:25:11	GET /administrator/ HTTP/1.1	4263	200.0
1	109.169.248.247	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	POST	NaN	NaN	2015-12-12 18:25:11	POST /administrator/index.php HTTP/1.1	4494	200.0
2	46.72.177.4	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	GET	NaN	NaN	2015-12-12 18:31:08	GET /administrator/ HTTP/1.1	4263	200.0
3	46.72.177.4	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	POST	NaN	NaN	2015-12-12 18:31:08	POST /administrator/index.php HTTP/1.1	4494	200.0
4	83.167.113.100	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...	NaN	NaN	GET	NaN	NaN	2015-12-12 18:31:25	GET /administrator/ HTTP/1.1	4263	200.0

```
top10_countries =
   merged.groupby('country').size().sort_values(ascending=False).head(10)
top10_countries[::-1].plot(kind='barh')
```

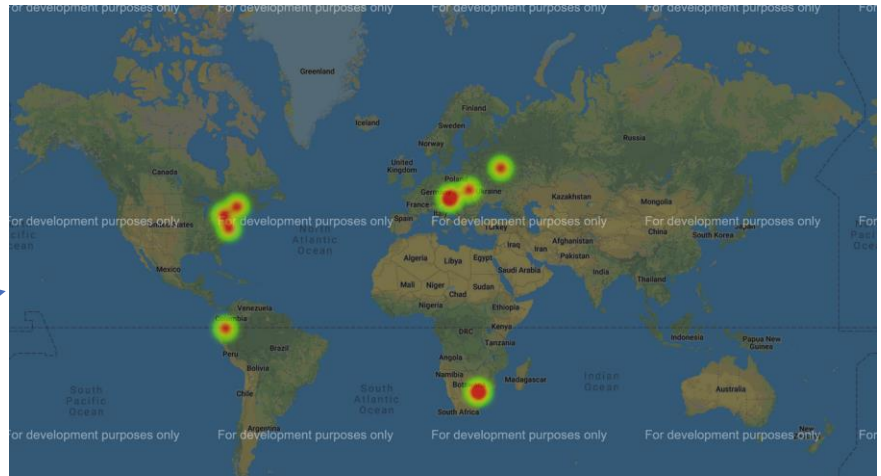


Final Results of Log File Analysis

```
top10_cities =
    merged.groupby('city').size().sort_values(ascending=False).head(10)
top10_cities[::-1].plot(kind='barh');
```




```
from gmplot import GoogleMapPlotter
lats, longs = [], []
for city in top10_cities.index:
    latitude = merged.loc[city == merged['city'], 'latitude']
    longitude = merged.loc[city == merged['city'], 'longitude']
    lats.append(latitude.iloc[0])
    longs.append(longitude.iloc[0])

g_map = GoogleMapPlotter(37, 95, 3)
g_map.heatmap(lats, longs, radius=20)
g_map.draw('results.html')
```



Binning

- Binning is the conversion of *continuous* (or many categorical) values into smaller sets of categorical values

```
df = pd.DataFrame(data,
     columns = ['gender', 'age', 'state'])
```

```
bins = [0, 18, 25, 35, 45, 55, 150]
df['age_group'] = pd.cut(df['age'], bins=bins)
```

```
labels = ['kid', 'early adult', 'young adult',
          'middle adult', 'older adult', 'senior']
df['age_name'] = pd.cut(df['age'], bins=bins, labels=labels)
print(df)
```

	gender	age	state	age_group	age_name
0	M	33	California	(25, 35]	young adult
1	M	55	Florida	(45, 55]	older adult
2	F	44	Maine	(35, 45]	middle adult
3	F	43	Idaho	(35, 45]	middle adult
4	F	64	Alaska	(55, 150]	senior
5	F	49	Ohio	(45, 55]	older adult
6	F	13	New York	(0, 18]	kid
7	M	37	California	(35, 45]	middle adult
8	M	61	Texas	(55, 150]	senior
9	M	27	Washington	(25, 35]	young adult
10	F	22	Florida	(18, 25]	early adult
11	M	55	New Jersey	(45, 55]	older adult
12	F	18	Nevada	(0, 18]	kid
13	M	27	Oregon	(25, 35]	young adult
14	F	26	Arizona	(25, 35]	young adult
15	M	21	Utah	(18, 25]	early adult
16	F	19	Oregon	(18, 25]	early adult
17	M	67	Colorado	(55, 150]	senior

Your Turn! - Task 5-1

Pandas Groupby

Using Pandas to read Batting.csv, answer the two following questions:

1. Which team has hit the most home runs (cumulative)?
2. Which team hit the most home runs in 2015?

Then, plot the total home runs hit per decade to see which decade had the most home runs

	playerID	yearID	stint	teamID	lgID	G	AB	R	H	2B	...	RBI	SB	CS	BB	SO	IBI
0	abercda01	1871	1	TRO	NaN	1	4.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	Nal
1	addybo01	1871	1	RC1	NaN	25	118.0	30.0	32.0	6.0	...	13.0	8.0	1.0	4.0	0.0	Nal
2	allisar01	1871	1	CL1	NaN	29	137.0	28.0	40.0	4.0	...	19.0	3.0	1.0	2.0	5.0	Nal
3	allisdo01	1871	1	WS3	NaN	27	133.0	28.0	44.0	10.0	...	27.0	1.0	1.0	0.0	2.0	Nal
4	ansonca01	1871	1	RC1	NaN	25	120.0	29.0	39.0	11.0	...	16.0	6.0	2.0	2.0	1.0	Nal

Work from the task5_1_starter.py, or copy its contents into a new Notebook file if you prefer to work within Jupyter.

Time Series and DataFrames

- Time Series and DataFrames are structures whose index is time-based

```
log = pd.read_csv('../resources/new_access.log', sep='\s+',
                  usecols=(0, 3, 5, 6, 7, 9),
                  names=['addr', 'req_date', 'request', 'status', 'size', 'browser'],
                  error_bad_lines=False)
log.req_date = pd.to_datetime(log.req_date, format='[%d/%b/%Y:%H:%M:%S')

date_based_log = log.set_index('req_date')
print(date_based_log.info())
print(date_based_log.head())
```

Notice we made our *req_date* column an index now

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 463915 entries, 2015-12-12 18:25:11 to 2017-02-08 10:39:23
Data columns (total 5 columns):
addr       463915 non-null object
request    463915 non-null object
status     463915 non-null int64
size       463914 non-null object
browser    463860 non-null object
dtypes: int64(1), object(4)
memory usage: 21.2+ MB
```

req_date	addr	request	status	size	browser
2015-12-12 18:25:11	109.169.248.247	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:25:11	109.169.248.247	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:31:08	46.72.177.4	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:31:08	46.72.177.4	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:31:25	83.167.113.100	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...

Using the Time Series

- Time Series has special capabilities for matching values:

```
print(date_based_log['2015-12-12'].shape)
```

(358, 5)

Gives back just records that fall on this date

```
print(date_based_log['2015'].shape)
```

(14148, 5)

Just records from that year

```
print(date_based_log['2016-01'].shape)
```

(28224, 5)

Just records from that month

```
print(date_based_log['2015-12-12 18:00:00':'2015-12-12 18:30:00'])
```

req_date	addr	request	status	size	browser
2015-12-12 18:25:11	109.169.248.247	GET /administrator/ HTTP/1.1	200	4263	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...
2015-12-12 18:25:11	109.169.248.247	POST /administrator/index.php HTTP/1.1	200	4494	Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20...

Just records within the given 30 minute range

Reading Excel Spreadsheet Files

- read_excel() can read a Microsoft Excel spreadsheet table into a DataFrame

```
pd.read_excel(io, sheetname=0, header=0, skiprows=None, skip_footer=0,
              index_col=None, names=None, na_values=None,
              convert_float=True, thousands=None, converters=None)
```

io -	URL or path to an Excel-compatible file
sheetname -	string name or int for the sheet to read
header -	row to use for the column labels
skiprows -	rows to skip at the beginning
skip_footer -	rows to skip at the end
index_col -	column to use as the column labels of the DataFrame
names -	list of column names to use
converters -	dict of functions for converting values in columns
thousands -	thousands separator for parsing string columns to numeric
convert_float -	convert floats to int
na_values -	strings to use in place of NaN or NA values

Using read_excel()

temperatures.xlsx

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

	High	Low
Colorado Springs	77	48
Canon City	81	49
Pueblo	84	50

	Humidity
Colorado Springs	22
Canon City	18
Pueblo	19
Denver	25

```
sat_temps    = pd.read_excel('temperatures.xlsx', sheetname='Saturday')
sun_temps    = pd.read_excel('temperatures.xlsx', sheetname=1)
sat_humidity = pd.read_excel('temperatures.xlsx', sheetname='Humidity')

sat_merged = pd.concat([sat_temps, sat_humidity], axis=1)
merged = pd.concat([sat_merged, sun_temps], keys=['Saturday', 'Sunday'])
```

		High	Humidity	Low
Saturday	Canon City	82.0	18.0	52.0
	Colorado Springs	78.0	22.0	50.0
	Denver	NaN	25.0	NaN
	Pueblo	83.0	19.0	53.0
Sunday	Colorado Springs	77.0	NaN	48.0
	Canon City	81.0	NaN	49.0
	Pueblo	84.0	NaN	50.0

Analysis: Black Friday vs Cyber Monday

- We'll use Pandas read_html() to compare and analyze data related to Black Friday sales vs Cyber Monday sales:

```
url1 = 'https://en.wikipedia.org/wiki/Black_Friday_(shopping)'
url2 = 'https://en.wikipedia.org/wiki/Cyber_Monday'

bfriday = pd.read_html(url1, header=0)[1]
cyber = pd.read_html(url2, header=0)[1]

print(bfriday)
```

We took the second table on the page and used row 0 for the column names

	Year	Date	Survey published	Shoppers (millions)	Average spent	Total spent	Consumers polled	Margin for error
0	2020	Nov 27	NaN	NaN	NaN	NaN	NaN	NaN
1	2019	Nov 29	NaN	NaN	NaN	NaN	NaN	NaN
2	2018	Nov 23	NaN	NaN	NaN	NaN	NaN	NaN
3	2017	Nov 24	Nov 28[130]	174.0	$335.47	$58.3 billion	3242.0	+/- 1.7%
4	2016	Nov 25	NaN	NaN	NaN	NaN	NaN	NaN
5	2015	Nov 27	NaN	NaN	NaN	NaN	NaN	NaN
6	2014[131]	Nov 28	Nov 30	233.0	$380.95	$50.9 billion	4631.0	1.5%
7	2013	Nov 29	Dec 1	249.0	$407.02	$57.4 billion	4864.0	1.7%
8	2012	Nov 23	Nov 25	247.0	$423.66	$59.1 billion	4005.0	1.6%
9	2011	Nov 25	Nov 27	226.0	$398.62	$52.5 billion	3826.0	1.6%
10	2010	Nov 26	Nov 28	212.0	$365.34	$45.0 billion	4306.0	1.5%
11	2009	Nov 27	Nov 29	195.0	$343.31	$41.2 billion	4985.0	1.4%
12	2008	Nov 28	Nov 30	172.0	$372.57	$41.0 billion	3370.0	1.7%
13	2007	Nov 23	Nov 25	147.0	$347.55	$34.6 billion	2395.0	1.5%
14	2006	Nov 24	Nov 26	140.0	$360.15	$34.4 billion	3090.0	1.5%
15	2005	Nov 25	Nov 27	132.0	$301.81	$26.8 billion	NaN	NaN

This data requires LOTS of cleaning!

Cleaning the Black Friday Table

```
print(bfriday.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16 entries, 0 to 15
Data columns (total 8 columns):
Year                    16 non-null object
Date                    16 non-null object
Survey published        11 non-null object
Shoppers (millions)     11 non-null float64
Average spent           11 non-null object
Total spent             11 non-null object
Consumers polled        10 non-null float64
Margin for error        10 non-null object
dtypes: float64(2), object(6)
memory usage: 1.1+ KB
```

Drop unneeded columns and rows with NaN values in the remaining columns

Remove the footnotes and $ and typed our columns

```
bfriday = bfriday.drop(['Margin for error', 'Survey published', 'Date',
                        'Consumers polled'], axis=1).dropna()

bfriday.Year = bfriday.Year.map(lambda yr: str(yr).split('[')[0]).astype(int)

bfriday['Average spent'] = bfriday['Average spent'].str.lstrip('$').astype(float)

bfriday['Total spent (billions)'] = bfriday['Total spent'].str.lstrip('$')

                                                  .str.strip('billion').astype(float)
bfriday.drop(['Total spent'], axis=1, inplace=True)
```

Removed the 'billion' word

Black Friday DataFrame Cleaned Up

	Year	Shoppers (millions)	Average spent	Total spent (billions)
3	2017	174.0	335.47	58.3
6	2014	233.0	380.95	50.9
7	2013	249.0	407.02	57.4
8	2012	247.0	423.66	59.1
9	2011	226.0	398.62	52.5
10	2010	212.0	365.34	45.0
11	2009	195.0	343.31	41.2
12	2008	172.0	372.57	41.0
13	2007	147.0	347.55	34.6
14	2006	140.0	360.15	34.4
15	2005	132.0	301.81	26.8

Cleaning the Cyber Monday Table

Don't need all these columns

	Day	Year	Sales(millionsof USD)	% Change
0	November 27	2006	$610	NaN
1	November 26	2007	$730	20%
2	December 1	2008	$8	
3	November 30	2009	$887	4.7%
4	November 29	2010	$1,028	16%
5	November 28	2011	$1,251	22%
6	November 26	2012	$1,465	17%
7	December 2	2013	$1,735	18%
8	December 1	2014	$2,038[23]	17%
9	November 30	2015	$2,280	12%
10	November 28	2016	$2,671	17%
11	November 27	2017	$3,364	26%
12	November 26	2018	$7,900	19.3%

Need to remove $, commas, footnotes

Cleaning the Cyber Monday DataFrame

```
cyber.drop(['Day', '% Change'], axis=1, inplace=True)
cyber.dropna(inplace=True)
cyber = cyber.rename({'Sales(millionsof USD)': 'Cyber Mon. (millions)'},
                     axis=1)
```

Drop unneeded columns

Rename other columns

```
cyber['Cyber Mon. (millions)'] =
  cyber['Cyber Mon. (millions)'].str.lstrip('$')
                                .str.replace(',', '')
                                .map(lambda val: str(val).split('[')[0])
                                .astype(int)
```

Remove unwanted $ signs, replace commas with empty strings, and remove the footnotes

Cleaned Up Version

	Year	Cyber Mon. (millions)
0	2006	610
1	2007	730
2	2008	846
3	2009	887
4	2010	1028
5	2011	1251
6	2012	1465
7	2013	1735
8	2014	2038
9	2015	2280
10	2016	2671
11	2017	3364
12	2018	7900

Merging the DataFrames

```
merged = pd.merge(bfriday, cyber, on='Year').sort_values(by='Year')
```

```
merged.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10 entries, 9 to 0
Data columns (total 5 columns):
Year                    10 non-null int32
Shoppers (millions)     10 non-null float64
Average spent           10 non-null float64
Total spent (billions)  10 non-null float64
Cyber Mon. (millions)   10 non-null int32
dtypes: float64(3), int32(2)
memory usage: 400.0 bytes
```

Provides another way to see if there are any NaN values

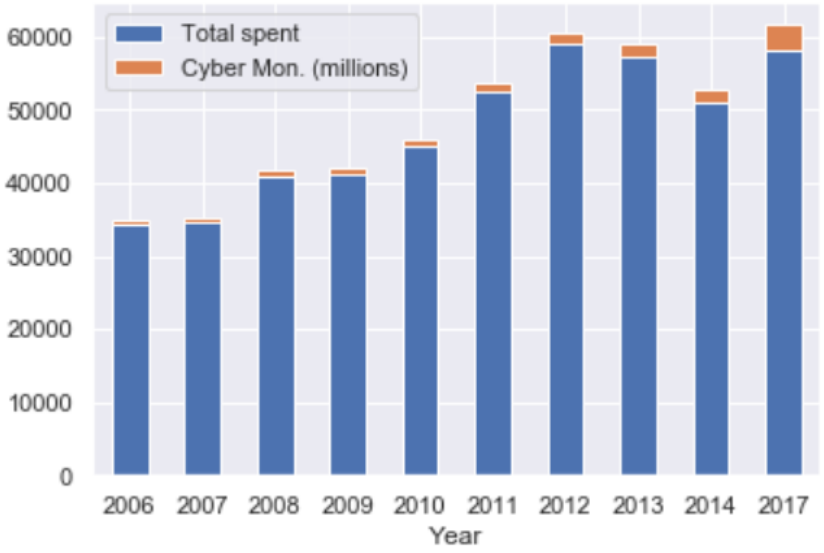
```
pd.isnull(merged).sum()
```

```
Year                    0
Shoppers (millions)     0
Average spent           0
Total spent (billions)  0
Cyber Mon. (millions)   0
dtype: int64
```

Final Results: Black Friday vs Cyber Monday

```
merged['Total spent'] = merged['Total spent (billions)'] * 1000
merged.plot(kind='bar', x='Year',
        y=['Total spent', 'Cyber Mon. (millions)'], rot=0, stacked=True)
```

	Year	Shoppers (millions)	Average spent	Total spent (billions)	Cyber Mon. (millions)
9	2006	140.0	360.15	34.4	610
8	2007	147.0	347.55	34.6	730
7	2008	172.0	372.57	41.0	846
6	2009	195.0	343.31	41.2	887
5	2010	212.0	365.34	45.0	1028
4	2011	226.0	398.62	52.5	1251
3	2012	247.0	423.66	59.1	1465
2	2013	249.0	407.02	57.4	1735
1	2014	233.0	380.95	50.9	2038
0	2017	174.0	335.47	58.3	3364



Using read_json()

- Pandas read_json() supports reading JSON-based data even in various formats:

```
import pandas as pd
data = '''
    [
        {"name": "Fang",    "type": "Dog",    "age": 3},
        {"name": "Aragog",  "type": "Spider", "age": 1},
        {"name": "Hedwig",  "type": "Owl",    "age": 2}
    ]
'''

df = pd.read_json(data, orient='records')
```

	age	name	type
0	3	Fang	Dog
1	1	Aragog	Spider
2	2	Hedwig	Owl

Different values for orient are used for different JSON structures: records, split, index, columns, and values

For data that doesn't match any of these formats, use Pandas' json_normalize() method. An example of retrieving live JSON data and using json_normalize() can be found in 14_normalizing_json_data.ipynb

Pandas Idioms

```
df = pd.DataFrame(data=np.arange(1, 10).reshape((3, 3)),
                  columns=['first', 'second', 'third'])
```

	first	second	third
0	1	2	3
1	4	5	6
2	7	8	9

- Split DataFrame based on a criterion

```
criterion = (df['first'] < 4)
df_one = df[criterion]
df_two = df[~criterion]
```

	first	second	third
0	1	2	3

	first	second	third
1	4	5	6
2	7	8	9

- Assign values to a column based on another column

```
df.loc[df['second'] >= 5, 'third'] = -1
```

	first	second	third
0	1	2	3
1	4	5	-1
2	7	8	-1

Pandas Idioms *(continued)*

```
df = pd.DataFrame(data=np.arange(1, 10).reshape((3, 3)),
                  columns=['first', 'second', 'third'])
```

	first	second	third
0	1	2	3
1	4	5	6
2	7	8	9

- Membership

```
df[df.index.isin([0, 1]) & df['second'].isin([1, 2, 3])]
```

	first	second	third
0	1	2	3

- Complement

```
df[~(df.index.isin([0, 1]) & df['second'].isin([1, 2, 3]))]
```

	first	second	third
1	4	5	6
2	7	8	9

Pandas Idioms *(continued)*

- Find first of each group

```
data = [
    ['Dick Cabbage', 'Tulsa'],
    ['Tina Turnip', 'Oklahoma City'],
    ['Elvis Parsley', 'Tulsa'],
    ['Antonio Banana', 'Norman'],
    ['Howie Mango', 'Oklahoma City'],
    ['Tom Shanks', 'Norman'],
]
df2 = pd.DataFrame(data=data, columns=['name', 'city'])
city_groups = df2.groupby('city')
print(city_groups.first().reset_index())
```

	city	name
0	Norman	Antonio Banana
1	Oklahoma City	Tina Turnip
2	Tulsa	Dick Cabbage

Use `city_groups.nth(1)` to get the second item of each group

Summary

- Merging Pandas DataFrames can be performed in a similar way to operating on tables within a database

- Pandas provides easier methods than NumPy to acquire data from external sources
  - These include:
    - read_csv()
    - read_json()
    - read_html()
    - read_sql()
    - read_sas()

Chapter 6

Python, Pandas,

and the Database

Making Use of Other Tools
Available for Python

Overview

Database Data Retrieval

Pandas and the Database

Python Database API 2.0

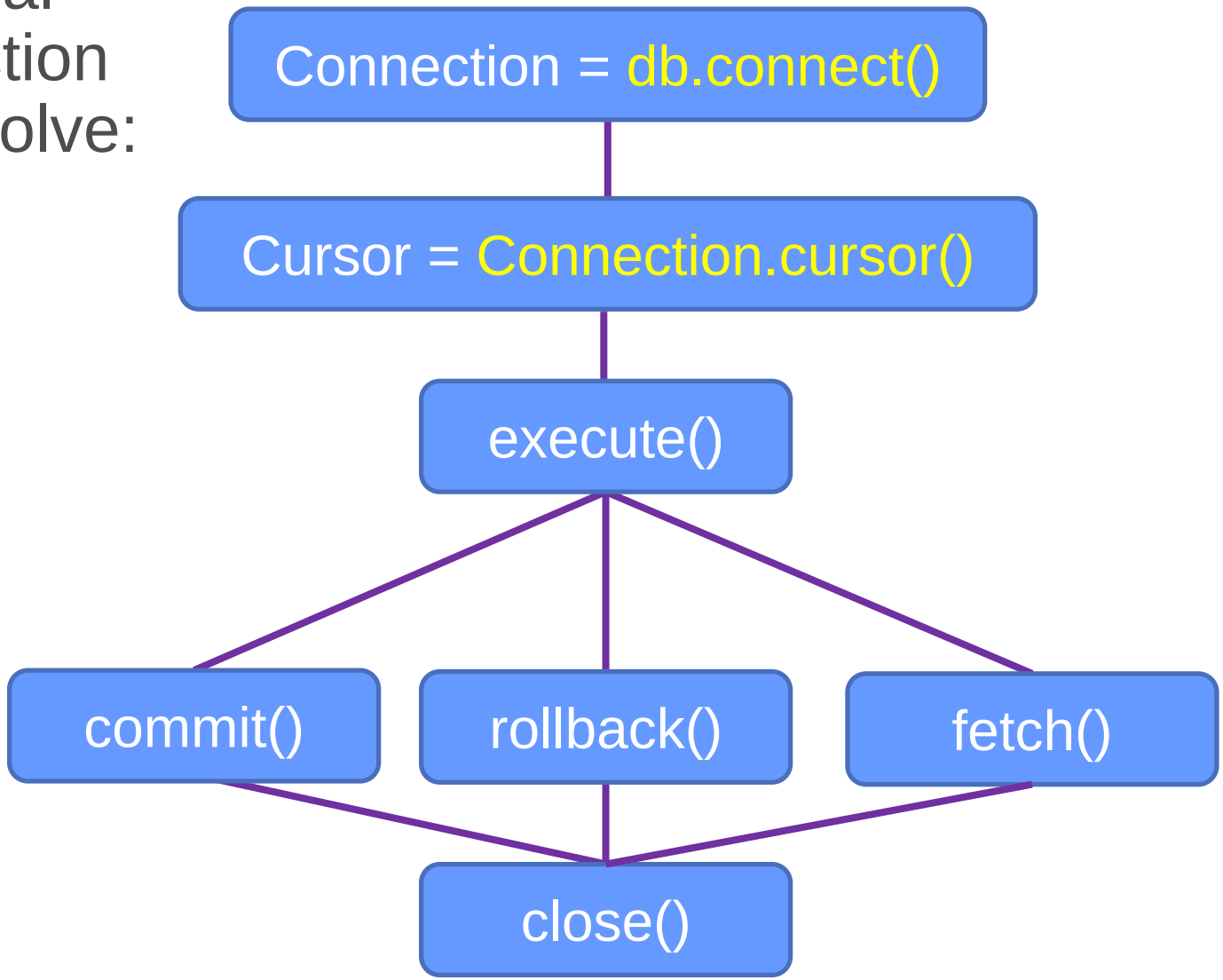
- The Python DB API 2.0 defines a common interface for modules to connect to and work with relational databases

- The interface defines:
  - Connection Objects and Transactions
  - Cursors Object Operations
  - Input, Output Data Types
  - Error Handling
  - Two-Phase Commits

There is no standard library module. Vendors or 3rd parties must provide the needed module.

Connections Cursors and More

- A typical interaction will involve:



Connections

- A connection object can be obtained from the connect() object's constructor
  - Parameters supplied will be database-specific
- Connection object methods:
  - cursor() - returns a cursor (next slide…)
  - commit() - for DB drivers not supporting TXs, this should provide void functionality
  - rollback() - optional for databases that do not support transactions
  - close() - close connection, connection no longer avail., close without commit() causes a rollback()

Cursors

- Cursors are used for fetch and execution
  - Multiple cursors made from the same connection are not isolated
  - Cursors support the following methods:

fetchone()
fetchmany(size=cursor.arraysize)
fetchall()
execute(sql, params)
executemany(sql, [params])
callproc(name, params)

nextset()
setinputsizes(sizes)
setoutputsize(size, col)
rowcount
description
close()

Fetch Methods

- fetchone() – returns a tuple representing one row of data or *None*
- fetchmany(size=n) – returns a list of n tuples
- fetchall() – returns all rows as a list of tuples

- You may also iterate over the cursor directory:

```
cursor.execute("SELECT account_id, name, balance FROM accounts")
for row in cursor:
    acct_num = row[0]
    name = row[1]
    balance = row[2]
```

Execute Methods

- execute(sql, params) – executes the provided sql

```
data = ('Bob', 100.0, 0.05, 'C')
cursor.execute(
"INSERT INTO accounts(name,balance,rate,acct_type) VALUES \
(?,?,?,?)", data)
```

- executemany(sql, [params]) – execute sql repeatedly against all supplied params

```
data = [('John Smith', 5500.0, 0.025, 'C'),
        ('Sally Jones', 6710.11, 0.025, 'C'),
        ('Fred Green', 2201.73, 0.035, 'S'),
        ('Ollie Engle', 187.30, 0.025, 'S'),
        ('Gomer Pyle', 12723.10, 0.015, 'C')
        ]

cursor.executemany("INSERT INTO accounts \
    (name, balance, rate, acct_type) VALUES (?,?,?,?)", data)
```

SQLite

- SQLite is an in-process relational database
  - Don't have to start a separate application to run it
  - Runs entirely within your current Python app
  - To use it import the module sqlite3
  - Can be an in-memory or file-based database
  - Not typically used in large-scale production, but useful for development, testing, smaller efforts
- Connect using the sqlite.connect() method

```
import sqlite3
connection = sqlite3.connect('mysqlite.db')
```

Creating the Database

```
import sqlite3

connection = None
try:
    connection = sqlite3.connect('schools.db')
    cursor = connection.cursor()
    cursor.execute(DROP_SCHOOLS_SQL)
    cursor.execute(CREATE_SCHOOLS_SQL)
    cursor.executemany('INSERT INTO schools(school_id, \
                        fullname, city, state, country)\
                        VALUES (?,?,?,?,?)', school_data)
    connection.commit()
    print('Data loaded into schools table')
except sqlite3.Error as err:
    if connection:
        connection.rollback()
    print('Data not loaded into schools table')
    print('Error: {0}'.format(err))
finally:
    if connection:
        connection.close()
```

School_data contains a list of tuples of schools

Using 'with' in SQLite3

The with control can be used on the connection object. It will *commit()* if no errors occur otherwise it will automatically *rollback()* if an error occurs.

```
try:
    with sqlite3.connect('schools.db') as connection:
        cursor = connection.cursor()
        cursor.execute(DROP_SCHOOLS_SQL)
        cursor.execute(CREATE_SCHOOLS_SQL)
        cursor.executemany(INSERT_RECORD, school_data)
        print('Data loaded into schools table')
except sqlite3.Error as err:
    print('Data not loaded into schools table')
    print('Error: {0}'.format(err))
finally:
    if connection:
        connection.close()
```

The connection is not automatically closed by the with control.

Accessing Data

```
import sqlite3

school_data = []
connection = None
state = input('Schools from which state: ').upper()
try:
    connection = sqlite3.connect('schools.db')
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM schools WHERE state=?', (state,))
    for sch in cursor:
        school_data.append((sch[0], sch[1], sch[2], sch[3],
                            sch[4]))
except sqlite3.Error as e:
    print('Error: {0}'.format(e))
finally:
    if connection:
        connection.close()
```

or append(sch)

Accessing Data By Column Names

```
import sqlite3
from collections import namedtuple
School = namedtuple('School', 'school_id name city state country')
school_data = []
connection = None
state = input('Schools from which state: ').upper()
try:
    connection = sqlite3.connect('schools.db')
    connection.row_factory = sqlite3.Row
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM schools WHERE state=?', (state,))
    for sch in cursor:
        school_data.append(School(sch['school_id'],
                                  sch['fullname'], sch['city'],
                                  sch['state'], sch['country']))
except sqlite3.Error as e:
    print('Error: {0}'.format(e))
finally:
    if connection:
        connection.close()
```

Enables accessing rows by index names not numbers

Working with Other RDBMSs

- Different databases require installing separate modules:

MySQL	pip install MySQL-python	(MySQLdb Driver)
MySQL	pip install mysql-connector-python	(Connector Python Driver)
MS SQL Server	pip install pyodbc	(Pyodbc Driver)
PostgreSQL	Win: http://stickpeople.com/projects/python/win-psycopg/ OS X: export PATH=$PATH:/Applications/Postgres.app/Contents/Versions/9.3/bin/ pip install psycopg2	
IBM DB2	pip install ibm_db	(ibm_db Driver)
Oracle 10g - 12.1	pip install cx_Oracle	(cx_Oracle Driver)

MySQL Example

This example uses the connector-python MySQL driver

```
import mysql.connector
try:
    conn = mysql.connector.Connect(**config)
except mysql.connector.Error as err:
    results.append('Conn. Error: {0}'
                               .format(err))
    return results

cursor = conn.cursor()

try:
    cursor.execute(DROP_SCHOOLS_SQL)
    cursor.execute(CREATE_SCHOOLS_SQL)
    cursor.executemany(INSERT_RECORD, school_data)
    conn.commit()
    results.append('Completed reading records.')
except (mysql.connector.Error, TypeError) as err:
    results.append('Operation Error: {0}'.format(err))
    conn.rollback()
finally:
    if conn:
        conn.close()
```

```
config = {
    'host': 'localhost',
    'port': 3306,
    'database': 'test',
    'user': 'user1',
    'password': 'password',
    'charset': 'utf8',
    'use_unicode': True,
    'get_warnings': True
}
```

Pandas and the Database

- Use pd.read_sql(sql, conn, params) to read data directly from a database into a DataFrame

- Arguments to read_sql() include:
  - sql - the sql statement to perform
  - conn - the already created connection to the database
  - index_col - the db column to use for the DataFrame index
  - params - list of parameters to use
  - columns - list of column names to use from the table

Pandas Database Example

```
import pandas as pd
import sqlite3

with sqlite3.connect('batting.db') as conn:
    df = pd.read_sql('SELECT hits, atbats FROM batting WHERE year >= ?',
                     conn, params=['1957'])

df = df[df.hits != 0]
df = df[df.atbats >= 502]

df['averages'] = df['hits'] / df['atbats']
print(df.describe())
print('50% batting average: {0}'
                            .format(df.describe()['averages']['50%']))
```

Loads up selected query data into a DataFrame

Removes undesired data rows

Add the new averages column to the DataFrame

Your Turn! - Task 6-1

- Using **`batting.db`**, determine the highest paid baseball player and the year played

Salaries table

year	team	league	playerid	salary

Players table

playerid	firstname	lastname

- While some steps can be performed within a database, we'll use Pandas for this exercise
  - Read the tables into two dataframes from batting.db (a sqlite3 database)
  - Use *df*['salary'].idxmax() to determine the row for the max salary
  - Use the row for the max salary to get the playerid and year
  - With the playerid, query the Players table to get the firstname and lastname

Summary

- Python doesn't have a single, standard module when working with a database
  - A different module is required for each database
- Python has a standard interface for working with databases called the Python DB API 2.0
- The standard Python distribution provides a built-in database module called sqlite3

Chapter 7

Data Visualization

Using Seaborn

Overview

Seaborn vs Matplotlib

Seaborn Datasets

Seaborn APIs

Seaborn vs Matplotlib

- *Seaborn* is a Python data visualization library built upon Matplotlib
  - Works with NumPy and Pandas
  - Understands Pandas labels when defining axes
  - Provides additional features beyond Matplotlib such as:
    - Easier use of color palettes and themes
    - Visualizations for linear regressions
    - Additional data plot types
- Seaborn, like Matplotlib, must be installed:

  conda install seaborn or pip install seaborn

  - If using Anaconda 4.3+, it will already be installed

Starting with Seaborn

- Seaborn is imported using: `import seaborn as sns`
  - If using it within Jupyter, it will require: `%matplotlib inline`
- Use the sns.set() call to configure different styles

```
sns.set(context='notebook', style='darkgrid', palette='deep', ...)
```

  - sns.set() by itself sets styles to the default values

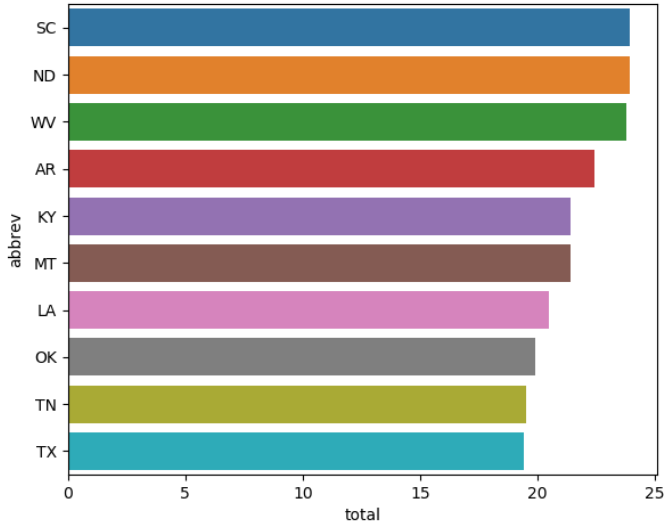
Retrieving Seaborn Datasets

- Seaborn has the ability to retrieve datasets from a github repository using load_dataset()
  - Returns a Pandas DataFrame

https://github.com/mwaskom/seaborn-data

anscombe.csv	Add anscombe dataset	4 years ago
attention.csv	Add attention dataset	4 years ago
brain_networks.csv	Add brain networks dataset	4 years ago
car_crashes.csv	Add 538 car crash dataset	3 years ago
dots.csv	Add dots dataset	8 months ago
exercise.csv	Add exercise dataset	4 years ago
flights.csv	Add flights dataset	4 years ago
fmri.csv	Change sorting of events in fmri data	7 months ago
gammas.csv	Make fake fmri data make a bit more sense	4 years ago
iris.csv	Add iris dataset	4 years ago
planets.csv	Add planets dataset	4 years ago
tips.csv	Add tips dataset	4 years ago
titanic.csv	Update titanic datset to remove index variable	4 years ago

Plotting Seaborn Datasets



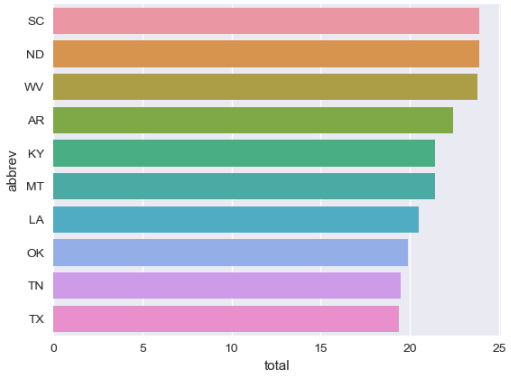

```
import matplotlib.pyplot as plt
import seaborn as sns

crashes = sns.load_dataset('car_crashes')
print(crashes.head())
print(crashes.shape)
most = crashes.sort_values(by='total', ascending=False).head(10)
sns.barplot(x='total', y='abbrev', data=most)
plt.show()
```

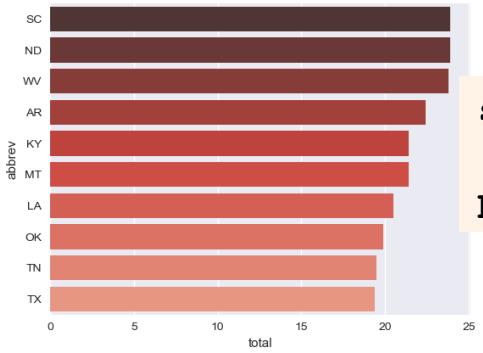
	total	speeding	alcohol	not_distracted	no_previous	ins_premium	ins_losses	abbrev
0	18.8	7.332	5.640	18.048	15.040	784.55	145.08	AL
1	18.1	7.421	4.525	16.290	17.014	1053.48	133.93	AK
2	18.6	6.510	5.208	15.624	17.856	899.47	110.35	AZ
3	22.4	4.032	5.824	21.056	21.280	827.34	142.39	AR
4	12.0	4.200	3.360	10.920	10.680	878.41	165.63	CA

(51, 8)

Different Seaborn Sets

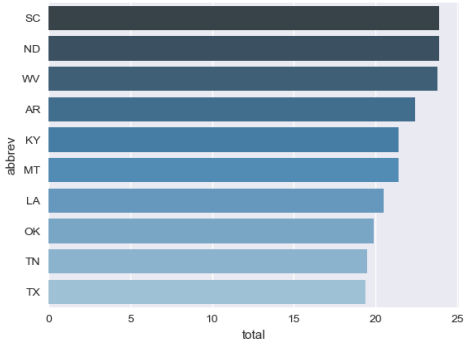
```
sns.set()
sns.barplot(x='total', y='abbrev', data=most)
plt.show()
```

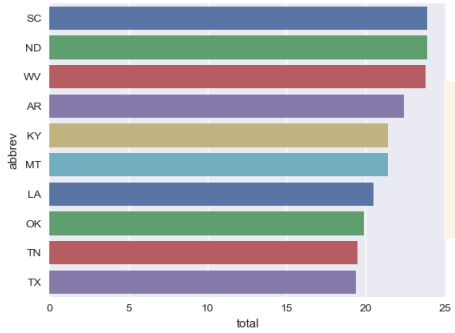





```
sns.barplot(x='total', y='abbrev', data=most,
            palette='Reds_d')
plt.show()
```

```
sns.barplot(x='total', y='abbrev', data=most,
            palette='Blues_d')
plt.show()
```



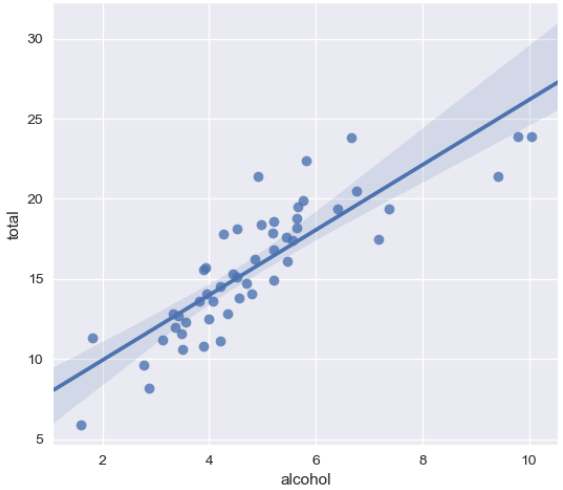



```
sns.barplot(x='total', y='abbrev', data=most,
            palette='deep')
plt.show()
```

Seaborn Plotting

- Seaborn provides numerous plot functions
  - Most accept a DataFrame using the data= parameter
  - Provide x= and y= parameters to define axis data values

```
sns.lmplot(x='alcohol', y='total', data=crashes)
```



- Seaborn provides the following types of plots:

countplot()
violinplot()
boxplot()
Categorical
barplot()
pointplot()
lvplot()
stripplot()
swarmplot()

pairplot()
factorplot()
Axis
jointplot()
lmplot()

kdeplot()
rugplot()
Distribution
distplot()

heatmap()
Matrix
clustermap()

regplot()
Regression
residplot()

Understanding Your Data

- Seaborn can help provide *exploratory analysis* of our data
  - We can plot multiple *features* against our chosen *response* (we are using *total*)

Our Response Variable

Our Features

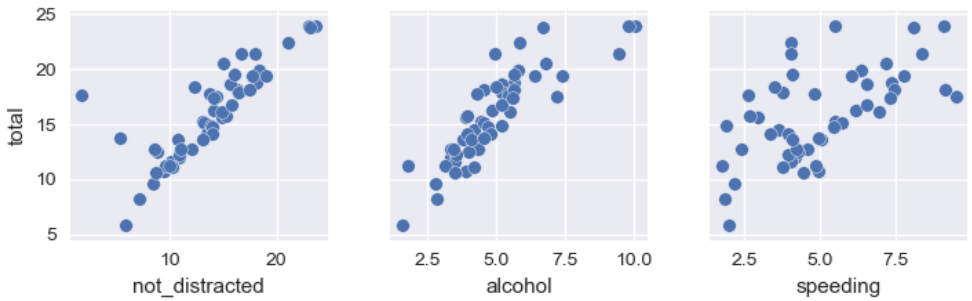
	total	speeding	alcohol	not_distracted	no_previous	ins_premium	ins_losses	abbrev
0	18.8	7.332	5.640	18.048	15.040	784.55	145.08	AL
1	18.1	7.421	4.525	16.290	17.014	1053.48	133.93	AK
2	18.6	6.510	5.208	15.624	17.856	899.47	110.35	AZ
3	22.4	4.032	5.824	21.056	21.280	827.34	142.39	AR
4	12.0	4.200	3.360	10.920	10.680	878.41	165.63	CA

Because the response, *total,* is continuous, we can use a regression...

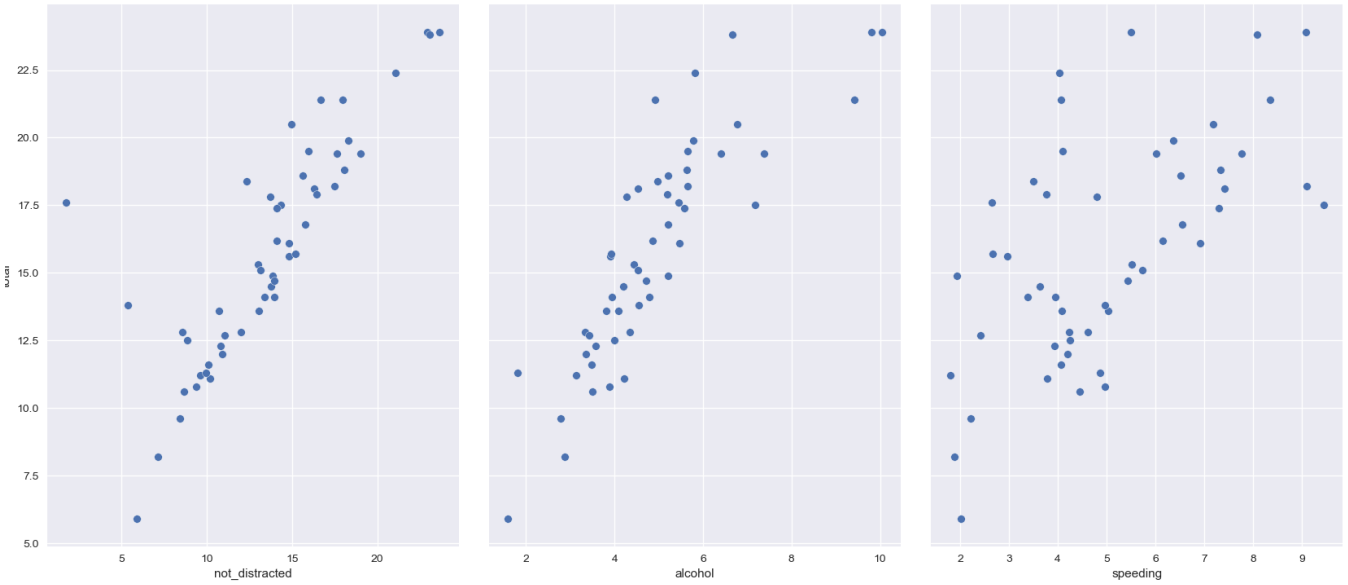
Python 3
Data Analysis

Using pairplot() to Understand Features

```
sns.pairplot(crashes, x_vars=['not_distracted', 'alcohol', 'speeding'], y_vars='total')
```



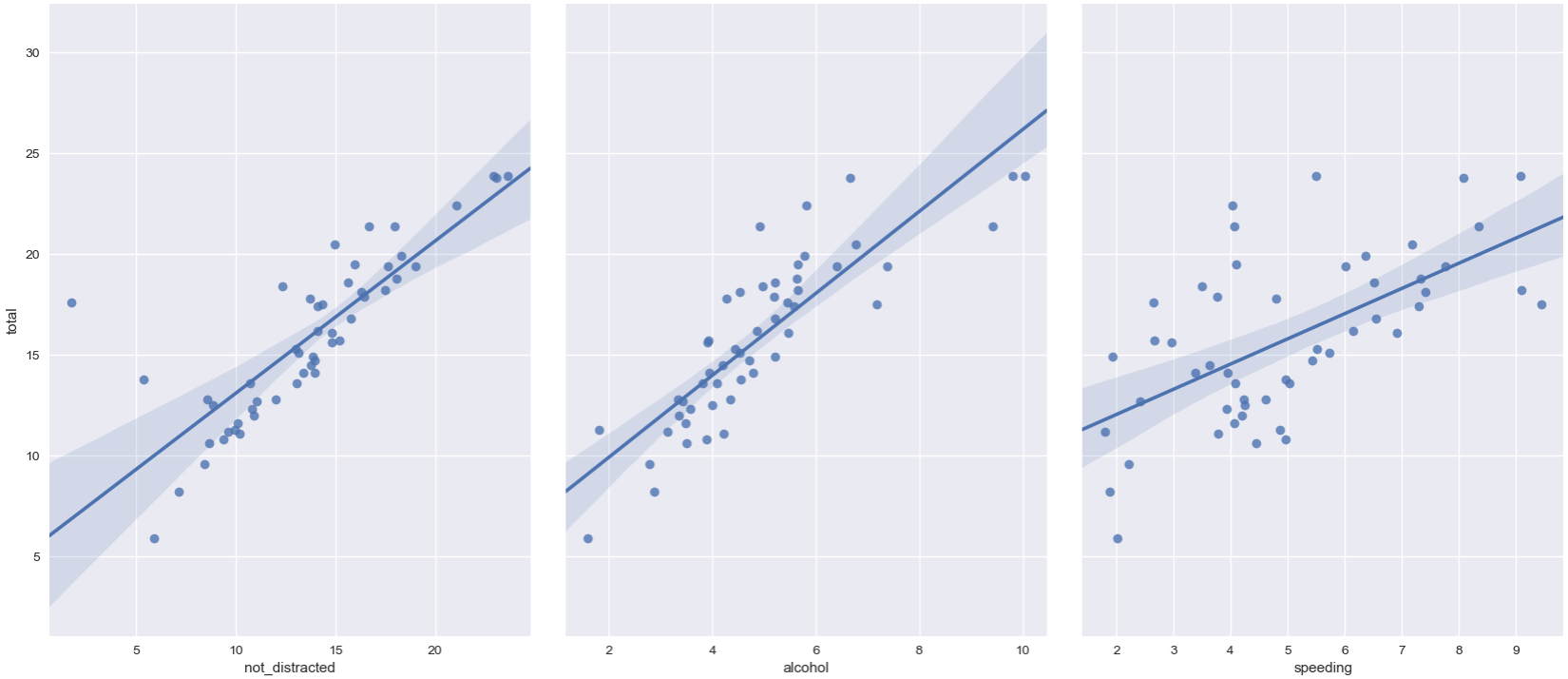

```
sns.pairplot(crashes, x_vars=['not_distracted', 'alcohol', 'speeding'], y_vars='total',
             aspect=0.75, size=7.5)
```



Python 3
Data Analysis

Adding a Linear Model and Confidence Interval

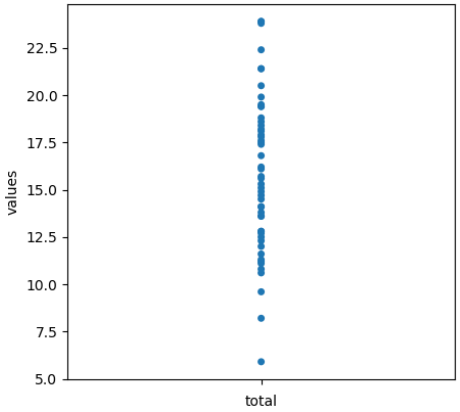
```
sns.pairplot(crashes, x_vars=['not_distracted', 'alcohol', 'speeding'], y_vars='total',
             kind='reg')
```



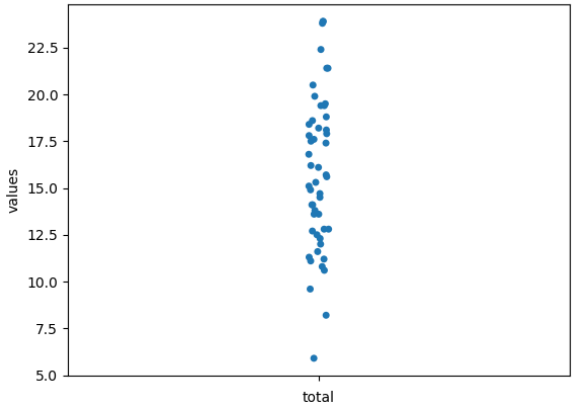
Strip, Swarm, and Box Plots

- Strip plots show a scatter plot when one axis is categorical
  - Jitter provides easier view when data points overlap

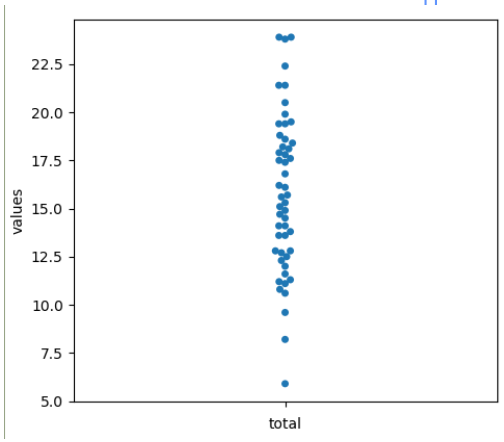
```
crashes = sns.load_dataset('car_crashes')
```



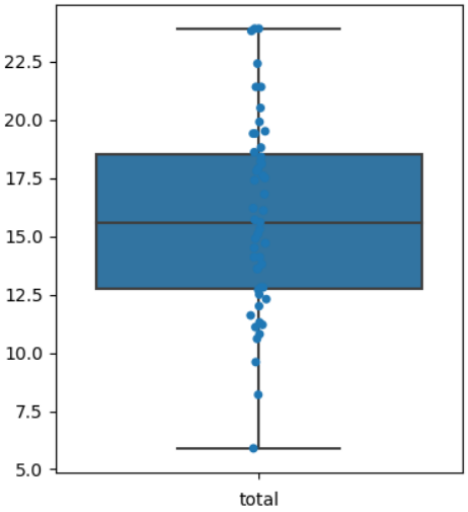

```
sns.stripplot(y='total',
              data=crashes)
```




```
sns.stripplot(y='total',
              data=crashes,
              jitter=0.02)
```




```
sns.swarmplot(y='total',
              data=crashes)
```




```
sns.boxplot(y='total',
            data=crashes)
```

Swarm and box plots can be useful in determining higher densities and outlier detection

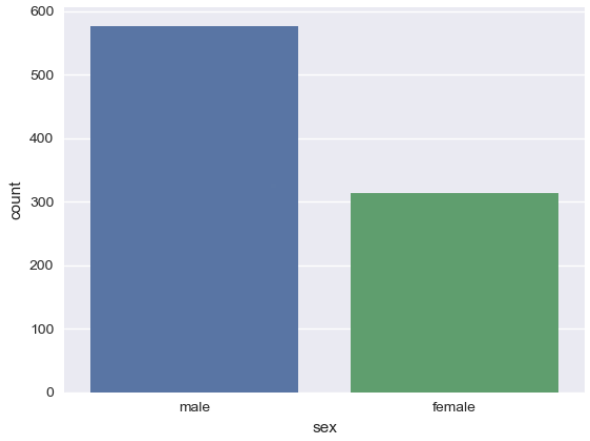
Count Plots

- Count plots can show the count totals for categories as a bar

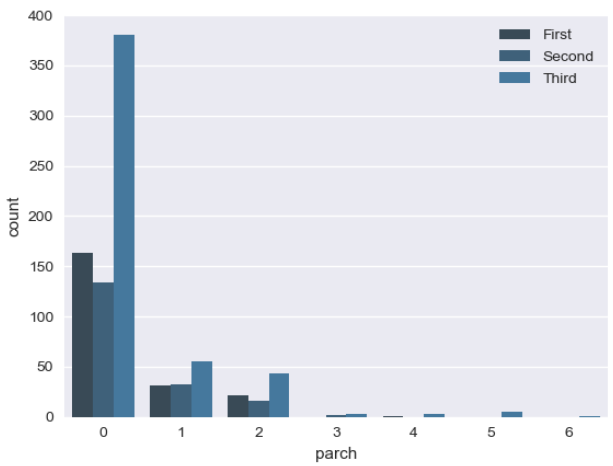
```
titanic = sns.load_dataset('titanic')
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

```
sns.countplot(x='sex',
              data=titanic)
```




```
sns.set(palette='Blues_d')
sns.countplot(x='parch', data=titanic, hue='class')
```

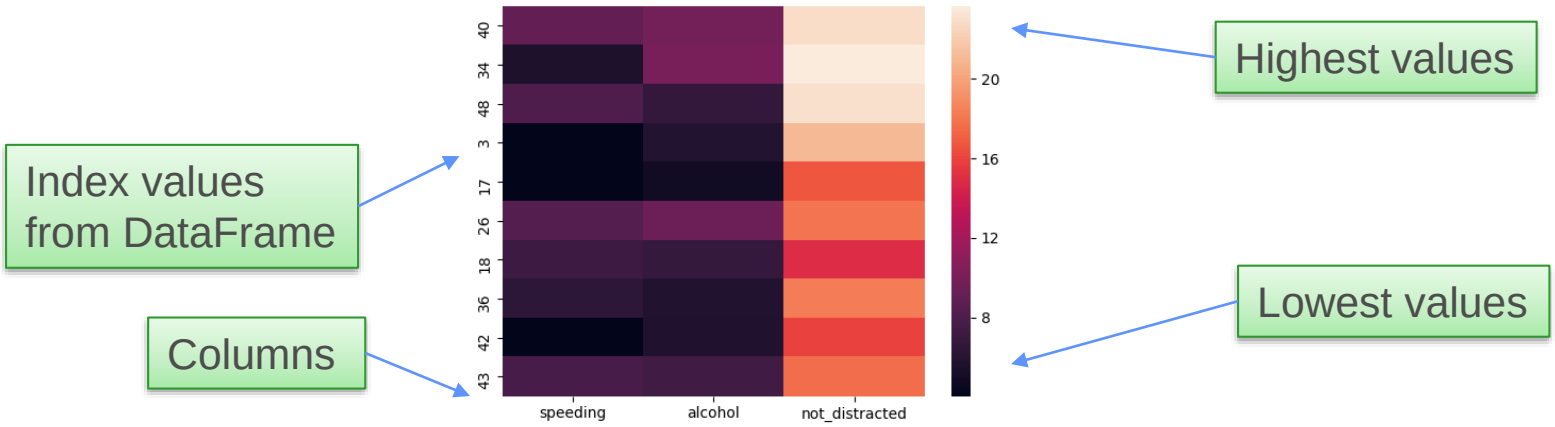


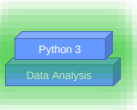
Heatmap

Heatmaps provide a visual representation of data values

	speeding	alcohol	not_distracted
40	9.082	9.799	22.944
34	5.497	10.038	23.661
48	8.092	6.664	23.086
3	4.032	5.824	21.056
17	4.066	4.922	16.692
26	8.346	9.416	17.976
18	7.175	6.765	14.965
36	6.368	5.771	18.308
42	4.095	5.655	15.990
43	7.760	7.372	17.654

```
import matplotlib.pyplot as plt
import seaborn as sns

crashes = sns.load_dataset('car_crashes')
most = crashes.sort_values(by='total', ascending=False).head(10)

sns.heatmap(most[['speeding', 'alcohol', 'not_distracted']])
plt.show()
```





Summary

- Seaborn can use several datasets for test purposes
- Seaborn provides numerous plot types
- These plot types are usually simpler to use than attempting the plots directly using Matplotlib

Your Turn! - Task 7-1

- Using Seaborn pairplot() plot the response (averages) against the features year, atbats, and hits
  - We will only examine year > 2000
  - Use a hue in the pairplot() set to the year
  - Does there appear to be a relationship between any of these features and the response?
  - Work from and complete task7_1_starter.py

Chapter 8

Introducing

Scikit-learn

Overview

Machine Learning

Scikit-learn

What is Scikit?

- Scikits (SciPy Toolkits) are a series of Python-based addons to SciPy
  - There are over 70+ Scikits
  - They are too specific or too large to be included in SciPy
  - List of toolkits can be found here: https://scikits.appspot.com/scikits
  - *Two toolkits* that predominate include:
    - Scikit-learn
      - Data mining / Machine learning toolkit
      - Allows for the creation and application of models against datasets to predict outcome
    - Scikit-image
      - Provides tools for image filtering, morphing, exposure, etc.

Scikit-learn (sklearn)

- Provides tools for data mining and machine learning
  - Built upon SciPy (Pandas, NumPy, Matplotlib)

Created by David Cournapeau in 2007 as a Google Summer of Code Project

- *Emphasizes modeling*
  - *Data loading and manipulation s*hould be accomplished beforehand using Pandas or NumPy

- Already installed in Anaconda
  - If not installed, you can use pip:

```
pip install scikit-learn
```

What is Machine Learning?

- Machine Learning is the act of predicting output from unknown data
  - The computer "learns" from the data it is provided by considering patterns, likelihood, and groupings
- ML models are used to *predict output responses* or *identify structure from the data*
  - Most ML models are mathematically described
- Machine learning commonly falls into two broad categories:
  - Supervised learning (most common learning form)
  - Unsupervised learning

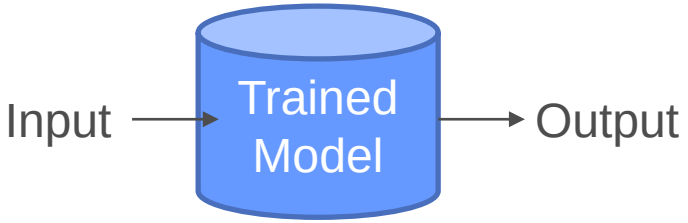
Important ML Terms

- *Observation* - *a row of data*, also referred to as a sample, record, or instance
- *Feature* - *a column of data*, also referred to as a predictor, input, independent variable, attribute, covariate
- *Label* - *output value*, response, target, outcome, dependent variable

Supervised Learning

- Sometimes referred to as *predictive modeling*
- Supervised learning involves creating a model that has been "trained" based on known data
  - It is called supervised because we have knowledge of the response for the provided inputs

- That model when properly expressed will accept input variables (X) and produce an predictive output (Y): $Y = f(X)$

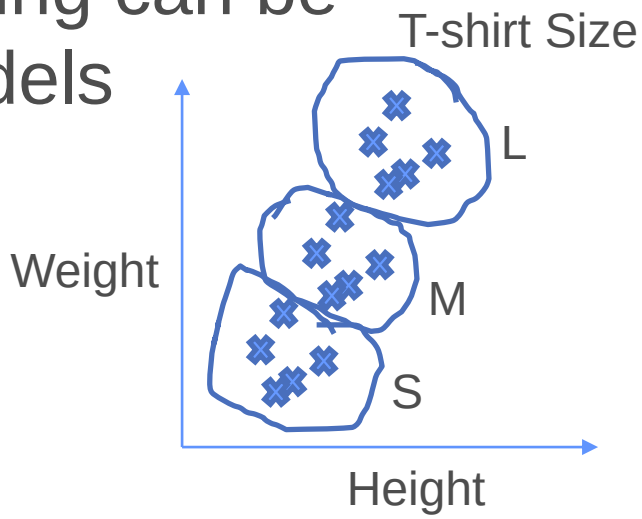


Unsupervised Learning

- Unsupervised learning uses features but *without a known response*
  - There's no knowledge of the outcome or results
  - Data is unlabeled (*there is no Y*)

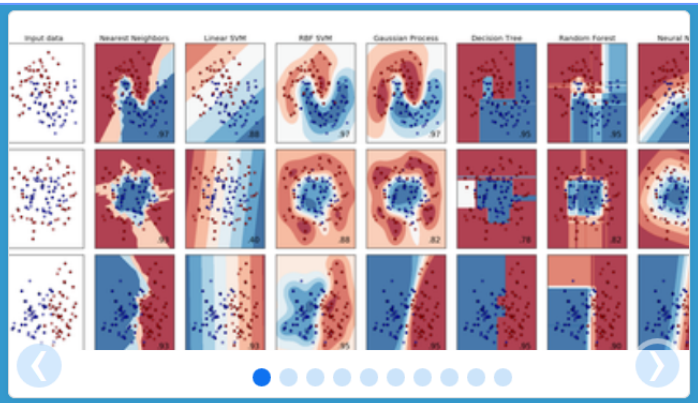
- Structure is extracted (learned) from the data

- Examples of unsupervised learning can be found in consumer behavior models
  - Segmenting consumers, voters, coins, pictures, animals or any items helps us determine how other items with similar characteristics may behave



Python 3
Data Analysis

Machine Learning and Scikit-learn Tools



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

http://scikit-learn.org

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ...

— Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ...

— Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, ...

— Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization.

— Examples

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics.

— Examples

Preprocessing

Feature extraction and normalization.

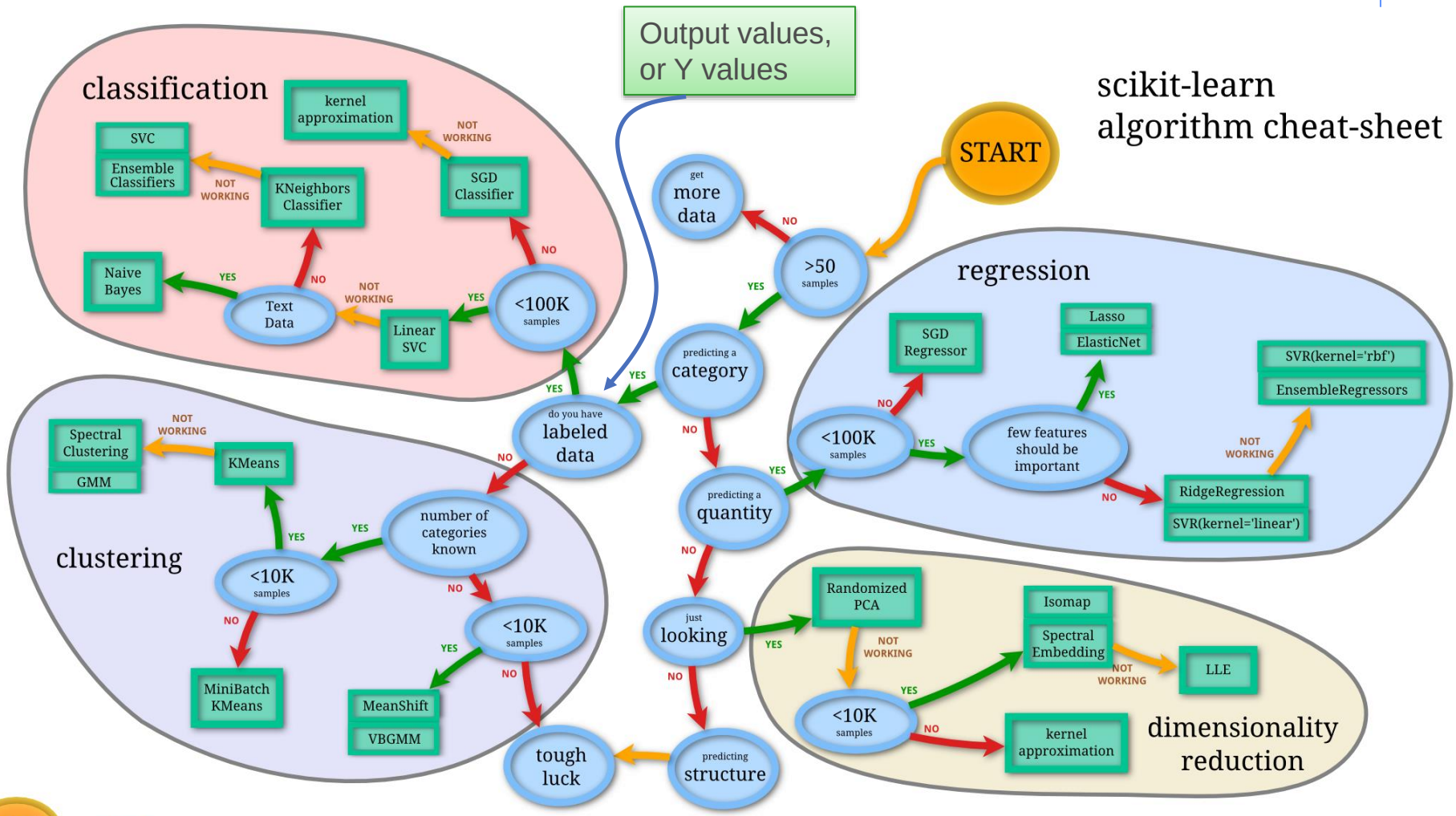
Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction.

— Examples

Scikit Supervised Learning Support

- Supervised learning comes in two broad forms:
  - **Regressions -** involve continuous response values
  - **Classifications -** discrete-valued (finite) responses
- Sklearn supervised learning algorithms include:
  - Ordinary Least Squares (OLS)
  - Ridge
  - Lasso
  - Elastic Net
  - Bayesian
  - Logistic
  - Support Vector Machines
  - Decision Trees
  - Ensemble Methods
  - Others...

Determining Which Algorithm To Use



Back

Regressions and Linear Regressions

- Regressions are a type of *supervised learning* in which the *response* is continuous
- Linear Regressions are a specific type of model used in regression problems
  - Advantages:
    - Easy to implement, well-understood by many
    - Fast algorithm (important for large datasets)
    - No tuning required, unlike KNN
    - Easy to interpret/understand
  - Disadvantages:
    - Not as accurate as other models due to its linear nature which doesn't often model the real-world

Linear Regressions

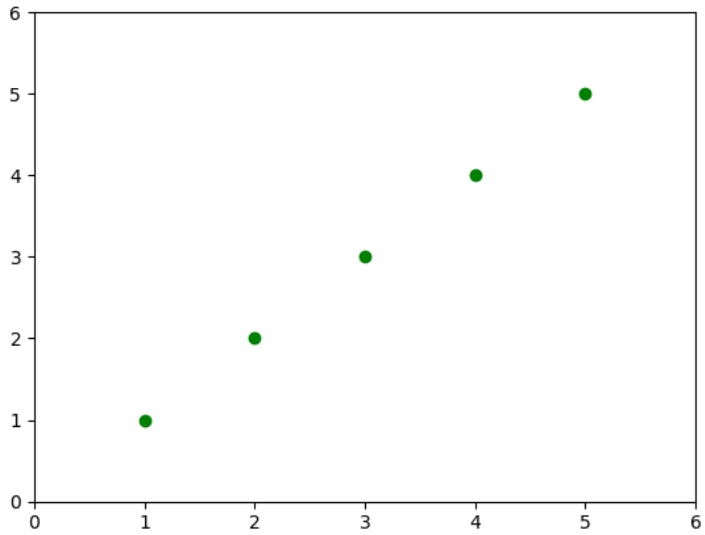
- Linear Regressions use a "best fit" model
  (smallest sum of squares difference Y - Y')
- Consider the following data points:

```
points = [(1, 1), (2, 2),
          (3, 3), (4, 4), (5, 5)]
```

```
data_array = np.array(points)

X = data_array[:, 0]
y = data_array[:, 1]
plt.plot(X, y, 'go')
plt.xlim(0, 6)
plt.ylim(0, 6)
plt.show()
```

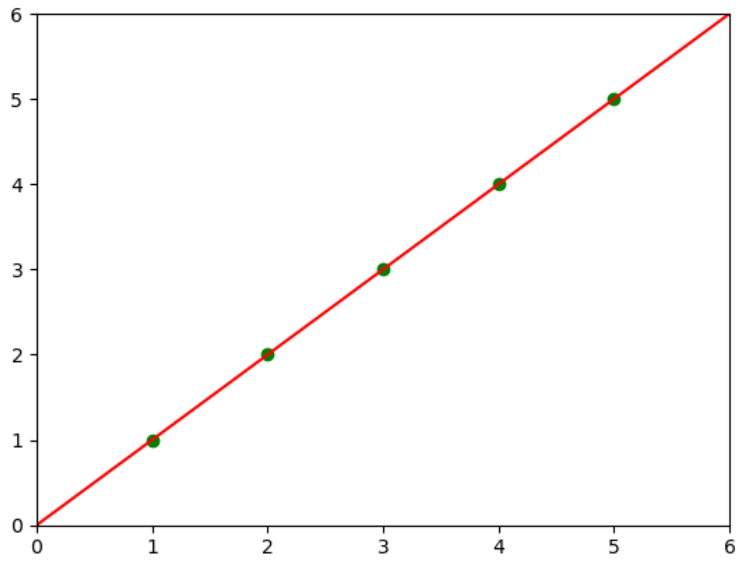
x-values
[1 2 3 4 5]

y-values
[1 2 3 4 5]



Linear Regression Plotted

- A line can be made that creates the smallest value of the difference of the sum of the squares
  - Y -Y' = 0 for all data values in this example



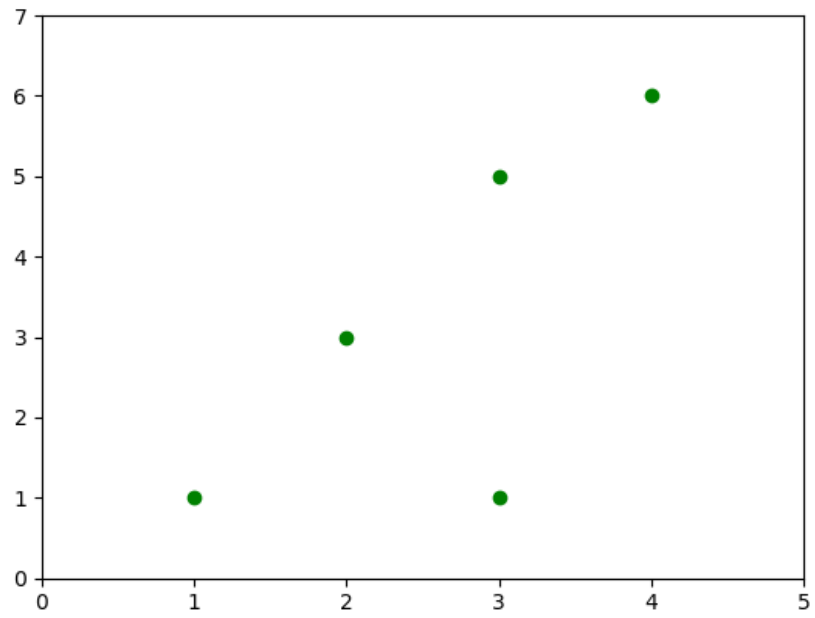

```
from sklearn.linear_model import LinearRegression

X_reshaped = X[:, None]

lm = LinearRegression()
lm.fit(X_reshaped, y)
x_test = np.linspace(0, 6, 37)[:, None]
y_test = lm.predict(x_test)
plt.plot(x_test, y_test, 'r-')
plt.show()
```

```
[
  [1]
  [2]
  [3]
  [4]
  [5]
]
```

Non-linear Data Points

- If data points are not-exactly on a line, a best fit will be made:

```
points = [(1, 1), (2, 3),
          (3, 1), (3, 5), (4, 6)]
```




```
data_array = np.array(points)

X = data_array[:, 0]
y = data_array[:, 1]
plt.plot(X, y, 'go')
plt.xlim(0, 5)
plt.ylim(0, 7)
plt.show()
```

[1 2 3 3 4]

[1 3 1 5 6]

Creating the Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

```
X_reshaped = X[:, None]
```

X

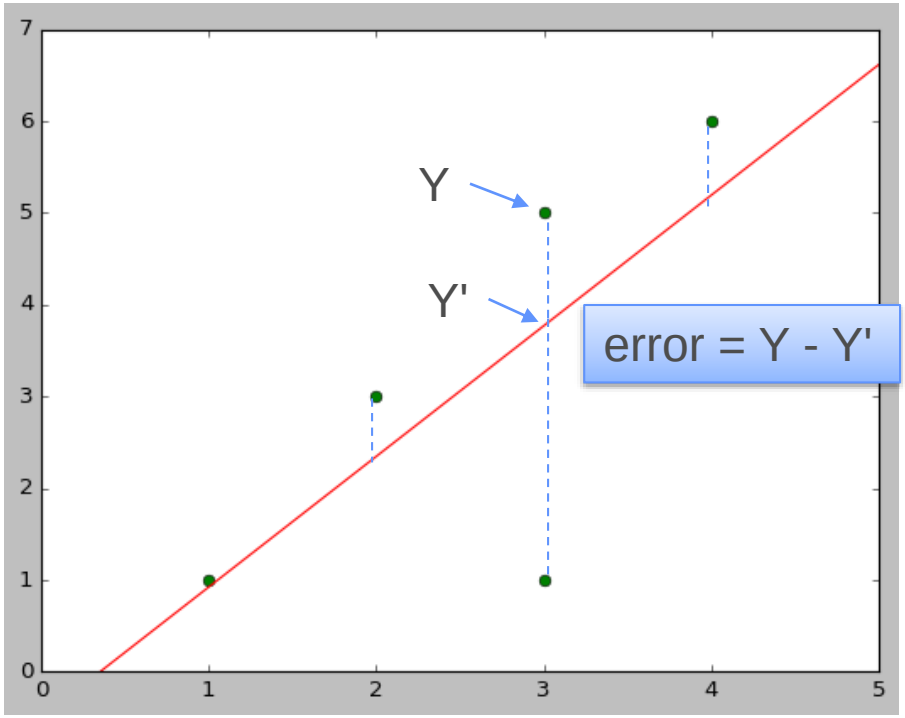
```
[1 2 3 3 4]
```

X-reshaped

```
[
   [1]
   [2]
   [3]
   [3]
   [4]
]
```

```
lm = LinearRegression()

lm.fit(X_reshaped, y)
x_test =
  np.linspace(0, 5, 31)[:, None]
y_test = lm.predict(x_test)
plt.plot(x_test, y_test, 'r-')
plt.show()
```

predict() can be used to "predict" future Y values from the model.



Determining Model Accuracy

- Additional values from the regression may be obtained:

```
print(lm.intercept_)
```

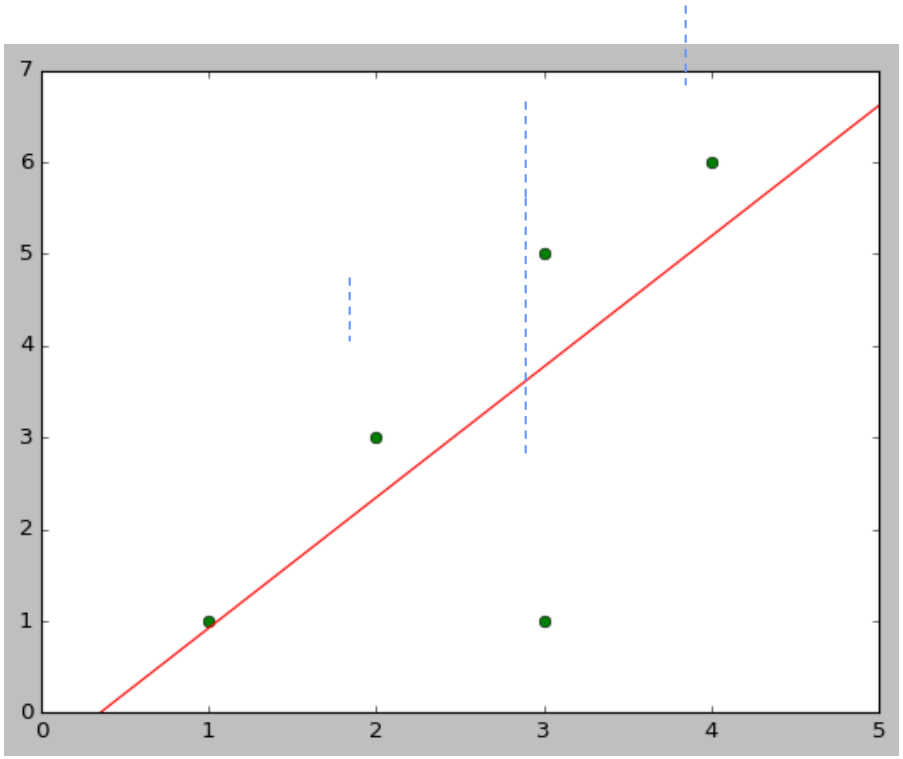
-0.5

```
print(lm.coef_)
```

[1.42307]

```
print(lm.score(x_values,
               y_values))
```

0.77



Alcohol-Related Crashes

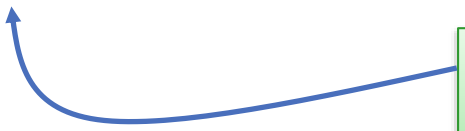
- As an example, we'll fit a model to our car crashes dataset by examining only the alcohol-related crashes against the total crashes

```
sns.lmplot(x='alcohol', y='total', data=crashes)
```

Creating Test and Training Data

- For proper model evaluation, samples should be broken into training data (to construct the model) and test data (to evaluate it)

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

crashes = sns.load_dataset('car_crashes')

X = crashes['alcohol']
y = crashes['total']

# splits 75-25 train-test by default
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)

lm = LinearRegression()
lm.fit(X_train[:, None], y_train)

y_pred = lm.predict(X=X_test[:, None])
```



These values are predicted based on the model, but how well do they hold up?

Evaluating the Model

- In a regression, accuracy is not useful, instead we'll use continuous value metrics for evaluating the model

```
print(y_pred)
print(y_test.values)
```

```
[ 13.9  16.5  15.1  25.4  16.5   9.4  12.2  17.3  13.1  16.5  17.8  12.912.9]
[ 14.1  18.6  13.8  21.4  16.8  11.3  11.2  17.4  12.3  14.9  22.4  11.6 10.6]
```

- We'll use the sklearn.metrics module to provide our error metric functions:

```
from sklearn.metrics import mean_squared_error

np.sqrt(mean_squared_error(y_test.values, y_pred))
```

2.11655482118

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (P_i - O_i)^2}{n}}$$

The RMSE can be directly compared to the Y test values to see how much error is introduced in this model

Summary

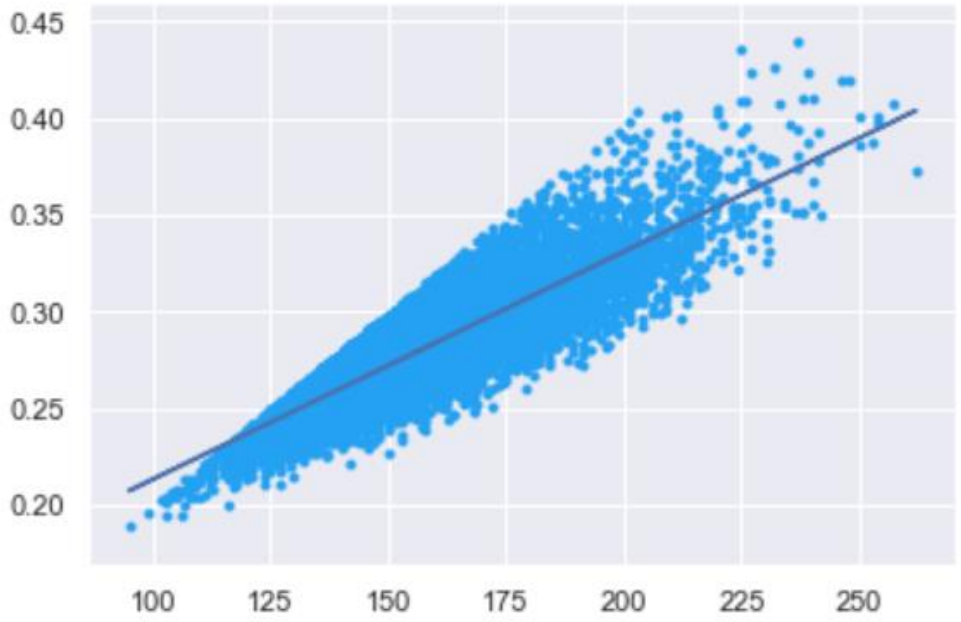
- Toolkits such as Scikit-learn provide ML algorithms that help build models that can accurately predict the behavior of systems given unknown data

- Both supervised and unsupervised algorithms exist within Scikit-learn

Your Turn! - Task 8-1

- Create a linear regression model that predicts batting average (the response) given the number of hits (the input) a person has

- Create a plot of hits vs averages and the linear regression line



Work from the provided Task 8-1 starter.ipynb file

Course Summary

What did we learn ?

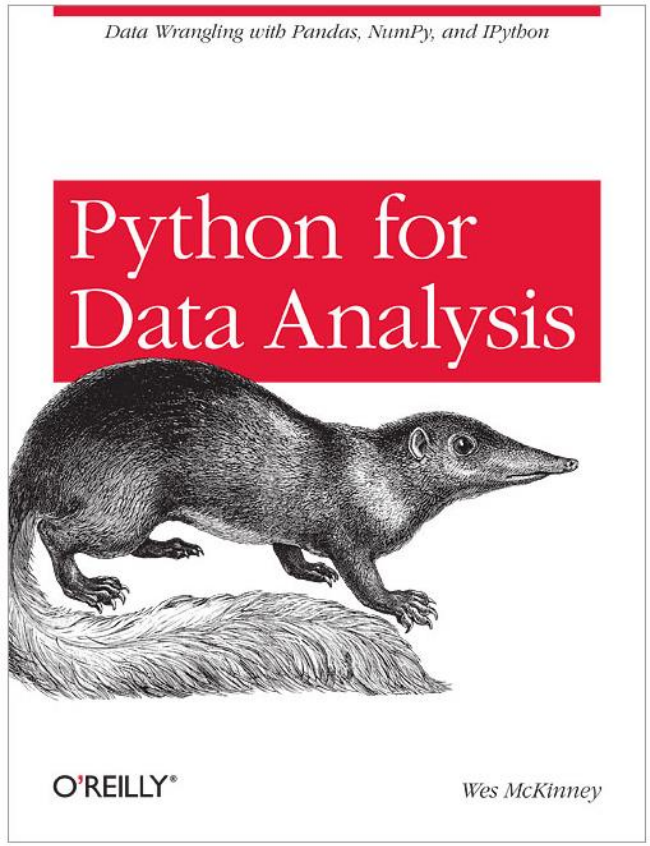
- Using NumPy, ndarray
- IPython
- Jupyter Notebooks
- Data Analysis Libraries
- Pandas Data Structures
- Series
- DataFrames
- Sorting DataFrame Data
- Grouping Data: Splitting, Applying, Combining
- Matplotlib
- Plotting Different Types
- Python Database APIs
- Pandas and Databases
- Working with CSV and Excel Data Files
- Indexing using loc[], iloc[]
- MultiIndexing
- Merging DataFrames
- Other Data Analysis Tools
- Creating, Working with Threads
- Other Performance and Data Analysis Tools Overview: Dask, Bottleneck, Numexpr, Scikit

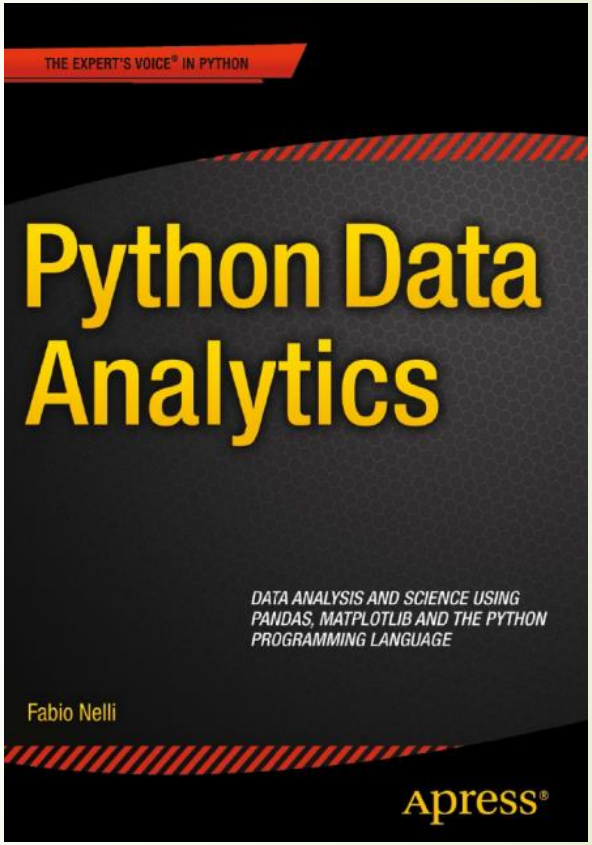
What's After This?

Advanced Data Analysis: Machine Learning with Python

The Machine Learning Model Building Process
Exploratory Data Analysis
Encoding Labels
One-hot Encoding
Cut and Binning
Sparse Matrices
Scikit-learn Supervised Learning
Linear Regression
Scoring Models
Regression Metrics
R-squared and p-values
Plotting Residuals
Multiple Variable Linear Regressions
Feature Selection
Correlation Matrices
Naïve Bayes Classifications
Language Processing
Confusion Matrices
K-Nearest Neighbors
Image Classification
Logistic Regressions
Imputing Values
Recursive Feature Elimination
Cross-validation
Support Vector Machines
Facial Recognition
Decision Trees/Random Forests
Boosting and Bagging
Grid Searches
K-Means
Dimensionality Reduction with PCA
Feature Scaling
Pipelining
(Intro) to Neural Networks and Tensorflow

Recommended Sources

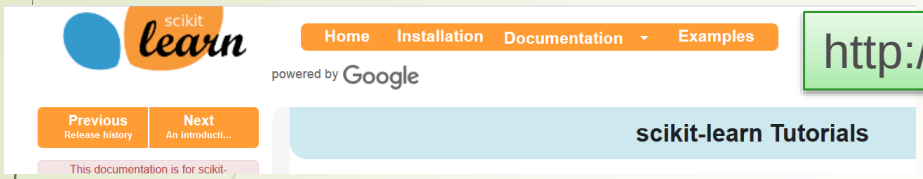




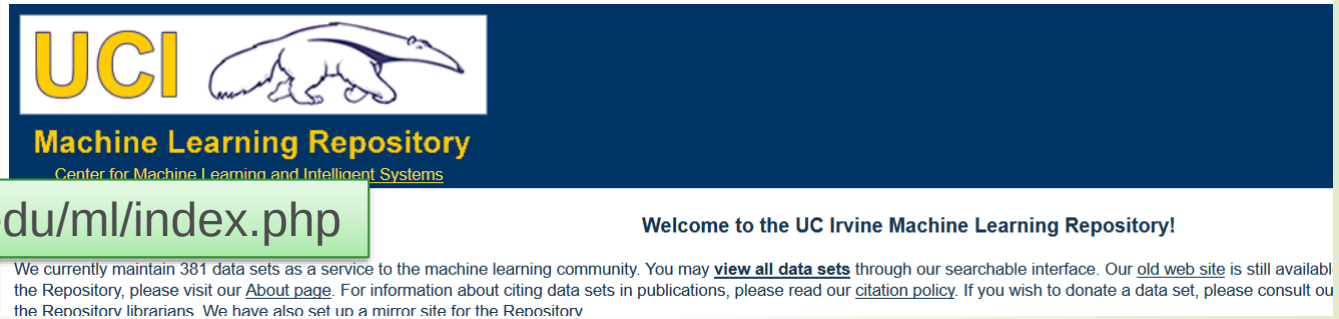
100 NumPy Exercises to keep you tuned:

http://www.labri.fr/perso/nrougier/teaching/numpy.100/

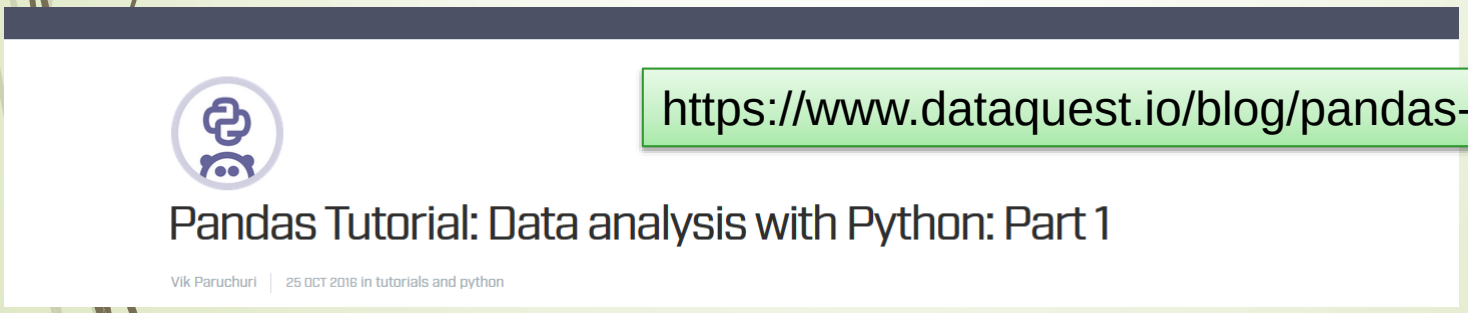
Other Resources



http://scikit-learn.org/stable/tutorial/index.html



http://archive.ics.uci.edu/ml/index.php



https://www.dataquest.io/blog/pandas-python-tutorial/

Evaluations

- Please take the time to fill out an evaluation
- All evaluations are read and considered

Thank you for your response and feedback which are critical to this process.

Questions