galvanize

gU

INTRODUCTION TO PYTHON

FOR DATA ANALYSIS

galvanize

About Galvanize

- **Accredited School: *GalvanizeU***
  - Fully accredited Master's Degree in Data Science
    - 12 Months program
    - (Internship included)
- **Un-accredited School**
  - Data Science Immersive (12 weeks)
  - Full Stuck (6 months)
  - Data Engineering (12/w, coming up)

About Me

- **Nir Kaldero**
- *Director of Science, Head of Galvanize Experts (gX)*
  - Faculty at gU / Appointed Professor at UNH
- **Email:** nir@galvanize.com
- **Twitter:** @NirKaldero

galvanize

AGENDA

- Why python?
- IPython-notebook project
- Getting started with Python
- Common Data Science/Analysis Libraries
- Tricks
- Where to go Next?

WHAT TO EXPECT?

My Side:

- Quick overview (Python / DS)
- Introduction
- Some advanced stuff to keep learning at home

Your Side:

- Get outside of your comfort zone
- Be patient (it takes time!)
- Practice at home and be in touch if you have questions

INTRODUCTION

WHY PYTHON?

- **Scalability**
- **Growing** [Data] Analytics Libraries & **Community**:
  - SciPy.org (for mathematics, science, and engineering)
  - StatsModels (Statistics)
  - Pandas (Frameworks)
  - SciKit-Learn (Machine Learning)
- **Graphics**
  - Libraries (ggplot, matplotlib)
  - APIs – Plot.ly
- **Easy to Learn and Code**

Q: What is Python?

Q: What is Python?

A: An **open source**, high-level, dynamic scripting language.

- **open source:** free! (both binaries and source files)

Q: What is Python?

A: An open source, **high-level**, dynamic scripting language.

- open source: free! (both binaries and source files)
- **high-level:** interpreted (add(a,b)) ; (not compiled '+')

Q: What is Python?

A: An open source, high-level, **dynamic** scripting language.

- open source: free! (both binaries and source files)
- high-level: interpreted (not compiled)
- **dynamic:** things that would typically happen at compile time happen at runtime instead (eg, *dynamic typing*)

DYNAMIC TYPING

```
>>> x = 1
>>> x
1
>>> x = 'horse'
>>> x
'horse'
```

Q: What is Python?

A: An open source, high-level, dynamic **scripting language**.

- open source: free! (both binaries and source files)
- high-level: interpreted (not compiled)
- dynamic: things that would typically happen at compile time happen at runtime instead (eg, *dynamic typing*)
- **scripting language:** “middleweight” (run one-by-one)

Q: Why Python?

- Command Line Interface (CLI) – for even quicker prototyping
  - Ipython notebook (great tool!)
- Straight- “sugar-~~free~~ lite” syntax
- Multiple programming paradigms
- Large set of available libraries
- Wide-use means extensive community support – stackoverflow, et al.

INTRO TO PYTHON

Python is an open source project which is maintained by a large and *very active community*.

It was originally created by Guido Van Rossum in the 1990s, who currently holds the title of Benevolent Dictator For Life (BDFL).

INTRO TO PYTHON

The presence of a BDFL means that Python has a *unified design philosophy*.

This design philosophy emphasizes *readability* and *ease of use*, and is codified in PEP8 (the Python style guide) and PEP20 (the Zen of Python).

NOTE: PEPs* are the public design specs that the language follows.

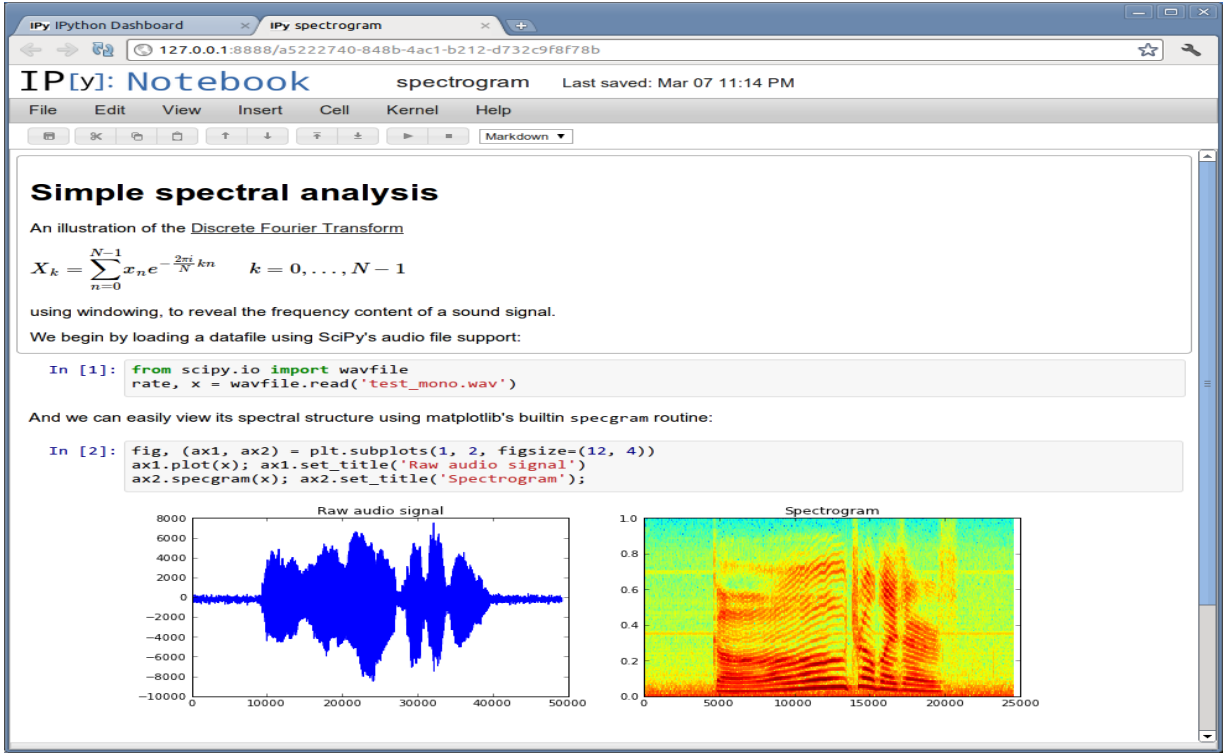
*Python Enhancement Proposals

PYTHON, HOW?

- Anaconda Project – Package
  - http://continuum.io/downloads
  - Completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing
  - 195+ of the most popular Python packages for science, math, engineering, data analysis
  - Completely free - including for commercial use and even redistribution

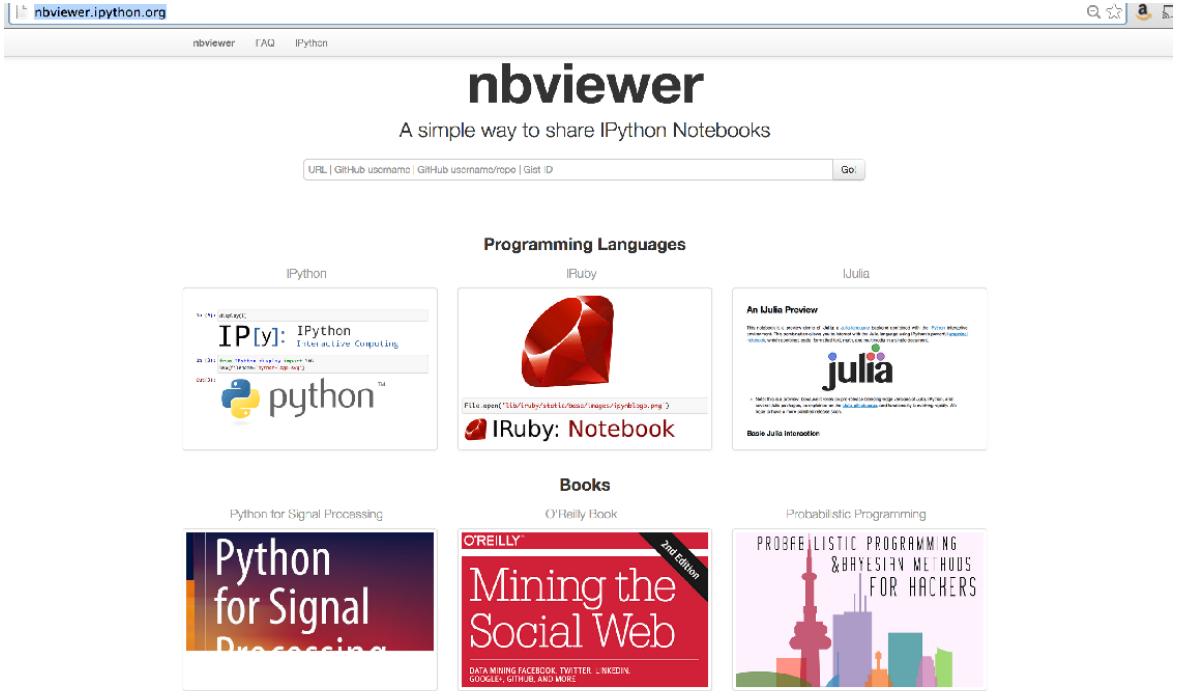
IPYTHON NOTEBOOK

- Powerful feature/tool
  - http://ipython.org/



VIEWER

- http://nbviewer.ipython.org/



II. PYTHON STRENGTHS & WEAKNESSES

STRENGTHS & WEAKNESSES

Python's popularity comes from the *strength of its design.*

The syntax looks like pseudocode, and *it is explicitly meant to be clear, compact, and easy to read.*

This is usually summarized by saying Python is an expressive language.

INTRO TO PYTHON

Python supports multiple programming paradigms, such as:

- Imperative programming
- Object oriented programming (OOP)
- Functional programming (really function-esqe)

STRENGTHS & WEAKNESSES

Ultimately, Python's most important strength is that it's easy to learn and easy to use.

Because there should be only one way to perform a given task, things frequently work the way you expect them to.

- paraphrased from PEP20 ("The Zen of Python")

Takeaway: This is a huge luxury!

http://www.python.org/dev/peps/pep-0020/

Q: Python sounds amazing. What is it bad at?

For one thing, Python is slower than a lower-level language (but keep in mind that this is a conscious tradeoff).

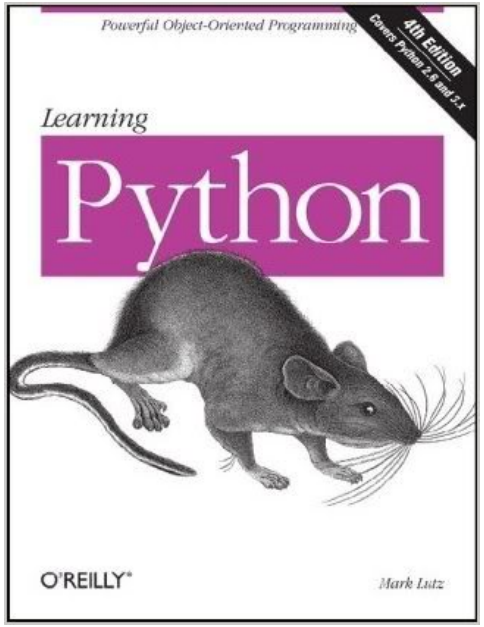
There are some other subtleties regarding dynamic typing that people occasionally dislike, but again this is intentional (and a matter of opinion).

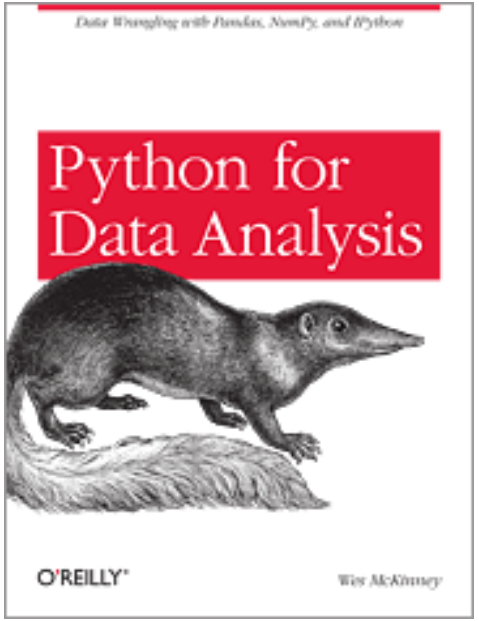
OTHER TOOLS?

- Julia (statistics)
- Java
- JavaScript
- R (Statistics)
- Matlab (Stats, costly)
- Stata (Stats, costly)
- C , etc,.

WHY PYTHON

BOOKS?





(PYTHON IN) THE DATA SCIENCE WORKFLOW

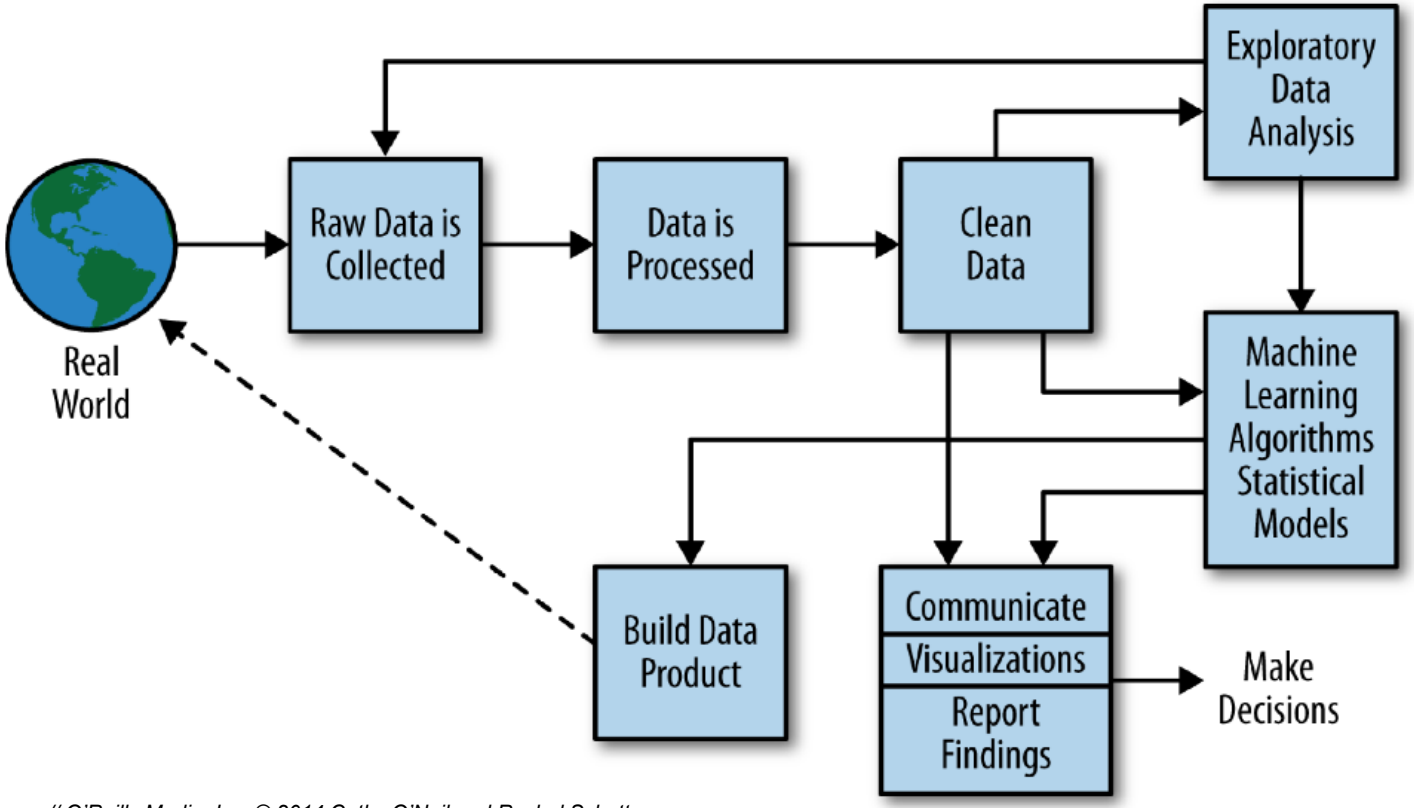
What is data science? Or Data Exploration?

The extraction of useful information and knowledge from large volumes of data, in order to improve decision-making
… or to tell a compelling story.

- Python is a useful tool for that purpose

Data Science for Business // O'Reilly Media, Inc © 2013 Foster Provost and Tom Fawcett

PYTHON AS A TOOL FOR DATA SCIENCE



Doing Data Science // O'Reilly Media, Inc. © 2014 Cathy O'Neil and Rachel Schutt.

DATA PREPARATION

- The goal of “Pre-processing” is to convert data into *a standard format*
- A *standard format* allows for input to algorithms to be standardized
- Some algorithms require inputs to be particularly formatted

Relevant Libraries:

- **NumPy** (working with Numbers/Math)
- **Pandas** (DataFrames)

ANALYSIS/MODEL

Try different algorithms to determine the optimal choice

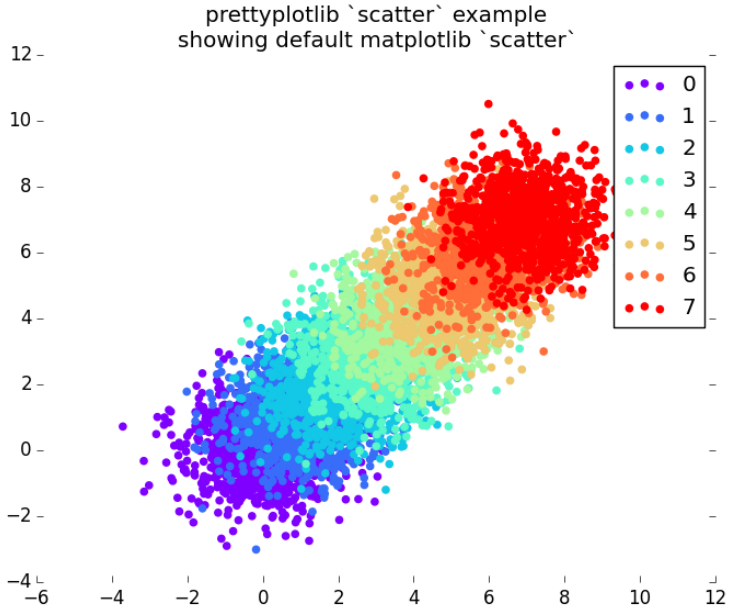
Relevant Libraries:

- SciPy
- scikit-learn
- StatsModels

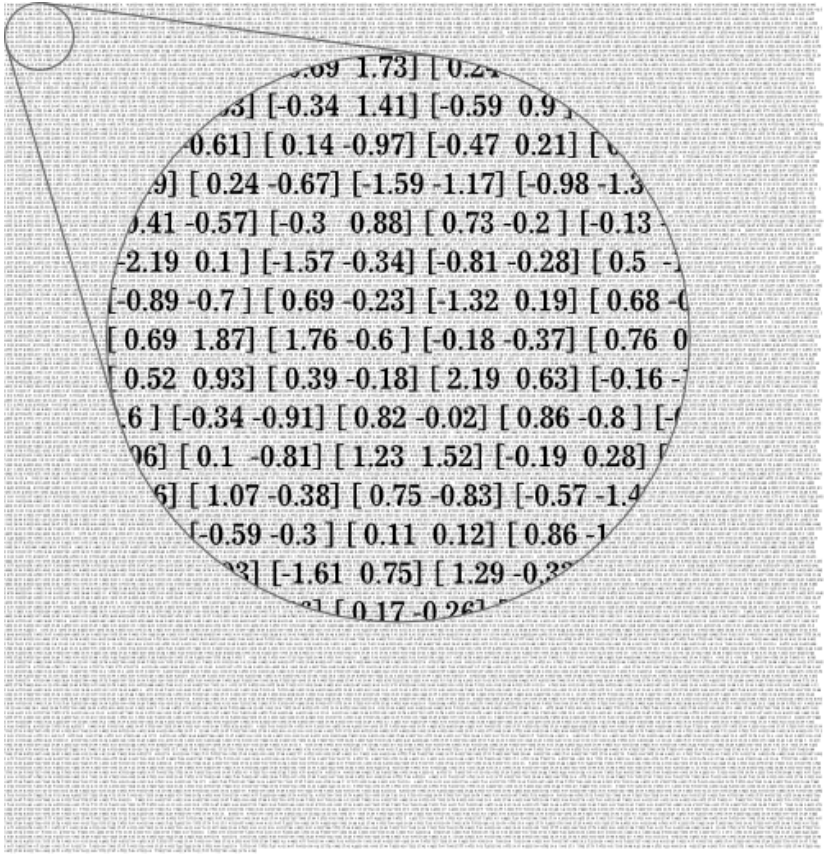
- “Black Box”

VISUALIZATION

Since seeing is believing…



Relevant Library: matplotlib



http://bit.ly/1gCXPtP // http://http://blog.olgabotvinnik.com

galvanize

USEFUL LIBRARIES

(ANALYSIS, MODELING, & VISUALIZATION)

DATA SCIENCE LIBRARY INTRODUCTION

NumPy	“Fundamental package for scientific computing”
Matplotlib	Plotting (& histograms, power spectra, bar charts, error charts, scatterplots, etc)
SciPy	“Fundamental library for scientific computing”
Pandas	Python Data Analysis (Data Frameworks)
Scikit	Application domain toolkits (Machine Learning)
StatsModels	Statistical Applications and Optimization Methods

NumPy

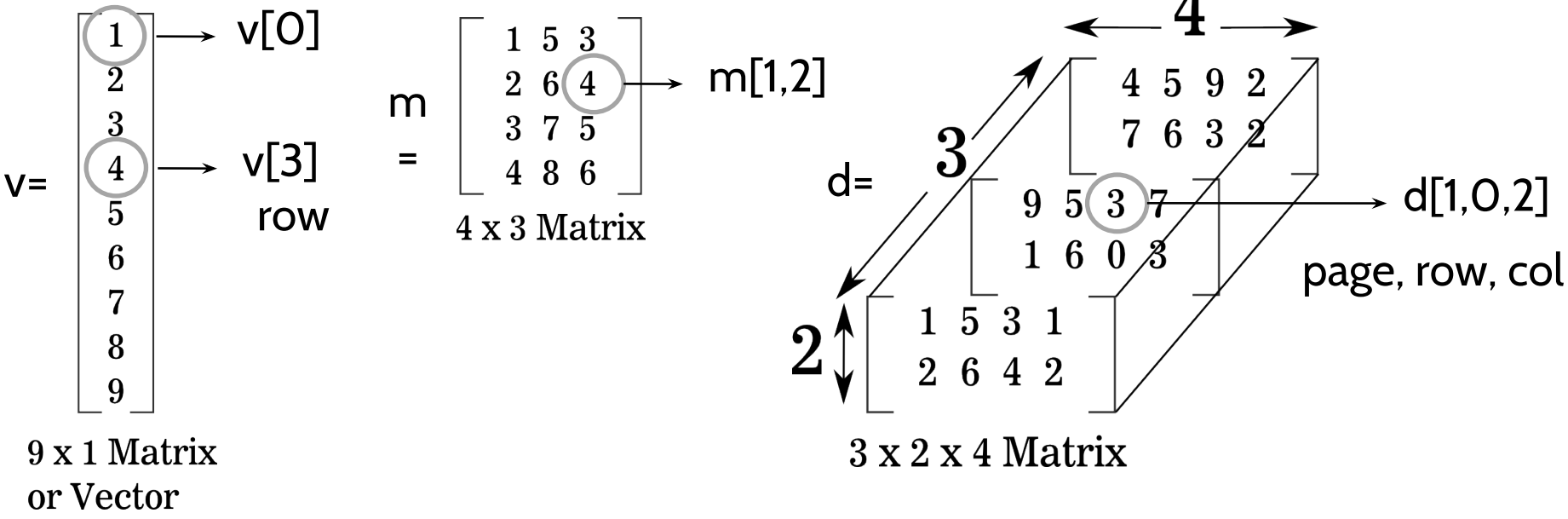
"add[s] support for large, multidimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays"

- Wikipedia

http://en.wikipedia.org/wiki/NumPy

DATA SCIENCE LIBRARY: NUMPY

Vectors, arrays, and matrices



BASIC DATA STRUCTURES

The most basic data structure is the None type.
This is the equivalent of NULL in other languages.

There are four basic numeric types:

1. int ($< 2^{63}$) / long ($\geq 2^{63}$)*
   * on 64-bit OS X/Linux, sys.maxint = 2**63-1
2. float (a “decimal”)
3. bool (True/False) or (1/0)
4. complex (“imaginary”)

```
>>> type(None)
<type 'NoneType'>
>>> type(1)
<type 'int'>
>>> type(2.5)
<type 'float'>
>>> type(True)
<type 'bool'>
>>> type(2+3j)
<type 'complex'>
```

http://docs.python.org/2/library/stdtypes.html#numeric-types-int-float-long-complex

BASIC DATA STRUCTURES

The next basic data type is the array, implemented in Python as a **list**.

A list is a (zero-based numbered), *ordered* collection of elements, and these elements can be of arbitrary type.

Lists are mutable, meaning they can be changed in-place.

```
>>> a = [1,'b',True]
>>> a[2]
True
>>> a[1]='aa'
>>> a
[1,'aa',True]
```

BASIC DATA STRUCTURES

Tuples: immutable arrays of arbitrary elements.

```
>>> x = (1,'a',2.5)
>>> x[0]
1
>>> x[0]='b' #trying to change a value
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a,b = (1,2)
>>> a
1
```

Tuples are frequently used behind the scenes in a special type of variable assignment called tuple packing/unpacking.

http://docs.python.org/2/library/stdtypes.html#numeric-types-int-float-long-complex

BASIC DATA STRUCTURES

The **string** type in Python represents an *immutable ordered* array of characters (note there is no char type).

Strings support *slicing* and *indexing* operations like arrays, and have many other string-specific functions as well.

String processing is one area where Python excels.

http://docs.python.org/2/library/stdtypes.html

BASIC DATA STRUCTURES

Associative arrays (or hash tables) are implemented in Python as the **dictionary** type. This is a very efficient and useful structure that Python's internal functions use extensively.

```
>>> this_class={'subject':'python for data science',
... 'instructor':'drew','time':'120', 'is_cool': True}
>>> this_class['subject']
'python for data science'
>>> this_class['is_cool']
True
```

Dictionaries are unordered collections of key-value pairs, and *dictionary keys must be immutable*.

http://docs.python.org/2/tutorial/datastructures.html#dictionaries

BASIC DATA STRUCTURES

Sets are *unordered mutable* collections of distinct elements.

```
>>> y =set([1,1,2,3,5,8])
>>> y
set([8, 1, 2, 3, 5])
```

These are particularly useful for *checking membership* of an element and for ensuring element *uniqueness*.

http://docs.python.org/2/library/sets.html

BASIC DATA STRUCTURES

Our final example of a *"data type" is the Python file object.* This example represents an open connection to a file on your laptop.

```
>>> with open('output_file.txt','w') as f:
...     f.write('test')
```

These are particularly easy to use in Python, especially using the with statement *context manager*, which automatically closes the file handle when it goes out of scope.

http://docs.python.org/2/library/stdtypes.html#file-objects

PYTHON CONTROL FLOW

CONTROL FLOW

Python has a number of control flow tools that will be familiar from other languages. The first is the **if-else** statement, whose compound syntax looks like this:

```
>>> x, y = False, False
>>> if x :
...     print 'apple'
... elif y :
...     print 'orange'
... else :
...     print 'sandwich'
...
sandwich
```

A **while loop** executes while a given condition evaluates to True.

```
>>> x = 0
>>> while True :
...     print 'HELLO!'
...     x += 1
...     if x >= 3 :
...         break
...
HELLO!
HELLO!
HELLO!
```

CONTROL FLOW

The familiar (& useful) **for loop** construct executes a block of code for a range of values.

```
>>> for k in range(4) :
...     print k**2
...
0
1
4
9
```

The object that a for loop iterates over is called (appropriately) an *iterable*.

CONTROL FLOW

A useful but possibly unfamiliar construct is the **try-except block**:

```
>>> try:
...     print undefined_variable
... except :
...     print 'An Exception has been caught'
...
An Exception has been caught
```

This is useful for catching and dealing with errors, also called exception handling.

FUNCTIONS

Python allows you to define **custom functions**:

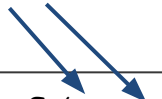
```
>>> def x_minus_3(x) :
...     return x - 3
...
>>> x_minus_3(12)
9
```

NOTE: Functions can *optionally* return a value with a return statement (as this example does).

FUNCTIONS

Functions can take a number of arguments as inputs, and these arguments can be provided in two ways:

1) **as positional arguments:**



```
>>> def f(x,y) :
...     return x - y
...
>>> f(4,2)
2
>>> f(2,4)
-2
```

FUNCTIONS

Functions can take a number of arguments as inputs, and these arguments can be provided in two ways:

2) **as keyword arguments** (this example with defaults):

```
>>> def g(arg1=10, arg2=20) :
...     return arg1 / float(arg2)
...
>>> g(arg2=100)
0.1
>>> g(1,20)
0.05
>>> g()
0.5
```

IMPORT

As introduced on the last slide, the **import** statement avails library objects/functions:

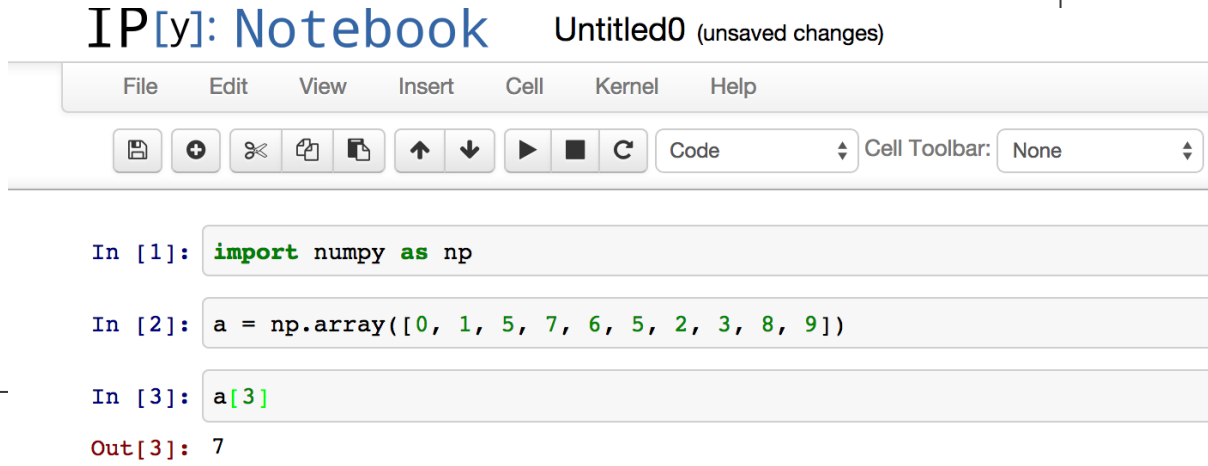
```
>>> import math
>>> math.pi
3.141592653589793
>>> from math import sin
>>> sin(math.pi/2)
1.0
>>> from math import *
>>> print e, log10(1000), cos(pi)
2.71828182846 3.0 -1.0
```

The three methods differ with respect to the interaction with the local namespace.

DATA SCIENCE LIBRARY: NUMPY

ndarray object creation, indexing and slicing (1-dimensional)

```
>>> import numpy as np
>>> a = np.array([0, 1, 5, 7, 6, 5, 2, 3, 8, 9])
>>> a[3]
7
>>> a[3:7]
array([7, 6, 5, 2])
>>> b=np.array([1,5,7])
>>> a[[b]]
array([1, 5, 3])
>>> a[a > 5]
array([7, 6, 8, 9])
```



DATA SCIENCE LIBRARY: NUMPY CREATING ARRAYS

More methods to quickly create ndarray objects (1D) and (2D)

```
>>> b = np.arange(1, 20, 3)
>>> b
array([ 1,  4,  7, 10, 13, 16, 19])
>>> a = np.ones((3, 3))
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
```

MATRIX MULTIPLICATION (REMINDER)

RULE:
Two matrices can be multiplied only when the number of columns in the first equals the number of rows in the second[1]

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \bullet \begin{bmatrix} 6 \\ 7 \end{bmatrix} = \begin{bmatrix} 34 \\ 47 \\ 60 \end{bmatrix}$$

$$3 \times 2 \quad = \quad 2 \times 1 \qquad\qquad 3 \times 1$$

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \bullet \begin{bmatrix} 6 \\ 7 \end{bmatrix} \rightarrow \begin{bmatrix} 1x6 + 4x7 \\ 2x6 + 5x7 \\ 3x6 + 6x7 \end{bmatrix} = \begin{bmatrix} 34 \\ 47 \\ 60 \end{bmatrix}$$

a b

numpy.dat(a,b)

[1] *http://en.wikipedia.org/wiki/Matrix_(mathematics)*

DATA SCIENCE LIBRARY: NUMPY 2-D MATRICES

2-d ndarray object (“a matrix”) can be defined; and operations applied

```
>>> a=np.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.shape
(2, 3)
>>> a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.shape
(3, 2)
>>> b=np.array([6, 7])
>>> np.dot(a.T,b)  # matrix multiplication
array([34, 47, 60])
```

DATA SCIENCE LIBRARY: NUMPY

ndarray operations

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b.sum(axis=0) #çolum
array([12, 15, 18, 21])
>>> b.sum(axis=1) #row
array([ 6, 22, 38])
>>> b.sum()
66
>>> b.min(axis=1)
array([0, 4, 8])
>>> b.max(axis=0)
array([ 8,  9, 10, 11])
```

DATA SCIENCE LIBRARY: NUMPY

Matric object and 2-d matrix indexing and slicing

```
>>> b=np.mat('1 2 3 4; 5 6 7 8; 9 10 11 12')
>>> b
matrix([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
>>> b[:,:3] #rows ; Col
matrix([[ 1,  2,  3],
        [ 5,  6,  7],
        [ 9, 10, 11]])
```

DATA SCIENCE LIBRARY: NUMPY

Matrix object – allows matrix arithmetic using operators

```
>>> b=np.mat('1 2 3 4; 5 6 7 8; 9 10 11 12')
>>> b
matrix([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
>>> a=np.mat('1;2;3;4')
>>> b*a
matrix([[ 30],
        [ 70],
        [110]])
```

DATA SCIENCE LIBRARY: NUMPY

n-dimension ndarray object (“matrices”) can also be created

```
>>> aa=np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
>>> aa
array([[[1, 2, 3],
        [4, 5, 6]],
       [[1, 2, 3],
        [4, 5, 6]]])
>>> bb=np.array([[[3],[4],[6]],[[6],[5],[7]]])
>>> aa.shape, bb.shape
((2, 2, 3), (2, 3, 1)) # page, row, col → 3D
>>> np.dot(aa,bb)
array([[[29],
         [37]],
        [[68],
         [91]]],
       [[[29],
         [37]],
        [[68],
         [91]]]])
```

DATA SCIENCE LIBRARY: NUMPY

ndarray element-wise operations – scalar and matrix

```
>>> aa = np.arange(5)#interval of 5 elements, start → 0
>>> aa
array([0, 1, 2, 3, 4])
>>> aa * 5
array([ 0,  5, 10, 15, 20])
>>> bb = np.array([2, 4, 6, 8, 10])
>>> np.multiply(aa, bb)
array([ 0, 4, 12, 24, 40])
>>> np.divide(aa, bb.astype(float)) #just to make sure
array([ 0.        ,  0.25      ,  0.33333333,  0.375     ,  0.4       ])
```

DATA SCIENCE LIBRARY: NUMPY

Matrix math

```
>>> a=np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]]
>>> b=np.array([[1],[2]])
>>> b
array([[1],
       [2]])
>>> a.shape, b.shape
((2, 2), (2, 1))
>>> np.dot(a,b)
array([[ 5],
       [11]])
>>> np.dot(b,a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
```

PYTHON CONT.

DATA SCIENCE LIBRARY: PANDAS

Pandas

> "… an open source… library providing high-performance… data structures and data analysis tools"
>
> - pandas website

http://pandas.pydata.org/

DATA SCIENCE LIBRARY: PANDAS

Series - one-dimensional labeled array

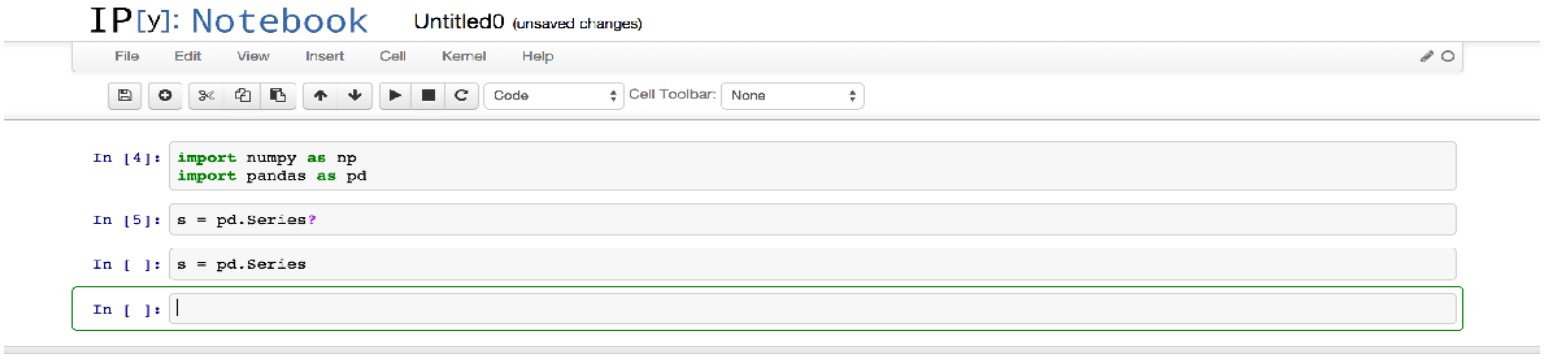
```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(3), index=['a', 'b', 'c’])
>>> s
a   -0.245247
b   -1.162124
c   -0.698275
dtype: float64
>>> s['a']
-0.24524714260468519
>>> s[0]
-0.24524714260468519
>>> d = {'a' : 0, 'b' : 1, 'c' : 2}
>>> pd.Series(d)
a    0
b    1
c    2
dtype: int64
```

1) Series is indexed by a series of *labels*
2) Series can be be *addressed or sliced* like an array
3) Series are somewhat akin to *dictionaries* and it is easy to convert a dictionary to a Series

DATA SCIENCE LIBRARY: PANDAS

"TRICK"/HELP(!)

```
>>> import pandas as pd, numpy as np
>>> s = pd.Series(np.random.randn(3), index=['a', 'b', 'c'])
```




```
Type:            type
String form:     <class 'pandas.core.series.Series'>
File:            /Users/nirkaldero/anaconda/lib/python2.7/site-packages/pandas/core/series.py
Init definition: pd.Series(self, data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
Docstring:
One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be any hashable type. The object
supports both integer- and label-based indexing and provides a host of
methods for performing operations involving the index. Statistical
methods from ndarray have been overridden to automatically exclude
missing data (currently represented as NaN)

Operations between Series (+, -, /, *, **) align values based on their
associated index values-- they need not be the same length. The result
index will be the sorted union of the two indexes.
```

IP[y]: Notebook Untitled0 (unsaved changes)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

```
In [4]: import numpy as np
        import pandas as pd

In [5]: s = pd.Series?

In [ ]: s = pd.Series

In [ ]:
```

```
Type:            type
String form:     <class 'pandas.core.series.Series'>
File:            /Users/nirkaldero/anaconda/lib/python2.7/site-packages/pandas/core/series.py
Init definition: pd.Series(self, data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
Docstring:
One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be any hashable type. The object
supports both integer- and label-based indexing and provides a host of
methods for performing operations involving the index. Statistical
methods from ndarray have been overridden to automatically exclude
missing data (currently represented as NaN)

Operations between Series (+, -, /, *, **) align values based on their
associated index values-- they need not be the same length. The result
index will be the sorted union of the two indexes.

Parameters
----------
data : array-like, dict, or scalar value
    Contains data stored in Series
index : array-like or Index (1d)
    Values must be unique and hashable, same length as data. Index
```

DATA SCIENCE LIBRARY: PANDAS

Series – vector operation support and index alignment

```
>>> s = pd.Series(np.arange(4), index=['a', 'b', 'c', 'd'])#
>>> s + s
a    0
b    2
c    4
d    6
dtype: int64
>>> t = pd.Series([30,40], index=['b', 'c' ])
>>> t
b    30
c    40
dtype: int64
>>> s + t
a   NaN
b    31
c    42
d   NaN
```

DATA SCIENCE LIBRARY: PANDAS

DataFrame – 2-d labeled data structure

```
>>> d = {'one' : [10., 20., 30., 40.], 'two' : [4., 3., 2., 1.]}
>>> pd.DataFrame(d, index=['a','b','c','d'])
   one  two
a   10    4
b   20    3
c   30    2
d   40    1
>>> dd={'0':pd.Series([1,2],index=['a','b']),
...     '1':pd.Series([15,25,35],index=['a','b','c'])}
>>> pd.DataFrame(dd)
    0   1
a   1  15
b   2  25
c NaN  35
```

DATA SCIENCE LIBRARY: PANDAS

DataFrame – data alignment and arithmetic operations

```
>>> df = pd.DataFrame(np.floor(np.random.randn(3,4)*10), columns=['A', 'B', 'C','D'])
>>> df #np.random.randn(3,4) → (row, column) ;  #floor - "floor toward zero"
   A   B   C   D
0  4  -1  12  18
1  8  12  -7  11
2  2  13   6  21
>>> df2 = pd.DataFrame(np.floor(np.random.randn(3,2)*10), columns=['B', 'C'])
>>> df2
    B   C
0  -2 -10
1  11   9
2   4   2
>>> df + df2
    A   B  C   D
0 NaN  -3  2 NaN
1 NaN  23  2 NaN
2 NaN  17  8 NaN
```

DATA SCIENCE LIBRARY: PANDAS

Panel – 3-d labeled data structure

```
>>> panel = pd.Panel(np.random.randn(5,3,2).round(decimals=1),
...                  items=['one', 'two', 'three','four','five'],
...                  major_axis=pd.date_range('1/1/2000', periods=3),
...                  minor_axis=['a', 'b'])
>>> panel.to_frame()#(np.random.randn(5,3,2) →  (start,end,steps)
                  one  two  three  four  five
major      minor
2000-01-01 a     -1.4 -0.1   -0.1  -0.9   2.9
           b     -0.1 -0.1    1.5   0.1   0.5
2000-01-02 a      0.9  0.3   -0.2  -1.1  -0.8
           b     -0.0 -0.6    0.0  -0.0   1.4
2000-01-03 a      0.1  2.0    0.2   2.4  -1.2
           b     -0.9 -1.8    1.0  -1.4  -0.9
```

LIBRARY: PANDAS

Some additional notable features:

- Data loading (flat files, Excel, MySQL)
- Data selection using indexes
- Group by (columns or indexes)
- Joins
- NumPy and custom functions vectorized via the .apply() method

CREATING A DATAFRAME

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'int_col' : [1,2,6,8,-1],
                   'float_col' : [0.1, 0.2,0.2,10.1,None],
                   'str_col' : ['a','b',None,'c','a']})

df
```

	float_col	int_col	str_col
0	0.1	1	a
1	0.2	2	b
2	0.2	6	None
3	10.1	8	c
4	NaN	-1	a

INDEXING

```
df.ix[:,['float_col','int_col']] #[rows, col]

   float_col  int_col
0        0.1        1
1        0.2        2
2        0.2        6
3       10.1        8
4        NaN       -1


df[['float_col','int_col']] #another way..

   float_col  int_col
0        0.1        1
1        0.2        2
2        0.2        6
3       10.1        8
4        NaN       -1
```

CONDITIONAL INDEXING

```
df['float_col'] > 0.15
0    False
1     True
2     True
3     True
4    False
Name: float_col, dtype: bool

df[df['float_col'] > 0.15]

float_col  int_col str_col
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
```

```
df

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

(MORE) CONDITIONAL INDEXING

```
df[(df['float_col'] > 0.1) & (df['int_col']>2)] #and

   float_col  int_col str_col
2        0.2        6    None
3       10.1        8       c

df[(df['float_col'] > 0.1) | (df['int_col']>2)] #or

   float_col  int_col str_col
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c

df[~(df['float_col'] > 0.1)] #invert

   float_col  int_col str_col
0        0.1        1       a
4        NaN       -1       a
```

COLUMN RENAMING

```
df2 = df.rename(columns={'int_col' : 'some_other_name'})

# this is a copy of the DataFrame
df2

   float_col  some_other_name str_col
0        0.1                1       a
1        0.2                2       b
2        0.2                6    None
3       10.1                8       c
4        NaN               -1       a

# this will modify the DataFrame in place.
df2.rename(columns={'some_other_name' : 'int_col'}, inplace = True)
```

```
df

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

REMOVING MISSING VALUES

```
df2

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a

df2.dropna()

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
3       10.1        8       c
```

REPLACE MISSING VALUES

```
df3 = df.copy()
df3
```

```
   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

```
mean = df3['float_col'].mean()
df3['float_col'].fillna(mean)
```

```
0     0.10
1     0.20
2     0.20
3    10.10
4     2.65
```

MAP & APPLY FUNCTIONS

```
# .map() works on a Series (in this case an extracted column of a DF)
df['str_col'].dropna().map(lambda x : 'map_' + x)

0    map_a
1    map_b
3    map_c
4    map_a
Name: str_col

# .apply() works a DataFrame
df[['int_col','float_col']].apply(np.sqrt)

    int_col  float_col
0  1.000000   0.316228
1  1.414214   0.447214
2  2.449490   0.447214
3  2.828427   3.178050
4       NaN        NaN
```

```
df

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

APPLYMAP

```
def this_fn(x):
    if type(x) is str:
      return 'applymap_' + x
    elif x:
      return 100 * x
    else:
      return

df.applymap(this_fn)
```

	float_col	int_col	str_col
0	10	100	applymap_a
1	20	200	applymap_b
2	20	600	None
3	1010	800	applymap_c
4	NaN	-100	applymap_a

```
df
```

	float_col	int_col	str_col
0	0.1	1	a
1	0.2	2	b
2	0.2	6	None
3	10.1	8	c
4	NaN	-1	a

VECTORIZED MATH OPERATIONS

```
df = pd.DataFrame(data={"A":[1,2], "B":[1.2,1.3]})

df["C"] = df["A"]+df["B”]
df
```

```
   A    B    C
0  1  1.2  2.2
1  2  1.3  3.3
```

```
df["D"] = df["A"]*3
df
```

```
   A    B    C  D
0  1  1.2  2.2  3
1  2  1.3  3.3  6
```

```
df["E"] = np.sqrt(df["A"])
df
```

```
   A    B    C  D         E
0  1  1.2  2.2  3  1.000000
1  2  1.3  3.3  6  1.414214
```

VECTORIZED STRING OPERATIONS

```
df = pd.DataFrame(data={"A":[1,2],
                        "B":[1.2,1.3],
                        "Z":["apple","pear"]})

df
```

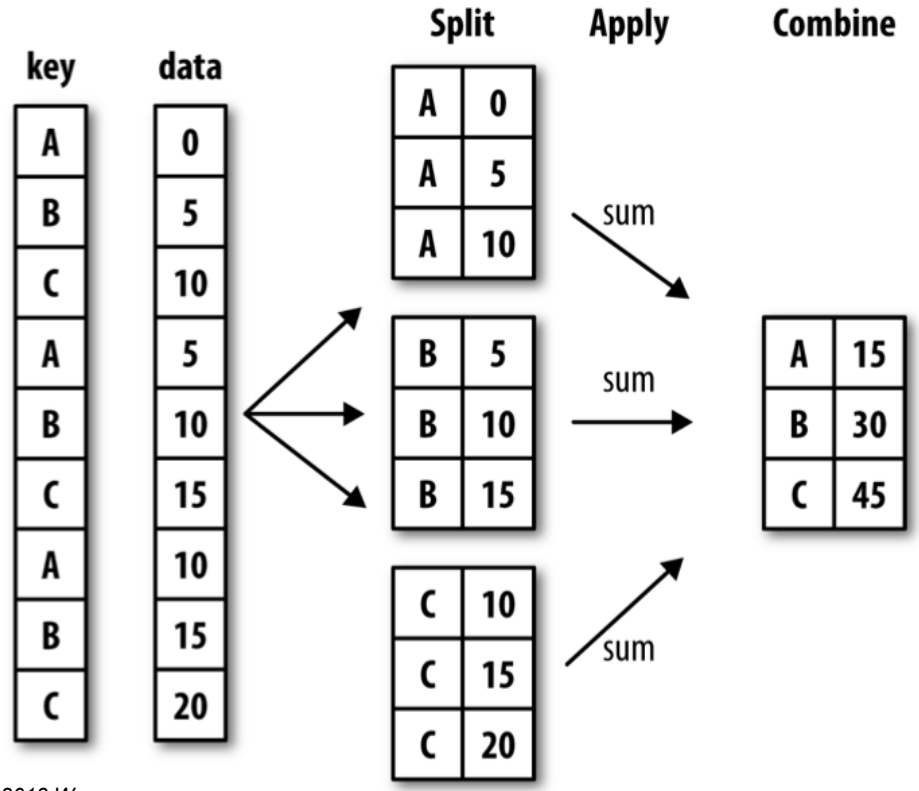
```
 A    B      Z
0  1  1.2  apple
1  2  1.3   pear
```

```
df["F"] = df.Z.str.upper()

df
```

```
   A    B      Z      F
0  1  1.2  apple  APPLE
1  2  1.3   pear   PEAR
```

GROUPBY



Python for Data Analysis // O'Reilly Media, Inc © 2013 Wes McKinney.

GROUPBY

```
grouped = df4['float_col'].groupby(df4['str_col'])
grouped.mean()

str_col
a            0.10
b            0.35
c           10.10

g2=df4.groupby(['float_col','str_col'])
g2.sum()

                    int_col
float_col str_col
0.1       a               0
0.2       b               2
0.5       b               6
10.1      c               8
```

```
df4

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.5        6       b
3       10.1        8       c
4        0.1       -1       a
```

NEW COLUMNS FROM EXISTING COLUMNS

```
df4['new_col']=df4['int_col']*10
df4
float_col  int_col str_col  new_col
0        0.1        1       a       10
1        0.2        2       b       20
2        0.5        6       b       60
3       10.1        8       c       80
4        0.1       -1       a      -10

df4['other_new_col']=df4['int_col']+103

df4
   float_col  int_col str_col  new_col  other_new_col
0        0.1        1       a       10            104
1        0.2        2       b       20            105
2        0.5        6       b       60            109
3       10.1        8       c       80            111
4        0.1       -1       a      -10            102
```

```
df4

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.5        6       b
3       10.1        8       c
4        0.1       -1       a
```

NEW COLUMNS FROM EXISTING COLUMNS

```
def sum_two_cols(series):
    return series['int_col'] + series['float_col']


df5 = df.copy()
df5['sum_col'] = df5.apply(sum_two_cols,axis=1)

df5
   float_col  int_col str_col  sum_col
0        0.1        1       a      1.1
1        0.2        2       b      2.2
2        0.2        6    None      6.2
3       10.1        8       c     18.1
4        0.1       -1       a     -0.9
```

```
df5

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

STATISTICS

```
df.describe()

       float_col   int_col
count    4.00000  5.000000
mean     2.65000  3.200000
std      4.96689  3.701351
min      0.10000 -1.000000
25%      0.17500  1.000000
50%      0.20000  2.000000
75%      2.67500  6.000000
max     10.10000  8.000000
```

```
df

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

MORE STATISTICS

```
# The cov method provides the covariance between suitable columns.
df.cov()

           float_col  int_col
float_col     19.803    12.04
int_col       12.040    13.70

# The corr method provides the correlation between suitable columns.

df.corr()

           float_col   int_col
float_col   1.000000  0.760678
int_col     0.760678  1.000000
```

```
df

   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

MERGE AND JOIN

```
other = pd.DataFrame({'str_col' : ['a','b'], 'some_val' : [1, 2]})
other
```

```
   some_val str_col
0         1       a
1         2       b
```

```
pd.merge(df,other,on='str_col',how='inner')
```

```
   float_col  int_col str_col  some_val
0        0.1        1       a         1
1        NaN       -1       a         1
2        0.2        2       b         2
```

```
df
```

```
   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

BASIC PLOTTING : LINES

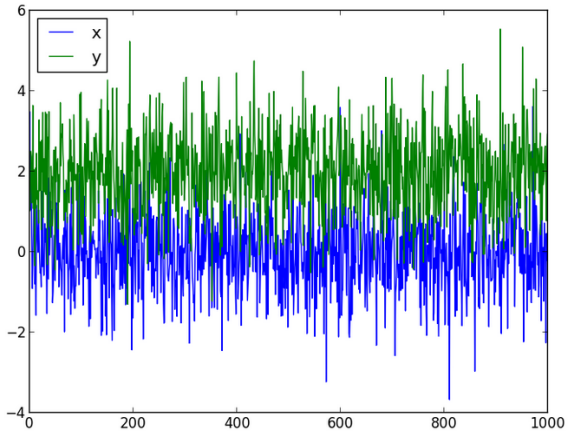
```
plot_df = pd.DataFrame(np.random.randn(1000,2),columns=['x','y'])

plot_df['y'] = plot_df['y'].map(lambda x : x + 1)

plot_df.head()
```

	x	y
0	0.609493	0.802296
1	-0.466751	2.943608
2	-0.614221	0.669065
3	-1.362734	1.766198
4	-0.890851	1.241256

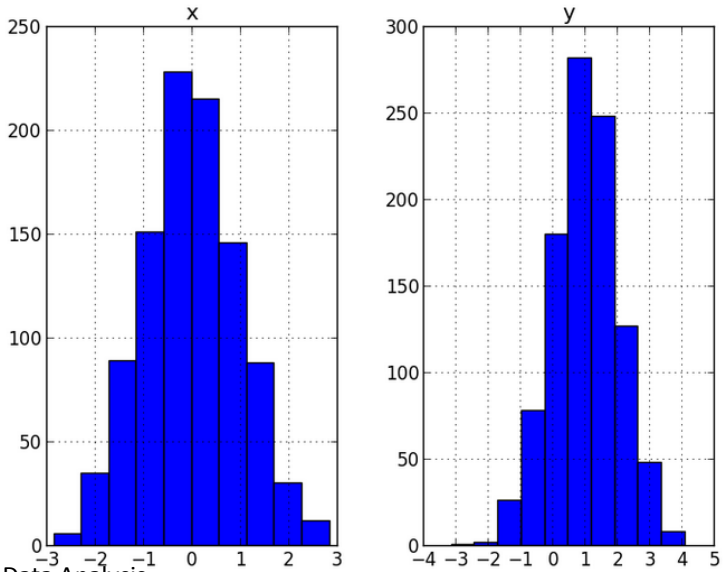
```
plot_df.plot()
<matplotlib.axes.AxesSubplot at 0x10e6ca
```



BASIC PLOTTING : HISTOGRAM

```
plot_df.hist()

array([[<matplotlib.axes.AxesSubplot object at 0x10aacae90>,
        <matplotlib.axes.AxesSubplot object at 0x10976d950>]],
dtype=object)
```



galvanize

RESOURCES & NEXT STEPS

RESOURCES FOR FURTHER EXPLORATION

Learn the command line interface (free HTML version)
http://cli.learncodethehardway.org

Take a Python tutorial (free HTML version)
http://learnpythonthehardway.org

Refresh your understanding of linear regression
https://www.khanacademy.org/math/probability/regression

RESOURCES FOR FURTHER EXPLORATION

Python Distribution (for Analytics) – Anaconda by Continuum Analytics
http://contimuum.io

On-line Python Integrated Development Environment (IDE)
http://wakari.io

Google for Education
https://developers.google.com/edu/python/set-up

Question:

Now that we know how to work with numbers and strings, let's write a program that might actually be useful!
Let's say you want to find out how much you weigh in stone.
A concise program can make short work of this task. Since a stone is 14 pounds, and there are about 2.2 pounds in a kilogram, the following formula should do the trick:

$$m_{stone} = \frac{m_{kg} \times 2.2}{14}$$

So, let's turn this formula into a program

Walkthrough

First step:

Ask the user: "What is your mass in kilograms?"

Hint: Use the function “input” to save the value.

Second Step: Make the calculation

Third Step: Print the results.

Question:

Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included).

The numbers obtained should be printed in a comma-separated sequence on a single line.

Hints:
Consider use range(#begin, #end) method

Question:

Write a program which can compute the factorial of a given numbers.

The results should be printed in a comma-separated sequence on a single line.

Suppose the following input is supplied to the program: 8

Then, the output should be: 40320

Hints:

In case of input data being supplied to the question, it should be assumed to be a console input.

CLASSES

Python supports **classes** with **member attributes** and **functions**:

```
>>> from math import pi
>>>
>>> class Circle() :
...     def __init__(self, r=1) :
...         self.radius = r
...     def area(self) :
...         return pi * (self.radius ** 2)
...
>>> c=Circle(4)
>>> c.radius
4
>>> c.area()
50.26548245743669
>>> 3.141592653589793 * 4 * 4
50.26548245743669
```