

# Language Fundamentals

Logit Academy

# DAY 1

# Schedule

	<b>Day 1</b>	<b>Day 2</b>	<b>Day 3</b>	<b>Day 4</b>	<b>Day 5</b>
Morning	Course Introduction	Introduction to Python	Matplotlib, Numpy	Statistics, Pandas	Introduction to SQL
<b>Break</b>					
Morning	Course Overview	Introduction to Python	Numpy	Pandas, Seaborn	SQLAlchemy
<b>Lunch</b>					
Afternoon	IPython, Slack	Exercises	Exercises	Exercises	Exercises
<b>Break</b>					
Afternoon	Introduction to Python	Exercises	Exercises	Exercises	Exercises

# What Is Python?

## ONE LINER

Python is an interpreted programming language that is easy to learn, easy to use, and comprehensive in terms of data science tools (data munging, stats, machine learning, NLP, etc.) and programming styles (from exploratory analysis to repeatable science to software engineering for production deployment).

## PYTHON HIGHLIGHTS

- Interpreted and interactive
- Automatic garbage collection
- Dynamic typing
- Object-oriented
- Free
- Portable / Cross-Platform
- Easy to Learn and Use
- “Batteries Included”

# Python data analytics platform and environment

# IPython

An enhanced  
interactive Python shell

## STANDARD PYTHON COMMANDS WITH NUMBERED In/Out

```
In [1]: a = 1
```

```
In [2]: a + 2
```

```
Out[2]: 3
```

## TIMING YOUR CODE

```
# One of the most useful features of IPython is built-in timing.  
# Use %timeit for one-line of code, and %%timeit for multiple lines.
```

```
In [3]: %%timeit
```

```
....: a = []  
....: for value in xrange(10):  
....:     a.append(value**2)  
....:
```

```
100000 loops, best of 3: 1.88 us per loop
```

```
In [4]: %timeit a = [value**2 for value in xrange(10)]  
1000000 loops, best of 3: 1.46 us per loop
```

# Function Info in IPython

## HELP USING ?

```
# Follow a command with '?' to print its documentation.
```

```
In [5]: len?
```

```
Type:          builtin_function_or_method
```

```
String form: <built-in function len>
```

```
Namespace:    Python builtin
```

```
Docstring:
```

```
len(object) -> integer
```

```
Return the number of items of a sequence or mapping.
```

# Function Info in IPython

## SHOW SOURCE CODE USING ??

```
In [6]: import numpy as np
```

```
In [7]: np.squeeze??
```

```
def squeeze(a):
    """Remove single-dimensional entries from the shape of a.
```

Examples

-----

```
>>> x = array([[1,1,1],[2,2,2],[3,3,3]])
```

```
>>> x.shape
```

```
(1, 3, 3)
```

```
>>> squeeze(x).shape
```

```
(3, 3)
```

"""

```
try:
```

```
    squeeze = a.squeeze
```

```
except AttributeError:
```

```
    return _wrapit(a, 'squeeze')
```

```
return squeeze()
```



?? can't show the source code for “extension” functions that are implemented in C.

# Directory Navigation in IPython

```
# Change directory (note Unix style forward slashes!)
In [8]: cd c:/python_class/Demos/speed_of_light
c:\python_class\Datasets\speed_of_light
```



Tab completion helps you find and type directory and file names.

```
# List directory contents (Unix style, not "dir").
```

```
In [9]: ls
```

```
Volume in drive C has no label.
Volume Serial Number is 5417-593D
Directory of c:\python_class\Datasets\speed_of_light
09/01/2008  02:53 PM    <DIR>        .
09/01/2008  02:53 PM    <DIR>        ..
09/01/2008  02:48 PM          1,188 exercise_speed_of_light.txt
09/01/2008  02:48 PM      2,682,023 measurement_description.pdf
09/01/2008  02:48 PM      187,087 newcomb_experiment.pdf
09/01/2008  02:48 PM          1,312 newcomb_histogram.dat
09/01/2008  02:48 PM          1,436 newcomb_histogram.py
09/01/2008  02:48 PM          1,232 newcomb_histogram2.py
                           6 File(s)      2,874,278 bytes
                           2 Dir(s) 11,997,437,952 bytes free
```

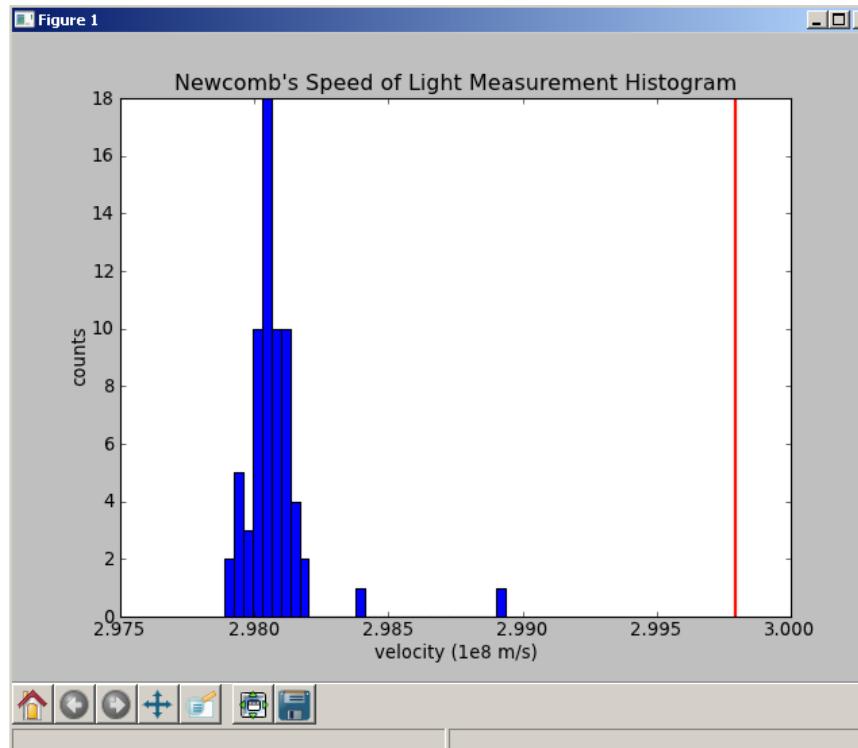
```
# tab completion
```

```
In [10]: %run newcomb_hi
```

```
newcomb_histogram.dat  newcomb_histogram.py
```

```
# execute a python file
```

```
In [11]: %run newcomb_histogram.py
```



# IPython History

## HISTORY COMMAND

```
# list previous commands. Use
# 'magic' % because 'hist' is
# histogram function in pylab
In [3]: %hist
a=1
a
```

## INPUT HISTORY

```
# list string from prompt[2]
In [4]: _i2
Out[4]: 'a\n'
```

## OUTPUT HISTORY

```
# grab previous result
In [5]: _
Out[5]: 'a\n'
```

```
# grab result from prompt[2]
In [6]: _2
Out[6]: 1
```



The up and down arrows scroll through your ipython input history.

# Reading Simple Tracebacks

## ERROR ADDING AN INTEGER TO A STRING

```
In [9]: 1 + "hello"
```

```
-----
```

```
TypeError      Traceback (most recent call last)
```

```
C:\...<ipython-input...> in <module>()
```

```
----> 1 1 + "hello"
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The “type” of error  
that occurred.

Location and code  
where error occurred.

Short message about  
why it occurred.

## ERROR TRYING TO ADD A NON-EXISTENT VARIABLE

```
# Again we fail when adding two variables, but note that the
# traceback tells us we have a completely different problem.
# In this case, our variable doesn't exist, so the operation fails.
In [10]: undefined_var + 1
```

```
...
```

```
NameError: name 'undefined_var' is not defined
```

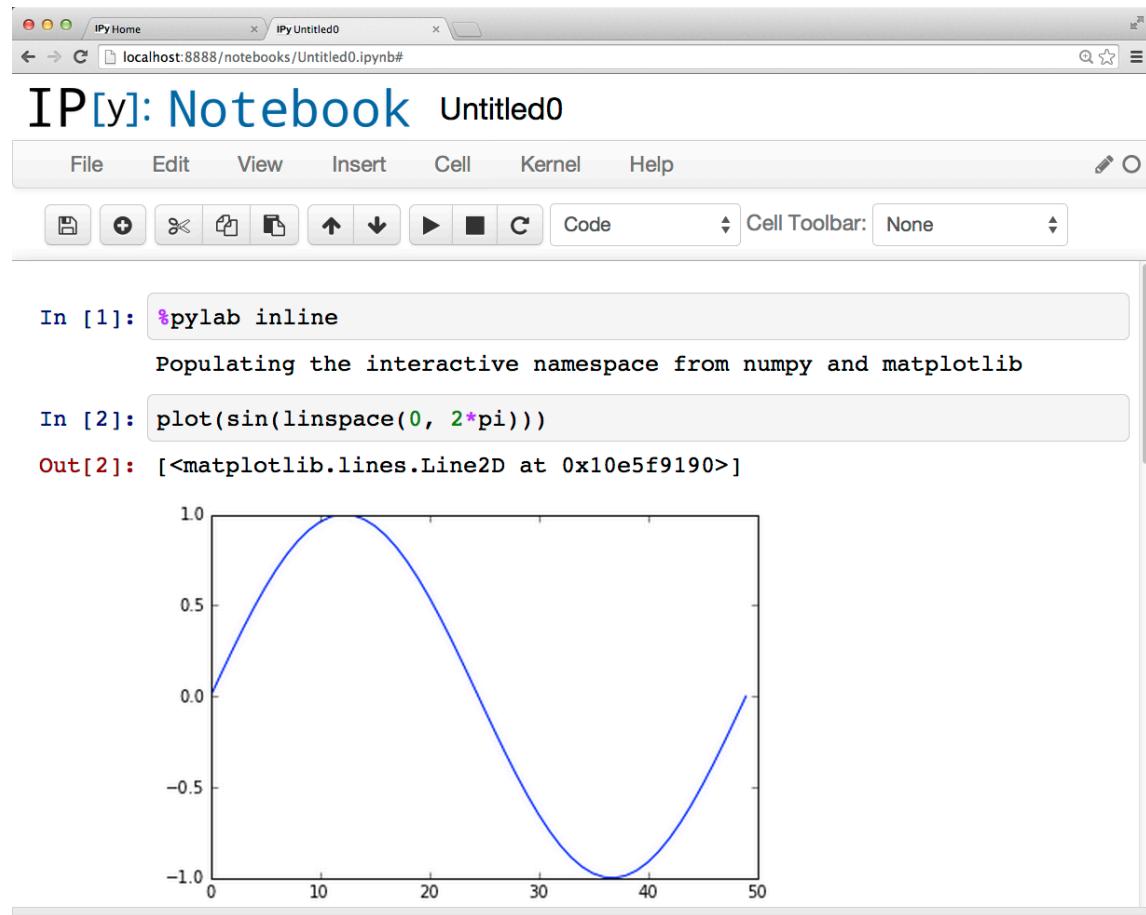
# IPython Notebook: Share your results

One .ipynb file with:

- code
- results
- inline figures
- formatted text (including equations)
- titles
- etc

Easy to share results:

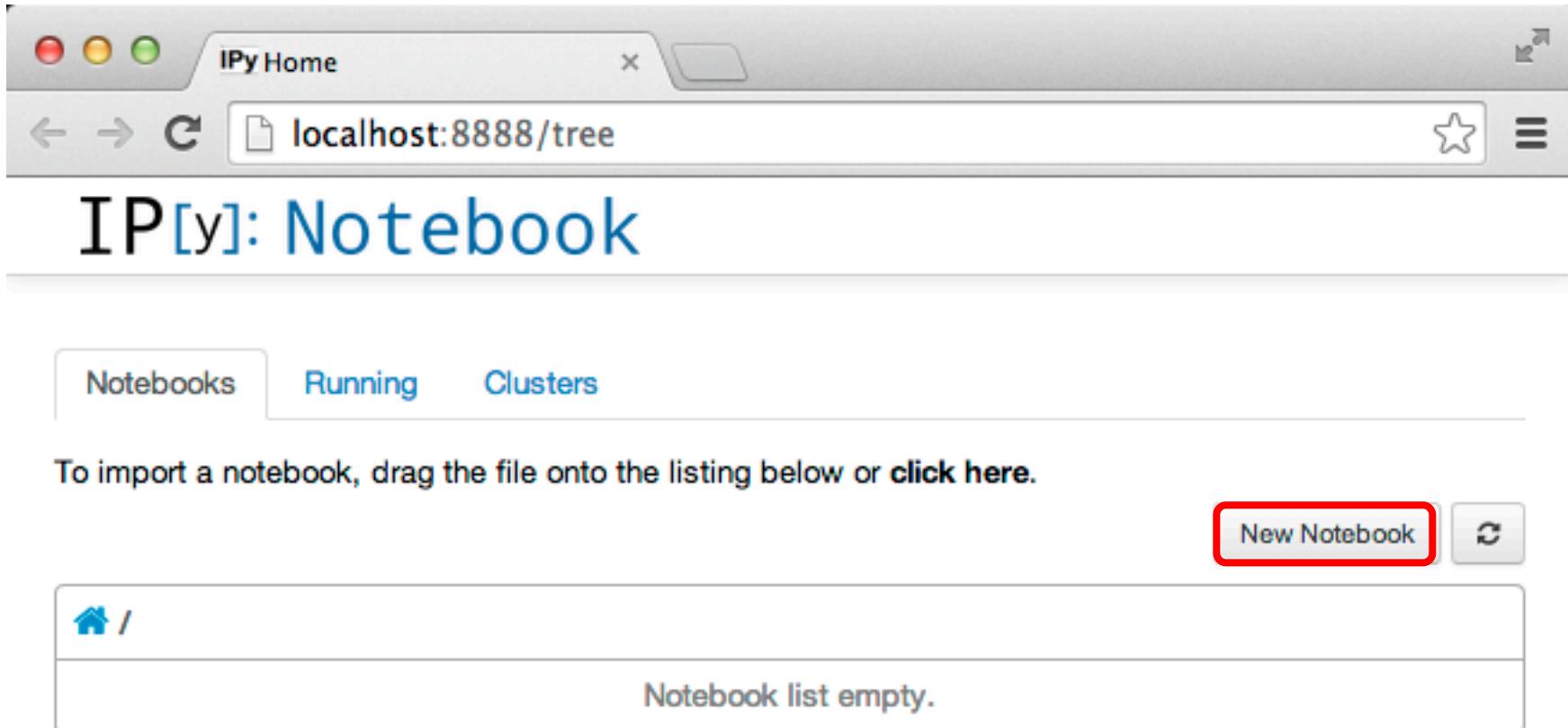
- results inline with code
- code is executable



# Creating a notebook from terminal

From a terminal/command prompt, start a notebook server that is viewed in your default web browser:

```
$ ipython notebook
```



The screenshot shows a web browser window titled "IPy Home" displaying the IPython Notebook interface at "localhost:8888/tree". The title bar includes standard OS X window controls (red, yellow, green) and a maximize/minimize button. The address bar shows the URL. Below the address bar, the text "IP[y]: Notebook" is displayed in large blue letters. A navigation menu bar below the title contains three tabs: "Notebooks" (selected), "Running", and "Clusters". A main content area contains the instruction "To import a notebook, drag the file onto the listing below or click here." followed by a red-bordered "New Notebook" button and a refresh icon. At the bottom, there is a navigation bar with icons for home and back/forward, and the message "Notebook list empty." is centered in the main content area.

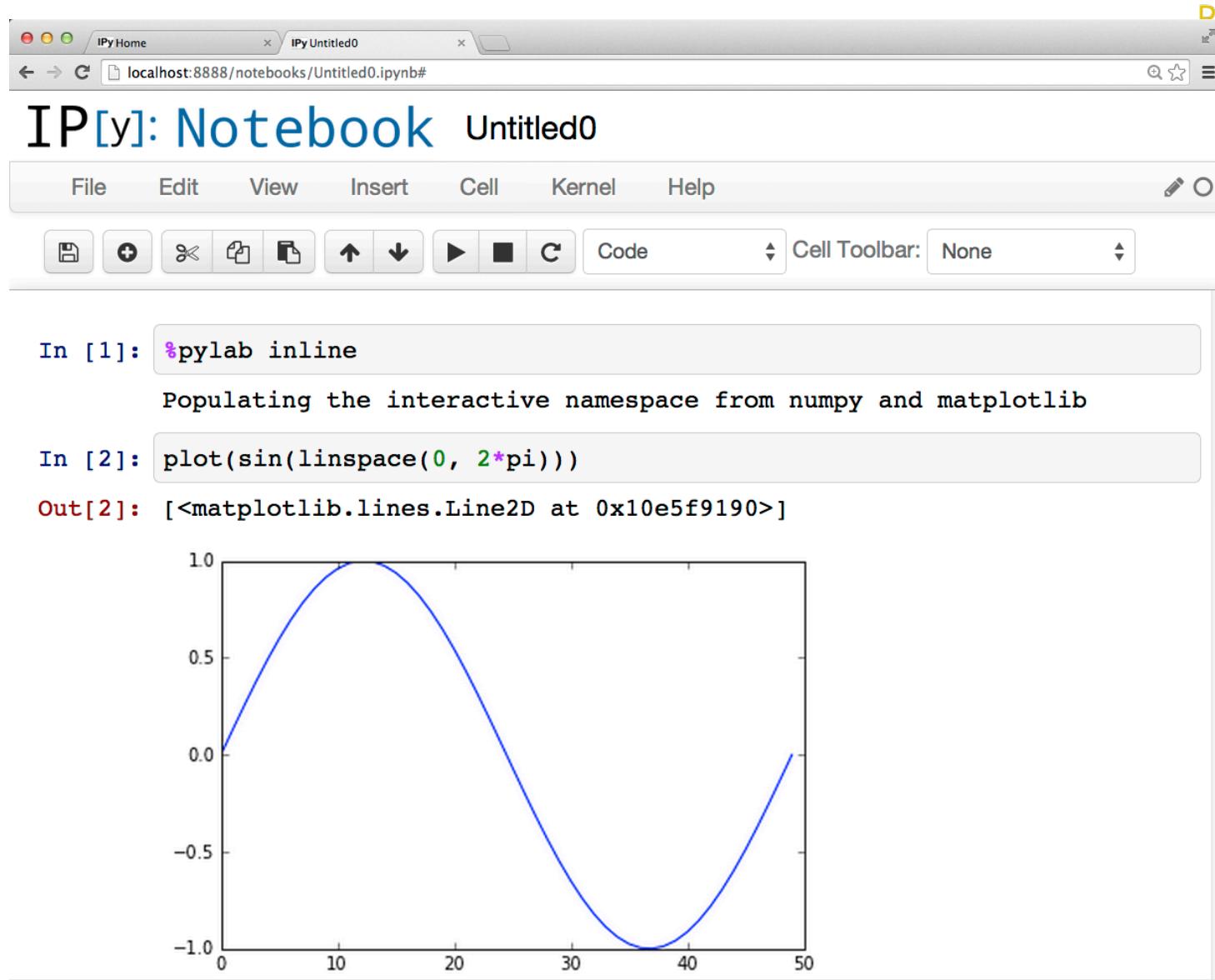
# IPython notebook features

1. Cells can group multiple commands. Execute cells with SHIFT-ENTER.
2. Make a cell a “Markdown” cell to create titles, control the font, ...
3. Cells can be deleted, moved around, inserted AND executed in random order.
4. Insert images, webpages, LaTeX formulas, YouTube videos, ... using `IPython.display` objects or functions:

```
In [2]: from IPython.display import Latex
Latex("$\int_a^b f(x) dx$")
```

```
Out[2]:  $\int_a^b f(x) dx$ 
```

# Inline figures in the notebook



The screenshot shows an IPython Notebook interface with the title "IP[y]: Notebook Untitled0". The toolbar includes standard file operations and a "Cell Toolbar" dropdown set to "None".

In [1]:

```
%pylab inline
```

Populating the interactive namespace from numpy and matplotlib

In [2]:

```
plot(sin(linspace(0, 2*pi)))
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x10e5f9190>]
```

A blue sine wave plot is displayed, showing the function  $\sin(x)$  for  $x \in [0, 2\pi]$ . The x-axis ranges from 0 to 50, and the y-axis ranges from -1.0 to 1.0. The curve starts at (0,0), reaches a peak of approximately 0.9 at  $x \approx 12$ , crosses the x-axis at  $x \approx 24$ , reaches a minimum of approximately -0.9 at  $x \approx 36$ , and returns to zero at  $x = 2\pi \approx 6.28$ .

# Introduction to Python Software Craftsmanship

# Software Engineering Quotes

*Programs should be written for people to read, and only incidentally for machines to execute.*

Structure and Interpretation of Computer Programs  
Harold Abelson and Gerald Sussman

# Software Engineering Quotes

*You need to have empathy not just for your users, but for your readers. It's in your interest, because you'll be one of them. Many a hacker has written a program only to find on returning to it six months later that he has no idea how it works.*

Hackers and Painters  
Paul Graham

# Software Engineering Quotes

*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*

Brian Kernighan  
co-author of “The C Programming Language”

# Software Carpentry

**Greg Wilson et. al., "Best Practices for Scientific Computing". <http://arxiv.org/abs/1210.0530>**

An 11-page paper giving recommendations for improving productivity and software reliability for those developing scientific software.

# Coding Modes

Python developers code in two modes:

**Interactive Mode:** Quick Iteration

Interactive Prompt and exploratory development.

**Production Mode:** Building for the Ages

Creating code that will be re-used by you or others.

# Naming Variables

Goal: Read the code and understand without having to think

# Typical Scientific Naming Convention

## STRAIGHT FROM FFTPACK

```
SUBROUTINE CFFTBL (N,C,CH,WA,IFAC)
  DIMENSION          CH(*)           ,C(*)           ,WA(*)           ,IFAC(*)
  NF = IFAC(2)
  NA = 0
  L1 = 1
  IW = 1
  DO 116 K1=1,NF
    IP = IFAC(K1+2)
    L2 = IP*L1
    IDO = N/L2
    IDOT = IDO+IDO
    IDL1 = IDOT*L1
    IF (IP .NE. 4) GO TO 103
    IX2 = IW+IDOT
    IX3 = IX2+IDOT
    IF (NA .NE. 0) GO TO 101
    CALL PASSB4 (IDOT,L1,C,CH,WA(IW),WA(IX2),WA(IX3))
    GO TO 102
<and on and on for 368 lines...>
```

# Primary Naming Consideration

A variable name should fully and accurately describe the entity and variable it represents.

## POOR NAME CHOICES

```
# Update Cash Balance after stock trade.  
c1 = n * ip  
c2 = c1 + compute_tc(ins, n)  
b -= c2
```

## DESCRIPTIVE NAME CHOICES

```
# Update Cash Balance after stock trade.  
instrument_cost = instrument_quantity * instrument_price  
trade_cost = instrument_cost + transaction_cost(instrument_name,  
                                                instrument_quantity)  
cash_balance -= trade_cost
```

# Using *extremely* short names

## LOOP INDICES I, J, and K

```
# This is ok.  
for i in xrange(10) :  
    scores[i] = 0  
  
# But this is better.  
events = xrange(10)  
for event in events:  
    decathlon_scores[event] = 0
```

# Using *extremely* short names

## INDUSTRY STANDARD VARIABLES IN “SMALL” CONTEXT

```
# Quick, what does each variable stand for?  
y = a * sin(w*t + phi)
```

# Using *extremely* short names

DATA SCIENCE

## INDUSTRY STANDARD VARIABLES IN “SMALL” CONTEXT

```
def sin_wave(t, a=1, w=2*pi, phi=0):  
    """  
        Return a sin wave form for time t.  
    """
```

### Inputs

-----

t: time in seconds

a: amplitude scale factor

w: frequency in radians/second

phi: phase shift in radians

### Returns

-----

y: sin wave output

"""

```
y = a * sin(w*t + phi)
```

```
return y
```

# Bad Code Comments

## REPEATING THE CODE

Comments that just repeat what's in the code are pretty much useless.

```
# Check if the printer is ready.  
if printer_status == 'ready':  
    document.print()
```

## INCORRECT COMMENTS

This comment is not even accurate. It likely got out of sync with the code when the bank implemented a minimum balance policy and the comment wasn't updated.

```
# Flag withdrawals that cause  
# customer balance to become  
# negative.  
new_balance = balance - withdrawal  
if new_balance < min_allowed_balance:  
    success = False
```

# Better Code Comments

## SUMMARY COMMENTS

Summarizing a few lines with a description of the code's intent is useful.

```
# Solve the dense linear system
# ZI=V for the currents I, given
# the impedance matrix Z and the
# driving voltage V.
lu_matrix, pivot = lu_factor(Z)
I = lu_solve((lu_matrix, pivot), v)
```

## FIXME COMMENTS

Flag design decisions and trade-offs that others should be aware of when editing code in the future.

```
# FIXME: Sales tax hard coded to
# 8.25%. This should be passed in
# or looked up with a function
# call.
price_total = price * (1.0825)
```

# Comments in production Python code

## PERCENTAGE OF PYTHON SOURCE LINES THAT ARE COMMENT LINES

<u>Project</u>	<u>Comments</u>
numpy	40.4%
scipy	37.2%
pandas	20.5%
matplotlib	27.8%
ipython	32.1%
traits	39.2%
chaco	27.4%

\* As determined by `cloc`: <http://cloc.sourceforge.net> on `master` versions as of 2014-12-17.

# Python Coding Standard

The Python Coding Standard is defined in Python Enhancement Proposal 8\* (PEP-8).

\* <http://www.python.org/dev/peps/pep-0008/>

# Testing Code

# Test Driven Development

---

**Overarching Concept:**

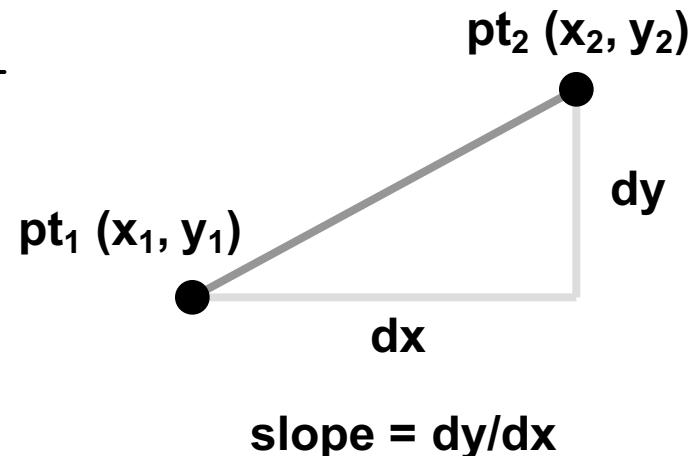
**Write tests as you write your code**

# Test Driven Development

## TEST

```
# test_fancy_math.py
from nose.tools import assert_almost_equal
from fancy_math import slope

def test_slope():
    pt1 = [0.0, 0.0]
    pt2 = [1.0, 2.0]
    s = slope(pt1, pt2)
    assert_almost_equal(s, 2)
```



## FANCY\_MATH

```
# Simplest function that passes, tests...
def slope(pt1, pt2):
    return 2
```

# TDD Rationale

---

Build only the features you need.  
(YAGNI Principle – you ain't gonna need it)

All the features are tested.

You are the first consumer of your API.  
(Helps in design process)

# Discovering tests: nosetests

## FROM THE TERMINAL

```
$ ls fancy_package/  
fancy_math.py  fancy_physics.py tests/  
  
$ ls fancy_package/tests/  
test_fancy_math.py test_fancy_physics.py  
  
$ nosetests -v  
test_fancy_math_solution.test_slope_float ... ok  
Integer division has the potential to break the slope function. ...  
ok  
Test for infinite slope function. ... ok  
-----  
Ran 3 tests in 0.095s  
  
OK
```

# Using Unittest Framework

## TESTS EMBEDDED IN CLASSES

```
# Test cases:  
#   - each method is a unit test  
#   - found by nosetests as well  
  
from unittest import TestCase  
  
from fancy_math import slope  
  
class TestModule(TestCase):  
  
    def test_slope(self):  
        pt1 = [0.0, 0.0]  
        pt2 = [1.0, 2.0]  
        s = slope(pt1, pt2)  
        self.assertAlmostEqual(s, 2)
```

# Timing and Profiling Code

# Ways to time execution

Timing inside the code (**Good**)

- the time module from std lib

Timing in ipython (**Better**):

- %timeit “magic command”
- -t option of ‘run’ (optionally –N also)

Profiling the code (**Best**):

- cProfile or line\_profiler package

# Timing in Python

## USE TIME PACKAGE

```

import time
from numpy.random import randn
from numpy import linspace, pi, exp, sin
from scipy.optimize import leastsq

def func(x,A,a,f,phi):
    return A*exp(-a*sin(f*x+phi))

def errfunc(params, x, data):
    return func(x, *params) - data

start = time.time()
params0 = [1,1,1,1]
x = linspace(0,2*pi,25)
ptrue = [3,2,1,pi/4]
true = func(x, *ptrue)
noisy = true + 0.3*randn(len(x))
pmin,ierr = leastsq(errfunc, params0,
                     args=(x, noisy))
print('Total: %f s' %(time.time()-start))

```

## USE IPYTHON TOOLS

```

# For a script
>>> run -t [-N10] test.py
IPython CPU timings (estimated):
  User   :      1.10 s.
  System :      0.00 s.
Wall time:      1.11 s.

```

```

# For operations/function call
>>> import numpy as np
>>> a = np.arange(1000)
>>> %timeit a**2
100000 loops, best of 3: 3.26 µs
per loop
>>> %timeit a**2.1
10000 loops, best of 3: 66.7 µs
per loop
>>> %timeit a*a
100000 loops, best of 3: 2.29 µs
per loop

```

# Profiling with cProfile

DATA SCIENCE

`cProfile` (and its pure python version, `profile`) are profiling tools in the standard library.

## WORKFLOW

The `cProfile` workflow has two main steps:

1. Run the code to be profiled via the `cProfile`'s `run()` (or `runctx()`) function. This counts and times function calls, and generates a profiling dataset.
2. Process and display the profile data. In the simplest case (e.g. `cProfile.run('foo()')`), a predefined report is generated and printed. For finer control, you can save the raw data to a file and process it using the `pstats` module.

These 2 steps can be executed automatically using the IPython `%run -p` magic command.

# Automatic method

The most convenient way to profile a script execution at the function level is to use the `-p` option of `%run` from within IPython:

```
>>> %run -p solve-sudoku.py easy-sudoku.txt

5752950 function calls (3568633 primitive calls) in 4.022 seconds

Ordered by: internal time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1006113/52612    2.336    0.000    3.762    0.000 solve-sudoku.py:94(eliminate)
964245/255618    0.542    0.000    3.218    0.000 solve-sudoku.py:102(<genexpr>)
206977/14262    0.247    0.000    3.850    0.000 {all}
   2575327    0.242    0.000    0.242    0.000 {len}
151887/12264    0.186    0.000    3.850    0.000 solve-sudoku.py:87(assign)
   457535    0.175    0.000    0.175    0.000 {method 'replace' of 'str' objects}
246847/64876    0.149    0.000    3.812    0.000 solve-sudoku.py:89(<genexpr>)
    75832    0.054    0.000    0.072    0.000 solve-sudoku.py:158(<genexpr>)
     1633    0.020    0.000    0.092    0.000 {min}
      366    0.014    0.000    2.199    0.006 solve-sudoku.py:129(initialize)
     41810    0.013    0.000    0.016    0.000 solve-sudoku.py:117(<genexpr>)
  4496/793    0.008    0.000    1.781    0.002 solve-sudoku.py:159(<genexpr>)
[...]
```

time in *this* function only

time in *this* function

+ all called functions

# Controlled method

## EXAMPLE

```
>>> import time
>>> def func(n):
...     if n < 0:
...         return
...     time.sleep(0.1*n)
...     func(n-1)
...     return
...
>>> import cProfile
>>> cProfile.run('func(3)')
    11 function calls (7 primitive calls) in 0.601 seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
5/1	0.000	0.000	0.601	0.601	<stdin>:1(func)
1	0.000	0.000	0.601	0.601	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	{method 'disable' of ...}
4	0.600	0.150	0.600	0.150	{time.sleep}

# Profiling with cProfile

## OUTPUT TABLE COLUMNS

<b>ncalls</b>	The number of calls. Counts of the form N/M indicate N actual calls (including recursive calls), and M 'primitive' (nonrecursive) calls.
<b>tottime</b>	The total time spent in the given function (and excluding time made in calls to sub-functions).
<b>percall</b>	The quotient of <b>tottime</b> divided by <b>ncalls</b> .
<b>cumtime</b>	The total time spent in this and all subfunctions (from invocation until exit). This figure is accurate even for recursive functions.
<b>percall</b>	The quotient of <b>cumtime</b> divided by primitive calls.
<b>filename:lineno(function)</b>	Provides the respective data of each function.

# line\_profiler and kernprof

`line_profiler` and `kernprof` are profiling tools developed by Robert Kern.

- `line_profiler` is a module for doing line-by-line profiling of functions.
- `kernprof` is a convenient script for running either `line_profiler` or the standard library's `cProfile` module.

## INSTALLATION

```
$ easy_install line_profiler
```

## TYPICAL WORKFLOW

1. Decorate the functions to be profiled with `@profile`.
2. Run your script using `kernprof.py` with the `-l` option. For example,

```
$ kernprof.py -l script_to_profile.py
```

3. Run the `line_profiler` module to display the results. For example,

```
$ python -m line_profiler script_to_profile.py.lprof
```

4. Adjust your code, and repeat steps 2-4.
5. Remove the `@profile` decorators.

# line\_profiler example

## http\_search.py

```
import re

PATTERN = r"https?:\/\/[ \w\-\_]+\(\.\. [ \w\-\_]+\)+([ \w\-
\.,@?^=%&; :/~/+\#]*[\w\-\@\?^=%&; /~/+\#])?""

@profile
def scan_for_http(f):
    addresses = []
    for line in f:
        result = re.search(PATTERN, line)
        if result:
            addresses.append(result.group(0))
    return addresses

if __name__ == "__main__":
    import sys
    f = open(sys.argv[1], 'r')
    addresses = scan_for_http(f)
    for address in addresses:
        print(address)
```



See demo/profiling directory  
for code.

# line\_profiler example

## Run kernprof.py and line\_profiler

```
$ kernprof.py -l http_search.py sample.html
...
http://sphinx.pocoo.org/
Wrote profile results to http_search.py.lprof

$ python -m line_profiler http_search.py.lprof
```

Timer unit: 1e-06 s

File: http\_search.py  
Function: scan\_for\_http at line 6  
Total time: 0.016079 s

Line #	Hits	Time	Per Hit	% Time	Line Contents
<hr/>					
6					@profile
7					def scan_for_http(f):
8	1	3	3.0	0.0	addresses = []
9	1350	2080	1.5	12.9	for line in f:
10	1349	12417	9.2	77.2	result = re.search(PATTERN, line)
11	1349	1513	1.1	9.4	if result:
12	39	65	1.7	0.4	addresses.append(result.group(0))
13	1	1	1.0	0.0	return addresses

# line\_profiler example

## http\_search2.py

```
import re

PATTERN = r"https?:\/\/[ \w\-\_]+\(\.\[ \w\-\_]+\)+([ \w\-
\.,@?^=%&;:/~\+\#]*[ \w\-\@\?^=%&;/~\+\#])?""

@profile
def scan_for_http(f):
    addresses = []
    pat = re.compile(PATTERN)
    for line in f:
        result = pat.search(line)
        if result:
            addresses.append(result.group(0))
    return addresses

if __name__ == "__main__":
    import sys
    f = open(sys.argv[1], 'r')
    addresses = scan_for_http(f)
    for address in addresses:
        print(address)
```

# line\_profiler example

RUN kernprof.py AND line\_profiler ON THE MODIFIED FILE

```
$ kernprof.py -l http_search2.py sample.html
```

...

```
Wrote profile results to http_search2.py.lprof
```

```
$ python -m line_profiler http_search2.py.lprof
```

```
Timer unit: 1e-06 s
```

```
File: http_search2.py
```

```
Function: scan_for_http at line 6
```

```
Total time: 0.00911 s
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
<hr/>					
6					@profile
7					def scan_for_http(f):
8	1	3	3.0	0.0	addresses = []
9	1	3117	3117.0	34.2	pat = re.compile(PATTERN)
10	1350	1995	1.5	21.9	for line in f:
11	1349	2507	1.9	27.5	result = pat.search(line)
12	1349	1415	1.0	15.5	if result:
13	39	72	1.8	0.8	addresses.append(result.group(0))
14	1	1	1.0	0.0	return addresses

# **pdb**

## The Python debugger

# What is **pdb**?

**pdb** is part of the standard library.

**pdb**, like Python, is interactive and interpreted, allowing for the execution of arbitrary Python code in the context of any stack frame.

**pdb** can debug a “post-mortem” condition, and can also be called under program control.

**ipdb** (not in std lib) is similar but includes tab completion and syntax highlighting.

# Starting pdb

- Run the script from the **command line** under debugger control

```
C:\> python -m pdb script.py [arg ...]
```

- Call a function from the **pdb** module in an **IPython session**:

```
>>> pdb.run(statement)
```

Execute the statement (given as a string) under debugger control

```
>>> pdb.runcall(function[, argument, ...])
```

Call the function (not a string) with the given arguments under debugger control

```
>>> pdb.pm()
```

Start the debugger at the point of the last exception

- Hard-code a breakpoint inside **a script** or a module:

```
import pdb; pdb.set_trace()
```

# pdb commands

- **pdb** runs as an interactive session having a specific set of commands.
- Some of the more common **pdb** commands:

## (Pdb) h(help)[command]

One of the most important! Lists all the commands available, or help on a specific command.

## (Pdb) u(p) / d(own)

Pop up or push down the execution stack.

## (Pdb) b(reak)[[filename:]lineno | function[, condition]]

Set a breakpoint at a specific file/line or function and optionally if a specific condition is met. If no args are given, list all the breakpoints & their numbers.

## (Pdb) s(tep) / n(ext)

Execute the current line only. **step** will push into a function call and **next** will execute the function call and move to the next statement in the current function.

## (Pdb) a(rgs)

Print the args for the current function.

# pdb commands

## (Pdb) l(ist) [first[, last]]

List the source code at the point of execution. Args **first** and **last** set a range for the number of lines printed. No args prints 11 lines around the current line, if only **first**, prints 11 lines around that line.

## (Pdb) j(ump) lineno

Jump to a line in the bottom-most frame only and execute from there. Not all jumps are possible!

## (Pdb) p / pp [expression]

Print or “pretty print” **expression** in the context of the current frame.

## (Pdb) a(lias) [name [command]]

Create an alias for **command** named **name**, or list all aliases.

Here are two useful aliases (especially when placed in a .pdbrc file):

#print all instance variables (usage "pi classInst")

alias pi for k in %1.\_\_dict\_\_.keys(): print "%1.",k,"=",%1.\_\_dict\_\_[k]

#print instance variables in self

alias ps pi self

# IPython and pdb

```
# ipython can call pdb automatically upon error
```

```
In [1]: pdb
```

```
Automatic pdb calling has been turned ON
```

```
In [2]: import middle
```

```
In [3]: middle.run()
```

```
-----  
IndexError                                     Traceback (most recent call last)  
Z:\projects\Training\pdb\<console>
```

```
Z:\projects\Training\pdb\middle.py  in run()  
 31      """  
 32          for i in range( 1, 11 ) :  
 33              l = make_list( i )  
---> 34          print "The middle item(s) in %s\n\tis/are %s\n" % (l, get_middle( l ))  
 35
```

```
Z:\projects\Training\pdb\middle.py  in get_middle(item_list)  
  9  
 10      if( num_items % 2 ) :  
---> 11          return item_list[half]  
 12  
 13      return item_list[(half - 1):(half + 1)]
```

```
IndexError: list index out of range  
> z:\projects\training\pdb\middle.py(11)get_middle()  
-> return item_list[half]
```

# Example session

## DEBUGGING FROM ORIGIN OF EXCEPTION

```
ipdb> 1
      """
6      num_items = len( item_list )
7      half = num_items * 2
8
9
10     if( num_items % 2 ) :
11         ->     return item_list[half]
12
13     return item_list[ (half - 1):(half + 1) ]
14
15
16     def make_list( size=0 ) :
```

*Print the source code around  
the line which raised the  
exception*

```
ipdb> item_list
['0']
ipdb> half
2
ipdb> c
```

*Print the contents of the list*

*Print the value of the index  
ah-ha!*

In [4]:

*Return to ipython*

# Example session

## DEBUGGING MIDDLE.PY FROM THE START

```
>>> import pdb
>>> import middle
>>> pdb.runcall( middle.run )
> z:\projects\pgtraining\pdb\middle.py(32) run()
-> for i in range( 1, 11 ) :
(Pdb) s
> z:\projects\pgtraining\pdb\middle.py(22) run()
-> l = make_list( i )
(Pdb) n
> z:\projects\pgtraining\pdb\middle.py(34) run()
-> print "The middle item(s) in %s\n\tis/are %s\n" % (l, get_middle( l ))
(Pdb) s
--Call--
> z:\projects\pgtraining\pdb\middle.py(2) get_middle()
-> def get_middle( item_list ) :
(Pdb) s
...
> z:\projects\pgtraining\pdb\middle.py(11) get_middle()
-> return item_list[half]
(Pdb) item_list
['0']
(Pdb) half
2
```

We know make\_list is ok, so skip over it with “next”

Continue to execute lines until we see something suspicious

Print the contents of the list  
Print the value of the index  
ah-ha!

# Other debugging tools

- ipdb (not in std lib) offers the same functionalities as pdb (set\_trace allowing to march through execution) but allow more interactive exploration thanks to the tab completion like in ipython. BUT still only allow 1 line evaluations.
- To do more exploration at a given point in an application, IPython can be invoked, with its embed function:

```
from IPython import embed ; embed()
```

It starts a normal ipython session with the namespace populated from the namespace of your application at the break point. To exit, ctrl-d.

# DAY 2

# Introduction to Python

## Data types

# Outline

---

- Data types:
  - Numerical types: int, long, float, complex
  - Booleans
  - Strings
  - Lists and tuples
  - Dictionaries and sets
  - Things to know about efficiency

# Interactive Calculator

```
# adding two values
>>> 1 + 1
2

# setting a variable
>>> a = 1
>>> a
1

# checking a variable's type
>>> type(a)
<type 'int'>

# an arbitrarily long integer
>>> a = 12345678901234567890
>>> a
12345678901234567890L

>>> type(a)
<type 'long'>

# Remove 'a' from the 'namespace'
>>> del a
>>> a
NameError: name 'a' is not
defined
```

```
# real numbers
>>> b = 1.4 + 2.3
>>> b
3.6999999999999997
# "prettier" version.
>>> print b
3.7

>>> type(b)
<type 'float'>

# complex numbers
>>> c = 2+1.5j
>>> c
(2+1.5j)
```

The four numeric types in Python on 64-bit architectures are:

 integer **8 byte (4 byte on Windows)**  
 long integer **Any precision**  
 float **8 byte, like C's double**  
 complex **16 byte**

The NumPy library, which we will see later, supports a larger number of numeric types

# More Interactive Calculation

## ARITHMETIC OPERATIONS

```
>>> 1+2- (3*4/ 6) **5+ (7%5)  
-27
```

## SIMPLE MATH FUNCTIONS

```
>>> abs (-3)  
3  
>>> max (0, min(10, -1, 4, 3))  
0  
>>> round(2.718281828)  
3.0
```

## OVERWRITING FUNCTIONS

```
# don't do this  
>>> max = 100  
  
# ...some time later...  
>>> x = max(4, 5)  
TypeError: 'int' object is not  
callable
```

## TYPE CONVERSION

```
>>> int(2.718281828)  
2  
>>> float(2)  
2.0  
>>> 1+2.0  
3.0
```

## IN-PLACE OPERATIONS

```
>>> b = 2.5  
>>> b += 0.5      # b = b + 0.5  
>>> b  
3.0  
# Also -=, *=, /=, etc.
```

# Logical expressions, bool data type

## COMPARISON OPERATORS

```
# <, >, <=, >=, ==, !=
>>> 1 >= 2
False
>>> 1 + 1 == 2
True
>>> 2**3 != 3**2
True
# Chained comparisons
>>> 1 < 10 < 100
True
```

## bool DATA TYPE

```
>>> q = 1 > 0
>>> q
True
>>> type(q)
<type 'bool'>
```

## and OPERATOR

```
>>> 1 > 0 and 5 == 5
True
# If first operand is false,
# the second is not evaluated.
>>> 1 < 0 and max(0,1,2) > 1
False
```

## or OPERATOR

```
>>> a = 50
>>> a < 10 or a > 90
False
# If first operand is true,
# the second is not evaluated.
>>> a = 0
>>> a < 10 or a > 90
True
```

## not OPERATOR

```
>>> not 10 <= a <= 90
True
```

# Strings

## CREATING STRINGS

```
# using double quotes
>>> s = "hello world"
>>> print s
hello world
# single quotes also work
>>> s = 'hello world'
>>> print s
hello world
```

## STRING OPERATIONS

```
# concatenating two strings
>>> "hello " + "world"
'hello world'

# repeating a string
>>> "hello " * 3
'hello hello hello '
```

## STRING LENGTH

```
>>> s = "12345"
>>> len(s)
5
```

## SPLIT/JOIN STRINGS

```
# split space-delimited words
>>> s = "hello world"
>>> wrd_lst = s.split()
>>> print wrd_lst
['hello', 'world']

# join words back together
# with a space in between
>>> space = ' '
>>> space.join(wrd_lst)
'hello world'
```

# Multi-line Strings

DATA SCIENCE

## TRIPLE QUOTES

```
# Strings in triple quotes retain line breaks
>>> a = """hello
... world"""
>>> print a
hello
world
```

## NEW LINE CHARACTER

```
# Including a newline character
>>> a = "hello\nworld"
>>> print a
hello
world
```

# A few string methods and functions

## REPLACING TEXT

```
>>> s = "hello world"  
>>> s.replace('world','logit')  
'hello logit'
```

## CONVERT TO UPPER CASE

```
>>> s.upper()  
'HELLO WORLD'
```

## REMOVE WHITESPACE

```
>>> s = "\t    hello\n"  
>>> s.strip()  
'hello'
```

## NUMBERS TO STRINGS

```
>>> str(1.1 + 2.2)  
'3.3'  
>>> repr(1.1 + 2.2)  
'3.3000000000000003'  
>>> str(1)  
'1'
```

## STRINGS TO NUMBERS

```
>>> int('23')  
23  
>>> int('FF', 16)  
255  
>>> float('23')  
23.0
```

# String Formatting

The `format()` method replaces any ***replacement fields*** in the string with the values given as arguments.

**Replacement field format:** {*<name>* *Optional* :*<format\_spec>*} *Optional*

```
# If 'name' is an integer, it refers to the argument position.  
>>> '{0} is greater than {1}'.format(100, 50)  
'100 is greater than 50'
```

```
# If 'name' is text, it refers to a keyword argument.  
>>> '{last}, {first}'.format(first='Ellen', last='Ripley')  
'Ripley, Ellen'
```

# String Formatting – Format spec

The optional format specification is used to control how the values are displayed. (See Appendix for details.)

```
# Fixed point format (and a named keyword argument).
>>> print '[{x:5.0f}]  [{x:5.1f}]  [{x:5.2f}]'.format(x=12.3456)
[ 12]  [ 12.3]  [12.35]

# Alignment (and using a numbered positional argument).
>>> print '[{0:<10s}]  [{0:>10s}]  [{0:^10s}]'.format('PYTHON')
[PYTHON      ]  [        PYTHON]  [    PYTHON    ]

# Alignment with fill character.
>>> template = '[{0:*<10s}]  [{0:*>10s}]  [{0:*^10s}]'
>>> print template.format('PYTHON')
[PYTHON****]  [****PYTHON]  [**PYTHON**]
```

# String Formatting – Format spec

```
>>> 'price: ${0:=-7.2f}'.format(3.4)
'price: $    3.40'
```

The *format spec* is a sequence of characters including:

- the *alignment* option,
- the *sign* option,
- the *width* (and *.precision*) option
- the *type code*.

## ALIGNMENT OPTION

### Char Meaning

<	Left aligned.
>	Right aligned.
=	(For numeric types only.) Pad after the sign but before the digits (e.g. +000000120).
^	Center within the available space.

If an alignment character is given, it may be preceded by a *fill character*.

## SIGN OPTION

### For numbers only.

### Char Meaning

+	Include a sign for positive and negative number.
-	Indicate sign for negative numbers only (default)
space	Include a leading space for positive numbers.

## STRING TYPE CODES

### Type Meaning

s String. This is the default, and may be omitted.

## INTEGER TYPE CODES

### Type Meaning

b	Binary format.
c	Character; converts int to unicode char.
d	Decimal integer (base 10).
o	Octal (base 8).
x	Hex (base 16), lower case.
X	Hex (base 16), upper case.
n	Number; same as 'd', but uses current locale.
None	Same as 'd'.

## FLOATING POINT TYPE CODES

### Type Meaning

e	Scientific notation.
E	Scientific notation, with upper case 'E'.
f	Fixed point.
F	Fixed point; same as 'f'.
g	General format.
G	General format; same as 'g', with upper case 'E' when necessary.
n	Number; same as 'g', but uses current locale.
%	Percentage. Multiplies by 100 and displays with 'f', followed by a percent sign.
None	Same as 'g'.

# String Formatting with %

## FORMAT OPERATOR %

```
# the % operator formats values
# to strings using C conventions.
>>> s = "some numbers:"
>>> x = 1.34
>>> y = 2
>>> t = "%s %f, %d" % (s,x,y)
>>> print t
some numbers: 1.340000, 2

>>> y = -2.1
>>> print "%f\n%f" % (x,y)
1.340000
-2.100000

>>> print "% f\n% f" % (x,y)
1.340000
-2.100000

>>> print "%4.2f" % x
1.34
```

## CONVERSION CODES

Conversion	Meaning
d or i	Signed integer decimal
o	Unsigned octal
u	Unsigned decimal
x	Unsigned hexadecimal (lowercase)
X	Unsigned hexadecimal (uppercase)
e	Floating point exponential format(lowercase)
E	Floating point exponential format(uppercase)
F or f	Floating point decimal format
G or g	Floating point format or exponential
c	Single character
r	Converts object using repr()
s	Converts object using str()

## CONVERSION FLAGS

Flag	Meaning
0	The conversion will be zero padded for numeric values.
-	The converted value is left adjusted (overrides the "0" conversion if both are given).
<space>	(a space)A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

# List objects

## LIST CREATION WITH BRACKETS

```
>>> a = [10,11,12,13,14]
>>> print a
[10, 11, 12, 13, 14]
```

## CONCATENATING LIST

```
# simply use the + operator
>>> [10, 11] + [12, 13]
[10, 11, 12, 13]
```

## REPEATING ELEMENTS IN LISTS

```
# the multiply operator
# does the trick
>>> [10, 11] * 3
[10, 11, 10, 11, 10, 11]
```

## range(start, stop, step)

```
# the range function is helpful
# for creating a sequence
>>> range(5)
[0, 1, 2, 3, 4]

>>> range(2,7)
[2, 3, 4, 5, 6]

>>> range(2,7,2)
[2, 4, 6]
```

# Indexing

## RETRIEVING AN ELEMENT

```
# list
# indices: 0  1  2  3  4
>>> a = [10,11,12,13,14]
>>> a[0]
10
```

## SETTING AN ELEMENT

```
>>> a[1] = 21
>>> print a
[10, 21, 12, 13, 14]
```

## OUT OF BOUNDS

```
>>> a[10]
Traceback (innermost last):
File "<interactive input>", line 1, in ?
IndexError: list index out of range
```

## NEGATIVE INDICES

```
# negative indices count
# backward from the end of
# the list
#
# indices: -5 -4 -3 -2 -1
>>> a = [10,11,12,13,14]
```

```
>>> a[-1]
14
>>> a[-2]
13
```



The first element in an array has `index=0` as in C. **Take note Matlab and Fortran programmers!**

# More on list objects

## LIST CONTAINING MULTIPLE TYPES

```
# list containing integer,  
# string, and another list  
>>> a = [10, 'eleven', [12,13]]  
>>> a[1]  
'eleven'  
>>> a[2]  
[12, 13]  
  
# use multiple indices to  
# retrieve elements from  
# nested lists  
>>> a[2][0]  
12
```

## LENGTH OF A LIST

```
>>> len(a)  
3
```

## DELETING OBJECT FROM LIST

```
# use the del keyword  
>>> del a[2]  
>>> a  
[10, 'eleven']
```

## DOES THE LIST CONTAIN x ?

```
# use in or not in  
>>> a = [10,11,12,13,14]  
>>> 13 in a  
True  
>>> 13 not in a  
False
```

# Common methods for lists

## `some_list.append( x )`

Add the element x to the end of the list some\_list.

## `some_list.count( x )`

Count the number of times x occurs in the list.

## `some_list.extend( sequence )`

Concatenate sequence onto this list.

## `some_list.index( x )`

Return the index of the first occurrence of x in the list.

## `some_list.insert( index, x )`

Insert x before the specified index.

## `some_list.pop( index )`

Return the element at the specified index. Also, remove it from the list.

## `some_list.remove( x )`

Delete the first occurrence of x from the list.

## `some_list.reverse( )`

Reverse the order of elements in the list.

## `some_list.sort( key )`

By default, sort the elements in ascending order. If a key function is given, apply it to each element to determine the value for sorting.

# Slicing

**var [lower:upper:step]**

Extracts a portion of a sequence by specifying a lower and upper bound.

The lower-bound element is included, but the upper-bound element *is not included*.

Mathematically: [lower, upper). The step value specifies the stride between elements.

## SLICING LISTS

```
# indices:  
#      -5 -4 -3 -2 -1  
#      0  1  2  3  4  
>>> a = [10,11,12,13,14]  
# [10,11,12,13,14]  
>>> a[1:3]  
[11, 12]  
  
# negative indices work also  
>>> a[1:-2]  
[11, 12]  
>>> a[-4:3]  
[11, 12]
```

## OMITTING INDICES

```
# omitted boundaries are  
# assumed to be the beginning  
# (or end) of the list  
  
# grab first three elements  
>>> a[:3]  
[10, 11, 12]  
# grab last two elements  
>>> a[-2:]  
[13, 14]  
# every other element  
>>> a[::-2]  
[10, 12, 14]
```

```
>>> a = [10,21,23,11,24]  
  
# add an element to the list  
>>> a.append(11)  
>>> print a  
[10,21,23,11,24,11]  
# how many 11s are there?  
>>> a.count(11)  
2  
# extend with another list  
>>> a.extend([5,4])  
>>> print a  
[10,21,23,11,24,11,5,4]  
# where does 11 first occur?  
>>> a.index(11)  
3  
# insert 100 at index 2?  
>>> a.insert(2, 100)  
>>> print a  
[10,21,100,23,11,24,11,5,4]
```

```
# pop the item at index=3  
>>> a.pop(3)  
23  
# remove the first 11  
>>> a.remove(11)  
>>> print a  
[10,21,100,24,11,5,4]  
# sort the list (in-place)  
# Note: use sorted(a) to  
#       return a new list.  
>>> a.sort()  
>>> print a  
[4,5,10,11,21,24,100]  
# reverse the list  
>>> a.reverse()  
>>> print a  
[100,24,21,11,10,5,4]
```

# Mutable vs. Immutable

## MUTABLE OBJECTS

```
# Mutable objects, such as
# lists, can be changed
# in place.

# insert new values into list
>>> a = [10,11,12,13,14]
>>> a[1:3] = [5,6]
>>> print a
[10, 5, 6, 13, 14]
```

## IMMUTABLE OBJECTS

```
# Immutable objects, such as
# integers and strings,
# cannot be changed in place.

# try inserting values into
# a string
>>> s = 'abcde'
>>> s[1:3] = 'xy'
Traceback (innermost last):
File "<interactive input>", line 1, in ?
TypeError: object doesn't support
        slice assignment

# here's how to do it
>>> s = s[:1] + 'xy' + s[3:]
>>> print s
'axyde'
```

# Tuple – Immutable Sequence

## TUPLE CREATION

```
>>> a = (10,11,12,13,14)  
>>> print a  
(10, 11, 12, 13, 14)
```

## PARENTHESES ARE OPTIONAL

```
>>> a = 10,11,12,13,14  
>>> print a  
(10, 11, 12, 13, 14)
```

## LENGTH-1 TUPLE

```
>>> (10,)  
(10,)  
>>> (10)  
10
```



(10) is not a tuple,  
but an integer  
with parentheses.

## TUPLES ARE IMMUTABLE

```
# create a list  
>>> a = range(10,15)  
[10, 11, 12, 13, 14]  
  
# cast the list to a tuple  
>>> b = tuple(a)  
>>> print b  
(10, 11, 12, 13, 14)
```

```
# try inserting a value  
>>> b[3] = 23  
TypeError: 'tuple' object doesn't  
support item assignment
```

# Tuple (un)packing

## (UN)PACKING TUPLES

```
# Creating a tuple without ()
>>> d = 1, 2, 3
>>> d
(1, 2, 3)

# Multiple assignments from a
# tuple
>>> a, b, c = d
>>> print b
2

# Multiple assignments
>>> a, b, c = 1, 2, 3
>>> print b
2
```

## WHY IS IT USEFUL?

We will see later on that this feature is very common in Python code, e.g.:

```
# Returning multiple values
def monomials(x):
    return 1, x, x**2
a0, a1, a2 = monomials(3)

# Iterating over tuples
friends = [('Guido', 28),
            ('Mario', 51),
            ('Antonio', 30)]
for (name, age) in friends:
    txt = "{} is {} years old"
    print txt.format(name, age)
```

# Dictionaries

Dictionaries store *key/value* pairs. Indexing a dictionary by a *key* returns the *value* associated with it. The *key* must be immutable.

## DICTIONARY EXAMPLE

```
# Create an empty dictionary using curly brackets.
>>> record = {}

# Each indexed assignment creates a new key/value pair.
>>> record['first'] = 'Ralf'
>>> record['last'] = 'Emmerson'
>>> record['born'] = 1803
>>> print record
{'first': 'Ralf', 'born': 1803, 'last': 'Emmerson'}
# Create another dictionary with initial entries.
>>> new_record = {'first': 'Ralph', 'middle':'Waldo'}
# Now update the first dictionary with values from the new one.
>>> record.update(new_record)
>>> print record
{'first': 'Ralph', 'middle': 'Waldo', 'last':'Emmerson',
 'born': 1803}
```

# Accessing and deleting keys and values

DATA SCIENCE

## ACCESS USING INDEX NOTATION

```
>>> print record['first']  
Ralph
```

## ACCESS WITH get(key, default)

The `get()` method returns the value associated with a key; the optional second argument is the return value if the key is not in the dictionary.

```
>>> record.get('born',0)  
1803  
>>> record.get('home', 'TBD')  
'TBD'  
>>> record['home']  
KeyError: ...
```

## REMOVE AN ENTRY WITH DEL

```
>>> del record['middle']  
>>> record  
{'born': 1803, 'first':  
'Ralph', 'last': 'Emmerson'}
```

## REMOVE WITH pop(key, default)

`pop()` removes the key from the dictionary and returns the value; the optional second argument is the return value if the key is not in the dictionary.

```
>>> record.pop('born', 0)  
1803  
>>> record  
{'first': 'Ralph', 'last':  
'Emmerson'}  
>>> record.pop('born', 0)  
0
```

```
# dict of animals:count pairs
>>> cargo = { 'cows': 1,
...                 'dogs': 5,
...                 'cats': 3}
```

```
# test for chickens
>>> 'chickens' in cargo
False
```

```
# get a list of all keys
>>> cargo.keys()
['cats', 'dogs', 'cows']
```

```
# get a list of all values
>>> cargo.values()
[3, 5, 1]
```

```
# return key/value tuples
>>> cargo.items()
[('cats', 3), ('dogs', 5),
 ('cows', 1)]
```

```
# How many cats?
>>> cargo['cats']
3
```

```
# Change the number of cats.
>>> cargo['cats'] = 10
>>> cargo['cats']
10
```

```
# Add some horses.
>>> cargo['horses'] = 5
>>> cargo['horses']
5
```

# Common methods for dictionaries

## `some_dict.clear( )`

Remove all key/value pairs from the dictionary, `some_dict`.

## `some_dict.copy()`

Create a copy of the dictionary

## `x in some_dict`

Test whether the dictionary contains the key `x`.

## `some_dict.keys( )`

Return a list of all the keys in the dictionary.

## `some_dict.values( )`

Return a list of all the values in the dictionary.

## `some_dict.items( )`

Return a list of all the key/value pairs in the dictionary.

# Set objects

## DEFINITION

A set is an *unordered collection of unique, immutable objects*.

## CONSTRUCTION

```
# an empty set
>>> s = set()
# convert a sequence to set
>>> t = set([1,2,3,1])
# note removal of duplicates
>>> t
set([1, 2, 3])
```

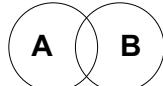
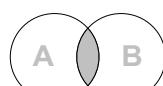
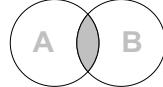
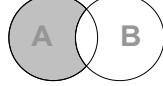
## ADD/REMOVE ELEMENTS

```
>>> t.add(5)
>>> t
set([1, 2, 3, 5])
>>> t.update([5,6,7])
>>> t
set([1, 2, 3, 5, 6, 7])
```

## REMOVE ELEMENTS

```
>>> t.remove(1)
set([2, 3, 5, 6, 7])
```

## SET OPERATIONS

>>> a = set([1,2,3,4])	
>>> b = set([3,4,5,6])	
>>> a.union(b)	
set([1, 2, 3, 4, 5, 6])	
>>> a.intersection(b)	
set([3, 4])	
>>> a.difference(b)	
set([1, 2])	
>>> a.symmetric_difference(b)	
set([1, 2, 5, 6])	

# Selecting a data type

Selecting the appropriate data type is important

	<b>insert</b>	<b>remove</b>	<b>find</b>	<b>ordered</b>
list	linear	linear	linear	✓
set	constant	constant	constant	✗
dict	constant	constant	constant	✗

Typical usages for each data type:

- Lists: Represent ordered collections of items, stacks, and queues [1]
- Sets: Represent collections of unique, unordered items
- Dicts: Represent registries, caches, mappings in general

[1] Also see the `deque` module for an efficient queue implementation

# Introduction to Python Control statements

# Outline

---

- If statements
- While loops
- For loops
  - List comprehensions
  - Looping patterns

# If statements

**if/elif/else** provides conditional execution of code blocks.

## IF STATEMENT FORMAT

```
if <condition>:  
    <statement 1>  
    <statement 2>  
elif <condition>:  
    <statements>  
else:  
    <statements>
```

## IF EXAMPLE

```
# a simple if statement  
>>> x = 10  
>>> if x > 0:  
...     Print 'Foo!'  
...     print 'x > 0'  
... elif x == 0:  
...     print 'x is 0'  
... else:  
...     print 'x is negative'  
... < hit return >  
Foo!  
x > 0
```

# Test Values

- zero, **None**, "", and empty objects are treated as False.
- All other objects are treated as True.

## EMPTY OBJECTS

```
# empty objects test as false
>>> x = []
>>> if x:
...     print 1
... else:
...     print 0
... < hit return >
0
```

It often pays to be explicit. If you are testing for an empty list, then test for:

```
if len(x):
    ...
    ...
```



This is clearer to future readers of your code. It also can avoid bugs where `x==None` may be passed in and unexpectedly go down this path.

# While loops

**while** loops iterate until a condition is met

```
while <condition>:  
    <statements>
```

## WHILE LOOP

```
# the condition tested is  
# whether lst is empty  
>>> lst = range(3)  
>>> while lst:  
...     print lst  
...     lst = lst[1:]  
... < hit return >  
[0, 1, 2]  
[1, 2]  
[2]
```

## BREAKING OUT OF A LOOP

```
# breaking from an infinite  
# loop  
>>> i = 0  
>>> while True:  
...     if i < 3:  
...         print i,  
...     else:  
...         break  
...     i = i + 1  
... < hit return >  
0 1 2
```

# For loops

for loops iterate over a sequence of objects

```
for <loop_var> in <sequence>:  
    <statements>
```

## TYPICAL SCENARIO

```
>>> for item in range(5):  
...     print item,  
... < hit return >  
0 1 2 3 4
```

```
# For a large range, xrange()  
# is faster and more memory  
# efficient.  
>>> for item in xrange(10**6):  
...     print item,  
... < hit return >  
0 1 2 3 4 5 6 7 8 9 10 11 ...
```

## LOOPING OVER A STRING

```
>>> for item in 'abcde':  
...     print item,  
... < hit return >  
a b c d e
```

## LOOPING OVER A SEQUENCE

```
>>> animals= ('dogs' , 'cats')  
>>> accum = ''  
>>> for animal in animals:  
...     accum += animal + ' '  
... < hit return >  
>>> print accum  
dogs cats
```

# List Comprehension

## LIST TRANSFORM WITH LOOP

```
# element by element transform of
# a list by applying an
# expression to each element
>>> a = [10,21,23,11,24]
>>> results=[]
>>> for val in a:
...     results.append(val+1)
>>> results
[11, 22, 24, 12, 25]
```

## FILTER-TRANSFORM WITH LOOP

```
# transform only elements that
# meet a criteria
>>> a = [10,21,23,11,24]
>>> results=[]
>>> for val in a:
...     if val>15:
...         results.append(val+1)
>>> results
[22, 24, 25]
```

## LIST COMPREHENSION

```
# list comprehensions provide
# a concise syntax for this sort
# of element by element
# transformation
>>> a = [10,21,23,11,24]
>>> [val+1 for val in a]
[11, 22, 24, 12, 25]
```

## LIST COMPREHENSION WITH FILTER

```
>>> a = [10,21,23,11,24]
>>> [val+1 for val in a if val>15]
[22, 24, 25]
```



Consider using a list comprehension whenever you need to transform one sequence to another.

# Looping Patterns

## MULTIPLE LOOP VARIABLES

```
# Looping through a sequence of
# tuples allows multiple
# variables to be assigned.
>>> pairs = [(0,'a'),(1,'b'),
...             (2,'c')]
>>> for index, value in pairs:
...     print index, value
0 a
1 b
2 c
```

## ENUMERATE

```
# enumerate -> index, item.
>>> y = ['a', 'b', 'c']
>>> for index, value in enumerate(y):
...     print index, value
0 a
1 b
2 c
```

## ZIP

```
# zip 2 or more sequences
# into a list of tuples
>>> x = [0, 1, 2]
>>> y = ['a', 'b', 'c']
>>> zip(x,y)
[(0,'a'), (1,'b'), (2,'c')]
>>> for index, value in zip(x,y):
...     print index, value
0 a
1 b
2 c
```

## REVERSED

```
>>> z = [(0,'a'),(1,'b'),(2,'c')]
for index, value in reversed(z):
...     print index, value
2 c
1 b
0 a
```

# Looping over a dictionary

```
>>> d = {'a':1, 'b':2, 'c':3}
```

## DEFAULT LOOPING (KEYS)

```
>>> for key in d:  
...     print key, d[key]  
  
a 1  
c 3  
b 2
```

## LOOPING OVER KEYS (EXPLICIT)

```
>>> for key in d.keys():  
...     print key, d[key]  
  
a 1  
c 3  
b 2
```

## LOOPING OVER VALUES

```
>>> for val in d.values():  
...     print val  
  
1  
3  
2
```

## LOOPING OVER ITEMS

```
>>> for key, val in d.items():  
...     print key, val  
  
a 1  
c 3  
b 2
```

# Introduction to Python

## Organizing code

# Outline

---

- Functions
- Modules
- Packages

# Functions

---

Functions are reusable snippets of code.

- Definition
- Positional and keyword arguments

# Anatomy of a function

DATA SCIENCE

The keyword **def** indicates the start of a function.

Function arguments are listed, separated by commas. They are passed by *assignment*.

```
def add(x, y):  
    """Add two numbers"""  
    return x + y
```

Indentation is used to indicate the contents of the function. It is *not* optional, but a part of the syntax.

A colon (:) terminates the function signature.

An optional **return** statement specifies the value returned from the function. If **return** is omitted, the function returns the special value **None**.

An optional **docstring** documents the function in a standard way for tools like ipython.

# Our new function in action

```
# We'll create our function  
# on the fly in the  
# interpreter.  
>>> def add(x,y):  
...     return x + y
```

```
# Test it out with numbers.  
>>> val_1 = 2  
>>> val_2 = 3  
>>> add(val_1,val_2)  
5
```

```
# How about strings?
```

```
>>> val_1 = 'foo'  
>>> val_2 = 'bar'  
>>> add(val_1,val_2)  
'foobar'
```

```
# Functions can be assigned  
# to variables.
```

```
>>> func = add  
>>> func(val_1, val_2)  
'foobar'
```

```
# How about numbers and strings?
```

```
>>> add('abc',1)
```

```
Traceback (innermost last):
```

```
File "<interactive input>", line 1, in ?
```

```
File "<interactive input>", line 2, in add
```

```
TypeError: cannot add type "int" to string
```

# Function Calling Conventions

DATA SCIENCE

## POSITIONAL ARGUMENTS

```
# The "standard" calling
# convention we know and love.

>>> def add(x, y):
...     return x + y

>>> add(2, 3)
5
```

## DEFAULT VALUES

```
# Arguments can be
# assigned default values.

>>> def quad(x,a=1,b=1,c=0):
...     return a*x**2 + b*x + c

# Use defaults for a, b and c.

>>> quad(2.0)
6.0
```

## KEYWORD ARGUMENTS

```
# specify argument names
>>> add(x=2, y=3)
5

# or even a mixture if you are
# careful with order
>>> add(2, y=3)
5
```

```
# Set b=3. Defaults for a & c.

>>> quad(2.0, b=3)
10.0

# Keyword arguments can be
# passed in out of order.

>>> quad(2.0, c=1, a=3, b=2)
17.0
```

# Function Calling Conventions

## VARIABLE NUMBER OF ARGS

```
# Pass in any number of
# arguments. Extra arguments
# are put in the tuple args.
>>> def foo(x, y, *args):
...     print x, y, args

>>> foo(2, 3, 'hello', 4)
2 3 ('hello', 4)
```

## VARIABLE KEYWORD ARGS

```
# Extra keyword arguments
# are put into the dict kw.
>>> def bar(x, y=1, **kw):
...     print x, y, kw

>>> bar(1, y=2, a=1, b=2)
1 2 {'a': 1, 'b': 2}
```

# Function Calling Conventions

## THE 'ANYTHING' SIGNATURE

```
# This signature takes any
# number of positional and
# keyword arguments.
>>> def foo(*args, **kw):
...     print args, kw

>>> foo(2, 3, x='hello', y=4)
(2, 3) {'x': 'hello', 'y': 4}
```

## MULTIPLE FUNCTION RETURNS

```
# To return multiple values
# from a function, we return
# a tuple containing those
# values. This is a common
# use of multiple (tuple)
# assignment.
>>> def functions(x):
...     y1 = x**2 + x
...     y2 = x**3 + x**2 + 2*x
...     return y1, y2

>>> a, b = functions(c)
```

# Expanding Function Arguments

## POSITIONAL ARGUMENT EXPANSION

```
>>> def add(x, y):  
...     return x + y
```

```
# '*' in a function call  
# converts a sequence into the  
# arguments to a function.  
>>> vars = [1,2]  
>>> add(*vars)  
3
```

## KEYWORD ARGUMENT EXPANSION

```
>>> def bar(x, y=1, **kw):  
...     print x, y, kw
```

```
# '**' expands a  
# dictionary into keyword  
# arguments for a function.  
vars = {'y':3, 'z':4}  
>>> bar(1, **vars)  
1, 3, {'z': 4}
```

# Evolution of a script

**Useful software often starts its life as a script.**

## EXPLORATORY SCRIPT

```
# Read data files into some structure.  
for file in files:  
    ...  
  
# Check for errors in data.  
if data > bad_value:  
    raise ValueError  
...  
  
# Execute one or more algorithms on the data.  
important_number = data * 2 + blah...  
...  
  
# Create a report about the results.  
print important_number  
...
```

# To A Function

Evolves to a function...

## ONE MEGA-FUNCTION

```
def display_data_report(files):
    # Read data files into some structure.
    for file in files:
        ...
    # Check for errors in data.
    if data > bad_value:
        raise ValueError
    ...
    # Execute one or more algorithms on the data.
    important_number = data * 2 + blah
    ...
    # Create a report about the results.
    print important_number
    ...
```

# Evolution Stops

## And Stops...

There are some short term benefits to this.

### MEGA-FUNCTION BENEFITS

- Easy (quick) to create from original script.
- Easy to read and modify during construction.
  - All the code is “in one place.”
  - Access to all variables at any time.  
(Global namespaces are nice that way.)
- Achieves some very minimal re-use.

# Evolution to a library

Long term benefits come from continuing to “refactor” this function until there is “one idea per function.”

## LOW LEVEL FUNCTION LIBRARY

```
def data_from_files(files):
    # Read data files into a structure.
    for file in files:
        ...

def check_for_errors(data):
    # Check for errors in data.
    if data > bad_value:
        raise ValueError
    ...

def calc_important_number(data):
    # Execute one or more algorithms.
    important_number = data * 2
    ...

def create_report(data, calc_data):
    # Create a report about results.
    print important_number
    ...
```

## DRIVER FUNCTIONS

```
def display_data_report(files):
    """
    "Driver" function that calls
    the low level library functions.
    """

    data = data_from_files(files)
    check_for_errors(data)
    res = calc_important_number(data)
    create_report(data, res)
```

# “One Idea Per Function” Benefits

Smaller, less complex snippets of code

- are easier for others (and you) to read in the future,
- have more potential for reuse,
- make it easier to modify behavior (decoupling!), and
- are easier to test.

# Don't Repeat Yourself

DATA SCIENCE

## DUPLICATED CODE

```
instrument1_prices = lookup_price(instrument1, start_date, stop_date)
instrument1_price_avg = mean(instrument1_prices)
print_summary(instrument1, instrument1_prices, instrument1_price_avg)

instrument2_prices = lookup_price(instrument2, start_date, stop_date)
instrument2_price_avg = mean(instrument2_prices)
print_summary(instrument2, instrument2_prices, instrument2_price_avg)
```

## DON'T REPEAT YOURSELF

```
# Refactor code so duplicated lines are in a function.
def summarize_price_info(instrument, start_date, stop_date):
    instrument_prices = lookup_price(instrument, start_date, stop_date)
    instrument_price_avg = mean(instrument_prices)
    print_summary(instrument, instrument_prices, instrument_price_avg)

# Now call the function for the two different instruments.
summarize_price_info(instrument1, start_date, stop_date)
summarize_price_info(instrument2, start_date, stop_date)
```

# Modules

# Modules and packages

Modules and packages are Python’s “libraries”, i.e. a collection of constants, functions, and classes.

# Importing a module

## BASIC IMPORTS

```
# The most basic import
>>> import numpy
>>> numpy.pi
3.141592653589793

# Use an 'alias'
>>> import numpy as np
>>> np.pi
3.141592653589793
```

## IMPORTING SPECIFIC SYMBOLS

```
# Select specific names to
# bring into the local
# namespace.
>>> from numpy import add, pi
>>> pi
3.141592653589793
>>> add(2, 3)
```

5

## IMPORTING \*EVERYTHING\*

```
# Pull *everything* into the
# local namespace.
>>> from numpy import *
>>> pi
3.141592653589793
>>> add(3, 4.5)
7.5
```

## MODULES ARE .PY FILES

Modules are just .py files.

```
# my_tools.py
def greetings():
    return "Hello everyone"
```

```
>>> import my_tools
>>> my_tools.greetings()
'Hello everyone'
```

# Modules

A Python file can be used as a script, or as a module, or both.

## EX.PY

```
# An example module that can
# be run as a script.

PI = 3.1416

def sum(lst):
    """ Sum the values in a
        list.
    """
    tot = 0
    for value in lst:
        tot = tot + value
    return tot
```

```
def add(x,y):
    " Add two values."
    a = x + y
    return a

def test():
    w = [0,1,2,3]
    assert( sum(w) == 6)
    print 'test passed'

# This code runs only if this
# module is the main program.
if __name__ == '__main__':
    test()
```

# Packages

## PACKAGES

Often a library will contain several modules. These are organized as a hierarchical directory structure, and imported using "dotted module names". The first and the intermediate names (if any) are called "packages".

Example:

```
>>> from email.utils import parseaddr  
>>> from email import utils  
>>> utils.parseaddr('John Doe <jdoe@company.com>')  
('John Doe', 'jdoe@company.com')
```

## PACKAGES ARE DIRECTORIES

```
foo/  
    __init__.py  
    bar.py (defines func)  
    baz.py (defines zap)
```

The file `__init__.py` indicates that `foo` is a package. It often is an empty file.

`utils` is a *module* in the package `email` .

# Setting up PYTHONPATH

PYTHONPATH is an environment variable (or set of registry entries on Windows) that lists the directories Python searches for modules.

## WINDOWS

- Right-click on My Computer
- Click Properties
- Click Advanced Tab
- Click Environment Variables Button at the bottom of the Advanced Tab
  - Click New to create PYTHONPATH or
  - Click Edit to change existing PYTHONPATH
- Changes take effect in the next Command Prompt or IPython session.

## UNIX: .cshrc

```
!! NOTE: The following should !!
!! all be on one line !!

setenv PYTHONPATH
$PYTHONPATH:$HOME/your_modules
```

## UNIX: .bashrc

```
PYTHONPATH=$PYTHONPATH:$HOME/your
_modules
export PYTHONPATH
```

# Naming Packages and Modules

## MODULE NAMES

```
# Module names should be lower case  
# with underscores.
```

```
# Yes  
foo_bar.py
```

```
# No  
FooBar.py
```

## PACKAGE NAMES

```
# Package directories should be all  
# lower case alpha-numeric characters.  
# Avoid underscores unless absolutely  
# necessary.
```

```
# Yes  
packagename
```

```
# No  
PackageName  
package_name
```

# Common Directory Structure

## PACKAGE/MODULE/TESTS LAYOUT

Example directory structure for Python libraries.

C:/

```
python_library/
    yourpackage/
        __init__.py
        some_module.py
        another_module.py
    tests/
        test_some_module.py
        test_another_module.py
```



The tests for module have  
are named similarly (test\_  
prefix) but live “one level  
below” in a tests directory.

# Standard Modules

Python has a large library of standard modules ("batteries included"):

**re** - regular expressions

**copy** – shallow and deep copy operations

**datetime** - time and date objects

**math, cmath** - real and complex math

**decimal, fractions** - arbitrary precision decimal and rational number objects

**os, os.path, shutil** - filesystem operations

**sqlite3** - internal SQLite database

**gzip, bz2, zipfile, tarfile** – compression and archiving formats

**csv, netrc** – file format handling

**xml** – various modules for handling XML

**htmllib** – an HTML parser

**httpplib, ftplib, poplib, socket, etc.** – modules for standard internet protocols

**cmd** – support for command interpreters

**pdb** – Python interactive debugger

**profile, cProfile, timeit** – Python profilers

**collections, heapq, bisect** – standard CS algorithms and data structures

**mmap** – memory-mapped files

**threading, Queue** – threading support

**multiprocessing** – process based ‘threading’

**subprocess** – executing external commands

**pickle, cPickle** – object serialization

**struct** – interpret bytes as packed binary data

**urllib2** – open and read from URLs

and many more... To see the content of one:

```
>>> dir(module_name)
```

# Selections from the Python Standard Library

# datetime – Dates and Times

```
>>> from datetime import date, time
```

## DATE OBJECT

```
# date(year, month, day)
# date in Gregorian calendar,
# assuming its permanence
>>> d1 = date(2007, 9, 25)
>>> d2 = date(2008, 9, 25)
>>> d1.strftime('%A %m/%d/%y')
'Tuesday 09/25/07'

# difference is timedelta
>>> print d2 - d1
366 days, 0:00:00
>>> (d2-d1).days
366
>>> print date.today()
2008-09-24
```

## TIME OBJECT

```
# time(hour, min, sec, us)
# local time of day
# always 24 hrs per day
>>> t1 = time(15, 38)
>>> t2 = time(18)
>>> t1.strftime('%I:%M %p')
'03:38 PM'

# difference is not supported
>>> print t2 - t1
Traceback ...
TypeError: unsupported operand ...
# use datetime objects for
# difference operation.
```

# datetime – Dates and Times

DATA SCIENCE

```
>>> from datetime import datetime, timedelta
```

## DATETIME OBJECT

```
# datetime(year, month, day,
          hr, min, sec, us)
# combination of date and time
>>> d1 = datetime.now()
>>> print d1
2008-09-24 14:20:30.978207
>>> d2 = d1 + timedelta(30)
>>> d2.strftime('%A %m/%d/%y')
'Friday 10/24/08'

# creating datetime from
# a format string
>>> datetime.strptime('2/10/01',
                      '%m/%d/%y')
datetime.datetime(2001, 2, 10, 0, 0)
```

## DATETIME FORMAT STRING

Directive	Meaning
%a (%A)	Abbrev. (full) weekday name.
%w	Weekday number [0 (Sun), 6]
%b (%B)	Abbrev. (full) month name
%d	Day of month [01, 31]
%H (%I)	Hour [00, 23] ([01, 12])
%j	Day of the year [001, 366]
%m	Month [01, 12]
%M	Minute [0, 59]
%p	AM or PM
%S	Second [00, 61]
%U (%W)	Week number of the year [00, 53] Sunday (Monday) as first day of week.
%Y (%Y)	Year without (with) century [00, 99]

# sys module

```
>>> import sys
```

Some frequently used attributes and functions—see the reference manual for complete details.

## Command Line Arguments

`sys.argv`

List of command line arguments.

`sys.argv[0]` is the name of the python script.

*Example:*

```
# File: print_args.py
import sys
print sys.argv
```

```
$ python print_args.py 1 foo
['print_args.py', '1', 'foo']
```

## Exception Information

`sys.exc_info()`

Returns a tuple (type, value, traceback)

`sys.exc_clear()`

Clear all exception information.

```
>>> try:
...     x = 1/0
... except Exception:
...     print sys.exc_info()
...
(<type 'exceptions.ZeroDivisionError'>,
ZeroDivisionError('integer division or
modulo by zero',), <traceback object at
0x9a8c8>)
>>>
```

# sys module

## Standard File Objects

`sys.stdin`

`sys.stdout`

`sys.stderr`

The interpreter's standard input, output and error streams.

`sys.__stdin__`

`sys.__stdout__`

`sys.__stderr__`

The original values of `sys.stdin`, `sys.stdout` and `sys.stderr` at the start of the program.

## Exit

`sys.exit(arg)`

Exit from Python. `arg` is optional. It can be an integer giving the exit status (defaults to zero). If not an integer, `None` is equivalent to passing 0, and any other argument is printed to `sys.stderr` and the exit status is 1.

## Python's module search path

`sys.path`

A list of strings that specifies the interpreter's search path for modules.

A program is free to modify this list dynamically.

# sys module

## Platform Information

### `sys.platform`

A string containing the platform identifier.

Windows: 'win32'

Mac OSX: 'darwin'

Linux: 'linux2'

### `sys.getwindowsversion()`

Return a tuple that describes the version of Windows currently running: *major*, *minor*, *build*, *platform*, and *service\_pack*. (More is included in Python 2.7.)

```
>>> sys.platform
```

```
'win32'
```

```
>>> sys.getwindowsversion()
```

```
(5, 1, 2600, 2, 'Service Pack 3')
```

See also the [platform](#) module in the standard library.

## Python Version

### `sys.version`

A string containing information about the Python version.

### `sys.version_info`

A tuple containing information about the Python version: *major*, *minor*, *micro*, *releaselevel* and *serial*.

```
>>> sys.version
```

```
'2.6.5 |EPD 6.2-2 (32-bit)|\n(r265:79063, May 7 2010, 13:28:19)\n[MSC v.1500 32 bit (Intel)]'
```

```
>>> sys.version_info
```

```
(2, 6, 5, 'final', 0)
```

# os module

```
>>> import os
```

## Path Operations

`os.remove(path)` `os.unlink(path)`

Remove a file from disk (file can be either the full path or a file from the current working directory will be removed).

`os.chdir(path)`

Change the current working directory to the provided path.

`os.getcwd()`

Return the current working directory.

`os.listdir(path)`

Return a list of strings containing all the files in the given path (does not include '.' or '..' in the listing).

## Separation Constants

`os.linesep` (e.g. '\n' or '\r\n')

Line separator in text mode.

`os.sep` (e.g. '/' or '\')

Path separator on file system.

`os.pathsep` (e.g. ':' or ';')

Search path separator (*i.e.* in environment variables).

## Others

`os.environ`

Dictionary of all environment variables

`os.urandom(len)`

String of random bytes

`os.error`

Error object

# os.path

## os.path – tests

`os.path.isfile(path)`

Test whether a path is a regular file.

`os.path.isdir(path)`

Test whether a path refers to an existing directory.

`os.path.exists(path)`

Test whether a path exists.

`os.path.isabs(path)`

Test whether a path is absolute.

## os.path – split and join

`os.path.split(path)`

Split a pathname. Returns the tuple (head, tail).

`os.path.join(a, *p)`

Join two or more path components.

## Others

`os.path.abspath(path)`

Return an absolute path.

`os.path.dirname(path)`

Return the directory component of a pathname.

`os.path.basename(path)`

Return the final component of a pathname.

`os.path.splitext(path)`

Split the extension from a pathname.

Returns (root, ext).

`os.path.expanduser(path)`

Expand ~ and ~user. If user or \$HOME is unknown, do nothing.

# DAY 3

# Core libraries for data processing

# Outline

---

- NumPy
- matplotlib
- SciPy
- Pandas

# NumPy

## The standard numerical library for Python

# NumPy: array and array functions

## NumPy [ Vectorized Array Data]

fft

random

linalg

NDArray  
multi-dimensional  
array object

UFunc  
fast array  
math operations

# NumPy arrays

- The array data structure
- Defining arrays
- Indexing and slicing
- Creating arrays
- Array calculations
- Advanced NumPy

# Getting Started

## IMPORT NUMPY

```
In [1]: from numpy import *
```

```
In [2]: __version__
```

```
Out[2]: 1.8.1
```

or

```
In [1]: from numpy import \
array, ...
```

Often at the command line, it is handy to import everything from NumPy into the command shell.

However, if you are writing scripts, it is easier for others to read and debug in the future if you use explicit imports.

## USING IPYTHON -PYLAB

```
C:\> ipython --pylab
```

```
In [1]: array([1,2,3])
```

```
Out[1]: array([1, 2, 3])
```

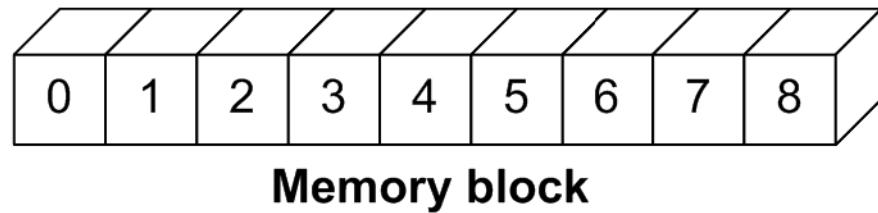
IPython has a ‘pylab’ mode where it imports all of NumPy and Matplotlib, into the namespace for you as a convenience. It also enables threading for showing plots.



While IPython is used for all the demos, ‘>>>’ is used on future slides instead of ‘In [1]’ to save space.

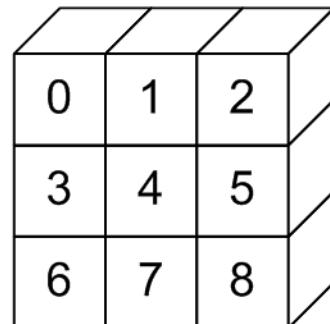
# The array data structure

# Array Data Structure



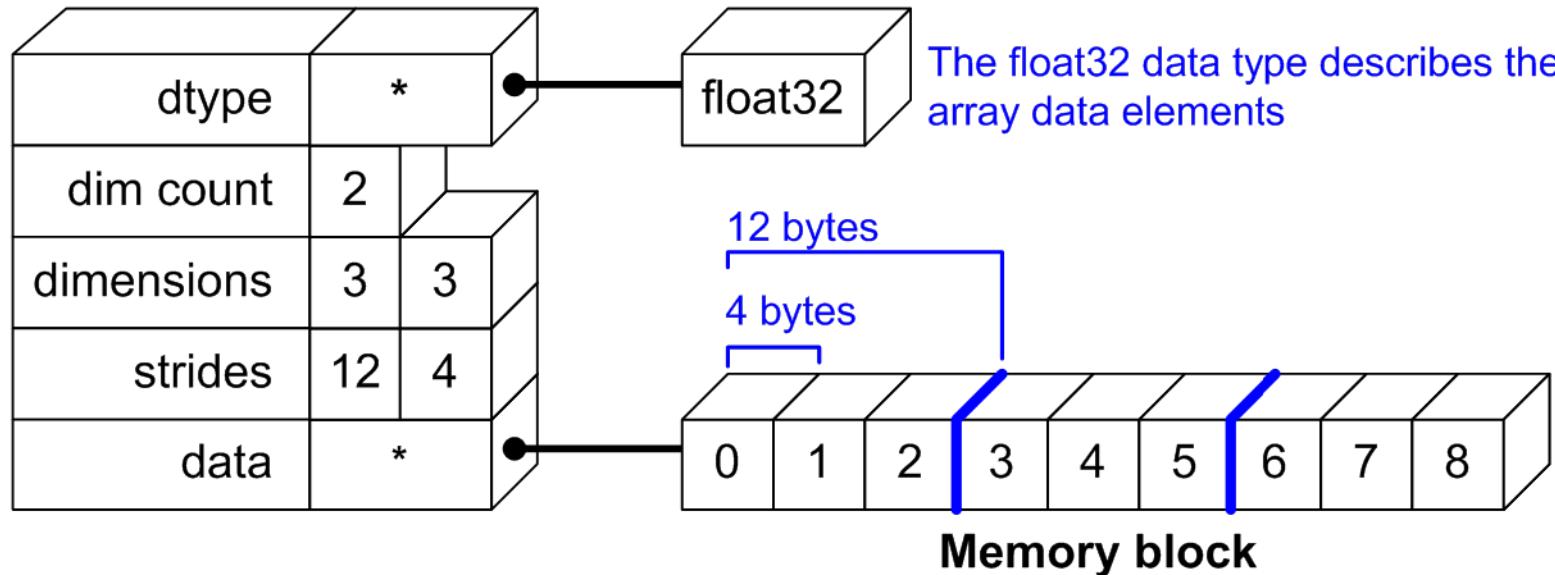
---

**Python View:**

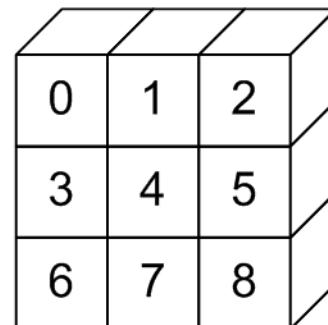


# Array Data Structure

## NDArray Data Structure



Python View:



# Operations on the array structure

Operations that only affect the array structure, not the data, can be executed without copying memory.

# Transpose

## TRANSPOSE

```
>>> a = array([[0,1,2],  
...                 [3,4,5]])  
>>> a.shape  
(2, 3)  
# Transpose swaps the order  
# of axes.  
>>> a.T  
array([[0, 3],  
       [1, 4],  
       [2, 5]])  
>>> a.T.shape  
(3, 2)
```

## TRANSPOSE RETURNS VIEWS

```
# Transpose does not move  
# values around in memory. It  
# only changes the order of  
# "strides" in the array  
>>> a.strides  
(12, 4)  
  
>>> a.T.strides  
(4, 12)
```

# Reshaping Arrays

## RESHAPE

```
>>> a = array([[0,1,2],  
...             [3,4,5]])  
  
# Return a new array with a  
# different shape (a view  
# where possible)  
>>> a.reshape(3,2)  
array([[0, 1],  
      [2, 3],  
      [4, 5]])  
  
# Reshape cannot change the  
# number of elements in an  
# array  
>>> a.reshape(4,2)  
ValueError: total size of new  
array must be unchanged
```

## SHAPE

```
>>> a = arange(6)  
>>> a  
array([0, 1, 2, 3, 4, 5])  
>>> a.shape  
(6,)  
  
# Reshape array in-place to  
# 2x3  
>>> a.shape = (2,3)  
>>> a  
array([[0, 1, 2],  
      [3, 4, 5]])
```

# Flattening Arrays

## FLATTEN (SAFE)

`a.flatten()` converts a multi-dimensional array into a 1-D array. The new array is a *copy* of the original data.

```
# Create a 2D array
>>> a = array([[0,1],
              [2,3]])
```

```
# Flatten out elements to 1D
>>> b = a.flatten()
>>> b
array([0,1,2,3])
```

```
# Changing b does not change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[0, 1],
       [2, 3]])
```

no change

## RAVEL (EFFICIENT)

`a.ravel()` is the same as `a.flatten()`, but returns a *reference* (or *view*) of the array if possible (i.e., the memory is contiguous). Otherwise the new array copies the data.

```
# Flatten out elements to 1-D
>>> b = a.ravel()
>>> b
array([0,1,2,3])
```

```
# Changing b does change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[10, 1],
       [2, 3]])
```

changed!

# Matplotlib Basics

[home](#) | [search](#) | [examples](#) | [gallery](#) | [docs](#) »

[modules](#) | [index](#)

## intro

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and [ipython](#) shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits.

matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the [screenshots](#), [thumbnail](#) gallery, and [examples](#) directory

For example, using "ipython -pylab" to provide an interactive environment, to generate 10,000 gaussian random numbers and plot a histogram with 100 bins, you simply need to type

```
x = randn(10000)
hist(x, 100)
```

For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users. The pylab mode provides all of the  [pyplot](#) plotting functions listed below, as well as non-plotting functions from  [numpy](#) and  [matplotlib.mlab](#).

## plotting commands

Function	Description
<a href="#">acorr</a>	plot the autocorrelation function

### News

Please [donate](#) to support matplotlib development.

matplotlib 1.0.1 is available for [download](#). See [what's new](#) and tips on [installing](#)

Sandro Tosi has a new book [Matplotlib for python developers](#) also at [amazon](#).

Build websites like matplotlib's, with [sphinx](#) and extensions for mpl plots, math, inheritance diagrams -- try the [sampledoc](#) tutorial.

### Videos

Watch the [SciPy 2009 intro](#) and [advanced](#) matplotlib tutorials

Watch a [talk](#) about matplotlib presented at [NIPS 08](#) [Workshop MLOSS](#) and one presented at [ChiPy](#).

### Toolkits

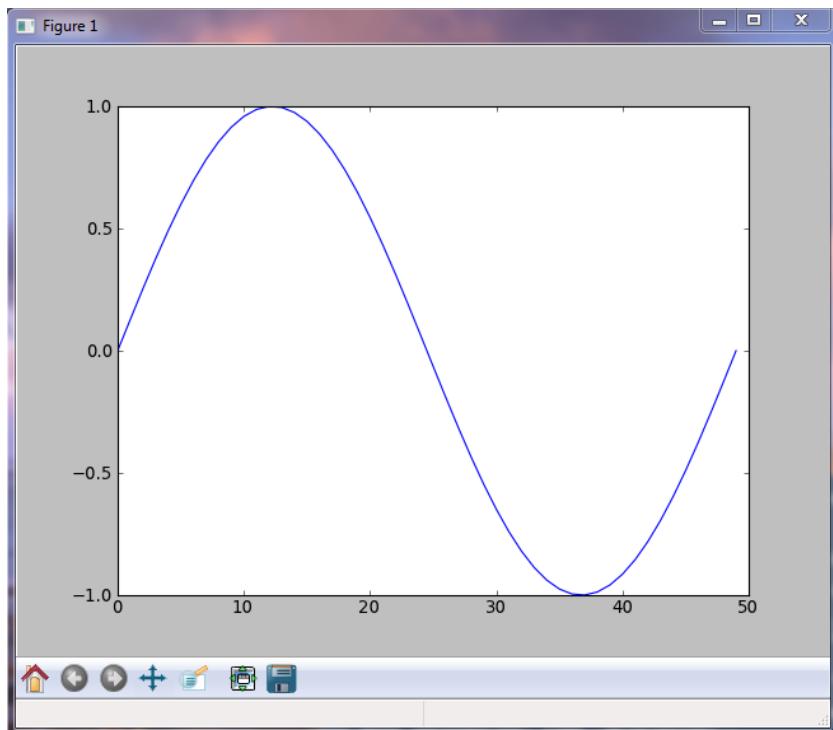
There are several matplotlib addon [toolkits](#), including the projection and mapping toolkit

145

# Line Plots

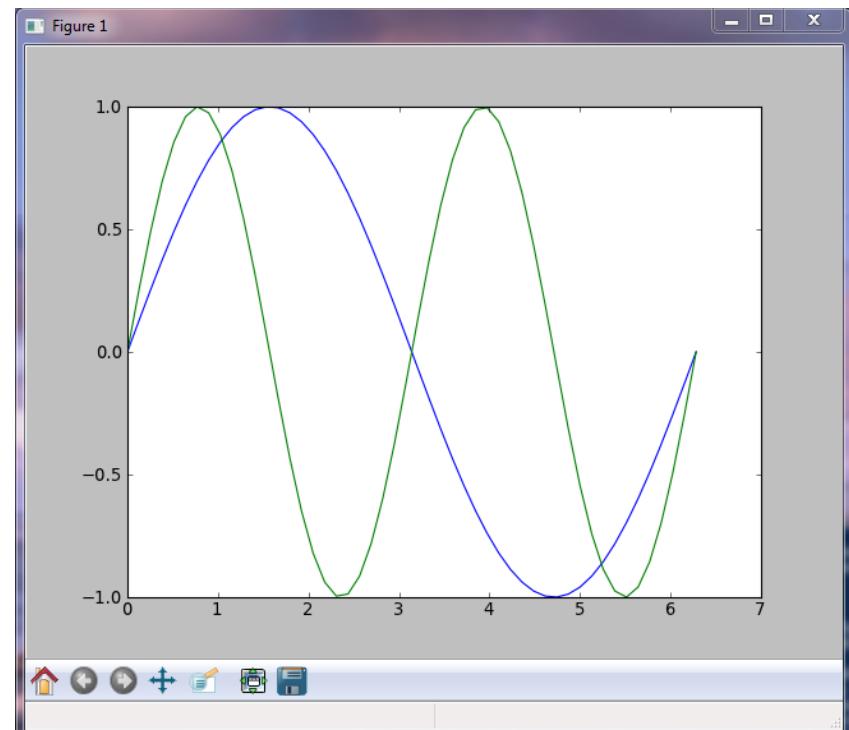
## PLOT AGAINST INDICES

```
>>> x = linspace(0,2*pi,50)  
>>> plot(sin(x))
```

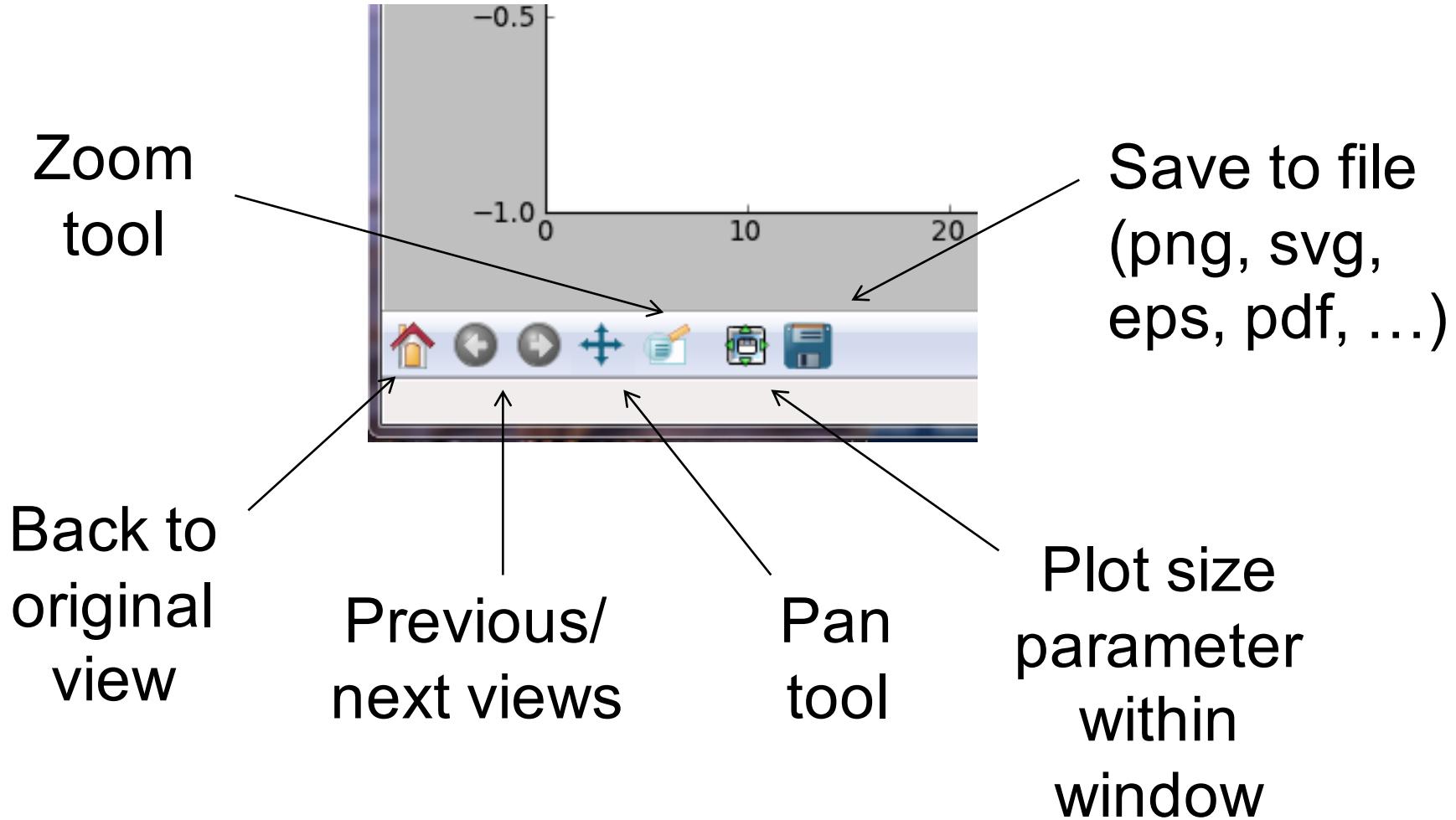


## MULTIPLE DATA SETS

```
>>> plot(x, sin(x),  
...         x, sin(2*x))
```



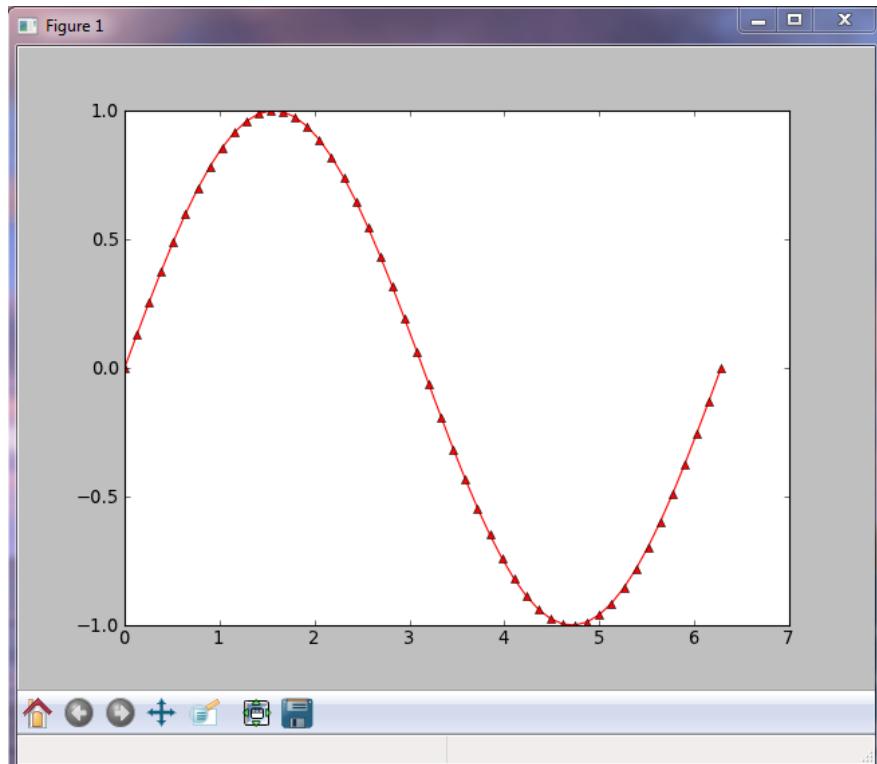
# Matplotlib Menu Bar



# Line Plots

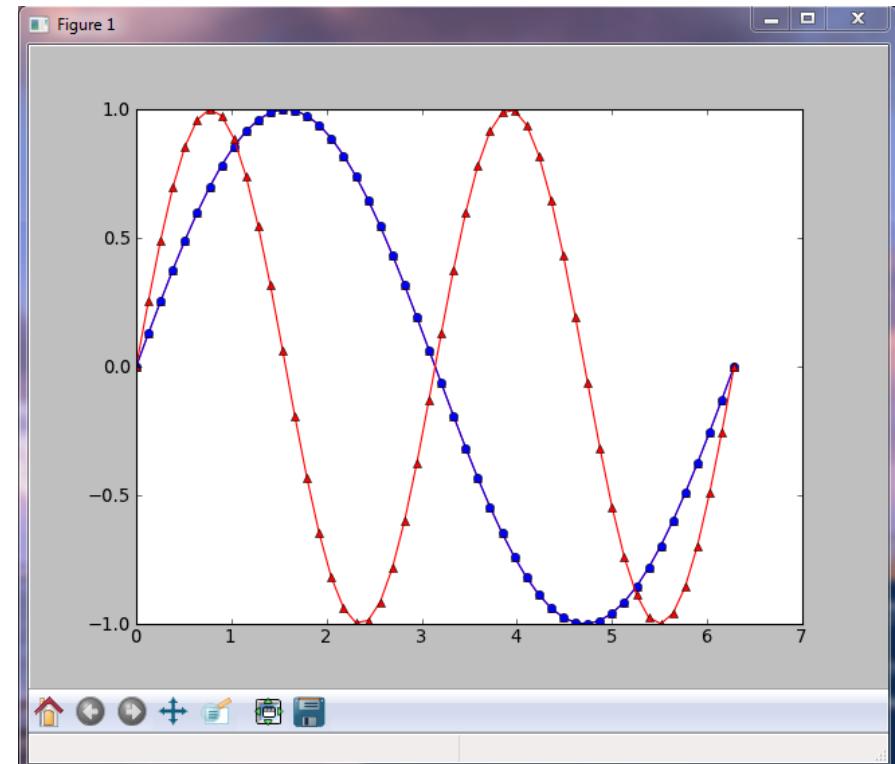
## LINE FORMATTING

```
# red, dot-dash, triangles  
>>> plot(x, sin(x), 'r-^')
```



## MULTIPLE PLOT GROUPS

```
>>> plot(x, sin(x), 'b-o',  
...       x, sin(2*x), 'r-^')
```

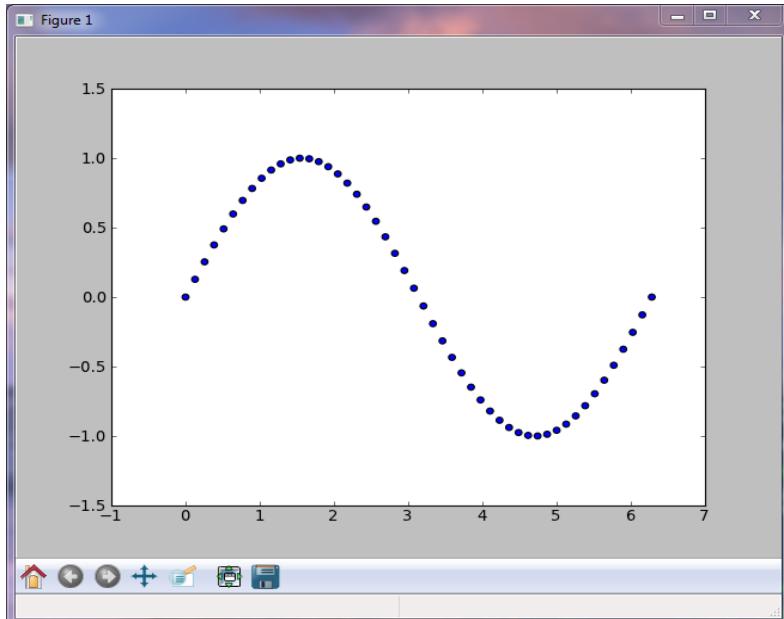


# Scatter Plots

DATA SCIENCE

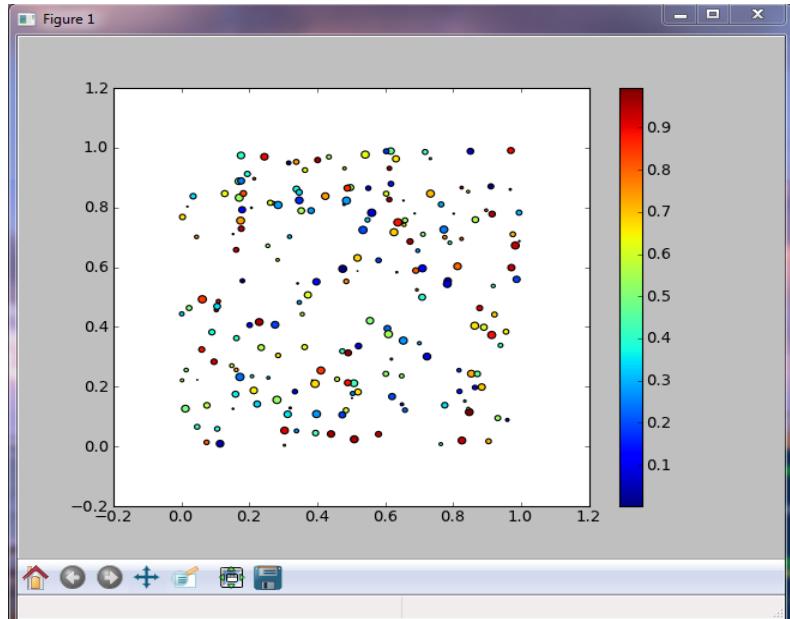
## SIMPLE SCATTER PLOT

```
>>> x = linspace(0,2*pi,50)
>>> y = sin(x)
>>> scatter(x, y)
```



## COLORMAPPED SCATTER

```
# marker size/color set with data
>>> x = rand(200)
>>> y = rand(200)
>>> size = rand(200)*30
>>> color = rand(200)
>>> scatter(x, y, size, color)
>>> colorbar()
```

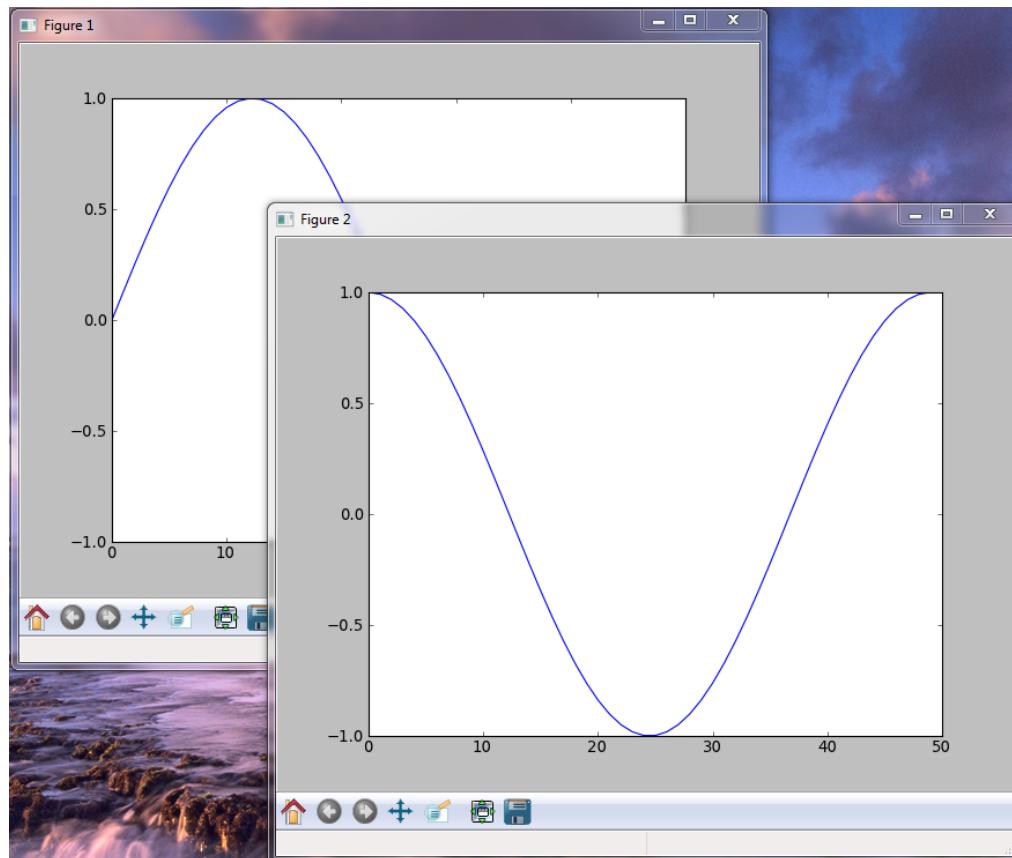


# Multiple Figures

```
>>> t = linspace(0,2*pi,50)
>>> x = sin(t)
>>> y = cos(t)

# Now create a figure
>>> figure()
>>> plot(x)

# Now create a new figure.
>>> figure()
>>> plot(y)
```



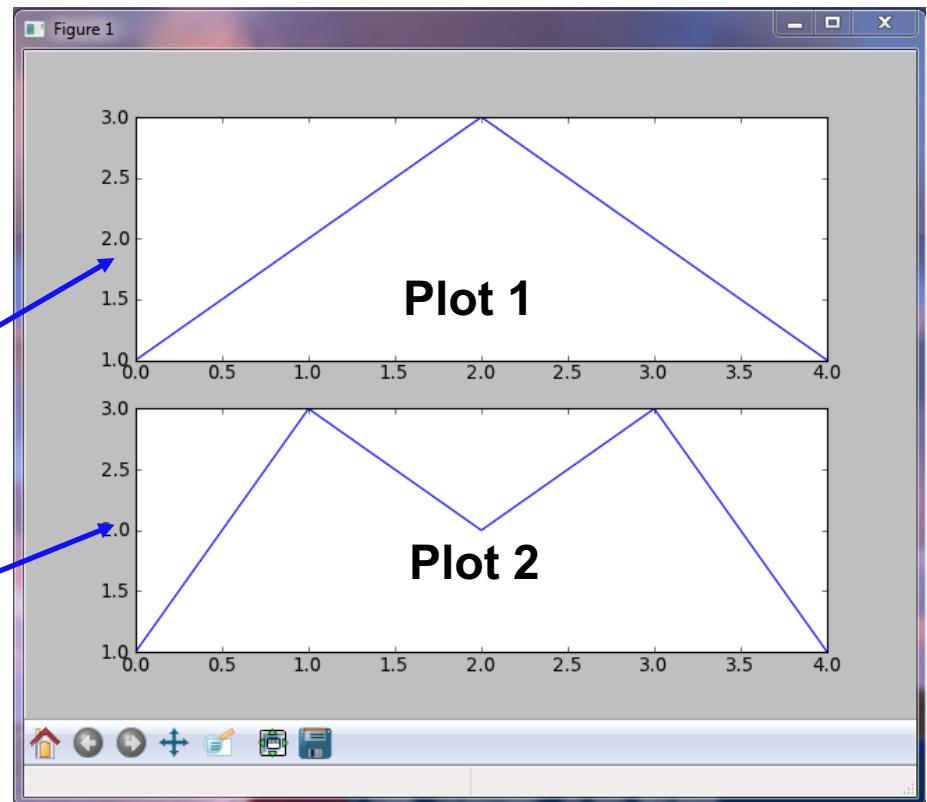
# Multiple Plots Using subplot

```
>>> x = array([1,2,3,2,1])  
>>> y = array([1,3,2,3,1])
```

```
# To divide the plotting area
```

```
    columns  
    |  
>>> subplot(2, 1, 1)  
>>> plot(x) |  
    rows      |  
            active plot
```

```
# Now activate a new plot  
# area.  
>>> subplot(2, 1, 2)  
>>> plot(y)
```



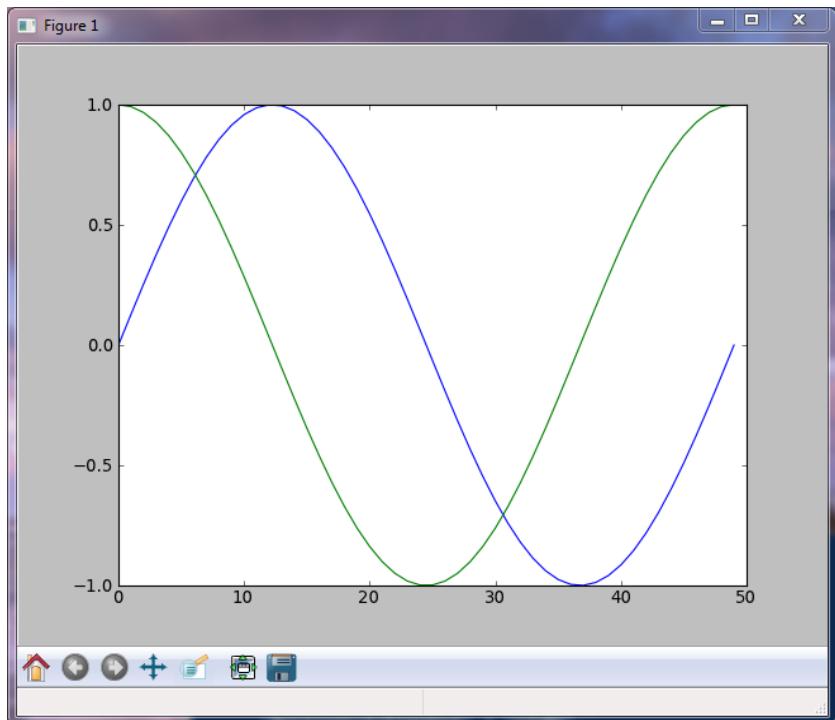
If this is used in a python script, a call to the function `show()` is required.

# Adding Lines to a Plot

DATA SCIENCE

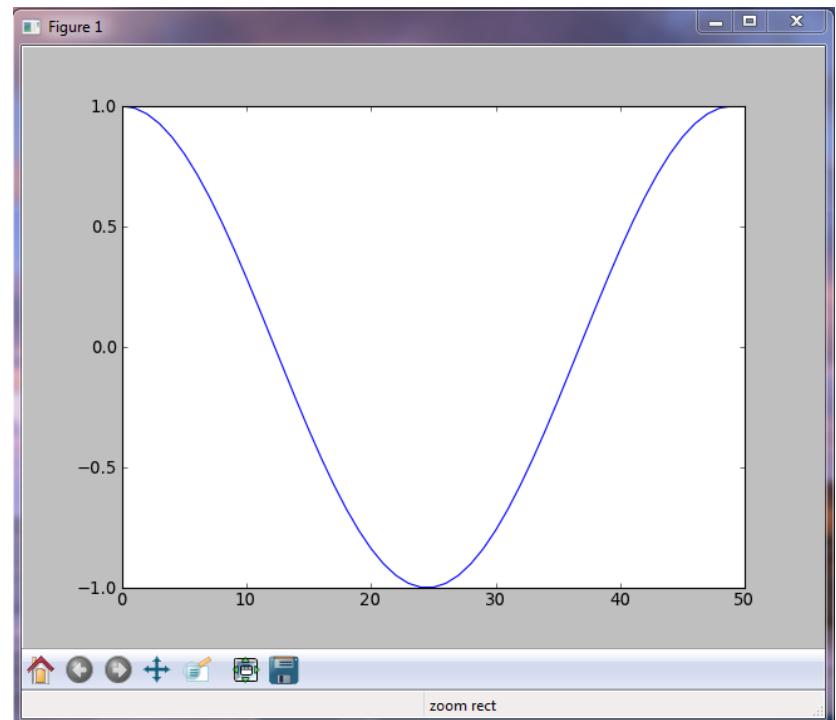
## MULTIPLE PLOTS

```
# By default, previous lines  
# are "held" on a plot.  
  
>>> plot(sin(x))  
>>> plot(cos(x))
```



## ERASING OLD PLOTS

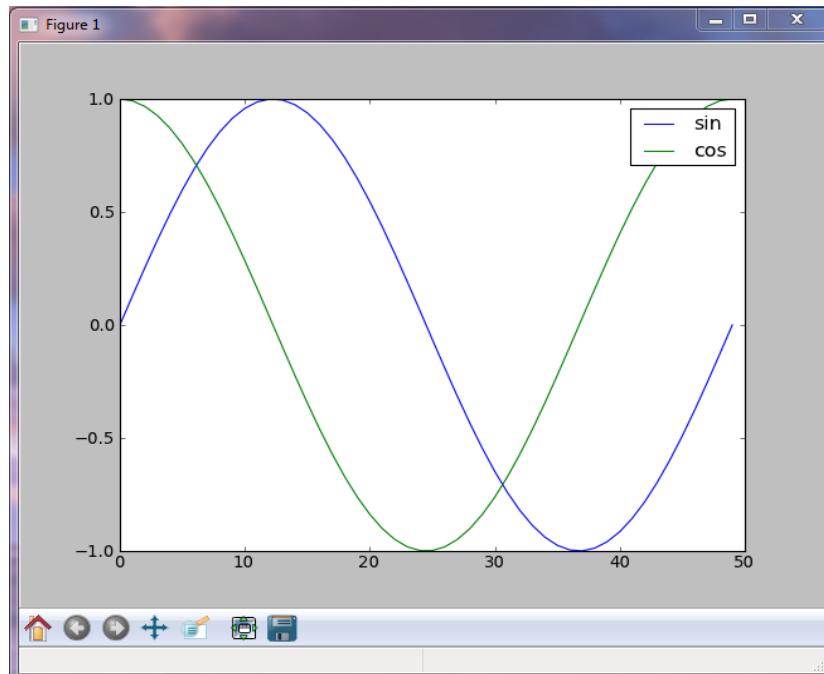
```
# Set hold(False) to erase  
# old lines  
  
>>> plot(sin(x))  
>>> hold(False)  
>>> plot(cos(x))
```



# Legend

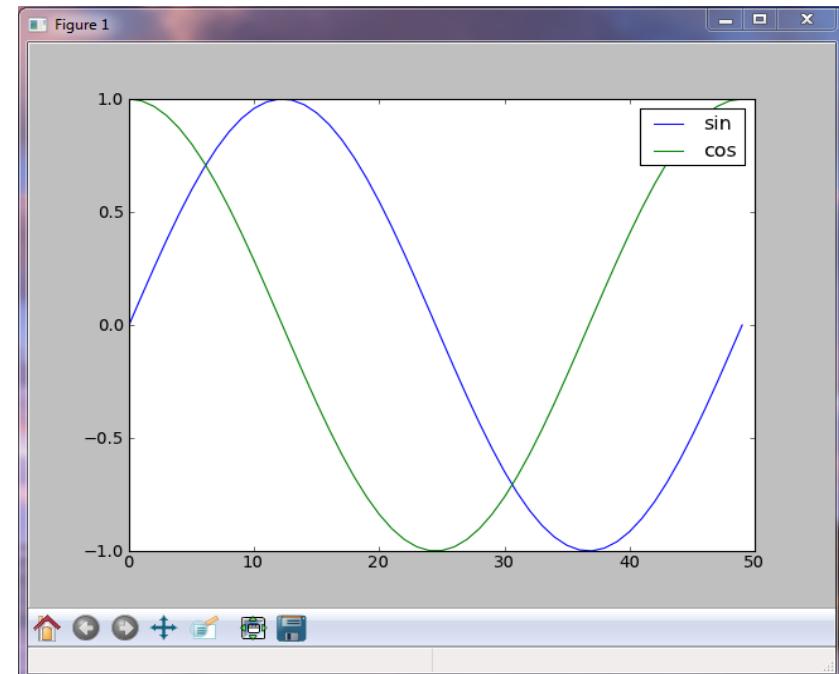
## LEGEND LABELS WITH PLOT

```
# Add labels in plot command.
>>> plot(sin(x), label='sin')
>>> plot(cos(x), label='cos')
>>> legend()
```



## LABELING WITH LEGEND

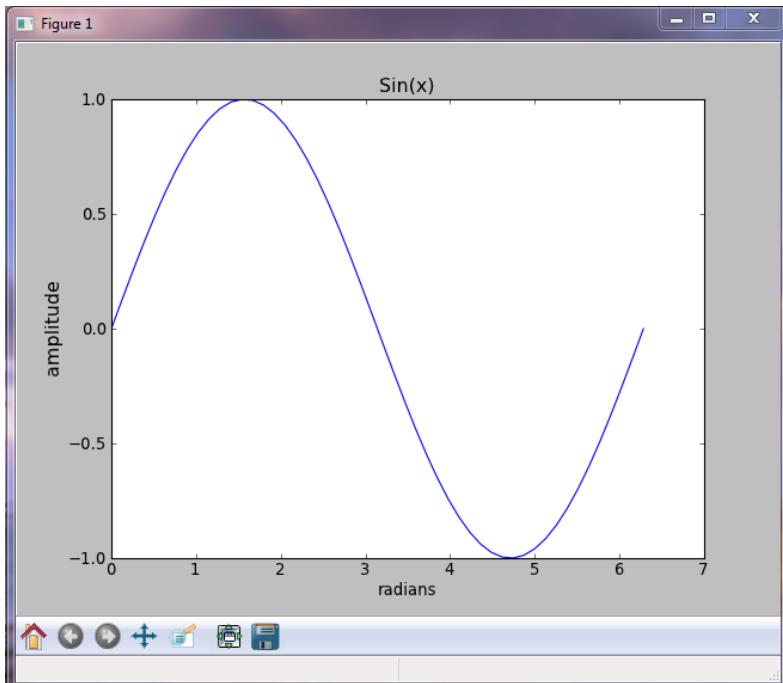
```
# Or as a list in legend().
>>> plot(sin(x))
>>> plot(cos(x))
>>> legend(['sin', 'cos'])
```



# Titles and Grid

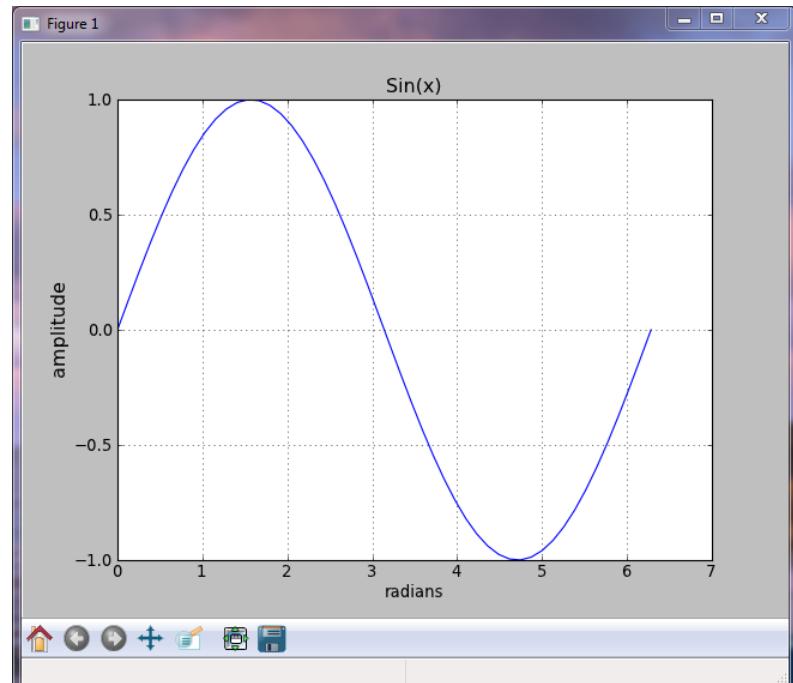
## TITLES AND AXIS LABELS

```
>>> plot(x, sin(x))  
>>> xlabel('radians')  
# Keywords set text properties.  
>>> ylabel('amplitude',  
...           fontsize='large')  
>>> title('Sin(x)')
```



## PLOT GRID

```
# Display gridlines in plot  
>>> grid()
```

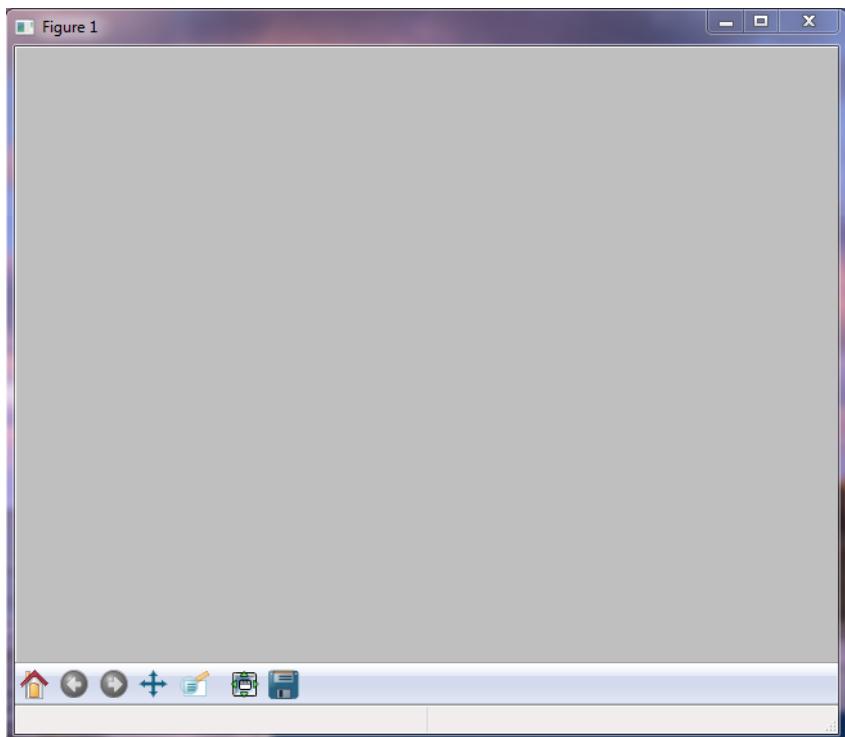


# Clearing and Closing Plots

DATA SCIENCE

## CLEARING A FIGURE

```
>>> plot(x, sin(x))  
# clf will clear the current  
# plot (figure).  
>>> clf()
```



## CLOSING PLOT WINDOWS

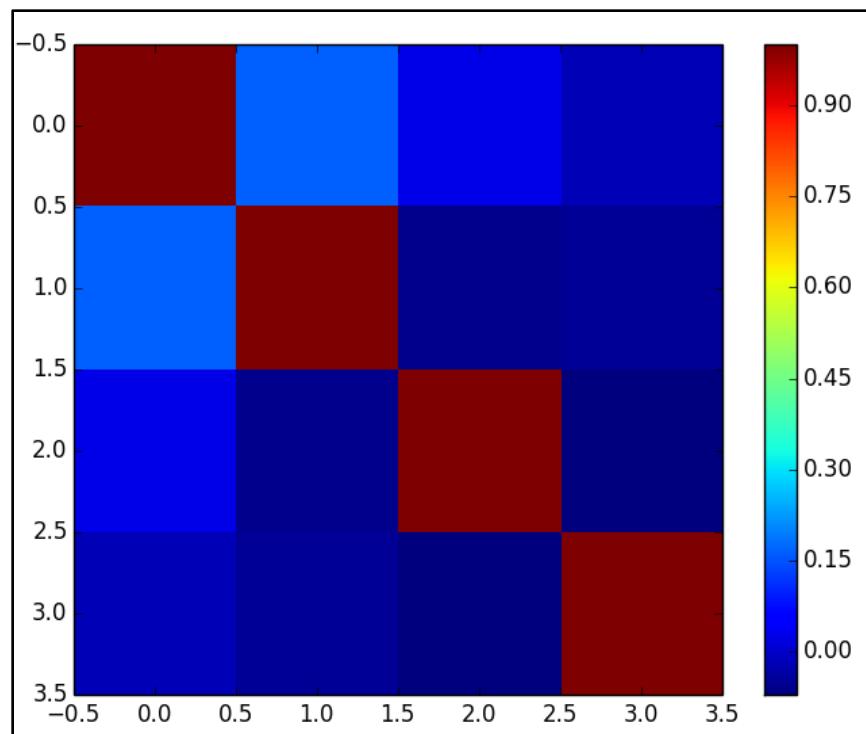
```
# close() will close the  
# currently active plot window.  
>>> close()  
  
# close('all') closes all the  
# plot windows.  
>>> close('all')
```

# Display Images (or plot surface)

## IMAGE PLOTS

```
>>> # Create some data
>>> e1 = rand(100)
>>> e2 = rand(100)*2
>>> e3 = rand(100)*10
>>> e4 = rand(100)*100
>>> corrmatrix = \
...     corrcoef([e1, e2, e3, e4])

>>> # Plot corr matrix as image
>>> imshow(corrmatrix,
... interpolation='nearest')
>>> colorbar()
```



# Plotting from Scripts

DATA SCIENCE

## INTERACTIVE MODE

```
# In IPython, plots show up
# as soon as a plot command
# is called.
>>> figure()
>>> plot(sin(x))
>>> figure()
>>> plot(cos(x))
```

## NON-INTERACTIVE MODE

```
# script.py
# In a script, you must call
# the show() command to display
# plots. Call it at the end of
# all your plot commands for
# best performance.

figure()
plot(sin(x))
figure()
plot(cos(x))

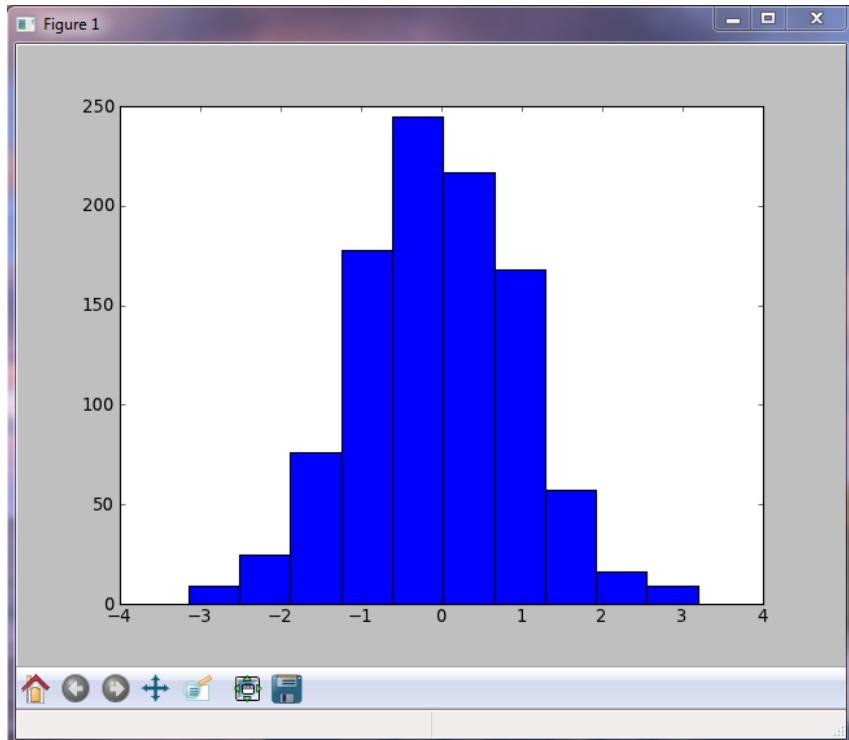
# Plots will not appear until
# this command is issued.

show()
```

# Histograms

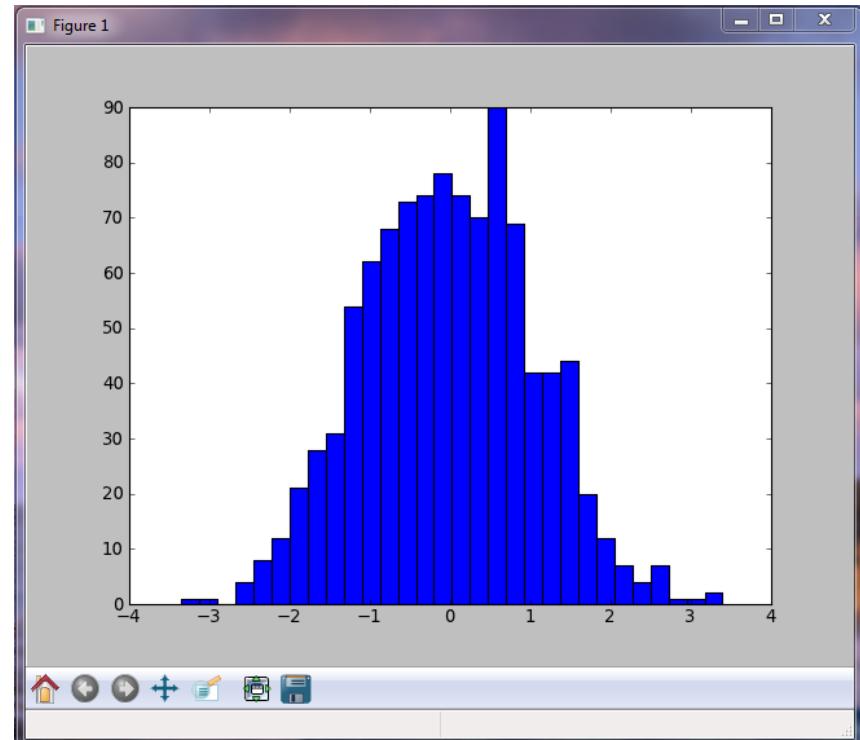
## HISTOGRAM

```
# plot histogram  
# defaults to 10 bins  
>>> hist(randn(1000))
```



## HISTOGRAM 2

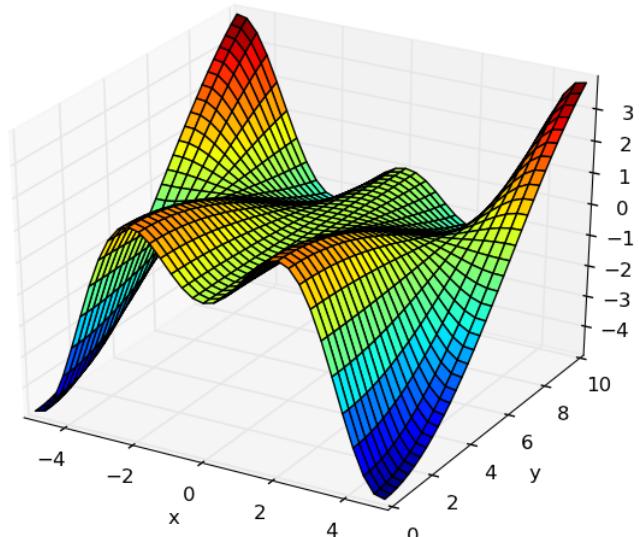
```
# change the number of bins  
>>> hist(randn(1000), 30)
```



# 3D Plots with Matplotlib

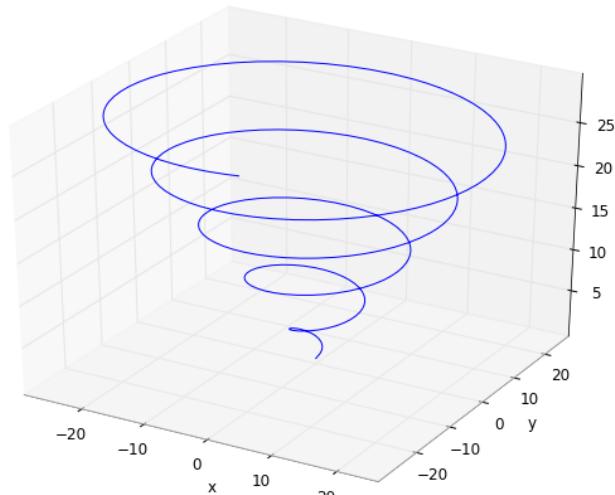
## SURFACE PLOT

```
>>> from mpl_toolkits.mplot3d import
    Axes3D
>>> x, y = mgrid[-5:5:35j, 0:10:35j]
>>> z = x*sin(x)*cos(0.25*y)
>>> fig = figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot_surface(x, y, z,
...                   rstride=1, cstride=1,
...                   cmap=cm.jet)
>>> xlabel('x'); ylabel('y')
```



## PARAMETRIC CURVE

```
>>> from mpl_toolkits.mplot3d import
    Axes3D
>>> t = linspace(0, 30, 1000)
>>> x, y, z = [t*cos(t), t*sin(t), t]
>>> fig = figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot(x, y, z)
>>> xlabel('x')
>>> ylabel('y')
```



# Numpy (continued)

# Introducing NumPy Arrays

## SIMPLE ARRAY CREATION

```
>>> a = array([0,1,2,3])  
>>> a  
array([0, 1, 2, 3])
```

## CHECKING THE TYPE

```
>>> type(a)  
numpy.ndarray
```

## NUMERIC ‘TYPE’ OF ELEMENTS

```
>>> a.dtype  
dtype('int32')
```

## NUMBER OF DIMENSIONS

```
>>> a.ndim  
1
```

## ARRAY SHAPE

```
# Shape returns a tuple  
# listing the length of the  
# array along each dimension.  
>>> a.shape  
(4,)
```

## BYTES PER ELEMENT

```
>>> a.itemsize  
4
```

## BYTES OF MEMORY USED

```
# Return the number of bytes  
# used by the data portion of  
# the array.  
>>> a.nbytes  
16
```

# Array Operations

## SIMPLE ARRAY MATH

```
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b
array([3, 5, 7, 9])
>>> a * b
array([ 2,  6, 12, 20])
>>> a ** b
array([ 1,   8,   81, 1024])
```

## MATH FUNCTIONS

```
# create array from 0 to 10
>>> x = arange(11.)

# multiply entire array by
# scalar value
>>> c = (2*pi)/10.
>>> c
0.62831853071795862
>>> c*x
array([ 0., 0.628, ..., 6.283])
```

## # in-place operations

```
>>> x *= c
>>> x
array([ 0., 0.628, ..., 6.283])
```

## # apply functions to array

```
>>> y = sin(x)
```

NumPy defines these constants:



pi = 3.14159265359  
e = 2.71828182846

# Setting Array Elements

## ARRAY INDEXING

```
>>> a[0]  
0  
>>> a[0] = 10  
>>> a  
array([10, 1, 2, 3])
```



## BEWARE OF TYPE COERCION

```
>>> a.dtype  
dtype('int32')  
  
# assigning a float into  
# an int32 array truncates  
# the decimal part  
>>> a[0] = 10.6  
>>> a  
array([10, 1, 2, 3])  
  
# fill has the same behavior  
>>> a.fill(-4.8)  
>>> a  
array([-4, -4, -4, -4])
```

# Multi-Dimensional Arrays

## MULTI-DIMENSIONAL ARRAYS

```
>>> a = array([[ 0,  1,  2,  3],  
               [10,11,12,13]])  
  
>>> a  
array([[ 0,  1,  2,  3],  
       [10,11,12,13]])
```

## SHAPE = (ROWS,COLUMNS)

```
>>> a.shape  
(2, 4)
```

## ELEMENT COUNT

```
>>> a.size  
8
```

## NUMBER OF DIMENSIONS

```
>>> a.ndim  
2
```

## GET/SET ELEMENTS

```
>>> a[1,3]  
13
```



```
>>> a[1,3] = -1  
>>> a  
array([[ 0,  1,  2,  3],  
       [10,11,12, -1]])
```

## ADDRESS SECOND (ONETH) ROW USING SINGLE INDEX

```
>>> a[1]  
array([10, 11, 12, -1])
```

# Slicing

**var [lower:upper:step]**

Extracts a portion of a sequence by specifying a lower and upper bound.

The lower-bound element is included, but the upper-bound element is **not** included.

Mathematically: [lower, upper). The step value specifies the stride between elements.

## SLICING ARRAYS

```
# indices:    0  1  2  3  4
>>> a = array([10,11,12,13,14])
# [10,11,12,13,14]
>>> a[1:3]
array([11, 12])

# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

## OMITTING INDICES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list

# grab first three elements
>>> a[:3]
array([10, 11, 12])
# grab last two elements
>>> a[-2:]
array([13, 14])
# every other element
>>> a[::-2]
array([10, 12, 14])
```

# Array Slicing

SLICING WORKS MUCH LIKE  
STANDARD PYTHON SLICING

```
>>> a[0,3:5]  
array([3, 4])
```

```
>>> a[4:,4:]  
array([[44, 45],  
      [54, 55]])
```

```
>>> a[:,2]  
array([2,12,22,32,42,52])
```

STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,:2]  
array([[20, 22, 24],  
      [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

# Slices Are References

Slices are references to memory in the original array.

Changing values in a slice also changes the original array.

```
>>> a = array((0,1,2,3,4))  
        
      # create a slice containing only the  
      # last element of a  
>>> b = a[2:4]  
>>> b  
array([2, 3])  
>>> b[0] = 10  
  
      # changing b changed a!  
>>> a  
array([ 0,  1, 10,  3,  4])
```

# Where

## 1 DIMENSION

```
# find the indices in array
# where expression is True
>>> a = array([0, 12, 5, 20])
>>> a > 10
array([False, True, False,
       True], dtype=bool)
```

```
# Note: it returns a tuple!
>>> where(a > 10)
(array([1, 3]),)
```

## n DIMENSIONS

```
# In general, the tuple
# returned is the index of the
# element satisfying the
# condition in each dimension.
>>> a = array([[0, 12, 5, 20],
              [1, 2, 11, 15]])
>>> loc = where(a > 10)
>>> loc
(array([0, 0, 1, 1]),
array([1, 3, 2, 3]))
```

```
# Result can be used in
# various ways:
>>> a[loc]
array([12, 20, 11, 15])
```

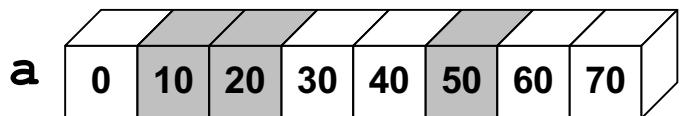
# Fancy Indexing

## INDEXING BY POSITION

```
>>> a = arange(0,80,10)  
  
# fancy indexing  
>>> indices = [1, 2, -3]  
>>> y = a[indices]  
>>> print(y)  
[10 20 50]
```

## INDEXING WITH BOOLEANS

```
# manual creation of masks  
>>> mask = array([0,1,1,0,0,1,0,0],  
...                      dtype=bool)  
  
# conditional creation of masks  
>>> mask2 = a < 30  
  
# fancy indexing  
>>> y = a[mask]  
>>> print(y)  
[10 20 50]
```



# Fancy Indexing in 2-D

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
```

```
array([ 1, 12, 23, 34, 45])
```

```
>>> a[3:, [0, 2, 5]]
```

```
array([[30, 32, 35],  
       [40, 42, 45],  
       [50, 52, 55]])
```

```
>>> mask = array([1,0,1,0,0,1],  
                  dtype=bool)
```

```
>>> a[mask,2]
```

```
array([2,22,52])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

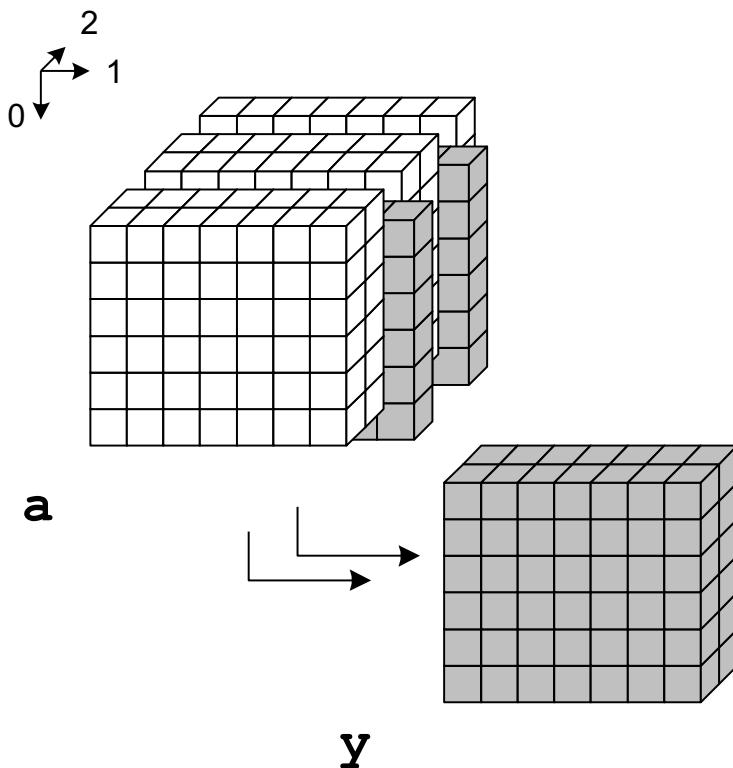


Unlike slicing, fancy indexing creates copies instead of a view into original array.

# 3D Example

## MULTIDIMENSIONAL

```
# retrieve two slices from a  
# 3D cube via indexing  
>>> y = a[:, :, [2, -2]]
```



# Creating arrays

# Array Constructor Examples

## FLOATING POINT ARRAYS

```
# Default to double precision
>>> a = array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

## REDUCING PRECISION

```
>>> a = array([0,1.,2,3],
...             dtype=float32)
>>> a.dtype
dtype('float32')
>>> a.nbytes
16
```

## UNSIGNED INTEGER BYTE

```
>>> a = array([0,1,2,3],
...             dtype=uint8)
>>> a.dtype
dtype('uint8')
>>> a.nbytes
4
```

# Array Creation Functions

## ARANGE

```
arange(start=None, stop, step=1,  
       dtype=None)
```

Nearly identical to Python's `range()`.

Creates an array of values in the range [start,stop) with the specified step value.

Allows non-integer values for start, stop, and step. Default `dtype` is derived from the start, stop, and step values.

```
>>> arange(4)
```

```
array([0, 1, 2, 3])
```

```
>>> arange(0, 2*pi, pi/4)
```

```
array([ 0.000, 0.785, 1.571,  
       2.356, 3.142, 3.927, 4.712,  
       5.497])
```

# Be careful...

```
>>> arange(1.5, 2.1, 0.3)
```

```
array([ 1.5, 1.8, 2.1])
```

## ONES, ZEROS

```
ones(shape, dtype=float64)  
zeros(shape, dtype=float64)
```

`shape` is a number or sequence specifying the dimensions of the array. If `dtype` is not specified, it defaults to `float64`.

```
>>> ones((2,3), dtype=float32)  
array([[ 1.,  1.,  1.],  
      [ 1.,  1.,  1.]],  
      dtype=float32)
```

```
>>> zeros(3)
```

```
array([ 0.,  0.,  0.])
```

# Array Creation Functions (cont.)



## IDENTITY

```
# Generate an n by n identity
# array. The default dtype is
# float64.
>>> a = identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> a.dtype
dtype('float64')
>>> identity(4, dtype=int)
array([[ 1,  0,  0,  0],
       [ 0,  1,  0,  0],
       [ 0,  0,  1,  0],
       [ 0,  0,  0,  1]])
```

## EMPTY AND FILL

```
# empty(shape, dtype=float64,
#       order='C')
>>> a = empty(2)
>>> a
array([1.78021120e-306,
       6.95357225e-308])

# fill array with 5.0
>>> a.fill(5.0)
array([5.,  5.])

# alternative approach
# (slightly slower)
>>> a[:] = 4.0
array([4.,  4.])
```

# Array Creation Functions (cont.)

## LINSPACE

```
# Generate N evenly spaced
# elements between (and
# including) start and
# stop values.
>>> linspace(0,1,5)
array([0., 0.25, 0.5, 0.75, 1.0])
```

## LOGSPACE

```
# Generate N evenly spaced
# elements on a log scale
# between base**start and
# base**stop (default base=10).
>>> logspace(0,1,5)
array([ 1., 1.77, 3.16, 5.62,
       10.])
```

## ARRAYS FROM/TO TXT FILES

### Data.txt

```
-- BEGINNING OF THE FILE
% Day, Month, Year, Skip, Avg Power
01, 01, 2000, x876, 13 % crazy day!
% we don't have Jan 03rd
04, 01, 2000, xfed, 55
```

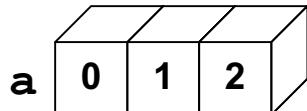
```
# loadtxt() automatically generates
# an array from the txt file
arr = loadtxt('Data.txt', skiprows=1,
               dtype=int, delimiter=",",
               usecols = (0,1,2,4),
               comments = "%")
```

```
# Save an array into a txt file
savetxt('filename', arr)
```

# Indexing with newaxis

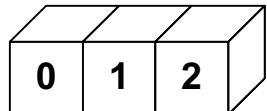
**newaxis** is a special index that inserts a new axis in the array at the specified location.

Each **newaxis** increases the array's dimensionality by 1.



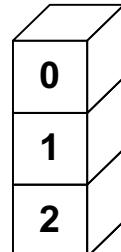
**1 X 3**

```
>>> shape(a)
(3, )
>>> y = a[newaxis,:]
>>> shape(y)
(1, 3)
```



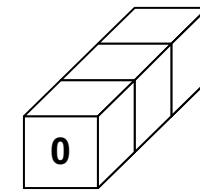
**3 X 1**

```
>>> y = a[:,newaxis]
>>> shape(y)
(3, 1)
```



**1 X 1 X 3**

```
> y = a[newaxis,newaxis,:]
> shape(y)
(1, 1, 3)
```



# “Flattening” Arrays

## a.flatten()

`a.flatten()` converts a multi-dimensional array into a 1-D array. The new array is a *copy* of the original data.

```
# Create a 2D array
>>> a = array([[0,1],
              [2,3]])

# Flatten out elements to 1D
>>> b = a.flatten()
>>> b
array([0,1,2,3])

# Changing b does not change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[ 0,  1],
       [ 2,  3]])
```

no change

## a.flat

`a.flat` is an *attribute* that returns an iterator object that accesses the data in the multi-dimensional array data as a 1-D array. It *references* the original memory.

```
>>> a.flat
<numpy.flatiter obj...>
>>> a.flat[:]
array(0,1,2,3)

>>> b = a.flat
>>> b[0] = 10
>>> a
array([[10,  1],
       [ 2,  3]])
```

changed!

# “(Un)raveling” Arrays

## a.ravel()

`a.ravel()` is the same as `a.flatten()`, but returns a *reference* (or *view*) of the array if possible (i.e., the memory is contiguous). Otherwise the new array copies the data.

```
# create a 2-D array
>>> a = array([[0,1],
              [2,3]])

# flatten out elements to 1-D
>>> b = a.ravel()
>>> b
array([0,1,2,3])

# changing b does change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[10,  1],
       [ 2,  3]])
```

changed!

## a.ravel() MAKES A COPY

```
# transpose array so memory
# layout is no longer contiguous
>>> aa = a.transpose()
>>> aa
array([[0,  2],
       [1,  3]])

# ravel creates a copy of data
>>> b = aa.ravel()
array([0,2,1,3])

# changing b doesn't change a
>>> b[0] = 10
>>> b
array([10,1,2,3])
>>> a
array([[ 0,  1],
       [ 2,  3]])
```

# Reshaping Arrays

## SHAPE

```
>>> a = arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape
(6,)
```

```
# reshape array in-place to
# 2x3
>>> a.shape = (2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```

## RESHAPE

```
# return a new array with a
# different shape
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
# reshape cannot change the
# number of elements in an
# array
>>> a.reshape(4,2)
ValueError: total size of new
array must be unchanged
```

# Type Casting

## ASARRAY

```
>>> a = array([1.5, -3],  
...             dtype=float32)  
>>> a  
array([ 1.5, -3.], dtype=float32)  
  
# upcast  
>>> asarray(a, dtype=float64)  
array([ 1.5, -3. ])  
  
# downcast  
>>> asarray(a, dtype=uint8)  
array([ 1, 253], dtype=uint8)  
  
# asarray is efficient.  
# It does not make a copy if the  
# type is the same.  
>>> b = asarray(a, dtype=float32)  
>>> b[0] = 2.0  
>>> a  
array([ 2., -3.], dtype=float32)
```

## ASTYPE

```
>>> a = array([1.5, -3],  
...             dtype=float64)  
>>> a.astype(float32)  
array([ 1.5, -3.], dtype=float32)  
  
>>> a.astype(uint8)  
array([ 1, 253], dtype=uint8)  
  
# astype is safe.  
# It always returns a copy of  
# the array.  
>>> b = a.astype(float64)  
>>> b[0] = 2.0  
>>> a  
array([1.5, -3.])
```

# NumPy dtypes

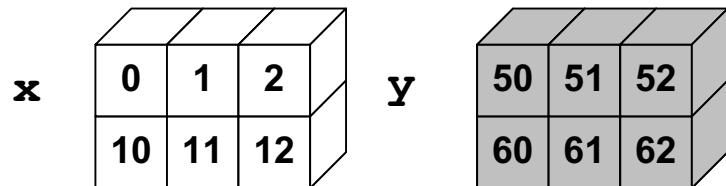
DATA SCIENCE

Type	Available NumPy types	Code	Comments
Boolean	bool	b	Elements are 1 byte in size.
Integer	int8, int16, int32, int64, int128, int	i	int defaults to the size of long in C for the platform.
Unsigned Integer	uint8, uint16, uint32, uint64, uint128, uint	u	uint defaults to the size of unsigned long in C for the platform.
Float	float16, float32, float64, float, longfloat	f	float is always a double precision floating point value (64 bits). longfloat represents large precision floats. Its size is platform dependent.
Complex	complex64, complex128, complex, longcomplex	c	The real and imaginary elements of a complex64 are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	str, unicode	S or a, U	For example, dtype='S4' would be used for an array of 4-character strings.
Datetime	datetime64, timedelta64	None	Allow operations between dates and/or times. New in 1.7.
Object	object	o	Represent items in array as Python objects.
Records	void	v	Used for arbitrary data structures.

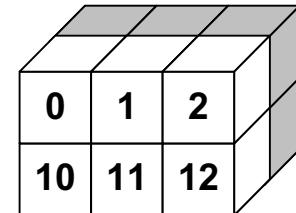
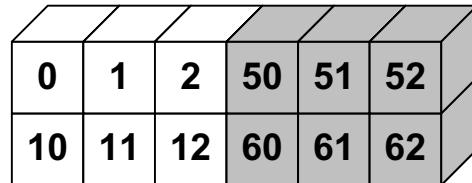
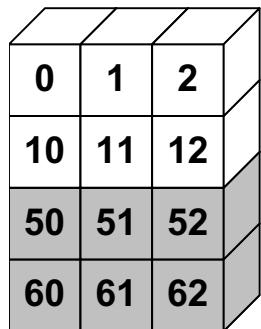
# Concatenate

`concatenate( (a0,a1,...,aN) ,axis=0)`

The input arrays `(a0,a1,...,aN)` are concatenated along the given `axis`. They must have the same shape along every axis *except* the one given.

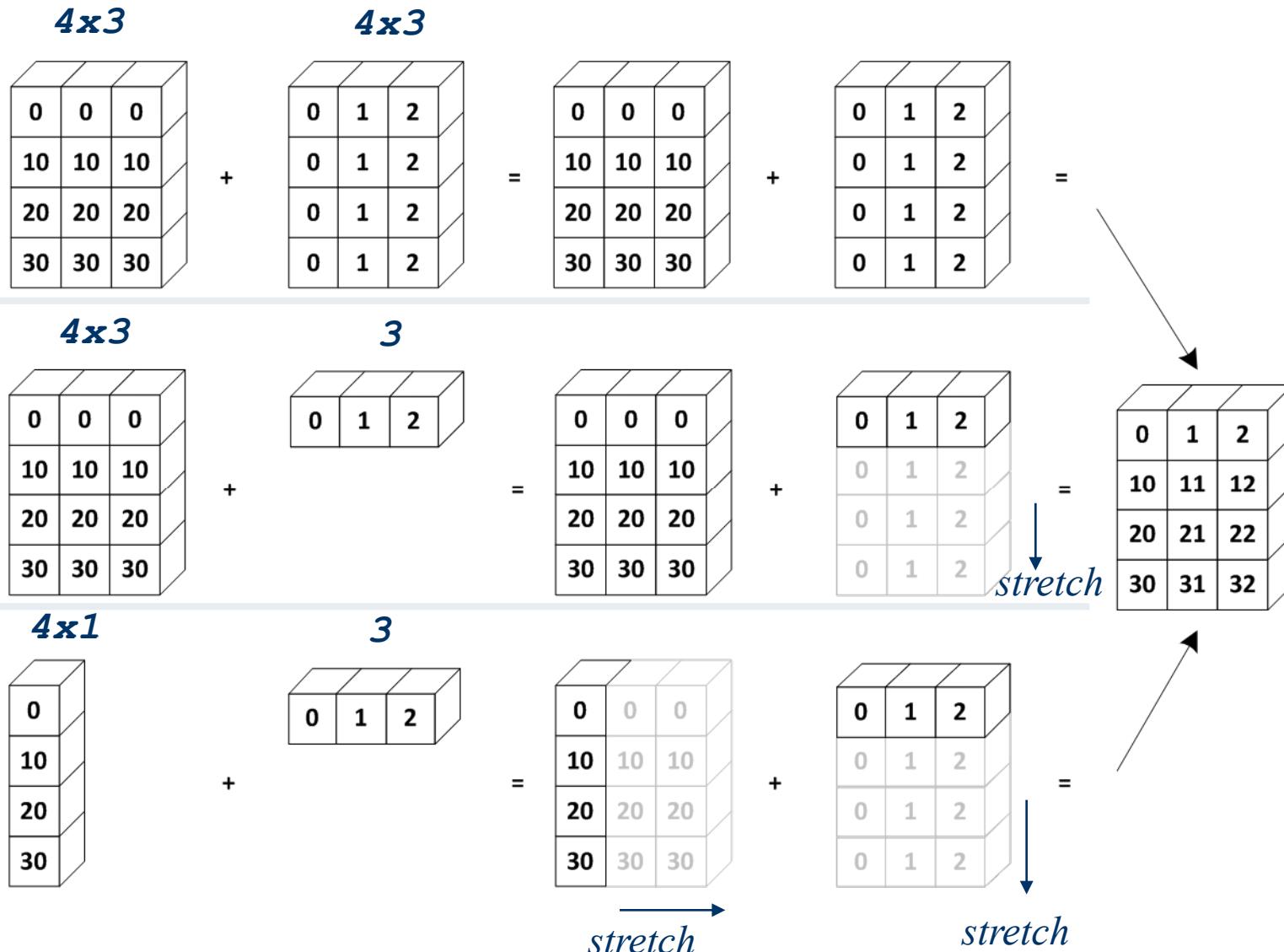


`>>> concatenate( (x,y) )`    `>>> concatenate( (x,y),1 )`    `>>> array( (x,y) )`



See also `vstack()`, `hstack()` and `dstack()` respectively.

# Array Broadcasting



# Array calculation methods

# Array Calculation Methods

## SUM FUNCTION

```
>>> a = array([[1,2,3],  
              [4,5,6]])  
  
# sum() defaults to adding up  
# all the values in an array.  
>>> sum(a)  
21  
  
# supply the keyword axis to  
# sum along the 0th axis  
>>> sum(a, axis=0)  
array([5, 7, 9])  
  
# supply the keyword axis to  
# sum along the last axis  
>>> sum(a, axis=-1)  
array([ 6, 15])
```

## SUM ARRAY METHOD

```
# a.sum() defaults to adding  
# up all values in an array.  
>>> a.sum()  
21  
  
# supply an axis argument to  
# sum along a specific axis  
>>> a.sum(axis=0)  
array([5, 7, 9])
```

## PRODUCT

```
# product along columns  
>>> a.prod(axis=0)  
array([ 4, 10, 18])  
  
# as a function  
>>> prod(a, axis=0)  
array([ 4, 10, 18])
```

# Min/Max

## MIN

```
>>> a = array([2.,3.,0.,1.])
>>> a.min(axis=0)
0.0
# Use NumPy's amin() instead
# of Python's built-in min()
# for speedy operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.0
```

## ARGMIN

```
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# as a function
>>> argmin(a, axis=0)
2
```

## MAX

```
>>> a = array([2.,3.,0.,1.])
>>> a.max(axis=0)
3.0
```

## # as a function

```
>>> amax(a, axis=0)
3.0
```

## ARGMAX

```
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# as a function
>>> argmax(a, axis=0)
1
```

# Statistics Array Methods

## MEAN

```
>>> a = array([[1,2,3],  
              [4,5,6]])  
  
# mean value of each column  
>>> a.mean(axis=0)  
array([ 2.5,  3.5,  4.5])  
>>> mean(a, axis=0)  
array([ 2.5,  3.5,  4.5])  
>>> average(a, axis=0)  
array([ 2.5,  3.5,  4.5])  
  
# average can also calculate  
# a weighted average  
>>> average(a, weights=[1,2],  
...           axis=0)  
array([ 3.,  4.,  5.])
```

## STANDARD DEV./VARIANCE

```
# Standard Deviation  
>>> a.std(axis=0)  
array([ 1.5,  1.5,  1.5])  
  
# variance  
>>> a.var(axis=0)  
array([2.25, 2.25, 2.25])  
>>> var(a, axis=0)  
array([2.25, 2.25, 2.25])
```

# Trig and math Functions

DATA SCIENCE

## TRIGONOMETRIC

<code>sin(x)</code>	<code>sinh(x)</code>
<code>cos(x)</code>	<code>cosh(x)</code>
<code>arccos(x)</code>	<code>arccosh(x)</code>
<code>arctan(x)</code>	<code>arctanh(x)</code>
<code>arcsin(x)</code>	<code>arcsinh(x)</code>
<code>arctan2(x,y)</code>	

## OTHERS

<code>exp(x)</code>	<code>log(x)</code>
<code>log10(x)</code>	<code>sqrt(x)</code>
<code>absolute(x)</code>	<code>conjugate(x)</code>
<code>negative(x)</code>	<code>ceil(x)</code>
<code>floor(x)</code>	<code>fabs(x)</code>
<code>hypot(x,y)</code>	<code>fmod(x,y)</code>
<code>maximum(x,y)</code>	<code>minimum(x,y)</code>

## VECTOR OPERATIONS

<code>dot(x,y)</code>	<code>vdot(x,y)</code>
<code>inner(x,y)</code>	<code>outer(x,y)</code>
<code>cross(x,y)</code>	<code>kron(x,y)</code>
<code>tensordot(x,y[,axis])</code>	

## hypot(x,y)

Element by element distance  
calculation using

$$\sqrt{x^2 + y^2}$$

# Other array functions

DATA SCIENCE

## TYPE HANDLING

<code>iscomplexobj</code>	<code>real_if_close</code>	<code>isnan</code>
<code>iscomplex</code>	<code>isscalar</code>	<code>nan_to_num</code>
<code>isrealobj</code>	<code>isneginf</code>	<code>common_type</code>
<code>isreal</code>	<code>isposinf</code>	<code>typename</code>
<code>imag</code>	<code>isinf</code>	
<code>real</code>	<code>isfinite</code>	

## OTHER USEFUL FUNCTIONS

<code>fix</code>	<code>unwrap</code>	<code>roots</code>
<code>mod</code>	<code>sort_complex</code>	<code>poly</code>
<code>amax</code>	<code>trim_zeros</code>	<code>any</code>
<code>amin</code>	<code>fliplr</code>	<code>all</code>
<code>ptp</code>	<code>flipud</code>	<code>disp</code>
<code>sum</code>	<code>rot90</code>	<code>unique</code>
<code>cumsum</code>	<code>eye</code>	<code>nansum</code>
<code>prod</code>	<code>diag</code>	<code>nanmax</code>
<code>cumprod</code>	<code>select</code>	<code>nanargmax</code>
<code>diff</code>	<code>extract</code>	<code>nanargmin</code>
<code>angle</code>	<code>insert</code>	<code>nanmin</code>

## SHAPE MANIPULATION

<code>atleast_1d</code>	<code>hstack</code>	<code>hsplit</code>
<code>atleast_2d</code>	<code>vstack</code>	<code>vsplit</code>
<code>atleast_3d</code>	<code>dstack</code>	<code>dsplit</code>
<code>expand_dims</code>	<code>column_stack</code>	<code>split</code>
<code>apply_over_axes</code>		<code>squeeze</code>
<code>apply_along_axis</code>		

# Vectorizing Functions

## SCALAR SINC FUNCTION

```
# special.sinc already available
# This is just for show.

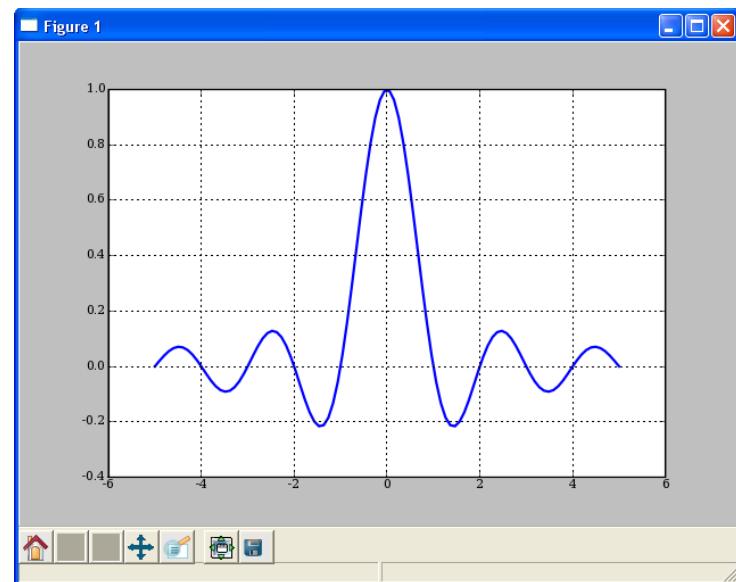
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

```
# attempt
>>> x = array((1.3, 1.5))
>>> sinc(x)
ValueError: The truth value of
an array with more than one
element is ambiguous. Use
a.any() or a.all()
```

## SOLUTION

```
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc(x)
array([-0.1981, -0.2122])

>>> x2 = linspace(-5, 5, 101)
>>> plot(x2, vsinc(x2))
```



# DAY 4

# "Advanced" NumPy

# Advanced NumPy overview

---

NumPy is the low-level core of most Python Data Science libraries.

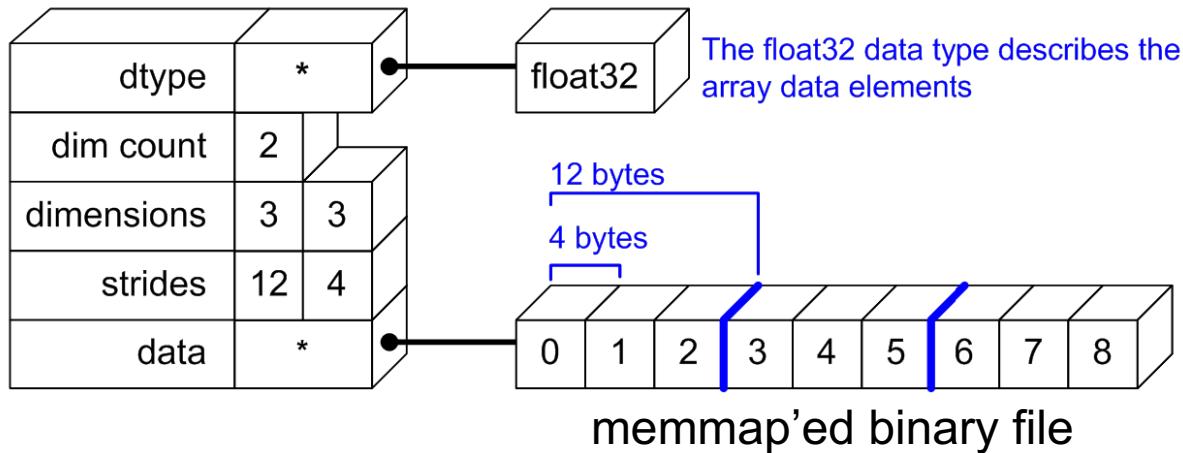
There are a couple of advanced NumPy topics that it's worth being aware of:

- structured arrays
- memmap'ed arrays

# memmap'ed arrays

The array data can come from any buffer-like storage, including memmap'ed files. Users can use the array object transparently, and the OS creates memory pages as needed.

**NDArray Data Structure**



**Python View:**

0	1	2
3	4	5
6	7	8

# Memory Mapped Arrays

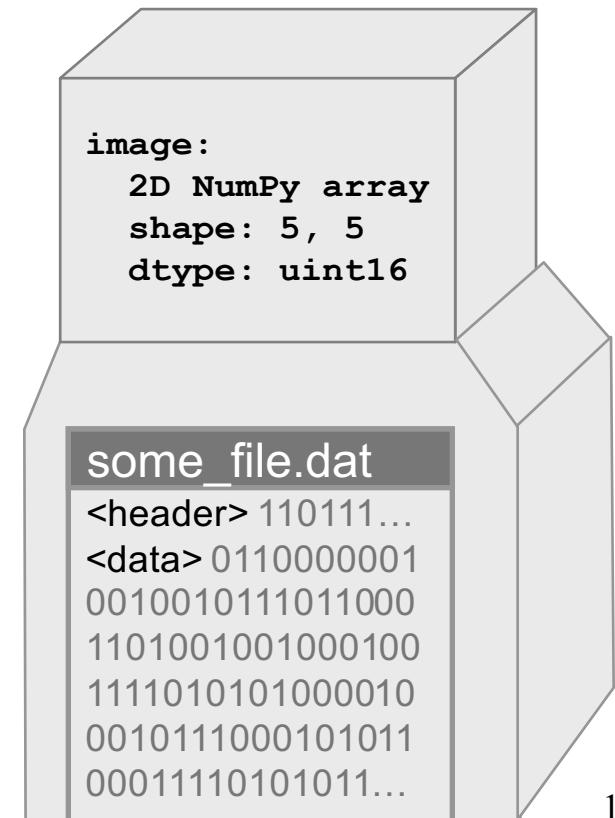
- Methods for Creating:
  - **memmap**: subclass of ndarray that manages the memory mapping details.
  - **frombuffer**: Create an array from a memory mapped buffer object.
  - **ndarray constructor**: Use the `buffer` keyword to pass in a memory mapped buffer.
- Limitations:
  - Files must be < 2GB on Python 2.4 and before.
  - Files must be < 2GB on 32-bit machines.
  - Python 2.5 and higher on 64 bit machines is theoretically "limited" to 17.2 *billion* GB (17 Exabytes).

# Memory Mapped Example

```
# Create a "memory mapped" array where
# the array data is stored in a file on
# disk instead of in main memory.
>>> from numpy import memmap
>>> image = memmap('some_file.dat',
                     dtype='uint16',
                     mode='r+',
                     shape=(5,5),
                     offset=header_size)

# Standard array methods work.
>>> mean_value = image.mean()

# Standard math operations work.
# The resulting scaled_image *is*
# stored in main memory. It is a
# standard numpy array.
>>> scaled_image = image * .5
```



# memmap

The `memmap` subclass of `array` handles opening and closing files as well as synchronizing memory with the underlying file system.

```
memmap(filename, dtype=uint8, mode='r+',  
       offset=0, shape=None, order=0)
```

**filename** Name of the underlying file. For all modes, except for 'w+', the file must already exist and contain at least the number of bytes used by the array.

**dtype** The numpy data type used for the array. This can be a "structured" dtype as well as the standard simple data types.

**offset** Byte offset within the file to the memory used as data within the array.

**mode** <see next slide>

**shape** Tuple specifying the dimensions and size of each dimension in the array. `shape=(5,10)` would create a 2D array with 5 rows and 10 columns.

**order** 'C' for row major memory ordering (standard in the C programming language) and 'F' for column major memory ordering (standard in Fortran).

# memmap -- mode

DATA SCIENCE

The mode setting for memmap arrays is used to set the access flag when opening the specified file using the standard mmap module.

```
memmap(filename, dtype=uint8, mode='r+',  
       offset=0, shape=None, order=0)
```

**mode** A string indicating how the underlying file should be opened.

'r' or 'readonly': Open an existing file as an array for reading.

'c' or 'copyonwrite': "Copy on write" arrays are "writable" as Python arrays, but they *never* modify the underlying file.

'r+' or 'readwrite': Create a read/write array from an existing file. The file will have "write through" behavior where changes to the array are written to the underlying file. Use the `flush()` method to ensure the array is synchronized with the file.

'w+' or 'write': Create the file or overwrite if it exists. The array is filled with zeros and has "write through" behavior similar to 'r+'.

## File Format:

header

rows (int32)	cols (int32)
--------------	--------------

data

64 bit floating point data...

```
# Create a dtype to represent the header.  
header_dtype = dtype([('rows', int32), ('cols', int32)])  
  
# Create a memory mapped array using this dtype. Note the shape is empty.  
header = memmap(file_name, mode='r', dtype=header_dtype, shape=())  
  
# Read the row and column sizes from using this structured array.  
rows = header['rows']  
cols = header['cols']  
  
# Create a memory map to the data segment, using rows, cols for shape  
# information and the header size to determine the correct offset.  
data = memmap(file_name, mode='r+', dtype=float64,  
              shape=(rows, cols), offset=header_dtype.itemsize)
```

# Structured arrays

Structured arrays allow interpreting the array elements as fields of multiple types. Combined with memmaps, it increases the opportunities for creating disk-backed arrays from binary files.

Elements of an array can be any fixed-size data structure!

```
name  char[10]
age    int
weight double
```

Brad	Jane	John	Fred
33	25	47	54
135.0	105.0	225.0	140.0
Henry	George	Brian	Amy
29	61	32	27
154.0	202.0	137.0	187.0
Ron	Susan	Jennifer	Jill
19	33	18	54
188.0	135.0	88.0	145.0

## EXAMPLE

```
>>> from numpy import dtype, empty
# structured data format
>>> fmt = dtype([('name', 'S10'),
   ('age', int),
   ('weight', float)
])
>>> a = empty((3,4), dtype=fmt)
```

# Structured Arrays

```
# "Data structure" (dtype) that describes the fields and
# type of the items in each array element.
>>> particle_dtype = dtype([('mass','float32'), ('velocity', 'float32')])
# This must be a list of tuples.
>>> particles = array([(1,1), (1,2), (2,1), (1,3)],
                      dtype=particle_dtype)
>>> print particles
[(1.0, 1.0) (1.0, 2.0) (2.0, 1.0) (1.0, 3.0)]
# Retrieve the mass for all particles through indexing.
>>> print particles['mass']
[ 1.  1.  2.  1.]
# Retrieve particle 0 through indexing.
>>> particles[0]
(1.0, 1.0)
# Sort particles in place, with velocity as the primary field and
# mass as the secondary field.
>>> particles.sort(order=('velocity','mass'))
>>> print particles
[(1.0, 1.0) (2.0, 1.0) (1.0, 2.0) (1.0, 3.0)]
```

# Nested Datatype

nested.dat

Time	Size	Position				Gain	Samples (2048) ...			
		Az	EI	Type	ID					
1172581077060	4108	0.715594	-0.148407	1	4	40	561	1467	997	-30
1172581077091	4108	0.706876	-0.148407	1	4	40	7	591	423	
1172581077123	4108	0.698157	-0.148407	1	4	40	49	-367	-565	-35
1172581077153	4108	0.689423	-0.148407	1	4	40	-55	-953	-1151	-30
1172581077184	4108	0.680683	-0.148407	1	4	40	-719	-1149	-491	38
1172581077215	4108	0.671956	-0.148407	1	4	40	-1503	-683	661	149
1172581077245	4108	0.663232	-0.148407	1	4	40	-2731	-281	2327	291
1172581077276	4108	0.654511	-0.148407	1	4	40	-3493	-159	3277	380
1172581077306	4108	0.645787	-0.148407	1	4	40	-3255	-247	3145	385
1172581077339	4108	0.637058	-0.148407	1	4	40	-2303	-101	2079	247
1172581077370	4108	0.628321	-0.148407	1	4	40	-1495	-553	571	107
1172581077402	4108	0.619599	-0.148407	1	4	40	-955	-1491	-1207	-25
1172581077432	4108	0.61087	-0.148407	1	4	40	-875	-3009	-2987	-93
1172581077463	4108	0.602148	-0.148407	1	4	40	-491	-3681	-4193	-175
1172581077497	4108	0.593438	-0.148407	1	4	40	167	-3501	-4573	-250
1172581077547	4108	0.584696	-0.148407	1	4	40	1007	-2613	-4463	-303
1172581077599	4108	0.575972	-0.148407	1	4	40	1261	-2155	-4299	-339
1172581077650	4108	0.567244	-0.148407	1	4	40	1537	-2633	-4945	-367
1172581077700	4108	0.558511	-0.148407	1	4	40	1105	-2701	-4120	420

# Nested Datatype (cont'd)

The data file can be extracted with the following code:

```
>>> dt = dtype([('time', uint64),  
...                 ('size', uint32),  
...                 ('position', [('az', float32),  
...                               ('el', float32),  
...                               ('region_type', uint8),  
...                               ('region_ID', uint16)]),  
...                 ('gain', uint8),  
...                 ('samples', int16, 2048)])  
  
>>> data = loadtxt('nested.dat', dtype=dt, skiprows = 2)  
>>> data['position']['az']  
array([ 0.71559399,  0.70687598,  0.69815701,  0.68942302,  
       0.68068302, ...], dtype=float32)
```

# SciPy

# SciPy overview

DATA SCIENCE

## CURRENT PACKAGES

- Special Functions (scipy.special)
- Signal Processing (scipy.signal)
- Image Processing (scipy.ndimage)
- Fourier Transforms (scipy.fftpack)
- Optimization (scipy.optimize)
- Numerical Integration (scipy.integrate)
- Linear Algebra (scipy.linalg)
- Input/Output (scipy.io)
- Statistics (scipy.stats)
- Clustering Algorithms (scipy.cluster)
- Sparse Matrices (scipy.sparse)
- Interpolation (scipy.interpolate)
- ... and more.

## SCIKITS

SciPy's “all in one” philosophy has been replaced by “scikits” projects:

- scikits.learn (sklearn), for Machine Learning functionality
- scikits.image, for image processing features
- statsmodel, for classical statistics (models, tests, etc)

# SciPy stats

# Data Science and statistics

- All Data Science methods are, in one way or another, statistical algorithms.
- A solid understanding of basic statistical ideas is necessary to:
  - Quality check the input data,
  - Verify that the data match the assumptions of each algorithm,
  - Check that the results are meaningful.

# Hypothesis testing

## Hypothesis testing checklist

## Example: one-sample t-test

Which effect to test?

Mean is different from  $\mu_0$

What would be true if the effect  
is *not* present?  
**“null hypothesis”**

Mean is equal to  $\mu_0$

How to measure the effect?  
**“statistic”**

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

Compute distribution of statistics  
under the null hypothesis  
(table or formula)

t-distribution  
Depends on the number of  
data points available

Compare statistic of observed  
data to that distribution.  
Is it highly unlikely? If so,  
discard null hypothesis

P-value  
Probability of a result as extreme  
or more as the one observed,  
under the null hypothesis

# Type I and Type II error

	Null hypothesis is true	Null hypothesis is false
Reject	Type I error “False positive”	All good
Do not reject	All good	Type II error “False negative”

- Type I error: incorrect rejection of a true null hypothesis
- Rate of Type I errors is fixed before the experiment is made, and is the threshold to which the P-value is compared
- Type II error: failure to reject a false null hypothesis
- Rate of Type II errors is usually high when the number of data points is low, or when a test is very generic

# t-tests

t-Tests: effect on the mean of one or two populations.

t-test variant	Null hypothesis	Assumptions	scipy.stats function
One-sample t-test	Data has given mean.	1. Independent samples 2. Normal distribution <sup>[2]</sup>	<code>ttest_1samp(data, mu0)</code>
Two-sample t-test, independent samples	Two independent sets of data have the same mean.	1. Independent samples 2. Normal distributions <sup>[2]</sup> 3. If <code>equal_var=True</code> , variances are the same	<code>ttest_ind(data_1, data_2, equal_var=True)</code>
Two-sample t-test, paired <sup>[1]</sup> samples	Two related sets of data have the same mean.	1. Independent samples 2. Normal distributions of pair differences <sup>[2]</sup> 3. Variance is the same	<code>ttest_rel(data_1, data_2)</code>

All t-test functions return a tuple, ( $t$ ,  $p$ )

$t$ : value of the  $t$  statistics,

$p$ : **two-tailed** p-value of the test.

(Divide by two to get one-tailed p-value.)

[1] E.g., repeated measurements  
(performance of students over time)

[2] The t-test is robust to violation of this assumption, but the power is reduced<sup>211</sup>

# More Hypothesis Tests

Test	Null hypothesis	<code>scipy.stats</code> function
ANOVA	Multiple samples $x_1, x_2, \dots$ have the same mean. Generalizes t-test to multiple group.	<code>f_oneway(x1, x2)</code>
Wilcoxon rank-sum test	Two independent samples are drawn from the same distribution. Alternative to t-test for non-normal distributions.	<code>ranksums(x, y)</code>
Kolmogorov-Smirnov test	Compare the distribution of a continuous variable to a known distribution. H0: the distr are identical	<code>kstest(x, 'norm')</code>
chi-square test	Compare the distribution of a categorical variable to a known distribution. H0: the distr are identical	<code>chisquare(x, f_exp=[16, 16, 8])</code>

Full list at <http://docs.scipy.org/doc/scipy/reference/stats.html>.

# Statistics

## scipy.stats — CONTINUOUS DISTRIBUTIONS

over 80  
continuous  
distributions!

### METHODS

**pdf** **entropy**

**cdf** **nnlf**

**rvs** **moment**

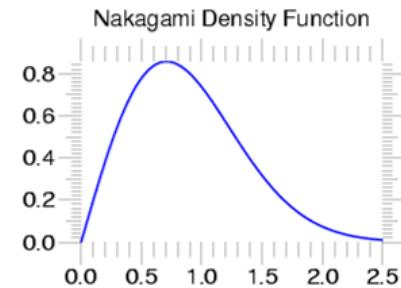
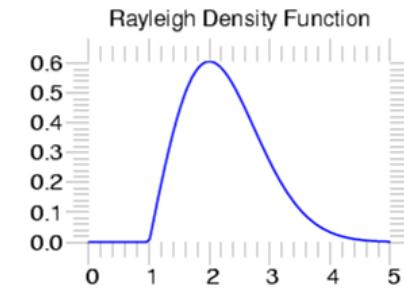
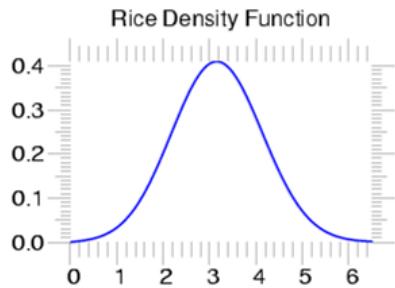
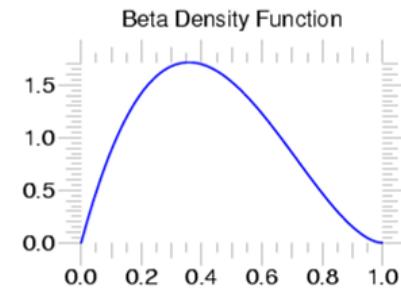
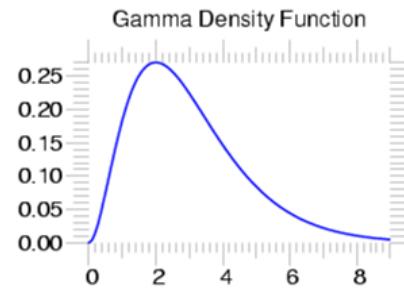
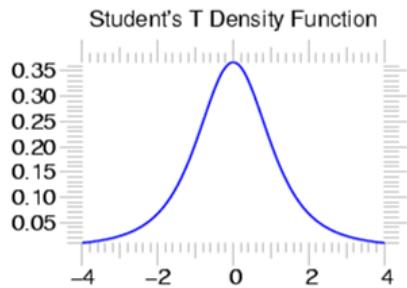
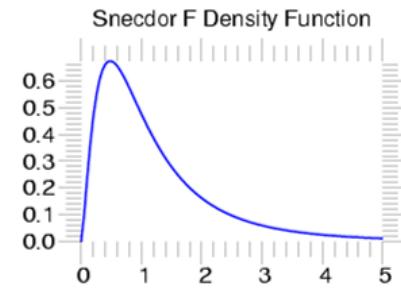
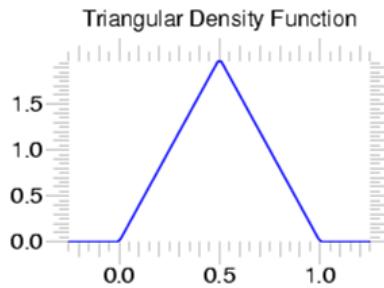
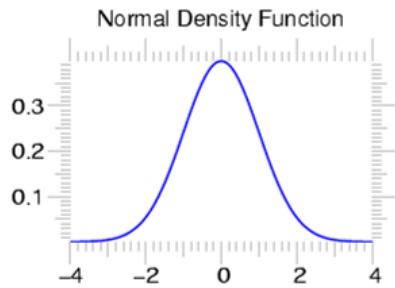
**ppf** **freeze**

**stats**

**fit**

**sf**

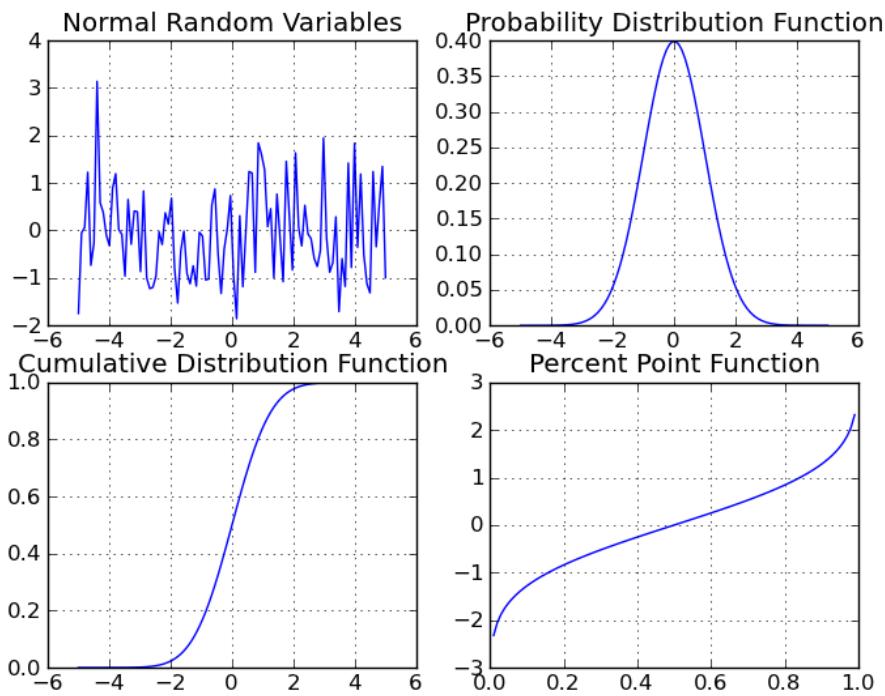
**isf**



# Using stats objects

## DISTRIBUTIONS

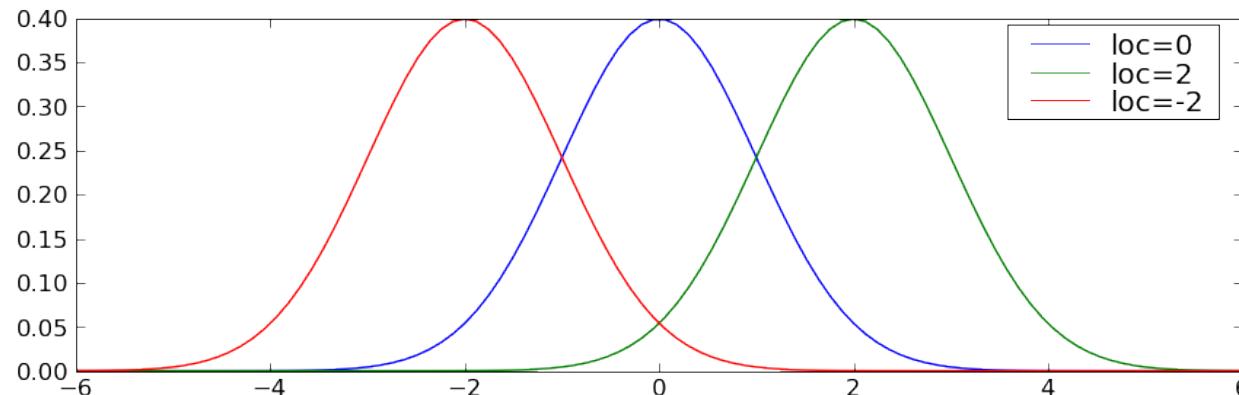
```
>>> from scipy.stats import norm  
# Sample normal dist. 100 times.  
>>> samp = norm.rvs(size=100)  
  
>>> x = linspace(-5, 5, 100)  
# Calculate probability dist.  
>>> pdf = norm.pdf(x)  
# Calculate cumulative dist.  
>>> cdf = norm.cdf(x)  
  
>>> x = linspace(0, 1, 100)  
# Calculate Percent Point Function  
>>> ppf = norm.ppf(x)
```



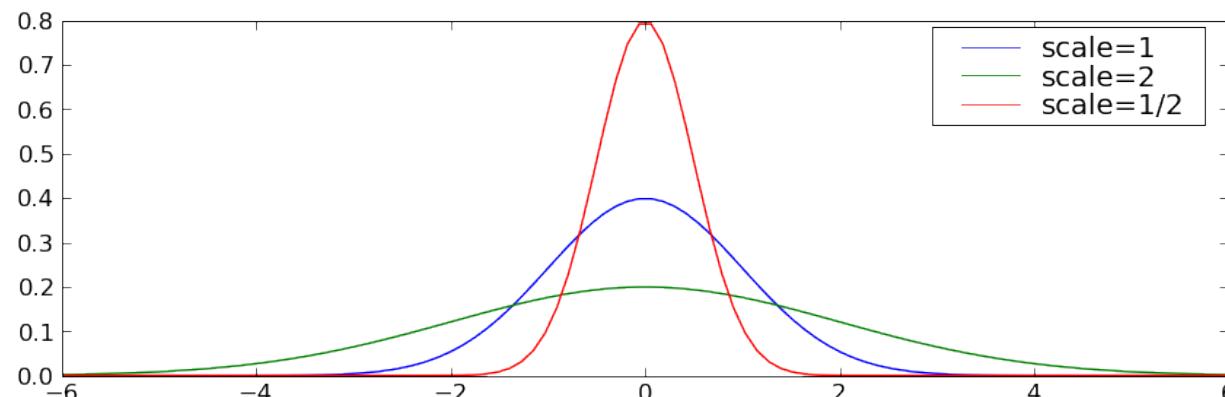
# Distribution objects

**Every distribution can be modified by `loc` and `scale` keywords**  
 (many distributions also have required `shape` arguments to select from a family)

**LOCATION (`loc`) --- shift left (<0) or right (>0) the distribution**



**SCALE (`scale`) --- stretch (>1) or compress (<1) the distribution**



# Example distributions

## NORM (norm) – $N( \mu, \sigma^2 )$

Only location and scale arguments:

location	mean	
scale	standard deviation	

## LOG NORMAL (lognorm)

$\log(S)$  is  $N( \mu, \sigma^2 )$

S is lognormal

one shape parameter!

location	offset from zero (rarely used)
scale	$e^\sigma$
shape	

# Setting location and Scale

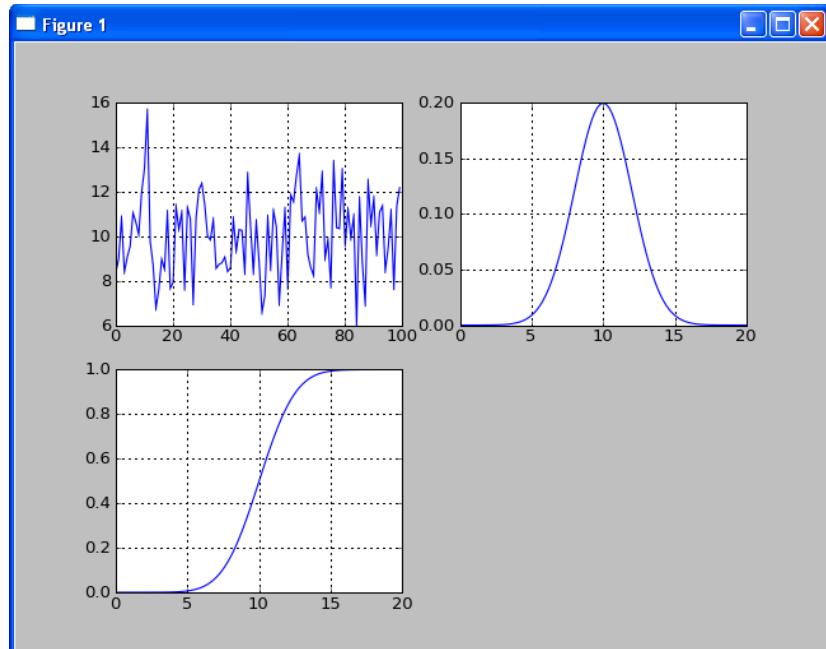
## NORMAL DISTRIBUTION

```
>>> from scipy.stats import norm
# Normal dist with mean=10 and std=2
>>> dist = norm(loc=10, scale=2)

>>> x = linspace(-5, 15, 100)
# Calculate probability dist.
>>> pdf = dist.pdf(x)
# Calculate cumulative dist.
>>> cdf = dist.cdf(x)

# Get 100 random samples from dist.
>>> samp = dist.rvs(size=100)

# Estimate parameters from data
>>> mu, sigma = norm.fit(samp)
>>> print "%4.2f, %4.2f" % (mu, sigma)
10.07, 1.95
```



**.fit returns best  
shape + (loc, scale)  
that explains the data**

# Statistics

## scipy.stats — Discrete Distributions

10 standard  
discrete  
distributions  
(plus any  
finite RV)

### METHODS

**pmf** **moment**

**cdf** **entropy**

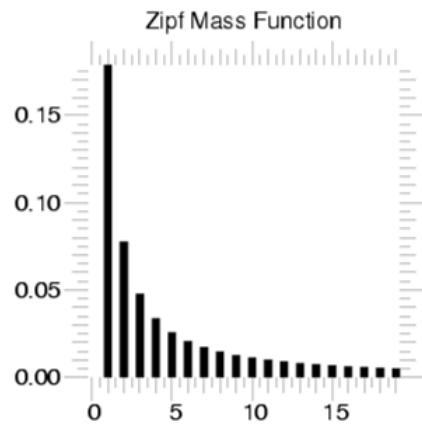
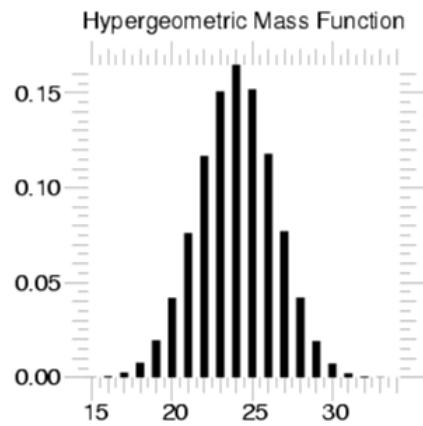
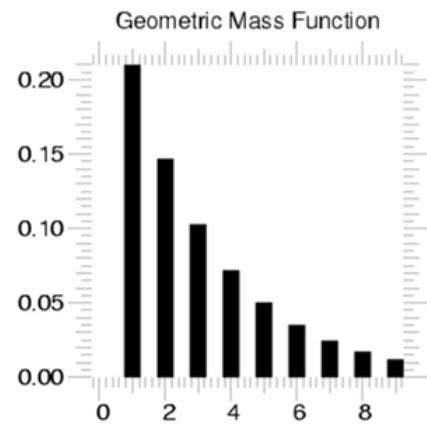
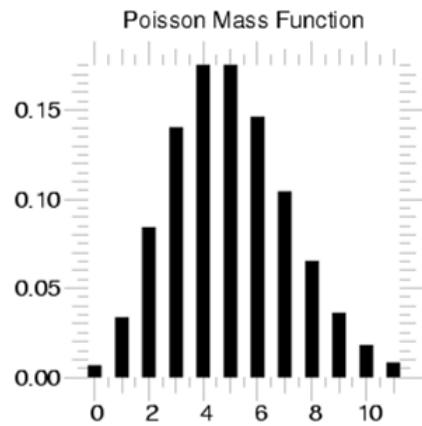
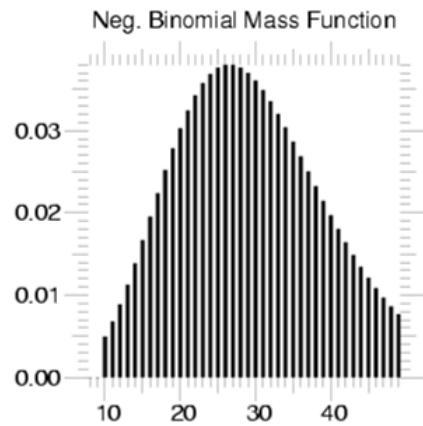
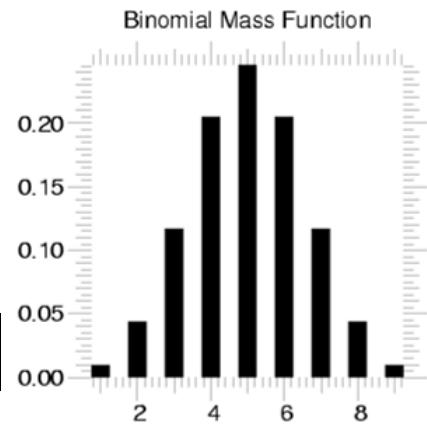
**rvs** **freeze**

**ppf**

**stats**

**sf**

**isf**



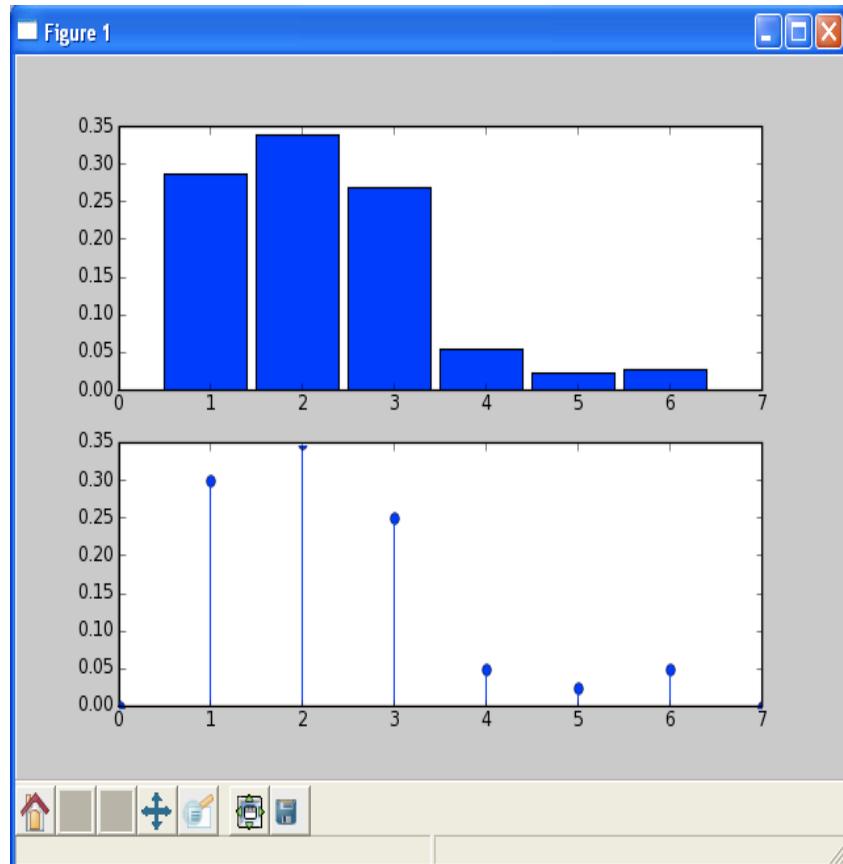
# Using stats objects

## CREATING NEW DISCRETE DISTRIBUTIONS

```
# Create loaded dice.
>>> from scipy.stats import rv_discrete
>>> xk = [1, 2, 3, 4, 5, 6]
>>> pk = [0.3, 0.35, 0.25, 0.05,
        0.025, 0.025]
>>> new = rv_discrete(name='loaded',
                      values=(xk,pk))
```

```
# Calculate histogram
>>> samples = new.rvs(size=1000)
>>> bins = linspace(0.5, 6.5, 7)
>>> subplot(211)
>>> hist(samples, bins=bins, normed=True)
```

```
# Calculate pmf
>>> x = range(0, 8)
>>> subplot(212)
>>> stem(x,new.pmf(x))
```



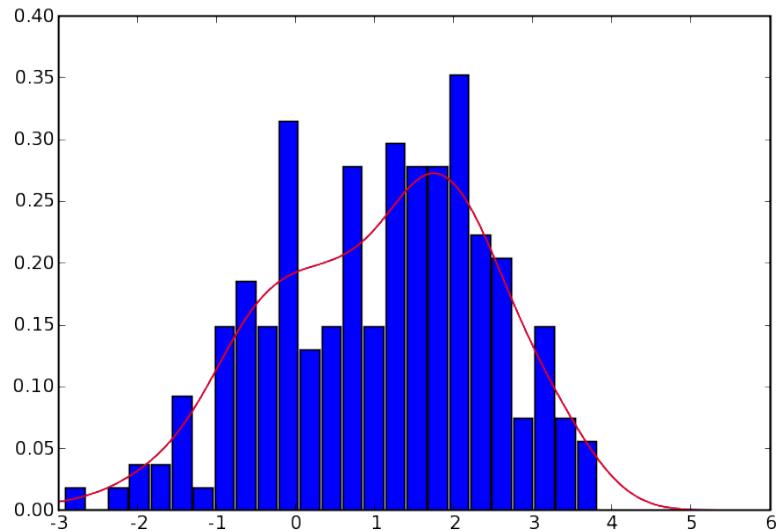
# Statistics

## CONTINUOUS DISTRIBUTION ESTIMATION USING GAUSSIAN KERNELS

```
# Sample two normal distributions
# and create a bimodal distribution
>>> rv1 = stats.norm()
>>> rv2 = stats.norm(2.0, 0.8)
>>> samples = hstack([rv1.rvs(size=100),
                     rv2.rvs(size=100)])

# Use a Gaussian kernel density to
# estimate the PDF for the samples.
>>> from scipy.stats.kde import gaussian_kde
>>> approximate_pdf = gaussian_kde(samples)
>>> x = linspace(-3, 6, 200)

# Compare the histogram of the samples to
# the PDF approximation.
>>> hist(samples, bins=25, normed=True)
>>> plot(x, approximate_pdf(x), 'r')
```



# Data Analysis with pandas

# Pandas

DATA SCIENCE

Pandas is a library that makes the analysis of complex, tabular datasets *easy*.

## FEATURES

- Defines **tabular data types**: database-like tables, with labelled rows and columns;
- **Data consolidation and data integration**: remove duplicates, clean data, manage missing values; automatically align tables by index;
- **Summarization**: create “pivot” tables;
- **In-memory, SQL-like operations**: join, aggregate (group by);
- Very flexible **import/export** of data;
- **Date and time** handling built-in, including timezones;
- **Easy visualization** based on Matplotlib.

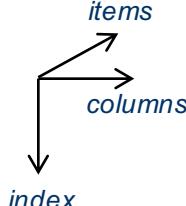
# Storing data in Pandas

# New Data Structures

PANDA = PANel DAta = multi-dimensional data in stats & econometrics.

Introduces 3 size-mutable, labeled data-structures:

- A `Series` is a 1D data-structure.
- A `DataFrame` is a 2D data-structure that can be viewed as a dictionary of `Series`.
- A `Panel` is a 3D data-structure that can be viewed as a dictionary of `DataFrames`.



The diagram illustrates a 3D Panel data structure. It is represented as a cube divided into a grid of cells. The vertical axis is labeled "index" at the bottom, with values 1 through 6. The horizontal axis is labeled "columns" at the top left, with labels "Rk", "Fst N", "Lst N", "DoB", "Gend", "Grade", and "offset". The depth axis is labeled "items" at the top left, with labels "TAMU" and "UT". The data in the cells is as follows:

		Rk	Fst N	Lst N	DoB	Gend	Grade	offset
1	UT	4	John	Doe	1981	M	4.18	-1.18
2	TAMU	2	Jill	Ford	1975	F	5.26	1.26
3	UT	5	Chris	Jones	Nan	M	3.91	0.91
4	TAMU	6	Dave	Smith	1965	M	1.23	Nan
5	UT	1	Ellen	Frank	1973	F	5.52	0.52
6	TAMU	3	Frank	Hart	1976	M	4.71	-0.71

# Series

## DEFINITION

Conceptually, `pandas.Series` are indexed arrays:

- NumPy arrays map a range of integers to values
- Series map arbitrary sets of labels to values
- Series may also be seen as a specialized, ordered dictionary where values all have the same type and are stored efficiently

```
>>> from pandas import *
>>> s = Series({'a':0,'b':1,'c':2,'d':3})
# Dict-like access can be label-based
>>> s['b']
1
```

The labels are accessed via the `s.index` attribute and the values by the `s.values` attribute (NumPy array).

# Creating Series

## FROM LIST AND DICT

```
# Data and corresponding indices can
# be stored in lists.
>>> index = ['a', 'b', 'c', 'd']
>>> Series(range(4), index=index,
           name='first series')
a    0
b    1
c    2
d    3
Name: first series
# data + indices in a dict
>>> d = {'a':0,'b':1,'c':2,'d':3}
>>> s = Series(d, name='first series')
>>> s.index
Index([a, b, c, d], dtype=object)
>>> s.values, type(s.values)
array([0, 1, 2, 3], dtype=int64)
numpy.ndarray
>>> s.dtype
dtype('int64')
```

## FROM A NUMPY ARRAY

```
>>> from numpy.random import randn
>>> Series(randn(4), index=index)
a    -1.062984
b    -0.961625
c    -0.720323
d     0.336753
```

## ACCESS OR ADD ELEMENTS

```
# Request existing values
>>> s['b']
1
# Modify an existing value
>>> s['b'] = 3
# Add new elements
>>> s['e'] = 5
>>> s
a    0
b    3
c    2
d    3
e    5
```

# Dataframes

## DEFINITION

A DataFrame object can be viewed as a dictionary of Series sharing a common index:

- Dataframes have both row (`index`) and column (`columns`) indices
- Each column may have a different type
- Adding a column is ‘cheap’

```
>>> s1 = Series({'a': 1, 'b': 2, 'c': 3})
>>> s2 = Series({'a': True, 'b': False, 'c': True})
>>> df = DataFrame({'col1': s1, 'col2': s2})
# Dict-like access is column-based
>>> df['col1']
a    1
b    2
c    3
Name: col1, dtype: int64
```

# Creating DataFrames

## FROM A DICT OF SERIES

```
# DF from a dict of series: keys are
# column names.
>>> s2=Series([-0.9, -1.7, 1.1],
             index=index[1:])
>>> d = {'A':s, 'B':s2}
>>> df = DataFrame(d)

          A      B
a    0   NaN
b    1   -0.9
c    2   -1.7
d    3    1.1

>>> df.index, df.columns
Index([a, b, c, d], dtype=object)
Index([A, B], dtype=object)
>>> df.shape, df.dtypes
(4, 2)
A           int64
B        float64
>>> df.values
array([[ 0. ,  nan],
       [ 1. , -0.9],
       [ 2. , -1.7],
       [ 3. ,  1.1]])
```

## FROM A NUMPY ARRAY

```
>>> DataFrame(randn(4,4), index=index,
              columns=['A','B','C','D'])
          A      B      C      D
a  0.28164 -0.36826  0.04011  1.25030
b -0.71049 -1.23956 -0.08504 -0.08336
c -1.29446  0.70709  1.39642  0.49035
d  0.74632 -0.03512 -0.69237  0.81488
```

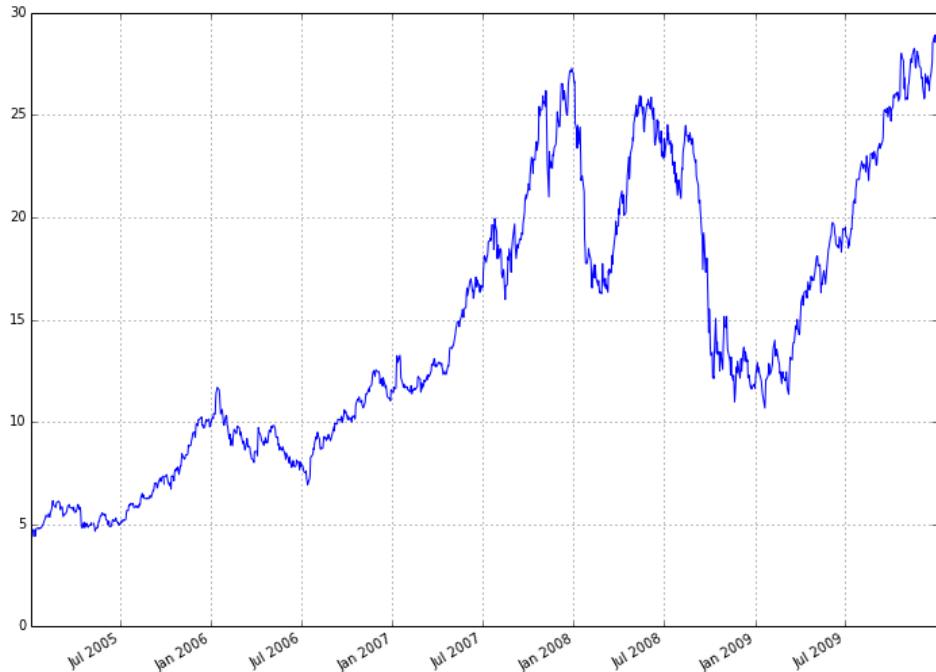
## ACCESS OR ADD COLUMNS

```
# Columns accessed like a dict...
>>> col1 = df['A']
# Create a new column
>>> df['Flag'] = df['B'] > 0
>>> df
          A      B   Flag
a    0   NaN  False
b    1   -0.9  False
c    2   -1.7  False
d    3    1.1   True
```

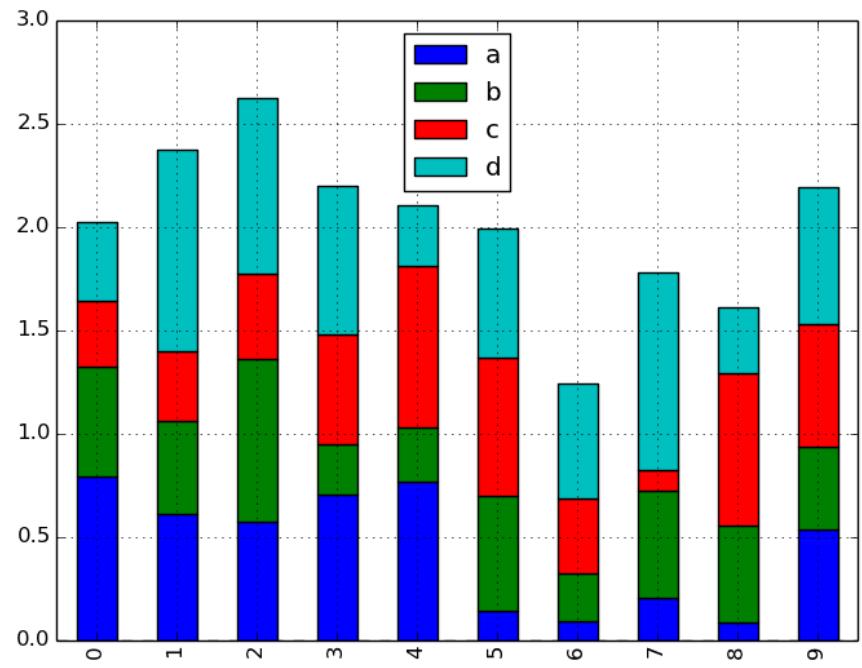
# Easy-to-use visualization

Pandas provides easy-to-use visualization methods.

```
>>> ts = df['AAPL']  
>>> ts.plot()
```



```
>>> df2 = DataFrame(rand(10,4),  
                    columns=list('abcd'))  
>>> df2.plot(kind='bar',  
                     stacked=True)
```



See the `pandas_plotting/ demo`.

# Accessing values in Pandas: Indexing

# Pandas and indexing

Series and DataFrames have powerful indexing capabilities:

- Values are accessible as NumPy arrays
- More interestingly: label-based indexing
- Indices allow automatic alignment: especially interesting with timeseries, and for NaN (missing data) handling (more on this later)

Essentially:

- Series[label] -> scalar
- Dataframe[label] -> column

```
>>> s = Series({'a': 0, 'b': 1,
   ...: 'c': 2})
>>> s['a']
0
>>> df = DataFrame({'A': s, 'B': -s})
>>> df['A']
a    0
b    1
c    2
```

```
# BUT if you do slicing
>>> df[:2] # first two rows !!
   A   B
a  0  0
b  1 -1
```

# Pandas and indexing (Cont.)

## LABEL-BASED VS POSITION-BASED INDEXING

Indexing operator [ ] has an ambiguity:

- Series[integer\_value]: position or label?
- DataFrame[integer\_value]: position or column name ?

API is a bit messy here, greatly improved in versions  $\geq 0.11.0$ :

- .loc attribute: purely “label”
- .iloc attribute: purely index-based, aka position (integer value)

```
>>> s = Series({'a': 0, 'b': 1, 'c': 2})
>>> s.iloc[1]
1
>>> s.iloc['a']
TypeError: the label [a] is not a proper indexer for this index ...
>>> s.loc['a']
0
>>> s.loc[0]
KeyError: 'the label [0] is not in the [index]'
```

# Indexing into Series

DATA SCIENCE

## ACCESSING 1 ELEMENT

```
>>> index = ['a', 'b', 'c', 'd']
>>> s = Series(range(4), index=index)
# Access elements based on position
>>> s.iloc[2]
2
# Access elements based on label
>>> s.loc['c']
2
# Indexing into a Series is equivalent
>>> s['c']
2
```

## SLICING ELEMENTS OUT

Form: `s.iloc[pos_lower:pos_upper:step]`

```
>>> s.iloc[:2]
a    0
b    1
# Every other element
>>> s.iloc[::2]
a    0
c    2
```

Form: `s.loc[label_lower:label_upper:step]`  
`>>> s.loc['a':'c']`

a	0
b	1
c	2 # upper limit included!



## FANCY-INDEXING

# Custom selection of elements  
`>>> s[[True, False, True, True]]`

a	0
c	2
d	3

# Masks can be created by comparing  
# values in the Series or another one  
`>>> s>1`

a	False
b	False
c	True
d	True
>>> s[s>1]	
c	2
d	3

# Indexing into DataFrames

## ACCESS ELEMENTS

```

>>> df
      A      B
a  0    NaN
b  1   -0.9
c  2   -1.7
d  3    1.1
# 1 (or more) column accessed like a
# dict...
>>> df['A']
a  0
b  1
c  2
d  3
... or like an object
>>> series2 = df.B
# Access all columns for 1 index
>>> df.loc['c']
A          2
B         -1.7
Name: c, dtype: float64
# or 1 element of the table
>>> df.loc['c','B']
-1.7
  
```

## SLICING ELEMENTS OUT

Form: `s.loc[row lower:row upper:step,  
 col lower:col upper:step]`

```

>>> sub_df = df.loc["c":, "A":"B"]
# Incomplete slicing assumes all
# elements in other dimensions.
>>> df.loc["c":]
      A      B
c  2   -1.7
d  3    1.1
  
```

## MIXED INDEXING

Mixed indexing using `.ix`:

```

>>> sub_df = df.ix[2, "B"]
-1.7
  
```

# Re-indexing

The index of a Pandas data-structure is the key that controls:

- how the data is displayed and ordered,
- how to align and combine different datasets.

The index can be:

- shuffled (and the values will follow),
- overwritten,
- transformed,
- set to the values of any of the columns of a DataFrame ,
- made of multiple sub-indices.

# Re-indexing Series

## RE-INDEXING

```
>>> index = ['a', 'b', 'c', 'd']
>>> s = Series(range(4), index=index)

# Select a different set of indices
>>> s.reindex(['c', 'b', 'a', 'e'])
c    2
b    1
a    0
e    NaN

# Sort by values. See s.sort_index()
# to sort based on index value.
>>> s.order(ascending=False)
d    3
c    2
b    1
a    0
```

## ALIGNMENT OF 2 SERIES

```
>>> s = Series(range(4), index=index)
>>> s2 = s.iloc[:-2]

>>> s2
a    0
b    1

# Operations automatically align on
# the index (different from NumPy)
>>> s + s2
a    0
b    2
c    NaN
d    NaN
```

# Re-indexing DataFrames

## RE-INDEXING DATAFRAMES

```
>>> df
      A      B   flags
a  0    NaN  False
b  1 -0.9  False
c  2 -1.7  False
d  3  1.1   True
>>> df.reindex(['c', 'a', 'b'])
      A      B   flags
c  2 -1.7  False
a  0    NaN  False
b  1 -0.9  False
# Sort a DF by a (list of) column(s)
>>> df.sort('B')
      A      B   flags
c  2 -1.7  False
b  1 -0.9  False
d  3  1.1   True
a  0    NaN  False
```

## INDEX TO/FROM A COLUMN

```
# Set dataframe column as index
>>> df2 = df.set_index('A')
>>> df2
      B   flags
A
0    NaN  False
1 -0.9  False
2 -1.7  False
3  1.1   True
# Opposite operation
>>> df2.reset_index()
      A      B   flags
0  0    NaN  False
1  1 -0.9  False
2  2 -1.7  False
3  3  1.1   True
```

# Dealing with date & time

## CREATING DATE/TIME INDEXES

```
# The index can be a list of
# dates+times locations that can be
# automatically generated
>>> date_range('1/1/2000', periods=4)
<class
'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-04
00:00:00]
Length: 4, Freq: D, Timezone: None
# Specify frequency: us,ms,S,T,H,D,B,
# W,M,3min, 2h20min, 2W,...
>>> r=date_range('1/1/2000', periods=72,
...     freq='H')
>>> i=date_range('1/1/2000', periods=4,
...     freq=datetools.YearEnd())
>>> i=date_range('1/1/2000', periods=4,
...     freq='3min')
>>> ts=Series(range(4), index=i)
2000-01-01 00:00:00    0
2000-01-01 00:03:00    1
2000-01-01 00:06:00    2
2000-01-01 00:09:00    3
Freq: 3T
```

## UP-/DOWN-SAMPLING

```
>>> ts.resample('T')
2000-01-01 00:00:00    0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    NaN
2000-01-01 00:03:00    1
2000-01-01 00:04:00    NaN
2000-01-01 00:05:00    NaN
2000-01-01 00:06:00    2
2000-01-01 00:07:00    NaN
2000-01-01 00:08:00    NaN
2000-01-01 00:09:00    3
Freq: T

# Group hourly data into daily
>>> ts2 = Series(randn(72), index=r)
>>> ts2.resample('D', how='mean',
...     closed='left', label='left')
01-Jan-2000    0.397501
02-Jan-2000    0.186568
03-Jan-2000    0.327240
Freq: D
```

# Dealing with date & time II

## TIME ALIGNMENT

```
# Data alignment based on time is one
# of Panda's most celebrated features
>>> daily = date_range('2000-01-01',
...     freq='D', periods=5)
>>> df = DataFrame(random.rand(5),
...     index=daily, columns=['A'])
>>> df
              A
2000-01-01  0.954140
2000-01-02  0.511243
2000-01-03  0.979188
2000-01-04  0.793727
2000-01-05  0.238190
>>> bidaily = pd.date_range('2000-01-01',
...     freq='2D', periods=3)
>>> df2 = pd.DataFrame(np.random.rand(3),
...     index=bidaily, columns=['B'])
>>> df2
              B
2000-01-01  0.007215
2000-01-03  0.797108
2000-01-05  0.440173
```

## TIME ALIGNMENT (CONT.)

```
>>> concat([df, df2], axis=1)
              A          B
2000-01-01  0.954140  0.007215
2000-01-02  0.511243      NaN
2000-01-03  0.979188  0.797108
2000-01-04  0.793727      NaN
2000-01-05  0.238190  0.440173
```

# Dealing with missing data

# Dealing with missing data

## PANDAS PHILOSOPHY

- To signal a missing value, Pandas stores a NaN (Not a Number) value defined in NumPy (`np.nan`).
- Unlike other packages (like NumPy), most operators in Pandas will ignore NaN values in a Pandas datastructure.

```
>>> import numpy as np  
>>> a = np.array([1,2,3,np.nan])  
>>> a.sum()  
nan  
>>> s = Series(a)  
>>> s.sum()
```

# Dealing with missing data

## FIND MISSING VALUES

```
>>> df
      s1    s2
a      1    NaN
b    NaN    NaN
c      3   3.5
d      4   4.5

# Boolean mask for all null values:
# np.nan and None .
# Use notnull method for the inverse
>>> df.isnull()
      s1    s2
a  False  True
b  True  True
c  False False
d  False False
```

## REMOVE/REPLACE NaN

```
# Replace missing values manually
>>> df[isnull(df)] = 0.
```

DATA SCIENCE

```
# Inverse operation
>>> df[df == 0] = np.nan
# Fill na from previous value
>>> df.fillna(method='ffill')
      s1    s2
a      1    NaN
b      1    NaN
c      3   3.5
d      4   4.5

# Remove all rows w/ missing values
>>> df.dropna(how='all')
      s1    s2
a      1    NaN
c      3   3.5
d      4   4.5

>>> df.dropna(how='any')
      s1    s2
c      3   3.5
d      4   4.5

# Interpolate NaNs away
>>> df.interpolate()
      s1    s2
a      1    NaN
b      2    NaN
c      3   3.5
d      4   4.5
```

# Computations and statistics

# Computations with DataFrames

**Rule 1:** Mathematical operators (+ - \* / exp, log, ...) apply element by element, on the values.

**Rule 2:** Reduction operations (mean, std, skew, kurt, sum, prod, ...) are applied column by column.

**Rule 3:** Operations between multiple Pandas object implement auto-alignment based on index first.

# Computations with Pandas

```
# Computations are applied
# column-by-column
>>> df
      A      B   Flags
a  0    NaN  False
b  1   -0.9  False
c  2   -1.7  False
d  3    1.1   True

>>> df.sum()
A      6.0
B     -1.5
Flags  1.0
dtype: float64

# Adding a series or re-scaling
>>> row = df.iloc[1]
>>> df - row
      A      B   Flag
a  -1    NaN  False
b   0     0  False
c   1   -0.8  False
d   2    2.0   True
```

## DATAFRAME REDUCTION

```
# 'apply' a custom function to
# columns. The function receives a
# column (Series) and returns a value
>>> f = lambda x: x.max() - x.min()
>>> df.apply(f, axis=0)
A      3.0
B      2.8
Flags  1.0
```

## DATAFRAME TRANSFORMATION

```
# applymap is similar but receives a
# value and return a value.
>>> df.applymap(lambda x: len(str(x)))
      A      B   Flags
a   1     3      5
b   1     4      5
c   1     4      5
d   1     3      4
```

# Statistical Analysis

## DESCRIPTIVE STATS

```
>>> df
      A          B   Flag
a  0       NaN  False
b  1     -0.9  False
c  2    -1.7  False
d  3     1.1   True

# Descriptive stats available:
# count, sum, mean, median, min, max,
# abs, prod, std, var, skew, kurt,
# quantile, cumsum, cumprod, cummax
# Stats on DF are column per column

>>> df.mean()
A        1.50
B       -0.50
flag     0.25

>>> df.mean(axis=1)
a    0.000000
b    0.033333
c    0.100000
d    1.700000

# min/max location (Series only)
>>> df['B'].argmin()
'c'
```

```
>>> df.describe()
```

	A	B	Flag
count	4.000000	3.000000	4
mean	1.500000	-0.500000	0.25
std	1.290994	1.442221	0.5
min	0.000000	-1.700000	False
25%	0.750000	-1.300000	0
50%	1.500000	-0.900000	0
75%	2.250000	0.100000	0.25
max	3.000000	1.100000	True

## WINDOWED STATS

```
# The same descriptive stats are
# available as "rolling stats":
# For example,
>>> t = s.rolling(window=20).mean()

# Custom function on ndarray supported
# with .apply:
>>> f = lambda x: x[1: -1].mean()
>>> x.rolling(20).apply(f)
```

# Correlations

## CORRELATIONS

```
# Correlation of Series
>>> ts.corr(ts2)
0.06666666666666693

# Pair-wise correlations of the columns.
# Optional argument: 'method', one of
# {'pearson', 'kendall', 'spearman'}
>>> corr_matrix = df.corr()

# Pair-wise covariance of the columns
>>> cov_matrix = df.cov()
```



For more stats, see `statsmodels`.

# DAY 5

# Data Filtering and Aggregation

# Split, apply and combine

## RATIONALE

It is often necessary to apply different operations on different subgroups

- Traditionally handled by SQL-based systems
- Pandas provides in-memory, sql-like set of operations

General ‘framework’: split, apply, combine (Hadley Wickham, R programmer):

- Splitting the data into groups (based on some criterion, e.g. column value)
- Applying a function to each group independently
- Combine the results back into a data structure (e.g. dataframe)

# Data aggregation: Split

DATA SCIENCE

## SPLIT WITH groupby()

```
>>> df
      A      B   Flag
a  0    NaN  False
b  1 -0.9  False
c  2 -1.7  False
d  3  1.1   True
e  4  0.5   True
# Group data by one column's value
>>> gb = df.groupby('Flag')
# gb is a groupby object
>>> gb.groups
{False: ['a', 'b', 'c'], True: ['d', 'e']}
# gb = iterator of tuples with
# group name and sub part of df
>>> for value, subdf in gb:
        print value
        print subdf
```

	A	B	Flag
a	0	NaN	False
b	1	-0.9	False
c	2	-1.7	False
			True
d	3	1.1	True
e	4	0.5	True

# Displays a subplot per group.

```
>>> gb.boxplot(column=["A", "B"])
```

## groupby() ON THE INDEX

```
>>> df2 = df.reset_index()
>>> even = lambda x: x%2 == 0
>>> gb2 = df2.groupby(even)
>>> gb2.groups
{False: [1, 3], True: [0, 2, 4]}
```

# Data aggregation: Apply

Three ways to apply: `aggregate` (or equivalently `agg`) if each series in each group is turned into one value, `transform` if each series in each group is modified but retains its length, or `apply` in the most general case.

## APPLY WITH `aggregate()` or `agg()`

```
>>> gb.sum()
A      B
Flag
False  3 -2.6
True   7  1.6
# More flexible but slower
>>> summed = gb.aggregate(np.sum)
# Given a list or dict
>>> gb.agg([np.mean, np.std])
A          B
mean       std  mean       std
Flag
False    1.0  1.000000 -1.3  0.565685
True     3.5  0.707107  0.8  0.424264
```

```
>>> gb.agg({'A': 'sum', 'B': 'std'})
```

	A	B
Flag		
False	3	0.565685
True	7	0.424264

## APPLY WITH `transform()`

```
>>> f = lambda x: x - x.mean()
>>> gb.transform(f)
```

	A	B
a	-1.0	NaN
b	0.0	0.4
c	1.0	-0.4
d	-0.5	0.3
e	0.5	-0.3

# Data aggregation II

## APPLY WITH apply()

```
# Computations from values in groups can be turned into a DF of calcs
>>> desc = lambda x: x.describe()
>>> gb['A'].apply(desc).unstack()

      count    mean        std    min    25%    50%    75%    max
Flag
False       3    1.0  1.000000     0    0.50    1.0   1.50     2
True        2    3.5  0.707107     3    3.25    3.5   3.75     4

>>> f = lambda group: DataFrame({'original':group,
                                 'demeaned': group - group.mean()})
>>> gb['A'].apply(f)

      demeaned  original
a         -1.0        0
b          0.0        1
c          1.0        2
d         -0.5        3
e          0.5        4
```

# Combining tables

# Merging

## DEFINITION

pandas.merge connects DataFrames based on one or more keys (close to SQL join).

Let's assume we are running a restaurant, and store customer information and orders coming in in different tables:

```
>>> customers = DataFrame({'id': range(3), 'name': ['john', 'alex', 'lucy']})  
>>> orders = DataFrame({'id': [1, 0, 1, 2], 'order': ['pasta', 'salad',  
                           'coke', 'fries']})
```

Let's now assume we want to connect customer names to their order. We need to use their id to make that connection:

```
>>> merge(customers, orders, on='id')
```

	id	name	order
0	0	john	salad
1	1	alex	pasta
2	1	alex	coke
3	2	lucy	fries

# Merging (Cont.)

## OUTER vs INNER JOINS

```
# Assume a mysterious order comes in
```

```
>>> orders = orders.append({'id': 3, 'order': 'pasta'}, ignore_index=True)
```

```
>>> merge(customers, orders, on='id')
```

	id	name	order
0	0	john	salad
1	1	alex	pasta
2	1	alex	coke
3	2	lucy	fries

```
>>> merge(customers, orders, on='id', how='outer')
```

	id	name	order
0	0	john	salad
1	1	alex	pasta
2	1	alex	coke
3	2	lucy	fries
4	3	NaN	pasta

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

# Data summarization

# Pivot tables

## PIVOTING

```
# Repeating columns can be viewed as
# an additional axis
```

```
>>> df
```

```
      date  variable     value
0  2000-01-03        A  0.469112
1  2000-01-04        A -0.282863
2  2000-01-05        A -1.509059
3  2000-01-03        B -1.135632
4  2000-01-04        B  1.212112
5  2000-01-05        B -0.173215
```

```
>>> df.pivot(index='date',
             columns='variable', values='value')
```

```
variable      A      B
date
2000-01-03  0.469112 -1.135632
2000-01-04 -0.282863  1.212112
2000-01-05 -1.509059 -0.173215
```

# Pivot tables II

## PIVOT\_TABLE

```
# Another way to reshape a DF and
# aggregate at the same time
>>> df
   A   B   C   D
0  foo one small  1
1  foo one large  2
2  foo one large  2
3  foo two small  3
4  foo two small  3
5  bar one large  4
6  bar one small  5
7  bar two small  6
8  bar two large  7

>>> table= df.pivot_table(
    index=['A', 'B'], columns=['C'],
    values='D', aggfunc=np.sum)
>>> table
           small  large
foo  one    1     4
      two    6    NaN
bar  one    5     4
      two    6     7
```

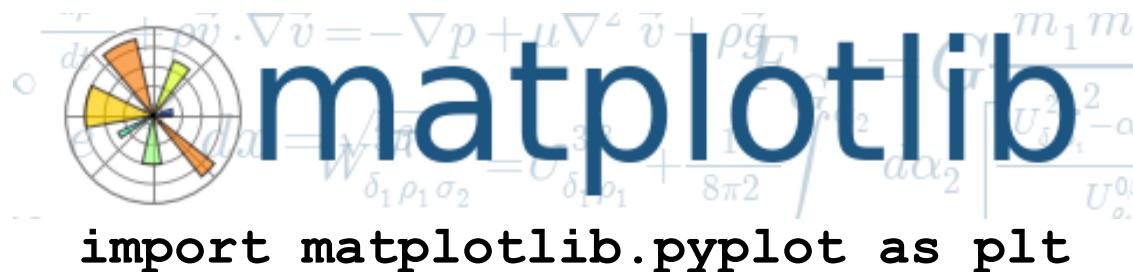
```
# The values arg is optional
>>> df["E"] = randn(9)
>>> df.pivot_table(index=['A', 'B'],
                    columns=['C'])
          D               E
          large  small  large  small
          A   B
bar  one    4     5  1.683667 -1.979804
      two    7     6 -1.790215 -0.595985
foo  one    2     1  1.256463 -0.305674
      two    NaN    3     NaN  1.172797
```

# Visual exploration

# Visually exploring data

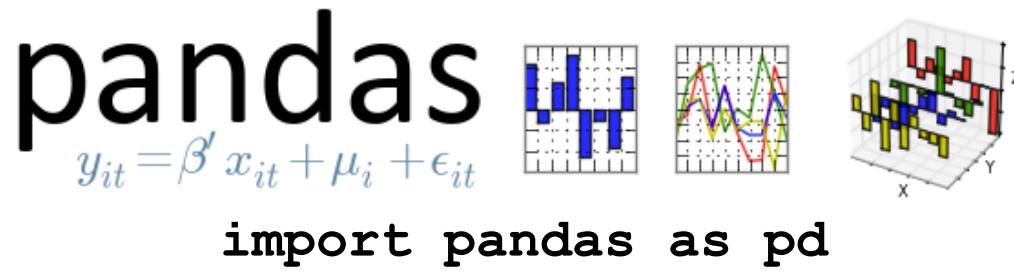
- *Exploratory analysis*: Visually explore data and formulate hypotheses
- Quality control: any obvious problem?
- Typical steps in visual exploration:
  - Look at distribution of variables
  - Look at relations between variables
  - Look at panels of plots with grouped data

# Tools



**matplotlib**

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



**pandas**

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

```
import pandas as pd
```

## Seaborn: statistical data visualization

```
import seaborn as sns
```

# Visualization

data from US government on fuel consumption information  
for different car makes and models

In [5]: `vehicles.head()`

Out[5]:

	barrels08	barrelsA08	charge120	charge240	city08	city08U	cityA08	cityA08U	cityCD	cityE	...	mfrcode	c240Dscr	charge240b	c240bDscr
0	15.689436	0.0	0.0	0.0	19	0.0	0	0.0	0.0	0.0	...	NaN	NaN	0.0	NaN
1	29.950562	0.0	0.0	0.0	9	0.0	0	0.0	0.0	0.0	...	NaN	NaN	0.0	NaN
2	12.195570	0.0	0.0	0.0	23	0.0	0	0.0	0.0	0.0	...	NaN	NaN	0.0	NaN
3	29.950562	0.0	0.0	0.0	10	0.0	0	0.0	0.0	0.0	...	NaN	NaN	0.0	NaN
4	17.337486	0.0	0.0	0.0	17	0.0	0	0.0	0.0	0.0	...	NaN	NaN	0.0	NaN

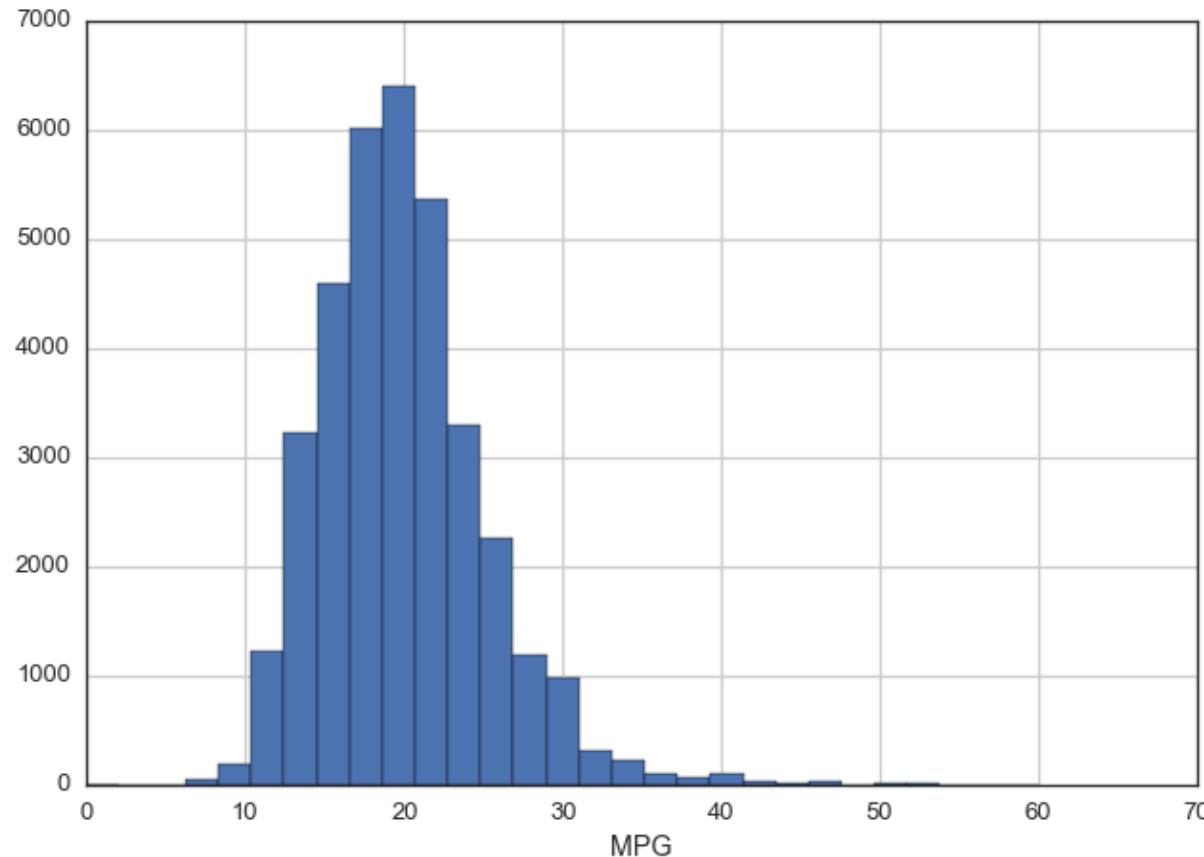
5 rows × 83 columns



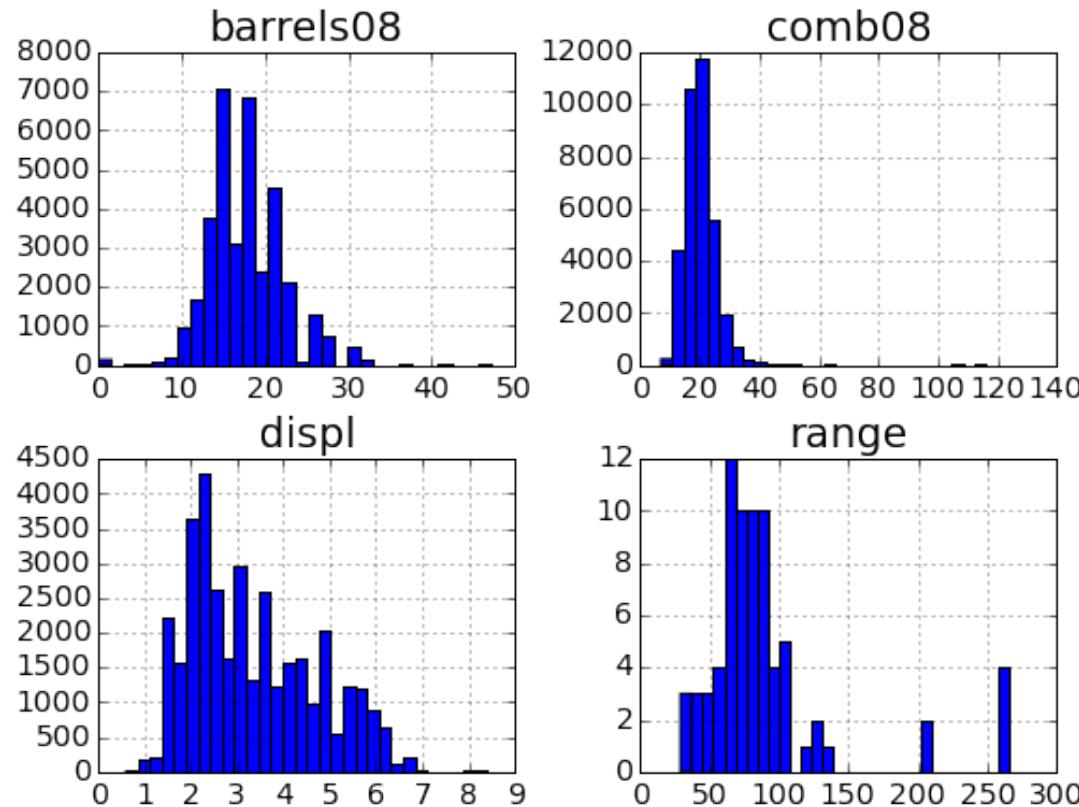
# logit

DATA SCIENCE

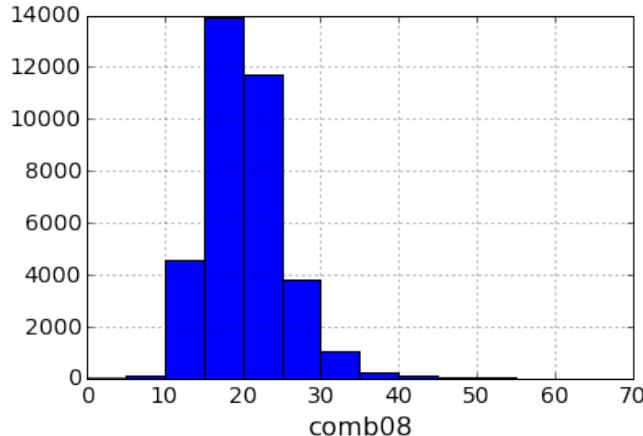
```
vehicles.comb08.hist(bins=np.linspace(0, 60, 30));  
plt.xlabel('MPG')
```



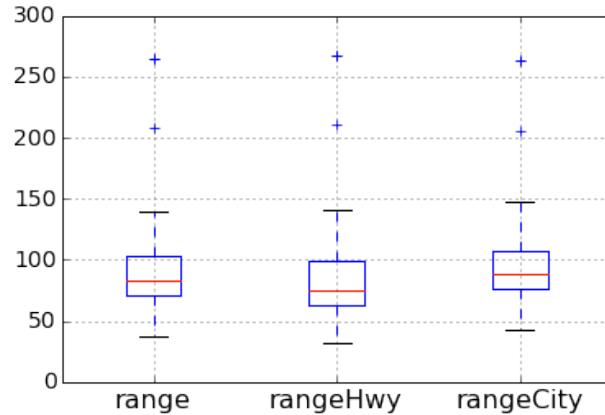
```
vehicles.hist(column=['comb08', 'range', 'barrels08', 'displ'], bins=30)
```



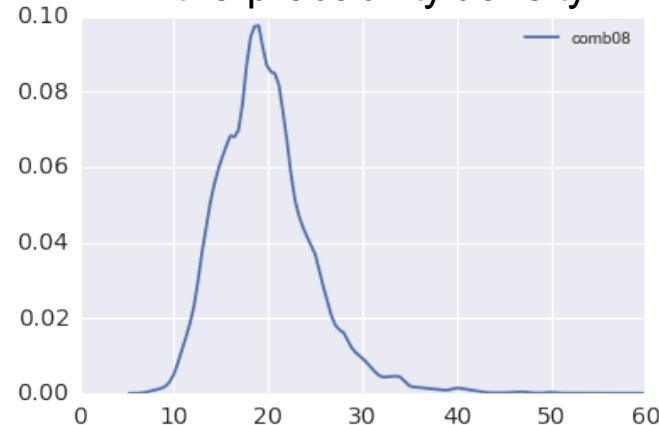
`hist` shows a classical histogram of the data



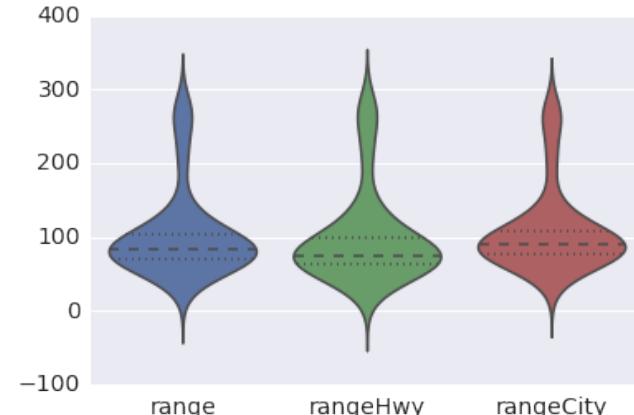
`boxplot` summarizes each data distribution with 5 points



`sns.kdeplot` displays an estimate of the probability density



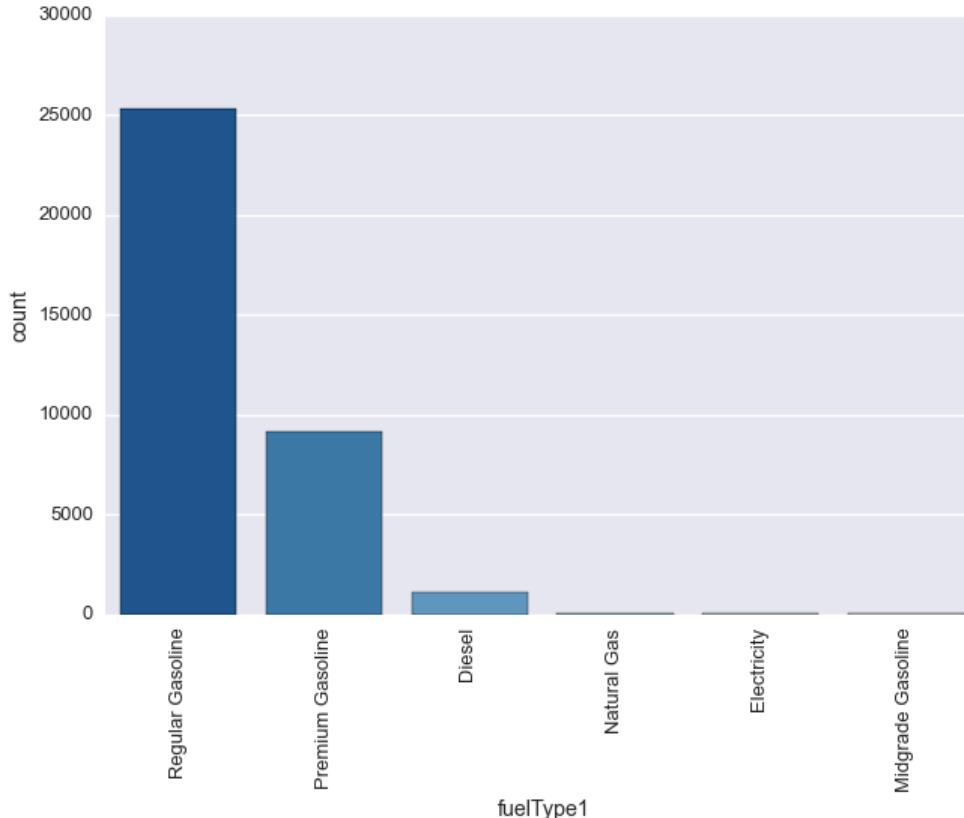
`sns.violinplot` shows an estimate of the shape of the density



# countplot

countplot is used to display the distribution of categorical data.

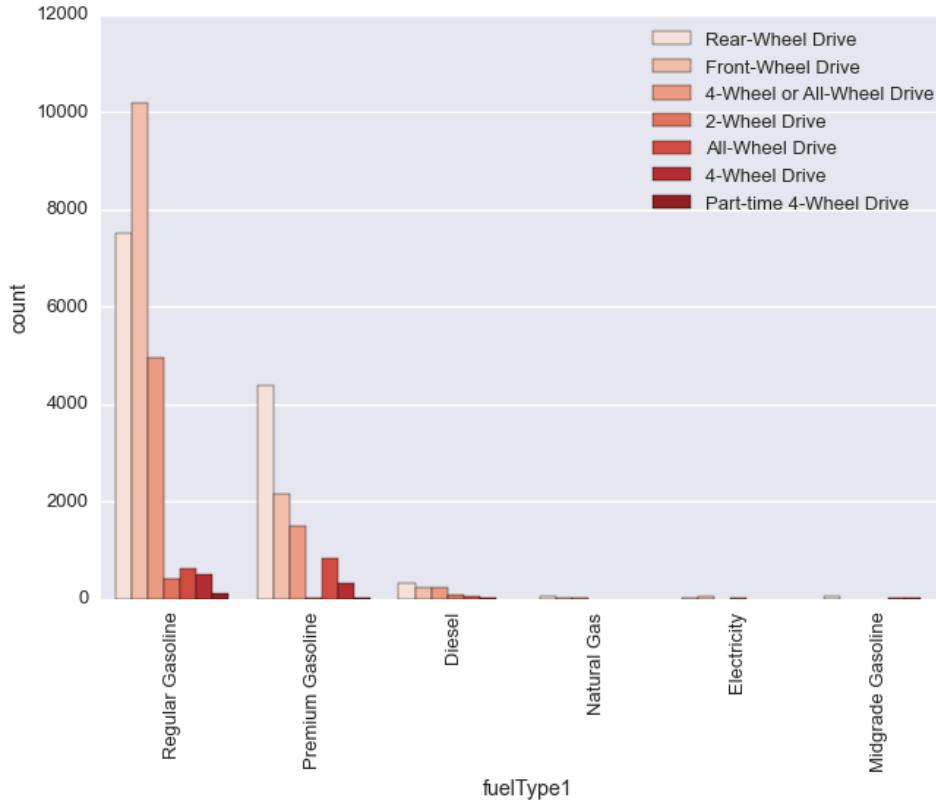
```
sns.countplot('fuelType1', data=vehicles)
```



# countplot

countplot, and many other Seaborn functions, accept a `hue` keyword that subdivides the plots by another categorical variable.

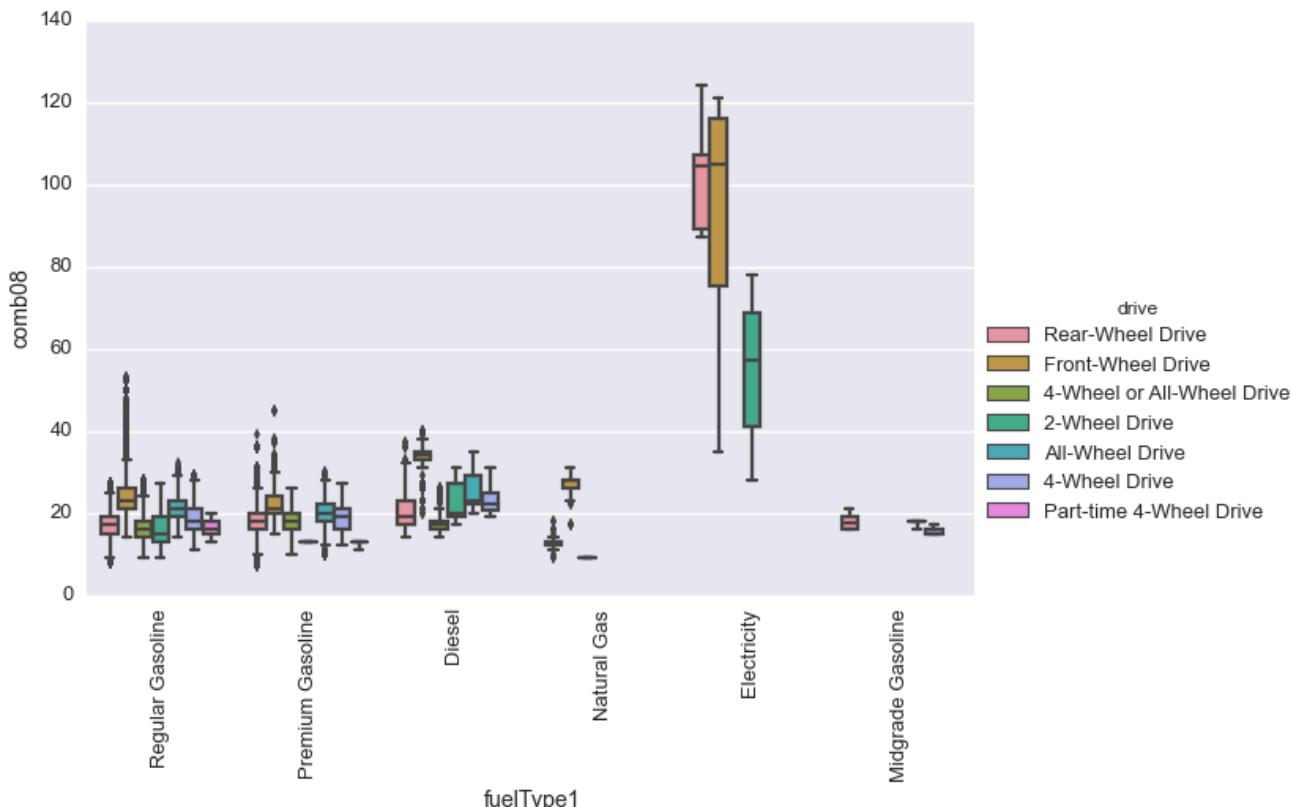
```
sns.countplot('fuelType1', hue='drive', data=vehicles)
```



# factorplot

`factorplot` is a more flexible plot that allows displaying the distribution of any quantity, grouped by a categorical variable.

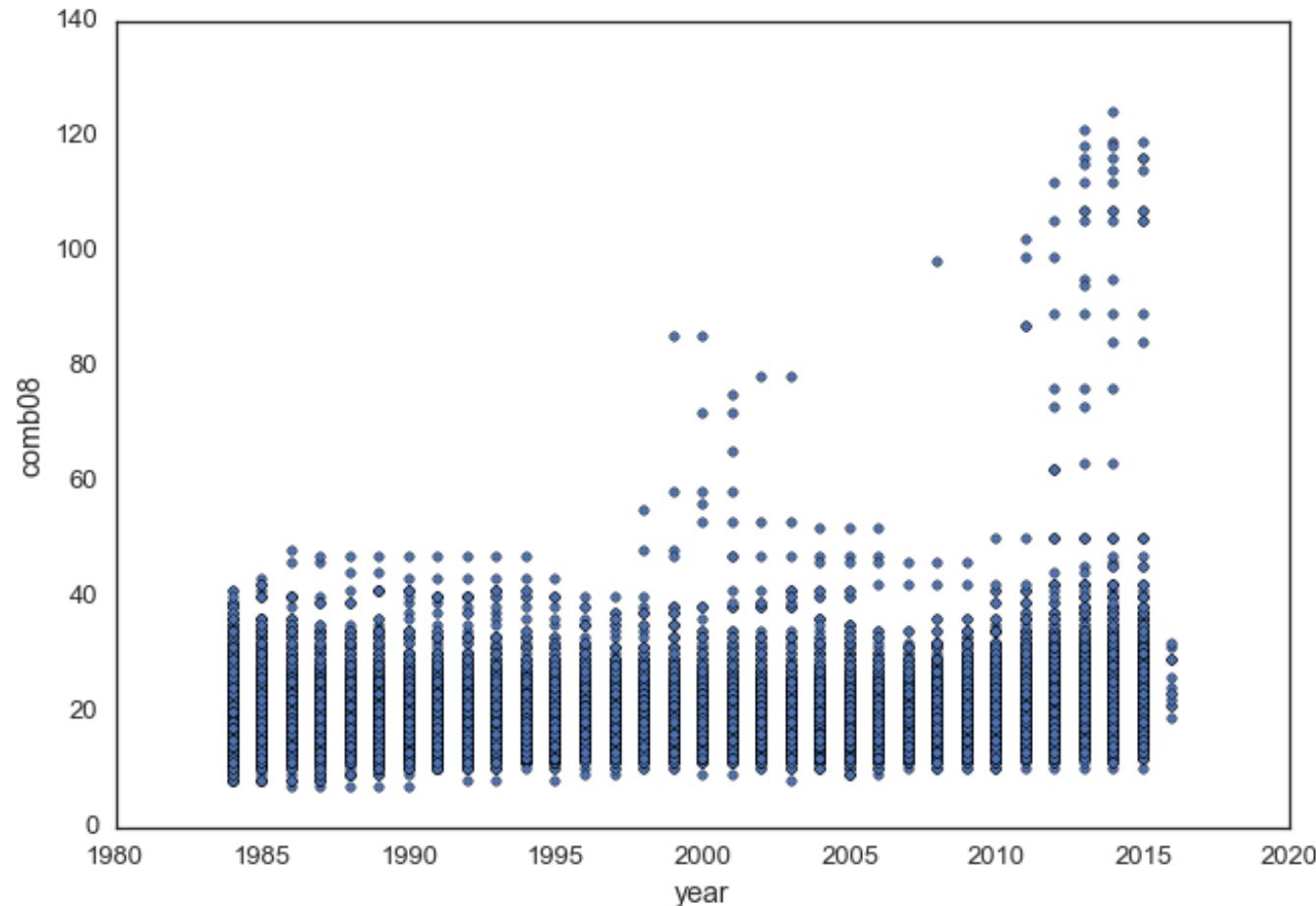
```
sns.factorplot(x='fuelType1', y='comb08', hue='drive',
                 kind='box', data=vehicles)
```



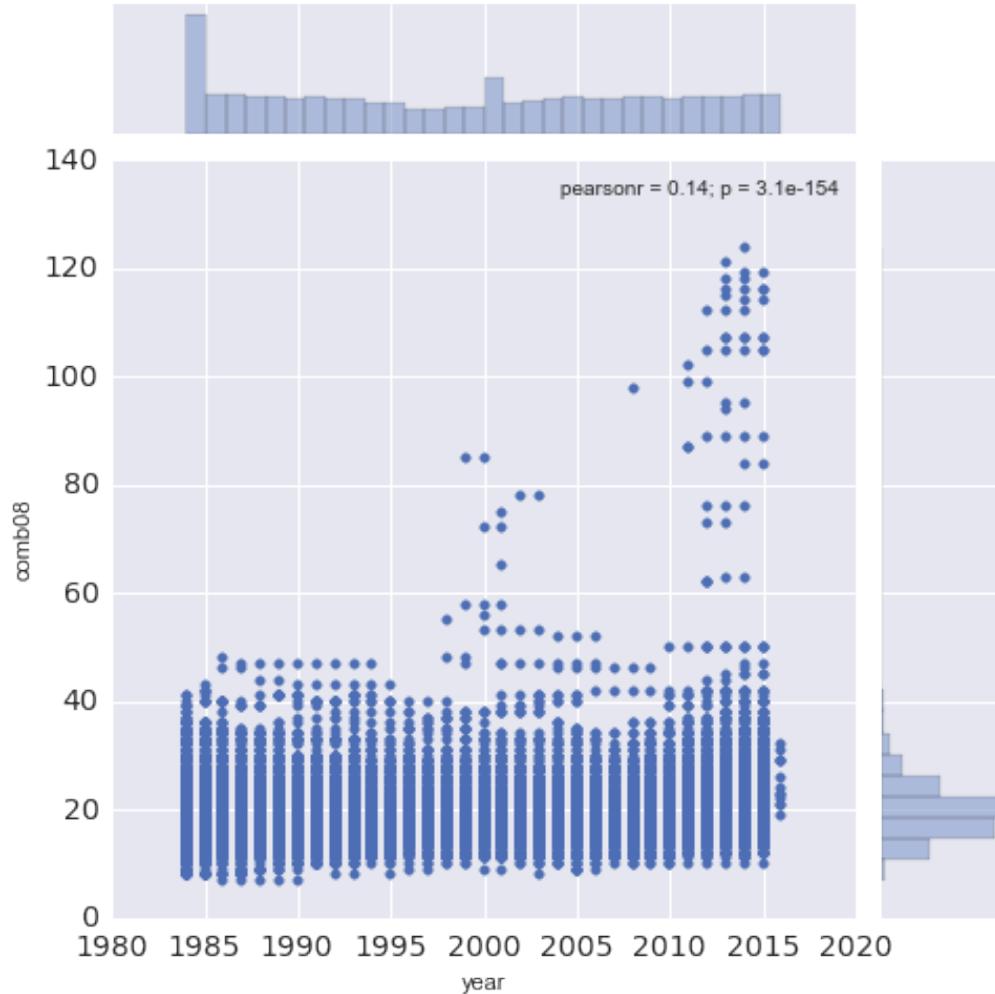
# Joint distributions of variables

DATA SCIENCE

```
vehicles.plot(x='year', y='comb08', kind='scatter')
```

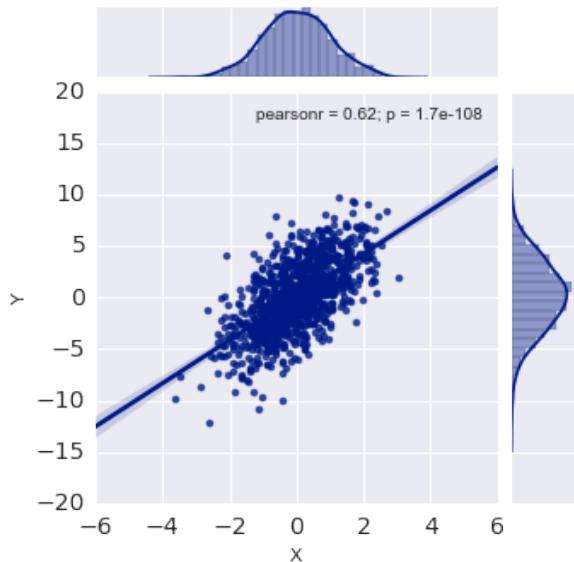


```
sns.jointplot("year", "comb08", data=vehicles)
```

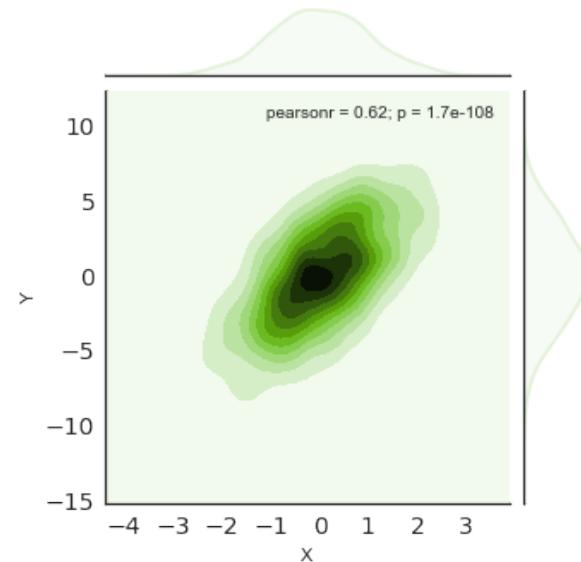


# Jointplot styles

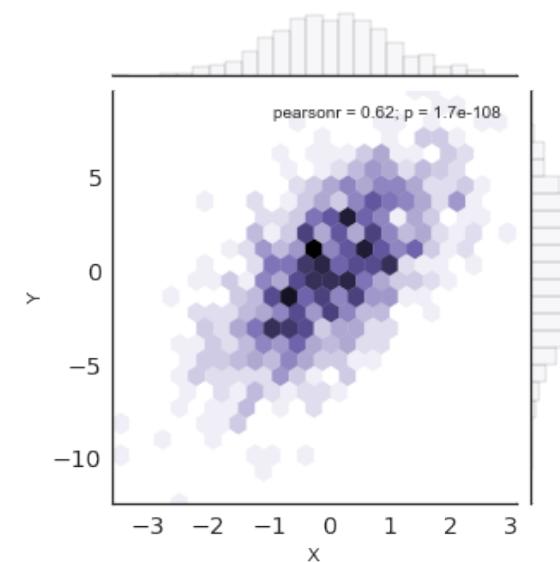
**kind='reg'**



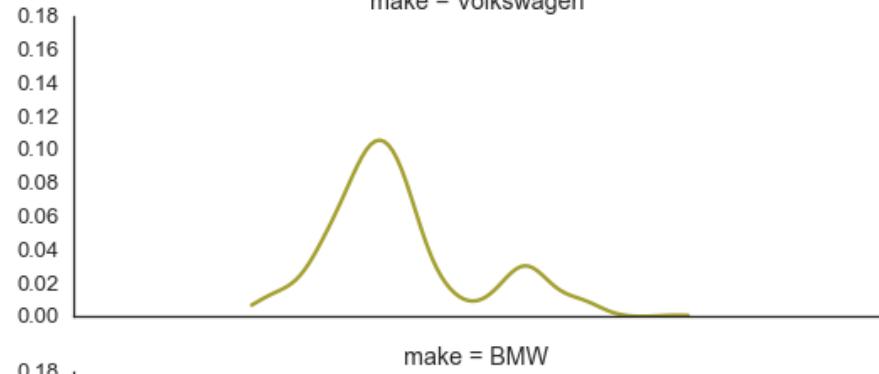
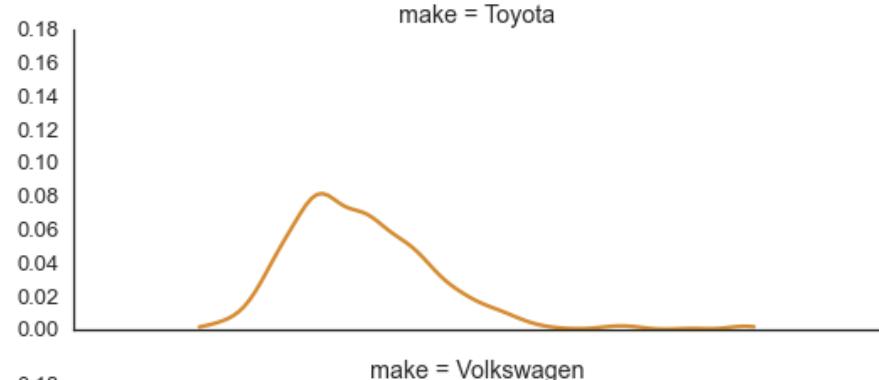
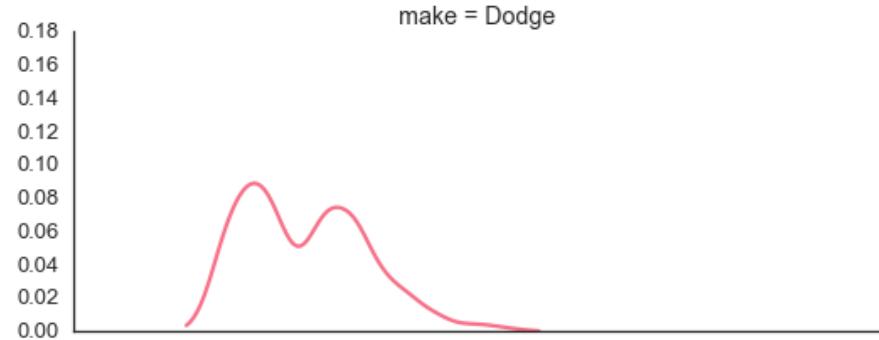
**kind='kde'**



**kind='hex'**



# More distributions of variables



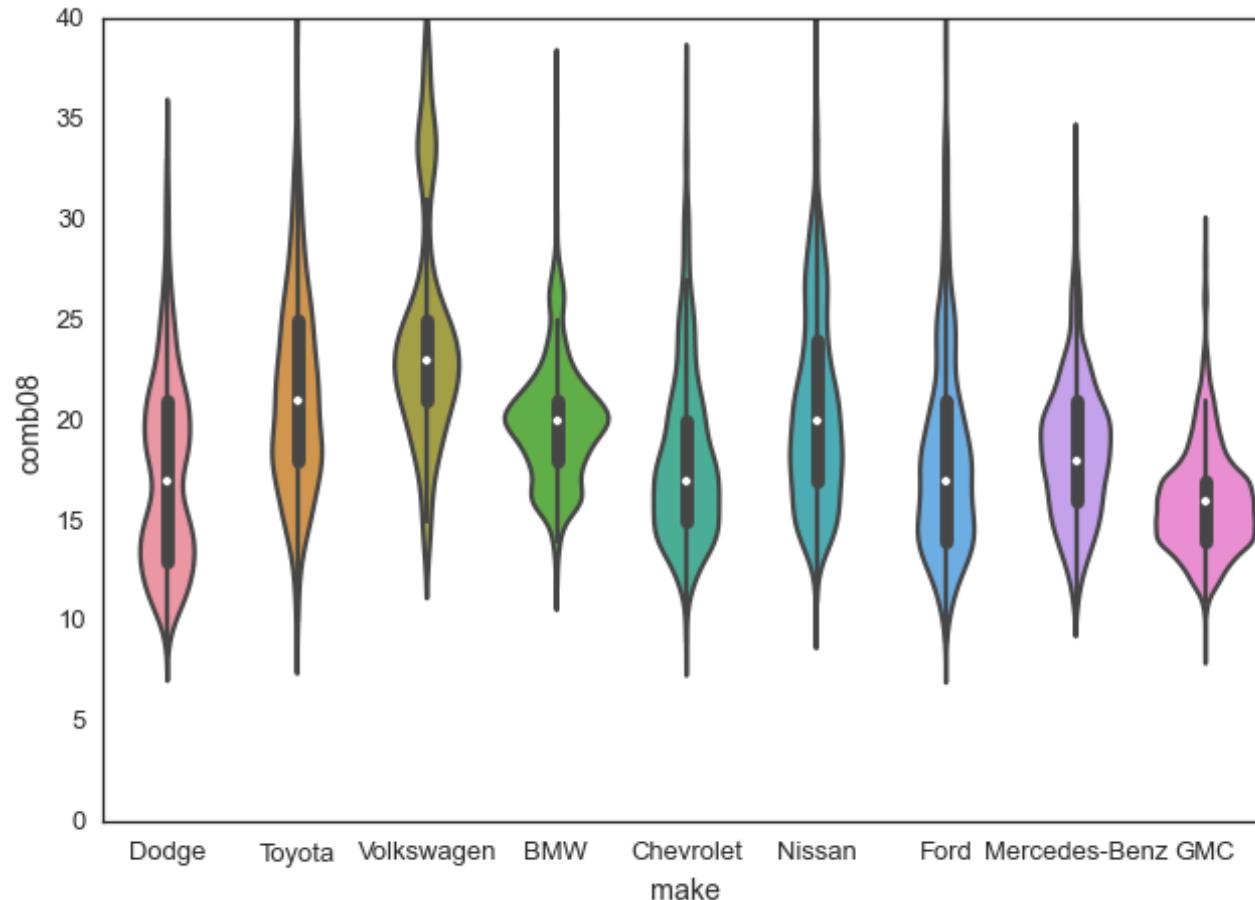
```
g = sns.FacetGrid(vehicles,  
                   row='make', hue='make')
```

```
g.map(sns.kdeplot, 'comb08')
```

# More distributions of variables

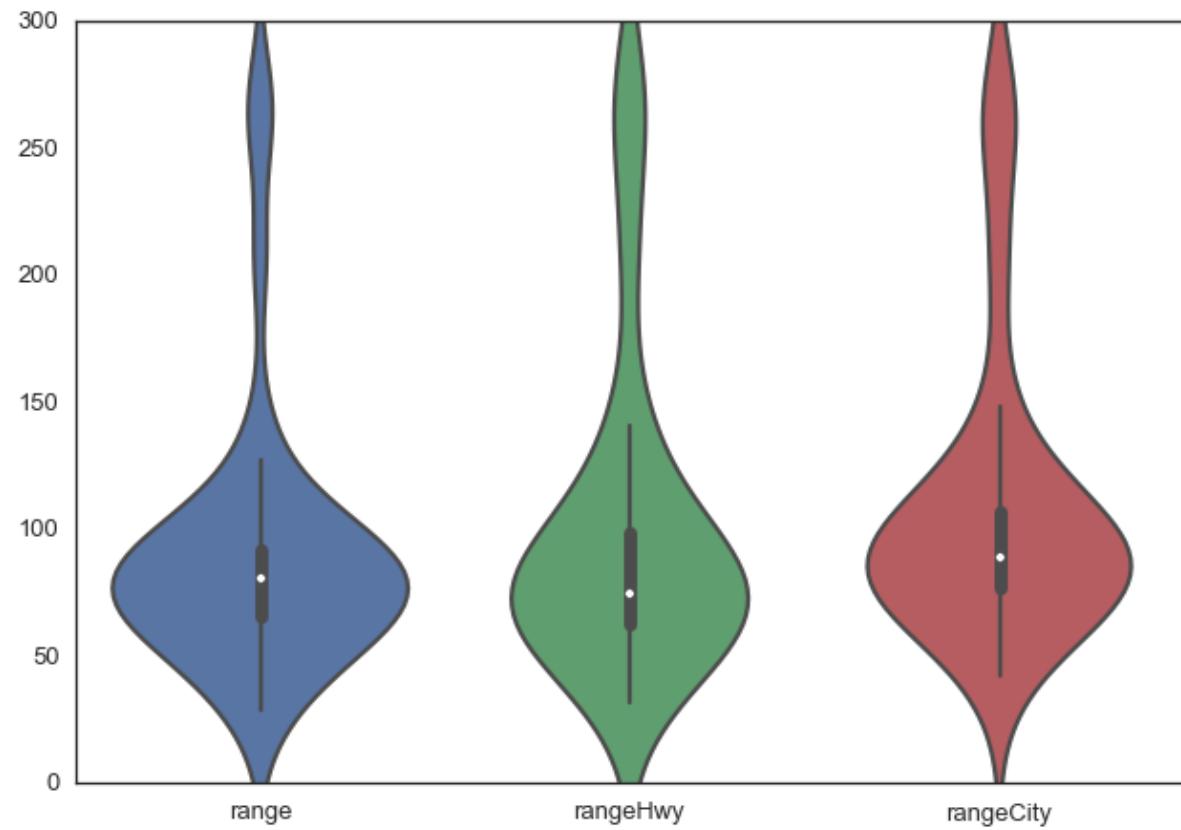
DATA SCIENCE

```
sns.violinplot(x='make' , y='comb08' , data=vehicles) ;  
plt.ylim([0 , 40])
```



# More distributions of variables

```
sns.violinplot(  
    data=vehicles[ ['range' , 'rangeHwy' , 'rangeCity']] );  
  
plt.ylim([0 , 300])
```



# Small multiples

*“At the heart of quantitative reasoning is a single question: Compared to what? Small multiple designs, multivariate and data bountiful, answer directly by visually enforcing comparisons of changes, of the differences among objects, of the scope of alternatives. **For a wide range of problems in data presentation, small multiples are the best design solution.**”*

*Edward Tufte (Envisioning Information, p. 67)*

Key idea: Visualize a grid of plots showing the same information, with data split in multiple groups.

Aliases: Trellis plot, panel chart

# FacetGrid

A FacetGrid object is responsible for creating the grid of plots. Each plot in the grid will contain data grouped by the DataFrame columns corresponding to its row and column:

```
grid = sns.FacetGrid(  
    dataframe, row='col_name',  
    col='col_name', hue='col_name')
```

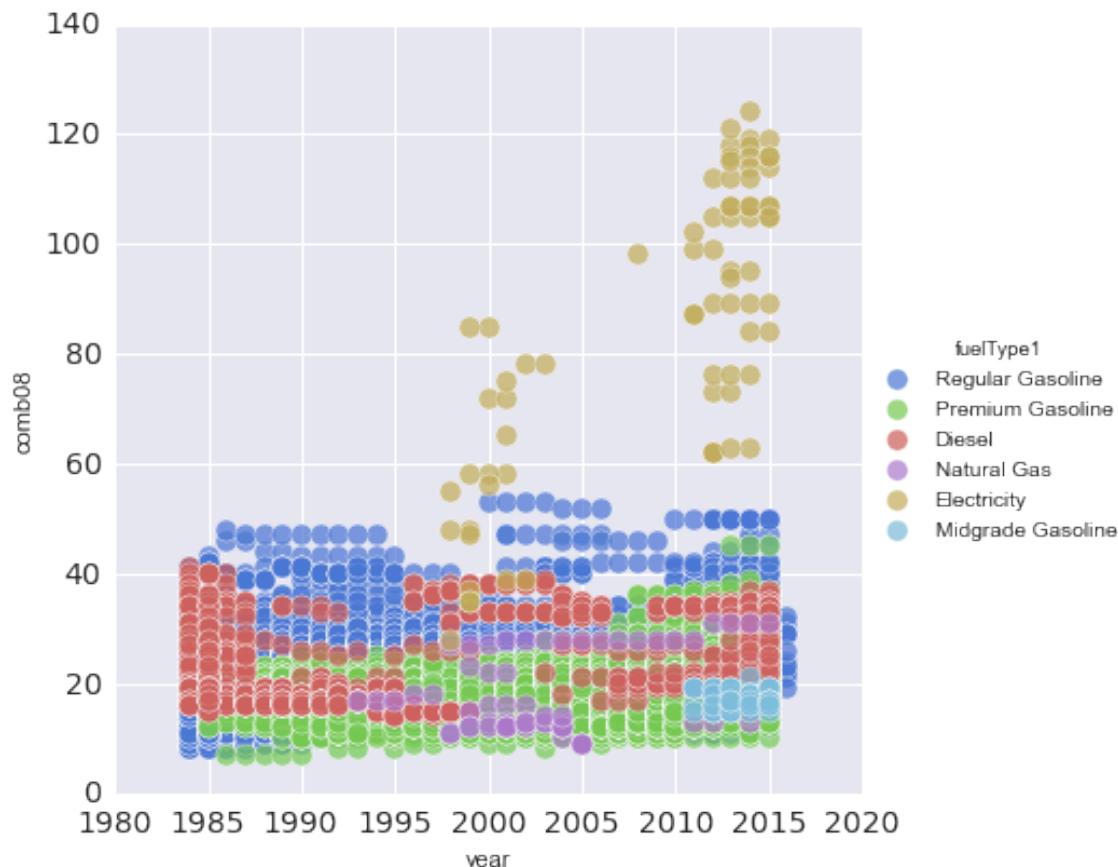
FacetGrid.map draws to all plots of the grid:

```
grid.map(plot_func, 'x_col_name', 'y_col_name')
```

# Example 1: 1-plot grid

Consumption over time, colored by primary fuel type:

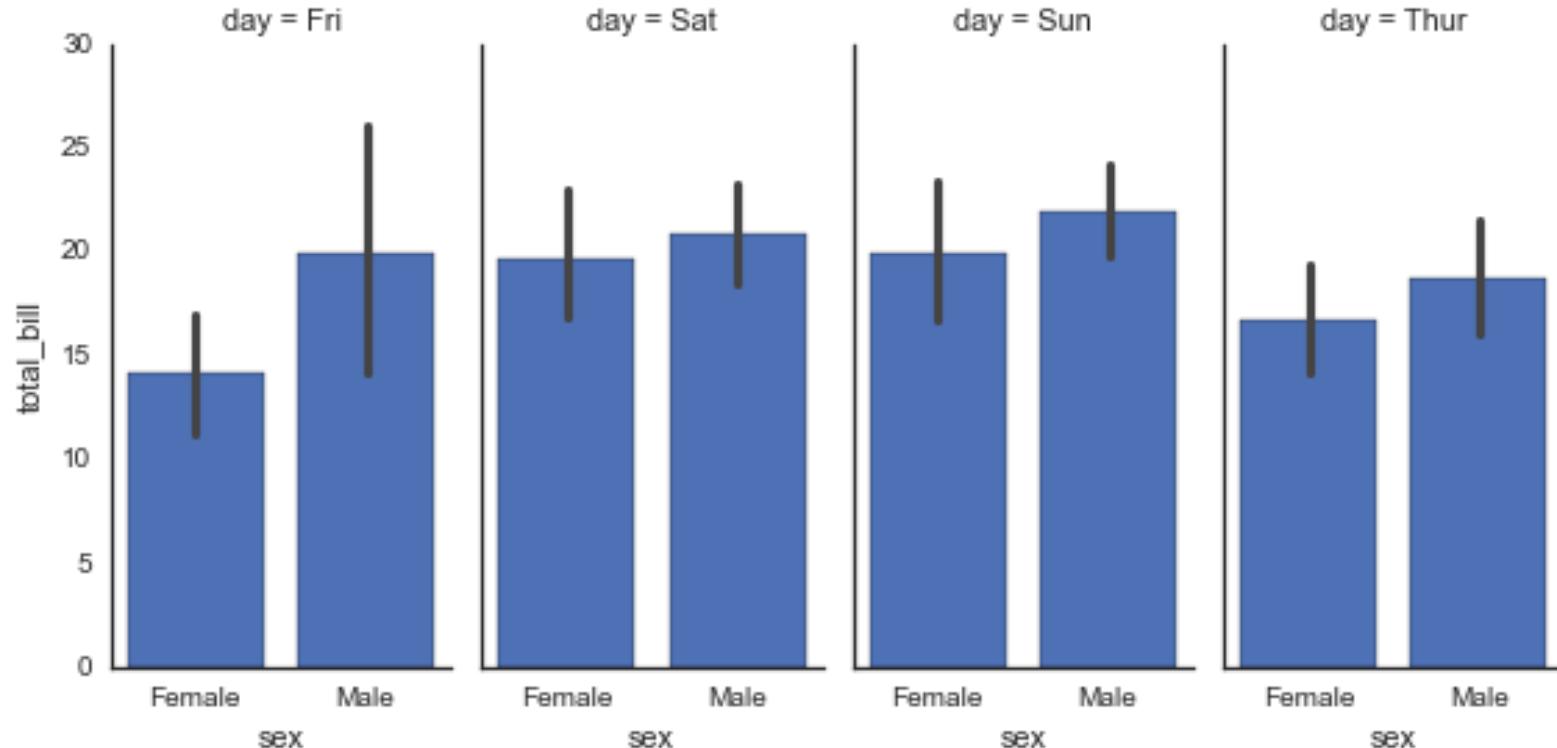
```
grid = sns.FacetGrid(vehicles, hue="fuelType1")
grid.map(plt.scatter, "year", "comb08", alpha=.7)
grid.add_legend()
```



# Example 2: 1-row grid

Total restaurant bill, grouped by day of the week and sex:

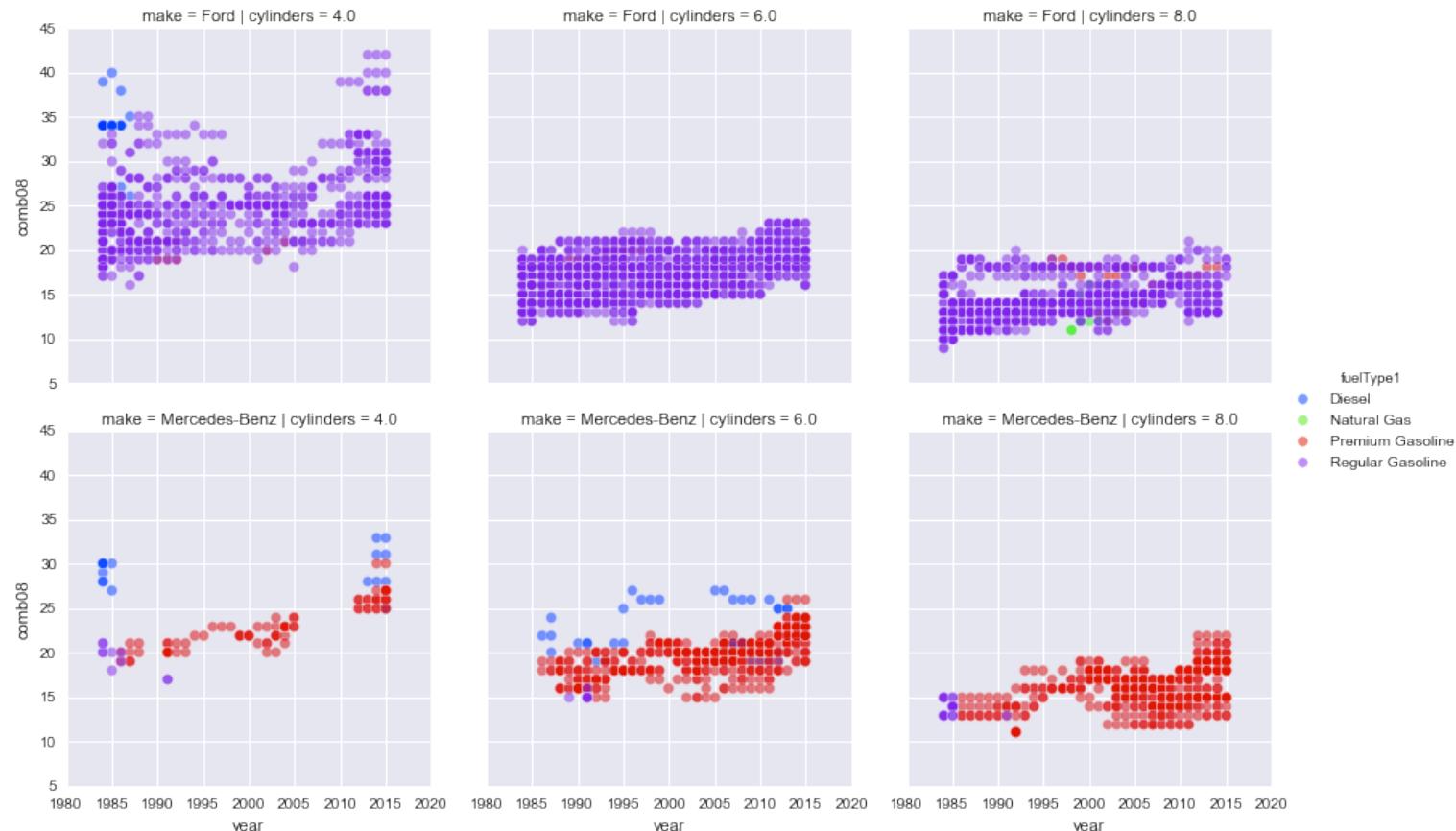
```
g = sns.FacetGrid(tips, col="day", aspect=.5)  
g.map(sns.barplot, "sex", "total_bill");
```



# Example 3: 5D plotting

Consumption over time, grouped by make and no. of cylinders:

```
grid = sns.FacetGrid(
    vehicles, row='make', col='cylinders', hue='fuelType1')
grid.map(plt.scatter, "year", "comb08", alpha=.5)
grid.add_legend()
```



# PairGrid

A PairGrid object is another plot grid, where each entry corresponds to a pair of a columns in a DataFrame. The plot displays *all* the data from the two columns.

```
grid = sns.PairGrid(data, hue='col_name', vars=None)
```

PairGrid methods allow drawing in all plots...

```
grid.map(plot_func)
```

on/off the diagonal only...

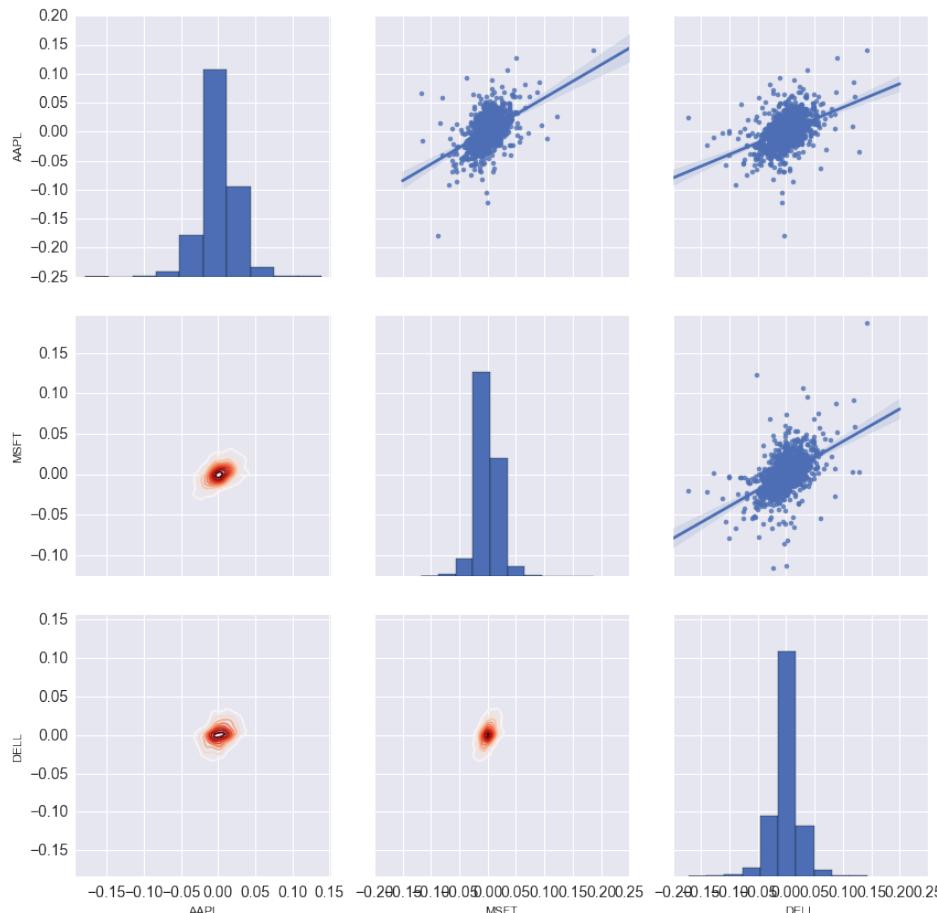
```
grid.map_diag(plot_func)  
grid.map_offdiag(plot_func)
```

or only on the upper / lower half of the grid

```
grid.map_upper(plot_func)  
grid.map_lower(plot_func)
```

# Example

```
grid = sns.PairGrid(returns, vars=['AAPL', 'MSFT', 'DELL'])  
grid.map_upper(sns.regplot)  
grid.map_diag(plt.hist)  
grid.map_lower(sns.kdeplot, linewidth=3.0, cmap='Reds')
```



# Grid-plot compatible plots

DATA SCIENCE

From Matplotlib:

`plt.hist`

`plt.scatter`

From Seaborn:

`sns.barplot`

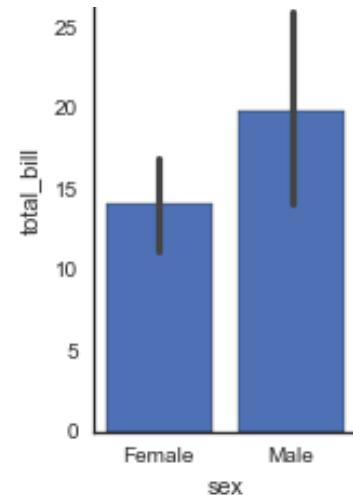
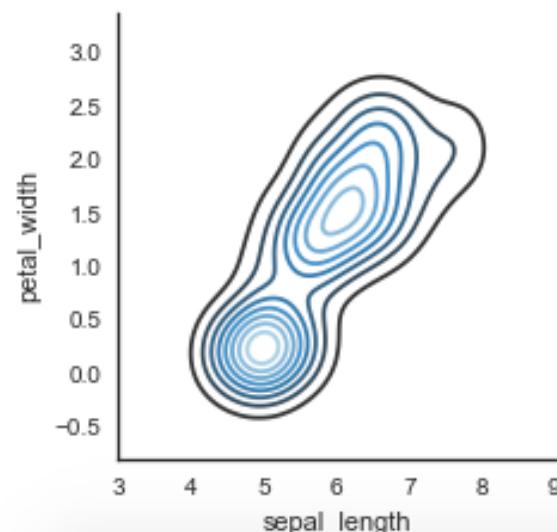
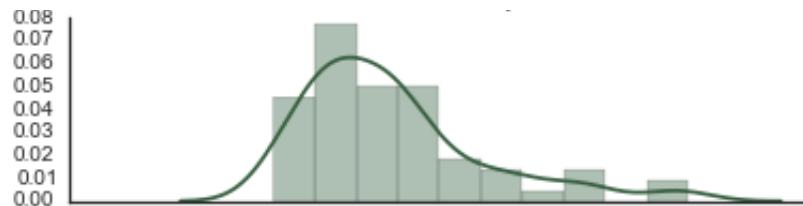
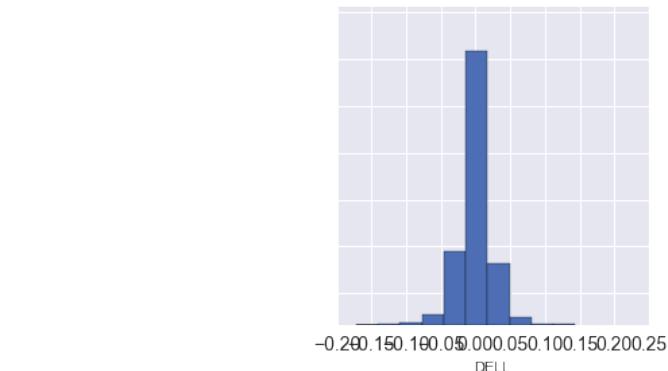
`sns.regplot`

`sns.kdeplot`

`sns.distplot`

`sns.pointplot`

...



# Data access

From file, the web, SQL, ...

# Outline

---

- Importing files
  - Low-level I/O in Python
  - Pandas I/O
  - Common file formats
- Databases
  - DB-API 2.0
  - SQL Alchemy
  - SQL queries in Pandas

# Importing files

# Reading files

## FILE INPUT EXAMPLE

```
>>> results = []
>>> f = open('c:\\rcs.txt','r')
# Read all the lines.
>>> lines = f.readlines()
>>> f.close()
# Discard the header.
>>> lines = lines[1:]

>>> for line in lines:
...     # split line into fields
...     fields = line.split()
...     # convert text to numbers
...     freq = float(fields[0])
...     vv = float(fields[1])
...     hh = float(fields[2])
...     # group & append to results
...     all = [freq,vv,hh]
...     results.append(all)
... < hit return >
```

## PRINTING THE RESULTS

```
>>> for i in results: print i
[100.0, -20.30..., -31.20...]
[200.0, -22.70..., -33.60...]
```

## EXAMPLE FILE: RCS.TXT

#	freq (MHz)	vv (dB)	hh (dB)
100	-20.3	-31.2	
200	-22.7	-33.6	



See demo/reading\_files directory  
for code.

# More compact version

## ITERATING ON A FILE AND LIST COMPREHENSIONS

```
>>> results = []
>>> f = open('c:\\rcs.txt', 'r')
>>> f.readline()
'#freq (MHz)  vv (dB)  hh (dB) \\n'
>>> for line in f:
...     all = [float(val) for val in line.split()]
...     results.append(all)
... < hit return >
>>> for i in results:
...     print i
... < hit return >
>>> f.close()
```

## EXAMPLE FILE: RCS.TXT

#freq (MHz)	vv (dB)	hh (dB)
100	-20.3	-31.2
200	-22.7	-33.6

# Writing files

## FILE OUTPUT

```
>>> # Mode 'w': create new file:  
>>> f = open('a.txt', 'w')  
>>> f.write('Hello, world!')  
>>> f.close()  
>>> open('a.txt').read()  
'Hello, world!'  
>>> # Use the 'with' statement:  
>>> with open('a.txt', 'w') as f:  
....     f.write('Wow!')  
  
....  
>>> open('a.txt').read()  
'Wow!'  
>>> # Mode 'a': append to file:  
>>> with open('a.txt', 'a') as f:  
....     f.write(' Boo.')  
  
....  
>>> open('a.txt').read()  
'Wow! Boo.'
```

## WRITE AND READ

```
>>> f = open('a.txt', 'w+')[1]  
>>> f.write('12 34 56')  
>>> f.seek(3)  
>>> f.read(2)  
'34'  
>>> f.close()
```

# csv – read and write CSV files

The `csv` module provides classes for reading and writing Comma Separated Value (CSV) files.

`csv.reader` - Read a list of comma-separated values from a file.

`csv.writer` - Output a list (or other iterable) to a file.

The format is configurable; in particular, the data delimiter does not have to be a comma.

Default “dialect” is compatible with Excel.

# csv examples

## READER

Sample data file “data.csv”:

```
"alpha 1", 100, -1.443  
"beta 3", 12, -0.0934  
"gamma 3a", 192, -0.6621  
"delta 2a", 15, -4.515
```

```
>>> import csv  
>>> rdr = csv.reader(  
                      open("data.csv"))  
>>> for row in rdr:  
...     print row  
['alpha 1', ' 100', ' -1.443']  
['beta 3', ' 12', ' -0.0934']  
['gamma 3a', ' 192', ' -0.6621']  
['delta 2a', ' 15', ' -4.515']
```

## WRITER

```
>>> import csv  
>>> data = [ ("One", 1, 1.5),  
             ("Two", 2, 8.0)]  
>>> f = open("out.csv", "w")  
>>> wrtr = csv.writer(f)  
>>> wrtr.writerows(data)  
>>> f.close()
```

Output file “out.csv”:

```
One,1,1.5  
Two,2,8.0
```

# csv options

## OPTIONS (not a complete list)

**delimiter:** str (a single character)

The field separator character

**doublequote:** bool

If True, a quote inside a string is doubled.

If False, prefix the quote with escapechar.

**escapechar:** str (a single character)

The character that indicates the following character has no special meaning.

**quotechar:** str (a single character)

Character used to quote strings containing delimiters or other special characters.

**quoting:** one of csv.QUOTE\_ constants

Controls when quotes should be generated by the writer and recognized by the reader.

## EXAMPLE

```
>>> data  
[ ('One', 1, 1.5), ('Two', 2, 8.0) ]  
>>> f = open("out.txt", "w")  
>>> wrtr = csv.writer(f,  
                      delimiter='|',  
                      quoting=csv.QUOTE_ALL)  
>>> wrtr.writerows(data)  
>>> f.close()
```

Output file “out.csv”:

```
"One"|"1"|"1.5"  
"Two"|"2"|"8.0"
```

# Pandas IO

Pandas provides a high-level interface to and from many file formats used in data science:

- .txt, CSV, json, HTML, clipboard, Excel (.xls .xlsx), pickle, HDF5, SQL, R (exp.), Stata .dta, ...

For any given format, there is

- a `read_**` function,
- a `to_**` method attached to all Pandas data objects.

You might need to install other libraries for some of the formats (Pandas will warn you if that is the case).

# Pandas' `read_table` example

## FEATURES

`read_table` can read tabular text (for example CSV files) into a DataFrame and implements the following:

- detect comments, headers and footers
- specify which column is the index
- specify the column names or which line is the column name,
- parse dates stored in one or several columns,
- handle multiple codes for missing data,
- read data by chunk (large files),
- custom value parsing based on column
- ...

## EXAMPLE

```
# Historical_data.csv
```

```
Date,AAPL,GOOGL,MSFT,PG,XOM
```

```
2005-01-03,64.78,197.4,26.8,-,51.02
```

```
2005-01-04,63.79,201.4,26.87,55.12,50.34
```

```
2005-01-05,64.46,193.45,26.84,55.28,49.83
```

```
...
```

```
>>> read_table('historical_data.csv',
                 sep=',', header=1,
                 index_col=0,
                 na_values=['-'],
                 parse_dates=True)
```

	AAPL	GOOGL	MSFT	PG	XOM
--	------	-------	------	----	-----

Date

2005-01-03	64.78	197.40	26.80	NaN	51.02
2005-01-04	63.79	201.40	26.87	55.12	50.34
2005-01-05	64.46	193.45	26.84	55.28	49.83
...					

# Reading large files in chunks

Pandas supports reading potentially very large files in chunks, e.g.:

```
>>> chunks = []
>>> reader = pd.read_csv('contributions_2012.csv',
...                      chunksize=100000)
>>> for table in reader:
...     new_yorkers = table['contbr_city'] == 'NEW YORK'
...     chunks.append(table[new_yorkers])
>>> new_york_contributions = pd.concat(chunks)
>>> print len(new_york_contributions)
25858

>>> print new_york_contributions.iloc[143]
cmte_id                  C00431171
cand_id                  P80003353
cand_nm      Romney, Mitt
contbr_st                   NY
contbr_occupation        EXECUTIVE
contb_receipt_amt            2500
contb_receipt_dt          22-JUN-11
...
```

# Pandas IO summary

## READING

Format	Method, Function, Class
txt, csv	read_table, read_csv
pickle	read_pickle
HDF5	read_hdf, HDFStore
SQL	read_sql_table
Excel	read_excel
R (exp.)	rpy.common.load_data

## WRITING

Format	Method, Function, Class
txt, csv	to_string, to_csv
html	to_html
pickle	to_pickle
HDF5	to_hdf, HDFStore
Excel	to_excel, ExcelWriter
R (exp.)	rpy.common.convert_to_r_dataframe

## EXAMPLES

```
# Excel
>>> writer = ExcelWriter('out.xlsx')
>>> df1.to_excel(writer, 'Sheet1')
>>> df2.to_excel(writer, 'Sheet2')
>>> writer.save()
>>> read_excel("out.xlsx", "Sheet2")
```

### # HDF5

```
>>> stor = HDFStore('foo.h5')
>>> stor['ser1'] = s
>>> s2 = stor['ser1']
>>> stor.close()
>>> s3 = read_hdf('foo.h5', 'ser1')
```

```
# Scrape tables from HTML webpages
read_html("http://www.bloomberg.com/markets/")
296
```

# Other common file formats

Many other file formats are supported in the scientific Python ecosystem:

File format	Package name(s)	Description
JSON	json	Human-friendly, lightweight data-interchange format.
XML	xml.etree.ElementTree, lxml	eXtensible Markup Language.
SAS	sas7bdat (to_dataframe)	SAS' own file format.
Matlab <= 7.1	scipy.io (loadmat, savemat)	Matlab's own file format.
HDF5, Matlab > 7.1	pytables, h5py	Hierarchical Data Format designed to store large amounts of numerical data.
NetCDF	netCDF4, scipy.io.netcdf	Format for array-based numerical data.
wav	scipy.io.wavfile	Audio files.
jpeg, png, ...	PIL, scipy.misc.pilutil, skimage	Common image formats.

# Databases

# Outline

---

- Learn how to access data in SQL databases
  - Low-level access with DB-API 2.0
  - SQLAlchemy, core expression language
  - SQL queries from Pandas

# DB-API 2.0

Python defines a common API for all Python DB libraries, called DB-API 2.0 and described in PEP

249. <https://www.python.org/dev/peps/pep-0249/>

Database	Module or Modules for Python 2.X
Oracle	<b>cx_Oracle</b> , DCOracl2, mxODBC, pyodbc
PostgreSQL	psycopg2, PyGreSQL, pyPgSQL, mxODBC, pyodbc, pg8000
MySQL	MySQLdb, mxODBC, pyodbc, myconnpym
Sqlite	<b>sqlite3</b> (included in standard library)
Microsoft SQL Server	adodbapi, pymssql, mssql, mxODBC, <b>pyodbc</b>
Ingres	ingresdbi
IBM DB2	lbtm_db, PyDB2, ceODBC, mxODBC, pyodbc
Sybase ASE	Sybase, mxODBC
Sybase SQL Anywhere	mxODBC, sqlanydb
SAP DB	sdb.dbapi, sapdbapi, mxODBC, sdb.sql, sapdb
Informix	InformixDB, mxODBC,
Firebird	KInterbasdb

# DB-API 2.0

## CONNECT

```
import <somedbmodule> as db
# Connect to the database.
# Extra arguments typically include user and password.
conn = db.connect(<dsn>, ...)
```

## EXECUTE STATEMENTS

```
# Get a cursor object.
c = conn.cursor()

c.execute("CREATE TABLE users(name TEXT, email TEXT, age INT, city TEXT)")
c.execute("INSERT INTO users VALUES ('Mike','mike@mike.com',23,'London')")
c.execute("INSERT INTO users VALUES ('Kim','kim@kim.net',34,'New York')")
conn.commit()

c.execute("SELECT * FROM users WHERE city='London' ORDER BY age")
for row in c: # Fetch records using the cursor as an iterator.
    print row
single = c.fetchone()          # Fetch one record.
list_of_lists = c.fetchall() # Fetch all rows in one list.
```

## CLOSE

```
c.close()
conn.close() # Closing without committing causes a rollback.
```

# DB-API 2.0: Dynamic queries

Typically, your queries will need to use variables from Python variables.

## DON'T DO THIS! INSECURE AND INEFFICIENT

```
>>> city = "London"  
# This is dangerous!  
>>> c.execute("SELECT * FROM users WHERE city='{}'".format(city))  
>>> c.fetchall()  
[(u'Mike', u'mike@mike.com', 23, u'London')]  
  
# For example, this malicious input will reveal all users.  
>>> city = "' OR '1'='1"  
>>> c.execute("SELECT * FROM users WHERE city='{}'".format(city))  
>>> c.fetchall()  
[(u'Mike', u'mike@mike.com', 23, u'London'),  
(u'Kim', u'kim@kim.net', 34, u'New York')]
```

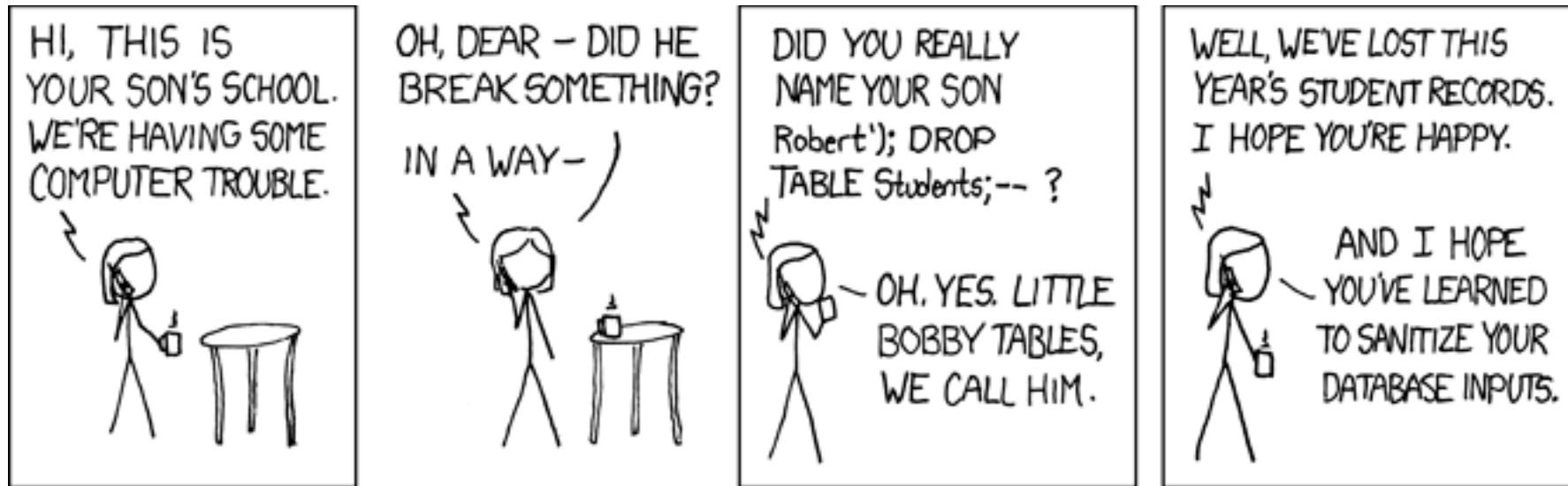
## USE PLACEHOLDERS

```
>>> city = ("London",)  
>>> c.execute("SELECT * FROM users WHERE city=?", city)  
  
>>> row = ("John", "john@corp.com", 63, "London")  
>>> c.execute("INSERT INTO users VALUES(?, ?, ?, ?)", row)  
# Some modules use a different place-holder instead of ?,  
# see db.paramstyle.
```

# DB-API 2.0: Dynamic queries

Typically, your queries will need to use variables from Python variables.

## DON'T DO THIS! INSECURE AND INEFFICIENT



## USE PLACEHOLDERS

```
>>> city = ("London",)
>>> c.execute("SELECT * FROM users WHERE city=?", city)
```

```
>>> row = ("John", "john@corp.com", 63, "London")
>>> c.execute("INSERT INTO users VALUES (?,?,?,?,?)", row)
# Some modules use a different place-holder instead of ?,  

# see db.paramstyle.
```

# DB-API 2.0: Transactions

- A transaction is sequence of operations performed as a single logical unit of work.
- Handled through the `Connection` object, using the `commit` and `rollback` connection methods.
- Use `Connection` as a context manager to group statements into a transaction, which is committed on successful exit.

## CONNECT

```
import sqlite3 as db
# Connect to the (sqlite-specific) in-memory database.
conn = db.connect(':memory:')
```

# DB-API 2.0: Transactions

## COMMIT TRANSACTION

```
# Create a simple key -> value table.
cursor = connection.cursor()

cursor.execute("CREATE TABLE example (id INT PRIMARY KEY, value VARCHAR(255))")

# Insert a single row, and explicitly commit the transaction.
cursor.execute("INSERT INTO example VALUES (1, 'hello')")
connection.commit()
cursor.execute("SELECT COUNT(*) FROM example").fetchone()
(1,)
```

## ROLLBACK TRANSACTIONS

```
# A `rollback` operation undoes all operations in a transaction.
cursor.execute("INSERT INTO example VALUES (2, 'bonjour')")
cursor.execute("INSERT INTO example VALUES (3, 'adieu')")
cursor.execute("SELECT COUNT(*) FROM example").fetchone()
(3,)

connection.rollback() # rollback the last two inserts
cursor.execute("SELECT COUNT(*) FROM example").fetchone()
(1,)

cursor.close()
connection.close()
```

# DB-API 2.0: Transactions

## CONTEXT MANAGER

```
# The context manager wraps a code block in a single transaction.
connection = sqlite3.connect("example.db")

with connection:
    cursor = connection.cursor()
    cursor.execute("CREATE TABLE example (id INT, value VARCHAR(255))")
    cursor.execute("INSERT INTO example VALUES (1, 'hello')")
    cursor.execute("INSERT INTO example VALUES (2, 'auf Wiedersehen')")

# The transaction was committed at the exit of the context manager.
cursor.execute("SELECT COUNT(*) FROM example").fetchone()
(2,)
```

## AUTOMATIC ROLLBACK

```
# The context manager automatically rolls back bad transactions.
with connection:
    cursor = connection.cursor()
    cursor.execute("INSERT INTO example VALUES (3, 'bonjour')")
    raise ValueError("dummy exception")
    cursor.execute("INSERT INTO example VALUES (4, 'adieu')")

with connection:
    cursor = connection.cursor()
    print cursor.execute("SELECT * FROM example").fetchall()
[(1, u'hello'), (2, u'auf Wiedersehen')]
```

# SQLAlchemy

- Abstracts over differences of DB backends.
- Used by famous Python users:  
Mozilla, reddit, Dropbox, Uber, ...

Two main components:

- Core Expression Language
  - Abstracts differences between databases: can emit queries for most DB backends.
  - SQL commands are built using Python statements instead of SQL strings.
  - Efficient and mostly transparent .
- Object Relational Mapper
  - Maps DB rows to Python objects.

# SQLAlchemy: concepts

- `create_engine` returns an engine that can connect to the database.
- A `Metadata` object contains information about the schema of the tables in the database.
- `Table.insert` method to insert data into an existing table
- To pass values, use a tuple or dictionary mapping column name to value (less error prone)
- Use the function `select` to create a “select” query
- Execute the query using a `Connection` object

The selection object returned by `select` provides methods to perform more complex SQL queries: `where`, `order_by`, `count`, `group_by`, ...

# SQLAlchemy: Creating tables

```
from sqlalchemy import (create_engine, Column, ForeignKey,
                        Integer, MetaData, String, Table)

# "sqlite://" is the equivalent of ":memory:" in DB-API
engine = create_engine('sqlite:///tutorial.db')

metadata = MetaData(bind=engine)

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('name', String(40)),
              Column('age', Integer),
              Column('password', String))

addresses = Table('addresses', metadata,
                   Column('id', Integer, primary_key=True),
                   Column('user_id', None, ForeignKey('users.id')),
                   Column('email_address', String, nullable=False))

# This creates the table (if it does not exist already).
metadata.create_all()
```

# SQLAlchemy: Inserting data

```
# "autoload = True" loads column name and types from the given table.  
users_again = Table("users", metadata, autoload=True)  
  
# `users.insert()` Create an Insert object on the table `users`  
# The `values` method fills in the data to be used.  
insert1 = users.insert().values(name="Joe", age=20, password="pass")  
  
# We can look at the command we have created by printing the object.  
print insert1  
INSERT INTO users (name, age, password) VALUES (?, ?, ?)  
  
# The placeholders will automatically be bound with the stored values.  
print insert1.parameters  
{'age': 20, 'password': 'pass', 'name': 'Joe'}
```

# SQLAlchemy: Inserting data

DATA SCIENCE

```
# Create three insert commands...

insert1 = users.insert().values(name="Joe", age=20, password="pass")
insert2 = users.insert().values(name="Martha", age=45, password="ding")
insert3 = users.insert().values(name="Martin", age=33, password="1234")

# ... and commit them in the same transaction to the DB.

with engine.connect() as conn:
    conn.execute(insert1)
    conn.execute(insert2)
    conn.execute(insert3)

ins = users.insert()

# Add values to the Insert object.

insert1 = ins.values(name="Joe", age=20, password="pass")
insert2 = ins.values(name="Martha", age=45, password="ding")
```

# SQLAlchemy: Inserting data

```
# Insert multiple items into the `addresses` table.

with engine.connect() as conn:

    # Here we pass the values in a list.

    conn.execute(addresses.insert(), [
        {'user_id': 1, 'email_address' : 'jack@yahoo.com'},
        {'user_id': 1, 'email_address' : 'jack@msn.com'},
        {'user_id': 2, 'email_address' : 'www@www.org'},
        {'user_id': 2, 'email_address' : 'wendy@aol.com'},
    ])
])
```

# SQLAlchemy: Selecting data

```
from sqlalchemy.sql import select
# Create a Select object to select all values from users
s = select([users])
print s
SELECT users.id, users.name, users.age, users.password
FROM users

# Execute the command.
with engine.connect() as conn:
    result = conn.execute(select([users]))
    print 'All Users (one by one):'
    for row in result:
        print row
All Users (one by one):
(1, u'Joe', 20, u'pass')
(2, u'Martha', 45, u'ding')
(3, u'Martin', 33, u'1234')
```

# SQLAlchemy: Selecting data

DATA SCIENCE

```
# Select just the names and the ages from the `users` table.  
# `users.c` is the object representing all columns for table `users`  
s = select([users.c.name, users.c.age])  
  
print s  
  
SELECT users.name, users.age  
FROM users
```

```
# Execute the command.  
  
with engine.connect() as conn:  
    result = conn.execute(select([users.c.name, users.c.age]))  
    print 'User names and ages:'  
    for row in result:  
        print row
```

User names and ages:

```
(u'Joe', 20)  
(u'Martha', 45)  
(u'Martin', 33)
```

# SQLAlchemy: Where

DATA SCIENCE

```
# Combining tables with rows that have matching IDs.

s = select([users, addresses]).where(users.c.id==addresses.c.user_id)

print s

SELECT users.id, users.name, users.age, users.password, addresses.id,
addresses.user_id, addresses.email_address

FROM users, addresses

WHERE users.id = addresses.user_id

with engine.connect() as conn:

    result = conn.execute(s)

    print 'Users and Addresses:'

    for row in result:

        print row
```

Users and Addresses:

```
(1, u'Joe', 20, u'pass', 1, 1, u'jack@yahoo.com')
(1, u'Joe', 20, u'pass', 2, 1, u'jack@msn.com')
(2, u'Martha', 45, u'ding', 3, 2, u'www@www.org')
(2, u'Martha', 45, u'ding', 4, 2, u'wendy@aol.com')
```

# SQLAlchemy: Logical operators

DATA SCIENCE

```
# Using logical operators in the selection.
```

```
from sqlalchemy.sql import and_, or_, not_
```

```
with engine.connect() as conn:
```

```
    s = select([users, addresses]).where(  
        and_(users.c.id==addresses.c.user_id, users.c.age>25))
```

```
result = conn.execute(s)
```

```
print 'Users and Addresses:'
```

```
for row in result:
```

```
    print row
```

Users and Addresses:

```
(2, u'Martha', 45, u'ding', 3, 2, u'www@www.org')
```

```
(2, u'Martha', 45, u'ding', 4, 2, u'wendy@aol.com')
```

# SQLAlchemy: Count, Update

```
# Count how many users are older than 21.  
  
s = select([users.c.id]).where(users.c.age>21).count()  
  
with engine.connect() as conn:  
    print conn.execute(s).fetchone()  
(2,)  
  
  
# Updating an existing row  
  
from sqlalchemy.sql import update  
# Create an Update object, etc...  
  
with engine.connect() as conn:  
    u = update(users).where(users.c.id==1).values(age=21)  
    conn.execute(u)  
  
with engine.connect() as conn:  
    s = select([users]).where(users.c.id==1)  
    print conn.execute(s).fetchone()  
(1, u'Joe', 21, u'pass')
```

# SQLAlchemy: Join

DATA SCIENCE

```
user_address_join = users.join(addresses)
print user_address_join
users JOIN addresses ON users.id = addresses.user_id

# Execute the command.
with engine.connect() as conn:
    s = select([users.c.name, addresses.c.email_address],
               from_obj=user_address_join)
    result = conn.execute(s)
    print 'Users and Addresses:'
    for row in result:
        print row
```

Users and Addresses:

```
(u'Joe', u'jack@yahoo.com')
(u'Joe', u'jack@msn.com')
(u'Martha', u'www@www.org')
(u'Martha', u'wendy@aol.com')
```

# SQL queries from Pandas

Pandas' `read_sql` can be used to execute SQL queries and automatically wrap result in a DataFrame:

```
# Load fuel consumption data.
table = pd.read_csv("vehicles.csv")

# Store data frame in the database in a table called
# 'vehicles'.
# ('db' is an SQLA engine or a DBAPI2 connection.)
table.to_sql("vehicles", db, if_exists="replace")
# `if_exists` is one of 'fail', 'replace', or 'append'

# Request data for vehicles using Diesel, from 1990.
# Can pass an SQL query (string) or an SQLA `Select` object
query = "SELECT year, comb08, highway08, city08 " \
        "FROM vehicles WHERE fuelType1='Diesel' AND year > "
1990
diesel_consumption = pd.read_sql(query, db)

diesel_consumption.head()
```

	year	comb08	highway08	city08
0	1993	19	22	18
1	1993	17	20	15
2	1993	17	20	15
3	1993	17	20	15
4	1993	17	20	15

# SQL and Pandas: workflow

It is in general much more efficient to execute computations on the SQL side and import the result in Pandas than vice versa.

The typical workflow is:

1. Run complex query on DB to select interesting subset of data (the full data might not fit in memory).
2. Import result in Pandas and analyze and plot.