GitHub
GIT CHEAT SHEET

V 1.1.1

Git is the open source distributed version control system that facilitates GitHub activities on your laptop or desktop. This cheat sheet summarizes commonly used Git command line instructions for quick reference.

INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Git for All Platforms
http://git-scm.com

CONFIGURE TOOLING

Configure user information for all local repositories

`$ git config --global user.name "[name]"`
Sets the name you want attached to your commit transactions
`$ git config --global user.email "[email address]"`
Sets the email you want attached to your commit transactions
`$ git config --global color.ui auto`
Enables helpful colorization of command line output

CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

`$ git init [project-name]`
Creates a new local repository with the specified name
`$ git clone [url]`
Downloads a project and its entire version history

MAKE CHANGES

Review edits and craft a commit transaction

`$ git status`
Lists all new or modified files to be committed
`$ git diff`
Shows file differences not yet staged
`$ git add [file]`
Snapshots the file in preparation for versioning
`$ git diff --staged`
Shows file differences between staging and the last file version
`$ git reset [file]`
Unstages the file, but preserve its contents
`$ git commit -m "[descriptive message]"`
Records file snapshots permanently in version history

GROUP CHANGES

Name a series of commits and combine completed efforts

`$ git branch`
Lists all local branches in the current repository
`$ git branch [branch-name]`
Creates a new branch
`$ git checkout [branch-name]`
Switches to the specified branch and updates the working directory
`$ git merge [branch]`
Combines the specified branch's history into the current branch
`$ git branch -d [branch-name]`
Deletes the specified branch

GIT CHEAT SHEET

REFACTOR FILENAMES
Relocate and remove versioned files

```
$ git rm [file]
```
Deletes the file from the working directory and stages the deletion

```
$ git rm --cached [file]
```
Removes the file from version control but preserves the file locally

```
$ git mv [file-original] [file-renamed]
```
Changes the file name and prepares it for commit

SUPPRESS TRACKING
Exclude temporary files and paths

```
*.log
build/
temp-*
```
A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

```
$ git ls-files --other --ignored --exclude-standard
```
Lists all ignored files in this project

SAVE FRAGMENTS
Shelve and restore incomplete changes

```
$ git stash
```
Temporarily stores all modified tracked files

```
$ git stash pop
```
Restores the most recently stashed files

```
$ git stash list
```
Lists all stashed changesets

```
$ git stash drop
```
Discards the most recently stashed changeset

REVIEW HISTORY
Browse and inspect the evolution of project files

```
$ git log
```
Lists version history for the current branch

```
$ git log --follow [file]
```
Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```
Shows content differences between two branches

```
$ git show [commit]
```
Outputs metadata and content changes of the specified commit

REDO COMMITS
Erase mistakes and craft replacement history

```
$ git reset [commit]
```
Undoes all commits after `[commit]`, preserving changes locally

```
$ git reset --hard [commit]
```
Discards all history and changes back to the specified commit

SYNCHRONIZE CHANGES
Register a repository bookmark and exchange version history

```
$ git fetch [bookmark]
```
Downloads all history from the repository bookmark

```
$ git merge [bookmark]/[branch]
```
Combines bookmark's branch into current local branch

```
$ git push [alias] [branch]
```
Uploads all local branch commits to GitHub

```
$ git pull
```
Downloads bookmark history and incorporates changes

Unix/Linux Command Reference

FOSSwire.com

File Commands

ls – directory listing
ls -al – formatted listing with hidden files
cd *dir* - change directory to *dir*
cd – change to home
pwd – show current directory
mkdir *dir* – create a directory *dir*
rm *file* – delete *file*
rm -r *dir* – delete directory *dir*
rm -f *file* – force remove *file*
rm -rf *dir* – force remove directory *dir* *
cp *file1 file2* – copy *file1* to *file2*
cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist
mv *file1 file2* – rename or move *file1* to *file2*
if *file2* is an existing directory, moves *file1* into directory *file2*
ln -s *file link* – create symbolic link *link* to *file*
touch *file* – create or update *file*
cat > *file* – places standard input into *file*
more *file* – output the contents of *file*
head *file* – output the first 10 lines of *file*
tail *file* – output the last 10 lines of *file*
tail -f *file* – output the contents of *file* as it grows, starting with the last 10 lines

Process Management

ps – display your currently active processes
top – display all running processes
kill *pid* – kill process id *pid*
killall *proc* – kill all processes named *proc* *
bg – lists stopped or background jobs; resume a stopped job in the background
fg – brings the most recent job to foreground
fg *n* – brings job *n* to the foreground

File Permissions

chmod *octal file* – change the permissions of *file* to *octal*, which can be found separately for user, group, and world by adding:

- 4 – read (r)
- 2 – write (w)
- 1 – execute (x)

Examples:
chmod 777 – read, write, execute for all
chmod 755 – rwx for owner, rx for group and world
For more options, see **man chmod**.

SSH

ssh *user@host* – connect to *host* as *user*
ssh -p *port user@host* – connect to *host* on port *port* as *user*
ssh-copy-id *user@host* – add your key to *host* for *user* to enable a keyed or passwordless login

Searching

grep *pattern files* – search for *pattern* in *files*
grep -r *pattern dir* – search recursively for *pattern* in *dir*
command* | grep *pattern – search for *pattern* in the output of *command*
locate *file* – find all instances of *file*

System Info

date – show the current date and time
cal – show this month's calendar
uptime – show current uptime
w – display who is online
whoami – who you are logged in as
finger *user* – display information about *user*
uname -a – show kernel information
cat /proc/cpuinfo – cpu information
cat /proc/meminfo – memory information
man *command* – show the manual for *command*
df – show disk usage
du – show directory space usage
free – show memory and swap usage
whereis *app* – show possible locations of *app*
which *app* – show which *app* will be run by default

Compression

tar cf *file.tar files* – create a tar named *file.tar* containing *files*
tar xf *file.tar* – extract the files from *file.tar*
tar czf *file.tar.gz files* – create a tar with Gzip compression
tar xzf *file.tar.gz* – extract a tar using Gzip
tar cjf *file.tar.bz2* – create a tar with Bzip2 compression
tar xjf *file.tar.bz2* – extract a tar using Bzip2
gzip *file* – compresses *file* and renames it to *file.gz*
gzip -d *file.gz* – decompresses *file.gz* back to *file*

Network

ping *host* – ping *host* and output results
whois *domain* – get whois information for *domain*
dig *domain* – get DNS information for *domain*
dig -x *host* – reverse lookup *host*
wget *file* – download *file*
wget -c *file* – continue a stopped download

Installation

Install from source:
./configure
make
make install
dpkg -i *pkg.deb* – install a package (Debian)
rpm -Uvh *pkg.rpm* – install a package (RPM)

Shortcuts

Ctrl+C – halts the current command
Ctrl+Z – stops the current command, resume with **fg** in the foreground or **bg** in the background
Ctrl+D – log out of current session, similar to **exit**
Ctrl+W – erases one word in the current line
Ctrl+U – erases the whole line
Ctrl+R – type to bring up a recent command
!! - repeats the last command
exit – log out of current session

* use with extreme caution.

Handy Reference Card #1

Core Mac OS X Keyboard Shortcuts

Top 25 Keyboard Shortcuts

Action	Press
Cut	⌘X
Copy	⌘C
Paste	⌘V
Undo	⌘Z
Select All	⌘A
Move to Trash	⌘⌫
Find	⌘F
Open	⌘O
Print	⌘P
Save	⌘S
New Window	⌘N
Close Window	⌘W
Quit Application	⌘Q
Switch Application	⌘⇥
Move to next Window in the active Application	⌘’
Force Quit	⌥⌘⎋
Get Dictionary description of a word in any of the Cocoa (Mac native) Applications	^⌘D whilst cursor is over word
Create a Text Clipping on the Desktop	Select text then click and drag to Desktop
Get Info	⌘I
Take a Picture of the Screen	⇧⌘3
Take a Picture of the Selection	⇧⌘4
Take a Picture of the Selected Window	⇧⌘4 then spacebar
Show/Hide The Dock	⌥⌘D
In Dashboard, Show/Hide Widget Dock	⌘=
Cycle to the next/previous “page” of widgets in Widget Dock	⌥ hover over a widget

⌘ command ⌫ delete ⌥ option ⎋ escape ^ control ⏏ eject ⇧ shift ⇥ tab

Handy Symbols

Symbol	Name	Shortcut
≤	Lesser than or equal to	⌥,
≥	Greater than or equal to	⌥.
÷	Division	⌥/
≠	Not equal to	⌥=
√	Square root radical	⌥V
≈	Approximately	⌥X
±	Plus/minus	⌥⇧=
°	Degree	⌥⇧8
¢	Cent sign	⌥4
©	Copyright	⌥G
®	Registered	⌥R
•	Dot	⌥8
™	Trademark	⌥2

Handy Startup Keys

Action	Press & Hold During Startup
Start up from an optical disk (CD)	C
Start up from the Diagnostic volume of the Install DVD	D
Start up from a NetBoot server	N
Start up in Target Disk Mode (on computers that offer this feature)	T
Start up in Verbose mode	⌘V
Start up in Single-user mode	⌘S
Reset the Parameter RAM (PRAM) (hold down until second chime)	⌥⌘PR
Boot into Open Firmware	⌥⌘OF
Start up in Safe Boot mode and temporarily disable login items and non-essential kernel extension files (Mac OS X 10.2 and later)	⇧
Access Startup Manager (to select an operating system)	⌥
Eject removable media	Click and hold mouse

⌘ command ⌫ delete ⌥ option ⎋ escape ^ control ⏏ eject ⇧ shift ⇥ tab

Python Notes/Cheat Sheet

Comments
from the hash symbol to the end of a line

Code blocks
Delineated by colons and indented code; and not the curly brackets of C, C++ and Java.

```
def is_fish_as_string(argument):
    if argument:
        return 'fish'
    else:
        return 'not fish'
```

Note: Four spaces per indentation level is the Python standard. Never use tabs: mixing tabs and spaces produces hard-to-find errors. Set your editor to convert tabs to spaces.

Line breaks
Typically, a statement must be on one line. Bracketed code - (), [] or {} - can be split across lines; or (if you must) use a backslash \ at the end of a line to continue a statement on to the next line (but this can result in hard to debug code).

Naming conventions

Style	Use
StudlyCase	Class names
joined_lower	Identifiers, functions; and class methods, attributes
_joined_lower	Internal class attributes
__joined_lower	Private class attributes # this use not recommended
joined_lower **ALL_CAPS**	Constants

Basic object types (not a complete list)

Type	Examples	
None	None	# singleton null object
Boolean	True, False	
integer	-1, 0, 1, sys.maxint	
long	1L, 9787L	# arbitrary length ints
float	3.14159265 inf, float('inf') -inf nan, float('nan')	 # infinity # neg infinity # not a number
complex	2+3j	# note use of j
string	'I am a string', "me too" '"multi-line string"', """+1""" r'raw string', b'ASCII string' u'unicode string'	
tuple	empty = () (1, True, 'dog')	# empty tuple # immutable list
list	empty = [] [1, True, 'dog']	# empty list # mutable list
set	empty = set() set(1, True, 'a')	# the empty set # mutable
dictionary	empty = {} {'a': 'dog', 7: 'seven', True: 1}	# mutable object
file	f = open('filename', 'rb')	

Note: Python has four numeric types (integer, float, long and complex) and several sequence types including strings, lists, tuples, bytearrays, buffers, and xrange objects.

Operators

Operator	Functionality
+	Addition (also string, tuple, list concatenation)
-	Subtraction (also set difference)
*	Multiplication (also string, tuple, list replication)
/	Division
%	Modulus (also a string format function, but use deprecated)
//	Integer division rounded towards minus infinity
**	Exponentiation
=, -=, +=, /=, *=, %=, //=, **=	Assignment operators
==, !=, <, <=, >=, >	Boolean comparisons
and, or, not	Boolean operators
in, not in	Membership test operators
is, is not	Object identity operators
|, ^, &, ~	Bitwise: or, xor, and, compliment
<<, >>	Left and right bit shift
;	Inline statement separator # inline statements discouraged

Hint: float('inf') always tests as larger than any number, including integers.

Modules
Modules open up a world of Python extensions that can be imported and used. Access to the functions, variables and classes of a module depend on how the module was imported.

Import method	Access/Use syntax
import math	math.cos(math.pi/3)
import math as m **# import using an alias**	m.cos(m.pi/3)
from math import cos,pi **# only import specifics**	cos(pi/3)
from math import * **# BADish global import**	log(e)

Global imports make for unreadable code!!!

Oft used modules

Module	Purpose
datetime **time**	Date and time functions
math	Core math functions and the constants pi and e
pickle	Serialise objects to a file
os **os.path**	Operating system interfaces
re	A library of Perl-like regular expression operations
string	Useful constants and classes
sys	System parameters and functions
numpy	Numerical python library
pandas	R DataFrames for Python
matplotlib	Plotting/charting for Python

If - flow control

```
if condition:   # for example: if x < 5:
    statements
elif condition: # optional – can be multiple
    statements
else:           # optional
    statements
```

For - flow control

```
for x in iterable:
    statements
else:           # optional completion code
    statements
```

While - flow control

```
while condition:
    statements
else:           # optional completion code
    statements
```

Ternary statement

id = *expression* if *condition* else *expression*

```
x = y if a > b else z - 5
```

Some useful adjuncts:

- pass - a statement that does nothing
- continue - moves to the next loop iteration
- break - to exit for and while loop

Trap: break skips the else completion code

Exceptions – flow control

```
try:
    statements
except (tuple_of_errors): # can be multiple
    statements
else:           # optional no exceptions
    statements
finally:        # optional all
    statements
```

Common exceptions (not a complete list)

Exception	Why it happens
AsserionError	Assert statement failed
AttributeError	Class attribute assignment or reference failed
IOError	Failed I/O operation
ImportError	Failed module import
IndexError	Subscript out of range
KeyError	Dictionary key not found
MemoryError	Ran out of memory
NameError	Name not found
TypeError	Value of the wrong type
ValueError	Right type but wrong value

Raising errors

Errors are raised using the raise statement

```
raise ValueError(value)
```

Creating new errors

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

Objects and variables (AKA identifiers)

- Everything is an object in Python (in the sense that it can be assigned to a variable or passed as an argument to a function)
- Most Python objects have methods and attributes. For example, all functions have the built-in attribute __doc__, which returns the doc string defined in the function's source code.
- All variables are effectively "pointers", not "locations". They are references to objects; and often called identifiers.
- Objects are strongly typed, not identifiers
- Some objects are immutable (int, float, string, tuple, frozenset). But most are mutable (including: list, set, dictionary, NumPy arrays, etc.)
- You can create our own object types by defining a new class (see below).

Booleans and truthiness

Most Python objects have a notion of "truth".

False	True
None	
0 **int(False) # → 0**	Any number other than 0 int(True) # → 1
"" **# the empty string**	" ", 'fred', 'False' # all other strings
() [] {} set() **# empty containers**	[None], (False), {1, 1} # non-empty containers, including those containing False or None.

You can use bool() to discover the truth status of an object.

```
a = bool(obj)           # the truth of obj
```

It is pythonic to use the truth of objects.

```
if container:          # test not empty
    # do something
while items:           # common looping idiom
    item = items.pop()
    # process item
```

Specify the truth of the classes you write using the __nonzero__() magic method.

Comparisons

Python lets you compare ranges, for example

```
if 1 < x < 100: # do something ...
```

Tuples

Tuples are immutable lists. They can be searched, indexed and iterated much like lists (see below). List methods that do not change the list also work on tuples.

```
a = ()                      # the empty tuple
a = (1,)  # ← note comma # one item tuple
a = (1, 2, 3)               # multi-item tuple
a = ((1, 2), (3, 4))        # nested tuple
a = tuple(['a', 'b'])       # conversion
```

Note: the comma is the tuple constructor, not the parentheses. The parentheses add clarity.

The Python swap variable idiom

```
a, b = b, a    # no need for a temp variable
```

This syntax uses tuples to achieve its magic.

String (immutable, ordered, characters)

```
s = 'string'.upper()          # STRING
s = 'fred'+'was'+'here'       # concatenation
s = ''.join(['fred', 'was', 'here']) # ditto
s = 'spam' * 3                # replication
s = str(x)                    # conversion
```

String iteration and sub-string searching

```
for character in 'str':       # iteration
    print (ord(character))    # 115 116 114
for index, character in enumerate('str')
    print (index, character)
if 'red' in 'Fred':           # searching
    print ('Fred is red')     # it prints!
```

String methods (not a complete list)

capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill

String constants (not a complete list)

```
from string import *          # I'm bad ...
print ((digits, hexdigits, letters,
        lowercase, uppercase, punctuation))
```

Old school string formatting (using % oper)

```
print ("It %s %d times" % ['occurred', 5])
# prints: 'It occurred 5 times'
```

Code	Meaning
s	String or string conversion
c	Character
d	Signed decimal integer
u	Unsigned decimal integer
H or h	Hex integer (upper or lower case)
f	Floating point
E or e	Exponent (upper or lower case E)
G or g	The shorter of e and f (u/l case)
%	Literal '%'

```
'%s' % math.pi     # --> '3.14159265359'
'%f' % math.pi     # --> '3.141593'
'%.2f' % math.pi   # --> '3.14'
'%.2e' % 3000      # --> '3.00e+03'
'%03d' % 5         # --> '005'
```

New string formatting (using format method)

Uses: 'template-string'.format(arguments)

Examples (using similar codes as above):

```
'Hello {}'.format('World')# 'Hello World'
'{}'.format(math.pi)      # ' 3.14159265359'
'{0:.2f}'.format(math.pi) # '3.14'
'{0:+.2f}'.format(5)      # '+5.00'
'{:.2e}'.format(3000)     # '3.00e+03'
'{:0>2d}'.format(5)       # '05' (left pad)
'{:x<3d}'.format(5)       # '5xx' (rt. pad)
'{:,}'.format(1000000)    # '1,000,000'
'{:.1%}'.format(0.25)     # '25.0%'
'{0}{1}'.format('a', 'b') # 'ab'
'{1}{0}'.format('a', 'b') # 'ba'
'{num:}'.format(num=7)    # '7' (named args)
```

List (mutable, indexed, ordered container)

Indexed from zero to length-1

```
a = []                        # the empty list
a = ['dog', 'cat', 'bird']    # simple list
a = [[1, 2], ['a', 'b']]      # nested lists
a = [1, 2, 3] + [4, 5, 6]     # concatenation
a = [1, 2, 3] * 456           # replication
a = list(x)                   # conversion
```

List comprehensions (can be nested)

Comprehensions: a tight way of creating lists

```
t3 = [x*3 for x in [5, 6, 7]] # [15, 18, 21]
z = [complex(x, y) for x in range(0, 4, 1)
          for y in range(4, 0, -1) if x > y]
# z --> [(2+1j), (3+2j), (3+1j)]
```

Iterating lists

```
L = ['dog', 'cat', 'turtle']
for item in L
    print (item)
for index, item in enumerate(L):
    print (index, item)
```

Searching lists

```
L = ['dog', 'cat', 'turtle']; value = 'cat'
if value in L:
    count = L.count(value)
    first_occurrence = L.index(value)
if value not in L:
    print 'list is missing {}'.format(value)
```

List methods (not a complete list)

Method	What it does
l.append(x)	Add x to end of list
l.extend(other)	Append items from other
l.insert(pos, x)	Insert x at position
del l[pos]	Delete item at pos
l.remove(x)	Remove first occurrence of x; An error if no x
l.pop([pos])	Remove last item from list (or item from pos); An error if empty list
l.index(x)	Get index of first occurrence of x; An error if x not found
l.count(x)	Count the number of times x is found in the list
l.sort()	In place list sort
l.reverse(x)	In place list reversal

List slicing

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8] # play data
x[2]     # 3rd element - reference not slice
x[1:3]   # 2nd to 3rd element (1, 2)
x[:3]    # The first three elements (0,1,2)
x[-3:]   # last three elements
x[:-3]   # all but the last three elements
x[:]     # every element of x – copies x
x[1:-1]  # all but first and last element
x[::3]  # (0, 3, 6, 9, …) 1st then every 3rd
x[1:5:2]# (1,3) start 1, stop >= 5, every2nd
```

Note: All Python sequence types support the above index slicing (strings, lists, tuples, bytearrays, buffers, and xrange objects)

Set (unique, unordered container)

A Python set is an unordered, mutable collection of unique hashable objects.

```
a = set()                       # empty set
a = {'red', 'white', 'blue'} # simple set
a = set(x)                      # convert list
```

Trap: {} creates empty dict, not an empty set

Set comprehensions

```
# a set of selected letters...
s = {e for e in 'ABCHJADC' if e not in 'AB'}
# --> {'H', 'C', 'J', 'D'}
# a set of tuples ...
s = {(x,y) for x in range(-1,2)
               for y in range (-1,2)}
```

Trap: set contents need to be immutable to be hashable. So you can have a set of tuples, but not a set of lists.

Iterating a set

```
for item in set:
    print (item)
```

Searching a set

```
if item in set:
    print (item)
if item not in set:
    print ('{} is missing'.format(item))
```

Set methods (not a complete list)

Method	What it does
len(s)	Number of items in set
s.add(item)	Add item to set
s.remove(item)	Remove item from set. Raise KeyError if item not found.
s.discard(item)	Remove item from set if present.
s.pop()	Remove and return an arbitrary item. Raise KeyError on empty set.
s.clear()	Remove all items from set
item in s	True or False
item not in s	True or False
iter(s)	An iterator over the items in the set (arbitrary order)
s.copy()	Get shallow copy of set
s.isdisjoint(o)	True if s has not items in common with other set o
s.issubset(o)	Same as set <= other
s.issuperset(o)	Same as set >= other
s.union(o[, ...])	Return new union set
s.intersection(o)	Return new intersection
s.difference(o)	Get net set of items in s but not others (Same as set – other)

Frozenset

Similar to a Python set above, but immutable (and therefore hashable).

```
f = frozenset(s)                # convert set
f = frozenset(o)                # convert other
```

Dictionary (indexed, unordered map-container)

A mutable hash map of unique key=value pairs.

```
a = {}                          # empty dictionary
a = {1: 1, 2: 4, 3: 9} # simple dict
a = dict(x)                     # convert paired data
# next example – create from a list
l = ['alpha', 'beta', 'gamma', 'delta']
a = dict(zip(range(len(l)), l))
# Example using string & generator
expression
s = 'a=apple,b=bird,c=cat,d=dog,e=egg'
a = dict(i.split("=") for i in s.split(","))
# {'a': 'apple', 'c': 'cat', 'b': 'bird',
#  'e': 'egg', 'd': 'dog'}
```

Dictionary comprehensions

Conceptually like list comprehensions; but it constructs a dictionary rather than a list

```
a = { n: n*n for n in range(7) }
# a -> {0:0, 1:1, 2:4, 3:9, 4:16, 5:25,6:36}
odd_sq = { n: n*n for n in range(7) if n%2 }
# odd_sq -> {1: 1, 3: 9, 5: 25}
# next example -> swaps the key:value pairs
a = { val: key for key, val in a.items() }
# next example -> count list occurrences
l = [1,2,9,2,7,3,7,1,22,1,7,7,22,22,9,0,9,0]
c = { key: l.count(key) for key in set(l) }
# c -> {0:2, 1:3, 2:2, 3:1, 7:4, 9:3, 22:3}
```

Iterating a dictionary

```
for key in dictionary:
    print (key)
for key, value in dictionary.items():
    print (key, value)
```

Searching a dictionary

```
if key in dictionary:
    print (key)
```

Merging two dictionaries

```
merged = dict_1.copy()
merged.update(dict_2)
```

Dictionary methods (not a complete list)

Method	What it does
len(d)	Number of items in d
d[key]	Get value for key or raise the KeyError exception
d[key] = value	Set key to value
del d[key]	deletion
key in d	True or False
key not in d	True or False
iter(d)	An iterator over the keys
d.clear()	Remove all items from d
d.copy()	Shallow copy of dictionary
d.get(key[, def])	Get value else default
d.items()	Dictionary's (k,v) pairs
d.keys()	Dictionary's keys
d.pop(key[, def])	Get value else default; remove key from dictionary
d.popitem()	Remove and return an arbitrary (k, v) pair
d.setdefault(k[,def])	If k in dict return its value otherwise set def
d.update(other_d)	Update d with key:val pairs from other
d.values()	The values from dict

Key functions (not a complete list)

Function	What it does
abs(num)	Absolute value of num
all(iterable)	True if all are True
any(iterable)	True if any are True
bytearray(source)	A mutable array of bytes
callable(obj)	True if obj is callable
chr(int)	Character for ASCII int
complex(re[, im])	Create a complex number
divmod(a, b)	Get (quotient, remainder)
enumerate(seq)	Get an enumerate object, with next() method returns an (index, element) tuple
eval(string)	Evaluate an expression
filter(fn, iter)	Construct a list of elements from *iter* for which *fn()* returns True
float(x)	Convert from int/string
getattr(obj, str)	Like obj.str
hasattr(obj, str)	True if obj has attribute
hex(x)	From in to hex string
id(obj)	Return unique (run-time) identifier for an object
int(x)	Convert from float/string
isinstance(o, c)	Eg. isinstance(2.1, float)
len(x)	Number of items in x; x is string, tuple, list, dict
list(iterable)	Make a list
long(x)	Convert a string or number to a long integer
map(fn, iterable)	Apply fn() to every item in iterable; return results in a list
max(a,b) **max(iterable)**	What it says on the tin
min(a,b) **min(iterable)**	Ditto
next(iterator)	Get next item from an iter
open(name[,mode])	Open a file object
ord(c)	Opposite of chr(int)
pow(x, y)	Same as x ** y
print (objects)	What it says on the tin takes end arg (default \n) and sep arg (default ' ')
range(stop) **range(start,stop)** **range(fr,to,step)**	integer list; stops < stop default start=0; default step=1
reduce(fn, iter)	Applies the two argument fn(x, y) cumulatively to the items of iter.
repr(object)	Printable representation of an object
reversed(seq)	Get a reversed iterator
round(n[,digits])	Round to number of digits after the decimal place
setattr(obj,n,v)	Like obj.n = v #name/value
sorted(iterable)	Get new sorted list
str(object)	Get a string for an object
sum(iterable)	Sum list of numbers
type(object)	Get the type of object
xrange()	Like range() but better: returns an iterator
zip(x, y[, z])	Return a list of tuples

Using functions

When called, functions can take positional and named arguments.

For example:

```
result = function(32, aVar, c='see', d={})
```

Arguments are passed by reference (ie. the objects are not copied, just the references).

Writing a simple function

```
def funct(arg1, arg2=None, *args, **kwargs):
    """explain what this function does"""
    statements
    return x     # optional statement
```

Note: functions are first class objects that get instantiated with attributes and they can be referenced by variables.

Avoid named default mutable arguments

Avoid mutable objects as default arguments. Expressions in default arguments are evaluated when the function is defined, not when it's called. Changes to mutable default arguments survive between function calls.

```
def nasty(value=[]):       # <-- mutable arg
    value.append('a')
    return value
print (nasty ()) # --> ['a']
print (nasty ()) # --> ['a', 'a']

def better(val=None):
    val = [] if val is None else val
    value.append('a')
    return value
```

Lambda (inline expression) functions:

```
g = lambda x: x ** 2      # Note: no return
print(g(8))               # prints 64
mul = lambda a, b: a * b  # two arguments
mul(4, 5) == 4 * 5        # --> True
```

Note: only for expressions, not statements.
Lambdas are often used with the Python functions filter(), map() and reduce().

```
# get only those numbers divisible by three
div3 = filter(lambda x: x%3==0,range(1,101))
```

Typically, you can put a lambda function anywhere you put a normal function call.

Closures

Closures are functions that have inner functions with data fixed in the inner function by the lexical scope of the outer. They are useful for avoiding hard constants. Wikipedia has a derivative function for changeable values of dx, using a closure.

```
def derivative(f, dx):
    """Return a function that approximates
       the derivative of f using an interval
       of dx, which should be appropriately
       small.
    """
    def _function(x):
        return (f(x + dx) - f(x)) / dx
    return _function #from derivative(f, dx)

f_dash_x = derivative(lambda x: x*x,0.00001)
f_dash_x(5) # yields approx. 10 (ie. y'=2x)
```

An iterable object
The contents of an iterable object can be selected one at a time. Such objects include the Python sequence types and classes with the magic method __iter__(), which returns an iterator. An iterable object will produce a fresh iterator with each call to iter().

```
iterator = iter(iterable_object)
```

Iterators
Objects with a next() (Python 2) or __next__() (Python 3) method, that:

- returns the next value in the iteration
- updates the internal note of the next value
- raises a StopIteration exception when done

Note: with the loop `for x in y:` if y is not an iterator; Python calls iter() to get one. With each loop, it calls next() on the iterator until a StopIteration exception.

```
x = iter('XY')  # iterate a string by hand
print (next(x)) # --> X
print (next(x)) # --> Y
print (next(x)) # --> StopIteration
```

Generators
Generator functions are resumable functions that work like iterators. They can be more space or time efficient than iterating over a list, (especially a very large list), as they only produce items as they are needed.

```
def fib(max=None):
    """ generator for Fibonacci sequence"""
    a, b = 0, 1
    while max is None or b <= max:
        yield b    # ← yield is like return
        a, b = b, a+b

[i for i in fib(10)] # → [1, 1, 2, 3, 5, 8]
```

Note: a return statement (or getting to the end of the function) ends the iteration.
Trap: a yield statement is not allowed in the try clause of a try/finally construct.

Messaging the generator

```
def resetableCounter(max=None):
    j = 0
    while max is None or j <= max:
        x = yield j # ← x gets the sent arg
        j = j+1 if x is None else x

x = resetableCounter(10)
print x.send(None)  # → 0
print x.send(5)     # → 5
print x.send(None)  # → 6
print x.send(11)    # → StopIteration
```

Trap: must send None on first send() call

Generator expressions
Generator expressions build generators, just like building a list from a comprehension. You can turn a list comprehension into a generator expression simply by replacing the square brackets [] with parentheses ().

```
[i for i in xrange(10)] # list comprehension
list(i for i in xrange(10)) # generated list
```

Classes
Python is an object-oriented language with a multiple inheritance class mechanism that encapsulates program code and data.

Methods and attributes
Most objects have associated functions or "methods" that are called using dot syntax:

```
obj.method(arg)
```

Objects also often have attributes or values that are directly accessed without using getters and setters (most unlike Java or C++)

```
instance = Example_Class()
print (instance.attribute)
```

Simple example

```
import math
class Point:
    # static class variable, point count
    count = 0

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)
        Point.count += 1

    def __str__(self):
        return \
        '(x={}, y={})'.format(self.x,
self.y)

    def to_polar(self):
        r = math.sqrt(self.x**2 + self.y**2)
        theta = math.atan2(self.y, self.x)
        return(r, theta)

    # static method – trivial example ...
    def static_eg(n):
        print ('{}'.format(n))
    static_eg = staticmethod(static_eg)

# Instantiate 9 points & get polar coords
for x in range(-1, 2):
    for y in range(-1, 2):
        p = Point(x, y)
        print (p) # uses __str__() method
        print (p.to_polar())
print (Point.count) # check static variable
Point.static_eg(9)  # check static method
```

The self
Class methods have an extra argument over functions. Usually named 'self'; it is a reference to the instance. It is not used in the method call; and is provided by Python to the method. Self is like 'this' in C++ & Java

Public and private methods and variables
Python does not enforce the public v private data distinction. By convention, variables and methods that begin with an underscore should be treated as private (unless you really know what you are doing). Variables that begin with double underscore are mangled by the compiler (and hence more private).

Inheritance

```
class DerivedClass1(BaseClass):
    statements
class DerivedClass2(module_name.BaseClass):
    statements
```

Multiple inheritance

```
class DerivedClass(Base1, Base2, Base3):
    statements
```

Decorators

Technically, decorators are just functions (or classes), that take a callable object as an argument, and return an analogous object with the decoration. We will skip how to write them, and focus on using a couple of common built in decorators.

Practically, decorators are syntactic sugar for more readable code. The *@wrapper* is used to transform the existing code. For example, the following two method definitions are semantically equivalent.

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Getters and setters

Although class attributes can be directly accessed, the property function creates a property manager.

```
class Example:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx,"Doc txt")
```

Which can be rewritten with decorators as:

```
class Example:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """Doc txt: I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Magic class methods (not a complete list)

Magic methods (which begin and end with double underscore) add functionality to your classes consistent with the broader language.

Magic method	What it does
__init__(self,[...])	Constructor
__del__(self)	Destructor pre-garbage collection
__str__(self)	Human readable string for class contents. Called by str(self)
__repr__(self)	Machine readable unambiguous Python string expression for class contents. Called by repr(self) Note: str(self) will call __repr__ if __str__ is not defined.
__eq__(self, other)	Behaviour for ==
__ne__(self, other)	Behaviour for !=
__lt__(self, other)	Behaviour for <
__gt__(self, other)	Behaviour for >
__le__(self, other)	Behaviour for <=
__ge__(self, other)	Behaviour for >=
__add__(self, other)	Behaviour for +
__sub__(self, other)	Behaviour for -
__mul__(self, other)	Behaviour for *
__div__(self, other)	Behaviour for /
__mod__(self, other)	Behaviour for %
__pow__(self, other)	Behaviour for **
__pos__(self, other)	Behaviour for unary +
__neg__(self, other)	Behaviour for unary -
__hash__(self)	Returns an int when hash() called. Allows class instance to be put in a dictionary
__len__(self)	Length of container
__contains__(self, i)	Behaviour for in and not in operators
__missing__(self, i)	What to do when dict key i is missing
__copy__(self)	Shallow copy constructor
__deepcopy__(self, memodict={})	Deep copy constructor
__iter__(self)	Provide an iterator
__nonzero__(self)	Called by bool(self)
__index__(self)	Called by x[self]
__setattr__(self, name, val)	Called by self.name = val
__getattribute__(self, name)	Called by self.name
__getattr__(self, name)	Called when self.name does not exist
__delattr__(self, name)	Called by del self.name
__getitem__(self, key)	Called by self[key]
__setitem__(self, key, val)	Called by self[key] = val
__delitem__(self, key)	del self[key]

IPython Notebook Shortcuts

Command Mode (press `Esc` to enable)

Enter: enter edit mode
Shift-Enter: run cell, select below
Ctrl-Enter: run cell
Alt-Enter: run cell, insert below
Y: to code
M: to markdown
R: to raw
1: to heading 1
2: to heading 2
3: to heading 3
4: to heading 4
5: to heading 5
6: to heading 6
Up: select cell above
K: select cell above
Down: select cell below
J: select cell below
A: insert cell above
B: insert cell below
X: cut selected cell
C: copy selected cell
Shift-V: paste cell above
V: paste cell below
Z: undo last cell deletion
D,D: delete selected cell
Shift-M: merge cell below
S: Save and Checkpoint
Ctrl-S: Save and Checkpoint
L: toggle line numbers
O: toggle output
Shift-O: toggle output scrolling
Esc: close pager
Q: close pager
H: show keyboard shortcut help dialog
I,I: interrupt kernel
0,0: restart kernel
Space: scroll down
Shift-Space: scroll up
Shift: ignore

Edit Mode (press `Enter` to enable)

Tab: code completion or indent
Shift-Tab: tooltip
Ctrl-]: indent
Ctrl-[: dedent
Ctrl-A: select all
Ctrl-Z: undo
Ctrl-Shift-Z: redo
Ctrl-Y: redo
Ctrl-Home: go to cell start
Ctrl-Up: go to cell start
Ctrl-End: go to cell end
Ctrl-Down: go to cell end
Ctrl-Left: go one word left
Ctrl-Right: go one word right
Ctrl-Backspace: delete word before
Ctrl-Delete: delete word after
Esc: command mode
Ctrl-M: command mode
Shift-Enter: run cell, select below
Ctrl-Enter: run cell
Alt-Enter: run cell, insert below
Ctrl-Shift-Subtract: split cell
Ctrl-Shift--: split cell
Ctrl-S: Save and Checkpoint
Up: move cursor up or previous cell
Down: move cursor down or next cell
Shift: ignore

Introductory Notes: Matplotlib

Preliminaries

Start by importing these Python modules

```
import numpy as np
import pandas as pd
from pandas import DataFrame, Series
import matplotlib.pyplot as plt
import matplotlib
```

Which Application Programming Interface?

The two worlds of Matplotlib

There are 2 broad ways of using pyplot:

1. The first (and most common) way is not pythonic. It relies on global functions to build and display a global figure using matplotlib as a global state machine. (This is an easy approach for interactive use).
2. The second way is pythonic and object oriented. You obtain an empty Figure from a global factory, and then build the plot explicitly using the methods of the Figure and the classes it contains. (This is the best approach for programmatic use).

While these notes focus on second approach, let's begin with a quick look at the first.

Using matplotlib in a non-pythonic way

1. Get some (fake) data - monthly time series

```
x = pd.period_range('1980-01-01',
    periods=410, freq='M')
x = x.to_timestamp().to_pydatetime()
y = np.random.randn(len(x)).cumsum()
```

2. Plot the data

```
plt.plot(x, y, label='FDI')
```

3. Add your labels and pretty-up the plot

```
plt.title('Fake Data Index')
plt.xlabel('Date')
plt.ylabel('Index')
plt.grid(True)
plt.figtext(0.995, 0.01, 'Footnote',
    ha='right', va='bottom')
plt.legend(loc='best', framealpha=0.5,
    prop={'size':'small'})
plt.tight_layout(pad=1)
plt.gcf().set_size_inches(8, 4)
```

4. SAVE the figure

```
plt.savefig('filename.png')
```

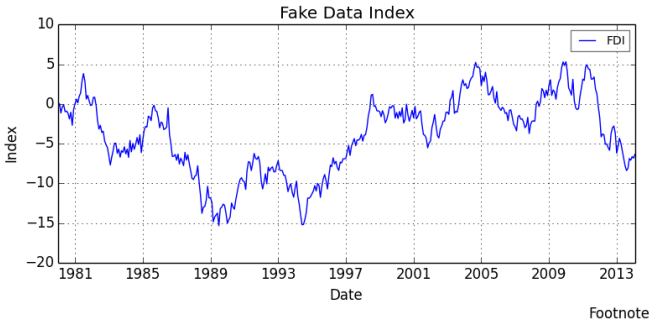
5. Finally, close the figure

```
plt.close()
```

Alternatively, SHOW the figure

With IPython, follow steps 1 to 3 above then

```
plt.show() # Note: also closes the figure
```



Matplotlib: intro to the object oriented way

The Figure

Figure is the top-level container for everything on a canvas. It was obtained from the global Figure factory.

```
fig = plt.figure(num=None, figsize=None,
      dpi=None, facecolor=None,
      edgecolor=None)
```

num – integer or string identifier of figure
    if num exists, it is selected
    if num is None, a new one is allocated
figsize – tuple of (width, height) in inches
dpi – dots per inch
facecolor – background; edgecolor – border

Iterating over the open figures

```
for i in plt.get_fignums():
    fig = plt.figure(i) # get the figure
    print (fig.number)  # do something
```

Close a figure

```
plt.close(fig.number) # close figure
plt.close()     # close the current figure
plt.close(i)    # close figure numbered i
plt.close(name) # close figure by str name
plt.close('all')# close all figures
```

An Axes or Subplot (a subclass of Axes)

An Axes is a container class for a specific plot. A figure may contain many Axes and/or Subplots. Subplots are laid out in a grid within the Figure. Axes can be placed anywhere on the Figure. There are a number of methods that yield an Axes, including:

```
ax = fig.add_subplot(2,2,1) # row-col-num
ax = fig.add_axes([0.1,0.1,0.8,0.8])
```

All at once

We can use the subplots factory to get the Figure and all the desired Axes at once.

```
fig, ax = plt.subplots()
fig,(ax1,ax2,ax3) = plt.subplots(nrows=3,
    ncols=1, sharex=True, figsize=(8,4))
```

Iterating the Axes within a Figure

```
for ax in fig.get_axes():
    pass # do something
```

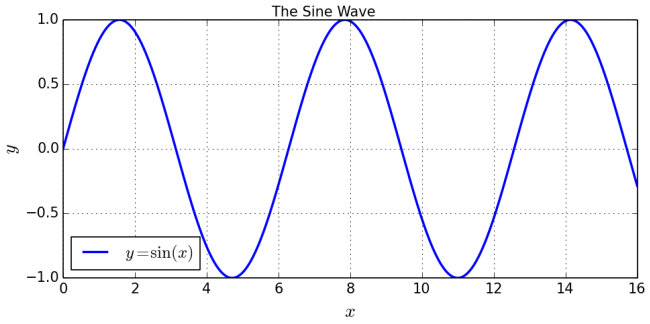
Remove an Axes from a Figure

```
fig.delaxes(ax)
```

Line plots – using ax.plot()

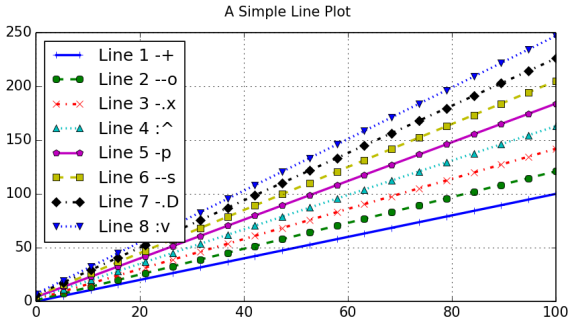
Single plot constructed with Figure and Axes

```
# --- get the data
x = np.linspace(0, 16, 800)
y = np.sin(x)
# --- get an empty figure and add an Axes
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1) # row-col-num
# --- line plot data on the Axes
ax.plot(x, y, 'b-', linewidth=2,
    label=r'$y=\sin(x)$')
# --- add title, labels and legend, etc.
ax.set_ylabel(r'$y$', fontsize=16);
ax.set_xlabel(r'$x$', fontsize=16)
ax.legend(loc='best')
ax.grid(True)
fig.suptitle('The Sine Wave')
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)
```



Multiple lines with markers on a line plot

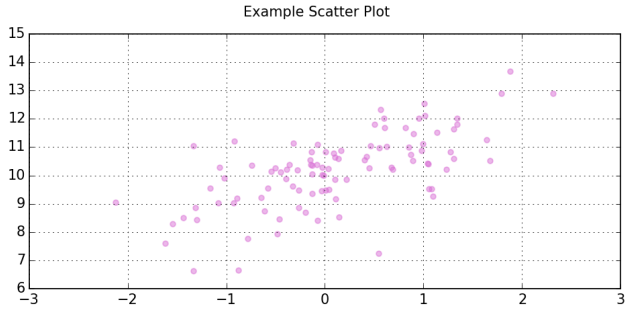
```
# --- get the Figure and Axes all at once
fig, ax = plt.subplots(figsize=(8,4))
# --- plot some lines
N = 8 # the number of lines we will plot
styles =  ['-', '--', '-.', ':']
markers = list('+ox^psDv')
x = np.linspace(0, 100, 20)
for i in range(N): # add line-by-line
    y = x + x/5*i + i
    s = styles[i % len(styles)]
    m = markers[i % len(markers)]
    ax.plot(x, y,
      label='Line '+str(i+1)+' '+s+m,
      marker=m, linewidth=2, linestyle=s)
# --- add grid, legend, title and save
ax.grid(True)
ax.legend(loc='best', prop={'size':'large'})
fig.suptitle('A Simple Line Plot')
fig.savefig('filename.png', dpi=125)
```



Scatter plots – using ax.scatter()

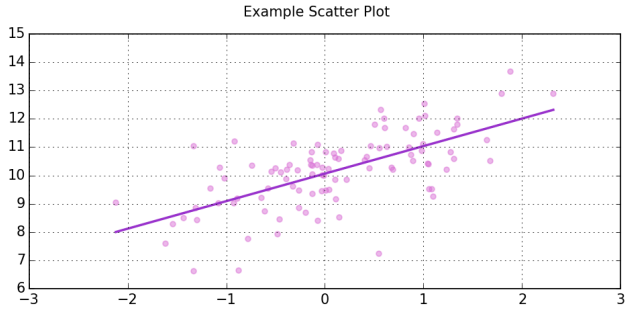
A simple scatter plot

```
x = np.random.randn(100)
y = x + np.random.randn(100) + 10
fig, ax = plt.subplots(figsize=(8, 3))
ax.scatter(x, y, alpha=0.5, color='orchid')
fig.suptitle('Example Scatter Plot')
fig.tight_layout(pad=2);
ax.grid(True)
fig.savefig('filename1.png', dpi=125)
```



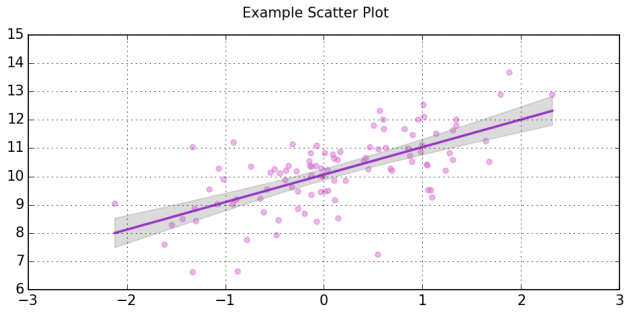
Add a regression line (using statsmodels)

```
import statsmodels.api as sm
x = sm.add_constant(x) # intercept
# Model: y ~ x + c
model = sm.OLS(y, x)
fitted = model.fit()
x_pred = np.linspace(x.min(), x.max(), 50)
x_pred2 = sm.add_constant(x_pred)
y_pred = fitted.predict(x_pred2)
ax.plot(x_pred, y_pred, '-',
    color='darkorchid', linewidth=2)
fig.savefig('filename2.png', dpi=125)
```



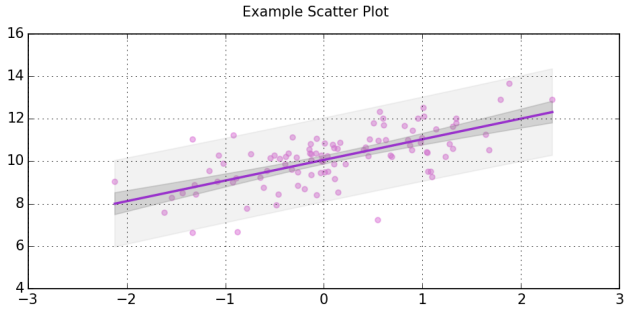
Add confidence bands for the regression line

```
y_hat = fitted.predict(x)
y_err = y - y_hat
mean_x = x.T[1].mean()
n = len(x)
dof = n - fitted.df_model - 1
from scipy import stats
t = stats.t.ppf(1-0.025, df=dof) # 2-tail
s_err = np.sum(np.power(y_err, 2))
conf = t * np.sqrt((s_err/(n-2))*(1.0/n +
    (np.power((x_pred-mean_x),2) /
    ((np.sum(np.power(x_pred,2))) -
    n*(np.power(mean_x,2))))))
upper = y_pred + abs(conf)
lower = y_pred - abs(conf)
ax.fill_between(x_pred, lower, upper,
    color='#888888', alpha=0.3)
fig.savefig('filename3.png', dpi=125)
```



Add a prediction interval for the regression line

```
from statsmodels.sandbox.regression.predstd\
 import wls_prediction_std
sdev, lower, upper =
wls_prediction_std(fitted,
    exog=x_pred2, alpha=0.05)
ax.fill_between(x_pred, lower, upper,
    color='#888888', alpha=0.1)
fig.savefig('filename4.png', dpi=125)
```

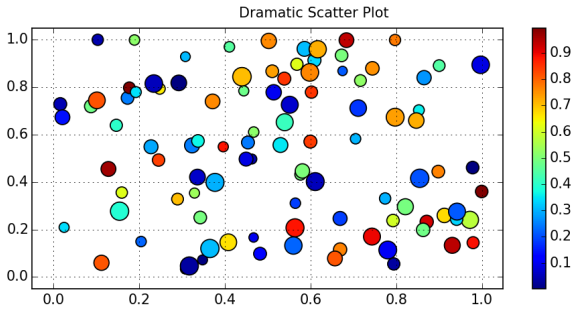


Note: The confidence interval relates to the location of the regression line. The predication interval relates to the location of data points around the regression line.

Changing the marker size and colour

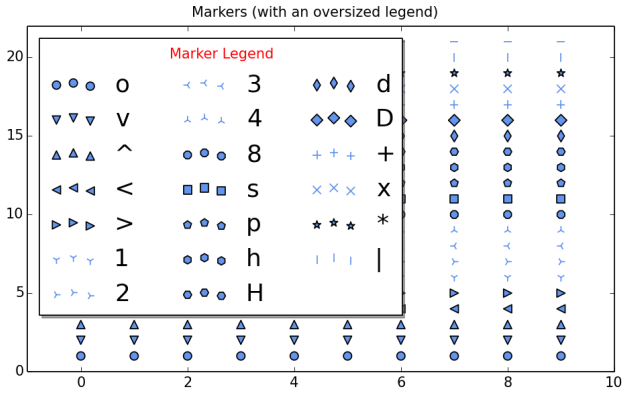
```
N = 100
x = np.random.rand(N)
y = np.random.rand(N)
size = ((np.random.rand(N) + 1) * 8) ** 2
colours = np.random.rand(N)
fig, ax = plt.subplots(figsize=(8,4))
l = ax.scatter(x, y, s=size, c=colours)
fig.colorbar(l)
ax.set_xlim((-0.05, 1.05))
ax.set_ylim((-0.05, 1.05))
fig.suptitle('Dramatic Scatter Plot')
fig.tight_layout(pad=2);
ax.grid(True)
fig.savefig('filename.png', dpi=125)
```

Note: matplotlib has a huge range of colour maps in addition to the default used here.



Changing the marker symbol

```
fig, ax = plt.subplots(figsize=(8,5))
markers = list('ov^<>12348sphHdD+x*|_')
N = 10
for i, m in enumerate(markers):
    x = np.arange(N)
    y = np.repeat(i+1, N)
    ax.scatter(x, y, marker=m, label=m,
        s=50, c='cornflowerblue')

ax.set_xlim((-1,N))
ax.set_ylim((0,len(markers)+1))
ax.legend(loc='upper left', ncol=3,
    prop={'size':'xx-large'},
    shadow=True, title='Marker Legend')
ax.get_legend().get_title().set_color("red")
fig.suptitle('Markers ' +
    '(with an oversized legend)')
fig.tight_layout(pad=2);
fig.savefig('filename.png', dpi=125)
```

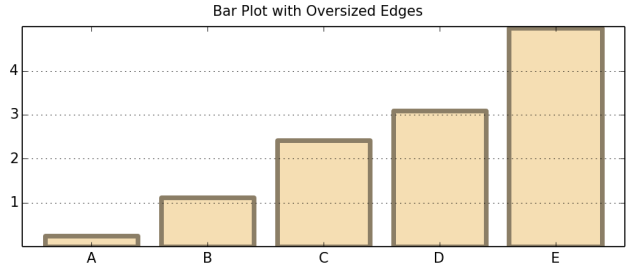


Bar plots – using ax.bar() and ax.barh()

A simple bar chart

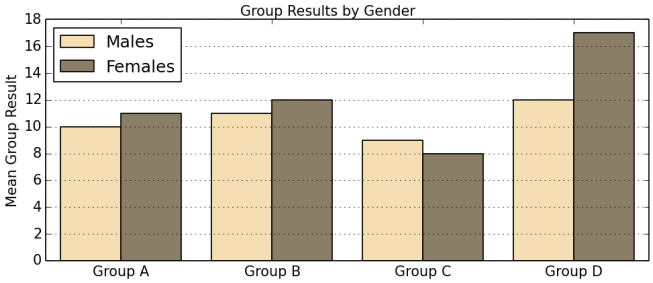
The bars in a bar-plot are placed to the right of the bar x-axis location by default. Centred labels require a little jiggling with the bar and label positions.

```
# --- get the data
N = 5
labels = list('ABCDEFGHIJKLM'[0:N])
data = np.array(range(N)) +
np.random.rand(N)
# --- plot the data
fig, ax = plt.subplots(figsize=(8, 3.5))
width = 0.8;
tickLocations = np.arange(N)
rectLocations = tickLocations-(width/2.0)
ax.bar(rectLocations, data, width,
    color='wheat',
    edgecolor='#8B7E66', linewidth=4.0)
# --- pretty-up the plot
ax.set_xticks(ticks= tickLocations)
ax.set_xticklabels(labels)
ax.set_xlim(min(tickLocations)-0.6,
    max(tickLocations)+0.6)
ax.set_yticks(range(N)[1:])
ax.set_ylim((0,N))
ax.yaxis.grid(True)
# --- title and save
fig.suptitle("Bar Plot with " +
    "Oversized Edges")
fig.tight_layout(pad=2)
fig.savefig('filename.png', dpi=125)
```



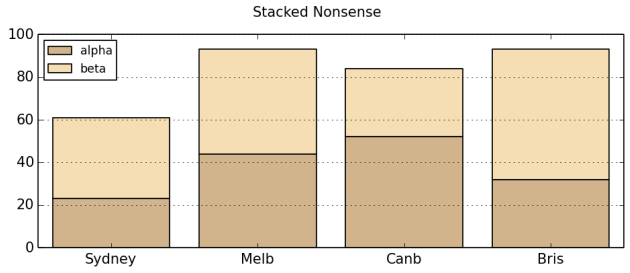
Side by side bar chart

```
# --- get the data
before = np.array([10, 11, 9, 12])
after = np.array([11, 12, 8, 17])
labels=['Group '+x for x in list('ABCD')]
# --- the plot – left then right
fig, ax = plt.subplots(figsize=(8, 3.5))
width = 0.4 # bar width
xlocs = np.arange(len(before))
ax.bar(xlocs-width, before, width,
     color='wheat', label='Males')
ax.bar(xlocs, after, width,
     color='#8B7E66', label='Females')
# --- labels, grids and title, then save
ax.set_xticks(ticks=range(len(before)))
ax.set_xticklabels(labels)
ax.yaxis.grid(True)
ax.legend(loc='best')
ax.set_ylabel('Mean Group Result')
fig.suptitle('Group Results by Gender')
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)
```



Stacked bar

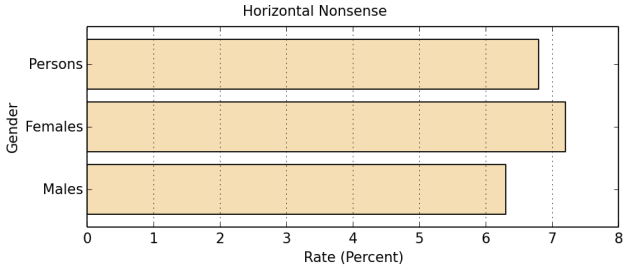
```
# --- get some data
alphas = np.array( [23, 44, 52, 32] )
betas = np.array( [38, 49, 32, 61] )
labels = ['Sydney', 'Melb', 'Canb', 'Bris']
# --- the plot
fig, ax = plt.subplots(figsize=(8, 3.5))
width = 0.8;
xlocations=np.array(range(len(alphas)+2))
adjlocs = xlocations[1:-1] - width/2.0
ax.bar(adjlocs, alphas, width,
    label='alpha', color='tan')
ax.bar(adjlocs, betas, width,
    label='beta', color='wheat',
    bottom=alphas)
# --- pretty-up and save
ax.set_xticks(ticks=xlocations[1:-1])
ax.set_xticklabels(labels)
ax.yaxis.grid(True)
ax.legend(loc='best', prop={'size':'small'})
fig.suptitle("Stacked Nonsense")
fig.tight_layout(pad=2)
fig.savefig('filename.png', dpi=125)
```

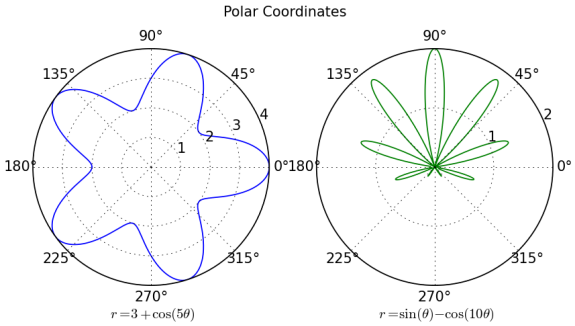


Horizontal bar charts

Just as tick placement needs to be managed with vertical bars; so with horizontal bars (which are above the y-tick mark)

```
labels = ['Males', 'Females', 'Persons']
data = [6.3, 7.2, 6.8]
width = 0.8
yTickPos = np.arange(len(data))
yBarPos = yTickPos - (width/2.0)
fig, ax = plt.subplots(figsize=(8, 3.5))
ax.barh(yBarPos,data,width,color='wheat')
ax.set_yticks(ticks= yTickPos)
ax.set_yticklabels(labels)
ax.set_ylim((min(yTickPos)-0.6,
    max(yTickPos)+0.6))
ax.xaxis.grid(True)
ax.set_ylabel('Gender');
ax.set_xlabel('Rate (Percent)')
fig.suptitle("Horizontal Nonsense")
fig.tight_layout(pad=2)
fig.savefig('filename.png', dpi=125)
```

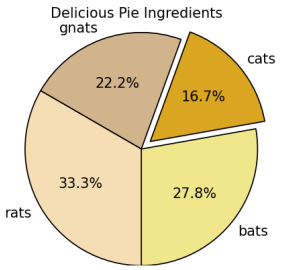




Pie Chart – using ax.pie()

As nice as pie

```
# --- get some data
data = np.array([5,3,4,6])
labels = ['bats', 'cats', 'gnats', 'rats']
explode = (0, 0.1, 0, 0) # explode cats
colrs=['khaki', 'goldenrod', 'tan', 'wheat']
# --- the plot
fig, ax = plt.subplots(figsize=(8, 3.5))
ax.pie(data, explode=explode,
    labels=labels, autopct='%1.1f%%',
    startangle=270, colors=colrs)
ax.axis('equal') # keep it a circle
# --- tidy-up and save
fig.suptitle("Delicious Pie Ingredients")
fig.savefig('filename.png', dpi=125)
```



Polar – using ax.plot()

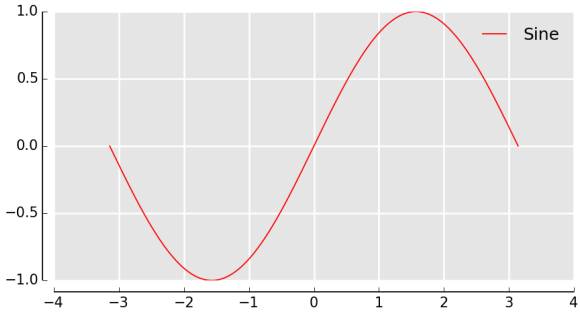
Polar coordinates

```
# --- theta
theta = np.linspace(-np.pi, np.pi, 800)
# --- get us a Figure
fig = plt.figure(figsize=(8,4))
# --- left hand plot
ax = fig.add_subplot(1,2,1, polar=True)
r = 3 + np.cos(5*theta)
ax.plot(theta, r)
ax.set_yticks([1,2,3,4])
# --- right hand plot
ax = fig.add_subplot(1,2,2, polar=True)
r = (np.sin(theta)) - (np.cos(10*theta))
ax.plot(theta, r, color='green')
ax.set_yticks([1,2])
# --- title, explanatory text and save
fig.suptitle('Polar Coordinates')
fig.text(x=0.24, y=0.05,
   s=r'$r = 3 + \cos(5 \theta)$')
fig.text(x=0.64, y=0.05,
   s=r'$r = \sin(\theta) - \cos(10' +
       r'\theta)$')
fig.savefig('filename.png', dpi=125)
```

Plot spines

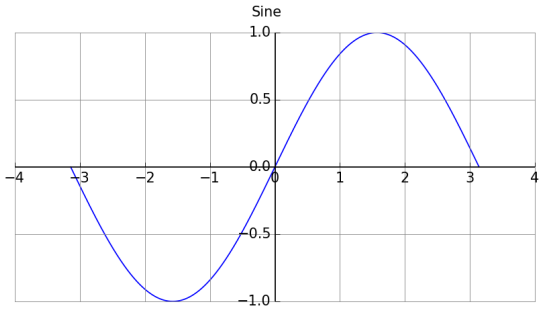
Hiding the top and right spines

```
x = np.linspace(-np.pi, np.pi, 800)
y = np.sin(x)
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(x, y, label='Sine', color='red')
ax.set_axis_bgcolor('#e5e5e5')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.spines['left'].set_position(
    ('outward',10))
ax.spines['bottom'].set_position(
    ('outward',10))
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# do the ax.grid() after setting ticks
ax.grid(b=True, which='both',
    color='white', linestyle='-',
    linewidth=1.5)
ax.set_axisbelow(True)
ax.legend(loc='best', frameon=False)
fig.savefig('filename.png', dpi=125)
```



Spines in the middle

```
x = np.linspace(-np.pi, np.pi, 800)
y = np.sin(x)
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(x, y, label='Sine')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position((
    'data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position((
    'data',0))
ax.grid(b=True, which='both',
    color='#888888', linestyle='-',
    linewidth=0.5)
fig.suptitle('Sine')
fig.savefig('filename.png', dpi=125)
```

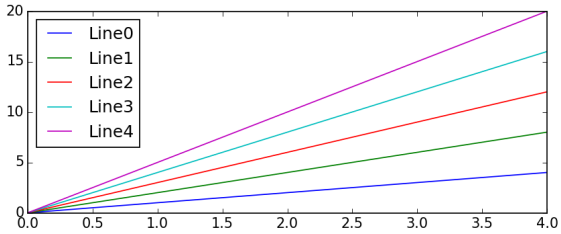


Legends

Legend within the plot

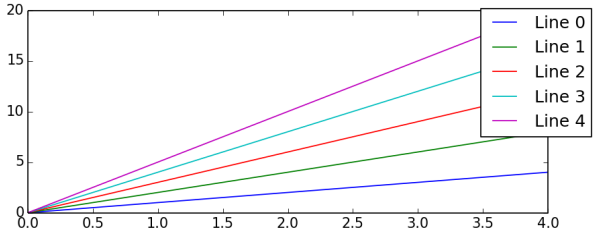
Use the 'loc' argument to place the legend

```
N = 5
x = np.arange(N)
fig, ax = plt.subplots(figsize=(8, 3))
for j in range(5):
  ax.plot(x, x*(j+1),label='Line'+str(j))

ax.legend(loc='upper left')
fig.savefig('filename.png', dpi=125)
```



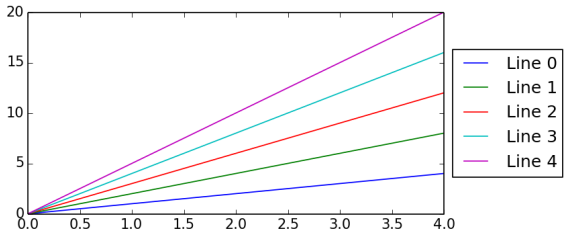
Legend slightly outside of the plot

```
N = 5
x = np.arange(N)
fig, ax = plt.subplots(figsize=(8, 3))
for j in range(5):
    ax.plot(x, x*(j+1),
        label='Line '+str(j))

ax.legend(bbox_to_anchor=(1.1, 1.05))
fig.savefig('filename.png', dpi=125)
```



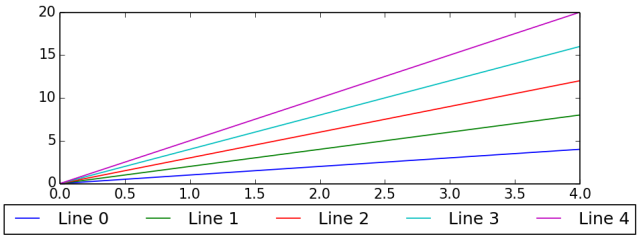
Legend to the right of the plot

```
N = 5
x = np.arange(N)
fig, ax = plt.subplots(figsize=(8, 3))
for j in range(5):
    ax.plot(x, x*(j+1),
        label='Line '+str(j))

box = ax.get_position()  # Shrink plot
ax.set_position([box.x0, box.y0,
    box.width * 0.8, box.height])
ax.legend(bbox_to_anchor=(1, 0.5),
    loc='center left')   # Put legend
fig.savefig('filename.png', dpi=125)
```



Legend below the plot

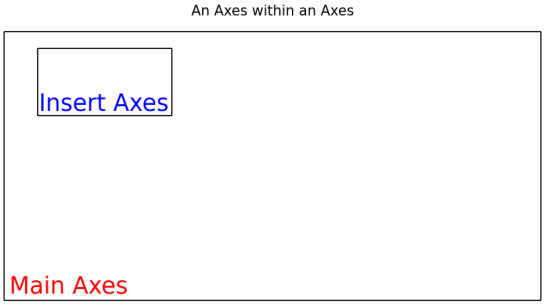
```
N = 5
x = np.arange(N)
fig, ax = plt.subplots(figsize=(8, 3))
for j in range(5):
    ax.plot(x, x*(j+1),
        label='Line '+str(j))

box = ax.get_position()
ax.set_position([box.x0,
    box.y0 + box.height * 0.15,
    box.width, box.height * 0.85])
ax.legend(bbox_to_anchor=(0.5, -0.075),
    loc='upper center', ncol=N)
fig.savefig('filename.png', dpi=125)
```



Multiple plots on a canvas

Using Axes to place a plot within a plot

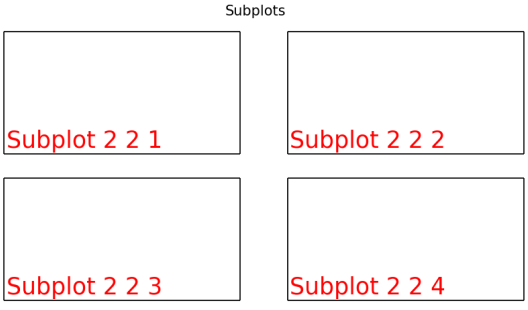
```
fig = plt.figure(figsize=(8,4))
fig.text(x=0.01, y=0.01, s='Figure',
    color='#888888', ha='left',
    va='bottom', fontsize=20)
# --- Main Axes
ax = fig.add_axes([0.1,0.1,0.8,0.8])
ax.text(x=0.01, y=0.01, s='Main Axes',
    color='red', ha='left', va='bottom',
    fontsize=20)
ax.set_xticks([]); ax.set_yticks([])
# --- Insert Axes
ax= fig.add_axes([0.15,0.65,0.2,0.2])
ax.text(x=0.01, y=0.01, s='Insert Axes',
    color='blue', ha='left', va='bottom',
    fontsize=20)
ax.set_xticks([]); ax.set_yticks([])
fig.suptitle('An Axes within an Axes')
fig.savefig('filename.png', dpi=125)
```



Figure

Simple subplot grid layouts

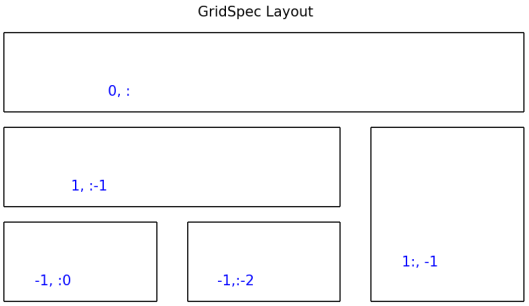
```
fig = plt.figure(figsize=(8,4))
fig.text(x=0.01, y=0.01, s='Figure',
    color='#888888', ha='left',
    va='bottom', fontsize=20)

for i in range(4):
    # fig.add_subplot(nrows, ncols, num)
    ax = fig.add_subplot(2, 2, i+1)
    ax.text(x=0.01, y=0.01,
        s='Subplot 2 2 '+str(i+1),
        color='red', ha='left',
        va='bottom', fontsize=20)
    ax.set_xticks([]); ax.set_yticks([])

ax.set_xticks([]); ax.set_yticks([])
fig.suptitle('Subplots')
fig.savefig('filename.png', dpi=125)
```



Figure

Using GridSpec layouts (like list slicing)

```
import matplotlib.gridspec as gs
gs = gs.GridSpec(3, 3) # nrows, ncols
fig = plt.figure(figsize=(8,4))
fig.text(x=0.01, y=0.01, s='Figure',
    color='#888888', ha='left',
    va='bottom', fontsize=20)
ax1 = fig.add_subplot(gs[0, :]) # row,col
ax1.text(x=0.2,y=0.2,s='0, :', color='b')
ax2 = fig.add_subplot(gs[1,:-1])
ax2.text(x=0.2,y=0.2,s='1, :-1', color='b')
ax3 = fig.add_subplot(gs[1:, -1])
ax3.text(x=0.2,y=0.2, s='1:, -1', color='b')
ax4 = fig.add_subplot(gs[-1,0])
ax4.text(x=0.2,y=0.2, s='-1, :0', color='b')
ax5 = fig.add_subplot(gs[-1,-2])
ax5.text(x=0.2,y=0.2, s='-1,:-2', color='b')
for a in fig.get_axes():
    a.set_xticks([])
    a.set_yticks([])

fig.suptitle('GridSpec Layout')
fig.savefig('filename.png', dpi=125)
```



Figure

Plotting – defaults

Configuration files

Matplotlib uses configuration files to set the defaults. So that you can edit it, the location of the configuration file can be found as follows:

```
print (matplotlib.matplotlib_fname())
```

Configuration settings

The current configuration settings

```
print (matplotlib.rcParams)
```

Change the default settings

```
plt.rc('figure', figsize=(8,4), dpi=125,
    facecolor='white', edgecolor='white')
plt.rc('axes', facecolor='#e5e5e5',
    grid=True, linewidth=1.0,
axisbelow=True)
plt.rc('grid', color='white', linestyle='-',
    linewidth=2.0, alpha=1.0)
plt.rc('xtick', direction='out')
plt.rc('ytick', direction='out')
plt.rc('legend', loc='best')
```

Cheat Sheet: The pandas DataFrame Object

Preliminaries

Always start by importing these Python modules

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas import DataFrame, Series
```

Note: these are the recommended import aliases

Cheat sheet conventions

Code examples

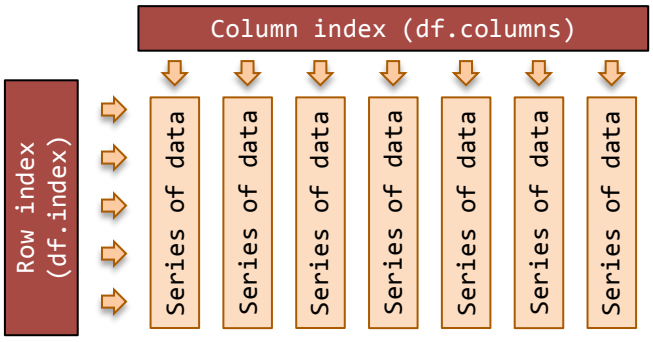
```
# Code examples are found in yellow boxes
# These are designed to be cut and paste
```

In the code examples, typically I use:

- `s` to represent a pandas Series object;
- `df` to represent a pandas DataFrame object;
- `idx` to represent a pandas Index object.
- Also: `t` – tuple, `l` – list, `b` – Boolean, `i` – integer, `a` – numpy array, st – string, d – dictionary, etc.

The conceptual model

DataFrame object: The pandas DataFrame is a two-dimensional table of data with column and row indexes (something like a spread sheet). The columns are made up of pandas Series objects (more below).



A DataFrame has two Indexes:

- Typically, the column index (df.columns) is a list of strings (observed variable names) or (less commonly) integers
- Typically, the row index (df.index) might be:
  - Integers - for case or row numbers;
  - Strings – for case names; or
  - DatetimeIndex or PeriodIndex – for time series

Series object: an ordered, one-dimensional array of data with an index. All the data in a Series is of the same data type. Series arithmetic is vectorised after first aligning the Series index for each of the operands.

```
s1 = Series(range(0,4))  # -> 0, 1, 2, 3
s2 = Series(range(1,5))  # -> 1, 2, 3, 4
s3 = s1 + s2             # -> 1, 3, 5, 7
```

Get your data into a DataFrame

Instantiate an empty DataFrame

```
df = DataFrame()
```

Load a DataFrame from a CSV file

```
df = pd.read_csv('file.csv')# often works
df = pd.read_csv('file.csv', header=0,
      index_col=0, quotechar='"',sep=':',
      na_values = ['na', '-', '.', ''])
```

Note: refer to pandas docs for all arguments

Get data from inline CSV text to a DataFrame

```
from io import StringIO
data = """, Animal, Cuteness, Desirable
row-1,      dog,    8.7,      True
row-2,      cat,    9.5,      True
row-3,      bat,    2.6,      False"""
df = pd.read_csv(StringIO(data),
      header=0, index_col=0,
      skipinitialspace=True)
```

Note: *`skipinitialspace=True`* allows a pretty layout

Load DataFrames from a Microsoft Excel file

```
# Each Excel sheet in a Python dictionary
workbook = pd.ExcelFile('file.xlsx')
d = {} # start with an empty dictionary
for sheet_name in workbook.sheet_names:
    df = workbook.parse(sheet_name)
    d[sheet_name] = df
```

Note: the parse() method takes many arguments like read_csv() above. Refer to the pandas documentation.

Load a DataFrame from a MySQL database

```
import pymysql
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://'
    +'USER:PASSWORD@HOST/DATABASE')
df = pd.read_sql_table('table', engine)
```

Data in Series then combine into a DataFrame

```
# Example 1 ...
s1 = Series(range(6))
s2 = s1 * s1
s2.index = s2.index + 2# misalign indexes
df = pd.concat([s1, s2], axis=1)

# Example 2 ...
s3 = Series({'Tom':1, 'Dick':4, 'Har':9})
s4 = Series({'Tom':3, 'Dick':2, 'Mar':5})
df = pd.concat({'A':s3, 'B':s4 }, axis=1)
```

Note: 1st method has in integer column labels
Note: 2nd method does not guarantee col order
Note: index alignment on DataFrame creation

Get a DataFrame from a Python dictionary

```
# default --- assume data is in columns
df = DataFrame({
        'col0' : [1.0, 2.0, 3.0, 4.0],
        'col1' : [100, 200, 300, 400]
    })
```

Get a DataFrame from data in a Python dictionary

```
# --- use helper method for data in rows
df = DataFrame.from_dict({ # data by row
        # rows as python dictionaries
        'row0' : {'col0':0, 'col1':'A'},
        'row1' : {'col0':1, 'col1':'B'}
    }, orient='index')

df = DataFrame.from_dict({ # data by row
        # rows as python lists
        'row0' : [1, 1+1j, 'A'],
        'row1' : [2, 2+2j, 'B']
    }, orient='index')
```

Create play/fake data (useful for testing)

```
# --- simple - default integer indexes
df = DataFrame(np.random.rand(50,5))

# --- with a time-stamp row index:
df = DataFrame(np.random.rand(500,5))
df.index = pd.date_range('1/1/2005',
       periods=len(df), freq='M')

# --- with alphabetic row and col indexes
#          and a "groupable" variable
import string
import random
r = 52 # note: min r is 1; max r is 52
c = 5
df = DataFrame(np.random.randn(r, c),
   columns = ['col'+str(i) for i in
         range(c)],
   index = list((string. ascii_uppercase+
         string.ascii_lowercase)[0:r]))
df['group'] = list(
   ''.join(random.choice('abcde')
         for _ in range(r)) )
```

Saving a DataFrame

Saving a DataFrame to a CSV file

```
df.to_csv('name.csv', encoding='utf-8')
```

Saving DataFrames to an Excel Workbook

```
from pandas import ExcelWriter
writer = ExcelWriter('filename.xlsx')
df1.to_excel(writer,'Sheet1')
df2.to_excel(writer,'Sheet2')
writer.save()
```

Saving a DataFrame to MySQL

```
import pymysql
from sqlalchemy import create_engine
e = create_engine('mysql+pymysql://' +
    'USER:PASSWORD@HOST/DATABASE')
df.to_sql('TABLE',e, if_exists='replace')
```

Note: if_exists → 'fail', 'replace', 'append'

Saving to Python objects

```
d = df.to_dict()          # to dictionary
str = df.to_string()      # to string
m = df.as_matrix()        # to numpy matrix
```

Working with the whole DataFrame

Peek at the DataFrame contents/structure

```
df.info()           # index & data types
dfh = df.head(i)    # get first i rows
dft = df.tail(i)    # get last i rows
dfs = df.describe() # summary stats cols
top_left_corner_df = df.iloc[:4, :4]
```

DataFrame non-indexing attributes

```
dfT = df.T    # transpose rows and cols
l = df.axes   # list row and col indexes
(r, c) = df.axes    # from above
s = df.dtypes # Series column data types
b = df.empty  # True for empty DataFrame
i = df.ndim   # number of axes (it is 2)
t = df.shape  # (row-count, column-count)
i = df.size   # row-count * column-count
a = df.values # get a numpy array for df
```

DataFrame utility methods

```
df = df.copy() # copy a DataFrame
df = df.rank() # rank each col (default)
df = df.sort_values(by=col)
df = df.sort_values(by=[col1, col2])
df = df.sort_index()
df = df.astype(dtype) # type conversion
```

DataFrame iteration methods

```
df.iteritems()# (col-index, Series) pairs
df.iterrows() # (row-index, Series) pairs

# example ... iterating over columns
for (name, series) in df.iteritems():
    print('Col name: ' + str(name))
    print('First value: ' +
        str(series.iat[0]) + '\n')
```

Maths on the whole DataFrame (not a complete list)

```
df = df.abs()  # absolute values
df = df.add(o) # add df, Series or value
s = df.count() # non NA/null values
df = df.cummax()  # (cols default axis)
df = df.cummin()  # (cols default axis)
df = df.cumsum()  # (cols default axis)
df = df.diff() # 1st diff (col def axis)
df = df.div(o) # div by df, Series, value
df = df.dot(o) # matrix dot product
s = df.max()   # max of axis (col def)
s = df.mean()  # mean (col default axis)
s = df.median()# median (col default)
s = df.min()   # min of axis (col def)
df = df.mul(o) # mul by df Series val
s = df.sum()   # sum axis (cols default)
df = df.where(df > 0.5, other=np.nan)
```

Note: The methods that return a series default to working on columns.

DataFrame select/filter rows/cols on label values

```
df = df.filter(items=['a', 'b']) # by col
df = df.filter(items=[5], axis=0) #by row
df = df.filter(like='x') # keep x in col
df = df.filter(regex='x') # regex in col
df = df.select(lambda x: not x%5)#5th rows
```

Note: select takes a Boolean function, for cols: axis=1
Note: filter defaults to cols; select defaults to rows

Working with Columns

Each DataFrame column is a pandas Series object

Get column index and labels

```
idx = df.columns        # get col index
label = df.columns[0]   # first col label
l = df.columns.tolist() # list col labels
```

Change column labels

```
df.rename(columns={'old1':'new1',
    'old2':'new2'}, inplace=True)
```

Note: can rename multiple columns at once.

Selecting columns

```
s = df['colName'] # select col to Series
df = df[['colName']] # select col to df
df = df[['a','b']]    # select 2 or more
df = df[['c','a','b']]# change col order
s = df[df.columns[0]] # select by number
df = df[df.columns[[0, 3, 4]] # by number
s = df.pop('c') # get col & drop from df
```

Selecting columns with Python attributes

```
s = df.a            # same as s = df['a']
# cannot create new columns by attribute
df.existing_column = df.a / df.b
df['new_column'] = df.a / df.b
```

Trap: column names must be valid identifiers.

Adding new columns to a DataFrame

```
df['new_col'] = range(len(df))
df['new_col'] = np.repeat(np.nan,len(df))
df['random'] = np.random.rand(len(df))
df['index_as_col'] = df.index
df1[['b','c']] = df2[['e','f']]
df3 = df1.append(other=df2)
```

Trap: When adding an indexed pandas object as a new column, only items from the new series that have a corresponding index in the DataFrame will be added. The receiving DataFrame is **not** extended to accommodate the new series. To merge, see below.
Trap: when adding a python list or numpy array, the column will be added by integer position.

Swap column contents – change column order

```
df[['B', 'A']] = df[['A', 'B']]
```

Dropping (deleting) columns (mostly by label)

```
df = df.drop('col1', axis=1)
df.drop('col1', axis=1, inplace=True)
df = df.drop(['col1','col2'], axis=1)
s = df.pop('col') # drops from frame
del df['col'] # even classic python works
df.drop(df.columns[0], inplace=True)
```

Vectorised arithmetic on columns

```
df['proportion']=df['count']/df['total']
df['percent'] = df['proportion'] * 100.0
```

Apply numpy mathematical functions to columns

```
df['log_data'] = np.log(df['col1'])
```

Note: Many more numpy mathematical functions.
Hint: Prefer pandas math over numpy where you can.

Set column values set based on criteria

```
df['b']=df['a'].where(df['a']>0,other=0)
df['d']=df['a'].where(df.b!=0,other=df.c)
```

Note: where other can be a Series or a scalar

Data type conversions

```
st =  df['col'].astype(str)# Series dtype
a = df['col'].values       # numpy array
pl = df['col'].tolist()    # python list
```

Note: useful dtypes for Series conversion: int, float, str
Trap: index lost in conversion from Series to array or list

Common column-wide methods/attributes

```
value = df['col'].dtype  # type of data
value = df['col'].size   # col dimensions
value = df['col'].count()# non-NA count
value = df['col'].sum()
value = df['col'].prod()
value = df['col'].min()
value = df['col'].max()
value = df['col'].mean() # also median()
value = df['col'].cov(df['col2'])
s =     df['col'].describe()
s =     df['col'].value_counts()
```

Find index label for min/max values in column

```
label =  df['col1'].idxmin()
label =  df['col1'].idxmax()
```

Common column element-wise methods

```
s = df['col'].isnull()
s = df['col'].notnull() # not isnull()
s = df['col'].astype(float)
s = df['col'].abs()
s = df['col'].round(decimals=0)
s = df['col'].diff(periods=1)
s = df['col'].shift(periods=1)
s = df['col'].to_datetime()
s = df['col'].fillna(0) # replace NaN w 0
s = df['col'].cumsum()
s = df['col'].cumprod()
s = df['col'].pct_change(periods=4)
s = df['col'].rolling_sum(periods=4,
                          window=4)
```

Note: also rolling_min(), rolling_max(), and many more.

Append a column of row sums to a DataFrame

```
df['Total'] = df.sum(axis=1)
```

Note: also means, mins, maxs, etc.

Multiply every column in DataFrame by Series

```
df = df.mul(s, axis=0) # on matched rows
```

Note: also add, sub, div, etc.

Selecting columns with .loc, .iloc and .ix

```
df = df.loc[:, 'col1':'col2'] # inclusive
df = df.iloc[:, 0:2]          # exclusive
```

Get the integer position of a column index label

```
j = df.columns.get_loc('col_name')
```

Test if column index values are unique/monotonic

```
if df.columns.is_unique: pass # ...
b = df.columns.is_monotonic_increasing
b = df.columns.is_monotonic_decreasing
```

Working with rows

Get the row index and labels

```
idx = df.index              # get row index
label = df.index[0]         # 1st row label
lst = df.index.tolist()     # get as a list
```

Change the (row) index

```
df.index = idx        # new ad hoc index
df = df.set_index('A')# col A new index
df = df.set_index(['A', 'B'])# MultiIndex
df = df.reset_index() # replace old w new
# note: old index stored as a col in df
df.index = range(len(df)) # set with list
df = df.reindex(index=range(len(df)))
df = df.set_index(keys=['r1','r2','etc'])
df.rename(index={'old':'new'},
                          inplace=True)
```

Adding rows

```
df = original_df.append(more_rows_in_df)
```

Hint: convert to a DataFrame and then append. Both DataFrames should have same column labels.

Dropping rows (by name)

```
df = df.drop('row_label')
df = df.drop(['row1','row2']) # multi-row
```

Boolean row selection by values in a column

```
df = df[df['col2'] >= 0.0]
df = df[(df['col3']>=1.0) |
                       (df['col1']<0.0)]
df = df[df['col'].isin([1,2,5,7,11])]
df = df[~df['col'].isin([1,2,5,7,11])]
df = df[df['col'].str.contains('hello')]
```

Trap: bitwise "or", "and" "not; (ie. | & ~) co-opted to be Boolean operators on a Series of Boolean
Trap: need parentheses around comparisons.

Selecting rows using isin over multiple columns

```
# fake up some data
data = {1:[1,2,3], 2:[1,4,9], 3:[1,8,27]}
df = DataFrame(data)

# multi-column isin
lf = {1:[1, 3], 3:[8, 27]} # look for
f = df[df[list(lf)].isin(lf).all(axis=1)]
```

Selecting rows using an index

```
idx = df[df['col'] >= 2].index
print(df.ix[idx])
```

Select a slice of rows by integer position
[inclusive-from : exclusive-to [: step]]
default start is 0; default end is len(df)

```
df = df[:]       # copy DataFrame
df = df[0:2]     # rows 0 and 1
df = df[-1:]     # the last row
df = df[2:3]     # row 2 (the third row)
df = df[:-1]     # all but the last row
df = df[::2]     # every 2nd row (0 2 ..)
```

Trap: a single integer without a colon is a column label for integer numbered columns.

Select a slice of rows by label/index
[inclusive-from : inclusive–to [: step]]

```
df = df['a':'c'] # rows 'a' through 'c'
```

Trap: doesn't work on integer labelled rows

Append a row of column totals to a DataFrame

```
# Option 1: use dictionary comprehension
sums = {col: df[col].sum() for col in df}
sums_df = DataFrame(sums,index=['Total'])
df = df.append(sums_df)

# Option 2: All done with pandas
df = df.append(DataFrame(df.sum(),
                    columns=['Total']).T)
```

Iterating over DataFrame rows

```
for (index, row) in df.iterrows(): # pass
```

Trap: row data type may be coerced.

Sorting DataFrame rows values

```
df = df.sort(df.columns[0],
                           ascending=False)
df.sort(['col1', 'col2'], inplace=True)
```

Sort DataFrame by its row index

```
df.sort_index(inplace=True) # sort by row
df = df.sort_index(ascending=False)
```

Random selection of rows

```
import random as r
k = 20 # pick a number
selection = r.sample(range(len(df)), k)
df_sample = df.iloc[selection, :]
```

Note: this sample is not sorted

Drop duplicates in the row index

```
df['index'] = df.index # 1 create new col
df = df.drop_duplicates(cols='index',
        take_last=True)# 2 use new col
del df['index']        # 3 del the col
df.sort_index(inplace=True)# 4 tidy up
```

Test if two DataFrames have same row index

```
len(a)==len(b) and all(a.index==b.index)
```

Get the integer position of a row or col index label

```
i = df.index.get_loc('row_label')
```

Trap: index.get_loc() returns an integer for a unique match. If not a unique match, may return a slice or mask.

Get integer position of rows that meet condition

```
a = np.where(df['col'] >= 2) #numpy array
```

Test if the row index values are unique/monotonic

```
if df.index.is_unique: pass # ...
b = df.index.is_monotonic_increasing
b = df.index.is_monotonic_decreasing
```

Working with cells

Selecting a cell by row and column labels

```
value = df.at['row', 'col']
value = df.loc['row', 'col']
value = df['col'].at['row']      # tricky
```

Note: .at[] fastest label based scalar lookup

Setting a cell by row and column labels

```
df.at['row', 'col'] = value
df.loc['row', 'col'] = value
df['col'].at['row'] = value      # tricky
```

Selecting and slicing on labels

```
df = df.loc['row1':'row3', 'col1':'col3']
```

Note: the "to" on this slice is inclusive.

Setting a cross-section by labels

```
df.loc['A':'C', 'col1':'col3'] = np.nan
df.loc[1:2,'col1':'col2']=np.zeros((2,2))
df.loc[1:2,'A':'C']=othr.loc[1:2,'A':'C']
```

Remember: inclusive "to" in the slice

Selecting a cell by integer position

```
value = df.iat[9, 3]          # [row, col]
value = df.iloc[0, 0]         # [row, col]
value = df.iloc[len(df)-1,
                len(df.columns)-1]
```

Selecting a range of cells by int position

```
df = df.iloc[2:4, 2:4] # subset of the df
df = df.iloc[:5, :5]   # top left corner
s = df.iloc[5, :] # returns row as Series
df = df.iloc[5:6, :] # returns row as row
```

Note: exclusive "to" – same as python list slicing.

Setting cell by integer position

```
df.iloc[0, 0] = value         # [row, col]
df.iat[7, 8] = value
```

Setting cell range by integer position

```
df.iloc[0:3, 0:5] = value
df.iloc[1:3, 1:4] = np.ones((2, 3))
df.iloc[1:3, 1:4] = np.zeros((2, 3))
df.iloc[1:3, 1:4] = np.array([[1, 1, 1],
                              [2, 2, 2]])
```

Remember: exclusive-to in the slice

.ix for mixed label and integer position indexing

```
value = df.ix[5, 'col1']
df = df.ix[1:5, 'col1':'col3']
```

Views and copies

From the manual: Setting a copy can cause subtle errors. The rules about when a view on the data is returned are dependent on NumPy. Whenever an array of labels or a Boolean vector are involved in the indexing operation, the result will be a copy.

Summary: selecting using the Index

Using the DataFrame index to select columns

```
s =  df['col_label']  # returns Series
df = df[['col_label']]# return DataFrame
df = df[['L1', 'L2']] # select with list
df = df[index]        # select with index
df = df[s]            #select with Series
```

Note: the difference in return type with the first two examples above based on argument type (scalar vs list).

Using the DataFrame index to select rows

```
df = df['from':'inc_to']# label slice
df = df[3:7]            # integer slice
df = df[df['col'] > 0.5]# Boolean Series
df = df.loc['label']    # single label
df = df.loc[container]  # lab list/Series
df = df.loc['from':'to']# inclusive slice
df = df.loc[bs]         # Boolean Series
df = df.iloc[0]         # single integer
df = df.iloc[container] # int list/Series
df = df.iloc[0:5]       # exclusive slice
df = df.ix[x]           # loc then iloc
```

Using the DataFrame index to select a cross-section

```
#     r and c can be scalar, list, slice
df.loc[r, c] # label accessor (row, col)
df.iloc[r, c]# integer accessor
df.ix[r, c]  # label access int fallback
df[c].iloc[r]# chained – also for .loc
```

Using the DataFrame index to select a cell

```
#     r and c must be label or integer
df.at[r, c]  # fast scalar label accessor
df.iat[r, c] # fast scalar int accessor
df[c].iat[r] # chained – also for .at
```

DataFrame indexing methods

```
v = df.get_value(r, c)  # get by row, col
df = df.set_value(r,c,v)# set by row, col
df = df.xs(key, axis) # get cross-section
df = df.filter(items, like, regex, axis)
df = df.select(crit, axis)
```

Note: the indexing attributes (.loc, .iloc, .ix, .at .iat) can be used to get and set values in the DataFrame.
Note: the .loc, iloc and .ix indexing attributes can accept python slice objects. But .at and .iat do not.
Note: .loc can also accept Boolean Series arguments
Avoid: chaining in the form df[col_indexer][row_indexer]
Trap: label slices are inclusive, integer slices exclusive.

Some index attributes and methods

```
# --- some Index attributes
b = idx.is_monotonic_decreasing
b = idx.is_monotonic_increasing
b = idx.has_duplicates
i = idx.nlevels     # num of index levels
# --- some Index methods
idx = idx.astype(dtype)# change data type
b = idx.equals(o)  # check for equality
idx = idx.union(o) # union of two indexes
i = idx.nunique()  # number unique labels
label = idx.min()  # minimum label
label = idx.max()  # maximum label
```

Joining/Combining DataFrames

Three ways to join two DataFrames:

- merge (a database/SQL-like join operation)
- concat (stack side by side or one on top of the other)
- combine_first (splice the two together, choosing values from one over the other)

Merge on indexes

```
df_new = pd.merge(left=df1, right=df2,
          how='outer', left_index=True,
          right_index=True)
```

How: 'left', 'right', 'outer', 'inner'
How: outer=union/all; inner=intersection

Merge on columns

```
df_new = pd.merge(left=df1, right=df2,
          how='left', left_on='col1',
          right_on='col2')
```

Trap: When joining on columns, the indexes on the passed DataFrames are ignored.
Trap: many-to-many merges on a column can result in an explosion of associated data.

Join on indexes (another way of merging)

```
df_new = df1.join(other=df2, on='col1',
                  how='outer')
df_new = df1.join(other=df2,on=['a','b'],
                  how='outer')
```

Note: DataFrame.join() joins on indexes by default. DataFrame.merge() joins on common columns by default.

Simple concatenation is often the best

```
df=pd.concat([df1,df2],axis=0)#top/bottom
df = df1.append([df2, df3])   #top/bottom
df=pd.concat([df1,df2],axis=1)#left/right
```

Trap: can end up with duplicate rows or cols
Note: concat has an ignore_index parameter

Combine_first

```
df = df1.combine_first(other=df2)

# multi-combine with python reduce()
df = reduce(lambda x, y:
        x.combine_first(y),
            [df1, df2, df3, df4, df5])
```

Uses the non-null values from df1. The index of the combined DataFrame will be the union of the indexes from df1 and df2.

Groupby: Split-Apply-Combine

The pandas "groupby" mechanism allows us to split the data into groups, apply a function to each group independently and then combine the results.

Grouping

```
gb = df.groupby('cat') # by one columns
gb = df.groupby(['c1','c2']) # by 2 cols
gb = df.groupby(level=0) # multi-index gb
gb = df.groupby(level=['a','b']) # mi gb
print(gb.groups)
```

Note: groupby() returns a pandas groupby object
Note: the groupby object attribute .groups contains a dictionary mapping of the groups.
Trap: NaN values in the group key are automatically dropped – there will never be a NA group.

Iterating groups – usually not needed

```
for name, group in gb:
    print (name)
    print (group)
```

Selecting a group

```
dfa = df.groupby('cat').get_group('a')
dfb = df.groupby('cat').get_group('b')
```

Applying an aggregating function

```
# apply to a column ...
s = df.groupby('cat')['col1'].sum()
s = df.groupby('cat')['col1'].agg(np.sum)
# apply to the every column in DataFrame
s = df.groupby('cat').agg(np.sum)
df_summary = df.groupby('cat').describe()
df_row_1s = df.groupby('cat').head(1)
```

Note: aggregating functions reduce the dimension by one – they include: mean, sum, size, count, std, var, sem, describe, first, last, min, max

Applying multiple aggregating functions

```
gb = df.groupby('cat')

# apply multiple functions to one column
dfx = gb['col2'].agg([np.sum, np.mean])
# apply to multiple fns to multiple cols
dfy = gb.agg({
    'cat': np.count_nonzero,
    'col1': [np.sum, np.mean, np.std],
    'col2': [np.min, np.max]
})
```

Note: gb['col2'] above is shorthand for df.groupby('cat')['col2'], without the need for regrouping.

Transforming functions

```
# transform to group z-scores, which have
# a group mean of 0, and a std dev of 1.
zscore = lambda x: (x-x.mean())/x.std()
dfz = df.groupby('cat').transform(zscore)

# replace missing data with group mean
mean_r = lambda x: x.fillna(x.mean())
dfm = df.groupby('cat').transform(mean_r)
```

Note: can apply multiple transforming functions in a manner similar to multiple aggregating functions above,

Applying filtering functions
Filtering functions allow you to make selections based on whether each group meets specified criteria

```
# select groups with more than 10 members
eleven = lambda x: (len(x['col1']) >= 11)
df11 = df.groupby('cat').filter(eleven)
```

Group by a row index (non-hierarchical index)

```
df = df.set_index(keys='cat')
s = df.groupby(level=0)['col1'].sum()
dfg = df.groupby(level=0).sum()
```

Pivot Tables: working with long and wide data

These features work with and often create hierarchical or multi-level Indexes; (the pandas MultiIndex is powerful and complex).

Pivot, unstack, stack and melt

Pivot tables move from long format to wide format data

```
# Let's start with data in long format
from StringIO import StringIO # python2.7
#from io import StringIO      # python 3
data = """Date,Pollster,State,Party,Est
13/03/2014, Newspoll, NSW,   red,   25
13/03/2014, Newspoll, NSW,   blue,  28
13/03/2014, Newspoll, Vic,   red,   24
13/03/2014, Newspoll, Vic,   blue,  23
13/03/2014, Galaxy,   NSW,   red,   23
13/03/2014, Galaxy,   NSW,   blue,  24
13/03/2014, Galaxy,   Vic,   red,   26
13/03/2014, Galaxy,   Vic,   blue,  25
13/03/2014, Galaxy,   Qld,   red,   21
13/03/2014, Galaxy,   Qld,   blue,  27"""
df = pd.read_csv(StringIO(data),
      header=0, skipinitialspace=True)

# pivot to wide format on 'Party' column
# 1st: set up a MultiIndex for other cols
df1 = df.set_index(['Date', 'Pollster',
    'State'])
# 2nd: do the pivot
wide1 = df1.pivot(columns='Party')

# unstack to wide format on State / Party
# 1st: MultiIndex all but the Values col
df2 = df.set_index(['Date', 'Pollster',
    'State', 'Party'])
# 2nd: unstack a column to go wide on it
wide2 = df2.unstack('State')
wide3 = df2.unstack() # pop last index

# Use stack() to get back to long format
long1 = wide1.stack()
# Then use reset_index() to remove the
# MultiIndex.
long2 = long1.reset_index()

# Or melt() back to long format
# 1st: flatten the column index
wide1.columns = ['_'.join(col).strip()
    for col in wide1.columns.values]
# 2nd: remove the MultiIndex
wdf = wide1.reset_index()
# 3rd: melt away
long3 = pd.melt(wdf, value_vars=
    ['Est_blue', 'Est_red'],
    var_name='Party', id_vars=['Date',
    'Pollster', 'State'])
```

Note: See documentation, there are many arguments to these methods.

Working with dates, times and their indexes

Dates and time – points and spans

With its focus on time-series data, pandas has a suite of tools for managing dates and time: either as a point in time (a Timestamp) or as a span of time (a Period).

```
t = pd.Timestamp('2013-01-01')
t = pd.Timestamp('2013-01-01 21:15:06')
t = pd.Timestamp('2013-01-01 21:15:06.7')
p = pd.Period('2013-01-01', freq='M')
```

Note: Timestamps should be in range 1678 and 2261 years. (Check Timestamp.max and Timestamp.min).

A Series of Timestamps or Periods

```
ts = ['2015-04-01 13:17:27',
        '2014-04-02 13:17:29']

# Series of Timestamps (good)
s = pd.to_datetime(pd.Series(ts))

# Series of Periods (often not so good)
s = pd.Series( [pd.Period(x, freq='M')
        for x in ts] )
s = pd.Series(
        pd.PeriodIndex(ts,freq='S'))
```

Note: While Periods make a very useful index; they may be less useful in a Series.

From non-standard strings to Timestamps

```
t = ['09:08:55.7654-JAN092002',
        '15:42:02.6589-FEB082016']
s = pd.Series(pd.to_datetime(t,
        format="%H:%M:%S.%f-%b%d%Y"))
```

Also: %B = full month name; %m = numeric month; %y = year without century; and more …

Dates and time – stamps and spans as indexes

An index of Timestamps is a DatetimeIndex.
An index of Periods is a PeriodIndex.

```
date_strs = ['2014-01-01', '2014-04-01',
             '2014-07-01', '2014-10-01']

dti = pd.DatetimeIndex(date_strs)

pid = pd.PeriodIndex(date_strs, freq='D')
pim = pd.PeriodIndex(date_strs, freq='M')
piq = pd.PeriodIndex(date_strs, freq='Q')

print (pid[1] - pid[0])  # 90 days
print (pim[1] - pim[0])  #  3 months
print (piq[1] - piq[0])  #  1 quarter

time_strs = ['2015-01-01 02:10:40.12345',
             '2015-01-01 02:10:50.67890']
pis = pd.PeriodIndex(time_strs, freq='U')

df.index = pd.period_range('2015-01',
        periods=len(df), freq='M')

dti = pd.to_datetime(['04-01-2012'],
  dayfirst=True) # Australian date format
pi = pd.period_range('1960-01-01',
  '2015-12-31', freq='M')
```

Hint: unless you are working in less than seconds, prefer PeriodIndex over DateTimeImdex.

Period frequency constants (not a complete list)

Name	Description
U	Microsecond
L	Millisecond
S	Second
T	Minute
H	Hour
D	Calendar day
B	Business day
W-{MON, TUE, …}	Week ending on …
MS	Calendar start of month
M	Calendar end of month
QS-{JAN, FEB, …}	Quarter start with year starting (QS – December)
Q-{JAN, FEB, …}	Quarter end with year ending (Q – December)
AS-{JAN, FEB, …}	Year start (AS - December)
A-{JAN, FEB, …}	Year end (A - December)

From DatetimeIndex to Python datetime objects

```
dti = pd.DatetimeIndex(pd.date_range(
  start='1/1/2011', periods=4, freq='M'))
s = Series([1,2,3,4], index=dti)
na = dti.to_pydatetime()     #numpy array
na = s.index.to_pydatetime() #numpy array
```

Frome Timestamps to Python dates or times

```
df['date'] = [x.date() for x in df['TS']]
df['time'] = [x.time() for x in df['TS']]
```

Note: converts to datatime.date or datetime.time. But does not convert to datetime.datetime.

From DatetimeIndex to PeriodIndex and back

```
df = DataFrame(np.random.randn(20,3))
df.index = pd.date_range('2015-01-01',
       periods=len(df), freq='M')
dfp = df.to_period(freq='M')
dft = dfp.to_timestamp()
```

Note: from period to timestamp defaults to the point in time at the start of the period.

Working with a PeriodIndex

```
pi = pd.period_range('1960-01','2015-12',
                     freq='M')
na = pi.values  # numpy array of integers
lp = pi.tolist() # python list of Periods
sp = Series(pi)# pandas Series of Periods
ss = Series(pi).astype(str) # S of strs
ls = Series(pi).astype(str).tolist()
```

Get a range of Timestamps

```
dr = pd.date_range('2013-01-01',
    '2013-12-31', freq='D')
```

Error handling with dates

```
# 1st example returns string not Timestamp
t = pd.to_datetime('2014-02-30')
# 2nd example returns NaT (not a time)
t = pd.to_datetime('2014-02-30',
      coerce=True)
# NaT like NaN tests True for isnull()
b = pd.isnull(t) # --> True
```

The tail of a time-series DataFrame

```
df = df.last("5M") # the last five months
```

Upsampling and downsampling

```
# upsample from quarterly to monthly
pi = pd.period_range('1960Q1',
        periods=220, freq='Q')
df = DataFrame(np.random.rand(len(pi),5),
        index=pi)
dfm = df.resample('M', convention='end')
# use ffill or bfill to fill with values

# downsample from monthly to quarterly
dfq = dfm.resample('Q', how='sum')
```

Time zones

```
t = ['2015-06-30 00:00:00',
      '2015-12-31 00:00:00']
dti = pd.to_datetime(t
      ).tz_localize('Australia/Canberra')
dti = dti.tz_convert('UTC')
ts = pd.Timestamp('now',
      tz='Europe/London')

# get a list of all time zones
import pyzt
for tz in pytz.all_timezones:
    print tz
```

Note: by default, Timestamps are created without time zone information.

Row selection with a time-series index

```
# start with the play data above
idx = pd.period_range('2015-01',
        periods=len(df), freq='M')
df.index = idx

february_selector = (df.index.month == 2)
february_data = df[february_selector]

q1_data = df[(df.index.month >= 1) &
   (df.index.month <= 3)]

mayornov_data = df[(df.index.month == 5)
   | (df.index.month == 11)]

totals = df.groupby(df.index.year).sum()
```

Also: year, month, day [of month], hour, minute, second, dayofweek [Mon=0 .. Sun=6], weekofmonth, weekofyear [numbered from 1], week starts on Monday], dayofyear [from 1], …

The Series.dt accessor attribute

DataFrame columns that contain datetime-like objects can be manipulated with the .dt accessor attribute

```
t = ['2012-04-14 04:06:56.307000',
     '2011-05-14 06:14:24.457000',
     '2010-06-14 08:23:07.520000']

# a Series of time stamps
s = pd.Series(pd.to_datetime(t))
print(s.dtype)      # datetime64[ns]
print(s.dt.second)  # 56, 24,  7
print(s.dt.month)   #  4,  5,  6

# a Series of time periods
s = pd.Series(pd.PeriodIndex(t,freq='Q'))
print(s.dtype)      # datetime64[ns]
print(s.dt.quarter) #  2,  2,  2
print(s.dt.year)    #  2012, 2011, 2010
```

Working with missing and non-finite data

Working with missing data
Pandas uses the not-a-number construct (np.nan and float('nan')) to indicate missing data. The Python None can arise in data as well. It is also treated as missing data; as is the pandas not-a-time construct (pandas.NaT).

Missing data in a Series

```
s = Series( [8,None,float('nan'),np.nan])
          #[8,     NaN,   NaN,   NaN]
s.isnull() #[False, True,  True,  True]
s.notnull()#[True,  False, False, False]
s.fillna(0)#[8,     0,     0,     0]
```

Missing data in a DataFrame

```
df = df.dropna() # drop all rows with NaN
df = df.dropna(axis=1) # same for cols
df=df.dropna(how='all') #drop all NaN row
df=df.dropna(thresh=2) # drop 2+ NaN in r
# only drop row if NaN in a specified col
df = df.dropna(df['col'].notnull())
```

Recoding missing data

```
df.fillna(0, inplace=True) # np.nan → 0
s = df['col'].fillna(0)    # np.nan → 0
df = df.replace(r'\s+', np.nan,
      regex=True) # white space → np.nan
```

Non-finite numbers
With floating point numbers, pandas provides for positive and negative infinity.

```
s = Series([float('inf'), float('-inf'),
     np.inf, -np.inf])
```

Pandas treats integer comparisons with plus or minus infinity as expected.

Testing for finite numbers
(using the data from the previous example)

```
b = np.isfinite(s)
```

Working with Categorical Data

Categorical data
The pandas Series has an R factors-like data type for encoding categorical data.

```
s = Series(['a','b','a','c','b','d','a'],
                        dtype='category')
df['B'] = df['A'].astype('category')
```

Note: the key here is to specify the "category" data type.
Note: categories will be ordered on creation if they are sortable. This can be turned off. See ordering below.

Convert back to the original data type

```
s = Series(['a','b','a','c','b','d','a'],
                        dtype='category')
s = s.astype('string')
```

Ordering, reordering and sorting

```
s = Series(list('abc'), dtype='category')
print (s.cat.ordered)
s=s.cat.reorder_categories(['b','c','a'])
s = s.sort()
s.cat.ordered = False
```

Trap: category must be ordered for it to be sorted

Renaming categories

```
s = Series(list('abc'), dtype='category')
s.cat.categories = [1, 2, 3] # in place
s = s.cat.rename_categories([4,5,6])
# using a comprehension ...
s.cat.categories = ['Group ' + str(i)
    for i in s.cat.categories]
```

Trap: categories must be uniquely named

Adding new categories

```
s = s.cat.add_categories([4])
```

Removing categories

```
s = s.cat.remove_categories([4])
s.cat.remove_unused_categories() #inplace
```

Working with strings

Working with strings

```
# assume that df['col'] is series of
strings
s = df['col'].str.lower()
s = df['col'].str.upper()
s = df['col'].str.len()

# the next set work like Python
df['col'] += 'suffix'       # append
df['col'] *= 2              # duplicate
s = df['col1'] + df['col2'] # concatenate
```

Most python string functions are replicated in the pandas DataFrame and Series objects.

Regular expressions

```
s = df['col'].str.contains('regex')
s = df['col'].str.startswith('regex')
s = df['col'].str.endswith('regex')
s = df['col'].str.replace('old', 'new')
df['b'] = df.a.str.extract('(pattern)')
```

Note: pandas has many more regex methods.

Basic Statistics

Summary statistics

```
s = df['col1'].describe()
df1 = df.describe()
```

DataFrame – key stats methods

```
df.corr()     # pairwise correlation cols
df.cov()      # pairwise covariance cols
df.kurt()     # kurtosis over cols (def)
df.mad()      # mean absolute deviation
df.sem()      # standard error of mean
df.var()      # variance over cols (def)
```

Value counts

```
s = df['col1'].value_counts()
```

Cross-tabulation (frequency count)

```
ct = pd.crosstab(index=df['a'],
        cols=df['b'])
```

Quantiles and ranking

```
quants = [0.05, 0.25, 0.5, 0.75, 0.95]
q = df.quantile(quants)
r = df.rank()
```

Histogram binning

```
count, bins = np.histogram(df['col1'])
count, bins = np.histogram(df['col'],
              bins=5)
count, bins = np.histogram(df['col1'],
              bins=[-3,-2,-1,0,1,2,3,4])
```

Regression

```
import statsmodels.formula.api as sm
result = sm.ols(formula="col1 ~ col2 +
          col3", data=df).fit()
print (result.params)
print (result.summary())
```

Smoothing example using rolling_apply

```
k3x5 = np.array([1,2,3,3,3,2,1]) / 15.0
s = pd.rolling_apply(df['col1'],
       window=7,
         func=lambda x: (x * k3x5).sum(),
          min_periods=7, center=True)
```

Cautionary note

This cheat sheet was cobbled together by bots roaming the dark recesses of the Internet seeking ursine and pythonic myths. There is no guarantee the narratives were captured and transcribed accurately. You use these notes at your own risk. You have been warned.

Version: This cheat sheet was last updated with Python 3.5 and pandas 0.18.0 in mind.

Errors: If you find any errors, please email me at markthegraph@gmail.com; (but please do not correct my use of Australian-English spelling conventions).

Pure Python

Types

```
a = 2           # integer
b = 5.0         # float
c = 8.3e5       # exponential
d = 1.5 + 0.5j  # complex
e = 4 > 5       # boolean
f = 'word'      # string
```

Lists

```
a = ['red', 'blue', 'green']       # manually initialization
b = list(range(5))                 # initialize from iteratable
c = [nu**2 for nu in b]            # list comprehension
d = [nu**2 for nu in b if nu < 3]  # conditioned list comprehension
e = c[0]                           # access element
f = c[1:2]                         # access a slice of the list
g = ['re', 'bl'] + ['gr']          # list concatenation
h = ['re'] * 5                     # repeat a list
['re', 'bl'].index('re')           # returns index of 're'
're' in ['re', 'bl']               # true if 're' in list
sorted([3, 2, 1])                  # returns sorted list
```

Dictionaries

```
a = {'red': 'rouge', 'blue': 'bleu'}         # dictionary
b = a['red']                                 # translate item
c = [value for key, value in a.items()]      # loop through contents
d = a.get('yellow', 'no translation found')  # return default
```

Strings

```
a = 'red'                      # assignment
char = a[2]                    # access individual characters
'red ' + 'blue'                # string concatenation
'1, 2, three'.split(',')       # split string into list
'.'.join(['1', '2', 'three'])  # concatenate list into string
```

Operators

```
a = 2             # assignment
a += 1 (*=, /=)   # change and assign
3 + 2             # addition
3 / 2             # integer (python2) or float (python3) division
3 // 2            # integer division
3 * 2             # multiplication
3 ** 2            # exponent
3 % 2             # remainder
abs(a)            # absolute value
1 == 1            # equal
2 > 1             # larger
2 < 1             # smaller
1 != 2            # not equal
1 != 2 and 2 < 3  # logical AND
1 != 2 or 2 < 3   # logical OR
not 1 == 2        # logical NOT
'a' in b          # test if a is in b
a is b            # test if objects point to the same memory (id)
```

Control Flow

```
# if/elif/else
a, b = 1, 2
if a + b == 3:
    print('True')
elif a + b == 1:
    print('False')
else:
    print('?')

# for
a = ['red', 'blue', 'green']
for color in a:
    print(color)

# while
number = 1
while number < 10:
    print(number)
    number += 1

# break
number = 1
while True:
    print(number)
    number += 1
    if number > 10:
        break

# continue
for i in range(20):
    if i % 2 == 0:
        continue
    print(i)
```

Functions, Classes, Generators, Decorators

```
# Function groups code statements and possibly
# returns a derived value
def myfunc(a1, a2):
    return a1 + a2

x = myfunc(a1, a2)

# Class groups attributes (data)
# and associated methods (functions)
class Point(object):
    def __init__(self, x):
        self.x = x
    def __call__(self):
        print(self.x)

x = Point(3)

# Generator iterates without
# creating all values at ones
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

x = [i for i in firstn(10)]

# Decorator can be used to modify
# the behaviour of a function
class myDecorator(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        print("call")
        self.f()

@myDecorator
def my_funct():
    print('func')

my_funct()
```

IPython

console

```
<object>?  # Information about the object
<object>.<TAB>  # tab completion

# measure runtime of a function:
%timeit range(1000)
100000 loops, best of 3: 7.76 us per loop

# run scripts and debug
%run
%run -d  # run in debug mode
%run -t  # measures execution time
%run -p  # runs a profiler
%debug  # jumps to the debugger after an exception

%pdb  # run debugger automatically on exception

# examine history
%history
%history ~1/1-5  # lines 1-5 of last session

# run shell commands
!make  # prefix command with "!"

# clean namespace
%reset
```

debugger

```
n              # execute next line
b 42           # set breakpoint in the main file at line 42
b myfile.py:42 # set breakpoint in 'myfile.py' at line 42
c              # continue execution
l              # show current position in the code
p data         # print the 'data' variable
pp data        # pretty print the 'data' variable
s              # step into subroutine
a              # print arguments that a function received
pp locals()    # show all variables in local scope
pp globals()   # show all variables in global scope
```

command line

```
ipython --pdb -- myscript.py argument1 --option1  # debug after exception
ipython -i -- myscript.py argument1 --option1     # console after finish
```

NumPy (`import numpy as np`)

array initialization

```
np.array([2, 3, 4])            # direct initialization
np.empty(20, dtype=np.float32) # single precision array of size 20
np.zeros(200)                  # initialize 200 zeros
np.ones((3,3), dtype=np.int32) # 3 x 3 integer matrix with ones
np.eye(200)                    # ones on the diagonal
np.zeros_like(a)               # array with zeros and the shape of a
np.linspace(0., 10., 100)      # 100 points from 0 to 10
np.arange(0, 100, 2)           # points from 0 to <100 with step 2
np.logspace(-5, 2, 100)        # 100 log-spaced from 1e-5 -> 1e2
np.copy(a)                     # copy array to new memory
```

indexing

```
a = np.arange(100)          # initialization with 0 - 99
a[:3] = 0                   # set the first three indices to zero
a[2:5] = 1                  # set indices 2-4 to 1
a[start:stop:step]          # general form of indexing/slicing
a[None, :]                  # transform to column vector
a[[1, 1, 3, 8]]             # return array with values of the indices
a = a.reshape(10, 10)       # transform to 10 x 10 matrix
a.T                         # return transposed view
b = np.transpose(a, (1, 0)) # transpose array to new axis order
a[a < 2]                    # values with elementwise condition
```

array properties and operations

```
a.shape               # a tuple with the lengths of each axis
len(a)                # length of axis 0
a.ndim                # number of dimensions (axes)
a.sort(axis=1)        # sort array along axis
a.flatten()           # collapse array to one dimension
a.conj()              # return complex conjugate
a.astype(np.int16)    # cast to integer
np.argmax(a, axis=1)  # return index of maximum along a given axis
np.cumsum(a)          # return cumulative sum
np.any(a)             # True if any element is True
np.all(a)             # True if all elements are True
np.argsort(a, axis=1) # return sorted index array along axis
```

boolean arrays

```
a < 2                       # returns array with boolean values
(a < 2) & (b > 10)          # elementwise logical and
(a < 2) | (b > 10)          # elementwise logical or
~a                          # invert boolean array
```

elementwise operations and math functions

```
a * 5             # multiplication with scalar
a + 5             # addition with scalar
a + b             # addition with array b
a / b             # division with b (np.NaN for division by zero)
np.exp(a)         # exponential (complex and real)
np.power(a, b)    # a to the power b
np.sin(a)         # sine
np.cos(a)         # cosine
np.arctan2(a, b)  # arctan(a/b)
np.arcsin(a)      # arcsin
np.radians(a)     # degrees to radians
np.degrees(a)     # radians to degrees
np.var(a)         # variance of array
np.std(a, axis=1) # standard deviation
```

inner / outer products

```
np.dot(a, b)                # inner product: a_mi b_in
np.einsum('ij,kj->ik', a, b) # einstein summation convention
np.sum(a, axis=1)           # sum over axis 1
np.abs(a)                   # return absolute values
a[None, :] + b[:, None]     # outer sum
a[None, :] * b[:, None]     # outer product
np.outer(a, b)              # outer product
np.sum(a * a.T)             # matrix norm
```

reading/ writing files

```
np.fromfile(fname/fobject, dtype=np.float32, count=5)  # binary data from file
np.loadtxt(fname/fobject, skiprows=2, delimiter=',')   # ascii data from file
np.savetxt(fname/fobject, array, fmt='%.5f')           # write ascii data
np.tofile(fname/fobject)                               # write (C) binary data
```

interpolation, integration, optimization

```
np.trapz(a, x=x, axis=1)  # integrate along axis 1
np.interp(x, xp, yp)      # interpolate function xp, yp at points x
np.linalg.lstsq(a, b)     # solve a x = b in least square sense
```

fft

```
np.fft.fft(a)               # complex fourier transform of a
f = np.fft.fftfreq(len(a))  # fft frequencies
np.fft.fftshift(f)          # shifts zero frequency to the middle
np.fft.rfft(a)              # real fourier transform of a
np.fft.rfftfreq(len(a))     # real fft frequencies
```

rounding

```
np.ceil(a)   # rounds to nearest upper int
np.floor(a)  # rounds to nearest lower int
np.round(a)  # rounds to neares int
```

random variables

```
from np.random import normal, seed, rand, uniform, randint
normal(loc=0, scale=2, size=100)  # 100 normal distributed
seed(23032)                       # resets the seed value
rand(200)                         # 200 random numbers in [0, 1)
uniform(1, 30, 200)               # 200 random numbers in [1, 30)
randint(1, 16, 300)               # 300 random integers in [1, 16)
```

Matplotlib (`import matplotlib.pyplot as plt`)

figures and axes

```
fig = plt.figure(figsize=(5, 2))  # initialize figure
ax = fig.add_subplot(3, 2, 2)     # add second subplot in a 3 x 2 grid
fig, axes = plt.subplots(5, 2, figsize=(5, 5)) # fig and 5 x 2 nparray of axes
ax = fig.add_axes([left, bottom, width, height]) # add custom axis
```

figures and axes properties

```
fig.suptitle('title')           # big figure title
fig.subplots_adjust(bottom=0.1, right=0.8, top=0.9, wspace=0.2,
                    hspace=0.5)  # adjust subplot positions
fig.tight_layout(pad=0.1, h_pad=0.5, w_pad=0.5,
                 rect=None)      # adjust subplots to fit into fig
ax.set_xlabel('xbla')           # set xlabel
ax.set_ylabel('ybla')           # set ylabel
ax.set_xlim(1, 2)               # sets x limits
ax.set_ylim(3, 4)               # sets y limits
ax.set_title('blabla')          # sets the axis title
ax.set(xlabel='bla')            # set multiple parameters at once
ax.legend(loc='upper center')   # activate legend
ax.grid(True, which='both')     # activate grid
bbox = ax.get_position()        # returns the axes bounding box
bbox.x0 + bbox.width            # bounding box parameters
```

plotting routines

```
ax.plot(x,y, '-o', c='red', lw=2, label='bla')  # plots a line
ax.scatter(x,y, s=20, c=color)                  # scatter plot
ax.pcolormesh(xx, yy, zz, shading='gouraud')    # fast colormesh
ax.colormesh(xx, yy, zz, norm=norm)             # slower colormesh
ax.contour(xx, yy, zz, cmap='jet')              # contour lines
ax.contourf(xx, yy, zz, vmin=2, vmax=4)         # filled contours
n, bins, patch = ax.hist(x, 50)                 # histogram
ax.imshow(matrix, origin='lower',
          extent=(x1, x2, y1, y2))              # show image
ax.specgram(y, FS=0.1, noverlap=128,
            scale='linear')                     # plot a spectrogram
```

Scipy (`import scipy as sci`)

interpolation

```
from scipy.ndimage import map_coordinates       # interpolates data
pts_new = map_coordinates(data, float_indices,  # at index positions
                          order=3)
```

Integration

```
from scipy.integrate import quad     # definite integral of python
value = quad(func, low_lim, up_lim)  # function/method
```