

Language Fundamentals

Logit Academy

Schedule

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

What Is Python?

ONE LINER

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

PYTHON HIGHLIGHTS

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

Python data analytics platform and environment

IPython

An enhanced
interactive Python shell

STANDARD PYTHON COMMANDS WITH NUMBERED In/Out

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

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

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

TIMING YOUR CODE

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

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

```
...: a = []
```

```
...: for value in xrange(10):
```

```
...:     a.append(value**2)
```

```
...:
```

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

```
In [4]: %timeit a = [value**2 for value in xrange(10)]
```

```
1000000 loops, best of 3: 1.46 us per loop
```

Function Info in IPython

HELP USING ?

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

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

```
Type:          builtin_function_or_method
```

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

```
Namespace:     Python builtin
```

```
Docstring:
```

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

Return the number of items of a sequence or mapping.

Function Info in IPython

SHOW SOURCE CODE USING ??

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

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

```
def squeeze(a):
    """Remove single-dimensional entries from the shape of a.
    Examples
    -----
    >>> x = array([[[[1,1,1],[2,2,2],[3,3,3]]]])
    >>> x.shape
    (1, 3, 3)
    >>> squeeze(x).shape
    (3, 3)
    """
    try:
        squeeze = a.squeeze
    except AttributeError:
        return _wrapit(a, 'squeeze')
    return squeeze()
```



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

Directory Navigation in IPython

Change directory (note Unix style forward slashes!)

In [8]: `cd c:/python_class/Demos/speed_of_light`

`c:\python_class\Demos\speed_of_light`



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

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

In [9]: `ls`

Volume in drive C has no label.

Volume Serial Number is 5417-593D

Directory of c:\python_class\Demos\speed_of_light

09/01/2008	02:53 PM	<DIR>	.
09/01/2008	02:53 PM	<DIR>	..
09/01/2008	02:48 PM	1,188	exercise_speed_of_light.txt
09/01/2008	02:48 PM	2,682,023	measurement_description.pdf
09/01/2008	02:48 PM	187,087	newcomb_experiment.pdf
09/01/2008	02:48 PM	1,312	newcomb_histogram.dat
09/01/2008	02:48 PM	1,436	newcomb_histogram.py
09/01/2008	02:48 PM	1,232	newcomb_histogram2.py
		6 File(s)	2,874,278 bytes
		2 Dir(s)	11,997,437,952 bytes free

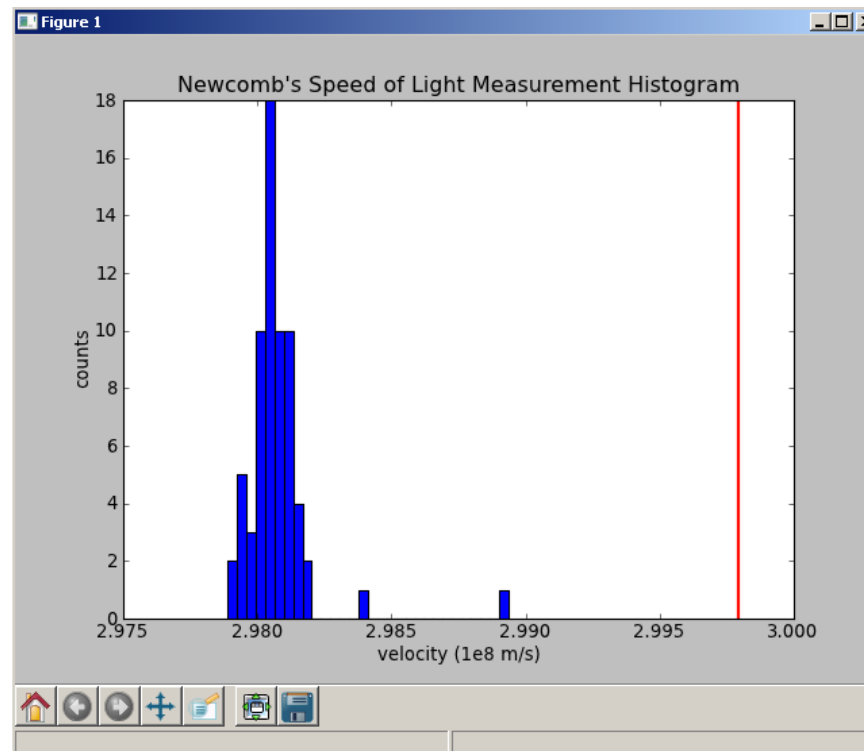
Running Scripts in IPython

tab completion

```
In [10]: %run newcomb_hi
newcomb_histogram.dat newcomb_histogram.py
```

execute a python file

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



HISTORY COMMAND

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

INPUT HISTORY

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

OUTPUT HISTORY

```
# grab previous result  
In [5]: _  
Out[5]: 'a\n'  
  
# grab result from prompt[2]  
In [6]: _2  
Out[6]: 1
```



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

Reading Simple Tracebacks

ERROR ADDING AN INTEGER TO A STRING

```
In [9]: 1 + "hello"
-----
TypeError: Traceback (most recent call last)
C:\...\<ipython-input...> in <module>()
----> 1 1 + "hello"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Location and code where error occurred.

The “type” of error that occurred.

Short message about why it occurred.

ERROR TRYING TO ADD A NON-EXISTENT VARIABLE

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

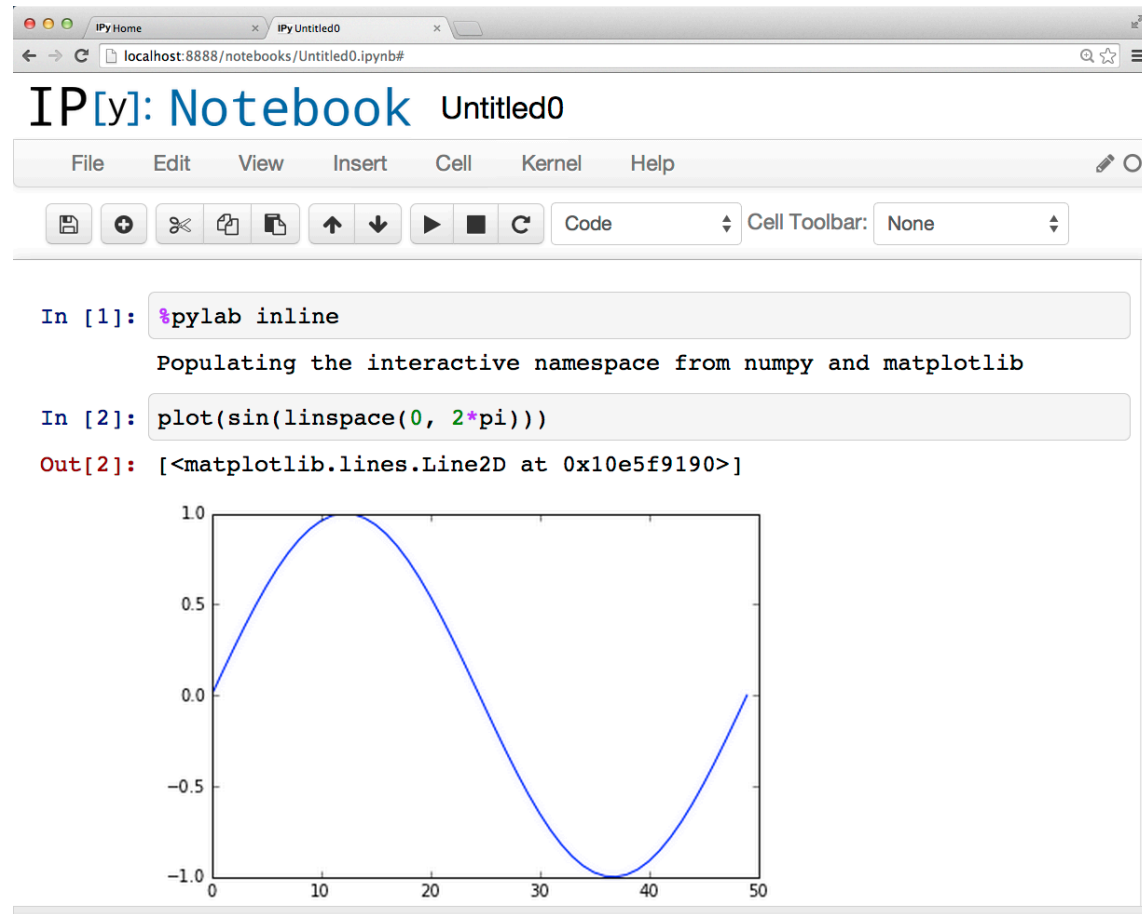
IPython Notebook: Share your results

One `.ipynb` file with:

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

Easy to share results:

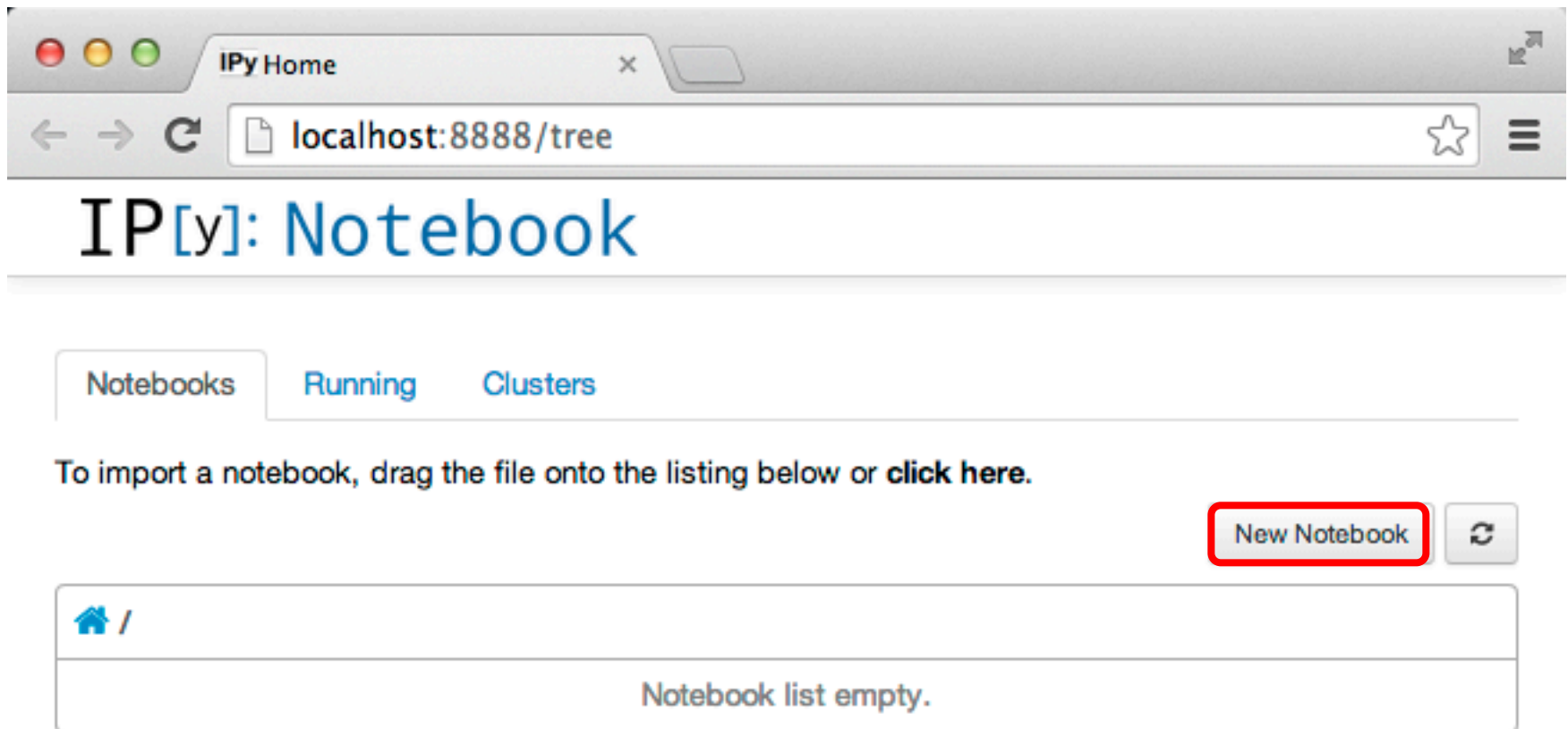
- results inline with code
- code is executable



Creating a notebook from terminal

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

```
$ ipython notebook
```



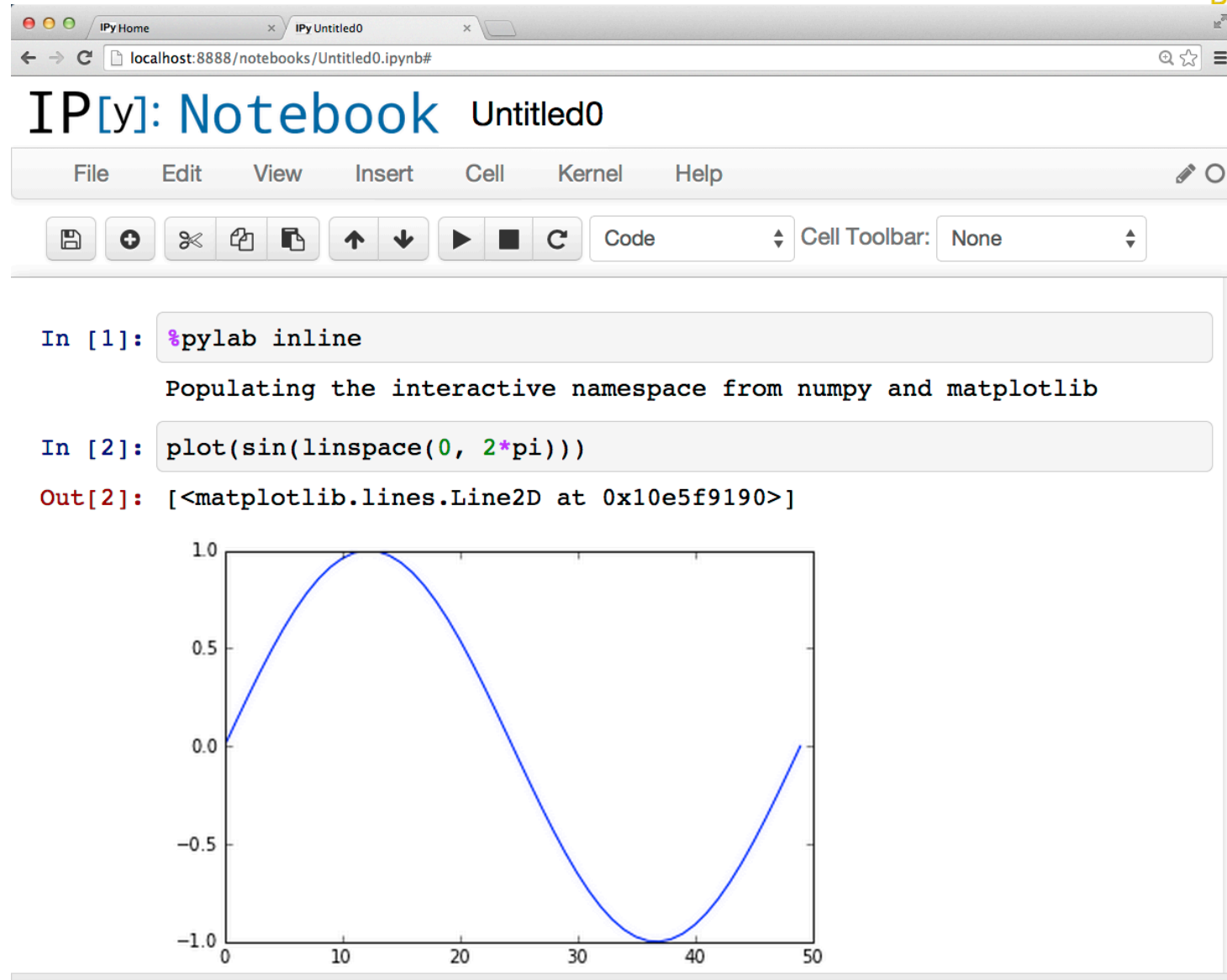
IPython notebook features

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

```
In [2]: from IPython.display import Latex  
        Latex("$\int_{a}^{b} f(x) \, dx$")
```

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

Inline figures in the notebook



Introduction to Python Software Craftsmanship

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

Structure and Interpretation of Computer Programs
Harold Abelson and Gerald Sussman

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

Hackers and Painters

Paul Graham

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

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

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

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

Python developers code in two modes:

Interactive Mode: Quick Iteration

Interactive Prompt and exploratory development.

Production Mode: Building for the Ages

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

Naming Variables

Goal: Read the code and
understand without having to think

Typical Scientific Naming Convention

STRAIGHT FROM FFTPACK

```
SUBROUTINE CFFTB1 (N,C,CH,WA,IFAC)
      DIMENSION      CH(*)          ,C(*)          ,WA(*)          ,IFAC(*)
      NF = IFAC(2)
      NA = 0
      L1 = 1
      IW = 1
      DO 116 K1=1,NF
        IP = IFAC(K1+2)
        L2 = IP*L1
        IDO = N/L2
        IDOT = IDO+IDO
        IDL1 = IDOT*L1
        IF (IP .NE. 4) GO TO 103
        IX2 = IW+IDOT
        IX3 = IX2+IDOT
        IF (NA .NE. 0) GO TO 101
        CALL PASSB4 (IDOT,L1,C,CH,WA(IW),WA(IX2),WA(IX3))
        GO TO 102
```

<and on and on for 368 lines...>

Primary Naming Consideration

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

POOR NAME CHOICES

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

DESCRIPTIVE NAME CHOICES

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

Using *extremely* short names

LOOP INDICES I, J, and K

```
# This is ok.
```

```
for i in xrange(10):  
    scores[i] = 0
```

```
# But this is better.
```

```
events = xrange(10)  
for event in events:  
    decathlon_scores[event] = 0
```

Using *extremely* short names

INDUSTRY STANDARD VARIABLES IN “SMALL” CONTEXT

Quick, what does each variable stand for?

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

Using *extremely* short names

INDUSTRY STANDARD VARIABLES IN “SMALL” CONTEXT

```
def sin_wave(t, a=1, w=2*pi, phi=0):  
    """  
    Return a sin wave form for time t.  
  
    Inputs  
    -----  
    t: time in seconds  
    a: amplitude scale factor  
    w: frequency in radians/second  
    phi: phase shift in radians  
  
    Returns  
    -----  
    y: sin wave output  
    """  
    y = a * sin(w*t + phi)  
    return y
```

Bad Code Comments

REPEATING THE CODE

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

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

INCORRECT COMMENTS

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

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

Better Code Comments

SUMMARY COMMENTS

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

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

FIXME COMMENTS

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

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

Comments in production Python code

PERCENTAGE OF PYTHON SOURCE LINES THAT ARE COMMENT LINES

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

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

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

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

Testing Code

Overarching Concept:

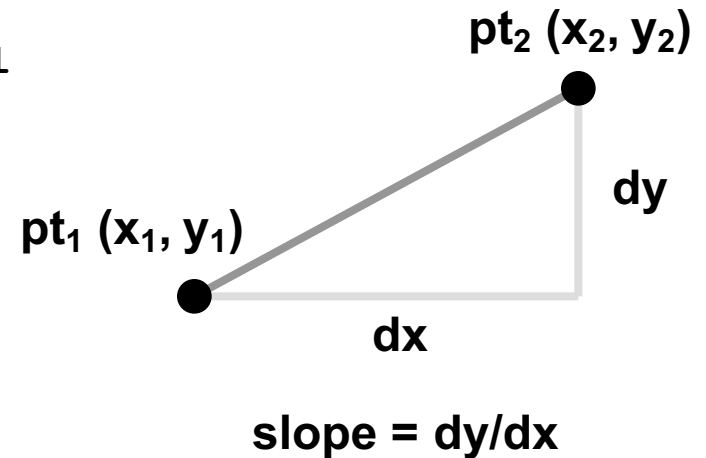
Write tests as you write your code

Test Driven Development

TEST

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

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



FANCY_MATH

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

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

All the features are tested.

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

Discovering tests: nosetests

FROM THE TERMINAL

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

Using unittest Framework

TESTS EMBEDDED IN CLASSES

```
# Test cases:
#   - each method is a unit test
#   - found by nosetests as well

from unittest import TestCase

from fancy_math import slope

class TestModule(TestCase):

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

Timing and Profiling Code

Ways to time execution

Timing inside the code (Good)

- the time module from std lib

Timing in ipython (Better):

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

Profiling the code (Best):

- cProfile or line_profiler package

Timing in Python

USE TIME PACKAGE

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

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

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

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

USE IPYTHON TOOLS

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

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

Profiling with cProfile

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

WORKFLOW

The `cProfile` workflow has two main steps:

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

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

Automatic method

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

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

```
5752950 function calls (3568633 primitive calls) in 4.022 seconds
```

```
Ordered by: internal time
```

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

time in *this* function only

time in this function
+ all called functions

Controlled method

EXAMPLE

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

Ordered by: standard name

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

Profiling with cProfile

OUTPUT TABLE COLUMNS

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

line_profiler and kernprof

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

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

INSTALLATION

```
$ easy_install line_profiler
```

TYPICAL WORKFLOW

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

```
$ kernprof.py -l script_to_profile.py
```
3. Run the `line_profiler` module to display the results. For example,

```
$ python -m line_profiler script.to_profile.py.lprof
```
4. Adjust your code, and repeat steps 2-4.
5. Remove the `@profile` decorators.

line_profiler example

http_search.py

```
import re

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

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

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



See demo/profiling directory
for code.

line_profiler example

Run kernprof.py and line_profiler

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

...

```
http://sphinx.pocoo.org/
```

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

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

Timer unit: 1e-06 s

File: http_search.py

Function: scan_for_http at line 6

Total time: 0.016079 s

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

line_profiler example

http_search2.py

```
import re

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

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

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

line_profiler example

RUN kernprof.py AND line_profiler ON THE MODIFIED FILE

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

...

Wrote profile results to http_search2.py.lprof

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

Timer unit: 1e-06 s

File: http_search2.py

Function: scan_for_http at line 6

Total time: 0.00911 s

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

`pdb`

The Python debugger

What is `pdb`?

`pdb` is part of the standard library.

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

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

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

Starting pdb

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

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

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

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

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

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

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

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

Start the debugger at the point of the last exception

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

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

pdb commands

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

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

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

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

Pop up or push down the execution stack.

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

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

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

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

(Pdb) **a(rgs)**

Print the args for the current function.

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

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

(Pdb) j(ump) lineno

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

(Pdb) p / pp [expression]

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

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

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

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

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

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

#print instance variables in self

alias ps pi self

IPython and pdb

ipython can call pdb automatically upon error

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

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

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

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

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

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

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

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


Example session

DEBUGGING FROM ORIGIN OF EXCEPTION

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

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

Print the contents of the list

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

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

In [4]:

Return to ipython

Example session

DEBUGGING MIDDLE.PY FROM THE START

```
>>> import pdb
>>> import middle
>>> pdb.runcall( middle.run )
> z:\projects\pgtraining\pdb\middle.py(32)run()
-> for i in range( 1, 11 ) :
(Pdb) s
> z:\projects\pgtraining\pdb\middle.py(33)run()
-> l = make_list( i )
(Pdb) n
> z:\projects\pgtraining\pdb\middle.py(34)run()
-> print "The middle item(s) in %s\n\tis/are %s\n" % (l, get_middle( l ))
(Pdb) s
--Call--
> z:\projects\pgtraining\pdb\middle.py(2)get_middle()
-> def get_middle( item_list ) :
(Pdb) s
...
> z:\projects\pgtraining\pdb\middle.py(11)get_middle()
-> return item_list[half]
(Pdb) item_list
['0']
(Pdb) half
2
```

We know make_list is ok, so skip over it with "next"

Continue to execute lines until we see something suspicious

Print the contents of the list

Print the value of the index ah-ha!

Other debugging tools

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

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

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

Introduction to Python

Data types


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

Interactive Calculator

```
# adding two values
>>> 1 + 1
2
# setting a variable
>>> a = 1
>>> a
1
# checking a variable's type
>>> type(a)
<type 'int'>
# an arbitrarily long integer
>>> a = 12345678901234567890
>>> a
12345678901234567890L
>>> type(a)
<type 'long'>
# Remove 'a' from the 'namespace'
>>> del a
>>> a
NameError: name 'a' is not
defined
```

```
# real numbers
>>> b = 1.4 + 2.3
>>> b
3.6999999999999997
# "prettier" version.
>>> print b
3.7
>>> type(b)
<type 'float'>
# complex numbers
>>> c = 2+1.5j
>>> c
(2+1.5j)
```

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

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

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

More Interactive Calculation

ARITHMETIC OPERATIONS

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

SIMPLE MATH FUNCTIONS

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

OVERWRITING FUNCTIONS

```
# don't do this
>>> max = 100

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

TYPE CONVERSION

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

IN-PLACE OPERATIONS

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



Logical expressions, bool data type

DATA SCIENCE

COMPARISON OPERATORS

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

bool DATA TYPE

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

and OPERATOR

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

or OPERATOR

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

not OPERATOR

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


CREATING STRINGS

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

STRING OPERATIONS

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

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

STRING LENGTH

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

SPLIT/JOIN STRINGS

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

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

Multi-line Strings

TRIPLE QUOTES

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

NEW LINE CHARACTER

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

A few string methods and functions

REPLACING TEXT

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

CONVERT TO UPPER CASE

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

REMOVE WHITESPACE

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

NUMBERS TO STRINGS

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

STRINGS TO NUMBERS

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

String Formatting

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

Replacement field format: `{<name>:<format_spec>}`

Optional Optional

If 'name' is an integer, it refers to the argument position.

```
>>> '{0} is greater than {1}'.format(100, 50)
'100 is greater than 50'
```

If 'name' is text, it refers to a keyword argument.

```
>>> '{last}, {first}'.format(first='Ellen', last='Ripley')
'Ripley, Ellen'
```

String Formatting – Format spec

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

Fixed point format (and a named keyword argument).

```
>>> print '[{x:5.0f}] [{x:5.1f}] [{x:5.2f}]'.format(x=12.3456)
[  12] [ 12.3] [12.35]
```

Alignment (and using a numbered positional argument).

```
>>> print '[{0:<10s}] [{0:>10s}] [{0:^10s}]'.format('PYTHON')
[PYTHON    ] [    PYTHON] [  PYTHON  ]
```

Alignment with fill character.

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

String Formatting – Format spec

DATA SCIENCE

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

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

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

ALIGNMENT OPTION

Char Meaning

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

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

SIGN OPTION

For numbers only.

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

STRING TYPE CODES

Type Meaning

- | Type | Meaning |
|------|--|
| s | String. This is the default, and may be omitted. |

INTEGER TYPE CODES

Type Meaning

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

FLOATING POINT TYPE CODES

Type Meaning

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

String Formatting with %

FORMAT OPERATOR %

the % operator formats values
to strings using C conventions.

```
>>> s = "some numbers:"
>>> x = 1.34
>>> y = 2
>>> t = "%s %f, %d" % (s,x,y)
>>> print t
some numbers: 1.340000, 2
```

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

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

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

CONVERSION CODES

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

CONVERSION FLAGS

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

List objects

LIST CREATION WITH BRACKETS

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

CONCATENATING LIST

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

REPEATING ELEMENTS IN LISTS

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

range(start, stop, step)

```
# the range function is helpful
# for creating a sequence
```

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

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

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


RETRIEVING AN ELEMENT

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

SETTING AN ELEMENT

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

OUT OF BOUNDS

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

NEGATIVE INDICES

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

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



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

More on list objects

LIST CONTAINING MULTIPLE TYPES

```
# list containing integer,  
# string, and another list  
>>> a = [10, 'eleven', [12, 13]]  
>>> a[1]  
'eleven'  
>>> a[2]  
[12, 13]
```

```
# use multiple indices to  
# retrieve elements from  
# nested lists  
>>> a[2][0]  
12
```

LENGTH OF A LIST

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

DELETING OBJECT FROM LIST

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

DOES THE LIST CONTAIN x ?

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

Common methods for lists

`some_list.append(x)`

Add the element `x` to the end of the list `some_list`.

`some_list.count(x)`

Count the number of times `x` occurs in the list.

`some_list.extend(sequence)`

Concatenate `sequence` onto this list.

`some_list.index(x)`

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

`some_list.insert(index, x)`

Insert `x` before the specified index.

`some_list.pop(index)`

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

`some_list.remove(x)`

Delete the first occurrence of `x` from the list.

`some_list.reverse()`

Reverse the order of elements in the list.

`some_list.sort(key)`

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

`var[lower:upper:step]`

Extracts a portion of a sequence by specifying a lower and upper bound. The lower-bound element is included, but the upper-bound element *is not included*. Mathematically: $[lower, upper)$. The step value specifies the stride between elements.

SLICING LISTS

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

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

OMITTING INDICES

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

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

Lists in action

```
>>> a = [10,21,23,11,24]

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

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

Mutable vs. Immutable

MUTABLE OBJECTS

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

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

IMMUTABLE OBJECTS

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

```
# try inserting values into  
# a string
```

```
>>> s = 'abcde'  
>>> s[1:3] = 'xy'
```

```
Traceback (innermost last):  
File "<interactive input>",line 1,in ?  
TypeError: object doesn't support  
slice assignment
```

```
# here's how to do it
```

```
>>> s = s[:1] + 'xy' + s[3:]  
>>> print s  
'axyde'
```

Tuple – Immutable Sequence

TUPLE CREATION

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

PARENTHESES ARE OPTIONAL

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

LENGTH-1 TUPLE

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



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

TUPLES ARE IMMUTABLE

```
# create a list
>>> a = range(10,15)
[10, 11, 12, 13, 14]

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

```
# try inserting a value
>>> b[3] = 23
```

TypeError: 'tuple' object doesn't
support item assignment

Tuple (un)packing

(UN)PACKING TUPLES

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

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

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

WHY IS IT USEFUL?

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

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

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


Dictionaries

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

DICTIONARY EXAMPLE

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

Accessing and deleting keys and values

DATA SCIENCE

ACCESS USING INDEX NOTATION

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

ACCESS WITH `get(key, default)`

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

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

REMOVE AN ENTRY WITH `DEL`

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

REMOVE WITH `pop(key, default)`

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

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

Dictionaries in action

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

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

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

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

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

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

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

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

Common methods for dictionaries

`some_dict.clear()`

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

`some_dict.copy()`

Create a copy of the dictionary

`x in some_dict`

Test whether the dictionary contains the key `x`.

`some_dict.keys()`

Return a list of all the keys in the dictionary.

`some_dict.values()`

Return a list of all the values in the dictionary.

`some_dict.items()`

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

Set objects

DEFINITION

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

CONSTRUCTION

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

ADD/REMOVE ELEMENTS

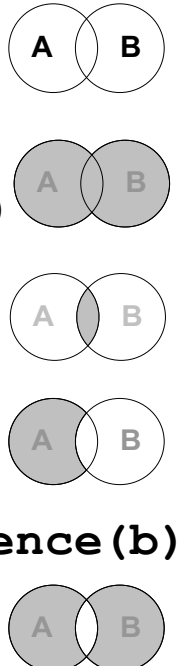
```
>>> t.add(5)
>>> t
set([1, 2, 3, 5])
>>> t.update([5,6,7])
>>> t
set([1, 2, 3, 5, 6, 7])
```

REMOVE ELEMENTS

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

SET OPERATIONS

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



The Venn diagrams show two overlapping circles, A and B. The union operation highlights both circles. The intersection operation highlights the overlapping region. The difference operation highlights the region of A that does not overlap with B. The symmetric difference operation highlights the regions of both A and B that do not overlap.

Selecting a data type

Selecting the appropriate data type is important

	insert	remove	find	ordered
list	linear	linear	linear	✓
set	constant	constant	constant	✗
dict	constant	constant	constant	✗

Typical usages for each data type:

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

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

Introduction to Python

Control statements

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

If statements

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

IF STATEMENT FORMAT

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

IF EXAMPLE

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

Test Values

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

EMPTY OBJECTS

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

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

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



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

While loops

while loops iterate until a condition is met

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

WHILE LOOP

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

BREAKING OUT OF A LOOP

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

For loops

for loops iterate over a sequence of objects

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

TYPICAL SCENARIO

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

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

LOOPING OVER A STRING

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

LOOPING OVER A SEQUENCE

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

List Comprehension

LIST TRANSFORM WITH LOOP

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

FILTER-TRANSFORM WITH LOOP

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

LIST COMPREHENSION

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

LIST COMPREHENSION WITH FILTER

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



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

Looping Patterns

MULTIPLE LOOP VARIABLES

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

ENUMERATE

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

ZIP

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

REVERSED

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

Looping over a dictionary

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

DEFAULT LOOPING (KEYS)

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

LOOPING OVER KEYS (EXPLICIT)

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

LOOPING OVER VALUES

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

LOOPING OVER ITEMS

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

Introduction to Python

Organizing code

- Functions
- Modules
- Packages

Functions are reusable snippets of code.

- Definition
- Positional and keyword arguments

Anatomy of a function

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

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

A colon (**:**) terminates the function signature.

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

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

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

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

Our new function in action

```
# We'll create our function
# on the fly in the
# interpreter.
```

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

```
# Test it out with numbers.
```

```
>>> val_1 = 2
>>> val_2 = 3
>>> add(val_1,val_2)
5
```

```
# How about strings?
```

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

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

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

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

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

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

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

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

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

Function Calling Conventions

POSITIONAL ARGUMENTS

```
# The "standard" calling
# convention we know and love.
>>> def add(x, y):
...     return x + y
```

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

KEYWORD ARGUMENTS

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

DEFAULT VALUES

```
# Arguments can be
# assigned default values.
>>> def quad(x,a=1,b=1,c=0):
...     return a*x**2 + b*x + c
```

```
# Use defaults for a, b and c.
>>> quad(2.0)
6.0
```

```
# Set b=3. Defaults for a & c.
>>> quad(2.0, b=3)
10.0
```

```
# Keyword arguments can be
# passed in out of order.
>>> quad(2.0, c=1, a=3, b=2)
17.0
```

Function Calling Conventions

VARIABLE NUMBER OF ARGS

```
# Pass in any number of  
# arguments. Extra arguments  
# are put in the tuple args.
```

```
>>> def foo(x, y, *args):  
...     print x, y, args
```

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

VARIABLE KEYWORD ARGS

```
# Extra keyword arguments  
# are put into the dict kw.
```

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

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

Function Calling Conventions

THE 'ANYTHING' SIGNATURE

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

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

MULTIPLE FUNCTION RETURNS

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

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

Expanding Function Arguments

POSITIONAL ARGUMENT EXPANSION

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

```
# '*' in a function call  
# converts a sequence into the  
# arguments to a function.
```

```
>>> vars = [1,2]
```

```
>>> add(*vars)
```

```
3
```

KEYWORD ARGUMENT EXPANSION

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

```
# '**' expands a  
# dictionary into keyword  
# arguments for a function.
```

```
vars = {'y':3, 'z':4}
```

```
>>> bar(1, **vars)
```

```
1, 3, {'z': 4}
```


Evolution of a script

Useful software often starts its life as a script.

EXPLORATORY SCRIPT

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

# Create a report about the results.
print important_number
...
```

Evolves to a function...

ONE MEGA-FUNCTION

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

Evolution Stops

And Stops...

There are some short term benefits to this.

MEGA-FUNCTION BENEFITS

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

Evolution to a library

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

LOW LEVEL FUNCTION LIBRARY

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

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

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

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

DRIVER FUNCTIONS

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

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

“One Idea Per Function” Benefits

DATA SCIENCE

Smaller, less complex snippets of code

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

Don't Repeat Yourself

DUPLICATED CODE

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

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

DON'T REPEAT YOURSELF

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

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

Modules

Modules and packages

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

Importing a module

BASIC IMPORTS

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

IMPORTING SPECIFIC SYMBOLS

```
# Select specific names to
# bring into the local
# namespace.
>>> from numpy import add, pi
>>> pi
3.141592653589793
>>> add(2, 3)
5
```

IMPORTING *EVERYTHING*

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

MODULES ARE .PY FILES

Modules are just .py files.

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

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

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

EX.PY

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

```
PI = 3.1416
```

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

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

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

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

Packages

PACKAGES

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

Example:

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

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

PACKAGES ARE DIRECTORIES

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

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

Setting up PYTHONPATH

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

WINDOWS

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

UNIX: .cshrc

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

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

UNIX: .bashrc

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

Naming Packages and Modules

MODULE NAMES

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

```
# Yes
```

```
foo_bar.py
```

```
# No
```

```
FooBar.py
```

PACKAGE NAMES

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

```
# Yes
```

```
packagename
```

```
# No
```

```
PackageName
```

```
package_name
```

Common Directory Structure

PACKAGE/MODULE/TESTS LAYOUT

Example directory structure for Python libraries.

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

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

Standard Modules

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

re - regular expressions

copy – shallow and deep copy operations

datetime - time and date objects

math, cmath - real and complex math

decimal, fractions - arbitrary precision
decimal and rational number objects

os, os.path, shutil - filesystem operations

sqlite3 - internal SQLite database

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

csv, netrc – file format handling

xml – various modules for handling XML

htmllib – an HTML parser

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

cmd – support for command interpreters

pdb – Python interactive debugger

profile, cProfile, timeit – Python profilers

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

mmap – memory-mapped files

threading, Queue – threading support

multiprocessing – process based ‘threading’

subprocess – executing external commands

pickle, cPickle – object serialization

struct – interpret bytes as packed binary data

urllib2 – open and read from URLs

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

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

Selections from the Python Standard Library

datetime – Dates and Times

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

DATE OBJECT

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

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

TIME OBJECT

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

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

datetime – Dates and Times

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

DATETIME OBJECT

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

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

DATETIME FORMAT STRING

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

sys module

```
>>> import sys
```

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

Command Line Arguments

`sys.argv`

List of command line arguments.

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

Example:

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

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

Exception Information

`sys.exc_info()`

Returns a tuple (type, value, traceback)

`sys.exc_clear()`

Clear all exception information.

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

Standard File Objects

`sys.stdin`

`sys.stdout`

`sys.stderr`

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

`sys.__stdin__`

`sys.__stdout__`

`sys.__stderr__`

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

Exit

`sys.exit(arg)`

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

Python's module search path

`sys.path`

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

A program is free to modify this list dynamically.

Platform Information

`sys.platform`

A string containing the platform identifier.

Windows: 'win32'

Mac OSX: 'darwin'

Linux: 'linux2'

`sys.getwindowsversion()`

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

```
>>> sys.platform
'win32'
```

```
>>> sys.getwindowsversion()
(5, 1, 2600, 2, 'Service Pack 3')
```

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

Python Version

`sys.version`

A string containing information about the Python version.

`sys.version_info`

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

```
>>> sys.version
'2.6.5 |EPD 6.2-2 (32-bit)|
(r265:79063, May  7 2010, 13:28:19)
[MSC v.1500 32 bit (Intel)]'
>>> sys.version_info
(2, 6, 5, 'final', 0)
```

os module

```
>>> import os
```

Path Operations

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

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

`os.chdir(path)`

Change the current working directory to the provided path.

`os.getcwd()`

Return the current working directory.

`os.listdir(path)`

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

Separation Constants

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

Line separator in text mode.

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

Path separator on file system.

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

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

Others

`os.environ`

Dictionary of all environment variables

`os.urandom(len)`

String of random bytes

`os.error`

Error object

os.path

os.path – tests

`os.path.isfile(path)`

Test whether a path is a regular file.

`os.path.isdir(path)`

Test whether a path refers to an existing directory.

`os.path.exists(path)`

Test whether a path exists.

`os.path.isabs(path)`

Test whether a path is absolute.

os.path – split and join

`os.path.split(path)`

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

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

Join two or more path components.

Others

`os.path.abspath(path)`

Return an absolute path.

`os.path.dirname(path)`

Return the directory component of a pathname.

`os.path.basename(path)`

Return the final component of a pathname.

`os.path.splitext(path)`

Split the extension from a pathname.

Returns (root, ext).

`os.path.expanduser(path)`

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