Introduction to Python for Science

Release 1

Chris Burns and Gaël Varoquaux

August 18, 2009

The workflow: IPython and a text editor

Interactive work to test and understand algorithm

Note: Reference document for this section:

IPython user manual: http://ipython.scipy.org/doc/manual/html/

1.1 Command line interaction

Start *ipython*:

```
In [1]: print('Hello world')
Hello world
```

Getting help:

```
In [2]: print?
Type:               builtin_function_or_method
Base Class:         <type 'builtin_function_or_method'>
String Form:        <built-in function print>
Namespace:          Python builtin
Docstring:
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
```

1.2 Elaboration of the algorithm in an editor

Edit *my_file.py*:

```
s = 'Hello world'
print(s)
```

Run it in ipython and explore the resulting variables:

```
In [3]: %run my_file.py
Hello word

In [4]: s
Out[4]: 'Hello word'

In [5]: %whos
Variable   Type    Data/Info
---------------------------
s          str     Hello word
```

Note: From a script to functions

- A script is not reusable, functions are.
- Important: break the process in small blocks.

Introduction to the Python language

Note: Reference document for this section:

Python tutorial: http://docs.python.org/tutorial/

2.1 Basic types

2.1.1 Numbers

- IPython as a calculator:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: 2**10
Out[2]: 1024

In [3]: (1 + 1j)*(1 - 1j)
Out[3]: (2+0j)
```

- scalar types: int, float, complex

```
In [4]: type(1)
Out[4]: <type 'int'>

In [5]: type(1.)
Out[5]: <type 'float'>

In [6]: type(1 + 0j)
Out[6]: <type 'complex'>
```

Warning: Integer division

```
In [7]: 3/2
Out[7]: 1

In [8]: from __future__ import division

In [9]: 3/2
Out[9]: 1.5
```

Trick: Use floats

```
In [10]: 3./2
Out[10]: 1.5
```

- Type conversion:

```
In [11]: float(1)
Out[11]: 1.
```

Exercise:

Compare two approximations of pi: 22/7 and 355/113
(pi = 3.14159265...)

2.1.2 Collections

Collections: list, dictionaries (and strings, tuples, sets, ...)

Lists

```
In [12]: l = [1, 2, 3, 4, 5]
```

- Indexing:

```
In [13]: l[2]
Out[13]: 3
```

Counting from the end:

```
In [14]: l[-1]
Out[14]: 5
```

- Slicing:

```
In [15]: l[3:]
Out[15]: [4, 5]

In [16]: l[:3]
Out[16]: [1, 2, 3]

In [17]: l[::2]
Out[17]: [1, 3, 5]

In [18]: l[-3:]
Out[18]: [3, 4, 5]
```

Syntax: *start:stop:stride*

- Operations on lists:

Reverse *l*:

```
In [19]: r = l[::-1]

In [20]: r
Out[20]: [5, 4, 3, 2, 1]
```

Append an item to *r*:

```
In [21]: r.append(3.5)

In [22]: r
Out[22]: [5, 4, 3, 2, 1, 3.5]
```

Extend a list with another list (in-place):

```
In [23]: l.extend([6, 7])

In [24]: l
Out[24]: [1, 2, 3, 4, 5, 6, 7]
```

Concatenate two lists:

```
In [25]: r + l
Out[25]: [5, 4, 3, 2, 1, 3.5, 1, 2, 3, 4, 5, 6, 7]
```

Sort *r*:

```
In [26]: r.sort()

In [27]: r
Out[27]: [1, 2, 3, 3.5, 4, 5]
```

Note: Methods:

r.sort: *sort* is a method of *r*: a special function to is applied to *r*.

Warning: Mutables:
Lists are mutable types: *r.sort* modifies in place *r*.

Note: Discovering methods:

In IPython: tab-completion (press tab)

```
In [28]: r.
r.__add__           r.__iadd__          r.__setattr__
r.__class__         r.__imul__          r.__setitem__
r.__contains__      r.__init__          r.__setslice__
r.__delattr__       r.__iter__          r.__sizeof__
r.__delitem__       r.__le__            r.__str__
r.__delslice__      r.__len__           r.__subclasshook__
r.__doc__           r.__lt__            r.append
r.__eq__            r.__mul__           r.count
r.__format__        r.__ne__            r.extend
r.__ge__            r.__new__           r.index
r.__getattribute__  r.__reduce__        r.insert
r.__getitem__       r.__reduce_ex__     r.pop
r.__getslice__      r.__repr__          r.remove
r.__gt__            r.__reversed__      r.reverse
r.__hash__          r.__rmul__          r.sort
```

Dictionaries

Dictionaries are a mapping between keys and values:

```
In [29]: d = {'a': 1, 'b':1.2, 'c':1j}

In [30]: d['b']
Out[30]: 1.2

In [31]: d['d'] = 'd'

In [32]: d
Out[32]: {'a': 1, 'b': 1.2, 'c': 1j, 'd': 'd'}

In [33]: d.keys()
Out[33]: ['a', 'c', 'b', 'd']

In [34]: d.values()
Out[34]: [1, 1j, 1.2, 'd']
```

Warning: Keys are not ordered

Note: Dictionnaries are an essential data structure

For instance to store precomputed values.

Strings

- Different string syntaxes:

```
a = 'Mine'
a = "Chris's"
a = '''Mine
       and not his'''
a = """Mine
       and Chris's"""
```

- Strings are collections too:

```
In [35]: s = 'Python is cool'

In [36]: s[-4:]
Out[36]: 'cool'
```

- And they have many useful methods:

```
In [37]: s.replace('cool', 'powerful')
Out[37]: 'Python is powerful'
```

Warning: Strings are not mutable

- String substitution:

```
In [38]: 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string')
Out[38]: 'An integer: 1; a float: 0.100000; another string: string'
```

More collection types

- **Sets:** non ordered, unique items:

```
In [39]: s = set(('a', 'b', 'c', 'a'))

In [40]: s
Out[40]: set(['a', 'b', 'c'])

In [41]: s.difference(('a', 'b'))
Out[41]: set(['c'])
```

Sets cannot be indexed:

```
In [42]: s[1]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

TypeError: 'set' object does not support indexing
```

- **Tuples:** non-mutable lists:

```
In [43]: t = 1, 2

In [44]: t
Out[44]: (1, 2)

In [45]: t[1]
Out[45]: 2

In [46]: t[1] = 2
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

TypeError: 'tuple' object does not support item assignment
```

2.2 Control Flow

Controls the order in which the code is executed.

2.2.1 if/else

```
In [1]: if 2**2 == 4:
   ...:     print('Totology')
   ...:
Totology
```

Blocks are delimited by indentation

```
In [2]: a = 10

In [3]: if a == 1:
   ...:     print(1)
   ...: elif a == 2:
   ...:     print(2)
   ...: else:
   ...:     print('A lot')
   ...:
A lot
```

2.2.2 for/range

Iterating with an index:

```
In [4]: for i in range(4):
   ...:     print(i)
   ...:
0
1
2
3
```

But most often, it is more readable to iterate over values:

```
In [5]: for word in ('cool', 'powerful', 'readable'):
   ...:     print('Python is %s' % word)
   ...:
Python is cool
Python is powerful
Python is readable
```

2.2.3 while/break/continue

Typical C-style while loop (Mandelbrot problem):

```
In [6]: z = 1 + 1j

In [7]: while abs(z) < 100:
   ...:     z = z**2 + 1
   ...:

In [8]: z
Out[8]: (-134+352j)
```

break out of enclosing for/while loop:

```
In [9]: z = 1 + 1j

In [10]: while abs(z) < 100:
   ....:     if z.imag == 0:
   ....:         break
   ....:     z = z**2 + 1
   ....:
   ....:
```

Rmk: *continue* the next iteration of a loop.

2.2.4 Conditional Expressions

- *if object*

  Evaluates to True: – any non-zero value

  – any sequence with a length > 0

  Evaluates to False: – any zero value

  – any empty sequence

- *a == b*

  Tests equality, with logics:

  ```
  In [19]: 1 == 1.
  Out[19]: True
  ```

- *a is b*

  Tests identity: both objects are the same

  ```
  In [20]: 1 is 1.
  Out[20]: False

  In [21]: a = 1

  In [22]: b = 1

  In [23]: a is b
  Out[23]: True
  ```

- *a in b*

  For any collection *b*: *b* contains *a*

If *b* is a dictionary, this tests that *a* is a key of *b*.

2.2.5 Advanced iteration

Iterate over any *sequence*

- You can iterate over any sequence (string, list, dictioary, file, ...)

```
In [11]: vowels = 'aeiouy'

In [12]: for i in 'powerful':
   ....:     if i in vowels:
   ....:         print(i),
   ....:
   ....:
o e u
```

Warning: Not safe to modify the sequence you are iterating over.

Keeping track of enumeration number

Common task is to iterate over a sequence while keeping track of the item number.

- Could use while loop with a counter as above. Or a for loop:

```
In [13]: for i in range(0, len(words)):
   ....:     print(i, words[i])
   ....:
   ....:
0 cool
1 powerful
2 readable
```

- But Python provides **enumerate** for this:

```
In [14]: for index, item in enumerate(words):
   ....:     print(index, item)
   ....:
   ....:
0 cool
1 powerful
2 readable
```

Looping over a dictionary

Use **iteritems**:

```
In [15]: d = {'a': 1, 'b':1.2, 'c':1j}

In [15]: for key, val in d.iteritems():
   ....:     print('Key: %s has value: %s' % (key, val))
   ....:
Key: a has value: 1
Key: c has value: 1j
Key: b has value: 1.2
```

2.2.6 List Comprehensions

Note: List comprehension

```
In [16]: [i**2 for i in range(4)]
Out[16]: [0, 1, 4, 9]
```

Exercise

Compute the decimals of Pi using the Wallis formula:

$$\pi = 2 \prod_{1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

2.3 Defining functions

2.3.1 Function definition

```
In [56]: def foo():
   ....:     print('in foo function')
   ....:
   ....:

In [57]: foo()
in foo function
```

2.3.2 Return statement

Functions can *optionally* return values.

```
In [6]: def area(radius):
   ...:     return 3.14 * radius * radius
   ...:

In [8]: area(1.5)
Out[8]: 7.0649999999999995
```

Note: By default, functions return `None`.

2.3.3 Parameters

Mandatory parameters (positional arguments)

```
In [81]: def double_it(x):
   ....:     return x * 2
   ....:

In [82]: double_it(3)
Out[82]: 6

In [83]: double_it()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/<ipython console> in <module>()

TypeError: double_it() takes exactly 1 argument (0 given)
```

Optional parameters (keyword or named arguments)

```
In [84]: def double_it(x=2):
   ....:     return x * 2
   ....:

In [85]: double_it()
Out[85]: 4

In [86]: double_it(3)
Out[86]: 6
```

Keyword arguments allow you to specify *default values*.

Warning: Default values are evaluated when the function is defined, not when it is called.

```
In [124]: bigx = 10

In [125]: def double_it(x=bigx):
   .....:     return x * 2
   .....:

In [126]: bigx = 1e9  # No big

In [128]: double_it()
Out[128]: 20
```

More involved example implementing python's slicing:

```
In [98]: def slicer(seq, start=None, stop=None, step=None):
   ....:     """Implement basic python slicing."""
   ....:     return seq[start:stop:step]
   ....:

In [101]: seuss = 'one fish, two fish, red fish, blue fish'.split()

In [102]: seuss
Out[102]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [103]: slicer(seuss)
Out[103]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [104]: slicer(seuss, step=2)
Out[104]: ['one', 'two', 'red', 'blue']

In [105]: slicer(seuss, 1, step=2)
Out[105]: ['fish,', 'fish,', 'fish,', 'fish']

In [106]: slicer(seuss, start=1, stop=4, step=2)
Out[106]: ['fish,', 'fish,']
```

2.3.4 Passed by value

Parameters to functions are passed by value.

When you pass a variable to a function, python passes the object to which the variable refers (the **value**). Not the variable itself.

If the **value** is immutable, the function does not modify the caller's variable. If the **value** is mutable, the function modifies the caller's variable.

```
In [1]: def foo(x, y):
   ...:     x = 23
   ...:     y.append(42)
   ...:     print('x is %d' % x)
   ...:     print('y is %d' % y)
   ...:

In [2]: a = 77    # immutable variable

In [3]: b = [99]  # mutable variable

In [4]: foo(a, b)
x is 23
y is [99, 42]

In [5]: print a
77

In [6]: print b    # mutable variable 'b' was modified
[99, 42]
```

Functions have a local variable table. Called a *local namespace*.

The variable `x` only exists within the function *foo*.

2.3.5 Global variables

Variables declared outside the function can be referenced within the function:

```
In [114]: x = 5
```

```
In [115]: def addx(y):
   .....:     return x + y
   .....:

In [116]: addx(10)
Out[116]: 15
```

But these "global" variables cannot be modified within the function, unless declared **global** in the function.

This doesn't work:

```
In [117]: def setx(y):
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:

In [118]: setx(10)
x is 10

In [120]: x
Out[120]: 5
```

This works:

```
In [121]: def setx(y):
   .....:     global x
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:

In [122]: setx(10)
x is 10

In [123]: x
Out[123]: 10
```

2.3.6 Variable number of parameters

Special forms of parameters:

- *args: any number of positional arguments packed into a tuple
- **kwargs: any number of keyword arguments packed into a dictionary

```
In [35]: def variable_args(*args, **kwargs):
   ....:     print 'args is', args
   ....:     print 'kwargs is', kwargs
   ....:

In [36]: variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'y': 2, 'x': 1, 'z': 3}
```

2.3.7 Docstrings

Documention about what the function does and it's parameters. General convention:

```
In [67]: def funcname(params):
   ....:     """Concise one-line sentence describing the function.
   ....:
   ....:     Extended summary which can contain multiple paragraphs.
   ....:     """
   ....:     # function body
   ....:     pass
   ....:

In [68]: funcname?
Type:               function
Base Class: <type 'function'>
String Form:        <function funcname at 0xeaa0f0>
Namespace:  Interactive
File:               /Users/cburns/src/scipy2009/.../<ipython console>
Definition: funcname(params)
Docstring:
    Concise one-line sentence describing the function.

    Extended summary which can contain multiple paragraphs.
```

2.3.8 Functions are objects

Functions are first-class objects, which means they can be:

- assigned to a variable
- an item in a list (or any collection)
- passed as an argument to another function.

```
In [38]: va = variable_args

In [39]: va('three', x=1, y=2)
args is ('three',)
kwargs is {'y': 2, 'x': 1}
```

2.3.9 Methods

Methods are functions attached to objects. You've seen these in our examples on **lists**, **dictionaries**, **strings**, etc...

Exercise

Implement the quicksort algorithm, as defined by wikipedia:

```
function quicksort(array)
    var list less, greater
    if length(array)  1
        return array
    select and remove a pivot value pivot from array
    for each x in array
        if x  pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

2.4 Exceptions handling in Python

2.4.1 Exceptions

Exceptions are raised by errors in Python:

```
In [1]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError: integer division or modulo by zero

In [2]: 1 + 'e'
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for +: 'int' and 'str'

In [3]: d = {1:1, 2:2}

In [4]: d[3]
---------------------------------------------------------------------------
KeyError: 3

In [5]: l = [1, 2, 3]

In [6]: l[4]
---------------------------------------------------------------------------
IndexError: list index out of range

In [7]: l.foobar
---------------------------------------------------------------------------
AttributeError: 'list' object has no attribute 'foobar'
```

Different types of exceptions for different errors.

2.4.2 Catching exceptions

try/except

```
In [8]: while True:
   ....:     try:
   ....:         x = int(raw_input('Please enter a number: '))
   ....:         break
   ....:     except ValueError:
   ....:         print('That was no valid number.  Try again...')
   ....:
   ....:
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1

In [9]: x
Out[9]: 1
```

try/finally

```
In [10]: try:
   ....:     x = int(raw_input('Please enter a number: '))
   ....: finally:
   ....:     print('Thank you for your input')
   ....:
   ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError: invalid literal for int() with base 10: 'a'
```

Important for resource management (e.g. closing a file)

Easier to ask for forgiveness than for permission

Don't enforce contracts before hand.

```
In [11]: def print_sorted(collection):
   ....:     try:
   ....:         collection.sort()
   ....:     except AttributeError:
   ....:         pass
   ....:     print(collection)
   ....:
   ....:

In [12]: print_sorted([1, 3, 2])
[1, 2, 3]

In [13]: print_sorted(set((1, 3, 2)))
set([1, 2, 3])

In [14]: print_sorted('132')
132
```

2.4.3 Raising exceptions

- Capturing and reraising an exception:

```
In [15]: def filter_name(name):
   ....:     try:
   ....:         name = name.encode('ascii')
   ....:     except UnicodeError, e:
   ....:         if name == 'Gaël':
   ....:             print('OK, Gaël')
   ....:         else:
   ....:             raise e
   ....:     return name
   ....:

In [16]: filter_name('Gaël')
OK, Gaël
Out[16]: 'Ga\xc3\xabl'

In [17]: filter_name('Stéfan')
---------------------------------------------------------------------------
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 2: ordinal not in range(128
```

- Exceptions to pass messages between parts of the code:

```
In [17]: def achilles_arrow(x):
   ....:     if abs(x - 1) < 1e-3:
   ....:         raise StopIteration
   ....:     x = 1 - (1-x)/2.
   ....:     return x
   ....:

In [18]: x = 0

In [19]: while True:
   ....:     try:
   ....:         x = achilles_arrow(x)
   ....:     except StopIteration:
   ....:         break
   ....:
   ....:

In [20]: x
Out[20]: 0.9990234375
```

Use exceptions to notify certain conditions are met (e.g. StopIteration) or not (e.g. custom error raising)

> **Warning:** Capturing and not raising exception can lead to difficult debuging.

2.5 Reusing code

2.5.1 Importing objects

```
In [1]: import os

In [2]: os
Out[2]: <module 'os' from '/usr/lib/python2.6/os.pyc'>

In [3]: os.listdir('.')
Out[3]:
['conf.py',
 'basic_types.rst',
 'control_flow.rst',
 'functions.rst',
 'python_language.rst',
 'reusing.rst',
 'file_io.rst',
 'exceptions.rst',
 'workflow.rst',
 'index.rst']
```

And also:

```
In [4]: from os import listdir
```

Importing shorthands:

```
In [5]: import numpy as np
```

> **Warning:**
>
> ```
> from os import *
> ```
>
> **Do not do it.**
>
> - Makes the code harder to read and understand: where do symbols come from?
> - Makes it impossible to guess the functionality by the context and the name (hint: *os.name* is the name of the OS), and to profit usefully from tab completion.
> - Restricts the variable names you can use: *os.name* might override *name*, or vise-versa.
> - Creates possible name clashes between modules.
> - Makes the code impossible to statically check for undefined symbols.

A whole set of new functionnality!

```
In [6]: from __future__ import braces
```

2.5.2 Creating modules

File *demo.py*:

```
" A demo module. "

def print_b():
    " Prints b "
    print('b')

def print_a():
    " Prints a "
    print('a')
```

Importing it in IPython:

```
In [6]: import demo

In [7]: demo?
Type:               module
Base Class: <type 'module'>
String Form:        <module 'demo' from 'demo.py'>
Namespace:  Interactive
File:               /home/varoquau/Projects/Python_talks/scipy_2009_tutorial/source/demo.py
Docstring:
    A demo module.


In [8]: demo.print_a()
a

In [9]: demo.print_b()
b
```

Warning: Module caching
Modules are cached: if you modify *demo.py* and re-import it in the old session, you will get the old one.
Solution:

```
In [10]: reload(demo)
```

2.5.3 '__main__' and module loading

File *demo2.py*:

```
def print_a():
    " Prints a "
    print('a')

print('b')

if __name__ == '__main__':
    print_a()
```

Importing it:

```
In [11]: import demo2
b

In [12]: import demo2
```

Running it:

```
In [13]: %run demo2
b
a
```

2.5.4 Standalone scripts

- Running a script from the command line:

```
$ python demo2.py
b
a
```

- On Unix, make the file executable:
  - 'chmod uog+x demo2.py'
  - add at the top of the file:

```
#!/usr/bin/env python
```

- Command line arguments:

```
import sys
print sys.argv
```

```
$ python file.py test arguments
['file.py', 'test', 'arguments']
```

Note: Don't implement option parsing yourself. Use modules such as *optparse*.

Exercise

Implement a script that takes a directory name as argument, and returns the list of '.py' files, sorted by name length.
Hint: try to understand the docstring of list.sort

2.6 File I/O in Python

2.6.1 Reading from a file

Open a file with the `open` function:

```
In [67]: fp = open("holy_grail.txt")

In [68]: fp
Out[68]: <open file 'holy_grail.txt', mode 'r' at 0xealec0>

In [69]: fp.
fp.__class__          fp.__new__          fp.fileno           fp.readline
fp.__delattr__        fp.__reduce__       fp.flush            fp.readlines
fp.__doc__            fp.__reduce_ex__    fp.isatty           fp.seek
fp.__enter__          fp.__repr__         fp.mode             fp.softspace
fp.__exit__           fp.__setattr__      fp.name             fp.tell
fp.__getattribute__   fp.__str__          fp.newlines         fp.truncate
fp.__hash__           fp.close            fp.next             fp.write
```

```
fp.__init__          fp.closed            fp.read              fp.writelines
fp.__iter__          fp.encoding          fp.readinto          fp.xreadlines
```

Close a file with the `close` method:

```
In [73]: fp.close()

In [74]: fp.closed
Out[74]: True
```

Can read one line at a time:

```
In [69]: first_line = fp.readline()

In [70]: first_line
Out[70]: "GUARD: 'Allo, daffy English kaniggets and Monsieur Arthur-King, who is\n"
```

Or we can read the entire file into a list:

```
In [75]: fp = open("holy_grail.txt")

In [76]: all_lines = fp.readlines()

In [77]: all_lines
Out[77]:
["GUARD: 'Allo, daffy English kaniggets and Monsieur Arthur-King, who is\n",
 '      afraid of a duck, you know!  So, we French fellows out-wit you a\n',
 '      second time!\n',
 ' \n',
...
 ' \n']

In [78]: all_lines[0]
Out[78]: "GUARD: 'Allo, daffy English kaniggets and Monsieur Arthur-King, who is\n"
```

2.6.2 Iterate over a file

Files are sequences, we can iterate over them:

```
In [81]: fp = open("holy_grail.txt")

In [82]: for line in fp:
   ....:     print line
   ....:
GUARD: 'Allo, daffy English kaniggets and Monsieur Arthur-King, who is

      afraid of a duck, you know!  So, we French fellows out-wit you a

      second time!
```

2.6.3 File modes

- Read-only: `r`
- Write-only: `w`
  - Note: Create a new file or *overwrite* existing file.
- Append a file: `a`
- Read and Write: `r+`
- Binary mode: `b`
  - Note: Use for binary files, especially on Windows.

2.6.4 Writing to a file

Use the `write` method:

```
In [83]: fp = open('newfile.txt', 'w')

In [84]: fp.write("I am not a tiny-brained wiper of other people's bottoms!")

In [85]: fp.close()

In [86]: fp = open('newfile.txt')

In [87]: fp.read()
Out[87]: "I am not a tiny-brained wiper of other people's bottoms!"
```

Update a file:

```
In [104]: fp = open('newfile.txt', 'r+')

In [105]: line = fp.read()

In [111]: line = "CHRIS: " + line + "\n"

In [112]: line
Out[112]: "CHRIS: I am not a tiny-brained wiper of other people's bottoms!\n"

In [113]: fp.seek(0)

In [114]: fp.write(line)

In [115]: fp.tell()
Out[115]: 64L

In [116]: fp.seek(0)

In [117]: fp.read()
Out[117]: "CHRIS: I am not a tiny-brained wiper of other people's bottoms!"

In [132]: fp.write("GAEL: I've met your children dear sir, yes you are!\n")

In [136]: fp.seek(0)

In [137]: fp.readlines()
Out[137]:
["CHRIS: I am not a tiny-brained wiper of other people's bottoms!\n",
 "GAEL: I've met your children dear sir, yes you are!\n"]
```

2.6.5 File processing

Often want to open the file, grab the data, then close the file:

```
In [54]: fp = open("holy_grail.txt")

In [60]: try:
   ....:     for line in fp:
   ....:         print line
   ....: finally:
   ....:     fp.close()
   ....:
GUARD: 'Allo, daffy English kaniggets and Monsieur Arthur-King, who is

      afraid of a duck, you know!  So, we French fellows out-wit you a

      second time!
```

With Python 2.5 use the `with` statement:

```
In [65]: from __future__ import with_statement

In [66]: with open('holy_grail.txt') as fp:
   ....:     for line in fp:
   ....:         print line
   ....:
GUARD: 'Allo, daffy English kaniggets and Monsieur Arthur-King, who is

      afraid of a duck, you know!  So, we French fellows out-wit you a

      second time!
```

This has the advantage that it closed the file properly, even if an exception is raised, and is more concise than the `try-finally`.

Note: The `from __future__` line isn't required in Python 2.6

Exercise

Write a function that will load the column of numbers in `data.txt` and calculate the min, max and sum values.

2.7 Standard Library

The Python Standard Library: http://docs.python.org/library/index.html

2.7.1 `sys` module

System specific information related to the Python interpreter.

Which version of python are you running and where is it installed:

```
In [117]: sys.platform
Out[117]: 'darwin'

In [118]: sys.version
Out[118]: '2.5.2 (r252:60911, Feb 22 2008, 07:57:53) \n
          [GCC 4.0.1 (Apple Computer, Inc. build 5363)]'

In [119]: sys.prefix
Out[119]: '/Library/Frameworks/Python.framework/Versions/2.5'
```

List of command line arguments passed to a Python script:

```
In [100]: sys.argv
Out[100]: ['/Users/cburns/local/bin/ipython']
```

`sys.path` is a list of strings that specifies the search path for modules. Initialized from PYTHONPATH:

```
In [121]: sys.path
Out[121]:
['',
 '/Users/cburns/local/bin',
 '/Users/cburns/local/lib/python2.5/site-packages/grin-1.1-py2.5.egg',
 '/Users/cburns/local/lib/python2.5/site-packages/argparse-0.8.0-py2.5.egg',
 '/Users/cburns/local/lib/python2.5/site-packages/urwid-0.9.7.1-py2.5.egg',
 '/Users/cburns/local/lib/python2.5/site-packages/yolk-0.4.1-py2.5.egg',
 '/Users/cburns/local/lib/python2.5/site-packages/virtualenv-1.2-py2.5.egg',
 ...
```

2.7.2 `os` module

"A portable way of using operating system dependent functionality."

Environment variables:

```
In [9]: import os

In [11]: os.environ.keys()
Out[11]:
['_',
 'FSLDIR',
 'TERM_PROGRAM_VERSION',
 'FSLREMOTECALL',
 'USER',
 'HOME',
 'PATH',
 'PS1',
 'SHELL',
 'EDITOR',
 'WORKON_HOME',
 'PYTHONPATH',
 ...

In [12]: os.environ['PYTHONPATH']
Out[12]: '.:/Users/cburns/src/utils:/Users/cburns/src/nitools:
/Users/cburns/local/lib/python2.5/site-packages/:
/usr/local/lib/python2.5/site-packages/:
```

```
/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5'

In [16]: os.getenv('PYTHONPATH')
Out[16]: '.:/Users/cburns/src/utils:/Users/cburns/src/nitools:
/Users/cburns/local/lib/python2.5/site-packages/:
/usr/local/lib/python2.5/site-packages/:
/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5'
```

Directory and file manipulation

Current directory:

```
In [17]: os.getcwd()
Out[17]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
```

List a directory:

```
In [31]: os.listdir(os.curdir)
Out[31]:
['.index.rst.swo',
 '.python_language.rst.swp',
 '.view_array.py.swp',
 '_static',
 '_templates',
 'basic_types.rst',
 'conf.py',
 'control_flow.rst',
 'debugging.rst',
 ...
```

Make a directory:

```
In [32]: os.mkdir('junkdir')

In [33]: 'junkdir' in os.listdir(os.curdir)
Out[33]: True
```

Rename the directory:

```
In [36]: os.rename('junkdir', 'foodir')

In [37]: 'junkdir' in os.listdir(os.curdir)
Out[37]: False

In [38]: 'foodir' in os.listdir(os.curdir)
Out[38]: True

In [41]: os.rmdir('foodir')

In [42]: 'foodir' in os.listdir(os.curdir)
Out[42]: False
```

Delete a file:

```
In [44]: fp = open('junk.txt', 'w')

In [45]: fp.close()

In [46]: 'junk.txt' in os.listdir(os.curdir)
Out[46]: True

In [47]: os.remove('junk.txt')

In [48]: 'junk.txt' in os.listdir(os.curdir)
Out[48]: False
```

Path manipulations

`os.path` provides common operations on pathnames.

```
In [70]: fp = open('junk.txt', 'w')

In [71]: fp.close()

In [72]: a = os.path.abspath('junk.txt')

In [73]: a
Out[73]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/junk.txt'

In [74]: os.path.split(a)
Out[74]: ('/Users/cburns/src/scipy2009/scipy_2009_tutorial/source',
          'junk.txt')

In [78]: os.path.dirname(a)
Out[78]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'

In [79]: os.path.basename(a)
Out[79]: 'junk.txt'

In [80]: os.path.splitext(os.path.basename(a))
Out[80]: ('junk', '.txt')

In [84]: os.path.exists('junk.txt')
Out[84]: True

In [86]: os.path.isfile('junk.txt')
Out[86]: True

In [87]: os.path.isdir('junk.txt')
Out[87]: False

In [88]: os.path.expanduser('~/local')
Out[88]: '/Users/cburns/local'

In [92]: os.path.join(os.path.expanduser('~'), 'local', 'bin')
Out[92]: '/Users/cburns/local/bin'
```

Walking a directory

`os.path.walk` generates a list of filenames in a directory tree.

```
In [10]: for dirpath, dirnames, filenames in os.walk(os.curdir):
   ....:     for fp in filenames:
   ....:         print os.path.abspath(fp)
   ....:
   ....:
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/.index.rst.swo
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/.view_array.py.swp
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/basic_types.rst
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/conf.py
/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/control_flow.rst
...
```

2.7.3 Pattern matching on files

The `glob` module provides convenient file pattern matching.

Find all files ending in `.txt`:

```
In [18]: import glob

In [19]: glob.glob('*.txt')
Out[19]: ['holy_grail.txt', 'junk.txt', 'newfile.txt']
```

Exercise

Write a program to search your PYTHONPATH for the module `site.py`.

2.8 Timing and Profiling

2.8.1 Timing your code

Note: The timeit module: http://docs.python.org/library/timeit.html

Use `timeit` to measure the execution time of code.

```
In [98]: %timeit [x+3 for x in range(10)]
100000 loops, best of 3: 3.91 us per loop
```

You can specify the number of times to execute the statement in a loop:

```
In [99]: %timeit -n 10 [x+3 for x in range(10)]
10 loops, best of 3: 8.82 us per loop
```

Compare the execution time of different functions:

```
In [103]: def slow(x):
   .....:     result = []
   .....:     for item in x:
   .....:         result.insert(0, item)
   .....:     return result
   .....:

In [104]: def fast(x):
   .....:     result = []
   .....:     for item in x:
   .....:         result.append(item)
   .....:     result.reverse()
   .....:     return result
   .....:

In [105]: %timeit slow(range(100))
10000 loops, best of 3: 64.6 us per loop

In [106]: %timeit fast(range(100))
10000 loops, best of 3: 34.1 us per loop
```

2.8.2 Profiling your code

The profile module: http://docs.python.org/library/profile.html

```
In [4]: import cProfile

In [5]: cProfile.runctx('slow(range(100))', globals(), locals())
         104 function calls in 0.000 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 tmp.py:128(slow)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
      100    0.000    0.000    0.000    0.000 {method 'insert' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {range}
```