

# Python 123 introduction

Jeremy Sanders

October 2011

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>The prompt</b>	<b>2</b>
<b>3</b>	<b>Writing scripts</b>	<b>2</b>
<b>4</b>	<b>Basic data types</b>	<b>3</b>
4.1	Numbers . . . . .	3
4.2	Strings . . . . .	3
4.3	Lists . . . . .	4
4.4	Tuples . . . . .	5
4.5	Dictionaries . . . . .	6
4.6	File objects . . . . .	6
<b>5</b>	<b>Subtleties</b>	<b>7</b>
<b>6</b>	<b>Controlling program flow</b>	<b>8</b>
6.1	Subroutines, procedures and functions . . . . .	8
6.2	Modules . . . . .	9
6.3	If statements . . . . .	9
6.4	For loops . . . . .	10
6.5	While loops . . . . .	12
6.6	Exceptions . . . . .	12
6.7	Objects . . . . .	13
<b>7</b>	<b>Standard modules</b>	<b>15</b>
7.1	math . . . . .	15
7.2	cmath . . . . .	15
7.3	sys . . . . .	15
7.4	os . . . . .	16
7.5	random . . . . .	16
7.6	glob . . . . .	16
7.7	subprocess . . . . .	16
7.8	re . . . . .	17
7.9	urllib . . . . .	17
<b>8</b>	<b>Other modules</b>	<b>18</b>
8.1	numpy . . . . .	18
8.2	scipy . . . . .	18
8.3	pyfits . . . . .	18
<b>A</b>	<b>Enabling Python mode in emacs</b>	<b>19</b>

<b>B Complete class example</b>	<b>20</b>
<b>C Python example questions</b>	<b>24</b>

## 1 Introduction

Python is a simple, flexible high-level computer language. It can be used to automate many tasks, do computation which isn't time sensitive, and make your life much easier! I highly recommend learning it early in your PhD. It also runs on a variety of computer systems, and is freely available, so you can run it on your own laptops or home computers.

You can also simply call Fortran, C or C++ programs for it, so that you can write time critical bits of a program in one of these languages and the rest in Python. It also has a lot of useful libraries (see Section 8) to handle numerics (numpy) and scientific calculations (e.g. integration in *scipy*), FITS files (*pyfits*). There is even a version of IRAF supporting Python (*pyraf*).

This is a brief introduction to Python, but there are books in the library and a lot of documentation online. Start looking at <http://www.python.org/doc/>.

## 2 The prompt

Python can be used interactively:

```
jss@xport10:~/text> /usr/local/bin/python
Python 2.4.3 (#2, Oct 6 2006, 07:52:30)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 10+10
20
>>> 'hello '+'there'
'hello there'
>>> a=10
>>> b=20
>>> a+b           # the value is printed at the prompt
30
>>> print a+b     # you need to print to print inside a script
30
>>> import math
>>> print math.sin(math.pi/4.0)
0.707106781187
>>> help(math.sin) # gets help on a function from documentation
```

Press `ctrl+d` to exit. `ipython` gives you a nicer environment than the standard python prompt, and allows you to run unix commands.

## 3 Writing scripts

You can write python scripts, which are written in a similar way to shell scripts. Use `emacs` to create `test.py` containing:

```
#!/usr/local/bin/python

# this is a comment...
print "What is your name?"
name=raw_input()
```

```
print "Hello,", name
```

To make it executable type `chmod +x test.py`, then run it by typing its name, `test.py` on the unix prompt. Scripts can be placed in `/home/username/bin/` to get added to the path to become commands that can be used anywhere.

## 4 Basic data types

### 4.1 Numbers

Numbers can be integers (`int`, whole numbers) or floating point (`float`). The usual operators apply, `+`, `-`, `*` and `/`. `**` means power, `%` means remainder. Brackets indicate the order of evaluation.

```
>>> 0 # integer
0
>>> 0. # float
0.0
>>> 1+2 # add integers
3
>>> 3.14*200 # multiply floats
628.0
>>> a=3.5
>>> a**2 # square
12.25
>>> b=(a**2)/4
>>> b
3.0625
>>> 10 % 3 # remainder of 10/3 (integers only)
1
>>> 10 / 3 # dividing integers by integers gives whole numbers
3
>>> 10. / 3 # dividing integer or float by float gives floats
3.3333333333333335
>>> 3e4 / 1000 # scientific notation 1.234e-10 can be used for floats
30.0
>>> int(3.141592) # convert float to int
3
>>> float(3) # convert int to float
3.0
>>> str(3) # convert int to string
'3'
```

### 4.2 Strings

Strings are collections of characters. You can select parts of the string with square brackets, join them with `+`, but they cannot be modified (they are *immutable*). You can assign new strings to the same name, but you cannot modify a string “in place”.

```
>>> a='hello there'
>>> b='fred'
>>> a + ' ' + b
'hello there fred'
>>> a[1:4] # characters count from 0, and end is +1
'ell'
```

```

>>> a[1:]      # select string from character 1 to end
'ello there'
>>> a[:4]      # select characters 0 through 3
'hell'
>>> a[:-1]     # negative indices count from end
'hello ther'
>>> len(b)     # string length
4
>>> a="double quotes also work"
>>> a="you can embed\nnew lines"
>>> print a
you can embed
new lines
>>> a = '  lots of space  '
>>> b = a.strip() # remove spaces from start and end
>>> print b
'lots of space'
>>> a.split()  # break a string into a list
['lots', 'of', 'space']
>>> 'hi, there, fred'.split(',') # split on commas
['hi', ' there', ' fred']
>>> str(4)     # convert number to string
'4'
>>> a = """This is a
multi line string. Which
can go over many lines, and include new
lines naturally"""
>>> del a     # delete variable (forget it)

```

Strings have lots of sophisticated methods (like split) which can find substrings, replace substrings, or change case:

```

>>> a="This is a test"
>>> a.find('test')           # get starting character of substring
10
>>> a[:a.find('test')]      # slice according to character
'this is a '
>>> 'SHOUTING'.lower()     # convert to lowercase
'shouting'
>>> a.replace('test', "fish") # replace substring
'This is a fish'

```

One useful feature is the formatting operator, % which formats numbers and strings

```

>>> '%s counts to %i sleep %.3f but %e' % ('fred', 10, 4.1, 3)
'fred counts to 10 sleep 4.100 but 3.000000e+00'

```

### 4.3 Lists

Lists are collections of any types of variable (even lists). They are very powerful. They can be modified (they are *mutable*).

```

>>> a=[1,2,3,'cow']
>>> a[1]           # index list
2
>>> a[2:4]        # slice list

```

```

[3, 'cow']
>>> a[2] = 'moo'           # assign element
>>> a
[1, 2, 'moo', 'cow']
>>> len(a)                 # length of a
4
>>> a.append('steak')     # add to list
>>> a
[1, 2, 'moo', 'cow', 'steak']
>>> a.insert(1, 1.5)      # insert into list
>>> a
[1, 1.5, 2, 'moo', 'cow', 'steak']
>>> a.append([1,2,3])     # lists can contain lists
>>> a
[1, 1.5, 2, 'moo', 'cow', 'steak', [1, 2, 3]]
>>> a.index('moo')       # find item in list
3
>>> a[-1][1]            # index item in list in list
2
>>> a = []              # a blank list
>>> a.append('foo')
>>> a
['foo']
>>> range(5)            # creates a list filled with numbers
[0, 1, 2, 3, 4]
>>> a = [5, 3, 1, 0, -1, 6]
>>> a.sort()            # sort list in-place
>>> a
[-1, 0, 1, 3, 5, 6]
>>> del a[0:2]          # delete first two elements

```

## 4.4 Tuples

Tuples are like lists but they cannot be changed. They can be used as dictionary keys, unlike lists.

```

>>> a=(0,1,2,3)         # make tuple
>>> a[-4:-1]           # slice tuple
(0, 1, 2)
>>> a[-3:-1]          # slice again
(1, 2)
>>> a[1]='fred'       # this doesn't work!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> list(a)            # convert tuple -> list
[0, 1, 2, 3]
>>> tuple(['hi', 'ho']) # convert list -> tuple
('hi', 'ho')

```

One-length tuples are a special case as otherwise they would be confused with brackets.

```

>>> a = (1,)          # a one-length tuple
>>> a = ()            # a zero-length tuple

```

## 4.5 Dictionaries

Dictionaries are very powerful objects mapping between different values. They are mutable.

```
>>> a = {} # empty dictionary
>>> a[3] = 10 # map key 3 to value 10
>>> a[4] = 'fred' # map key 4 to value 'fred'
>>> a[3] # prints out value mapped to 3
10
>>> a['amazing'] = 'wibbles'
>>> a['fred'] = 3.141592
>>> a['amazing']
'wibbles'
>>> a['amazing'] = 'spam'
>>> 'fred' in a # test for existence
True
>>> a = {'fred': 10, 'tim': 15, 'keith': -1}
>>> a['tim'] = a['fred'] + 42 # reassign value
>>> a
{'tim': 52, 'keith': -1, 'fred': 10}
>>> a.keys() # return list of keys
['tim', 'keith', 'fred']
>>> a.values() # return list of values
[52, -1, 10]
>>> a.items() # return tuple pairs of key,value
[('tim', 52), ('keith', -1), ('fred', 10)]
>>> a[(1,2,3)] = -1 # tuples can act as keys (e.g. grid)
>>> a[(1,2,3)]
-1
```

## 4.6 File objects

Files correspond to files on the disk. You can read or write to them. You can also make file-like objects which read webpages, or across the network. Suppose `fred.dat` contains letters of the alphabet on separate lines.

```
>>> f = open('fred.dat') # open file for reading
>>> print f.readline() # prints a (newline \n is also returned)
a

# or
>>> s=f.read() # would read entire file and put into s

# or
>>> lines = f.readlines() # reads each line and construct a list
>>> print lines
['a\n', 'b\n', ....]
>>> f.close() # close file
```

You can also write to files:

```
>>> f = open('out.dat', 'w') # open for writing mode
>>> print >>f, "a is", 42 # print to file
>>> f.write('This is a line\n') # write a string
>>> f.close()
```

Mode 'a' can append to a file (add to it).

## 5 Subtleties

Python does not have *variables* as such, it has objects which are referenced by names. The distinction is important sometimes for mutable objects (objects which can be modified), like a list or dict.

```
>>> a = [1,2,3,4]      # make list object
>>> b = a              # makes b point to same list object
>>> b.append(5)        # add item to list object
>>> a                  # a and b point to same object
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>> b = list(a)        # make b a new list based on a
>>> b.append(6)
>>> a                  # a is not modified
[1, 2, 3, 4, 5]
```

Really the command `a = b` assigns a new name “a” for the object “b”. This allows you to do this

```
def func(x):
    x.append(5)

a = [1,2,3]
func(a)
print a                # outputs [1, 2, 3, 5]
```

Note that immutable objects (like numbers, strings or tuples) do not have this property.

```
>>> a = 10             # makes a point to object 10
>>> b = a              # makes b point to object 10
>>> a = 11             # makes a point to object 11
>>> print b           # prints 10
```

so

```
def func(a):
    a = 10              # makes a point to 10, but this is
                       # not the same object which was passed

x = 11
func(x)
print x                # prints 11
```

## 6 Controlling program flow

To write proper programs, you need subroutines, if statements, loops and so on. Python has lots of these. Its unusual feature, however, is that it uses indentation to mark blocks of programs. In emacs the tab key automatically remembers the indentation. You use backspace to move back to the previous indentation. To get the proper emacs mode you may need to modify your `.emacs` file. See Appendix A.

### 6.1 Subroutines, procedures and functions

In Python subroutines, procedures and functions are basically the same thing, so I'll use the terms interchangeably. Here is an example program script demonstrating the use of a procedure, and showing the how people document Python programs.

```
#!/usr/local/bin/python

def hiThere(a, b):
    """Optional docstring describing the function:
    Arguments:
        a - Name of person a
        b - Name of person b
    Using docstrings is 'best practice'
    The help(x) command prints out the docstring for the function."""
    print "Hello", a
    print "Hi there", b

hiThere("Fred", "Jim")
x = 'mike'
y = 42
hiThere(x, y)
```

prints

```
Hello Fred
Hi there Jim
Hello mike
Hi there 42
```

Subroutines can return values with the return statement:

```
def add1(a):
    print 'yawn'
    return a + 1

def addsub1(a):
    return (a+1, a-1)

print add1(10)
a, b = addsub1(10)
print a, b

outputs:
yawn
11
11 9
```

Another useful feature is that functions and subroutines can have parameters which have “default values” which can be overridden. This is useful for optional parameters to functions.

```
def greet(name, greeting='Hello'):
    '''A function to greet someone'''
    print greeting, name

greet('world') # prints 'Hello world'
greet('cruel world', greeting='Goodbye') # prints 'Goodbye cruel world'
```

## 6.2 Modules

You can divide your program into different modules, to separate the different parts into reusable components. For instance if `mymodule.py` contains

```
"""Modules should also contain docstrings.

This module contains a useful function myfunc to print hello to
someone."""

def myfunc(a):
    print "Hello", a
```

In another python program, say, `mainprogram.py`, you could do

```
import mymodule

mymodule.myfunc('world')
```

which prints "Hello world". Alternatively you can do

```
from mymodule import *
myfunc('Fred')
```

Which imports everything into the current *namespace*. This could be considered bad practice as things in `mymodule` could override things in your main program. Avoid this by doing

```
from mymodule import myfunc
myfunc('foo bar')
```

Modules can import other modules. You can also use modules you have created interactively:

```
>>> import mymodule
>>> mymodule.myfunc('goodbye')
Hello goodbye
```

In fact there is a massive library of useful standard modules, which I will briefly overview in Section 7.

## 6.3 If statements

If statements test whether the thing to their right evaluates to `True` or `False` (which are special values), and execute different parts of the code. Boolean operators test whether statements are true:

```
>>> 5 == 6 # equal
False
>>> 5 != 6 # not equal
True
>>> 5 < 10 # less than
True
```

```

>>> 10 > 5      # greater than
True
>>> 5 <= 5     # less than or equal
True
>>> 5 >= 6     # greater or equal
False
>>> if 5 < 10:
...     print "hello"
...
hello
>>> a = [1, 2, 3]
>>> b = a
>>> b is a      # do two names point to the same object?
True
>>> b is not a  # do they point to different objects?
False
>>> b = [1, 2, 3]
>>> b is a      # two lists with the same contents are not the same list
False
>>> a == b      # but they are equal
True

```

None is a special value meaning “nothing”. You can test whether something is None by using `is None`. This is actually how you use an `if` statement. Can you predict the output?

```

def test(a):
    if a < 10:                # brackets aren't required
        print "a was less than 10"
    elif a <= 15:            # optional elif means else-if
        # multiple elif are allowed
        print "a was less than or equal to 15"
    else:                    # none of the if or elif match, optional
        print "a was greater than 15"

test(1)
test(12)
test(100)

```

You can join the various tests with `and`, `or` and `not`, and also brackets:

```

>>> 5 < 10 and 100 > 20 and (not 'fred' == 'blogs')
True

```

## 6.4 For loops

For loops loop over something, such as a list, tuple, string, dictionary or file.

```

a = ['foo', 'fred', 42]
for i in a:
    print i

```

outputs:

```

foo
fred
42

```

dictionaries:

```
a = {}
a['test'] = 42
a['bar'] = 6

for x, y in a.items():
    print x, y
```

outputs:

```
test 42
bar 6
```

Looping over files is very helpful. This example prints the sum of the numbers in a file (which are stored in separate lines).

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

f = open('test.dat')
sum = 0.
for line in f:
    sum = sum + float(line)
print sum
```

You can break out of a loop with the `break` statement, or skip to the next iteration with `continue`. `enumerate` is a very useful function which counts which number of the iteration you are on

```
a = ['cow', 'sheep', 'horse']
for num, val in enumerate(a):
    print num, val
```

outputs:

```
0 cow
1 sheep
2 horse
```

There is also the `xrange` function for looping over numbers:

```
>>> for x in xrange(5):
>>>     print x*0.1
>>>
0.0
0.1
0.2
0.3
0.4
```

Try `xrange(5, 10)` and `xrange(10, 5, -1)`.

As an aside, there is a shortcut version of loops called a *list comprehension* which is very convenient:

```
>>> a = [1,2,3,4]
>>> b = [i*2 for i in a]           # make a new list with values*2
>>> b
[2, 4, 6, 8]
>>> c = [i*3 for i in a if i != 2] # new list*3, excluding 2
>>> c
[3, 9, 12]
```

```
>>> d = ['a', 'dairy', 'farm']
>>> e = [len(x) for x in d]
>>> e
[1, 5, 4]
```

## 6.5 While loops

While loops loop while something is true.

```
>>> a = 1
>>> while a < 5:
>>>     print a
>>>     a = a + 1
>>>
outputs:
1
2
3
4
```

```
x = []
a = 1
while a < 5:
    x.append(a)
    a = a + 1
print x

outputs:
[1, 2, 3, 4]
```

break and continue also work.

## 6.6 Exceptions

When the program fails somewhere, an exception is generated. By default this stops the program. Examples include trying to open nonexistent files, or converting invalid strings to ints, e.g. `float('cow')`. You can *catch* exceptions to prevent the program from stopping and handle the error, e.g.

```
import sys

filename = 'stupid.dat'
try:
    f = open(filename)
except IOError:
    # the file did not open
    print "The filename", filename, "does not exist!"
    sys.exit(1)
    # exit program with a fail error code
# print file contents
print f.read()
```

or a more complex example which reads from the user

```
print "Enter some numbers, or a blank line to stop"
while True:
    # get a line from the user and remove any spaces around it
    line = raw_input().strip()
```

```

# break out of the loop if the user enters a blank line
if line == '':
    break

try:
    # try to convert text to float number
    a = float(line)
    print "The value you entered plus 10 is", a+10.
except ValueError:
    # conversion failed
    print "You entered an invalid number"

```

You can catch different kinds of exceptions. The name of the exception is printed out if it occurs and is not caught.

## 6.7 Objects

This is more complex, but objects are very useful. They are used to combine data together with *methods* which operate on them. They are used to keep data about one thing, e.g. a star, and the functions or procedures which operate on them in one place. This makes programs more robust. The way you used them is to define a *class* which describes the object and its methods. You can then create objects based on the class.

At the most simple level, you can make a class which can be used as a substitute of a C-style structure:

```

>>> class Empty(object):
>>>     pass                # does nothing, but is required if empty

>>> a = Empty()
>>> a.fred = 'smith'
>>> a.foo = [1,2,3,4]
>>> print a.fred
smith

```

A more complete example defines a *constructor* and some methods which operate on the data. They all access a special variable called *self* which refers to the object the method is applied to.

```

class Name(object):
    """This is an example class. A class defines a type of object.
    New objects are created based on a class."""

    def __init__(self, name):
        """This function __init__ is special. It is called when a new
        object is constructed.

        self is a special argument which refers to the object itself
        name here is a parameter to create the new object, you can use
        as many as you want
        """

        # store name in the object
        self.name = name

    def sayHello(self):
        """This is a class method which prints out the stored name.

        Again self is a special parameter which refers to the object
        """

```

```

    print "Hello", self.name

def joinName(self, othername):
    """This is another class method which returns the original
    name joined with another name."""
    return self.name + " " + othername

a = Name('Fred')           # makes a new object of type Name
                           # and calls Name.__init__(a, Fred)
a.sayHello()              # calls Name.sayHello(a)
b = a.joinName('Smith')   # calls function Name.joinName(a, 'Smith')
print b                   # prints out "Fred Smith"
a.test = 42               # assigns attribute of class directly
print a.test              # prints 42

b = Name('Joe')           # makes a different object of same class
print b.joinName('Bloggs') # prints "Joe Bloggs"

```

James Graham has provided a much more complete example of an object in Appendix B which demonstrates useful features such as “operator overloading”. It is a new class of number which stores its associated dimensions, so that it can be used for dimension checking.

## 7 Standard modules

These modules are documented at <http://www.python.org/doc/lib/lib.html> . I'll list some of the most useful ones. Note that the online documentation may list features for newer versions of Pythons than the one you are using.

### 7.1 math

See <http://www.python.org/doc/lib/module-math.html> . `math` supplies common mathematical functions, e.g.

```
import math

print math.sin(math.pi/4)    # sin(pi/4)
print math.log10(1e4)       # log base 10
print math.log(1e4)         # natural log

or

from math import *
print exp(1)                 # e**1
print sqrt(2)                # sqrt(2)
```

### 7.2 cmath

Python also supports complex numbers (I didn't tell you that earlier). You use them by specifying a suffix "j" or "J" which means the imaginary number. The `complex` function also creates them.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
>>> a = 2 + 0.5j
>>> a.real
2
>>> a.imag
0.5
```

The `cmath` module defines mathematical functions which work on complex numbers (e.g. `exp`, `sin`, `acos`...). See <http://www.python.org/doc/lib/module-cmath.html>

### 7.3 sys

See <http://www.python.org/doc/lib/module-sys.html> . `sys` has various useful parts which control the program.

```
import sys

print sys.argv                # the parameters to a script
                              # from the command line
print sys.argv[0]            # script name
print sys.argv[1]            # first parameter
```

```
...
sys.exit(0)           # normal program exit
sys.exit(1)           # error program exit
```

## 7.4 os

See <http://www.python.org/doc/lib/module-os.html> . os can be used to manipulate file, or running programs, or run external unix commands.

```
import os

os.remove('foo.dat')      # delete file foo.dat
os.mkdir('dir')           # make directory
os.rename('foo', 'bar')   # rename file

os.system('ls *.txt')     # execute unix shell command IMPORTANT!
                          # returns 0 for success
```

## 7.5 random

See <http://www.python.org/doc/lib/module-random.html> . Generate random numbers using random.

```
import random

print random.random()     # random number 0 <= r < 1
print random.randint(1, 10) # random integer 1 <= r <= 10

a = [1,2,3,4]
random.shuffle(a)         # randomise list order
print a
print random.gauss(2, 1)  # gaussian distributed random mean 2, sigma 1
```

## 7.6 glob

See <http://www.python.org/doc/lib/module-glob.html> . Generate lists of files from shell wildcard expansion.

```
import glob

x = glob.glob('*.txt')    # get a list of *.txt files
print x
```

may output (depending on files!)

```
['foo.txt', 'bar.txt', 'fred.txt']
```

## 7.7 subprocess

See <http://www.python.org/doc/lib/module-subprocess.html> . This is very useful. It allows you to run external unix commands, send them input and read their output.

```
import subprocess

# does unix command, ls -l, and returns exit status
```

```
subprocess.call(["ls", "-l"])

# this executes the ls command, and reads its output
# as a list containing each line of output
>>> proc = subprocess.Popen('ls', stdout=subprocess.PIPE)
>>> lines = proc.stdout.readlines()
>>> print lines
['adam\n', 'bin\n', 'code\n', ...]
```

## 7.8 re

See <http://www.python.org/doc/lib/module-re.html> . The `re` module lets you do lots of fancy text matching and substitution with *regular expressions*. See the book *Mastering regular expressions* to see how powerful (and how complex!) they can be. This example extracts all of the positive integers out of a string.

```
>>> import re
>>> re.findall('[0-9]+', 'Fred at 9 carrots on the 12th')
['9', '12']
```

## 7.9 urllib

This module reads web pages across the internet. See <http://www.python.org/doc/lib/module-urllib.html> .

```
>>> import urllib
>>> f = urllib.urlopen('http://www.google.co.uk')
>>> print f.readline()
<html><head><meta http-equiv="content-type" ...
```

This module can be used to download things from ADS, NED, for example.

## 8 Other modules

There are a variety of modules available for download. These include numerical libraries for handling matrices, tensors, vectors, etc, plotting routines, and libraries for integration, solving equations and so on. Other modules exist for combining Fortran and C with Python.

### 8.1 numpy

Numpy is a new library which can manipulate arrays of numbers very efficiently. You may come across numarray which is an older library but similar. See <http://www.scipy.org/Documentation> for details. It also has random number routines.

```
>>> import numpy
>>> a = numpy.identity(4)      # create identity matrix
>>> a                          # print it out
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
>>> a*4 - 1                    # do maths
array([[ 3, -1, -1, -1],
       [-1,  3, -1, -1],
       [-1, -1,  3, -1],
       [-1, -1, -1,  3]])
>>> numpy.invert(a)           # matrix inversion
array([[ -2, -1, -1, -1],
       [-1, -2, -1, -1],
       [-1, -1, -2, -1],
       [-1, -1, -1, -2]])
>>> numpy.arange(10)*3. / 2   # make series
array([ 0. ,  1.5,  3. ,  4.5,  ...])
>>> a[1,:]                    # slice matrix
array([0, 1, 0, 0])
```

### 8.2 scipy

SciPy contains lots of useful numerical method things such as integration. See <http://www.scipy.org/Documentation> for details.

```
>>> from scipy.integrate import quad
>>> def square(x):
...     return x**2
...
>>> quad(square, 0, 1)
(0.33333333333333331, 3.7007434154171879e-15)
```

### 8.3 pyfits

pyfits can read or create FITS files, as used in astronomy. There is a good introduction to that program here [http://www.stsci.edu/resources/software\\_hardware/pyfits](http://www.stsci.edu/resources/software_hardware/pyfits)

## A Enabling Python mode in emacs

You unfortunately need the following in `.emacs` to enable the python mode on the Sun system. Most Linux distributions support this automatically. In this case indentation is not important.

```
;; enable python
(setq auto-mode-alist
      (cons ('("\\.py$" . python-mode) auto-mode-alist))
      (setq interpreter-mode-alist
            (cons ('("python" . python-mode)
                  interpreter-mode-alist)))
      (autoload 'python-mode "python-mode" "Python editing mode." t)
```

To get this more easily, you can copy my `.emacs` file to get support (on the Sun system):

```
> cp /home/jss/.emacs /home/yourusername/
```

## B Complete class example

Written by James Graham.

```
class PhysScalar(object):
    """A physical quantity with an set of
    dimensions e.g. Length, Distance, etc. Operations involving
    physical scalars are checked for matching dimensions. This does
    not account for the possibility that the units are wrong
    e.g. adding feet to metres"""

    ValidDimensions = ["L", "M", "T", "K"]#Length, mass, time, temperature

    def __init__(self, value, dimensions):
        """value - The magnitude of the quantity
        dimensions - a dict of the form {dimension:power}
        e.g. {"M":2} for an area
        """
        self._value = float(value)
        self.dimensions=dimensions
        #Check all the dimensions are valid
        #This is strictly unnecessary b
        for item in dimensions.keys():
            if item not in self.ValidDimensions:
                raise ValueError, "Unrecognised dimension %s"%str(item)

    def __str__(self):
        """This method is called when we try to convert something to a
        string"""
        #Create a string like L**2 T**-1 for the dimensions
        dimensionsStr = " ".join(["%s**%s"%(item[0],str(item[1]))
                                   for item in self.dimensions.items()])
        return "%s %s"%(str(self._value), dimensionsStr)

    def __repr__(self):
        """This method is used to represent the current object as a sting
        e.g. on the interactive prompt"""
        return "PhysConst(%f, %s)"%(self._value, str(self.dimensions))

    #We use "magic" methods to define operators that will allow us to add,
    #subtract and multiply PhysConst objects. These operations all return
    #new objects

    def __add__(self, other):
        """Add two quantities if they have the same dimensions.
        Also allows the addition of dimensionless quantities
        i.e. plain python floats"""

        #Check if the object being added has a 'dimensions' attribute
        #If it does we assume it can be added by comparing values
        if hasattr(other, 'dimensions'):
            if self.dimensions == other.dimensions:
                return PhysScalar(self._value+other._value, self.dimensions)
            else:
                raise TypeError, "Incompatible dimensions"
```

```

    else:
        return PhysScalar(self._value+other, self.dimensions)

def __sub__(self,other):
    """Subtraction self-other"""
    #Implement subtraction as addition with a minus;
    #this isn't efficient but reduces code
    return self.__add__(-1*other)

def _operator(self, other, valueFunc, dimFunc, right=False):
    """Base function used for multiplication and division.
    valueFunc(selfValue, otherValue): a function takes two parameters and
    implements the operation (e.g. multiplcation, division) on the values
    dimFunc(power1, power2): a function that takes two parameters and
    implements the operation on the dimensions
    right: True if self on the rhs of the operation

    e.g. for multiplication valueFunc(x,y) should return x*y and
    dimFunc(x,y) should return x+y

    Limitations - No support for self on the rhs if the lhs object has a
    dimensions attribute but does not implement operators itself
    """

    #Dimensions of the output object
    newDims = {}

    if hasattr(other, 'dimensions'):
        #Add all the dimensions from the self to the output dimensions
        newDims.update(self.dimensions)

        #Now add the dimensions from other
        for key,value in other.dimensions.iteritems():
            if key in newDims:
                newDims[key] = dimFunc(newDims[key], value)
                if newDims[key] == 0:
                    #Delete dimensions that have zero value
                    del newDims[key]
            else:
                newDims[key] = dimFunc(0.0, value)
        return PhysScalar(valueFunc(self._value,other._value), newDims)
    else:
        for key,value in self.dimensions.iteritems():
            if right:
                value = dimFunc(value, 0.0)
            newDims[key] = value
        return PhysScalar(valueFunc(self._value,other), newDims)

def __mul__(self,other):
    def add(x,y):
        return x+y
    def mult(x,y):
        return x*y
    return self._operator(other, mult, add)

```

```

def __div__(self, other):
    def subtract(x,y):
        return x-y
    def divide(x,y):
        return x/y
    return self._operator(other, divide, subtract)

#These functions define operations other + self
__radd__ = __add__
#other*self
__rmul__ = __mul__
#other-self
def __rsub__(self, other):
    """Right hand subtraction other-self"""
    return -1*self.__sub__(other)
#For python 2.5
__truediv__ = __div__

def __rdiv__(self, other):
    """Right hand division used for e.g. 2.0/PhysScalar(2.0, {"L":1})"""
    def rsubtract(x,y):
        return y-x
    def rdivide(x,y):
        return y/x
    return self._operator(other, rdivide, rsubtract, True)

def __pow__(self, power):
    """Powers e.g. PhysScalar(2.0, {"L":1.0})**2"""
    newDims = {}
    for key, value in self.dimensions.iteritems():
        newDims[key] = value * power
    return PhysScalar(self._value**power, newDims)
"""
#Examples
>>> a = PhysScalar(2.0, {"M":1,"L":2,"T":-2})
>>> a
PhysConst(2.000000, {'M': 1, 'L': 2, 'T': -2})
>>> print a
2.0 M**1 L**2 T**-2
>>> a**2
PhysConst(4.000000, {'M': 2, 'L': 4, 'T': -4})
>>> b = PhysScalar(1.0, {"M":1,"L":2,"T":-2})
>>> a+b
PhysConst(3.000000, {'M': 1, 'L': 2, 'T': -2})
>>> a-b
PhysConst(1.000000, {'M': 1, 'L': 2, 'T': -2})
>>> a+2 #We allow adding/subtracting dimensionless scalars but we could prevent
this
PhysConst(4.000000, {'M': 1, 'L': 2, 'T': -2})
>>> c = PhysScalar(2.5, {"K":-2})
>>> a*c
PhysConst(5.000000, {'K': -2.0, 'M': 1, 'L': 2, 'T': -2})
>>> a/c

```

```
PhysConst(0.800000, {'K': 2.0, 'M': 1, 'L': 2, 'T': -2})
>>> 2/a
PhysConst(1.000000, {'M': -1.0, 'L': -2.0, 'T': 2.0})
>>> a+c
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "dimension_check.py", line 51, in __add__
    raise TypeError, "Incompatible dimensions"
TypeError: Incompatible dimensions
"""
```

## C Python example questions

1. At the command prompt, calculate  $1./6.$ , 2 to the power 85,  $\cos(4.12)$ ,  $\pi/3$ .
2. Join the strings 'Test' and '123' to make variable c
3. Remove the spaces from the string ' humbug '. Write it to a file `foo.dat`.
4. Make a list containing [5, 6, 7]. Add the item 9 to it. Insert 0 at the beginning of the list. Print out the 3rd item in the list now.
5. Print the values from 0 to 10 and its square as columns.
6. Take an input list of numbers in a file (stored in a column). Take the sum of the numbers, calculate their mean and standard deviations.
7. Count the number of occurrences of a word in a file. (Hint try the string split function, and a dictionary).
8. Write a program to run the unix command `ls -lrt` and print its last output line.