High lever scientific computing with Python

Romualdo Pastor
Departament de Fisica
Modul B4, 2a planta, Campus Nord
email: romualdo.pastor@upc.edu

Introduction

High-level vs low-level programming controversy

You might have heard about the controversy between high-level vs low-level programming languages for scientific computation

- ➡High-level, such as Python, are very easy to program, but very slow
- ➡Low-level, such as Fortran or C, are difficult to code, but very fast

The usual conclusion is that real computational science is to be performed in low-level languages, because they are the fastest

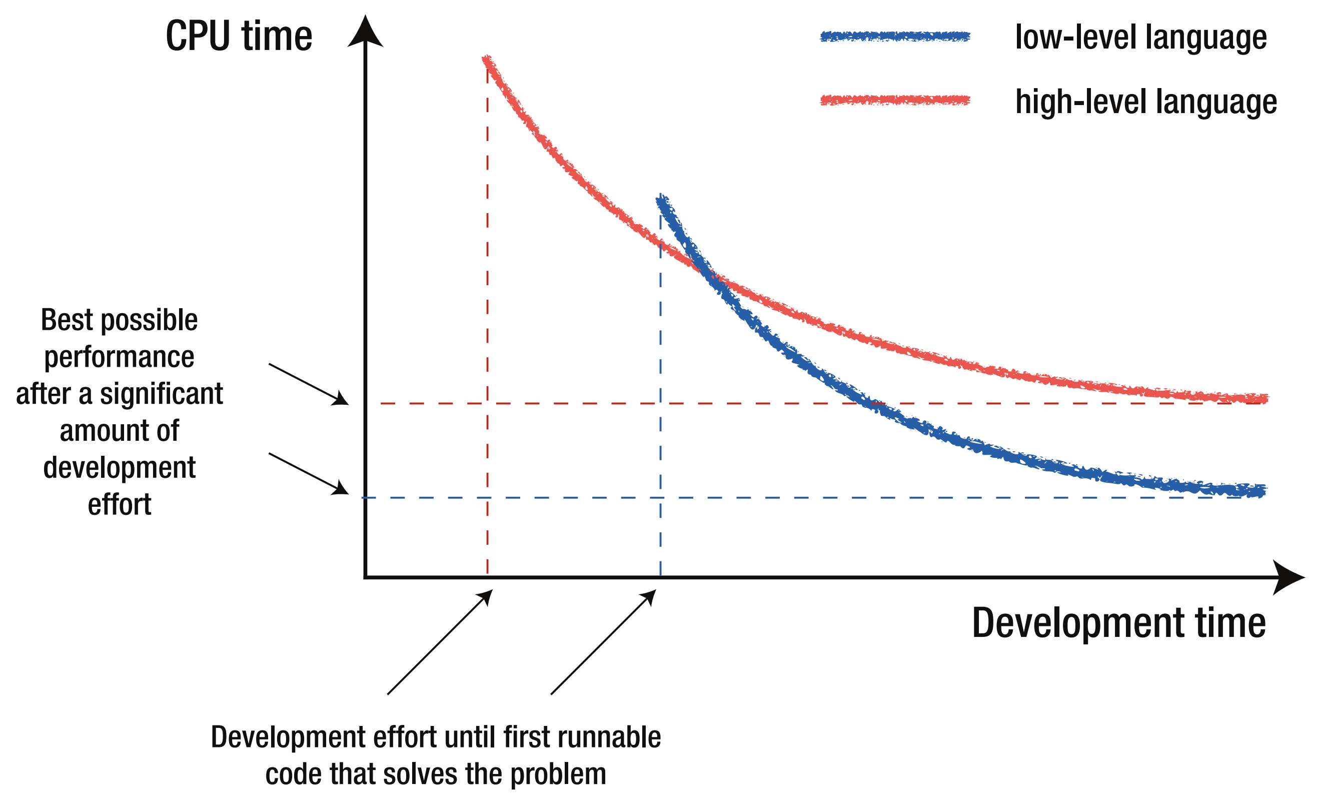
Trade-off between high- and low-level languages

This simple conclusion however does not take into account a few hidden facts

➡Computer CPU is cheap, and ever increasing in power

➡Programmer time (your time) is expensive

➡You don't always need the super-fastest code, but just a decent version, specially when you are experimenting with a new problem



"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**"

— Donald Knuth

What python offers for scientific programming

- ➡Very easy to get to a working program (specially to check new ideas)
- ➡A whole computing environment in which data analysis and graphics can be merged with calculations (Ipython)
- ➡Easy to profile, to detect computational bottlenecks
- ➡Easy to speed up
  - ❖You can easily link with highly optimized libraries in C/Fortran
  - ❖You can code specific time-consuming parts in C/Fortran
- ➡Huge ecosystem of user base and scientific libraries
  - ❖Many problems have already been solved by somebody else, google them

Introduction to Python

What is Python

Main characteristics of python

- ➡Interpreted:
  - ❖Not compiled. An interpreter reads and executes line by line the program files
- ➡Very high level
  - ❖No need to worry about low-level details such as memory management
- ➡Objected oriented:
  - ❖Everything is an object, even user defined functions
- ➡Dynamic:
  - ❖Variables can change type
- ➡Extensible
  - ❖Parts of the code can be written in faster C/Fortran and used then in Python
- ➡Open source
  - ❖Completely free to install and read its base code
- ➡Easy to learn
- ➡Easy to write

Why Python

- ➡ Programs are shorter, due to the higher level of the language (avoiding boilerplate code)
  - ❖ Less bugs
- ➡ Running is slower but writing can be much faster than a compiled language
  - ❖ The total time writing + executing can be much shorter with Python, depending of the problem
  - ❖ Plus, extensibility allows to write directly in C/Fortran critically slow parts of the code
- ➡ Batteries included
  - ❖ Python comes with a standard library to perform many tasks, which can avoid having to write code in many situations: Just load the library
- ➡ Third party libraries
  - ❖ A very large community of users have developed a large number of high quality libraries for a very large number of purposes
    - ▸ Among them, scientific computing

Why Python

- As it is a general-purpose language, it is much easier to do simple, not related with scientific computation or number crunching, with it
    - String operations, downloading data from the web, reading and writing data/ text files in specific formats, etc.
        - Try to do that in C/Fortran

Installing Python

Pythons comes installed by default in Linux and Mac OSX

You can install it in Windows using packaged versions, available from different sources

The preferable and simplest way to install a full Python environment is by means of one of different scientific distributions

One of the best one: Anaconda distribution, freely available at

https://store.continuum.io/cshop/anaconda/

Full package with most of the third party libraries we will need for scientific purposes

- Please, install it in your laptops for the next session, we will be using it in the future.

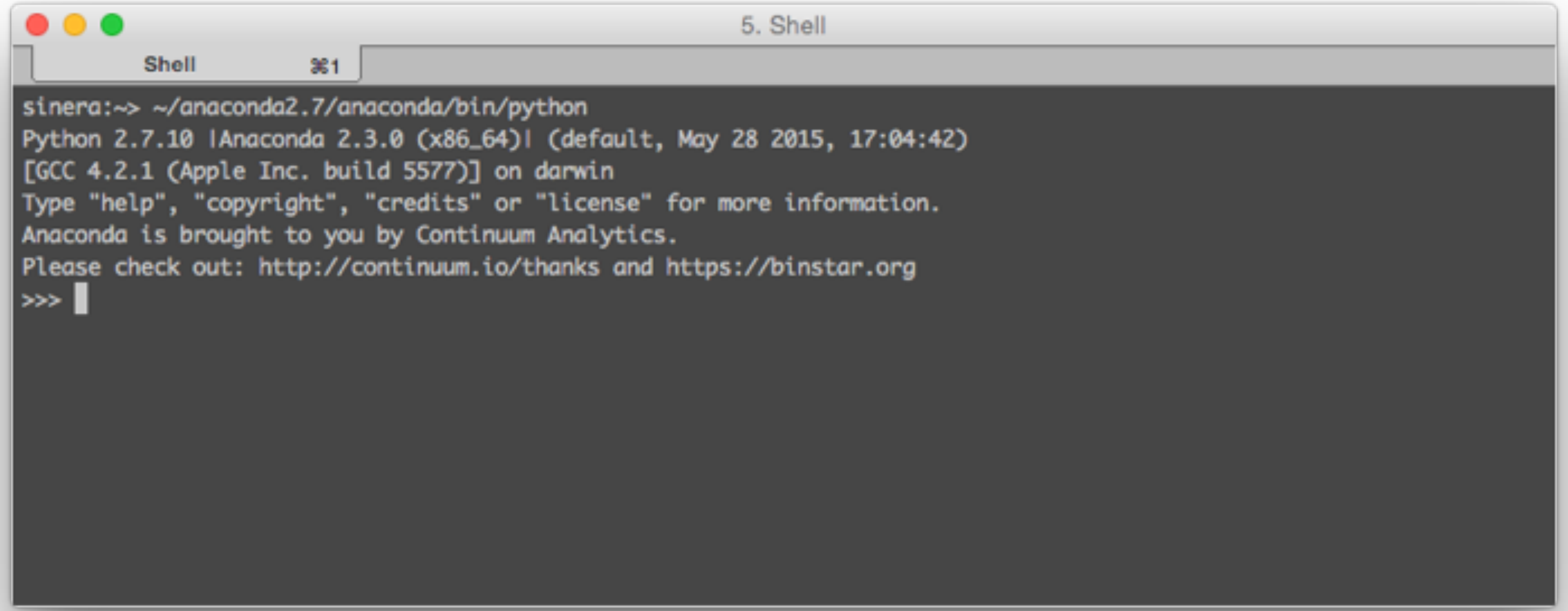
At present, two versions of Python are available: 2.7 and 3.4

- Please, select version 2.7

Python Brainstorming

How to start Python

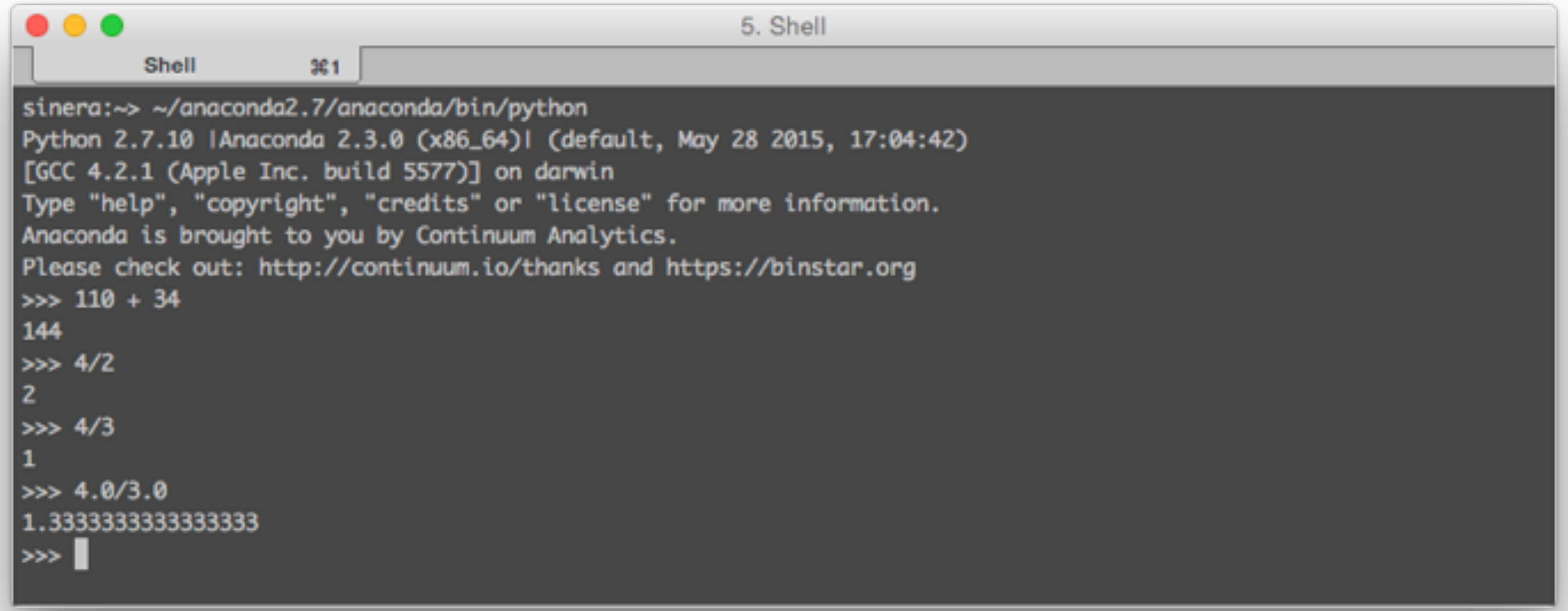
Interactive:



```
sinera:~> ~/anaconda2.7/anaconda/bin/python
Python 2.7.10 |Anaconda 2.3.0 (x86_64)| (default, May 28 2015, 17:04:42)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> 
```

Otherwise, it can be started from the launcher (specially in windows)

You can use interactive Python as a high-level calculator



```
sinera:~> ~/anaconda2.7/anaconda/bin/python
Python 2.7.10 |Anaconda 2.3.0 (x86_64)| (default, May 28 2015, 17:04:42)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> 110 + 34
144
>>> 4/2
2
>>> 4/3
1
>>> 4.0/3.0
1.3333333333333333
>>> 
```

Python data types

Numbers: `f_num = 10.0   # a float`

`i_num = 25     # an integer`

Booleans: `True and False`

Strings : `my_string = “Always look on the bright side of life.”`

Lists : `my_list = [“yo", 24, "blah", "go away”)]`

`my_list[1] -> 24    # list indexes starts with 0 !!`

✦Lists are mutable: you can change the value

Tuples : `my_tup = (1, 4, 32, "yoyo", ['fo', ‘moog'])`

`my_tup[2] -> 32`

✦Tuples are immutable : you cannot change the value

Dictionaries : `my_dict = {'a': 24.5, 'mo': 'fo', 42: ‘answer'}`

`my_dict[‘mo’] -> ‘fo’`

Sets: Unordered list of unique elements:

```
names = ["John", "Peter", "Alice", “John”]
set(names) -> {'Alice', 'John', 'Peter'}
```

Operators in Python

+ : Addition of numbers and concatenation of strings and lists

- ➡"Hello " + "World" = "Hello World"
- ➡[1,2] + [2,3] = [1,2,2,3]

- : Subtraction on numbers

* : Multiplication of numbers and strings/lists by numbers

- ➡"Hello" * 2 = "HelloHello"
- ➡[1,2] * 2 = [1,2,1,2]

/ : Division of numbers: Watch out! integer division of integers in Python 2.7

- ➡5/4 = 1
- ➡5.0 / 4.0 = 1.25
  - ❖Force integers to floats if needed: float(5)/float(4) = 1.24

% : Modulus (residual of division)

- ➡5 % 3 = 2

Shorthand notations: a = a op b -> a op = b

- ➡a = a + b -> a += b ; a = a * b -> a *= b

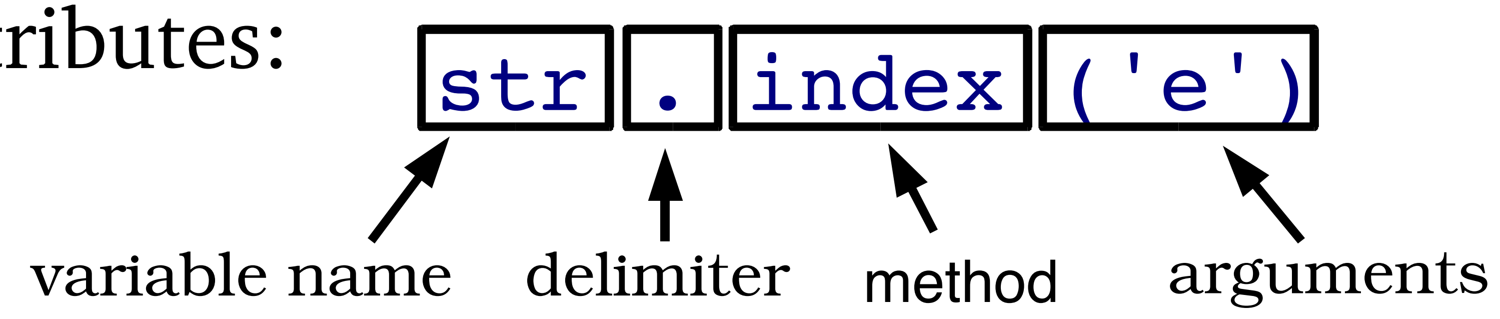
Everything in Python in an object

Everything in a Python is an objet:

➡A data container that has associated a different number of functions, called methods, that can operate on the value of the data

Methods are called with an dot infix notation

Object Attributes:



"Everything" is an object, even functions

➡You can pass a function as an argument to other functions!

There are different methods associated to each data type. Can explore them using the `dir()` function

```
>>> dir(str)
['capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', … ]
```

List methods and functions

```
>>> lst = ['3', 45, 'frogger', 2]
>>> lst[2]
'frogger'
>>> del lst[2]
>>> lst
['3', 45, 2]
>>> lst.append('help')
>>> lst
['3', 45, 2, ‘help']
```

`list[n:m]` : slicing; yields the elements in list from n to m-1

- `list[:m]` : from beginning to m-1
- `list[n:-1]` : from n to end

`list.append(x)` : add x to the end of the list

`list.sort()` : sort the list

`list.reverse()` : reverse the list

`len(list)` : Number of elements in the list

Dictionary methods and functions

len(*d***)** **:** number of items in *d*

*d***[***k***]** **:** item of *d* with key *k*

*d***[***k***] =** *x :* associate key *k* with value *x*

del *d***[***k***]** **:** delete item with key *k*

k **in** *d :* test if *d* has an item with key *k*

*d***.items()** **:** a copy of the (*key*, *value*) pairs in *d*

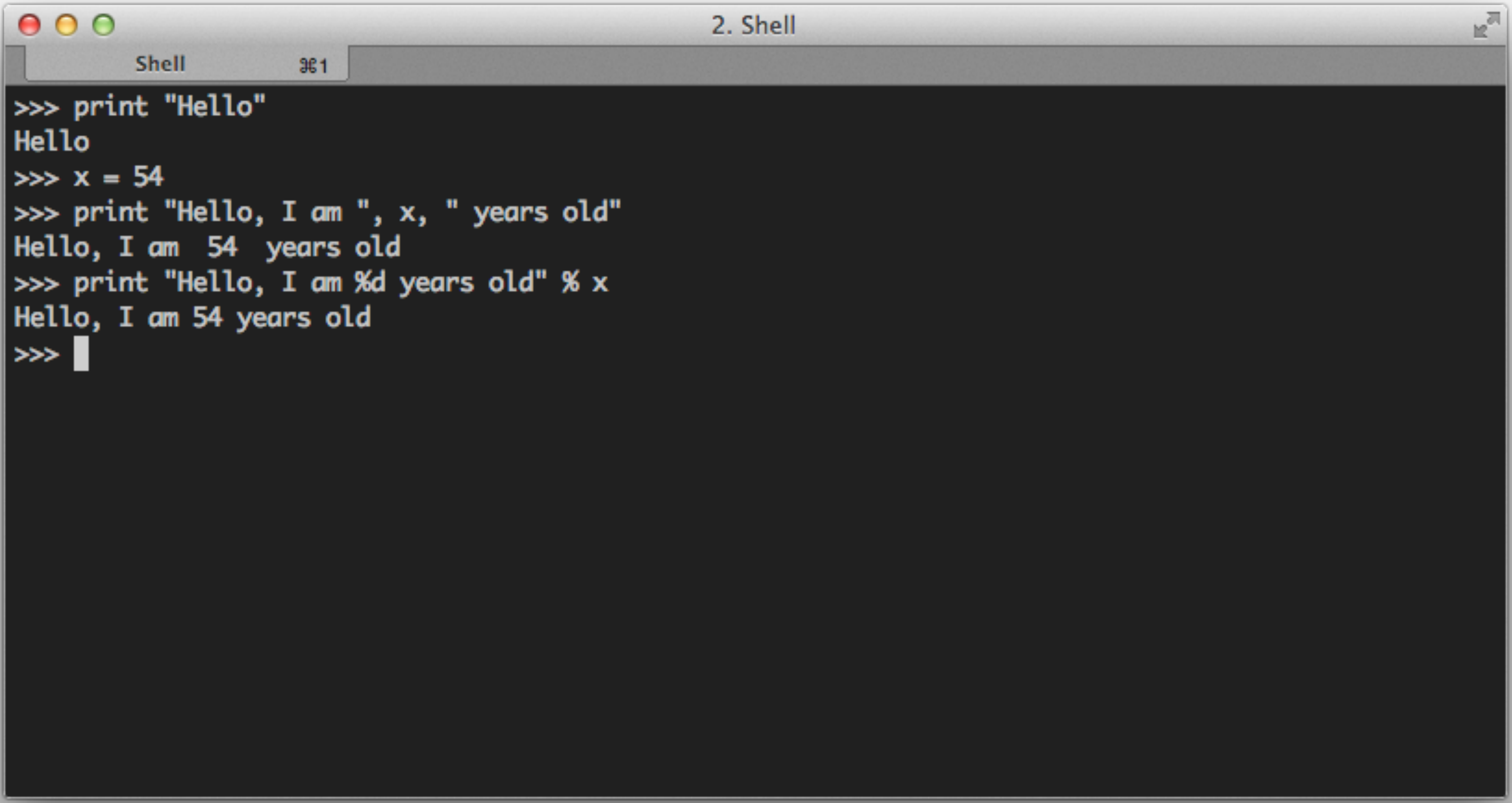
*d***.keys()** **:** a copy of the list of keys in *d*

*d***.values()** **:** a copy of the list of values in *d*

Printing information

`print` statement

Usage:



```
>>> print "Hello"
Hello
>>> x = 54
>>> print "Hello, I am ", x, " years old"
Hello, I am  54  years old
>>> print "Hello, I am %d years old" % x
Hello, I am 54 years old
>>> 
```

`if` conditionals

```
if conditional1:
    statement_1
    ...
    statement_n
elif conditional2:
    statements
else:
    statements
```

The code block of if, elif and else, indicated by indentation and a colon (:)

Indentation level must be the same in the whole block

`for` statement

```
for name in iterable:
    statement_1
    statement_2
    ….
    statement_n
```

`iterable` is any object that can be iterated: lists, tuples, dictionaries

- ➡ name takes the values in the lists in order
- ➡ in dictionaries, in takes the values of the keys, in no particular order
  - ❖ For dictionaries
    - `for (key, value) in d.iteritems()` : iterate over keys and values

`range([start,]stop[,step])`: make a list of integers in the progressing by step each time

Useful for defining traditional for loops:

`for i in range(10)` : i takes the values 0,…,9

Alternatively, use xrange(n) for very large n: uses less memory

while loops

```
while conditional:
    statement1
    statement2
    ...
    statementn
```

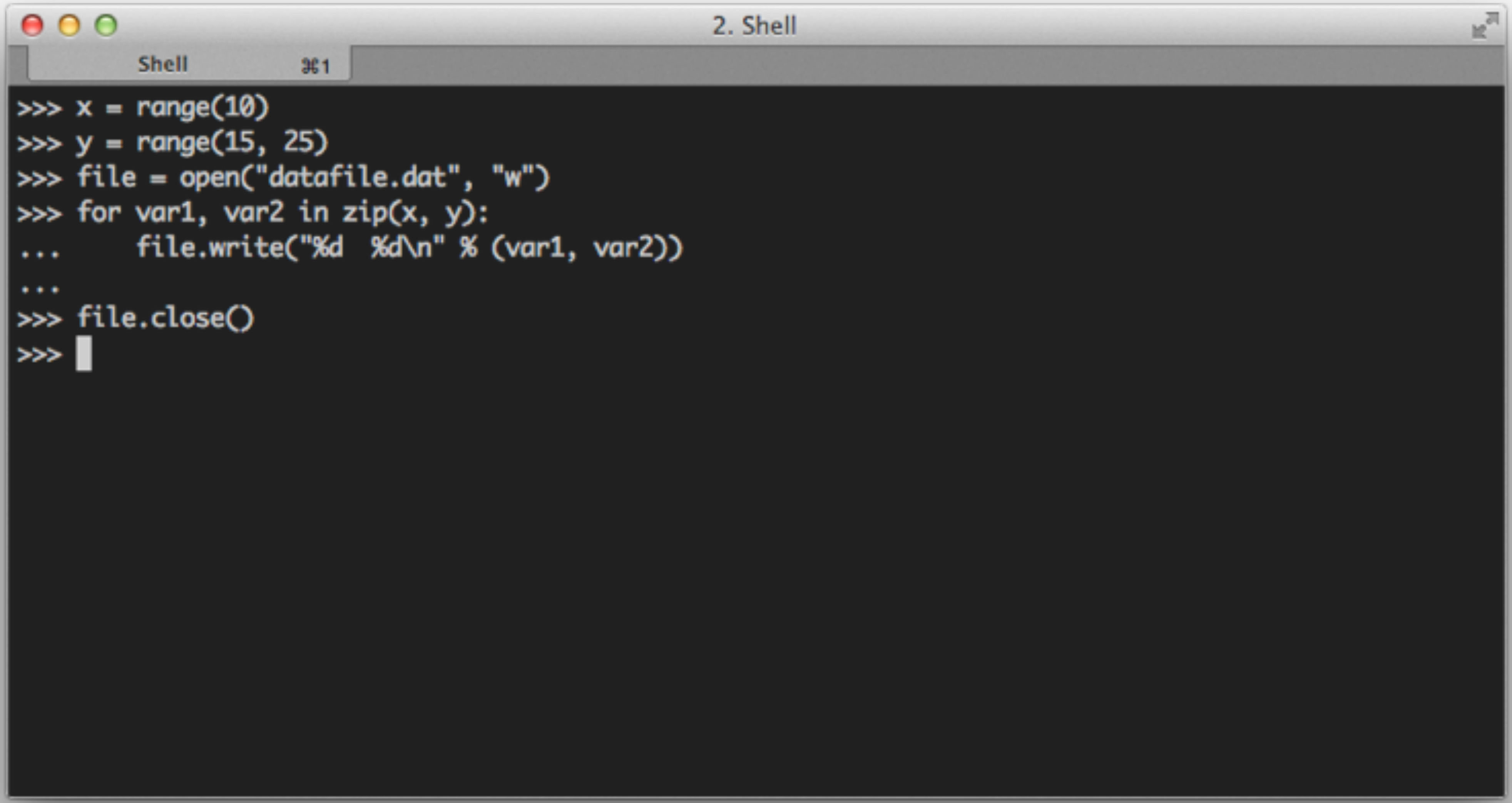
Repeats the block while the conditional is true

`while True:` Repeats the block of code forever
We must exit the loop with a break

Writing a file

Assume we have two lists of numbers, corresponding to a variable and the result to apply a function, x and y

To store them in a file:



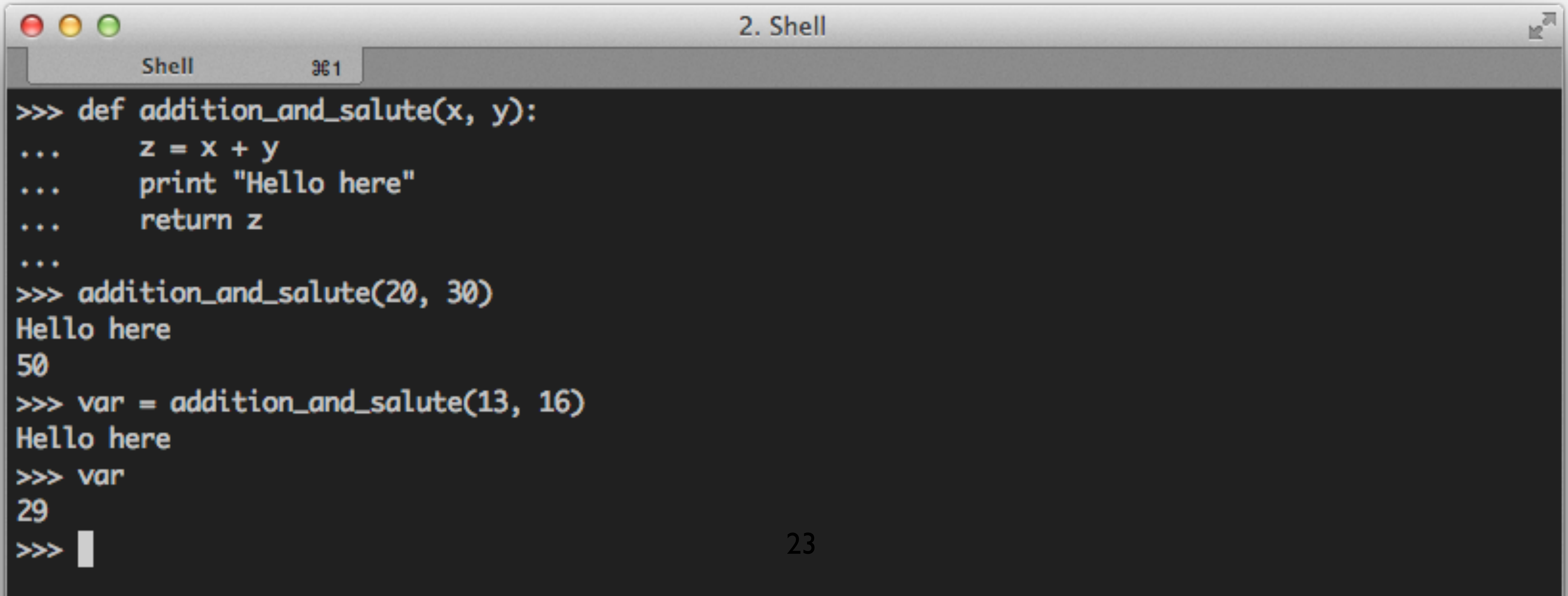
```
>>> x = range(10)
>>> y = range(15, 25)
>>> file = open("datafile.dat", "w")
>>> for var1, var2 in zip(x, y):
...     file.write("%d  %d\n" % (var1, var2))
...
>>> file.close()
>>> 
```

Functions

```
def name([arg1, arg2, ...]):
     statement1
     ...
     statementn
     return [expression]
```

Performs some operations, possible using the optional arguments, and returns a value.

The return value can be any object: number, string, list, dictionary, or other function

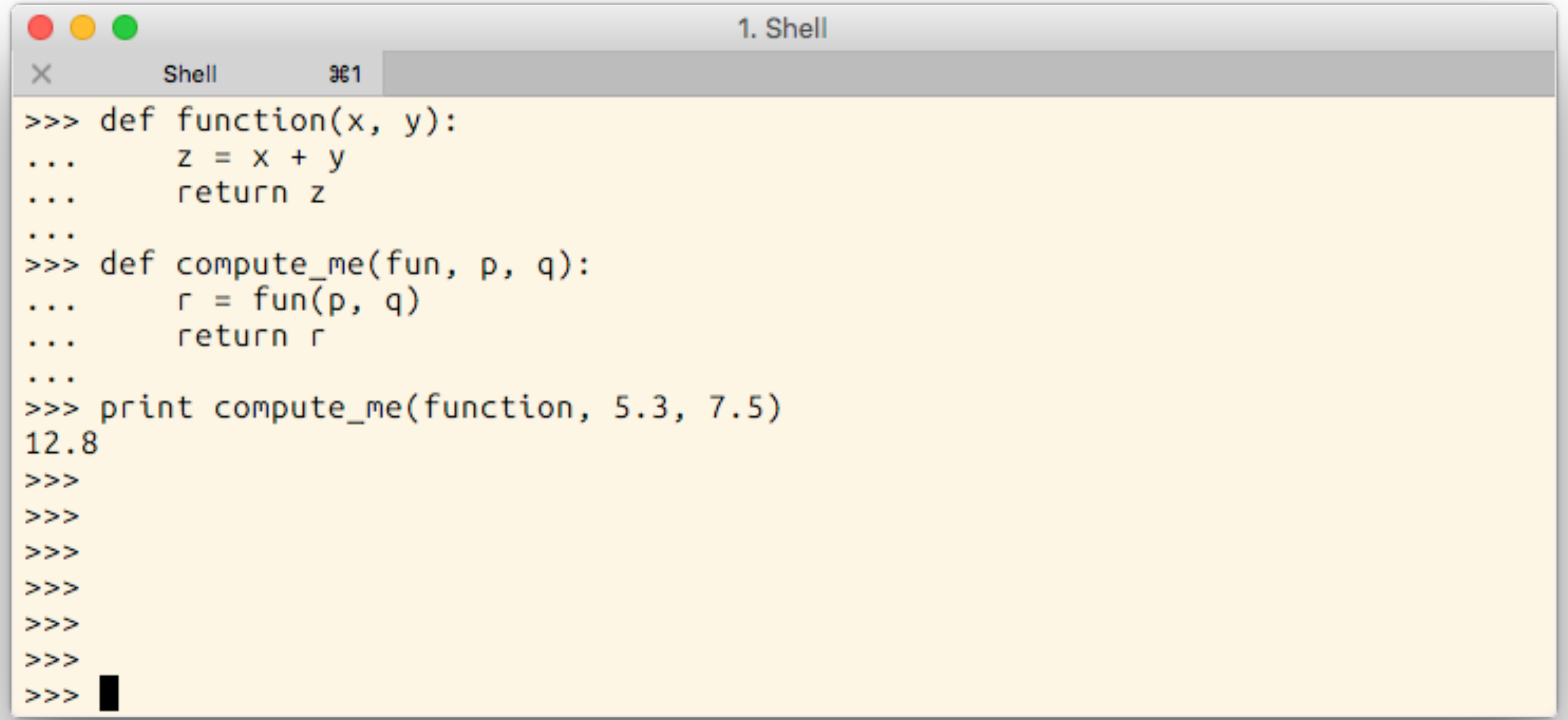


```
>>> def addition_and_salute(x, y):
...     z = x + y
...     print "Hello here"
...     return z
...
>>> addition_and_salute(20, 30)
Hello here
50
>>> var = addition_and_salute(13, 16)
Hello here
>>> var
29
>>> 
```

Functions

Since functions are objects, we can use a function as the argument of another function

Extremely powerful feature



```
>>> def function(x, y):
...     z = x + y
...     return z
...
>>> def compute_me(fun, p, q):
...     r = fun(p, q)
...     return r
...
>>> print compute_me(function, 5.3, 7.5)
12.8
>>>
>>>
>>>
>>>
>>>
>>>
>>> 
```

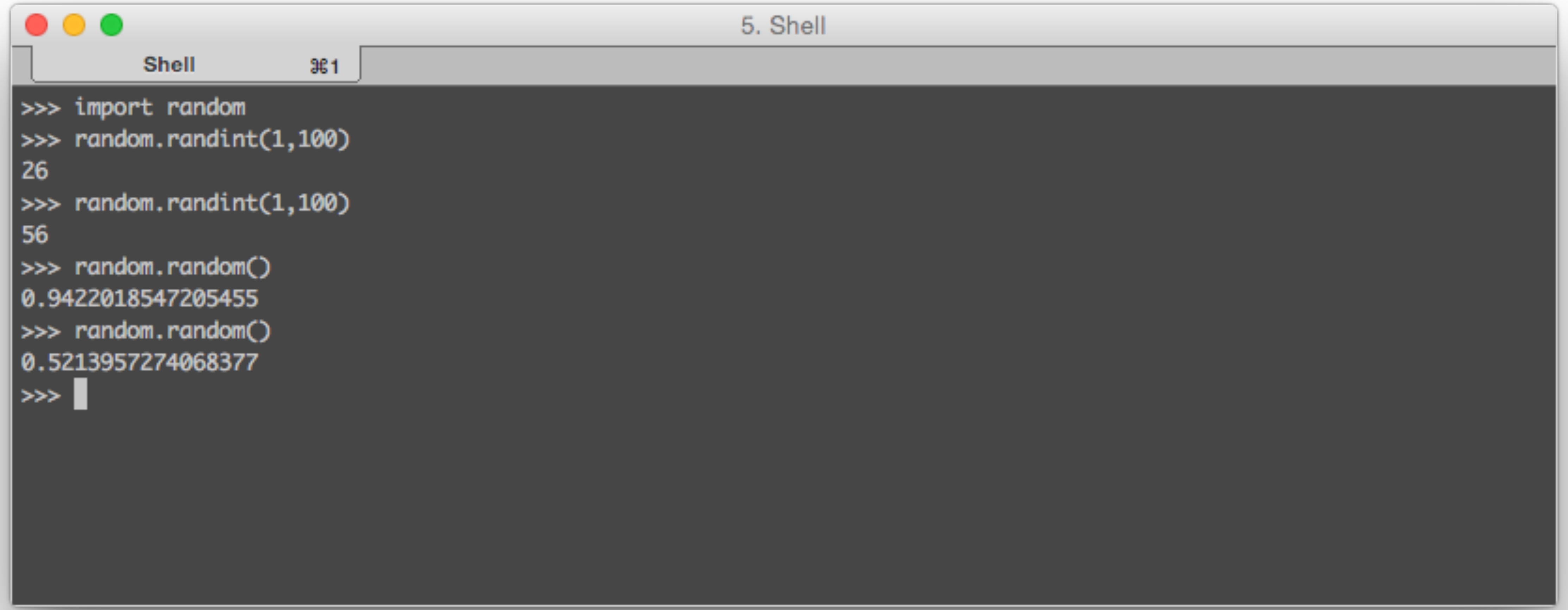
Adding modules

Files containing definitions, variables and functions in Python, that can be reused

There are many modules in the Python distribution that you can import to add increased functionally (batteries included), plus third party libraries

You can use functions in a module using an infix dot notation

Ex.: `random` module includes a Mersenne-Twister random number generator



```
>>> import random
>>> random.randint(1,100)
26
>>> random.randint(1,100)
56
>>> random.random()
0.9422018547205455
>>> random.random()
0.5213957274068377
>>> 
```

Executing a program (script)

Interactive sessions are useful to learn, perform simple calculations or experiments

More complex programs can be written in a file, using any text editor

Python programs must have the extension `.py`

Execute them using : `python my_file.py`

Python workflow and programming tips

Python workflow and programming tips

A possible workflow for python is to use the interactive session for experimentation and a text editor to write the programs that are later run on the interpreter

This a basic approach, common in other compiled languages

As we will see, there are far better approaches in programming with Python

➡Python notebook

Programming tips: Do not reinvent the wheel

Many problems in scientific computing involve tasks that are common in the field

➡ Generating random numbers

➡ Solving equations (algebraic or differential)

➡ Eigenvalues and eigenvectors

➡ Special functions (Bessel, etc)

➡ Optimization (finding maxima or minima)

➡ …

Many of this tasks have been considered and solved by other people, in much smarter and efficient ways you can think of

So, do not reinvent the wheel, google for it instead:

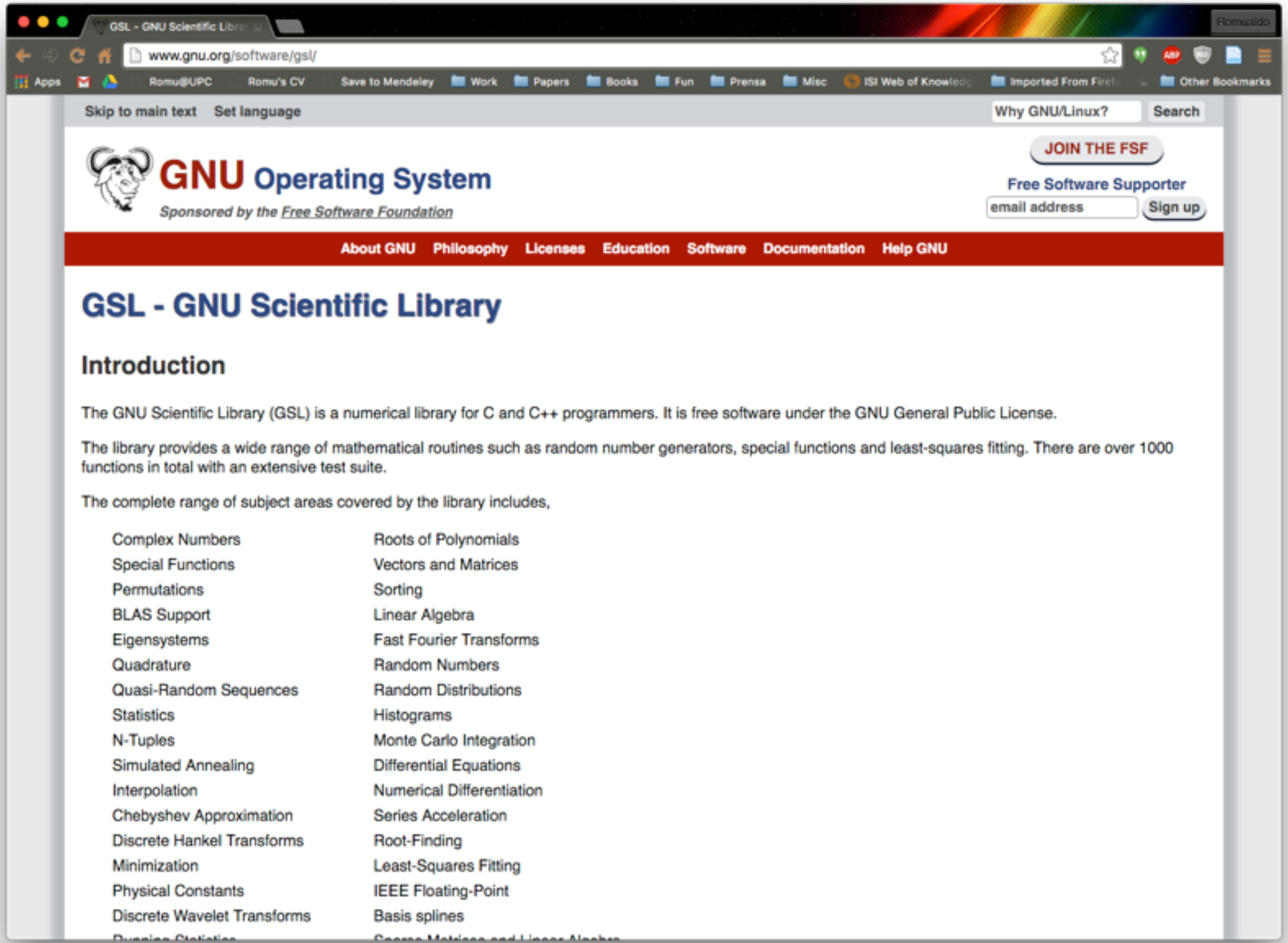
➡ Libraries in Python

➡ Gnu Scientific Library for C/Fortran (General)

➡ BLAS, LAPACK (Linear algebra)

Example

GNU Scientific Library



GSL - GNU Scientific Library

Introduction

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

The complete range of subject areas covered by the library includes,

Complex Numbers	Roots of Polynomials
Special Functions	Vectors and Matrices
Permutations	Sorting
BLAS Support	Linear Algebra
Eigensystems	Fast Fourier Transforms
Quadrature	Random Numbers
Quasi-Random Sequences	Random Distributions
Statistics	Histograms
N-Tuples	Monte Carlo Integration
Simulated Annealing	Differential Equations
Interpolation	Numerical Differentiation
Chebyshev Approximation	Series Acceleration
Discrete Hankel Transforms	Root-Finding
Minimization	Least-Squares Fitting
Physical Constants	IEEE Floating-Point
Discrete Wavelet Transforms	Basis splines

Programming tips: Make your code readable

You are not writing to for the computer, you are writing for other people (specially yourself in a few week's time)

- ➡Indent your code properly to help identify structures blocks in the code
  - ❖Write different blocks at different levels of indentation
  - ❖Python forces to use indentation to declare blocks
  - ❖Other languages (C/Fortran) are inmune to white space: Use indentation for clarity
- ➡Use an text editor with syntax highlighting
  - ❖Different colors for different keywords in the code
  - ❖Helps again in identifying the different parts that compose the code

Example

```
! compute number pi
! using montecarlo a method
program monte_pi
implicit none

integer niter,i,j
integer seed
real*4 count
real *8 x,y,pi(100),z
real*8 rand

seed = 35791246
call srand (seed)
do j= 1,100
print *, j
niter = niter+100
count =0
do i=1,niter
x=rand()
y=rand()
z= x*x +y*y
if (z .le. 1) then
count =count+1
end if
end do
pi(j)= count/niter*4.
print *, "Iter ", niter
print *, "Value ", pi(j)
end do
end program monte_pi
```

Example

```fortran
! compute number pi
! using montecarlo a method
program monte_pi
implicit none

integer niter,i,j
integer seed
real*4 count
real *8 x,y,pi(100),z
real*8 rand

seed = 35791246
call srand (seed)
do j= 1,100
print *, j
niter = niter+100
count =0
do i=1,niter
x=rand()
y=rand()
z= x*x +y*y
if (z .le. 1) then
count =count+1
end if
end do
pi(j)= count/niter*4.
print *, "Iter ", niter
print *, "Value ", pi(j)
end do
end program monte_pi
```

```fortran
! compute number pi
! using montecarlo a method
program monte_pi
  implicit none

  integer niter,i,j
  integer seed
  real*4 count
  real *8 x,y,pi(100),z
  real*8 rand

  seed = 35791246
  call srand (seed)
  do j= 1,100
     print *, j
     niter = niter+100
     count =0
     do i=1,niter
        x=rand()
        y=rand()
        z= x*x +y*y
        if (z .le. 1) then
           count =count+1
        end if
     end do
     pi(j)= count/niter*4.
     print *, "Iter ", niter
     print *, "Value ", pi(j)
  end do
end program monte_pi
```

Example

```fortran
! compute number pi
! using montecarlo a method
program monte_pi
implicit none

integer niter,i,j
integer seed
real*4 count
real *8 x,y,pi(100),z
real*8 rand

seed = 35791246
call srand (seed)
do j= 1,100
print *, j
niter = niter+100
count =0
do i=1,niter
x=rand()
y=rand()
z= x*x +y*y
if (z .le. 1) then
count =count+1
end if
end do
pi(j)= count/niter*4.
print *, "Iter ", niter
print *, "Value ", pi(j)
end do
end program monte_pi
```

```fortran
! compute number pi
! using montecarlo a method
program monte_pi
  implicit none

  integer niter,i,j
  integer seed
  real*4 count
  real *8 x,y,pi(100),z
  real*8 rand

  seed = 35791246
  call srand (seed)
  do j= 1,100
     print *, j
     niter = niter+100
     count =0
     do i=1,niter
        x=rand()
        y=rand()
        z= x*x +y*y
        if (z .le. 1) then
           count =count+1
        end if
     end do
     pi(j)= count/niter*4.
     print *, "Iter ", niter
     print *, "Value ", pi(j)
  end do
end program monte_pi
```

```fortran
! compute number pi
! using montecarlo a method
program monte_pi
  implicit none

  integer niter,i,j
  integer seed
  real*4 count
  real *8 x,y,pi(100),z
  real*8 rand

  seed = 35791246
  call srand (seed)
  do j= 1,100
     print *, j
     niter = niter+100
     count =0
     do i=1,niter
        x=rand()
        y=rand()
        z= x*x +y*y
        if (z .le. 1) then
           count =count+1
        end if
     end do
     pi(j)= count/niter*4.
     print *, "Iter␣", niter
     print *, "Value␣", pi(j)
  end do
end program monte_pi
```

Programming tips: Use meaningful variables

- ➡Avoid single letter names, unless to be used in loops
- ➡Avoid reusing names (in Python)
- ➡Avoid names that vary only by case or punctuation
- ➡Avoid vague names: data, input, output, do_stuff(), process_files(), params, object …

Use instead descriptive names which are long enough to described what is the value stored in the variable

You can use

- ➡Camel Case: EnergyPerUnit
- ➡Snake Case: energy_per_unit

Beware of Fortran, it is not case sensitive

Incidentally: Avoid magic numbers!!

- ➡Use variables instead (with capital letters, for example)

Choose wisely your text editor

Many good code editors are available for all platforms, supporting facilities to help coding. Do not stick with one that does not support

- ➡Syntax highlighting
- ➡Auto-indentation
- ➡Code completion [extra, extremely very helpful]
  - ❖Do not be afraid of long variable names: Use an editor with code completion!

Viable multiplatform code editors:

- ➡vim [free, hard to master]
- ➡emacs [free, hard to master, infinitely costumizable]
- ➡Light Table http://www.lighttable.com/ [free, similar in spirit to vim]
- ➡Sublime Text http://www.sublimetext.com/ [not free but usable, quite easy]
- ➡Atom https://atom.io/ [free, similar to Sublime Text]

Programming tips: DRY

Don’t repeat yourself

➡ If you see that a piece of code that keeps popping up in the same program, encapsulate it inside a function

Benefits:

➡ Limits the number of bugs: Less code, less bugs

➡ Allows to reuse code

  ❖ A useful function can be extracted from the code and reused on a different program

Example

```
# Some dynamical simulation in which you look for
# nearest neighbors in a 1d lattice of size L

# Some code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)

# Some more code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)
```

Example

```
# Some dynamical simulation in which you look for
# nearest neighbors in a 1d lattice of size L

# Some code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)

# Some more code

# look for nearest neighbors
nearest = []
# left
x1 = x -1
if x1 == -1:
    x1 = L - 1
    nearest.append(x1)
# right
x1 = x + 1
if x1 == L:
    x1 = 0
    nearest.append(x1)
```

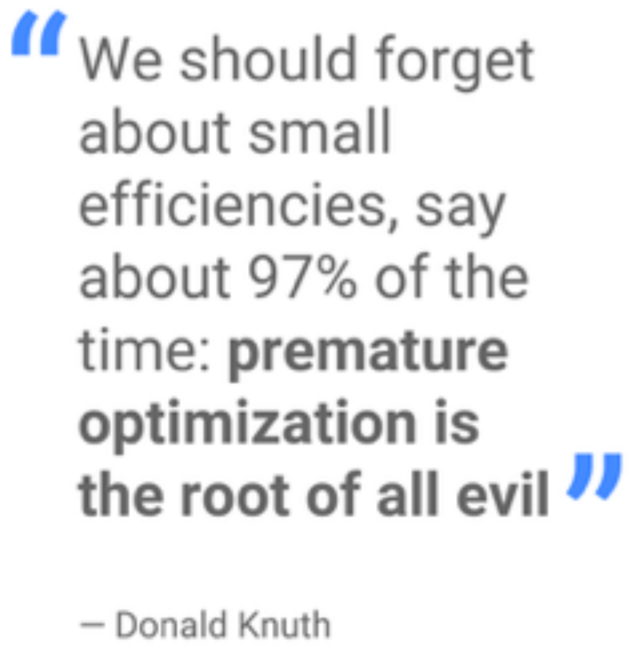
```
# Same code, using a function

def nn_pbc(x, L):
    nearest = []
    x1 = x -1
    if x1 == -1:
        x1 = L - 1
    nearest.append(x1)

    # right
    x1 = x + 1
    if x1 == size:
        x1 = 0
    nearest.append(x1)

    return(nearest)

# Some code

nearest = nn_pbc(x, L)

# Some code

nearest = nn_pbc(x, L)
```

Programming tips: KISS

Keep it simple, stupid

Try to start with the simplest implementation

- ➡ Go only for more complex things when you need it, for example, to speed up a particularly slow part of the code
- ➡ Do so only after benchmarking your code to identify bottlenecks
  - ❖ Do not spend a lot of time optimizing things from 1 sec to 0.01 secs



“We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**”

— Donald Knuth

Programming tips: Comment your code

Comment your code adequately

- ➡Consider you can be reading it in a few months: Make it easy for yourself to understand what you wrote

Avoid Obvious Comments: Don't write what code is doing

- ➡Some comments are better replaced by a good variable or function name

Avoid extra-commenting

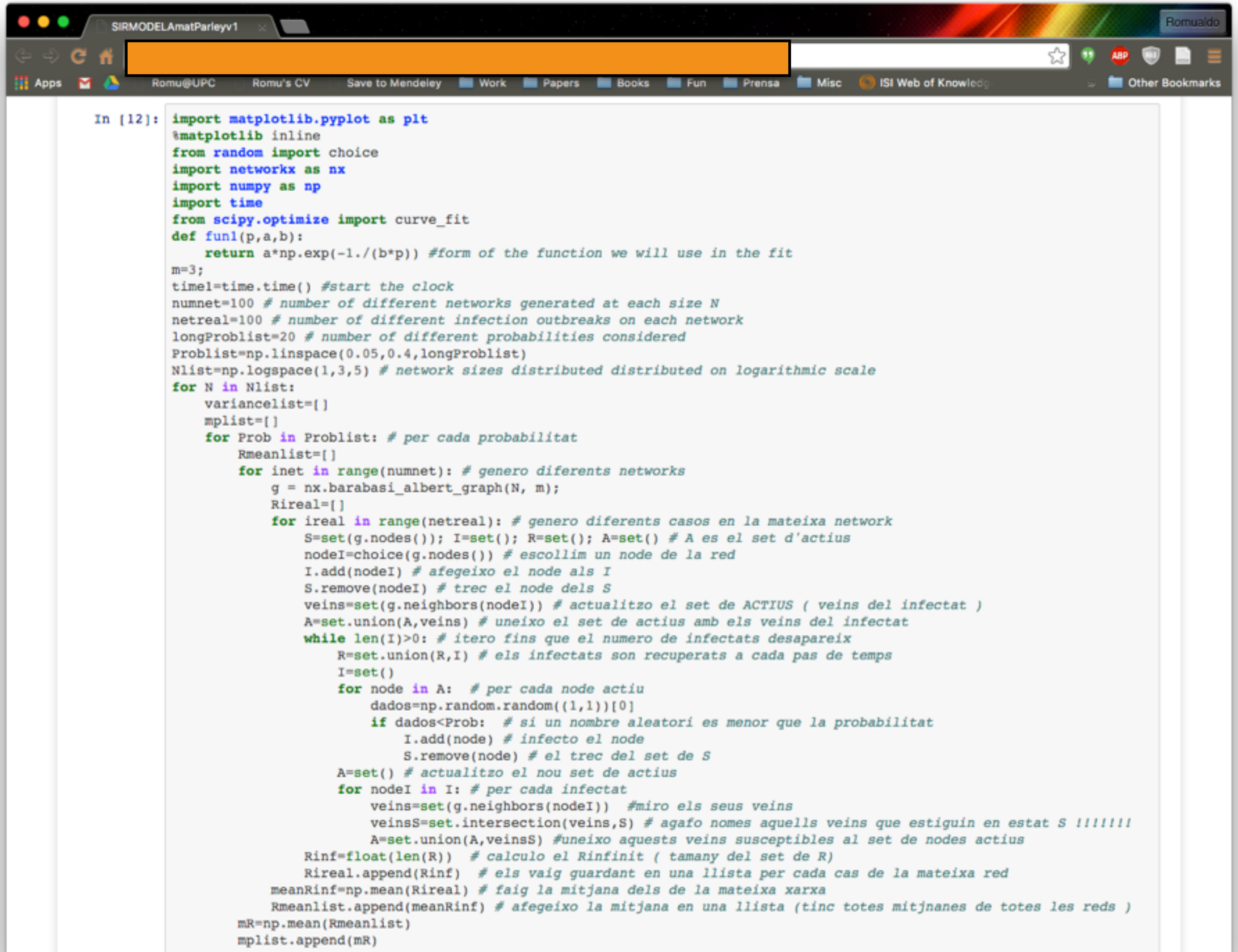
- ➡More is not always better

Group the blocks of code using white space and comments

Programming technique:

- ➡Comment before coding
  - ❖Think first about what you want to code, and implement it later

Example



```
In [12]: import matplotlib.pyplot as plt
         %matplotlib inline
         from random import choice
         import networkx as nx
         import numpy as np
         import time
         from scipy.optimize import curve_fit
         def fun1(p,a,b):
             return a*np.exp(-1./(b*p)) #form of the function we will use in the fit
         m=3;
         time1=time.time() #start the clock
         numnet=100 # number of different networks generated at each size N
         netreal=100 # number of different infection outbreaks on each network
         longProblist=20 # number of different probabilities considered
         Problist=np.linspace(0.05,0.4,longProblist)
         Nlist=np.logspace(1,3,5) # network sizes distributed distributed on logarithmic scale
         for N in Nlist:
             variancelist=[]
             mplist=[]
             for Prob in Problist: # per cada probabilitat
                 Rmeanlist=[]
                 for inet in range(numnet): # genero diferents networks
                     g = nx.barabasi_albert_graph(N, m);
                     Rireal=[]
                     for ireal in range(netreal): # genero diferents casos en la mateixa network
                         S=set(g.nodes()); I=set(); R=set(); A=set() # A es el set d'actius
                         nodeI=choice(g.nodes()) # escollim un node de la red
                         I.add(nodeI) # afegeixo el node als I
                         S.remove(nodeI) # trec el node dels S
                         veins=set(g.neighbors(nodeI)) # actualitzo el set de ACTIUS ( veins del infectat )
                         A=set.union(A,veins) # uneixo el set de actius amb els veins del infectat
                         while len(I)>0: # itero fins que el numero de infectats desapareix
                             R=set.union(R,I) # els infectats son recuperats a cada pas de temps
                             I=set()
                             for node in A:  # per cada node actiu
                                 dados=np.random.random((1,1))[0]
                                 if dados<Prob:  # si un nombre aleatori es menor que la probabilitat
                                     I.add(node) # infecto el node
                                     S.remove(node) # el trec del set de S
                             A=set() # actualitzo el nou set de actius
                             for nodeI in I: # per cada infectat
                                 veins=set(g.neighbors(nodeI))  #miro els seus veins
                                 veinsS=set.intersection(veins,S) # agafo nomes aquells veins que estiguin en estat S !!!!!!!
                                 A=set.union(A,veinsS) #uneixo aquests veins susceptibles al set de nodes actius
                         Rinf=float(len(R))  # calculo el Rinfinit ( tamany del set de R)
                         Rireal.append(Rinf)  # els vaig guardant en una llista per cada cas de la mateixa red
                     meanRinf=np.mean(Rireal) # faig la mitjana dels de la mateixa xarxa
                     Rmeanlist.append(meanRinf) # afegeixo la mitjana en una llista (tinc totes mitjnanes de totes les reds )
                 mR=np.mean(Rmeanlist)
                 mplist.append(mR)
```

Programming tips: Program modularization and refactoring

When programs are small, it makes sense to store them in a single file

For very large programs, it makes sense to split them into separate files (modules)

Modules can contain:

- ➡Variables, declarations and constants that are used in all the program, allowing to modify them easily at a single point
- ➡Groups of related functions

Programming tips: Program modularization and refactoring

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent interchangeable pieces (modules) that contain everything to perform only one aspect of the desired functionality

Modules do not need to know about how something else is done

- ❖ Encapsulation

Modularity, mixed with the use of functions, allows you to:

- ➡ More easily solve a problem by breaking it down into elementary pieces
- ➡ Fix problems and find bugs in specific sections
- ➡ Reuse individual modules (useful functions) on other programs

Programming tips: Use version control

Programming, specially in science, is a complex business.

You start with one version of the program, and then proceed to manyfold modifications, advancing one step forward and one backwards, usualy in an experimental way

Using a version control system (git, github) allows to:

➡Revert files back to a previous state, in case of errors

➡Revert an entire project back to a previous state

➡Compare changes over time

➡Know who and when last modified something that does not work

➡Recover if you screw things up severely

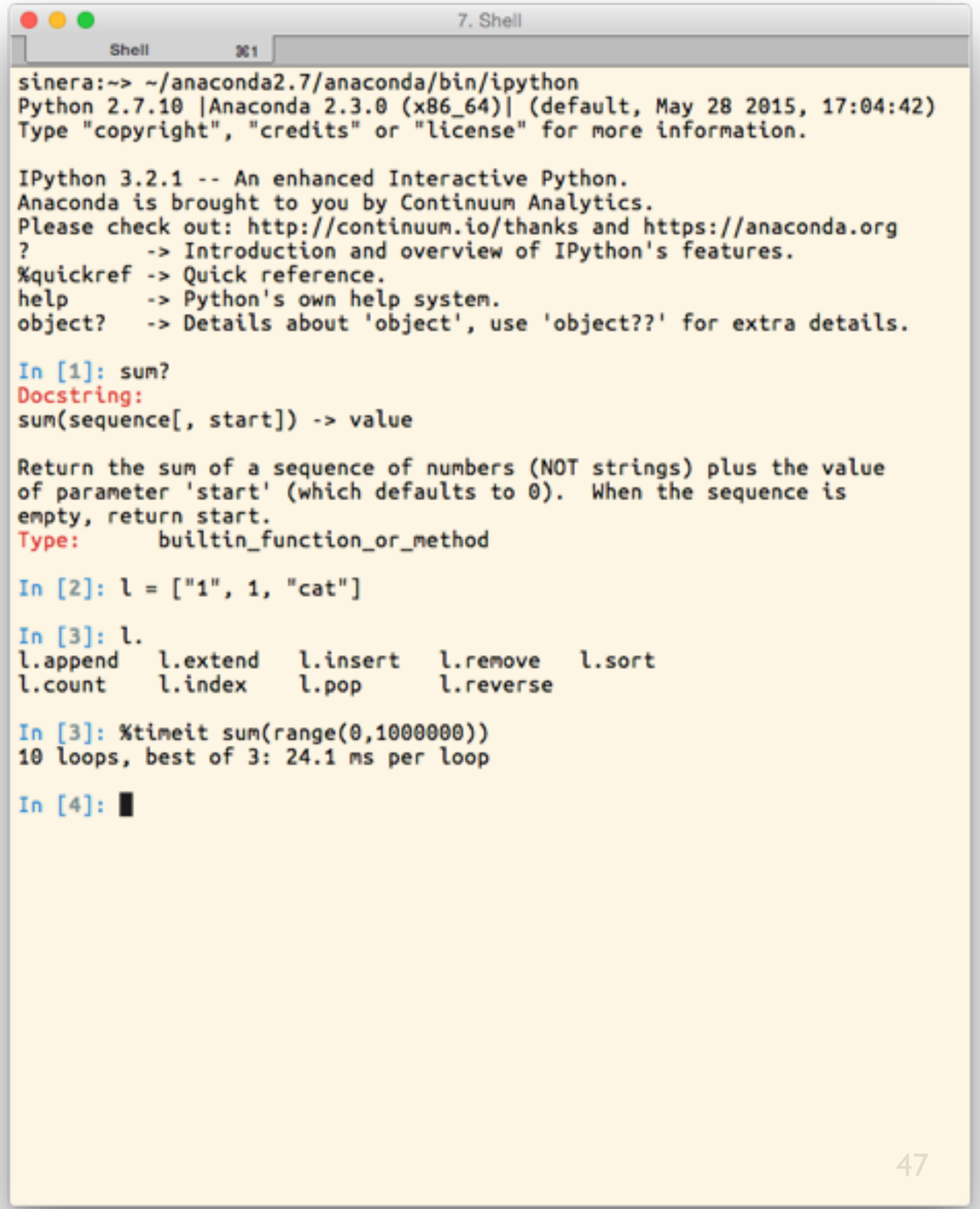
➡Work on different, parallel branches of the code, implementing and experimenting new ideas

➡Manage work in the code by a team of people

Python interfaces for scientific programming

Ipython

Ipython is an alternative shell for interactive python with many enhancements which are particularly useful for scientific programming and experimentation

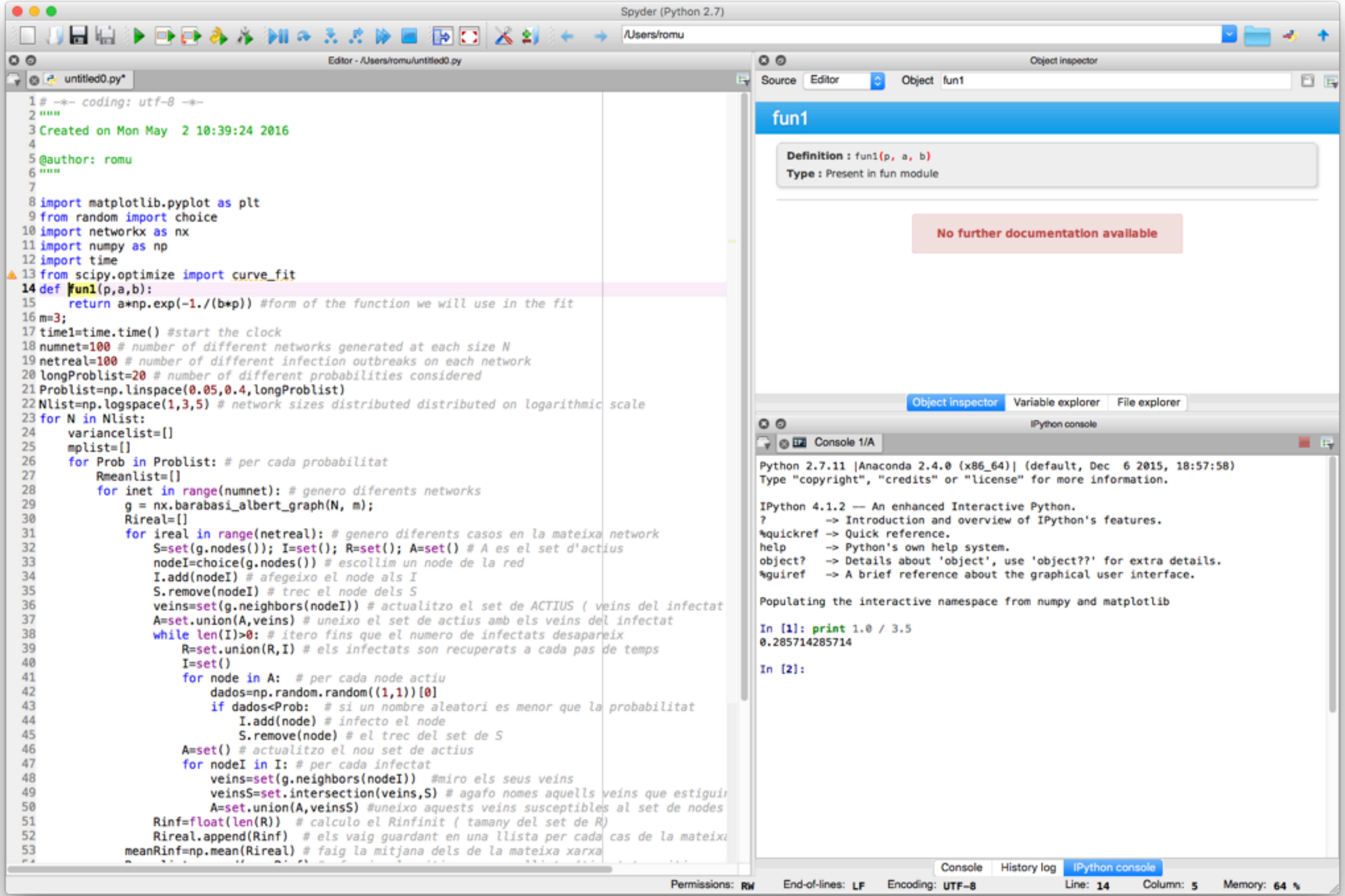


```
sinera:~> ~/anaconda2.7/anaconda/bin/ipython
Python 2.7.10 |Anaconda 2.3.0 (x86_64)| (default, May 28 2015, 17:04:42)
Type "copyright", "credits" or "license" for more information.

IPython 3.2.1 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: sum?
Docstring:
sum(sequence[, start]) -> value

Return the sum of a sequence of numbers (NOT strings) plus the value
of parameter 'start' (which defaults to 0).  When the sequence is
empty, return start.
Type:      builtin_function_or_method

In [2]: l = ["1", 1, "cat"]

In [3]: l.
l.append   l.extend   l.insert   l.remove   l.sort
l.count    l.index    l.pop      l.reverse

In [3]: %timeit sum(range(0,1000000))
10 loops, best of 3: 24.1 ms per loop

In [4]: █
```

Ipython main features:

- Command history
- Inline help (with ?)
  - Ex: sum? (Intro)
- Tab completion
  - Ex: Completion of variables of functions
  - Ex: Completion of methods
- Magic commands (starting with %)
  - Ex: `%timeit`
  - Ex: `%run` program.py
  - `%lsmagic`: list of magic commands

Spyder: An IDE for python



```
# -*- coding: utf-8 -*-
"""
Created on Mon May  2 10:39:24 2016

@author: romu
"""

import matplotlib.pyplot as plt
from random import choice
import networkx as nx
import numpy as np
import time
from scipy.optimize import curve_fit
def fun1(p,a,b):
    return a*np.exp(-1./(b*p)) #form of the function we will use in the fit
m=3;
time1=time.time() #start the clock
numnet=100 # number of different networks generated at each size N
netreal=100 # number of different infection outbreaks on each network
longProblist=20 # number of different probabilities considered
Problist=np.linspace(0.05,0.4,longProblist)
Nlist=np.logspace(1,3,5) # network sizes distributed distributed on logarithmic scale
for N in Nlist:
    variancelist=[]
    mplist=[]
    for Prob in Problist: # per cada probabilitat
        Rmeanlist=[]
        for inet in range(numnet): # genero diferents networks
            g = nx.barabasi_albert_graph(N, m);
            Rireal=[]
            for ireal in range(netreal): # genero diferents casos en la mateixa network
                S=set(g.nodes()); I=set(); R=set(); A=set() # A es el set d'actius
                nodeI=choice(g.nodes()) # escollim un node de la red
                I.add(nodeI) # afegeixo el node als I
                S.remove(nodeI) # trec el node dels S
                veins=set(g.neighbors(nodeI)) # actualitzo el set de ACTIUS ( veins del infectat
                A=set.union(A,veins) # uneixo el set de actius amb els veins del infectat
                while len(I)>0: # itero fins que el numero de infectats desapareix
                    R=set.union(R,I) # els infectats son recuperats a cada pas de temps
                    I=set()
                    for node in A:  # per cada node actiu
                        dados=np.random.random((1,1))[0]
                        if dados<Prob:  # si un nombre aleatori es menor que la probabilitat
                            I.add(node) # infecto el node
                            S.remove(node) # el trec del set de S
                    A=set() # actualitzo el nou set de actius
                    for nodeI in I: # per cada infectat
                        veins=set(g.neighbors(nodeI))  #miro els seus veins
                        veinsS=set.intersection(veins,S) # agafo nomes aquells veins que estiguir
                        A=set.union(A,veinsS) #uneixo aquests veins susceptibles al set de nodes
                Rinf=float(len(R))  # calculo el Rinfinit ( tamany del set de R)
                Rireal.append(Rinf)  # els vaig guardant en una llista per cada cas de la mateixa
            meanRinf=np.mean(Rireal) # faig la mitjana dels de la mateixa xarxa
```

fun1

Definition : fun1(p, a, b)
Type : Present in fun module

No further documentation available

```
Python 2.7.11 |Anaconda 2.4.0 (x86_64)| (default, Dec  6 2015, 18:57:58)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

Populating the interactive namespace from numpy and matplotlib

In [1]: print 1.0 / 3.5
0.285714285714

In [2]:
```

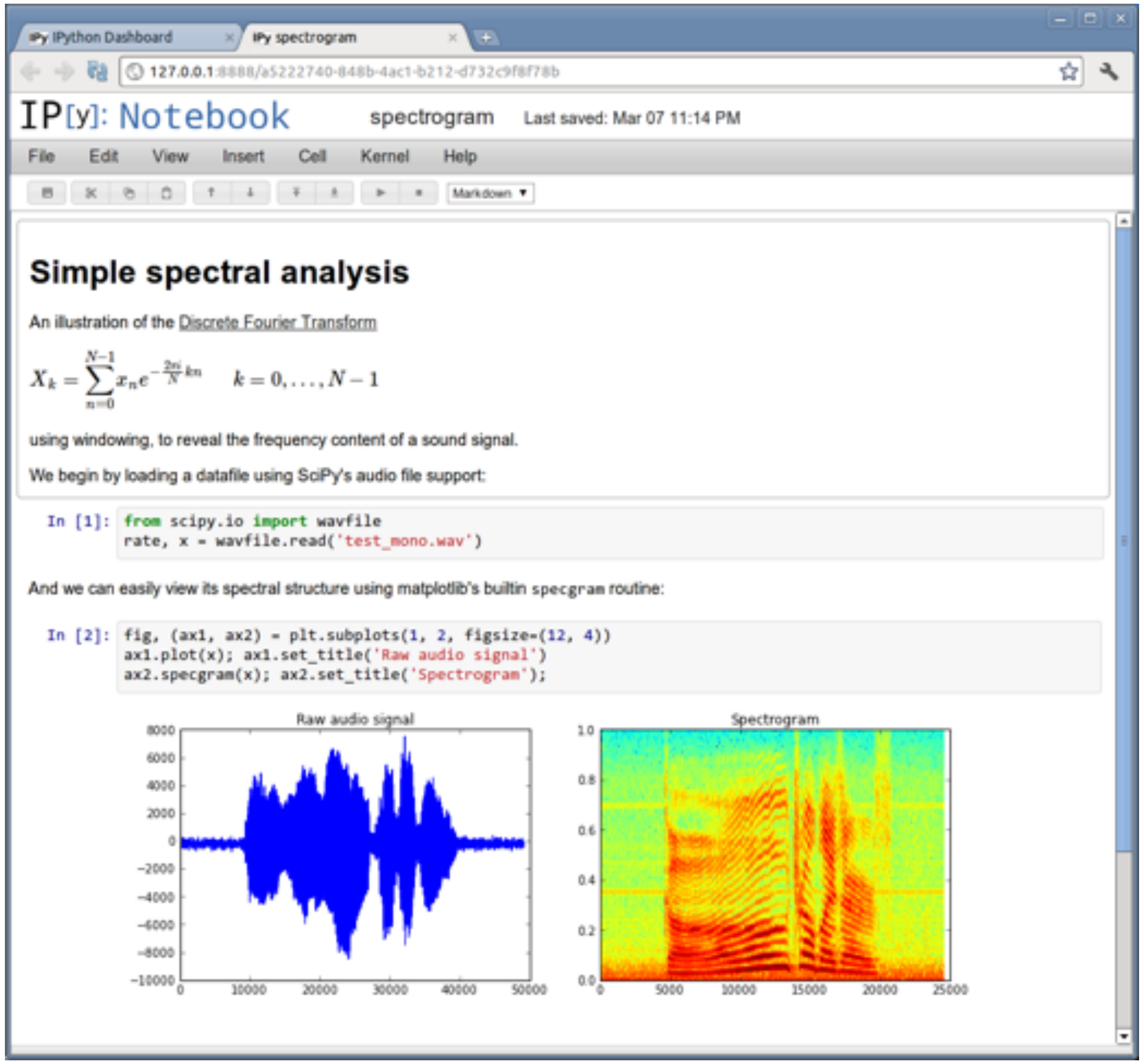
Permissions: RW   End-of-lines: LF   Encoding: UTF-8   Line: 14   Column: 5   Memory: 64 %

Ipython notebook

Interactive python environment that combines execution, text, math, plots and rich multimedia

Execute `jupyter notebook` on a terminal (or from the Anaconda launcher)

Better if Chrome or Firefox are your default web browsers (don't use Explorer)



Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

```
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```