Introducción a PYTHON para cálculo científico

Luis Rández

Universidad de Zaragoza

curso 2023-24

¿Qué es PYTHON?

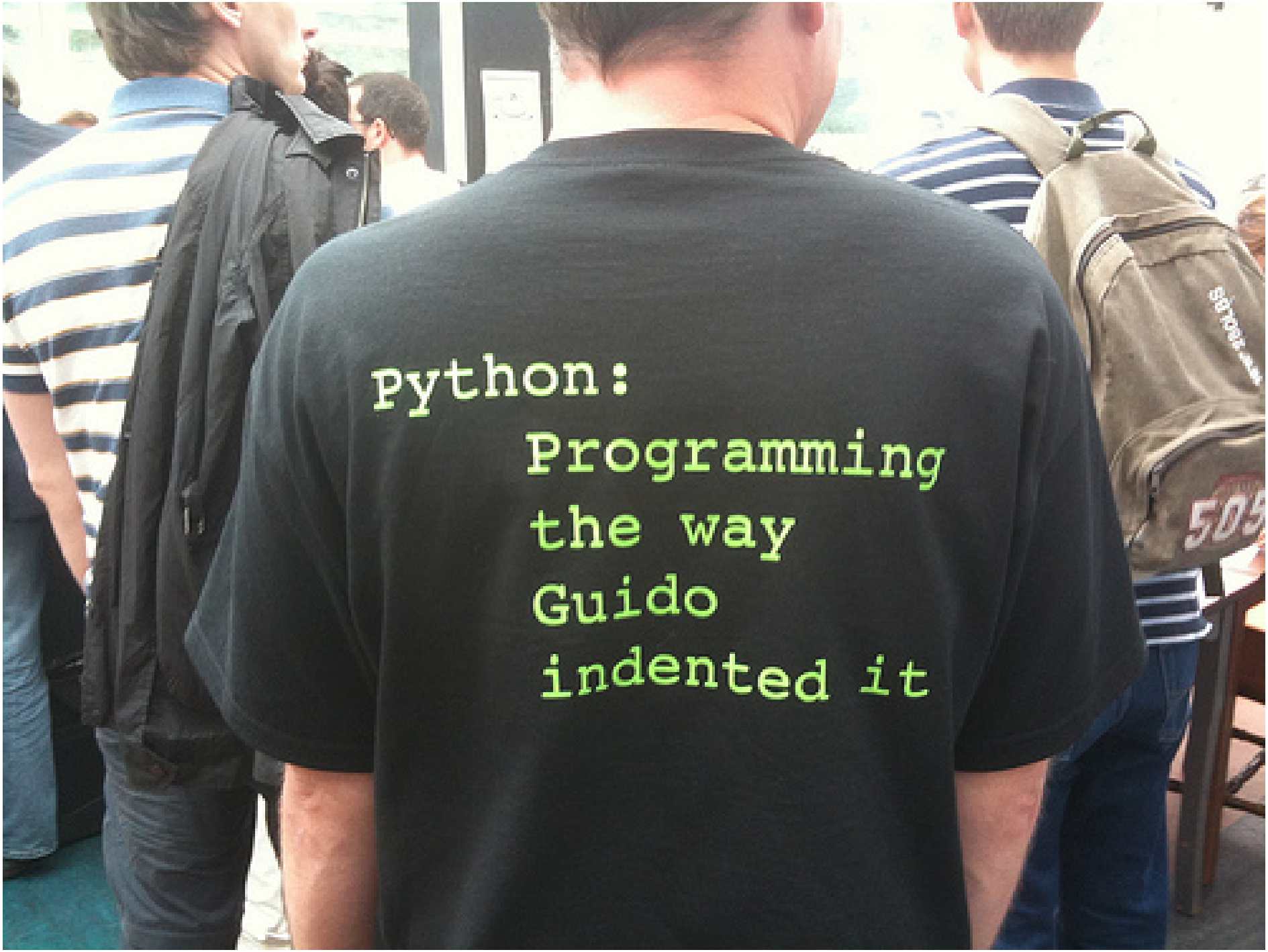
PYTHON es un lenguaje de programación de tipo *script* creado por Guido van Rossum a principios de los años 90, cuyo nombre proviene del grupo «Monty Python». El objetivo es un lenguaje con una sintaxis muy limpia y código legible.

Figura: Los humoristas británicos *Monty Python*

Características de PYTHON

- Lenguaje de propósito general
- Interactivo
- Interpretado: No hay que compilar explicitamente
- Lenguaje de muy alto nivel
- Tipado dinámico: una misma variable puede tomar valores de distinto tipo en distintos momentos
- Multiplataforma: Symbian, Unix, Windows, Android, iOS...
- Conexiones con otros lenguajes (FORTRAN, C, LaTeX, CUDA[a], Excel...)
- Manejo de excepciones
- Orientado a objetos
- Sintaxis clara y sencilla. Mantenimiento fácil
- Código abierto

[a] La tarjeta gráfica debe ser NVIDIA


Python:
Programming
the way
Guido
indented it

¿Por qué PYTHON?

- desarrollo rápido de código
- lenguaje de muy alto nivel
- sintaxis clara y sencilla. Mantenimiento fácil
- gran cantidad de librerías
- lenguaje de propósito general

¿Quién usa PYTHON?

- Google
- Yahoo
- Industrial Light & Magic
- Disney
- NASA
- ⋮

Conjunto de palabras clave de PYTHON

palabras clave de PYTHON

`and`	`as`	`assert`	`break`
`class`	`continue`	`def`	`del`
`elif`	`else`	`except`	`exec`
`finally`	`for`	`from`	`global`
`if`	`import`	`in`	`is`
`lambda`	`not`	`or`	`pass`
`print`	`raise`	`return`	`try`
`while`	`with`	`yield`	

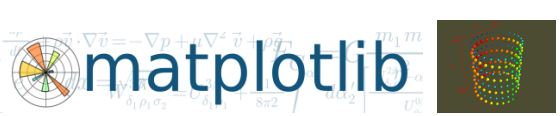
Librerías útiles para cálculo científico en PYTHON

IPYTHON

Intérprete mejorado de PYTHON

Librerías

- **numpy**: proporciona herramientas para la generación y manipulación de `arrays`.
- **scipy**: optimización, Fourier, cuadratura e integración numérica... (depende de NUMPY)
- **matplotlib**: Gráficos 2D y 3D
- **pil**: Python Imaging Library
- **sympy**: cálculo simbólico
- ⋮



Instalar PYTHON + NUMPY + SCIPY + MATPLOTLIB + ...

ANACONDA suministra un conjunto de herramientas para el análisis de datos y su visualización. Esta distribución amplia las características típicas de PYTHON con librerías especializadas para cálculo científico

Quizás la solución más sencilla...

`https://www.anaconda.com`

Otra solución simple...

Google colab `https://colab.research.google.com/` Permite la ejecución de PYTHON *online* sin necesidad de instalar nada en modo local.

NUMPY - Manipulación de matrices multidimensionales

NUMPY es la librería fundamental para cálculo científico con PYTHON.

Contenidos

- Matrices multidimensionales (*arrays*)
- Selección avanzada de secciones de matrices (*slicing*)
- Redimensionado de matrices (*reshape*)

Librerías básicas

- Funciones básicas de álgebra lineal
- Transformadas básicas de Fourier
- Generación de números aleatorios

- NUMPY puede ampliarse con funciones escritas en `C` o `Fortran`.
- Es de código abierto.

SCIPY - Herramientas científicas para PYTHON

SCIPY es una librería que proporciona herramientas científicas para PYTHON y depende de NUMPY.

Módulos de SCIPY

- Estadística
- Optimización
- Cuadratura numérica
- Álgebra lineal
- Transformadas de Fourier
- Procesamiento de señales e imágenes
- Integradores de EDO
- ⋮

- Está apoyado por Enthought Python Distribution
- Es de código abierto.

MATPLOTLIB - Gráficos y dibujos en PYTHON

MATPLOTLIB es una librería de PYTHON para la realización de figuras y gráficos de calidad:

- histogramas
- dibujos de curvas
- curvas de nivel
- campos vectoriales
- superficies
- ⋮

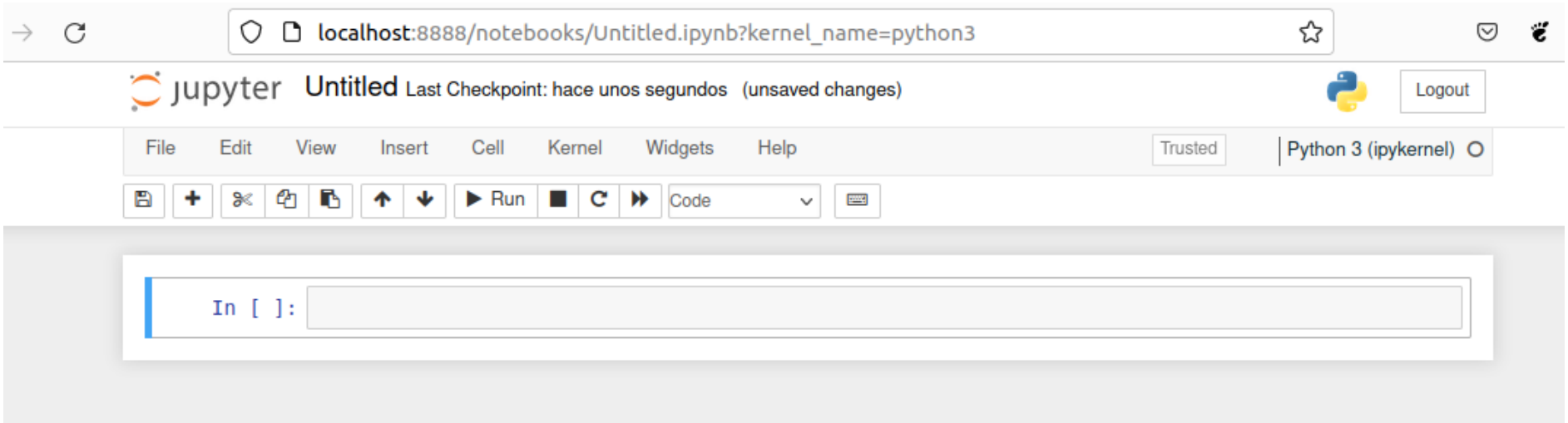
PYLAB es un sistema de librerías cuyo objetivo es transformar PYTHON para cálculo científico en un entorno similar a matlab. Está compuesto por:

NUMPY, SCIPY, MATPLOTLIB, IPYTHON

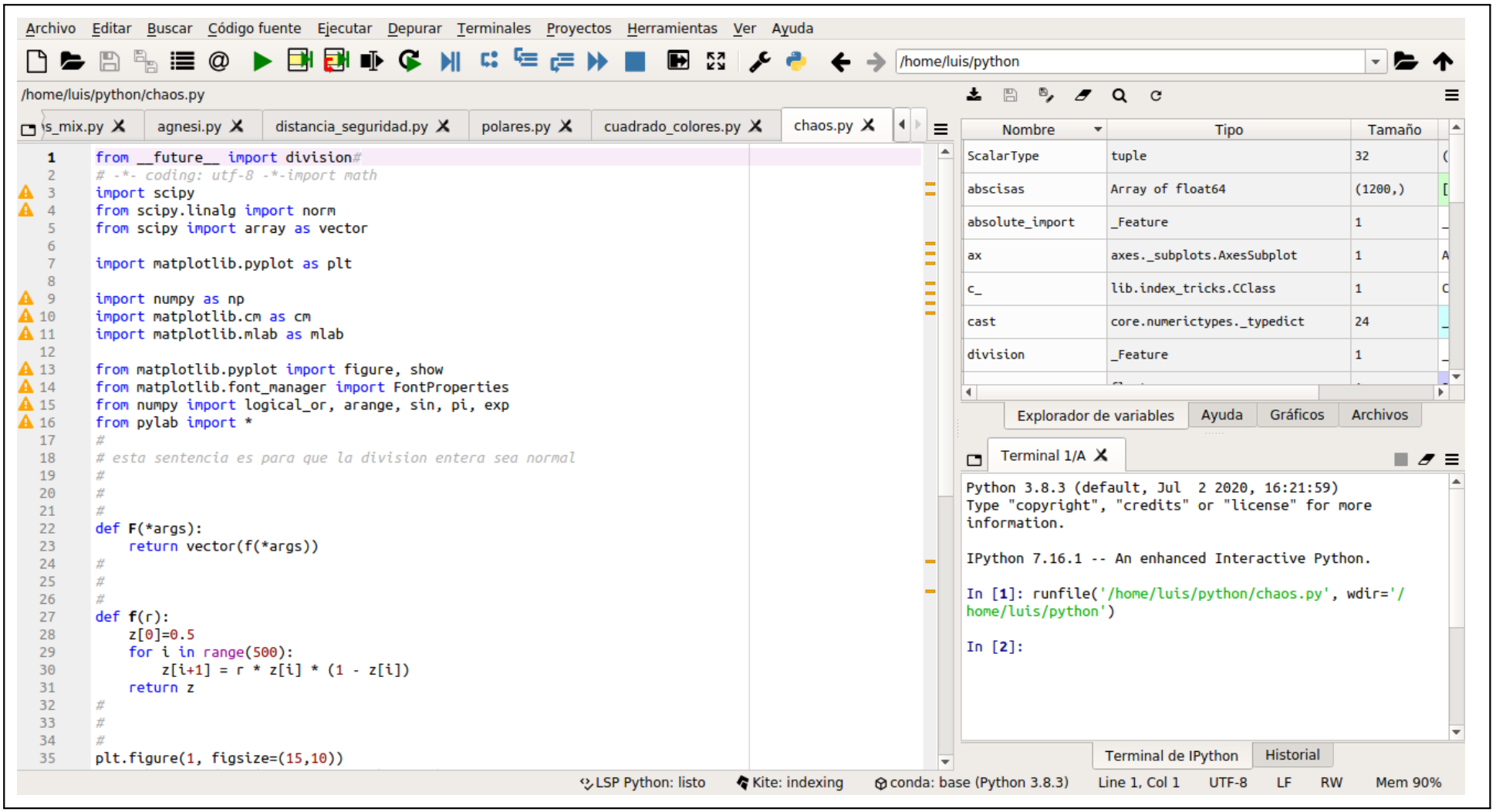
Intérprete mejorado de comandos

```
ubuntu-laptop:$ ipython --pylab
Python 3.7.7 (default, Mar 13 2020, 13:32:22)
Type "copyright", "credits" or "license"  ...

IPython 7.13.0 -- An enhanced Interactive Python. Type '?'...
In [1]:
```

Empleando Jupyter



Entorno integrado Spyder



Módulos

Los módulos son librerías que amplian las funciones y clases de PYTHON para realizar tareas específicas. Se pueden cargar al inicio de varias maneras:

```
In [1]: import numpy as np    # carga numpy
In [2]: help (np)             # ayuda sobre numpy
In [3]: dir(np)               # todas funciones de numpy
In [4]: np.info('topico')     # info de topico numpy
In [5]: np.info('fft')        # info de fft
```

Las librerías se cargan de la forma `from numpy import *` o bien con `import numpy` o `import numpy as np`. En estos dos últimos casos, una función de NUMPY hay que llamarla `numpy.funcion` o `np.funcion` respectivamente.

```
In [1]: import numpy as np  / import numpy
In [2]: print (np.sin(1))   / print (numpy.sin(1))
0.8414709848078965

In [1]: from numpy import *
In [2]: print (sin(1))
```

IPYTHON. Aritmética

```
In [1]: 2**388
Out[1]: 630432099142311667396464641602297820881275828327447146687172694467931548343955369782628260078158650252906047844909056

In [2]: 3/2 # <- comentario
Out[2]: 1.5

In [3]: q, r = divmod(11, 7) # cociente y resto

In [5]: print (q, r)
1 4
In [6]:  1+2  # ERROR en python pero no en ipython
------------------------------------------------------------
Indentation error: unexpected indent  (python)
```

IPYTHON. Aritmética

PYTHON al igual que MATLAB/OCTAVE trabaja en aritmética de coma flotante que usa el estándar IEEE-754.

```
In [1]: from numpy import *
In [2]: a=-inf        # a= -Inf tambien vale
In [3]: print (2**a, inf/inf, inf-inf)
0.0  nan  nan
In [4]: print (sqrt(-1.)) # antes de scipy
RuntimeWarning: invalid value encountered in sqrt
nan
In [5]: from scipy import *
In [6]: print (sqrt(-1.)) # despues de scipy
1j
In [7]: 10.0**400 # en matlab el resultado es inf
OverflowError  Traceback (most recent call last) ...
OverflowError: (34, 'Numerical result out of range')

In [8]: x = 10.0**200; y = x*x; print (y)
inf
```

IPYTHON. Aritmética

```
In [1]: sin(0)/0
Out[1]: nan

In [2]: x=1.0/0
---------------------------------------------------------------
ZeroDivisionError                Traceback (most recent call last)
...
ZeroDivisionError: float division by zero

In [3]: from numpy import *

In [4]: finfo(float).eps # eps en matlab/python 2.**(-52)
Out[4]: 2.22044604925031e-16

In [5]: y=10**1000   # Dara error?
```

Tipos numéricos y lógico

entero	coma flotante	complejo	lógico
`int`	`float`	`complex`	`bool`

```
In [1]: i=2**130        #  tipo entero
In [2]: i=pow(2,130)
In [3]: print(i)        # precision arbitraria en enteros
1361129467683753853853498429727072845824
In [4]: float(_)        # _ equivalente a Ans en matlab
Out[4]: 1.3611294676837539e+39
In [5]: a=2.1           # tipo coma flotante
In [6]: int(a)
Out[6]: 2
In [7]: z=1.5 + 3j      #  z=complex(1.5,3) tipo complejo
In [8]: z=1.5 + 3*1j; print (z)
(1.5+3j)
```

Tipos numéricos y lógico

```
In [1]: real(z), imag(z)
Out[1]: (1.5, 3.0)
In [2]: conjugate(z)
Out[2]: (1.5-3j)

In [3]: abs(z)
Out[3]: 3.3541019662496847
In [4]: z=(3>4) # tipo logico
In [5]: print(z)
False
In [6]: type(z)
Out[6]: bool
```

El tipo de una variable puede cambiar varias veces en el mismo programa.

Más operaciones aritméticas

Además existen en PYTHON notaciones compactas para modificar el valor de una variable.

`+=`	`c += a`	`c = c+a`	suma
`-=`	`c -= a`	`c = c-a`	resta
`*=`	`c *= a`	`c = c*a`	producto
`/=`	`c /= a`	`c = c/a`	división
`%=`	`c %= a`	`c = c%a`	resto de la división c/a
`**=`	`c **= a`	`c = c**a`	potencia
`//=`	`c //= a`	`c = c//a`	división entera

Operaciones lógicas y comparaciones

Operaciones lógicas

x or y	x and y	not x

Comparaciones

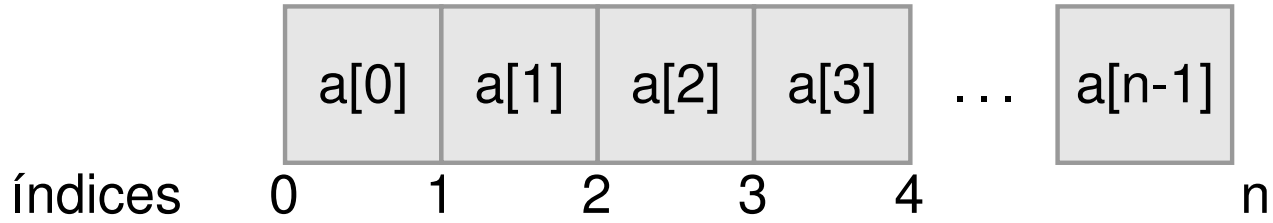
igualdad	menor o igual	mayor o igual	distinto
==	<=	>=	!=

```
In [1]: (3<4) or (1<0)
Out[1]: True
In [2]: not (3<4)
Out[2]: False
In [3]: 4 == 9
Out[3]: False
In [4]: 4 != 9
Out[4]: True
```

Colecciones de tipos. Listas 

Una lista es una colección de objetos que pueden ser de varios tipos.

```
In [1]: lista=[1, 2, 2<1, 1+3*1j] # entre corchetes
In [2]: type(lista)
list
# CUIDADO programadores fortran/matlab
In [3]: lista[0] # ojo! los indices empiezan en cero
Out[3]: 1
In [4]:  lista[-1]  # ultimo elemento
Out[4]: (1+3j)
In [5]: lista[1:3] # notacion como en matlab/fortran?
Out[5]: [2, False]
In [6]: lista[1:]
Out[6]: [2, False, (1+3j)]
In [7]: lista[::2]
Out[7]:[1, False]
```



`a[i:j]` contiene los elementos entre los índices `i` y `j` y consta de `(j-i)` elementos.

Propiedades

- El índice de una lista empieza en **0** como en C.
- Las listas son *mutables* [a]
- Los elementos de una lista pueden ser de tipos distintos.
- Se pueden añadir y quitar elementos con `append` y `pop` respectivamente.
- Se pueden concatenar listas con + y $*$.

[a] Los objetos mutables pueden cambiar su valor pero mantienen su `id`, la dirección de memoria del objeto. Todos los tipos de números son inmutables.

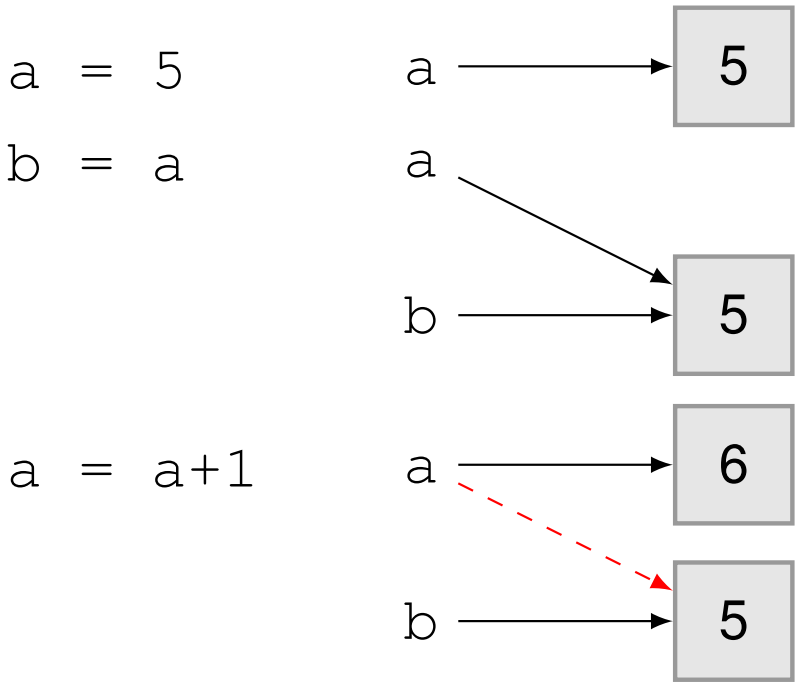
```
In [1]: lista=[1,2,3,4,False]
In [2]: lista.pop(4)  # quita el elemento [4] de lista
Out[2]: False
In [3]: print (lista)
Out[3]: [1, 2, 3, 4]
In [4]: lista.append(True) # agrega True al final
Out[4]: print (lista)
[1, 2, 3, 4, True]
In [5]: lista+lista[::-1]
Out[5]: [1, 2, 3, 4, True, True, 4, 3, 2, 1]
In [6]: lista*2
Out[6]: [1, 2, 3, 4, True, 1, 2, 3, 4, True]
In [7]: lista.sort()
In [8]: lista   # True == 1   y   False == 0
Out[8]: [1, True, 2, 3, 4]
In [9]: lista=[[1,2,3],[[[1j,2]],'a']] # anidar listas
In [10]: print (lista)
[[1, 2, 3], [[[1j, 2]], 'a']]
```

Algunas funciones sobre listas

`len(L)`	Dimensión de la lista `L`
`L.append(x)`	Añade el elemento `x` al final de `L`
`L2.extend(L1)`	Añade la lista `L1` al final de la lista `L2`
`L.insert(i,x)`	Inserta el elemento `x` en la posición dada de `L`
`L.remove(x)`	Borra la primera aparición del elemento `x` en `L`
`L.pop([i])`	Borra el elemento en la posición dada de la lista `L`
`L.index(x)`	Devuelve el índice en la lista de la primera aparición del elemento `x`, y si no está se produce un error.
`L.count(x)`	Número de veces que aparece `x` en `L`
`L.sort()`	Ordena la lista `L`
`L.reverse()`	Devuelve la lista `L` al revés.

Variables mutables/inmutables

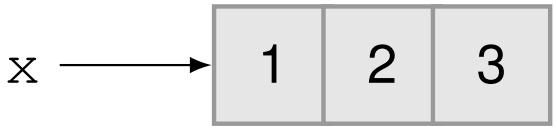
Al ejecutar las sentencias siguientes, las variables `a` y `b` apuntan hacia la misma dirección de memoria. Al redefinir `a=a+1` hace que PYTHON redirija el nuevo valor de `a` hacia otra dirección de memoria donde está el resultado `a+1`.



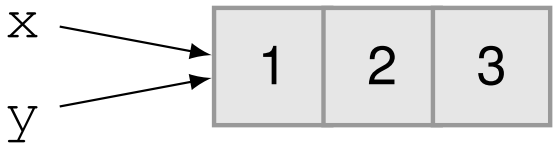
Al modificar una variable inmutable, `Python` genera otro valor dejando el original intacto. Cambiando `a` dejará el valor de `b` intacto ya que direccionan a distintos valores. Si ejecutamos `del a` el valor de `b` sigue siendo `5`.

En el caso de una lista, que es mutable, lo que ocurre es:

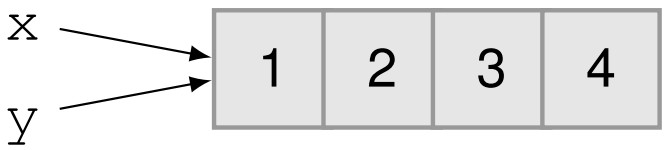
```
x =[1,2,3]
```




```
y=x
```




```
x.append(4)
```

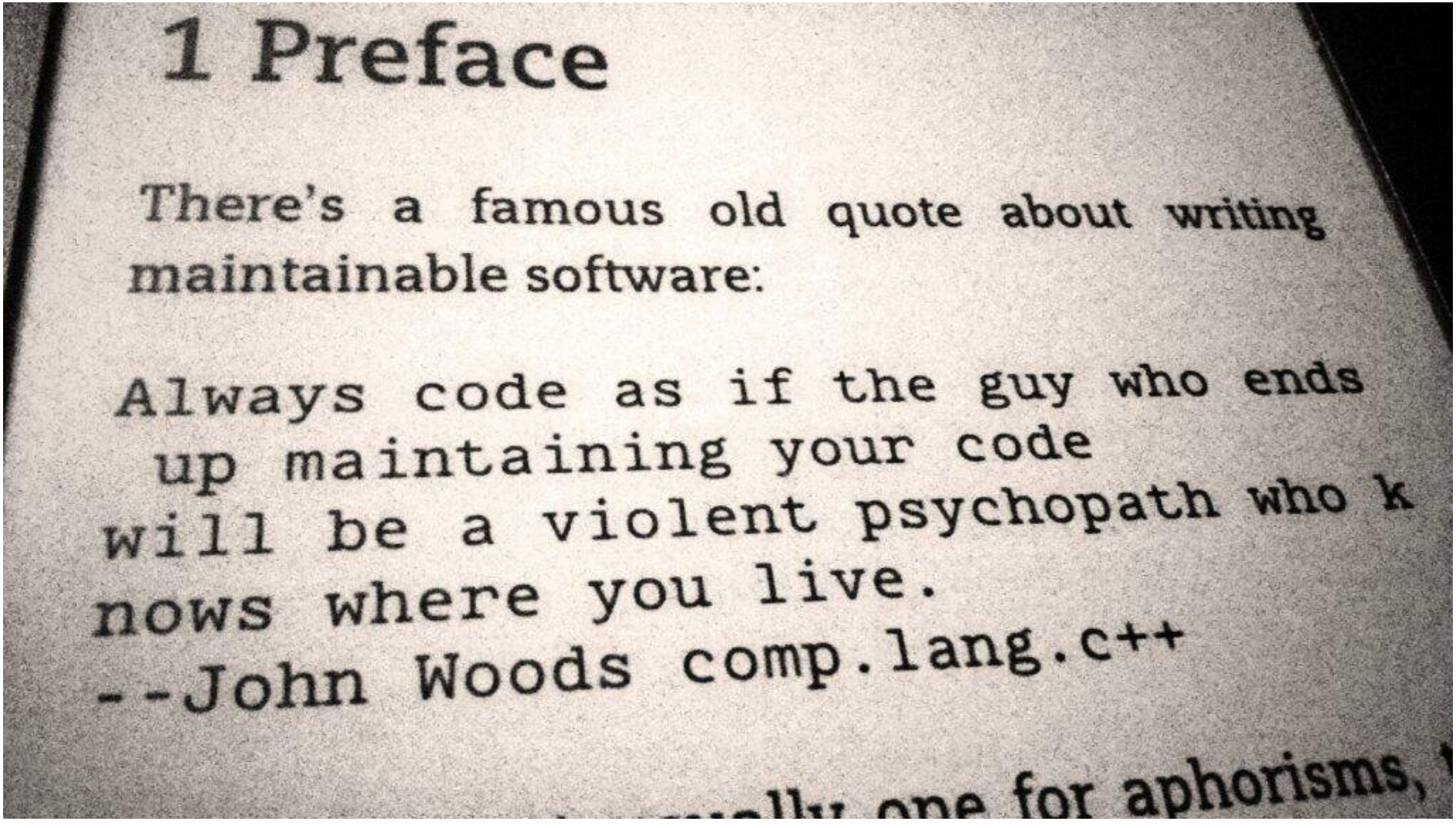


Copiar o no copiar

Para que un vector sea una **copia** totalmente independiente de otro deberemos utilizar el módulo `copy`.

```
In [1]: import copy  # copia dura listas
In [2]: a=[1,2,3,4]
In [3]: b=a
In [4]: c=copy.copy(a)

# id es el identificador unico de cada objeto

In [5]: print (id(a), id(b), id(c) )
140597721749408 140597721749408 140597721874512

In [6]: a.append(5) # agregar un elemento a la lista a

In [7]: print (a, b, c)
[1, 2, 3, 4, 5] [1, 2, 3, 4, 5] [1, 2, 3, 4]
```

Tener en cuenta...



1 Preface

There's a famous old quote about writing maintainable software:

```
Always code as if the guy who ends
 up maintaining your code
will be a violent psychopath who k
nows where you live.
--John Woods comp.lang.c++
```

one for aphorisms,

Controles de flujo `if/elif/else`

En PYTHON se usa el sangrado para definir bloques de código, bucles `if`, `for`, `while`, funciones...

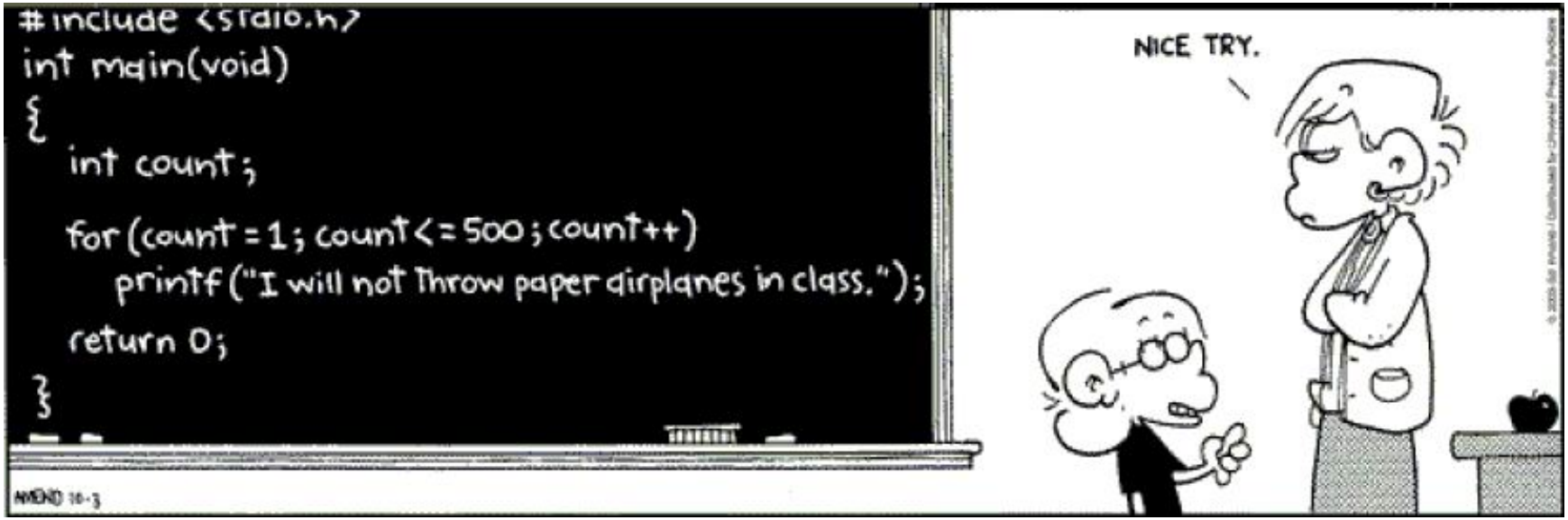
```
if condicion_if:
    bloque_condicion_if
elif condicion_elif:
    bloque_condicion_elif
else:
    bloque_else
```

Notas

- El sangrado (4 espacios en blanco) es necesario[a]
- No hay `end` como en `matlab/octave`, `fortran`

[a] Algunas versiones de IPYTHON no son tan ortodoxas

Control de flujo `for`



Control de flujo `for`

```
for contador:
    bloque_for
```

```
for i in range(5):    # 0,1,2,3,4
    print(i**2, end=" ")   # separados por espacio
0 1 4 9 16
```

```
for color in ['rojo','ocre','azul']:
    print('el color es %s ' % color)
el color es rojo
el color es ocre
el color es azul
```

Controles de flujo `while/break/continue`

Tienen el mismo significado que en `matlab`...

```
while condicion:
    bloque_while
```

```
# Generacion de la sucesion de Fibonacci
a, b = 0, 1     # asignacion multiple
while a<90:
    a, b = b, a+b
    if b>10 and b<40:
        continue # continua con la iteracion siguiente
    print (b, end=" ")
1 2 3 5 8 55 89 144 233
```

```
# Iteracion de Newton para calcular raiz de 2
x0, ite = 2.0, 0
while (1>0):
    x1 = x0/2.0 + 1.0/x0;
    ite = ite + 1;
    print( 'ite= {0}, x1= {1:.18}'.format(ite, x1) )
    if abs(x1-x0) < 1e-12 or ite > 10:
        break
    x0=x1;
```

```
ite= 1, x= 1.5
ite= 2, x= 1.4166666666666652
ite= 3, x= 1.4142156862745098
ite= 4, x= 1.4142135623746899
ite= 5, x= 1.4142135623730492
ite= 6, x= 1.4142135623730492
```

Listas por comprensión

Una lista por comprensión es una construcción que permite generar una lista basada en listas existentes. La forma de definirla es muy similar a la definición de un conjunto en matemáticas. Por ejemplo,

$$S = \{2n \,|\, n \in \mathbf{N}, n^2 <= 121\}$$

```
In [1]: S = [2*n for n in range(101) if n**2 <=121 ]
In [2]: print (S)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

```
In [3]: S=[ [x,y] for x in range(2) for y in 'abc']
In [4]: S
Out[4]: [[0,'a'],[0,'b'],[0,'c'],[1,'a'],[1,'b'],[1,'c']]

In [5]: R=range(3)
In [6]: H=[ 1./(i+j+1) if i<=j else 0 for i in R for j in R]
In [7]: H
Out[7]: [1.0, 0.5, 0.3333333, 0, 0.333333, 0.25, 0, 0, 0.20]
```

Funciones Ejercicio

La forma típica para definir una función es:

```
def nombre_funcion(args):
    bloque_funcion
```

Por defecto, las funciones en PYTHON devuelven `None`, aunque lo usual es emplear **`return`** `resultados`. En principio, las variables en la función tienen su propio espacio, i.e., son locales.

```
In [1]: def fun(x):    # es necesario pasar x
   ...:     return sin(x*cos(x**5))
   ...:
In [2]: fun(8.0)    # tambien vale fun(x=8.0)
Out[2]: 0.15743233177672805
In [3]: fun()          # da error
---------------------------------------------------------------
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:fun() missing 1 required positional argument: 'x'
---------------------------------------------------------------
```

Ficheros

En PYTHON los ficheros se abren con `open(fichero, modo)`. El primer parámetro es el nombre del fichero y el segundo una variable *string* conteniendo caracteres que describen la forma de abrir el fichero, `r` sólo lectura, `r+` lectura y escritura, `w` sólo escritura, `a` para añadir registros nuevos al final del fichero, etc.

```
In [1]: f = open("file.txt") # por defecto solo lee, 'r'
In [2]: f
Out[2]: <_io.TextIOWrapper name='file.txt' mode='r'...>
In [3]: f.close()                  # cerrar el fichero
```

```
In [4]: cat file.txt
primera linea
segunda linea
tercera linea
```

Para la lectura del fichero, podemos emplear:

`f.read()`	Devuelve una variable *string* conteniendo el fichero completo. Las líneas están separadas por `\n`. Si se ha alcanzado el final `f.read()` devuelve la *string* vacía.
`f.readline()`	Devuelve una variable *string* conteniendo una línea del fichero. Cuando el final se ha alcanzado `f.readline()` devuelve la *string* vacía.
`f.readlines()`	Devuelve una lista conteniendo todas las líneas del fichero.

```
In [1]: f.read()
'primera linea\nsegunda linea\ntercera linea'
In [2]: f.read()
''
```

```
In [1]: f.readline()
'primera linea\n'
In [2]: f.readline()
'segunda linea\n'
In [3]: f.read()
'tercera linea'
In [4]: f.read()
''
```

```
In [5]: f.readlines()
['primera linea\n', 'segunda linea\n', 'tercera linea']
```

```
f = open('file.txt','r')
for line in f:
    print(line)
f.close()
```

Para escribir en un fichero, usamos `fwrite(cadena)`, y si queremos guardar números, antes hay que transformarlos a variables *string*.

```
In [1]: f=open("file1.txt","w")
In [2]: linea='una aprox de pi es ' + str(355/113.)+"\n"
In [3]: f.write(linea)
In [4]: linea=r'segunda linea \n'  # raw string \n
In [5]: f.write(linea)
In [6]: linea='tercera linea \n'
In [7]: f.write(linea)
In [8]: f.close()
```

El contenido del fichero `file1.txt` es:

```
una aprox de pi es 3.141592920353982
segunda linea \ntercera linea
```

Numpy

En NUMPY el tipo fundamental es el `array`, que básicamente es una lista con un sólo tipo (entero, coma flotante o complejo).

```
In [1]: import numpy as np
In [2]: a = np.array([0,1,3.0])
In [3]: print (a, a[0], a[-1])
[0. 1. 3.] 0.0 3.0
In [4]: type(a)
Out[4]: <type 'numpy.ndarray'>

In [5]: A = np.array([[1,2],[2,3.0]])
In [6]: print (A)
[[ 1.  2.]
 [ 2.  3.]]
In [7]: C = np.eye(2) - np.ones(2)
In [8]: print (C)
array([[ 0., -1.],
       [-1.,  0.]])
```

```
In [1]: A = np.array( [ [1.5,2,3], [4,5,6] ] )
In [2]: print (A)
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
In [3]: A.dtype
Out[3]: dtype('float64')
In [4]: C = np.array( [[1,2], [3,4]], dtype=complex)
In [5]: print (C)
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])

In [6]: np.linspace( 0, 2, 5 )
Out[6]: array([ 0.,  0.5, 1., 1.5, 2. ])

In[7]: R = np.range(3)
In[8]: H=[1/(i+j+1) if i<=j else 0 for i in R for j in R]
In[9]: np.array(H).reshape(3,3)
Out[9]:
array([[1.        , 0.5       , 0.33333333],
       [0.        , 0.33333333, 0.25      ],
       [0.        , 0.        , 0.2       ]])
```

Producto de matrices 

```
In [7]: A = np.array( [[1,1], [2.,3]])

In [8]: B = np.array( [[1,-1], [2.,-4]])

# en matlab es A*B
In [9]: print (np.dot(A, B)) # producto matricial
[[  3.  -5.]
 [  8. -14.]]

# en matlab es A.*B
In [10]: print (A*B)      # producto elemento a elemento
[[  1.   -1.]
 [  4.  -12.]]
```

```
In [69]: A = np.fromfunction(lambda i,j: 2*i+j, (3,3))
In [70]: print (A)
[[ 0.,  1.,  2.],
 [ 2.,  3.,  4.],
 [ 4.,  5.,  6.]])
In [71]: A[0,0]  # elemento 1,1
Out[71]: 0.0
In [72]: A[1]    # segunda fila
Out[72]: array([ 2.,  3.,  4.])
In [73]: A[-1]   # ultima fila
Out[73]: array([ 4.,  5.,  6.])
In [74]: A[:,1]  # segunda columna
Out[74]: array([ 1.,  3.,  5.])
In [75]: A[:,-1] # ultima columna
Out[75]: array([ 2.,  4.,  6.])
In [76]: A[:,1]+A[1,:] # suma de fila y columna !!!
Out[76]: array([ 3.,  6.,  9.])
```

Notar que en principio no hay diferencia entre vectores fila y columna.

Notación :

a[0,3:5] a[4:,4:] a[:,1] a[2::2,::2]

Broadcasting

Anteriormente, hemos visto que es posible hacer operaciones con `arrays` de tamaño/forma diferente si NUMPY puede transformarlos para que sea posible. Esto se llama **broadcasting**.

```
In [1]: a = np.arange(3)
In [2]: print (a)
[0 1 2]
In [3]: a.shape = (3,1)   # a=np.arange(3).reshape(3,1)
In [4]: print (a)
[[0]
 [1]
 [2]]
In [5]: b = np.array([1,1,1])
In [6]: print (a+b)
[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

```
In [1]: a=np.array([[0],[1],[2]]); b=np.array([1,1,1]);
In [2]: A=np.array([[1,2,3],[4,5,6],[7,8,9]])
In [3]: print (a+b)   # vector columna + fila
[[1 1 1]
 [2 2 2]
 [3 3 3]]
In [4]:  print (a+A) # vector columna + matriz
[[ 1  2  3]
 [ 5  6  7]
 [ 9 10 11]]
In [5]: print (b+A) # vector fila + matriz
[[ 2  3  4]
 [ 5  6  7]
 [ 8  9 10]]
In [6]: print (np.dot(A,a)) # matriz * vector columna
[[ 8]
 [17]
 [26]]
In [7]: print (np.dot(A,b)) # matriz * vector fila
[ 6 15 24]
# probar np.dot(b,A), np.dot(b,a)...
```

asignación y `copy()`

```
In [1]: import numpy as np
In [2]: a = np.arange(4)
In [3]: a
Out[3]: array([0, 1, 2, 3])
In [4]: b = a
In [5]: c = a.copy() # c = np.copy(a)
In [6]: a[1:3]=999

In [7]: print (a, b, c)
[  0 999 999 3]    [  0 999 999 3]    [0 1 2 3]

In [8]: print (id(a), id(b), id(c))
62764864 62764864 62394656
```

Ficheros. `loadtxt/savetxt` en NUMPY

En `Ipython` es posible utilizar comandos del tipo `cd`, `pwd`...

```
# guardar una matriz en un fichero ASCII
In [3]: np.savetxt('fich.txt', np.random.random((3,2)))
In [4]: ls fich*
fich.txt

In [5]: cat fich.txt
3.25916017078421882 4e-01 4.1460603037109744 48e-02
9.8844718122868713 28e-01 8.5280813900026830 60e-01
8.3488971348487994 73e-01 5.6735287069147621 87e-01

# cargar fichero en una matriz con loadtxt
In [6]: A = np.loadtxt('fich.txt')
In [7]: A
array([[ 0.32591602,  0.0414606 ],
       [ 0.98844718,  0.85280814],
       [ 0.83488971,  0.56735287]])
```

MATLAB	PYTHON
`fichero[.m]`	`run fichero.py` `execfile('fichero.py')`
`quit`	`CTRL + D`
`help tópico`	`help (tópico) info(tópico)`

`rand(m,n)`
`randn(m,n)`
	`from numpy import *`
	`random.uniform(a,b,(m,n))`
	`random.standard_normal(a,b,(m,n))`

Matriz $m \times n$ de números aleatorios unifor. o normal. distribuidos en $[a, b]$.	
`a=[1,2,3]`	`a=array([1,2,3])`
`a=[1;2;3]`	`a=array([1,2,3]).reshape(-1,1)`
`a(2:end)`	`a[1:]`
Elimina el primer elemento de un vector	
`[v,i] = max(a)`	`v,i = a.max(0),a.argmax(0)`
`[a; b]`	`vstack((a,b))`
`[a, b]`	`hstack((a,b))`
`zeros(m,n)`	`zeros((m,n), float)`
`eye(n)`	`identity(n)` o `eye(n)`
`diag([4, 5, 6])`	`diag((4,5,6))`

MATLAB	PYTHON
`reshape(a,m,n)`	`a.reshape(m,n)`
`a(:)'`	`a.flatten(0)`
`a(:)`	`a.flatten(1)`
Convierte una matriz a un vector, (0,filas), (1,columnas)	
`a(1,:)`	`a[0,]` Primera fila
`a(:,1)`	`a[:,0]` Primera columna
`a(end-1:end,:)`	`a[-2:,]` Últimas dos filas
`a(a>0.1) = 9;`	`a[a>0.1]=9`
`a'`	`a.conj().transpose()` o `a.conj().T`
`a.'` o `transpose(a)`	`a.transpose()` o `a.T`
`det(a)`	`linalg.det(a)` Determinante
`inv(a)`	`linalg.inv(a)` Inversa
`pinv(a)`	`linalg.pinv(a)` Pseudo-inversa
`norm(a)`	`norm(a)` Normas
`svd(a)`	`linalg.svd(a)` Valores singulares
`chol(a)`	`linalg.cholesky(a)` factorización

MATLAB	PYTHON
`[v,V] = eig(a)`	`v,V=linalg.eig(a)` Valores, vectores propios
`[l,u,p] = lu(a)`	`from scipy import linalg` `p,l,u=linalg.lu(a)` Factorización LU
`eps`	`finfo(float).eps`
`1i, 1j`	`1j` Unidad imaginaria
`a * b`	`dot(a,b)`
`a .* b`	`a * b`
`a.^3`	`a**3`
`[Q,R,P]=qr(a,0)`	`Q,R = linalg.qr(a)` Factorización QR
`b = a` `b=a(2,:)`	`b = a.copy()` Copia de un vector `b = a[1,:].copy()`
`a .* (a>0.5)`	`a * (a>0.5)`
`atan2(x,y)`	`from math import *` `atan2(x,y)`
`kron(a,b)`	`kron(a,b)`

`sum(a)`
`sum(a')`
`sum(a(:))`
	`a.sum(axis=0)` suma cada columna
	`sum(a,axis=1)` suma cada fila
	`a.sum()` o `sum(a)` suma total

MATLAB	PYTHON
`abs(z)`	`abs(z)` valor absoluto
`real(z)`	`z.real` Real parte real
`imag(z)`	`z.imag` parte imaginaria
`conj(z)`	`z.conj()` complejo conjugado
	`from scipy import linalg`
`expm(a)`	`scipy.linalg.expm(a)` exponencial matricial
`logm(a)`	`scipy.linalg.logm(a)` logaritmo matricial
`size(a)`	`a.shape`
`size(a,2)`	`a.shape[1]`
`length(a)`	`size(a, axis=1)`
`length(a(:))`	`a.size`
`ndims(a)`	`a.ndim`