Técnicas Computacionales Básicas
Tema 1: Introducción a Python

Sebastian L. Hidalgo

8 Sep. 2013

1 Tema 1: Introducción a Python.

1.1 Leguajes de alto nivel: interpretados y compilados.

Podemos dividir los lenguajes de programación en dos grupos: lenguajes de bajo y alto nivel. Los lenguajes de bajo nivel son lenguajes de nivel próximo a la máquina: sus instrucciones requieren de una sintaxis predefinida con fuerte relación con los detalles del procesador. Los lenguajes de alto nivel tienen un nivel de abstracción superior, más cercano al lenguaje natural y alejado de las características de sintaxis del procesador. Como ejemplos de lenguajes de bajo nivel podemos cintar *Ensamblador* y *Código Máquina*. Existen multitud de lenguajes de alto nivel, como por ejemplo, *Perl, Python, Octave, FORTRAN, C++, Ruby, BASIC y COBOL*.

Una de las limitaciones de los lenguajes de bajo nivel es que se requiere de ciertos conocimientos de programación para realizar las secuencias de instrucciones lógicas, aunque el rendimiento es algo mayor que en los de alto nivel. Los lenguajes de alto nivel se crearon para que el usuario común pudiese solucionar un problema de procesamiento de datos de una manera más fácil y rápida en un lenguaje más próximo al humano.

De entre los lenguajes de alto nivel, podemos distinguir también dos categorías: lenguajes compilados y lenguajes interpretados. Los primeros necesitan de un software adicional (compilador) que traduce el código fuente a lenguaje máquina para que pueda ejecutarse. Los lenguajes interpretados ejecutan las instrucciones directamente sin necesidad de compilación. La ejecución de un código en un lenguaje compilado es más rápida que en uno interpretado, aunque son menos flexibles cuando es necesario su depuración en la búsqueda de errores o en la implementación de mejoras. Ejemplos de lenguajes compilados de alto nivel son *FORTRAN, C++, COBOL y BASIC*. Ejemplos de lenguajes interpretados son *IDL, Matlab, R, Octave, Perl, Ruby y Python*.

Python es un lenguaje desarrollado por Guido van Rossum a finales de la década de los 80. Su nombre proviene del grupo de comedia Monty Python. Es de código abierto y soportado por la mayoría de los sistemas operativos: GNU/Linux, Windows y MacOS. Sus aplicaciones son muy extensas, siendo usado por organizaciones como Google, el CERN o la NASA. Forma parte de un gran número de sistemas operativos, entornos gráficos y programas, como GNU/Linux, KDE, GIMP, Inkscape, Libreoffice, Maya, Blender, etc.

1.2 Instalación y arranque.

1. **Instalación en Windows**: Python no viene preinstalado en Windows, así que descargamos un instalador oficial:

   Para 32 bits: `http://www.python.org/ftp/python/3.3.2/python-3.3.2.msi`
   Para 64 bits: `http://www.python.org/ftp/python/3.3.2/python-3.3.2.amd64.msi`

2. **Instalación en MAC OS X**: Python viene preinstalado en Mac, pero debido a las restricciones de Apple, a menudo está uno o dos años sin actualizar. La recomendación es descargar e instalar

una versión nueva desde la página oficial de Python:

Para 32 bits: `http://www.python.org/ftp/python/3.3.2/python-3.3.2-macosx10.6.dmg`
Para 64 bits: `http://www.python.org/ftp/python/3.3.2/python-3.3.2-macosx10.5.dmg`

3. **Instalación en GNU/Linux**: Dado que Python forma parte de todas las distribuciones de GNU/Linux ya viene preinstalada. Si queremos instalar python 3, desde el gestor de paquetes puede hacerse fácilmente. También desde una consola: por ejemplo, si tenemos gestor de paquetes yum: **yum install python3**

Además, es conveniente instalar módulos o librerías que amplían las funciones que pueden usarse con Python. En este curso vamos a usar los siguente módulos:

1. **Numpy**: Paquete de propósito general para procesar vectores y matrices de múltiples dimensiones, transformada de Fourier discreta, álgebra lineal y generación de números aleatorios. Instalación: **yum install python3-numpy**
2. **Scipy**: Paquete orientado a manipulación multidimensional de arrays, rutinas numéricas tales como integración numérica, ecuaciones diferenciales y optimización. Instalación: **yum install python3-scipy**
3. **Matplotlib**: Librería para producir figuras en 2D. Instalación: **yum install python3-matplotlib**
4. **Ipython**: Entorno para trabajar de forma interactiva y fácil con Python. Instalación: **yum install python3-ipython**
5. **Math**: Librería incluida en Python que contiene constantes y funciones matemáticas.

Python se inicia desde una terminal de comandos estándar de Linux (o MacOS, o ejecutando el programa desde el menú de Windows) sin más que escribir python. En el caso de querer iniciar la consola ipython hay que escribir ipython. Aparecerá el prompt ⋙ (en la consola estándar de Python) y entonces pueden empezar a utilizarlo.

Alternativamente, podemos arrancar python desde una terminal usando la interfaz ipython3 que añade funciones extras a python, tales como hisotrial, completar nombres, comandos para el sistema operativo, etc. Para iniciar ipython para python3, escribimos:

```
[shidalgo@laurel]$ ipython3
Python 3.3.2 (default, Aug 23 2013, 19:00:04)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.2 -- 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.
# ipython also support autocomplete with tabular key.
```

Nuesto primer programa en Python:

```
In [1]: print('Hola python')
Hola python
```

Para obtener ayuda sobre cómo ejecutar un comando de python, escribimos:

```
In [3]: print?
Type:        builtin_function_or_method
String Form:<built-in function print>
Namespace:   Python builtin
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
In [4]:
```

Para salir de python, escribimos *exit*

```
In [5]: exit
```

Podemos ejecutar también programas escritos en python desde un fichero sin necesidad de iniciar la consola de python o ipython:

```
[shidalgo@laurel ~]$ echo "print('Hola python')" > pro.py
[shidalgo@laurel ~]$ ipython3 pro.py
Hola python
[shidalgo@laurel ~]$ python3 pro.py
Hola python
[shidalgo@laurel ~]$
```

Alternativamente, podemos usar la sintaxis estándard #! en un fichero de texto que contenga un programa escrito en python y ejecutar directamente el fichero, por ejemplo, si modificamos el programa anterior *pro.py* añadiendo una línea que diga que lo que hay escrito más abajo es lenguaje python:

```
[shidalgo@laurel ~]$ cat pro.py
#!/usr/bin/python3.3
print('Hola python')
[shidalgo@laurel ~]$
```

Podemos ejecutar directamente el fichero (primero, hemos de darle permiso de ejcución):

```
[shidalgo@laurel ~]$ chmod +x pro.py
[shidalgo@laurel ~]$ ./pro.py
Hola python
[shidalgo@laurel ~]$
```

1.3 Tipos de datos.

Los tipos de datos más usado que vienen en python (built-in, para distinguirlos de los que crea el usuario) se dividen en:

- *None* : Representa ausencia de valor en una variable o argumentos que no pasan a una función.
- *Numeric* : Tipos numéricos que pueden dividirse en
  - Integer (int) : Números enteros.
  - Float (float) : Números de coma flotante.
  - Complex (complex) : Numeros complejos.
- *Boolean* : Variables lógicas que toman los valores *False* o *True*. Se consideran parte de los *int*.
- *String* : Cadenas de caracteres. Van entre comillas siempre para distinguirlos de las variables.

En python podemos preguntar qué tipo es el de una variable:

```
In [1]: type(2)
Out[1]: builtins.int

In [2]: type(2.3)
Out[2]: builtins.float

In [3]: type(2+3j)
Out[3]: builtins.complex

In [4]: type(True)
Out[4]: builtins.bool

In [5]: type('hola')
Out[5]: builtins.str
```

1.4 Operadores.

Para realizar operaciones entre los diferente tipos de datos, se usa:

Operaciones lógicas	
Operación	Resultado
x **or** y	True (=1) si x o bien y son True, de otro modo False (=0)
x **and** y	True (=1) si x e y son True, de otro modo False (=0)
not x	True (=1) si x=False, False (=0) si x=True

Operaciones comparación	
Operación	Resultado
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Operaciones aritméticas	
Operación	Resultado
x + y	sum of x and y
x - y	difference of x and y
x * y	product of x and y
x / y	quotient of x and y
x // y	floored quotient of x and y
x % y	remainder of x / y
-x	x negated
+x	x unchanged
abs(x)	absolute value or magnitude of
int(x)	x converted to integer
float(x)	x converted to floating point
complex(re, im)	a complex number with real part re, imaginary part im. im defaults to zero.
c.conjugate()	conjugate of the complex number c
divmod(x, y)	the pair (x // y, x % y)
pow(x, y)	x to the power y
x ** y	x to the power y

La prioridad en la ejecución (separadas por ;) es la siguiente: $**$; $*$; / ; % ; + ; $-$

Prioridad en las operaciones.

```
In [31]: x=3.5

In [2]: x**2
Out[2]: 12.25

In [3]: x+x**2
Out[3]: 15.75

In [4]: x**(2./3.)
Out[4]: 2.3052181460292234

In [5]: x**(2/3)
Out[5]: 2.3052181460292234

In [6]: 123%8
Out[6]: 3

In [7]: x+x/2+x*x-x**1.25-x%2
Out[7]: 11.21276160046417
```

Python contiene funciones predefinidas:

Fucntion	Description
abs(x)	Return the absolute value of a number
all(iterable)	Return True if all elements of the iterable are true (or if the iterable is empty).
any(iterable)	Return True if any element of the iterable is true. If the iterable is empty, return False.
ascii(object)	Return a string containing a printable representation of an object.
bin(x)	Convert an integer number to a binary string.
bool([x])	Convert a value to a Boolean.
bytearray()	Return a new array of bytes.
bytes()	Return a new 'bytes' object, which is an immutable sequence of integers in the range 0 <= x < 256.
callable(object)	Return True if the object argument appears callable, False if not.
chr(i)	Return the string representing a character whose Unicode codepoint is the integer i.
classmethod(function)	Return a class method for function.
compile()	Compile the source into a code or AST object.
complex([real[,imag]])	Create a complex number with the value real + imag*j or convert a string or number to a complex number.
delattr(object, name)	The function deletes the named attribute, provided the object allows it.
dict()	Create a new dictionary.
dir()	Return the list of names in the current local scope.
divmod(a,b)	Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division.
enumerate()	Return an enumerate object.
eval(expression)	The expression argument is parsed and evaluated as a Python expression.
exec()	This function supports dynamic execution of Python code.
filter(function, iterable)	Construct an iterator from those elements of iterable for which function returns true.
float(x)	Convert a string or a number to floating point.
format(value[, format_spec])	Convert a value to a 'formatted' representation, as controlled by format_spec.
frozenset()	Return a new frozenset object, optionally with elements taken from iterable.
getattr()	Return the value of the named attribute of object. name must be a string.
globals()	Return a dictionary representing the current global symbol table.
hasattr(object, name)	The arguments are an object and a string. The result is True if the string is the name of one of the object's attributes, False if not.
hash(object)	Return the hash value of the object (if it has one).
help([object])	Invoke the built-in help system.
hex(x)	Convert an integer number to a hexadecimal string.
id(object)	Return the 'identity' of an object.
import	Import modules and/or attributes.
input([prompt])	If the prompt argument is present, it is written to standard output without a trailing newline.
int(x)	Convert a number or string x to an integer, or return 0 if no arguments are given.
isinstance(object, classinfo)	Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual) subclass thereof.
issubclass(class, classinfo)	Return true if class is a subclass (direct, indirect or virtual) of classinfo.
iter()	Return an iterator object.
len()	Return the length (the number of items) of an object.

list()	Builds a list.
locals()	Update and return a dictionary representing the current local symbol table.
map()	Return an iterator that applies function to every item of iterable, yielding the results.
max()	Return the largest item in an iterable or the largest of two or more arguments.
memoryview()	Return a 'memory view' object created from the given argument.
min()	Return the smallest item in an iterable or the smallest of two or more arguments.
next(iterator)	Retrieve the next item from the iterator.
object()	Return a new featureless object.
oct()	Convert an integer number to an octal string.
open()	Open file and return a corresponding file object.
ord()	Return an integer representing the Unicode code point of a character.
pow(x, y[, z])	Return x to the power y; if z is present, return x to the power y, modulo z.
print(object)	Print objects to the stream file.
property()	Return a property attribute.
range()	Creates a sequece of numbers equally spaced.
repr()	Return a string containing a printable representation of an object.
reversed()	Return a reverse iterator.
round(number[, ndigits])	Return the floating point value number rounded to ndigits digits after the decimal point.
set()	Return a new set object, optionally with elements taken from iterable.
setattr()	The function assigns the value to the attribute, provided the object allows it.
slice()	Return a slice object representing the set of indices specified by range(start, stop, step).
sorted(iterable)	Return a new sorted list from the items in iterable.
staticmethod()	Return a static method for function.
str()	Return a str version of object.
sum(iterable[, start])	Sums start and the items of an iterable from left to right and returns the total.
super()	Return a proxy object that delegates method calls to a parent or sibling class of type.
tuple()	Return a tuple.
type()	With one argument, return the type of an object.
vars([object])	Return the dictionary attribute for a module, class, instance, or any other object.
zip()	Make an iterator that aggregates elements from each of the iterables.

1.5 Módulos.

Además de las operaciones definidas anteriormente y que vienen incorparadas en Python, existen funcionalidades extras para prácticamente cualquier proyecto que queramos hacer con Python. Para extender estas funcionalidades es necesario importar en Python módulos adicionales. En este curso usaremos los más comunes en ciencia: *numpy, scipy y matplotlib*. Para usarlos, en la consola de python escribimos:

Importando módulos.

```
In [1]: import numpy
In [2]: import scipy
In [3]: import matplotlib
In [4]: dir(numpy)          # List everything included in numpy
# For example, numpy includes sin function, we use it as:
In [5]: numpy.sin(numpy.pi/2)
Out[5]: 1.0
# To avoid the use of large names in functions, as numpy.sin, we can import the
    library with a short name:
In [6]: reset   # clear all variables and modules imported.
Once deleted, variables cannot be recovered. Proceed (y/[n])? y

In [7]: import numpy as np
In [8]: np.sin(np.pi/2)
Out[8]: 1.0
# We can import just functions one by one from any module:
In [9]: from numpy import cos
In [10]: from numpy import pi
In [11]: cos(pi/2)
Out[11]: 6.123233995736766e-17   # Watch this! The precssion of operations is
    very important!
# Other choice is to import all functions from one module:
In [12]: from numpy import *
In [13]: sin(pi/2)
Out[13]: 1.0

# Different modules could include the same function. It is not recomended to mix
    the same function from different modules.
```

Generalmente el valor de una variable se asigna haciendo por ejemplo $d = 3.0$, pero se puede hacer que pida una entrada por teclado usando la función input() de la forma siguiente:

Indroducción de datos por pantalla:

```
In [1]:  d = input('introduce data: ')
introduce data: 3

In [2]: d        # d is a string by default
Out[2]: '3'
In [3]: d=float(d)
In [4]: print(d)
3.0
In [5]: d=float(d) # converted to float
In [6]: print(d)
3.0
```

1.6 Operaciones y estructuras de datos.

1.6.1 Operaciones con números

Comentando líneas de código

```
In [1]: # this is the first comment
In [2]: spam = 1  # and this is the second comment
In [3]:           # ... and now a third!
In [4]: text = "# This is not a comment because it's inside quotes."
In [5]: print(text)
# This is not a comment because it's inside quotes.
```

Los números enteros (5, 2, 10) tienen tipo *int*. Si poseen decimal (5.0, 2.1, 10.01, etc) son de tipo *float*. En Python 3 (a diferencia de Python 2.7), la división de dos *int* da como resultado un *float*

Operaciones básicas 1

```
In [1]: 2+2
Out[1]: 4
In [2]: 50 - 5*6
Out[2]: 20
In [3]: (50 - 5*6) / 4
Out[3]: 5.0
In [4]: 8 / 5    # division always returns a floating point number
Out[4]: 1.6
```

Para obtener un *int* descartando la parte decimal, usamos el operador //

Operaciones básicas 2

```
In [1]: 17 / 3     # classic division returns a float
Out[1]: 5.666666666666667
In [2]: 17 // 3    # floor division discards the fractional part
Out[2]: 5
In [3]: 17 % 3     # the % operator returns the remainder of the division
Out[3]: 2
In [4]: 5 * 3 + 2  # result * divisor + remainder
Out[4]: 17
```

Para operar con potencias:

Operaciones básicas 3

```
In [1]: 2 ** 7    # 2 to the power of 7
Out[1]: 128
```

Para asignar valores a variables:

Operaciones básicas 4

```
In [1]: width = 20

In [2]: height = 5 * 9

In [3]: width * height
Out[3]: 900

In [4]: x=1; y=2; c=3   # Variables can be assigned in the same line by using ";"
```

El caracter especial _ almacena el último valor mostrado:

Operaciones básicas 5

```
In [1]: 200 / 10
Out[1]: 20.0
In [2]: _
Out[2]: 20.0
In [3]: tax = 12.5 / 100
In [4]: price = 100.50
In [5]: price * tax
Out[5]: 12.5625
In [6]: price + _
Out[6]: 113.0625
In [7]: round(_,2)
Out[7]: 113.06
```

1.6.2 Operaciones con caracteres

Asignar cadenas de caracteres a una variable, usamos comillas simples (' ') o dobles (" "), el resultado es el mismo.

Operaciones con caracteres 1

```
In [1]: 'spam eggs'                    # single quotes
Out[1]: 'spam eggs'
In [2]: 'doesn\'t'                     # use \' to escape the single quote...
Out[2]: "doesn't"
In [3]: "doesn't"                      # use \' to escape the single quote...
Out[3]: "doesn't"
In [4]: '"Yes," he said.'
Out[4]: '"Yes," he said.'
In [5]: "\"Yes,\" he said."
Out[5]: '"Yes," he said.'
In [6]: '"Isn\'t," she said.'
Out[6]: '"Isn\'t," she said.'
```

```
In [7]: s = 'First line.\nSecond line.'  # \n means newline
In [8]: s
Out[8]: 'First line.\nSecond line.'
In [9]: print(s)                         # with print(), \n produces a new line
First line.
Second line.
```

Las cadenas de caracteres pueden concatenarse:

Operaciones con caracteres 2

```
In [1]: 3 * 'un' + 'ium'          # 3 times 'un', followed by 'ium'
Out[1]: 'unununium'
In [2]: 'Py' 'thon'               # Two or more string literals next to each
   other are automatically concatenated.
Out[2]: 'Python'

In [3]: prefix = 'Py'             # This only works with two literals though, not
    with variables or expressions:
In [4]: prefix 'thon'             # can't concatenate a variable and a string
   literal
  File "<ipython-input-113-37b5e5a6971f>", line 1
    prefix 'thon'
                ^
SyntaxError: invalid syntax

In [5]: ('un' * 3) 'ium'
  File "<ipython-input-114-826b8aeb7d3b>", line 1
    ('un' * 3) 'ium'
                   ^
SyntaxError: invalid syntax

In [6]: prefix + 'thon'           # If you want to concatenate variables or a
   variable and a literal, use +:
Out[6]: 'Python'
In [7]: ('un' * 3) + 'ium'
Out[7]: 'unununium'
```

Pueden usarse los paréntesis para unir largas cadenas de caracteres:

Operaciones con caracteres 3

```
In [1]: text = (' Put several strings within parentheses '
   .....: 'to have them joined together.')

In [2]: text
Out[2]: ' Put several strings within parentheses to have them joined together.'
```

Las variables de caracteres (*strings*) pueden ser indexadas:

Operaciones con caracteres 4

```
In [1]: word = 'Python'
In [2]: word[0]  # character in position 0 (0 is the first index)
Out[2]: 'P'
In [3]: word[5]  # character in position 5
Out[3]: 'n'
# Indices may also be negative numbers, to start counting from the right
In [4]: word[-1]  # last character.
Out[4]: 'n'
In [5]: word[-2]  # second-last character
Out[5]: 'o'
In [6]: word[-6]
Out[6]: 'P'
# In addition to indexing, slicing is also supported. Slicing allows you to
    obtain substring
In [7]: word[0:2]  # characters from position 0 (included) to 2 (excluded)
Out[7]: 'Py'
In [8]: word[2:5]  # characters from position 2 (included) to 5 (excluded)
Out[8]: 'tho'
In [9]: word[:2]  # character from the beginning to position 2 (excluded)
Out[9]: 'Py'
In [10]: word[4:]  # characters from position 4 (included) to the end
Out[10]: 'on'
In [11]: word[-2:] # characters from the second-last (included) to the end
Out[11]: 'on'
```

Para recordar cómo funcionan los índices

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1
```

Modificación de las variables *str*

```
#Python strings cannot be changed, they are immutable:
In [12]: word[0] = 'J'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-153-197b67ffdd83> in <module>()
----> 1 word[0] = 'J'
TypeError: 'str' object does not support item assignment

# If you need a different string, you should create a new one:
In [13]: 'J' + word[1:]
Out[13]: 'Jython'
In [14]: word[:2] + 'py'
Out[14]: 'Pypy'
#The built-in function len() returns the length of a string:
In [15]: s = 'supercalifragilisticexpialidocious'
In [16]: len(s)
Out[16]: 34
```

1.6.3 Estructura de datos: listas

Python permite componer los datos en listas (lists) donde se puede combinar diferentes tipos de variables: int, flt, str, etc.

Listas 1 : asignación

```
In [1]: squares = [1, 2, 4, 9, 16, 25]
In [2]: squares[0]  #  Indexing: indexing returns the item
Out[2]: 1
In [3]: squares[-1]
Out[3]: 25

In [4]: squares[-3:]  # slicing returns a new list
Out[4]: [9, 16, 25]
In [5]: squares[:]
Out[5]: [1, 2, 4, 9, 16, 25]

In [6]: squares + [36, 49, 64, 81, 100]      # operations with lists: concatenate
     but not math sum
Out[6]: [1, 2, 4, 9, 16, 25, 36, 49, 64, 81, 100] # only works with list, not
    combination of lists and numbers

In [7]: cubes = [1, 8, 65, 125]  # It is possible to change their content:
    something's wrong here
In [8]: 4 ** 3  # the cube of 4 is 64, not 65!
Out[8]: 64
In [9]: cubes[2] = 4 ** 3  # replace the wrong value
```

```
In [10]: cubes
Out[10]: [1, 8, 64, 125]

# You can also add new items at the end of the list, by using the append() or
    insert():
In [11]: cubes.append(216)  # add the cube of 6
In [12]: cubes.append(7 ** 3)  # and the cube of 7 at the end of the list
In [13]: cubes
Out[13]: [1, 8, 64, 125, 216, 343]
In [14]: cubes.insert(2, 27)   # Add 27 in position  2 (i.e. in third place of
    the list)
In [15]: print(cubes)
[1, 8, 27, 64, 125, 216, 343]
```

Listas 2 : concatenación

```
#Assignment to slices is also possible, and this can even change the size of the
    list or clear it entirely:
In [16]: letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
In [17]: letters
Out[17]: ['a', 'b', 'c', 'd', 'e', 'f', 'g']

In [18]: letters[2:5] = ['C', 'D', 'E']   # replace some values
In [19]: letters
Out[19]: ['a', 'b', 'C', 'D', 'E', 'f', 'g']
In [20]: letters[2:5] = []                # now remove them
In [21]: letters
Out[21]: ['a', 'b', 'f', 'g']

In [22]: letters[:] = []                  # clear the list by replacing all the
    elements with an empty list
In [23]: letters
Out[23]: []

In [24]: letters = ['a', 'b', 'c', 'd']   # The built-in function len() also
    applies to lists:
In [25]: len(letters)
Out[25]: 4

In [26]: datos = [24, "Alonso", [6.7, 3.6, 5.9]]    # Mixed list (int, str, list)
In [27]: datos
Out[27]: [24, 'Alonso', [6.7, 3.6, 5.9]]

# It is possible to nest lists (create lists containing other lists), for example
    :
In [28]: a = ['a', 'b', 'c']
In [29]: n = [1, 2, 3]
```

```
In [30]: x = [a, n]
In [31]: x
Out[31]: [['a', 'b', 'c'], [1, 2, 3]]
In [32]: x[0]
Out[32]: ['a', 'b', 'c']
In [33]: x[0][1]
Out[33]: 'b'

# We can operat with list in similar way as strings:
In [34]: lts = ['a', 'b', 'c', 1, 2, 3]
In [35]: print(lts*3)
['a', 'b', 'c', 1, 2, 3, 'a', 'b', 'c', 1, 2, 3, 'a', 'b', 'c', 1, 2, 3]
```

1.6.4 Estructura de datos: tuplas

Las tuplas son listas que no pueden modificarse. Se definen al igual que las listas pero usando paréntesis en lugar de corchetes:

Tuplas

```
In [1]: a = ('a', 'b', 'c', 1, 2, 3)
In [2]: type(a)
Out[2]: builtins.tuple

In [3]: a = 'a', 'b', 'c', 1, 2, 3    # We can also avoid parentheses
In [4]: type(a)
Out[4]: builtins.tuple

In [5]: t=a*3                         # We can operate them as in lists
In [6]: t
Out[6]: ('a', 'b', 'c', 1, 2, 3, 'a', 'b', 'c', 1, 2, 3, 'a', 'b', 'c', 1, 2, 3)
In [8]: t[2]
Out[8]: 'c'

In [9]: t[2]='x'                      # But it cannot be modified
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-cd1e1201d2c0> in <module>()
----> 1 t[2]='x'
TypeError: 'tuple' object does not support item assignment
```

1.6.5 Estructura de datos: diccionarios

Los diccionarios son listas en las que cada elemento se identifica no con un supuesto índice, sino con un nombre o clave, por lo que siempre se usan en parejas clave-valor separadas por ":". La clave va primero y siempre entre comillas y luego su valor, que puede ser en principio cualquier tipo de dato de Python; cada pareja clave-valor se separa por comas y todo se encierra entre llaves. Por ejemplo, podemos crear un diccionario con los datos básicos de una persona:

Diccionarios

```
# We create a key "Nombre" con with value "Juan", other with key "Apellido" and
    value "Martnez", etc.
In [1]: datos = {'Nombre': 'Juan', 'Apellido': 'Martinez', 'Edad': 21, 'Altura':
    1.67}
In [2]: type(datos)
Out[2]: builtins.dict
In [3]: print(datos['Nombre'])     # We can access to the values using the keys
Juan
In [4]: datos.keys()               # We can list all the keys in the dictionary
Out[4]: dict_keys(['Apellido', 'Edad', 'Nombre', 'Altura'])
In [5]: datos.values()             # We can list all the values in the dictionary
    in the same order given by datos.keys()
Out[5]: dict_values(['Martinez', 21, 'Juan', 1.67])
```

1.6.6 Estructura de datos: arrays

Los arrays son similares a las listas pero con la diferencia de que podemos realizar operaciones matemáticas sobre ellas. Pueden considerarse como vectores o matrices. Para usar arrays, debemos importar el módulo numpy:

Arrays - Defición

```
In [1]: import numpy as np
In [2]: x = np.array([2.0, 4.6, 9.3, 1.2])      # create directly an array
In [3]: notas = [ 9.8, 7.8, 9.9, 8.4, 6.7]      # create a list
In [4]: notas = np.array(notas)                 # create an array from a list
In [5]: numeros = np.arange(10.)                # create automatically an array of
     floats from 0 to 9.
In [6]: print(numeros)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
In [7]: print(numeros*3)                        # we can operate with arrays
[  0.   3.   6.   9.  12.  15.  18.  21.  24.  27.]
```

Los arrays se indexan prácticamente igual que las listas y las cadenas de texto; veamos algunos ejemplos:

Arrays - Indexado

```
In [8]: print(numeros[3:8]) # Elements from 4th to 8th (indexes 3 to 7). Note
    that the last index is not included
[ 3.  4.  5.  6.  7.]
In [9]: print(numeros[:4])  # Elements from 1st to 4th (indexes 0 to 3)
[ 0.  1.  2.  3.]
In [10]: print(numeros[5:])  # Indexes from 5 (included) up to the end.
[ 5.  6.  7.  8.  9.]
In [11]: print(numeros[-3])  # 3rd index starting from the end.
7.0
In [12]: print(numeros[:])   # All the elements of the array
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
In [13]: print(numeros[2:8:2]) # Elements from index 2 until 7 but taken by 2
[ 2.  4.  6.]
```

Al igual que las listas, podemos ver el tamaño de un array unidimensional con len(), aunque la manera correcta de conocer la forma de un array es usando el método shape():

Arrays - Propiedades

```
In [14]: print(np.shape(numeros))  # The result is one element only since the
    array is unidimensional.
(10,)
In [15]: enteros = np.arange(6)
In [16]: print(enteros)
[0 1 2 3 4 5]
In [17]: type(enteros)             # Variable type
Out[17]: numpy.ndarray
In [18]: type(enteros[0])          # element type within the variable.
Out[18]: numpy.int64
In [19]: decimales=enteros.astype('float')   # the type of the elements within
    the variable can be changed
In [20]: print(decimales)
[ 0.  1.  2.  3.  4.  5.]
In [21]: type(decimales)           # but the variable does not change.
Out[21]: numpy.ndarray
In [22]: type(decimales[0])
Out[22]: numpy.float64
```

Podemos hacer operaciones básicas con arrays:

Arrays - Operaciones

```
In [38]:decimales=[ 0. 1. 2. 3. 4. 5.]
In [39]: x = np.array([5.6, 7.3, 7.7, 2.3, 4.2, 9.2])
In [40]: print(x+decimales)                                  # Sum
[ 5.6 8.3 9.7 5.3 8.2 14.2]

In [41]: print(x*decimales)                                  # Multiplication
[ 0. 7.3 15.4 6.9 16.8 46. ]
In [42]: print(x/decimales)                                  # Division
[ Inf 7.3 3.85 0.76666667 1.05 1.84 ]

In [44]: z = concatenate((x, decimales))                     # Concatenate
In [45]: print(z)
[ 5.6 7.3 7.7 2.3 4.2 9.2 0. 1. 2. 3. 4. 5. ]
In [46]: z = concatenate((x,[7]))
In [47]: print(z)
[ 5.6 7.3 7.7 2.3 4.2 9.2 7. ]

In [5]: z.max()                                              # Maximun value of the
   elements within an array
(alternatively: max(z))

Out[5]: 9.1999999999999993
In [6]: z.min()                                              # Minimun value
Out[6]: 2.2999999999999998
In [7]: z.mean()                                             # Mean value
Out[7]: 6.1857142857142851
In [8]: z.std()                                              # Standard deviation
Out[8]: 2.1603098795103919
In [9]: z.sum()                                              # Sum of all elements of
   the array
Out[9]: 43.299999999999997
In [16]: median(z)                                           # Median
Out[16]: 7.0
```

Podemos usar los arrays como datos lógicos (*True o False*)

Arrays - Lógicos

```
In [19]: A = np.array([True, False, True])
In [20]: B = np.array([False, False, True])
In [22]: A*B
Out[22]: array([False, False, True], dtype=bool)
In [29]: C = array([1, 2, 3])
In [30]: A*C
Out[30]: array([1, 0, 3])
In [31]: B*C
Out[31]: array([0, 0, 3])

# Numerical or logical indexes can be used in arrays:
In [1]: mi_array = np.arange(0,100,10) # Array with integers from 0 to 9
In [2]: print(mi_array)
[ 0 10 20 30 40 50 60 70 80 90]
In [3]: indices1 = np.arange(0,10,2)   # Array with integers from 0 to 9 by steps
    of 2
In [4]: print(indices1)                # It can be used to select elements
[0 2 4 6 8]
In [5]: indices2 = np.array([False, True, True, False, False, True, False, False,
    False, True]) # The boolean indexes start from element 1 until the end.
In [6]: print(mi_array[indices1])
[ 0 20 40 60 80]
In [7]: print(mi_array[indices2])
[10 20 50 80 90]

# An array can be created using a boolean operator
In [8]: mayores50 = mi_array > 50
In [9]: print(mayores50)
[False False False False False False True True True True]
# It can be used to select those elements that match the condition
In [10]: print(mi_array[mayores50])
[60 70 80 90]
```

Arrays multidimensionales: numpy permite trabajar con arrays de dimensiones superiores a 1.

Arrays - Multidimensionales

```
# 3 rows and 3 columns array
In [56]: arr0 = np.array([[10,20,30],[9, 99, 999],[0, 2, 3]])
In [57]: print(arr0)
[[ 10 20 30]
[ 9 99 999]
[ 0 2 3]]
# Zeros with 2 rows and 3 columns
In [57]: arr1 = np.zeros((2,3))
In [59]: print(arr1)
[[ 0. 0. 0.]
[ 0. 0. 0.]]
# Ones array with 4 rows and 1 column
In [62]: arr2 = np.ones((4,1))
In [63]: print(arr2)
[[ 1.]
[ 1.]
[ 1.]
[ 1.]]
# Unidimensional array  of 9 elements. We can change the shape to a 3x3 array
In [64]: arr3 = np.arange(9).reshape((3,3))
In [65]: print(arr3)
[[0 1 2]
[3 4 5]
[6 7 8]]
In [69]: arr2.shape
Out[69]: (4, 1)
In [70]: arr3.shape
Out[70]: (3, 3)

# To access the elements in an array, we have to use two indexes [r,c]: r is the
    row, c the column:
# The first row and column element in the array is then [0,0]
In [86]: arr0[0,0]
Out[86]: 10
# All the rows of the first column:
In [87]: arr0[:,0]
Out[87]: array([10, 9, 0])
# All the columns of the first row:
In [88]: arr0[0,:]
Out[88]: array([10, 20, 30])
# First and second column of first row
In [89]: arr0[0,:2]
Out[89]: array([10, 20])
```

```
# We can also work with multidimensional arrays using indexes:
# Change first element
In [91]: arr0[0,0] = 88
# Change first and second column of the second row
In [92]: arr0[1,:2] = [50,60]
# Multiply by 10 la last row
In [93]: arr0[-1,:] = arr0[-1,:]*10
In [94]: print(arr0)
array([[ 88,  20,  30],
[ 50,  60, 999],
[  0,  20,  30]])
```

1.7 Lectura y escritura de ficheros.

Podemos leer y escribir ficheros de forma sencilla:

Creando ficheros

```
In [49]: fs = open('prueba.txt', 'w')
In [50]: type(fs)
Out[50]: <type 'file'>
In [48]: fichero_leer = open('mi_fichero.txt', 'r')     # Open the file
    mi_fichero.txt for reading (must exist)
In [49]: fichero_escribir = open('mi_fichero.txt', 'w') # Open the file
    mi_fichero.txt for writting. Overwrite if it exists, create if not.
In [50]: fichero_escribir = open('mi_fichero.txt', 'a') # Open the file
    mi_fichero.txt to add some text. If exist, add text to the end of the file. If
     not, it is created. escribiendo al final del fichero.
In [51]: fichero_leer = open('mi_fichero.txt', 'rw')    # Open the file
    mi_fichero.txt for reading and writting

# Example:
In [52]: fs = open('prueba.txt', 'w')
In [53]: type(fs)
Out[53]: <type 'file'>
In [54]: fs.write('Hola, estoy escribiendo un texto a un fichero')
In [55]: fs.close()                    # Close the file, necessary to write
    within.
```

Formato en los ficheros

```
In [55]: fs = open('prueba.txt', 'a')
In [56]: fs.write("\n\n")                       # Dejo dos lineas en blanco
In [57]: fs.write("Esta es una linea nueva\n")
In [58]: fs.write("Y esta es otra linea\n")
In [59]: fs.close()
```

Para escribir ficheros de de datos:

Crear fichero de datos

```
In [60]: from numpy import arange
In [61]: from numpy import exp
In [62]: fsalida = open('datos.txt', 'w')
In [63]: for i in arange(100.):
....:         fsalida.write('%d %10.4f\n' % (i, exp(i))) # Esto escribe un fichero
    con dos columnas y 100 filas.
.....:

In [63]: fsalida.close()
```

Para leer ficheros de datos:

Leer datos

```
In [1]: fdatos = open('datos.txt', 'r')  # Abrimos el fichero "datos.txt" para
   lectura
In [2]: x_datos = []                     # Creamos una lista para la primera
   columna
In [3]: y_datos = []                     # Creamos una lista para la segunda
   columna
In [4]: for linea in fdatos:
....: x, y = linea.split()                # Se separa cada linea en dos columnas
   tomando como separador los espacios en blanco. Para separar, por ejemplo, por
   comas: split(',')
....: x_datos.append(float(x))            # A~nado el elemento x a la lista
   x_dato
....: y_datos.append(float(y))            # A~nado el elemento y a la lista
   y_dato
....:
In [5]: fdatos.close()
In [6]: print(x_datos)
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0,
   15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
   28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0,
   41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0, 50.0, 51.0, 52.0, 53.0,
   54.0, 55.0, 56.0, 57.0, 58.0, 59.0, 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0,
   67.0, 68.0, 69.0, 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0, 78.0, 79.0,
   80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0, 88.0, 89.0, 90.0, 91.0, 92.0,
   93.0, 94.0, 95.0, 96.0, 97.0, 98.0, 99.0]

# Usando este modo de escritura y lectura, los datos siempre se escriben y leen
   como {\it str} y por lo tanto, x_datos e y_datos son de tipo {\it lust}
In [7]: type(x_datos)
Out[8]: builtins.list
```

```
# Si queremos operar con x_datos e y_datos, debemos convertirlos a un tipo num\'
    erico:
In [9]: x_datos, y_datos = array(x_datos), array(y_datos)
In [10]: type(x_datos)
Out[11]: numpy.ndarray
In [12]: x_datos.shape
Out[13]: (100,)
```

Una manera alternativa y más sencilla de escribir y leer arrays es usando los métodos savetxt() y loadtxt() de numpy, que escriben y leen ficheros de texto pero con un formato prefijado por defecto.:

Escritura y lectura de datos

```
# Guardamos en el fichero "datos2.txt" con dos filas que contienen los arrays
    x_datos e y_datos
In [1]: from numpy import arange
In [2]: from numpy import exp
In [3]: from numpy import savetxt
In [4]: fsalida = open('datos2.txt', 'w')
In [5]: x_datos=arange(100.)
In [6]: y_datos=exp(x_datos)
In [7]: savetxt('datos2.txt', (x_datos, y_datos)) # Esto almacena x_datos en la
    primera fila del fichero 'datos2.txt' e y_datos en la segunda fila
In [8]: x, y = loadtxt("datos2.txt") # Leemos el fichero que acabamos de crear y
    almacenamos los arrays en x e y
In [9]: type(x)
Out[10]: numpy.ndarray
In [11]: type(y)
Out[12]: numpy.ndarray

# Tambien podemos cargar los datos en una sola variable, un array de dos
    dimensiones en este caso:
In [13]: d=loadtxt('datos2.txt')
In [14]: shape(d)
Out[15]: (2, 100)

# loadtxt lee correctamente cualquier fichero con diferente estructura, por
    ejemplo un fichero de 5 filas y 3 colunas con cabecera,
tal como el fichero datos3.txt:
# tiempo x1 x2
1.0 1.2 1.4
2.0 2.1 2.3
3.0 3.3 3.0
4.0 4.2 4.2
5.0 5.1 5.1

# Cualquier linea que comienze por el caracter comentario estandard '#' es
    ignorada:
```

```
In [16]: ddd=loadtxt('datos3.txt')
In [17]: shape(ddd)
Out[18]: (5, 3)
In [19]: type(ddd)
Out[20]: numpy.ndarray
```

Una función muy útil antes de empezar a programar: magic function %cpaste:

Para ahorrar en la introducción de código en ipython

```
# If we have some code written in a .txt file, it can be paste into ipython
    console using %cpase:

In [41]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:arr0 = np.array([[10,20,30],[9, 99, 999],[0, 2, 3]])   # here we just copy and
    paste our code from other source.
:print(arr0)
:--
[[ 10  20  30]                                          # remember to introduce
    "--"  to indicate the end of the paste code.
 [  9  99 999]
 [  0   2   3]]
```

1.8 Funciones definidas por el usuario

Además de importar módulos ya desarrollados, podemos crear nuestros propios módulos/funciones y usarlo como funciones. Por ejemplo, podemos crear el fichero fibo.py que contiene lo siguiente:

Creando funciones 1

```
In [1]: from math import *      # import all the functions of math module

In [2]: def esfera(r):          # Define a function to calculate the parameters of
     a sphere
   ...:     pir = pi*r          # pir is a local variable, it is not returned
    after the end of the function.
   ...:     lon = 2.*pir
   ...:     sup = 4.*pir*r
   ...:     vol = superficie*r/3.
   ...:     return [lon,sup,vol]   # results are given in a list
   ...:

In [3]: l,s,v=esfera(3.215)   # Evaluate the function for r=3.215

In [5]: print('The length of the maximun circle is: %.3f' % l)   # print the
    resutl
```

```
The length of the maximun circle is: 20.200

In [6]: print('The area of the sphere is: %.3f' % s)
The area of the sphere is: 129.889

In [7]: print('The volume of the sphere is: %.3f' % v)
The volume of the sphere is: 139.198
```

Creando funciones 2

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Ahora arrancamos python y cargamos el módulo que hemos creado:

Creando funciones / módulos de usuario

```
>>> import fibo               # Load fibo module
>>> fibo.fib(100)             # Write Fibonacci serie up to 100 (but it cannot be
    assigned to any variable)
1 1 2 3 5 8 13 21 34 55 89
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]  # fibo2 gives the Fibonacci serie in list
    format, so it can be used as variable
>>> k=fibo.fib2(100)
>>> print(k)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

# If the function has to be used ofen, we can assign it a local name:
>>> fib2=fibo.fib2
>>> fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

1.8.1 Funciones recursivas

Una de las posibilidades de python es la de usar funciones recursivas, es decir, funciones que se llaman a sí mismas. Por ejemplo:

Funciones recursivas: Fibonacci

```
In [24]: def fib(n):
    ....:         if n == 1:
    ....:                 return 1
    ....:         elif n == 0:
    ....:                 return 0
    ....:         else:
    ....:                 return fib(n-1) + fib(n-2)
    ....:
In [25]: fib(10)
Out[26]: 55
```

Funciones recursivas: Factorial

```
In [29]: def fac(n):
    ....:     if n < 1:
    ....:         return 1
    ....:     return n*fac(n-1)
    ....:

In [30]: fac(0)
Out[30]: 1
In [31]: fac(6)
Out[31]: 720
```

1.9 Representación gráfica.

1.9.1 Conceptos básicos

Para representar figuras usamos el módulo matplotlib

Figuras

```

In [1]: import matplotlib          # Import module
In [2]: matplotlib.use('Qt4Agg')   # I we are in KDE desktop, we have to use the
    Qt4Agg backend, if not TK backend is load by default.
In [3]: from pylab import *        # Pylab alows to use several functions to plot
     figures
In [4]: ion()                      # Plots are interactive (each plot is shown
    immediately)
In [5]: x=arange(10.)
In [6]: plot(x)                    # plotting x
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x7f366c05b350>]

In [8]: clf() # clear the figure
In [9]: x2=x**2
In [10]: x3=x**3
In [11]: plot(x, x,'b.', x, x2,'rd', x, x3,'g^') # plotting three curves in the
    same figure
```

La función plot puede incluir varias opciones para el dibujo de los datos: tipo de marca, anchura de línea, etc. Estas opciones son:

Color in figures		Lines and markers	
Symbol	Color	Symbol	Description
'b'	blue	'-'	solid line
'g'	green	'_'	dashed line
'r'	red	'-.'	dot-dashed line
'c'	cyan	':'	dot line
'm'	magenta	'.'	dot
'y'	yellow	','	Pixel
'k'	black	'o'	solid circle
'w'	white	'v'	down triangle
		'^'	up triangle
		'<'	left triangle
		'>'	right triangle
		's'	square
		'p'	Pentagon
		'*'	star
		'+'	cross
		'x'	X
		'D'	diamond
		'd'	slim diamond

Parameter	Short form	Values
alpha		float (0.0=transparent to 1.0=opaque)
color	c	matplotlib color
label		string
markeredgecolor	mec	matplotlib color
markeredgewidth	mew	float (in points)
markerfacecolor	mfc	matplotlib color
markersize	ms	float (in points)
linestyle	ls	'–' '_' '–.' ':' 'None'
linewidth	lw	float (in points)
marker		'+' '*' ',' '.' '1' '2' '3' '4' '<' '>'
		'D' 'H' '^' '_' 'd' 'h' 'o' 'p' 's' 'v' 'x' '|'
		TICKUP TICKDOWN TICKLEFT TICKRIGHT

Un ejemplo de como usar estas opciones sería:

Figuras

```
In [13]: plot(x, lw=5, c='y', marker='o', ms=10, mfc='red')

# How to change the axis limits:
In [12]: xlim(-1,11) # New limits in OX
Out[12]: (-1, 11)
In [13]: ylim(-50,50) # New limits in OY
Out[13]: (-50, 50)

# The properties of the plots can be changed if we asign a variable to them:
In [16]: p1, p2, p3, = plot(x, x,'b.',x, x2, 'rd', x, x3, 'g^')
In [17]: p1.set_marker('o')  # marker type change in plot 1
In [18]: p3.set_color('y') # color change in plot 3
In [19]: show()   # show the changes
In [20]: clf()   # clean figure
```

1.9.2 Texto en las figuras

Las funciones más importantes para añadir texto a las figuras son:

Figuras

```
In [9]: x = arange(0, 5, 0.05)
In [10]: p, = plot(x,log10(x)*sin(x**2))
In [12]: xlabel('Eje X')  # OX Label
In [13]: ylabel('Eje Y')  # OY Label
In [14]: title('Mi grafica')  # Title
In [15]: text(1, -0.4, 'Nota')  # Title at (1, -0.4)

# It is possible to use Latex in the figures:
In [29]: x = arange(0, 6*pi,0.1)
In [30]: y1 = sin(x)/x
In [31]: y2 = sin(x)*exp(-x)
In [32]: p1, p2, = plot(x, y1, x, y2)
In [33]: texto1 = text(2, 0.6, r'$\frac{\sin(x)}{x}$', fontsize=20)
In [34]: texto2 = text(13, 0.2, r'$\sin(x) \cdot e^{x}$', fontsize=16)
In [35]: grid() # Grid in the figure
In [36]: title('Representacion de dos funciones')
In [37]: xlabel('Tiempo / s')
In [38]: ylabel('Amplitud / cm')
```

1.9.3 Representación de funciones

Las funciones pueden representarse definiendo en primer lugar la función y luego generando un array con el intervalo de la variable independiente que se quiere representar:

Representación de funciones

```
In [30]: def f1(x):
    ....:     y = sin(x)
    ....:     return y
    ....:
In [31]: def f2(x):
    ....:     y = sin(x)+sin(5.0*x)
    ....:     return y
    ....:
In [32]: def f3(x):
    ....:     y = sin(x)*exp(-x/10.)
    ....:     return y
    ....:
In [33]: x = arange(0, 10*pi, 0.1)  # array de valores que quiero representar
In [34]: p1, p2, p3, = plot(x, f1(x), x, f2(x), x, f3(x))
In [35]: legend( ('Funcion 1', 'Funcion 2', 'Funcion 3') )   # Add legend
In [36]: xlabel('Tiempo / s')
In [37]: ylabel('Amplitud / cm')
In [38]: title('Representacion de tres funciones')
```

1.9.4 Comandos predefinidos para representación gráfica

Matplotlib contiene una larga lista funciones para crear figuras:

Function	Description
acorr	Plot the autocorrelation of x.
annotate	Create an annotation: a piece of text referring to a data point.
arrow	Add an arrow to the axes.
autoscale	Autoscale the axis view to the data (toggle).
axes	Add an axes to the figure.
axhline	Add a horizontal line across the axis.
axhspan	Add a horizontal span (rectangle) across the axis.
axis	Convenience method to get or set axis properties.
axvline	Add a vertical line across the axes.
axvspan	Add a vertical span (rectangle) across the axes.
bar	Make a bar plot.
barbs	Plot a 2-D field of barbs.
barh	Make a horizontal bar plot.
box	Turn the axes box on or off.
boxplot	Make a box and whisker plot.
broken_barh	Plot horizontal bars.
cla	Clear the current axes.

clabel	Label a contour plot.
clf	Clear the current figure.
clim	Set the color limits of the current image.
close	Close a figure window.
cohere	Plot the coherence between x and y.
colorbar	Add a colorbar to a plot.
contour	Plot contours.
contourf	Plot contours.
csd	Plot cross-spectral density.
delaxes	Remove an axes from the current figure.
draw	Redraw the current figure.
errorbar	Plot an errorbar graph.
eventplot	Plot identical parallel lines at specific positions.
figimage	Adds a non-resampled image to the figure.
figlegend	Place a legend in the figure.
figtext	Add text to figure.
figure	Creates a new figure.
fill	Plot filled polygons.
fill_between	Make filled polygons between two curves.
fill_betweenx	Make filled polygons between two horizontal curves.
findobj	Find artist objects.
gca	Return the current axis instance.
gcf	Return a reference to the current figure.
gci	Get the current colorable artist.
get_figlabels	Return a list of existing figure labels.
get_fignums	Return a list of existing figure numbers.
grid	Turn the axes grids on or off.
hexbin	Make a hexagonal binning plot.
hist	Plot a histogram.
hist2d	Make a 2D histogram plot.
hlines	Plot horizontal lines.
hold	Set the hold state.
imread	Read an image from a file into an array.
imsave	Save an array as in image file.
imshow	Display an image on the axes.
ioff	Turn interactive mode off.
ion	Turn interactive mode on.
ishold	Return the hold status of the current axes.
isinteractive	Return status of interactive mode.
legend	Place a legend on the current axes.
locator_params	Control behavior of tick locators.
loglog	Make a plot with log scaling on both the x and y axis.
margins	Set or retrieve autoscaling margins.
matshow	Display an array as a matrix in a new figure window.
minorticks_of	Remove minor ticks from the current plot.
minorticks_on	Display minor ticks on the current plot.
over	Call a function with hold(True).

pause	Pause for interval seconds.
pcolor	Create a pseudocolor plot of a 2-D array.
pcolormesh	Plot a quadrilateral mesh.
pie	Plot a pie chart.
plot	Plot lines and/or markers to the Axes.
plot_date	Plot with data with dates.
plotfile	Plot the data in in a file.
polar	Make a polar plot.
psd	Plot the power spectral density.
quiver	Plot a 2-D field of arrows.
quiverkey	Add a key to a quiver plot.
rc	Set the current rc params.
rc_context	Return a context manager for managing rc settings.
rcdefaults	Restore the default rc params.
rgrids	Get or set the radial gridlines on a polar plot.
savefig	Save the current figure.
sca	Set the current Axes instance to ax.
scatter	Make a scatter plot of x vs y, where x and y are sequence like objects of the same lengths.
sci	Set the current image.
semilogx	Make a plot with log scaling on the x axis.
semilogy	Make a plot with log scaling on the y axis.
set_cmap	Set the default colormap.
setp	Set a property on an artist object.
show	Display a figure.
specgram	Plot a spectrogram.
spy	Plot the sparsity pattern on a 2-D array.
stackplot	Draws a stacked area plot.
stem	Create a stem plot.
step	Make a step plot.
streamplot	Draws streamlines of a vector flow.
subplot	Return a subplot axes positioned by the given grid definition.
subplot2grid	Create a subplot in a grid.
subplot_tool	Launch a subplot tool window for a figure.
subplots	Create a figure with a set of subplots already made.
subplots_adjust	Tune the subplot layout.
suptitle	Add a centered title to the figure.
switch_backend	Switch the default backend.
table	Add a table to the current axes.
text	Add text to the axes.
thetagrids	Get or set the theta locations of the gridlines in a polar plot.
tick_params	Change the appearance of ticks and tick labels.
ticklabel_format	Change the ScalarFormatter used by default for linear axes.
tight_layout	Automatically adjust subplot parameters to give specified padding.
title	Set a title of the current axes.
tricontour	Draw contours on an unstructured triangular grid.
tricontourf	Draw contours on an unstructured triangular grid.

tripcolor	Create a pseudocolor plot of an unstructured triangular grid.
triplot	Draw a unstructured triangular grid as lines and/or markers.
twinx	Make a second axes that shares the x-axis.
twiny	Make a second axes that shares the y-axis.
vlines	Plot vertical lines.
xcorr	Plot the cross correlation between x and y.
xkcd	Turns on xkcd sketch-style drawing mode.
xlabel	Set the x axis label of the current axis.
xlim	Get or set the x limits of the current axes.
xscale	Set the scaling of the x-axis.
xticks	Get or set the x-limits of the current tick locations and labels.
ylabel	Set the y axis label of the current axis.
ylim	Get or set the y-limits of the current axes.
yscale	Set the scaling of the y-axis.
yticks	Get or set the y-limits of the current tick locations and labels.

1.10 Control de flujo: instrucciones compuestas

Un programa consiste en una serie de sentencias que se ejecutan una detrás de otra. A veces será necesario cambiar esta secuencia de ejecución de manera que nuestro programa ejecute un u otra parte del código según ciertas condiciones. Para ello se utilizan las sentencias de control de flujo o instrucciones compuestas, que nos permiten interactuar con el programa, tomar decisiones, saltar hacia adelante o hacia atrás y ejecutar sentencias en un orden diferente al natural. Son un elemento básico de cualquier lenguaje programación. Vamos a ver aquí los mś importantes.

1.10.1 *if*

Es la más conicida. Ejecuta una parte del código bajo determinada condición.

instrucción *if*

```
In [1]: x = int(input("Please enter an integer: "))
Please enter an integer: 42
:if x < 0:     # check if it is negative.
:      x = 0
:      print('Negative changed to zero')
:elif x == 0:        # if not, check if it is =0
:      print('Zero')
:elif x == 1:         # if not, check if it is =1
:      print('Single')
:else:                    # If nothing of the previous requirements, do next:
:      print('More')!
More
```

Puede omitirse en este caso *else*. La palabra clave *elif* es una contracción usada para *else if* y es útil para evitar una identicación excesiva.

1.10.2 *For*

Realiza una misma acción o serie de acciones o sentencia varias veces. Una manera habitual de susar el for es para recorrer con una variable cada uno de los elementos de una lista. Una manera básica de usarla en un programa de Python sería

instrucción *for*

```
In [4]: words = ['cat', 'window', 'defenestrate']
In [5]: for w in words:          # w takes all the values in 'words'
   ...:     print(w, len(w))
   ...:
cat 3
window 6
defenestrate 12
```

Si se necesita modificar la variable o lista sobre la que se está inteactuando mientras se está dentro del loop, es recomendable hacer primero una copia:

instrucción *for*

```
In [6]: for w in words[:]:  # The loop works in a copy of the 'words' list no
   directly in 'words'
   ...:     if len(w) > 6:
   ...:         words.insert(0, w)
   ...:

In [7]: words
Out[7]: ['defenestrate', 'cat', 'window', 'defenestrate']
```

1.10.3 *While*

while se usa para repetir una instrucción mientras esta sea válida:

instrucción *while*

```
In [3]: cuentas = 0

In [4]: while cuentas < 6:
   ...:     print(cuentas)
   ...:     cuentas = cuentas + 1  # cuentas is increased if match the condition
   ...:
0
1
2
3
4
5
```

Podemos utilizar el bucle while con una condición negativa usando *while* not de la forma siguiente:

instrucción *while not*

```
In [6]: while not x == 5:
   ...:     x = x + 1
   ...:     print("x = %d" % x)
   ...:
x = 1
x = 2
x = 3
x = 4
x = 5
```

Hay que tener cuidado cuando se hace una comparación entre números *float* mediante una igualdad exacta. Debido a la precisión finita de los ordenadores, es posible que una determinada igualdad nunca se cumpla exactamente y por lo tanto la ejecución del bucle nunca se detendrá:

instrucción *while* comparando *flt*

```
In [9]: x = 0.0

In [10]: while not x == 1.0:
   ....:     x = x + 0.1
   ....:     print("x = %19.17f" % x)
   ....:
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
x = 0.50000000000000000
x = 0.59999999999999998
x = 0.69999999999999996
x = 0.79999999999999993
x = 0.89999999999999991
x = 0.99999999999999989  # Loop should end here, but it continues
x = 1.09999999999999987
x = 1.19999999999999996
x = 1.30000000000000004
...
```

1.10.4 *Break*, *Continue* and *Else* en bucles

Los bucles pueden contener un comando *else* que se ejecuta cuando el bucle termina al pasar por todos los elementos de una lista o bien cuando la condición a la que se refiere es falsa (por ejemplo, con *while*). Si usamos *brake* el bucle puede romperse antes de que se agote la lista sobre la que actúa:

instrucción *break*

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

continue se usa para continuar con la siguiente iteración en un bucle:

instrucción *continue*

```
In [8]: for num in range(2, 10):
   ...:     if num % 2 == 0:
   ...:         print("Found an even number", num)
   ...:         continue
   ...:     print("Found a number", num)
   ...:
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

1.11 Bibliografía

http://docs.python.org/3/index.html

http://matplotlib.org/