
Computación Científica

Release 2012-10-25

Jorge A Pérez Prieto y Teodoro Roca Cortés

October 25, 2012

Introducción a la programación con Python

Para crear un algoritmo para resolver un problema matemático, necesitamos conocer un lenguaje de programación e implementarlo en un ordenador. Un lenguaje de programación, como los lenguajes naturales, son una serie de órdenes que nos permiten “hablar” con un ordenador pero a diferencia de los naturales, los lenguajes de programación son estrictos y no permiten ambigüedades. Existen cientos de lenguajes de programación, cada uno con sus peculiaridades, pero solo con unos pocos se han programado la mayoría de los programas que usamos diario como navegadores, editores de texto, aplicaciones en *smartphones* o muchas páginas web como Wikipedia o Facebook. Entre los lenguajes más usados están C/C++, Java o Python, aunque en ciencia también son muy populares FORTRAN y Matlab, entre otros.

En cualquier caso, el objetivo de un lenguaje de programación es ofrecer un idioma relativamente sencillo con el que dar órdenes a un ordenador; éste, no comprende directamente ningún lenguaje de programación, únicamente el llamado “lenguaje de máquina”, que es propio de cada tipo de ordenador y por tanto el programa, hecho en cualquier lenguaje, debe traducirse a este lenguaje de máquina.

Según cómo se convierte un código o programa a lenguaje de máquina, los lenguajes se clasifican en dos tipos, los compilados, como C/C++ o FORTRAN, los cuales mediante una orden se traduce (compila) completamente el programa una única vez generándose un programa ejecutable que se puede usar cuantas veces queramos. Por otro lado están los lenguajes interpretados como Python, Perl o Ruby, que se van traduciendo y ejecutando línea a línea. Por regla general los programas compilados son más rápidos, pero los interpretados tienen la ventaja de que son mucho más interactivos y flexibles y es mucho más fácil y rápido programar y corregir errores con ellos y por eso se usan mucho en el análisis de datos.

Python, el lenguaje que vamos a usar, es un lenguaje de interpretado que puede utilizarse como un programa ejecutable desde una terminal de comandos o también, de manera interactiva mediante una consola de Python. Python incorpora una consola por defecto, pero también existen otras con características más útiles para el análisis científico de datos. Un ejemplo es **ipython** que se trata

de una consola de Python mejorada, incluyendo completado de funciones y variables, funcionalidad de los comandos básicos de la consola del sistema (*cd*, *ls*, *pwd*, etc.), comandos adicionales (llamados *comandos mágicos*) y un largo etcétera. Esta es la consola de Python que usaremos preferentemente durante este curso.

2.1 Empezando con Python

Python se inicia desde una terminal de comandos estándar de Linux (o Mac, 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:

```
>>> print('Hola, esto es Python')
Hola, esto es Python
```

Para obtener ayuda sobre cómo ejecutar un comando de Python cualquiera, por ejemplo para imprimir por pantalla, basta con escribir lo siguiente:

```
>>> help(print)
Type:
builtin_function_or_method
Base Class:
<type 'builtin_function_or_method'>
String Form:
<built-in function print>
Namespace:
Python builtin
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout)
Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
```

Pero si trabajamos con la consola avanzada `ipython`, podemos hacer lo mismo escribiendo el comando con un símbolo de interrogación al final, por ejemplo: `print?`.

Nota: Cuando iniciamos una sesión de trabajo con `ipython`, suele ser muy útil guardar todo lo que vamos haciendo para usarlo posteriormente (repasar o estudiar lo que se hizo) o continuar a partir de dónde lo dejamos. Para ello se empieza la sesión con el comando `%logstart` con el objeto de grabarla en un fichero, por ejemplo:

```
%logstart -o registro_21_noviembre_2012.txt
```

donde el parámetro opcional `-o` guarda también la salida (respuesta) de python a modo de comentario en el fichero `registro_7_octubre_2011.txt`. Al iniciar `ipython` en otra ocasión, podemos hacer que antes lea el fichero de la sesión que nos interese, escribiendo:

```
ipython -lp registro_21_noviembre_2012.txt
```

con lo que recuperamos la sesión grabada anteriormente y podemos continuar a partir de donde lo dejamos.

2.2 Tipos básicos de datos. Variables

En cualquier lenguaje de programación existen distintos tipos de datos, que podemos almacenar y operar de forma diferente con ellos y que poseen distintas propiedades. Los más comunes son las llamadas **cadenas de texto o string** (indicadas siempre entre comillas) y los **números** (**int** o **float**, enteros o de coma flotante respectivamente):

```
>>> print("Esta es una linea de texto")
Esta es una linea de texto
>>> print(28, 23.459851)      # estos son un entero y un coma flotante
28, 23.459851
```

Las **variables** son componentes fundamentales de un lenguaje de programación y no son más que un nombre que se refiere a un registro en el computador (una serie de bits) que contiene uno o varios datos, que pueden ser de distinto tipo. Veamos unos ejemplos:

```
>>> frase = "Esta es una linea de texto"
>>> num = 22
>>> num*2
44
>>> frase*2
Esta es una linea de textoEsta es una linea de texto
```

Aquí, `frase` es una variable tipo *string* mientras que `num` es otra variable numérica entera. Nótese que mientras se multiplicó `num` por 2 de forma conocida, la cadena de texto `frase` fué duplicada al multiplicarla por 2.

Con el comando `type()` podemos saber el tipo de dato que se asigna a una variable si en cualquier momento no recordamos como la definimos:

```
>>> type(frase)
<type 'str'>
>>> type(num)
<type 'int'>
```

Los nombres de las variables pueden ser cualquier combinación de letras y números (siempre que no empiece por número) pero no están permitidos caracteres especiales como tildes, puntos,

espacios en blanco, etc. Algunas palabras están además reservadas ya por el propio Python para su uso como: `def`, `class`, `return`, etc. por lo que tampoco las podremos usar como nombres de variables.

Nótese que `num` la hemos definido de tipo entero estricto y el **cálculo entre números enteros siempre da como resultado otro número entero**, redondeándose al más cercano en caso de no ser entero exacto, por ejemplo:

```
>>> 113/27  
4           # en lugar de 4.1851851851851851
```

Si queremos usar números decimales, también llamados de coma flotante, debe emplearse directamente un valor decimal (`float`); en estos casos:

```
>>> 113.0/27.0  
4.1851851851851851
```

```
>>> type(113.0/27.0)  
<type 'float'>
```

Como dijimos en el tema anterior, Python emplea números de 64 bits (por defecto) en los `float`. En cualquier operación es muy importante usar los enteros y `float` correctamente y tener cuidado al mezclarlos, de otro modo se obtendrá un resultado no deseado llegando a hacernos creer que el computador se ha equivocado.

Los tipos de datos pueden ser convertidos de unos otros, empleando `str()` para convertir a cadena de texto, `int()` a entero y `float()` a coma flotante:

```
>>> float(3)  
3.0  
  
>>> int(3.1416)  
3  
  
>>> str(34)  
'34'
```

Para el caso de los `float`, se pueden redondear con `round()`, que redondea al entero más próximo. La funciones `ceil()` y `floor()` del módulo o paquete `math` redondean hacia arriba y hacia abajo respectivamente:

```
>>> print(round(4.4)) , (round(4.5))  
4.0 5.0  
>>> # Importo todas las funciones matemáticas del módulo math  
>>> from math import *  
>>> print(ceil(4.4)) , (ceil(4.5))  
5.0 5.0  
  
>>> print(floor(4.4)) , (floor(4.5))
```

4.0 4.0

Más adelante veremos qué son los módulos, que ofrecen otras funciones predefinidas muy interesantes y aprenderemos cómo importarlos y a utilizarlos.

2.3 Operadores aritméticos y lógicos

Con **python** se pueden hacer las operaciones aritméticas habituales usando los símbolos correspondientes:

Operación	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
Exponenciación	**
Residuo o resto	%

La prioridad en la ejecución (de mayor a menor, separados por ;) es la siguiente: **; *, /, %; +, - .:

```
>>> x=3.5
>>> x**2
12.25
>>> x+x**2
15.75
>>> x**(2./3.)
2.3052181460292234
>>> x**(2/3)
1.0
>>> 123%8
3
>>> x+x/2+x*x-x**1.25-x%2
11.212761600464169
```

Los **operadores lógicos** son aquellos operadores que permiten comparar valores entre sí. En concreto se usan:

Operacion	Simbolo
Igual a (comparación)	==
Distinto de (comparación)	!= o <>
Mayor que, Menor que	>, <
Mayor o igual, Menor o igual	>=, <=
y, o	and, or
cierto, falso	True, False

Como resultado de una operación lógica, obtenemos como resultado un elemento, **True** o **False**,

según se verifique o no la operación. Estos elemento lógicos los podemos usar a su vez para otras operaciones. Veamos algunos ejemplos:

```
>>> resultado = 8 > 5
>>> print resultado
True
>>> resultado = (4 > 8) or (3 > 2)
True
>>> resultado = True and False
>>> print resultado
False
>>> resultado = (4 > 8) and (3 > 2)
>>> print resultado
False
```

2.4 Cadenas de texto

Las cadenas de texto, como hemos visto, no son mas que texto formado por letras y números de cualquier longitud y son fácilmente manipulables. Para poder hacerlo cada carácter de una cadena de texto tiene asociado un índice que indica su posición en la cadena, siendo 0 el de la izquierda del todo (primero), 1 el siguiente hacia la derecha y así sucesivamente hasta el último. Veamos ejemplos:

```
>>> # Variable "frase" que contiene una cadena de texto
>>> frase = "Si he logrado ver más lejos, ha sido porque \
    he subido a hombros de gigantes"
>>> print frase[0]          # Primera letra de la cadena
S

>>> print frase[10]         # Decimoprimera letra, con índice 10
a
>>> print len(frase)       # Longitud de la cadena
76

>>> print frase[18:29]      # Sección de la cadena
más lejos

>>> print frase[68:]        # Desde el índice 68 al final
gigantes

>>> print frase[:10]        # Desde el principio al carácter de
Si he logr                                     # índice 10, sin incluirlo
```

También se pueden referir con índices contando desde la derecha, usando índices negativos, siendo -1 el primero por la derecha:

```
>>> print(frase[-1])          # El último carácter, contando desde la derecha
S
>>> print(frase[len(frase)-1]) # El último carácter, contando desde la izquierda
S
>>> print( frase[-1] == frase[len(frase)-1] ) # Compruebo si son iguales
True
```

Existen varios métodos o funciones específicas para tratar y manipular cadenas de texto, éstos nos permiten cambiar de varias maneras las cadenas. Veamos algunos:

```
>>> frase_mayusculas = frase.upper()      # Cambia a mayúsculas y lo guardo en
>>> print(frase_mayusculas)               # la variable frase_mayusculas
SI HE LOGRADO VER MAS LEJOS, HA SIDO PORQUE HE SUBIDO A HOMBROS DE GIGANTES

>>> frase_minusculas = frase.lower()      # Cambia a minúsculas y lo guardo en la var
>>> print(frase_mayusculas)
si he logrado ver más lejos, ha sido porque he subido a hombros de gigantes

>>> frase.replace("hombres", "la chepa") # Reemplaza una cadena de texto por otra
'Si he logrado ver mas lejos, ha sido porque he subido a la chepa de gigantes'
```

Advertencia: Recordar que el índice en las cadenas de texto y en general cualquier lista de números, empieza siempre con el 0, por lo que el primer elemento de una cadena de texto o de una lista es `frase[0]` y no `frase[1]`. Al escribir `frase[10]` estamos tomando el elemento ordinal 11, no el 10.

2.5 Impresión de texto y de números

Las cadenas de texto se pueden concatenar o unir con `+`:

```
>>> "Esta es un frase" + " y esta es otra"
'Esta es un frase y esta es otra'
```

Sin embargo, la concatenación sólo es posible para texto (*string*), por lo que no se pueden concatenar letras y números. Una posibilidad es convertir los números a *string*:

```
>>> a, b = 10, 10**2    # Defino dos numeros, a=10 y b=10**2
>>> print(str(a) + " elevado al cuadrado es " + str(b))
10 elevado al cuadrado es 100
```

Una manera más práctica y correcta de hacer esto es imprimiendo los números con el **formato** que queramos; veamos como hacerlo:

```
>>> # Calculo el logaritmo base 10 de 2 e imprimo el resultado con 50 decimales
>>> print("%.50f") % log10(2.0**100)
30.10299956639811824743446777574717998504638671875000
```

```
>>> # Otro ejemplo usando texto, enteros y decimales
>>> print("El %s de %d es %f.") % ('cubo', 10, 10.**3)
El cubo de 10 es 1000.000000.
```

Aquí se reemplaza cada símbolo `%s` (para cadenas de texto), `%d` (para enteros) o `%f` (para floats) sucesivamente con los valores después de `%` que están entre paréntesis. En caso de los floats se puede utilizar el formato `%10.5f`, que significa imprimir 10 caracteres en total, incluído el punto, usando 5 decimales. Se puede escribir también *floats* en formato científico utilizando `%e`, por ejemplo:

```
>>> print("%.5e" % 0.0003567)
3.56700e-04
```

Los **formatos** son muy útiles a la hora de expresar el resultado de un cálculo con los dígitos significativos solamente o con la indicación del error en el resultado. Así por ejemplo, si el resultado de un cálculo o de una medida es 3.1416 ± 0.0001 podemos expresarlo como:

```
>>> # resultado de un cálculo obtenido con las cifras decimales que proporciona el
>>> resultado = 3.1415785439847501
>>> # este es su error con igual número de cifras decimales
>>> error = 0.0001345610900435
>>> # así expresamos de forma correcta el resultado
>>> print("El resultado del experimento es %.4f +/- %.4f" ) % (resultado, error)
El resultado del experimento es 3.1416 +/- 0.0001
```

2.6 Estructuras de datos. Listas, tuplas y diccionarios

Los datos se pueden almacenar en variables univalueadas como ya hemos visto. No obstante, en **variables estructuradas** también pueden almacenarse uno o más datos. Los tipos de variables estructuradas que ofrece Python son las llamadas **listas**, **tuplas** y **diccionarios** que se definen de la siguiente forma:

2.6.1 Listas

Se trata de un conjunto de números, cadenas de texto u otras listas (aquí tendríamos listas de listas), ordenadas de alguna manera:

```
>>> # Lista de datos string
>>> alumnos = ['Miguel', 'Maria', 'Luisma', 'Fran', 'Luisa', 'Ruyman']

>>> # Lista de enteros
>>> edades = [14, 29, 19, 12, 37, 15, 42]
```

```
>>> # lista de datos mixto(entero, string y lista)
>>> datos = [24, "Juan Carlos", [6.7, 3.6, 5.9]]
```

En el último ejemplo pueden comprobar que es posible mezclar varios tipos de datos, como enteros, *strings* y hasta otras listas. Se puede utilizar la función `len()` para ver el número de elementos de una lista:

```
>>> len(alumnos)
6
```

Es posible utilizar el método `split()` para separar por medio de los espacios una cadena de texto cualquiera y colocar los elementos resultantes en una lista:

```
>>> frase = "Burocracia, su lechuguita."
>>> palabras = frase.split()      # separo la frase por espacios en blanco
>>> print(palabras)
['Burocracia,', 'su', 'lechuguita.'] # el resultado es una lista
>>> palabras2 = frase.split(',')   # separo la frase por comas
>>> print(palabras2)
['Burocracia', ' su lechuguita.']}
```

Como se ve en el ejemplo anterior, el método `split()` separa (por defecto con espacios en blanco) los caracteres de la cadena de texto y da como resultado una lista con los elementos. Se puede usar otro separador, como la coma en este ejemplo, usándolo como parámetro. Nótese que en este caso la coma ya no está en ningún elemento de la lista, ya que ahora se usa de separador.

Existen varias formas de añadir nuevos elementos a una lista existente:

```
>>> alumnos.append('Iballa')      # Añade "Iballa" al final de la lista
>>> print(alumnos)
['Miguel', 'Maria', 'Luisma', 'Fran', 'Luisa', 'Ruyman', 'Iballa']

>>> alumnos.insert(3, 'Jairo')     # Añade "Jairo" en la posición 3
>>> print(alumnos)
['Miguel', 'Maria', 'Luisma', 'Jairo', 'Fran', 'Luisa', 'Ruyman', 'Iballa']

>>> alumnos.index("Jairo")
>>> 3
```

En la última orden del ejemplo anterior hemos usado `index()` para **encontrar el índice o posición** del primer elemento de la lista que vale “Jairo”.

Es posible **ordenar** una lista con el método `sort()`:

```
>>> alumnos.sort()
>>> print(alumnos)
['Fran', 'Iballa', 'Jairo', 'Luisa', 'Luisma', 'Maria', 'Miguel', 'Ruyman']
```

Para **extraer** un elemento de la lista podemos usar los métodos `pop()` y `remove()`:

```
>>> alumnos.pop(2)      # Extraigo el elemento número 2 y lo elimino de la lista  
'Jairo'  
>>> print(alumnos)  
['Fran', 'Iballa', 'Luisa', 'Luisma', 'Maria', 'Miguel', 'Ruyman']  
  
>>> alumnos.remove('Maria')  # Elimino el elemento "Maria" (primera ocurrencia)  
>>> print(alumnos)  
['Fran', 'Iballa', 'Luisa', 'Luisma', 'Miguel', 'Ruyman']
```

Las listas se manipulan de manera similar a las cadenas de texto, utilizando índices que indican la posición de cada elemento siendo **0** el primer elemento de la lista y **-1** el último (si se empieza a numerar por el final):

```
>>> alumnos[2:6]  
['Luisma', 'Fran', 'Luisa', 'Ruyman']  
  
>>> print(alumnos[0], alumnos[-1])  
('Fran', 'Ruyman')
```

Una función muy útil es la función `range()`, que permite crear una lista de números enteros. Por ejemplo, para crear un lista de 10 elementos, de 0 a 9, e imprimirlas podemos hacer esto:

```
>>> print(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Con esta función se puede crear también una lista de números indicando el inicio, final y el intervalo entre dos consecutivos. Por ejemplo, para crear una lista con números enteros de 100 a 200 a intervalos de 20 podemos escribir:

```
>>> print(range(100,200,20))  
[100, 120, 140, 160, 180]
```

Nótese que el último número, 200, no se incluye la lista. La función `range()` se emplea para generar listas de números enteros solamente. Más adelante veremos cómo crear listas similares de *floats*.

Con las variables listas podemos operar de forma parecida a las cadenas de texto, por ejemplo:

```
>>> a=range(100,200,20)  
>>> print(a)  
[100, 120, 140, 160, 180]  
>>> print(a+a)  
[100, 120, 140, 160, 180, 100, 120, 140, 160, 180]  
>>> print(a*3)  
[100, 120, 140, 160, 180, 100, 120, 140, 160, 180, 100, 120, 140, 160, 180]
```

Como se ve en este ejemplo, se pueden unir dos listas usando `+` o incluso multiplicarla por un número entero, que equivale a sumar varias veces la misma lista.

2.6.2 Tuplas: listas inalterables

Las tuplas son listas que no se pueden modificar o alterar y se definen enumerando sus elementos entre paréntesis en lugar de corchetes, por ejemplo:

```
>>> lista_alumnos = ('Miguel', 'Maria', 'Luisma', 'Fran', 'Luisa', 'Ruyman')
```

También podemos definirlas como una variable con varios valores separados por comas; *python* interpreta esto como una tupla aunque no esté entre paréntesis:

```
>>> c = 1, 3          # Defino una variable con dos valores separados por comas
>>> print(c)
>>> (1, 3)
```

Podemos comprobarlo viendo el tipo de dato del que se trata:

```
>>> type(c)
<type 'tuple'>
>>> print(c[0])    # imprimo el primer elemento de la tupla
1
>>> print(c[1])    # imprimo el segundo elemento de la tupla
3
```

Si intentan insertar, quitar o reordenar la tupla con los comandos que ya conocemos para las listas, o en definitiva modificar la tupla verán que es imposible contestando siempre con un error.

2.6.3 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:

```
>>> datos = {'Nombre': 'Juan', 'Apellido': 'Martinez', 'Edad': 21, 'Altura': 1.67}
>>> type(datos)
<type 'dict'>
```

En este caso hemos creado una clave “Nombre” con valor “Juan”, otra clave “Apellido” con valor “Martínez”, etc. Al crear los datos con esta estructura, podemos acceder a los valores de las claves fácilmente:

```
>>> print( datos['Nombre'] )
Juan
```

También podemos conocer todas las claves y los valores de un diccionario usando los métodos `keys()` y `values()` respectivamente:

```
>>> datos.keys()  
['Apellidos', 'Nombre', 'Altura']  
  
>>> datos.values()  
['Martinez', 'Juan', 1.669999999999999]
```

2.7 Módulos y paquetes de Python

Python viene con muchos módulos que ofrecen funcionalidades adicionales muy interesantes. Uno de ellos es el paquete de funciones matemáticas básicas `math`, otro el paquete de utilidades del sistema `sys` y muchos más. Se puede importar un paquete cualquiera haciéndolo implícitamente, o sea importando el paquete completo o bien nombrando una, varias o todas sus funciones; veamos cómo hacerlo:

```
>>> import math                      # importa el paquete math  
>>> import math as M                # importa el paquete math llamándolo M  
>>> from math import sin, cos, pi  # importa las funciones sin, cos y pi de math  
>>> from math import *              # importa todas las funciones de math
```

Podemos ver un listado de las funciones que ofrece un módulo usando la función `dir()`:

```
>>> import math  
>>> dir(math)      # Lista todas las funciones y subpaquete del modulo math  
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',  
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',  
'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',  
'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p',  
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',  
'tanh', 'trunc']
```

Para conocer otros paquetes de la librería estándar pueden consultar el “tutorial” oficial de Python o la guía oficial de la Librería de Python. Por nuestra parte iremos introduciendo más adelante otros paquetes numéricos de Python más avanzados que nos aporta éstas y muchas otras funciones matemáticas muy útiles para la computación científica.

2.8 Ejercicios

1. Calcular a mano cada una de las expresiones siguientes y comprobar, con la ayuda de Python en el ordenador, si el resultado es correcto:

- $16^{**}(1/2)+1/2$
- $16^{**}(1./2)+1/2$

- $3e3/10$
 - $15/5e-3+1$
2. Evaluar en punto flotante las siguientes funciones para los valores de $x=-0.5, 1.1, 2, 3.1$:
- $x^4 + x^3 + 2x^2 - x + 3$
 - $4x \sqrt[3]{|\sin x + \tan x|}$
 - $\frac{x^3 - 5x}{(x^2 + \frac{1}{2})^2}$
3. Calcular a mano cada una de las expresiones siguientes y comprobar, con la ayuda de Python en el ordenador, si el resultado es correcto:
- `int(exp(2*log(3)))`
 - `round(4*sen(3*pi/2))`
 - `abs(log10(0.01)*sqrt(25))`
 - `round(3.21123*log10(1000), 3)`
4. Calculen a mano cada una de las expresiones siguientes y comprueben, con la ayuda de Python en el ordenador, si el resultado es correcto:
- `str(2.1) + str(1.2)`
 - `int(str(5) + str(7))`
 - `int('5' + '7')`
 - `str(5 + 7)`
 - `str(int(83.6) + float(7))`
5. Dada la frase de S. Hawking: “Dios no solo juega a los dados, a veces los tira donde no se pueden ver”, ¿qué funciones o comandos de Python utilizarían para responder o ejecutar lo siguiente?:
- ¿Cuántas letras tiene? ¿y palabras?
 - Pasa a una variable las 15 primeras letras. Pasa a una lista las cinco primeras palabras.
 - ¿Cuántas letras tiene la última palabra?
 - Concatenaen (unan) el primer tercio de la frase con el último tercio.
6. Crear una variable *string* con todas las letras del abecedario en minúsculas juntas, sin incluir la eñe, es decir “abcd...”.
- Buscar la posición en la cadena de texto de todas las vocales.
 - Crea otra variable igual, pero formada por las mayúsculas (es decir “ABCD...”).
 - Usa los índices en cadenas de texto para extraer cada una de las letras necesarias para escribir “Bohr”, concatenándolas (uniéndolas con +).

- Divide la cadena de mayúsculas por la mitad y crea una nueva variable con segunda mitad primero y después la primera, separadas por un guión.
7. Crear una lista con el apellido de diez físicos/as importantes en la historia. Usando métodos e índices de listas hagan lo siguiente:
- Ordenar la lista alfabéticamente.
 - Añadir al final de la lista a Curie si no estaba.
 - Sacar de la lista a Newton (lo habrás puesto, ¿no?).
 - En su lugar incluir en la lista a Chandrasekar (¿o ya lo habían puesto?).
 - Quitar el primer nombre de la lista.
8. Generar una lista de enteros que vaya de 1 a 19, ambos inclusive, de dos en dos. Usando los índices de listas hacer lo siguiente:
- Añadir el 20 a la lista.
 - Invertir el orden de la lista (usa la ayuda de Python si no sabes cómo).
 - Imprimir la lista, salvo el primero y el último.
 - Crear una nueva variable que contenga los cinco primeros elementos de la lista y otra con los cinco penúltimos.
 - Sumar todos los elementos de la lista y divídalo entre su longitud (número de elementos).
9. Crean una variable del tipo *string* con el valor del número π , con al menos seis decimales. Dividan la cadena de texto con el número π en una lista que tenga en el primer elemento la parte entera y en el segundo elemento la parte decimal. Recuerden que sólo las cadenas de texto se pueden separar en listas, no los números.
10. El valor de π se puede obtener del módulo *math* con *pi*. Impriman su valor mostrando sólo tres decimales.
11. Crear una variable que valga 174 (entero) y calcular su raíz cuadrada que almacenan en otra variable. Imprimir ahora una frase que diga “La raíz cuadrada de 174 es 13.1909.” usando las variables creadas junto con el formato de impresión adecuado.
12. Consideremos una lista de python llamada “cosas” que posee distintos tipos de datos:

```
cosas = ['a=', '3.14', ['perro', 'gato', 'liebre', 'cabra'],
         'coche', 1e4, 4]
```

Para esta lista, hacer lo siguiente:

- Contar el número de elementos de la lista
- Calcular el resultado de dividir el segundo elemento por el último.
- Extraer la lista de animales y decir cuántos hay.

- Contar el número de letras en el tercer elemento de la lista de animales.
 - Añadir ‘elefante’ a la lista de animales dentro de la lista “cosas”, ordenándola alfabéticamente.
13. Crean una lista de Python llamada “planetas” con los nombres de los planetas del Sistema Solar hasta Neptuno. Se pide que:
- Utilicen una función de Python para conocer la longitud de la lista (es decir, saber cuantos planetas incluye la lista).
 - Comprueben con algún operador de Python que el último planeta de la lista es Neptuno (¿es cierto o falso?)
 - Añadan al final de la lista los planetas enanos: Pluton, Ceres, Eris, Makemake y Haumea.
 - A partir de esta lista y usando los índices de listas creen tres nuevas listas que incluyan respectivamente: a) Los planetas terrestres (hasta Marte, inclusive), b) los planetas gaseosos (el resto, hasta Neptuno), c) los planetas enanos recién añadidos.
 - Impriman por pantalla todas las listas creadas hasta ahora diciendo que son cada una de ellas.
 - Eliminen la Tierra de la lista original y coloquen en su lugar la Luna.
 - Añadan la Tierra al principio de la lista original.
 - Ordenen alfabéticamente la lista completa de planetas.

Programas ejecutables

4.1 Reutilizando el código

Además de usar **python** de manera interactiva desde una consola, es posible crear programas o *scripts* (guiones) ejecutables; de hecho es la forma más habitual de usarlo. Un programa o *script* es un fichero de texto con los comandos o sentencias de Python escritos de manera consecutiva y que se ejecutarán, paso a paso al ejecutar el *script* desde una consola de comandos del sistema operativo. Bastará utilizar un editor de textos cualquiera (*kate*, *gedit*, *notepad*, etc.), crear un fichero con el nombre que queramos y con extensión **.py** para que éste lo identifique como un programa en Python. En él escribiremos las sentencias o comandos uno detrás de otro, en el orden en que queramos que sean ejecutados.

El siguiente ejemplo es un pequeño programa que llamaremos **cubo.py** que calcula el cubo de un número cualquiera dado por el usuario:

```
#!/usr/bin/python
#-*- coding: utf-8 -*-

# Mensaje de bienvenida
print("Programa de calculo del cubo de un numero.\n")

# Numero de entrada
x = 23.0

# Calculo el valor del cubo de x
y = x**3

# Imprimo el resultado
print("El cubo de %.2f es %.2f" % (x, y))
```

El programa se puede ejecutar ahora desde una consola desde el directorio en donde tenemos el archivo con el programa, escribiendo en ella

```
$ python cubo.py
```

y la respuesta del programa será:

```
Programa de calculo del cubo de un numero.
```

```
El cubo de 23.00 es 12167.00
```

En la primera línea del programa hemos escrito **#!/usr/bin/python** para que no haga falta llamar al intérprete y se pueda ejecutar como un programa normal, aunque para ello se debe tener naturalmente permisos de ejecución. La segunda línea, **#-*- coding: utf-8 -*-** hemos indicado el tipo de codificación UTF-8, para poder poner caracteres especiales como tildes y eñes.

De todas formas, ya que la primera línea indica en qué directorio está el ejecutable de Python, el programa también se puede ejecutar escribiendo únicamente:

```
$ ./cubo.py
```

Esta primera línea depende del sistema operativo que está instalado en el ordenador, que en el ejemplo se supone para Linux (o Mac). En el caso de Windows debemos escribir en la primera línea de programa algo similar a esto:

```
#!C:/Python/python.exe -u # o el lugar en donde resida el  
                          # ejecutable python.exe en Windows
```

pero puede variar según la instalación de Python del ordenador. En todo caso la ejecución de un programa siempre funcionará usando `python cubo.py`.

Hay que fijarse que en el programa muy a menudo hay sentencias *comentadas*, que son las que empiezan con `#`. Estas líneas no son ejecutables y **son ignoradas** por el intérprete de Python; se usan solamente para añadir notas o comentarios. Es muy recomendable incluir comentarios de este tipo que expliquen lo que el programa va haciendo, porque nos ayudarán a recordar qué hace o cómo funciona el programa cuando volvamos a leerlo nosotros, u otros usuarios, y queramos reutilizarlo o modificarlo. Se pueden incluir comentarios de varias líneas, por ejemplo al inicio del programa para explicar lo que hace, usando comillas triples; todo lo que vaya entre ellas será un comentario y por tanto ignorado por el intérprete.:

```
# Este es un comentario de una linea  
print("La raíz cuadrada de 2 es %.2f" % sqrt(2.))  
  
"""Este un comentario que  
usa varias lineas y está limitado  
por comillas triples. Todo lo que esté  
entre ellas no se intenta ejecutar  
"""  
print("El cubo de 2 es %.1f" % 2.0**3)
```

4.2 Definiendo funciones

En capítulos anteriores ya hemos aprendido a usar funciones; algunas de ellas están predefinidas (`abs()`, `round()`, etc.) mientras que otras deben importarse de módulos antes de poder ser utilizadas (`sin()`, `sqrt()` se importan del módulo `math`). No obstante, además de las funciones incluidas en Python, el usuario puede crear las suyas propias, que permiten reutilizar partes del código de un programa sin necesidad de repetirlo varias veces. Las funciones para poder ser utilizadas o llamadas o invocadas, antes hay que definirlas. Veamos por ejemplo, como podemos escribir un programa con una sencilla función para calcular el cubo de un número y otra para imprimir un mensaje:

```
#!/usr/bin/python
#-*- coding: utf-8 -*-

# ahora definimos una función que calcula el cubo de un número cualquiera
def cubo(x):
    y = x**3
    return y

# Ahora utilizamos la función para calcular el cubo de 4
resultado = cubo(4.0)

# imprimo el resultado
print(resultado)
```

Definimos la función `cubo()` que calculará el cubo de un número cualquiera. Nótese que la definición de la función se hace en líneas identadas (con sangría) y acaba con la sentencia `return` para devolver el valor calculado de la función, sin imprimirlo necesariamente. Esto es lo más habitual cuando se crea un función: hacer todas las operaciones necesarias y luego utilizar `return` para devolver un valor. Después de utilizar la función, el resultado puede imprimirse, volcarse a una variable, o lo que el programador necesite. También podemos definir funciones que dependan de varios parámetros (separados por comas); lo único que debemos hacer a la hora de utilizarla después será invocarla con el mismo número de parámetros y los valores que queramos en el mismo orden que en la definición. Alternativamente, podemos definir una función en la que hacemos varios cálculos que queremos que nos devuelva; en este caso, los podemos devolver por medio del `return` en una lista que contenga dichos resultados. Más adelante veremos un ejemplo.

No obstante, no todas las funciones devuelven un valor y en este caso los llamamos **procedimientos**. Veamos un ejemplo interesante, la función que llamamos `saludo()` en el que no calculamos nada y por tanto no devolvemos un valor:

```
#!/usr/bin/python
#-*- coding: utf-8 -*-

# definimos una función que muestra en pantalla un mensaje de saludo
def saludo(nombre):
```

```
print("Buenas tardes %s, " % nombre)

# Utilizamos ahora la función para saludar a Juan
saludo('Juan')
# Imprime: Buenas tardes, Juan
```

En este caso no calculamos nada y por lo tanto, no necesitamos acabar con un `return` la definición del procedimiento. Lo que hacemos es ordenar que se imprima una variable y por lo tanto siempre se imprime una línea cada vez que se llama a la función.

Advertencia: En Python la **identación es obligatoria**, porque se emplea para **separar los bloques** que indican donde empiezan y terminan las funciones (también como veremos más adelante los bucles y condicionales). Si la identación no es correcta, se obtendrá un resultado equivocado o nos devolverá un error de identación.

4.2.1 Variables locales y globales

En la definición de las funciones es posible definir y utilizar variables cuya acción y definición queda circunscrita a la propia función. Son las llamadas *variables locales* que fuera de la función no tienen validez. Veamos un ejemplo. Calculemos el volumen, área superficial y la longitud de cualquier circunferencia máxima en una esfera de radio R definiendo una función que llamamos `esfera()`:

```
#!/usr/bin/python
#-*- coding: utf-8 -*-

# importamos todas las funciones del módulo math
from math import *

# ahora definimos una función que calcula los parámetros de la esfera
def esfera(r):
    pir = pi*r
    longitud = 2.*pir
    superficie = 4.*pir*r
    volumen = superficie*r/3.
    # devolvemos una lista con los resultados
    return [longitud,superficie,volumen]

# Ahora utilizamos la función para el caso de una esfera de radio 3.215
l,s,v = esfera(3.215)

# imprimimos el resultado
print('La longitud del circulo maximo es: %.3f' % l)
print('La superficie de la esfera es: %.3f' % s)
print('El volumen de la esfera es: %.3f' % v)
```

Nótese que en la definición de la función usamos la variable *pir* que es interna o *local*, es decir, sólo se usa dentro de la función ya que no la devolvemos al hacer el return. Por lo tanto, si la quisieramos usar (o saber el valor que tiene) fuera del cuerpo de la función, en el programa principal, no podríamos. Los parámetros de entrada en una función también son variables locales; en este caso *r*. Las demás variables definidas son variables llamadas *globales* ya que las podemos usar fuera del cuerpo de la función. Es conveniente no mezclarlas, aunque podemos hacerlo si tenemos cuidado; es decir, una misma variable podemos usarla fuera y dentro del cuerpo de la función y a pesar de que se llamen igual dentro del cuerpo de la función tendrá el valor que sea (como local) y fuera de él también lo tendrá, como global.

4.3 Entrada de datos por pantalla

Generalmente el valor de una variable se asigna haciendo por ejemplo `pi=3.1416`, pero se puede hacer que pida una entrada por teclado usando la función `raw_input()` de la forma siguiente (pueden probarlo en la consola de python):

```
>>> entrada_numero = raw_input('Dame un numero: ')
Dame un numero:      # Escribimos 22 con el teclado y pulsamos Enter/Intro
```

y esperará hasta que se le de un número o cualquier otro carácter (por ejemplo, el damos 22). Es muy importante saber que el valor que se asigna a la variable, en este caso *entrada_numero*, es siempre un *string*, aunque se introduzca un número. Por ejemplo, pueden comprobar que en el ejemplo anterior el valor de la variable *entrada_numero* es “22”, con comillas, o sea un *string*. Si queremos operar aritméticamente con el resultado debemos pasarlo a *int* o a *float*, según cómo lo vayamos a usar, por ejemplo:

```
entrada_numero = float(entrada_numero)
```

Ahora, la variable *entrada_numero* es del tipo *float* y vale 22.0. Con la ayuda de esta función podemos crear programas que pidan uno o varios números (u otro tipo de dato) al usuario y hacer cálculos tan complejos como queramos con ellos, sin tener que cambiar los valores de entrada en la correspondiente línea del programa cada vez que se utilice. Por ejemplo, podríamos modificar el programa anterior de manera que pida al usuario el radio de la esfera cuando se ejecute el programa:

```
#!/usr/bin/python
#-*- coding: utf-8 -*-
# este programa calcula el volumen, la superficie y la longitud
# del círculo máximo de una esfera de radio r que solicita por pantalla
# teo
# marzo 2012

# importamos todas las funciones del módulo math
from math import *
```

```
# ahora definimos una función que calcula los parámetros de la esfera
def esfera(r):
    pir = pi*r
    longitud = 2.*pir
    superficie = 4.*pir*r
    volumen = superficie*r/3.
    # devolvemos una lista con los resultados
    return [longitud,superficie,volumen]

# ahora solicitamos por pantalla el radio de la esfera
radio = float(raw_input('Deme el radio de la esfera: '))

# Ahora utilizamos la función, guardando el resultado en tres variables
l,s,v = esfera(radio)

# imprimimos el resultado
print('La longitud del circulo maximo es: %.3f' % l)
print('La superficie de la esfera es: %.3f' % s)
print('El volumen de la esfera es: %.3f' % v)
```

De esta forma, si queremos reutilizar el programa para el cálculo en el caso de otra esfera de radio diferente no tenemos que cambiar nada del mismo y podemos volver a ejecutarlo tal cual está sin problemas, introduciendo el radio que queramos cada vez que la usemos.

4.4 Definiendo un módulo personal

Ya hemos visto como las funciones ayudan a hacer los programas más legibles y cortos al evitar que tengamos que escribir una y otra vez los mismos algoritmos en un mismo programa. Sin embargo a medida que vayamos aprendiendo a programar iremos haciendo más y más programas. Es muy posible que en varios de estos programas usemos las mismas funciones con lo que acabaremos teniendo que escribir la misma función en cada programa. Para evitar esta cuestión lo mejor es definir nuestro propio módulo que incluya las funciones que vamos definiendo.

Un módulo no es más que una colección de funciones que podemos utilizar desde cualquier programa. Ya hemos visto que la distribución estándar de Python ofrece un buen número de módulos predefinidos que se agrupan normalmente por ámbitos de aplicación: funciones matemáticas en `math`; las de números aleatorios en `random`; las que tratan con cadenas en `string`, etc... Así pues, ¿para qué vamos a re-escribir continuamente en nuestros programas funciones que nosotros ya hemos escrito antes?. Pues para evitarlo, creamos un módulo.

Para ello, creamos un fichero de texto que llamamos por ejemplo: `teolib.py` y vamos a incluir en él todas las funciones que vayamos creando. El sufijo o *extensión py* significa que lo que habrá dentro del fichero va a ser código Python. Incluyamos en él las funciones que hayamos definido en este capítulo:

```
def cubo(x):
    y = x**3
    return y

def saludo(nombre):
    print("Buenas tardes %s," % nombre)

def esfera(r):
    pir=pi*r
    longitud = 2.*pir
    superficie = 4.*pir*r
    volumen = superficie*r/3.
    return [longitud,superficie,volumen]
```

Listo. Ya podemos utilizar o importar el módulo en cualquier programa sin más que hacer:

```
>>> # importamos la función esfera del módulo teolib
>>> from teolib import esfera

>>> # o bien, importamos todas las funciones del módulo teolib
>>> from teolib import *
```

Cuando importas por vez primera *teolib.py* verás que Python crea automáticamente un fichero llamado *teolib.pyc*; este fichero contiene una versión del módulo más fácil de cargar en memoria que el original, pero ininteligible para nosotros ya que está codificado en “código binario”. De esta forma Python acelera la carga del módulo en sucesivas ocasiones sin que tengamos que preocuparnos nosotros de nada. Si se borra el fichero *teolib.pyc*, tranquilos no hay que preocuparse; simplemente cuando se vuelva a importar Python va a volver a crearlo; si se modifica el *teolib.py* añadiendo una función o modificando una existente, Python regenera de forma automática el fichero *teolib.pyc* para que siempre esté sincronizado con el *teolib.py*. Bueno, pues ya estamos listos para probar si esto funciona; háganlo y verán lo útil que resulta a la larga.

4.5 Estructura de un programa o *script*

Al hacer un programa es siempre conveniente guardar una cierta estructura que ayudará a entenderlo y corregirlo. No hay reglas definidas para ello pero una buena práctica podría ser la siguiente:

1. Primero poner los comentarios con el tipo de codificación, objeto del programa, nombre del programador y fecha de la última modificación.
2. Después incluir las sentencias que importan los módulos que nos van a hacer falta para el desarrollo del algoritmo solución del problema.
3. Después incluimos la definición de las funciones y procedimientos que vayamos a utilizar.
4. A continuación vendría el grueso del algoritmo o programa principal.

5. Finalmente, incluimos las órdenes de salida o de escritura de la solución del problema plantead.

4.6 Ejercicios

1. Escriba una función que calcule la distancia cartesiana entre dos puntos cualesquiera de coordenadas (x_1, y_1) y (x_2, y_2) .
2. Escriba un programa que calcule la densidad media de cualquier planeta, pidiendo como entrada su masa y su radio medio.
3. La variación de temperatura de un cuerpo a temperatura inicial T_0 en un ambiente a T_s cambia de la siguiente manera:

$$T = T_s + (T_0 - T_s) e^{-kt}$$

con t en horas y siendo k un parámetro que depende del cuerpo (usemos $k=0.45$). Una lata de refresco a 5°C queda en la guantera del coche a 40°C. ¿Qué temperatura tendrá 1, 5, 12 y 14 horas?. Encuentren las horas que tendríamos que esperar para que el cuerpo esté a 0.5°C menos que la temperatura ambiente. Definan funciones adecuadas para realizar ambos cálculos para cualquier tiempo y cualquier hora respectivamente.

4. Para el cálculo de la letra del DNI se calcula el residuo 23 (también llamado módulo) del número, es decir, el resto que se obtiene de la división entera del número del DNI entre 23. El resultado será siempre un valor entre 0 y 22 y cada uno de ellos tiene asignado una letra según la siguiente tabla:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

Escriban un programa que solicite el número de DNI al ususario y calcule la letra que le corresponde.

5. Escriba su propio módulo o librería que contenga todas las funciones que haya hecho para resolver los problemas anteriores. Escriba un programa principal en el que las utilice y de esta forma comprueba si funciona correctamente.

Control de flujo

Hemos visto que un programa o *script* consiste en una serie de sentencias que se ejecutan una detrás de otra siguiendo así el flujo natural de ejecución. No obstante, puede convenirnos cambiar esta secuencia de ejecución, de manera que nuestro programa haga una acción u otra según ciertas condiciones. Para ello se utilizan las **sentencias de control de flujo**, 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 presente en cualquier lenguaje de programación y en este capítulo veremos las más importantes.

5.1 El bucle *for*

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

```
for isla in ['La Gomera', 'Maui', 'Cuba', 'Sri Lanka']:
    print("La isla de %s" % isla)

"""Resultado:
La isla de La Gomera
La isla de Maui
La isla de Cuba
La isla de Sri Lanka
"""
```

En este caso *isla* es una **variable muda** (no definida previamente) que va tomando el valor de cada uno de los elementos de la lista que queramos de forma sucesiva; el bucle finaliza cuando se terminen los elementos de la lista. Es preciso darse cuenta de la importancia de la identación del bucle ya que Python reconoce que el bucle termina cuando la identación cambia.

Naturalmente, también es posible hacerlo definiendo antes la lista en una variable:

```
islas = ['La Gomera', 'Maui', 'Cuba', 'Sri Lanka']
for isla in islas:
    print("La isla de %s" % isla)
```

Aquí es especialmente útil la función `range()` que, como ya vimos, crea automáticamente una lista de números enteros; con ella, podemos usar un bucle `for` para hacer cálculos con los elementos de una lista de números; por ejemplo:

```
.. sourcecode:: python

for i in range(2,12,2): print("El cubo de %d es %d" % (i, i**3) )

"""
Resultado El cubo de 2 es 8 El cubo de 4 es 64 El cubo de 6 es 216 El cubo de 8 es
512 El cubo de 10 es 1000 """
```

El mismo resultado al que llegamos al principio se puede llegar utilizando el bucle `for` de otra forma. Consiste en **definir una variable índice para escoger o recorrer los elementos de la lista**; en este caso, la variable índice recorrerá otra lista (con idéntico número de elementos de la lista que se quiere recorrer) que contiene el valor de los índices; la podemos crear usando `range(len())`. Veamos un ejemplo:

```
islas = ['La Gomera', 'Maui', 'Cuba', 'Sri Lanka']
for j in range(len(islas)):
    print("%d- La isla de %s" % (j, islas[j]))

"""
Resultado
0- La isla de La Gomera
1- La isla de Maui
2- La isla de Cuba
3- La isla de Sri Lanka
"""
```

Aquí, con la función `range(len(islas))` hemos creado una lista nueva de tantos elementos (números enteros en este caso) como elementos tiene la lista `islas`, es decir, desde 0 hasta `len(islas)-1` (la longitud de la lista); después hacemos recorrer al índice `j` esta lista e identificamos los elementos de la otra lista con el índice apropiado `islas[j]`. Esta forma de utilizar el bucle `for` nos permite manejar elementos de otras listas que tengan la misma longitud.

Por otro lado, los bucles `for` nos permiten crear nuevas listas fácilmente, por ejemplo:

```
# Importo la función log10 del módulo math
b = [2.3, 4.6, 7.5, 10.]

from math import log10

c = [log10(x) for x in b]

print(c)
```

```
# Resultado:
# [0.36172783601759284, 0.66275783168157409, 0.87506126339170009, 1.0]
```

Al hacer esto hemos creado una nueva lista que contiene el logaritmo en base 10 de cada uno de números en la lista `b`. Con respecto a la función logaritmo, generalmente se denota el de base 10 como acabamos de ver, mientras que la función `log()` calcula el **logaritmo natural** o de base e . Con la función `log()` también se puede calcular el logaritmo de cualquier base, indicándola como un segundo parámetro; por ejemplo, para calcular el logaritmo en **base 3** de 5 haríamos `log(5, 3)`.

Otra aplicación muy interesante del bucle `for` es la suma de series de números. Supongamos que queremos calcular el sumatorio siguiente $\sum_{n=1}^{10} \frac{1}{n^2}$. Para ello debemos definir una variable en la que vamos acumulando los términos del sumatorio; hagamos un programa en un fichero para este cálculo.:

```
suma = 0.0      # Variable para ir acumulando los términos del sumatorio

print "n    termino n    sumatorio hasta n"
print "-----"

for n in range(1,11):
    term = 1/n**2.0          # calculamos el término del sumatorio
    suma = term + suma       # acumulamos el término en la variable suma
    print("%3d %10.6f %10.6f" % (n, term, suma)) # imprimimos el resultado

"""RESULTADO QUE OBTENEMOS EN PANTALLA AL EJECUTARLO
n    termino n    sumatorio hasta n
-----
1    1.000000    1.000000
2    0.250000    1.250000
3    0.111111    1.361111
4    0.062500    1.423611
5    0.040000    1.463611
6    0.027778    1.491389
7    0.020408    1.511797
8    0.015625    1.527422
9    0.012346    1.539768
10   0.010000    1.549768
"""
```

En este ejemplo definimos una variable `suma` con valor inicial cero para ir acumulando en ella cada uno de los términos de la suma, que metemos en la variable `term` y sumándola en cada ciclo del bucle hasta que termina. El valor final de la suma será el que tenga la variable `suma` al terminar el bucle.

Si no conocemos el número de veces que hay que ejecutar las operaciones (es decir, el número de términos de la serie a sumar) existe otra forma de definir los bucles utilizando la sentencia `while`.

5.2 El bucle *while*

En este bucle se ejecutan una o varias operaciones mientras cierta condición que definimos sea cierta. Por ejemplo para imprimir los números naturales del 0 al 5 podemos hacer:

```
cuentas = 0

while cuentas < 6:
    print(cuentas)
    cuentas = cuentas + 1

0
1
2
3
4
5
```

En este ejemplo se define inicialmente un valor 0 para la variable `cuentas` y su valor se va **redefiniendo**, aumentado en 1 e imprimiéndolo. Mientras `cuentas` sea menor que 6 las sentencias dentro del bucle *while* seguirán ejecutándose y se detendrá cuando la condición deje de cumplirse, es decir cuando `cuentas` valga 6, algo que podemos comprobar después del bucle sin más que escribir:

```
print(cuentas)
# Imprime 6
```

Nótese que, a diferencia del bucle *for*, en este caso debemos incrementar explícitamente el valor de la variable que interviene en la condición, de no haber aumentado el valor de `cuentas` en cada ciclo del bucle, su valor nunca cambiaría y tendríamos bucle infinito. Si esto nos ocurre podemos parar el programa con las teclas *Ctrl+C*.

Veamos otro ejemplo. Supongamos que tenemos unos ahorros en el banco (unos míseros 100 euros) y queremos saber el tiempo que nos llevará tener cierta cantidad (p.e. 500 euros) gracias a los intereses que producen. Lo que hacemos es crear un bucle *while* en el que añadiremos anualmente los intereses y vamos contando los años. Cuando lleguemos a la cantidad deseada el bucle se detendrá y tendremos los años que nos llevaría; el código del programa podría ser este:

```
mis_ahorros = 100                                # Partimos de 100 euros
interes = 1.05                                     # Interés del 5% anual
anhos = 0                                         # Tiempo de inicio, cuando tenemos 100 euros

while mis_ahorros < 500:                            # Queremos llegar a tener 500 euros
    mis_ahorros = mis_ahorros * interes            # Añado los intereses anuales a los ahorros
    anhos = anhos + 1                             # Añado un año a la cuenta de años ya que
                                                # cada ciclo while equivale a un año.
```

```
print("Me llevará %d años ahorrar %d euros." % (anhos, mis_ahorros))

# Resultado que obtendremos al final del programa:
# Me llevará 33 años ahorrar 500 euros.
```

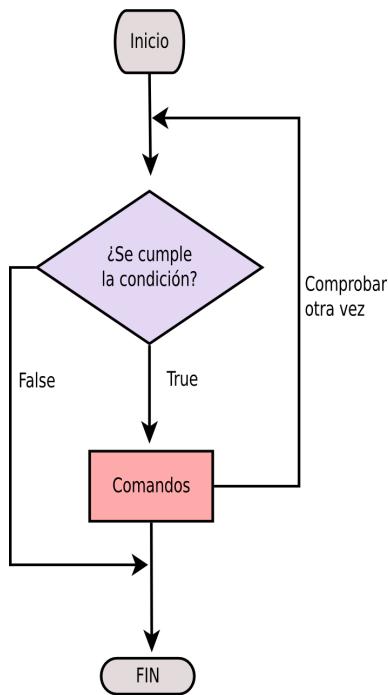


Figura 5.1: Diagrama de flujo de una sentencia *while*.

Podemos utilizar el bucle `while`` con una condición negativa, es decir de que `**no se cumpla**`, usando `''while not` de la forma siguiente:

```
x = 0
while not x == 5:
    x = x + 1
    print("x = %d" % x)

""" Resultado que obtenemos del programa:
x = 1
x = 2
x = 3
x = 4
x = 5
"""


```

Sin embargo, hay que tener cuidado cuando se comparan, mediante una igualdad exacta, números decimales o de coma flotante *floats* entre sí. 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. Comprobemos este aspecto con un ejemplo en el que imprimimos los números que van de 0.0 a 1.0 a intervalos de 0.1,:

```
x = 0.0
# Mientras x no sea exactamente 1.0, suma 0.1 a la variable *x*
while not x == 1.0:
    x = x + 0.1
    print("x = %19.17f" % x)

""" Resultado que obtenemos:
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
x = 0.5000000000000000
x = 0.5999999999999998
x = 0.6999999999999996
x = 0.7999999999999993
x = 0.8999999999999991
x = 0.9999999999999989      <-- El bucle while debió detenerse aquí, pero no lo hizo
x = 1.0999999999999987
x = 1.1999999999999996
x = 1.3000000000000004
.
.
.
<-- Presiono Ctrl+C para detener el programa
"""

y así sucesivamente; el bucle no se para nunca. El código anterior produce un bucle que no se para nunca porque la condición x == 1.0 nunca se da exactamente debido a la precisión limitada de los ordenadores (el valor más cercano es 0.9999999999999989 pero no 1.0). La conclusión que podemos extraer de aquí es que es preferible no comparar nunca variables o números de tipo *float* exactamente. Una opción para resolver el problema anterior es usar intervalos de precisión, por ejemplo:
```

```
x = 0.0
while abs(x - 1.0) > 1e-8:
    x = x + 0.1
    print("x = %19.17f" % x)
```

```
""" Resultado que obtenemos:
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
x = 0.5000000000000000
x = 0.5999999999999998
x = 0.6999999999999996
x = 0.7999999999999993
x = 0.8999999999999991
```

```
x = 0.9999999999999989
"""


```

de esta forma cuando `x` se acerque a 0.1 con una precisión igual o inferior a 1×10^{-8} se detendrá el bucle.

Advertencia: Recuerden que en Python la **identación es obligatoria**, porque se emplea para **separar los bloques** que indican donde empiezan y terminan los bucles y otros condicionales. Si la identación no es correcta, se obtendrá un resultado equivocado o nos devolverá un error de identación.

Por otro lado, como **regla general** para usar el bucle *for* o el *while* podemos decir que cuando sepamos el número de veces que deseamos calcular el bucle es mejor utilizar el *for* mientras que en los demás casos es más recomendable usar el *while*.

5.3 Sentencias condicionadas *if-else*

Este conjunto de sentencias realizan una o varias operaciones si una determinada condición que imponemos es cierta. De no cumplirse tal condición, se pueden realizar opcionalmente otras operaciones o detener el proceso de cálculo. Un ejemplo de su utilización es el siguiente:

```
c = 12

if c>0:                      # comprueba si es positivo
    print("La variable c es positiva")
elif c<0:                      # si no lo es, comprueba si es negativo
    print("La variable c es negativa")
else:                          # Si nada de lo anterior se cumple, haz lo siguiente
    print("La variable c vale 0")
```

En el ejemplo anterior, primero se define una variable `c` con valor entero 12 y luego se emplea la sentencia `if` para comprobar si `c` es positiva, luego se usa `elif` para que en caso de no cumplirse el `if` anterior, compruebe si `c` es negativa. Si no se cumple ninguna de las condiciones anteriores y sólo en ese caso, se ejecutan las sentencias que vienen después de `else`. Hay que notar que en caso de cumplirse la primera condición `if`, el bucle se interrumpe y el intérprete ya no continúa comprobando las posibles condiciones `elif` (pueden haber varias) o `else` final.

Advertencia: Es muy importante darse cuenta nuevamente los **bloques de identación** o espacios que separan los condicionales `if-else` en el ejemplo anterior. Al ejecutarse el código y en contrarse el primer `if`, el intérprete de Python sabe que todo lo que viene después de los ":" e identado con espacios es lo que debe ejecutarse en caso de cumplirse la condición y esto termina cuando se encuentra con un bloque de identación inferior, que en este caso es la sentencia `elif`.

Veamos lo anterior con un ejemplo usando sólo la sentencia `if`. Tenemos dos números diferentes y

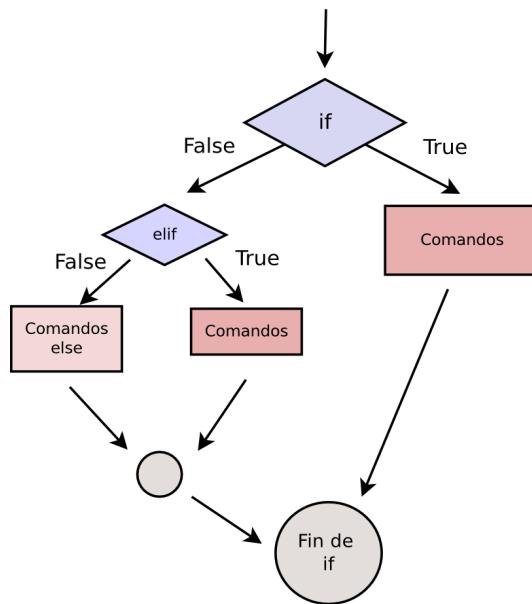


Figura 5.2: Diagrama de flujo de una sentencia if-else.

sólo queremos calcular y escribir su diferencia si el primero es mayor que el segundo; el programa podría ser:

```
a, b = 9.3, 12.0
```

```
if a > b:
    c = a - b
    print ("La variable a es mayor que b")
print ("El valor de c es %f" % c)
```

En este ejemplo se comprueba si a es mayor que b y en ese caso se calcula su diferencia e imprime un mensaje. Luego se imprime el valor de c en otra sentencia, pero como esa línea está ya fuera del bloque de identación del `if`, el condicional termina justo antes y esa sentencia se intentará imprimir aunque la condición del `if` no se cumpla, ya que no está contenido en ella. El código correcto sería simplemente

```
a, b = 9.3, 12.0
```

```
if a > b:
    c = a - b
    print ("La variable a es mayor que b")
    print ("El valor de c es %f" % c)
```

Así, el valor de c sólo imprime si el condicional se cumple.

Hay varias variantes para escribir estas sentencias que se irán viendo y aprendiendo con el tiempo. Por ejemplo, cuando después de un condicional hay **una única sentencia**, ésta se puede escribir en la misma línea sin necesidad de identar:

```
if a > b: print ("La variable a es mayor que b")
else: print ("La variable a es menor o igual que b")
```

Evidentemente, los elementos de control de flujo pueden contener a su vez otros elementos de flujo; por ejemplo, después de un `if` puede venir otra vez otro `if` con otras condiciones. También es posible reproducir el comportamiento de un `while` combinando `if` y `for`, etc., pero como siempre, hay que ser muy cuidadoso con los bloques de identación.

5.4 Declaraciones `break` y `continue` y sentencia `else` en bucles

La sentencia `break` se puede usar para interrumpir el bloque más cercano en un bucle `while` o `for`. De manera similar, `continue` continua con la siguiente iteración dentro del bucle más próximo. La sentencia `else` en bucles permite ejecutar una acción cuando el bucle termina una lista (en bucles `for`) o cuando la condición es falsa (con `while`) pero no si el bucle se interrumpe usando `break`. Veamos un ejemplo:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print("%d es igual a %d * %d." % (n, x, n/x))
            break # corto el bucle for
        else:
            # El bucle termina sin encontrar factor
            print("%d es numero primo." % n)

2 es numero primo.
3 es numero primo.
4 es igual a 2*2.
5 es numero primo.
6 es igual a 2*3.
7 es numero primo.
8 es igual a 2*4.
9 es igual a 3*3.
```

En este ejemplo usamos `break` para cortar el bucle `for` más interno si el `if` se cumple (es `True`) y así evitar que muestre multiplicaciones equivalentes (e.g.: $3 \times 4 = 4 \times 3$); podemos comprobar lo que ocurre si no pusiésemos `break`. Es útil para por ejemplo, evitar que un bucle siga corriendo si ya se consiguió la condición. Con la sentencia `else` imprimimos un aviso si el bucle termina (el `for` agota la lista) sin llegar a usar `break`, lo que indica que ningún número de la lista es múltiplo suyo y por tanto es primo.

La declaración `continue` indica continuar con el siguiente elemento del bucle más interior, interrumpiendo el ciclo actual. Vámoslo con un ejemplo:

```
for k in range(8):
    if k > 4:
        print("%d es mayor que 4." % k)
        continue
    print("%d es menor o igual que 4." % k)

0 es menor o igual que 4.
1 es menor o igual que 4.
2 es menor o igual que 4.
3 es menor o igual que 4.
4 es menor o igual que 4.
5 es mayor que 4.
6 es mayor que 4.
7 es mayor que 4.
```

En este caso, con `continue` evitamos que se ejecute el último `print()` si `k>4`, continuando el bucle con el siguiente elemento de la lista. Fíjate que pudimos haber hecho un código similar usando una sentencia `if-else`.

5.5 Atrapando los errores. Sentencia `try-except`

Hemos visto que cuando existe algún error en el código, Python detiene la ejecución y nos devuelve una **excepción o mensaje de error** indicándonos qué fue lo que ocurrió. Por ejemplo, supongamos que por algún motivo hacemos una división por cero:

In [4]: `a, b = 23, 0`

In [5]: `a/b`

```
-----
ZeroDivisionError                                                 Traceback (most recent call last)

/home/japp/<ipython console> in <module>()
ZeroDivisionError: integer division or modulo by zero
```

Al hacer esto, nos avisa del error indicando el tipo en la última línea, `ZeroDivisionError`, terminando la ejecución. Que Python nos dé tanta información al ocurrir una excepción es muy útil pero muy a menudo sabemos que estos errores pueden ocurrir y lo ideal es estar preparado capturando la excepción y actuar en consecuencia en lugar de interrumpir el programa. Para hacer esto podemos usar la sentencia `try-except`, que nos permite “probar” (Try) una sentencia y capturar un eventual error y hacer algo al respecto (except) en lugar de detener el programa. En el caso anterior podríamos hacer lo siguiente:

```
a, b = 23, 0
```

```
try:
```

```
    a/b
```

```
except:  
    print("Hay un error en los valores de entrada")
```

Ahora el código intenta ejecutar a/b y de haber algún tipo de error imprime el mensaje indicado y sigue adelante en lugar abortar la ejecución del programa. Al hacer esto hemos “capturado” la excepción o error evitando que el programa se detenga, suponiendo que éste puede continuar a pesar del error. Nótese que de esta manera no sabemos qué tipo de error ha ocurrido, que antes se indicaba con la clave `ZeroDivisionError`, que es uno de los muchos tipos de errores que Python reconoce. Si quisiésemos saber exactamente qué tipo de error ocurrió, debemos especificarlo en `except`, por ejemplo:

```
a, b, c = 23, 0, "A"  
  
try:  
    a/b  
except ZeroDivisionError:  
    print("Error, division por cero.")  
except TypeError:  
    print("Error en el tipo de dato.")  
  
# Resultado:  
# Error, division por cero.  
  
try:  
    a/c  
except ZeroDivisionError:  
    print("Error, division por cero.")  
except TypeError:  
    print("Error en el tipo de dato.")  
  
# Resultado:  
# Error en el tipo de dato.
```

De esta manera, sabemos exactamente qué tipo de error se cometió en cada caso, una división por cero o un error en el tipo de dato (que es lo que indica `TypeError`). Naturalmente, si no ocurriese ninguno de estos errores específicamente, Python daría un error y terminaría el programa de manera habitual. Pueden consultar la documentación oficial de Python para tener más información sobre la captura de excepciones y los tipos de errores reconocidos.

5.6 Una aplicación interesante: Raíces de ecuaciones algebraicas cualquiera

A menudo en ciencia nos encontramos con que los modelos con los que intentamos describir la realidad de un fenómeno natural nos lleva a tener que resolver ecuaciones polinómicas o tras-

cententes (que no tienen solución algebraica) u otras en las que, por su complejidad, la solución es complicada. La búsqueda de las raíces de tales ecuaciones en un intervalo prefijado podemos hacerlo con el cálculo numérico de forma sencilla utilizando lo que sabemos hasta ahora.

El método más sencillo es el llamado *método del bisector* que nos permite encontrar una raíz de una función real cualquiera $f(x)$, continua en el intervalo $[a, b]$ donde $f(a)$ y $f(b)$ tienen diferente signo. Recuerden que el Teorema de Bolzano nos dice que esto es posible y lo que hay que hacer es implementar un programa en Python para resolverlo en el caso de funciones de una sola variable real.

El método consiste en dividir el intervalo dado $[a, b]$ por la mitad y llamemos m al punto medio. Si el signo de $f(m)$ es diferente al de $f(a)$ aplicamos otra vez el método al intervalo $[a, m]$; si, por el contrario, el signo de $f(m)$ es diferente al de $f(b)$ aplicamos otra vez el método al intervalo $[m, b]$. El nuevo intervalo en cualquier caso es menor que el anterior y contiene la raíz buscada, por lo tanto lo único que tenemos que hacer es repetir el método anterior tantas veces como sea necesario hasta que $f(m) = 0$ o, como ya sabemos, es mejor utilizar la condición $|f(m)| < \epsilon$, donde ϵ es un número tan pequeño como queramos. Este es un método iterativo, es decir, que se repite tantas veces como sea necesario hasta que se cumpla la condición que impongamos a la solución.

5.7 Ejercicios

1. Escriban un programa que calcule el factorial de un número n entero y positivo. Es decir, calculen:

$$n! = \prod_{k=1}^n k \quad \text{para } n \geq 0$$

2. Crean una lista con 10 números enteros con valores distintos y arbitrarios de 0 a 100. Programen una función que encuentre el mayor de ellos y que dé su posición en la lista.
3. Utilizando la lista anterior, crear una lista nueva que incluya los números que son primos y otra que incluya sus índices en la lista original.
4. Generen una lista con 100 números enteros aleatorios de -100 a 100, utilizando la función `randint()` (escriban `from numpy.random import randint` y luego `nums = randint(-100, 100, 100)`). Separen en tres listas distintas los números negativos, los positivos y los mayores de 50, pero de manera que la suma de los números de cada lista no sea mayor que 200; es decir, vayan llenando las listas mientras se cumpla la condición.
5. Obtener un valor de π calculando la suma siguiente para $n=200$:

$$4 \sum_{k=1}^n \frac{(-1)^{k+1}}{2k - 1}$$

6. Modifiquen el programa anterior para obtener el valor de π con una precisión determinada (por ejemplo 10^{-6}) comparado con el valor más aproximado tomado del módulo `math`.

7. Diseñe un programa que calcule volumen de una esfera, cilindro o un cono. El programa debe preguntar primero qué es lo que se desea calcular y luego pedir los datos necesarios según lo que se elija.
8. Dada la lista de notas del alumnado de una clase, decir quien ha obtenido aprobado (entre 5 y 6.9), notable (entre 7 y 8.9), sobresaliente (más de 9) o ha suspendido.

Alumnado	Nota
Miguel	6.7
Maria	4.9
Iballa	9.8
Fran	5.0
Luisa	6.7
Ruyman	8.0
Ana	6.2

Suponiendo que todos los nombres de mujer terminan con “a” (lo que casualmente en este ejemplo es cierto), decir del alumnado quien es varón o mujer.

9. Escribir un programa que calcule la suma de los elementos necesarios de la serie siguiente:

$$\sum_{i=0}^{i=n,2} \frac{1}{(i+1)(i+3)},$$

para obtenerla con 5 cifras significativas. Como resultado dar el valor de la suma y el número n de sumandos sumados.

10. La nota final de la asignatura de Computación Científica (p) se calcula añadiendo a la nota del examen final (z) una ponderación de la evaluación continua (c) a lo largo del curso de la forma:

$$p = 0.6c + z \frac{(10 - 0.6c)}{10}$$

El alumno estará aprobado cuando la nota final p sea mayor o igual a cinco, siempre que z supere un tercio de la nota máxima ($z > 10/3$); en caso contrario, se queda con $p=z$. Un grupo de alumnos ha obtenido las siguientes calificaciones en la evaluación continua y en el examen final:

c	8.2 0.0 9.0 5.0 8.4 7.2 5.0 9.2 4.9 7.9
z	7.1 5.1 8.8 3.1 4.6 2.0 4.1 7.4 4.4 8.8

Hagan un programa que calcule sus notas finales indicando además quién ha aprobado y quién ha suspendido. Calcular también la nota media de la evaluación continua, del examen final y de la nota final. En todos los resultados debe mostrarse una única cifra decimal.

11. Dada la serie geométrica cuya suma exacta es:

$$\sum_{n=0}^{\infty} ax^n = \frac{a}{1-x}$$

válida siempre que $0 < x < 1$ y siendo a un número real cualquiera, escribir un programa que calcule esta suma con diez cifras significativas solamente para cualquier valor de x y a , comprobando que se cumple la condición necesaria para x . Dar como resultado el valor de la suma y el número de sumandos sumados para obtenerla.

12. Se llama sucesión de Fibonacci a la colección de n números para la que el primer elemento es cero, el segundo 1 y el resto es la suma de los dos anteriores. Por ejemplo, la sucesión para $n=5$ es (0, 1, 1, 2, 3). Crear un programa que calcule la lista de números para cualquier n .
13. Escriban un programa que proporcione el desglose en el número mínimo de billetes y monedas de una cantidad entera cualquiera de euros dada. Recuerden que los billetes y monedas de uso legal disponibles hasta 1 euro son de: 500, 200, 100, 50, 20, 10, 5, 2 y 1 euros. Para ello deben solicitar al usuario un número entero, debiendo comprobar que así se lo ofrece y desglosar tal cantidad en el número mínimo de billetes y monedas que el programa escribirá finalmente en pantalla.
14. Escriba un programa que pida el valor de dos números enteros n y m cualesquiera y calcule el sumatorio de todos los números pares comprendidos entre ellos (incluyéndolos en el caso de que sean pares). Compruebe que efectivamente los números proporcionados son enteros.
15. Encuentren el cero de la función $f(x) = x - \tan(x)$ en el intervalo [0,15]
16. Una de las formas de medir distancias en el cosmos es utilizar el llamado módulo de distancias, en el que la diferencia entre las magnitudes relativa m_v (proporcional al logaritmo del flujo recibido) y absoluta M_v (proporcional al logaritmo del flujo recibido si el astro se encontrara a 10 parsec de distancia) de un astro es $m_v - M_v = 5 \log\left(\frac{r}{10}\right) + ar$, donde, r es la distancia en parsec y a es el coeficiente de absorción interestelar. Lo aplicamos a un caso concreto. En la Gran Nube de Magallanes, en 1987 se descubrió una supernova la SN1987A que en su máximo de luminosidad alcanzó una magnitud de $m_v = 3$. Sabiendo que este tipo de supernovas tienen una magnitud absoluta de $M_v = -19.3$ y que el coeficiente de absorción interestelar es de $a = 0.8 \times 10^{-4}$, ¿a qué distancia se encuentra la galaxia?.

Cálculo numérico con Numpy

Hasta ahora hemos venido para hacer algunos cálculos científicos hemos venido utilizando el paquete `math` y utilizando listas como sucesiones de números y bucles para hacer operaciones matemáticas en ellas. No obstante, las listas tienen muy poca utilidad cuando se trata de hacer cálculos científicos con muchos números a la vez. Esta situación es muy común en ciencia e ingeniería. Para solucionar estos problemas de cálculo numérico en Python está el paquete llamado `numpy` que vamos a introducir en este capítulo.

7.1 Listas y arrays

Los **arrays** son un tipo nuevo de dato, similar a las listas, pero orientados especialmente al cálculo numérico. En cierto modo se pueden considerar como vectores o matrices (parecidos a como se los conoce en álgebra) y son un tipo de dato fundamental para el cálculo científico de cierto nivel utilizando tipos de datos estructurados (conjuntos de datos).

El inconveniente principal de las listas, que es el tipo básico de dato estructurado en Python, es que no está pensado para el cálculo matemático o numérico; veámolo con un ejemplo:

```
In [1]: lista = range(5)           # Lista de numeros de 0 a 4

In [2]: print(lista*2)
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]

In [3]: print(lista*2.5)
-----
TypeError                                         Traceback (most recent call last)
/home/japp/<ipython console> in <module>()
      1
      2 TypeError: can't multiply sequence by non-int of type 'float'
```

En el ejemplo anterior vemos cómo al multiplicar una lista por un número entero, el resultado es concatenar la lista original tantas veces como indica el número, en lugar de multiplicar cada uno de sus elementos por este número. Peor aún, al multiplicarlo por un número no entero da un error, al no poder crear una fracción de una lista. Si quisieramos hacer esto, se podría resolver iterando cada uno de los elementos de la lista con un bucle `for`, por ejemplo:

```
In [4]: lista_nueva = [i*2.5 for i in lista]
In [5]: print(lista_nueva)
[0.0, 2.5, 5.0, 7.5, 10.0]
```

aunque esta técnica es ineficiente y lenta, sobre todo cuando queremos evaluar funciones, polinomios o cualquier otra operación matemática que aparece en cualquier problema científico.

Cuando realmente queremos hacer cálculos con listas de números, debemos usar los *arrays*. El módulo `numpy` nos da acceso a los *arrays* y a una enorme cantidad de métodos y funciones aplicables a los mismos. Obviamente, `numpy` incluye funciones matemáticas básicas similares al módulo `math`, las completa con otras más elaboradas y además incluye algunas utilidades de números aleatorios, ajuste lineal de funciones y otras muchas que iremos viendo más adelante y que pueden comprobar usando la ayuda como ya sabemos.

7.2 Creando arrays

Primero debemos importar el módulo `numpy` en sí o bien todas sus funciones:

```
In [6]: import numpy                                     # Cargar el modulo numpy, o bien
In [7]: import numpy as np                                # cargar el modulo numpy, llamándolo np, o bien
In [8]: from numpy import *                               # cargar todas funciones de numpy
```

Si cargamos el módulo solamente, accederemos a las funciones como `numpy.array()` o `np.array()`, según cómo importemos el módulo; si en lugar de eso importamos todas las funciones, accederemos a ellas directamente (e.g. `array()`). Por comodidad usaremos por ahora esta última opción, aunque muy a menudo veremos que usa la notación `np.array()`, especialmente cuando trabajamos con varios módulos distintos.

Un *array* se puede crear explícitamente o a partir de una lista de la forma siguiente:

```
In [9]: x = array([2.0, 4.6, 9.3, 1.2])                # Creacion de un array directamente
In [10]: notas = [ 9.8, 7.8, 9.9, 8.4, 6.7]           # Crear un lista
In [11]: notas = array(notas)                          # y convertir la lista a array
```

Existen métodos para crear arrays automáticamente:

```
In [12]: numeros = arange(10.)                         # Array de numeros(floats) de 0 a 9
In [13]: print(numeros)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
In [14]: lista_ceros = zeros(10)                         # Array de 10 ceros (floats)
```

```
In [15]: print(lista_ceros)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.]  
  
In [16]: lista_unos = ones(10)                      # Array de 10 unos (floats)  
In [17]: print(lista_unos)
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.]  
  
In [18]: otra_lista = linspace(0,30,8)              # Array de 8 números, de 0 a 30 am
In [19]: print(otra_lista)
[ 0.          4.28571429   8.57142857  12.85714286  17.14285714
 21.42857143  25.71428571  30.         ]
```

7.3 Indexado de arrays

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

```
In [18]: print(numeros[3:8])                      # Elementos desde el tercero al séptimo
[ 3.  4.  5.  6.  7.]  
  
In [19]: print(numeros[:4])                        # Elementos desde el primero al cuarto
[ 0.  1.  2.  3.]  
  
In [20]: print(numeros[5:])                         # Elementos desde el quinto al final
[ 5.  6.  7.  8.  9.]  
  
In [21]: print(numeros[-3])                        # El antepenúltimo elemento (devuelve un ele
7.  
  
In [24]: print(numeros[:])                          # Todo el array, equivalente a print(numeros)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]  
  
In [25]: print(numeros[2:8:2])                     # Elementos del segundo al séptimo, pero salteando uno cada dos
[ 2.  4.  6.]
```

7.4 Algunas propiedades de los arrays

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()`:

```
In [28]: print(len(numeros))
10
```

```
In [29]: print(numeros.shape)
(10,)
```

Nótese que el resultado del método `shape()` es una tupla, en este caso con un solo elemento ya que el *array* `numeros` es unidimensional.

Si creamos un *array* con `arange()` usando un número entero, el array que se creará será de enteros. Es posible cambiar todo el array a otro tipo de dato (como a *float*) usando el método `astype()`:

```
In [31]: enteros = arange(6)
```

```
In [32]: print(enteros)
[0 1 2 3 4 5]
```

```
In [33]: type(enteros)
Out[33]: <type 'numpy.ndarray'>
```

```
In [34]: type(enteros[0])
Out[34]: <type 'numpy.int32'>
```

```
In [35]: decimales = enteros.astype('float')
```

```
In [36]: type(decimales)
Out[36]: <type 'numpy.ndarray'>
```

```
In [37]: type(decimales[0])
Out[37]: <type 'numpy.float64'>
```

```
In [38]: print(decimales)
[ 0.  1.  2.  3.  4.  5.]
```

```
In [38]: print(decimales.shape)      # Forma o tamaño del array
(6, )
```

7.5 Operaciones con arrays

Los arrays permiten hacer operaciones aritméticas básicas entre ellos en la forma que uno esperaría que se hicieran, es decir, haciendo elemento a elemento; para ello siempre ambos arrays deben tener la misma longitud; por ejemplo:

```
In [39]: x = array([5.6, 7.3, 7.7, 2.3, 4.2, 9.2])
```

```
In [40]: print(x+decimales)
[ 5.6  8.3  9.7  5.3  8.2 14.2]
```

```
In [41]: print(x*decimales)
[ 0.    7.3  15.4   6.9  16.8  46. ]

In [42]: print(x/decimales)
[         Inf   7.3           3.85           0.76666667  1.05          1.84      ]
```

Como podemos apreciar las operaciones se hacen elemento a elemento, por lo que ambas deben tener la misma forma (`shape()`). Fíjense que en la división el resultado del primer elemento es indefinido/infinito (Inf) debido a la división por cero.

Varios arrays se pueden unir con el método `concatenate()`, que también se puede usar para añadir elementos nuevos:

```
In [44]: z = concatenate((x, decimales))

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. ]
```

Es muy importante fijarse que los *arrays* o listas a unir **deben darse como una tupla** y de ahí los elementos entre paréntesis como `(x, [7])` o `(x, [2, 4, 7])` o `(x, array([2, 4, 7]))`.

Además de las operaciones aritméticas básicas, los *arrays* de numpy tienen **métodos** o funciones específicas para ellas más avanzadas. Algunas de ellas son las siguientes:

```
In [5]: z.max()    # Valor máximo de los elemnts del array
Out[5]: 9.199999999999999

In [6]: z.min()    # Valor mínimo de los elemnts del array
Out[6]: 2.299999999999998

In [7]: z.mean()   # Valor medio de los elemnts del array
Out[7]: 6.1857142857142851

In [8]: z.std()    # Desviación típica de los elemnts del array
Out[8]: 2.1603098795103919

In [9]: z.sum()    # Suma de todos los elementos del array
Out[9]: 43.299999999999997

In [16]: median(z) # Mediana de los elemnts del array
Out[16]: 7.0
```

Los métodos, que se operan de la forma `z.sum()` también pueden usarse como funciones de tipo `sum(z)`, etc. Consulten el manual de numpy para conocer otras propiedades y métodos de los

arrays o simplemente acudan y consulten la “ayuda” de las funciones que quieran utilizar.

Una gran utilidad de los *arrays* es la posibilidad de usarlos con datos booleanos (**True** o **False**) y operar entre ellos o incluso mezclados con *arrays* con números. Veamos algunos ejemplos:

```
In [19]: A = array([True, False, True])
In [20]: B = 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])
```

En este ejemplo vemos cómo al multiplicar dos *arrays* booleanos el resultado es otro *array* booleano con el resultado que corresponda, pero al multiplicar los *arrays* booleanos con *arrays* numéricos, el resultado es un *array* numérico con los mismos elementos, pero con los elementos que fueron multiplicados por **False** iguales a cero.

También es posible usar los *arrays* como índices de otro *array* y como índices se pueden usar *arrays* numéricos o booleanos. El resultado será en este caso un *array* con los elementos que se indique en el *array* de índices numérico o los elementos correspondientes a **True** en caso de usar un *array* de índices booleano. Veámoslo con un ejemplo:

```
# Array con enteros de 0 a 9
In [37]: mi_array = arange(0,100,10)

# Array de indices numericos con numeros de 0-9 de 2 en 2
In [38]: indices1 = arange(0,10,2)

# Array de indices booleanos
In [39]: indices2 = array([False, True, True, False, False, True, False, False])

In [40]: print(mi_array)
[ 0 10 20 30 40 50 60 70 80 90]

In [43]: print(mi_array[indices1])
[ 0 20 40 60 80]

In [44]: print(mi_array[indices2])
[10 20 50 80 90]
```

También es muy sencillo crear *arrays* booleanos usando operadores lógicos y luego usalos como índices, por ejemplo:

```
# Creo un array usando un operador booleano
In [50]: mayores50 = mi_array > 50

In [51]: print(mayores50)
[False False False False False False  True  True  True  True]

# Lo utilizo como índices para seleccionar los que cumplen esa condición
In [52]: print(mi_array[mayores50])
[60 70 80 90]
```

7.6 Arrays multidimensionales

Hasta ahora sólo hemos trabajado con *arrays* con una sola dimensión, pero numpy permite trabajar con *arrays* de más dimensiones. Un array de dos dimensiones podría ser por ejemplo un array que tuviera como elementos un sistema de ecuaciones o una imagen. Para crearlos podemos hacerlo declarándolos directamente o mediante funciones como `zero()` o `ones()` dando como parámetro **una tupla** con la forma del *array* final que queramos; o también usando `arange()` y crear un *array* unidimensional y luego cambiar su forma. Veamos algunos ejemplos:

```
# Array de 3 filas y tres columnas, creado implícitamente
In [56]: arr0 = array([[10,20,30],[9, 99, 999],[0, 2, 3]])
In [57]: print(arr0)
[[ 10  20  30]
 [  9  99 999]
 [  0   2   3]]

# Array de ceros con 2 filas y 3 columnas
In [57]: arr1 = zeros((2,3))
In [59]: print(arr1)
[[ 0.  0.  0.]
 [ 0.  0.  0.]]

# Array de unos con 4 filas y una columna
In [62]: arr2 = ones((4,1))
In [63]: print(arr2)
[[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]]]

# Array unidimensional de 9 elementos y cambio su forma a 3x3
In [64]: arr3 = 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)
```

Como vemos en la última línea, la forma o `shape()` de los *arrays* se sigue dando como una tupla, con la dimensión de cada eje separado por comas; en ese caso la primera dimensión son las cuatro filas y la segunda dimensión o eje es una columna. Es por eso que al usar las funciones `zero()`, `ones()`, `reshape()`, etc. hay que asegurarse que el parámetro de entrada es una tupla con la longitud de cada eje. Cuando usamos la función `len()` en un *array* bidimensional, el resultado es la longitud del primer eje o dimensión, es decir, `len(arr2)` es 4.

El acceso a los elementos es el habitual, pero ahora hay que tener en cuenta el eje al que nos referimos; además podemos utilizar ":" como comodín para referirnos a todo el eje. Por ejemplo:

```
# Primer elemento de la primera fila y primera columna (0, 0)  
In [86]: arr0[0,0]  
Out[86]: 10  
# Primera columna  
In [87]: arr0[:,0]  
Out[87]: array([10,  9,  0])  
# Primera fila  
In [88]: arr0[0,:]  
Out[88]: array([10, 20, 30])  
# Elementos 0 y 1 de la primera fila  
In [89]: arr0[0,:2]  
Out[89]: array([10, 20])
```

Igualmente podemos manipular un *array* bidimensional usando sus índices:

```
# Asigno el primer elemento a 88  
In [91]: arr0[0,0] = 88  
# Asigno elementos 0 y 1 de la segunda fila  
In [92]: arr0[1,:2] = [50,60]  
# Multiplico por 10 la última fila  
In [93]: arr0[-1,:] = arr0[-1,:]*10  
  
In [94]: print(arr0)  
array([[ 88,  20,  30],  
       [ 50,  60, 999],  
       [  0,   20,   30]])
```

7.7 Ejercicios

1. Crear un programa que resuelva la ecuación de segundo grado $ax^2+bx+c = 0$ para cualquier valor de a , b y c comprobando el valor del discriminante $\Delta = b^2 - 4ac$.

2. La función **arctan2** del módulo math o de numpy permite calcular el arcotangente de (y,x) medido en radianes, lo que permite mantener la información sobre los cuadrantes. El valor medio de n ángulos se puede realizar con la siguiente ecuación:

$$M\theta = \text{arctan2} \left(\frac{1}{n} \cdot \sum_{j=1}^n \sin \theta_j, \frac{1}{n} \cdot \sum_{j=1}^n \cos \theta_j \right)$$

Calcule el valor medio de los ángulos 12.3, 10.1, 11.9, 12.4, 10.4 y 10.9.

Lectura y escritura de ficheros

Muy a menudo tenemos datos iniciales para un cálculo o medidas de un experimento en un fichero (de texto). Para poder manipular estos datos y calcular con ellos debemos aprender a leerlos como números o arrays. Igualmente, el resultado de un cálculo o un análisis es necesario volcarlo a un fichero de texto en lugar de mostrarlo por pantalla para conservar el resultado. Esto es especialmente necesario cuando los resultados son *arrays* largos o cuando tenemos que procesar un gran número de ficheros. Vamos a ver cómo leer y escribir ficheros de texto, es decir letras y números y signos de puntuación, con Python.

9.1 Creando un fichero sencillo

El primer paso para manipular un fichero, ya sea para crearlo, leerlo o escribir en él es crear lo que en Python se llama una **instancia** a ese fichero; una instancia es una llamada o referencia, en este caso a un fichero, a la que se le asigna un nombre. Esto consiste simplemente en “abrir” el fichero en modo de lectura, escritura, para añadir o una combinación de estos, según lo que necesitemos:

```
In [48]: fichero_leer = open('mi_fichero.txt', 'r')
In [48]: fichero_escribir = open('mi_fichero.txt', 'w')
In [48]: fichero_escribir = open('mi_fichero.txt', 'a')
In [48]: fichero_leer = open('mi_fichero.txt', 'rw')
```

En los ejemplos anteriores se ha abierto un fichero de varias maneras posibles, donde hemos indicado en el primer parámetro el nombre del fichero y el segundo el **modo de apertura**:

```
'r'  -> Abre el fichero mi_fichero.txt para leer (debe existir previamente)
'w'  -> Abre el fichero mi_fichero.txt para escribir. Si no
          existe lo crea y si existe sobrescribirá el contenido que tenga.
'a'  -> Abre el fichero mi_fichero.txt para añadir texto. Si no
          existe lo crea y si existe continua escribiendo al final del fichero.
'rw' -> Abre el fichero mi_fichero.txt para leer y escribir
```

Vamos a crear un fichero y escribir algo en él. Lo primero es abrir el fichero en modo escritura:

```
In [49]: fs = open('prueba.txt', 'w')
In [50]: type(fs)
Out[50]: <type 'file'>
```

aquí la variable `fs` es una instancia o llamada al fichero. A partir de ahora cualquier operación que se haga en el fichero se hace utilizando esta instancia `fs` y no el nombre del fichero en sí. Escribamos ahora algo de contenido, para esto se emplea el método `write()` a la instancia del fichero:

```
In [51]: fs.write('Hola, estoy escribiendo un texto a un fichero')
In [52]: fs.write('Y esta es otra linea')
In [53]: fs.write( str(exp(10)) )
In [54]: fs.close() # Cerramos el fichero
```

Al terminar de trabajar con el fichero debemos cerrarlo con el método `close()`, es aquí cuando realmente se escribe el fichero y no hay que olvidar cerrarlo siempre al terminar de trabajar con él. Una vez cerrado, se puede abrir con un editor de textos cualquiera como Kate o gEdit. Veremos que cada orden de escritura se ha hecho consecutivamente y no línea a línea. Si queremos añadir líneas nuevas debemos ponerlas explícitamente con `\n` que es el código ASCII para el “intro” o el “return”, es decir para generar una nueva línea:

```
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()
```

De igual manera podemos usar un bucle `for` para escribir una lista de datos:

```
In [61]: fsalida = open('datos.txt', 'w')
In [62]: for i in range(100):
....:     fsalida.write('%d %10.4f\n' % (i, exp(i)))
In [63]: fsalida.close()
```

De esta forma creamos un fichero de nombre `datos.txt` en el que hay escrito, a dos columnas, los cien primeros números enteros positivos y su exponencial (con 10 caracteres en total y cuatro decimales). Podemos ver el contenido de este fichero con cualquier editor de texto o desde una consola de linux usando `cat` como ya sabemos.

9.2 Lectura de ficheros

Ahora podemos leer este fichero de datos u otro similar que ya exista. Una vez abierto el fichero para lectura, podemos crear una lista vacía para cada columna de datos y luego con un bucle leerlo línea por línea separando cada una en columnas con el método `split()`. Veámoslo con

un ejemplo: queremos leer el fichero *datos.txt* que acabamos de crear antes. Contiene 100 filas con dos columnas, la primera un número y la segunda su exponencial. Lo podríamos hacer de esta forma:

```
In [69]: fdatos = open('datos.txt', 'r')      # Abrimos el fichero "datos.txt" para lectura
In [70]: x_datos = []                          # Creamos una lista para la primera columna
In [71]: y_datos = []                          # Creamos una lista para la segunda columna
In [73]: for linea in fdatos:
....:     x, y = linea.split()                 # Se separa cada línea en dos columnas
....:     x_datos.append(float(x))            # Añado el elemento x a la lista x_datos
....:     y_datos.append(float(y))            # Añado el elemento y a la lista y_datos
....:
....:

In [75]: fdatos.close()
```

Es importante recordar que los datos (números, letras, caracteres, etc...) se escriben y leen como variables *string*. Por lo tanto, si queremos operar con ellos debemos transformarlos en los tipos de variables apropiadas (int, float, list, array, ...).

```
In [77]: type(x_datos)
Out[77]: <type 'list'>

In [78]: len(x_datos)
Out[78]: 100

In [79]: x_datos, y_datos = array(x_datos), array(y_datos)

In [80]: type(x_datos)
Out[80]: <type 'numpy.ndarray'>

In [81]: x_datos.shape
Out[81]: (100,)
```

Ahora que tenemos todos los datos en arrays los podemos manipular como tales. Recuerda que *split()* separa, por defecto, datos en el texto separados por espacios; si queremos separar por comas u otro carácter debemos incluirlo como parámetro: *split(',')'*.

9.3 Lectura y escritura de ficheros de datos con numpy

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

```
# Guardamos en el fichero "datos2.txt" (creándolo) dos columnas
# que contienen los arrays x_datos e y_datos
```

```
savetxt("datos2.txt", (x_datos, y_datos))

# Leemos el fichero que acabamos de crear y
# almacenamos los arrays en x e y
x, y = loadtxt("datos2.txt")
```

Hay que notar que si guardamos varios arrays en un fichero de esta manera, `savetxt()` guarda cada uno en una fila, de manera que `x_datos` ocupa la primera fila del fichero y `y_datos` la segunda, en lugar de en columnas y al leerlo con `loadtxt()`, la variable `x` contiene toda la primera fila e `y` la segunda. No obstante, en el caso de que sólo queramos guardar un array unidimensional, se escribe en una única columna.

Por otro lado, si leemos el fichero y cargamos los datos en un único array veamos la forma que tiene:

```
d = loadtxt("datos2.txt")
d.shape()
(2, 100)
```

es decir, un array bidimensional de 2×100 , que corresponde primero a las dos filas y luego 100 columnas. Esto quiere decir que si tenemos un fichero de texto con varias columnas (como es habitual), como `datos.txt` que contiene:

```
# 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
```

podemos leerlo de esta manera:

```
d = loadtxt("datos.txt") # d es un array de 6 x 3
t = d[0,:]    # columna 1 (tiempo)
x1 = d[1,:]   # columna 2 (x1)
x2 = d[2,:]]  # columna 1 (x2)
```

pero si queremos pasarlo directamente a *arrays* individuales debemos usar el parámetro `unpack=True`, que lo que hace es intercambiar filas por columnas (traspuesta del *array*):

```
t, x1, x2 = loadtxt("datos.txt", unpack=True)
```

Lo anterior sólo es necesario si el fichero tiene varias columnas, si sólo tiene una no hace falta usar `unpack=True`.

Supongamos ahora que estamos interesados en leer los datos de un fichero a partir de la línea 50 y no desde el principio y que además las columnas están separadas por comas en lugar de espacios; podemos indicarlo con parámetros adicionales:

```
x, y = loadtxt("datos.txt", skiprows=49, delimiter=",")
```

Consulten la ayuda de la función `loadtxt()` de `numpy` para conocer otras opciones de lectura, como seleccionar columnas determinadas, definir otro símbolo de comentario, entre otras opciones.

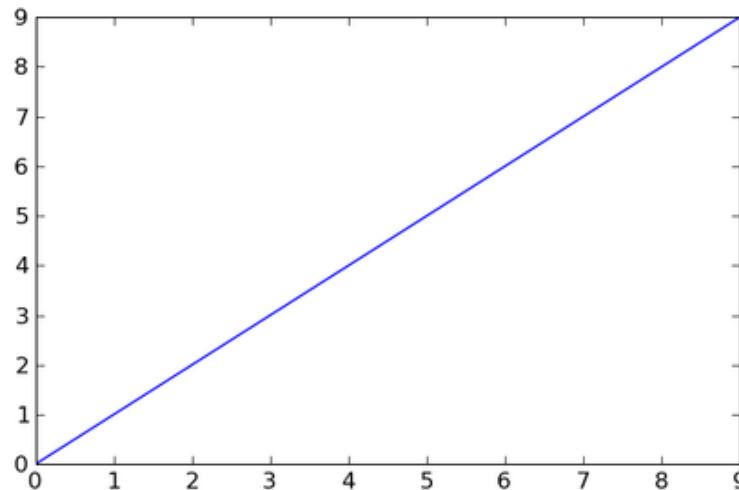
9.4 Ejercicios

1. En el fichero `datos_2col.txt` hay dos columnas de datos. Calcular la raíz cuadrada de los valores de la primera columna y el cubo de la segunda y escribirlos a un nuevo fichero de dos columnas, pero solo para las entradas cuyos valores de la segunda columna de datos original sean mayor o igual a 0.5.
2. El fichero `datos_4col.txt` contiene cuatro columnas de datos. Escribir un fichero de datos con cuatro columnas que incluya el número de entrada (empezando por 1), el promedio entre las dos primeras columnas, el promedio entre las dos últimas y la diferencia entre ambas medias.
3. Usando la función `randint()` de `numpy.random`, generen una lista de 30 números enteros aleatorios. Calcúlen la media aritmética y la desviación estándar de grupos de elementos de 5 en 5 poniendo los resultados en dos arrays diferentes; es decir, obtener un array con las medias y otro con las desviaciones estándar, que serán de longitud 30/5. Crear un fichero con tres columnas que contenga el número de línea, la media y la desviación estándar.
4. Calcule en un `array` los valores que toma la función *seno-cociente*: $\text{sen}(\theta)/\theta$ para valores de θ entre -45 y 45 grados a intervalos de 0.1 grados. Escriba el resultado en un fichero que incluya en una columna el ángulo θ en grados y en otra el valor del seno-cociente correspondiente.

Representación gráfica de funciones y datos

La representación de funciones y/O datos científicos mediante gráficos resulta ser fundamental para expresar una gran variedad de resultados. Cualquier informe, artículo o resultado a menudo se expresa de manera mucho más clara mediante gráficos. Python posee varios paquetes gráficos; nosotros usaremos `matplotlib`, una potente librería gráfica de alta calidad también para gráficos bidimensionales y sencilla de manejar. Matplotlib posee el módulo `pylab`, que es la interfaz para hacer gráficos bidimensionales. Veamos un ejemplo sencillo:

```
>>> from pylab import *      # importar todas las funciones de pylab
>>> y = arange(10.)          # array de floats, de 0.0 a 9.0
>>> plot(y)                 # generar el gráfico de la función y=x
>>> [<matplotlib.lines.Line2D object at 0x9d0f58c>]
>>> show()                  # mostrar el gráfico en pantalla
```



Hemos creado un gráfico que representa diez puntos en un *array* y luego lo hemos mostrado con `show()`; esto es así porque normalmente solemos hacer varios cambios en la gráfica, mostrándolos todos juntos. Sin embargo, cuando trabajamos interactivamente, por ejemplo con la consola `ipython` podemos activar el **modo interactivo** para que cada cambio que se haga en la gráfica se muestre en el momento, mediante la función `ion()`, de esta manera no hace falta poner `show()` para mostrar la gráfica cada vez que se haga `plot()`:

```
In [1]: ion()                      # Activo el modo interactivo
In [2]: plot(x)                    # Hago un plot que se muestra sin hacer show()
Out[2]: [<matplotlib.lines.Line2D object at 0x9ffde8c>]
```

Recuerden que este modo interactivo sólo está disponible en la consola avanzada `ipython` pero no lo está en la consola estándar de Python. Otra posibilidad es iniciar `ipython` en modo `pylab`, haciendo `ipython -pylab`, de esta manera se carga automáticamente `pylab`, se activa el modo interactivo, y además se importa el módulo `numpy` y todas sus funciones.

Fíjense cómo el comando `plot()` que hemos usado hasta ahora devuelve una lista de instancias de cada dibujo. En este caso es una lista con un sólo elemento, una instancia `Line2D`. Podemos capturar esta instancia para referirnos a este dibujo más adelante haciendo:

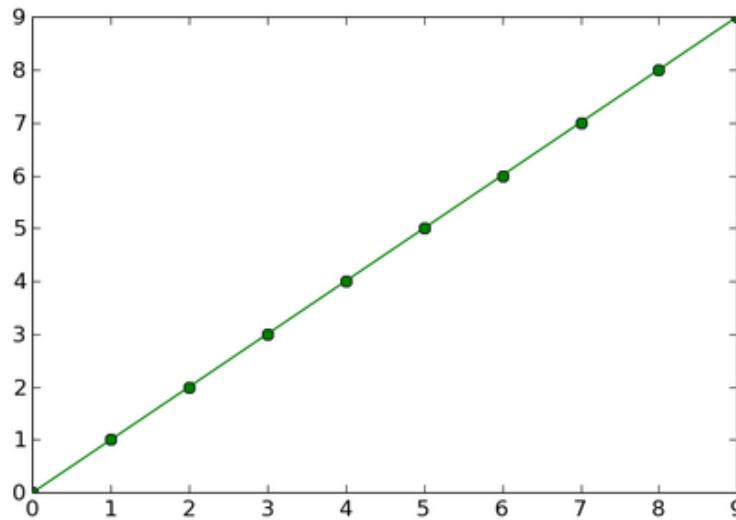
```
In [3]: mi_dibujo, = plot(x)
```

Ahora la variable `mi_dibujo` es una instancia o “referencia” a la línea del dibujo, que podremos manipular posteriormente con métodos que se aplican a esa instancia. Nótese que después de `mi_dibujo` hay una coma; esto es para indicar que `mi_dibujo` debe tomar el valor del primer (y en este caso el único) elemento de la lista y no la lista en sí, que es lo que habría ocurrido de haber hecho `mi_dibujo = plot(x)` (erróneamente).

La sintaxis básica de `plot()` es simplemente `plot(x, y)`, pero si no se incluye la lista `x`, ésta se reemplaza por el **número de elemento** o índice de la lista `y`, por lo que es equivalente a hacer `plot(range(len(y)), y)`. En la gráfica del ejemplo anterior no se ven diez puntos, sino una línea continua uniendo esos puntos, que es como se dibuja por defecto. Si queremos pintar los puntos debemos hacerlo con un parámetro adicional, por ejemplo:

```
In [4]: plot(x,'o')    # pinta 10 puntos como o
Out[4]: [<matplotlib.lines.Line2D object at 0x8dd3cec>]
```

```
In [5]: plot(x,'o-')   # igual que antes pero ahora los une con una linea continua
Out[5]: [<matplotlib.lines.Line2D object at 0x8dd9e0c>]
```



En este caso el ‘o’ se usa para dibujar puntos gruesos y si se añade ‘-‘ también dibuja la línea continua. En realidad, lo que ha sucedido es que se dibujaron dos gráficos uno encima del otro; si queremos que se cree un nuevo gráfico cada vez que hacemos `plot()`, debemos añadir el parámetro `hold=False` a `plot()`:

```
mi_dibujo, = plot(x*2, 'o', hold="False")
```

El tercer parámetro de la función `plot()` (o segundo, si no se incluye la variable `x`) se usa para indicar el símbolo y el color del marcador. Admite distintas letras que representan de manera única el color, el símbolo o la línea que une los puntos; por ejemplo, si hacemos `plot(x, 'bx-')` pintará los puntos con marcas “x”, de color azul (“b”) y los unirá además con líneas continuas del mismo color. A continuación se indican otras opciones posibles:

Colores

Símbolo	Color
‘b’	Azul
‘g’	Verde
‘r’	Rojo
‘c’	Cian
‘m’	Magenta
‘y’	Amarillo
‘k’	Negro
‘w’	Blanco

Marcas y líneas

Símbolo	Descripción
‘_’	Línea continua
‘_’	Línea a trazos
‘_.’	Puntos y rayas
‘:’	Línea punteada
‘.’	Marcador punto
‘;’	Marcador pixel
‘o’	Marcador círculo relleno
‘v’	Marcador triángulo hacia abajo
‘^’	Marcador triángulo hacia arriba
‘<’	Marcador triángulo hacia la izquierda
‘>’	Marcador triángulo hacia la derecha
‘s’	Marcador cuadrado
‘p’	Marcador pentágono
‘*’	Marcador estrella
‘+’	Marcador cruz
‘x’	Marcador X
‘D’	Marcador diamante
‘d’	Marcador diamante delgado

Para borrar toda la figura se puede usar la función `clf()`, mientras que `cla()` sólo borra lo que hay dibujado dentro de los ejes y no los ejes en si.

Se pueden representar varias **parejas de datos** con sus respectivos símbolos en una misma figura, aunque para ello siempre es obligatorio incluir el valor del eje x:

In [8]: `clf() # Limpiamos toda la figura`

In [9]: `x2=x**2 # definimos el array x2`

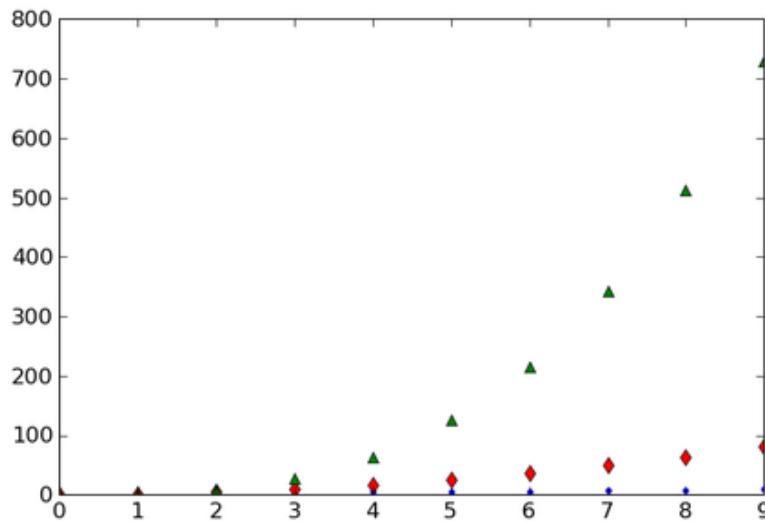
In [10]: `x3=x**3 # definimos el array x3`

In [11]: `# dibujamos tres curvas en el mismo gráfico y figura`

In [12]: `plot(x, x,'b.', x, x2,'rd', x, x3,'g^')`

Out[13]:

```
[<matplotlib.lines.Line2D object at 0x8e959cc>,
<matplotlib.lines.Line2D object at 0x8eb75cc>,
<matplotlib.lines.Line2D object at 0x8eb788c>]
```



Es posible cambiar el intervalo mostrado en los ejes con `xlim()` e `ylim()`:

In [12]: `xlim(-1,11)` #nuevos límites para el eje OX
Out[12]: (-1, 11)

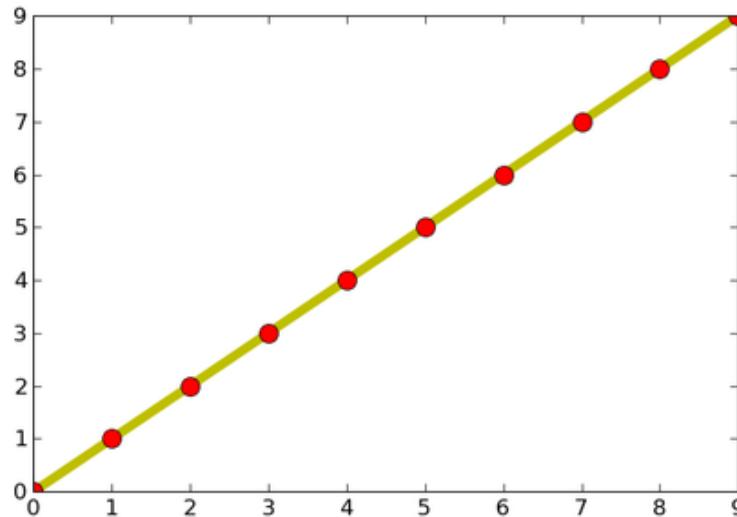
In [13]: `ylim(-50,850)` #nuevos límites para el eje OY
Out[13]: (-50, 850)

Además del marcador y el color indicado de la manera anterior, se pueden cambiar muchas otras propiedades de la gráfica como parámetros de `plot()` independientes como los de la tabla adjunta:

Parámetro	Valores
alpha	float (0.0=transparente a 1.0=opaco)
color o c	Un color de matplotlib
label	string (cadena de texto)
markeredgecolor o mec	Un color de matplotlib
markeredgewidth o mew	float en puntos
markerfacecolor o mfc	Un color de matplotlib
markersize o ms float	float en puntos
linestyle o ls	'-' '--' '-.' ':' 'None'
linewidth o lw	float en puntos
marker	'+' '*' ',' '.' '1' '2' '3' '4' '<' '>' 'D' 'H' '^' '_' 'd' 'h' 'o' 'p' 's' 'v' 'x' 'l' TICKUP TICKDOWN TICKLEFT TICKRIGHT

Un ejemplo usando más opciones sería este:

```
In [15]: plot(x, lw='5', c='y', marker='o', ms=10, mfc='red')
Out[15]: [<matplotlib.lines.Line2D object at 0x8f0d14c>]
```



También es posible cambiar las propiedades de la gráfica una vez creada, para ello debemos **capturar las instancias** de cada dibujo en una variable y cambiar sus parámetros. En este caso a menudo hay que usar `show()` para actualizar el gráfico,:

```
In [15]: # Hago tres dibujos, capturando sus instancias en las variables p1, p2 y p3
In [16]: p1, p2, p3 = plot(x, x,'b.',x, x2, 'rd', x, x3, 'g^')

In [17]: p1.set_marker('o')                      # Cambio el símbolo de la gráfica 1
In [18]: p3.set_color('y')                      # Cambio el color de la gráfica 3
In [19]: show()                                # Muestro en dibujo por pantalla
```

10.1 Trabajando con texto dentro del gráfico

Existen funciones para añadir texto (etiquetas) a los ejes de la gráfica y a la gráfica en sí; éstos son los más importantes:

```
In [9]: x = arange(0, 5, 0.05)

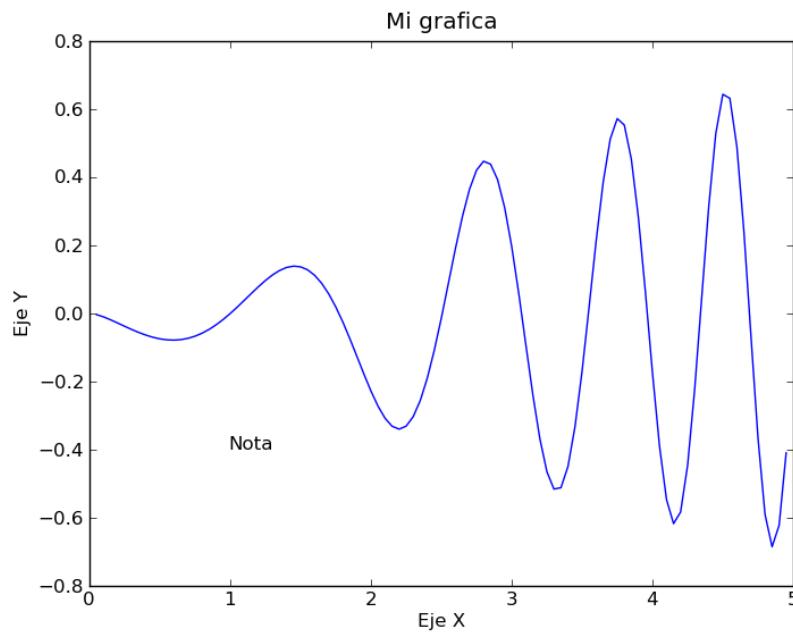
In [10]: p, = plot(x,log10(x)*sin(x**2))

In [12]: xlabel('Eje X')           # Etiqueta del eje OX
Out[12]: <matplotlib.text.Text object at 0x99112cc>

In [13]: ylabel('Eje Y')           # Etiqueta del eje OY
Out[13]: <matplotlib.text.Text object at 0x99303cc>

In [14]: title('Mi grafica')      # Título del gráfico
Out[14]: <matplotlib.text.Text object at 0x993802c>

In [15]: text(1, -0.4, 'Nota')    # Texto en coordenadas (1, -0.4)
```



En este ejemplo, se usó la función `text()` para añadir un texto arbitrario en la gráfica, cuya posición se debe dar en **las unidades de la gráfica**. Cuando se utilizan textos también es posible usar fórmulas con formato LaTeX. Veamos un ejemplo,:

```
In [28]: clf()      #limpio la figura

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]: text01 = text(2, 0.6, r'$\frac{\sin(x)}{x}$', fontsize=20)

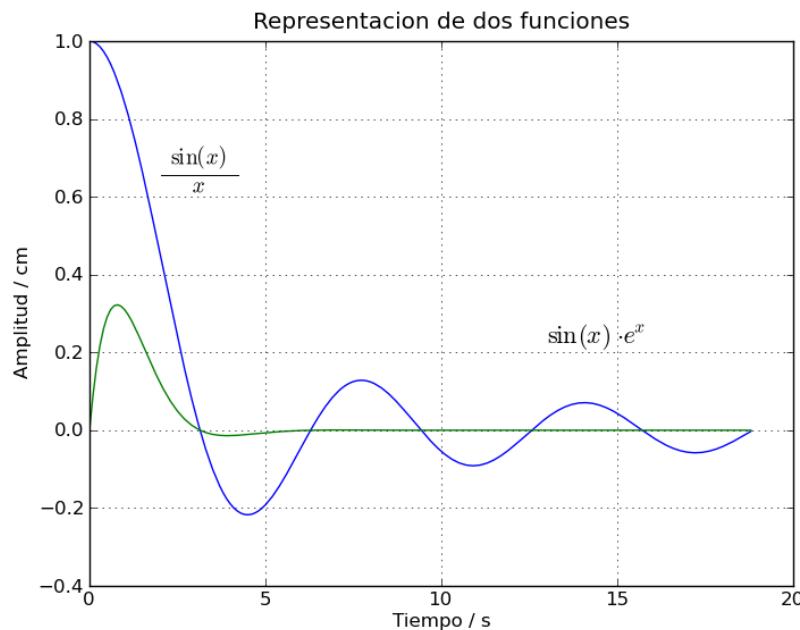
In [34]: texto2 = text(13, 0.2, r'$\sin(x) \cdot e^{x}$', fontsize=16)

In [35]: grid()          # Añado una malla al gráfico

In [36]: title('Representacion de dos funciones')
Out[36]: <matplotlib.text.Text object at 0x99fb78c>

In [37]: xlabel('Tiempo / s')
Out[37]: <matplotlib.text.Text object at 0x94f172c>

In [38]: ylabel('Amplitud / cm')
Out[38]: <matplotlib.text.Text object at 0x94f20ec>
```



Aquí hemos usado código LaTeX para escribir fórmulas matemáticas, para lo que siempre hay que escribir entre `r'$ formula '$` y he usado un tamaño de letra mayor con el parámetro `fontsize`. En la última línea hemos añadido una malla con la función `grid()`.

Nota: LaTeX es un sistema de escritura orientado a contenidos matemáticos muy popular en ciencia e ingeniería. Puedes ver una introducción a LaTeX en los cursos CISLA de la ULL: <http://csla.osl.ull.es/octubre06/htc/apuntes/latex>.

10.2 Representación gráfica de funciones

Visto el ejemplo anterior, vemos que en Python es muy fácil representar gráficamente una función matemática. Para ello, debemos definir la función y luego generar un *array* con el intervalo de valores de la variable independiente que se quiere representar. Definamos algunas funciones trigonométricas y luego representémoslas gráficamente:

```
>>> def f1(x):
.....:     y = sin(x)
.....:     return y
.....:

>>> def f2(x):
.....:     y = sin(x)+sin(5.0*x)
.....:     return y
.....:
```

```
>>> def f3(x):
....:     y = sin(x)*exp(-x/10.)
....:     return y
....:

>>> # array de valores que quiero representar
>>> x = arange(0, 10*pi, 0.1)

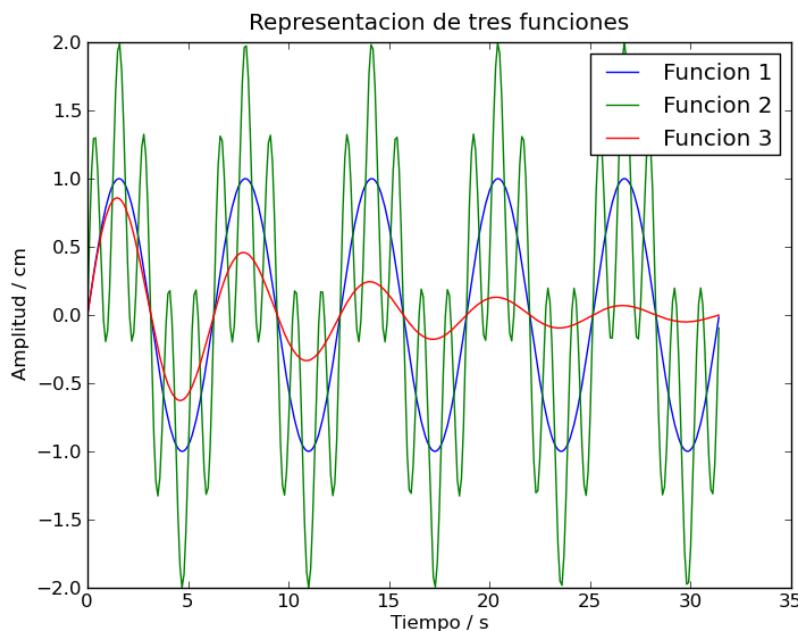
>>> p1, p2, p3 = plot(x, f1(x), x, f2(x), x, f3(x))

>>> # Añado una leyenda al gráfico
>>> legend( ('Funcion 1', 'Funcion 2', 'Funcion 3') )
>>> <matplotlib.legend.Legend object at 0xbb4b0ac>

>>> xlabel('Tiempo / s')
>>> <matplotlib.text.Text object at 0xa06764c>

>>> ylabel('Amplitud / cm')
>>> <matplotlib.text.Text object at 0xa0c32cc>

>>> title('Representacion de tres funciones')
>>> <matplotlib.text.Text object at 0xa0c3e8c>
```



Nótese que hemos añadido una leyenda con la función `legend()` que admite como entrada una **tupla** con *strings* que corresponden consecutivamente a cada una de las curvas del gráfico.

10.3 Histogramas

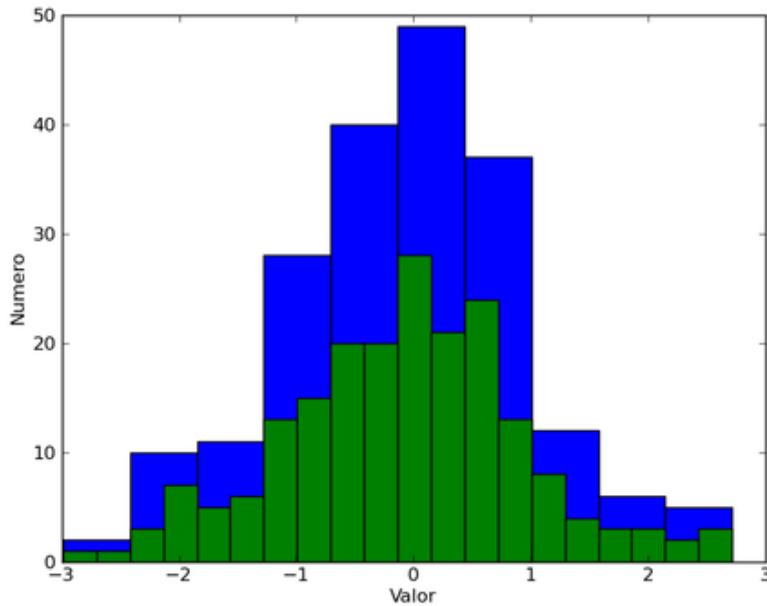
Cuando tenemos un conjunto de datos numéricos, por ejemplo como consecuencia de la medida de una cierta magnitud y queremos representarlos gráficamente para ver la distribución subyacente de los mismos se suelen usar los gráficos llamados histogramas(ya lo vimos en capítulos anteriores). Los histogramas representan el número que veces que los valores del conjunto caen dentro de un intervalo dado, frente a los diferentes intervalos en los que queramos dividir el intervalo de variación general de los valores del conjunto. En Python podemos hacer histogramas muy fácilmente con la función `hist()` indicando como parámetro un *array* con los números del conjunto. Si no se indica nada más, se generará un histograma con 10 intervalos (llamados *bins*, en inglés) en los que se divide la diferencia entre el máximo y el mínimo valor del conjunto. Veamos un ejemplo:

```
>>> # Importo el módulo de numeros aleatorios de scipy
>>> from scipy import random
>>> # utilizo la función randn() del modulo random para generar
>>> # un array de números aleatorios con distribución normal
>>> nums = random.randn(200) # array con 200 números aleatorios
>>> # Genero el histograma
>>> hist(nums)
>>>
(array([ 2, 10, 11, 28, 40, 49, 37, 12, 6, 5]),
array([-2.98768497, -2.41750815, -1.84733134, -1.27715452, -0.70697771,
       -0.13680089,  0.43337593,  1.00355274,  1.57372956,  2.14390637,
       2.71408319]),
<a list of 10 Patch objects>)
```

Vemos que los números del *array* se dividieron automáticamente en 10 intervalos (o *bins*) y cada barra representa para cada uno de ellos el número de valores que caen dentro. Si en lugar de usar sólo 10 divisiones queremos usar 20 por ejemplo, debemos indicarlo como un segundo parámetro:

```
>>> hist(nums, bins=20)
```

El la figura de abajo se muestra el resultado de superponer ambos histogramas. Nótese que la función `hist()` devuelve una tupla con tres elementos, que son un array con el número elementos en cada intervalo, un array con el punto del eje *OX* donde empieza cada intervalo y una lista con referencias a cada una de las barras para modificar sus propiedades (consulten el manual de `matplotlib` para encontrar más información y mayores posibilidades de uso).



10.4 Figuras diferentes

Se pueden hacer cuantas figuras independientes (en ventanas distintas) queramos con la función `figure(n)` donde n es el número de la figura. Cuando se crea una figura al hacer `plot()` se hace automáticamente `figure(1)`, como aparece en el título de la ventana. Podríamos crear una nueva figura independiente escribiendo `figure(2)`, en ese momento todos los comandos de aplican a figura activa, la figura 2. Podemos regresar a la primera escribiendo `figure(1)` para trabajar nuevamente en ella, por ejemplo:

```
>>> p1, = plot(sin(x))      # Crea una figura en una ventana (Figure 1)
>>> figure(2)                # Crea una nueva figura vacía en otra ventana (Figure 2)
>>> p2, = plot(cos(x))      # Dibuja el gráfico en la figura 2
>>> title('Funcion coseno')  # Añade un título a la figura 2
>>> figure(1)                 # Activo la figura 1
>>> title('Funcion seno')    # Añade un título a la figura 2
```

10.5 Varios gráficos en una misma figura

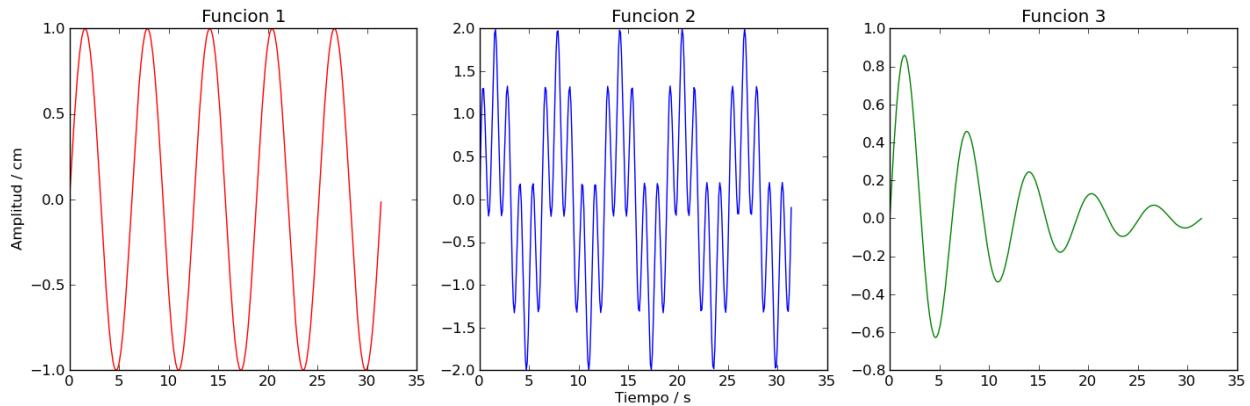
En ocasiones nos interesa mostrar varios gráficos diferentes en un misma figura o ventana. Para ello podemos usar la función `subplot()`, indicando entre paréntesis un número con tres dígitos. El primer dígito indica el número de filas en los que se dividirá la figura, el segundo el número de columnas y el tercero se refiere al gráfico con el que estamos trabajando en ese momento. Por

ejemplo, si quisieramos representar las tres funciones anteriores usando tres gráficas en la misma figura, una al lado de la otra y por lo tanto con una fila y tres columnas, haríamos lo siguiente:

```
>>> # Figura con una fila y tres columnas, activo primer subgráfico
>>> subplot(131)
>>> p1, = plot(x,f1(x),'r-')
>>> # Etiqueta del eje Y, que es común para todas
>>> ylabel('Amplitud / cm')
>>> title('Funcion 1')

>>> # Figura con una fila y tres columnas, activo segundo subgráfico
>>> subplot(132)
>>> p2, = plot(x,f2(x),'b-')
>>> # Etiqueta del eje X, que es común para todas
>>> xlabel('Tiempo / s')
>>> title('Funcion 2')

>>> # Figura con una fila y tres columnas, activo tercer subgráfico
>>> subplot(133)
>>> p3, = plot(x, f3(x),'g-')
>>> title('Funcion 3')
```



Al igual que con varias figuras, para dibujar en un gráfico hay que activarlo. De esta forma, si acabamos de dibujar el segundo gráfico escribiendo antes `subplot(132)` y queremos cambiar algo del primero, debemos activarlo con `subplot(131)` y en ese momento todas funciones de gráficas que hagamos se aplicarán a él.

10.6 Representando datos experimentales

Cuando estamos en el laboratorio (u observatorio) haciendo un experimento y midiendo algún parámetro físico solemos obtener un conjunto de números que normalmente guardaremos en un fichero de texto en cualquier ordenador. Representar datos leídos de un fichero en lugar de gene-

rarlos directamente como hemos hecho hasta ahora, es tan sencillo como leer los datos y pasarlos a *arrays* de numpy. Una vez hecho, se representan gráficamente como ya hemos visto. Supongamos que tenemos los resultados de un experimento en una tabla de dos columnas almacenada en el fichero “datos_2col.txt”, para poder dibujarlos en un gráfico hacemos,:

```
>>> # Leemos el fichero de dos columnas de datos, pasándolo a un array
>>> datos = loadtxt('datos_2col.txt')
>>> datos.shape
>>> (100, 2)
>>> # es decir, 100 filas, 2 columnas

>>> col2, = plot(datos[:,1], 'b.')    # Segunda columna, con puntos azules (b)
>>> col1, = plot(datos[:,0], 'r.')    # Primera columna, con puntos rojos (r)

>>> # Trazo una línea horizontal en la coordenada y=4 de color verde (g)
>>> axhline(4, color='g')
>>> <matplotlib.lines.Line2D object at 0x169c6d8c>

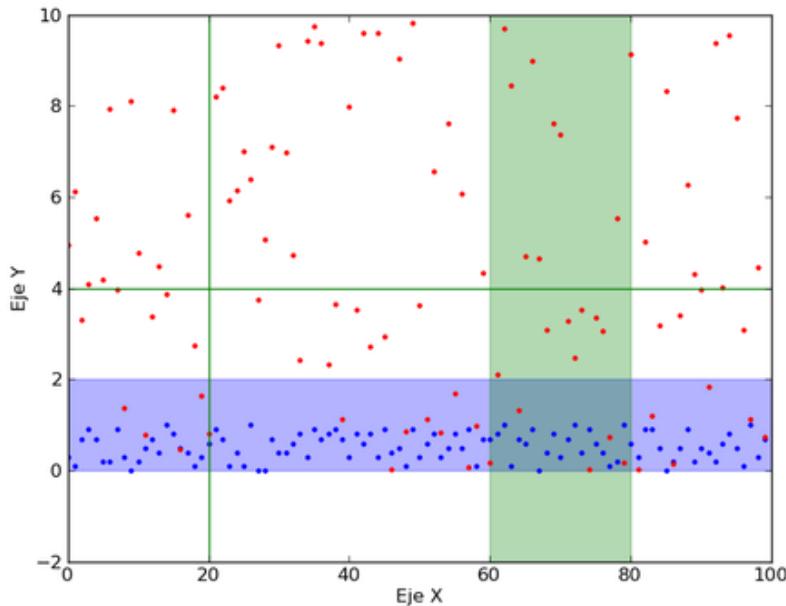
>>> # Trazo una línea vertical en la coordenada x=30 de color verde (g)
>>> axvline(20, color='g')
>>> <matplotlib.lines.Line2D object at 0xac5986c>

>>> # Dibujo una banda horizontal de y=0 a y=2 de color azul
>>> # y 30% de transparencia (alpha=0.3)
>>> axhspan(0, 2, alpha=0.3, color='b')
>>> <matplotlib.patches.Polygon object at 0xac59c4c>

>>> # Dibujo una banda vertical de x=60 a x=80 de color verde
>>> # y 30% de transparencia
>>> axvspan(60, 80, alpha=0.3, color='g')
>>> <matplotlib.patches.Polygon object at 0xac59a0c>

>>> # Etiqueto los ejes
>>> xlabel('Eje X')
>>> <matplotlib.text.Text object at 0xad043ac>
>>> ylabel('Eje Y')
>>> <matplotlib.text.Text object at 0xad0b52c>
```

En este caso hemos usado además algunas funciones para crear líneas y bandas horizontales y verticales.



10.7 Datos experimentales con barras de error

Cuando se trabaja con datos de laboratorio es muy habitual tener errores asociados a los datos que se van tomando; al representarlos gráficamente después es muy conveniente dibujar también las barras de error de cada dato tomado. En Python esto se puede hacer fácilmente usando la función `errorbar()` en lugar de `plot()` o junto con ella. A la hora de usarla hay que incluir en su sintaxis los errores como parámetros usando *floats* si son errores iguales para todos los puntos o bien un array representando el error de cada punto. Veamos un ejemplo:

```
>>> # Datos de x e y
>>> x = arange(0.1, 5.0, 0.1)
>>> y = exp(-x)

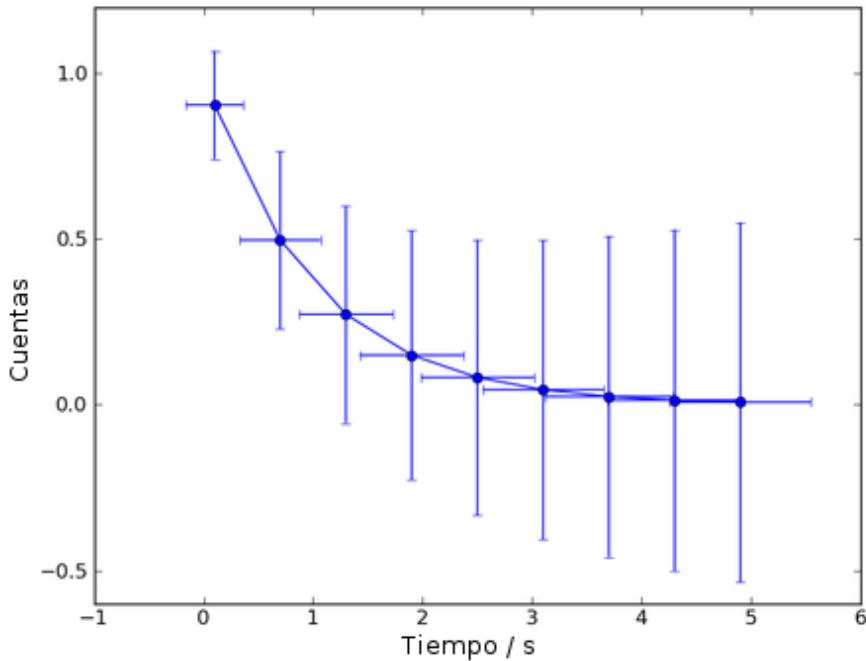
>>> # Error constante en x e y
>>> err_x = 0.1
>>> err_y = 0.2

>>> errorbar(x, y, xerr=err_x, yerr=err_y)

>>> # Si los errores de x e y son distintos en cada punto,
>>> # se ponen en un array. Supongamos que sean:
>>> err_y = 0.1 + 0.2*sqrt(x)
>>> err_x = 0.1 + err_y

>>> # Gráfico con la barra de error en x e y, usando
```

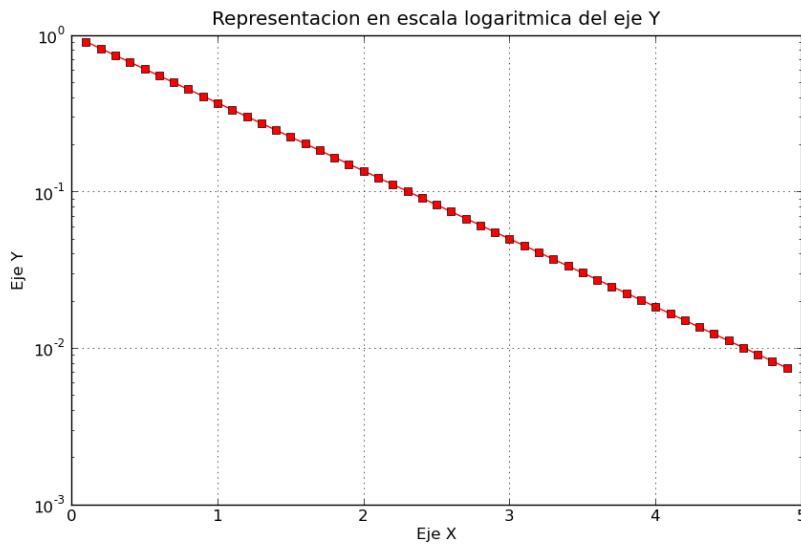
```
>>> # línea continua y puntos (fmt='-o')
>>> errorbar(x, y, xerr=err_x, yerr=err_y, fmt='-o')
```



Hay veces que para algunos tipos de datos, conviene representar alguno de los ejes o ambos en escala logarítmica para apreciar mejor la evolución de la gráfica. Podemos usar las funciones `semilogx()`, `semilogy()` o `loglog()` para hacer un gráfico en escala logarítmica en el eje x, en el eje y o en ambos, respectivamente. Por ejemplo, para representar el gráfico anterior con el eje y en escala logarítmica, podemos hacer lo siguiente:

```
>>> # Eje y en escala logarítmica
>>> p1, = semilogy(x, y,'rs-')

>>> grid()
>>> xlabel('Eje X')
>>> ylabel('Eje Y')
>>> title('Representacion en escala logaritmica del eje Y')
```

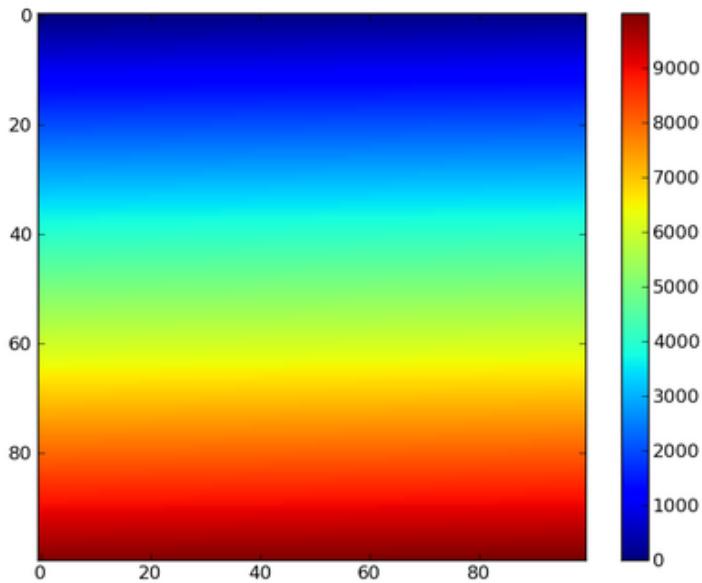


10.8 Representación de datos bidimensionales (imágenes)

Los datos bidimensionales como su nombre indica son valores de una magnitud física representada por una función que tiene dos variables independientes, normalmente x e y ; se trata pues de representar funciones del tipo $z=z(x,y)$, donde z puede representar flujo luminoso, presión atmosférica, altura del terreno, etc... Hay varias formas de representar estos datos bidimensionales (como podría ser una imagen), una de ellas es usar escalas de colores para representar los diferentes valores de z para pares de valores (x,y) . En Python lo hacemos usando la función `imshow()`:

```
>>> # Creo un array 1D 100x100 de valores de 0.0 a 99999.0
>>> datos1D = arange(10000.) # array unidimensional con 10000 elementos
>>> # Ahora cambio la forma (reshape) del array redistribuyendo sus elementos,
>>> # de 10000 en 1D a un array de 100 filas y 100 columnas en 2D,
>>> # como si fuese una imagen.
>>> datos2D = datos1D.reshape(100,100)
>>> datos2D.shape
>>> (100, 100)

>>> # Representamos gráficamente el array bidimensional
>>> imshow(datos2D)
>>> # Añadimos una paleta (barra de color) para indicar la equivalencia
>>> # de los colores con los valores de la función
>>> colorbar()
>>> # Cambio a la paleta de colores gray() (por defecto es jet())
>>> gray()
```



10.9 Guardando las figuras creadas

Después de crear una figura con cualquiera de los procedimientos descritos hasta ahora podemos guardarla con la función `savefig()` poniendo como parámetro el nombre del fichero con su extensión. El formato de grabado se toma automáticamente de la extensión del nombre. Los formatos disponibles en Python son los más usuales: png, pdf, ps, eps y svg. Por ejemplo:

```
>>> savefig("mi_primer_grafica.eps") # Guardo la figura en formato eps  
>>> savefig("mi_primer_grafica.png") # Guardo la figura en formato png
```

Si el gráfico se va usar para imprimir, por ejemplo en una publicación científica o en un informe, es recomendable usar un formato vectorial como Postscript (ps) o Postscript encapsulado (eps), pero si es para mostrar por pantalla o en una web, el más adecuado es un formato de mapa de bits como png o jpg.

Consulta la web de matplotlib (<http://matplotlib.sourceforge.net/>) para ver muchas más propiedades y ejemplos de esta librería.

10.10 Ejercicios

1. Representar gráficamente las siguientes funciones:

$$f(x) = ae^{-\frac{(x-x_0)^2}{2c^2}} \quad f(x) = \frac{b}{(x - x_0)^2 + b^2}$$

usando los valores $a=2.0$, $x_0 = 10.0$, $c=5.0$ y $b=0.5$, en el intervalo de x [-50,+50]. Comprueba cómo afecta a las gráficas distintos valores de los parámetros c y b .

- La curva plana llamada trocoide, una generalización de la cicloide, es la curva descrita por un punto P situado a una distancia b del centro de una circunferencia de radio a , a medida que rueda (sin deslizar) por una superficie horizontal. Tiene por coordenadas (x,y) las siguientes:

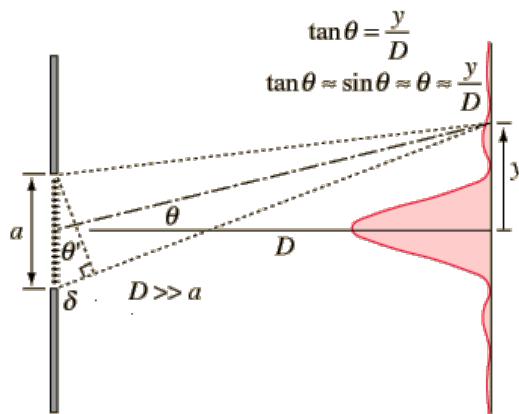
$$x = a\phi - b\sin\phi \quad , \quad y = a - b\cos\phi$$

Escribir un programa que dibuje tres curvas (contínuas y sin símbolos), en el mismo gráfico cartesiano (OX,OY), para un intervalo $\phi = [0.0, 18.0]$ (en radianes) y para los valores de $a=5.0$ y $b=2.0$, 5.0 y 8.0 . Rotular apropiadamente los ejes e incluir una leyenda con los tres valores de que distinguen las tres curvas.

- Dibujar las diferentes trayectorias de los proyectiles disparados por un cañón situado en un terreno horizontal para diferentes ángulos de elevación (inclinación respecto de la horizontal) en un intervalo de tiempo de 0 a 60 s. El cañón proporciona una velocidad inicial de 300 m/s. Dibujarlas para los ángulos de elevación siguientes: 20, 30, 40, 50, 60 y 70 grados y suponer que el cañón está situado en el origen de coordenadas. Rotular apropiadamente los ejes e insertar una leyenda que identifique las diferentes trayectorias. Recordar que el proyectil no puede penetrar en el suelo de forma que hay que establecer los límites apropiados para el dibujo.
- Cuando una fuente de luz coherente atraviesa una rendija delgada, se produce difracción de la luz, cuyo patrón de intensidad en la aproximación de Fraunhofer está dado por:

$$I(\theta) = I_0 \left(\frac{\sin \beta}{\beta} \right)^2 \quad \beta = \frac{\pi a \sin \theta}{\lambda}$$

donde a es el ancho de la rendija, λ la longitud de onda de la luz, I_0 la intensidad en el eje y θ el ángulo de la posición medida con el eje de la rendija (ver dibujo). Representar gráficamente la intensidad del patrón de difracción para $\lambda = 400nm$, $\lambda = 650nm$ y $\lambda = 800nm$ usando $I_0 = 1$ y $a=0.04mm$ en el intervalo $-\pi/20 < \theta < +\pi/20$. Comprobar cual es el efecto del patrón de difracción al duplicar el ancho de la rendija.



5. La variación de temperatura de un objeto a temperatura T_0 en un ambiente a T_s cambia de la siguiente manera:

$$T = T_s + (T_0 - T_s)e^{-kt}$$

con t en horas y k un parámetro que depende del objeto. a) Representar gráficamente la variación de la temperatura con el tiempo, partiendo de una $T_0 = 5^\circ\text{C}$ a lo largo de 24 horas suponiendo $k=0.45$ y temperatura ambiente de 40°C . b) Superponer sobre esta curva las curvas correspondientes a otros objetos con $k=0.3$ y $k=0.6$ con distinto color y trazado, identificándolas con una leyenda.

6. Representen nuevamente la curva del apartado a) del ejercicio anterior superponiendo además las curvas correspondientes a temperaturas iniciales distintas, de $T_0 = -5^\circ\text{C}$ y $T_0 = 15^\circ\text{C}$. Para $T_0 = 5^\circ\text{C}$ representen en una figura aparte cómo cambian las curvas con temperaturas ambiente de 20°C y 50°C , además de la de 40°C . Identifiquen cada curva y etiqueta correctamente los ejes en todas las gráficas.
7. Con la serie de Gregory-Leibnitz para el cálculo de π usada anteriormente:

$$4 \sum_{k=1}^n \frac{(-1)^{k+1}}{2k-1}$$

el valor obtenido de π se acerca lentamente al verdadero con cada término que se añade. Calcúlen todos los valores que va tomando π con cada término añadido hasta llegar a un error absoluto de 10^{-6} y representen en una figura con dos gráficas (usando `subplot()`) los primeros 300 valores que toma π frente al número de términos usados en una de ellas y los 300 últimos en la otra (usa líneas continuas). En otra figura distinta con otras dos gráficas análogamente representen el valor absoluto de la diferencia entre el valor calculado y el real frente al número de elementos.

8. El fichero `medidas_I131.txt` (directorio de datos en el aula virtual) contiene medidas de masa de yodo 131 radioactivo hechas diariamente para medir su coeficiente de desintegración.

La primera columna es la masa residual en gramos y la segunda es el error de la medida. Representen gráficamente las medidas de masa frente al tiempo incluyendo barras de error usando puntos sin unir con líneas y etiquetando los ejes apropiadamente.

9. El movimiento de un oscilador amortiguado se puede expresar la siguiente manera:

$$x = A_0 e^{-k\omega t} \cos(\omega t + \delta)$$

Siendo A_0 la amplitud inicial, ω la frecuencia angular de oscilación y k el factor de amortiguamiento. Representar gráficamente el movimiento de un oscilador amortiguado de amplitud inicial de 10 cm y frecuencia de 10 ciclos/s y $\delta = \pi/8$ con factores de amortiguamiento de 0.1, 0.4, 0.9 y 1.1 durante 10 s. Incluya una leyenda identificativa de las curvas dibujadas.

Para el gráfico correspondiente a $k=0.1$ unir con líneas a trazos los valores máximos por un lado y los valores mínimos por otro del movimiento oscilatorio. Nótese que corresponden a las curvas para las que $x = A_0$ y $x = -A_0$.

10. La curva plana llamada epicicloide, tiene por coordenadas cartesianas (x,y) las siguientes:

$$x = (a + b) \cos \phi - b \cos\left(\frac{a}{b} + 1\right)\phi, \quad y = (a + b) \sin \phi - b \sin\left(\frac{a}{b} + 1\right)\phi$$

Escribir un programa en el que se dibuje la curva (continua y sin símbolo) para un intervalo $\phi = [0, 6\pi]$ y para los valores de $a = 1/3$ y $b = 1/8$. Rotular apropiadamente los ejes e incluir en el título los valores de a , b y el intervalo de valores de ϕ utilizado.