# Computación Científica - Introducción a Python

***Release 0.1***

**Jorge A Pérez Prieto y Teodoro Roca Cortés**

January 30, 2011

# Índice general

Contenido:

CAPÍTULO 1

# Introducción a Python

Python puede utilizarse como un programa ejecutable desde una terminal de comandos o de manera interactiva mediante una consola de Python. Python incorpora una consola por defecto, pero existen otras con características útiles para el análisis científico de datos. **ipython** es 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 etc.; es la consola de Python que usaremos para este curso.

## 1.1 Empezando con Python

Para inicial la consola IPython sólo hay que escribir `ipython` (o `python` para la consola estándar),en la terminal de comandos de Linux y Mac o ejecutar el programa desde el menú de Windows:

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

Para obtener ayuda sobre un comando basta escribir el comando seguido de "?":

```
>>> 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.
```

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 trabajamos con la terminal, puede ser muy útil guardar todo lo que vamos haciendo para usarlo porteriormente o continuar donde estábamos. Al iniciar ipython, se puede empezar grabar la sesión con el comando **%logstart**, por ejemplo:

```
%logstart -o registro_3_octubre_2009.txt
```

donde el parámetro opcional `-o` guarda también la salida a modo de comentario en el fichero registro_3_octubre_2010.txt. Al iniciar `ipython` en otro ocasión, podemos hacer que antes lea ese fichero de sesión escribiendo:

```
ipython -lp registro_3_octubre_2010.txt
```

para recuperar todo casi como lo teníamos, en medida de lo posible.

## 1.2 Tipos básicos de datos

En cualquier lenguaje de programación existen distintos tipos de datos, que podemos almacenar y operar de forma diferente y que poseen distintas propiedades. Los más comunes son las **cadenas de texto** o **string**, indicadas siempre entre comillas y los **números**.:

```
>>> print("Esta es una linea de texto")
>>> Esta es una linea de texto
>>> print(28)
>>> 28
```

La **variables** son un componente fundamental de un lenguaje de programación y no son más que un nombre que se refiere a un registro que contiene uno o varios datos, que puede ser de distinto tipo:

```
>>> 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'
```

Nótese que la cadena de texto "frase" fué duplicada al multiplicarla por dos. Con el comando `type()` de Python podemos saber el tipo de dato que es una variable:

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

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

Las variables pueden ser cualquier combinación de letras y números (siempre que no empiece por número) y pero no están permitidos caracteres especiales como tildes, puntos, espacios en blanco, etc. Algunas palabras están además reservadas ya por el propio Python, como `def`, `class`, `return`, etc. por lo que tampoco se pueden usar como variables.

Nótese que `num` es un entero `int` 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:

```
>>> 113/27
>>> 4        # en lugar de 4.18
```

Si queremos usar números 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'>
```

Python emplea número 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 o equivocado.

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

```
>>> 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 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 nuevas funciones y cómo usarlos.

## 1.3 Operadores aritméticos

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: **; *, /, %; +, - .

## 1.4 Operadores lógicos

Estos operadores permiten comparar valores entre sí:

| Operacion | Simbolo |
|---|---|
| Igualdad (comparación) | == |
| Mayor/Menor | >, < |
| Mayor o igual/Menor o igual | >=, =< |
| and or | y, o |
| true false | cierto, falso |

Veamos algunos ejemplos:

```
>>> 8 > 5
>>> True

>>> (4 > 8) or (3 > 2)
>>> True

>>> True and False
>>> False

>>> (4 > 8) and (3 > 2)
>>> False
```

## 1.5 Cadenas de texto

Las cadenas de texto (llamdas *string*) no son mas que texto formado por letras y números de cualquier longitud y son fácilmente manipulables. Cada caracter de una cadena de texto tiene asociado un índice que indica su posición en la cadena, siendo 0 el de la izquierda de todo, 1 el siguiente, etc. hasta el último:

```
>>> frase = "Burocracia, su lechuguita"    # Variable "frase" que contiene una cadena de texto
>>> print(frase[0])          # Primera letra de la cadena
>>> B

>>> print(frase[4])          # Quinta letra, con índice 4
>>> c

>>> len(frase)               # Longitud de la cadena de texto, incluyendo espacios en blanco
>>> 25

>>> print(frase[3:10])       # Imprime de cuarto caracter (índice 3) al decimo (indice 9)
>>> ocracia

>>> print(frase[3:])         # Imprime desde el cuarto caracter hasta el final
>>> ocracia, su lechuguita

>>> print(frase[:6])         # Imprime del inicio a sexto caracter (índice 5)
>>> Burocr
```

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

Recuerda que los índices y en general cualquier lista de números se **empieza siempre con 0**, por lo que el primer

elemento de una lista es `frase[0]` y no `frase[1]`. Al escribir `frase[10]` estamos tomando el elemento 11 no el 10.

Existen varios métodos o funciones específicas para tratar y manipular candenas de texto. Veamos algunos:

```
>>> frase.split()                                  # Separa la cadena por espacios a una lista
>>> ['Burocracia,', 'su', 'lechuguita']

>>> frase_mayusculas = frase.upper()               # Cambia a mayusculas y lo guardo en la variable fr
>>> print(frase_mayusculas)
>>> BUROCRACIA, SU LECHUGUITA

>>> frase_minusculas = frase.lower()               # Cambia a minúculas y lo guardo en la variable fra
>>> print(frase_mayusculas)
>>> burocracia, su lechuguita

>>> frase.replace('lechuguita', 'bocata')          # Reemplaza una cadena de texto por otra
>>> 'Burocracia, su bocata'
```

## 1.6 Impresión de texto y de números

La 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 usando el formateo de números:

```
>>> # Imprimo el resultado con 50 decimales
>>> print("%.50f") % log10(2.**100)
>>> 30.10299956639811824743446777574717998504638671875000

>>> print("El %s de %d es %f.") % ('cubo', 10, 10.**3)
>>> El cubo de 10 es 1000.000000.
```

Aqui 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
```

## 1.7 Estructuras de datos

Los datos se pueden almacenar en variables univaluadas como hemos visto. También pueden almacenarse en variables estructuradas que contienen uno o más datos. Los tipos de datos estructurados que ofrece Python son las **listas**, **tuplas** y **diccionarios** y se definen de la siguiente forma:

### 1.7.1 Listas

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

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

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

>>> datos =  [24, "Juan Carlos", [6.7, 3.6, 5.9]]                       # lista de datos mixto
```

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

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

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

```
>>> alumnos.append('Iballa')                  # Añade "Iballa" al final de la lista
>>> alumnos.insert(3, 'Jairo')                # Añade "Jairo" en la posición 3
```

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

```
>>> alumnos.sort()
```

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

```
>>> alumnos.pop(2)                      # Elimino el elemento número 2
>>> 'Jairo'

>>> alumnos.remove('Maria')             # Elimino el elemento "Maria" (primera ocurrencia)
```

La 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:

```
>>> alumnos[2:6]
>>> ['Luisma', 'Fran', 'Luisa', 'Ruyman']

>>> print(alumnos[0], alumnos[-1])
>>> ('Miguel', '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 podemos hacer esto:

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

Se puede indicar crear una serie de números indicando el inicio, final y el intervalo entre dos consecutivos. Por ejemplo, para crear una lista con números de 100 a 200 a intervalos de 20 haríamos:

```
>>> 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*.

## 1.8 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:

```
>>> # Un tupla de (lista no cambiable) de alumnos
>>> lista_alumnos = ('Miguel', 'Maria', 'Luisma', 'Fran', 'Luisa', 'Ruyman')
```

Si definimos una variable con varios valores separados por comas, Python interpreta esto como una tupla aunque no esté entre paréntesis:

```
>>> # Defino dos variables distintas a y b
>>> a, b = 1, 3
>>> print(a)
>>> 1
>>> print(b)
>>> 3
>>> # Defino una variable con dos valores separados por comas, que se interpreta como una tupla
>>> c = 1, 3
>>> print(c)
>>> (1, 3)
```

En el ejemplo anterior definimos al principio dos variables, `a` y `b`, pero al hacer luego `c = 1, 3` lo que estamos haciendo es crear una tupla con esos dos elementos. 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
```

## 1.9 Diccionarios

Los diccionarios son listas en las que cada elemento se identifica con un nombre, por lo que siempre se usan en parejas clave-valor separado 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', 1.6699999999999999]
```

## 1.10 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`. Se puede importar un paquete haciéndolo implícitamente, osea importando el paquete en sí o bien una, varias o todas sus funciones:

```
>>> 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', 'c
```

Para conocer otros paquetes de la librería estándar consulta el tutorial oficial de Python o guía oficial de la Librería de Python. Más adelante veremos otro paquete numérico de Python más avanzado que nos aporta éstas y muchas otras funciones matemáticas útiles.

CAPÍTULO 2

# Programas ejecutables

## 2.1 Reutilizando el código

Con Python, además de usarse de manera interactiva con una consola, es posible crear programas o *scripts* (guiones) ejecutables. Un *script* es un fichero de texto con los comandos de Python escritos de manera consecutiva y que se ejecutarán al ajecutar el *script*. Basta utilizar un editor de textos cualquiera (kate, gedit, notepad, etc.) con extensión **.py** para que éste lo identifique como un programa de Python.

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 de Linux (o de la de Windows) escribiendo en ella

```
python cubo.py
```

En la primera línea del programa hemos puesto ademas **#!/usr/bin/python** para que además no haga falta llamar al intérprete y se pueda ejecutar como un programa normal, aunque para ello 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.

Hay que fijarse que a menudo hay lineas *comentadas*, que empiezan con **#**. Estas líneas no son ejecutables y **son ignoradas** por el intérprete de Python y se usan para añadir notas o comentarios. Es muy recomendable incluir comentarios de este tipo que expliquen lo que el programa va haciendo porque nos ayudarán más adelante a recordar qué hace o cómo funciona el código cuando volvamos a leerlo nosotros u otros usuarios del código. 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 ", 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 ", 2.0**3)
```

## 2.2 Definiendo funciones

Además de las funciones incluídas en Python, el usuario puede crear las suyas propias, que permiten reutilizar código. Veamos por ejemplo una sencilla función para calcular el cubo de un número y otra para imprimir un mensaje:

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

print( cubo(4.0) )

def saludo(nombre):
   print("Hola%s, cómo estas?"% nombre)

saludo('Juan')
Hola Juan, cómo estas?
```

Nótese que el primer caso hemos usado la sentencia **return** para devolver un valor, sin imprimirlo necesariamente. Esto es lo más habitual cuando se crea un función, hacer todas las operaciones necestarias y luego utilizar **return** para devolver un valor; luego, utilizarse la función, este retultado puede entonces imprimirse, volcarse a una variable, o lo que el programador necesite. En el segundo caso de arriba siempre se imprime una línea cada vez que se llama a la función, porque se pide en la función.

**Advertencia:** En Python la identación es obligatoria, porque se emplea para separar los bloques e indican donde empiezan y terminan los bucles y condicionales. Si la identación no es correcta, se obtendrá un resultado equivocado o tendremos un error de identación.

## 2.3 Entrada de datos

Generalmente el valor de una variable se asigna haciendo por ejemplo pi=3.1416, pero se puede pedir una entrada por teclado usando la función **raw_input()** de la forma siguiente:

```
entrada_numero = raw_input('Dame un numero: ' )
Dame un numero: 22
```

Es muy importante notar que el valor que se obtiene es siempre un *string*, aunque se introduzca un número. Por ejemplo, en el ejemplo anterior el valor de la variable "entrada_numero" es "22", comillas, osea un *string*, como podría comprobarse haciendo **type(entrada_numero)**. Si queremos operar aritméticamente con el resultado debemos pasarlo a entero o a *float*, según cómo lo vayamos a usar, por ejemplo:

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

Con la ayuda de sta función podemos crear programas que pidan uno o varios números (u otro tipo de dato) al usuario y hacer cálculos complejos con ellos sin tener que incluir los valores de entrada en el fichero cada vez que se utilice. Ahora podríamos modificar el pequeño programa del inicio, que calcula el cubo de un número, de manera que lo pida al usuario cuando se ejecute el programa:

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

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

# Recoje un numero dado por el usuario y pasa a float
x = float(raw_input("Valor del numero? "))

# Defino una funcion que calcula el cubo de un numero
def cubo(x):
    return x**3

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

## 2.4 Ejercicios

1. Dada la frase "Dios no solo juega a los dados, a veces los tira donde no se pueden ver"
   - ¿Cuantas 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?
   - Concatena (une) el primer tercio de la fase con el último tercio.
2. Escribe una función que calcule la distancia cartesiana entre dos puntos cualesquiera de coordenadas $(x_1, y_1)$ y $(x_2, y_2)$.
3. Escriba un programa que calcule la densidad media de cualquier planeta, admitiendo como entrada la masa y el radio medio de éste.
4. 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. Una lata de refresco a 5ºC queda en la guantera del coche a 40ºC. ¿Qué temperatura tendrá 1, 5, 12 y 14 horas? Utiliza *k=0.45*. Encuentra las horas que tendría que estar para estar a 0.5ºC menos que la temperatura ambiente. Define funciones adecuadas para realizar ambos cálculos para cualquier tiempo y cualquier hora respectivamente.

# Control de flujo

Un programa consiste en una serie de sentencias que se ejecutan una detrás de otra siguiendo así el flujo natural. No obstante, puede convenirnos cambiar esta secuencia de ejecución y para ello se entroducen algunas **sentencias de control de flujo**. Estas nos permiten interactuar con el programa, tomar decisiones y ejecutar sentencias en un orden diferente al natural. Veamos las más importantes.

## 3.1 El bucle for

Realiza una misma acción o serie de acciones o sentencias repetidas veces. Una manera habitual de usar **for** es para recorrer con una variable **cada uno de los elementos** de una lista:

```
In [3]: for isla in ['La Gomera', 'Maui', 'Cuba', 'Sri Lanka']:
    print("La isla de %s" % isla)
   ...:
   ...:
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 declarada previamente) que va tomando el valor de cada uno de los elementos de la lista; el bucle finaliza cuando la lista se termina. También es posible hacerlo con una variable lista:

```
islas = ['La Gomera', 'Maui', 'Cuba', 'Sri Lanka']

for isla in islas:
   print("La isla de  %s" % isla)

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

Una función muy útil es 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:

```
In [11]: for i in range(2,12,2):
   ....:     print("El cubo de %d es %d" % (i, i**3) )
   ....:
   ....:
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 resultado de antes se puede obtener también iterando los índices de la lista en lugar de lo elementos, creando una lista de los índices usando **range()**:

```
for i in range(len(islas)):
    print("%d- La isla de%s"% (i, islas[i]))

""" Imprime:
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()** hemos creado una lista de tantos números como elementos tiene la lista "islas", desde 0 hasta **len(islas)** (la longitud de la lista) e iterar los índices.

Los bucles **for** nos permiten crear nuevas listas fácilmente:

```
In [23]: b = [2.3, 4.6, 7.5, 10.]

In [24]: from math import log10        # Importo la función ''log10'' del módulo ''math''

In [25]: c = [log10(x) for x in b]

In [26]: print(c)
[0.36172783601759284, 0.66275783168157409, 0.87506126339170009, 1.0]
```

al hacer esto hemos creado una nueva lista que contiene el logaritmo de 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 de **base 3** de 5 haríamos **log(5,3)**.

Una aplicación muy interesante es suma de series de números. Supongamos que queremos calcular el sumatorio $\sum_{n=1}^{10} \frac{1}{n^2}$. Debemos definir una variable muda en la que vamos sumando o acumulando los términos del sumatorio:

```
a = 0.0   # Variable muda, para ir acumulando los términos de la suma
for n in range(1,11):
    term = 1/n**2.0     # Término del sumatorio
    a = term + a        # Acumulamos el término
    print(" %3d  %10.6f  %10.6f" % (i, term, a))

"""RESULTADO
  n   termino_n  valor sumatorio
  ------------------------------
  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 *a* 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 *a* al terminar el bucle.

## 3.2 Sentencias if-then-else

Realizan una o varias operaciones si una determinada condición es cierta. De no cumplirse, se pueden realizar otras operaciones.

```
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 es positivo, luego se usa **elif** para que en caso de no ser cumplirse el **if** anterior, compruebe si es negativa. Si no se cumple ninguna condición anterior 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.

Es muy importante notar nuevamente los **bloques de identación** o espacios que separan los condicionales **if-then-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 solo la sentencia **if**:

```
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 ante 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.

Sin embargo, cuando después de un condicional hay **una única sentencia**, ésta se puede escribir en la misma línea:

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

## 3.3 El bucle while

Se ejecutan una o varias operaciones mientras cierta condición que definimos sea cierta. Por ejemplo:

```
cuentas = 0

while cuentas < 10:
   print(cuentas)
   cuentas = cuentas + 1
```

```
0
1
2
3
4
5
6
7
8
9
```

En este ejemplo se define inicialmente un valor `0` para la variable `cuentas` y su valor se va redefiniendo, aunmentado su valor en `1` e imprimiéndolo. Mientras sea menor que 10 las sentencias dentro del bucle **while** seguirán ejecutándose y se detendrá cuando valga 10, algo que podemos comprobar después del bucle:

```
print(cuentas)
# Imprime 10
```

Veamos otro ejemplo. Supongamos que tenemos unos ahorros en el banco y queremos saber el tiempo que nos llevará tener cierta cantidad gracias a los intereses. Lo que hacemos es crear un bucle **while** en el que añadiremos anualmente los intereses, contando los años. Cuando lleguemos a la cantidad deseada el bucle se detendrá y tendremos los años que nos llevaría:

```
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 anhos ahorrar %d euros." % (anhos, mis_ahorros))

# Resultado
# Me llevará 33 anhos ahorrar 500 euros.
```

Podemos utilizar **while** con la condicion de que **no se cumpla** usando **while not**:

```
x = 0
while not x == 10:
  x = x + 1
  print("x =%d" % x)

 # Resultado
 x = 1
 x = 2
 x = 3
 x = 4
 x = 5
 x = 6
 x = 7
 x = 8
 x = 9
 x = 10
```

Sin embargo, hay que tener cuidado cuando se comparan *floats* entre sí, ya que debido a precisión finita de los ordenadores, es posible que una determinada igualdad nunca se cumpla exactamente. Veamos este ejemplo:

```
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
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
x = 0.50000000000000000
x = 0.59999999999999998
x = 0.69999999999999996
x = 0.79999999999999993
x = 0.89999999999999991
x = 0.99999999999999989     <-- El bucle while debió detenerse aquí, pero no lo hizo
x = 1.09999999999999987
x = 1.19999999999999996
x = 1.30000000000000004
  .
  .
  .
```

y así sucesivamente, no se pararía nunca. El código anterior **produce un bucle infinito** porque la condición **x == 1.0** nunca se da exactamente debido a la precisión limitada de los ordenadores, el valor más cercano es 0.9999999999 pero no 1.0. La conclusión es que es preferible **no comprar nunca *floats* exactamente**. Una opción 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
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
```

```
x = 0.50000000000000000
x = 0.59999999999999998
x = 0.69999999999999996
x = 0.79999999999999993
x = 0.89999999999999991
x = 0.99999999999999989
```

## 3.4 Atrapando los errores

Hemos visto que cuando existe algún error en el código, Python detiene la ejecución y nos devuelve una **excepción o mensaje de error** indicándonos que fué 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 no permiter "probar" (Try) una sentencia y capturar un error y hacer algo al respecto (except) en caso de haberlo en lugar de detener el programa. Para el caso anterior podemos hacer lo siguiente:

```
In [7]: a, b = 23, 0
In [8]: 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 el 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**:

```
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. Consulta la documentación oficial de Python para más información sobre la captura de excepciones y los tipos de errores reconocidos.

## 3.5 Ejercicios

1. Escribe un progama que calcule el factorial de un número *n* cualquiera. Es decir, calcula:

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

2. Crea una lista con lista con 10 números con valores distintos y arbitrarios, con valores de 0 a 100. Crea una función que encuentre el mayor y que dé su posición en la lista.
3. Con 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. Crea un programa que calcule el valor medio y la desviación estándar de una lista cualquiera de números.
5. Genera una lista con lista con 100 números enteros aleatorios de -100 a 100 con la función **randint** (haz `from numpy.random import randint` y luego `nums = randint(-100, 100, 100)`. Separa en tres listas distintas los números negativos, los positivos y los mayores de +50, de manera que la suma de los números de cada lista no sea mayor que 200, es decir, completar las listas mientras que no se llegue a ese número.
6. Calcular el valor de $\pi$ empleando la siguiente expresión:

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

7. Modifica el programa anterior para obtener el valor de $\pi$ con una precisión determinada (por ejemplo $10^{-5}$) comparado con el valor real tomado del módulo math.
8. Haz 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.
9. Dada la lista de notas de los alumnos de una clase, decir quien ha sacado aprobado (más de 5), notable (más de 7), sobresaliente (más de 9) o ha suspendido.

| Alumno | 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 chica terminan con "a" (lo que casualmente en este ejemplo es cierto), decir si cada uno es chico o chica.

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

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

para obtenerla con 5 cifras significativas. La suma hasta $n = \infty$ es 0.5. Como resultado dar el valor de la suma y el número *n* de sumandos sumados.

11. 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 contínua (*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 contínua 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 |

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

12. 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.

# Arrays con Numpy

Los **arrays** son un tipo de dato similar a las listas, pero orientadas especialmente al cálculo numérico. En cierto modo se pueden considerar como vectores o matrices y son un tipo de dato fundamental para el cálculo con tipos de datos de 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; 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>()

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, en lugar de multiplicar cada uno de sus elementos. Peor aún, al multiplicarlo por un número no entero da un error, al no poder crear una fracción de una lista. Esto se podría resolver iterando cada uno de los elementos de la lista con un bucle **for**:

```
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.

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 arrays. Además, numpy incluye funciones matemáticas básicas similares al módulo **math** además de algunas utilidades de números aleatorios, ajuste lineal de funciones, etc.

## 4.1 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(), etc., especialmente cuando trabajamos con varios módulos distintos.

Un *array* se puede crear explícitamente o a partir de una lista.

```
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 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
In [15]: print(lista_ceros)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

In [16]: lista_unos = ones(10)                   # Array de 10 unos
In [17]: print(lista_unos)
[ 1.  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
In [19]: print(otra_lista)
[  0.           4.28571429   8.57142857  12.85714286  17.14285714
  21.42857143  25.71428571  30.        ]
```

## 4.2 Indexado de arrays

Los *arrays* se indexan prácticamente igual que las listas y las cadenas de texto:

```
In [18]: print(numeros[3:8])          # Elementos desde el tercero al septimo
[ 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 elemento, no un array
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 septimo, pero saltando de dos en d
[ 2.  4.  6.]
```

## 4.3 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, )
```

## 4.4 Operaciones con arrays

Los arrays permiten hacer operaciones aritméticas básicas entre ellos:

```
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      ]
```

Los operaciones se hacen elemento a elemento, por lo que ambas deben tener la misma forma (*shape*). Fíjate que en la división el resultado del primer elemento es indefinido (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 importante fijarse que los de *arrays* o listas a unir **debe 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 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 en el array
Out[5]: 9.1999999999999993

In [6]: z.min()   # Valor mínimo en el array
Out[6]: 2.2999999999999998

In [7]: z.mean()  # Valor medio
Out[7]: 6.1857142857142851

In [8]: z.std()   # Desviación típica
Out[8]: 2.1603098795103919

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

In [16]: median(z) # Mediana
Out[16]: 7.0
```

Los *métodos*, que se operan se manera **z.sum()** también pueden usarse como funciones de tipo **sum(z)**, etc. Consulta el manual de **numpy** para conocer otras propiedades y métodos de los *arrays*.

Una gran utilidad es la posibilidad de usar *arrays* con datos booleanos (**True** o **False**) y operar entre ellos o 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 es resultado es otro array booleano con el resultado que corresponda, pero al multiplicar 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 índices numericos con numeros de 0 9 de 2 en 2
In [38]: indices1 = arange(0,10,2)

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

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]
```

## 4.5 Arrays multidimensionales

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

## 4.6 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 descriminante $\Delta = b^2 - 4ac$.

2. La media aritmética, pesada y geométrica de una serie de números se definen respectivamente como

$$\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i, \qquad \bar{x} = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i}, \qquad \bar{x} = \prod_{i=1}^{n} x_i$$

   Calcular estos valores para la lista de números 34.4, 30.1, 29.8, 33.5, 30.9, 31.1 y pesos 0.9, 0.79, 0.84, 0.6, 0.88, 0.78.

3. Cree un programa que calcule la desviación estándar de los números en un array, es decir, calcular:

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2}.$$

4. Usando la función randint de numpy.random, genere una lista de 30 números enteros. Calcule la media y desviación estándar entre grupos de 5 en 5 poniendo los resultados en arrays, es decir, obtener un array con las medias y otro con las desviaciones estándar, que serán de longtud 30/5. Crear un fichero con tres columnas que contenga el número de línea, la media y la desviación estándar.

5. 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 siguente ecuación:

$$M\theta = \arctan 2 \left( \frac{1}{n} \cdot \sum_{j=1}^{n} \sin \theta_j, \frac{1}{n} \cdot \sum_{j=1}^{n} \cos \theta_j \right)$$

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

6. Calcule en un *array* los valores que toma la función seno cociente $sen(\theta)/\theta$ para valores de $\theta$ entre -45º y 45º a intervalos de 0.1º. Escriba el resultado en un fichero que incluya en una columna el ángulo $\theta$ en grados y en otra el valor de seno cociente correspondiente.

7. Para el cálculo de la letra del DNI se calcula el módulo 23 del número, es decir, 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
   ```

   Escriba un programa que calcule letra de cualquier DNI. El programa debe comprobar que la entrada tiene ocho dígitos.

8. Calcule todos los números inferiores a 500 que son multiplos de 5 y 7.

9. 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`.

# Lectura y escritura de ficheros

Muy a menudo tenemos datos de un cálculo o medidas de un experimento en un fichero de texto. Para poder manipular estos datos debemos aprender a leerlos como números o arrays para poder analizarlos. 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 gran cantidad de números de ficheros generados automáticament. Vamos a ver cómo leer y escribir ficheros de texto con Python.

## 5.1 Creando un fichero sencillo

El primer paso para manipular un fichero, ya sea para crearlo, leerlo o escribir en él es crear 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)
'w'  -> Abre el fichero mi_fichero.txt para escribir. Si no
        existe lo crea y si existe sobrescribe el contenido.
'a'  -> Abre el fichero mi_fichero.txt para añadir. Si no
        existe lo crea y si existe continua escribiendo en 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 al fichero se hace a 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 teminar 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 abrir con un editor de textos 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 el código ASCII para 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 arange(100.):
   ....:     fsalida.write(' %d   %10.4f\n' % (i, exp(i)))      # Escribe usando 10 caracteres en tota
In [63]: fsalida.close()                                       # con 4 decimales
```

## 5.2 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" 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 = []
In [71]: y_datos = []
In [73]: for linea in fdatos:
   ....:     x, y = linea.split()            # Se separa cada línea en 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()

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 por espacios, si queremos separar por comas u otro caracter debemos incluirlo como parámetro: **split(',')**.

## 5.3 Lectura y escritura con Numpy

Una manera alternativa de guardar y leer arrays es usando los métodos **savetxt()** y **loadtxt()** de **numpy**, que guardan y leen ficheros de texto con **una línea por columna**.

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

x, y = loadtxt("datos2.txt")
```

Consulta la ayuda de la función leadtxt() de **numpy** para conocer otras opciones de lectura como seleccionar columnas determinadas, usar distintos delimitadores de columnas, etc.

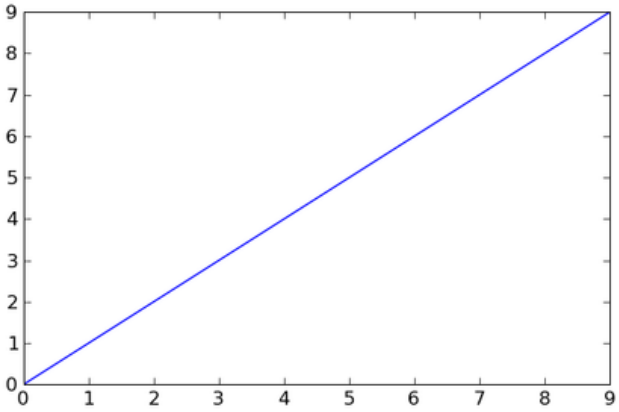
## 5.4 Ejercicios

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.

# Representación gráfica de datos

La representación de 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 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
>>> x = arange(10.)           # array de floats, de 0.0 a 9.0
>>> plot(x)                   # generar el gráfico
>>> [<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 hacemos varios cambios en la gráfica antes de mostrarla, 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 menera no hace falta poner `show()` para mostrar la gráfica:

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

Otra posibilidad es iniciar ipython en modo `pylab`, haciendo `ipython -pylab`, de esta manera se carga automáticamente pylab y se activa el modo interactivo, aparte de importar el módulo `numpy` y todas sus funciones.

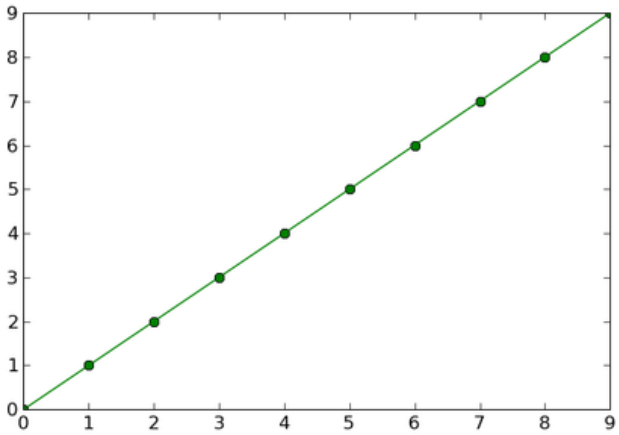
Fíjate cómo el comando `plot()` devuelve una lista de instancias de cada dibujo. En este caso es una lista con un sólo elemento, una instancia `Line2D`. Podemos capturar esta instancia para referirnos a este dibujo más adelante haciendo:

```
>>> 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 `x`, éste se reemplaza por el **número de elemento**, 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 contínua uniendo esos puntos, que es como se dibuja por defecto. Si queremos pintar los puntos debemos hacerlo con un parámetro adicional, por ejemplo:

```
>>> plot(x,'o')      # Pinta diez puntos como "O"
>>> [<matplotlib.lines.Line2D object at 0xa57ffac>]

>>> plot(x,'o-')     # Igual que antes, pero uniéndolos además con una línea contínua
>>> [<matplotlib.lines.Line2D object at 0xa58e80c>]
```




En este caso el **'o'** se usa para dibujar puntos gruesos y si se añade **-** también dibuja la línea contínua. En realidad 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 :func'plot()':

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

El tercer parámetro (o segundo, si no se incluye la x) , donde se indica 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 contínuas. Estas son otras opciones posibles:

**Marcas y líneas**

| Símbolo | Descripción |
|---|---|
| '-' | Línea sólida |
| '–' | 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 |

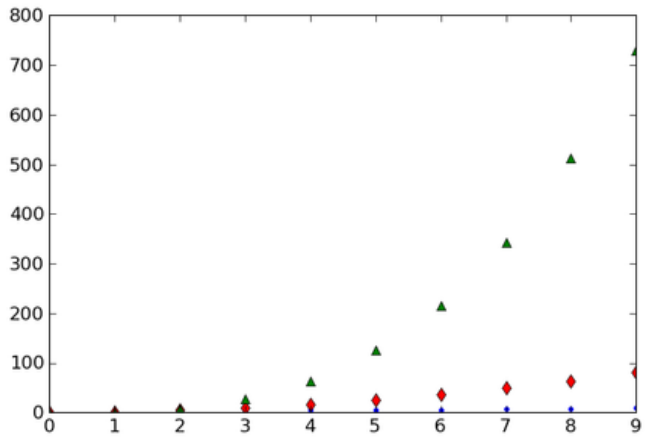
**Colores**

| Símbolo | Color |
|---|---|
| 'b' | Azul |
| 'g' | Verde |
| 'r' | Rojo |
| 'c' | Cian |
| 'm' | Magenta |
| 'y' | Amarillo |
| 'k' | Negro |
| 'w' | Blanco |

Para borrar toda la figura se puede usar la función :func'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:

```
>>> clf()  # Limpio la figura
>>> x2 = x**2
>>> x3 = x**3

>>> plot(x, x,'b.', x, x2, 'rd', x, x3, 'g^')
>>>
[<matplotlib.lines.Line2D object at 0xa94cf6c>,
<matplotlib.lines.Line2D object at 0xa95b72c>,
<matplotlib.lines.Line2D object at 0xa95ba4c>]
```




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

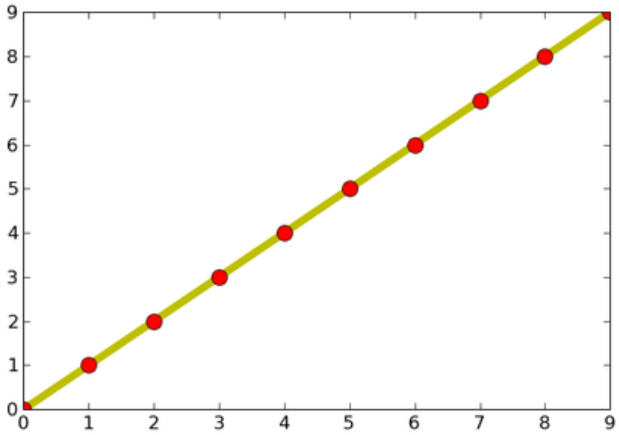
```
>>> xlim(-1,11)    # nuevos límites del eje x
>>> (-1, 11)
>>> ylim(-50,850) # nuevos límites del eje y
>>> (-50, 850)
```

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:

| 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 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’ ‘|’ TICKUP TICKDOWN TICKLEFT TICKRIGHT |

Un ejemplo usando más opciones sería este:

```
>>> plot(x, lw='5', c='y', marker='o', ms=10, mfc='red')
```




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.:

```
>>> p1, p2, p3 = plot(x, x,'b.',    # Hago tres dibujos, campturando sus
           x, x2, 'rd', x, x3, 'g^')   #  instacias en las variables p1, p2 y p3
>>> p1.set_marker('o')              # Cambio el símbolo de la gráfica 1
>>> p3.set_color('y')               # Cambio el color de la gráfica 3
>>> show()                          # Muestro en dibujo por pantalla
```

## 6.1 Trabajando con texto

Exiten funciones para añadir texto a los ejes y a la gráfica en sí, éstos son los más importantes:

```
>>> p1, p2, p3 = plot(x, x, x, x2, x, x3)

>>> xlabel('Eje X')        # Etiqueta del eje X
>>> <matplotlib.text.Text object at 0xad2d4ac>

>>> ylabel('Eje Y')        # Etiqueta del eje Y
>>> <matplotlib.text.Text object at 0xad328cc>

>>> title('Mi grafica')    # Título del gráfico
>>> <matplotlib.text.Text object at 0xad394ac>

>>> text(7, 200, 'Nota')   # Texto en coodenadas (7,200)
>>> <matplotlib.text.Text object at 0xa987e2c>
```

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:

```
>>> x = arange(0, 6*pi,0.1)

>>> y1 = sin(x)/x
>>> y2 = sin(x)*exp(-x)

>>> p1, p2 = plot(x, y1, x, y2)
```

```
>>> texto1 = text(2, 0.6, r'$\frac{\sin(x)}{x}$', fontsize=20)
>>> texto2 = text(13, 0.2, r'$\sin(x) e^{x}$', fontsize=16)

>>> grid()        # Añado una malla al gráfico
```

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()`.

---

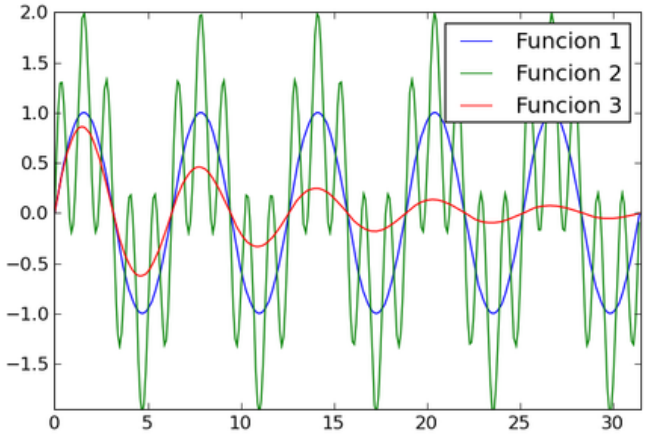
**Nota:** LaTeX es un sistema de escritura orientado a contenidos matemáticos muy popular en ciencia e ingeniería. Puedes ver una introdución a LaTeX en los cursos ISLA de la ULL: http://cisla.osl.ull.es/octubre06/htc/apuntes/latex

---

## 6.2 Representación gráfica de funciones

Visto el ejemplo anterior, vemos que es muy fácil representar gráficamente una función matemática. Para ello, debemos definir la función y luego generar un *array* con el intervalo de valores que se quieren representar. Definamos algunas funciones trigonométricas y luego representémolas 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>
```




En la última línea hemos añadido una leyenda con la función `legend()` que admite como entrada una **tupla** con *strings* correspondiendo consecutivamente a cada uno de los gráficos.

## 6.3 Histogramas

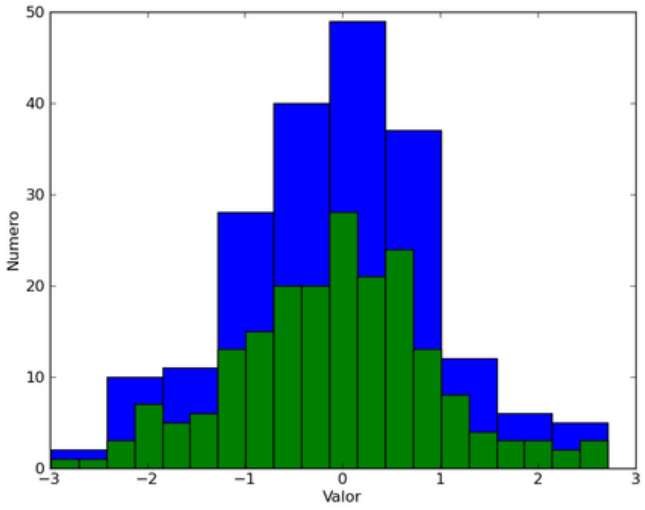
Los histogramas son gráficos que representan el número que veces que se repite ciertos valores dentro de un intervalo, frente a su valor. Podemos hacer histogramas muy fácilmente con la función `hist()` indicando como parámetros un *array* con los números a representar. Si no se indica nada mas, se generará un histograma con 10 divisiones (llamadas *bins*, en inglés). 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 grupos (o *bins*) y cada barra representa cada una de esas divisiones, con el número de valores que caen en cada intervalo. Si en lugar usar sólo 10 divisiones queremos usae digamos 20, debemos indicarlo como un segundo parámetro:

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

El la figura de abajo se muestra el resultado de superponer ambos histogramas. La función `hist()` devuelve una tupla con tres elementos, que son un array con el número elementos en cada división, un array con el punto en eje *X* donde empieza cada división y una lista con referencias a cada una de las barras para modificar sus propiedades (consulta el manual de `matplotlib` para más información).




## 6.4 Figuras múltiples

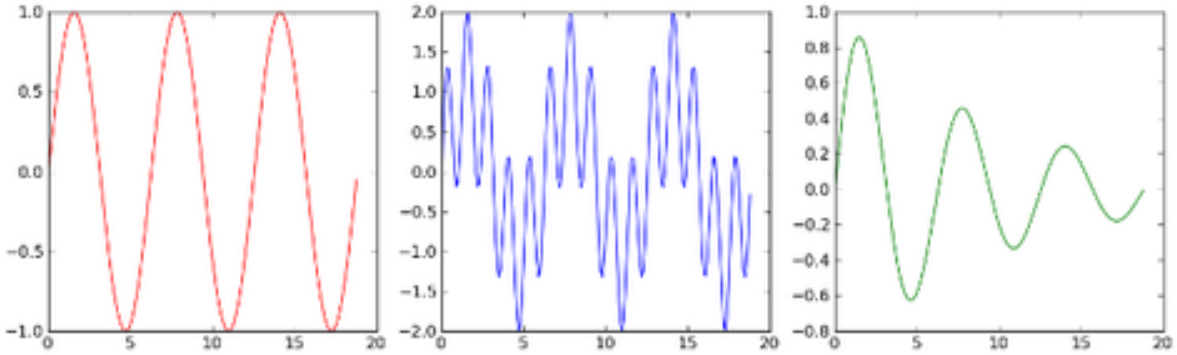
Se pueden hacer cuantas figuras independientes (en ventanas distintas) queramos con la función `figure(n)()` donde *n* es el número de la figura. Cuando se crea una figura al hacer `plot()` se hace automáticamente `figure(1)()`, como aparece en el título de la ventana. Podríamos crear una nueva figura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

```
>>> 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
```

## 6.5 Varios gráficos en una figura

En ocasiones nos interesa mostrar varios gráficos en un misma figura o ventana. Para ello podemos usar la función `subplot()`, indicando entre paréntesis un número con tres dígitos cuyo primer dígito indica en 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 quisiéramos 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:

```
>>> subplot(131)       # Figura con una fila y tres columnas, activo primer subgráfico
>>> p1, = plot(x,f1(x),'r-')

>>> subplot(132)       # Figura con una fila y tres columnas, activo segundo subgráfico
>>> p2, = plot(x,f2(x),'b-')

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



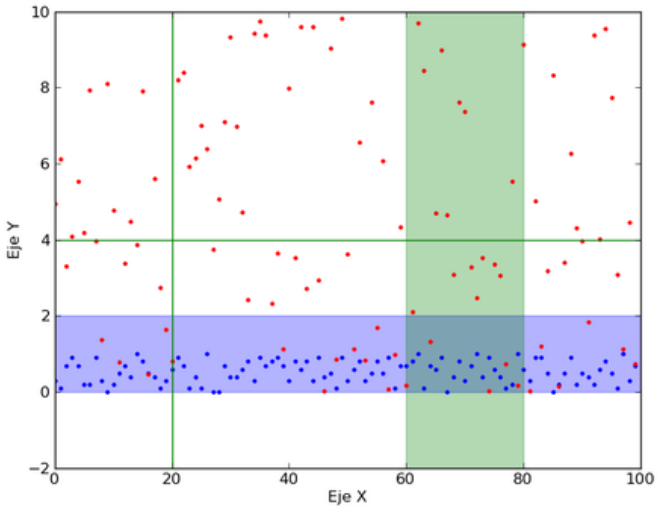
Al igual que con varias figuras, para dibujar en un gráfico hay que activarlo, así, si acabamos de dibujar el segundo gráfico escribiendo antes **subplot(132)** y queremos cambiar algo del primero, debemos activarlo con `subplot(131)()` y en ese momento todas funciones de gráficas se aplicarán a él.

## 6.6 Representando datos de laboratorio

Representar datos leídos de un fichero en lugar de generarlos directamente, es tan fácil como leer los datos y pasar los a *arrays* de numpy. Una vez hecho, se grafican como hemos visto:

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

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

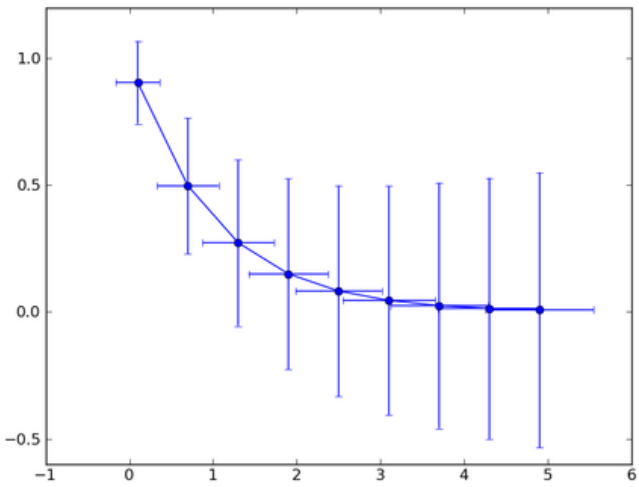
En este caso hemos usado además algunas funciones para crear líneas y bandas horizontales y verticales.




## 6.7 Barras de error

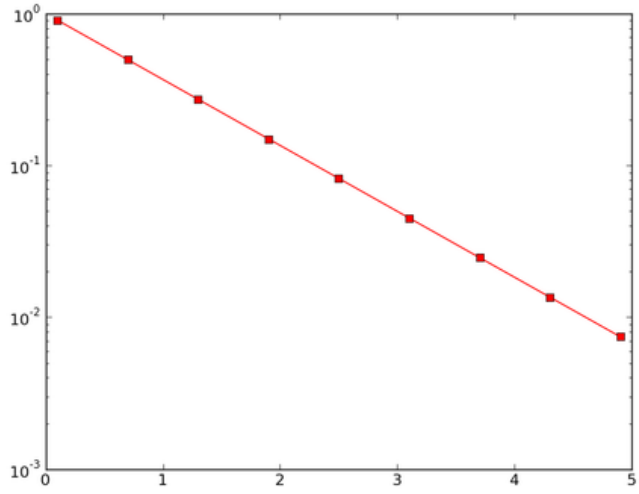
Cuando se trabaja con datos de laboratorio es muy habitual dibujar barras de error en los puntos representados. Esto se puede hacer usando la función `errorbar()` en lugar de `plot()` o junto con ella. Su sintáxis es similar, pero no igual, a la que hay que incluir los errores como parámetros usando *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
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')
```




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, y o 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-')
```

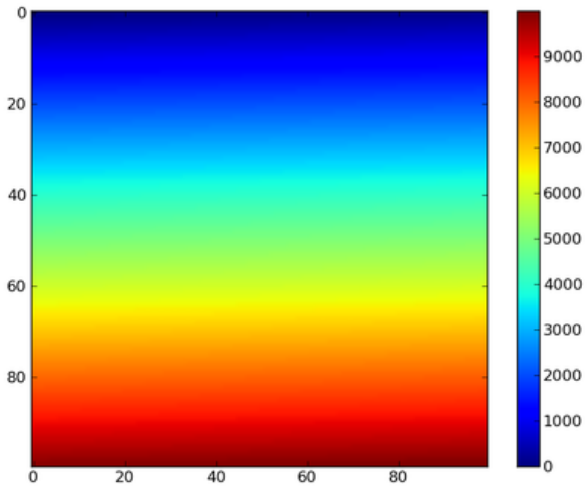



## 6.8 Datos bidimensionales

Es posible representar datos bidimensionales, como podría ser una imagen, usando la función `imshow()`:

```
>>> # Creo un array 2D 100x100 de valores de 0.0 a 99999.0
>>> datos2D = arange(10000.).reshape(100,100)
>>> datos2D.shape
>>> (100, 100)
```

```
>>> # Gráfico del array bidimensional
>>> imshow(datos2D)
>>> # Añado una barra de color para indicar los nieveles
>>> colorbar()
>>> # Cambio a la paleta de colores gray() (por defecto es jet())
>>> gray()
```




## 6.9 Guardando las imágenes

Después de crear una imagen 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 son png, pdf, ps, eps y svg. Por ejemplo:

```
>>> savefig("mi_primera_grafica.eps")   # Guardo la figura en formato eps
>>> savefig("mi_primera_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 Postcript (ps) o Postcript 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.

## 6.10 Ejercicios

1. Representar gráficamente las siguientes funciones:

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

   usando los valores *a=2.0*, $x_0 = 10{,}0$, *c=5,0* y *b=0.5* en el intervalo x=[-50,+50].

2. Con la serie de Gregory-Leibniz para el cálculo $\pi$ usada anteriormente:

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

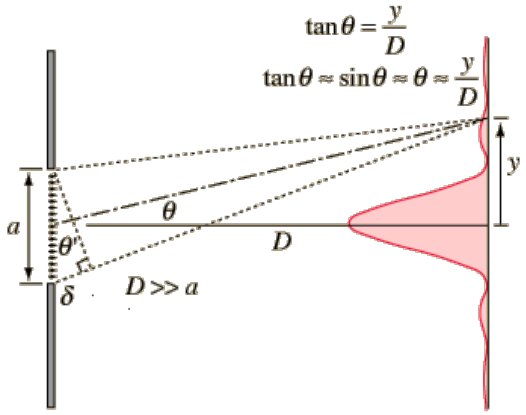
el valor obtenido de $\pi$ se acerca lentamenta al verdadero con cada término. Calcular todos los valores que va tomando $\pi$ con cada término hasta llegar a un error absoluto de $10^{-6}$ y representa en una gráfica ese valor frente número de elementos sumados, usando líneas contínuas, limitándonos a los 300 primeros elementos. El otro gráfico representar el valor de $\pi$ en los últimos 300 elementos. En gráficas dintintas, representar esta vez el valor abosuluto de la diferencia entre el valor calculado y el real frente al número de elementos, reprentando igualmente los 300 primeros en una gráfica y los 300 últimos en otra.

3. El fichero medidas_I131.txt (tema 6 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. Representar gráficamente los datos incluyendo barras de error usando puntos sin líneas, etiquetando los ejes.

4. 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:

$$I(\theta) = I_0(\frac{sen\beta}{\beta})^2 \qquad\qquad \beta = \frac{\pi a sen\theta}{\lambda}$$

donde $a$ es el ancho de la rendija, $\lambda$ la longitud de onda de la luz, $I_0$ la intensidad en el eje y $\theta$ 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. Como ya hemos visto, 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 cuerpo. a) Representa gráficamente la variación de la temperatura con el tiempo, partiendo de una $T_0 = 5°C$ a lo largo de 24 horas suponiendo *k=0.45* y temperatura ambiente de 40ºC. b) Superpon sobre esta curva las curvas correspondientes a cuerpos con *k=0.3* y *k=0.6* con distinto color y trazado identificándolas con una leyenda.

6. Representa nuevamente la curva del apartado a) del ejercicio anterior superponiendo además las curvas correspondientes a temperaturas iniciales distintas, de $T_0 = -5°C$ y $T_0 = 15°C$. Para $T_0 = 5°C$ representa un una gráfica aparte cómo cambian las curvas con temperaturas ambiente de 20ºC y 50ºC, además de la de 40ºC. Identifica cada curva y etiqueta correctamente los ejes en todas las gráficas.

7. 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:

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

   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.

8. El movimiento de 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, $\omega$ la fecuencia de oscilación y *k* el factor de amortiguamiento. Representar gráficamente el movimiento de un oscilador forzado de amplitud inicial de 10cm y frecuencia de 10 ciclos por segundo y $\delta = \pi/8$ con factores de amortiguamiento de 0.1, 0.4, 0.9 y 1.1.

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

# Ajuste de datos experimentales: Método de mínimos cuadrados

Un trabajo habitual de laboratorio es la creación de un modelo matemático de cierto comportamiento físico empleando datos experimentales. Por ejemplo, si se toman medidas de la amplitud de las oscilaciones de un péndulo, es posible obtener una función oscilatoria analítica que describa ese movimiento con la frecuencia y amplitud adecuadas para cualquier instante de tiempo.

Aunque existen muchas técnicas para el ajuste de funciones a datos experimentales, el método más común es el de *mínimos cuadrados*. Supongamos que tenemos una serie de medidas $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)$, siendo *x* la variable independiente e *y* la dependiente. Para un modelo $f(x)$ de estos datos, hay un error *r* para cada medida de $r_1 = y_1 - f(x_1), r_2 = y_2 - f(x_2), \ldots, r_n = y_n - f(x_n)$. Según el método de mínimos cuadrados, la mejor función de ajuste *f(x)* es aquella en la que

$$R = d_1^2 + d_2^2 + \cdots + d_n^2 = \sum_{i=1}^{n} d_i^2 = \sum_{i=1}^{n} |y_i - f(x_i)|$$

es minimo. La forma canónica de este problema es

$$Xa = Y$$

en la que *X* es una matriz *MxN* para *M* medidas experimentales y *N* grados de libertad y el objetivo es minimizar $|Ax - y|$. En el caso del ajuste a un polinomio de grado *N*, el modelo sería de la forma

$$y(x) = a_N x^N + \cdots + a_2 x^2 + a_1 x + a_0$$

para el que habría que escojer la combinación de parámetros $a_i$ que mejor se ajusten a los datos. Así, la matriz *A*, llamada *matriz de Vandermonde* tiene esta forma:

$$\begin{bmatrix} x_1^N & \cdots & x_1^2 & x_1 & 1 \\ x_2^N & \cdots & x_2^2 & x_2 & 1 \\ \vdots & & & & \\ x_M^N & \cdots & x_M^2 & x_M & 1 \end{bmatrix}$$

en ella, cada fila corresponde a una medida experimental $Ax = y_i$.

## 7.1 Ajuste a polinomios con Python

La función **polyfit()** de **Numpy** permite ajuste de datos experimentales a polinomios de cualquier orden. La sintaxis básica es

```
parametros = polyfit(x, y, n)
```

en donde $x$ e $y$ son los datos experimentales y $n$ grado del polinomio a ajustar. El resultado es una lista con los parámetros del polinomio. Si a polyfit se le incluye la opción *full=True*, además de los parámetros devuelve el residuo y otros datos (ver ayuda de la función **polyfit()**).

Consideremos el siguiente ajuste a una recta de una serie de datos $x$ e $y$:

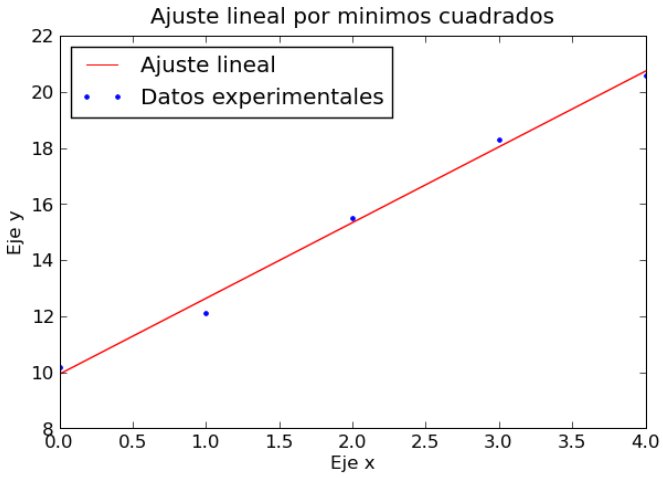
```
# Importo todas las funciones de numpy si no lo he hecho
from numpy import *

# Datos experimentales
x = array([ 0.,  1.,  2.,  3.,  4.])
y = array([ 10.2 ,  12.1,  15.5 ,  18.3,  20.6 ])

# Ajuste a una recta (polinomio de grado 1)
p = polyfit(x, y, 1)

print(p)
# imprime [ 2.7   9.94]
```

en este ejemplo **polyfit()** devuelve la lista de parámetros $p$ de la recta, por lo que el modelo lineal $f(x) = ax + b$ de nuestros datos será:

$$y(x) = p_0 x + p_1 = 2{,}7x + 9{,}94$$

Ahora podemos dibujar los datos experimentales y la recta ajustada:

```
# Valores de y calculados del ajuste
y_ajuste = p[0]*x + p[1]

p_datos, = plot(x, y, 'b.')                                    # Dibujamos los datos experimentales
p_ajuste, = plot(x, y_ajuste, 'r-')    # Dibujamos la recta de ajuste

title('Ajuste lineal por minimos cuadrados')

xlabel('Eje x')
ylabel('Eje y')

legend(('Datos experimentales', 'Ajuste lineal'), loc="upper left")
```




Como se ve en este ejemplo, la salida por defecto de **polyfit()** es un array con los parámetros del ajuste. Sin embargo, si se pide una salida detalla con el parámetro *full=True* (por defecto *full=False*), el resultado es una tupla con el array de parámetros, el residuo, el rango, los valores singulares y la condición relativa. Nos interesa especialmente el residuo del ajuste, que es la suma cuadrática de todos los resíduos $\sum_{i=1}^{n} |y_i - f(x_i)| * *2$. Para el ejemplo anterior tendríamos lo siguiente:

```
# Ajuste a una recta, con salida completa
resultado = polyfit(x, y, 1, full=True)

print(resultado)
""" Imprime tupla
(array([ 2.7 ,  9.94]),                 # Parámetros del ajuste
 array([ 0.472]),                       # Suma de residuos
 2,                                     # Rango de la matriz del sistema
 array([ 2.52697826,  0.69955764]),     # Valores singulares
 1.1102230246251565e-15)                # rcond
 """
```

Si estamos trabajando con polinomios, puede que nos interese usar las funciones **polyval()** o **poly1d()** de numpy. Se utilizan para evaluar y generar funciones polinómicas respectivamente, a partir de una lista u array de parámetros. Por ejemplo, si del ejemplo anterior tenemos un array *p* con los parámetros del ajuste lineal:

```
# Evaluo el polinomio en x=5.4
print polyval(p, 5.4)
# Imprime 24.520000000000003

# Creo una funcion polinomica de parametros p
mi_recta = poly1d(p)

# Ahora mi_recta es una funcion que puedo evaluar

# Evaluo la fucion en x=5.4
print mi_recta(5.4)
# imprime 24.520000000000003
```

Como es de esperar, **polyfit()** sólo puede ajustar polinomios pero a veces nos gustaría ajustar otras funciones. En muchos casos, sin embargo, es posible linealizar la función con un cambio de variable adecuado y ajustar esta última. Por ejemplo la función

$$y = \frac{b}{x + a}$$

se puede linealizar a $y'(x) = a'x + b'$ haciendo el cambio

$$y' = \frac{1}{y} \qquad a = \frac{a'}{b'} \qquad b = \frac{1}{a'}.$$

ahora basta con ajustar la recta $y'(x) = a'x + b'$ y recuperar los parámetros *a* y *b* de la recta original.

En otros casos en los que no es posible tal cosa, se trataría de hacer ajustes a funciones no polinómicas, aunque esto queda fuera del alcance de este curso. Si tienes interés, consulta la documentación del módulo **optimize** de **scipy**. Por ejemplo, puedes importar la función **leastsq()** del módulo **optimize** haciendo **from scipy.optimize import leastsq**) para hacer ajustes por mínimos cuadrados de funciones arbitrarias, sean lineales o no. Puedes consultar el módulo **scipy.optimize** ver todos los métodos de optimización disponibles.

### 7.1.1 Ejercicios

1. Representar gráficamente los siguientes datos y hacer un ajuste por mínimos cuadrados a un polinomio de grado tres de los datos representando la curva resultante.

   ```
   x = 3.1 6.3 9.9 12.6 21.4
   y = 50.1 190.2 499.0 720.8 1130.0
   ```

   Superponer los ajuste que resultarían de ajustes a polinomios de orden 1 y 2.

2. El fichero de texto medidas_PT_H.txt (archivos Tema 7) posee medidas de presión y temperatura para 10 mol de hidrógeno, que se somete a distintas temperaturas a volumen constante. Este experimento se realiza en tres envases con volúmenes distintos. Suponiendo que el gas se comporta idealmente y por tanto que se verifica que PV=nRT, representar los datos y realizar un ajuste lineal P(T) para cada volumen. ¿Cuánto vale la constante de gases ideales según el experimento?

3. El fichero de texto medidas_PV_He.txt (archivos Tema 7) posee medidas de presión y volumen para 0.1 mol de helio, que se comprime sistemáticamente a temperatura constante. Este experimento se realiza a tres temperaturas distintas. Suponiendo que el gas se comporta idealmente y por tanto que se verifica que PV=nRT, representar los datos y realizar un ajuste lineal P(V) para cada temperatura. ¿Cuánto vale la constante de gases ideales según el experimento?

4. Un isótopo radioactivo sigue una ley de desintegración exponencial de la forma $N(t) = N_0 e^{-kt}$, donde $N(t)$ es la cantidad de material que queda en un tiempo *t*, $N_0$ la cantidad original (en *t=0*) y *k* es la tasa de decaimiento del isótopo. La semivida de un isótopo es el tiempo que tarda una muestra de ese elemento en disminuir hasta la mitad, es decir $N(T) = N_0/2$.

   En un laboratorio se mide cada 12 minutos la masa en gramos de cierto elemento radioactivo. Estos datos se encuentran en el fichero medidas_sustancia_radioactiva.txt (archivos Tema 7). Representar gráficamente las medidas con punto y hacer un ajuste por mínimos cuadrados del modelo teórico de desintegración radioactiva para conocer la tasa de decaimiento del isótopo.

5. Para una muestra que contiene 10g de yodo 131 (semivida de 8 días), se hacen diariamente cinco medidas independientes a lo largo de 60 días. Esas medidas están en el fichero medidas_decaimiento_yodo131b.txt (archivos Tema 7), donde cada fila corresponde a cada una de las 5 medidas realizadas diariamente en gramos de yodo que queda. Representar en un gráfico con puntos las cinco medidas con colores distinto para cada una y ajustar a cada una la curva teórica de decaimiento. Imprimir por pantalla los parámetros de cada uno de los cinco ajustes.

6. Cualquier metal de longitud $L_0$ a temperatura inicial $T0$, que es sometido posteriomente a una temperatura $T$ sufre una dilatación o contración dada aproximadamente por $\delta L = \alpha \Delta T)$ donde $\Delta T$ es la diferencia de temperaturas y $\alpha$ el coeficiente de dilatación característico del metal.

   En un laboratorio se mide la dilatación que experimentan cuatro varillas de metal de distinto material de longitud inicial $L_0 = 10cm$ al ir aumentando progresivamente su temperatura en un grado; estos datos se encuentran en el fichero medidas_dilatacion_metales.txt. Representar gráficamente las medidas en una única figura con un

color distinto para cada metal y calcular el factor de dilatación *a* para cada uno ajustando el modelo teórico a los datos.

CAPÍTULO 8

# Cálculo Numérico

Aunque el paquete `Numpy` ofrece ciertas funcionalidades matemáticas además de la manipulación básica de arrays, como el paquete `linalg` para álgebra lineal y `random` para números aleatorios, en `Scipy` encontraremos todas las herramientas matemáticas que podamos necesitar. `Scipy` es una colección de paquetes de algoritmos y herramientas matemáticas para distintas tareas, que también utiliza `Numpy`. `Scipy` posee varios subpaquetes que deben importarse independientemente cuando se vayan a utilizar; éstos son algunos de ellos:

| Subpaquete | Descripción |
|---|---|
| odr | Regresión de distancias ortogonales (ODR) |
| misc | Funciones varias (lectura de imagenes, factorial, etc.) |
| fftpack | Algoritmos para transformada de Fourier discreta |
| io | Entrada y salida de datos |
| stats | Funciones estadisticas |
| lib | Envoltorios (*wrappers*) de Python a librerías externas |
| integrate | Integración numérica |
| ndimage | Imagenes n-dimensionales |
| linalg | Álgebra lineal |
| interpolate | Herramientas de interpolación |
| optimize | Herramientas de optimización |
| signal | Tratamiento de señales |

Para ver la lista completa de subpaquetes consultar la ayuda de scipy como `help(scipy)()` (haciendo antes `import scipy` para tener todos los nombres asociados a `scipy`) y consultar su página web para ver la documentación completa (*Link www.scipy.org*). Una manera práctica de trabajar es importar el espacio de nombres de `scipy`, es decir el nombre de sus paquetes y funciones principales y luego importar el o los paquetes que vaya a usar, como en este ejemplo:

```
>>> from scipy import *          # importa el nombre de los subpaquetes únicamente
>>> import optimize, stats       # importa los paquete optimize y stats
```

## 8.1 Integración numérica

El subpaquete `integrate` ofrece varias herramientas de integración numérica con distintos métodos. Podemos ver todos los disponibles consultando su ayuda:

```
>>> from scipy import *
>>> help(integrate)
```

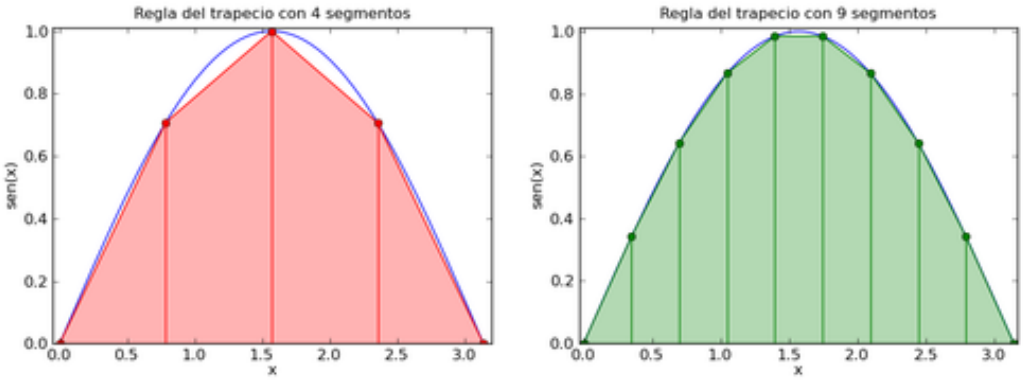
```
Integration routines
====================

Methods for Integrating Functions given function object.

quad          -- General purpose integration.
dblquad       -- General purpose double integration.
tplquad       -- General purpose triple integration.
fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
quadrature    -- Integrate with given tolerance using Gaussian quadrature.
romberg       -- Integrate func using Romberg integration.

Methods for Integrating Functions given fixed samples.

trapz         -- Use trapezoidal rule to compute integral from samples.
cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
simps         -- Use Simpson's rule to compute integral from samples.
romb          -- Use Romberg Integration to compute integral from
                 (2**k + 1) evenly-spaced samples.

See the special module's orthogonal polynomials (special) for Gaussian
    quadrature roots and weights for other weighting factors and regions.

Interface to numerical integrators of ODE systems.

odeint        -- General integration of ordinary differential equations.
ode           -- Integrate ODE using VODE and ZVODE routines.
```

Existen por ejemplo implementaciones del método del trapecio o el método de Simpsom, ambos métodos simples basados en muestras fijas. Las funciones `trapz()` y `simps()`, que emplean dichos métodos para al cálculo numérico de integrales, adminten como entrada un array `y` de valores a integrar y otro array con la variable independiente `x`; si no se incluye un segundo parámetro, el espaciado entre los elementos de `y` de 1 por defecto, aunque este valor se puede asignar con el parámetro opcional `dx=1`.

Supongamos que queremos integrar numéricamente la función $\sin(x)$ de 0 a $pi$, cuyo valor exacto es 2. Veamos cómo se puede calcular numéricamente con distinto número de muestras:

```
>>> from scipy import integrate

>>> #  Generamos 5 muestras de la función seno, de 0 a pi
>>> x = linspace(0, pi, 5)
>>> y = sin(x)

>>> # Integramos el array por el método del trapecio
>>> integrate.trapz(y,x)
>>> 1.8961188979370398

>>> # Si aunmentamos el número de muestras,
>>> # la intragración se acerca más al valor analítico

>>> #  Generamos 20 muestras de la función seno, de 0 a pi
>>> x = linspace(0, pi, 20)
>>> y = sin(x)

>>> # Integramos el array por el método del trapecio
>>> integrate.trapz(y,x)
>>> 1.9954413183201947
```

```
>>> # Integramos por el método de Simpson
>>> integrate.simps(y,x)
>>> 1.999977188106568
```

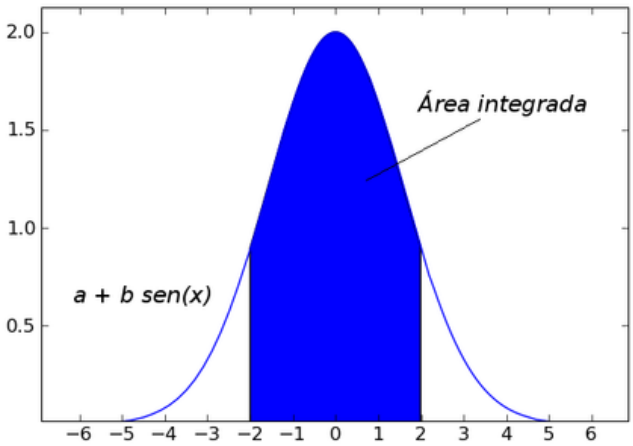



Como se ve, el método de Simpson da un valor más cercano al verdadero, que del trapecio, ya que el primero emplea polinomios de grado 2 para la integración.

En la prática, la función de uso general más eficiente es `quad(func, a, b)()`, que integra por cuadratura de Clenshaw-Curtis [1] una función de Python `func()` entre `a` y `b`. Consideremos como ejemplo el cálculo del área una semicircunferencia de radio unidad calculando la integral bajo la curva. Para ello definimos una función de circunferencia e integramos entre -1 y 1:

La función `quad()` devuelve por defecto el valor de la integral, que en este caso vale $\frac{1}{2}\pi$ y una estimación del error en el proceso de integración numérica. Los límites de integración pueden ser $+\infty$ o $-\infty$ usando los símbolos `+Inf` o `-Inf`:

```
def func1(x):
    return 2.0*exp(-x**2/5.)

int1, err1 = integrate.quad(func1, -2, +2)          # integración entre -2 y +2
int2, err2 = integrate.quad(func1, -Inf, +Inf)      # integración entre -infinito y +infinito

print(int1, int2)
(6.294530963693763, 7.9266545952120211)
```




[1] Para detalles, ver por ejemplo http://en.wikipedia.org/wiki/Clenshaw-Curtis_quadrature

Es posible incluir varios parámetros a la función empleando el parámetro opcional `args` como una tupla de parámetros (ver la ayuda de `quad()`:

```
# Definimos la función a integrar, incluyendo parámetros
def f2(x,a,b):
     return a+ b*sin(x)

# Definimos unos parámetros de entrada
     p=0.2;q=1

# Integramos numéricamente, incluyendo parámetros
integrate.quad(f2, 0, pi, args=(p,q))
(2.6283185307179586, 2.9180197488520396e-14)
```

Se pueden calcular integrales dobles empleando de manera similar `dblquad()`, aunque los límites de la segunda integral se deben poner como funciones de Python.:

```
# Función a integrar
def f1(x,y):
   return x+y

# Límites de la primera integral
a, b = 0, 2

# Definición de limites de segunda integral
def gfun(x):
   return 0

def hfun(x):
   return 10

integrate.dblquad(f1, 0, 2, gfun, hfun)
# Resultado: (120.00000000000001, 1.3322676295501881e-12)
```

## 8.2 Álgebra matricial

Una maneras de ver *arrays* bidimensionales de `Numpy` es como matrices, aunque en realidad los *arrays*, cuando se opera algebraicamente con ellos, no se manipulan como matrices. Por ejemplo el producto de dos arrays bidimensionales **NxM** se raliza elemento a elemento y no como un producto algebraico de matrices. Veamos unos ejemplos:

```
>>> # Dos matrices nxn
>>> A = array([[3, 6, 7], [2, 6, 2], [10, 9, 1]])
>>> B = array([[4, 5, 5], [8, 3, 4], [3, 11, 2]])

>>> print(A)
[[ 3  6  7]
 [ 2  6  2]
 [10  9  1]]

>>> print(B)
[[ 4  5  5]
 [ 8  3  4]
 [ 3 11  2]]

>>> # Producto elemento a elemento entre matrices
>>> print(A*B)
[[12 30 35]
```

```
 [16 18  8]
 [30 99  2]]
```

Sin embargo `Numpy` permite hacer el producto punto entre matrices con la función func:*dot()*:

```
>>> # Producto punto entre matrices
>>> print(dot(A,B))
[[ 81 110  53]
 [ 62  50  38]
 [115  88  88]]
```

Si se va a operar a menudo con matrices es conveniente usar el comando `mat()` de `numpy()`, es una abreviatura de `matrix`. Un elemento `matrix` es idéntico a un array y se crea de igual manera o a partir de *arrays*, pero se comporta como una matriz:

```
>>> # Creación elemento matriz (igual que un array)
>>> C = mat([[4, 5, 5], [8, 3, 4], [3, 11, 2]])
>>> type(C)
>>> <class 'numpy.core.defmatrix.matrix'>
>>> # Conversión de array a matriz
>>> A = mat(A)
>>> type(A)
>>> <class 'numpy.core.defmatrix.matrix'>
>>> # Producto matricial
>>> print(A*C)
[[ 81 110  53]
 [ 62  50  38]
 [115  88  88]]
```

## 8.3 Rutinas básicas con matrices

La inversa de una matriz $\mathbf{A}$ es una matrix $\mathbf{B}$ tal que $\mathbf{AB} = \mathbf{I}$ donde $\mathbf{I}$ es la llamada **matriz identidad** que consiste en una matriz en la que los elementos en la diagonal son unos y son ceros en el resto. Normalmente $\mathbf{B}$ se denota como $\mathbf{B} = \mathbf{A}^{-1}$ . En Scipy, la inversa de una matriz de un array Numpy se puede calcular haciendo `linalg.inv(A)`, o usando `A.I` si A es una matriz. Por ejemplo, consideremos

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

entonces:

$$\mathbf{A^{-1}} = \frac{\mathbf{1}}{\mathbf{25}} \begin{bmatrix} -37 & 9 & 22 \\ 14 & 2 & -9 \\ 4 & -3 & 1 \end{bmatrix} = \begin{bmatrix} -1{,}48 & 0{,}36 & 0{,}88 \\ 0{,}56 & 0{,}08 & -0{,}36 \\ 0{,}16 & -0{,}12 & 0{,}04 \end{bmatrix}.$$

este cálculo lo haríamos con `Scipy` de la siguiente manera:

```
>>> A = mat([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
>>> A
matrix([[1, 3, 5],
                  [2, 5, 1],
                  [2, 3, 8]])
>>> A.I
matrix([[-1.48,  0.36,  0.88],
                  [ 0.56,  0.08, -0.36],
                  [ 0.16, -0.12,  0.04]])
```

```
>>> from scipy import linalg
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
                [ 0.56,  0.08, -0.36],
                [ 0.16, -0.12,  0.04]])
```

## 8.4 Resolución de sistemas de ecuaciones lineales

Con Scipy es muy fácil resolver un sistema de ecuaciones empleando el comando `linalg.solve`. Este comando tiene como parámetros de entrada la matriz y el vector de términos independientes. Si la matriz es simétrica el proceso de cálculo se puede acelerar si se indica como parámetro. Supongamos que queremos resolver el siguiente sistema de ecuaciones:

$$\begin{aligned} x + 3y + 5z &= 10 \\ 2x + 5y + z &= 8 \\ 2x + 3y + 8z &= 3 \end{aligned}$$

Podemos encontrar la solución usando la matriz inversa:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9{,}28 \\ 5{,}16 \\ 0{,}76 \end{bmatrix}.$$

Sin embargo, es mejor usar el comando `linalg.solve` ya que es más rápido y numéricamente más estable, aunque en este caso el resultado es el mismo:

```
>>> A = mat('[1 3 5; 2 5 1; 2 3 8]')    # Las filas se separan con ";"
>>> b = mat('[10;8;3]')
>>> A.I*b                               # Usando la matriz inversa
matrix([[-9.28],
                 [ 5.16],
                 [ 0.76]])
>>> linalg.solve(A,b)                    # Usando la funcion ``linalg.solve(A,b)``
array([[-9.28],
                [ 5.16],
                [ 0.76]])
```

### 8.4.1 Cálculo del determinante

Supongamos que $a_{ij}$ son los elementos de la matriz $\mathbf{A}$ y $M_{ij} = |\mathbf{A}_{ij}|$ será el determinante de la matriz que se obtiene elimiando la *i*-esima fila y la *j*-esima columna de $\mathbf{A}$. Entonces para cualquier fila *i*:

$$|\mathbf{A}| = \sum_j (-1)^{i+j} a_{ij} M_{ij}.$$

Con Scipy el determinante se puede calcular con **linalg.det**. Por ejemplo, el determinante de la matriz $\mathbf{A}$

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

es

$$\begin{aligned} |\mathbf{A}| &= 1 \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3 \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5 \begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1\,(5 \cdot 8 - 3 \cdot 1) - 3\,(2 \cdot 8 - 2 \cdot 1) + 5\,(2 \cdot 3 - 2 \cdot 5) = -25. \end{aligned}$$

Con Scipy se calcula tan fácilmente como:

```
>>> A = mat([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
>>> linalg.det(A)
-25.000000000000004
```

### 8.4.2 Ejercicios

1. Calcular numéricamente las siguientes integrales

$$2\pi \int_0^1 x^3 \sqrt{1+9x^4}\, dx \qquad \int_0^{2\pi} e^{-x} \sin(10x)\, dx$$

2. Calcular numéricamente el área más pequeña comprendida entre un círculo $x^2 + y^2 = 25$ y la recta *x=3*.
3. Escribir un programa en el que dibujen la función $\frac{lnx}{1-x}$ en el intervalo [0,5]. Calcular el área bajo de la figura formada por los ejes OX, OY y esta curva en el intervalo [0,1]. Su resultado exacto es $-\pi^2/6$. Calcular los errores absoluto y relativo con el que se ha obtenido el resultado, dando sólo las cifras significativas.
4. Crear una función que calcule numéricamente la siguiente integral admitiendo parámetros de entrada *m* y *n*:

$$\pi \int_0^1 \frac{x^m - x^n}{\ln x} dx = \ln \frac{m+1}{n+1}$$

5. Resolver el sistema **AX=B** donde:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & -1 & 3 & 5 \\ 0 & 0 & 2 & 5 \\ -2 & -6 & -6 & 1 \end{bmatrix} \quad y \quad \mathbf{B} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

# Cálculo Simbólico

En el capítulo anterior vimos cómo es posible usar técnicas numéricas para resolver problemas matemáticos complejos, como integrales, sistemas de ecuaciones, etc. En Física e ingeniería es prácticamente más habitual resolver ciertos problemas numéricamente debido a la dificultad o incluso imposibilidad de hacerlo analíticamente. Sin embargo, a menudo necesitamos resolver problemas metemáticos de manera analítica necesariamente, en ocasiones pueden ser problemas complejos. Por ejemplo, es posible que para obtener la primitiva cierta integral recorramos a técnicas de substitución, integración por partes, etc. o bien tengremos que consultar un libro de tablas de integrales.

Afortunadamente, los ordenadores también nos pueden ayudar al cálculo analítico y Python posee para ello módulo Sympy. Sympy es una librería de **cálculo simbólico** que permite resolver analíticamente múltiples problemas matemáticos, entre ellos derivadas, integrales, series numéricas o límites. Veamos algunas posibilidades de esta librería.

## 9.1 Introducción a Sympy

Al igual que en Python existen varios tipos de datos numéricos como enteros (int), decimales (float) o booleanos (bool: True, False, etc.), Sympy posee tres tipos de datos propios: **Real**, **Rational** e **Integer**, es decir, números reales, racionales y enteros. Esto quiere decir que Rational(1,2) representa 1/2, Rational(5,2) a 5/2, etc. y no 0.5 o 2.5.

```
>>> from sympy import *
>>> a = Rational(1,2)

>>> a
1/2

>>> a*2
1

>>> Rational(2)**50/Rational(10)**50
1/888178419700125232338905334472656 25
```

También existen algunas constantes especiales, como el número **e** o $\pi$, si embargo éstos se tratan con símbolos y no tienen un valor númerico determinado. Eso quiere decir que no se puede obtener un valor numérico de una operación usando el *pi* de Sympy, como *(1+pi)*, como lo haríamos con el de Numpy, que es numérico:

```
>>> pi**2
pi**2
```

```
>>> pi.evalf()
3.14159265358979

>>> (pi+exp(1)).evalf()
5.85987448204884
```

como se ve, sin embargo, se puede usar el método **evalf()** para evaluar una expresión para tener un valor en punto flotante (*float*). Es posible incluso representar matemáticamente el símbolo infinito, mediante `oo`:

```
>>> oo > 99999
True
>>> oo + 1
oo
```

Para hacer operaciones simbólicas hay que definir explícitamente los símbolos que vamos a usar, que serán en general las variables y otros elementos de nuestras ecuaciones:

```
>>> x = Symbol('x')
>>> y = Symbol('y')
```

Y ahora ya podemos manipularlos como queramos:

```
>>> x+y+x-y
2*x

>>> (x+y)**2
(x + y)**2

>>> ((x+y)**2).expand()
2*x*y + x**2 + y**2
```

Es posible hacer una substitución de variables usando `subs(viejo, nuevo)`:

```
>>> ((x+y)**2).subs(x, 1)
(1 + y)**2

>>> ((x+y)**2).subs(x, y)
4*y**2
```

## 9.2 Operaciones algebraicas

Podemos usar `apart(expr, x)` para hacer una descomposición parcial de fracciones:

```
>>>  1/( (x+2)*(x+1) )
       1
---------------
(2 + x)*(1 + x)

>>>  apart(1/( (x+2)*(x+1) ), x)
  1          1
------- - --------
1 + x      2 + x

>>>  (x+1)/(x-1)
-(1 + x)
---------
 1 - x
```

```
>>>  apart((x+1)/(x-1), x)
      2
1 - ------
    1 - x
```

Para unirlas, podemos usar `together(expr, x)`:

```
>>>  together(1/x + 1/y + 1/z)
x*y + x*z + y*z
---------------
     x*y*z

>>> together(apart((x+1)/(x-1), x), x)
-1 - x
-------
 1 - x

>>> together(apart(1/( (x+2)*(x+1) ), x), x)
       1
---------------
(2 + x)*(1 + x)
```

## 9.3 Cálculo de límites

Sympy puede calcular límites usando la función **limit()** con la siguientesintaxis: **limit(función, variable, punto)**, lo que calcularía el limite de f(x) cuando variable -> punto::

```
>>> x = Symbol("x")
>>> limit(sin(x)/x, x, 0)
1
```

es posible incluso usar límites infinitos:

```
>>> limit(x, x, oo)
oo

>>> limit(1/x, x, oo)
0

>>> limit(x**x, x, 0)
1
```

## 9.4 Cálculo de derivadas

La función de Sympy para calcular la derivada de cualquier función es `diff(func, var)`. Veamos algunos ejemplos:

```
>>> x = Symbol('x')
>>> diff(sin(x), x)
cos(x)
>>> diff(sin(2*x), x)
2*cos(2*x)
```

```
>>> diff(tan(x), x)
1 + tan(x)**2
```

Puedes comprobar que es correcto calculando el límite:

```
>>> limit((tan(x+y)-tan(x))/y, y, 0)
1 + tan(x)**2
```

También se pueden calcular derivadas de orden superior indicando el orden de la derivada como un tercer parámetro opcional de la función **diff**():

```
>>> diff(sin(2*x), x, 1)          # Derivada de orden 1
2*cos(2*x)

>>> diff(sin(2*x), x, 2)          # Derivada de orden 2
-4*sin(2*x)

>>> diff(sin(2*x), x, 3)          # Derivada de orden 3
-8*cos(2*x)
```

## 9.5 Expansión de series

Para la expansión de series se aplica el método `.series(var, punto, orden)` a la serie que se desea expandir:

```
>>> cos(x).series(x, 0, 10)
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)
>>> (1/cos(x)).series(x, 0, 10)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 + O(x**10)


e = 1/(x + y)
s = e.series(x, 0, 5)

pprint(s)
```

La función **pprint** de Sympy imprime el resultado de manera más legible:

```
     4    3    2
1   x    x    x    x
- + -- - -- + -- - -- + O(x**5)
y    5    4    3    2
    y    y    y    y
```

## 9.6 Integración

La integración definida e indefinida de funciones es una de las funcionalidades más interesantes de Sympy. Veamos algunos ejemplos:

```
>>> integrate(6*x**5, x)
x**6
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(log(x), x)
-x + x*log(x)
```

```
>>> integrate(2*x + sinh(x), x)
cosh(x) + x**2
```

También con funciones especiales:

```
>>> integrate(exp(-x**2)*erf(x), x)
pi**(1/2)*erf(x)**2/4
```

También es posible calcular integrales definidas:

```
>>> integrate(x**3, (x, -1, 1))
0
>>> integrate(sin(x), (x, 0, pi/2))
1
>>> integrate(cos(x), (x, -pi/2, pi/2))
2
```

Y también integrales impropias:

```
>>> integrate(exp(-x), (x, 0, oo))
1
>>> integrate(log(x), (x, 0, 1))
-1
```

Algunas integrales definidas complejas es necesario definirlas como objeto **Integral()** y luego evaluarlas con el método **evalf()**:

```
>>> integ = Integral(sin(x)**2/x**2, (x, 0, oo))
>>> integ.evalf()
>>> 1.6
```

## 9.7 Ecuaciones algebraicas y álgebra lineal

Otra sorprendente utilidad de Sympy es su capacidad para resolver sistemas de ecuacuaciones fácilmente:

```
>>> # Una ecuación, resolver x
>>> solve(x**4 - 1, x)
[I, 1, -1, -I]

# Sistema de dos ecuaciones. Resuelve x e y
>>>  solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
{y: 1, x: -3}
```

Sympy tiene su propio tipo de dato Matriz, independiente del de Numpy/Scipy. Con él se pueden definir matrices numéricas o simbólicas y operar con ellas:

```
>>> # Matriz identidad 2x2
>>> Matrix([[1,0], [0,1]])
[1, 0]
[0, 1]

>>> x = Symbol('x')
>>> y = Symbol('y')
>>> A = Matrix([[1,x], [y,1]])
>>> A
[1, x]
[y, 1]
```

```
>>> A**2
[1 + x*y,     2*x]
[    2*y, 1 + x*y]
```

Existen otros constructores similares a arrays de Numpy pero para matrices:

```
>>> # Matrices de unos
>>> ones((5,5))
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]

>>> # Matriz identidad
>>> eye(3)
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]
```

Hay que fijarse en que muchas de las funciones anteriores ya existen en Numpy con el mismo nombre (ones(), eye(), etc.), por lo que si queremos usar ambas debemos importar los paquetes con otro nombre, Por ejemplo:

```
>>> import numpy as np
>>> from sympy import *

>>> # Funcion eye de Sympy (matriz)
>>> eye(3)
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]

>>> # Funcion eye de Numpy (array)
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> # Matriz de Numpy
>>> np.matrix(np.eye(3))
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

Es posible operar entre ellas, salvo que las matrices de Numpy no pueden operar con símbolos, algo que se puede hacer con Sympy. La selección de elementos de matrices de Sympy se hace de manera idéntica a los arrays o matrices de Numpy:

```
>>> # Multiplico la matriz identidad por x
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x, 0, 0]
[0, x, 0]
[0, 0, x]

>>> # Substituyo x por 4 en la matriz
>>> M.subs(x, 4)
[4, 0, 0]
```

```
[0, 4, 0]
[0, 0, 4]

>>> # Substituyo la variable x por y en la matriz
>>> y = Symbol('y')
>>> M.subs(x, y)
[y, 0, 0]
[0, y, 0]
[0, 0, y]

>>>  def f(x): return 1.5*x**2
....:

>>>  eye(3).applyfunc(f)
[1.5,   0,   0]
[  0, 1.5,   0]
[  0,   0, 1.5]


>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))

>>> # Determinante de la matriz
>>> M.det()
-28
# Matriz inversa
>>>  M.inv()
[-3/14, 1/14,  1/2]
[-1/28, 5/28, -1/4]
[  3/7, -1/7,    0]

>>> # Substituyo algunos elementos de la matriz
>>> M[1,2] = x
>>> M[2,1] = 2*x
>>> M[1,1] = sqrt(x)
>>> M
[1,        2, 3]
[3, x**(1/2), x]
[2,      2*x, 1]
```

Podemos resolver un sistema de ecuaciones por el método LU:

```
>>> # Matriz 3x3
>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> # Matriz 1x3
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> # Resuelvo el sistema por LU
>>> soln = A.LUsolve(b)
>>> soln
[3]
[7]
[5]
```

## 9.8 Ejercicios

1. Calcular la derivada de $x^3 \arccos\left(\frac{2}{x}\right)$.

2. Calcular la derivada segunda de $e^{-x}\log(x)$.
3. Calcular los diez primeros elementos de la serie:

$$\sum_{n=0} \frac{1}{(1+n)^{5/2}}.$$

4. Calcular la siguiente integral:

$$y(x) = \int \frac{\cos(x)}{\sin(x)} dx.$$

   Evaluar el resultado para x=0.5.

5. Calcular la integral siguiente y evaluar su valor en x=1.5:

$$y(x) = \int \frac{1}{1+e^x} dx$$

6. Comprobar las siguientes integrales definidas:

$$\int_0^1 \frac{1}{\sqrt{\log\left(\frac{1}{x}\right)}} dx = \sqrt{\pi} \qquad \int_0^1 x\log(1+x)\, dx = \frac{1}{4} \qquad \int_0^\infty e^{-x}\sin(x)\, dx = \frac{1}{2}$$