

# Índice general

---

<b>1</b>	<b>Introducción a Python</b>	3
1.1	Instalación de Python	4
1.2	Manejo básico de Python	4
1.3	IPython Notebook	8
<b>2</b>	<b>El lenguaje Python</b>	11
2.1	Aspectos básicos del lenguaje	11
2.2	Módulos	22
2.3	Control de flujo	26
2.4	Funciones definidas por el usuario	29
2.5	Ejercicios	33
<b>3</b>	<b>Aspectos avanzados</b>	35
3.1	Algo más sobre funciones	35
3.2	Entrada y salida de datos	41
3.3	Más sobre estructuras de datos	47
3.4	Excepciones	51
3.5	Ejercicios	52
<b>4</b>	<b>NumPy</b>	55
4.1	<i>Arrays</i>	55
4.2	Funciones para crear y modificar <i>arrays</i>	58

## 2 ÍNDICE GENERAL

4.3	<i>Slicing</i> .....	62
4.4	Operaciones .....	64
4.5	<i>Broadcasting</i> .....	70
4.6	Otros métodos de interés .....	75
4.7	Indexación sofisticada .....	78
4.8	Operaciones con <i>slicing</i> .....	81
4.9	Lectura de ficheros .....	82
4.10	Búsqueda de información .....	82
4.11	Ejercicios .....	85

<b>5</b>	<b>SciPy</b> .....	91
5.1	Optimización sin restricciones .....	91
5.2	Optimización con restricciones .....	97
5.3	Interpolación de datos .....	99
5.4	Resolución de ecuaciones diferenciales .....	101
5.5	Ejercicios .....	102

<b>6</b>	<b>Gráficos con Matplotlib</b> .....	105
6.1	Gráficos interactivos .....	106
6.2	Añadiendo opciones .....	112
6.3	Configurando varios elementos del gráfico .....	114
6.4	Gráficos y objetos .....	118
6.5	Gráficos 3D .....	121
6.6	Ejercicios .....	125

<b>7</b>	<b>Programación Orientada a Objetos</b> .....	129
7.1	Definiendo clases .....	130
7.2	Controlando entradas y salidas .....	138
7.3	Ejercicios .....	142

# 1

## Introducción a Python

---

Python es un lenguaje de programación creado por Guido Van Rossum a finales de los ochenta. Su nombre deriva de la afición de su creador al grupo de humor inglés *Monty Python*. Se trata de un lenguaje de alto nivel, interpretado, interactivo y de propósito general cuyo diseño hace especial hincapié en una sintaxis limpia y una buena legibilidad.

Los lectores con conocimiento de algún lenguaje de programación encontrarán en Python un lenguaje sencillo, versátil y que proporciona código fácilmente legible. Para aquéllos que no están familiarizados con la programación, Python supone un primer contacto agradable pues los programas pueden ser comprobados y depurados con facilidad, permitiendo al usuario concentrarse más en el problema a resolver que en los aspectos concretos de la programación.

Aproximadamente a partir de 2005, la inclusión de algunas extensiones especialmente diseñadas para el cálculo numérico han permitido hacer de Python un lenguaje muy adecuado para la computación científica, disponiendo hoy en día de una colección de recursos equivalente a la que podemos encontrar en un entorno bien conocido como MATLAB, y que continua en permanente crecimiento.

Python es software de código abierto que está disponible en múltiples plataformas (GNU/Linux, Unix, Windows, Mac OS, etc.). Se encuentra en la actualidad con dos versiones en funcionamiento. La mayor parte del código que se encuentra escrito en Python sigue las especificaciones de la versión 2, aunque hace ya algún tiempo que la versión 3 se encuentra disponible. En estas notas se usará la versión 2 del lenguaje.

**1 1****INSTALACIÓN DE PYTHON**

Python viene instalado por defecto en los sistemas Linux y OSX, y se puede instalar de forma sencilla en los sistemas Windows desde la página oficial [www.python.org](http://www.python.org). Sin embargo, diversos módulos de interés, y entre ellos, los dedicados a la programación científica que veremos en los Capítulos 4, 5 y 6, requieren de instalaciones separadas. Existen diversas posibilidades para realizar la instalación de otros módulos, pero nosotros vamos a optar por una solución simple y eficiente, que consiste en la instalación de la distribución de Python ANACONDA.

ANACONDA Python es una distribución que contiene el núcleo básico de Python y un conjunto de módulos entre los que se encuentran todos los que vamos a emplear en este texto. Además incluye la consola IPython y el entorno IPython Notebook que veremos en las siguientes secciones, entre otras herramientas de interés. La descarga de esta distribución se puede realizar desde la página de la empresa que la desarrolla <https://www.continuum.io/downloads>

Allí encontraremos descargas para los sistemas Linux, Windows y Mac en versiones para 32 y 64 bits, y en la versión 2.7 o 3.5 de Python. Como hemos comentado antes, aquí usaremos la versión 2 de Python, por lo que habría que descargar el instalador para la versión 2.7. En la misma página de descarga tenemos instrucciones directas para su instalación, que son bastante simples.

Durante la instalación en los sistemas Windows se pregunta si queremos que el intérprete Python que instala ANACONDA sea el intérprete por defecto en el sistema, ya que ANACONDA convive bien con otras versiones de Python en el mismo sistema, y si queremos añadir ANACONDA a la variable PATH. Responderemos afirmativamente a ambas cuestiones. De igual modo, en los sistemas Linux se nos pedirá que ajustemos la variable PATH del sistema para que esté accesible el entorno ANACONDA.

Una vez instalado, podemos ejecutar el programa **anaconda-navigator** que aparece en la lista de programas (en Windows) o desde la consola en Linux, que nos permitirá ejecutar alguno de los programas que comentaremos en la siguiente sección.

**1 2****MANEJO BÁSICO DE PYTHON**

En esta sección veremos algunos aspectos generales relacionados con el uso del intérprete y la creación de *scripts*, para, en la siguiente sección describir el entorno IPython Notebook (ahora denominado Jupyter Notebook) que recomendamos vivamente para trabajar con Python.

Inicialmente en Python podemos trabajar de dos formas distintas: a través de la consola o mediante la ejecución de *scripts* o *guiones* de órdenes. El primer

método es bastante útil cuando queremos realizar operaciones inmediatas y podemos compararlo con el uso de una calculadora avanzada. El uso de *scripts* de órdenes corresponde a la escritura de código Python que es posteriormente ejecutado a través del intérprete.

Para iniciar una consola Python bastará escribir la orden `python` en una terminal,<sup>1</sup> obteniéndose algo por el estilo:

```
Python 2.7.9 (default, Mar 1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

que nos informa de la versión que tenemos instalada y nos señala el *prompt* `>>>` del sistema, el cual indica la situación del terminal a la espera de órdenes. Podemos salir con la orden `exit()` o pulsando las teclas `ctrl` + `D` (`ctrl` + `Z` en Windows)

Una vez dentro del intérprete podemos ejecutar órdenes del sistema, por ejemplo

```
>>> print "Hola Mundo"
Hola Mundo
>>>
```

Obviamente la orden `print` imprime la *cadena de texto* o *string* `Hola Mundo` que va encerrada entre comillas para indicar precisamente que se trata de un *string*. Una vez ejecutada la orden el intérprete vuelve a mostrar el *prompt*.

La otra alternativa a la ejecución de órdenes con Python es la creación de un *script*. Se trata de un archivo de texto en el que listamos las órdenes Python que pretendemos ejecutar. Para la edición del archivo nos vale cualquier editor de texto sin formato. Escribiendo el comando

```
print "Hola Mundo"
```

en un archivo,<sup>2</sup> lo salvamos con un nombre cualquiera, por ejemplo `hola.py`, en el que la extensión ha de ser `.py`.<sup>3</sup> Podemos ejecutar el código sencillamente escribiendo en una consola la orden `python hola.py` (obviamente situándonos correctamente en el *path* o ruta donde se encuentre el archivo). También es posible hacer ejecutable el código Python escribiendo en la primera línea del archivo<sup>4</sup>

<sup>1</sup>En lo que sigue, usaremos un sistema Linux, pero es sencillo adaptarse a otros sistemas. Por ejemplo en Windows, podemos abrir una terminal con el programa **Anaconda Prompt** instalado con la distribución ANACONDA.

<sup>2</sup>Para diferenciar la escritura de órdenes en el intérprete de los comandos que introduciremos en los archivos los ilustraremos con fondos de diferente color.

<sup>3</sup>Atención, en los sistemas Windows la extensión suele estar oculta, por lo que si creamos el fichero con **Notepad**, por ejemplo, a la hora de guardarlo deberíamos seleccionar **All Files** en el tipo de archivo. En caso contrario se guardará con extensión `.txt`.

<sup>4</sup>Esto es lo que se conoce como el *shebang*, y es el método estándar para poder ejecutar un programa interpretado como si fuera un binario. Windows no tiene soporte para el *shebang*.

```
#!/usr/bin/env python
```

y dando permisos de ejecución al archivo con la orden `chmod a+x hola.py` desde una consola. En tal caso podemos ejecutarlo escribiendo `./hola.py` en una consola.

Se pueden utilizar codificaciones diferentes de la ASCII<sup>5</sup> en los *scripts* de Python añadiendo justo detrás del *shebang* la línea

```
# -*- coding: codificación -*-
```

donde *codificación* se refiere al código de caracteres que empleemos (típicamente `utf-8`). El empleo de caracteres no ASCII en un *script* sin esta línea produce errores.

Cuando queremos escribir una orden de longitud mayor a una línea debemos usar el carácter de escape `\`, tanto en el intérprete como en los *scripts*:

```
>>> print "esto es una orden \
... de más de una línea"
esto es una orden de más de una línea
>>> 15 - 23 + 38 \
... -20 + 10
20
```

Aunque en el caso de operaciones aritméticas, la apertura de un paréntesis hace que no resulte obligatorio el carácter de escape:

```
>>> (24 + 25
... - 34)
15
```


## 1 2 1 Entornos de Desarrollo Integrados

Los denominados IDE (*Integrated Development Environment*) son programas que facilitan el desarrollo de código incluyendo típicamente un editor de código fuente acompañado de una consola o herramientas de compilación automáticas, y en ocasiones algún complemento de depuración, o listado de variables presentes, etc. En el caso de Python, existen diversos entornos de este tipo entre los que podemos citar *IDLE*, *Stani's Python Editor*, *Eric IDE*, *NinJa IDE*, *Spyder*, entre otros. Este último viene instalado con la distribución ANACONDA y se puede ejecutar desde **Anaconda Navigator**. Son herramientas interesantes para escribir código de forma más cómoda que el uso aislado de un editor de texto.

## 1 2 2 La consola IPython

En lugar del intérprete Python habitual existe una consola interactiva denominada IPython con una serie de características muy interesantes que

<sup>5</sup>Es decir, codificaciones que admiten caracteres acentuados.

facilitan el trabajo con el intérprete. Entre ellas podemos destacar la presencia de *autocompletado*, característica que se activa al pulsar la tecla de tabulación y que nos permite que al teclear las primeras letras de una orden aparezcan todas las órdenes disponibles que comienzan de esa forma. También existe un operador `?` que puesto al final de una orden nos muestra una breve ayuda acerca de dicha orden, así como acceso al historial de entradas recientes con la tecla .

De forma idéntica a la apertura de una consola Python, escribiendo `ipython` en un terminal obtenemos:<sup>6</sup>

```
Python 2.7.9 (default, Mar 1 2015, 12:57:24)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object', use 'object??' for extra
                  details.

In [1]:
```

Obsérvese que ahora el *prompt* cambia, y en lugar de `>>>` aparece `In [1]:`. Cada vez que realizamos una entrada el número va aumentando:

```
In [1]: 23*2
Out[1]: 46
```

```
In [2]:
```

Si como ocurre en este caso, nuestra entrada produce una salida `Out[1]: 46`, podemos usar la numeración asignada para reutilizar el dato mediante la variable `_1`,

```
In [2]: _1 + 15
Out[2]: 61
```

que hace referencia al valor almacenado en la salida `[1]`. O también

```
In [3]: _ * 2 # _ hace referencia al último valor
Out[3]: 122
```

```
In [4]: _2 + _
Out[4]: 183
```

Además, esta consola pone a nuestra disposición comandos del entorno (`cd`, `ls`, etc.) que nos permiten movernos por el árbol de directorios desde dentro de la consola, y comandos especiales, conocidos como *funciones mágicas* que proveen de funcionalidades especiales a la consola. Estos comandos comienzan

<sup>6</sup>Desde *Anaconda Navigator* disponemos de esta terminal a través de *Jupyter QtConsole*.

por el carácter `%` aunque si no interfieren con otros nombres dentro del sistema se puede prescindir de este carácter e invocar sólo el nombre del comando. Entre los más útiles está el comando `run` con el que podemos ejecutar desde la consola un *script* de órdenes. Por ejemplo, para ejecutar el creado anteriormente:

```
In [5]: run hola.py
Hola Mundo
```

El lector puede probar a escribir `%` y pulsar el tabulador para ver un listado de las funciones mágicas disponibles.

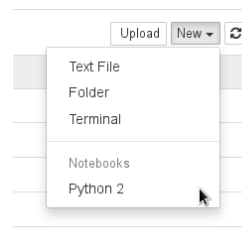
## 1 3

### IPYTHON NOTEBOOK

El IPython Notebook es una variación de la consola IPython, que usa un navegador web como interfaz y que constituye un entorno de computación que mezcla la edición de texto con el uso de una consola. El proyecto ha evolucionado hacia el entorno Jupyter, que soporta otros lenguajes, además de Python. Es una forma muy interesante de trabajar con Python pues aún las buenas características de la consola IPython, con la posibilidad de ir editando las entradas las veces que sean necesarias. Además, permiten añadir texto en diferentes formatos (LaTeX inclusive) e incluso imágenes, por lo que se pueden diseñar páginas interactivas con código e información.

Puesto que este es el entorno que preferimos para trabajar, describiremos con un poco de detalle su funcionamiento general. Para correr el entorno hemos de escribir en una terminal la orden `jupyter notebook`, lo que nos abrirá una ventana en un navegador web, con título *Home*, con un listado de los *notebooks* disponibles y la posibilidad de navegar en un árbol de directorios, así como de editar ficheros desde el navegador. Los *notebooks* son ficheros con extensión `.ipynb` que pueden ser importados o exportados con facilidad desde el propio entorno web.

Si no disponemos de un *notebook* previo, podemos crear uno nuevo pulsando sobre el desplegable **New** (arriba a la derecha), eligiendo el tipo deseado, en nuestro caso **Python 2** (véase la figura adjunta). Esto abre automáticamente una nueva ventana del navegador a la vez que crea un nuevo fichero *Untitled* con extensión `.ipynb` en la carpeta donde hemos iniciado el entorno. La nueva ventana del navegador nos muestra el *notebook* creado, en la que podemos cambiar el título fácilmente sin más que clicar sobre el mismo.



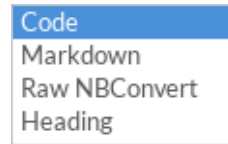
Nuevo Notebook

El concepto básico del entorno Jupyter son las *celdas*, que son cuadros donde insertar texto que puede admitir diferentes formatos de entrada que



pueden seleccionarse a través del menú desplegable del centro de la barra de herramientas (véase la figura adjunta).



Básicamente nos interesan las celdas tipo **Code**, que contendrán código en lenguaje Python, y que aparecerán numeradas como en la consola IPython, y las de tipo **Markdown**, en las que podemos escribir texto marcado por este tipo de lenguaje,<sup>7</sup> o incluso texto en formato L<sup>A</sup>T<sub>E</sub>X, que nos permite añadir información contextual al código que estemos escribiendo.



Tipos de celdas

Por ejemplo, si en una celda estilo **Markdown** escribimos lo siguiente:

```
# Cabecera
```

y a continuación pulsamos  + , que supone la ejecución del contenido de la celda, obtendremos:

## Cabecera

que es el resultado de interpretar el símbolo **#** antes de una palabra, que supone darle formato de título de primer nivel. Si la entrada hubiera sido:

```
### Cabecera
```

entonces la salida es:


## Cabecera


es decir, el símbolo **###** se refiere a una cabecera de tercer nivel. En el menú del *notebook*: **Help** » **Markdown** se puede acceder a la sintaxis básica del lenguaje *Markdown*.


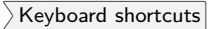
Cada *notebook* dispone de una barra de herramientas típica para guardar, cortar, pegar, etc., y botones para ejecutar el contenido de una celda o para, en caso de necesidad, interrumpir la ejecución. Otras funciones están accesibles a través del menú. Aquí sólo citaremos la opción **File** » **Make a Copy...**, que realiza una copia del *notebook* y **File** » **Download as** que proporciona una exportación del *notebook* a un archivo de diverso formato: desde el propio formato *.ipynb*, a un fichero *.py* con el contenido de todas las celdas (las de tipo **Markdown** aparecen como comentarios), o también ficheros HTML o PDF.

Para cerrar un *notebook* usaremos la opción del menú **File** » **Close and Halt**. Para cerrar completamente el entorno debemos volver a la terminal desde la que ejecutamos la orden `jupyter notebook` y pulsar **ctrl** + **C**; a continuación se nos pedirá confirmación para detener el servicio, que habrá que hacer

<sup>7</sup> *Markdown* es un lenguaje de marcado ligero que permite formatear de forma fácil y legible un texto.

pulsando . Si por error hemos cerrado la ventana *Home* del navegador podemos recuperarla en la dirección `http://localhost:8888`

Si ya disponemos de un *notebook* y queremos seguir trabajando sobre él, podemos abrirlo desde la ventana *Home* del navegador moviéndonos en el árbol de directorios hasta encontrarlo. Obsérvese que por restricciones de seguridad, el servicio no da acceso a directorios por encima del de partida, que coincide con el directorio desde el que se ha ejecutado la orden `jupyter notebook`. Para poder cargar un *notebook* que no esté accesible de este modo, debemos usar el botón de  que nos permitirá localizar en nuestro ordenador el fichero adecuado.

Para finalizar con esta breve introducción a Jupyter, queremos hacer referencia al estupendo conjunto de atajos de teclado disponibles que permite realizar ciertas tareas de forma rápida, como crear celdas por encima o por debajo de la celda activa, juntar o dividir el contenido de celdas, definir el tipo de celda, etc. La información está accesible desde el menú  .

# 2

## El lenguaje Python

---

En este capítulo comenzaremos a ver los aspectos básicos del lenguaje: variables, módulos, bucles, condicionales y funciones. Usaremos multitud de ejemplos para ilustrar la sintaxis del lenguaje y lo haremos desde un entorno Jupyter Notebook, por lo que el código irá apareciendo en celdas de entrada y sus correspondientes salidas.

### 2 1

## ASPECTOS BÁSICOS DEL LENGUAJE

Python es un lenguaje *dinámicamente tipado*, lo que significa que las variables pueden cambiar de tipo en distintos momentos sin necesidad de ser previamente declaradas. Las variables son identificadas con un nombre, que debe obligatoriamente comenzar por una letra y en el que se hace la distinción entre mayúsculas y minúsculas, y son definidas mediante el operador de asignación `=`.

### 2 1 1

### Variables numéricas

Veamos algunos ejemplos:

```
a = 2      # define un entero
b = 5.     # define un número real
c = 3+1j   # define un número complejo
d = complex(3,2) # define un número complejo
```

Obsérvese la necesidad de poner un punto para definir el valor como real y no como entero, el uso de `j` en lugar de `i` en los números complejos junto con la necesidad de anteponer un número, y el uso de la función `complex`. Nótese también que la asignación de una variable no produce ninguna salida.

Podemos recuperar el tipo de dato de cada variable con la orden `type`,

```
print type(a), type(b), type(c)
```

```
<type 'int'> <type 'float'> <type 'complex'>
```

Como vemos, Python asigna el tipo a cada variable en función de su definición. Nótese también el uso de la coma con la orden `print`.

Es importante resaltar la diferencia entre los tipos numéricos, pues si no somos cuidadosos podemos caer en el siguiente error:

```
a = 5; b = 2 # definición múltiple de variables
print a+b
print a/b
```

```
7
```

```
2
```

Claramente `a+b` calcula la suma de los valores de las variables, sin embargo `a/b` parece que no calcula correctamente la división. En realidad la respuesta es correcta dado que ambas variables son enteros, y por tanto se realiza la división entre enteros, que corresponde a la parte entera de la división. Esto es lo que se conoce como el carácter *fuertemente tipado* de Python.

Si lo que esperamos es obtener la división real debemos escribir al menos uno de los números en forma real, lo que se hace con el comando `float`:

```
a/float(b)
```

```
2.5
```

Obsérvese que en esta última entrada no ha hecho falta la orden `print` para imprimir el resultado, a diferencia de la entrada anterior. Esto es debido a que el entorno Jupyter no se comporta exactamente como una consola; si la entrada consta de una única orden, el resultado es el mismo que en la consola, pero si en la entrada hay más de una sentencia, entonces sólo se imprimirá la última, a menos que usemos el comando `print`.

Cuando Python opera con números de distinto tipo, realiza la operación transformando todos los números involucrados al mismo tipo, según una jerarquía establecida que va de enteros a reales y luego a complejos:

```
a = 3.
b = 2+3j
c = a+b # suma de real y complejo
print c
print type(c)
```

```
(5+3j)
```

```
<type 'complex'>
```

Los operadores aritméticos habituales en Python son: + (suma), - (resta), \* (multiplicación), / (división), \*\* (potenciación, que también se puede realizar con la función `pow`), // (división entera), que da la parte entera de la división entre dos reales, y el operador % (módulo), que proporciona el resto de la división entre dos números.

Es importante no olvidar el carácter fuertemente tipado que puede observarse en el siguiente ejemplo:

```
a = 5.; b = 3.  
print a/b  
print a//b
```

```
1.6666666666666667  
1.0
```

Nótese que el resultado de la división entera entre dos reales es un real. Del mismo modo:

```
print a % b  
print int(a) % int(b)
```

```
2.0  
2
```

donde se ha hecho uso de la función de conversión a entero `int`.

## 2.1.2 Objetos

Python sigue el paradigma de la *Programación Orientada a Objetos* (POO). En realidad, todo en Python es un objeto. Podemos entender un objeto como un tipo especial de variable en la que no sólo se almacena un valor, o conjunto de valores, sino para el que tenemos disponible también una serie de características y de funciones concretas, que dependerán del objeto en cuestión.

Por ejemplo, si creamos un número complejo

```
a = 3+2j
```

estaremos creando un objeto para el cual tenemos una serie de propiedades, o en el lenguaje de la POO, de *atributos*, como pueden ser su parte real y su parte imaginaria:

```
print a.real  
print a.imag
```

```
3.0  
2.0
```

Los atributos son características de los objetos a las que se accede mediante el operador `.` de la forma `objeto.atributo`.

Cada tipo de objeto suele tener disponible ciertos *métodos*. Un método es una función que actúa sobre un objeto con una sintaxis similar a la de un atributo, es decir, de la forma `objeto.método(argumentos)`. Por ejemplo, la operación de conjugación es un método del objeto complejo:

```
a.conjugate()
```

```
(3-2j)
```

Los paréntesis indican que se trata de una función y son necesarios. En caso contrario, si escribimos

```
a.conjugate
```

```
<function conjugate>
```

el intérprete nos indica que se trata de una función, pero no proporciona lo esperado.

En el entorno Jupyter Notebook, pulsando el tabulador después de escribir `objeto.` nos aparece un menú desplegable que nos muestra los atributos y funciones accesibles al objeto.

### 2.1.3 Listas

Las *listas* son colecciones de datos de cualquier tipo (inclusive listas) que están indexadas, comenzando desde 0:

```
a = [ 1, 2., 3+1j, [3,0] ]
type(a)
```

```
<type 'list'>
```

```
print a[1]
print a[4]
```

```
2.0
```

---

```

IndexError                                Traceback (
  most recent call last)
<ipython-input-23-7eba33f48202> in <module>()
----> 1 print a[4]
```

```
IndexError: list index out of range
```

Hemos definido una lista encerrando sus elementos (de tipos diversos) entre corchetes y separándolos por comas. Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista y el índice del elemento entre corchetes, teniendo en cuenta que el primer elemento tiene índice 0 y por tanto el segundo corresponde al índice 1. Si intentamos acceder al elemento `a[4]` obtenemos un error, pues dicho elemento no existe.

La salida de error en Python es amplia, y entre otras cosas nos marca el lugar donde se ha producido, el tipo de error, en este caso `IndexError`, y en la última línea nos da una breve explicación. En lo que sigue, para simplificar las salidas de errores sólo mostraremos la última línea.

Si algún elemento de la lista es otra lista, podemos acceder a los elementos de esta última usando el corchete dos veces, como en el siguiente ejemplo:

```
print a[3]
print a[3][1]
```

```
[3, 0]
0
```

Las listas son estructuras de datos muy potentes que conviene aprender a manejar con soltura. Podemos consultar los métodos a los que tenemos acceso en una lista usando la función de autocompletado. El siguiente ejemplo es autoexplicativo y muestra el funcionamiento de alguno de estos métodos:

```
a = [25, 33, 1, 15, 33]
a.append(0) # agrega 0 al final
print a
```

```
[25, 33, 1, 15, 33, 0]
```

```
a.insert(3,-1) # inserta -1 en la posición 3
print a
```

```
[25, 33, 1, -1, 15, 33, 0]
```

```
a.reverse() # invierte el orden
print a
```

```
[0, 33, 15, -1, 1, 33, 25]
```

```
a.pop() # elimina el último elemento y lo devuelve
```

```
25
```

```
print a
```

```
[0, 33, 15, -1, 1, 33]
```

```
print a.pop(3) # elimina el elemento de índice 3
print a
```

```
-1
[0, 33, 15, 1, 33]
```

```
a.extend([10,20,30]) # añade elementos a la lista
print a
```

```
[0, 33, 15, 1, 33, 10, 20, 30]
```

Nótese la diferencia con el siguiente:

```
a.append([10,20,30]) # añade el argumento a la lista
print a
```

```
[0, 33, 15, 1, 33, 10, 20, 30, [10, 20, 30]]
```

Por otra parte, es frecuente que Python utilice los operadores aritméticos con diferentes tipos de datos y distintos resultados. Por ejemplo, los operadores suma y multiplicación pueden aplicarse a listas, con el siguiente resultado:

```
a = [1,2,3]; b = [10,20,30]
print a*3
print a+b
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 10, 20, 30]
```

La última acción se podría haber obtenido usando el método `extend`:

```
a.extend(b)
print a
```

```
[1, 2, 3, 10, 20, 30]
```

Nótese que el uso del método hubiera sido equivalente a escribir `a=a+b`. En general, el uso de métodos proporciona mejor rendimiento que el uso de otras acciones, pero hemos de ser conscientes de que el objeto sobre el que se aplica puede quedar modificado al usar un método.

### ***Slicing***

Una de las formas más interesantes de acceder a los elementos de una lista es mediante el operador de corte o *slicing*, que permite obtener una parte de los elementos de una lista:



```
a = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
print a[2:5] # accedemos a los elementos 2,3,4
```

```
[7, 6, 5]
```

Como vemos, el *slicing* [*n:m*] accede a los elementos de la lista desde *n* hasta *m* (el último sin incluir). Admite un parámetro adicional, y cierta flexibilidad en la notación:

```
print a[1:7:2] # desde 1 hasta 6, de 2 en 2
```

```
[8, 6, 4]
```

```
print a[:3] # al omitir el primero se toma desde el inicio
```

```
[9, 8, 7]
```

```
print a[6:] # al omitir el último se toma hasta el final
```

```
[3, 2, 1, 0]
```

Aunque el acceso a índices incorrectos genera error en las listas, no ocurre lo mismo con el *slicing*:

```
print a[:20]
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
print a[12:15] # si no hay elementos, resulta vacío
```

```
[]
```

En las listas, y por supuesto también con el *slicing*, se pueden usar índices negativos que equivalen a contar desde el final:

```
print a[-1] # -1 refiere la última posición
```

```
0
```

```
print a[-5:-3]
```

```
[4, 3]
```

```
print a[3:-3]
```

```
[6, 5, 4, 3]
```

El *slicing* también permite añadir, borrar o reemplazar elementos en las listas:

```
a = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
a[1:3] = [] # borra elementos 1 y 2
print a
```

```
[9, 6, 5, 4, 3, 2, 1, 0]
```

```
a[2:5] = [-1,-2,-3,-4] # reemplaza elementos 2 a 4
print a
```

```
[9, 6, -1, -2, -3, -4, 2, 1, 0]
```

```
a[1:1] = [0,1,2] # añade la lista en la posición 1
print a
```

```
[9, 0, 1, 2, 6, -1, -2, -3, -4, 2, 1, 0]
```

```
a[:0] = a[-5:-1] # añadimos al inicio
print a
```

```
[-3, -4, 2, 1, 9, 0, 1, 2, 6, -1, -2, -3, -4, 2, 1, 0]
```

```
a[:] = [] # vaciamos la lista
print a
```

```
[]
```

### 2.1.4 Cadenas de caracteres

Las *cadenas* no son más que texto encerrado entre comillas:

```
a = "Hola"; b = 'mundo'
print a
print b
print type(a)
```

```
Hola
mundo
<type 'str'>
```

en las que se puede comprobar que da igual definir las con comillas simples o dobles, lo que es útil si queremos cadenas que incluyan estos caracteres:

```
a = "Esto es una 'string'"
print a
```

Esto es una 'string'

Si queremos construir cadenas con más de una línea usamos la triple comilla """:

```
a = """Esto es un cadena
muy larga que tiene
muchas líneas"""
print a
```

Esto es un cadena  
muy larga que tiene  
muchas líneas

En ocasiones es preciso incluir caracteres no ASCII en una cadena, para lo cual Python provee de soporte Unicode. Para señalar que una cadena posee caracteres *extraños* usaremos el indicador `u` al comienzo de la misma (antes de las comillas). Por ejemplo

```
cadena = u'á é í ó ú'
```

En otros momentos, necesitaremos que la cadena de caracteres sea interpretada tal cual (sin caracteres de escape), para lo cual usaremos una `r` (por *raw string*) precediendo a la cadena:

```
cadena = r'\LaTeX'
```

Podemos acceder a los elementos individuales de una cadena mediante índices, como si fuera una lista:

```
cadena = "Hola mundo"
print cadena[0]
print cadena[4]
```

'H'  
, ,

pero las cadenas son *inmutables*, esto es, no es posible alterar sus elementos (veremos este asunto más adelante en la sección 3.3.2):

```
cadena[5] = 'M' # error: la cadena no es modificable
```

TypeError: 'str' object does not support item assignment

En particular, esto significa que cualquier transformación que llevemos a cabo con un método no alterará la cadena original.

El *slicing* también funciona con las cadenas de caracteres

```
a = "Esto es una cadena de caracteres"
print a[:19]
print a[19:]
```

```
'Esto es una cadena '
'de caracteres'
```

Hay una gran cantidad de métodos para manejar *strings* que permiten cambiar la capitalización, encontrar caracteres dentro de una cadena o separar cadenas en trozos en función de un carácter dado. Emplazamos al lector a usar la ayuda en línea del intérprete para aprender el funcionamiento de éstos y otros métodos.

### 2.1.5 Diccionarios

En algunas ocasiones es interesante disponer de listas que no estén indexadas por números naturales, sino por cualquier otro elemento (*strings* habitualmente, o cualquier tipo de dato inmutable). Python dispone de los *diccionarios* para manejar este tipo de listas:

```
colores = {'r': 'rojo', 'g': 'verde', 'b': 'azul'}
print type(colores)
```

```
<type 'dict'>
```

```
print colores['r']
```

```
'rojo'
```

```
colores['k'] = 'negro' # añadimos un nuevo elemento
print colores
```

```
{'k': 'negro', 'r': 'rojo', 'b': 'azul', 'g': 'verde'}
```

Observar que el orden dentro de los elementos de la lista es irrelevante pues la indexación no es numerada.

El objeto que se usa como índice se denomina *clave*. Podemos pensar entonces en un diccionario como un conjunto no ordenado de pares, **clave: valor** donde cada clave ha de ser única (para ese diccionario). Podemos acceder a ellas usando los métodos adecuados:

```
print colores.keys() # claves
print colores.values() # valores
```

```
['negro', 'rojo', 'azul', 'verde']
```

Entre los diversos métodos accesibles para un diccionario disponemos del método `pop` que permite eliminar una entrada en el diccionario:

```
print colores.pop('b') # devuelve el valor eliminado
print colores
```

```
'azul'
{'k': 'negro', 'r': 'rojo', 'g': 'verde'}
```

## 2.1.6 Tuplas

Para terminar con los tipos de datos principales haremos mención a las *tuplas*. Las tuplas son un tipo de dato similar a las listas (es decir, una colección indexada de datos) pero que no pueden alterarse una vez definidas (son inmutables):

```
a = (1,2.,3+1j,"hola")
print type(a)
```

```
<type 'tuple'>
```

```
print a[2]
```

```
(3+1j)
```

La definición es similar a la de una lista, salvo que se usan paréntesis en lugar de corchetes, y el acceso a los elementos es idéntico.

```
a[0] = 10. # error: no se puede modificar una tupla
```

```
TypeError: 'tuple' object does not support item
assignment
```

Pero como podemos observar, no es posible alterar sus elementos ni tampoco añadir otros nuevos.

Las tuplas son muy útiles para empaquetar y desempaquetar datos, como puede verse en el siguiente ejemplo:

```
a = 1,2,'hola' # creamos una tupla (sin paréntesis!)
print type(a)
```

```
<type 'tuple'>
```

```
x, y, z = a  # desempaquetamos la tupla en variables x,y,z
print x
print y
print z
```

```
1
2
'hola'
```

Atención a la creación de tuplas de un único elemento:

```
a = (1)
b = (1,)
c = 1,
print type(a), type(b), type(c)
```

```
<type 'int'> <type 'tuple'> <type 'tuple'>
```

Como veremos luego, las tuplas son útiles para pasar un conjunto de datos a una función. También están detrás de la asignación múltiple de variables:

```
a = s,t,r = 1, 'dos', [1,2,3]
print a
print t
```

```
(1, 'dos', [1, 2, 3])
'dos'
```

que es particularmente útil para intercambiar valores de variables:

```
a,b = 0,1
b,a = a,b  # intercambiamos a y b
print a, b
```

```
1 0
```

## 2.2

## MÓDULOS

Una de las características principales de Python es su modularidad. La mayoría de funciones accesibles en Python están empaquetadas en *módulos*, que precisan ser cargados previamente a su uso, y sólo unas pocas funciones son cargadas con el núcleo principal. Por ejemplo, no se dispone de forma inmediata de la mayor parte de funciones matemáticas comunes si no se

ha cargado antes el módulo apropiado. Por ejemplo, la función seno no está definida:

```
sin(3.)
```

```
NameError: name 'sin' is not defined
```

Si queremos poder usar ésta u otras funciones matemáticas debemos *importar* el módulo con la orden `import`:

```
import math
```

Ahora tenemos a nuestra disposición todas las funciones del módulo matemático. Puesto que todo en Python es un objeto (incluidos los módulos), el lector entenderá perfectamente que el acceso a las funciones del módulo se haga de la forma `math.función`:

```
math.sin(3.)
```

```
0.1411200080598672
```

Para conocer todas las funciones a las que tenemos acceso dentro del módulo disponemos de la orden `dir`<sup>1</sup>

```
dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Una de las características más apreciadas de Python es su extensa biblioteca de módulos que nos proveen de funciones que permiten realizar las tareas más diversas. Además, esta modularización del lenguaje hace que los programas creados puedan ser reutilizados con facilidad. Sin embargo, no suele ser bien aceptada la necesidad de anteponer el nombre del módulo para tener acceso a sus funciones. Es posible evitar el tener que hacer esto si cargamos los módulos del siguiente modo:

```
from math import *
```

Ahora, si queremos calcular  $\sqrt{2}$ , escribimos

---

<sup>1</sup>Sin argumentos, la orden `dir()` devuelve un listado de las variables actualmente definidas.

```
sqrt(2)
```

```
1.4142135623730951
```

Lógicamente, esta forma de cargar los módulos tiene ventajas evidentes en cuanto a la escritura de órdenes, pero tiene también sus inconvenientes. Por ejemplo, es posible que haya más de un módulo que use la misma función, como es el caso de la raíz cuadrada, que aparece tanto en el módulo `math` como en el módulo `cmath`. De manera que podemos encontrarnos situaciones como la siguiente:

```
import math
import cmath
math.sqrt(-1)
```

```
ValueError: math domain error
```

```
cmath.sqrt(-1)
```

```
1j
```

Como vemos, hemos cargado los módulos `math` y `cmath` y calculado la raíz cuadrada de  $-1$  con la función `sqrt` que posee cada módulo. El resultado es bien distinto: la función raíz cuadrada del módulo `math` no permite el uso de números negativos, mientras que la función `sqrt` del módulo `cmath` sí. Si en lugar de cargar los módulos como en el último ejemplo los hubiésemos cargado así:

```
from cmath import *
from math import *
```

¿qué ocurrirá al hacer `sqrt(-1)`? Como el lector puede imaginar, la función `sqrt` del módulo `cmath` es sobrescrita por la del módulo `math`, por lo que sólo la última es accesible.

Existe una tercera opción para acceder a las funciones de los módulos que no precisa importarlo al completo. Así,

```
from cmath import sqrt
from math import cos, sin
```

nos deja a nuestra disposición la función raíz cuadrada del módulo `cmath` y las funciones trigonométricas seno y coseno del módulo `math`. Es importante señalar que con este método de importación no tenemos acceso a ninguna otra función de los módulos que no hubiera sido previamente importada. Esta última opción es de uso más frecuente en los *scripts*, debido a que con ella cargamos exclusivamente las funciones que vamos a necesitar y de esa forma mantenemos el programa con el mínimo necesario de recursos.



En el interactivo es más frecuente cargar el módulo al completo, y es aconsejable hacerlo sin el uso de `*`. De hecho, hay una posibilidad adicional que nos evita tener que escribir el nombre del módulo al completo, seguido del punto para usar una función. Si realizamos una importación del módulo como sigue.

```
import math as m
```

entonces no es necesario escribir `math.` para acceder a la funciones, sino

```
m.cos(m.pi)
```

```
-1.0
```

### 2.2.1 Otros módulos de interés

La cantidad de módulos disponibles en Python es enorme, y en estas notas veremos con detenimiento algunos de ellos relacionados con la computación científica. No obstante, conviene conocer algunos otros módulos de la biblioteca estándar que se muestran en la tabla 2.1:

Cuadro 2.1: Algunos módulos de la biblioteca estándar

Módulo	Descripción
<code>math</code>	Funciones matemáticas
<code>cmath</code>	Funciones matemáticas con complejos
<code>fractions</code>	Números racionales
<code>os</code>	Funcionalidades del sistema operativo
<code>sys</code>	Funcionalidades del intérprete
<code>re</code>	Coincidencia en patrones de cadenas
<code>datetime</code>	Funcionalidades de fechas y tiempos
<code>pdb</code>	Depuración
<code>random</code>	Números aleatorios
<code>ftplib</code>	Conexiones FTP
<code>MySQLdb</code>	Manejo de bases de datos MySQL
<code>smtplib</code>	Envío de e-mails

**2 3****CONTROL DE FLUJO****2 3 1 Bucles**

Una característica esencial de Python es que la sintaxis del lenguaje impone obligatoriamente que escribamos con cierta claridad. Así, los bloques de código deben ser obligatoriamente sangrados:

```
for i in range(3):  
    print i
```

```
0  
1  
2
```

La sintaxis de la orden `for` es simple: la variable `i` recorre la lista generada por `range(3)`, finalizando con dos puntos `(:)` obligatoriamente. La siguiente línea debe ser sangrada, bien con el tabulador, bien con espacios (uno es suficiente, aunque lo habitual es cuatro), aunque podemos ver que el entorno Jupyter o la consola IPython realizan la sangría por nosotros. Para indicar el final del bucle debemos volver al sangrado inicial.

Como podemos ver, la función `range(n)` crea una lista de números de  $n$  elementos, comenzando en 0. Pero es posible que el rango comience en otro valor, o se incremente de distinta forma:

```
print range(5, 10)  
print range(1, 10, 3)  
print range(-10, -100, -30)
```

```
[5, 6, 7, 8, 9]  
[1, 4, 7]  
[-10, -40, -70]
```

Nótese que el bucle en Python corre a través de la lista y no de los índices de ésta, como se muestra en el siguiente ejemplo:

```
a = ['hola', 'mundo']  
for b in a:  
    print b
```

```
hola  
mundo
```

No es seguro modificar la lista sobre la que se está iterando dentro del bucle. En caso de querer hacerlo es preciso iterar sobre una copia.

De igual modo que iteramos sobre una lista, puede hacerse sobre una tupla o un diccionario. En este último caso, es posible acceder de varias formas:

```
colores = {'r': 'rojo', 'g': 'verde', 'b': 'azul'}
for i in colores.keys():
    print i,colores[i]
```

```
r rojo
b azul
g verde
```

o simultáneamente, con método el `iteritems`:

```
for x,y in colores.iteritems():
    print x,y
```

```
r rojo
b azul
g verde
```

### 2 3 2 Condicionales

La escritura de sentencias condicionales es similar a la de los bucles `for`, usando el sangrado de línea para determinar el bloque:

```
if 5%3 == 0:
    print "5 es divisible entre 3"
elif 5%2 == 0:
    print "5 es divisible por 2"
else:
    print "5 no divisible ni por 2 ni por 3"
```

```
5 no es divisible ni por 2 ni por 3
```

La orden `if` evalúa la operación lógica “el resto de la división de 5 entre 3 es igual a cero”; puesto que la respuesta es negativa, se ejecuta la segunda sentencia (`elif`), que evalúa si “el resto de la división de 5 entre 2 es igual a cero”; como esta sentencia también es negativa se ejecuta la sentencia `else`. Es posible poner todos los `elif` que sean necesarios (o incluso no ponerlos), y el bloque `else` no es obligatorio.

En Python, al igual que C, cualquier número distinto de cero es verdadero, mientras que cero es falso. Las condiciones pueden ser también una cadena de texto o una lista, que serán verdaderas si tienen longitud no nula, y falsas si son vacías. Por su parte, los *operadores de comparación* en Python son `==` (igual), `!=` (distinto), `>` (mayor que), `>=` (mayor o igual que), `<` (menor que) y `<=` (menor o igual que), y los *operadores lógicos* son `and`, `or` y `not`.

**2.3.3 Bucles condicionados**

Obsérvese este ejemplo con la sentencia `while`:

```
a,b = 0,1 # Inicialización de la sucesión de Fibonacci
while b<20:
    print b,
    a,b = b,a+b
```

```
1 1 2 3 5 8 13
```

Es interesante analizar un poco este breve código que genera unos cuantos términos de la sucesión de Fibonacci. En especial, hemos de prestar atención a cómo usamos tuplas para las asignaciones múltiples que realizamos en la primera y última líneas; en la primera hacemos `a=0` y `b=1` y en la última se realiza la asignación `a=b` y `b=a+b`, en la que debemos notar que, antes de realizar la asignación, se evalúan los lados derechos (de izquierda a derecha).

También disponemos de las sentencias `break` y `continue` para terminar o continuar, respectivamente, los bucles `for` o `while`. Asimismo, la sentencia `else` tiene sentido en un bucle y se ejecuta cuando éste ha terminado, en el caso de `for`, o cuando es falso, en el caso de `while`. Veamos el siguiente ejemplo:

```
1 for n in range(2,10):
2     for x in range(2,n):
3         if n % x == 0: # si n es divisible por x
4             print n,'es igual a',x,'*',n/x
5             break
6     else:
7         print n,'es primo'
```

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

El `break` de la línea 5 interrumpe la búsqueda que hace el bucle que comienza en la línea 2. Si este bucle finaliza sin interrupción, entonces se ejecuta el bloque `else` de la línea 6.

Por último, Python dispone también de la orden `pass` que no tiene ninguna acción, pero que en ocasiones es útil para estructurar código que aún no ha sido completado, por ejemplo

```
for n in range(10):
    if n % 2 == 0:
        print "n es par"
    else:
        pass    # ya veremos qué hacemos aquí
```

## 2 4

## FUNCIONES DEFINIDAS POR EL USUARIO

Las funciones son trozos de código que realizan una determinada tarea. Vienen definidas por la orden `def` y a continuación el nombre que las define seguido de dos puntos. Siguiendo la sintaxis propia de Python, el código de la función está sangrado. La principal característica de las funciones es que permiten pasarles argumentos de manera que la tarea que realizan cambia en función de dichos argumentos.

```
def fibo(n):    # sucesión de Fibonacci hasta n
    a,b = 0,1
    while b < n:
        print b,
        a, b = b, a+b
```

Ahora efectuamos la llamada a la función:

```
fibo(1000)    # llamada a la función con n=1000
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Como puede verse, esta función imprime los términos de la sucesión de Fibonacci menores que el valor `n` introducido.

Si quisiéramos almacenar dichos términos en una lista, podemos usar la orden `return` que hace que la función pueda devolver algún valor, si es necesario:

```
1 def fibolist(n):    # lista de Fibonacci hasta n
2     sucesion=[]    # creación de lista vacía
3     a,b = 0,1
4     while b < n:
5         sucesion.append(b)    # añadir b a la lista sucesion
6         a, b = b, a+b
7     return sucesion
8
9 fibolist(100)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
a = fibolist(250)
print a
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

A diferencia de la función `fibo` definida antes, la función `fibolist` devuelve la lista creada a través de la orden `return` (línea 7). Si `return` no va acompañado de ningún valor, se retorna `None`, al igual que si se alcanza el final de la función sin encontrar `return`:

```
a = fibo(50)
```

```
1 1 2 3 5 8 13 21 34
```

```
print a
```

```
None
```

Si queremos devolver más de un valor, lo podemos empaquetar en una tupla.

### 2.4.1 Importando funciones definidas por el usuario

Aunque las funciones pueden ser definidas dentro del intérprete para su uso es más habitual almacenarlas en un fichero, bien para poder ser ejecutadas desde el mismo, o bien para ser importadas como si se tratara de un módulo. Para guardar el contenido de una celda del entorno Jupyter a un archivo, bastará usar la *magic function*, `%%file` seguido del nombre del archivo:

```
%%file tang.py
from math import sin,cos,pi
def tangente(x):
    if cos(x) != 0:
        return sin(x)/cos(x)
    else:
        print "La tangente es infinita"
        return

x = tangente(pi)
print 'La tangente de pi es',x
```

Si ahora ejecutamos el archivo desde la consola:

```
$ python tang.py
La tangente de pi es -1.22460635382e-16
```

Aunque también podemos ejecutar desde el entorno Jupyter:

```
run tang.py
```

La tangente de pi es  $-1.22460635382e-16$

En lugar de ejecutar la función definida también podemos cargarla como si se tratara de un módulo (reiniciar Jupyter en este punto):

```
import tang
```

La tangente de pi es  $-1.22460635382e-16$

y por tanto tendremos acceso a la función:

```
tang.tangente(3)
```

$-0.1425465430742778$

o a las variables definidas en la función:

```
print tang.x
```

$-1.2246063538223773e-16$

Sin embargo,

```
print x
```

NameError: name 'x' is not defined

Nótese cómo al cargar el módulo, éste se ejecuta y además nos proporciona todas las funciones y variables presentes en el módulo (inclusive las importadas por él), pero anteponiendo siempre el nombre del módulo. Obsérvese que la variable `x` no está definida, mientras que sí lo está `tang.x`.

Si realizamos la importación con `*` (reiniciar Jupyter):

```
x = 5
from tang import *
```

La tangente de pi es  $-1.22460635382e-16$

```
print tangente(pi/4)
print x
```

$0.9999999999999999$

$-1.2246063538223773e-16$

tendremos todas las funciones presentes en el módulo (salvo las que comiencen por `_`) sin necesidad de anteponer el nombre, pero como podemos observar

en el ejemplo, podemos alterar las variables propias que tengamos definidas, razón por la cual no recomendamos este tipo de importación.

Finalmente si realizamos la importación selectiva:

```
from tang import tangente
```

La tangente de pi es  $-1.22460635382e-16$

```
print tangente(3)
print x
```

$-0.1425465430742778$

NameError: name 'x' is not defined

la función `tangente` está disponible, pero nada más. Aun así, obsérvese que se ha ejecutado el módulo en el momento de la importación.

Para evitar que el código se ejecute cuando importamos podemos separar la función del resto del código del siguiente modo:

```
from math import sin,cos,pi
def tangente(x):
    if cos(x) != 0:
        return sin(x)/cos(x)
    else:
        print "La tangente es infinita"
        return

if __name__ == "__main__":
    x = tangente(pi)
    print 'La tangente de pi es',x
```

Lo que sucede es que cuando ejecutamos el código con `python tang.py` desde la consola (o con `run tang.py` desde el entorno Jupyter), la variable `__name__` toma el valor `'__main__'`, por lo que el fragmento final se ejecuta, lo que no ocurrirá si lo importamos.

Es importante resaltar que por razones de eficiencia, los módulos se importan una sola vez por sesión en el intérprete, por lo que si son modificados es necesario reiniciar la sesión o bien volver a cargar el módulo con la orden `reload(módulo)`.

## ¿Dónde están los módulos?

Cuando se importa un módulo de nombre `tang` el intérprete busca un fichero `tang.py` en el directorio actual o en la lista de directorios dados por la variable `PYTHONPATH`. Si dicha variable no está configurada, o el fichero no se encuentra allí, entonces se busca en una lista de directorios que depende



de la instalación que se tenga. Podemos acceder a esta lista con la variable `sys.path` del módulo `sys`.

**2 5****EJERCICIOS**

**E.1** ¿Cuál de las siguientes órdenes produce un error y por qué?

- (a) `a = complex(3,1)`
- (b) `a = complex(3)+ complex(0,1)`
- (c) `a = 3+j`
- (d) `a = 3+(2-1)j`

**E.2** ¿Cuál de las siguientes sentencias producirá un error y por qué?

- (a) `a = [1,[1,1,[1]]]; a[1][2]`
- (b) `a = [1,[1,1,[1]]]; a[2]`
- (c) `a = [1,[1,1,[1]]]; a[0][0]`
- (d) `a = [1,[1,1,[1]]]; a[1,2]`

**E.3** Dada la cadena `s = 'Hola mundo'`, encuentra el método adecuado de producir las cadenas separadas `a = 'Hola'` y `b = 'mundo'`.

**E.4** Explica el resultado del siguiente código:

```
s=0
for i in range(100):
    s += i
print s
```

Usa el comando `sum` para reproducir el mismo resultado en una línea.

**E.5** Usa el comando `range` para producir las listas

- (a) `[10, 8, 6, 4, 2, 0]`
- (b) `[10, 8, 6, 4, 2, 0, 2, 4, 6, 8, 10]`

**E.6** Explica el resultado del siguiente código:

```
s=[0]
for i in range(100):
    s.append(s[i-1]+i)
print s
```

**E.7** Cómo puedes generar la siguiente lista:

[0, 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105]

**E.8** Construye el vigésimo elemento de la siguiente sucesión definida por recurrencia:

$$u_{n+3} = u_{n+2} - u_{n+1} + 3u_n, \quad u_0 = 1, \quad u_1 = 2, \quad u_3 = 3.$$

**E.9** Escribir una función `reverse` que acepte una cadena de caracteres y devuelva como resultado la cadena de caracteres invertida en orden, es decir, el resultado de `reverse('hola')` debe ser `'aloh'`.

**E.10** Usando el Ejercicio E.9, escribir una función `is_palindromo` que identifique palíndromos, (esto es, palabras que leídas en ambas direcciones son iguales, por ejemplo *radar*). Es decir, `is_palindromo('radar')` debería dar como resultado `True`.

# 3

## Aspectos avanzados

---

### 3 1

#### ALGO MÁS SOBRE FUNCIONES

##### 3 1 1 Documentando funciones

Se puede documentar una función en Python añadiendo una cadena de documentación justo detrás de la definición de función:

```
def mifuncion(parametro):  
    """  
    Esta función no hace nada.  
  
    Absolutamente nada  
    """  
    pass
```

En tal caso, la función `help(mifuncion)` o escribiendo `mifuncion?` en IPython nos mostrará la documentación de la función.

##### 3 1 2 Argumentos de entrada

Es posible definir funciones con un número variable de argumentos. Una primera opción consiste en definir la función con valores por defecto:

```
def fibonacci(n,a=0,b=1):  
    sucesion = []  
    while b < n:  
        sucesion.append(b)  
        a,b = b,b+a  
    return sucesion
```

La llamada a esta función puede realizarse de diversos modos.

- Usando sólo el argumento posicional: `fibonacci(100)`.

- Pasando sólo alguno de los argumentos opcionales (junto con el posicional): `fibonacci(100,3)`, en cuyo caso `a=3` y `b=1`.
- Pasando todos los argumentos: `fibonacci(10,2,3)`.

También es posible realizar la llamada usando los nombres de las variables por defecto, así

```
fibonacci(100,b=3)  # equivale a fibonacci(100,0,3)
fibonacci(100,b=4,a=2)  # equivale a fibonacci(100,2,4)
fibonacci(b=4,a=2)  # error: falta el argumento n
```

Otra opción es incluir un número indeterminado de parámetros en una tupla, que es llamada como *\*argumento*; por ejemplo:

```
def media(*valores):
    if len(valores) == 0: # len = longitud de la tupla
        return 0.0
    else:
        sum = 0.
        for x in valores:
            sum += x # equivale a sum = sum + x
        return sum / len(valores)
```

La función calcula el valor medio de un número indeterminado de valores de entrada que son empaquetados con el argumento `*valores` en una tupla, de manera que ahora podemos ejecutar:

```
print media(1,2,3,4,5,6)
print media(3,6,7,8)
```

```
3.5
6.0
```

Si la función tiene además argumentos por defecto, entonces podemos empaquetar la llamada a través de un diccionario:

```
def funarg(obligatorio,*otros,**opciones):
    print obligatorio
    print '-'*40
    print otros
    print '*'*40
    print opciones
```

```
funarg("otro",2,3,4)
```

```
otro
```

---

```
(2, 3, 4)
*****
{}
```

```
funarg("hola",a=2,b=4)
```

```
hola
```

---

```
()
*****
{'a': 2, 'b': 4}
```

```
funarg("hola","uno","dos","tres",a=2,b=3,c='fin')
```

```
hola
```

---

```
('uno', 'dos', 'tres')
*****
{'a': 2, 'c': 'fin', 'b': 3}
```

No sólo la definición de función permite el empaquetado de argumentos, sino también es posible usarlo en las llamadas. Por ejemplo, la función `media` definida antes se puede llamar de la forma

```
media(*range(1,11))
```

```
5.5
```

o en la función `fibonacci`,

```
d = {'a':5, 'b':3}
fibonacci(50,**d)
```

```
[3, 8, 11, 19, 30, 49]
```

```
c = (2,3)
fibonacci(50,*c)
```

```
[3, 5, 8, 13, 21, 34]
```

### 3.1.3 Funciones lambda

Las funciones *lambda* son funciones anónimas de una sola línea que pueden ser usadas en cualquier lugar en el que se requiera una función. Son útiles para reducir el código, admiten varios parámetros de entrada o salida y no pueden usar la orden `return`:

```
f = lambda x,y: (x+y, x**y)
f(2,3)
```

(5, 8)

Son especialmente útiles si se quiere devolver una función como argumento de otra:

```
import math
def seno(w):
    return lambda t: math.sin(t*w)

f = seno(math.pi)
g = seno(2*math.pi)
print f(.5), g(.5)
```

1.0, 1.2246063538223773e-16

### 3.1.4 Variables globales y locales

Las variables globales son aquellas definidas en el cuerpo principal del programa, mientras que las variables locales son aquellas variables internas a una función que sólo existen mientras la función se está ejecutando, y que desaparecen después. Por ejemplo, si definimos la siguiente función

```
def area_rectangulo(lado):
    area = lado*lado
    return area
```

y ejecutamos

```
print area_rectangulo(3)
```

9

y luego

```
print area
```

NameError: name 'area' is not defined

observamos que la variable `area` no se encuentra en el espacio de nombres por tratarse de una variable local de la función `area_rectangulo`. ¿Qué ocurre si definimos esta variable previamente?

```
area = 10
def area_rectangulo(lado):
    area = lado*lado
    return area
```

```
print area_rectangulo(3)
print area
```

```
9
10
```

Como vemos, la variable `area` ahora sí existe en el espacio de nombres global, pero su valor no se ve alterado por la función, pues ésta ve su variable `area` de manera local.

¿Y si tratamos de imprimir el valor de `area` antes de su evaluación?

```
area = 10
def area_rectangulo(lado):
    print area
    area = lado*lado
    return area
```

```
print area_rectangulo(3)
```

```
UnboundLocalError: local variable 'area' referenced
before assignment
```

Entonces obtenemos un error debido a que intentamos asignar de manera local una variable que aun no está definida (para la función).

Cualquier variable que se cambie o se cree dentro de una función es local, a menos que expresamente indiquemos que esa variable es global, lo que se hace con la orden `global`:

```
area = 10
def area_rectangulo(lado):
    global area
    print area
    area = lado*lado
    return area
```

```
print area_rectangulo(3)
```

```
10
9
```

Obsérvese que ahora

```
print area
```

```
9
```

ha cambiado su valor, debido a que ya no hay una variable local `area` en la función.

### Variables locales y objetos mutables

Sin embargo, el empleo de ciertos métodos sobre variables que son mutables (véase la sección 3.3.2) en el cuerpo de una función puede modificar de manera global estas variables sin necesidad de ser declaradas como globales. Obsérvese el siguiente ejemplo:

```
def fun(a,b):
    a.append(0)
    b = a+b
    return a,b
```

```
a = [1]
b = [0]
print fun(a,b)
```

```
[1, 0] [1, 0, 0]
```

La función `fun` usa el método `append` para agregar el elemento 0 a la lista de entrada `a`, quedando por tanto la lista modificada por el método. Por su parte, la asignación que hacemos en `b=a+b` crea una nueva variable local `b` dentro de la función, que no altera a la variable `b` de fuera de ese espacio. De este modo,

```
print a,b
```

```
[1, 0] [0]
```

es decir, `a` ha sido modificada, pero `b` no, aun tratándose ambas variables de listas. Sin embargo, en el primer caso, la lista es modificada mediante un método, mientras que en el segundo, se crea una variable local que es la que se modifica.

Por último señalar que las variables del espacio global se pueden usar en la definición de parámetros por defecto, pero son evaluadas sólo en la definición de la función, y no en cada llamada:



```
a = 2
def pot(x,y=a):
    return x**y
pot(2), pot(2,3)
```

```
(4, 8)
```

```
a = 4
pot(2)
```

```
4
```

### 3 2

## ENTRADA Y SALIDA DE DATOS

Es habitual que nuestros programas necesiten datos externos que deben ser pasados al intérprete de una forma u otra. Típicamente, para pasar un dato a un programa interactivamente mediante el teclado (la entrada estándar) podemos usar la función `raw_input`

```
raw_input('Introduce algo\n') # \n salta de línea
```

```
Introduce algo
```

```
hola 2
```

```
'hola 2'
```

Obsérvese que la entrada estándar es convertida a una cadena de caracteres, que probablemente deba ser tratada para poder usar los datos introducidos.

En un *script* es más frecuente el paso de parámetros en el momento de la ejecución. Para leer los parámetros introducidos disponemos de la función `argv` del módulo `sys`. Por ejemplo, si tenemos el siguiente código en un archivo llamado `entrada.py`

```
import sys
print sys.argv
```

entonces, la siguiente ejecución proporciona:

```
$ python entrada.py 2 25 "hola"
['entrada.py', '2', '25', 'hola']
```

Como vemos, `sys.argv` almacena en una lista todos los datos de la llamada, de manera que es fácil manejarlos teniendo presente que son considerados como cadenas de caracteres, por lo que es probable que tengamos que realizar una conversión de tipo para tratarlos correctamente.

Por ejemplo, si tenemos el archivo `fibo.py` con el siguiente contenido

```
# -*- coding: utf-8 -*-
import sys

def fibonacci(n,a=0,b=1):
    """Términos de la sucesión de Fibonacci hasta orden n.
    """
    sucesion = []
    while b < n:
        sucesion.append(b)
        a,b = b,b+a
    return sucesion

if __name__ == "__main__":
    x = int(sys.argv[1]) # convertimos a entero la entrada
    print fibonacci(x)
```

haciendo la siguiente ejecución se obtiene

```
$ python fibo.py 30
[1, 1, 2, 3, 5, 8, 13, 21]
```

Y si olvidamos hacer la llamada con el parámetro obtendremos error:

```
$ python fibo.py
Traceback (most recent call last):
  File "fibo.py", line 14, in <module>
    x = int(sys.argv[1])
IndexError: list index out of range
```

Desde el entorno Jupyter, usaríamos run:

```
run fibo.py 30
```

```
[1, 1, 2, 3, 5, 8, 13, 21]
```

### 3.2.1 Salida formateada

La salida estándar (a pantalla) se realiza como ya hemos visto con la orden `print`. Para impresiones sencillas nos bastará con concatenar adecuadamente cadenas de caracteres:

```
x = 3; y = 4
print "El valor de 'x' es " + str(x) + " y el de 'y' es "
      + str(y)
```

```
El valor de 'x' es 3 y el de 'y' es 4
```

donde es preciso convertir el entero a *string* con la función `str`. Algunos métodos para los *strings*, como por ejemplo `ljust` o `rjust` permiten mejorar la salida:

```
for x in range(5):
    print str(x**2).ljust(2), str(x**3).rjust(3)
```

```
0    0
1    1
4    8
9   27
16  64
```

Sin embargo, el método `format` permite un mejor control de la impresión:

```
for x in range(1,6):
    print '{0:2d}   {1:3d}   {2:4d}'.format(x, x**2, x**3)
```

```
1     1     1
2     4     8
3     9    27
4    16    64
5    25   125
```

En este caso, el contenido de las llaves hace referencia a los elementos del método `format` que le sigue, de manera que los números que van antes de los dos puntos corresponden a los índices del objeto que sigue a `format` y lo que va después de los dos puntos define el modo en el que se imprimirán dichos objetos. Aquí van unos ejemplos:

```
print '{0}{1}{0}'.format('abra', 'cad')
```

```
abracadabra
```

```
print 'Coordenadas: {lat}, {long}'.format(lat='38.56N',
    long='-3.28W')
```

```
Coordenadas: 38.56N, -3.28W
```

Con ciertos caracteres podemos controlar aun más la impresión:

```
print '{:<30}'.format('alineación a izquierda')
```

```
alineación a izquierda
```

```
print '{:>30}'.format('alineación a derecha')
```

```
alineación a derecha
```

```
print '{:^30}'.format('alineación centrada')
```

alineación centrada

```
print '{:*^40}'.format('alineación centrada con relleno')
```

```
*****alineación centrada con relleno*****
```

```
print '{0:d} {0:2.3f} {0:3.2e}'.format(2)
```

```
2 2.000 2.00e+00
```

### 3.2.2 Lectura y escritura en archivos

La orden `open` crea un objeto tipo archivo que nos permitirá la lectura y/o escritura en el mismo. La sintaxis de la orden es

```
f = open('file.txt', 'w') # abrimos 'file.txt' para
    escritura
```

donde el primer argumento es el nombre del archivo y el segundo el modo de acceso. Los modos son `r` para sólo lectura, `r+` para lectura y escritura, `w` sólo para escritura (si el archivo existe, lo sobrescribirá) y `a` para escritura agregando contenido al archivo existente (o creando uno nuevo si no existe).

Si por ejemplo, disponemos de un archivo `archivo.py` cuyo contenido es

```
# coding: utf-8
print "Primera línea"
print "Segunda línea"
```

entonces podemos abrirlo y leerlo con:

```
f = open('archivo.py', 'r')
print f.read()
```

```
# coding: utf-8
print "Primera línea"
print "Segunda línea"
```

El método `read` del objeto archivo lee el archivo al que referencia el objeto creado. Si se le proporciona un argumento numérico entonces lee un número determinado de *bytes* del archivo;<sup>1</sup> cuando es llamado sin argumento, lee el archivo al completo. Cada llamada a `read` mueve el puntero de lectura del archivo la último lugar leído, de manera que la llamada siguiente lee a partir de ahí. Por eso:

```
f.read()
```

<sup>1</sup>Un *byte* puede considerarse un carácter, aunque hay caracteres que ocupan más de uno.

```
''
```

produce una cadena vacía, pues el puntero se halla al final del archivo.

Podemos usar el método `seek` para situarnos en cualquier posición del archivo.

```
f.seek(0) # Vamos al inicio del archivo
f.read()
```

```
'# coding: utf-8\nprint "Primera l\xc3\xadnea"\nprint "
    Segunda l\xc3\xadnea"'
```

Nótese la diferencia entre la orden y la impresión: la cadena que contiene el archivo es leída en formato de representación interna, que difiere de la representación que se muestra con la orden `print`.

También es posible leer línea a línea con `readline`

```
f.seek(0)
f.readline()
```

```
'# coding: utf-8\n'
```

```
print f.readline()
```

```
print "Primera línea"
```

```
f.readline()
```

```
'print "Segunda l\xc3\xadnea"'
```

o con `f.readlines()` que almacena cada una de las líneas del archivo en una lista:

```
f.seek(0)
for x in f.readlines():
    print x,
```

```
# coding: utf-8
print "Primera línea"
print "Segunda línea"
```

Otra opción aún más cómoda es iterar sobre el propio objeto:

```
f.seek(0)
for x in f:
    print x,
```

```
# coding: utf-8
print "Primera línea"
print "Segunda línea"
```

Para cerrar un archivo usaremos el método `close`:

```
f.close() # cerramos el archivo vinculado al objeto f
```

La escritura en un archivo se lleva a cabo con el método `write`. Para ello es necesario que el archivo se abra en modo escritura (o lectura/escritura):

```
f = open('hola.py', 'w') # sólo escritura (nuevo archivo)
for x in ['primera', 'segunda', 'tercera']:
    f.write(x + '\n')
```

```
f.read()
```

```
IOError: File not open for reading
```

Al haber abierto el archivo en modo sólo escritura, no podemos leer. Será preciso volver a abrir:

```
f.close()
f = open('hola.py', 'r+') # lectura y escritura
print f.read()
```

```
primera
segunda
tercera
```

```
f.write('cuarta\n')
f.seek(0)
f.readlines()
```

```
['primera\n', 'segunda\n', 'tercera\n', 'cuarta\n']
```

Podemos escribir en cualquier punto del archivo si nos situamos correctamente con el método `seek`, el cual permite usar además un parámetro adicional que nos indica desde dónde nos movemos (0 desde el inicio –por defecto si se omite–, 1 desde la última referencia, 2 desde el final):

```
f.seek(8) # nos colocamos en la posición 8
f.read(7) # leemos 7 bytes
```

```
'segunda'
```

```
f.seek(9,1) # posición 9 desde la última referencia
f.read(6)
```

```
'cuarta'
```

```
f.seek(-7,2) # posición -7 desde el final
f.read(6)
```

```
'cuarta'
```

También disponemos del método `tell` que nos indica con un entero la posición en la que se encuentra el puntero del archivo.

La escritura con `write` también se puede hacer en pantalla si la usamos como un objeto tipo archivo con el atributo `stdout` del módulo `sys`:

```
import sys
g = sys.stdout
g.write('Hola mundo\n')
```

```
Hola mundo
```

lo que puede ser útil para programar las salidas por pantalla durante la elaboración de código con `write`, de modo que cuando queramos direccionar la salida a un archivo sólo necesitamos cambiar la salida.

### 3 3

## MÁS SOBRE ESTRUCTURAS DE DATOS

### 3 3 1

### Listas por comprensión

El lenguaje Python se adapta muy bien a la creación de listas construidas a partir de otras mediante algún mecanismo que las modifique. Veamos algunos ejemplos:

```
a = range(5) # lista inicial
[x**2 for x in a] # cuadrados de los elementos de a
```

```
[0, 1, 4, 9, 16]
```

```
[[x**2,x**3] for x in a]
```

```
[[0, 0], [1, 1], [4, 8], [9, 27], [16, 64]]
```

```
a = range(10)
[x**3 for x in a if x%2 == 0] # cubo de los múltiplos de 2
```

```
[0, 8, 64, 216, 512]
```

```
a = [x for x in range(1,7) if x%2==0]
print a
```

```
[2, 4, 6]
```

```
b = [x for x in range(1,7) if (x+1)%2==0]
print b
```

```
[1, 3, 5]
```

```
[x*y for x in a for y in b] # recorremos dos listas
```

```
[2, 6, 10, 4, 12, 20, 6, 18, 30]
```

```
[a[i]*b[i] for i in range(len(a))]
```

```
[2, 12, 30]
```

El último ejemplo se puede construir de forma más *pythonic*<sup>2</sup> con la orden `zip` que permite iterar sobre dos o más secuencias al mismo tiempo:

```
[x*y for x,y in zip(a,b)]
```

```
[2, 12, 30]
```

En una lista, podemos obtener simultáneamente el índice de iteración y el elemento de la lista con la orden `enumerate`. El lector podrá imaginar el resultado del siguiente ejemplo:

```
a = range(10); a.reverse()
[x*y for x,y in enumerate(a)]
```

En realidad `enumerate` es lo que se denomina un *iterador*, es decir, un objeto sobre el que se pueden recorrer sus elementos. Otro iterador es `reversed` que como parece lógico, crea un iterador con el orden inverso. Así, el último ejemplo equivale a

```
[x*y for x,y in enumerate(reversed(range(10)))]
```

---

<sup>2</sup>En el mundo de la programación en Python se suele usar este adjetivo para referirse a una forma de escribir código que usa las características propias y peculiares del lenguaje Python que auna sencillez, brevedad, elegancia y potencia.



### 3.3.2 Copia y mutabilidad de objetos

Hemos mencionado anteriormente que las tuplas y las cadenas de caracteres son objetos inmutables, mientras que las listas son mutables. Debemos añadir también que los diferentes tipos de números son inmutables y los diccionarios son mutables. Ahora bien, ¿qué significa exactamente que los números sean inmutables? ¿Quiere decir que no los podemos modificar?

En realidad estas cuestiones están relacionadas con el modo en el que Python usa las variables. A diferencia de otros lenguajes, en los que una variable esencialmente referencia una posición en memoria, cuyo contenido podemos modificar, en Python, una variable en realidad no es más que una referencia al objeto, no contiene su valor, sino una referencia a él. Lo podemos ver con un sencillo ejemplo; la orden `id` nos muestra el identificador de un objeto, que básicamente es la dirección de memoria en la que se encuentra. Si escribimos:

```
x = 5 # x apunta al objeto 5
id(x)
```

```
160714880
```

lo primero que hace Python es crear el objeto 5, al que le asigna la variable `x`. Ahora entenderemos por qué ocurre lo siguiente:

```
y = 5 # y apunta al objeto 5
id(y)
```

```
160714880
```

No hemos creado un nuevo objeto, sino una nueva referencia para el mismo objeto, por lo que tienen el mismo identificador. Lo podemos comprobar con la orden `is`:

```
x is y
```

```
True
```

¿Qué ocurre si alteramos una de las variables?

```
x = x + 2 # x apunta al objeto 7
id(x)
```

```
160714856
```

vemos que el identificador cambia. Python evalúa la expresión de la derecha, que crea un nuevo objeto, y a continuación asigna la variable `x` al nuevo objeto, por eso ésta cambia de identificador. Obsérvese que:

```
id(7)
```

```
160714856
```

y ahora:

```
x is y
```

```
False
```

Con las listas pasa algo similar, salvo que ahora está permitido modificar el objeto (porque son mutables), por lo que las referencias hechas al objeto continuarán apuntado a él:

```
a = [1,2]
id(a) # identificador del objeto
```

```
3074955372L
```

```
a[0] = 3 # modificamos el primer elemento
print a
```

```
[3, 2]
```

```
id(a) # el identificador no cambia
```

```
3074955372L
```

La L del identificador se refiere al tipo de entero que devuelve (`long`).

La consecuencia de que las listas sean mutables se puede ver en el siguiente ejemplo:

```
x = range(3)
y = x # y referencia lo mismo que x
x[:0] = x # modificamos x
print y # y también se modifica
```

```
[0, 1, 2, 0, 1, 2]
```

Sin embargo,

```
x = x+[3] # nuevo objeto
y
```

```
[0, 1, 2, 0, 1, 2]
```

¿Por qué no se ha modificado y en este caso? La respuesta es que se ha creado un nuevo objeto, al que se referencia con x, por lo que x ya no apunta al objeto

anterior, pero sí y. ¿Cómo podemos entonces copiar una lista? Podemos para ello usar el *slicing*:

```
x = range(10)
y = x[:] # copiamos x a y
print id(x), id(y)
```

```
3074955372L 3075028972L
```

### 3 4

## EXCEPCIONES

Para evitar los errores en el momento de la ejecución Python dispone de las secuencias de control **try-except**, que hacen que, en caso de que se produzca una excepción, ésta pueda ser tratada de manera que la ejecución del programa no se vea interrumpida. Por ejemplo, supongamos que tenemos la siguiente situación, que obviamente, producirá una excepción:

```
for a in [1,0]:
    b = 1/a
    print b
```

```
1
ZeroDivisionError: integer division or modulo by zero
```

Para evitarla, hacemos uso de las órdenes **try** y **except** del siguiente modo:

```
for a in [1,0]:
    try:
        b = 1/a
    except:
        print "Se ha producido un error"
        b = 0
    print b
```

```
1
Se ha producido un error
0
```

El funcionamiento de estas sentencias es simple: el fragmento de código dentro de la orden **try** trata de ejecutarse; si se produce una excepción, se salta al fragmento de código correspondiente a la orden **except**, y éste se ejecuta. De este modo, el programa no se ve interrumpido por el error.

La orden **except** admite mayor sofisticación, pues permite tratar cada excepción en función de su tipo. Podemos verlo en el siguiente ejemplo:

```

for a in [0, '1']:
    try:
        b = 1/a
    except ZeroDivisionError:
        print "División por cero"
        b = 0
    except TypeError:
        print "División inválida"
        b = 1
    print b

```

```

División por cero
0
División inválida
1

```

### 3 5

## EJERCICIOS

**E.1** Calcula  $\sum_{i=1}^{100} \frac{1}{\sqrt{i}}$

**E.2** Calcula la suma de los números pares entre 1 y 100.

**E.3** Calcular la suma de todos los múltiplos de 3 o 5 que son menores que 1000. Por ejemplo, la suma de todos los múltiplos de 3 o 5 que son menores que 10 es:  $3 + 5 + 6 + 9 = 23$

**E.4** Utiliza la función `randint` del módulo `random` para crear una lista de 100 números aleatorios entre 1 y 10.

**E.5** El Documento Nacional de Identidad en España es un número finalizado con una letra, la cual se obtiene al calcular el resto de la división de dicho número entre 23. La letra asignada al número se calcula 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

Escribe un programa que lea el número por teclado y muestre la letra asignada. Por ejemplo, al número *28594978* le debe corresponder la letra *K*. Tres líneas de código debería ser suficientes.

**E.6** Usando un bucle `while` escribe un programa que pregunte por un número por pantalla hasta que sea introducido un número impar múltiplo de 3 y de 7 que sea mayor que 100.

**E.7** Crea una lista aleatoria de números enteros (usar Ejercicio E.4) y luego escribe un programa para que elimine los valores impares de dicha lista.

**E.8** Escribir una función `histograma` que admita una lista aleatoria de números enteros entre 1 y 5 (usar Ejercicio E.4) y elabore un diagrama de frecuencias de aparición de cada uno de los elementos de la lista según el formato que sigue: por ejemplo, si la lista es

[1, 4, 2, 1, 4, 4, 5, 5, 5, 3]

entonces la función debe imprimir:

```
1 **
2 *
3 *
4 ***
5 ***
```

**E.9** Escribir una función `menorymayor` que acepte un conjunto indeterminado de números y obtenga el mayor y el menor de entre todos ellos. Es decir, la función debería hacer lo siguiente:

```
menorymayor(2,5,1,3,6,2)
```

(1, 6)

```
menorymayor(3,1,10,9,9,7,5,4)
```

(1, 10)

**E.10** Escribir una función que acepte como parámetros dos valores  $a$  y  $b$ , y una función  $f$  tal que  $f(a)f(b) < 0$  y que, mediante el método de bisección, devuelva un cero de  $f$ . La aproximación buscada será controlada por un parámetro  $\varepsilon$  que por defecto ha de ser  $10^{-4}$ .

**E.11** Considera la siguiente función:

```
def media(*valores):
    if len(valores) == 0: # len = longitud de la tupla
        return 0
    else:
        sum = 0
        for x in valores:
            sum += x
        return sum / len(valores)
```

Explica por qué se obtiene el siguiente resultado:

```
media(*range(1,5))
```

0

y corrígelo para que que proporcione el resultado correcto (2.5).

**E.12** Considera la función cuyo encabezado es el siguiente:

```
def varios(param1, param2, *otros, **mas):
```

El funcionamiento de la función es como sigue:

```
varios(1, 2)
```

```
1
2
```

```
varios(1, 2, 4, 3)
```

```
1
2
****
***
```

```
varios(3,4,a=1,b=3)
```

```
3
4
—
—
```

¿Cuál será el resultado de la siguiente llamada?

```
varios(2,5,1,a=2,b=3)
```

Escribe el código de la función.

**E.13** Crea un fichero de nombre `hist.py` que incluya la función `histograma` del Ejercicio E.8. Luego crea un nuevo archivo Python de manera que acepte un número  $n$  como entrada, cree una lista aleatoria de números enteros entre 1 y 5 de longitud  $n$ , y llame a la función `histograma` para imprimir el recuento de cifras.

# 4

## NumPy

---

NumPy que es el acrónimo de *Numeric Python* es un módulo fundamental para el cálculo científico con Python. Con él se dispone de herramientas computacionales para manejar estructuras con una gran cantidad de datos, diseñadas para obtener un buen nivel de rendimiento en su manejo

Hemos de tener en cuenta que en un lenguaje de programación interpretado como lo es Python, el acceso secuencial a estructuras que contengan una gran cantidad de datos afecta mucho al rendimiento de los programas. Por ello es imperativo evitar, en la medida de lo posible, el usar bucles para recorrer uno a uno los elementos de una estructura de datos. En ese sentido, el módulo NumPy incorpora un nuevo tipo de dato, el *array*, similar a una lista, pero que es computacionalmente mucho más eficiente. El módulo incorpora además una gran cantidad de métodos que permiten manipular los elementos del *array* de forma no secuencial, lo que se denomina *vectorización*, y que ofrece un alto grado de rendimiento. Es por ello que los algoritmos programados en el módulo SciPy, que veremos en el capítulo siguiente, están basados en el uso extensivo de *arrays* de NumPy.

---

### 4.1

#### ARRAYS

A diferencia de las listas, cuyos elementos pueden ser de cualquier tipo, un *array* de NumPy debe tener todos sus elementos del mismo tipo. Para definirlo será necesario importar el módulo, y usar la orden **array**:

```
import numpy as np
a = np.array([1.,3,6])
print a
```

```
[ 1.  3.  6.]
```

Si aparecen elementos de distinto tipo, como es el caso, el *array* es definido según la jerarquía de tipos numéricos existente. Como vemos, hemos definido un *array* con un real y dos enteros, y éste se ha considerado como real. El atributo `dtype` nos lo confirma

```
type(a)
print a.dtype
```

```
<type 'numpy.ndarray'>
float64
```

Nótese que en NumPy, el tipo de datos numéricos es un poco más extenso, incluyendo reales en doble precisión (`float64`), simple precisión (`float32`), entre otros. Por defecto, el tipo `float` de Python, corresponde a `float64`. Es importante tener en cuenta el tipo con el que se define un *array*, pues pueden ocurrir errores no deseados:

```
a = np.array([1,3,5])
print a.dtype
```

```
int64
```

```
a[0] = 2.3 # La modificación no es la esperada
a
```

```
array([2, 3, 5])
```

Como el *array* es entero, todos los datos son convertidos a ese tipo. No obstante, podemos cambiar de tipo todo el *array*,

```
a = a.astype(float) # cambio de tipo en nuevo array
```

```
array([ 2.,  3.,  5.])
```

```
a[0] = 3.6
print a
```

```
[ 3.6  3.  6. ]
```

Para evitarnos el tener que cambiar de tipo lo más sencillo es definir desde el principio el *array* usando elementos del tipo deseado, o indicándolo expresamente como en el siguiente ejemplo:

```
a = np.array([2,5,7],float)
a.dtype
```

```
dtype('float64')
```



Los *arrays* también pueden ser multidimensionales:

```
a = np.array([[1.,3,5],[2,1,8]])
print a
```

```
[[ 1.  3.  5.]
 [ 2.  1.  8.]]
```

y el acceso a los elementos puede hacerse con el doble corchete, o el doble índice:

```
print a[0][2], a[0,2]
```

```
5.0 5.0
```

Tenemos diversos atributos y funciones para obtener información relevante del *array*:

```
a = np.array([[1,3,5],[2,8,7]])
a.ndim # número de ejes
```

```
2
```

```
a.shape # dimensiones de los ejes
```

```
(2, 3)
```

```
a.size # número total de elementos
```

```
6
```

```
len(a) # longitud de la primera dimensión
```

```
2
```

Finalmente, la orden `in` nos permite saber si un elemento es parte o no de un *array*:

```
2 in a # el elemento 2 está en a
```

```
True
```

```
4 in a # el elemento 4 no está en a
```

```
False
```

## 4.2

## FUNCIONES PARA CREAR Y MODIFICAR ARRAYS

La orden `arange` es similar a `range`, pero crea un *array*:

```
np.arange(10) # por defecto comienza en 0
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(5,15) # nunca incluye el último valor
```

```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
np.arange(2,10,3) # podemos especificar un paso
```

```
array([2, 5, 8])
```

Nótese que todos los *arrays* creados son enteros. No obstante, esta orden también admite parámetros reales

```
np.arange(1,3,0.1)
```

```
array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,
        1.8,  1.9,  2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,
        2.7,  2.8,  2.9])
```

útiles para dividir un intervalo en subintervalos de igual longitud. Nótese que el final no se incluye.

Si queremos dividir un intervalo con un número preciso de subdivisiones, tenemos la orden `linspace`:

```
np.linspace(0,1,10) # intervalo (0,1) en 10 puntos
```

```
array([ 0.          ,  0.11111111,  0.22222222,
        0.33333333,  0.44444444,  0.55555556,  0.66666667,
        0.77777778,  0.88888889,  1.          ])
```

```
np.linspace(0,1,10,endpoint=False) # sin extremo final
```

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,
        0.8,  0.9])
```

El parámetro opcional `retstep` nos devuelve además la amplitud de cada subintervalo:

```
np.linspace(1,2,5,retstep=True)
```

```
(array([ 1.   ,  1.25,  1.5   ,  1.75,  2.   ]), 0.25)
```

#### 4.2.1 Modificación de forma

Con el método `reshape` podemos cambiar la forma de un *array*:

```
np.arange(15).reshape(3,5) # cambio de forma
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

o también,

```
np.arange(15).reshape(3,5,order='F')
```

```
array([[ 0,  3,  6,  9, 12],
       [ 1,  4,  7, 10, 13],
       [ 2,  5,  8, 11, 14]])
```

La opción por defecto equivale a `order='C'`. Las iniciales 'C' y 'F' corresponden a los lenguajes C y FORTRAN, respectivamente, y se refieren al modo en el que estos lenguajes almacenan los datos multidimensionales.

El método `transpose`, que se puede abreviar como `.T`, en *arrays* bidimensionales es equivalente a la transposición de matrices:

```
a = np.arange(10).reshape(2,5)
print a.T
```

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

y con `flatten` obtenemos *arrays* unidimensionales:

```
print a
```

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

```
print a.flatten() # hacemos 1D for filas
```

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

```
print a.flatten(order='F') # por columnas (estilo FORTRAN)
```

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

Nótese que todas estas funciones y métodos crean nuevos objetos, es decir, no modifican los existentes.

#### 4.2.2 Arrays con estructura

Las siguientes funciones son útiles para crear *arrays* de un tamaño dado (y reservar memoria en ese momento):

```
a = np.zeros(5) # array 1D de 5 elementos
b = np.zeros((2,4)) # array 2D, 2 filas 4 columnas
c = np.ones(b.shape) # array 2D con unos
print a
print b
print c
```

```
[ 0.  0.  0.  0.  0.]
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

O bien

```
d = np.zeros_like(b) # d es como b
e = np.ones_like(c) # e es como c
```

La matriz identidad se obtiene con la función `eye`:

```
print np.eye(3) # matriz identidad de orden 3
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

que admite diferentes llamadas:

```
print np.eye(2,4)
```

```
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]]
```

o también

```
print np.eye(4,3,1)
print np.eye(4,3,-1)
```

```
[[ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

La función `diag` extrae la diagonal de un array 2D

```
print np.diag(np.eye(3)) # extracción de la diagonal
```

```
[ 1.  1.  1.]
```

o crea una matriz cuya diagonal es un *array* 1D dado:

```
print np.diag(np.arange(1,4)) # matriz con diagonal dada
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

```
print np.diag(np.arange(1,3),-1) # la diagonal se puede
    mover
```

```
[[0 0 0]
 [1 0 0]
 [0 2 0]]
```

```
print np.diag(np.arange(2,4),2) # más de un lugar
```

```
[[0 0 2 0]
 [0 0 0 3]
 [0 0 0 0]
 [0 0 0 0]]
```

Además, el módulo NumPy posee un submódulo de generación de *arrays* aleatorios usando distintas distribuciones. Por ejemplo,

```
print np.random.random((2,3)) # array aleatorio entre 0 y
    1
```

```
[[ 0.39677787  0.31882267  0.99927024]
 [ 0.22642506  0.3584889   0.73443862]]
```

```
print np.random.randint(1,10,(2,2)) # array 2x2 de enteros
entre 1 y 9
```

```
[[1 3]
 [8 4]]
```

#### 4.3

### SLICING

Al igual que con las listas, podemos acceder a los elementos de un *array* mediante *slicing*, que en NumPy además admite más sofisticación. Por ejemplo, la asignación sobre un trozo de una lista funciona de diferente modo que en los *arrays*:

```
a = range(5)
a[1:4] = [6] # eliminamos elementos en la lista
print a
```

```
[0, 6, 4]
```

```
b = np.arange(5)
b[1:4] = 6 # no son necesarios los corchetes
print b # cambian todos los elementos del slice
```

```
[0 6 6 6 4]
```

En los *arrays* multidimensionales, podemos acceder a alguna de sus dimensiones de diferentes formas:

```
a = np.arange(9).reshape(3,3)
print a
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

```
print a[1] # segunda fila
```

```
[3 4 5]
```

```
print a[2,:] # tercera fila
```

```
[6 7 8]
```

```
print a[:,2] # segunda columna
```

```
[2 5 8]
```

Hay una diferencia notable entre el *slicing* en una lista y en un *array* de NumPy. En el primero, el *slicing* crea un nuevo objeto mientras que en el segundo no. Recordemos qué ocurría con las listas:

```
a = range(5)
b = a # b es el mismo objeto
c = a[:] # c es un nuevo objeto
a[0] = -1 # modificamos a
b # se modifica
```

```
[-1, 1, 2, 3, 4]
```

```
c # no se modifica
```

```
[0, 1, 2, 3, 4]
```

sin embargo, con los *arrays*:

```
a = np.arange(5)
b = a # b es el mismo objeto
c = a[:] # c también
c[0] = -1 # modificamos c
b[1] = 10 # modificamos b
print a # todos quedan modificados
```

```
[-1 10  2  3  4]
```

Atención, no es necesario que el *slice* corresponda a todo el objeto:

```
a = np.arange(9).reshape(3,3)
b = a[1:,1:]
b[0,0] = -1 # modificamos un elemento del slice
print a # se modifica
```

```
[[ 0  1  2]
 [ 3 -1  5]
 [ 6  7  8]]
```

¿Cómo podemos entonces copiar un *array*? Para ello disponemos del método `copy`

```
a = np.arange(9).reshape(3,3)
b = a[1:,1:].copy() # copia en nuevo objeto
b[0,0] = -1 # modificamos b
print b
```

```
[[-1  5]
 [ 7  8]]
```

```
print a # no se ha modificado
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

## 4 4

### OPERACIONES

Cuando usamos operadores matemáticos con *arrays*, las operaciones son llevadas a cabo elemento a elemento:

```
a = np.arange(5.); b = np.arange(10.,15)
print a, b
```

```
[ 0.  1.  2.  3.  4.] [ 10.  11.  12.  13.  14.]
```

```
print a+b
```

```
[ 10.  12.  14.  16.  18.]
```

```
print a*b
```

```
[ 0.  11.  24.  39.  56.]
```

```
print a/b
```

```
[ 0.          0.09090909  0.16666667  0.23076923
  0.28571429]
```

```
print b**a
```

```
[ 0.00000000e+00  1.00000000e+00  4.09600000e+03
 1.59432300e+06  2.68435456e+08]
```

Para *arrays* bidimensionales, la multiplicación sigue siendo elemento a elemento, y no corresponde a la multiplicación de matrices. Para ello usamos



la función `dot`.

```
a = np.arange(1.,5).reshape(2,2)
b = np.arange(5.,9).reshape(2,2)
print a
print b
```

```
[[ 1.  2.]
 [ 3.  4.]]
[[ 5.  6.]
 [ 7.  8.]]
```

```
print a*b # producto elemento a elemento
```

```
[[ 5. 12.]
 [21. 32.]]
```

```
print np.dot(a,b) # producto matricial
```

```
[[ 19. 22.]
 [ 43. 50.]]
```

El cálculo con *arrays* es muy flexible, permitiéndonos cierta relajación a la hora de escribir las operaciones: por ejemplo, si empleamos funciones matemáticas de NumPy sobre un *array*, éstas se llevan a cabo elemento a elemento:

```
print np.sin(a)
```

```
[[ 0.84147098  0.90929743]
 [ 0.14112001 -0.7568025 ]]
```

```
print np.exp(b)
```

```
[[ 148.4131591   403.42879349]
 [1096.63315843 2980.95798704]]
```

Nótese que `sin` y `exp` son funciones matemáticas definidas en el módulo NumPy (entre otras muchas). Sin embargo, usar la función seno del módulo `math`:

```
import math
math.sin(a)
```

```
TypeError: only length-1 arrays can be converted to
Python scalars
```

no funciona puesto que no admite *arrays*.

No obstante, es posible *vectorizar* cualquier función que definamos con la función `vectorize`. Obsérvese el siguiente ejemplo: si tenemos nuestra propia función

```
import math
def myfunc(a):
    return math.sin(a)
```

entonces, podemos usarla para trabajar sobre *arrays* del siguiente modo:

```
vfunc = np.vectorize(myfunc)
a = np.arange(4)
print vfunc(a)
```

```
[ 0.          0.84147098  0.90929743  0.14112001]
```

### Arrays 1D vs. matrices

Hay una sutil diferencia entre un *array* unidimensional y una matriz fila o columna. Obsérvese los siguiente ejemplos:

```
a = np.arange(3)
print a
```

```
[0 1 2]
```

```
print a.T
```

```
[0 1 2]
```

```
print a.shape
```

```
(3,)
```

Como podemos ver, la operación de transposición sobre un *array* unidimensional no tiene sentido. De este modo, los productos matriciales  $a^T a$  y  $aa^T$  son indistinguibles para Python:

```
np.dot(a.T, a)
```

```
5
```

```
np.dot(a, a.T)
```

```
5
```

de hecho, se obtiene lo mismo sin necesidad de usar la transposición:

```
np.dot(a, a)
```

5

Para evitar este comportamiento, el truco está en convertir el arreglo unidimensional en uno bidimensional, en el que una de las dimensiones es uno. Se puede hacer con el método `reshape`:

```
b = a.reshape(1,3)
print np.dot(b.T, b)
```

```
[[0 0 0]
 [0 1 2]
 [0 2 4]]
```

```
print np.dot(b, b.T)
```

5

o bien, de una forma un poco más sofisticada (pero más útil, pues no es necesario conocer las dimensiones precisas para usar `reshape`), con la orden `newaxis`, que crea una nueva dimensión en el *array*:

```
c = a[np.newaxis]
print np.dot(c.T, c)
```

```
[[0 0 0]
 [0 1 2]
 [0 2 4]]
```

Obsérvese que

```
a[np.newaxis].shape
```

(1, 3)

No obstante, existe la función `outer` para el producto tensorial

```
print np.outer(a, a)
```

```
[[0 0 0]
 [0 1 2]
 [0 2 4]]
```

así como `inner` y `cross`

```
print np.inner(a, a)
```

5

```
print np.cross(a,a)
```

```
[0 0 0]
```

para el producto interior y el producto vectorial.

#### 4.4.1 Álgebra Lineal

El módulo NumPy posee un submódulo, `linalg` para realizar operaciones matemáticas comunes en Álgebra Lineal, como el cálculo de determinantes, inversas, autovalores y solución de sistemas. Vemos unos ejemplos a continuación:

```
a = np.random.randint(0,5,(3,3)).astype('float')
print a
```

```
[[ 1.  4.  4.]
 [ 0.  0.  2.]
 [ 2.  0.  0.]
```

```
print np.linalg.det(a) # determinante
```

```
16.0
```

El cálculo de autovalores y autovectores se obtiene con la función `eig`

```
val, vec = np.linalg.eig(a) # autovalores y autovectores
print val
print vec
```

```
[ 4.0+0.j          -1.5+1.32287566j -1.5-1.32287566j]
[[-0.87287156+0.j          0.07216878-0.57282196j
  0.07216878+0.57282196j]
 [-0.21821789+0.j          0.57735027+0.j
  0.57735027-0.j          ]
 [-0.43643578+0.j          -0.43301270+0.38188131j
  -0.43301270-0.38188131j]]
```

donde hay que tener en cuenta que los autovectores corresponden a las columnas de `vec`.

Para el cálculo de la matriz inversa:

```
np.linalg.inv(a) # inversa
```

```
[[ 0.      0.      0.5   ]
 [ 0.25  -0.5   -0.125]
 [ 0.      0.5    0.    ]]
```

Y para resolver sistemas, creamos en primer lugar el segundo miembro:

```
b = np.random.randint(0,5,3).astype('float')
print b
```

```
[ 2.  3.  1.]
```

y resolvemos con `solve`

```
x = np.linalg.solve(a,b) # resolución del sistema ax=b
print x
```

```
[ 0.5  -1.125  1.5   ]
```

Comprobamos:

```
print np.dot(a,x) - b # comprobación
```

```
[ 0.  0.  0.]
```

#### 4.4.2 Concatenación

Otra de las operaciones comunes que se realizan con *arrays* es la concatenación de varios *arrays*. Funciona así:

```
a = np.arange(5)
b = np.arange(10,15)
print np.concatenate((a,b))
```

```
[ 0  1  2  3  4 10 11 12 13 14]
```

Si lo que queremos es unir los *arrays* 1D en una matriz de dos filas:

```
print np.concatenate((a[np.newaxis],b[np.newaxis]))
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]]
```

Nótese la necesidad de añadir una dimensión extra para que se consideren los *arrays* como vectores fila.

Para *arrays* multidimensionales:

```
a = np.arange(6).reshape(2,3)
b = np.arange(10,16).reshape(2,3)
print a
print b
```

```
[[0 1 2]
 [3 4 5]]
[[10 11 12]
 [13 14 15]]
```

```
print np.concatenate((a,b)) # por defecto es por filas
```

```
[[ 0  1  2]
 [ 3  4  5]
 [10 11 12]
 [13 14 15]]
```

```
print np.concatenate((a,b),axis=1) # ahora por columnas
```

```
[[ 0  1  2 10 11 12]
 [ 3  4  5 13 14 15]]
```

## 4.5

### BROADCASTING

Es evidente que cuando las dimensiones de los *arrays* no concuerdan, no podemos realizar las operaciones:

```
a = np.array([1,2,3],float)
b = np.array([2,4],float)
a+b
```

```
ValueError: operands could not be broadcast together
      with shapes (3) (2)
```

No obstante, existe cierta flexibilidad:

```
print a+1
```

```
[ 2.  3.  4.]
```

Cuando los *arrays* no concuerden en dimensiones, Python tratará de proyectarlos de manera que pueda realizarse la operación en cuestión. Este procedimiento es conocido como *broadcasting*. En el ejemplo anterior es bastante

natural lo que se hace, se suma a cada elemento del *array* el escalar. Pero el *broadcasting* en Python admite más sofisticación:

```
a = np.arange(1.,7).reshape(3,2)
b = np.array([-1.,3])
print a
print b
```

```
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]]
[-1.  3.]
```

```
print a+b
```

```
[[ 0.  5.]
 [ 2.  7.]
 [ 4.  9.]]
```

Python automáticamente proyecta el *array* **b**, repitiéndolo las veces necesarias para que concuerde con la dimensión de **a**, es decir, sumamos a cada fila de **a**, el correspondiente **b**.

En ocasiones, puede haber cierta ambigüedad en el *broadcasting*:

```
a = np.arange(4.).reshape(2,2)
b = np.arange(1.,3)
print a
print b
```

```
[[ 0.  1.]
 [ 2.  3.]]
[ 1.  2.]
```

```
print a+b # por defecto, el broadcasting es por filas
```

```
[[ 1.  3.]
 [ 3.  5.]]
```

pero, ¿cómo podemos hacer el *broadcasting* por columnas? La solución es poner **b** como columna usando **newaxis**

```
print a+b[np.newaxis].T
```

```
[[ 1.  2.]
 [ 4.  5.]]
```

o alternativamente,

```
print a+b[:,np.newaxis]
```

```
[[ 1.  2.]
 [ 4.  5.]]
```

Para entender mejor el funcionamiento de `newaxis` es conveniente echar un vistazo a las dimensiones de los ejes de los *arrays* anteriores:

```
print a.shape, b.shape
```

```
(2, 2) (2,)
```

Nótese que `b` es esencialmente un matriz fila (un *array* unidimensional), por lo que `a+b` es equivalente a

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 3 & 5 \end{pmatrix}$$

(se repite por filas). Mientras que,

```
print a.shape, b[:,np.newaxis].shape
```

```
(2,2) (2, 1)
```

significa que el *array* `b[:,np.newaxis]` es equivalente a una matriz columna, por lo que en la operación a realizar se repiten las columnas:

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$$

En definitiva, `newaxis` lo que hace es añadir una nueva dimensión, convirtiendo el *array* 1-dimensional en uno 2-dimensional. Si escribimos `b[np.newaxis]`, la nueva dimensión será la primera, mientras que si hacemos `b[:,np.newaxis]` el nuevo eje aparecerá en la segunda dimensión.

Veamos un ejemplo más interesante. Supongamos que tenemos dos matrices

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{pmatrix}$$

En este caso podemos calcular los productos matriciales  $AB$  y  $BA$ , con los que obtenemos matrices  $2 \times 2$  y  $3 \times 3$ , respectivamente:



```
A = np.array([i for i in range(12) if (i+1)%2==0]).reshape
(2,3)
B = np.array([i for i in range(1,13) if i%2==0]).reshape
(3,2)
print np.dot(A,B)
```

```
[[ 70  88]
 [178 232]]
```

```
print np.dot(B,A)
```

```
[[ 30  42  54]
 [ 62  90 118]
 [ 94 138 182]]
```

Estas son las operaciones “lógicas” que se puede realizar con estas dos matrices, pero podríamos pensar en alguna más. Por ejemplo, si consideramos la primera columna de  $A$  y la primera fila de  $B$ , podríamos construir la matriz  $2 \times 2$  obtenida al considerar el producto tensorial de vectores. Aunque este producto tensorial se puede llevar a cabo mediante operaciones matriciales como hicimos antes con `dot` o `outer`, el *broadcasting* también nos lo permite. Por ejemplo,

```
a = np.array([1,7])
b = np.array([2,4])
print a[:,np.newaxis]*b
```

```
[[ 2  4]
 [14 28]]
```

De este modo podemos proceder con las columnas de  $A$  y filas de  $B$ , respectivamente. Un sencillo bucle nos proporciona dicho producto:

```
for i in range(A.shape[1]):
    print A[:,i][np.newaxis].T*B[i,:]
```

```
[[ 2  4]
 [14 28]]
[[18 24]
 [54 72]]
[[ 50  60]
 [110 132]]
```

Esto es, hemos calculado

$$\begin{pmatrix} 1 \\ 7 \end{pmatrix} \begin{pmatrix} 2 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 14 & 28 \end{pmatrix}, \quad \begin{pmatrix} 3 \\ 9 \end{pmatrix} \begin{pmatrix} 6 & 8 \end{pmatrix} = \begin{pmatrix} 18 & 24 \\ 54 & 72 \end{pmatrix},$$

$$\begin{pmatrix} 5 \\ 11 \end{pmatrix} \begin{pmatrix} 10 & 12 \end{pmatrix} = \begin{pmatrix} 50 & 60 \\ 110 & 132 \end{pmatrix}$$

La ventaja que nos aporta NumPy es que es posible realizar esta misma operación sin necesidad del bucle. Nótese que queremos multiplicar tres vectores columna, por los correspondientes tres vectores filas. Esto es, vamos a realizar 3 operaciones de vectores  $2 \times 1$  con vectores  $1 \times 2$ . De este modo, debemos considerar a **A** como 3 vectores columna (lo que corresponde a tamaño (3,2,1))

```
print A[np.newaxis].T
```

```
[[[ 1]
   [ 7]]

 [[ 3]
   [ 9]]

 [[ 5]
   [11]]]
```

```
A[np.newaxis].T.shape
```

```
(3, 2, 1)
```

De igual modo, **B** debe tener tamaño (3,1,2)

```
print B[:,np.newaxis]
```

```
[[[ 2  4]

 [[ 6  8]

 [[10 12]]]
```

```
B[:,np.newaxis].shape
```

```
(3, 1, 2)
```

Así,

```
print A[np.newaxis].T*B[:,np.newaxis]
```

```
[[[ 2  4]
  [14 28]]

 [[18 24]
  [54 72]]

 [[50 60]
  [110 132]]]
```

nos da el resultado esperado. ¿Puede el lector encontrar la forma de obtener los productos tensoriales de los vectores fila de  $A$  con los correspondientes vectores columna de  $B$ ? (véase Ejercicio E.8).

## 4 6

### OTROS MÉTODOS DE INTERÉS

Mostramos a continuación algunos métodos útiles:

```
a = np.array([x**2 for x in range(15) if x%3==1])
print a
```

```
[ 1  16  49 100 169]
```

```
a.sum() # suma de todos los elementos
```

```
335
```

```
a.prod() # producto de todos los elementos
```

```
13249600
```

```
a.min() # menor elemento
```

```
1
```

```
a.max() # mayor elemento
```

```
169
```

```
a.argmax() # índice del menor elemento
```

```
0
```

```
a.argmax() # índice del mayor elemento
```

```
4
```

```
print a.clip(10,50) # si a<10 vale 10; si a>50 vale 50
```

```
[10 16 49 50 50]
```

Para *arrays* multidimensionales, la opción `axis` permite hacer las operaciones anteriores (excepto `clip`) a lo largo de ejes diferentes:

```
a = np.arange(12).reshape(3,4)
print a
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
a.sum()
```

```
66
```

```
print a.sum(axis=0)
```

```
[12 15 18 21]
```

```
print a.sum(axis=1)
```

```
[ 6 22 38]
```

#### 4.6.1 Comparaciones

También es interesante la comparación entre *arrays*:

```
a = np.array([1.,3,0]); b=np.array([0,3,2.])
a < b
```

```
array([False, False,  True], dtype=bool)
```

```
a == b
```

```
array([False,  True, False], dtype=bool)
```

Obsérvese que se crea un nuevo *array*, de tipo `bool` (booleano), correspondiente a la comparación de elemento con elemento. El *broadcasting* también puede usarse en este contexto:

```
a > 2
```

```
array([False,  True, False], dtype=bool)
```

Disponemos también de métodos sobre los *arrays* booleanos para determinar si son ciertos todos los elementos:

```
c = a < 3 # asignamos a un array
print c
```

```
[ True False  True]
```

```
c.any() # hay alguno cierto?
```

```
True
```

```
c.all() # son todos ciertos?
```

```
False
```

Aunque la comparación  $2 < x < 3$  es correcta si  $x$  es un número, no es válida para *arrays*. Para comparaciones más sofisticadas disponemos de las funciones `logical_and`, `logical_or` y `logical_not`:

```
print a
```

```
[ 1.  3.  0.]
```

```
np.logical_and(2 < a, a < 3)
```

```
array([False, False, False], dtype=bool)
```

```
np.logical_or(2 < a, a < 1)
```

```
array([False,  True,  True], dtype=bool)
```

También existe la función `where` que crea un nuevo *array* a partir de una comparación:

```
a = np.array([ 2.,  3.,  0.,  5.])
print np.where(a != 0., 1/a, -1)
```

```
[ 0.5          0.33333333 -1.          0.2          ]
```

Como podemos ver, si el elemento del *array* no es cero, considera su inverso, en caso contrario, lo hace igual a -1. Si usamos esta función sin los dos últimos argumentos, nos proporciona los índices en los que la condición es cierta:

```
np.where(a!=0)
```

```
(array([0, 1, 3]),)
```

Aun no hemos mencionado un par de constantes especiales disponibles con el módulo NumPy, `nan` y `inf`, que representan los valores *no es un número* e *infinito*, respectivamente:

```
a = np.array([-1.,0,1])
print 1/a
```

```
[ -1.  inf   1.]
```

```
print np.sqrt(a)
```

```
[ nan   0.   1.]
```

En NumPy tenemos un par de funciones para preguntar si estos valores están presentes:

```
print np.isinf(1/a)
print np.isnan(np.sqrt(a))
```

```
[False  True False]
[ True False False]
```

## 4.7

### INDEXACIÓN SOFISTICADA

Además del *slicing* que hemos visto, los *arrays* de NumPy se pueden indexar a través de *arrays* con enteros o variables booleanas, denominados *máscaras*. Veamos algunos ejemplos:

```
a = np.random.randint(1,20,6) # enteros aleatorios
print a
```

```
[11  9 12 15 11  7]
```

```
mask = (a%3 == 0) # múltiplos de 3
print mask
```

```
[False  True  True  True False False]
```

```
print a[mask] # extracción de elementos de a con mask
```

```
[ 9 12 15]
```

```
print np.where(mask) # índices de elementos extraídos
```

```
(array([1, 2, 3]),)
```

```
a[mask] = -1 # asignación sobre la máscara
print a
```

```
[11 -1 -1 -1 11 7]
```

La máscara también puede estar formada por enteros:

```
mask = np.random.randint(0,5,5)
print mask # máscara de índices enteros
```

```
[2 4 3 2 0]
```

```
a = np.random.random(7)
print a # array aleatorio
```

```
[ 0.72321035  0.36614      0.76326975  0.11787665
  0.41174471  0.39349619
  0.08833142]
```

```
print a[mask] # selección según la máscara (nuevo objeto)
```

```
[ 0.76326975  0.41174471  0.11787665  0.76326975
  0.72321035]
```

```
a[[3,1]] = 0 # asignación a las posiciones 3 y 1
print a
```

```
[ 0.72321035  0.          0.76326975  0.
  0.41174471  0.39349619
  0.08833142]
```

Las máscaras pueden ser incluso más sofisticadas, posibilitando el cambio de forma del *array*:

```
a = np.arange(10)[::-1] # orden inverso!
print a
```

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

```
mask = np.random.randint(0,10,4).reshape(2,2)
print mask # máscara 2x2 aleatoria
```

```
[[1 1]
 [0 6]]
```

```
print a[mask] # a cambia de forma
```

```
[[8 8]
 [9 3]]
```

Sin embargo, cuando el *array* es multidimensional, hay que tener más cuidado con la indexación. Obsérvese el siguiente ejemplo:

```
a = np.arange(12).reshape(3,4)
print a
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Ahora accedemos a la tercera fila, columnas primera y cuarta

```
print a[2,[0,3]]
```

```
[ 8 11]
```

Si queremos acceder a las filas segunda y tercera, con las mismas columnas, podemos usar el *slicing*:

```
print a[1:3,[0,3]]
```

```
[[ 4  7]
 [ 8 11]]
```

Pero si intentamos hacer lo mismo con los índices explícitos:

```
print a[[1,2],[0,3]]
```

```
[ 4 11]
```

el resultado no es el esperado. La razón detrás de este comportamiento está en la forma en la que NumPy almacena y accede a los datos de un *array*, y no trataremos de explicarla aquí. Simplemente deberemos tener en cuenta que el acceso simultáneo a un *array* multidimensional con índices sin emplear *slicing* no funciona correctamente: NumPy interpreta un acceso a los elementos [1,0] y [2,3]. Para obtener lo esperado usando índices explícitamente hemos de usar la función `ix_`:



```
print a[np.ix_([1,2],[0,3])]
```

```
[[ 4  7]
 [ 8 11]]
```

## 4 8

### OPERACIONES CON *SLICING*

Un problema muy habitual en computación científica es el cálculo de derivadas de funciones discretas. Por ejemplo, si tenemos una función  $u$  aproximada por sus valores en un conjunto discreto de puntos  $x_i$ , de tal forma que  $u_i \equiv u(x_i)$ , para calcular la derivada discreta de dicha función se usan los típicos cocientes incrementales:

$$u'(x_i) \approx \frac{u_{i+1} - u_i}{x_{i+1} - x_i}$$

Si disponemos de los valores de la función  $u$  y de los puntos  $x_i$  en sendos *arrays*, un sencillo bucle nos proporcionaría el cálculo buscado:

```
x = np.linspace(0,1,10)
u = np.random.rand(10)
up = np.zeros(u.shape[0]-1) # reserva de memoria
for i in range(len(x)-1):
    up[i] = (u[i+1]-u[i])/(x[i+1]-x[i])
print up
```

```
[-1.41907443 -2.95081187  4.47554157 -0.88413512
 -7.47152493  7.69239807 -7.67930642  8.45498184
 -3.05845911]
```

Pero tal y como comentamos en la introducción de este capítulo, debemos evitar siempre que sea posible el uso de bucles para construir y/o acceder a los elementos de un *array*. En este caso, nos bastará observar, por ejemplo, que los numeradores de la derivada numérica se pueden escribir

$$(u_1 - u_0, u_2 - u_1, \dots, u_n - u_{n-1}) = (u_1, u_2, \dots, u_n) - (u_0, u_1, \dots, u_{n-1})$$

que en Python tiene la representación mediante *slicing*: `u[1:]-u[:-1]`. De este modo,

```
uz = (u[1:]-u[:-1])/(x[1:]-x[:-1])
print uz
```

```
[-1.41907443 -2.95081187  4.47554157 -0.88413512
 -7.47152493  7.69239807 -7.67930642  8.45498184
 -3.05845911]
```

## 4 9

## LECTURA DE FICHEROS

El módulo NumPy posee funciones para la lectura rápida de ficheros de datos similar a la que ofrece MATLAB. La orden `loadtxt` permite leer ficheros de datos numéricos y almacenarlos en un *array*. Por ejemplo, si el contenido del archivo `fichero.dat` es

```
2 1
4 3
6 5
8 7
10 9
```

entonces lo podemos leer de la forma:

```
d = np.loadtxt('fichero.dat')
print d
```

```
[[ 2.  1.]
 [ 4.  3.]
 [ 6.  5.]
 [ 8.  7.]
 [10.  9.]]
```

De forma similar tenemos la orden `savetxt` para guardar *arrays* de forma rápida. La siguiente orden

```
c = d**2
np.savetxt('nuevo.dat',c)
```

crea un fichero de nombre `nuevo.dat` con el siguiente contenido

```
4.0000000000000000e+00 1.0000000000000000e+00
1.6000000000000000e+01 9.0000000000000000e+00
3.6000000000000000e+01 2.5000000000000000e+01
6.4000000000000000e+01 4.9000000000000000e+01
1.0000000000000000e+02 8.1000000000000000e+01
```

No obstante es importante resaltar que estos mecanismos no son los más eficientes para manejar una gran cantidad de datos.

## 4 10

## BÚSQUEDA DE INFORMACIÓN

El módulo NumPy es enorme y contiene gran cantidad de métodos y funciones para realizar todo tipo de tareas. Para buscar la función adecuada, NumPy nos provee de algunas funciones para obtener dicha información. La función `info` tiene el mismo efecto que el uso de `?` en IPython:

```
np.info(np.dot)
```

```
dot(a, b, out=None)
```

```
Dot product of two arrays.
```

```
For 2-D arrays it is equivalent to matrix multiplication
, and for 1-D
```

```
arrays to inner product of vectors (without complex
conjugation). For
```

```
N dimensions it is a sum product over the last axis of `
a` and
```

```
the second-to-last of `b`::
```

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

```
...
```

Disponemos además de la función `source` que nos proporciona el código fuente de la función en cuestión:

```
np.source(np.diag)
```

```
In file: /usr/local/lib/python2.7/dist-packages/numpy/
lib/twodim_base.py
```

```
def diag(v, k=0):
```

```
    """
```

```
    Extract a diagonal or construct a diagonal array.
```

```
    See the more detailed documentation for ``numpy.
```

```
    diagonal`` if you use this
```

```
    function to extract a diagonal and wish to write to
    the resulting array;
```

```
    whether it returns a copy or a view depends on what
    version of numpy you
    are using.
```

```
    ...
```

```
    """
```

```
    v = asarray(v)
```

```
    s = v.shape
```

```
    if len(s) == 1:
```

```
        n = s[0]+abs(k)
```

```
        res = zeros((n, n), v.dtype)
```

```

    if k >= 0:
        i = k
    else:
        i = (-k) * n
    res[:n-k].flat[i::n+1] = v
    return res
elif len(s) == 2:
    return v.diagonal(k)
else:
    raise ValueError("Input must be 1- or 2-d.")

```

aunque es necesario advertir que puede que dicho código no esté disponible para algunas funciones.

Por último, cuando no conocemos el nombre exacto de la función, o su posible existencia, podemos realizar una búsqueda por concepto para ver qué funciones en NumPy están relacionadas con ese tema. Por ejemplo, si estamos buscando algún tipo de descomposición matricial y queremos ver qué posee NumPy, podemos usar la función `lookfor`:

```
np.lookfor('decomposition')
```

Search results for 'decomposition'

---

```

numpy.linalg.svd
    Singular Value Decomposition.
numpy.linalg.cholesky
    Cholesky decomposition.
numpy.linalg._umath_linalg.cholesky_lo
    cholesky decomposition of hermitian positive-
    definite matrices.
numpy.polyfit
    Least squares polynomial fit.
numpy.ma.polyfit
    Least squares polynomial fit.
numpy.linalg.pinv
    Compute the (Moore–Penrose) pseudo-inverse of a
    matrix.
numpy.polynomial.Hermite.fit
    Least squares fit to data.
numpy.polynomial.HermiteE.fit
    Least squares fit to data.
numpy.polynomial.Laguerre.fit
    Least squares fit to data.
numpy.polynomial.Legendre.fit
    Least squares fit to data.

```

```

numpy.polynomial.Chebyshev.fit
    Least squares fit to data.
numpy.polynomial.Polynomial.fit
    Least squares fit to data.
numpy.polynomial.hermite.hermfit
    Least squares fit of Hermite series to data.
numpy.polynomial.laguerre.lagfit
    Least squares fit of Laguerre series to data.
numpy.polynomial.legendre.legfit
    Least squares fit of Legendre series to data.
numpy.polynomial.chebyshev.chebfit
    Least squares fit of Chebyshev series to data.
numpy.polynomial.hermite_e.hermefit
    Least squares fit of Hermite series to data.
numpy.polynomial.polynomial.polyfit
    Least-squares fit of a polynomial to data.

```

**4 11****EJERCICIOS**

**E.1** Crear las siguientes matrices:

$$A = \begin{pmatrix} 10 & 13 & 16 & 19 & 22 & 25 & 28 \\ 11 & 14 & 17 & 20 & 23 & 26 & 29 \\ 12 & 15 & 18 & 21 & 24 & 27 & 30 \end{pmatrix}, \quad B = \begin{pmatrix} 12 & 18 & 24 & 30 & 36 \\ 42 & 48 & 54 & 60 & 66 \\ 72 & 78 & 84 & 90 & 96 \end{pmatrix}$$

$$C = \begin{pmatrix} 0 & 2 & 3 & 4 & 6 & 8 & 9 \\ 10 & 12 & 14 & 15 & 16 & 18 & 20 \end{pmatrix}$$

**E.2** Escribe una función tenga como argumento de entrada una matriz y devuelva 1 si es simétrica,  $-1$  si es antisimétrica y 0 en caso contrario.

**E.3** Crear una función para construir una matriz de tamaño  $n$  con una estructura como la siguiente:

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}$$

**E.4** Considera un conjunto aleatorio de 100 puntos del plano en el rectángulo

$(0, 1) \times (0, 1)$ . Conviértelos a coordenadas polares y encuentra el punto más lejano al origen en coordenadas cartesianas.

**E.5** Crea una función cuyo argumento de entrada sea una matriz (o *array* 2D) y cuya salida sean tres matrices  $D$ ,  $L$  y  $U$ , correspondientes a su parte diagonal, triangular inferior y superior, respectivamente, de manera que  $A = D + L + U$ . Asegurate de que el objeto de entrada sea efectivamente un *array* 2D cuadrado, y que en caso contrario, se imprima un mensaje de aviso.

**E.6** Construye la siguiente matriz

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 54 & 55 \end{pmatrix}$$

usando *broadcasting* de la siguiente manera

```
b=np.arange(6); c=10*b
a=b+c[:,np.newaxis]
```

Ahora, mediante *slicing*, obtén las siguientes submatrices

$$\begin{pmatrix} 13 & 14 \end{pmatrix} \quad \begin{pmatrix} 44 & 45 \\ 54 & 55 \end{pmatrix} \quad \begin{pmatrix} 12 \\ 22 \\ 32 \end{pmatrix} \quad \begin{pmatrix} 20 & 22 & 24 \\ 40 & 42 & 44 \end{pmatrix}$$

**E.7** Crea una matriz cuadrada de tamaño  $8 \times 8$  con elementos 0 y 1 dispuestos como en un tablero de ajedrez. Por ejemplo, para dimensión 4, la matriz sería:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

**E.8** Dadas las matrices

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}, \quad B = \begin{pmatrix} 8 & 4 \\ 7 & 3 \\ 6 & 2 \\ 5 & 1 \end{pmatrix}$$

Calcular las matrices resultantes de multiplicar tensorialmente los vectores fila de  $A$  con los respectivos vectores columna de  $B$ . Debes obtener 2 matrices de tamaño  $4 \times 4$ . Calcula también las 4 matrices de tamaño  $2 \times 2$  obtenidas al multiplicar tensorialmente las columnas de  $A$  con las filas de  $B$ .

**E.9** Usando *broadcasting* (ver Ejercicio E.6), obtener las matrices siguientes:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 4 & 9 & 16 \\ 4 & 9 & 16 & 25 \\ 9 & 16 & 25 & 36 \\ 16 & 25 & 36 & 49 \end{pmatrix}$$

**E.10** Crear una función para construir una matriz de tamaño  $n$  con una estructura como la siguiente:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 1 & 3 & 3 & \cdots & n-1 & n \\ 1 & 2 & 5 & \cdots & n-1 & n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 2 & 3 & \cdots & 2n-3 & n \\ 1 & 2 & 3 & \cdots & n-1 & 2n-1 \end{pmatrix}$$

**E.11** Para valores de  $n$  entre 2 y 20, calcula el determinante de la matriz siguiente:

$$\begin{pmatrix} 1 & n & n & \cdots & n \\ n & 2 & n & \cdots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n & n & n & \cdots & n \end{pmatrix}$$

**E.12** Encuentra la inversa de las matrices del tipo siguiente:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

**E.13** Construye una función para calcular, en función del tamaño, los autovalores de las matrices del tipo siguiente y comprueba que su producto

coincide con el determinante:

$$\begin{pmatrix} 2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2 \end{pmatrix}$$

**E.14** Construye una función tal que, dado un *array* de longitud cualquiera, construya el conocido como determinante de Vandermonde.

$$\begin{vmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ x_1^2 & x_2^2 & x_3^2 & \cdots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & x_3^{n-1} & \cdots & x_n^{n-1} \end{vmatrix}$$

A continuación calcula su valor, y comprueba que coincide con la fórmula

$$\prod_{1 \leq i < j \leq n} (x_j - x_i)$$

¿Podrías escribir el código para calcular esta fórmula usando un solo bucle?  
¿Y sin bucles?

**E.15** Crea un array aleatorio de enteros entre 0 y 100 bidimensional, de tamaño  $1000 \times 1000$ . Cuenta el número de múltiplos de 5 que hay en cada fila y luego calcula la media de los valores obtenidos. Repite el proceso 100 veces. Indicación: usa el método [mean](#).

**E.16** Considera un conjunto de  $n$  puntos uniformemente espaciados en el intervalo  $(0, 1)$ ,  $x_0 < x_1 < \cdots < x_{n-1}$ . Dada una función  $u$ , sabemos que una aproximación de la derivada segunda en un punto  $x$  viene dada por

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

Para la función  $u(x) = \sin(x)$ , calcula los valores reales de su derivada segunda en los puntos  $x_i$  interiores del intervalo y compáralos con los valores obtenidos en la aproximación.

**E.17** Considera una matriz de la forma

$$A = \begin{pmatrix} \mathbf{0} & \mathbf{I} \\ K & D \end{pmatrix}$$



donde  $\mathbf{0}$  y  $\mathbf{I}$  son la matriz nula y la matriz identidad de tamaño  $2 \times 2$ , respectivamente, y  $K$  y  $D$  son matrices  $2 \times 2$  con la siguiente forma:

$$K = \begin{pmatrix} k & 0.5 \\ 0.5 & k \end{pmatrix} \quad D = \begin{pmatrix} -d & 1 \\ 1 & -d \end{pmatrix}$$

siendo  $k$  y  $d$  parámetros reales.

Construye una función que tenga como argumentos de entrada  $k$  y  $d$ , siendo  $k = -1000$  el valor por defecto para  $k$ , y que tenga como salida los autovalores de la matriz  $A$ .

**E.18** Resuelve el siguiente sistema de ecuaciones:

$$\left. \begin{array}{rcl} 2x - y - z & = & 4 \\ 3x + 4y - 2z & = & 11 \\ 3x - 2y + 4z & = & 11 \end{array} \right\}$$

**E.19** Crea una función para resolver un sistema de ecuaciones de la forma  $A\mathbf{x} = \mathbf{b}$  mediante el método iterativo de Gauss-Jacobi, que puede expresarse de la forma:

$$\mathbf{x}^{(i+1)} = D^{-1} \left( -(L + U)\mathbf{x}^{(i)} + \mathbf{b} \right)$$

donde  $D$ ,  $L$  y  $U$  corresponden a la descomposición de la matriz  $A$  dada en el Ejercicio E.5. Aplícala a la resolución del sistema del Ejercicio E.18.

**E.20** El método de Monte Carlo para calcular el área de un recinto plano consiste en generar puntos aleatorios en un recinto más grande que lo contenga, y cuyo área conozcamos. El área corresponderá a la proporción de puntos que caen dentro del recinto buscado. Calcula de este modo el área del círculo unidad.

Indicación: bastará hacerlo para un cuarto de círculo.



SciPy, que es el acrónimo de *Scientific Python*, es un módulo compuesto por un buen número de algoritmos matemáticos útiles para la resolución de una amplia variedad de problemas de índole científico.

En el mundo de la programación es recomendable *no reinventar la rueda*, es decir, no volver a hacer algo que ya está hecho (y posiblemente de una forma mejor). Esta situación surge en multitud de ocasiones cuando, en el proceso de la resolución de un determinado problema, debemos usar algún algoritmo conocido y que con seguridad ya ha sido programado con anterioridad. En tales circunstancias suele ser una sabia decisión el usar la implementación ya realizada en lugar de volverla a programar. SciPy proporciona precisamente esto: una colección de algoritmos matemáticos ya programados de forma eficiente de los que únicamente necesitamos saber cómo usarlos.

SciPy está compuesto por una serie de submódulos aplicables a una amplia diversidad de campos. La tabla 5.1 muestra los módulos disponibles:

Puesto que no pretendemos en estas notas llevar a cabo una descripción exhaustiva de cada uno de estos submódulos, mostraremos de forma breve cómo usar alguna función de los módulos `optimize`, `interpolate` e `integrate`, con los cuales sentar las ideas básicas para aprender a usar cualquiera de los otros.

---

## 5 1

### OPTIMIZACIÓN SIN RESTRICCIONES

El módulo `optimize` proporciona una serie de algoritmos para la resolución de problemas de optimización y el cálculo de raíces. En estas notas se ha usado la versión 0.18.1 de SciPy, por lo que es posible que si el lector usa una versión anterior algunas funciones no estén disponibles.

El módulo permite resolver problemas de optimización con y sin restricciones para funciones de una o varias variables, usando diversos algoritmos

Cuadro 5.1: Módulos presentes en SciPy

Submódulo	Descripción
<code>cluster</code>	Algoritmos de agrupamiento
<code>constants</code>	Constantes físicas y matemáticas
<code>fftpack</code>	Rutinas para la Transformada Rápida de Fourier
<code>integrate</code>	Integración de ecuaciones diferenciales ordinarias
<code>interpolate</code>	Interpolación y splines regulares
<code>io</code>	Entrada y salida de datos
<code>linalg</code>	Álgebra lineal
<code>ndimage</code>	Procesamiento de imágenes
<code>odr</code>	Regresión ortogonal
<code>optimize</code>	Optimización y cálculo de raíces
<code>signal</code>	Procesamiento de señales
<code>sparse</code>	Matrices dispersas
<code>spatial</code>	Estructuras de datos espaciales
<code>special</code>	Funciones especiales
<code>stats</code>	Funciones estadísticas
<code>weave</code>	Integración con C/C++

que incluyen optimización global, métodos de mínimos cuadrados, métodos quasi-Newton, programación secuencial cuadrática, entre otros. Dado que no es nuestra intención ser muy exhaustivos, veremos sólo algún ejemplo sencillo del uso de alguno de estos métodos. En cualquier caso, la elección del método más adecuado al problema a resolver precisa de un conocimiento general de los problemas de optimización que se escapa del alcance de estas notas.

Como primer ejemplo consideraremos la función de Rosenbrock de  $N$  variables, definida por

$$f(x_0, \dots, x_{N-1}) = \sum_{i=1}^{N-1} [100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2]$$

de la cual se conoce que alcanza su mínimo en el punto  $(1, \dots, 1)$ , siendo  $f(1, \dots, 1) = 0$ .

En primer lugar, crearemos una función que nos proporcione los valores de la función de Rosenbrock:

```
def rosenbrook(x):
    return sum(100.*(x[1:]-x[:-1]**2)**2 + (1-x[:-1])**2)
```

Y a continuación llamaremos al optimizador usando el nombre de la función como parámetro de entrada. Típicamente, en los algoritmos de optimización es preciso dar un punto inicial a partir del cual poner en marcha el procedimiento; además, nosotros proporcionamos algunas opciones adicionales en forma de diccionario:

```
from scipy.optimize import minimize

x0 = np.array([1.3, 0.7, 3.9, 3.9, 1.2, 1.6, 2.1])
opciones = {'xtol':1.e-8, 'disp':True}
opt = minimize(rosenbrook, x0, method='Powell', options=
    opciones)
```

siendo el resultado:

```
Optimization terminated successfully.
Current function value: 0.000000
Iterations: 26
Function evaluations: 2870
```

Las opciones proporcionadas aquí se refieren al parámetro que regula el criterio de parada (`xtol`) y a la información de salida (`disp`). Si esta última es `False`, entonces el algoritmo no devuelve información a menos que la extraigamos del objeto `opt` creado en la llamada. Esta información puede obtenerse simplemente con

```
print opt
```

resultando:

```
direc: array([[ 1.00000000e+00,  0.00000000e+00,
  0.00000000e+00,
  0.00000000e+00,  0.00000000e+00,  0.00000000
  0.00000000e+00],
 [ -1.16176646e-06, -1.38842362e-06,  5.31922868
  e-06,
  1.18136018e-05,  4.32075655e-06, -1.96558476
  e-06,
  9.91332650e-06],
 [ 1.87538194e-08,  4.17940823e-08,  6.17951780
  e-07,
  6.04908798e-07,  9.14032521e-07,  1.69204568
  e-06,
  2.60026901e-06],
```

```

[ -1.33951405e-03, -2.63439121e-03, -6.99973806
  e-03,
 -9.52271263e-03, -2.15130721e-02, -4.47290037
  e-02,
 -9.44484188e-02],
[ 0.00000000e+00, 0.00000000e+00, 0.00000000
  e+00,
 0.00000000e+00, 1.00000000e+00, 0.00000000
  e+00,
 0.00000000e+00],
[ -1.11668317e-01, -1.97386502e-01, -1.16436208
  e-01,
 -6.49484932e-02, -1.82185447e-02, 2.81508404
  e-03,
 8.48222299e-03],
[ -6.41742238e-10, -1.70859402e-09, -2.51610081
  e-09,
 -6.56698002e-09, -1.31379941e-08, -2.62980032
  e-08,
 -5.15356896e-08]])
fun: 2.2378384321174616e-21
message: 'Optimization terminated successfully.'
nfev: 2870
nit: 26
status: 0
success: True
x: array([ 1., 1., 1., 1., 1., 1., 1.])

```

O si queremos detalles concretos, invocando el atributo correspondiente del objeto, como por ejemplo, el punto donde se alcanza el óptimo:

```
print opt.x
```

```
[ 1.  1.  1.  1.  1.  1.  1.]
```

El método empleado en este ejemplo es una variante del conocido como método de Powell, que es un método de direcciones conjugadas que no usa información sobre la derivada de la función objetivo. De hecho, ni siquiera es preciso que la función sea diferenciable.

El resto de métodos disponibles para la función `minimize` puede consultarse con el comando `info`. Por ejemplo, está disponible el método del Gradiente Conjugado de Polak-Ribière, que sí precisa información de la derivada de la función objetivo. Dicha información puede venir dada por el usuario a través de una función, o puede ser estimada numéricamente por el propio algoritmo. Estas dos opciones son controladas por el parámetro `jac`, que puede ser tanto un valor booleano como una función. Si `jac` es una función, entonces

se entiende que ésta proporciona el valor del gradiente, mientras que si es un valor booleano y es `True` entonces significa que el gradiente de la función objetivo es devuelto por la propia función objetivo. Si `jac=False` entonces el gradiente se estimará numéricamente.

En el ejemplo anterior, dado que la derivada de la función de Rosenbrook es:

$$\begin{aligned}\frac{\partial f}{\partial x_0} &= -400x_0(x_1 - x_0^2) - 2(1 - x_0) \\ \frac{\partial f}{\partial x_j} &= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j), \quad 1 \leq j \leq N - 2 \\ \frac{\partial f}{\partial x_{N-1}} &= 200(x_{N-1} - x_{N-2}^2)\end{aligned}$$

podemos, o bien crear una función que devuelva este gradiente:

```
def grad_rosenbrook(x):
    der = np.zeros_like(x)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    der[1:-1] = 200*(x[1:-1]-x[:-2]**2) - 400*x[1:-1]*(x
        [2:] - x[1:-1]**2) - 2*(1-x[1:-1])
    return der
```

o bien hacer que la propia función objetivo devuelva también el gradiente, esto es, definiríamos la función de la forma siguiente:

```
def newrosenbrook(x):
    f = sum(100.*(x[1:]-x[:-1]**2)**2 + (1-x[:-1])**2)
    der = np.zeros_like(x)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    der[1:-1] = 200*(x[1:-1]-x[:-2]**2) - 400*x[1:-1]*(x
        [2:] - x[1:-1]**2) - 2*(1-x[1:-1])
    return f, der
```

De este modo, podríamos usar el método de optimización de alguna de las siguientes formas. El punto inicial y las opciones usadas son:

```
x0 = np.array([1.3, 0.7, 3.9, 3.9, 1.2, 1.6, 2.1])
opciones = {'disp': True}
```

Primera opción: usamos la función objetivo y su derivada como funciones independientes. El parámetro `jac` es igual al nombre de la función que calcula el gradiente del objetivo:

```
# primer método: gradiente como función independiente
```

```
opt1 = minimize(rosenbrock, x0, method='CG', jac=
    grad_rosenbrock, options=opciones)
```

```
Optimization terminated successfully.
    Current function value: 0.000000
    Iterations: 103
    Function evaluations: 205
    Gradient evaluations: 205
```

```
# óptimo:
print opt1.x
```

```
[ 1.00000002  1.00000004  1.00000007  1.00000015
  1.00000003  1.00000006
  1.00000122]
```

Segunda forma: usamos la función `nuevarosenbrock` que devuelve la función objetivo y su gradiente de forma simultánea. En esta llamada, el parámetro `jac=True`:

```
# segundo método: gradiente obtenido en la función
    objetivo
```

```
opt2 = minimize(newrosenbrock, x0, method='CG', jac=True,
    options=opciones)
```

```
Optimization terminated successfully.
    Current function value: 0.000000
    Iterations: 103
    Function evaluations: 205
    Gradient evaluations: 205
```

```
# óptimo:
print opt2.x
```

```
[ 1.00000002  1.00000004  1.00000007  1.00000015
  1.00000003  1.00000006
  1.00000122]
```

Por último, prescindimos del gradiente y hacemos que el método lo calcule de forma aproximada. Ahora `jac=False`:



```
# tercer método: gradiente calculado numéricamente

opt3 = minimize(rosenbrook, x0, method='CG', jac=False,
               options=opciones)
```

```
Optimization terminated successfully.
      Current function value: 0.000000
      Iterations: 146
      Function evaluations: 2466
      Gradient evaluations: 274
```

```
# óptimo
print opt3.x
```

```
[ 1.000000004  1.000000009  1.000000018  1.000000035
  1.000000071  1.000000144
  1.000000291]
```

Lógicamente, el resultado de las dos primeras optimizaciones es idéntico, pues se han usado la misma información, aunque dispuesta de diferente modo. No ocurre así con la tercera opción, pues en este caso el método ha usado un gradiente numérico, que evidentemente requiere un mayor número de iteraciones, no sólo del método, sino también del número de evaluaciones de la función objetivo.

## 5.2

### OPTIMIZACIÓN CON RESTRICCIONES

SciPy también dispone de diversos algoritmos para resolver problemas de optimización con restricciones del tipo siguiente:

$$\begin{aligned} &\text{Minimizar} && f(\mathbf{x}) \\ &\text{sujeito a} && h_i(\mathbf{x}) = 0, \quad 1 \leq i \leq n, \\ & && g_j(\mathbf{x}) \geq 0, \quad 1 \leq j \leq m, \\ & && l_k \leq x_k \leq u_k, \quad 1 \leq k \leq N, \end{aligned}$$

con  $\mathbf{x} = (x_1, \dots, x_N)$ . Esto es, un problema con  $N$  variables,  $n$  restricciones de igualdad y  $m$  restricciones de desigualdad, además de acotaciones simples sobre cada variable. Como ejemplo particular consideremos:

$$\begin{aligned} &\text{Minimizar} && x_0^2 + 2x_1^2 - 2x_0x_1 - 2x_0 \\ &\text{sujeito a} && x_0^3 - x_1 = 0, \\ & && x_1 \geq 1. \end{aligned}$$

Para resolver este tipo de problemas con el módulo `optimize` creamos en primer lugar funciones para la función objetivo y su derivada:

```
def fobj(x):
    return x[0]**2 + 2*x[1]**2 - 2*x[0]*x[1] - 2*x[0]

def grad_fobj(x):
    return np.array([ 2*(x[0]-x[1]-1), 4*x[1]-2*x[0] ])
```

Las restricciones, por su parte, se definen a través de diccionarios con claves `'type'`, para determinar si se trata de restricciones de igualdad o desigualdad, y `'fun'` y `'jac'`, para definir las restricciones y sus derivadas:

```
constr1 = {'type':'eq', 'fun': lambda x: x[0]**3-x[1], '
           'jac': lambda x: np.array([3*x[0]**2,-1.])}
constr2 = {'type':'ineq', 'fun': lambda x: x[1]-1, 'jac':
           lambda x: np.array([0.,1.])}
constr = (constr1,constr2)
```

La optimización se lleva a cabo con una llamada similar a las anteriores

```
res = minimize(fobj, [-1.0,1.0], jac=grad_fobj,
               constraints=constr, method='SLSQP',options={'disp':
               True})
```

en el que el punto inicial es  $(-1, 1)$ . El resultado es

```
Optimization terminated successfully.      (Exit mode 0)
Current function value: -1.00000018311
Iterations: 9
Function evaluations: 14
Gradient evaluations: 9
```

El objeto `res` contiene toda la información del proceso:

```
print res

fun: -1.0000001831052137
   jac: array([-1.99999982,  1.99999982,  0.          ])
message: 'Optimization terminated successfully.'
   nfev: 14
   nit: 9
   njev: 9
status: 0
success: True
   x: array([ 1.00000009,  1.          ])
```

## 5 3

## INTERPOLACIÓN DE DATOS

El módulo `interpolate` proporciona diversos métodos y funciones para la interpolación de datos en una o varias dimensiones. Por simplicidad en estas notas sólo veremos cómo se utiliza la función `interp1d` para la interpolación de datos en una dimensión. La llamada a este método devuelve una función que puede ser evaluada en cualquier punto. Dicha función es obtenida mediante interpolación sobre un conjunto de datos proporcionado en una serie de puntos.

Por ejemplo, supongamos que tenemos el siguiente conjunto de puntos:

(0.00,0.00), (0.20,0.56), (0.40,0.93), (0.60,0.97), (0.80,0.68), (1.00,0.14)

que definimos con los siguientes *arrays*

```
x = np.linspace(0,1,6)
y = np.array([0.,0.56,0.93,0.97,0.68,0.14])
```

y que aparecen representados en la Figura 5.1a.

Para obtener una función lineal a trozos que pase por esos puntos nos bastará usar la función `interp1d` del siguiente modo:

```
from scipy.interpolate import interp1d
f = interp1d(x,y)
```

Ahora `f` es una función que interpola dichos datos:

```
f = interp1d(x,y)
print f(x)
```

```
[ 0.      0.56  0.93  0.97  0.68  0.14]
```

En la Figura 5.1b hemos dibujado la función `f` resultante. El cálculo de cualquier valor sobre dicha función se hará de la forma habitual:

```
print f(0.23)
```

```
0.6155
```

La interpolación realizada es, por defecto, lineal a trozos; pero la función `interp1d` admite otras opciones. Por ejemplo, podemos hacer

```
f1 = interp1d(x,y,kind='cubic')
```

y obtener así una interpolación basada en splines de tercer orden. Otras opciones son `'slinear'` o `'quadratic'` para interpolación por splines de primer

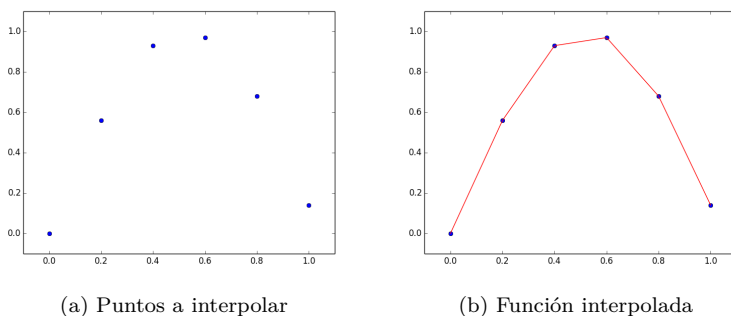


Figura 5.1: Interpolación de datos unidimensional

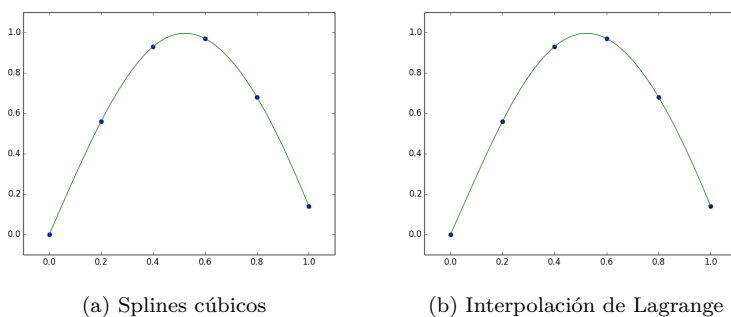


Figura 5.2: Otras interpolaciones

o segundo orden, respectivamente. La Figura 5.2a muestra la interpolación cúbica.

Por último, disponemos de la función `lagrange` para calcular el polinomio de interpolación de Lagrange de un conjunto de puntos. En este caso, la función devuelve un objeto tipo *polinomio* de NumPy (que puede considerarse una función). La Figura 5.2b muestra un gráfico de dicho polinomio.

```
from scipy.interpolate import lagrange
p = lagrange(x,y)
print p
```

$$-1.563 \times x^5 + 6.771 \times x^4 - 9.479 \times x^3 + 1.604 \times x^2 + 2.807 \times x$$

## 5 4

## RESOLUCIÓN DE ECUACIONES DIFERENCIALES

El submódulo `integrate` posee un integrador, `odeint` para ecuaciones diferenciales ordinarias muy sencillo de usar que permite resolver de forma rápida ecuaciones o sistemas diferenciales de primer orden. Esta función usa las rutinas LSODA del paquete ODEPACK que es una librería clásica escrita en lenguaje FORTRAN.

Veamos un ejemplo de uso de esta función. Consideremos la ecuación del oscilador amortiguado. Para unos parámetros  $\omega$  y  $\gamma$ , que son constantes que tienen que ver con las características elásticas y la amortiguación debida a la viscosidad, la ecuación se puede escribir en la forma:

$$y'' + \gamma y' + \omega y = 0$$

con condiciones iniciales  $y(0) = y_0$  e  $y'(0) = y'_0$ .

Al tratarse de una ecuación de segundo orden, y dado que el integrador resuelve ecuaciones o sistemas de primer orden, debemos reescribir la ecuación como un sistema de primer orden. Definiendo un vector

$$\mathbf{y} = \begin{pmatrix} y \\ y' \end{pmatrix}$$

entonces la ecuación anterior es equivalente al sistema siguiente:

$$\mathbf{y}' = \begin{pmatrix} y' \\ -\omega y - \gamma y' \end{pmatrix}, \quad \mathbf{y}(0) = \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}$$

Para resolver el sistema con Python, importamos en primer lugar la función `odeint`:

```
from scipy.integrate import odeint
```

y a continuación definimos los datos del problema:

```
omega = 2.
gamma = 0.5
y0 = [2.5, 0] # condiciones iniciales

# segundo miembro de y'=f(y,x)
def func(y,x):
    return [y[1], -omega*y[0] - gamma*y[1]]
```

Finalmente, definimos un *array* de puntos sobre los que calcular la solución y realizamos la llamada:

```
x = np.arange(0,8,0.05)
y = odeint(func, y0, x)
```

De esta forma, `y` es un *array* 2D que contiene en su primera columna la solución de la ecuación, y la segunda columna será su derivada.

El método usa una aproximación del gradiente de la función del segundo miembro. Si podemos obtener este gradiente de forma exacta, podemos mejorar el rendimiento del método. La llamada se haría entonces del siguiente modo:

```
def func_grad(y,x):
    return [[0,1],[-w,2*g]]
y = odeint(func, y0, x, Dfun=func_grad)
```

## 5 5

### EJERCICIOS

**E.1** Resolver el siguiente problema de optimización:

$$\begin{aligned} &\text{Maximizar} && x e^{xy} \\ &\text{sujeto a} && x^2 + y = 0 \end{aligned}$$

**E.2** Resolver el siguiente problema de optimización:

$$\begin{aligned} &\text{Minimizar} && (x-1)^2 + (y-2)^2 \\ &\text{sujeto a} && x - 2y + 2 \geq 0 \\ &&& -x - 2y + 6 \geq 0 \\ &&& x, y \geq 0 \end{aligned}$$

**E.3** Considera la función  $f(x) = -3x^3 + 2x^2 + 8x - 1$ . Genera un número determinado de puntos de la misma y obtén la interpolación de Lagrange. ¿Cuáles son los coeficientes del polinomio interpolador resultante? ¿Cambia en función del número de puntos que se generan?

**E.4** Para la función  $f$  del Ejercicio E.3, construye la interpolación lineal a trozos y mediante splines cúbicos en los puntos de abscisa 1, 2, 3 y 4. ¿Cuánto valen los interpoladores en el punto 2.5? ¿Y en 0?

**E.5** Resolver el siguiente problema (ecuación de Airy):

$$\begin{aligned} &y'' - xy = 0 \\ y(0) &= \frac{1}{\sqrt[3]{3^2}\Gamma(\frac{2}{3})}, \quad y'(0) = -\frac{1}{\sqrt[3]{3}\Gamma(\frac{1}{3})} \end{aligned}$$

donde  $\Gamma(x)$  denota la función Gamma de Euler. La solución exacta de esta ecuación es la función de Airy  $y(x) = \text{Ai}(x)$ .

Indicación:. Las funciones Ai y  $\Gamma$  están en el submódulo [special](#).





Matplotlib es un conjunto de librerías de Python para la construcción de gráficos 2D especialmente adecuada para la visualización de datos y la creación de imágenes de calidad acorde con los estándares de publicación. Está particularmente diseñada para el uso de *arrays* de NumPy y posee una interfaz muy similar a la de MATLAB.

La librería proporciona un buen manejo de gráficos de calidad en múltiples formatos, permitiendo la interacción con el área gráfica además de integrarse con diferentes herramientas de desarrollo de GUIs.<sup>1</sup>

La librería es muy amplia, por lo que en estas notas sólo daremos un breve vistazo de su manejo para la visualización de datos. Esencialmente podemos trabajar con la librería de dos formas: usando `pylab`, que es un módulo independiente que proporciona también la funcionalidad de NumPy y que es adecuado para la creación interactiva y rápida de gráficos, o bien, usando el submódulo `pyplot` con el que podemos tener un control más fino sobre el gráfico. En estas notas usaremos ambas alternativas para mostrar su funcionamiento.

### El *Backend*

El uso diverso que permite Matplotlib (embeber gráficos en GUIs, creación de ficheros gráficos en algún format específico, etc.) precisa de una comunicación adecuada con el entorno en el que se va a generar el gráfico. El *backend* es la herramienta que hace el trabajo detrás del telón, y necesita ser definido de forma correcta. En general, una instalación estándar de Matplotlib habrá seleccionado correctamente el *backend*, en cuyo caso el usuario no ha de preocuparse de este tema. Sin embargo, si el funcionamiento de los gráficos no es el adecuado, es posible que sea necesario modificar el *backend* por defecto. En el entorno Jupyter Notebook disponemos de la *función mágica* `%matplotlib`

---

<sup>1</sup> *Graphical User Interface*: interfaz gráfica de usuario.

para seleccionarlo. Si queremos los gráficos integrados en el propio entorno entonces escribiremos

```
%matplotlib notebook
```

mientras que si queremos que los gráficos aparezcan en una ventana propia bastará poner

```
%matplotlib
```

```
Using matplotlib backend: Qt4Agg
```

que nos informa del *backend* seleccionado. Para manejar gráficos de forma interactiva sugerimos usar esta segunda opción.

## 6.1

### GRÁFICOS INTERACTIVOS

Comenzaremos con el uso interactivo con el módulo `pylab`. La ventaja del uso de este módulo es que carga tanto el módulo NumPy como las funciones apropiadas de Matplotlib, pero lo hace de forma masiva, por lo que se tiende a usar otras alternativas. No obstante, para construir gráficos sencillos es bastante cómodo.

#### 6.1.1 Interactividad

Una vez cargado el módulo `pylab`, lo primero que haremos será saber si la sesión es o no interactiva, para lo cual usamos la función:

```
from pylab import *  
isinteractive()
```

```
True
```

El resultado será `True` o `False`, en función de la consola usada. El resultado nos dirá qué sucederá cuando creemos un elemento gráfico. Si ha sido `True`, entonces la siguiente función

```
figure()
```

```
<matplotlib.figure.Figure at 0x7f83db020290>
```

creará una nueva ventana, como la mostrada en la Figura 6.1. La salida de la función nos informa del objeto creado y la dirección de memoria donde reside. Dado que esta información no es relevante por ahora, prescindiremos en lo sucesivo de mostrarla.

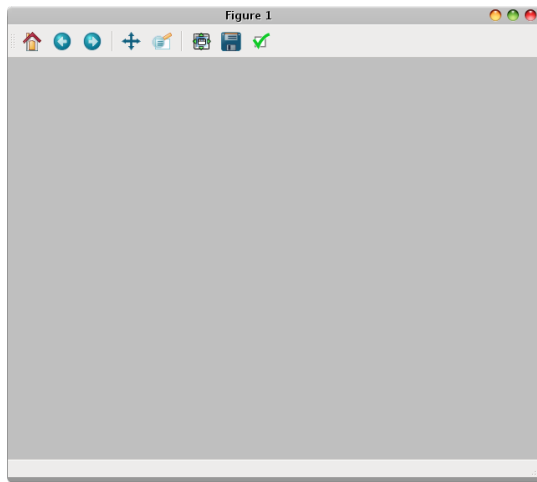


Figura 6.1: Ventana creada por la función `figure()`

Si por el contrario, la salida de la función `isinteractive()` es `False`, entonces la orden `figure()` dará el mismo resultado, pero no se mostrará ventana alguna. Para ver el gráfico será preciso invocar la orden `show()`. Esta es la diferencia fundamental entre tener la sesión en interactivo o no. Cualquier cambio en una sesión interactiva se ve reflejado al momento sobre la gráfica, mientras que si la sesión interactiva no está activada habrá que usar la orden `show()` (para mostrar la ventana gráfica por primera vez) o `draw()` para actualizar los cambios hechos al dibujo.

En cualquier caso, podemos activar o desactivar la interactividad mediante:

```
ion() # activa la sesión interactiva
ioff() # desactiva la sesión interactiva
```

## 6.1.2 Figuras y gráficos

La orden `figure()` crea una ventana con título *Figure* más un número entero que irá incrementándose sucesivamente. Es posible hacer la llamada de la forma `figure(num)`, bien para crear la figura con la numeración que deseemos, o bien, si dicha figura existe, para hacerla activa. En lugar de un número entero es posible pasar un *string* como argumento, que se convertirá en el título de la ventana creada.

La orden

```
plot([1,3,2])
```

crea un gráfico en la ventana como el de la Figura 6.2. El comando `plot` es sencillo de usar; si se le proporciona una lista o *array*, crea una línea que une

los puntos de abscisa dada por el índice de la lista, y ordenada dada por el valor de la lista. En el caso de la Figura 6.2, se crea una línea que une los puntos (0, 1), (1, 3) y (2, 2).

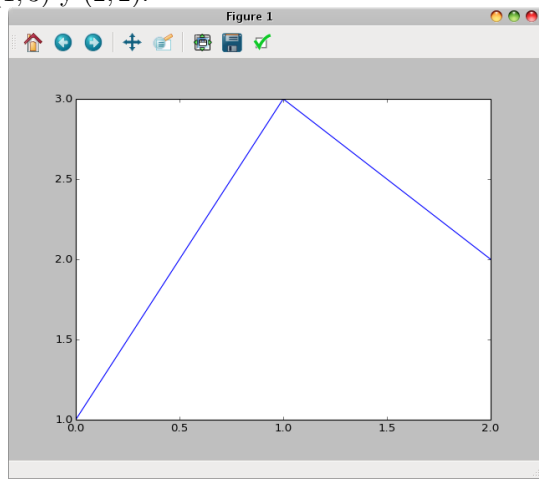


Figura 6.2: Gráfico dentro de la figura

El gráfico se crea dentro de la figura que esté activa, si hubiera una, o directamente se crearía una nueva figura para contenerlo. Obsérvese que el gráfico se crea con unos ejes, que por defecto se escalan al tamaño del gráfico creado, se etiquetan con valores oportunos y la línea es coloreada de forma automática. Es importante tener clara la diferencia entre la *ventana* gráfica, creada por la orden `figure`, y los *ejes* creados con `plot`, pues son objetos distintos, que posteriormente aprenderemos a manejar.

Si en lugar de proporcionar al comando `plot` una lista, le damos dos, entonces la primera lista corresponderá a las abscisas y la segunda a las ordenadas de los puntos (en particular, esto implica que ambas listas deben tener la misma longitud):

```
x = arange(0, 2.1, 0.5)
y = x**2
plot(x, y)
```

El resultado puede verse en la Figura 6.3. Nótese que hemos usado la función `arange` de NumPy (recuérdese que `pylab` importa éste módulo) y que por tanto, `x` e `y` son *arrays*. Obsérvese también cómo el gráfico es reescalado para poder mostrar la nueva línea en el eje que ya estaba creado.

Podríamos haber hecho que el nuevo comando `plot` borrara el dibujo anterior, en lugar de añadirse al existente. La función `hold` es la encargada de activar o desactivar el estado de concurrencia, esto es, si los sucesivos dibujos se mostrarán junto a los anteriores, o bien éstos serán borrados y sustituidos por el último. Se puede cambiar de estado invocándola sin parámetro, o bien

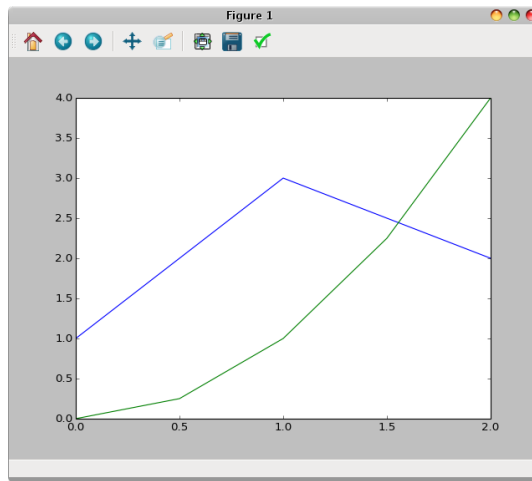


Figura 6.3: Dos gráficas en el mismo eje

activarlo o desactivarlo mediante `hold(True)` o `hold(False)`, respectivamente. La función `ishold()` nos proporciona el estado de concurrencia actual.

Para cerrar una ventana bastará usar la orden

```
close() # cierra la Figura activa
close(num) # cierra la Figura num
```

Si lo que queremos es borrar los gráficos de la figura activa sin cerrar la ventana disponemos de

```
cla() # borra los gráficos pero mantiene el eje
clf() # borra los ejes pero mantiene la figura
```

### 6.1.3 Subplots

El posible tener varios ejes distintos en la misma ventana gráfica, para lo cual usaremos la orden:

```
subplot(n,m,k)
```

la cual divide la figura en  $n$  filas y  $m$  columnas y crea unos ejes en la posición  $k$  (contando de izquierda a derecha y de arriba a abajo). La comas de separación entre  $n$ ,  $m$  y  $k$  no son necesarias (a menos que alguno de los valores tenga más de un dígito). Por ejemplo,

```
subplot(234)
```

abre una ventana como la de la Figura 6.4a y selecciona dicho eje como el eje activo. Nótese que la figura es dividida en dos filas de tres columnas cada

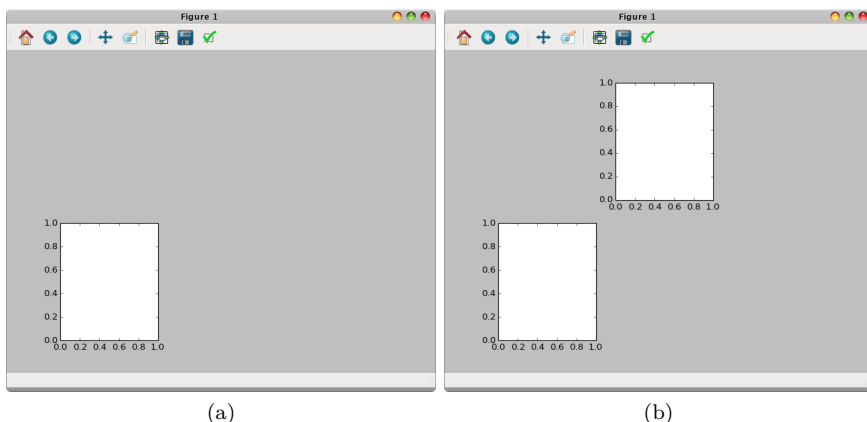


Figura 6.4: Subplots

una, y se crean los ejes en la posición 4 (contando por filas). Si a continuación escribimos

```
subplot(232)
```

entonces en la misma figura se crean unos ejes en la posición 2 (ver Figura 6.4b), que serán los nuevos ejes activos. ¿Qué ocurrirá si ahora escribimos `subplot(211)`? En este caso, la estructura es sobrescrita y aparecen los ejes en la posición que muestra la Figura 6.5, siendo éste último el nuevo eje activo.

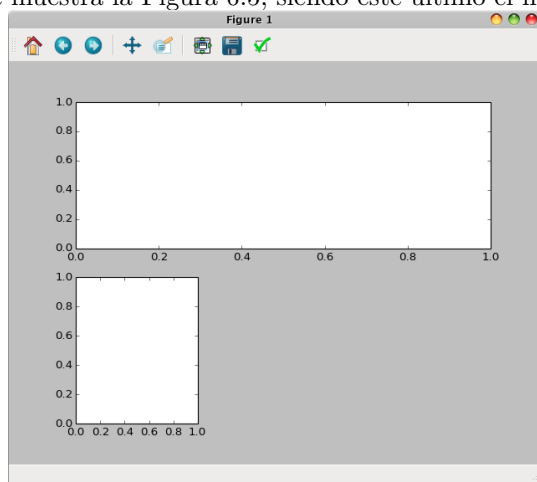


Figura 6.5: Distintos ejes en la misma figura

Como puede observarse, la función `subplot` permite organizar de forma estructurada cualquier combinación de ejes que se desee. El comando `plot` di-

bujará el gráfico correspondiente sobre el eje activo. Por ejemplo, la Figura 6.6 corresponde a las siguientes órdenes:

```
subplot(221)
subplot(222)
subplot(212)
x = linspace(0,1,10)
y = sin(x)
z = cos(x)
w = sqrt(x)
plot(x,w) # dibuja sobre el eje activo (212)
subplot(221) # nuevo eje activo (221)
plot(x,y) # dibuja sobre el eje activo (221)
subplot(222) # cambiamos de eje activo al (222)
plot(x,z) # dibuja sobre el eje activo (222)
```

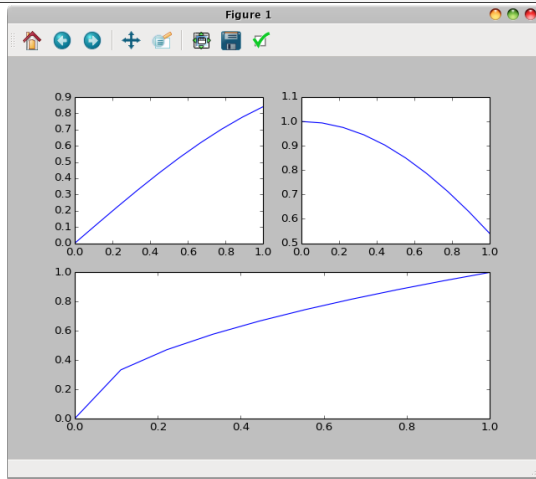


Figura 6.6: Varios gráficos en la misma figura

## 6.1.4 Axes

Es posible organizar los ejes creados en una figura de forma no estructurada con el comando `axes`:

```
axes()
```

crea unos ejes por defecto, que equivale a hacer `subplot(111)`. Si a continuación escribimos:

```
axes([0.2,0.5,0.3,0.3])
```

obtendremos uno ejes como los de la Figura 6.7.

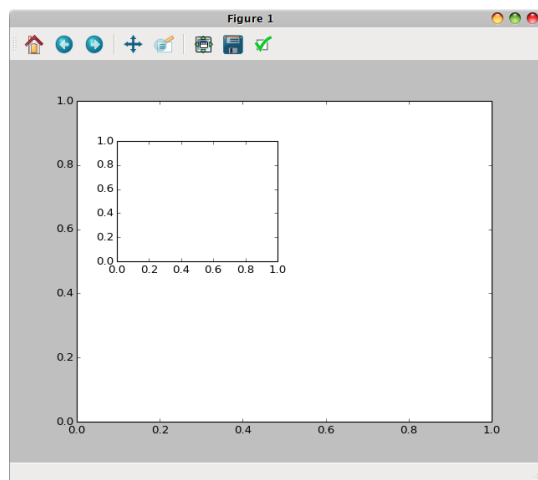


Figura 6.7: Varios ejes no estructurados

Los dos primeros números en el argumento de la función `axes` hacen referencia a las coordenadas de la esquina inferior izquierda, y los otros dos, a la anchura y altura, respectivamente, de los ejes a situar. Las coordenadas están normalizadas entre 0 y 1.<sup>2</sup>

## 6.2

### AÑADIENDO OPCIONES

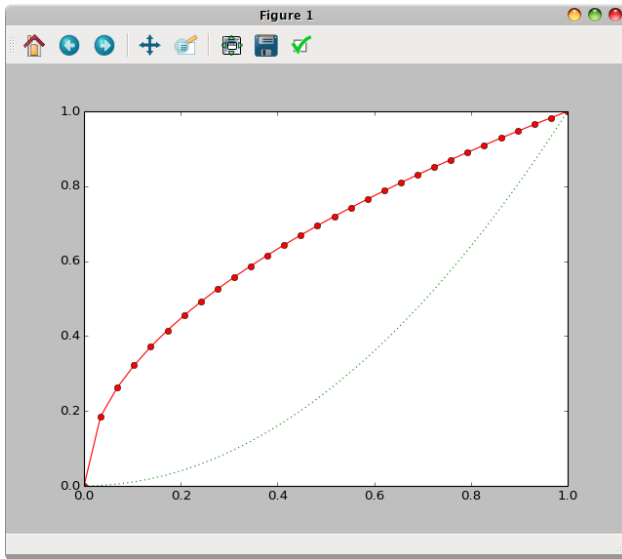
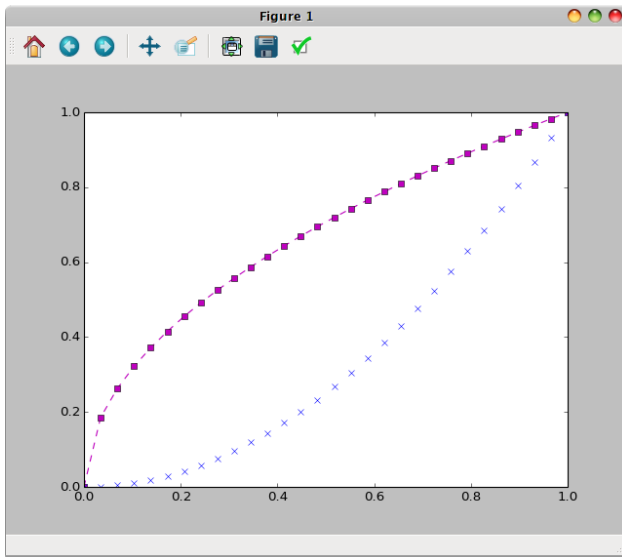
El comando `plot` admite varias secuencias de datos y una serie interminable de opciones para controlar todos los aspectos del gráfico. Algunas de estas opciones son equivalentes a las de MATLAB. La Figura 6.8 muestra un par de ejemplos.

Como podemos ver en la Figura 6.8, podemos usar, tras los *arrays* que determinan los puntos del gráfico, una cadena con diversos caracteres con los que configurar algunos aspectos del tipo de línea usada, como el color, el estilo de línea o los marcadores que señalan cada uno de los puntos del gráfico. Por ejemplo, en la Figura 6.8a, los datos *x*, *y* se dibujan según la cadena `'r-o'`, que significa que se ha usado color rojo, línea continua y círculos como marcadores, mientras que la cadena `'g:'` hace uso del color verde, con línea punteada para dibujar la pareja *x*, *z*. En la Figura 6.8b, `'m--s'` dibuja en color morado, línea discontinua y marcadores cuadrados, y la cadena `'bx'` dibuja en azul sólo marcadores con forma de cruz. La tabla 6.1 muestra una breve representación de las opciones más comunes. Para una lista completa véase la ayuda del

<sup>2</sup>Nótese que el posicionamiento de los ejes por defecto corresponde a las coordenadas normalizadas `[0.125, 0.1, 0.775, 0.8]`.



```
x = linspace(0,1,30)
y = sqrt(x)
z = x**2
```

(a) `plot(x,y,'r-o',x,z,'g:')`(b) `plot(x,y,'m--s',x,z,'bx')`Figura 6.8: Ejemplos de uso de `plot`

Cuadro 6.1: Opciones del comando `plot`

Carácter	Color	Carácter	Marcador
b	azul	.	punto
g	verde	o	círculo
r	rojo	^	triángulo
y	amarillo	*	estrella
m	magenta	x	cruz
k	negro	s	cuadrado
w	blanco	+	signo más

Carácter	Estilo de línea
-	línea continua
--	línea discontinua
:	línea punteada
-.	línea semipunteada

comando `plot`.

Además de este tipo de opciones abreviadas, el comando `plot` permite configurar muchos otros aspectos del gráfico a través de argumentos opcionales, algunos de los cuales tienen el mismo efecto que los ya vistos. Puesto que no es nuestra intención ser exhaustivos, mostraremos algunos ejemplos de opciones en la siguiente sección a la vez que introducimos algo más de control sobre los gráficos.

63

CONFIGURANDO VARIOS ELEMENTOS DEL GRÁFICO

Títulos y leyendas

Podemos incluir un título al eje del gráfico a dibujar con el comando

```
title(cadena)
```

También es posible distinguir cada uno de las líneas trazadas con `plot` mediante una leyenda, usando una etiqueta definida por el argumento opcional

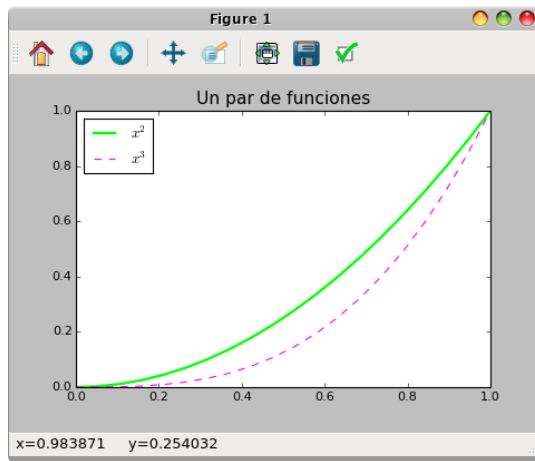


Figura 6.9: Título y leyenda

`label`. La leyenda se activa con el comando `legend`, que entre otros argumentos permite situar la leyenda en posiciones predeterminadas con la opción `loc`, el estilo de fuente, etc. Una vez más la ayuda del comando proporciona todas las posibilidades. La Figura 6.9 muestra el resultado de las siguientes órdenes:

```
x = linspace(0,1,20)
y = x**2
z = x**3
plot(x,y,linewidth=2,label='$x^2$',color=(0,1,0))
plot(x,z,linestyle='dashed',color=(1,0,1),label='$x^3$')
title('Un par de funciones',fontsize=14)
legend(loc=0)
```

Nótese que en las cadenas de caracteres que conforman las etiquetas para la leyenda se ha usado notación  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ . También se han usado otras opciones del comando `plot`. La leyenda debe ser activada después de los comandos `plot` y recogerá todas las etiquetas en función del orden.

### Texto y anotaciones

La Figura 6.10 ha sido generada con el siguiente código:

```
axes(axisbg=(0.35,0.25,0.75))
# datos
x = linspace(0,1,20)
y = x**2
plot(x,y,'b-o') # dibujo
# anotaciones
```

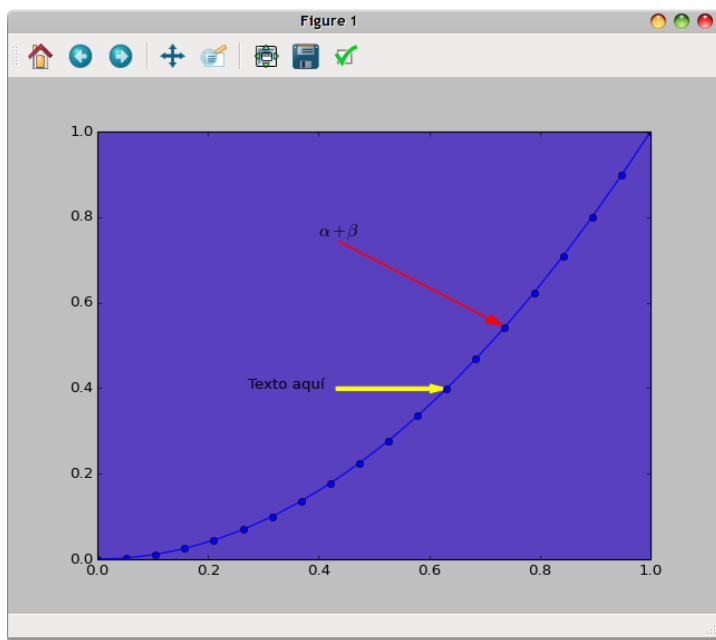


Figura 6.10: Texto y anotaciones

```
text(x[12]-0.22,y[12],u'Texto aquí',fontsize = 12,
     horizontalalignment='right')
arrow(x[12]-0.2,y[12],0.2,0., color='yellow',
      length_includes_head = "True", width=0.008, head_width
      =0.02)
text(x[14]-.3,y[14]+0.2,r'$\alpha+\beta$',
     verticalalignment='bottom',horizontalalignment='center',
     ,fontsize=14)
arrow(x[14]-.3,y[14]+0.2,0.27,-0.18,color='red')
```

Obsérvense algunas de las opciones empleadas: `axisbg` proporciona el color de fondo de los ejes; en este caso, el color se ha determinado a través de una tupla de valores reales entre 0 y 1 en formato *RGB*.<sup>3</sup>

La orden `text` sitúa una cadena de caracteres en las coordenadas determinadas por los dos primeros argumentos. En el ejemplo, los datos con los que se ha construido la curva han sido usados para determinar tales coordenadas. Puesto que la cadena contenía acentos (en el primer `text`) y notación  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  (en el segundo) las hemos pasado como Unicode y *raw*, respectivamente. El resto de opciones son evidentes.

También hemos incluido flechas para señalar objetos en el gráfico con la orden `arrow`, la cual precisa de cuatro coordenadas; las dos primeras señalan el

<sup>3</sup>Red, Green, Blue.

origen del vector, y las dos segundas las coordenadas del desplazamiento (que no las coordenadas del extremo). Las opciones empleadas son autoexplicativas.

### Etiquetas para los ejes

Echemos un vistazo al ejemplo de la Figura 6.11, el cual ha sido generado con el siguiente código:

```
scatter(random.rand(1000),random.rand(1000))
xlabel('Valores en X')
ylabel('Valores en Y')
xlim(-1,2)
ylim(0,1.5)
xticks([-1,0,0.5,1,2])
yticks(arange(0,1.6,0.1))
minorticks_on()
```

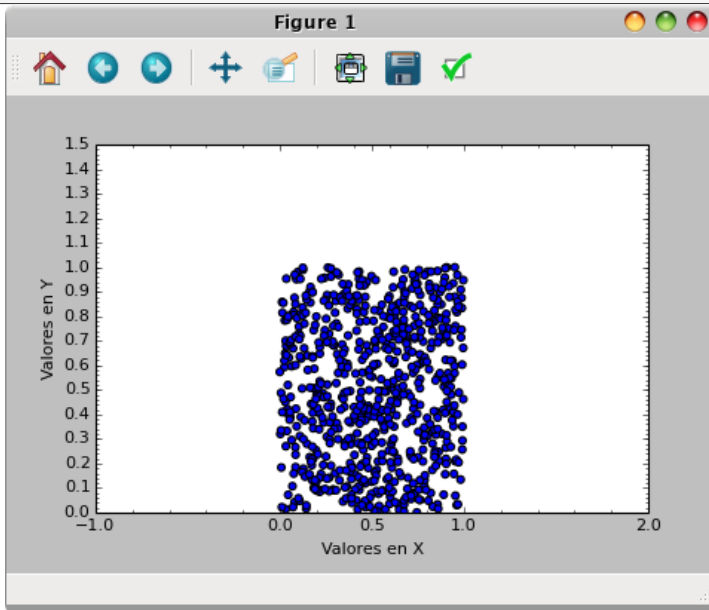


Figura 6.11: Etiquetas en los ejes

Como podemos ver, este gráfico ha sido generado con la orden `scatter` que en lugar de dibujar líneas, dibuja un conjunto de puntos (sin relacionar) cuyas coordenadas vienen dadas por dos listas (en nuestro caso, dos *arrays* aleatorios). El resto de órdenes establece leyendas para los ejes (con `xlabel` e `ylabel`), los límites que determinan los ejes del gráfico (con `xlim` e `ylim`), y las marcas que se muestran en cada eje (con `xticks` e `yticks`), que son definidas a través de una lista o un *array*. Por último, la orden `minorticks_on()` activa las marcas de subdivisión en ambos ejes.

## Otros tipos de gráficos

La cantidad de tipos de gráficos diferentes que Matplotlib puede generar es enorme, por lo que es muy recomendable echarle un vistazo a la galería que aparece en la página web del proyecto ([matplotlib.org/gallery](http://matplotlib.org/gallery)). No sólo se pueden apreciar las posibilidades de creación de gráficos sino que además se puede ver el código con el que se generan.<sup>4</sup>

### 6 4

## GRÁFICOS Y OBJETOS

En las secciones anteriores hemos visto cómo funciona el módulo `pylab` de forma interactiva. Esta forma de trabajar es útil para probar ejemplos sencillos, pero desde nuestro punto de vista, tenemos un mayor control de lo que sucede en un gráfico si manejamos adecuadamente los objetos de los que está compuesto. En esencia, lo que necesitamos es almacenar los objetos con los que construimos un gráfico en variables, y usar los métodos que provee Python para irlos modificando.

Por otra parte, en lugar de trabajar con el módulo `pylab`, en esta sección usaremos directamente `pyplot` y NumPy, los cuales importaremos de la forma estándar:

```
import matplotlib.pyplot as plt
import numpy as np
```

Crearemos una figura, la cual asignamos a una variable para acceder adecuadamente a los métodos disponibles.

```
fig = plt.figure()
```

Los objetos gráficos tienen su propia jerarquía, por ejemplo, en una figura podemos incluir varios ejes (tal y como hacíamos con `subplot`):

```
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)
```

Al manejar objetos, a veces es necesario actualizar el estado del dibujo para que las acciones hechas sobre él se muestren, aun estando en modo interactivo. Para ello usaremos la orden:

```
plt.draw()
```

Creamos unos datos y dibujamos en cada uno de los ejes:

---

<sup>4</sup>Sólo hay que tener en cuenta la versión que se tiene instalada de Matplotlib, y la versión que se use en el ejemplo, pues en ocasiones algunas características no están cubiertas por versiones inferiores.

```
x = np.linspace(0,4,100)
y = np.cos(2*x*np.pi)*np.exp(-x)
z = np.sin(3*x*np.pi)
a = ax1.plot(x,y,x,z)
b, = ax2.plot(x,z)
```

Ahora la variable `a` es una lista que contiene dos objetos gráficos de tipo `Line2D`, y `b` es un sólo objeto gráfico de este tipo. Obsérvese el uso de la coma para almacenar el objeto gráfico y no la lista.<sup>5</sup> Ahora podemos acceder a las diversas propiedades de cada uno de los objetos gráficos usando métodos:

```
a[0].set_linewidth(2)
a[1].set_color('magenta')
b.set_label(r'$\sin x$')
b.set_linestyle('--')
ax2.legend()
b.set_marker('s')
b.set_markerfacecolor('r')
b.set_markersize(3)
plt.draw()
```

El resultado puede verse en la Figura 6.12. Las distintas opciones para el objeto `Line2D` pueden consultarse en la ayuda. Por supuesto, también se pueden emplear las opciones del mismo modo que en la secciones anteriores.

Para tratar distintos elementos de un gráfico, también es posible usar la función `setp` que asigna una (o varias) propiedades a un objeto. Por ejemplo,

```
plt.setp(a, linewidth=3)
plt.setp(ax2, title=u'Título')
```

haría que las dos curvas de eje superior tengan ambas grosor 3, y que el eje inferior luzca un título. Las mismas propiedades que hemos modificado en los ejemplos anteriores pueden modificarse con esta orden.

#### 6.4.1 Un poco más de sofisticación

Veamos algún ejemplo más en el que mostraremos otras propiedades del gráfico que podemos controlar fácilmente. En este caso, vamos a usar el *backend* `notebook` para ver cómo se genera el gráfico en el entorno Jupyter:

```
x = np.linspace(0,np.pi,100)
y = np.sin(2*x*np.pi)*np.cos(3*np.pi*x)
f = plt.figure()
ax = f.add_subplot(111)
b = ax.plot(x,y)
```

<sup>5</sup>Podríamos haber hecho también `a, c = ax1.plot(x,y,x,z)` para almacenar los elementos de la lista en variables separadas.

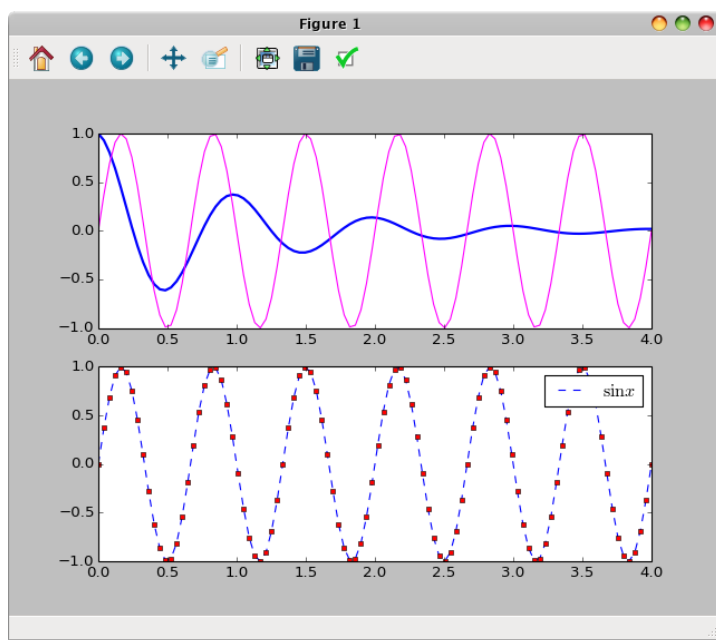


Figura 6.12

```
plt.setp(b,linewidth=2,color='red') # propiedades de la
    curva

ax.axis('tight') # ajuste de los ejes al gráfico
ax.grid(True) # rejilla

# etiquetas del eje X
plt.xticks([0, np.pi/4, np.pi/2,np.pi/4*3, np.pi],['$0$',
    r'\frac{\pi}{4}$', r'\frac{\pi}{2}$', r'\frac{3\pi}{4}$', r'\pi$'])

ax.set_yticks([-1,-.5, 0,.5, 1]) # marcas del eje Y
# etiquetas para las marcas del eje Y
ax.set_yticklabels(['$-1$',r'$-\frac{1}{2}$', r'$0$', r'$\frac{1}{2}$', r'$1$'],fontsize=16,color='blue',rotation
    =30)

# banda de resaltado
band = ax.axvspan(2*np.pi/5,3*np.pi/5)
band.set_color('red') # ponemos color
band.set_alpha(0.2) # ponemos transparencia
```



Vemos ahora que la celda de salida está ocupada por el gráfico de la Figura 6.13.

La función `axis` muestra y/o establece las propiedades de los ejes. En concreto, el argumento `'tight'` hace que los ejes se ajusten a los datos del gráfico. Otras posibilidades son `'off'`, `'equal'` o `'scaled'`.

La orden `grid` activa o desactiva (con `True` o `False`, respectivamente) la malla que puede verse de fondo en el gráfico.

Es posible especificar, no sólo dónde se sitúan las marcas de los ejes, sino también, la etiqueta que lleva cada una. En este ejemplo se ha hecho de forma diferente para cada eje. Con la función `xticks`, que admite una o dos listas, señalamos la posición de las marcas con la primera lista, y, si existe, la cadena de caracteres que se imprimirá en cada etiqueta con la segunda. Nótese el uso de notación  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ .

En el eje *Y* hemos determinado las marcas mediante el método `set_yticks` y las etiquetas con `set_yticklabels`. Esta segunda opción nos permite además especificar color, tamaño de fuente o rotación, entre otras propiedades.

Además hemos añadido un nuevo objeto en el gráfico, una banda vertical de resaltado con la función `axvspan`, a la que hemos modificado el color y la transparencia con los métodos adecuados.

Finalmente, podemos salvar el fichero gráfico a disco con la función

```
f.savefig('grafico.png',format='png')
```

para la que basta precisar el nombre del fichero a guardar y el formato del mismo.<sup>6</sup> Para otras opciones, consultar la ayuda.

## 6 5

### GRÁFICOS 3D

Aunque la librería Matplotlib fue diseñada en principio para trabajar con gráficos bidimensionales también incorpora la posibilidad de realizar gráficos 3D, aunque hemos de señalar que existen otras alternativas interesantes como MayaVi.

Para usar gráficos 3D con Matplotlib precisamos además realizar la siguiente importación:

```
from mpl_toolkits.mplot3d import Axes3D
```

y a continuación, usamos la opción `projection` a la hora de crear unos ejes:

```
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
```

<sup>6</sup>También podemos salvar los gráficos tanto desde la ventana gráfica como desde el gráfico incrustado en el entorno Jupyter.

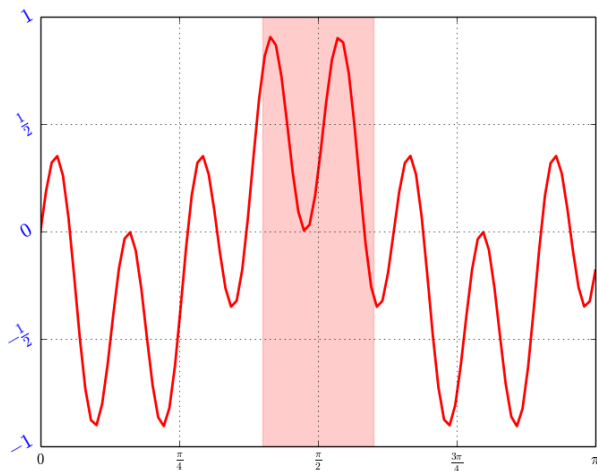


Figura 6.13

Podemos dibujar curvas con el método `plot` vinculado a este tipo de ejes, usando tres listas o arreglos que proporcionan las coordenadas de los puntos de la curva. Por ejemplo,

```
t = np.linspace(-4*np.pi,4*np.pi,100)
z = np.linspace(-2,2,100)
r = z**2+1
x = r*np.sin(t)
y = r*np.cos(t)
b = ax.plot(x,y,z,linewidth=2)
```

da lugar al gráfico de la Figura 6.14

Para dibujar superficies se emplea la misma técnica que en MATLAB, esto es, es necesario crear dos matrices de datos que generan los puntos de una malla bidimensional sobre la que se define la función a dibujar. Por ejemplo, si queremos dibujar el grafo de la función

$$f(x, y) = \sin \left( 2\pi \sqrt{x^2 + y^2} \right)$$

en el dominio  $[-1, 1] \times [-1, 1]$  hemos de preparar los datos de la siguiente forma:

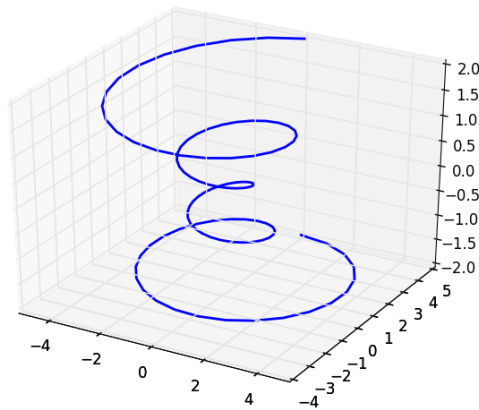


Figura 6.14: Curva en 3D

```
x = np.linspace(-1,1,150)
X1,Y1=np.meshgrid(x,x) # mallado fino
Z1=np.sin(2*np.pi*np.sqrt(X1**2+Y1**2))
y = np.linspace(-1,1,20)
X2,Y2=np.meshgrid(y,y) # mallado grueso
Z2=np.sin(2*np.pi*np.sqrt(X2**2+Y2**2))
```

y ahora dibujamos (véase la Figura 6.15).

```
fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(121,projection='3d')
bx = fig.add_subplot(122,projection='3d')
surf = ax.plot_surface(X1,Y1,Z1)
wire = bx.plot_wireframe(X2,Y2,Z2)
```

Con la orden `contourf` se pueden dibujar mapas de contorno en distintos ejes y diferentes posiciones (Figura 6.16):

```
fig = plt.figure()
ax = fig.gca(projection='3d') # gca: get current axes
ax.plot_wireframe(X2,Y2,Z2)
ax.contourf(X2,Y2,Z2,zdir='z',offset=-1)
cset = ax.contourf(X2,Y2,Z2,zdir='y',offset=1)
plt.setp(cset,alpha=0.2)
```

El parámetro `zdir` señala el eje sobre el que se dibujan los contornos, mientras

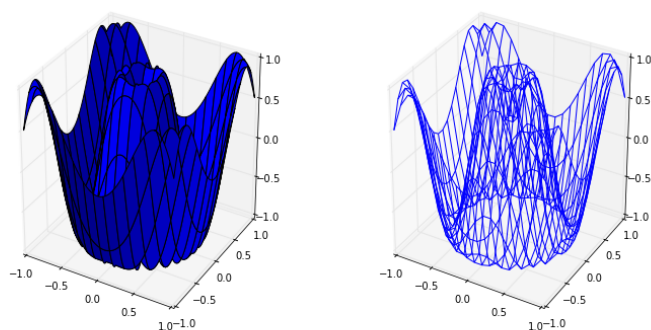


Figura 6.15: Superficies en 3D

que `offset` señala el nivel en el que se muestran (si este parámetro no aparece, se dibuja cada contorno en su nivel correspondiente). Nótese la selección de transparencia sobre el objeto `cset` con el parámetro `alpha` y el comando `setp`.

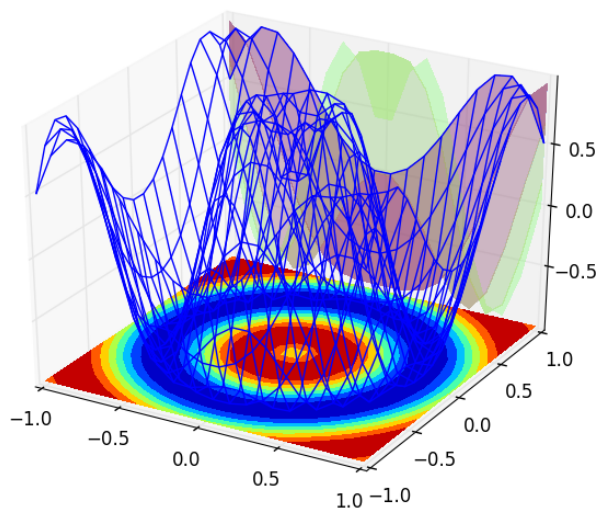


Figura 6.16: Curvas de nivel en 3D

## 6 6

## EJERCICIOS

**E.1** Considera un conjunto aleatorio de 100 puntos en el rectángulo  $[-3, 3] \times [-3, 3]$ . Dibuja de color azul aquéllos que se encuentren dentro del círculo unidad, de color rojo los que se encuentren fuera del círculo unidad y dentro del círculo de radio 2 y dibuja en verde los que están fuera del círculo de radio 2 y dentro de círculo de radio 3. El resto, déjalos en negro. Usando un marcador distinto, determina el más lejano y el más cercano al origen.

Indicación: para dibujar puntos aislados usa el comando `scatter`. El parámetro `s` permite modificar el tamaño del marcador. Usa máscaras para evitar los bucles.

**E.2** En el Ejercicio E.17 del Tema 4 se definen unas matrices  $A$  en función de los parámetros  $k$  y  $d$ . Para  $k = -1000$ , considera 100 valores de  $d$  entre 0 y 100 y dibuja en el plano complejo los autovalores de dichas matrices.

Indicación: los números complejos se pueden dibujar con `scatter` separando parte real y parte imaginaria.

**E.3** Considera la función  $f(x) = \sin(3x) \cos(5x - 1)$  en el intervalo  $[0, 1]$ . Encuentra los máximos y mínimos usando la función `minimize_scalar` del módulo `optimize` de SciPy. Dibuja la función y señala los puntos obtenidos, anotándolos con texto.

Indicación: la función `minimize_scalar` usa una lista a modo de intervalo para acotar el mínimo, aunque no asegura que el mínimo encontrado caiga en dicho intervalo. Usa intervalos adecuados para localizar los máximos y mínimos buscados.

**E.4** Reproducir de la forma más aproximada los gráficos de la Figura E.4.

**E.5** Dibujar las siguientes funciones en los recintos indicados:

(a)  $f(x, y) = e^{-x^2 - y^2}$  en  $[-2, 2]^2$ .

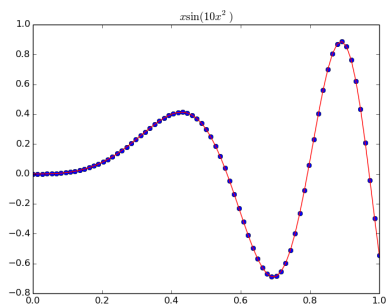
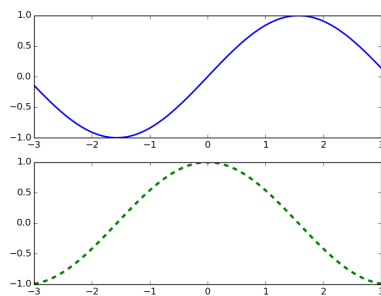
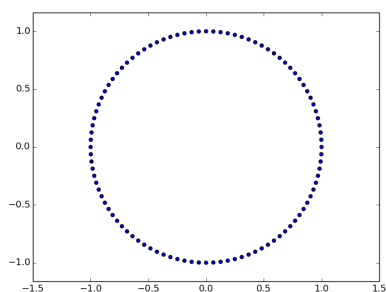
(b)  $f(x, y) = e^{x^2} (x^4 + y^4)$  en  $[-1, 1]^2$ .

(c) El cono  $f(x, y) = \sqrt{x^2 + y^2}$  sobre el círculo unidad. Usar coordenadas polares.

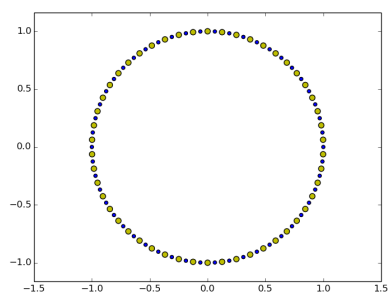
(d) La superficie en polares  $z = (r^2 - 1)^2$  sobre el círculo de centro origen y radio 1.25.

(e) La esfera unidad y la esfera de radio dos. Usar transparencia.

**E.6** Considera los puntos  $(0, 0)$ ,  $(1, 3)$ ,  $(2, -1)$ ,  $(3, 2)$ ,  $(4, 2)$  y  $(5, -1)$ . Dibújalos usando triángulos de color verde. A continuación, calcula la función interpoladora lineal, el spline cúbico y el polinomio interpolador de Lagrange. Dibuja cada uno de ellos en un color distinto y etiquétalos para que aparezca una leyenda.

(a)  $x \sin(10x^2)$ (b)  $\sin x$  y  $\cos x$ 

(c) 100 pts. en el círculo unida d



(d) 100 pts. en el círculo unidad

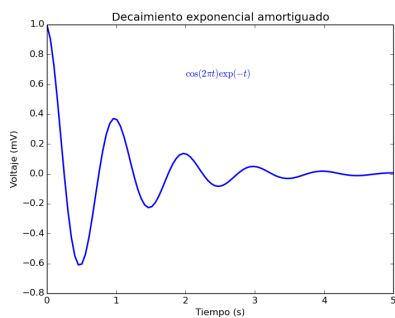
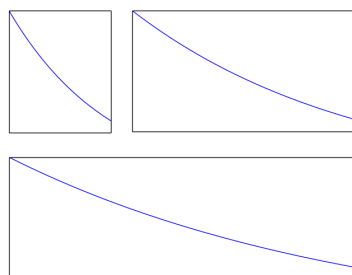
(e)  $\cos(2\pi t)e^t$ (f)  $e^{-x}$  en  $(0, 1)$ 

Figura 6.17: Ejercicio E.4

**E.7** Considera el siguiente código que genera tres líneas  $l_1$ ,  $l_2$  y  $l_3$ :

```
from pylab import *
t1 = linspace(0.0, 2.0, 20)
t2 = linspace(0.0, 2.0, 100)
f = figure(1)
ax = f.add_subplot(111)
l1,      = ax.plot(t2, exp(-t2))
l2, l3 = ax.plot(t2, sin(2*pi*t2), t1, log(1+t1))
```

Realiza las siguientes modificaciones añadiendo nuevas líneas al código:

- Dibuja las líneas  $l_1$  y  $l_3$  con un grosor de 2 puntos, y  $l_2$  con un grosor de 3 puntos.
- Colorea  $l_1$  en azul,  $l_2$  en verde y  $l_3$  en negro.
- La línea  $l_1$  debe ir en discontinua,  $l_2$  con puntos y rayas, y  $l_3$  en línea continua.
- Añade marcadores cuadrados de color verde con bordes rojos en la línea  $l_3$ .





# 7

## Programación Orientada a Objetos

---

En los temas anteriores hemos podido comprobar que en la programación en Python es constante el uso de objetos. En este tema vamos a ver cómo podemos crear nuestros propios objetos junto con sus atributos y métodos mediante las denominadas *clases*.

No es frecuente en computación científica el uso de clases pues, básicamente, la resolución de problemas científicos se basa en la introducción de datos, la realización de los cálculos oportunos y la correspondiente salida; esto es, el comportamiento típico de una función. Sin embargo, vamos a ver cómo los objetos pueden facilitarnos la programación necesaria para resolver un problema.

Por ejemplo, supongamos que queremos estudiar el comportamiento de una estructura de barras como la de la Figura 7.1. Se trata de un conjunto de puntos, denominados *nodos*, que sirven como puntos de unión de una serie de barras de un determinado material. Para describir la estructura precisaremos de las coordenadas en el plano<sup>1</sup> de los nodos y de las conexiones existentes entre éstos, que determinarán dónde se encuentran las barras.

Parece lógico almacenar los datos de la estructura en un par de *arrays*: uno conteniendo las coordenadas de los nodos y otro que almacena las barras existentes entre dos nodos. Por ejemplo, la estructura de la Figura 7.2 estaría definida por:

```
coord = np.array([[ 0., 0.], [ 1., 0.], [ 0., 1.], [ 1., 1.]])
conex = np.array([[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]])
```

Nótese que el número de nodos vendrá dado por `coord.shape[0]`, y el número de barras por `conex.shape[0]`, mientras que los números del *array* de conexiones se refieren a la numeración impuesta en los nodos. Así, por ejemplo, la barra 4 conecta los nodos 1 y 3, (luego `conex[4]=[1, 3]`, cuyas coordenadas vendrán dadas por `coord[2,:]` y `coord[3,:]`).

---

<sup>1</sup>También se podría hacer en el espacio sin dificultad.

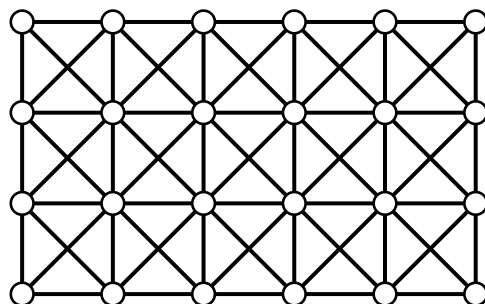


Figura 7.1: Estructura de barras

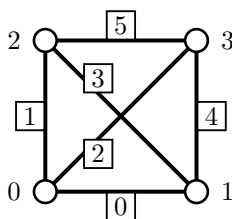


Figura 7.2: Estructura básica

Supongamos ahora que queremos crear una estructura de barras con una configuración regular como la de la Figura 7.1. No es difícil construir una función para obtener los *arrays* de coordenadas y conexiones para una estructura de este tipo. Típicamente se construiría una función cuyos parámetros de entrada fueran el número de nodos que vamos a tener en cada dimensión y cuya salida fueran los *arrays* de coordenadas y conexiones correspondientes.

No obstante, vamos a crear esta estructura mediante *objetos*. Inicialmente podrá parecer que es más complicado proceder de este modo, pero más adelante veremos que merece la pena diseñar la estructura así, pues nos facilitará la implementación de nuevas posibilidades.

## 7.1

### DEFINIENDO CLASES

La Programación Orientada a Objetos requiere de una planificación previa de los elementos que intervienen en el diseño de un objeto. Así, nuestras estructuras están compuestas de nodos y barras. Cada nodo ocupa un punto del plano y están numerados, mientras que cada barra es esencialmente una lista de dos nodos. La propia estructura ocupa unas dimensiones en el plano que habrá que señalar. De este modo, vamos a comenzar definiendo un objeto

sencillo: un punto en el plano.

Para ello vamos a usar las *clases*. Una clase es habitualmente definida como el *molde* de un objeto. Nosotros podemos pensar en las clases como los fragmentos de código que definen un objeto, sus atributos y sus métodos.

Para definir un clase para un objeto *punto* escribiríamos lo siguiente:

```
class Point(object):  
    """  
    describe un punto de coordenadas (x,y)  
    """  
    def __init__(self, xx, yy):  
        self.x = xx  
        self.y = yy
```

Las clases se definen con la palabra clave `class` seguida del nombre asignado a la clase y que define el tipo de objeto, y como parámetros, los objetos de los cuales hereda (veremos el concepto de herencia un poco más abajo).<sup>2</sup> Es una convención ampliamente usada nombrar las clases definidas por el usuario con la inicial en mayúsculas. También es muy conveniente documentar adecuadamente la clase.

Como es habitual en Python, la sangría marcará el fragmento de código correspondiente a la clase. A continuación, aunque en Python no es obligatorio, aparece el denominado *constructor*. Se trata del método `__init__` que se ejecuta cuando la clase se *instancia*. El proceso de instanciación no es más que la definición de un objeto perteneciente a esta clase.

Puesto que `__init__` es un método, esto es, una función, se define como ya vimos con la palabra clave `def`. Los argumentos de los métodos de una clase son un poco peculiares pues el primero de ellos siempre es `self`, que se refiere al propio objeto.<sup>3</sup> El resto de argumentos deberá aparecer en el momento de instanciar al objeto, y los podemos entender como los argumentos de entrada en la creación del objeto.

De este modo, para definir un objeto punto, instanciamos su clase del siguiente modo:

```
p = Point(2.,3.)
```

En principio no hay mucho más que hacer con un objeto de este tipo. Podemos acceder a sus atributos o bien modificarlos:

```
p.x
```

```
2.0
```

```
p.y
```

<sup>2</sup>Por defecto, las clases se definen a través de la herencia con la clase `object`.

<sup>3</sup>Esto es un convenio universalmente aceptado pero podría usarse cualquier otro nombre.

```
3.0
```

```
p.x = 5.
```

Si imprimimos el objeto directamente con la orden `print`:

```
print p
```

```
<__main__.Point object at 0x7f7ae060f110>
```

sólo obtenemos información sobre el mismo. Esto es debido a que no hemos especificado cómo imprimir adecuadamente el objeto. Para hacer esto se define el método `__str__` dentro de la clase:

```
def __str__(self):
    return "{0},{1}".format(self.x, self.y)
```

Ahora (habrá que volver a ejecutar la clase),

```
p = Point(2.,3.)
print p
```

```
(2.0, 3.0)
```

A continuación vamos a construir los objetos de tipo *nodo*. Básicamente este objeto no es más que un punto junto con un identificador. Una primera opción podría ser esta:

```
class Nodo(object):
    """
    describe un nodo mediante identificador y punto
    """
    def __init__(self,n,a,b):
        self.id = n
        self.p = Point(a,b)
```

Entonces,

```
a = Nodo(0,1.,3.)
a.id
```

```
0
```

```
print a.p
```

```
(1.0, 3.0)
```

```
b.p.x
```

```
1.0
```

Sin embargo, dado que hay una gran similitud entre los objetos tipo punto y los objetos tipo nodo, otra opción consiste en apoyarse en el concepto de *herencia*, que no es más que el establecimiento de una relación entre dos clases, de manera que los atributos y métodos de una puedan ser usados en la otra. En nuestro caso es evidente que los atributos `x` e `y` de la clase `Point` deberán mantenerse en la nueva clase que vamos a crear, por lo que podemos aprovecharnos del constructor de la clase `Point` usando el comando `super`:

```
class Node(Point):
    """
    clase que hereda de la clase Point
    """
    numberNode = 0
    def __init__(self, xx, yy):
        super(Node, self).__init__(xx, yy)
        self.id = Node.numberNode
        Node.numberNode += 1
```

Como podemos observar, en la definición de la clase se hace referencia a la clase de la que se hereda, denominada clase *padre*. El constructor de la clase `Node` llama al constructor de la clase `Point` a través de la orden `super`, es decir, realiza una instanciación de la clase de la que hereda, en este caso un objeto `Point`. Ahora incluimos también un atributo para identificar al nodo que funciona automáticamente a través de un contador `numberNode`, de manera que cada nuevo nodo que creemos tendrá asignado un identificador en orden creciente. Si no hubiéramos definido un método `__init__` para esta clase, se habría usado el método de la clase padre de la que hereda. Ahora podemos hacer

```
a = Node(1.,2.)
b = Node(0.,1.)
print a
```

```
(1.0,2.0)
```

```
a.id
```

```
0
```

```
b.id
```

```
1
```

Nótese que para la impresión del objeto `Node` se está usando el método `__str__` de la clase `Point`. Si quisiéramos una impresión distinta habría que definir nuevamente el método `__str__` para esta clase.

Ahora no debe ser difícil para el lector entender la clase para las barras siguiente:

```
class Bar(object):
    """
    define una barra soportada por dos nodos
    """
    def __init__(self, n1, n2):
        if n1.id == n2.id:
            print "Error: no hay barra"
            return
        elif n1.id < n2.id:
            self.orig = n1
            self.fin = n2
        else:
            self.orig = n2
            self.fin = n1
    def __str__(self):
        return "Barra de extremos los nodos {0} y {1}".
            format(self.orig.id, self.fin.id)
```

Al constructor hemos de proporcionarle dos nodos de diferente identificador, pues en caso contrario no habría barra. Definimos los atributos `orig` y `fin` como los nodos que conforman la barra y convenimos en señalar como nodo origen aquél cuyo identificador sea menor. De este modo, la instanciación de una barra se haría de la forma siguiente:

```
barra = Bar(a,b)
print barra
```

Barra de extremos los nodos 0 y 1

Con esto hemos definido los elementos esenciales que participan en la construcción de una estructura de barras como la de la Figura 7.1. Ahora vamos a construir la estructura, que como comentamos al inicio, consta esencialmente de nodos y barras. Los parámetros de entrada podrían ser dos puntos que determinen el rectángulo sobre el que crear la estructura, junto con el número de nodos a usar en cada dimensión.

Una posibilidad vendría dada por el siguiente código:

```
class Truss(object):
    """
    genera una estructura rectangular de barras
    - nx: numero de nodos en abscisas.
```

```

- ny: numero de nodos en ordenadas.
- p1: vértice inferior izquierdo (clase punto).
- p2: vértice superior derecho (clase punto).

genera 6 barras entre 4 nodos
"""
def __init__(self, p1, p2, nx, ny):
    # comprobación de la integridad del rectángulo
    if p2.x-p1.x < 1.e-6 or p2.y-p1.y < 1.e-6:
        print "Rectángulo incorrecto"
        return

    self.nNodos = (nx + 1) * (ny + 1)
    self.nBarras = 4*nx*ny+ny+nx
    self.nodos = []
    self.barras = []

    Node.numberNode = 0

    # construcción de nodos
    nodx = np.linspace(p1.x, p2.x, nx+1)
    nody = np.linspace(p1.y, p2.y, ny+1)

    for yy in nody:
        for xx in nodx:
            self.nodos.append(Node(xx, yy))

    # construcción de barras
    for j in range(ny):
        for i in range(nx):
            n1 = i+ j*(nx+1)
            n2 = n1 + 1
            n3 = n1 + nx + 1
            n4 = n3 + 1
            # barras en cada elemento
            b1 = Bar(self.nodos[n1], self.nodos[n2])
            b2 = Bar(self.nodos[n1], self.nodos[n3])
            b3 = Bar(self.nodos[n1], self.nodos[n4])
            b4 = Bar(self.nodos[n2], self.nodos[n3])
            self.barras.extend([b1, b2, b3, b4])
            # barras finales a la derecha
            self.barras.append(Bar(self.nodos[n2], self.
                                  nodos[n4]))

    # barras de la línea superior
    indice=ny*(nx+1)+1
    for j in range(nx):
        self.barras.append(Bar(self.nodos[indice+j-1],

```

```
self.nodos[indice+j]))
```

Nótese que hemos definido un par de listas: **nodos** y **barras** en las que almacenar los elementos que nos interesan. Ponemos el contador del identificador de nodos a cero, de manera que cada vez que tengamos una estructura, los nodos se creen comenzando con el identificador en 0. Tal y como está construido, el identificador de cada nodo coincide con el índice que ocupa en la lista **nodos**, lo que nos simplifica la búsqueda de los nodos.

Para obtener los nodos y las barras disponemos de las listas anteriores, pero será más cómodo si definimos unos métodos que nos proporcionen directamente la información que realmente queríamos precisar de la estructura, esto es, las coordenadas de los nodos y los índices correspondientes a cada barra.

Por ello, a la clase anterior le añadimos los siguientes métodos:

```
def get_coordinate(self):
    coordenadas=[]
    for nod in self.nodos:
        coordenadas.append([nod.x,nod.y])
    return np.array(coordenadas)

def get_connection(self):
    conexiones=[]
    for bar in self.barras:
        conexiones.append([bar.orig.id,bar.fin.id])
    return np.array(conexiones)
```

La estructura más sencilla que podemos montar, correspondiente a la Figura 7.2, sería:

```
a = Point(0.,0.); b = Point(1.,1.)
m = Truss(a,b,1,1)
m.get_coordinate()
```

```
array([[ 0.,  0.],
       [ 1.,  0.],
       [ 0.,  1.],
       [ 1.,  1.]])
```

```
m.get_connection()
```



```
array([[0, 1],
       [0, 2],
       [0, 3],
       [1, 2],
       [1, 3],
       [2, 3]])
```

Es evidente que podríamos haber creado una función que tuviera como entrada las coordenadas de los puntos del rectángulo y el número de nodos a usar en cada dimensión, y cuya salida fuera precisamente los dos *arrays* que hemos obtenido; posiblemente hubiera sido incluso más sencillo de implementar. Sin embargo, como ahora veremos, es más conveniente el uso de clases porque nos va a permitir una flexibilidad aun mayor.

### 7.1.1 Modificando clases

Supongamos que ahora queremos dibujar la estructura obtenida. Si hubiéramos implementado una función tendríamos dos opciones: o bien modificamos la función creada para añadirle la parte gráfica, o bien implementamos la parte gráfica en una función aparte, que reciba los *arrays* que definen la estructura y los dibuje.

La primera opción puede resultar un engorro, pues cada vez que ejecutemos la función obtendremos los *arrays* y el gráfico y habrá ocasiones en las que queramos crear sólo la información de la estructura y otras en las que sólo queramos dibujar. La segunda opción nos obliga a llamar primero a la función para obtener los *arrays* de coordenadas y conexiones, y luego pasarlos a la nueva función para dibujar.

Sin embargo, implementar un nuevo método dentro de la clase para que construya el gráfico es mucho más cómodo, pues podremos invocarlo independientemente de que construyamos o no los *arrays* de coordenadas y conexiones. Podríamos añadir a la clase **Truss** algo así:

```
def plotting(self):
    plt.ion()
    fig = plt.figure()
    bx = fig.add_subplot(111)

    for bar in self.barras:
        bx.plot([bar.orig.x, bar.fin.x], [bar.orig.y, bar
            .fin.y], 'k-o', linewidth=2)
    bx.axis('equal')
    plt.show()
```

De este modo, una vez creada una estructura, nos bastará con invocar al método `plotting` para obtener el gráfico correspondiente.

Algo similar ocurre si queremos modificar la estructura eliminando alguna barra: bastará con implementar un método adecuadamente:

```
def remove_bar(self, n1, n2):
    if n2 < n1:
        n1, n2 = n2, n1
    elif n1 == n2:
        print "Nodos incorrectos"

    for bar in self.barras:
        if bar.orig.id == n1 and bar.fin.id == n2:
            self.barras.remove(bar)
            self.nBarras -= 1
            return
    else:
        print "No existe tal barra"
```

¿Se imagina el lector qué habría ocurrido si hubiéramos implementado una función que incluyera la parte gráfica a la vez que la creación de la estructura? La eliminación a posteriori de barras nos hubiera impedido dibujar la estructura resultante. Con las clases, simplemente hemos de ir añadiendo métodos que se ejecutan de forma separada sobre el mismo objeto.

## 7.2

### CONTROLANDO ENTRADAS Y SALIDAS

Veamos un segundo ejemplo de la utilidad de usar clases en la programación de algoritmos matemáticos. En este caso vamos a implementar los clásicos métodos de Jacobi y Gauss-Seidel para la resolución iterativa de sistemas de ecuaciones lineales.

Dado un sistema de ecuaciones lineales de la forma  $A\mathbf{x} = \mathbf{b}$ , donde  $A$  es una matriz cuadrada de orden  $n$  y  $\mathbf{x}$  y  $\mathbf{b}$  son vectores de  $n$  componentes, un método iterativo para resolver este sistema se puede escribir de la forma:

$$\mathbf{x}^{(k+1)} = M\mathbf{x}^{(k)} + \mathbf{c}, \quad \text{con } \mathbf{x}^{(0)} \text{ dado}, \quad (7.1)$$

donde  $M$  y  $\mathbf{c}$  son una matriz y un vector, respectivamente, que definen el método a usar. En concreto, si realizamos una descomposición de la matriz  $A$  de la forma

$$A = D + L + U$$

donde  $D$  es una matriz diagonal,  $L$  es triangular inferior y  $U$  es triangular superior, entonces el método de Jacobi se define mediante

$$M = D^{-1}(-L - U), \quad \mathbf{c} = D^{-1}\mathbf{b},$$

mientras que el método de Gauss-Seidel se escribe con

$$M = (D + L)^{-1}(-U), \quad \mathbf{c} = (D + L)^{-1}\mathbf{b}.$$

Es sencillo crear una función cuyos parámetros de entrada sean la matriz  $A$  y el segundo miembro  $b$ , y que devuelva la solución del método iterativo escogido. La matriz  $M$  y el vector  $c$  se pueden obtener mediante funciones independientes,

```
def jacobi(A,b):
    D = np.diag(np.diag(A))
    L = np.tril(A,-1)
    U = np.triu(A,1)
    M = np.dot(np.linalg.inv(D),(-L-U))
    c = np.dot(np.linalg.inv(D),b)
    return M,c
def seidel(A,b):
    D = np.diag(np.diag(A))
    L = np.tril(A,-1)
    U = np.triu(A,1)
    M = np.dot(np.linalg.inv(D+L),(-U))
    c = np.dot(np.linalg.inv(D+L),b)
    return M,c
```

y el método iterativo queda:

```
def iterativo(A,b,metodo=jacobi):
    x0 = np.zeros(A.shape[0])
    eps = 1.e-8
    M,c = metodo(A,b)
    x = np.ones_like(x0)
    k=0
    while np.linalg.norm(x-x0)>eps*np.linalg.norm(x0):
        x0 = x.copy()
        x = np.dot(M,x0) + c
        k+=1
    print "Iteraciones realizadas:",k
    return x
```

El código es sencillo y no necesita mucha explicación. Se define el vector inicial  $\mathbf{x}^{(0)}$ , el parámetro  $\varepsilon$  y las matrices que definen el método, y se realiza la iteración descrita en (7.1) hasta obtener que

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon \|\mathbf{x}^{(k)}\|$$

y se imprime el número de iteraciones realizadas. Podemos comprobar el funcionamiento del código en el siguiente ejemplo:

$$\begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 6 \\ 25 \\ -11 \\ 15 \end{pmatrix}$$

```
A = np.array
    ([[10,-1,2,0.],[-1,11,-1,3],[2,-1,10,-1],[0,3,-1,8]])
b = np.array([6.,25,-11,15])
```

```
iterativo(A,b)
```

```
Iteraciones realizadas: 23
array([ 1.,  2., -1.,  1.])
```

```
iterativo(A,b,seidel)
```

```
Iteraciones realizadas: 10
array([ 1.,  2., -1.,  1.])
```

En principio, sería un código que cumple a la perfección con nuestros propósitos iniciales. No obstante, vamos a implementar una versión del mismo algoritmo usando clases:

```
class Resolucion(object):
    def __init__(self,A,b,metodo=jacobi):
        self.A = A
        self.b = b
        self.metodo = metodo
        self.x0 = np.zeros(self.A.shape[0])
        self.eps = 1.e-8
    def iteracion(self):
        self.k=0
        M,c = self.metodo(self.A,self.b)
        x0 = self.x0
        x = np.ones_like(x0)
        while np.linalg.norm(x-x0) > self.eps*np.linalg.
            norm(x0):
            x0 = x.copy()
            x = np.dot(M,x0) + c
            self.k += 1
        return x
```

Podemos ver que la clase `Resolucion` tiene dos métodos: el constructor, que realiza la inicialización de datos y el método `iteracion` que lleva a cabo las iteraciones, de igual modo que antes. A diferencia del código anterior, aquí no se imprime el número de iteraciones. Para ejecutar ahora el algoritmo mediante la clase anterior escribiremos:

```
a = Resolucion(A,b)
a.iteracion()
```

```
array([ 1.,  2., -1.,  1.])
```

Aunque no hemos impreso el número de iteraciones, lo podemos obtener usando el atributo `k`:

```
a.k
```

```
23
```

¿Cuál es la ventaja de usar la clase frente a la función? Obviamente, ambos códigos hacen lo mismo, pero como vamos a ver a continuación, la clase es mucho más flexible. Por ejemplo, si incluimos la función como parte de algún otro código y ésta es ejecutada muchas veces, posiblemente hubiera sido más conveniente no haber incluido la impresión del número de iteraciones pues ahora nos ensuciará la salida del otro código. Por supuesto que podemos hacer una versión de la función sin la impresión de las iteraciones, pero eso supone tener que mantener varias versiones del mismo código. Cuando eso ocurre es frecuente que, al cabo de un tiempo, el programador se encuentre perdido entre tantas versiones.

Otra ventaja está en la facilidad para volver a correr el algoritmo con distintos parámetros. Por ejemplo, para correr el método de Gauss-Seidel no es necesario crear un nuevo objeto, nos bastaría con:

```
a.metodo=seidel
a.iteracion()
```

```
array([ 1.,  2., -1.,  1.])
```

```
a.k
```

```
10
```

Si quisiéramos cambiar el parámetro  $\varepsilon$  en la función, tendríamos que modificar el código directamente. Ahora con la clase lo podemos modificar desde fuera:

```
a.eps=1.e-4
a.iteracion()
```

```
array([ 1.00000538,  2.00000122, -1.00000183,
        0.99999931])
```

```
a.k
```

```
6
```

Por ejemplo podemos llevar a cabo una comparativa de precisión y número de iteraciones en cada método:

```

for m in [jacobi,seidel]:
    a.metodo=m
    print a.metodo.__name__
    for eps in [10**x for x in range(-2,-13,-2)]:
        a.eps=eps
        b=a.iteracion()
        print "Precisión: {0:2.0e} --- Iteraciones: {1:3d}
              {}".format(eps,a.k)

```

```

jacobi
Precisión: 1e-02 --- Iteraciones: 7
Precisión: 1e-04 --- Iteraciones: 12
Precisión: 1e-06 --- Iteraciones: 18
Precisión: 1e-08 --- Iteraciones: 23
Precisión: 1e-10 --- Iteraciones: 28
Precisión: 1e-12 --- Iteraciones: 34
seidel
Precisión: 1e-02 --- Iteraciones: 3
Precisión: 1e-04 --- Iteraciones: 6
Precisión: 1e-06 --- Iteraciones: 8
Precisión: 1e-08 --- Iteraciones: 10
Precisión: 1e-10 --- Iteraciones: 11
Precisión: 1e-12 --- Iteraciones: 13

```

Obviamente podríamos haber hecho algo similar con la función definiendo oportunamente los parámetros de entrada para dotarla de más flexibilidad, pero una vez más tendríamos que modificar el código de la misma. Éste es precisamente el hecho que queremos resaltar en cuanto a la ventaja de usar clases en lugar de funciones para la programación científica: si diseñamos adecuadamente los atributos y métodos de la clase, disponemos de acceso completo a los mismos, tanto para introducir como para extraer datos. No sólo eso; además, las modificaciones que realicemos sobre la clase no tienen por qué afectar al constructor, por lo que podemos seguir usándola del mismo modo sin necesidad de mantener diversas versiones del mismo código.

## 7.3

### EJERCICIOS

**E.1** Redefine el constructor de la clase `Bar` de la sección 7.1 de manera que al instanciar un nuevo objeto se imprima la información del objeto creado. Por ejemplo, debería funcionar del siguiente modo:

```
n1 = Node(0,0.,1.)
n2 = Node(1,1.,2.)
barra = Bar(n1,n2)
```

Se ha creado una barra de extremos (0.0,1.0) y (1.0,2.0)

**E.2** Para la clase **Truss** de la sección 7.1, escribir un método para que el comando **print** proporcione información sobre el número de nodos y el número de barras de la estructura.

**E.3** Define una clase que contenga dos métodos: **getString** con el que obtener una cadena de texto introducida por teclado y **printString** que imprima la cadena obtenida en mayúsculas, dejando un espacio de separación entre cada letra. Debería funcionar del siguiente modo:

```
a = InputOutString()
a.getString()
```

← entrada por teclado

```
a.printString()
```

H O L A