Índice general

1 Introducción a Python

Python es un lenguaje de programación creado por Guido Van Rossum entre finales de los ochenta y principios de los noventa, apareciendo su primera versión estable en 1994. 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. Además es un lenguaje multiparadigma que soporta programación imperativa, programación orientada a objetos, y en menor medida, programación funcional.

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 continúa 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 que no son completamente compatibles. La mayor parte del código que se encuentra escrito en Python sigue las especificaciones de la versión 2, aunque hace ya algunos años que la versión 3 se encuentra disponible. Esta versión fue diseñada para rectificar algunos defectos fundamentales del diseño del lenguaje que no podían ser implementados manteniendo la compatiblidad con la versión 2. En estas notas se usará la versión 3 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`. Sin embargo, diversos módulos de interés, y entre ellos, los dedicados a la programación científica que veremos en los capítulos 4, 5, 6 y 7, 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 por defecto 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.anaconda.com/download`

Allí encontraremos descargas para los sistemas Linux, Windows y Mac en versiones para 32 y 64 bits, y en la versión 2.7 o 3.6 de Python (en el momento de escribir estas notas). Como hemos comentado antes, aquí usaremos la versión 3 de Python, por lo que habría que descargar el instalador para la versión 3.6. 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 Mac) 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 encarecidamente 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,[1] obteniéndose algo por el estilo:

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017,
    13:09:58)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
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 función `print` imprime la *cadena de texto* o *string* `Hola Mundo` que aparece como argumento, y 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 la orden

```
print("Hola Mundo")
```

en un archivo,[2] lo salvamos con un nombre cualquiera, por ejemplo `hola.py`, en el que la extensión ha de ser `.py`.[3] Podemos ejecutar el código sencillamente

[1] 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.

[2] Para diferenciar la escritura de órdenes en el intérprete de los comandos que introduciremos en los archivos los ilustraremos con fondos de diferente color.

[3] 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`.

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[4]

```
#!/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.[5]

Cuando queremos escribir una orden de longitud mayor a una línea debemos usar el carácter de escape `\` como continuación de línea, 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
```

Python usa el salto de línea como fin de sentencia a menos que haya paréntesis, corchetes, llaves o triples comillas abiertas, en cuyo caso no es necesario el carácter de escape. Por ejemplo,

```
>>> (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* o *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.

[4] 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*.

[5] En muchos sistemas el intérprete Python por defecto es el de la versión 2, por lo que habría que escribir `python3` en lugar de `python` en la línea del *shebang*.

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 facilitan el trabajo con el intérprete. Entre ellas podemos destacar la presencia del *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:[6]

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017,
    13:09:58)
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- 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]`. En cualquier consola Python, el último valor obtenido siempre puede usarse mediante `_`,

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

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

[6] Desde `Anaconda Navigator` disponemos de esta misma terminal a través de `qtconsole`.

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* (*magic functions*) que proveen de funcionalidades especiales a la consola. Estos comandos comienzan por el carácter `%` aunque si no interfieren con otros nombres dentro del entorno 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:

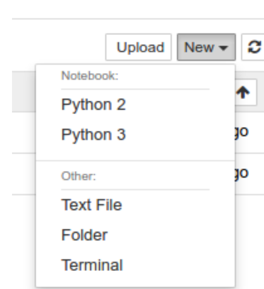
```
In [5]: run hola.py
Hola Mundo
```

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 variante 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 aúna 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 podemos escribir en una terminal la orden `jupyter-notebook`,[7] lo que nos abrirá una ventana en un navegador web 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.



Nuevo *Notebook*

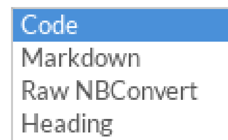
Si no disponemos de un *notebook* previo, podemos crear uno nuevo pulsando sobre el desplegable New (arriba a la derecha), eligiendo el tipo deseado, en nuestro caso Python 3 (véase la figura adjunta). Esto abre automáticamente una nueva ventana del navegador a la vez que crea un nuevo fichero `Untitled`

[7] O lanzarlo a través de `Anaconda Navigator`.

con extensión `.ipynb` en la carpeta donde nos encontremos. 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.

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,[8] o incluso texto en formato LaTeX, 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, entre otros.

[8] *Markdown* es un lenguaje de marcado ligero que permite formatear de forma fácil y legible un texto.

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 pulsando y. 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 Upload 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 Help › Keyboard shortcuts.

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[1] y en el que se hace la distinción entre mayúsculas y minúsculas, y son definidas mediante el operador de asignación `=`. No están permitidos las palabras reservadas de la Tabla 2.1.[2]

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
```

[1]Específicamente, el primer carácter de un identificador ha de ser cualquier carácter considerado como letra en Unicode. También es posible usar el guión bajo (o *underscore*) `_` aunque éste suele estar reservado para identificadores con un significado especial. El resto de caracteres pueden ser letras, dígitos o *underscore* de Unicode.

[2]Tampoco es recomendable usar ninguno de los nombres que se obtienen al ejecutar la sentencia `dir(__builtins__)`. Véase la definición de la función `dir` en la sección 2.2.

Tabla 2.1: Palabras reservadas

and	continue	except	global	lambda	pass	while
as	def	False	if	None	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	True	
class	else	from	is	or	try	

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 sin usar ningún operador en medio, y el uso de la función `complex`. Nótese también que la asignación de una variable no produce ninguna salida. También podemos ver en el ejemplo el uso del carácter `#` para introducir comentarios en un código Python.

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

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

```
<class 'int'> <class 'float'> <class 'complex'>
```

Como vemos, Python asigna el tipo (o *clase*) a cada variable en función de su definición. Nótese también el uso de la coma con la función `print`.[3]

2.1.2 Operadores aritméticos

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 números, y el operador `%` (módulo), que proporciona el resto de la división entre dos números.

Asimismo es importante destacar el carácter *fuertemente tipado* del lenguaje Python, que puede observarse en los siguientes ejemplos:

```
a = 5; b = 3
print(a//b)
```

```
1
```

esto es, la división entera entre 5 y 3 es 1, como número entero.[4] Sin embargo,

[3] Por defecto, la coma introduce un espacio entre los argumentos de la función, que corresponde al valor por defecto del parámetro `sep`.

[4] Nótese también el uso del `;` para escribir más de una sentencia en la misma línea. No obstante, su uso no es recomendado porque el código pierde legibilidad.

```
c = 5.
d = 3.
print(c//d)
```

```
1.0
```

da como resultado 1, como número real. Esto se debe a que el resultado se expresa en el mismo tipo que los datos con los que se opera. Así pues, el lector podrá entender el porqué del siguiente resultado:

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

```
2.0
2
```

Nótese el uso de la función de conversión a entero `int`.[5] La única excepción a esta regla ocurre con la división. Si escribimos

```
print(5/3)
```

```
1.6666666666666667
```

el resultado, lógicamente, no es un entero.[6]

Finalmente, cuando Python opera con números de distinto tipo, realiza la operación transformando todos los números involucrados al mismo tipo, según una jerarquía de tipos 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)
<class 'complex'>
```

Existen también *operadores aumentados de asignación* cuyo significado permite abreviar expresiones como

```
a = a + b
```

del siguiente modo:

[5] La función `float` hace la conversión a número real.

[6] Por el contrario, en Python 2, el resultado de esa misma operación es 1, como número entero. Esta es una de las diferencias importantes entre las dos versiones.

```
a += b
```

Están presentes para el resto de operadores aritméticos. No obstante, como veremos posteriormente en la sección 2.5, en determinados casos estas dos últimas expresiones no son completamente equivalentes.

2.1.3 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.

Nótese en los dos últimos ejemplos que hemos prescindido de la función `print` para obtener los resultados. Esto es debido a que el entorno Jupyter Notebook funciona de forma similar a una consola, devolviendo el resultado de una expresión sin necesidad de imprimirla explícitamente. Sin embargo, si tenemos más de una sentencia en la misma celda, sólo aparecerá el resultado de la última evaluación. Este comportamiento por defecto puede ser modificado si escribimos en una celda de un *notebook* lo siguiente:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

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 4 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)
```

```
list
```

Como podemos ver, 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.

```
print(a[1])
print(a[4])
```

```
2.0
---------------------------------------------------------------------------
IndexError                                Traceback (
    most recent call last)
<ipython-input-2-7efc04aa6ae5> in <module>()
      1 print(a[1])
----> 2 print(a[4])

IndexError: list index out of range
```

Sin embargo, 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 éste se ha producido, el tipo de error (`IndexError`, en este caso) 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
```

También disponemos de la función `len` para obtener la longitud de una lista:

```
len(a)
```

```
4
```

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. Los siguientes ejemplos son autoexplicativos y muestran 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+=b` usando el operador de asignación aumentado.

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. Un error bastante frecuente consiste en asignar la salida de un método a una nueva variable. Por ejemplo,

```
b = a.reverse()
print(b)
```

```
None
```

Como podemos observar, `b` es el resultado de la llamada al método, que no retorna ningún valor, de ahí la aparición de `None` (volveremos a esto en la sección 2.4). Lo que ha ocurrido es que `a` se ha modificado en memoria. Si lo que queremos es conservar la lista original y la invertida, deberíamos proceder así:

```
a = [1,2,3,10,20,30]
b = a[:]
b.reverse()
print(a)
print(b)
```

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

Es imprescindible realizar la copia de `a` de la forma `b = a[:]` como posteriormente explicaremos en la sección 2.5.

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]
```

Nótese la diferencia con:

```
a[1:2] = [20,30]
print(a)
```

```
[9, 20, 30, 1, 2, 6, −1, −2, −3, −4, 2, 1, 0]
```

en el que hemos reemplazado el elemento que opupa la posición 1.

Si queremos añadir al inicio, escribiremos:

```
a[:0] = a[-5:-1]
print(a)
```

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

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

```
[]
```

2.1.5 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
<class 'str'>
```

en las que se puede comprobar que da igual definirlas con comillas simples o dobles, siempre que empiecen y terminen por el mismo carácter, lo que es útil si queremos cadenas que incluyan estos caracteres:

```python
a = "Esto es una 'string'"
print(a)
```

```
Esto es una 'string'
```

En caso de necesitar ambos caracteres en la cadena, podemos *escapar* los símbolos con `\'` o `\"`:

```python
a = "Cadena que contiene '\"'"
b = 'Y ahora con "\'"'
print(a)
print(b)
```

```
Cadena que contiene '"'
Y ahora con "'"
```

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

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

```
Esto es un cadena
muy larga que tiene
muchas líneas
```

Y si queremos crear una cadena muy larga sin necesidad de usar el carácter de continuación, podemos hacerlo del siguiente modo:

```python
a = ("Cadena expandida "
     "en varias líneas "
     "pero impresa en una")
print(a)
```

```
Cadena expandida en varias líneas pero impresa en una
```

Los paréntesis hacen el efecto de operador de concatenación.

En Python 3, todas las cadenas contienen caracteres Unicode, y la codificación por defecto es `UTF-8`. Si en algún momento necesitamos que la cadena de caracteres sea interpretada tal cual (sin caracteres de escape), debemos usar una `r` (por *raw string*) precediendo a la cadena:

```python
cadena = r'\nhola'
print(cadena)
```

```
\nhola
```

Como podemos ver, la orden `print` interpreta la secuencia `\n` de forma literal, lo cual puede ser útil en determinadas circunstancias. Sin embargo,

```
cadena = '\nhola'
print(cadena)
```

```

hola
```

ahora el carácter `\n` se ha interpretado como un salto de línea.

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

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

```
H
' '
```

pero las cadenas son *inmutables*, esto es, no es posible alterar sus elementos (veremos este asunto más adelante en la sección 2.5):

```
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, sino que devolverá una nueva cadena.

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
```

Al igual que con las listas, la función `len` proporciona la longitud de la cadena de caracteres, y los operadores `+` y `*` (multiplicación por un entero) funcionan como concatenación y replicación de la cadena, respectivamente.

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, entre otros muchos. Emplazamos al lector a usar la ayuda en línea para aprender el funcionamiento de esos métodos.

2.1.6 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 contenedores:

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

```
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 del diccionario es irrelevante pues la indexación no es numerada, y además no se puede predecir.

Alternativamente, podemos usar la función `dict` actuando sobre una secuencia de pares para definir un diccionario:

```
colores = dict([['k','negro'], ['r','rojo'], ['b','azul'],
    ['g','verde']])
```

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
```

```
dict_keys(['r', 'k', 'g', 'b'])
dict_values(['rojo', 'negro', 'verde', 'azul'])
```

que devuelven *vistas*[7] de los objetos que contienen las claves y los valores.

A veces es útil disponer de una lista con las claves o los valores, que se puede obtener con la función `list`:

```
claves = list(colores.keys())
valores = list(colores.values())
print(claves)
print(valores)
```

```
['r', 'k', 'g', 'b']
['rojo', 'negro', 'verde', 'azul']
```

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

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

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

o el método `clear` que elimina todos los elementos del diccionario:

```
colores.clear()
print(colores)
```

```
{}
```

2.1.7 Conjuntos

Los conjuntos en Python son estructuras de datos desordenadas que no permiten repetición de elementos, y además éstos han de ser datos inmutables (véase la sección 2.5). Son especialmente útiles cuando queremos eliminar repeticiones de datos en otras estructuras. Se pueden definir usando llaves

```
colors = {'green', 'blue', 'red', 'green', 'yellow'}
print(colors)
```

```
{'yellow', 'green', 'red', 'blue'}
```

Obsérvese cómo la repetición de elementos es eliminada y cómo el orden es aleatorio. Al igual que los diccionarios, no es posible predecir el orden en el que se accede a los elementos. El tipo de un conjunto es

[7] Una vista de un objeto es otro objeto que está dinámicamente enlazado con el principal, de manera que una modificación de éste produce una modificación automática de la vista.

```
print(type(colors))
```

```
<class 'set'>
```

La función `set` usada sobre una secuencia también define un conjunto:

```
a = set('abracadabra')
print(a)
```

```
{'r', 'b', 'd', 'c', 'a'}
```

y es la forma más cómoda de encontrar los elementos no repetidos de cualquier estructura de datos (en el caso anterior, equivale a las letras únicas de la cadena `'abracadabra'`).

Nótese que la asignación

```
d = {}
```

define un diccionario, mientras que si queremos definir el conjunto vacío hemos de emplear la sentencia

```
a = set()
```

Los conjuntos también disponen de una interesante colección de métodos que pueden ser consultados por el lector usando la ayuda en línea.

2.1.8 Tuplas

Otro de los tipos de datos principales en Python son 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")
type(a)
```

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

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

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

En realidad los paréntesis no son necesarios, por lo que también es admisible la definición:

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

```
tuple
```

Atención a la creación de tuplas de un único elemento (véase el ejercicio E2.4).

Este tipo de definiciones permite el *empaquetado* de un conjunto de datos en un único objeto. Lo interesante del empaquetado es que ahora podemos *desempaquetar*:

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

```
1
2
hola
```

Y este procedimiento puede ser llevado a cabo de forma simultánea, lo que da lugar a 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, por ejemplo, para intercambiar valores sin necesidad de usar una variable auxiliar,

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

```
1 0
```

Es importante resaltar en qué orden se evalúan y asignan las variables en la asignación múltiple. Primero se evalúa la parte derecha de la expresión, en orden de izquierda a derecha, y luego se realiza la asignación de uno en uno, siguiendo el mismo orden. El siguiente ejemplo es una muestra de ello:

```
i, k, x = 1, 2, [0,1,2,3]
i, x[i] = i+k, i
print(x)
```

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

En estas sentencias es necesario que el número de objetos a empaquetar coincida con el número de variables:

```
a, b = 1,2,3
```

```
ValueError: too many values to unpack (expected 2)
```

pero es posible empaquetar de forma más genérica:[8]

```
a, *b = 1,2,3
print(a,b)
```

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

o también

```
a, *b, c, d = 1,2,3,4,5,6
print(a,b,c,d)
```

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

El empaquetado y desempaquetado de objetos funciona de forma idéntica con las listas.

Como veremos luego, las tuplas son útiles para pasar y devolver un conjunto de datos a una función.

2.1.9 Booleanos y comparaciones

En Python disponemos de las variables booleanas `True` y `False` y la función `bool` que convierte un valor a booleano, siempre y cuando tenga sentido tal conversión. En Python, al igual que en C, cualquier número distinto de 0 es verdadero, mientras que `0`, `0.0` y `0j` son falsos. Además, la variable `None` y las listas, cadenas, diccionarios, conjuntos o tuplas vacías son falsas; el resto son verdaderas.]

[8]Esta característica no está presente en Python 2.

```
a = []
b = 1,
c = 'a'
print(bool(a))
print(bool(b))
print(bool(c))
```

```
False
True
True
```

Por otro lado, hay que tener en cuenta que la clase `bool` es una subclase de los números enteros, lo que explica el siguiente comportamiento:

```
a = True
a + a
```

```
2
```

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`, que pueden ser aplicados a datos que no sean expresiones booleanas. Por ejemplo:

```
a = 5
b = 'y'
print(a and b)
print(b and a)
print(a or b)
```

```
y
5
5
```

donde podemos observar que el resultado corresponde al operando que determina su valor. También está permitida la triple comparación

```
2 < 3 <= 5
```

```
True
```

En Python 3 ya no es posible comparar entre objetos de distinto tipo; las comparaciones entre cadenas se hacen byte a byte, y las listas o tuplas se comparan elemento a elemento (lo cual puede no ser posible si los elementos no son comparables):

```
[1, 'a'] < [0, 'b']
```

```
False
```

```
[1, 'a'] < [1, 1]
```

```
TypeError: unorderable types: str() < int()
```

En particular, si todos los elementos de una lista son comparables,[9] podemos ordenarlas usando la función `sorted` o el método `sort`:

```
x = [1, 2, -3, 5, -1, 6, 1, 3, -4]
x.sort()
print(x)
```

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

```
x = list('AbraCadaBra')
sorted(x)
```

```
['A', 'B', 'C', 'a', 'a', 'a', 'a', 'b', 'd', 'r', 'r']
```

Nótese que la función crea una nueva lista mientras que el método la modifica. Tanto la función como el método admiten un parémetro opcional `key` que debe ser una función, y que es llamada para cada uno de los elementos de la lista, realizándose la ordenación sobre la transformación de dicha lista. Por ejemplo, podemos usar la función `abs` (valor absoluto):

```
x = [1, 2, -3, 5, -1, 6, 1, 3, -4]
x.sort(key=abs)
print(x)
```

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

También se puede ordenar en sentido inverso con el parámetro `reverse`.[10]

```
x = list('AbraCadaBra')
sorted(x,key=str.lower,reverse=True)
```

```
['r', 'r', 'd', 'C', 'b', 'B', 'A', 'a', 'a', 'a', 'a']
```

[9] Esto ocurre no sólo con las listas, sino con cualquier objeto *iterable*. Básicamente, un iterable es un objeto capaz de devolver sus miembros de uno en uno.

[10] Aquí usamos el método `lower` que actúa sobre cadenas, escribiéndolas en minúsculas. Pasamos el argumento `str.lower` al parámetro `key` pues es la forma de escribir un método (de la clase *string*) como una función. Dicho de otro modo, si `s` es una cadena, `s.lower()` y `str.lower(s)` hacen lo mismo.

Operador de pertenencia

Finalmente, en Python disponemos del operador de pertenencia `in`, (o no pertenencia, `not in`) que busca un objeto dentro de cualquier otro objeto iterable. Por ejemplo,

```
a = [0, 1, 2, 3, 4, 5]
print(7 in a)
print(3 in a)
print(6 not in a)
```

```
False
True
True
```

y que también sirve para buscar subcadenas de caracteres:

```
s = 'cadena'
print('ca' in s)
print('cd' in s)
```

```
True
False
```

Nótese que en los diccionarios, la búsqueda se realiza en las claves; para buscar en los valores debemos usar el método `values`:

```
d = {'a':1, 'b':2, 'c':3}
print('a' in d)
print(1 in d)
print(2 in d.values())
```

```
True
False
True
```

2.2 MÓDULOS

Una de las características principales de Python es su modularidad. La mayoría de funciones accesibles en Python están empaquetas 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 de cualquier objeto disponemos de la orden `dir`:

```
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']
```

Usada sin argumentos, la orden `dir()` devuelve un listado de las variables actualmente definidas. Con un objeto como argumento nos proporciona una lista con todos los atributos y métodos asociados a dicho objeto.

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

```
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` (para funciones matemáticas con complejos). 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. Es posible escribir la misma importación del siguiente modo

```
import math, cmath
```

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.

Durante una sesión interactiva es más frecuente cargar el módulo al completo, aunque 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. Podemos realizar una importación abreviada del módulo como sigue:

```
import math as m
```

de modo que ya no es necesario escribir `math.` para acceder a la funciones, sino

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

```
-1.0
```

Este tipo de importación suele ser más habitual cuando cargamos *submódulos*, esto es, módulos que existen dentro de otros módulos, de manera que la escritura completa se vuelve tediosa. Por ejemplo,

```
import matplotlib.pyplot as plt
```

2.2.1 La biblioteca estándar

Una de las frases más escuchadas cuando nos iniciamos en el mundo Python es que *Python trae las pilas incluidas*. Con esto se trata de reflejar el hecho de que Python trae consigo una extensa biblioteca de aplicaciones que nos facilita realizar un gran cantidad de tareas. Aunque en estas notas sólo veremos con detenimiento algunos de los módulos relacionados con la computación científica, conviene conocer algunos otros módulos de la biblioteca estándar como los que se muestran en la tabla 2.2.

Para obtener un listado de los módulos disponibles podemos usar la función `help`

```
help()
```

```
help> 

Welcome to Python 3.6's help utility!
...
```

en la que hemos abreviado la salida obtenida. La función nos ofrece la posibilidad de introducir cualquier función o directamente la palabra `modules`,

Tabla 2.2: Algunos módulos de la biblioteca estándar

Módulo	**Descripción**
`math`	Funciones matemáticas
`cmath`	Funciones matemáticas con complejos
`fractions`	Números racionales
`statistics`	Estadística
`os`	Funcionalidades del sistema operativo
`shutil`	Administración de archivos y directorios
`sys`	Funcionalidades del intérprete
`re`	Coincidencia en patrones de cadenas
`datetime`	Funcionalidades de fechas y tiempos
`pdb`	Depuración
`random`	Números aleatorios
`ftplib`	Conexiones FTP
`MySQLdb`	Manejo de bases de datos MySQL
`sqlite3`	Manejo de bases de datos SQLite
`xml`	Manejo de archivos XML
`smtplib`	Envío de e-mails
`zlib`	Compresión de archivos
`csv`	Manejo de archivos CSV
`json`	Manejo de ficheros JSON
`xmlrpc`	Llamadas a procedimientos remotos
`timeit`	Medición de rendimiento
`collections`	Más tipos de contenedores
`optparse`	Manejo de opciones en la línea de comandos

y nos proporcionará una lista de los módulos disponibles. Escribiendo a continuación el nombre del módulo deseado nos mostrará la ayuda relativa al mismo.

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 correspondientes a ciertas estructuras, como los bucles, 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 sucesión 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). 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.

Si tenemos bucles anidados, deberemos sangrar doblemente el bucle interior:

```
for i in range(3):
    for k in range(2):
        print(i+k,end=' ')
```

```
0 1 1 2 2 3
```

Obsérvese también en este ejemplo el uso del argumento opcional `end` en la llamada a la función `print` que establece el carácter final de impresión, que por defecto es un salto de línea, y aquí hemos cambiado a un simple espacio, lo que conlleva que las impresiones realizadas se hagan en la misma línea.

Como podemos ver, la orden `range(n)` representa una sucesión de números de n elementos, comenzando en 0.[11] Es posible que el rango comience en otro

[11] En Python 2, `range` creaba un objeto tipo lista, pero en Python 3, esta orden es un nuevo tipo de dato.

valor, o se incremente de distinta forma. El siguiente ejemplo muestra algunas listas de números generadas por diferentes llamadas a `range`:

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

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

La diferencia entre la lista generada por `range` y el objeto `range` es que la primera reside en la memoria de forma completa, mientras que la segunda no, lo que la hace computacionalmente más eficiente.

Nótese que los bucles en Python corren a través de la secuencia de elementos que sigue a `in`, (que como ya comentamos, puede ser cualquier objeto iterable) y no de los índices de la sucesión, como se muestra en los siguientes ejemplos:

```
a  = 'hola mundo'
for b in a.split():
    for s in b:
        print(s,end=' ')
    print()
```

```
h o l a
m u n d o
```

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

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

o simultáneamente, con el método `items`:

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

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

2.3.2 Condicionales

La escritura de sentencias condicionales es similar a la de los bucles `for`, usando los dos puntos y 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.

2.3.3 Bucles condicionados

Un bucle condicionado es un bloque de código que va a ser repetido mientras que cierta condición sea cierta. La estructura la determina la sentencia `while`.

Obsérvese el siguiente ejemplo:

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

```
0 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 línea; 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). El bloque va a seguir ejecutándose mientras que el valor de `b` sea menor que `20`.

Interrupción y continuación

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 finalizado completamente, en el caso de `for`, o cuando es falso, y no se ha interrumpido, en el caso de `while`.

Veamos los siguientes ejemplos:

```
1 for k in range(2,10):
2     for x in range(2,k):
3         if not k % x:  # si k es divisible por x
4             print(k,'es igual a',x,'*',k//x)
5             break
6     else:
7         print(k,'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. Nótese también que en la línea 3, en lugar de `k%x == 0` escribimos `not k%x` que es equivalente (recuérdese que cero es `False`) y ligeramente más eficiente.

En lo que se refiere al bucle `while–else`:

```
k = 1; mynumber = 7
while k < 5:
    if k == mynumber:
        break
    print(k,end=' ')
    k += 1
else:
    print("No")
```

```
1 2 3 4 No
```

vemos que la parte del `else` es ejecutada en este caso, pues el bucle ha finalizado sin interrupción (`mynumber = 7` hace que no se llegue a ejecutar la sentencia `break`); pero si ponemos

```
k = 1; mynumber = 3
while k < 5:
    if k == mynumber:
        break
    print(k,end=' ')
    k += 1
else:
    print("No")
```

```
1 2
```

entonces el bucle es interrumpido puesto que ahora sí alcanzamos el valor que hemos establecido para `mynumber`, y por tanto la parte de `else` no se ejecuta.

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 k in range(10):
    if not k % 2:
        print(k,"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(k):  # sucesión de Fibonacci hasta k
    a,b = 0,1
    print(a,end=' ')
    while b < k:
        print (b,end=' ')
        a, b = b, a+b
```

Ahora efectuamos la llamada a la función:

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

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

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

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

```
[0, 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)
```

```
0 1 1 2 3 5 8 13 21 34
```

```
print(a)
```

```
None
```

Cuando queremos devolver más de un valor en una función nos bastará empaquetarlos como 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 cell*,[12] `%%writefile` seguida del nombre del archivo:

```
%%writefile fibo.py
def fibolist(k):
    a,b = 0,1
    sucesion = [a]
    while b < k:
        sucesion.append(b)
        a, b = b, a+b
    return sucesion

x = 100
print(fibolist(x))
```

Si ahora ejecutamos el archivo desde la terminal:

```
$ python fibo.py
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Aunque también podemos ejecutar desde el entorno Jupyter:

```
run fibo.py
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

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

```
import fibo
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

y por tanto tendremos acceso a la función:

```
fibo.fibolist(50)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

o a las variables definidas en la función:

```
print(fibo.x)
```

```
100
```

[12] A diferencia de las *funciones mágicas*, que se denotan con un único signo de porcentaje, las *celdas mágicas* son instrucciones del entorno Jupyter que afectan a todo el contenido de la celda en la que se encuentran y llevan un doble signo de porcentaje.

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 `fibo.x`.

Si realizamos la importación con `*` (reiniciar previamente el núcleo):

```
x = 5
from fibo import *
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
print(fibolist(200))
print(x)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
100
```

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: (reiniciar primero el núcleo)

```
from fibo import fibolist
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
print(fibolist(30))
print(x)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
NameError: name 'x' is not defined
```

la función `fibolist` está disponible, pero nada más. Si quisiéramos la variable `x` habría que haberla importado explícitamente. 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:

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

if __name__ == "__main__":
    x = 100
    print(fibolist(x))
```

Lo que sucede es que cuando ejecutamos el código con `python fibo.py` desde la consola (o con `run fibo.py` desde el entorno Jupyter), la variable *especial*[13] `__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`*`)` del módulo `importlib`.

2.4.2 ¿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 de entorno* `PYTHONPATH`.[14] 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`.

Esta variable es una lista que podemos modificar en el transcurso de una sesión para que determinados directorios estén al alcance del intérprete y podamos acceder a la importación de los ficheros almacenados en ellos. En la sección 2.6 veremos cómo instalar módulos externos.

2.5 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

[13]En Python se conocen como variables especiales aquellas que comienzan y terminan con un doble guión bajo (doble *underscore*, conocido en el mundo Python como *dunder*).

[14]Una variable de entorno es un valor que controla un determinado comportamiento en un sistema operativo. En concreto, el `PYTHONPATH` es una lista de directorios que Python recorrerá para buscar un módulo.

añadir también que los diferentes tipos de números son inmutables y los diccionarios y conjuntos 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. Podríamos decir que el operador de asignación en Python no realiza una copia de valores, sino un etiquetado de los mismos.

Lo podemos ver con un sencillo ejemplo; la orden `id` nos muestra el identificador de un objeto, el cual podemos pensar como su dirección en memoria. Se trata de un identificador único para cada objeto que está presente. 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
```

```
139937026980680
```

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

```
[3, 2]
```

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

```
139937026980680
```

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

```
x = list(range(3))
y = x  # y referencia lo mismo que x
x.append(3) # modificamos x
print(y) # y también se modifica
```

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

una modificación de la lista `x` también modifica la lista `y`. Sin embargo,

```
x = list(range(3))
y = x
x = x + [3]  # nuevo objeto
print(y)
print(x)
```

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

¿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`.

En definitiva, si modificamos una lista mediante asignación, un método, o el operador de asignación aumentado, ésta se modifica en memoria, por lo que cualquier otra referencia a la lista quedará modificada. El lector puede comprobar que si en el ejemplo anterior sustituimos la línea `x = x + [3]` por `x += [3]`, el resultado es diferente.[15]

¿Cómo podemos entonces copiar una lista sin que el nuevo objeto quede enlazado al primero? Podemos para ello usar el *slicing*:

```python
x = list(range(10))
y = x[:] # copiamos x a y
```

De este modo, si ahora modificamos `x`, `y` no queda modificada

```python
x[:5] = []
print(x)
print(y)
```

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

Sin embargo, la copia mediante *slicing* es lo que se conoce como una copia *superficial* (*shallow copy*):

```python
x = [1, [1,2]]
y = x[:]
```

Si ahora modificamos un elemento de la lista `x[1]`,

```python
x[1][1] = 0
print(y)
```

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

vemos que también se ha modificado la copia. ¿Por qué ocurre esto? La copia superficial crea una copia de las referencias a cada uno de los elementos de la lista, por lo que si uno de dichos elementos es un objeto mutable, volvemos a tener el mismo problema: aunque lo modifiquemos, sus referencias apunta al mismo objeto. Para hacer una copia completamente independiente hemos de

[15] Es decir, el operador aumentado no funciona exactamente como una abreviatura, como ya comentamos al final de la subsección 2.1.2.

realizar lo que se denomina como una copia *profunda* (*deep copy*), para la que necesitamos el uso del módulo `copy`:

```
import copy
x = [1, [1,2]]
y = copy.deepcopy(x)
x[1][1] = 0
print(y)
```

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

Nótese también que este módulo posee una función `copy` que realiza una copia superficial de los objetos, es decir, es equivalente a la copia por *slicing* para las listas.[16]

Como hemos mencionado antes, los diccionarios y conjuntos son también objetos mutables, lo que significa que tienen el mismo comportamiento que las listas:

```
a = {'r': 'rojo','g':'verde'}
b = a
a['k'] = 'negro'
print(b)
```

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

Por tanto, si queremos realizar una copia de un diccionario (o un conjunto) que no esté enlazada con el original hemos de usar su propio método `copy` (que realiza una copia superficial):

```
c = a.copy()
a.pop('r')
print(a)
print(c)
```

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

Para copias profundas habría que usar la función `deepcopy` del módulo `copy`.

2.5.1 La orden `del`

Aunque la orden `del` sugiere el borrado de un objeto, el funcionamiento en Python está relacionado con la forma en la que se usan las referencias a objetos. Si usamos `del` sobre una variable, o mejor dicho, sobre una referencia

[16] Desde la versión 3.3 de Python las listas poseen también el método `copy` que realiza una copia superficial (funciona de forma idéntica a la copia por *slicing*).

a un objeto, lo que se hace es eliminar dicha referencia al objeto. Si el objeto no posee ninguna otra referencia, entonces Python lo pone accesible para el *recolector de basura* que finalmente se encargará de eliminarlo. Por ejemplo,

```
x = 8
print(x)
```

```
8
```

```
del x
print(x)
```

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

En objetos mutables, como las listas, `del` puede aplicarse a elementos o *slices* eliminándolos de dicha lista:

```
a = [1, 2, 3, 4, 5]
del a[:2]
print(a)
```

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

La orden `del` es útil para eliminar referencias que quedan activas después de realizar ciertas operaciones. Por ejemplo, si hacemos un bucle

```
for x in range(3):
    print(x,end=' ')
print(x)
```

```
0 1 2 2
```

vemos que después del bucle, la variable `x` sigue existiendo. Para evitar esto podríamos escribir `del x` y eliminar dicha referencia.

En ocasiones, si la variable del bucle no se usa dentro de éste,[17]

```
for x in range(5):
    print(random.randint(1,5),end=' ')
```

```
2 1 4 2 2
```

entonces podemos usar la variable *desechable* `_`:[18]

[17]La función `randint` del módulo `random` genera números enteros aleatorios en el rango dado.

[18]Recuérdese que en el entorno IPython o Jupyter Notebook, el significado de esta variable `_` es almacenar el último resultado obtenido.

```
for _ in range(5):
    print(random.randint(1,5),end=' ')
```

```
1 4 2 5 5
```

En particular, la variable desechable es útil para descartar algún elemento de una tupla (o un argumento de salida de una función) que no nos interese recuperar en un desempaquetado. Por ejemplo, el siguiente código sólo se queda con la extensión del nombre del archivo:[19]

```
import os.path
archivo = "nombre.pdf"
_ , ext = os.path.splitext(archivo)
print(ext)
```

```
.pdf
```

2.6 INSTALACIÓN DE MÓDULOS EXTERNOS

En la sección 2.4.2 hemos visto cómo importar un módulo creado por nosotros de manera directa, pero suele ser habitual que los programas más elaborados posean diversos módulos distribuidos en diferentes directorios, siguiendo una cierta jerarquía, configurando lo que se denomina un *paquete*. Si bien es posible modificar la variable `sys.path` para que el intérprete pueda encontrar el directorio oportuno, esto puede ser perjudicial en caso de que se repitan módulos con el mismo nombre.[20]

Para esta situación, Python dispone de un mecanismo sencillo consistente en añadir, en cada directorio que queramos considerar como un paquete, un archivo `__init__.py`, que en los casos más simples puede estar vacío, y en otros puede ejecutar código encargado de inicializar el paquete. Por ejemplo, supongamos que tenemos una carpeta de nombre `solvers` con el contenido estructurado en la figura 2.1.

Supongamos además que en el archivo `system.py` hay una función denominada `sistema`, y en los archivos `gauss.py`, `jacoby.py` y `seidel.py` existen sendas funciones `gauss_method`, `jacoby_method` y `seidel_method`, respectivamente. Obsérvese que en cada carpeta del módulo existe un archivo `__init__.py` que hará que cada subcarpeta sea considerada como un submódulo. Si consideramos inicialmente que todos los archivos `__init__.py` están vacíos, y suponiendo que

[19] La función `splitext` del submódulo `os.path` devuelve una tupla con el nombre y la extensión del archivo.

[20] Por lo que antes de dar un nombre a un módulo propio es importante saber si dicho módulo existe previamente. Una simple importación nos lo confirmaría.

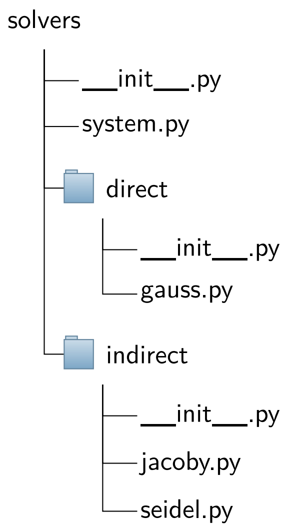


Figura 2.1: Contenido de la carpeta `solvers`

la carpeta `solvers` está accesible a Python, entonces para poder acceder a la función `sistema` tendríamos que realizar la importación del siguiente modo:

```
import solvers.system
```

y disponer de la función `solvers.system.sistema`, o bien usar la importación

```
from solvers.system import *
```

y tener acceso directo a la función `sistema`. Para poder usar las funciones definidas en `gauss_method` habría que escribir

```
from solvers.direct import gauss
```

y usar las funciones `gauss.gauss_method`.

Como podemos apreciar, no es quizás la forma más cómoda de trabajar con el paquete. Si configuramos adecuadamente los archivos `__init__.py` veremos que la importación es más razonable. Por ejemplo, si el contenido del archivo `__init__.py` de la carpeta principal es el siguiente

```
from solvers.system import sistema
```

entonces importando del siguiente modo:

```
import solvers
```

dispondremos de la función `solvers.sistema`.

Los archivos `__init__.py` también permiten configurar qué funciones o módulos se importan con `*`. Por ejemplo, si escribimos

```
from solvers.indirect import *
```

no se importa nada; esto es debido a que el fichero `__init__.py` de la carpeta `indirect` está vacío. Si escribimos en dicho fichero

```
__all__ = ['jacobi','seidel']
```

entonces la misma importación anterior nos da acceso a `jacobi.jacobi_method` y `seidel.seidel_method`. Si la variable `__all__` definida en ese archivo `__init__.py` sólo contuviera la lista `['seidel']`, entonces la importación hecha sólo daría acceso a la función `seidel.seidel_method`.

Para tener acceso a las funciones de los submódulos podríamos escribir, por ejemplo, en el archivo `__init__.py` de la carpeta `direct`

```
from solvers.direct.gauss import gauss_method
```

de manera que al escribir

```
import solvers.direct
```

tendremos acceso a la función de la forma `solvers.direct.gauss_method`. Una forma más cómoda sería importar de forma abreviada

```
import solvers.direct as direct
```

y poder usar la función `direct.gauss_method`.

Finalmente, si queremos configurar el paquete para poder acceder a las funciones de las subcarpetas de forma directa, podemos escribir en el archivo `__init__.py` de la carpeta `solvers`

```
from solvers.system import sistema
from .direct.gauss import gauss_method
from .indirect.jacobi import jacobi_method
from .indirect.seidel import seidel_method
```

y podemos dejar el resto de archivos `__init__.py` vacíos. En tal caso, realizando la importación

```
import solvers
```

tendremos bajo el objeto `solvers` todas las funciones del módulo. Este tipo de importación usando rutas relativas es especialmente útil si en alguno de los módulos de nuestro paquete queremos otro módulo del mismo paquete.

En definitiva, cuando tenemos diversos módulos en diferentes carpetas, este tipo de configuraciones nos permitirá acceder convenientemente a ellos.

2.6.1 Instalación de un paquete descargado

La amplia comunidad de usuarios de Python hace que tengamos a nuestra disposición una enorme cantidad de módulos para realizar todo tipo de tareas. En esta sección vamos a abordar la cuestión de cómo instalar un módulo creado por terceros que nos hayamos descargado desde internet.[21]

En principio, bastaría descargarse el archivo oportuno, descomprimirlo y dar acceso a Python a la carpeta creada (algo similar a lo comentado con la carpeta `solvers`). Sin embargo, la situación más habitual es la de encontrarnos con un módulo que ha sido empaquetado de una forma específica con `setuptools`, un módulo de Python para construir y distribuir paquetes. Reconoceremos fácilmente este tipo de paquetes, pues en la carpeta principal aparecerá un archivo `setup.py`. El procedimiento de instalación clásico en estos casos consiste en situarse en la carpeta en cuestión y escribir en la terminal

```
$ python setup.py install
```

Con esa orden se ejecutarán todas las tareas necesarias para instalar el módulo en algún lugar en el que Python podrá encontrarlo desde cualquier sitio,[22] sin necesidad de modificar el `PYTHONPATH` o `sys.path`.

Si no tenemos permiso para poder escribir en determinadas carpetas donde se instalan los módulos por defecto siempre es posible realizar una instalación local. La opción más simple consiste en usar la opción `--user` en la orden anterior:

```
$ python setup.py install --user
```

Esto instalará el módulo en una carpeta local predeterminada (usualmente bajo `$HOME/.local/`).

Si por el contrario queremos que el módulo esté en una carpeta específica determinada por nosotros, en primer lugar hemos de añadir la carpeta en cuestión al `PYTHONPATH` (en caso de que aun no lo esté), que se podría hacer de la forma:[23]

```
$ export PYTHONPATH=/ruta/donde/instalar
```

y luego ejecutar lo siguiente:

```
$  python setup.py install --install-lib /ruta/donde/instalar
```

[21] Ni que decir tiene que hemos de ser precavidos a la hora de ejecutar código cuya procedencia no ofrezca garantías de seguridad.

[22] Es probable que se necesiten permisos especiales para poder copiar ciertos archivos en los directorios de destino.

[23] Atención, esta orden sólo establece el `PYTHONPATH` mientras la terminal sigue abierta. Para hacerlo permanente hay que definir esta variable en el archivo oportuno.

2.6.2 Instalación desde un repositorio

El sitio oficial desde donde poder descargar todo tipo de paquetes es PyPI[24] que es un repositorio que en la actualidad consta de más de 120000 paquetes. Una vez encontrado el paquete deseado podemos descargarlo e instalarlo tal y como hemos comentado en la sección anterior, pero también es posible usar `pip`, una herramienta para gestionar la instalación de paquetes Python desde PyPI. Desde la página principal de PyPI podemos obtener `pip` (básicamente se trata de descargar el archivo `get-pip.py` y ejecutarlo con el intérprete). Si no se tienen los permisos adecuados será necesario realizar una instalación local mediante

```
$ python get-pip.py --user
```

Una vez instalada la herramienta, es sumamente sencillo instalar cualquier paquete del repositorio. Desde la terminal, bastará escribir

```
$ pip install paquete
```

Esta orden se encargará de bajar no sólo el paquete en cuestión, sino también todas sus dependencias,[25] así como realizar las tareas necesarias para su completa instalación.[26]

2.6.3 Instalación de paquetes con ANACONDA

La distribución de ANACONDA tiene su propio instalador de paquetes, denominado `conda` que tiene un funcionamiento similar a `pip`. De hecho, ANACONDA trae su propio `pip` que funciona de idéntico modo al comentado más arriba. ¿Cuál es la diferencia entre `conda` y `pip`, puesto que ambos hacen idéntica labor? La diferencia fundamental es que `pip` puede tener dificultades para manejar dependencias que no son puramente de Python, y en ocasiones no es posible instalar ciertos paquetes debido a que faltan herramientas para realizar todas las labores necesarias. `conda` por su parte sí es capaz de gestionar esos aspectos en los que `pip` falla, aunque sólo es posible instalar aquéllos paquetes que la distribución ANACONDA pone a disposición de los usuarios.

2.7 EJERCICIOS

E2.1 ¿Cuál de las siguientes órdenes produce un error y por qué?

(a) `a = complex(3,1)`

---

[24] *the Python Package Index* — `https://pypi.python.org/pypi`

[25] Esto es, los paquetes que se requieren para que el módulo se ejecute sin problemas.

[26] Eventualmente, algunas tareas requieren tener instalados ciertos compiladores.

(b) `a = complex(3)+ complex(0,1)`

(c) `a = 3+j`

(d) `a = 3+(2-1)j`

E2.2 ¿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]`

E2.3 Dada la cadena `s = 'Hola mundo'`, encuentra el método adecuado de producir las cadenas separadas `a = 'Hola'` y `b = 'mundo'`.

E2.4 De las siguientes definiciones, ¿cuáles de ellas corresponden a una tupla?

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

E2.5 Explica el siguiente comportamiento:

```
a = [1,2]
print(a.index(1))
print(a.index(2))
a[a.index(1)], a[a.index(2)] = 2,1
print(a)
```

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

E2.6 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.

E2.7 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]`

E2.8 Construir una función que calcule la media aritmética y la media geométrica de una lista de valores que entra como argumento. La media aritmética es

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

y la geométrica es

$$\tilde{x} = \sqrt[n]{x_1 \cdot x_2 \cdots x_n}$$

E2.9 Construye el vigésimo elemento de la siguiente sucesión definida por recurrencia:

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

E2.10 Escribe una función `mcd` que calcule el máximo común divisor de dos números enteros.

E2.11 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'`.

E2.12 Usando el ejercicio E2.11, escribir una función `is_palindromo` que indentifique palíndromos, (esto es, palabras que leídas en ambas direcciones son iguales, por ejemplo *radar*). Es decir, `is_palindromo('radar')` debería dar como resultado `True`.

E2.13 Modifica la función `fibolist` de la página 43 para que admita dos parámetros de entrada, `k` y `p` y que devuelva el primer elemento de la sucesión de fibonacci menor que `k` y múltiplo de `p`.

E2.14 Dado un número natural cualquiera, por ejemplo, 1235, se trata de crear una función que imprima el número tal y como sigue:

```
 *    ***    ***   *****
**   *   *  *   *  *
 *   *  *       *  *
 *     *      **    ***
 *    *         *      *
 *   *      *   *  *   *
***  *****   ***    ***
```

Para ello usar las siguientes listas (el símbolo `'␣'` denota un espacio en blanco):

```
cero = ["␣␣***␣␣",
        "␣*␣␣␣*␣",
        "*␣␣␣␣␣*",
        "*␣␣␣␣␣*",
        "*␣␣␣␣␣*",
        "␣*␣␣␣*␣",
```

```
        "␣␣***␣␣"]
uno = ["␣*␣", "**␣", "␣*␣", "␣*␣", "␣*␣", "␣*␣", "***"]
dos = ["␣***␣", "*␣␣␣*", "*␣␣*␣", "␣␣*␣␣", "␣*␣␣␣", "*␣␣␣␣
    ", "*****"]
tres = ["␣***␣", "*␣␣␣*", "␣␣␣␣*", "␣␣**␣", "␣␣␣␣*", "*␣␣␣
    *", "␣***␣"]
cuatro = ["␣␣␣*␣␣", "␣␣**␣␣", "␣*␣*␣␣", "*␣␣*␣␣", "******"
    , "␣␣␣*␣␣",
        "␣␣␣*␣␣"]
cinco = ["*****", "*␣␣␣␣", "*␣␣␣␣", "␣***␣", "␣␣␣␣*", "*␣␣
    ␣*", "␣***␣"]
seis = ["␣***␣", "*␣␣␣␣", "*␣␣␣␣", "****␣", "*␣␣␣*", "*␣␣␣
    *", "␣***␣"]
siete = ["*****", "␣␣␣␣*", "␣␣␣*␣", "␣␣*␣␣", "␣*␣␣␣", "*␣␣
    ␣␣", "*␣␣␣␣"]
ocho = ["␣***␣", "*␣␣␣*", "*␣␣␣*", "␣***␣", "*␣␣␣*", "*␣␣␣
    *", "␣***␣"]
nueve = ["␣****", "*␣␣␣*", "*␣␣␣*", "␣****", "␣␣␣␣*", "␣␣␣
    ␣*", "␣␣␣␣*"]

digitos = [cero,uno,dos,tres,cuatro,cinco,seis,siete,ocho,
    nueve]
```

E2.15 Modificando mínimamente la solución del ejercicio anterior, conseguir ahora que la impresión para 1235 sea:

```
 1    222    333   55555
11   2   2  3   3  5
 1   2  2       3  5
 1     2      33    555
 1    2         3      5
 1   2      3   3  5   5
111  22222   333    555
```

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 Jupyter Notebook 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(k,a=0,b=1):
    sucesion = [a]
    while b < k:
        sucesion.append(b)
        a,b = b,a+b
    return sucesion
```

La llamada a esta función puede realizarse de diversos modos.

- Usando sólo el argumento posicional: `fibonacci(100)`.
- Explicitando el argumento posicional: `fibonacci(k=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=1,k=100,a=0) #equivale a fibonacci(100)
```

pero

```
fibonacci(b=4,a=2)
```

```
TypeError: fibonacci() missing 1 required positional
    argument: 'k'
```

genera error pues falta el argumento posicional; ni tampoco se puede escribir:

```
fibonacci(k=100,1,1)
```

```
SyntaxError: positional argument follows keyword
    argument
```

pues los argumentos posicionales deben preceder a los nombrados.

Todas estas posibilidades hacen que en determinados momentos sea preferible *obligar* al usuario a realizar llamadas a una función especificando explícitamente determinados argumentos. Si cambiamos la cabecera de la función `fibonacci` por:

```
def fibonacci(k,*,a=0,b=1):
```

el `*` significa que todos los argumentos que aparecen después han de ser explicitados en la llamada. Es decir, podemos seguir usando

```
fibonacci(100)
```

pero ya no

```
fibonacci(100,2)
```

```
TypeError: fibonacci() takes 1 positional argument but 2
     were given
```

pues a partir del primer argumento hemos de explicitar los demás; es decir, llamadas válidas podrían ser:

```
fibonacci(100,a=3,b=2)
fibonacci(100,b=2) # por defecto a=0
fibonacci(k=100,a=1) # por defecto b=1
```

Número indeterminado de argumentos

Python admite la inclusión de un número indeterminado de parámetros en una llamada como una tupla de la forma `*argumento` ; por ejemplo:

```
def media(*valores):
      if len(valores) == 0: # len = longitud de la tupla
          return 0.0
      else:
          suma = 0.
          for x in valores:
               suma += x  # equivale a suma = suma + x
          return suma / 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(3,5,7,8))
```

```
3.5
5.75
```

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}
```

Por esta razón es muy habitual ver la cabecera de muchas funciones en Python escritas del siguiente modo:

```
def funcion(*args,**kwargs):
```

que nos permite pasar (o no) un número indeterminado de argumentos.

No sólo la definición de función permite el empaquetado de argumentos, sino que 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)
```

```
[5, 3, 8, 11, 19, 30, 49]
```

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

```
[2, 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 o para pasar funciones como argumentos que no queremos escribir formalmente. Por ejemplo, supongamos que queremos ordenar la siguiente lista:

```
a = [[1,2], [2,0], [1,-5], [0,1], [3,-1]]
sorted(a)
```

```
[[0, 1], [1, −5], [1, 2], [2, 0], [3, −1]]
```

¿Cómo podríamos ordenarla sólo en función de la segunda componente? Recordemos que la función `sorted` posee un parámetro `key` que es una función que transforma previamente la lista a ordenar, entonces:

```
sorted(a,key=lambda e:e[1])
```

```
[[1, −5], [3, −1], [2, 0], [0, 1], [1, 2]]
```

hace el efecto deseado.

3.1.4 Variables globales y locales

Las variables globales son aquéllas 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_cuadrado(lado):
    area = lado*lado
    return area
```

y ejecutamos

```
print(area_cuadrado(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_cuadrado`. ¿Qué ocurre si definimos esta variable previamente?

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

```
print(area_cuadrado(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 acceder al valor de `area` antes de su evaluación?

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

```
print(area_cuadrado(3))
```

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

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

Sin embargo, si no hay asignación, es posible usar variables externas a la función

```
area = 10
def area_cuadrado(lado):
    print(area)
    return lado*lado
print(area_cuadrado(3))
```

```
10
9
```

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_cuadrado(lado):
    global area
    print(area)
    area = lado*lado
    return area
```

```
print(area_cuadrado(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, sino que se trata de una variable global.

Variables locales y objetos mutables

Sin embargo, el empleo de ciertos métodos sobre variables que son mutables (véase la sección 2.5) 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 siendo ambas variables de tipo lista. 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.

Variables por defecto y objetos mutables

Por último señalar que las variables del espacio global se pueden usar en la definición de parámetros por defecto, pero puesto que las funciones en Python son objetos de primera clase, es decir, tienen el mismo comportamiento que cualquier otro objeto, se evalúan en el momento de su definición y no en cada llamada, ocurriendo lo siguiente:

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

```
(4, 8)
```

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

```
4
```

Como podemos ver en la última evaluación, aunque el valor de `a` ha cambiado, la función sigue teniendo presente el valor de `a` en el momento en el que se definió.

Nótese que esto ocurre sólo para los argumentos de la función, como puede verse en el siguiente ejemplo:

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

```
4
8
```

En particular, cuando un parámetro por defecto en una función es un objeto mutable pueden ocurrir comportamientos *extraños* . Obsérvese el siguiente ejemplo:

```
def insertar(valores=[]):
    valores.append(1)
    return valores
```

```
insertar([0])
```

```
[0, 1]
```

```
insertar()
```

```
[1]
```

```
insertar()
```

```
[1, 1]
```

```
insertar()
```

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

Como vemos, la función no se comporta como esperaríamos dado que la variable `valores` es una lista, de modo que es evaluada en el momento de la definición de la función y cualquier modificación que se haga mediante métodos modifica el objeto en memoria. En el ejercicio E3.13 proponemos al lector que trate de resolver este inconveniente.

3.1.5 Recursividad

Python permite definir funciones recurrentes, es decir, que se llamen a sí mismas, lo cual es un método inteligente para resolver cierto tipo de problemas. El ejemplo típico de recursividad aparece cuando definimos el *factorial* de un número natural k, que se define por:

$$k! = k \cdot (k-1) \cdot (k-2) \cdots 2 \cdot 1$$

esto es, el producto de todos los números naturales menores o iguales que el número dado. Si observamos que

$$k! = k \cdot (k-1)! \tag{3.1}$$

es sencillo escribir una función recursiva para realizar el cálculo:

```
def mifactorial(k):
    if k == 1:
        return k
    else:
        return k*mifactorial(k-1)
```

Como vemos, básicamente la función devuelve el equivalente a la ecuación (3.1), sólo que es necesario hacer un tratamiento especial cuando se trata del valor 1, pues si no, entraríamos en un bucle infinito. Si hubiéramos escrito:

```
def bad_factorial(k):
    return k*bad_factorial(k)
bad_factorial(5)
```

```
RecursionError: maximum recursion depth exceeded
```

Python genera un error al haber superado el máximo de niveles de recursión permitidos. En realidad, las funciones recursivas están muy limitadas por este hecho. Si usamos la función `mifactorial`

```
mifactorial(1000)
```

```
RecursionError: maximum recursion depth exceeded in
    comparison
```

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 `input`

```
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

```
import sys

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

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

haciendo la siguiente ejecución se obtiene

```
$ python fibo.py 30
[0, 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 la función mágica `run`:

```
run fibo.py 30
```

```
[0, 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 función `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: [1]

```
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] `ljust` o `rjust` llamado con un parámetro (que denota la longitud total de la cadena a imprimir) justifca a izquierda o derecha, respectivamente.

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

En este caso, el contenido del interior 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. A continuación mostramos algunos ejemplos:

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

```
abracadabra
```

También podemos usar nombres de variables en lugar de índices para pasar los argumentos de `format`:

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

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

o directamente usando un diccionario:

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

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

E incluso, es posible usar métodos de las variables que pasamos:

```
for x in [1+2j,3+1j,-2-5j]:
    print('Parte real: {num.real}; Parte compleja: {num.
        imag}'.format(num=x))
```

```
Parte real: 1.0; Parte compleja: 2.0
Parte real: 3.0; Parte compleja: 1.0
Parte real: -2.0; Parte compleja: -5.0
```

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*********
```

Nótese en los ejemplos anteriores que si `format` sólo lleva un parámetro no es necesario especificarlo (no ponemos nada en la especificación del formato antes de los dos puntos).

También disponemos de formatos específicos para manejar la impresión de números

```
print('{0:d} {0:0>3d} {0:2.3f} {0:3.2e}'.format(2))
```

```
2 002 2.000 2.00e+00
```

en el que podemos observar que `d` hace referencia a la impresión con enteros, en la que la anteposición de un número se refiere a la reserva de espacios para dicho número, y tal y como se ha visto en los ejemplos anteriores, la secuencia `0>` rellena con ceros a la izquierda.

Por su parte, los formatos `x.yf` y `x.ye` sirven para formatear números reales en formato decimal y científico, respectivamente, donde `x` es la reserva de espacio total e `y` el número de cifras de la parte decimal. Si `x` es menor que el número de espacios necesario para la impresión completa de la cifra, simplemente es ignorado.

Por último, señalar que si en la cadena de caracteres que vamos a formatear con el método `format` aparecen llaves que deseamos imprimir, éstas han de escribirse por duplicado:

```
print('{} va entre {{llaves}}'.format('Esto'))
```

```
Esto va entre {llaves}
```

3.2.2 Lectura de 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). El modo por defecto es `r`.

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

```
El Ingenioso Hidalgo
Don Quijote de la Mancha
```

entonces podemos abrirlo y leerlo con:

```
f = open('quijote.txt','r')
print(f.read())
```

```
El Ingenioso Hidalgo
Don Quijote de la Mancha
```

El método `read` del objeto `f` creado lee el archivo al que referencia dicho objeto. Si se le proporciona un argumento numérico entonces lee un número determinado de caracteres del archivo;[2] cuando es llamado sin argumento, lee el archivo al completo. Cada llamada a `read` mueve el puntero de lectura del archivo al último lugar leído, de manera que la llamada siguiente lee a partir de ahí. Por eso:

```
f.read()
```

```
''
```

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
print(f.read(20))
```

```
El Ingenioso Hidalgo
```

hemos leído (e impreso) los primeros 20 caracteres. Y si ahora escribimos

```
f.read(4)
```

[2]Hay que tener en cuenta que también hay que considerar los caracteres de tabulación, fin de línea, etc.

```
'\nDon'
```

leemos los siguientes 4 caracteres, uno de los cuales es el caracter de fin de línea. Nótese la diferencia entre el método `f.read()` y la impresión resultante de `print(f.read())`: en el primero la cadena que contiene el archivo es leída en formato de representación interna, que difiere de la representación que se muestra con `print`, en el que la cadena es impresa.

También es posible leer línea a línea con `readline`

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

```
'El Ingenioso Hidalgo\n'
```

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

```
Don Quijote de la Mancha
```

```
f.readline()
```

```
''
```

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,end='')
```

```
El Ingenioso Hidalgo
Don Quijote de la Mancha
```

Otra opción aún más cómoda es iterar sobre el propio objeto:[3]

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

que produce exactamente el mismo resultado que antes.

Para cerrar un archivo usaremos el método `close`:

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

[3]Lo que significa que `f` es un *iterador* (véase la sección 3.3.1).

También existe una forma de manejar los archivos que aisla los errores que puedan ocurrir y que cierra automáticamente el archivo una vez que estemos fuera del entorno. Se trata de la orden `with` que funciona del siguiente modo:

```
with open('archivo.py') as f:
    read_data = f.read()
```

Fuera del bloque correspondiente a `with` no existe el objeto `f`.

3.2.3 Escritura

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):

```
with (open('hola.py','w') as f:
    for x in ['primera','segunda','tercera']:
        f.write(x+'\n')
```

Una vez fuera del entorno debemos volver a abrir para leerlo:

```
with open('hola.py') as f:
    print(f.read())
```

```
primera
segunda
tercera
```

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 necesitaremos cambiar la salida.

De forma similar funciona el parámetro `file` de la función `print`, el cual redirecciona la salida hacia el objeto que determinemos. Por ejemplo,

```
with open('archivo.txt','w') as f:
    print("Hola mundo",file=f)
```

Mucho más *agresivo* sería redireccionar toda la salida por pantalla a un archivo, reescribiendo la variable `stdout`. Si hacemos,

```
f = open('archivo.txt','w')
sys.stdout = f
```

cualquier impresión dirigida a la pantalla se escribirá en el fichero `archivo.txt`. Esto no es muy recomendable, pues cualquier impresión generada por cualquier módulo que carguemos (y que quizás no controlamos) va a ir a dicho archivo, lo que puede no ser una buena idea.

3.3 LISTAS POR COMPRENSIÓN, ITERADORES Y GENERADORES

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)
[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]
```

Nótese que es un poco mas *pythonic*[4] escribir

```
[x**3 for x in range(10) if not x%2]
```

debido a que un entero distinto de cero tiene valor `True`.

De forma similar creamos ahora dos listas:

```
a = [x for x in range(1,7) if not x%2]
b = [x for x in range(1,7) if x%2]
print(a)
print(b)
```

[4] 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.

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

que podemos recorrer en forma anidada:

```
[x*y for x in a for y in b]
```

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

Pero si queremos recorrerlas simultáneamente:

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

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

El último ejemplo es una muestra de código *unpythonic*, pues se puede construir de forma más *pythonic* 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]
```

La sintaxis se puede exportar para crear diccionarios o conjuntos por comprensión:

```
s = "contamos las vocales de esta frase y las guardamos en
    un diccionario"
d = {x: s.count(x) for x in s if x in 'aeiou'}
print(d)
```

```
{'i': 3, 'e': 5, 'o': 6, 'a': 9, 'u': 2}
```

o también se puede usar la función `dict`

```
dict((x, s.count(x)) for x in s if x in 'aeiou')
```

De forma similar, se pueden definir conjuntos por comprensión, por ejemplo, de múltiplos de 13 menores que 100:

```
s = {x for x in range (1 , 101) if not x % 13}
t = set(x for x in range (1 , 101) if not x % 13)
print(s==t)
print(s)
```

```
True
{65, 39, 13, 78, 52, 26, 91}
```

Finalmente, nótese que en la versión 3 de Python las variables declaradas dentro de una lista por comprensión ya no se filtran fuera de ésta, es decir,

```
[i for i in range(5)]
print(i)
```

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

3 3 1 Iteradores

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 = list(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 que establece una secuencia sobre la que 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)))]
```

La diferencia entre un *iterable* (véase la nota al pie de la pág. 31) y un *iterador* es algo sutil. Un iterador es un objeto que se puede obtener de un iterable a través de la función `iter`. En realidad, podemos implementar un bucle tanto con un iterable como con un iterador:

```
a = [1,2,3]
for x in a:
    print(x,end=' ')
```

```
1 2 3
```

```
b = iter(a)
for x in b:
    print(x,end=' ')
```

```
1 2 3
```

pues al fin y al cabo, cuando hacemos un bucle sobre un iterable, en realidad lo que se hace es definir el iterador asociado a dicho iterable y recorrerlo. En un iterador tenemos definido el método `__next__` al que podemos acceder también con la función `next`, que nos proporciona el siguiente elemento de la secuencia. En los iterables no tenemos esta posibilidad:

```
a = 'hola'
next(a)
```

```
TypeError: 'str' object is not an iterator
```

```
b = iter(a)
next(b)
```

```
'h'
```

En esencia, un iterador es una especie de fábrica de valores que permanece inactiva hasta que es invocada con la función `next`, con la que devuelve el siguiente valor, y pasa de nuevo a la inactividad.

3 3 2 `map` y `filter`

Una alternativa a la creación rápida de listas es el uso de la función `map` que permite aplicar una función a una lista. Por ejemplo, podemos incluir los dígitos de un número en una lista del siguiente modo. En primer lugar convertimos el número a cadena de caracteres, y a continuación lo convertimos en lista:

```
a = 1235151
b = list(str(a))
print(b)
```

```
['1', '2', '3', '5', '1', '5', '1']
```

de este modo tenemos una lista con los dígitos, pero como cadena de caracteres. Si ahora queremos convertir las cadenas a números, podemos usar una lista por comprensión usando la función `int`:

```
[int(x) for x in b]
```

```
[1, 2, 3, 5, 1, 5, 1]
```

o bien usar la función `map` cuyos argumentos serán la función que vamos a aplicar a cada uno de los elementos de la lista (o iterable), y el iterable:

```
map(int,b)
```

```
<map at 0x7fbf0599d7b8>
```

Podemos ver que el resultado no parece ser el esperado, pero en realidad lo que hemos construido es un iterador. Podemos obtener fácilmente la lista si hacemos:

```
list(map(int,b))
```

```
[1, 2, 3, 5, 1, 5, 1]
```

Pero si lo que queremos es iterar sobre la lista, será mucho más eficiente iterar sobre el objeto `map` pues, a diferencia de la lista, que es almacenada completamente en memoria, este objeto sólo almacena en memoria el siguiente objeto a iterar. Por ejemplo, podemos recuperar el número inicial con el siguiente bucle:

```
s = 0
for i,x in enumerate(map(int,reversed(b))):
    s += 10**i*x
print(s)
```

```
1235151
```

La función `map` es eficiente combinada con `lambda`, y puede trabajar sobre más de una secuencia:

```
a = [1, 3, 5]
b = [2, 4, 6]
list(map(lambda x,y: x*y,a,b))
```

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

Por su parte, `filter` permite filtrar todos los elementos de una secuencia que satisfacen cierta condición. Funciona de forma parecida a `map`, con la excepción de que la función suministrada ha de devolver un booleano:

```
s = "vamos a contar las vocales de esta cadena"
len(list(filter(lambda x: x in 'aeiou',s)))
```

```
15
```

En el ejemplo, definimos una cadena `s` a la que vamos a contar sus vocales. En lugar de preguntar si cada elemento de la cadena es igual a `'a'`, `'e'`, `'i'`, `'o'`, `'u'`, simplemente comprobamos si está en la cadena `'aeiou'`. La función `lambda` definida será `True` si la letra es una vocal y `False` en caso contrario. Lo que hace `filter` es filtrar sólo los resultados positivos (es decir, las vocales), y finalmente contamos los elementos resultantes con `len`.

3.3.3 Generadores

Una de las ventajas de `map` frente a las listas por comprensión es que no almacena la lista resultante en memoria, sino sólo la generación de ésta, de forma que su rendimiento es muy alto. No obstante, se puede conseguir lo mismo con las listas por comprensión si en lugar de corchetes usamos paréntesis. En tal caso, lo que obtenemos es un *generador*, que es un tipo especial de iterador, que se crea de forma más sencilla, bien mediante el mecanismo de las listas por comprensión con paréntesis (las denominadas *expresiones generadoras*), o bien mediante una *función generadora* que básicamente es una función que en lugar de `return` usa `yield`.

El comportamiento de una función generadora es curioso: la primera vez que llamamos a la función, cuando encontramos `yield` se retorna el valor obtenido como es habitual, pero los valores locales de la función y el puntero de ejecución son congelados en ese momento, y en la siguiente llamada a la función, se prosigue desde el punto en el que se quedó. Veamos un sencillo ejemplo:

```
def fibonacci():
    a, b  = 0, 1
    yield a
    while True:
        yield b
        a, b = b, a+b
```

Una vez más estamos creando la sucesión de Fibonacci, pero en este caso sin límite, y los estamos retornando con `yield`. Si ahora hacemos

```
g = fibonacci()
```

estamos creando un generador (un iterador, de hecho) que va a ir devolviendo uno a uno los valores de la sucesión. Por ejemplo, los 10 primeros términos los obtenemos con:[5]

```
for _ in range(10):
    print(next(g),end=' ')
```

```
0 1 1 2 3 5 8 13 21 34
```

Los generadores consumen los valores utilizados y no podemos volver a ellos, por lo que si nuevamente hacemos

```
for _ in range(10):
    print(next(g),end=' ')
```

[5] Nótese el uso de la variable desechable `_` vista al final de la sección 2.5.1.

```
55 89 144 233 377 610 987 1597 2584 4181
```

obtenemos los siguientes 10 términos.

Si aplicamos la función `list` sobre un generador nos dará una lista con todos los elementos.[6]

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 y el error pueda ser debidamente tratado. Por ejemplo, supongamos que tenemos la siguiente situación, que obviamente, producirá una excepción:

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

```
1.0
ZeroDivisionError: division by zero
```

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

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

```
1.0
Se ha producido un error
```

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 se encarga de manejar convenientemente el error que se haya producido.

El bloque `except` no tiene por qué ser único, y además admite mayor sofisticación pues permite tratar cada excepción en función de su tipo. Podemos verlo en el siguiente ejemplo:

[6] ¡Atención, en el ejemplo anterior bloquearemos la memoria del ordenador pues la sucesión generada tiene infinitos términos!

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

```
División por cero
División inválida
```

También es posible agrupar varios tipos de excepciones mediante una tupla:

```
for a in [1,'1',0]:
    try:
        b = 1/a
        print(b)
    except(ZeroDivisionError,TypeError):
        print("No se puede realizar la división")
```

```
1.0
No se puede realizar la división
No se puede realizar la división
```

Sólo el tipo de excepción que coinciden con alguna de las que aparecen junto a `except`, o que es compatible con alguna de éstas,[7] son tratadas en ese bloque. El resto genera un mensaje de error usual.

En general,suele ser recomendable tratar cada tipo de excepción de forma individual, pero cuando usamos varios bloques `except` debemos prestar atención al orden en el que se pone cada excepción. Obsérvese el siguiente ejemplo:

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

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

El acceso a elementos que no existen en una lista genera un `IndexError`. Ahora, supongamos que intentamos capturar la excepción del siguiente modo:

[7] Las excepciones se agrupan por clases siguiendo una jerarquía de dependencias entre una clase y sus clases derivadas. Decimos que las excepciones son compatibles si pertenecen a la misma clase o a cualquiera de sus clases derivadas.

```
a = [0,1,2]
try:
    a[3] = 0
except LookupError:
    print("Error en la búsqueda")
except IndexError:
    print("Error de indexación")
```

```
Error en la búsqueda
```

Vemos que el fragmento que ha capturado la excepción es el relativo a `LookupError`, ya que el error de tipo `IndexError` es una clase derivada de `LookupError`. Sería más correcto cambiar los bloques `except` de orden, de manera que la excepciones de clases derivadas aparezcan antes que las excepciones de clases más genéricas, teniendo de este modo una captura más precisa de la excepción.

`else` y `finally`

La estructura `try–except` admite un bloque opcional con `else` que debe aparecer después de todas las sentencias `except`. El bloque `else` se ejecutará si el bloque `try` no produce ninguna excepción. En principio, el código del bloque `else` se podría poner junto con el del bloque `try`, pero podría ocurrir que generase un error capturado por la excepción que estuviéramos omitiendo. Obsérvese el siguiente ejemplo:

```
for a in [1,'1']:
    try:
        b = 1/a
        print(b)
        f = open('archivo.txt','r')
    except:
        print("No se puede realizar la división")
```

```
1.0
No se puede realizar la división
No se puede realizar la división
```

Como vemos, la excepción está pensanda para capturar el problema de la división, sin embargo se produce una excepción adicional que enmascara el caso en el que sí se puede realizar la división. Por ello es más adecuado, o bien explicitar la excepción que queremos capturar,[8] o bien acotar en todo lo posible el código que queremos analizar. Una mejor opción sería:

[8]De hecho, no se recomienda en absoluto capturar excepciones de forma genérica.

```
for a in [1,'1']:
    try:
        b = 1/a
        print(b)
    except:
        print("No se puede realizar la división")
    else:
        f = open('archivo.txt','r')
```

```
1.0
FileNotFoundError: [Errno 2] No such file or directory:
    'archivo.txt'
```

Por otra parte, podemos añadir a un bloque `try`–`except` una cláusula final para ser ejecutada en cualquier circunstancia. Para ello disponemos de la sentencia `finally` que marcará un bloque de código que se ejecutará antes de abandonar el bloque `try`, con independencia de que se produzca o no una excepción.

El bloque `finally` se ejecutará siempre, incluso si la excepción no ha sido capturada o se ha producido en el bloque `except` o `else`, o aunque abandonemos el bloque `try` mediante alguna otra interrupción (vía `break`, `continue` o `return`).

En el ejemplo siguiente:

```
a = 1
try:
    b = 1/a
    print(b)
except:
    print("No se puede realizar la división")
else:
    f = open('archivo.txt','r')
finally:
    print('ejecución final')
```

```
1.0
ejecución final
FileNotFoundError: [Errno 2] No such file or directory:
    'archivo.txt'
```

vemos que, aunque se produce un error antes de llegar al bloque `finally`, éste se ejecuta y luego se muestra la excepción.

Opcionalmente, es posible obtener información específica de cualquier excepción mediante la siguiente estructura:

```
for a in ['1',1,0]:
    try:
        b = 1/a
        print(b)
    except Exception as e:
        print(type(e))
        print("No se puede realizar la división")
        print(e)
    print()
```

```
<class 'TypeError'>
No se puede realizar la división
unsupported operand type(s) for /: 'int' and 'str'

1.0

<class 'ZeroDivisionError'>
No se puede realizar la división
division by zero
```

donde `e` almacena la información de la excepción ocurrida. En el ejemplo anterior `Exception` puede ser sustituido por cualquier tipo de excepciones deseada.

3.4.1 Lanzar excepciones

La orden `raise` nos permite lanzar una determinada excepción, o la última excepción capturada (si estamos dentro de un bloque `try–except`).

Por ejemplo, en cualquier parte de un código podemos escribir:

```
raise NameError('Mensaje de error')
```

```
NameError: Mensaje de error
```

y el programa se detendría con dicho mensaje. Fuera de un bloque `try–except` y sin argumento,

```
raise
```

```
RuntimeError: No active exception to reraise
```

da lugar a un error de ejecución que nos informa de que no hay capturada ninguna excepción. Mientras que dentro de un bloque `try–except`, relanza dicha excepción:

```python
a = 0
try:
    b = 1/a
except Exception:
    print("Ha ocurrido un error")
    raise
```

```
Ha ocurrido un error
ZeroDivisionError: division by zero
```

3.5 OTRAS SENTENCIAS ÚTILES

3.5.1 `any` y `all`

Python dispone de las funciones `any` y `all` que aplicadas sobre un iterable devuelven `True` si alguno (para `any`) o todos (para `all`) sus elementos son ciertos. Si el iterable es vacío, `any` devuelve `False` y `all` devuelve `True`.

En el ejemplo siguiente, creamos una lista de 5 enteros aleatorios entre 1 y 4 y vemos si hay alguno par,

```python
import random
a = [random.randint(1,4) for x in range(5)]
print(a)
any([not x%2 for x in a])
```

```
[3, 1, 1, 1, 1]
False
```

o si todos son impares

```python
all(x%2 for x in a)
```

```
True
```

Nótese que en este último ejemplo hemos usado un generador en lugar de una lista.

3.5.2 Expresiones condicionales

En algunos lenguajes de programación existe el *operador ternario* cuya sintaxis (proveniente del lenguaje C) es `e1 ? e2 : e3`. Esta sentencia evalúa una expresión booleana `e1` y toma el valor `e2` si ésta es cierta y `e3` si es falsa. En Python disponemos de la denominada *expresión condicional*, que funciona de forma equivalente como `e2 if e1 else e3`.

Por ejemplo, la siguiente función eleva al cuadrado si el número es par, y divide por dos si es impar:

```
def fun(x):
    return x**2 if not x % 2 else x / 2
print(fun(10))
print(fun(11))
```

```
100
5.5
```

3.5.3 Concatenación efectiva de cadenas

Entre los diversos métodos para manejar cadenas de caracteres está el método `join` que permite unir una colección de cadenas. Es especialmente útil cuando queremos concatenar diversas cadenas usando el mismo *string* como elemento intermedio. Por ejemplo:

```
a = "uno", "dos", "tres"
'-'.join(a)
```

```
'uno—dos—tres'
```

Como podemos ver, estamos recorriendo el iterable `a` y uniéndole la cadena sobre la que usamos el método `join`. Lo podemos usar para invertir rápidamente una cadena (véase ejercicio E2.11)

```
a = "hola"
''.join(reversed(a))
```

```
'aloh'
```

o de forma más sofisticada para hacer el ejercicio E2.14:

```
a = 1235
for y in range(7):
    print('  '.join([digitos[x][y] for x in map(int,str(a)
        )]))
```

(la lista `digitos` aparece definida en el citado ejercicio). Obsérvese el funcionamiento de este breve código: en primer lugar `map(int,str(a))` define un iterador sobre cada uno de los dígitos que componen `a`, de modo que

```
[digitos[x] for x in map(int,str(a))]
```

es una lista por compresión con las *números* impresos con `*`. Finalmente hay que recorrer uno a uno cada elemento que compone esta lista (que es lo que lleva a cabo el bucle `for`).

3 6 EJERCICIOS

E3.1 Calcula $\displaystyle\sum_{i=1}^{100} \frac{1}{\sqrt{i}}$

E3.2 Calcula la suma de los números pares entre 1 y 100.

E3.3 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: $3 + 5 + 6 + 9 = 23$

E3.4 Utiliza la función `randint` del módulo `random` para crear una lista de 100 números aleatorios entre 1 y 10.

E3.5 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.

E3.6 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.

E3.7 Crea una lista aleatoria de números enteros (usar ejercicio E3.4) y luego escribe un programa para que elimine los valores impares de dicha lista.

E3.8 Escribir una función `histograma` que admita una lista aleatoria de números enteros entre 1 y 5 (usar ejercicio E3.4) 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 ***
```

E3.9 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:

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

```
(1,6)
```

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

```
(1,10)
```

E3.10 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 ε que por defecto ha de ser 10^{-4}.

E3.11 Considera la siguiente función:

```python
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:

```python
media(*range(5,10))
```

```
1.0
```

y corrígelo para que que proporcione el resultado correcto (7.0).

E3.12 Considera la función cuyo encabezado es el siguiente:

```python
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.

E3.13 Modifica la función `insertar` de la página 67 para que tenga el comportamiento correcto.

E3.14 Repite el ejercicio E2.11 usando una función recursiva.

E3.15 Crea un fichero de nombre `hist.py` que incluya la función `histograma` del ejercicio E3.8. 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.

E3.16 Repite el ejercicio E2.15 modificando el código de la página 88 que hace referencia al ejercicio E2.14.

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, con el que podemos resolver una gran variedad de problemas científicos, y que veremos en el capítulo siguiente, están basados en el uso extensivo de *arrays* de NumPy.

4.1 ARRAYS

Para empezar a usar este módulo lo importaremos del siguiente modo

```
import numpy as np
```

que se ha convertido prácticamente en un estándar universal.

Como comentamos más arriba, el módulo introduce un nuevo tipo de objeto, el *array*, similar a una lista, pero con una diferencia fundamental: así como en las listas los elementos pueden ser de cualquier tipo, un *array* de

NumPy debe tener todos sus elementos del mismo tipo.[1] Para definirlo será necesario usar la orden `array`:

```
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

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

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

Nótese que en NumPy, los tipos de datos numéricos son un poco más extensos, incluyendo reales en doble precisión (`float64`) y 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
print(a)
```

```
[ 2.  3.  5.]
```

[1] Parte de la ventaja computacional que ofrece NumPy estriba en este aspecto, pues las operaciones entre *arrays* no tendrán que chequear el tipo de cada uno de sus elementos.

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

```
[ 3.6  3.   5. ]
```

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
```

4.1.1 Atributos básicos

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
```

```python
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*:

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

```
True
```

```python
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*:

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

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

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

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

```python
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

```python
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:[2]

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

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

y con el método `flatten` obtenemos *arrays* unidimensionales:

```
print(a)
print(a.flatten())
```

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

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

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

4.2.2 *Arrays* con estructura

Las siguientes funciones son útiles para crear *arrays* de un tamaño dado, y reservar memoria en ese momento, pues es importante señalar que no podemos modificar el tamaño de un *array*, salvo que lo creemos de nuevo:

```
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.]]
```

[2] Si se usa con *arrays* multidimensionales invierte sus dimensiones.

O bien

```
d = np.zeros_like(b) # d es como b
e = np.ones_like(c) # e es como c
f = np.empty_like(a) # f es como a, sin datos
```

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.]]
```

que creemos que no precisan explicación.

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]]
```

```python
print(np.diag(np.arange(1,3),-1)) # movemos la diagonal
```

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

```python
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, aquí creamos una matriz 2×3 con elementos de una distribución normal de media 0 y varianza 1:

```python
print(np.random.normal(0,1,(2,3)))
```

```
[[ 2.30182001  0.48023222 -0.42059946]
 [ 0.07596116 -0.20566722 -0.71062033]]
```

o una matriz 2×2 de enteros aleatorios entre 1 y 9 (el segundo no se incluye)

```python
print(np.random.randint(1,10,(2,2)))
```

```
[[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 = list(range(5))
a[1:4] = [6]
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]
```

puesto que el *array* no puede cambiar de tamaño.

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]) # tercera columna
```

```
[2 5 8]
```

Nótese que no hay diferencia en la salida de los dos últimos resultados: ambos son *arrays* unidimensionles. En la sección 4.4.1 trataremos este hecho con más detalle.

Hay un aspecto importante que diferencia el *slicing* en una lista y en un *array* de NumPy. En una lista, el *slicing* crea un nuevo objeto mientras que en un *array* no; lo que se obtiene con el *slicing* de un *array* es lo que se denomina una *vista* del objeto. Recordemos qué ocurría con las listas:

```
a = list(range(5))
b = a[2:5]
a[-1] = -1 # modificamos a
print(a)
print(b) # b no se modifica
```

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

Sin embargo,

```
a = np.arange(5)
b = a[2:5]
print(b) # antes de la modificación de a
a[-1] = -1
print(a)
print(b) # después de la modificación de a
```

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

¿Qué ventajas puede tener este *extraño* comportamiento? Si por ejemplo estamos trabajando con un *array* y con su transpuesto, y en algún momento hemos de modificar el *array* original, tiene sentido que también queramos realizar la modificación en su transpuesto. Gracias a este comportamiento, no es necesario realizar explícitamente la modificación, pues ambos objetos, el *array* y su transpuesto están enlazados de forma permanente (es decir, el transpuesto es una *vista*, esto es, otra forma de ver el objeto original).

¿Cómo podemos entonces modificar un *array*, pero no su transpuesto? Disponemos para ello del método `copy` que nos permite realizar copias del objeto, que ya no tienen el comportamiento de las *vistas*:

```
a = np.arange(9).reshape(3,3)
b = a[:2:,:2].copy() # copia en nuevo objeto
a[0,0] = -1 # modificamos a
print(a)
print(b) # b no se modifica
```

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

Es importante señalar que los métodos `reshape` y `transpose` crean vistas del objeto, mientras que `flatten` crea una copia.

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.]]
```

Pero desde la versión 3.5 de Python se ha introducido el operador `@` como sustituto de `dot`, que es más engorroso de escribir:

```
print(a @ b)
```

```
[[ 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* como argumento.

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]
```

Si embargo, el rendimiento de una función vectorizada frente a una función definida desde NumPy es muy inferior. El entorno Jupyter Notebook dispone de la *magic cell* `%%timeit` que nos permite dar una estimación del tiempo de cómputo de la ejecución de una celda. Si creamos un *array* de gran tamaño

```
a = np.arange(100000)
```

podemos evaluar lo que tarda en calcular la función vectorizada `vfunc` del siguiente modo:

```
%%timeit
b = vfunc(a)
```

```
45.4 ms ± 1.01 ms per loop (mean ± std. dev. of 7 runs,
    10 loops each)
```

Como vemos, tarda alrededor de 45 milisegundos en realizar el cálculo. Básicamente la función `%%timeit` lleva a cabo una repetición del cómputo de una celda un determinado número de veces y obtiene una media de los tiempos obtenidos.[3] Haciendo lo mismo con la función seno de NumPy:

```
%%timeit
b = np.sin(a)
```

[3]Por ello, es fundamental no incluir órdenes de impresión en una celda con `%%timeit`.

```
5.12 ms ± 49.8 µs per loop (mean ± std. dev. of 7 runs,
    100 loops each)
```

obtenemos un tiempo casi 9 veces más rápido.

4.4.1 *Arrays* 1D vs. matrices

Como hemos podido observar al hacer *slicing* sobre un *array* multidimensional, hay una sutil diferencia entre un *array* unidimensional y una matriz fila o columna. Obsérvese el siguiente ejemplo:

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

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

```python
print(a.T)
print(a.T.shape)
```

```
[0 1 2]
(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:

```python
a.T @ a
```

```
5
```

```python
a @ a.T
```

```
5
```

de hecho, se obtiene lo mismo sin necesidad de usar la transposición:[4]

```python
a @ a
```

```
5
```

[4] Existe además la función `outer` para el producto tensorial de *arrays* unidimensionales (vectores) así como `inner` y `cross` para el producto interior y el producto vectorial.

Ello es debido a que tanto `a` como `a.T` son *arrays* unidimensionales, que son objetos distintos a las matrices.[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, es decir, en una matriz fila (o columna, según se quiera). Se puede hacer con el método `reshape`:

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

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

```
print(b @ b.T)
```

```
[[5]]
```

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

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

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

Obsérvese que

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

```
(1, 3)
```

4.4.2 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:

[5] El módulo `numpy` también dispone de un tipo de objeto `matrix`, pero no lo vamos a usar en este texto.

```
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(a @ x - b)
```

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

4.4.3 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 debemos considerarlos como matrices añadiéndoles una dimensión:

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

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

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]]
```

el pegado por defecto se hace por filas:

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

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

Para hacerlo por columnas es preciso añadir la opción `axis=1` (por defecto es `axis=0`)

```
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 una 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

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

con las que pretendemos realizar la siguiente operación: si consideramos la primera columna de A y la primera fila de B, podríamos construir la matriz 2×2 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. En concreto,

```
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:

```
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)
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 el *broadcasting* en 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 matrices 2×1 con matrices 1×2. De este modo, debemos considerar a `A` como 3 matrices columna (lo que corresponde a un 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 el ejercicio E4.8).

4.6 OTRAS OPERACIONES 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.argmin() # 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 (nótese que no es necesario escribir `a!=0`, sino simplemente `a`):

```
np.where(a)
```

```
(array([0, 1, 3]),)
```

Aun no hemos mencionado un par de constantes especiales disponibles con el módulo NumPy, `nan` e `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 que representan índices:

```
mask = np.random.randint(0,5,5)
print(mask) # máscara de índices enteros
```

```
[2 4 3 2 0]
```

```
a = np.array([11,9,12,15,11,7])
print(a)
```

```
[11  9 12 15 11  7]
```

```
print(a[mask]) # selección según la máscara (nuevo objeto)
```

```
[12 11 15 12 11]
```

```
a[[3,1]] = 0 # asignación a las posiciones 3 y 1
print(a)
```

```
[11  0 12  0 11  7]
```

Las máscaras pueden ser incluso más sofisticadas, posibilitando el cambio de forma del *array*:

```
b = np.arange(10)[::-1] # orden inverso!
print(b)
```

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

```
mask = np.random.randint(0,10,4).reshape(2,2)
print(mask) # máscara 2x2 aletoria
```

```
[[1 1]
 [0 6]]
```

```
print(b[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, columas 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 los 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 UN EJEMPLO DEL USO DEL *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 del siguiente modo:

$$(u_1 - u_0, u_2 - u_1, \dots u_n - u_{n-1}) = (u_1, u_2, \dots, u_n) - (u_0, u_1, \dots, u_{n-1})$$

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]
```

proporciona exactamente lo mismo. Emplazamos al lector a usar `%%timeit` para comparar los tiempos de ambos métodos.

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

```
4.000000000000000000e+00 1.000000000000000000e+00
1.600000000000000000e+01 9.000000000000000000e+00
3.600000000000000000e+01 2.500000000000000000e+01
6.400000000000000000e+01 4.900000000000000000e+01
1.000000000000000000e+02 8.100000000000000000e+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(<función>)
```

aunque es necesario advertir que puede que dicho código no esté disponible para algunas funciones.

Por último, cuando no concemos 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.
```

que nos proporciona un listado de todas las funciones en las que aparece el término *decomposition*.

4.11 ACELERACIÓN DE CÓDIGO

Hemos comentado al inicio del capítulo que para obtener un buen rendimiento con NumPy es obligado evitar a toda costa el uso de bucles. Sin embargo, en ocasiones es difícil eludir el uso de bucles. En esta sección mostraremos el uso del módulo `numba` gracias al cual es posible obtener tiempos de ejecución próximos a los que se obtendrían al compilar código em lenguaje C, o similar.

Veamos el funcionamiento a través del siguiente ejemplo. El *fractal de Mandelbrot* es un objeto del plano complejo que se genera de forma muy sencilla. Dado un número complejo cualquiera c, se construye la sucesión por recurrencia siguiente:

$$z_0 = 0, \qquad z_{n+1} = z_n^2 + c, \quad n \geq 0$$

En función del número c escogido, esta sucesión puede estar acotada o no. Por ejemplo, si $c = 2$, la sucesión que se genera es $0, 2, 6, 38, \dots$ que es claramente no acotada; pero si $c = -1$, la sucesión queda $0, -1, 0, -1, \dots$ que sí está acotada. El conjunto de Mandelbrot se define como el conjunto de números complejos c tales que la sucesión anterior permanece acotada. Es decir, -1 es un punto que está en el conjunto de Mandelbrot, pero 2 no pertenece a dicho conjunto.

Se puede probar que, para un c dado, si algún elemento de la sucesión verifica que $|z_n| > 2$, entonces la sucesión no está acotada, lo que nos permite establecer un método para determinar si un número del plano complejo pertenece o no al fractal. La siguiente función, genera un *array* 2D que corresponde a puntos del plano complejo analizados, y que vale 1 (`True`) si el punto está en el conjunto, y 0 (`False`) en caso contrario.

```
def mandelplot(siz, lim, xint, yint):
    img = np.zeros([siz, siz], bool)
    xamp=xint[1]-xint[0]
    yamp=yint[1]-yint[0]

    for y in range(siz):
        for x in range(siz):
            c = complex(x/siz*xamp + xint[0],y/siz*yamp +
                yint[0])
            z = 0
            for i in range(lim):
                z = z**2 + c
                if abs(z) > 2:
                    img[y,x] = False
                    break
            else:
                img[y,x] = True
    return img
```

Nótese que recorremos un conjunto de puntos de tamaño `siz`×`siz` en un rectángulo definido por unas tuplas `xint`, `yint`. El parámetro `lim` especifica cuántos elementos de la sucesión hemos de recorrer para decidir si el punto está o no en el conjunto.

A continuación definimos los parámetros para la llamada a la función:

```
puntos = 2000
limite = 200
xint=(-2.,1.)
yint=(-1.5,1.5)
```

y la propia llamada:

```
img= mandelplot(puntos,limite,xint,yint)
```

Una vez obtenida la matriz, podemos dibujar el *array* obtenido del siguiente modo:[6]

[6]En el Tema 6 se tratarán los gráficos con Matplotlib.

```
%matplotlib
import matplotlib.pyplot as plt

asp = (yint[1]-yint[0])/(xint[1]-xint[0])
plt.imshow(img,interpolation='bilinear',cmap='binary',
    aspect=asp)
plt.xticks([])
plt.yticks([])
plt.show()
```

dando lugar a la figura 4.1.

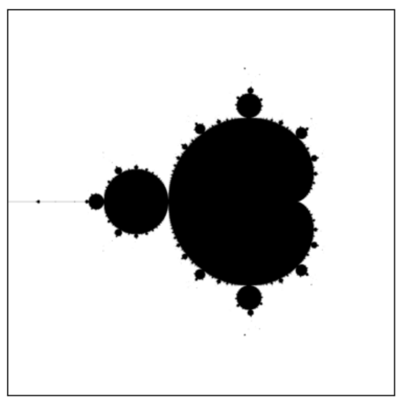


Figura 4.1: Fractal de Mandelbrot

Como el lector habrá podido comprobar si ha ejecutado las líneas anteriores, la evaluación de la función `mandelplot` toma su tiempo. Si lo medimos con `%%timeit`:[7]

```
%%timeit -r 1 -n 1
img= mandelplot(puntos,limite,xint,yint)
```

```
1min 19s ± 0 ns per loop (mean ± std. dev. of 1 run, 1
    loop each)
```

Nótese que la función consta de tres bucles anidados, los dos primeros recorriendo una matriz de 2000×2000, y no hemos usado ninguna función de NumPy que pueda acelerar el proceso.

Si bien es posible plantear el código para hacer uso de algunas funciones de NumPy, no parece que puedan evitarse todos los bucles. En estos casos es

[7] Usamos las opciones `-r 1 -n 1` para realizar una única iteración, dada que la evaluación toma demasiado tiempo.

cuando el módulo `numba` nos proporciona un resultado sorprendente con un esfuerzo mínimo.

Básicamente `numba` usa una infraestructura de compilación denominada LLVM (*Low Level Virtual Machine*) que permite compilar el código Python obteniendo un rendimiento similar al de C o FORTRAN. El uso de `numba` es bastante simple, funcionando a través de lo que se denomina un *decorador*. Bastará con cargar el módulo y escribir antes de la definición de la función el decorador; la función `mandelplot` quedaría:

```
from numba import jit

@jit
def mandelplot(siz, lim, xint, yint):
  ⋮
```

Si ahora volvemos a ejecutar

```
%%timeit -r 1 -n 1
img= mandelplot(puntos,limite,xint,yint)
```

```
2.16 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1
    loop each)
```

obtenemos un redimiento 30 veces más rápido. La función `jit` posee una serie de opciones que en ocasiones permiten afinar más la optimización, pero que no vamos a tratar en este texto.

4 12 EJERCICIOS

E4.1 Crear las siguientes matrices:

$$A = \begin{pmatrix} 10 & 13 & 16 & 19 & 22 & 25 & 28 \\ 11 & 14 & 17 & 20 & 23 & 26 & 29 \\ 12 & 15 & 18 & 21 & 24 & 27 & 30 \end{pmatrix}, \quad B = \begin{pmatrix} 12 & 18 & 24 & 30 & 36 \\ 42 & 48 & 54 & 60 & 66 \\ 72 & 78 & 84 & 90 & 96 \end{pmatrix}$$

$$C = \begin{pmatrix} 0 & 2 & 3 & 4 & 6 & 8 & 9 \\ 10 & 12 & 14 & 15 & 16 & 18 & 20 \end{pmatrix}$$

E4.2 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.

E4.3 Crear una función para construir una matriz de tamaño n con una estructura como la siguiente:

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}$$

E4.4 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.

E4.5 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.

E4.6 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

$$\begin{pmatrix} 13 & 14 \end{pmatrix} \quad \begin{pmatrix} 44 & 45 \\ 54 & 55 \end{pmatrix} \quad \begin{pmatrix} 12 \\ 22 \\ 32 \end{pmatrix} \quad \begin{pmatrix} 20 & 22 & 24 \\ 40 & 42 & 44 \end{pmatrix}$$

E4.7 Crea una matriz cuadrada de tamaño 8×8 con elementos 0 y 1 dispuestos como en un tablero de ajedrez. Por ejemplo, para dimensión 4, la matriz sería:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

E4.8 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 4×4. Calcula también las 4 matrices de tamaño 2×2 obtenidas al multiplicar tensorialmente las columnas de A con las filas de B.

E4.9 Usando *broadcasting* (ver ejercicio E4.6), obtener las matrices siguientes:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 4 & 9 & 16 \\ 4 & 9 & 16 & 25 \\ 9 & 16 & 25 & 36 \\ 16 & 25 & 36 & 49 \end{pmatrix}$$

E4.10 Crear una función para construir una matriz de tamaño n con una estructura como la siguiente:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 1 & 3 & 3 & \cdots & n-1 & n \\ 1 & 2 & 5 & \cdots & n-1 & n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 2 & 3 & \cdots & 2n-3 & n \\ 1 & 2 & 3 & \cdots & n-1 & 2n-1 \end{pmatrix}$$

E4.11 Para valores de n entre 2 y 20, calcula el determinante de la matriz siguiente:

$$\begin{pmatrix} 1 & n & n & \cdots & n \\ n & 2 & n & \cdots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n & n & n & \cdots & n \end{pmatrix}$$

E4.12 Encuentra la inversa de las matrices del tipo siguiente:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

E4.13 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:

$$\begin{pmatrix} 2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2 \end{pmatrix}$$

E4.14 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?

E4.15 Crea un array aleatorio de enteros entre 0 y 100 bidimensional, de tamaño 1000×1000. 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`.

E4.16 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.

E4.17 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 2×2, respectivamente, y K y D son matrices 2×2 con la siguiente forma:

$$K = \begin{pmatrix} k & 0.5 \\ 0.5 & k \end{pmatrix} \qquad D = \begin{pmatrix} -d & 1 \\ 1 & -d \end{pmatrix}$$

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.

E4.18 Resuelve el siguiente sistema de ecuaciones:

$$\left.\begin{array}{rcl} 2x - y - z & = & 4 \\ 3x + 4y - 2z & = & 11 \\ 3x - 2y + 4z & = & 11 \end{array}\right\}$$

E4.19 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 E4.5. Aplícala a la resolución del sistema del ejercicio E4.18.

E4.20 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.

5 SciPy

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 0.19.1 de SciPy, por lo que es posible que si el lector usa una versión distinta 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

Tabla 5.1: Módulos presentes en SciPy

Submódulo	**Descripción**
`cluster`	Algoritmos de agrupamiento
`constants`	Constantes físicas y matemáticas
`fftpack`	Rutinas para la Transformada Rápida de Fourier
`integrate`	Integración de ecuaciones diferenciales ordinarias
`interpolate`	Interpolación y splines regulares
`io`	Entrada y salida de datos
`linalg`	Álgebra lineal
`ndimage`	Procesamiento de imágenes
`odr`	Regresión ortogonal
`optimize`	Optimización y cálculo de raíces
`signal`	Procesamiento de señales
`sparse`	Matrices dispersas
`spatial`	Estructuras de datos espaciales
`special`	Funciones especiales
`stats`	Funciones estadísticas

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 Rosenbrook de N variables, definida por

$$f(x_0,\dots,x_{N-1}) = \sum_{i=1}^{N-1} \left[100(x_i - x_{i-1}^2)^2 + (1-x_{i-1})^2\right]$$

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 Rosenbrook:

```
def rosenbrook(x):
    return np.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:

```
import numpy as np
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
             e+00,
          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:

$$\frac{\partial f}{\partial x_0} = -400x_0(x_1 - x_0^2) - 2(1 - x_0)$$
$$\frac{\partial f}{\partial x_j} = 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j),\ 1 \leq j \leq N-2$$
$$\frac{\partial f}{\partial x_{N-1}} = 200(x_{N-1} - x_{N-2}^2)$$

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 = np.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:

```
opt1 = minimize(rosenbrook, x0, method='CG', jac=
    grad_rosenbrook, options=opciones)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 103
         Function evaluations: 205
         Gradient evaluations: 205
```

```
print(opt1.x)  # óptimo
```

```
[ 1.00000002  1.00000004  1.00000007  1.00000015
   1.0000003   1.0000006
  1.00000122]
```

Segunda forma: usamos la función **nuevarosenbrook** que devuelve la función objetivo y su gradiente de forma simultánea. En esta llamada, el parámetro `jac=True`:

```
opt2 = minimize(newrosenbrook, x0, method='CG', jac=True,
    options=opciones)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 103
         Function evaluations: 205
         Gradient evaluations: 205
```

```
print(opt2.x)  # óptimo
```

```
[ 1.00000002  1.00000004  1.00000007  1.00000015
   1.0000003   1.0000006
  1.00000122]
```

Por último, prescindimos del gradiente y hacemos que el método lo calcule de forma aproximada. Ahora `jac=False`:

```
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
```

```python
print(opt3.x)
```

```
[ 1.00000004  1.00000009  1.00000018  1.00000035
   1.00000071  1.00000144
  1.00000291]
```

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:

$$\begin{array}{rl} \text{Minimizar} & f(\mathbf{x}) \\ \text{sujeto a} & h_i(\mathbf{x}) = 0, \quad 1 \leq i \leq n, \\ & g_j(\mathbf{x}) \geq 0, \quad 1 \leq j \leq m, \\ & l_k \leq x_k \leq u_k, \quad 1 \leq k \leq N, \end{array}$$

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:

$$\begin{array}{rl} \text{Minimizar} & x_0^2 + 2x_1^2 - 2x_0x_1 - 2x_0 \\ \text{sujeto a} & x_0^3 - x_1 = 0, \\ & x_1 \geq 1. \end{array}$$

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] ])
```

La 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:

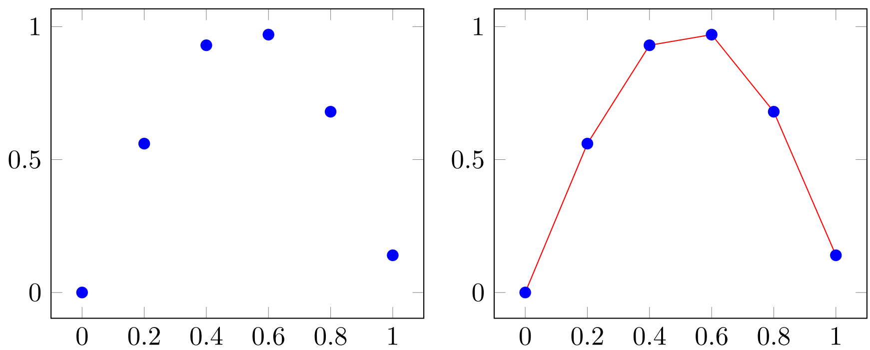
```
print(f(0.23))
```

```
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')
```

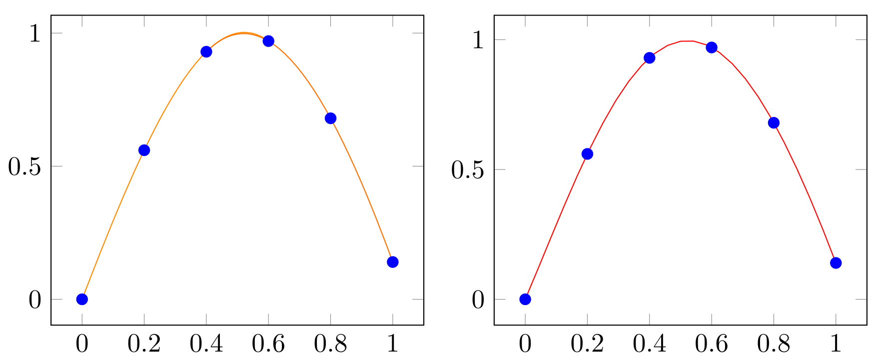
y obtener así una interpolación basada en *splines* de tercer orden. Otras opciones son `'slinear'` o `'quadratic'` para interpolación por *splines* de primer



(a) Puntos a interpolar (b) Función interpolada

Figura 5.1: Interpolación de datos unidimensional

o segundo orden, respectivamente. La figura 5.2a muestra la interpolación cúbica.



(a) *Spline* cúbico (b) Interpolación de Lagrange

Figura 5.2: Otras interpolaciones

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)
```

```
          5           4           3           2
-1.563 x + 6.771 x - 9.479 x + 1.604 x + 2.807 x
```

5 4

RESOLUCIÓN DE ECUACIONES DIFERENCIALES

El submódulo `integrate` del 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 ω y γ, 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`: del

```
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. En el capítulo siguiente veremos cómo podemos dibujar estos datos.

El método usa una aproximación del gradiente de la función del segundo miembro. Si podemos obtener este gradiente de forma exacta, podremos 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

E5.1 Resolver el siguiente problema de optimización:

$$\begin{array}{rl} \text{Maximizar} & xe^{xy} \\ \text{sujeto a} & x^2 + y = 0 \end{array}$$

E5.2 Resolver el siguiente problema de optimización:

$$\begin{array}{rl} \text{Minimizar} & (x-1)^2 + (y-2)^2 \\ \text{sujeto a} & x - 2y + 2 \geq 0 \\ & -x - 2y + 6 \geq 0 \\ & x, y \geq 0 \end{array}$$

E5.3 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?

E5.4 Para la función f del ejercicio E5.3, construye la interpolación lineal a trozos y mediante *splines* cúbicos en los puntos de abcisa 1, 2, 3 y 4. ¿Cuánto valen los interpoladores en el punto 2.5? ¿Y en 0?

E5.5 Resolver el siguiente problema (ecuación de Airy):

$$y'' - xy = 0$$
$$y(0) = \frac{1}{\sqrt[3]{3^2}\Gamma(\frac{2}{3})}, \quad y'(0) = -\frac{1}{\sqrt[3]{3}\Gamma(\frac{1}{3})}$$

donde $\Gamma(x)$ denota la función Gamma de Euler. La solucion exacta de esta ecuación es la función de Airy $y(x) = \mathrm{Ai}(x)$.
Indicación:. Las funciones Ai y Γ están en el submódulo `special`.

6 Gráficos con Matplotlib

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 particulamente 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.[1]

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. La versión de Matplotlib usada aquí es la 2.0.2.

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 disponemos de la *función mágica* `%matplotlib` para seleccionarlo. Si queremos los gráficos integrados en el propio entorno entonces

[1] *Graphical User Interface*: interfaz gráfica de usuario.

escribiremos

```
%matplotlib notebook
```

mientras que si queremos que los gráficos aparezcan en una ventana propia bastará poner

```
%matplotlib
```

```
Using matplotlib backend: TkAgg
```

que nos informa del *backend* seleccionado. Para manejar gráficos de forma interactiva sugerimos usar esta segunda opción. Es importante señalar que no podemos cambiar el *backend* una vez elegido a menos que reiniciemos el núcleo del entorno Jupyter.

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.

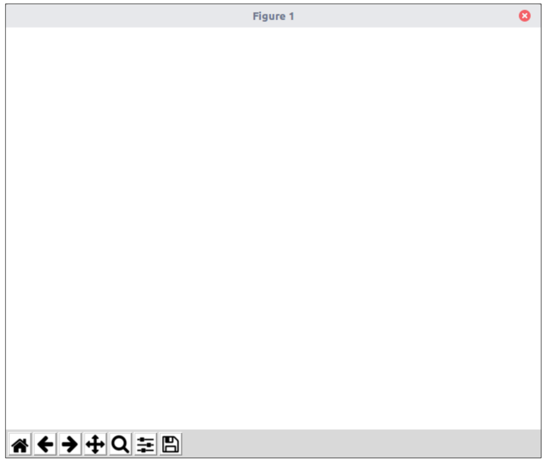


Figura 6.1: Ventana creada por la función `figure()`

Si por el contrario, la salida del 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 dados por los índices de la lista, y cuyas ordenadas son los correspondientes valores de la lista. En el caso de la figura 6.2, se crea una línea que une los puntos $(0,1)$, $(1,3)$ y $(2,2)$.

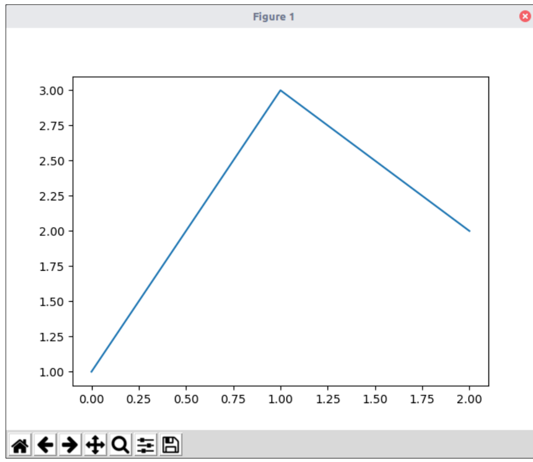


Figura 6.2: Gráfico dentro de la figura

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.[2]

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

[2]En lo que sigue, mostraremos simplemente los gráficos resultantes sin el marco de las ventanas.

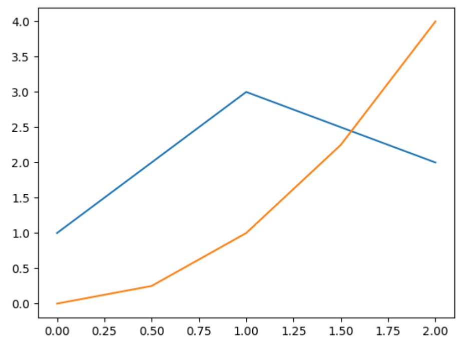


Figura 6.3: Dos gráficas en el mismo eje

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 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

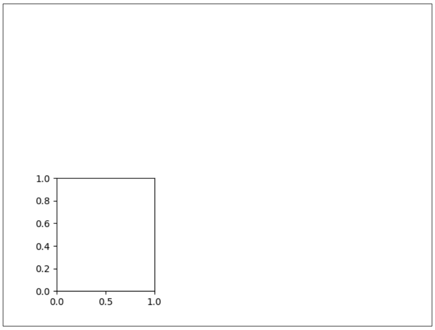
El posible tener varios ejes distintos en la misma ventana gráfica, para lo cual usaremos la orden:

```
subplot(n,m,k)
```

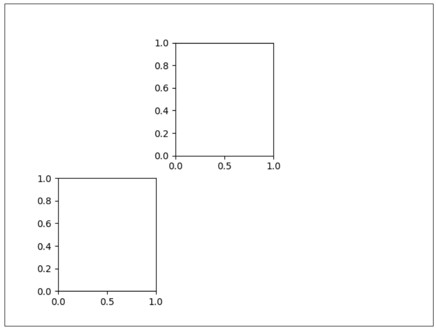
la cual divide la figura en n filas y m columnas y crea unos ejes en la posición k (contando de izquierda a derecha y de arriba a abajo). La comas de separación entre n, m y k no son necesarias (a menos que alguno de los valores tenga más de un dígito). Por ejemplo,

```
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



(a)



(b)

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.

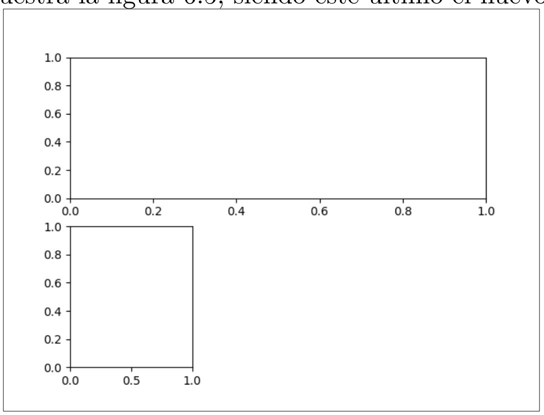


Figura 6.5: Distintos ejes en la misma figura

Como puede observarse, la función `subplot` permite organizar de forma estructurada cualquier combinación de ejes que se desee. El comando `plot` dibujará 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)
```

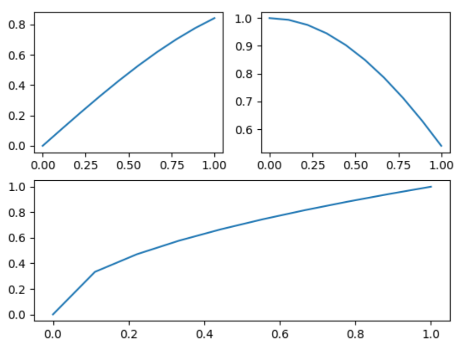


Figura 6.6: Varios gráficos en la misma figura

6.1.4 Axes

Es posible organizar los ejes creados en una figura de forma no estructurada con el comando `axes`:

```
axes()
```

que crea unos ejes por defecto, equivalentes 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.

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.[3]

[3]Nótese que el posicionamiento de los ejes por defecto corresponde a las coordenadas

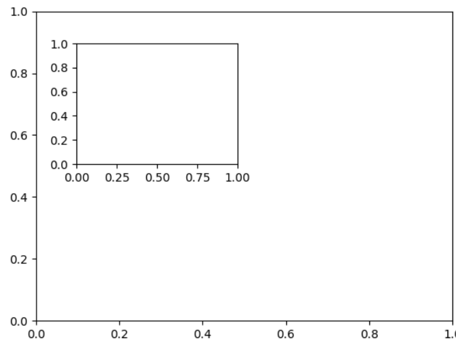


Figura 6.7: Ejes no estructurados

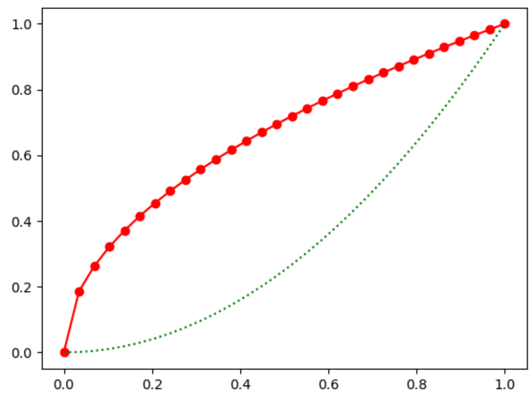
6.2 AÑADIENDO OPCIONES

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.

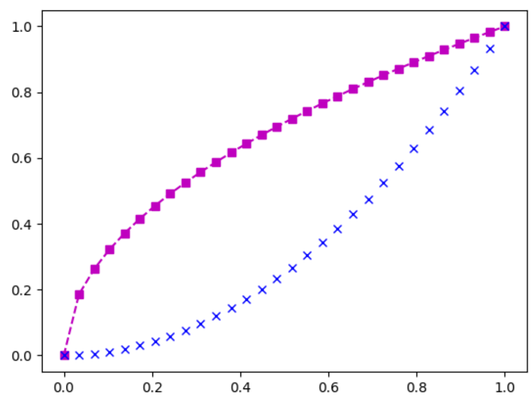
Como podemos ver en la figura 6.8, podemos usar, tras los *arrays* que determinan los puntos del gráfico, una cadena con diversos caracteres con los que configurar algunos aspectos de la línea usada, como el color, el estilo de línea o los marcadores que señalan cada uno de los puntos del gráfico. Por 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 magenta, línea discontinua y marcadores cuadrados, y la cadena `'bx'` dibuja en azul sólo marcadores con forma de cruz. La siguiente tabla muestra una breve representación de las opciones más comunes. Para una lista completa véase la ayuda del comando `plot`.

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`

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

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.

6.3 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 una de las líneas trazadas con `plot` mediante una leyenda, usando una etiqueta definida por el argumento opcional `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 LaTeX. 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.

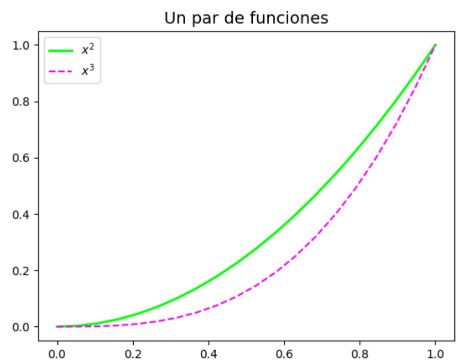


Figura 6.9: Título y leyenda

Texto y anotaciones

La figura 6.10 ha sido generada con el siguiente código:

```
axes(facecolor=(0.15,0.75,0.75))
x = linspace(0,1,20)
y = x**2
plot(x,y,'b-o')
text(x[12]-0.22,y[12],'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: `facecolor` proporciona el

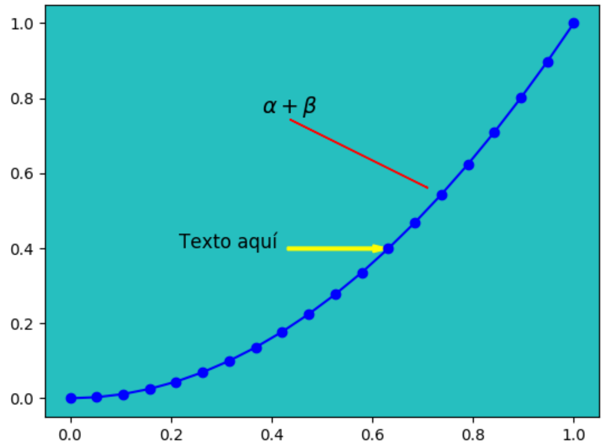


Figura 6.10: Texto y anotaciones

color de fondo de los ejes;[4] en este caso, el color se ha determinado a través de una tupla de valores reales entre 0 y 1 en formato *RGB*.[5]

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 en el segundo `text` contiene notación LaTeX que involucra un carácter especial la hemos pasado como *raw*. 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 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:

[4] Esta opción sustituye a `axisbg`, aunque ésta última sigue en uso.
[5] Red, Green, Blue.

```
from numpy.random import rand
scatter(rand(1000),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()
```

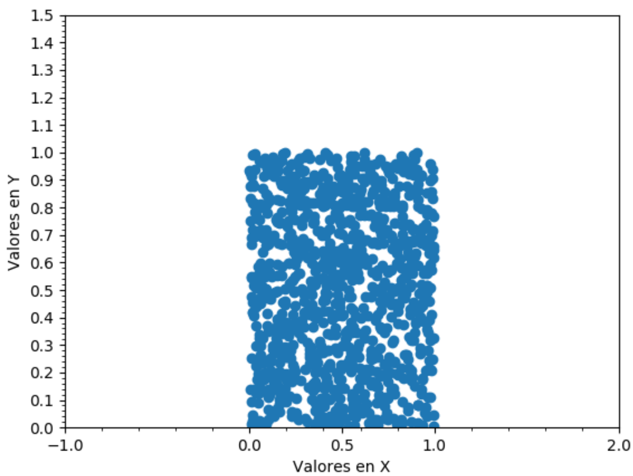


Figura 6.11: Etiquetas en los ejes

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`). No sólo se pueden apreciar las posibilidades de creación de gráficos sino que además se puede ver el código con el que se generan.[6] Puesto que este código usa extensivamente los objetos y métodos del módulo, veremos en la siguiente sección cómo trabajar con éstos.

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 (objetos), 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)
```

Creamos unos datos y dibujamos en cada uno de los ejes:

```
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

[6] 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.

para almacenar el objeto gráfico y no la lista.[7] 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)
```

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.

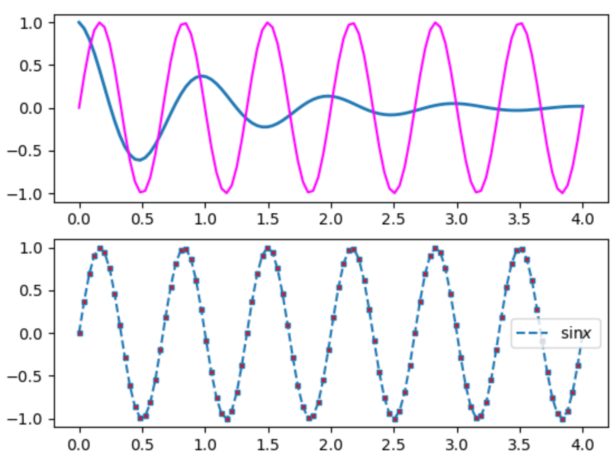


Figura 6.12

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.

[7]Podríamos haber hecho también `a, c = ax1.plot(x,y,x,z)` para almacenar los elementos de la lista en variables separadas.

6.4.1 Un poco más de sofisticación

Veamos algún ejemplo más en el que mostraremos otras propiedades del gráfico que podemos controlar fácilmente.

```
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)
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
```

El código anterior genera 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 LaTeX.

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.[8] Para otras opciones, consultar la ayuda.

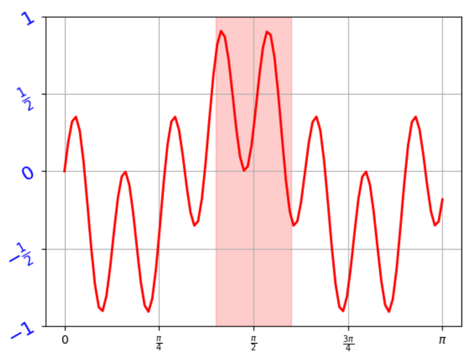


Figura 6.13

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')
```

[8]También podemos salvar los gráficos tanto desde la ventana gráfica como desde el gráfico incrustado en el entorno Jupyter.

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

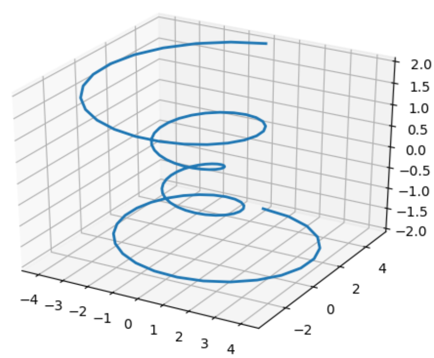


Figura 6.14: Curva en 3D

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:

```
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)
```

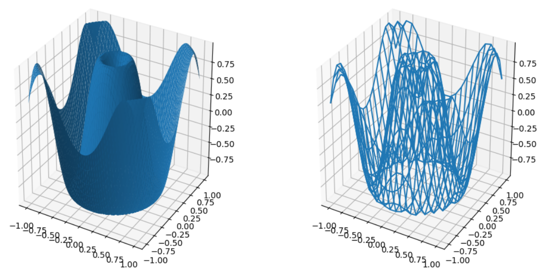


Figura 6.15: Superfices en 3D

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,alpha=0.2)
```

El parámetro `zdir` señala el eje sobre el que se dibujan los contornos, mientras 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`.

6 6 EJERCICIOS

E6.1 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 cículo 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.

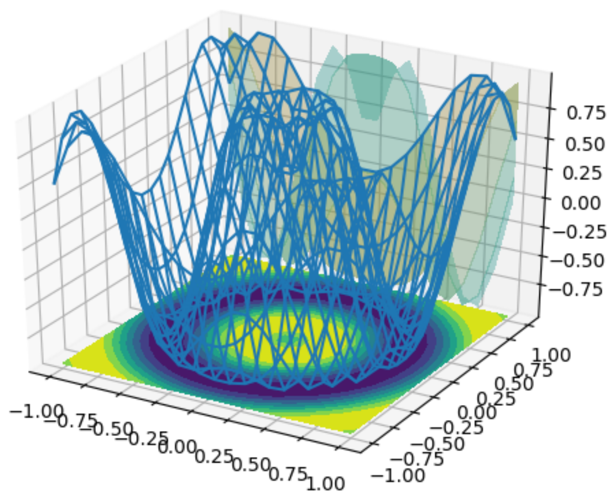


Figura 6.16: Curvas de nivel en 3D

E6.2 En el ejercicio E4.17 del Tema 4 de 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.

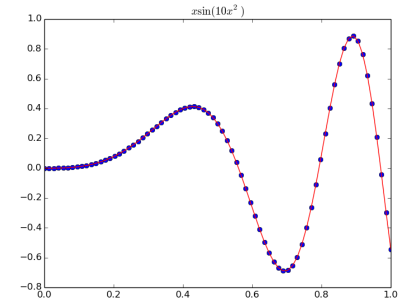
E6.3 Considera la función $f(x) = \sin(3x)\cos(5x-1)$ en el intervalo $[0, 1]$. Encuentra los máximos y mínimos usando la función `minimize_scalar` del módulo `optimize` de SciPy. Dibuja la función y señala los puntos obtenidos, anotándolos con texto.
Indicación: la función `minimize_scalar` usa una 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.

E6.4 Reproducir de la forma más aproximada los gráficos de la figura E6.4.

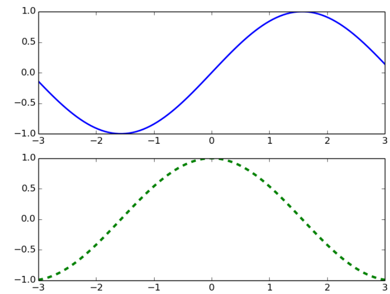
E6.5 Dibujar las siguientes funciones en los recintos indicados:

(a) $f(x, y) = e^{-x^2-y^2}$ en $[-2, 2]^2$.

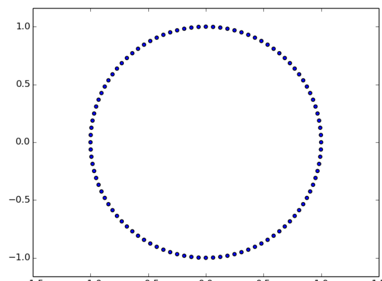
(b) $f(x, y) = e^{x^2}(x^4 + y^4)$ en $[-1, 1]^2$.



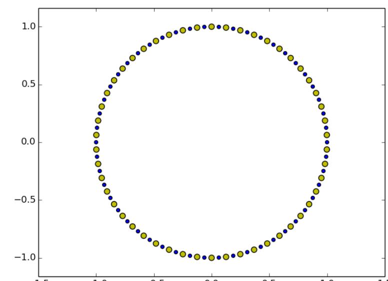
(a) $x\,\mathrm{sen}(10x^2)$



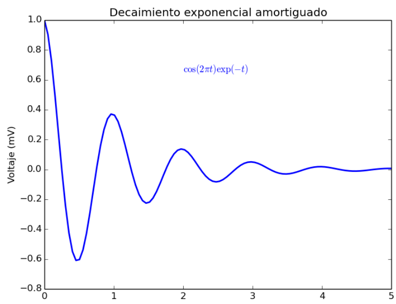
(b) $\mathrm{sen}\,x$ y $\cos x$



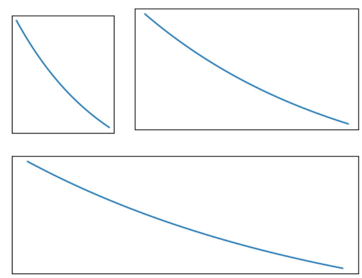
(c) 100 ptos. en el círculo unida d



(d) 100 ptos. en el círculo unidad



(e) $\cos(2\pi t)e^t$



(f) e^{-x} en $(0,1)$

Figura 6.17: Ejercicio E6.4

(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.

E6.6 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.

E6.7 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 SymPy

El módulo SymPy es una librería escrita en Python para realizar cálculo simbólico que se ha convertido en una alternativa muy eficiente a otros CAS (*Computer Algebraic System*) como pueden ser *Maple* o *Mathematica*. En estas notas usaremos la versión 1.1.1.

7.1 VARIABLES SIMBÓLICAS

A diferencia de lo recomendado en capítulos anteriores en referencia a la importación masiva de módulos, es habitual importar el módulo SymPy de forma masiva:

```
from sympy import *
```

Una de las características sobresalientes de SymPy es la espléndida impresión de las respuestas, especialmente en el entorno Jupyter Notebook. Para activar este tipo de salidas hemos de ejecutar la función

```
init_printing()
```

De este modo si ahora escribimos,

```
sqrt(3) + sqrt(12)
```

$3\sqrt{3}$

lo que obtenemos es una respuesta exacta de la operación introducida (no una aproximación real, como ocurriría si empleáramos el módulo `math`), y además perfectamente formateada usando MathML.[1]

[1] *Mathematical Markup Language* es un lenguaje de marcado para poder expresar notación matemática que permite su visualización inmediata en navegadores web, entre otras aplicaciones.

Pero sin duda, la potencia del cálculo simbólico reside en el manejo de variables simbólicas, que nos permite manipular cualquier expresión matemática. Usaremos la función `symbols` para definir variables simbólicas y la notación matemática habitual para escribir cualquier expresión.

```
x, y, z = symbols('x y z')
expr = x + y**2 - log(z)
expr
```

$x + y^2 - \log(z)$

La función `symbols` admite como parámetro una cadena de caracteres separada por espacios o comas y asigna cada una de ellas a una variable. Así, en la entrada anterior se han creado tres variables simbólicas `x`, `y`, `z` asociadas a las expresiones x, y y z. Es importante señalar que las variables asociadas nada tienen que ver con el nombre asignado.

```
a, b, c, d = symbols('alpha,x_1 a,hola')
a, b, c, d
```

$(\alpha, \quad x_1, \quad a, \quad hola)$

En el ejemplo anterior hemos asignado la variable simbólica `a` a la expresión α, la cual es interpretada por SymPy mediante su correspondiente símbolo griego. Del mismo modo, la variable `b` representa a la expresión x_1, mientras que `c` se ha asignado a a, lo cual es confuso, pero válido. Por último, la variable simbólica `d` hace referencia a una expresión con un nombre arbitrario. Obviamente se recomienda asignar a cada expresión una variable que la represente de forma precisa sin crear confusión. También es importante señalar que son las variables simbólicas, esto es, la salida de la función `symbols`, las que debemos manipular a la hora de realizar los cálculos:

```
c**2
```

x_1^2

```
alpha + 1
```

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

El argumento de la función `symbols` admite cierta flexibilidad para poder definir un número arbitrario de variables simbólicas. Por ejemplo, podemos obtener una colección de variables definiendo

```
a = symbols('a1:10')
a
```

$(a_1, \quad a_2, \quad a_3, \quad a_4, \quad a_5, \quad a_6, \quad a_7, \quad a_8, \quad a_9)$

En el entorno Jupyter, si en lugar de `a` escribimos `print(a)` no obtenemos la salida formateada con MathML sino en formato habitual, lo cual puede ser un engorro si queremos imprimir varias salidas bien formateadas en una misma celda. Si en lugar de `print` usamos `display` obtendremos la salida deseada.

7.1.1 Hipótesis

La función `symbols` permite usar un parámetro adicional con el que imponer hipótesis sobre las variables creadas. Por ejemplo, las siguientes líneas definen una variable entera y una real

```
n = symbols('n', integer=True)
x = symbols('x', real=True)
```

Si ahora escribimos

```
cos(2*pi*n)
```

1

observamos que, en efecto, el coseno de un múltiplo entero de 2π es 1. Sin embargo,

```
cos(2*pi*x)
```

$\cos(2\pi x)$

no es posible obtener una simplificación de la expresión $\cos(2\pi x)$ si x es real.

Las hipótesis posibles son `commutative`, `complex`, `imaginary`, `real`, `integer`, `odd`, `even`, `prime`, `composite`, `zero`, `nonzero`, `rational`, `irrational`,, `algebraic`, `transcendental`, `finite`, `infinite`, `negative`, `nonnegative`, `positive`, `nonpositive`, `hermitian`, `antihermitian`.

Podemos preguntar si una variable es de alguno de estos tipos con el método `is_`*tipo*, teniendo en cuenta que todas las variables pertenecen al mayor conjunto posible. Por ejemplo,

```
print(n.is_complex)
print(n.is_irrational)
print(n.is_even)
```

```
True
False
None
```

La respuesta puede ser cierta, falsa, o bien `None`, que equivale a que no se puede saber a priori. Más adelante veremos cómo usar las hipótesis para ayudar en las simplificaciones.

7.1.2 Sustituciones

Otro aspecto a tener en cuenta es la distinción entre variables de Python y variables simbólicas. Por ejemplo,

```
x = symbols('x')
expr = x + 1
x = 2
expr
```

$x + 1$

Al cambiar el valor de la variable `x` a `2`, ésta deja de ser una variable simbólica, pero la expresión simbólica `expr` permanece inalterada. Si lo que pretendemos es obtener el valor de una expresión al sustituir una variable simbólica por un determinado valor hemos de usar el método `subs`:

```
a, b = symbols('a b')
expr = a**2 + b**4
expr.subs({a:5,b:2})
```

41

que tiene como argumento un diccionario en el que asignamos los valores correspondientes. Si queremos sustituir un único valor, no es necesario el diccionario

```
expr.subs(a,3)
```

$b^4 + 9$

La sustitución también puede hacerse sobre cualquier expresión simbólica:

```
expr.subs(a,b**2)
```

$2b^4$

7.1.3 Manejo de números

SymPy permite manejar números reales con precisión arbitraria y realizar cálculos exactos, pero hay que tener la precaución de definir correctamente los valores. Para manejar números enteros disponemos de la función `Integer`

```
Integer(1)/Integer(3)
```

$\frac{1}{3}$

que podemos escribir abreviadamente como

```
Integer(1)/3
```

para obtener el mismo resultado. Otra alternativa para manejar números racionales nos la da la función `Rational`. Así,

```
Rational(1,3) + 1
```

$$\frac{4}{3}$$

Hay que ser cuidadoso con las expresiones numéricas en Python o SymPy, pues

```
print(1/3)
print(Integer(1/3))
```

```
0.3333333333333333
0
```

son diferentes debido a que en el primer caso no estamos usando SymPy, de ahí que la respuesta sea el `float` que proporciona Python tal cual, mientras que en el segundo caso se ha aplicado la función `Integer` al número real, de ahí que se obtenga su parte entera. Del mismo modo,

```
print(cos(2/3*pi))
print(cos(2*pi/3))
```

```
cos(0.666666666666667*pi)
-1/2
```

¿Puede el lector deducir por qué se produce este comportamiento? Es importante tener presente la diferencia que resulta al usar operaciones numéricas en Python o con el módulo SymPy. No obstante, a veces el resultado puede ser engañoso,

```
print(2**(1/2))
print(Pow(2,1/2))
Pow(2,Integer(1)/2)
```

```
1.4142135623730951
1.41421356237310
```

$$\sqrt{2}$$

Dado que la función `Pow` de SymPy es equivalente a la potencia, los dos últimos resultados deberían ser los mismos, y de hecho lo son:

```
print(2**(1/2)*2**(1/2))
print(Pow(2,1/2)*Pow(2,1/2))
Pow(2,Integer(1)/2)*Pow(2,Integer(1)/2)
```

```
2.0000000000000004
2.00000000000000
```

2

En el primer caso, dado que `2**(1/2)` es un `float`, el resultado no es exactamente 2 debido a los errores de redondeo. No es así en los otros dos casos, pues en ambos se está usando la expresión *exacta* de $\sqrt{2}$; lo que ocurre es que `Pow(2,1/2)` proporciona una expresión decimal, mientras que la última es una expresión simbólica.

En SymPy podemos obtener el valor numérico con un número arbitrario de cifras decimales. El método `evalf`, sin parámetros, nos proporciona una aproximación con 15 cifras:

```
sqrt(2).evalf()
```

1.4142135623731

mientras que si llamamos con un parámetro entero el resultado aparecerá con el número de cifras decimales precisado con el parámetro

```
sqrt(2).evalf(50)
```

1.4142135623730950488016887242096980785696718753769

Para evaluar numéricamente una expresión con sustitución, disponemos de un parámetro `subs` que funciona de forma muy similar al método del mismo nombre:

```
expr = a**2 + b**4
expr.evalf(subs = {a:Rational(1,3), b:sqrt(2)})
```

4.11111111111111

Existen dos expresiones equivalentes para el método `evalf`: el método `n` y la función `N`, de manera que las siguientes expresiones son idénticas

```
sqrt(2).evalf(5)
N(sqrt(2),5)
sqrt(2).n(5)
```

También pueden ser usados sin ningún parámetro.

Finalmente, el uso de números complejos en SymPy difiere de la forma habitual en Python. La unidad imaginaria es `I` y se usa como un símbolo más, por lo que la definición de un número complejo ha de llevar los correspondientes signos aritméticos (a diferencia de lo que ocurre con los números complejos en Python). Es decir, los números $1 + 3i$ y $4 - i$ se escribirán

```
1+3*I
4-I
```

El número e también tiene un símbolo propio en SymPy, que es `E`, por lo que conviene evitar usar esta letra como variable simbólica.

7 2 SIMPLIFICACIÓN

Uno de los aspectos notables del cálculo simbólico es la simplificación de expresiones. SymPy dispone de un buen número de funciones para efectuar diversas simplificaciones sobre diferentes tipos de expresiones. De entre todas, la más general es la función, que también puede usarse como método, `simplify`, que trata de encontrar la forma más sencilla de una determinada expresión.

```
x, y, z = symbols('x y z')
expr = sin(x)**2 + cos(x)**2
expr.simplify()
```

1

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

$x - 1$

El problema es que a veces no es fácil determinar qué se entiende por la *forma más sencilla* de una expresión. Por ejemplo, podríamos querer obtener la siguiente expresión

$$(x-2)(x-3) = x^2 - 5x + 6$$

```
simplify((x-2)*(x-3))
```

$(x - 3)(x - 2)$

```
(x**2-5*x+6).simplify()
```

$x^2 - 5x + 6$

pero como podemos observar, las simplificaciones no ha tenido efecto. Es aquí donde debemos usar otras funciones más específicas:

```
expand((x-2)*(x-3))
```

$x^2 - 5x + 6$

```
(x**2-5*x+6).factor()
```

$(x - 3)(x - 2)$

Estas funciones no son exclusivas de polinomios:

```
expand((sin(x)+2*cos(x))**3)
```

$\sin^3(x) + 6\sin^2(x)\cos(x) + 12\sin(x)\cos^2(x) + 8\cos^3(x)$

```
factor(cos(x)**2 + 2*cos(x)*sin(x) + sin(x)**2)
```

$(\sin(x) + \cos(x))^2$

La función `collect` agrupa potencias de una expresión:

```
expr = x**2*y - x**2 - 4*x*y + 4*x + 4*y - 4
collect(expr,x)
```

$x^2(y-1) + x(-4y+4) + 4y - 4$

mientras que `cancel` simplifica factores comunes en expresiones racionales, de forma similar a `factor`, aunque la salida es ligeramente diferente:

```
expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(
    x**2 - 1)
cancel(expr)
```

$\frac{1}{x-1}\left(y^2 - 2yz + z^2\right)$

```
factor(expr)
```

$\frac{(y-z)^2}{x-1}$

Para expresiones racionales, también es muy útil la función `apart` que realiza una descomposición en suma de fracciones irreducibles:

```
expr = (x**3-3*x**2+x+2)/(x**2+2*x+1)
```

$x - 5 + \frac{10}{x+1} - \frac{3}{(x+1)^2}$

7.2.1 Expresiones trigonométricas, logarítmicas y potenciales

Para otro tipo de expresiones disponemos de funciones más específicas, que mostramos en los siguientes ejemplos:

```
expand_trig(sin(2*x))
```

$2\sin(x)\cos(x)$

```
trigsimp(cos(x)**2-sin(x)**2)
```

$\cos(2x)$

```
powsimp(x**a*x**b)
```

x^{a+b}

Sin embargo, hay que tener en cuenta que ciertas simplificaciones sólo son correctas bajo hipótesis concretas sobre las variables. Por ejemplo, es bien conocido que

$$x^a y^a = (xy)^a$$

pero en realidad, esta expresión no es cierta si $x = -1$, $y = -1$ y $a = \frac{1}{2}$, pues $x^a y^a = \sqrt{-1}\sqrt{-1} = i \cdot i = -1$, mientras que $(xy)^a = \sqrt{(-1)(-1)} = 1$. De este modo,

```
powsimp(x**a*y**a)
```

$x^a y^a$

no realiza la simplificación, pues como hemos visto, no es cierta en general. Pero si imponemos las hipótesis adecuadas,

```
x, y = symbols('x y', positive=True)
a = symbols('a', real=True)
powsimp(x**a*y**a)
```

$(xy)^a$

Si no queremos tener que lidiar con las hipótesis pertinentes, podemos usar el parámetro `force`:

```
x, y, a = symbols('x y a')
expand_power_base((x*y)**a, force=True)
```

$x^a y^a$

```
expand_log(log(x*y), force=True)
```

$\log(x) + \log(y)$

Finalmente, entre otras muchas funciones específicas para manipular y simplificar expresiones, mostramos la función `rewrite` con la que escribir una expresión en función de otra. Por ejemplo,

```
sin(x).rewrite(tan)
```

$$\frac{2\tan\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right) + 1}$$

7.2.2 Identidades

En SymPy, el signo `==` no representa una igualdad simbólica, es decir,

```
(x+1)**2 == x**2 + 2*x + 1
```

```
False
```

debido a que ambas expresiones son estructuralmente diferentes. Si queremos averiguar si una identidad es o no cierta tenemos dos opciones: establecer una ecuación entre ambas y simplificar,

```
a = (x+1)**2
b = x**2 + 2*x +1
Eq(a,b).simplify()
```

```
True
```

o bien simplificar su diferencia:

```
simplify(a-b)
```

0

7 3 RESOLUCIÓN DE ECUACIONES ALGEBRAICAS

Incialmente, SymPy ha usado la función `solve` para resolver tanto ecuaciones como sistemas, pero las últimas versiones del módulo recomiendan el uso de otras alternativas.

La función `solveset` permite resolver ecuaciones algebraicas de cualquier tipo. Su sintaxis es sencilla, debemos proporcionar una ecuación y la variable que queremos resolver. Por ejemplo, para resolver la ecuación $x^3 = 1$ escribiremos

```
x, y = symbols('x y')
solveset(Eq(x**3,1),x)
```

$$\left\{1, \quad -\frac{1}{2} - \frac{\sqrt{3}i}{2}, \quad -\frac{1}{2} + \frac{\sqrt{3}i}{2}\right\}$$

Como vemos, el resultado es un conjunto con las soluciones de la ecuación. Alternativamente, podemos escribir la ecuación como una igualdad a cero, y pasarla directamente

```
a, b, c = symbols('a b c')
solveset( a*x**2 + b*x + c, x)
```

$$\left\{\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right), \quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right\}$$

Para resolver un conjunto de ecuaciones con respecto a varias variables, disponemos de las funciones `linsolve` y `nonlinsolve` para sistemas lineales y no lineales, respectivamente. Podemos pasar las expresiones de las ecuaciones y variables como listas,

$$\begin{cases} x^2 + 2y^2 = 4 \\ x - 3y = 2 \end{cases}$$

```
nonlinsolve([x**2 + 2*y**2 - 4, x-3*y-2], [x,y])
```

$$\left\{\left(-\frac{14}{11}, \quad -\frac{12}{11}\right), \quad (2, \quad 0)\right\}$$

o en el caso de sistemas lineales, también como matrices[2]

```
M = Matrix([[1,1,1,1],[2,3,1,5]])
sistema = A, b = M[:,:-1], M[:,-1]
linsolve(sistema,[x,y,z])
```

$$\{(-2z - 2, \quad z + 3, \quad z)\}$$

Si `solveset` no es capaz de encontrar solución devuelve un *conjunto condicional*

```
solveset(cos(x)-x,x)
```

$$\{x \mid x \in \mathbb{C} \wedge -x + \cos(x) = 0\}$$

que no hay que confundir con el caso en el que no hay solución

```
solveset(exp(x),x)
```

$$\emptyset$$

en cuyo caso devuelve el conjunto vacío.

7 4 ÁLGEBRA MATRICIAL

La matrices o vectores en SymPy se definen mediante la función `Matrix`, de forma similar a un *array* de NumPy, pero además del carácter simbólico de sus elementos, es decir, que se pueden incluir variables simbólicas entre sus elementos y los números son tratados de forma exacta, hay alguna que otra sutil diferencia.

Para definir una matriz damos una lista con las filas de la misma

```
A = Matrix([[x,0,1],[2,0,-1],[1,-1,y]])
A
```

[2]En la sección 7.4 se puede ver la sintaxis de la función `Matrix`.

$$\begin{bmatrix} x & 0 & 1 \\ 2 & 0 & -1 \\ 1 & -1 & y \end{bmatrix}$$

pero si proporcionamos una única lista

```
B = Matrix([1,0,-1])
B
```

$$\begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix}$$

lo que obtenemos es una matriz columna, o vector. El operador de multiplicación denota, por defecto, la multiplicación matricial, y obviamente las dimensiones han de ser compatibles:

```
B.T * A
```

$$\begin{bmatrix} x-1 & 1 & -y+1 \end{bmatrix}$$

Los operadores aritméticos son los habituales y se puede calcular la inversa mediante

```
A**-1
```

$$\begin{bmatrix} \frac{2}{2x+4} & \frac{2}{2x+4} & 0 \\ \frac{-4y-2}{-2x-4} & \frac{x(2y+1)-x-2}{-2x-4} & \frac{2x+4}{-2x-4} \\ \frac{2}{x+2} & -\frac{x}{x+2} & 0 \end{bmatrix}$$

o el determinante:

```
A.det()
```

$$-x-2$$

Hay aspectos en los que el manejo se hace más sencillo que con NumPy, como la extracción de filas o columnas

```
A.row(1)
```

$$\begin{bmatrix} 2 & 0 & -1 \end{bmatrix}$$

```
A.col(0)
```

$$\begin{bmatrix} x \\ 2 \\ 1 \end{bmatrix}$$

la inserción de nuevas filas o columnas

```
A.col_insert(1,B)
```

$$\begin{bmatrix} x & 1 & x & 0 \\ 2 & 0 & 2 & 0 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

o su eliminación

```
A.row_del(2)
A
```

$$\begin{bmatrix} x & 0 & 1 \\ 2 & 0 & -1 \end{bmatrix}$$

Nótese que este último método no devuelve nada, pero la matriz es modificada internamente, a diferencia de los anteriores, que retornan la nueva matriz, pero no modifican la original.

7.4.1 Construcción de matrices

Al igual que en NumPy tenemos diversas funciones para la creación de matrices con cierta estructura

```
A = eye(3)
B = zeros(2)
C = ones(3,2)
A, B, C
```

$$\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}\right)$$

aunque el uso puede ser ligeramente distinto. Por ejemplo, con la función `diag`

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

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

```
diag([1,2],eye(2))
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

7.4.2 Álgebra lineal

SymPy también dispone de algunas funciones para obtener el núcleo de una matriz

```
A = Matrix([[3, -1, 1, 0, 1], [1, 0, 0, 0, 1]])
k = A.nullspace()
k
```

$$\left[\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -2 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right]$$

esto es, una base del espacio de vectores tales que $A\mathbf{x} = \mathbf{0}$, como fácilmente podemos comprobar

```
for x in k:
    print(A*x)
```

```
Matrix([[0], [0]])
Matrix([[0], [0]])
Matrix([[0], [0]])
```

También podemos obtener el espacio imagen de A, o espacio de las columnas,

```
A.columnspace()
```

$$\left[\begin{bmatrix} 3 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right]$$

o el rango de la misma

```
A.rank()
```

2

Sin embargo, si la matriz contiene símbolos indefinidos no se realiza un estudio en función de los mismos, por lo que los resultados no son correctos. Por ejemplo,

```
A = Matrix([[x,0,1],[2,0,-1],[1,-1,y]])
A.rank()
```

3

Pero si $x=-2$,

```
A.subs(x,-2).rank()
```

2

Autovalores, autovectores, diagonalización y forma de Jordan son fáciles de obtener con SymPy,

```
A = Matrix([[1,0,2,-6],[0,1,-1,3],[0,0,1,3],[0,0,0,2]])
A.eigenvals()
```

$$\{1:3,\quad 2:1\}$$

Como vemos, los autovalores vienen en forma de diccionario. Esta matriz tiene un autovalor 1 de multiplicidad 3 y otro autovalor 2 de multiplicidad 1. Para los autovectores,

```
A.eigenvects()
```

$$\left[\left(1,\quad 3,\quad \left[\begin{bmatrix}1\\0\\0\\0\end{bmatrix},\ \begin{bmatrix}0\\1\\0\\0\end{bmatrix}\right]\right),\ \left(2,\quad 1,\quad \left[\begin{bmatrix}0\\0\\3\\1\end{bmatrix}\right]\right)\right]$$

obtenemos una lista con los autovalores, multiplicidad y autovectores asociados a cada autovalor. La forma de Jordan se obtiene:

```
A.jordan_form()
```

$$\left(\begin{bmatrix}2&0&1&0\\-1&0&0&0\\0&1&0&3\\0&0&0&1\end{bmatrix},\ \begin{bmatrix}1&1&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&2\end{bmatrix}\right)$$

donde la primera de las matrices es la matriz de paso, y la segunda la forma de Jordan.

7 5 CÁLCULO

7 5 1 Límites

SymPy puede calcular el límite de una función en un punto, inclusive en el infinito:

```
x = symbols('x')
limit(sin(3*x)/x,x,0)
```

3

```
limit(x*tan(pi/x),x,oo)
```

π

Nótese el uso del símbolo `oo` para infinito. También es posible obtener límites laterales

```
limit(1/x,x,0,dir="-")
```

$-\infty$

Además, la función `Limit` nos proporcional el objeto sin evaluar

```
Limit((exp(2*x)-1)/x,x,0)
```

$$\lim_{x\to 0^+}\left(\frac{1}{x}\left(e^{2x}-1\right)\right)$$

y podemos evaluarlo con el método `doit`:

```
_.doit()
```

2

Obsérvese que por defecto, el límite realizado es por la derecha.

7 5 2 Derivación

La derivación se lleva a cabo con la función `diff`:

```
x, y = symbols('x y')
f = cos(x**2)
diff(f,x)
```

$-2x\sin\left(x^2\right)$

La función también puede ser usada como método, y se permite bastante flexibilidad para indicar derivadas de orden superior. Por ejemplo, la derivada tercera

```
f.diff(x,3)
```

$4x\left(2x^{2}\sin\left(x^{2}\right)-3\cos\left(x^{2}\right)\right)$

aunque también se puede escribir así:

```
f.diff(x,x,x)
```

SymPy permite derivar expresiones respecto de varias variables:

```
diff(sin(x*y),x,y,y)
```

$-x\left(xy\cos\left(xy\right)+2\sin\left(xy\right)\right)$

que corresponde a la derivada con respecto a x una vez, y con respecto a y dos veces. Lo que en notación matemática es

```
Derivative(sin(x*y),x,y,y)
```

$$\frac{\partial^{3}}{\partial x\partial y^{2}}\sin\left(xy\right)$$

Al igual que antes, podemos evaluar con el método `doit`.

7.5.3 Integrales

Para integrar usaremos la función o método `integrate`. La integral puede ser tanto indefinida, esto es, el cálculo de una primitiva, la cual se lleva a cabo sin adición de constantes:

```
integrate(cos(x),x)
```

$\sin(x)$

como definida, en la que debemos proporcionar una tupla con la variable de integración y los límites:

```
integrate(exp(-x),(x,0,oo))
```

1

Como vemos, se pueden calcular integrales impropias, y con respecto a varias variables:

```
integrate(exp(-x**2 - y**2), (x, -oo, oo), (y, -oo, oo))
```

π

La función `Integral` nos proporciona el objeto sin evaluar:

```
Integral(1/(1+x**2),x)
```

$$\int \frac{1}{x^2+1}\,dx$$

```
_.doit()
```

$\operatorname{atan}(x)$

7.5.4 Series de Taylor

Podemos obtener la expansión en forma de serie de Taylor alrededor de un punto, hasta un orden dado

```
f = log(1+x)
f.series(x,0,6)
```

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} + \mathcal{O}\left(x^6\right)$$

y si no queremos que aparezca el término de orden:

```
_.removeO()
```

$$\frac{x^5}{5} - \frac{x^4}{4} + \frac{x^3}{3} - \frac{x^2}{2} + x$$

Por supuestp, es posible cambiar el punto respecto del que realizar el desarrollo:

```
exp(-x).series(x,1,4).removeO()
```

$$-\frac{(x-1)^3}{6e} + \frac{(x-1)^2}{2e} - \frac{1}{e}(x-1) + e^{-1}$$

7.6 GRÁFICOS CON SYMPY

Aunque ya conocemos las capacidades gráficas del módulo Matplotllib con el que podemos representar todo tipo de funciones, el módulo SymPy nos da también la posibilidad de representar funciones de manera más rápida y sencilla. Por ejemplo, si queremos representar la función $f(x) = \log(x)$ con Matplotlib, habrá que ser cuidadosos a la hora de elegir el *array* de abcisas, pues la función no está definida para valores menores o iguales que cero. Con SymPy no es necesario prestar atención a estos detalles. Bastará escribir

```
x = symbols('x')
plot(log(x))
```

El resultado podemos verlo en la figura 7.1.

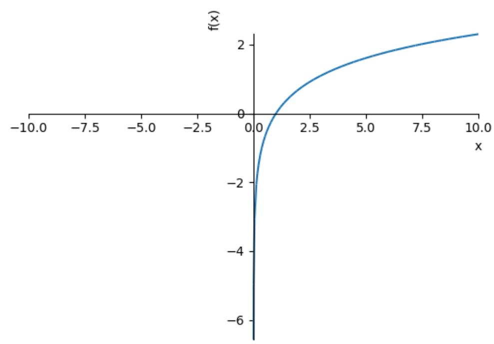
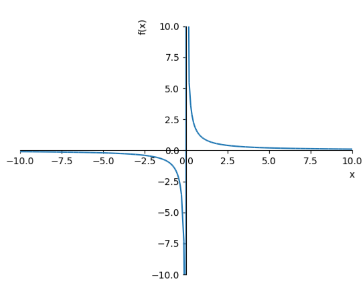
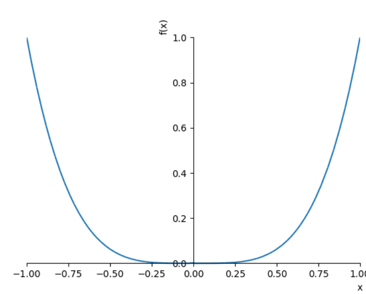


Figura 7.1: Gráfico de la función $f(x) = \log(x)$

A veces será necesario especificar el intervalo en el que queremos dibujar (figura 7.2a), y si la función se va a infinito, se puede mejorar la salida limitando el recorrido (véase la figura 7.2b).



(a) `plot(1/x, ylim=(-10, 10))`



(b) `plot(x**4, (x,-1,1))`

Figura 7.2: Ejemplos del comando `plot`

Aunque es posible configurar algunos parámetros del gráfico, con SymPy no disponemos de toda la potencia de Matplotlib, por lo que su uso se restringe a esbozar rápidamente el gráfico de una función. No obstante, podemos representar varias funciones a la vez y cambiar su color:

```
p1 = plot(x*x,(x,-1,1),show=False)
p1[0].line_color = (1,0,0)
p2 = plot(x,(x,0,1),show=False)
p2[0].line_color = 'blue'
p2.append(p1[0])
p2.show()
```

El resultado puede verse en la figura 7.3. Obsérvese el uso de `show=False` para no mostrar inicialmente el gráfico, y cómo le asignamos un color: en este caso es necesario hacerlo sobre el objeto `p1[0]` y no sobre `p1`, pudiéndose dar el color en formato RGB o mediante el nombre. Usamos también el método `append` para juntar ambos gráficos que luego mostramos con el método `show`.

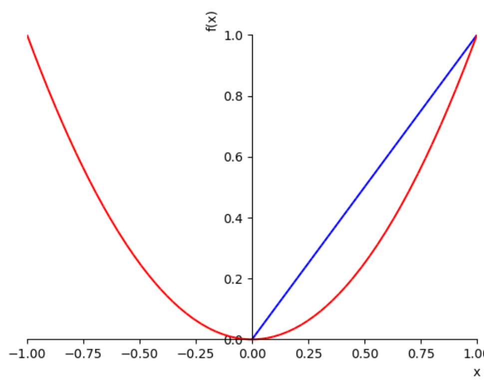


Figura 7.3: Representación de varias funciones

En SymPy encontramos funciones adicionales que permiten representar más tipos de gráficos, como veremos a continuación.

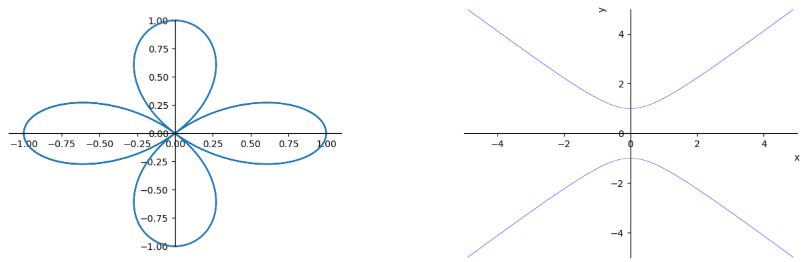
Con `plot_parametric` es posible dibujar curvas paramétricas (véase la figura 7.4a):

```
from sympy.plotting import plot_parametric
plot_parametric((cos(2*x)*cos(x),cos(2*x)*sin(x)))
```

Mucho más interesante es la función `plot_implicit` con la que se obtiene la curva resultante de una ecuación en el plano (figura 7.4b):

```
x, y = symbols('x y')
plot_implicit(Eq(y**2-x**2,1))
```

Por otro lado, con `plot3d` se pueden dibujar superficies en 3D (véase la figura 7.5):



(a) Curva $(\cos(2x)\cos(x), \cos(2x)\,\mathrm{sen}(x))$ (b) Ecuación $y^2 - x^2 = 1$

Figura 7.4: Otros gráficos

```
from sympy.plotting import plot3d
plot3d(sin(x*y),(x,-pi/2,pi/2),(y,-pi/2,pi/2))
```

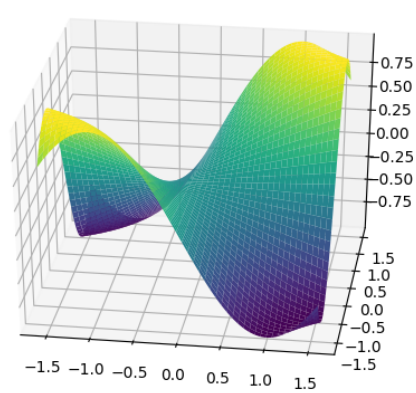


Figura 7.5: Grafo de la función $f(x, y) = \mathrm{sen}(xy)$ en $[-\frac{\pi}{2}, \frac{\pi}{2}]^2$

Para curvas paramétricas en el espacio está `plot3d_parametric_line` (figura 7.6)

```
from sympy.plotting import plot3d_parametric_line
plot3d_parametric_line(x*cos(4*x),x*sin(4*x),x,(x,-2*pi,2*
    pi))
```

y para superfices paramétricas `plot3d_parametric_surface` (figura 7.7)

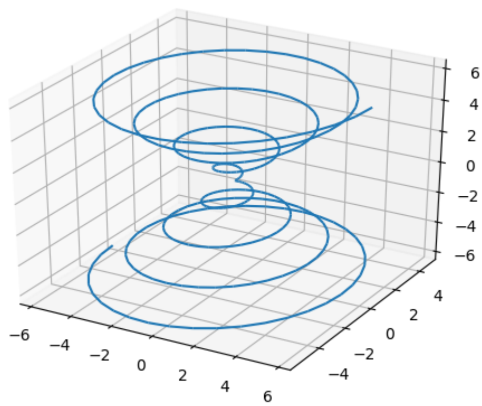


Figura 7.6: Curva $(x\cos(4x), x\,\mathrm{sen}(4x), x)$ para $x \in (-2\pi, 2\pi)$

```
from sympy.plotting import plot3d_parametric_surface
plot3d_parametric_surface((2-cos(x))*cos(y),(2-cos(x))*sin
    (y),sin(x),(x,0,2*pi),(y,0,2*pi))
```

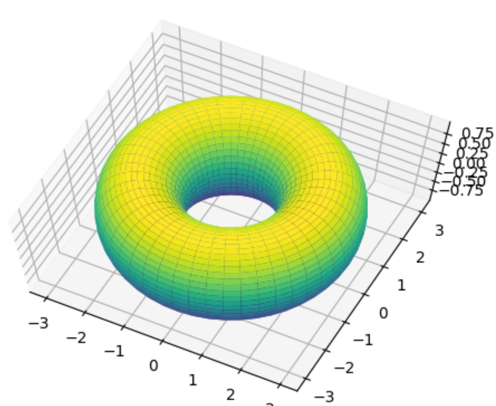


Figura 7.7: Superficie $((2-\cos x)\cos y, (2-\cos x), \cos y, \sin y)\;\; x, y \in (0, 2\pi)$

7 7

EJERCICIOS

E7.1 Averiguar si son ciertas las siguientes identidades:

$$\frac{x^3 - y^3}{x - y} = x^2 + xy + y^2 \qquad x - y = (\sqrt{x} - \sqrt{y})(\sqrt{x} + \sqrt{y})$$

E7.2 Calcular los siguientes límites de sucesiones:

$$\lim_{n\to\infty} \left(n^5 + n^3 + 1\right)^{\frac{1}{n}} \qquad \lim_{n\to\infty} \left(\frac{n - \sqrt{n}}{n + \sqrt{n}}\right)^{\frac{1}{\sqrt{n+1} - \sqrt{n}}}$$

E7.3 Calcular los siguiente límites de funciones:

$$\lim_{x\to 0^-} \left(\frac{2x + |x|}{4x - |x|}\right) \qquad \lim_{x\to 0} \left(\frac{\sqrt{x+1} - 1}{\sqrt[3]{x+1} - 1}\right)$$

E7.4 Comprobar que la derivada de $f(x) = \arctan\left(\frac{\operatorname{sen} x}{1+\cos x}\right)$ es una constante y averiguar su valor.

E7.5 Dada la función $f(x) = (1 + x)^\alpha$, comprobar que

$$f^{n)}(0) = \alpha(\alpha - 1)(\alpha - 2)\cdots(\alpha - n + 1)$$

para valores de $n = 1, \ldots, 10$.

E7.6 Encontrar la solución de las siguientes ecuaciones

$$2\cos(x) + \operatorname{sen}(2 * x) = 0 \qquad f'(x) = 0 \text{ para } f(x) = \sqrt{(x + 3)(x^2 + 1)}$$

E7.7 Realizar la descomposición en fracciones irreducibles de

$$\frac{x^2 - 5x + 9}{x^2 - 5x + 6}$$

e integrar término a término. Luego comprobar que el resultado coincide con la integral de la expresión completa.

8 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 8.1. 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[1] 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 8.2 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[1,:]` y `coord[3,:]`).

[1] También se podría hacer en el espacio sin dificultad.

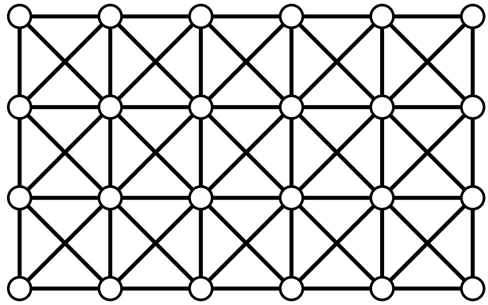


Figura 8.1: Estructura de barras

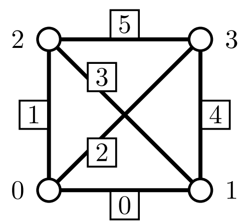


Figura 8.2: Estructura básica

Supongamos ahora que queremos crear una estructura de barras con una configuración regular como la de la figura 8.1. 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.

8.1 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:
    """
    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). 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.[2] 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,

```
p.x
```

```
2.0
```

```
p.y
```

[2]Esto es un convenio universalmente aceptado pero podría usarse cualquier otro nombre.

```
3.0
```

o incluso modificarlos:

```
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)
```

El método `__str__` es uno de los llamados *métodos especiales*, que están asociados al comportamiento de ciertas funciones en Python. En concreto, el método `__str__` es invocado cuando usamos la función `print`. Existen otros métodos de este tipo útiles cuando manejamos otras funciones u operadores (véase el ejercicio E8.6).

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:
    """
    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
```

```python
print(a.p)
```

```
(1.0,3.0)
```

```python
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`:

```python
class Node(Point):
    """
    clase que hereda de la clase Point
    """
    numberNode = 0
    def __init__(self,xx,yy):
        super().__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 padre 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

```python
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:
    """
    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 8.1. 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:
    """
    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 8.2, 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.

8.1.1 Añadiendo métodos

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.

8.2 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:

$$\mathbf{x}^{(k+1)} = M\mathbf{x}^{(k)} + \mathbf{c}, \quad \text{con } \mathbf{x}^{(0)} \text{ dado,} \tag{8.1}$$

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 $\mathbf{b}$, y que devuelva la solución del método iterativo escogido. La matriz M y el vector $\mathbf{c}$ 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 ε y las matrices que definen el método, y se realiza la iteración descrita en (8.1) hasta obtener que

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon \|\mathbf{x}^{(k)}\|.$$

Finalmente 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:
    def __init__(self,A,b,metodo=jacobi):
        self.A = A
        self.b = b
        self.metodo = metodo
        self.eps = 1.e-8
    def iteracion(self):
        self.k = 0
        M,c = self.metodo(self.A,self.b)
        x0 = np.zeros(self.A.shape[0])
        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 ε 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:[3]

```
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. En

[3]Nótese el uso del atributo `__name__` para una función, que nos devuelve su nombre.

conclusión, las clases son mucho más fáciles de mantener y actualizar.

8 3 EJERCICIOS

E8.1 Redefine el constructor de la clase `Bar` de la sección 8.1 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)
```

E8.2 Añade un nuevo método `long` a la clase `Bar` de manera que `barra.long()` devuelva la longitud de `barra`.

E8.3 Para la clase `Truss` de la sección 8.1, escribir un método para que el comando `print` proporcione información sobre el número de nodos y el número de barras de la estructura.

E8.4 Usando el método `long` definido en el ejercicio E8.2, añadir un método a la clase `Truss` que proporcione la longitud total de todas las barras de la estructura.

E8.5 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()
```

```
hola
```
← entrada por teclado

```
a.printString()
```

```
H O L A
```

E8.6 Definir una clase para trabajar con números racionales, de manera que los objetos se creen del siguiente modo:

```
a = Rational(10,3)
```

Implementa los siguientes métodos

- `reduce`: para que el objeto creado sea convertido a una fracción irreducible. Habrá que calcular el máximo común divisor de ambos números (véase ejercicio E2.10 del Capítulo 2) y dividir entre él. Este método se debe llamar en el constructor para modificar el objeto introducido inicialmente.

- `__str__` para que imprima el número racional $\frac{4}{3}$ como `4/3`, o bien $\frac{4}{2}$ como `2`
- `__add__` y `__mul__` para que realice la suma y el producto entre dos números racionales mediante los operadores `+` y `*`, respectivamente.

Los resultados deben mostrar:

```
a = Rational(6,4)
b = Rational(8,6)
print(a)
print(b)
```

```
3/2
4/3
```

```
print(a+b)
print(a*b)
```

```
17/6
2
```

Índice alfabético