Curso de Python Cientifico

Versión 1.0

Jorge A. Perez Prieto

12 de mayo de 2017

Índice general

1. Introducción a Python 3
2. Módulos, paquetes y la librería estándar de Python 25
3. Estilo de codificación y buenas prácticas 37
4. Tratamiento de errores. Sentencia try-except 41
5. Programación orientada a objetos con Python 47
6. Cálculo numérico con Numpy 51
7. Representación gráfica de funciones y datos 65
8. La librería científica Scipy 89
9. Tratamiento de datos con Pandas 101
10. Cálculo simbólico con Sympy 109
11. Envío emails con Python 117
12. Lectura y escritura de hojas Excel 119
13. APÉNDICE B: Instalación de Python y otras herramientas 123
14. APÉNDICE A: Documentación y referencia 125

Curso de introducción a Python 3 con aplicaciones científicas.

Fecha 15 a 19 de mayo de 2017 de 9h a 13h

Lugar IGN, Santa Cruz de Tenerife

Edición 1.0

Fecha 12 de mayo de 2017

Documentos PDF - ePUB - datos.zip

CAPÍTULO 1

Introducción a Python

Python es un lenguaje de programación **interpretado**, como lo son Matlab o IDL, pero a diferencia de ellos, es de uso general y no específicamente orientado a ciencia o ingeniería. A diferencia de lenguajes interpretados como C/C++ o Fortran, que hay que compilar para crear un programa ejecutable, Python se ejecuta (interpreta) línea a línea, lo que lo hace ideal para en análisis interactivo de datos.

Además de incluir una extensa librería propia, existen módulos de terceros para prácticamente cualquier cosa. Python incluye una terminal estándar sobre la que se pueden ejecutar comandos, pero existen otras alternativas, como **ipython**, mucho más completas. Además Python se puede ejecutar como un programa ejecutable desde la línea comandos.

Nota: Python 2 vs Python 3

Desde 2008 coexisten dos ramas de Python, Python 2 y Python 3. Aunque son muy similares, existen alguna diferencias entre ellas por lo que no son completamente compatibles. La versión 2.7 será la última de Python 2, por lo que la recomendación habitual es usar Python 3.

Si embargo, aun hay muchos programas y sistemas que usan Python 2 y es aún la versión instalada por defecto en Linux en MacOS, por lo que conviene conocer las diferencias entre ambos.

En general usaremos Python 3, pero hacemos referencias a Python 2 alguna vez para recordar esta diferencias.

Empezaremos con la terminal estándar de Python escribiendo python en la consola o con el Navigator de Anaconda:

```
Python 3.4.3 (default, Apr  7 2015, 11:18:25)
[GCC 4.9.2 20150212 (Red Hat 4.9.2-6)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Esto es Python 3")
Esto es Python 3
>>>
```

Los tres símbolos de mayor (>>>) es el *prompt* que espera los comandos. Con la consola podemos hacer las operaciones matemáticas básicas directamente:

```
>>> 23*119
2737
>>> 34 - 57 * (13 + 3)**2
>>> -14558
```

Sin embargo, ahora ya no estamos en la terminal de comandos del sistema, por lo no podemos interactuar con el directamente y los comandos del SO no funcionan:

```
>>> pwd
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pwd' is not defined
```

Desde luego existen formas de llamar a commandos y programas del sistema como ya veremos, pero esto nos muestra algunas limitaciones de la consolar estándar. Por este y otros motivos nos conviene usar una consola de comandos avanzada como es `IPython`. Si tenemos una instalación estándar de Python e IPython, podemos usar lanzar `IPython` desde la línea de comandos con `ipython`; si usamos Anaconda podemos usar la Jupyter Qtconsole, que es una versión gráfica mejorada de `IPython`.

```
[japp@vega ~]$ ipython3
Python 3.4.3 (default, Apr  7 2015, 11:18:25)
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]: ls
Documentos/  imagen.png  texto.txt

In [2]: pwd
/home/japp
```

Como vemos, al menos los comandos básicos del sistema funcionan directamente. Lo más destacado de `IPython` son los **comandos mágicos**, que son funcionalidades adicionales muy útiles; todas empiezan con "**%**" y podemos ver una refencia rápida de estos con `%magic` y una referencia general con `%quickref`. Pronto usarmos más comandos mágicos, por ahora empezaremos con `%logstart` para guardar un registro de nuestra sesión de trabajo.

```
In [4]: %logstart -o introduccion_a_python-16May2017.py
Activating auto-logging. Current session state plus future input saved.
Filename       : introduccion_a_python-16May2017.py
Mode           : backup
Output logging : True
Raw input log  : False
Timestamping   : False
State          : active

In [5]:
```

donde el parámetro opcional `-o` guarda también la salida (respuesta) de python en el fichero `introduccion_a_python-16May2017.py`. De esta manera tendremos en un fichero todo lo que hicimos durante la sesión. Con los comandos `%logoff` y `%logon` podemos, respectivamente, detener

temporalmente y continuar el registro de sesión que hayamos iniciado previamente con `%logstart`. El nombre del fichero es libre y tambien puede tener cualquier extesión. En este ejemplo usamos `.py` porque en el fondo contiene código python, pero en realidad no es exactamente un ejecutable porque contiene todos los errores que hayamos podido cometer en el camino.

Ahora por fin podemos seguir con los tipos de datos en Python. Como casi todos los lenguajes de programación, Python distingue entre enteros (`int`), coma flotante (`float`) y cadenas de texto (`str`) aunque posee otros muchos tipos de datos. Como es habitual también en muchos lenguages, en **Python 2** las operaciones entre enteros siempre devuelven un entero, por ejemplo:

```
# Usando Python 2
113/27
4

113/27.0
4.1851851851851851
```

Esto ocurre porque el tipo de dato de salida es siempre el de mayor precisión entre los valores de entrada; si solo se usan enteros, el **tipo de dato** devuelto será un entero aunque su valor numérico no lo sea. Esta situación cambia en Python 3, que cambia a `float` en divisiones, pero hay que tenerlo muy en cuenta con Python 2.

```
# Usando Python 3
113/27
4.185185185185185
```

La variables pueden llamarse usando cualquier cadena alfanumérica, pero sin caracteres especiales como espacios, o `&, $, *,` etc., siempre que no empiece con un número. Además distingue entre mayúsculas y minúsculas y no es necesario declararlas previamente.

```
In [10]: frase = "Esta es una linea de texto"
In [11]: num = 22
In [12]: num*2
Out[12]: 44
In [13]: frase*2
Esta es una linea de textoEsta es una linea de texto
```

Aquí, `frase` es una variable tipo `str` (string) mientras que `num` es otra variable numérica entera `int`. Nótese que mientras se multiplicó num por 2 de forma conocida, la cadena de texto frase fue duplicada al multiplicarla por 2. En este caso hemos operado con dos tipos de datos distintos, un `string` y un `int`, algo que muchos lenguajes de programación produce un error por no tener sentido, sin embargo, Python interpreta el producto de un string y un int como la unión o concatenación de la misma cadena de texto varias veces, y es por eso que vemos frase duplicada. Los tipos de dato básico de Python son `int`, `float`, y `complex`. En Python 2 existe el entero `long`, pero en Python solo hay enteros tipos `int` que básicamente se comporta como `long` en Python 2.

A lo largo del trabajo podemos acabar definiendo muchas variables de distinto tipo. Con el comando `type()` podemos saber el tipo de dato que se asigna a una variable si en cualquier momento no recordamos como la definimos:

```
In [15]: type(frase)
Out[15]: str
In [16]: type(num)
Out[16]: int
```

```
In [17]: complejo = 1.2 + 5.0j    # complex(1.2, 5.0)
In [18]: type(complejo)
Out[18]: complex
```

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

```
In [20]: float(3)
3.0

In [21]: int(3.1416)
3

In [22]: str(34)
'34'
```

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

```
In [30]: print(round(4.4)) , (round(4.5))
Out[30]: 4.0 5.0
```

Además de los operadores ariméticos conocidos `**` (exponenciación, `^` es `XOR` en operaciones bit a bit), `*, /, %, +\` y `-`, tenemos los operadores lógicos:

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

Como resultado de una operación lógica, obtenemos como respuesta un elemento, `True` o `False`, según se verifique o no la operación. Estos elementos lógicos los podemos usar a su vez para otras operaciones. Veamos algunos ejemplos:

```
In  [40]: resultado = 8 > 5
In  [41]: print(resultado)
True

In  [42]: resultado = (4 > 8) or (3 > 2)
In  [43]: print(resultado)
True

In  [44]: resultado = True and False
In  [45]: print(resultado)
False

In  [46]: resultado = (4 > 8) and (3 > 2)
In  [47]: print(resultado)
False
```

Usando los operadores lógicos, podemos consultar si una variable es de un tipo concreto:

```
In  [50]: numero = 10.0
In  [51]: type(numero) == int
Out [51]: False
In  [52]: type(numero) == float
Out [52]: True
```

En este caso `int` y `float` no son cadenas de texto, sino un indicador del tipo de dato. Esto se puede usar para cualquier tipo de dato más complejos, no únicamente números o cadenas, los cuales veremos más adelante.

1.1 Cadenas de texto

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

```
In [52]: # Variable "frase" que contiene una cadena de texto
In [53]: frase = "Si he logrado ver más lejos, ha sido porque he subido a␣
 ↪hombros de gigantes"
In [54]: print frase[0]        # Primera letra de la cadena
S

In [55]: print(frase[10])      # Decimoprimera letra, con índice 10
a
In [56]: print(len(frase))     # Longitud de la cadena
76

In [57]: print(frase[18:29])   # Seccion de la cadena
más lejos

In [58]: print(frase[68:])     # Desde el indice 68 hasta el final
gigantes

In [58]: print(frase[:10])     # Desde el principio al caracter de
Si he logr                     # indice 10, sin incluirlo
```

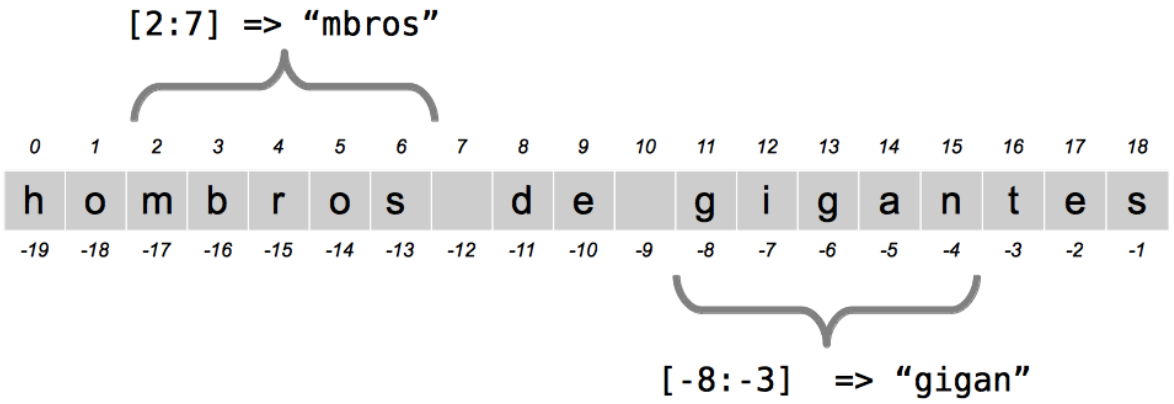


Figura 1.1: Indexado de cadenas de texto usando índices positivos empezando desde la izquierda con 0 y negativos, empezando desde la derecha con -1.

El comando len() nos da el número de caracteres (longitud) de la cadena de texto, incluyendo espacios en blanco y caracteres especiales:

```
In  [60]: len(frase)          # 75 es el número de caracteres de la cadena
Out [60]: 75
In  [61]: print(frase[len(frase)-1]) # El último carácter, contando desde␣
 ↪la izquierda
s
In  [62]: print( frase[-1] == frase[len(frase)-1] ) # Compruebo si son␣
 ↪iguales
True
```

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

```
In  [63]: print(frase[-1])    # El último carácter, contando desde la␣
 ↪derecha
s
```

Si hacemos `dir()` de la variable frase, veremos los métodos (funciones) que se pueden aplicar `frase` por ser un `string`:

```
In  [65]: type(frase)
Out [65]: str

In  [66]: dir(frase)
Out [66]: ['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__doc__',
 .
 .
  'upper',
 'zfill']
```

los primeros, que empiezan y terminan con "__" son *variables internas* del módulo, un tipo de métodos y propiedades especiales; el resto son los métodos normales. Una manera más práctica de verlos, gracias a IPython es escribir un punto después de `frase` (u otro tipo de variable) y luego presionar el tabulador:

```
In [10]: frase.<tab>
frase.capitalize  frase.isalnum     frase.lstrip      frase.splitlines
frase.center      frase.isalpha     frase.partition   frase.startswith
frase.count       frase.isdigit     frase.replace     frase.strip
frase.decode      frase.islower     frase.rfind       frase.swapcase
frase.encode      frase.isspace     frase.rindex      frase.title
frase.endswith    frase.istitle     frase.rjust       frase.translate
frase.expandtabs  frase.isupper     frase.rpartition  frase.upper
frase.find        frase.join        frase.rsplit      frase.zfill
frase.format      frase.ljust       frase.rstrip
frase.index       frase.lower       frase.split
```

Es así además como se aplican los método en Python, con la sintaxis `variable.metodo`. Aquí hay algunos ejemplos:

```
In  [70]: frase_mayusculas = frase.upper()     # Cambia a mayusculas y lo␣
↪guardo en
In  [71]: print(frase_mayusculas)              # la variable frase_
↪mayusculas
SI HE LOGRADO VER MAS LEJOS, HA SIDO PORQUE HE SUBIDO A HOMBROS DE GIGANTES

In  [72]: frase_minusculas = frase.lower()     # Cambia a minúculas y lo␣
↪guardo en
                                          # la variable frase_minusculas
In  [73]: print(frase_mayusculas)
si he logrado ver más lejos, ha sido porque he subido a hombros de gigantes

In  [74]: # Reemplaza una cadena de texto por otra
In  [75]: frase.replace("hombros", "la chepa")
    'Si he logrado ver mas lejos, ha sido porque he subido a la chepa de␣
↪gigantes'
```

Este último comando devuelve una nueva cadena de texto cambiada, que vemos por pantalla, sin cambiar la original. Podemos comprobar que la frase no se ha alterado de forma permanente haciendo `print(frase)`.

Para cambiar la variable frase deberemos volver a definir la cadena de texto:

```
In  [76]: # Reemplaza definitivamente una cadena de texto por otra
In  [77]: frase = frase.replace("hombros", "la chepa")
In  [78]: print(frase)
'Si he logrado ver mas lejos, ha sido porque he subido a la chepa de␣
↪gigantes'
```

1.2 Formato de texto

A menudo queremos imprimir texto con valores numéricos de algún tipo, pero esto no es posible porque string y números son tipos de datos distintos. La forma más básica de mezcar cadenas y números es convirtiendo los números a cadenas y concatenándolas:

```
In  [80]: a, b = 10, 10**2   # Definimos dos numeros, a=10 y b=10**2
In  [81]: print(str(a) + " elevado al cuadrado es " + str(b))
10 elevado al cuadrado es 100
```

Si estamos usando Python 3, la impresión es algo más flexible, pudiendo un número indefinido de parámetros de distinto tipo separados por comas y `print()` los une con un espacio o con la cadena que le indiquemos en el parámetro `sep`, que es opcional:

In [80]: print(a, “elevado al cuadrado es”, b) 10 elevado al cuadrado es 100

In [81]: print(a, ” elevado al cuadrado es”, b, sep=”;”) 10;elevado al cuadrado es;100

Si embargo, la manera más práctica y correcta de hacer esto es imprimiendo los números con el formato que queramos con la sintaxis de formato que Python hereda de C:

```
In [82]: # Calculamos el logaritmo base 10 de 2 e imprimimos
          # el resultado con 50 decimales
In [83]: print("%.50f" % log10(2.0**100))
30.10299956639811824743446777574717998504638671875000
In [84]: # Otro ejemplo usando texto, enteros y decimales
```

```
In [85]: print("El %s de %d es %f." % ('cubo', 10, 10.**3) )
El cubo de 10 es 1000.000000.
```

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

```
In [86]: print("%.5e" % 0.0003567)
3.56700e-04

In [87]: # Otra forma de hacerlo (sin imprimir)
In [88]: resultado = format(0.0003567, ".5e")
In [89]: resultado
Out[89]: '3.56700e-04'  # es un string
```

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

```
In [90]: # resultado de un cálculo obtenido con las cifras
In [91]: # decimales que proporciona el ordenador
In [92]: resultado = 3.1415785439847501
In [93]: # este es su error con igual número de cifras decimales
In [94]: error = 0.0001345610900435
In [95]: # así expresamos de forma correcta el resultado
In [96]: print("El resultado del experimento es %.4f +/- %.4f" ) %␣
↪(resultado, error)
El resultado del experimento es 3.1416 +/- 0.0001
```

Además de este sistema, Python tiene un (casi) lenguaje para dar formato a las cadenas usando el método `format()`. Es sistema es menos conocido pero mucho más flexible y ponente:

```
In [96]: print("{:.2f}".format(3.1415926))
3.14
```

Número	Formato	Salida	Descripción
3.1415926	{:.2f}	3.142	decimal places
3.1415926	{:+.2f}	3.142	decimal places with sign
-1	{:+.2f}	-12	decimal places with sign
2.71828	{:.0f}	3	No decimal places
5	{:0>2d}	5	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1000000	Number format with comma separator
0.25	{:.2 %}	25.00 %	Format percentage
1000000000	{:.2e}	1000000000	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)

Aquí hay algunos ejemplos de cómo se usan:

```
frase2 = "A hombros de {}".format("gigantes")
frase3 = " {0} es mejor que {1} ".format("Python", "IDL")
resultado = "El {operacion} de {numero} es {resultado}".format(operacion=
↪"cubo", numero=7, resultado=7**3)

# Devuelve:
A hombros de gigantes
Python es mejor que IDL
El cubo de 7 es 343
```

Esta especie de lenguaje tiene muchas más opciones, lo mejor es consultar la documentaciónoficial o alguna guia.

1.3 Estructuras de datos

Además de los tipos de datos convencionales, Python viene de fábrica con otro tipo de datos más complejos, las **estructuras de datos**. que permiten organizar y manipular varios elementos. El más común es la lista (`list`), que se crea con corchetes con los elementos separados por comas:

```
estrellas = ["Alhena", "Mizar", "Cor Caroli", "Nunki", "Sadr"]
datos = ["Beta pictoris", 1.6, [1, 2, -3]]
```

Con el método `split()` podemos separar un `string` en una lista de elementos, separando por defecto por espacios; se puede añadir un segundo parámetro opcional para separar por otro(s) caracter.

```
In [100]: palabras = frase.split()
In [101]: print(palabras)
['hombros', 'de', 'gigantes']
```

La listas se indexan prácticamente igual que las cadenas, donde cada elemento tiene un índice:

```
In [102]: estrellas[-1]   # El último elemento
Out[102]: 'Sadr'

In [103]: estrellas[1:3]  # Otra lista, del segundo (índice 1) al tercero␣
↪(indice 2)
Out[103]: ['Mizar', 'Cor Caroli']
```

En el último ejemplo es importante darse cuenta que `estrellas[1:3]` **no incluye el último elemento**, de manera devuelve dos elementos; esto es fácil de prever restando el último del primero: `3-1 = 2` elementos.

Con el método `range()` tenemos un iterador de números enteros. Un iterador es un tipo de dato que posee métodos para iterar los elementos que contiene, que son similares a los de una lista. En realidad en Python 2 la función `range()` no devuelve un iterador, sino directamente una lista de enteros. En Python 3 podemos convertir el iterador `range` con `list(range)`.

Admite hasta tres parámetros (ver `help(range)` o `range?`), aunque sólo uno es obligatorio, que es el número de elementos, que irá de `0` a ese número, sin incluirlo:

```
In [104]: # Iterador de enteros de 0 a 4
In [105]: list(range(5))
Out[105]: range(0, 5)
```

```
In [106]: list(range(5))  # convierto el iterador a lista (no necesario en␣
↪Python 2)
Out[106]: [0, 1, 2, 3, 4]

In [107]: # Lista de enteros de 10 a 15
In [108]: list(range(10, 16))
Out[108]: [10, 11, 12, 13, 14, 15]

In [108]: # Lista de enteros de 10 a 20, dos en dos
In [109]: list(range(10, 21, 2))
Out[109]: [10, 12, 14, 16, 18, 20]
```

Las listas tienen métodos similares a las cadenas de texto y también varios métodos para manipularlas y transformalas:

```
In [110]: len(estrellas)
Out[110]: 5

In [111]: estrellas.
estrellas.append   estrellas.index    estrellas.remove
estrellas.count    estrellas.insert   estrellas.reverse
estrellas.extend   estrellas.pop      estrellas.sort

In [112]: estrellas.append("Ras Algethi")  # Añado Ras Algethi al final de␣
↪la lista

In [113]: print(estrellas)
['Alhena', 'Mizar', 'Cor Caroli', 'Nunki', 'Sadr', 'Ras Algethi']

In [114]: estrellas.insert(3, "Hamal")     # Añado Hamal en el cuarto␣
↪lugar (índice 3)

In [115]: print(estrellas)
['Alhena', 'Mizar', 'Cor Caroli', 'Hamal', 'Nunki', 'Sadr', 'Ras Algethi']

In [116]: estrellas.pop()                  # Estraigo el último elemento␣
↪(lo devuelve)
Out[116]: 'Ras Algethi'

In [117]: estrellas.pop(3)                 # Estrae el elemento de índice 3
Out[117]: 'Hamal'

In [118]: estrellas.remove("Nunki")        # Elimina la primera ocurrencia␣
↪de Hamal

In [119]: estrellas.sort()                 # Ordena la lista alfabéticamente

In [120]: estrellas
Out[120]: ['Alhena', 'Cor Caroli', 'Mizar', 'Sadr']
```

Aunque podemos manipular mucho las listas, existen funciones como `map()` o `filter()` que nos permite iterar con sus elementos; además el módulo itertools ofrece métodos adicionales para trabajar con listas y otros iterables. Veremos algunos ejemplos más adelante.

Otro tipo de dato estructurado son las **tuplas**, que básicamente son lista inalterables. Tienen propiedades de indexado similares, pero no poseen métodos para alterarlos porque no se pueden cambiar. Se declaran entre paréntesis en lugar de corchetes y se suelen usar en parámetros de funciones y datos que no suelen

modificar. Cuando definimos una serie de datos de cualquier tipo, separados por comas, se considera una tupla, aunque no esté entre paréntesis.

```
In [122]: c = (1, 3)              # Defino una tupla
In [123]: print(c)
(1, 3)
In [124] parametros = 4, 5, 2, -1, 0  # Creo otra tupla al indicar varios␣
↪datos separados
                                  # por comas, aunque no estén entre␣
↪paréntesis
In [125] type(parametros)
<type 'tuple'>
```

Los **diccionarios** son estructuras que en lugar de índices numéricos tienen índices de cadenas declarados por nosotros, lo que es muy cómodo para describir propiedades.

```
In [128] star = {'name': 'Hamal', 'mag': 2.00, 'SpT': 'K2III', 'dist': 66}
In [129] type(star)
Out[129] <type 'dict'>
```

En este caso hemos creado una clave "name" con valor "Hamal", otra clave "mag" con valor 2.00, etc. Al crear los datos con esta estructura, podemos acceder a los valores de las claves fácilmente, así como definir nuevas parejas clave-valor:

```
In [130]: print( star['name'] )
Hamal

In [131]: star['paralax'] =  49.56     # Añado un nuevo dato
```

Hay que notar que ya no podemos acceder a los datos por su índice, ya que los diccionarios se indexan exclusivamente por la clave y obtenendremos un error si lo hacemos. Además, el orden de los elementos no será necesariamente el con el que lo hemos creado, porque éste se pieder al no tener índices numéricos:

```
In [134]: print(star[0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0

In [135]: print(star)  # demonios, no estan en el orden original!
{'dist': 66, 'SpT': 'K2III', 'name': 'Hamal', 'mag': 2.0}
```

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

```
In [135]: star.keys()
Out[135]:['dist', 'SpT', 'name', 'mag']

In [136]: star.values()
Out[136]: [66, 'K2III', 'Hamal', 2.0]
```

Finalmente, también existen los `set` que se podrían llamar también *conjuntos*, que son colecciones sin ordenar de elementos no duplicados. Son útiles probar pertenencias a listas eliminar entradas duplicadas, además de tener operaciones matemáticas como unión, intersección, etc.

```
In [140]: estrellas1 = set( ("Alhena", "Mizar", "Cor Caroli") )
In [141]: estrellas2 = set( ("Mizar", "Cor Caroli", "Nunki", "Sadr") )

In [141]: "Alhena" in estrellas1
Out[141]: True

In [142]: "Alhena" in estrellas2
Out[142]: False

In [143]: # Union
In [144]: estrellas1 | estrellas2
Out[144]: {'Alhena', 'Cor Caroli', 'Mizar', 'Nunki', 'Sadr'}

In [145]: # Intersección
In [146]: estrellas1 & estrellas2
Out[147]: {'Cor Caroli', 'Mizar'}

In [148]: # Diferencia
In [149]: estrellas1 - estrellas2
Out[149]: {'Alhena'}

In [150]: # Diferencia simétrica (que son únicos en cada grupo)
In [150]: estrellas1 ^ estrellas2
Out[151]: {'Alhena', 'Nunki', 'Sadr'}
```

1.4 Usando funciones externas: módulos

Si queremos usar funciones matemáticas debemos importar el módulo `math` para acceder a ellas:

```
In [160]: import math
In [161]: print math.sin(0.5*math.pi)
1.0
In [162]: 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']

In [164]: a, b = 4.4, 4.5
In [165]: print(math.ceil(a)), (math.ceil(b))
Out[165]: 5.0 5.0

In [166]: print(floor(a)), (math.floor(b))
Out[166]: 4.0 4.0
```

Arriba usamos `dir()` para ver el contenido del módulo, sus métodos y propiedades disponibles, que no son más que la libreria matemática estándar de C. Además del listado, podemos obtener ayuda de un módulo o método (función) usando el comando `help()`. Más adelante veremos cómo funcionan los módulos y paquetes en Python.

1.5 Programas ejecutables

En lugar de escribir en una consola, vamos a crear un pequeño programa ejecutable con un editor de texto (de código):

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

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

# Numero de entrada
x = 23.0

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

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

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

```
$ python cubo.py
```

y la respuesta del programa será:

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

El cubo de 23.00 es 12167.00
```

En la primera línea del programa hemos escrito **#!/usr/bin/python** para indicar la ruta donde tenemos instalado python. La segunda línea, **#-*- coding: utf-8 -*-** hemos indicado el tipo de codificación UTF-8, para poder poner caracteres especiales como tildes y eñes.

Ya que la primera línea indica en qué directorio está el ejecutable de python, el programa también se puede ejecutar como un comando o programa del sistema, escribiendo únicamente:

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

Si quieres ejecutar el programa desde dentro de la consola estándar de python, puedes usar lo `execfile()`:

```
>>> execfile('cubo.py')
```

de igual manera, si estamos usando IPython podemos usar el comando mágico run, que en fondo es una llamada a `execfile()`:

```
run cubo.py
```

La ventaja en este último caso con IPython es que una vez ejecutado, las variables y funciones definidas en el programa lo estarán ahora lo estarán en la sesión de IPython, lo que es muy útil para probar y corregir nuestro programa interactivamente.

```
print(x)
23.0
```

es decir, vemos ahora tenemos la variable x está definida en nuestra sesión.

Podemos definir funciones reutilizables con el comando `def()`, por ejemplo, para usarlo en nuestro programa anterior:

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

# Utilizamos la funcion para calcular el cubo de 4
resultado = cubo(4.0)
```

Ahora podemos llamar a la función `cubo()` cuando queramos.

¡ALTO!: Aquí ha pasado algo importante. Fíjense en la definición de `cubo(x)` ¿cómo sabe Python donde empieza y termina la función si no hay llaves que abren ni cierran o palabras clave del tipo "DO BEGIN" o "END" como en otros lenguajes? La clave está en el **sangrado** (identación), que es **obligatorio en Python** ya que es como se indican dónde empiezan y terminan los bloques de código. No hay número específico de espacios que haya que poner (pero lo recomendado es cuatro), lo importante es que las líneas estén en el mismo bloque de sangrado. En nuestro ejemplo sencillo, la función empieza después de ":" y termina donde el margen vuelve a ser el original.

También es fundamental no mezclar espacios y tabulación, porque producen errores de sangrado (*indentation error*), por lo que se recomendable configurar el editor de manera que escriba siempre espacios al pulsar el tabulador.

En Python, la variables definidas dentro de una función **son locales**, lo que quiere decir que sólo están definidas dentro de la función y no están accesibles fuera. Podemos comprobar esto con una pequeña modificación del programa:

```
In [168]: def cubo(x):
    ...:     potencia = 3
    ...:     return x**potencia
    ...:

In [169]: cubo(3.5)
Out[169]: 42.875

In [170]: print(potencia)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-7baa8b437297> in <module>()
----> 1 potencia

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

Como vemos, aunque función funciona correctamente, la variable `potencia` que definimos dentro no existe fuera de la función. Esto igualmente cierto para funciones definidas dentro de funciones. Cuando creamos variables y funciones fuera de una función, como hecho hasta ahora, son todas **globales**.

Además de sentencia de `def` que crea funciones ordinarias, existen la **funciones anónimas** o **función lambda** que permiten definir funciones simples en una sola línea:

```
cubo = lambda x: x**3

cubo(3)
```

El comando `def()`, éste permite varios parámetros en entrada, que pueden ser opciones si se incluye un valor por defecto:

```
def saludo(nombre, apellido=""):
    print "Hola {} {}".format(nombre, apellido)

saludo('Carmen')
Hola Carmen
```

Los valores de los parámetros se asignan en el orden en el que estándefinidos en la función, pero si se usan con su nombre, pueden ponerse en cualquier orden. Por ejemplo:

```
saludo(apellido="Prieto", nombre="Gloria")
Hola Gloria Prieto
```

Es posible tener un número indeterminado de parámetros, dados en forma de tupla o diccionario, en cuyo caso en la definición los parámetros deben empezar con * o ** respectivamente:

```
def datos_estrella(*args, **kwargs):
    print "Nombre", args
    print "Datos", kwargs
   .....:

datos_estrella('Altair', mag=2.2, SpT="A7V")
Nombre ('Altair',)
Datos {'SpT': 'A7V', 'mag': 2.2}
```

En este ejemplo vemos que la funcion `datos_estrella()` admite un número indefinido de parámetros sin nombre, que se agruparán en una lista llamada `args` y también un número indefinido de parámetros con nombre (con parejas nombre=valor), que se agruparán en una lista llamada `kwargs` en nuestro ejemplo.

1.6 Uso de funciones con iterables

Podemos usar funciones con tipos de datos basicos como strings o float, pero no podemos aplicarlos directamente a listas o tuplas. Por ejemplo, no podemos hacer `cubo(range(5))` esperando obtener una lista de resultado. Si embargo, podemos usar algunos métodos como `map()` o `filter()` para operar recursivamente con iterables, como mencionamos antes. Veamos este ejemplo:

```
n = range(10)

def raiz(x):
    return x**0.5

# Aplica una función a un iterable
resultado = map(raiz, n)  # Devuelve una lista (python 2) o iterable␣
↪(python 3)

# Devuelve una lista (python 2) o iterable (python 3) para los elementos
```

```
# del iterable que son True
mayores5 = filter((lambda x: x >=5), n)
```

1.7 Entrada de datos en programas

Los programas ejecutables que creamos necesitan a menudo datos de entrada que pueden ser distintos cada vez que se usan, e incluirlos directamente como variables dentro del código puede no ser muy eficiente ya que tendríamos que abrir y modificar el fichero cada vez que cambiamos los párámetros.

Podemos usar la función `input()` (`raw_input()` en Python 2) para pedir entradas por teclado al usuario. Modificamos nuestro pequeño programa para usarlo:

```
# Numero de entrada
x = input('Dame un numero: ')

x = float(x)

resultado = cubo(x)

print("El cubo de {} es {}".format(x, resultado))
```

Sin embargo, `input()` devuelve siempre un *string* aunque se de un valor numérico, por eso hay que transformarlo antes a entero o float para operar con el. Una alternativa a convertir strings en valores numéricos es emplear la función `eval()`, que evalúa un string como si fuese código y devuelve el resultado, incluyendo una lista si el string contiene comas:

```
x = 3.1416
eval("x**3")  # Hace la operación y devuelve un float
31.006494199296

eval("10, 20, 30")  # Si es un string con comas devuelve una tupla
(10, 20, 30)
```

Una manera alternativa de dar parámetros de entrada a un programa es usar argumentos justo después de la llamada del programa, para lo que necesitamos la función `argv` del módulo `sys`. Se usa de esta manera:

```
from sys import argv

# Los parámetros pasados están en una lista de strings en argv, en la que
# el primer elemento es el nombre del programa y el segundo el primer␣
↪parámetro
x = argv[1]

x = float(x)

resultado = cubo(x)
```

Ahora podemos llamar al programa poniendo los parámetros después:

```
# Para calcular el cubo de 6.5
./cubo.py 6.5
```

Se puede añadir un número indefinido de parámetros, pero como vemos debe hacerse sin nombres. Una opción más sofisticada, pero también algo más compleja, es emplear el módulo argparse, que permite definir parámetros de entrada complejos con varias opciones. Consideremos un programa que puede calcular la raíz o el cubo de un número, según decida el usuario. Podemos hacerlo de la siguiente manera:

```
import argparse
from math import sqrt

parser = argparse.ArgumentParser()

# Dos argumentos obligatorios posibles, el numero a calcular
# y la operacion a realizar
parser.add_argument("-n", "--numero", help="Numero a calcular",
                    type=float, required=True)

parser.add_argument("-o", "--oper", help="Operacion a realizar: cubo o raiz
↪",
                    type=str, default='cubo')

args = parser.parse_args()

# Funciones de las operaciones que puede hacer el programa
def cubo(x):
    y = x**3
    return y

def raiz(x):
    return sqrt(x)

# Hacemos una operación u otra según la opcion elegida por el usuario
if args.oper == 'cubo':
    print("El cubo de {0} es {1}".format(args.numero, cubo(args.numero)))
elif args.oper == 'raiz':
    print("La raiz de {0} es {1}".format(args.numero, raiz(args.numero)))
else:
    print("Error, operacion desconocida")
```

Ahora podemos añadir argumentos durante la ejecución del programa y además genera un mensaje de ayuda con los argumentos disponibles usando `--help`.

```
./cubo-argparse.py --help
usage: cubo-argparse.py [-h] [-n NUMERO] [-o OPER]

optional arguments:
  -h, --help            show this help message and exit
  -n NUMERO, --numero NUMERO
                        Numero a calcular
  -o OPER, --oper OPER  Operacion a realizar: cubo o raiz


./cubo-argparse.py -n 10 -o cubo
El cubo de 10.0 es 1000.0

./cubo-argparse.py -n 10 --oper raiz
La raiz de 10.0 es 3.16227766017
```

1.8 Control de flujo

Python tiene los elementos de control de flujo más comunes: **if-then-else**, **for** y **while**, con la peculiaridad que se usan los bloques de identación para delimitarlos. El bucle **for** se suele usar para recorrer iterables (listas, tuplas, etc.), deteniéndose cuando se agota la lista:

```
for nombre in estrellas:
    print(nombre)

Alhena
Cor Caroli
Mizar
Sadr
```

Alternativamente, podemos recorrer una lista de índices y llamar a cada elemento con su índice:

```
for i in range( len(estrellas) ):
    print("%d) %s. " % (i+1, estrellas[i]))

1) Alhena.
2) Cor Caroli.
3) Mizar.
4) Sadr.
```

Si estamos trabajando con un diccionario, no podemos hacer esto directamente porque no tienen índices numéricos, pero podemos emplear el método `iteritems()` para iterar parejas `clave:valor`:

```
for clave, valor in star.iteritems():
    print("{0}: {1}".format(clave, valor))

dist: 66
SpT: K2III
name: Hamal
mag: 2.0
```

En este ejemplo tenemos que usar dos variables mudas (clave y valor) en lugar de una para capturar cada clave y valor del diccionario.

Los bucles **for** nos permiten crear nuevas lístas dinámicamente:

```
# Importo la función log10 del módulo math
from math import log10

b = [2.3, 4.6, 7.5, 10.]

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

print(c)
# Resultado:
# [0.36172783601759284, 0.66275783168157409, 0.87506126339170009, 1.0]
```

¡Momento! Qué pasaría si en la lista de números `b` está el `0`?

```
In [78]: b = [0, 2.3, 4.6, 7.5, 10.]

In [79]: c = [log10(x) for x in b]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-79-bcb6e1180ea3> in <module>()
----> 1 c = [log10(x) for x in b]

ValueError: math domain error
```

Desde luego que da un error ¿qué hacemos? Aquí está la magia de Python:

```
c = [log10(x) for x in b if x > 1]
```

El bucle **while** repite una serie de órdenes mientras una condición sea cierta (vale `True`):

```
cuentas = 0

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

De manera similar poder hacer que `while` funcione mientras **no se cumpla una condición** usando `while not`:

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

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

En el ejemplo anterior hemos hecho una comparación de igualdad `x == 5` usando enteros, pero hay que tener cuidado cuando se comparan, mediante una igualdad exacta, números decimales o de coma flotante floats entre sí. Debido a la precisión finita de los ordenadores, es posible que una determinada igualdad nunca se cumpla exactamente y por lo tanto la ejecución del bucle nunca se detendrá. Podemos comprobar esto con un ejemplo en el que imprimimos los números que van de 0.0 a 1.0 a intervalos de 0.1,:

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

""" Resultado que obtenemos:
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
x = 0.50000000000000000
x = 0.59999999999999998
x = 0.69999999999999996
```

```
x = 0.79999999999999993
x = 0.89999999999999991
x = 0.99999999999999989  <-- El bucle while debió detenerse aquí, pero no
↪lo hizo
x = 1.09999999999999987
x = 1.19999999999999996
x = 1.30000000000000004
  .
  .
  .                      <-- Presionar Ctrl+C para detener el programa
"""
```

y así el bucle no se para nunca. El código anterior produce un **bucle infinito** porque la condición `x == 1.0` nunca se da exactamente (el valor más cercano es 0.99999999999999989 pero no 1.0). La conclusión que podemos extraer de aquí es que es preferible no comparar nunca variables o números de tipo float exactamente.

Una opción para resolver el problema anterior es usar rangos de precisión en el que definimos lo que para nosotros es **suficientemente cercano** al valor deseado. Así, con el ejemplo anterior podríamos hacer:

```
x = 0.0
# Condición de que nuestro numero se acerque a 1.0
# al menos en 1e-8
while abs(x - 1.0) > 1e-8:
    x = x + 0.1
    print("x = %19.17f" % x)

""" Resultado que obtenemos:
x = 0.10000000000000001
x = 0.20000000000000001
x = 0.30000000000000004
x = 0.40000000000000002
x = 0.50000000000000000
x = 0.59999999999999998
x = 0.69999999999999996
x = 0.79999999999999993
x = 0.89999999999999991
x = 0.99999999999999989
"""
```

Finalmente, el **if-then-else** funciona de forma habitual:

```
c = 12

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

Existen algunos elementos que podemos usar en las secuencias de control de flujo. La sentencia **break** permite interrumpir el bloque más cercano en un bucle **while** o **for**. De manera similar, **continue** continúa con la siguiente iteración dentro del bucle más próximo. La sentencia **else en bucles** permite ejecutar una acción cuando el bucle termina una lista (en bucles **for**) o cuando la condición es falsa (con while) pero no si el bucle se interrumpe usando **break**. Veamos el ejemplo de un programa que calcula los

números primos entre 2 y 10:

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

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

En este ejemplo usamos **break** para cortar el bucle for más interno si el **if se cumple** (es `True`) y así evitar que muestre multiplicaciones equivalentes (e.g.: 3*4 = 4*3); podemos comprobar lo que ocurre si no pusiésemos **break**. Es útil para por ejemplo, evitar que un bucle siga corriendo si ya se consiguió la condición.

Algo muy interesante en este ejemplo es que usamos la sentencia **else** con **for**, en lugar del usarla con **if** como es habitual. Un else en **for** se ejecuta **cuando la lista en el for llega al final**, sin cortarse. De esta manera imprimimos un aviso si el bucle termina (el for agota la lista) sin llegar a usar break, lo que indica que ningún número de la lista es múltiplo suyo y por tanto es primo.

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

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

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

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

1.9 Ejercicios

1. La variación de temperatura de un cuerpo a temperatura inicial T_0 en un ambiente a T_s cambia de la siguiente manera:

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

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

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

   ```
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

   T R W A G M Y F P D X B N J Z S Q V H L C K E
   ```

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

3. Obtener un valor de π calculando la suma siguiente para n=200:

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

4. Se llama sucesión de Fibonacci a la colección de *n* números para la que el primer elemento es cero, el segundo 1 y el resto es la suma de los dos anteriores. Por ejemplo, la sucesión para *n=5* es (0, 1, 1, 2, 3). Crear un programa que calcule la lista de números para cualquier *n*.

5. Genera una lista que contenga el cuadrado de los números pares y el cubo de los impares entre 1 y 100 (inclusive). Calcula cuantos números de esa lista debes sumar para que el resultado de la suma sea lo más cercano posible, pero inferior, a un millón.

6. Escribir un programa que proporcione el desglose en el número mínimo de billetes y monedas de una cantidad entera cualquiera de euros dada. Recuerden que los billetes y monedas de uso legal disponibles hasta 1 euro son de: 500, 200, 100, 50, 20, 10, 5, 2 y 1 euros. Para ello deben solicitar al usuario un número entero, debiendo comprobar que así se lo ofrece y desglosar tal cantidad en el número mínimo de billetes y monedas que el programa escribirá finalmente en pantalla.

CAPÍTULO 2

Módulos, paquetes y la librería estándar de Python

Como ya hemos visto, podemos importar nuevos módulos de la librería extándar de Python o de terceros con **import <modulo>**, pero hay varias maneras de hacerlo:

```
In [10]: import math           # importa el módulo math
In [11]: import math as M      # importa el módulo math llamándolo M
In [12]: from math import sin, cos, pi  # importa las funciones sin, cos y␣
↪ pi de math
In [13]: from math import *    # importa todas las funciones de math
```

De manera similar podemos crear un módulo propio que puede usarse como un programa independiente o importase como un módulo y poder reutilizar sus funciones:

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

"""Programa de calculo del cubo de un numero"""

__author__ = "Jorge"
__copyright__ = "Curso de Python"
__credits__ = ["Pepe", "José Luis", "Roberto"]
__license__ = "GPL"
__version__ = "1.0"
__email__ = "japp@denebola.org"
__status__ = "Development"


def cubo(x):
    """Calcula el cubo de un numero"""
    y = x**3
    return y

if __name__ == "__main__":
    x = int( input("Dame un numero: ") )
    y = cubo(x)
    print("El cubo de %.2f es %.2f" % (x, y))
```

Bien, ahora podemos usar este programa como un ejecutable como ya hemos hecho o importarlo como un módulo y usar la función cubo(). La primera de comentario multilínea, limitada por comillas triples, se asiga automáticamente a la variable mágica **doc** como la documentación del módulo o programa y el resto de variables especiales como información adicional. Igualmente la primera línea de def() es la documentación de la función. La variable especial **name** es el nombre del módulo cuando se usa como tal, que en este caso vale cubo, pero tiene valor "**main**" cuando se ejecuta como un programa. De esta manera distinguimos cuando el código se está ejecutando como un programa o está siendo llamado como un módulo.

```
import cubo

In [20]: cubo.__doc__
Out[20]: 'Programa de calculo del cubo de un numero'

In [21]: cubo.cubo(3)
Out[21]: 27

In [22]: cubo.cubo.__doc__
Out[22]: 'Calcula el cubo de un numero'

In [23]: cubo.__version__
Out[23]: '1.0'
```

Para poder importar un módulo nuestro, debe estar en el directorio donde lo estamos llamando, o bien estar en una ruta incluida en el PATH de la librería o bien en la variable PYTHONPATH.

```
$ echo $PYTHONPATH
:/home/japp/codigo/lib/:/usr/local/aspylib/:/usr/local/lib/python2.7/dist-
↪packages/
```

Alternativamente, se puede incluir el **PATH** en el programa ejecutable añadiéndolo a la lista sys.path:

```
import sys
sys.path.append('/home/japp/mis_modulos/')
```

En Windows, funciona de forma idéntica pero usando las rutas de Windows:

```
sys.path.append('C:\mis_modulos')
```

Para modificar de forma temporal el `PYTHONPATH` en Windows haríamos:

```
C:\>set PATH=C:\Program Files\Python 3.6;%PATH%
C:\>set PYTHONPATH=%PYTHONPATH%;C:\mis_modulos
C:\>python
```

Si se quiere añadir permanentemente es algo más complicado. Desde el botón de inicio hay que buscar Propiedades del sistema (System properties) -> Advanced system settings y pinchar en el botón de variables de entorno, donde se pueden modificar la variables de entorno del sistema (solo el administrador).

2.1 Estructura de un paquete de Python

Los paquetes de python son un espacio de nombres que contiene varios módulos o paquetes, a veces relacionados entre ellos aunque no tiene porqué. Se crean en un directorio que debe incluir obligatoria-

mente un fichero especial llamado **__init__.py** que es el que indica que se trata de un paquete y luego pueden haber otros módulos e incluso otros paquetes. La siguiente es la estructura típica de un paquete:

```
mi_paquete/
    __init__.py
    modulo1.py
    modulo2.py
    utiles/
        __init__py
        utiles1.py
        config.py
```

El fichero **__init__.py** puede y suele estar vacío, aunque se puede usar para importar modulos comunes entre paquetes.

```
import mi_paquete

from mi_paquete import utiles1
```

2.2 La librería estándar de Python

La instalación básica de Python viene con una muy completa librería de módulos para todo tipo de tareas, incluyendo acceso a ficheros y directorios, compresión de ficheros, ejecución recurrente (multihilo), email, html, xml, csv y un largo etcétera. Lo más conveniente es consultar la documentación de la librería estándar para tener una idea de todo lo disponible, pero podemos probar los más importantes.

2.3 Creación y administración de ficheros

La forma más directa y práctica de interactuar con el sistema, independientemente de la plataforma, es empleando el módulo `os`, que básicamente es una interfaz para sistema operativo del ordenador que ejecuta el programa.

```
import os

os.chdir("/home/japp/Documentos/")

os.getcwd()
# /home/japp/Documentos/

# Esto no imita a ls, no distingue ficheros y directorios
ficheros = os.listdir(".")  # hay que poner una ruta

for fichero in ficheros:
    print os.path.isdir(fichero)  # .isfile(), islink()
```

Para mayor flexibilidad en la selección de ficheros, por ejemplo usar caracteres comodín, se puede usar el paquete `glob`:

```
from glob import glob

ficheros = glob("*.txt")
```

```
# Son listas también pero con una ruta relativa, así que no funciona igual␣
↪que listdir
ficheros = glob("/home/japp/")    # no devuelve nada
ficheros = glob("/home/japp/*")   # Esto si

os.mkdir("registro")
# os.makedirs('/home/japp/Documentos/datos/pruebas')  # Linux, Mac
# os.makedirs('C:\\Mis Documentos\\datos\\pruebas')   # Windows

os.chmod("registro", 0700)

os.rename("registro", "registros")
```

2.3.1 Lectura y escritura de ficheros de texto

Si queremos leer o escribir ficheros de texto primero hay que abrirlos en el modo adecuado (**r, w, a**) para terner una **instacia del fichero**, que luego se puede leer a `string` de varias formas.

```
# Leo un fichero CSV con código y nombre de paises
fichero = open("paises.csv")

contenido = fichero.read()  # Lo mete todo en un único string
fichero.close()

len(contenido)

print(contenido[:30])
#'nombre, name, nom, iso2, iso3,'

fichero = open("paises.csv")
lineas = fichero.readlines()  # Lee línea a línea, devuelve una lista
fichero.close()

len(lineas)
247
```

De haber querido separar por columnas, pudimos haber usado algo como:

```
nombre, name, nom, iso2, iso3, phone_code = lineas.split(";")
```

justo después de `readlines()`, al hacerlo, `split()` devuelve una lista de dos elementos (en este caso) que **desempaquetamos** en las variables que damos a la izquierda.

Podemos igualmente escribir un fichero creando un fichero en modo lectura y usar el método `write(str)` para guardar una cadena de texto o bien usar `writelines(lista)` para guardar el contenido de una lista de strings.

¿Y si el fichero es remoto? hay varias maneras de resolverlo, pero lo más cómodo es con el módulo `urllib`:

```
import urllib.request
import csv

# Fichero remoto
# https://gist.github.com/brenes/1095110
```

```
url = "https://gist.githubusercontent.com/brenes/1095110/raw/
↪f8eeb4a7efb257921e6236ef5ce2dbc13c50c059/paises.csv"

# Terremotos del día de USGS
# url = "https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_day.
↪csv"

# Leemos remotamente el fichero csv
respuesta = urllib.request.urlopen(url)

# Pasamos la instancia a un string
contenido = respuesta.read()  # o readlines()

# Usamos esta vez el módulo csv de Python para interpretar el CSV
reader = csv.reader(contenido)
```

Ahora probemos a hacer una selección de los paises que empiezan por "P", pero en su nombre, no en su código

```
# Lista de paises que empiezan por P, vacía al principio
lineas_P = []

for linea in lineas:
    codigo, nombre = linea.split(";")
        if nombre.startswith('P'):
            lineas_P.append(linea)

# Abro el fichero donde voy a guardar
f_out = open("paises_P.txt", "w")

f_out.writelines(lineas_P)
f_out.close()
```

El fichero resultante es un fichero igual que el anterior pero solo con los paises que empiezan con "P", uno por línea, pero es es línea a línea porque el fichero original incluye caracteres de nueva línea. El método `writelines(lista)` no escribe a líneas y éstas deben añadirse explícitamente:

```
# Lista de numeros enteros, que paso a string y añado nueva línea
numeros = [str(n)+"\n" for n in range(100)]

f_out = open("numeros.txt", "w")
f_out.writelines(numeros)
f_out.close()
```

Es posible guardar tambien variable en binario para usarlas después, empleando `shelve()`:

```
import shelve

shelf_file = shelve.open('datos')

shelf_file['numeros'] = numeros
shelf_file.close()

# Al cerrar el fichero se guardan los datos, que se pueden recuperar␣
↪abriendo el fichero.
```

```
shelf_file = shelve.open('datos')
shelf_file['numeros']
```

El módulo `os` tiene otros métodos útiles para interactuar con el sistema y los procresos que ejecuta.

```
os.getlogin()
#'japp'

os.getgroups()
#[191, 256, 294, 329, 350, 2000]

os.getenv('HOME')

os.putenv('HOME', '/scratch/japp')

os.uname()
# ('Linux', 'vega', '4.1.13-100.fc21.x86_64', '#1 SMP Tue Nov 10 13:13:20␣
↪UTC 2015', 'x86_64')
```

Si se desea más información sobre el equipo, se puede emplear el módulo `platform`, que da información más completa y detallada sobre y ordenador y el SO:

```
import platform

print('uname:', platform.uname())

print('system   :', platform.system())
print('node     :', platform.node())
print('release  :', platform.release())
print('version  :', platform.version())
print('machine  :', platform.machine())
print('processor:', platform.processor())
print('distribution:', " ".join(platform.dist()) )   # Linux, mac_ver()␣
↪para OS X

"""
uname: ('Linux', 'vega', '4.1.13-100.fc21.x86_64', '#1 SMP Tue Nov 10␣
↪13:13:20 UTC 2015', 'x86_64', 'x86_64')

system   : Linux
node     : vega
release  : 4.1.13-100.fc21.x86_64
version  : #1 SMP Tue Nov 10 13:13:20 UTC 2015
machine  : x86_64
processor: x86_64
distribution: fedora 21 Twenty One
"""
```

Si se desea mayor control sobre los ficheros y directorios, el módulo `shutil` permite operaciones con ficheros a alto nivel.

```
improt shutil

shutil.copy('paises.csv', 'paises-copy.csv')  # Copia un fichero

shutil.copytree("/home/japp/Documentos", "/home/japp/Documentos-copia")
↪# Copia el directorio y su contenido
```

```
shutil.move('paises-copy.csv', '/home/japp/Documentos/')    # Mueve un␣
↪fichero
```

¿Cómo borrar cosas? Existen tres métodos principales:

```
os.unlink(path)        # Borra el fichero en path
os.rmdir(path)         # Borra el directorio en path, que debe estar vacío
shutil.rmtree(path)    # Borra path recursivamente
```

Si queremos borrar con más cuidado podemos usar condicionales:

```
for filename in os.listdir("."):
    if filename.endswith('.csv'):
        os.unlink(filename)
```

En el ejemplo anterior hemos hecho un listado sencillo del directorio en el que estamos. Para hacer una exploración recursiva de un directorio, distinguiendo en ficheros y directorios, podemos usar `os.walk()`:

```
for directorio, subdirectorios, ficheros in os.walk("/home/japp/Documentos/
↪"):
   print('El directorio ' + directorio)
```

`os.walk()` devuelve una tupla de tres elementos con el nombre del directorio actual, una lista de subdirectorios que contiene y una lista de ficheros que contiene.

Con el módulo `zip` se pueden leer y escribir ficheros zip:

```
fichero_zip = zipfile.ZipFile('datos', 'w')
ficheros = ['medidas_PV_He.txt', 'medidas_radio.txt', 'bright_star.tsv']

for fichero in ficheros:
    newZip.write(fichero, compress_type=zipfile.ZIP_DEFLATED)

fichero_zip.close()

fichero_zip = zipfile.ZipFile("datos.zip")
fichero_zip.namelist()

# informacion sobre un fichero en concreto del zip
bright_star_info = fichero_zip.getinfo('bright_star.tsv')
bright_star_info.file_size
# 926482

bright_star_info.compress_size
# 248269

# Extraigo el contenido
fichero_zip.extract('bright_star.tsv', '/home/japp/Documents/')
fichero_zip.extractall()  # todos los ficheros
fichero_zip.close()
```

2.4 Llamadas al sistema

La forma más sencilla de ejecutar comandos sistema, por ejemplo para lanzar programas o ejecutar comandos de la consola es el método `os.system()`

```
import os

os.system('touch /home/japp/Documents')
```

Sin embargo `system()` es muy limitado y no permite recoger el resultado la ejecución, de haberla. Mucho más útil y potente es el módulo `subprocess`:

```
import subprocess

# Uso básico similar a os.system()
subprocess.call(['ls', '-l'])
```

Puesto que los canales de entrada y salida del proceso `call()` están ligados a la entrada y salida padre, no puede capturar la salida del comando que ejecuta, como ocurre con `os.system()`. Si queremos capturar la salida podemos emplear `check_output()` y luego procesar el resultado como queramos.

```
output = subprocess.check_output(['ps', '-x'])

print(output)
"""
  PID TTY      STAT   TIME COMMAND
 3901 ?        S      0:00 sshd: invweb@pts/2
 3902 pts/2    Ss     0:00 -bash
 4248 pts/2    Sl     0:02 gedit cdb_import.py
 4527 ?        Sl     0:00 /usr/libexec/dconf-service
 6134 ?        Sl     0:15 /usr/local/apache//bin/httpd -k start
13324 pts/2    Sl+    0:00 /usr/bin/python /usr/bin/ipython
13613 pts/2    R+     0:00 ps -x
26515 ?        S      0:03 sshd: invweb@pts/0
26516 pts/0    Ss+    0:00 -bash
"""

# Separo for filas
output_lines = output.split("\n")

# Trozo que contiene el comando
output_lines[1][27:]

# Busco los procesos que usan Python
resultados = []

for line in output_lines:
     if 'python' in line.lower():
         resultados.append(line[:5])  # Me quedo con trozo que tiene el PID

print(resultados)
```

Usando tuberías directamente podemos usar parámetros para indicar la entrada y salida y capturar errores. Veamos este ejemplo de una función que llama al `ping` del sistema:

```
def esta_viva(hostname):
    """
    Hace un ping a una maquina para saber si esta conectada
    """

    ping = subprocess.Popen(["ping", "-n", "-c 1", hostname],␣
↪stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out, error = ping.communicate()

    if error == "":
        print("El ordenador {} está conectado".format(hostname))
        return True
    else:
        print("El ordenador {} está KO".format(hostname))
        return False

esta_viva('vega')
```

2.5 Trabajando con fechas y tiempo

La librería estándar de Python incluye varios módulos para tratar y manipular fechas, tiempo e intervalos. Como con otros módulos, una vez importado el módulo se define un objeto específico que permite hacer malabarismos con fechas y tiempo. El módulo principal es `datetime`, que permite trabajar con fechas y tiempo mientras que el módulo `time`, ofrece métodos avanzados para tiempo, ignorando la fecha.

```
import datetime

print("La fecha y hora actuales: " , datetime.datetime.now()  # Devuelve␣
↪un objeto datetime
print("Fecha y hora en string con formato: " , datetime.datetime.now().
↪strftime("%Y-%m-%d %H:%M")


print("Año actual: ", datetime.date.today().strftime("%Y"))
print("Mes del año: ", datetime.date.today().strftime("%B"))
print("Semana del año: ", datetime.date.today().strftime("%W"))
print("Número de día de la semana: ", datetime.date.today().strftime("%w"))
print("Día del año: ", datetime.date.today().strftime("%j"))
print("Día del mes: ", datetime.date.today().strftime("%d"))
print("Día día de la semana: ", datetime.date.today().strftime("%A"))

import time

print("Segundos desde inicio de época: %s" %time.time())

# Para una fecha específica

fecha = datetime.date(1937, 10, 8)  #year, month, day
print(fecha.strftime("%A"))
# Friday

print(fecha.strftime("%b %d %Y %H:%M:%S"))
# Oct 08 1937 00:00:00
```

En el ejemplo anterior usamos el método `strftime()` para obtener un string en el formato deseado según la sintaxis de fechas de Python. De manera similar podemos usar `strptime()` para convertir un string de fecha a un objeto `date` o `datetime` de Python:

```
# Fecha en string
fecha_str = "2017-05-16 10:30:00"

# Formato en el que está la fecha en string
fecha_fmt = "%Y-%m-%d %H:%M:%S"

# Objeto datetime a partir de la fecha en string
fecha = datetime.datetime.strptime(fecha_str, fecha_fmt)

print(fecha.strftime("%A %d %B, %Y"))
# 'Tuesday 16 May, 2017'

# Cambio de idioma
import locale

idioma = locale.setlocale(locale.LC_TIME, "es_ES")
print(fecha.strftime("%A %d %B, %Y"))
# martes 16 mayo, 2017
```

```
# Intervalos de tiempo y operaciones con fechas

hoy = datetime.date.today()
print('Hoy:', hoy)

un_dia = datetime.timedelta(days=1)
print('Lapso de un día:', one_day)

ayer = hoy - un_dia
print('Ayer:', ayer)

manhana = hoy + un_dia
print('Manhana :', manhana)

print('Manhana - ayer:', manhana - ayer)
print('Ayer - manhana:', ayer - manhana)

ayer > hoy
False

ayer < hoy
True
```

Hay que tener en cuenta que los tiempos se toman de ordenador, incluyendo la zona horaria, por lo que generalmente serán en **hora local**. Si queremos convertir a otra zona horaria, debemos usar el módulo `pytz`:

```
# Hora local canaria actual
hora_local = datetime.datetime.now()
# datetime.datetime(2017, 5, 12, 10, 30, 0, 379146)

# Hora actual en UTC
hora_utc = datetime.datetime.utcnow()
# datetime.datetime(2017, 5, 12, 9, 30, 0, 226718)
```

```
from pytz import timezone

hora_us_pacific = hora_utc.replace(tzinfo=timezone('US/Pacific'))
```

Finalmente, el módulo `calendar` ofrece alguna funciones de calendario:

```
import calendar

cal = calendar.month(2017, 5)
print(cal)
      May 2017
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31


print(calendar.TextCalendar(calendar.MONDAY).formatyear(2017, 2, 1, 1, 2))
```

2.6 Conexión remota por FTP y SSH

La librería estándar de Python incluye un módulo ftp con todas las funcionalidades necesarias. Veamos un ejemplo para copiar ficheros locales al FTP público del IAC.

```
import ftplib
import os
from glob import glob

# Origen de los datos
origin_dir = "/home/japp/Documents/"

# directorio destino (en el servidor externo)
destination_dir = "in/curso_python"

# Lista de los ficheros a copiar, todos los *.py
files = glob(origin_dir + "*.py")

ftp = ftplib.FTP("ftp.iac.es")
login = ftp.login("japp@iac.es", "anonymous")
ftp.cwd(destination_dir)

os.chdir(origin_dir)

for filename in files:
    infile = open(filename, 'r')
    ftp.storlines('STOR ' + os.path.basename(filename), infile)
    infile.close()
```

Hay que fijarse que solo copia ficheros de uno en uno, si se quiere copiar recursivamente hay que implementar una copia recursiva con `os.walk()` o similar e ir creado directorios con `mkd()`.

No hay un módulo específico para `ssh`, pero se puede usar el del sistema usando el módulo `pexpect`, que permite manejar envío de información entre un servidor y un cliente, en ambas direcciones.

```
import pexpect
import time

host = "vega"
password = "secreto"

ssh_newkey = 'Are you sure you want to continue connecting'
ssh = pexpect.spawn("ssh {}".format(host))
i = ssh.expect([ssh_newkey, 'password:', host, pexpect.EOF], timeout=10)

if i == 0:
    ssh.sendline('yes')
    # Se dan una lista de posibles salidas del comando: nueva key,
    # la contraseña o el prompt del sistema si no pide contraseña
    i = ssh.expect([ssh_newkey, 'password: $', pexpect.EOF])
    ssh.sendline(password)
    ssh.expect(pexpect.EOF)
elif i == 1:
    ssh.sendline(password)
    ssh.expect(pexpect.EOF, timeout=10)
elif i == 2:
    pass

# Extraigo el resultado del comando
p = pexpect.spawn("df -h")
print(p.read())


ssh.close()
```

CAPÍTULO 3

Estilo de codificación y buenas prácticas

Además de una correcta y ordenada estructura general que deben tener los programa, es conveniente mantener ciertas **buenas prácticas de codificación** y el **estilo de codificación** recomendado. Estas normas no son obligatorias, como lo es la propia sintaxis del lenguaje, pero conviene seguir las recomendaciones de los desarrolladores de Python para facilitar la lectura del programa y ayudar a encontrar posibles errores.

Un ejemplo básico para entender a lo que nos referimos es el sangrado, que como hemos visto en Python en obligatorio, pero mientras la estructura de bloques sea correcta, a Python no le importa el número de espacios que se usen. Pues bien, aunque a Python le da igual, la recomendación es **usar cuatro espacios** (no tabuladores) para la sangrar bloques. Hay otras normas similares muy sencillas que debemos intentar seguir, como estas:

Cuando sea posible, define variables con nombres que tengan algún sentido o que puedas identificar fácilmente, no importa que sean más largas. Por ejemplo, en un programa podríaamos escribir:

```
a = 10.  # altura
b = 3.5  # base
print("El volumen es %.1f" % (a*b))
```

pero, ¿qué significan `a` y `b`? lo sabemos por el comentario (bien hecho), pero si más adelante nos encontramos con esas variables, tendremos que recordar cual es cual. Es mejor usar nombres con significado:

```
altura = 10.
base = 3.5
print("El volumen es %.1f" % (altura*base))
```

De hecho podemos usar el nombre para dar más información sobre la variable:

```
velocidad_metros_segundo = 12.5
angulo_radianes = 1.3
```

Las líneas de codigo no deben ser muy largas, como mucho 72 caracteres. Si se tiene una línea larga, se puede cortar con una barra invertida (\) y continuar en la siguiente línea:

```
print("Esta es una frase muy larga, se puede cortar con una \
       y seguir en la línea inferior.")
```

Dentro de paréntesis, corchetes o llaves, no dejar espacios inmediatamente dentro de ellos:

```
SÍ:  funcion(num[1], {pares: 2})
NO:  funcion( num[ 1 ], { pares: 2 } )
```

Justo después de coma, punto y coma y punto, separar con un espacio, para mayor claridad, pero no antes:

```
SÍ:  print x, y; x, y = y, x
NO:  print x , y ; x , y = y , x
```

Aunque en Python se pueden hacer varias declaraciones en una línea, se recomienda hacer sólo una en cada línea:

```
SÍ:  a = 10
     b = 20
NO:  a = 10; b = 20

SÍ:  if a > 3.14:
        print(a)
NO:  if a > 3.14: print(a)
```

3.1 Manipulación de listas

Aunque combinar iterables con elementos de control de flujo para manipular listas es muy sencillo con Python, hay métodos específicos más eficientes para hacer lo mismo. Pensemos el fitrado de datos de una lista:

```
# Seleccionar los números positivos
numeros = [-3, 2, 1, -8, -2, 7]
positivos = []
for i in positivos:
    if i > 4:
        positivos.append(i)
```

Aunque técnicamente es correcto, es más eficiente hacer algo como esto usando las posibilidades de Python:

```
numeros = [-3, 2, 1, -8, -2, 7]
positivos = [i for i in a if i > 4]

# o también:
positivos = filter(lambda x: x > 4, numeros)
```

Igualmente, aunque se puede hacer esto:

```
# Suma 3 a cada elemento de la lista
numeros = [-3, 2, 1, -8, -2, 7]
for i in range(len(numeros)):
    numeros[i] += 3
```

Es mejor hacer esto otro:

```
numeros = [-3, 2, 1, -8, -2, 7]
numeros = [i + 3 for i in numeros]

# o también:
numeros = map(lambda i: i + 3, numeros)
```

3.1.1 Documentación de código

Casi tan importante como la escritura de código, es su correcta documentación, una parte fundamental de cualquier programa que a menudo se infravalora o simplemente se ignora. Aparte de los comentarios entre el código explicando cómo funciona, el elemento básico de documentación de Python es el *Docstring* o cadena de documentación, que ya hemos visto. Simplemente es una cadena de texto con triple comillas que se coloca justo después de la definición de función o clase (ver programación orientada objetos, más adelante) que sirve de documentación a ese elemento.

```
def potencia(x, y):
    """
    Calcula la potencia arbitraria de un número
    """

    return x**y

  # Acceso a la documentación
  potencia.__doc__
  help(potencia)
```

Además de esta documentación básica, lo correcto es detallar mejor en el *Docstring* qué hace y cómo se usa la función o clase y los parámetros que necesita. Se recomienda usar el estilo de documentación del software de documentación sphinx, que emplea reStructuredText como lenguaje de marcado.

Veamos un ejemplo de una función bien documentada:

```
"""
power(x1, x2[, out])

First array elements raised to powers from second array, element-wise.

Raise each base in `x1` to the positionally-corresponding power in
`x2`.  `x1` and `x2` must be broadcastable to the same shape. Note that an
integer type raised to a negative integer power will raise a ValueError.

Parameters
----------
x1 : array_like
    The bases.
x2 : array_like
    The exponents.

Returns
-------
y : ndarray
    The bases in `x1` raised to the exponents in `x2`.
```

```
See Also
--------
float_power : power function that promotes integers to float

Examples
--------
Cube each element in a list.

>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> np.power(x1, 3)
array([  0,   1,   8,  27,  64, 125])

Raise the bases to different exponents.

>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> np.power(x1, x2)
array([  0.,   1.,   8.,  27.,  16.,   5.])

The effect of broadcasting.

>>> x2 = np.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> np.power(x1, x2)
array([[ 0,  1,  8, 27, 16,  5],
       [ 0,  1,  8, 27, 16,  5]])
"""
```

En este ejemplo de la función `power()` de `numpy` no solo se explica qué hace la función, sino que indica los parámetros de entrada y salida e incluso da algunos ejemplos de uso.

CAPÍTULO 4

Tratamiento de errores. Sentencia try-except

Hemos visto que cuando ocurren algún error en el código, Python detiene la ejecución y nos devuelve una **excepción**, que no es más que una señal que ha occurrido un funcionamiento no esperado o error en el programa, indicándonos aproximadamente qué fue lo que ocurrió.

Supongamos que tenemos un pequeño programa que por algún motivo realiza una división por cero.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May  1 12:00:00 2017

@author: japp
"""

a, b = 20, 0

resultado = a/b

print(resultado)
```

Al ejecutarlo tendremos el siguiente mensaje:

```
japp@vega:~$ python error.py
Traceback (most recent call last):
  File "test/errors.py", line 11, in <module>
    resultado = a/b
ZeroDivisionError: integer division or modulo by zero
```

En este ejemplo, el origen del error es fácil de identificar y el intérprete nos indica la línea en que ocurre. Si el programa es más complejo el posible que error se propague por varias partes del programa y no sea tan evidente encontrar el origen, pero el trazado del error (*Traceback*) que da el intérprete muestra el camino que siguió el código que finalmente produjo el error. Supongamos un programa como el anterior pero en el que el cálculo se hace en varios pasos:

```
a, b = 20, 0

def division(x, y):
    return x/y


def imprime_resultado(x, y):
    resultado = division(x, y)

    print("La división de {} entre {} es {}".format(resultado))

imprime_resultado(a, b)
```

```
japp@vega:~$ python error.py
Traceback (most recent call last):
  File "test/errors.py", line 22, in <module>
    imprime_resultado()
  File "test/errors.py", line 17, in imprime_resultado
    resultado = division(a, b)
  File "test/errors.py", line 13, in division
    return x/y
ZeroDivisionError: division by zero
```

Aunque el error es el mismo, el mensaje que devuelve el intérprete es mucho más largo porque sigue la traza del error hasta llegar al origen, que se muestra al final de la lista (linea 13), permitiéndonos ver cómo ha pasado por partes de código hasta llegar al origen del problema.

Con este mensaje, nos avisa del error indicando el tipo en la última línea, `ZeroDivisionError`, terminando la ejecución. Que Python nos dé tanta información al ocurrir una excepción es muy útil pero muy a menudo sabemos que estos errores pueden ocurrir y lo ideal es estar preparado capturando la excepción y actuar en consecuencia en lugar de interrumpir el programa o al menos, antes de interrumpirlo hacer procesos alternativos o preparar un cierre limpio del programa.

Para hacer esto podemos usar la sentencia `try-except`, que nos permite *probar* (Try) una sentencia y capturar un eventual error y hacer algo al respecto (except) en lugar de detener el programa directamente. En el ejemplo anterior podríamos hacer lo siguiente:

```
a, b = 23, 0
try:
    resultado = a/b
except:
    print("Hay un error en los valores de entrada")
```

Ahora el código intenta ejecutar `a/b` y de haber algún tipo de error imprime el mensaje indicado y sigue adelante en lugar abortar la ejecución del programa. Al hacer esto hemos "capturado" la excepción o error evitando que el programa se detenga, suponiendo que éste puede continuar a pesar del error. Nótese que de esta manera no sabemos qué tipo de error ha ocurrido, que antes se indicaba con la clave `ZeroDivisionError`, que es uno de los muchos tipos de errores que Python reconoce. Esto no es una buena práctica porque perdemos información sobre qué fue exactemente lo que causó el error, simplemente sabremos que "algo fue mal".

Podemos usar esta técnica para pedir al usuario un tipo de dato determinado que se comprueba constantemente:

```
while True:
    try:
        x = int(input("Dame un numero: "))
        break  # Si no da error, corto el while con break
    except ValueError:
        print("Eso no es un número, prueba otra vez...")
```

Si quisiéramos distinguir el tipo de error ocurrido, para tomar distintas acciones o mensajes, debemos especificarlo en `except`, por ejemplo:

```
a, b, c = 23, 0, "A"

try:
    resultado = a/b
except ZeroDivisionError:
    print("Error, division por cero.")
except TypeError:
    print("Error en el tipo de dato.")

# Resultado:
# Error, division por cero.

try:
    resultado = a/c
except ZeroDivisionError:
    print("Error, division por cero.")
except TypeError:
    print("Error en el tipo de dato.")

# Resultado:
# Error en el tipo de dato.
```

De esta manera, sabemos exactamente qué tipo de error se cometió en cada caso, una división por cero o un error en el tipo de dato (que es lo que indica `TypeError`).

```
import sys
a, b, c = 23, 0, "A"

try:
    resultado = a/b
except (ZeroDivisionError, ValueError):
    print("Error, division por cero o tipo de dato incorrecto.")
except:
    # El metodo exc_info() nos da informacion sobre la ejecucion
    # del programa y los errores si los hay
    print("Error inesperado:", sys.exc_info()[0])
    raise

# Devuelve:
# Error, division por cero o tipo de dato incorrecto.
```

En esto ocurre una división por cero y el error es capturado con el primer `except` sin indicar cual de las dos posibles excepciones indicadas en la tupla ha ocurrido (ZeroDivisionError, ValueError). Sin embargo si en el `try` hacemos la operación `resultado = a/c`, el error que ocurre no es del tipo `ZeroDivisionError` o `ValueError` (este último salta cuando en una operación o función el tipo de dato es correcto, pero no su valor, por ejemplo logaritmo de un número negativo) y el error será

capturado por el segundo `except` indicando que ocurrió un error no esperado, pero al usar la sentencia `raise` hacemos haga saltar el error ocurrido. De hecho, con `raise` es posible hacer saltar cualquier error si nuestro programa lo requiere.

Hay que recordar que la captura de excepciones no son para evitar que un programa se detenga, si no poder saber qué error ha ocurrido y se es posible tomar medidas alternativas. Si ocurre una excepción y el programa ya no puede seguir ejecutarse, o quiere seguir haciéndolo a pesar de haber una excepción, es posible que tengamos que hacer operaciones de notificación o de limpieza (enviar un mensaje, borrar ficheros temporales, cerrar ficheros que abrimos para leer, guardar un registro, etc.) antes de detener el programa o continuar con bloque de código siguiente. Para hacer esto podemos emplear la sentencia `finally` si no se cumple

```
try:
    a, b = eval(input("Dame dos numeros para dividir: "))
    inverse = a / b
except ValueError:
    print("Debes dar valores numericos")
except ZeroDivisionError:
    print("División por cero")
finally:
    print("Gracias por usar el program.")
```

La sentencias `finally` se ejecuta siempre en cualquier caso.

Conviene consultar la documentación oficial de Python para tener más información sobre la captura de excepciones y los tipos de errores reconocidos.

4.1 Encontrando errores con el depurador

Cuando los programas son más complicados no es sencillo encontrar el origen de errores como los ejemplos que acabamos de ver. Cuando las cosas se complican podemos un depurador, una poderosa herramienta para trazar y encontrar errores. Aunque existen varios depuradores para Python, incluso con interfaz gráfica, el depurador por defecto de Python, `pdb` hace un gran trabajo ayudándonos a encontrar errores ocultos.

Lo primero que debemos hacer es importar el depurador, que se hace como un módulo cualquiera y luego añadir un punto de traza en nuestro código donde queremos empezar a analizar. Consideremos en el ejemplo que hicimos hace poco y añadásmosle el depurador:

```
import pdb

a, b = 20, 0

pdb.set_trace()

def division(x, y):
    return x/y

def imprime_resultado(x, y):

    resultado = division(x, y)

    print("La división de {} entre {} es {}".format(x, y, resultado))
```

```
imprime_resultado(a, b)
```

Hemos puesto un **punto de control** justo después de definir las variables. Al encontrarse esta línea hará lo siguiente:

1. Detener la ejecución del programa.
2. Mostrar línea actual (el siguiente comando que va a ejecutar).
3. Esperar por el usuario para de alguna entrada.

Es decir, el programa se detiene en la línea donde está `pdb.set\_trace()` y aparece el *prompt* del depurador (`(Pdb)`) e indica el siguiente línea a ejecutar:

```
japp@vega:~$ python3 errors.py
> errors.py(15)<module>()
-> def division(x, y):
(Pdb)
```

Ahora podemos ejecutar algunos comandos del depurar según lo que queramos hacer:

s(tep) Ejecuta la línea actual incluso dentro de una función, deteniéndose en la siguiente línea.

n(ext) Continúa con la ejecución hasta la siguiente línea de la función actual o hasta que encuentre un return. La diferencia con `step` es que no entra dentro de funciones, siendo la ejecución más rápida.

c(ont(inue)) Continúa con la ejecución y solo se detiene si encuentra un punto de control.

r(eturn) Continúa con la ejecución de la función actual hasta que encuentra un return.

l(ist) [primero[, ultimo]] Muestra 11 línea de código en torno a la línea actual. Si añaden los argumentos `[primero[, ultimo]]` se muestran las líneas indicadas.

p expresión Evalúa una expresión en contexto actual imprimiendo su valor.

q(uit) Sale del depurador abortando la ejecución.

Nuestro programa está ahora detenido en la definición de la función `division()`; si usamos el comando `n+<Enter>` el depurador continuará con la siguiente línea en el programa principal sin entrar en el bloque de `division()` y se detendrá otra vez en la definición de `imprime_resultado()` y si repetimos lo mismo llegará a la ejecución de `imprime_resultado()` sin haber entrado en la definición anterior. Estando en la línea `imprime_resultado()` podemos ahora usar `s+<Enter>` para hacer lo mismos que `s` pero esta vez entrar en el bloque que define `imprime_resultado()` deteniéndose en `resultado = division(x, y)`; ahí podemos hacer lo mismo, usar `s` para entrar en la definición de `division()` y terminar encontrando el origen del problema, la división `x/y` en el `return`.

Si ya sospechamos que error está en función `division()`, podemos poner ahí el punto de control, o poner varios si lo necesitamos y saltar con `c` (continue) hasta el siguiente punto de control.

Para evaluar los valores que tienen las variables en un punto concreto de la ejecución, podemos imprimir sus valores con el comando `p` para comprobar si es lo que esperamos:

```
japp@vega:~$ python3 errors.py
-> def division(x, y):
(Pdb) p a, b
```

```
(20, 0)
(Pdb)
```

Incluso es posible cambiar los valores de las constantes en plena ejecución y continuar hasta el final. Si la variable que queremos cambiar coincide con algún comando del depurador (n, c, b, etc.), debemos añadir `!` para indicar que es una variable y no el parámetro:

```
japp@vega:~$ python3 errors.py
-> def division(x, y):
(Pdb) !b = 2.5
(Pdb) c
La división de 20 entre 2.5 es 8.0
japp@vega:~$
```

Así, hemos cambiado el valor cd `b` a `2.5` y continuado la ejecución hasta el final con `c` y el programa finaliza sin error.

CAPÍTULO 5

Programación orientada a objetos con Python

Python permite varios **paradigmas de programación**, incluyendo la programación orientada a objetos (POO). La POO es una manera de estructurar el código que le hace especiamente efectivo organizando y reutilizando código, aunque su naturaleza abstracta hace que no sea muy intuitivo cuando se empieza.

La programación orientada a objetos en Python es opcional y de hecho hasta ahora no la hemos usado directamente, aunque indirectamente la hemos estado usando desde el principio. Aunque su mayor ventaja aparece con los programas largos y más complejos, es muy útil entender cómo funciona la POO, ya que es así como Python funciona por dentro.

La idea básica es sencilla. Si tenemos un tipo de dato más complejo de los que hemos visto hasta ahora como listas o diccionarios y queremos crear un nuevo **tipo de dato** con propiedades particulares, podemos definirlo con una **clase**, algo parecido a una función `def`. Supongamos que queremos crear un tipo de dato llamado Estrella (Star), que para empezar solo tendrá un nombre, podemos escribir:

```
# Creamos star.py

class Star(object):
    """Clase para estrellas"""

    def __init__(self, name):
        self.name = name

    # Método especial que se llama cuando se hace print
    def __str__(self):
        return "Estrella {}".format(self.name)
```

La clase tiene una función principal especial que construye el elemento de la clase (llamado *objeto*) que es `__init__()` y que se ejecuta cuando crea un nuevo objeto de esa clase; hemos puesto `name` como parámetro único obligatorio, pero no tiene porqué tener ninguno.

La misteriosa variable `self` con la que empieza cada función (llamadas métodos en los objetos), se refiere al objeto en concreto que estamos creado, esto se verá más claro con un ejemplo. Ahora ya podemos crear objetos tipo Star:

```
import star

estrella1 = star.Star("Altair")

print(estrella1)
# Estrella Altair

print(estrella1.name)
# Altair
```

Al crear el objeto con nombre `estrella1`, que lo que la definición de la clase llamamos `self` tenemos un tipo de dato nuevo con la propiedad `name`. Ahora podemos añadir algunos métodos que se pueden aplicar al objeto Star:

```
class Star:
    """Clase para estrellas

    Ejemplo de clases con Python

    Fichero: star.py
    """

    # Numero total de estrellas
    num_stars = 0

    def __init__(self, name):
        self.name = name
        Star.num_stars += 1

    def set_mag(self, mag):
        self.mag = mag

    def set_par(self, par):
        """Asigna paralaje en segundos de arco"""
        self.par = par

    def get_mag(self):
        print "La magnitud de {} de {}".format(self.name, self.mag)

    def get_dist(self):
        """Calcula la distancia en parsec a partir de la paralaje"""
        print "La distacia de {} es  {:.2f} pc".format(self.name, 1/self.
↪par)

    def get_stars_number(self):
        print "Numero total de estrellas: {}".format(Star.num_stars)
```

Ahora podemos hacer más cosas un objeto `Star`:

```
import star

# Creo una instancia de estrella
altair = star.Star('Altair')

altair.name
# Devuelve 'Altair'
```

```
altair.set_par(0.195)

altair.get_stars_number()
# Devuelve: Numero total de estrellas: 1

# Uso un método general de la clase
star.pc2ly(5.13)
# Devuelve: 16.73406

altair.get_dist()
# Devuelve: La distancia de Altair es  5.13 pc

# Creo otra instancia de estrella
otra = star.Star('Vega')

otra.get_stars_number()
# Devuelve: Numero total de estrellas: 2

altair.get_stars_number()
# Devuelve: Numero total de estrellas: 2
```

¿No resulta familiar todo esto? es similar a los métodos y propiedades de elementos de Python como strings o listas, que también son objetos definidos en clases con su métodos.

Los objetos tienen una interesante propiedad llamada **herencia** que permite reusar propiedades de otros objeto. Supongamos que nos interesa un tipo de estrella en particular llamada *enana blanca*, que son estrellas con algunas propiedades especiales, por lo que necesitaremos todas las propiedades del objeto `Star` y alguna nueva que añadiremos:

```
class WBStar(Star):
    """Clase para Enanas Blancas (WD)"""

    def __init__(self, name, type):
        """Tipo de WD: dA, dB, dC, dO, dZ, dQ"""
        self.name = name
        self.type = type
        Star.num_stars += 1

    def get_type(self):
        return self.type

    def __str__(self):
        return "Enana Blanca {} de tipo {}".format(self.name, self.type)
```

Ahora, como parámtro de `class`, en lugar de poner `object` para crear un objeto nuevo, hemos puesto `Star` para que **herede** las propuedades de esa clase. Así, al crear un objeto `WDStar` estamos creando un objeto distinto, con todas las propiedades y métodos de `Star` y una propiedad nueva llamada `type`. Además sobrescribimos el resultado al imprimir con `print` definiendo el método especial `__str__`.

Como vemos, los métodos, que son las funciones asociadas a los objetos, sólo se aplican a ellos. Si en nuestro fichero la clase, que hemos llamado `star.py` y que contiene por ahora las clases `Star` y `WDStar` añadimos una función normal, ésta se puede usar de forma habitual:

```
class Star(Star):
  ...
```

```
class WBStar(Star):
  ...


def pc2ly(dist):
    """Convierte parsec a anhos luz"""
    return dist*3.262
```

Y como siempre:

```
import star

# Convierte parsecs en años luz
distancia_ly = Star.pc2ly(10.0)
```

CAPÍTULO 6

Cálculo numérico con Numpy

Aunque Python tiene varios tipos de datos estructurados, en la práctica no son nada adecuados para cálculo numérico. Veamos un ejemplo de un cálculo numérico básico empleando listas:

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

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

In [3]: print(lista*2.5)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/home/japp/<ipython console> in <module>()

TypeError: can't multiply sequence by non-int of type 'float'
```

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

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

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

Cuando realmente queremos hacer cálculos con listas de números, debemos usar los arrays. El módulo numpy nos da acceso a los arrays y a una enorme cantidad de métodos y funciones aplicables a los mismos. Naturalmente, `numpy` incluye funciones matemáticas básicas similares al módulo `math`, las completa con otras más elaboradas y además incluye algunas utilidades de números aleatorios, ajuste lineal de funciones y muchas otras.

Para tabajar con numpy y los arrays, importamos el módulos de alguna manera:

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

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

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

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

Existen métodos para crear arrays automáticamente:

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

In [14]: lista_ceros = zeros(10)              # Array de 10 ceros (floats)
In [15]: print(lista_ceros)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

In [16]: lista_unos = ones(10)                # Array de 10 unos (floats)
In [17]: print(lista_unos)
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]

In [18]: otra_lista = linspace(0,30,8)        # Array de 8 números, de 0 a␣
↪30 ambos incluidos
In [19]: print(otra_lista)
[  0.           4.28571429   8.57142857  12.85714286  17.14285714
  21.42857143  25.71428571  30.        ]
```

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

```
In [18]: print(numeros[3:8])            # Elementos desde el tercero al␣
↪septimo
[ 3.  4.  5.  6.  7.]

In [19]: print(numeros[:4])             # Elementos desde el primero al␣
↪cuarto
[ 0.  1.  2.  3.]

In [20]: print(numeros[5:])             # Elementos desde el quinto al final
[ 5.  6.  7.  8.  9.]

In [21]: print(numeros[-3])             # El antepenúltimo elemento␣
↪(devuelve un elemento, no un array)
7.
```

```
In [24]: print(numeros[:])              # Todo el array, equivalente a␣
↪print(numeros)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]

In [25]: print(numeros[2:8:2])          # Elementos del segundo al septimo,␣
↪pero saltando de dos en dos
[ 2.  4.  6.]
```

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

```
In [28]: print(len(numeros))
10
In [29]: print(numeros.shape)
(10,)
```

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

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

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

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

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

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

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

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

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

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

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

6.1 Operaciones con arrays

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

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

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

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

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

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

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

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

In [45]: print(z)
[ 5.6  7.3  7.7  2.3  4.2  9.2  0.   1.   2.   3.   4.   5. ]

In [46]: z = concatenate((x,[7]))

In [47]: print(z)
[ 5.6  7.3  7.7  2.3  4.2  9.2  7. ]
```

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

Para añadir elementos, numpy tiene las funciones `insert()` y `append()`, que funcionan de manera similar a sus equivalentes en listas, pero en este caso son funciones y no métodos que se aplican a un array, si no que el array en cuestión hay que darlo como parámetro:

```
# Añadimos el elemento 100 al array z, al final
In [55]: z = append(z, 100)
In [56]: print(z)
[   5.6    7.3    7.7    2.3    4.2    9.2    7.   100. ]

 # Añadimos el elemento 200 al array z, en el tercer puesto (índice 2)
In [57]: z = insert(z, 2, 200)
In [58]: print(z)
[   5.6    7.3  200.     7.7    2.3    4.2    9.2    7.   100. ]
```

Como se ve, a diferencia de las listas, el primer parámetro es el array y luego el elemento que se quiere añadir, en el caso de `append()` y el array, la posición y luego elemento a añadir en el caso de `append()`. Esto se debe a que estas funciones **devuelven una copia del array** sin modificar el original como hacen los métodos de listas correspondientes. Si en lugar de un elemento a insertar se da una lista y otro array, añade todos los elementos de la lista (a `append()` habría que dar también una lista de posiciones, como segundo parámetro).

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

```
In [5]: z.max()    # Valor máximo de los elementos del array
Out[5]: 9.1999999999999993

In [6]: z.min()    # Valor mínimo de los elementos del array
Out[6]: 2.2999999999999998

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

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

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

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

Los métodos, que se operan de la forma `z.sum()` también pueden usarse como funciones de tipo `sum(z)`, etc. Consulten el manual de numpy para conocer otras propiedades y métodos de los arrays o simplemente acudan y consulten la "ayuda" de las funciones que quieran utilizar.

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

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

In [22]: A*B
Out[22]: array([False, False,  True], dtype=bool)

In [29]: C = array([1, 2, 3])

In [30]: A*C
Out[30]: array([1, 0, 3])

In [31]: B*C
Out[31]: array([0, 0, 3])
```

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

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

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

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

# Array de índices booleanos
```

```
In [39]: indices2 = array([False, True, True, False, False, True, False,␣
↪False, True, True])

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

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

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

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

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

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

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

6.2 Arrays multidimensionales

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

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

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

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

```
# Array unidimensional de 9 elementos y cambio su forma a 3x3
In [64]: arr3 = arange(9).reshape((3,3))
In [65]: print(arr3)
[[0 1 2]
[3 4 5]
[6 7 8]]

In [69]: arr2.shape
Out[69]: (4, 1)
```

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

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

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

Igualmente podemos modificar un array bidimensional usando sus índices:

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

In [94]: print(arr0)
array([[ 88,  20,  30],
       [ 50,  60, 999],
       [  0,  20,  30]])
```

6.3 Cambiando el tamaño de arrays

Hemos visto que es fácil quitar y poner elementos nuevos en un array unidimensional. Pero con dos o más dimensiones es algo más complicado porque estamos limitados a la estructura y número de elementos del array. Podemos cambiar la forma (`shape`) de un array a otra que tenga el mismo número de elementos fácilmente usado `reshape()`:

```
In [91]: numeros = arange(10000)  # Array unidimensional de 10000 números
In [92]: numeros_2D = numeros.reshape((100, 100))
In [93]: numeros_3D = numeros.reshape((100, 10, 10))

In [94]: numeros.shape
Out[94]: (10000,)

In [95]: numeros_2D.shape
Out[95]: (100, 100)

In [95]: numeros_3D.shape
Out[95]: (100, 10, 10)
```

Para añadir más filas o columnas a un array, la forma más efectiva en crear un array nuevo con la forma deseada y luego añadir las filas o columnas, por ejemplo:

> In [100]: A = arange(0, 10) In [101]: B = arange(100, 1100, 100) In [102]: C = np.zeros((len(A), 2)) # 10 filas, dos columnas
>
> In [103]: C[:,0] = A In [104]: C[:,1] = B

Existen otros métodos de manipulación de la forma de los arrays como `hstack()`, `vstack()` o `tile()` entre otras.

6.4 Fitros y máscaras de arrays. Arrays enmascarados.

Una de las mejores utilidades de numpy es trabajar con índices y máscaras de datos para limitar o seleccionar parte de los datos. Supongamos que tenemos un array de datos, pero que solo nos interesa los positivos, que queremos manipular después. Hay varias formas de seleccionarlos definiendo un array máscara con la condición que nos interesa:

```
In [264]: datos = array([3, 7, -2, 6, 7, -8, 11, -1, -2, 8])

In [265]: datos
Out[265]: array([ 3,  7, -2,  6,  7, -8, 11, -1, -2,  8])

In [266]: mask = datos >= 0

In [267]: mask
Out[267]: array([ True,  True, False,  True,  True, False,  True, False,␣
↪False,  True], dtype=bool)

In [268]: datos*mask
Out[268]: array([ 3,  7,  0,  6,  7,  0, 11,  0,  0,  8])

In [269]: datos[mask]
Out[269]: array([ 3,  7,  6,  7, 11,  8])
```

Usando un array `mask` de booleanos, podemos operar con el el array de datos, cuando un valor se multiplica por `True` es equivalente a **multiplicarse por 1** y si es con `False`, a** multiplicarse por 0**. Por eso el resultado es un array del mismo tamaño, pero los elementos que no cumplen la condición se hacen 0.

Si por el contrario usarmos usamos `mask` como un **array de índices**, el resultado es un array con los elementos cuyo índice corresponda con `True`, ignorando los de índice `False`. Usaremos uno u otro

según lo que queramos hacer, el truco consiste es crear de manera correcta la máscara de datos.

Veamos el caso de un array 2D con dos columnas, pero queremos limitar todos los datos en criterios en las dos columnas. Primero creamos una máscara como producto de las dos condiciones, y luego la usamos como array de índices en el array original:

```
In [270]: from numpy import random

In [278]: datos2 = random.randint(-10, 20, (10,2))

In [279]: datos2
Out[279]:
array([[  0,  10],
       [  5,  18],
       [ 19,   4],
       [  4,  19],
       [ -2,  -2],
       [ 11, -10],
       [ -4,   4],
       [  5,   6],
       [ 13,  13],
       [  7,  13]])

# Solo queremos los datos que el la columna 0 sean mayores
# que 0 pero menores que 10 en la columna 1
In [284]: condicion1 = datos2[:,0] > 0
In [285]: condicion1 = datos2[:,1] < 10
In [286]: mask_col0 = condicion1*condicion1

In [287]: mask_col0
Out[287]: array([False, False,  True, False, False,  True, False,  True,␣
↪False, False], dtype=bool)

In [288]: datos2[mask_col0]
Out[288]:
array([[ 19,   4],
       [ 11, -10],
       [  5,   6]])
```

Como se ve, el resultado en un array de dos columna, donde en la primera columna son todos positivos y en la segunda menores que +10. ¿Y si queremos que al menos se cumpla una condición? Simplemente tenermos que sumar las dos máscaras (las de cada columna) en lugar de multiplicarla, básicamente es como multiplicar o sumar unos o ceros (`True` o `False`).

Como este tipo de operaciones tienen mucho potencial y pueden llegar a se complejas. `numpy` tiene un módulo que puede ayudar en estos casos, que es el de arrays enmascarados (`numpy.ma`). Se trata de un tipo de datos que permite ignorar algunos elementos de un array según ciertas condiciones. Vemos un ejemplo:

```
In [300]: import numpy.ma as ma
In [301]: x = array([1, 2, 3, -1, 5])

In [302]: # Enmascaramos en cuarto elemento

In [303]: mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])

In [304]: print(mx.mean())
```

```
2.75

In [305]: print(x.mean())
2.0
```

Como se ve, el array enmascarado ha ignorado el valor que se enmascara con `True` dando un resultado distinto en la media, pero no lo ha eliminado del array.

```
In [313]: x.view(ma.MaskedArray)
Out[313]:
masked_array(data = [ 1  2  3 -1  5],
             mask = False,
       fill_value = 999999)


In [314]: mx.view(ma.MaskedArray)
Out[314]:
masked_array(data = [1 2 3 -- 5],
             mask = [False False False  True False],
       fill_value = 999999)
```

Podemos usar algunas funciones de `ma` para crear las máscaras, como `masked_greater()`, `masked_inside()`, `masked_where()`, etc.

```
In [320]: a = np.arange(4)
In [321]: print(a)
array([0, 1, 2, 3])

In [322]: ma.masked_where(a <= 2, a)
masked_array(data = [-- -- -- 3],
      mask = [ True  True  True False],
      fill_value=999999)
```

6.5 Arrays estructurados

Numpy tiene funcionalidades para crear y manipular arrays con contenidos complejos o estructurados, también llamado *record arrays*, que permiten tratar arrays por estructuras o por campos de estruturas. Vemos un ejemplo con un array con distintos tipos de datos:

```
In [350]: x = zeros((2,), dtype=('i4, f4, a10'))
In [351]: x[:] = [(1, 2., 'Que'), (2, 3.0, "pasa")]
In [352]: print(x)
array([(1, 2.0, 'Que'), (2, 3.0, 'pasa')],
     dtype=[('f0', '>i4'), ('f1', '>f4'), ('f2', '|S10')])
```

Se trata de un array 1D con dos entradas (o *records*) y cada una de ellas posee tres elementos de distinto tipo, indicados con la propiedad `dtype`. En este caso son un **entero de 32 bit**, un **decimal de 32 bit**, y un **string de longitud máxima 10**. El `dtype` de numpy describe cómo interpretar cada elemento en bytes de bloques de memoria fijos. No sólo se trata de si son *float*, *int*, etc., el `dtype` describe lo siguiente:

- Tipo de dato (int, float, objeto Python, etc.)
- Tamaño del dato (cuantos bytes puede ocupar)

- Orden de bytes de datos (little-endian o big-endian)
- Si son datos estructurado (por ejemplo mezcla de tipos de dato), también:
  - Nombre de los campos
  - Tipo de dato de cada campo
  - Qué parte del bloque de memoria ocupa cada campo
  - Si en dato es un sub-array, su forma y tipo de dato

De manera resumida, para definir el tipo de cada elemento podemos usar una de las siguientes cadenas:

```
b1, i1, i2, i4, i8, u1, u2, u4, u8, f2, f4, f8, c8, c16, a<n>
```

que representan, respectivamente, **bytes**, **ints**, **unsigned ints**, **floats**, **complex** y **strings de longitud fija**. También se pueden usar los tipos de datos estándar de Python equivalentes (`int`, `float`, etc.)

Teniendo un array estructurado como el anterior, podemos ver cada elemento haciendo el indexado habitual:

```
In [355]: print(x[0])
(1, 2.0, 'Que')
```

Pero también podemos acceder a cualquier campo (columna) usando el nombre que se le ha asignado (`f0, f1, f2, ..., f<N-1>`):

```
In [356]: x['f0']
array([1, 2], dtype=int32)

In [357]: y = x['f1']
In [358]: print(y)
array([ 2.,  3.], dtype=float32)

In [359]: y[:] = 10*y
In [360]: print(y)
array([ 20.,  30.], dtype=float32)

In [361]: print(x)
array([(1, 20.0, 'Que'), (2, 30.0, 'pasa')],
      dtype=[('f0', '<i4'), ('f1', '<f4'), ('f2', 'S10')])
```

En el ejemplo anterior vemos cómo modificar el contenido de una columna, pero con la definición de `y = x['f1']` no hemos hecho una copia de la columna, si no **una referencia** a ella que apunta a la misma zona de memoria, por lo que si modificamos `y` también cambiaremos el array original `x`. Si realmente queremos una copia de una columna sin modificar la original, podemos usar `y2 = x['f1'].copy()`.

Veamos otro ejemplo con distintos tipos de datos y estructura:

```
In [370]: x = np.zeros(3, dtype='3i1, f4, (2,3)f8')
In [371]: print(x)
array([([0, 0, 0], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]),
       ([0, 0, 0], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]),
       ([0, 0, 0], 0.0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])],
      dtype=[('f0', 'i1', (3,)), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
```

Sin embargo, existes varias formas o sintaxis para crear array estructurdos (mejor ver la documentación). Por ejemplo usarmos diccionarios, podemos asignar un nombre a nuestras columnas:

```
In [372]: x = zeros(3, dtype={'names':['columna1', 'columna2'], 'formats':[
↪'i4','f4']})
In [373]: print(x)
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('columna1', '<i4'), ('columna2', '<f4')])

In [374]: print(x['columna1'])
array([0, 0, 0], dtype=int32)
```

Gracias al uso de campos (columnas) por nombre, podemos crear arrays a partir de columnas de un array:

```
In [390]: x = np.zeros(3, dtype='i, i, i')

In [391]: print(x)
array([(0, 0, 0), (0, 0, 0), (0, 0, 0)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<i4')])

In [392]: # Modificamos un par de entradas del array
In [393]: x[1] = 1, 2, 3

In [393]: x[2] = 40, 50, 60

In [395]: print(x)
array([(0, 0, 0), (1, 2, 3), (40, 50, 60)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<i4')])

In [396]: print( x[ ['f0', 'f2'] ] )
array([(0, 0), (1, 3), (40, 60)],
      dtype=[('f0', '<i4'), ('f2', '<i4')])
```

Es decir, tenemos un nuevo array hecho con la primera y tercera columna del original, usando una lista con con los nombres de los campos.

6.6 Lectura y escritura de datos con numpy

`numpy` incluye algunos métodos de lectura de ficheros de texto que nos pueden facilitar la vida si son relativamente sencillos. En más sencillo es `loadtxt()`; si todos las columnas del fichero son numéricas, basta con indicar el delimitador de columnas si es distinto de espacios:

```
In [400]: # Leo un fichero de datos con campos delimitados por ";"
In [401]: data = loadtxt("medidas_radio.txt", delimiter=";")

In [402]: data.shape
Out[402]: (480, 2)
```

Si hay más de una columna como en este ejemplo, `loadtxt()` devuelve un array bidimiensional en el que **primera dimensión o eje son las filas** y el segundo las columnas, de maneras que el fichero que acabamos de leer tiene 480 filas y dos columnas. Quizás sea más práctico poner la columnas por separado, para lo que podemos hacer:

```
In [405]: tiempo = data[:,0]   # tiempo, la primera columna
In [406]: masa = data[:,1]     # masa, la segunda columna
```

pero si al leer añadimos el parámetro `unpack=True`, `loadtxt()` desempaqueta por columnas en lugar de por filas (como si invirtiera el array), perminiéndonos desempaquetar en variables las columnas, que ahora están en el eje 0:

```
In [410]: tiempo, masa = loadtxt("medidas_radio.txt", delimiter=";",␣
↪unpack=True)
```

Si el fichero a leer tiene distintos tipos datos (string y float), hay que indicar con el parámetro `dtype` la lista de tipos de dato que tienen las columnas que queremos leer. En este caso es más práctico usar el método `genfromtxt()`, que similar a `loadtxt()` pero más flexible para leer columnas de distinto tipo. Si usamos `genfromtxt()` con el parámetro `dtype`, que puede ser una lista con tuplas nombre-tipo, podemos indicar el nombre de la columna y el tipo dato que contiene, creando un array estructurado como vimos antes:

```
In [420]: dtypes = [('tiempo', 'float'), ('masa', 'float')]

In [421]: data = np.genfromtxt('medidas_radio.txt', dtype=dtypes)
```

En este caso ya no hay que usar el desempaquetado, porque tenemos un array estructurado con columnas con nombre `data['tiempo']` y `data['masa']`.

De manera parecida, podemos usar `savetxt()` para guardar un fichero de datos por columnas:

```
# Guardo un array datos en un fichero de texto, con los campos
# delimitados por tabulador (\t) en formato float con dos decimales
# y le paso una cabecera
In [426]: savetxt('datos.tsv', data, delimiter='\t', fmt='%.2f', header=
↪"tiempo\t masa")
```

En este ejemplo delimitamos las columnas por tabuladores (`\t`), escribimos los números como floats con dos decimales (`%.2f`, por defecto es notación científica) y añadimos un string que hace de cabecera.

6.7 Cálculo matricial con numpy

`numpy` incluye algunas funciones para álgebra y cálculo matricial, en el que es clave el tipo de dato `matrix`. Es similar al array, pero opera como una matrix y tiene método propios de las matrices.

```
In [425]: A = array([[1,2,3], [4,5,6], [7,8,9]])

In [426]: Am = mat(A)

In [428]: A*A  # Aquí hace producto matricial, no elemento a elemento como␣
↪con arrays
Out[428]:
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])

In [429]: Am*Am
Out[429]:
matrix([[ 30,  36,  42],
```

```
        [ 66,  81,  96],
        [102, 126, 150]])

In [433]: Am.T      # Matriz traspuesta

In [437]: Am.diagonal()
Out[437]: matrix([[1, 5, 9]])

In [438]: linalg.det(Am)    # Determinante
Out[438]: 6.6613381477509402e-16


In [441]: linalg.eigvals(Am)   # Autovalores
```

CAPÍTULO 7

Representación gráfica de funciones y datos

Hay una gran variedad de módulos para hacer gráficos de todo tipo con Python, pero el estárdar *de facto* en ciencia es matplotlib. Se trata de un paquete grande y relativamente complejo que entre otros contiene dos módulos básicos, `pyplot` y `pylab`.

`pyplot` ofrece una interfaz fácil para crear gráfico fácilmente, automatizando la creación de figuras y ejes automáticamente cuando hace un gráfico. Por otra parte, `pylab` combina la funcionalida de `pyplot` para hacer gráficos con funcionalidad de `numpy` para hacer cálculos con arrays usando un único espacio de nombres muy parecido a Matlab.

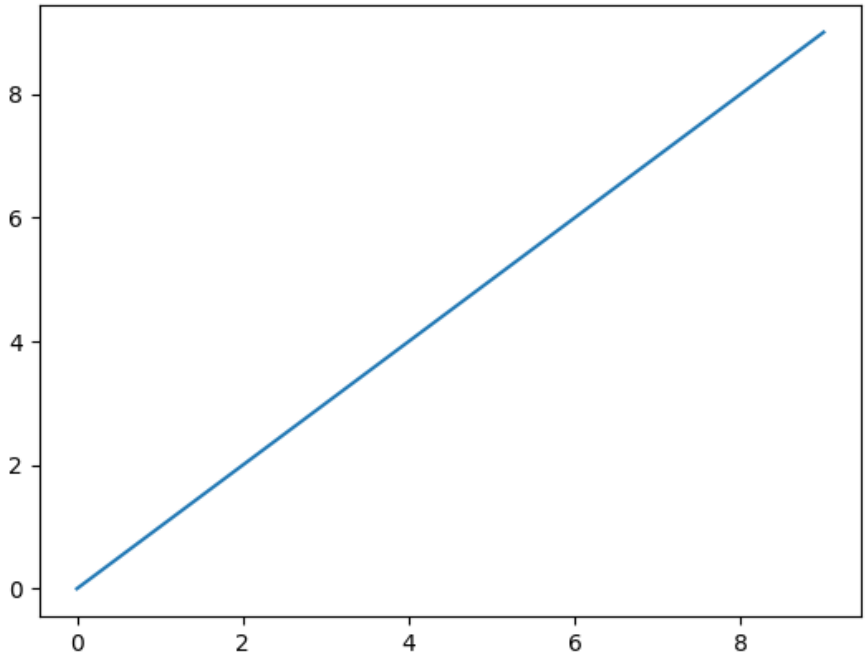
Por esto, es posible que nos encontremos dos formas comunes de usar la interfaz de `matplotlib`:

```
# importar todas las funciones de pylab
from pylab import *

# importar el módulo pyplot
import matplotlib.pyplot as plt
```

Al usar la primera opción también importamos `numpy` como `np`, entre otras cosas. En general, se recomienda usar pylab cuando se trabaja interactivamente y **pyplot** cuando se usan programas.

Empecemos creando el gráfico más sencillo posible:

```
In [1]: from pylab import *       # importar todas las funciones de pylab
In [2]: x = arange(10.)           # array de floats, de 0.0 a 9.0

In [3]: plot(x)                   # generar el gráfico de la función y=x
Out[3]: [<matplotlib.lines.Line2D object at 0x9d0f58c>]

In [4]: show()                    # mostrar el gráfico en pantalla
```



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

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

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

```
import numpy
import matplotlib
from matplotlib import pylab, mlab, pyplot
np = numpy
plt = pyplot

from IPython.display import display
from IPython.core.pylabtools import figsize, getfigs

from pylab import *
from numpy import *
```

Fíjense cómo el comando `plot()` que hemos usado hasta ahora devuelve una lista de instancias de cada dibujo. Una *instancia* es una referencia a un elemento que creamos, en este caso la línea en gráfica. En este caso es una lista con un sólo elemento, una instancia `Line2D`. Podemos guardar esta instancia

para referirnos a este dibujo (a la línea en concreto) más adelante haciendo:

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

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

```
a = [3, 5]
# Así ``a`` es una lista, que contiene dos valores

a, b = [3, 5]
# Así desempaquetamos los elementos de la lista y a=3 y b=5
# Esto funciona porque pusimos tantas variables como elementos en la lista
```

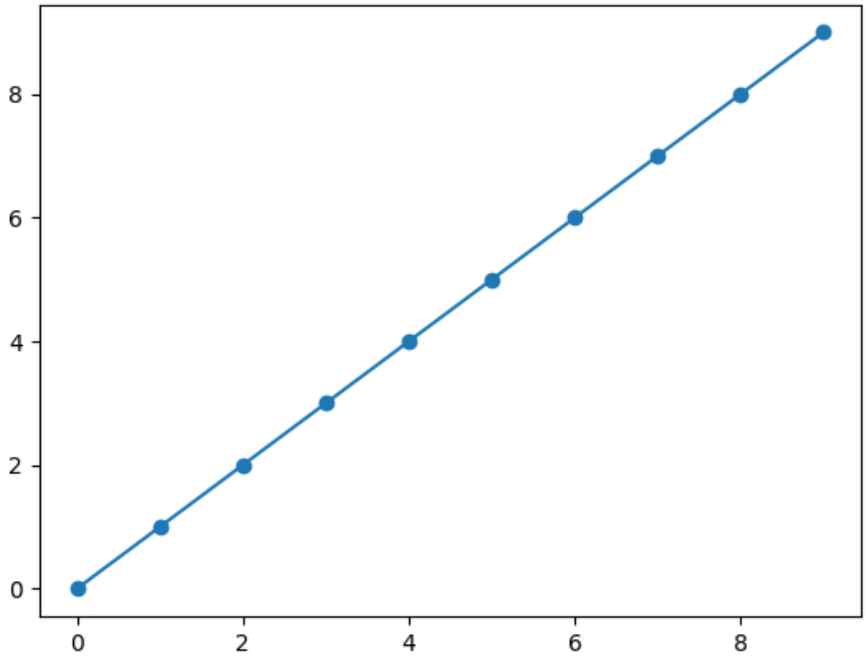
Pero si la lista sólo tiene un elemento ¿cómo desempaquetamos ese elemento?. Veamos:

```
a = [3]  # Así, ``a`` es una lista y no el número 3

a, = [3] # Si añadimos una coma indicamos que queremos meter ese único
         # elemento en una variable, en lugar de usar la lista
```

Y esto es justo lo que hicimos con `mi_dibujo, = plot(x)`, para hacer que `mi_dibujo` contenga una instancia y no una lista de instancias, que es lo que devuelve `plot()`.

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

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

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



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

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

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

Colores

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

Marcas y líneas

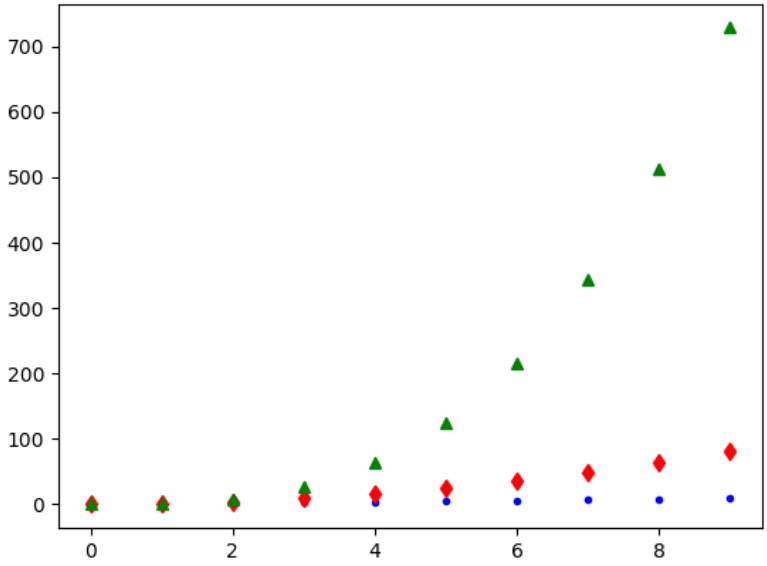
Símbolo	Descripción
'-'	Línea continua
'–'	Línea a trazos
'-.'	Línea a puntos y rayas
':'	Línea punteada
'.'	Símbolo punto
','	Símbolo pixel
'o'	Símbolo círculo relleno
'v'	Símbolo triángulo hacia abajo
'^'	Símbolo triángulo hacia arriba
'<'	Símbolo triángulo hacia la izquierda
'>'	Símbolo triángulo hacia la derecha
's'	Símbolo cuadrado
'p'	Símbolo pentágono
'*'	Símbolo estrella
'+'	Símbolo cruz
'x'	Símbolo X
'D'	Símbolo diamante
'd'	Símbolo diamante delgado

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

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

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

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

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

In [11]: # dibujamos tres curvas en el mismo gráfico y figura
In [12]: plot(x, x,'b.', x, x2,'rd', x, x3,'g^')
Out[13]:
[<matplotlib.lines.Line2D object at 0x8e959cc>,
<matplotlib.lines.Line2D object at 0x8eb75cc>,
<matplotlib.lines.Line2D object at 0x8eb788c>]
```

Esta lista de salida de `plot()` contiene 3 instancias que se refieren a 3 elementos diferentes de la gráfica.



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

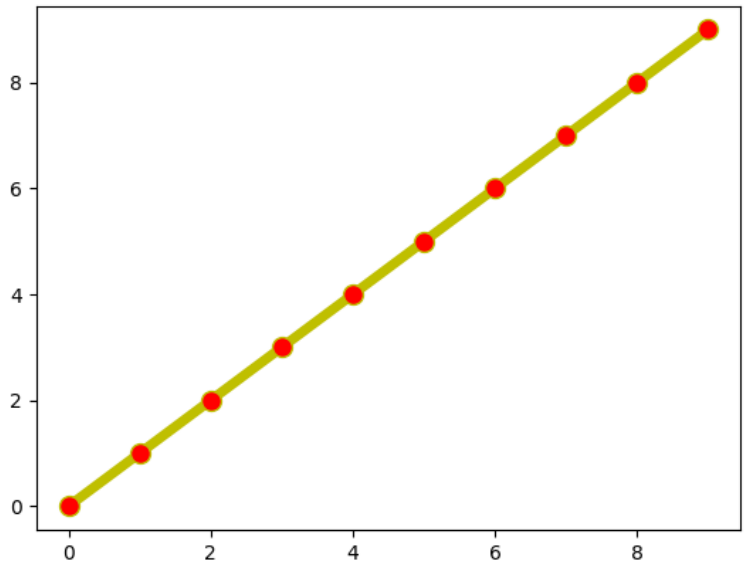
```
In [12]: xlim(-1,11)      # nuevos límites para el eje OX
Out[12]: (-1, 11)

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

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

Parámetro	Significado y valores
alpha	grado de transparencia, float (0.0=transparente a 1.0=opaco)
color o c	Color de matplotlib
label	Etiqueta con cadena de texto, string
markeredgecolor o mec	Color del borde del símbolo
markeredgewidth o mew	Ancho del borde del símbolo, float (en número de puntos)
markerfacecolor o mfc	Color del símbolo
markersize o ms	Tamaño del símbolo, float (en número de puntos)
linestyle o ls	Tipo de línea, '-' '–' '-.' ':' 'None'
linewidth o lw	Ancho de la línea, float (en número de puntos)
marker	Tipo de símbolo,'+' '*' ',' '.' '1' '2' '3' '4' '<' '>' 'D' 'H' '^' '_' 'd' 'h' 'o' 'p' 's' 'v' 'x' 'l' TICKUP TICKDOWN TICKLEFT TICKRIGHT

Un ejemplo usando más opciones sería este:

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



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

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

>>> show()                  # Muestro en dibujo por pantalla
>>> p1.set_marker('o')      # Cambio el símbolo de la gráfica 1
>>> p3.set_color('y')       # Cambio el color de la gráfica 3
>>> draw()                  # Hacer los cambios
```

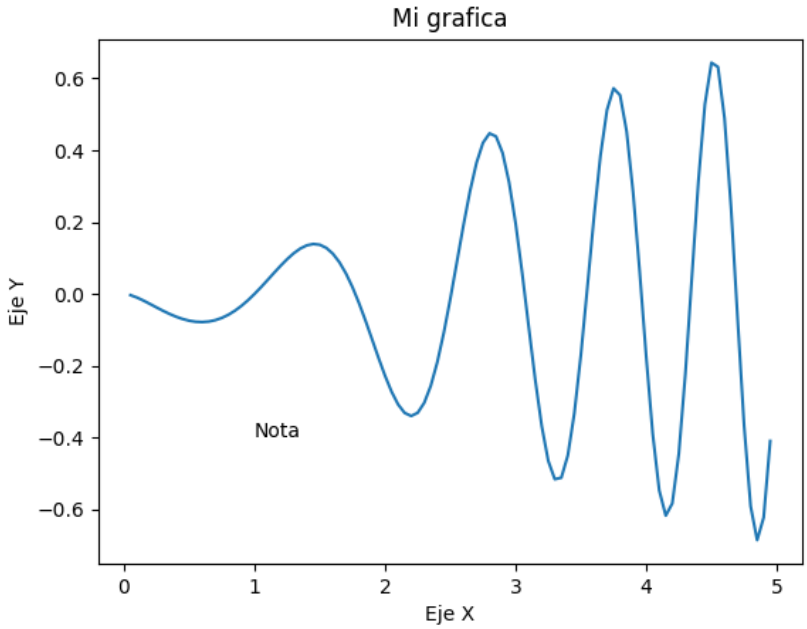
usando instancias similares poder cambiar prácticamente todas las propiedades de nuestro gráfico sin tener que rehacerlo. Por tanto es buena costumbre guardar las instancias en variables cuando trabajemos interactivamente.

7.1 Trabajando con texto dentro del gráfico

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

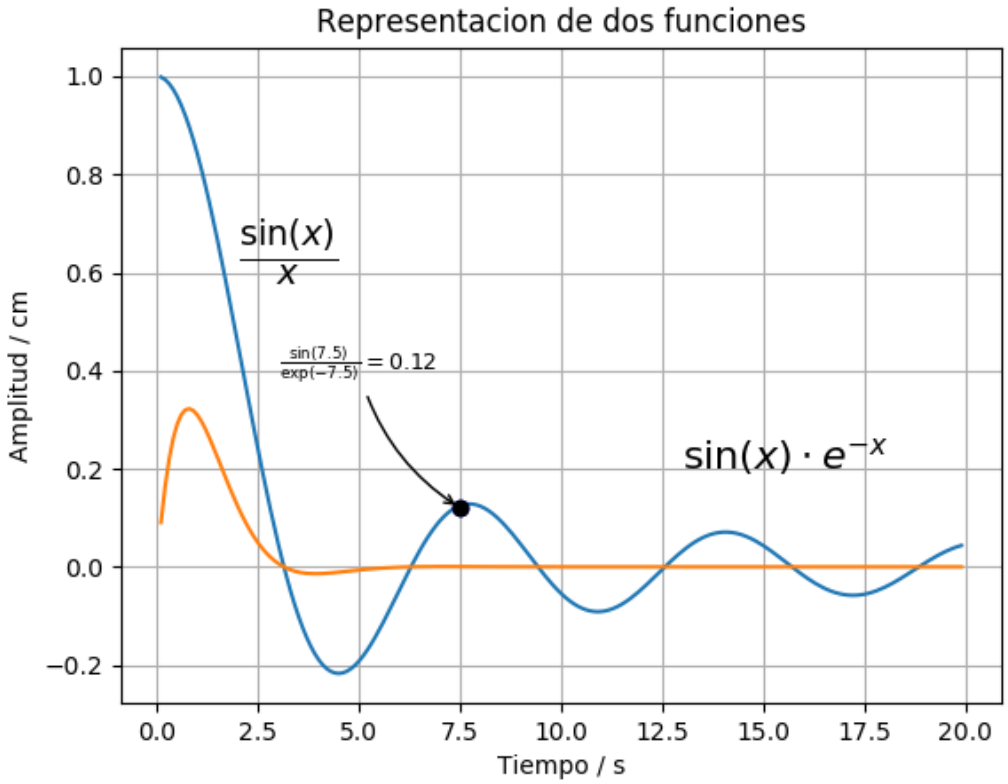
```
In [9]: x = arange(0, 5, 0.05)

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

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

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

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

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



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

```
from math import *
from numpy import *

t = arange(0.1, 20, 0.1)

y1 = sin(t)/t
y2 = sin(t)*exp(-t)
p1, p2 = plot(t, y1, t, y2)

# Texto en la gráfica en coordenadas (x,y)
texto1 = text(2, 0.6, r'$\frac{\sin(x)}{x}$', fontsize=20)
texto2 = text(13, 0.2, r'$\sin(x) \cdot e^{-x}$', fontsize=16)

# Añado una malla al gráfico
grid()

title('Representacion de dos funciones')
xlabel('Tiempo / s')
ylabel('Amplitud / cm')

show()
```



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

También podemos hacer señalizaciones de partes de la gráfica usando flechas, para lo que hay que indicar las coordenadas del punto a señalar, a donde apuntará la flecha y las coordenadas de texto asociado:

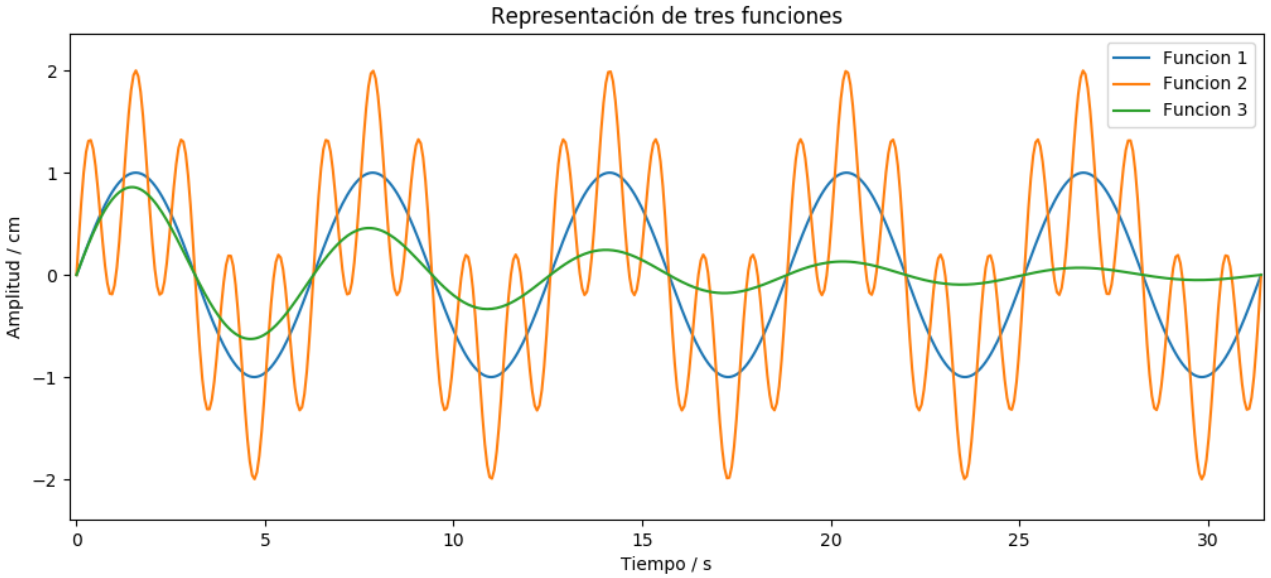
```
# Punto a señalar en la primera gráfica
px = 7.5
py = sin(py)/py

# Pinto las coordenadas con un punto negro
punto = plot([px], [py], 'bo')

# Hago un señalización con flecha
nota = plt.annotate(r'$\frac{\sin(7.5)}{\exp(-7.5)} = 0.12$',
         xy=(px, py), xycoords='data',
         xytext=(3, 0.4), fontsize=9,
         arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
```

Nota: LaTeX es un sistema de escritura orientado a contenidos matemáticos muy popular en ciencia e ingeniería. Puedes ver una buena introducción a LaTeX en esta dirección (pdf): http://www.ctan.org/tex-archive/info/lshort/spanish.

7.2 Representación gráfica de funciones

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

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

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


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

# array de valores que quiero representar
x = linspace(0, 10*pi, 800)

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

# Añado leyenda, tamaño de letra 10, en esquina superior derecha
legend(('Funcion 1', 'Funcion 2', 'Funcion 3'),
prop = {'size':10}, loc = 'upper right')

xlabel('Tiempo / s')
ylabel('Amplitud / cm')
title('Representacion de tres funciones')

# Creo una figura (ventana), pero indico el tamaño (x,y) en pulgadas
figure(figsize=(12, 5))

show()
```



Nótese que hemos añadido una leyenda con la función `legend()` que admite como entrada una **tupla** con *strings* que corresponden consecutivamente a cada una de las curvas del gráfico. Alternativamente, se puede usar el parámetro `label` en cada `plot()` para identificar la gráfica y luego usar `legend()` sin parámetros, que usará la `label` indicada en cada gráfica como etiqueta.

```
# Plots con label
p1 = plot(x, f1(x), label='Funcion 1')
p2 = plot(x, f2(x), label='Funcion 2')
p3 = plot(x, f3(x), label='Funcion 3')

# Ahora se puede usar legend sin etiqueta, pero indico
# dónde quiero que se coloque
legend(loc='lower right')
```

7.3 Histogramas

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

```
# Importo el módulo de numeros aleatorios de numpy
from numpy import random

# utilizo la función randn() del modulo random para generar
# un array de números aleatorios con distribución normal
nums = random.randn(200)  # array con 200 números aleatorios

# Genero el histograma
h = hist(nums)
```

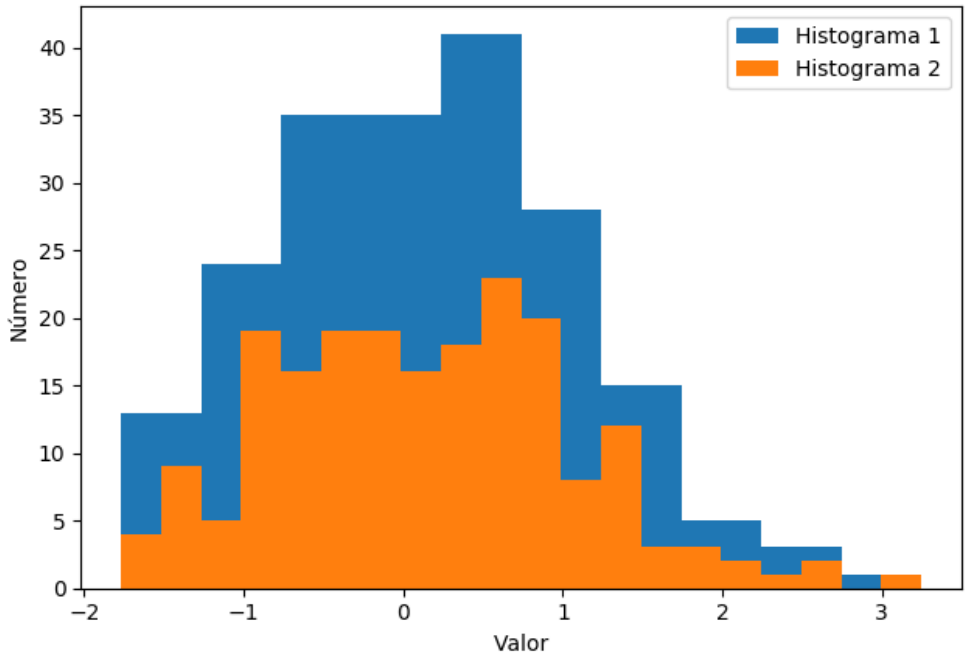
```
"""
(array([ 2, 10, 11, 28, 40, 49, 37, 12,  6,  5]),
array([-2.98768497, -2.41750815, -1.84733134, -1.27715452, -0.70697771,
    -0.13680089,  0.43337593,  1.00355274,  1.57372956,  2.14390637,
    2.71408319]),
<a list of 10 Patch objects>)
"""
```

Si no se le proporciona ningún otro argumento a `randn()` produce *floats* alrededor de 0 y con una varianza = 1.

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

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

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



7.4 Varias ventanas de gráficos

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

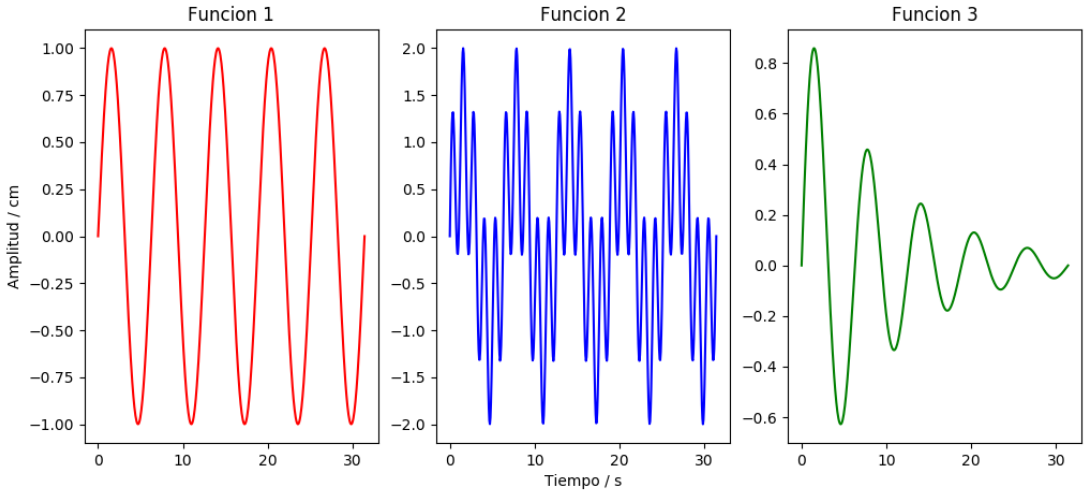
activa, la figura 2. Podemos regresar a la primera escribiendo `figure(1)` para trabajar nuevamente en ella, por ejemplo:

```
p1, = plot(sin(x))          # Crea una figura en una ventana (Figure 1)

figure(2)                   # Crea una nueva figura vacía en otra ventana␣
↪(Figure 2)
p2, = plot(cos(x))          # Dibuja el gráfico en la figura 2
title('Funcion coseno')     # Añade un título a la figura 2

figure(1)                   # Activo la figura 1
title('Funcion seno')       # Añade un título a la figura 2
```

7.5 Varios gráficos en una misma figura

En ocasiones nos interesa mostrar varios gráficos diferentes en una misma figura o ventana. Para ello podemos usar la función `subplot()`, indicando entre paréntesis un número con tres dígitos. El primer dígito indica el número de filas en los que se dividirá la figura, el segundo el número de columnas y el tercero se refiere al gráfico con el que estamos trabajando en ese momento. Por ejemplo, si quisiéramos representar las tres funciones anteriores usando tres gráficas en la misma figura, una al lado de la otra y por lo tanto con una fila y tres columnas, haríamos lo siguiente:

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

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

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



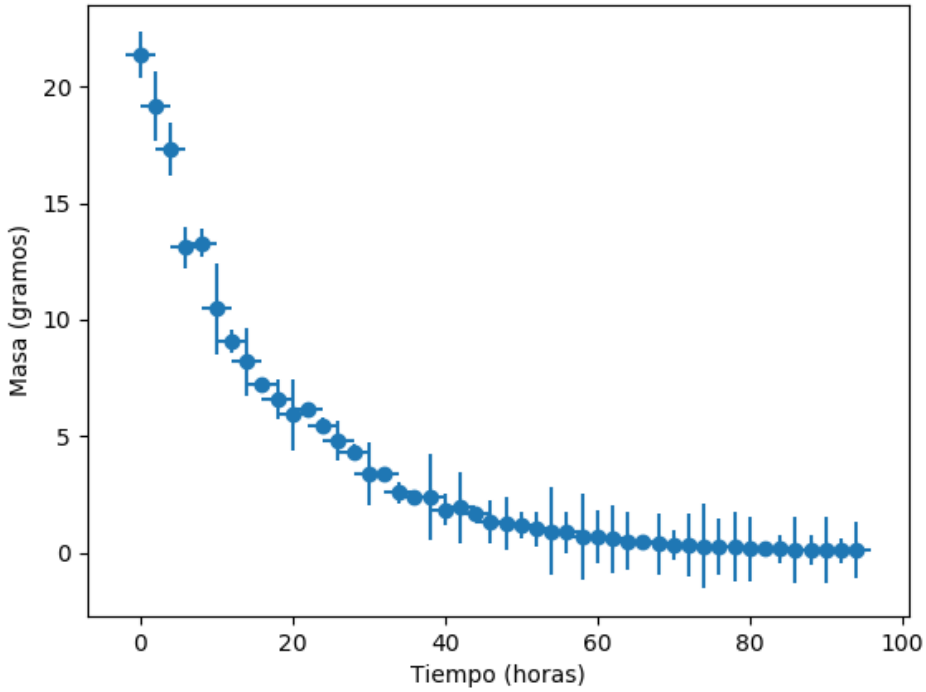
Al igual que con varias figuras, para dibujar en un gráfico hay que activarlo. De esta forma, si acabamos de dibujar el segundo gráfico escribiendo antes `subplot(132)` y queremos cambiar algo del primero, debemos activarlo con `subplot(131)` y en ese momento todas funciones de gráficas que hagamos se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

7.6 Datos experimentales con barras de error

Para representar barras error, `matplotlib` tiene una función específica alternativa a `plot()` que es `errorbar()`, que funciona de forma parecida pero no igual a `plot()`. La principal difirencia es que es obligatorio usar datos para `x` e `y` y como tercer parámetro se da el un float o array con el error en `y`, siendo opcional el error en `x`. Para dar el formato se debe usar el parámetro `fmt=""` como un string, que es similar al usado el `plot()`.

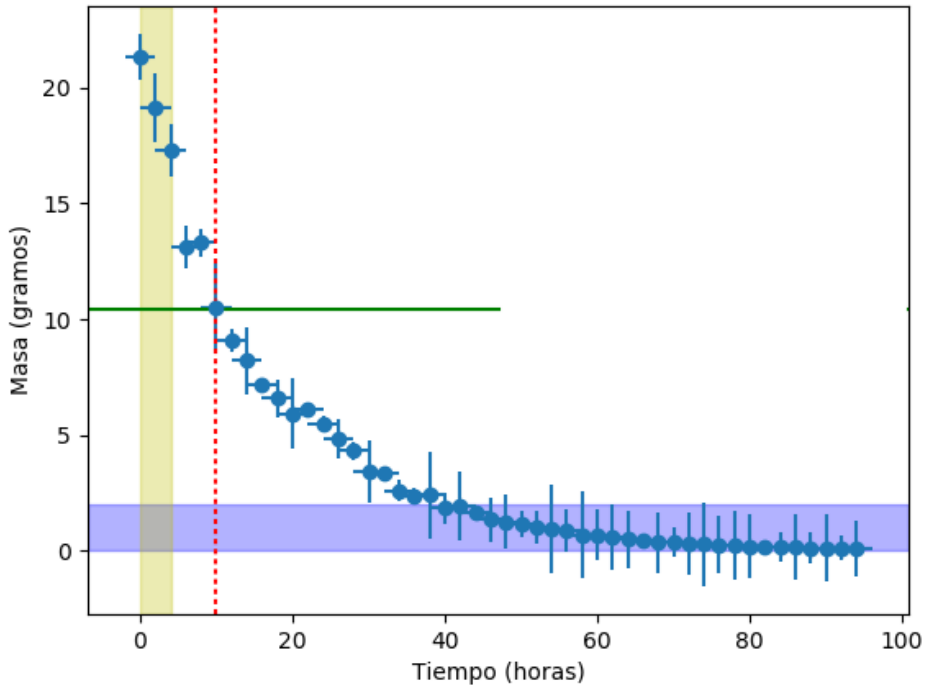
Probemos a representar unos datos de medidas de desintegración radioactiva que tienen datos de error:

```
# El fichero tiene tres columnas separadas por ";"
# Tiempo en horas, masa en gramos y error de la medida de la masa
tiempo, masa, error = loadtxt("medidas_radio2.txt", delimiter=";",␣
↪unpack=True)

# Doy un error fijo al eje x, que es opcional
xerror = 2.0

# Dibujo las medidas con errores, que pueden darse como un número
# fijo o un array, dibujando solo punto grandes ("o")
errorbar(tiempo, masa, yerr=error, xerr=xerror, fmt="o")

xlabel("Tiempo (horas)")
ylabel("Masa (gramos)")
savefig("grafica_desintegracion.png")
```



Podemos usar algunos elementos de dibujo para resaltar zonas del gráfico con línea o bandas horizontales y verticales. Las coordenadas se ponen siempre en coordenadas de la gráfica, pero se pueden poner límites opcionales que se indican como fracción del eje, siendo 0 el inicio del eje y 1 el final.

```
# Trazo una línea vertical en la coordenada x=10 color rojo (r)
# y con trazo punteado
axvline(10, color='r', ls="dotted")

# Línea horizontal en la coordenada y=10.4 color verde (g)
# que termina en la limitad de la gráfica (0.5, va de 0 a 1)
axhline(10.4, color='g', xmax=0.5)

# Banda horizontal de y=0 a y=2 de color azul
# y 30% de transparencia (alpha=0.3)
axhspan(0, 2, alpha=0.3, color='b')

# Banda vertical de x=0 a x=4 de color amarillo
# y 30% de transparencia
axvspan(0, 4, alpha=0.3, color='y')
```

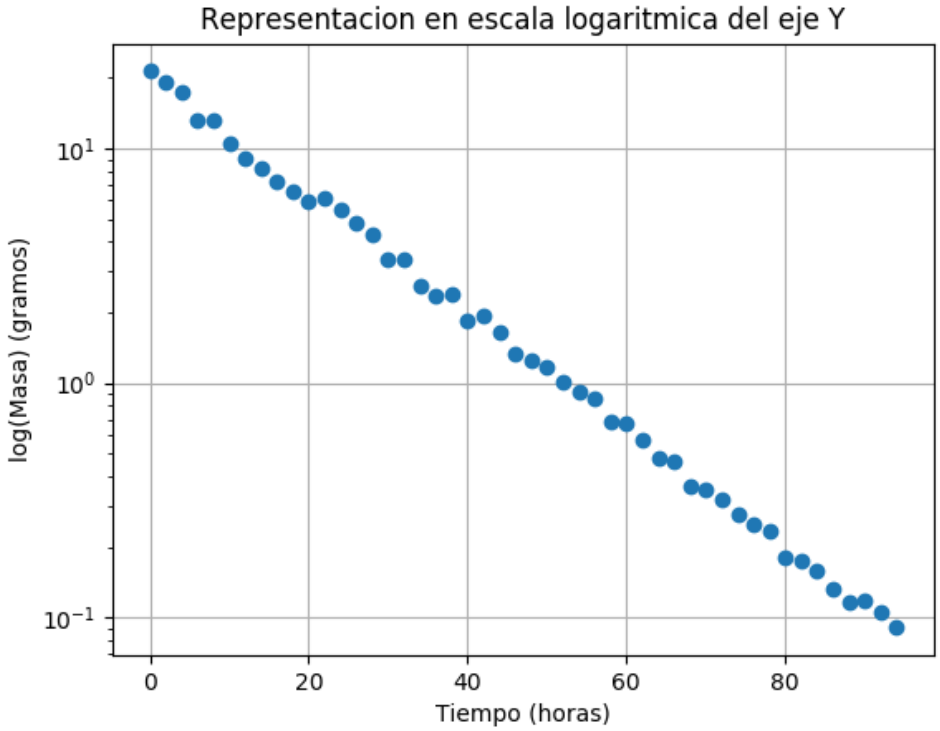


Fijémonos en el las marcas y etiquetas de los ejes, que se imprimieron automáticamente según los datos representados. Estos se pueden cambiar con las funciones `xticks()` y `yticks()`. Si se usan sin parámetros, simplemente nos devuelve una tupla de dos elementos: y array de índices de posición y una lista de etiquetas de texto. Pero si le datos como parámetro una tupla con posiciones y etiquetas será las que use para el dibujar el eje. En nuestra gráfica, el eje `y` va de 0 a 20 de cinco en cinco, pero supongamos que queremos poner más marcadores. Si le datos a `yticks()` un solo parametro de array, serán los valores que use como marcadores y etiquetas:

```
In[1]: yticks()  Marcadores y etiquetas de la gráfica acutual
Out[1]:
(array([  1.00000000e-03,   1.00000000e-02,   1.00000000e-01,
          1.00000000e+00,   1.00000000e+01,   1.00000000e+02,
          1.00000000e+03]), <a list of 7 Text yticklabel objects>)

In[2]: # Creo más marcardores en el eje Y, el indice y la etiqueta es la␣
↪misma
In[3]: yticks(range(0,20,2))


In[5]: # Marcadores en un array de numeros y etiquetas en un array de␣
↪strings
In[6]: marcadores = range(0, 30, 5)
In[7]: etiquetas = ["A", "B","C","D","E", "F"]
In[8]: yticks(marcadores, etiquetas)
```

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

```
# Eje y en escala logarítmica
p1, = semilogy(d[0], d[1], 'o')

grid()
xlabel("Tiempo (horas)")
ylabel("log(Masa) (gramos)")
title('Representacion en escala logaritmica del eje Y')
```

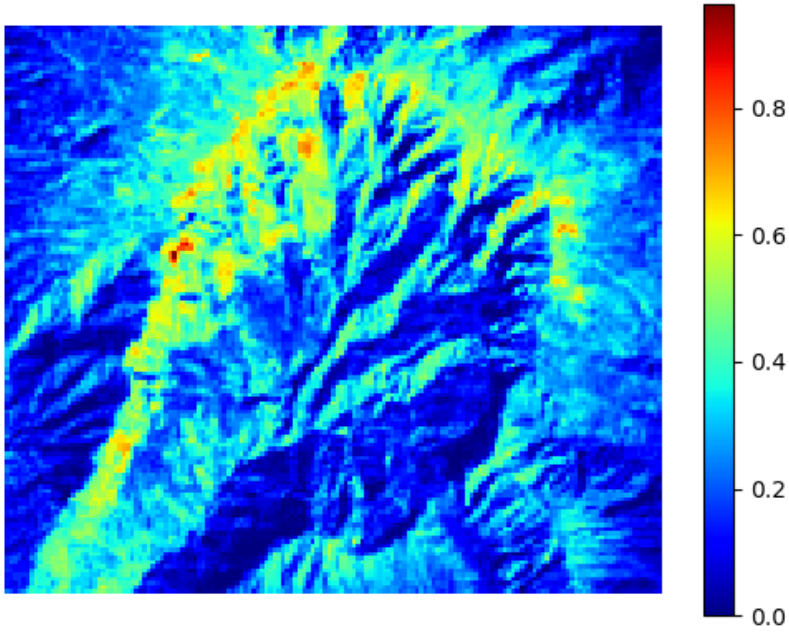


7.7 Representación de datos bidimensionales

Los datos bidimensionales son valores de una magnitud física representada por una función que tiene dos variables independientes, normalmente *x* e *y*; se trata pues de representar funciones del tipo *z=z(x,y)*, donde *z* puede representar flujo luminoso, presión atmosférica, altura del terreno, etc. Podemos usar la función `imshow()` para mostrar imágenes bidimensionales, que deben ser arrays 2D de numpy. La función `imread()` de `matplotlib` nos permite leer una imagen de bit en los formatos más comunes (jpg, png, svg), que lee datos multidimensionales a un array de tañano `MxNxR`, siendo M las filas, N las columnas y R la banda en caso de ser una imagen multicanal (a color RGB, por ejemplo).

```
# Leo la imagen png a array de numpy
img = imread("la_palma.png")

# imagen en color, tres bandas
imshow(img)

# Tomo la banda R
R = img[:, :, 0]
imshow(R, cmap=gray())
```

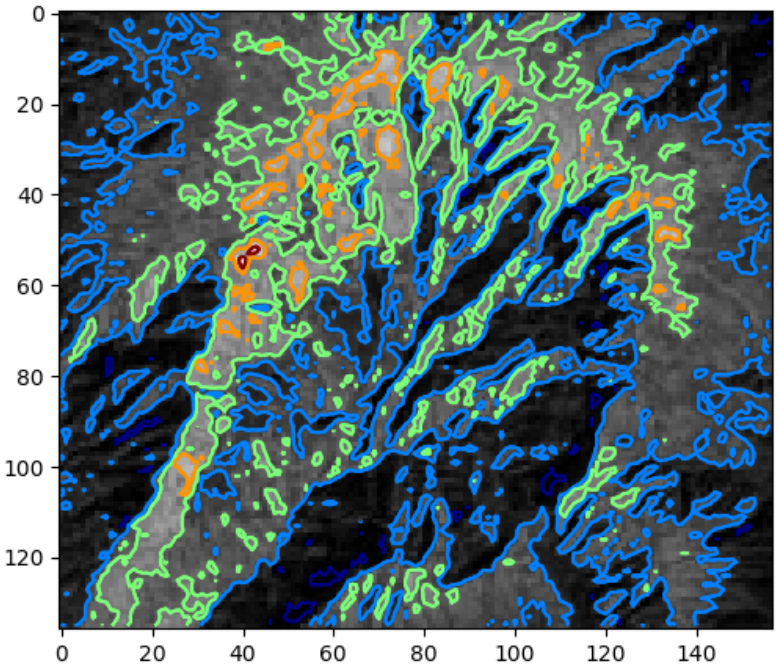
```
# Hacemos un zoom y escojemos la zona de interes
# xlim e ylim necesitan enteros
xlim0, xlim1 = array(xlim(), dtype=int)
ylim0, ylim1 = array(ylim(), dtype=int)

print(ylim0, ylim1)
# (328, 192)

# Hago una selección basada en los límites actuales de la grafica
# Atencion!: Hay que fijarse que Y es la primera dimension (primer eje) y␣
↪por
# eso va de primero y ademas el origen está arriba a la izquierda, por lo
# hay que poner poner primero el segundo limite ylim1
seleccion = R[ylim1:ylim0, xlim0:xlim1]

# Quito los ejes
axis('off')

# Muestro la grafica con barra de color
imshow(seleccion, cmap=jet())
cb = colorbar()
```



Sobre las imágenes 2D es posible crear contornos de niveles con `contour()`, dando los niveles como un array de números con los niveles deseados:

```
# Limpio la figura
clf()

# Muestro la imagen en gris y creo contornos
# con mapa de color jet()
```

```
imshow(seleccion, cmap=gray())
contour(seleccion, levels=arange(0,1,0.2), color=jet())
```



7.8 Guardando las figuras

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

```
savefig("mi_primera_grafica.eps")           # Guardo la figura en formato␣
↪eps
savefig("mi_primera_grafica.png", dpi=300)  # Guardo la figura en formato␣
↪png a 300 DPI
```

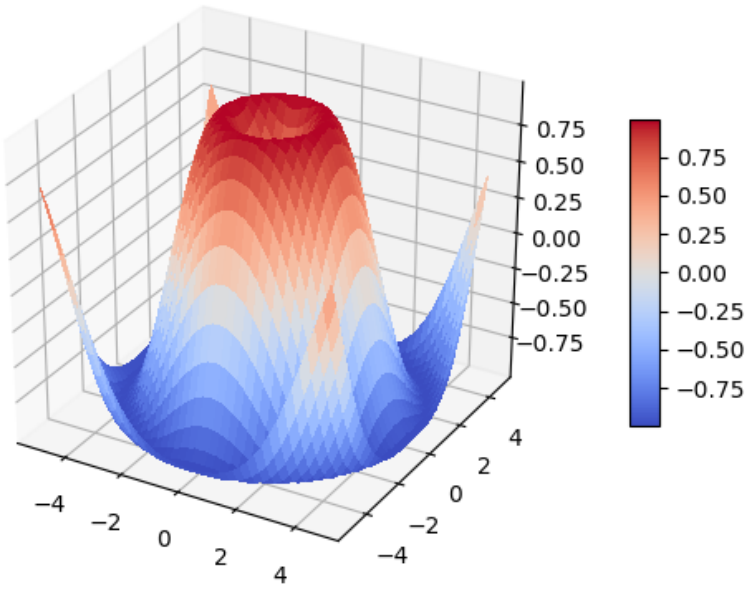
Si el gráfico se va usar para imprimir, por ejemplo en una publicación científica o en un informe, es recomendable usar un formato vectorial como Postscript (ps) o Postscript encapsulado (eps), pero si es para mostrar por pantalla o en una web, el más adecuado es un formato de mapa de bits como png o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

7.9 Gráficos 3D

Aunque matplotlib está especializado en gráficos 2D, incluye un **toolkit** para hacer gráficos 3D de muchos tipos usando OpenGL, que nos resolverá casi todas las necesidades para gráficos de este tipo.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D, get_test_data
```

```
from matplotlib import cm

# Figura
fig = plt.figure()

# Tomo el eje actual y defino una proyección 3D
ax = gca(projection='3d')

# Dibujo 3D
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)

# el metodo meshgrid devuelve una matriz de coordenadas
# a partir de vectores de coordendas, que usamos para
# los datos del eje Z
X, Y = np.meshgrid(X, Y)

R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Grafico surface en 3D
surface = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,␣
↪linewidth=0, antialiased=False)

# Límites del eje Z
ax.set_zlim(-1.01, 1.01)

# Barra de nivel, un poco más pequeña
fig.colorbar(surf, shrink=0.5, aspect=10)
```



Si queremos hacer lo mismo con una imagen, debemos crear igualmente las matrices de coordenadas X e Y para cada pixel de la imagen. Podemos usar `mgrid()` que usa índices:

```
xx, yy = np.mgrid[0:seleccion.shape[0], 0:seleccion.shape[1]]

ax.plot_surface(xx, yy, seleccion ,rstride=1, cstride=1, cmap=plt.cm.gray,␣
↪linewidth=0)
```

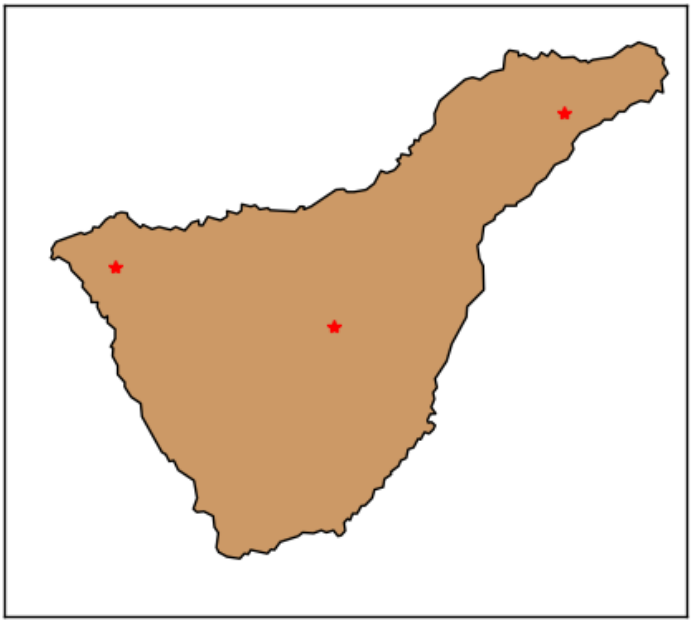
7.10 Mapas geográficos con Basemap

Se pueden crear mapas geográficos con `matplotlib` y representar datos sobre ellos usando la extensión (*toolkit*) basemap, que no viene por defecto con `matplotlib` por lo que hay que instalar previamente.

```
pip3 install basemap      # Instalación de Python estandar
conda install basemap     # Con Anaconda-Python
```

Para hacer un mapa hay que crear un instancia de mapa con coordenadas de centro, proyección, resolución y márgenes. Luego podemos crear un lista de coordenadas de latitud y longitud en grados y transformarlas a coordendas del mapa para poder representarla con cualquier función de `pyplot()`, generalmente `plot()` o `scatter()`.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

# Coordendas de Tenerife, en grados
lat, lon = 28.2419, -16.5937

# Mapa en alta resolución (h) con proyección mercator
map = Basemap(projection='merc', lat_0 = lat, lon_0 = lon,
    resolution = 'h', area_thresh = 0.1,
    llcrnrlon=lon-0.5, llcrnrlat=lat-0.5,
    urcrnrlon=lon+0.5, urcrnrlat=lat+0.5)

# Líneas de costas y paises, aunque en este ejemplo no se ven
map.drawcoastlines()
map.drawcountries()

map.fillcontinents(color = '#cc9966')
map.drawmapboundary()

# Coordenadas para dibujar en el map
lats = [28.5068, 28.2617, 28.3310]
lons = [-16.2531, -16.5526, -16.8353]

x,y = map(lons, lats)
map.plot(x, y, '*r', markersize=6)

plt.show()
```



7.11 Ejercicios

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

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

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

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

3. Con la serie de Gregory-Leibnitz para el cálculo de π usada anteriormente en el problema 5.5:

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

el valor obtenido de π se acerca lentamenta al verdadero con cada término que se añada. Calculen todos los valores que va tomando π con cada término añadido y representar en una figura con dos gráficas (usando `subplot()`) los primeros 300 valores que toma π frente al número de términos usados en una de ellas y el valor absoluto de la diferencia entre el valor calculado y el real frente al número de elementos en la otra.

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

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

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

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

CAPÍTULO 8

La librería científica Scipy

Scipy es el paquete científico (es decir, un módulo que tiene otros módulos) más completo, que incluye interfases a librerías científicas muy conocidas como LAPACK, BLAS u ODR entre muchas otras . Si importamos `scipy` para tener su espacio de nombres y consultamos la ayuda, vemos todos los módulos que posee:

```
In [1]: import scipy
In [2]: help(scipy)
```

```
cluster                      --- Vector Quantization / Kmeans
fftpack                      --- Discrete Fourier Transform algorithms
integrate                    --- Integration routines
interpolate                  --- Interpolation Tools
io                           --- Data input and output
lib                          --- Python wrappers to external libraries
lib.lapack                   --- Wrappers to LAPACK library
linalg                       --- Linear algebra routines
misc                         --- Various utilities that don't have
                                 another home.
ndimage                      --- n-dimensional image package
odr                          --- Orthogonal Distance Regression
optimize                     --- Optimization Tools
signal                       --- Signal Processing Tools
sparse                       --- Sparse Matrices
sparse.linalg                --- Sparse Linear Algebra
sparse.linalg.dsolve         --- Linear Solvers
sparse.linalg.dsolve.umfpack --- :Interface to the UMFPACK library:
                                 Conjugate Gradient Method (LOBPCG)
sparse.linalg.eigen.lobpcg   --- Locally Optimal Block Preconditioned
                                 Conjugate Gradient Method (LOBPCG) [*]
special                      --- Airy Functions [*]
lib.blas                     --- Wrappers to BLAS library [*]
sparse.linalg.eigen          --- Sparse Eigenvalue Solvers [*]
stats                        --- Statistical Functions [*]
lib                          --- Python wrappers to external libraries
                                 [*]
```

```
lib.lapack                  --- Wrappers to LAPACK library [*]
integrate                   --- Integration routines [*]
ndimage                     --- n-dimensional image package [*]
linalg                      --- Linear algebra routines [*]
spatial                     --- Spatial data structures and algorithms
special                     --- Airy Functions
stats                       --- Statistical Functions
```

8.1 Ajustes lineales y no lineales

Si simplemente necesitamos hacer ajustes básicos de polinomios, lo podemos hacer fácilmente sólo con numpy:

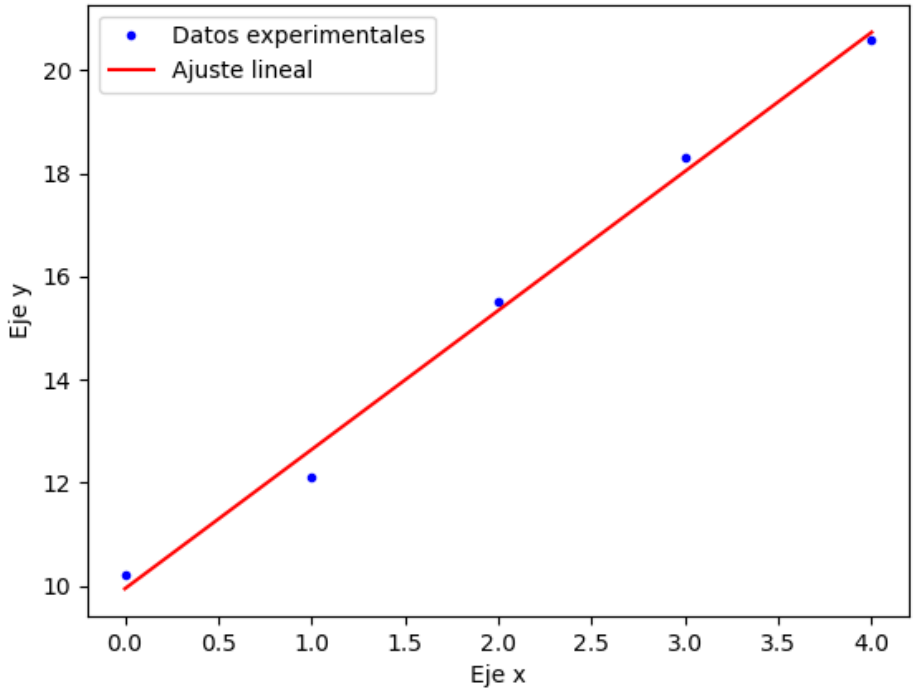
```
# Importo todas las funciones de numpy, si no lo he hecho
from numpy import *

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

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

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

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

$y(x) = p_0 x + p_1 = 2.7x + 9.94$

Ahora podemos dibujar los datos experimentales y la recta ajustada:

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

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

title('Ajuste lineal por minimos cuadrados')

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

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



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

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

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

8.2 Ajuste de funciones generales

Si queremos hacer un ajuste general, no necesariamente polinómico, debemos usar alguno de los métodos del paquete `optimize`, que contiene varios optimizadores locales y globales. El más común es `leastsq` que, al ser un optimizador, hay que definir previamente una **función residuo** que es la que realmente se va minimizar.

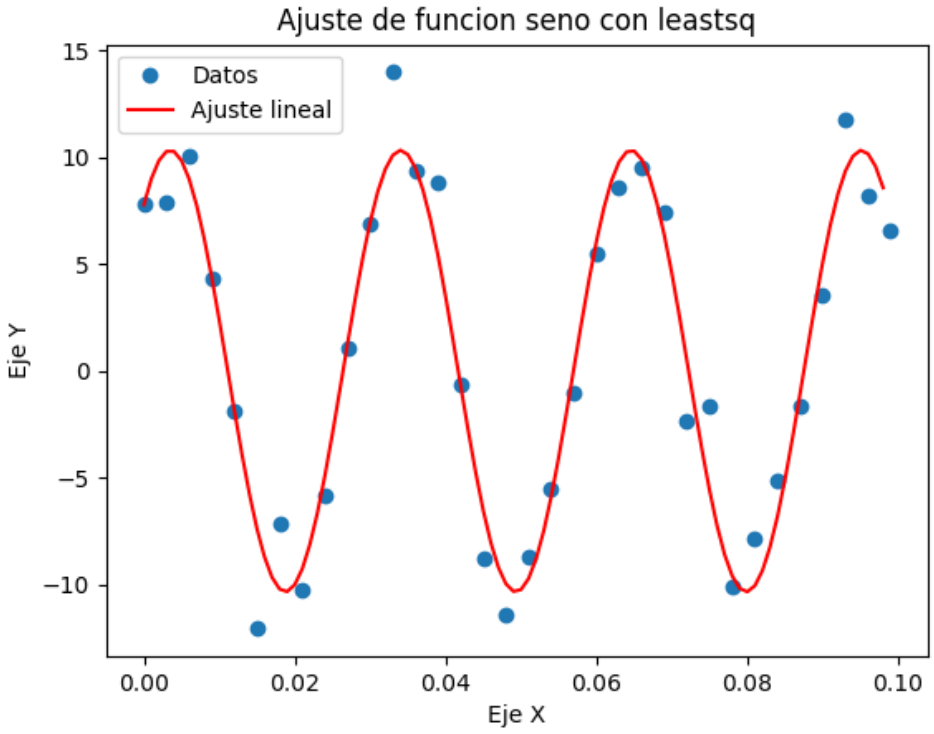
```
from pylab import *
from scipy.optimize import leastsq

# Datos de laboratorio
```

```
datos_y = array([ 2.9, 6.1, 10.9, 12.8, 19.2])
datos_x = array([ 1.0, 2.0, 3.0, 4.0, 5.0])

# Función para calcular los residuos, donde
# se calcula (datos - modelo)
def residuos(p, y, x):
    error  = y - (p[0]*x + p[1])
    return error

# Parámetros iniciales estimados
# y = p0[0]*x  + p0[0]

p0 = [2.0, 0.0]

# Hacemos  el ajuste por minimos cuadrados con leastsq(). El primer␣
↪parámetro
# es la funcion de residuos, luego los parámetro iniciales y una tupla con␣
↪los
# argumentos de la funcion de residuos, en este caso, datos_y y datos_x en
# ese orden, porque así se definió la función de error
ajuste = leastsq(residuos, p0, args=(datos_y, datos_x))

# El resultado es una lista, cuyo primer elemento es otra
# lista con los parámetros del ajuste
print(ajuste[0])
# array([ 3.93, -1.41])
```

Veamos otro ejemplo para ajustar una función seno:

```
from pylab import *
from scipy.optimize import leastsq
from scipy import random

# Generamos unos datos artificiales para hacer el ejemplo
# A datos_y se le añade "ruido" que simula error de
# medida, añadiendole un valor aleatorio
datos_x = arange(0, 0.1, 0.003)
A, k, theta = 10.0, 33.3, pi/5.0
y_real = A*sin(2*pi*k*datos_x + theta)
datos_y = y_real + 2*random.randn(len(datos_x))

# Ahora se trata de ajustar estos datos una función
# modelo tipo senoidal A*sin(2*pi*k*x+theta)

# Defino la funcion de residuos
def residuos(p, y, x):
    A, k, theta = p
    error = y - A*sin(2*pi*k*x + theta)
    return error

# Parámetros iniciales
# y = p[0]*sin(2*pi*p[1]*x + p[2])
# Si estos se alejan mucho del valor real
# la solución no convergerá
p0 = [8.0, 40.0, pi/3]

# hacemos  el ajuste por minimos cuadrados
```

```
ajuste = leastsq(residuos, p0, args=(datos_y, datos_x))

# El resultado es una lista, cuyo primer elemento es otra
# lista con los parámetros del ajuste.
print(ajuste[0])
# array([ -9.787095  ,  32.91201348,  -2.3390355 ]

# Ahora muestro los datos y el ajuste gráficamente

plot(datos_x, datos_y, 'o')  # datos

# Defino la funcion modelo, para representarla gráficamente
def funcion(x, p):
    return p[0]*sin(2*pi*p[1]*x + p[2])

# genero datos a partir del modelo para representarlo
x1 = arange(0, datos_x.max(), 0.001)  # array con muchos puntos de x
y1 = funcion(x1, ajuste[0])           # valor de la funcion modelo en los x

plot(x1, y1, 'r-')
xlabel('Eje X')
ylabel('Eje Y')
title('Ajuste de funcion seno con leastsq')
legend(('Datos', 'Ajuste lineal'))
show()
```



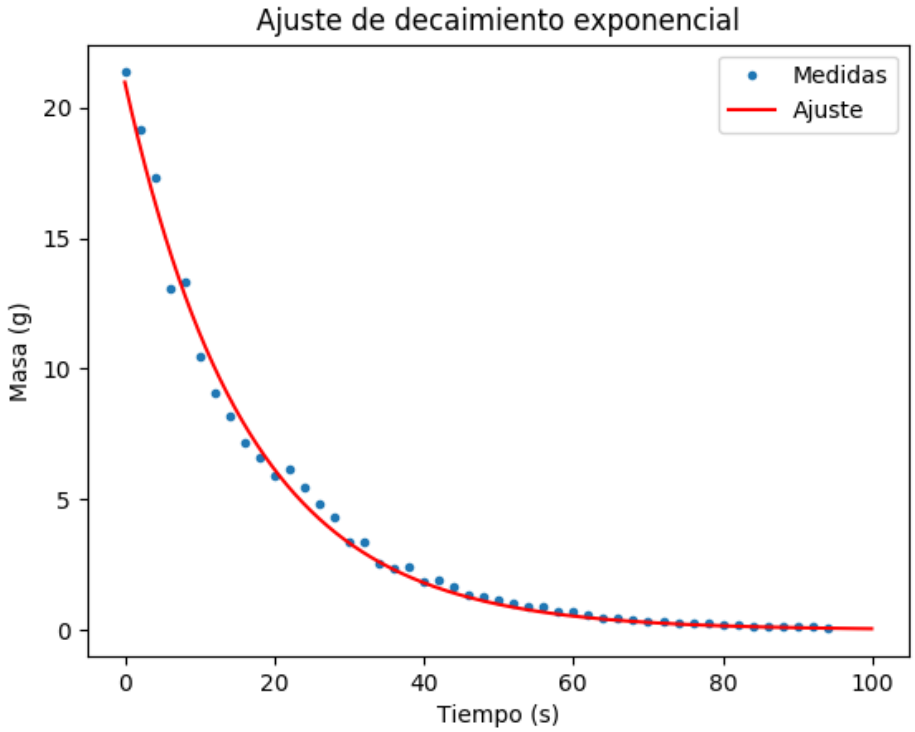
Este ejemplo es bastante alaborado porque hemos usado un optimizador general para hacer un ajuste, pero podemos usar `curve_fit()` para ahorrarnos la función residuo.

La anterior es una manera artesanal de hacer el ajuste, al construir la función de error. Para un ajuste de datos

```
from scipy.optimize import curve_fit

curve_fit(funcion, datos_x, datos_y, p0)

(array([ -9.787095  ,  32.91201348,  -2.3390355 ]),
 array([[ 0.20148401, -0.00715614,  0.00215931],
        [-0.00715614,  0.07184634, -0.02241144],
        [ 0.00215931, -0.02241144,  0.00925902]]))
```

Vamo a probar esto con una ley de decaimiento exponencial:

```
# Cargo los datos experimentales a un array
# Los datos están delimitados por ";" y uso unpack
# para que ponga primero las columas y pueda desenpaquetarlas
# en  variables

tiempo, masa, error = loadtxt("medidas_radio.txt", delimiter=";",␣
↪unpack=True)

# Compruebo como ser ven los datos
plot(tiempo, masa, '.')

def decaimiento(x, a, b):
    """Ley de decaimiento exponencial"""
    return a * exp(-b*x)

# Ajuste de los datos
# curve_fit devuelve dos variables, los parámetros del ajuste y
# la matriz de covarianza
popt, pcov = curve_fit(decaimiento, tiempo, masa)

# Ahora creo una curva teórica a partir del modelo ajustado
times = np.arange(0, 100, 0.1)
model = decaimiento(times, *popt)

plot(times, model, '-r')

legend(('Medidas', 'Ajuste'))
xlabel("Tiempo (s)")
ylabel("Masa (g)")
title("Ajuste de decaimiento exponencial")

# Guardo la grafica
savefig("ajuste.png")

# Para mostrar la gráfica por pantalla
show()
```



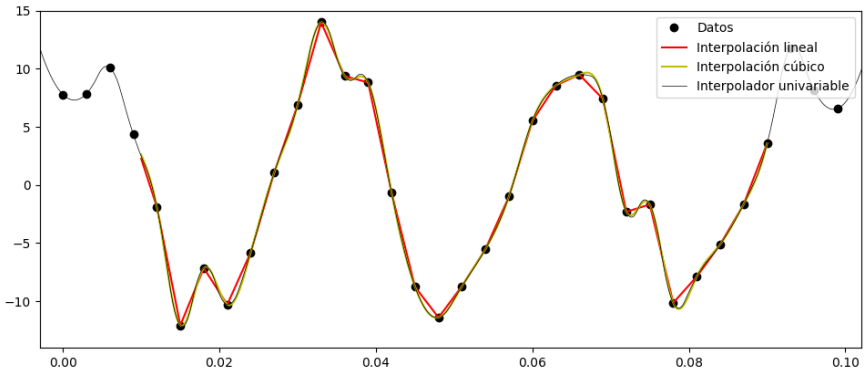
8.3 Interpolación de datos

El módulo `interpolate` contiene rutinas de interpolación basadas en la conocida librería FITPACK en Fortran; resulta muy útil en partes de datos donde no hay suficientes medidas. Los interpoladores funcionan creando una función interpoladora de orden predefinido, usando los datos de medidas. Luego se aplica esta función de interpolación a un array de datos más denso que la muestre. Consideremos los datos senoidales que vimos antes:

```
from scipy.interpolate import interp1d

interpolador_lineal = interp1d(datos_x, datos_y)
interpolador_cubico = interp1d(datos_x, datos_y, kind='cubic')

x_inter = linspace(0.01, 0.09, 500)
y_inter_l = interpolador_lineal(x_inter)
y_inter_c = interpolador_cubico(x_inter)

plot(datos_x, datos_y, 'ok', label="Datos")
plot(x_inter, y_inter_l, 'r', label="Interpolación lineal")
plot(x_inter, y_inter_c, 'y', label="Interpolación cúbico")

# Si usamos InterpolatedUnivariateSpline podemos interpolar fuera
# del rango de los datos
from scipy.interpolate import InterpolatedUnivariateSpline

# Array de valores más denso que los datos originales, pero
# dentro del rango de los datos
x_inter = linspace(-0.01, 0.11, 500)
```

```
interpolador = InterpolatedUnivariateSpline(datos_x, datos_y, k=2)
y_inter_u = interpolador(x_inter)

plot(x_inter, y_inter_u, 'k', lw=0.5, label="Interpolador univariable")

xlim(-0.003, 0.102)
ylim(-14, 15)
legend()
```

Aunque este es el método más usado para interpolar, tiene el problema que no permite interpolar puntos fuera del rango de las medidas. Esto se puede resolver con la función `InterpolatedUnivariateSpline()`



8.4 Integración numérica

Intentemos integrar numéricamente la integral:

$\int_{-2}^{2} 2 * \exp(\frac{-x^2}{5})\ dx$

```
def func1(x):
    return 2.0*exp(-x**2/5.0)

# integración entre -2 y +2
int1, err1 = integrate.quad(func1, -2, +2)
print(int1,err1)
(6.294530963693763, 6.9883332051087914e-14)
```

Si ahora hacemos la misma integral pero desde $-\infty$ hasta $+\infty$ obtendremos toda el área bajo la curva definida en el integrando:

```
# integración entre -infinito y +infinito
In [10]: int2, err2 = integrate.quad(func1, -Inf, +Inf)
# y el resultado obtenido es:
In [11]: print(int1, err1)
(7.9266545952120211, 7.5246691415403668e-09)
```

8.5 Manipulación de arrays 2D: imágenes

El módulo `ndimage` ofrece manipulación y filtrado básicos de imágenes, dados como array de numpy.

```
In [20]: from scipy import misc
In [21]: from scipy import ndimage as nd

In [22]: img = misc.face(gray=True)
In [23]: shifted_img = nd.shift(img, (50, 50))
In [24]: shifted_img2 = nd.shift(img, (50, 50), mode='nearest')
In [25]: rotated_img = nd.rotate(img, 30)

In [26]: cropped_img = img[50:-50, 50:-50]
In [27]: zoomed_img = nd.zoom(img, 2)  # Interpola la imagen

In [28]: zoomed_img.shape
# (1536, 2048)
```

```
In [30]: noisy_img = np.copy(img).astype(np.float)

In [31]: noisy_img += img.std()*0.5*np.random.standard_normal(img.shape)

In [32]: blurred_img = nd.gaussian_filter(noisy_img, sigma=3)
In [33]: median_img = nd.median_filter(blurred_img, size=5)

In [34]: from scipy import signal
In [35]: wiener_img = signal.wiener(blurred_img, (5,5))

In [40]: subplot(221)
In [41]: imshow(img)

In [42]: subplot(222)
In [43]: imshow(blurred_img)

In [44]: subplot(223)
In [45]: imshow(median_img)

In [46]: subplot(224)
In [47]: imshow(wiener_img)
```

Para más utilidades de manipulación de imágenes conviene usar http://scikit-image.org o http://opencv.org

8.6 Módulo de constantes físicas

Scipy contiene un prático módulo de constantes físicas.

```
In [1]: from scipy import constants as C

In [2]: C.c  # Velocidad de la luz en m/s
Out[2]: 299792458.0

In [4]: C.e  # Carga del electrón
Out[4]: 1.602176565e-19

In [6]: C.atmosphere
Out[7]: 101325.0

In [7]: C.mmHg
Out[7]: 133.32236842105263

In [8]: C.Julian_year
Out[8]: 31557600.0

In [9]: C.Avogadro
Out[9]: 6.02214129e+23

In [10]: C.parsec
Out[10]: 3.085677581305729e+16
```

```
In [11]: C.Stefan_Boltzmann
Out[11]: 5.670373e-08

In [13]: C.convert_temperature(np.array([0, 100.0]), 'Celsius', 'Kelvin')
 ↪# Conversor temps
Out[13]: array([ 273.15,  373.15])

In [14]: C.day  # Dia en segundos
Out[14]: 86400.0

In [15]: C.pico  # Prefijos del SI
Out[15]: 1e-12

In [16]: C.oz
Out[16]: 0.028349523124999998

In [17]: Constantes físicas de CODATA 2014
In [18]: C.find('atm')
Out[18]: ['standard atmosphere']
In [19]: C.physical_constants['standard atmosphere']
Out[19]: (101325.0, 'Pa', 0.0)
```

8.7 Ejercicios

1. El fichero de texto medidas_PV_He.txt posee medidas de presión y volumen para 0.1 mol de helio, que se comprime sistemáticamente a temperatura constante. Este experimento se realiza a tres temperaturas distintas. Suponiendo que el gas se comporta idealmente y por tanto que se verifica que PV=nRT, representar los datos y realizar un ajuste lineal P(V) para cada temperatura. ¿Cuánto vale la constante de gases ideales según el experimento?
2. Para una muestra que contiene 10g de yodo 131 (semivida de 8 días), se hacen diariamente cinco medidas independientes a lo largo de 60 días. Esas medidas están en el fichero medidas_decaimiento_yodo131b.txt, donde cada fila corresponde a cada una de las 5 medidas realizadas diariamente en gramos de yodo que queda. Representar en un gráfico con puntos las cinco medidas con colores distintos para cada una y ajustar a cada una la curva teórica de decaimiento. Imprimir por pantalla los parámetros de cada uno de los cinco ajustes.
3. Hacer una función que cree una máscara circular (todos los valores son cero, excepto una zona circular de radio `R` y centrada en un punto arbitrario) con un array 2D para la imagen del castor usada anteriormente `face = misc.face(gray=True)`.

face = misc.face(gray=True)

CAPÍTULO 9

Tratamiento de datos con Pandas

Pandas es la librería para lectura y análisis de datos más completa y potente que existe. Básicamente, ofrece nuevos tipos de datos estructurados que permiten datos heterogéneos y su tratamiento de forma muy flexible. `pandas` trabaja sobre varios tipos estructurados:

- **Series y TimeSeries**: Datos unidimensional con índices
- **DataFrame**: Datos bidimensionales tipo tabla
- **Panel**: Datos de dimensión superior, como cubos de datos

9.1 Series

Son la estructura más sencilla, similar a los arrays de numpy. Se pueden crear a partir de diccionarios o de arrays de numpy:

```
In [1]: datos = [2, 3, 4, 5, 6]

# Serie con indices numéricos, como listas
In [2]: serie = pd.Series(datos)
In [3]: serie[2]
Out[3]: 4

# Serie con indices string, como dict
In [5]: serie = pd.Series(datos, index=['a', 'b', 'c', 'd', 'e'])
In [6]: serie['c']
Out[6]: 4

# Pero los indices también funcionan
In [7]: serie[2]
Out[7]: 4
```

9.2 DataFrame

Es el tipo de datos más últil, similar a una tabla de hoja de cálculo o de base de datos. Se pueden crear con listas, diccionarios, arrays de numpy y con Series.

```
# Diccionario con dos series, A y B, que tienen distinto número de valores
In [20]: datos = {'A' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
....:       'B' : pd.Series([10., 20., 30., 40.], index=['a', 'b', 'c', 'd
↪'])}

# Creamos el DataFrame
In [21]: df = pd.DataFrame(datos)

In [22]: df
Out[23]:
     A     B
a  1.0  10.0
b  2.0  20.0
c  3.0  30.0
d  NaN  40.0
```

Aquí ya ha pasado algo interesante, la columna A tiene tres elementos y la B tiene 4, pero el DataFrame no tiene problema con esto asignando `NaN` (non a number) a fila `d` de la columna A.

```
# Nombres de filas y columnas
In [25]: df.index
Out[25]: Index([u'a', u'b', u'c', u'd'], dtype='object')

In [26]: df.columns
Out[26]: Index([u'A', u'B'], dtype='object')
```

Es más fácil manipular DataFrames o Series que array:

```
# Creamos una nueva columnas C, donde todos los valores son "Si"
# La columna se crea al final
In [30]: df['C'] = "Si"

# Una nueva coluna con valores distintos, pero última fila no tiene medida
In [31]: df['C'] = [-1, -2, -3, NaN]  # En lugar de NaN se puede usar None

# Columna a partir de otras columnas
In [33]: df['D'] = df['A'] >  df['B']

# Si queremos añadir una columna en otro sitio que no sea el final
# podemos usar insert, indicando la posición (0 el primero), nombre y␣
↪valores
In [34]: df.insert(1, 'E', range(4))

# Borramos una columna
In [35]: del df['E']
```

Los DataFrames tiene distintos tipos de indexado:

```
# Seleciona columnas, devuelve un Serie
In [40]: df['A']

# Selecciona una fila por etiqueta
```

```
In [41]: df.loc['a']

# Selecciona una fila por numero
In [42]: df.iloc['a']

# Selecciona un rango de filas, devuelve un DataFrame
In [43]: df[5:10]
```

Para reemplazar un valor o rango, se debe usar `.loc()` o `.iloc()`:

```
# Cambio el valor de fila a, columna C
In [44]: df.loc['a', 'C'] = 3.5
```

Pandas se entiende bien con `numpy` y se pueden aplicar funciones numéricas de `numpy` a DataFrames o convertirlos a arrays:

```
In [45]: np.log10(df)
In [46]: df_array = np.asarray(df)
```

9.3 Lectura de ficheros con Pandas

Pandas posee funciones para leer datos en varios formatos, incluyendo json, csv, html, hadf y Excel. Veamos un ejemplo leyendo un CSV que contiene distinto tipos de datos, cuyas columnas están separadas por ";" y contiene floats con separador decimal con coma. Además el fichero contiene fecha en formato `YYYY-MM-DD H:m`, que queremos interpretar como fecha de Python para poder manipular fechas.

```
# Tupla con los nombres de las columnas, porque el fichero no tiene
In [75]: nombres = "id", "fecha", "N", "E", "H"
In [76]: data = pd.read_csv("DATOS_LEICA_Puntos_II.txt", sep=";",
                  encoding="unicode-escape", decimal=",", parse_dates=[
↪'fecha'],
                  names=nombres, index_col='fecha')
```

Ahora veamos que datos y estructura tiene el fichero:

```
# Forma de los datos, igual que arrays
In [80]: data.shape
Out[80]: (14194, 5)

# Información básica de la tabla, número y tipos de datos
In [81]: data.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14194 entries, 0 to 14193
Data columns (total 5 columns):
id       14194 non-null object
fecha    14194 non-null datetime64[ns]
N        14194 non-null float64
E        14194 non-null float64
H        14194 non-null float64
dtypes: datetime64[ns](1), float64(3), object(1)
memory usage: 554.5+ KB

# Cabecera de la tabla con las primeras entradas
In [82]: data.head()
```

```
Out[82]:
            id                fecha           N             E        H
0     134218028 2015-04-01 00:01:13  457205.564  4081808.210  261.271
1     134218029 2015-04-01 00:05:35  457205.564  4081808.210  261.271
2     134218030 2015-04-01 01:01:13  457205.564  4081808.209  261.271
3     134218031 2015-04-01 01:05:35  457205.564  4081808.209  261.271
4     134218032 2015-04-01 02:01:13  457205.564  4081808.209  261.271


# Cambio el numero de columnas mostradas
In [83]: pd.set_option('display.max_columns', 3)

# Mostrar la 10 primeras entradas
In [84]: data.head(10)
Out[84]:
              id   ...            H
0     134218028   ...     261.271
1     134218029   ...     261.271
2     134218030   ...     261.271
3     134218031   ...     261.271
4     134218032   ...     261.271
5     134218033   ...     261.271
6     134218034   ...     261.270
7     134218035   ...     261.270
8     134218036   ...     261.270
9     134218037   ...     261.270

[10 rows x 5 columns]

# Ordenamos la tabla por N
In [84]: data_sortN = data.sort_values('N')  # o por varios ['N', 'H']
```

Cuando hayamos terminado de manipular los datos, podemos guardarlo a alguno de los formatos que pandas soporta:

```
# A CSV, separado por ;
In [90]: data.to_csv("datos.csv", sep=";", encoding="unicode-escape")

# A LaTeX! con longtable
In [91]: data.to_latex("datos.csv", longtable=True, encoding="unicode-
↪escape")

# Así es, lo copia al portapapeles
In [92]: data.to_clipboard()
```

9.4 Selección de datos

Gracias al potente sistema de índices de Pandas, la selección de datos casi no tiene límites:

```
# Todos los datos, en los que H > 272
In [100]: data_H_272 = data[data['H'] > 272]

# Datos de abril de 2014, usando comparaciones con string de la fecha
In [101]: data_201404 = data[(data['date'] > '2015-04-01') & (data['date']
↪< '2015-05-01')]
```

Pero gracias a que la columna de fecha es `datetime` de Python, podemos usarla para seleccionar:

```
# Todos los datos de 2016
In [103]: data_2016 = data[data['fecha'].dt.year == 2016]

# Valores de N, en array
In [104]: data_2016['N'].values
Out[104]:
array([ 457062.129,  457098.672,  457086.598, ...,  457205.576,
    457205.576,  457205.576])

# Valores mayores que la mediana
In [105]: data_2016[data_2016['N'] > data_2016['N'].median()]
```

El método anterior usando índices es similar al que ya conocemos de arrays de numpy, pero podemos usar el método `query()` para hacer consultas al estilo base de datos, usando cadenas que deben devolver un booleano al interpretarse:

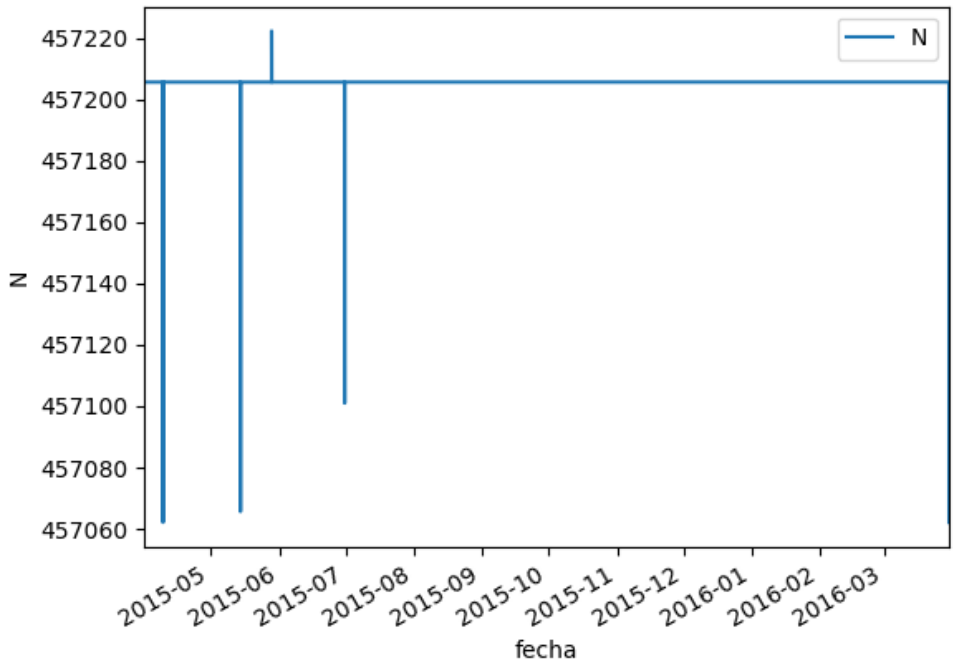
```
# Todos los datos de 2016 con H > 272
In [104]: data_H_272 = data.query('H > 272 and fecha.dt.year > 2016')
```

9.5 Gráficos con Pandas

Ahora que podemos seleccionar los datos y pasarlos a arrays, es muy fácil dibujar los datos con `matplotlib`. Afortunadamente, `pandas` está muy integrado con `matplotlib` y las estructuras de `pandas` contienen métodos para graficar usando `matplotlib`.

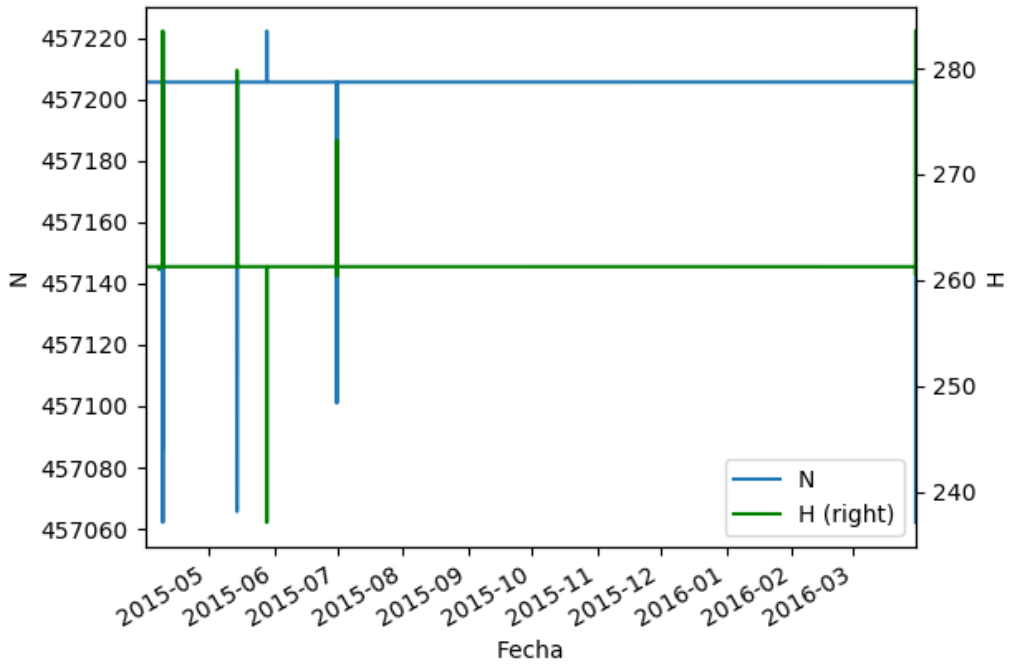
La forma más sencilla de graficar es usando el método `plot()` indicando las columnas `x` e `y`. Si usamos DataFrames directamente, usará como eje X común los índices, salvo que hayamos indicado otra columna que haga índices al leer el fichero, como hicimos (index_col='fechas') y con el resto hará una gráfica por columna.

```
# Dibuja todas las columnas frente a fecha
In [109]: data.plot()

# Dibuja solo N frente a fecha
In [110]: data.plot(y='N')

# Podemos usar también matplotlib para el resto de los elementos
In [111]: plt.xlabel('Fecha')
In [112]: plt.ylabel('N')
```



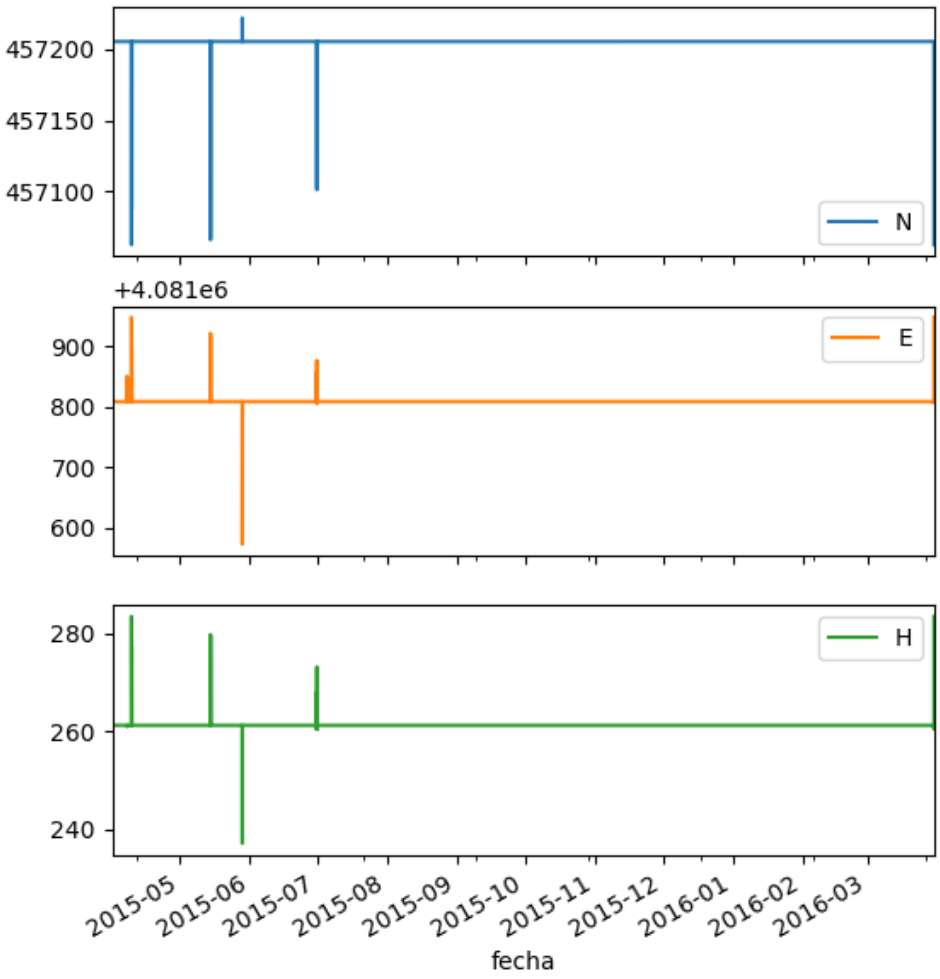
Sin embargo, las columnas N, E y H tienen escalas distintas, por lo que conviene usar uno o más ejes verticales:

```
# Empiezo dibujando solo N frente fecha
In [120]: ax = data.N.plot(legend=True)

# Añado H, indicando que use un eje y secundario
In [121]: data.H.plot(secondary_y=True, style='g', legend=True)

# Pongo las etiquetas a la gráfica
In [122]: ax.set_ylabel('N')
In [123]: ax.set_xlabel('Fecha')
In [124]: ax.right_ax.set_ylabel('H')
```



Si hay muchas gráficas en una misma figura, podemos usar los subplots de matplotlib, de lo que pandas se encarga automáticamente:

```
# Tres gráficas verticales
In [130]: data.plot(subplots=True, figsize=(5, 12));
```


457200
457150
457100
N
+4.081e6
900
800
700
600
E
280
260
240
H
2015-05
2015-06
2015-07
2015-08
2015-09
2015-10
2015-11
2015-12
2016-01
2016-02
2016-03
fecha

CAPÍTULO 10

Cálculo simbólico con Sympy

Sympy permite hacer operaciones analíticas o con símbolos en lugar de con valores numéricos Al igual que en Python existen varios tipos de datos numéricos como enteros (int), decimales (float) o booleanos (bool:True, False, etc.), `Sympy` posee tres tipos de datos propios: Real, Rational e Integer, es decir, números reales, racionales y enteros. Estoquiere decir que `Rational(1,2)` representa 1/2, `Rational(5,2)` a 5/2, etc. en lugar de 0.5 o 2.5.

```
>>> import sympy as sp
>>> a = sp.Rational(1,2)

>>> a
1/2

>>> a*2
1

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

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

```
>>> sp.pi**2
pi**2

>>> sp.pi.evalf()
3.14159265358979

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

como se ve, sin embargo, se puede usar el método `evalf()` para evaluar una expresión para tener un valor en punto flotante (float).

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

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

Y ahora ya podemos manipularlos como queramos:

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

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

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

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

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

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

10.1 Operaciones algebraicas

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

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

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

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

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

10.2 Cálculo de límites

Sympy puede calcular límites usando la función `limit()` con la siguiente sintaxis: `limit(función, variable, punto)`, lo que calcularía el limite de $f(x)$ cuando varia-

ble -> punto:

```
.. code:: ipython3
```

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

es posible incluso usar límites infinitos:

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

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

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

10.2.1 Cálculo de derivadas

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

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

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

Se puede comprobar que es correcto calculando el límite:

```
>>> dx = sp.Symbol('dx')
>>> sp.limit( (tan(x+dx)-tan(x) )/dx, dx, 0)
1 + tan(x)**2
```

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

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

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

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

10.3 Expansión de series

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

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


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

pprint(s)
```

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

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

Si usamos una **qtconsole** (iniciada con `ipython qtconsole`) e ejecutamos `init_printing()`, las fórmulas se mostrarán con LaTeX, si está instalado.

10.4 Integración simbólica

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

```
>>> sp.integrate(6*x**5, x)
x**6
>>> sp.integrate(sp.sin(x), x)
-cos(x)
>>> sp.integrate(sp.log(x), x)
-x + x*log(x)
>>> sp.integrate(2*x + sp.sinh(x), x)
cosh(x) + x**2
```

También con funciones especiales:

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

También es posible calcular integrales definidas:

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

Y también integrales impropias:

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

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

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

10.5 Ecuaciones algebraicas y álgebra lineal

También se pueden resolver sistemas de ecuaciones de manera simbólica:

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

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

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

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

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

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

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

```
>>> import numpy as np

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

```
[0, 0, 1]

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

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

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

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

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

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

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


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

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

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

```
>>> M
[1,         2, 3]
[3, x**(1/2), x]
[2,       2*x, 1]
```

Podemos resolver un sistema de ecuaciones por el método LU:

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

10.6 Gráficos con Sympy

Sympy trae su propio módulo para hacer gráficos, que en realidad **utiliza matplotlib**, pero con la ventaja de que no es necesario darle valores numéricos para representar las funciones, si bien se pueden indicar límites.

```
sp.plot(sp.sin(x)*sp.log(x))

sp.plot(sp.sin(2*sp.sin(2*sp.sin(x))))
```

10.7 Exportando a LaTeX

Cuando hemos terminado los cálculos, podemos pasar a LaTeX la ecuación que queremos:

```
In [5]: int1 = sp.Integral( sp.sin(x)**3 /(2*x), x)

In [6]: sp.latex(int1)
Out[6]: '\\int \\frac{\\sin^{3}{\\left (x \\right )}}{2 x}\\, dx'
```

CAPÍTULO 11

Envío emails con Python

Dependiendo del protocolo, la librería estándar de Python contiene tres módulos de Email, `smtplib`, `imaplib` y `poplib`, además de otras librerías relacionadas con el correo electrónico.

Veremos un ejemplo usando el protocolo `SMTP`, el más usado. Aunque no hay instarlar ningún módulo adicional, será necesario tener instalado en el ordenador algún servidor de correo como **Postfix** (Linux, MacOS y similares) o algún equivalente en Windows.

Haremos un programa que envíe un mensaje personalizado a una lista de personas, para lo que haremos una lista de destinatarios que contiene su nombre de pila y el email, asi que será una lista de tuplas `(nombre, email)`. Luego crearemos un texto que será el contenido del email, que debe contener una variable de formato para poder reemplazar por el nombre del destinatario.

```
import smtplib
from email.mime.text import MIMEText

# Lista de destinatarios, que está compuesta por
# tuplas de dos elemento, nombre y email
destinatarios = [
    ("Irene", "irene@example.com"),
    ("Matías", "matias@example.com"),
    ("Carme", "carme@example.com"),
    ("Pepe", "pepe@example.com")
]


# Mensaje general para todos
mensaje = """
Hola {nombre},

este es un email enviado con Python.

Chao,
Darth Vader
"""
```

Ahora hay que crear un bucle for que recorra la lista de destinatarios tomando el nombre y metiéndolo en `mensaje` y tomando el email correspondiente para enviar:

```
for destinatario in destinatarios:
    # Usamos try-except para captuar errores en caso de que ocurran y así
    # el programa puede continuar con el resto de la lista
    try:
        # Añadimos el nombre al mensaje
        # Usamos un texto de texto plano (otra opción es html
        msg = MIMEText(mensaje.format(nombre=destinatario[0]), 'plain')

        msg['From'] = "Darth Vader <vader@deathstar.com>"
        msg['Subject'] = "[spam] Email masivo usando Python"

        # Destinatario en formato "nombre <email>"
        msg['To'] = "%s <%s>" % destinatario

        # Creamos la instancia de email con el servidor local, que debe
        # tener instalado un servidor de correo como postfix o similar.
        servidor = smtplib.SMTP('localhost')

        # Ahora se envía en mensaje
        servidor.send_message(msg)
        servidor.quit()

        print("Mensaje enviado a {}".format(destinatario[0]))

    except:
        print("Hubo un error enviando el email a {}".
↪format(destinatario[1]) )
```

Es posible usar servicios externos de email como **Gmail** o **Yahoo Mail**, pero en ese caso hay que tener una cuenta en el servio e identificarse con usuario y contraseña:

```
server = smtplib.SMTP('smtp.gmail.com', 587)
server.login(gmail_user, gmail_password)
```

CAPÍTULO 12

Lectura y escritura de hojas Excel

Hay varios módulos de lectura y escritura de hojas Excel e incluso algunos paquetes como `csv` o `pandas` tienen funciones para su lectura, pero con `openpyxl` se tiene un control total de la hoja de cálculo, incluyendo fórmulas y gráficos. Si usas **LibreOffice**, el módulo recomendado es `pyoo`, que además también lee hojas Excel, pero necesita que LibreOffice esté activo para funcionar.

Veamos un ejemplo sencillo para leer la hoja `La Palma.xlsx`, que contiene datos de fechas, números y texto en varias hojas del libro.

```
import openpyxl

# Leo el documento y lo paso a una variable que es un
# objeto workbook, que luego habrá que separar en hojas
wb = openpyxl.load_workbook('data/La Palma.xlsx')

# Primero vemos los nombres de las hojas que tiene el libro
# Devuelve una lista de python
hojas = wb.get_sheet_names()

# Escojo la primera por su nombre
# La variable es un objeto <sheet>
DA = wb.get_sheet_by_name('DOS AGUAS')

# Una columna
A = DA['A']

# Primera celda de la columna
valor = A[1]

# Longitud
tamanho = len(A)

# Puedo saber los límites de la hora viendo el valor
# máximo de filas y columas
# En ocasiones estas dos funcionen parecen no funcionar bien
print("Ultima fila", DA.max_row)
```

```
print("Ultima columna", DA.max_column)

# Si falla max_row se puede recorrer la columna para
# saber cuando termina
ultima_fila = 0
for i, celda in enumerate(A):
    if celda.value is None:
        ultima_fila = i
        break

# Se puede acceder al valor de cada celda con las coordenadas
# como string usando el método value
H1 = DA['H1'].value
H2 = DA['H2'].value

# Tupla de celdas (fijarse cada elemento es una tupla de objetos <cell>)
codigos = DA['A2':'A10']

# Me interesa más trabajar con el valor de la celda que con un objeto
↪<cell>
# así que hago un bucle para hacer una lista con los valores de cada celda
# Lista de string con los codigos
codigos = [celda[0].value for celda in DA['A2':'A10']]

# También se puede recorrer un bloque de tamaño NxM
# que empeza por columnas y luego baja una fila
for fila_celdas in DA['E2':'F10']:
    for celda in fila_celdas:
        print(celda.coordinate, celda.value)
    print('--- Fin de fila ---')
```

Lamentablemente no hay una función para pasar datos a array de `numpy`, pero se podría hacer un bucle como el anterior, de forma más compacta:

```
datos = np.array([[i.value for i in j] for j in DA['H2':'I10']])

# Un formato más cómodo es convertir este array a estructurado
datos_e = np.core.records.fromarrays(datos.T, names='Na, K', formats=
↪'float, float')
```

Ahora, al tener un **array estructurado** podemos aceder a las columnas de datos haciendo `datos_e['Na']`, etc.

Por otro lado, la fechas se leen como objeto `datetime` de Python, lo que no permite manipularlas como queramos.

12.1 Creando una hoja Excel

Veamos ahora cómo escribir una hoja Excel. Tomaremos algunos datos de la hoja anterior y los guardaremos en una nueva.

```
# Empezamos creando un libro vacío, que ya viene con una hoja
wb = openpyxl.Workbook()

# Tomamos la única hoja que existe, que es la activa
```

```
hoja = wb.active  # sheet

# Cambio el nombre de la hoja
hoja.title = "DOS AGUAS"

# Creo otra hoja en el libro
wb.create_sheet(index=1, title='Otra hoja')
# Y así se borra, si hace falta
wb.remove_sheet(wb.get_sheet_by_name('Otra hoja'))

# Ahora le pongo un título para la primera hoja
hoja['A1'] = "Medidas Dos Aguas"
hoja['A2'] = "2013 - 2016"

# Añado estilo a las celdas de título
from openpyxl.styles import Font

hoja['A1'].font = Font(size=20, bold=True)
hoja['A2'].font = Font(size=16, italic=True)

# Uso la hoja DA que tenía de antes y creo listas con
# los valores de algunas columnas
fechas = [celda[0].value for celda in DA['C2':'C10']]
Na = [celda[0].value for celda in DA['H2':'H10']]
K = [celda[0].value for celda in DA['I2':'I10']]
NH4 = [celda[0].value for celda in DA['J2':'J10']]

# Escribimos datos para la cabecera de los datos
hoja['A4'] = "Fecha"
hoja['B4'] = "Na"
hoja['C4'] = "K"
hoja['D4'] = "NH4"

# Fila en la que empiezo a escribir
fila = 5

# Ahora recorremos con un for la lista de fechas y lo hago con enumerate
# para tener también el índice y poder tomar el elemento correspondient
# de las otras lista

for i, fecha in enumerate(fechas):
    # No quiero todos los datos, solo los de 2016
    if fecha.year == 2016:
        # Empiezo a escribir en la fila 5
        hoja['A%d' % (fila)] = fecha.strftime("%Y-%m-%d")
        hoja['B%d' % (fila)] = Na[i]
        hoja['C%d' % (fila)] = K[i]
        hoja['D%d' % (fila)] = NH4[i]

        # Operación absurda para mostrar cómo se añaden formulas
        hoja['E%d' % (fila)] = '=SQRT(B%d)' % (fila)  # '=SUM(B1:B8)'
        fila += 1

# Guardo la hoja
wb.save('data/datos2016.xlsx')
```

CAPÍTULO 13

APÉNDICE B: Instalación de Python y otras herramientas

Ya que Python es un lenguaje de uso general, existen funcionalidades para prácticamente cualquier proyecto o actividad. Aunque la instalación básica de Python ya viene con gran cantidad de funcionalidades y módulos, para trabajos específicos como el análisis científico de datos, son necesarios módulos adicionales. Para este curso usaremos los módulos científicos en Python más populares, que es necesario instalar además del propio Python.

Los módulos básicos que usaremos son los siguientes:

- **Scipy/Numpy** - Paquete científico y numérico básico.
- **Matplotlib** - Librería gráfica para gráficos 2D.
- **IPython** - No es un módulo de Python, sino una consola avanzada, que usaremos en lugar de la de estándar de Python

13.1 Python en Windows o Mac

Tanto en Windows como en Mac es posible instalar Python y luego instalar los módulos científicos necesarios, aunque hay varios proyectos que hacen esto por nosotros, es decir, instalar Python y todos los módulos científicos en un paso. El proyecto más completo y recomendado es Anaconda , un software que reúne Python y gran cantidad de módulos. Basta descargar la versión de **Anaconda - Python 3.6** para el sistema operativo que usemos (lo más probable en 64bits) y seguir las instrucciones.

Nota: Descargar Python-Anaconda

13.2 Python para Linux

Aunque también hay **Anaconda para Linux**, en este caso puede sea más práctico usar la instalación de Python propia del sistema. En Linux Python ya viene instalado, pero la **versión 2.7** y es recomendable

usar la **versión 3**, que hay que instalar si aún no lo está (en versiones recientes de Ubuntu también está Python 3). La manera más fácil instalar software en Linux es usar la consola de comandos, además del administrador de paquetes de Python `pip`:

```
# en Fedora y similares
$ sudo dnf install python36           # Instala Python 3, quizás ya lo␣
↪tienes
$ sudo dnf install python3-pip        # Si es que no lo tienes, puede que␣
↪sí

# en Ubuntu y similares (Debian)
$ sudo apt-get update                 # Actualiza los paquetes antes que␣
↪nada
$ sudo apt-get install python3        # Instala Python 3, casi seguro que␣
↪ya lo tienes
$ sudo apt-get install python3-pip    # Instala el administrador de␣
↪módulos PIP
```

Ahora que tenemos Python 3 y `pip`, podemos usarlo para instalar el resto de módulos necesarios:

```
$ sudo pip3 install scipy         # también instala numpy como dependencia,␣
↪scipy lo necesita
$ sudo pip3 install matplotlib
$ sudo pip3 install pandas
$ sudo pip3 install ipython
$ sudo pip3 install spyder        # Editor para Python
```

Si por algún motivo ya teníamos instadado algún módulo, podemos actualizarlo con `pip3 install --upgrade <modulo>`.

Si tienes problemas con la instalación de Python 3, no es mucho problema usar Python 2 sólo habría que instalar los módulos con `pip` (en lugar de pip3):

```
pip install scipy matplotlib pandas ipython spyder
```

13.3 Editores de texto

Además del uso de Python interactivamente con la terminal de comandos, también necesitaremos hacer programas ejecutables, por lo que necesitaremos un editor de texto/código adecuado. Existen muchas opciones para todos los sistemas operativos y suelen todos venir con alguno útil, aunque a nosotros nos interesa un editor sencillo que de destaque con colores el código y ayude a la edición.

Recordar que hablamos de editores de texto **sin formato** (texto plano) y **NO procesadores de texto** tipo Word o LibreOffice, que generan textos con formato.

El editor de código que usaremos para el curso es Spyder, que **ya viene con Anaconda**, pero hay otras opciones según el sistema operativo:

- **Windows:** Notepad++.
- **Linux:** Gedit o Kate
- **Mac:** TextWrangler.

CAPÍTULO 14

APÉNDICE A: Documentación y referencia

14.1 Python general

- python.org - Sitio oficial, para tutorial y los módulos de la librería estándar.
- Learn Python the hardway - El nombre asusta un poco, pero es una buena introducción a Python en general.
- Automate the Boring Stuff with Python - Estupendo libro online e impreso con ejemplos de tareas útiles con Python: Trabajar con Excel, PDF, Word, Emails, tiempo, etc. Incluye una introducción a Python.
- Python 3 Module of the Week - Módulos de la librería estándar de Python explicados en detalle. Los ejemplos son bastante densos, pero muy detallados.

14.2 Python científico

- Scipy - Sitio principal de referencia para paquetes científicos
- Matplotlib - La librería gráfica estándar *de facto*.
- Python Scientific Lecture Notes <– **RECOMENDADO**
- Scipy Cookbook - Muchas recetas Scipy y Python científico en general.
- Curso de Computación Científica con Python, de 1ro de Física de la ULL.

14.3 Python para sismología

- ObsPy - Framework de Python para sismología. Muy completa librería para tratamiento de datos sismológicos, incluyendo lectura remota de datos en muchos formatos y gráficos especializados.
- PyRocko - Librería para crear herramientas de análisis de datos sismológicos.

- Pisces-DB - Lectura, escritura y creación de bases de datos sismológicos. Integrado con ObsPy.
- VarPy - Pequeña librería para análisis de datos vulcanológicos y física de rocas.

14.4 Python para astronomía

- AstroPython - Sitio principal de referencia para paquetes y herramientas astronómicos en Python.
- Practical Python for Astronomers Tutorial <– **RECOMENDADO**
- Astropy - Documentación oficial de Astropy, incluye tutoriales con ejemplos.