
Clases de Python

Juan Fiol

2020

Dictado de las clases

1. Clase 0: Introducción al lenguaje Python orientado a Ingenierías y Física	3
2. Clase 1: Introducción al lenguaje	15
3. Clase 2: Tipos de datos y control	31
4. Clase 3: Control de flujo y tipos complejos	53
5. Clase 4: Técnicas de iteración y Funciones	75
6. Clase 5: Funciones	93
7. Clase 6: Programación Orientada a Objetos	109
8. Clase 7: Control de versiones y biblioteca standard	125
9. Clase 8: Introducción a Numpy	143
10. Clase 9: Visualización	165
11. Clase 10: Más información sobre Numpy	205
12. Clase 11: Introducción al paquete Scipy	245
13. Clase 12: Un poco de graficación 3D	283
14. Clase 13: Interpolación y ajuste de curvas (fiteo)	305
15. Clase 14: Gráficos interactivos	349
16. Ejercicios	365
17. Material extra	385
18. Material adicional	393

Institución Instituto Balseiro - Univ. Nac. de Cuyo

Fecha Febrero a abril de 2020

Docentes Juan Fiol y Pablo Maceira

CAPÍTULO 1

Clase 0: Introducción al lenguaje Python orientado a Ingenierías y Física

1.1 Autor

Juan Fiol

1.2 Licencia



Esta obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional.

1.3 Python y su uso en ingenierías y ciencias

El objetivo de este curso es realizar una introducción al lenguaje de programación Python para su uso en el trabajo científico y técnico/tecnológico. Si bien este curso *en el final finaliza y empieza por adelante* vamos a tratar algunos de los temas más básicos sólo brevemente. Es recomendable que se haya realizado anteriormente un curso de *Introducción a la programación*, y tener un mínimo de conocimientos y experiencia en programación.

¿Qué es y por qué queremos aprender/utilizar **Python**?

El lenguaje de programación Python fue creado al principio de los 90 por Guido van Rossum, con la intención de ser un lenguaje de alto nivel, con una sintaxis clara, limpia y que intenta ser muy legible. Es un lenguaje de propósito general por lo que puede utilizarse en un amplio rango de aplicaciones.

Desde sus comienzos ha evolucionado y se ha creado una gran comunidad de desarrolladores y usuarios, con especializaciones en muchas áreas. En la actualidad existen grandes comunidades en aplicaciones tan disímiles como desarrollo web, interfaces gráficas (GUI), distintas ramas de la ciencia tales como física, astronomía, biología, ciencias de la computación. También se encuentran muchas aplicaciones en estadística, economía y análisis de finanzas

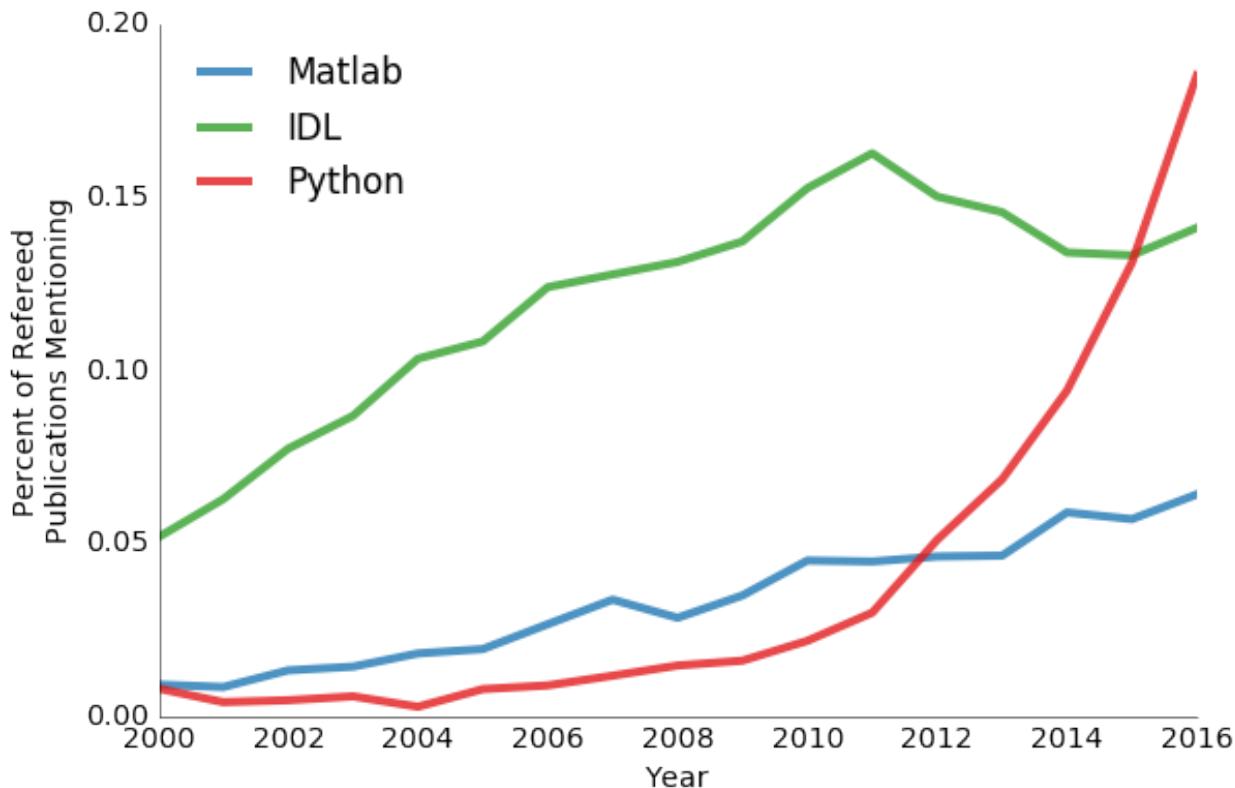
en la bolsa, en interacción con bases de datos, y en el procesamiento de gran número de datos como se encuentran en astronomía, biología, meteorología, etc.

En particular, Python encuentra un nicho de aplicación muy importante en varios aspectos muy distintos del trabajo de ingeniería, científico, o técnico. Por ejemplo, es una lenguaje muy poderoso para analizar y graficar datos experimentales, incluso cuando se requiere procesar un número muy alto de datos. Presenta muchas facilidades para cálculo numérico, se puede conjugar de forma relativamente sencilla con otros lenguajes más tradicionales (Fortran, C, C++), e incluso se puede usar como marco de trabajo, para crear una interfaz consistente y simple de usar en un conjunto de programas ya existentes.

Python es un lenguaje interpretado, como Matlab o IDL, por lo que no debe ser compilado. Esta característica trae aparejadas ventajas y desventajas. La mayor desventaja es que para algunas aplicaciones –como por ejemplo cálculo numérico intensivo– puede ser considerablemente más lento que los lenguajes tradicionales. Esta puede ser una desventaja tan importante que simplemente nos inhabilita para utilizar este lenguaje y tendremos que recurrir (volver) a lenguajes compilados. Sin embargo, existen alternativas que, en muchos casos permiten superar esta deficiencia.

Por otro lado existen varias ventajas relacionadas con el desarrollo y ejecución de los programas. En primer lugar, el flujo de trabajo: *Escribir-ejecutar-modificar-ejecutar-modificar-ejecutar-modificar-ejecutar-* es más ágil. Es un lenguaje pensado para mantener una gran modularidad, que permite reusar el código con gran simpleza. Otra ventaja de Python es que trae incluida una biblioteca con utilidades y extensiones para una gran variedad de aplicaciones **que son parte integral del lenguaje**. Además, debido a su creciente popularidad, existe una multiplicidad de bibliotecas adicionales especializadas en áreas específicas. Por estas razones el tiempo de desarrollo: desde la idea original hasta una versión que funciona correctamente puede ser mucho menor que en otros lenguajes.

A modo de ejemplo veamos un gráfico, que hizo [Juan Nunez-Iglesias](#) basado en código de T. P. Robitaille y actualizado C. Beaumont, correspondiente a la evolución hasta 2016 del uso de Python comparado con otros lenguajes/entornos en el ámbito de la Astronomía.



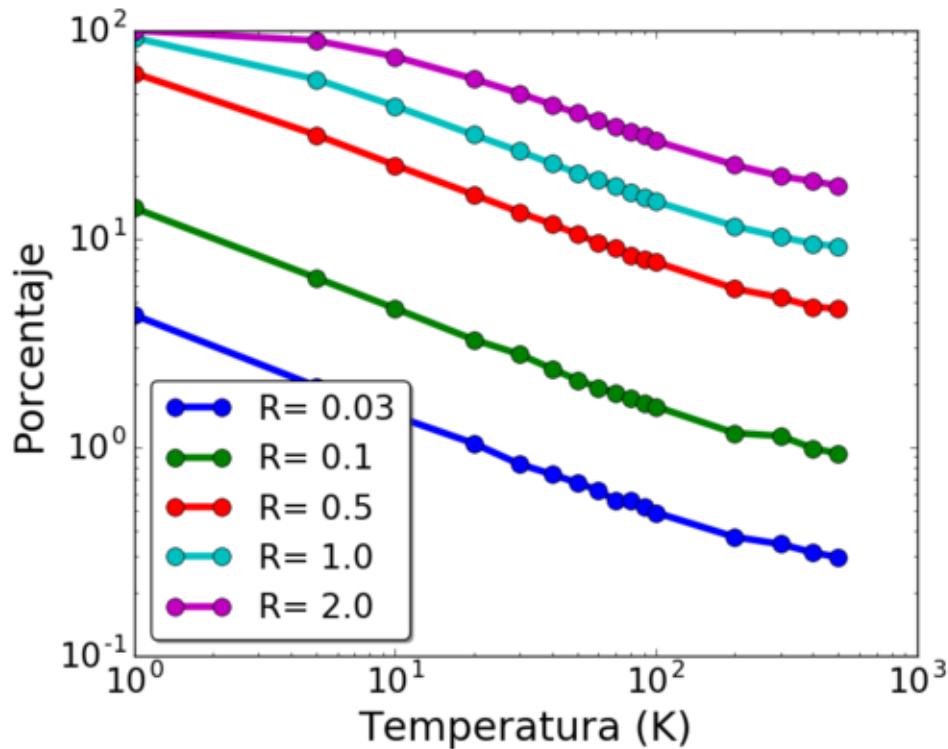
Un último punto que no puede dejar de mencionarse es que Python es libre (y gratis). Esto significa que cada versión nueva puede simplemente descargarse e instalarse sin limitaciones, sin licencias. Además, al estar disponible el código

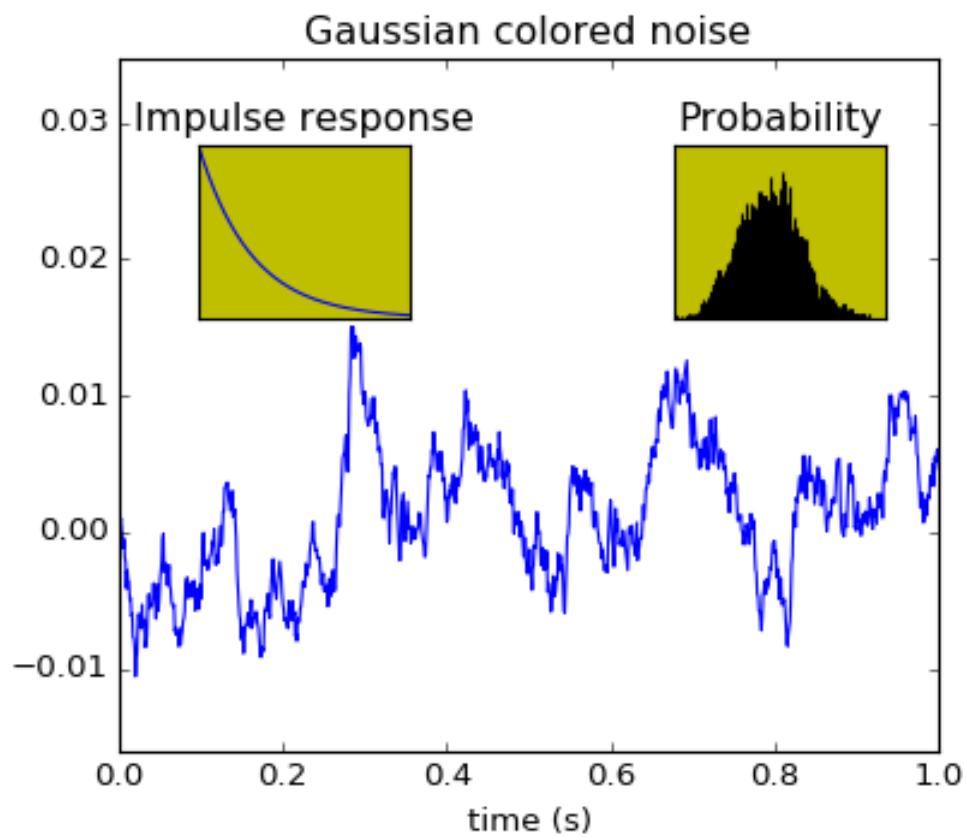
fuente uno podría modificar el lenguaje –una situación que no es muy probable que ocurra– o podría mirar cómo está implementada alguna función –un escenario bastante más probable– para copiar (o tomar inspiración en) alguna funcionalidad que necesitamos en nuestro código.

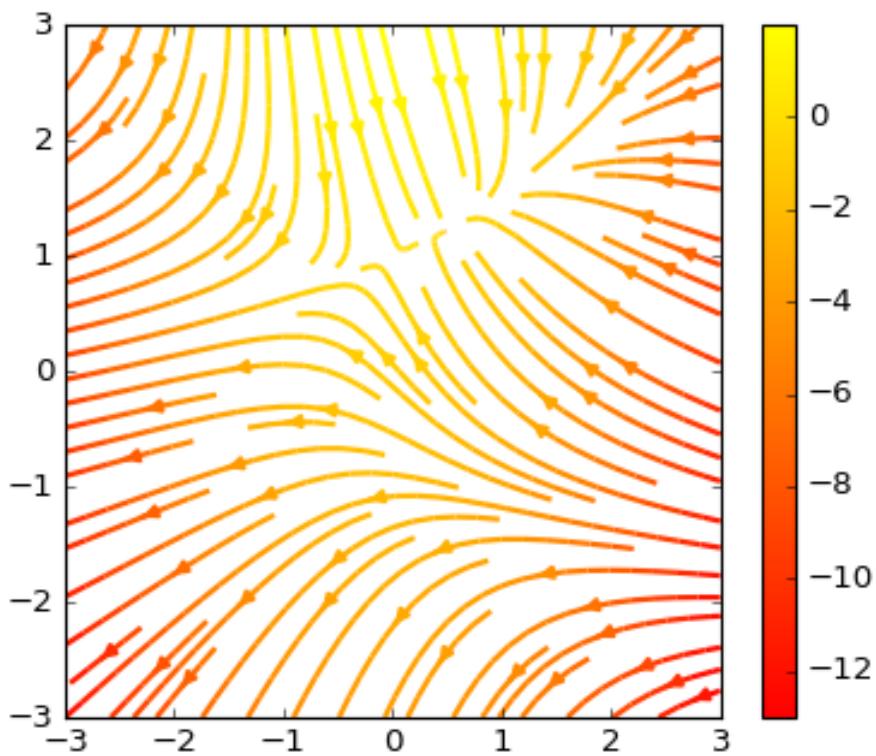
1.4 Visita y excursión a aplicaciones de Python

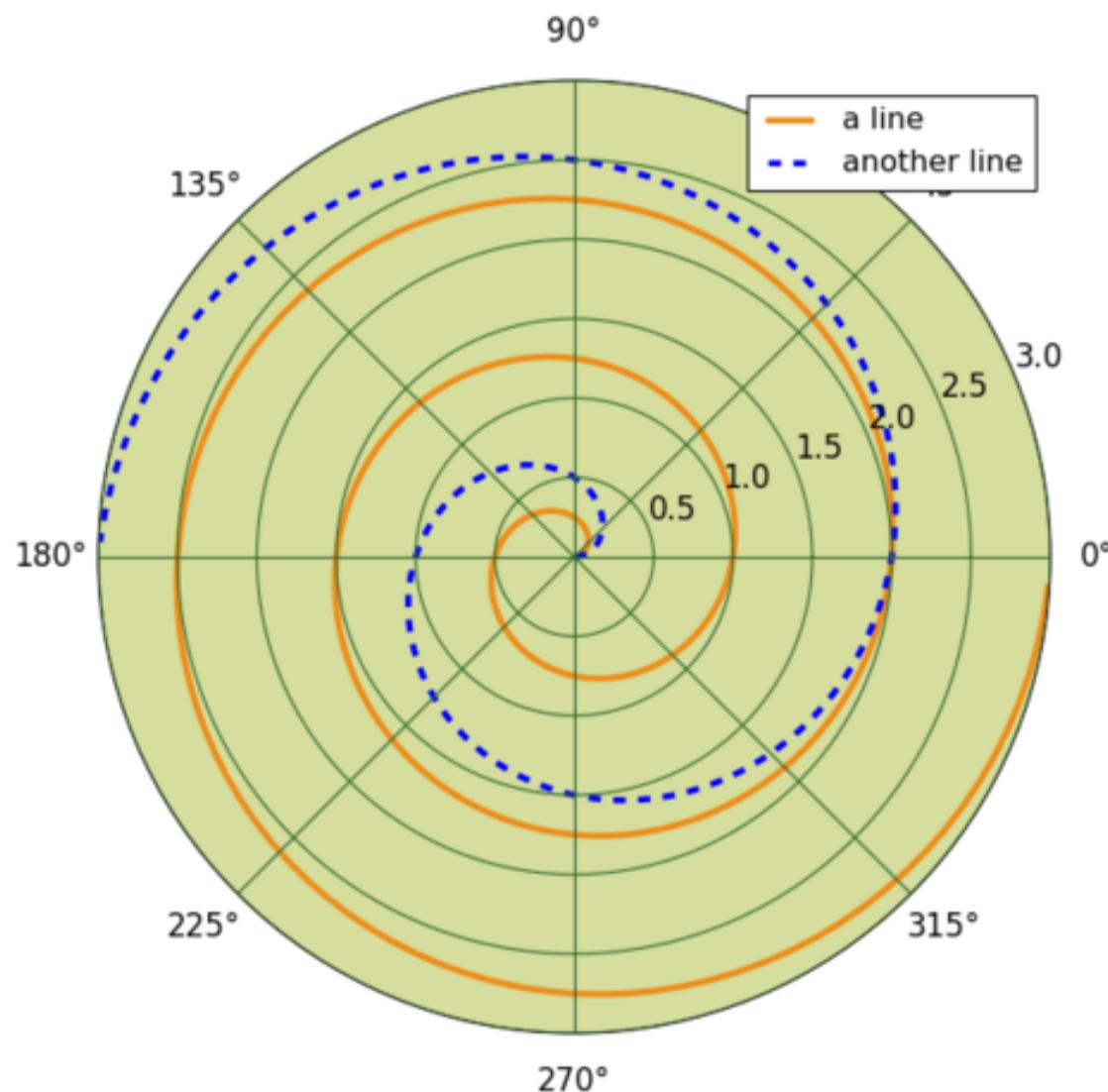
1.4.1 Graficación científica en 2D

La biblioteca **matplotlib** es una de las mejores opciones para hacer gráficos en 2D, con algunas posibilidades para graficación 3D. Los siguientes ejemplos fueron todos creados con matplotlib. El primer gráfico lo hicimos para nuestro uso, y los demás son ejemplos de uso:



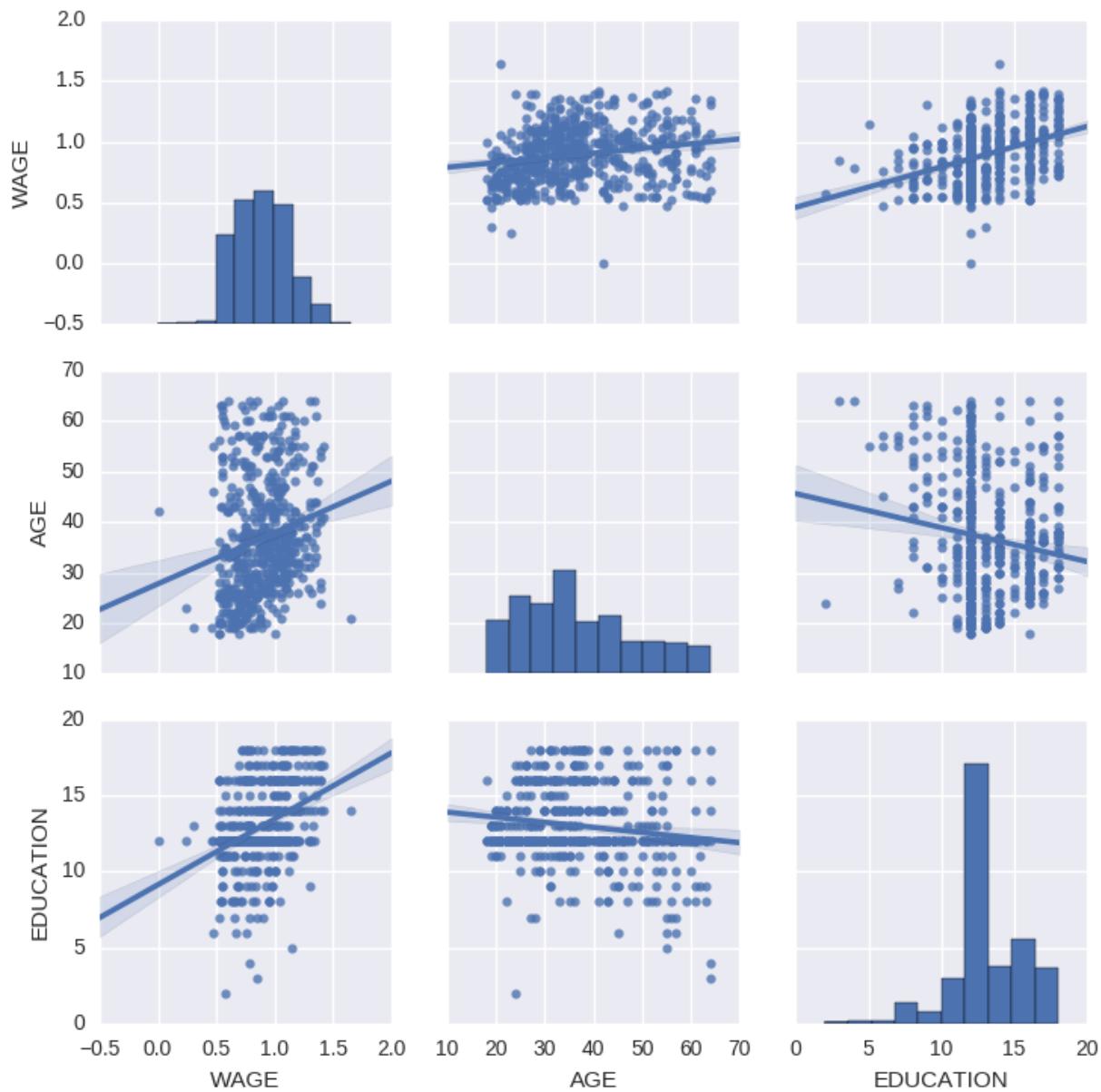






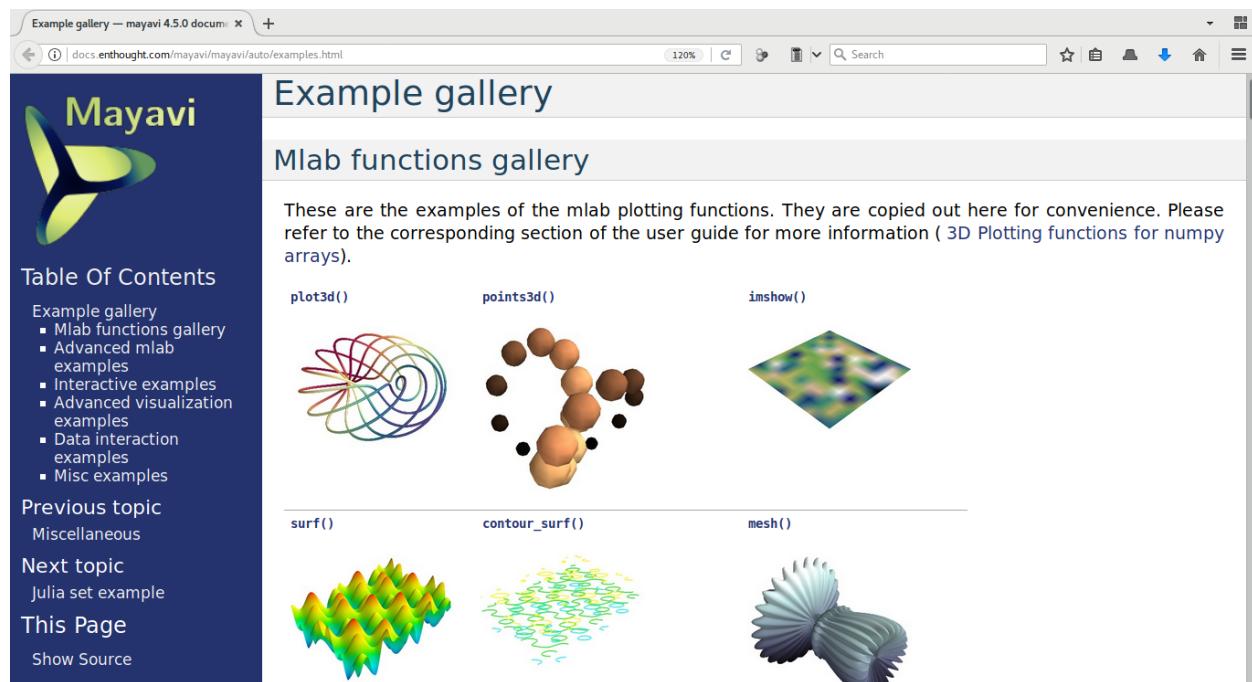
El mejor lugar para un acercamiento es posiblemente la Galería de matplotlib

El siguiente es un ejemplo de *seaborn*, un paquete para visualización estadística (tomado de Scipy Lecture Notes)



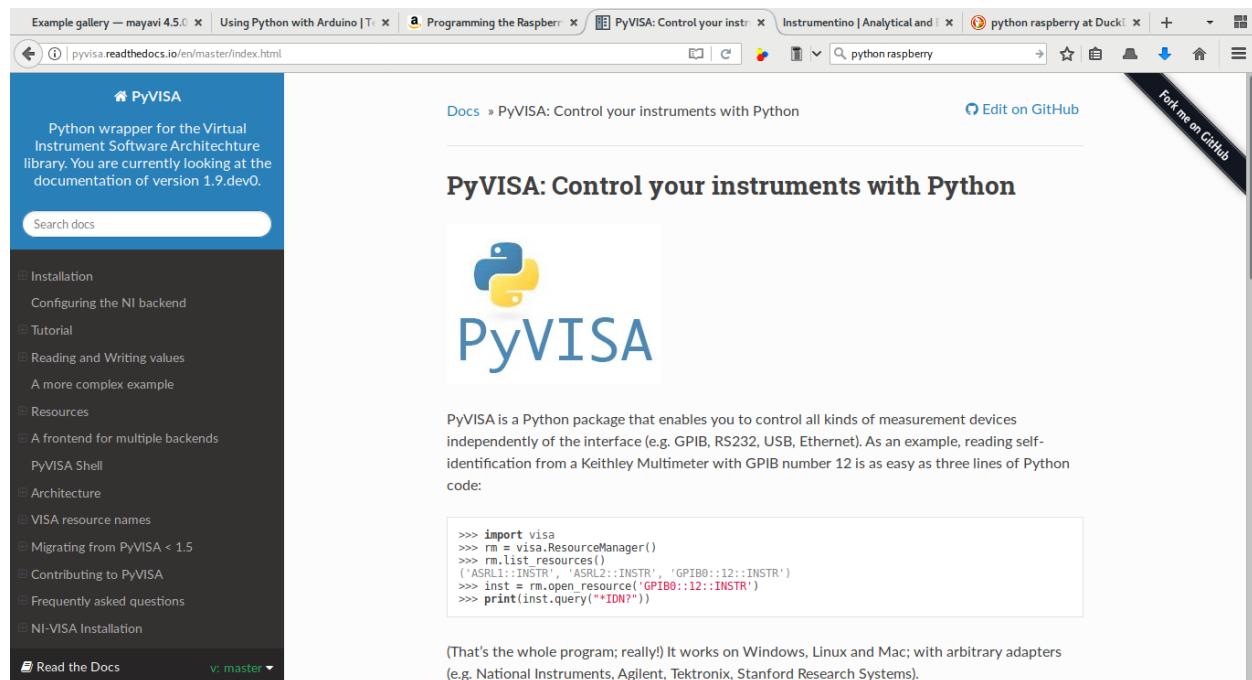
1.4.2 Graficación en 3D

Matplotlib tiene capacidades para realizar algunos gráficos en 3D, si no son demasiado complejos. Para realizar gráficos de mayor complejidad, una de las más convenientes y populares bibliotecas/entornos es Mayavi:



1.4.3 Programación de dispositivos e instrumentos

Python ha ido agregando capacidades para la programación de instrumentos (osciloscopios, tarjetas,), dispositivos móviles, y otros tipos de hardware. Si bien el desarrollo no es tan maduro como el de otras bibliotecas.



The screenshot shows a web browser displaying a tutorial titled 'USING PYTHON WITH ARDUINO'. The page includes a navigation bar with links to 'ABOUT US', 'ARDUINO LESSONS', 'USING PYTHON WITH ARDUINO' (which is highlighted in green), 'BEAGLEBONE BLACK', 'RASPBERRY PI WITH LINUX LESSONS', '3D PRINTING', and 'ENGINEERING CAREER'. On the left sidebar, there's a 'SEARCH' button and a 'Custom Search' link. Below that is a 'RECENT POSTS' section listing various engineering and design tutorials. The main content area features a photograph of an Arduino board connected to a breadboard with two infrared sensors and a motor. A caption below the image reads: 'This Circuit combines the simplicity of Arduino with the Power of Python'. To the right of the image is a 'RECENT POSTS' sidebar with links to other tutorials like 'Sketchup Tutorial LESSON 9: Designing and Printing Box with Movable Hinges' and 'Keys to a Successful Engineering Career LESSON 5: How to Interview for a Job'.

1.4.4 Otras aplicaciones

- Desarrollo web (Django, Cheetah3, Nikola,)
- Python embebido en otros programas:
 - Diseño CAD (Freecad,)
 - Diseño gráfico (Blender, Gimp, I)

1.5 Aplicaciones científicas

Vamos a aprender a programar en Python, y a utilizar un conjunto de bibliotecas creadas para uso científico y técnico:

En este curso vamos a trabajar principalmente con Ipython y Jupyter, y los paquetes científicos Numpy, Scipy y Matplotlib.



Figura 1: Herramientas

1.6 Bibliografía

Se ha logrado constituir una gran comunidad en torno a Python, y en particular en torno a las aplicaciones científicas, por lo que existe mucha información disponible. En la preparación de estas clases se leyó, inspiró, copió, adaptó material de las siguientes fuentes:

1.6.1 Accesible en línea

- La documentación oficial de Python
- El Tutorial de Python, también en español
- Documentación de Numpy
- Documentación de Scipy
- Documentación de Matplotlib, en particular la Galería
- Introduction to Python for Science
- El curso de Python científico
- Las clases de Scipy Scipy Lectures
- Scipy Cookbook
- Computational Statistics in Python

1.6.2 Libros

- The Python Standard Library by Example de Doug Hellman, Addison-Wesley, 2017
- Python Cookbook de David Beazley, Brian K. Jones, O'Reilly Pub., 2013.
- Elegant Scipy de Harriet Dashnow, Stéfan van der Walt, Juan Nunez-Iglesias, O'Reilly Pub., 2017.
- Scientific Computing with Python 3 de Claus Führer, Jan Erik Solem, Olivier Verdier, Packt Pub., 2016.
- Interactive Applications Using Matplotlib de Benjamin V Root, Packt Pub., 2015.
- Mastering Python Regular Expressions de Félix López, Víctor Romero, Packt Pub., 2014,

1.7 Otras referencias de interés

- La documentación de jupyter notebooks
- Otras bibliotecas útiles:
 - Pandas
 - Sympy
 - Información para usuarios de Matlab
- Blogs y otras publicaciones
 - The Glowing Python
 - Python for Signal Processing
 - Ejercicios en Numpy

- Videos de Curso para Científicos e Ingenieros

CAPÍTULO 2

Clase 1: Introducción al lenguaje

2.1 Cómo empezar: Instalación y uso

Python es un lenguaje de programación interpretado, que se puede ejecutar sobre distintos sistemas operativos, esto se conoce como multiplataforma (suele usarse el término *cross-platform*). Además, la mayoría de los programas que existen (y posiblemente todos los que nosotros escribamos) pueden ejecutarse tanto en Linux como en windows y en Mac sin realizar ningún cambio.

Nota: Hay dos versiones activas del lenguaje Python.

- **Python2.X** (Python 2) es una versión madura, estable, y con muchas aplicaciones, y utilidades disponibles. No se desarrolla pero se corrigen los errores.
 - **Python3.X** (Python 3) es la versión presente y futura. Se introdujo por primera vez en 2008, y produjo cambios incompatibles con Python 2. Por esa razón se mantienen ambas versiones y algunos de los desarrollos de Python 3 se portan a Python 2. En este momento la mayoría de las utilidades de Python 2 han sido modificadas para Python 3 por lo que, salvo muy contadas excepciones, no hay razones para seguir utilizando Python 2.
-

2.1.1 Instalación

En este curso utilizaremos **Python 3**

Para una instalación fácil de Python y los paquetes para uso científico se pueden usar alguna de las distribuciones:

- [Anaconda](#). (Linux, Windows, MacOs)
- [Canopy](#). (Linux, Windows, MacOs)
- [Winpython](#). (Windows)
- [Python\(x,y\)](#). (Windows, no actualizado desde 2015)

En linux se podría instalar alguna de estas distribuciones pero puede ser más fácil instalar directamente todo lo necesario desde los repositorios. Por ejemplo en Ubuntu:

```
`sudo apt-get install ipython3 ipython3-notebook spyder python3-matplotlib python3-
˓→numpy python3-scipy`  
o, en Fedora 28, en adelante:  
`sudo dnf install python3-ipython python3-notebook python3-matplotlib python3-numpy_
˓→python3-scipy`
```

- Editores de Texto:
 - En windows: [Notepad++](#), [Jedit](#), (no Notepad o Wordpad)
 - En Linux: cualquier editor de texto (gedit, geany, kate, nano, emacs, vim,)
 - En Mac: TextEdit funciona, sino TextWrangler, [JEdit](#),
- Editores Multiplataforma e IDEs
 - [spyder](#). (IDE - También viene con Anaconda, y con Python(x,y)).
 - [Atom](#) Moderno editor de texto, extensible a través de paquetes (más de 3000).
 - [Pycharm](#). (IDE, una versión comercial y una libre, ambos con muchas funcionalidades)

2.1.2 Documentación y ayudas

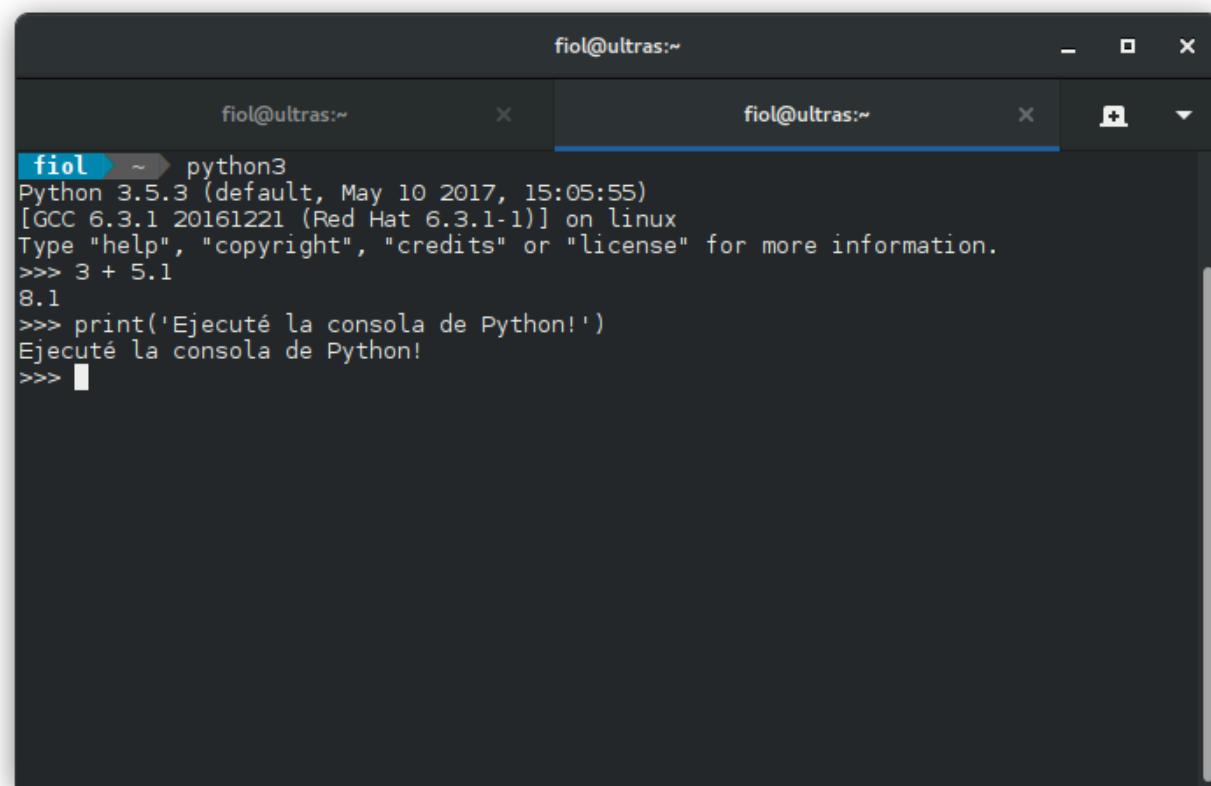
Algunas fuentes de ayuda *constante* son:

- La documentación oficial de Python
- En particular el [Tutorial](#), también [en español](#) y la referencia de bibliotecas
- En una terminal, puede obtener información sobre un paquete con `pydoc <comando>`
- En una consola interactiva de **Python**, mediante `help(<comando>)`
- La documentación de los paquetes:
 - [Numpy](#)
 - [Matplotlib](#), en particular la [galería](#)
 - [Scipy](#)
- Buscar palabras clave + python en un buscador. Es particularmente útil el sitio [stackoverflow](#)

2.1.3 Uso de Python: Interactivo o no

Interfaces interactivas (consolas/terminales, notebooks)

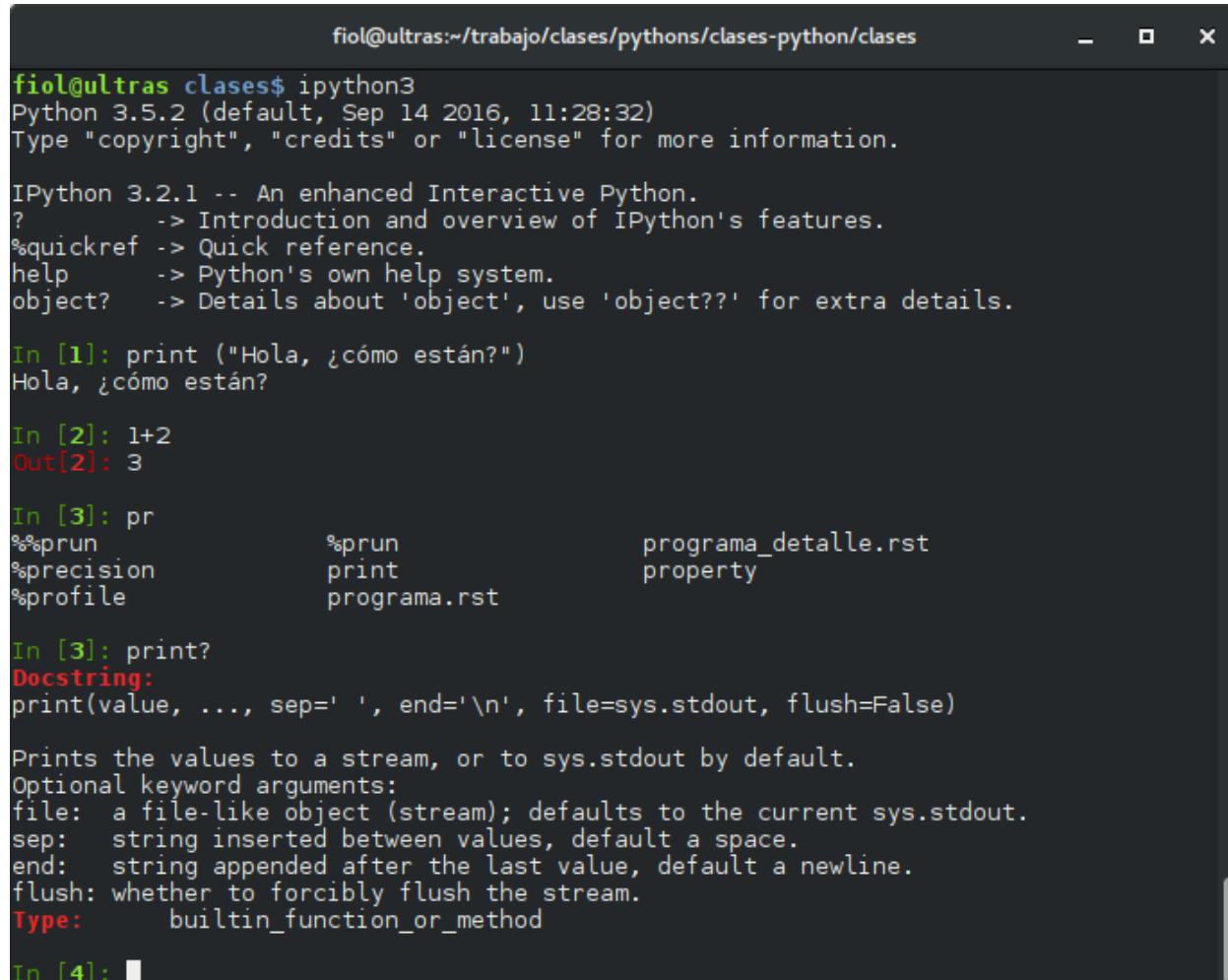
Hay muchas maneras de usar el lenguaje Python. Es un lenguaje **interpretado e interactivo**. Si ejecutamos la consola (`cmd.exe` en windows) y luego `python`, se abrirá la consola interactiva



The screenshot shows a terminal window with two tabs open. The active tab is titled "fiol@ultras:~" and contains a Python 3 interactive session. The session starts with the Python version and build information, followed by a calculation of 3 + 5.1, and a print statement outputting "Ejecuté la consola de Python!". The second tab is also titled "fiol@ultras:~".

```
fiol ~ python3
Python 3.5.3 (default, May 10 2017, 15:05:55)
[GCC 6.3.1 20161221 (Red Hat 6.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 3 + 5.1
8.1
>>> print('Ejecuté la consola de Python!')
Ejecuté la consola de Python!
>>>
```

En la consola interactiva podemos escribir sentencias o pequeños bloques de código que son ejecutados inmediatamente. Pero *la consola interactiva* estándar no tiene tantas características de conveniencia como otras, por ejemplo **IPython** que viene con accesorios de *comfort*.



The screenshot shows a terminal window with the following content:

```
fiol@ultras:~/trabajo/clases/pythons/clases-pyton/clases
fiol@ultras clases$ ipython3
Python 3.5.2 (default, Sep 14 2016, 11:28:32)
Type "copyright", "credits" or "license" for more information.

IPython 3.2.1 -- 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]: print ("Hola, ¿cómo están?")
Hola, ¿cómo están?

In [2]: 1+2
Out[2]: 3

In [3]: pr
%%prun          %prun          programa_detalle.rst
%precision      print          property
%profile        programa.rst

In [3]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method

In [4]:
```

La consola IPython supera a la estándar en muchos sentidos. Podemos autocompletar (<TAB>), ver ayuda rápida de cualquier objeto (?), etc.

Programas/scripts

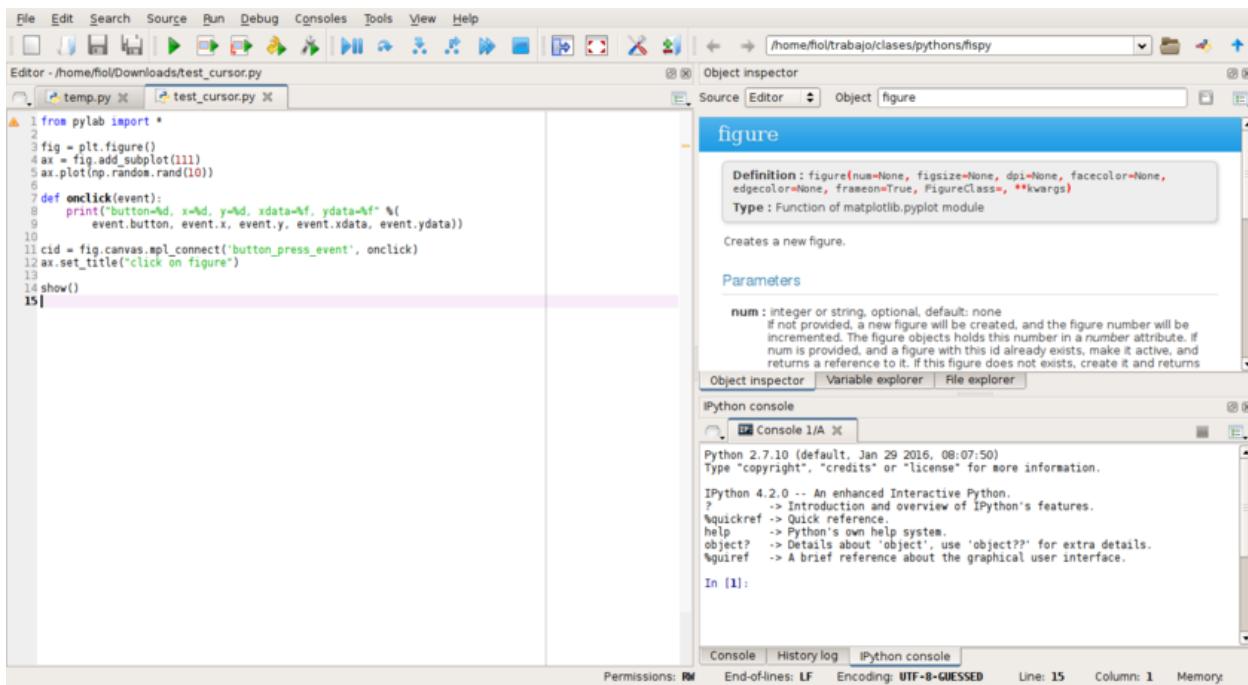
Una forma muy común/poderosa de usar Python es en forma *no interactiva*, escribiendo *programas* o *scripts*. Esto es, escribir nuestro código en un archivo con extensión .py para luego ejecutarlo con el intérprete. Por ejemplo, podemos crear un archivo *hello.py* (al que se le llama *módulo*) con este contenido:

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

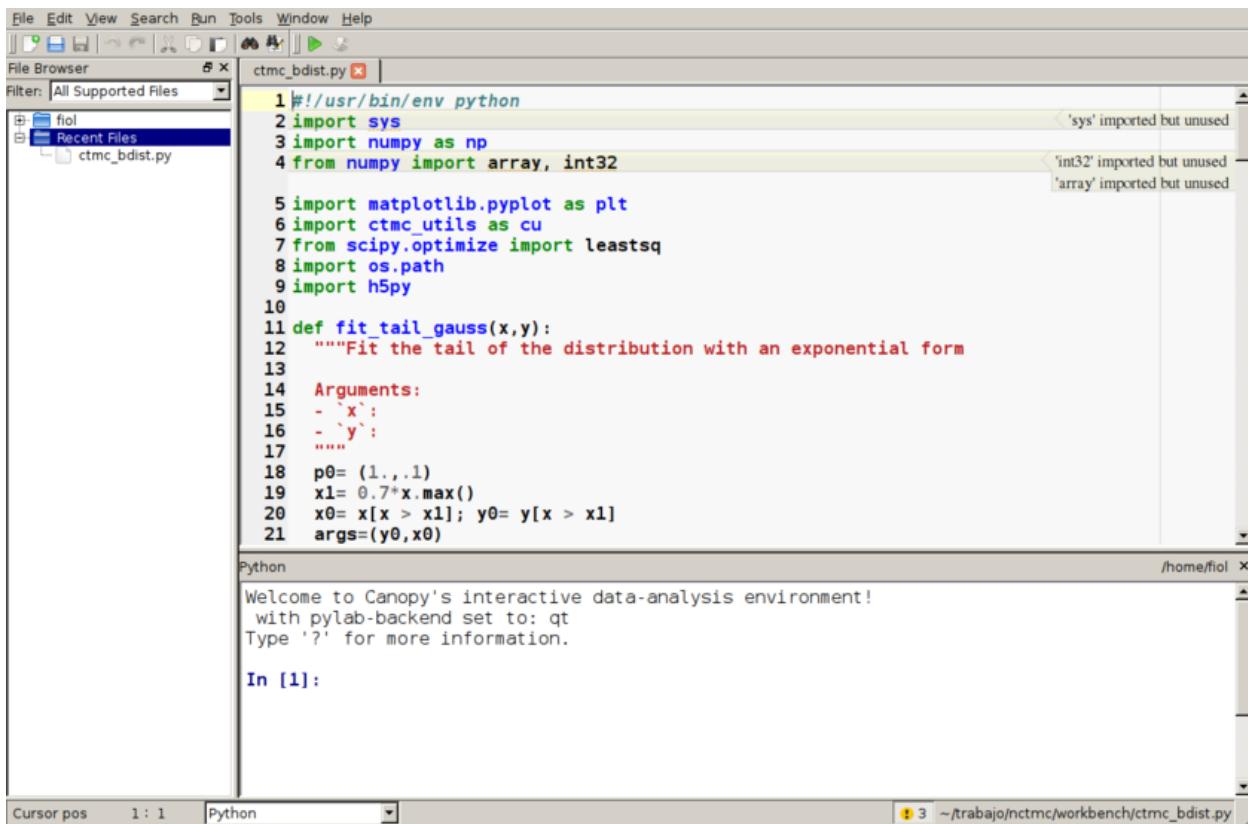
Si ejecutamos `python hello.py` o `ipython hello.py` se ejecutará el interprete Python y obtendremos el resultado esperado (impresión por pantalla de Hola Mundo!, sin las comillas)

Python no exige un editor específico y hay muchos modos y maneras de programar. Lo que es importante al programar en Python es que la *indentación* define los bloques (definición de loops, if/else, funciones, clases, etc). Por esa razón es importante que el tabulado no mezcle espacios con caracteres específicos de tabulación. La manera que recomendaría es usar siempre espacios (uno usa la tecla [TAB] pero el editor lo traduce a un número determinado de espacios). La indentación recomendada es de **4** espacios (pero van a notar que yo uso **2**).

Un buen editor es **Spyder** que tiene características de IDE (entorno integrado: editor + ayuda + consola interactiva).



Otro entorno integrado, que funciona muy bien, viene instalado con **Canopy**.



En ambos casos se puede ejecutar todo el módulo en la consola interactiva que incluye. Alternativamente, también se puede seleccionar **sólo** una porción del código para ejecutar.

2.1.4 Notebooks de Jupyter

Para trabajar en forma interactiva es muy útil usar los *Notebooks* de Jupyter. El notebook es un entorno interactivo enriquecido. Podemos crear y editar celdas código Python que se pueden editar y volver a ejecutar, se pueden intercalar celdas de texto, fórmulas matemáticas, y hacer que los gráficos se muestren inscritados en la misma pantalla o en ventanas separadas. Además se puede escribir texto con formato (como este que estamos viendo) con secciones, títulos. Estos archivos se guardan con extensión *.ipynb*, que pueden exportarse en distintos formatos tales como html (estáticos), en formato PDF, LaTeX, o como código python puro. (.py)

2.2 Ejercicios 01 (a)

1. Abra una terminal (consola) de Ipython y utilícela como una calculadora para realizar las siguientes acciones:
 - Suponiendo que, de las cuatro horas de clases, tomamos un descanso de 15 minutos y nos distraemos otros 13 minutos, calcular cuántos minutos efectivos de trabajo tendremos en las 16 clases.
 - Para la cantidad de alumnos presentes en el aula: ¿cuántas horas-persona de trabajo hay involucradas?
2. Muestre en la consola de Ipython:
 - el nombre de su directorio actual
 - los archivos en su directorio actual
 - Cree un subdirectorio llamado `tmp`
 - si está usando linux, la fecha y hora
 - Borre el subdirectorio `tmp`
3. Abra un editor de textos y escriba las líneas necesarias para imprimir por pantalla las siguientes frases (una por línea). Guarde y ejecute su programa.
 - Hola, por primera vez
 - Hola, hoy es mi día de escribir frases intrascendentes
 - Hola, nuevamente, y espero que por última vez
 - $E = mc^2$
 - Adiós

Ejecute el programa.

2.3 Comandos de Ipython

2.3.1 Comandos de Navegación

IPython conoce varios de los comandos más comunes en Linux. En la terminal de IPython estos comandos funcionan independientemente del sistema operativo (sí, incluso en windows). Estos se conocen con el nombre de **comandos mágicos** y comienzan con el signo porcentaje %. Para obtener una lista de los comandos usamos `%lsmagic`:

```
%lsmagic
```

```
Available line magics:
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cat
↪ %cd %clear %colors %conda %config %connect_info %cp %debug %dhist %dirs
↪ %doctest_mode %ed %edit %env %gui %hist %history %killbgscripts %ldir
↪ %less %lf %lk %ll %load %load_ext %loadpy %logoff %logon %logstart
↪ %logstate %logstop %ls %lsmagic %lx %macro %magic %man %matplotlib %mkdir
↪ %more %mv %notebook %page %pastebin %pdb %pdef %pdoc %pfile %pinfo
↪ %pinfo2 %pip %popd % pprint %precision %prun %psearch %psource %pushd %pwd
↪ %pycat %pylab %qtconsole %quickref %recall %rehashx %reload_ext %rep %rerun_
↪ %reset %reset_selective %rm %rmdir %run %save %sc %set_env %store %sx
↪ %system %tb %time %timeit %unalias %unload_ext %who %who_ls %whos %xdel
↪ %xmode

Available cell magics:
%% ! %%HTML %%SVG %%bash %%capture %%debug %%file %%html %%javascript %%js %
↪ %% latex %% markdown %% perl %% prun %% pypy %% python %% python2 %% python3 %% ruby
↪ %% script %% sh %% svg %% sx %% system %% time %% timeit %% writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

2.3.2 Algunos de los comandos mágicos

Algunos de los comandos mágicos más importantes son:

- %cd *direct* (Nos ubica en la carpeta direct)
- %ls (muestra un listado del directorio)
- %pwd (muestra el directorio donde estamos trabajando)
- %run *filename* (corre un dado programa)
- %hist (muestra la historia de comandos utilizados)
- %mkdir *dname* (crea un directorio llamado dname)
- %cat *fname* (Muestra por pantalla el contenido del archivo fname)
- Tab completion: Apretando [TAB] completa los comandos o nombres de archivos.

En la consola de IPython tipee %cd ~ (*i.e.* %cd – espacio – tilde, y luego presione [RETURN]. Esto nos pone en el directorio HOME (default).

Después tipee %pwd (print working directory) y presione [RETURN] para ver en qué directorio estamos:

```
%cd ~
```

```
/home/fiol
```

```
%pwd
```

```
'/home/fiol'
```

En windows, el comando pwd va a dar algo así como:

Clases de Python

```
In [3]: pwd  
Out[3]: C:\\\\Users\\\\usuario
```

Vamos a crear un directorio donde guardar ahora los programas de ejemplo que escribamos. Lo vamos a llamar scripts.

Primero vamos a ir al directorio que queremos, y crearlo. En mi caso lo voy a crear en mi HOME.

```
%cd
```

```
/home/fiol
```

```
%mkdir scripts
```

```
%cd scripts
```

```
/home/fiol/scripts
```

Ahora voy a escribir una línea de **Python** en un archivo llamado *prog1.py*. Y lo vamos a ver con el comando %cat

```
%cat prog1.py
```

```
print ("hola y chau")
```

```
%run prog1.py
```

```
hola y chau
```

```
%hist
```

```
%lsmagic  
%cd ~  
%pwd  
%cd  
%mkdir scripts  
%mkdir scripts  
%cd scripts  
%cat prog1.py  
%pycat prog1.py  
%cat prog1.py  
%run prog1.py  
%hist
```

Hay varios otros comandos mágicos en IPython. Para leer información sobre el sistema de comandos mágicos utilice:

```
%magic
```

Finalmente, para obtener un resumen de comandos con una explicación breve, utilice:

```
%quickref
```

2.3.3 Comandos de Shell

Se pueden correr comandos del sistema operativo (más útil en linux) tipeando ! seguido por el comando que se quiere ejecutar. Por ejemplo:

comandos

```
!echo " " >> prog1.py
```

```
!echo "print('hola otra vez')" >> prog1.py
```

```
%cat prog1.py
```

```
print("hola y chau")  
print('hola otra vez')
```

```
%run prog1.py
```

```
hola y chau  
hola otra vez
```

```
!date
```

```
Mon 03 Feb 2020 09:57:19 AM -03
```

2.4 Ejercicios 01 (b)

Inicie una terminal de *Jupyter* y realice las siguientes operaciones:

4. Para cubos de lados de longitud $L = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
 5. Para esferas de radios $r = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
 6. Fíjese si alguno de los valores de $x = 2,05, x = 2,11, x = 2,21$ es un cero de la función $f(x) = x^2 + x/4 - 1/2$.
-

2.5 Conceptos básicos de Python

2.5.1 Características generales del lenguaje

Python presenta características modernas. Posiblemente su característica más visible/notable es que la estructuración del código está fuertemente relacionada con su legibilidad:

- Es un lenguaje interpretado (no se compila separadamente)
- Provee tanto un entorno interactivo como de programas separados

- Las funciones, bloques, ámbitos están definidos por la indentación
- Tiene una estructura altamente modular, permitiendo su reusabilidad
- Es un lenguaje de *tipeado dinámico*, no tenemos que declarar el tipo de variable antes de usarla.

Python es un lenguaje altamente modular con una biblioteca standard que provee de funciones y tipos para un amplio rango de aplicaciones, y que se distribuye junto con el lenguaje. Además hay un conjunto muy importante de utilidades que pueden instalarse e incorporarse muy fácilmente. El núcleo del lenguaje es pequeño, existiendo sólo unas pocas palabras reservadas:

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

2.5.2 Tipos de variables

Python es un lenguaje de muy alto nivel y por lo tanto trae muchos *tipos* de datos ya definidos:

- Números: enteros, reales, complejos
- Tipos lógicos (booleanos)
- Cadenas de caracteres (strings) y bytes
- Listas: una lista es una colección de cosas, ordenadas, que pueden ser todas distintas entre sí
- Diccionarios: También son colecciones de cosas, pero no están ordenadas y son identificadas con una etiqueta
- Conjuntos, tuples,

Tipos simples: Números

Hay varios tipos de números en Python. Aquí definimos y asignamos valor a distintas variables:

```
a = 13
b = 1.23
c = a + b
print(a, type(a))
print(b, type(b))
print(c, type(c))
```

```
13 <class 'int'>
1.23 <class 'float'>
14.23 <class 'float'>
```

Esta es una de las características de Python. Se define el tipo de variable en forma dinámica, al asignarle un valor.

De la misma manera se cambia el tipo de una variable en forma dinámica, para poder operar. Por ejemplo en el último caso, la variable `a` es de tipo `int`, pero para poder sumarla con la variable `b` debe convertirse su valor a otra de tipo `float`.

```
print (a, type(a))
a = 1.5 * a
print (a, type(a))
```

```
13 <class 'int'>
19.5 <class 'float'>
```

Ahora, la variable `a` es del tipo `float`.

En Python 3 la división entre números enteros da como resultado un número flotante

```
print(20/5)
print(type(20/5))
print(20/3)
```

```
4.0
<class 'float'>
6.666666666666667
```

Advertencia: En *Python 2.x* la división entre números enteros es entera

Por ejemplo, en cualquier versión de Python 2 tendremos: $1/2 = 3/4 = 0$. Esto es diferente en *Python 3* donde $1/2=0.5$ y $3/4=0.75$.

```
print(1/2)
print(20/3)
```

```
0.5
6.666666666666667
```

```
%%python2
print 1/2
print 20/3
```

```
0
6
```

Nota: La función `print`

Estuvimos usando, sin hacer ningún comentario, la función `print(arg1, arg2, arg3, ..., sep=' ', end='\n', file=sys.stdout, flush=False)` acepta un número variable de argumentos. Imprime por pantalla todos los argumentos que se le pasan separados por el string `sep` (cuyo valor por defecto es un espacio), y termina con el string `end` (con valor por defecto *newline*).

```
help(print)
```

```
print(3,2,'hola')
print(4,1,'chau')
```

```
3 2 hola
4 1 chau
```

Clases de Python

```
print(3,2,'hola',sep='++++',end=' -> ')
print(4,1,'chau',sep='++++')
```

```
3++++2++++hola -> 4++++1++++chau
```

Advertencia: En *Python 2.x* no existe la función `print()`.

Se trata de un comando. Para escribir las sentencias anteriores en Python 2 sólo debemos omitir los paréntesis.

Números complejos

Los números complejos son parte standard del lenguaje, y las operaciones básicas que están incorporadas en forma nativa pueden utilizarse normalmente

```
z1 = 3 + 1j
z2 = 2 + 2.124j
print ('z1 =', z1, ', z2 =', z2)
```

```
z1 = (3+1j) , z2 = (2+2.124j)
```

```
print('1.5j * z2 + z1 = ', 1.5j * z2 + z1) # sumas, multiplicaciones de números_
↪complejos
print('z2^2 = ', z2**2) # potencia de números complejos
print('conj(z1) = ', z1.conjugate())
```

```
1.5j * z2 + z1 = (-0.1859999999999994+4j)
z2^2 = (-0.5113760000000003+8.496j)
conj(z1) = (3-1j)
```

```
print ('Im(z1) = ', z1.imag)
print ('Re(z1) = ', z1.real)
print ('abs(z1) = ', abs(z1))
```

```
Im(z1) = 1.0
Re(z1) = 3.0
abs(z1) = 3.1622776601683795
```

Operaciones

Las operaciones aritméticas básicas son:

- adición: +
- sustracción: -
- multiplicación: *
- división: /
- potencia: **
- módulo: %

- división entera: //

Las operaciones se pueden agrupar con parentesis y tienen precedencia estándar.

División entera (//) significa quedarse con la parte entera de la división (sin redondear).

Nota: Las funciones matemáticas están incluidas en el lenguaje.

En particular las funciones elementales: trigonométricas, hiperbólicas, logaritmos no están incluidas. En todos los casos es fácil utilizarlas porque las proveen módulos. Lo veremos pronto.

```
print('división de 20/3:      ', 20/3)
print('parte entera de 20/3:   ', 20//3)
print('fracción restante de 20/3:', 20/3 - 20//3)
print('Resto de 20/3:          ', 20%3)
```

```
división de 20/3:      6.666666666666667
parte entera de 20/3:   6
fracción restante de 20/3: 0.666666666666667
Resto de 20/3:          2
```

Tipos simples: Booleanos

Los tipos lógicos o *booleanos*, pueden tomar los valores *Verdadero* o *Falso* (True o False)

```
t = False
print('t is True?', t == True)
print('t is False?', t == False)
```

```
t is True? False
t is False? True
```

```
c = (t == True)
print('t is True?', c)
print(type(c))
```

```
t is True? False
<class 'bool'>
```

Hay un tipo *especial*, el elemento None.

```
print ('True == None: ', True == None)
print ('False == None: ', False == None)
a = None
print ('type(a): ', type(a))
print (bool(None))
```

```
True == None: False
False == None: False
type(a): <class 'NoneType'>
False
```

Aquí hemos estado preguntando si dos cosas eran iguales o no (igualdad). También podemos preguntar si una es la otra (identidad):

Clases de Python

```
d = 1
```

```
a = None
b= True
c = a
print ('b is True: ', b is True)
print ('a is None: ', a is None)
print ('c is a: ', c is a)
```

```
b is True:  True
a is None:  True
c is a:  True
```

Operadores lógicos

Los operadores lógicos en Python son muy explícitos:

```
A == B  (A igual que B)
A > B  (A mayor que B)
A < B  (A menor que B)
A >= B  (A igual o mayor que B)
A <= B  (A igual o menor que B)
A != B  (A diferente que B)
A in B  (A incluido en B)
A is B  (Identidad: A es el mismo elemento que B)
```

y a todos los podemos combinar con **not**, que niega la condición

```
print ('£20/3 == 6?', 20/3 == 6)
print ('£20//3 == 6?', 20//3 == 6)
print ('£20//3 >= 6?', 20//3 >= 6)
print ('£20//3 > 6?', 20//3 > 6)
```

```
£20/3 == 6? False
£20//3 == 6? True
£20//3 >= 6? True
£20//3 > 6? False
```

```
a = 1001
b = 1001
print ('a == b:', a == b)
print ('a is b:', a is b)
print ('a is not b:', a is not b)
```

```
a == b:  True
a is b:  False
a is not b:  True
```

Note que en las últimas dos líneas estamos fijándonos si las dos variables son la misma (identidad), y no ocurre aunque vemos que sus valores son iguales.

Warning: En algunos casos **Python** puede reusar un lugar de memoria.

Por razones de optimización, en algunos casos **Python** puede utilizar el mismo lugar de memoria para dos variables que tienen el mismo valor, cuando este es pequeño.

Por ejemplo, la implementación que estamos usando, utiliza el mismo lugar de memoria para dos números enteros iguales si son menores o iguales a 256. De todas maneras, es claro que deberíamos utilizar el símbolo `==` para probar igualdad y la palabra `is` para probar identidad.

```
a = 11
b = 11
print (a, ': a is b:', a is b)
```

```
11 : a is b: True
```

```
b=2*b
print(a,b,a is b)
```

```
11 22 False
```

Acá utilizó otro lugar de memoria para guardar el nuevo valor de `b` (22).

Esto sigue valiendo para otros números:

```
a = 256
b = 256
print (a, ': a is b:', a is b)
```

```
256 : a is b: True
```

```
a = 257
b = 257
print (a, ': a is b:', a is b)
```

```
257 : a is b: False
```

En este caso, para valores mayores que 256, ya no usa el mismo lugar de memoria. Tampoco lo hace para números de punto flotante.

```
a = -256
b = -256
print (a, ': a is b:', a is b)
print(type(a))
```

```
-256 : a is b: False
<class 'int'>
```

```
a = 1.5
b = 1.5
print (a, ': a is b:', a is b)
print(type(a))
```

```
1.5 : a is b: False
<class 'float'>
```

2.6 Ejercicios 01 (c)

7. Para el número complejo $z = 1 + 0,5i$
 - Calcular z^2, z^3, z^4, z^5 .
 - Calcular los complejos conjugados de z, z^2 y z^3 .
 - Escribir un programa, utilizando formato de strings, que escriba las frases:
 - El conjugado de $z=1+0.5j$ es $1-0.5j$
 - El conjugado de $z=(1+0.5j)^2$ es (con el valor correspondiente)
-

CAPÍTULO 3

Clase 2: Tipos de datos y control

3.1 Escenas del capítulo anterior:

En la clase anterior preparamos la infraestructura:

- Instalamos los programas y paquetes necesarios.
- Aprendimos como ejecutar: una consola usual, de ipython, o iniciar un *notebook*
- Aprendimos a utilizar la consola como una calculadora
- Aprendimos a utilizar comandos mágicos y enviar algunos comandos al sistema operativo
- Aprendimos como obtener ayuda
- Iniciamos los primeros pasos del lenguaje

Veamos un ejemplo completo de un programa (semi-trivial):

```
# Definición de los datos
r = 9.
pi = 3.14159
#
# Cálculos
A = pi*r**2
As = 4 * A
V = 4*A*r/3
#
# Salida de los resultados
print("Para un círculo de radio {} cm, el área es {:.3f} cm2".format(r,A))
print("Para una esfera de radio {} cm, el área es {:.2f} cm2".format(r,As))
print("Para una esfera de radio {} cm, el volumen es {:.2f} cm3".format(r,V))
```

En este ejemplo simple, definimos algunas variables (`r` y `pi`), realizamos cálculos y sacamos por pantalla los resultados. A diferencia de otros lenguajes, python no necesita una estructura rígida, con definición de un programa principal (`main`).

3.2 Tipos de variables

Si vamos a discutir los distintos tipos de variables debemos asegurarnos que todos tenemos una idea (parecida) de qué es una variable.

Declaración, definición y asignación de valores a variables

3.2.1 Tipos simples

- Números enteros:
- Números Enteros
- Números Reales o de punto flotante
- Números Complejos

3.2.2 Disgresión: Objetos

En python, la forma de tratar datos es mediante *objetos*. Todos los objetos tienen, al menos:

- un tipo,
- un valor,
- una identidad.

Además, pueden tener:

- componentes
- métodos

Los *métodos* son funciones que pertenecen a un objeto y cuyo primer argumento es el objeto que la posee. Veamos algunos ejemplos cotidianos:

```
a = 3                                     # Números enteros
print(type(a))
a.bit_length()
```

```
<class 'int'>
```

```
2
```

```
a = 12312
print(type(a))
a.bit_length()
```

```
<class 'int'>
```

```
14
```

En estos casos, usamos el método `bit_length` de los enteros, que nos dice cuántos bits son necesarios para representar un número.

```
# bin nos da la representación en binarios
print(bin(3))
print(bin(a))
```

```
0b11
0b11000000011000
```

Los números de punto flotante también tienen algunos métodos definidos. Por ejemplo podemos saber si un número flotante corresponde a un entero:

```
b = -3.0
b.is_integer()
```

```
True
```

```
c = 142.25
c.is_integer()
```

```
False
```

o podemos expresarlo como el cociente de dos enteros, o en forma hexadecimal

```
c.as_integer_ratio()
```

```
(569, 4)
```

```
s= c.hex()
print(s)
```

```
0x1.1c8000000000p+7
```

Acá la notación, compartida con otros lenguajes (C, Java), significa:

```
[sign] ['0x'] integer ['. fraction] ['p' exponent]
```

Entonces 0x1.1c8p+7 corresponde a:

```
(1 + 1./16 + 12./16**2 + 8./16**3)*2.0**7
```

```
142.25
```

Veamos como último ejemplo, los números complejos

```
z = 1 + 2j
zc = z.conjugate()          # Método que devuelve el conjugado
zr = z.real                 # Componente, parte real
zi = z.imag                 # Componente, parte imaginaria
```

```
print(z, zc, zr, zi, zc.imag)
```

```
(1+2j) (1-2j) 1.0 2.0 -2.0
```

3.2.3 Strings: Secuencias de caracteres

Una cadena o *string* es una **secuencia** de caracteres (letras, números, símbolos).

Se pueden definir con comillas, comillas simples, o tres comillas (simples o dobles). Comillas simples o dobles producen el mismo resultado. Sólo debe asegurarse que se utilizan el mismo tipo para abrir y para cerrar el *string*.

Tres comillas (simples o dobles) sirven para incluir una cadena de caracteres en forma textual, incluyendo saltos de líneas.

Operaciones

En **Python** ya hay definidas algunas operaciones como suma (composición o concatenación), producto por enteros (repetición).

```
saludo = 'Hola Mundo'          # Definición usando comillas simples
saludo2 = "Hola Mundo"         # Definición usando comillas dobles
```

Los *strings* se pueden definir **equivalentemente** usando comillas simples o dobles. De esta manera es fácil incluir comillas dentro de los *strings*

```
otro= "that's all"
dijo = 'Él dijo: "hola" y yo no dije nada'
```

```
otro
```

```
"that's all"
```

```
dijo
```

```
'Él dijo: "hola" y yo no dije nada'
```

```
mas = "Finalmente, yo dije: \"Hola\" también"
```

```
mas
```

```
'Finalmente, yo dije: "Hola" también'
```

```
otromas = 'pSS€→"\\"oó@ñ'
```

```
otromas
```

```
'pSS€→" \"oó@ñ'
```

Para definir *strings* que contengan más de una línea, manteniendo el formato se pueden utilizar tres comillas (dobles o simples):

```
Texto_largo = '''Aquí me pongo a cantar
Al compás de la vigüela,
Que el hombre que lo desvela
Una pena extraordinaria
Como la ave solitaria
Con el cantar se consuela.'''

```

Podemos imprimir los strings

```
print (saludo, '\n')
print (Texto_largo, '\n')
print (otro)
```

Hola Mundo

Aquí me pongo a cantar
Al compás de la vigüela,
Que el hombre que lo desvela
Una pena estraordinaria
Como la ave solitaria
Con el cantar se consuela.

that's all

Podemos imprimir varios strings simplemente usándolos como argumentos a la función print, o sumándolos

```
print (saludo, "+", otro)
print (saludo, otro)
print (saludo + ' ' + otro + '\n') # Suma de strings
```

Hola Mundo + that's all
Hola Mundo that's all
Hola Mundo that's all

```
a = '1'  
b = 1
```

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

```
1 <class 'str'>
1 <class 'int'>
```

```
print (2*a)  
print (2*b)
```

11
?

La longitud de una cadena de caracteres se puede calcular con la función `len()`

```
print ('longitud del saludo =', len(saludo), 'caracteres')
```

longitud **del** saludo = 10 caracteres

```
n = int((30-len(saludo)//2))
print ((n-4)* '<', "Centrado manual simple", (n-4)* '>')
print (n*' *', saludo, n*' *)
```

Métodos de Strings

Los *strings* poseen varias cualidades y funcionalidades. Por ejemplo:

- Se puede iterar sobre ellos, o quedarse con una parte (slicing)
- Tienen métodos (funciones que se aplican a su *dueño*)

Veamos en primer lugar cómo se hace para seleccionar parte de un *string*

```
s = "0123456789"
print ('Primer caracter :', s[0])
print ("Segundo caracter :", s[1])
print ('Los tres primeros:', s[0:3])
print ('Todos a partir del tercero:', s[3:])
print ('Los últimos dos :', s[-2:])
print ('Todos menos los últimos dos:', s[:-2])
```

```
Primer caracter : 0
Segundo caracter : 1
Los tres primeros: 012
Todos a partir del tercero: 3456789
Los últimos dos : 89
Todos menos los últimos dos: 01234567
```

```
print(s)
print (s[:5] + s[-2:])
print(s[0:5:2])
print (s[::-2])
print (s[::-1])
print (s[::-3])
```

```
0123456789
0123489
024
02468
9876543210
9630
```

Veamos cómo se puede operar sobre un *string*:

```
a = "La mar estaba serena!"
print(a)
```

```
La mar estaba serena!
```

Por ejemplo, en python es muy fácil reemplazar una cadena por otra:

```
b = a.replace('e', 'a')
print(b)
```

```
La mar astaba sarana!
```

o separar las palabras:

```
print(b.split())
```

```
[ 'La', 'mar', 'astaba', 'sarana!' ]
```

Estos son métodos que tienen definidos los *strings*.

Un método es una función que está definida junto con el objeto. En este caso el string. Hay más información sobre los métodos de las cadenas de caracteres en: [String Methods](#)

Veamos algunos ejemplos más:

```
a = 'Hola Mundo!'
b = "Somos los colectiveros que cumplimos nuestro deber!"
c = Texto_largo
print ('\n', "Programa 0 en cualquier lenguaje:\n\t\t\t" + a, '\n')
print (80*'-')
print ('Otro texto:', b, sep='\n')
print ('Longitud del texto: ', len(b), 'caracteres')
```

```
Programa 0 en cualquier lenguaje:
Hola Mundo!
```

```
-----
Otro texto:
Somos los colectiveros que cumplimos nuestro deber!
Longitud del texto: 51 caracteres
```

Buscar y reemplazar cosas en un string:

```
b.find('l')
```

```
6
```

```
b.find('l', 7)
```

```
12
```

```
b.find('le')
```

```
12
```

```
help(b.find)
```

```
Help on built-in function find:
```

```
find(...) method of builtins.str instance
S.find(sub[, start[, end]]) -> int

Return the lowest index in S where substring sub is found,
such that sub is contained within S[start:end]. Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.
```

```
print (b.replace('que','y')) # Reemplazamos un substring
print (b.replace('e','u',2)) # Reemplazamos un substring sólo 2 veces
```

Clases de Python

```
Somos los colectiveros y cumplimos nuestro deber!
Somos los coluctivuros que cumplimos nuestro deber!
```

```
# Un ejemplo que puede interesarnos un poco más:
label = " = T/ t + u "
print('tipo de label: ', type(label))
print ('Resultados corresponden a:', label, ' (en m/s2)')
```

```
tipo de label: <class 'str'>
Resultados corresponden a: = T/ t + u (en m/s2)
```

Formato de strings

En python el formato de strings se realiza con el método `format()`. Esta función busca en el strings las llaves y las reemplaza por los argumentos. Veamos esto con algunos ejemplos:

```
a = 2019
m = 'Feb'
d = 11
s = "Hoy es el día {} de {} de {}".format(d, m, a)
print(s)
print("Hoy es el día {}/{}/{}".format(d,m,a))
print("Hoy es el día {0}/{1}/{2}".format(d,m,a))
print("Hoy es el día {2}/{1}/{0}".format(d,m,a))
```

```
Hoy es el día 11 de Feb de 2019
Hoy es el día 11/Feb/2019
Hoy es el día 11/Feb/2019
Hoy es el día 2019/Feb/11
```

```
fname = "datos-{}-{}-{}".format(a,m,d)
print(fname)
```

```
datos-2019-Feb-11
```

```
pi = 3.141592653589793
s1 = "El valor de es {}".format(pi)
s2 = "El valor de con cuatro decimales es {:.4f}".format(pi)
s3 = "El valor de con seis decimales es {:.6f}".format(pi)
print(s1)
print(s2)
print(s3)
print("{:03d}".format(5))
```

```
El valor de es 3.141592653589793
El valor de con cuatro decimales es 3.1416
El valor de con seis decimales es 3.141593
005
```

3.3 Ejercicios 02 (a)

1. Centrado manual de frases

- Utilizando la función `len()` centre una frase corta en una pantalla de 80 caracteres. Utilice la frase: Primer ejercicio con caracteres
 - Agregue subrayado a la frase anterior
-

3.4 Conversión de tipos

Como comentamos anteriormente, y se ve en los ejemplos anteriores, uno no define el tipo de variable *a-priori* sino que queda definido al asignársele un valor (por ejemplo `a=3` define a como una variable del tipo entero).

Si bien **Python** hace la conversión de tipos de variables en algunos casos, **no hace magia**, no puede adivinar nuestra intención si no la explicitamos.

```
a = 3                      # a es entero
b = 3.1                     # b es real
c = 3 + 0j                   # c es complejo
print ("a es de tipo {0}\nb es de tipo {1}\n{c} es de tipo {2}".format(type(a), type(b),
→ type(c)))
print ("'a + b' es de tipo {0} y 'a + c' es de tipo {1}".format(type(a+b), type(a+c)))
```

```
a es de tipo <class 'int'>
b es de tipo <class 'float'>
c es de tipo <class 'complex'>
'a + b' es de tipo <class 'float'> y 'a + c' es de tipo <class 'complex'>
```

```
print (1+'1')
```

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

<ipython-input-44-9b48e3080d7d> in <module>
----> 1 print (1+'1')
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Sin embargo, si le decimos explícitamente qué conversión queremos, todo funciona bien

```
print (str(1) + '1')
print (1 + int('1'))
print (1 + float('1.e5'))
```

```
11
2
100001.0
```

```
# a menos que nosotros **nos equivoquemos explícitamente**
print (1 + int('z'))
```

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

<ipython-input-46-61ad22f41838> in <module>
      1 # a menos que nosotros **nos equivoquemos explícitamente**
----> 2 print (1 + int('z'))

ValueError: invalid literal for int() with base 10: 'z'
```

3.5 Ejercicios 02 (b)

2. Para la cadena de caracteres:

```
s = '''Aquí me pongo a cantar
Al compás de la vigüela,
Que el hombre que lo desvela
Una pena estraordinaria
Como la ave solitaria
Con el cantar se consuela.'''
```

- Cuente la cantidad de veces que aparecen los substrings es, la, que, co, en los siguientes dos casos: distinguiendo entre mayúsculas y minúsculas, y no distinguiendo. Imprima el resultado.
- Cree una lista, donde cada elemento es una línea del string s y encuentre la de mayor longitud. Imprima por pantalla la línea y su longitud.
- Forme un nuevo string de 10 caracteres que contenga los 5 primeros y los 5 últimos del string anterior s. Imprima por pantalla el nuevo string.
- Forme un nuevo string que contenga los 10 caracteres centrales de s (utilizando un método que pueda aplicarse a otros strings también). Imprima por pantalla el nuevo string.
- Cambie todas las letras m por n y todas las letras n por m en s. Imprima el resultado por pantalla.
- Debe entregar un programa llamado 02_SuApellido.py (con su apellido, no la palabra SuApellido). El programa al correrlo con el comando python3 SuApellido_02.py debe imprimir:

```
Nombre Apellido
Clase 2
Distinguiendo: 2 5 1 2
Sin distinguir: 2 5 2 4
Que el hombre que lo desvela : longitud=28
Aquí uela.
desvela
Un
Aquí ne pomgo a camtar
Al compás de la vigüela,
Que el honbre que lo desvela
Uma pema estraordimaria
```

(continué en la próxima página)

(proviene de la página anterior)

Cono la ave solitaria
Com el camtar se comsuela.

3.6 Tipos contenedores: Listas

Las listas son tipos compuestos (pueden contener más de un valor). Se definen separando los valores con comas, encerrados entre corchetes. En general las listas pueden contener diferentes tipos, y pueden no ser todos iguales, pero suelen utilizarse con ítems del mismo tipo.

- Los elementos no son necesariamente homogéneos en tipo
- Elementos ordenados
- Acceso mediante un índice
- Están definidas operaciones entre Listas, así como algunos métodos
 - `x in L` (fx es un elemento de L?)
 - `x not in L` (fx no es un elemento de L?)
 - `L1 + L2` (concatenar L1 y L2)
 - `n*L1` (n veces L1)
 - `L1*n` (n veces L1)
 - `L[i]` (Elemento i-ésimo)
 - `L[i:j]` (Elementos i a j)
 - `L[i:j:k]` (Elementos i a j, elegidos uno de cada k)
 - `len(L)` (longitud de L)
 - `min(L)` (Mínimo de L)
 - `max(L)` (Máximo de L)
 - `L.index(x, [i])` (Índice de x, iniciando en i)
 - `L.count(x)` (Número de veces que aparece x en L)
 - `L.append(x)` (Agrega el elemento x al final)

Veamos algunos ejemplos:

```
cuadrados = [1, 9, 16, 25]
```

En esta línea hemos declarado una variable llamada `cuadrados`, y le hemos asignado una lista de cuatro elementos. En algunos aspectos las listas son muy similares a los *strings*. Se pueden realizar muchas de las mismas operaciones en strings, listas y otros objetos sobre los que se pueden iterar (*iterables*).

Las listas pueden accederse por posición y también pueden rebanarse (*slicing*)

Nota: La indexación de iteradores empieza desde cero (como en C)

Clases de Python

```
cuadrados[0]
```

```
1
```

```
cuadrados[3]
```

```
25
```

```
cuadrados[-1]
```

```
25
```

```
cuadrados[:3:2]
```

```
[1, 16]
```

```
cuadrados[-2:]
```

```
[16, 25]
```

Como se ve los índices pueden ser positivos (empezando desde cero) o negativos empezando desde -1.

cuadrados:	1	9	16	25
índices:	0	1	2	3
índices negativos:	-4	-3	-2	-1

Nota: La asignación entre listas **no copia**

```
a = cuadrados
a is cuadrados
```

```
True
```

```
print(a)
cuadrados[0] = -1
print(a)
print(cuadrados)
```

```
[1, 9, 16, 25]
[-1, 9, 16, 25]
[-1, 9, 16, 25]
```

```
a is cuadrados
```

```
True
```

```
b = cuadrados.copy()
print(b)
```

(continué en la próxima página)

(proviene de la página anterior)

```
print(cuadrados)
cuadrados[0]==2
print(b)
print(cuadrados)
```

```
[-1, 9, 16, 25]
[-1, 9, 16, 25]
[-1, 9, 16, 25]
[-2, 9, 16, 25]
```

3.6.1 Operaciones sobre listas

Veamos algunas operaciones que se pueden realizar sobre listas. Por ejemplo, se puede fácilmente:

- concatenar dos listas,
- buscar un valor dado,
- agregar elementos,
- borrar elementos,
- calcular su longitud,
- invertirla

Empecemos concatenando dos listas, usando el operador suma

```
L1 = [0, 1, 2, 3, 4, 5]
```

```
L = 2*L1
```

```
L
```

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

```
2*L == L + L
```

```
True
```

```
L.index(3) # Índice del elemento de valor 3
```

```
3
```

```
L.index(3,4) # Índice del valor 3, empezando del cuarto
```

```
9
```

```
L.count(3) # Cuenta las veces que aparece el valor "3"
```

```
2
```

Las listas tienen definidos métodos, que podemos ver con la ayuda incluida, por ejemplo haciendo `help(list)`

```
help(list)
```

Help on class list in module builtins:

```
class list(object)
| list(iterable=(), /)
|
| Built-in mutable sequence.
|
| If no argument is given, the constructor creates a new empty list.
| The argument must be an iterable if specified.
|
| Methods defined here:
|
|     __add__(self, value, /)
|         Return self+value.
|
|     __contains__(self, key, /)
|         Return key in self.
|
|     __delitem__(self, key, /)
|         Delete self[key].
|
|     __eq__(self, value, /)
|         Return self==value.
|
|     __ge__(self, value, /)
|         Return self>=value.
|
|     __getattribute__(self, name, /)
|         Return getattr(self, name).
|
|     __getitem__(...)
|         x.__getitem__(y) <==> x[y]
|
|     __gt__(self, value, /)
|         Return self>value.
|
|     __iadd__(self, value, /)
|         Implement self+=value.
|
|     __imul__(self, value, /)
|         Implement self*=value.
|
|     __init__(self, /, args, **kwargs)
|         Initialize self. See help(type(self)) for accurate signature.
|
|     __iter__(self, /)
|         Implement iter(self).
|
|     __le__(self, value, /)
|         Return self<=value.
|
```

```
| __len__(self, /)
|     Return len(self).
|
| __lt__(self, value, /)
|     Return self<value.
|
| __mul__(self, value, /)
|     Return self*value.
|
| __ne__(self, value, /)
|     Return self!=value.
|
| __repr__(self, /)
|     Return repr(self).
|
| __reversed__(self, /)
|     Return a reverse iterator over the list.
|
| __rmul__(self, value, /)
|     Return value*self.
|
| __setitem__(self, key, value, /)
|     Set self[key] to value.
|
| __sizeof__(self, /)
|     Return the size of the list in memory, in bytes.
|
| append(self, object, /)
|     Append object to the end of the list.
|
| clear(self, /)
|     Remove all items from list.
|
| copy(self, /)
|     Return a shallow copy of the list.
|
| count(self, value, /)
|     Return number of occurrences of value.
|
| extend(self, iterable, /)
|     Extend list by appending elements from the iterable.
|
| index(self, value, start=0, stop=9223372036854775807, /)
|     Return first index of value.
|
| Raises ValueError if the value is not present.
|
| insert(self, index, object, /)
|     Insert object before index.
|
| pop(self, index=-1, /)
|     Remove and return item at index (default last).
|
| Raises IndexError if list is empty or index is out of range.
```

```
/ remove(self, value, /)
/     Remove first occurrence of value.
/
/ Raises ValueError if the value is not present.
/
/ reverse(self, /)
/     Reverse *IN PLACE.
/
/ sort(self, /, , key=None, reverse=False)
/     Stable sort *IN PLACE.
/
-----
| Static methods defined here:
|
| __new__(*args, **kwargs) from builtins.type
|     Create and return a new object. See help(type) for accurate_
|     signature.
|
-----
| Data and other attributes defined here:
|
| __hash__ = None
```

Si queremos agregar un elemento al final utilizamos el método `append`:

```
print(L)
```

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

```
L.append(8)
```

```
print(L)
```

```
[0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 8]
```

```
L.append([9, 8, 7])
print(L)
```

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

Si queremos insertar un elemento en una posición que no es el final de la lista, usamos el método `insert()`. Por ejemplo para insertar el valor 6 en la primera posición:

```
L.insert(0, 6)
print(L)
```

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

```
L.insert(7, 6)
print(L)
```

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

```
L.insert(-2,6)
print(L)
```

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

En las listas podemos sobreescribir uno o más elementos

```
L[0:3] = [2,3,4]
print(L)
```

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

```
L[-2:] =[0,1]
print(L)
```

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

```
print(L)
L.remove(3) # Remueve la primera ocurrencia de 3
print(L)
```

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

3.6.2 Función (y tipo) range

Hay un tipo de variable llamado `range`. Se crea mediante cualquiera de los siguientes llamados:

```
range(stop)
range(start, stop, step)
```

```
range(2)
```

```
range(0, 2)
```

```
type(range(2))
```

```
range
```

```
range(0,2)
```

```
range(0, 2)
```

```
list(range(2,9))
```

```
[2, 3, 4, 5, 6, 7, 8]
```

```
list(range(2, 9, 2))
```

```
[2, 4, 6, 8]
```

3.6.3 Comprensión de Listas

Una manera sencilla de definir una lista es utilizando algo que se llama *Comprensión de listas*. Como primer ejemplo veamos una lista de *números cuadrados* como la que escribimos anteriormente. En lenguaje matemático la definiríamos como $S = \{x^2 : x \in \{0 \dots 9\}\}$. En python es muy parecido.

Podemos crear la lista cuadrados utilizando compresiones de listas

```
cuadrados = [i**2 for i in range(10)]  
cuadrados
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Una lista con los cuadrados sólo de los números pares también puede crearse de esta manera, ya que puede incorporarse una condición:

```
L = [a**2 for a in range(2, 21) if a % 2 == 0]  
L
```

```
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
```

```
sum(L)
```

```
1540
```

```
list(reversed(L))
```

```
[400, 324, 256, 196, 144, 100, 64, 36, 16, 4]
```

3.7 Módulos

Los módulos son el mecanismo de Python para reusar código. Además, ya existen varios módulos que son parte de la biblioteca *standard*. Su uso es muy simple, para poder aprovecharlo necesitaremos saber dos cosas:

- Qué funciones están ya definidas y listas para usar
- Cómo acceder a ellas

Empecemos con la segunda cuestión. Para utilizar las funciones debemos *importarlas* en la forma `import modulo`, donde modulo es el nombre que queremos importar.

Esto nos lleva a la primera cuestión: cómo saber ese nombre, y qué funciones están disponibles. La respuesta es: **la documentación**.

Una vez importado, podemos utilizar constantes y funciones definidas en el módulo con la notación de punto: `modulo.funcion()`.

3.7.1 Módulo math

El módulo **math** contiene las funciones más comunes (trigonométricas, exponenciales, logaritmos, etc) para operar sobre números de *punto flotante*, y algunas constantes importantes (pi, e, etc). En realidad es una interface a la biblioteca math en C.

```
import math
# algunas constantes y funciones elementales
raiz5pi= math.sqrt(5*math.pi)
print (raiz5pi, math.floor(raiz5pi), math.ceil(raiz5pi))
print (math.e, math.floor(math.e), math.ceil(math.e))
# otras funciones elementales
print (math.log(1024,2), math.log(27,3))
print (math.factorial(7), math.factorial(9), math.factorial(10))
print ('Combinatorio: C(6,2):',math.factorial(6)/(math.factorial(4)*math.
    ↪factorial(2)))
```

```
3.963327297606011 3 4
2.718281828459045 2 3
10.0 3.0
5040 362880 3628800
Combinatorio: C(6,2): 15.0
```

A veces, sólo necesitamos unas pocas funciones de un módulo. Entonces para abreviar la notación combiene importar sólo lo que vamos a usar, usando la notación:

```
from xxx import yyy

from math import sqrt, pi, log
import math
raiz5pi = sqrt(5*pi)
print (log(1024, 2))
print (raiz5pi, math.floor(raiz5pi))
```

```
10.0
3.963327297606011 3
```

```
import math as m
m.sqrt(3.2)
```

```
1.7888543819998317
```

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

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

<ipython-input-87-1158665f2c67> in <module>
      1 import math
----> 2 print(math.sqrt(-1))
```

```
ValueError: math domain error
```

3.7.2 Módulo cmath

Para trabajar con números complejos este módulo no es adecuado, para ello existe el módulo **cmath**

```
import cmath
print('Usando cmath (-1)^0.5: ', cmath.sqrt(-1))
print(cmath.cos(cmath.pi/3 + 2j))
```

```
Usando cmath (-1)^0.5:  1j
(1.8810978455418161-3.1409532491755083j)
```

Si queremos calcular la fase (el ángulo que forma con el eje x) podemos usar la función `phase`

```
z = 1 + 0.5j
cmath.phase(z)          # Resultado en radianes
```

```
0.4636476090008061
```

```
math.degrees(cmath.phase(z))    # Resultado en grados
```

```
26.56505117707799
```

3.8 Ejercicios 02 (c)

3. Manejos de listas:

- Cree la lista **N** de longitud 50, donde cada elemento es un número entero de 1 a 50 inclusive (Ayuda: vea la expresión `range`).
- Invierta la lista.
- Extraiga una lista **N2** que contenga sólo los elementos pares de **N**.
- Extraiga una lista **N3** que contenga sólo aquellos elementos que sean el cuadrado de algún número entero.

4. Cree una lista de la forma `L = [1, 3, 5, ..., 17, 19, 19, 17, ..., 3, 1]`

5. Operación rara sobre una lista:

- Defina la lista `L = [0, 1]`
- Realice la operación `L.append(L)`
- Ahora imprima `L`, e imprima el último elemento de `L`.
- Haga que una nueva lista `L1` que tenga el valor del último elemento de `L` y repita el inciso anterior.

6. Utilizando el string: `python s1='En un lugar de la Mancha de cuyo nombre no quiero acordarme'` y utilizando los métodos de strings:

- Obtenga la cantidad de caracteres.
- Imprima la frase anterior pero con cada palabra empezando en mayúsculas.
- Cuente cuantas letras a tiene la frase, ¿cuántas vocales tiene?
- Imprima el string `s1` centrado en una línea de 80 caracteres, rodeado de guiones en la forma:

-----En un lugar de la Mancha de cuyo nombre no quiero acordarme-----

- Obtenga una lista **L1** donde cada elemento sea una palabra.
 - Cuente la cantidad de palabras en **s1** (utilizando python).
 - Ordene la lista **L1** en orden alfabético.
 - Ordene la lista **L1** tal que las palabras más cortas estén primero.
 - Ordene la lista **L1** tal que las palabras más largas estén primero.
 - Construya un string **s2** con la lista del resultado del punto anterior.
 - Encuentre la palabra más larga y la más corta de la frase.
7. Escriba un script que encuentre las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$. Los valores de los parámetros defínalos en el mismo script, un poco más arriba.
8. Considere un polígono regular de N lados inscripto en un círculo de radio unidad:
- Calcule el ángulo interior del polígono regular de N lados (por ejemplo el de un triángulo es 60 grados, de un cuadrado es 90 grados, y de un pentágono es 108 grados). Exprese el resultado en grados y en radianes para valores de $N = 3, 5, 6, 8, 9, 10, 12$.
 - ¿Puede calcular la longitud del lado de los polígonos regulares si se encuentran inscriptos en un círculo de radio unidad?
9. Escriba un *script* (llamado *distancial.py*) que defina las variables velocidad y posición inicial v_0 , z_0 , la aceleración g , y la masa $m = 1$ kg a tiempo $t = 0$, y calcule e imprima la posición y velocidad a un tiempo posterior t . Ejecute el programa para varios valores de posición y velocidad inicial para $t = 2$ segundos. Recuerde que las ecuaciones de movimiento con aceleración constante son:

$$\begin{aligned}v &= v_0 - gt \\z &= z_0 + v_0 t - \frac{gt^2}{2}.\end{aligned}$$

3.8.1 Adicionales

11. Calcular la suma:

$$s_1 = \frac{1}{2} \left(\sum_{k=0}^{100} k \right)^{-1}$$

Ayuda: busque información sobre la función `sum()`

12. Construir una lista **L2** con 2000 elementos, todos iguales a 0.0005. Imprimir su suma utilizando la función `sum` y comparar con la función que existe en el módulo `math` para realizar suma de números de punto flotante.
-

CAPÍTULO 4

Clase 3: Control de flujo y tipos complejos

4.1 Control de flujo

4.1.1 if/elif/else

En todo lenguaje necesitamos controlar el flujo de una ejecución segun una condición Verdadero/Falso (booleana). *Si (condición) es verdadero hacé (bloque A); Sino hacé (Bloque B)*. En pseudo código:

```
Si condición 1:  
    bloque A  
sino y condición 2:  
    bloque B  
sino:  
    bloque B
```

y en Python es muy parecido!

```
if condición_1:  
    bloque A  
elif condicion_2:  
    bloque B  
elif condicion_3:  
    bloque C  
else:  
    Bloque final
```

En un `if`, la conversión a tipo `boolean` es implícita. El tipo `None` (vacío), el `0`, una secuencia (lista, tupla, string) (o conjunto o diccionario, que ya veremos) vacía siempre evalua a `False`. Cualquier otro objeto evalua a `True`.

Podemos tener multiples condiciones. Se ejecutará el primer bloque cuya condición sea verdadera, o en su defecto el bloque `else`. Esto es equivalente a la sentencia `switch` de otros lenguajes.

```
Nota = 7  
if Nota >= 8:
```

(continué en la próxima página)

(proviene de la página anterior)

```
print ("Aprobó cómodo, felicidades!")
elif 6 <= Nota < 8:
    print ("Bueno, al menos aprobó!")
elif 4 <= Nota < 6 :
    print ("Bastante bien, pero no le alcanzó")
else:
    print("Debe esforzarse más!")
```

```
Bueno, al menos aprobó!
```

4.1.2 Iteraciones

Sentencia for

Otro elemento de control es el que permite *iterar* sobre una secuencia (o *iterador*). Obtener cada elemento para hacer algo. En Python se logra con la sentencia `for`. En lugar de iterar sobre una condición aritmética hasta que se cumpla una condición (como en C o en Fortran) en Python la sentencia `for` itera sobre los ítems de una secuencia en forma ordenada

```
for elemento in range(10):
    print(elemento, end=', ')
```

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

Veamos otro ejemplo:

```
Lista = ['auto', 'casa', "perro", "gato", "árbol", "lechuza"]
for L in Lista:
    print(L.count("a"), len(L))
    print(L)
```

```
1 4
auto
2 4
casa
0 5
perro
1 4
gato
0 5
árbol
1 7
lechuza
```

```
suma = 0
for elemento in range(11):
    suma += elemento
    print("x={}, suma parcial={}".format(elemento, suma))
print ('Suma total =', suma)
```

```
x=0, suma parcial=0
x=1, suma parcial=1
```

(continué en la próxima página)

(proviene de la página anterior)

```
x=2,    suma parcial=3
x=3,    suma parcial=6
x=4,    suma parcial=10
x=5,    suma parcial=15
x=6,    suma parcial=21
x=7,    suma parcial=28
x=8,    suma parcial=36
x=9,    suma parcial=45
x=10,   suma parcial=55
Suma total = 55
```

Notar que utilizamos el operador asignación de suma: `+=`.

```
suma += elemento
```

es equivalente a:

```
suma = suma + elemento
```

que corresponde a realizar la suma de la derecha, y el resultado asignarlo a la variable de la izquierda.

Por supuesto, para obtener la suma anterior podemos simplemente usar las funciones de python:

```
print (sum(range(11))) # El ejemplo anterior puede escribirse usando sum y range
```

```
55
```

Loops: for, enumerate, continue, break, else

Veamos otros ejemplos de uso del bloque `for`.

```
suma = 0
cuadrados = []
for i,elem in enumerate(range(3,21)):
    if elem % 2:      # Si resto (%) es diferente de cero -> Impares
        continue
    suma += elem**2
    cuadrados.append(elem**2)
    print (i, elem, elem**2, suma)    # Imprimimos el índice y el elem al cuadrado
print ("sumatoria de números pares al cuadrado entre 3 y 20:", suma)
print ('cuadrados= ', cuadrados)
```

```
1 4 16 16
3 6 36 52
5 8 64 116
7 10 100 216
9 12 144 360
11 14 196 556
13 16 256 812
15 18 324 1136
17 20 400 1536
sumatoria de números pares al cuadrado entre 3 y 20: 1536
cuadrados= [16, 36, 64, 100, 144, 196, 256, 324, 400]
```

```
suma = 0
cuadrados = []
for i,elem in enumerate(range(3,21)):
    if elem % 2 == 0:          # Si resto (%) es diferente de cero -> Impares
        suma += elem**2
        cuadrados.append(elem**2)
    print (i, elem, elem**2, suma)  # Imprimimos el índice y el elem al cuadrado
print ("sumatoria de números pares al cuadrado entre 3 y 20:", suma)
print ('cuadrados= ', cuadrados)
```

```
1 4 16 16
3 6 36 52
5 8 64 116
7 10 100 216
9 12 144 360
11 14 196 556
13 16 256 812
15 18 324 1136
17 20 400 1536
sumatoria de números pares al cuadrado entre 3 y 20: 1536
cuadrados= [16, 36, 64, 100, 144, 196, 256, 324, 400]
```

Puntos a notar:

- Inicializamos una variable entera en cero y una lista vacía
- range(3,30) nos da consecutivamente los números entre 3 y 29 en cada iteración.
- enumerate nos permite iterar sobre algo, agregando un contador automático.
- La línea condicional if elem% 2: es equivalente a if (elem% 2) != 0: y es verdadero si elem no es divisible por 2 (número impar)
- La sentencia continue hace que se omita la ejecución del resto del bloque por esta iteración
- El método append agrega el elemento a la lista

Antes de seguir veamos otro ejemplo de uso de enumerate. Consideremos una iteración sobre una lista como haríamos normalmente en otros lenguajes:

```
L = "I've had a perfectly wonderful evening. But this wasn't it.".split()
```

```
L
```

```
["I've",
 'had',
 'a',
 'perfectly',
 'wonderful',
 'evening.',
 'But',
 'this',
 "wasn't",
 'it.']}
```

```
for j in range(len(L)):
    print('Índice: {} -> {} ({}) caracteres'.format(j, L[j], len(L[j])))
```

```
Índice: 0 -> I've (4 caracteres)
Índice: 1 -> had (3 caracteres)
Índice: 2 -> a (1 caracteres)
Índice: 3 -> perfectly (9 caracteres)
Índice: 4 -> wonderful (9 caracteres)
Índice: 5 -> evening. (8 caracteres)
Índice: 6 -> But (3 caracteres)
Índice: 7 -> this (4 caracteres)
Índice: 8 -> wasn't (6 caracteres)
Índice: 9 -> it. (3 caracteres)
```

Si bien esta es una solución al problema. Python ofrece la función enumerate que agrega un contador automático

```
for ind, elem in enumerate(L):
    print('Índice: {} -> {} ({}) caracteres'.format(ind, elem, len(elem)))
```

```
Índice: 0 -> I've (4 caracteres)
Índice: 1 -> had (3 caracteres)
Índice: 2 -> a (1 caracteres)
Índice: 3 -> perfectly (9 caracteres)
Índice: 4 -> wonderful (9 caracteres)
Índice: 5 -> evening. (8 caracteres)
Índice: 6 -> But (3 caracteres)
Índice: 7 -> this (4 caracteres)
Índice: 8 -> wasn't (6 caracteres)
Índice: 9 -> it. (3 caracteres)
```

Veamos otro ejemplo, que puede encontrarse en la documentación oficial:

```
for n in range(2, 20):
    for x in range(2, n):
        # print ('valor de n,x = {},{}'.format(n,x))
        if n % x == 0:
            print(' {}:2d) = {} x {}'.format(n,x,n//x))
            break
    else:
        # Salío sin encontrar un factor, entonces ...
        print(' {}:2d) es un número primo'.format(n))
```

```
2 es un número primo
3 es un número primo
4 = 2 x 2
5 es un número primo
6 = 2 x 3
7 es un número primo
8 = 2 x 4
9 = 3 x 3
10 = 2 x 5
11 es un número primo
12 = 2 x 6
13 es un número primo
14 = 2 x 7
15 = 3 x 5
16 = 2 x 8
17 es un número primo
18 = 2 x 9
19 es un número primo
```

Puntos a notar:

- Acá estamos usando dos *loops* anidados. Uno recorre *n* entre 2 y 9, y el otro *x* entre 2 y *n*.
- La comparación `if n % x == 0`: chequea si *x* es un divisor de *n*
- La sentencia `break` interrumpe el *loop* interior (sobre *x*)
- Notar la alineación de la sentencia `else`. No está referida a `if` sino a `for`. Es opcional y se ejecuta cuando el *loop* se termina normalmente (sin `break`)

While

Otra sentencia de control es *while*: que permite iterar mientras se cumple una condición. El siguiente ejemplo imprime la serie de Fibonacci (en la cuál cada término es la suma de los dos anteriores)

```
a, b = 0, 1
while b < 5000:
    print(b, end=' ')
    a, b = b, a+b
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

4.2 Ejercicios 03 (a)

1. Imprimir los números que no son divisibles por 2, 3, 5 o 7 de los primeros 100 números naturales
2. Calcule la suma

$$s_2 = \sum_{k=1}^{\infty} \frac{(-1)^k(k+1)}{2k^3 + k^2}$$

con un error relativo estimado menor a $\epsilon = 10^{-5}$. Imprima por pantalla el resultado, el valor máximo de *k* computado y el error relativo estimado.

4.3 Tipos complejos de datos

4.3.1 Listas y tuples

Listas y tuples son colecciones *ordenadas* sobre las que se puede iterar (moverse de un elemento al siguiente), y también tienen **métodos** que simplifican muchas operaciones de uso común.

Ambos pueden contener elementos de cualquier tipo, y además no todos los elementos tienen que ser del mismo tipo. Por esta razón uno se refiere a estos tipos como **contenedores**.

Listas y tuples son bastante similares. La diferencia es que una lista puede modificarse (agregar, borrar o modificar sus elementos) y una tuple es inmutable.

Si bien en la práctica puede haber algunas diferencias de optimización, la principal diferencia es la mutabilidad o no del tipo. Desde el punto de vista del usuario podemos hacer una diferencia de modo de uso. Si bien no es un imperativo del lenguaje, las listas podemos utilizarla principalmente para agrupar datos donde cada uno de ellos tiene un valor diferente pero cumplen la misma función (datos homogéneos), como en:

```
Temp_min = [13, 12, 7, 9, 11, 9, 13, 12, 13]
Temp_max = [23, 21, 22, 24, 27, 25, 22, 28, 26]
```

donde un número, si bien diferente a otro, representa en cada caso el mismo dato (temperatura mínima o máxima) mientras que las *tuples* se utilizan para agrupar datos donde cada uno de ellos no representa lo mismo (como una versión simple de estructuras en C). Por ejemplo, podríamos guardar los datos climáticos por día

```
clima = []
clima_ayer = (13, 23, 78, "soleado", "6:30", "19:47")
clima_hoy = (12, 21, 87, "soleado", "6:30", "19:48")
clima.append(clima_ayer)
clima.append(clima_hoy)
print(clima)
```

```
[(13, 23, 78, 'soleado', '6:30', '19:47'), (12, 21, 87, 'soleado', '6:30', '19:48')]
```

Veamos algunas definiciones y ejemplos utilizando listas y tuples:

```
clima[0][4]
```

```
'6:30'
```

```
l1 = [1, 2, 3]
l2 = ['bananas', 'manzanas', 'naranjas', 'uvas']
t1 = (1, 2, 3)
t2 = (1, 2, 3, 'manzanas')
```

```
print (' tipo de l1:', type(l1), '\n tipo de t1:', type(t1))
print ('Primer elemento de l1, t1 y l2: ', l1[0], t1[0], l2[0])
print (' f11 is t1? ', l1 is t1)
```

```
tipo de l1: <class 'list'> ,
tipo de t1: <class 'tuple'>
Primer elemento de l1, t1 y l2:  1 1 bananas
f11 is t1? False
```

```
t3 = t2
print ('f3 is t2? ', t3 is t2)
l5 = list(t2)
print ('f5 is t2? ', l5 is t2)
print (t3)
print (l5)
```

```
f3 is t2? True
f5 is t2? False
(1, 2, 3, 'manzanas')
[1, 2, 3, 'manzanas']
```

```
# Tampoco son "iguales" aunque tengan los mismos elementos
print ('f15 == t2? ', l5 == t2)
print ('f15 == list(t2)? ', l5 == list(t2))
```

```
f15 == t2? False
f15 == list(t2)? True
```

Clases de Python

Mutabilidad

Como mencionamos, una diferencia importante entre listas y tuples es que las tuples son inmutables. Veamos que pasa cuando tratamos de modificar una y otra:

```
print ('l1 original: ',l1)
l1[0]=9
print ('l1 modificado: ',l1)      # Lista modificada
```

```
l1 original: [1, 2, 3]
l1 modificado: [9, 2, 3]
```

```
print ('Modificamos tuples?')
print ('t1 original: ',t1)
t1[0]= 9
```

```
Modificamos tuples?
t1 original: (1, 2, 3)
```

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

<ipython-input-48-1b4951c49fa3> in <module>
      1 print ('Modificamos tuples?')
      2 print ('t1 original: ',t1)
----> 3 t1[0]= 9

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

```
a = "Hola Mundo"
b = "Chau Mundo"
print ('a original:',a)
print ('Primer elemento:', a[0])
print ('b original:', b, ' -> id:', id(b))
b = a[:4]
print('b modificado:', b, ' -> id:', id(b))
a[0] = 'u'
print ('modificado:', a)
```

```
a original: Hola Mundo
Primer elemento: H
b original: Chau Mundo  -> id: 139848166324784
b modificado: Hola   -> id: 139848166366448
```

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

<ipython-input-49-202f10b6c062> in <module>
      6 b = a[:4]
      7 print('b modificado:', b, ' -> id:', id(b))
----> 8 a[0] = 'u'
      9 print ('modificado:', a)
```

(continué en la próxima página)

(proviene de la página anterior)

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

Nota: Esto nos dice que los *strings* son inmutables.

No se puede cambiar partes de un string (un carácter). Sin embargo se puede modificar completamente (porque lo que está haciendo es destruyéndolo y creando uno nuevo).

Como en el caso de *Strings*, a las listas y tuples se les puede calcular el número de elementos (su longitud) utilizando la función `len`. Además tiene métodos que son de utilidad, para ordenar, agregar (`append`) un elemento al final en forma eficiente, extenderla, etc.

Puede encontrarse más información en [la Biblioteca de Python](#).

4.3.2 Diccionarios

Los diccionarios son colecciones de objetos *en principio heterogéneos* que no están ordenados y no se refieren por índice (como `L[3]`) sino por un nombre o clave (llamado **key**). Las claves pueden ser cualquier objeto inmutable (cadenas, números, tuplas) y los valores pueden ser cualquier tipo de objeto. Las claves no se pueden repetir pero los valores sí.

```
d0 = {'a': 123}
```

```
d0['a']
```

```
123
```

```
d0
```

```
{'a': 123}
```

```
d1 = {'nombre': 'Juan',
      'apellido': 'García',
      'edad': 109,
      'dirección': '''Av Bustillo 9500,'''',
      'cod': 8400,
      1: ['hola', 'chau'],
      'ciudad': "Bariloche"}
```

```
print ('Nombre: ', d1['nombre'])
print ('\n' + 80*'+' + '\n\tDiccionario:')
print (d1)
```

```
Nombre: Juan
```

```
+++++++++++++++++++++
Diccionario:
{'nombre': 'Juan', 'apellido': 'García', 'edad': 109, 'dirección': 'Av Bustillo 9500,
→', 'cod': 8400, 1: ['hola', 'chau'], 'ciudad': 'Bariloche'}
```

Clases de Python

```
d1['cod']
```

```
8400
```

```
d1['tel'] = {'cel':1213, 'fijo':23848}
```

```
d1
```

```
{'nombre': 'Juan',
'apellido': 'García',
'edad': 109,
'dirección': 'Av Bustillo 9500,',
'cod': 8400,
1: ['hola', 'chau'],
'ciudad': 'Bariloche',
'tel': {'cel': 1213, 'fijo': 23848}}
```

```
d1['tel']
```

```
{'cel': 1213, 'fijo': 23848}
```

```
dd = d1['tel']
print(dd['cel'])
print(d1['tel']['cel'])
```

```
1213
```

```
1213
```

Los diccionarios pueden pensarse como pares *key, valor*. Para obtener todas las claves (*keys*), valores, o pares (clave, valor) usamos:

```
print ('\n' + 70*'+' + '\n\tkeys:')
print (list(d1.keys()))
print ('\n' + 70*'+' + '\n\tvalues:')
print (list(d1.values()))
print ('\n' + 0*'+' + '\n\titems:')
print (list(d1.items()))

+++++
keys:
['nombre', 'apellido', 'edad', 'dirección', 'cod', 1, 'ciudad', 'tel']

+++++
values:
['Juan', 'García', 109, 'Av Bustillo 9500,', 8400, ['hola', 'chau'], 'Bariloche', {
→'cel': 1213, 'fijo': 23848}]

+++++
items:
[('nombre', 'Juan'), ('apellido', 'García'), ('edad', 109), ('dirección', 'Av
→Bustillo 9500,'), ('cod', 8400), (1, ['hola', 'chau']), ('ciudad', 'Bariloche'), (
→'tel', {'cel': 1213, 'fijo': 23848})]
```

```
it = list(d1.items())
it
```

```
[('nombre', 'Juan'),
 ('apellido', 'García'),
 ('edad', 109),
 ('dirección', 'Av Bustillo 9500,'),
 ('cod', 8400),
 (1, ['hola', 'chau']),
 ('ciudad', 'Bariloche'),
 ('tel', {'cel': 1213, 'fijo': 23848})]
```

```
dict(it)
```

```
{'nombre': 'Juan',
 'apellido': 'García',
 'edad': 109,
 'dirección': 'Av Bustillo 9500,',
 'cod': 8400,
 1: ['hola', 'chau'],
 'ciudad': 'Bariloche',
 'tel': {'cel': 1213, 'fijo': 23848}}
```

```
it.append( ([a], 'hola') )
it
```

```
[('nombre', 'Juan'),
 ('apellido', 'García'),
 ('edad', 109),
 ('dirección', 'Av Bustillo 9500,'),
 ('cod', 8400),
 (1, ['hola', 'chau']),
 ('ciudad', 'Bariloche'),
 ('tel', {'cel': 1213, 'fijo': 23848}),
 (['Hola Mundo'], 'hola')]
```

```
dict(it)
```

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

<ipython-input-65-29b977625508> in <module>
----> 1 dict(it)

TypeError: unhashable type: 'list'
```

```
print ('\n' + 50*'+'+ '\n\tDatos:')
print (d1['nombre'] + ' ' + d1['apellido'])
print (d1[u'dirección'])
print (d1['ciudad'])
print ('\n' + 50*'+')
```

Clases de Python

```
+++++
 Datos:
Juan García
Av Bustillo 9500,
Bariloche
+++++
```

```
d1['pais']= 'Argentina'

d1['ciudad']= "San Carlos de Bariloche"
print ('\\n' + 70*'+'+ '\\n\\tDatos:')
print (d1['nombre'] + ' ' + d1['apellido'])
print (d1[u'dirección'])
print (d1['ciudad'])
print (d1['pais'])
```

```
+++++
 Datos:
Juan García
Av Bustillo 9500,
San Carlos de Bariloche
Argentina
+++++
```

```
d2 = {'provincia': 'Río Negro', 'nombre':'José'}
print (70*'+'+'\nOtro diccionario:')
print ('d2=',d2)
print (70*'')
```

```
*****
Otro diccionario:
d2= {'provincia': 'Río Negro', 'nombre': 'José'}
*****
```

Vimos que se pueden asignar campos a diccionarios. También se pueden completar utilizando otro diccionario.

```
print ('d1=',d1)
d1.update(d2) # Corregimos valores o agregamos nuevos si no existen
print ('d1=',d1)
print (80*'')
```

```
d1= {'nombre': 'Juan', 'apellido': 'García', 'edad': 109, 'dirección': 'Av_
→Bustillo 9500,', 'cod': 8400, 1: ['hola', 'chau'], 'ciudad': 'San Carlos de_
→Bariloche', 'tel': {'cel': 1213, 'fijo': 23848}, 'pais': 'Argentina'}
d1= {'nombre': 'José', 'apellido': 'García', 'edad': 109, 'dirección': 'Av_
→Bustillo 9500,', 'cod': 8400, 1: ['hola', 'chau'], 'ciudad': 'San Carlos_
→de Bariloche', 'tel': {'cel': 1213, 'fijo': 23848}, 'pais': 'Argentina',_
→'provincia': 'Río Negro'}
*****
```

```
# Para borrar un campo de un diccionario usamos `del`
print ('provincia' in d1)
if 'provincia' in d1:
    #print( d1['provincia'])
    del d1['provincia']
print ('provincia' in d1)
```

```
True
False
```

El método pop nos devuelve un valor y lo borra del diccionario

```
d1
```

```
{'nombre': 'José',
'apellido': 'García',
'edad': 109,
'dirección': 'Av Bustillo 9500',
'cod': 8400,
1: ['hola', 'chau'],
'ciudad': 'San Carlos de Bariloche',
'tel': {'cel': 1213, 'fijo': 23848},
'pais': 'Argentina'}
```

```
d1[1]
```

```
['hola', 'chau']
```

```
d1.pop(1)
```

```
['hola', 'chau']
```

```
d1
```

```
{'nombre': 'José',
'apellido': 'García',
'edad': 109,
'dirección': 'Av Bustillo 9500',
'cod': 8400,
'ciudad': 'San Carlos de Bariloche',
'tel': {'cel': 1213, 'fijo': 23848},
'pais': 'Argentina'}
```

```
# Como crear un diccionario vacío:
d3 = {}
print(d3, type(d3))
```

```
{} <class 'dict'>
```

4.3.3 Conjuntos

Los conjuntos (set ()) son grupos de claves únicas e inmutables.

```
mamiferos = {'perro', 'gato', 'león', 'perro'}
domesticos = {'perro', 'gato', 'gallina', 'ganso'}
aves = {"chimango", "bandurria", 'gallina', 'cónedor', 'ganso'}
```

```
mamiferos
```

Clases de Python

```
{'gato', 'león', 'perro'}
```

```
mamiferos.intersection(domesticos)
```

```
{'gato', 'perro'}
```

```
# También se puede utilizar el operador "&" para la intersección  
mamiferos & domesticos
```

```
{'gato', 'perro'}
```

```
mamiferos.union(domesticos)
```

```
{'gallina', 'ganso', 'gato', 'león', 'perro'}
```

```
# También se puede utilizar el operador "/" para la unión  
mamiferos | domesticos
```

```
{'gallina', 'ganso', 'gato', 'león', 'perro'}
```

```
aves.difference(domesticos)
```

```
{'bandurria', 'chimango', 'cónedor'}
```

```
# También se puede utilizar el operador "-" para la diferencia  
aves - domesticos
```

```
{'bandurria', 'chimango', 'cónedor'}
```

```
domesticos - aves
```

```
{'gato', 'perro'}
```

```
ll= list(aves.difference(domesticos))  
print (ll)  
ll.sort(reverse=True)  
print (ll)  
print (ll[1])
```

```
['chimango', 'bandurria', 'cónedor']  
['cónedor', 'chimango', 'bandurria']  
chimango
```

```
# Como crear un conjunto vacío. Notar que: conj = {} hubiera creado un diccionario.  
conj = set()  
print (conj, type(conj))
```

```
set() <class 'set'>
```

Modificar conjuntos

Para agregar o borrar elementos a un conjunto usamos los métodos: add, update, y remove

```
c = set([1, 2, 2, 3, 5])  
c
```

```
{1, 2, 3, 5}
```

```
c.add(4)
```

```
c
```

```
{1, 2, 3, 4, 5}
```

```
c.add(4)  
c
```

```
{1, 2, 3, 4, 5}
```

```
c.update((8, 7, 6))
```

```
c
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

```
c.remove(2)
```

```
c
```

```
{1, 3, 4, 5, 6, 7, 8}
```

```
c.remove(2)
```

```
-----  
KeyError
```

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

```
<ipython-input-95-b4d051d0e502> in <module>  
----> 1 c.remove(2)
```

```
KeyError: 2
```

```
c.discard(2)
```

```
c
```

```
{1, 3, 4, 5, 6, 7, 8}
```

4.4 Ejercicios 03 (b)

3. Cree dos listas: una con los números que no son múltiplos de ninguno de 2,7,11,13 y otra con los que no son múltiplos de 3,5,17. Considere los primeros 5000 números naturales. Cree una nueva lista donde combine las dos listas anteriores ordenada en forma creciente.
-

4.5 Escritura y lectura a archivos

Nuestros programas necesitan interactuar con el mundo exterior. Hasta ahora utilizamos la función `print()` para imprimir por pantalla mensajes y resultados. Para leer o escribir un archivo primero debemos abrirlo, utilizando la función `open()`

```
f = open('../data/names.txt') # Abrimos el archivo (para leer)
```

```
f
```

```
<_io.TextIOWrapper name='../data/names.txt' mode='r' encoding='UTF-8'>
```

```
s = f.read() # Leemos el archivo
```

```
f.close() # Cerramos el archivo
```

Básicamente esta secuencia de trabajo es la que uno utilizará siempre. Sin embargo, hay un potencial problema, que ocurrirá si hay algún error entre la apertura y el cierre del archivo. Para ello existe una sintaxis alternativa

```
with open('../data/names.txt') as f:  
    s = f.read()
```

La palabra `with` es una palabra reservada del lenguaje y la construcción se conoce como *contexto*. Básicamente dice que todo lo que está dentro del bloque se realizará en el contexto en que `f` es el objeto de archivo abierto para lectura.

4.5.1 Ejemplos

Vamos a repasar algunos de los conceptos discutidos las clases anteriores e introducir algunas nuevas funcionalidades con ejemplos

Ejemplo 03-1

```
!head ../data/names.txt # Muestro el contenido del archivo
```

```
Aaa  
Aaron  
Aba  
Ababa  
Ada  
Ada  
Adam  
Adlai
```

(continué en la próxima página)

(proviene de la página anterior)

Adrian
Adrienne

```
# %load scripts/03_ejemplo_1.py
#!/usr/bin/env python3

fname = '../data/names.txt'
n = 0                                # contador
minlen = 3                             # longitud mínima
maxlen = 4                             # longitud máxima

with open(fname, 'r') as fi:
    lines = fi.readlines()              # El resultado es una lista

for line in lines:
    if minlen <= len(line.strip()) <= maxlen:
        n += 1
        print(line.strip(), end=', ')   # No Newline

print('\n')
if minlen == maxlen:
    mensaje = 'Encontramos {} palabras que tienen {} letras'.format(n, minlen)
else:
    mensaje = 'Encontramos {} palabras que tienen entre {} y {} letras'\
        .format(n, minlen, maxlen)

print(mensaje)
```

Aaa, Aba, Ada, Ada, Adam, Ala, Alan, Alex, Alf, Ama, Ami, Amir, Amos, Amy, Ana, Andy, Ann, Anna, Anna, Anne, Anya, Arne, Art, Axel, Bart, Bea, Ben, Bert, Beth, Bib, Bill, Bob, Bob, Boob, Boyd, Brad, Bret, Bub, Bud, Carl, Cary, Case, Cdc, Chet, Chip, Clay, Clem, Cody, Cole, Cory, Cris, Curt, Dad, Dale, Dan, Dana, Dani, Dave, Dawn, Dean, Deb, Debi, Deed, Del, Dick, Did, Dion, Dirk, Dod, Don, Donn, Dora, Dori, Dory, Doug, Drew, Dud, Duke, Earl, Eddy, Eke, Eli, Elsa, Emil, Emma, Enya, Ere, Eric, Erik, Esme, Eva, Evan, Eve, Ewe, Eye, Fay, Fred, Gag, Gaia, Gail, Gale, Gary, Gay, Gene, Gig, Gigi, Gil, Gill, Glen, Gog, Greg, Guy, Hal, Hank, Hans, Harv, Hein, Herb, Hohn, Hon, Hope, Hsi, Huey, Hugh, Huh, Hui, Hume, Hurf, Hwa, Iain, Ian, Igor, Iii, Ilya, Ima, Imad, Ira, Isis, Izzy, Jack, Jade, Jan, Jane, Jarl, Jay, Jean, Jef, Jeff, Jem, Jen, Jess, Jill, Jim, Jin, Jiri, Joan, Job, Jock, Joe, Joel, John, Jon, Jong, Joni, Joon, Jos, Jose, Josh, Juan, Judy, Juha, Jun, June, Juri, Kaj, Kari, Karl, Kate, Kay, Kee, Kees, Ken, Kenn, Kent, Kiki, Kim, King, Kirk, Kit, Knut, Kory, Kris, Kurt, Kyle, Kylo, Kyu, Lana, Lar, Lara, Lars, Lea, Leah, Lee, Leif, Len, Leo, Leon, Les, Lex, Liam, Lila, Lin, Lisa, List, Liz, Liza, Lois, Lola, Lord, Lori, Lou, Loyd, Luc, Lucy, Lui, Luis, Luke, Lum, Lynn, Mac, Mah, Mann, Mara, Marc, Mark, Mary, Mat, Mats, Matt, Max, May, Mayo, Meg, Mick, Miek, Mike, Miki, Milo, Moe, Mott, Mum, Mwa, Naim, Nan, Nate, Neal, Ned, Neil, Nhan, Nici, Nick, Nils, Ning, Noam, Noel, Non, Noon, Nora, Norm, Nou, Novo, Nun, Ofer, Olaf, Old, Ole, Oleg, Olof, Omar, Otto, Owen, Ozan, Page, Pam, Pap, Part, Pat, Paul, Pdp, Peep, Pep, Per, Pete, Petr, Phil, Pia, Piet, Pim, Ping, Pip, Poop, Pop, Pria, Pup, Raif, Raj, Raja, Ralf, Ram, Rand, Raul, Ravi, Ray, Real, Rees, Reg, Reid, Rene, Renu, Rex, Ric, Rich, Rick, Rik, Rob, Rod, Rolf, Ron, Root, Rose, Ross, Roy, Rudy, Russ, Ruth, S's, Saad, Sal, Sam, Sara, Saul, Scot, Sean, Sees, Seth, Shai, Shaw, Shel, Sho, Sid, Sir, Sis, Skef, Skip, Son, Spy, Sri, Ssi, Stan, Stu, Sue, Sus, Suu, Syd, Syed, Syun, Tad, Tai, Tait, Tal, Tao, Tara, Tat, Ted, Teet, Teri, Tex, Thad, The, Theo, Tim, Timo, Tip, Tit, Tnt, Toby, Todd, Toft, Tom, Tony, Toot, Tor, Tot, Tran, Trey, Troy, Tuan, Tuna, Uma, Una, Uri, Urs, Val, Van, Vern, Vic, Vice, Vick, Wade, Walt, Wes, Will, Win, Wolf, Wow, Zoe, Zon,

(continué en la próxima página)

(provine de la página anterior)

Encontramos 420 palabras que tienen entre 3 y 4 letras

Hemos utilizado aquí:

- Apertura, lectura, y cerrado de archivos
- Iteración en un loop `for`
- Bloques condicionales (if/else)
- Formato de cadenas de caracteres con reemplazo
- Impresión por pantalla

La apertura de archivos se realiza utilizando la función `open` (este es un buen momento para mirar su documentación) con dos argumentos: el primero es el nombre del archivo y el segundo el modo en que queremos abrilo (en este caso la `r` indica lectura).

Con el archivo abierto, en la línea 9 leemos línea por línea todo el archivo. El resultado es una lista, donde cada elemento es una línea.

Recorremos la lista, y en cada elemento comparamos la longitud de la línea con ciertos valores. Imprimimos las líneas seleccionadas

Finalmente, escribimos el número total de líneas.

Veamos una leve modificación de este programa

Ejemplo 03-2

```
# %load scripts/03_ejemplo_2.py
#!/usr/bin/env python3

"""Programa para contar e imprimir las palabras de una longitud dada"""

fname = '../data/names.txt'

n = 0                      # contador
minlen = 3                  # longitud mínima
maxlen = 3                  # longitud máxima

with open(fname, 'r') as fi:
    for line in fi:
        p = line.strip().lower()
        if (minlen <= len(p) <= maxlen) and (p == p[::-1]):
            n += 1
            print('{:02d}: {}'.format(n, p), end=', ')
            # Vamos numerando las
            →coincidencias
print('\n')
if minlen == maxlen:
    mensaje = ('Encontramos un total de {} palabras capicúa que tienen {} letras'.
               format(n, minlen))
else:
    mensaje = 'Encontramos un total de {} palabras que tienen\
entre {} y {} letras'.format(n, minlen, maxlen)

print(mensaje)
```

```
(01): aaa, (02): aba, (03): ada, (04): ada, (05): ala, (06): ama, (07): ana, (08):_  
bib, (09): bob, (10): bob, (11): bub, (12): cdc, (13): dad, (14): did, (15): dod,_  
e(16): dud, (17): eke, (18): ere, (19): eve, (20): eve, (21): ewe, (22): eye, (23):_  
gag, (24): gig, (25): gog, (26): huh, (27): iii, (28): mum, (29): nan, (30): non,_  
s(31): nun, (32): pap, (33): pdp, (34): pep, (35): pip, (36): pop, (37): pup, (38): s  
's, (39): sis, (40): sus, (41): tat, (42): tit, (43): tnt, (44): tot, (45): wow,
```

Encontramos un total de 45 palabras capicúa que tienen 3 letras

Aquí en lugar de leer todas las líneas e iterar sobre las líneas resultantes, iteramos directamente sobre el archivo abierto. Además incluimos un string al principio del archivo, que servirá de documentación, y puede accederse mediante los mecanismos usuales de ayuda de Python.

Imprimimos el número de palabra junto con la palabra, usamos `02d`, indicando que es un entero (`d`), que queremos que el campo sea de un mínimo número de caracteres de ancho (en este caso 2). Al escribirlo como `02` le pedimos que complete los vacíos con ceros.

```
s = "Hola\n y chau"  
with open('tmp.txt', 'w') as fo:  
    fo.write(s)
```

```
!cat tmp.txt
```

```
Hola  
y chau
```

4.6 Ejercicios 03 (c)

4. Realice un programa que:

- Lea el archivo `names.txt`
- Guarde en un nuevo archivo (llamado `pares.txt`) palabra por medio del archivo original (la primera, tercera,) una por línea, pero en el orden inverso al leído
- Agregue al final de dicho archivo, las palabras pares pero separadas por un punto y coma (;
- En un archivo llamado `longitudes.txt` guarde las palabras ordenadas por su longitud, y para cada longitud ordenadas alfabéticamente.
- En un archivo llamado `letras.txt` guarde sólo aquellas palabras que contienen las letras `w, x, y, z`, con el formato:
 - `w: Walter,`
 - `x: Xilofón,`
 - `y: .`
 - `z: .`
- Cree un diccionario, donde cada `key` es la primera letra y cada valor es una lista, cuyo elemento es una tuple (palabra, longitud). Por ejemplo:

```
d['a'] = [ ('Aaa', 3), ('Anna', 4), ... ]
```

5. Las funciones de Bessel de orden n cumplen las relaciones de recurrencia

$$J_{n-1}(x) - \frac{2n}{x} J_n(x) + J_{n+1}(x) = 0$$

$$J_0^2(x) + \sum_{n=1}^{\infty} 2J_n^2(x) = 1$$

Para calcular la función de Bessel de orden N , se empieza con un valor de $M \gg N$, y utilizando los valores iniciales $J_M = 1$, $J_{M+1} = 0$ se utiliza la primera relación para calcular todos los valores de $n < M$. Luego, utilizando la segunda relación se normalizan todos los valores.

Nota: Estas relaciones son válidas si $M \gg x$ (use algún valor estimado, como por ejemplo $M = N + 20$).

Utilice estas relaciones para calcular $J_N(x)$ para $N = 3, 4, 7$ y $x = 2, 5, 5, 7, 10$. Para referencia se dan los valores esperados

$$\begin{aligned} J_3(2,5) &= 0,21660 \\ J_4(2,5) &= 0,07378 \\ J_7(2,5) &= 0,00078 \\ J_3(5,7) &= 0,20228 \\ J_4(5,7) &= 0,38659 \\ J_7(5,7) &= 0,10270 \\ J_3(10,0) &= 0,05838 \\ J_4(10,0) &= -0,21960 \\ J_7(10,0) &= 0,21671 \end{aligned}$$

6. Imprima por pantalla una tabla con valores equiespaciados de x entre 0 y 180, con valores de las funciones trigonométricas de la forma:

```
"""
=====
| x | sen(x) | cos(x) | tan(-x/4) |
=====
| 0 | 0.000 | 1.000 | -0.000 |
| 10 | 0.174 | 0.985 | -0.044 |
| 20 | 0.342 | 0.940 | -0.087 |
| 30 | 0.500 | 0.866 | -0.132 |
| 40 | 0.643 | 0.766 | -0.176 |
| 50 | 0.766 | 0.643 | -0.222 |
| 60 | 0.866 | 0.500 | -0.268 |
| 70 | 0.940 | 0.342 | -0.315 |
| 80 | 0.985 | 0.174 | -0.364 |
| 90 | 1.000 | 0.000 | -0.414 |
| 100 | 0.985 | -0.174 | -0.466 |
| 110 | 0.940 | -0.342 | -0.521 |
| 120 | 0.866 | -0.500 | -0.577 |
| 130 | 0.766 | -0.643 | -0.637 |
| 140 | 0.643 | -0.766 | -0.700 |
| 150 | 0.500 | -0.866 | -0.767 |
| 160 | 0.342 | -0.940 | -0.839 |
| 170 | 0.174 | -0.985 | -0.916 |
=====
"""

```

7. Dada una lista de números, vamos a calcular valores relacionados a su estadística.

- Calcular los valores de la media aritmética, la media geométrica y la media armónica, dados por:

$$A(x_1, \dots, x_n) = \bar{x} = \frac{x_1 + \dots + x_n}{n}$$

$$G(x_1, \dots, x_n) = \sqrt[n]{x_1 \cdots x_n}$$

$$H(x_1, \dots, x_n) = \frac{n}{\frac{1}{x_1} + \dots + \frac{1}{x_n}}$$

- Calcular la desviación estándar:

$$\sigma \equiv \sqrt{\frac{1}{n} \sum_i (x_i - \bar{x})^2}$$

- Calcular la mediana, que se define como el valor para el cual la mitad de los valores de la lista es menor que ella. Si el número de elementos es par, se toma el promedio entre los dos adyacentes.

Realizar los cálculos para las listas de números:

```
L1 = [6.41, 1.28, 11.54, 5.13, 8.97, 3.84, 10.26, 14.1, 12.82, 16.67, 2.56, 17.95, 7.  
→ 69, 15.39]  
L2 = [4.79, 1.59, 2.13, 4.26, 3.72, 1.06, 6.92, 3.19, 5.32, 2.66, 5.85, 6.39, 0.53]
```

- La *moda* se define como el valor que ocurre más frecuentemente en una colección. Note que la moda puede no ser única. En ese caso debe obtener todos los valores. Calcule la moda de las siguientes listas de números enteros:

```
L = [8, 9, 10, 11, 10, 6, 10, 17, 8, 8, 5, 10, 14, 7, 9, 12, 8, 17, 10, 12, 9, 11, 9,  
→ 12, 11, 11, 6, 9, 12, 5, 12, 9, 10, 16, 8, 4, 5, 8, 11, 12]
```

7. Dada una lista de direcciones en el plano, expresadas por los ángulos en grados a partir de un cierto eje, calcular la dirección promedio, expresada en ángulos. Pruebe su algoritmo con las listas:

```
t1 = [0, 180, 370, 10]  
t2 = [30, 0, 80, 180]  
t3 = [80, 180, 540, 280]
```


CAPÍTULO 5

Clase 4: Técnicas de iteración y Funciones

En la clase anterior introdujimos tipos complejos en adición a las listas: tuples, diccionarios (dict), conjuntos (set). Algunas técnicas usuales de iteración sobre estos objetos.

5.1 Iteración sobre conjuntos (set)

```
conj = set("Hola amigos, como están") # Creamos un conjunto desde un string  
conj
```

```
{' ', ',', ' ', 'H', 'a', 'c', 'e', 'g', 'i', 'l', 'm', 'n', 'o', 's', 't', 'á'}
```

```
# Iteramos sobre los elementos del conjunto  
for elem in conj:  
    print(elem, end='')
```

```
cs omHgtaánlei,
```

Comparemos que pasa cuando lo ejecutamos reiteradamente como un script en Python

```
# %load scripts/04_ejemplo_1.py  
conj = set("Hola amigos, como están")  
for elem in conj:  
    print (elem, end='')  
print ('\n')
```

```
cs omHgtaánlei,
```

5.2 Iteración sobre elementos de dos listas

Consideremos las listas:

Clases de Python

```
temp_min = [-3.2, -2, 0, -1, 4, -5, -2, 0, 4, 0]
temp_max = [13.2, 12, 13, 7, 18, 5, 11, 14, 10, 10]
```

Queremos imprimir una lista que combine los dos datos:

```
for t1, t2 in zip(temp_min, temp_max):
    print('La temperatura mínima fue {} y la máxima fue {}'.format(t1, t2))
```

```
La temperatura mínima fue -3.2 y la máxima fue 13.2
La temperatura mínima fue -2 y la máxima fue 12
La temperatura mínima fue 0 y la máxima fue 13
La temperatura mínima fue -1 y la máxima fue 7
La temperatura mínima fue 4 y la máxima fue 18
La temperatura mínima fue -5 y la máxima fue 5
La temperatura mínima fue -2 y la máxima fue 11
La temperatura mínima fue 0 y la máxima fue 14
La temperatura mínima fue 4 y la máxima fue 10
La temperatura mínima fue 0 y la máxima fue 10
```

Como vemos, la función `zip` combina los elementos, tomando uno de cada lista

Podemos mejorar la salida anterior por pantalla si volvemos a utilizar la función `enumerate`

```
for j, t1, t2 in enumerate(zip(temp_min, temp_max)):
    print('El día {} la temperatura mínima fue {} y la máxima fue {}'.format(1+j, t[0], t[1]))
```

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

<ipython-input-8-6170f6c3b697> in <module>
----> 1 for j, t1, t2 in enumerate(zip(temp_min, temp_max)):
      2     print('El día {} la temperatura mínima fue {} y la máxima fue {}'.format(1+j, t[0], t[1]))

ValueError: not enough values to unpack (expected 3, got 2)
```

¿Cuál fue el problema acá? ¿Qué retorna `zip`?

```
list(zip(temp_min, temp_max))
```

```
[(-3.2, 13.2),
 (-2, 12),
 (0, 13),
 (-1, 7),
 (4, 18),
 (-5, 5),
 (-2, 11),
 (0, 14),
 (4, 10),
 (0, 10)]
```

```
for j, t in enumerate(zip(temp_min, temp_max), 1):
    print('El día {} la temperatura mínima fue {} y la máxima fue {}'.format(j, t[0], t[1]))
```

```
El día 1 la temperatura mínima fue -3.2 y la máxima fue 13.2
El día 2 la temperatura mínima fue -2 y la máxima fue 12
El día 3 la temperatura mínima fue 0 y la máxima fue 13
El día 4 la temperatura mínima fue -1 y la máxima fue 7
El día 5 la temperatura mínima fue 4 y la máxima fue 18
El día 6 la temperatura mínima fue -5 y la máxima fue 5
El día 7 la temperatura mínima fue -2 y la máxima fue 11
El día 8 la temperatura mínima fue 0 y la máxima fue 14
El día 9 la temperatura mínima fue 4 y la máxima fue 10
El día 10 la temperatura mínima fue 0 y la máxima fue 10
```

```
# ¿Qué pasa cuando una se consume antes que la otra?
for t1, t2 in zip([1,2,3,4,5],[3,4,5]):
    print(t1,t2)
```

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

Podemos utilizar la función `zip` para sumar dos listas término a término. `zip` funciona también con más de dos listas

```
for j,t1,t2 in zip(range(1,len(temp_min)+1),temp_min, temp_max):
    print('El día {} la temperatura mínima fue {} y la máxima fue {}'.
          format(j, t1, t2))
```

```
El día 1 la temperatura mínima fue -3.2 y la máxima fue 13.2
El día 2 la temperatura mínima fue -2 y la máxima fue 12
El día 3 la temperatura mínima fue 0 y la máxima fue 13
El día 4 la temperatura mínima fue -1 y la máxima fue 7
El día 5 la temperatura mínima fue 4 y la máxima fue 18
El día 6 la temperatura mínima fue -5 y la máxima fue 5
El día 7 la temperatura mínima fue -2 y la máxima fue 11
El día 8 la temperatura mínima fue 0 y la máxima fue 14
El día 9 la temperatura mínima fue 4 y la máxima fue 10
El día 10 la temperatura mínima fue 0 y la máxima fue 10
```

```
tmedia = []
for t1, t2 in zip(temp_min, temp_max):
    tmedia.append((t1+t2)/2)
print(tmedia)
```

```
[5.0, 5.0, 6.5, 3.0, 11.0, 0.0, 4.5, 7.0, 7.0, 5.0]
```

También podemos escribirlo en forma más compacta usando comprensiones de listas

```
tm = [(t1+t2)/2 for t1,t2 in zip(temp_min,temp_max)]
print(tm)
```

```
[5.0, 5.0, 6.5, 3.0, 11.0, 0.0, 4.5, 7.0, 7.0, 5.0]
```

5.3 Más sobre diccionarios

5.3.1 Creación

```
d0 = {} # Equivalente a: dict()  
d1 = {'S': 'A1', 'Z': 13, 'A': 27, 'M': 26.98153863 }  
d2 = {'A': 27, 'M': 26.98153863, 'S': 'A1', 'Z': 13 }  
d3 = dict( [ ('S', 'A1'), ('Z', 13), ('A', 27), ('M', 26.98153863) ] )
```

```
d0
```

```
{ }
```

```
d1
```

```
{'S': 'A1', 'Z': 13, 'A': 27, 'M': 26.98153863}
```

```
d2 == d1
```

```
True
```

```
d2 is d1
```

```
False
```

```
d6 = d2.copy()  
print(d6 == d2)  
print(d6 is d2)
```

```
True
```

```
False
```

```
d3 == d1
```

```
True
```

```
d4 = dict(zip(temp_min, temp_max)) # temp_min tendrá "keys" y temp_max los "values"  
d5 = {n: n**2 for n in range(6)}
```

```
d4
```

```
{-3.2: 13.2, -2: 11, 0: 10, -1: 7, 4: 10, -5: 5}
```

```
d5
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

¿Qué espera que produzca el siguiente código?

```
tempo = {j: {'Tmin': t[0], 'Tmax': t[1]}
         for j, t in enumerate(zip(temp_min, temp_max), 1)}
```

```
tempo
```

```
{1: {'Tmin': -3.2, 'Tmax': 13.2},
 2: {'Tmin': -2, 'Tmax': 12},
 3: {'Tmin': 0, 'Tmax': 13},
 4: {'Tmin': -1, 'Tmax': 7},
 5: {'Tmin': 4, 'Tmax': 18},
 6: {'Tmin': -5, 'Tmax': 5},
 7: {'Tmin': -2, 'Tmax': 11},
 8: {'Tmin': 0, 'Tmax': 14},
 9: {'Tmin': 4, 'Tmax': 10},
10: {'Tmin': 0, 'Tmax': 10}}}
```

5.3.2 Iteraciones sobre diccionarios

```
for k in tempo:
    print('La temperatura máxima del día {} fue {} y la mínima {}'
          .format(k, tempo[k]['Tmin'], tempo[k]['Tmax']))
```

```
La temperatura máxima del día 1 fue -3.2 y la mínima 13.2
La temperatura máxima del día 2 fue -2 y la mínima 12
La temperatura máxima del día 3 fue 0 y la mínima 13
La temperatura máxima del día 4 fue -1 y la mínima 7
La temperatura máxima del día 5 fue 4 y la mínima 18
La temperatura máxima del día 6 fue -5 y la mínima 5
La temperatura máxima del día 7 fue -2 y la mínima 11
La temperatura máxima del día 8 fue 0 y la mínima 14
La temperatura máxima del día 9 fue 4 y la mínima 10
La temperatura máxima del día 10 fue 0 y la mínima 10
```

Cómo comentamos anteriormente, cuando iteramos sobre un diccionario estamos moviéndonos sobre las (k)eyes

```
tempo.keys()
```

```
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
7 in tempo
```

```
True
```

```
7 in tempo.keys()
```

```
True
```

```
11 in tempo
```

```
False
```

Clases de Python

Para referirnos al valor tenemos que hacerlo en la forma `temp[k]`, y no siempre es una manera muy clara de escribir las cosas. Otra manera similar, pero más limpia en este caso sería:

```
list(temp.items())
```

```
[ (1, {'Tmin': -3.2, 'Tmax': 13.2}),
  (2, {'Tmin': -2, 'Tmax': 12}),
  (3, {'Tmin': 0, 'Tmax': 13}),
  (4, {'Tmin': -1, 'Tmax': 7}),
  (5, {'Tmin': 4, 'Tmax': 18}),
  (6, {'Tmin': -5, 'Tmax': 5}),
  (7, {'Tmin': -2, 'Tmax': 11}),
  (8, {'Tmin': 0, 'Tmax': 14}),
  (9, {'Tmin': 4, 'Tmax': 10}),
  (10, {'Tmin': 0, 'Tmax': 10})]
```

```
for k, v in temps.items():
    print('La temperatura máxima del día {} fue {} y la mínima {}'
          .format(k,v['Tmin'], v['Tmax']))
```

```
La temperatura máxima del día 1 fue -3.2 y la mínima 13.2
La temperatura máxima del día 2 fue -2 y la mínima 12
La temperatura máxima del día 3 fue 0 y la mínima 13
La temperatura máxima del día 4 fue -1 y la mínima 7
La temperatura máxima del día 5 fue 4 y la mínima 18
La temperatura máxima del día 6 fue -5 y la mínima 5
La temperatura máxima del día 7 fue -2 y la mínima 11
La temperatura máxima del día 8 fue 0 y la mínima 14
La temperatura máxima del día 9 fue 4 y la mínima 10
La temperatura máxima del día 10 fue 0 y la mínima 10
```

Si queremos iterar sobre los valores podemos utilizar simplemente:

```
for v in temps.values():
    print(v)
```

```
{'Tmin': -3.2, 'Tmax': 13.2}
{'Tmin': -2, 'Tmax': 12}
{'Tmin': 0, 'Tmax': 13}
{'Tmin': -1, 'Tmax': 7}
{'Tmin': 4, 'Tmax': 18}
{'Tmin': -5, 'Tmax': 5}
{'Tmin': -2, 'Tmax': 11}
{'Tmin': 0, 'Tmax': 14}
{'Tmin': 4, 'Tmax': 10}
{'Tmin': 0, 'Tmax': 10}
```

Remarquemos que los diccionarios no tienen definidos un orden por lo que no hay garantías que la próxima vez que ejecutemos cualquiera de estas líneas de código el resultado sea exactamente el mismo. Además, si queremos imprimirlos en un orden predecible debemos escribirlo explícitamente. Por ejemplo:

```
l=list(temps.keys())
l.sort(reverse=True)
```

```
l
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
for k in l:
    print(k, temps[k])
```

```
10 {'Tmin': 0, 'Tmax': 10}
9 {'Tmin': 4, 'Tmax': 10}
8 {'Tmin': 0, 'Tmax': 14}
7 {'Tmin': -2, 'Tmax': 11}
6 {'Tmin': -5, 'Tmax': 5}
5 {'Tmin': 4, 'Tmax': 18}
4 {'Tmin': -1, 'Tmax': 7}
3 {'Tmin': 0, 'Tmax': 13}
2 {'Tmin': -2, 'Tmax': 12}
1 {'Tmin': -3.2, 'Tmax': 13.2}
```

La secuencia anterior puede escribirse en forma más compacta como

```
for k in sorted(list(temps), reverse=True):
    print(k, temps[k])
```

```
10 {'Tmin': 0, 'Tmax': 10}
9 {'Tmin': 4, 'Tmax': 10}
8 {'Tmin': 0, 'Tmax': 14}
7 {'Tmin': -2, 'Tmax': 11}
6 {'Tmin': -5, 'Tmax': 5}
5 {'Tmin': 4, 'Tmax': 18}
4 {'Tmin': -1, 'Tmax': 7}
3 {'Tmin': 0, 'Tmax': 13}
2 {'Tmin': -2, 'Tmax': 12}
1 {'Tmin': -3.2, 'Tmax': 13.2}
```

```
list(temps)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
for k in sorted(list(temps.keys()), reverse=True):
    print(k, temps[k])
```

```
10 {'Tmin': 0, 'Tmax': 10}
9 {'Tmin': 4, 'Tmax': 10}
8 {'Tmin': 0, 'Tmax': 14}
7 {'Tmin': -2, 'Tmax': 11}
6 {'Tmin': -5, 'Tmax': 5}
5 {'Tmin': 4, 'Tmax': 18}
4 {'Tmin': -1, 'Tmax': 7}
3 {'Tmin': 0, 'Tmax': 13}
2 {'Tmin': -2, 'Tmax': 12}
1 {'Tmin': -3.2, 'Tmax': 13.2}
```

5.4 Ejercicios 04 (a)

1. Realice un programa para:

- Leer los datos del archivo **aluminio.dat** y poner los datos del elemento en un diccionario de la forma:

```
d = {'S': 'Al', 'Z':13, 'A':27, 'M': '26.98153863(12)', 'P': 1.0000, 'MS':26.  
      ↵9815386(8)'}
```

- Modifique el programa anterior para que las masas sean números (`float`) y descarte el valor de la incertezza (el número entre paréntesis)
- Agregue el código necesario para obtener una impresión de la forma:

```
Elemento: Al  
Número Atómico: 13  
Número de Masa: 27  
Masa: 26.98154
```

Note que la masa sólo debe contener 5 números decimales

5.5 Funciones

5.5.1 Las funciones son objetos

Las funciones en **Python**, como en la mayoría de los lenguajes, usan una notación similar a la de las funciones matemáticas, con un nombre y uno o más argumentos entre paréntesis. Por ejemplo, ya usamos la función `sum` cuyo argumento es una lista o una *tuple* de números

```
a = [1, 3.3, 5, 7.5, 2.2]  
sum(a)
```

```
19.0
```

```
b = tuple(a)  
sum(b)
```

```
19.0
```

```
sum
```

```
<function sum(iterable, start=0, /)>
```

```
print
```

```
<function print>
```

En **Python** `function` es un objeto, con una única operación posible: podemos llamarla, en la forma:
`func(lista-de-argumentos)`

Como con todos los objetos, podemos definir una variable y asignarle una función (algo así como lo que en C sería un puntero a funciones)

```
f = sum
help(f)
```

Help on built-in function sum in module builtins:

```
sum(iterable, start=0, /)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

```
print('ff is sum? ', f is sum)
print('f(a)=', f(a), ' sum(a)=', sum(a))
```

```
ff is sum?  True
f(a)= 19.0  sum(a)= 19.0
```

También podemos crear un diccionario donde los valores sean funciones:

```
funciones = {'suma': sum, 'minimo': min, 'maximo': max}
```

```
funciones['suma']
```

```
<function sum(iterable, start=0, /)>
```

```
funciones['suma'](a)
```

```
19.0
```

```
print('\n', 'a =', a, '\n')
for k, v in funciones.items():
    print(k, v(a))
```

```
a = [1, 3.3, 5, 7.5, 2.2]
```

```
suma 19.0
minimo 1
maximo 7.5
```

5.5.2 Definición básica de funciones

Tomemos el ejemplo del tutorial de la documentación de Python. Vimos, al introducir el elemento de control **while** una forma de calcular la serie de Fibonacci. Usemos ese ejemplo para mostrar como se definen las funciones

```
def fib(n):
    """Devuelve una lista con los términos
    de la serie de Fibonacci hasta n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
```

(continué en la próxima página)

Clases de Python

(proviene de la página anterior)

```
a, b = b, a+b  
return result
```

```
fib(100)
```

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

```
fib
```

```
<function __main__.fib(n)>
```

Puntos a notar: * Las funciones se definen utilizando la palabra `def` seguida por el nombre, * A continuación, entre paréntesis se escriben los argumentos, en este caso el entero `n`, * La función devuelve (*retorna*) algo, en este caso una lista. Si una función no devuelve algo explícitamente, entonces devuelve `None`. * Lo que devuelve la función se especifica mediante la palabra reservada `return` * Al principio de la definición de la función se escribe la documentación

```
help(fib)
```

```
Help on function fib in module __main__:
```

```
fib(n)  
    Devuelve una lista con los términos  
    de la serie de Fibonacci hasta n.
```

```
help(fib)
```

```
fib.__doc__
```

```
'Devuelve una lista con los términosn de la serie de Fibonacci hasta n.'
```

Como segundo ejemplo, consideremos el ejercicio donde pedimos la velocidad y altura de una pelota en caída libre. Pero esta vez definimos una función para realizar los cálculos:

```
h_0 = 500                      # altura inicial en m  
v_0 = 0                         # velocidad inicial en m/s  
g = 9.8                          # aceleración de la gravedad en m/s^2  
def caida(t):  
    v = v_0 - g*t  
    h = h_0 - v_0*t - g*t**2/2.  
    return v,h
```

```
print(caida(1))
```

```
(-9.8, 495.1)
```

```
v, h = caida(1.5)
```

```
print('Para t = {0}, la velocidad será v={1}\  
y estará a una altura {2:.2f}'.format(1.5, v, h))
```

```
Para t = 1.5, la velocidad será v=-14.700000000000001 y estará a una altura 488.98
```

```
v, h = caida(2.2)
print('Para t = {0}, la velocidad será v={1}, y estará a una altura {2}'.format(
    2.2, v, h))
```

```
Para t = 2.2, la velocidad será v=-21.560000000000002, y estará a una altura 476.284
```

Podemos mejorar considerablemente la funcionalidad si le damos la posibilidad al usuario de dar la posición y la velocidad iniciales

```
g = 9.8                                # aceleración de la gravedad en m/s^2
def caida2(t, h_0, v_0):
    """Calcula la velocidad y posición de una partícula a tiempo t, para condiciones
    →iniciales dadas
    h_0 es la altura inicial
    v_0 es la velocidad inicial
    Se utiliza el valor de aceleración de la gravedad g=9.8 m/s^2
    """
    v = v_0 - g*t
    h = h_0 - v_0*t - g*t**2/2.
    return v,h
```

```
v,h = caida2(2.2, 100, 12)
print('''Para caída desde {h0}m, con vel. inicial {v0}m/s, a t = {0},
la velocidad será v={1}, y estará a una altura {2}'''.
      format(2.2, v, h, h0=100, v0=12))
```

```
Para caída desde 100m, con vel. inicial 12m/s, a t = 2.2,
la velocidad será v=-9.560000000000002, y estará a una altura 49.883999999999986
```

Notemos que podemos llamar a esta función de varias maneras. Podemos llamarla con la constante, o con una variable indistintamente. En este caso, el argumento está definido por su posición: El primero es la altura inicial (`h_0`) y el segundo la velocidad inicial (`v_0`).

```
v0 = 12
caida2(2.2, 100, v0)
```

```
(-9.560000000000002, 49.883999999999986)
```

Pero en Python podemos usar el nombre de la variable. Por ejemplo:

```
caida2(v_0=v0,t=2.2, h_0=100)
```

```
(-9.560000000000002, 49.883999999999986)
```

5.5.3 Argumentos de las funciones

Ámbito de las variables en los argumentos

Consideremos la siguiente función

Clases de Python

```
def func1(x):
    print('x entró a la función con el valor', x)
    x = 2
    print('El nuevo valor de x es', x)
```

```
x = 50
print('Originalmente x vale', x)
func1(x)
print('Ahora x vale', x)
```

```
Originalmente x vale 50
x entró a la función con el valor 50
El nuevo valor de x es 2
Ahora x vale 50
```

```
def func2(x):
    print('x entró a la función con el valor', x)
    print('Id adentro:', id(x))
    x = [2, 7]
    print('El nuevo valor de x es', x)
    print('Id adentro:', id(x))
```

```
x = [50]
print('Originalmente x vale', x)
func2(x)
print('Ahora x vale', x)
print('Id afuera:', id(x))
```

```
Originalmente x vale [50]
x entró a la función con el valor [50]
Id adentro: 139793404910720
El nuevo valor de x es [2, 7]
Id adentro: 139793405075776
Ahora x vale [50]
Id afuera: 139793404910720
```

Qué está pasando acá? Cuando se realiza la llamada a la función, se le pasa una copia del nombre `x`. Cuando le damos un nuevo valor dentro de la función, como en el caso `x = [2]`, entonces esta copia apunta a un nuevo objeto que estamos creando. Por lo tanto, la variable original –definida fuera de la función– no cambia.

En el primer caso, como los escalares son inmutables (de la misma manera que los strings y tuplas) no puede ser modificada, y al reasignarla siempre estamos haciendo apuntar la copia a una nueva variable.

Consideremos estas variantes:

```
def func3a(x):
    print('x entró a la función con el valor', x)
    x.append(2)
    print('El nuevo valor de x es', x)
```

```
x = [50]
print('Originalmente x vale', x)
func3a(x)
print('Ahora x vale', x)
```

```
Originalmente x vale [50]
x entró a la función con el valor [50]
El nuevo valor de x es [50, 2]
Ahora x vale [50, 2]
```

```
def func4(x):
    print('x entró a la función con el valor', x)
    x[0] = 2
    print('El nuevo valor de x es', x)
```

```
x = [50]
print('Originalmente x vale', x)
func4(x)
print('Ahora x vale', x)
```

```
Originalmente x vale [50]
x entró a la función con el valor [50]
El nuevo valor de x es [2]
Ahora x vale [2]
```

Por otro lado, cuando asignamos directamente un valor a uno o más de sus elementos o agregamos elementos a la lista, la copia sigue apuntando a la variable original y el valor de la variable, definida originalmente afuera, cambia.

Funciones con argumentosopcionales

Las funciones pueden tener muchos argumentos. En **Python** pueden tener un número variable de argumentos y pueden tener valores por *default* para algunos de ellos. En el caso de la función de caída libre, vamos a extenderlo de manera que podamos usarlo fuera de la tierra (o en otras latitudes) permitiendo cambiar el valor de la gravedad y asumiendo que, a menos que lo pidamos explícitamente se trata de una simple caída libre:

```
def caida_libre(t, h0, v0 = 0., g=9.8):
    """Devuelve la velocidad y la posición de una partícula en
    caída libre para condiciones iniciales dadas

    Parameters
    -----
    t : float
        el tiempo al que queremos realizar el cálculo
    h0: float
        la altura inicial
    v0: float (opcional)
        la velocidad inicial (default = 0.0)
    g: float (opcional)
        valor de la aceleración de la gravedad (default = 9.8)

    Returns
    -----
    (v, h): tuple of floats
        v= v0 - g*t
        h= h0 - v0*t - g*t**2/2.

    """
    v = v0 - g*t
    h = h0 - v0*t - g*t**2/2.
    return v,h
```

Clases de Python

```
# Desde 1000 metros con velocidad inicial cero
print( caida_libre(2,1000))
```

```
(-19.6, 980.4)
```

```
# Desde 1000 metros con velocidad inicial hacia arriba
print(caida_libre(1, 1000, 10))
```

```
(0.1999999999999993, 985.1)
```

```
# Desde 1000 metros con velocidad inicial cero
print(caida_libre(h0= 1000, t=2))
```

```
(-19.6, 980.4)
```

```
# Desde 1000 metros con velocidad inicial cero en la luna
print( caida_libre( v0=0, h0=1000, t=14.2857137))
```

```
(-139.9999426000002, 8.19999820135417e-05)
```

```
# Desde 1000 metros con velocidad inicial cero en la luna
print( caida_libre( v0=0, h0=1000, t=14.2857137, g=1.625))
```

```
(-23.2142847625, 834.1836870663262)
```

```
help(caida_libre)
```

```
Help on function caida_libre in module __main__:
```

```
caida_libre(t, h0, v0=0.0, g=9.8)
    Devuelve la velocidad y la posición de una partícula en
    caída libre para condiciones iniciales dadas

    Parameters
    -----
    t : float
        el tiempo al que queremos realizar el cálculo
    h0: float
        la altura inicial
    v0: float (opcional)
        la velocidad inicial (default = 0.0)
    g: float (opcional)
        valor de la aceleración de la gravedad (default = 9.8)

    Returns
    -----
    (v,h): tuple of floats
        v= v0 - g*t
        h= h0 - v0*t -g*t^2/2
```

Tipos mutables en argumentosopcionales

Hay que tener cuidado cuando usamos valores por defecto con tipos que pueden modificarse dentro de la función. Consideremos la siguiente función:

```
def func2b(x1, x=[]):
    print('x entró a la función con el valor', x)
    x.append(x1)
    print('El nuevo valor de x es', x)
```

```
func2b(1)
```

```
x entró a la función con el valor []
El nuevo valor de x es [1]
```

```
func2b(2)
```

```
x entró a la función con el valor [1]
El nuevo valor de x es [1, 2]
```

Notar: No se pueden usar argumentos con *nombre* antes de los argumentos requeridos (en este caso t).

Tampoco se pueden usar argumentos sin su *nombre* después de haber incluido alguno con su nombre. Por ejemplo no son válidas las llamadas:

```
caida_libre(t=2, 0.)
caida_libre(2, v0=0., 1000)
```

Número variable de argumentos y argumentos *keywords*

Se pueden definir funciones que toman un número variable de argumentos (como una lista), o que aceptan un diccionario como argumento. Este tipo de argumentos se llaman argumentos *keyword* (`kwargs`). Una buena explicación se encuentra en el [Tutorial de la documentación](#). Ahora vamos a usar otra explicación rápida. Consideremos la función f, que imprime sus argumentos:

```
def f(p, *args, **kwargs):
    print("p: {}, tipo: {}".format(p, type(p)))
    print("args: {}, tipo: {}".format(args, type(args)))
    print("kwargs: {}, tipo: {}".format(kwargs, type(kwargs)))
```

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

```
p: 1, tipo: <class 'int'>
args: (2, 3), tipo: <class 'tuple'>
kwargs: {}, tipo: <class 'dict'>
```

```
f(1,2,3,4,5,6)
```

```
p: 1, tipo: <class 'int'>
args: (2, 3, 4, 5, 6), tipo: <class 'tuple'>
kwargs: {}, tipo: <class 'dict'>
```

En este ejemplo, el primer valor se asigna al argumento requerido p, y los siguientes a una variable que se llama args, que es del tipo tuple

Clases de Python

```
f(1, 2, 3, 5, anteultimo= 9, ultimo = -1)
```

```
p: 1, tipo: <class 'int'>
args: (2, 3, 5), tipo: <class 'tuple'>
kwargs: {'anteultimo': 9, 'ultimo': -1}, tipo: <class 'dict'>
```

```
f(1, (1, 2, 3), 4, ultimo=-1)
```

Con `*args` se indica *mapear todos los argumentos posicionales no explícitos a una tupla llamada “args”*. Con `**kwargs` se indica mapear todos los argumentos de palabra clave no explícitos a un diccionario llamado `kwargs`. Esta acción de convertir un conjunto de argumentos a una tuple o diccionario se conoce como *empacar* o *empaquetar* los datos.

NOTA: Por supuesto, no es necesario utilizar los nombres `args` y `kwargs`. Podemos llamarlas de cualquier otra manera! los simbolos que indican cantidades arbitrarias de parametros son `*` y `**`. Además es posible poner parametros comunes antes de los parametros arbitrarios, como se muestra en el ejemplo.

Exploraremos otras variantes de llamadas a la función:

```
f(1, ultimo=-1)
```

```
p: 1, tipo: <class 'int'>
args: (), tipo: <class 'tuple'>
kwargs: {'ultimo': -1}, tipo: <class 'dict'>
```

```
f(1, ultimo=-1, 2)
```

```
File "<ipython-input-59-acd06ccc380f>", line 1
    f(1, ultimo=-1, 2)
               ^
SyntaxError: positional argument follows keyword argument
```

```
f(ultimo=-1, p=2)
```

```
p: 2, tipo: <class 'int'>
args: (), tipo: <class 'tuple'>
kwargs: {'ultimo': -1}, tipo: <class 'dict'>
```

Un ejemplo de una función con número variable de argumentos puede ser la función `multiplica`:

```
def multiplica(*args):
    s = 1
    for a in args:
        s *= a
    return s
```

```
multiplica(2, 5)
```

```
10
```

```
multiplica(2, 3, 5, 9, 12)
```

3240

5.6 Ejercicios 4 (b)

2. Escriba funciones para analizar la divisibilidad de enteros:

- La función `es_divisible1(x)` que retorna verdadero si `x` es divisible por alguno de `2, 3, 5, 7` o falso en caso contrario.
- La función `es_divisible_por_lista` que cumple la misma función que `es_divisible1` pero recibe dos argumentos: el entero `x` y una variable del tipo lista que contiene los valores para los cuáles debemos examinar la divisibilidad. Las siguientes expresiones deben retornar el mismo valor:

```
es_divisible1(x)
es_divisible_por_lista(x, [2, 3, 5, 7])
es_divisible_por_lista(x)
```

- La función `es_divisible_por` cuyo primer argumento (mandatorio) es `x`, y luego puede aceptar un número indeterminado de argumentos:

```
es_divisible_por(x)  # retorna verdadero siempre
es_divisible_por(x, 2) # verdadero si x es par
es_divisible_por(x, 2, 3, 5, 7) # igual resultado que es_divisible1(x) e igual a ↵
    ↵ es_divisible_por_lista(x)
es_divisible_por(x, 2, 3, 5, 7, 9, 11, 13) # o cualquier secuencia de argumentos ↵
    ↵ debe funcionar
```

CAPÍTULO 6

Clase 5: Funciones

6.1 Empacar y desempacar secuencias o diccionarios

Cuando en **Python** creamos una función que acepta un número arbitrario de argumentos estamos utilizando una habilidad del lenguaje que es el empaquetamiento y desempaquetamiento automático de variables.

Al definir un número variable de argumentos de la forma:

```
def f (*v) :  
    ...
```

y luego utilizarla en alguna de las formas:

```
f(1)  
f(1, 'hola')  
f(a, 2, 3.5, 'hola')
```

Python automáticamente convierte los argumentos en una única tupla:

```
f(1)           --> v = (1,)  
f(1, 'hola')   --> v = (1, 'hola')  
f(a, 2, 3.5, 'hola') --> v = (a, 2, 3.5, 'hola')
```

Análogamente, cuando utilizamos funciones podemos desempacar múltiples valores en los argumentos de llamada a las funciones.

Si definimos una función que recibe un número determinado de argumentos

```
def g(a, b, c) :  
    ...
```

y definimos una lista (o tupla)

```
t1 = [a1, b1, c1]
```

Clases de Python

podemos realizar la llamada a la función utilizando la notación asterisco o estrella

```
g(*t1)          --> g(a1, b1, c1)
```

Esta notación no se puede utilizar en cualquier contexto. Por ejemplo, es un error tratar de hacer

```
t2 = *t1
```

pero en el contexto de funciones podemos desempacarlos para convertirlos en varios argumentos que acepta la función usando la expresión con asterisco. Veamos lo que esto quiere decir con la función `caida_libre()` definida anteriormente

```
def caida_libre(t, h0, v0 = 0., g=9.8):
    """Devuelve la velocidad y la posición de una partícula en
    caída libre para condiciones iniciales dadas

    Parameters
    -----
    t : float
        el tiempo al que queremos realizar el cálculo
    h0: float
        la altura inicial
    v0: float (opcional)
        la velocidad inicial (default = 0.0)
    g: float (opcional)
        valor de la aceleración de la gravedad (default = 9.8)

    Returns
    -----
    (v, h): tuple of floats
        v= v0 - g*t
        h= h0 - v0*t - g*t^2/2

    """
    v = v0 - g*t
    h = h0 - v0*t - g*t**2/2.
    return v, h
```

```
datos = (5.4, 1000., 0.)           # Una lista (tuple en realidad)
print (caida_libre(*datos))
```

```
(-52.92000000000001, 857.116)
```

En la llamada a la función, la expresión `*datos` le indica al intérprete Python que la secuencia (tuple) debe convertirse en una sucesión de argumentos, que es lo que acepta la función.

Similarmente, para desempacar un diccionario usamos la notación `**diccionario`:

```
# diccionario, caída libre en la luna
otros_datos = {'t':5.4, 'h0': 1000., "g" : 1.625}
v, h = caida_libre(**otros_datos)
print ('v={}, h={}'.format(v,h))
```

```
v=-8.775, h=976.3075
```

En resumen:

- la notación `(*datos)` convierte la tuple (o lista) en los tres argumentos que acepta la función caída libre. Los siguientes llamados son equivalentes:

```
caida_libre(*datos)
caida_libre(datos[0], datos[1], datos[2])
caida_libre(5.4, 1000., 0.)
```

- la notación `(**otros_datos)` desempaca el diccionario en pares `clave=valor`, siendo equivalentes los dos llamados:

```
caida_libre(**otros_datos)
caida_libre(t=5.4, h0=1000., g=0.2)
```

6.2 Funciones que devuelven funciones

Las funciones pueden ser pasadas como argumento y pueden ser retornadas por una función, como cualquier otro objeto (números, listas, tuples, cadenas de caracteres, diccionarios, etc). Veamos un ejemplo simple de funciones que devuelven una función:

```
def crear_potencia(n):
    "Devuelve la función x^n"
    def potencia(x):
        "potencia {}-ésima de x".format(n)
        return x**n
    return potencia
```

```
f = crear_potencia(3)
cubos = [f(j) for j in range(5)]
```

6.3 Ejercicios 05 (a)

1. Escriba una función `crear_sen(A, w)` que acepte dos números reales A, w como argumentos y devuelva la función $f(x)$.

Al evaluar la función f en un dado valor x debe dar el resultado: $f(x) = A \sin(wx)$ tal que se pueda utilizar de la siguiente manera:

```
f = crear_sen(3, 1.5)
f(2)          # Debería imprimir el resultado de 3*sin(1.5*2)=0.4233600241796016
```

2. Utilizando conjuntos (`set`), escriba una función que compruebe si un string contiene todas las vocales. La función debe devolver `True` o `False`.

6.4 Funciones que toman como argumento una función

```
def aplicar_fun(f, L):
    """Aplica la función f a cada elemento del iterable L y devuelve una lista con los resultados.

    IMPORTANTE: Notar que no se realiza ninguna comprobación de validez
    """
    return [f(x) for x in L]
```

```
import math as m
Lista = list(range(1,10))
t = aplicar_fun(m.sin, Lista)
```

```
t
```

```
[0.8414709848078965,
 0.9092974268256817,
 0.1411200080598672,
 -0.7568024953079282,
 -0.9589242746631385,
 -0.27941549819892586,
 0.6569865987187891,
 0.9893582466233818,
 0.4121184852417566]
```

El ejemplo anterior se podría escribir

```
Lista = list(range(5))
aplicar_fun(crear_potencia(3), Lista)
```

```
[0, 1, 8, 27, 64]
```

Notar que definimos la función `aplicar_fun()` que recibe una función y una secuencia, pero no necesariamente una lista, por lo que podemos aplicarla directamente a `range`:

```
aplicar_fun(crear_potencia(3), range(5))
```

```
[0, 1, 8, 27, 64]
```

Además, debido a su definición, el primer argumento de la función `aplicar_fun()` no está restringida a funciones numéricas pero al usarla tenemos que asegurar que la función y el iterable (lista) que pasamos como argumentos son compatibles.

Veamos otro ejemplo:

```
s = ['hola', 'chau']
print(aplicar_fun(str.upper, s))
```

```
['HOLA', 'CHAU']
```

donde `str.upper` es una función definida en **Python**, que convierte a mayúsculas el string dado `str.upper('hola') = 'HOLA'`.

6.5 Aplicacion 1: Ordenamiento de listas

Consideremos el problema del ordenamiento de una lista de strings. Como vemos el resultado usual no es necesariamente el deseado

```
s1 = ['Estudiantes', 'caballeros', 'Python', 'Curso', 'pc', 'aereo']
print(s1)
print(sorted(s1))
```

```
['Estudiantes', 'caballeros', 'Python', 'Curso', 'pc', 'aereo']
['Curso', 'Estudiantes', 'Python', 'aereo', 'caballeros', 'pc']
```

Acá sorted es una función, similar al método `str.sort()` que mencionamos anteriormente, con la diferencia que devuelve una nueva lista con los elementos ordenados. Como los elementos son *strings*, la comparación se hace respecto a su posición en el abecedario. En este caso no es lo mismo mayúsculas o minúsculas.

```
s2 = [s.lower() for s in s1]
print(s2)
print(sorted(s2))
```

```
['estudiantes', 'caballeros', 'python', 'curso', 'pc', 'aereo']
['aereo', 'caballeros', 'curso', 'estudiantes', 'pc', 'python']
```

Possiblemente queremos el orden que obtuvimos en segundo lugar pero con los elementos dados originalmente (con sus mayúsculas y minúsculas originales). Para poder modificar el modo en que se ordenan los elementos, la función `sorted` (y el método `sort`) tienen el argumento opcional `key`

```
help(sorted)
```

```
Help on built-in function sorted in module builtins:
```

```
sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

Como vemos tiene un argumento opcional `key` que es una función. Veamos algunos ejemplos de su uso

```
sorted(s1, key=str.lower)
```

```
['aereo', 'caballeros', 'Curso', 'Estudiantes', 'pc', 'Python']
```

Como vemos, los strings están ordenados adecuadamente. Si queremos ordenarlos por longitud de la palabra

```
sorted(s1, key=len)
```

```
['pc', 'Curso', 'aereo', 'Python', 'caballeros', 'Estudiantes']
```

Supongamos que queremos ordenarla alfabéticamente por la segunda letra

```
def segunda(a):
    return a[1]

sorted(s1, key=segunda)
```

```
[ 'caballeros', 'pc', 'aereo', 'Estudiantes', 'Curso', 'Python' ]
```

6.6 Funciones anónimas

En ocasiones como esta suele ser más rápido (o conveniente) definir la función, que se va a utilizar una única vez, sin darle un nombre. Estas se llaman funciones *lambda*, y el ejemplo anterior se escribiría

```
sorted(s1, key=lambda a: a[1])
```

```
[ 'caballeros', 'pc', 'aereo', 'Estudiantes', 'Curso', 'Python' ]
```

Si queremos ordenarla alfabéticamente empezando desde la última letra:

```
sorted(s1, key=lambda a: a[::-1])
```

```
[ 'pc', 'Python', 'aereo', 'Curso', 'Estudiantes', 'caballeros' ]
```

6.7 Ejemplo 1: Integración numérica

Veamos en más detalle el caso de funciones que reciben como argumento otra función, estudiando un caso usual: una función de integración debe recibir como argumento al menos una función a integrar y los límites de integración:

```
# %load scripts/05_ejemplo_1.py
def integrate_simpsons(f, a, b, N=10):
    """Calcula numéricamente la integral de la función en el intervalo dado
    utilizando la regla de Simpson

    Keyword Arguments:
    f -- Función a integrar
    a -- Límite inferior
    b -- Límite superior
    N -- El intervalo se separa en 2*N intervalos
    """
    h = (b - a) / (2 * N)
    I = f(a) + f(b)
    for j in range(1, N + 1):
        x2j = a + 2 * j * h
        x2jm1 = a + (2 * j - 1) * h
        I += 2 * f(x2j) + 4 * f(x2jm1)
    return I * h / 3
```

En este ejemplo programamos la fórmula de integración de Simpson para obtener la integral de una función $f(x)$ provista por el usuario, en un dado intervalo:

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[f(x_0) + 2 \sum_{j=1}^{n/2} f(x_{2j}) + 4 \sum_{j=1}^{n/2} f(x_{2j-1}) - f(x_n) \right]$$

¿Cómo usamos la función de integración?

```
def potencia2(x):
    return x**2

integrate_simps(potencia2, 0, 3, 7)
```

9.0

Acá definimos una función, y se la pasamos como argumento a la función de integración.

6.7.1 Uso de funciones anónimas

Veamos como sería el uso de funciones anónimas en este contexto

```
integrate_simps(lambda x: x**2, 0, 3, 7)
```

9.0

La notación es un poco más corta, que es cómodo pero no muy relevante para un caso. Si queremos, por ejemplo, aplicar el integrador a una familia de funciones la notación se simplifica notablemente:

```
print('Integrales:')
a = 0
b = 3
for n in range(6):
    I = integrate_simps(lambda x: (n + 1) * x**n, a, b, 10)
    print('I ( {} x^{}, {}, {} ) = {:.5f}'.format(n + 1, n, a, b, I))
```

```
Integrales:
I ( 1 x^0, 0, 3 ) = 3.00000
I ( 2 x^1, 0, 3 ) = 9.00000
I ( 3 x^2, 0, 3 ) = 27.00000
I ( 4 x^3, 0, 3 ) = 81.00000
I ( 5 x^4, 0, 3 ) = 243.00101
I ( 6 x^5, 0, 3 ) = 729.00911
```

Este es un ejemplo de uso de las funciones anónimas lambda. Recordemos que la forma general de las funciones lambda es:

```
lambda x,y,z: expresión_de(x,y,z)
```

por ejemplo en el ejemplo anterior, para calcular $(n + 1)x^n$, hicimos:

```
lambda x: (n+1) * x**n
```

6.8 Ejemplo 2: Polinomio interpolador

Veamos ahora una función que retorna una función. Supongamos que tenemos una tabla de puntos (x, y) por los que pasan nuestros datos y queremos interpolar los datos con un polinomio.

Sabemos que dados N puntos, hay un único polinomio de grado N que pasa por todos los puntos. En este ejemplo utilizamos la fórmula de Lagrange para obtenerlo.

```
%load scripts/ejemplo_05_2.py
```

```
# %load scripts/ejemplo_05_2.py
def polinomio_interp(x, y):
    """Devuelve el polinomio interpolador que pasa por los puntos (x_i, y_i)

    Warning: La implementación es numéricamente inestable. Funciona para algunos
    puntos (menor a 20)

    Keyword Arguments:
    x -- Lista con los valores de x
    y -- Lista con los valores de y
    """

M = len(x)

def polin(xx):
    """Evalúa el polinomio interpolador de Lagrange"""
    P = 0

    for j in range(M):
        pt = y[j]
        for k in range(M):
            if k == j:
                continue
            fac = x[j] - x[k]
            pt *= (xx - x[k]) / fac
        P += pt
    return P

return polin
```

Lo que obtenemos al llamar a esta función es una función

```
f = polinomio_interp([0,1], [0,2])
```

```
f
```

```
<function __main__.polinomio_interp.<locals>.polin(xx)>
```

```
help(f)
```

```
Help on function polin in module __main__:

polin(xx)
    Evalúa el polinomio interpolador de Lagrange
```

```
f(3.4)
```

```
6.8
```

Este es el resultado esperado porque queremos el polinomio que pasa por dos puntos (una recta), y en este caso es la recta $y = 2x$. Veamos cómo usarlo, en forma más general:

```
# %load scripts/ejemplo_05_3
#from ejemplo_05_2 import polinomio_interp

xmax = 5
step = 0.2
N = int(5 / step)

x2, y2 = [1, 2, 3], [1, 4, 9]    # x^2
f2 = polinomio_interp(x2, y2)

x3, y3 = [0, 1, 2, 3], [0, 2, 16, 54]  # 2 x^3
f3 = polinomio_interp(x3, y3)

print('\n x    f2(x)    f3(x)\n' + 18 * '-')
for j in range(N):
    x = step * j
    print('{:.1f}  {:.5.2f}  {:.6.2f}'.format(x, f2(x), f3(x)))
```

x	f2(x)	f3(x)
0.0	0.00	0.00
0.2	0.04	0.02
0.4	0.16	0.13
0.6	0.36	0.43
0.8	0.64	1.02
1.0	1.00	2.00
1.2	1.44	3.46
1.4	1.96	5.49
1.6	2.56	8.19
1.8	3.24	11.66
2.0	4.00	16.00
2.2	4.84	21.30
2.4	5.76	27.65
2.6	6.76	35.15
2.8	7.84	43.90
3.0	9.00	54.00
3.2	10.24	65.54
3.4	11.56	78.61
3.6	12.96	93.31
3.8	14.44	109.74
4.0	16.00	128.00
4.2	17.64	148.18
4.4	19.36	170.37
4.6	21.16	194.67
4.8	23.04	221.18

6.9 Ejercicios 05 (b)

3. Escriba una serie de funciones que permitan trabajar con polinomios. Vamos a representar a un polinomio como una lista de números reales, donde cada elemento corresponde a un coeficiente que acompaña una potencia
 - Una función que devuelva el orden del polinomio (un número entero)
 - Una función que sume dos polinomios y devuelva un polinomio (objeto del mismo tipo)

- Una función que multiplique dos polinomios y devuelva el resultado en otro polinomio
- Una función devuelva la derivada del polinomio (otro polinomio).
- Una función que, acepte el polinomio y devuelva la función correspondiente.

4. Vamos a describir un **sudoku** como un array bidimensional de 9×9 números, cada uno de ellos entre 1 y 4.

Escribir una función que tome como argumento una grilla (Lista bidimensional de 9×9) y devuelva verdadero si los números corresponden a la resolución correcta y falso en caso contrario. Recordamos que para que sea válido debe cumplirse que:

- Los números están entre 1 y 9
- En cada fila no deben repetirse
- En cada columna no deben repetirse
- En todas las regiones de 3×3 que no se solapan, empezando de cualquier esquina, no deben repetirse.

6.10 Funciones que aceptan y devuelven funciones (Decoradores)

Consideremos la siguiente función `mas_uno`, que toma como argumento una función y devuelve otra función.

```
def mas_uno(func):
    "Devuelve una función"
    def fun(args):
        "Agrega 1 a cada uno de los elementos y luego aplica la función"
        xx = [x+1 for x in args]
        y= func(xx)
        return y
    return fun
```

```
ssum= mas_uno(sum)                      # h es una función
mmin= mas_uno(min)                      # f es una función
mmax= mas_uno(max)                      # g es una función
```

```
a = [0, 1, 3.3, 5, 7.5, 2.2]
print(a)
print(sum(a), ssum(a))
print(min(a), mmin(a))
print(max(a), mmax(a))
```

6.10.1 Notación para decoradores

Podemos aplicar la función tanto a funciones intrínsecas como a funciones definidas por nosotros

```
def parabola(v):
    return [x**2 + 2*x for x in v]
```

```
mparabola = mas_uno(parabola)
```

```
print(parabola(a))
print(mpabola(a))
```

Notemos que al decorar una función estamos creando una enteramente nueva

```
def parabola # Borramos el objeto
```

```
parabola(a)
```

```
mparabola(a)
```

Algunas veces queremos eso, renombrar la función original (se lo llama decorar):

```
def parabola(v):
    return [x**2 + 2*x for x in v]
mparabola = mas_uno(parabola)
del parabola
parabola = mparabola
del mparabola
```

Son un montón de líneas, podemos simplificarlo:

```
def parabola(v):
    return [x**2 + 2*x for x in v]
parabola = mas_uno(parabola)
```

Esta es una situación que puede darse frecuentemente en algunas áreas. Se decidió simplificar la notación, introduciendo el uso de @. Lo anterior puede escribirse como:

```
@mas_uno
def mi_parabola(v):
    return [x**2 + 2*x for x in v]
```

La única restricción para utilizar esta notación es que la línea con el decorador debe estar inmediatamente antes de la definición de la función a decorar

```
mi_parabola(a)
```

6.10.2 Algunos Usos de decoradores

```
mi_parabola(3)
```

El problema aquí es que definimos la función para tomar como argumentos una lista (o al menos un iterable de números) y estamos tratando de aplicarla a un número. Podemos definir un decorador para asegurarnos que el tipo es correcto (que no es del todo correcto)

```
def test_argumento_list_num(f):
    def check(v):
        if (type(v) == list):
            return f(v)
        else:
            print("Error: El argumento debe ser una lista")
    return check
```

```
mi_parabola = test_argumento_list_num(mi_parabola)
```

```
mi_parabola(3)
```

```
mi_parabola(a)
```

Supongamos que queremos simplemente extender la función para que sea válida también con argumentos escalares. Definimos una nueva función que utilizaremos como decorador

```
def hace_argumento_list(f):
    def check(v):
        "Corrige el argumento si no es una lista"
        if type(v) == list:
            return f(v)
        else:
            return f([v])
    return check
```

```
@hace_argumento_list
def parabola(v):
    return [x**2 + 2*x for x in v]
```

```
print(parabola(3))
print(parabola([3]))
```

6.11 Atrapar y administrar errores

Python tiene incorporado un mecanismo para atrapar errores de distintos tipos, así como para generar errores que den información al usuario sobre usos incorrectos del código.

En primer lugar consideremos lo que se llama un error de sintaxis. El siguiente comando es sintácticamente correcto y el intérprete sabe como leerlo

```
print("hola")
```

```
hola
```

mientras que, si escribimos algo que no está permitido en el lenguaje

```
print("hola"))
```

```
File "<ipython-input-22-3e8c6f917d42>", line 1
    print("hola"))
               ^
SyntaxError: invalid syntax
```

El intérprete detecta el error y repite la línea donde lo identifica. Este tipo de errores debe corregirse para poder seguir con el programa.

Consideremos ahora el código siguiente, que es sintácticamente correcto pero igualmente causa un error

```
a = 1
b = 0
z = a / b
```

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

<ipython-input-23-fd0e24f40c98> in <module>
    1 a = 1
    2 b = 0
----> 3 z = a / b

ZeroDivisionError: division by zero
```

Cuando se encuentra un error, **Python** muestra el lugar en que ocurre y de qué tipo de error se trata.

```
print (hola)
```

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

<ipython-input-24-c6948929a301> in <module>
----> 1 print (hola)

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

Este mensaje da un tipo de error diferente. Ambos: ZeroDivisionError y NameError son tipos de errores (o excepciones). Hay una larga lista de tipos de errores que son parte del lenguaje y puede consultarse en la documentación de [Built-in Exceptions](#).

6.11.1 Administración de excepciones

Cuando nuestro programa aumenta en complejidad, aumenta la posibilidad de encontrar errores. Esto se incrementa si se tiene que interactuar con otros usuarios o con datos externos. Consideremos el siguiente ejemplo simple:

```
%cat ../data/ej_clase5.dat
```

```
1 2
2 6
3 9
4 12
5.5 30.25
```

```
with open("../data/ej_clase5.dat") as fi:
    for l in fi:
        t = l.split()
        print("t = {}".format(t))           # Línea sólo para inspección
        m = int(t[0])
        n = int(t[1])
        print("m = {}, n = {}, m x n = {}".format(m, n, m*n))
```

```
t = ['1', '2']
m = 1, n = 2, m x n = 2
t = ['2', '6']
```

(continué en la próxima página)

(proviene de la página anterior)

```
m = 2, n = 6, m x n = 12
t = ['3', '9']
m = 3, n = 9, m x n = 27
t = ['4', '12']
m = 4, n = 12, m x n = 48
t = ['5.5', '30.25']
```

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

<ipython-input-26-55192d3677b3> in <module>
      3     t = l.split()
      4     print("t = {}".format(t))          # Línea sólo para inspección
----> 5     m = int(t[0])
      6     n = int(t[1])
      7     print("m = {}, n = {}, m x n = {}".format(m,n, m*n))

ValueError: invalid literal for int() with base 10: '5.5'
```

En este caso se levanta una excepción del tipo `ValueError` debido a que este valor (5.5) no se puede convertir a `int`. Podemos modificar nuestro programa para manejar este error:

```
with open("../data/ej_clase5.dat") as fi:
    for l in fi:
        t = l.split()
        try:
            m = int(t[0])
            n = int(t[1])
            print("m = {}, n = {}, m x n = {}".format(m,n, m*n))
        except:
            print("Error: t = {} no puede convertirse a entero".format(t))
```

```
m = 1, n = 2, m x n = 2
m = 2, n = 6, m x n = 12
m = 3, n = 9, m x n = 27
m = 4, n = 12, m x n = 48
Error: t = ['5.5', '30.25'] no puede convertirse a entero
```

En este caso podríamos ser más precisos y especificar el tipo de excepción que estamos esperando

```
with open("../data/ej_clase5.dat") as fi:
    for l in fi:
        t = l.split()
        try:
            m = int(t[0])
            n = int(t[1])
            print("m = {}, n = {}, m x n = {}".format(m,n, m*n))
        except(ValueError):
            print("Error: t = {} no puede convertirse a entero".format(t))
```

```
m = 1, n = 2, m x n = 2
m = 2, n = 6, m x n = 12
m = 3, n = 9, m x n = 27
```

(continué en la próxima página)

(provine de la página anterior)

```
m = 4, n = 12, m x n = 48
Error: t = ['5.5', '30.25'] no puede convertirse a entero
```

```
with open("../data/ej_clase5.dat") as fi:
    for l in fi:
        t = l.split()
        try:
            m = int(t[0])
            n = int(t[1])
            print("m = {}, n = {}, m x n = {}".format(m,n, m*n))
        except(ValueError):
            print("Error: t = {} no puede convertirse a entero".format(t))
        except(IndexError):
            print('Error: La linea "{}" no contiene un par'.format(l.strip()))
```

```
m = 1, n = 2, m x n = 2
m = 2, n = 6, m x n = 12
m = 3, n = 9, m x n = 27
m = 4, n = 12, m x n = 48
Error: t = ['5.5', '30.25'] no puede convertirse a entero
```

La forma general

La declaración `try` funciona de la siguiente manera:

- Primero, se ejecuta el *bloque try* (el código entre las declaraciones `try` y `except`).
- Si no ocurre ninguna excepción, el *bloque except* se saltea y termina la ejecución de la declaración `try`.
- Si ocurre una excepción durante la ejecución del *bloque try*, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada `except`, se ejecuta el *bloque except*, y la ejecución continúa luego de la declaración `try`.
- Si ocurre una excepción que no coincide con la excepción nombrada en el `except`, esta se pasa a declaraciones `try` de más afuera; si no se encuentra nada que la maneje, es una *excepción no manejada*, y la ejecución se frena con un mensaje como los mostrados arriba.

El mecanismo es un poco más complejo, y permite un control más fino que lo descripto aquí.

6.11.2 Levantando excepciones

Podemos forzar a que nuestro código levante una excepción usando `raise`. Por ejemplo:

```
x = 1
if x > 0:
    raise Exception("x = {}, no debería ser positivo".format(x))
```

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

<ipython-input-30-c3c4a53042e7> in <module>
      1 x = 1
      2 if x > 0:
----> 3     raise Exception("x = {}, no debería ser positivo".format(x))
```

(continué en la próxima página)

(proviene de la página anterior)

```
Exception: x = 1, no debería ser positivo
```

O podemos ser más específicos, y dar el tipo de error adecuado

```
x = 1
if x > 0:
    raise ValueError("x = {}, no debería ser positivo".format(x))
```

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

```
<ipython-input-31-3c0ee90e0415> in <module>
      1 x = 1
      2 if x > 0:
----> 3     raise ValueError("x = {}, no debería ser positivo".format(x))
```

```
ValueError: x = 1, no debería ser positivo
```

Clase 6: Programación Orientada a Objetos

7.1 Breve introducción a Programación Orientada a Objetos

Vimos como escribir funciones que realizan un trabajo específico y nos devuelven un resultado. La mayor parte de nuestros programas van a estar diseñados con un hilo conductor principal, que utiliza una serie de funciones para realizar el cálculo. De esta manera, el código es altamente reusable.

Hay otras maneras de organizar el código, particularmente útil cuando un conjunto de rutinas comparte un dado conjunto de datos. En ese caso, puede ser adecuado utilizar un esquema de programación orientada a objetos.

7.2 Clases y Objetos

Una Clase define características, que tienen los objetos de dicha clase. En general la clase tiene: un nombre y características (campos o atributos y métodos).

Un Objeto en programación puede pensarse como la representación de un objeto real, de una dada clase. Un objeto real tiene una composición y características, y además puede realizar un conjunto de actividades (tiene un comportamiento). Cuando programamos, las partes son los datos, y el comportamiento son los métodos.

Ejemplos de la vida diaria serían: Una clase *Bicicleta*, y muchos objetos del tipo bicicleta (mi bicicleta, la suya, etc). La definición de la clase debe contener la información de qué es una bicicleta (dos ruedas, manubrio, etc) y luego se realizan muchas copias del tipo bicicleta (los objetos).

Se dice que los **objetos** son instancias de una **clase**, por ejemplo ya vimos los números enteros. Cuando definimos: `a = 3` estamos diciendo que `a` es una instancia (objeto) de la clase `int`.

Los objetos pueden guardar datos (en este caso `a` guarda el valor `3`). Las variables que contienen los datos de los objetos se llaman usualmente campos o atributos. Las acciones que tienen asociadas los objetos se realizan a través de funciones internas, que se llaman métodos.

Las clases se definen con la palabra reservada `class`, veamos un ejemplo simple:

Clases de Python

```
class Punto:  
    "Clase para describir un punto en el espacio"  
  
    def __init__(self, x, y, z):  
        self.x = x  
        self.y = y  
        self.z = z
```

```
P1 = Punto(0.5, 0.5, 0)
```

```
P1
```

```
<__main__.Punto at 0x7f95188cb610>
```

```
P1.x
```

```
0.5
```

Como vemos, acabamos de definir una clase de tipo Punto. A continuación definimos un *método __init__* que hace el trabajo de inicializar el objeto.

Algunos puntos a notar:

- La línea `P1 = Punto(0.5, 0.5, 0)` crea un nuevo objeto del tipo Punto. Notar que usamos paréntesis como cuando llamamos a una función pero Python sabe que estamos llamando a una clase.
- El método `__init__` es especial y es el Constructor de objetos de la clase. Es llamado automáticamente al definir un nuevo objeto de esa clase. Por esa razón, le pasamos los dos argumentos al crear el objeto.
- El primer argumento del método, `self`, debe estar presente en la definición de todos los métodos pero no lo pasamos como argumento cuando hacemos una llamada a la función. **Python** se encarga de pasarlo en forma automática. Lo único relevante de este argumento es que es el primero para todos los métodos, el nombre `self` puede cambiarse por cualquier otro **pero, por convención, no se hace**.

```
P2 = Punto()
```

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

```
<ipython-input-5-b35d1ccc6ec0> in <module>  
----> 1 P2 = Punto()
```

```
TypeError: __init__() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Por supuesto la creación del objeto falla si no le damos ningún argumento porque los argumentos de `__init__` no son opcionales. Modifiquemos eso y aprovechamos para definir algunos otros métodos que pueden ser útiles:

```
from math import atan2  
  
class Punto:  
    "Clase para describir un punto en el espacio"  
  
    def __init__(self, x=0, y=0, z=0):  
        "Inicializa un punto en el espacio"
```

(continué en la próxima página)

(proviene de la página anterior)

```

self.x = x
self.y = y
self.z = z
return None

def angulo_azimuthal(self):
    "Devuelve el ángulo que forma con el eje x, en radianes"
    return atan2(self.y, self.x)

```

```
P1 = Punto(0.5, 0.5)
```

```
P1.angulo_azimuthal()
```

```
0.7853981633974483
```

```
P2 = Punto()
```

```
P2.x
```

```
0
```

```
help(P1)
```

```

Help on Punto in module __main__ object:

class Punto(builtins.object)
|   Punto(x=0, y=0, z=0)
|
|   Clase para describir un punto en el espacio
|
|   Methods defined here:
|
|   __init__(self, x=0, y=0, z=0)
|       Inicializa un punto en el espacio
|
|   angulo_azimuthal(self)
|       Devuelve el ángulo que forma con el eje x, en radianes
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)

```

```
P1.__dict__
```

```
{'x': 0.5, 'y': 0.5, 'z': 0}
```

```

print(P1.angulo_azimuthal())
print(Punto.angulo_azimuthal(P1))

```

```
0.7853981633974483  
0.7853981633974483
```

Vemos que al hacer la llamada a los métodos, omitimos el argumento `self`. El lenguaje traduce nuestro llamado: `P1.angulo_azimuthal()` como `Punto.angulo_azimuthal(P1)` ya que `self` se refiere al objeto que llama al método.

7.3 Herencia

Una de las características de la programación orientada a objetos es la alta reutilización de código. Uno de los mecanismos más importantes es a través de la herencia. Cuando definimos una nueva clase, podemos crearla a partir de un objeto que ya exista. Por ejemplo, utilizando la clase `Punto` podemos definir una nueva clase para describir un vector en el espacio:

```
class Vector(Punto):  
    "Representa un vector en el espacio"  
  
    def suma(self, v2):  
        "Calcula un vector que contiene la suma de dos vectores"  
        print("Aún no implementada la suma de dos vectores")  
        # código calculando v = suma de self + v2  
        # ...  
  
    def producto(self, v2):  
        "Calcula el producto interno entre dos vectores"  
        print("Aún no implementado el producto interno de dos vectores")  
        # código calculando el producto interno pr = v1 . v2  
  
    def abs(self):  
        "Devuelve la distancia del punto al origen"  
        print("Aún no implementado la norma del vector")  
        # código calculando el producto interno pr = v1 . v2
```

Acá hemos definido un nuevo tipo de objeto, llamado `Vector` que se deriva de la clase `Punto`. Veamos cómo funciona:

```
v1 = Vector(2, 3.1)  
v2 = Vector()
```

```
v1
```

```
<__main__.Vector at 0x7f951885e650>
```

```
v1.x
```

```
2
```

```
v1.angulo_azimuthal()
```

```
0.9978301839061905
```

```
v1.x, v1.y, v1.z
```

```
(2, 3.1, 0)
```

```
v2.x, v2.y, v2.z
```

```
(0, 0, 0)
```

```
v = v1.suma(v2)
```

Aún no implementada la suma de dos vectores

```
print(v)
```

None

```
class Vector(Punto):
    "Representa un vector en el espacio"

    def __add__(self, v2):
        "Calcula un vector que contiene la suma de dos vectores"
        print("Aún no implementada la suma de dos vectores")
        # código calculando v = suma de self + v2
        # ...

    def producto(self, v2):
        "Calcula el producto interno entre dos vectores"
        print("Aún no implementado el producto interno de dos vectores")
        # código calculando el producto interno pr = v1 . v2

    def abs(self):
        "Devuelve la distancia del punto al origen"
        print("Aún no implementado la norma del vector")
        # código calculando el producto interno pr = v1 . v2
```

```
v1 = Vector(1,2,3)
v2 = Vector(1,2,-3)
```

```
v1 + v2
```

Aún no implementada la suma de dos vectores

Los métodos que habíamos definido para los puntos del espacio, son accesibles para el nuevo objeto. Además podemos agregar (extender) el nuevo objeto con otros atributos y métodos.

Como vemos, aún no está implementado el cálculo de las distintas funciones, eso forma parte del siguiente

7.4 Ejercicios 06 (a)

1. Implemente los métodos `suma`, `producto` y `abs`

- `suma()` debe retornar un objeto del tipo `Vector` y contener en cada componente la suma de las componentes de los dos vectores que toma como argumento.
- `producto` toma como argumentos dos vectores y retorna un número real
- `abs` toma como argumentos el propio objeto y retorna un número real

Su uso será el siguiente:

```
v1 = Vector(1, 2, 3)
v2 = Vector(3, 2, 1)
v = v1.suma(v2)
pr = v1.producto(v2)
a = v1.abs()
```

7.5 Objetos y clases

7.6 Atributos de clases y de instancias

Las variables que hemos definido pertenecen a cada objeto. Por ejemplo cuando hacemos

```
class Punto:
    "Clase para describir un punto en el espacio"
    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        return None

    def angulo_azimuthal(self):
        "Devuelve el ángulo que forma con el eje x, en radianes"
        return atan2(self.y, self.x)
```

```
p1 = Punto(1, 2, 3)
p2 = Punto(4, 5, 6)
```

cada vez que creamos un objeto de una dada clase, tiene un dato que corresponde al objeto. En este caso tanto `p1` como `p2` tienen un atributo llamado `x`, y cada uno de ellos tiene su propio valor:

```
print(p1.x, p2.x)
```

```
1 4
```

De la misma manera, en la definición de la clase nos referimos a estas variables como `self.x`, indicando que pertenecen a una instancia de una clase (o, lo que es lo mismo: un objeto específico).

También existe la posibilidad de asociar variables (datos) con la clase y no con una instancia de esa clase (objeto). En el siguiente ejemplo, la variable `num_puntos` no pertenece a un `punto` en particular sino a la clase del tipo `Punto`

```
class Punto:
    "Clase para describir un punto en el espacio"
```

(continúe en la próxima página)

(proviene de la página anterior)

```

num_puntos = 0
def __init__(self, x=0, y=0, z=0):
    "Inicializa un punto en el espacio"
    self.x = x
    self.y = y
    self.z = z
    Punto.num_puntos += 1
    return None

```

```

print('Número de puntos:', Punto.num_puntos)
p1 = Punto(1,1,1)
p2 = Punto()
print(p1, p2)
print('Número de puntos:', Punto.num_puntos)

```

```

Número de puntos: 0
<__main__.Punto object at 0x7f4d4884bdd0> <__main__.Punto object at 0x7f4d4884bf90>
Número de puntos: 2

```

p1.__dict__

```
{'x': 1, 'y': 1, 'z': 1}
```

Si estamos contando el número de puntos que tenemos, podemos crear métodos para acceder a ellos y/o manipularlos. Estos métodos no se refieren a una instancia en particular (p1 o p2 en este ejemplo) sino al tipo de objeto Punto (a la clase)

```

del p1
del p2

```

```

class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

    def borrar(self):
        "Borra el punto"
        Punto.num_puntos -= 1

    @classmethod
    def total(cls):
        "Imprime el número total de puntos"
        print("En total hay {} puntos definidos".format(cls.num_puntos))

```

```

print('Número de puntos:', Punto.num_puntos)
p1 = Punto(1,1,1)

```

(continué en la próxima página)

(proviene de la página anterior)

```
p2 = Punto()  
print(p1, p2)  
Punto.total()
```

```
Número de puntos: 0  
<__main__.Punto object at 0x7f4d488557d0> <__main__.Punto object at 0x7f4d488559d0>  
En total hay 2 puntos definidos
```

```
p1.total()
```

```
En total hay 2 puntos definidos
```

```
p1.borrar()  
Punto.total()
```

```
En total hay 1 puntos definidos
```

Sin embargo, no estamos removiendo p1, sólo estamos actualizando el contador:

```
p1.x
```

```
1
```

7.7 Algunos métodos especiales

Hay algunos métodos que **Python** interpreta de manera especial. Ya vimos uno de ellos: `__init__`, que es llamado automáticamente cuando se crea una instancia de la clase.

Similarmente, existe un método `__del__` que Python llama automáticamente cuando borramos un objeto

```
del p1  
del p2
```

```
class Punto:  
    "Clase para describir un punto en el espacio"  
  
    num_puntos = 0  
  
    def __init__(self, x=0, y=0, z=0):  
        "Inicializa un punto en el espacio"  
        self.x = x  
        self.y = y  
        self.z = z  
        Punto.num_puntos += 1  
        return None  
  
    def __del__(self):  
        "Borra el punto y actualiza el contador"  
        Punto.num_puntos -= 1  
  
@classmethod
```

(continué en la próxima página)

(proviene de la página anterior)

```
def total(cls):
    "Imprime el número total de puntos"
    print("En total hay {} puntos definidos".format(cls.num_puntos))
```

```
p1 = Punto(1,1,1)
p2 = Punto()
Punto.total()
del p2
Punto.total()
```

```
En total hay 2 puntos definidos
En total hay 1 puntos definidos
```

p1

```
<__main__.Punto at 0x7f4d4884b210>
```

p2

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

<ipython-input-17-32960d173fa8> in <module>
----> 1 p2

NameError: name 'p2' is not defined
```

Como vemos, al borrar (con `del` en este caso) el objeto, automáticamente se actualiza el contador.

7.7.1 Métodos `__str__` y `__repr__`

El método `__str__` es especial, en el sentido en que puede ser utilizado aunque no lo llamemos explícitamente en nuestro código. En particular, es llamado cuando usamos expresiones del tipo `str(objeto)` o automáticamente cuando se utilizan las funciones `format` y `print()`. El objetivo de este método es que sea legible para los usuarios.

```
p1 = Punto(1,1,1)
```

```
print(p1)
```

```
<__main__.Punto object at 0x7f4d22fc7e50>
```

Rehagamos la clase para definir vectores

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
```

(continué en la próxima página)

(provine de la página anterior)

```
self.x = x
self.y = y
self.z = z
Punto.num_puntos += 1
return None

def __del__(self):
    "Borra el punto y actualiza el contador"
    Punto.num_puntos -= 1

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print("En total hay {} puntos definidos".format(cls.num_puntos))
```

```
p1 = Punto(1,1,0)
```

```
print(p1)
```

```
(x = 1, y = 1, z = 0)
```

```
ss = 'punto en el espacio: {}'.format(p1)
ss
```

```
'punto en el espacio: (x = 1, y = 1, z = 0)'
```

```
p1
```

```
<__main__.Punto at 0x7f4d22f6ea90>
```

Como vemos, si no usamos la función `print()` o `format()` sigue mostrándonos el objeto (que no es muy informativo). Esto puede remediarse agregando el método especial `__repr__`. Este método es el que se llama cuando queremos inspeccionar un objeto. El objetivo de este método es que de información sin ambigüedades.

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

    def __del__(self):
        "Borra el punto y actualiza el contador"
        Punto.num_puntos -= 1
```

(continué en la próxima página)

(proviene de la página anterior)

```

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

def __repr__(self):
    return "Punto(x={}, y={}, z={})".format(self.x, self.y, self.z)

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print("En total hay {} puntos definidos".format(cls.num_puntos))

```

p2 = Punto(0.3, 0.3, 1)

p2

Punto(x=0.3, y=0.3, z=1)

p2.x = 5
p2

Punto(x=5, y=0.3, z=1)

Como vemos ahora tenemos una representación del objeto, que nos da información precisa.

7.7.2 Método __call__

Este método, si existe es ejecutado cuando llamamos al objeto. Si no existe, es un error llamar al objeto:

p2()

```

-----
TypeError                                     Traceback (most recent call last)

<ipython-input-31-dd18cc1831f4> in <module>
----> 1 p2()

TypeError: 'Punto' object is not callable

```

```

class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z

```

(continué en la próxima página)

(provine de la página anterior)

```
Punto.num_puntos += 1
return None

def __del__(self):
    "Borra el punto y actualiza el contador"
    Punto.num_puntos -= 1

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

def __repr__(self):
    return "Punto(x={}, y={}, z={})".format(self.x, self.y, self.z)

def __call__(self):
    return "Ejecuté el objeto: {}".format(self)
#    return str(self)
#    return "{}".format(self)

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print("En total hay {} puntos definidos".format(cls.num_puntos))
```

```
p3 = Punto(1,3,4)
p3
```

```
Punto(x=1, y=3, z=4)
```

```
p3()
```

```
'Ejecuté el objeto: (x = 1, y = 3, z = 4) '
```

7.8 Ejercicios 06 (b)

2. Utilizando la definición de la clase Punto:

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

    def __del__(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```

    "Borra el punto y actualiza el contador"
Punto.num_puntos -= 1

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

def __repr__(self):
    return "Punto(x={}, y={}, z={})".format(self.x, self.y, self.z)

def __call__(self):
    return self.__str__()

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print("En total hay {} puntos definidos".format(cls.num_puntos))

```

Complete la implementación de la clase Vector con los métodos pedidos

```

class Vector(Punto):
    "Representa un vector en el espacio"

    def suma(self, v2):
        "Calcula un vector que contiene la suma de dos vectores"
        print("Aún no implementada la suma de dos vectores")
        # código calculando v = suma de self + v2
        # ...

    def producto(self, v2):
        "Calcula el producto interno entre dos vectores"
        print("Aún no implementado el producto interno de dos vectores")
        # código calculando el producto interno pr = v1 . v2

    def abs(self):
        "Devuelve la distancia del punto al origen"
        print("Aún no implementado la norma del vector")
        # código calculando la magnitud del vector

    def angulo_entre_vectores(self, v2):
        "Calcula el ángulo entre dos vectores"
        print("Aún no implementado el ángulo entre dos vectores")
        angulo = 0
        # código calculando angulo = arccos(v1 * v2 / (|v1||v2|))
        return angulo

    def coordenadas_cilindricas(self):
        "Devuelve las coordenadas cilíndricas del vector como una tupla (r, theta, z)"
        print("No implementada")

    def coordenadas_esfericas(self):
        "Devuelve las coordenadas esféricas del vector como una tupla (r, theta, phi)"
        print("No implementada")

```

3. **PARA ENTREGAR:** Cree una clase Polinomio para representar polinomios. La clase debe guardar los datos representando todos los coeficientes. El grado del polinomio será *menor o igual a 9* (un dígito).

Nota: Utilice el archivo **polinomio_06.py** en el directorio **data**, que renombrará de la forma usual **Apellido_06.py**. Se le pide que programe:

- Un método de inicialización `__init__` que acepte una lista de coeficientes. Por ejemplo para el polinomio $4x^3 + 3x^2 + 2x + 1$ usaríamos:

```
>>> p = Polinomio([1, 2, 3, 4])
```

- Un método `grado` que devuelva el orden del polinomio

```
>>> p = Polinomio([1, 2, 3, 4])
>>> p.grado()
3
```

- Un método `get_coeficientes`, que devuelva una lista con los coeficientes:

```
>>> p.get_coeficientes()
[1, 2, 3, 4]
```

- Un método `set_coeficientes`, que fije los coeficientes de la lista:

```
>>> p1 = Polinomio()
>>> p1.set_coeficientes([1, 2, 3, 4])
>>> p1.get_coeficientes()
[1, 2, 3, 4]
```

- El método `suma_pol` que le sume otro polinomio y devuelva un polinomio (objeto del mismo tipo)
- El método `mul` que multiplica al polinomio por una constante y devuelve un nuevo polinomio
- Un método, `derivada`, que devuelva la derivada de orden `n` del polinomio (otro polinomio):

```
>>> p1 = p.derivada()
>>> p1.get_coeficientes()
[2, 6, 12]
>>> p2 = p.derivada(n=2)
>>> p2.get_coeficientes()
[6, 24]
```

- Un método que devuelva la integral (antiderivada) del polinomio de orden `n`, con constante de integración `cte` (otro polinomio).

```
>>> p1 = p.integrada()
>>> p1.get_coeficientes()
[0, 1, 1, 1, 1]
>>>
>>> p2 = p.integrada(cte=2)
>>> p2.get_coeficientes()
[2, 1, 1, 1, 1]
>>>
>>> p3 = p.integrada(n=3, cte=1.5)
>>> p3.get_coeficientes()
[1.5, 1.5, 0.75, 0.1666666666666666, 0.0833333333333333, 0.05]
```

- El método `eval`, que evalúe el polinomio en un dado valor de `x`.

```
>>> p = Polinomio([1,2,3,4])
>>> p.eval(x=2)
49
>>>
>>> p.eval(x=0.5)
3.25
```

- **(Si puede)** Un método `from_string` que crea un polinomio desde un string en la forma:

```
>>> p = Polinomio()
>>> p.from_string('x^5 + 3x^3 - 2 x+x^2 + 3 - x')
>>> p.get_coeficientes()
[3, -3, 1, 3, 0, 1]
>>>
>>> p1 = Polinomio()
>>> p1.from_string('y^5 + 3y^3 - 2 y + y^2+3', var='y')
>>> p1.get_coeficientes()
[3, -2, 1, 3, 0, 1]
```

- Escriba un método llamado `__str__`, que devuelva un string (que define cómo se va a imprimir el polinomio). Un ejemplo de salida será:

```
>>> p = Polinomio([1,2.1,3,4])
>>> print(p)
4 x^3 + 3 x^2 + 2.1 x + 1
```

- Escriba un método llamado `__call__`, de manera tal que al llamar al objeto, evalúe el polinomio (use el método `eval` definido anteriormente)

```
>>> p = Polinomio([1,2,3,4])
>>> p(x=2)
49
>>>
>>> p(0.5)
3.25
```

- Escriba un método llamado `__add__(self, p)`, que evalúe la suma de polinomios usando el método `suma_pol` definido anteriormente. Eso permitirá usar la operación de suma en la forma:

```
>>> p1 = Polinomio([1,2,3,4])
>>> p2 = Polinomio([1,2,3,4])
>>> p1 + p2
```


Clase 7: Control de versiones y biblioteca standard

Nota: Esta clase está copiada (muy fuertemente) inspirada en las siguientes fuentes:

- [Lecciones de Software Carpentry](#), y la versión en castellano).
 - El libro [Pro Git](#) de Scott Chacon y Ben Straub, y la versión en castellano. Ambos disponibles libremente.
 - El libro [Mastering git](#) escrito por Jakub Narbski.
-

8.1 ¿Qué es y para qué sirve el control de versiones?

El control de versiones permite ir grabando puntos en la historia de la evolución de un proyecto. Esta capacidad nos permite:

- Acceder a versiones anteriores de nuestro trabajo (número ilimitado)
- Trabajar en forma paralela con otras personas sobre un mismo documento.

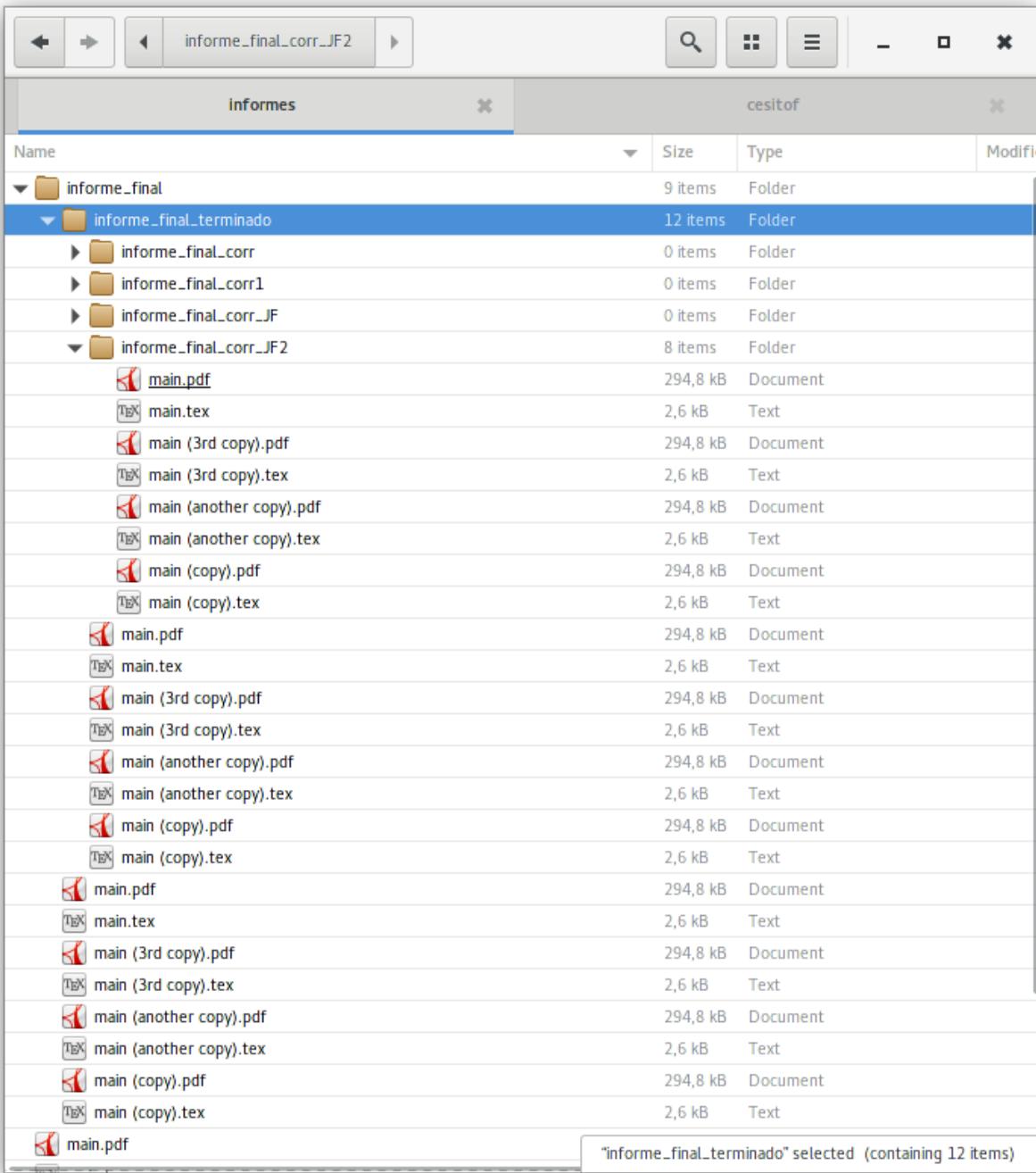
Habitualmente, nos podemos encontrar con situaciones como esta:

o, más gracioso pero igualmente común, esta otra:

Todos hemos estado en esta situación alguna vez: parece ridículo tener varias versiones casi idénticas del mismo documento. Algunos procesadores de texto nos permiten lidiar con esto un poco mejor, como por ejemplo el [Track Changes de Microsoft Word](#), el [historial de versiones de Google](#), o el [Track-changes de LibreOffice](#).

Estas herramientas permiten solucionar el problema del trabajo en colaboración. Si tenemos una versión de un archivo (documento, programa, etc) podemos compartirlo con los otros autores para modificar, y luego ir aceptando o rechazando los cambios propuestos.

Algunos problemas aún aparecen cuando se trabaja intensivamente en un documento, porque al aceptar o rechazar los cambios no queda registro de cuáles eran las alternativas. Además, estos sistemas actúan sólo sobre los documentos; en nuestro caso puede haber datos, gráficos, etc que cambien (o que queremos estar seguros que no cambiaron y estamos usando la versión correcta).



"FINAL".doc



FINAL.doc!



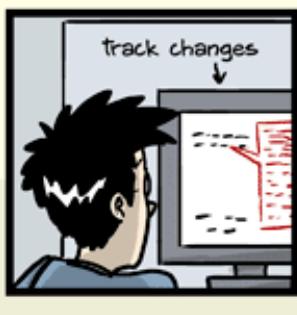
FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10.#@\$%WHYDID
ICOMETOGRAD SCHOOL????.doc



JORGE CHAM © 2012

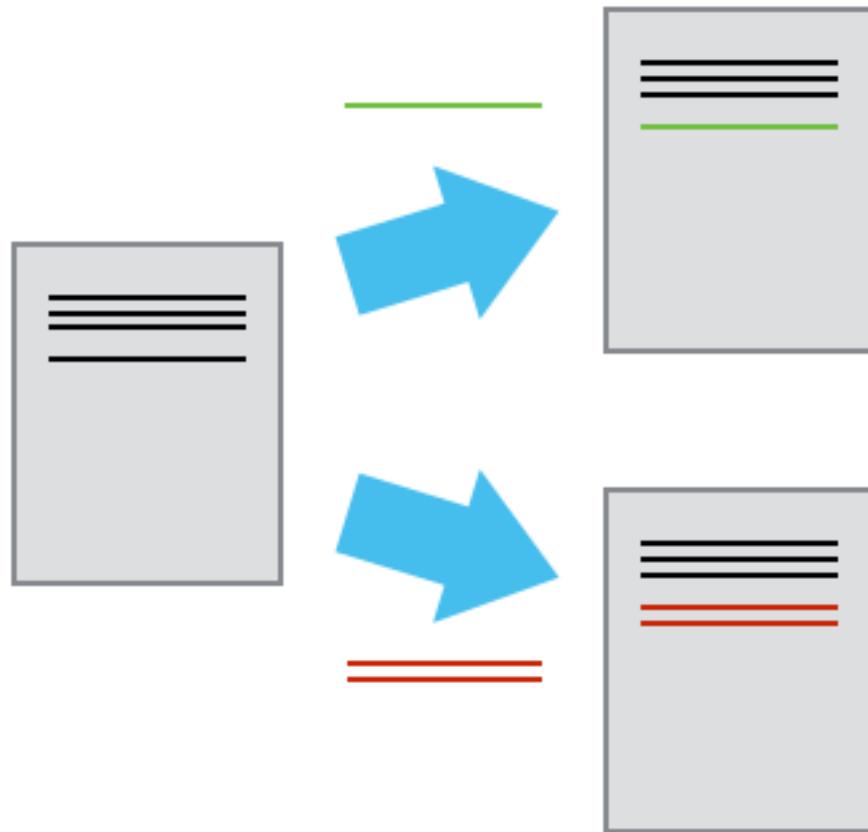
WWW.PHDCOMICS.COM

En el fondo, la manera de evitar esto es manteniendo una buena organización. Una posible buena manera es designar una persona responsable, que vaya llevando la contabilidad de quién hizo qué correcciones, las integre en un único documento, y vaya guardando copias de todos los documentos que recibe en un lugar separado. Cuando hay varios autores (cuatro o cinco) éste es un trabajo bastante arduo y con buenas posibilidades de pequeños errores. Los sistemas de control de versiones tratan de automatizar la mayor parte del trabajo para hacer más fácil la colaboración, manteniendo registro de los cambios que se hicieron desde que se inició el documento, y produciendo poca interferencia, permitiendo al mismo tiempo trabajar de manera similar a como lo hacemos habitualmente.

Consideremos un proyecto con varios archivos y autores. En este esquema de trabajo, podemos compartir una versión de todos los archivos del proyecto

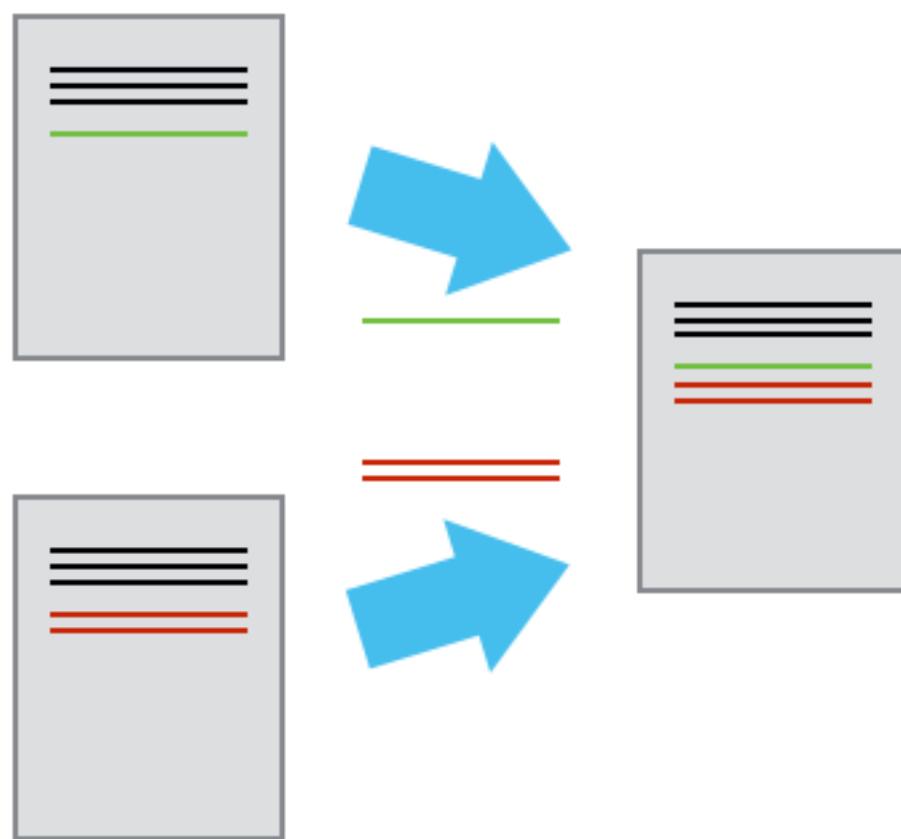
8.1.1 Cambios en paralelo

Una de las ventajas de los sistemas de control de versiones es que cada autor hace su aporte en su propia copia (en paralelo)



y después estos son compatibilizados e incorporados en un único documento.

En casos en que los autores trabajen en zonas distintas la compaginación se puede hacer en forma automática. Por otro lado, si dos personas cambian la misma frase obviamente se necesita tomar una decisión y la compaginación no puede (ni quiere) hacerse automáticamente.



8.1.2 Historia completa

Otra característica importante de los sistemas de control de versiones es que guardan la historia completa de todos los archivos que componen el proyecto. Imaginen, por ejemplo, que escribieron una función para resolver parte de un problema. La función no sólo hace su trabajo sino que está muy bien escrita, es elegante y funciona rápidamente. Unos días después encuentran que toda esa belleza era innecesaria, porque el problema que resuelve la función no aparece en su proyecto, y por supuesto la borra. La versión oficial no tiene esa parte del código.

Dos semanas, y varias líneas de código, después aparece un problema parecido y queríamos tener la función que borramos

Los sistemas de control de versiones guardan toda la información de la historia de cada archivo, con un comentario del autor. Este modo de trabajar nos permite recuperar (casi) toda la información previa, incluso aquella que en algún momento decidimos descartar.

8.2 Instalación y uso: Una versión breve

8.2.1 Instalación

Vamos a describir uno de los posibles sistemas de control de versiones, llamado *git*

En Linux, usando el administrador de programas, algo así como:

en Ubuntu:

```
$ sudo apt-get install git
```

o usando *dnf* en Fedora:

```
$ sudo dnf install git
```

En Windows, se puede descargar [Git for Windows](#) desde [este enlace](#), ejecutar el instalador y seguir las instrucciones. Viene con una terminal y una interfaz gráfica.

8.2.2 Interfaces gráficas

Existen muchas interfaces gráficas, para todos los sistemas operativos.

Ver por ejemplo [Git Extensions](#), [git-cola](#), [Code Review](#), o cualquiera de esta larga lista de [interfaces gráficas a Git](#).

8.2.3 Documentación

Buscando en internet el término *git* se encuentra mucha documentación. En particular el libro *Pro Git* tiene información muy accesible y completa.

El programa se usa en la forma:

```
$ git <comando> [opciones]
```

Por ejemplo, para obtener ayuda directamente desde el programa, se puede utilizar cualquiera de las opciones:

```
$ git config help
$ git config -h
$ git config --help
```

8.2.4 Configuración básica

Una vez que está instalado, es conveniente configurarlo desde una terminal, con los comandos:

```
$ git config --global user.name "Juan Fiol"
$ git config --global user.email "fiol@cab.cnea.gov.ar"
```

Si necesitamos usar un proxy para acceder fuera del lugar de trabajo:

```
$ git config --global http.proxy proxy-url
$ git config --global https.proxy proxy-url
```

Acá hemos usado la opción *--global* para que las variables configuradas se apliquen a todos los repositorios con los que trabajamos.

Si necesitamos desabilitar una variable, por ejemplo el proxy, podemos hacer:

```
$ git config --global unset http.proxy
$ git config --global unset https.proxy
```

8.2.5 Creación un nuevo repositorio

Si ya estamos trabajando en un proyecto, tenemos algunos archivos de trabajo, sin control de versiones, y queremos empezar a controlarlo, inicializamos el repositorio local con:

```
$ git init
```

Este comando sólo inicializa el repositorio, no pone ningún archivo bajo control de versiones.

8.2.6 Clonación de un repositorio existente

Otra situación bastante común ocurre cuando queremos tener una copia local de un proyecto (grupo de archivos) que ya existe y está siendo controlado por git. En este caso utilizamos el comando *clone* en la forma:

```
$ git clone <url-del-repositorio> [nombre-local]
```

donde el argumento *nombre-local* es opcional, si queremos darle a nuestra copia un nombre diferente al que tiene en el repositorio

Ejemplos:

```
$ git clone /home/fiol/my-path/programa
$ git clone /home/fiol/my-path/programa programa-de-calculo
$ git clone https://fiolj@bitbucket.org/fiolj/version-control.git
$ git clone https://gitlab.com/fiolj/intro-python-IB.git
$ git clone https://github.com/fiolj/intro-python-IB.git
```

Los dos primeros ejemplos realizan una copia de trabajo de un proyecto alojado también localmente. En el segundo caso le estamos un nuevo nombre a la copia de trabajo.

En los últimos tres ejemplos estamos copiando proyectos alojados en repositorios remotos, cuyo uso es bastante popular: [bitbucket](#), [gitlab](#), y [github](#).

Lo que estamos haciendo con estos comandos es copiar no sólo la versión actual del proyecto sino toda su historia. Después de ejecutar este comando tendremos en nuestra computadora cada versión de cada uno de los archivos del proyecto, con la información de quién hizo los cambios y cuándo se hicieron.

Una vez que ya tenemos una copia local de un proyecto vamos a querer trabajar: modificar los archivos, agregar nuevos, borrar alguno, etc.

8.2.7 Ver el estado actual

Para determinar qué archivos se cambiaron utilizamos el comando *status*:

```
$ cd my-directorio  
$ git status
```

8.2.8 Creación de nuevos archivos y modificación de existentes

Después de trabajar en un archivo existente, o crear un nuevo archivo que queremos controlar, debemos agregarlo al registro de control:

```
$ git add <nuevo-archivo>  
$ git add <archivo-modificado>
```

Esto sólo agrega la versión actual del archivo al listado a controlar. Para incluir una copia en la base de datos del repositorio debemos realizar lo que se llama un *ícommit*

```
$ git commit -m "Mensaje para recordar que hice con estos archivos"
```

La opción *-m* y su argumento (el *string* entre comillas) es un mensaje que dejamos grabado, asociado a los cambios realizados. Puede realizarse el *commit* sin esta opción, y entonces *git* abrirá un editor de texto para que escribamos el mensaje (que no puede estar vacío).

8.2.9 Actualización de un repositorio remoto

Una vez que se añaden o modifican los archivos, y se agregan al repositorio local, podemos enviar los cambios a un repositorio remoto. Para ello utilizamos el comando:

```
$ git push
```

De la misma manera, si queremos obtener una actualización del repositorio remoto (porque alguien más la modificó), utilizamos el (los) comando(s):

```
$ git fetch
```

Este comando sólo actualiza el repositorio, pero no modifica los archivos locales. Esto se puede hacer, cuando uno quiera, luego con el comando:

```
$ git merge
```

Estos dos comandos, pueden generalmente reemplazarse por un único comando:

```
$ git pull
```

que realizará la descarga desde el repositorio remoto y la actualización de los archivos locales en un sólo paso.

8.2.10 Puntos importantes

Control de versiones	Historia de cambios y ñundoz ilimitado
Configuración	<i>git config</i> , con la opción <i>-global</i>
Creación	<i>git init</i> inicializa el repositorio
	<i>git clone</i> copia un repositorio
Modificación	<i>git status</i> muestra el estado actual
	<i>git add</i> pone archivos bajo control
	<i>git commit</i> graba la versión actual
Explorar las versiones	<i>git log</i> muestra la historia de cambios
	<i>git diff</i> compara versiones
	<i>git checkout</i> recupera versiones previas
Comunicación con remotos	<i>git push</i> Envía los cambios al remoto
	<i>git pull</i> copia los cambios desde remoto

8.3 Algunos módulos (biblioteca standard)

Los módulos pueden pensarse como bibliotecas de objetos (funciones, datos, etc) que pueden usarse según la necesidad. Hay una biblioteca standard con rutinas para muchas operaciones comunes, y además existen muchos paquetes específicos para distintas tareas. Veamos algunos ejemplos:

8.3.1 Módulo sys

Este módulo da acceso a variables que usa o mantiene el intérprete Python

```
import sys

sys.path

['/home/fiol/trabajo/clases/clase-python/clases',
 '/usr/lib64/python37.zip',
 '/usr/lib64/python3.7',
 '/usr/lib64/python3.7/lib-dynload',
 '',
 '/home/fiol/.local/lib/python3.7/site-packages',
 '/home/fiol/.local/lib/python3.7/site-packages/sphinx_fortran-1.0.1-py3.7.egg',
 '/usr/lib64/python3.7/site-packages',
 '/usr/lib/python3.7/site-packages',
 '/usr/lib/python3.7/site-packages/IPython/extensions',
 '/home/fiol/.ipython']
```

```
sys.getfilesystemencoding()
```

```
'utf-8'
```

```
sys.getsizeof(1)
```

```
28
```

```
help(sys.getsizeof)
```

```
Help on built-in function getsizeof in module sys:  
  
getsizeof(...)  
    getsizeof(object, default) -> int  
  
    Return the size of object in bytes.
```

Vemos que para utilizar las variables (path) o funciones (getsizeof) debemos referirlo anteponiendo el módulo en el cuál está definido (sys) y separado por un punto.

Cuando hacemos un programa, con definición de variables y funciones. Podemos utilizarlo como un módulo, de la misma manera que los que ya vienen definidos en la biblioteca standard o en los paquetes que instalamos.

8.3.2 Módulo os

El módulo os tiene utilidades para operar sobre nombres de archivos y directorios de manera segura y portable, de manera que pueda utilizarse en distintos sistemas operativos. Vamos a ver ejemplos de uso de algunas facilidades que brinda:

```
import os  
  
print(os.curdir)  
print(os.pardir)  
print(os.getcwd())
```

```
.  
..  
/home/fiol/trabajo/clases/clase-python/clases
```

```
cur = os.getcwd()  
par = os.path.abspath("..")  
print(cur)  
print(par)
```

```
/home/fiol/trabajo/clases/clase-python/clases  
/home/fiol/trabajo/clases/clase-python
```

```
print(os.path.abspath(os.curdir))  
print(os.getcwd())
```

```
/home/fiol/trabajo/clases/clase-python/clases  
/home/fiol/trabajo/clases/clase-python/clases
```

```
print(os.path.basename(cur))  
print(os.path.splitdrive(cur))
```

```
clases  
(', '/home/fiol/trabajo/clases/clase-python/clases')
```

```
print(os.path.commonprefix((cur, par)))
archivo = os.path.join(par, 'este', 'otro.dat')
print(archivo)
print(os.path.split(archivo))
print(os.path.splitext(archivo))
print(os.path.exists(archivo))
print(os.path.exists(cur))
```

```
/home/fiol/trabajo/clases/clase-python
/home/fiol/trabajo/clases/clase-python/este/otro.dat
('/home/fiol/trabajo/clases/clase-python/este', 'otro.dat')
('/home/fiol/trabajo/clases/clase-python/este/otro', '.dat')
False
True
```

Como es aparente de estos ejemplos, se puede acceder a todos los objetos (funciones, variables) de un módulo utilizando simplemente la línea `import <modulo>` pero puede ser tedioso escribir todo con prefijos (como `os.path.abspath`) por lo que existen dos alternativas que pueden ser más convenientes. La primera corresponde a importar todas las definiciones de un módulo en forma implícita:

```
from os import *
```

Después de esta declaración usamos los objetos de la misma manera que antes pero obviando la parte de `os`.

```
path.abspath(curdir)
```

```
'/home/fiol/trabajo/clases/clase-python/clases'
```

Esto es conveniente en algunos casos pero no suele ser una buena idea en programas largos ya que distintos módulos pueden definir el mismo nombre, y se pierde información sobre su origen. Una alternativa que es conveniente y permite mantener mejor control es importar explícitamente lo que vamos a usar:

```
from os import curdir, pardir, getcwd
from os.path import abspath
print(abspath(pardir))
print(abspath(curdir))
print(abspath(getcwd()))
```

```
/home/fiol/trabajo/clases/clase-python
/home/fiol/trabajo/clases/clase-python/clases
/home/fiol/trabajo/clases/clase-python/clases
```

Además podemos darle un nombre diferente al importar módulos u objetos

```
import os.path as path
from os import getenv as ge
```

```
help(ge)
```

```
Help on function getenv in module os:

getenv(key, default=None)
    Get an environment variable, return None if it doesn't exist.
    The optional second argument can specify an alternate default.
    key, default and the result are str.
```

Clases de Python

```
ge('HOME')
```

```
'/home/fiol'
```

```
path.realpath(curdir)
```

```
'/home/fiol/trabajo/clases/clase-python/clases'
```

Acá hemos importado el módulo `os.path` (es un sub-módulo) como `path` y la función `getenv` del módulo `os` y la hemos renombrado `ge`.

```
help(os.walk)
```

```
Help on function walk in module os:
```

```
walk(top, topdown=True, onerror=None, followlinks=False)
    Directory tree generator.
```

For each directory `in` the directory tree rooted at `top` (including `top` itself, but excluding `..` and `...`), yields a 3-tuple

```
    dirpath, dirnames, filenames
```

`dirpath` **is** a string, the path to the directory. `dirnames` **is** a `list` of the names of the subdirectories `in` `dirpath` (excluding `..` and `...`). `filenames` **is** a `list` of the names of the non-directory files `in` `dirpath`. Note that the names `in` the lists are just names, **with** no path components. To get a full path (which begins **with** `top`) to a file **or** directory `in` `dirpath`, do `os.path.join(dirpath, name)`.

If optional arg '`topdown`' **is** true **or** not specified, the triple `for` a directory **is** generated before the triples `for` any of its subdirectories (directories are generated top down). If `topdown` **is** false, the triple `for` a directory **is** generated after the triples `for` all of its subdirectories (directories are generated bottom up).

When `topdown` **is** true, the caller can modify the `dirnames` `list` **in-place** (e.g., via `del` **or** `slice` assignment), **and** `walk` will only recurse into the subdirectories whose names remain `in` `dirnames`; this can be used to prune the search, **or** to impose a specific order of visiting. Modifying `dirnames` when `topdown` **is** false has no effect on the behavior of `os.walk()`, since the directories `in` `dirnames` have already been generated by the time `dirnames` itself **is** generated. No matter the value of `topdown`, the `list` of subdirectories **is** retrieved before the tuples `for` the directory **and** its subdirectories are generated.

By default errors `from the` `os.scandir()` call are ignored. If optional arg '`onerror`' **is** specified, it should be a function; it will be called **with** one argument, an `OSError` instance. It can report the error to `continue with` the `walk`, **or** `raise` the exception to abort the `walk`. Note that the filename **is** available **as** the `filename` attribute of the exception `object`.

By default, `os.walk` does **not** follow symbolic links to subdirectories on systems that support them. In order to get this functionality, `set` the optional argument '`followlinks`' to true.

(continué en la próxima página)

(proviene de la página anterior)

Caution: **if** you **pass** a relative pathname **for** top, don't change the current working directory between resumptions of walk. walk never changes the current directory, **and** assumes that the client doesn't either.

Example:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([getsize(join(root, name)) for name in files]), end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS') # don't visit CVS directories
```

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('./'):
    print(root, "consume ", end="")
    print(sum([getsize(join(root, name)) for name in files])/1024, end="")
    print(" kbytes en ", len(files), "non-directory files")
    if '.ipynb_checkpoints' in dirs:
        dirs.remove('.ipynb_checkpoints') # don't visit CVS directories
```

```
./ consume 14067.50390625 kbytes en 81 non-directory files
./13_fft_files consume 1075.671875 kbytes en 16 non-directory files
./scripts consume 24.359375 kbytes en 32 non-directory files
./scripts/interfacing consume 4.5634765625 kbytes en 6 non-directory files
./scripts/animaciones consume 17.5595703125 kbytes en 11 non-directory files
./09_intro_visualizacion_files consume 421.7919921875 kbytes en 29 non-directory_
↪files
./14_interactivo_files consume 364.30078125 kbytes en 9 non-directory files
./05_algunos_ejemplos_files consume 14.99609375 kbytes en 1 non-directory files
./15_interfacing_y_animaciones_files consume 4.05859375 kbytes en 1 non-directory_
↪files
./version-control consume 55.796875 kbytes en 10 non-directory files
./version-control/_images consume 200.9833984375 kbytes en 4 non-directory files
./version-control/_sources consume 15.384765625 kbytes en 4 non-directory files
./version-control/_static consume 827.212890625 kbytes en 25 non-directory files
./version-control/_static/css consume 116.8828125 kbytes en 2 non-directory files
./version-control/_static/js consume 19.341796875 kbytes en 2 non-directory files
./version-control/_static/fonts consume 4132.6220703125 kbytes en 13 non-directory_
↪files
./version-control/_static/fonts/Lato consume 5672.4013671875 kbytes en 16 non-
↪directory files
./version-control/_static/fonts/RobotoSlab consume 786.3271484375 kbytes en 8 non-
↪directory files
./__pycache__ consume 0.828125 kbytes en 1 non-directory files
./08_intro_numpy_files consume 11.6572265625 kbytes en 1 non-directory files
./13_graficacion3d_files consume 818.3916015625 kbytes en 25 non-directory files
./13_misclaneas_files consume 1638.3798828125 kbytes en 17 non-directory files
./12_fiteos_files consume 797.744140625 kbytes en 28 non-directory files
./10_mas_numpy_files consume 124.7763671875 kbytes en 7 non-directory files
./11_intro_scipy_files consume 232.6796875 kbytes en 10 non-directory files
```

8.3.3 Módulo subprocess

El módulo subprocess permite ejecutar nuevos procesos, proveerle de datos de entrada, y capturar su salida.

```
import subprocess as sub

sub.run(["ls", "-l"])

CompletedProcess(args=['ls', '-l'], returncode=0)
```

En esta forma, la función `run` ejecuta el comando `ls` (listar) con el argumento `-l`, y **no** captura la salida. Si queremos guardar la salida, podemos usar el argumento `stdout`:

```
ll = sub.run(["ls", "-l"], stdout=sub.PIPE)
```

La variable `ll` tiene el objeto retornado por `run`, y podemos acceder a la salida mediante `ll.stdout`

```
ff= ll.stdout.splitlines()

for f in ff:
    if 'ipynb' in str(f) and '04_' in str(f):
        print(f.decode('utf-8'))
```



```
-rw-r--r--. 1 fiol fiol      5731 feb 26 14:20 04_ejercicios.ipynb
-rw-r--r--. 1 fiol fiol     40101 feb 27 09:15 04_funciones.ipynb
-rw-r--r--. 1 fiol fiol     32878 feb 27 09:22 04_iteraciones.ipynb
```

8.3.4 Módulo glob

El módulo `glob` encuentra nombres de archivos (o directorios) utilizando patrones similares a los de la consola. La función más utilizada es `glob.glob()`. Veamos algunos ejemplos de uso:

```
import glob

nb_clase4= glob.glob('04*.ipynb')

nb_clase4

['04_ejercicios.ipynb', '04_funciones.ipynb', '04_iteraciones.ipynb']

nb_clase4.sort()

nb_clase4

['04_ejercicios.ipynb', '04_funciones.ipynb', '04_iteraciones.ipynb']

nb_clases1a4 = glob.glob('0[0-4]*.ipynb')

nb_clases1a4
```

```
[ '04_ejercicios.ipynb',
  '04_funciones.ipynb',
  '01_ejercicios.ipynb',
  '00_introd_y_excursion.ipynb',
  '04_iteraciones.ipynb',
  '02_tipos_y_control.ipynb',
  '03_ejercicios.ipynb',
  '01_instala_y_uso.ipynb',
  '03_bases_python.ipynb',
  '02_ejercicios.ipynb',
  '01_introd_python.ipynb']
```

```
for f in sorted(nb_clases1a4):
    print('Clase en archivo {}'.format(f))
```

```
Clase en archivo 00_introd_y_excursion.ipynb
Clase en archivo 01_ejercicios.ipynb
Clase en archivo 01_instala_y_uso.ipynb
Clase en archivo 01_introd_python.ipynb
Clase en archivo 02_ejercicios.ipynb
Clase en archivo 02_tipos_y_control.ipynb
Clase en archivo 03_bases_python.ipynb
Clase en archivo 03_ejercicios.ipynb
Clase en archivo 04_ejercicios.ipynb
Clase en archivo 04_funciones.ipynb
Clase en archivo 04_iteraciones.ipynb
```

8.3.5 Módulo Argparse

Este módulo tiene lo necesario para hacer rápidamente un programa para utilizar por línea de comandos, aceptando todo tipo de argumentos y dando información sobre su uso.

```
import argparse
VERSION = 1.0

parser = argparse.ArgumentParser(
    description='''Mi programa que acepta argumentos por línea de comandos''')

parser.add_argument('-V', '--version', action='version',
                    version='%(prog)s version {}'.format(VERSION))

parser.add_argument('-n', '--entero', action=store, dest='n', default=1)

args = parser.parse_args()
```

Más información en la [biblioteca standard](#) y en [Argparse](#) en Python Module of the week

8.3.6 Módulo re

Este módulo provee la infraestructura para trabajar con *regular expressions*, es decir para encontrar expresiones que verifican cierta forma general. Veamos algunos conceptos básicos y casos más comunes de uso.

Búsqueda de un patrón en un texto

Empecemos con un ejemplo bastante común. Para encontrar un patrón en un texto podemos utilizar el método `search()`

```
import re
```

```
busca = 'un'
texto = 'Otra vez vamos a usar "Hola Mundo"'

match = re.search(busca, texto)

print('Encontré "{}"\nen: "{}"'.format(match.re.pattern, match.string))
print('En las posiciones {} a {}'.format(match.start(), match.end()))
```

```
Encontré "un"
en:
 "Otra vez vamos a usar "Hola Mundo""
En las posiciones 29 a 31
```

Acá buscamos una expresión (el substring `un`). Esto es útil pero no muy diferente a utilizar los métodos de strings. Veamos como se definen los patrones.

Definición de expresiones

Vamos a buscar un patrón en un texto. Veamos cómo se definen los patrones a buscar.

- La mayoría de los caracteres se identifican consigo mismo (si quiero encontrar `gato`, uso como patrón `gato`)
- Hay unos pocos caracteres especiales (metacaracteres) que tienen un significado especial, estos son:

```
. ^ $ * + ? { } [ ] \ | ( )
```

- Si queremos encontrar uno de los metacaracteres, tenemos que precederlos de `\`. Por ejemplo si queremos encontrar un corchete usamos `\[`
- Los corchetes `[ž y ñ]` se usan para definir una clase de caracteres, que es un conjunto de caracteres que uno quiere encontrar.
 - Los caracteres a encontrar se pueden dar individualmente. Por ejemplo `[gato]` encontrará cualquiera de `g, a, t, o`.
 - Un rango de caracteres se puede dar dando dos caracteres separados por un guión. Por ejemplo `[a-z]` dará cualquier letra entre `a` y `z`. Similarmente `[0-5] [0-9]` dará cualquier número entre `00` y `59`.
 - Los metacaracteres pierden su significado especial dentro de los corchetes. Por ejemplo `[.*)]` encontrará cualquiera de `, ñ*,)ž`.
- El punto `.` indica *cualquier carácter*
- Los símbolos `*, +, ?` indican repetición:
 - `?`: Indica 0 o 1 aparición de lo anterior
 - `*`: Indica 0 o más apariciones de lo anterior
 - `+`: Indica 1 o más apariciones de lo anterior

```

busca = "[a-z]+@[a-z]+\.[a-z]+" # Un patrón para buscar direcciones de email
texto = "nombre@server.com, apellido@server1.com, nombre1995@server.com, "
      ↪UnNombreyApellido, nombre.apellido82@servidor.com.ar, Nombre.Apellido82@servidor.
      ↪com.ar".split(',')
print(texto, '\n')

for direc in texto:
    m= re.search(busca, direc)
    print('Para la línea:', direc)
    if m is None:
        print('    No encontré dirección de correo!')
    else:
        print('    Encontré la dirección de correo:', m.string)

```

```

['nombre@server.com', ' apellido@server1.com', ' nombre1995@server.com', ' '
 ↪UnNombreyApellido', ' nombre.apellido82@servidor.com.ar', ' Nombre.
 ↪Apellido82@servidor.com.ar']

Para la línea: nombre@server.com
    Encontré la dirección de correo: nombre@server.com
Para la línea: apellido@server1.com
    No encontré dirección de correo!
Para la línea: nombre1995@server.com
    No encontré dirección de correo!
Para la línea: UnNombreyApellido
    No encontré dirección de correo!
Para la línea: nombre.apellido82@servidor.com.ar
    No encontré dirección de correo!
Para la línea: Nombre.Apellido82@servidor.com.ar
    No encontré dirección de correo!

```

- Acá la expresión [a-z] significa todos los caracteres en el rango a hasta z.
- [a-z]+ significa cualquier secuencia de una letra o más.
- Los corchetes también se pueden usar en la forma [abc] y entonces encuentra *cualquiera* de a, b, o c.

Vemos que no encontró todas las direcciones posibles. Porque el patrón no está bien diseñado. Un poco mejor sería:

```

busca = "[a-zA-Z0-9.]+@[a-zA-Z.]+# Un patrón para buscar direcciones de email

print(texto, '\n')

for direc in texto:
    m= re.search(busca, direc)
    print('Para la línea:', direc)
    if m is None:
        print('    No encontré dirección de correo!')
    else:
        print('    Encontré la dirección de correo:', m.group())

```

```

['nombre@server.com', ' apellido@server1.com', ' nombre1995@server.com', ' '
 ↪UnNombreyApellido', ' nombre.apellido82@servidor.com.ar', ' Nombre.
 ↪Apellido82@servidor.com.ar']

Para la línea: nombre@server.com
    Encontré la dirección de correo: nombre@server.com
Para la línea: apellido@server1.com

```

(continúe en la próxima página)

(proviene de la página anterior)

```
Encontré la dirección de correo: apellido@server
Para la línea: nombre1995@server.com
Encontré la dirección de correo: nombre1995@server.com
Para la línea: UnNombreyApellido
No encontré dirección de correo:
Para la línea: nombre.apellido82@servidor.com.ar
Encontré la dirección de correo: nombre.apellido82@servidor.com.ar
Para la línea: Nombre.Apellido82@servidor.com.ar
Encontré la dirección de correo: Nombre.Apellido82@servidor.com.ar
```

Los metacaracteres no se activan dentro de clases (adentro de corchetes). En el ejemplo anterior el punto . actúa como un punto y no como un metacaracter. En este caso, la primera parte: [a-zA-Z0-9.] + significa: Encontrar cualquier letra minúscula, mayúscula, número o punto, una o más veces cualquiera de ellos

Repetición de un patrón

Si queremos encontrar strings que presentan la secuencia una o más veces podemos usar `findall()` que devuelve todas las ocurrencias del patrón que no se superponen. Por ejemplo:

```
texto = 'abbaaabbbbaaaaa'
busca = 'ab'

mm = re.findall(busca, texto)
print(mm)

for m in mm:
    print('Encontré {}'.format(m))
```

```
[ 'ab', 'ab']
Encontré ab
Encontré ab
```

```
p = re.compile('abc*')
m= p.findall('acholaboy')
print(m)
m= p.findall('acholabcoynd sabcccs slabc labdc abc')
print(m)
```

```
[ 'ab']
[ 'abc', 'abccc', 'abc', 'ab', 'abc']
```

Si va a utilizar expresiones regulares es recomendable que lea más información en la [biblioteca standard](#), en el [HOWTO](#) y en [Python Module of the week](#).

CAPÍTULO 9

Clase 8: Introducción a Numpy

9.1 Algunos ejemplos

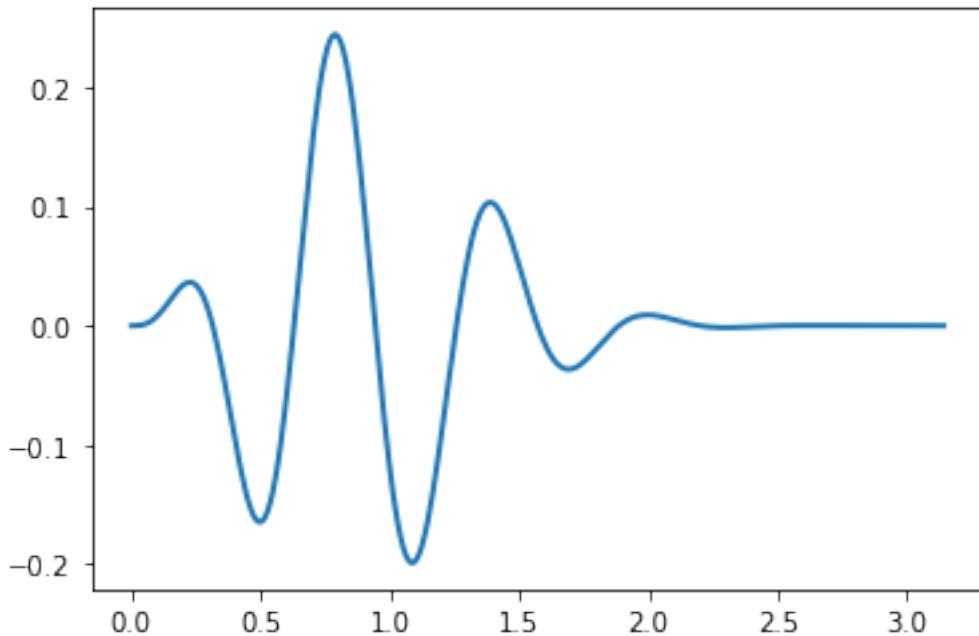
Dos paquetes que van a resultar muy importantes para nosotros son los paquetes **numpy** y **matplotlib**. Como con todos los módulos, se cargan utilizando la palabra `import`, tal como hicimos en los ejemplos anteriores. Existen variantes en la manera de importar los módulos que son equivalentes. En este caso le vamos a dar un alias que sea más corto de tipar. Después podemos utilizar sus funciones y definiciones.

9.1.1 Graficación de datos de archivos

```
import numpy as np
import matplotlib.pyplot as plt

x, y = np.loadtxt('../data/ejemplo_plot_07_1.dat', unpack=True)
plt.plot(x, y)

[<matplotlib.lines.Line2D at 0x7fa11af9cf0>]
```



Como vemos es muy simple cargar datos de un archivo y graficarlos. Veamos qué datos hay en el archivo:

```
!head ./data/ejemplo_plot_07_1.dat
```

```
#      x          f (x)
0.000000e+00 0.000000e+00
1.050700e-02 1.157617e-05
2.101400e-02 9.205287e-05
3.152100e-02 3.075650e-04
4.202800e-02 7.187932e-04
5.253499e-02 1.378428e-03
6.304199e-02 2.328857e-03
7.354899e-02 3.600145e-03
8.405599e-02 5.208356e-03
```

Hay dos columnas, en la primera fila hay texto, y en las siguientes hay valores separados por un espacio. En la primera línea, la función `np.loadtxt()` carga estos valores a las variables `x` e `y`, y en la segunda los graficamos. Inspeccionemos las variables

```
len(x)
```

```
300
```

```
x[:10]
```

```
array([0.          , 0.010507  , 0.021014  , 0.031521  , 0.042028  ,
       0.05253499, 0.06304199, 0.07354899, 0.08405599, 0.09456299])
```

```
type(x), type(y)
```

```
(numpy.ndarray, numpy.ndarray)
```

Como vemos, el tipo de la variable **no es una lista** sino un nuevo tipo: **ndarray**, o simplemente **array**. Veamos cómo trabajar con ellos.

9.1.2 Comparación de listas y arrays

Comparemos como operamos sobre un conjunto de números cuando los representamos por una lista, o por un array:

```
dlist = [1.5, 3.8, 4.9, 12.3, 27.2, 35.8, 70.2, 90., 125., 180.]
```

```
d = np.array(dlist)
```

```
d is dlist
```

```
False
```

```
print(dlist)
```

```
[1.5, 3.8, 4.9, 12.3, 27.2, 35.8, 70.2, 90.0, 125.0, 180.0]
```

```
print(d)
```

```
[ 1.5  3.8  4.9  12.3  27.2  35.8  70.2  90.  125.  180. ]
```

Veamos cómo se hace para operar con estos dos tipos. Si los valores representan ángulos en grados, hagamos la conversión a radianes ($\text{radian} = \pi/180$ grado)

```
from math import pi
drlist= [a*pi/180 for a in dlist]
```

```
print(drlist)
```

```
[0.02617993877991494, 0.06632251157578452, 0.08552113334772216, 0.21467549799530256, ↵
↳ 0.47472955654245763, 0.62482787221397, 1.2252211349000193, 1.5707963267948966, 2. ↵
↳ 1816615649929116, 3.141592653589793]
```

```
dr= d*(np.pi/180)
```

```
print(dr)
```

```
[0.02617994 0.06632251 0.08552113 0.2146755 0.47472956 0.62482787
1.22522113 1.57079633 2.18166156 3.14159265]
```

Vemos que el modo de trabajar es más simple ya que los array permiten trabajar con operaciones elemento-a-elemento mientras que para las listas tenemos que usar comprensiones de listas. Veamos otros ejemplos:

```
print([np.sin(a*pi/180) for a in dlist])
```

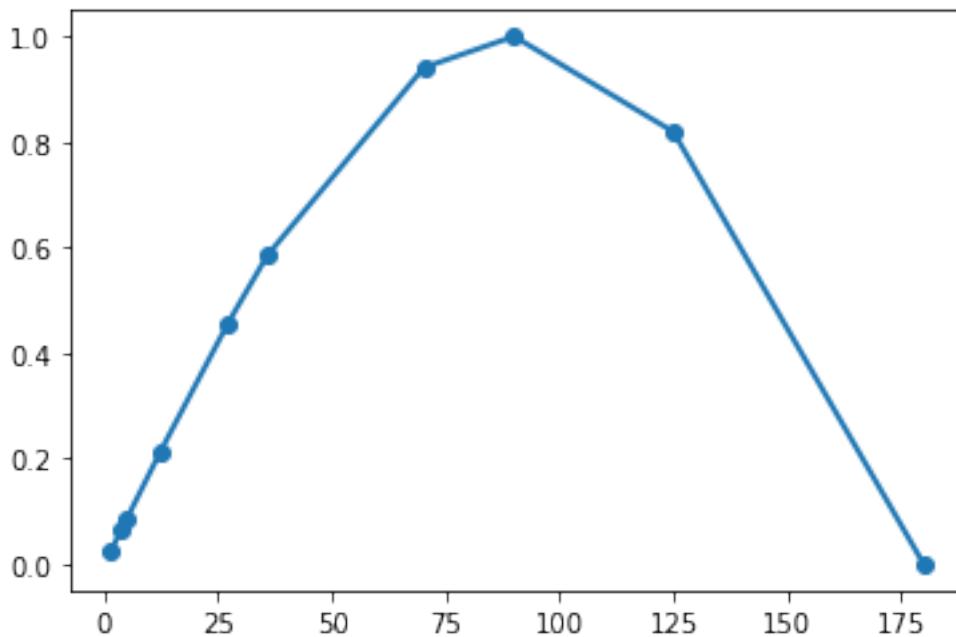
```
[0.02617694830787315, 0.06627390040000014, 0.08541692313736747, 0.21303038627497659, ↵
↳ 0.4570979270586942, 0.5849576749872154, 0.9408807689542255, 1.0, 0.819152044288992, ↵
↳ 1.2246467991473532e-16]
```

```
print(np.sin(np.deg2rad(d)))
```

```
[2.61769483e-02 6.62739004e-02 8.54169231e-02 2.13030386e-01  
 4.57097927e-01 5.84957675e-01 9.40880769e-01 1.00000000e+00  
 8.19152044e-01 1.22464680e-16]
```

Además de la simplicidad para trabajar con operaciones que actúan sobre cada elemento, el paquete tiene una gran cantidad de funciones y constantes definidas (como por ejemplo `np.pi` para π).

```
plt.plot(d, np.sin(np.deg2rad(d)), 'o-')  
plt.show()
```



9.1.3 Generación de datos equiespaciados

Para obtener datos equiespaciados hay dos funciones complementarias

```
a1 = np.arange(0,190,10)  
a2 = np.linspace(0,180,19)
```

```
a1
```

```
array([ 0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110, 120,  
       130, 140, 150, 160, 170, 180])
```

```
a2
```

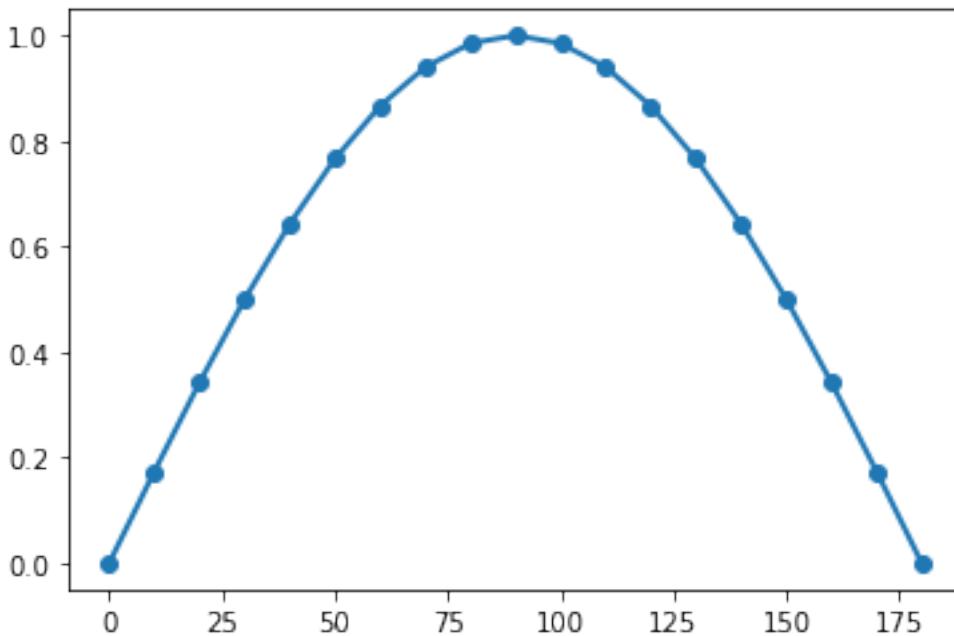
```
array([ 0.,  10.,  20.,  30.,  40.,  50.,  60.,  70.,  80.,  90., 100.,  
       110., 120., 130., 140., 150., 160., 170., 180.])
```

Como vemos, ambos pueden dar resultados similares, y es una cuestión de conveniencia cual utilizar. El uso es:

```
np.arange([start[, stop[, step[,], dtype=None]
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

Mientras que a `arange()` le decimos cuál es el paso a utilizar, a `linspace()` debemos (podemos) darle como tercer argumento el número de valores que queremos.

```
plt.plot(a2, np.sin(np.deg2rad(a2)), 'o-')
plt.show()
```



```
# Pedimos que devuelva el paso también
v1, step1 = np.linspace(0,10,20, endpoint=True, retstep=True)
v2, step2 = np.linspace(0,10,20, endpoint=False, retstep=True)
```

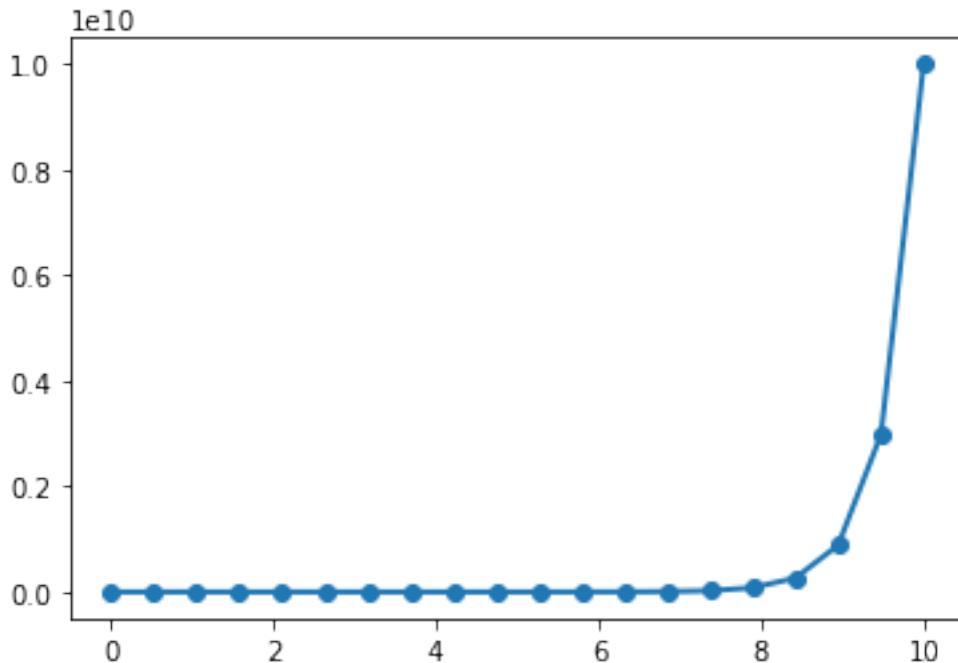
```
print(step1)
print(step2)
```

```
0.5263157894736842
0.5
```

Además de valores linealmente espaciados podemos obtener valores espaciados en escala logarítmica

```
w= np.logspace(0,10,20)
```

```
plt.plot(v1, w, 'o-')
plt.show()
```



```
w1 = np.logspace(0,2,3) # Start y Stop son los exponentes  
print(w1)
```

```
[ 1. 10. 100.]
```

```
w2 = np.geomspace(1,100,3) # Start y Stop son los valores  
print(w2)
```

```
[ 1. 10. 100.]
```

9.2 Características de arrays en Numpy

Numpy define unas nuevas estructuras llamadas *ndarrays* o *arrays* para trabajar con vectores de datos, en una dimensión o más dimensiones (matrices). Los arrays son variantes de las listas de python preparadas para trabajar a mayor velocidad y menor consumo de memoria. Por ello se requiere que los arrays sean menos generales y versátiles que las listas usuales. Analicemos brevemente las diferencias entre estos tipos y las consecuencias que tendrá en su uso para nosotros.

9.2.1 Uso de memoria de listas y arrays

Las listas son sucesiones de elementos, completamente generales y no necesariamente todos iguales. Un esquema de su representación interna se muestra en el siguiente gráfico para una lista de números enteros (Las figuras y el análisis de esta sección son de www.python-course.eu/numpy.php)

Básicamente en una lista se guarda información común a cualquier lista, un lugar de almacenamiento que referencia donde buscar cada uno de sus elementos (que puede ser un objeto diferente) y luego el lugar efectivo para guardar cada elemento. Veamos cuanta memoria se necesita para guardar una lista de enteros:

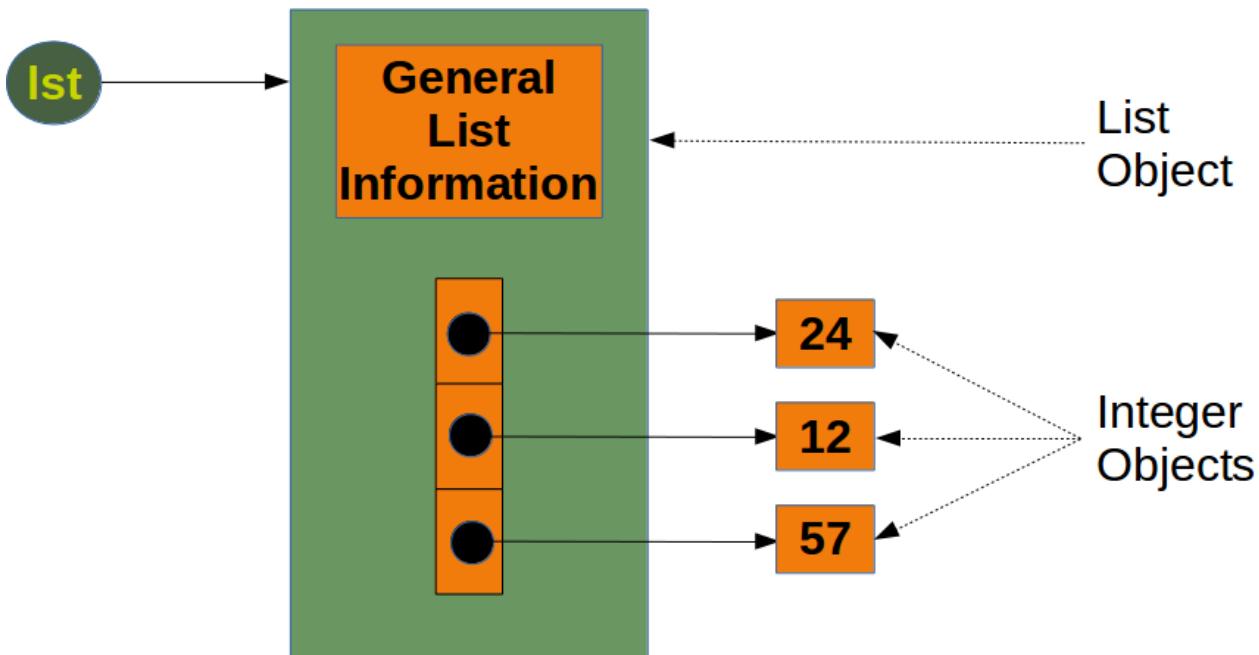


Figura 1: Representación en memoria de una lista

```
from sys import getsizeof
lst = [24, 12, 57]
size_of_list_object = getsizeof(lst)      # only green box
#size_of_elements = getsizeof(lst[0]) + getsizeof(lst[1]) + getsizeof(lst[2])
size_of_elements = sum(getsizeof(l) for l in lst)
total_list_size = size_of_list_object + size_of_elements
print("Tamaño sin considerar los elementos: ", size_of_list_object)
print("Tamaño de los elementos: ", size_of_elements)
print("Tamaño total: ", total_list_size)
```

```
Tamaño sin considerar los elementos:  96
Tamaño de los elementos:  84
Tamaño total:  180
```

Para calcular cuánta memoria se usa en cada parte de una lista analicemos el tamaño de distintos casos:

```
print('Una lista vacía ocupa: {} bytes'.format(getsizeof([])))
print('Una lista con un elem: {} bytes'.format(getsizeof([24])))
print('Una lista con 2 elems: {} bytes'.format(getsizeof([24,12])))
print('Un entero en Python : {} bytes'.format(getsizeof(24)))
```

```
Una lista vacía ocupa: 72 bytes
Una lista con un elem: 80 bytes
Una lista con 2 elems: 88 bytes
Un entero en Python : 28 bytes
```

Vemos que la Información general de listas ocupa **64 bytes**, y la referencia a cada elemento entero ocupa adicionalmente **8 bytes**. Además, cada elemento, un entero de Python, en este caso ocupa **28 bytes**, por lo que el tamaño total de una **lista** de n números enteros será:

$$M_L(n) = 64 + n \times 8 + n \times 28$$

En contraste, los *arrays* deben ser todos del mismo tipo por lo que su representación es más simple (por ejemplo, no es necesario guardar sus valores separadamente)

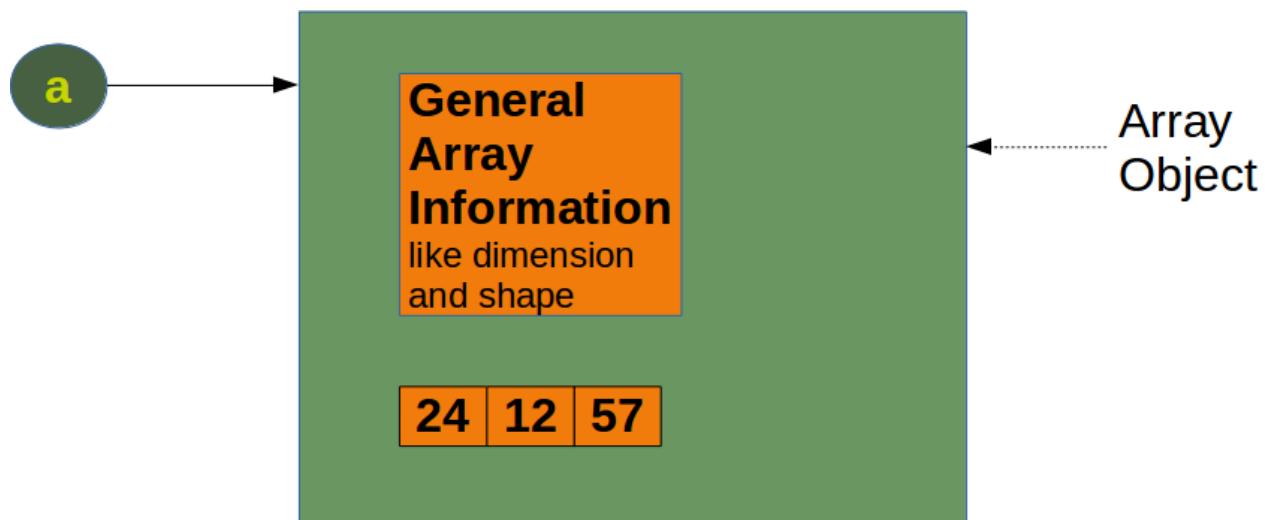


Figura 2: Representación en memoria de una lista

```
a = np.array(lst)
print(getsizeof(a))
```

```
120
```

Para analizar como se distribuye el consumo de memoria en un array vamos a calcular el tamaño de cada uno de los elementos como hicimos con las listas:

```
print('Un array vacío ocupa: {} bytes'.format(getsizeof(np.array([]))))
print('Un array con un elem: {} bytes'.format(getsizeof(np.array([24]))))
print('Un array con 2 elems: {} bytes'.format(getsizeof(np.array([24,12]))))
print('Un entero de Numpy es: {}'.format(type(a[0])))
```

```
Un array vacío ocupa: 96 bytes
Un array con un elem: 104 bytes
Un array con 2 elems: 112 bytes
Un entero de Numpy es: <class 'numpy.int64'>
```

Vemos que la información general sobre arrays ocupa **96 bytes** (en contraste a **64** para listas), y por cada elemento otros **8 bytes** adicionales (`numpy.int64` corresponde a 64 bits), por lo que el tamaño total será:

$$M_a(n) = 96 + n \times 8$$

```
from sys import getsizeof
lst1 = list(range(1000))
total_list_size = getsizeof(lst1) + sum(getsizeof(l) for l in lst1)
print("Tamaño total de la lista: ", total_list_size)
a1 = np.array(lst1)
print("Tamaño total de array: ", getsizeof(a1))
```

```
Tamaño total de la lista: 37116
Tamaño total de array: 8096
```

9.2.2 Velocidad de Numpy

Una de las grandes ventajas de usar *Numpy* está relacionada con la velocidad de cálculo. Veamos (superficialmente) esto

```
# %load scripts/timing.py
# Ejemplo del libro en www.python-course.eu/numpy.php

import numpy as np
from timeit import Timer
Ndim = 10000

def pure_python_version():
    X = range(Ndim)
    Y = range(Ndim)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return Z

def numpy_version():
    X = np.arange(Ndim)
    Y = np.arange(Ndim)
    Z = X + Y
    return Z

timer_obj1 = Timer("pure_python_version()", "from __main__ import pure_python_version")
timer_obj2 = Timer("numpy_version()", "from __main__ import numpy_version")
t1 = timer_obj1.timeit(10)
t2 = timer_obj2.timeit(10)

print("Numpy es en este ejemplo {:.3f} más rápido".format(t1 / t2))
```

```
Numpy es en este ejemplo 194.500 más rápido
```

Como vemos, utilizar *Numpy* puede ser considerablemente más rápido que usar *Python puro*.

9.3 Creación de arrays en Numpy

Un *array* en *numpy* es un tipo de variable parecido a una lista, pero está optimizado para realizar trabajo numérico.

Todos los elementos deben ser del mismo tipo, y además de los valores, contiene información sobre su tipo. Veamos algunos ejemplos de cómo crearlos y utilizarlos:

9.3.1 Creación de Arrays unidimensionales

```
i1 = np.array([1, 2, 3, 1, 5, 1, 9, 22, 0])
r1 = np.array([1.4 ,2.3 ,3.0 ,1, 5, 1, 9, 22, 0])
```

```
print(i1)
print(r1)
```

```
[ 1  2  3  1  5  1   9  22   0]
[ 1.4  2.3  3.   1.   5.   1.   9.   22.   0. ]
```

```
print('tipo de i1: {} \ntipo de r1: {}'.format(i1.dtype, r1.dtype))
```

```
tipo de i1: int64
tipo de r1: float64
```

```
print('Para i1:\n Número de dimensiones: {} \n Longitud: {}'.format(np.ndim(i1), len(i1)))
```

```
Para i1:
Número de dimensiones: 1
Longitud: 9
```

```
print('Para r1:\n Número de dimensiones: {} \n Longitud: {}'.format(np.ndim(r1), len(r1)))
```

```
Para r1:
Número de dimensiones: 1
Longitud: 9
```

9.3.2 Arrays multidimensionales

Podemos crear explícitamente *arrays* multidimensionales con la función `np.array` si el argumento es una lista anidada

```
L = [ [1, 2, 3], [.2, -.2, -1], [-1, 2, 9], [0, 0.5, 0] ]
```

```
A = np.array(L)
```

```
A
```

```
array([[ 1. ,  2. ,  3. ],
       [ 0.2, -0.2, -1. ],
       [-1. ,  2. ,  9. ],
       [ 0. ,  0.5,  0. ]])
```

```
print(A)
```

```
[[ 1.  2.  3. ]
 [ 0.2 -0.2 -1. ]
 [-1.  2.  9. ]
 [ 0.  0.5  0. ]]
```

```
print(np.ndim(A), A.ndim) # Ambos son equivalentes
```

```
2 2
```

```
print(len(A))
```

```
4
```

Vemos que la dimensión de `A` es 2, pero la longitud que me reporta **Python** corresponde al primer eje. Los *arrays* tienen un atributo que es la forma (`shape`)

```
print(A.shape)
```

```
(4, 3)
```

```
r1.shape # una tupla de un solo elemento
```

```
(9,)
```

9.3.3 Otras formas de creación

Hay otras maneras de crear **numpy arrays**. Algunas, de las más comunes es cuando necesitamos crear un array con todos ceros o unos o algún valor dado

```
a= np.zeros(5)
```

```
a.dtype # El tipo default es float de 64 bits
```

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

```
print(a)
```

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

```
i= np.zeros(5, dtype=int)
```

```
print(i)
```

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

```
i.dtype
```

```
dtype('int64')
```

```
c= np.zeros(5,dtype=complex)
print(c)
print(c.dtype)
```

```
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]  
complex128
```

En lugar de inicializarlo en cero podemos inicializarlo con algún valor

```
np.ones(5, dtype=complex) + 1j      # Algo similar pero inicializando a unos
```

```
array([1.+1.j, 1.+1.j, 1.+1.j, 1.+1.j, 1.+1.j])
```

Ya vimos que también podemos inicializarlos con valores equiespaciados con `np.arange()`, con `np.linspace()` o con `np.logspace()`

```
v = np.arange(2, 15, 2) # Crea un array con una secuencia (similar a la función range)
```

Para crear *arrays* multidimensionales usamos:

```
np.ones((4, 5))
```

```
array([[1., 1., 1., 1., 1.],  
       [1., 1., 1., 1., 1.],  
       [1., 1., 1., 1., 1.],  
       [1., 1., 1., 1., 1.]])
```

```
np.ones((4, 3, 6))
```

```
array([[[1., 1., 1., 1., 1., 1.],  
        [1., 1., 1., 1., 1., 1.],  
        [1., 1., 1., 1., 1., 1.]],  
  
       [[[1., 1., 1., 1., 1., 1.],  
         [1., 1., 1., 1., 1., 1.],  
         [1., 1., 1., 1., 1., 1.]],  
  
        [[[1., 1., 1., 1., 1., 1.],  
          [1., 1., 1., 1., 1., 1.],  
          [1., 1., 1., 1., 1., 1.]]],  
  
       [[[1., 1., 1., 1., 1., 1.],  
         [1., 1., 1., 1., 1., 1.],  
         [1., 1., 1., 1., 1., 1.]]])
```

```
np.eye(4)
```

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

```
np.eye(3, 7)
```

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

En este último ejemplo hemos creado matrices con unos en la diagonal y ceros en todos los demás lugares.

9.4 Acceso a los elementos

El acceso a los elementos tiene una forma muy parecida a la de las listas (pero no exactamente igual).

```
print(r1)
```

```
[ 1.4  2.3  3.  1.  5.  1.  9.  22.  0. ]
```

Si queremos uno de los elementos usamos la notación:

```
print(r1[0], r1[3], r1[-1])
```

```
1.4 1.0 0.0
```

y para tajadas (*slices*)

```
print(r1[:3])
```

```
[1.4 2.3 3. ]
```

```
print(r1[-3:])
```

```
[ 9. 22.  0.]
```

```
print(r1[5:7])
```

```
[1. 9.]
```

```
print(r1[0:8:2])
```

```
[1.4 3.  5.  9. ]
```

Como con vectores unidimensionales, con arrays multidimensionales, se puede ubicar un elemento o usar *slices*:

```
arr = np.arange(55).reshape((5,11))
```

```
arr
```

```
array([[ 0,  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, 32],
       [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43],
       [44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54]])
```

```
print("primer y segundo elementos", arr[0,0], arr[0,1])
```

```
primer y segundo elementos 0 1
```

```
print('Slicing parte de la segunda fila :', arr[1, 2:4])
print('Todas las filas, tercera columna :', arr[:, 2])
```

```
Slicing parte de la segunda fila : [13 14]
Todas las filas, tercera columna : [ 2 13 24 35 46]
```

```
print( 'Primera fila :\n', arr[0], '\nes igual a :\n', arr[0,:,:])
```

```
Primera fila :
[ 0  1  2  3  4  5  6  7  8  9 10]
es igual a :
[ 0  1  2  3  4  5  6  7  8  9 10]
```

```
print( 'Segunda fila :\n', arr[1], '\nes igual a :\n', arr[1,:,:])
```

```
Segunda fila :
[11 12 13 14 15 16 17 18 19 20 21]
es igual a :
[11 12 13 14 15 16 17 18 19 20 21]
```

```
print( 'Primera columna:', arr[:,0])
```

```
Primera columna: [ 0 11 22 33 44]
```

```
print( 'Última columna :\n', arr[:, -1])
```

```
Última columna :
[10 21 32 43 54]
```

```
print( 'Segunda fila, elementos impares (0,2,...) : ', arr[1,::2])
```

```
Segunda fila, elementos impares (0,2,...) : [11 13 15 17 19 21]
```

```
print( 'Segunda fila, todos los elementos impares : ', arr[1,1::2])
```

Cuando el *slicing* se hace de la forma `[i:f:s]` significa que tomaremos los elementos entre `i` (inicial), hasta `f` (final, no incluido), pero tomando sólo uno de cada `s` (stride) elementos

X[:, 1]	X[0, 9:]									
0	1	2	3	4	5	6	7	8	9	10
X[1, ::2]	11	12	13	14	15	16	17	18	19	20
	22	23	24	25	26	27	28	29	30	31
	33	34	35	36	37	38	39	40	41	42
	44	45	46	47	48	49	50	51	52	53
										54
X[2:, 3:5]	X[-1, -1]									

En Scipy Lectures at <http://scipy-lectures.github.io> hay una descripción del acceso a arrays.

9.5 Ejercicios 08 (a)

1. Genere arrays en 2d, cada uno de tamaño 10x10 con:
 - a. Un array con valores 1 en la diagonal principal y 0 en el resto (Matriz identidad).
 - b. Un array con valores 0 en la diagonal principal y 1 en el resto.
 - c. Un array con valores 1 en los bordes y 0 en el interior.
 - d. Un array con números enteros consecutivos (empezando en 1) en los bordes y 0 en el interior.
2. Diga qué resultado produce el siguiente código, y explíquelo

```
# Ejemplo propuesto por Jake VanderPlas
print(sum(range(5),-1))
from numpy import *
print(sum(range(5),-1))
```

9.6 Propiedades de Numpy arrays

9.6.1 Propiedades básicas

Hay tres propiedades básicas que caracterizan a un array:

- **shape**: Contiene información sobre la forma que tiene un array (sus dimensiones: vector, matriz, o tensor)
- **dtype**: Es el tipo de cada uno de sus elementos (todos son iguales)

- **stride:** Contiene la información sobre como recorrer el array. Por ejemplo si es una matriz, tiene la información de cuántos bytes en memoria hay que pasar para pasar de una fila a la siguiente y de una columna a la siguiente.

```
arr = np.arange(55).reshape((5,11))
```

```
print('shape :', arr.shape)
print('dtype :', arr.dtype)
print('strides:', arr.strides)
```

```
shape : (5, 11)
dtype : int64
strides: (88, 8)
```

```
print(np.arange(55).shape)
print(arr.shape)
```

```
(55,)
(5, 11)
```

9.6.2 Otras propiedades y métodos de los array

Los array tienen atributos que nos dan información sobre sus características:

```
print('Número total de elementos :', arr.size)
print('Número de dimensiones      :', arr.ndim)
print('Memoria usada              : {} bytes'.format(arr.nbytes))
```

```
Número total de elementos : 55
Número de dimensiones     : 2
Memoria usada              : 440 bytes
```

Además, tienen métodos que facilitan algunos cálculos comunes. Veamos algunos de ellos:

```
print('Mínimo y máximo           :', arr.min(), arr.max())
print('Suma y producto de sus elementos :', arr.sum(), arr.prod())
print('Media y desviación standard   :', arr.mean(), arr.std())
```

```
Mínimo y máximo           : 0 54
Suma y producto de sus elementos : 1485 0
Media y desviación standard   : 27.0 15.874507866387544
```

Para todos estos métodos, las operaciones se realizan sobre todos los elementos. En array multidimensionales uno puede elegir realizar los cálculos sólo sobre un dado eje:

```
print('Para el array:\n', arr)
```

```
Para el array:
[[ 0  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 32]
 [33 34 35 36 37 38 39 40 41 42 43]
 [44 45 46 47 48 49 50 51 52 53 54]]
```

```
print('La suma de todos los elementos es : ', arr.sum())
```

```
La suma de todos los elementos es : 1485
```

```
print('La suma de elementos de las filas es :', arr.sum(axis=1))
```

```
La suma de elementos de las filas es : [ 55 176 297 418 539]
```

```
print('La suma de elementos de las columnas es :', arr.sum(axis=0))
```

```
La suma de elementos de las columnas es : [110 115 120 125 130 135 140 145 150 155  
→160]
```

9.7 Operaciones sobre arrays

9.7.1 Operaciones básicas

Los array se pueden usar en operaciones:

```
# Suma de una constante
arr1 = 1 + arr[:,::-1] # Creamos un segundo array
```

```
arr1
```

```
array([[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1],
       [22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12],
       [33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23],
       [44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34],
       [55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45]])
```

```
# Multiplicación por constantes y suma de arrays
2* arr + 3*arr1
```

```
array([[ 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23],
       [ 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78],
       [143, 142, 141, 140, 139, 138, 137, 136, 135, 134, 133],
       [198, 197, 196, 195, 194, 193, 192, 191, 190, 189, 188],
       [253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243]])
```

```
# División por constantes
arr/5
```

```
array([[ 0., 0.2, 0.4, 0.6, 0.8, 1., 1.2, 1.4, 1.6, 1.8, 2. ],
       [ 2.2, 2.4, 2.6, 2.8, 3., 3.2, 3.4, 3.6, 3.8, 4., 4.2],
       [ 4.4, 4.6, 4.8, 5., 5.2, 5.4, 5.6, 5.8, 6., 6.2, 6.4],
       [ 6.6, 6.8, 7., 7.2, 7.4, 7.6, 7.8, 8., 8.2, 8.4, 8.6],
       [ 8.8, 9., 9.2, 9.4, 9.6, 9.8, 10., 10.2, 10.4, 10.6, 10.8]])
```

```
# Multiplicación entre arrays
arr * arr1
```

```
array([[ 0,  10,  18,  24,  28,  30,  30,  28,  24,  18,  10],
       [ 242,  252,  260,  266,  270,  272,  272,  270,  266,  260,  252],
       [ 726,  736,  744,  750,  754,  756,  756,  754,  750,  744,  736],
       [1452, 1462, 1470, 1476, 1480, 1482, 1482, 1480, 1476, 1470, 1462],
       [2420, 2430, 2438, 2444, 2448, 2450, 2450, 2448, 2444, 2438, 2430]])
```

```
arr / arr1
```

```
array([[ 0.          ,  0.1         ,  0.22222222,  0.375        ,
         0.57142857,  0.83333333,  1.2         ,  1.75        ,
         2.66666667,  4.5         ,  10.         ],
       [ 0.5          ,  0.57142857,  0.65        ,
         0.73684211,  0.83333333,  0.94117647,  1.0625      ,
         1.35714286,  1.53846154,  1.75         ],
       [ 0.66666667,  0.71875     ,  0.77419355,  0.83333333,  0.89655172,
         0.96428571,  1.03703704,  1.11538462,  1.2         ,
         1.29166667,  1.39130435],
       [ 0.75        ,  0.79069767,  0.83333333,  0.87804878,  0.925        ,
         0.97435897,  1.02631579,  1.08108108,  1.13888889,  1.2         ,
         1.26470588],
       [ 0.8          ,  0.83333333,  0.86792453,  0.90384615,  0.94117647,
         0.98        ,  1.02040816,  1.0625      ,  1.10638298,  1.15217391,
         1.2         ]])
```

Como vemos, están definidas todas las operaciones por constantes y entre arrays. En operaciones con constantes, se aplican sobre cada elemento del array. En operaciones entre arrays se realizan elemento a elemento (y el número de elementos de los dos array debe ser compatible).

9.7.2 Comparaciones

También se pueden comparar dos arrays elemento a elemento

```
v = np.linspace(0,19,20)
w = np.linspace(0.5,18,20)
```

```
print (v)
print (w)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.
 18. 19.]
[ 0.5          1.42105263  2.34210526  3.26315789  4.18421053  5.10526316
 6.02631579  6.94736842  7.86842105  8.78947368  9.71052632 10.63157895
11.55263158 12.47368421 13.39473684 14.31578947 15.23684211 16.15789474
17.07894737 18.          ]
```

```
# Comparación de un array con una constante
print(v > 12)
```

```
[False False False False False False False False False
 False True  True  True  True  True  True]
```

```
# Comparación de un array con una constante
print(v > w)
```

```
[False False False False False False True True True True True
 True True True True True True True]
```

9.7.3 Funciones definidas en Numpy

Algunas de las funciones definidas en numpy se aplican a cada elemento. Por ejemplo, las funciones matemáticas:

```
np.sin(arr1)
```

```
array([[-0.99999021, -0.54402111,  0.41211849,  0.98935825,  0.6569866 ,
       -0.2794155 , -0.95892427, -0.7568025 ,  0.14112001,  0.90929743,
       0.84147098],
      [-0.00885131,  0.83665564,  0.91294525,  0.14987721, -0.75098725,
       -0.96139749, -0.28790332,  0.65028784,  0.99060736,  0.42016704,
       -0.53657292],
      [ 0.99991186,  0.55142668, -0.40403765, -0.98803162, -0.66363388,
       0.27090579,  0.95637593,  0.76255845, -0.13235175, -0.90557836,
       -0.8462204 ],
      [ 0.01770193, -0.83177474, -0.91652155, -0.15862267,  0.74511316,
       0.96379539,  0.29636858, -0.64353813, -0.99177885, -0.42818267,
       0.52908269],
      [-0.99975517, -0.55878905,  0.39592515,  0.98662759,  0.67022918,
       -0.26237485, -0.95375265, -0.76825466,  0.12357312,  0.90178835,
       0.85090352]])
```

```
np.exp(-arr**2/2)
```

```
array([[1.00000000e+000,  6.06530660e-001,  1.35335283e-001,
       1.11089965e-002,  3.35462628e-004,  3.72665317e-006,
       1.52299797e-008,  2.28973485e-011,  1.26641655e-014,
       2.57675711e-018,  1.92874985e-022],
      [5.31109225e-027,  5.38018616e-032,  2.00500878e-037,
       2.74878501e-043,  1.38634329e-049,  2.57220937e-056,
       1.75568810e-063,  4.40853133e-071,  4.07235863e-079,
       1.38389653e-087,  1.73008221e-096],
      [7.95674389e-106,  1.34619985e-115,  8.37894253e-126,
       1.91855567e-136,  1.61608841e-147,  5.00796571e-159,
       5.70904011e-171,  2.39425476e-183,  3.69388307e-196,
       2.09653176e-209,  4.37749104e-223],
      [3.362440407e-237,  9.50144065e-252,  9.87710872e-267,
       3.77724997e-282,  5.31406836e-298,  2.75032531e-314,
       0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
       0.00000000e+000,  0.00000000e+000],
      [0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
       0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
       0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
       0.00000000e+000,  0.00000000e+000]])
```

Podemos elegir elementos de un array basados en condiciones:

```
v[w > 9]
```

```
array([10., 11., 12., 13., 14., 15., 16., 17., 18., 19.])
```

Aquí eligiendo elementos de v basados en una condición sobre los elementos de w. Veamos en más detalle:

```
c = w > 9 # Array con condición
print(c)
x = v[c]    # Creamos un nuevo array con los valores donde c es verdadero
print(x)
print(len(c), len(v), len(x))
```

```
[False False False False False False False False False True True
 True True True True True True True]
[10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
20 20 10
```

También podemos hacer todo tipo de operaciones (suma, resta, multiplicación,.) entre *arrays*

9.8 Ejercicios 08 (b)

3. Escriba una función `suma_potencias(p, n)` (utilizando arrays y **Numpy**) que calcule la operación

$$s_2 = \sum_{k=0}^n k^p$$

4. Usando las funciones de numpy `sign` y `maximum` definir las siguientes funciones, que acepten como argumento un array y devuelvan un array con el mismo *shape*:
- función de Heaviside, que vale 1 para valores positivos de su argumento y 0 para valores negativos.
 - La función escalón, que vale 0 para valores del argumento fuera del intervalo $(-1, 1)$ y 1 para argumentos en el intervalo.
 - La función rampa, que vale 0 para valores negativos de x y x para valores positivos.

9.8.1 Lectura y escritura de datos a archivos

Numpy tiene funciones que permiten escribir y leer datos de varias maneras, tanto en formato *texto* como en *binario*. En general el modo *texto* ocupa más espacio pero puede ser leído y modificado con un editor.

Datos en formato texto

```
np.savetxt('test.out', arr, fmt='%.2e', header="x      y      \n z      z2", comments="#")
!cat test.out
```

```
arr2 = np.loadtxt('test.out', comments="#")
print(arr2)
```

```
print(arr2.shape)
print(arr2.ndim)
print(arr2.size)
```

9.9 Ejercicios 08 (c)

5. **PARA ENTREGAR. Caída libre** Cree un programa que calcule la posición y velocidad de una partícula en caída libre para condiciones iniciales dadas (h_0, v_0), y un valor de gravedad dados. Se utilizará la convención de que alturas y velocidades positivas corresponden a vectores apuntando hacia arriba (una velocidad positiva significa que la partícula se aleja de la tierra).

El programa debe realizar el cálculo de la velocidad y altura para un conjunto de tiempos equiespaciados. El usuario debe poder decidir o modificar el comportamiento del programa mediante opciones por línea de comandos.

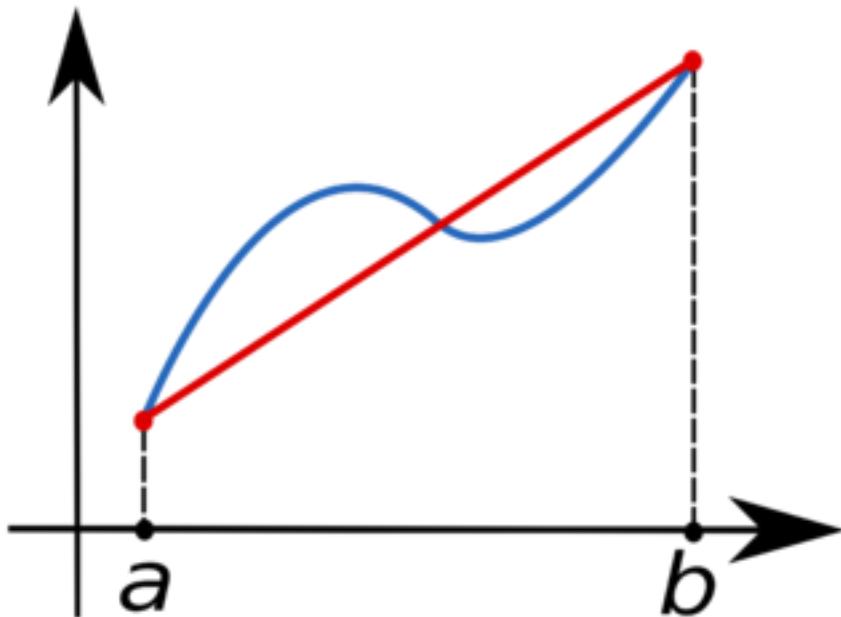
El programa debe aceptar las siguientes opciones por líneas de comando: `-v vel 0`, equivalentemente `--velocidad=vel`, donde `vel` es el número dando la velocidad inicial en m/s. El valor por defecto será 0. `-h alt 0`, equivalentemente `--altura=alt`, donde `alt` es un número dando la altura inicial en metros. El valor por defecto será 1000. La altura inicial debe ser un número positivo. `-g grav`, donde `grav` es el módulo del valor de la aceleración de la gravedad en m/s^2 . El valor por defecto será 9.8. `-o nombre 0`, equivalentemente `--output=nombre`, donde `nombre` será el nombre de un archivo donde se escribirán los resultados. Si el usuario no usa esta opción, debe imprimir por pantalla (`sys.stdout`). `-n N 0`, equivalentemente `--Ndatos=N`, donde `N` es un número entero indicando la cantidad de datos que deben calcularse. Valor por defecto: 100. `--ti=instante_inicial` indica el tiempo inicial de cálculo. Valor por defecto: 0. No puede ser mayor que el tiempo de llegada a la posición $h = 0$. `--tf=tiempo_final` indica el tiempo inicial de cálculo. Valor por defecto será el correspondiente al tiempo de llegada a la posición $h = 0$.

NOTA: Envíe el programa llamado **Suapellido_08.py** en un adjunto por correo electrónico, con asunto: **Suapellido_08**, antes del día lunes 9 de Marzo.

6. Queremos realizar numéricamente la integral

$$\int_a^b f(x)dx$$

utilizando el método de los trapecios. Para eso partimos el intervalo $[a, b]$ en N subintervalos y aproximamos la curva en cada subintervalo por una recta



La línea azul representa la función $f(x)$ y la línea roja la interpolación por una recta (figura de https://en.wikipedia.org/wiki/Trapezoidal_rule)

Si llamamos x_i ($i = 0, \dots, n$, con $x_0 = a$ y $x_n = b$) los puntos equiespaciados, entonces queda

$$\int_a^b f(x)dx \approx \frac{h}{2} \sum_{i=1}^n (f(x_i) + f(x_{i-1})).$$

- Escriba una función `trapz(x, y)` que reciba dos arrays unidimensionales `x` e `y` y aplique la fórmula de los trapecios.
- Escriba una función `trapzf(f, a, b, npts=100)` que recibe una función `f`, los límites `a`, `b` y el número de puntos a utilizar `npts`, y devuelve el valor de la integral por trapecios.
- Calcule la integral logarítmica de Euler:

$$\text{Li}(t) = \int_2^t \frac{1}{\ln x} dx$$

usando la función `'trapzft'` para valores de `npts=10, 20, 30, 40, 50, 60`

CAPÍTULO 10

Clase 9: Visualización

Para graficar datos y funciones vamos a usar la biblioteca **Matplotlib**. Vamos a empezar discutiendo algunas de las características más generales del trabajo con esta biblioteca y mostrar algún ejemplo relativamente sencillo. Matplotlib está dividido en tres partes o capas conceptualmente bien delimitadas:

- Una parte es la que hace el trabajo más pesado administrando cada parte del gráfico (líneas, texto, figuras, etc)
- Una segunda parte que permite un uso simple de las funciones anteriores: El módulo **pyplot**.
- Una tercera parte que se encarga de presentar la figura en el formato adecuado. Esto es lo que se llama el *Backend* y se encarga de mostrar la figura en los distintos sistemas de ventanas, o en formatos de archivos correspondientes. Algunos ejemplos de *backend* son: PS (copias PostScript), SVG (Scalable Vector Graphics), Agg (salida PNG de muy buena calidad), Cairo (png, pdf, ps, svg), GTK (interactivo, permite integrar matplotlib con aplicaciones Gtk+, que usa GNOME), PDF, WxWidgets (interactivo), Qt (interactivo).

Nosotros vamos a concentrarnos principalmente en aprender a utilizar **pyplot**

10.1 Interactividad

10.1.1 Trabajo con ventanas emergentes

Para trabajar en forma interactiva con gráficos vamos a hacerlo desde una terminal de `Ipython`

```
import matplotlib.pyplot as plt # o equivalentemente:  
# from matplotlib import pyplot as plt  
plt.plot([0,1,4,9,16,25])
```

El comando (la función) `plot()` crea el gráfico pero no lo muestra. Lo hacemos explícitamente con el comando `show()`

```
plt.show()
```

Vemos que es muy simple graficar datos.

Algunas cosas a notar:

1. Se abre una ventana
2. Se bloquea la terminal (no podemos dar nuevos comandos)
3. Si pasamos el *mouse* sobre el gráfico nos muestra las coordenadas.
4. Además del gráfico hay una barra de herramientas: .. image:: figuras/matplotlib_toolbar.png

De derecha a izquierda tenemos:

- **Grabar:** Este botón abre una ventana para guardar el gráfico en alguno de los formatos disponibles.
- **Configuración de subplots:** Permite modificar el tamaño y posición de cada gráfico en la ventana.
- **Agrandar (zoom) a rectángulo:**
 - Si elegimos una región mientras apretamos el botón **izquierdo**, esta será la nueva región visible ocupando toda la ventana.
 - Si elegimos una región mientras apretamos el botón **derecho**, pone toda la vista actual en esta región.
- **Desplazar y agrandar (Pan and zoom):** Este botón cumple dos funciones diferentes:
 - Con el botón izquierdo desplaza la vista.
 - Con el botón derecho la vista se agrandará achicará en los ejes horizontal y vertical en una cantidad proporcional al movimiento.

Si se oprime las teclas x o y las dos funciones quedan restringidas al eje correspondiente.

- **Home, Back, Forward** son botones que nos llevan a la vista original, una vista hacia atrás o hacia adelante respectivamente

Si ocurre, como en este caso, que proveemos sólo una lista de datos, la función `plot()` la interpreta como los valores correspondientes al eje vertical (eje y), y toma el índice del dato para la variable independiente (eje x). Si queremos dar valores explícitamente para el eje x debemos pasarlos como primer argumento.

```
plt.plot([0,1,2,3,4,5],[0,1,4,9,16,25])
plt.show()
```

Como vemos, para que muestre la ventana debemos hacer un llamado explícito a la función `show()`. Esto es así porque muchas veces queremos realizar más de una operación sobre un gráfico antes de mostrarlo. Sin embargo, hay varias alternativas respecto a la interactividad de matplotlib (e ipython) que permiten adecuarla a nuestro flujo de trabajo. La manera más común en una terminal es utilizando la función `ion()` (`interactive on`) para hacerlo interactivo y la función `ioff()` para no interactivo.

```
plt.ion()          # Prendemos el modo interactivo
plt.plot([0,1,2,3,4,5],[0,1,4,9,16,25])
```

En el modo interactivo no sólo `plot()` tiene implícito el comando `show()` sino que podemos seguir ingresando comandos con el gráfico abierto.

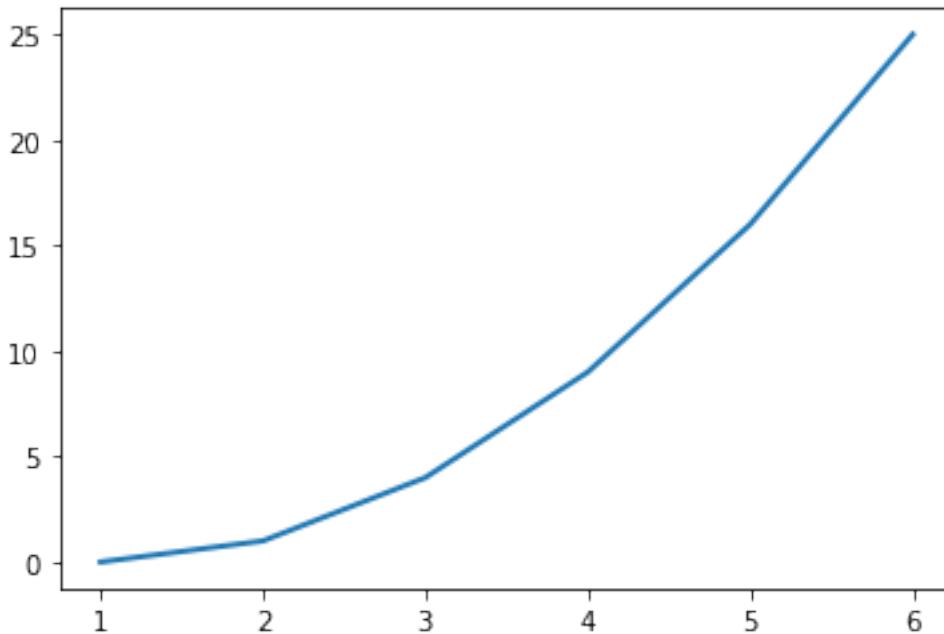
10.1.2 Trabajo sobre notebooks

Para trabajar en *ipython notebooks* suele ser conveniente realizar los gráficos dentro de la misma página donde realizamos los cálculos. Si esto no ocurre automáticamente, se puede obtener con la siguiente línea:

```
%matplotlib inline
```

```
import matplotlib.pyplot as plt
# %matplotlib inline
plt.plot([1,2,3,4,5,6],[0,1,4,9,16,25])
```

```
[<matplotlib.lines.Line2D at 0x7fa9f8f9ef90>]
```



En la práctica vamos a usar siempre **Matplotlib** junto con **Numpy**.

```
import numpy as np
import matplotlib.pyplot as plt
```

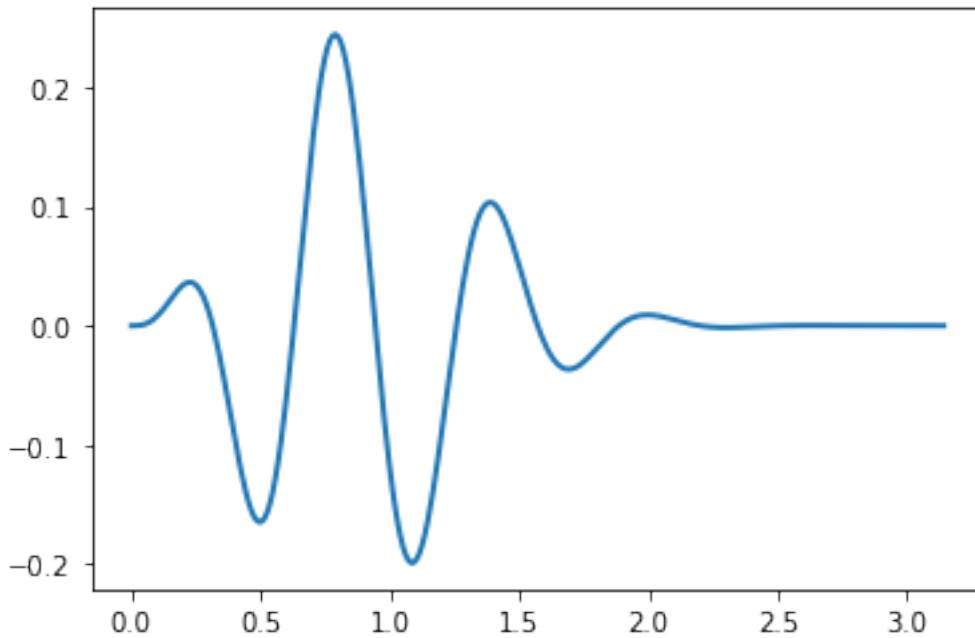
10.2 Gráficos simples

El paquete *Matplotlib* permite visualizar datos guardados en un archivo con unas pocas líneas

```
fdatos = '../data/ej_oscil_aten_err.dat'
```

```
x, y, yexp = np.loadtxt(fdatos, unpack=True)
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f8f1a790>]
```



Como vemos, es la curva que graficamos la clase anterior.

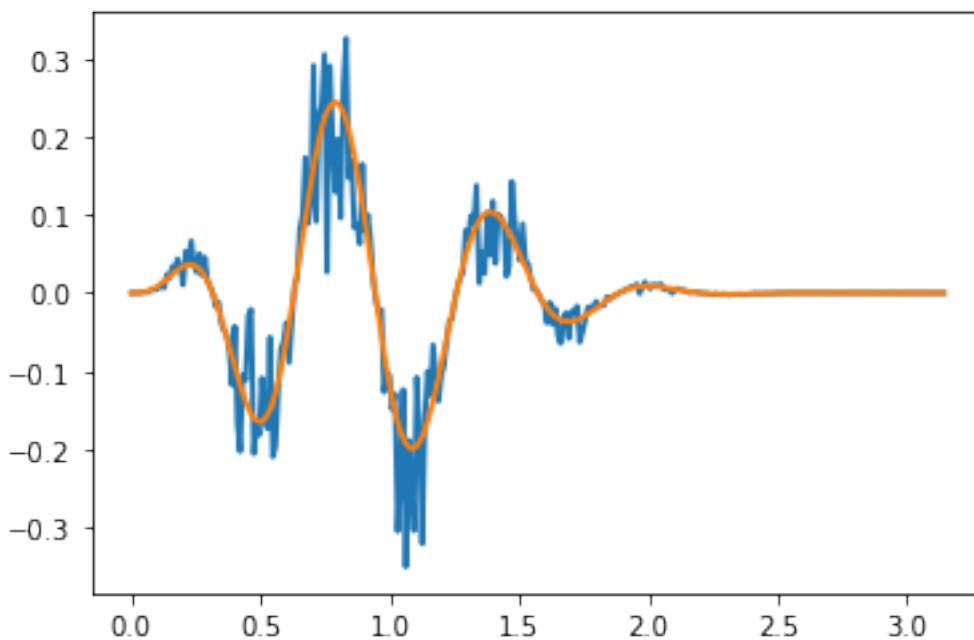
Si miramos el archivo vamos a ver que tiene una columna más que corresponde a los valores medidos. En consecuencia le asignamos esta tercera columna a una variable adicional `yexp` al leerlo.

```
!head ../data/ej_oscil_aten_err.dat
```

```
#      x          teo          exp
0.0000000e+00 0.0000000e+00 0.0000000e+00
1.0507000e-02 1.1576170e-05 1.4544540e-05
2.1014000e-02 9.2052870e-05 7.5934893e-05
3.1521000e-02 3.0756500e-04 1.8990066e-04
4.2028000e-02 7.1879320e-04 6.1217896e-04
5.2534990e-02 1.3784280e-03 1.2133173e-03
6.3041990e-02 2.3288570e-03 9.5734774e-04
7.3548990e-02 3.6001450e-03 3.5780825e-03
8.4055990e-02 5.2083560e-03 4.4485492e-03
```

```
# Graficamos las segunda y tercera columna como función de la primera
plt.plot(x,yexp, x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f8f07850>,
 <matplotlib.lines.Line2D at 0x7fa9f8f07ad0>]
```



10.3 Formato de las curvas

En los gráficos anteriores usamos la función `plot()` en sus formas más simples.

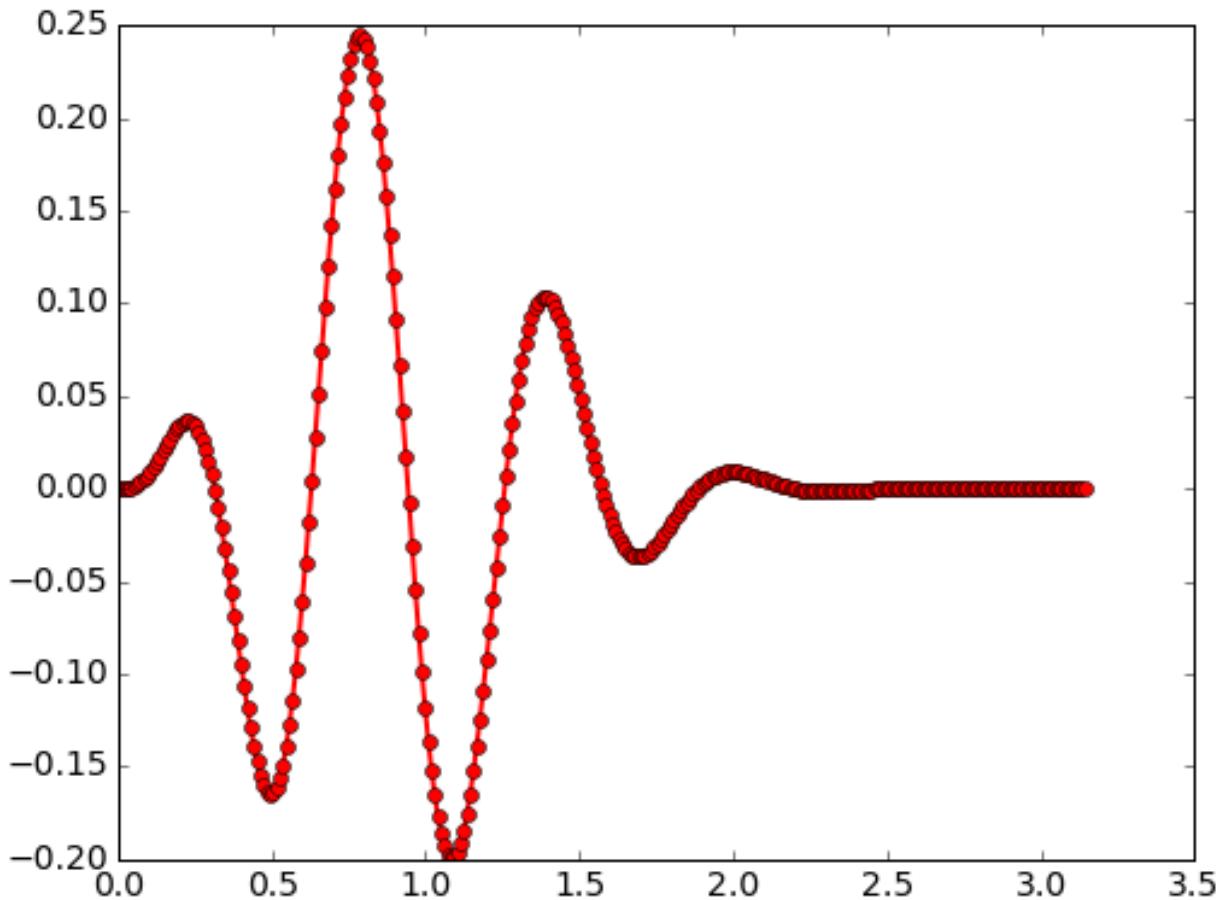
```
plot(y)
plot(x,y)
plot(x1,y1, x2, y2)
```

dando a la función `plot()` la información mínima necesaria para graficar los datos. Usualmente queremos poder elegir cómo vamos a graficar nuestros datos.

10.3.1 Líneas, símbolos y colores

La forma más simple de elegir el modo de graficación de la curva es a través de un tercer argumento. Este argumento, que aparece inmediatamente después de los datos (`x` e `y`), permite controlar el tipo de línea o símbolo utilizado en la graficación. En el caso de la línea sólida se puede especificar con un guión (-) o simplemente no poner nada, ya que línea sólida es el símbolo por defecto. Las dos especificaciones anteriores son equivalentes. También se puede elegir el color, o el símbolo a utilizar con este argumento:

```
plot(x,y, 'g-')
plot(x,y, 'ro')
plot(x,y, 'r-o')
```

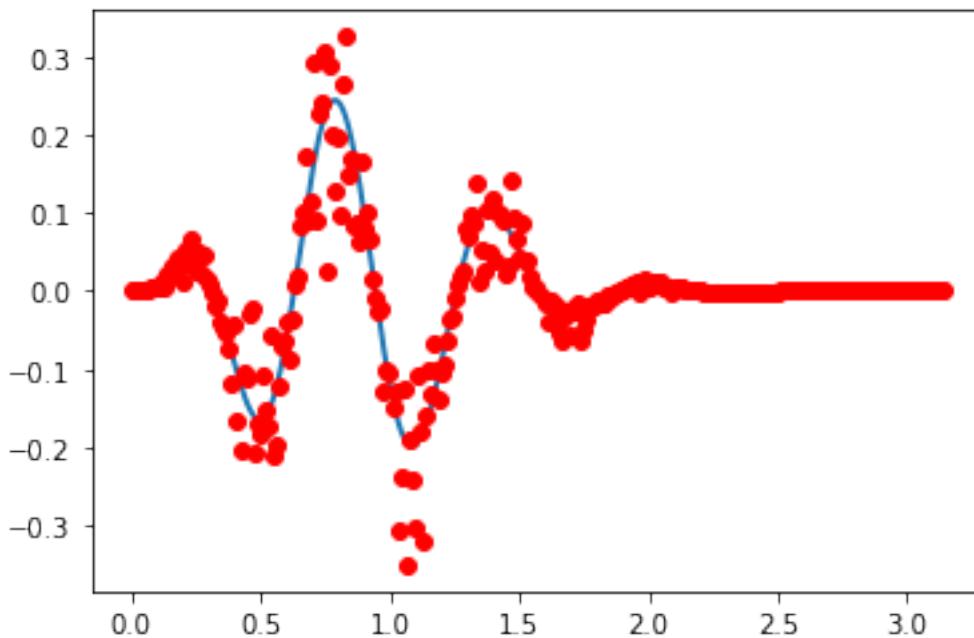


Para obtener círculos usamos una especificación que corresponde a o. Además podemos poner en este argumento el color. En este caso elegimos graficar en color rojo (r), con una línea (-) + círculos (o).

Con esta idea modificamos el gráfico anterior

```
plt.plot(x,y,'-o', x,yexp, 'ro')
```

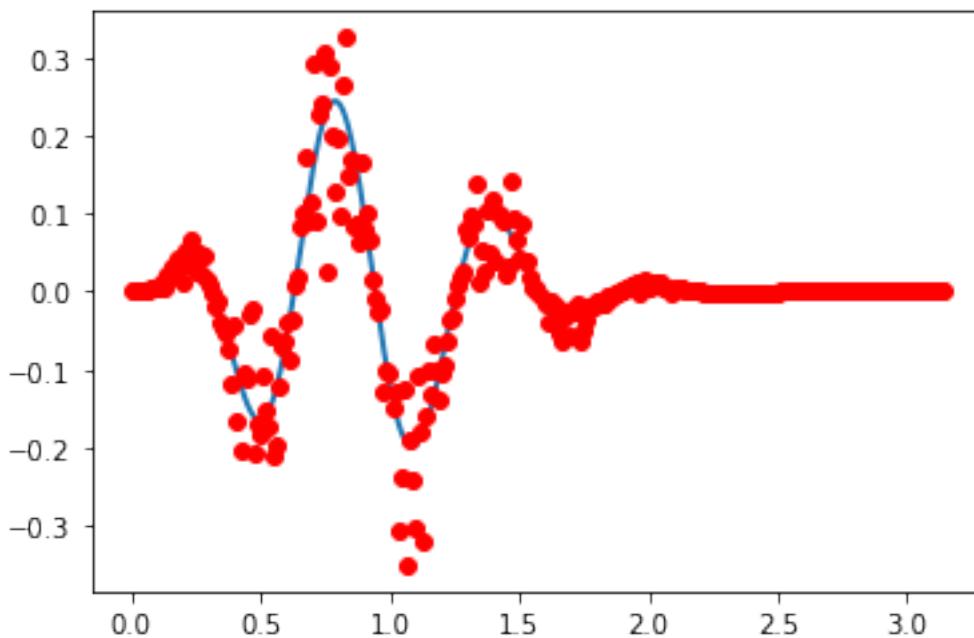
```
[<matplotlib.lines.Line2D at 0x7fa9f8857d90>,
 <matplotlib.lines.Line2D at 0x7fa9f8857fd0>]
```



Para graficar más de una curva, en este formato simple, podemos ponerlo todo en la misma función `plot()` en la forma `plot(x1, y1, [formato], x2, y2, [formato2])` pero muchas veces es más legible separar los llamados a la función, una para cada curva.

```
plt.plot(x,y, '-')
plt.plot(x,yexp, 'or')
```

```
[<matplotlib.lines.Line2D at 0x7fa9f8805a90>]
```



Al separar los llamados a la función `plot()` obtenemos un código más claro, principalmente cuando agregamos opciones de formato.

Los siguientes caracteres pueden utilizarse para controlar el símbolo de graficación:

Símbolo	Descripción
-	solid line style
--	dashed line style
-.	dash-dot line style
:	dotted line style
.	point marker
,	pixel marker
o	circle marker
v	triangle down marker
^	triangle up marker
<	triangle left marker
>	triangle right marker
1	tri down marker
2	tri up marker
3	tri left marker
4	tri right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin diamond marker
	vline marker
_	hline marker

Los colores también pueden elegirse usando los siguientes caracteres:

Letra	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Por ejemplo, utilizando:

```
plt.plot(x, y1, 'gs', x, y2, '-k^', x, y3, '--r' )
```

obtenemos: .. image:: figuras/simple_varios.png

La función `plot()` acepta un número variable de argumentos. Veamos lo que dice la documentación

```
Signature: plt.plot(*args, **kwargs)
Docstring:
```

(continué en la próxima página)

(provien de la página anterior)

```
Plot y versus x as lines and/or markers.
```

Call signatures::

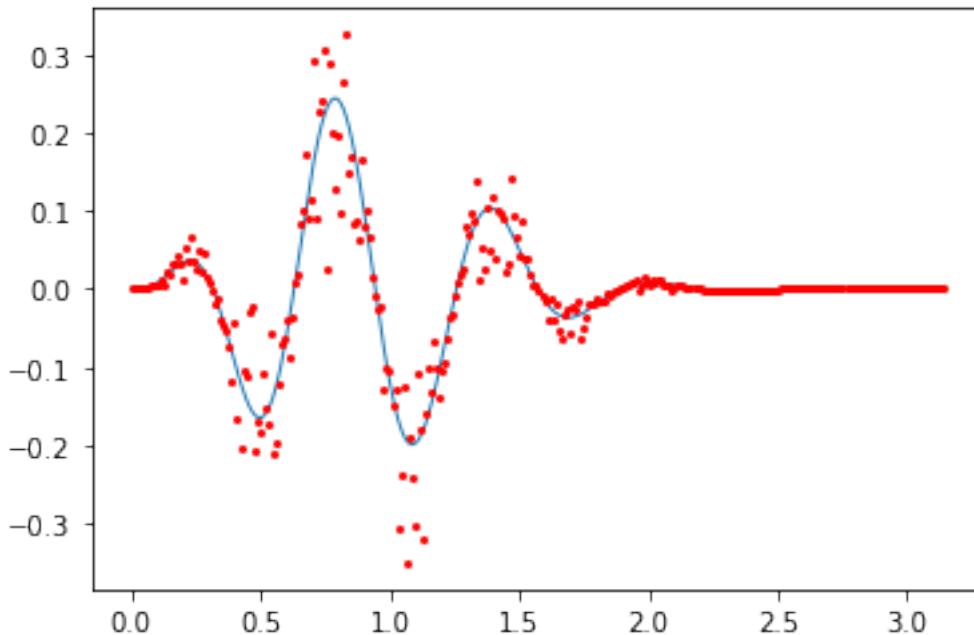
```
plot([x], y, [fmt], data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

En particular, podemos usar los argumentos *keywords* (pares nombre-valor) para cambiar el modo en que se grafican los datos. Algunos de los más comunes son:

Argumento	Valor
linestyle	{-, -, -., :, , }
linewidth	número real
color	un color
marker	{o, s, d, .}
markersize	número real
markeredgecolor	color
markerfacecolor	color
markevery	número entero

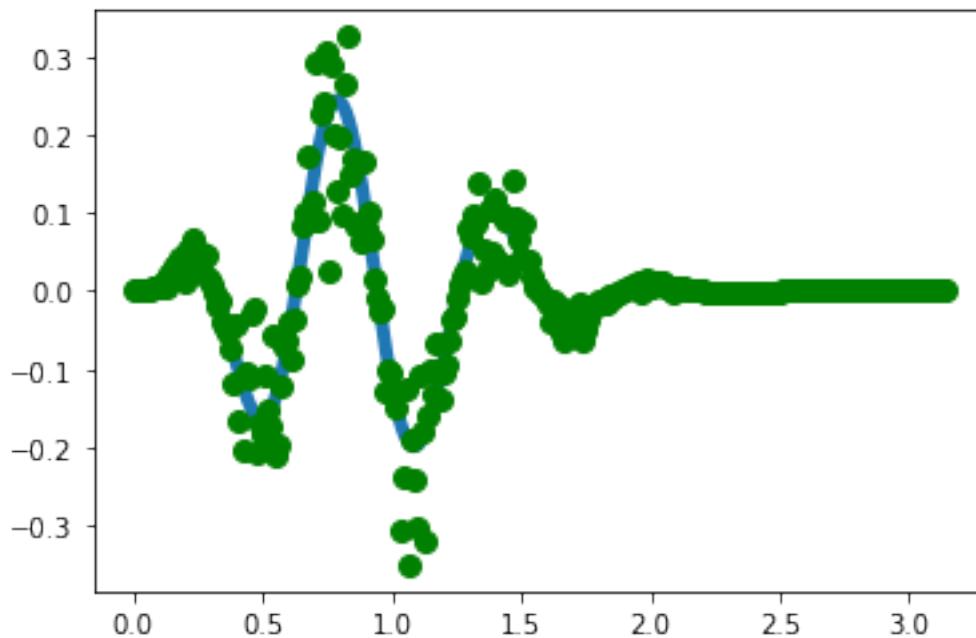
```
plt.plot(x,y,linewidth=1)
plt.plot(x,yexp, 'o', color='red', markersize=2)
```

[<matplotlib.lines.Line2D at 0x7fa9f86fba50>]



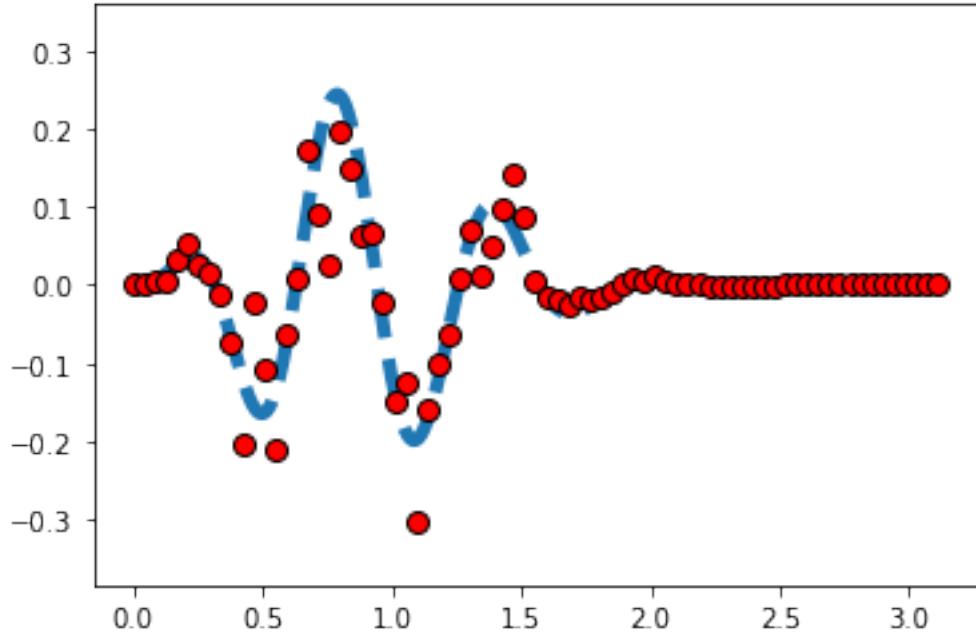
```
plt.plot(x,y,linewidth=5)
plt.plot(x,yexp, 'o', color='green', markersize=8)
```

[<matplotlib.lines.Line2D at 0x7fa9f86ea410>]



```
plt.plot(x,y,linewidth=5, linestyle='dashed')
plt.plot(x,yexp, 'o', color='red', markersize=8, markeredgecolor='black',markevery=4)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f85c9990>]
```



10.3.2 Nombres de ejes y leyendas

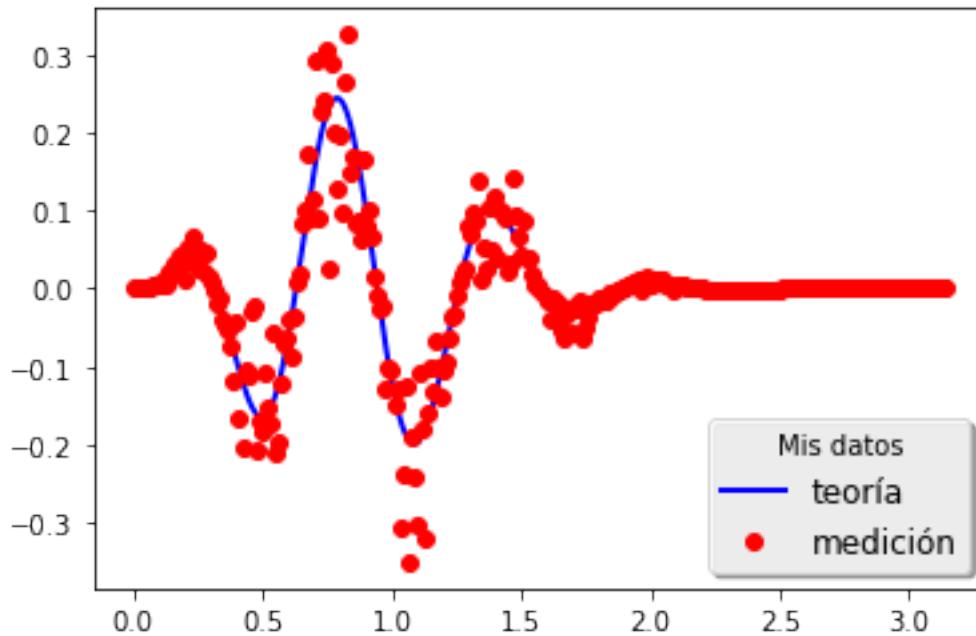
Vamos ahora a agregar nombres a los ejes y a las curvas.

Para agregar nombres a las curvas, tenemos que agregar un `label`, en este caso en el mismo comando `plot()`, y

luego mostrarlo con ‘legend()

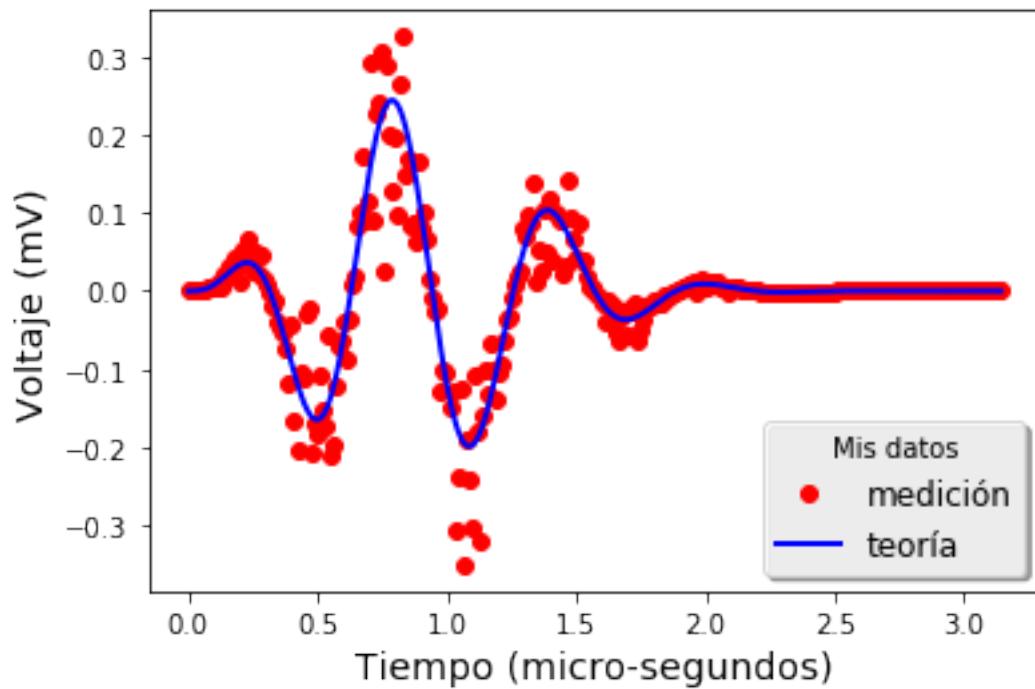
```
plt.plot(x,y, '-b', label="teoría")
plt.plot(x,yexp, 'or', label="medición")
plt.legend(loc="lower right", title="Mis datos")
```

```
<matplotlib.legend.Legend at 0x7fa9f8559c50>
```



Para agregar nombres a los ejes usamos xlabel y ylabel:

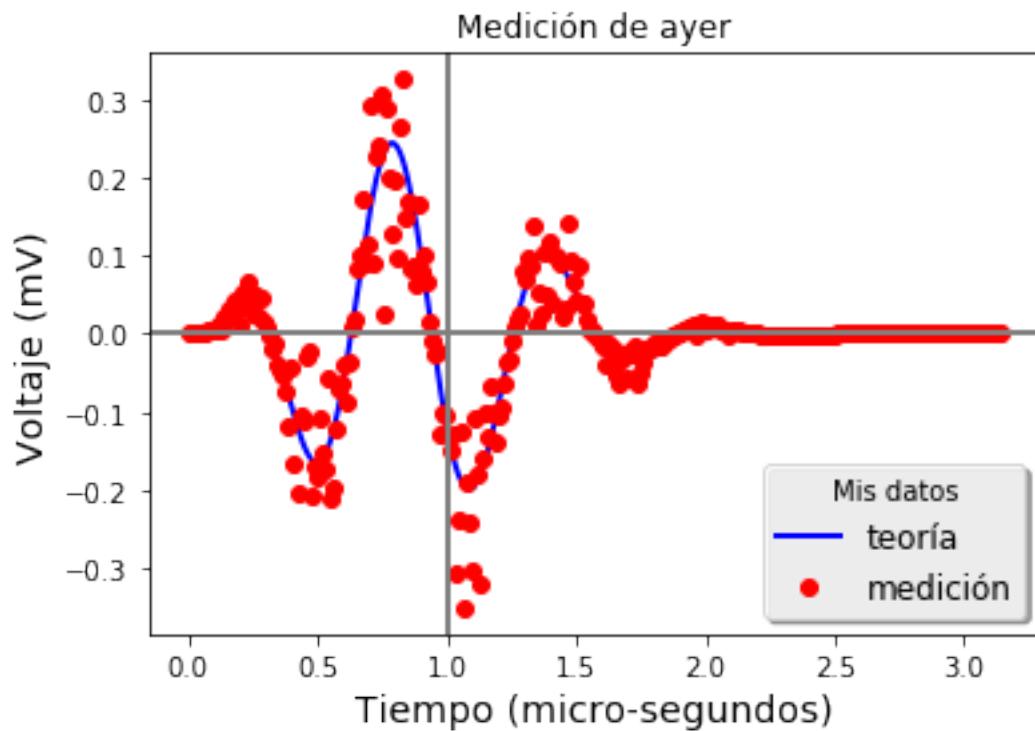
```
plt.plot(x,yexp, 'or', label="medición")
plt.plot(x,y, '-b', label="teoría")
plt.legend(loc="lower right", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)");
```



Los títulos a la figura se pueden agregar con title

```
plt.plot(x,y, '-b', label="teoría")
plt.plot(x,yexp, 'or', label="medición")
plt.legend(loc="lower right", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)")
plt.title("Medición de ayer")
plt.axvline(x=1, color='gray')
plt.axhline(color='gray')
```

```
<matplotlib.lines.Line2D at 0x7fa9f83d68d0>
```



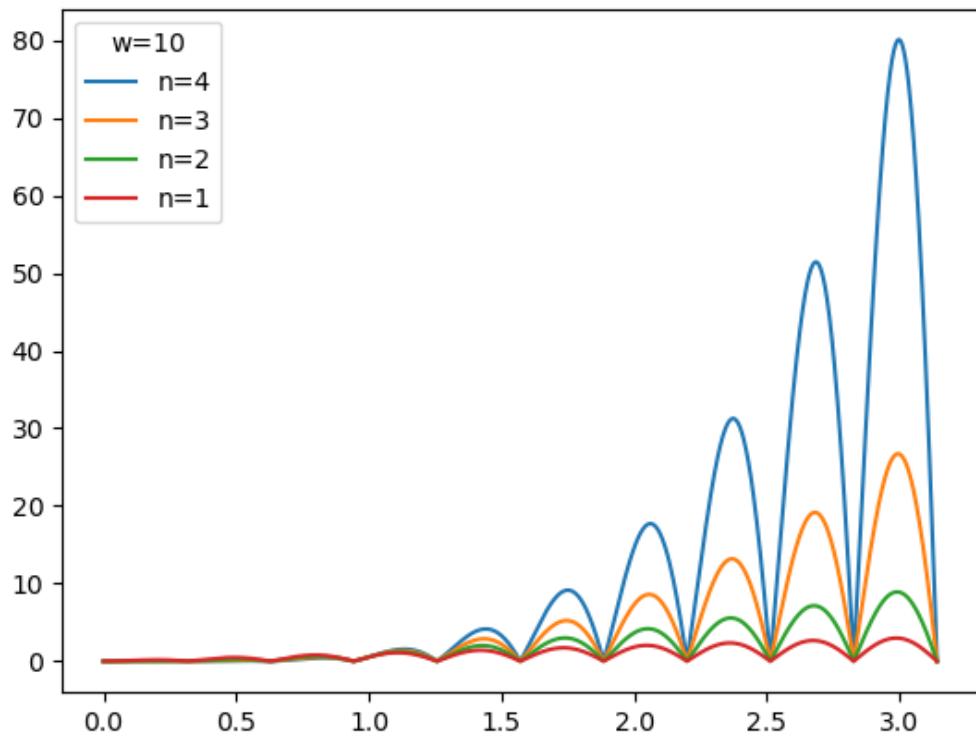
Acá además agregamos una línea vertical y una horizontal.

10.4 Ejercicios 09 (a)

1. Realizar un programa para visualizar la función

$$f(x, n, w) = x^n |\sin(wx)|$$

El programa debe realizar el gráfico para $w = 10$, con cuatro curvas para $n = 1, 2, 3, 4$, similar al que se muestra en la siguiente figura

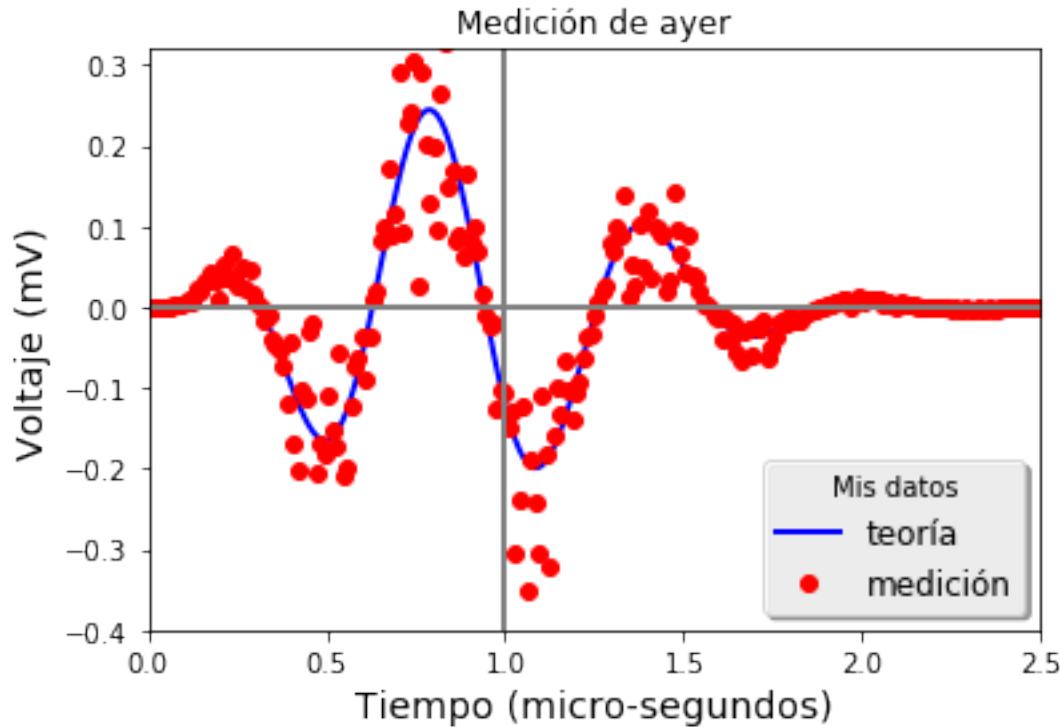


10.5 Escalas y límites de graficación (vista)

Para cambiar los límites de graficación se puede usar las funciones `xlim` para el eje horizontal y `ylim` para el vertical

```
plt.plot(x,y, '-b', label="teoría")
plt.plot(x,yexp, 'or', label="medición")
plt.legend(loc="lower right", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)")
plt.title("Medición de ayer")
plt.axvline(x=1, color='gray')
plt.axhline(color='gray')
plt.xlim((0,2.5))
plt.ylim((-0.4, 0.32))
```

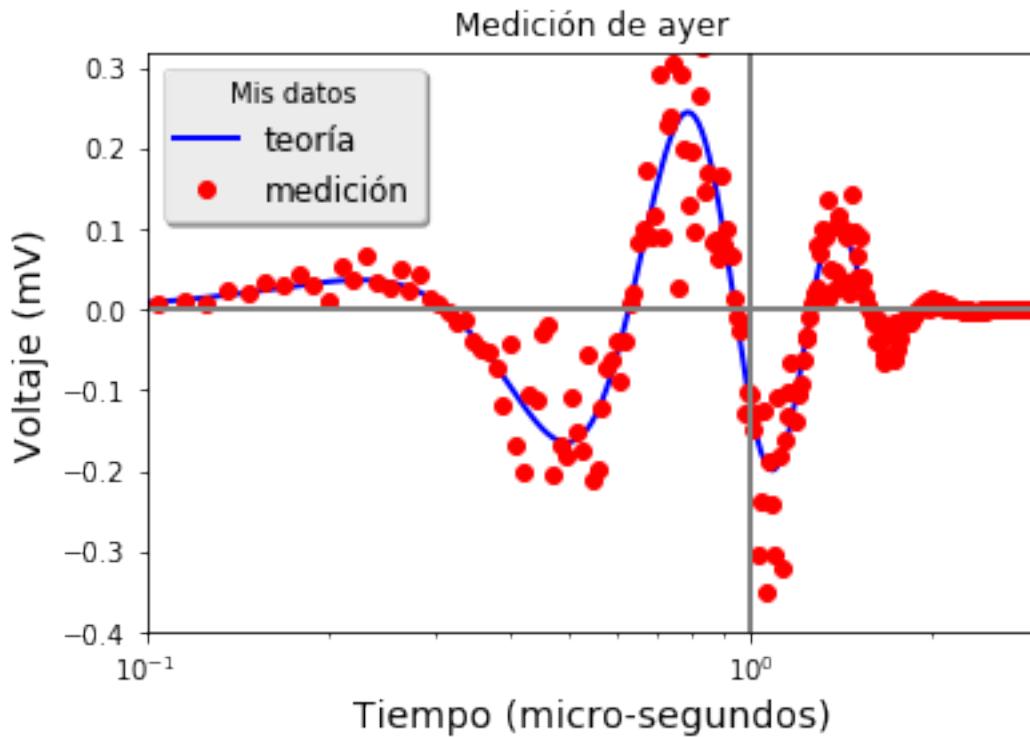
```
(-0.4, 0.32)
```



Para pasar a escala logarítmica usamos `xscale` o `yscale`

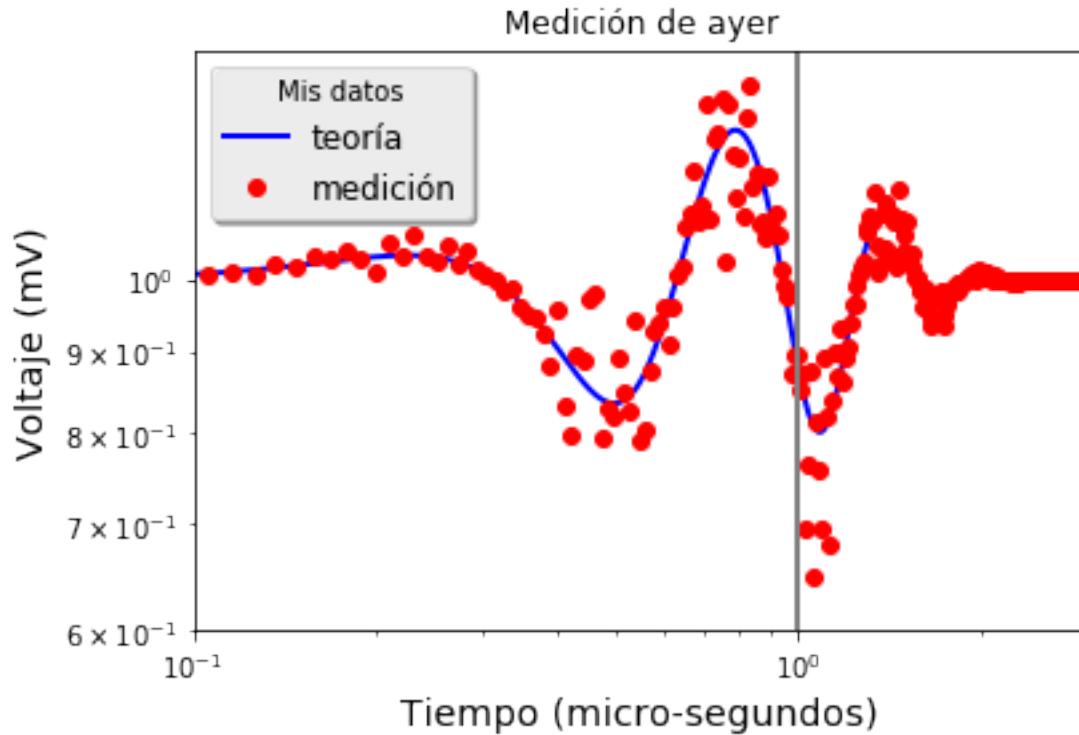
```
plt.plot(x,y, '-b', label="teoría")
plt.plot(x,yexp, 'or', label="medición")
plt.legend(loc="best", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)")
plt.title("Medición de ayer")
plt.axvline(x=1, color='gray')
plt.axhline(color='gray')
plt.xlim((1.e-1,3))
plt.xscale('log')
plt.ylim((-0.4, 0.32))
```

(-0.4, 0.32)



```
plt.plot(x, 1+y, '-b', label="teoría")
plt.plot(x, 1+yexp, 'or', label="medición")
plt.legend(loc="best", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)")
plt.title("Medición de ayer")
plt.axvline(x=1, color='gray')
plt.axhline(color='gray')
plt.xlim((1.e-1,3))
plt.xscale('log')
plt.yscale('log')
plt.ylim((0.6, 1.4))
```

```
(0.6, 1.4)
```



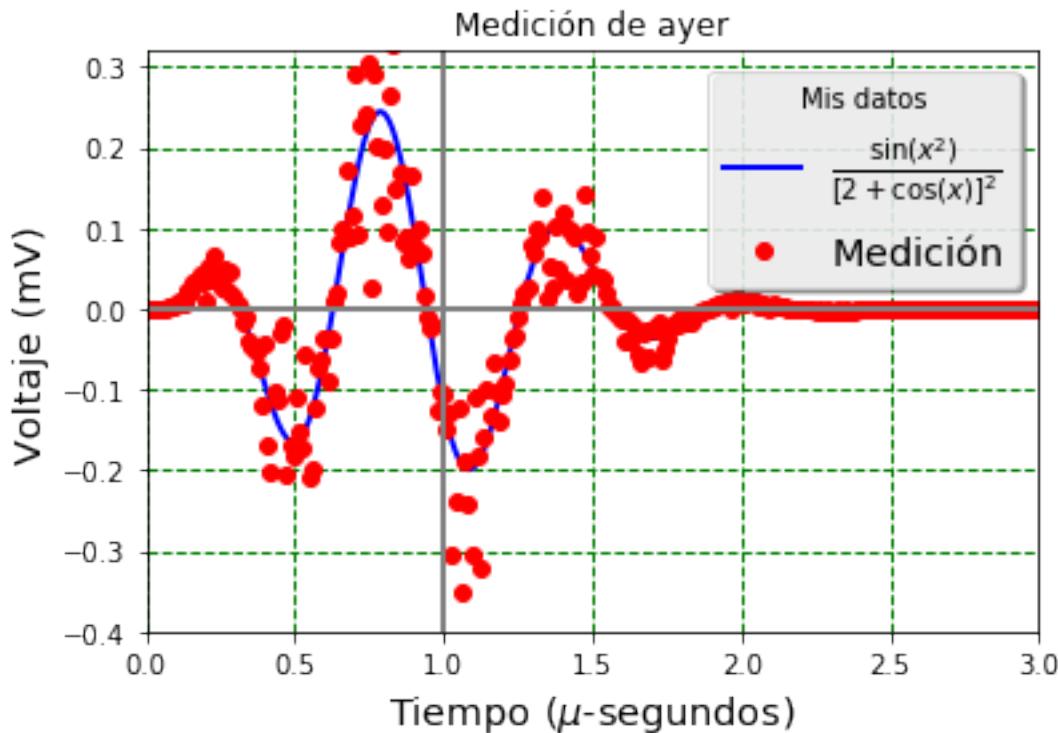
10.6 Exportar las figuras

Para guardar las figuras en alguno de los formatos disponibles utilizamos la función `savefig()`.

```

foname = 'ej_plot_osc'
plt.plot(x,y, '-b', label=r"$\frac{\sin(x^2)}{[2 + \cos(x)]^2}$")
plt.plot(x,yexp, 'or', label="$\mathbf{Medición}$")
plt.legend(loc="best", title="Mis datos", fontsize='x-large')
plt.xlabel(r'Tiempo ($\mu$-segundos)', fontsize='x-large')
plt.ylabel("Voltaje (mV)", fontsize='x-large')
plt.title("Medición de ayer")
plt.axvline(x=1, color='gray')
plt.axhline(color='gray')
plt.xlim((0,3))
plt.ylim((-0.4, 0.32))
plt.grid(color='green', linestyle='dashed', linewidth=1)
plt.savefig('{}.png'.format(foname), dpi=200)
plt.savefig('{}.pdf'.format(foname))

```



```
help(plt.grid)
```

Help on function `grid` in module `matplotlib.pyplot`:

```
grid(b=None, which='major', axis='both', **kwargs)
Configure the grid lines.

Parameters
-----
b : bool or None, optional
    Whether to show the grid lines. If any kwargs are supplied,
    it is assumed you want the grid on and b will be set to True.

    If b is None and there are no kwargs, this toggles the
    visibility of the lines.

which : {'major', 'minor', 'both'}, optional
    The grid lines to apply the changes on.

axis : {'both', 'x', 'y'}, optional
    The axis to apply the changes on.

**kwargs : .Line2D properties
    Define the line properties of the grid, e.g.::

        grid(color='r', linestyle='-', linewidth=2)

    Valid kwargs are
```

```

agg_filter: a filter function, which takes a (m, n, 3) float array and ↴
↪ a dpi value, and returns a (m, n, 3) array
alpha: float
animated: bool
antialiased or aa: bool
clip_box: .Bbox
clip_on: bool
clip_path: [(~matplotlib.path.Path, .Transform) | .Patch | None]
color or c: color
contains: callable
dash_capstyle: {'butt', 'round', 'projecting'}
dash_joinstyle: {'miter', 'round', 'bevel'}
dashes: sequence of floats (on/off ink in points) or (None, None)
drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps- ↴
↪ post'}, default: 'default'
figure: .Figure
fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
gid: str
in_layout: bool
label: object
linestyle or ls: {'-', '--', '-.', ':', ''}, (offset, on-off-seq), ...
linewidth or lw: float
marker: marker style
markeredgecolor or mec: color
markeredgewidth or mew: float
markerfacecolor or mfc: color
markerfacecoloralt or mfcalc: color
markersize or ms: float
markevery: None or int or (int, int) or slice or List[int] or float or ↴
↪ (float, float)
path_effects: .AbstractPathEffect
picker: float or callable[[Artist, Event], Tuple[bool, dict]]
pickradius: float
rasterized: bool or None
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
solid_capstyle: {'butt', 'round', 'projecting'}
solid_joinstyle: {'miter', 'round', 'bevel'}
transform: matplotlib.transforms.Transform
url: str
visible: bool
xdata: 1D array
ydata: 1D array
zorder: float

```

Notes

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the `.Line2D` objects comprising the grid. Therefore, to set grid zorder, use `.set_axisbelow` or, for more control, call the `~matplotlib.axis.Axis.set_zorder` method of each axis.

Acá también hemos utilizado formato tipo LaTeX para parte del texto. Si utilizamos una expresión encerrada entre los

símbolos \$, matplotlib interpreta que está escrito en (un subconjunto) de LaTeX.

Matplotlib tiene un procesador de símbolos interno para mostrar la notación en LaTeX que reconoce los elementos más comunes, o puede elegirse utilizar un procesador LaTeX externo.

10.7 Dos gráficos en la misma figura

Hay varias funciones que permiten poner más de un gráfico en la misma figura. Veamos un ejemplo utilizando la función `subplots()`

```
%pwd
```

```
'/home/fiol/trabajo/clases/pythons/clases-python/clases'
```

```
# %load scripts/ejemplo_08_5.py
#! /usr/bin/ipython3

""" Script realizado durante la clase 8. Dos figuras """
from os.path import join

import numpy as np
import matplotlib.pyplot as plt
plt.ion()

fname = 'ej_oscil_aten_err'
# Levantamos los datos
pardir = '...'
datfile = join(pardir, 'data/{}.dat'.format(fname))

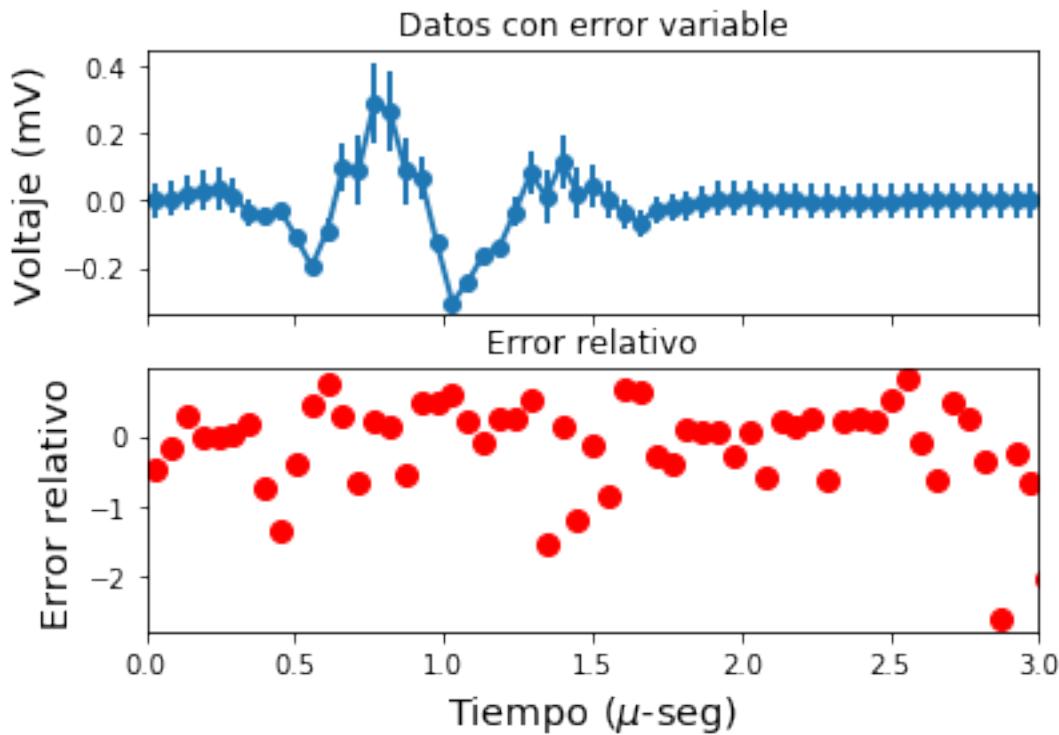
x1, y1, y2 = np.loadtxt(datfile, unpack=True)
# Vamos a graficar sólo algunos valores (uno de cada 5)
x = x1[3:-10:5]
y = y1[3:-10:5]
yexp = y2[3:-10:5]

# Ejemplo de barras de error que dependen del eje x
error = 0.05 + 0.3 * y

fig, (ax0, ax1) = plt.subplots(num='subplots', nrows=2, sharex=True)
ax0.errorbar(x, yexp, yerr=error, fmt='-o')
ax1.plot(x, 2 * (yexp - y) / (yexp + y), 'or', markersize=8)

# Límites de graficación y títulos
ax0.set_title('Datos con error variable')
ax1.set_title('Error relativo')
ax0.set_ylabel('Voltaje (mV)', fontsize='x-large')
ax1.set_xlabel(r'Tiempo ($\mu$-seg)', fontsize='x-large')
ax1.set_ylabel('Error relativo', fontsize='x-large')
ax1.set_xlim((0, 3))

# Guardamos el resultado
plt.savefig('{}.png'.format(fname), dpi=72)
```

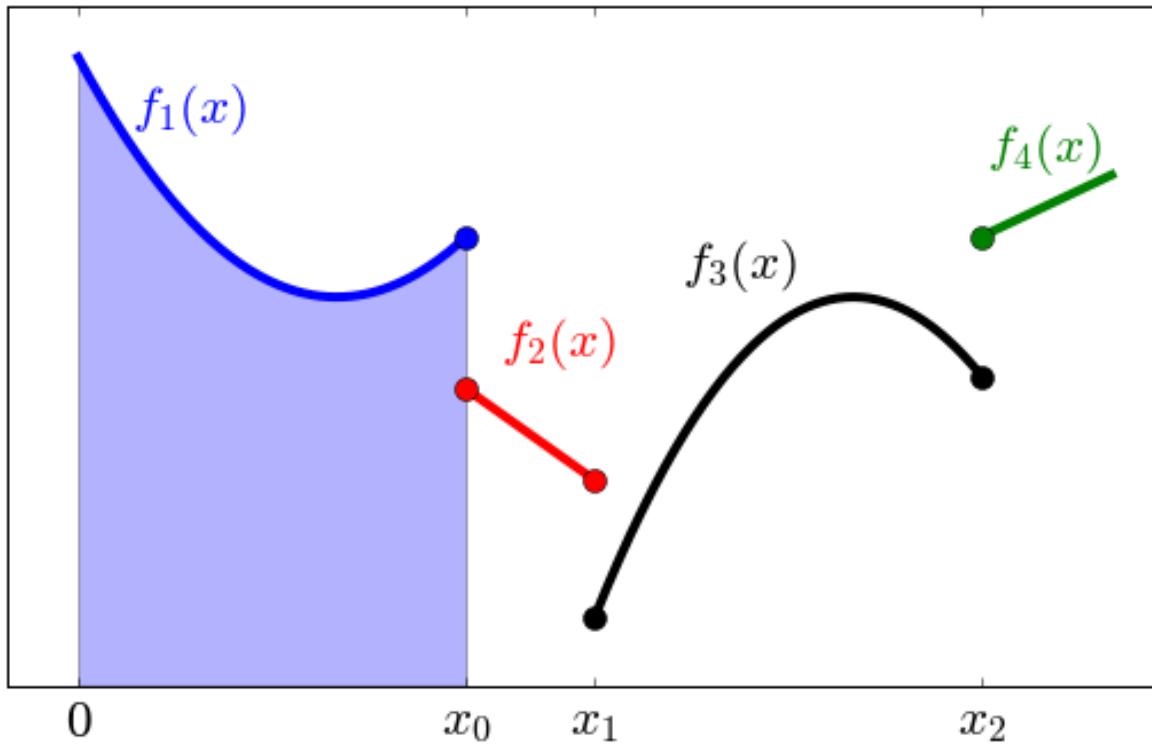


10.8 Ejercicios 09 (b)

2. Para la función definida a trozos:

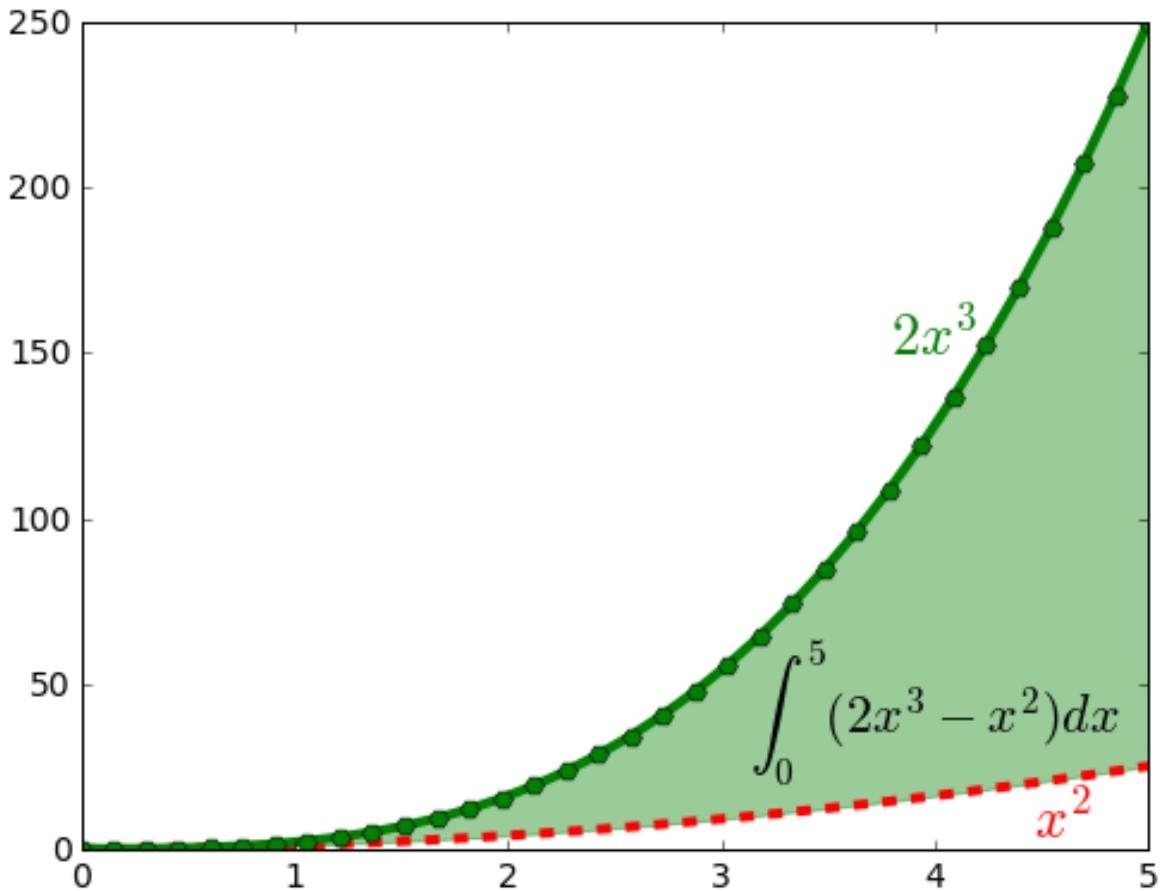
$$f(x) = \begin{cases} f_1(x) = x^2/8 & -\pi < x \leq \pi/2 \\ f_2(x) = -0,3x & \pi/2 < x < \pi \\ f_3(x) = -(x - 2\pi)^2/6 & \pi \leq x \leq 5\pi/2 \\ f_4(x) = (x - 2\pi)/5 & 5\pi/2 < x \leq 3\pi \end{cases}$$

realizar la siguiente figura de la manera más fiel posible.



Pistas: Buscar información sobre `plt.fill_between()` y sobre `plt.xticks` y `plt.yticks`.

3. Rehacer la siguiente figura:



10.9 Personalizando el modo de visualización

Matplotlib da la posibilidad de modificar el estilo de la graficación en distintas etapas.

10.9.1 Archivo de configuración

Cuando uno carga el módulo busca un archivo de configuración llamado `matplotlibrc`

1. Primero busca un archivo de configuración en el directorio de trabajo también lo lee. En cada caso sobreescribe las variables.
2. Si la variable `MATPLOTLIBRC` existe (para el usuario), busca el archivo `$MATPLOTLIBRC/matplotlibrc`
3. Luego lee un archivo de configuración global del usuario, que dependiendo del sistema operativo puede ser:
 - * En Linux, `config/matplotlib/matplotlibrc` (o en `$XDG_CONFIG_HOME/matplotlib/matplotlibrc` si la variable `XDG_CONFIG_HOME` existe)
 - * En otras plataformas puede estar en algún lugar como: `C:\Documents and Settings\USUARIO\.matplotlib`
4. Finalmente lee el archivo global de la instalación, `INSTALL/matplotlib/mpl-data/matplotlibrc`, donde `INSTALL` se refiere al lugar de instalación

En cualquier caso, podemos obtener el directorio y archivo de configuración con las funciones:

```
import matplotlib

matplotlib.get_configdir()

'~/home/fiol/.config/matplotlib'

matplotlib.matplotlib_fname()

'~/home/fiol/.config/matplotlib/matplotlibrc'

!head -n 40 '~/home/fiol/.config/matplotlib/matplotlibrc'

# -- mode: Conf[Colon]; --
## MATPLOTLIBRC FORMAT
# This is a sample matplotlib configuration file - you can find a copy
# of it on your system in
# site-packages/matplotlib/mpl-data/matplotlibrc. If you edit it
# there, please note that it will be overwritten in your next install.
# If you want to keep a permanent local copy that will not be
# overwritten, place it in HOME/.matplotlib/matplotlibrc (unix/linux
# like systems) and C:Documents and Settingsyourname.matplotlib
# (win32 systems).
#
# This file is best viewed in a editor which supports python mode
# syntax highlighting. Blank lines, or lines starting with a comment
# symbol, are ignored, as are trailing comments. Other lines must
# have the format
#     key : val # optional comment
#
# Colors: for the color values below, you can either use - a
# matplotlib color string, such as r, k, or b - an rgb tuple, such as
# (1.0, 0.5, 0.0) - a hex string, such as ff00ff or #ff00ff - a scalar
# grayscale intensity such as 0.75 - a legal html color name, eg red,
# blue, darkslategray

#### CONFIGURATION BEGINS HERE

# the default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# CocoaAgg MacOSX Qt4Agg TkAgg WX WXAgg Agg Cairo GDK PS PDF SVG
# Template
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH) as
# 'module://my_backend'
# backend      : Qt5Agg
# backend      : GTK3Cairo
backend : TkAgg

# If you are using the Qt4Agg backend, you can choose here
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
#backend.qt4 : PyQt4          # PyQt4 | PySide
```

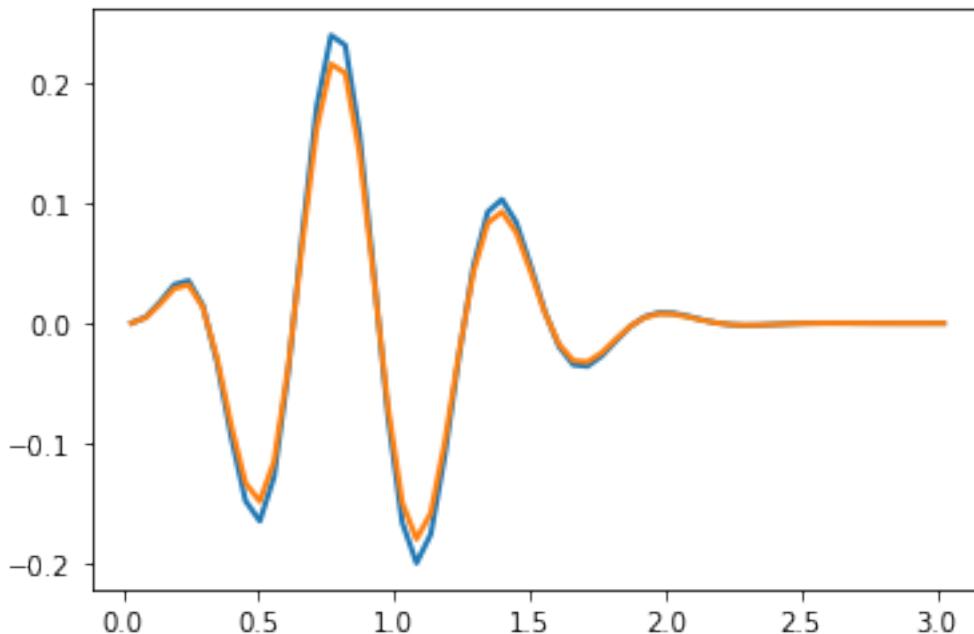
10.9.2 Hojas de estilo

Matplotlib ha incorporado en los últimos años un paquete que permite cambiar estilos fácilmente utilizando los mismos nombres para los parámetros que hay en el archivo de configuración `matplotlibrc`.

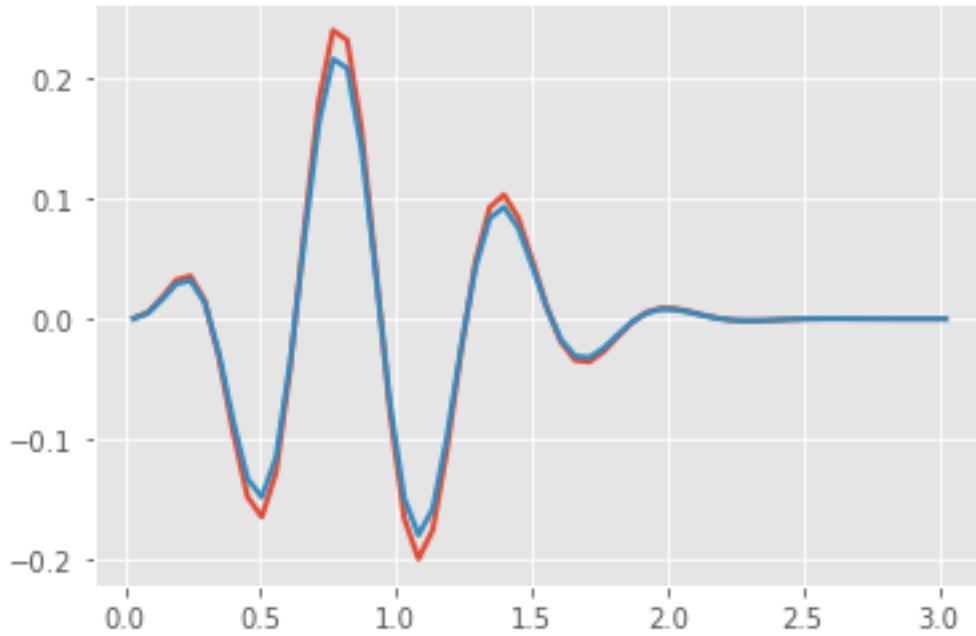
Este paquete tiene pre-definidos unos pocos estilos, entre ellos varios que emulan otros paquetes o programas. Veamos un ejemplo:

```
plt.plot(x,y, x, 0.9*y)
```

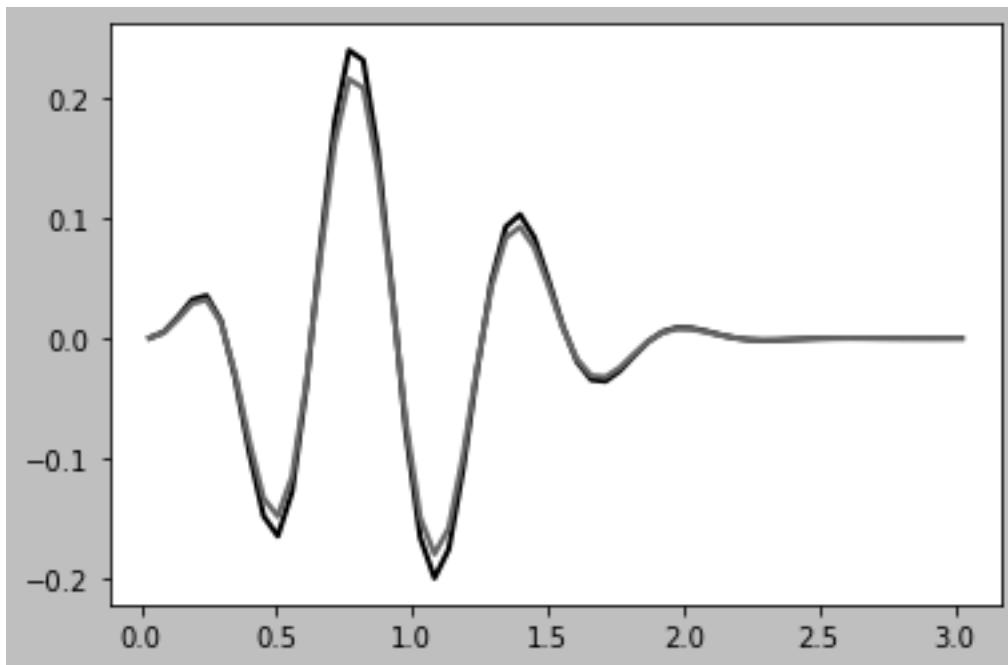
```
[<matplotlib.lines.Line2D at 0x7fa9f8255e50>,
 <matplotlib.lines.Line2D at 0x7fa9f8255290>]
```



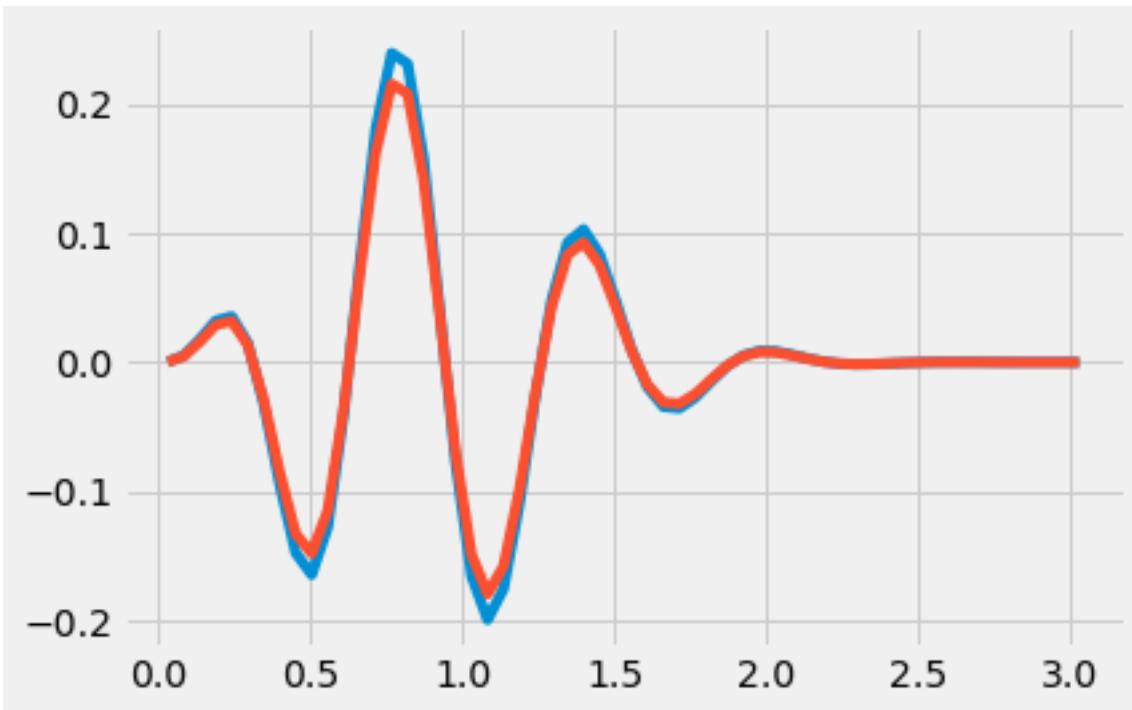
```
with plt.style.context('ggplot'):
    plt.plot(x,y, x, 0.9*y)
```



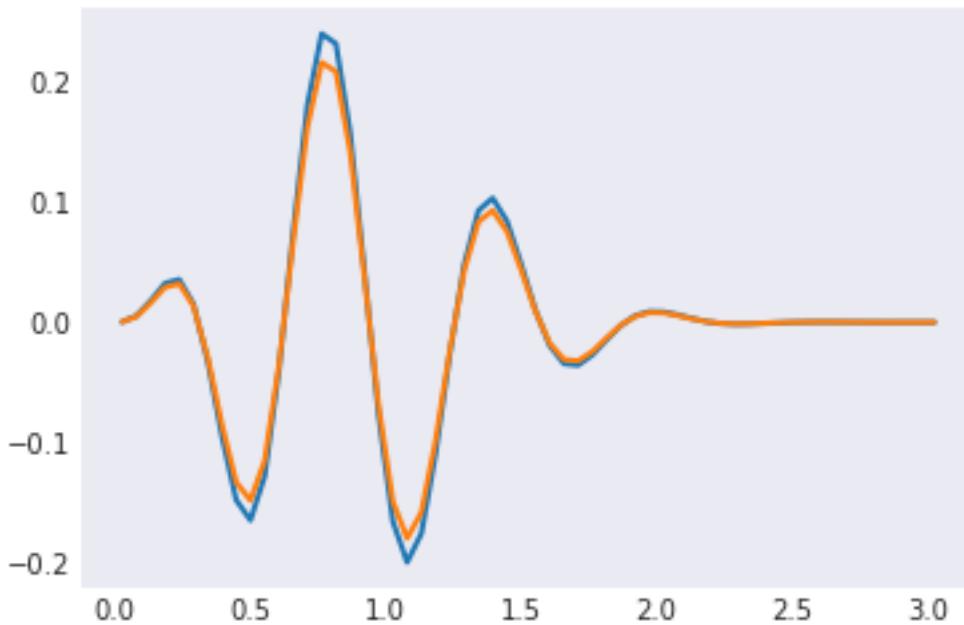
```
with plt.style.context('grayscale'):
    plt.plot(x,y, x, 0.9*y)
```



```
with plt.style.context('fivethirtyeight'):
    plt.plot(x,y, x, 0.9*y)
```



```
with plt.style.context('seaborn-dark'):
    plt.plot(x,y, x, 0.9*y)
```



Los estilos disponibles están guardados en la variable `available` (una lista)

```
plt.style.available
```

```
['seaborn-dark',
 'bmh',
```

(continué en la próxima página)

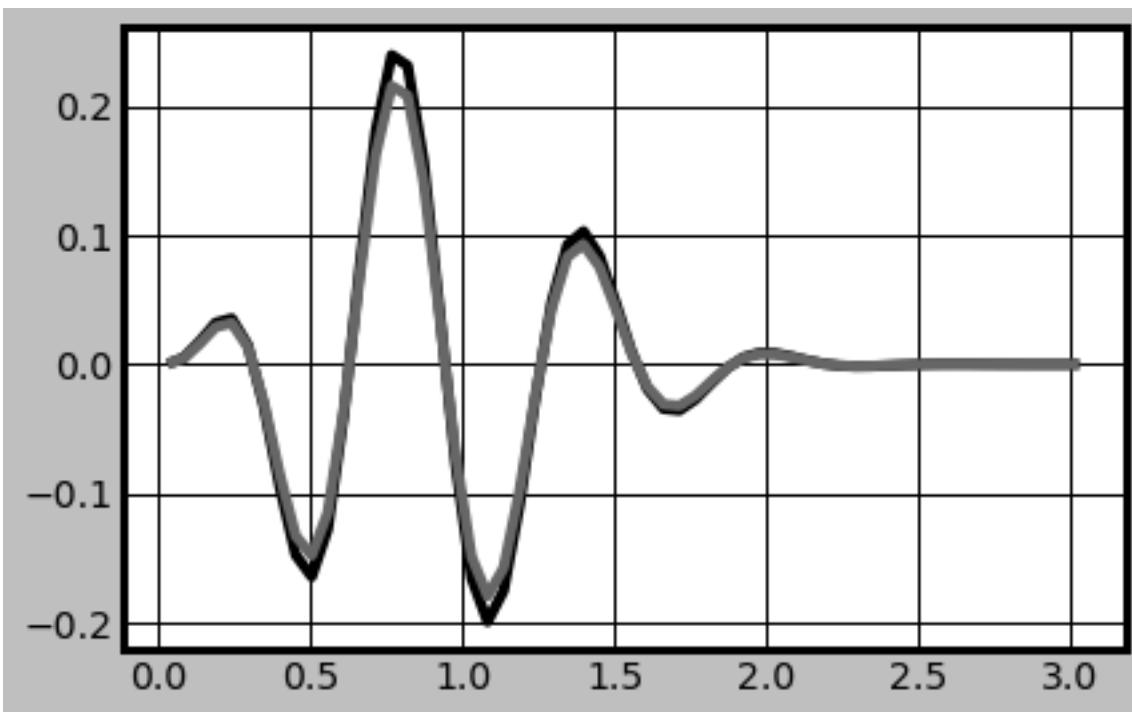
(proviene de la página anterior)

```
'seaborn-paper',
'seaborn-poster',
'seaborn-deep',
'ggplot',
'seaborn-dark-palette',
'seaborn-bright',
'seaborn-notebook',
'seaborn-pastel',
'seaborn-darkgrid',
'seaborn-colorblind',
'seaborn-whitegrid',
'fast',
'dark_background',
'classic',
'seaborn-muted',
'_classic_test',
'seaborn',
'grayscale',
'seaborn-talk',
'fivethirtyeight',
'seaborn-white',
'tableau-colorblind10',
'seaborn-ticks',
'Solarize_Light2',
'presentation',
'paper',
'darker']
```

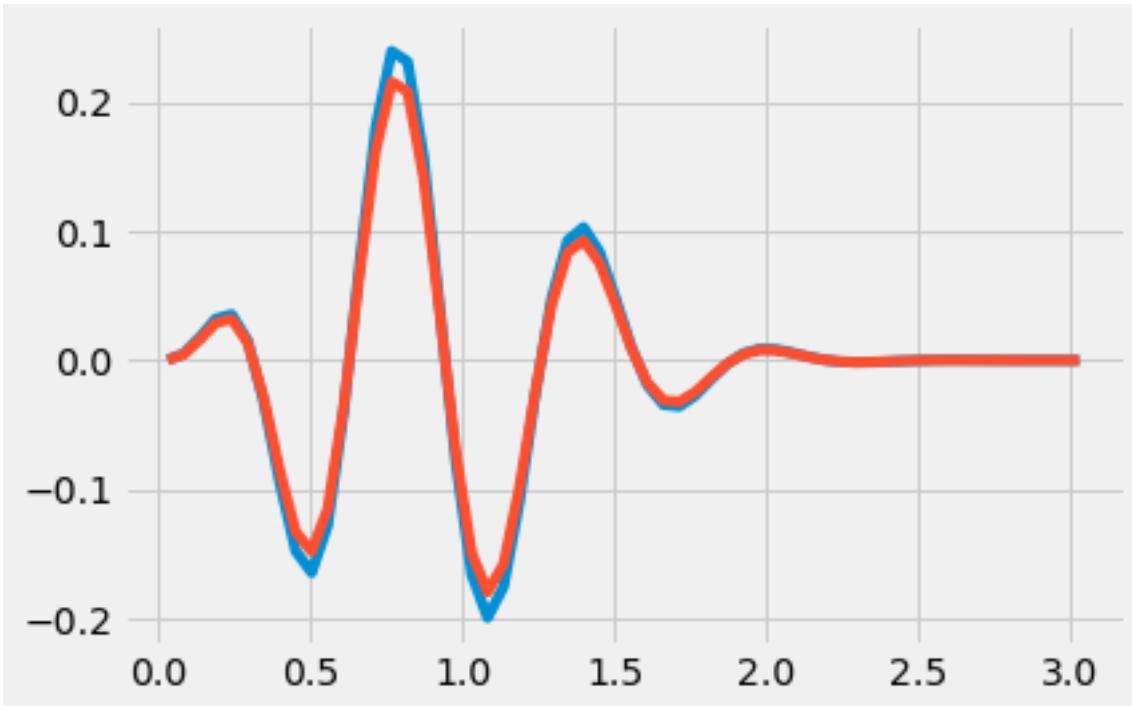
Combinando estilos

Los estilos pueden combinarse. En este caso, debe pasarse una lista de *strings* con los nombres de los estilos a aplicar. Se aplican en forma secuencial. Si dos estilos definen diferentes valores para una variable, el posterior sobreescribe los valores previos.

```
with plt.style.context(['fivethirtyeight','grayscale']):
    plt.plot(x,y, x, 0.9*y)
```



```
with plt.style.context(['grayscale', 'fivethirtyeight']):
    plt.plot(x,y, x, 0.9*y)
```



Creación de estilos propios

Podemos crear estilos propios, modificando los defaults con una sintaxis similar a la del archivo de configuración. Por ejemplo creamos un archivo estilo_test con algunos parámetros

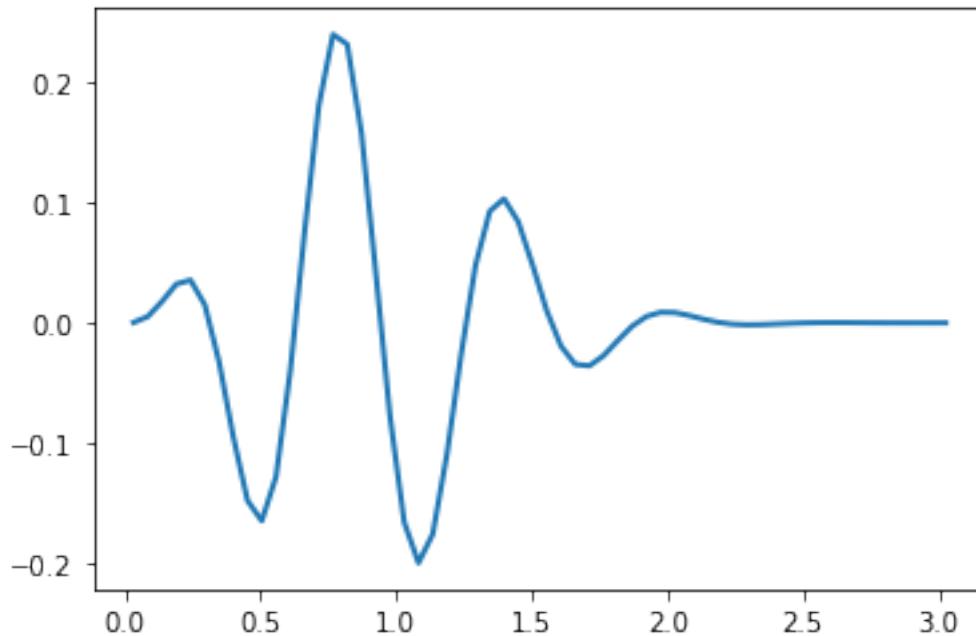
```
!echo "lines.linewidth : 5" > estilo_test
!echo "xtick.labelsize: 24" >> estilo_test
```

```
!cat estilo_test
```

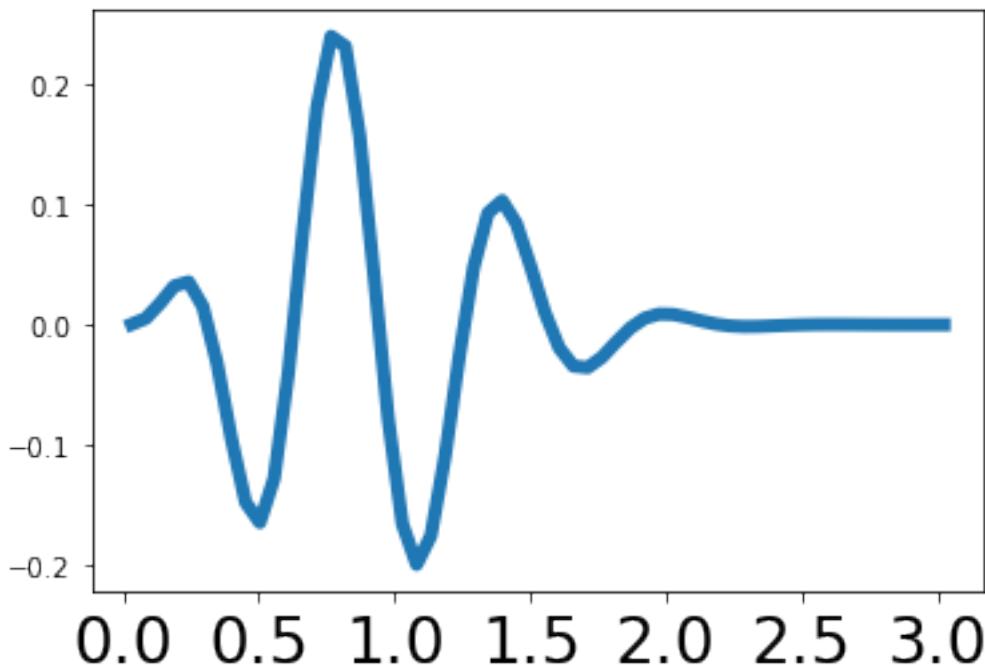
```
lines.linewidth : 5
xtick.labelsize: 24
```

```
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f7e2e150>]
```



```
with plt.style.context('./estilo_test'):
    plt.plot(x,y)
```



Para encontrar el lugar donde guardar las hojas de estilo podemos utilizar las funciones de *matplotlib*:

```
matplotlib.get_configdir()
```

```
'/home/fiol/.config/matplotlib'
```

```
ls -1 /home/fiol/.config/matplotlib/stylelib/
```

```
darker.mplstyle
darker.mplstyle~
paper.mplstyle
presentation.mplstyle
```

10.9.3 Modificación de parámetros dentro de programas

Podemos cambiar directamente los parámetros dentro de nuestros programas modificando el diccionario `matplotlib.rcParams`

```
import matplotlib as mpl
mpl.rcParams
```

```
/usr/lib/python3.7/site-packages/IPython/lib/pretty.py:689:_
  ↪MatplotlibDeprecationWarning:
The examples.directory rcparam was deprecated in Matplotlib 3.0 and will be removed_
  ↪in 3.2. In the future, examples will be found relative to the 'datapath' directory.
    output = repr(obj)
```

```
RcParams({_internal.classic_mode': False,
          'agg.path.chunksize': 0,
          'animation.avconv_args': []},
```

(continué en la próxima página)

(proviene de la página anterior)

```
'animation.avconv_path': 'avconv',
'animation.bitrate': -1,
'animation.codec': 'h264',
'animation.convert_args': [],
'animation.convert_path': 'convert',
'animation.embed_limit': 20.0,
'animation.ffmpeg_args': [],
'animation.ffmpeg_path': 'ffmpeg',
'animation.frame_format': 'png',
'animation.html': 'none',
'animation.html_args': [],
'animation.writer': 'ffmpeg',
'axes.autolimit_mode': 'data',
'axes.axisbelow': 'line',
'axes.edgecolor': 'black',
'axes.facecolor': 'white',
'axes.formatter.limits': [-7, 7],
'axes.formatter.min_exponent': 0,
'axes.formatter.offset_threshold': 4,
'axes.formatter.use_locale': False,
'axes.formatter.use_mathtext': False,
'axes.formatter.useoffset': True,
'axes.grid': False,
'axes.grid.axis': 'both',
'axes.grid.which': 'major',
'axes.labelcolor': 'black',
'axes.labelpad': 4.0,
'axes.labelsize': 14.0,
'axes.labelweight': 'normal',
'axes.linewidth': 0.8,
'axes.prop_cycle': cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']),
'axes.spines.bottom': True,
'axes.spines.left': True,
'axes.spines.right': True,
'axes.spines.top': True,
'axes.titlepad': 6.0,
'axes.titlesize': 'large',
'axes.titleweight': 'normal',
'axes.unicode_minus': True,
'axes.xmargin': 0.05,
'axes.ymargin': 0.05,
'axes3d.grid': True,
'backend': 'module://ipykernel.pylab.backend_inline',
'backend_fallback': True,
'boxplot.bootstrap': None,
'boxplot.boxprops.color': 'black',
'boxplot.boxprops.linestyle': '-',
'boxplot.boxprops.linewidth': 1.0,
'boxplot.capprops.color': 'black',
'boxplot.capprops.linestyle': '-',
'boxplot.capprops.linewidth': 1.0,
'boxplot.flierprops.color': 'black',
'boxplot.flierprops.linestyle': 'none',
'boxplot.flierprops.linewidth': 1.0,
'boxplot.flierprops.marker': 'o',
'boxplot.flierprops.markeredgecolor': 'black',
```

(continué en la próxima página)

(proviene de la página anterior)

```

'boxplot.flierprops.markeredgewidth': 1.0,
'boxplot.flierprops.markerfacecolor': 'none',
'boxplot.flierprops.markersize': 6.0,
'boxplot.meanline': False,
'boxplot.meanprops.color': 'C2',
'boxplot.meanprops.linestyle': '--',
'boxplot.meanprops.linewidth': 1.0,
'boxplot.meanprops.marker': '^',
'boxplot.meanprops.markeredgecolor': 'C2',
'boxplot.meanprops.markerfacecolor': 'C2',
'boxplot.meanprops.markersize': 6.0,
'boxplot.medianprops.color': 'C1',
'boxplot.medianprops.linestyle': '-',
'boxplot.medianprops.linewidth': 1.0,
'boxplot.notch': False,
'boxplot.patchartist': False,
'boxplot.showbox': True,
'boxplot.showcaps': True,
'boxplot.showfliers': True,
'boxplot.showmeans': False,
'boxplot.vertical': True,
'boxplot.whiskerprops.color': 'black',
'boxplot.whiskerprops.linestyle': '-',
'boxplot.whiskerprops.linewidth': 1.0,
'boxplot.whiskers': 1.5,
'contour.corner_mask': True,
'contour.negative_linestyle': 'dashed',
'datapath': '/usr/share/matplotlib/mpl-data',
'date.autoformatter.day': '%Y-%m-%d',
'date.autoformatter.hour': '%m-%d %H',
'date.autoformatter.microsecond': '%M:%S.%f',
'date.autoformatter.minute': '%d %H:%M',
'date.autoformatter.month': '%Y-%m',
'date.autoformatter.second': '%H:%M:%S',
'date.autoformatter.year': '%Y',
'docstring.hardcopy': False,
'errorbar.capsize': 0.0,
'examples.directory': '',
'figure.autolayout': False,
'figure.constrained_layout.h_pad': 0.04167,
'figure.constrained_layout.hspace': 0.02,
'figure.constrained_layout.use': False,
'figure.constrained_layout.w_pad': 0.04167,
'figure.constrained_layout.wspace': 0.02,
'figure.dpi': 72.0,
'figure.edgecolor': (1, 1, 1, 0),
'figure.facecolor': (1, 1, 1, 0),
'figure.figsize': [6.0, 4.0],
'figure.frameon': True,
'figure.max_open_warning': 20,
'figure.subplot.bottom': 0.125,
'figure.subplot.hspace': 0.2,
'figure.subplot.left': 0.125,
'figure.subplot.right': 0.9,
'figure.subplot.top': 0.88,
'figure.subplot.wspace': 0.2,
'figure.titlesize': 'large',

```

(continué en la próxima página)

(proviene de la página anterior)

```
'figure.titleweight': 'normal',
'font.cursive': ['Apple Chancery',
                 'Textile',
                 'Zapf Chancery',
                 'Sand',
                 'Script MT',
                 'Felipa',
                 'cursive'],
'font.family': ['sans-serif'],
'font.fantasy': ['Comic Sans MS',
                 'Chicago',
                 'Charcoal',
                 'Impact',
                 'Western',
                 'Humor Sans',
                 'xkcd',
                 'fantasy'],
'font.monospace': ['DejaVu Sans Mono',
                    'Bitstream Vera Sans Mono',
                    'Computer Modern Typewriter',
                    'Andale Mono',
                    'Nimbus Mono L',
                    'Courier New',
                    'Courier',
                    'Fixed',
                    'Terminal',
                    'monospace'],
'font.sans-serif': ['DejaVu Sans',
                     'Bitstream Vera Sans',
                     'Computer Modern Sans Serif',
                     'Lucida Grande',
                     'Verdana',
                     'Geneva',
                     'Lucid',
                     'Arial',
                     'Helvetica',
                     'Avant Garde',
                     'sans-serif'],
'font.serif': ['DejaVu Serif',
                'Bitstream Vera Serif',
                'Computer Modern Roman',
                'New Century Schoolbook',
                'Century Schoolbook L',
                'Utopia',
                'ITC Bookman',
                'Bookman',
                'Nimbus Roman No9 L',
                'Times New Roman',
                'Times',
                'Palatino',
                'Charter',
                'serif'],
'font.size': 10.0,
'font.stretch': 'normal',
'font.style': 'normal',
'font.variant': 'normal',
'font.weight': 'normal',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'grid.alpha': 1.0,
'grid.color': '#b0b0b0',
'grid.linestyle': '-',
'grid.linewidth': 0.8,
'hatch.color': 'black',
'hatch.linewidth': 1.0,
'hist.bins': 10,
'image.aspect': 'equal',
'image.cmap': 'viridis',
'image.composite_image': True,
'image.interpolation': 'nearest',
'image.lut': 256,
'image.origin': 'upper',
'image.resample': True,
'interactive': True,
'keymap.all_axes': ['a'],
'keymap.back': ['left', 'c', 'backspace', 'MouseButton.BACK'],
'keymap.copy': ['ctrl+c', 'cmd+c'],
'keymap.forward': ['right', 'v', 'MouseButton.FORWARD'],
'keymap.fullscreen': ['f', 'ctrl+f'],
'keymap.grid': ['g'],
'keymap.grid_minor': ['G'],
'keymap.help': ['f1'],
'keymap.home': ['h', 'r', 'home'],
'keymap.pan': ['p'],
'keymap.quit': ['ctrl+w', 'cmd+w', 'q'],
'keymap.quit_all': ['W', 'cmd+W', 'Q'],
'keymap.save': ['s', 'ctrl+s'],
'keymap.xscale': ['k', 'L'],
'keymap.yscale': ['l'],
'keymap.zoom': ['o'],
'legend.borderaxespad': 0.5,
'legend.borderpad': 0.4,
'legend.columnspacing': 2.0,
'legend.edgecolor': '0.8',
'legend.facecolor': 'inherit',
'legend.fancybox': True,
'legend.fontsize': 'large',
'legend.framealpha': 0.8,
'legend.frameon': True,
'legend.handleheight': 0.7,
'legend.handlelength': 2.0,
'legend.handletextpad': 0.8,
'legend.labelspacing': 0.5,
'legend.loc': 'best',
'legend.markerscale': 1.0,
'legend.numpoints': 1,
'legend.scatterpoints': 1,
'legend.shadow': True,
'legend.title_fontsize': None,
'lines.antialiased': True,
'lines.color': 'C0',
'lines.dash_capstyle': 'butt',
'lines.dash_joinstyle': 'round',
'lines.dashdot_pattern': [6.4, 1.6, 1.0, 1.6],
'lines.dashed_pattern': [3.7, 1.6],
'lines.dotted_pattern': [1.0, 1.65],
```

(continué en la próxima página)

(proviene de la página anterior)

```
'lines.linestyle': '-',
'lines.linewidth': 2.0,
'lines.marker': 'None',
'lines.markeredgecolor': 'auto',
'lines.markeredgewidth': 1.0,
'lines.markerfacecolor': 'auto',
'lines.markersize': 6.0,
'lines.scale_dashes': True,
'lines.solid_capstyle': 'projecting',
'lines.solid_joinstyle': 'round',
'markers.fillstyle': 'full',
'mathtext.bf': 'sans:bold',
'mathtext.cal': 'cursive',
'mathtext.default': 'it',
'mathtext.fallback_to_cm': True,
'mathtext.fontset': 'dejavusans',
'mathtext.it': 'sans:italic',
'mathtext.rm': 'sans',
'mathtext.sf': 'sans',
'mathtext.tt': 'monospace',
'patch.antialiased': True,
'patch.edgecolor': 'black',
'patch.facecolor': 'C0',
'patch.force_edgecolor': False,
'patch.linewidth': 1.0,
'path.effects': [],
'path.simplify': True,
'path.simplify_threshold': 0.1111111111111111,
'path.sketch': None,
'path.snap': True,
'pdf.compression': 6,
'pdf.fonttype': 42,
'pdf.inheritcolor': False,
'pdf.use14corefonts': False,
'pgf.preamble': '',
'pgf.rcfonts': True,
'pgf.texsystem': 'xelatex',
'polaraxes.grid': True,
'ps.distiller.res': 6000,
'ps.fonttype': 42,
'ps.papersize': 'a4',
'ps.useafm': False,
'ps.usedistiller': False,
'savefig.bbox': 'tight',
'savefig.directory': '',
'savefig.dpi': 'figure',
'savefig.edgecolor': 'white',
'savefig.facecolor': 'white',
'savefig.format': 'png',
'savefig.frameon': True,
'savefig.jpeg_quality': 95,
'savefig.orientation': 'portrait',
'savefig.pad_inches': 0.1,
'savefig.transparent': False,
'scatter.edgecolors': 'face',
'scatter.marker': 'o',
'svg.fonttype': 'path',
```

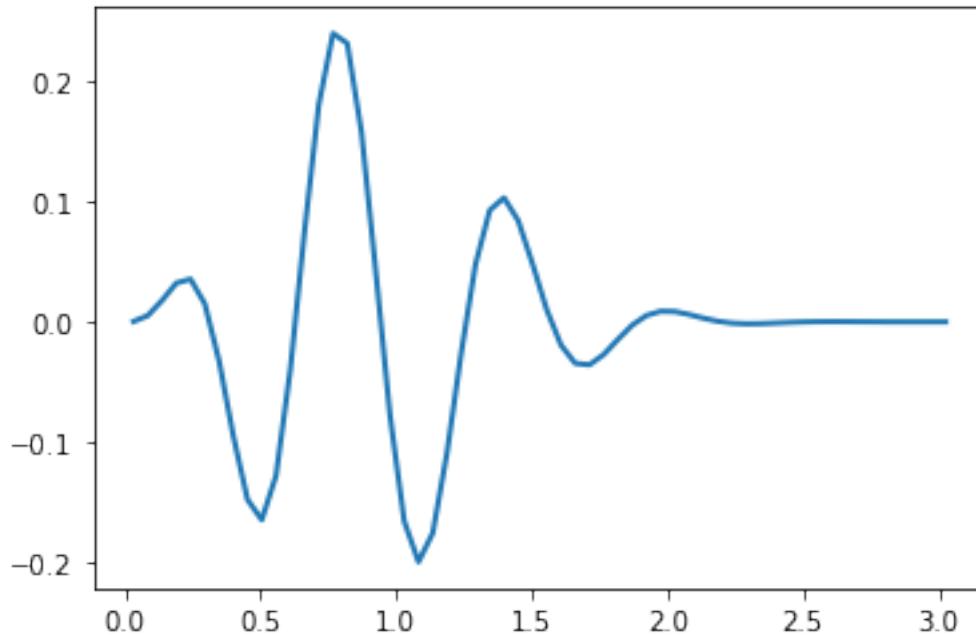
(continué en la próxima página)

(proviene de la página anterior)

```
'svg.hashsalt': None,
'svg.image_inline': True,
'text.antialiased': True,
'text.color': 'black',
'text.hinting': 'auto',
'text.hinting_factor': 8,
'text.latex.preamble': '',
'text.latex.preview': False,
'text.latex_unicode': True,
'text.usetex': False,
'timezone': 'UTC',
'tk.window_focus': False,
'toolbar': 'toolbar2',
'verbose.fileo': 'sys.stdout',
'verbose.level': 'silent',
'webagg.address': '127.0.0.1',
'webagg.open_in_browser': True,
'webagg.port': 8988,
'webagg.port_retries': 50,
'xtick.alignment': 'center',
'xtick.bottom': True,
'xtick.color': 'black',
'xtick.direction': 'out',
'xtick.labelbottom': True,
'xtick.labelsizes': 'medium',
'xtick.labeltop': False,
'xtick.major.bottom': True,
'xtick.major.pad': 3.5,
'xtick.major.size': 3.5,
'xtick.major.top': True,
'xtick.major.width': 0.8,
'xtick.minor.bottom': True,
'xtick.minor.pad': 3.4,
'xtick.minor.size': 2.0,
'xtick.minor.top': True,
'xtick.minor.visible': False,
'xtick.minor.width': 0.6,
'xtick.top': False,
'ytick.alignment': 'center_baseline',
'ytick.color': 'black',
'ytick.direction': 'out',
'ytick.labelleft': True,
'ytick.labelright': False,
'ytick.labelsizes': 'medium',
'ytick.left': True,
'ytick.major.left': True,
'ytick.major.pad': 3.5,
'ytick.major.right': True,
'ytick.major.size': 3.5,
'ytick.major.width': 0.8,
'ytick.minor.left': True,
'ytick.minor.pad': 3.4,
'ytick.minor.right': True,
'ytick.minor.size': 2.0,
'ytick.minor.visible': False,
'ytick.minor.width': 0.6,
'ytick.right': False})
```

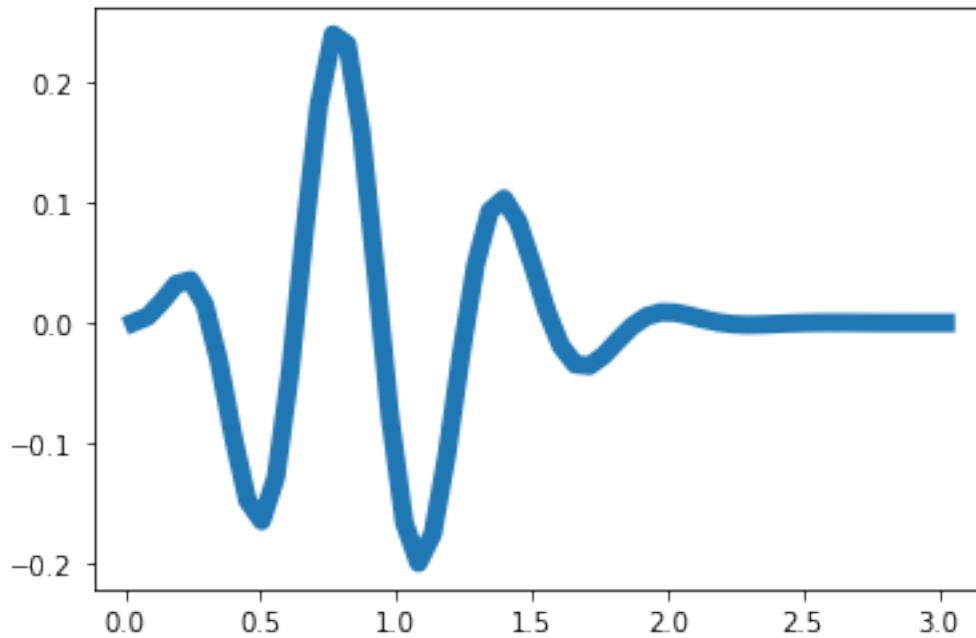
```
# Plot con valores default  
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f8031f10>]
```



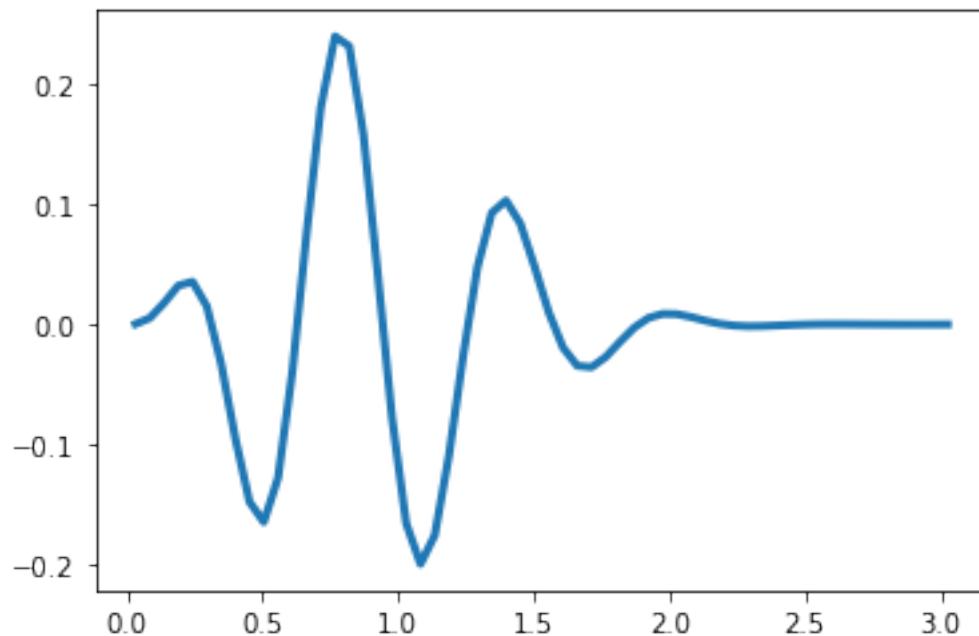
```
# Modificamos el valor default de ancho de linea  
mpl.rcParams['lines.linewidth'] = 7  
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f8065750>]
```



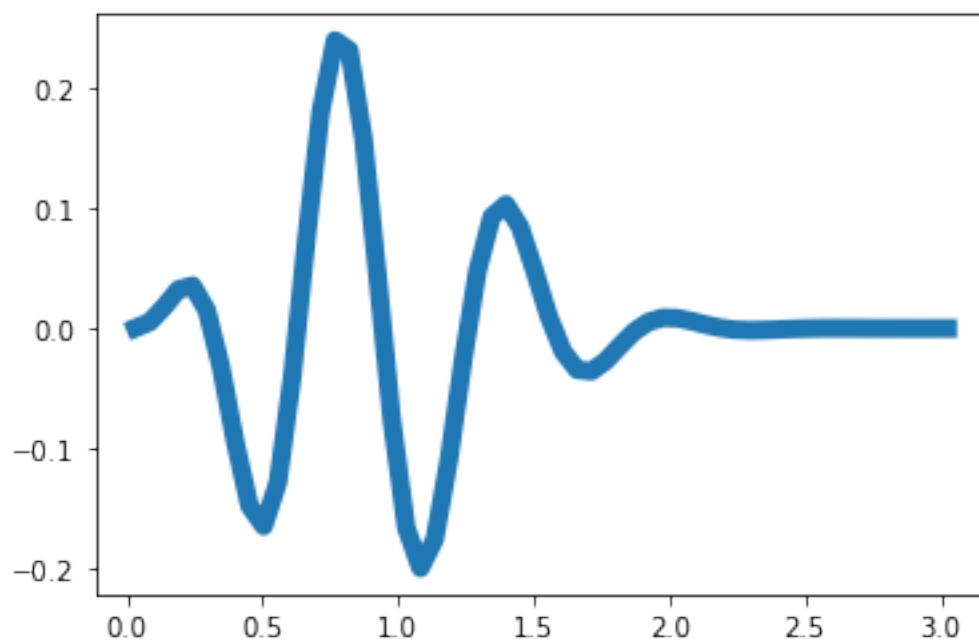
```
# El nuevo valor default podemos sobreescribirlo para este plot particular  
plt.plot(x,y, lw=3)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f7ea4910>]
```



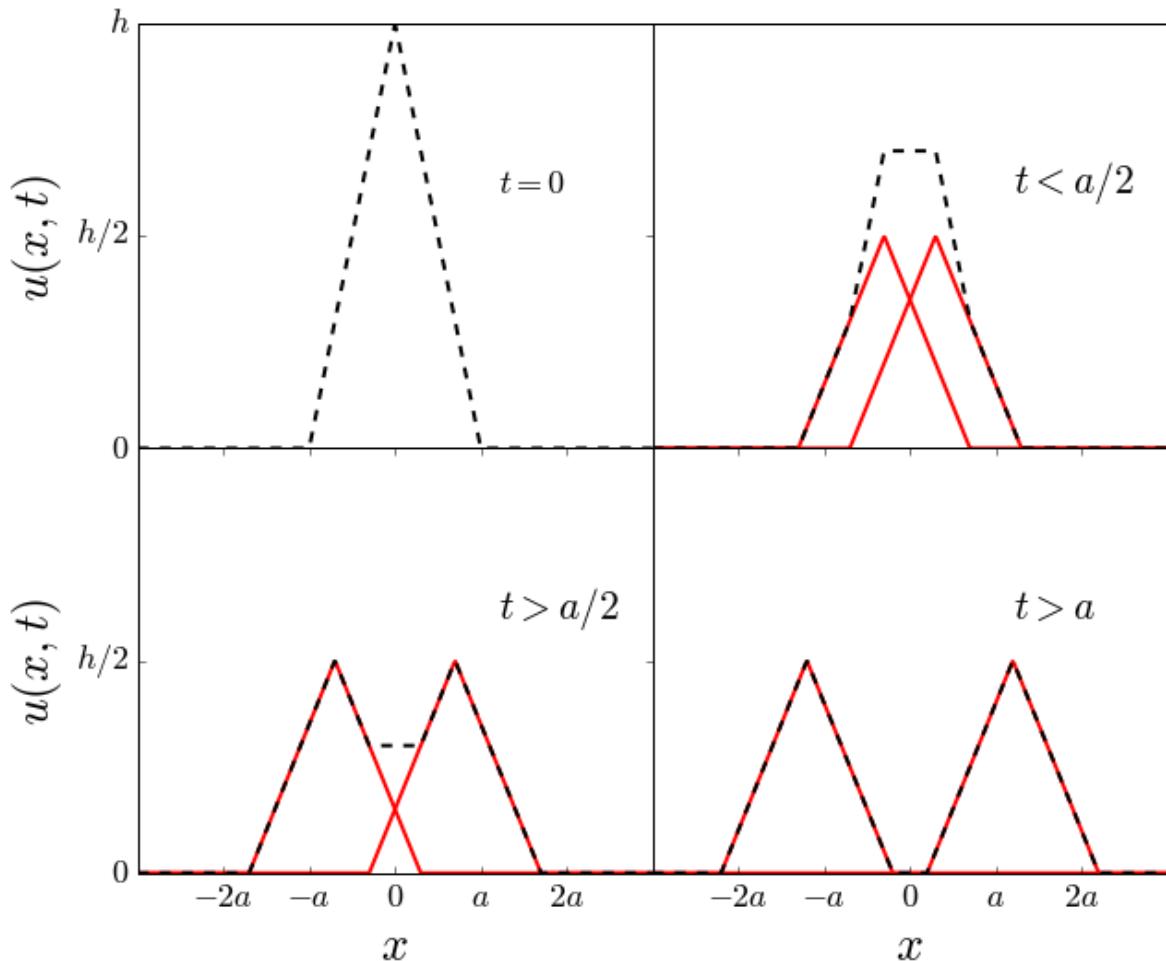
```
# Sin embargo, el nuevo valor default no es modificado  
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa9f7e5ac10>]
```



10.10 Ejercicios 09 (c)

4. Notando que la curva en color negro corresponde a la suma de las dos curvas en rojo, rehacer la siguiente figura:



5. Crear una hoja de estilo que permita hacer gráficos adecuados para posters y presentaciones. Debe modificar los tamaños para hacerlos legibles a mayores distancias (sugerencia 16pt). El tamaño de la letra de los nombres de ejes y en las leyendas debe ser mayor también. Las líneas deben ser más gruesas (sugerencia: ~4), los símbolos de mayor tamaño (sugerencia ~10).

CAPÍTULO 11

Clase 10: Más información sobre Numpy

11.1 Creación y operación sobre Numpy arrays

Vamos a ver algunas características de los arrays de Numpy en un poco más de detalle

11.1.1 Funciones para crear arrays

Vimos varios métodos que permiten crear e inicializar arrays

```
import numpy as np
import matplotlib.pyplot as plt

a= {}
a['empty unid']= np.empty(10)      # Creación de un array de 10 elementos
a['zeros unid']= np.zeros(10)      # Creación de un array de 10 elementos,
                                  # inicializados en cero
a['zeros bidi']= np.zeros((5,2))   # Array bidimensional 10 elementos con *shape* 5x2
a['ones bidi']= np.ones((5,2))    # Array bidimensional 10 elementos con *shape* 5x2,
                                  # inicializado en 1
a['arange']= np.arange(10)        # Array inicializado con una secuencia
a['lineal']= np.linspace(0,10,5)  # Array inicializado con una secuencia,
                                  # equiespaciada
a['log']= np.logspace(0,2,10)    # Array inicializado con una secuencia con espaciado
                                # logarítmico
a['diag']= np.diag(np.arange(5)) # Matriz diagonal a partir de un vector

for k,v in a.items():
    print('Array {}:\n {}'.format(k,v), 80*'')
```

```
Array empty unid:
[ 4.65293215e-310  0.00000000e+000  2.61691551e-219  6.93734310e-310
 6.93734306e-310  2.12298976e+064  6.93734309e-310  4.65293213e-310
 -5.27140143e-104  6.93734309e-310]
```

```
*****
Array zeros unid:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
*****
Array zeros bidi:
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
*****
Array ones bidi:
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
*****
Array arange:
[0 1 2 3 4 5 6 7 8 9]
*****
Array lineal:
[ 0.   2.5   5.   7.5 10. ]
*****
Array log:
[ 1.          1.66810054   2.7825594   4.64158883   7.74263683
 12.91549665  21.5443469   35.93813664  59.94842503 100.        ]
*****
Array diag:
[[0 0 0 0 0]
 [0 1 0 0 0]
 [0 0 2 0 0]
 [0 0 0 3 0]
 [0 0 0 0 4]]
*****
```

La función `np.tile(A, reps)` permite crear un array repitiendo el patrón A las veces indicada por `reps` a lo largo de cada eje

```
a = np.arange(1,6,2)
a
```

```
array([1, 3, 5])
```

```
np.tile(a, 2)
```

```
array([1, 3, 5, 1, 3, 5])
```

```
a1=np.tile(a, (1,2))
```

```
a1.shape
```

```
(1, 6)

a1

array([[1, 3, 5, 1, 3, 5]])

b = [[1,2],[3,4]]

print(b)

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

np.tile(b, (1,2))

array([[1, 2, 1, 2],
       [3, 4, 3, 4]])

np.tile(b, (2,1))

array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

En general, el argumento `reps = (nrows, ncols)` indica el número de repeticiones en filas (hacia abajo) y columnas (hacia la derecha), creando nuevas dimensiones si es necesario

```
a

array([1, 3, 5])

np.tile(a, (3,2))

array([[1, 3, 5, 1, 3, 5],
       [1, 3, 5, 1, 3, 5],
       [1, 3, 5, 1, 3, 5]])
```

11.1.2 Funciones que actúan sobre arrays

Numpy incluye muchas funciones matemáticas que actúan sobre arrays completos (de una o más dimensiones). La lista completa se encuentra en la [documentación](#) e incluye:

```
x = np.linspace(np.pi/180, np.pi, 7)
y = np.geomspace(10,100,7)

print(x)
print(y)
print(x+y)                      # Suma elemento a elemento
print(x*y)                      # Multiplicación elemento a elemento
print(y/x)                       # División elemento a elemento
print(x//2)                      # División entera elemento a elemento
```

```
[0.01745329 0.53814319 1.05883308 1.57952297 2.10021287 2.62090276
 3.14159265]
[ 10.          14.67799268  21.5443469   31.6227766   46.41588834
 68.12920691 100.          ]
[ 10.01745329 15.21613586  22.60317998  33.20229957  48.5161012
 70.75010967 103.14159265]
[1.74532925e-01 7.89886174e+00 2.28118672e+01 4.99489021e+01
 9.74832459e+01 1.78560026e+02 3.14159265e+02]
[572.95779513 27.27525509  20.34725522  20.02046006  22.10056375
 25.99455727 31.83098862]
[0. 0. 0. 0. 1. 1. 1.]
```

```
print('x =', x)
print('square\n', x**2)                      # potencias
print('sin\n', np.sin(x))                     # Seno (np.cos, np.tan)
print("tanh\n", np.tanh(x))                  # tang hiperb (np.sinh, np.cosh)
print('exp\n', np.exp(-x))                   # exponenciales
print('log\n', np.log(x))                     # logaritmo en base e (np.log10)
print('abs\n', np.absolute(x))                # Valor absoluto
print('resto\n', np.remainder(x, 2))         # Resto
```

```
x = [0.01745329 0.53814319 1.05883308 1.57952297 2.10021287 2.62090276
 3.14159265]
square
[3.04617420e-04 2.89598089e-01 1.12112749e+00 2.49489282e+00
 4.41089408e+00 6.86913128e+00 9.86960440e+00]
sin
[1.74524064e-02 5.12542501e-01 8.71784414e-01 9.99961923e-01
 8.63101882e-01 4.97478722e-01 1.22464680e-16]
tanh
[0.01745152 0.49158114 0.78521683 0.91852736 0.97046433 0.9894743
 0.99627208]
exp
[0.98269813 0.58383131 0.34686033 0.20607338 0.12243036 0.07273717
 0.04321392]
log
[-4.04822697 -0.61963061  0.05716743  0.45712289  0.7420387   0.96351882
 1.14472989]
abs
[0.01745329 0.53814319 1.05883308 1.57952297 2.10021287 2.62090276
 3.14159265]
resto
[0.01745329 0.53814319 1.05883308 1.57952297 0.10021287 0.62090276
 1.14159265]
```

11.1.3 Productos entre arrays y productos vectoriales

```
# Creamos arrays unidimensionales (vectores) y bidimensionales (matrices)
v1 = np.array([2, 3, 4])
v2 = np.array([1, 1, 1])
A = np.arange(1,13,2).reshape(2, 3)
B = np.linspace(0.5,11.5,12).reshape(3, 4)
```

```
print(A)
```

```
[ [ 1  3  5]
 [ 7  9 11] ]
```

```
print (B)
```

```
[ [ 0.5  1.5  2.5  3.5]
 [ 4.5  5.5  6.5  7.5]
 [ 8.5  9.5 10.5 11.5]]
```

```
print (v1*v2)
```

```
[2 3 4]
```

```
print (A*v1)
```

```
[ [ 2   9 20]
 [14  27 44] ]
```

Los productos se realizan elemento a elemento, si queremos obtener productos internos o productos entre matrices (o matrices y vectores)

```
print (v1, '.', v2, '=', np.dot(v1, v2))
```

```
[2 3 4] . [1 1 1] = 9
```

```
print( A, 'x', v1, '=', np.dot(A, v1))
```

```
[ [ 1  3  5]
 [ 7  9 11]] x [2 3 4] = [31 85]
```

```
print (A.shape, B.shape)
```

```
(2, 3) (3, 4)
```

```
print( 'A x B = \n', np.dot(A, B) )
```

```
A x B =
[[ 56.5  65.5  74.5  83.5]
[137.5 164.5 191.5 218.5]]
```

```
print( 'B^t x A^t =\n ', np.dot(B.T, A.T) )
```

```
B^t x A^t =
[[ 56.5 137.5]
[ 65.5 164.5]
[ 74.5 191.5]
[ 83.5 218.5]]
```

Además, el módulo numpy.linalg incluye otras funcionalidades como determinantes, normas, determinación de autovalores y autovectores, descomposiciones, etc.

11.1.4 Comparaciones entre arrays

La comparación, como las operaciones y aplicación de funciones se realiza elemento a elemento.

Funciones	Operadores
greater(x1, x2, /[, out, where, casting,])	(x1 > x2)
greater_equal(x1, x2, /[, out, where,])	(x1 >= x2)
less(x1, x2, /[, out, where, casting,])	(x1 < x2)
less_equal(x1, x2, /[, out, where, casting,])	(x1 == x2)
not_equal(x1, x2, /[, out, where, casting,])	(x1 != x2)
equal(x1, x2, /[, out, where, casting,])	(x1 == x2)

```
z = np.array((-1, 3, 4, 0.5, 2, 9, 0.7))
```

```
print(x)
print(y)
print(z)
```

```
[0.01745329 0.53814319 1.05883308 1.57952297 2.10021287 2.62090276
 3.14159265]
[ 10.           14.67799268  21.5443469   31.6227766   46.41588834
 68.12920691 100.          ]
[-1.    3.    4.    0.5   2.    9.    0.7]
```

```
c1 = x <= z
c2 = np.less_equal(z,y)
c3 = np.less_equal(x,y)
print(c1)
print(c2)
print(c3)
```

```
[False  True  True False False  True False]
[ True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True]
```

```
c1 # Veamos que tipo de array es:
```

```
array([False,  True,  True, False, False,  True, False])
```

```
np.sum(c1), np.sum(c2), c3.sum()
```

```
(3, 7, 7)
```

Como vemos, las comparaciones nos dan un vector de variables lógicas. Cuando queremos combinar condiciones no funciona usar las palabras `and` y `or` de *Python* porque estaríamos comparando los dos elementos (arrays completos).

```
print(np.logical_and(c1, c2))
print(c1 & c2)
print(np.logical_and(c2, c3))
print(c2 & c3)
```

```
[False  True  True False False  True False]
[False  True  True False False  True False]
[ True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True]
```

```
print(np.logical_or(c1, c2))
print(c1 | c2)
print(np.logical_or(c2, c3))
print(c2 | c3)
```

```
[ True  True  True  True  True  True  True]
```

```
print(np.logical_xor(c1, c2))
print(np.logical_xor(c2, c3))
```

```
[ True False False  True  True False  True]
[False False False False False False]
```

11.2 Atributos de arrays

Los array tienen otras propiedades, que pueden explorarse apretando <TAB> en una terminal o notebook de IPython o leyendo la documentación de Numpy, o utilizando la función `dir(arr)` (donde `arr` es una variable del tipo array) o `dir(np.ndarray)`.

En la tabla se muestra una lista de los atributos de los numpy array

arr.T	arr.copy	arr.getfield	arr.put	arr.squeeze
arr.all	arr.ctypes	arr.imag	arr.ravel	arr.std
arr.any	arr.cumprod	arr.item	arr.real	arr.strides
arr.argmax	arr.cumsum	arr.itemset	arr.repeat	arr.sum
arr.argmin	arr.data	arr.itemsize	arr.reshape	arr.swapaxes
arr.argsort	arr.diagonal	arr.max	arr.resize	arr.take
arr.astype	arr.dot	arr.mean	arr.round	arr.tofile
arr.base	arr.dtype	arr.min	arr.searchsorted	arr.tolist
arr.byteswap	arr.dump	arr nbytes	arr.setasflat	arr.tostring
arr.choose	arr.dumps	arr.ndim	arr.setfield	arr.trace
arr.clip	arr.fill	arr.newbyteorder	arr.setflags	arr.transpose
arr.compress	arr.flags	arr.nonzero	arr.shape	arr.var
arr.conj	arr.flat	arr.prod	arr.size	arr.view
arr.conjugate	arr.flatten	arr.ptp	arr.sort	

Exploraremos algunas de ellas

11.2.1 reshape

```
arr= np.arange(12)                                # Vector
print("Vector original:\n", arr)
arr2= arr.reshape((3,4))                          # Le cambiamos la forma a matriz de 3x4
print("Cambiando la forma a 3x4:\n", arr2)
arr3= np.reshape(arr,(4,3))                      # Le cambiamos la forma a matriz de 4x3
print("Cambiando la forma a 4x3:\n", arr3)
```

```
Vector original:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
Cambiando la forma a 3x4:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Cambiando la forma a 4x3:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
arr2[0,0] = 5
arr2[2,1] = -9
```

```
print(arr2)
```

```
[[ 5  1  2  3]
 [ 4  5  6  7]
 [ 8 -9 10 11]]
```

```
print(arr)
```

```
[ 5  1  2  3  4  5  6  7  8 -9 10 11]
```

```
print(arr3)
```

```
[[ 5  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [-9 10 11]]
```

```
try:
    arr.reshape((3,3))    # Si la nueva forma no es adecuada, falla
except ValueError as e:
    print("Error: la nueva forma es incompatible:", e)
```

```
Error: la nueva forma es incompatible: cannot reshape array of size 12 into shape (3,
 ↵3)
```

11.2.2 transpose

```
print('Transpose:\n', arr2.T)
print('Transpose:\n', np.transpose(arr3))
```

```
Transpose:
[[ 5  4  8]
 [ 1  5 -9]
 [ 2  6 10]
 [ 3  7 11]]
Transpose:
[[ 5  3  6 -9]
 [ 1  4  7 10]
 [ 2  5  8 11]]
```

11.2.3 min, max

Las funciones para encontrar mínimo y máximo pueden aplicarse tanto a vectores como a arrays⁴ con más dimensiones. En este último caso puede elegirse si se trabaja sobre uno de los ejes:

```
print(arr2)
print(np.max(arr2))
print(np.max(arr2, axis=0))
print(np.max(arr2, axis=1))
```

```
[[ 5  1  2  3]
 [ 4  5  6  7]
 [ 8 -9 10 11]]
11
[ 8  5 10 11]
[ 5  7 11]
```

```
np.max(arr2[1, :])
```

```
7
```

El primer eje (`axis=0`) corresponde a las columnas (convención del lenguaje C), y por lo tanto dará un valor por cada columna.

Si no damos el argumento opcional `axis` ambas funciones nos darán el mínimo o máximo de todos los elementos. Si le damos un eje nos devolverá el mínimo a lo largo de ese eje.

11.2.4 argmin, argmax

Estas funciones trabajan de la misma manera que `min` y `max` pero devuelve los índices en lugar de los valores.

```
print(np.argmax(arr2))
print(np.argmax(arr2, axis=0))
print(np.argmax(arr2, axis=1))
```

```
11
[2 1 2 2]
[0 3 3]
```

11.2.5 sum, prod, mean, std

```
print(arr2)
print('sum', np.sum(arr2))
print('sum, 0', np.sum(arr2, axis=0))
print('sum, 1', np.sum(arr2, axis=1))
```

```
[[ 5  1  2  3]
 [ 4  5  6  7]
 [ 8 -9 10 11]]
sum 53
sum, 0 [17 -3 18 21]
sum, 1 [11 22 20]
```

```
print(np.prod(arr2))
print(np.prod(arr2, axis=0))
print(np.prod(arr2, axis=1))
```

```
-199584000
[160 -45 120 231]
[ 30 840 -7920]
```

```
print(arr2.mean(), '=', arr2.sum()/arr2.size)
print(np.mean(arr2, axis=0))
print(np.mean(arr2, axis=1))
print(np.std(arr2))
print(np.std(arr2, axis=1))
```

```
4.416666666666667 = 4.416666666666667
[ 5.66666667 -1.           6.           7.          ]
[2.75 5.5 5. ]
4.9742391936411305
[1.47901995 1.11803399 8.15475322]
```

11.2.6 cumsum, cumprod, trapz

Las funciones `cumsum` y `cumprod` devuelven la suma y producto acumulativo recorriendo el array, opcionalmente a lo largo de un eje

```
print(arr2)
```

```
[[ 5  1  2  3]
 [ 4  5  6  7]
 [ 8 -9 10 11]]
```

```
# Suma todos los elementos anteriores y devuelve el array unidimensional
print(arr2.cumsum())
```

```
[ 5  6  8 11 15 20 26 33 41 32 42 53]
```

```
# Para cada columna, en cada posición suma los elementos anteriores
print(arr2.cumsum(axis=0))
```

```
[[ 5  1  2  3]
 [ 9  6  8 10]
 [17 -3 18 21]]
```

```
# En cada fila, el valor es la suma de todos los elementos anteriores de la fila
print(arr2.cumsum(axis=1))
```

```
[[ 5  6  8 11]
 [ 4  9 15 22]
 [ 8 -1  9 20]]
```

```
# Igual que antes pero con el producto
print(arr2.cumprod(axis=0))
```

```
[[ 5   1   2   3]
 [20   5  12  21]
 [160 -45 120 231]]
```

La función trapz evalúa la integral a lo largo de un eje, usando la regla de los trapecios (la misma que nosotros programamos en un ejercicio)

```
print(np.trapz(arr2, axis=0))
print(np.trapz(arr2, axis=1))
```

```
[10.5  1.  12.  14. ]
 [ 7.  16.5 10.5]
```

```
# el valor por default de axis es -1
print(np.trapz(arr2))
```

```
[ 7.  16.5 10.5]
```

11.2.7 nonzero

Devuelve una *tupla* de arrays, una por dimensión, que contiene los índices de los elementos no nulos

```
# El método copy() crea un nuevo array con los mismos valores que el original
arr4 = arr2.copy()
arr4[1,:2] = arr4[2,2:] = 0
arr4
```

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

```
# Vemos que arr2 no se modifica al modificar arr4.
arr2
```

```
array([[ 5,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8, -9, 10, 11]])
```

```
np.nonzero(arr4)
```

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

```
np.transpose(arr4.nonzero())
```

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

```
arr4[arr4.nonzero()]
```

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

11.3 Convertir un array a unidimensional (ravel)

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

```
print(a)
```

```
[[1 2]  
 [3 4]]
```

```
b= np.ravel(a)
```

```
print(a.shape, b.shape)  
print(b)
```

```
(2, 2) (4,)  
[1 2 3 4]
```

```
b.base is a
```

```
True
```

ravel tiene un argumento opcional order

```
np.ravel(a, order='C') # order='C' es el default
```

```
array([1, 2, 3, 4])
```

```
np.ravel(a, order='F')
```

```
array([1, 3, 2, 4])
```

El método `flatten` hace algo muy parecido a `ravel`, la diferencia es que `flatten` siempre crea una nueva copia del array, mientras que `ravel` puede devolver una nueva vista del mismo array.

```
a.flatten()
```

```
array([1, 2, 3, 4])
```

11.4 Ejercicios 10 (a)

- Dado un array `a` de números, creado por ejemplo usando:

```
a = np.random.uniform(size=100)
```

Encontrar el número más cercano a un número escalar dado (por ejemplo `x=0.5`). Utilice los métodos discutidos.

11.5 Copias de arrays y vistas

Para poder controlar el uso de memoria y su optimización, **Numpy** no siempre crea un nuevo vector al realizar operaciones. Por ejemplo cuando seleccionamos una parte de un array usando la notación con `:` (*slicing*) devuelve algo que parece un nuevo array pero que en realidad es una nueva vista del mismo array. Lo mismo ocurre con el método `reshape`

```
x0 = np.linspace(1,24,24)
print(x0)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.
 19. 20. 21. 22. 23. 24.]
```

```
y0 = x0[::2]
print(y0)
```

```
[ 1.  3.  5.  7.  9. 11. 13. 15. 17. 19. 21. 23.]
```

El método `base` nos da acceso al objeto que tiene los datos. Por ejemplo, en este caso

```
print(x0.base)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.
 19. 20. 21. 22. 23. 24.]
```

```
print(y0.base)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.
 19. 20. 21. 22. 23. 24.]
```

Clases de Python

```
y0.base is x0.base
```

```
True
```

```
type(x0), type(y0)
```

```
(numpy.ndarray, numpy.ndarray)
```

```
y0.size, x0.size
```

```
(12, 24)
```

```
y0[1] = -1  
print(x0)
```

```
[ 1.  2. -1.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.  
 19. 20. 21. 22. 23. 24.]
```

En este ejemplo, el array `y0` está basado en `x0`, o –lo que es lo mismo– el objeto base de `y0` es `x0`. Por lo tanto, al modificar uno, se modifica el otro.

Las funciones `reshape` y `transpose` también devuelven **vistas** del array original en lugar de una nueva copia

```
x0 = np.linspace(1,24,24)  
print(x0)  
x1 = x0.reshape(6,-1)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.  
 19. 20. 21. 22. 23. 24.]
```

```
print(x1)
```

```
[[ 1.  2.  3.  4.]  
 [ 5.  6.  7.  8.]  
 [ 9. 10. 11. 12.]  
 [13. 14. 15. 16.]  
 [17. 18. 19. 20.]  
 [21. 22. 23. 24.]]
```

```
print(x1.base is x0.base)
```

```
True
```

```
x2 = x1.transpose()  
print(x2.base is x0.base)
```

```
True
```

```
x2
```

```
array([[ 1.,  5.,  9., 13., 17., 21.],
       [ 2.,  6., 10., 14., 18., 22.],
       [ 3.,  7., 11., 15., 19., 23.],
       [ 4.,  8., 12., 16., 20., 24.]])
```

Las vistas son referencias al mismo conjunto de datos, pero la información respecto al objeto puede ser diferente. Por ejemplo en el anterior `x0`, `x1` y `x` son diferentes objetos pero con los mismos datos (no sólo iguales)

```
print(x1.base is x0.base)
print(x2.base is x0.base)
print(x0.shape, x0.strides, x0.dtype)
print(x1.shape, x1.strides, x1.dtype)
print(x2.shape, x2.strides, x2.dtype)
```

```
True
True
(24,) (8,) float64
(6, 4) (32, 8) float64
(4, 6) (8, 32) float64
```

Los datos en los tres objetos están compartidos:

```
print('original')
print('x2 = ',x2)
x0[-1] = -1
print('x0 = ',x0)
```

```
original
x2 = [[ 1.  5.  9. 13. 17. 21.]
       [ 2.  6. 10. 14. 18. 22.]
       [ 3.  7. 11. 15. 19. 23.]
       [ 4.  8. 12. 16. 20. 24.]]
x0 = [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.
       19. 20. 21. 22. 23. -1.]
```

```
print('cambiado')
print('x2 = ',x2)
```

```
cambiado
x2 = [[ 1.  5.  9. 13. 17. 21.]
       [ 2.  6. 10. 14. 18. 22.]
       [ 3.  7. 11. 15. 19. 23.]
       [ 4.  8. 12. 16. 20. -1.]]
```

```
print('x1 = ',x1)
```

```
x1 = [[ 1.  2.  3.  4.]
       [ 5.  6.  7.  8.]
       [ 9. 10. 11. 12.]
       [13. 14. 15. 16.]
       [17. 18. 19. 20.]
       [21. 22. 23. -1.]]
```

11.6 Indexado avanzado

11.6.1 Indexado con secuencias de índices

Consideremos un vector simple, y elijamos algunos de sus elementos

```
x = np.linspace(0, 3, 7)
x
```

```
array([0., 0.5, 1., 1.5, 2., 2.5, 3.])
```

```
# Standard slicing
v1=x[1::2]
v1
```

```
array([0.5, 1.5, 2.5])
```

Esta es la manera simple de seleccionar elementos de un array, y como vimos lo que se obtiene es una vista del mismo array. **Numpy** permite además seleccionar partes de un array usando otro array de índices:

```
# Array Slicing con indices ind
i1 = np.array([1,3,-1,0])
v2 = x[i1]
```

```
print(x)
print(x[i1])
```

```
[0. 0.5 1. 1.5 2. 2.5 3.]
[0.5 1.5 3. 0. ]
```

```
print(v1.base is x.base)
print(v2.base is x.base)
```

```
True
False
```

```
x[[1,2,-1]]
```

```
array([0.5, 1., 3.])
```

Los índices negativos funcionan en exactamente la misma manera que en el caso simple.

Es importante notar que cuando se usan arrays índices, lo que se obtiene es un nuevo array (no una vista), y este nuevo array tiene las dimensiones (shape) del array de índices

```
i2 = np.array([[1,0],[2,1]])
v3= x[i2]
print(v3)
print('x shape:', x.shape)
print('v3 shape:', v3.shape)
```

```
[[0.5 0. ]
 [1.  0.5]]
x  shape: (2, )
v3 shape: (2,  2)
```

11.6.2 Índices de arrays multidimensionales

```
y = np.arange(12,0,-1).reshape(3,4)+0.5
y
```

```
array([[12.5, 11.5, 10.5, 9.5],
       [ 8.5,  7.5,  6.5, 5.5],
       [ 4.5,  3.5,  2.5, 1.5]])
```

```
print(y[0])                      # Primera fila
print(y[2])                      # Última fila
```

```
[12.5 11.5 10.5 9.5]
[4.5 3.5 2.5 1.5]
```

```
i = np.array([0,2])
print(y[i])                      # Primera y última fila
```

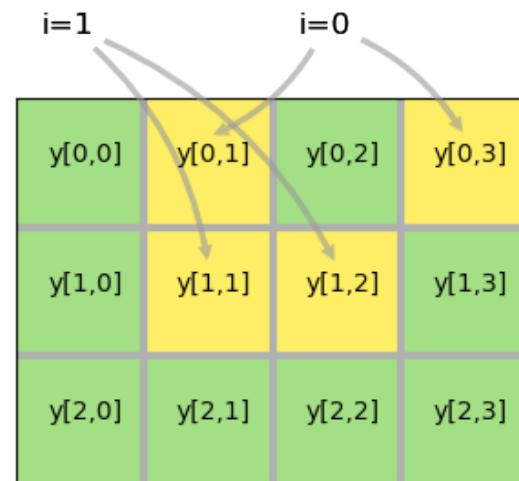
```
[[12.5 11.5 10.5 9.5]
 [ 4.5  3.5  2.5 1.5]]
```

Si usamos más de un array de índices para seleccionar elementos de un array multidimensional, cada array de índices se refiere a una dimensión diferente. Consideremos el array y

```
print(y)
```

```
[[12.5 11.5 10.5 9.5]
 [ 8.5  7.5  6.5 5.5]
 [ 4.5  3.5  2.5 1.5]]
```

12.5	11.5	10.5	9.5
8.5	7.5	6.5	5.5
4.5	3.5	2.5	1.5



Si queremos elegir los elementos en los lugares $[0, 1]$, $[1, 2]$, $[0, 3]$, $[1, 1]$ (en ese orden) podemos crear dos array de índices con los valores correspondientes a cada dimensión

```
i = np.array([0, 1, 0, 1])
j = np.array([1, 2, 3, 1])
print(y[i, j])
```

```
[11.5  6.5  9.5  7.5]
```

11.6.3 Indexado con condiciones

Además de usar notación de *slices*, e índices también podemos seleccionar partes de arrays usando una matriz de condiciones. Primero creamos una matriz de condiciones c

```
c = False*np.empty((3, 4), dtype='bool')
print(c)
```

```
[[False False False False]
 [False False False False]
 [False False False False]]
```

```
False*np.empty((3, 4))
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
c[i, j] = True # Aplico la notación de índice avanzado
print(c)
```

```
[[False True False True]
 [False True True False]
 [False False False False]]
```

Como vemos, c es una matriz con la misma forma que y. Esto permite seleccionar los valores donde el array de condiciones es verdadero:

```
y[c]
```

```
array([11.5, 9.5, 7.5, 6.5])
```

Esta es una notación potente. Por ejemplo, si en el array anterior queremos seleccionar todos los valores que sobrepasan cierto umbral (por ejemplo, los valores mayores a 7)

```
print(y)
c1 = (y > 7)
print(c1)
```

```
[[12.5 11.5 10.5 9.5]
 [ 8.5  7.5  6.5  5.5]
 [ 4.5  3.5  2.5  1.5]]
 [[ True  True  True  True]]
```

(continué en la próxima página)

(proviene de la página anterior)

```
[ True  True False False]
[False False False False]]
```

El resultado de una comparación es un array donde cada elemento es un variable lógica (True o False). Podemos utilizarlo para seleccionar los valores que cumplen la condición dada. Por ejemplo

```
y[c1]
```

```
array([12.5, 11.5, 10.5, 9.5, 8.5, 7.5])
```

De la misma manera, si queremos todos los valores entre 4 y 7 (incluidos), podemos hacer

```
y[(y >= 4) & (y <= 7)]
```

```
array([6.5, 5.5, 4.5])
```

Como mostramos en este ejemplo, no es necesario crear la matriz de condiciones previamente.

Numpy tiene funciones especiales para analizar datos de array que sirven para quedarse con los valores que cumplen ciertas condiciones. La función nonzero devuelve los índices donde el argumento no se anula:

```
c1 = (y>=4) & (y <=7)
np.nonzero(c1)
```

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

Esta es la notación de avanzada de índices, y nos dice que los elementos cuya condición es diferente de cero (True) están en las posiciones: [1,2], [1,3], [2,0].

```
indx, indy = np.nonzero(c1)
print('indx =', indx)
print('indy =', indy)
```

```
indx = [1 1 2]
indy = [2 3 0]
```

```
for i,j in zip(indx, indy):
    print('y[{},{}]={}'.format(i,j,y[i,j]))
```

```
y[1,2]=6.5
y[1,3]=5.5
y[2,0]=4.5
```

```
print(np.nonzero(c1))
print(np.transpose(np.nonzero(c1)))
print(y[np.nonzero(c1)])
```

```
(array([1, 1, 2]), array([2, 3, 0]))
[[1 2]
 [1 3]
 [2 0]]
[6.5 5.5 4.5]
```

El resultado de `nonzero()` se puede utilizar directamente para elegir los elementos con la notación de índices avanzados, y su transpuesta es un array donde cada elemento es un índice donde no se anula.

Existe la función `np.argwhere()` que es lo mismo que `np.transpose(np.nonzero(a))`.

Otra función que sirve para elegir elementos basados en alguna condición es `np.compress(condition, a, axis=None, out=None)` que acepta un array unidimensional como condición

```
c2 = np.ravel(c1)
print(c2)
print(np.compress(c2,y))
```

```
[False False False False False  True  True  True False False False]
[6.5 5.5 4.5]
```

La función `extract` es equivalente a convertir los dos vectores (condición y datos) a una dimensión (`ravel`) y luego aplicar `compress`

```
np.extract(c1, y)
```

```
array([6.5, 5.5, 4.5])
```

11.6.4 Función where

La función `where` permite operar condicionalmente sobre algunos elementos. Por ejemplo, si queremos convolucionar el vector `y` con un escalón localizado en la región `[2, 8]`:

```
np.where((y > 2) & (y < 8), y, 0)
```

```
array([[0., 0., 0., 0.],
       [0., 7.5, 6.5, 5.5],
       [4.5, 3.5, 2.5, 0.]])
```

Por ejemplo, para implementar la función de Heaviside

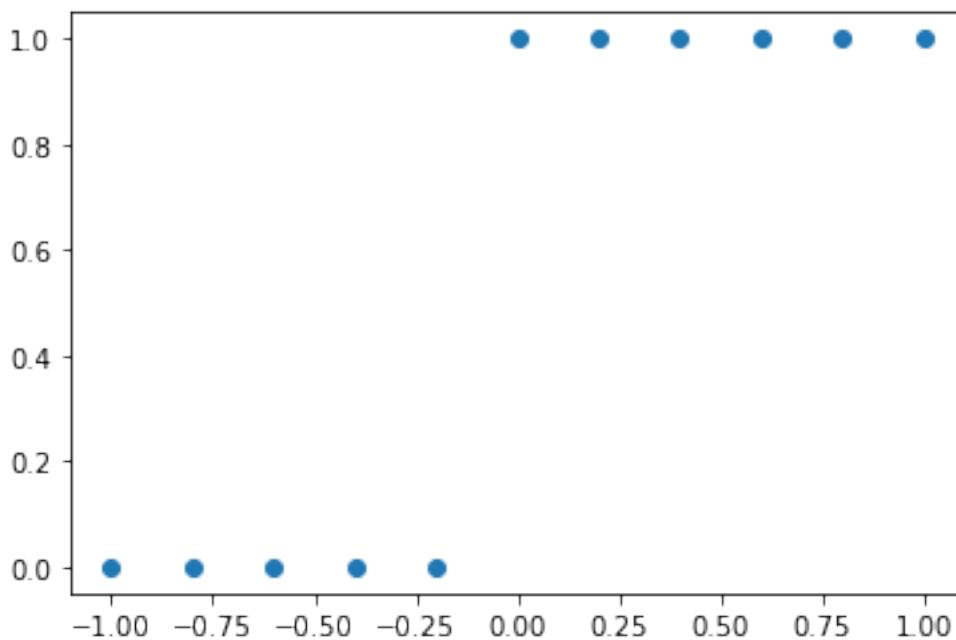
```
import matplotlib.pyplot as plt

def H(x):
    return np.where(x < 0, 0, 1)
x = np.linspace(-1,1,11)
H(x)
```

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

```
plt.plot(x,H(x), 'o')
```

```
[<matplotlib.lines.Line2D at 0x7fb489fa5a10>]
```



11.7 Extensión de las dimensiones (*Broadcasting*)

Vimos que en **Numpy** las operaciones (y comparaciones) se realizan elemento a elemento. Sin embargo usamos expresiones del tipo `y > 4` donde comparamos un ndarray con un escalar. En este caso, lo que hace **Numpy** es extender automáticamente el escalar a un array de las mismas dimensiones que `y`

```
4 -> 4*np.ones(y.shape)
```

Hagamos esto explícitamente

```
y
```

```
array([[12.5, 11.5, 10.5, 9.5],
       [8.5, 7.5, 6.5, 5.5],
       [4.5, 3.5, 2.5, 1.5]])
```

```
y4 = 4*np.ones(y.shape)
np.all((y > y4) == (y > 4)) # np.all devuelve True si **TODOS** los elementos son_
```

-iguales

```
True
```

De la misma manera, hay veces que podemos operar sobre arrays de distintas dimensiones

```
y4
```

```
array([[4., 4., 4., 4.],
       [4., 4., 4., 4.],
       [4., 4., 4., 4.]])
```

```
y + y4
```

```
array([[16.5, 15.5, 14.5, 13.5],  
       [12.5, 11.5, 10.5, 9.5],  
       [ 8.5,  7.5,  6.5,  5.5]])
```

```
y + 4
```

```
array([[16.5, 15.5, 14.5, 13.5],  
       [12.5, 11.5, 10.5, 9.5],  
       [ 8.5,  7.5,  6.5,  5.5]])
```

Como vemos eso es igual a `y + 4*np.ones(y.shape)`. En general, si Numpy puede transformar los arreglos para que todos tengan el mismo tamaño, lo hará en forma automática.

Las reglas de la extensión automática son:

1. La extensión se realiza por dimensión. Dos dimensiones son compatibles si son iguales o una de ellas es 1.
2. Si los dos arrays difieren en el número de dimensiones, el que tiene menor dimensión se llena con 1 (unos) en el primer eje.

Veamos algunos ejemplos:

```
x = np.arange(0,40,10)  
xx = x.reshape(4,1)  
y = np.arange(3)
```

```
print(x.shape, xx.shape, y.shape)
```

```
(4,) (4, 1) (3,)
```

```
print(xx)
```

```
[[ 0]  
 [10]  
 [20]  
 [30]]
```

```
print(y)
```

```
[0 1 2]
```

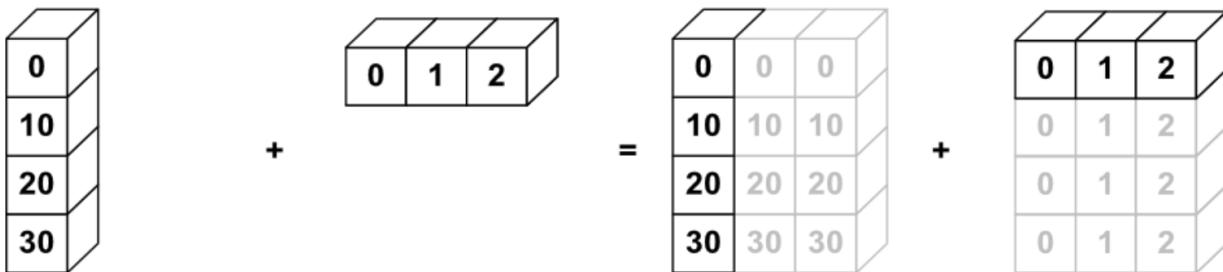
```
print(xx+y)
```

```
[[ 0  1  2]  
 [10 11 12]  
 [20 21 22]  
 [30 31 32]]
```

Lo que está pasando es algo así como:

- `xx -> xxx`
- `y ->yyy`

- $\text{xx} + \text{y} \rightarrow \text{xxx} + \text{yyy}$



donde xxx, yyy son versiones extendidas de los vectores originales:

```
xxx = np.tile(xx, (1, y.size))
yyy = np.tile(y, (xx.size, 1))
```

```
print(xxx)
```

```
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
```

```
print(yyy)
```

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

```
print(xxx + yyy)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

11.8 Unir (o concatenar) arrays

Si queremos unir dos *arrays* para formar un tercer *array* Numpy tiene una función llamada `concatenate`, que recibe una secuencia de arrays y devuelve su unión a lo largo de un eje.

11.8.1 Apilamiento vertical

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8], [9, 10]])
print('a=\n', a)
print('b=\n', b)
```

```
a=
[[1 2]
 [3 4]]
b=
[[ 5  6]
 [ 7  8]
 [ 9 10]]
```

```
# El eje 0 es el primero, y corresponde a apilamiento vertical
np.concatenate((a, b), axis=0)
```

```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

```
np.concatenate((a, b))          # axis=0 es el default
```

```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

```
np.vstack((a, b))      # Une siempre verticalmente (primer eje)
```

```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

Veamos cómo utilizar esto cuando tenemos más dimensiones.

```
c = np.array([[[1, 2], [3, 4]], [[-1,-2], [-3,-4]]])
d = np.array([[[5, 6], [7, 8]], [[9,10], [-5, -6]], [[-7, -8], [-9,-10]]])
print('c: shape={}\n'.format(c.shape), c)
print('\nd: shape={}\n'.format(d.shape), d)
```

```
c: shape=(2, 2, 2)
[[[ 1  2]
 [ 3  4]]

 [[-1 -2]
 [-3 -4]]]

d: shape=(3, 2, 2)
[[[ 5   6]
 [ 7   8]

 [[ 9 10]
 [-5 -6]]]
```

(continué en la próxima página)

(provine de la página anterior)

```
[ [ -7  -8]
 [ -9 -10] ]]
```

Como tienen todas las dimensiones iguales, excepto la primera, podemos concatenarlos a lo largo del eje 0 (verticalmente)

```
np.vstack((c,d))
```

```
array([[[ 1,  2],
       [ 3,  4]],

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

      [[ 5,  6],
       [ 7,  8]],

      [[ 9, 10],
       [-5, -6]],

      [[ -7, -8],
       [ -9, -10]]])
```

```
e=np.concatenate((c,d),axis=0)
```

```
print(e.shape)
print(e)
```

```
(5, 2, 2)
[[[ 1  2]
 [ 3  4]

 [[ -1 -2]
 [ -3 -4]]

 [[ 5  6]
 [ 7  8]]

 [[ 9 10]
 [-5 -6]]

 [[ -7 -8]
 [ -9 -10]]]]
```

11.8.2 Apilamiento horizontal

Si tratamos de concatenar `a` y `b` a lo largo de otro eje vamos a recibir un error porque la forma de los arrays no es compatible.

```
b.T
```

```
array([[ 5,  7,  9],
       [ 6,  8, 10]])
```

```
print(a.shape, b.shape, b.T.shape)
```

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

```
np.concatenate((a, b.T), axis=1)
```

```
array([[ 1,  2,  5,  7,  9],
       [ 3,  4,  6,  8, 10]])
```

```
np.hstack((a,b.T)) # Como vstack pero horizontalmente
```

```
array([[ 1,  2,  5,  7,  9],
       [ 3,  4,  6,  8, 10]])
```

11.9 Enumerate para ndarrays

Para iterables en **Python** existe la función enumerate que devuelve una tupla con el índice y el valor. En **Numpy** existe un iterador multidimensional llamado `ndenumerate()`

```
print(b)
```

```
[[ 5  6]
 [ 7  8]
 [ 9 10]]
```

```
for (i,j), x in np.ndenumerate(b):
    print('x[{},{}]->{}'.format(i,j,x))
```

```
x[0,0]->5
x[0,1]->6
x[1,0]->7
x[1,1]->8
x[2,0]->9
x[2,1]->10
```

11.10 Generación de números aleatorios

Python tiene un módulo para generar números al azar, sin embargo vamos a utilizar el módulo de **Numpy** llamado `random`. Este módulo tiene funciones para generar números al azar siguiendo varias distribuciones más comunes. Veamos qué hay en el módulo

```
dir(np.random)
```

```
['Generator',
 'MT19937',
 'PCG64',
 'Philox',
 'RandomState',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'SFC64',
'SeedSequence',
'__RandomState_ctor',
'__all__',
'__builtins__',
'__cached__',
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__path__',
'__spec__',
'_pickle',
'absolute_import',
'beta',
'binomial',
'bit_generator',
'bounded_integers',
'bytes',
'chisquare',
'choice',
'common',
'default_rng',
'dirichlet',
'division',
'exponential',
'f',
'gamma',
'generator',
'geometric',
'get_state',
'gumbel',
'hypergeometric',
'laplace',
'logistic',
'lognormal',
'logseries',
'mt19937',
'mtrand',
'multinomial',
'multivariate_normal',
'negative_binomial',
'noncentral_chisquare',
'noncentral_f',
'normal',
'pareto',
'pcg64',
'permutation',
'philox',
'poisson',
'power',
'print_function',
'rand',
'randint',
'randn',
'random',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'random_integers',
'random_sample',
'ranf',
'rayleigh',
'sample',
'seed',
'set_state',
'sfc64',
'shuffle',
'standard_cauchy',
'standard_exponential',
'standard_gamma',
'standard_normal',
'standard_t',
'test',
'triangular',
'uniform',
'venmises',
'wald',
'weibull',
'zipf']
```

11.10.1 Distribución uniforme

Si elegimos números al azar con una distribución de probabilidad uniforme, la probabilidad de que el número elegido caiga en un intervalo dado es simplemente proporcional al tamaño del intervalo.

```
x= np.random.random((4,2))
y = np.random.random(8)
print(x)
```

```
[[0.77878726 0.88352463]
 [0.09660936 0.26329854]
 [0.09022693 0.50868008]
 [0.59441451 0.83330015]]
```

```
y
```

```
array([0.96525023, 0.46979077, 0.15988329, 0.02590613, 0.71784142,
       0.03085417, 0.04947806, 0.50099472])
```

```
help(np.random.random)
```

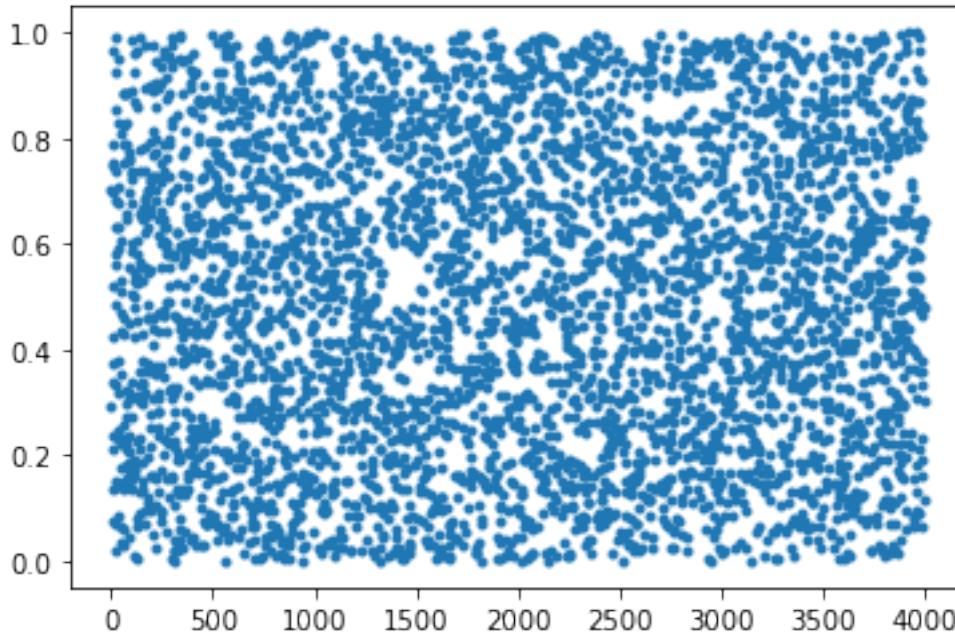
Help on built-in function random:

```
random(...) method of numpy.random.mtrand.RandomState instance
    random(size=None)
```

Return random floats in the half-open interval [0.0, 1.0). Alias for `random_sample` to ease forward-porting to the new random API.

Como se infiere de este resultado, la función `random` (o `random_sample`) nos da una distribución de puntos aleatorios entre 0 y 1, uniformemente distribuidos.

```
plt.plot(np.random.random(4000), '.')
plt.show()
```



```
help(np.random.uniform)
```

Help on built-in function uniform:

```
uniform(...) method of numpy.random.mtrand.RandomState instance
    uniform(low=0.0, high=1.0, size=None)
```

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Parameters

low : float or array_like of floats, optional

Lower boundary of the output interval. All values generated will be greater than or equal to *low*. The default value is 0.

high : float or array_like of floats

Upper boundary of the output interval. All values generated will be less than *high*. The default value is 1.0.

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., (m, n, k), then

*m * n * k* samples are drawn. If *size* is None (default),

a single value is returned if *low* and *high* are both scalars.

Otherwise, *np.broadcast(low, high).size* samples are drawn.

Returns

```
-----  
out : ndarray or scalar  
      Drawn samples from the parameterized uniform distribution.
```

See Also

```
-----  
randint : Discrete uniform distribution, yielding integers.  
random_integers : Discrete uniform distribution over the closed  
                  interval [low, high].  
random_sample : Floats uniformly distributed over [0, 1).  
random : Alias for random_sample.  
rand : Convenience function that accepts dimensions as input, e.g.,  
       rand(2,2) would generate a 2-by-2 array of floats,  
       uniformly distributed over [0, 1).
```

Notes

```
-----  
The probability density function of the uniform distribution is
```

```
.. math:: p(x) = \frac{1}{b - a}
```

anywhere within the interval [a, b), and zero elsewhere.

When high == low, values of low will be returned.

If high < low, the results are officially undefined
and may eventually raise an error, i.e. do not rely on this
function to behave when passed arguments satisfying that
inequality condition.

Examples

```
-----  
Draw samples from the distribution:
```

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)  
True  
>>> np.all(s < 0)  
True
```

Display the histogram of the samples, along with the
probability density function:

```
>>> import matplotlib.pyplot as plt  
>>> count, bins, ignored = plt.hist(s, 15, density=True)  
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')  
>>> plt.show()
```

11.10.2 Distribución normal (Gaussiana)

Una distribución de probabilidad normal tiene la forma Gaussiana

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

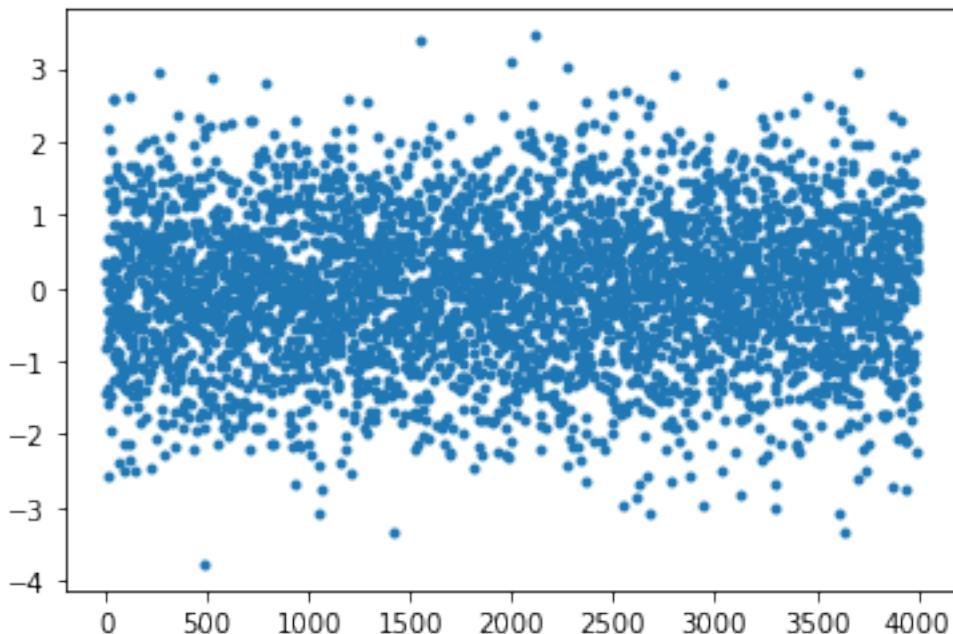
En **Numpy** la función que nos da elementos con esa distribución de probabilidad es:

```
np.random.normal(loc=0.0, scale=1.0, size=None)
```

donde: - `loc` es la posición del máximo (valor medio) - `scale` es el ancho de la distribución - `size` es el número de puntos a calcular (o forma)

```
z = np.random.normal(size=4000)
```

```
plt.plot(z, '.')
plt.show()
```



```
np.random.normal(size=(3,5))
```

```
array([[-0.50819971,  0.39541277, -0.42435484,  1.436731 , -1.6231248 ],
       [ 0.1785728 ,  0.27730798, -2.12114776,  2.27803301, -0.23488334],
      [-2.54623689,  1.30979596,  0.38841641,  0.10824249,  0.70373435]])
```

11.10.3 Histogramas

Para visualizar los números generados y comparar su ocurrencia con la distribución de probabilidad vamos a generar histogramas usando *Numpy* y *Matplotlib*

```
h,b = np.histogram(z, bins=20)
```

```
b
```

```
array([-3.77072208, -3.40965446, -3.04858684, -2.68751921, -2.32645159,
       -1.96538397, -1.60431635, -1.24324873, -0.88218111, -0.52111349,
       -0.16004587,  0.20102175,  0.56208937,  0.92315699,  1.28422461,
       1.64529223,  2.00635985,  2.36742747,  2.72849509,  3.08956271,
       3.45063034])
```

```
h
```

```
array([ 1,  5,  9, 22, 64, 128, 213, 302, 397, 516, 587, 567, 481,
       304, 223, 107, 48, 16,   8,   2])
```

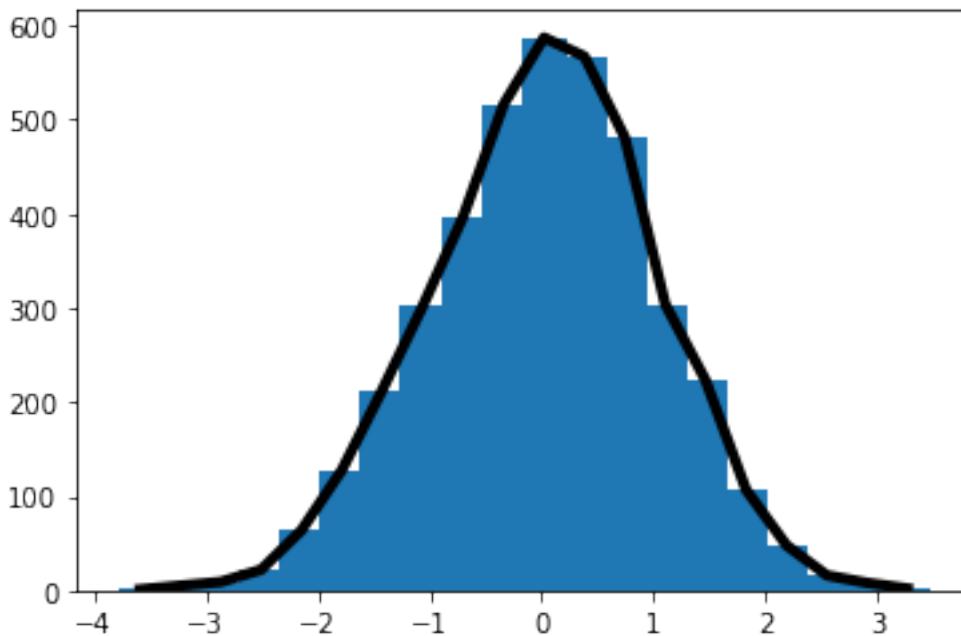
```
b.size, h.size
```

```
(21, 20)
```

La función retorna b: los límites de los intervalos en el eje x y h las alturas

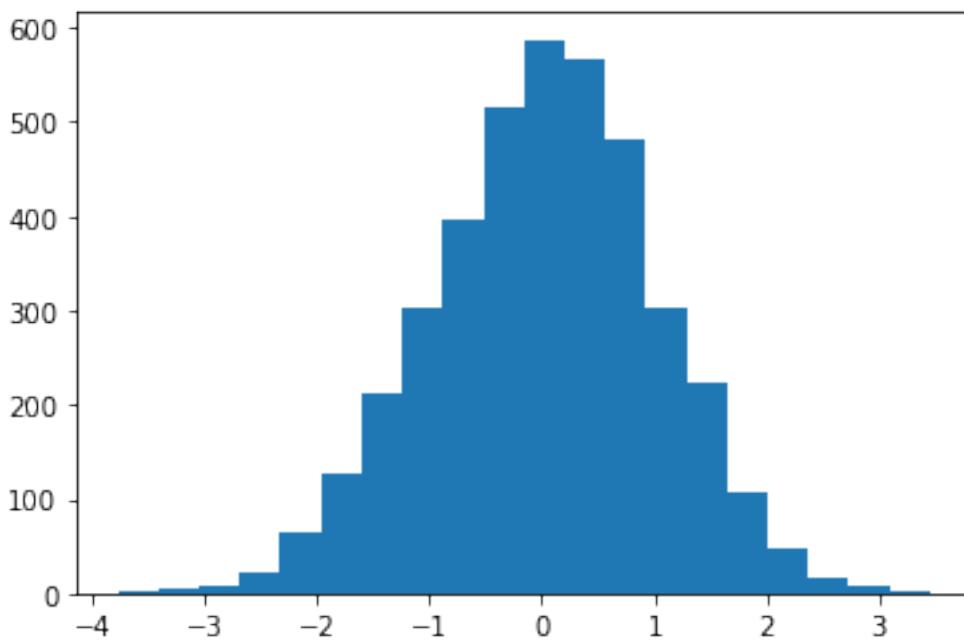
```
x = (b[1:] + b[:-1])/2
```

```
plt.bar(x,h, align="center", width=0.4)
plt.plot(x,h, 'k', lw=4)
plt.show()
```



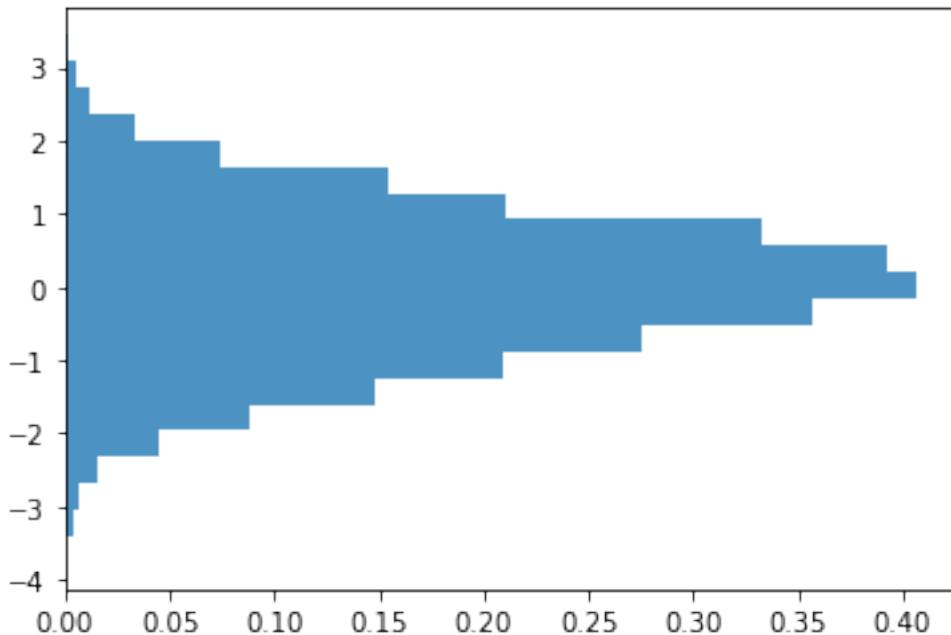
Matplotlib tiene una función similar, que directamente realiza el gráfico

```
h1, b1, p1 = plt.hist(z, bins=20)
#x1 = (b1[:-1] + b1[1:])/2
#plt.plot(x1, h1, '-k', lw=4)
plt.show()
```



Veamos otro ejemplo, agregando algún otro argumento opcional

```
plt.hist(z, bins=20, density=True, orientation='horizontal', alpha=0.8, histtype=
    'stepfilled')
plt.show()
```



En este último ejemplo, cambiamos la orientación a `horizontal` y además normalizamos los resultados, de manera tal que la integral bajo (a la izquierda de, en este caso) la curva sea igual a 1.

11.10.4 Distribución binomial

Cuando ocurre un evento que puede tener sólo dos resultados (verdadero, con probabilidad p , y falso con probabilidad $(1 - p)$) y lo repetimos N veces, la probabilidad de obtener el resultado con probabilidad p es

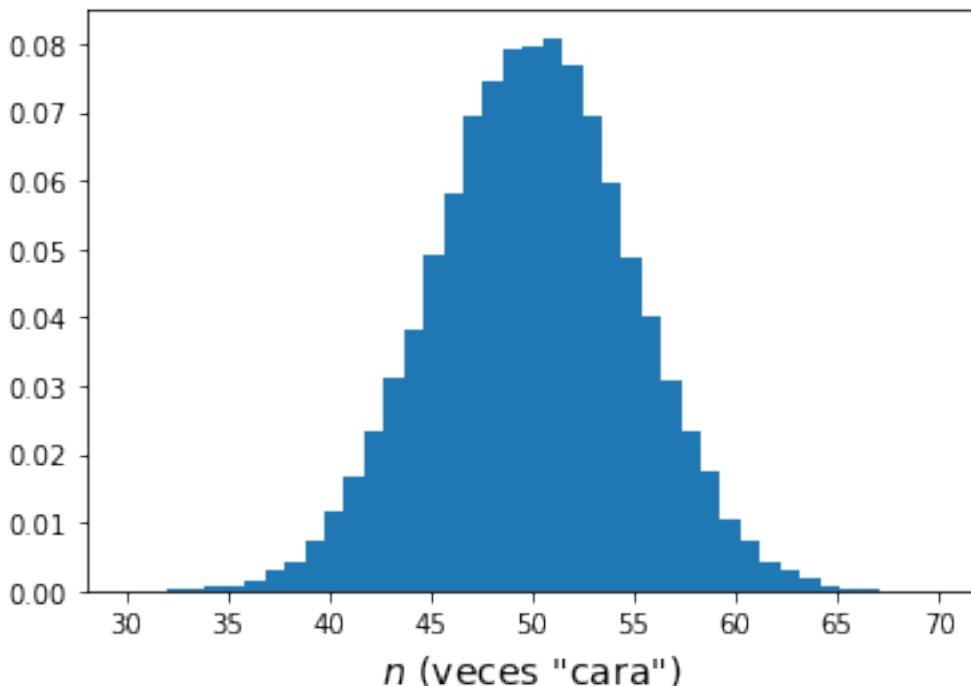
$$P(n) = \binom{N}{n} p^n (1 - P)^{N-n},$$

Para elegir números al azar con esta distribución de probabilidad **Numpy** tiene la función `binomial`, cuyo primer argumento es N y el segundo p . Por ejemplo si tiramos una moneda 100 veces, y queremos saber cuál es la probabilidad de obtener cara n veces podemos usar:

```
zb = np.random.binomial(100, 0.5, size=30000)
```

```
plt.hist(zb, bins=41, density=True, range=(30,70))
plt.xlabel('$n$ (veces "cara")')
```

```
Text(0.5, 0, '$n$ (veces "cara")')
```



```
help(np.random.binomial)
```

Help on built-in function binomial:

```
binomial(...) method of numpy.random.mtrand.RandomState instance
    binomial(n, p, size=None)
```

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer ≥ 0 and p is in the interval $[0,1]$. (n may be

```
input as a float, but it is truncated to an integer in use)

Parameters
-----
n : int or array_like of ints
    Parameter of the distribution,  $\geq 0$ . Floats are also accepted,
    but they will be truncated to integers.
p : float or array_like of floats
    Parameter of the distribution,  $\geq 0$  and  $\leq 1$ .
size : int or tuple of ints, optional
    Output shape. If the given shape is, e.g.,  $(m, n, k)$ , then
     $m * n * k$  samples are drawn. If size is None (default),
    a single value is returned if n and p are both scalars.
    Otherwise, np.broadcast(n, p).size samples are drawn.

Returns
-----
out : ndarray or scalar
    Drawn samples from the parameterized binomial distribution, where
    each sample is equal to the number of successes over the n trials.

See Also
-----
scipy.stats.binom : probability density function, distribution or
    cumulative density function, etc.

Notes
-----
The probability density for the binomial distribution is


$$\text{.. math:: } P(N) = \text{binom}\{n\}\{N\}p^N(1-p)^{n-N},$$


where  $n$  is the number of trials,  $p$  is the probability
of success, and  $N$  is the number of successes.

When estimating the standard error of a proportion in a population by
using a random sample, the normal distribution works well unless the
product  $p*n \leq 5$ , where  $p$  = population proportion estimate, and  $n$  =
number of samples, in which case the binomial distribution is used
instead. For example, a sample of 15 people shows 4 who are left
handed, and 11 who are right handed. Then  $p = 4/15 = 27\%.$   $0.27*15 = 4$ ,
so the binomial distribution should be used in this case.

References
-----
.. [1] Dalgaard, Peter, "Introductory Statistics with R",
    Springer-Verlag, 2002.
.. [2] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill,
    Fifth Edition, 2002.
.. [3] Lentner, Marvin, "Elementary Applied Statistics", Bogden
    and Quigley, 1972.
.. [4] Weisstein, Eric W. "Binomial Distribution." From MathWorld--A
    Wolfram Web Resource.
    http://mathworld.wolfram.com/BinomialDistribution.html
```

```
.. [5] Wikipedia, "Binomial distribution",
      https://en.wikipedia.org/wiki/Binomial_distribution
```

Examples

Draw samples from the distribution:

```
>>> n, p = 10, .5 # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

Este gráfico ilustra la probabilidad de obtener n veces un lado (cara) si tiramos 100 veces una moneda, como función de n .

11.11 Ejercicios de Clase 10

2. Vamos a estudiar la frecuencia de aparición de cada dígito en la serie de Fibonacci, generada siguiendo las reglas:

$$a_1 = a_2 = 1, \quad a_i = a_{i-1} + a_{i-2}.$$

Se pide:

1. Crear una función que acepta como argumento un número entero N y retorna una secuencia (lista, tupla, diccionario o *array*) con los elementos de la serie de Fibonacci.
2. Crear una función que devuelva un histograma de ocurrencia de cada uno de los dígitos en el primer lugar del número. Por ejemplo para los primeros 8 valores ($N = 8$): 1, 1, 2, 3, 5, 8, 13, 21 tendremos que el 1 aparece 3 veces, el 2 aparece 2 veces, 3, 5, 8 una vez. Normalizar los datos dividiendo por el número de valores N .
3. Utilizando las dos funciones anteriores graficar el histograma para un número N grande y comparar los resultados con la ley de Benford

$$P(n) = \log_{10} \left(1 + \frac{1}{d} \right).$$

4. **PARA ENTREGAR:** Estimar el valor de π usando diferentes métodos basados en el método de Monte Carlo:

1. Crear una función para calcular el valor de π usando el método de cociente de áreas. Para ello:
 - Generar puntos en el plano dentro del cuadrado de lado unidad cuyo lado inferior va de $x = 0$ a $x = 1$
 - Contar cuantos puntos caen dentro del (cuarto de) círculo unidad. Este número tiende a ser proporcional al área del círculo

- La estimación de π será igual a cuatro veces el cociente de números dentro del círculo dividido por el número total de puntos.
2. Crear una función para calcular el valor de π usando el método del valor medio: Este método se basa en la idea de que el valor medio de una función se puede calcular de la siguiente manera:

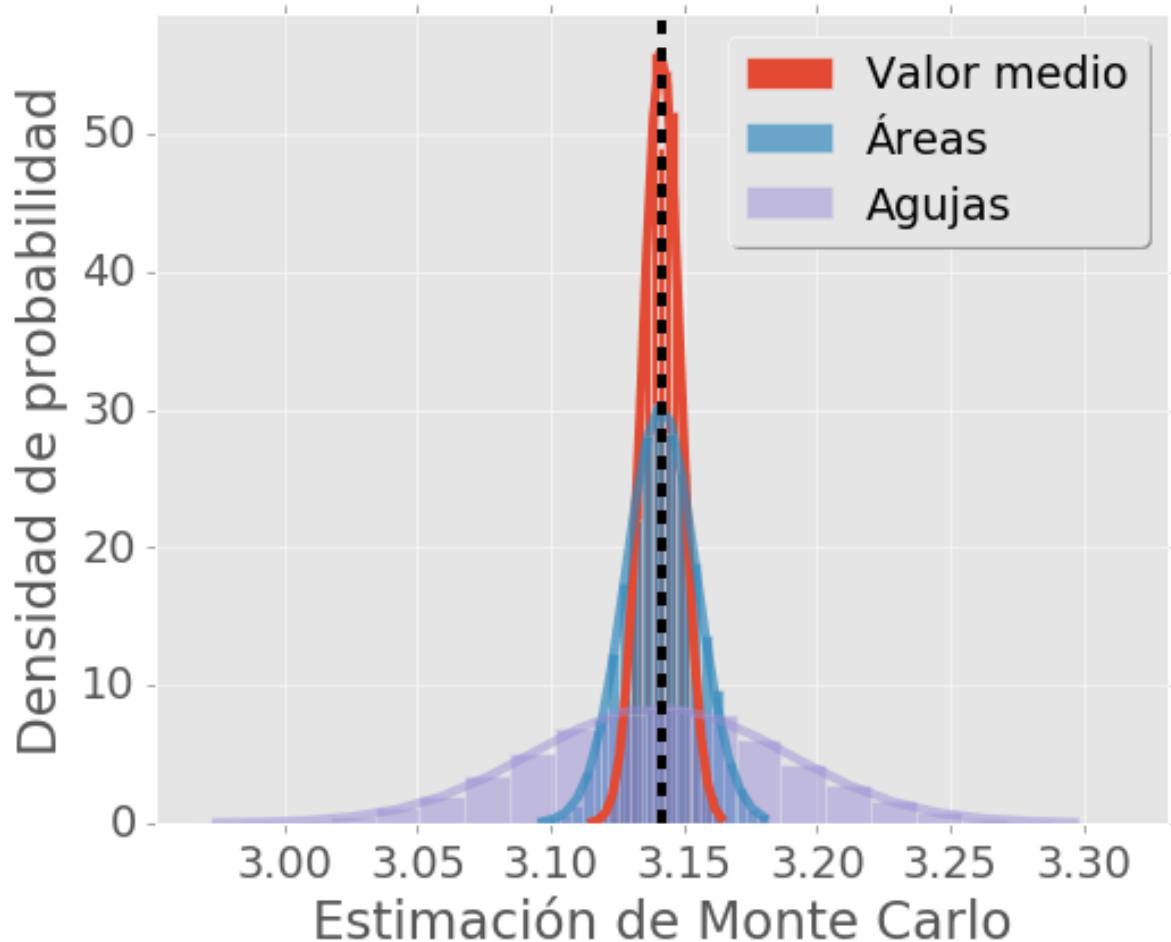
$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) dx$$

Tomando la función particular $f(x) = \sqrt{1-x^2}$ entre $x = 0$ y $x = 1$, obtenemos:

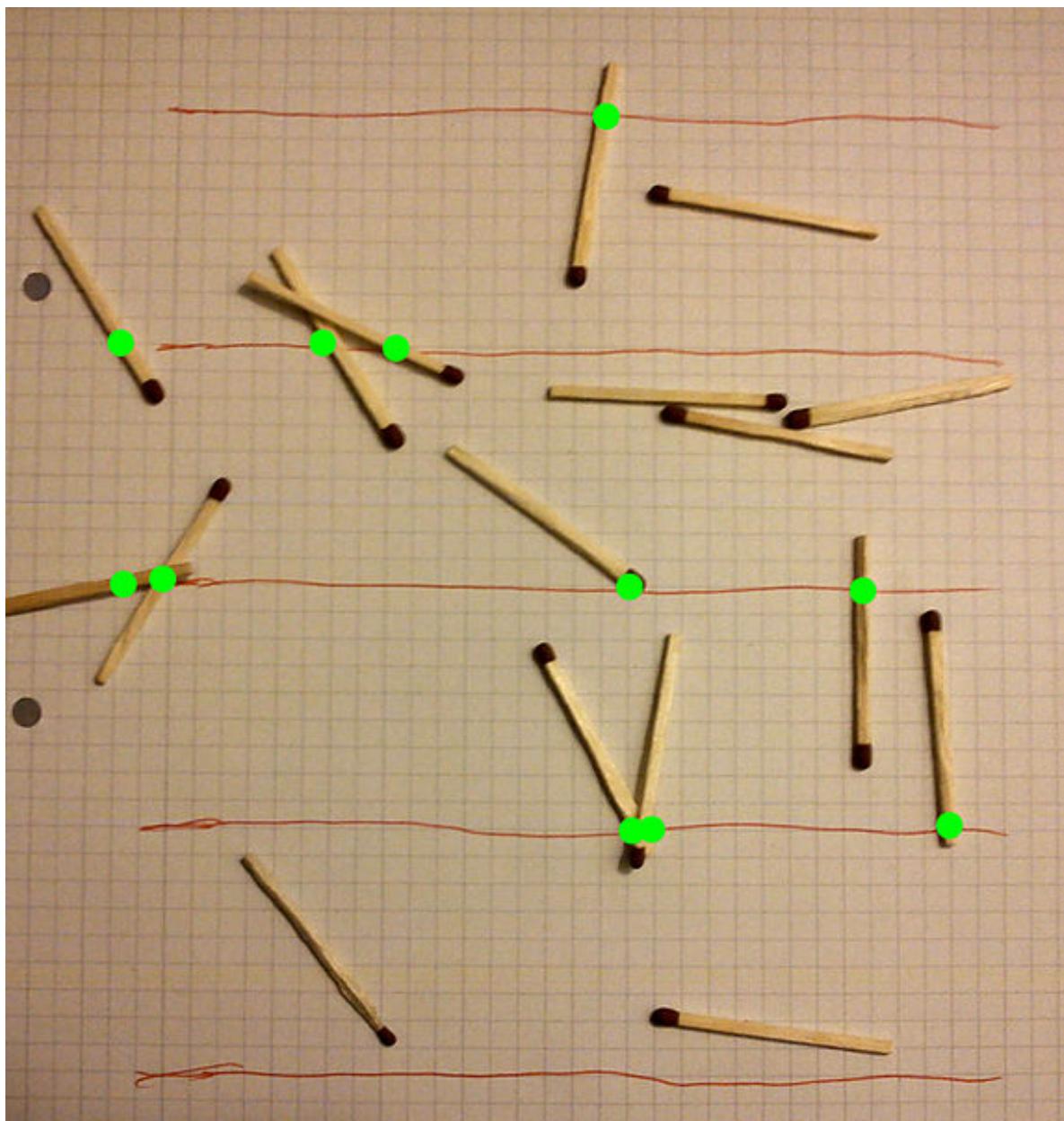
$$\langle f \rangle = \int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$

Entonces, tenemos que estimar el valor medio de la función f y, mediante la relación anterior obtener $\pi = 4\langle f(x) \rangle$. Para obtener el valor medio de la función notamos que si tomamos X es una variable aleatoria entre 0 y 1, entonces el valor medio de $f(X)$ es justamente $\langle f \rangle$. Su función debe entonces

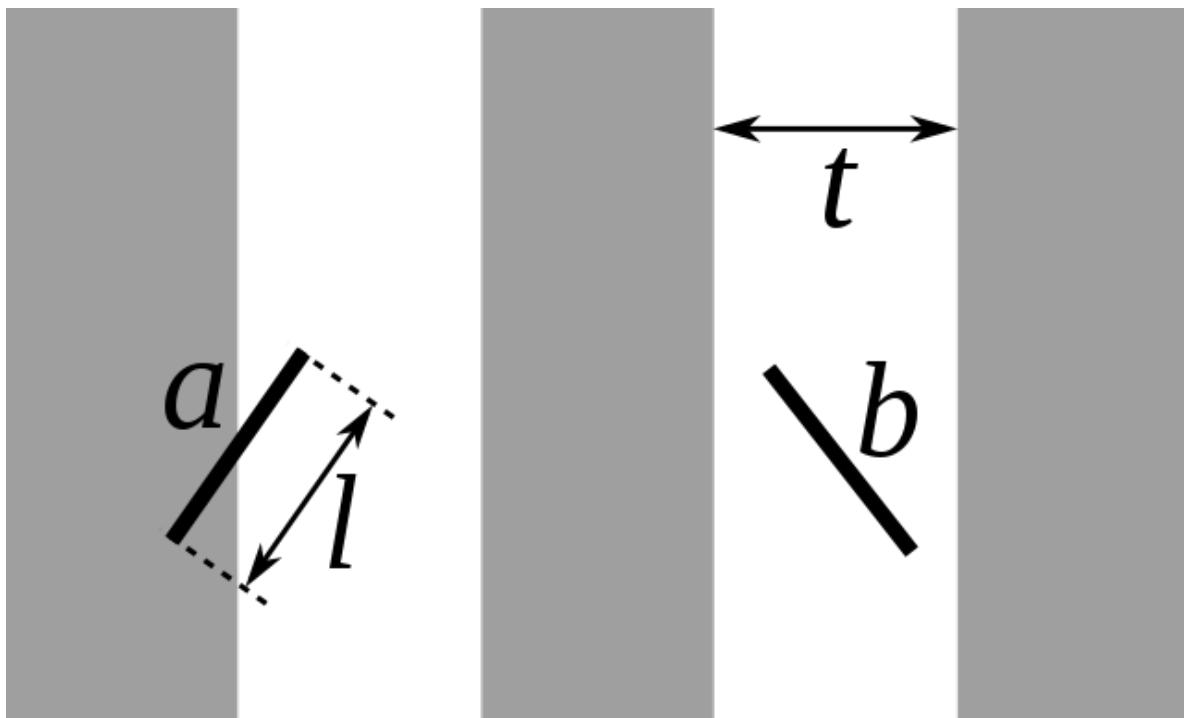
- Generar puntos aleatoriamente en el intervalo $[0, 1]$
 - Calcular el valor medio de $f(x)$ para los puntos aleatorios x .
 - El resultado va a ser igual al valor de la integral, y por lo tanto a $\pi/4$.
3. Utilizar las funciones anteriores con diferentes valores para el número total de puntos N . En particular, hacerlo para 20 valores de N equiespaciados logarítmicamente entre 100 y 10000. Para cada valor de N calcular la estimación de π . Realizar un gráfico con el valor estimado como función del número N con los dos métodos (dos curvas en un solo gráfico)
4. Para $N = 15000$ repetir el experimento muchas veces (al menos 1000) y realizar un histograma de los valores obtenidos para π con cada método. Graficar el histograma y calcular la desviación standard. Superponer una función Gaussiana con el mismo ancho. El gráfico debe ser similar al siguiente (*el estilo de graficación no tiene que ser el mismo*)



5. El método de la aguja del bufón se puede utilizar para estimar el valor de π , y consiste en tirar agujas (o palitos, fósforos, etc) al azar sobre una superficie rayada



Por simplicidad vamos a considerar que la distancia entre rayas t es mayor que la longitud de las agujas ℓ



La probabilidad de que una aguja cruce una línea será:

$$P = \frac{2\ell}{t\pi}$$

por lo que podemos calcular el valor de π si estimamos la probabilidad P . Realizar una función que estime π utilizando este método y repetir las comparaciones de los dos puntos anteriores pero ahora utilizando este método y el de las áreas.

CAPÍTULO 12

Clase 11: Introducción al paquete Scipy

El paquete **Scipy** es una colección de algoritmos y funciones construida sobre **Numpy** para facilitar cálculos y actividades relacionadas con el trabajo técnico/científico.

12.1 Una mirada rápida a Scipy

La ayuda de `scipy` contiene (con `help(scipy)` entre otras cosas)

```
Contents
-----
SciPy imports all the functions from the NumPy namespace, and in
addition provides:

Subpackages
-----
Using any of these subpackages requires an explicit import. For example,
``import scipy.cluster``.

::

cluster                  --- Vector Quantization / Kmeans
fftpack                  --- Discrete Fourier Transform algorithms
integrate                --- Integration routines
interpolate              --- Interpolation Tools
io                       --- Data input and output
linalg                   --- Linear algebra routines
linalg.blas               --- Wrappers to BLAS library
linalg.lapack              --- Wrappers to LAPACK library
misc                      --- Various utilities that don't have
                           another home.
ndimage                  --- n-dimensional image package
odr                      --- Orthogonal Distance Regression
optimize                 --- Optimization Tools
```

(continué en la próxima página)

(proviene de la página anterior)

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	--- Sparse Eigenvalue Solvers
sparse.linalg.eigen.lobpcg	--- Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)
spatial	--- Spatial data structures and algorithms
special	--- Special functions
stats	--- Statistical Functions

Más información puede encontrarse en la [documentación oficial de Scipy](#)

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

12.2 Funciones especiales

En el submódulo `scipy.special` están definidas un número de funciones especiales. Una lista general de las funciones definidas (De cada tipo hay varias funciones) es:

- Airy functions
- Elliptic Functions and Integrals
- Bessel Functions
- Struve Functions
- Raw Statistical Functions
- Information Theory Functions
- Gamma and Related Functions
- Error Function and Fresnel Integrals
- Legendre Functions
- Ellipsoidal Harmonics
- Orthogonal polynomials
- Hypergeometric Functions
- Parabolic Cylinder Functions
- Mathieu and Related Functions
- Spheroidal Wave Functions
- Kelvin Functions
- Combinatorics
- Other Special Functions
- Convenience Functions

```
from scipy import special
```

12.2.1 Funciones de Bessel

Las funciones de Bessel son soluciones de la ecuación diferencial:

$$x^2 \frac{d^2y}{dx^2} + x \frac{dy}{dx} + (x^2 - \nu^2)y = 0.$$

Para valores enteros de ν se trata de una familia de funciones que aparecen como soluciones de problemas de propagación de ondas en problemas con simetría cilíndrica.

```
np.info(special.jv)
```

```
jv(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K',  
    dtype=None, subok=True[, signature, extobj])
```

`jv(v, z)`

Bessel function of the first kind of real order and complex argument.

Parameters

`v` : array_like
 Order (float).

`z` : array_like
 Argument (float or complex).

Returns

`J` : ndarray
 Value of the Bessel function, $J_v(z)$.

Notes

For positive v values, the computation is carried out using the AMOS [1]_ `zbesj` routine, which exploits the connection to the modified Bessel function I_v ,

```
.. math:::  
    J_v(z) = \exp(v\pi i/2) I_v(-imath z) qquad (\text{Im } z > 0)  
  
    J_v(z) = \exp(-v\pi i/2) I_v(imath z) qquad (\text{Im } z < 0)
```

For negative v values the formula,

```
.. math::: J_{-v}(z) = J_v(z) \cos(\pi v) - Y_v(z) \sin(\pi v)
```

is used, where $Y_v(z)$ is the Bessel function of the second kind, computed using the AMOS routine `zbesy`. Note that the second term is exactly zero for integer v ; to improve accuracy the second term is explicitly omitted for v values such that $v = \text{floor}(v)$.

Not to be confused with the spherical Bessel functions (see `spherical_jn`).

See also

jve : J_v with leading exponential behavior stripped off.
spherical_jn : spherical Bessel functions.

References

.. [1] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions
of a Complex Argument and Nonnegative Order",
<http://netlib.org/amos/>

```
np.info(special.jn_zeros)
```

jn_zeros(n, nt)

Compute zeros of integer-order Bessel function $J_n(x)$.

Parameters

n : int
 Order of Bessel function
nt : int
 Number of zeros to **return**

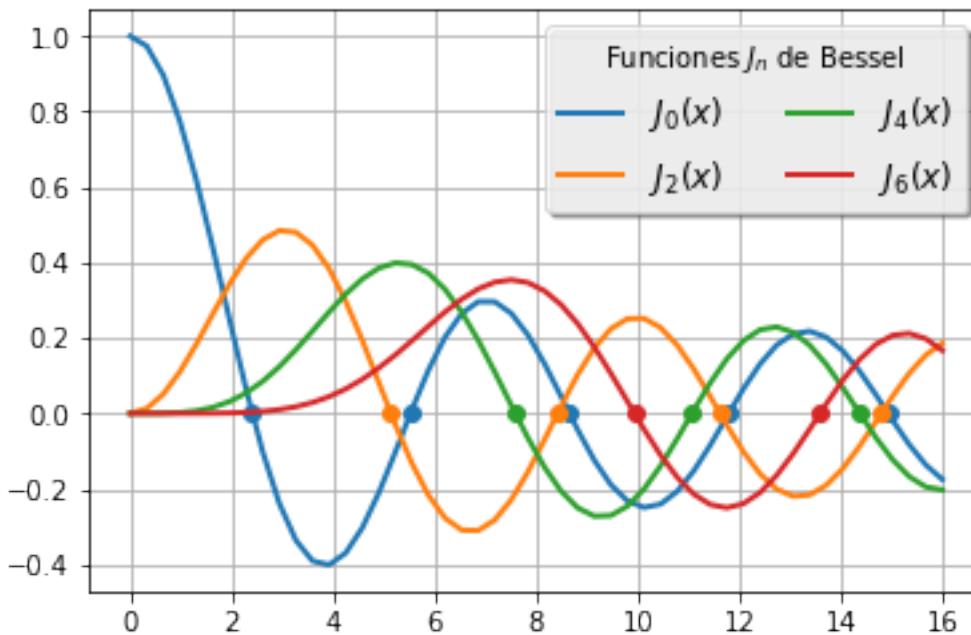
References

.. [1] Zhang, Shanjie **and** Jin, Jianming. "Computation of Special
Functions", John Wiley and Sons, 1996, chapter 5.
https://people.sc.fsu.edu/~jburkardt/f_src/special_functions/special_functions.html

```
# Ceros de la función de Bessel  
# Los tres primeros valores de x en los cuales se anula la función de Bessel de orden  
# 4.  
special.jn_zeros(4, 3)
```

```
array([ 7.58834243, 11.06470949, 14.37253667])
```

```
x = np.linspace(0, 16, 50)  
for n in range(0, 8, 2):  
    p = plt.plot(x, special.jn(n, x), label='$J_{\{n\}}(x)$'.format(n))  
    z = special.jn_zeros(n, 6)  
    z = z[z < 15]  
    plt.plot(z, np.zeros(z.size), 'o', color=p[0].get_color())  
  
plt.legend(title='Funciones $J_n$ de Bessel', ncol=2);  
plt.grid(True)
```



```
# jn es otro nombre para jv
print(special.jn == special.jv)
print(special.jn is special.jv)
```

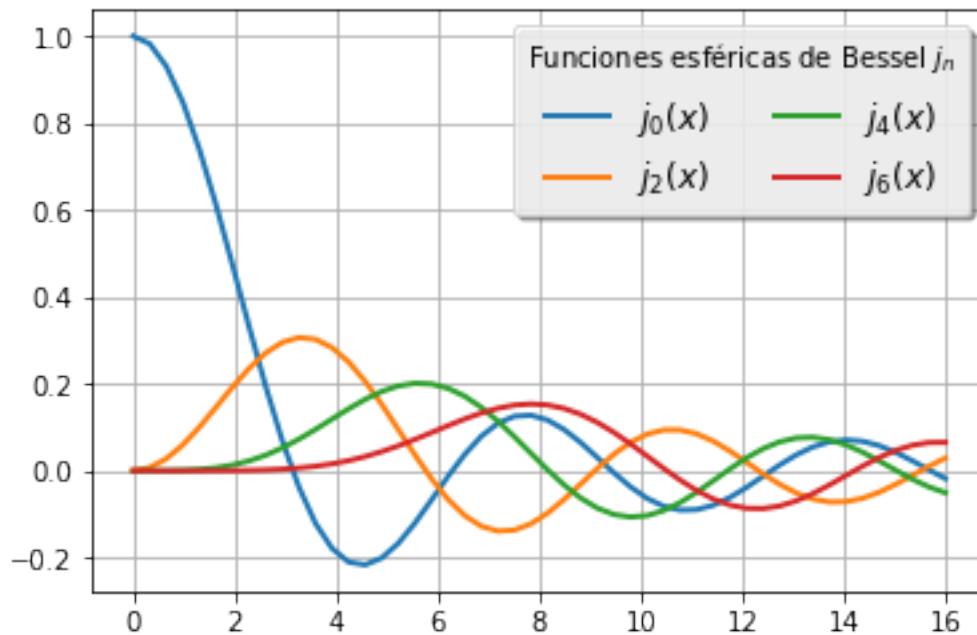
```
True
True
```

Como vemos, hay funciones para calcular funciones de Bessel. Aquí mostramos los órdenes enteros pero también se pueden utilizar órdenes ν reales. La lista de funciones de Bessel (puede obtenerse de la ayuda sobre `scipy.special`) es:

- Bessel Functions
- Zeros of Bessel Functions
- Faster versions of common Bessel Functions
- Integrals of Bessel Functions
- Derivatives of Bessel Functions
- Spherical Bessel Functions
- Riccati-Bessel Functions

Por ejemplo, podemos calcular las funciones esféricas de Bessel, que aparecen en problemas con simetría esférica:

```
x = np.linspace(0, 16, 50)
for n in range(0,7,2):
    p= plt.plot(x, special.spherical_jn(n, x), label='$j_{\{n\}}(x)$'.format(n))
plt.legend(title='Funciones esféricas de Bessel $j_n$', ncol=2);
plt.grid(True)
```



12.2.2 Función Error

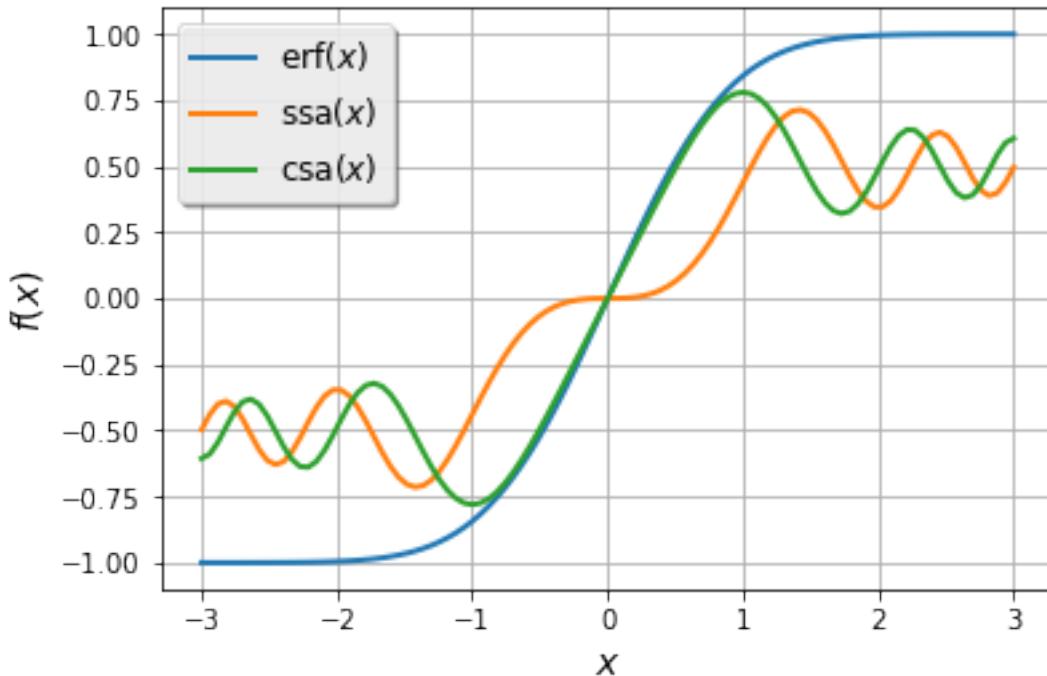
La función error es el resultado de integrar una función Gaussiana

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt,$$

mientras que las integrales seno y coseno de Fresnel están definidas por:

$$\begin{aligned} \text{ssa} &= \int_0^z \sin(\pi/2t^2) dt \\ \text{csa} &= \int_0^z \cos(\pi/2t^2) dt \end{aligned}$$

```
x = np.linspace(-3, 3, 100)
f = special.fresnel(x)
plt.plot(x, special.erf(x), '--', label=r'$\operatorname{erf}(x)$')
plt.plot(x, f[0], '--', label=r'$\operatorname{ssa}(x)$')
plt.plot(x, f[1], '--', label=r'$\operatorname{csa}(x)$')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.legend(loc='best')
plt.grid(True)
```



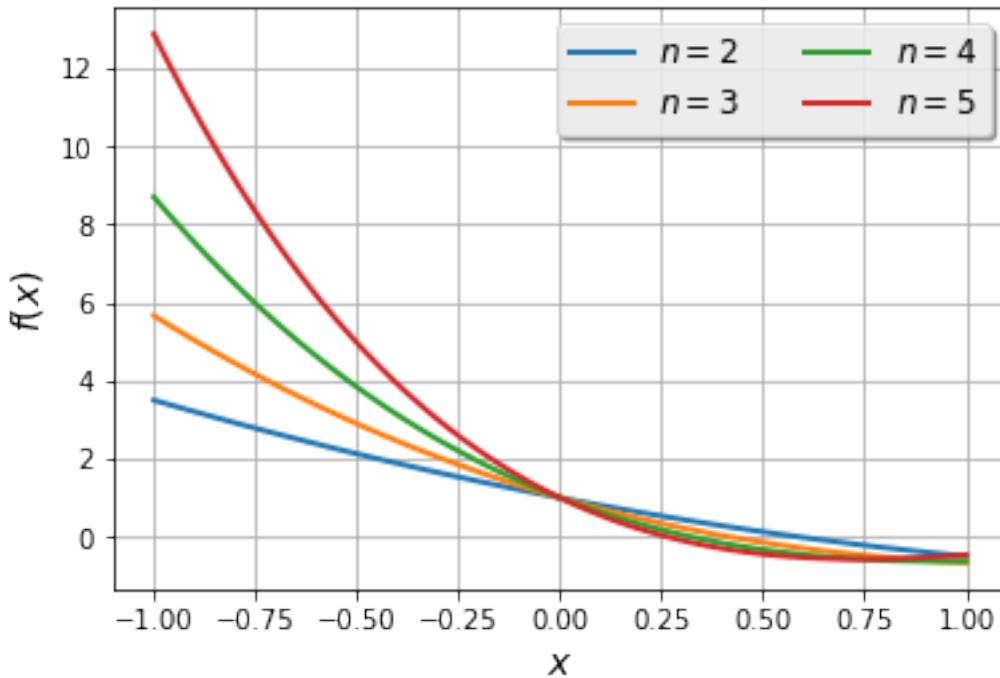
12.2.3 Evaluación de polinomios ortogonales

`Scipy.special` tiene funciones para evaluar eficientemente polinomios ortogonales

Por ejemplo si queremos, evaluar los polinomios de Laguerre, solución de la ecuación diferencial:

$$x \frac{d^2}{dx^2} L_n + (1-x) \frac{d}{dx} L_n + nL_n = 0$$

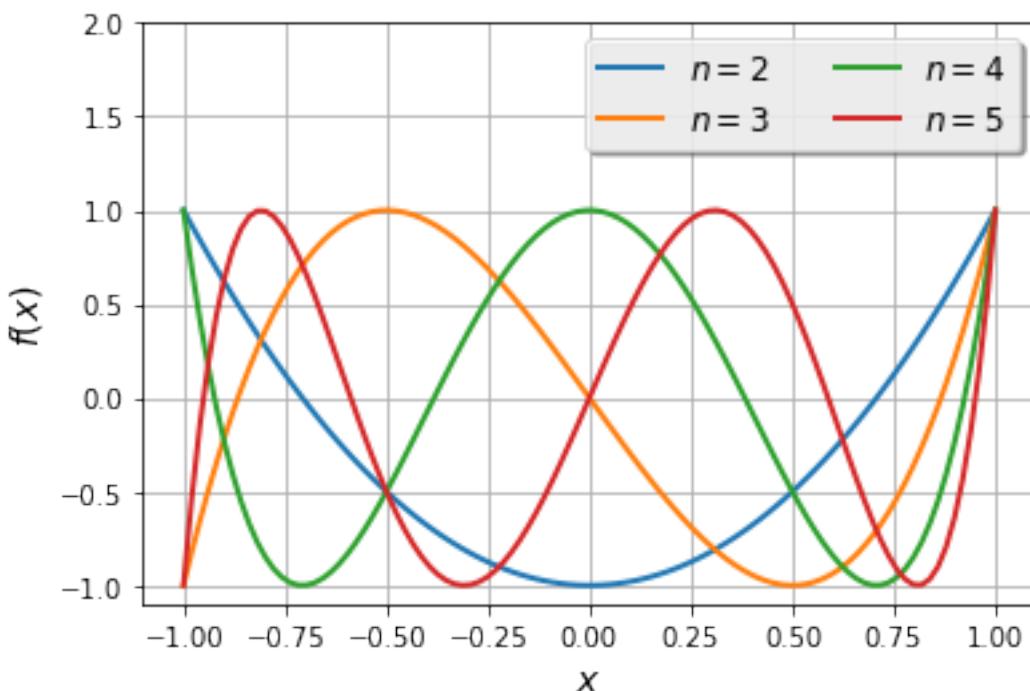
```
x = np.linspace(-1, 1, 100)
for n in range(2, 6):
    plt.plot(x, special.eval_laguerre(n, x), '-',
              label=r'$n={}$'.format(n))
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.legend(loc='best', ncol=2)
plt.grid(True)
```



Los polinomios de Chebyshev son solución de

$$(1-x^2) \frac{d^2}{dx^2} T_n - x \frac{d}{dx} T_n + n^2 T_n = 0$$

```
x = np.linspace(-1, 1, 100)
for n in range(2, 6):
    plt.plot(x, special.eval_chebyt(n, x), '-',
              label=r'$n={}$'.format(n))
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.legend(loc='best', ncol=2)
plt.ylim((-1.1,2))
plt.grid(True)
```



12.2.4 Factorial, permutaciones y combinaciones

Hay funciones para calcular varias funciones relacionadas con combinatoria

La función `comb()` da el número de maneras de elegir k de un total de N elementos. Sin repeticiones está dada por:

$$\frac{N!}{k!(N-k)!}$$

mientras que si cada elemento puede repetirse, la fórmula es:

$$\frac{(N+k-1)!}{k!(N-1)!}$$

```
N = 10
k = np.arange(2, 4)
```

```
special.comb(N, k)
```

```
array([ 45., 120.])
```

```
# Si usamos exact=True, k no puede ser un array
special.comb(N, 3, exact=True)
```

```
120
```

```
special.comb(N, k, repetition=True)
```

```
array([ 55., 220.])
```

El número de permutaciones se obtiene con la función `perm()`, y está dado por:

$$\frac{N!}{(N - k)!}$$

```
special.perm(N, k)
```

```
array([ 90., 720.])
```

que corresponde a:

$$\frac{10!}{(10 - 3)!} = 10 \cdot 9 \cdot 8$$

Los números factorial ($N!$) y doble factorial ($N!!$) son:

```
N= np.array([3,6,8])
print("{}! = {}".format(N, special.factorial(N)))
print("{}!! = {}".format(N, special.factorial2(N)))
```

```
[3 6 8]! = [6.000e+00 7.200e+02 4.032e+04]
[3 6 8]!! = [ 3.  48. 384.]
```

12.3 Integración numérica

Scipy tiene rutinas para integrar numéricamente funciones o tablas de datos. Por ejemplo para integrar funciones en la forma:

$$I = \int_a^b f(x) dx$$

la función más utilizada es `quad`, que llama a distintas rutinas del paquete **QUADPACK** dependiendo de los argumentos que toma. Entre los aspectos más notables está la posibilidad de elegir una función de peso entre un conjunto definido de funciones, y la posibilidad de elegir un dominio de integración finito o infinito.

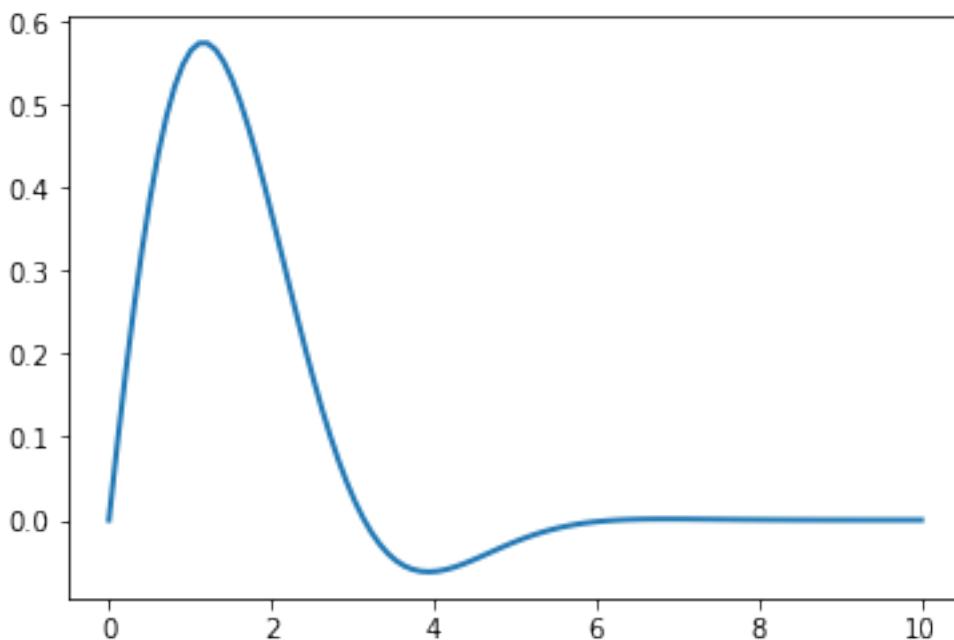
```
from scipy import integrate
```

```
x = np.linspace(0., 10, 100)
```

```
def f1(x):
    return np.sin(x)*np.exp(-np.square(x+1)/10)
```

```
plt.plot(x,f1(x))
```

```
[<matplotlib.lines.Line2D at 0x7f9293d0c390>]
```



```
integrate.quad(f1, 0, 1)
```

```
(0.34858491873298725, 3.870070028144515e-15)
```

```
np.info(integrate.quad)
```

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08,
      limit=50, points=None, weight=None, wvar=None, wopts=None, maxp1=50,
      limlst=50)
```

Compute a definite integral.

Integrate func from a to b (possibly infinite interval) using a technique from the Fortran library QUADPACK.

Parameters

`func : {function, scipy.LowLevelCallable}`

A Python function or method to integrate. If `func` takes many arguments, it is integrated along the axis corresponding to the first argument.

If the user desires improved integration performance, then `f` may be a `scipy.LowLevelCallable` with one of the signatures::

```
double func(double x)
double func(double x, void *user_data)
double func(int n, double *xx)
double func(int n, double *xx, void *user_data)
```

The ```user_data``` is the data contained in the `scipy.LowLevelCallable`. In the call forms with ```xx```, ```n``` is the length of the ```xx```

array which contains ``xx[0] == x`` and the rest of the items are numbers contained in the ``args`` argument of quad.

In addition, certain ctypes call signatures are supported for backward compatibility, but those should not be used in new code.

`a : float`

Lower limit of integration (use `-numpy.inf` for `-infinity`).

`b : float`

Upper limit of integration (use `numpy.inf` for `+infinity`).

`args : tuple, optional`

Extra arguments to pass to `func`.

`full_output : int, optional`

Non-zero to return a dictionary of integration information.

If non-zero, warning messages are also suppressed and the message is appended to the output tuple.

`Returns`

`y : float`

The integral of func from `a` to `b`.

`abserr : float`

An estimate of the absolute error in the result.

`infodict : dict`

A dictionary containing additional information.

Run `scipy.integrate.quad_explain()` for more information.

`message`

A convergence message.

`explain`

Appended only with 'cos' or 'sin' weighting and infinite integration limits, it contains an explanation of the codes in `infodict['ierlst']`

`Other Parameters`

`epsabs : float or int, optional`

Absolute error tolerance.

`epsrel : float or int, optional`

Relative error tolerance.

`limit : float or int, optional`

An upper bound on the number of subintervals used in the adaptive algorithm.

`points : (sequence of floats,ints), optional`

A sequence of break points in the bounded integration interval where local difficulties of the integrand may occur (e.g., singularities, discontinuities). The sequence does not have to be sorted.

`weight : float or int, optional`

String indicating weighting function. Full explanation for this and the remaining arguments can be found below.

`wvar : optional`

Variables for use with weighting functions.

`wopts : optional`

Optional input for reusing Chebyshev moments.

`maxpl : float or int, optional`

An upper bound on the number of Chebyshev moments.
`limlst` : int, optional
Upper bound on the number of cycles (≥ 3) for use with a sinusoidal weighting and an infinite end-point.

See Also

`dblquad` : double integral
`tplquad` : triple integral
`nquad` : n-dimensional integrals (uses `quad` recursively)
`fixed_quad` : fixed-order Gaussian quadrature
`quadrature` : adaptive Gaussian quadrature
`odeint` : ODE integrator
`ode` : ODE integrator
`simps` : integrator for sampled data
`romb` : integrator for sampled data
`scipy.special` : for coefficients and roots of orthogonal polynomials

Notes

**Extra information for quad() inputs and outputs*

If `full_output` is non-zero, then the third output argument (`infodict`) is a dictionary with entries as tabulated below. For infinite limits, the range is transformed to $(0,1)$ and the optional outputs are given with respect to this transformed range. Let `M` be the input argument `limit` and let `K` be `infodict['last']`. The entries are:

'neval'
The number of function evaluations.
'last'
The number, `K`, of subintervals produced in the subdivision process.
'alist'
A rank-1 array of length `M`, the first `K` elements of which are the left end points of the subintervals in the partition of the integration range.
'blist'
A rank-1 array of length `M`, the first `K` elements of which are the right end points of the subintervals.
'rlist'
A rank-1 array of length `M`, the first `K` elements of which are the integral approximations on the subintervals.
'elist'
A rank-1 array of length `M`, the first `K` elements of which are the moduli of the absolute error estimates on the subintervals.
'iord'
A rank-1 integer array of length `M`, the first `L` elements of which are pointers to the error estimates over the subintervals with `L=K` if $K \leq M/2+2$ or `L=M+1-K` otherwise. Let `I` be the sequence `infodict['iord']` and let `E` be the sequence `infodict['elist']`. Then `E[I[1]], ..., E[I[L]]` forms a decreasing sequence.

If the input argument `points` is provided (i.e. it is not `None`), the following additional outputs are placed in the output dictionary. Assume the `points` sequence is of length `P`.

```
'pts'  
    A rank-1 array of length P+2 containing the integration limits  
    and the break points of the intervals in ascending order.  
    This is an array giving the subintervals over which integration  
    will occur.  
'level'  
    A rank-1 integer array of length M (=limit), containing the  
    subdivision levels of the subintervals, i.e., if (aa,bb) is a  
    subinterval of (pts[1], pts[2]) where pts[0] and pts[2]  
    are adjacent elements of infodict['pts'], then (aa,bb) has level 1  
    if  $|bb-aa| = |pts[2]-pts[1]| * 2^{*-1}$ .  
'ndin'  
    A rank-1 integer array of length P+2. After the first integration  
    over the intervals (pts[1], pts[2]), the error estimates over some  
    of the intervals may have been increased artificially in order to  
    put their subdivision forward. This array has ones in slots  
    corresponding to the subintervals for which this happens.
```

Weighting the integrand

The input variables, `weight` and `wvar`, are used to weight the integrand by a select list of functions. Different integration methods are used to compute the integral with these weighting functions. The possible values of `weight` and the corresponding weighting functions are.

<code>weight</code>	Weight function used	<code>wvar</code>
'cos'	<code>cos(w*x)</code>	<code>wvar = w</code>
'sin'	<code>sin(w*x)</code>	<code>wvar = w</code>
'alg'	<code>g(x) = ((x-a)**alpha) * ((b-x)**beta)</code>	<code>wvar = (alpha, beta)</code>
'alg-loga'	<code>g(x)*log(x-a)</code>	<code>wvar = (alpha, beta)</code>
'alg-logb'	<code>g(x)*log(b-x)</code>	<code>wvar = (alpha, beta)</code>
'alg-log'	<code>g(x)*log(x-a)*log(b-x)</code>	<code>wvar = (alpha, beta)</code>
'cauchy'	<code>1/(x-c)</code>	<code>wvar = c</code>

`wvar` holds the parameter `w`, `(alpha, beta)`, or `c` depending on the weight selected. In these expressions, `a` and `b` are the integration limits.

For the '`cos`' and '`sin`' weighting, additional inputs and outputs are available.

For finite integration limits, the integration is performed using a Clenshaw-Curtis method which uses Chebyshev moments. For repeated calculations, these moments are saved in the output dictionary:

```
'momcom'
```

The maximum level of Chebyshev moments that have been computed, i.e., if M_c is `infodict['momcom']` then the moments have been computed for intervals of length $|b-a| * 2^{*(-l)}$, $l=0,1,\dots,M_c$.

`'nnlog'`
A rank-1 integer array of length $M(=limit)$, containing the subdivision levels of the subintervals, i.e., an element of this array is equal to l if the corresponding subinterval is $|b-a| * 2^{*(-l)}$.

`'chebmo'`
A rank-2 array of shape $(25, \maxpl)$ containing the computed Chebyshev moments. These can be passed on to an integration over the same interval by passing this array as the second element of the sequence `wopts` and passing `infodict['momcom']` as the first element.

If one of the integration limits is infinite, then a Fourier integral is computed (assuming $w \neq 0$). If `full_output` is 1 and a numerical error is encountered, besides the error message attached to the output tuple, a dictionary is also appended to the output tuple which translates the error codes in the array `info['ierlst']` to English messages. The output information dictionary contains the following entries instead of `'last'`, `'alist'`, `'blist'`, `'rlist'`, and `'elist'`:

`'lst'`
The number of subintervals needed for the integration (call it K_f).

`'rslist'`
A rank-1 array of length $M_f=limlst$, whose first K_f elements contain the integral contribution over the interval $(a+(k-1)c, a+kc)$ where $c = (2*\text{floor}(|w|) + 1) * \pi / |w|$ and $k=1,2,\dots,K_f$.

`'ierlst'`
A rank-1 array of length M_f containing the error estimate corresponding to the interval in the same position in `infodict['rslist']`.

`'ierlst'`
A rank-1 integer array of length M_f containing an error flag corresponding to the interval in the same position in `infodict['rslist']`. See the explanation dictionary (last entry in the output tuple) for the meaning of the codes.

Examples

Calculate $\int_0^4 x^2 dx$ and compare with an analytic result

```
>>> from scipy import integrate
>>> x2 = lambda x: x**2
>>> integrate.quad(x2, 0, 4)
(21.33333333333332, 2.3684757858670003e-13)
>>> print(4**3 / 3.) # analytical result
21.3333333333
```

Calculate $\int_0^\infty e^{-x} dx$

```
>>> invexp = lambda x: np.exp(-x)
>>> integrate.quad(invexp, 0, np.inf)
(1.0, 5.842605999138044e-11)

>>> f = lambda x,a : a*x
>>> y, err = integrate.quad(f, 0, 1, args=(1,))
>>> y
0.5
>>> y, err = integrate.quad(f, 0, 1, args=(3,))
>>> y
1.5

Calculate  $\int_0^1 x^2 + y^2 dx$  with ctypes, holding
y parameter as 1::

testlib.c =>
    double func(int n, double args[n]){
        return args[0]*args[0] + args[1]*args[1]; }
compile to library testlib.*

::

from scipy import integrate
import ctypes
lib = ctypes.CDLL('/home/.../testlib.*') #use absolute path
lib.func.restype = ctypes.c_double
lib.func.argtypes = (ctypes.c_int,ctypes.c_double)
integrate.quad(lib.func,0,1,(1))
#(1.333333333333333, 1.4802973661668752e-14)
print((1.0**3/3.0 + 1.0) - (0.0**3/3.0 + 0.0)) #Analytic result
# 1.333333333333333
```

Be aware that pulse shapes and other sharp features as compared to the size of the integration interval may not be integrated correctly using this method. A simplified example of this limitation is integrating a y-axis reflected step function with many zero values within the integrals bounds.

```
>>> y = lambda x: 1 if x<=0 else 0
>>> integrate.quad(y, -1, 1)
(1.0, 1.1102230246251565e-14)
>>> integrate.quad(y, -1, 100)
(1.0000000002199108, 1.0189464580163188e-08)
>>> integrate.quad(y, -1, 10000)
(0.0, 0.0)
```

```
[((0, xmax), integrate.quad(f1,0,xmax)[0]) for xmax in np.arange(1,5)]
```

```
[((0, 1), 0.34858491873298725),
 ((0, 2), 0.8600106383901718),
 ((0, 3), 1.0438816972950689),
 ((0, 4), 1.0074874684274517)]
```

La rutina devuelve dos valores. El primero es la estimación del valor de la integral y el segundo una estimación del

error absoluto. Además, la función acepta límites de integración infinitos ($\pm\infty$, definidos en **Numpy**)

```
integrate.quad(f1,-np.inf,np.inf)
```

```
(-0.3871487639489655, 5.459954581847885e-09)
```

12.3.1 Ejemplo de función fuertemente oscilatoria

```
k = 200
L = 2*np.pi
a = 0.1
def f2(x):
    return np.sin(k*x)*np.exp(-a*x)
```

```
# Valor exacto de la integral
I=k/a**2*(np.exp(-a*L)-1)/(1-k**2/a**2)
print(I)
```

```
0.0023325601276845158
```

```
Iq= integrate.quad(f2,0,L)
```

```
/usr/lib64/python3.7/site-packages/scipy/integrate/quadpack.py:385: IntegrationWarning: The maximum number of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties. If the position of a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges. Perhaps a special-purpose integrator should be used.
warnings.warn(msg, IntegrationWarning)
```

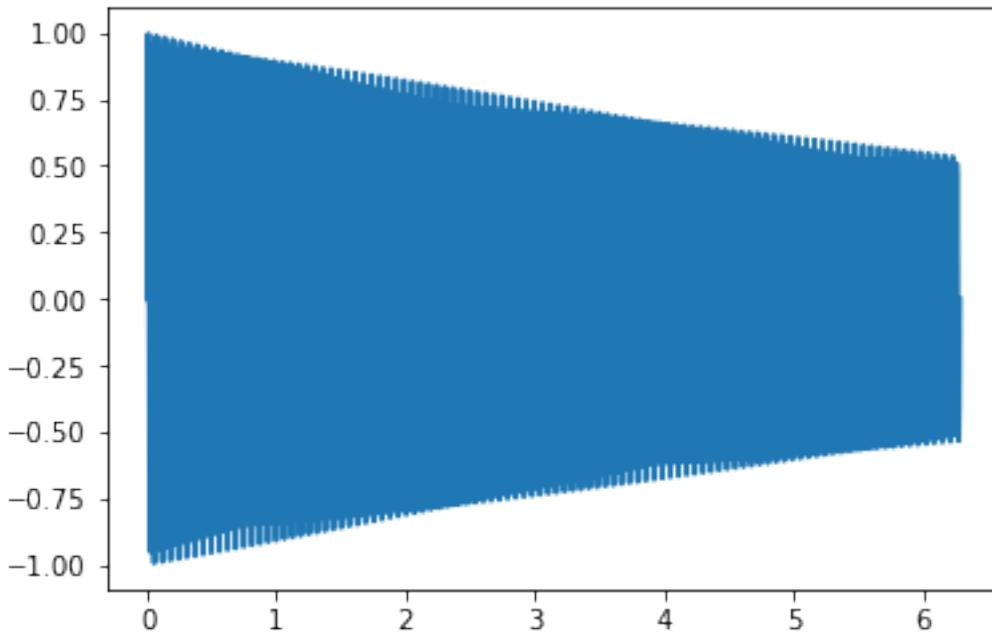
```
I_err = (I-Iq[0])/I           # Error relativo con el valor exacto
print("I= {:.5g} ± {:.5g}\nError relativo= {:.6g}\n".format(*Iq, I_err))
```

```
I= -0.0043611 ± 0.019119
Error relativo= 2.86965
```

El error relativo entre el valor obtenido numéricamente y el valor exacto I es grande. Esto se debe a la naturaleza del integrando. Grafiquemos sólo una pequeña parte

```
x = np.linspace(0,L,1500)
plt.plot(x, f2(x))
```

```
[<matplotlib.lines.Line2D at 0x7f9293c91940>]
```



La rutina `quad` es versatil y tiene una opción específica para integrandos oscilatorios, que permite calcular las integrales de una función f multiplicadas por una función oscilatoria

$$I = \int_a^b f(x) \text{weight}(wx) dx$$

Para ello debemos usar el argumento `weight` y `wvar`. En este caso usaremos `weight='sin'`

```
# La función sin es el factor oscilatorio:
def f3(x):
    return np.exp(-a*x)
```

```
I= integrate.quad(f3, 0, L, weight='sin', wvar=k)
```

```
I_err = (I-Is[0])/I           # Error relativo con el valor exacto
print("I= {:.5g} ± {:.5g}\nError relativo= {:.6g}\n".format(*Is, I_err))
```

```
I= 0.0023326 ± 1.1788e-33
Error relativo= 5e-07
```

El error relativo obtenido respecto al valor exacto es varios órdenes de magnitud menor. Comparemos los tiempos de ejecución:

```
%timeit integrate.quad(f2, 0, L)
```

```
/usr/lib64/python3.7/site-packages/scipy/integrate/quadpack.py:385:
IntegrationWarning: The maximum number of subdivisions (50) has been achieved.
If increasing the limit yields no improvement it is advised to analyze
the integrand in order to determine the difficulties. If the position of a
local difficulty can be determined (singularity, discontinuity) one will
probably gain from splitting up the interval and calling the integrator
on the subranges. Perhaps a special-purpose integrator should be used.
warnings.warn(msg, IntegrationWarning)
```

```
8.17 ms ± 85.2 ts per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit integrate.quad(f3,0,L, weight='sin', wvar=k)
```

```
61.1 ts ± 585 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Usar un integrador más específico para el integrando no sólo nos da un mejor resultado sino que el tiempo de ejecución es más de 100 veces más corto.

12.3.2 Funciones de más de una variable

Consideremos el caso en que queremos integrar alguna función especial. Podemos usar Scipy para realizar la integración y para evaluar el integrando. Como `special.jn` depende de dos variables, tenemos que crear una función intermedia que dependa sólo de la variable de integración

```
integrate.quad(lambda x: special.jn(0,x), 0 , 10)
```

```
(1.0670113039567362, 7.434789460651883e-14)
```

En realidad, la función `quad` permite el uso de argumentos que se le pasan a la función a integrar. La forma de llamar al integrador será en general:

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08,
      limit=50, points=None, weight=None, wvar=None, wopts=None, maxpl=50,
      limlst=50)
```

El argumento `args` debe ser una tupla, y contiene los argumentos extra que acepta la función a integrar, esta función debe llamarse en la forma `func(x, *args)`. O sea que siempre la integramos respecto a su primer argumento. Apliquemos esto a la función de Bessel. En este caso, la variable a integrar es el segundo argumento de `special.jn`, por lo que creamos una función con el orden correcto de argumentos:

```
def bessel_n(x, n):
    return special.jn(n,x)
```

```
integrate.quad(bessel_n, 0, 10, args=(0,))
```

```
(1.0670113039567362, 7.434789460651883e-14)
```

```
print('n      \int_0^10 J_n(x) dx')
for n in range(6):
    print(n,': ', integrate.quad(bessel_n, 0, 10, args=(n,))[0])
```

```
n      int_0^10 J_n(x) dx
0 : 1.0670113039567362
1 : 1.2459357644513482
2 : 0.9800658116190144
3 : 0.7366751370811073
4 : 0.8633070530086401
5 : 1.1758805092851239
```

12.4 Álgebra lineal

El módulo de álgebra lineal se solapa un poco con funciones similares en **Numpy**. Ambos usan finalmente una implementación de bibliotecas conocidas (LAPACK, BLAS). La diferencia es que **Scipy** asegura que utiliza las optimizaciones de la librería ATLAS y presenta algunos métodos y algoritmos que no están presentes en **Numpy**.

Una de las aplicaciones más conocidas por nosotros es la rotación de vectores. Como bien sabemos rotar un vector es equivalente a multiplicarlo por la matriz de rotación correspondiente. Esquemáticamente:

(Gentileza de xkcd)

```
from scipy import linalg
```

Este módulo tiene funciones para trabajar con matrices, descriptas como *arrays* bidimensionales.

```
arr = np.array([[3, 2, 1], [6, 4, 1], [12, 8, 13.3]])
print(arr)
```

```
[[ 3.   2.   1. ]
 [ 6.   4.   1. ]
 [12.   8.  13.3]]
```

```
A = np.array([[1, -2, -3], [1, -1, -1], [-1, 3, 1]])
print(A)
```

```
[[ 1 -2 -3]
 [ 1 -1 -1]
 [-1  3  1]]
```

```
# La matriz transpuesta
A.T
```

```
array([[ 1,  1, -1],
       [-2, -1,  3],
       [-3, -1,  1]])
```

12.4.1 Productos y normas

Norma de un vector

La norma está dada por

$$\|v\| = \sqrt{v_1^2 + \dots + v_n^2}$$

```
v = np.array([2,1,3])
linalg.norm(v)          # Norma
```

```
3.7416573867739413
```

```
linalg.norm(v) == np.sqrt(np.sum(np.square(v)))
```

```
True
```

Producto interno

El producto entre una matriz y un vector está definido en **Numpy** mediante la función `dot()`

```
w1 = np.dot(A, v)           # Multiplicación de matrices
w1
```

```
array([-9, -2,  4])
```

```
w2 = np.dot(v, A)
w2
```

```
array([ 0,  4, -4])
```

```
np.dot(v.T, A)
```

```
array([ 0,  4, -4])
```

```
print(v.shape, A.shape)
```

```
(3,) (3, 3)
```

El producto interno entre vectores se calcula de la misma manera

$$\langle v, w \rangle$$

```
np.dot(v, w1)
```

```
-8
```

y está relacionado con la norma

$$\|v\| = \sqrt{\langle v, v \rangle}$$

```
linalg.norm(v) == np.sqrt(np.dot(v,v))
```

```
True
```

```
np.dot(v,A)
```

```
array([ 0,  4, -4])
```

```
v.shape
```

```
(3,)
```

```
v2 = np.reshape(v, (3,1))
```

```
v2.shape
```

```
(3, 1)
```

```
np.dot(A, v2)
```

```
array([[-9],  
      [-2],  
      [ 4]])
```

```
np.dot(A, v2).shape
```

```
(3, 1)
```

Ahora las dimensiones de v2 y A no coinciden para hacer el producto matricial

```
np.dot(v2, A)
```

En versiones nuevas de Python la multiplicación de matrices puede realizarse con el operador @

```
np.dot(v2, A)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-60-c31ca8fbc4d7> in <module>()  
----> 1 np.dot(v2, A)
```

```
ValueError: shapes (3,1) and (3,3) not aligned: 1 (dim 1) != 3 (dim 0)
```

```
A @ v2
```

```
array([[-9],  
      [-2],  
      [ 4]])
```

```
v3 = np.dot(v2.T, A)
```

```
print(v3)
v3.shape
```

```
[[ 0  4 -4]]
```

```
(1, 3)
```

Notemos que el producto interno se puede pensar como un producto de matrices. En este caso, el producto de una matriz de 3×1 , por otra de 1×3 :

$$v^t w = \begin{pmatrix} -9 & -2 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$$

donde estamos pensando al vector como columna.

Producto exterior

El producto exterior puede ponerse en términos de multiplicación de matrices como

$$v \otimes w = vw^t = \begin{pmatrix} -9 \\ -2 \\ 4 \end{pmatrix} \begin{pmatrix} 2 & 1 & 3 \end{pmatrix}$$

```
oprod = np.outer(v,w1)
print(oprod)
```

```
[[ -18 -4   8]
 [ -9  -2   4]
 [-27 -6  12]]
```

12.4.2 Aplicación a la resolución de sistemas de ecuaciones

Vamos a usar `scipy.linalg` para obtener determinantes e inversas de matrices. Vamos a usarlo para resolver un sistema de ecuaciones lineales:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \end{cases}$$

Esta ecuación se puede escribir en forma matricial como

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Veamos un ejemplo concreto. Supongamos que tenemos el siguiente sistema

$$\begin{cases} x_1 + 2x_2 + 3x_3 = 1 \\ 2x_1 + x_2 + 3x_3 = 2 \\ 4x_1 + x_2 - x_3 = 1 \end{cases}$$

por lo que, en forma matricial será:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \\ 4 & 1 & -1 \end{pmatrix}$$

y

$$b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

```
A = np.array([[1,2,3],[2,1,3],[4,1,-1]])
b = np.array([[1,2,3]]).T
print('A=', A, '\n')
print('b=', b, '\n')
```

```
A= [[ 1  2  3]
 [ 2  1  3]
 [ 4  1 -1]]
```

```
b= [[1]
 [2]
 [3]]
```

```
x = np.dot(linalg.inv(A), b)
print('Resultado:\n', x)
```

```
Resultado:
[[ 0.83333333]
 [-0.16666667]
 [ 0.16666667]]
```

12.4.3 Descomposición de matrices

Si consideramos el mismo problema de resolución de ecuaciones

$$Ax = b$$

pero donde debemos resolver el problema para un valor dado de los coeficientes (la matriz A) y muchos valores distintos del vector b , suele ser útil realizar lo que se llama la descomposición LU de la matriz.

Si escribimos a la matriz A como el producto de tres matrices $A = PLU$ donde P es una permutación de las filas, L es una matriz triangular inferior (Los elementos por encima de la diagonal son nulos) y U una triangular superior. En este caso los dos sistemas:

$$Ax = b \quad \text{y} \quad PAx = Pb$$

tienen la misma solución. Entonces podemos resolver el sistema en dos pasos:

$$Ly = b$$

con

$$y = Ux.$$

En ese caso, resolvemos una sola vez la descomposición LU , y luego ambas ecuaciones se pueden resolver eficientemente debido a la forma de las matrices.

```
A = np.array([[1,3,4],[2,1,3],[4,1,2]])

print('A=', A, "\n")

P, L, U = linalg.lu(A)
print("PLU=", np.dot(P, np.dot(L, U)))
print("\nLU=", np.dot(L, U))
print("\nL=", L)
print("\nU=", U)
```

```
A= [[1 3 4]
 [2 1 3]
 [4 1 2]]

PLU= [[1. 3. 4.]
 [2. 1. 3.]
 [4. 1. 2.]]

LU= [[4. 1. 2.]
 [1. 3. 4.]
 [2. 1. 3.]]

L= [[1. 0. 0.]
 [0.25 1. 0.]
 [0.5 0.18181818 1.]]]

U= [[4. 1. 2.]
 [0. 2.75 3.5]
 [0. 0. 1.36363636]]
```

12.4.4 Autovalores y autovectores

La necesidad de encontrar los autovalores y autovectores de una matriz aparece en muchos problemas de física e ingeniería. Se trata de encontrar el escalar λ y el vector (no nulo) v tales que

$$Av = \lambda v$$

```
with np.printoptions(precision=3):
    B = np.array([[0,1.,1],[2,1,0], [3,4,5]])
    print(B, '\n')
    u, v = linalg.eig(B)
    c = np.dot(v,np.dot(np.diag(u), linalg.inv(v)))
    print(c, '\n')
    print(np.real_if_close(c), '\n')
    print('')
    print('Autovalores=', u, '\n')
    print('Autovalores=', np.real_if_close(u))
```

```
[[0. 1. 1.]
 [2. 1. 0.]
 [3. 4. 5.]]]

[[-1.892e-16+0.j 1.000e+00+0.j 1.000e+00+0.j]
 [ 2.000e+00+0.j 1.000e+00+0.j 3.053e-16+0.j]
```

(continué en la próxima página)

(provine de la página anterior)

```
[ 3.000e+00+0.j  4.000e+00+0.j  5.000e+00+0.j]]  
[ [-1.892e-16  1.000e+00  1.000e+00]  
 [ 2.000e+00  1.000e+00  3.053e-16]  
 [ 3.000e+00  4.000e+00  5.000e+00] ]  
  
Autovalores= [ 5.854+0.j -0.854+0.j  1.     +0.j]  
  
Autovalores= [ 5.854 -0.854  1.     ]
```

Veamos como funciona para la matriz definida anteriormente

```
print(A)  
u, v = linalg.eig(A)  
print(np.real_if_close(np.dot(v,np.dot(np.diag(u), linalg.inv(v)))))  
print("Autovalores=", np.real_if_close(u))  
print("Autowectores=", np.real_if_close(v))
```

```
[[1 3 4]  
 [2 1 3]  
 [4 1 2]]  
[[1. 3. 4.]  
 [2. 1. 3.]  
 [4. 1. 2.]]  
Autovalores= [ 7.10977223 -2.10977223 -1.          ]  
Autowectores= [[-0.63273853 -0.66101705 -0.33333333]  
 [-0.49820655 -0.25550401 -0.66666667]  
 [-0.59281716  0.70553112  0.66666667]]
```

```
np.real_if_close?
```

12.4.5 Rutinas de resolución de ecuaciones lineales

Scipy tiene además de las rutinas de trabajo con matrices, rutinas de resolución de sistemas de ecuaciones. En particular la función `solve()`

```
solve(a, b, sym_pos=False, lower=False, overwrite_a=False, overwrite_b=False,  
      debug=False, check_finite=True)  
  
Solve the equation ``a x = b`` for ``x``.  
  
Parameters  
-----  
a : (M, M) array_like  
    A square matrix.  
b : (M,) or (M, N) array_like  
    Right-hand side matrix in ``a x = b``.  
...  
...
```

```
a = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])  
b = np.array([2, 4, -1])  
x = linalg.solve(a, b)  
x
```

```
array([ 2., -2.,  9.])
```

```
np.allclose(np.dot(a, x) , b)
```

```
True
```

```
np.dot(a,x) == b
```

```
array([ True,  True,  True])
```

Para sistemas de ecuaciones grandes, la función `solve()` es más rápida que invertir la matriz

```
A1 = np.random.random((2000,2000))
b1 = np.random.random(2000)
```

```
%timeit linalg.solve(A1,b1)
```

```
451 ms ± 89.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit np.dot(linalg.inv(A1),b1)
```

```
962 ms ± 9.37 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

12.5 Entrada y salida de datos

12.5.1 Archivos comprimidos en Python

Existen varias formas de reducir el tamaño de los archivos de datos. Varios factores, tales como el sistema operativo, nuestra familiaridad con cada uno de ellos, le da una cierta preferencia a algunos de los métodos disponibles. Veamos cómo hacer para leer y escribir algunos de los siguientes formatos: `zip`, `gzip`, `bz2`

```
import gzip
```

```
import bz2
```

```
fi= gzip.open('..../data/palabras.words.gz')
a = fi.read()
fi.close()
```

```
l= a.splitlines()
print(l[:10])
```

```
[b'xc3x81frica', b'xc3x81ngela', b'xc3xa1baco', b'xc3xa1bsida',  
 ↪ b'xc3xa1bside', b'xc3xa1cana', b'xc3xa1caro', b'xc3xa1cates', b'xc3xa1cido',  
 ↪ b'xc3xa1cigos']
```

Con todo esto podríamos escribir (si tuviéramos necesidad) una función que puede leer un archivo en cualquiera de estos formatos

```
import gzip
import bz2
from os.path import splitext
import zipfile

def abrir(fname, mode='r'):
    if fname.endswith('gz'):
        fi= gzip.open(fname, mode=mode)
    elif fname.endswith('bz2'):
        fi= bz2.open(fname, mode=mode)
    elif fname.endswith('zip'):
        fi= zipfile.ZipFile(fname, mode=mode)
    else:
        fi = open(fname, mode=mode)
    return fi
```

```
ff= abrir('../data/palabras.words.gz')
a = ff.read()
ff.close()
```

```
l = a.splitlines()
```

```
for p in l[:10]:
    print(p, p.decode())
```

```
b'xc3x81frica' África
b'xc3x81ngela' Ángela
b'xc3xa1baco' ábaco
b'xc3xa1bsida' ábsida
b'xc3xa1bside' ábside
b'xc3xa1cana' ácana
b'xc3xa1caro' ácaro
b'xc3xa1cates' ácates
b'xc3xa1cido' ácido
b'xc3xa1cigos' ácigos
```

```
!ls ../data/palabras*
```

```
../data/palabras_en.words.gz  ../data/palabras.words.gz
../data/palabras.tar.gz
```

```
f2 = abrir('../data/palabras.words.zip')
b= f2.read('palabras.words')
f2.close()
```

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

<ipython-input-95-2d95de7e1ee6> in <module>()
----> 1 f2 = abrir('../data/palabras.words.zip')
      2 b= f2.read('palabras.words')
      3 f2.close()
```

(continué en la próxima página)

(proviene de la página anterior)

```
<ipython-input-90-7fea08c0a3b1> in abrir(fname, mode)
  10      fi= bz2.open(fname, mode=mode)
  11  elif fname.endswith('zip'):
--> 12      fi= zipfile.ZipFile(fname, mode=mode)
  13  else:
  14      fi = open(fname, mode=mode)

/usr/lib64/python3.7/zipfile.py in __init__(self, file, mode, compression, allowZip64,
   compresslevel)
 1202         while True:
 1203             try:
-> 1204                 self.fp = io.open(file, filemode)
 1205             except OSError:
 1206                 if filemode in modeDict:

FileNotFoundException: [Errno 2] No such file or directory: '../data/palabras.words.zip'
```

b == a

12.5.2 Entrada/salida con Numpy

Datos en formato texto

Veamos un ejemplo (apenas) más complicado, de un archivo en formato de texto, donde antes de la lista de números hay un encabezado

```
import numpy as np
import matplotlib.pyplot as plt
```

```
!head ../data/tof_signal_5.dat
```

```
# tiempo    cuentas
4.953125e-06 -7.940000e-05
4.963125e-06 -5.930000e-05
4.973125e-06 -8.945000e-05
4.983125e-06 -7.940000e-05
4.993125e-06 -6.935000e-05
5.003125e-06 -6.935000e-05
5.013125e-06 -9.950000e-05
5.023125e-06 -5.930000e-05
5.033125e-06 -5.930000e-05
```

```
x0 = np.loadtxt('../data/tof_signal_5.dat')
```

```
x0.shape, type(x0)
```

```
((1000, 2), numpy.ndarray)
```

```
x0[0].shape
```

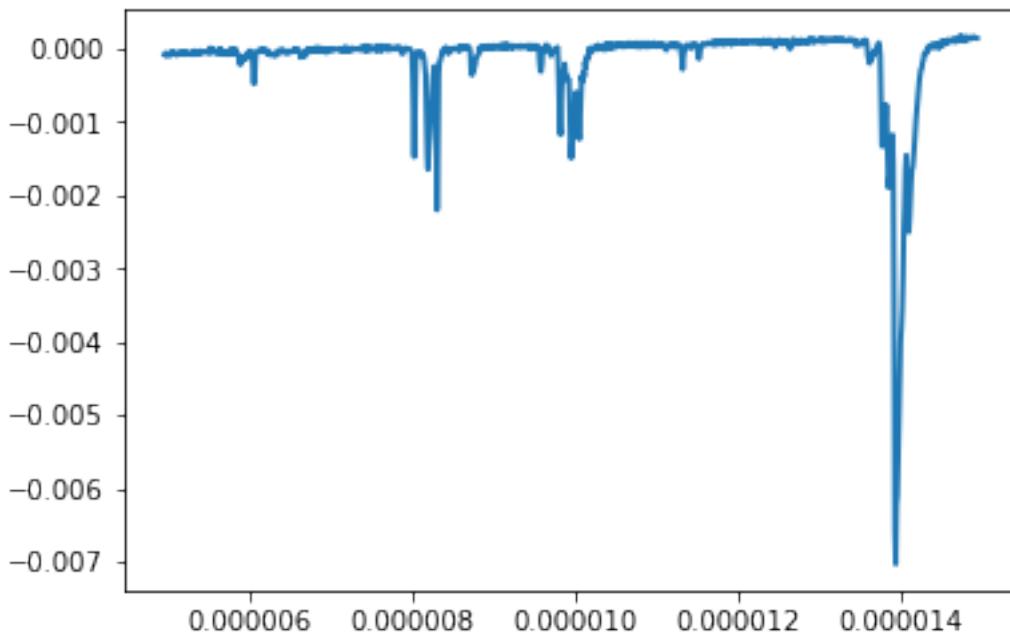
```
(2,)
```

```
x0[0]
```

```
array([ 4.953125e-06, -7.940000e-05])
```

```
plt.plot(x0[:,0], x0[:,1])
```

```
[<matplotlib.lines.Line2D at 0x7f926044ab00>]
```



La manera más simple de leer datos de un archivo es a través de `loadtxt()`.

```
np.info(np.loadtxt)
loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None,
        converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0,
        encoding='bytes')
Load data from a text file.

Each row in the text file must have the same number of values.
```

En su forma más simple sólo necesita como argumento el nombre del archivo. En este caso, había una primera línea que fue ignorada porque empieza con el carácter # que indica que la línea es un comentario.

Veamos otro ejemplo, donde las líneas que son parte de un encabezado se saltean, utilizando el argumento `skiprows`

```
fdatos= '../data/exper_col.dat'
!head ../data/exper_col.dat
```

```
x1 = np.loadtxt(fdatos, skiprows=5)
print(X1.shape)
print(X1[0])
```

```
(76, 4)
[ 9.901      15.35198465  12.12121212  14.86049333]
```

Como el archivo tiene cuatro columnas el array X tiene dimensiones (74, 4) correspondiente a las 74 filas y las 4 columnas. Si sólo necesitamos un grupo de estos datos podemos utilizar el argumento `usecols = (c1, c2)` que nos permite elegir cuáles son las columnas a leer:

```
x, y = np.loadtxt(fdatos, skiprows=5, usecols=[0, 2], unpack=True)
print(x.size, y.size)
```

```
76 76
```

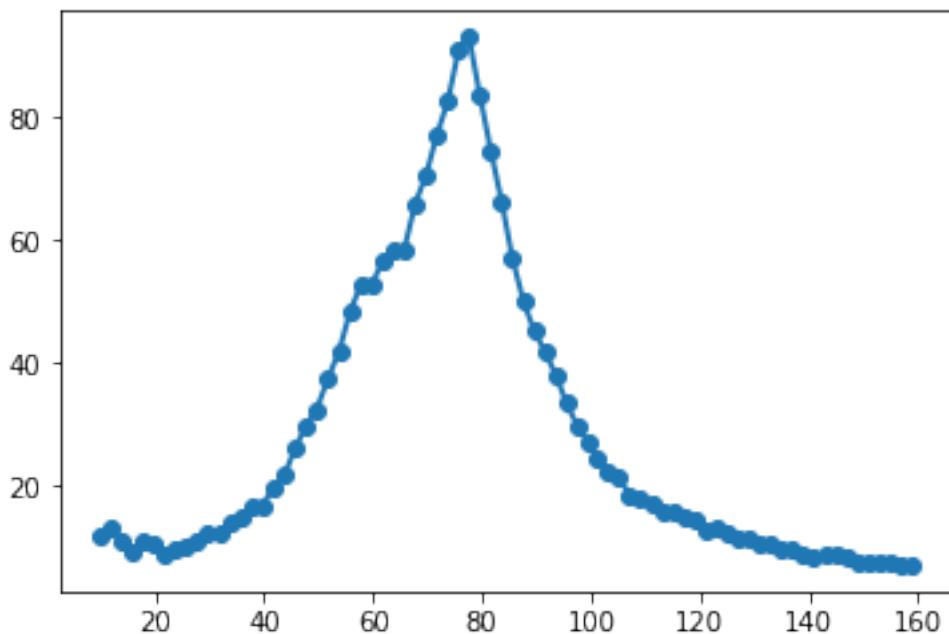
```
Y = np.loadtxt(fdatos, skiprows=5, usecols=[0, 2])
print(Y.size, Y[0])
```

```
152 [ 9.901      12.12121212]
```

En este ejemplo, mediante el argumento `unpack=True`, le indicamos a la función `loadtxt` que desempaque lo que lee en variables diferentes (`x`, `y` en este caso)

```
plt.plot(x,y, 'o-')
```

```
[<matplotlib.lines.Line2D at 0x7f926062a0f0>]
```



Como numpy se especializa en manejar números, tiene muchas funciones para crear arrays a partir de información numérica a partir de texto o archivos (como los CSV, por ejemplo). Ya vimos como leer datos con `loadtxt`. También se pueden generar desde un string:

Clases de Python

```
np.fromstring(u"1.0 2.3    3.0 4.1    -3.1", sep=" ", dtype=float)
```

```
array([ 1.,  2.3,  3.,  4.1, -3.1])
```

Para guardar datos en formato texto podemos usar, de la misma manera,

```
Y = np.vstack((x,y)).T  
print(Y.shape)
```

```
(76, 2)
```

```
np.savetxt('tmp.dat', Y)
```

```
!head tmp.dat
```

```
9.9009999999999801e+00 1.212121212121209979e+01  
1.18810000000000023e+01 1.338496506439940070e+01  
1.37929999999999926e+01 1.101369465980729956e+01  
1.58130000000000061e+01 9.490067063140580572e+00  
1.78019999999999960e+01 1.106306508676360068e+01  
1.978399999999999892e+01 1.056836569579290064e+01  
2.180600000000000094e+01 9.041259351048690718e+00  
2.38019999999999960e+01 9.743805123897519849e+00  
2.56799999999999972e+01 1.000583998442670008e+01  
2.769900000000000162e+01 1.093034161826770045e+01
```

La función `savetxt()` tiene varios argumentos opcionales:

```
np.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='',  
comments='# ', encoding=None)
```

Por ejemplo, podemos darle un formato de salida con el argumento `fmt`, y darle un encabezado con `header`

```
np.savetxt('tmp.dat', Y, fmt='%.6g', header="Energ Exper")  
!head tmp.dat
```

```
# Energ Exper  
9.901 12.1212  
11.881 13.385  
13.793 11.0137  
15.813 9.49007  
17.802 11.0631  
19.784 10.5684  
21.806 9.04126  
23.802 9.74381  
25.68 10.0058
```

Datos en formato binario

```
np.save('test.npy', X1) # Grabamos el array a archivo  
X2 = np.load('test.npy') # Y lo leemos
```

```
# Veamos si alguno de los elementos difiere
print('X1=', X1[:10])
print('X2=', X2[:10])
```

```
X1= [[ 9.901      15.35198465 12.12121212 14.86049333]
[11.881     17.25443986 13.38496506 12.137559  ]
[13.793     17.54513159 11.01369466 12.33403468]
[15.813     14.67147284  9.49006706 10.68943707]
[17.802     15.05448826 11.06306509 11.11859838]
[19.784     12.99029519 10.5683657 10.77717061]
[21.806     12.19847748  9.04125935 10.50844347]
[23.802     13.57028821  9.74380512 10.46262448]
[25.68      13.16199377 10.00583998 9.76919784]
[27.699     14.91028557 10.93034162 11.29189365]]
X2= [[ 9.901      15.35198465 12.12121212 14.86049333]
[11.881     17.25443986 13.38496506 12.137559  ]
[13.793     17.54513159 11.01369466 12.33403468]
[15.813     14.67147284  9.49006706 10.68943707]
[17.802     15.05448826 11.06306509 11.11859838]
[19.784     12.99029519 10.5683657 10.77717061]
[21.806     12.19847748  9.04125935 10.50844347]
[23.802     13.57028821  9.74380512 10.46262448]
[25.68      13.16199377 10.00583998 9.76919784]
[27.699     14.91028557 10.93034162 11.29189365]]
```

```
print('Alguna diferencia?', np.any(X1-X2))
```

```
fAlguna diferencia? False
```

12.5.3 Ejemplo de análisis de palabras

```
# %load scripts/10_palabras.py
#!/usr/bin/ipython
import numpy as np
import matplotlib.pyplot as plt
import gzip
ifiname = '../data/palabras.words.gz'

letras = [0] * 512
with gzip.open(ifiname, mode='r') as fi:
    for l in fi.readlines():
        c = ord(l.decode('utf-8')[0])
        letras[c] += 1

nmax = np.nonzero(letras)[0].max() + 1
z = np.array(letras[:nmax])
# nmin = z.nonzero()[0].min()      # Máximo valor diferente de cero
nmin = np.argwhere(z != 0).min()
#plt.ioff()
with plt.style.context(['seaborn-talk', 'presentation']):
    fig = plt.figure(figsize=(10, 8))
    #plt.clf()
    plt.bar(np.arange(nmin, nmax), z[nmin:nmax])
    plt.xlabel('Letras con y sin acentos')
```

(continué en la próxima página)

(proviene de la página anterior)

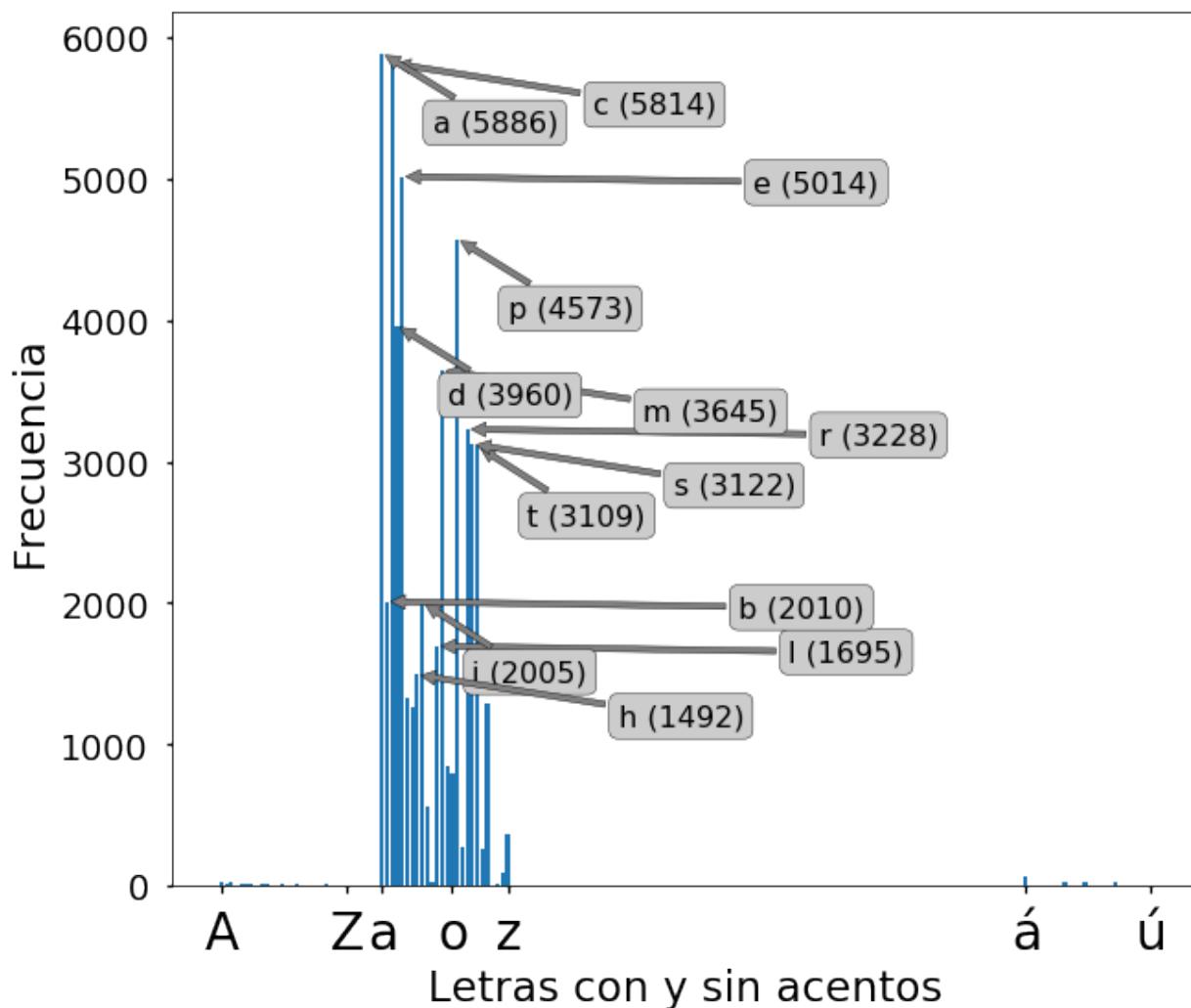
```
plt.ylabel('Frecuencia')

labels = ['A', 'Z', 'a', 'o', 'z', 'á', 'ú']
ll = [r'$\mathsf{{}}$'.format(t) for t in labels]
ts = [ord(t) for t in labels]
plt.xticks(ts, ll, fontsize='xx-large')

x0 = 0.5 * ord('á') + ord('z')
y0 = 0.2 * z.max()
umbral = 0.25
lista = (z > umbral * z.max()).nonzero()[0]

dx = [10, 40, 70]
dy = [-550, -350, -100]

for j, t in enumerate(reversed(lista)):
    plt.annotate('{} ({})'.format(chr(t), z[t]), xy=(t, z[t]), xycoords='data',
                 xytext=(t + dx[j % 3], z[t] + dy[j % 3]), bbox=dict(boxstyle="round",
                fc="0.8"),
                 arrowprops=dict(arrowstyle="simple", fc="0.5")
    )
```



12.5.4 Entrada y salida en Scipy

El submódulo `io` tiene algunas utilidades de entrada y salida de datos que permite interactuar con otros paquetes/programas. Algunos de ellos son:

- Archivos IDL ([Interactive Data Language](#))
 - `scipy.io.readsav()`
- Archivos de sonido wav, con `scipy.io.wavfile`
 - `scipy.io.wavfile.read()`
 - `scipy.io.wavfile.write()`
- Archivos fortran sin formato, con `scipy.io.FortranFile`
- Archivos Netcdf (para gran número de datos), con `scipy.io.netcdf`
- Archivos de matrices de Matlab

```
from scipy import io as sio
a = np.ones((3, 3)) + np.eye(3, 3)
print(a)
sio.savemat('datos.mat', {'a': a}) # savemat espera un diccionario
data = sio.loadmat('datos.mat', struct_as_record=True)
print(data['a'])
```

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

```
data
```

```
{'__header__': b'MATLAB 5.0 MAT-file Platform: posix, Created on: Mon Mar 25 10:27:05
2019',
 '__version__': '1.0',
 '__globals__': [],
 'a': array([[2., 1., 1.],
 [1., 2., 1.],
 [1., 1., 2.]])}
```

12.6 Ejercicios 11 (a)

1. En el archivo `palabras.words.gz` hay una larga lista de palabras, en formato comprimido. Siguiendo la idea del ejemplo dado en clases realizar un histograma de las longitudes de las palabras.
2. Modificar el programa del ejemplo de la clase para calcular el histograma de frecuencia de letras en las palabras (no sólo la primera). Considere el caso insensible a la capitalización: las mayúsculas y minúsculas corresponden a la misma letra (á es lo mismo que Á y ambas corresponden a a).
3. Utilizando el mismo archivo de palabras, Guardar todas las palabras en un array y obtener los índices de las palabras que tienen una dada letra (por ejemplo la letra j), los índices de las palabras con un número dado de letras (por ejemplo 5 letras), y los índices de las palabras cuya tercera letra es una vocal. En cada caso, dar luego las palabras que cumplen dichas condiciones.
4. En el archivo `colision.npy` hay una gran cantidad de datos que corresponden al resultado de una simulación. Los datos están organizados en trece columnas. La primera corresponde a un parámetro, mientras que las 12 restantes corresponde a cada una de las tres componentes de la velocidad de cuatro partículas. Calcular y graficar:
 5. la distribución de ocurrencias del primer parámetro.
 6. la distribución de ocurrencias de energías de la tercera partícula.
 7. la distribución de ocurrencias de ángulos de la cuarta partícula, medido respecto al tercer eje.
 8. la distribución de energías de la tercera partícula cuando la cuarta partícula tiene un ángulo menor a 90 grados con el tercer eje.

Realizar los cuatro gráficos utilizando un formato adecuado para presentación (charla o poster).

5. Leer el archivo `colision.npy` y guardar los datos en formato texto con un encabezado adecuado. Usando el comando mágico `%timeit` o el módulo `timeit`, comparar el tiempo que tarda en leer los datos e imprimir el último valor utilizando el formato de texto y el formato original `npy`. Comparar el tamaño de los dos archivos.

6. El submódulo **scipy.constants** tiene valores de constantes físicas de interés. Usando este módulo compute la constante de Stefan-Boltzmann σ utilizando la relación:

$$\sigma = \frac{2\pi^5 k_B^4}{15 h^3 c^2}$$

Confirme que el valor obtenido es correcto comparando con la constante para esta cantidad en `scipy.constants`.

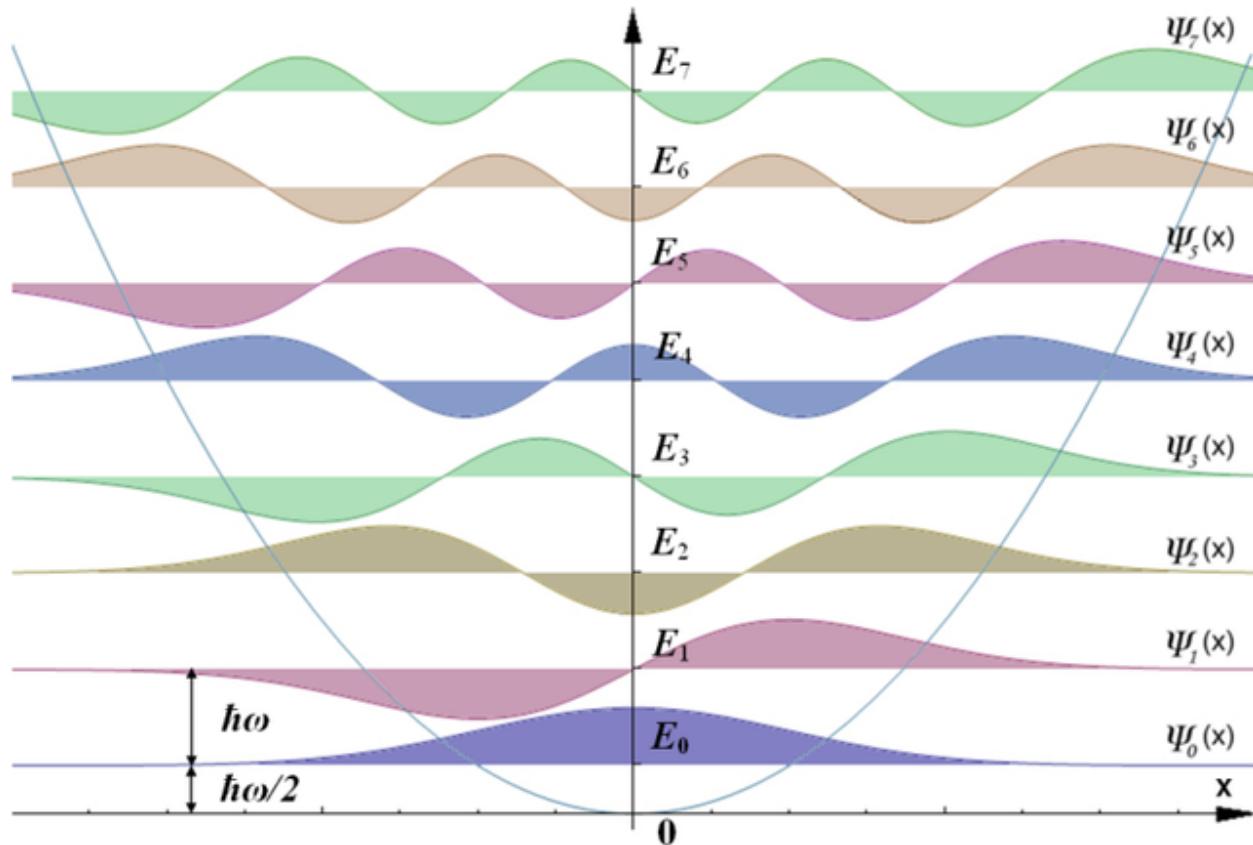
7. Usando **Scipy** y **Matplotlib** grafique las funciones de onda del oscilador armónico unidimensional para las cuatro energías más bajas ($n = 1, 2, 3, 4$), en el intervalo $[-5, 5]$. Asegúrese de que están correctamente normalizados.

Las funciones están dadas por:

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \cdot \left(\frac{\omega}{\pi}\right)^{1/4} \cdot e^{-\frac{\omega x^2}{2}} \cdot H_n(\sqrt{\omega} x), \quad n = 0, 1, 2, \dots$$

donde H_n son los polinomios de Hermite, y usando $\omega = 2$.

Trate de obtener un gráfico similar al siguiente (tomado de wikipedia. Realizado por By AllenMcC. - File: HarmOsziFunktionen.jpg, CC BY-SA 3.0)



CAPÍTULO 13

Clase 12: Un poco de graficación 3D

```
import numpy as np
import matplotlib.pyplot as plt
```

13.1 Algunas funciones útiles de Numpy

13.1.1 Vectorización de funciones escalares

Si bien en **Numpy** las funciones están vectorizadas, hay ocasiones en que las funciones son el resultado de una simulación, optimización, o integración, y si bien la paralelización puede ser trivial, no puede ser utilizada directamente con un vector. Para ello existe la función `vectorize()`. Veamos un ejemplo, calculando la función `coseno()` como la integral del `seno()`

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
```

```
def mi_cos(t):
    return 1-integrate.quad(np.sin, 0, t)[0]
```

```
mi_cos(np.pi/4)
```

```
0.7071067811865476
```

Para calcular sobre un conjunto de datos, debemos realizar una iteración llamando a esta función en cada paso:

```
x = np.linspace(-np.pi,np.pi,30)
```

```
try:
    mi_cos(x)
```

(continué en la próxima página)

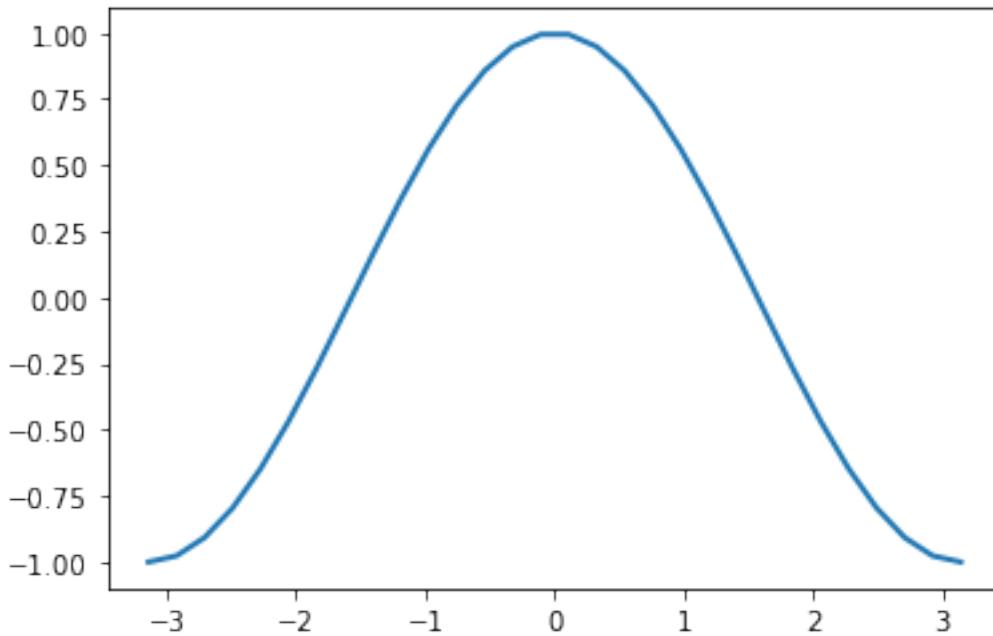
(provienec de la página anterior)

```
except Exception as e:  
    print("No funcionó: \n", e)
```

```
No funcionó:  
The truth value of an array with more than one element is ambiguous. Use a.any() or  
a.all()
```

```
y = np.zeros(x.size)  
for i,xx in enumerate(x):  
    y[i] = mi_cos(xx)  
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7ff77f2e7290>]
```

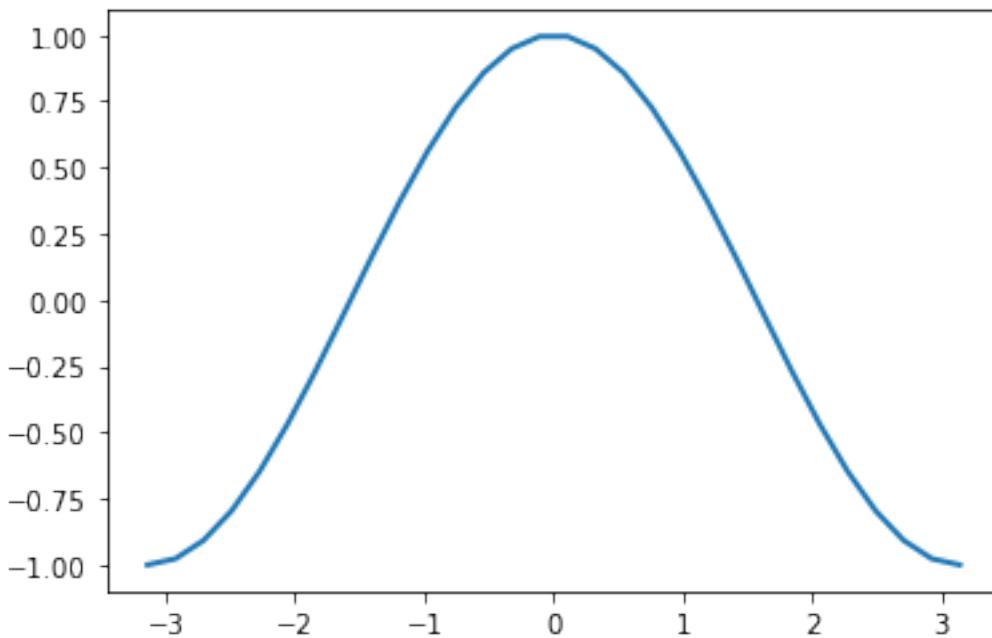


Como conveniencia, para evitar tener que hacer explícitamente el bucle `for` existe la función `vectorize`, que toma como argumento a una función que toma y devuelve escalares, y retorna una función equivalente que acepta arrays:

```
coseno = np.vectorize(mi_cos)
```

```
plt.plot(x, coseno(x), '-')
```

```
[<matplotlib.lines.Line2D at 0x7ff77f295290>]
```



Un segundo ejemplo podría ser la función que hicimos para calcular π utilizando el método de Monte Carlo

```
def estima_pi(N):
    y = np.random.random((2, N))
    c = np.square(y).sum(axis=0) < 1
    return 4 * c.sum() / N

pi_areas = np.vectorize(estima_pi) # Versión que acepta arrays
```

```
estima_pi(10000)
```

```
3.1176
```

```
N = 1000*np.ones(10, dtype='int')
print('N=', N)
print('shape=', N.shape)
pi_areas(N)
```

```
N=
[1000 1000 1000 1000 1000 1000 1000 1000 1000 1000]
shape=(10,)
```

```
array([3.116, 3.092, 3.06, 3.112, 3.112, 3.164, 3.112, 3.2, 3.032,
       3.14])
```

```
Ns = 10000 * np.ones(2000, dtype=int)
y = pi_areas(Ns)
pi = y.mean()
error = y.std()

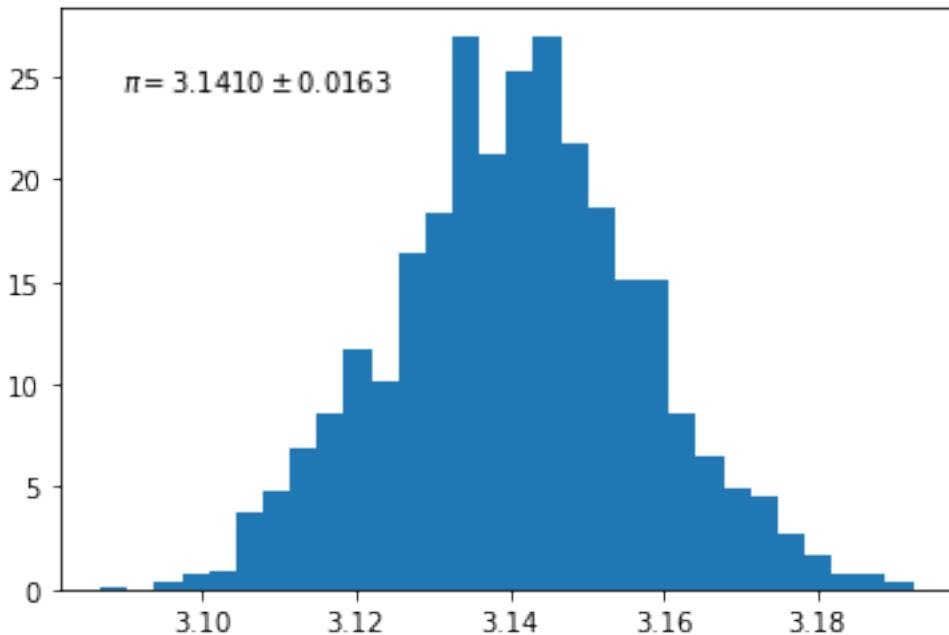
d= plt.hist(y,bins=30, density=True)
xpos= d[1].min() +(d[1].max() - d[1].min()) / 40
plt.text(xpos, 0.9*d[0].max(), r"\pi = {:.4f} \pm {:.4f}".format(pi,error))
```

(continué en la próxima página)

(provien de la página anterior)

```
print('Estimación de   {:.4f} ± {:.4f}'.format(pi, error))
```

```
Estimación de   3.1410 ± 0.0163
```



13.2 Gráficos y procesamiento sencillo en 2D

13.2.1 Histogramas en 2D

Así como trabajamos con histogramas de arrays unidimensionales en forma sencilla usando `plt.hist()` o `np.histogram()`, podemos hacerlo de una manera similar trabajando en el plano. Empecemos creando algunos datos

```
np.random.seed(0)
n = 100000
x = np.r_[np.random.normal(size=n), np.random.normal(loc=3, size=n)]
y = 2.0 + 4.0 * x - x**2 / 5 + 2.0 * \
    np.r_[np.random.normal(size=n), np.random.normal(loc=-3, size=n)]
```

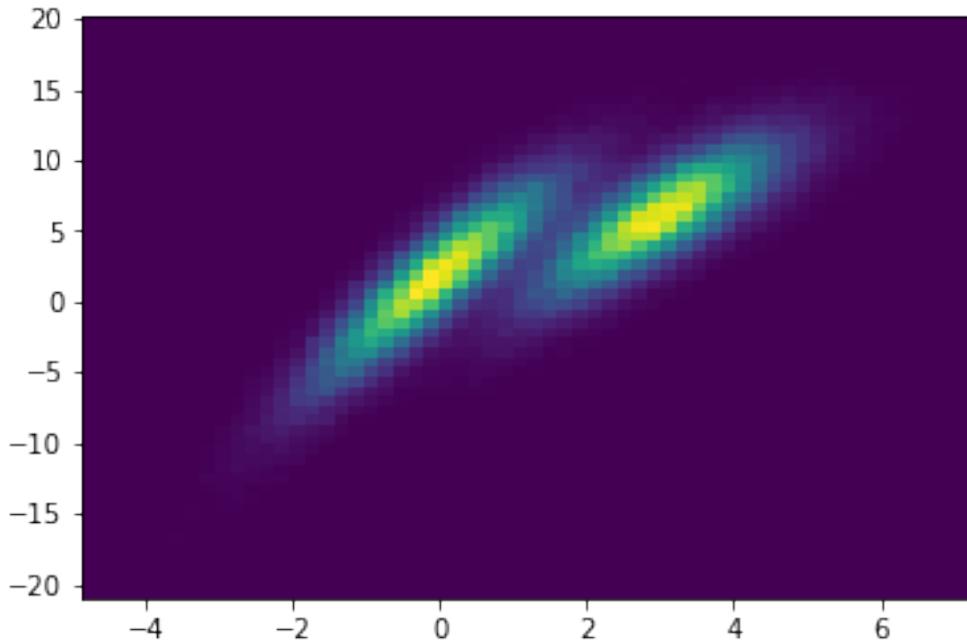
Acá la notación `r_[]` hace concatenación por filas. Veamos que forma tienen `x` e `y`

```
x.shape
```

```
(200000,)
```

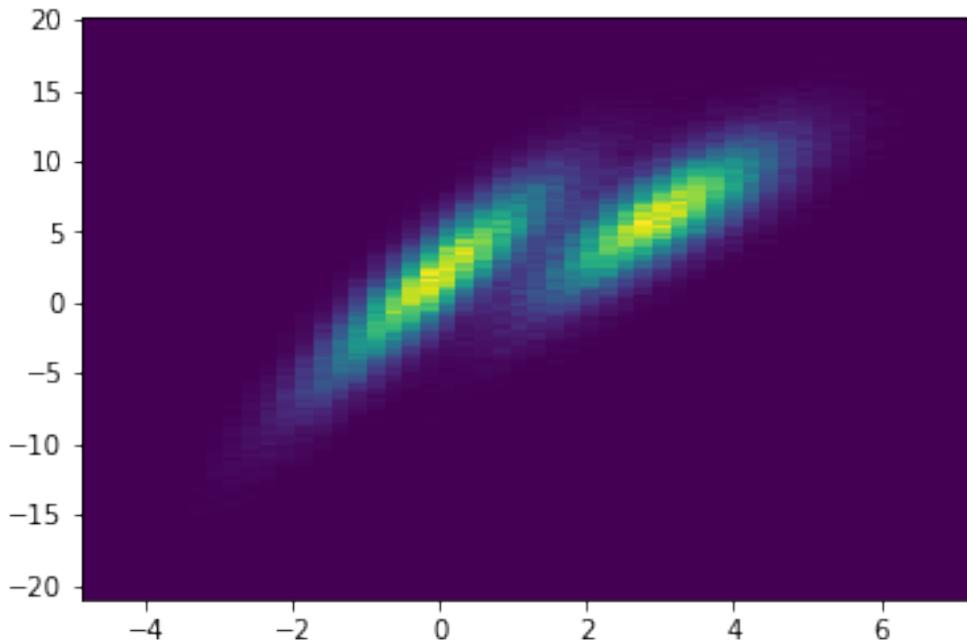
Para crear el histograma usamos simplemente la función `hist2d`. Que realiza la grilla acorde al argumento `bins` y luego calcula el histograma en dos dimensiones

```
#fig1= plt.figure()
H= plt.hist2d(x, y, bins=60)
```



Aquí pusimos igual número de cajas en cada dimensión. También podemos pasárselle un array con distinto número de cajas

```
fig1= plt.figure()  
H= plt.hist2d(x, y, bins=[50,150])
```



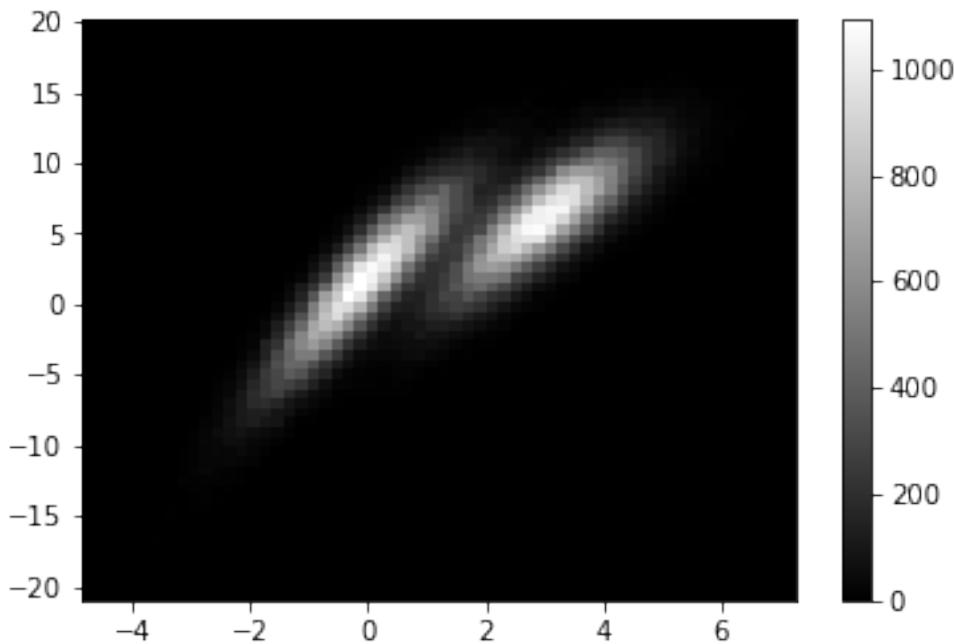
Por supuesto podemos cambiar el esquema de colores utilizado. Para ello le damos explícitamente el argumento `cmap` especificando el colormap deseado:

```
fig1= plt.figure()
```

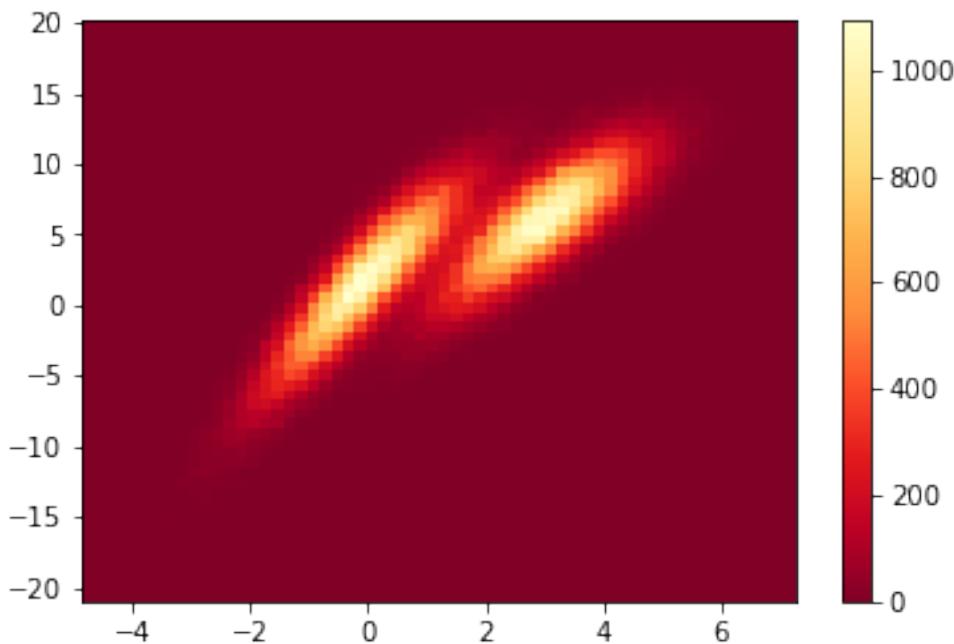
(continué en la próxima página)

(proviene de la página anterior)

```
H= plt.hist2d(x, y, bins=60, cmap='gray')
plt.colorbar();
```

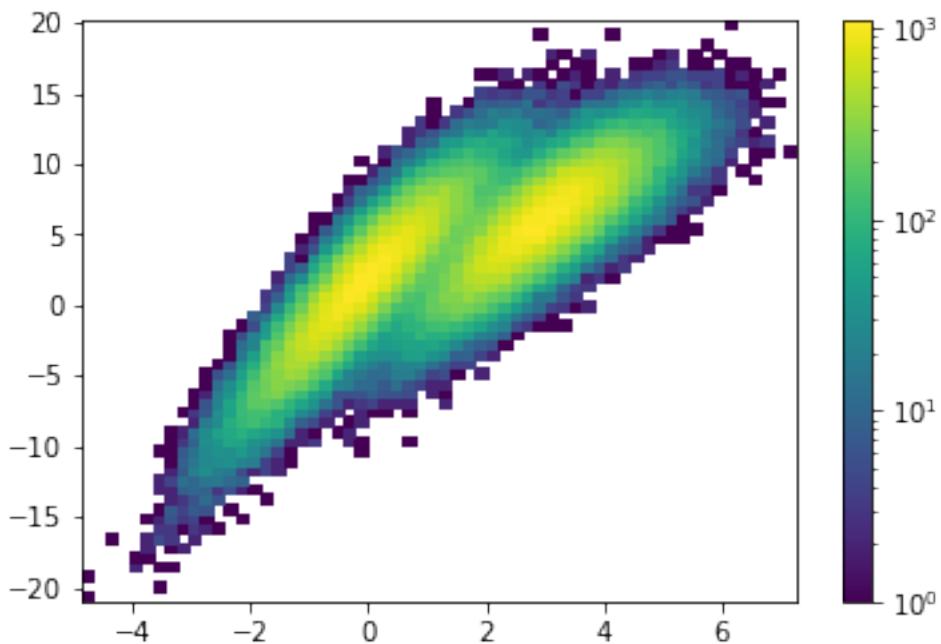


```
fig1= plt.figure()
H= plt.hist2d(x, y, bins=60, cmap=plt.cm.YlOrRd_r)
plt.colorbar();
```

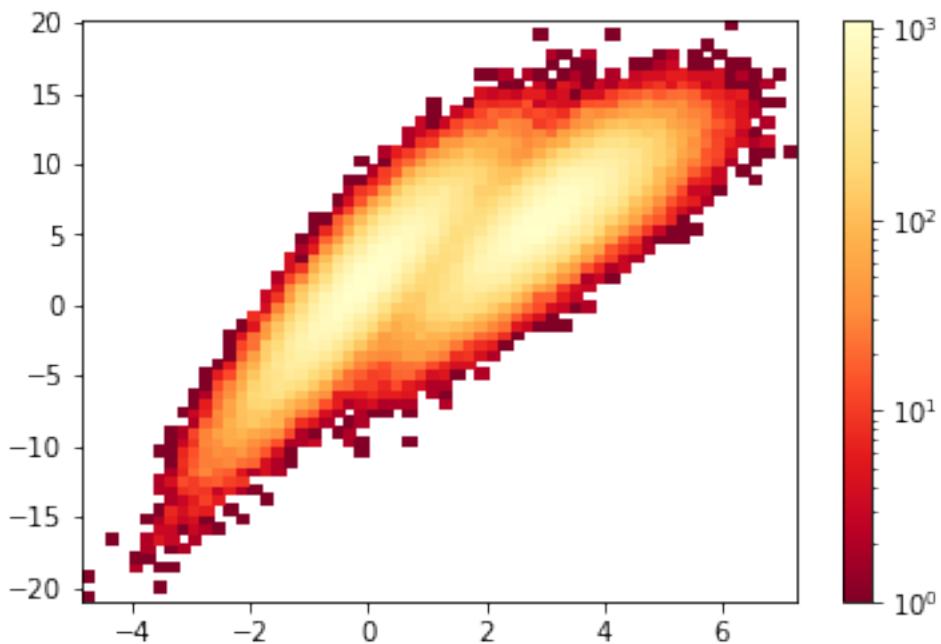


De la misma manera, si queremos realizarlo en escala logarítmica debemos pasarle una escala de colores adecuada

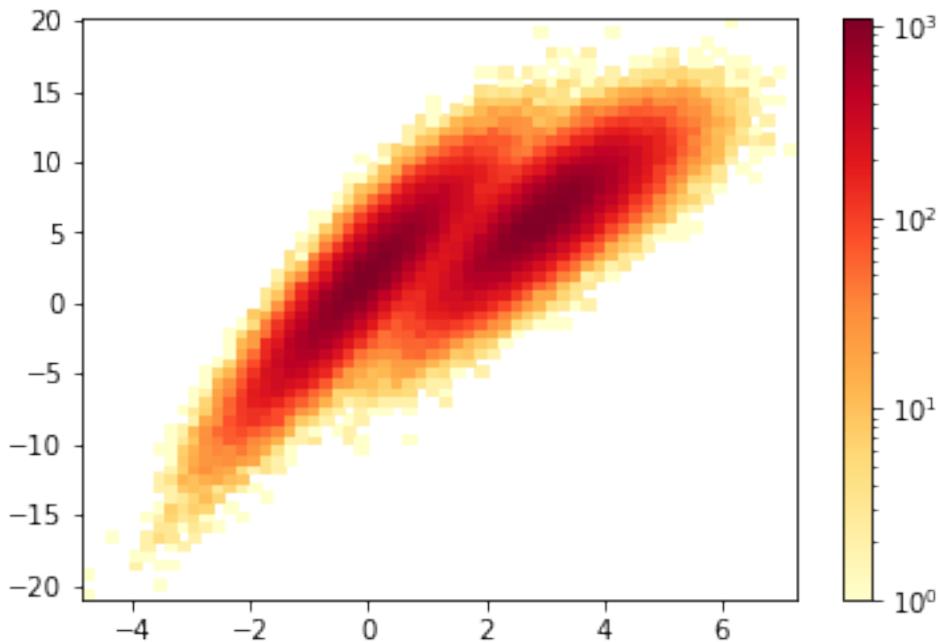
```
from matplotlib.colors import LogNorm
fig1= plt.figure()
H= plt.hist2d(x, y, bins=60, norm=LogNorm())
plt.colorbar();
```



```
from matplotlib.colors import LogNorm
fig1= plt.figure()
H= plt.hist2d(x, y, bins=60, cmap=plt.cm.YlOrRd_r, norm=LogNorm())
plt.colorbar();
```



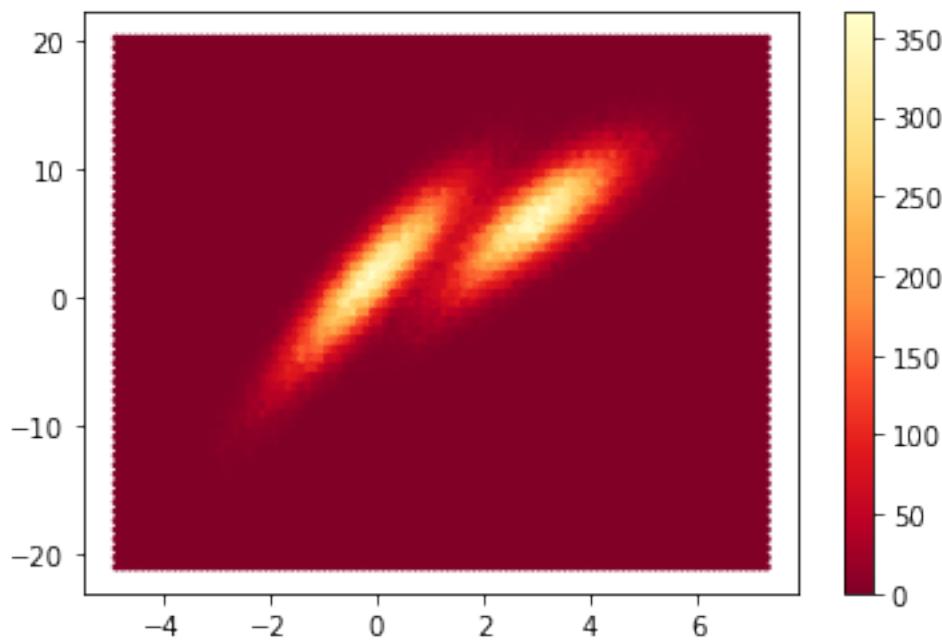
```
from matplotlib.colors import LogNorm
fig1= plt.figure()
H= plt.hist2d(x, y, bins=60, cmap=plt.cm.YlOrRd, norm=LogNorm())
plt.colorbar();
```



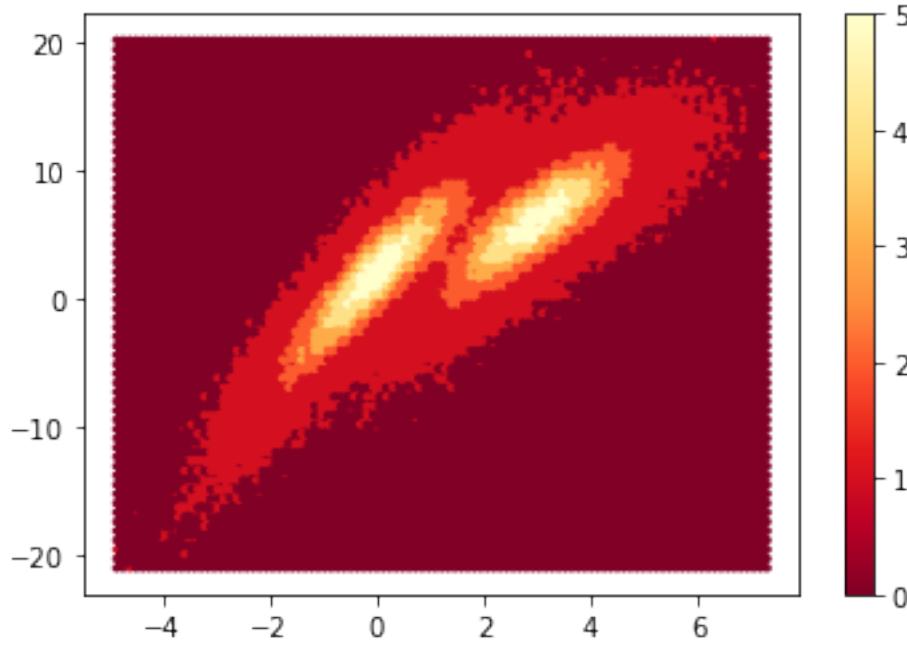
13.2.2 Histogramas con partición hexagonal

Vemos que `plt.hist()` realiza una partición rectangular del dominio. Puede ser más agradable a la vista realizar una partición en hexágonos. Para ello está la función `plt.hexbin()`

```
fig1= plt.figure()
plt.hexbin(x, y, cmap=plt.cm.YlOrRd_r)
plt.colorbar();
```

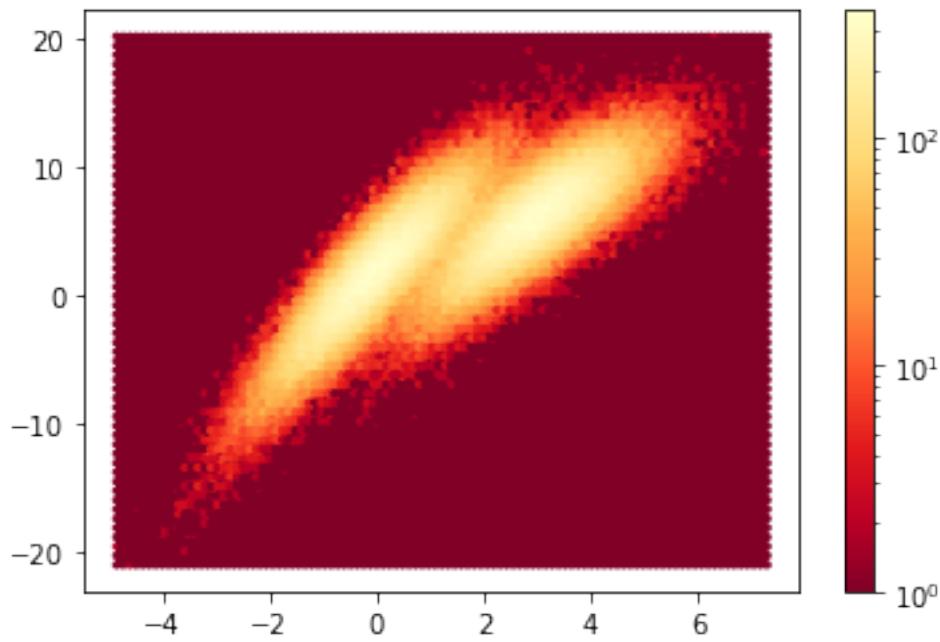


```
fig1= plt.figure()  
plt.hexbin(x, y, bins=6, cmap=plt.cm.YlOrRd_r)  
plt.colorbar();
```

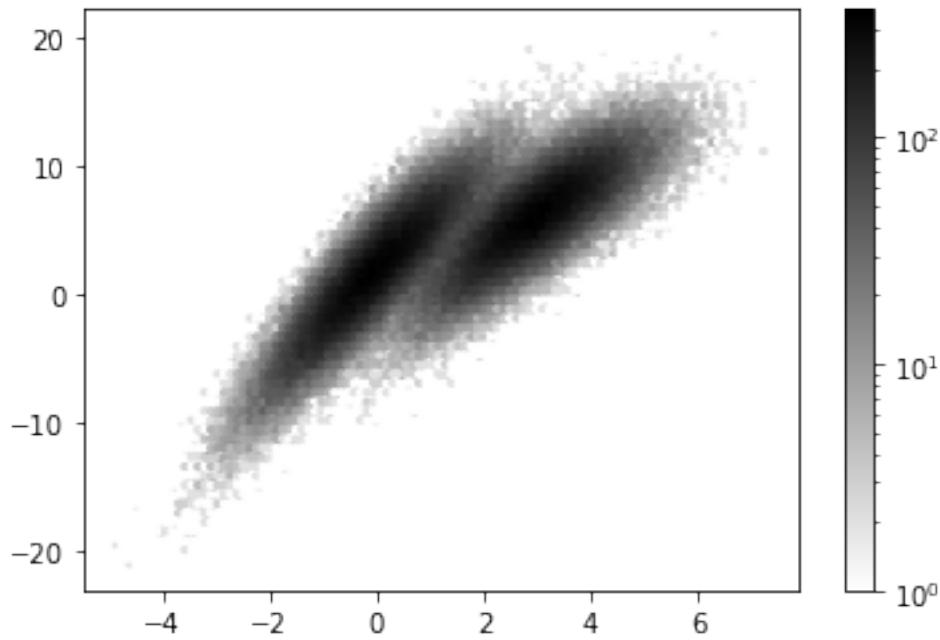


o, en escala logarítmica:

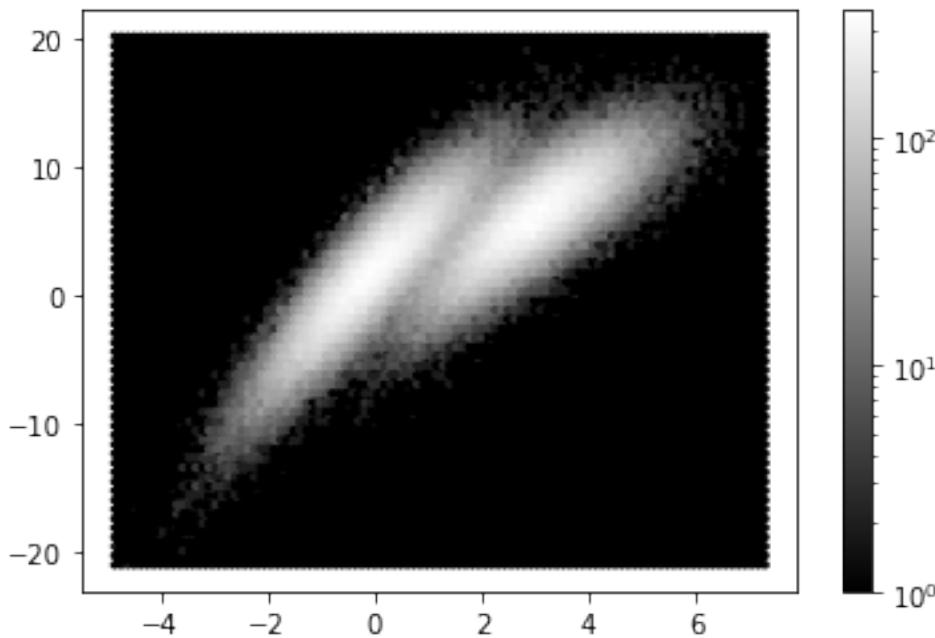
```
fig1= plt.figure()  
plt.hexbin(x, y, bins='log', cmap=plt.cm.YlOrRd_r)  
plt.colorbar();
```



```
fig1= plt.figure()
plt.hexbin(x, y, bins='log', cmap='gray_r')
plt.colorbar();
```



```
fig1= plt.figure()
plt.hexbin(x, y, bins='log', cmap='gray')
plt.colorbar();
```



13.2.3 Gráficos de contornos

```
H[0].shape, H[1].shape, H[2].shape
```

```
((60, 60), (61,), (61,))
```

```
H[2]
```

```
array([-20.97716804, -20.29168082, -19.60619359, -18.92070637,
       -18.23521914, -17.54973191, -16.86424469, -16.17875746,
       -15.49327024, -14.80778301, -14.12229579, -13.43680856,
       -12.75132133, -12.06583411, -11.38034688, -10.69485966,
       -10.00937243, -9.3238852 , -8.63839798, -7.95291075,
       -7.26742353, -6.5819363 , -5.89644908, -5.21096185,
       -4.52547462, -3.8399874 , -3.15450017, -2.46901295,
       -1.78352572, -1.09803849, -0.41255127,  0.27293596,
       0.95842318,  1.64391041,  2.32939763,  3.01488486,
       3.70037209,  4.38585931,  5.07134654,  5.75683376,
       6.44232099,  7.12780822,  7.81329544,  8.49878267,
       9.18426989,  9.86975712, 10.55524434, 11.24073157,
      11.9262188 , 12.61170602, 13.29719325, 13.98268047,
      14.6681677 , 15.35365493, 16.03914215, 16.72462938,
      17.4101166 , 18.09560383, 18.78109106, 19.46657828,
      20.15206551])
```

```
x0 = 0.5*(H[1][1:]+ H[1][:-1])
y0 = 0.5*(H[2][1:]+ H[2][:-1])
X, Y = np.meshgrid(x0, y0)
Z = H[0]
```

Clases de Python

```
X.shape, Y.shape, Z.shape
```

```
((60, 60), (60, 60), (60, 60))
```

```
np.all(X[0] == X[1])
```

```
True
```

```
np.all(Y[0] == Y[1])
```

```
False
```

```
np.all(Y[:,0] == Y[:,2])
```

```
True
```

```
X[0,:10], X[1,:10]
```

```
(array([-4.75096788, -4.54866832, -4.34636877, -4.14406921, -3.94176966,
       -3.7394701 , -3.53717055, -3.33487099, -3.13257144, -2.93027188]),
 array([-4.75096788, -4.54866832, -4.34636877, -4.14406921, -3.94176966,
       -3.7394701 , -3.53717055, -3.33487099, -3.13257144, -2.93027188]))
```

Nota: ¿Qué hace meshgrid aquí?

La función `meshgrid` crea matrices de coordenadas en n-dimensiones, basadas en n vectores unidimensionales de coordenadas

Consideremos un caso con vectores más simples:

```
x = np.arange(4, dtype='int')
y = np.arange(4, 7, dtype='int')
XX, YY = np.meshgrid(x, y)
```

La función `meshgrid` crea pares (x, y) iterando sobre cada valor de y para cada valor de x . En este caso para los vectores:

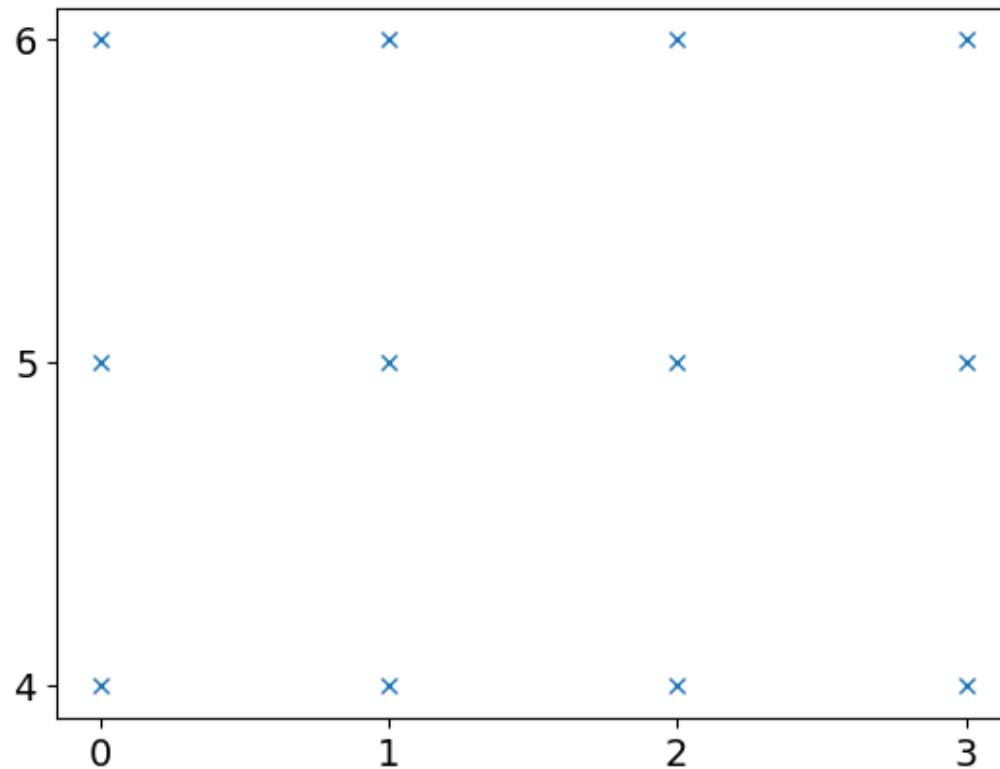
```
x = [0 1 2 3]
y = [4 5 6]
```

crea las matrices

```
XX = [[0 1 2 3]
      [0 1 2 3]
      [0 1 2 3]]
YY = [[4 4 4 4]
      [5 5 5 5]
      [6 6 6 6]]
```

que contiene todos los pares posibles (x, y), como se ve en la siguiente figura:

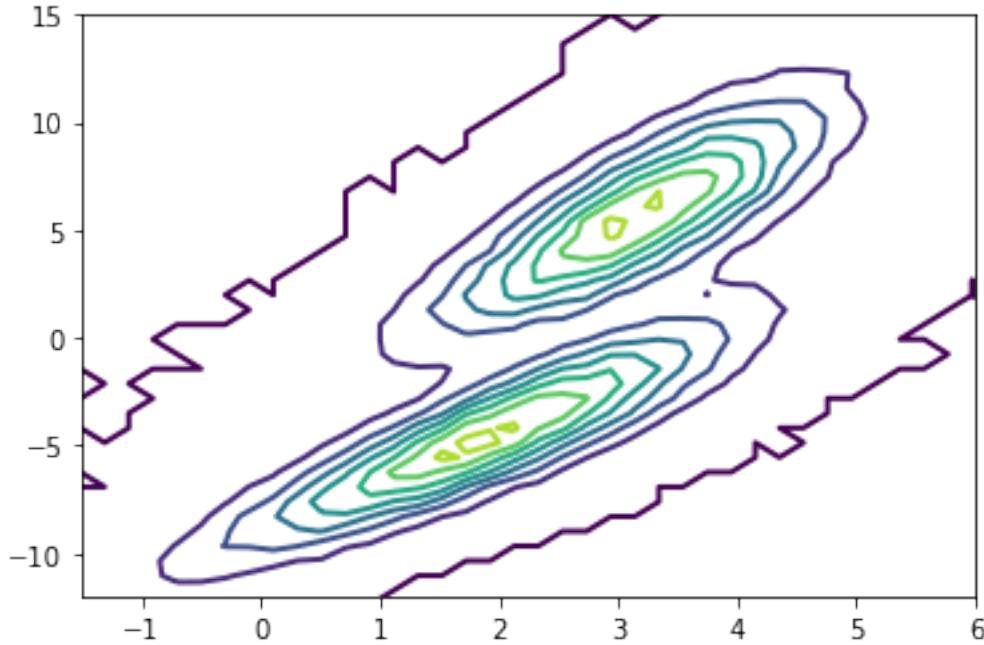
```
plt.plot(XX, YY, 'x', color='C0')
```



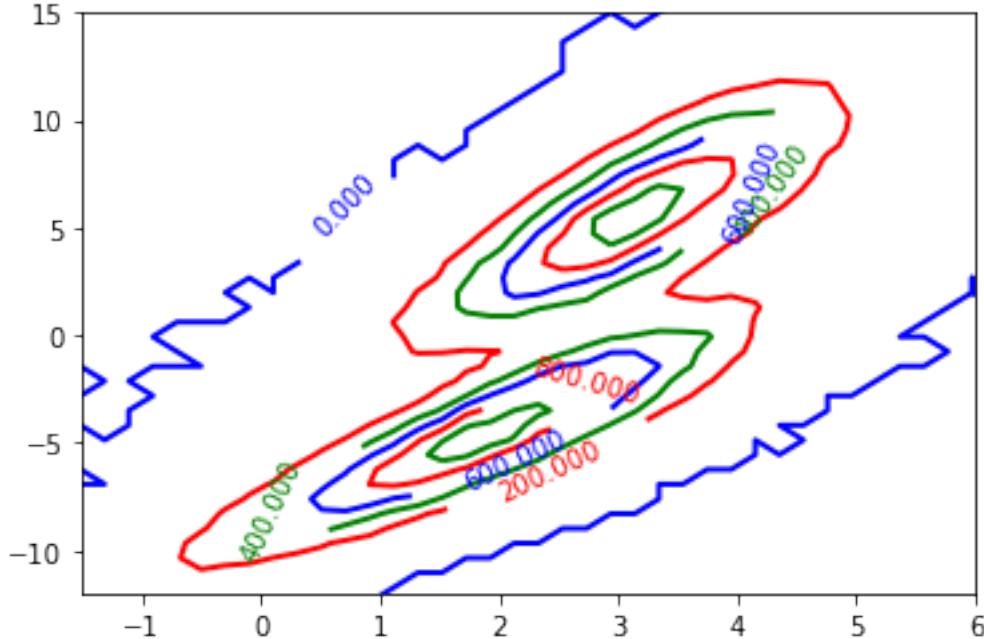
Vamos a usar los datos para hacer los gráficos de contornos

```
fig1= plt.figure()  
CS = plt.contour(X, Y, Z)  
plt.xlim((-1.5,6))  
plt.ylim((-12,15))
```

```
(-12, 15)
```



```
CS = plt.contour(X, Y, Z, 5, colors=('b','r','g'))
plt.clabel(CS, fontsize=10, inline=1)
plt.xlim((-1.5,6))
plt.ylim((-12,15));
```



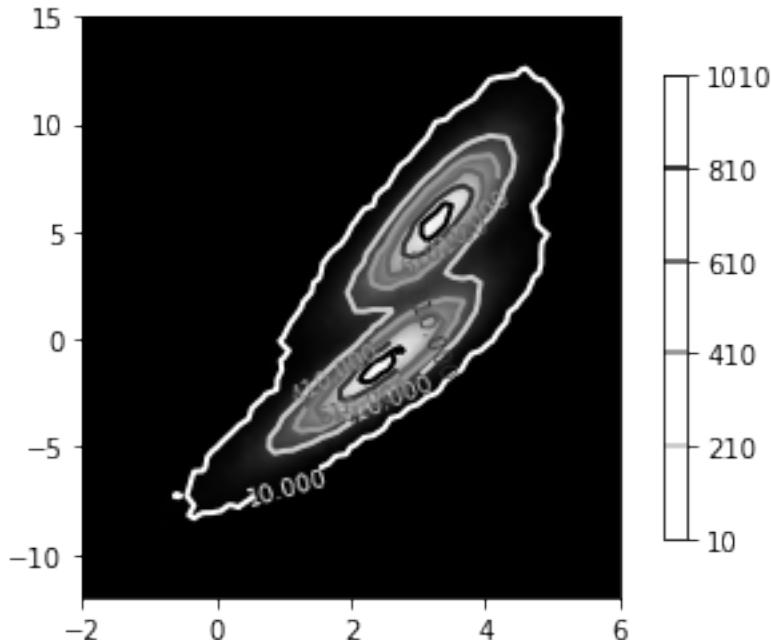
También podemos mostrar la imagen con los contornos superpuestos:

```
fig1= plt.figure()
im = plt.imshow(Z, interpolation='bilinear', origin='lower',
                 extent=(-2,6,-12,15), aspect=0.32, cmap=plt.cm.gray)
```

(continué en la próxima página)

(proviene de la página anterior)

```
levels = np.arange(10, 1210, 200)
CS = plt.contour(Z, levels, origin='lower', linewidths=2,
                  extent=(-2,6,-12,15), cmap=plt.cm.gray_r )
plt.clabel(CS, fontsize=9, inline=1)
CB = plt.colorbar(CS, shrink=0.8)
```



13.2.4 Superficies y contornos

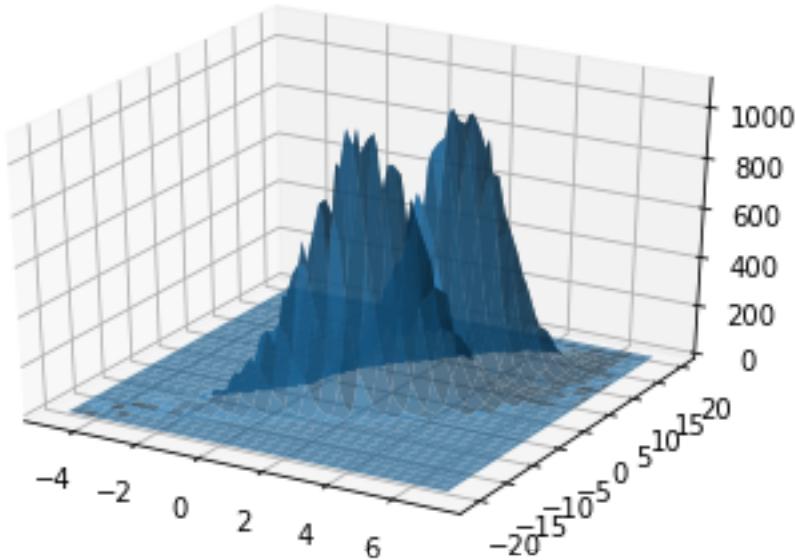
Superficies

Para realizar gráficos realmente en 3D debemos importar Axes3D

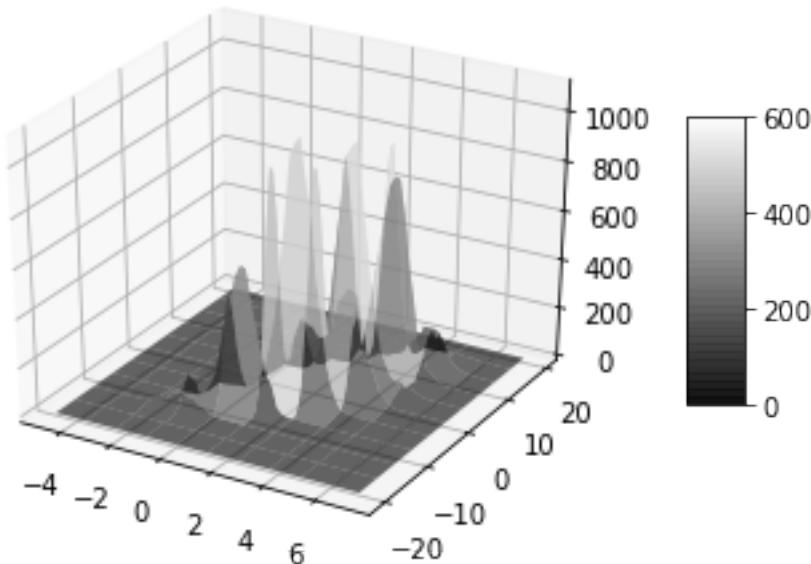
```
from mpl_toolkits.mplot3d import Axes3D
# plt.style.use('classic')
```

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z, alpha=.7)
```

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7ff77ca10c50>
```



```
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.gray, rstride=6, cstride=6, lw=1, alpha=0.
    ↪)
ax.set_zlim(0, 1100.)
fig.colorbar(surf, shrink=0.5, aspect=5);
```



```
help(ax.plot_surface)
```

Help on method `plot_surface` in module `mpl_toolkits.mplot3d.axes3d`:

```
plot_surface(X, Y, Z, args, norm=None, vmin=None, vmax=None, lightsource=None,
    ↪ **kwargs) method of matplotlib.axes._subplots.Axes3DSubplot instance
Create a surface plot.
```

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the `*cmap` argument.

.. note::

The `rcount` and `ccount` kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

Parameters

`X, Y, Z` : 2d arrays
Data values.

`rcount, ccount` : int

Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Defaults to 50.

.. versionadded:: 2.0

`rstride, cstride` : int

Downsampling stride in each direction. These arguments are mutually exclusive with `rcount` and `ccount`. If only one of `rstride` or `cstride` is set, the other defaults to 10.

'classic' mode uses a default of `rstride = cstride = 10` instead of the new default of `rcount = ccount = 50`.

`color` : color-like

Color of the surface patches.

`cmap` : Colormap

Colormap of the surface patches.

`facecolors` : array-like of colors.

Colors of each individual patch.

`norm` : Normalize

Normalization for the colormap.

`vmin, vmax` : float

Bounds for the normalization.

`shade` : bool

Whether to shade the facecolors. Defaults to True. Shading is always disabled when `cmap` is specified.

`lightsource` : `~matplotlib.colors.LightSource`

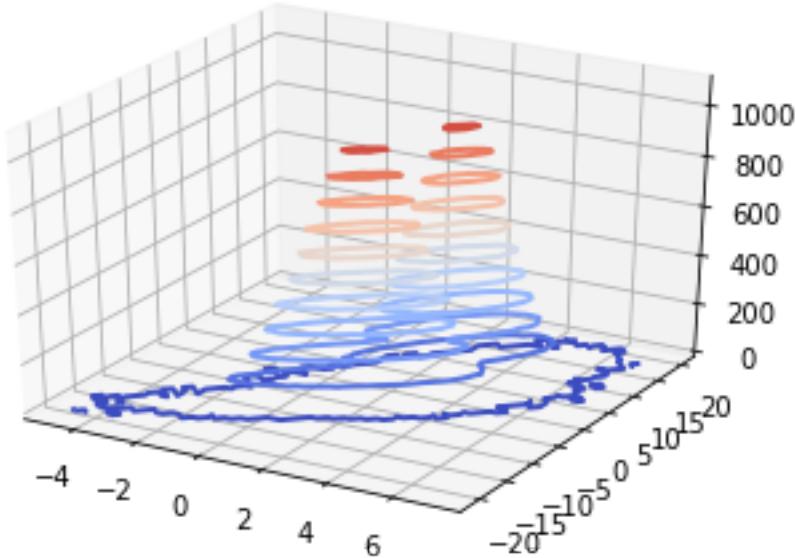
The lightsource to use when `shade` is True.

`**kwargs`

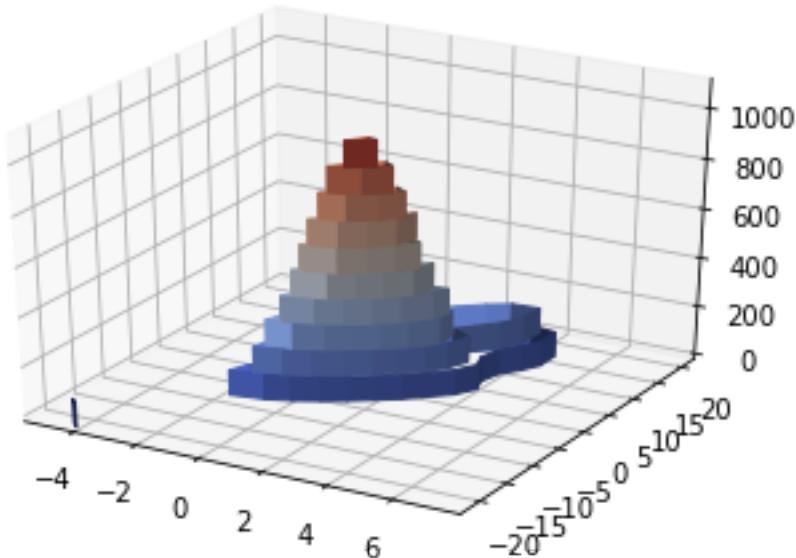
Other arguments are forwarded to `.Poly3DCollection`.

Contornos en 3D

```
fig = plt.figure()
ax = fig.gca(projection='3d')
cset = ax.contour(X, Y, Z, 10, cmap=plt.cm.coolwarm)
```



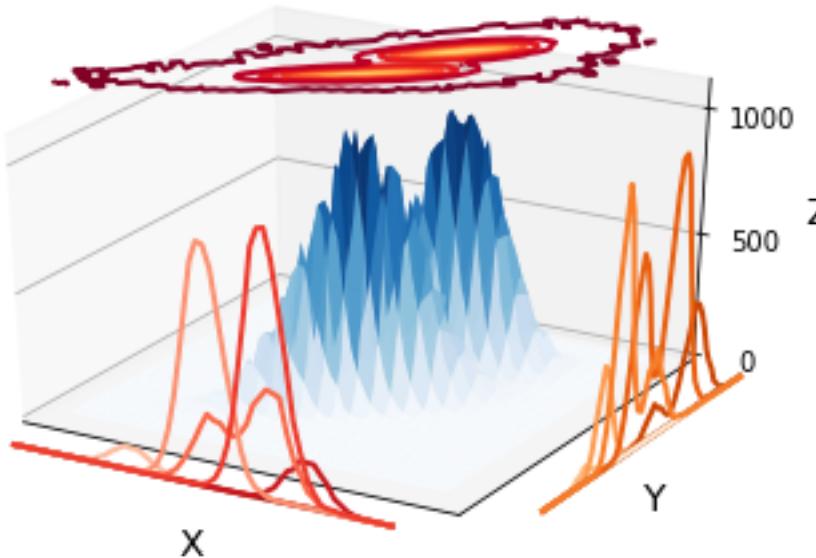
```
fig = plt.figure()
ax = fig.gca(projection='3d')
cset = ax.contour(X, Y, Z, 10, extend3d=True, cmap=plt.cm.coolwarm)
```



```

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z, lw=0, alpha=0.9, cmap=plt.cm.Blues)
cset = ax.contour(X, Y, Z, zdir='z', offset=1300, cmap=plt.cm.YlOrRd_r)
cset = ax.contour(X, Y, Z, zdir='x', offset=10, cmap=plt.cm.Oranges)
cset = ax.contour(X, Y, Z, zdir='y', offset=-30, cmap=plt.cm.Reds)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_xticks([])
ax.set_yticks([])
ax.set_zticks([0, 500, 1000]);

```

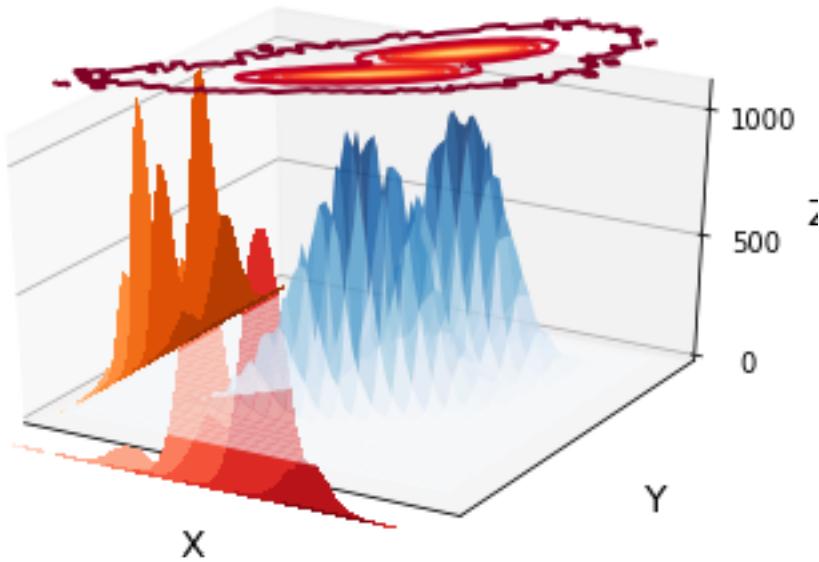


Veamos otro ejemplo. Modifiquemos el gráfico anterior para llenar los contornos laterales, utilizando la función `contourf()` en lugar de `contour()`

```

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z, lw=0, alpha=0.6, cmap=plt.cm.Blues)
cset = ax.contour(X, Y, Z, zdir='z', offset=1300, cmap=plt.cm.YlOrRd_r)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-5, cmap=plt.cm.Oranges)
cset = ax.contourf(X, Y, Z, zdir='y', offset=-30, cmap=plt.cm.Reds)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_xticks([])
ax.set_yticks([])
ax.set_zticks([0, 500, 1000]);

```

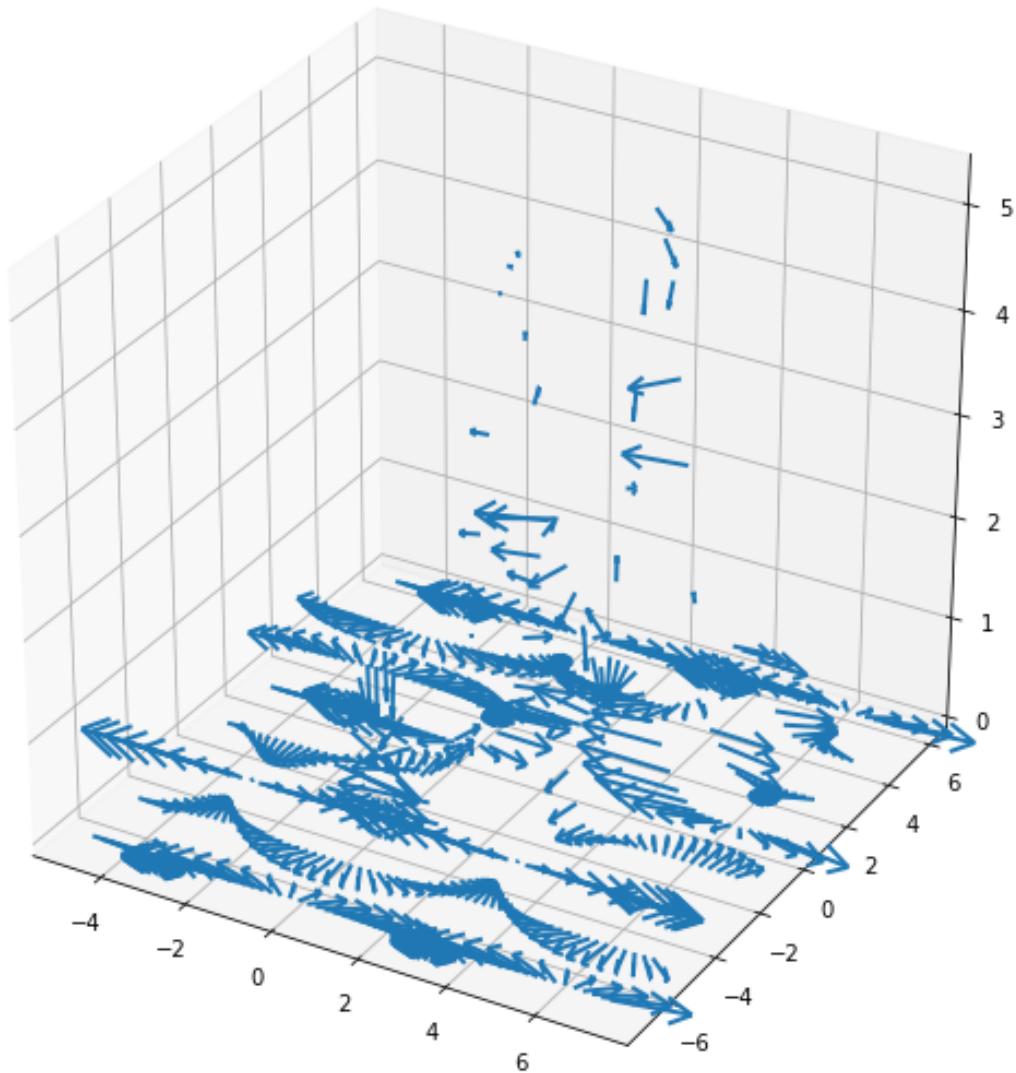


Gráficos de campos vectoriales

Para realizar gráficos de campos (de velocidades, fuerzas, etc) podemos utilizar la función `quiver()`, que grafica flechas en cada punto, con una dirección y longitud dada

```
fig = plt.figure(figsize=(10,10))
ax = fig.gca(projection='3d')
d=8
x = X[::d]
y = Y[::d]/3
z = Z[::d]/200
# Creamos las direcciones para cada punto
u = 5*np.sin(x) * np.cos(y) * np.cos(z)
v = - 2* np.cos( x) * np.sin( y) * np.cos(z)
w = np.cos(x) * np.cos(y) * np.sin(z)

ax.quiver(x, y, z, u, v, w, length=0.5, arrow_length_ratio=0.3);
```



Veamos un ejemplo de la documentación de **Matplotlib**

```
fig = plt.figure(figsize=(9,9))
ax = fig.gca(projection='3d')

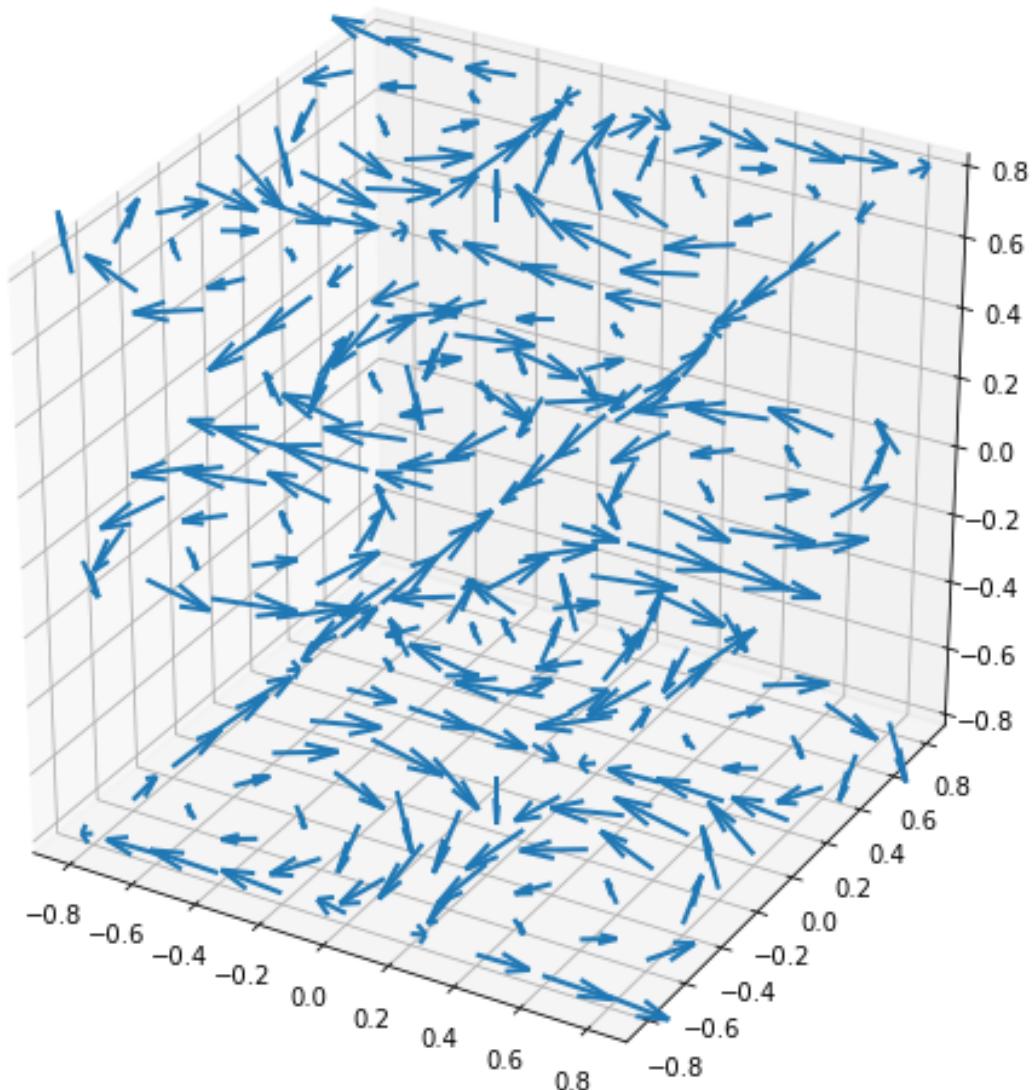
# Make the grid
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))

# Make the direction data for the arrows
u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0)) * np.cos(np.pi * x) * np.cos(np.pi * y) *
    np.sin(np.pi * z))
```

(continué en la próxima página)

(provien de la página anterior)

```
ax.quiver(x, y, z, u, v, w, length=0.3, arrow_length_ratio=0.5);
```



Más información sobre este tipo de gráficos en http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html

CAPÍTULO 14

Clase 13: Interpolación y ajuste de curvas (fiteo)

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('presentation')
fsize= (9,6)
```

14.1 Interpolación

Muchas veces tenemos mediciones de datos variando algún parámetro en las condiciones, y estos datos están medidos a intervalos mayores de los que deseamos. En estos casos es común tratar de inferir los valores que tendrían las mediciones para valores intermedios de nuestro parámetro. Una opción es interpolar los datos. Algunas facilidades para ello están en el subpaquete **interpolate** del paquete **Scipy**.

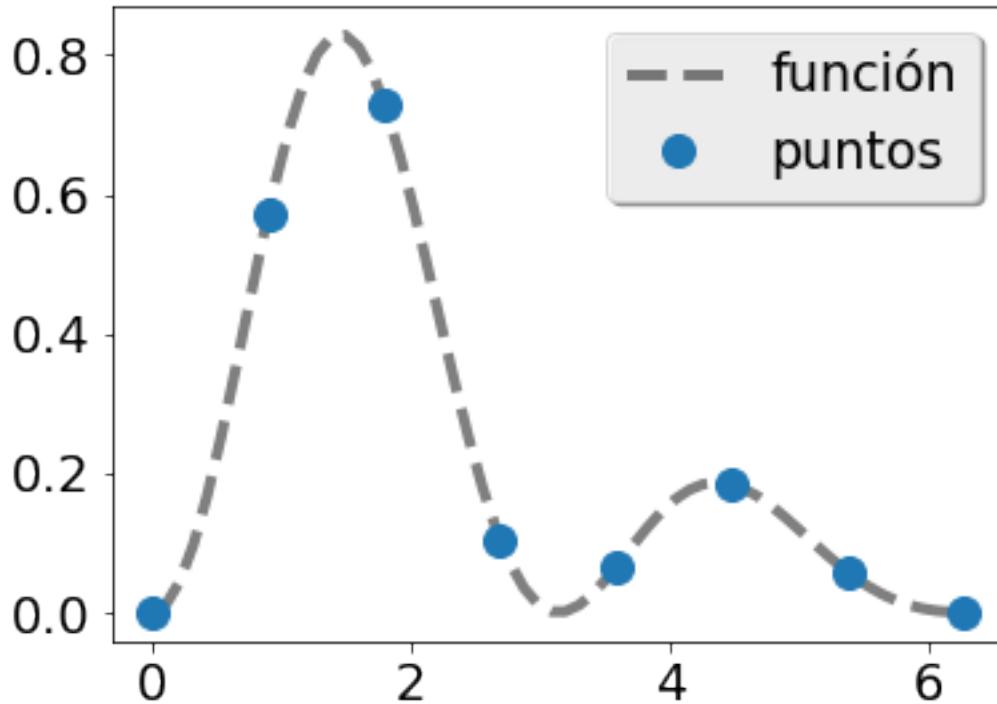
Generemos algunos datos experimentales

```
def fmodel(x):
    return (np.sin(x) **2*np.exp(-(x/3.5)**2)
```

```
x0 = np.linspace(0., 2*np.pi, 60)
y0 = fmodel(x0)
x = np.linspace(0., 2*np.pi, 8)
y = fmodel(x)
```

```
plt.plot(x0,y0,'--k', label='función', alpha=0.5)
plt.plot(x,y,'o', markersize=12, label='puntos')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7faba251dbd0>
```



Acá hemos simulado datos con una función oscilante con un decaimiento exponencial.

Ahora, importamos el submódulo `interpolate` del módulo `scipy`, adecuado para interpolar:

```
from scipy import interpolate
```

La interpolación funciona en dos pasos. En el primer paso realizamos todos los cálculos y obtenemos la función interpolante, y en una segunda etapa utilizamos esa función para interpolar en los nuevos puntos sobre el eje x que necesitamos.

Utilizamos los *arrays* `x` e `y` como los pocos datos experimentales obtenidos

```
print("x = {}".format(x))
```

```
x = [0.          0.8975979  1.7951958  2.6927937  3.5903916  4.48798951
 5.38558741  6.28318531]
```

y creamos la función interpolante basada en estos puntos:

```
interpol_lineal = interpolate.interp1d(x, y)
```

```
interpol_lineal # función
```

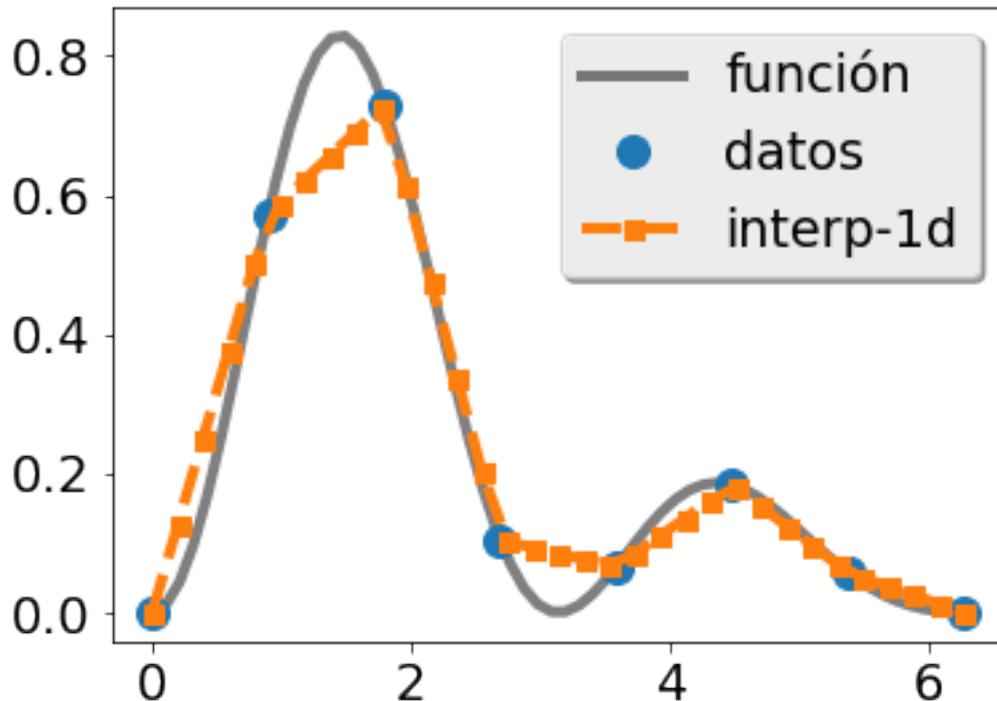
```
<scipy.interpolate.interpolate.interp1d at 0x7faba291d350>
```

Ahora, creamos un conjunto de puntos `x1` donde queremos evaluar la función interpolando entre datos medidos

```
x1 = np.linspace(0, 2*np.pi, 33)
y1_l = interpol_lineal(x1)
```

```
plt.plot(x0,y0, '-k', label='función', alpha=0.5)
plt.plot(x, y,'o', markersize=12, label='datos')
plt.plot(x1, y1_l,'--s', markersize=7, label='interp-1d')
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7faba2458710>
```



Como vemos, la función que creamos consiste de tramos rectos entre los puntos datos.

Para realizar interpolaciones lineales (una recta entre pares de puntos) también se puede utilizar la rutina `interp()` del módulo **Numpy**, cuyos argumentos requeridos son: los nuevos puntos `x1` donde queremos interpolar, además de los valores originales de `x` y de `y` de la tabla a interpolar:

```
y1_12= np.interp(x1,x,y)
```

Notar que `y1_12` da exactamente los mismos valores que `y1_1`

```
np.all(y1_12 == y1_1)
```

```
True
```

Si bien el uso de `np.interp` es más directo, porque no se crea la función intermedia, cuando creamos la función con `interp1d` podemos aplicarla a diferentes conjuntos de valores de `x`:

```
interpol_lineal(np.linspace(0, 2*np.pi, 12))
```

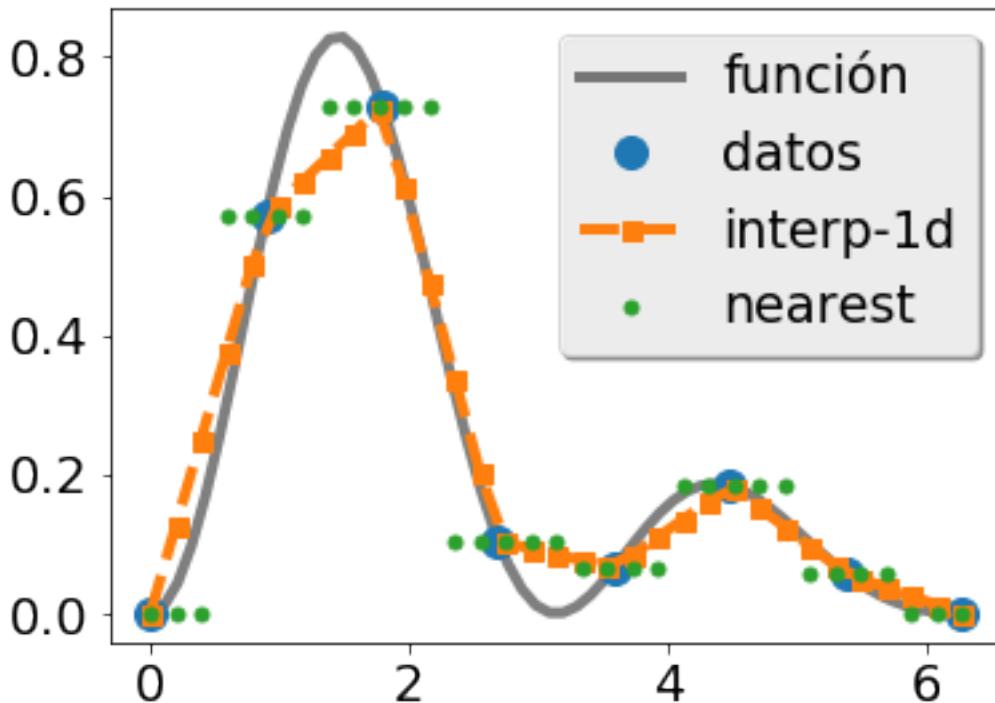
```
array([0.0000000e+00, 3.64223643e-01, 6.15515285e-01, 7.16230928e-01,
       3.88910634e-01, 9.71667086e-02, 7.27120078e-02, 1.19301459e-01,
      1.72109481e-01, 9.17229560e-02, 3.64455561e-02, 2.39038977e-33])
```

La interface `interp1d()` tiene un argumento opcional, `kind`, que define el tipo de interpolación a utilizar. Cuando utilizamos el argumento `nearest` utiliza para cada valor el más cercano

```
interpol_near = interpolate.interp1d(x, y, kind='nearest')
y1_n = interpol_near(x1)
```

```
plt.plot(x0,y0, '-k', label='función', alpha=0.5)
plt.plot(x, y, 'o', markersize=12, label='datos')
plt.plot(x1, y1_l,'--s', markersize=7, label='interp-1d')
plt.plot(x1, y1_n,'.', label='nearest')
plt.legend(loc='best');
print(x1.size, x1.size, x.size)
```

```
33 33 8
```



14.1.1 Interpolación con polinomios

Scipy tiene rutinas para interpolar los datos usando un único polinomio global con grado igual al número de puntos dados:

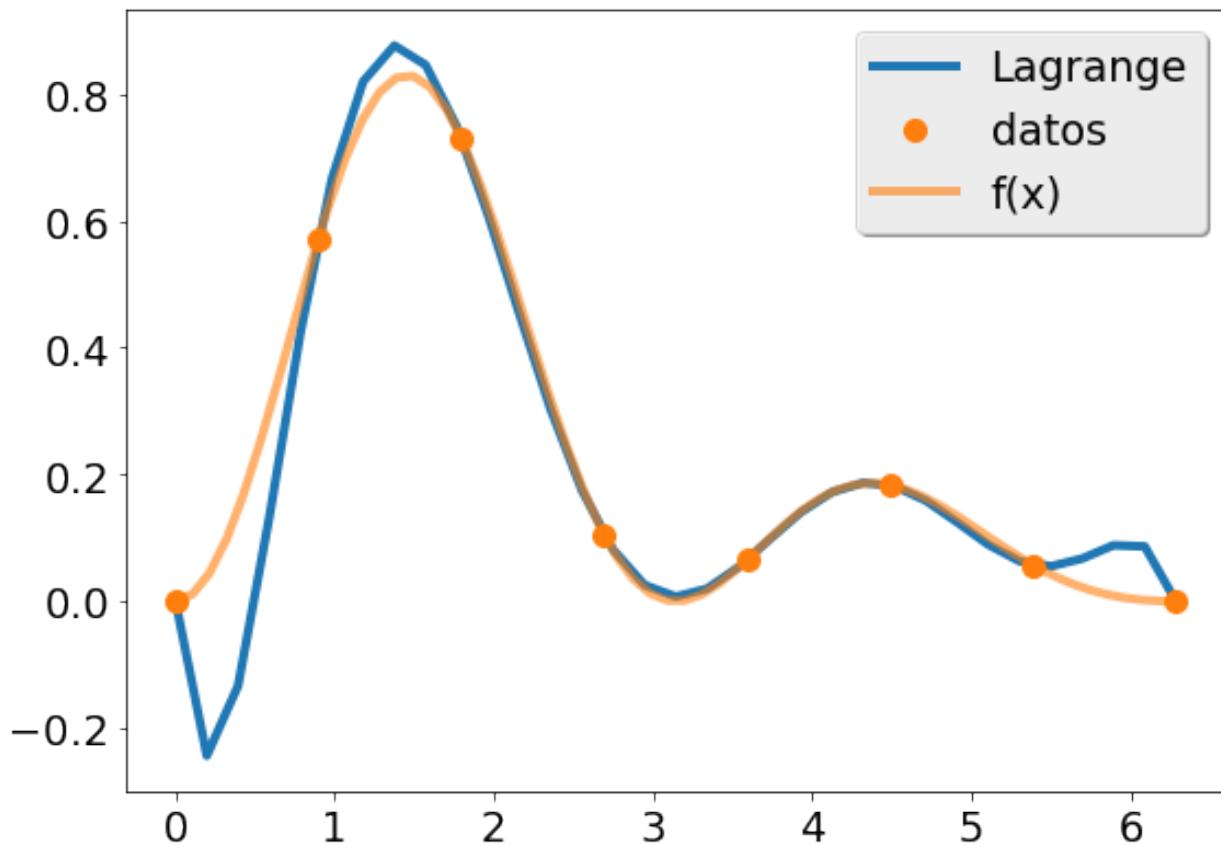
```
f = interpolate.lagrange(x, y)
y2 = f(x1)
plt.figure(figsize=fsize)
plt.plot(x1,y2,'-', label='Lagrange')
plt.plot(x,y,'o', label='datos')
```

(continuó en la próxima página)

(proviene de la página anterior)

```
plt.plot(x0,y0,'-', color='C1', label='f(x)', alpha=0.7)
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7fabaf57590>
```



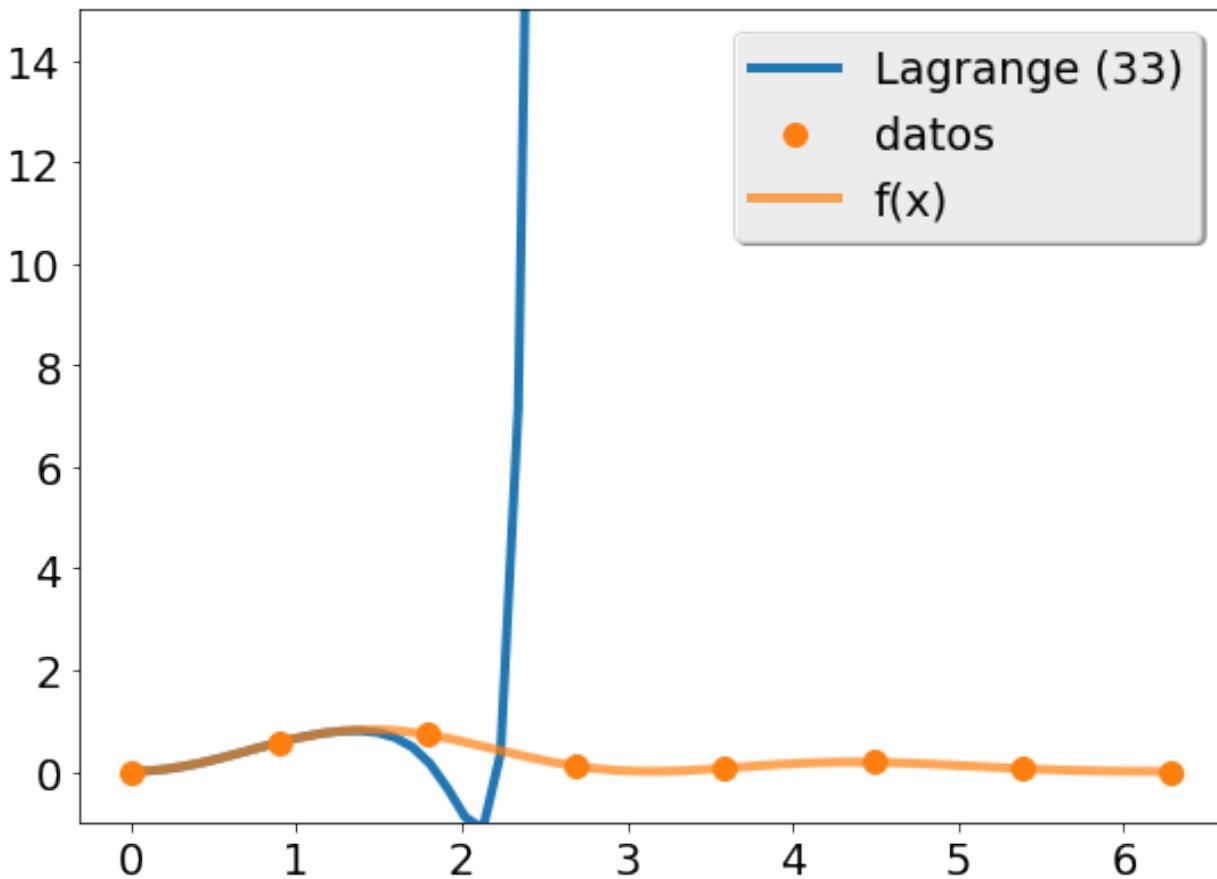
Los polinomios interpolantes pueden tener problemas, principalmente en las puntas, o cuando el grado del polinomio es muy alto. Consideremos por ejemplo el caso donde tenemos una tabla $x_1 \ f(x_1)$ con muchos datos y queremos interpolar sobre una nueva tabla de valores x_0

```
print('Número de datos:', x1.size)
```

```
Número de datos: 33
```

```
plt.figure(figsize=fsize)
f1 = interpolate.lagrange(x1, fmodel(x1))
plt.plot(x0,f1(x0),'-', label='Lagrange ({})'.format(x1.size))
plt.plot(x,y,'o', label='datos')
plt.plot(x1,fmodel(x1),'-', color='C1', label='f(x)', alpha=0.7)
plt.ylim((-1,15))
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7fabaf00e90>
```



De todas maneras, para los casos en que es aplicable, existen dos implementaciones: `interpolate.lagrange()` y una segunda llamada `interpolate.barycentric_interpolate()` que está basada en un trabajo de 2004 y es numéricamente más estable.

14.1.2 Splines

- Las *Splines* son interpolaciones por polinomios de a trazos, que se eligen para que no sólo los valores sino también sus derivadas coincidan dando una curva suave.
- Para eso, si se pide que la aproximación coincida con los valores tabulados en los puntos dados, la aproximación es efectivamente una **interpolación**.
- Cubic Splines* se refiere a que utilizamos polinomios cúbicos en cada trozo.

El argumento opcional `kind` de la interface `interp1d()`, que define el tipo de interpolación a utilizar, acepta valores del tipo `string` que pueden ser: `linear`, `nearest`, `zero`, `slinear`, `quadratic`, `cubic`, o un número entero indicando el orden.

```
interp = {}
for k in ['zero', 'slinear', 'quadratic', 'cubic']:
    interp[k] = interpolate.interp1d(x,y, kind=k)
```

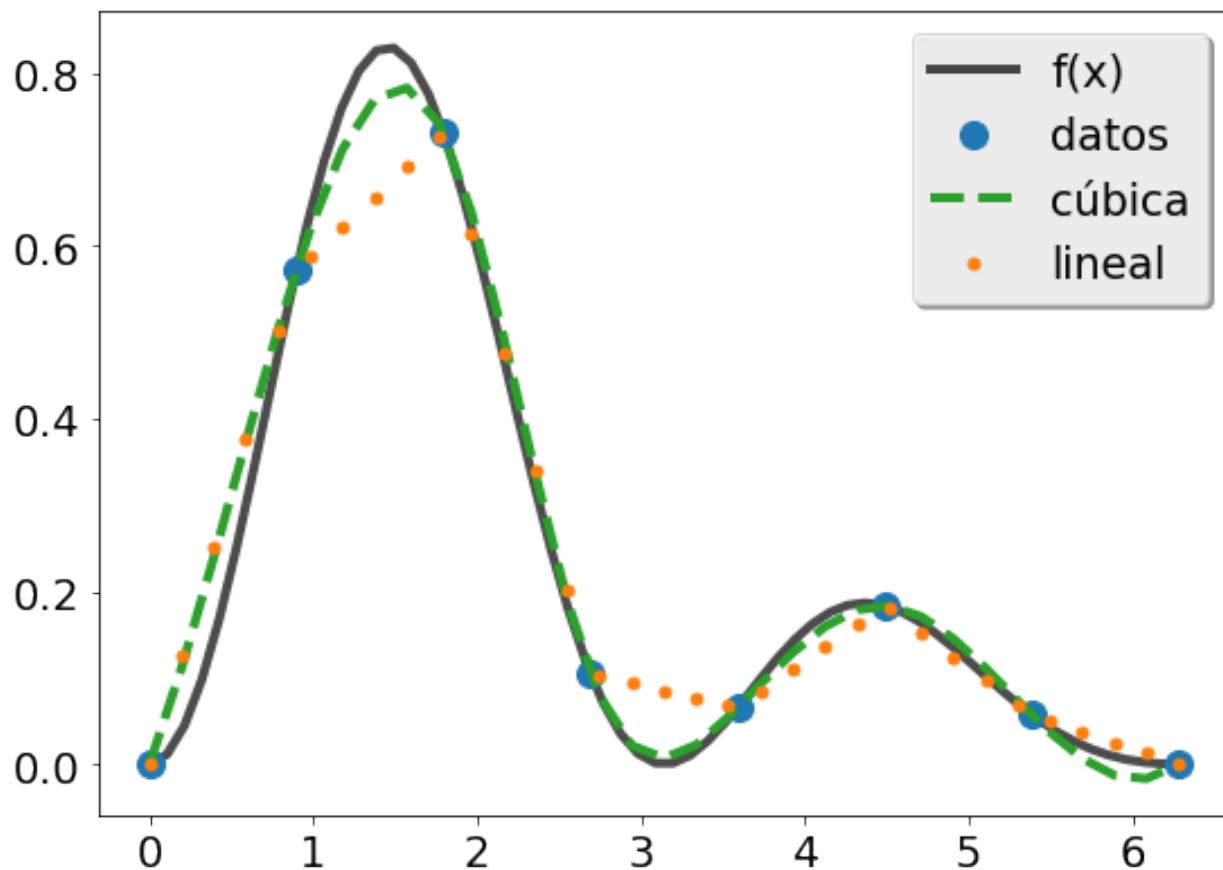
```
fig= plt.figure(figsize=fsiz)
plt.plot(x0,y0,'-k', alpha=0.7, label='f(x)')
plt.plot(x,y,'o', markersize=12, label='datos')
```

(continué en la próxima página)

(proviene de la página anterior)

```
plt.plot(x1,interp['cubic'](x1), '--', color='C2', label=u'cúbica')
plt.plot(x1,interp['slinear'](x1), '.', label='lineal')
plt.legend(loc='best')
```

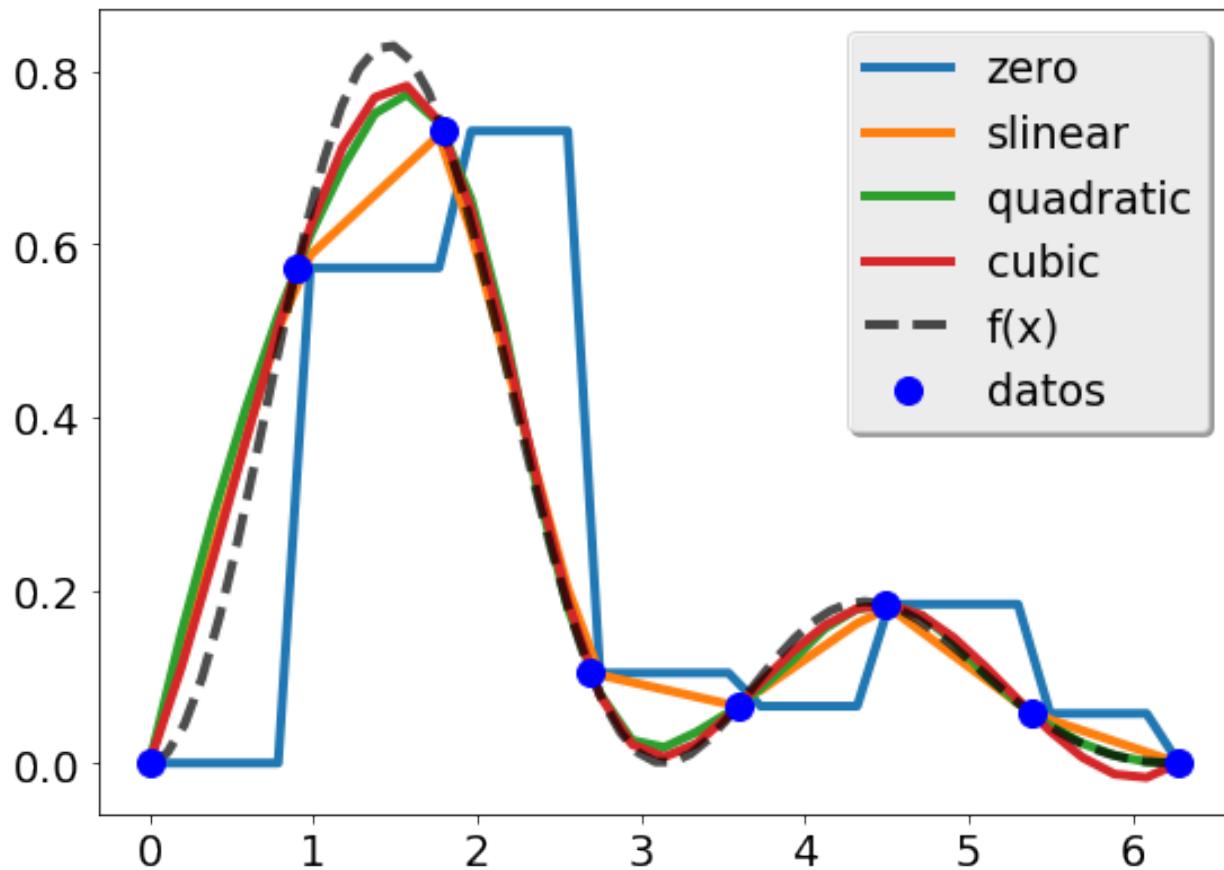
<matplotlib.legend.Legend at 0x7fababd14510>



Tratamos de incluir todo en un sólo gráfico (y rogamos que se entienda algo)

```
plt.figure(figsize=fsize)
for k, v in interp.items():
    plt.plot(x1, v(x1), label=k)
plt.plot(x0,y0,'--k', alpha=0.7, label='f(x)')
plt.plot(x,y,'ob', markersize=12, label='datos')
plt.legend(loc='best')
```

<matplotlib.legend.Legend at 0x7fabab8e710>



En resumen, los métodos disponibles en `interpolate.interp1d` son:

- `linear`: Interpolación lineal, utilizando rectas (default)
- `nearest` : Valor constante correspondiente al dato más cercano
- `zero` o `0` : Una spline de orden cero. Toma el valor a la izquierda
- `slinear` o `1` : Spline de orden 1. Igual a `linear`
- `quadratic` o `2` : Spline de segundo orden
- `cubic` o `3` : Spline de tercer orden

Como vemos de los argumentos `zero`, `slinear`, `quadratic`, `cubic` para especificar splines de cero, primer, segundo y tercer orden se puede pasar como argumento un número. En ese caso se utiliza siempre `splines` y el número indica el orden de las splines a utilizar:

```
for k,s in zip([0,1,2,3], ['zero','slinear','quadratic','cubic']):
    num = interpolate.interp1d(x,y, kind=k)
    tipo = interpolate.interp1d(x,y, kind=s)
    print("{} == {}? -> {}".format(k,s,np.allclose(num(x1), tipo(x1))))
```

```
f0 == zero? -> True
f1 == slinear? -> True
f2 == quadratic? -> True
f3 == cubic? -> True
```

Además La interpolación lineal simple es, en la práctica, igual a la interpolación por splines de primer orden:

```
np.allclose(interp['slinear'](x1), interpol_lineal(x1)) # También son iguales
```

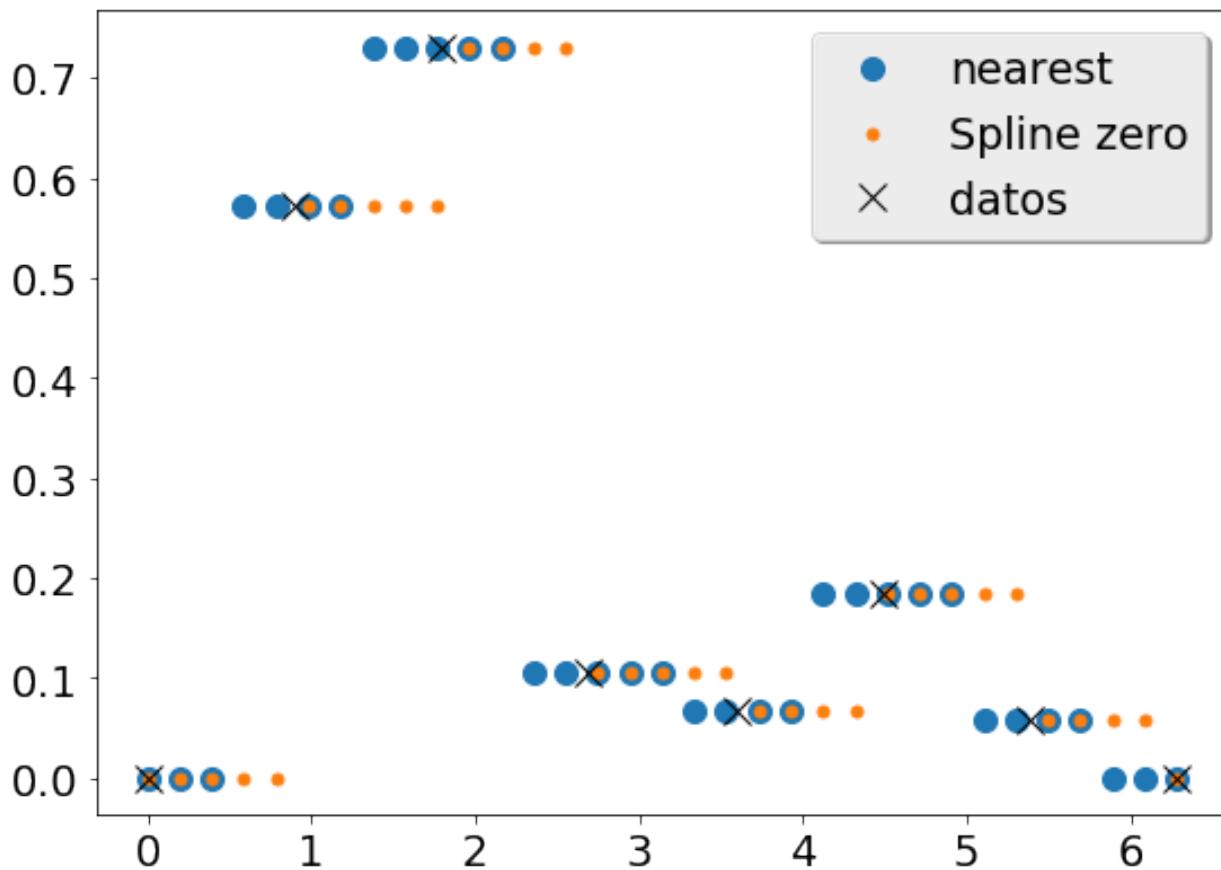
```
True
```

Finalmente, veamos que la interpolación nearest toma para cada nuevo valor x_1 el valor $y_1(x_1)$ igual a $y(x_0)$ donde x_0 es el valor más cercano a x_1 mientras que la spline de orden cero (zero) toma el valor más cercano a la izquierda:

```
alfa=1
plt.figure(figsize=fsize)
plt.plot(x1, y1_n, 'o', label='nearest', alpha=alfa)
plt.plot(x1, interp['zero'](x1), '.', label='Spline zero'.format(k), alpha=alfa)
plt.plot(x,y,'xk', markersize=12, label='datos')

plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7faba07aaf10>
```



El submódulo `signal` tiene rutinas adicionales para realizar *splines*, que permiten agregar un alisado, pero en este caso ya no interpolan estrictamente sino que puede ser que la aproximación no pase por los puntos dados.

14.1.3 B-Splines

Hay otra opción para realizar interpolación con Splines en Scipy. Las llamadas **B-Splines** son funciones diseñadas para generalizar polinomios, con un alto grado de **localidad**.

Para definir las **B-Splines** necesitamos dos cosas:

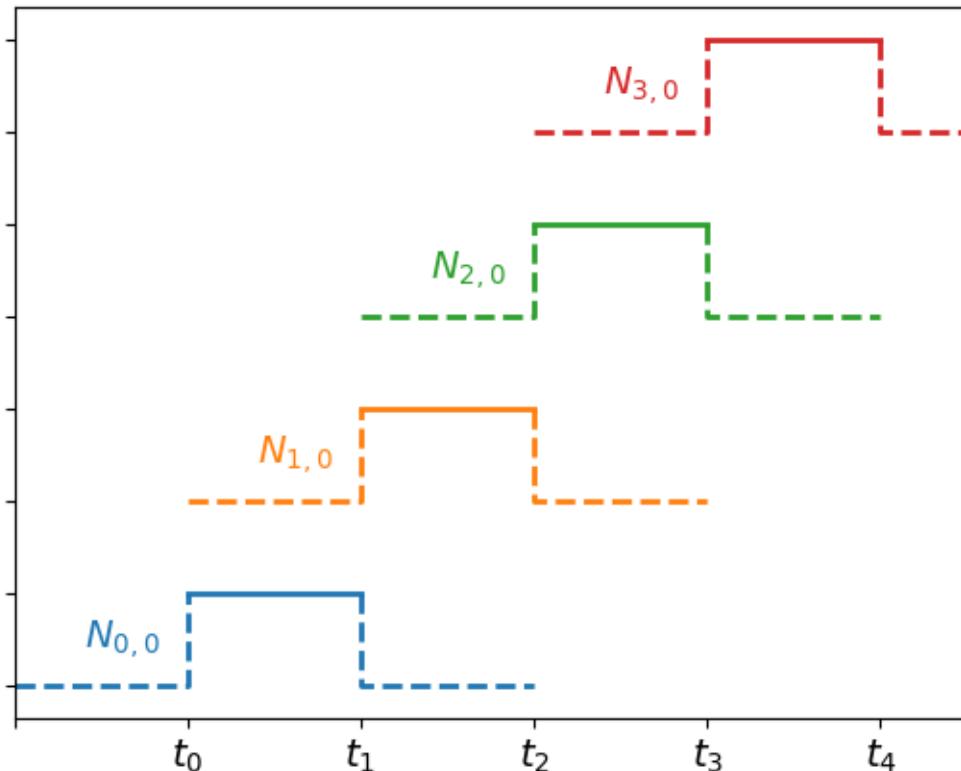
1. Elegir el grado de los polinomios (mayor o igual a 0)
2. Dividir el intervalo en n nodos

Las funciones se definen mediante la recursión:

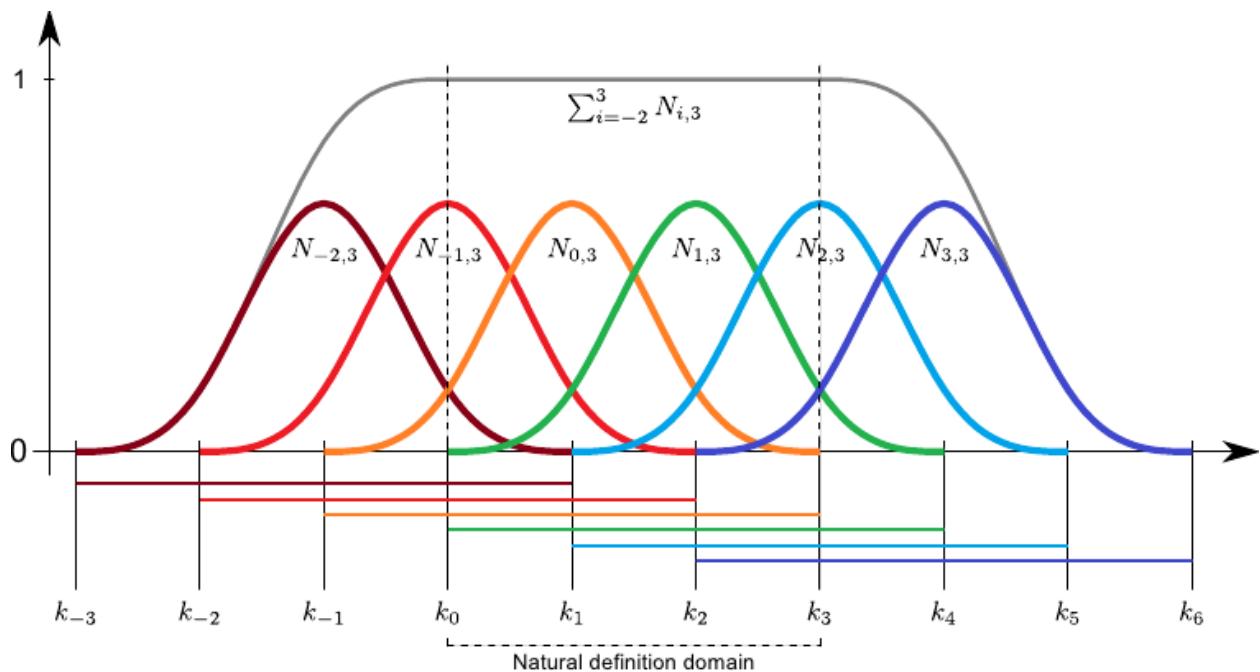
$$N_{i,0}(x) = 1, \quad \text{si } t_i \leq x < t_{i+1}, \quad \text{sino } 0,$$

$$N_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} N_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} N_{i+1,k-1}(x)$$

Las más simples, cuando el orden es $k=0$, son funciones constantes a trozos



Para $k > 0$ las funciones se calculan por recurrencia en término de dos funciones del orden anterior. Entonces, siempre serán diferentes de cero sólo en un intervalo finito. En ese intervalo presentan un único máximo y luego decaen suavemente. Las más usuales son las de orden $k = 3$:



(Figura de <http://www.brnt.eu/phd>)

La idea es encontrar una función $f(x)$ que sea suave y pase por la tabla de puntos (x, y) dados, o cerca de ellos con la condición que tanto la función como algunas de sus derivadas sea suave. La función $f(x)$ se describe como una expansión en la base de Splines (y de ahí el nombre *B-Splines*)

$$f(x) = \sum_{j=0} a_{i,j} B_j(x) \quad \forall \quad x_i < x \leq x_{i+1}$$

La aproximación se elige de tal manera de optimizar el número de elementos de la base a utilizar con la condición que el error cuadrático a los puntos sea menor a un cierto valor umbral s

$$\sum_{i=1}^n |f(x_i) - y_i|^2 \leq s$$

Veamos cómo usar la implementación de **Scipy** para interpolar datos. En primer lugar creamos la representación en B-Splines de nuestra tabla de datos (x, y) :

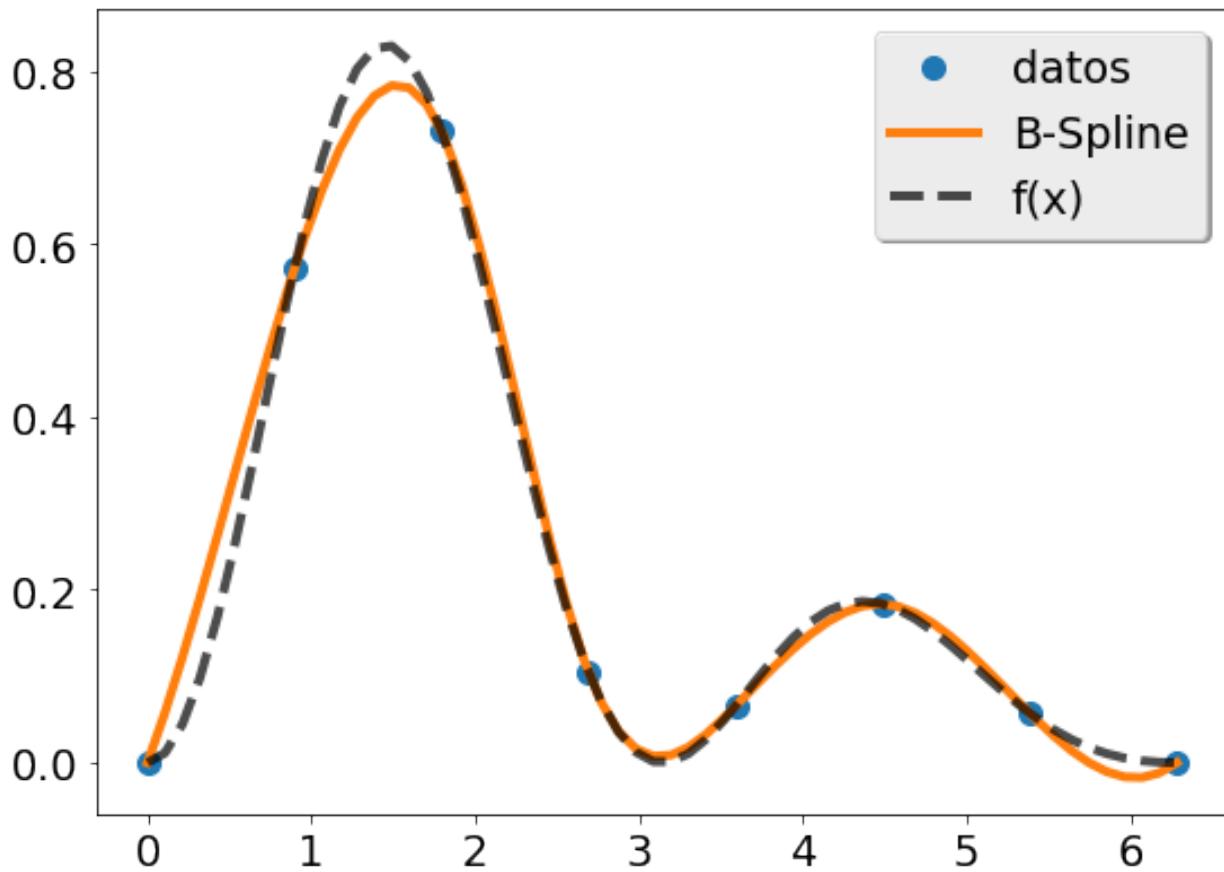
```
tck0 = interpolate.splrep(x, y)
```

Acá, otra vez estamos operando en dos pasos. En el primero creamos la representación de las splines para los datos dados. Como no pusimos explícitamente el orden, utiliza el valor por default $k=3$.

En el segundo paso obtenemos los valores interpolados sobre la grilla $x2$:

```
x2 = np.linspace(0, 2*np.pi, 60) # Nuevos puntos donde interpolar
y_s0 = interpolate.splev(x2, tck0) # Valores interpolados: y_s0[j] = f(x2[j])
```

```
plt.figure(figsize=fsiz)
plt.plot(x,y,'o', markersize=12, label='datos')
plt.plot(x2,y_s0,'-', label=r'B-Spline')
plt.plot(x0,y0,'--k', alpha=0.7, label='f(x)')
plt.legend(loc='best');
```



Estas funciones interpolan los datos con curvas continuas y con derivadas segundas continuas.

14.1.4 Lines are guides to the eyes

Sin embargo, estas rutinas no necesariamente realizan *interpolación* en forma estricta, pasando por todos los puntos dados, sino que en realidad realiza una aproximación minimizando por cuadrados mínimos la distancia a la tabla de puntos dados.

Esto es particularmente importante cuando tenemos datos que tienen dispersión. En esos casos necesitamos curvas que no interpolen, es decir que *no necesariamente* pasen por todos los puntos.

La rutina `splrep` tiene varios argumentos opcionales. Entre ellos un parámetro de suavizado `s` que corresponde a la condición de distancia entre la aproximación y los valores de la tabla mencionado anteriormente.

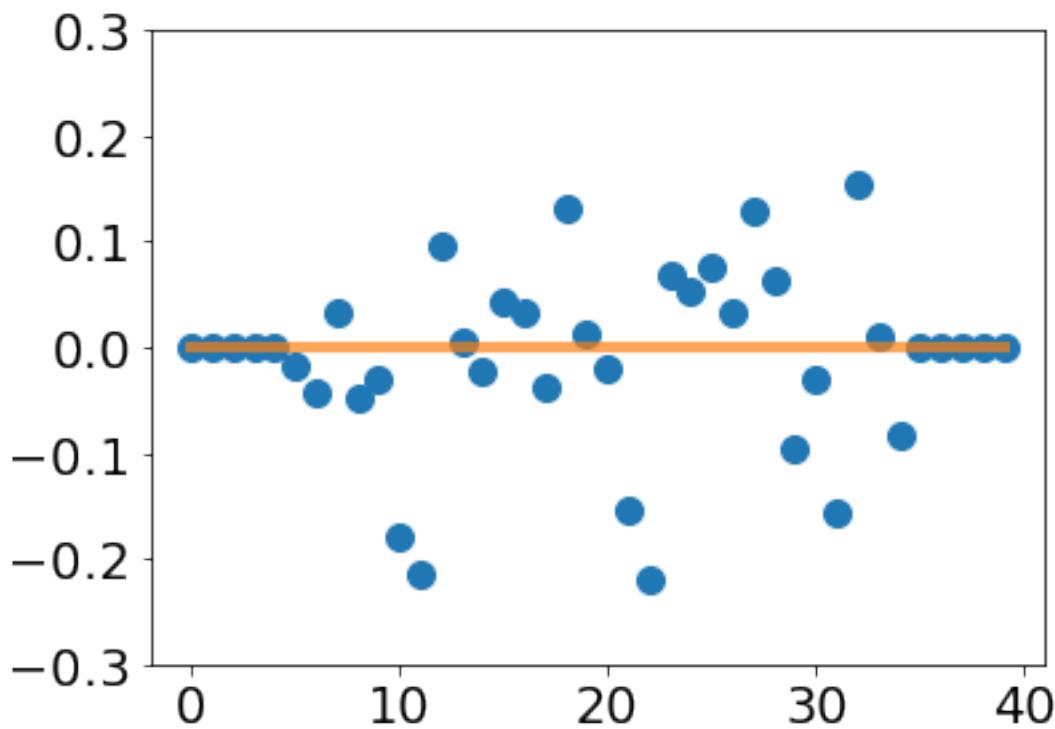
Para ver como funciona, creamos una tabla de valores x , y con $x \in [0, 2\pi]$ **no necesariamente equiespaciados**, $y = \sin(x)/2$, donde le agregamos algo de ruido a y

```
# Creamos dos tablas de valores x, y
x3 = np.linspace(0., 2*np.pi, 40)
x4 = np.linspace(0., 2*np.pi, 40)
x3[5:-5] -= 0.5*(0.5-np.random.random(30)) # Le agregamos una separación al azar
x3.sort() # Los ordenamos
plt.plot(x3-x4, 'o', label='data')
```

(continué en la próxima página)

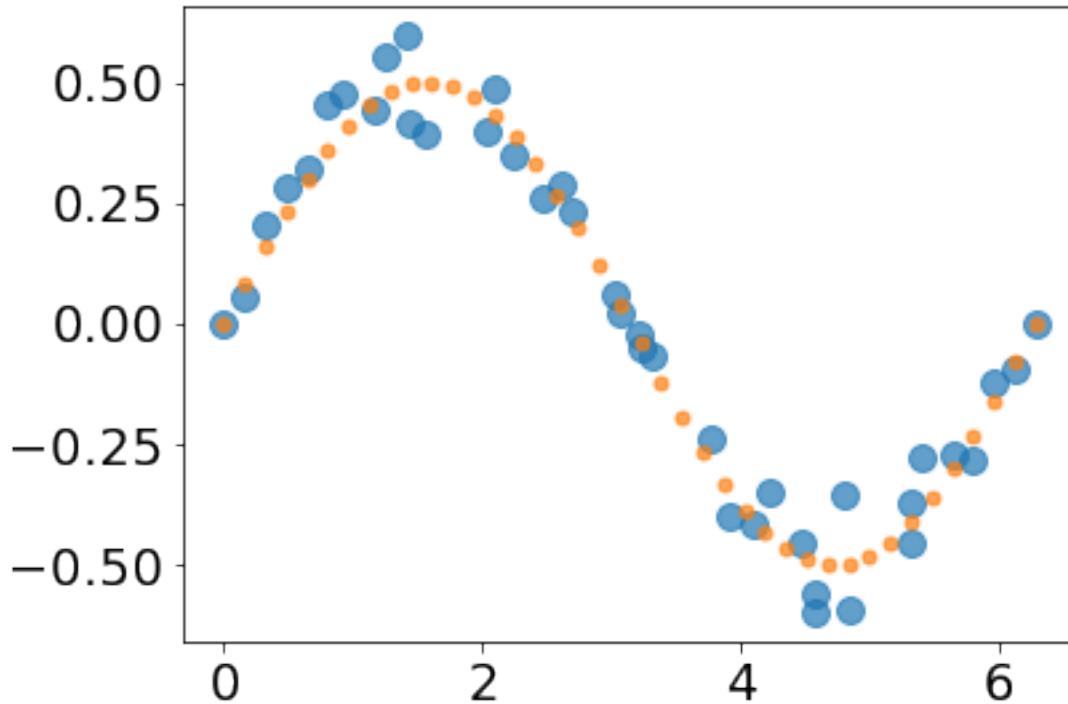
(provienec de la página anterior)

```
plt.plot(x3-x3, '-', alpha=0.7, label='sin incerteza')
plt.ylim((-0.3,0.3));
```



```
y4 = 0.5* np.sin(x4)
y3 = 0.5* np.sin(x3) * (1+ 0.6*(0.5-np.random.random(x3.size)))
```

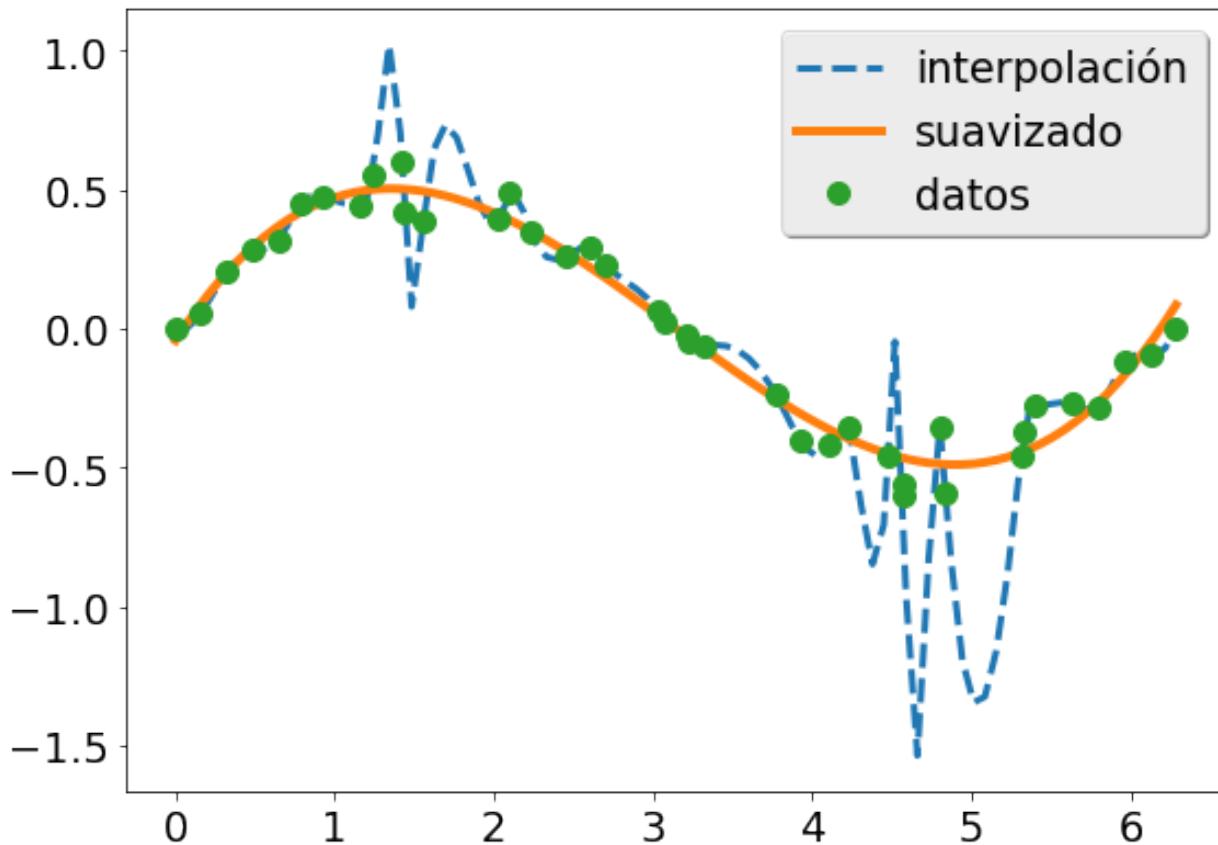
```
# Los puntos a interpolar tienen "ruido" en las dos direcciones x-y
plt.plot(x3,y3,'o', x4,y4, '.', alpha=0.7 );
```



```
# Grilla donde evaluar la función interpolada
x1 = np.linspace(0, 2*np.pi, 90)
```

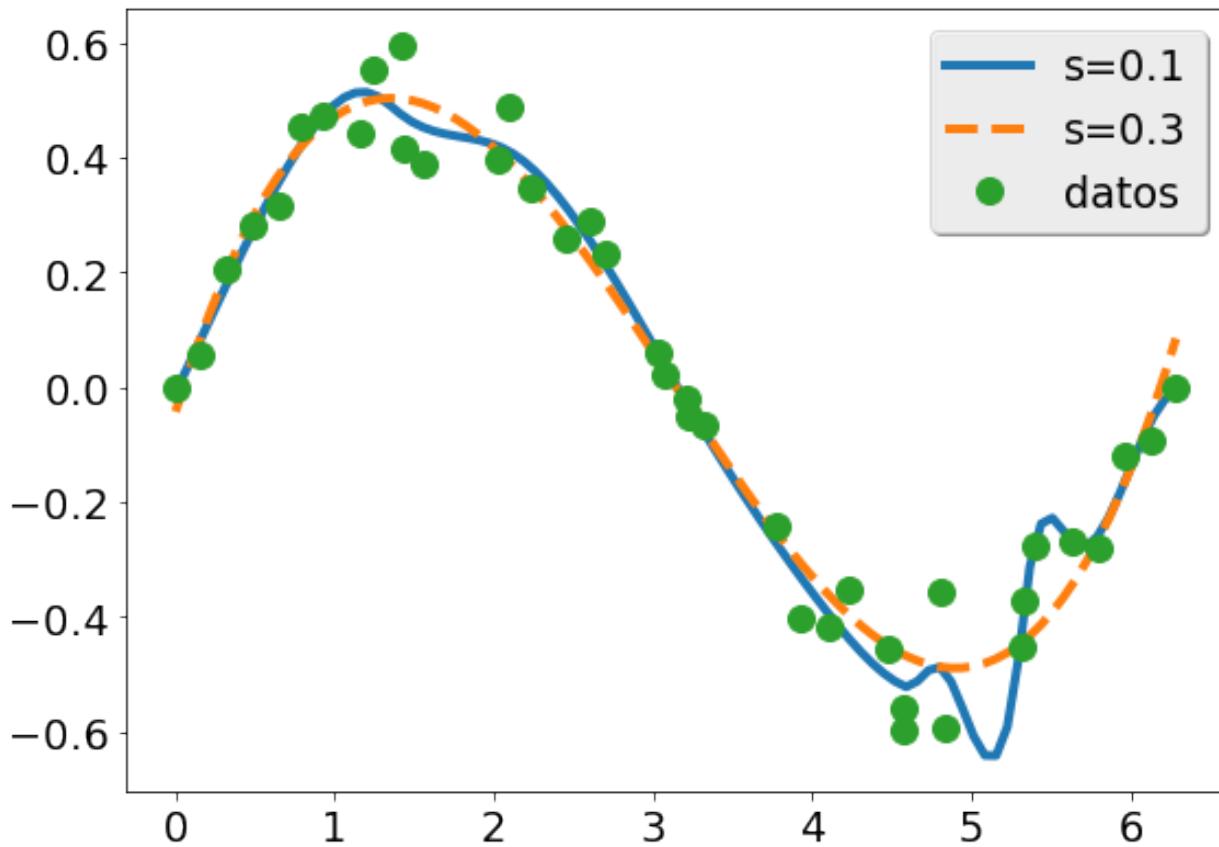
```
tck0 = interpolate.splrep(x3,y3, s=0) # Interpolación con B-Splines
y_s0 = interpolate.splev(x1,tck0)      # Evaluamos en la nueva grilla
tck3 = interpolate.splrep(x3,y3,s=0.3) # Aproximación suavizada
y_s3 = interpolate.splev(x1,tck3)      # Evaluamos en la nueva grilla
```

```
plt.figure(figsize=fsize)
plt.plot(x1,y_s0,'--', lw=3, label=u'interpolación' )
plt.plot(x1,y_s3, "-", label=u'suavizado');
plt.plot(x3,y3,'o', label='datos' )
plt.legend(loc='best');
```



El valor del parámetro s determina cuánto se suaviza la curva. El valor por default $s=0$ obliga al algoritmo a obtener una solución que no difiere en los valores de la tabla, un valor de s mayor que cero da cierta libertad para obtener la aproximación que se acerque a todos los valores manteniendo una curva relativamente suave. El suavizado máximo corresponde a $s=1$. Veamos cómo cambia la aproximación con el factor s :

```
tck1 = interpolate.splrep(x3,y3, s=0.1) # Interpolación con suavizado
y_s1 = interpolate.splev(x1,tck1)
plt.figure(figsize=fsize)
plt.plot(x1,y_s1, "-", label=u's=0.1');
plt.plot(x1,y_s3, "--", label=u's=0.3');
plt.plot(x3,y3,'o', markersize=12, label='datos' )
plt.legend(loc='best');
```



14.1.5 Cantidades derivadas de *splines*

De la interpolación (suavizada) podemos calcular, por ejemplo, la derivada.

```
yder= interpolate.splev(x1,tck3,der=1) # Derivada
```

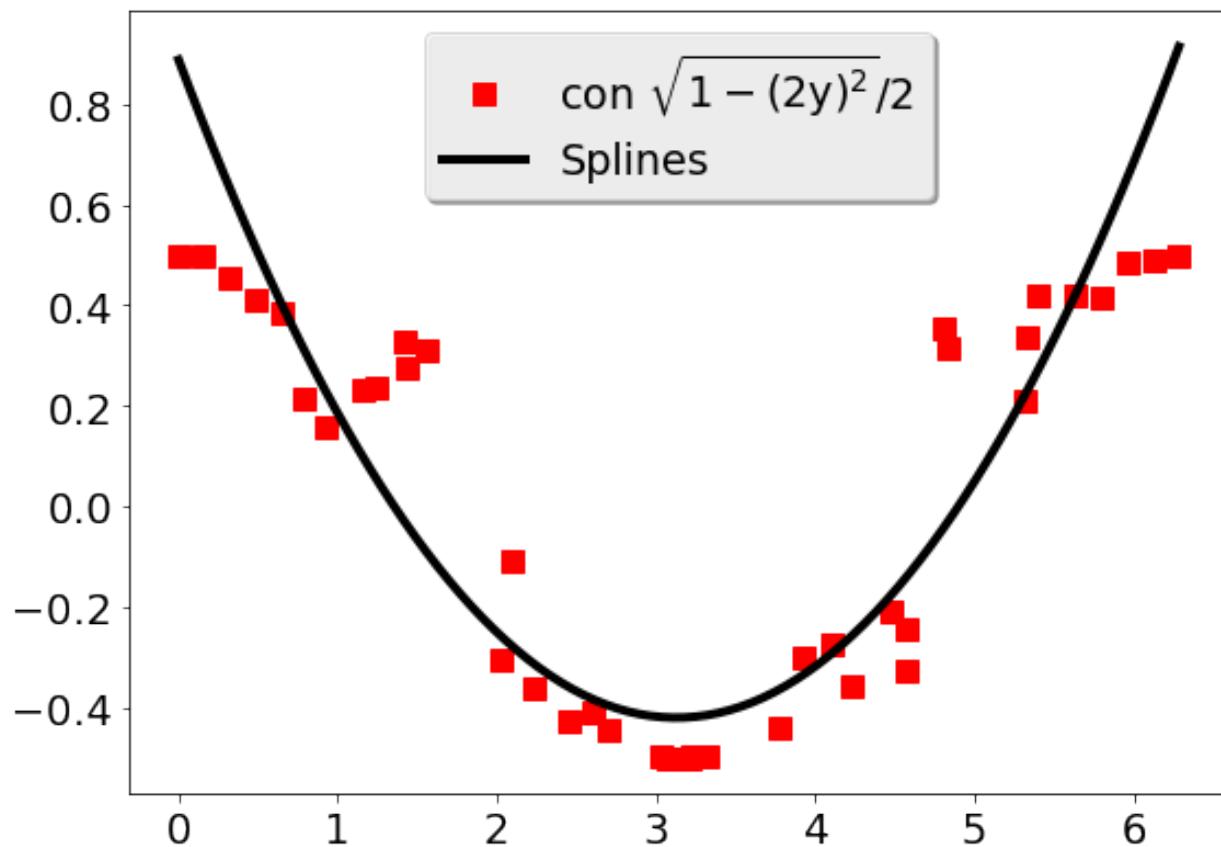
Si tenemos sólo los datos podríamos tratar de calcular la derivada como el coseno

$$y' = 0,5\sqrt{1 - (2y)^2}$$

Comparemos los dos resultados:

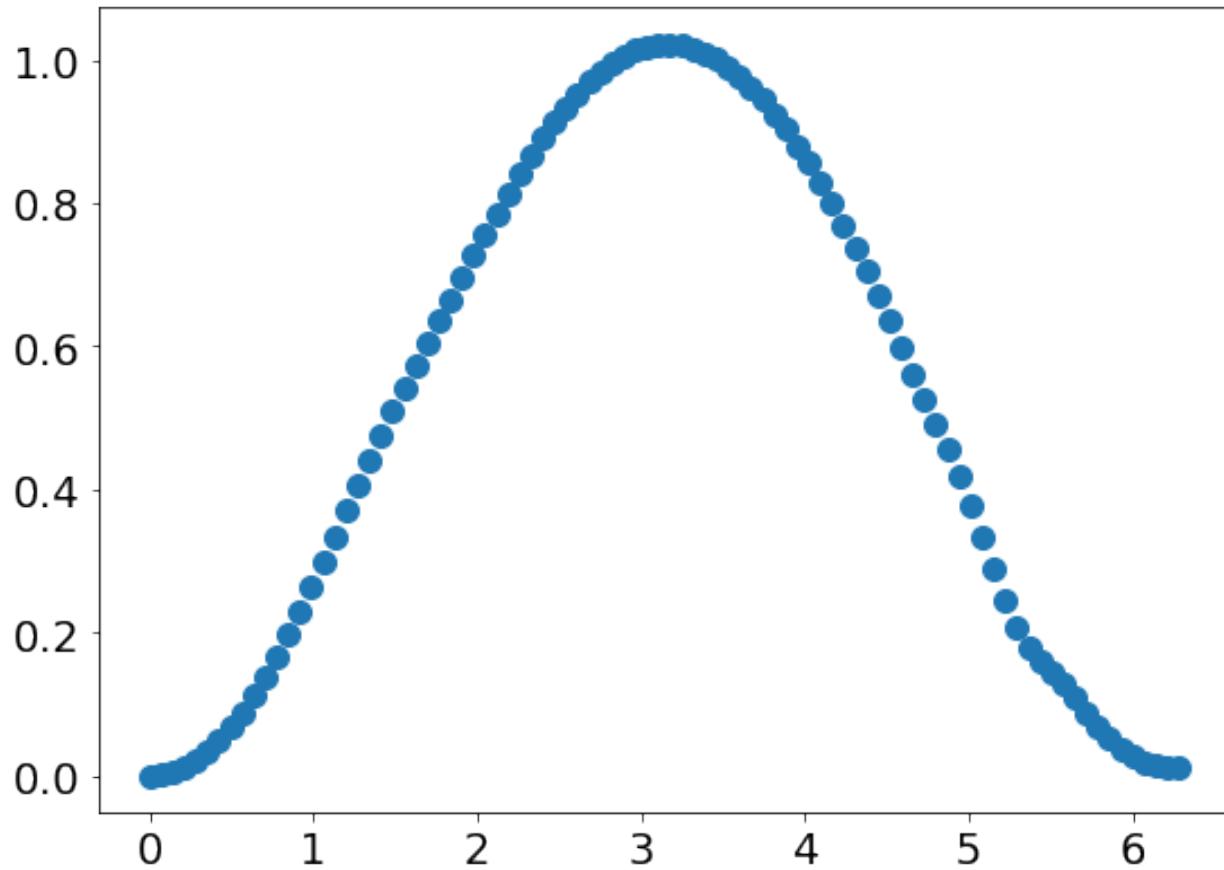
```
cond = (x3 > np.pi/2) & (x3 < 3*np.pi/2)
yprimal = np.where(cond, -1, 1) * 0.5*np.sqrt(np.abs(1 - (2*y3)**2))
```

```
plt.figure(figsize=fsize)
plt.plot(x3, yprimal,"sr", label=r"con $\sqrt{1-(2y)^2}/2$")
plt.plot(x1,yder,'-k', label=u'Splines')
plt.legend(loc='best');
```

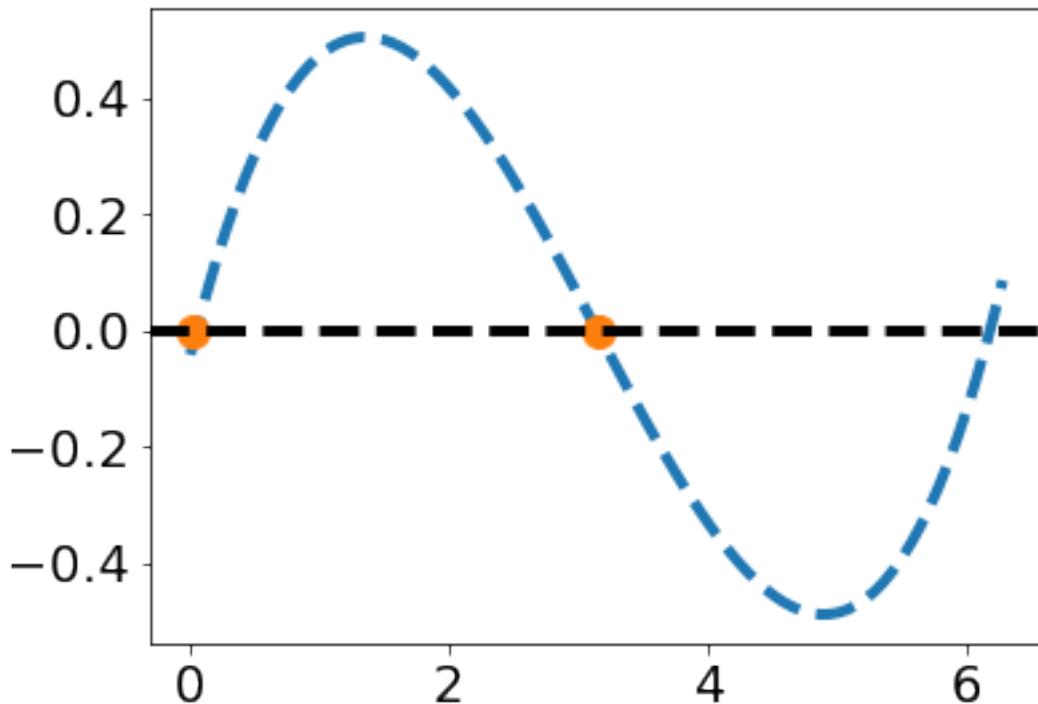


o podemos calcular la integral, o las raíces de la función

```
plt.figure(figsize=fsize)
yt= np.array([interpolate.splint(0,t,tck1) for t in x1])
plt.plot(x1,yt,'o');
```



```
plt.plot(x1,y_s3, "--", label=u's=0.3');
plt.plot(interpolate.sproot(tck1), [0,0], 'o', markersize=12)
plt.axhline(0, color='k',ls='--');
```



14.2 Interpolación en dos dimensiones

Un ejemplo del caso más simple de interpolación en dos dimensiones podría ser cuando tenemos los datos sobre una grilla, que puede no estar equiespaciada y necesitamos los valores sobre otra grilla (quizás para graficarlos). En ese caso podemos usar `scipy.interpolate.interp2d()` para interpolar los datos a una grilla equiespaciada. El método necesita conocer los datos sobre la grilla, y los valores de x e y a los que corresponden.

Definamos una tabla de valores con nuestros datos sobre una grilla $x - y$, no-equiespaciada en la dirección y :

```
import numpy as np
import matplotlib.pyplot as plt
def f(x,y):
    return 5*y*(1-x)*np.cos(4*np.pi*x) * np.exp(-y/2)

# Nuestra tabla de valores
x = np.linspace(0, 4, 13)
y = np.array([0, 1, 2, 3.75, 3.875, 3.9375, 4])
```

```
#  
X, Y = np.meshgrid(x, y)  
Z = f(X, Y)
```

```
# Grilla en la cual interpolar
x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
# Notar que le tenemos que pasar los arrays unidimensionales x e y
f1 = interpolate.interp2d(x, y, Z, kind='linear')
z1 = f1(x2, y2)
f3 = interpolate.interp2d(x, y, Z, kind='cubic')
```

(continué en la próxima página)

(proviene de la página anterior)

```
z3 = f3(x2, y2)
f5 = interpolate.interp2d(x, y, z, kind='quintic')
z5 = f5(x2, y2)
```

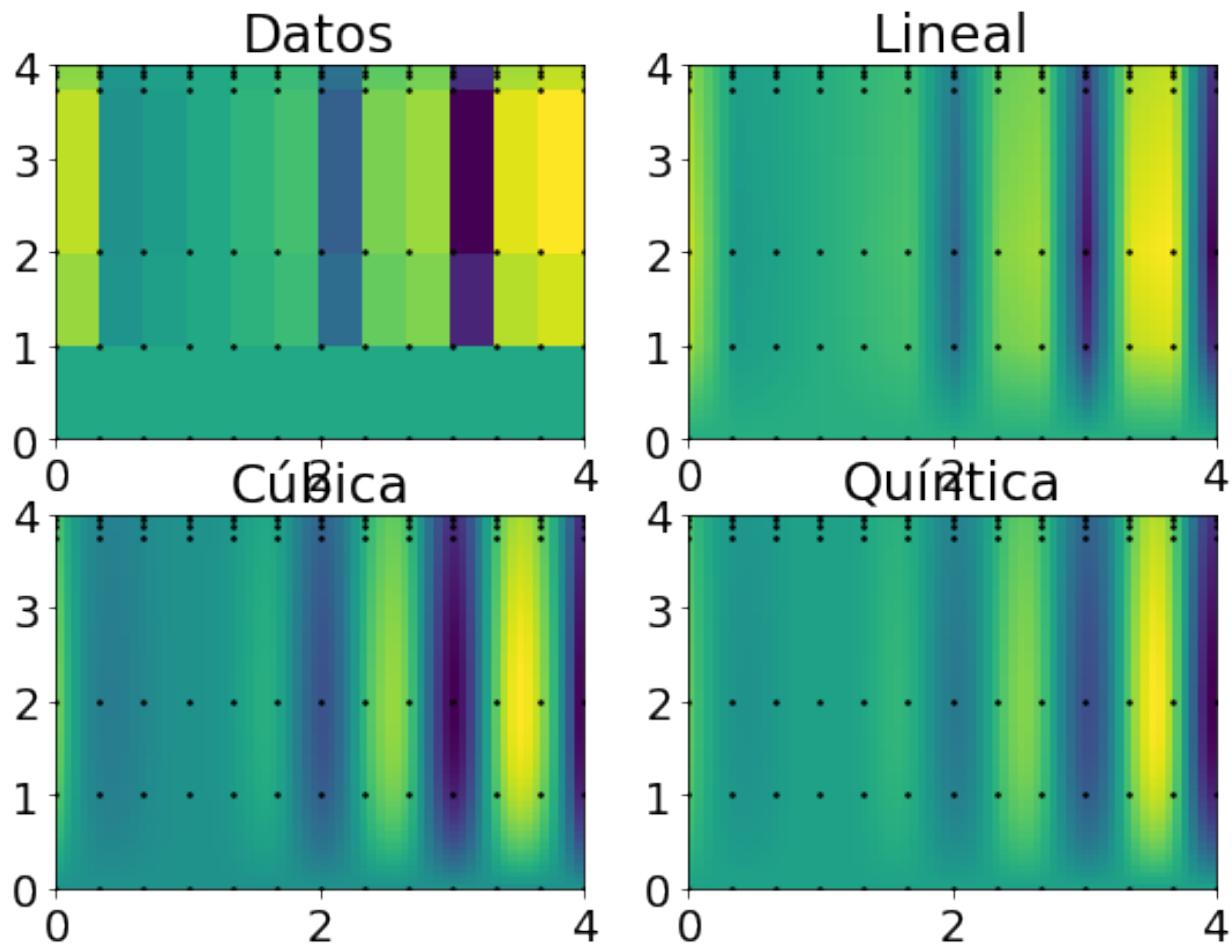
Ahora tenemos los valores de la función sobre la grilla determinada por todos los posibles pares (x, y) de la tabla original. En el caso de $f1$ utilizamos una interpolación lineal para cada punto, en $f3$ una interpolación cúbica, y en $f5$ una interpolación de orden 5. Grafiquemos los resultados

```
fig, ax = plt.subplots(figsize=fsize, nrows=2, ncols=2)

# Solo para graficar
X2, Y2 = np.meshgrid(x2, y2)

# Agregamos los puntos de la grilla original
for i,j,a in np.ndenumerate(ax):
    ax[i,j].plot(X,Y,'.k', markersize=3)

ax[0,0].pcolormesh(X, Y, Z)
ax[0,0].set_title('Datos')
ax[0,1].pcolormesh(X2, Y2, Z1)
ax[0,1].set_title('Lineal')
ax[1,0].pcolormesh(X2, Y2, Z3)
ax[1,0].set_title('Cúbica')
ax[1,1].pcolormesh(X2, Y2, Z5)
ax[1,1].set_title('Quíntica')
plt.show()
```



Acá usamos `numpy.meshgrid()` que permite generar grillas bidimensionales a partir de dos vectores unidimensionales. Por ejemplo de

```
x, Y = np.meshgrid(x,y)
```

Repitamos un ejemplo simple de uso de `meshgrid()`

```
a = np.arange(3)
b = np.arange(3,7)
A,B = np.meshgrid(a,b)
print(' a=', a, '\n', 'b=', b)
```

```
a= [0 1 2]
b= [3 4 5 6]
```

```
print('A=\n', A, '\n')
print('B=\n', B)
```

```
A=
[[0 1 2]
[0 1 2]
[0 1 2]
[0 1 2]]
```

(continué en la próxima página)

(provine de la página anterior)

```
B=
[[3 3 3]
 [4 4 4]
 [5 5 5]
 [6 6 6]]
```

14.3 Interpolación sobre datos no estructurados

Si tenemos datos, correspondientes a una función o una medición sólo sobre algunos valores (x, y) que no se encuentran sobre una grilla, podemos interpolarlos a una grilla regular usando `griddat()`. Veamos un ejemplo de uso

```
# Generamos los datos
def f(x, y):
    s = np.hypot(x, y)          # Calcula la hipotenusa
    phi = np.arctan2(y, x)      # Calcula el ángulo
    tau = s + s*(1-s)/5 * np.sin(6*phi)
    return 5*(1-tau) + tau
# Generamos los puntos x,y,z en una grilla para comparar con la interpolación
# Notar que es una grilla de 100x100 = 10000 puntos
x = np.linspace(-1,1,100)
y = np.linspace(-1,1,100)
X, Y = np.meshgrid(x,y)
T = f(X, Y)
```

Aquí `T` contiene la función sobre todos los puntos (x, y) obtenidos de combinar cada punto en el vector `x` con cada punto en el vector `y`. Notar que la cantidad total de puntos de `T` es `T.size = 10000`

Elegimos `npts=400` puntos al azar de la función que vamos a usar como tabla de datos a interpolar usando distintos métodos de interpolación

```
npts = 400
px, py = np.random.choice(x, npts), np.random.choice(y, npts)
Z = f(px, py)
```

Para poder mostrar todos los métodos juntos, vamos a interpolar dentro del loop `for`

```
# Graficación:
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=fsize, )

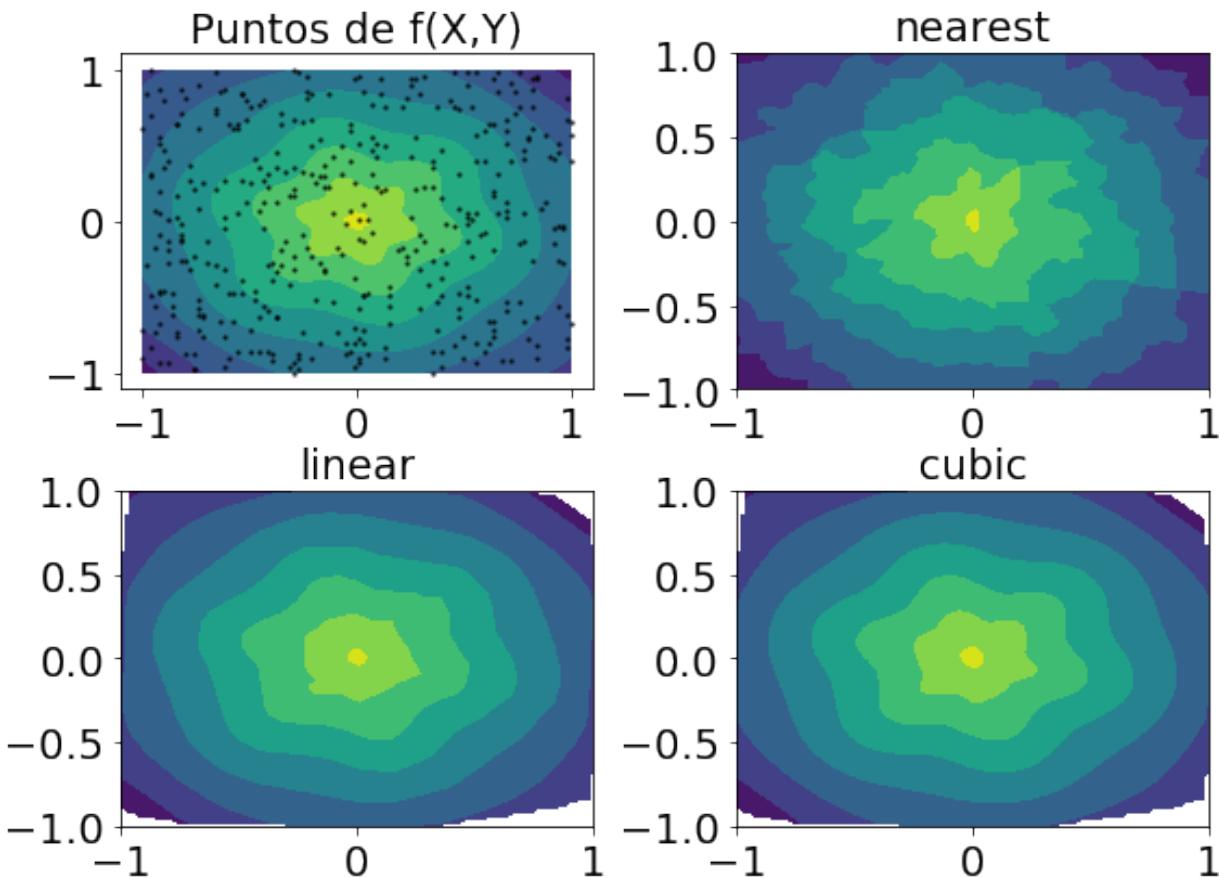
# Graficamos la función sobre la grilla estructurada a modo de ilustración
# Graficamos los puntos seleccionados
ax[0,0].contourf(X, Y, T)
ax[0,0].scatter(px, py, c='k', alpha=0.6, marker='.', s=8)
ax[0,0].set_title('Puntos de f(X,Y)', fontsize='large')

# Interpolamos usando los distintos métodos y graficamos
for i, method in enumerate(['nearest', 'linear', 'cubic']):
    Ti = interpolate.griddata((px, py), Z, (X, Y), method=method)
    r, c = (i+1) // 2, (i+1) % 2
    ax[r,c].contourf(X, Y, Ti)
```

(continué en la próxima página)

(provine de la página anterior)

```
ax[r,c].set_title('{}'.format(method), fontsize='large')
plt.subplots_adjust(hspace=0.3, wspace=0.3)
```



En la primera figura (arriba a la izquierda) graficamos la función evaluada en el total de puntos (10000 puntos) junto con los puntos utilizados para el ajuste. Los otros tres gráficos corresponden a la función evaluada siguiendo el ajuste correspondiente en cada caso.

14.4 Fiteos de datos

14.4.1 Ajuste con polinomios

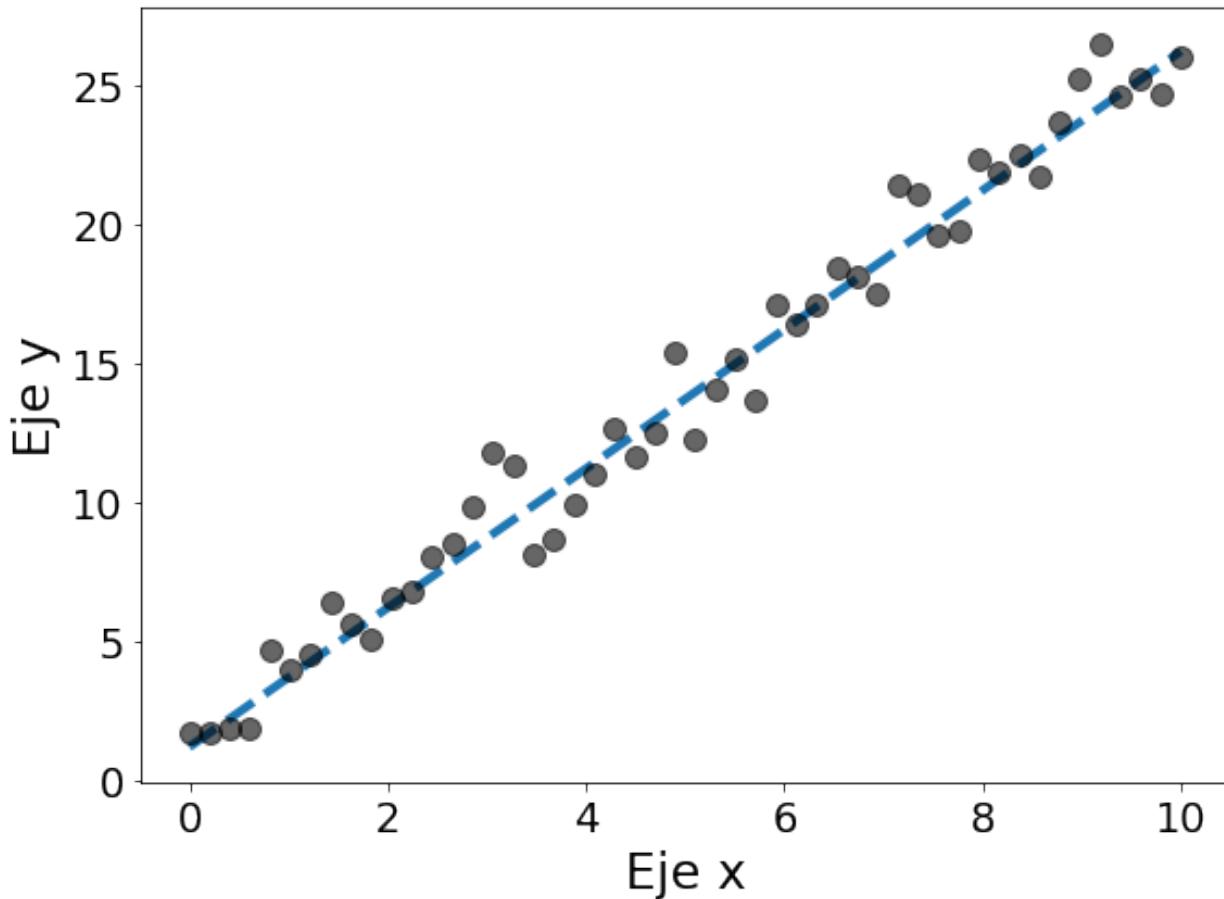
Habitualmente realizamos ajustes sobre datos que tienen incertezas o ruido. Generemos estos datos (con ruido normalmente distribuido)

```
plt.figure(figsize=fsize)
x = np.linspace(0, 10, 50)
y0 = 2.5*x + 1.2
ruido = np.random.normal(loc= 0., scale= 1, size= y0.size)
y = y0 + ruido
plt.plot(x,y0, '--')
plt.plot(x,y, 'ok', alpha=0.6)
```

(continué en la próxima página)

(proviene de la página anterior)

```
plt.xlabel("Eje x")
plt.ylabel("Eje y");
```



Ahora vamos a ajustar con una recta

$$y = mx + b \quad \equiv \quad f(x) = p[0]x + p[1]$$

Es una regresión lineal (o una aproximación con polinomios de primer orden)

```
p = np.polyfit(x,y,1)
# np.info(np.polyfit) # para obtener más información
```

```
print(p)
print(type(p)) # Qué tipo es?
```

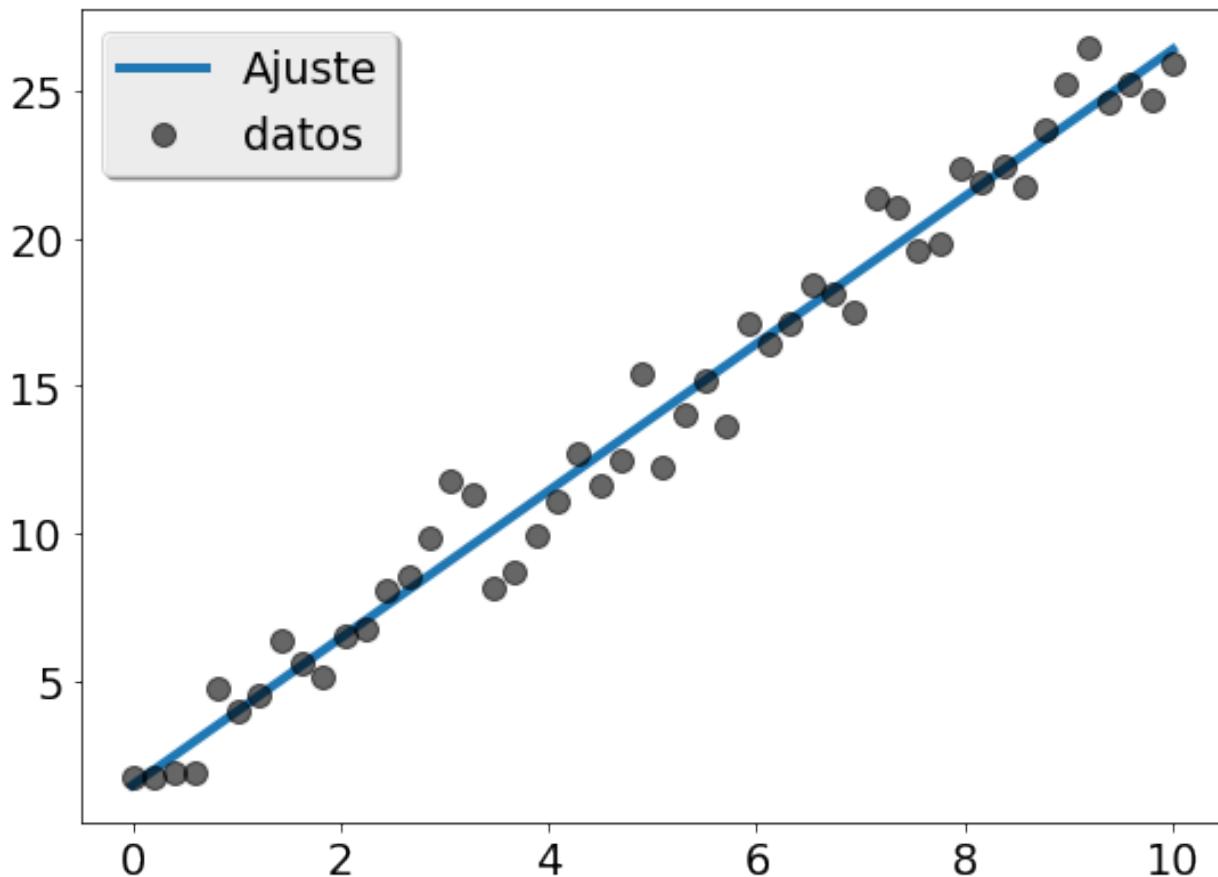
```
[2.49458444 1.44519004]
<class 'numpy.ndarray'>
```

Nota: ¿Qué devuelve polyfit () ?

Un array correspondiente a los coeficientes del polinomio de fiteo. En este caso, como estamos haciendo un ajuste lineal, nos devuelve los coeficientes de un polinomio de primer orden (una recta)

```
y = p[0]*x + p[1]
```

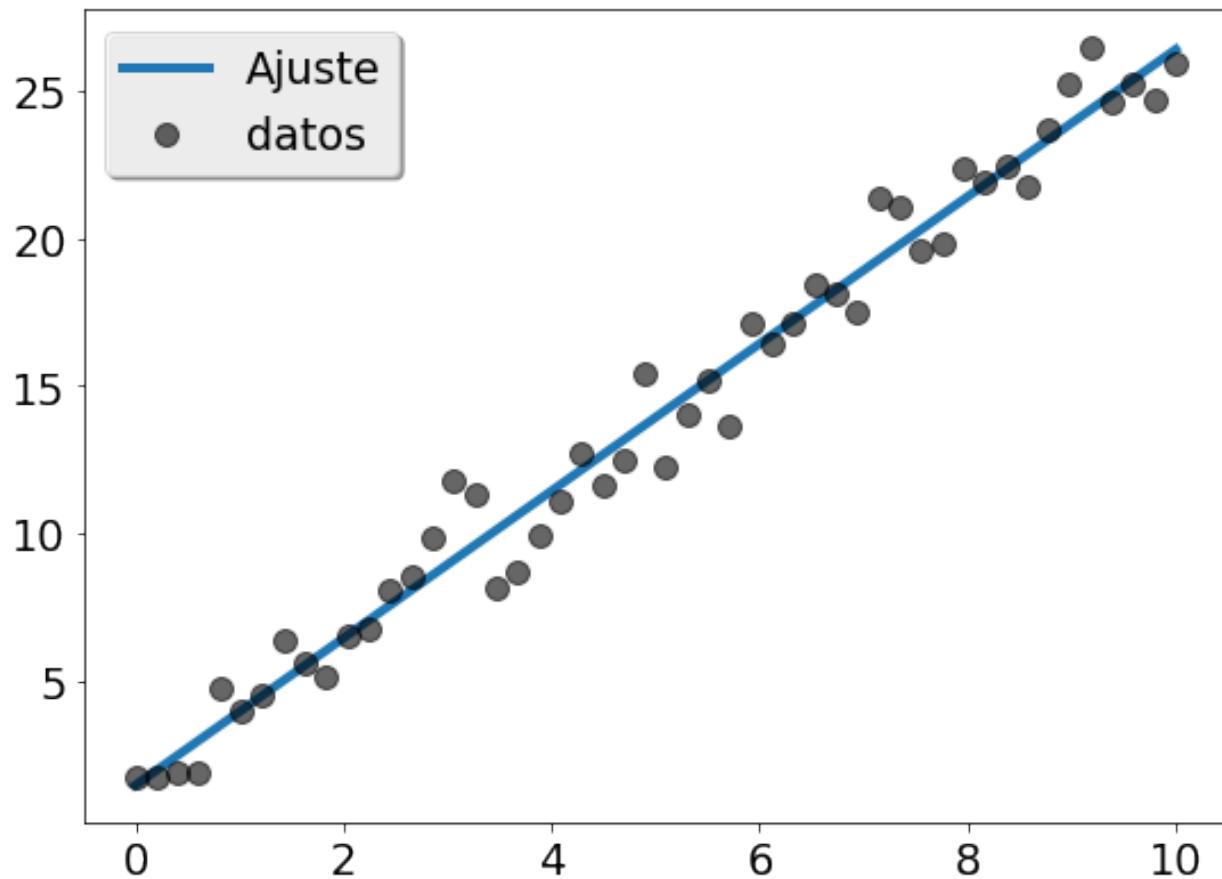
```
plt.figure(figsize=fsize)
plt.plot(x, p[0]*x + p[1], '-', label='Ajuste')
plt.plot(x,y,'ok', label='datos', alpha=0.6)
plt.legend(loc='best');
```



Ahora en vez de escribir la recta explícitamente le pedimos a **numpy** que lo haga usando los coeficientes que encontramos mediante el fiteo (utilizando la función *polyval*)

```
y = np.polyval(p,x)
```

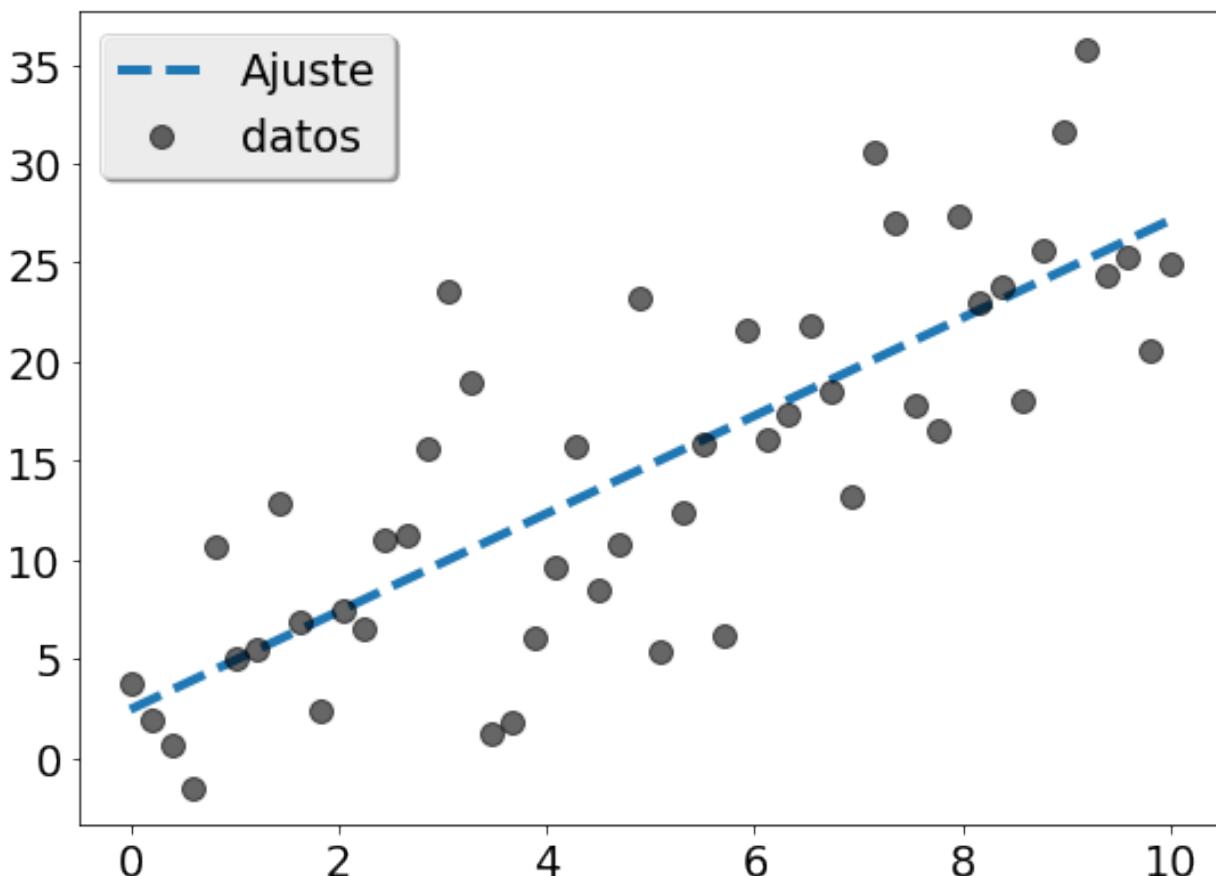
```
plt.figure(figsize=fsize)
plt.plot(x, np.polyval(p,x), '-', label='Ajuste')
plt.plot(x,y,'ok', label='datos', alpha=0.6)
plt.legend(loc='best');
```



Como vemos, arroja exactamente el mismo resultado.

Si los datos tienen mucho ruido lo que obtenemos es, por supuesto, una recta que pasa por la nube de puntos:

```
y= y0 + 5*ruido
p = np.polyfit(x, y , 1)
plt.figure(figsize=fsiz)
plt.plot(x,np.polyval(p,x), '--', label='Ajuste')
plt.plot(x,y, 'ok', alpha=0.6, label='datos')
plt.legend(loc='best');
```

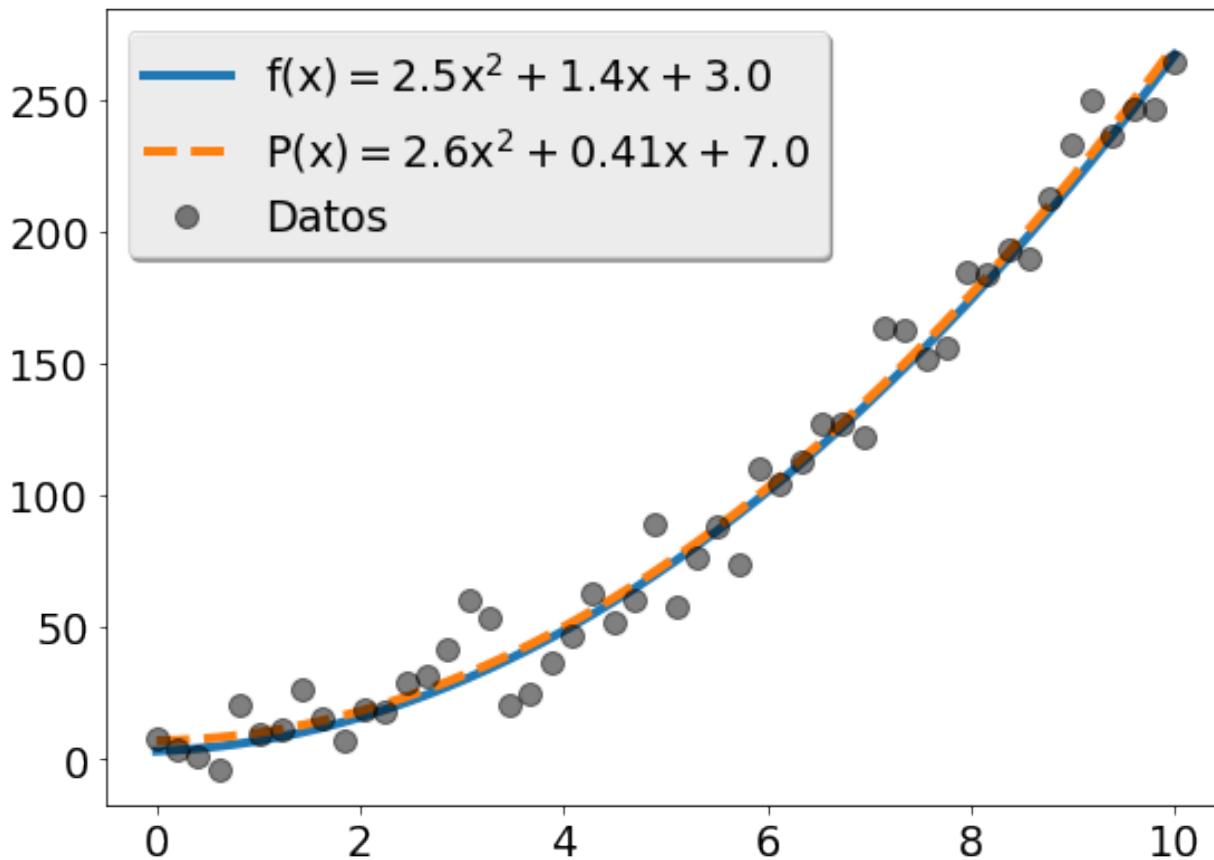


Similarmente podemos usar polinomios de orden superior. Por ejemplo, para utilizar paráolas sólo tenemos que cambiar el orden n del polinomio en el argumento de `polyfit(x, y, n)`:

```
# Generamos los datos
a = [2.5, 1.4, 3.]
y0 = np.polyval(a,x)
y= y0 + 10*ruido
```

```
# Ajustamos con un polinomio de segundo grado
p = np.polyfit(x, y, 2)
```

```
plt.figure(figsize=fsize)
plt.plot(x,y0,'-', label="$f(x)={0:.2}x^2 + {1:.2} x + {2:.2}$".format(*a))
plt.plot(x,np.polyval(p,x), '--', label="$P(x)={0:.2}x^2 + {1:.2} x + {2:.2}$".
         format(*p))
plt.plot(x,y,'ok', alpha=0.7,label='Datos')
plt.legend(loc='best');
```



14.5 Fiteos con funciones arbitrarias

Vamos ahora a fitear una función que no responde a la forma polinomial.

El submódulo `optimize` del paquete `scipy` tiene rutinas para realizar ajustes de datos utilizando funciones arbitrarias

Utilicemos una función complicada:

```
# string definido para la leyenda
sfuncion= r'${0:.3}\$, \sin ({1:.2}\$, x {3:+.2f}) \$, \exp (-{2:.2f} \$ x)$'

def fit_func(x, a, b, c, d):
    y= a*np.sin(b*x-d)*np.exp(-c*x)
    return y
```

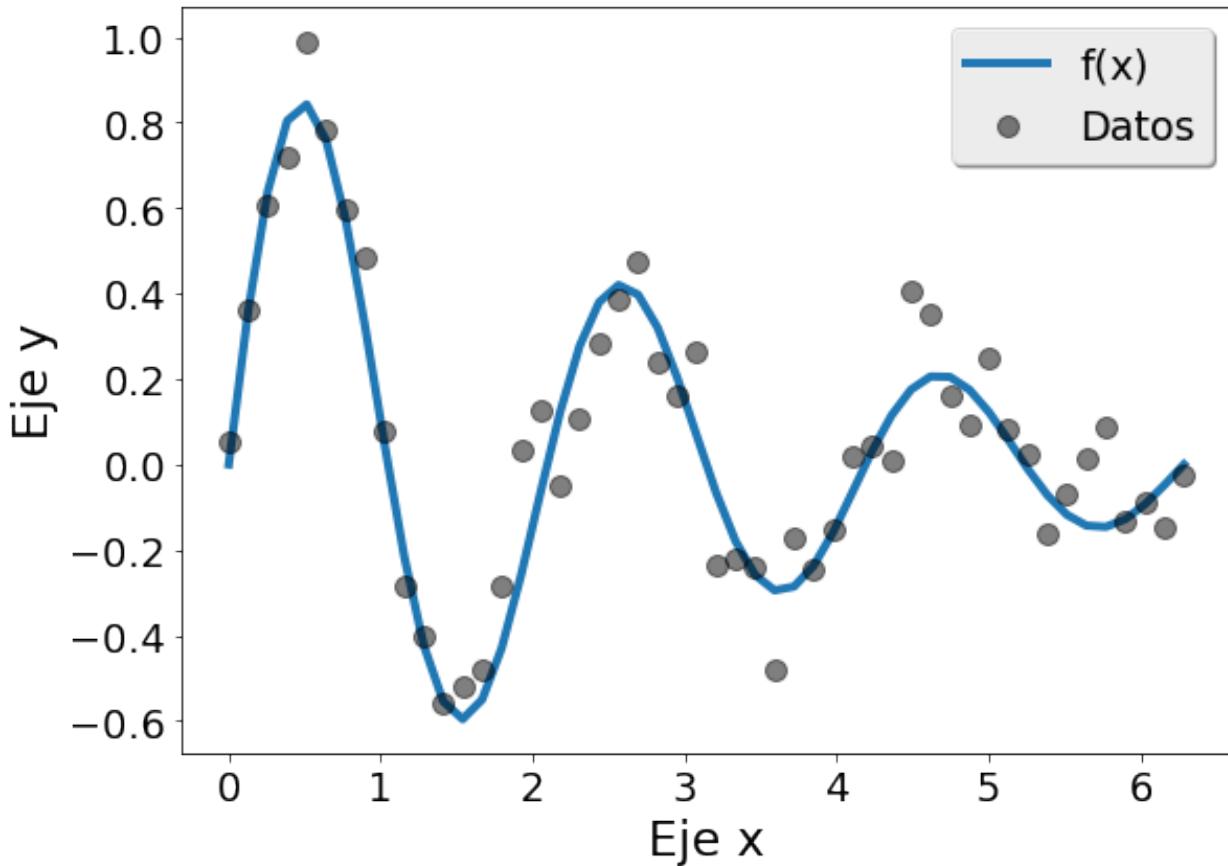
Generamos ahora datos con dispersión basados en la función

$$f(x) = a \sin(bx - d)e^{-cx}$$

```
x = np.linspace(0., 2*np.pi, 50)
y0= fit_func(x, 1., 3., 1/3, 0 )
y = y0 + 0.1*ruido
```

y los graficamos

```
plt.figure(figsize=fsize)
plt.plot(x,y0,'-',label="$f(x)$")
plt.plot(x,y,'ok',alpha=0.5,label='Datos') # repeated from above
plt.xlabel("Eje x") # labels again
plt.ylabel("Eje y")
plt.legend(loc='best');
```



Ahora vamos a interpolar los datos utilizando funciones del paquete **Scipy**.

En primer lugar vamos a utilizar la función `curve_fit`:

```
from scipy.optimize import curve_fit
```

```
# ¿Qué hace esta función?
help(curve_fit)
```

Help on function `curve_fit` in module `scipy.optimize.minpack`:

```
curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False, check_
finite=True, bounds=(-inf, inf), method=None, jac=None, **kwargs)
    Use non-linear least squares to fit a function, f, to data.
```

```
Assumes ydata = f(xdata, *params) + eps

Parameters
-----
f : callable
    The model function, f(x, ...). It must take the independent
    variable as the first argument and the parameters to fit as
    separate remaining arguments.
xdata : An M-length sequence or an (k,M)-shaped array for functions with  $\hookrightarrow k$ 
    predictors
        The independent variable where the data is measured.
ydata : M-length sequence
        The dependent data --- nominally f(xdata, ...)
p0 : None, scalar, or N-length sequence, optional
    Initial guess for the parameters. If None, then the initial
    values will all be 1 (if the number of parameters for the function
    can be determined using introspection, otherwise a ValueError
    is raised).
sigma : None or M-length sequence or MxM array, optional
    Determines the uncertainty in ydata. If we define residuals as
    r = ydata - f(xdata, *popt), then the interpretation of sigma
    depends on its number of dimensions:
        - A 1-d sigma should contain values of standard deviations of
            errors in ydata. In this case, the optimized function is
            chisq = sum((r / sigma) ** 2).
        - A 2-d sigma should contain the covariance matrix of
            errors in ydata. In this case, the optimized function is
            chisq = r.T @ inv(sigma) @ r.
.. versionadded:: 0.19

None (default) is equivalent of 1-d sigma filled with ones.
absolute_sigma : bool, optional
    If True, sigma is used in an absolute sense and the estimated  $\hookrightarrow$ 
    parameter
        covariance pcov reflects these absolute values.

    If False, only the relative magnitudes of the sigma values matter.
    The returned parameter covariance matrix pcov is based on scaling
    sigma by a constant factor. This constant is set by demanding that the
    reduced chisq for the optimal parameters popt when using the
    scaled sigma equals unity. In other words, sigma is scaled to
    match the sample variance of the residuals after the fit.
    Mathematically,
        pcov(absolute_sigma=False) = pcov(absolute_sigma=True) * chisq(popt) /
 $\hookrightarrow (M-N)$ 
    check_finite : bool, optional
        If True, check that the input arrays do not contain nans or infs,
        and raise a ValueError if they do. Setting this parameter to
        False may silently produce nonsensical results if the input arrays
        do contain nans. Default is True.
bounds : 2-tuple of array_like, optional
```

```

Lower and upper bounds on parameters. Defaults to no bounds.
Each element of the tuple must be either an array with the length_
↳equal
to the number of parameters, or a scalar (in which case the bound is
taken to be the same for all parameters.) Use np.inf with an
appropriate sign to disable bounds on all or some parameters.

.. versionadded:: 0.17
method : {'lm', 'trf', 'dogbox'}, optional
Method to use for optimization. See least_squares for more details.
Default is 'lm' for unconstrained problems and 'trf' if bounds are
provided. The method 'lm' won't work when the number of observations
is less than the number of variables, use 'trf' or 'dogbox' in this
case.

.. versionadded:: 0.17
jac : callable, string or None, optional
Function with signature jac(x, ...) which computes the Jacobian
matrix of the model function with respect to parameters as a dense
array_like structure. It will be scaled according to provided sigma.
If None (default), the Jacobian will be estimated numerically.
String keywords for 'trf' and 'dogbox' methods can be used to select
a finite difference scheme, see least_squares.

.. versionadded:: 0.18
kwargs
Keyword arguments passed to leastsq for method='lm' or
least_squares otherwise.

>Returns
-----
popt : array
    Optimal values for the parameters so that the sum of the squared
    residuals of f(xdata, *popt) - ydata is minimized
pcov : 2d array
    The estimated covariance of popt. The diagonals provide the variance
    of the parameter estimate. To compute one standard deviation errors
    on the parameters use perr = np.sqrt(np.diag(pcov)).
    How the sigma parameter affects the estimated covariance
    depends on absolute_sigma argument, as described above.
    If the Jacobian matrix at the solution doesn't have a full rank, then
    'lm' method returns a matrix filled with np.inf, on the other hand
    'trf' and 'dogbox' methods use Moore-Penrose pseudoinverse to compute
    the covariance matrix.

>Raises
-----
ValueError
    if either ydata or xdata contain NaNs, or if incompatible options
    are used.

>RuntimeError

```

if the least-squares minimization fails.

OptimizeWarning
if covariance of the parameters can not be estimated.

See Also

`least_squares` : Minimize the sum of squares of nonlinear functions.
`scipy.stats.linregress` : Calculate a linear least squares regression for two sets of measurements.

Notes

With `method='lm'`, the algorithm uses the Levenberg-Marquardt algorithm through `leastsq`. Note that this algorithm can only deal with unconstrained problems.

Box constraints can be handled by methods '`trf`' and '`dogbox`'. Refer to the docstring of `least_squares` for more information.

Examples

>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.optimize import curve_fit

>>> def func(x, a, b, c):
... return a * np.exp(-b * x) + c

Define the data to be fit with some noise:

```
>>> xdata = np.linspace(0, 4, 50)  
>>> y = func(xdata, 2.5, 1.3, 0.5)  
>>> np.random.seed(1729)  
>>> y_noise = 0.2 * np.random.normal(size=xdata.size)  
>>> ydata = y + y_noise  
>>> plt.plot(xdata, ydata, 'b-', label='data')
```

Fit for the parameters `a`, `b`, `c` of the function `func`:

```
>>> popt, pcov = curve_fit(func, xdata, ydata)  
>>> popt  
array([ 2.55423706,  1.35190947,  0.47450618])  
>>> plt.plot(xdata, func(xdata, *popt), 'r-'  
...           label='fit: a=%5.3f, b=%5.3f, c=%5.3f' % tuple(popt))
```

Constrain the optimization to the region of $0 \leq a \leq 3$, $0 \leq b \leq 1$ and $0 \leq c \leq 0.5$:

```
>>> popt, pcov = curve_fit(func, xdata, ydata, bounds=(0, [3., 1., 0.5]))  
>>> popt  
array([ 2.43708906,  1.          ,  0.35015434])  
>>> plt.plot(xdata, func(xdata, *popt), 'g--'  
...           label='fit: a=%5.3f, b=%5.3f, c=%5.3f' % tuple(popt))
```

```
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend()
>>> plt.show()
```

En su forma más simple toma los siguientes argumentos:

```
Parameters
-----
f : callable
    The model function, f(x, ...). It must take the independent
    variable as the first argument and the parameters to fit as
    separate remaining arguments.
xdata : An M-length sequence or an (k,M)-shaped array for functions with k predictors
    The independent variable where the data is measured.
ydata : M-length sequence
    The dependent data --- nominally f(xdata, ...)
p0 : None, scalar, or N-length sequence, optional
    Initial guess for the parameters. If None, then the initial
    values will all be 1 if the number of parameters for the function
    can be determined using introspection, otherwise a ValueError
    is raised).
```

El primero: *f* es la función que utilizamos para modelar los datos, y que dependerá de la variable independiente *x* y de los parámetros a ajustar.

También debemos darle los valores tabulados en las direcciones *x* (la variable independiente) e *y* (la variable dependiente).

Además, debido a las características del cálculo numérico que realiza suele ser muy importante darle valores iniciales a los parámetros que queremos ajustar.

Veamos lo que devuelve:

```
Returns
-----
popt : array
    Optimal values for the parameters so that the sum of the squared error
    of ``f(xdata, *popt) - ydata`` is minimized
pcov : 2d array
    The estimated covariance of popt. The diagonals provide the variance
    of the parameter estimate.
```

El primer *array* tiene los parámetros para best-fit, y el segundo da la estimación del error: la matriz de covarianza

Ya tenemos los valores a ajustar guardados en arrays *x* e *y*

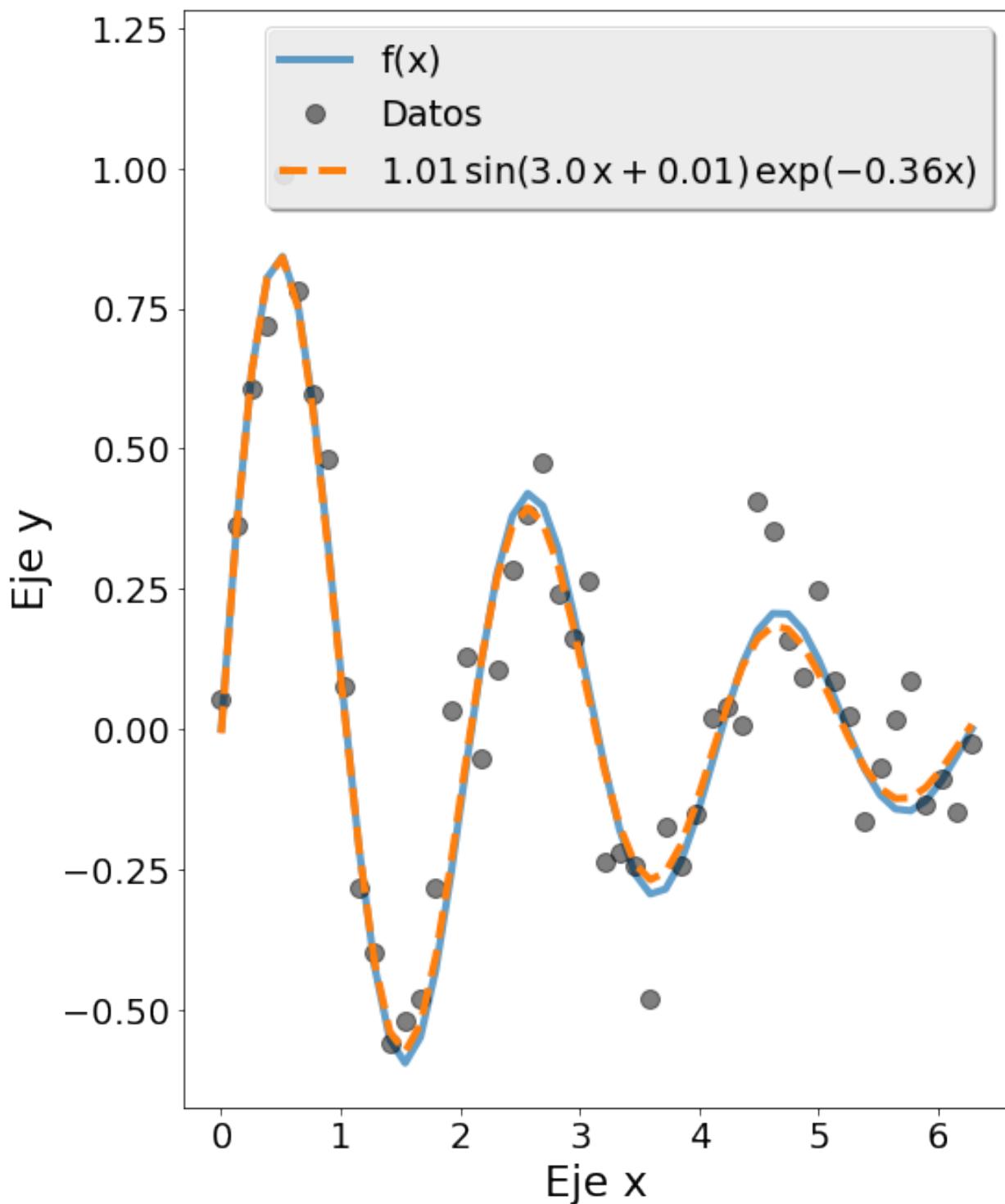
```
#initial_guess= None
initial_guess= [1., -1., 1., 0.2]
params, p_covarianza = curve_fit(fit_func, x, y, initial_guess)
```

```
plt.figure(figsize=(8,10))
plt.plot(x,y0,'-', alpha=0.7,label="$f(x)$")
plt.plot(x,y,'ok', alpha=0.5, label='Datos') # repeated from above
label=sfuncion.format(*params)
```

(continué en la próxima página)

(proviene de la página anterior)

```
plt.plot(x, fit_func(x, *params), '--', label=label)
plt.xlabel("Eje x") # labels again
plt.ylabel("Eje y")
plt.legend(loc='best');
ylim = plt.ylim()
plt.ylim((ylim[0], 1.2*ylim[1]));
```

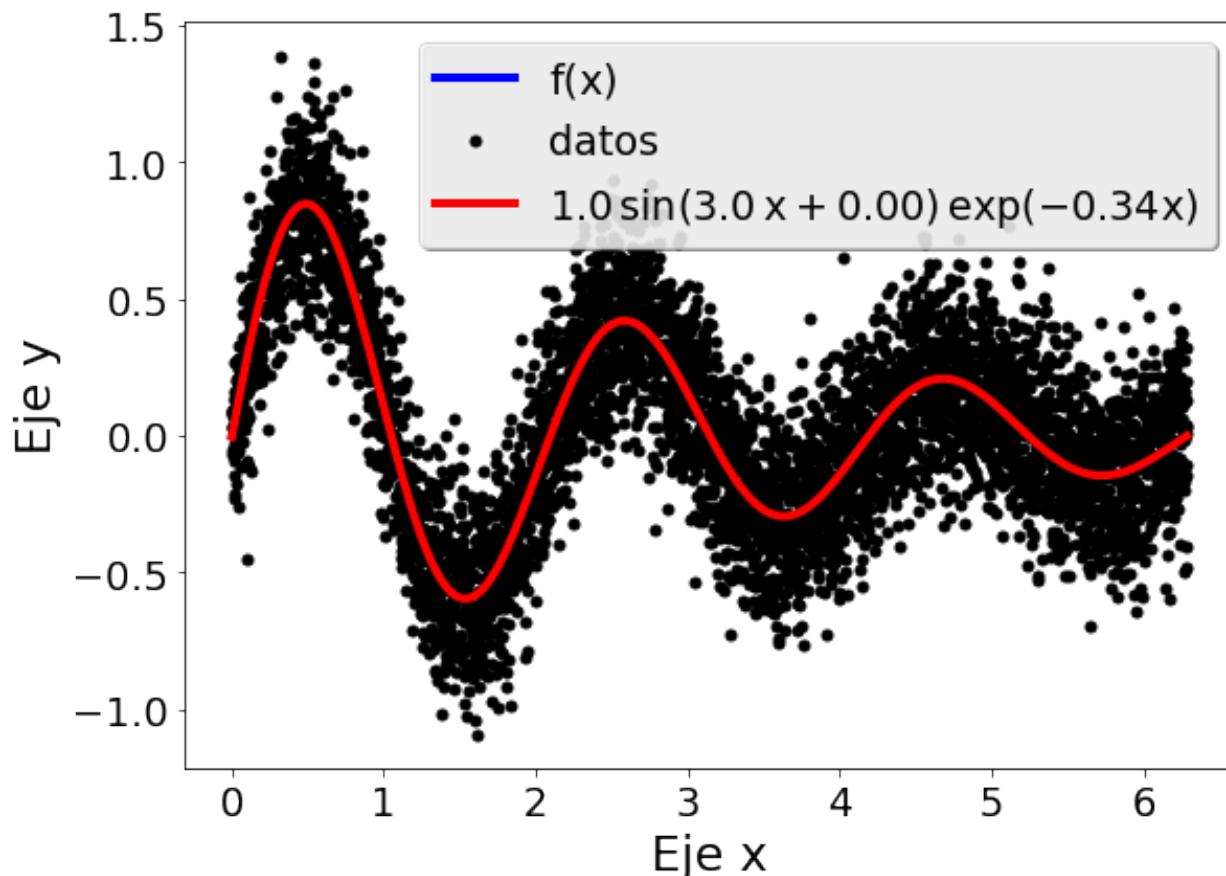


Veamos otro ejemplo similar, con muchos datos y dispersión

```
# Puntos "experimentales" con dispersión
x = np.linspace(0., 2*np.pi, 5000)
y0= fit_func(x, 1., 3., 1/3, 0 )
y = y0 + 0.2* np.random.normal(loc= 0., scale= 1, size= y0.size);
# Fiteo
```

```
initial_guess= [1., 3., 1., 0.2]
params, p_covarianza = curve_fit(fit_func, x, y, initial_guess)
```

```
# Graficación
plt.figure(figsize=fsiz)
plt.plot(x,y0,'-b',label="$f(x)$")
plt.plot(x,y,'.k',label='datos') # repeated from above
label=sfuncion.format(*params)
plt.plot(x,fit_func(x, *params), '-r', label=label)
plt.xlabel("Eje x") # labels again
plt.ylabel("Eje y")
plt.legend(loc='best');
```



La función `curve_fit()` que realiza el ajuste devuelve dos valores: El primero es un *array* con los valores de los parámetros obtenidos, y el segundo es un *array* con los valores correspondientes a la matriz de covarianza, cuyos elementos de la diagonal corresponden a la varianza para cada parámetro

```
np.diagonal(p_covarianza)
```

```
array([1.71310018e-04, 4.19376481e-05, 4.61186624e-05, 1.50808631e-04])
```

Así, por ejemplo el primer parámetro (correspondiente a la amplitud a) toma en estos casos el valor:

```
for j,v in enumerate(['a','b', 'c', 'd']):
    print("{} = {:.5g} ± {:.3g}".format(v, params[j], np.sqrt(p_covarianza[j,j])))
```

```
a = 1.0039 ± 0.0131
b = 2.9996 ± 0.00644
c = 0.33548 ± 0.00675
d = 0.0047726 ± 0.0123
```

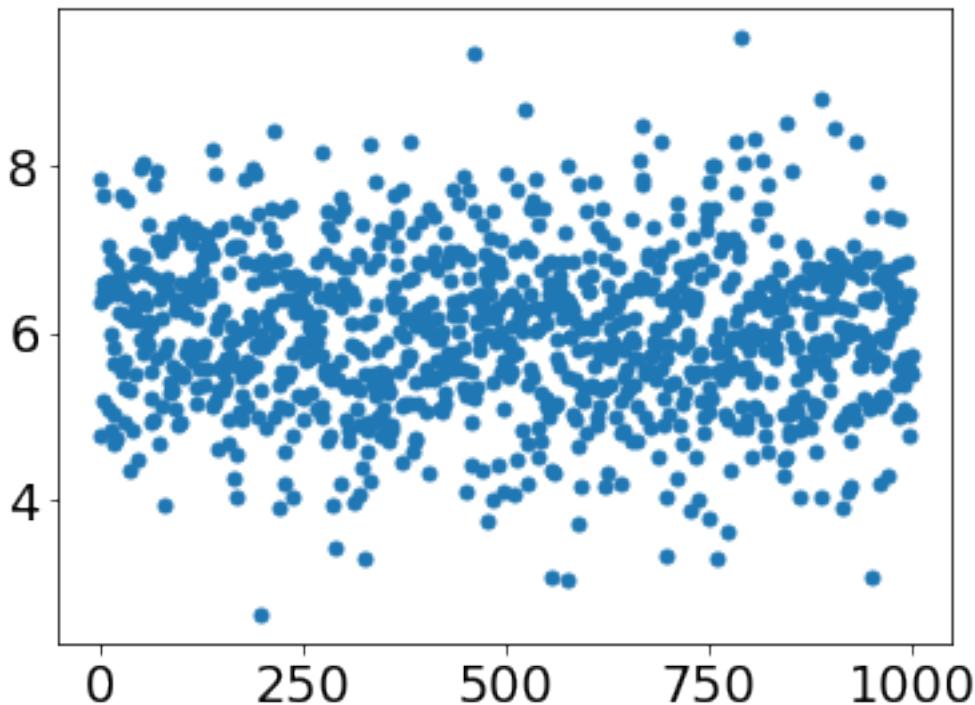
14.5.1 Ejemplo: Fiteo de picos

Vamos a suponer que los datos son obtenidos mediante la repetición de mediciones

```
# Realizamos 1000 mediciones, eso nos da una distribución de valores
r = np.random.normal(loc=6, size=1000)
```

```
# Veamos qué obtuvimos
plt.plot(r, '.')
```

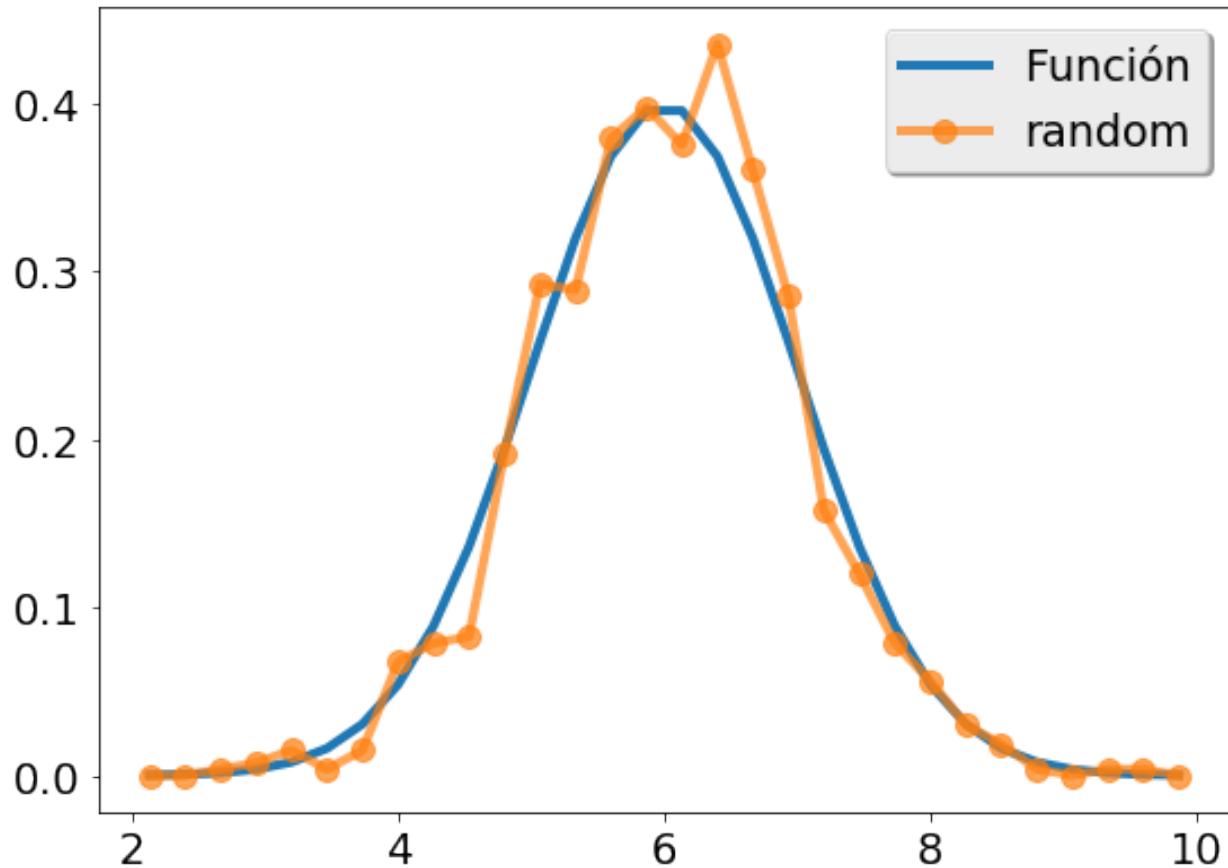
```
[<matplotlib.lines.Line2D at 0x7faba0290650>]
```



```
y, x = np.histogram(r, bins=30, range=(2,10), density=True)
x = (x[1:]+x[:-1])/2 # Calculamos los centros de los intervalos
```

Vamos a graficar el histograma junto con la función Gaussiana con el valor correspondiente de la Función Densidad de Probabilidad (pdf)

```
from scipy import stats
b = stats.norm.pdf(x, loc=6)
plt.figure(figsize=fsiz)
plt.plot(x,b,'-', label=u'Función')
plt.plot(x,y,'o-', alpha=0.7, label='random')
plt.legend(loc='best');
```

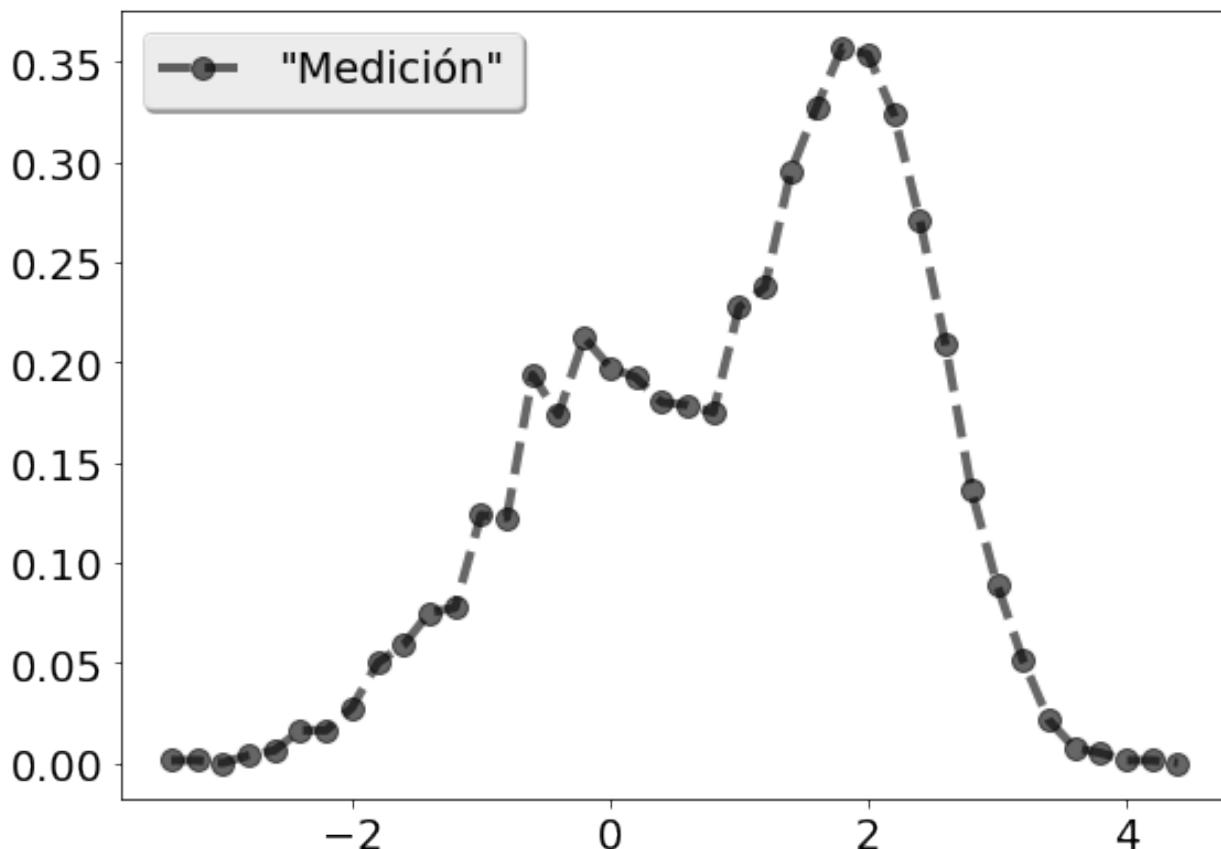


Con esta idea supongamos que tenemos datos, correspondiente a dos picos, diferentes pero que no podemos distinguirlos completamente porque se superponen.

Generemos los datos:

```
npoints= 2000
r = np.r_[np.random.normal(size=npoints), np.random.normal(loc=2, scale=.6, size=npoints)]
y,x = np.histogram(r , bins = 40, range = (-3.5,4.5), density=True)
x = (x[1:]+x[:-1])/2
```

```
plt.figure(figsize=fsize)
plt.plot(x,y,'o--k', alpha=0.6, label='Medición')
plt.legend(loc='best');
```



Ahora, por razones físicas (o porque no tenemos ninguna idea mejor) suponemos que esta curva corresponde a dos picos del tipo Gaussiano, sólo que no conocemos sus posiciones, ni sus anchos ni sus alturas relativas. Creamos entonces una función que describa esta situación: La suma de dos Gaussianas, cada una de ellas con su posición y ancho característico, y un factor de normalización diferente para cada una de ellas. En total tendremos seis parámetros a optimizar:

```
def modelo(x, *coeffs):
    "Suma de dos Gaussianas, con pesos dados por coeffs[0] y coeffs[3], respectivamente"
    return coeffs[0]*stats.norm.pdf(x, loc=coeffs[1], scale=coeffs[2]) + \
        coeffs[3]*stats.norm.pdf(x, loc=coeffs[4], scale=coeffs[5])
```

```
help(modelo)
```

```
Help on function modelo in module __main__:

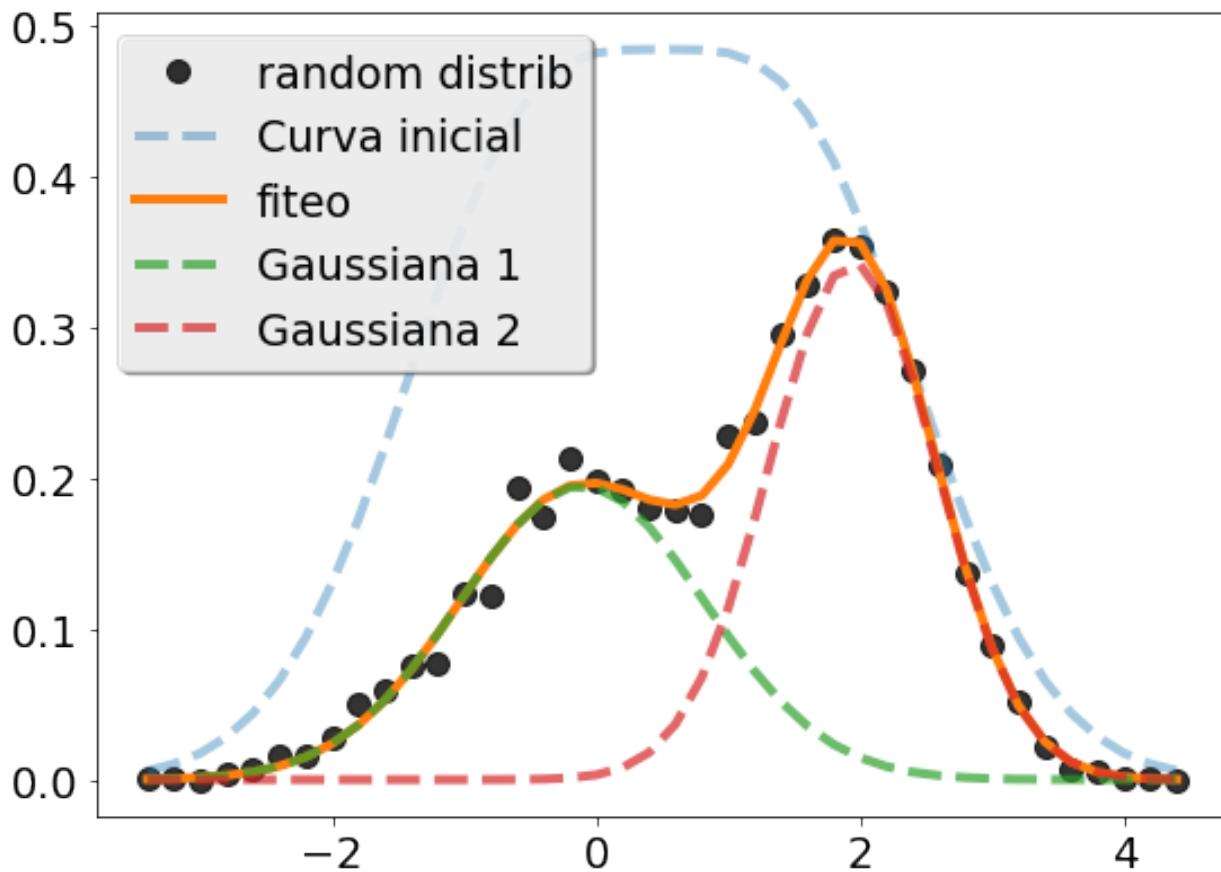
modelo(x, *coeffs)
    Suma de dos Gaussianas, con pesos dados por coeffs[0] y coeffs[3], respectivamente
```

Ahora es muy fácil realizar el fiteo. Mirando el gráfico proponemos valores iniciales, donde los tres primeros valores corresponde a los parámetros de la primera Gaussiana y los tres últimos a los de la segunda:

```
c0 = np.array([1., -0.5, 1., 1., 1.5, 1.])
c, cov = curve_fit(modelo, x, y, p0 = c0)
print(c)
```

```
[ 0.45146109 -0.10575116  0.92503718  0.54616588  1.94483535  0.63591132]
```

```
plt.figure(figsize=fsize)
plt.plot(x, y,'ok', alpha=0.8, label='random distrib')
plt.plot(x, modelo(x,*c0), '--', alpha=0.4, label='Curva inicial')
plt.plot(x, modelo(x,*c), '-.', label='fiteo')
plt.plot(x,c[0]*stats.norm.pdf(x,loc=c[1], scale=c[2]), '--', alpha=0.7, label=
    'Gaussiana 1')
plt.plot(x,c[3]*stats.norm.pdf(x,loc=c[4], scale=c[5]), '--', alpha=0.7, label=
    'Gaussiana 2')
plt.legend( loc = 'best' );
```



Veamos un ejemplo de ajuste de datos en el plano mediante la función

$$f(x, y) = (a \sin(2x) + b) \exp(-c(x + y - \pi)^2/4)$$

```
def func(x, a, b, c):
    return (a * np.sin(2 * x[0]) + b) * np.exp(-c * (x[1] + x[0] - np.pi)**2 / 4)

# Límites de los datos. Usados también para los gráficos
limits = [0, 2 * np.pi, 0, 2 * np.pi] # [x1_min, x1_max, x2_min, x2_max]

sx = np.linspace(limits[0], limits[1], 100)
sy = np.linspace(limits[2], limits[3], 100)
```

(continué en la próxima página)

(provine de la página anterior)

```

# Creamos las grillas
X1, X2 = np.meshgrid(sx, sy)
forma = X1.shape

# Los pasamos a unidimensional
x1_1d = X1.reshape((1, np.prod(forma)))
x2_1d = X2.reshape((1, np.prod(forma)))

# xdata[0] tiene los valores de x
# xdata[1] tiene los valores de y
xdata = np.vstack((x1_1d, x2_1d))

# La función original que vamos a ajustar.
# Sólo la usamos para comparar el resultado final con el deseado
original = (3, 1, 0.5)
z = func(xdata, *original)

# Le agregamos ruido. Estos van a ser los datos a ajustar
z_noise = z + .2 * np.random.randn(len(z))

# Hacemos el fiteo
ydata = z_noise
p0 = (1, 4, 1)
popt, pcov = curve_fit(func, xdata, ydata, p0=p0)

print(50*"-")
print("Valores de los parámetros")
print("----- -- --- -----")
print("Real    : {}\\nInicial: {}\\nAjuste : {}".format(original, p0, popt))

z_0 = func(xdata, *p0)
Z = z.reshape(forma)
Z_noise = z_noise.reshape(forma)
Z_0 = z_0.reshape(forma)

z_fit = func(xdata, *popt)
Z_fit = z_fit.reshape(forma)

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(12, 7))

titulos = [["Función Real", "Función inicial"],
           ["Datos con ruido", "Ajuste a los datos"]]

datos = [[Z, Z_0], [Z_noise, Z_fit]]

for k, a in np.ndenumerate(titulos):
    ax[k[0], k[1]].set_title(a)
    im = ax[k[0], k[1]].pcolormesh(X1, X2, datos[k[0]][k[1]])
    ax[k[0], k[1]].axis(limits)
    fig.colorbar(im, ax=ax[k[0], k[1]])

plt.subplots_adjust(hspace=0.3, wspace=0.)

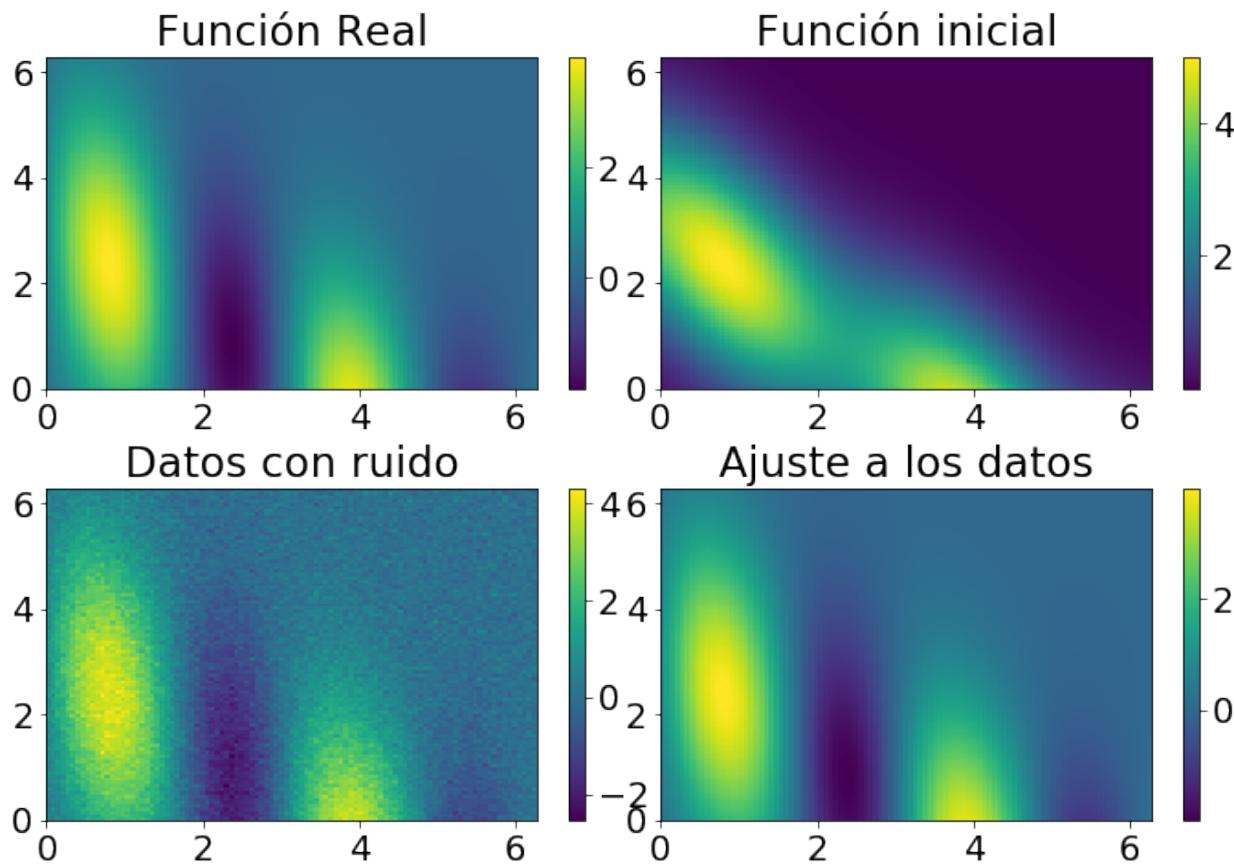
```

Valores de los parámetros

(continué en la próxima página)

(provien de la página anterior)

```
Real   : (3, 1, 0.5)
Inicial: (1, 4, 1)
Ajuste : [2.99645944 1.00044713 0.4988615 ]
```



En este gráfico tenemos:

- Arriba a la izquierda se encuentra para referencia la función real de la que derivamos los datos (que no se utiliza en los ajustes).
- Abajo a la izquierda se grafican los datos que se van a ajustar
- Arriba a la derecha se encuentra graficada la función con los parámetros iniciales, antes de realizar el ajuste. Como se ve, es muy diferente a los datos, y a la función deseada.
- Abajo a la derecha se grafica la función utilizando los parámetros que se obtienen con el ajuste

14.6 Ejercicios 13

1. En el archivo **co_nrg.dat** se encuentran los datos de la posición de los máximos de un espectro de CO₂ como función del número cuántico rotacional J (entero). Haga un programa que lea los datos. Los ajuste con polinomios (elija el orden adecuado) y grafique luego los datos (con símbolos) y el ajuste con una línea sólida roja. Además, debe mostrar los parámetros obtenidos para el polinomio.
2. Queremos hacer un programa que permita fittear una curva como suma de N funciones gaussianas:

1. Haga una función, que debe tomar como argumento los arrays con los datos: x , y , y valores iniciales para las Gaussianas: `fit_gaussianas(x, y, *params)` donde `params` son los $3N$ coeficientes (tres coeficientes para cada Gaussiana). Debe devolver los parámetros óptimos obtenidos.
 2. Realice un programita que grafique los datos dados y la función que resulta de sumar las gaussianas en una misma figura.
 3. *Si puede* agregue líneas o flechas indicando la posición del máximo y el ancho de cada una de las Gaussianas obtenidas.
-

CAPÍTULO 15

Clase 14: Gráficos interactivos

15.1 Trabajo simple con imágenes

Vamos a empezar leyendo y mostrando algunas imágenes, para entender cómo es su representación. Para leer y escribir imágenes vamos a usar el paquete adicional `imageio`. **Scipy** tiene funciones (con el mismo nombre) para realizar este trabajo en el submódulo `misc` pero está planeado que desaparezcan en un futuro no muy lejano.

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

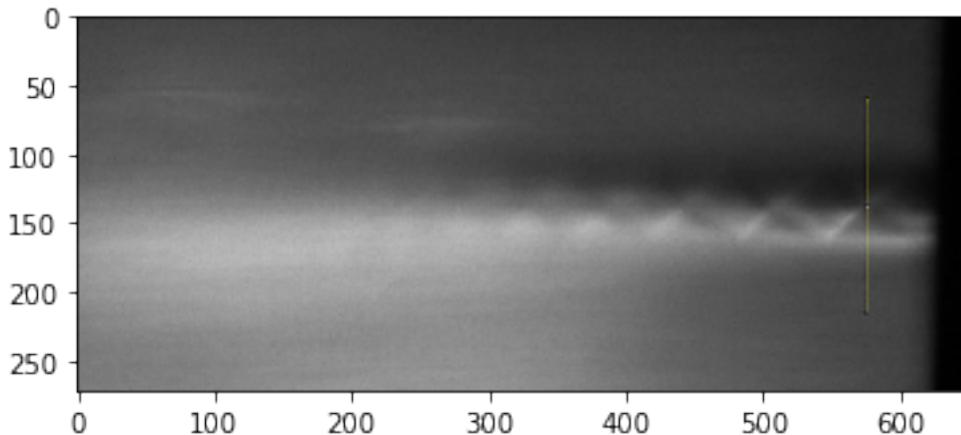
El paquete `imageio` debe ser instalado separadamente. Si no está presente en el sistema, usa la función `imread()` de `scipy.misc` (que emite un mensaje avisando que pronto no va a estar disponible)

```
try:
    from imageio import imread
except:
    from scipy.misc import imread
```

El siguiente ejemplo es una figura tomada de algunas pruebas que hicimos en el laboratorio hace unos años. Es el resultado de la medición de flujo en toberas

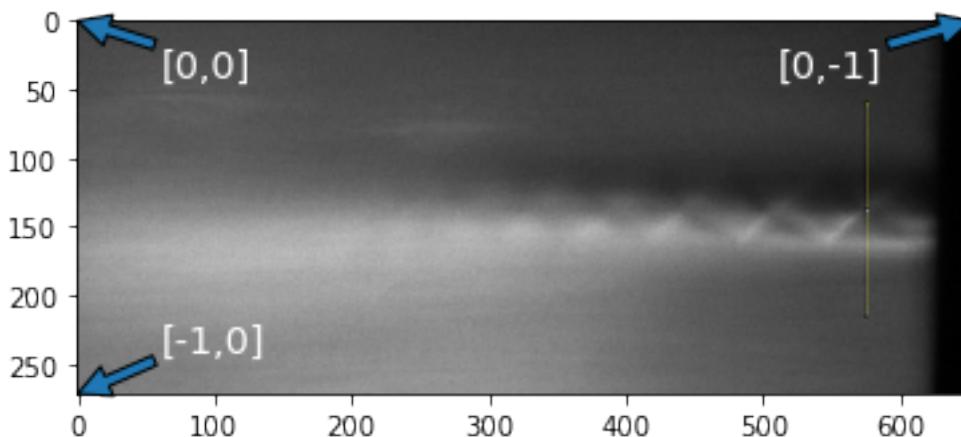
```
imag1= imread('figuras/imagen_flujo.jpg')
print('La imagen "imag1" es del tipo:', type(imag1), 'con "shape"', imag1.shape)
plt.imshow(imag1);
```

```
La imagen "imag1" es del tipo: <class 'imageio.core.util.Array'> con "shape" (272, 652, 3)
```



La representación de la imagen es una matriz, donde cada elemento corresponde a un pixel, y cada pixel tiene tres valores. El elemento $[0, 0]$ corresponde al pixel ubicado en la esquina superior izquierda, el elemento $[-1, 0]$ al pixel ubicado en la esquina inferior izquierda, mientras que el $[0, -1]$ a la esquina superior derecha:

```
plt.imshow(imag1)
color='white'
plt.annotate("[0,0]",(0,0), (60,40), arrowprops={}, fontsize='x-large', color=color)
plt.annotate("[-1,0]",(0,272), (60,240), arrowprops={}, fontsize='x-large', color=color)
plt.annotate("[0,-1]",(652,0), (510,40), arrowprops={}, fontsize='x-large', color=color);
```



En consecuencia podemos ver qué valores toma cada pixel

```
print(imag1[0,0])          # El primer elemento
print(imag1[0,1])          # El segundo elemento
print(imag1.min(),imag1.max()) # y sus valores mínimo y máximo
```

```
[65 65 65]
[66 66 66]
0 255
```

Como vemos en cada pixel el valor está dado por un array de tres números enteros

```
imag1.dtype
```

```
dtype('uint8')
```

Como originalmente teníamos una figura en escala de grises, podemos convertir los tres colores a una simple escala, por ejemplo promediando los tres valores. La función `imread()` puede interpretar la figura como una escala de grises con los argumentos adecuados:

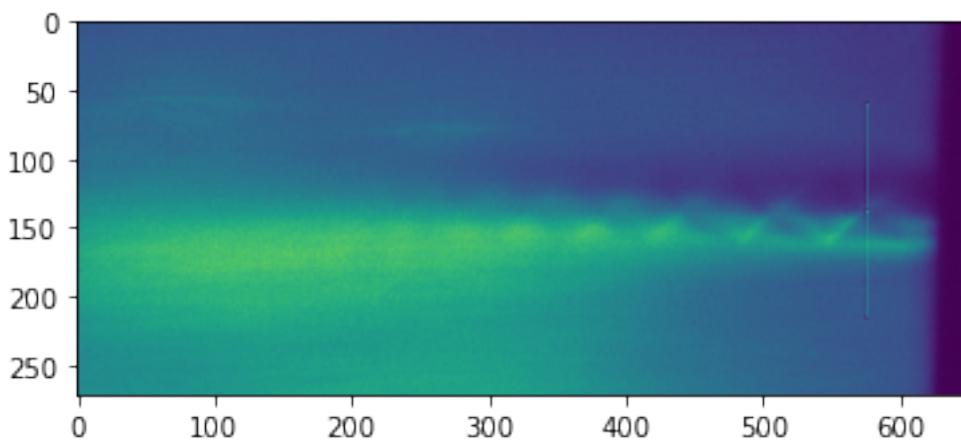
```
imag2= imread('figuras/imagen_flujo.jpg', as_gray=True)
```

La variable `imag2` contiene ahora una matriz con las dimensiones de la imagen (272 x 652) pero con sólo un valor por cada pixel

```
print(imag2.shape)
print(imag2[0,0])
```

```
(272, 652)
65.0
```

```
plt.imshow(imag2);
```

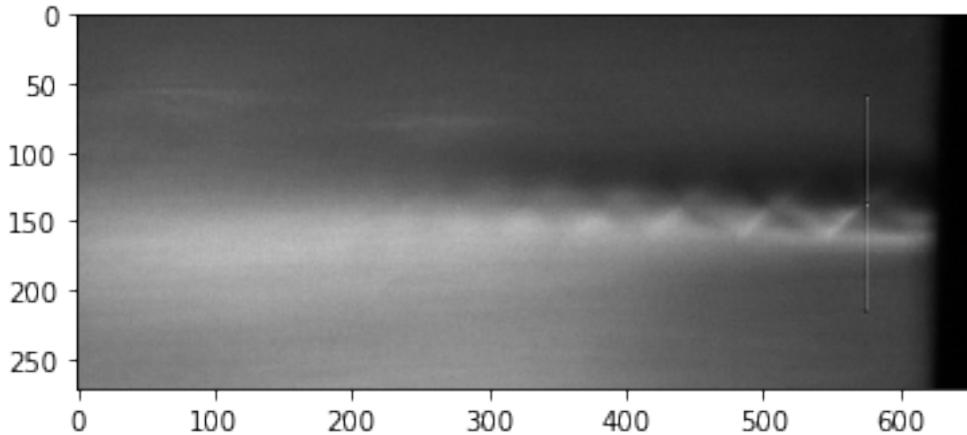


Nota ¿Qué pasó acá?

La función `imshow()` está interpretando el valor de cada pixel como una posición en una cierta escala de colores (**colormap**). Como no especificamos cuál queremos utilizar, se usó el `cmap default`.

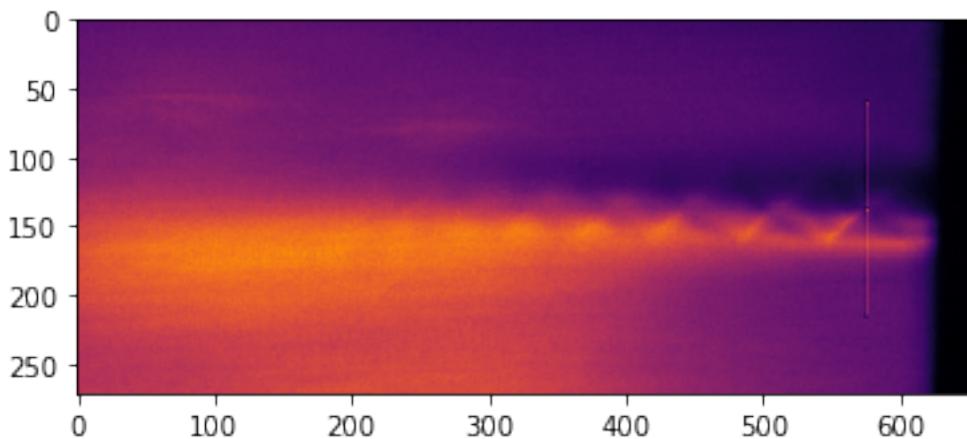
Especifiquemos el **colormap** a utilizar para la graficación:

```
plt.imshow(imag2, cmap='gray');
```

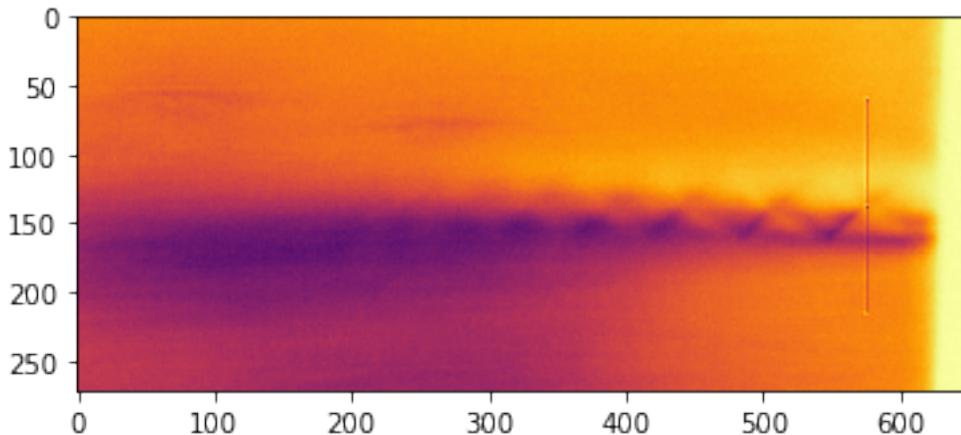


Los `cmap` aparecen también en el contexto de graficación de gráficos de contorno (`contour()` y `contourf()`). Al asociar un valor a un mapa de colores, la misma imagen puede mostrarse de diferentes maneras. Veamos otros ejemplos de *colormap*

```
plt.imshow(imag2, cmap='inferno');
```

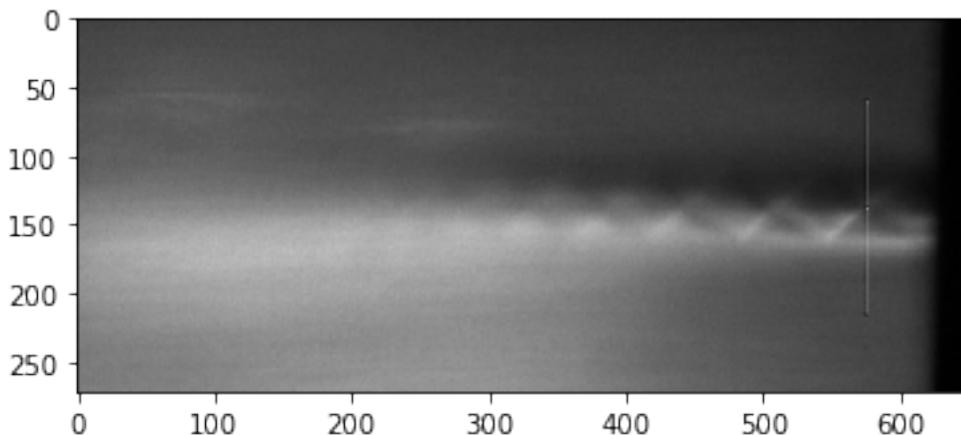


```
plt.imshow(imag2, cmap='inferno_r');
```



La referencia de ubicación de los `cmap` existentes está en: http://matplotlib.org/examples/color/colormaps_reference.html

```
plt.imshow(imag2, cmap='gray');
```



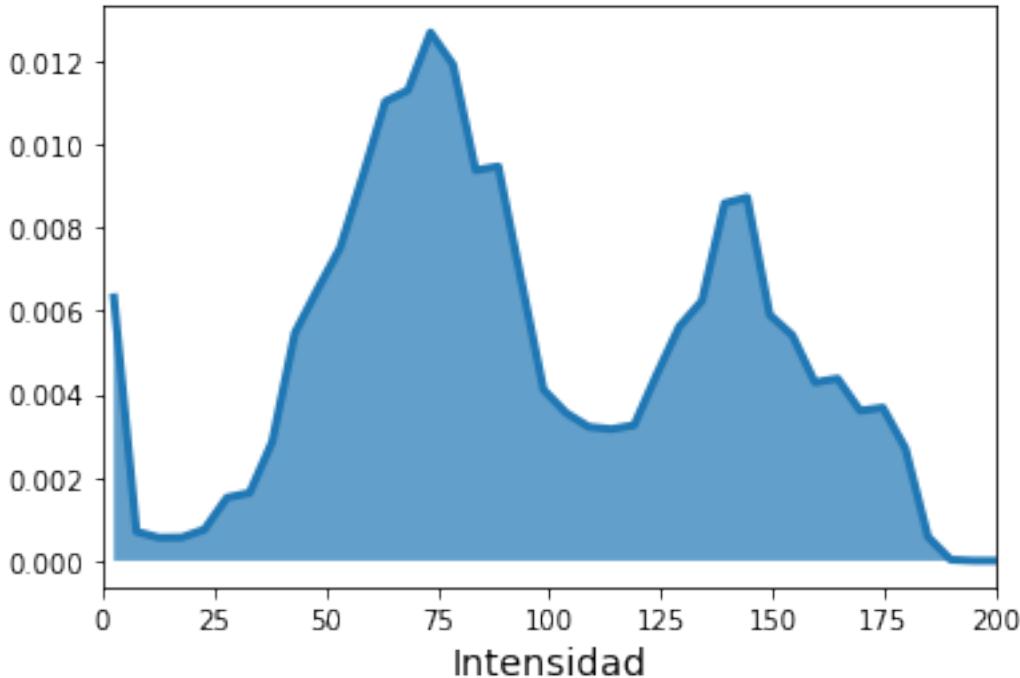
15.1.1 Análisis de la imagen

La imagen es una representación en mapa de colores de los valores en la matriz. Esta es una representación que da muy buena información cualitativa sobre las características de los datos. Para analizar los datos a veces es más fácil hacer cortes o promediar en alguna dirección los datos.

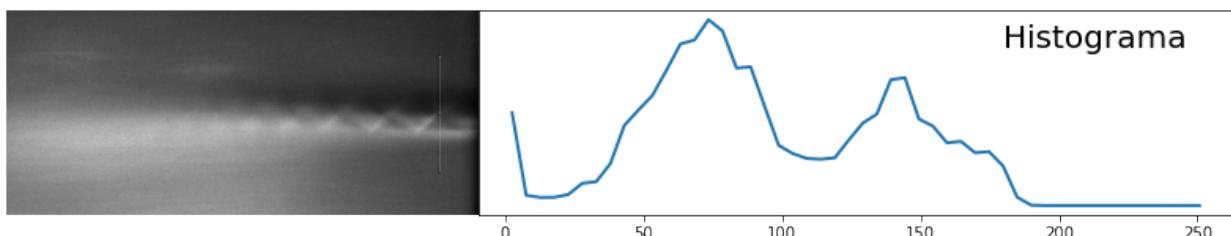
Histograma de intensidades

Un ejemplo es el cálculo de un histograma de intensidades, analizando toda la imagen.

```
hist, bin_edges = np.histogram(imag2, bins=50, density=True)
bin_centers = 0.5*(bin_edges[:-1] + bin_edges[1:])
plt.fill_between(bin_centers, 0, hist, alpha=0.7)
plt.plot(bin_centers, hist, color='C0', lw=3)
plt.xlabel('Intensidad')
plt.xlim(0,200);
```



```
# Creamos una figura con los dos gráficos
fig, ax= plt.subplots(figsize=(10,2), ncols=2)
# En el gráfico de la izquierda mostramos la imagen en escala de grises
ax[0].imshow(imag2, cmap=plt.cm.gray, interpolation='nearest')
ax[0].axis('off') # Eliminamos los dos ejes
#
# Graficamos a la derecha el histograma
ax[1].plot(bin_centers, hist, lw=2)
ax[1].text(180, 0.85*hist.max(), 'Histograma', fontsize=20)
ax[1].set_yticks([]) # Sólo valores en el eje x
plt.subplots_adjust(wspace=-0.20, top=1, bottom=0.1, left=-0.2, right=1)
```



Estos histogramas, son útiles pero no dan información sobre las variaciones de intensidad con la posición. De alguna manera estamos integrando demasiado. Cuando vemos la imagen, vemos un mapa de intensidad en dos dimensiones. Al hacer el histograma sobre toda la figura perdemos completamente la información sobre la posición.

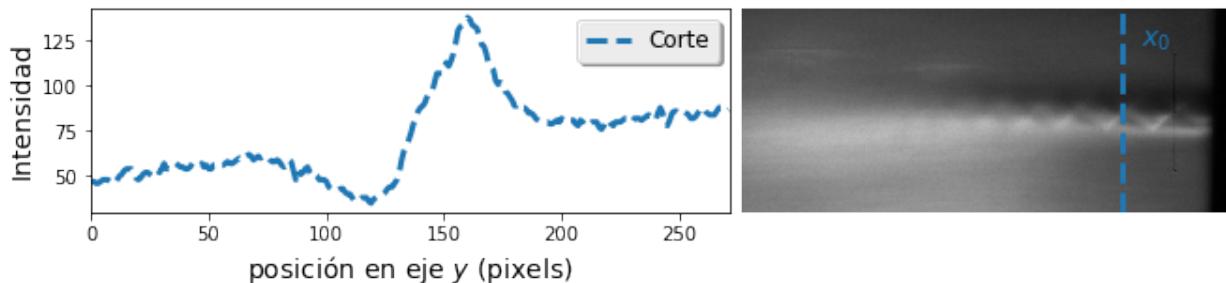
Cortes en una dirección

Un análisis intermedio podemos hacerlo haciendo cortes a lo largo de alguna línea y analizando la intensidad. Por ejemplo, podemos elegir una línea vertical en un punto x_0 , y analizar como varía la intensidad a lo largo de esa línea:

```
x0 = int(imag2.shape[1]*7/9)      # Elegimos un punto en el eje x
print('posición en eje x={} de un total de {}'.format(x0, imag2.shape[1]))
```

posición en eje x=507 de un total de 652

```
# Creamos la figura con dos subplots
fig, (ax2, ax1) = plt.subplots(ncols=2, figsize=(10,2))
# graficamos la imagen en el subplot de la derecha
ax1.imshow(imag2, cmap=plt.cm.gray)
# y agregamos la línea vertical en el punto elegido
ax1.axvline(x0, ls='--', lw=3)
ax1.text(1.05*x0, 50, '$x_0$', fontsize='x-large', color='C0')
ax1.axis('off')
#
# Creamos linea como un array 1D con los datos a lo largo de la línea deseada
# y la graficamos
linea = imag2[:,x0]
ax2.plot(linea, '--', lw=3, label='Corte')
ax2.set_xlabel(u'posición en eje $y$ (pixels)')
ax2.set_ylabel('Intensidad')
ax2.legend(loc='best')
ax2.set_xlim((0,len(linea)))
# Ajustamos el margen izquierdo y la distancia entre los dos subplots
plt.subplots_adjust(wspace=-0.1, left=0)
```



15.2 Ejercicios 14 (a)

1. Modificar el ejemplo anterior para presentar en una figura tres gráficos, agregando a la izquierda un panel donde se muestre un corte horizontal. El corte debe estar en la mitad del gráfico ($y_0 = 136$). En la figura debe mostrar la posición del corte (similarmente a como se hizo con el corte en x) con una línea de otro color.

15.3 Gráficos interactivos (widgets)

Veamos cómo se puede hacer este tipo de trabajo en forma interactiva. Para ello **Matplotlib** tiene un submódulo `widgets` con rutinas que están diseñadas para funcionar con cualquier *backend* interactivo. (más información en: http://matplotlib.org/api/widgets_api.html)

15.3.1 Cursor

Empecemos estudiando como agregar un indicador de la posición del cursor a un gráfico.

```
# Archivo: ejemplo_cursor.py

from matplotlib.widgets import Cursor
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(8, 6))

x, y = 4 * (np.random.rand(2, 100) - .5)
ax.plot(x, y, 'o')
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

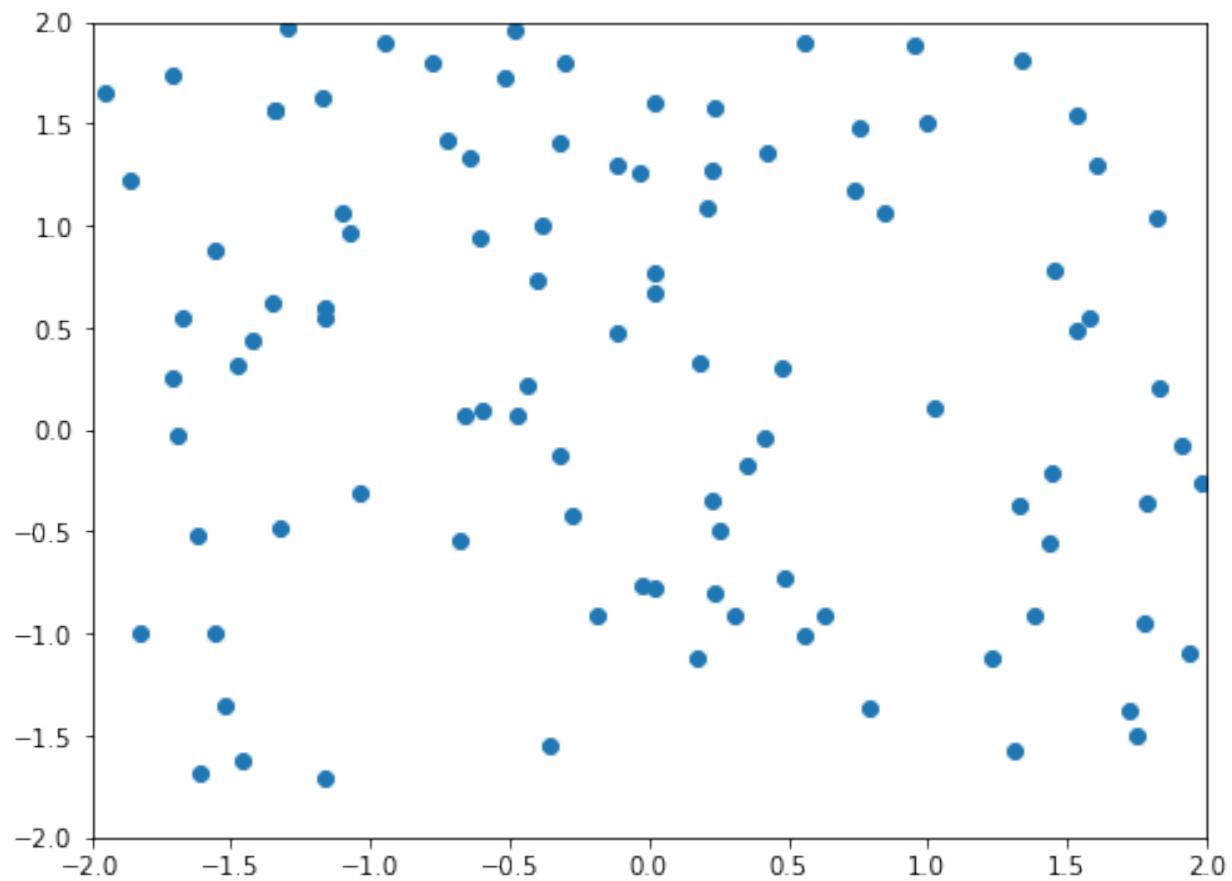
# Usamos: useblit = True en el backend gtkagg
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)

plt.show()
```

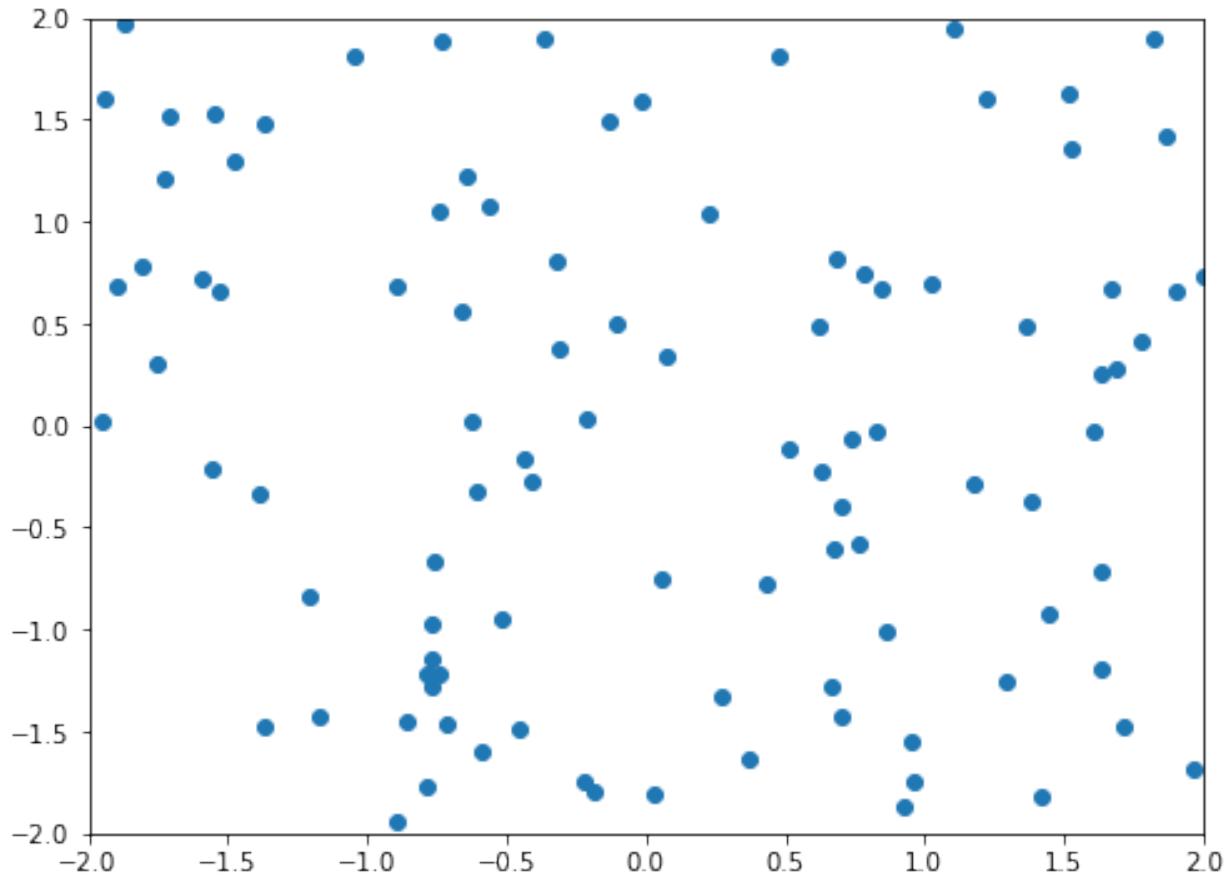
En este caso, el programa está escrito separadamente y lo podemos ejecutar desde la notebook (como en este caso) o desde una terminal independientemente. Para ejecutar un script en forma interactiva desde la *notebook* de *Jupyter* debemos setear el *backend* a una opción interactiva (que no es el valor por default en las notebooks). En este caso vamos a usar el backend *tk*

```
fig, ax = plt.subplots(figsize=(8, 6))

x, y = 4 * (np.random.rand(2, 100) - .5)
ax.plot(x, y, 'o')
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2);
```



```
%matplotlib tk  
import matplotlib.pyplot as plt  
%run scripts/ejemplo_cursor.py
```



En este ejemplo simple conocemos casi todas las líneas (creamos la figura y graficamos). Las líneas novedosas y relevantes son

1. La primera línea importando la función `Cursor()` para describir el *cursor* o *mouse*:

```
from matplotlib.widgets import Cursor
```

2. La línea en que usamos la función `Cursor`

```
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)
```

que crea el objeto `Cursor`. La forma de esta función es:

```
Cursor(ax, horizOn=True, vertOn=True, useblit=False, **lineprops)
```

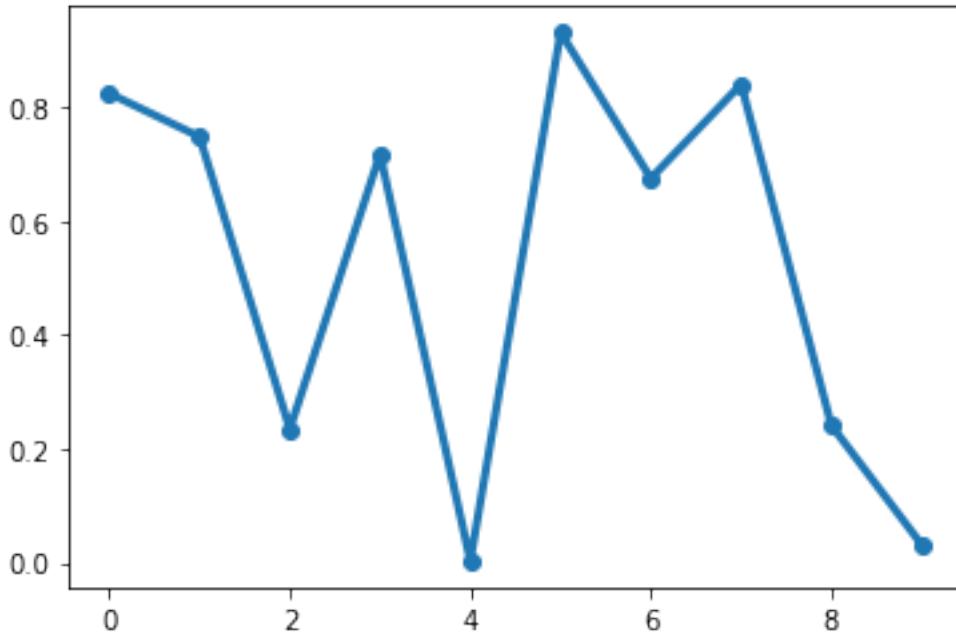
y toma como argumento el eje en el cuál agregamos el cursor. Como argumentos opcionales se puede controlar la visibilidad de la línea horizontal (`horizOn`) y vertical (`vertOn`) que pueden tomar valores lógicos `True` o `False`. Además tiene argumentos *keyword* para controlar la apariencia de las líneas. En este ejemplo pusimos explícitamente que queremos una línea roja con un grosor igual a 2.

15.3.2 Manejo de eventos

Para que la interactividad sea útil es importante obtener datos de nuestra interacción con el gráfico. Esto se obtiene con lo que se llama manejo de eventos (Event handling).

Para recibir *events* necesitamos **escribir y conectar** una función que se activa cuando ocurre el evento (*callback*). Veamos un ejemplo simple pero importante, donde imprimimos las coordenadas donde se presiona el *mouse*.

```
# Mostramos la figura (sin interactividad)
%matplotlib inline
%run scripts/ejemplo_callback.py
```

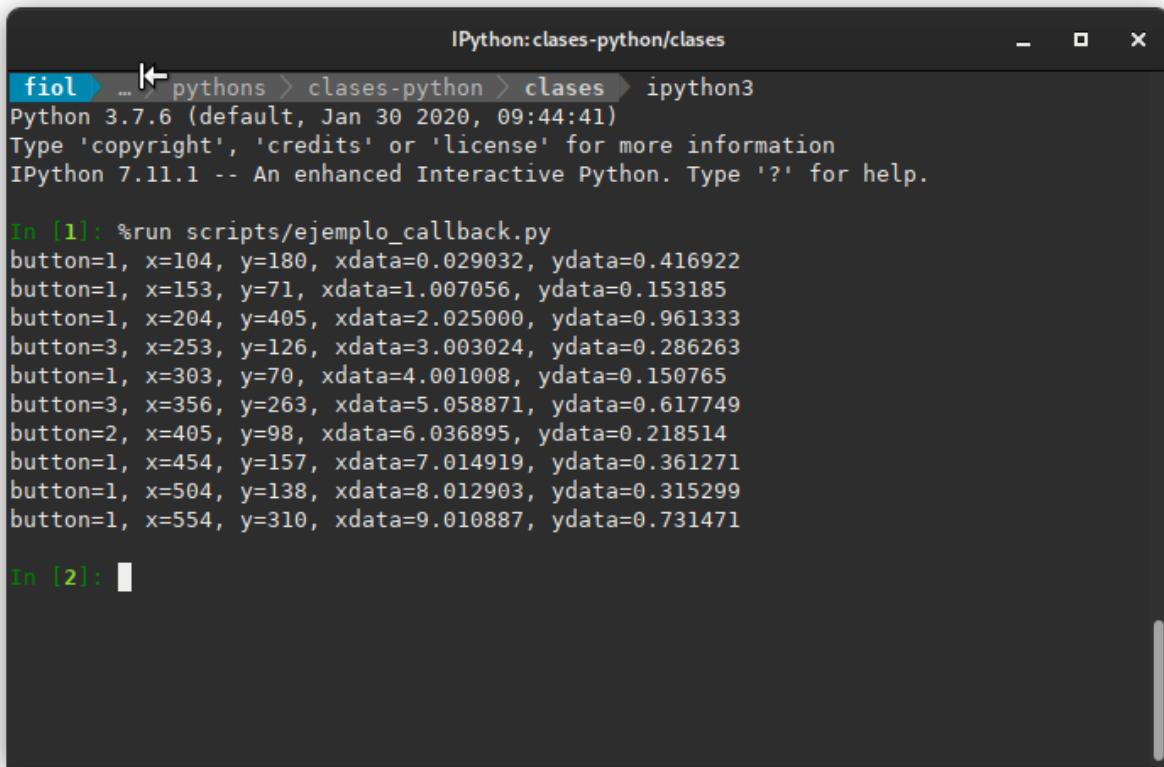


```
# Archivo: ejemplo_callback.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(np.random.rand(10), 'o-', lw=3)

def onclick(event):
    print('button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          (event.button, event.x, event.y, event.xdata, event.ydata))

cid = fig.canvas.mpl_connect('button_press_event', onclick)
plt.show()
```



The screenshot shows an IPython terminal window titled "IPython: clases-pyton/clases". The command line shows the path: "fiol > ... > pythons > clases-pyton > clases > ipython3". The Python version is "Python 3.7.6 (default, Jan 30 2020, 09:44:41)". It displays the following text:

```
In [1]: %run scripts/ejemplo_callback.py
button=1, x=104, y=180, xdata=0.029032, ydata=0.416922
button=1, x=153, y=71, xdata=1.007056, ydata=0.153185
button=1, x=204, y=405, xdata=2.025000, ydata=0.961333
button=3, x=253, y=126, xdata=3.003024, ydata=0.286263
button=1, x=303, y=70, xdata=4.001008, ydata=0.150765
button=3, x=356, y=263, xdata=5.058871, ydata=0.617749
button=2, x=405, y=98, xdata=6.036895, ydata=0.218514
button=1, x=454, y=157, xdata=7.014919, ydata=0.361271
button=1, x=504, y=138, xdata=8.012903, ydata=0.315299
button=1, x=554, y=310, xdata=9.010887, ydata=0.731471

In [2]:
```

En este ejemplo utilizamos el método `mpl_connect` del objeto `canvas`.

- El objeto `canvas` es el área donde se dibuja la figura.
- La función `mpl_connect` realiza la conexión de la función (que aquí llamamos `onclick`) con la figura. Esta función toma como argumento el *event* (que, para nosotros, es interpretado automáticamente por Matplotlib)
- El objeto `event` es de tipo `button_press_event`. Se dispara cuando apretamos un botón del *mouse* y `matplotlib` le pasa como argumento un objeto del tipo `event` que contiene información. Nosotros estamos imprimiendo la siguiente información que contiene `event` por pantalla:
 - `event.button`: indica que botón se presionó
 - `event.x, event.y`: dan la información sobre el índice en los ejes horizontal y vertical
 - `event.xdata, event.ydata`: dan los valores de los datos en los ejes.

Puede leer más información sobre el manejo de eventos en http://matplotlib.org/users/event_handling.html.

15.3.3 Ejemplos integrados

Continuando con esta idea vamos a usar la capacidad de poder interactuar con el gráfico para elegir una zona del gráfico de la cual obtenemos información sobre los datos.

```
# Archivo: scripts/analizar_figura_1.py

import matplotlib.pyplot as plt
import imageio
from matplotlib.widgets import Cursor
```

(continué en la próxima página)

(proviene de la página anterior)

```

img = imageio.imread('../figuras/imagen_flujo.png', as_gray=True)
ymax = img.max()

def seleccionar(event):
    """Secuencia:
    1. Encuentro el punto donde el mouse hizo 'click'
    2. Le doy valores a la linea vertical
    3. Le doy valores a la curva en el grafico de la derecha
    4. y 5. Grafico los nuevos valores
    """
    x0 = event.xdata
    n0 = int(x0)
    11.set_data([[n0, n0], [0., 1.]])
    12.set_data(range(img.shape[0]), img[:, n0])
    11.figure.canvas.draw()
    12.figure.canvas.draw()

# Defino la figura
fig, (ax1, ax2) = plt.subplots(figsize=(12, 4), ncols=2)

# Mostramos la imagen como un mapa de grises
ax1.imshow(img, cmap='gray', interpolation='nearest')
ax1.axis('off')

# Agrego la linea inicial en un valor inicial
x0 = 100
11 = ax1.axvline(x0, color='r', ls='--', lw=3)

# Grafico de la derecha
12, = ax2.plot(img[:, x0], 'r-', lw=2, label='corte')
ax2.set_ylim(0, ymax)
ax2.set_xlabel(u'posición en eje $y$ (pixels)')
ax2.set_ylabel('Intensidad')
ax2.legend(loc='best')

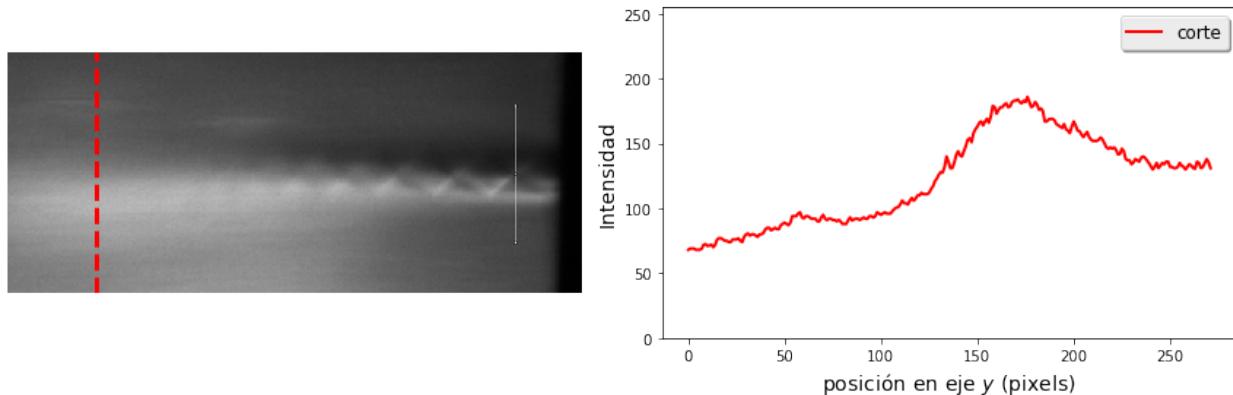
fig.tight_layout()

# Agrego el cursor y conecto la accion de presionar a la funcion click
cursor = Cursor(ax1, horizOn=False, vertOn=True, useblit=True,
                 color='blue', linewidth=1)
fig.canvas.mpl_connect('button_press_event', seleccionar)

plt.show()

```

```
%matplotlib tk
%run scripts/analizar_figura_1.py
```



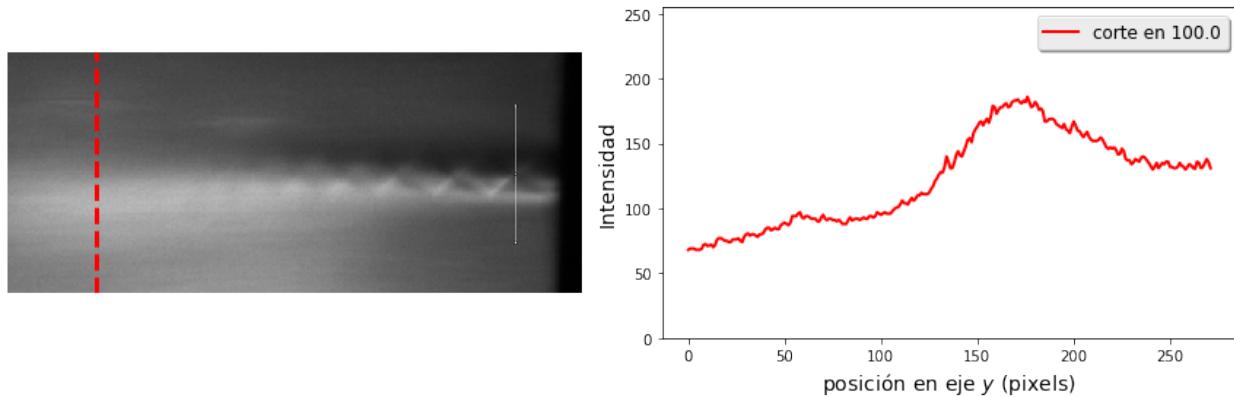
Este es un ejemplo un poco más largo (y un poquito más complejo).

1. Importamos los módulos y funciones necesarias.
 - `matplotlib.pyplot as plt` para casi todo
 - `imageio` para leer la figura
 - `from matplotlib.widgets import Cursor` para importar el objeto `Cursor` que nos muestra la posición del mouse.
2. Leemos la imagen de archivo, creamos la figura y la mostramos.
3. Elegimos un valor de x inicial (igual a 100) y agregamos una línea vertical en ese punto.
4. Creamos la figura de la derecha con los datos tomados de la columna correspondiente de la matriz que representa la imagen.
5. Agregamos *labels* y ajustamos las distancias
6. Mostramos el cursor y le conectamos el evento (standard) `button_press_event` a nuestra función `seleccionar()`.
7. La función `seleccionar()` toma como argumento el evento que se dispara por interacción con el usuario. El argumento `event` lo pasa automáticamente `Matplotlib`. En este caso es un *click* del mouse en una zona del gráfico.

La función `seleccionar(event)` : 1. Del argumento `event` extrae la posición en el eje horizontal y lo asigna a la variable `x0`. 2. El índice en el eje horizontal (`n0`). 2. Actualiza los datos de la línea `l1` con valores para la línea vertical en el panel izquierdo. 4. Actualiza la línea `l2` en la figura de la derecha con el corte en `x0`. 5. Actualiza el dibujo de las líneas sobre el *canvas*.

El siguiente ejemplo es muy similar al anterior. Sólo estamos actualizando la leyenda, para tener información del punto seleccionado.

```
%run scripts/analizar_figura_2.py
```



```
# Archivo: analizar_figura_2.py
import imageio
import matplotlib.pyplot as plt
from scipy import misc
from matplotlib.widgets import Cursor

img = imageio.imread('../figuras/imagen_flujo.png', as_gray=True)
ymax = img.max()

def click(event):
    """Secuencia:
    1. Encuentro el punto donde el mouse hizo 'click'
    2. Le doy valores a la linea vertical
    3. Le doy valores a la curva en el grafico de la derecha
    4. y 5. Grafico los nuevos valores
    """
    x0 = event.xdata
    n0 = int(x0)
    l1.set_data([[n0, n0], [0., 1.]])
    l2.set_data(range(img.shape[0]), img[:, n0])
    leg2.texts[0].set_text('corte en {:.1f}'.format(x0))
    l1.figure.canvas.draw()
    l2.figure.canvas.draw()

# Defino la figura
# Defino la figura
fig, (ax1, ax2) = plt.subplots(figsize=(12, 4), ncols=2)

ax1.imshow(img, cmap="gray", interpolation='nearest')
ax1.axis('off')
# Agrego la linea inicial en un valor inicial
x0 = 100
l1 = ax1.axvline(x0, color='r', ls='--', lw=3)

# Grafico de la derecha
l2, = ax2.plot(img[:, x0], 'r-', lw=2, label='corte en {:.1f}'.format(x0))
ax2.set_xlim((0, ymax))
ax2.set_xlabel(u'posición en eje $y$ (pixels)')
ax2.set_ylabel('Intensidad')
leg2 = ax2.legend(loc='best')
```

(continué en la próxima página)

(proviene de la página anterior)

```
fig.tight_layout()

# Agrego el cursor y conecto la accion de presionar a la funcion click
cursor = Cursor(ax1, horizOn=False, vertOn=True, useblit=True,
                 color='blue', linewidth=1)
fig.canvas.mpl_connect('button_press_event', click)

plt.show()
```

Las diferencias más notables con el ejemplo anterior son:

1. Al crear la leyenda, asignamos el objeto creado a la variable `leg2`, en la línea:

```
leg2 = ax2.legend(loc='best')
```

2. En la función `click(event)` (equivalente a seleccionar (evento) en el ejemplo anterior) actualizamos el texto de la leyenda con el valor de `x0`:

```
leg2texts[0].set_text('corte en {:.1f}'.format(x0))
```

15.4 Ejercicios 14 (b)

2. Modificar el ejemplo anterior ([anализировать_фигуру_2.py](#)) para presentar tres gráficos, agregando a la izquierda un panel donde se muestre el corte horizontal de la misma manera que en el ejercicio anterior. Al seleccionar con el mouse debe mostrar los dos cortes (horizontal y vertical).
-

CAPÍTULO 16

Ejercicios

16.1 Ejercicios de Clase 01

1. Abra una terminal (consola) de Ipython y utilícela como una calculadora para realizar las siguientes acciones:
 - Suponiendo que, de las cuatro horas de clases, tomamos un descanso de 15 minutos y nos distraemos otros 13 minutos, calcular cuántos minutos efectivos de trabajo tendremos en las 16 clases.
 - Para la cantidad de alumnos presentes en el aula: ¿cuántas horas-persona de trabajo hay involucradas?
2. Muestre en la consola de Ipython:
 - el nombre de su directorio actual
 - los archivos en su directorio actual
 - Cree un subdirectorio llamado `tmp`
 - si está usando linux, la fecha y hora
 - Borre el subdirectorio `tmp`
3. Abra un editor de textos y escriba las líneas necesarias para imprimir por pantalla las siguientes frases (una por línea). Guarde y ejecute su programa.
 - Hola, por primera vez
 - Hola, hoy es mi día de escribir frases intrascendentes
 - Hola, nuevamente, y espero que por última vez
 - $E = mc^2$
 - Adiós

Ejecute el programa.

Inicie una terminal de *Jupyter* y realice las siguientes operaciones:

4. Para cubos de lados de longitud $L = 1, 3, 5$ y 8 , calcule su superficie y su volumen.

5. Para esferas de radios $r = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
6. Fíjese si alguno de los valores de $x = 2,05, x = 2,11, x = 2,21$ es un cero de la función $f(x) = x^2 + x/4 - 1/2$.
7. Para el número complejo $z = 1 + 0,5i$
 - Calcular z^2, z^3, z^4, z^5 .
 - Calcular los complejos conjugados de z, z^2 y z^3 .
 - Escribir un programa, utilizando formato de strings, que escriba las frases:
 - El conjugado de $z=1+0.5j$ es $1-0.5j$
 - El conjugado de $z=(1+0.5j)^2$ es (con el valor correspondiente)

16.2 Ejercicios de Clase 02

1. Centrado manual de frases

a. Utilizando la función `len()` centre una frase corta en una pantalla de 80 caracteres. Utilice la frase:
Primer ejercicio con caracteres

b. Agregue subrayado a la frase anterior

2. Para la cadena de caracteres:

```
s = '''Aquí me pongo a cantar
Al compás de la vigüela,
Que el hombre que lo desvela
Una pena estraordinaria
Como la ave solitaria
Con el cantar se consuela.'''
```

- Cuente la cantidad de veces que aparecen los substrings `es`, `la`, `que`, `co`, en los siguientes dos casos: distinguiendo entre mayúsculas y minúsculas, y no distinguiendo. Imprima el resultado.
- Cree una lista, donde cada elemento es una línea del string `s` y encuentre la de mayor longitud. Imprima por pantalla la línea y su longitud.
- Forme un nuevo string de 10 caracteres que contenga los 5 primeros y los 5 últimos del string anterior `s`. Imprima por pantalla el nuevo string.
- Forme un nuevo string que contenga los 10 caracteres centrales de `s` (utilizando un método que pueda aplicarse a otros strings también). Imprima por pantalla el nuevo string.
- Cambie todas las letras `m` por `n` y todas las letras `n` por `m` en `s`. Imprima el resultado por pantalla.
- Debe entregar un programa llamado `02_SuApellido.py` (con su apellido, no la palabra `SuApellido`). El programa al correrlo con el comando `python3 SuApellido_02.py` debe imprimir:

```
Nombre Apellido
Clase 2
Distinguiendo: 2 5 1 2
Sin distinguir: 2 5 2 4
Que el hombre que lo desvela : longitud=28
Aquí uela.
desvela
Un
Aquí ne pomgo a camtar
Al compás de la vigüela,
```

(continué en la próxima página)

(proviene de la página anterior)

Que el honbre que lo desvela
 Uma pema estraordimaria
 Cono la ave solitaria
 Com el camtar se comsuela.

3. Manejos de listas:

- Cree la lista **N** de longitud 50, donde cada elemento es un número entero de 1 a 50 inclusive (Ayuda: vea la expresión range).
- Invierta la lista.
- Extraiga una lista **N2** que contenga sólo los elementos pares de **N**.
- Extraiga una lista **N3** que contenga sólo aquellos elementos que sean el cuadrado de algún número entero.

4. Cree una lista de la forma $L = [1, 3, 5, \dots, 17, 19, 19, 17, \dots, 3, 1]$

5. Operación rara sobre una lista:

- Defina la lista $L = [0, 1]$
- Realice la operación $L.append(L)$
- Ahora imprima L , e imprima el último elemento de L .
- Haga que una nueva lista $L1$ que tenga el valor del último elemento de L y repita el inciso anterior.

6. Utilizando el string: `python s1='En un lugar de la Mancha de cuyo nombre no quiero acordarme'` y utilizando los métodos de strings:

- Obtenga la cantidad de caracteres.
- Imprima la frase anterior pero con cada palabra empezando en mayúsculas.
- Cuente cuantas letras a tiene la frase, ¿cuántas vocales tiene?
- Imprima el string $s1$ centrado en una línea de 80 caracteres, rodeado de guiones en la forma:

-----En un lugar de la Mancha de cuyo nombre no quiero
 acordarme-----

- Obtenga una lista **L1** donde cada elemento sea una palabra.
- Cuente la cantidad de palabras en $s1$ (utilizando python).
- Ordene la lista **L1** en orden alfabético.
- Ordene la lista **L1** tal que las palabras más cortas estén primero.
- Ordene la lista **L1** tal que las palabras más largas estén primero.
- Construya un string **s2** con la lista del resultado del punto anterior.
- Encuentre la palabra más larga y la más corta de la frase.

7. Escriba un script que encuentre las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$. Los valores de los parámetros defínalos en el mismo script, un poco más arriba.8. Considere un polígono regular de N lados inscripto en un círculo de radio unidad:

- Calcule el ángulo interior del polígono regular de N lados (por ejemplo el de un triángulo es 60 grados, de un cuadrado es 90 grados, y de un pentágono es 108 grados). Exprese el resultado en grados y en radianes para valores de $N = 3, 5, 6, 8, 9, 10, 12$.

- ¿Puede calcular la longitud del lado de los polígonos regulares si se encuentran inscriptos en un círculo de radio unidad?
9. Escriba un *script* (llamado *distancia1.py*) que defina las variables velocidad y posición inicial v_0 , z_0 , la aceleración g , y la masa $m = 1$ kg a tiempo $t = 0$, y calcule e imprima la posición y velocidad a un tiempo posterior t . Ejecute el programa para varios valores de posición y velocidad inicial para $t = 2$ segundos. Recuerde que las ecuaciones de movimiento con aceleración constante son:

$$v = v_0 - gt$$
$$z = z_0 + v_0t - gt^2/2.$$

16.2.1 Adicionales

11. Calcular la suma:

$$s_1 = \frac{1}{2} \left(\sum_{k=0}^{100} k \right)^{-1}$$

Ayuda: busque información sobre la función `sum()`

12. Construir una lista `L2` con 2000 elementos, todos iguales a `0.0005`. Imprimir su suma utilizando la función `sum` y comparar con la función que existe en el módulo `math` para realizar suma de números de punto flotante.

16.3 Ejercicios de Clase 03

1. Imprimir los números que no son divisibles por 2, 3, 5 o 7 de los primeros 100 números naturales
2. Calcule la suma

$$s_2 = \sum_{k=1}^{\infty} \frac{(-1)^k(k+1)}{2k^3+k^2}$$

con un error relativo estimado menor a $\epsilon = 10^{-5}$. Imprima por pantalla el resultado, el valor máximo de k computado y el error relativo estimado.

3. Cree dos listas: una con los números que no son múltiplos de ninguno de 2,7,11,13 y otra con los que no son múltiplos de 3,5,17. Considere los primeros 5000 números naturales. Cree una nueva lista donde combine las dos listas anteriores ordenada en forma creciente.

4. Realice un programa que:

- Lea el archivo `names.txt`
- Guarde en un nuevo archivo (llamado `pares.txt`) palabra por medio del archivo original (la primera, tercera,) una por línea, pero en el orden inverso al leído
- Agregue al final de dicho archivo, las palabras pares pero separadas por un punto y coma (;
- En un archivo llamado `longitudes.txt` guarde las palabras ordenadas por su longitud, y para cada longitud ordenadas alfabéticamente.
- En un archivo llamado `letras.txt` guarde sólo aquellas palabras que contienen las letras `w, x, y, z`, con el formato:
 - `w: Walter, .`
 - `x: Xilofón,`
 - `y: .`

- Z: .
 - Cree un diccionario, donde cada *key* es la primera letra y cada valor es una lista, cuyo elemento es una tuple (palabra, longitud). Por ejemplo:

```
d['a'] = [('Aaa', 3), ('Anna', 4), ...]
```

5. Las funciones de Bessel de orden n cumplen las relaciones de recurrencia

$$J_{n-1}(x) - \frac{2n}{x} J_n(x) + J_{n+1}(x) = 0$$

$$J_0^2(x) + \sum_{n=1}^{\infty} 2J_n^2(x) = 1$$

Para calcular la función de Bessel de orden N , se empieza con un valor de $M \gg N$, y utilizando los valores iniciales $J_M = 1$, $J_{M+1} = 0$ se utiliza la primera relación para calcular todos los valores de $n < M$. Luego, utilizando la segunda relación se normalizan todos los valores.

Nota: Estas relaciones son válidas si $M \gg x$ (use algún valor estimado, como por ejemplo $M = N + 20$).

Utilice estas relaciones para calcular $J_N(x)$ para $N = 3, 4, 7$ y $x = 2, 5, 5, 7, 10$. Para referencia se dan los valores esperados

$$\begin{aligned} J_3(2,5) &= 0,21660 \\ J_4(2,5) &= 0,07378 \\ J_7(2,5) &= 0,00078 \\ J_3(5,7) &= 0,20228 \\ J_4(5,7) &= 0,38659 \\ J_7(5,7) &= 0,10270 \\ J_3(10,0) &= 0,05838 \\ J_4(10,0) &= -0,21960 \\ J_7(10,0) &= 0,21671 \end{aligned}$$

6. Imprima por pantalla una tabla con valores equiespaciados de x entre 0 y 180, con valores de las funciones trigonométricas de la forma:

```
"""
=====
| x | sen(x) | cos(x) | tan(-x/4) |
=====
| 0 | 0.000 | 1.000 | -0.000 |
| 10 | 0.174 | 0.985 | -0.044 |
| 20 | 0.342 | 0.940 | -0.087 |
| 30 | 0.500 | 0.866 | -0.132 |
| 40 | 0.643 | 0.766 | -0.176 |
| 50 | 0.766 | 0.643 | -0.222 |
| 60 | 0.866 | 0.500 | -0.268 |
| 70 | 0.940 | 0.342 | -0.315 |
| 80 | 0.985 | 0.174 | -0.364 |
| 90 | 1.000 | 0.000 | -0.414 |
| 100 | 0.985 | -0.174 | -0.466 |
| 110 | 0.940 | -0.342 | -0.521 |
| 120 | 0.866 | -0.500 | -0.577 |
=====
```

(continué en la próxima página)

(proviene de la página anterior)

```

/130 | 0.766 | -0.643 | -0.637 |
/140 | 0.643 | -0.766 | -0.700 |
/150 | 0.500 | -0.866 | -0.767 |
/160 | 0.342 | -0.940 | -0.839 |
/170 | 0.174 | -0.985 | -0.916 |
=====
"""

```

7. Dada una lista de números, vamos a calcular valores relacionados a su estadística.

- Calcular los valores de la media aritmética, la media geométrica y la media armónica, dados por:

$$A(x_1, \dots, x_n) = \bar{x} = \frac{x_1 + \dots + x_n}{n}$$

$$G(x_1, \dots, x_n) = \sqrt[n]{x_1 \cdots x_n}$$

$$H(x_1, \dots, x_n) = \frac{n}{\frac{1}{x_1} + \dots + \frac{1}{x_n}}$$

- Calcular la desviación estándar:

$$\sigma \equiv \sqrt{\frac{1}{n} \sum_i (x_i - \bar{x})^2}$$

- Calcular la mediana, que se define como el valor para el cual la mitad de los valores de la lista es menor que ella. Si el número de elementos es par, se toma el promedio entre los dos adyacentes.

Realizar los cálculos para las listas de números:

16.4 Ejercicios de Clase 04

1. Realice un programa para:

- Leer los datos del archivo **aluminio.dat** y poner los datos del elemento en un diccionario de la forma:

```
d = { 'S': 'Al', 'Z':13, 'A':27, 'M': '26.98153863(12)', 'P': 1.0000, 'MS':26.
      ↪9815386(8) }
```

- Modifique el programa anterior para que las masas sean números (`float`) y descarte el valor de la incertezza (el número entre paréntesis)
- Agregue el código necesario para obtener una impresión de la forma:

```
Elemento: Al
Número Atómico: 13
Número de Masa: 27
Masa: 26.98154
```

Note que la masa sólo debe contener 5 números decimales

2. Escriba funciones para analizar la divisibilidad de enteros:

- La función `es_divisible1(x)` que retorna verdadero si `x` es divisible por alguno de `2, 3, 5, 7` o falso en caso contrario.

- La función `es_divisible_por_lista` que cumple la misma función que `es_divisible1` pero recibe dos argumentos: el entero `x` y una variable del tipo lista que contiene los valores para los cuáles debemos examinar la divisibilidad. Las siguientes expresiones deben retornar el mismo valor:

```
es_divisible1(x)
es_divisible_por_lista(x, [2, 3, 5, 7])
es_divisible_por_lista(x)
```

- La función `es_divisible_por` cuyo primer argumento (mandatorio) es `x`, y luego puede aceptar un número indeterminado de argumentos:

```
es_divisible_por(x) # retorna verdadero siempre
es_divisible_por(x, 2) # verdadero si x es par
es_divisible_por(x, 2, 3, 5, 7) # igual resultado que es_divisible1(x) e igual a ↵
                                ↵es_divisible_por_lista(x)
es_divisible_por(x, 2, 3, 5, 7, 9, 11, 13) # o cualquier secuencia de argumentos ↵
                                ↵debe funcionar
```

16.5 Ejercicios de Clase 05

1. Escriba una función `crear_sen(A, w)` que acepte dos números reales `A, w` como argumentos y devuelva la función $f(x)$.

Al evaluar la función f en un dado valor x debe dar el resultado: $f(x) = A \sin(wx)$ tal que se pueda utilizar de la siguiente manera:

```
f = crear_sen(3, 1.5)
f(2)           # Debería imprimir el resultado de 3*sin(1.5*2)=0.4233600241796016
```

2. Utilizando conjuntos (`set`), escriba una función que compruebe si un string contiene todas las vocales. La función debe devolver `True` o `False`.
3. Escriba una serie de funciones que permitan trabajar con polinomios. Vamos a representar a un polinomio como una lista de números reales, donde cada elemento corresponde a un coeficiente que acompaña una potencia
 - Una función que devuelva el orden del polinomio (un número entero)
 - Una función que sume dos polinomios y devuelva un polinomio (objeto del mismo tipo)
 - Una función que multiplique dos polinomios y devuelva el resultado en otro polinomio
 - Una función devuelva la derivada del polinomio (otro polinomio).
 - Una función que, acepte el polinomio y devuelva la función correspondiente.
4. Vamos a describir un **sudoku** como un array bidimensional de 9×9 números, cada uno de ellos entre 1 y 4. Escribir una función que tome como argumento una grilla (Lista bidimensional de 9×9) y devuelva verdadero si los números corresponden a la resolución correcta y falso en caso contrario. Recordamos que para que sea válido debe cumplirse que:
 - Los números están entre 1 y 9
 - En cada fila no deben repetirse
 - En cada columna no deben repetirse
 - En todas las regiones de 3×3 que no se solapan, empezando de cualquier esquina, no deben repetirse.

16.6 Ejercicios de Clase 06

1. Implemente los métodos `suma`, `producto` y `abs`

- `suma()` debe retornar un objeto del tipo `Vector` y contener en cada componente la suma de las componentes de los dos vectores que toma como argumento.
- `producto` toma como argumentos dos vectores y retorna un número real
- `abs` toma como argumentos el propio objeto y retorna un número real

Su uso será el siguiente:

```
v1 = Vector(1,2,3)
v2 = Vector(3,2,1)
v = v1.suma(v2)
pr = v1.producto(v2)
a = v1.abs()
```

2. Utilizando la definición de la clase `Punto`:

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

    def __del__(self):
        "Borra el punto y actualiza el contador"
        Punto.num_puntos -= 1

    def __str__(self):
        s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
        return s

    def __repr__(self):
        return "Punto(x={}, y={}, z={})".format(self.x, self.y, self.z)

    def __call__(self):
        return self.__str__()

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print("En total hay {} puntos definidos".format(cls.num_puntos))
```

Complete la implementación de la clase `Vector` con los métodos pedidos

```
class Vector(Punto):
    "Representa un vector en el espacio"
```

(continué en la próxima página)

(proviene de la página anterior)

```

def suma(self, v2):
    "Calcula un vector que contiene la suma de dos vectores"
    print("Aún no implementada la suma de dos vectores")
    # código calculando v = suma de self + v2
    # ...

def producto(self, v2):
    "Calcula el producto interno entre dos vectores"
    print("Aún no implementado el producto interno de dos vectores")
    # código calculando el producto interno pr = v1 . v2

def abs(self):
    "Devuelve la distancia del punto al origen"
    print("Aún no implementado la norma del vector")
    # código calculando la magnitud del vector

def angulo_entre_vectores(self, v2):
    "Calcula el ángulo entre dos vectores"
    print("Aún no implementado el ángulo entre dos vectores")
    angulo = 0
    # código calculando angulo = arccos(v1 * v2 / (|v1| * |v2|))
    return angulo

def coordenadas_cilindricas(self):
    "Devuelve las coordenadas cilíndricas del vector como una tupla (r, theta, z)"
    print("No implementada")

def coordenadas_esfericas(self):
    "Devuelve las coordenadas esféricas del vector como una tupla (r, theta, phi)"
    print("No implementada")

```

3. **PARA ENTREGAR:** Cree una clase **Polinomio** para representar polinomios. La clase debe guardar los datos representando todos los coeficientes. El grado del polinomio será *menor o igual a 9* (un dígito).

Nota: Utilice el archivo **polinomio_06.py** en el directorio **data**, que renombrará de la forma usual *Apellido_06.py*. Se le pide que programe:

- Un método de inicialización **__init__** que acepte una lista de coeficientes. Por ejemplo para el polinomio $4x^3 + 3x^2 + 2x + 1$ usaríamos:

```
>>> p = Polinomio([1, 2, 3, 4])
```

- Un método **grado** que devuelva el orden del polinomio

```
>>> p = Polinomio([1, 2, 3, 4])
>>> p.grado()
3
```

- Un método **get_coeficientes**, que devuelva una lista con los coeficientes:

```
>>> p.get_coeficientes()
[1, 2, 3, 4]
```

- Un método **set_coeficientes**, que fije los coeficientes de la lista:

```
>>> p1 = Polinomio()
>>> p1.set_coeficientes([1, 2, 3, 4])
>>> p1.get_coeficientes()
[1, 2, 3, 4]
```

- El método `suma_pol` que le sume otro polinomio y devuelva un polinomio (objeto del mismo tipo)
- El método `mul` que multiplica al polinomio por una constante y devuelve un nuevo polinomio
- Un método, `derivada`, que devuelva la derivada de orden n del polinomio (otro polinomio):

```
>>> p1 = p.derivada()
>>> p1.get_coeficientes()
[2, 6, 12]
>>> p2 = p.derivada(n=2)
>>> p2.get_coeficientes()
[6, 24]
```

- Un método que devuelva la integral (antiderivada) del polinomio de orden n , con constante de integración `cte` (otro polinomio).

```
>>> p1 = p.integrada()
>>> p1.get_coeficientes()
[0, 1, 1, 1, 1]
>>>
>>> p2 = p.integrada(cte=2)
>>> p2.get_coeficientes()
[2, 1, 1, 1, 1]
>>>
>>> p3 = p.integrada(n=3, cte=1.5)
>>> p3.get_coeficientes()
[1.5, 1.5, 0.75, 0.1666666666666666, 0.0833333333333333, 0.05]
```

- El método `eval`, que evalúe el polinomio en un dado valor de x .

```
>>> p = Polinomio([1, 2, 3, 4])
>>> p.eval(x=2)
49
>>>
>>> p.eval(x=0.5)
3.25
```

- **(Si puede)** Un método `from_string` que crea un polinomio desde un string en la forma:

```
>>> p = Polinomio()
>>> p.from_string('x^5 + 3x^3 - 2 x+x^2 + 3 - x')
>>> p.get_coeficientes()
[3, -3, 1, 3, 0, 1]
>>>
>>> p1 = Polinomio()
>>> p1.from_string('y^5 + 3y^3 - 2 y + y^2+3', var='y')
>>> p1.get_coeficientes()
[3, -2, 1, 3, 0, 1]
```

- Escriba un método llamado `__str__`, que devuelva un string (que define cómo se va a imprimir el polinomio). Un ejemplo de salida será:

```
>>> p = Polinomio([1, 2.1, 3, 4])
>>> print(p)
4 x^3 + 3 x^2 + 2.1 x + 1
```

- Escriba un método llamado `__call__`, de manera tal que al llamar al objeto, evalúe el polinomio (use el método `eval` definido anteriormente)

```
>>> p = Polinomio([1, 2, 3, 4])
>>> p(x=2)
49
>>>
>>> p(0.5)
3.25
```

- Escriba un método llamado `__add__(self, p)`, que evalúe la suma de polinomios usando el método `suma_pol` definido anteriormente. Eso permitirá usar la operación de suma en la forma:

```
>>> p1 = Polinomio([1, 2, 3, 4])
>>> p2 = Polinomio([1, 2, 3, 4])
>>> p1 + p2
```

16.7 Ejercicios de Clase 08

- Genere arrays en 2d, cada uno de tamaño 10x10 con:
 - Un array con valores 1 en la diagonal principal y 0 en el resto (Matriz identidad).
 - Un array con valores 0 en la diagonal principal y 1 en el resto.
 - Un array con valores 1 en los bordes y 0 en el interior.
 - Un array con números enteros consecutivos (empezando en 1) en los bordes y 0 en el interior.
- Diga qué resultado produce el siguiente código, y explíquelo

```
# Ejemplo propuesto por Jake VanderPlas
print(sum(range(5), -1))
from numpy import *
print(sum(range(5), -1))
```

- Escriba una función `suma_potencias(p, n)` (utilizando arrays y **Numpy**) que calcule la operación

$$s_2 = \sum_{k=0}^n k^p$$

- Usando las funciones de `numpy sign` y `maximum` definir las siguientes funciones, que acepten como argumento un array y devuelvan un array con el mismo *shape*:
 - función de Heaviside, que vale 1 para valores positivos de su argumento y 0 para valores negativos.
 - La función escalón, que vale 0 para valores del argumento fuera del intervalo $(-1, 1)$ y 1 para argumentos en el intervalo.
 - La función rampa, que vale 0 para valores negativos de x y x para valores positivos.

5. PARA ENTREGAR. Caída libre Cree un programa que calcule la posición y velocidad de una partícula en caída libre para condiciones iniciales dadas (h_0, v_0), y un valor de gravedad dados. Se utilizará la convención de que alturas y velocidades positivas corresponden a vectores apuntando hacia arriba (una velocidad positiva significa que la partícula se aleja de la tierra).

El programa debe realizar el cálculo de la velocidad y altura para un conjunto de tiempos equiespaciados. El usuario debe poder decidir o modificar el comportamiento del programa mediante opciones por línea de comandos.

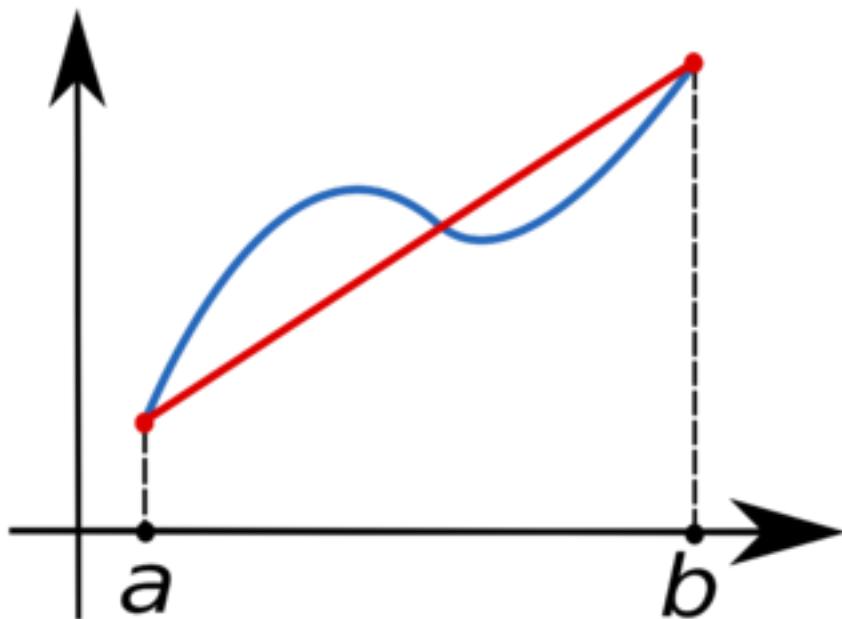
El programa debe aceptar las siguientes opciones por líneas de comando: `-v vel 0`, equivalentemente `--velocidad=vel`, donde `vel` es el número dando la velocidad inicial en m/s. El valor por defecto será 0. `-h alt 0`, equivalentemente `--altura=alt`, donde `alt` es un número dando la altura inicial en metros. El valor por defecto será 1000. La altura inicial debe ser un número positivo. `-g grav`, donde `grav` es el módulo del valor de la aceleración de la gravedad en m/s^2 . El valor por defecto será 9.8. `-o nombre 0`, equivalentemente `--output=nombre`, donde `nombre` será el nombre de un archivo donde se escribirán los resultados. Si el usuario no usa esta opción, debe imprimir por pantalla (`sys.stdout`). `-n N 0`, equivalentemente `--Ndatos=N`, donde `N` es un número entero indicando la cantidad de datos que deben calcularse. Valor por defecto: 100. `--ti=instante_inicial` indica el tiempo inicial de cálculo. Valor por defecto: 0. No puede ser mayor que el tiempo de llegada a la posición $h = 0$. `--tf=tiempo_final` indica el tiempo final de cálculo. Valor por defecto será el correspondiente al tiempo de llegada a la posición $h = 0$.

NOTA: Envíe el programa llamado **Suapellido_08.py** en un adjunto por correo electrónico, con asunto: **Suapellido_08**, antes del día lunes 9 de Marzo.

6. Queremos realizar numéricamente la integral

$$\int_a^b f(x)dx$$

utilizando el método de los trapecios. Para eso partimos el intervalo $[a, b]$ en N subintervalos y aproximamos la curva en cada subintervalo por una recta



La línea azul representa la función $f(x)$ y la línea roja la interpolación por una recta (figura de https://en.wikipedia.org/wiki/Trapezoidal_rule)

Si llamamos x_i ($i = 0, \dots, n$, con $x_0 = a$ y $x_n = b$) los puntos equiespaciados, entonces queda

$$\int_a^b f(x)dx \approx \frac{h}{2} \sum_{i=1}^n (f(x_i) + f(x_{i-1})).$$

- Escriba una función `trapz(x, y)` que reciba dos arrays unidimensionales `x` e `y` y aplique la fórmula de los trapecios.
- Escriba una función `trapzf(f, a, b, npts=100)` que recibe una función `f`, los límites `a`, `b` y el número de puntos a utilizar `npts`, y devuelve el valor de la integral por trapecios.
- Calcule la integral logarítmica de Euler:

$$\text{Li}(t) = \int_2^t \frac{1}{\ln x} dx$$

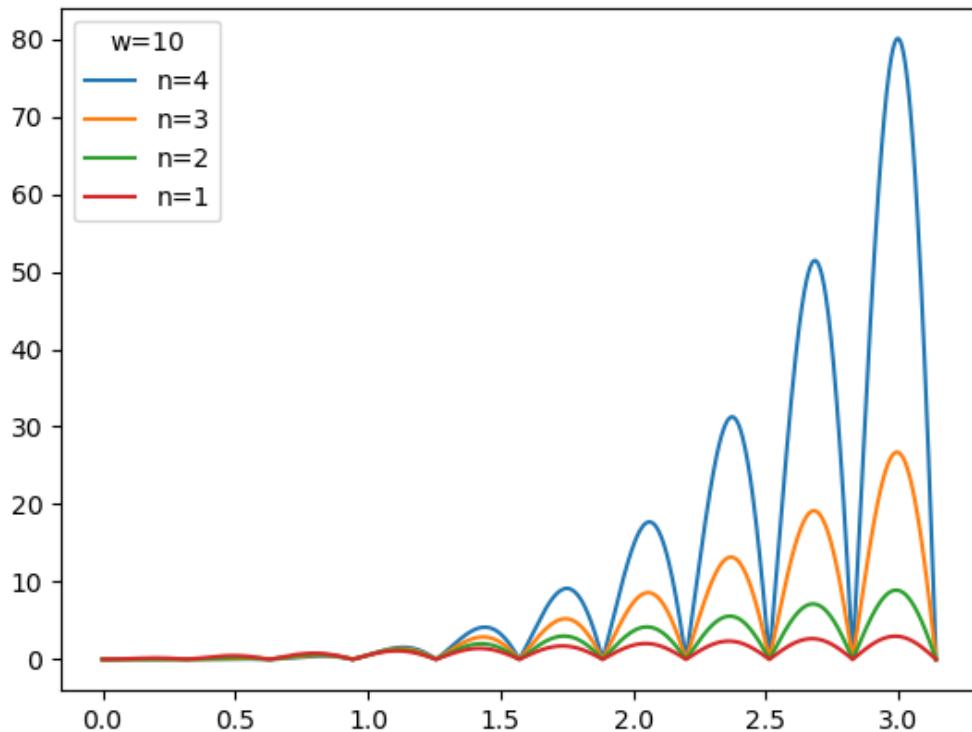
usando la función `trapzft` para valores de `npts=10, 20, 30, 40, 50, 60`

16.8 Ejercicios de Clase 09

1. Realizar un programa para visualizar la función

$$f(x, n, w) = x^n |\sin(wx)|$$

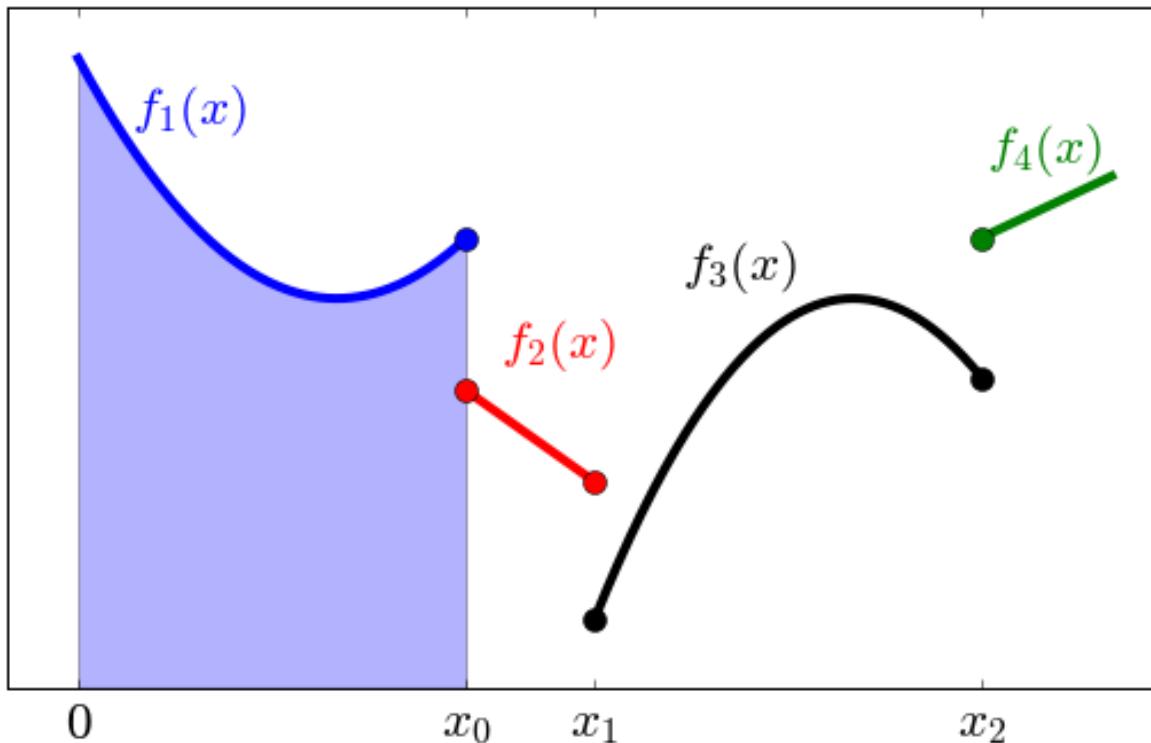
El programa debe realizar el gráfico para $w = 10$, con cuatro curvas para $n = 1, 2, 3, 4$, similar al que se muestra en la siguiente figura



2. Para la función definida a trozos:

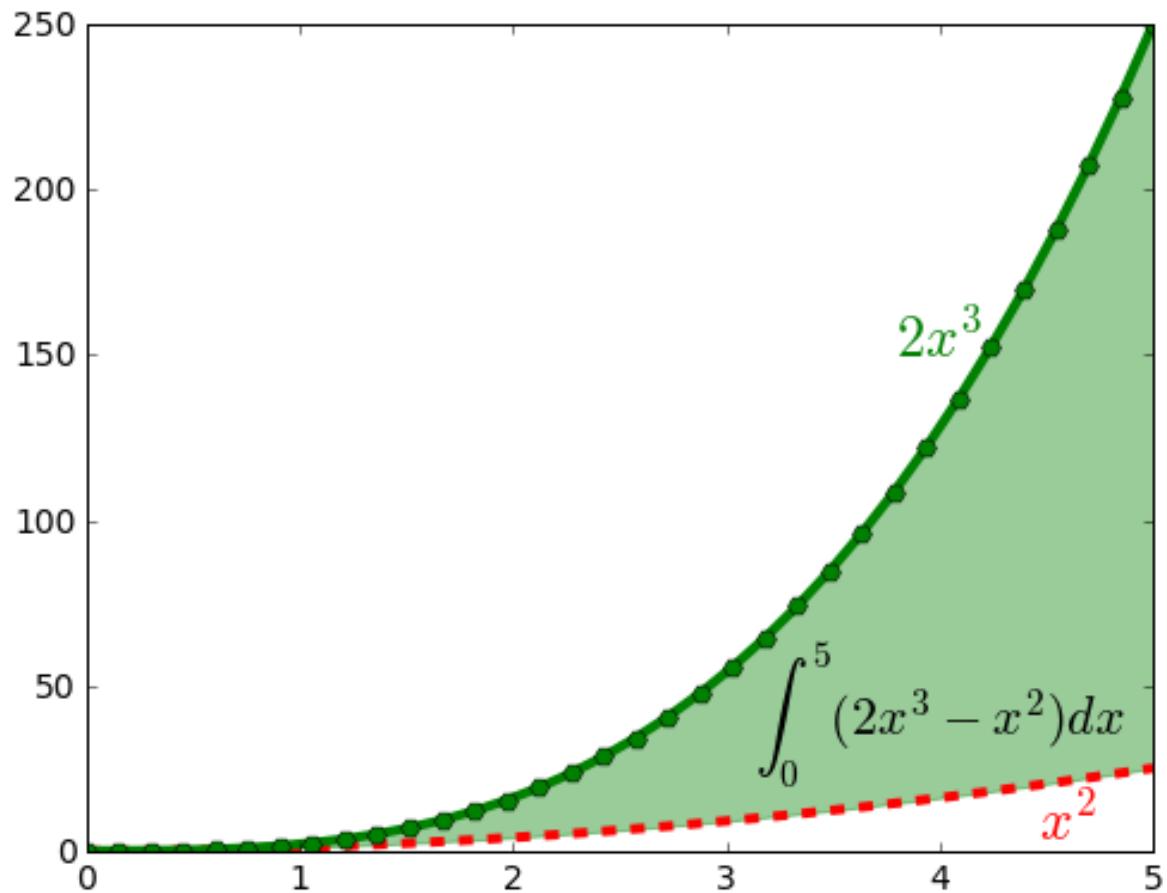
$$f(x) = \begin{cases} f_1(x) = x^2/8 & -\pi < x \leq \pi/2 \\ f_2(x) = -0,3x & \pi/2 < x < \pi \\ f_3(x) = -(x - 2\pi)^2/6 & \pi \leq x \leq 5\pi/2 \\ f_4(x) = (x - 2\pi)/5 & 5\pi/2 < x \leq 3\pi \end{cases}$$

realizar la siguiente figura de la manera más fiel posible.

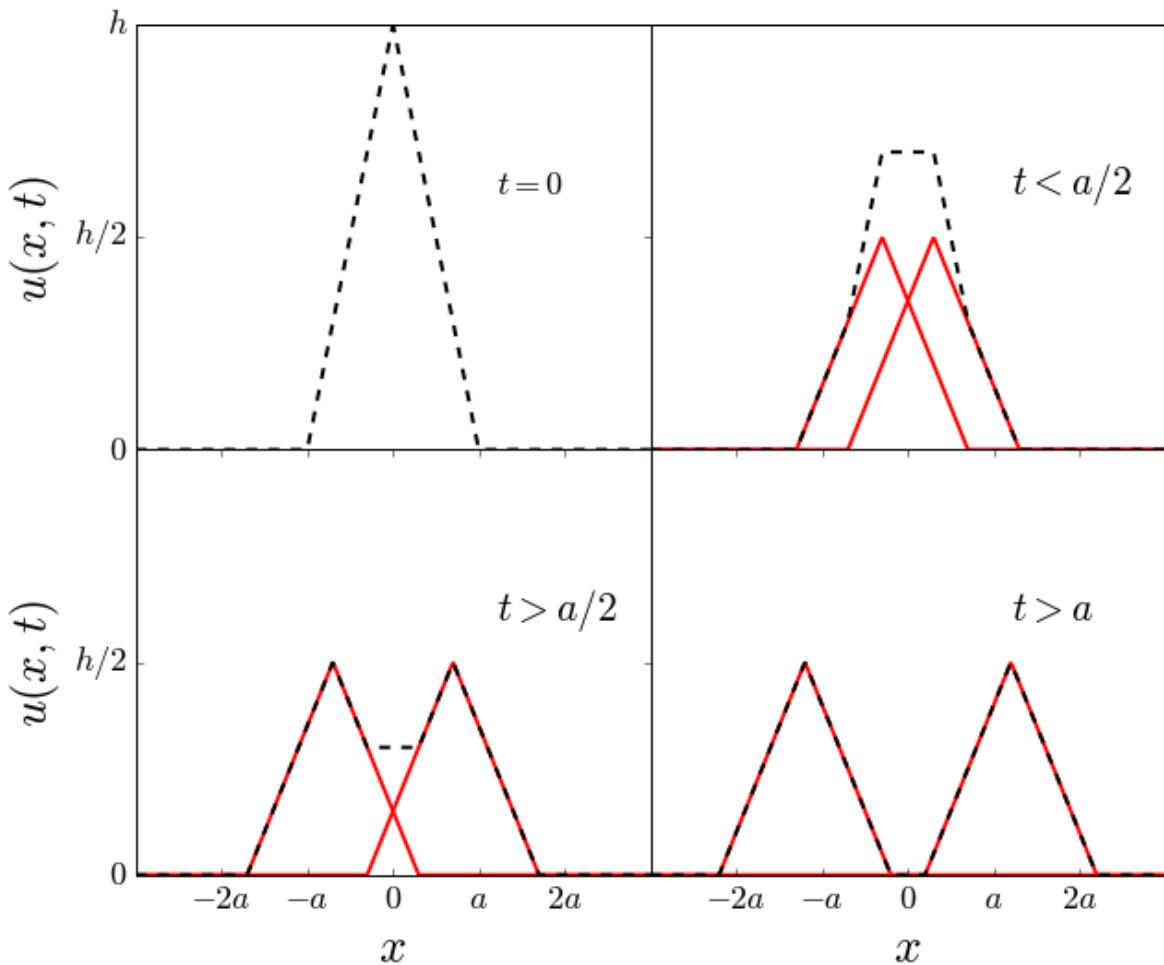


Pistas: Buscar información sobre `plt.fill_between()` y sobre `plt.xticks` y `plt.yticks`.

3. Rehacer la siguiente figura:



4. Notando que la curva en color negro corresponde a la suma de las dos curvas en rojo, rehacer la siguiente figura:



5. Crear una hoja de estilo que permita hacer gráficos adecuados para posters y presentaciones. Debe modificar los tamaños para hacerlos legibles a mayores distancias (sugerencia 16pt). El tamaño de la letra de los nombres de ejes y en las leyendas debe ser mayor también. Las líneas deben ser más gruesas (sugerencia: ~4), los símbolos de mayor tamaño (sugerencia ~10).

16.9 Ejercicios de Clase 10

- Dado un array a de números, creado por ejemplo usando:

```
a = np.random.uniform(size=100)
```

Encontrar el número más cercano a un número escalar dado (por ejemplo $x=0.5$). Utilice los métodos discutidos.

16.10 Ejercicios de Clase 11

- En el archivo `palabras.words.gz` hay una larga lista de palabras, en formato comprimido. Siguiendo la idea del ejemplo dado en clases realizar un histograma de las longitudes de las palabras.
- Modificar el programa del ejemplo de la clase para calcular el histograma de frecuencia de letras en las palabras (no sólo la primera). Considere el caso insensible a la capitalización: las mayúsculas y minúsculas corresponden

a la misma letra (á es lo mismo que Á y ambas corresponden a a).

3. Utilizando el mismo archivo de palabras, Guardar todas las palabras en un array y obtener los índices de las palabras que tienen una dada letra (por ejemplo la letra j), los índices de las palabras con un número dado de letras (por ejemplo 5 letras), y los índices de las palabras cuya tercera letra es una vocal. En cada caso, dar luego las palabras que cumplen dichas condiciones.
4. En el archivo `colision.npy` hay una gran cantidad de datos que corresponden al resultado de una simulación. Los datos están organizados en trece columnas. La primera corresponde a un parámetro, mientras que las 12 restantes corresponde a cada una de las tres componentes de la velocidad de cuatro partículas. Calcular y graficar:
5. la distribución de ocurrencias del primer parámetro.
6. la distribución de ocurrencias de energías de la tercera partícula.
7. la distribución de ocurrencias de ángulos de la cuarta partícula, medido respecto al tercer eje.
8. la distribución de energías de la tercera partícula cuando la cuarta partícula tiene un ángulo menor a 90 grados con el tercer eje.

Realizar los cuatro gráficos utilizando un formato adecuado para presentación (charla o poster).

5. Leer el archivo `colision.npy` y guardar los datos en formato texto con un encabezado adecuado. Usando el comando mágico `%timeit` o el módulo `timeit`, comparar el tiempo que tarda en leer los datos e imprimir el último valor utilizando el formato de texto y el formato original `npy`. Comparar el tamaño de los dos archivos.
6. El submódulo `scipy.constants` tiene valores de constantes físicas de interés. Usando este módulo compute la constante de Stefan-Boltzmann σ utilizando la relación:

$$\sigma = \frac{2\pi^5 k_B^4}{15 h^3 c^2}$$

Confirme que el valor obtenido es correcto comparando con la constante para esta cantidad en `scipy.constants`

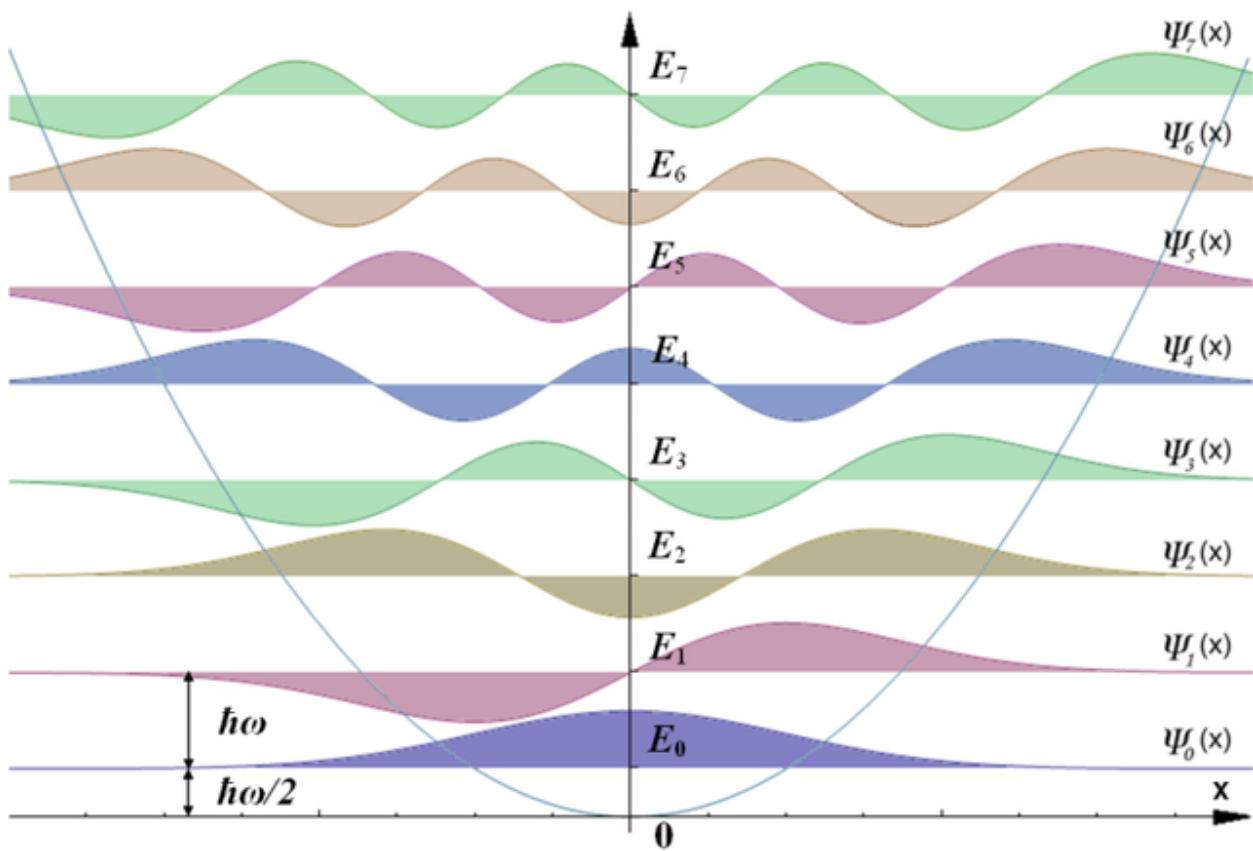
7. Usando **Scipy** y **Matplotlib** grafique las funciones de onda del oscilador armónico unidimensional para las cuatro energías más bajas ($n = 1, 2, 3, 4$), en el intervalo $[-5, 5]$. Asegúrese de que están correctamente normalizados.

Las funciones están dadas por:

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \cdot \left(\frac{\omega}{\pi}\right)^{1/4} \cdot e^{-\frac{\omega x^2}{2}} \cdot H_n(\sqrt{\omega}x), \quad n = 0, 1, 2, \dots$$

donde H_n son los polinomios de Hermite, y usando $\omega = 2$.

Trate de obtener un gráfico similar al siguiente (tomado de wikipedia. Realizado por By AllenMcC. - File: HarmOsziFunktionen.jpg, CC BY-SA 3.0)



16.11 Ejercicios de Clase 13

- En el archivo **co_nrg.dat** se encuentran los datos de la posición de los máximos de un espectro de CO₂ como función del número cuántico rotacional J (entero). Haga un programa que lea los datos. Los ajuste con polinomios (elija el orden adecuado) y grafique luego los datos (con símbolos) y el ajuste con una línea sólida roja. Además, debe mostrar los parámetros obtenidos para el polinomio.
- Queremos hacer un programa que permita fittear una curva como suma de N funciones gaussianas:
 - Haga una función, que debe tomar como argumento los arrays con los datos: x , y , y valores iniciales para las Gaussianas: `fit_gaussianas(x, y, *params)` donde `params` son los $3N$ coeficientes (tres coeficientes para cada Gaussiana). Debe devolver los parámetros óptimos obtenidos.
 - Realice un programita que grafique los datos dados y la función que resulta de sumar las gaussianas en una misma figura.
 - Si puede* agregue líneas o flechas indicando la posición del máximo y el ancho de cada una de las Gaussianas obtenidas.

16.12 Ejercicios de Clase 14

- Modificar el ejemplo anterior para presentar en una figura tres gráficos, agregando a la izquierda un panel donde se muestre un corte horizontal. El corte debe estar en la mitad del gráfico ($y_0 = 136$). En la figura debe mostrar la posición del corte (similarmente a como se hizo con el corte en x) con una línea de otro color.

2. Modificar el ejemplo anterior (**analizar_figura_2.py**) para presentar tres gráficos, agregando a la izquierda un panel donde se muestre el corte horizontal de la misma manera que en el ejercicio anterior. Al seleccionar con el *mouse* debe mostrar los dos cortes (horizontal y vertical).

CAPÍTULO 17

Material extra

17.1 Programa detallado

Autor Juan Fiol

Version \$Revision: 1\$

Copyright Libre

17.1.1 Clase 1: *Introducción, Instalación y ejemplos básicos de Python*

- Introducción a la materia, objetivos, metodología
- Introducción al lenguaje de programación
- Instalación y preparación de entorno
- Distintos usos de Python (Terminales, notebook, scripts)
- Comandos de Ipython
- Documentación interna y externa
- Ejemplos
- Ejercicios

17.1.2 Clase 2: *Conceptos básicos: Tipos*

- Biblioteca Standard y paquetes
- Tipos de variables, asignaciones, operaciones y métodos:
 - Variables numéricas
 - Variables lógicas

- Variables de texto: strings
- Introducción a la entrada y salida de datos
- Ejercicios

17.1.3 Clase 3: *Conceptos básicos: Tipos y Control de flujo*

- Tipos de variables, asignaciones, operaciones y métodos:
 - Más operaciones con strings
 - Listas y tuples
 - Diccionarios
 - Conjuntos
- Introducción a control de flujo
- Ejercicios

17.1.4 Clase 4: *Control de flujo, funciones*

- **Más información sobre tipos y control de flujo**
 - Loops e iteraciones sobre listas
 - Loops e iteraciones sobre diccionarios
- **Funciones**
 - Definición de funciones
 - Funciones con argumentos mandatorios
 - Número variable de argumentos
- Ejercicios

17.1.5 Clase 5: *Módulos y funciones*

- Detalles sobre argumentos de funciones
- Uso de módulos
- Manejo de archivos y estructura de directorios
- Entrada y salida de datos, archivos comprimidos
- Ejercicios

17.1.6 Clase 6 *Breve introducción a la programación orientada a objetos en Python*

- Definición y características
- Objetos y clases: diseños y encapsulamiento
- Herencia
- Desarrollo de un ejemplo

- Ejercicios

17.1.7 Clase 7: Paquetes científicos: Introducción a Numpy y Matplotlib

- Introducción a Numpy
- Variables del tipo arreglo (*arrays*)
 - Creación manual y mediante funciones
 - Tipos de datos
 - Arrays multidimensionales
- Cómo seleccionar partes de arreglos: *indexing* y *Slicing*
- Algunas operaciones y métodos sobre arrays
- Ejercicios

17.1.8 Clase 8: Introducción a visualización: *Matplotlib*

- Introducción a Matplotlib
- Creación de gráficos simples
- Títulos y leyendas
- Textos y anotaciones
- Otros tipos de gráficos
- Documentación y ayuda
- Ejercicios

17.1.9 Clase 9: Manipulación de vectores

- Operaciones entre vectores, concatenación de vectores
- Uso de condicionales y arrays lógicos
- Métodos de arrays, funciones y manipulación de arrays
- Cambio de forma de arrays multidimensionales
- Ejercicios

17.1.10 Clase 10: Entrada y salida de vectores

- Ejemplo: Creación de histogramas y visualización
- Aplicación de funciones a vectores
- Lectura y escritura de tablas a archivos
 - Datos en formato texto
 - Datos en formatos numpy
 - Datos en otros formatos

- Ejercicios

17.1.11 Clase 11: *Introducción al paquete científico Scipy*

- **Introducción rápida a Scipy**

- Funciones especiales
- Integración numérica
- Minimización y optimización
- Álgebra lineal

- Documentación y ayuda

- Ejercicios

17.1.12 Clase 12: *Interpolación y ajuste de curvas (fiteo)*

- Interpolación de datos

- Ajuste de datos por cuadrados mínimos con polinomios

- Ajuste de curvas con funciones arbitrarias

- Fiteos utilizando el paquete científico Scipy

- Ejemplo: Fiteo de picos

- Documentación y ayuda

- Ejercicios

17.1.13 Clase 13: *Ejemplo de aplicación a casos (cuasi) reales*

- Ejemplo de procesamiento de datos

- Tratamiento de línea de base
- Búsqueda de picos
- Fiteo de picos
- Cálculo de máximos y áreas

- Ejercicios

17.1.14 Clase 14: *Ejemplos en más dimensiones*

- Integración numérica en 2D

- Fiteos en el plano

- Graficación en dos dimensiones

- Ejercicios

17.1.15 Clase 15: *Transformadas de Fourier*

- Introducción a transformadas de Fourier
- Transformada rápida de Fourier (FFT)
- Transformadas de Fourier en 2D, imágenes
- Ejercicios

17.1.16 Clase 16: *Introducción breve a otras librerías científicas*

- Manejo de gran número de datos: Pandas
- Matemática simbólica: Sympy

17.2 Apéndice: Método de las agujas

17.2.1 Lineamiento del método

En el problema de las agujas se describe un método para estimar el valor de π .

Se sabe que, si se tiran agujas al azar sobre una superficie con rayas, la probabilidad de que una aguja cruce una línea será:

$$P = \frac{2\ell}{t\pi}$$

donde ℓ es la longitud de la aguja y t es la separación entre líneas.

Desde este resultado podemos estimar el valor de π como:

$$\pi = \frac{2\ell}{tP}$$

ya que conocemos t , ℓ .

Sólo necesitamos estimar P . Para ello vamos a simular que tiramos un gran número de agujas sobre una superficie con líneas paralelas y vamos a calcular la probabilidad de que una aguja cruce alguna línea como

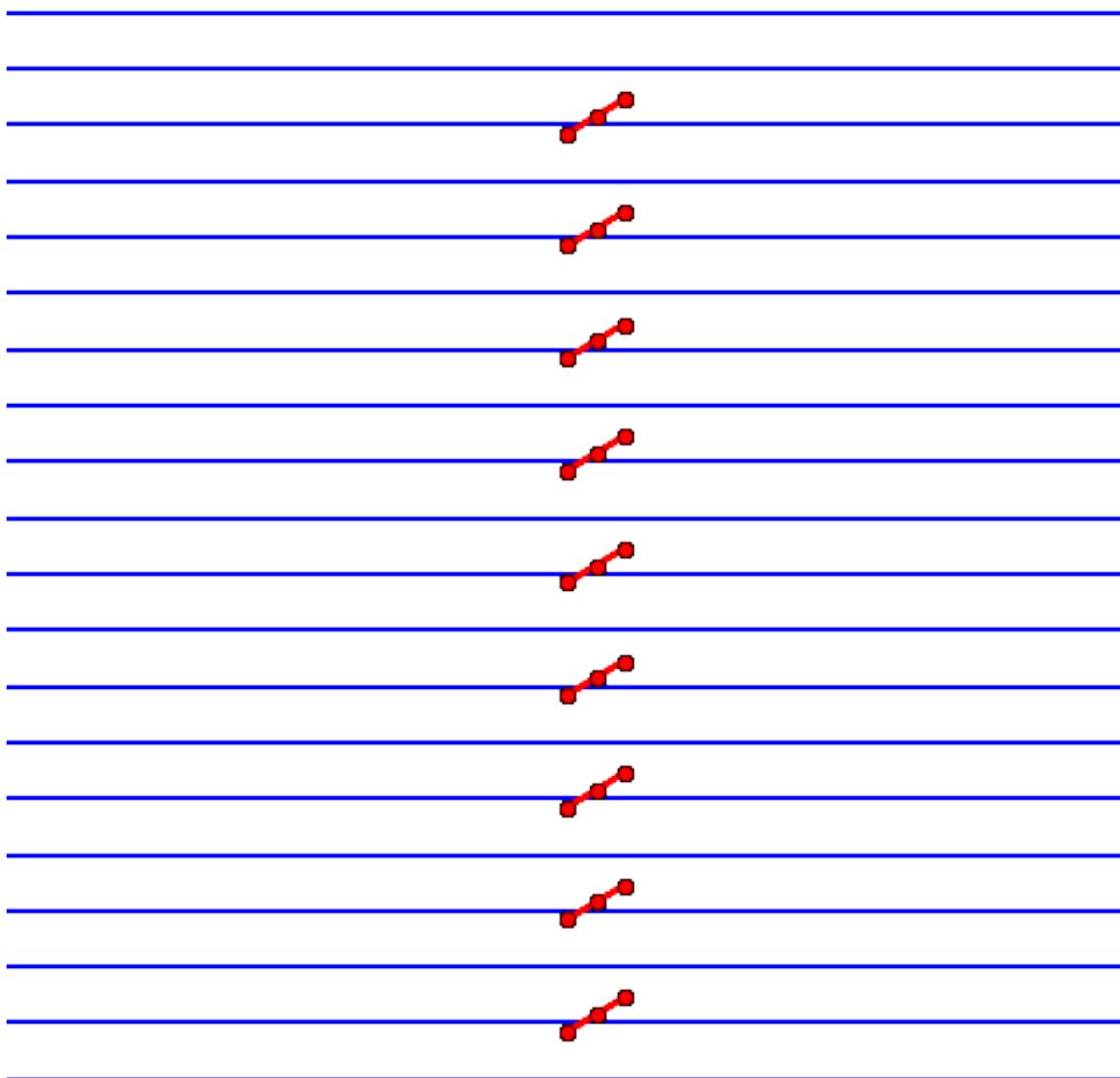
$$P = \frac{\text{Número de agujas que cruzan alguna línea}}{\text{Número total de agujas arrojadas}}$$

y reemplazando este resultado en la ecuación anterior obtenemos el valor estimado de π .

Para realizar el problema, vamos a modelar cada aguja con sólo la información necesaria. Por ejemplo, podemos usar un par de números (x, y) especificando un punto en el plano como el centro de la aguja. Con este punto y el ángulo que forma la aguja con la recta horizontal tenemos todos los datos para describirla.

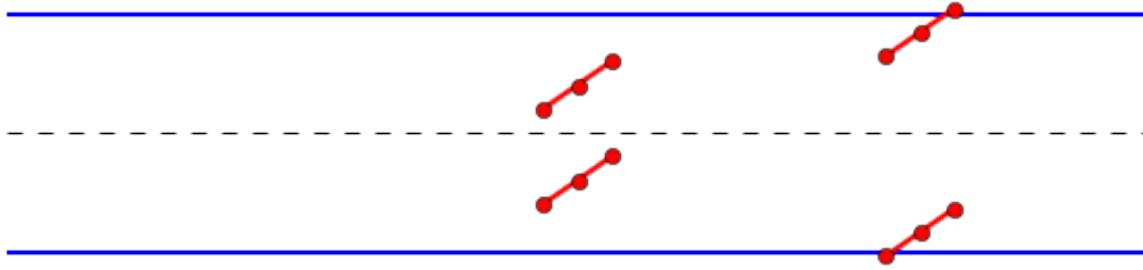
17.2.2 Simplificación del problema

Consideremos ahora el problema en que arrojamos una aguja y cayó en algunos de los lugares marcados en la figura (la aguja está ilustrada como la línea en rojo, con sus dos extremos y el centro señalados como círculos). Es evidente que cualquiera de los lugares marcados es equivalente a cualquier otro.



Consideremos la primera aguja desde arriba (en la tercera línea). Si estamos modelando la aguja como el punto del centro y el ángulo, entonces esta aguja cayó en el segundo espacio entre líneas. Si la aguja hubiera caído a cualquier distancia $\Delta y = nt$ de la posición actual, el resultado hubiera sido que en lugar de tocar la tercera línea, hubiera tocado otra. El cálculo de la probabilidad (y por consiguiente de π) no depende de qué línea toca.

Teniendo esto en cuenta podemos considerar sólo uno de los espacios entre líneas, y tirar todas las agujas entre un par de líneas como se muestra en la siguiente figura. Notar que el problema de muchas líneas, considerado en la figura anterior es simplemente la repetición de este problema muchas veces y no aporta nada nuevo.



Entonces, el problema se ha reducido a elegir un par (x, y) entre las dos líneas (inferior y superior) que señala la posición del centro de nuestra aguja, luego debemos elegir un ángulo, y con estos datos fijarnos si la aguja cruza una de las líneas.

Como simplificación adicional notemos que el problema es simétrico respecto del centro del espacio (línea punteada en la figura) y cualquiera de las dos agujas de la zona superior es equivalente a su contraparte de la zona inferior. Entonces podemos simplificar el problema un paso más y considerar sólo la mitad del espacio entre líneas.

Finalmente, notemos que la posición en el eje horizontal no cambia si la aguja cruza o no la línea y por lo tanto no juega ningún papel en el cálculo de la probabilidad.

17.2.3 Método de resolución

En resumen, una posible resolución requiere los siguientes pasos:

1. Elegir al azar la posición del centro (sólo la coordenada y) entre valores 0 y $t/2$.
2. Elegir al azar el ángulo que forma la aguja con la recta horizontal.
3. Calcular si la aguja atraviesa la línea localizada en $y = 0$.

CAPÍTULO 18

Material adicional

- Puede descargar las Clases en formato pdf