
Clases de Python

Juan Fiol

2022

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: Tipos complejos y control de flujo	53
5. Clase 4: Funciones	77
6. Clase 5: Entrada y salida, decoradores, y errores	99
7. Clase 6: Programación Orientada a Objetos	113
8. Clase 7: Control de versiones y biblioteca standard	131
9. Clase 8: Introducción a Numpy	151
10. Clase 9: Visualización	175
11. Clase 10: Más información sobre Numpy	211
12. Clase 11: Introducción al paquete Scipy	253
13. Clase 12: Un poco de graficación 3D	283
14. Clase 13: Interpolación y ajuste de curvas (fiteo)	291
15. Clase 14: Animaciones e interactividad	335
16. Clase 15: Interfaces con otros lenguajes	357
17. Clase 16: Programación funcional con Python	383
18. Ejercicios	391
19. Material extra	417
20. Material adicional	425

Institución Instituto Balseiro - Univ. Nac. de Cuyo

Fecha Febrero a abril de 2022

Docentes Juan Fiol y Flavio Colavecchia

CAPÍTULO 1

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

Autor: Juan Fiol

Licencia:

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



1.1 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++),

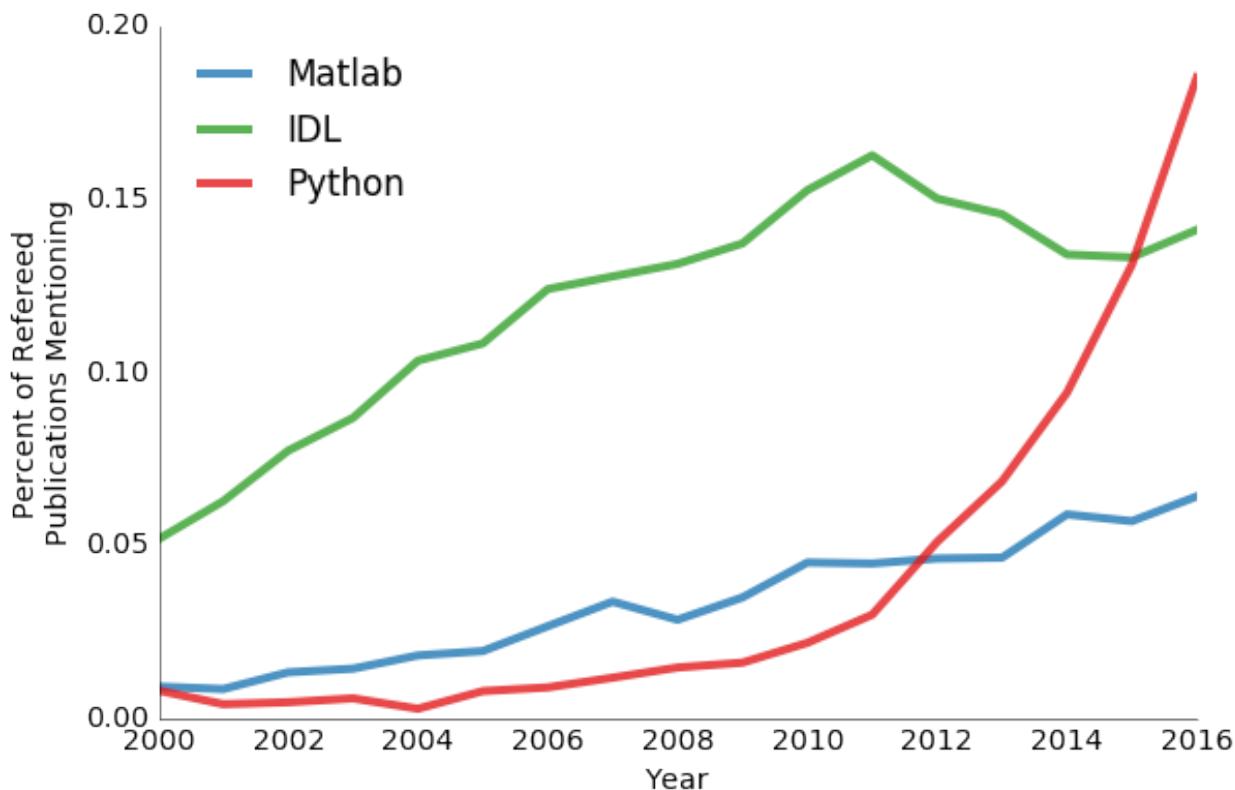
e incluso se puede usar como “marco de trabajo”, para crear una interfaz consistente y simple de usar en un conjunto de programas ya existentes.

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.

El uso de Python en comparación con otros lenguajes científicos de alto nivel creció rápidamente durante los últimos diez años.

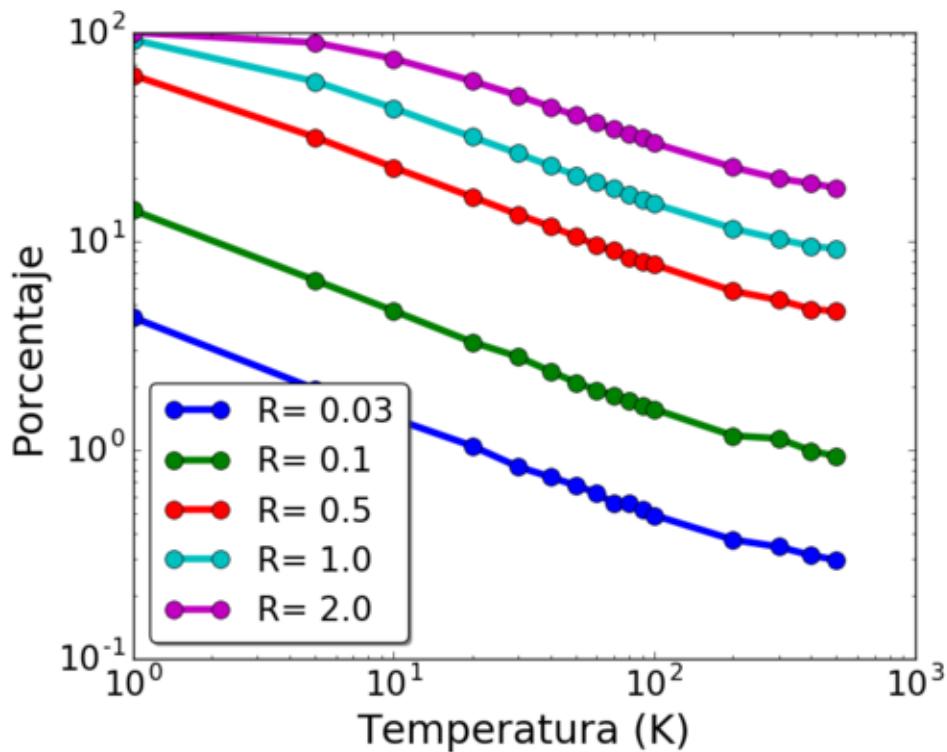


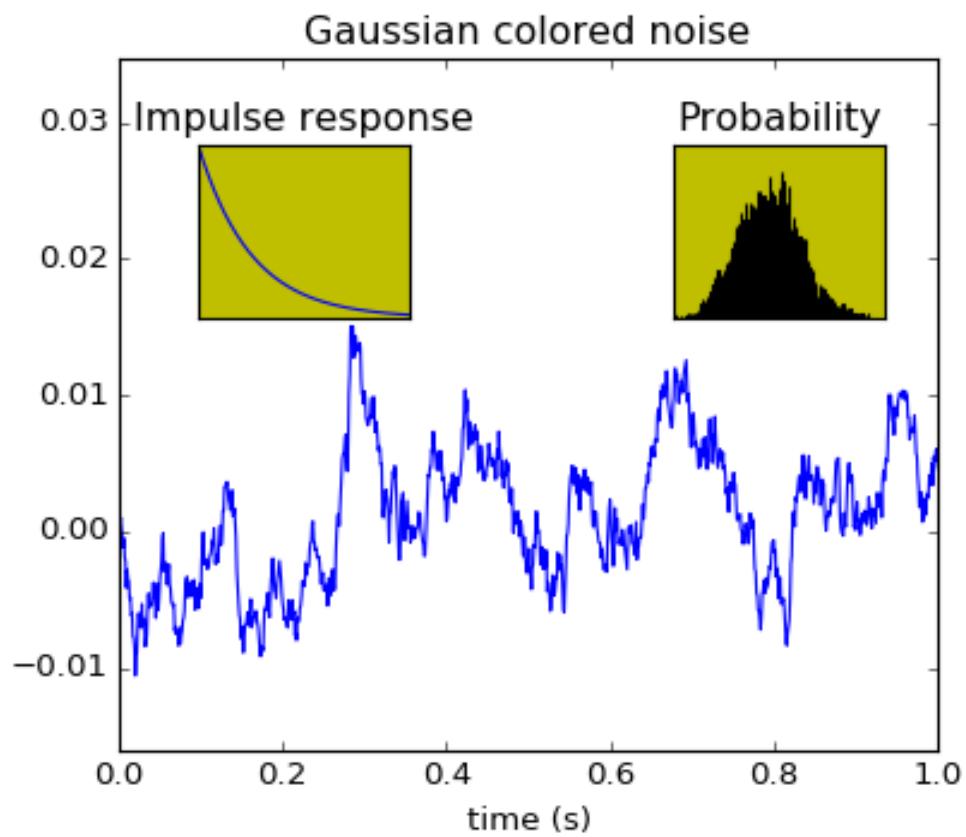
Un último punto que no puede dejar de mencionarse es que Python es libre (y gratis). Esto significa que cada versión nueva puede simplemente descargarse e instalarse sin limitaciones, sin licencias. Además, al estar disponible el código fuente uno podría modificar el lenguaje –una situación que no es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

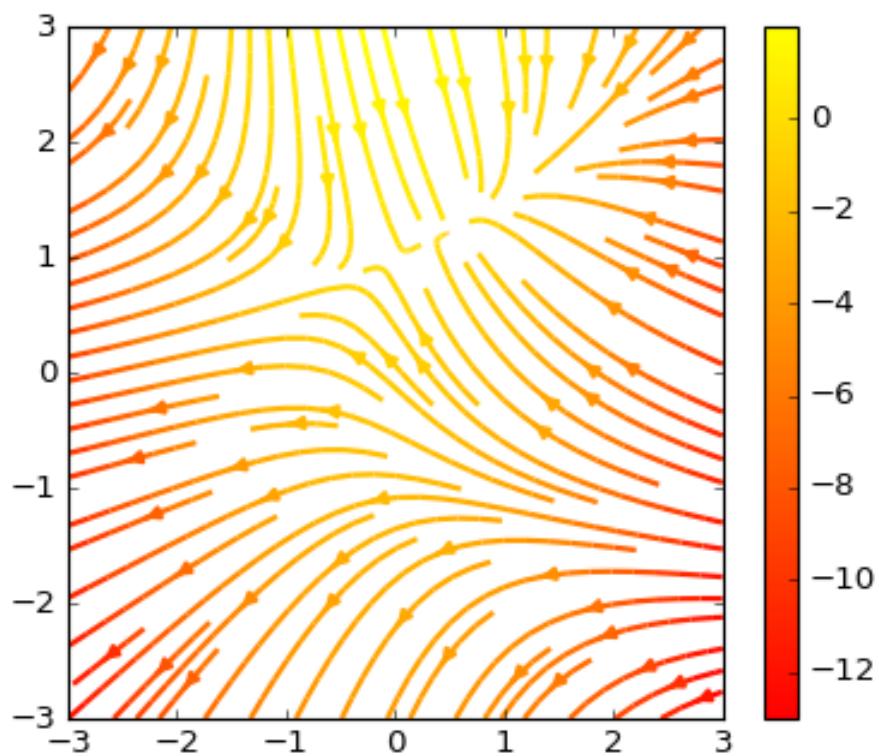
1.2 Visita y excursión a aplicaciones de Python

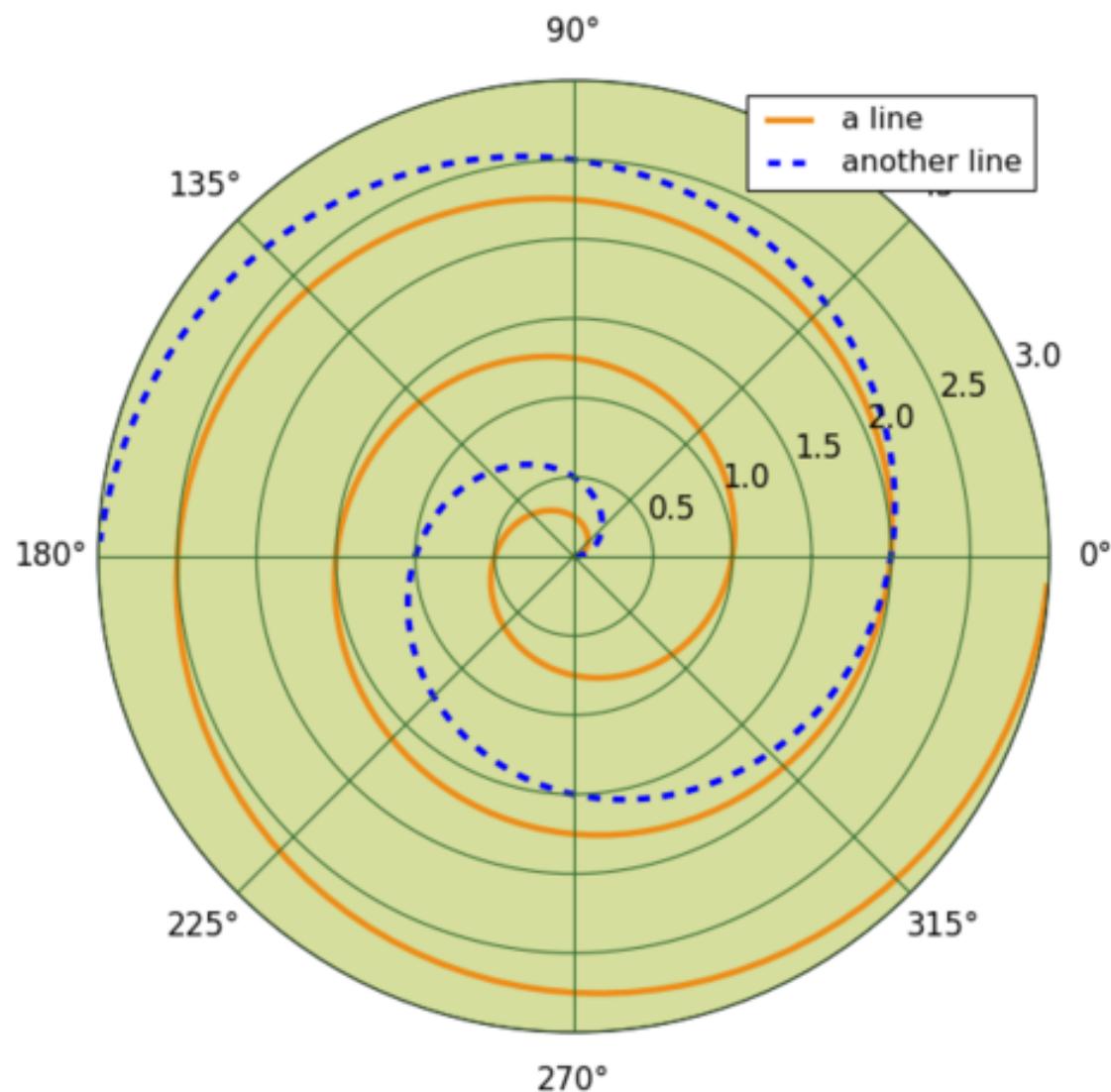
1.2.1 Graficación científica en 2D

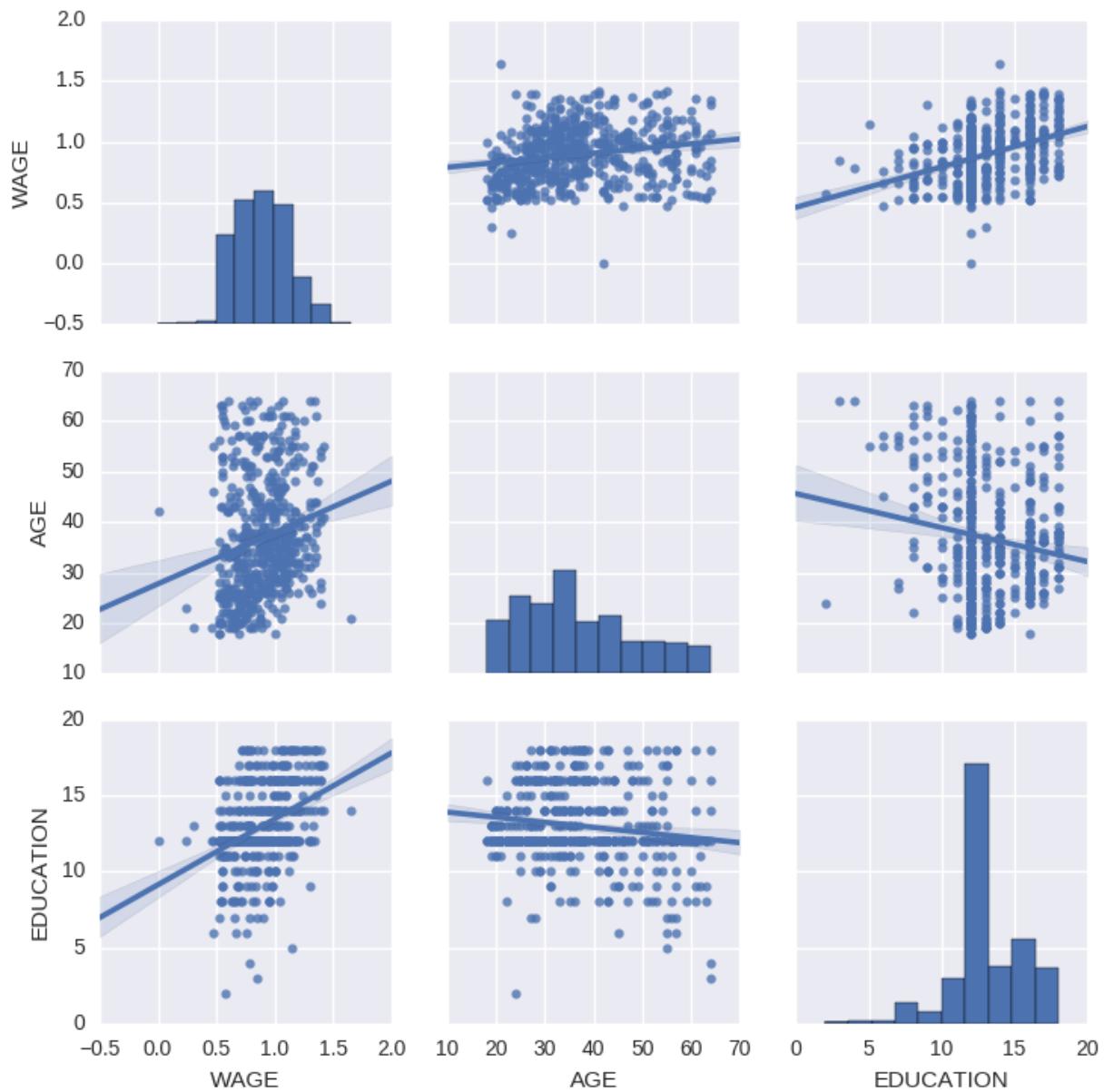
La biblioteca **matplotlib** es una de las mejores opciones para hacer gráficos en 2D, con algunas posibilidades para graficación 3D.









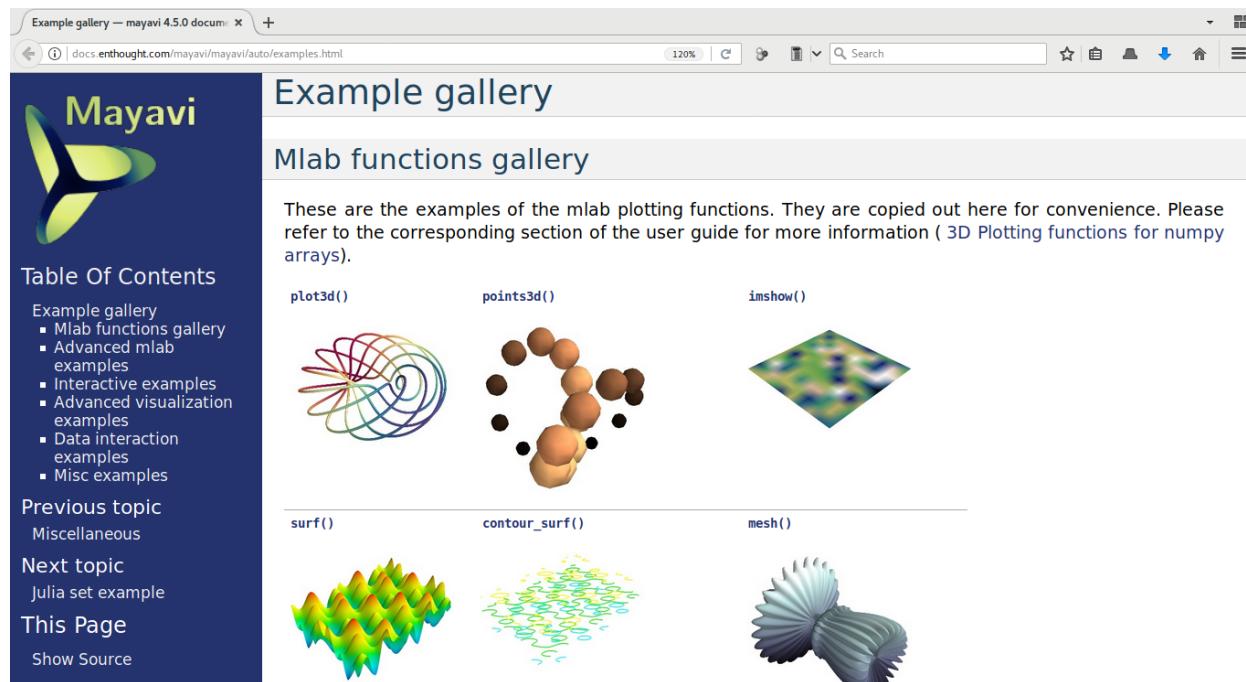


El mejor lugar para un acercamiento es posiblemente la Galería de matplotlib

El último ejemplo está creado utilizando *seaborn*, un paquete para visualización estadística (tomado de Scipy Lecture Notes)

1.2.2 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:



1.2.3 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.

A screenshot of a web browser displaying the PyVISA documentation. The page title is "PyVISA: Control your instruments with Python". It features the PyVISA logo and a brief description of the package. Below that is a code snippet demonstrating how to read self-identification from a Keithley Multimeter using three lines of Python code. The left sidebar contains a navigation menu with links like "Installation", "Tutorial", "Reading and Writing values", "Resources", "A frontend for multiple backends", "PyVISA Shell", "Architecture", "VISA resource names", "Migrating from PyVISA < 1.5", "Contributing to PyVISA", "Frequently asked questions", and "NI-VISA Installation".

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 blue servo 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'.

1.2.4 Otras aplicaciones

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

1.3 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

1.4 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:

1.4.1 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

1.4.2 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,

1.5 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. Su uso ha disminuido mucho en los últimos años y esencialmente todo el ecosistema de bibliotecas se ha convertido a Python-3
 - **Python3.X** (Python 3) es la versión actual. 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 gran 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 en aplicaciones nuevas.
-

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

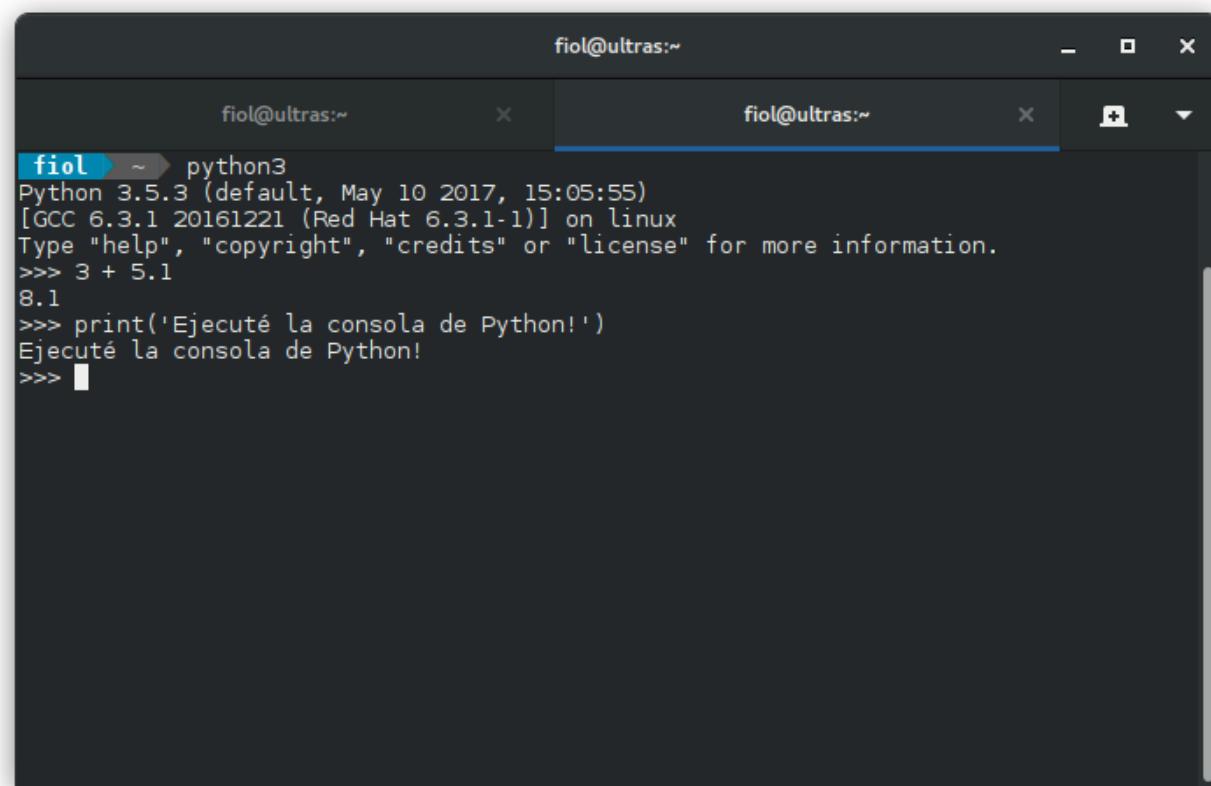
Algunas fuentes de ayuda *constante* son:

- 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!". A cursor is visible at the end of the print statement.

```
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*”.

```
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

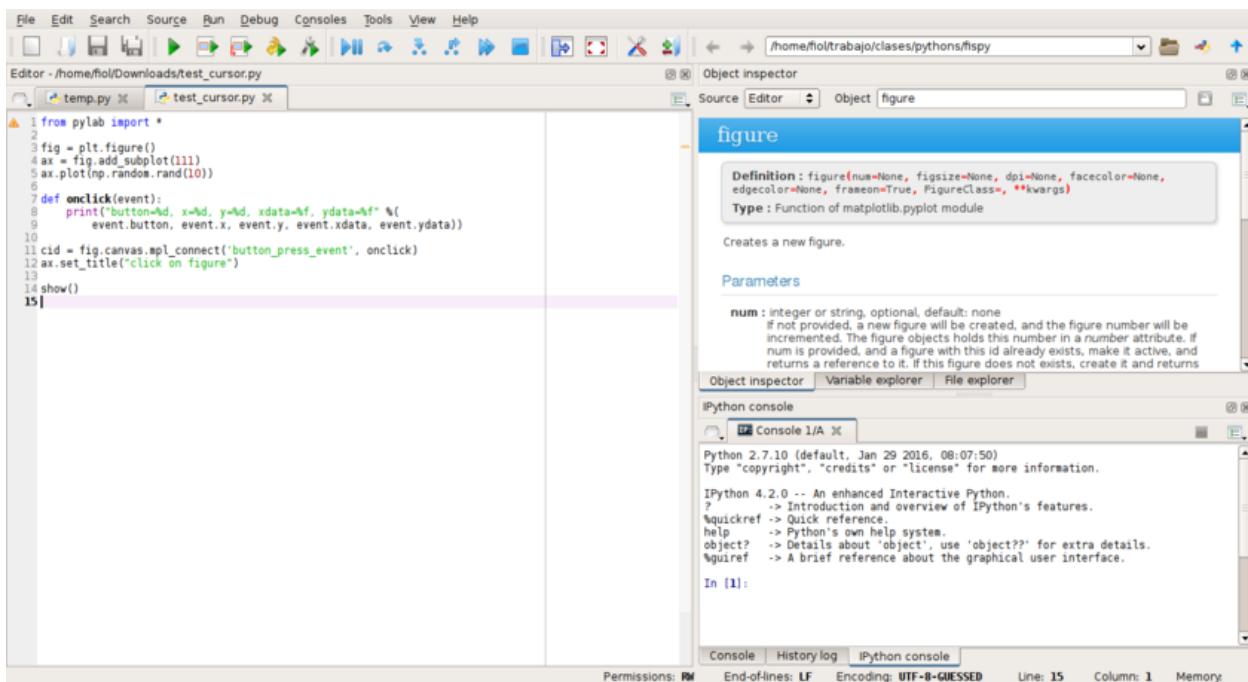
Una forma muy común/poderosa de usar Python es en forma *no interactiva*, escribiendo *programas* o *scripts*. Esto es, escribir nuestro código en un archivo con extensión .py para luego ejecutarlo con el intérprete. Por ejemplo, podemos crear un archivo *hello.py* (al que se le llama *módulo*) con este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

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

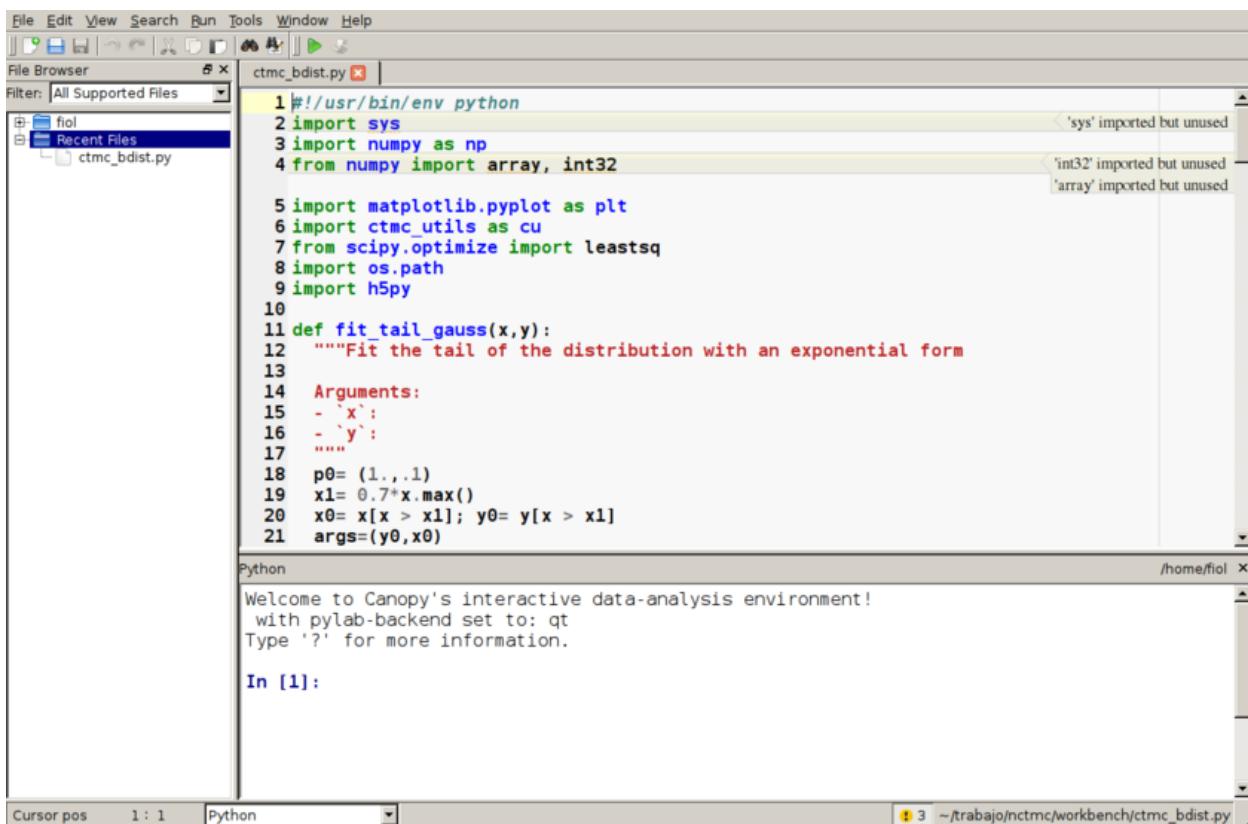
Si ejecutamos `python hello.py` o `ipython hello.py` se ejecutará el interprete Python y obtendremos el resultado esperado (impresión por pantalla de “Hola Mundo!”, sin las comillas)

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 Comandos de Ipython

2.2.1 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 %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.
```

2.2.2 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:

```
%pwd
```

```
'/home/fiol/Clases/IntPython/clases-pyton/clases'
```

```
%cd ~
```

```
/home/fiol
```

```
%pwd
```

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

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

```
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("programa 1")
```

```
%run prog1.py
```

```
programa 1
```

```
%hist
```

```
%lsmagic  
%pwd  
%cd ~  
%pwd  
%cd  
%mkdir scripts  
%cd scripts  
%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
```

2.2.3 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 "1+2" >> prog1.py
```

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

```
%cat prog1.py
```

```
print("programa 1")  
1+2  
print('hola otra vez')
```

```
%run prog1.py
```

```
programa 1  
hola otra vez
```

```
!date
```

```
mié 02 feb 2022 15:35:57 -03
```

2.3 Ejercicios 01 (a)

1. Abra una terminal (consola) o notebook y utilícela como una calculadora para realizar las siguientes acciones:
 - Suponiendo que, de las cuatro horas de clases, tomamos dos descansos de 15 minutos cada uno 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 en este curso?
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, muestre la fecha y hora
 - Borre el subdirectorio `tmp`
3. Para cubos de lados de longitud $L = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
4. Para esferas de radios $r = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
5. Fíjese si alguno de los valores de $x = 2,05, x = 2,11, x = 2,21$ es un cero de la función $f(x) = x^2 + x/4 - 1/2$.

2.4 Conceptos básicos de Python

2.4.1 Características generales del lenguaje

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

- Las funciones, bloques, ámbitos están definidos por la indentación
- Es un lenguaje interpretado (no se compila separadamente)
- Provee tanto un entorno interactivo como ejecución de programas completos
- 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 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	

2.4.2 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. Veamos un ejemplo donde 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'>
```

Acá usamos la función `type()` que retorna el tipo de su argumento. Acá ilustramos una de las características salientes de Python: El tipo de variable se define 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`.

Lo que está pasando acá en realidad es que la variable `a` del tipo entero en la primera, en la segunda línea se destruye (después de ser multiplicada por `1.5`) y se crea una nueva variable del tipo `float` que se llama `a` a la que se le asigna el valor real.

En Python 3 la división entre números enteros da como resultado un número de punto 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 $3/4=0.75$.

Nota: La función print

Estuvimos usando, sin hacer ningún comentario, la función `print(arg1, arg2, arg3, ..., sep=' ', end='\n', file=sys.stdout, flush=False)` que acepta un número variable de argumentos. Esta función 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)
```

Help on built-in function print in module builtins:

```
print(*args, **kwargs)
      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.
```

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

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

```
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 y separar la palabra `print` de sus argumentos con un espacio.

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 operaciones 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: 0.6666666666666667
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):

```
a = 1280
b = 1280
print ('b is a: ', b is a)
```

```
b is a: False
```

```
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)
```

Clases de Python

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

Acá vemos que `None` es “único”, en el sentido de que si dos variables son `None`, entonces son el mismo objeto.

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. Veamos algunos ejemplos

```
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.

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

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

Este es un detalle de implementación y nuestros programas no deberían depender de este comportamiento.

```
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 la implementación que estamos usando, se 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.

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'>
```

2.5 Ejercicios 01 (b)

6. 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 $z=1+0.5i$ es $1-0.5j$ ”
 - “El conjugado de $z=(1+0.5i)^2$ es ...” (con el valor correspondiente)
-

CAPÍTULO 3

Clase 2: Tipos de datos y control

Nota: 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
- Vimos algunos comandos mágicos y como enviar 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",r," cm, el área es",A,"cm2")
print("Para una esfera de radio",r," cm, el área es",As,"cm2")
print("Para una esfera de radio",r," cm, el volumen es",V,"cm3")
```

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.

3.1 Tipos simples: Números

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

Nota: 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.

Todos los números, al igual que otros tipos, son objetos y tienen definidos algunos métodos que pueden ser útiles.

```
a = 3                                     # Números enteros
print(type(a), a.bit_length(), sep="\n")
```

```
<class 'int'>
2
```

```
b = 127
print(type(b))
print(b.bit_length())
```

```
<class 'int'>
7
```

En estos casos, usamos el método `bit_length` de los enteros, que nos dice cuántos bits son necesarios para representar un número. Para verlo utilicemos la función `bin()` que nos da la representación en binario de un número entero

```
# bin nos da la representación en binarios
print(a, "=", bin(a), "->", a.bit_length(),"bits")
print(b, "=", bin(b), "->", b.bit_length(),"bits")
```

```
3 = 0b11 -> 2 bits
127 = 0b1111111 -> 7 bits
```

Vemos que el número 3 se puede representar con dos bits, y para el número 127 se necesitan 7 bits.

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)
```

0x1.1c80000000000p+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

Recordemos, 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 Tipos compuestos

En Python, además de los tipos simples (números y booleanos, entre ellos) existen tipos compuestos, que pueden contener más de un valor de algún tipo. Entre los tipos compuestos más importantes vamos a referirnos a:

- **Strings**

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

Ejemplo: s = "abc" (el elemento s[0] tiene el valor "a").

- **Listas**

Las listas son tipos que pueden contener más de un elemento de cualquier tipo. Los tipos de los elementos pueden ser diferentes. Las listas se definen separando los diferentes valores con comas, encerrados entre corchetes. Se puede referir a un elemento por su índice.

Ejemplo: `L = ["a", 1, 0.5 + 1j]` (el elemento `L[0]` es igual al *string* "a").

- **Tuplas**

Las tuplas se definen de la misma manera que las listas pero con paréntesis en lugar de corchetes. Ejemplo: `T = ("a", 1, 0.5 + 1j)`.

- **Diccionarios**

Los diccionarios son contenedores a cuyos elementos se los identifica con un nombre (*key*) en lugar de un índice. Se los puede definir dando los pares `key:value` entre llaves

Ejemplo: `D = {'a': 1, 'b': 2, 1: 'holo', 2: 3.14}` (el elemento `D['a']` es igual al número 1).

3.3 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*

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

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

```
saludo, saludo2
```

```
('Hola Mundo', 'Hola Mundo')
```

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 = '"Cómo te va" dijo el murguista a la muchacha'
```

```
otro
```

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

```
dijo
```

```
'"Cómo te va" dijo el murguista a la muchacha'
```

```
respondio = "Le dijo \"Bien\" y lo dejó como si nada"
```

```
respondio
```

```
'Le dijo "Bien" y lo dejó como si nada'
```

```
consimbolos = 'þ€→\"oó@¬'
```

```
consimbolos
```

```
'þ€→"oó@¬'
```

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 estraordinaria
    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
```

En Python se puede utilizar cualquier carácter que pueda ingresarse por teclado, ya que por default Python-3 trabaja usando la codificación UTF-8, que incluye todos los símbolos que se nos ocurran. Por ejemplo:

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

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

3.3.1 Operaciones

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

```
s = saludo + ' -> ' + otro + '\n'  
s = s + "chau"  
print (s) # Suma de strings
```

Hola Mundo -> that's all
chau

Como vemos la suma de *strings* es simplemente la concatenación de los argumentos. La multiplicación de un entero n por un *string* es simplemente la suma del *string* n veces:

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

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

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

```
print ("Multiplicación por enteros de strings:", 7*a)
print ("Multiplicación por enteros de enteros:", 7*b)
```

```
Multiplicación por enteros de strings: 11111111  
Multiplicación por enteros de enteros: 7
```

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

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

longitud **del** saludo = 10 caracteres

Podemos usar estas operaciones para realizar el centrado manual de una frase:

```
titulo = "Centrado manual simple"
n = int((60-len(titulo))/2)      # Para un ancho de 60 caracteres
print ((n)* '<', titulo , (n)* '>')
#
saludo = 'Hola Mundo'
n = int((60-len(saludo))/2)      # Para un ancho de 60 caracteres
print (n*'*', saludo, n*'*)
```

3.3.2 Iteración y 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*

Indexado 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*)

Podemos referirnos a un carácter o una parte de una cadena de caracteres mediante su índice. Los índices en **Python** empiezan en 0.

```
s = "0123456789"
print ('Primer caracter :', s[0])
print ("Segundo caracter :", s[1])
print ("El último carácter :", s[-1])
print ("El anteúltimo carácter :", s[-2])
```

```
Primer carácter : 0
Segundo carácter : 1
El último carácter : 9
El anteúltimo carácter : 8
```

También podemos elegir un subconjunto de caracteres:

```
print ('Los tres primeros: ', s[:3])
print ('Todos a partir del tercero: ', s[3:])
print ('Los últimos dos: ', s[-2:])
print ('Todos menos los últimos dos:', s[:-2])
```

```
Los tres primeros: Hol
Todos a partir del tercero: a Mundo -> that's all
chau
Los últimos dos: au
Todos menos los últimos dos: Hola Mundo -> that's all
ch
```

Estas “subcadenas” son cadenas de caracteres, y por lo tanto pueden utilizarse de la misma manera que cualquier otra cadena:

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

```
Holau
```

La selección de elementos y subcadenas de una cadena *s* tiene la forma

```
s[i: f: p]
```

donde `i`, `f`, `p` son enteros. La notación se refiere a la subcadena empezando en el índice `i`, hasta el índice `f` recorriendo con paso `p`. Casos particulares de esta notación son:

- Un índice simple. Por ejemplo `s[3]` se refiere al tercer elemento
- Un índice negativo se cuenta desde el final, empezando desde `-1`
- Si el paso `p` no está presente el valor por defecto es `1`. Ejemplo: `s[2:4] = s[2:4:1]`
- Si se omite el primer índice, el valor asumido es `0`. Ejemplo: `s[:2:1] = s[0:2:1]`
- Si se omite el segundo índice, el valor asumido es `-1`. Ejemplo: `s[1::1] = s[1:-1:1]`
- Notar que puede omitirse más de un índice. Ejemplo: `s[::2] = s[0:-1:2]`

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

```
Hola Mundo -> that's all
chau
Hl
Hl ud >ta' l
hu
uahc
lla s'taht >- odnuM aloH
uclsa n o
```

Veamos algunas operaciones que se pueden realizar 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(a.split())
```

```
['La', 'mar', 'estaba', 'serena!']
```

En este caso, tanto `replace()` como `split()` son métodos que ya tienen definidos los *strings*.

Un método es una función que está definida junto con el tipo de objeto. En este caso el string. Hay más información sobre todos 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 ('\nPrimer programa en cualquier lenguaje:\n\t\t' + a, 2*'\\n')
print (80*'-')
print ('Otro texto:', b, sep='\n\t')
print ('Longitud del texto: ', len(b), 'caracteres')
```

Primer programa 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)
```

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

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

3.3.3 Formato de strings

En python se le puede dar formato a los strings de distintas maneras. Vamos a ver dos opciones: - Uso del método `format` - Uso de “f-strings”

El método `format()` es una función que busca en el strings las llaves y las reemplaza por los argumentos. Veamos esto con algunos ejemplos:

```
a = 2022
m = 'Feb'
d = 8
s = "Hoy es el día {} de {} de {}".format(d, m, a)
print(s)
```

(continué en la próxima página)

(provine de la página anterior)

```
print("Hoy es el día {}-{}-{}".format(d,m,a))
print("Hoy es el día {}-{}-{}".format(d,m,a))
print("Hoy es el día {}-{}-{}".format(d,m,a))
```

```
Hoy es el día 8 de Feb de 2022
Hoy es el día 8/Feb/2022
Hoy es el día 8/Feb/2022
Hoy es el día 2022/Feb/8
```

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

```
datos-2022-Feb-8.dat
```

```
pi = 3.141592653589793
s1 = "El valor de es {}".format(pi)
s2 = "El valor de con cuatro decimales es {:.4f}".format(pi)
print(s1)
print(s2)
print("El valor de con seis decimales es {:.6f}".format(pi))
print("{:03d}".format(5))
print("{:3d}".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
5
```

Más recientemente se ha implementado en Python una forma más directa de intercalar datos con caracteres literales, mediante *f-strings*:

```
print(f"el valor de es {pi}")
print(f"El valor de con seis decimales es {pi:.6f}")
print(f"El valor de 2 con seis decimales es {2*pi:.6f}")
```

```
el valor de es 3.141592653589793
El valor de con seis decimales es 3.141593
El valor de 2 con seis decimales es 6.283185
```

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
print ("a es de tipo {0}\nb es de tipo {1}\nnc 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)

Input In [62], 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)

Input In [64], 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 (a)

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 ENTREGAR. 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 02_SuApellido.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 pena estraordinaria
Como 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` (*¿x es un elemento de L?*)
 - `x not in L` (*¿x 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)

```
cuadrados[0]
```

```
1
```

```
cuadrados[3]
```

```
25
```

Clases de Python

```
cuadrados[-1]
```

```
25
```

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

```
[1, 16]
```

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

```
[16, 25]
```

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)
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)`

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)
```

Clases de Python

```
[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 Tuplas

Las tuplas son objetos similares a las listas, sobre las que se puede iterar y seleccionar partes según su índice. La principal diferencia es que son inmutables mientras que las listas pueden modificarse. Los ejemplos anteriores del tipo `L[0] = -9` resulta en un error si lo intentamos con tuplas

```
L1 = [0,1,2,3,4,5] # Las listas se definen con corchetes
T1 = (0,1,2,3,4,5) # Las tuplas se definen con paréntesis
```

```
L1[0] = -1
print(f"L1[0] = {L1[0]}")
print(f'L1[0] = {L1[0]}')
```

```
L1[0] = -1
L1[0] = -1
```

```
try:
    T1[0] = -1
    print(f"T1[0] = {T1[0]}")
except:
    print('Tuples son inmutables')
```

```
Tuples son inmutables
```

Las tuplas se usan cuando uno quiere crear una “variable” que no va a ser modificada. Además códigos similares con tuplas pueden ser un poco más rápidos que si usan listas.

Un uso común de las tuplas es el de asignación simultánea a múltiples variables:

```
a, b, c = (1, 3, 5)
```

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

```
1 3 5
```

```
# Los paréntesis son opcionales en este caso
a, b, c = 4, 5, 6
print(a,b,c)
```

```
4 5 6
```

Un uso muy común es el de intercambiar el valor de dos variables

```
print(a,b)
a, b = b, a           # swap
print(a,b)
```

```
4 5
5 4
```

3.6.3 Rangos

Los objetos de tipo `range` representan una secuencia inmutable de números y se usan habitualmente para ejecutar un bucle `for` un número determinado de veces. El formato es:

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

```
range(2)
```

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

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

```
range
```

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

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

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

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

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

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

3.6.4 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]
```

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

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))
```

3.7.2 Módulo cmath

El módulo `math` no está diseñado para trabajar con números complejos, 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 (b)

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 $L1$ tenga el valor del último elemento de L y repita el inciso anterior.
 6. Utilizando funciones y métodos de *strings* en la cadena de caracteres:

```
s1='En un lugar de la Mancha de cuyo nombre no quiero acordarme'
```

- 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 de la oración.
 - 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:

$$v = v_0 - gt$$

$$z = z_0 + v_0 t - gt^2/2.$$

3.8.1 Adicionales

9. Calcular la suma:

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

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

10. 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: Tipos complejos y control de flujo

4.1 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.

4.1.1 Creación

```
t1 = list(range(1,11))
t2 = [2*i**2 for i in t1]
```

```
d01 = {}
d02 = 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'), ('A', 27), ('Z', 13), ('M', 26.98153863) ] )
d4 = {n: n**2 for n in range(6) }
```

Acá estamos creando diccionarios de diferentes maneras:

- d01 y d02 corresponden a diccionarios vacíos
- d1 y d2 se crean utilizando el formato clave: valor
- d3 se crea a partir de una lista de 2-tuplas donde el primer elemento de cada tupla es la clave y el segundo el valor
- d4 se crea mediante una “comprensión de diccionarios”

```
print(d01)
print(d02)
```

Clases de Python

```
{ }  
{ }
```

```
print(d4)
```

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

Notar que los diccionarios d1, d2, d3 tienen las mismas claves y valores, pero se crean con distinto orden

```
print(d1)  
print(f" {d1 == d2} = {d1 == d3} ")
```

```
{'S': 'A1', 'Z': 13, 'A': 27, 'M': 26.98153863}  
(d1 == d2) = True y (d1 == d3) = True
```

Como ocurre con otros tipos complejos, al realizar una asignación de un diccionario a otro, no se crea un nuevo objeto

```
d5 = d2  
print(d5 == d2)  
print(d5 is d2)
```

```
True  
True
```

```
d1 is d2
```

```
False
```

y, por lo tanto, si modificamos uno de ellos también estamos modificando el otro.

Para realizar una copia independiente utilizamos el método `copy()`:

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

```
True  
False
```

4.1.2 Selección de elementos

Para seleccionar un elemento de un diccionario, se lo llama por su clave (`key`)

```
d1['A']
```

```
27
```

```
d1['M']
```

```
26.98153863
```

```
d1["S"]
```

```
'Al'
```

Un uso muy común de los diccionarios es la descripción de estructuras complejas, donde cada campo tiene un significado, como podría ser por ejemplo una agenda

```
entrada = {'nombre': 'Juan',
           'apellido': 'García',
           'edad': 109,
           'dirección': '''Av Bustillo 9500,'''',
           'cod': 8400,
           'ciudad': "Bariloche"}
```

```
print ('Nombre: ', entrada['nombre'])
print ('\nDiccionario:')
print ((len("Diccionario:") * "-") + "\n")
print (entrada)
```

```
Nombre: Juan
```

```
Diccionario:
```

```
-----
{'nombre': 'Juan', 'apellido': 'García', 'edad': 109, 'dirección': 'Av Bustillo 9500,
→', 'cod': 8400, 'ciudad': 'Bariloche'}
```

```
entrada['cod']
```

```
8400
```

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

```
entrada
```

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

Un diccionario puede tener elementos de distinto tipo, tanto en claves como en valores

```
telefono = entrada['tel']
print(telefono['cel'])
print(entrada['tel']['cel'])
```

```
1213
1213
```

4.1.3 Acceso a claves y valores

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

```
print ('\n\nKeys:')
print (list(entrada.keys()))
print ('\n\nValues:')
print (list(entrada.values()))
print ('\n\nItems:')
print (list(entrada.items()))
```

```
Keys:
['nombre', 'apellido', 'edad', 'dirección', 'cod', 'ciudad', 'tel']

Values:
['Juan', 'García', 109, 'Av Bustillo 9500,', 8400, 'Bariloche', {'cel': 1213, 'fijo': ↵23848}]

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

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

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

```
dict(it)
```

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

4.1.4 Modificación o adición de campos

Si queremos modificar un campo o agregar uno nuevo simplemente asignamos un nuevo valor como lo haríamos para una variable. En el siguiente ejemplo agregamos un nuevo campo indicando el “pais” y modificamos el valor de la ciudad:

```
entrada['pais']= 'Argentina'
entrada['ciudad']= "San Carlos de Bariloche"
# imprimimos
print ('\n\nDatos:\n')
print (entrada['nombre'] + ' ' + entrada['apellido'])
print (entrada[u'dirección'])
print (entrada['ciudad'])
print (entrada['pais'])
```

Datos:

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

```
d2 = {'provincia': 'Río Negro', 'nombre': 'José'}
print (60*'+'+'\nOtro diccionario:')
print ('d2=', d2)
print (60*'+')
```

Otro diccionario:

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

Vimos que se pueden asignar campos a diccionarios. También se pueden completar utilizando otro diccionario, usando el método `update()`

```
print (f'{entrada = }')
entrada.update(d2) # Corregimos valores o agregamos nuevos si no existen
print ("\nNuevo valor:\n")
print (f'{entrada = }')
```

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

Nuevo valor:

```
entrada = {'nombre': 'José', 'apellido': 'García', 'edad': 109, 'dirección': 'Av_
↪Bustillo 9500,', 'cod': 8400, '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 (f'{provincia' in entrada = }')
if 'provincia' in entrada:
    del entrada['provincia']
print (f'{provincia' in entrada = })
```

Clases de Python

```
'provincia' in entrada = True
'provincia' in entrada = False
```

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

```
entrada[1] = [2,3]           # Agregamos el campo `1`
```

```
entrada
```

```
{'nombre': 'José',
'apellido': 'García',
'edad': 109,
'dirección': 'Av Bustillo 9500,',
'cod': 8400,
'ciudad': 'San Carlos de Bariloche',
'tel': {'cel': 1213, 'fijo': 23848},
'pais': 'Argentina',
1: [2, 3]}
```

```
entrada.pop(1)
```

```
[2, 3]
```

```
entrada
```

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

4.2 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
```

```
{'gato', 'león', 'perro'}
```

Para crear un conjunto vacío utilizamos la palabra set (). Notar que: conj = {} crearía un diccionario:

```
conj = set()
print(conj, type(conj))
```

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

4.2.1 Operaciones entre conjuntos

```
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'}
```

4.2.2 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)
```

Clases de Python

```
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}
```

Para remover un elemento que pertenece al conjunto usamos `remove()`

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

```
c
```

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

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

```
KeyErrorTraceback (most recent call last)  
<ipython-input-52-b4d051d0e502> in <module>  
----> 1 c.remove(2)
```

```
KeyError: 2
```

pero da un error si el elemento que queremos remover no pertenece al conjunto. Si no sabemos si el elemento existe, podemos usar el método `discard()`

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

```
c
```

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

4.3 Control de flujo

4.3.1 if/elif/else

En todo lenguaje necesitamos controlar el flujo de una ejecución segun una condición Verdadero/Falso (booleana). *Si (condicion) 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 C
```

y en Python es muy parecido!

```
if condición_1:
    bloque A
elif condición_2:
    bloque B
elif condición_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:
    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("Siga participando!")
```

Bueno, al menos aprobó!

4.3.2 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=', ')
```

Clases de Python

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

Veamos otro ejemplo, iterando sobre una lista:

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

```
auto
casa
perro
gato
árbol
lechuza
banana
```

En estos ejemplos, en cada iteración `L` toma sucesivamente los valores de `Lista`. La primera vez es `L='auto'`, la segunda `L='casa',...` El cuerpo del *loop for*, como todos los bloques en **Python** está definido por la **indentación**. La última línea está fuera del loop y se ejecuta al terminar todas las iteraciones del `for`.

```
for L in Lista:
    print(f'En la palabra {L} hay {L.count("a")} letras "a"')

print(f'\nLa palabra más larga es {max(Lista, key=len)}')
```

```
En la palabra auto hay 1 letras "a"
En la palabra casa hay 2 letras "a"
En la palabra perro hay 0 letras "a"
En la palabra gato hay 1 letras "a"
En la palabra árbol hay 0 letras "a"
En la palabra lechuza hay 1 letras "a"
En la palabra banana hay 3 letras "a"
```

```
La palabra más larga es lechuza
```

Otro ejemplo:

```
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
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 otras características del bloque `for`.

```
suma = 0
cuadrados = []
for i,elem in enumerate(range(3,30)):
    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
19 22 484 2020
21 24 576 2596
23 26 676 3272
25 28 784 4056
sumatoria de números pares al cuadrado entre 3 y 20: 4056
cuadrados= [16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784]
```

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

Clases de Python

- 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(f'Índice: {j} -> {L[j]} ({len(L[j])} caracteres)')
```

```
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 j, elem in enumerate(L):
    print(f'Índice: {j} -> {elem} ({len(elem)} caracteres)')
```

```
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):
        if n % x == 0:
            print( f'{n:2d} = {x} x {n//x}' )
```

(continué en la próxima página)

(proviene de la página anterior)

```

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
```

```

a, b = 0, 1
while b < 5000:
    a, b = b, a+b
    if b == 8:
        continue
    print(b, end=' ')

```

```
1 2 3 5 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

4.4 Ejercicios 03 (a)

1. De los primeros 100 números naturales imprimir aquellos que no son divisibles por alguno de 2, 3, 5 o 7.
2. Usando estructuras de control, calcule la suma:

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

1. Incluyendo todos los valores de k
2. Incluyendo únicamente los valores pares de k .
3. 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. 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 |
=====|
"""
```

4.5 Técnicas de iteración

Introdujimos tipos complejos: strings, listas, tuples, diccionarios (dict), conjuntos (set). Veamos algunas técnicas usuales de iteración sobre estos objetos.

4.5.1 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='')
```

```
tá oanm gls, Heic
```

4.5.2 Iteración sobre elementos de dos listas

Consideremos las listas:

```
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, t 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]))
```

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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 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 cuándo 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(f'El día {j} la temperatura mínima fue {t1} y la máxima fue {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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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]
```

4.5.3 Iteraciones sobre diccionarios

```
temps = {j:{'Tmin': temp_min[j], 'Tmax':temp_max[j]} for j in range(len(temp_min))}
```

```
for k in temps:
    print(f'La temperatura máxima del día {k} fue {temps[k]["Tmax"]} y la mínima
          ↪{temps[k]["Tmin"]}')
```

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

```
temps
```

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

(continué en la próxima página)

Clases de Python

(provine de la página anterior)

```
8: {'Tmin': 4, 'Tmax': 10},  
9: {'Tmin': 0, 'Tmax': 10}}
```

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

```
temps.keys()
```

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

```
7 in temps
```

```
True
```

```
7 in temps.keys()
```

```
True
```

```
11 in temps
```

```
False
```

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

```
list(temps.items())
```

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

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

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

```
1
```

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

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

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

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

```
list(temp)
```

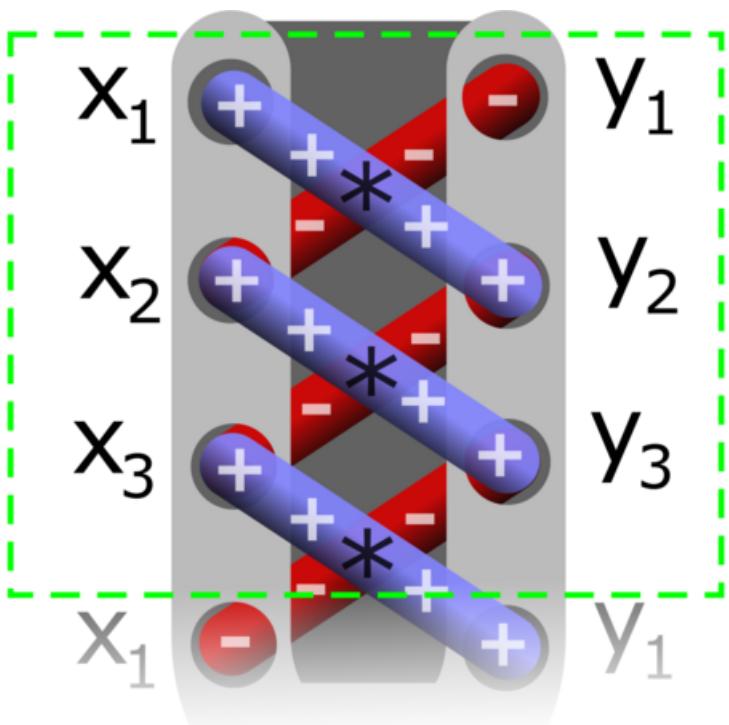
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

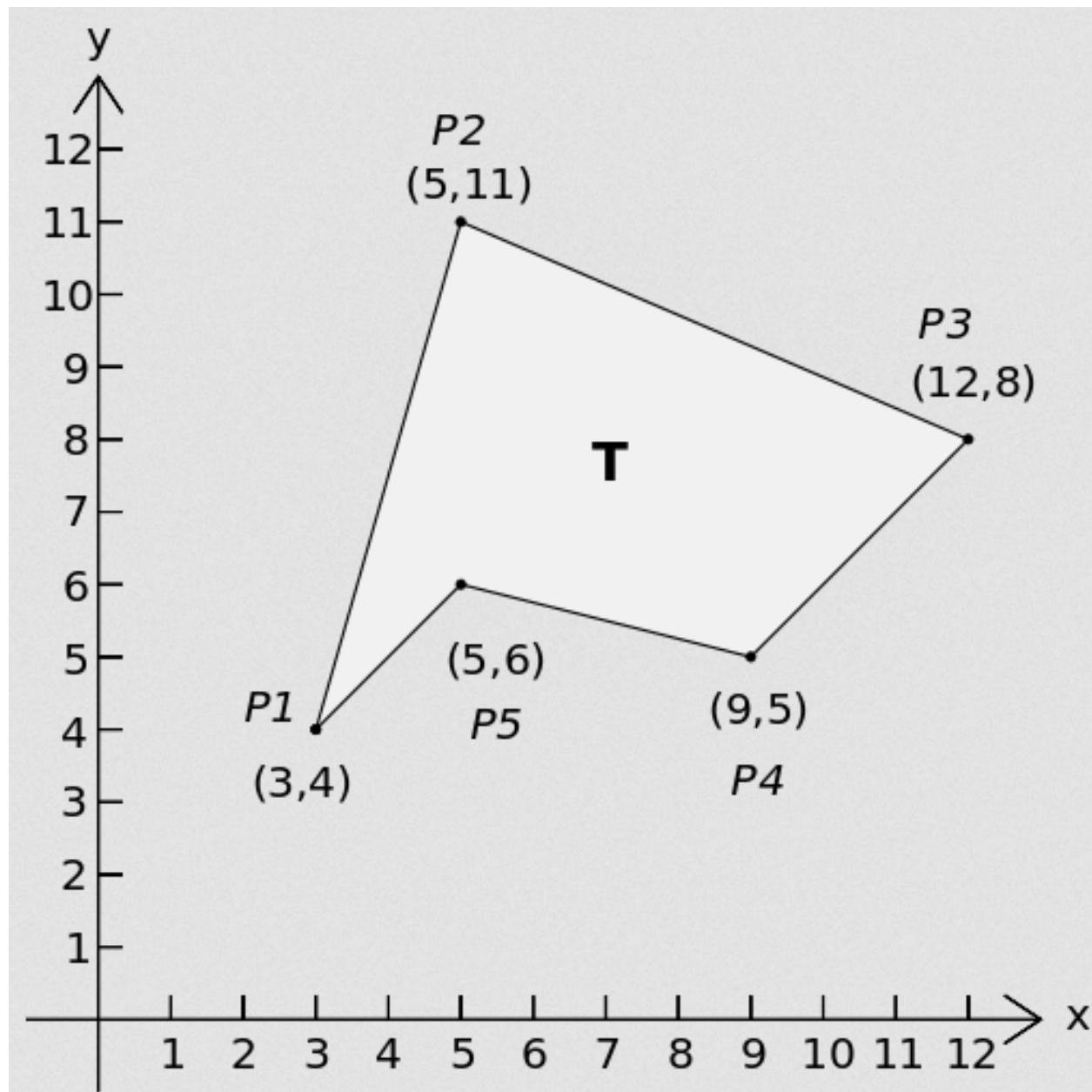
```
9 {'Tmin': 0, 'Tmax': 10}
8 {'Tmin': 4, 'Tmax': 10}
7 {'Tmin': 0, 'Tmax': 14}
6 {'Tmin': -2, 'Tmax': 11}
5 {'Tmin': -5, 'Tmax': 5}
4 {'Tmin': 4, 'Tmax': 18}
3 {'Tmin': -1, 'Tmax': 7}
2 {'Tmin': 0, 'Tmax': 13}
1 {'Tmin': -2, 'Tmax': 12}
0 {'Tmin': -3.2, 'Tmax': 13.2}
```

4.6 Ejercicios 03 (b)

5. Un método para calcular el área de un polígono (no necesariamente regular) que se conoce como fórmula del área de Gauss o fórmula de la Lazada (*shoelace formula*) consiste en describir al polígono por sus puntos en un sistema de coordenadas. Cada punto se describe como un par (x, y) y la fórmula del área está dada mediante la suma de la multiplicación de los valores en una diagonal a los que se le resta los valores en la otra diagonal, como muestra la figura



$$2A = (x_1y_2 + x_2y_3 + x_3y_4 + \dots) - (x_2y_1 + x_3y_2 + x_4y_3 + \dots)$$



- Utilizando una descripción adecuada del polígono, implementar la fórmula de Gauss para calcular su área y aplicarla al ejemplo de la figura.
 - Verificar que el resultado no depende del punto de inicio.
6. 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

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

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:

$$\begin{aligned} A(x_1, \dots, x_n) &= \bar{x} = \frac{x_1 + \dots + x_n}{n} \\ G(x_1, \dots, x_n) &= \sqrt[n]{x_1 \cdots x_n} \\ H(x_1, \dots, x_n) &= \frac{n}{\frac{1}{x_1} + \dots + \frac{1}{x_n}} \end{aligned}$$

- 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 la siguiente lista 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]
```

8. 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: Funciones

5.1 Las funciones son objetos

Para utilizar una función en **Python**, como en la mayoría de los lenguajes, se usa 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 (de cualquier tipo)

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

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

También podemos crear un diccionario donde los valores sean funciones:

```
funciones = {'suma': sum, 'mínimo': min, 'máximo': 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  
mínimo = 1  
máximo = 7.5
```

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:
```

(continué en la próxima página)

(proviene de la página anterior)

```
result.append(a)
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)>
```

```
help(fib)
```

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

```
fib(n)
    Devuelve una lista con los términos
    de la serie de Fibonacci hasta 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 el *string* de documentación

```
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)
```

Clases de Python

```
print(f'Para t = {1.5}, la velocidad será v={v:.2f} m/s\\n y estará a una altura {h:.2f} m')
```

```
Para t = 1.5, la velocidad será v=-14.70 m/s y estará a una altura 488.98 m
```

```
v, h = caida(2.2)
print(f'Para t = {2.2}, la velocidad será v={v:.2f} m/s y estará a una altura {h:.2f} m')
```

```
Para t = 2.2, la velocidad será v=-21.56 m/s y estará a una altura 476.28 m
```

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 {}m, con vel. inicial {}m/s, a t = {}, la velocidad será v={}, y estará a una altura {}'.
      format(2.2, v, 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)
```

5.3 Argumentos de las funciones

5.3.1 Ámbito de las variables en los argumentos

Consideremos la siguiente función

```
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('Fuera de la función: Originalmente x vale', x)
func1(x)
print('Fuera de la función: Ahora x vale', x)
```

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

Vemos que la variable `x` que utilizamos como argumento de la función debe ser diferente a la variable `x` que se define fuera de la función, ya que su valor no cambia al salir.

Consideremos ahora la siguiente función:

```
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))
```

La función es muy parecida, sólo que le estamos dando a la variable `x` dentro de la función un nuevo valor del tipo lista. Además usamos la función `id()` para obtener la identidad de la variable

```
x = 50
print('Fuera de la función: Originalmente x vale', x)
print('Fuera de la función: Id:', id(x))
func2(x)
print('Fuera de la función: Ahora x vale', x)
print('Fuera de la función: Id:', id(x))
```

```
Fuera de la función: Originalmente x vale 50
Fuera de la función: Id: 139797659567888
x entró a la función con el valor 50
Id adentro: 139797659567888
El nuevo valor de x es [2, 7]
Id adentro: 139797568094976
Fuera de la función: Ahora x vale 50
Fuera de la función: Id: 139797659567888
```

```
x = [50]
print('Fuera de la función: Originalmente x vale',x)
func2(x)
print('Fuera de la función: Ahora x vale',x)
print('Fuera de la función: Id:', id(x))
```

```
Fuera de la función: Originalmente x vale [50]
x entró a la función con el valor [50]
Id adentro: 139797568104128
El nuevo valor de x es [2, 7]
Id adentro: 139797568135296
Fuer de la función: Ahora x vale [50]
Fuer de la función: Id: 139797568104128
```

```
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: 139797619434112
El nuevo valor de x es [2, 7]
Id adentro: 139797568135296
Ahora x vale [50]
Id afuera: 139797619434112
```

¿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 se crea una nueva variable y el nombre x queda asociado a la nueva variable.
- 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 reasignar el nombre siempre se crea una nueva variable (para cualquier tipo).

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]
```

Como no estamos redefiniendo la variable, sino que la estamos modificando, el nuevo valor se mantiene al terminar la ejecución de la función. Otra variante:

```
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 modificamos la variable (solo se puede para tipos mutables), asignando un valor a uno o más de sus elementos o agregando/removiendo elementos, la copia sigue apuntando a la variable original y el valor de la variable, definida originalmente afuera, cambia.

5.3.2 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
```

5.3.3 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]
```

El argumento opcional `x` tiene como valor por defecto una lista vacía, entonces esperaríamos que el valor de `x` sea igual a `x1`, y en este caso imprima “El nuevo valor de `x` es [2]”. Sin embargo, entre llamadas mantiene el valor de `x` anterior. El valor por defecto se fija en la definición y en el caso de tipos mutables puede modificarse.

Nota: 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)
```

5.3.4 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 dar una explicación rápida. Consideremos la función `f`, que imprime sus argumentos:

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

Clases de Python

```
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`

```
f(1.5, 2, 3, 5, anteultimo= 9, ultimo = -1)
```

```
p: 1.5, tipo: <class 'float'>
args: (2, 3, 5), tipo: <class 'tuple'>
kwargs: {'anteultimo': 9, 'ultimo': -1}, tipo: <class 'dict'>
```

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

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

En estas otras llamadas a la función, todos los argumentos que se pasan indicando el nombre se asignan a un diccionario.

Al definir una función, con la construcción `*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-105-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`

5.4 Ejercicios 4 (a)

- 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
```

5.5 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 “empaqueamiento” y “desempaqueamiento” automático de variables.

Al definir un número variable de argumentos de la forma:

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

Clases de Python

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]
```

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

    """

```

(continué en la próxima página)

(proviene de la página anterior)

```
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)
```

5.6 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)]
```

5.7 Ejercicios 04 (b)

2. 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
```

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

5.8 Funciones que toman como argumento una función

```
def aplicar_fun(f, L):
    """Aplica la función f a cada elemento del iterable L
    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'`.

5.9 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

Clases de Python

```
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']
```

5.10 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: str.lower(a[::-1]))
```

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

5.11 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
    """

```

(continué en la próxima página)

(proviene de la página anterior)

```

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.

5.11.1 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(f'I ( {n+1} x^{n} , {a}, {b} ) = {I:.5f}')

```

```

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`. 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
```

5.12 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(f'{x:.1f} {f2(x):5.2f} {f3(x):6.2f}')
```

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

(continué en la próxima página)

(proviene de la página anterior)

4.4	19.36	170.37
4.6	21.16	194.67
4.8	23.04	221.18

5.13 Ejercicios 04 (c)

4. 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.

5. **PARA ENTREGAR.** Describimos una grilla de **sudoku** como un string de nueve líneas, cada una con 9 números, con números entre 1 y 9. Escribir un conjunto de funciones que permitan chequear si una grilla de sudoku es correcta. Para que una grilla sea correcta deben cumplirse las siguientes condiciones

- Los números están entre 1 y 9
 - En cada fila no deben repetirse
 - En cada columna no deben repetirse
 - En todas las regiones de 3x3 que no se solapan, empezando de cualquier esquina, no deben repetirse
1. Escribir una función que convierta un string con formato a una lista bidimensional. El string estará dado con nueve números por línea, de la siguiente manera (los espacios en blanco en cada línea pueden variar):

```
sudoku = """145327698
839654127
672918543
496185372
218473956
753296481
367542819
984761235
521839764"""
```

2. Escribir una función `check_repetidos()` que tome por argumento una lista (unidimensional) y devuelva verdadero si la lista tiene elementos repetidos y falso en caso contrario (puede ser conveniente explorar el uso de `set`).
3. Escribir la función `check_sudoku()` que toma como argumento una grilla (como una lista bidimensional de 9x9) y devuelva verdadero si los números corresponden a la resolución correcta del Sudoku y falso en caso contrario. Note que debe verificar que los números no se repiten en filas, ni en columnas ni en recuadros de 3x3. Para obtener la posición de los recuadros, puede investigar que hacen las líneas de código:

```
j, k = (i // 3) * 3, (i % 3) * 3
r = [grid[a][b] for a in range(j, j+3) for b in range(k, k+3)]
```

suponiendo que `grid` es el nombre de nuestra lista bidimensional, cuando `i` toma valores entre 0 y 8.

CAPÍTULO 6

Clase 5: Entrada y salida, decoradores, y errores

6.1 Funciones que aceptan y devuelven funciones (Decoradores)

Vimos que las funciones pueden tomar como argumento otra función, pueden devolver una función, y también pueden hacer las dos cosas simultáneamente.

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)          # ssum es una función
mmin= mas_uno(min)          # mmin es una función
mmax= mas_uno(max)          # mmax 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))
```

```
[0, 1, 3.3, 5, 7.5, 2.2]
19.0 25.0
0 1
7.5 8.5
```

Podemos aplicar la función tanto a funciones “intrínsecas” como a funciones definidas por nosotros

Clases de Python

```
def parabola(v):
    return [x**2 + 2*x for x in v]
```

```
mparabola = mas_uno(parabola)
```

```
print(parabola(a))
print(mparabola(a))
```

```
[0, 3, 17.49, 35, 71.25, 9.240000000000002]
[3, 8, 27.08999999999996, 48, 89.25, 16.64]
```

Notemos que al decorar una función estamos creando una enteramente nueva

```
del parabola # Borramos el objeto
```

```
parabola(a)
```

```
-----
NameError Traceback (most recent call last)
Input In [8], in <module>
----> 1 parabola(a)
```

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

```
mparabola(a)
```

```
[3, 8, 27.08999999999996, 48, 89.25, 16.64]
```

Algunas veces queremos simplemente modificar la función original

6.1.1 Notación para decoradores

Al procedimiento de modificar una función original, para crear una nueva y renombrar la nueva con el nombre de la original se lo conoce como “decorar”, y tiene una notación especial

```
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, no todas necesarias, y podemos simplificarlo:

```
def parabola(v):
    return [x**2 + 2*x for x in v]
parabola = mas_uno(parabola)
```

Como 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)
```

```
[3, 8, 27.089999999999996, 48, 89.25, 16.64]
```

6.1.2 Algunos usos de decoradores

Un uso común de los decoradores es agregar código de verificación de los argumentos de las funciones. Por ejemplo, la función que definimos anteriormente falla si le damos un punto x y queremos obtener el valor y de la parábola (más uno):

```
mi_parabola(3)
```

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

Input In [14], in <module>
----> 1 mi_parabola(3)

Input In [1], in mas_uno.<locals>.fun(args)
 3 def fun(args):
 4     "Agrega 1 a cada uno de los elementos y luego aplica la función"
----> 5     xx = [x+1 for x in args]
 6     y= func(xx)
 7     return y

TypeError: 'int' object is not iterable
```

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 único valor. Podemos definir un decorador para asegurarnos que el tipo es correcto (aunque esta implementación podría ser mejor)

```
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)
```

```
Error: El argumento debe ser una lista
```

```
mi_parabola(a)
```

```
[3, 8, 27.089999999999996, 48, 89.25, 16.64]
```

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]
```

Esto es equivalente a:

```
def parabola(v):
    return [x**2 + 2*x for x in v]
parabola = hace_argumento_list(parabola)
```

```
print(parabola(3))
print(parabola([3]))
```

```
[15]
[15]
```

6.2 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"))
```

```
Input In [2]
    print("hola"))
           ^
SyntaxError: unmatched ')'
```

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)

Input In [3], 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)

Input In [4], 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](#).

6.2.1 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
```

(continué en la próxima página)

Clases de Python

(provine de la página anterior)

```
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']
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)

Input In [6], 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
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.

6.2.2 “Crear” excepciones

Podemos forzar a que nuestro código cree una excepción usando `raise`. Por ejemplo:

```
x = 1
if x > 0:
    raise Exception(f"x = {x}, no debería ser positivo")
```

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

Input In [9], in <module>
  1 x = 1
  2 if x > 0:
----> 3     raise Exception(f"x = {x}, no debería ser positivo")

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(f"x = {x}, no debería ser positivo")
```

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

Input In [10], in <module>
  1 x = 1
  2 if x > 0:
----> 3     raise ValueError(f"x = {x}, no debería ser positivo")

ValueError: x = 1, no debería ser positivo
```

6.3 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
```

```
print(s[:100])
```

```
Aaa
Aaron
Aba
Ababa
Ada
Ada
Adam
Adlai
Adrian
Adrienne
Agatha
Agnetha
Ahmed
Ahmet
Aimee
Al
Ala
Alain
```

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 fi:
    s = fi.read()
print(s[:50])
```

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

```
# fi todavía existe pero está cerrado
fi
```

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

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.

6.3.1 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
Adrian
Adrienne
```

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

fname = '../data/names.txt'
n = 0                                # contador
minlen = 3                             # longitud mínima
maxlen = 3                             # 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, Ala, Alf, Ama, Ami, Amy, Ana, Ann, Art, Bea, Ben, Bib, Bob, Bob,
→Bub, Bud, Cdc, Dad, Dan, Deb, Del, Did, Dod, Don, Dud, Eke, Eli, Ere, Eva, Eve, Eve,
→Ewe, Eye, Fay, Gag, Gay, Gig, Gil, Gog, Guy, Hal, Hon, Hsi, Huh, Hui, Hwa, Ian,
→Iii, Ima, Ira, Jan, Jay, Jef, Jem, Jen, Jim, Jin, Job, Joe, Jon, Jos, Jun, Kaj, Kay,
→Kee, Ken, Kim, Kit, Kyu, Lar, Lea, Lee, Len, Leo, Les, Lex, Lin, Liz, Lou, Luc,
→Lui, Lum, Mac, Mah, Mat, Max, May, Meg, Moe, Mum, Mwa, Nan, Ned, Non, Nou, Nun, Old,
→Ole, Pam, Pap, Pat, Pdp, Pep, Per, Pia, Pim, Pip, Pop, Pup, Raj, Ram, Ray, Reg,
→Rex, Ric, Rik, Rob, Rod, Ron, Roy, S's, Sal, Sam, Sho, Sid, Sir, Sis, Son, Spy, Sri,
→Ssi, Stu, Sue, Sus, Suu, Syd, Tad, Tai, Tal, Tao, Tat, Ted, Tex, The, Tim, Tip,
→Tit, Tnt, Tom, Tor, Tot, Uma, Una, Uri, Urs, Val, Van, Vic, Wes, Win, Wow, Zoe, Zon,
```

(continué en la próxima página)

(proviene de la página anterior)

```
Encontramos 166 palabras que tienen 3 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,  
e(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
```

6.4 Archivos comprimidos

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'xc3xalbaco', b'xc3xalbsida',  
e b'xc3xalbside', b'xc3xalcona', b'xc3xalcaro', b'xc3xalcates', b'xc3xalcido',  
e b'xc3xalcigos']
```

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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()
```

```
print(l[0])
```

b'xc3x81frica'

6.5 Ejercicios 05 (a)

1. 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), ... ]
```

2. 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

CAPÍTULO 7

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, una de ellas es 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. En esta modalidad programamos distintas entidades, donde cada una tiene un comportamiento, y determinamos una manera de interactuar entre ellas.

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 0x7f7eb59560e0>
```

Vemos que `P1` es un objeto del tipo `Punto` que está alojado en una dada dirección de memoria (dada por ese número largo hexadecimal). Para referirnos a los *atributos* de `P1` se utiliza notación “de punto”:

```
P1.x, P1.z
```

```
(0.5, 0)
```

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 y creando un objeto.
- 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: Punto.__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:
```

(continué en la próxima página)

(provine de la página anterior)

```
"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(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)
```

El objeto P1 es del tipo Punto y tiene definidos los métodos `__init__()` (el constructor) y el método `angulo_azimuthal()` que programamos para obtener el ángulo. Además tiene el método `__dict__` que provee un diccionario con los datos del objeto:

```
P1.__dict__
```

```
{'x': 0.5, 'y': 0.5, 'z': 0}
```

Cuando ejecutamos uno de los métodos de un objeto, es equivalente a hacer la llamada al método de la clase, dando como primer argumento el objeto en cuestión:

```
pp = Punto(0.1, "s", [1,2])
```

```
pp
```

```
<__main__.Punto at 0x7f7eb59558d0>
```

```
pp.azimuthal()
```

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

<ipython-input-15-da71cb3c85b2> in <module>
----> 1 pp.azimuthal()

AttributeError: 'Punto' object has no attribute 'azimuthal'
```

```
print(P1.angulo_azimuthal())
print(Punto.angulo_azimuthal(P1))
```

```
0.7853981633974483
0.7853981633974483
```

Al hacer la llamada a un método de una “instancia de la Clase” (o un objeto), 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.

7.2.1 Métodos especiales

Volviendo a mirar la definición de la clase, vemos que `__init__()` es un método “especial”. No necesitamos ejecutarlo explícitamente ya que Python lo hace automáticamente al crear cada objeto de la clase dada. En *Python* el usuario/programador tiene acceso a todos los métodos y atributos. Por convención los nombres que inician con guion bajo se presupone que no son para ser utilizados directamente. En particular, los que están rodeados por dos guiones bajos tienen significado especial y *Python* los va a utilizar en forma automática en distintas ocasiones.

7.3 Herencia

Una de las características de la programación orientada a objetos es la facilidad de 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 existe. 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 0x7f7eb5955060>
```

```
v1.x
```

```
2
```

```
v1.angulo_azimutal()
```

```
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
```

Volviendo a la definición de un objeto, vimos que `__init__()` es un método “especial”. Otro método especial es `__add__()` que nos permite definir la operación suma entre objetos:

```
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 `__add__`, `producto` y `abs`

- `__add__()` 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 con el valor del producto interno
- `abs` toma como argumentos el propio objeto y retorna el número real correspondiente

Su uso será el siguiente:

```
v1 = Vector(1,2,3)
v2 = Vector(3,2,1)
v = v1 + v2
pr = v1.producto(v2)
a = v1.abs()
```

7.5 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"

    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
```

Clases de Python

En este ejemplo estamos creando el objeto `Punto` y en la variable `num_puntos` de la clase estamos llevando la cuenta de cuantos puntos hemos creado. Al crear un nuevo punto (con el método `__init__()`) aumentamos el valor de la variable en uno.

```
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 0x7f10e46183a0> <__main__.Punto object at 0x7f10e461a650>
Número de puntos: 2
```

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
```

```
print(Punto.num_puntos)
```

```
2
```

Nuestra implementación tiene una falla, al borrar los objetos no actualiza el contador, descontando uno cada vez.

```
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(f"En total hay {cls.num_puntos} puntos definidos")
```

En esta versión agregamos un método para actualizar el contador (`borrar()`) y además agregamos un método para imprimir el número de puntos total definidos.

Notar que utilizamos el decorador `@classmethod` antes de la definición, que convierte al método en un método de la clase en lugar de ser un método del objeto (la instancia). Los métodos de clase no reciben como argumento un objeto (como `p1`) sino la clase (`Punto`).

Como en otros casos, el uso del decorador es una conveniencia sintáctica en lugar de llamar a la función intrínseca `classmethod()`.

```
print('Número de puntos:', Punto.num_puntos)
Punto.total()
p1 = Punto(1,1,1)
p2 = Punto()
print(p1, p2)
Punto.total()
```

```
Número de puntos: 0
En total hay 0 puntos definidos
<__main__.Punto object at 0x7f10e4619600> <__main__.Punto object at 0x7f10e4619900>
En total hay 2 puntos definidos
```

```
Punto.total()
p1.borrar()
Punto.total()
```

```
En total hay 2 puntos definidos
En total hay 1 puntos definidos
```

Sin embargo, en esta implementación no estamos realmente removiendo p1, sólo estamos actualizando el contador:

```
print(f"p1 = {p1}")
print(f"p1.x = {p1.x}")
```

```
p1 = <__main__.Punto object at 0x7f10e4619600>
p1.x = 1
```

7.6 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.

7.6.1 Método `__del__()`

Similarmente, existe un método `__del__()` que Python llama automáticamente cuando borramos un objeto.

```
del p1
del p2
```

Podemos utilizar esto para implementar la actualización del contador de puntos

```
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
```

(continué en la próxima página)

(provine de la página anterior)

```
return None

def __del__(self):
    "Borra el punto y actualiza el contador"
    Punto.num_puntos -= 1

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print(f"En total hay {cls.num_puntos} puntos definidos")
```

```
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 0x7f10e46184f0>
```

```
p2
```

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

<ipython-input-16-32960d173fa8> in <module>
----> 1 p2

NameError: name 'p2' is not defined
```

```
print(p1)
```

```
<__main__.Punto object at 0x7f10e46184f0>
```

Como vemos, al borrar el objeto, automáticamente se actualiza el contador.

7.6.2 Métodos `__str__` y `__repr__`

El método `__str__` también 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 0x7f10e4618640>
```

Rehagamos la clase para definir puntos

```
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 = f"Punto en el espacio con coordenadas: x = {self.x}, y = {self.y}, z = {self.z}"
        return s

    @classmethod
    def total(cls):
        "Imprime el número total de puntos"
        print(f"En total hay {cls.num_puntos} puntos definidos")
```

```
p1 = Punto(1,1,0)
```

```
print(p1)
```

```
Punto en el espacio con coordenadas: x = 1, y = 1, z = 0
```

```
ss = 'p1 = {}'.format(p1)
ss
```

```
'p1 = Punto en el espacio con coordenadas: x = 1, y = 1, z = 0'
```

```
p1
```

```
<__main__.Punto at 0x7f10e461b190>
```

Como vemos, si no usamos la función `print()` o `format()` sigue mostrándonos el objeto (que no es muy informativo). Esto puede remediarlo 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"
```

(continúe en la próxima página)

(provine 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

def __del__(self):
    "Borra el punto y actualiza el contador"
    Punto.num_puntos -= 1

def __str__(self):
    return f"Punto en el espacio con coordenadas: x = {self.x}, y = {self.y}, z = {self.z}"

def __repr__(self):
    return f"Punto(x = {self.x}, y = {self.y}, z = {self.z})"

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print(f"En total hay {cls.num_puntos} puntos definidos")
```

```
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.

7.6.3 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()
```

(continué en la próxima página)

(provien de la página anterior)

```
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
        Punto.num_puntos += 1
        return None

    def __del__(self):
        "Borra el punto y actualiza el contador"
        Punto.num_puntos -= 1

    def __str__(self):
        return f"Punto en el espacio con coordenadas: x = {self.x}, y = {self.y}, z = {self.z}"

    def __repr__(self):
        return f"Punto(x = {self.x}, y = {self.y}, z = {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(f"En total hay {cls.num_puntos} puntos definidos")
```

```
p3 = Punto(1, 3, 4)
p3
```

```
Punto(x = 1, y = 3, z = 4)
```

```
p3()
```

```
'Ejecuté el objeto: Punto en el espacio con coordenadas: x = 1, y = 3, z = 4'
```

7.6.4 Métodos `__add__`, `__mul__`

Además del método `__add__()` visto anteriormente, que es llamado automáticamente cuando se utiliza la operación suma, existe el método `__mul__()` que se ejecuta al utilizar la operación multiplicación.

7.7 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):
        "Borra el punto y actualiza el contador"
        Punto.num_puntos -= 1

    def __str__(self):
        return f"Punto en el espacio con coordenadas: x = {self.x}, y = {self.y}, z = {self.z}"

    def __repr__(self):
        return f"Punto(x = {self.x}, y = {self.y}, z = {self.z})"

    def __call__(self):
        return self.__str__()

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print(f"En total hay {cls.num_puntos} puntos definidos")
```

Complete la implementación de la clase Vector con los métodos pedidos

```
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 __mul__(self, v2):
        "Calcula el producto interno entre dos vectores"
```

(continúe en la próxima página)

(provine de la página anterior)

```

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
```

- Un método `from_string` (**si puede**) 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>>  
>>> 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
```

- Escriba los métodos llamados `__mul__(self, value)` y `__rmul__(self, value)`, que devuelvan el producto de un polinomio por un valor constante, llamando al método `mul` definido anteriormente. Eso permitirá usar la operación producto en la forma:

```
>>> p1 = Polinomio([1,2,3,4])  
>>> k = 3.5  
>>> p1 * k  
>>> k * p1
```


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 Narębski.
-

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 («undo ilimitado»)
- Trabajar en forma paralela con otras personas sobre un mismo documento.

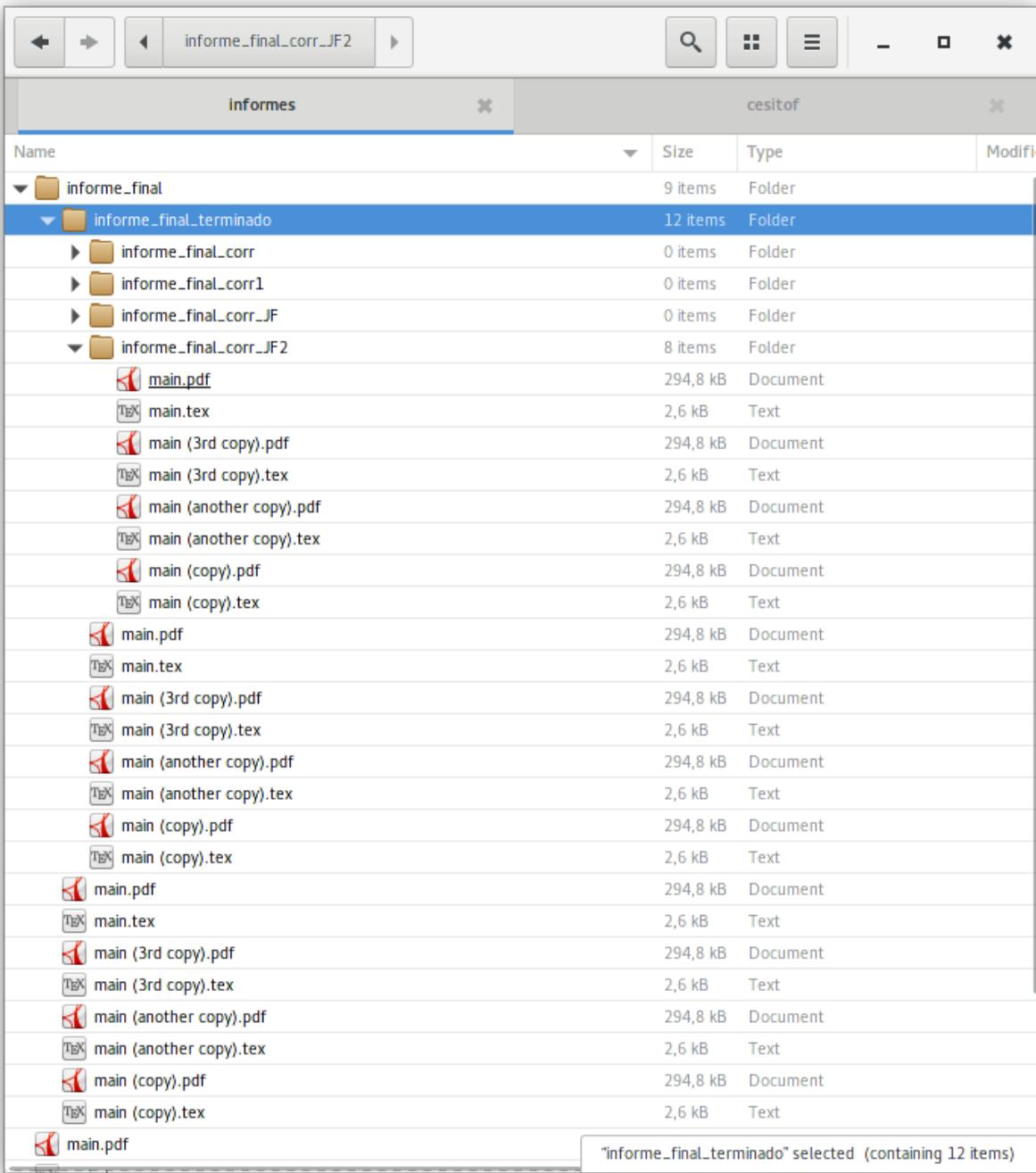
Habitualmente, nos podemos encontrar con situaciones como esta:

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



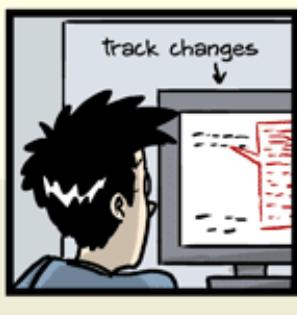
FINAL_rev.2.doc



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

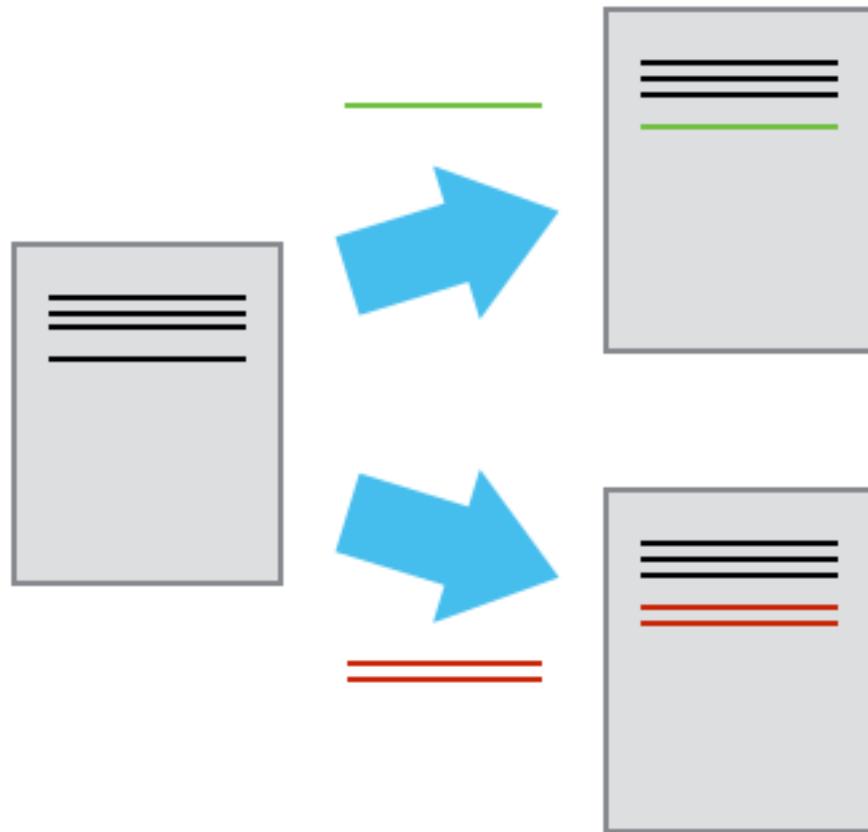
WWW.PHDCOMICS.COM

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

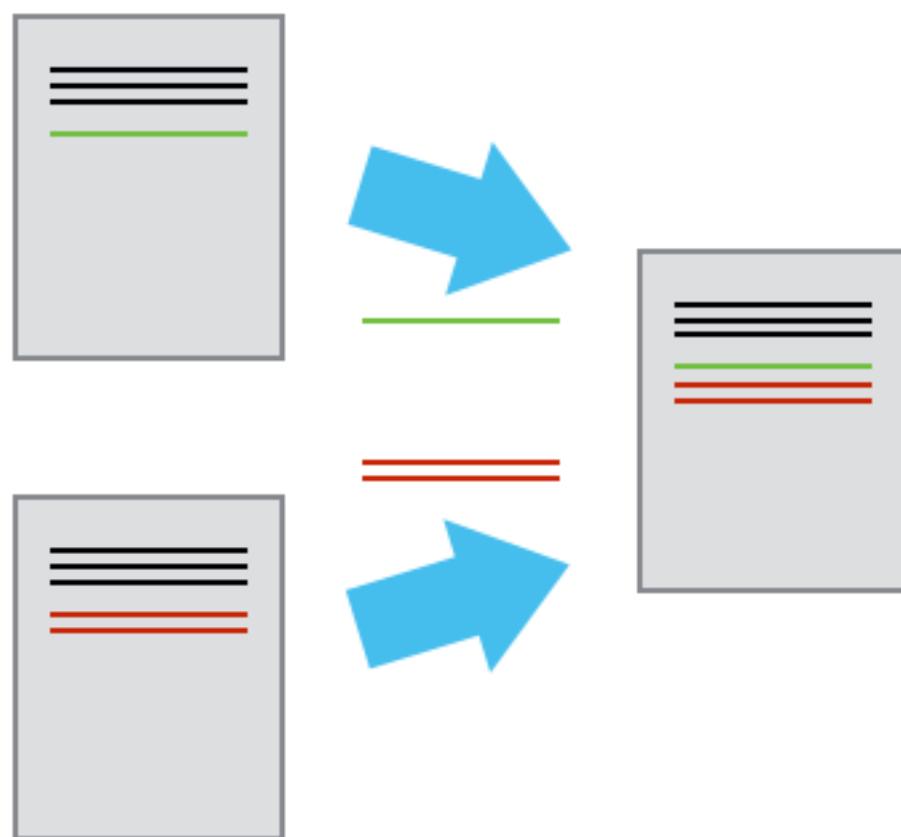
8.1.1 Cambios en paralelo

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 help  
$ git --help
```

que nos da información sobre su uso, y cuáles son los comandos disponibles. Si queremos obtener información sobre un comando en particular, agregamos el comando de interés. Para el comando de configuración sería:

```
$ git config --help
$ git help config
```

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 de 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://github.com/fiolj/intro-python-IB.git
$ git clone https://github.com/fiolj/intro-python-IB.git clase-ib
```

Los dos primeros ejemplos realizan una copia de trabajo de un proyecto alojado también localmente. En el segundo y cuarto casos les estamos dando un nuevo nombre a la copia local 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 «commit»

```
$ 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 «undo» 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/Clases/IntPython/clases-python/clases',
 '/usr/lib64/python310.zip',
 '/usr/lib64/python3.10',
 '/usr/lib64/python3.10/lib-dynload',
 '',
 '/home/fiol/.local/lib/python3.10/site-packages',
 '/usr/lib64/python3.10/site-packages',
 '/usr/lib/python3.10/site-packages',
 '/usr/lib/python3.10/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/Clases/IntPython/clases-python/clases
```

```
cur = os.getcwd()  
par = os.path.abspath("..")  
print(cur)  
print(par)
```

```
/home/fiol/Clases/IntPython/clases-python/clases  
/home/fiol/Clases/IntPython/clases-python
```

```
print(os.path.abspath(os.curdir))  
print(os.getcwd())
```

```
/home/fiol/Clases/IntPython/clases-python/clases  
/home/fiol/Clases/IntPython/clases-python/clases
```

```
print(os.path.basename(cur))
print(os.path.splitdrive(cur))
```

```
clases
('', '/home/fiol/Clases/IntPython/clases-pyton/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/Clases/IntPython/clases-pyton
/home/fiol/Clases/IntPython/clases-pyton/este/otro.dat
('/home/fiol/Clases/IntPython/clases-pyton/este', 'otro.dat')
('/home/fiol/Clases/IntPython/clases-pyton/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/Clases/IntPython/clases-pyton/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/Clases/IntPython/clases-pyton
/home/fiol/Clases/IntPython/clases-pyton/clases
/home/fiol/Clases/IntPython/clases-pyton/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)
```

Clases de Python

```
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.
```

```
ge('HOME')
```

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

```
path.realpath(curdir)
```

```
'/home/fiol/Clases/IntPython/clases-pyton/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

(continué en la próxima página)

(provine de la página anterior)

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.

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 16900.9248046875 kbytes en 91 non-directory files
./09_intro_visualizacion_files consume 340.0712890625 kbytes en 23 non-directory
→files
./09_mas_numpy_matplotlib_files consume 87.013671875 kbytes en 5 non-directory files
./10_entrada_salida_files consume 210.708984375 kbytes en 13 non-directory files
./10_mas_numpy_files consume 120.490234375 kbytes en 6 non-directory files
./11_intro_scipy_files consume 77.0185546875 kbytes en 4 non-directory files
./12_fiteos_files consume 1325.1513671875 kbytes en 46 non-directory files
./13_graficacion3d_files consume 818.3916015625 kbytes en 25 non-directory files
./13_misclaneas_files consume 414.314453125 kbytes en 10 non-directory files
./14_fft_files consume 1186.970703125 kbytes en 27 non-directory files
./14_interactivo_files consume 271.9951171875 kbytes en 6 non-directory files
./explicacion_ejercicio_agujas_files consume 8.73828125 kbytes en 2 non-directory
→files
./scripts consume 757.1904296875 kbytes en 52 non-directory files
./scripts/animaciones consume 370.13671875 kbytes en 11 non-directory files
./scripts/interfacing consume 1062.0107421875 kbytes en 22 non-directory files
./scripts/interfacing/__pycache__ consume 1.69140625 kbytes en 2 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
```

(continúe en la próxima página)

(provine de la página anterior)

```
./version-control/_static/css consume 116.8828125 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
./version-control/_static/js consume 19.341796875 kbytes en 2 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 43425 feb 14 15:32 04_1_funciones.ipynb
-rw-r--r--. 1 fiol fiol 34790 feb 14 16:42 04_2_func_args.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_1_funciones.ipynb', '04_2_func_args.ipynb']
```

```
nb_clase4.sort()
```

```
nb_clase4
```

```
['04_1_funciones.ipynb', '04_2_func_args.ipynb']
```

```
nb_clases1a4 = glob.glob('0[0-4]*.ipynb')
```

```
nb_clases1a4
```

```
['00_introd_y_excursion.ipynb',
 '02_2_listas.ipynb',
 '02_1_tipos_y_control.ipynb',
 '03_1_tipos_control.ipynb',
 '03_2_iteraciones_tipos.ipynb',
 '01_1_instala_y_uso.ipynb',
 '01_2_introd_python.ipynb',
 '04_1_funciones.ipynb',
 '04_2_func_args.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_1_instala_y_uso.ipynb
Clase en archivo 01_2_introd_python.ipynb
Clase en archivo 02_1_tipos_y_control.ipynb
Clase en archivo 02_2_listas.ipynb
Clase en archivo 03_1_tipos_control.ipynb
Clase en archivo 03_2_iteraciones_tipos.ipynb
Clase en archivo 04_1_funciones.ipynb
Clase en archivo 04_2_func_args.ipynb
```

8.3.5 Módulo pathlib

El módulo `pathlib` es “relativamente nuevo” y tiene funcionalidades para trabajar con rutas de archivos y directorios con una tratamiento de programación orientada a objetos. Este módulo define un objeto `Path` que contiene mucha de la funcionalidad que usualmente se obtenía sólo de los módulos `os` y `glob`. Veamos algunos ejemplos simples de su uso

```
from pathlib import Path
```

```
direct = Path('.')
print(direct)
```

```
.
```

El objeto tiene un iterador que nos permite recorrer todo el directorio. Por ejemplo si queremos listar todos los subdirectorios:

```
[x for x in direct.iterdir() if x.is_dir()]
```

```
[PosixPath('.ipynb_checkpoints'),
 PosixPath('09_intro_visualizacion_files'),
 PosixPath('09_mas_numpy_matplotlib_files'),
 PosixPath('10_entrada_salida_files'),
 PosixPath('10_mas_numpy_files'),
 PosixPath('11_intro_scipy_files'),
 PosixPath('12_fiteos_files'),
 PosixPath('13_graficacion3d_files'),
 PosixPath('13_miscelaneas_files'),
 PosixPath('14_fft_files'),
 PosixPath('14_interactivo_files'),
 PosixPath('explicacion_ejercicio_agujas_files'),
 PosixPath('scripts'),
 PosixPath('version-control'),
 PosixPath('figuras')]
```

Trabajo con rutas de archivos

```
print(direct.absolute())
```

```
/home/fiol/Clases/IntPython/clases-python/clases
```

```
p = direct / ".."
print(p)
print(p.resolve())
```

```
..
/home/fiol/Clases/IntPython/clases-python
```

Podemos reemplazar el módulo glob utilizando este objeto:

```
for fi in sorted(direct.glob("0[1-7]*.ipynb")) :
    print(fi)
```

```
01_1_instala_y_uso.ipynb
01_2_introd_python.ipynb
02_1_tipos_y_control.ipynb
02_2_listas.ipynb
03_1_tipos_control.ipynb
03_2_iteraciones_tipos.ipynb
04_1_funciones.ipynb
04_2_func_args.ipynb
05_1_decoradores.ipynb
05_2_excepciones.ipynb
05_3_inout.ipynb
06_1_objetos.ipynb
06_2_objetos.ipynb
07_ejemplo_oop.ipynb
07_modulos_biblioteca.ipynb
```

```
fi = direct / "programa_detalle.rst"
if fi.exists():
    s= fi.read_text()
print(s)
```

```
.. _prog-detalle:

Programa Detallado
=====

:Autor: Juan Fiol
:Version: Revision: 2022
:Copyright: Libre
```

8.3.6 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](#)

8.3.7 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:{}\n  "{}"'.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 “[” y “]” 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 “.”, “*”, “.”).
- 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!
```

(continúe en la próxima página)

(provine de la página anterior)

```
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
  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)  
print(type(mm[0]))  
for m in mm:  
    print('Encontré {}'.format(m))
```

```
['ab', 'ab']  
<class 'str'>  
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

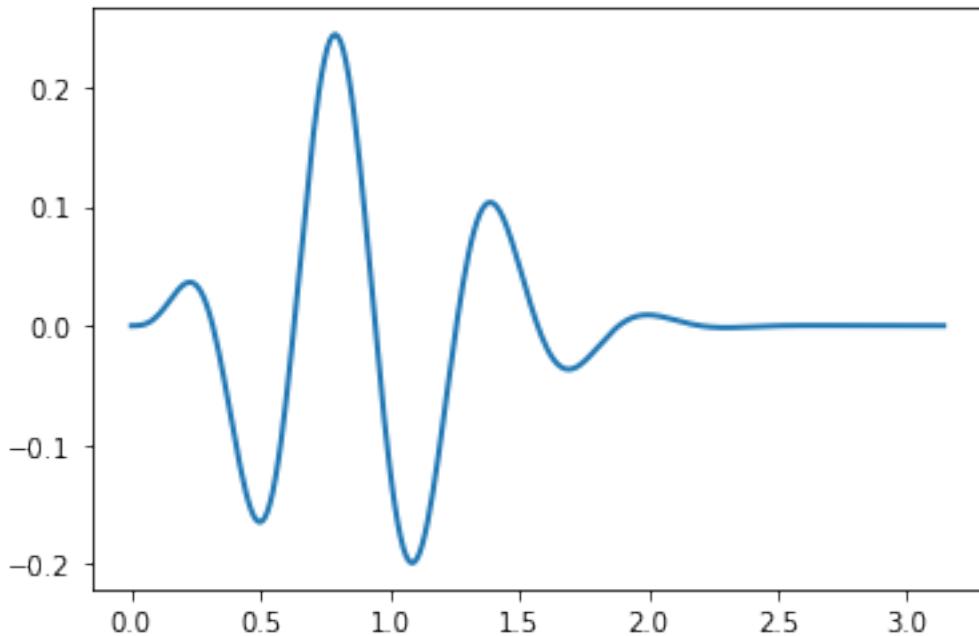
Dos paquetes que van a resultar muy importantes para nosotros son los paquetes **numpy** y **matplotlib**. Como con todos los módulos, se cargan utilizando la palabra `import`, tal como hicimos en los ejemplos anteriores. Existen variantes en la manera de importar los módulos que son “equivalentes”. En este caso le vamos a dar un alias que sea más corto de tippear. 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 0x7f8c3925e710>]
```



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])
```

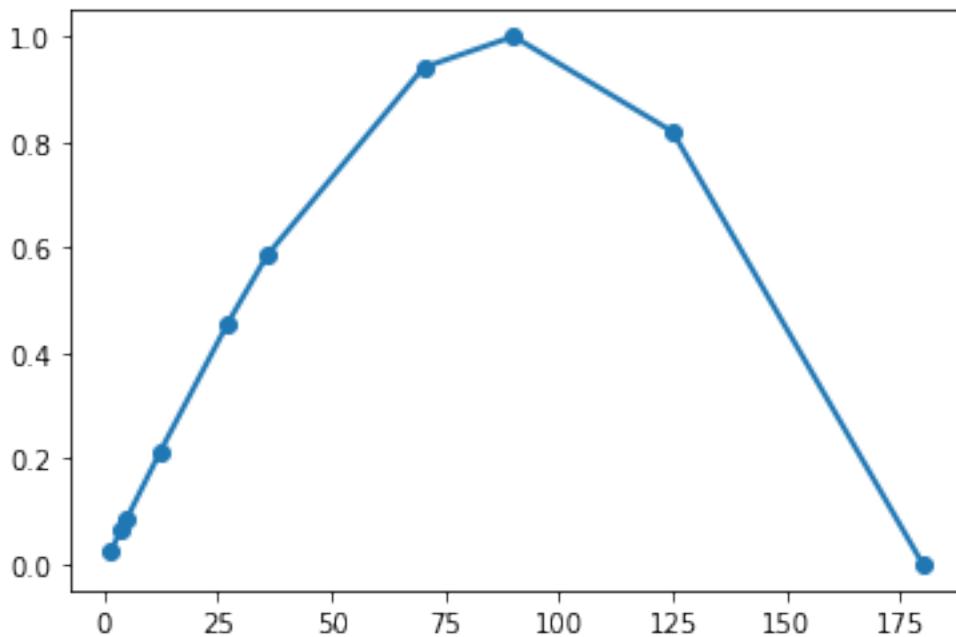
```
[0.02617694830787315, 0.06627390040000014, 0.08541692313736747, 0.21303038627497659, 0.4570979270586942, 0.5849576749872154, 0.9408807689542255, 1.0, 0.819152044288992, 1.2246467991473532e-16]
```

```
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
```

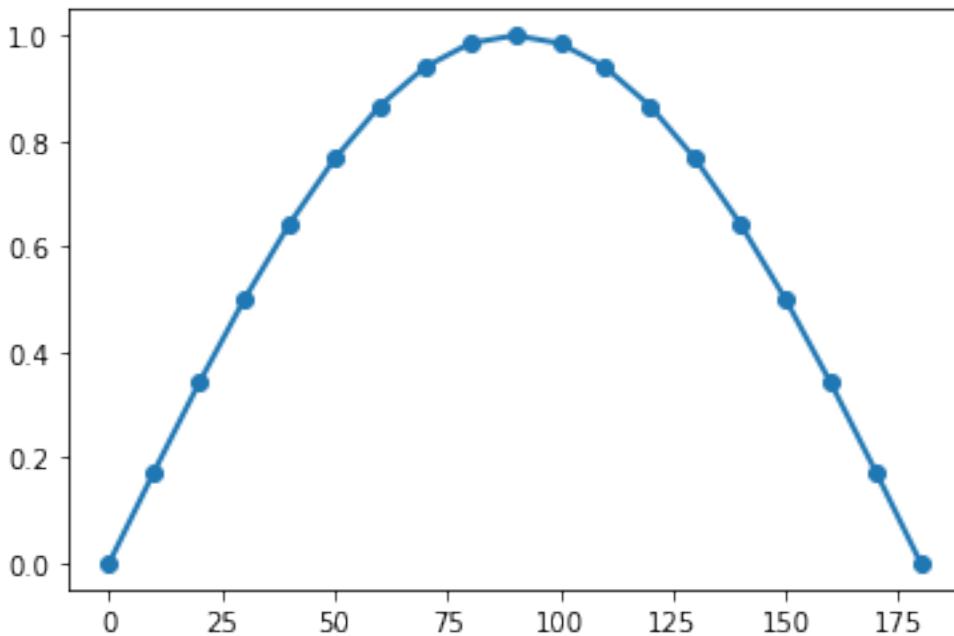
```
array([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90., 100.,  
       110., 120., 130., 140., 150., 160., 170., 180.])
```

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()
```



```
np.arange(0,180.,10, dtype=int)
```

```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120,
       130, 140, 150, 160, 170])
```

```
# 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)
```

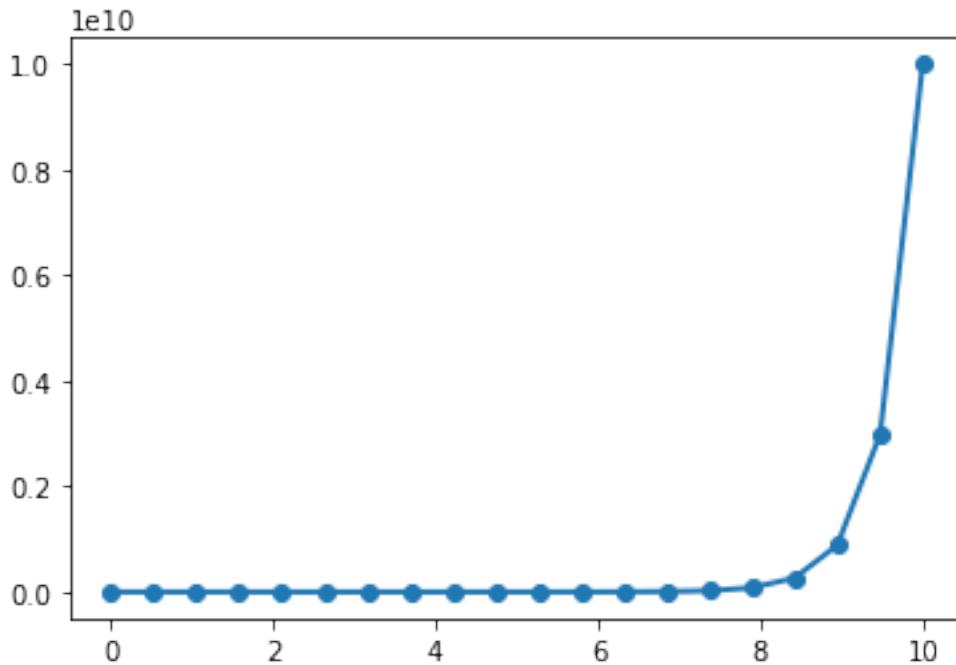
```
print(step1)
print(step2)
```

```
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:

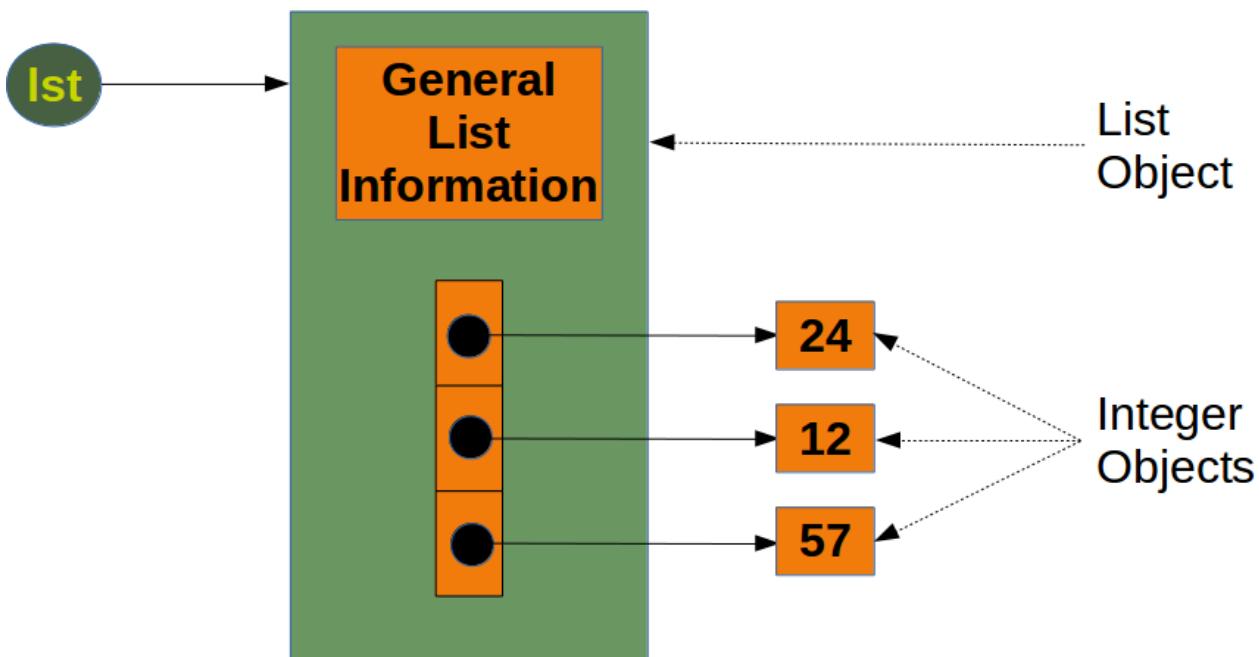


Figura 1: Representación en memoria de una lista

```
from sys import getsizeof
lst = [24, 12, 57]
size_of_list_object = getsizeof(lst)      # La lista sin sus datos
#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: 120
Tamaño de los elementos: 84
Tamaño total: 204
```

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('Una lista con 3 elems: {} bytes'.format(getsizeof([24,12,57])))
print('Un entero en Python : {} bytes'.format(getsizeof(24)))
```

```
Una lista vacía ocupa: 56 bytes
Una lista con un elem: 64 bytes
Una lista con 2 elems: 72 bytes
Una lista con 3 elems: 120 bytes
Un entero en Python : 28 bytes
```

Vemos que la “Información general de listas” ocupa **56 bytes**, y la referencia a cada elemento entero ocupa adicionalmente **8 bytes**, por lo que la lista ocupa **120 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:

$$M_L(n) = 56 + n \times 8 + n \times 28$$

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)

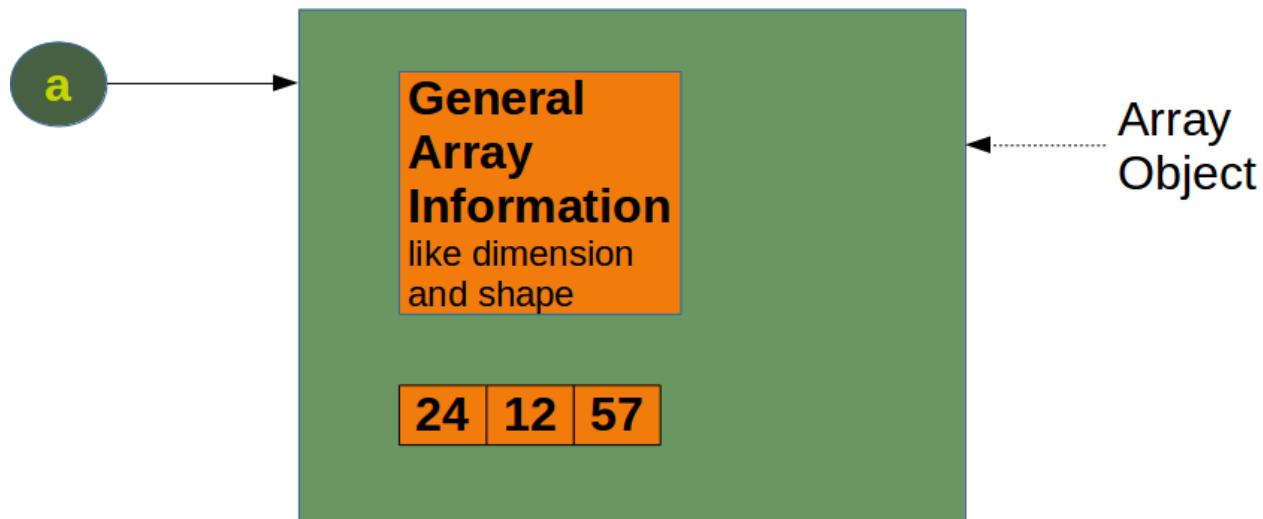


Figura 2: Representación en memoria de una lista

```
a = np.array(lst)
print(getsizeof(a))
```

```
128
```

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])))
print('Un entero de Numpy usa: {}'.format(getsizeof(a[0])))
```

```
Un array vacío ocupa: 104 bytes
Un array con un elem: 112 bytes
Un array con 2 elems: 120 bytes
Un entero de Numpy es: <class 'numpy.float64'>
Un entero de Numpy es: 32
```

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))
```

(continué en la próxima página)

(proviene de la página anterior)

```
total_list_size = getssizeof(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: ", getssizeof(a1))
```

```
Tamaño total de la lista: 36052
Tamaño total de array: 8104
```

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(f"Numpy es en este ejemplo {t1 / t2:.3f} más rápido")
```

```
Numpy es en este ejemplo 41.016 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)      # Algo similar pero inicializando a unos
```

```
array([1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.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))
```

Clases de Python

```
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 pares : ', arr[1,1::2])
```

```
Segunda fila, todos los elementos pares : [12 14 16 18 20]
```

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	32
33	34	35	36	37	38	39	40	41	42	43
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:
 1. Un array con valores 1 en la “diagonal principal” y 0 en el resto (Matriz identidad).
 2. Un array con valores 0 en la “diagonal principal” y 1 en el resto.
 3. Un array con valores 1 en los bordes y 0 en el interior.
 4. 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.

```
import numpy as np
```

```
arr = np.arange(55).reshape((5, 11))+1
```

```
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          : 1 55
Suma y producto de sus elementos : 1540 6711489344688881664
Media y desviación standard   : 28.0 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:
[[ 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 55]]
```

```
print( 'La suma de todos los elementos es      :', arr.sum())
```

```
La suma de todos los elementos es      : 1540
```

```
print( 'La suma de elementos de las filas es :', arr.sum(axis=1))
```

```
La suma de elementos de las filas es : [ 66 187 308 429 550]
```

```
print( 'La suma de elementos de las columnas es :', arr.sum(axis=0))
```

```
La suma de elementos de las columnas es : [115 120 125 130 135 140 145 150 155 160
                                         ↪165]
```

```
print(arr.min(axis=0))
print(arr.min(axis=1))
```

```
[ 1  2  3  4  5  6  7  8  9 10 11]
 [ 1 12 23 34 45]
```

9.7 Operaciones sobre arrays

9.7.1 Operaciones básicas

Los array se pueden usar en operaciones:

```
arr + 1
```

```
array([[ 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, 55, 56]])
```

```
# Suma de una constante
arr1 = 1 + arr[:,::-1]           # Creamos un segundo array
```

```
arr1
```

```
array([[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2],
       [23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13],
       [34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24],
       [45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35],
       [56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46]])
```

```
# Multiplicación por constantes y suma de arrays
-2 * arr + 3 * arr1
```

```
array([[ 34,  29,  24,  19,  14,   9,   4,  -1,  -6, -11, -16],
       [ 45,  40,  35,  30,  25,  20,  15,  10,   5,   0,  -5],
       [ 56,  51,  46,  41,  36,  31,  26,  21,  16,  11,   6],
       [ 67,  62,  57,  52,  47,  42,  37,  32,  27,  22,  17],
       [ 78,  73,  68,  63,  58,  53,  48,  43,  38,  33,  28]])
```

```
# División por constantes
arr / 5
```

```
array([[ 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, 11. ]])
```

```
# Multiplicación entre arrays
arr * arr1
```

```
array([[ 12,   22,   30,   36,   40,   42,   42,   40,   36,   30,   22],
       [ 276,  286,  294,  300,  304,  306,  306,  304,  300,  294,  286],
       [ 782,  792,  800,  806,  810,  812,  812,  810,  806,  800,  792],
       [1530, 1540, 1548, 1554, 1558, 1560, 1560, 1558, 1554, 1548, 1540],
       [2520, 2530, 2538, 2544, 2548, 2550, 2550, 2548, 2544, 2538, 2530]])
```

```
arr / arr1
```

```
array([[0.08333333, 0.18181818, 0.3           , 0.44444444, 0.625           ,
       0.85714286, 1.16666667, 1.6           , 2.25           , 3.33333333,
       5.5           ],
      [0.52173913, 0.59090909, 0.66666667, 0.75           , 0.84210526,
       0.94444444, 1.05882353, 1.1875         , 1.33333333, 1.5           ,
       1.69230769],
      [0.67647059, 0.72727273, 0.78125        , 0.83870968, 0.9           ,
       0.96551724, 1.03571429, 1.11111111, 1.19230769, 1.28           ,
       1.375         ],
      [0.75555556, 0.79545455, 0.8372093        , 0.88095238, 0.92682927,
       0.975         , 1.02564103, 1.07894737, 1.13513514, 1.19444444,
       1.25714286],
      [0.80357143, 0.83636364, 0.87037037        , 0.90566038, 0.94230769,
       0.98039216, 1.02           , 1.06122449, 1.10416667, 1.14893617,
       1.19565217]])
```

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 True]
```

```
# Comparación de un array con otro
print(v > w)
```

```
[False False False False False False 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([[-0.53657292, -0.99999021, -0.54402111,  0.41211849,  0.98935825,
       0.6569866 , -0.2794155 , -0.95892427, -0.7568025 ,  0.14112001,
       0.90929743],
      [-0.8462204 , -0.00885131,  0.83665564,  0.91294525,  0.14987721,
       -0.75098725, -0.96139749, -0.28790332,  0.65028784,  0.99060736,
       0.42016704],
      [ 0.52908269,  0.99991186,  0.55142668, -0.40403765, -0.98803162,
       -0.66363388,  0.27090579,  0.95637593,  0.76255845, -0.13235175,
       -0.90557836],
      [ 0.85090352,  0.01770193, -0.83177474, -0.91652155, -0.15862267,
       0.74511316,  0.96379539,  0.29636858, -0.64353813, -0.99177885,
       -0.42818267],
      [-0.521551 , -0.99975517, -0.55878905,  0.39592515,  0.98662759,
       0.67022918, -0.26237485, -0.95375265, -0.76825466,  0.12357312,
       0.90178835]])
```

```
np.exp(-arr**2/2)
```

```
array([[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, 3.36244047e-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, 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(w), 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 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
```

```
% x      y
% z      z2
1.00e+00 2.00e+00 3.00e+00 4.00e+00 5.00e+00 6.00e+00 7.00e+00 8.00e+00 9.00e+00 1.
↪00e+01 1.10e+01
1.20e+01 1.30e+01 1.40e+01 1.50e+01 1.60e+01 1.70e+01 1.80e+01 1.90e+01 2.00e+01 2.
↪10e+01 2.20e+01
2.30e+01 2.40e+01 2.50e+01 2.60e+01 2.70e+01 2.80e+01 2.90e+01 3.00e+01 3.10e+01 3.
↪20e+01 3.30e+01
3.40e+01 3.50e+01 3.60e+01 3.70e+01 3.80e+01 3.90e+01 4.00e+01 4.10e+01 4.20e+01 4.
↪30e+01 4.40e+01
4.50e+01 4.60e+01 4.70e+01 4.80e+01 4.90e+01 5.00e+01 5.10e+01 5.20e+01 5.30e+01 5.
↪40e+01 5.50e+01
```

```
arr2 = np.loadtxt('test.out', comments="% ")
print(arr2)
```

```
[[ 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. 55.]]
```

```
print(arr2.shape)
print(arr2.ndim)
print(arr2.size)
```

```
(5, 11)
2
55
```

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:

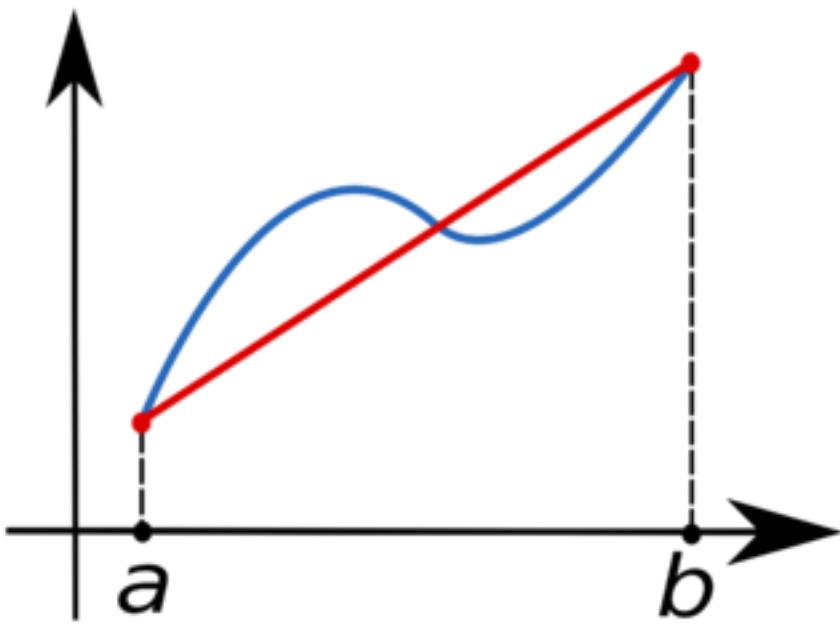
- `-v vel` o, equivalentemente `--velocidad=vel`, donde `vel` es el número dando la velocidad inicial en m/s. El valor por defecto será 0.
- `-a alt` o, equivalentemente `--altura=alt`, donde `alt` es un número dando la altura inicial en metros. El valor por defecto será 1000. La altura inicial debe 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` o, 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` o, 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 **08_Su apellido.py** en un adjunto por correo electrónico, con asunto: **08_Su apellido**, antes del día miércoles 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 'trapzf' 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: una interfaz con el usuario. Un ejemplo es el submódulo **pyplot**.
- Una tercera componente 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])
```

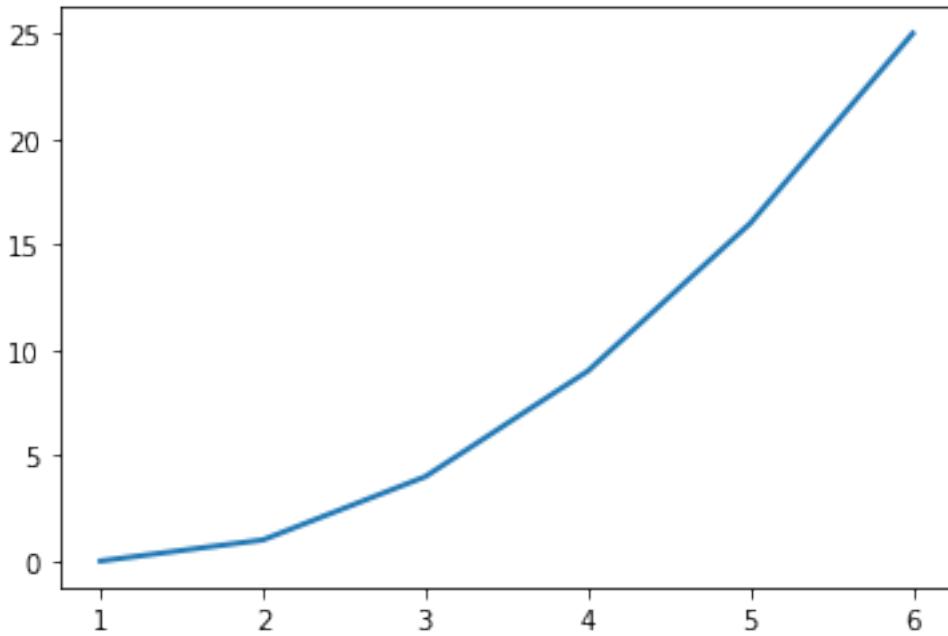
En el modo interactivo no sólo `plot()` tiene implícito el comando `show()` sino que podemos seguir ingresando comandos con el gráfico abierto.

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:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4,5,6],[0,1,4,9,16,25])
```

```
[<matplotlib.lines.Line2D at 0x7f656e0570d0>]
```



En la práctica vamos a usar siempre **Matplotlib** junto con **Numpy**.

```
import numpy as np
import matplotlib.pyplot as plt
```

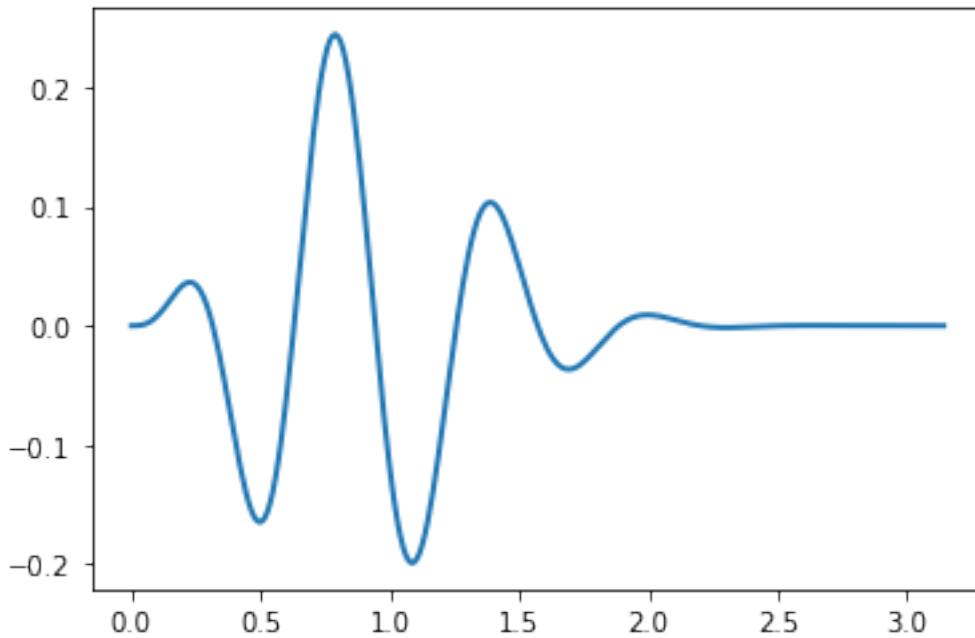
10.2 Gráficos simples

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 0x7f6524c57130>]
```



Como vemos, es la curva que graficamos la clase anterior.

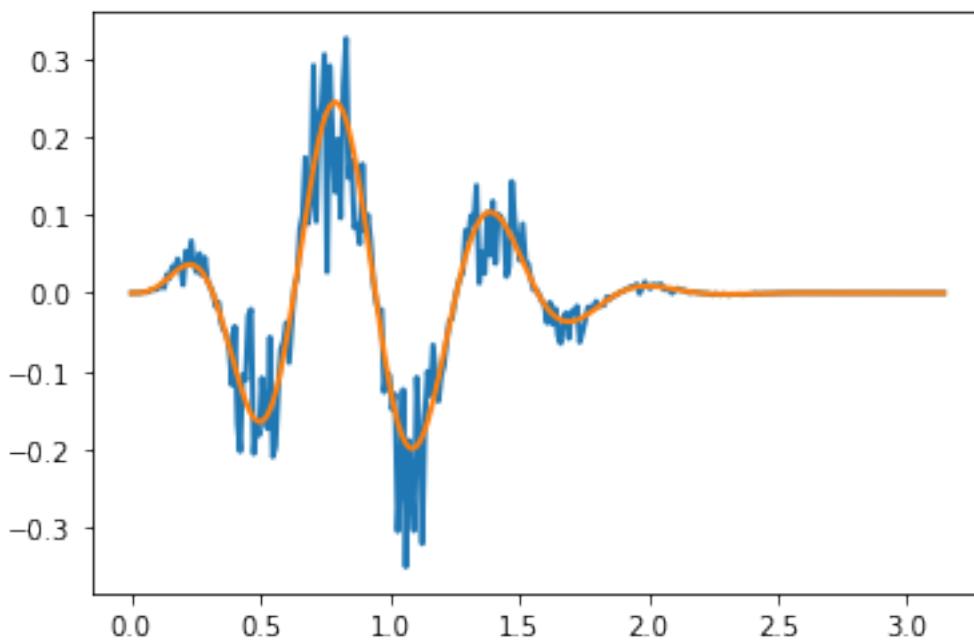
Si miramos el archivo vamos a ver que tiene una columna más que corresponde a los valores medidos. En consecuencia le asignamos esta tercera columna a una variable adicional `yexp` al leerlo.

```
!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 columnas como función de la primera
plt.plot(x,yexp, x,y)
```

```
[<matplotlib.lines.Line2D at 0x7f6524ad1f60>,
 <matplotlib.lines.Line2D at 0x7f6524ad1f90>]
```



10.3 Formato de las curvas

En los gráficos anteriores usamos la función `plot()` en sus formas más simples.

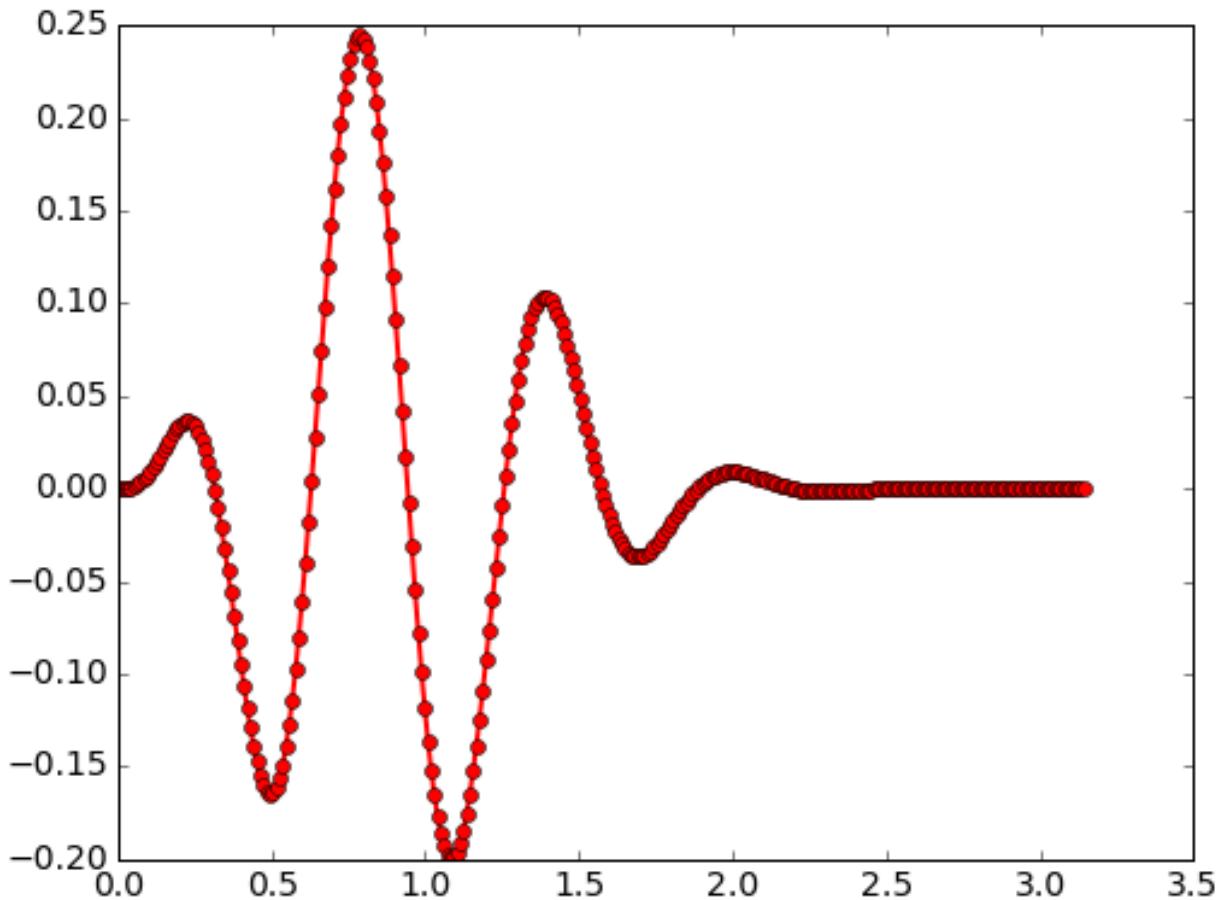
```
plot(y)
plot(x,y)
plot(x1,y1, x2, y2)
```

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')
```

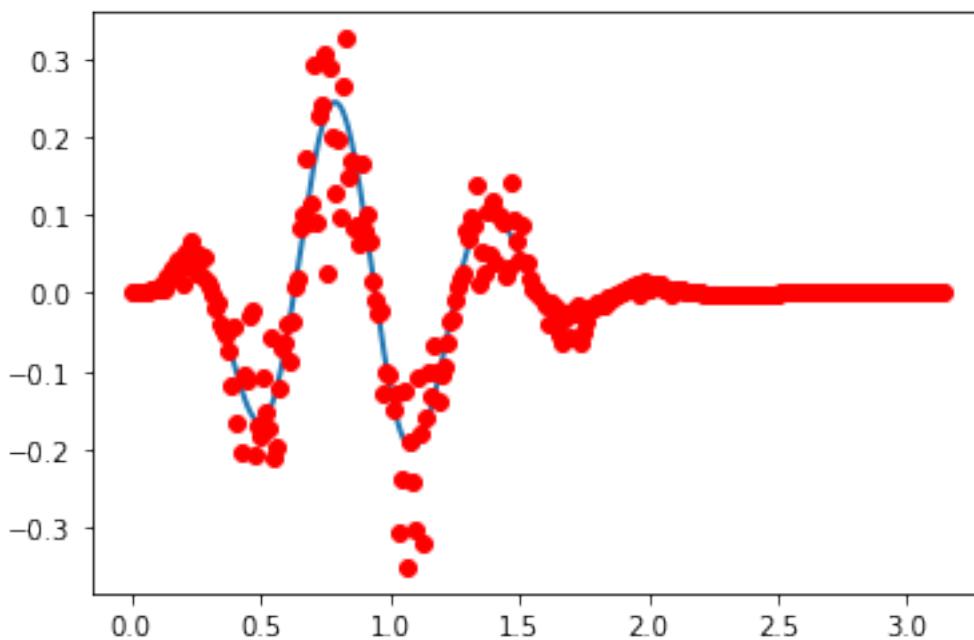


Para obtener círculos usamos una especificación que corresponde a ‘o’. Además podemos poner en este argumento el color. En este caso elegimos graficar en color “rojo (r), con una línea (-) + círculos (o)”.

Con esta idea modificamos el gráfico anterior

```
plt.plot(x,y, '-o', x,yexp, 'ro')
```

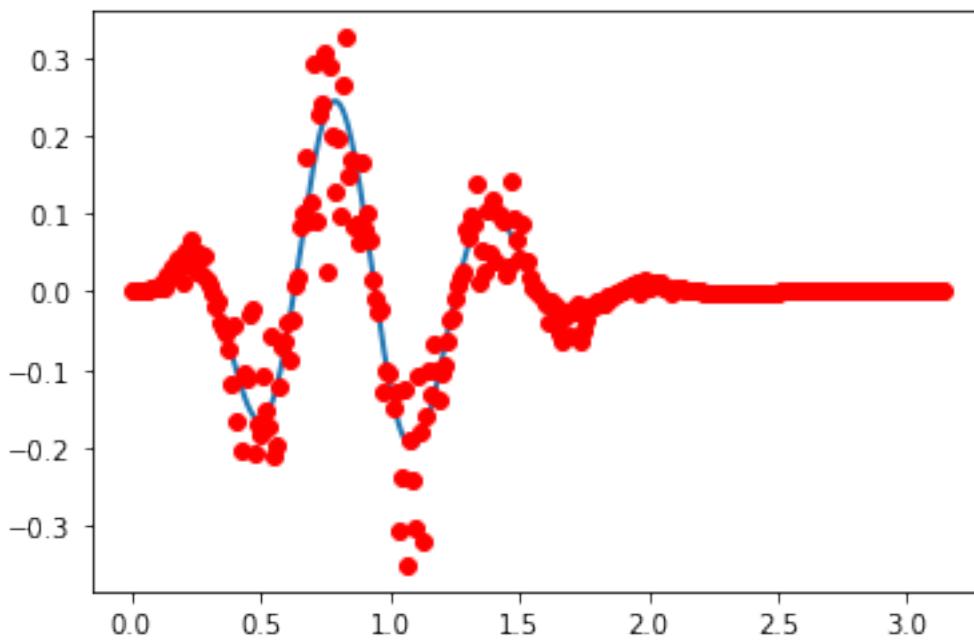
```
[<matplotlib.lines.Line2D at 0x7f6524b49060>,
 <matplotlib.lines.Line2D at 0x7f6524b49180>]
```



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 0x7f65249c4820>]
```



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
'l'	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::

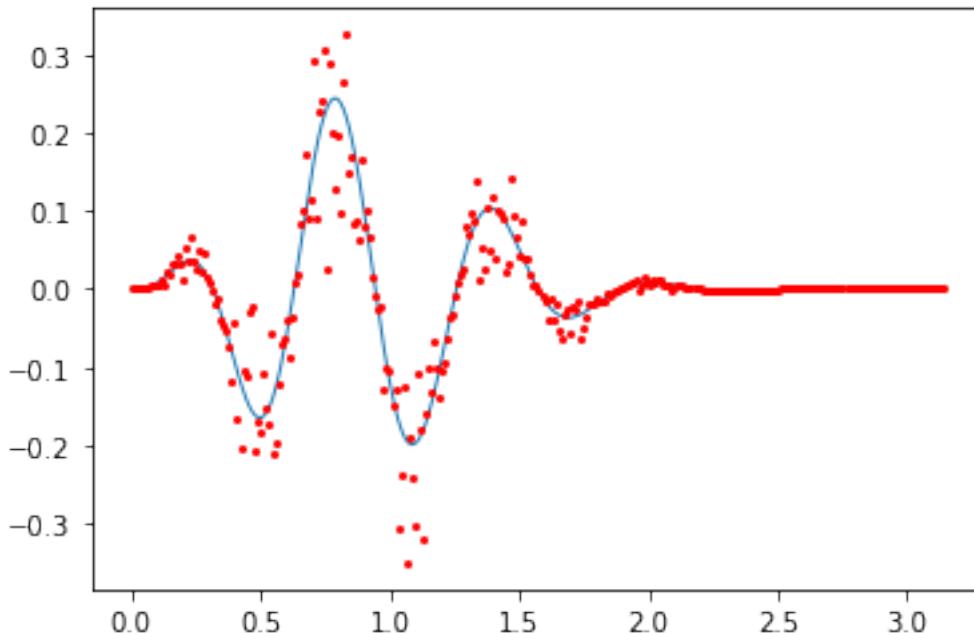
```
plot([x], y, [fmt], data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

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

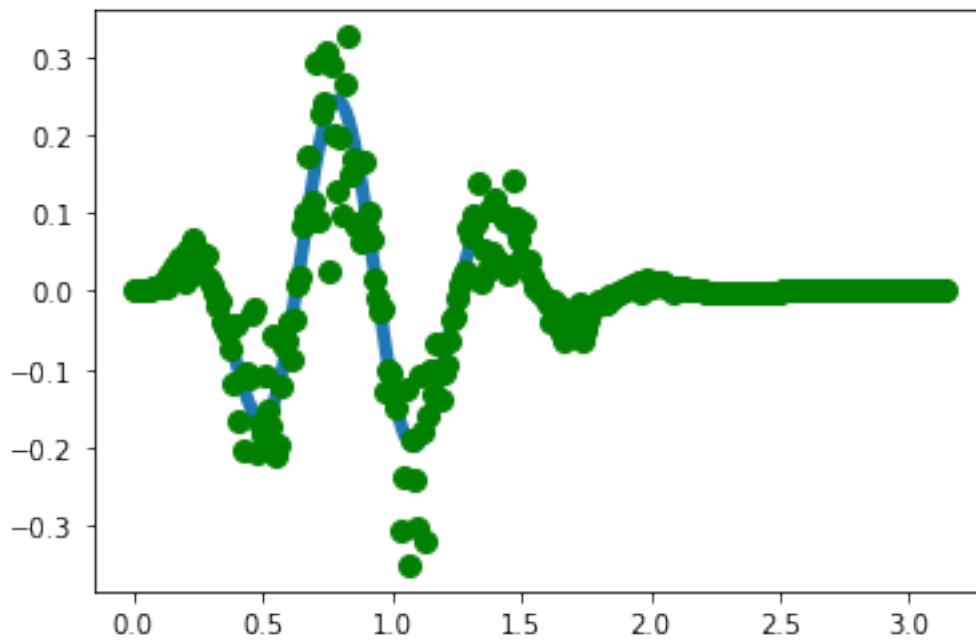
```
plt.plot(x,y,linewidth=1)
plt.plot(x,yexp, 'o', color='red', markersize=2)
```

```
[<matplotlib.lines.Line2D at 0x7f6524a0b970>]
```



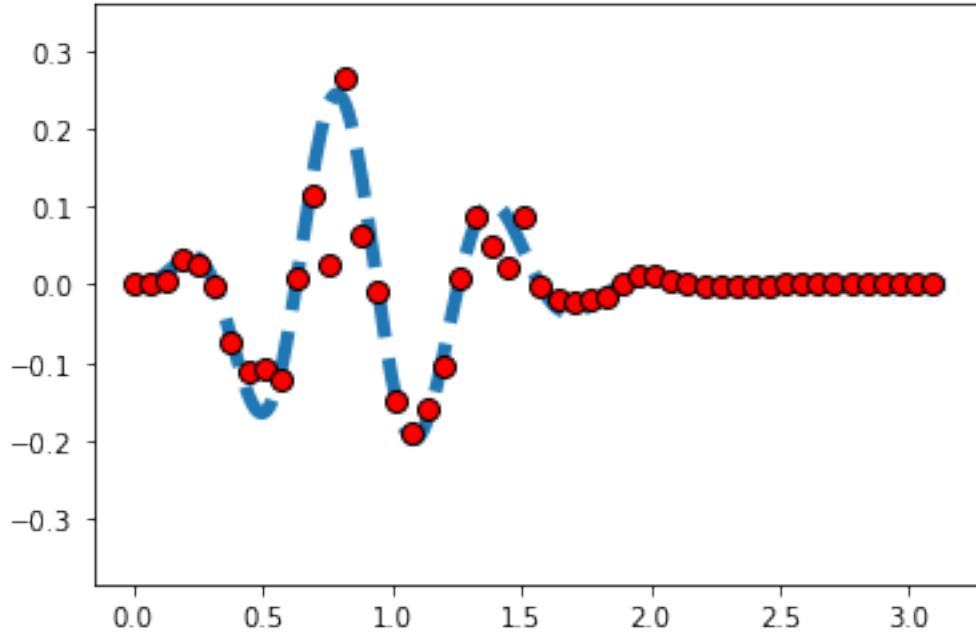
```
plt.plot(x,y,linewidth=5)
plt.plot(x,yexp, 'o', color='green', markersize=8)
```

```
[<matplotlib.lines.Line2D at 0x7f6524886d40>]
```



```
plt.plot(x,y,linewidth=5, linestyle='dashed')
plt.plot(x,yexp, 'o', color='red', markersize=8, markeredgecolor='black',markevery=6)
```

```
[<matplotlib.lines.Line2D at 0x7f6524962860>]
```



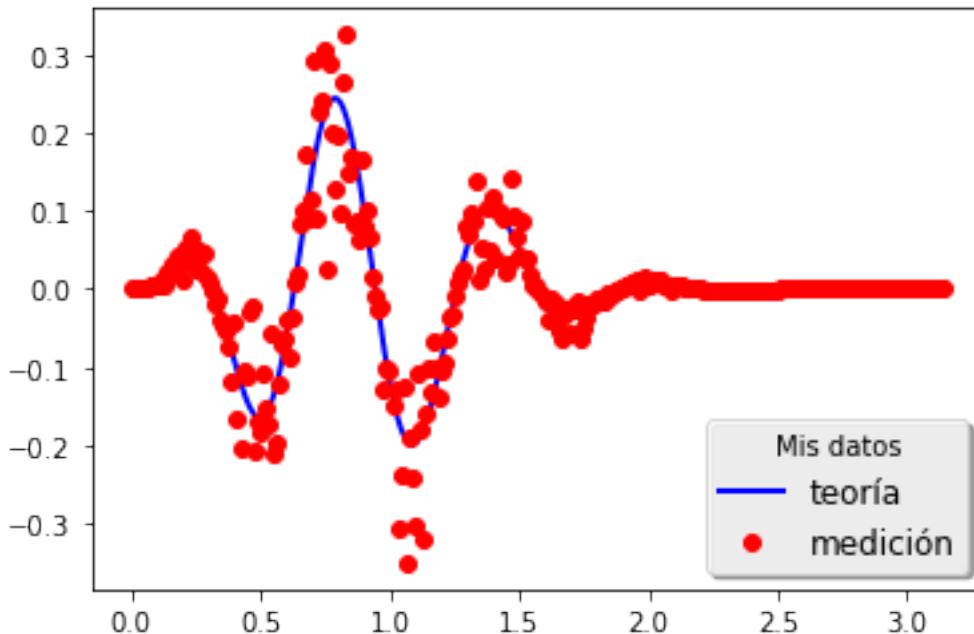
10.3.2 Nombres de ejes y leyendas

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()`

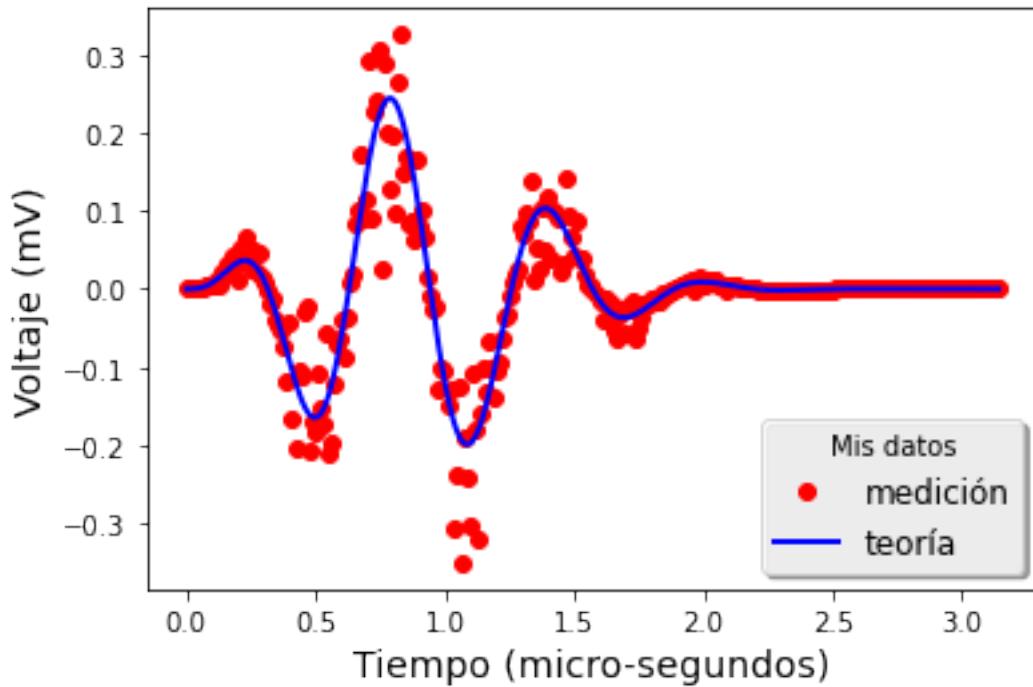
```
plt.plot(x,y, '-b', label="teoría")
plt.plot(x,yexp, 'or', label="medición")
# plt.legend()
plt.legend(loc="lower right", title="Mis datos")
```

```
<matplotlib.legend.Legend at 0x7f6524717e80>
```



Para agregar nombres a los ejes usamos `xlabel` y `ylabel`:

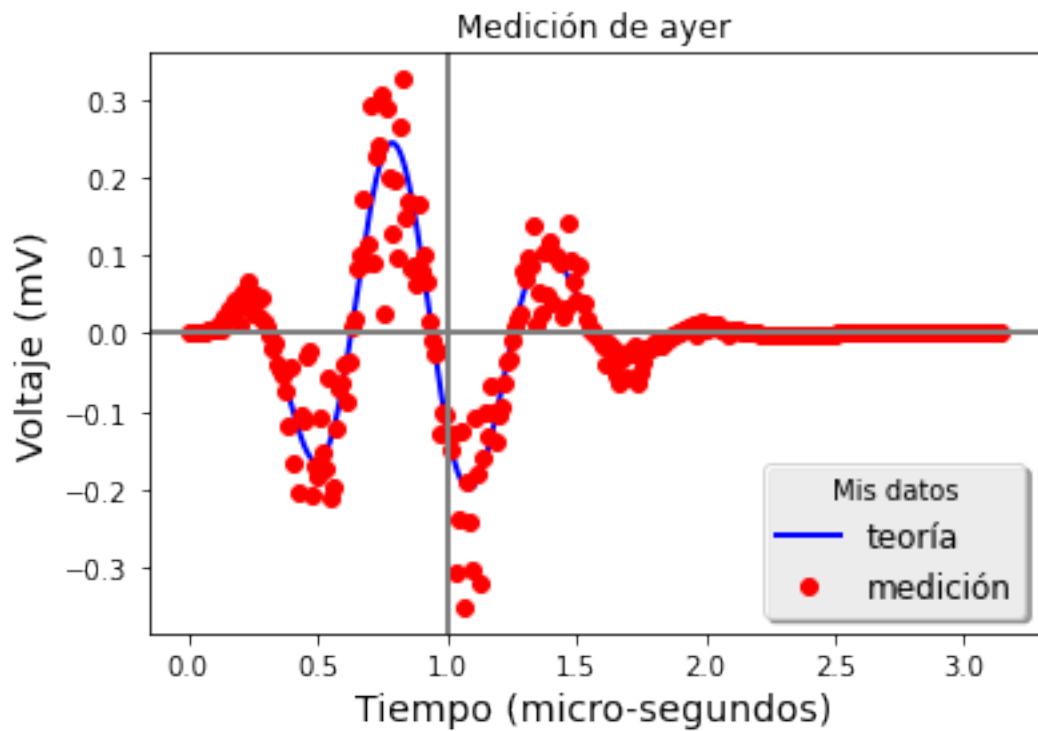
```
plt.plot(x,yexp, 'or', label="medición")
plt.plot(x,y, '-b', label="teoría")
plt.legend(loc="lower right", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)");
```



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 0x7f652460f9a0>
```



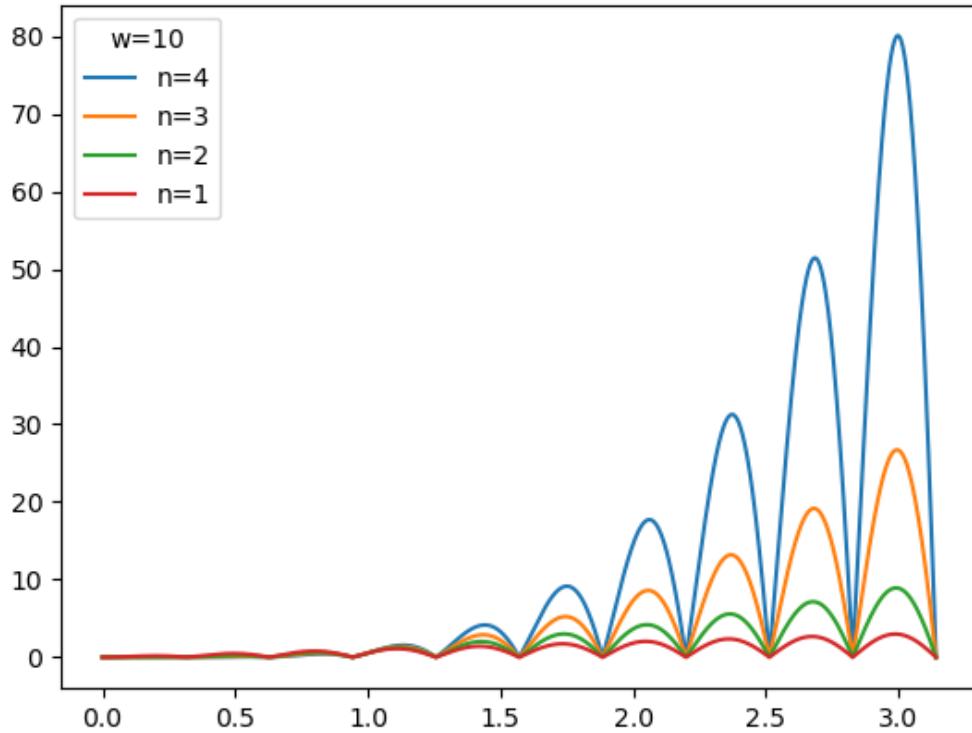
Acá además agregamos una línea vertical y una horizontal.

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

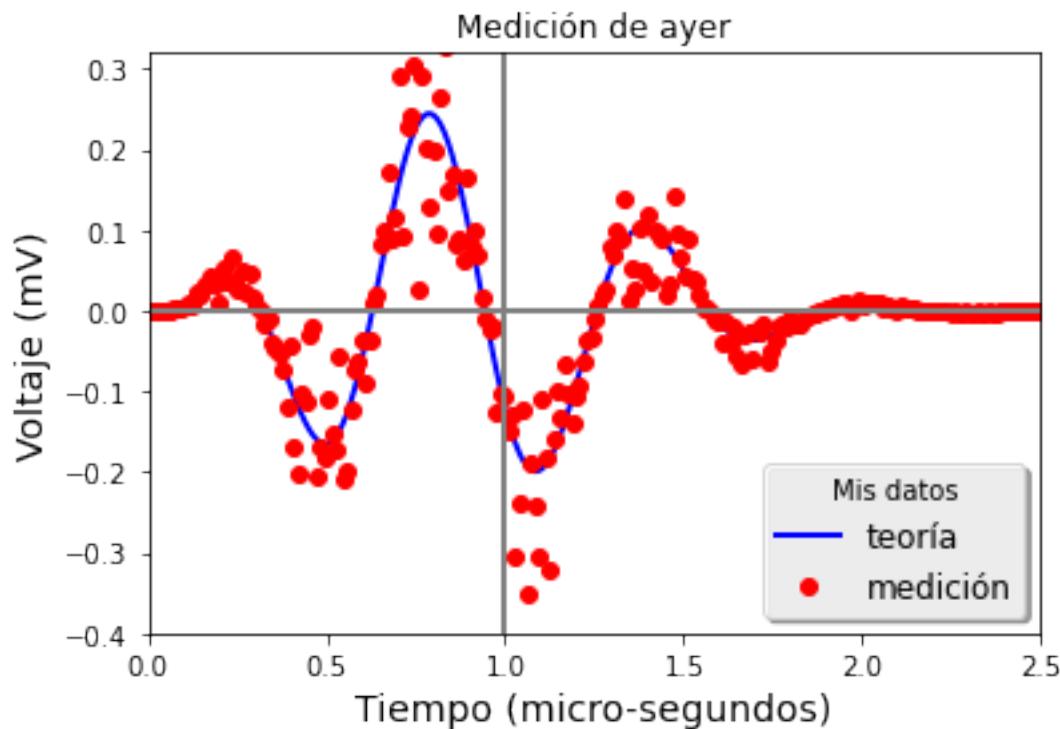


10.5 Escalas y límites de graficación (vista)

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))
```

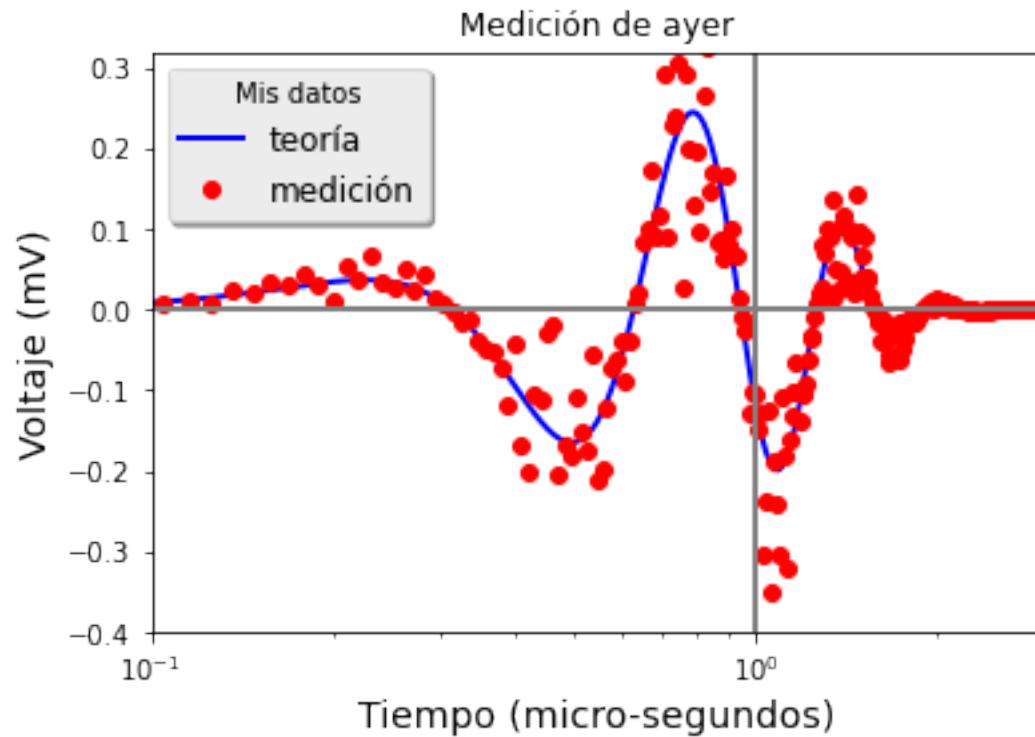
```
(-0.4, 0.32)
```



Para pasar a escala logarítmica usamos xscale o yscale

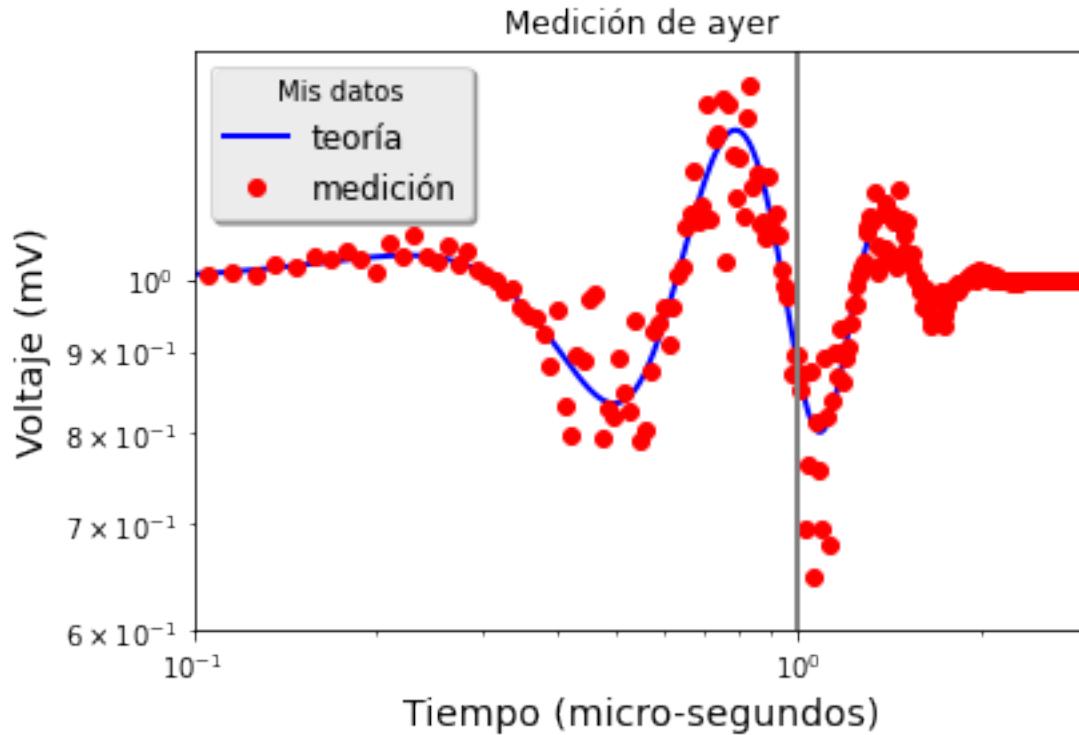
```
plt.plot(x,y, '-b', label="teoría")
plt.plot(x,yexp, 'or', label="medición")
plt.legend(loc="best", title="Mis datos")
plt.xlabel('Tiempo (micro-segundos)')
plt.ylabel("Voltaje (mV)")
plt.title("Medición de ayer")
plt.axvline(x=1, color='gray')
plt.axhline(color='gray')
plt.xlim((1.e-1,3))
plt.xscale('log')
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)
```



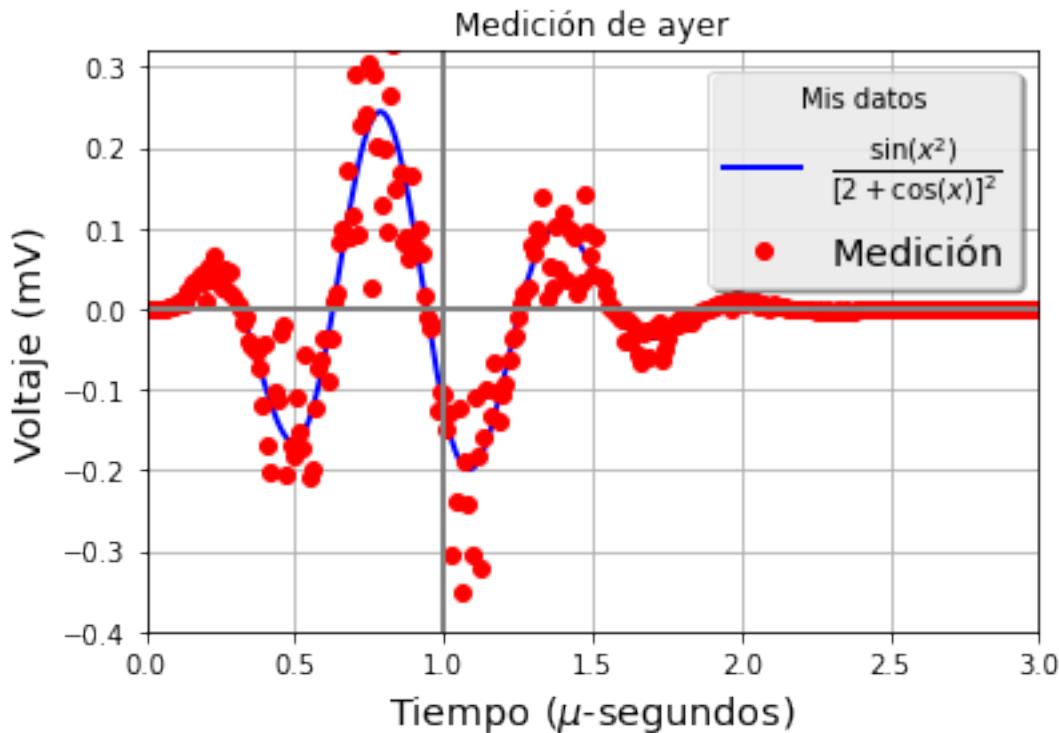
10.6 Exportar las figuras

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="$\mathit{Medición}$")
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()
#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(visible=None, which='major', axis='both', **kwargs)
Configure the grid lines.

Parameters
-----
visible : bool or None, optional
    Whether to show the grid lines. If any kwargs are supplied, it
    is assumed you want the grid on and visible will be set to True.

    If visible 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 keyword arguments are:
```

```

Properties:
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: scalar or None
animated: bool
antialiased or aa: bool
clip_box: .Bbox
clip_on: bool
clip_path: Patch or (Path, Transform) or None
color or c: color
dash_capstyle: .CapStyle or {'butt', 'projecting', 'round'}
dash_joinstyle: .JoinStyle or {'miter', 'round', 'bevel'}
dashes: sequence of floats (on/off ink in points) or (None, None)
data: (2, N) array or two 1D arrays
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 string, ~.path.Path or ~.markers.MarkerStyle
markeredgecolor or mec: color
markeredgegewidth 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) or list[bool]
path_effects: .AbstractPathEffect
picker: float or callable[[Artist, Event], tuple[bool, dict]]
pickradius: float
rasterized: bool
sketch_params: (scale: float, length: float, randomness: float)
snap: bool or None
solid_capstyle: .CapStyle or {'butt', 'projecting', 'round'}
solid_joinstyle: .JoinStyle or {'miter', 'round', 'bevel'}
transform: unknown
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 `~.Artist.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()`

```
# %load scripts/ejemplo_08_5.py
#!/usr/bin/ipython3

""" Script realizado durante la clase 9. 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))

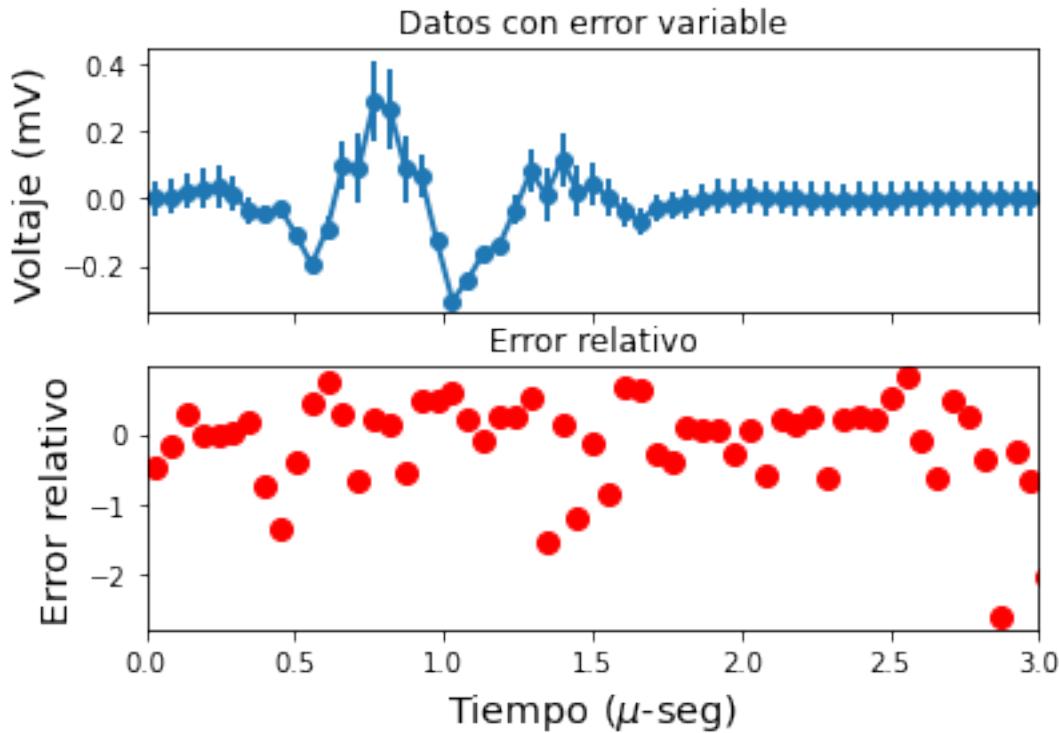
x1, y1, y2 = np.loadtxt(datfile, unpack=True)
# Vamos a graficar sólo algunos valores (uno de cada 5)
x = x1[3:-10:5]
y = y1[3:-10:5]
yexp = y2[3:-10:5]

# 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)
```



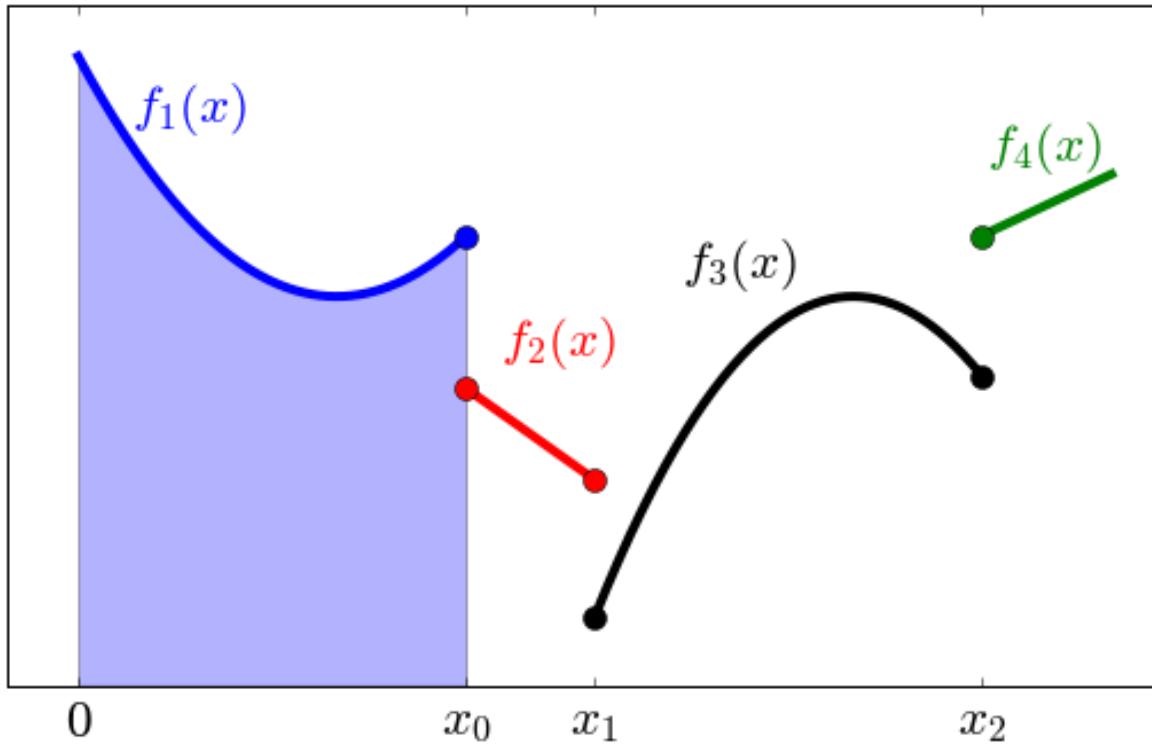
En este ejemplo utilizamos un enfoque diferente al utilizado anteriormente. Matplotlib presenta también una interface orientada a objetos. La función `subplots()` devuelve un par `fig, axis` que son dos objetos utilizados para representar la *figura* y un *eje*. Los métodos de estos objetos presentan funcionalidad similar a las funciones del módulo `pyplot`.

10.8 Ejercicios 09 (b)

2. Para la función definida a trozos:

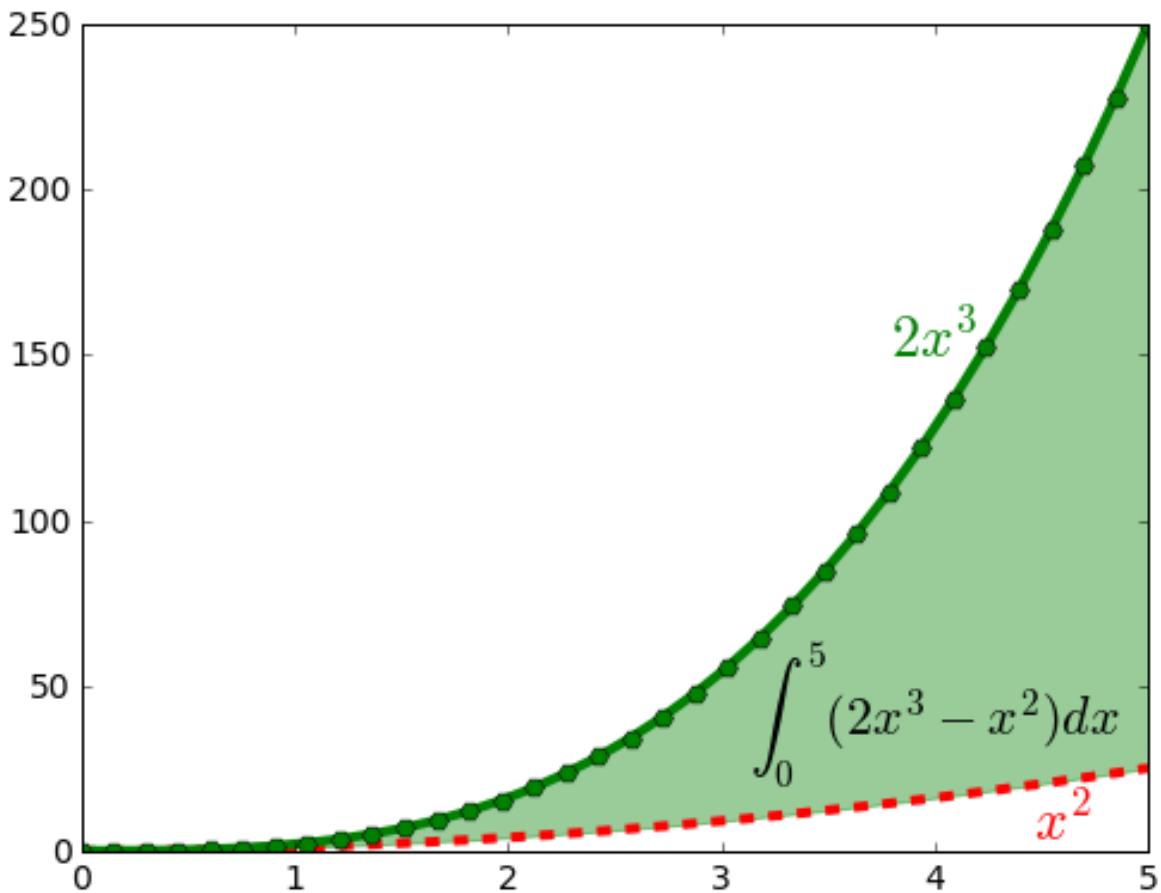
$$f(x) = \begin{cases} f_1(x) = x^2/8 & -\pi < x \leq \pi/2 \\ f_2(x) = -0,3x & \pi/2 < x < \pi \\ f_3(x) = -(x - 2\pi)^2/6 & \pi \leq x \leq 5\pi/2 \\ f_4(x) = (x - 2\pi)/5 & 5\pi/2 < x \leq 3\pi \end{cases}$$

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`.

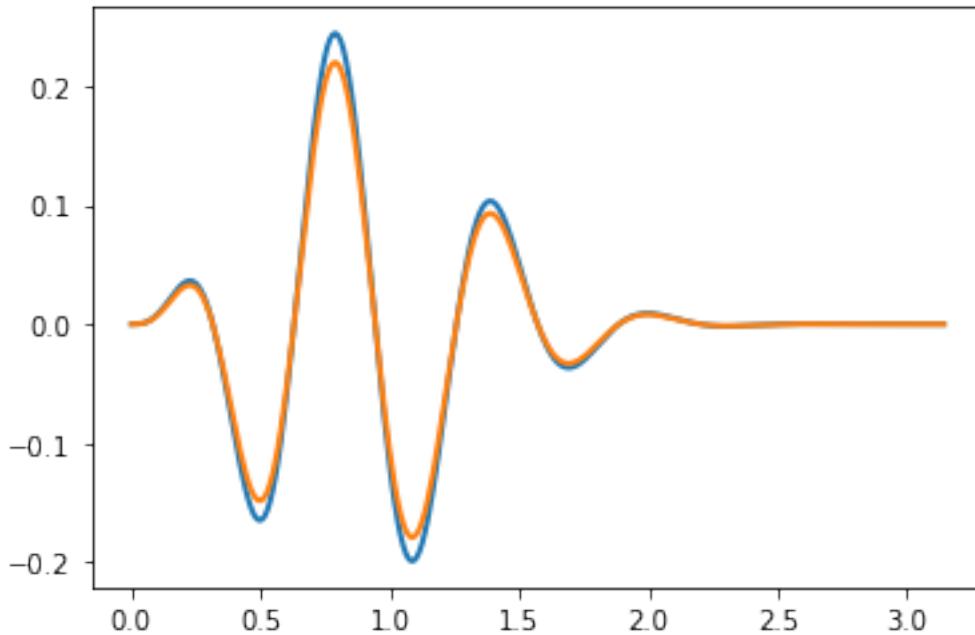
Este paquete tiene pre-definidos unos pocos estilos, entre ellos varios que emulan otros paquetes o programas. Veamos un ejemplo:

```
import numpy as np
import matplotlib.pyplot as plt
```

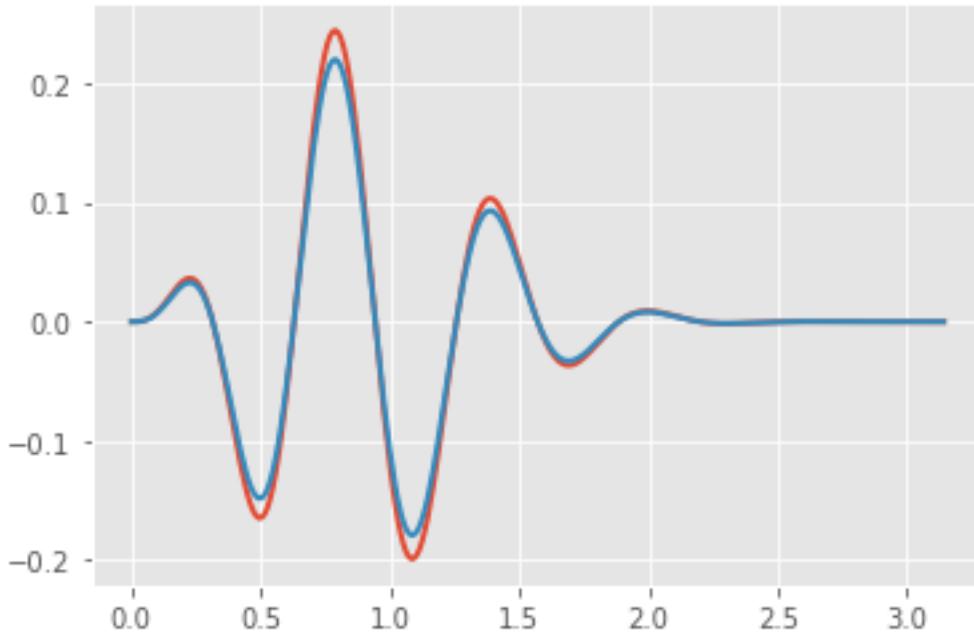
```
fdatos = '../data/ej_oscil_aten_err.dat'
x, y, yexp = np.loadtxt(fdatos, unpack=True)
```

```
plt.plot(x,y, x, 0.9*y)
```

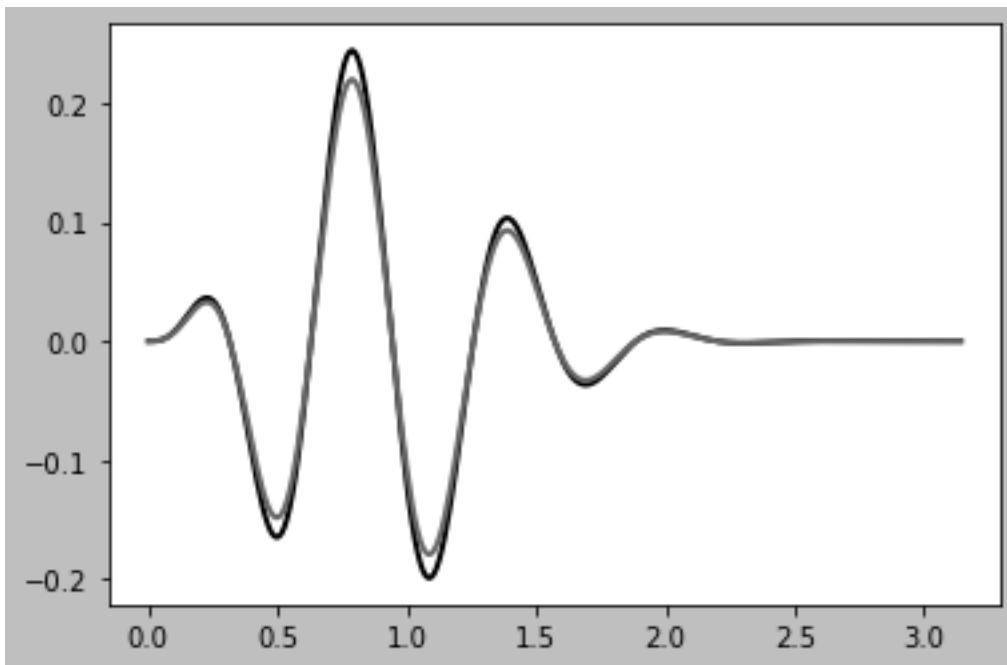
```
[<matplotlib.lines.Line2D at 0x7fa45ee06920>,
 <matplotlib.lines.Line2D at 0x7fa45ee06950>]
```



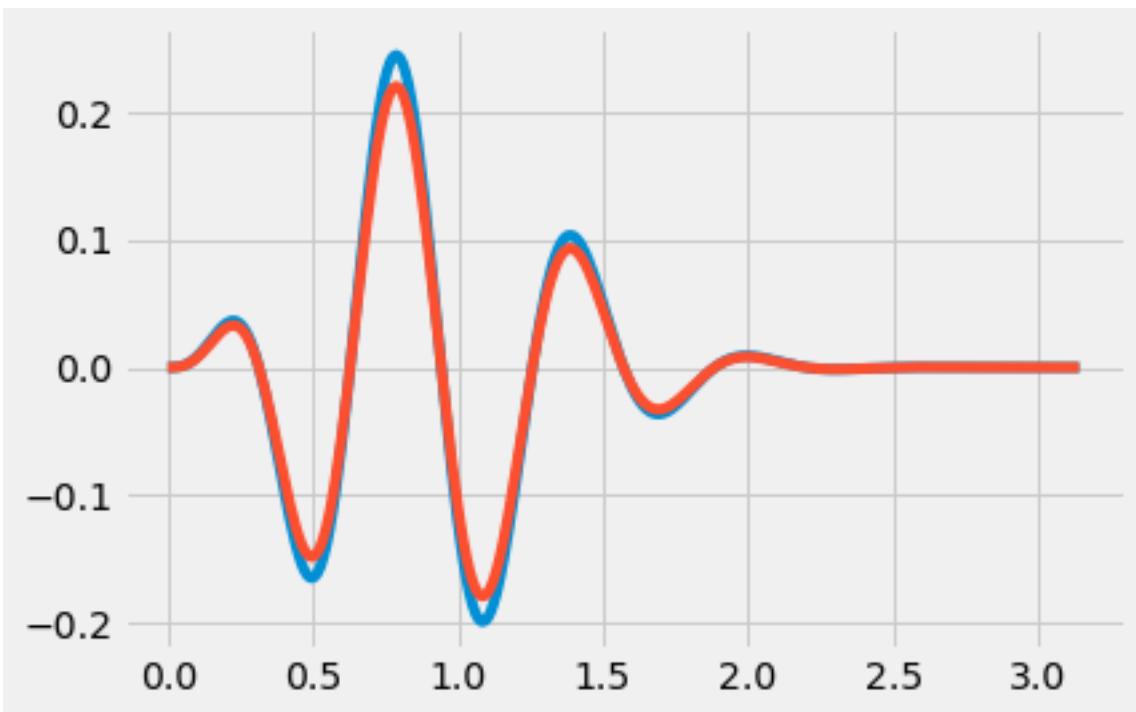
```
with plt.style.context('ggplot'):
    plt.plot(x,y, x, 0.9*y)
```



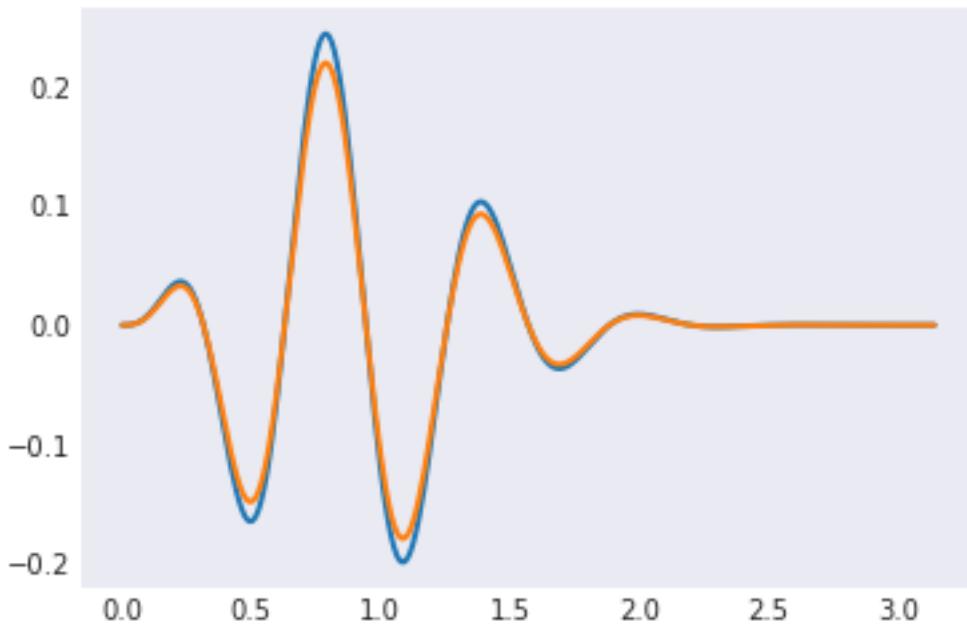
```
with plt.style.context('grayscale'):
    plt.plot(x,y, x, 0.9*y)
```



```
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
```

```
['Solarize_Light2',
 '_classic_test_patch',
```

(continué en la próxima página)

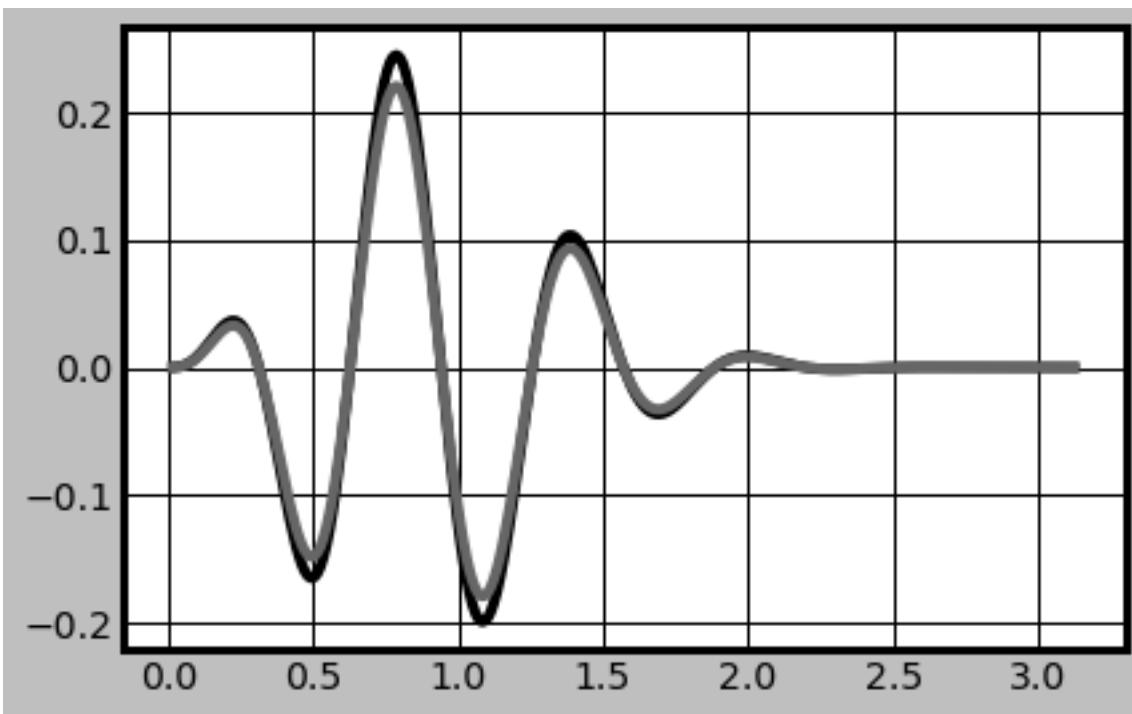
(proviene de la página anterior)

```
'_mpl-gallery',
'_mpl-gallery-nogrid',
'bmh',
'classic',
'dark_background',
'darker',
'fast',
'fivethirtyeight',
'ggplot',
'grayscale',
'latex',
'paper',
'presentation',
'seaborn',
'seaborn-bright',
'seaborn-colorblind',
'seaborn-dark',
'seaborn-dark-palette',
'seaborn-darkgrid',
'seaborn-deep',
'seaborn-muted',
'seaborn-notebook',
'seaborn-paper',
'seaborn-pastel',
'seaborn-poster',
'seaborn-talk',
'seaborn-ticks',
'seaborn-white',
'seaborn-whitegrid',
'tableau-colorblind10']
```

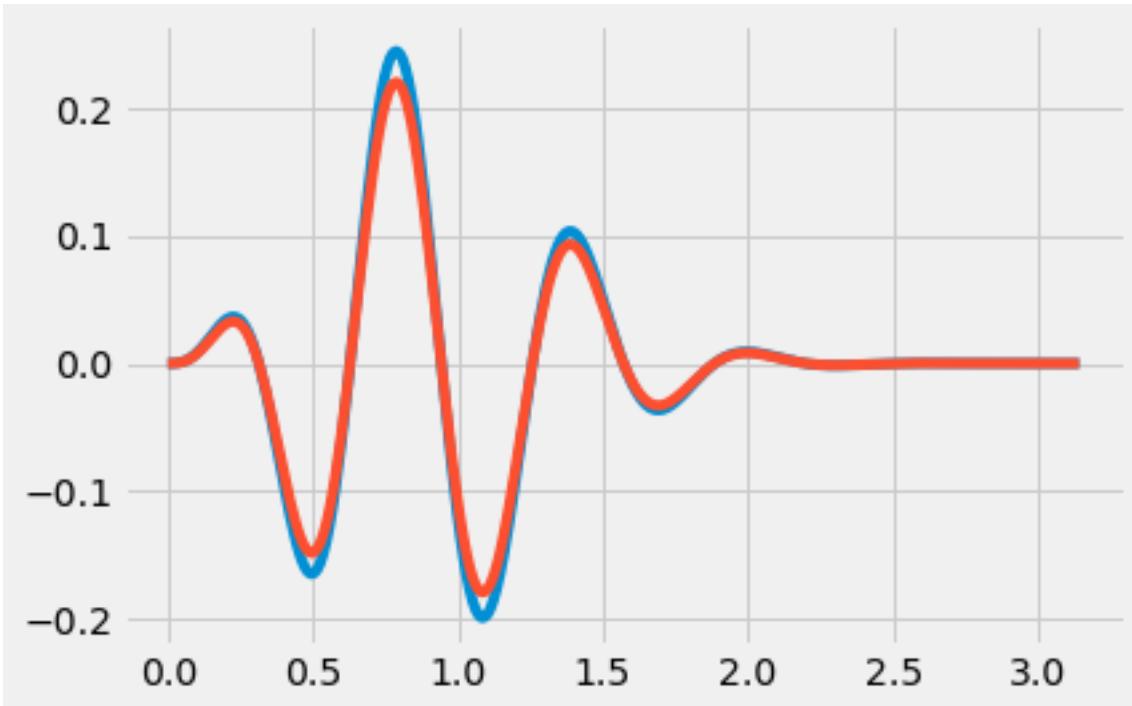
Combinando estilos

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

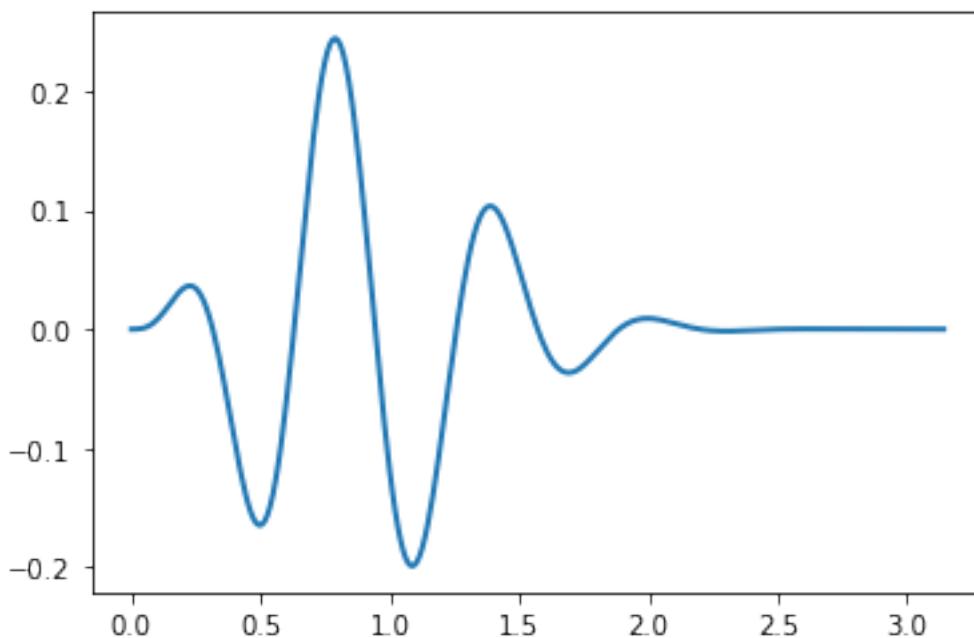
```
!echo "lines.linewidth : 5" > estilo_test  
!echo "xtick.labelsize: 24" >> estilo_test
```

```
!cat estilo_test
```

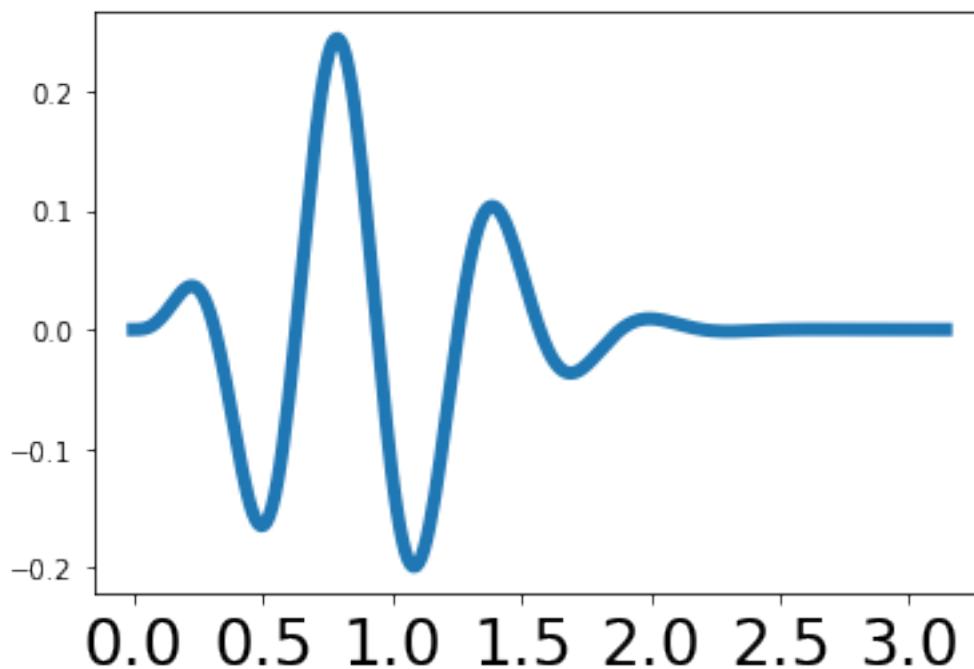
```
lines.linewidth : 5  
xtick.labelsize: 24
```

```
plt.plot(x,y)
```

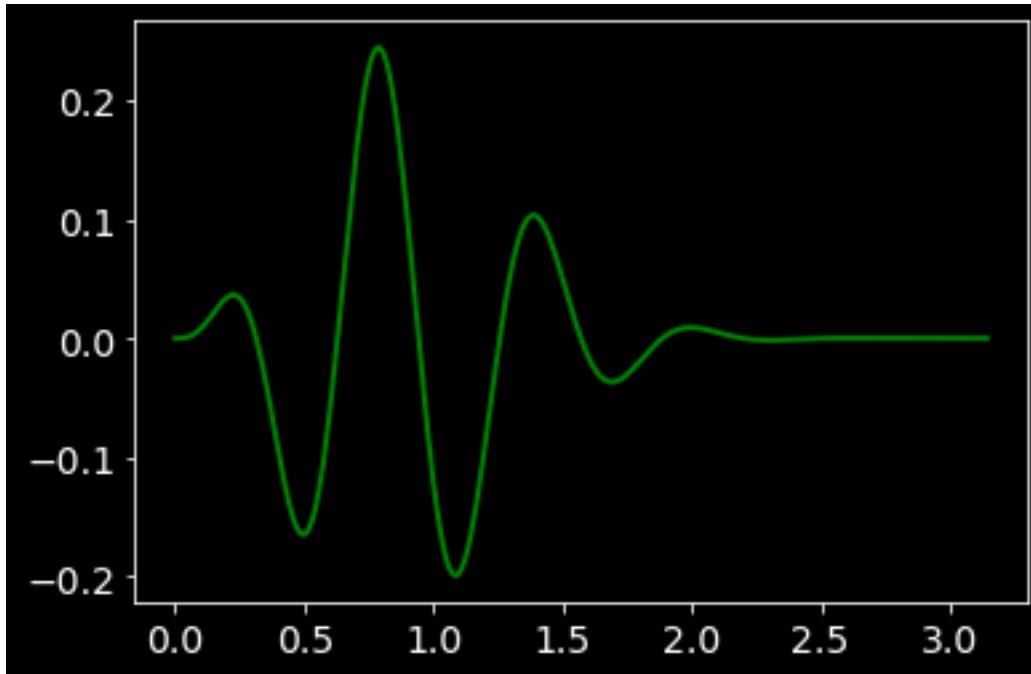
```
[<matplotlib.lines.Line2D at 0x7fa415857460>]
```



```
with plt.style.context('./estilo_test'):  
    plt.plot(x,y)
```



```
with plt.style.context('darker'):
    plt.plot(x,y,'g')
```



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  
latex.mplstyle  
paper.mplstyle  
presentation.mplstyle
```

```
!cat /home/fiol/.config/matplotlib/stylelib/darker.mplstyle
```

```
# -- mode: conf --
font.size      : 14
lines.color    : white
lines.linewidth : 2      # line width in points
lines.markersize : 8
patch.edgecolor : white

text.color : white

axes.facecolor : black
axes.edgecolor : white
axes.labelcolor : white
# Seaborn dark palette
axes.prop_cycle: cycler('color', ['001C7F', '017517', '8C0900', '7600A1',
                                   'B8860B', '006374'])

xtick.color : white
ytick.color : white

grid.color : white

figure.facecolor : black
figure.edgecolor : black

savefig.facecolor : black
savefig.edgecolor : black
```

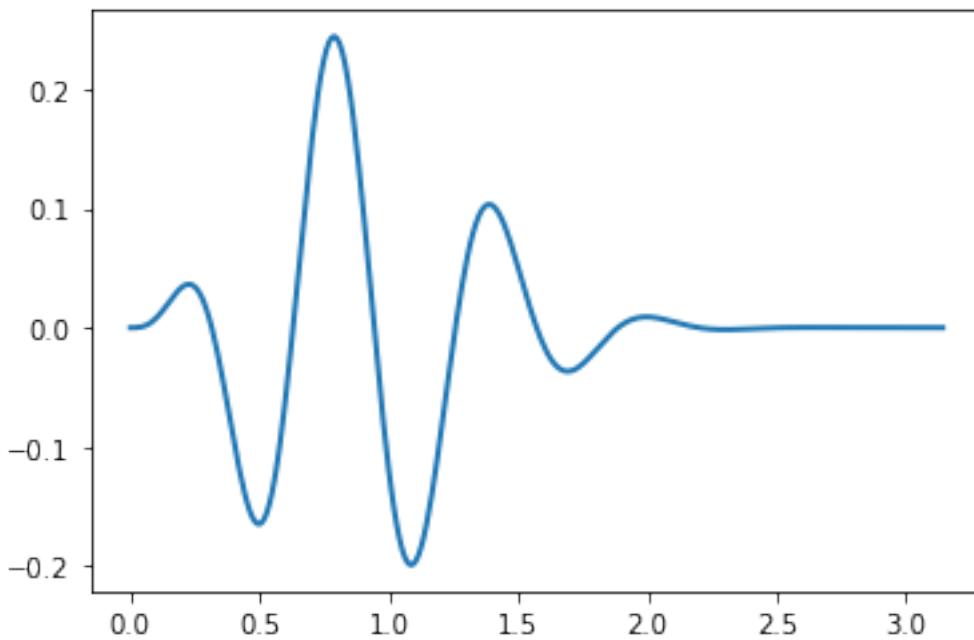
10.9.3 Modificación de parámetros dentro de programas

Podemos cambiar directamente los parámetros dentro de nuestros programas modificando el diccionario `matplotlib.rcParams`

```
import matplotlib as mpl
mpl.rcParams
```

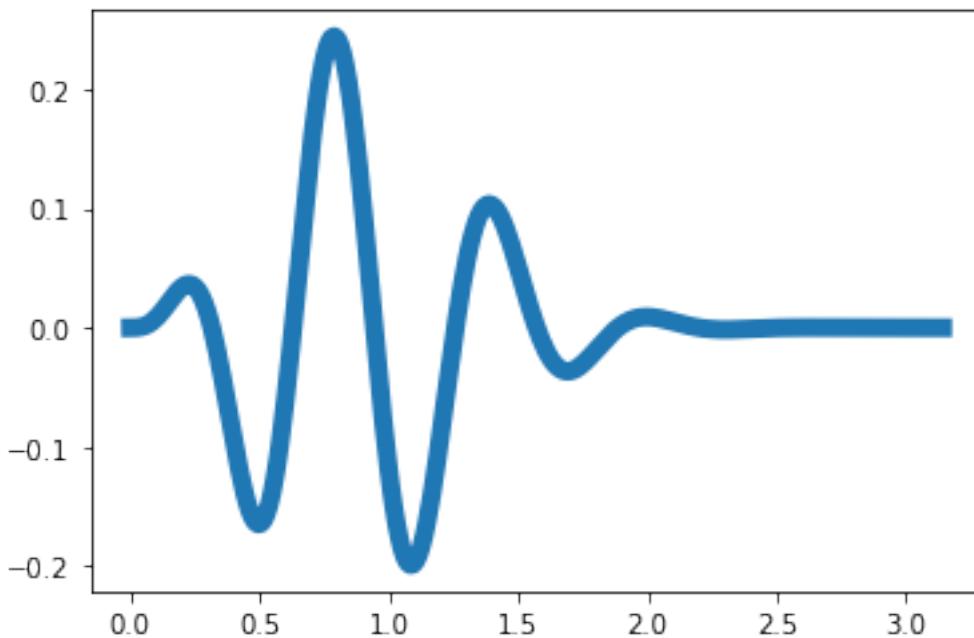
```
# Plot con valores default
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa4155b6d10>]
```



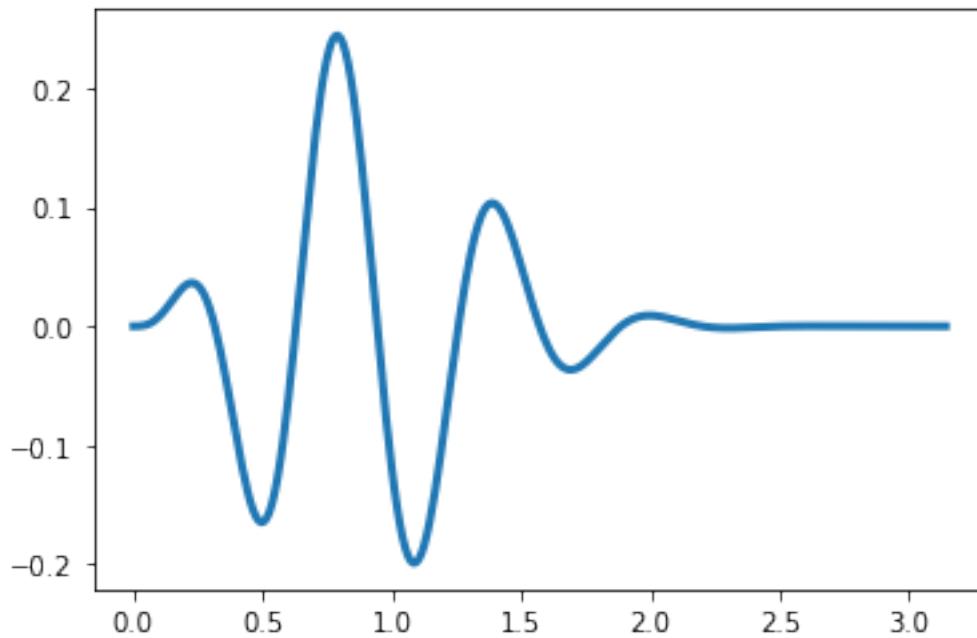
```
# Modificamos el valor default de ancho de línea  
mpl.rcParams['lines.linewidth'] = 7  
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa4158150f0>]
```



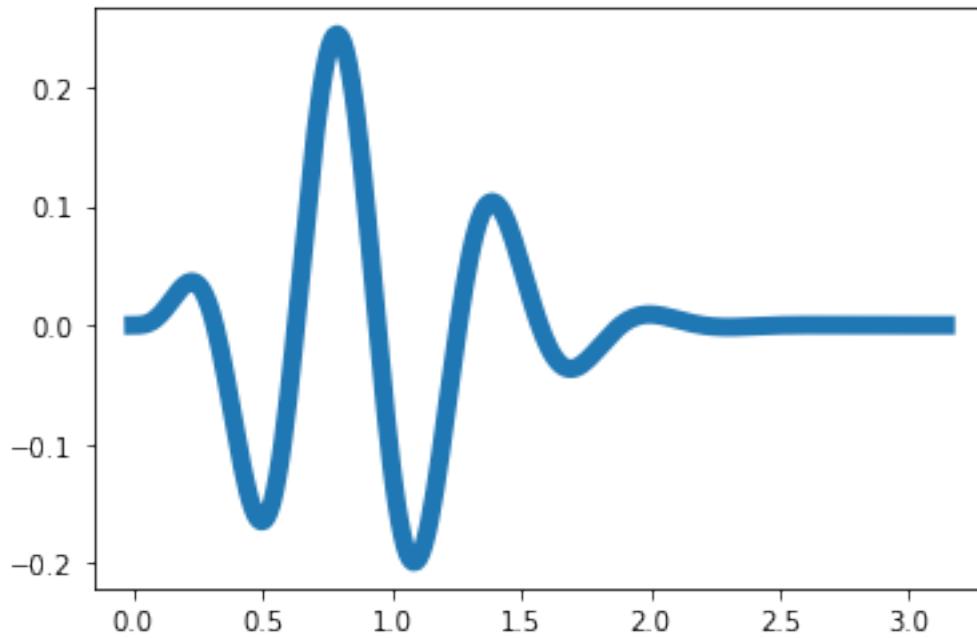
```
# El nuevo valor default podemos sobreescribirlo para este plot particular  
plt.plot(x,y, lw=3)
```

```
[<matplotlib.lines.Line2D at 0x7fa41592f0d0>]
```



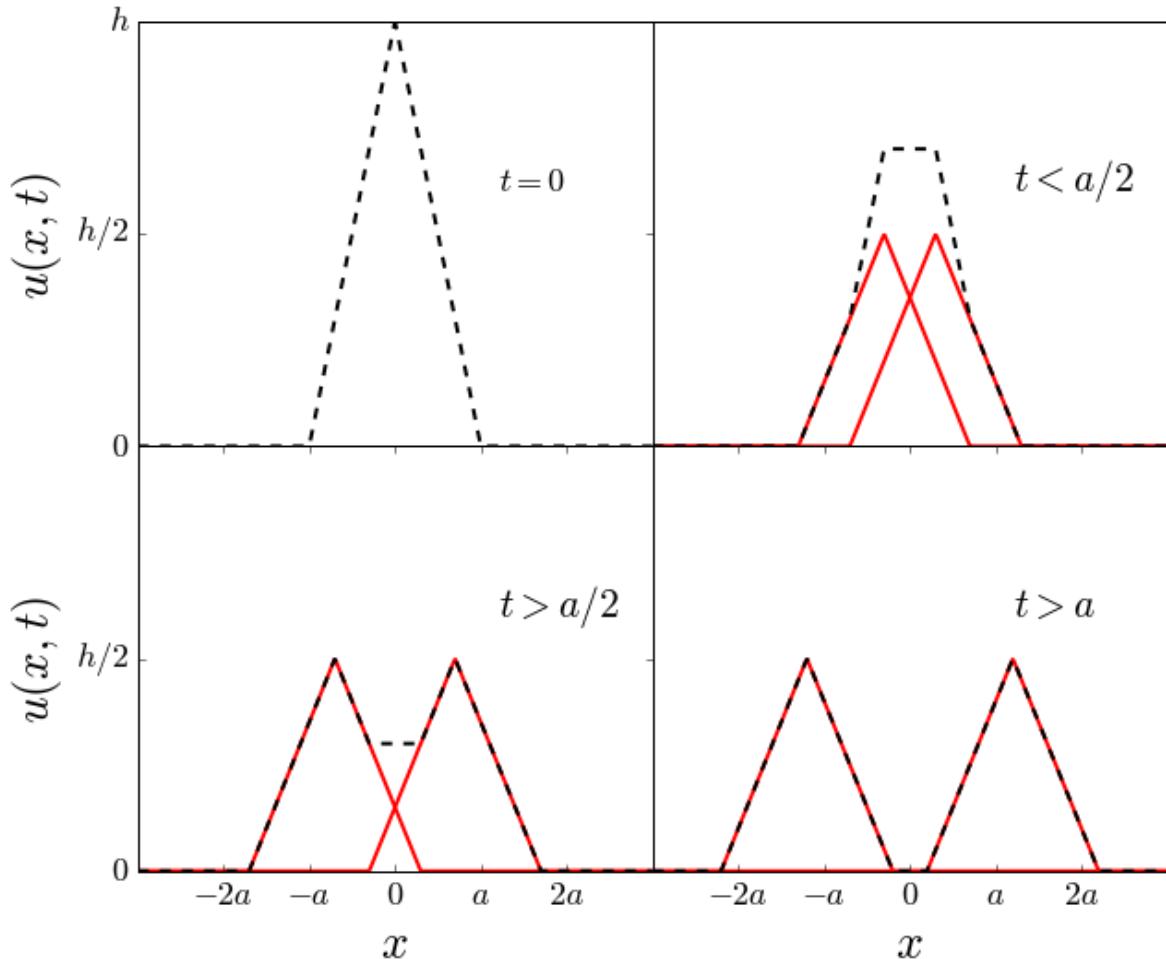
```
# Sin embargo, el nuevo valor default no es modificado  
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fa415a718a0>]
```



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

Vemos 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:
 [ 38.06704666  62.77821128  87.4893759  112.20054053 136.91170515
 161.62286977 186.33403439 211.04519901 235.75636364 241.64      ]
*****
```

```
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 1 0 0]  
 [0 0 2 0]  
 [0 0 0 3]  
 [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(arr2))
```

```
Transpose:
[[ 5  4  8]
 [ 1  5 -9]
 [ 2  6 10]
 [ 3  7 11]]
Transpose:
[[ 5  4  8]
 [ 1  5 -9]
 [ 2  6 10]
 [ 3  7 11]]
```

11.2.3 min, max

Las funciones para encontrar mínimo y máximo pueden aplicarse tanto a vectores como a arrays` 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 *tuple* 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 Conveniencias con arrays

11.3.1 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])
```

11.3.2 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(arr2)
```

```
[[ 5  1  2  3]
 [ 4  5  6  7]
 [ 8 -9 10 11]]
```

```
for (i,j), x in np.ndenumerate(arr2):
    print(f'x[{i},{j}]-> {x}')
```

```
x[0,0]-> 5
x[0,1]-> 1
x[0,2]-> 2
x[0,3]-> 3
x[1,0]-> 4
x[1,1]-> 5
x[1,2]-> 6
x[1,3]-> 7
x[2,0]-> 8
x[2,1]-> -9
x[2,2]-> 10
x[2,3]-> 11
```

11.3.3 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, integración u otro cálculo complejo, y si bien la paralelización puede ser trivial, el cálculo debe ser realizado para cada valor de algún parámetro y no puede ser realizado directamente con un vector. Para ello existe la función `vectorize()`. Veamos un ejemplo, calculando la función *coseno()* como la integral del *seno()*

```
def my_trapz(f, a, b):
    x = np.linspace(a,b,100)
    y = f(x)
    return ((y[1:]+y[:-1])*(x[1:]-x[:-1])).sum()/2
```

```
def mi_cos(t):
    return 1-my_trapz(np.sin, 0, t)
```

```
mi_cos(np.pi/4)
```

```
0.7071083173516677
```

que se compara bastante bien con el valor esperado del coseno:

```
np.cos(np.pi/4)
```

```
0.7071067811865476
```

Para calcular sobre un conjunto de datos:

```
x = np.linspace(0,np.pi,30)
```

```
print(mi_cos(x))
```

```
-28.998610761224644
```

Obtuvimos un valor único que claramente no puede ser el coseno de ningún ángulo. Si calculamos el coseno con el mismo argumento (vectorial) obtenemos un vector de valores como se espera:

```
print(np.cos(x))
```

```
[ 1.          0.99413796  0.97662056  0.94765317  0.90757542  0.85685718
  0.79609307  0.72599549  0.64738628  0.56118707  0.46840844  0.37013816
  0.26752834  0.161782   0.05413891 -0.05413891 -0.161782   -0.26752834
 -0.37013816 -0.46840844 -0.56118707 -0.64738628 -0.72599549 -0.79609307
 -0.85685718 -0.90757542 -0.94765317 -0.97662056 -0.99413796 -1.        ]
```

Si el cálculo fuera más complejo y no tuviéramos la posibilidad de realizarlo en forma vectorial, debemos realizar una iteración llamando a esta función en cada paso:

```
y = []
for xx in x:
    y.append(mi_cos(xx))
print(np.array(y))
```

```
[ 1.          0.99413796  0.97662057  0.94765322  0.90757557  0.85685753
  0.7960938   0.72599683  0.64738854  0.56119061  0.46841375  0.37014576
```

(continué en la próxima página)

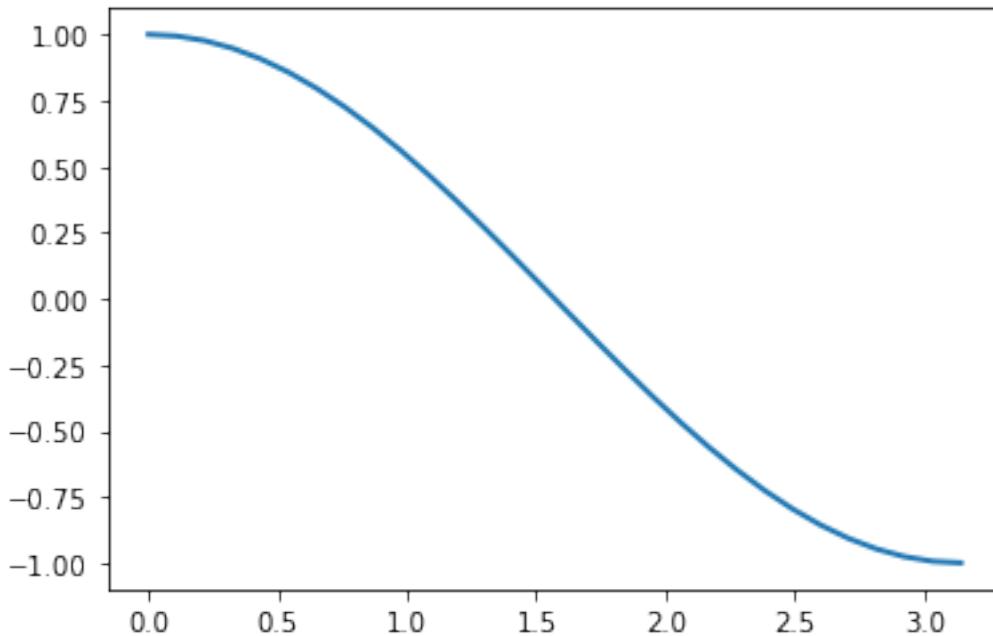
(provien de la página anterior)

```
0.26753886  0.16179613  0.05415741 -0.05411524 -0.16175232 -0.26749179
-0.37009386 -0.46835555 -0.56112475 -0.64731379 -0.72591214 -0.79599826
-0.85675045 -0.90745645 -0.9475218 -0.97647677 -0.99398196 -0.99983216]
```

```
y = np.zeros(x.size)
for i,xx in enumerate(x):
    y[i] = mi_cos(xx)
```

```
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7f7b6aef8130>]
```

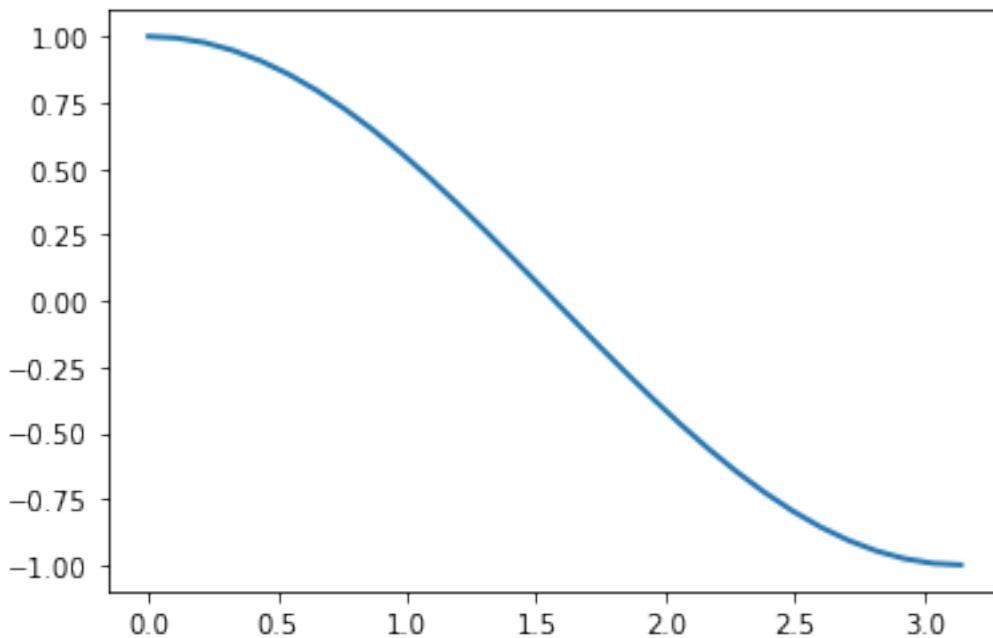


Como conveniencia, para evitar tener que hacer explícitamente el bucle `for` existe la función `vectorize`, que toma como argumento a una función que toma y devuelve escalares, y retorna una función equivalente que acepta arrays:

```
coseno = np.vectorize(mi_cos)
```

```
plt.plot(x, coseno(x), '-')
```

```
[<matplotlib.lines.Line2D at 0x7f7b6ad588b0>]
```



11.4 Ejercicios 10 (a)

1. 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()`

```
import numpy as np
import matplotlib.pyplot as plt
```

```
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 atributo `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.]
```

```
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.]]
```

(continué en la próxima página)

Clases de Python

(proviene de la página anterior)

```
[ 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(x)
print(v3)
print('x  shape:', x.shape)
print('v3  shape:', v3.shape)
```

```
[0. 0.5 1. 1.5 2. 2.5 3. ]
[[0.5 0. ]
 [1. 0.5]]
x  shape: (7,)
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]
```

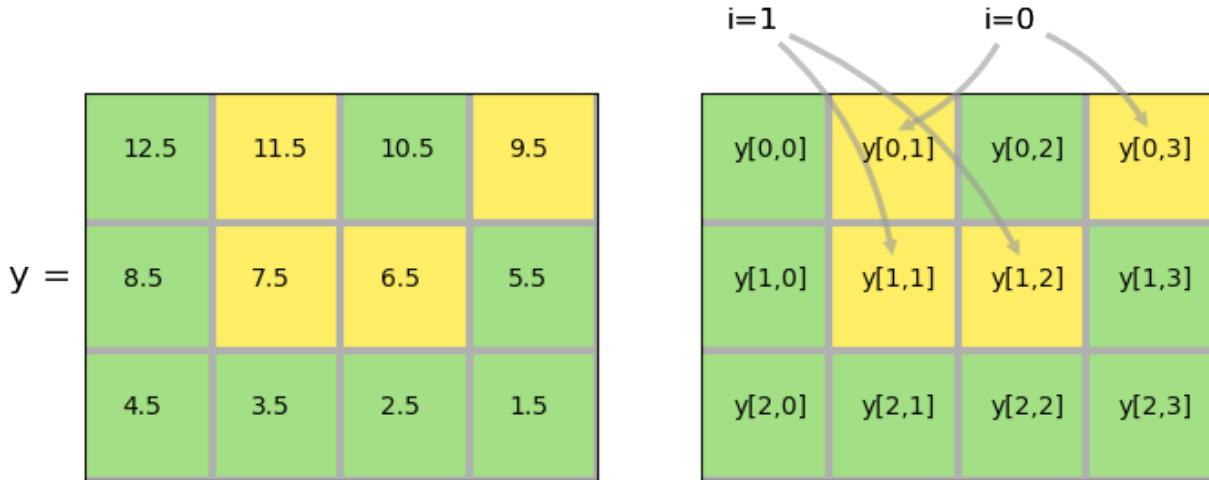
```
i = np.array([0,2])
print(y[i])                      # Primera y última fila
```

```
[[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]]
```



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:

Clases de Python

```
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]
 [ True  True False False]
 [False False False False]]
```

El resultado de una comparación es un array donde cada elemento es una 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(y)
print(np.compress(c2,y))
```

```
[False False False False False  True  True  True  True False False]
[[12.5 11.5 10.5  9.5]
 [ 8.5  7.5  6.5  5.5]
 [ 4.5  3.5  2.5  1.5]]
[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])
```

```
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]]
```

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

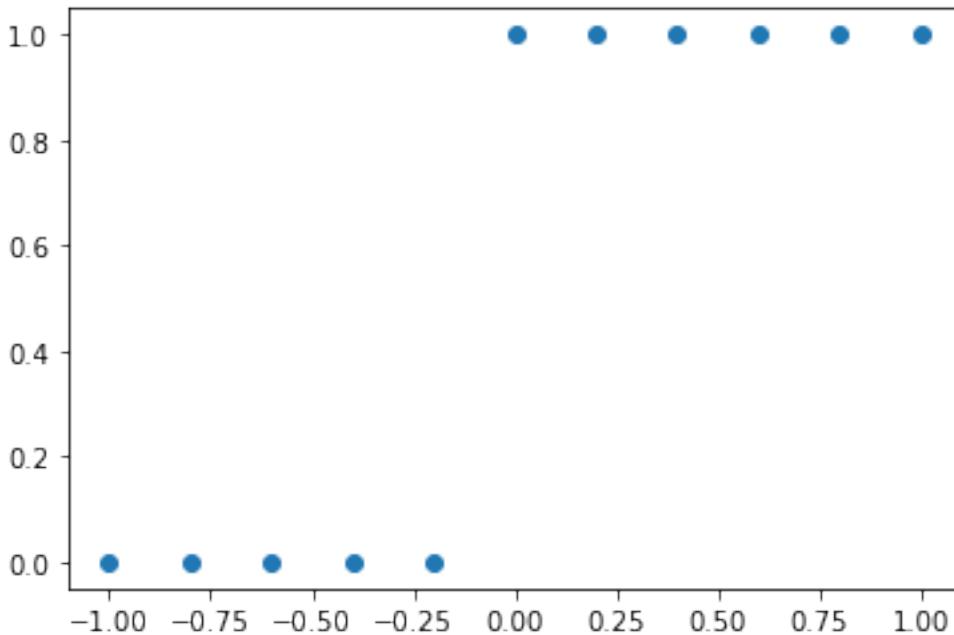
```
import matplotlib.pyplot as plt

def H(x):
    return np.where(x < 0, 0, 1)
x = np.linspace(-1, 1, 11)
H(x)
```

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

```
plt.plot(x, H(x), 'o')
```

```
[<matplotlib.lines.Line2D at 0x7fdb375c7370>]
```



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)
```

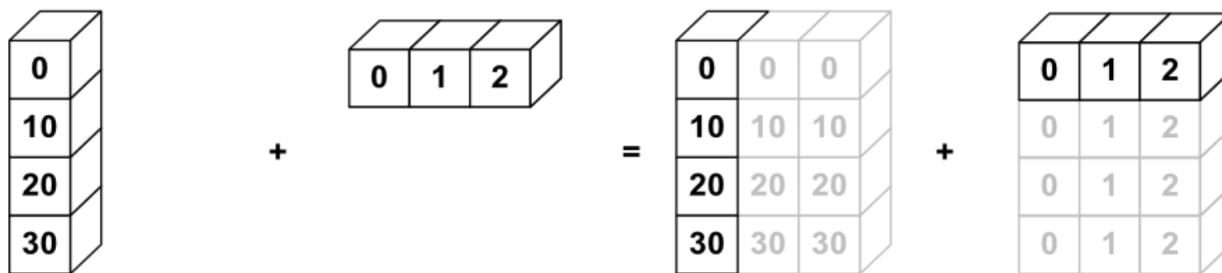
```
[0 1 2]
```

```
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
- xx + y -> xxx + 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]])
```

Clases de Python

```
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]]

 [[ -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)
```

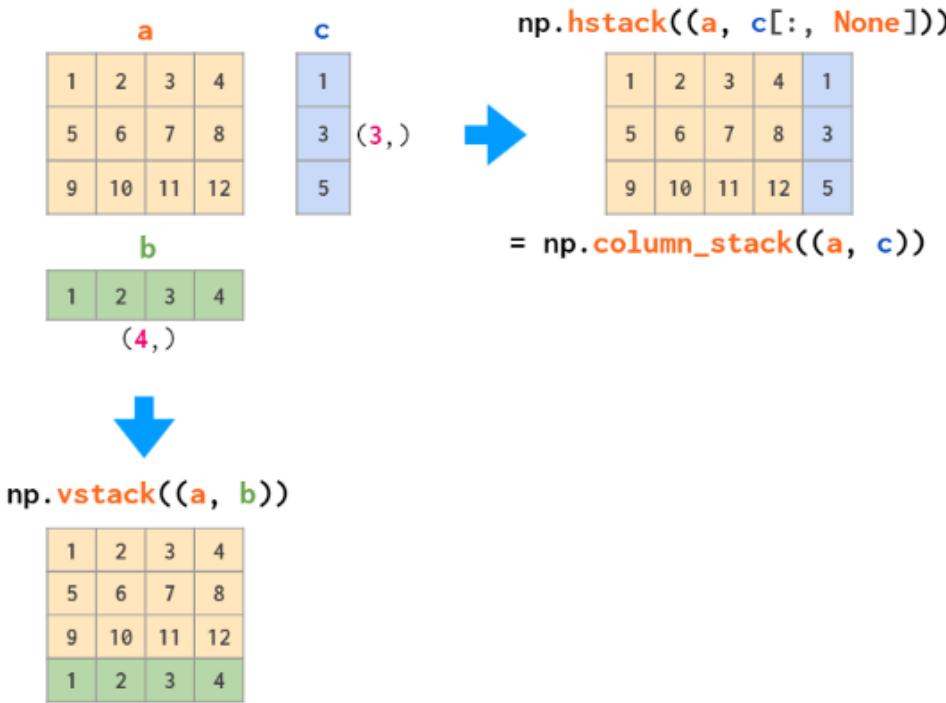
```
(2, 2) (3, 2) (2, 3)
```

```
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 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 que hay en el módulo

```
dir(np.random)
```

```
['BitGenerator',
 'Generator',
 'MT19937',
 'PCG64',
 'PCG64DXSM',
 'Philox',
 'RandomState',
 'SFC64',
 'SeedSequence',
 '__RandomState_ctor',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__']
```

(continué en la próxima página)

(proviene de la página anterior)

```
'_bounded_integers',
'_common',
'_generator',
'_mt19937',
'_pcg64',
'_philox',
'_pickle',
'_sfc64',
'beta',
'binomial',
'bit_generator',
'bytes',
'chisquare',
'choice',
'default_rng',
'dirichlet',
'exponential',
'f',
'gamma',
'geometric',
'get_state',
'gumbel',
'hypergeometric',
'laplace',
'logistic',
'lognormal',
'logseries',
'mtrand',
'multinomial',
'multivariate_normal',
'negative_binomial',
'noncentral_chisquare',
'noncentral_f',
'normal',
'pareto',
'permutation',
'poisson',
'power',
'rand',
'randint',
'randn',
'random',
'random_integers',
'random_sample',
'ranf',
'rayleigh',
'sample',
'seed',
;set_state',
'shuffle',
'standard_cauchy',
'standard_exponential',
'standard_gamma',
'standard_normal',
'standard_t',
'test',
'triangular',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'uniform',
'veonmises',
'wald',
'weibull',
'zipf']
```

11.9.1 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)
```

```
[[0.89398968 0.661581]
 [0.01149467 0.80208523]
 [0.76960027 0.67185184]
 [0.30586482 0.9476556]]
```

```
y
```

```
array([0.10805669, 0.08933825, 0.45591515, 0.93319955, 0.11794976,
       0.31490434, 0.21031763, 0.72402371])
```

```
help(np.random.random)
```

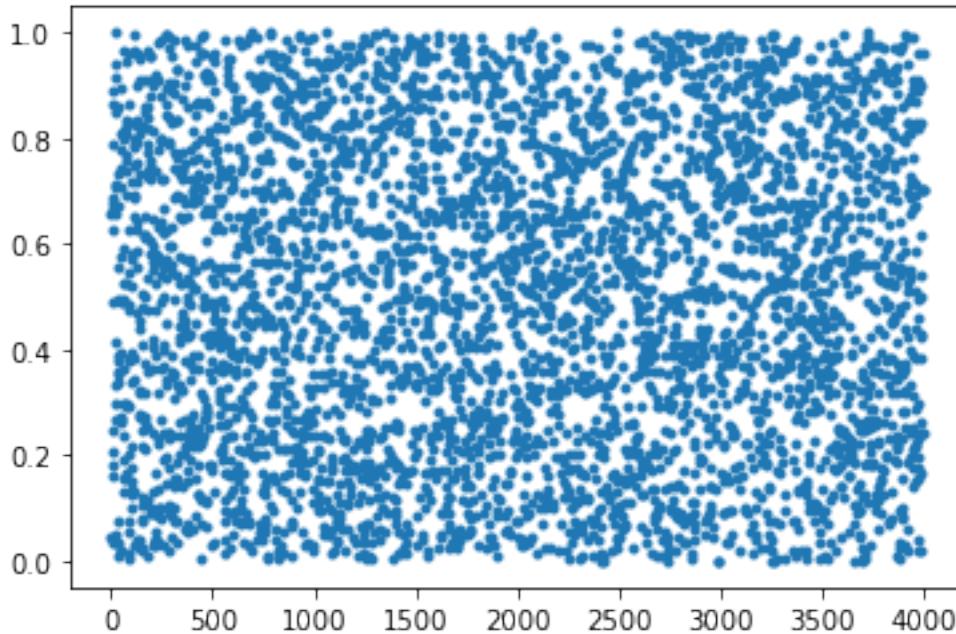
Help on built-in function random:

```
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)
```

11.9.2 Distribución normal (Gaussiana)

Una distribución de probabilidad normal tiene la forma Gaussiana

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

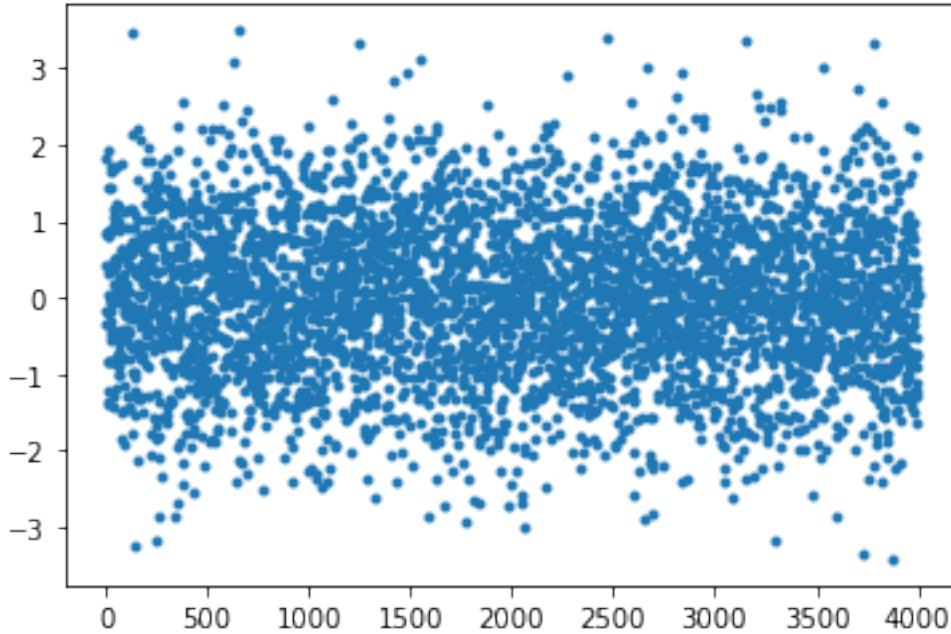
En **Numpy** la función que nos da elementos con esa distribución de probabilidad es:

```
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([[ 1.89330409, -0.82678319,  0.28456255, -0.20158241, -0.93699684],
       [-0.65001554, -0.07082357,  0.28431164,  1.03876825, -0.75345833],
       [-1.3222013 ,  0.55723735, -0.01766518,  0.13955381, -0.75881249]])
```

11.9.3 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([-3.41410287, -3.06867165, -2.72324042, -2.3778092 , -2.03237797,
       -1.68694674, -1.34151552, -0.99608429, -0.65065307, -0.30522184,
       0.04020938,  0.38564061,  0.73107183,  1.07650306,  1.42193429,
       1.76736551,  2.11279674,  2.45822796,  2.80365919,  3.14909041,
       3.49452164])
```

```
h
```

```
array([ 5,    9,   27,   45,  104,  177,  275,  390,  488,  554,  548,  451,  350,
       267,  161,   88,   33,   14,     8,      6])
```

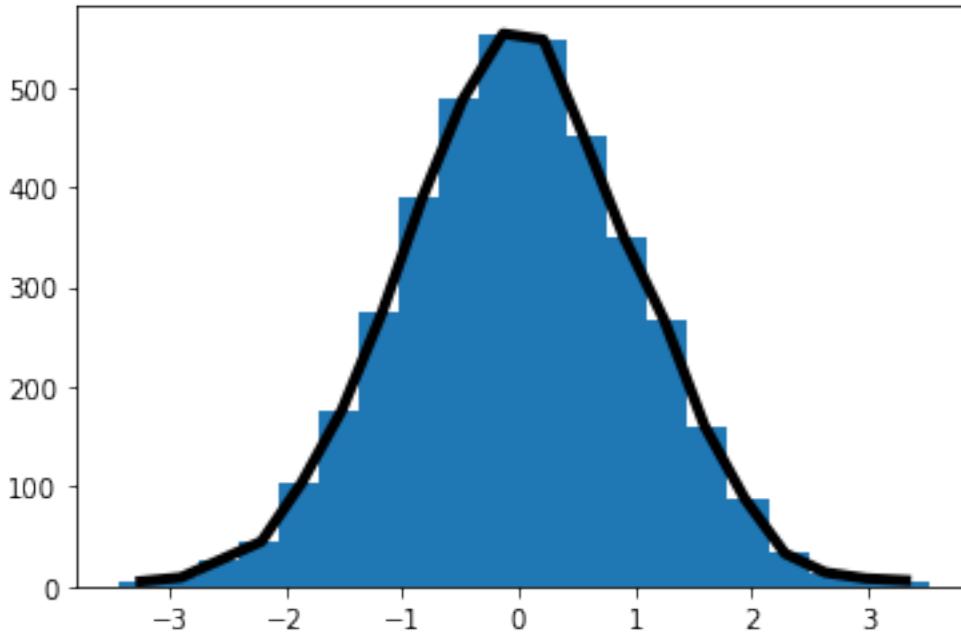
```
b.size, h.size
```

```
(21, 20)
```

La función retorna `b`: los límites de los intervalos en el eje x y `h` las alturas

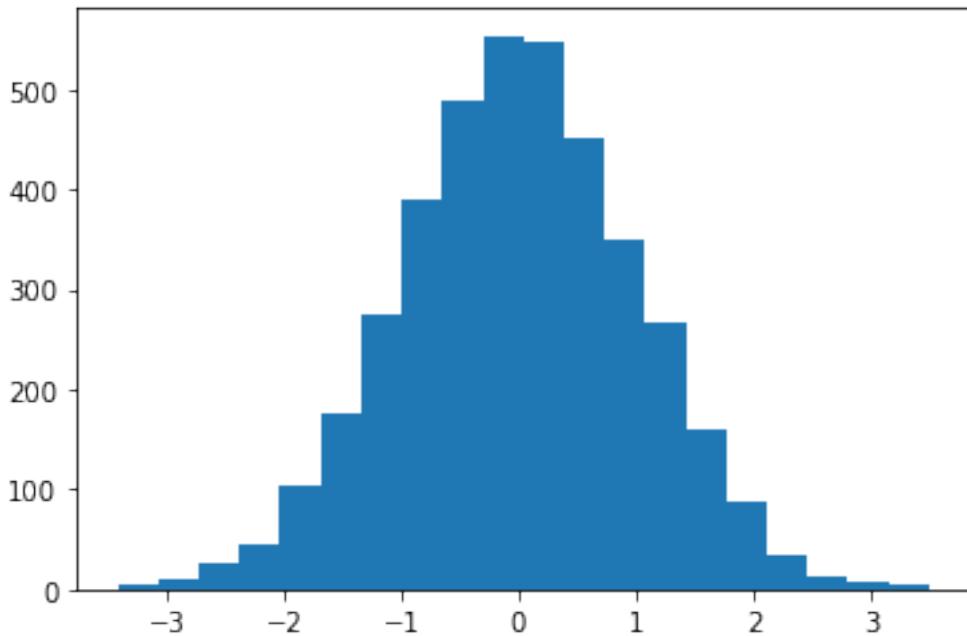
```
x = (b[1:] + b[:-1])/2
```

```
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()
```

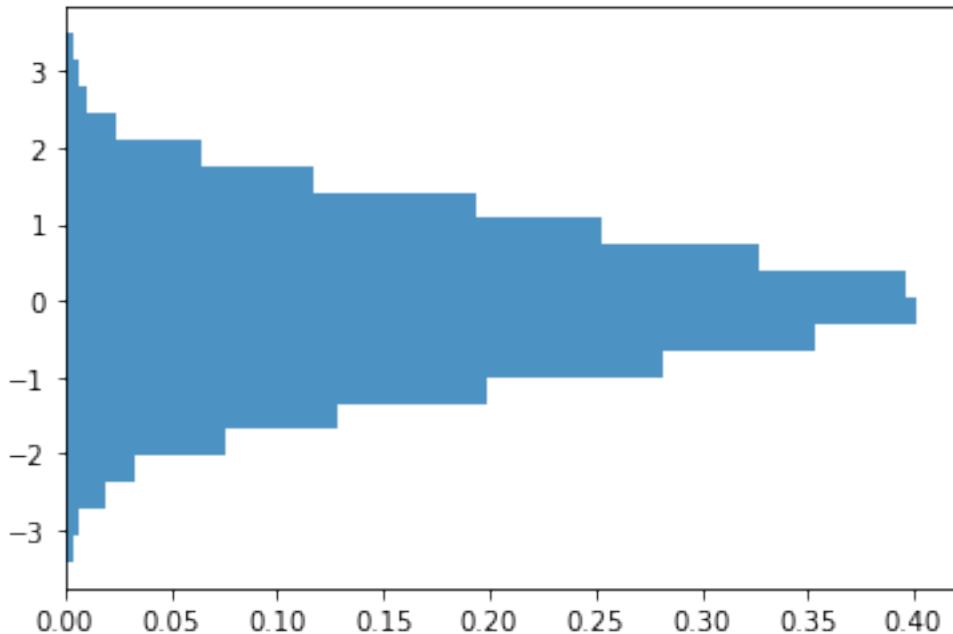


```
print(h1.size, b1.size)
```

```
20 21
```

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.

11.9.4 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

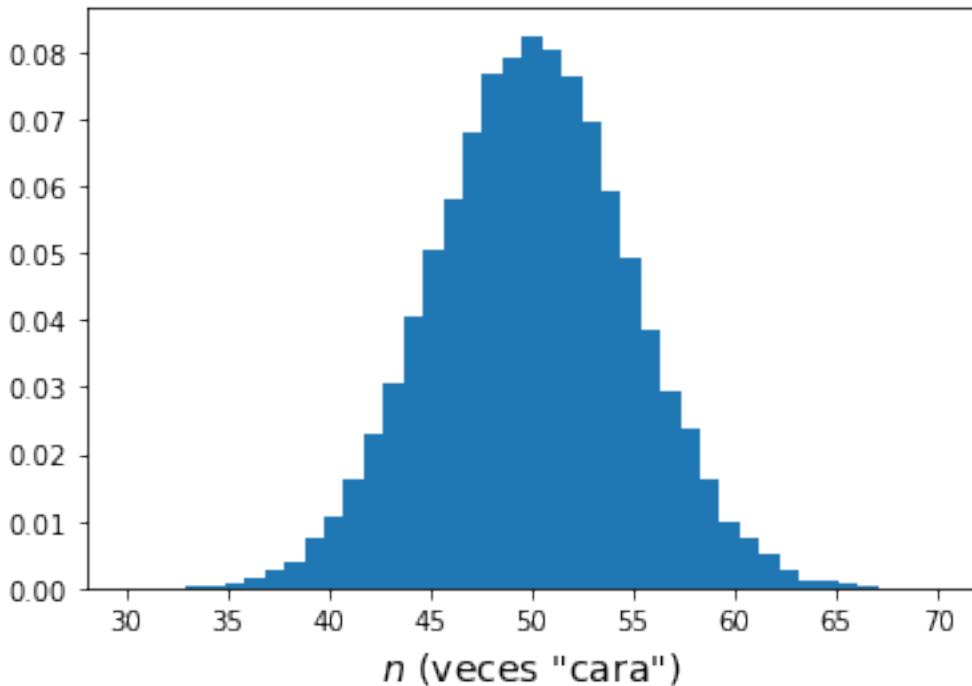
$$P(n) = \binom{N}{n} p^n (1-p)^{N-n},$$

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)
```

Este gráfico ilustra la probabilidad de obtener n veces un lado (cara) si tiramos 100 veces una moneda, como función de n .

11.10 Ejercicios 10 (b)

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:

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) dx$$

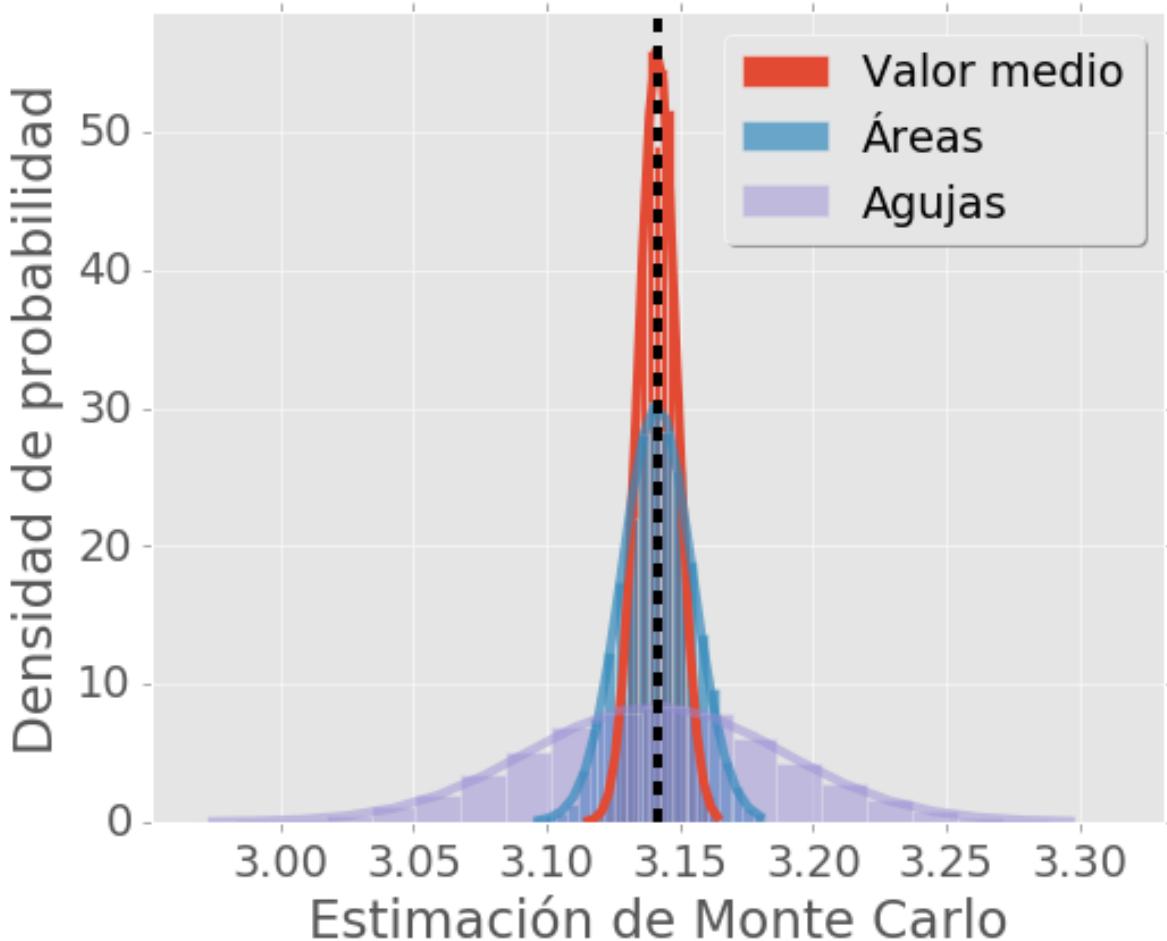
Tomando la función particular $f(x) = \sqrt{1-x^2}$ entre $x = 0$ y $x = 1$, obtenemos:

$$\langle f \rangle = \int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$

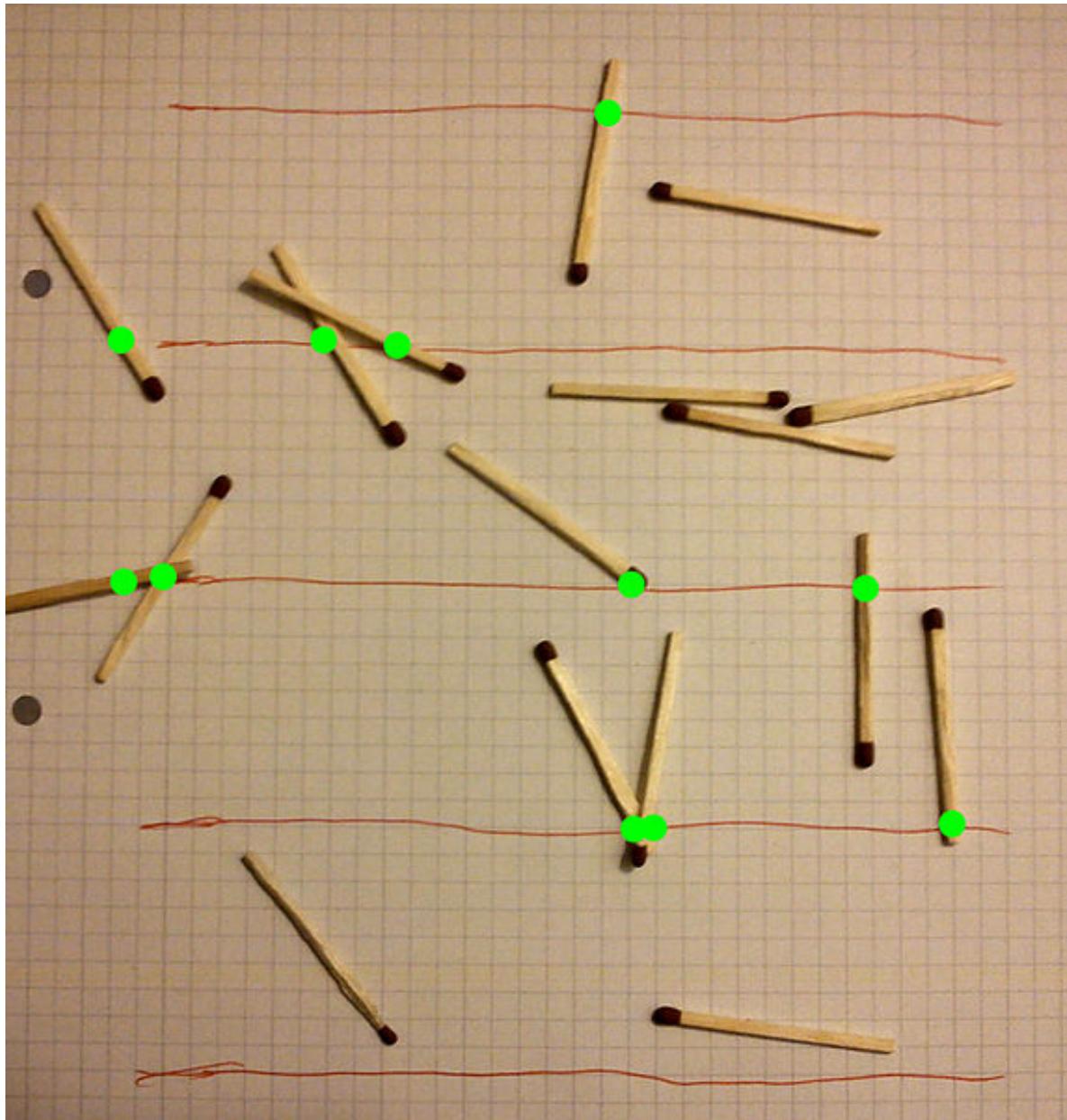
Entonces, tenemos que estimar el valor medio de la función f y, mediante la relación anterior obtener $\pi = 4\langle f(x) \rangle$. Para obtener el valor medio de la función notamos que si tomamos X es una variable aleatoria entre 0 y 1, entonces el valor medio de $f(X)$ es justamente $\langle f \rangle$. 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)

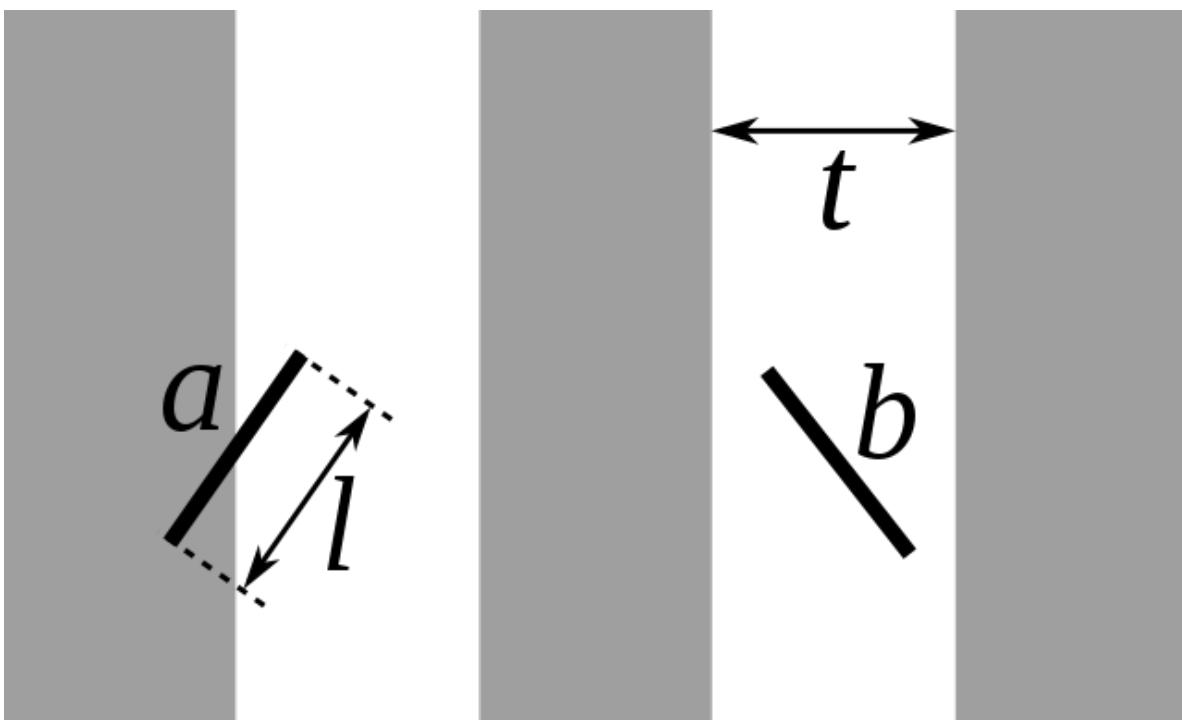
4. Para $N = 15000$ repetir el “experimento” muchas veces (al menos 1000) y realizar un histograma de los valores obtenidos para π con cada método. Graficar el histograma y calcular la desviación standard. Superponer una función Gaussiana con el mismo ancho. El gráfico debe ser similar al siguiente (*el 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 functions J_n .

Compute nt zeros of the Bessel functions $J_n(x)$ on the interval $(0, \infty)$. The zeros are returned in ascending order. Note that this interval excludes the zero at $x=0$ that exists for $n > 0$.

Parameters

n : int
 Order of Bessel function
nt : int
 Number of zeros to return

Returns

ndarray
 First n zeros of the Bessel function.

See Also

jv

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

Examples

```
>>> import scipy.special as sc
```

We can check that we are getting approximations of the zeros by evaluating them with jv.

```
>>> n = 1
```

```
>>> x = sc.jn_zeros(n, 3)
>>> x
array([ 3.83170597,  7.01558667, 10.17346814])
>>> sc.jv(n, x)
array([-0.00000000e+00,  1.72975330e-16,  2.89157291e-16])
```

Note that the zero at $x = 0$ for $n > 0$ is not included.

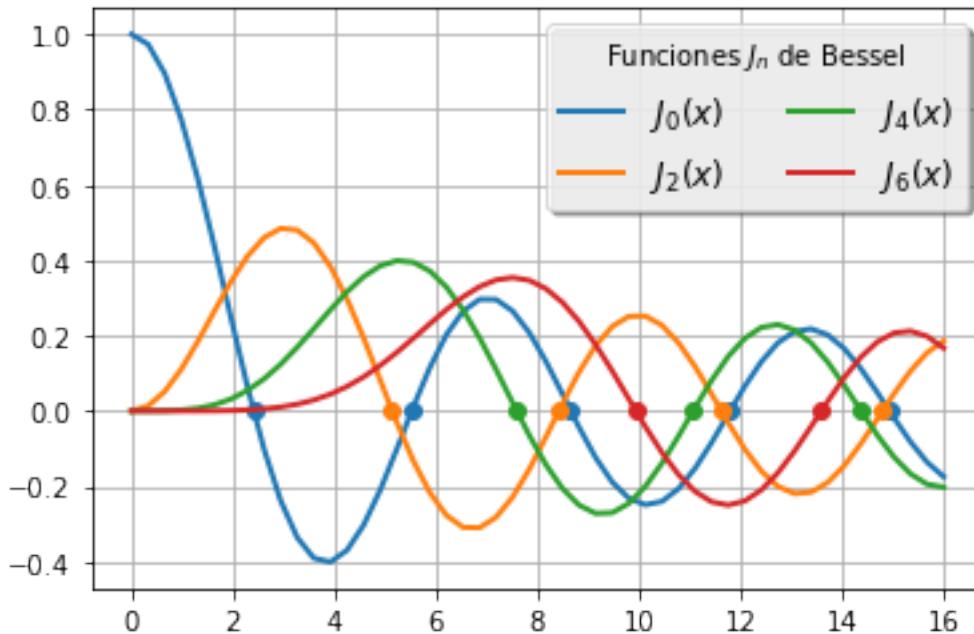
```
>>> sc.jv(1, 0)
0.0
```

```
# 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)
```

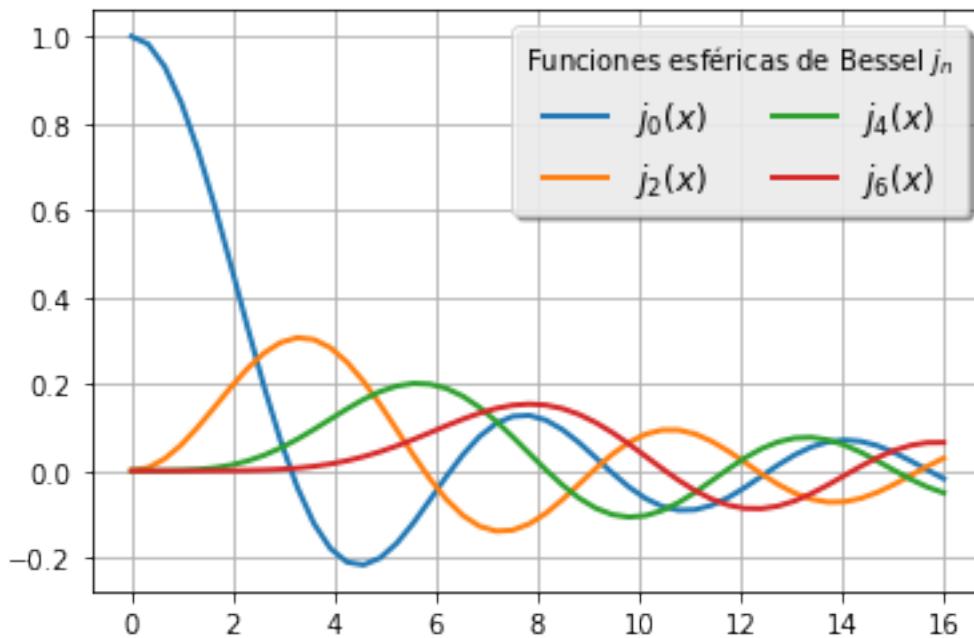
```
True  
True
```

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='$j_{\text{ }}{}_{\text{ }}\{\}(\text{x})$'.format(n))
plt.legend(title='Funciones esféricas de Bessel $j_n$', ncol=2);
plt.grid(True)
```



12.2.2 Función Error

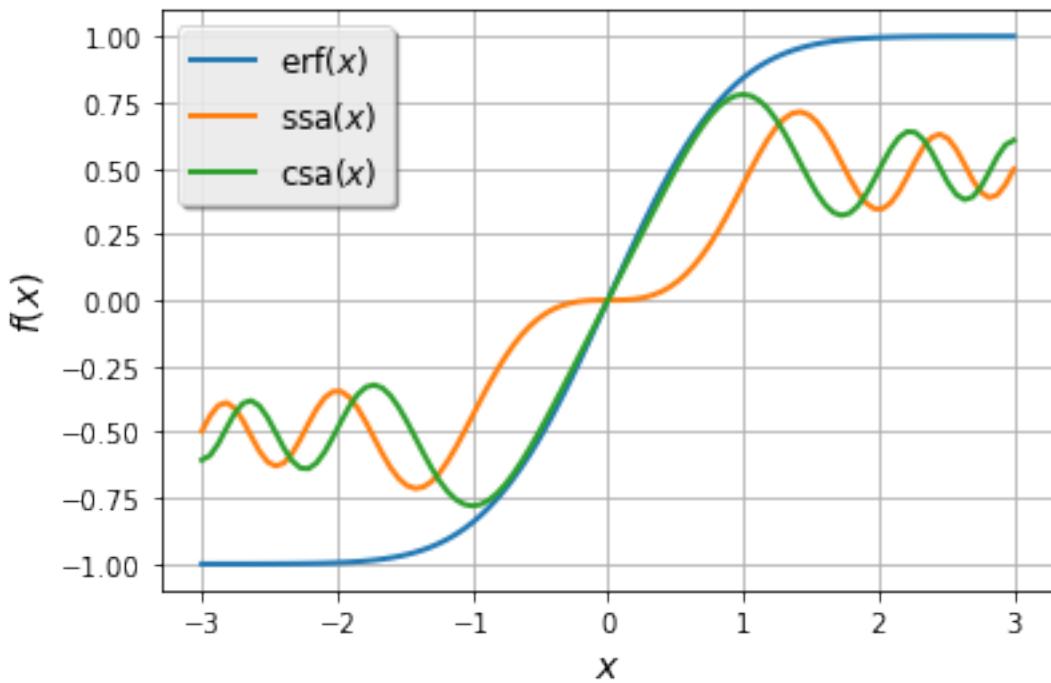
La función error es el resultado de integrar una función Gaussiana

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt,$$

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)
```



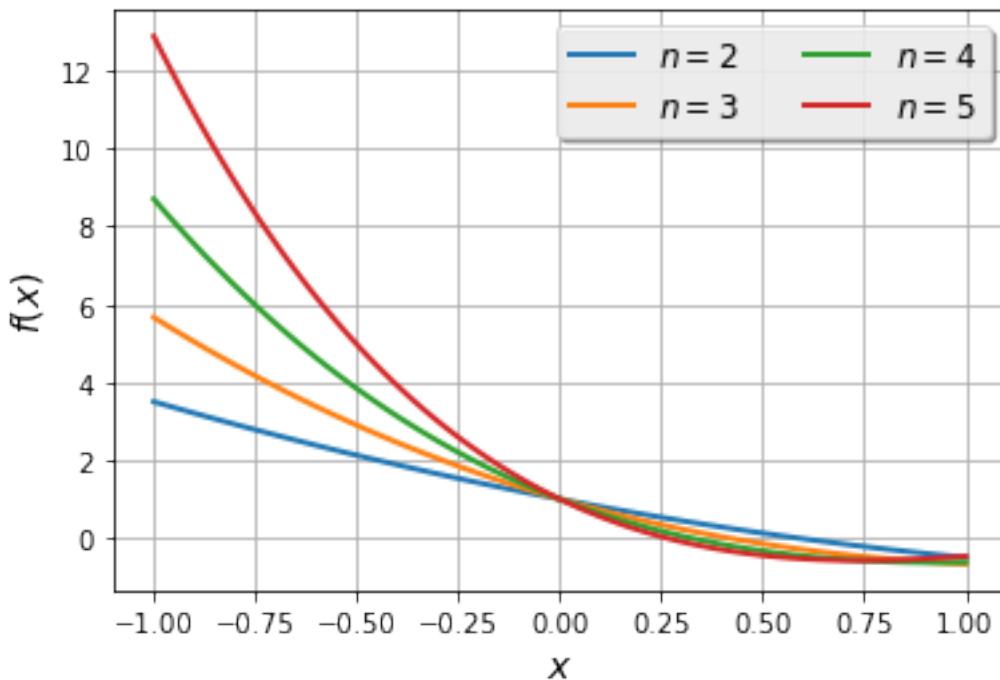
12.2.3 Evaluación de polinomios ortogonales

Scipy.special tiene funciones para evaluar eficientemente polinomios ortogonales

Por ejemplo si queremos, evaluar los polinomios de Laguerre, solución de la ecuación diferencial:

$$x \frac{d^2}{dx^2} L_n + (1-x) \frac{d}{dx} L_n + n L_n = 0$$

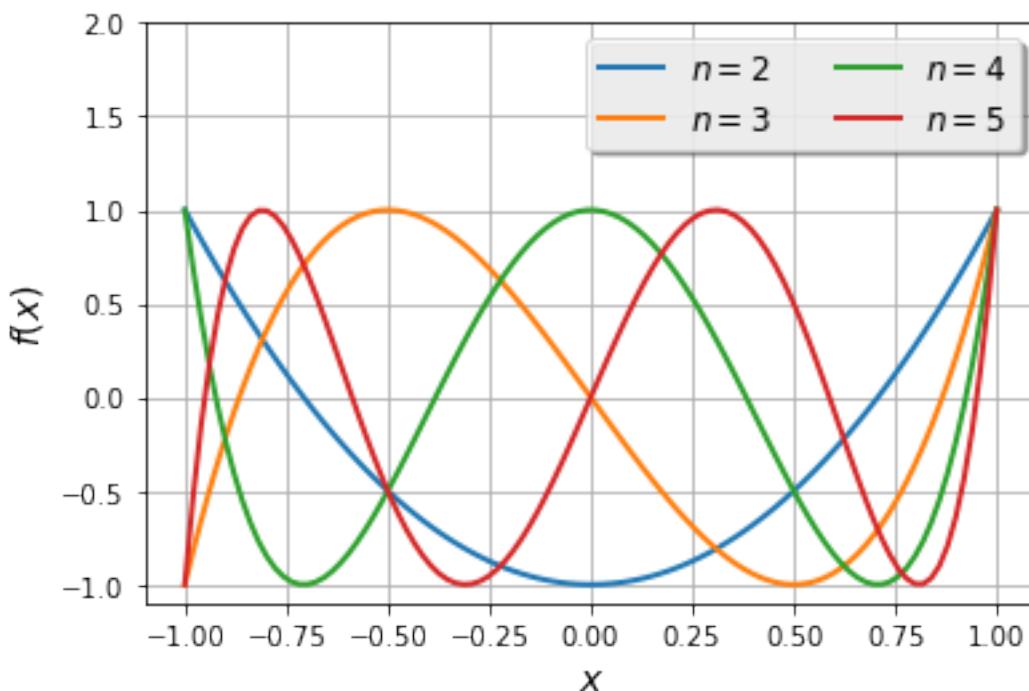
```
x = np.linspace(-1, 1, 100)
for n in range(2, 6):
    plt.plot(x, special.eval_laguerre(n, x), '-',
              label=r'$n={}$'.format(n))
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.legend(loc='best', ncol=2)
plt.grid(True)
```



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=f'$n={n}$')
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(f"{N}! = {special.factorial(N)}")
print(f"{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.

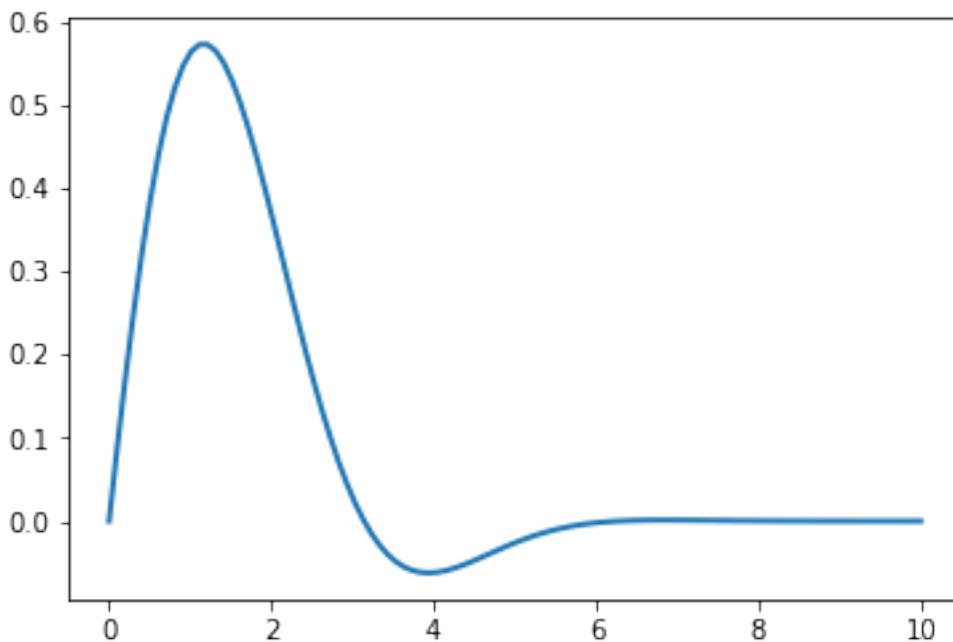
```
from scipy import integrate
```

```
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 0x7f0548a9fa90>]
```



```
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. Default is 1.49e-8. `quad` tries to obtain an accuracy of ``abs(i-result) <= max(epsabs, epsrel*abs(i))`` where ``i`` = integral of `func` from `a` to `b`, and ``result`` is the numerical approximation. See `epsrel` below.

epsrel : float or int, optional

Relative error tolerance. Default is 1.49e-8.

If ``epsabs <= 0``, `epsrel` must be greater than both 5e-29 and ``50 * (machine epsilon)``. See `epsabs` above.

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. Note that this option cannot be used in conjunction with ``weight``.

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
simpson : 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<=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 l  
    if |bb-aa| = |pts[2]-pts[1]| * 2**(-l).  
'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, and these do not support specifying break points. The possible values of *weight* and the corresponding weighting functions are.

=====	=====	=====
weight	Weight function used	wvar
=====	=====	=====
'cos'	cos(w*x)	wvar = w
'sin'	sin(w*x)	wvar = w
'alg'	g(x) = ((x-a)**alpha) * ((b-x)**beta)	wvar = (alpha, beta)
'alg-loga'	g(x)*log(x-a)	wvar = (alpha, beta)
'alg-logb'	g(x)*log(b-x)	wvar = (alpha, beta)
'alg-log'	g(x)*log(x-a)*log(b-x)	wvar = (alpha, beta)
'cauchy'	1/(x-c)	wvar = c
=====	=====	=====

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,...,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).
'rslst'
    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*floor(|w|) + 1) * pi / |w|
    and k=1,2,...,K_f.
'erlst'
    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))
#(1.33333333333333, 1.4802973661668752e-14)
print((1.0**3/3.0 + 1.0) - (0.0**3/3.0 + 0.0)) #Analytic result
# 1.33333333333333
```

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, 5.4599545822826244e-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)
```

```
<ipython-input-28-909b8a42d90d>:1: 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.
Iq= integrate.quad(f2,0,L)
```

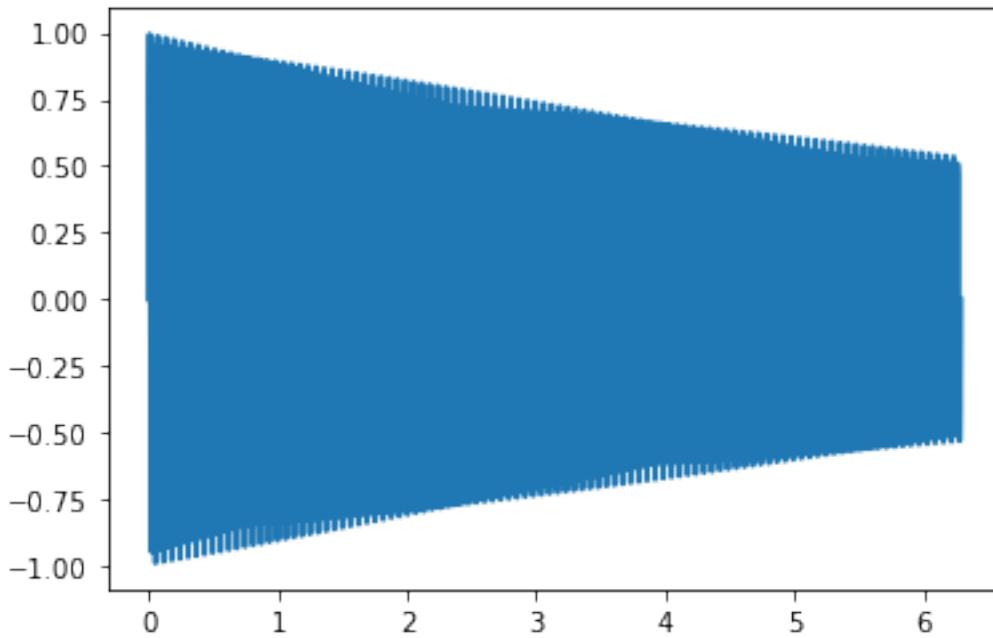
```
I_err = (I-Iq[0])/I          # Error relativo con el valor exacto
print("I= {:.5g} ± {:.5g}\nError relativo= {:.6g}\n".format(*Iq, I_err))
```

```
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 0x7f0548b0e710>]
```



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)
```

```
Is= integrate.quad(f3,0,L, weight='sin', wvar=k)
```

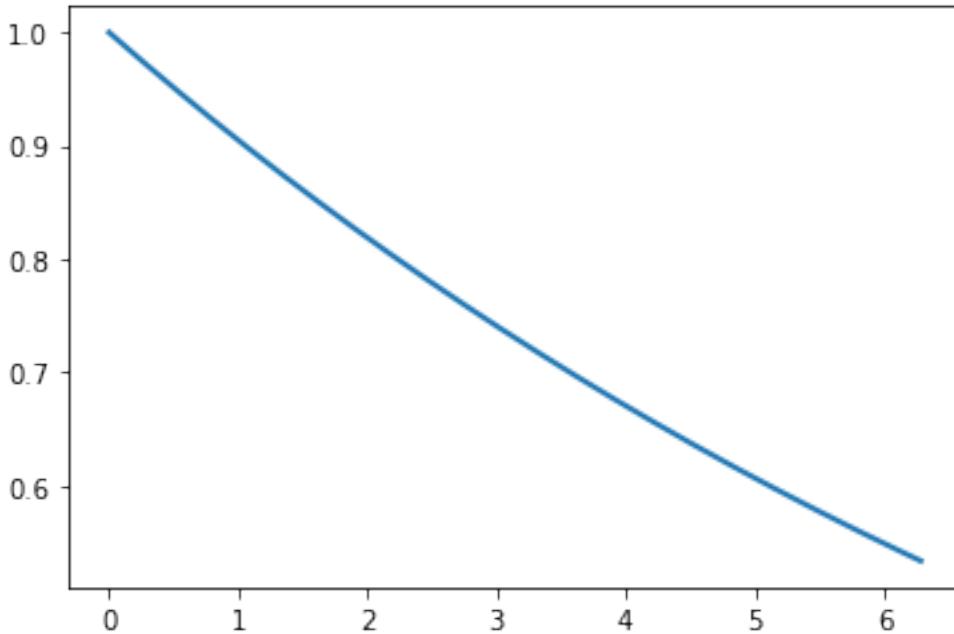
```
I_err = (I-Is[0])/I           # Error relativo con el valor exacto
print("I= {:.5g} ± {:.5g}\nError relativo= {:.6g}\n".format(*Is, I_err))
```

```
I= 0.0023326 ± 1.1788e-33
Error relativo= 5e-07
```

Esto es así, porque una vez que sepáramos el comportamiento oscilatorio, la función es suave y fácilmente integrable

```
plt.plot(x, f3(x))
```

```
[<matplotlib.lines.Line2D at 0x7f0548b7a4d0>]
```



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)
```

```
<magic-timeit>:1: 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.
```

```
3.34 ms ± 246 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit integrate.quad(f3, 0, L, weight='sin', wvar=k)
```

```
24.8 µs ± 785 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
```

Nota: Para calcular integrales múltiples existen rutinas que hacen llamados sucesivos a la rutina `quad()`. Esto incluye rutinas para integrales dobles (rutina `dblquad()`), triples (rutina `tplquad()`) y en general n-dimensionales (rutina `nquad()`)

12.4 Ejercicios 11 (a)

1. Graficar para valores de $k = 1, 2, 5, 10$ y como función del límite superior L , el valor de la integral:

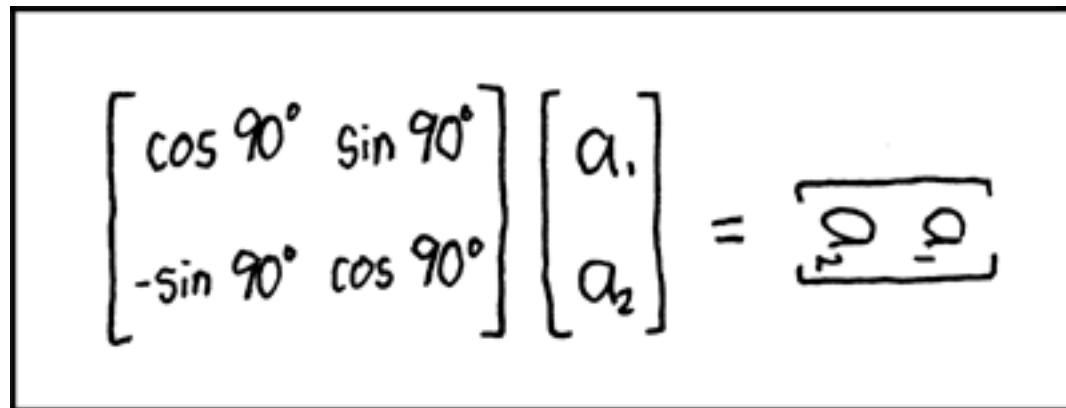
$$I(k, L) = \int_0^L x^k e^{-kx/2} \sin(kx) dx$$

con rango de variación de L entre 0 y 2π .

12.5 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)
```

```
A = np.array([[1, -2, -3], [1, -1, -1], [-1, 3, 1]])
print(A)
```

```
# La matriz transpuesta
A.T
```

12.5.1 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
```

```
linalg.norm(v) == np.sqrt(np.sum(np.square(v)))
```

Producto interno

El producto entre una matriz y un vector está definido en **Numpy** mediante las funciones `dot()`, o `matmul()`, o mediante el operador `@`:

```
w1 = np.dot(A, v)                      # Multiplicación de matrices  
w1  
  
np.allclose(np.dot(A, v), np.matmul(A, v))  # dot y matmul son equivalentes  
  
np.allclose(A @ v, np.matmul(A, v))        # También son equivalentes al operador @  
  
w2 = np.dot(v, A)  
w2  
  
np.dot(v.T, A) == np.dot(v, A)            # Si es unidimensional, el vector se transpone  
                                         ↪automáticamente  
  
print(v.shape, A.shape)
```

El producto interno entre vectores se calcula de la misma manera

$$\langle v, w \rangle$$

```
np.dot(v, w1)
```

y está relacionado con la norma

$$\|v\| = \sqrt{\langle v, v \rangle}$$

```
linalg.norm(v) == np.sqrt(np.dot(v, v))
```

```
np.dot(v, A)
```

```
v.shape
```

```
v2 = np.reshape(v, (3, 1))
```

```
v2.shape
```

```
np.dot(A, v2)
```

```
np.dot(A, v2).shape
```

Ahora las dimensiones de `v2` y `A` no coinciden para hacer el producto matricial

```
np.dot(v2, A)
```

```
np.dot(v2, A)
```

Notemos que el producto interno se puede pensar como un producto de matrices. En este caso, el producto de una matriz de 3x1, por otra de 1x3:

$$v^t w = \begin{pmatrix} -9 & -2 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$$

donde estamos pensando al vector como columna.

Producto exterior

El producto exterior puede ponerse en términos de multiplicación de matrices como

$$v \otimes w = vw^t = \begin{pmatrix} -9 \\ -2 \\ 4 \end{pmatrix} (2 \quad 1 \quad 3)$$

```
oprod = np.outer(v,w1)
print(oprod)
```

12.5.2 Aplicación a la resolución de sistemas de ecuaciones

Vamos a usar `scipy.linalg` para obtener determinantes e inversas de matrices. Veamos como 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')
```

```
x = np.dot(linalg.inv(A), b)
print('Resultado:\n', x)
```

12.5.3 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)
```

12.5.4 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))

```

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("Autovectores=" , np.real_if_close(v))

```

np.real_if_close?

12.5.5 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

```

`np.allclose(np.dot(a, x) , b)`

`np.dot(a,x) == b`

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)
```

```
%timeit np.dot(linalg.inv(A1),b1)
```

12.6 Entrada y salida de datos

12.6.1 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
```

```
X0 = np.loadtxt('../data/tof_signal_5.dat')
```

```
X0.shape, type(X0)
```

```
X0[0].shape
```

```
X0[0]
```

```
plt.plot(X0[:,0], X0[:,1])
```

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])
```

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)
```

```
Y = np.loadtxt(fdatos, skiprows=5, usecols=[0, 2])
print (Y.size, Y[0])
```

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-')
```

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:

```
np.fromstring(u"1.0 2.3    3.0 4.1    -3.1", sep=" ", dtype=float)
```

Para guardar datos en formato texto podemos usar, de la misma manera,

```
Y = np.vstack((x,y)).T
print(Y.shape)
```

```
np.savetxt('tmp.dat', Y)
```

```
!head tmp.dat
```

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
```

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])
```

```
print('`Alguna diferencia?', np.any(X1-X2))
```

12.6.2 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')
    plt.ylabel('Frecuencia')

    labels = ['A', 'Z', 'a', 'o', 'z', 'á', 'ú']
    ll = [r'$\mathbf{{}}$'.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")
                     )
```

12.6.3 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'])
```

```
data
```

12.7 Ejercicios 11 (b)

2. 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.
3. 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').
4. 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.
5. 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:
 6. la distribución de ocurrencias del primer parámetro.
 7. la distribución de ocurrencias de energías de la tercera partícula.
 8. la distribución de ocurrencias de ángulos de la cuarta partícula, medido respecto al tercer eje.
 9. 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`.

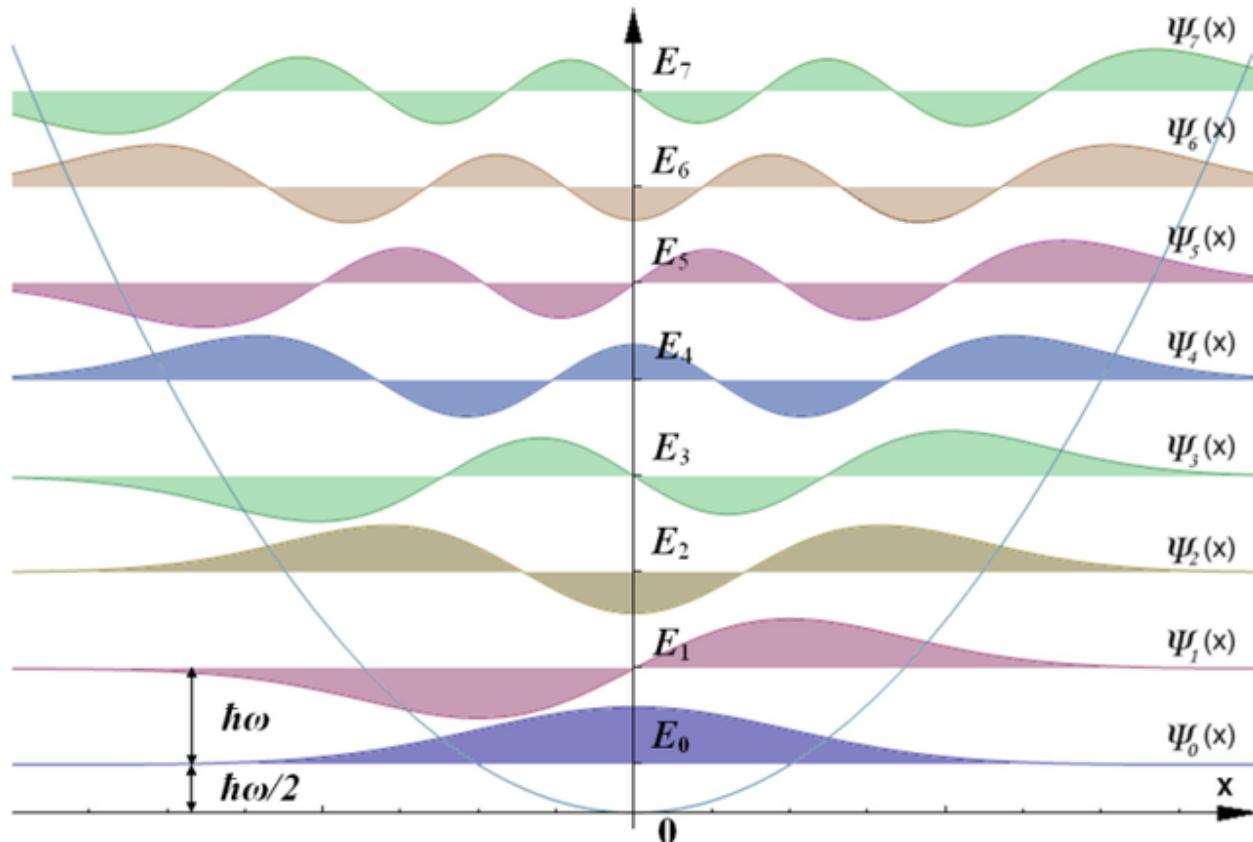
7. Usando **Scipy** y **Matplotlib** grafique las funciones de onda del oscilador armónico unidimensional para las cuatro energías más bajas ($n = 1, 2, 3, 4$), en el intervalo $[-5, 5]$. Asegúrese de que están correctamente normalizados.

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 plotly.graph_objects as go
```

13.1 Gráficos y procesamiento sencillo en 2D

13.1.1 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 = 10000
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
```

Para crear el histograma usamos simplemente la función `Histogram2d`.

```
H= go.Figure(go.Histogram2d(
    x = x
    ,y = y))
H.show()
```

```
H = go.Figure(go.Histogram2d(
    x = x
    ,nbinsx = 60
```

(continué en la próxima página)

(proviene de la página anterior)

```
,y = y  
,nbinsy = 60)  
H.show()
```

Aquí pusimos igual número de “cajas” en cada dimensión. También podemos pasárle un array con distinto número de cajas

```
H = go.Figure(go.Histogram2d(  
    x = x  
    ,nbinsx = 60  
    ,y = y  
    ,nbinsy = 150))  
H.show()
```

Por supuesto podemos cambiar el esquema de colores utilizado. Para ello le damos explícitamente el argumento `cmap` especificando el “colormap” deseado:

Se puede definir el número de bins de esta otra forma:

```
H = go.Figure(go.Histogram2d(  
    x = x  
    ,autobinx=False  
    ,xbins=dict(start=-5, end=7.5, size=0.5)  
    ,y = y  
    ,nbinsy = 60))  
H.show()
```

Por supuesto podemos cambiar el esquema de colores utilizado. Para ello le damos explícitamente el argumento `cmap` especificando el “colormap” deseado:

```
H = go.Figure(go.Histogram2d(  
    x = x  
    ,autobinx=False  
    ,xbins=dict(start=-5, end=7.5, size=0.5)  
    ,y = y  
    ,nbinsy = 60  
    ,colorscale='YlGnBu'  
    ))  
H.show()
```

Se puede elegir el valor máximo de Z:

```
H = go.Figure(go.Histogram2d(  
    x = x  
    ,autobinx=False  
    ,xbins=dict(start=-5, end=7.5, size=0.5)  
    ,y = y  
    ,nbinsy = 60  
    ,zmax = 200  
    ,zauto = False  
    ,colorscale='YlGnBu'  
    ))  
H.show()
```

O por ejemplo, se puede agregar el valor de cada bin:

```
H = go.Figure(go.Histogram2d(
    x = x
    ,nbinsx = 30
    ,y = y
    ,nbinsy = 30
    ,colorscale='YlGnBu'
    ,texttemplate= "%{z}"
))
H.show()
```

13.1.2 Gráficos de contornos

```
Z,xedges,yedges = np.histogram2d(x,y,bins=20)
```

```
X, Y = np.meshgrid(xedges, yedges)
```

```
X.shape, Y.shape, Z.shape
```

```
np.all(X[0] == X[1])
```

```
np.all(Y[0] == Y[1])
```

```
np.all(Y[:,0] == Y[:,2])
```

```
X[0,:10], X[1,:10]
```

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:

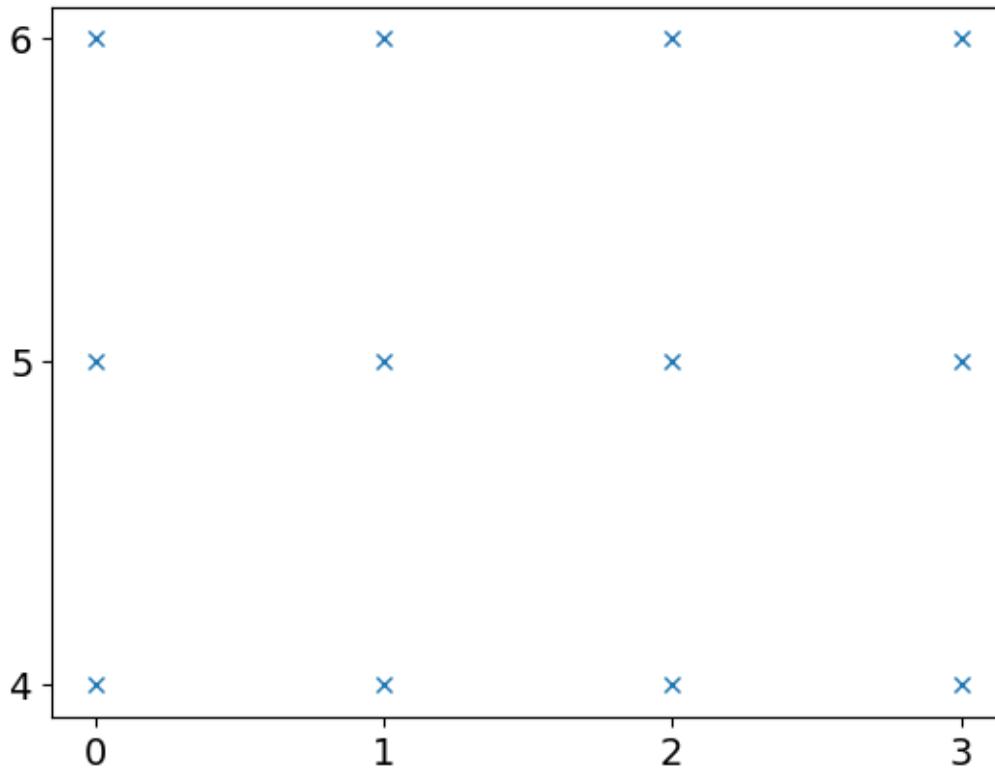
```
x = [0 1 2 3]
y = [4 5 6]
```

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= go.Figure(data =
    go.Contour(
        x = xedges
        ,y = yedges
        ,z = Z))
fig1.show()
```

```
fig1= go.Figure(data =
    go.Contour(
        x = xedges
        ,y = yedges
        ,z = Z
        ,colorscale='rainbow'))
fig1.show()
```

```
fig1= go.Figure(data =
    go.Contour(
        x = xedges
        ,y = yedges
        ,z = Z
        ,contours = dict(
            coloring='lines'
            ,showlabels=True
```

(continué en la próxima página)

(provine de la página anterior)

```

        ,labelfont = dict(
            size = 11,
        #
            color = 'white',
        ))
#
    ,line_smoothing=0.5
    ,contours_coloring='heatmap' # can also be 'lines', or 'none'

))
fig1.show()

```

También podemos mostrar la imagen con los contornos superpuestos:

13.1.3 Superficies y contornos

Superficies

Realizar gráficos “realmente” en 3D es tan simple como cambiar Contour por Surface

```

fig1= go.Figure(data =
    go.Surface(
        x = xedges
        ,y = yedges
        ,z = Z))
fig1.show()

```

```

fig1= go.Figure(data =
    go.Surface(
        x = xedges
        ,y = yedges
        ,z = Z
        ,opacity=0.7))
fig1.show()

```

```
fig1.write_html('3dplot.html')
```

Contornos en 3D

```

fig1= go.Figure(data =
    go.Surface(
        x = xedges
        ,y = yedges
        ,z = Z
        ,contours = {
            "z": {"show": True}
            , "x": {"show": True, "color": "white"}
        }))
fig1.show()

```

```

fig1= go.Figure(data =
    go.Surface(
        x = xedges
        ,y = yedges

```

(continué en la próxima página)

(provine de la página anterior)

```
, z = Z
, contours = {
    "z": {"show": True, "start": 0, "end": 600, "size":50}
    , "x": {"show": True, "color": "white"}
})
fig1.update_traces(contours_z=dict(show=True, usecolormap=True,
highlightcolor="limegreen", project_z=True))
fig1.show()
```

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

Veamos un ejemplo de la documentación de `Matplotlib`

```
import pandas as pd

df = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/vortex.csv
")

fig = go.Figure(data = go.Cone(
    x=df['x'],
    y=df['y'],
    z=df['z'],
    u=df['u'],
    v=df['v'],
    w=df['w'],
    colorscale='Blues',
    sizemode="absolute",
    sizeref=40))

fig.update_layout(scene=dict(aspectratio=dict(x=1, y=1, z=0.8),
    camera_eye=dict(x=1.2, y=1.2, z=0.6)))
fig.show()
```

```
# Make the grid
x = np.arange(0, 1, 0.2)
y = np.arange(0, 1, 0.2)
z = np.arange(0, 1, 0.2)

# 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 * y) #* np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0)* np.sin(np.pi * z)) #* np.cos(np.pi * x) * np.cos(np.pi * y)_
#* np.sin(np.pi * x) * np.sin(np.pi * y)

fig = go.Figure(
    go.Cone(
        x = x
        ,y = y
        ,z = z
        ,u = u
```

(continué en la próxima página)

(proviene de la página anterior)

```
, v = v
,w = w
,sizemode="absolute"
,sizeref=2
,anchor="tip"
))
fig.show()
```

Streamtubes of the ABC Flow: Flowing from Y-Plane

```
def vector_field(x, y, z, A=1, B=np.sqrt(2./3), C=np.sqrt(1./3)):
    return A*np.sin(z) + C*np.cos(y), B*np.sin(x) + A*np.cos(z), C*np.sin(y) + B*np.
    ↪cos(x)
```

```
x, y, z=np.mgrid[0: 2*np.pi:30j, 0:2*np.pi:30j, 0:2*np.pi:30j]
u, v, w=vector_field(x, y, z)
```

```
x.shape
u.shape
```

```
fig = go.Figure(
    go.Streamtube(
        x=x.flatten(),
        y=y.flatten(),
        z=z.flatten(),
        u=u.flatten(),
        v=v.flatten(),
        w=w.flatten(),
        maxdisplayed=3500,
        sizeref=0.3,
        reversescale=True,
        showscale=True,
    )
)
fig.show()
```


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**.

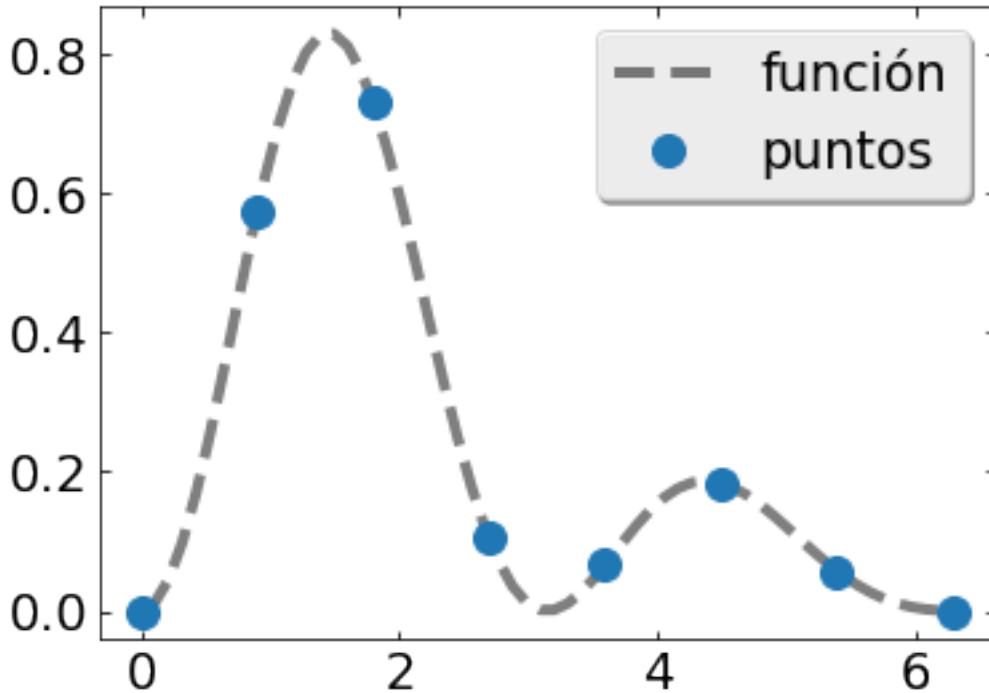
Generemos algunos “datos experimentales”

```
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 0x7f211a1021d0>
```



Acá hemos simulado datos con una función oscilante con un decaimiento exponencial.

Ahora, importamos el submódulo `interpolate` del módulo `scipy`, que nos permite interpolar los datos:

```
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 los valores en los nuevos puntos sobre el eje x que necesitamos.

Utilizamos los *arrays* `x` e `y` como los pocos “datos experimentales” obtenidos

```
print(f" {x = }")
```

```
x = array([0.          , 0.8975979 , 1.7951958 , 2.6927937 , 3.5903916 ,
4.48798951, 5.38558741, 6.28318531])
```

y creamos la función interpolante basada en estos puntos:

```
interp_lineal = interpolate.interp1d(x, y)
```

```
interp_lineal # función
```

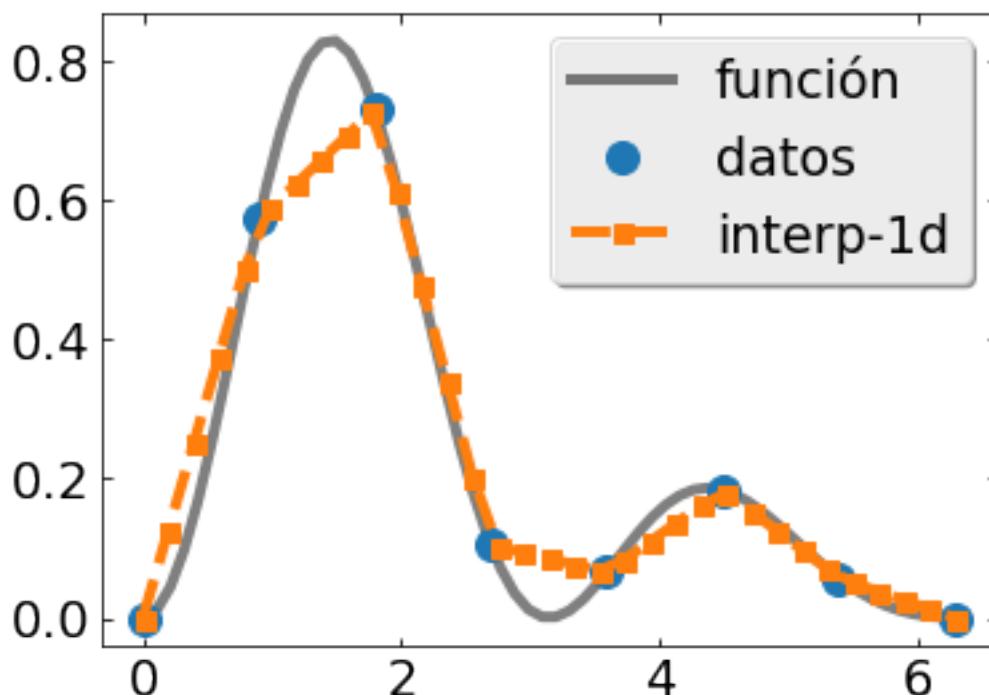
```
<scipy.interpolate.interpolate.interp1d at 0x7f20d013c540>
```

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 = interp_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 0x7f20d00db8b0>
```



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))
```

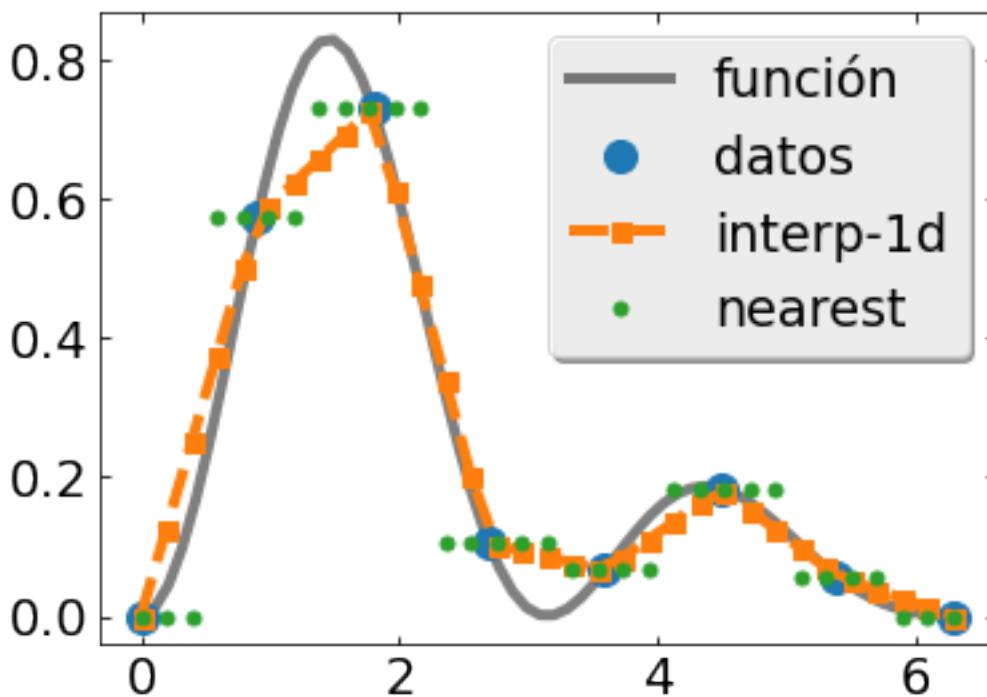
```
array([0.00000000e+00, 3.64223643e-01, 6.15515285e-01, 7.16230928e-01,
       3.88910634e-01, 9.71667086e-02, 7.27120078e-02, 1.19301459e-01,
       1.72109481e-01, 9.17229560e-02, 3.64455561e-02, 2.39038977e-33])
```

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:

```
def fm(x):
    return x**4 - x**3*np.sin(x/6)

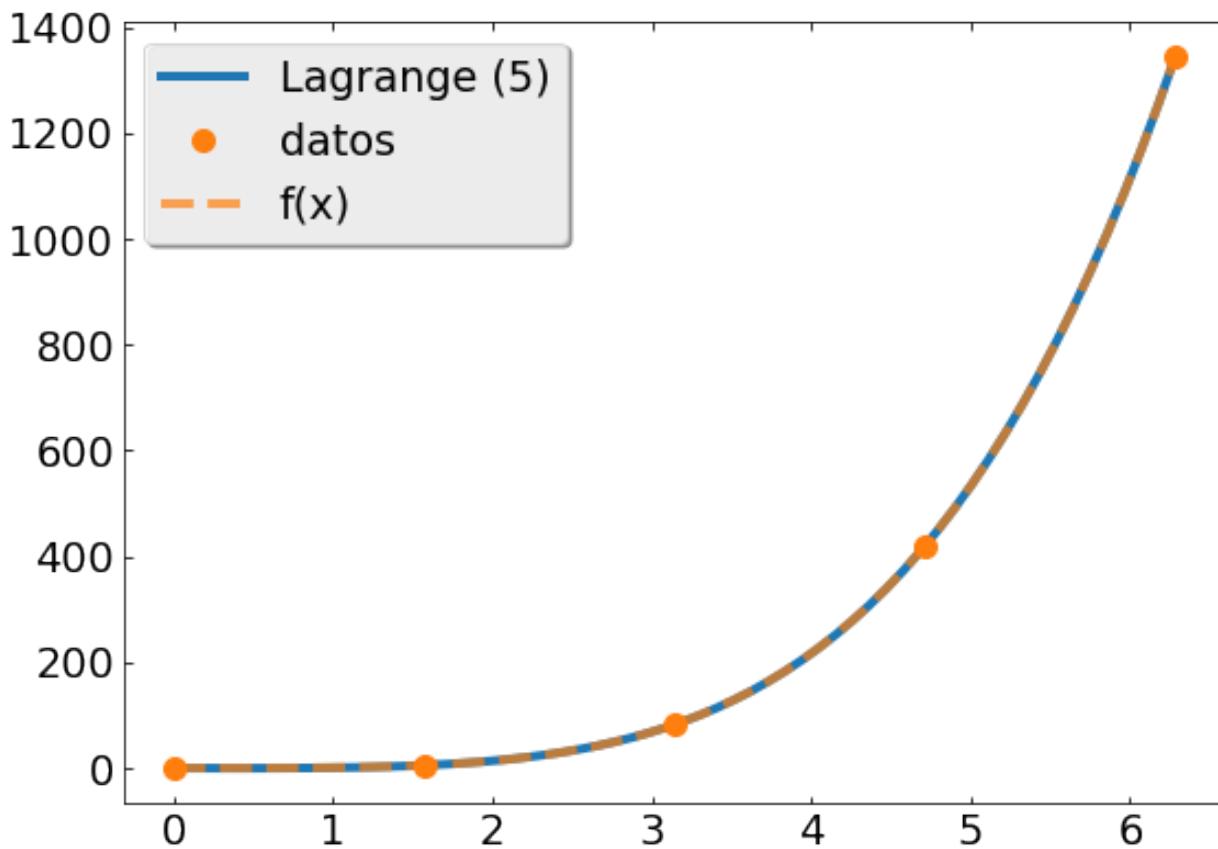
x0 = np.linspace(0., 2*np.pi, 60)
y0 = fm(x0)
x = np.linspace(0., 2*np.pi, 5)
y = fm(x)
#
# Creamos el polinomio interpolador
f = interpolate.lagrange(x, y)
y1 = f(x0)
#
```

(continué en la próxima página)

(provieniente de la página anterior)

```
plt.figure(figsize=fsize)
plt.plot(x0,y1,'-', label=f'Lagrange ({x.size})')
plt.plot(x,y,'o', label='datos')
plt.plot(x0,y0,'--', color='C1', label='f(x)', alpha=0.7)
plt.legend(loc='best')
```

<matplotlib.legend.Legend at 0x7f20cf07340>



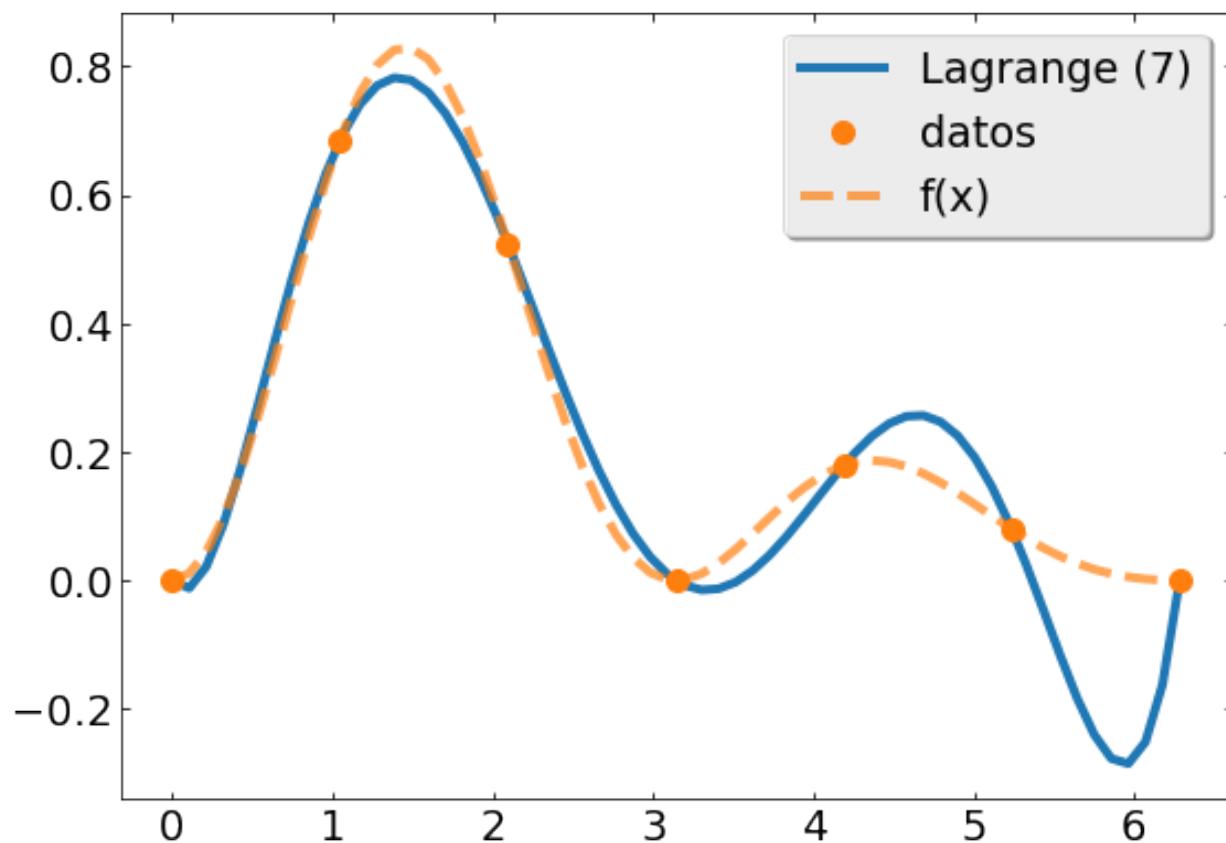
Acá la función `lagrange()` devuelve un polinomio del tipo `poly1d`, que puede ser ejecutado (como hicimos en la celda anterior)

`type(f)``numpy.poly1d``f.coeffs``array([0.9463431 , -0.80568571, 1.98841541, -1.56609363, 0.])`

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 `x1 f(x1)` con muchos datos y queremos interpolar sobre una nueva tabla de valores `x0`. Incluso para un grado intermedio de polinomio se observan oscilaciones entre los puntos

```
x0 = np.linspace(0., 2*np.pi, 60)
x1 = np.linspace(0, 2*np.pi, 7)
y1 = fmodel(x1)
f1 = interpolate.lagrange(x1, y1)
plt.figure(figsize=fsiz)
plt.plot(x0,f1(x0),'-', label=f'Lagrange ({x1.size})')
plt.plot(x1,y1,'o', label='datos')
plt.plot(x0,fmodel(x0), '--', color='C1', label='f(x)', alpha=0.7)
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x7f20cfb225c0>
```



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

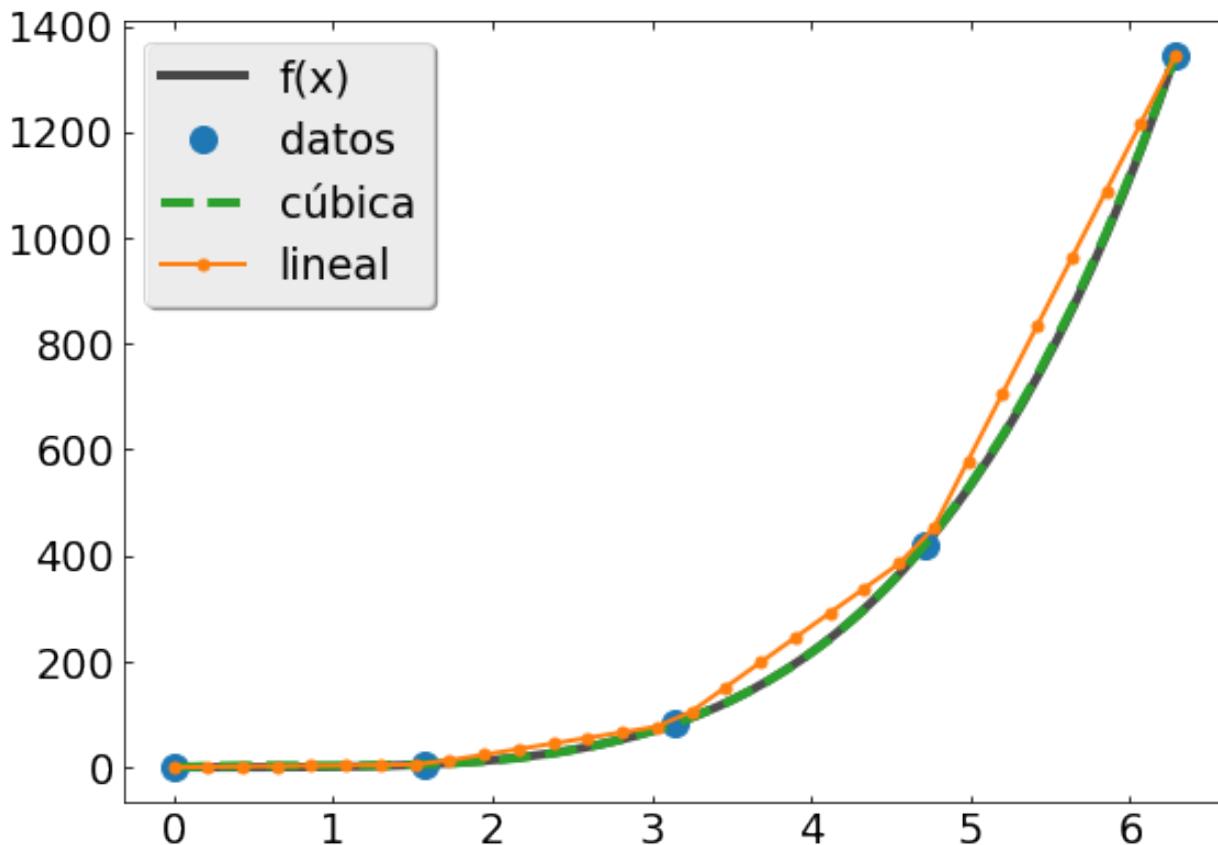
- Las *Splines* son interpolaciones por polinomios de trazos, que se eligen para que no sólo los valores sino también sus derivadas coincidan dando una curva suave.
- Para eso, si se pide que la aproximación coincida con los valores tabulados en los puntos dados, la aproximación es efectivamente una **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.

```
x1 = np.linspace(0, 2*np.pi, 30)
interp = {}
for k in ['zero', 'slinear', 'quadratic', 'cubic']:
    interp[k] = interpolate.interp1d(x, y, kind=k)
```

```
fig = plt.figure(figsize=fsize)
plt.plot(x0,y0,'-k', alpha=0.7, label='f(x)')
plt.plot(x,y,'o', markersize=12, label='datos')
plt.plot(x1(interp['cubic'](x1), '--', color='C2', label=u'cúbica')
plt.plot(x1(interp['slinear'](x1), '-.', lw=2, label='lineal')
plt.legend(loc='best')
```

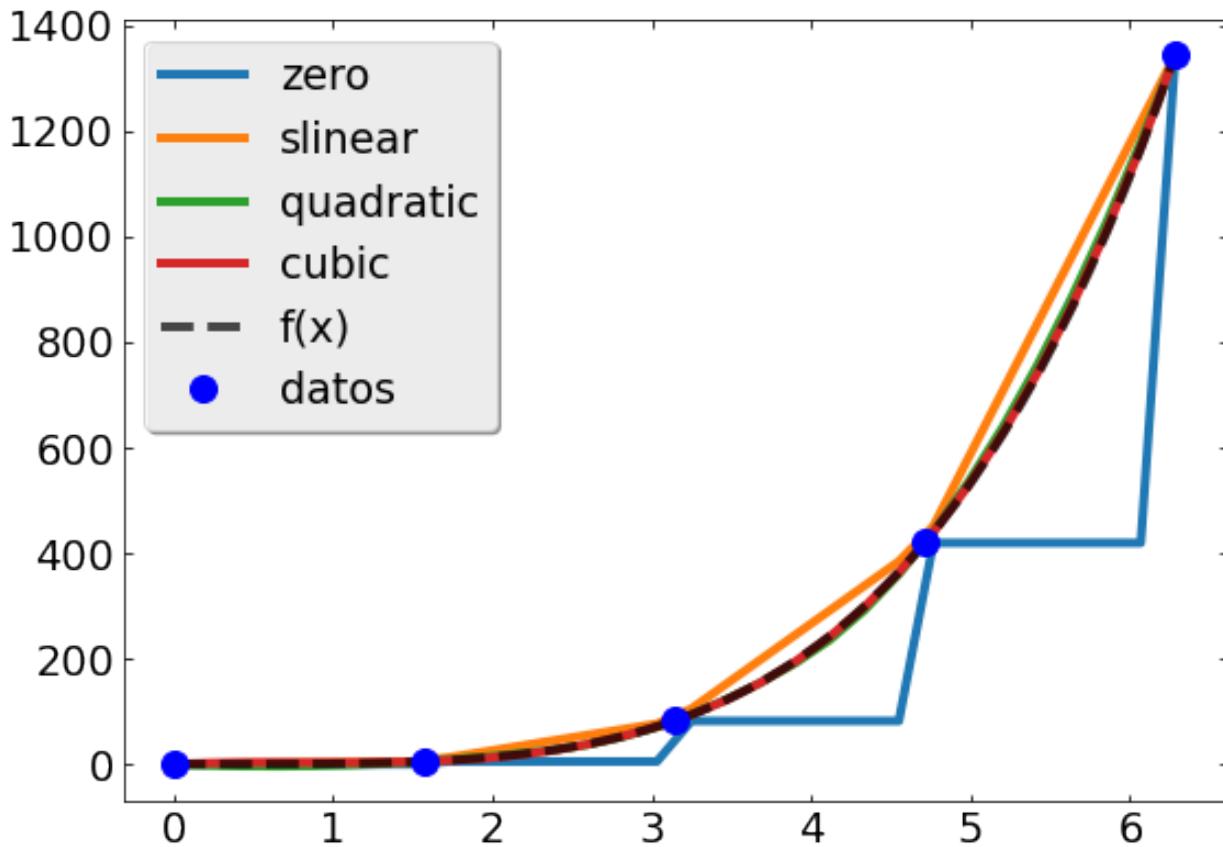
```
<matplotlib.legend.Legend at 0x7f20cfcc5d630>
```



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 0x7f20cf8454b0>
```



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(f"?`{k} == {s}? -> {np.allclose(num(x1), tipo(x1))}")
```

```
?`0 == zero? -> True
?`1 == slinear? -> True
?`2 == quadratic? -> True
?`3 == cubic? -> True
```

Además La interpolación lineal simple es, en la práctica, igual a la interpolación por splines de primer orden:

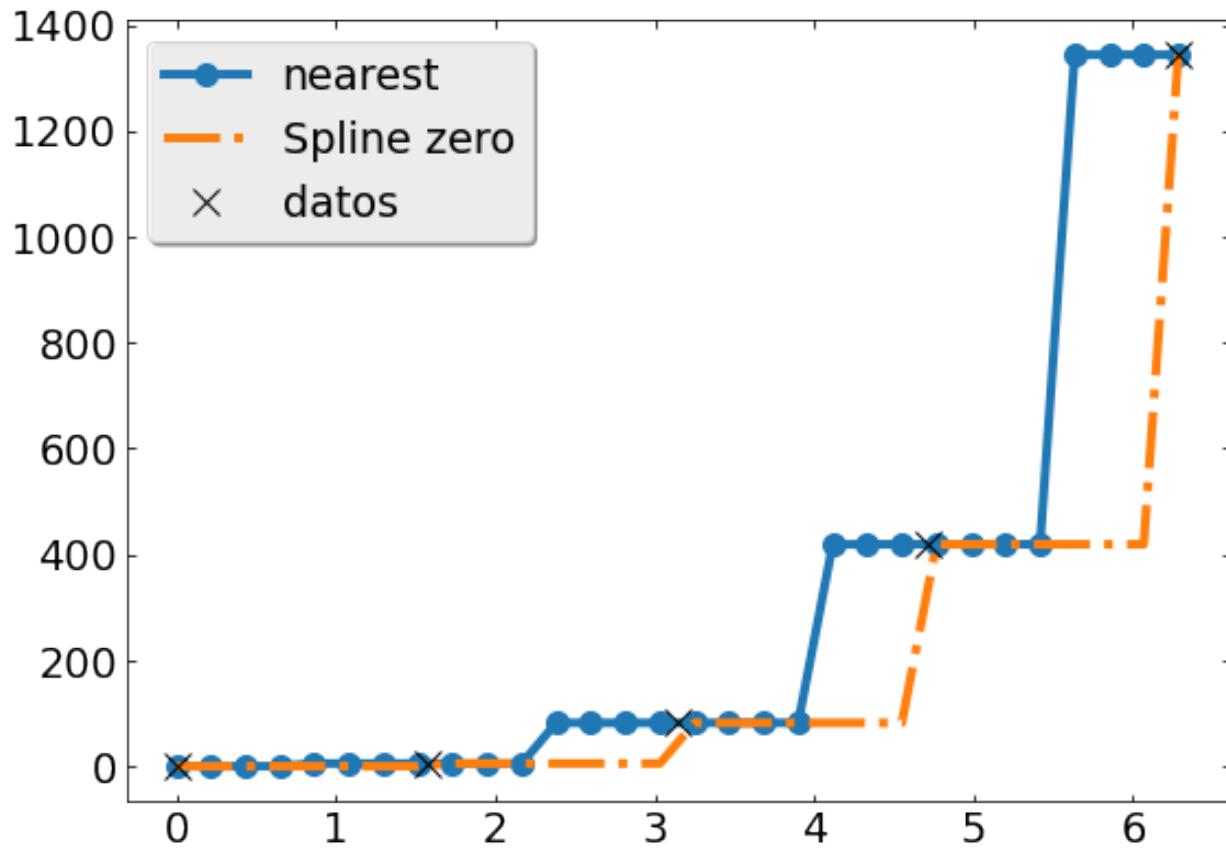
```
interpol_lineal = interpolate.interp1d(x, y)
np.allclose(interp['slinear'](x1), interpol_lineal(x1)) # También son iguales
```

```
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 del dato**:

```
interpol_near = interpolate.interp1d(x, y, kind='nearest')
alfa=1
plt.figure(figsize=fsiz)
plt.plot(x1, interpol_near(x1), '-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');
```



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:

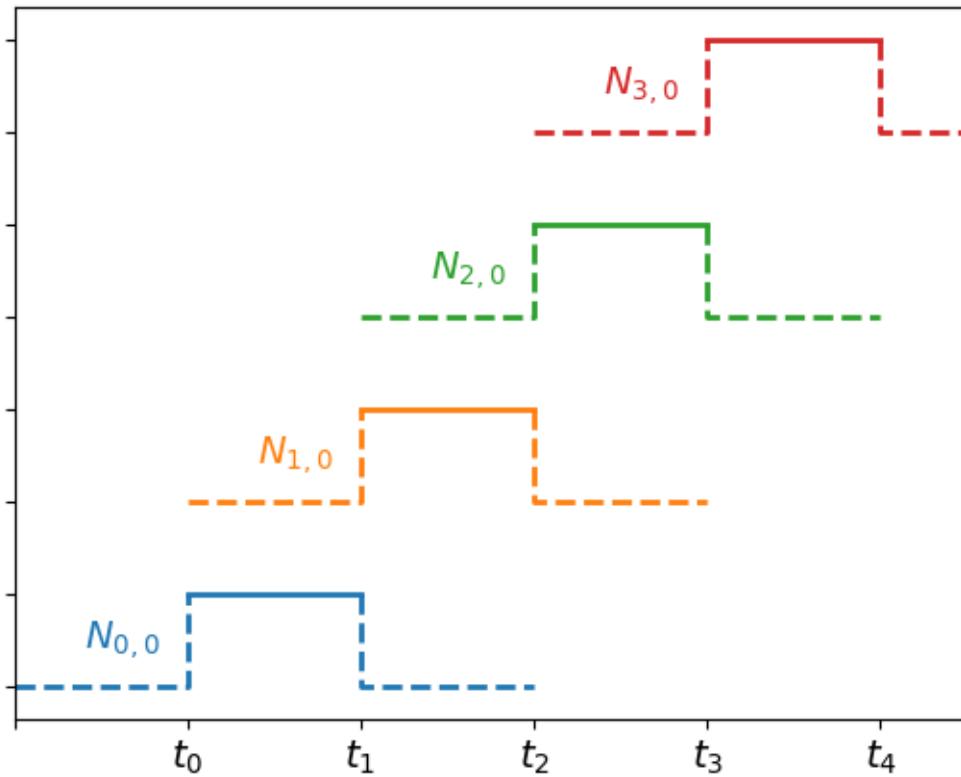
1. Elegir el grado de los polinomios (mayor o igual a 0)
2. Dividir el intervalo en n “nodos”

Las funciones se definen mediante la recursión:

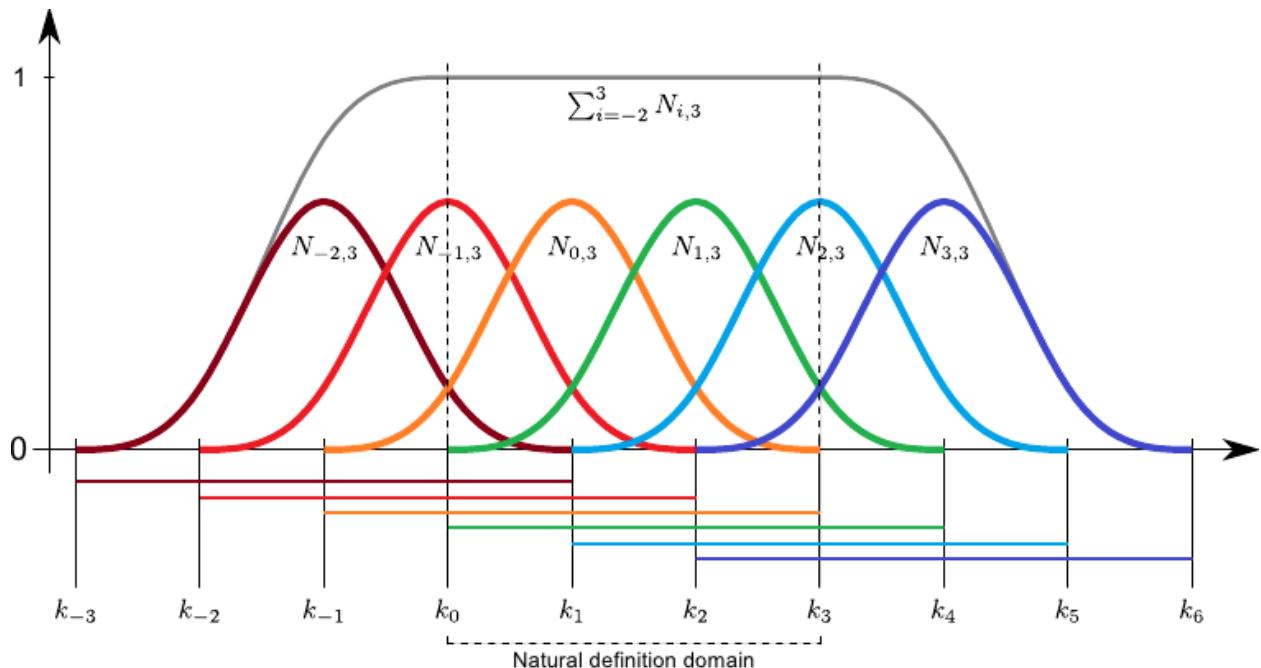
$$N_{i,0}(x) = 1, \quad \text{si } t_i \leq x < t_{i+1}, \quad \text{sino } 0,$$

$$N_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} N_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} N_{i+1,k-1}(x)$$

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*)

$$f(x) = \sum_{j=0}^n a_{i,j} N_j(x) \quad \forall \quad x_i < x \leq x_{i+1}$$

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) :

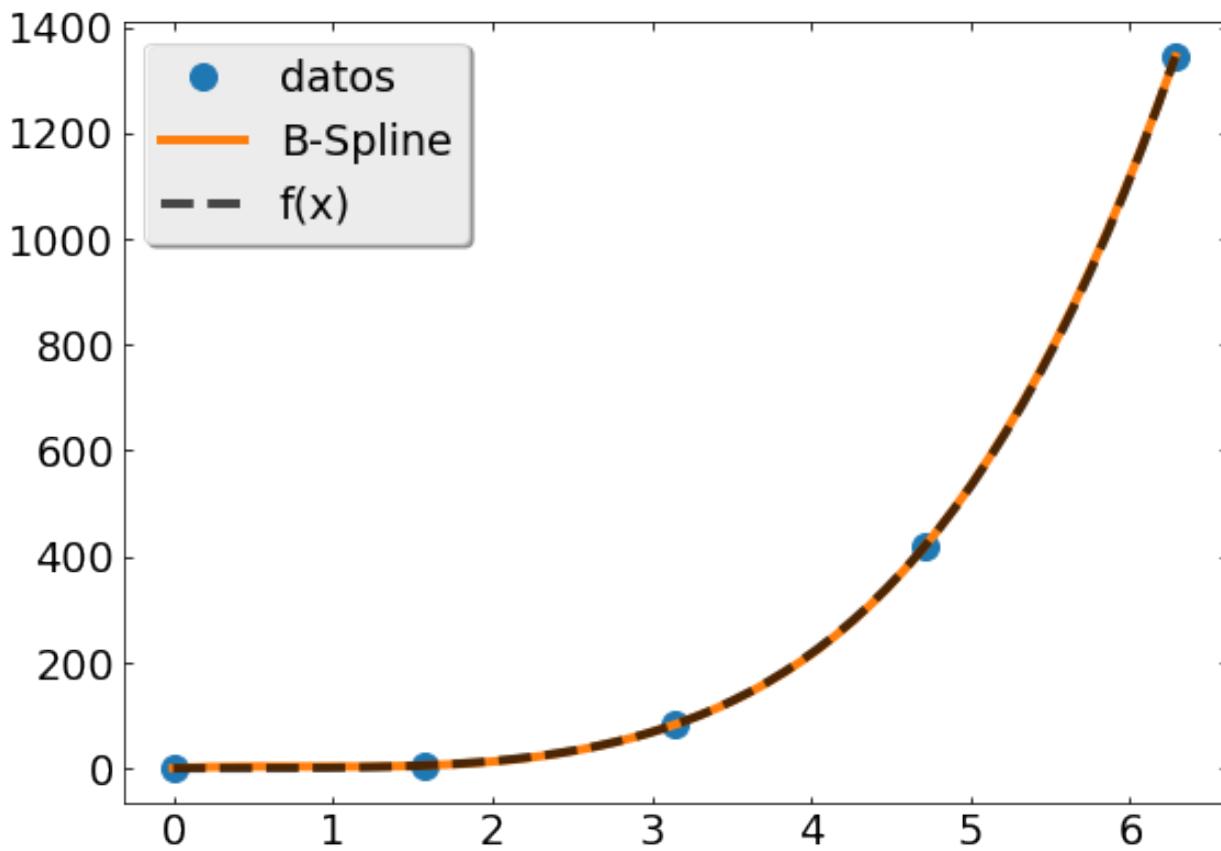
```
tck0 = interpolate.splrep(x, y)
```

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.

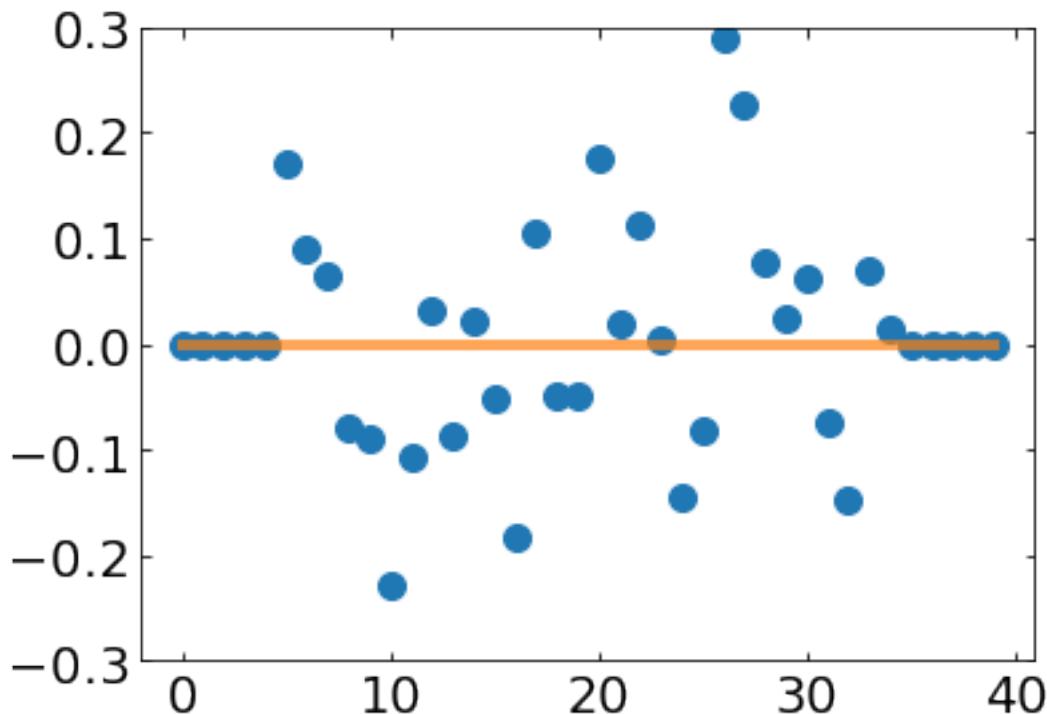
Para ver como funciona, creamos una tabla de valores `x`, `y` con $x \in [0, 2\pi]$ **no necesariamente equiespaciados**, $y = \sin(x)/2$, donde le agregamos algo de ruido a `y`

```
# 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] -= 0.7*(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)

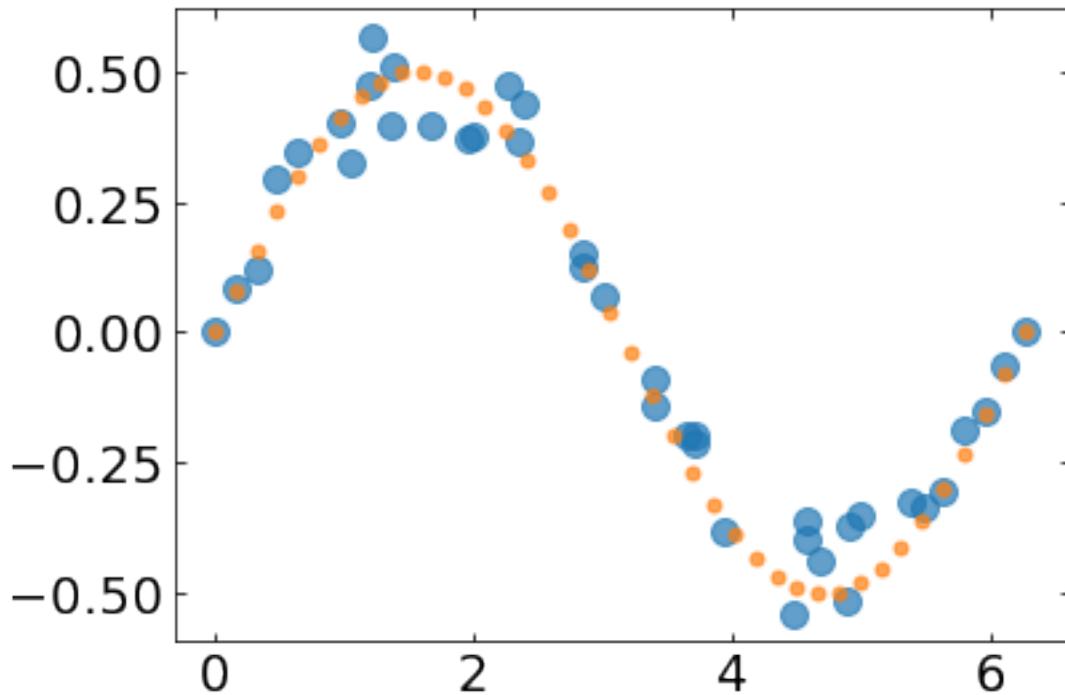
(provienie de la página anterior)

```
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)))
```

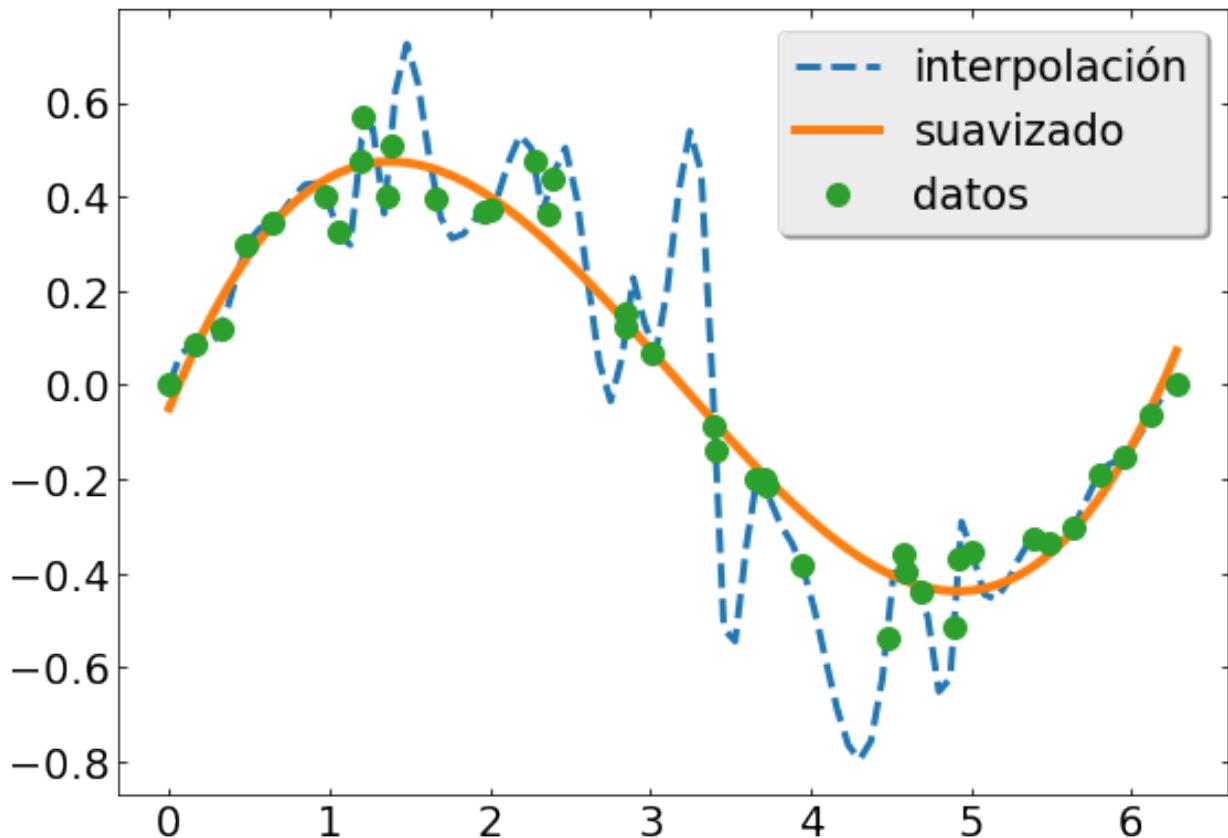
```
# Los puntos a interpolar tienen "ruido" en las dos direcciones x-y
plt.plot(x3,y3,'o', x4,y4, '.', alpha=0.7 );
```



```
# Grilla donde evaluar la función interpolada
x1 = np.linspace(0, 2*np.pi, 90)
```

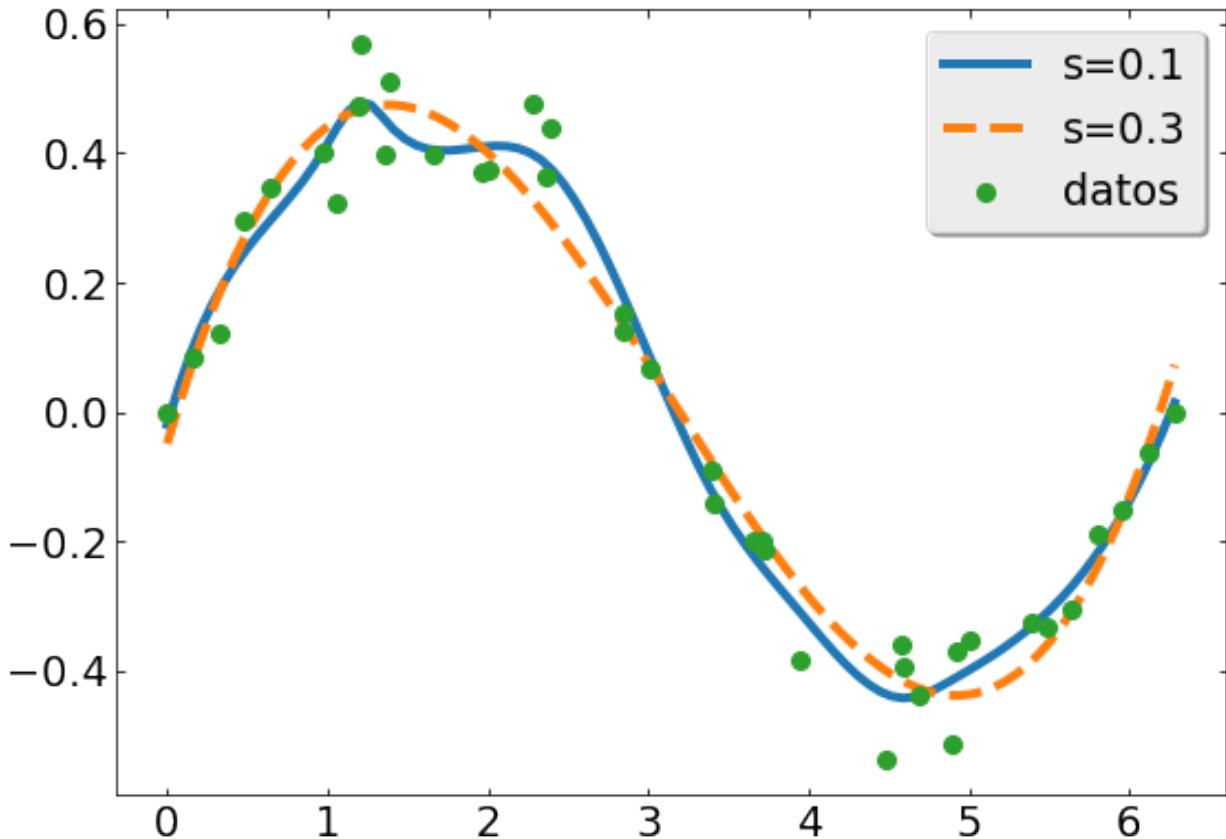
```
tck0 = interpolate.splrep(x3,y3, s=0) # Interpolación con B-Splines
y_s0 = interpolate splev(x1,tck0) # Evaluamos en la nueva grilla
tck3 = interpolate.splrep(x3,y3,s=0.3) # Aproximación suavizada
y_s3 = interpolate splev(x1,tck3) # Evaluamos en la nueva grilla
```

```
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');
```

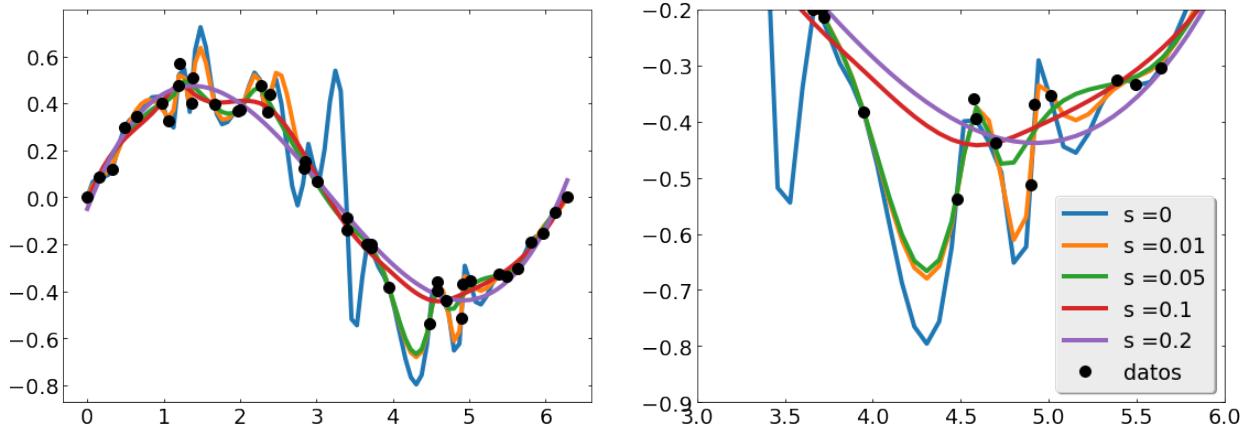


El valor del parámetro s determina cuánto se suaviza la curva. El valor por default $s=0$ obliga al algoritmo a obtener una solución que no difiere en los valores de la tabla, un valor de s mayor que cero da cierta libertad para obtener la aproximación que se acerque a todos los valores manteniendo una 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='s=0.1');
plt.plot(x1,y_s3, "--", label='s=0.3');
plt.plot(x3,y3,'o', markersize=8, label='datos' )
plt.legend(loc='best');
```



```
fig, (ax0, ax1) = plt.subplots(figsize=(2.1*fsize[0], fsize[1]), ncols=2)
for s in [0, 0.01, 0.05, 0.1, 0.2]:
    tck1 = interpolate.splrep(x3,y3, s=s) # Interpolación con suavizado
    ys = interpolate splev(x1,tck1)
    ax0.plot(x1,ys, "--", label=f'{s =j}');
    ax1.plot(x1,ys, "--", label=f'{s =j}');
    ax0.plot(x3,y3, 'ok', label='datos')
    ax1.plot(x3,y3, 'ok', label='datos')
plt.xlim(3,6)
ax1.set_ylim(-0.9, -0.2)
ax1.legend(loc='best');
```



14.1.5 Cantidades derivadas de *splines*

De la interpolación (suavizada) podemos calcular, por ejemplo, la derivada.

```
yderiv = interpolate.splev(x1,tck3,der=1) # Derivada
```

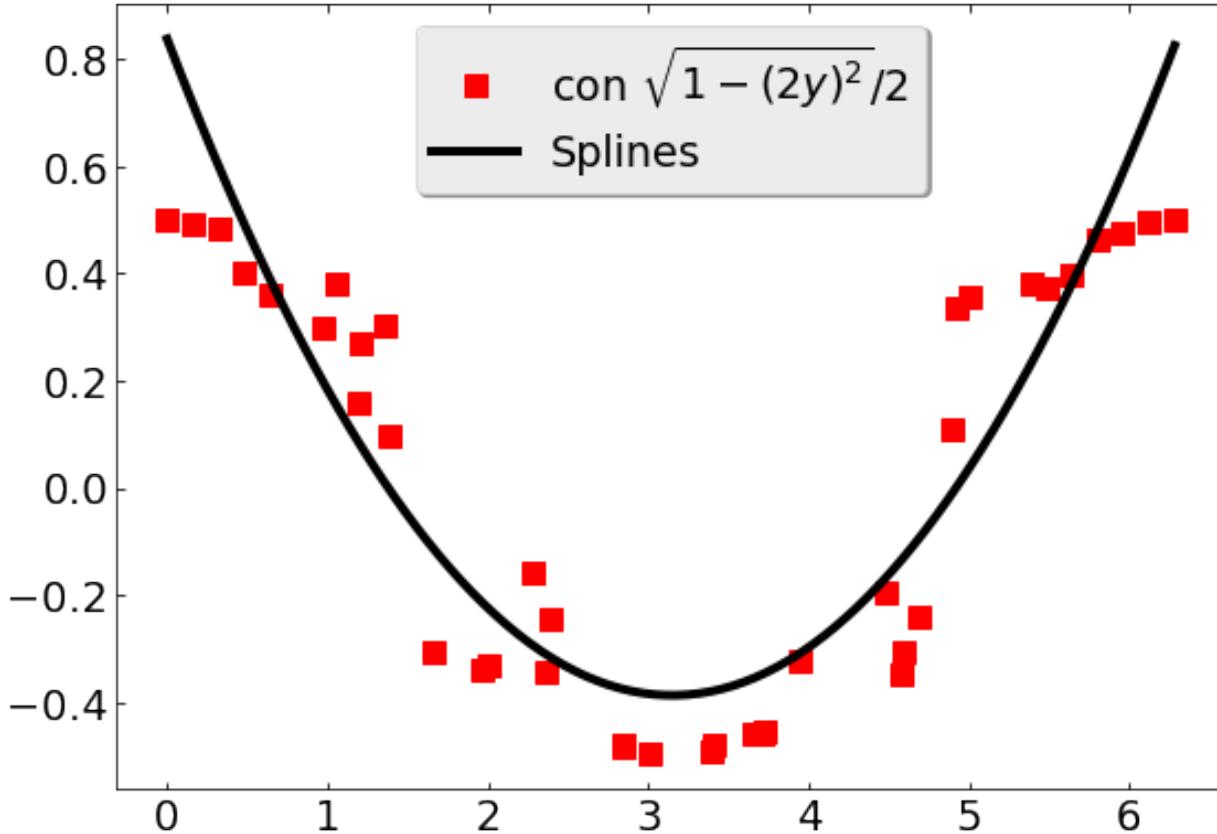
Si tenemos sólo los datos podríamos tratar de calcular la derivada en forma numérica como el coseno

$$y' = 0,5\sqrt{1 - (2y)^2}$$

Comparemos los dos resultados:

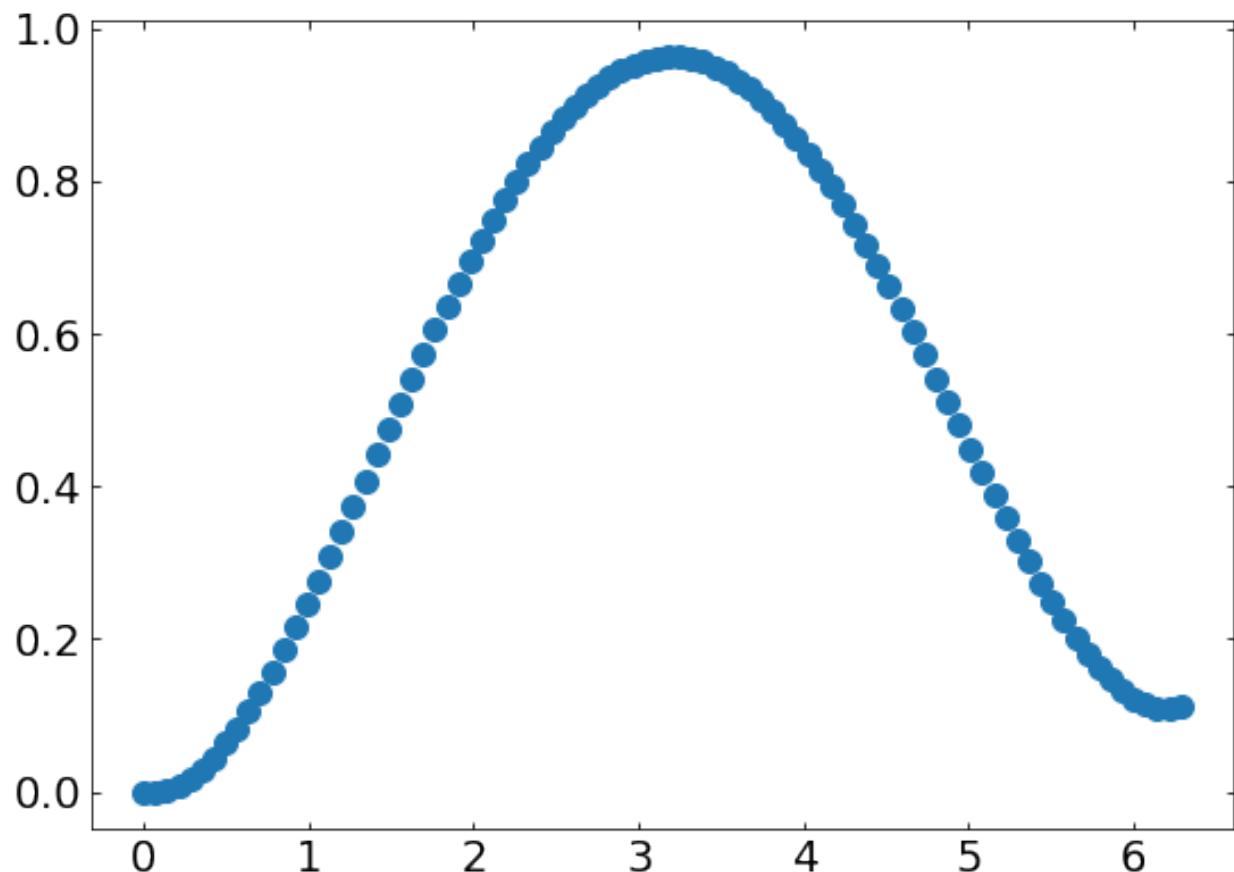
```
cond = (x3 > np.pi/2) & (x3 < 3*np.pi/2)
yprimal = np.where(cond, -1, 1) * 0.5*np.sqrt(np.abs(1 - (2*y3)**2))
```

```
plt.figure(figsize=fsize)
plt.plot(x3, yprimal,"sr", label=r"con $\sqrt{1-(2y)^2}/2$")
plt.plot(x1,yderiv,'-k', label=u'Splines')
plt.legend(loc='best');
```



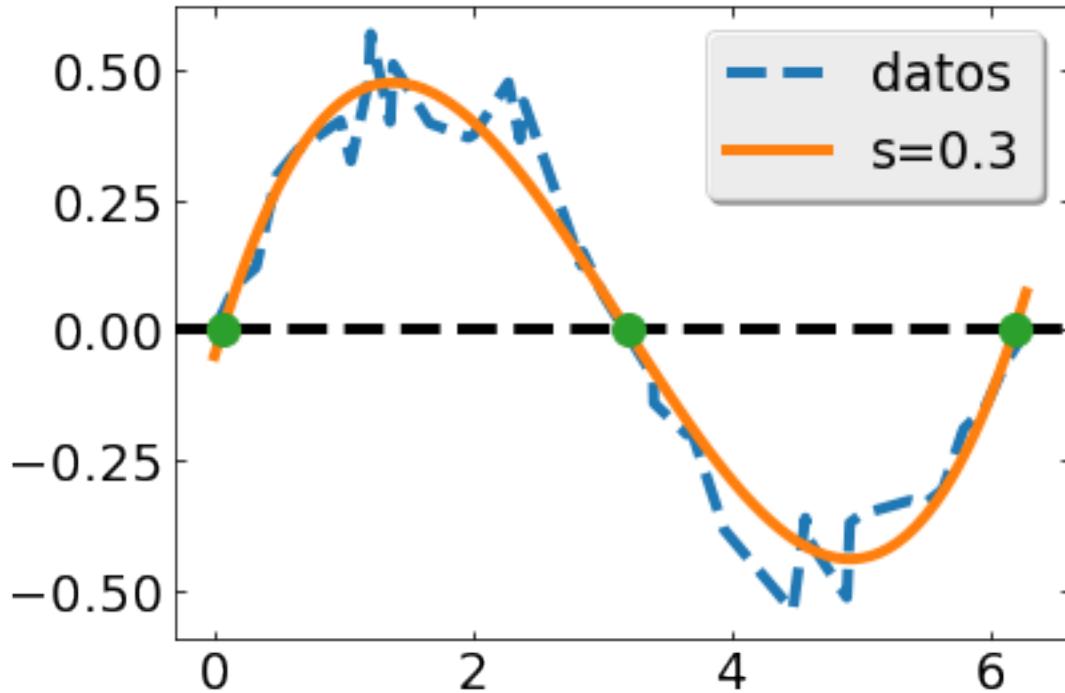
o podemos calcular la integral, o las raíces de la función

```
plt.figure(figsize=fsize)
yt= np.array([interpolate.splint(0,t,tck3) for t in x1])
plt.plot(x1,yt,'o');
```



```
raices = interpolate.sproot(tck3)
```

```
plt.axhline(0, color='k', ls='--');
plt.plot(x3,y3, "--", label=u'datos');
plt.plot(x1,y_s3, "--", label=u's=0.3');
plt.plot(raices, np.zeros(len(raices)), 'o', markersize=12)
plt.legend();
```



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)
```

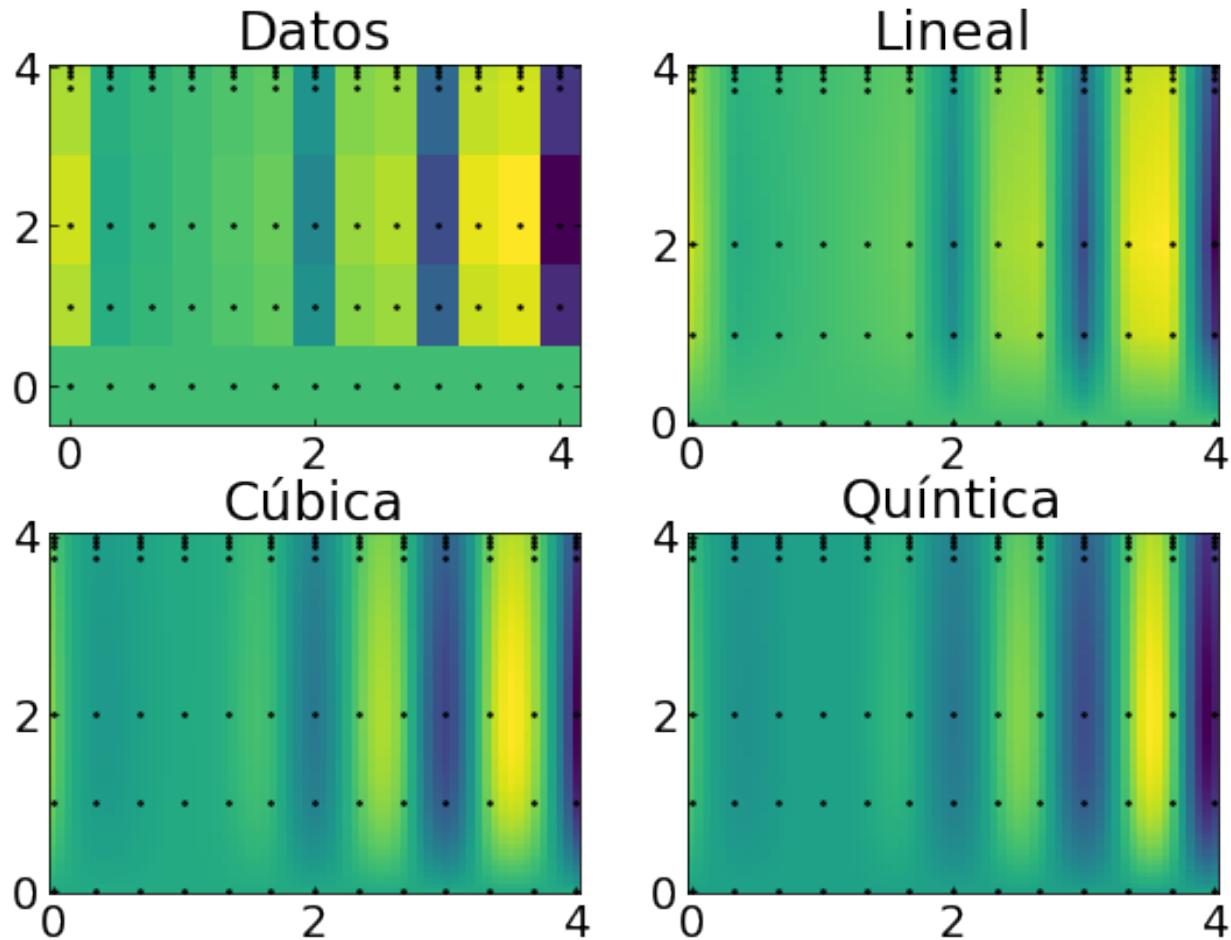
Ahora tenemos los valores de la función sobre la grilla determinada por todos los posibles pares (x, y) de la tabla original. En el caso de $f1$ utilizamos una interpolación lineal para cada punto, en $f3$ una interpolación cúbica, y en $f5$ una interpolación de orden 5. Grafiquemos los resultados

```
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, shading='auto')
ax[0,0].set_title('Datos')
ax[0,1].pcolormesh(X2, Y2, z1, shading='auto')
ax[0,1].set_title('Lineal')
ax[1,0].pcolormesh(X2, Y2, z3, shading='auto')
ax[1,0].set_title('Cúbica')
ax[1,1].pcolormesh(X2, Y2, z5, shading='auto')
ax[1,1].set_title('Quíntica');
fig.subplots_adjust(hspace=0.3)
```



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 `meshgrid()` para entender su uso

```
a = np.arange(3)
b = np.arange(3, 7)
print(' a=', a, '\n', 'b=', b)
```

```
a= [0 1 2]
b= [3 4 5 6]
```

```
A,B = np.meshgrid(a,b)
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)

(proviene 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 `griddata()`. 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)

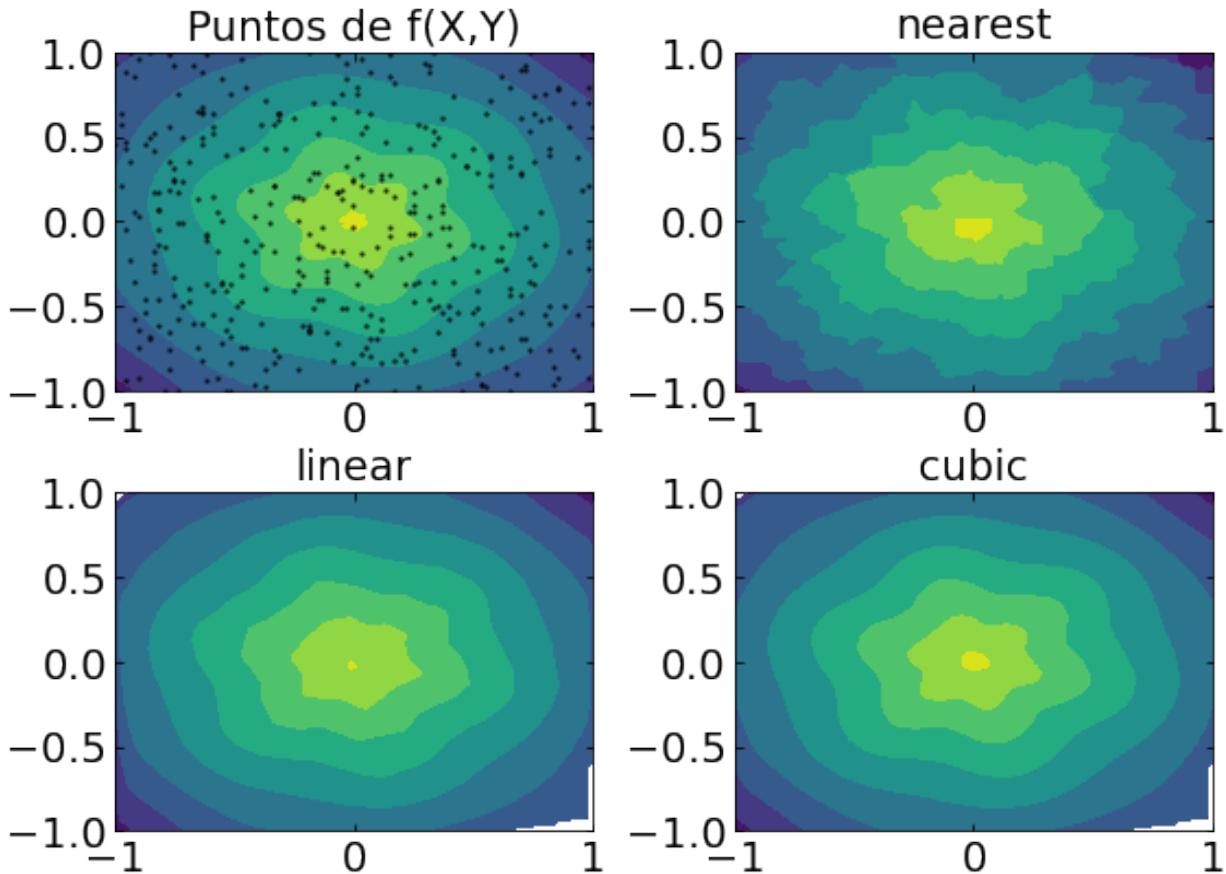
(provine de la página anterior)

```

ax[r,c].contourf(X, Y, Ti)
ax[r,c].set_title('{}\n'.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)

```

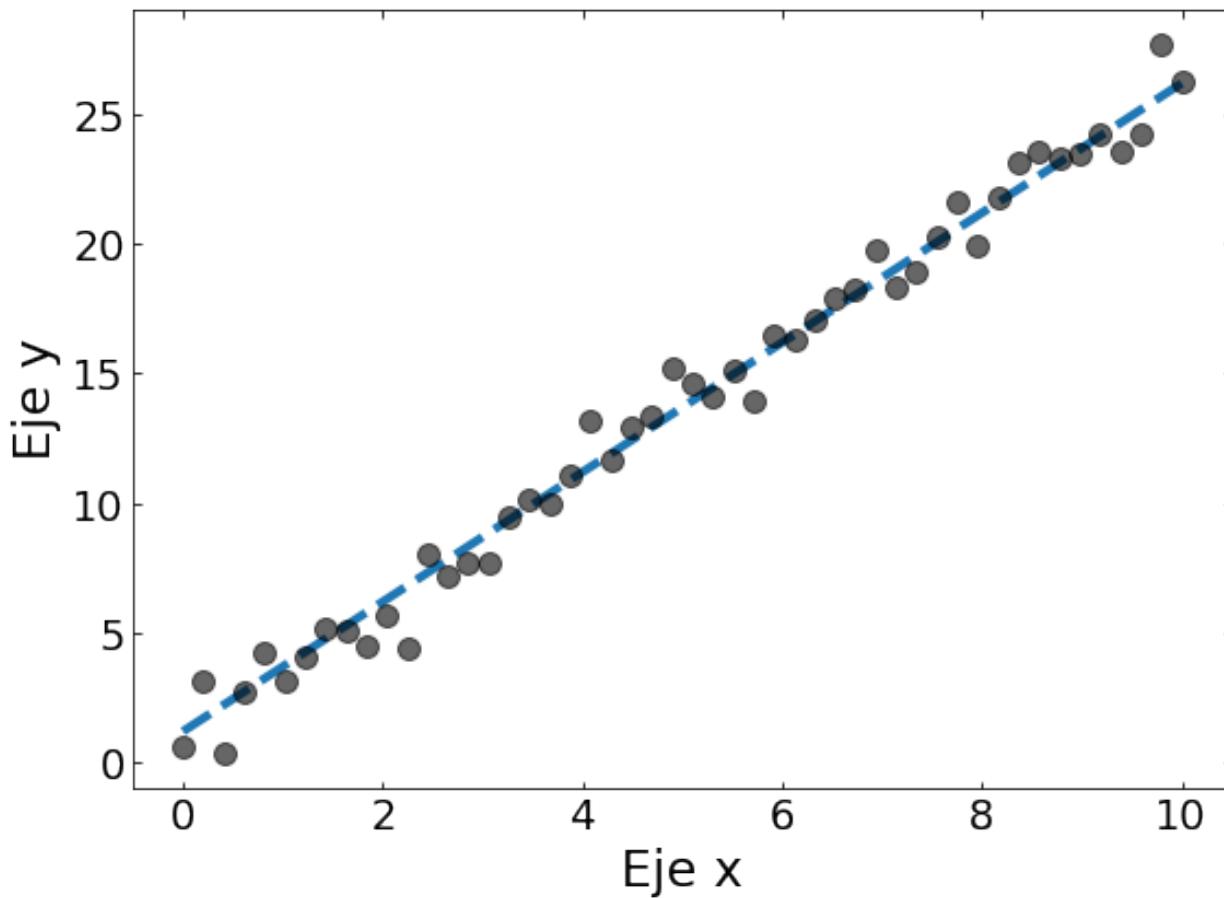
plt.figure(figsize=fsize)
x = np.linspace(0, 10, 50)
y0 = 2.5*x + 1.2
ruido = np.random.normal(loc= 0., scale= 1, size= y0.size)
y = y0 + ruido
plt.plot(x,y0, '--')

```

(continué en la próxima página)

(proviene de la página anterior)

```
plt.plot(x,y, 'ok', alpha=0.6)
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?
```

```
[2.55557379 0.91800682]
<class 'numpy.ndarray'>
```

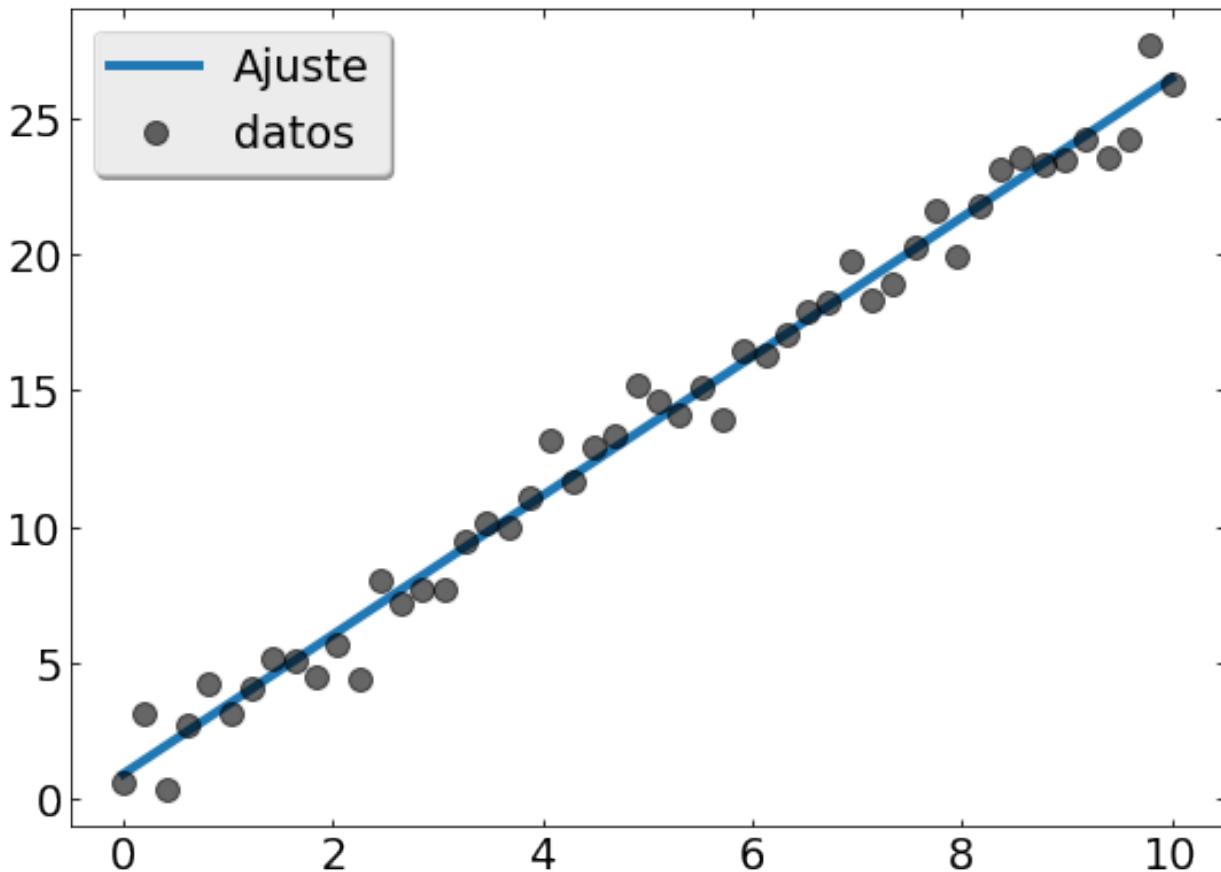
Nota: ¿Qué devuelve polyfit () ?

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]
```

```
np.polyfit?
```

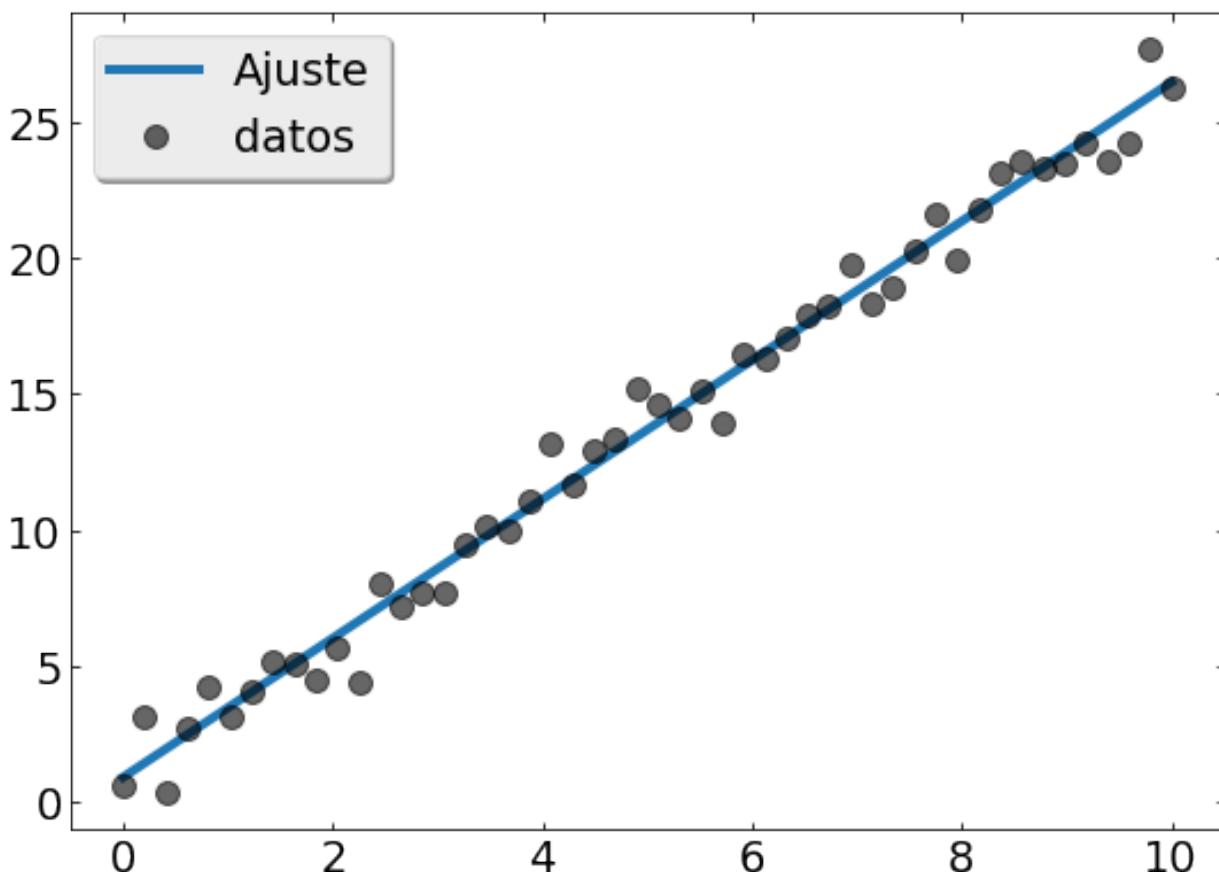
```
plt.figure(figsize=fsize)
plt.plot(x, p[0]*x + p[1], '--', label='Ajuste')
plt.plot(x,y,'ok', label='datos', alpha=0.6)
plt.legend(loc='best');
```



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)
```

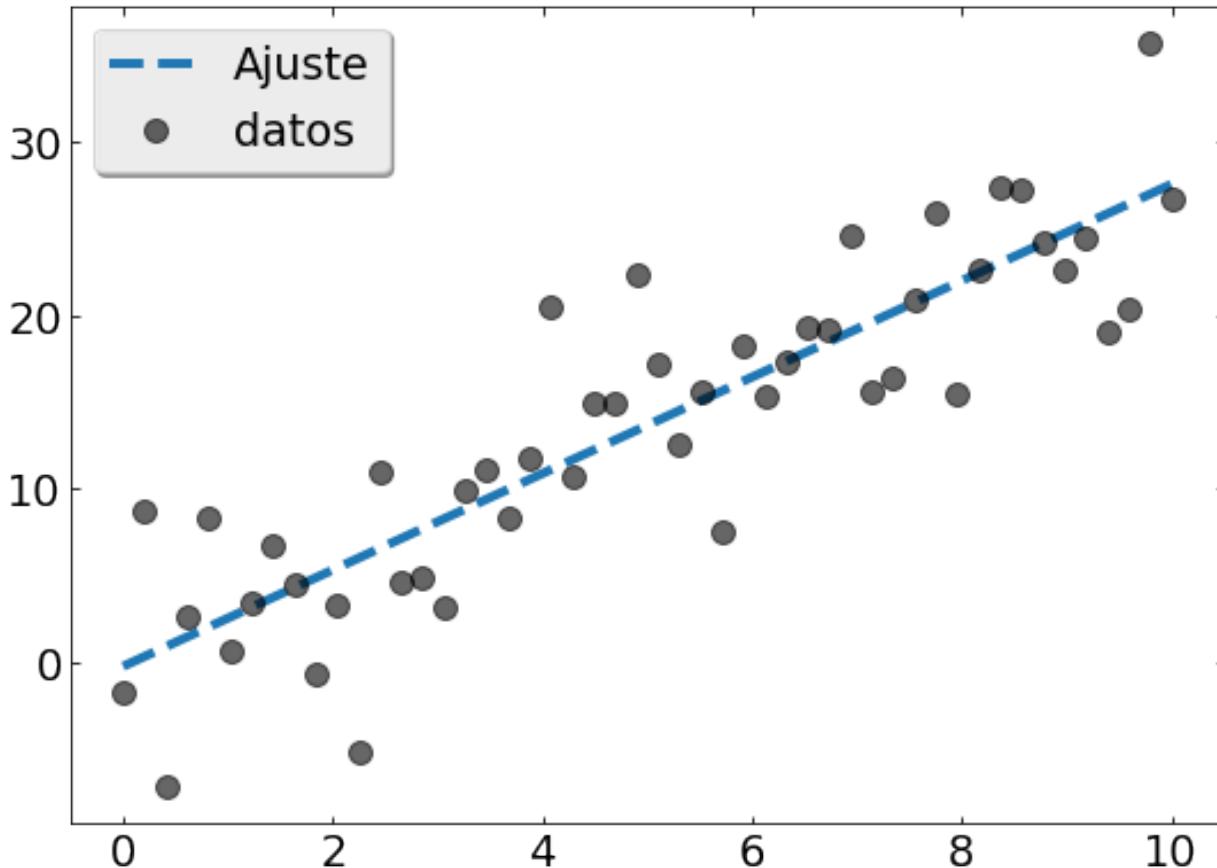
```
plt.figure(figsize=fsize)
plt.plot(x, np.polyval(p,x), '--', label='Ajuste')
plt.plot(x,y,'ok', label='datos', alpha=0.6)
plt.legend(loc='best');
```



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=fsize)
plt.plot(x,np.polyval(p,x), '--', label='Ajuste')
plt.plot(x,y, 'ok', alpha=0.6, label='datos')
plt.legend(loc='best');
```

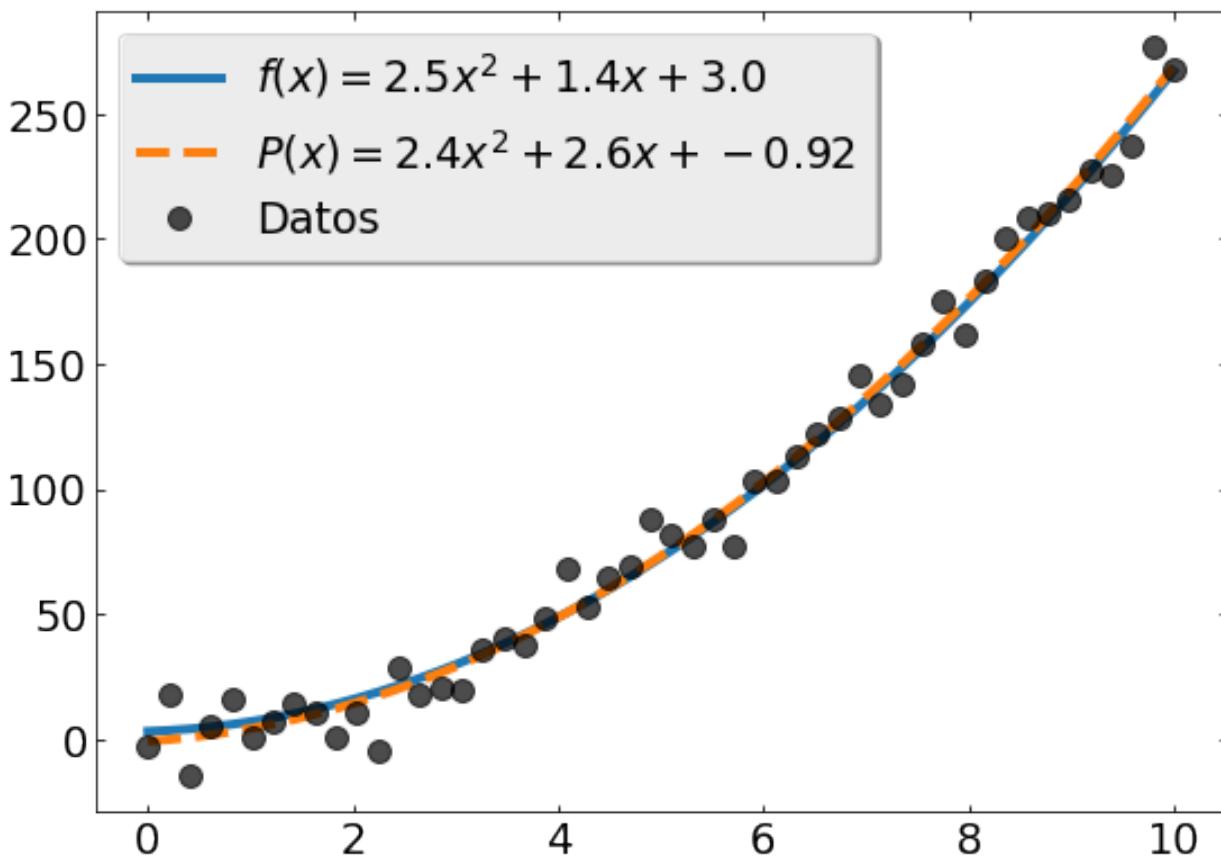


Similarmente podemos usar polinomios de orden superior. Por ejemplo, para utilizar paráolas sólo tenemos que cambiar el orden n del polinomio en el argumento de `polyfit(x, y, n)`:

```
# 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 Ejercicios 13 (a)

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.

14.6 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
```

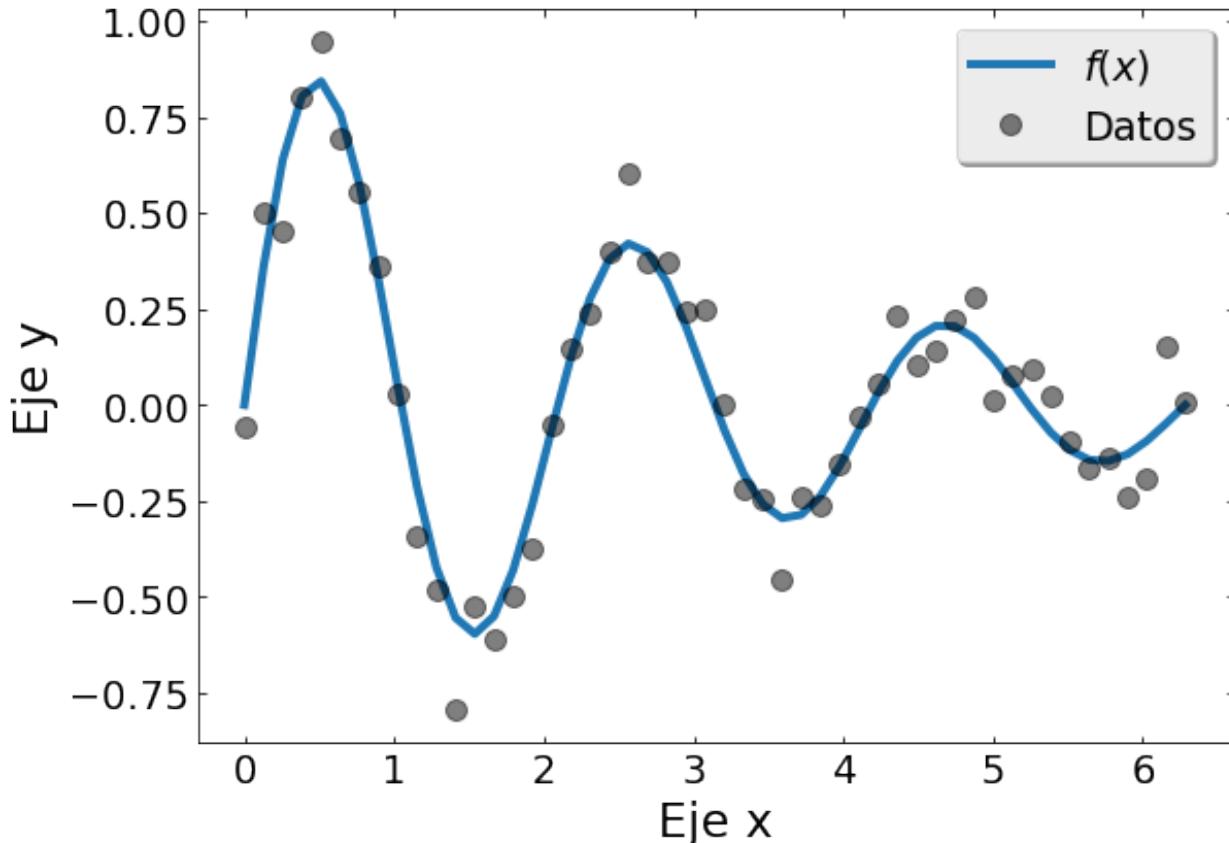
Generamos ahora “datos” con dispersión basados en la función

$$f(x) = a \sin(bx - d)e^{-cx}$$

```
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 $ydata = f(xdata, *params) + \text{eps}$.

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 : array_like or object

The independent variable where the data is measured.

Should usually be an M-length sequence or an (k, M) -shaped array for functions with k predictors, but can actually be any object.

ydata : array_like

The dependent data, a length M array – nominally $f(xdata, ...)$.

p0 : array_like, optional

Initial guess for the parameters (length N). 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 $M \times M$ array, optional

Determines the uncertainty in $ydata$. If we define residuals as $r = ydata - f(xdata, *popt)$, then the interpretation of σ depends on its number of dimensions:

- A 1-D σ should contain values of standard deviations of errors in $ydata$. In this case, the optimized function is $\text{chisq} = \sum((r / \sigma) ** 2)$.

- A 2-D σ should contain the covariance matrix of errors in $ydata$. In this case, the optimized function is $\text{chisq} = r.T @ \text{inv}(\sigma) @ r$.

.. versionadded:: 0.19

None (default) is equivalent of 1-D σ filled with ones.

absolute_sigma : bool, optional

If True, σ is used in an absolute sense and the estimated parameter

covariance pcov reflects these absolute values.

If False (default), only the relative magnitudes of the σ 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. Default is `False`.

Mathematically,

$$\text{pcov}(\text{absolute_sigma}=\text{False}) = \text{pcov}(\text{absolute_sigma}=\text{True}) * \text{chisq}(\text{popt}) / (M-N)$$

`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` : 2-D 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 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)
>>> rng = np.random.default_rng()
>>> y_noise = 0.2 * rng.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([2.56274217, 1.37268521, 0.47427475])
>>> 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 <= a <= 3,
0 <= b <= 1 and 0 <= c <= 0.5:

>>> popt, pcov = curve_fit(func, xdata, ydata, bounds=(0, [3., 1., 0.5]))
>>> popt
array([2.43736712, 1.           , 0.34463856])
>>> 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
```

(continué en la próxima página)

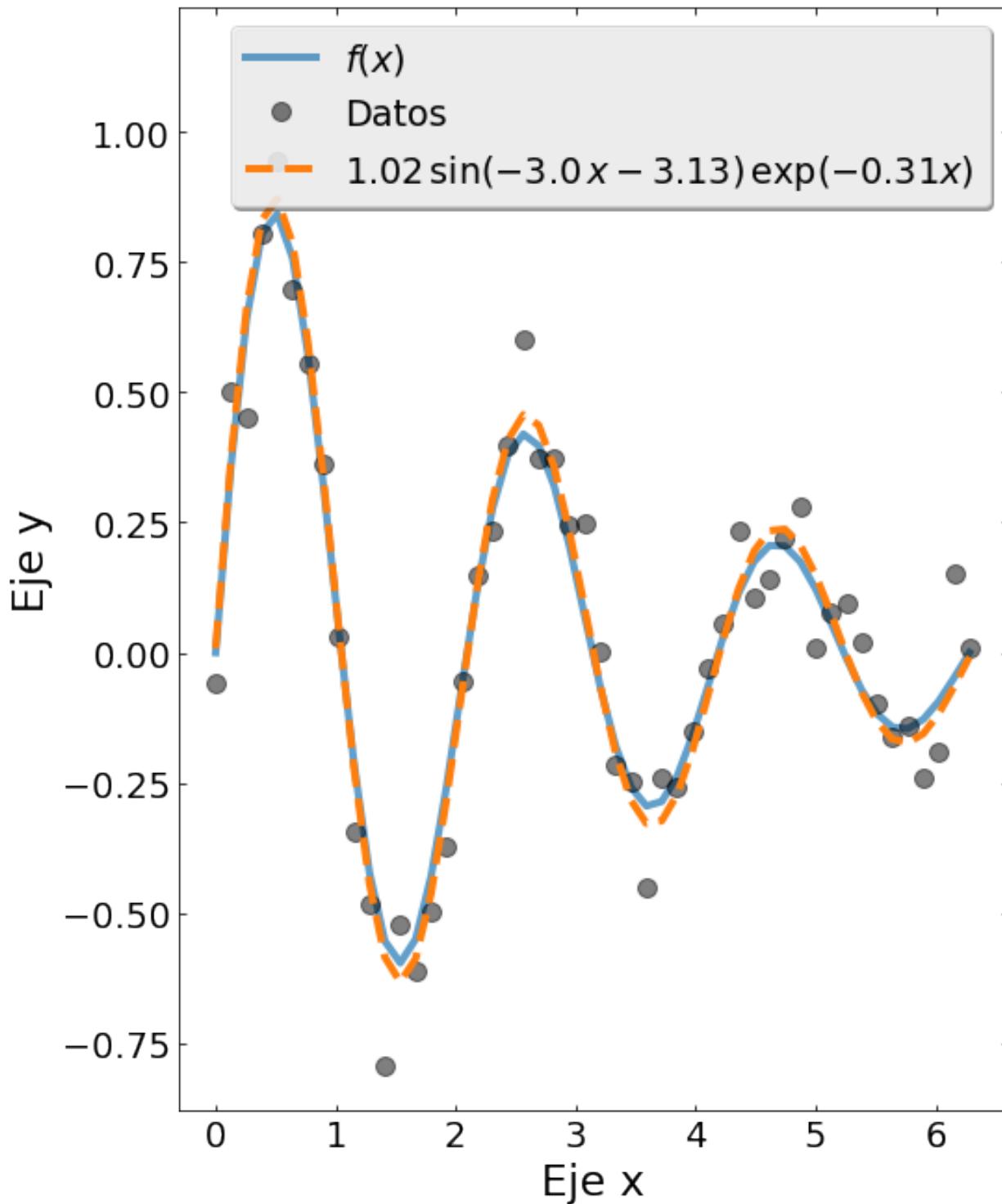
(proviene de la página anterior)

```
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)  
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]));
```

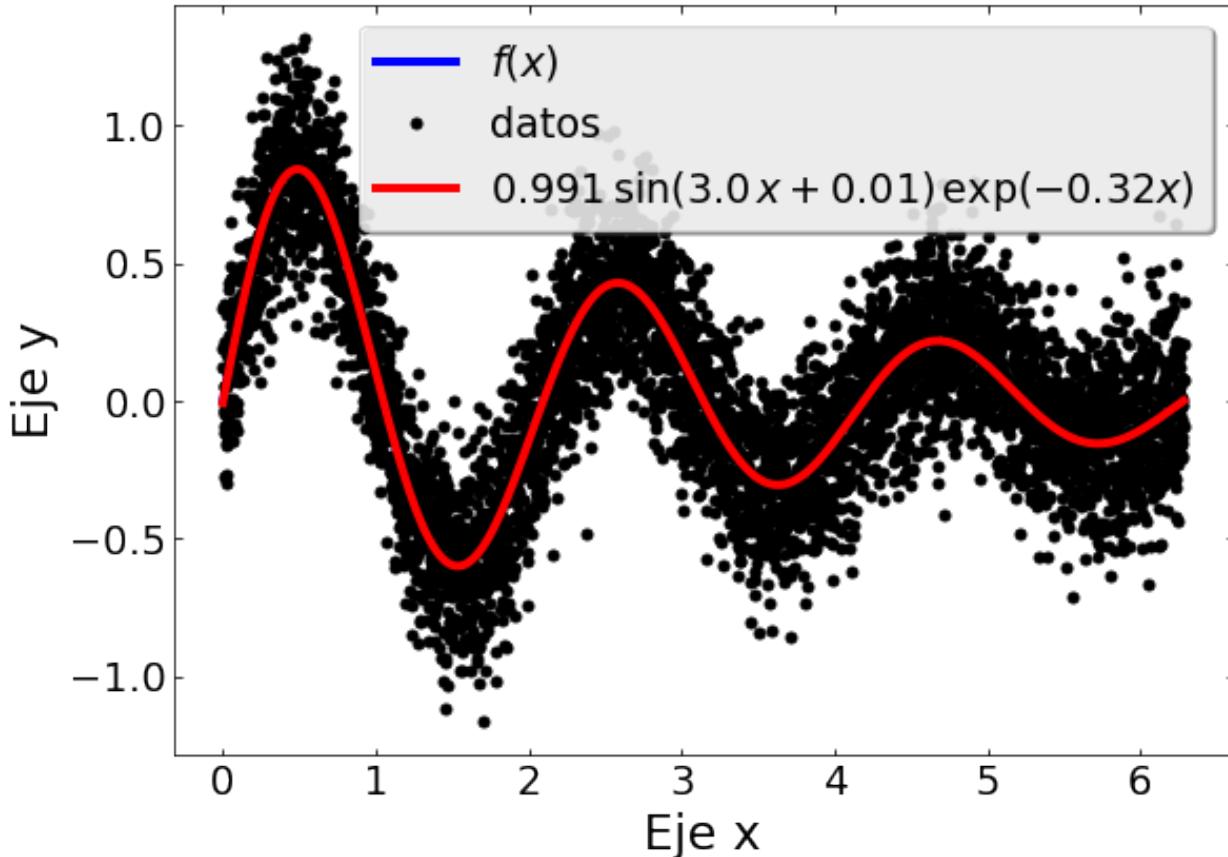


Veamos otro ejemplo similar, con muchos datos y dispersión

```
# 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([1.65511425e-04, 3.92802040e-05, 4.30581667e-05, 1.49599645e-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 = 0.99086 ± 0.0129
b = 3.0015 ± 0.00627
c = 0.32261 ± 0.00656
d = 0.0079556 ± 0.0122
```

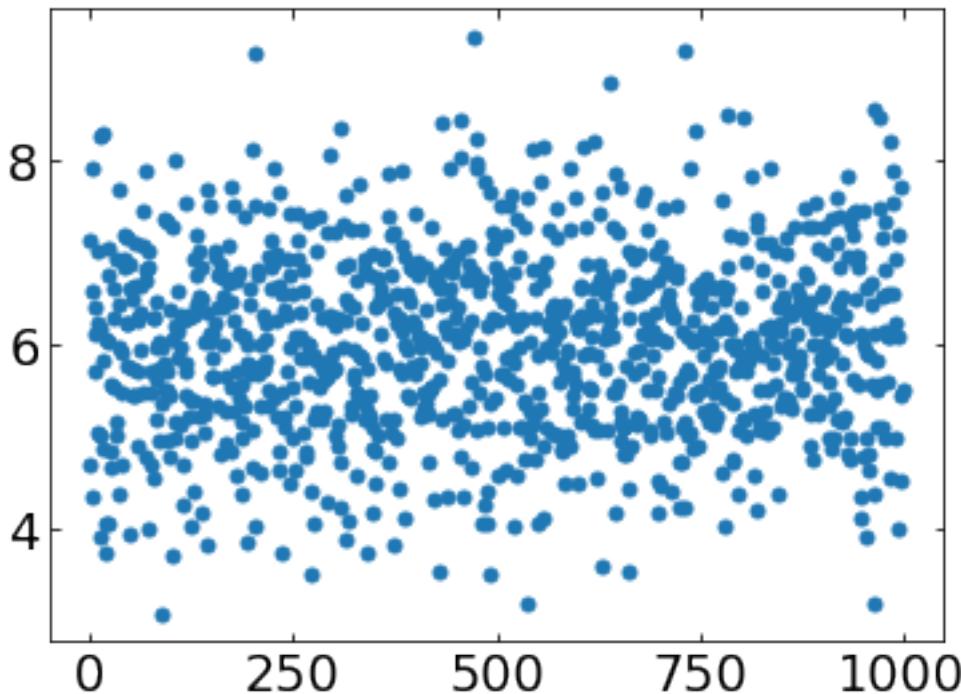
14.6.1 Ejemplo: Fiteo de picos

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, '.')
```

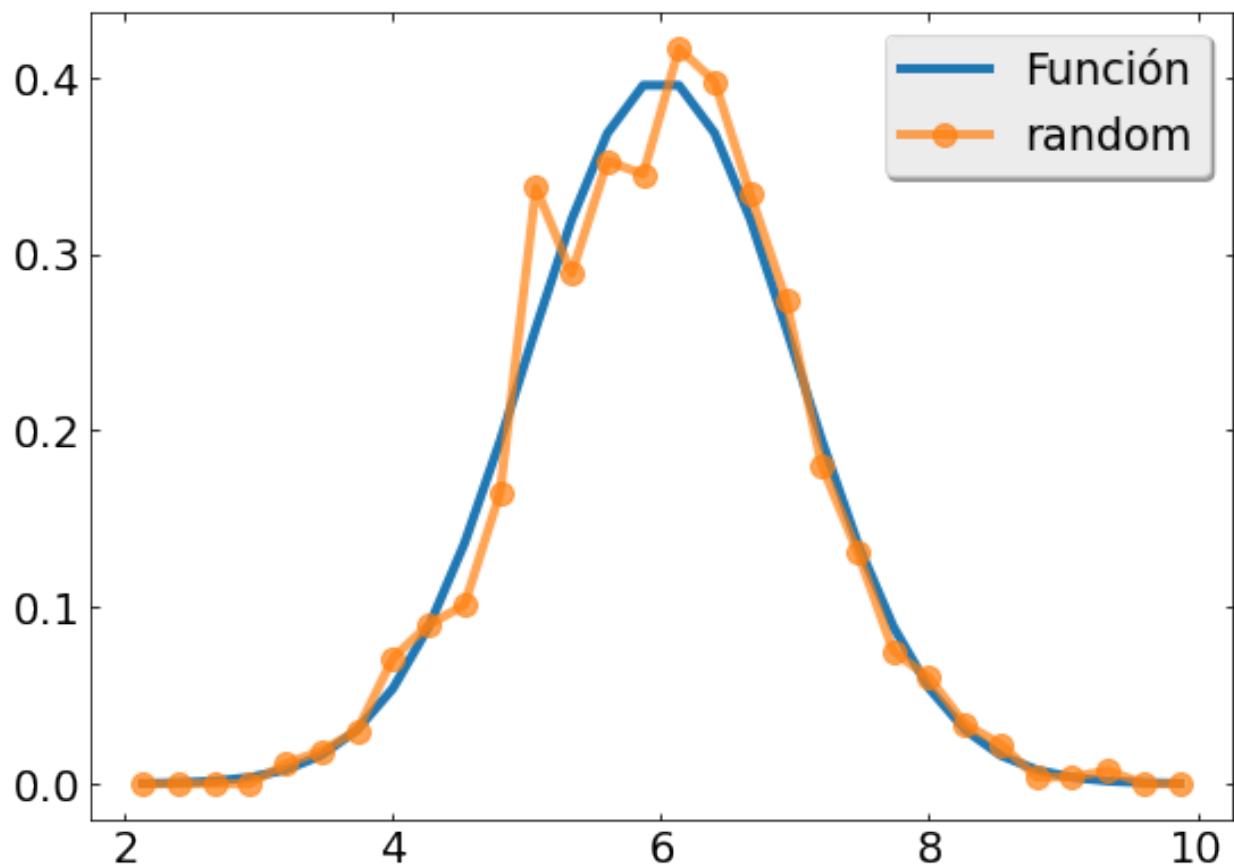
```
[<matplotlib.lines.Line2D at 0x7f20cced3640>]
```



```
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=fsize)
plt.plot(x,b, '-.', label=u'Función')
plt.plot(x,y, 'o-', alpha=0.7, label='random')
plt.legend(loc='best');
```

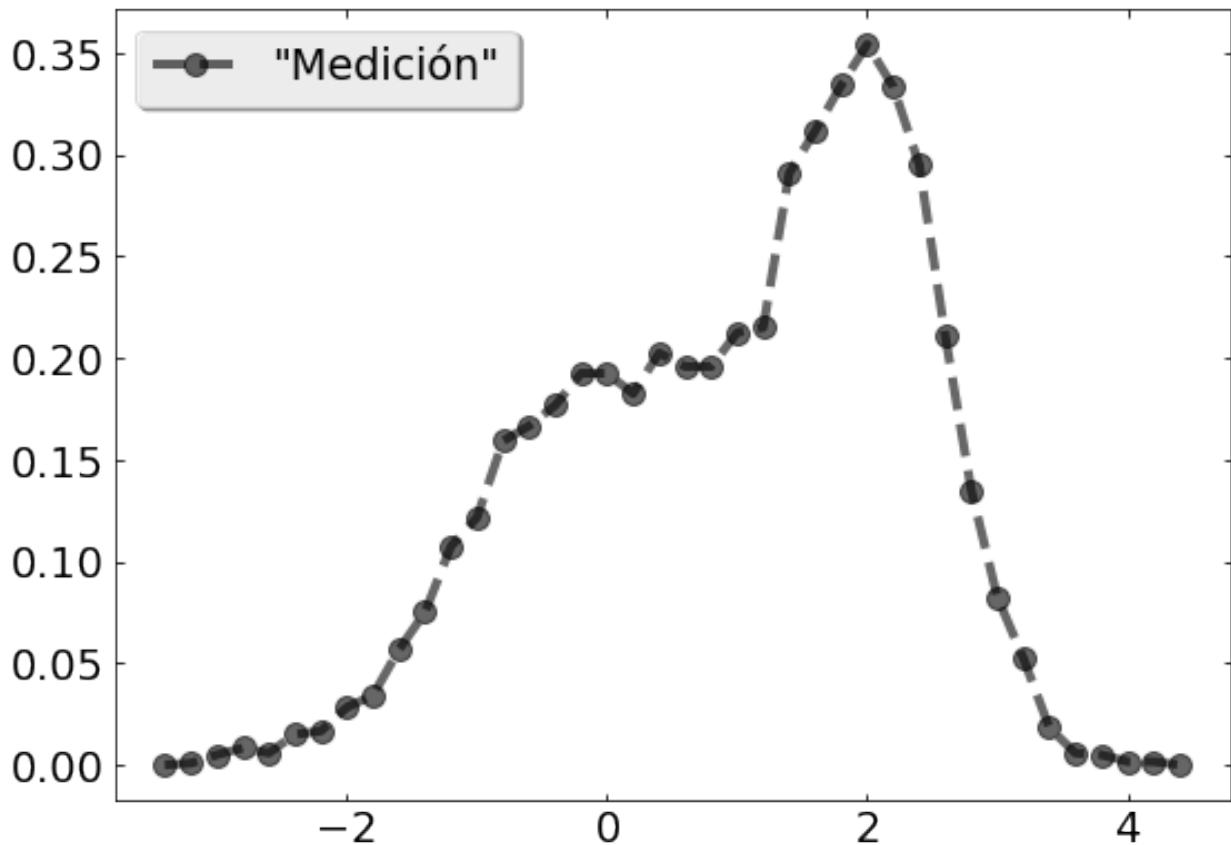


Usando esta idea generemos “datos” correspondiente a dos picos, que son diferentes y están separados pero que no podemos distinguirlos completamente porque se superponen.

Generemos los datos:

```
npoints = 3000
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"
    G = stats.norm.pdf
    return coeffs[0]*G(x, loc=coeffs[1], scale=coeffs[2]) + \
        coeffs[3]*G(x, loc=coeffs[4], scale=coeffs[5])
```

```
help(modelo)
```

```
Help on function modelo in module __main__:

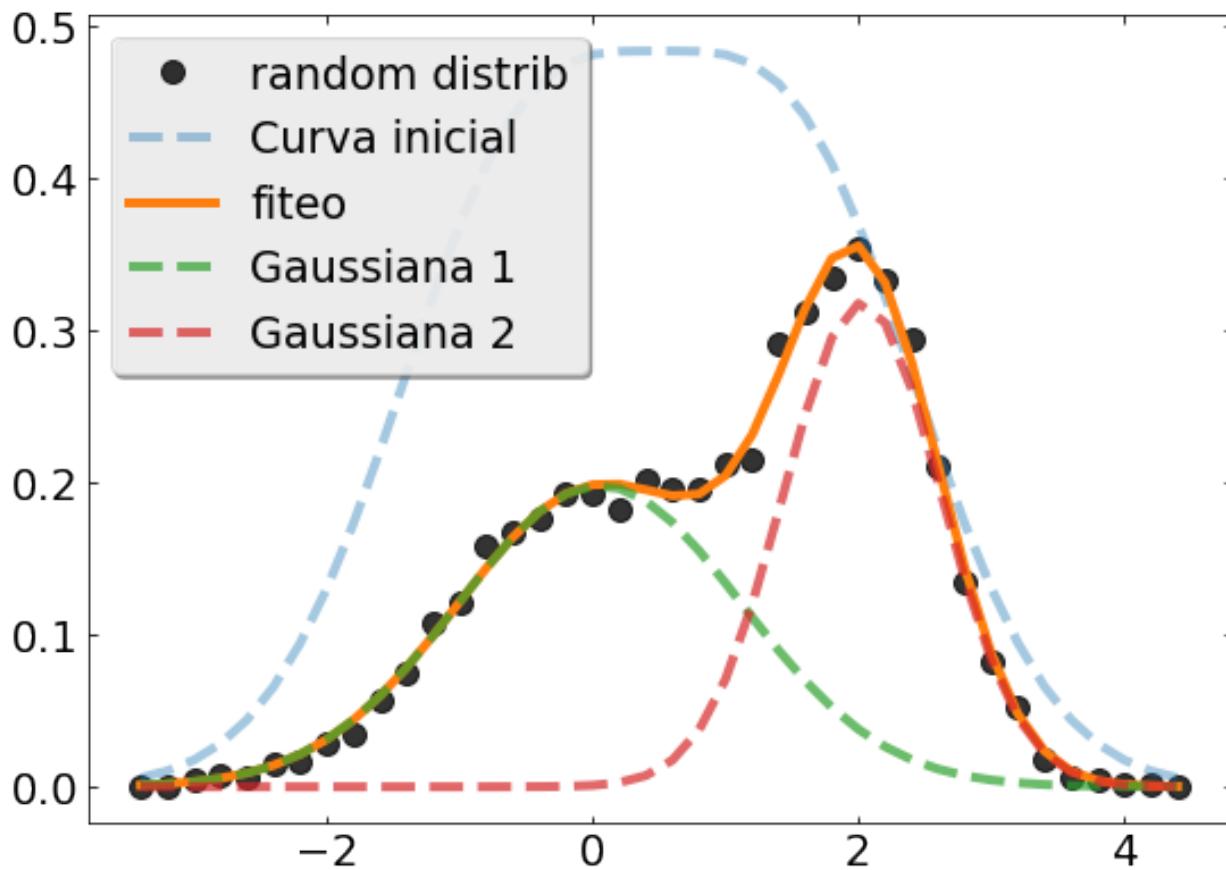
modelo(x, *coeffs)
    Suma de dos Gaussianas, con pesos dados por coeffs[0] y coeffs[3], respectivamente
```

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)
```

```
[0.53045911 0.05504444 1.06839904 0.47316387 2.02859541 0.59299279]
```

```
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)

(proviene 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 + 0.2 * 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]], shading='auto')
    ax[k[0], k[1]].axis(limits)
    fig.colorbar(im, ax=ax[k[0], k[1]])

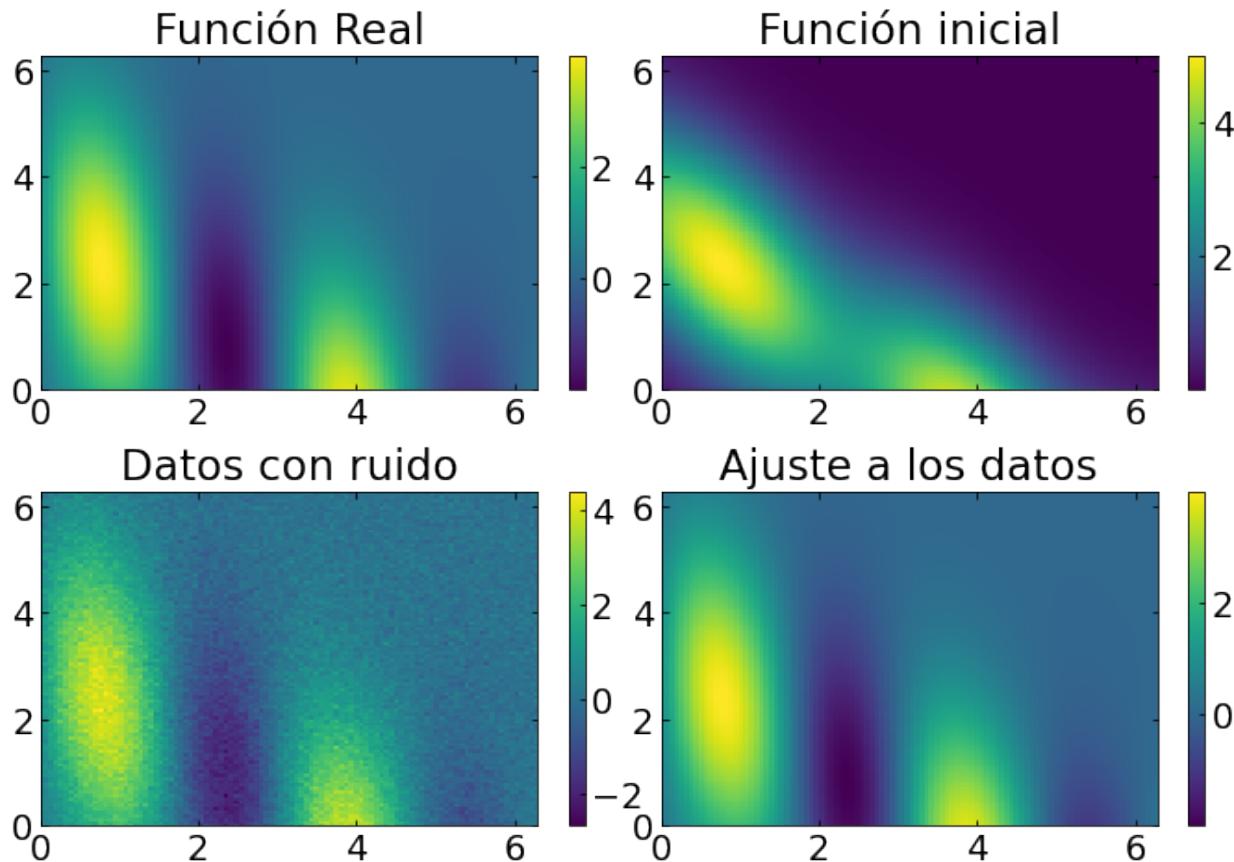
plt.subplots_adjust(hspace=0.3, wspace=0.)
```

Valores de los parámetros

(continué en la próxima página)

(proviene de la página anterior)

```
Real    : (3, 1, 0.5)
Inicial: (1, 4, 1)
Ajuste  : [2.98879056 0.99982914 0.5006972 ]
```



```
ydata.shape, Z.shape
```

```
((10000,), (100, 100))
```

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

14.7 Ejercicios 13 (b)

2. Queremos hacer un programa que permita fitear 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: Animaciones e interactividad

15.1 Animaciones con Matplotlib

Matplotlib tiene funciones para hacer animaciones de una manera conveniente. Hay excelente información sobre el tema en:

- La documentación (con ejemplos)
- Tutorial en Pythonic Perambulations

Vamos a ver brevemente cómo hacer animaciones, en pocos Pasos

15.1.1 Una animación simple en pocos pasos

```
cd scripts/animaciones
```

```
/home/fiol/Clases/IntPython/clases-python/clases/scripts/animaciones
```

```
%matplotlib tk  
%run ejemplo_animation_1.py
```

```
import numpy as np  
import matplotlib.pyplot as plt  
import matplotlib.animation as animation  
plt.ioff()  
# Creamos los datos  
xmax = 2*np.pi  
Npts= 50  
  
x = np.linspace(0, xmax, Npts)  
data = np.vstack([x, np.sin(x)])
```

(continué en la próxima página)

(proviene de la página anterior)

```
def update_line(num, data, line):
    line.set_data(data[:, :num])
    return line,
# Creamos la figura e inicializamos
# Graficamos una línea sin ningún punto
# Fijamos las condiciones de graficación
fig1, ax = plt.subplots(figsize=(12,8))
L, = plt.plot([], [], '-o') # equivalente a la siguiente
# L = plt.plot([], [], '-o')[0]
ax.set_xlim(0, xmax)
ax.set_ylim(-1.1, 1.1)
ax.set_xlabel('x')
ax.set_title('Animación de una oscilación')

#
line_ani = animation.FuncAnimation(fig1, update_line, Npts, fargs=(data, L),
                                   interval=100, blit=True)

plt.show()
```

Este código da como resultado una función oscilante que se va creando. Este es un ejemplo simple que puede ser útil para graficar datos de una medición o de un cálculo más o menos largo.

Preparación general

Como vemos, después de importar el submódulo `animation` (además de lo usual):

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
plt.ioff()
```

nos aseguramos que estamos trabajando en modo **no** interactivo (con `plt.ioff()`).

Creamos los datos para graficar

Creación de datos para graficar

Creamos los datos para mostrar en la animación.

```
xmax = 2*np.pi
Npts = 50
x = np.linspace(0, xmax, Npts)
data = np.vstack([x, np.sin(x)])
```

Acá `data` es un array 2D, con los datos x, y .

Preparación de la figura

A continuación preparamos la zona de graficación:

1. Creamos la figura y eje
2. Creamos las líneas de graficación (una en este caso)
3. Fijamos los límites de graficación
4. Agregamos el texto, que va a ser invariante durante la animación

```
fig1, ax = plt.subplots(figsize=(12,8))
L, = plt.plot([0], [0], '-o', lw=3)
ax.set_xlim(0, xmax)
ax.set_ylim(-1.1, 1.1)
ax.set_xlabel('x')
ax.set_title('Animación de una oscilación')
```

Como sabemos, el llamado a `plot()` devuelve una lista de líneas (de un solo elemento). A este elemento lo llamamos `L`, y ya le damos las características que queremos que tenga. En este caso, fijamos el símbolo (círculos), con líneas de ancho 3. Vamos a modificar esta línea `L` en cada cuadro de la animación.

Función para actualizar la línea

Debemos crear una función que modifique las curvas en cada cuadro.

```
def update_line(num, data, line):
    line.set_data(data[:, :num])
    return line,
```

Esta función debe recibir como argumento el número de cuadro, que acá llamamos `num`. Además, en este caso recibe los datos a graficar, y la línea a modificar.

Esta función devuelve una línea `L`, que es la parte del gráfico que queremos que se actualice en cada frame.

Notemos acá que no es necesario que tome como argumento los datos guardados en `data` y la línea `line`, ya que son variables globales a las que hay acceso dentro del *script*. De la misma manera no es necesario que devuelva la línea, por la misma razón.

Animación de la figura

Finalmente llamamos a la función que hace la animación: `animation.FuncAnimation()`

```
import matplotlib.animation as animation
np.info(animation.FuncAnimation)
```

```
FuncAnimation(fig, func, frames=None, init_func=None, fargs=None,
              save_count=None, , cache_frame_data=True, **kwargs)
```

*Makes an animation by repeatedly calling a function *func.*

.. note::

You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

Parameters

fig : `~matplotlib.figure.Figure`
The figure object used to get needed events, such as draw or resize.

func : callable

The function to call at each frame. The first argument will be the next value in `frames`. Any additional positional arguments can be supplied via the `fargs` parameter.

The required signature is::

```
def func(frame, fargs) -> iterable_of_artists
```

If `blit == True`, `*func` must return an iterable of all artists that were modified or created. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

frames : iterable, int, generator function, or None, optional
Source of data to pass `func` and each frame of the animation

- If an iterable, then simply use the values provided. If the iterable has a length, it will override the `save_count` kwarg.
- If an integer, then equivalent to passing `range(frames)`
- If a generator function, then must have the signature::

```
def gen_function() -> obj
```

- If `None`, then equivalent to passing `itertools.count`.

In all of these cases, the values in `frames` is simply passed through to the user-supplied `func` and thus can be of any type.

init_func : callable, optional

A function used to draw a clear frame. If not given, the results of drawing from the first item in the `frames` sequence will be used. This function will be called once before the first frame.

The required signature is::

```
def init_func() -> iterable_of_artists
```

If `blit == True`, `init_func` must return an iterable of artists to be re-drawn. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

fargs : tuple or None, optional

Additional arguments to pass to each call to `func`.

```

save_count : int, default: 100
    Fallback for the number of values from frames to cache. This is
    only used if the number of frames cannot be inferred from frames,
    i.e. when it's an iterator without length or a generator.

interval : int, default: 200
    Delay between frames in milliseconds.

repeat_delay : int, default: 0
    The delay in milliseconds between consecutive animation runs, if
    repeat is True.

repeat : bool, default: True
    Whether the animation repeats when the sequence of frames is completed.

blit : bool, default: False
    Whether blitting is used to optimize drawing. Note: when using
    blitting, any animated artists will be drawn according to their zorder;
    however, they will be drawn on top of any previous artists, regardless
    of their zorder.

cache_frame_data : bool, default: True
    Whether frame data is cached. Disabling cache might be helpful when
    frames contain large objects.

```

Methods:

```

new_frame_seq -- Return a new sequence of frame information.
new_saved_frame_seq -- Return a new sequence of saved/cached frame_
information.
pause -- Pause the animation.
resume -- Resume the animation.
save -- Save the animation as a movie file by drawing every frame.
to_html5_video -- Convert the animation to an HTML5 <video> tag.
to_jshtml -- Generate HTML representation of the animation.

```

```
line_anim = animation.FuncAnimation(fig1, update_line, Npts,
                                    fargs=(data, L), interval=100, blit=True)
```

La función FuncAnimation() toma como argumentos:

- la figura (fig1) donde se realiza el gráfico.
- Una función a la que llama antes de dibujar cada *frame* (update_line),
- El argumento interval da el tiempo entre cuadros, en milisegundos.
- El argumento fargs es una tuple con los argumentos que necesita la función update_line(). En este caso (data, L).
- El argumento blit=True hace que sólo se actualicen las partes que cambian en la animación, mientras que las partes estáticas no se dibujan en cada cuadro.

Es importante que el objeto creado por FuncAnimation() no se destruya. Esto lo podemos asegurar asignando el objeto resultante a una variable, en este caso line_anim.

Opcional: grabar la animación a un archivo

Podemos grabar la animación a un archivo usando el método `save()` o el método `to_html5_video()` del objeto (`anim`) que devuelve la función `FuncAnimation()`.

Para poder grabar a archivo las animaciones se necesita tener instalados programas externos (alguno de *ffmpeg*, *avconv*, *imagemagick*). Ver https://matplotlib.org/api/animation_api.html para más información.

15.1.2 Segundo ejemplo simple: Quiver

Para hacer una animación de un campo de fuerzas o velocidades necesitamos datos en tres dimensiones. El siguiente ejemplo sigue los pasos de la animación anterior, excepto en la creación de datos y la graficación, que en lugar de usar `plot()` usa `quiver()`:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
plt.style.use('ggplot')

plt.ioff()

# ##### Creación de datos #####
# Creación de datos
x = np.linspace(-3, 3, 91)
t = np.linspace(0, 25, 30)
y = np.linspace(-3, 3, 91)
X3, Y3, T3 = np.meshgrid(x, y, t)
sinT3 = np.sin(2*np.pi*T3 /
               T3.max(axis=2)[..., np.newaxis])

G = (X3**2 + Y3**2)*sinT3
# Graficar una flecha cada step puntos
step = 10
x_q, y_q = x[::step], y[::step]

# Create U and V vectors to plot
U = G[::step, ::step, :-1].copy()
V = np.roll(U, shift=3, axis=2)

# ##### Figura y ejes.
fig1, ax = plt.subplots(figsize=(12,8))

ax.set_aspect('equal')

ax.set(xlim=(-4, 4), ylim=(-4, 4))

qax = ax.quiver(x_q, y_q, U[..., 0], V[..., 0],
                 scale=100)

def animate(i):
    qax.set_UVC(U[..., i], V[..., i])

anim = animation.FuncAnimation(fig1, animate, interval=100, frames=len(t)-1,
                                repeat=True)
```

(continúe en la próxima página)

(provine de la página anterior)

```
# anim.save('quiver.gif', writer='imagemagick')
anim.save('quiver.mp4')
plt.show()
```

```
%run ejemplo_quiver.py
```

Comentarios:

- Se utilizó la función `quiver()` para generar un campo vectorial. La forma de esta función es:

```
quiver([X, Y], U, V, [C], **kw)
```

X, Y define the arrow locations, *U, V* define the arrow directions, and *C* optionally sets the color.

- Se utilizaron *Ellipsis*, por ejemplo en casos como:

```
U[..., 0]
```

Las elipsis (tres puntos o la palabra *Ellipsis*) indican todo el rango para todas las dimensiones que no se dan explícitamente. En este ejemplo el *array U* tiene tres dimensiones, por lo que tendremos:

```
U[..., 0] = U[:, :, 0]
```

En general, las elipses reemplazan a los dos puntos en todas las dimensiones no dadas explícitamente

```
a = np.arange(36)
a2 = a.reshape((6,-1))
a4 = a.reshape((2,3,2,3))
```

```
print(a2[:,0])
print(a2[..., 0])
```

```
[ 0  6 12 18 24 30]
[ 0  6 12 18 24 30]
```

```
print(a4[0,:,:,:0])
print(a4[0,...,0])
```

```
[[ 0  3]
 [ 6  9]
 [12 15]]
[[ 0  3]
 [ 6  9]
 [12 15]]
```

```
(a4[...,0] == a4[:,:,:,:0]).all()
```

```
True
```

- Uso de `np.roll(a, shift, axis=None)` que mueve elementos una distancia `shift` a lo largo del eje `axis`, y cuando pasan la última posición los reintroduce al principio. Por ejemplo, en una dimensión:

```
x = np.arange(10)
print(x)
print(np.roll(x, 2))
```

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

15.1.3 Tercer ejemplo

Veamos un ejemplo similar al primero, pero donde vamos cambiando los límites de los ejes en forma manual, a medida que los datos lo requieren

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Calcula los datos en tiempo real.
def data_gen(t=0):
    cnt = 0
    while cnt < 1000:
        cnt += 1
        t += 0.1
        yield t, np.sin(2 * np.pi * t) * np.exp(-t / 10.)

# Necesitamos que se puede acceder a estas variables
# desde varias funciones -> globales
fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
xdata, ydata = [], []

def init():
    ax.grid()
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line, 

def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    # Si los datos salen del eje, agrandamos el eje
    # Despu s tenemos que redibujar el canvas manualmente
    if t >= xmax:
        ax.set_xlim(xmin, 2 * xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
```

(contin   en la pr  xima p  gina)

(provine de la página anterior)

```

return line,
ani = animation.FuncAnimation(fig, run, data_gen, blit=False,
                               interval=30, repeat=False, init_func=init)
plt.show()

```

```
%run animate_decay.py
```

```

plt.style.reload_library()
plt.style.use('default')

```

15.2 Ejercicios 14 (a)

1. Utilizando Matplotlib:

- Hacer un gráfico donde dibuje una parábola $y = x^2$ en el rango $[-5, 5]$.
- En el mismo gráfico, agregar un círculo en $x = -5$.
- El círculo debe moverse siguiendo la curva, como se muestra en la figura

2. PARA ENTREGAR. Caída libre 2:

Modificar el ejercicio de la clase 8 de caída libre que entregó, para aceptar dos nuevas opciones:

- La opción `--vx` permite dar una velocidad inicial en la dirección horizontal.
- La opción `--animate`, tal que cuando se utilice, el programa muestre una animación de la trayectoria.
- La animación tiene que tener un cartel indicando el tiempo, y la velocidad y altura correspondiente a ese tiempo.
- Agregue una “cola fantasma” a la partícula, que muestre posiciones anteriores.

Envíe el programa llamado **14_Suapellido.py** en un adjunto por correo electrónico, con asunto: **14_Suapellido**

15.3 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.

```

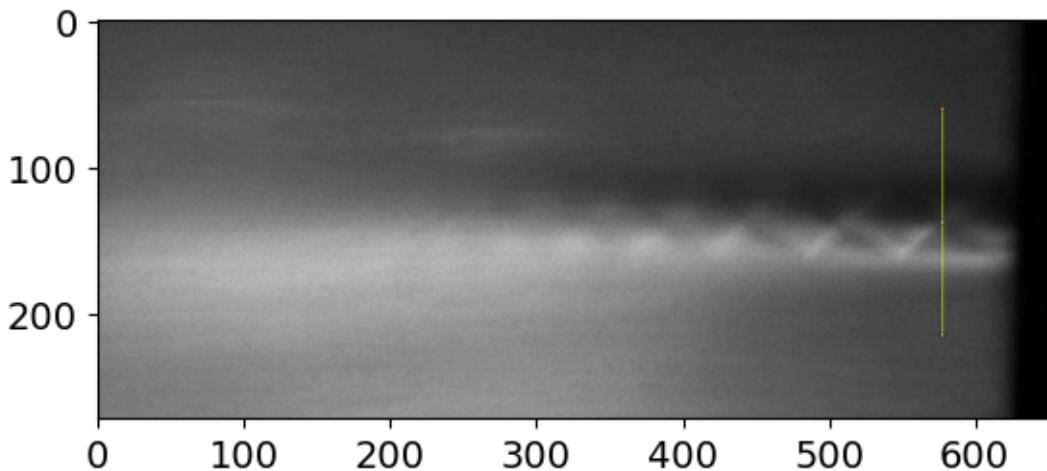
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from PIL import Image

```

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 = plt.imread('figuras/imagen_flujo.jpg')
print(f'La imagen "imag1" es del tipo: {type(imag1)} con "shape" {imag1.shape}')
plt.imshow(imag1);
```

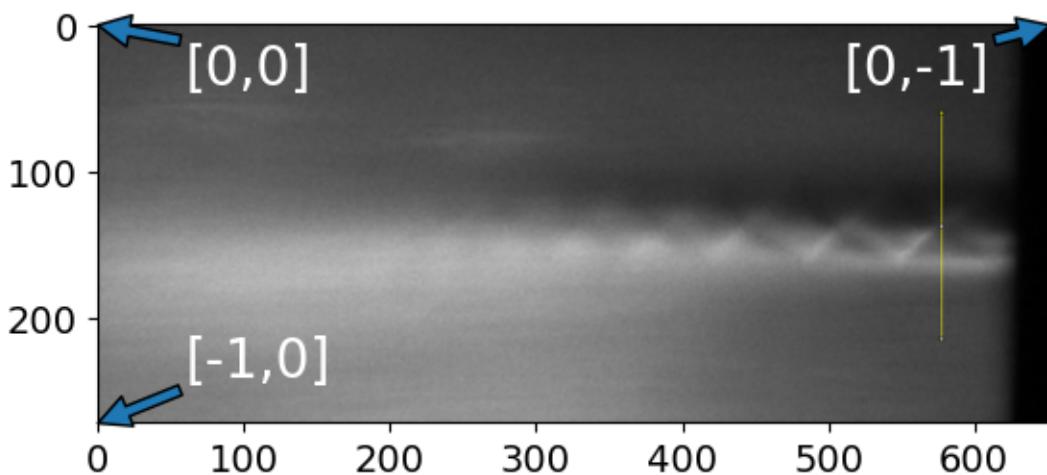
```
La imagen "imag1" es del tipo: <class 'numpy.ndarray'> con "shape" (272, 652, 3)
```



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:

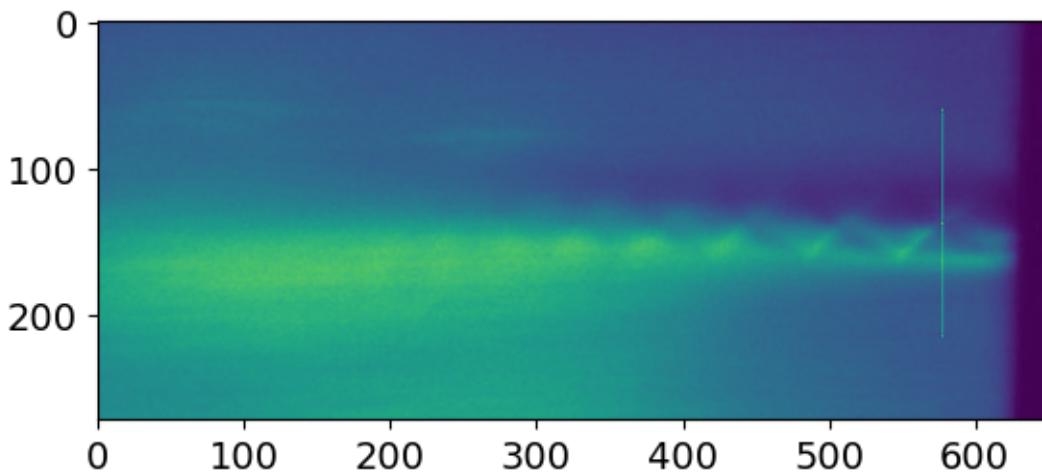
```
imag2 = plt.imread('figuras/imagen_flujo_gray.jpg')
```

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
```

```
plt.imshow(imag2);
```

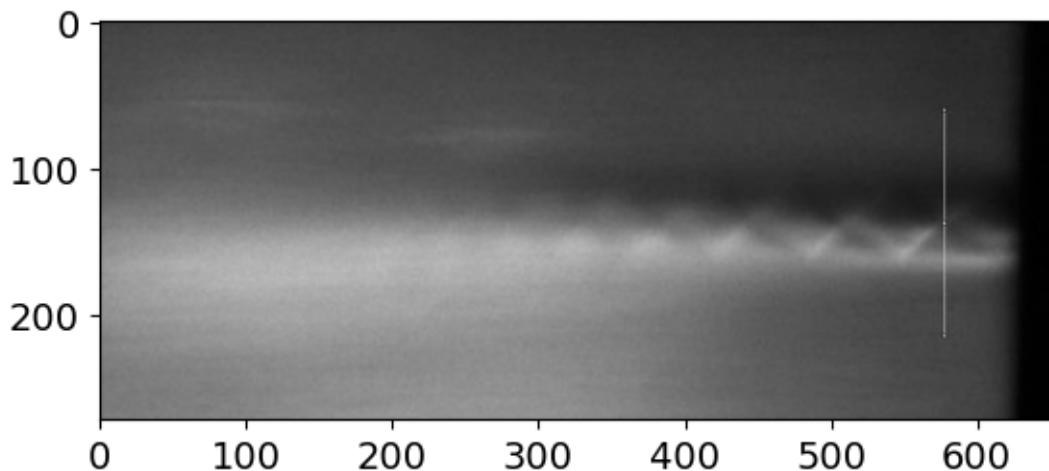


Nota ¿Qué pasó acá?

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:

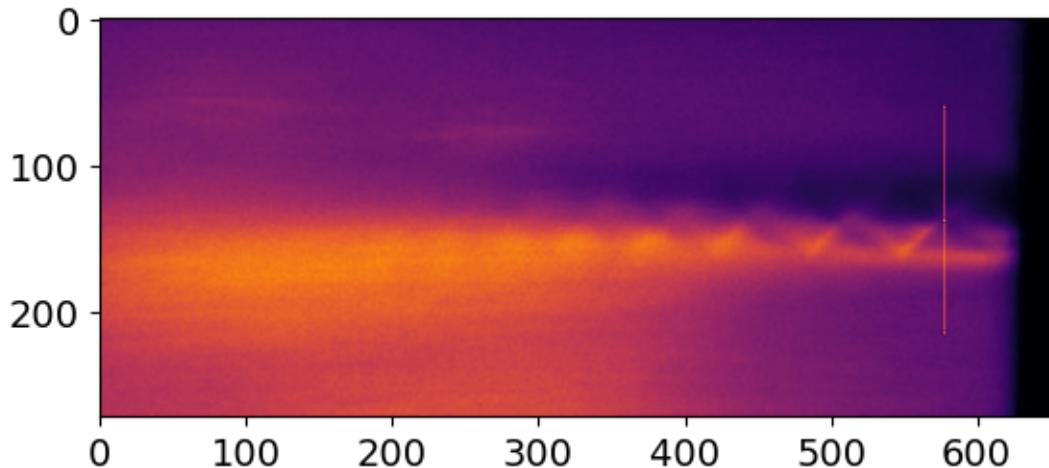
```
plt.imshow(imag2, cmap='gray');
```



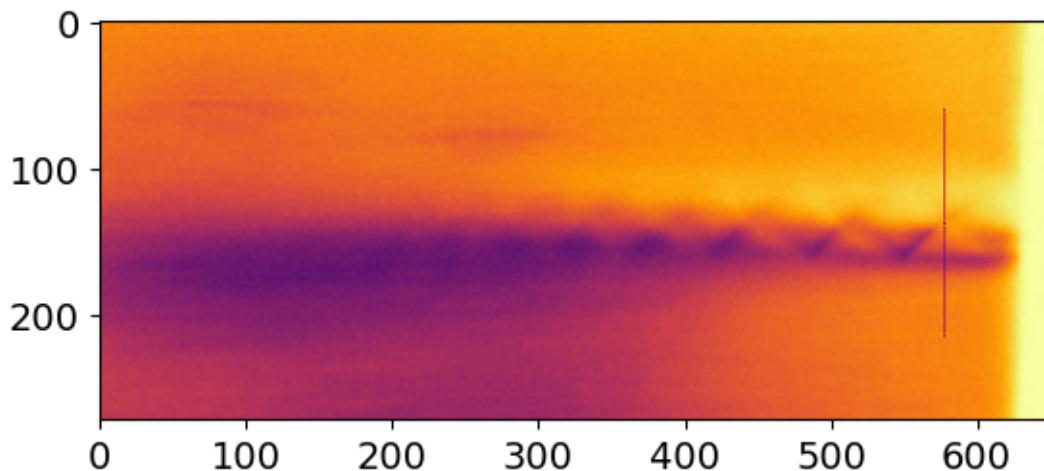
Los `cmap` aparecen también en el contexto de graficación de gráficos de contorno (`contour()` y `contourf()`).

Al asociar un valor a un mapa de colores, la misma imagen puede mostrarse de diferente maneras. Veamos otros ejemplos de *colormap*

```
plt.imshow(imag2, cmap='inferno');
```

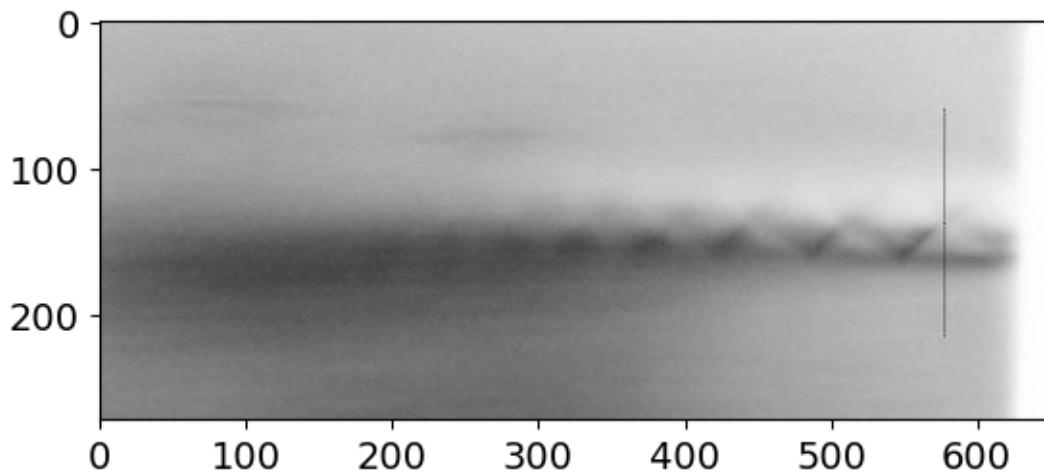


```
plt.imshow(imag2, cmap='inferno_r');
```



La referencia de ubicación de los `cmap` existentes está en: http://matplotlib.org/examples/color/colormaps_reference.html

```
plt.imshow(imag2, cmap='gray_r');
```



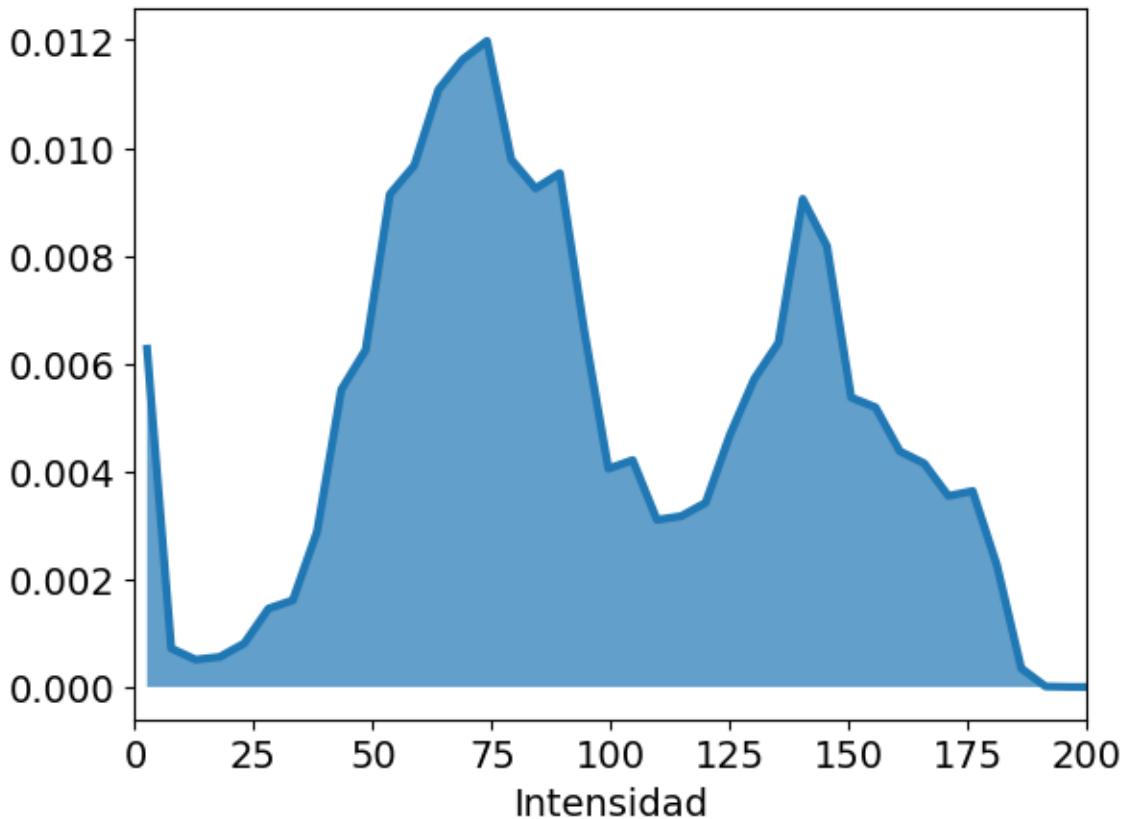
15.3.1 Análisis de la imagen

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.

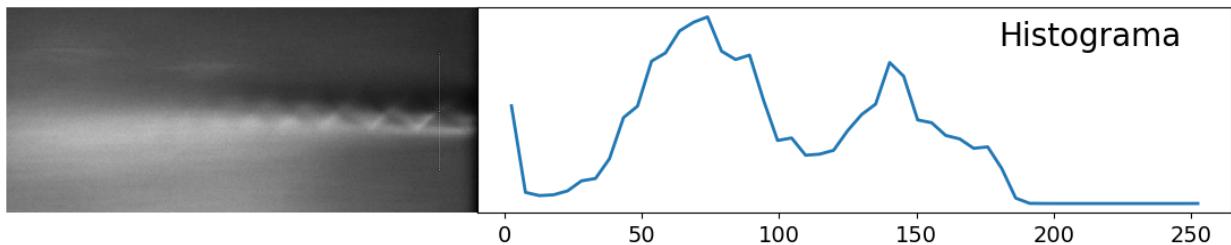
Histograma de intensidades

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.

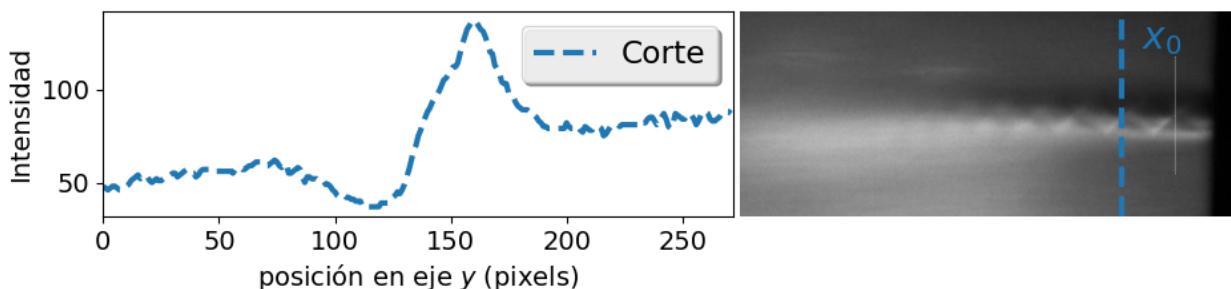
Cortes en una dirección

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(f'posición en eje x={x0} de un total de {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)
```



15.4 Ejercicios 14 (b)

3. 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.
-

15.5 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)

15.5.1 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)

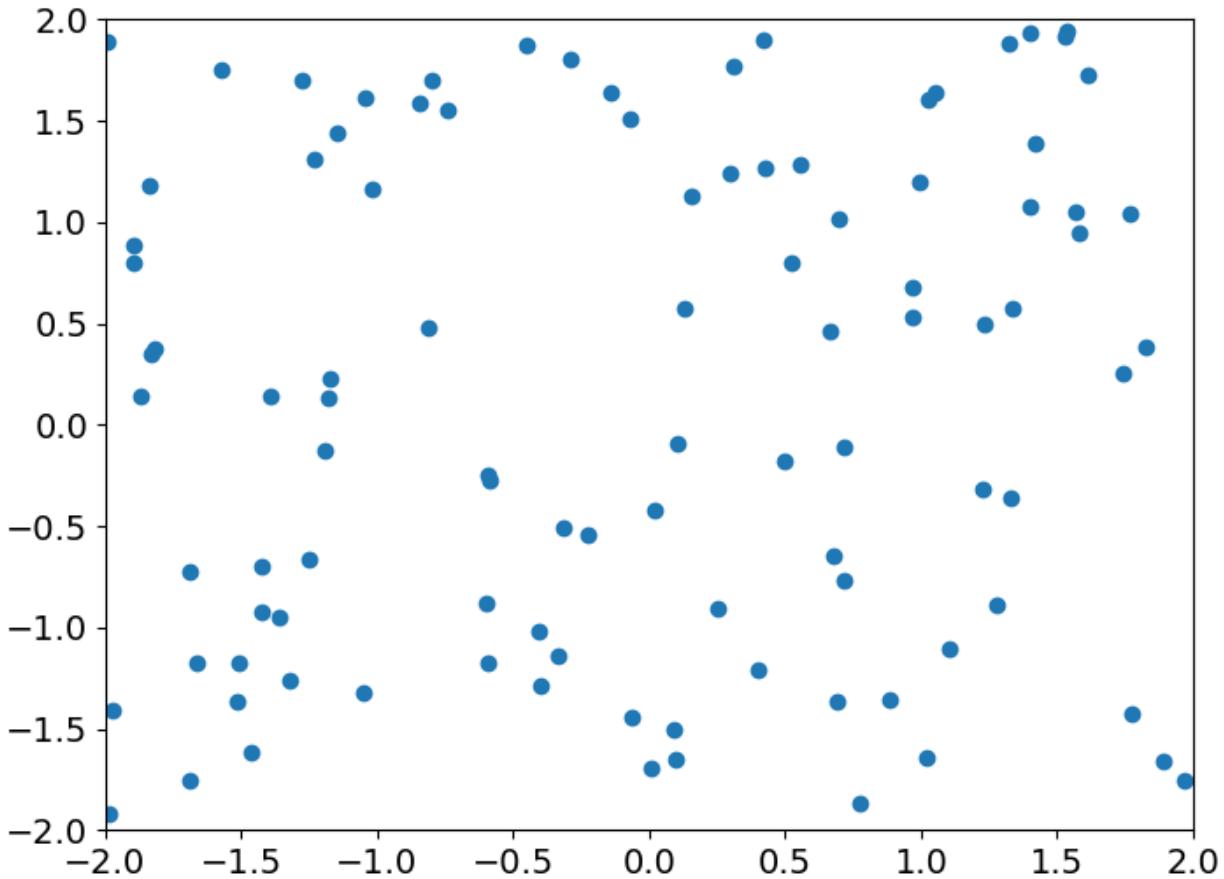
# Usamos: useblit = True en el backend gtkagg
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)

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)
```

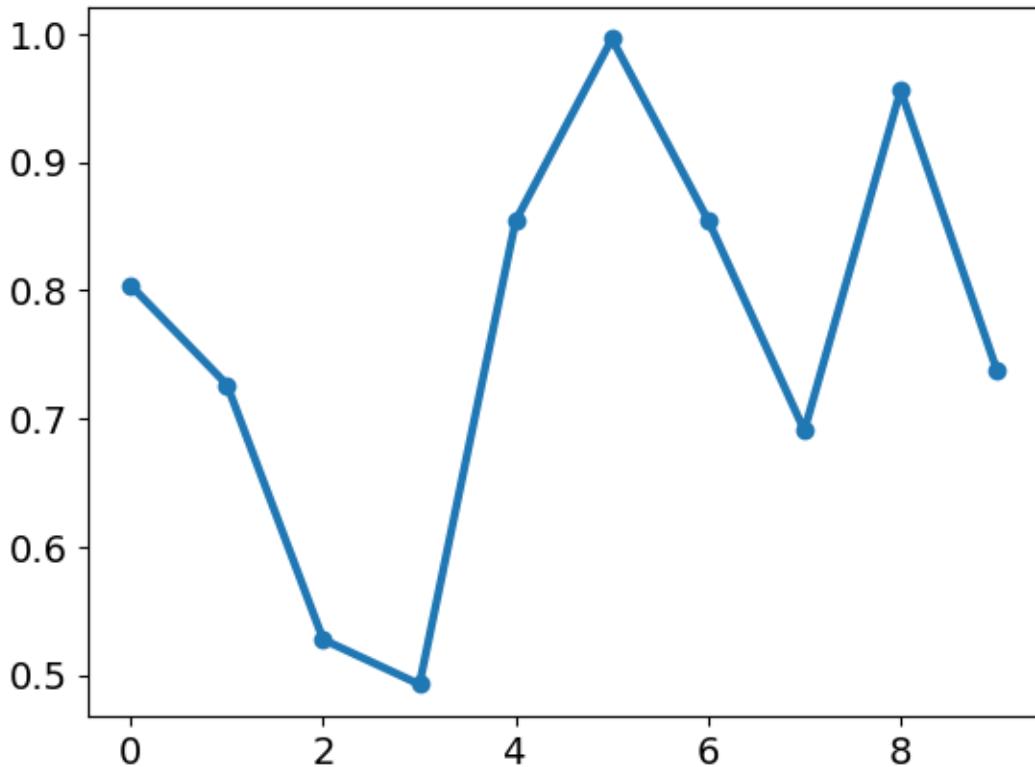
y toma como argumento el eje en el cuál agregamos el cursor. Como argumentos opcionales se puede controlar la visibilidad de la línea horizontal (horizOn) y vertical (vertOn) que pueden tomar valores lógicos True o False. Además tiene argumentos *keyword* para controlar la apariencia de las líneas. En este ejemplo pusimos explícitamente que queremos una línea roja con un grosor igual a 2.

15.5.2 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
```



```
<Figure size 640x480 with 0 Axes>
```

```
# 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()
```

```

IPython:clases-pyton/clases
fiol > ... / pythons / clases-pyton / clases > ipython3
Python 3.7.6 (default, Jan 30 2020, 09:44:41)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.11.1 -- An enhanced Interactive Python. Type '?' for help.

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.

15.5.3 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
from matplotlib.widgets import Cursor

```

(continué en la próxima página)

(proviene de la página anterior)

```
img = plt.imread('..../figuras/imagen_flujo_gray.jpg')
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)
    l1.set_data([[n0, n0], [0., 1.]])
    l2.set_data(range(img.shape[0]), img[:, n0])
    l1.figure.canvas.draw()
    l2.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
l1 = ax1.axvline(x0, color='r', ls='--', lw=3)

# Grafico de la derecha
l2, = ax2.plot(img[:, x0], 'r-', lw=2, label='corte')
ax2.set_xlim(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 matplotlib.pyplot as plt
from scipy import misc
from matplotlib.widgets import Cursor

img = plt.imread('../figuras/imagen_flujo_gray.jpg')
ymax = img.max()

def click(event):
    """Secuencia:
    1. Encuentro el punto donde el mouse hizo 'click'
    2. Le doy valores a la línea 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 línea inicial en un valor inicial
x0 = 100
```

(continué en la próxima página)

(proviene de la página anterior)

```
11 = ax1.axvline(x0, color='r', ls='--', lw=3)

# Grafico de la derecha
12, = 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')

fig.tight_layout()

# Agrego el cursor y conecto la acción de presionar a la función 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))
```

15.6 Ejercicios 14 (c)

4. 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

Clase 15: Interfaces con otros lenguajes

16.1 Interface con lenguaje C

Existen varias formas de utilizar bibliotecas o códigos hechos en C desde Python. Nosotros veremos el uso de `Ctypes`, sin embargo existen otras alternativas como `Cython`, `CFI`, `pybind11` y `Boost.Python`.

16.1.1 Ejemplo 1: Problema a resolver

Supongamos que queremos resolver el problema de la rotación de vectores en el espacio usando los tres ángulos de Euler.

```
import numpy as np
pwd
'/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/curso-
→python'
```

Si ya tenemos un módulo donde están programadas las funciones necesarias

```
# %load rotacion_p.py
#!/usr/bin/ipython3
import numpy as np

def matrix_rotation(angles):
    cx, cy, cz = np.cos(angles)
    sx, sy, sz = np.sin(angles)
    R = np.zeros((3, 3))
    R[0, 0] = cx * cz - sx * cy * sz
    R[0, 1] = cx * sz + sx * cy * cz
    R[0, 2] = sx * sy
```

(continué en la próxima página)

(proviene de la página anterior)

```
R[1, 0] = -sx * cz - cx * cy * sz
R[1, 1] = -sx * sz + cx * cy * cz
R[1, 2] = cx * sy

R[2, 0] = sy * sz
R[2, 1] = -sy * cz
R[2, 2] = cy
return R

def rotate(angles, v):
    return np.dot(matrix_rotation(angles), v)
```

es fácil utilizarlas. Las importamos y utilizamos

```
# import rotacion_p as rotp
N = 100
# Ángulos de Euler
angle = np.random.random(3)
# Definimos N vectores tridimensionales
v = np.random.random((3, N))
```

```
# y= rotpp.rotate(angle, v)
y = rotate(angle,v)
```

```
print(angle)
print(y[:,0:5].T)
```

```
[0.50865471 0.8181404 0.53468601]
[[ 1.14137158 -0.09559654 -0.105005 ]
 [ 1.00659971 -0.09141267 -0.14818529]
 [ 0.96071663 -0.39351893  0.09300136]
 [ 0.98322967 -0.12991836 -0.05407704]
 [ 0.75230624  0.10485721  0.80743319]]
```

16.1.2 Interfaces con C

Veamos cómo trabajar si tenemos el código para realizar las rotaciones en C.

Primer ejemplo: Nuestro código

El código en C que tenemos es:

```
typedef struct {
    float m[3][3];
} m3x3;

typedef struct {
    float a[3];
} v3;
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
float * rotate(float angles[3], float *v, int N) {

    m3x3 R = matrix_rotation(angles);

    float* y = (float*)malloc(3*N*sizeof(float));
    v3 p;

    printf("%p\n", y);
    for(int i=0; i<N; i++) {
        // p = &y[i*3];
        p = matmul3(R, &v[i*3]);
        y[i*3+0] = p.a[0];
        y[i*3+1] = p.a[1];
        y[i*3+2] = p.a[2];
        // printf("%6.3f %6.3f %6.3f \n", y[i*3+0], y[i*3+1], y[i*3+2]);
    }
    return y;
}
```

```
cd interfacing_C
```

```
/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/curso-
→python/interfacing_C
```

```
!cat rotacion.c
```

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

typedef struct {
    float m[3][3];
} m3x3;

typedef struct {
    float a[3];
} v3;

// !> matrix_rotation
// !! Crea la matriz de rotación correspondiente a los tres ángulos de Euler
// !!
// !! @param angles
// !! @return R

m3x3 matrix_rotation(float angles[3]) {

    m3x3 R;

    float cx = cos(angles[0]);
```

```

float cy = cos(angles[1]);
float cz = cos(angles[2]);
float sx = sin(angles[0]);
float sy = sin(angles[1]);
float sz = sin(angles[2]);

R.m[0][0] = cx*cz - sx*cy*sz;
R.m[0][1] = cx*sz + sx*cy*cz;
R.m[0][2] = sx*sy;

R.m[1][0] = -sx*cz - cx*cy*sz;
R.m[1][1] = -sx*sz + cx*cy*cz;
R.m[1][2] = cx*sy;

R.m[2][0] = sy*sz;
R.m[2][1] = -sy*cz;
R.m[2][2] = cy;

return R;
}

v3 matmul3(m3x3 A, float b[3]) {

static v3 a;

for(int i=0; i < 3; i++){
    a.a[0] = A.m[0][0] * b[0] + A.m[0][1] * b[1] + A.m[0][2] * b[2];
    a.a[1] = A.m[1][0] * b[0] + A.m[1][1] * b[1] + A.m[1][2] * b[2];
    a.a[2] = A.m[2][0] * b[0] + A.m[2][1] * b[1] + A.m[2][2] * b[2];
}
// printf("%6.3f %6.3f %6.3f n",a[0],a[1],a[2]);

return a;
}

float * rotate(float angles[3], float v, int N) {

m3x3 R = matrix_rotation(angles);

float y = (float*)malloc(3*N*sizeof(float));
v3 p;

printf("%pn",y);
for(int i=0; i<N; i++){
    // p = &y[i*3];
    p = matmul3(R,&v[i*3]);
    y[i*3+0] = p.a[0];
    y[i*3+1] = p.a[1];
    y[i*3+2] = p.a[2];
    // printf("%6.3f %6.3f %6.3f n",y[i*3+0],y[i*3+1],y[i*3+2]);
}

```

```

    }

    return y;
}

int main() {
    float angle[3] = { 0.0f, 0.0f, 3.141592f/2.0f};

    m3x3 R;

    int NDIM = 2;
    float* v = (float*)malloc(3*NDIM*sizeof(float));

    float *y;

    for (size_t j = 0; j < NDIM; j++)
    {
        v[j*3+0] = (j*3+0)*1.0f;
        v[j*3+1] = (j*3+1)*1.0f;
        v[j*3+2] = (j*3+2)*1.0f;
    }

    v[0] = (1)*1.0f;
    v[1] = (0)*1.0f;
    v[2] = (0)*1.0f;
    v[3] = (0)*1.0f;
    v[4] = (1)*1.0f;
    v[5] = (0)*1.0f;

    R = matrix_rotation(angle);

    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            printf("%6.3f",R.m[i][j]);
            printf("   ");
        }
        printf("\n");
    }

    y = rotate(angle,v,NDIM);
    printf("%6.3f %6.3f %6.3f n",y[0],y[1],y[2]);
    printf("%6.3f %6.3f %6.3f n",y[3],y[4],y[5]);
    // for (size_t i = 0; i < NDIM; i++)
    // {
    //     printf("%6.3f %6.3f %6.3f n",y[i*3],y[i*3+1],y[i*3+2]);
    // }
}

```

CTypes

No vamos a usar directamente Ctypes, sino a través de NumPy, que provee algunas funciones convenientes para acceder al código C.

El primer paso es compilar nuestro código y generar una biblioteca:

```
$ gcc -fpic -Wall -shared rotacion.c -o librotacion.so
```

Si uno trabaja en Windows, generará una dll

```
cl.exe -c rotacion.c  
link.exe /DLL /OUT:rotacion.dll
```

```
!gcc -fpic -Wall -shared rotacion.c -o librotacion.so
```

```
!ls
```

```
[31mlibrotacion.so[m[m rotacion.c
```

En segundo lugar, importamos el módulo ctypeslib

```
import numpy.ctypeslib as ctl
```

Este módulo nos provee de la función `load_library` para importar la biblioteca

```
help(ctl.load_library)
```

```
Help on function load_library in module numpy.ctypeslib:  
  
load_library(libname, loader_path)  
    It is possible to load a library using  
    >>> lib = ctypes.cdll[<full_path_name>] # doctest: +SKIP  
  
    But there are cross-platform considerations, such as library file extensions,  
    plus the fact Windows will just load the first library it finds with that name.  
    NumPy supplies the load_library function as a convenience.  
  
Parameters  
-----  
libname : str  
    Name of the library, which can have 'lib' as a prefix,  
    but without an extension.  
loader_path : str  
    Where the library can be found.  
  
Returns  
-----  
ctypes.cdll[libpath] : library object  
    A ctypes library object  
  
Raises  
-----  
OSError  
    If there is no library with the expected extension, or the  
    library is defective and cannot be loaded.
```

```
rotc = ctl.load_library('librotacion.so', '.')
```

Una vez cargada la biblioteca, tenemos que definir adecuadamente cómo pasar los argumentos a la función `rotate` de C:

```
float * rotate(float angles[3], float *v, int N)
```

Para eso se utiliza la función `argtypes` que recibe una lista de tipos. Notemos que los dos primeros argumentos son arreglos de C (o sea, punteros), mientras que el último es un entero.

```
npflags = ['C_CONTIGUOUS'] # Requires a C contiguous array in memory

float_1d_type = ctl.ndpointer(dtype=np.float32, ndim=1, flags=npflags) # Puntero a float, 1D
float_2d_type = ctl.ndpointer(dtype=np.float32, ndim=2, flags=npflags) # Puntero a float, 2D
```

Con estos tipos de datos, defino los tipos de argumentos, que son tres en total. El último es un dato de tipo entero, para lo cual se usa directamente `c_intp`.

```
rotc.rotate.argtypes = [float_1d_type, float_2d_type, ctl.c_intp]
```

Hagamos un ejemplo sencillo con $N=2$

```
# import rotacion_p as rotp
N = 2
# Ángulos de Euler
angle = np.random.random(3).astype(np.float32)
# Definimos N vectores tridimensionales
v = np.random.random((3, N)).astype(np.float32)
```

Las funciones que dispongo en C reciben tipos `float`, es decir que me tengo que asegurar esto a través del método `astype`.

Ahora tenemos que definir el tipo de dato de salida, que retorna C a través de un puntero a `float`, `float*`. Para esto usamos el método `restype`. Como a priori no sé qué tipo de rango tiene mi arreglo de salida, tengo que definirlo explícitamente.

```
rotc.rotate.restype = ctl.ndpointer(dtype=np.float32, shape=(N, 3))
```

Hay que tener precaución con el manejo de arreglos, que es muy distinto en C y Python. En Python son objetos, de los cuales yo puedo tener distintas vistas, slices, etc. Hay que recordar que en principio estas son formas de acceder al mismo objeto, pero no se pueden traducir directamente a C, que necesita un arreglo contiguo de datos.

```
v = np.array([[1, 0], [0, 1], [0, 0]]).astype(np.float32)
vt = v.T.copy()

print(np.shape(v))
print(np.shape(v.T))
```

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

Veamos, `v` es un arreglo de 3 filas y 2 columnas, que contiene *dos* vectores de tres dimensiones que se desean rotar, organizados como columnas. Esto *no* es lo que necesito mi arreglo en C, que es tiene los vectores organizados contiguamente en un solo arreglo unidimensional. Entonces, tengo que transformarlo. Para eso usamos el `.T`. Ojo que además, hay que crear un objeto nuevo con `copy()`, sino es una vista del mismo objeto `v`.

```
angle90 = np.array([0, 0, np.pi/2], dtype = np.float32)
print(angle90)
```

```
[0. 0. 1.5707964]
```

```
yf = rotc.rotate(angle90,
                  vt,
                  N)
y = rotate(angle90, v)
```

```
np.set_printoptions(suppress=True)

print(y)
print(yf.T)
np.allclose(y, yf.T)
```

```
[[-0.00000004 1.
 [-1. -0.00000004]
 [ 0. 0. ] ]
 [[-0.00000004 1.
 [-1. -0.00000004]
 [ 0. 0. ] ]]
```

```
True
```

16.2 Interface con lenguaje Fortran

16.2.1 Ejemplo 1: Problema a resolver

Supongamos que queremos resolver el problema de la rotación de vectores en el espacio usando los tres ángulos de Euler.

```
import numpy as np
```

```
pwd
```

```
'/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/curso-
→python'
```

Si ya tenemos un módulo donde están programadas las funciones necesarias

```
# %load rotacion_p.py
#!/usr/bin/ipython3
import numpy as np

def matrix_rotation(angles):
    cx, cy, cz = np.cos(angles)
    sx, sy, sz = np.sin(angles)
    R = np.zeros((3, 3))
    R[0, 0] = cx * cz - sx * cy * sz
```

(continué en la próxima página)

(proviene de la página anterior)

```
R[0, 1] = cx * sz + sx * cy * cz
R[0, 2] = sx * sy

R[1, 0] = -sx * cz - cx * cy * sz
R[1, 1] = -sx * sz + cx * cy * cz
R[1, 2] = cx * sy

R[2, 0] = sy * sz
R[2, 1] = -sy * cz
R[2, 2] = cy
return R

def rotate(angles, v):
    return np.dot(matrix_rotation(angles), v)
```

es fácil utilizarlas. Las importamos y utilizamos

```
# import rotacion_p as rotp
N = 100
# Ángulos de Euler
angle = np.random.random(3)
# Definimos N vectores tridimensionales
v = np.random.random((3, N))
```

```
# y= rotp.rotate(angle, v)
y = rotate(angle,v)
```

```
print(angle)
print(y[:,0:5].T)
```

```
[0.36341885 0.58028317 0.4486011 ]
[[ 4.22770593e-01  3.33318934e-01 -5.49571740e-04]
 [ 6.05335674e-01  4.34396149e-01  6.87862512e-01]
 [ 9.84128217e-01  4.35323746e-01 -1.62964229e-01]
 [ 3.17757248e-01  4.72584037e-01  3.75531685e-01]
 [ 7.35722302e-01  1.58547744e-01  6.73141163e-01]]
```

16.2.2 Interfaces con Fortran

Veamos cómo trabajar si tenemos el código para realizar las rotaciones en Fortran

Primer ejemplo: Nuestro código

El código en Fortran que tenemos es:

```
function rotate(theta, v, N) result(y)
  implicit none
  integer :: N
  real(8), dimension(3), intent(IN) :: theta
  real(8), dimension(3,N), intent(IN) :: v
  real(8), dimension(3,N) :: y
```

(continué en la próxima página)

(proviene de la página anterior)

```

real(8), dimension(3,3) :: R
real(8) :: cx, cy, cz, sx, sy, sz

! Senos y Cosenos de los tres ángulos de Euler
cx = cos(theta(1)); cy = cos(theta(2)); cz = cos(theta(3))
sx = sin(theta(1)); sy = sin(theta(2)); sz = sin(theta(3))

! Matriz de rotación
R(1,1) = cx*cz - sx*cy*sz
R(1,2) = cx*sz + sx*cy*cz
R(1,3) = sx*sy

R(2,1) = -sx*cz - cx*cy*sz
R(2,2) = -sx*sz + cx*cy*cz
R(2,3) = cx*sy

R(3,1) = sy*sz
R(3,2) = -sy*cz
R(3,3) = cy

! Aplicamos la rotación
y = matmul(R, v)
end function rotate

```

```
cd interfacing_F
```

```
/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/curso-
→python/interfacing_F
```

```
cd interfacing_F
!cat rotacion.f90
```

```

module rotaciones

contains
  !> matrix_rotation
  !! Crea la matriz de rotación correspondiente a los tres ángulos de Euler
  !!
  !! @param angles
  !! @return R
  function matrix_rotation(angles) result(R)
    implicit none
    real(8), dimension(3), intent(IN) :: angles
    real(8), dimension(3,3) :: R
    real(8) :: cx, cy, cz, sx, sy, sz

    cx = cos(angles(1)); cy = cos(angles(2)); cz = cos(angles(3))
    sx = sin(angles(1)); sy = sin(angles(2)); sz = sin(angles(3))

    R(1,1) = cx*cz - sx*cy*sz
    R(1,2) = cx*sz + sx*cy*cz
    R(1,3) = sx*sy

    R(2,1) = -sx*cz - cx*cy*sz

```

```

R(2,2) = -sx*sz + cx*cy*cz
R(2,3) = cx*sy

R(3,1) = sy*sz
R(3,2) = -sy*cz
R(3,3) = cy
end function matrix_rotation

!> rotate
!!
!! @param angles
!! @param v
!! @param N longitud del vector v
!! @return y
function rotate(angles, v, N) result(y)
    implicit none
    integer, intent(IN) :: N
    real(8), dimension(3), intent(IN) :: angles
    real(8), dimension(3,N), intent(IN) :: v
    real(8), dimension(3,N) :: y

    y = matmul(matrix_rotation(angles), v)
end function rotate

end module rotaciones

program test_rotation
    USE rotaciones !, only: rotate
    implicit none
    integer, parameter :: Ndim=1000
    real(8), dimension(3) :: angle
    real(8), dimension(3, Ndim) :: v, y

    integer :: n, Nloops,i,j
    real(8) :: time_begin, time_end

    real(8), dimension(3, 3) :: R

    ! Selecciono Ndim vectores tridimensionales al azar
    call RANDOM_NUMBER(v)
    Nloops = 10000

    CALL CPU_TIME ( time_begin )
    do n=1,Nloops
        call RANDOM_NUMBER(angle)
        y= rotate(angle, v, Ndim)
    end do
    CALL CPU_TIME ( time_end )

    write (,'(A,1PE12.2,A)') '# Tiempo usado:', (time_end - time_begin)/(1.
    ↪e-3*Nloops), ' milisegundos por 1000 rotaciones'

    ! Test a simple case

```

```
! angle(1) = 1.0
! angle(2) = 2.0
! angle(3) = 3.0

! R = matrix_rotation(angle)
! ! do i=1,3
! !   write(*,'(3F8.3)') R(i,1),R(i,2),R(i,3)
! ! end do

! do j=0,4
!     v(1,j+1) = j*3+0
!     v(2,j+1) = j*3+1
!     v(3,j+1) = j*3+2
! end do
! y= rotate(angle, v, 5)

! do i=1,5
!   write(*,'(3F8.3)') y(1,i),y(2,i),y(3,i)
! end do

end program test_rotation
```

F2PY

F2PY -Fortran to Python interface generator- es una utilidad que permite generar una interfaz para utilizar funciones y datos definidos en Fortran desde Python.

Información sobre la interfaz entre Fotran y Python, y en particular sobre F2PY, puede encontrarse en:

- [Scipy cookbook](#)
- [F2PY Users Guide and Reference Manual](#)
- [Fortran Best Practices](#)
- http://websrv.cs.umt.edu/isis/index.php/F2py_example

El primer paso es utilizar esta utilidad:

```
$ f2py3 -c rotacion.f90 -m rotacion_f
```

```
cd interfacing_F
```

```
/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/curso-
→python/interfacing_F
```

```
!f2py3 -c rotacion.f90 -m rotacion_f
```

```
[39mrunning build[0m
[39mrunning config_cc[0m
[39muniforming config_cc, config, build_clib, build_ext, build commands ↴
→--compiler options[0m
[39mrunning config_fc[0m
[39muniforming config_fc, config, build_clib, build_ext, build commands ↴
→--fcompiler options[0m
```

```
[39mrunning build_src[0m
[39mbuild_src[0m
[39mbuilding extension "rotacion_f" sources[0m
[39mf2py options: [] [0m
[39mf2py:> /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.
↳macosx-10.9-x86_64-3.9/rotacion_fmodule.c[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.
↳macosx-10.9-x86_64-3.9[0m
Reading fortran codes...
    Reading file 'rotacion.f90' (format:free)
Post-processing...
    Block: rotacion_f
        Block: rotaciones
            Block: matrix_rotation
appenddecl: "dimension" not implemented.
            Block: rotate
appenddecl: "dimension" not implemented.
            Block: test_rotation
Post-processing (stage 2)...
    Block: rotacion_f
        Block: unknown_interface
            Block: rotaciones
                Block: matrix_rotation
                Block: rotate
            Block: test_rotation
Building modules...
    Building module "rotacion_f"...
        Constructing F90 module support for "rotaciones"...
        Creating wrapper for Fortran function "matrix_rotation"("matrix_
↳rotation")...
            Constructing wrapper function "rotaciones.matrix_"
↳rotation"...
                r = matrix_rotation(angles)
        Creating wrapper for Fortran function "rotate"("rotate")...
            Constructing wrapper function "rotaciones.rotate"...
                y = rotate(angles,v,[n])
    Wrote C/API module "rotacion_f" to file "/var/folders/18/
↳8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/
↳rotacion_fmodule.c"
        Fortran 90 wrappers are saved to "/var/folders/18/8wj5kxln1dzbpqvfs_
↳35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/rotacion_
↳f-f2pywrappers2.f90"
[39m adding '/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/
↳src.macosx-10.9-x86_64-3.9/fortranobject.c' to sources.[0m
[39m adding '/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/
↳src.macosx-10.9-x86_64-3.9' to include_dirs.[0m
[39mcopying /Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/f2py/
↳src/fortranobject.c -> /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9[0m
[39mcopying /Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/f2py/
↳src/fortranobject.h -> /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9[0m
[39m adding '/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/
↳src.macosx-10.9-x86_64-3.9/rotacion_f-f2pywrappers2.f90' to sources.[0m
```

```
[39mbuild_src: building npy-pkg config files[0m
[39mrunning build_ext[0m
[39mcustomize UnixCCompiler[0m
[39mcustomize UnixCCompiler using build_ext[0m
[39mget_default_fcompiler: matching types: '['gnu95', 'nag', 'absoft', 'ibm',_
↪'intel', 'gnu', 'g95', 'pg']'[0m
[39mcustomize Gnu95FCompiler[0m
[39mFound executable /usr/local/bin/gfortran[0m
[39mcustomize Gnu95FCompiler[0m
[39mcustomize Gnu95FCompiler using build_ext[0m
[39mbuilding 'rotacion_f' extension[0m
[39mcompiling C sources[0m
[39mC compiler: clang -Wno-unused-result -Wsign-compare -Wunreachable-code_
↪--DNDEBUG -fwrapv -O2 -Wall -fPIC -O2 -isystem /Users/flavioc/miniconda3/
↪include -arch x86_64 -I/Users/flavioc/miniconda3/include -fPIC -O2 -isystem_
↪/Users/flavioc/miniconda3/include -arch x86_64
[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/
↪var[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/
↪folders[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/
↪folders/18[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/
↪folders/18/8wj5kxln1dzbpqvfs_35mz00000gn[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/
↪folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/
↪folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl[0m
[39mcreating /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/
↪folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86-
↪64-3.9[0m
[39mcompile options: '-DNPY_DISABLE_OPTIMIZATION=1 -I/var/folders/18/
↪8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9_
↪--I/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include_
↪--I/Users/flavioc/miniconda3/include/python3.9 -c'[0m
[39mclang: /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.
↪macosx-10.9-x86_64-3.9/fortranobject.c[0m
[39mclang: /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.
↪macosx-10.9-x86_64-3.9/rotacion_fmodule.c[0m
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↪tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/rotacion_fmodule.c:16:
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↪tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/fortranobject.h:13:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↪numpy/core/include/numpy/arrayobject.hIn file included from :/var/folders/
↪18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/
↪fortranobject.c4:::
2In file included from :
/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/
↪include/numpy/ndarrayobject.hIn file included from :/var/folders/18/
↪8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/
↪fortranobject.h12:::
13In file included from :
```

```

/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/
↳numpy/ndarraytypes.h In file included from :/Users/flavioc/miniconda3/lib/
↳python3.9/site-packages/numpy/core/include/numpy/arrayobject.h 1969:::
4 [1m:
/Users/flavIn file included from /Users/flavioc/miniconda3/lib/python3.9/
↳site-packages/numpy/core/include/numpy/ndarrayobject.h:12:
ioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/numpy/npy_1_7_
↳deprecated_api.h In file included from /Users/flavioc/miniconda3/lib/python3.
↳9/site-packages/numpy/core/include/numpy/ndarraytypes.h:1969:
[1m:17:2/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/
↳include/numpy/npy_1_7_deprecated_api.h: :[0m 17[0;1;35m:warning 2: :[0m
↳[1m[0m"Using deprecated NumPy API, disable it with " "#define NPY_
↳NO_DEPRECATED_API NPY_1_7_API_VERSION" [-W#warnings] [0;1;35m[0m warning
: [0m[1m"Using deprecated NumPy API, disable it with " "#define NPY_
↳NO_DEPRECATED_API NPY_1_7_API_VERSION" [-W#warnings] [0m
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^
[0m#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^
[0m[1m/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.
↳macosx-10.9-x86_64-3.9/rotacion_fmodule.c:109:12: [0m[0;1;35m warning:_
↳[0m[1m unused function 'f2py_size' [-Wunused-function] [0m
static int f2py_size(PyArrayObject* var, ...)
[0;1;32m
[0m2 warnings generated.
1 warning generated.
[39mcompiling Fortran 90 module sources[0m
[39mFortran f77 compiler: /usr/local/bin/gfortran -Wall -g -ffixed-form_
↳-fno-second-underscore -arch x86_64 -fPIC -O3 -funroll-loops
Fortran f90 compiler: /usr/local/bin/gfortran -Wall -g -fno-second-underscore_
↳-arch x86_64 -fPIC -O3 -funroll-loops
Fortran fix compiler: /usr/local/bin/gfortran -Wall -g -ffixed-form_
↳-fno-second-underscore -Wall -g -fno-second-underscore -arch x86_64 -fPIC_
↳-O3 -funroll-loops[0m
[39mcompile options: '-I/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9 -I/Users/flavioc/miniconda3/lib/
↳python3.9/site-packages/numpy/core/include -I/Users/flavioc/miniconda3/
↳include/python3.9 -c'
extra options: '-J/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmp8h2z4zyl/ -I/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/
↳' [0m
[39mgfortran:f90: rotacion.f90[0m
[01m[Krotacion.f90:56:24:[m[K

      56 |   integer :: n, Nloops,i,j
      |                           [01;35m[K1[m[K
[01;35m[KWarning:[m[K Unused variable '[01m[Ki[m[K' declared at [01;
↳35m[K(1)[m[K [[01;35m[K8;;https://gcc.gnu.org/onlinedocs/gcc/
↳Warning-Options.html#index-Wunused-variable]8;;[m[K]
[01m[Krotacion.f90:56:26:[m[K

      56 |   integer :: n, Nloops,i,j
      |                           [01;35m[K1[m[K
[01;35m[KWarning:[m[K Unused variable '[01m[Kj[m[K' declared at [01;
↳35m[K(1)[m[K [[01;35m[K8;;https://gcc.gnu.org/onlinedocs/gcc/
↳Warning-Options.html#index-Wunused-variable]8;;[m[K]

```

```
[01m[Krotacion.f90:59:31:[m[K

    59 |     real(8), dimension(3, 3) :: R
        |
        [01;35m[K1[m[K
[01;35m[KWarning:[m[K Unused variable '[01m[Kr[m[K' declared at [01;
→35m[K(1)[m[K [[01;35m[K]8;;https://gcc.gnu.org/onlinedocs/gcc/
→Warning-Options.html#index-Wunused-variable]8;;[m[K]
[01m[Krotacion.f90:44:0:[m[K

    44 |         y = matmul(matrix_rotation(angles), v)
        |
[01;35m[KWarning:[m[K '[01m[K__var_1_mma.offset[m[K' is used uninitialized_
→in this function [[01;35m[K]8;;https://gcc.gnu.org/onlinedocs/gcc/
→Warning-Options.html#index-Wuninitialized-Wuninitialized]8;;[m[K]
[01m[Krotacion.f90:44:0:[m[K [01;35m[KWarning:[m[K '[01m[K__var_1_
→mma.dim[0].lbound[m[K' is used uninitialized in this function [[01;
→35m[K]8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
→#index-Wuninitialized-Wuninitialized]8;;[m[K]
[01m[Krotacion.f90:44:0:[m[K [01;35m[KWarning:[m[K '[01m[K__var_1_
→mma.dim[0].ubound[m[K' is used uninitialized in this function [[01;
→35m[K]8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
→#index-Wuninitialized-Wuninitialized]8;;[m[K]
[01m[Krotacion.f90:44:0:[m[K [01;35m[KWarning:[m[K '[01m[K__var_1_
→mma.dim[1].lbound[m[K' is used uninitialized in this function [[01;
→35m[K]8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
→#index-Wuninitialized-Wuninitialized]8;;[m[K]
[01m[Krotacion.f90:44:0:[m[K [01;35m[KWarning:[m[K '[01m[K__var_1_
→mma.dim[1].ubound[m[K' is used uninitialized in this function [[01;
→35m[K]8;;https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
→#index-Wuninitialized-Wuninitialized]8;;[m[K]
[39mcompiling Fortran sources[0m
[39mFortran f77 compiler: /usr/local/bin/gfortran -Wall -g -ffixed-form_
→-fno-second-underscore -arch x86_64 -fPIC -O3 -funroll-loops
Fortran f90 compiler: /usr/local/bin/gfortran -Wall -g -fno-second-underscore_
→-arch x86_64 -fPIC -O3 -funroll-loops
Fortran fix compiler: /usr/local/bin/gfortran -Wall -g -ffixed-form_
→-fno-second-underscore -Wall -g -fno-second-underscore -arch x86_64 -fPIC_
→-O3 -funroll-loops[0m
[39mcompile options: '-I/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9 -I/Users/flavioc/miniconda3/lib/
→python3.9/site-packages/numpy/core/include -I/Users/flavioc/miniconda3/
→include/python3.9 -c'
extra options: '-J/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmp8h2z4zyl/ -I/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/
→'[0m
[39mgfortran:f90: /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/rotacion_f-f2pywrappers2.f90[0m
[39m/usr/local/bin/gfortran -Wall -g -arch x86_64 -Wall -g -undefined_
→dynamic_lookup -bundle /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmp8h2z4zyl/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.
→macosx-10.9-x86_64-3.9/rotacion_fmodule.o /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/fortranobject.o /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/rotacion.o /var/
```

```

→folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/var/folders/18/
→8wj5kxln1dzbpqvfs_35mz00000gn/T/tmp8h2z4zyl/src.macosx-10.9-x86_64-3.9/
→rotacion_f-f2pywrappers2.o -L/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/gcc/
→x86_64-apple-darwin20/10.2.0 -L/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/
→gcc/x86_64-apple-darwin20/10.2.0/../../.. -L/usr/local/Cellar/gcc/10.2.
→_0_4/lib/gcc/10/gcc/x86_64-apple-darwin20/10.2.0/../../.. -lgfortran -o .
→rotacion_f.cpython-39-darwin.so[0m
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libgfortran.
→dylib) was built for newer macOS version (11.2) than being linked (10.9)
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libquadmath.
→dylib) was built for newer macOS version (11.2) than being linked (10.9)
Removing build directory /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmp8h2z4zyl

```

```
from rotacion_f import rotaciones as rotf
```

```
yf = rotf.rotate(angle, v)
print(y[:,0:5].T)
print(yf[:,0:5].T)
```

```
[[ 4.22770593e-01  3.33318934e-01 -5.49571740e-04]
 [ 6.05335674e-01  4.34396149e-01  6.87862512e-01]
 [ 9.84128217e-01  4.35323746e-01 -1.62964229e-01]
 [ 3.17757248e-01  4.72584037e-01  3.75531685e-01]
 [ 7.35722302e-01  1.58547744e-01  6.73141163e-01]]
[[ 4.22770593e-01  3.33318934e-01 -5.49571740e-04]
 [ 6.05335674e-01  4.34396149e-01  6.87862512e-01]
 [ 9.84128217e-01  4.35323746e-01 -1.62964229e-01]
 [ 3.17757248e-01  4.72584037e-01  3.75531685e-01]
 [ 7.35722302e-01  1.58547744e-01  6.73141163e-01]]
```

```
np.allclose(yf,y)
```

```
True
```

Veamos qué es exactamente lo que importamos:

```
np.info(rotf.rotate)
```

```
y = rotate(angles,v,[n])
```

Wrapper for rotate.

Parameters

angles : input rank-1 array('d') with bounds (3)

v : input rank-2 array('d') with bounds (3,n)

Other Parameters

n : input int, optional

Default: shape(v,1)

Returns

```
y : rank-2 array('d') with bounds (3,n) and rotate storage
```

Como vemos, estamos usando la función `rotate` definida en Fortran. Notar que:
* Tiene tres argumentos.
* Dos argumentos requeridos: `angles` y `v`
* Un argumento, correspondiente a la dimensión `n` que F2PY automáticamente detecta como opcional.

Segundo Ejemplo: Código heredado

La conversión que realizamos con f2py3 la podríamos haber realizado en dos pasos:

```
$ f2py3 rotacion.f90 -m rotacion_f -h rotacion.pyf
$ f2py3 -c rotacion.pyf -m rotacion_f
```

En el primer paso se crea un archivo *signature* que después se utiliza para crear el módulo que llamaremos desde **Python**. Haciéndolo en dos pasos nos permite modificar el texto del archivo *.pyf* antes de ejecutar el segundo comando.

Esto es útil cuando el código original no es lo suficientemente “moderno”, no tiene toda la información necesaria sobre los argumentos o es un código que uno no quiere o puede editar. Veamos que forma tienen con un ejemplo más simple (tomado de la [guía de usuario](#)):

```
SUBROUTINE FIB(A,N)
C
C      CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
      DO I=1,N
        IF (I.EQ.1) THEN
          A(I) = 0.0D0
        ELSEIF (I.EQ.2) THEN
          A(I) = 1.0D0
        ELSE
          A(I) = A(I-1) + A(I-2)
        ENDIF
      ENDDO
      END
```

```
!f2py3 --overwrite-signature fib1.f -m fib1 -h fib1.pyf
```

```
Reading fortran codes...
  Reading file 'fib1.f' (format:fix,strict)
Post-processing...
  Block: fib1
    Block: fib
Post-processing (stage 2)...
Saving signatures to file "./fib1.pyf"
```

```
!cat fib1.pyf
```

```
! -- f90 --
! Note: the context of this file is case sensitive.

python module fib1 ! in
  interface ! in :fib1
    subroutine fib(a,n) ! in :fib1:fib1.f
      real*8 dimension(n) :: a
```

```

        integer, optional, check(len(a)>=n), depend(a) :: n=len(a)
    end subroutine fib
end interface
end python module fib1

! This file was auto-generated with f2py (version:1.21.2).
! See http://cens.ioc.ee/projects/f2py2e/

```

El contenido del archivo fib1.pyf es:

```

python module fib1 ! in
    interface ! in :fib1
        subroutine fib(a,n) ! in :fib1:fib1.f
            real*8 dimension(n) :: a
            integer, optional, check(len(a)>=n), depend(a) :: n=len(a)
        end subroutine fib
    end interface
end python module fib1

```

Este código indica que tenemos una subrutina que toma dos argumentos: - a es un array - n es un entero opcional, que tiene que ser mayor que len(a)

```
!f2py3 -c fib1.pyf fib1.f > /dev/null
```

```

In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpqlprsqk7/src.macossx-10.9-x86_64-3.9/fib1module.c:16:
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpqlprsqk7/src.macossx-10.9-x86_64-3.9/fortranobject.h:13:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/arrayobject.h:4:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarrayobject.h:12:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarraytypes.h:1969:
[1m/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/
↳numpy/npy_1_7_deprecated_api.h:17:2: [0m[0;1;35mwarning: [0m[1m"Using_
↳deprecated NumPy API, disable it with "           "#define NPY_NO_DEPRECATED_
↳API NPY_1_7_API_VERSION" [-W#warnings][0m
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^
[0mIn file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpqlprsqk7/src.macossx-10.9-x86_64-3.9/fortranobject.c:2:
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpqlprsqk7/src.macossx-10.9-x86_64-3.9/fortranobject.h:13:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/arrayobject.h:4:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarrayobject.h:12:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarraytypes.h:1969:
[1m/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/
↳numpy/npy_1_7_deprecated_api.h:17:2: [0m[0;1;35mwarning: [0m[1m"Using_
↳deprecated NumPy API, disable it with "           "#define NPY_NO_DEPRECATED_
↳API NPY_1_7_API_VERSION" [-W#warnings][0m
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^
[0m[1m/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmpqlprsqk7/
↳src.macossx-10.9-x86_64-3.9/fib1module.c:109:12: [0m[0;1;35mwarning:_

```

```
→ [0m[1munused function 'f2py_size' [-Wunused-function] [0m
static int f2py_size(PyArrayObject* var, ...)
[0;1;32m
[0m2 warnings generated.
1 warning generated.
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libgfortran.
→dylib) was built for newer macOS version (11.2) than being linked (10.9)
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libquadmath.
→dylib) was built for newer macOS version (11.2) than being linked (10.9)
```

```
ls
```

```
[31mfib1.cpython-39-darwin.so[m[m*      [1m[36mfib3.cpython-39-darwin.so.dSYM[m[m/
fib1.f                      fib3.f
fib1.pyf                    rotacion.f90
[31mfib2.cpython-39-darwin.so[m[m*      rotacion.pyf
[1m[36mfib2.cpython-39-darwin.so.dSYM[m[m/  [31mrotacion_f.cpython-39-darwin.so[m[m*
fib2.pyf                    rotaciones.mod
[31mfib3.cpython-39-darwin.so[m[m*
```

```
import fib1
```

```
print(fib1.fib.__doc__)
```

```
fib(a, [n])
```

Wrapper for fib.

Parameters

a : input rank-1 array('d') with bounds (n)

Other Parameters

n : input int, optional
Default: len(a)

```
a = np.zeros(12)
fib1.fib(a)
print(a)
```

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

```
a = np.zeros(12)
fib1.fib(a,8)
print(a)
```

```
[ 0.  1.  1.  2.  3.  5.  8. 13.  0.  0.  0.  0.]
```

```
a = np.zeros(12)
fib1.fib(a,18)
print(a)
```

```

-----
error                                         Traceback (most recent call last)

Input In [20], in <cell line: 2>()
    1 a = np.zeros(12)
----> 2 fib1.fib(a,18)
    3 print(a)

error: (len(a)>=n) failed for 1st keyword n: fib:n=18

```

Esta es una de las características de F2PY: hace un chequeo básico de los argumentos. Hay otro error que no llega a atrapar: Si le pasamos un array que no es del tipo indicado, falla (sin avisar). Éste claramente no es el comportamiento deseado:

```

a = np.zeros(12, dtype=int)
fib1.fib(a)
print(a)

```

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

Vamos a modificar el archivo de *signature* para enseñarle dos cosas:

- El entero es un argumento de entrada (requerido)
- El *array* a es un archivo de salida **exclusivamente**. Entonces no debemos dárselo. La parte `dimension(n)` y `depend(n)` indica que debe crear un vector de ese tamaño.

```
!cat fib1.pyf
```

```

! -- f90 --
! Note: the context of this file is case sensitive.

python module fib1 ! in
    interface ! in :fib1
        subroutine fib(a,n) ! in :fib1:fib1.f
            real*8 dimension(n) :: a
            integer, optional, check(len(a)>=n), depend(a) :: n=len(a)
        end subroutine fib
    end interface
end python module fib1

! This file was auto-generated with f2py (version:1.21.2).
! See http://cens.ioc.ee/projects/f2py2e/

```

```
!cat fib2.pyf
```

```

! -- f90 --
! Note: the context of this file is case sensitive.

python module fib2
    interface
        subroutine fib(a,n)
            real*8 dimension(n), intent(out), depend(n) :: a
            integer intent(in) :: n
        end subroutine fib
    end interface
end python module fib2

```

```
    end subroutine fib
  end interface
end python module fib2

! This file was auto-generated with f2py (version:1.21.2).
! See http://cens.ioc.ee/projects/f2py2e/
```

```
!f2py3 -c fib2.pyf fib1.f > /dev/null
```

```
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpbx5qc9hg/src.macosx-10.9-x86_64-3.9/fortranobject.c:2:
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpbx5qc9hg/src.macosx-10.9-x86_64-3.9/fortranobject.h:13:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/arrayobject.h:4:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarrayobject.h:12:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarraytypes.h:1969:
[1m/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/
↳numpy/npy_1_7_deprecated_api.h:17:2: [0m[0;1;35mwarning: [0m[1m"Using_
↳deprecated NumPy API, disable it with "           "#define NPY_NO_DEPRECATED_
↳API NPY_1_7_API_VERSION" [-W#warnings][0m
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^
[0mIn file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpbx5qc9hg/src.macosx-10.9-x86_64-3.9/fib2module.c:16:
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
↳tmpbx5qc9hg/src.macosx-10.9-x86_64-3.9/fortranobject.h:13:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/arrayobject.h:4:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarrayobject.h:12:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
↳numpy/core/include/numpy/ndarraytypes.h:1969:
[1m/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/
↳numpy/npy_1_7_deprecated_api.h:17:2: [0m[0;1;35mwarning: [0m[1m"Using_
↳deprecated NumPy API, disable it with "           "#define NPY_NO_DEPRECATED_
↳API NPY_1_7_API_VERSION" [-W#warnings][0m
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^
[0m[1m/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmpbx5qc9hg/
↳src.macosx-10.9-x86_64-3.9/fib2module.c:102:12: [0m[0;1;35mwarning:_
↳[0m[1munused function 'f2py_size' [-Wunused-function][0m
static int f2py_size(PyArrayObject* var, ...)
[0;1;32m
[0m2 warnings generated.
1 warning generated.
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libgfortran.
↳dylib) was built for newer macOS version (11.2) than being linked (10.9)
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libquadmath.
↳dylib) was built for newer macOS version (11.2) than being linked (10.9)
```

```
import fib2
print(fib2.fib.__doc__)
```

```
a = fib(n)
```

Wrapper for fib.

Parameters

n : input int

Returns

a : rank-1 array('d') with bounds (n)

```
fib2.fib(9)
```

```
array([ 0.,  1.,  1.,  2.,  3.,  5.,  8., 13., 21.])
```

```
print(fib2.fib(14))
```

```
[ 0.  1.  1.  2.  3.  5.  8. 13. 21. 34. 55. 89. 144. 233.]
```

La segunda manera de arreglar este problema, en lugar de modificar el archivo de *signature* podría haber sido modificar el código (o hacer una rutina intermedia). Agregando comentarios de la forma Cf2py no influimos en la compilación Fortran pero F2PY los reconoce. En este caso le damos la información sobre la intención de los argumentos en el código:

```
SUBROUTINE FIB(A,N)
C
C      CALCULATE FIRST N FIBONACCI NUMBERS
C
      INTEGER N
      REAL*8 A(N)
Cf2py intent(in) n
Cf2py intent(out) a
Cf2py depend(n) a
      DO I=1,N
        IF (I.EQ.1) THEN
          A(I) = 0.0D0
        ELSEIF (I.EQ.2) THEN
          A(I) = 1.0D0
        ELSE
          A(I) = A(I-1) + A(I-2)
        ENDIF
      ENDDO
    END
```

```
!f2py3 -c fib3.f -m fib3 > /dev/null
```

```
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmpkg02pumo/src.macosx-10.9-x86_64-3.9/fib3module.c:16:
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/
→tmpkg02pumo/src.macosx-10.9-x86_64-3.9/fortranobject.h:13:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
→numpy/core/include/numpy/arrayobject.h:4:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
→numpy/core/include/numpy/ndarrayobject.h:12:
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/
```

```
→numpy/core/include/numpy/ndarraytypes.h:1969:  
[1m/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/  
→numpy/npy_1_7_deprecated_api.h:17:2: [0m[0;1;35mwarning: [0m[1m"Using  
→deprecated NumPy API, disable it with " "#define NPY_NO_DEPRECATED_  
→API NPY_1_7_API_VERSION" [-W#warnings] [0m  
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^  
[0mIn file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/  
→tmpkg02pumo/src.macosx-10.9-x86_64-3.9/fortranobject.c:2:  
In file included from /var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/  
→tmpkg02pumo/src.macosx-10.9-x86_64-3.9/fortranobject.h:13:  
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/  
→numpy/core/include/numpy/arrayobject.h:4:  
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/  
→numpy/core/include/numpy/ndarrayobject.h:12:  
In file included from /Users/flavioc/miniconda3/lib/python3.9/site-packages/  
→numpy/core/include/numpy/ndarraytypes.h:1969:  
[1m/Users/flavioc/miniconda3/lib/python3.9/site-packages/numpy/core/include/  
→numpy/npy_1_7_deprecated_api.h:17:2: [0m[0;1;35mwarning: [0m[1m"Using  
→deprecated NumPy API, disable it with " "#define NPY_NO_DEPRECATED_  
→API NPY_1_7_API_VERSION" [-W#warnings] [0m  
#warning "Using deprecated NumPy API, disable it with " [0;1;32m ^  
[0m[1m/var/folders/18/8wj5kxln1dzbpqvfs_35mz00000gn/T/tmpkg02pumo/  
→src.macosx-10.9-x86_64-3.9/fib3module.c:102:12: [0m[0;1;35mwarning: [0m  
→[0m[1munused function 'f2py_size' [-Wunused-function] [0m  
static int f2py_size(PyArrayObject* var, ...)  
[0;1;32m ^  
[0m2 warnings generated.  
1 warning generated.  
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libgfortran.  
→dylib) was built for newer macOS version (11.2) than being linked (10.9)  
ld: warning: dylib (/usr/local/Cellar/gcc/10.2.0_4/lib/gcc/10/libquadmath.  
→dylib) was built for newer macOS version (11.2) than being linked (10.9)
```

```
import fib3  
print(fib3.fib.__doc__)
```

```
a = fib(n)  
  
Wrapper for fib.  
  
Parameters  
-----  
n : input int  
  
Returns  
-----  
a : rank-1 array('d') with bounds (n)
```

```
print(fib2.fib.__doc__)
```

```
a = fib(n)  
  
Wrapper for fib.
```

Parameters

```
-----
n : input int

Returns
-----
a : rank-1 array('d') with bounds (n)
como vemos, son exactamente iguales.
```

F2PY para código en C

Es posible usar F2PY para código escrito en C, pero en ese caso debemos escribir el *signature file* a mano.

Para código en C es conveniente utilizar **Cython**. Cython es un lenguaje de programación pensado para hacer más fácil escribir extensiones a Python en C. Uno escribe el código en Cython, y luego es traducido a C, con optimizaciones.

Cython también puede utilizarse con Fortran de una manera similar a cómo se usa con C. Para más información ver la [documentación oficial](#)

CAPÍTULO 17

Clase 16: Programación funcional con Python

La programación funcional es un paradigma de programación, de la misma manera que otros paradigmas, como la programación orientada a objetos, o la programación estructurada.

Existen lenguajes de programación que son directamente funcionales, esto es, implementan las reglas de la programación funcional directamente (por ejemplo, Lisp, Haskell, F#, etc.). Desde un punto de vista histórico, la programación funcional tiene su origen en la visión de Alonzo Church del problema de la decisión (*Entscheidungsproblem*), y es complementaria a la más conocida, propuesta por Alan Turing.

Python es un lenguaje orientado a objetos (todo elemento del lenguaje es un objeto), de modo tal que no es posible hablar de un paradigma funcional en Python, sino mas bien de un *estilo* de programación funcional.

Un trabajo interesante es el siguiente: 'Why Functional Programming Matters: <http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf>'.

17.1 Los errores al programar

En el continuo devenir de la programación, uno se encuentra, principalmente, resolviendo errores. Un resumen de los errores posibles en un código se pueden encontrar en la expresión

```
i = i+1
```

En esta expresión podemos encontrar tres tipos de errores: - *Error de lectura* : el valor de *i* en el lado derecho no es el que efectivamente uno desearía, es decir, el código está leyendo un valor incorrecto. - *Error de escritura* : el valor de *i* en el lado izquierdo no es el que efectivamente uno desearía, es decir, estamos guardando la expresión en una variable incorrecta. - *Error de cálculo* : que se produce, por ejemplo, porque no queremos sumar 1 sino 2, o queremos restar el valor de *i*.

Existe un cuarto tipo de error que aparece y tiene que ver con un *error de flujo*, en el cual el código se ejecuta en una rama que no es la deseada, debido a que una condición lógica no se cumple tal como se esperaba. O por ejemplo, el orden en que se ejecutan las sentencias no es el adecuado:

```
# Función que calcula (x+1) (x+2)
def f(x):
```

(continué en la próxima página)

(proviene de la página anterior)

```
x = x+1  
y = x+1  
return x*y
```

```
# Función que calcula (x+1) (x+2) ?? Mmmmm.....  
def g(x):  
    y = x+1  
    x = x+1  
    return x*y
```

```
print(f(3))  
print(g(3))
```

```
20  
16
```

17.2 Los errores en notebooks

Además de las complejidades propias de la programación, que están asociadas al *dominio* donde se encuentra el problema que uno quiere resolver, y a las dificultades que eso implica; los *notebooks* introducen también una dificultad adicional: uno puede redefinir los datos en celdas posteriores, pero puede volver ‘atrás’ en el código y recalcular otra celda. Veamos un [ejemplo](#):

```
data = [1, 2, 3, 4]
```

```
def prom(a):  
    s = sum(a)  
    n = len(a)  
    return s/n
```

```
prom(data)
```

```
2.5
```

```
data = "Some data"  
print(len(data))
```

```
9
```

17.3 Mutabilidad

Los problemas que vemos arriba se deben a la *mutabilidad*: las *variables* pueden cambiar (esto es, ser reescritas) a lo largo del código. Ahora bien, pareciera que la mutabilidad es intrínseca a la computación, al fin y al cabo, en el hardware hay una cantidad limitada de memoria y de registros que son continuamente reescritos para que nuestro código corra. Sin embargo, los lenguajes de programación de alto nivel que usamos nos alejan (afortunadamente) del requerimiento de mantener el estado de la memoria y los registros explícitamente en el código (y en el algoritmo en nuestra cabeza).



Figura 1: Más código

La pregunta que cabe entonces es ¿cómo hacer un código que prevenga la mutabilidad, pero que a la vez me permita transformar los datos para resolver mi problema? La respuesta viene de la mano de un ente muy conocido en matemáticas: *las funciones*

17.4 Funciones

Una función desde el punto de vista matemático es una relación que a cada elemento de un conjunto le asocia exactamente un elemento de otro conjunto. Estos conjuntos pueden ser números, vectores, matrices en el mundo matemático,

$$y = f(x)$$

o, en un mundo más físico, peras, manzanas, nombres, apellidos, objetos varios:

Figura 2: una función

Estas funciones tienen dos características fundamentales para usar en programación: - Permiten “transformar” un valor en otro - El valor original **no** se modifica

Es decir que el uso de funciones, al estilo matemático, en un código resuelven el problema de la mutabilidad, pero a la vez me permiten “transformar”, es decir, crear nuevos valores a partir del valor original.

17.4.1 Funciones puras

El análogo computacional de las funciones matemáticas se llaman *funciones puras*. Una función se dice pura cuando:

- Siempre retorna el mismo valor de salida para el mismo valor de entrada
- No tiene efectos colaterales (*side effects*)

17.4.2 Funciones de primer orden o primera clase

Un lenguaje se dice que tiene funciones de primera clase cuando son tratadas exactamente igual que otros valores o variables.

17.4.3 Funciones de orden superior

Un lenguaje que permite pasar funciones como argumentos se dice que acepta funciones de orden superior.

```
def square(x):
    return x*x
```

```
def next(x):
    return x+1
```

```
a = 4
b = next(a)
c = next(next(a))
```

```
print(a)
print(b)
print(c)
```

```
4
5
6
```

```
def h(x):
    return (next(x)) * (next(next(x)))
```

print(h(3))

20

Si se tiene funciones puras, es posible componerlas

```
def compose(f, g):
    return lambda x: f(g(x))

next2 = compose(next, next)

print(next2(a))
```

6

17.5 Inmutabilidad

Usando funciones puras se garantiza la inmutabilidad de los valores hacia adentro de la función. Pero, ¿qué sucede afuera? Python, al no ser un lenguaje funcional *per se*, no tiene la capacidad de establecer la inmutabilidad de cualquier valor, excepto para los casos de strings y tuplas, además, obviamente, de las expresiones literales.

Queda entonces en el programador la responsabilidad de no mutar los datos...

... o usar anotaciones de tipos

```
def cube(x: int) -> int:
    return x*x*x
```

print(cube(2))

8

Nótese que Python NO chequea los tipos de datos, no tiene manera en forma nativa de hacerlo. Por eso puedo ejecutar la función `cube` con floats, por ejemplo:

```
print(cube(3.0))
```

27.0

Para poder utilizar la anotación de tipos en forma efectiva, se puede recurrir a `mypy` <<http://mypy-lang.org/index.html>>`. Esta es una aplicación que me permite comprobar tipos de datos anotados en Python. Para instalar `mypy` usamos:

```
conda install mypy
```

Clases de Python

```
cd mypy_example
```

```
/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/curso-
→python/mypy_example
```

```
!cat cube.py
```

```
def cube(x: int) -> int:
    return x*x*x

def Hola(name: str) -> str:
    return 'Hola ' + name

def Hola2(name):
    return 'Hola ' + name

if __name__ == "__main__":
    a = cube(2)
    print(a)

    b = cube(3.0)    # Esto no da error en Python, mypy si lo captura
    print(b)

    print(Hola('Juan'))
    print(Hola2(3)) # Esto da un error de concatenación

    print(Hola(3)) # Esto da un error de concatenación, y además mypy lo captura
```

```
!python3 cube.py
```

```
8
27.0
Hola Juan
Traceback (most recent call last):
  File "/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/
→curso-python/mypy_example/cube.py", line 24, in <module>
    print(Hola2(3)) # Esto da un error de concatenación
  File "/Users/flavioc/Library/Mobile Documents/com~apple~CloudDocs/Documents/cursos/
→curso-python/mypy_example/cube.py", line 8, in Hola2
    return 'Hola ' + name
TypeError: can only concatenate str (not "int") to str
```

```
!mypy cube.py
```

```
cube.py:19: [1m[31merror:[m Argument 1 to [m[1m"cube"[m has incompatible type [m[1m
→"float"[m; expected [m[1m"int"[m[m
cube.py:26: [1m[31merror:[m Argument 1 to [m[1m"Hola"[m has incompatible type [m[1m
→"int"[m; expected [m[1m"str"[m[m
[1m[31mFound 2 errors in 1 file (checked 1 source file)[m
```

Es posible que uno quiera usar `mypy` sobre un archivo de notebook `ipynb`. Para eso hay que instalar la aplicación nbQA más detalles acá.

17.6 No más loops

Si las funciones deben ser puras, y las ‘variables’ dejan de ser variables y pasan a ser valores, entonces no puede haber loops en mi código. Un loop necesita invariablemente un contador (`i = i+1`) que necesariamente es una variable mutable. Así que así nomás, de un plumazo no existen más loops.

¿Entonces? Entonces, todos los loops se reemplazan por llamados a funciones recursivas, o se utilizan funciones de orden superior:

```
# Filter

l = [1, 2, 3, 4, 5, 6]

def es_par(x):
    return (x%2 == 0)

pares = list(filter(es_par, l))
print(pares)
```

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

```
# Filter usando list comprehension
list(x for x in l if es_par(x))
```

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

```
# Map
siguientes = list(map(next, l))
print(siguientes)
```

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

El módulo `functools` provee la función `reduce`, que complementa a `map` y `filter`.

```
# Reduce
from functools import *
import operator

# Suma usando el predicado desde el módulo `operator`
suma = reduce(operator.add, l, 0)
print(suma)
```

```
21
```

```
help(reduce)
```

```
Help on built-in function reduce in module _functools:

reduce(...)
```

(continué en la próxima página)

(proviene de la página anterior)

```
reduce(function, sequence[, initial]) -> value
```

Apply a function of two arguments cumulatively to the items of a sequence, **from left** to right, so **as** to reduce the sequence to a single value.

For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates $((1+2)+3)+4)+5$. If initial **is** present, it **is** placed before the items of the sequence **in** the calculation, **and** serves **as** a default when the sequence **is** empty.

```
# Suma usando el predicado como lambda
otra_suma = reduce(lambda x,y: x+y, 1)
print(otra_suma)
```

21

```
# Suma definiendo la propia función suma
def add(x,y):
    return x+y

y_otra_suma = reduce(add,1)
print(y_otra_suma)
```

21

La suma de los cuadrados de una lista:

```
suma_cuadrados = reduce(lambda x,y: x+y, map(square,l))
print(suma_cuadrados)
```

91

CAPÍTULO 18

Ejercicios

18.1 Ejercicios de Clase 01

1. Abra una terminal (consola) o notebook y utilícela como una calculadora para realizar las siguientes acciones:
 - Suponiendo que, de las cuatro horas de clases, tomamos dos descansos de 15 minutos cada uno 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 en este curso?
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, muestre la fecha y hora
 - Borre el subdirectorio `tmp`
3. Para cubos de lados de longitud $L = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
4. Para esferas de radios $r = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
5. Fíjese si alguno de los valores de $x = 2,05, x = 2,11, x = 2,21$ es un cero de la función $f(x) = x^2 + x/4 - 1/2$.
6. 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 $z=1+0.5i$ es $1-0.5j$ ”
 - “El conjugado de $z=(1+0.5i)^2$ es ...” (con el valor correspondiente)

18.2 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 ENTREGAR. 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 02_SuApellido.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
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, ..., 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 $L1$ tenga el valor del último elemento de L y repita el inciso anterior.

6. Utilizando funciones y métodos de *strings* en la cadena de caracteres:`s1='En un lugar de la Mancha de cuyo nombre no quiero acordarme'`

- 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 de la oración.
 - 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:

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

18.2.1 Adicionales

9. Calcular la suma:

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

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

10. 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.

18.3 Ejercicios de Clase 03

1. De los primeros 100 números naturales imprimir aquellos que no son divisibles por alguno de 2, 3, 5 o 7.
 2. Usando estructuras de control, calcule la suma:

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

1. Incluyendo todos los valores de k
 2. Incluyendo únicamente los valores pares de k .
 3. 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. 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 |
=====
```

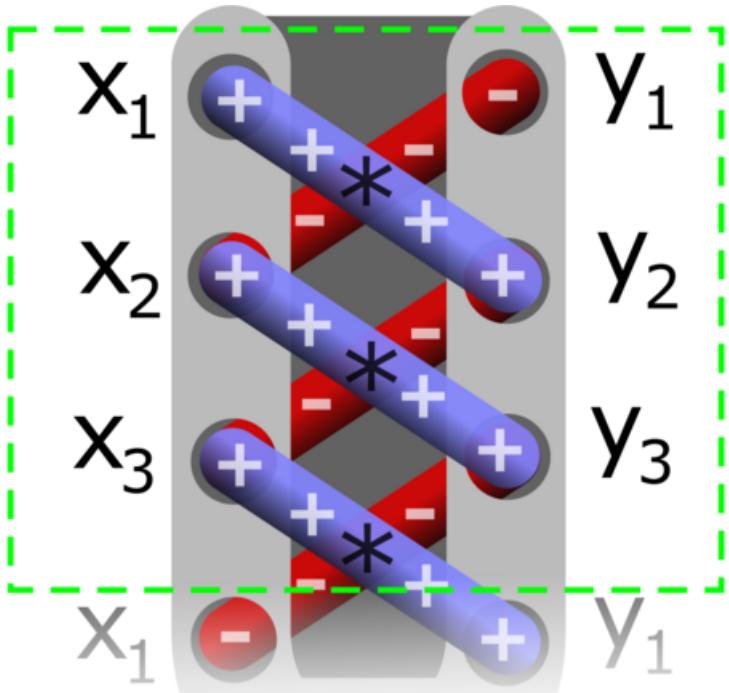
(continué en la próxima página)

(proviene de la página anterior)

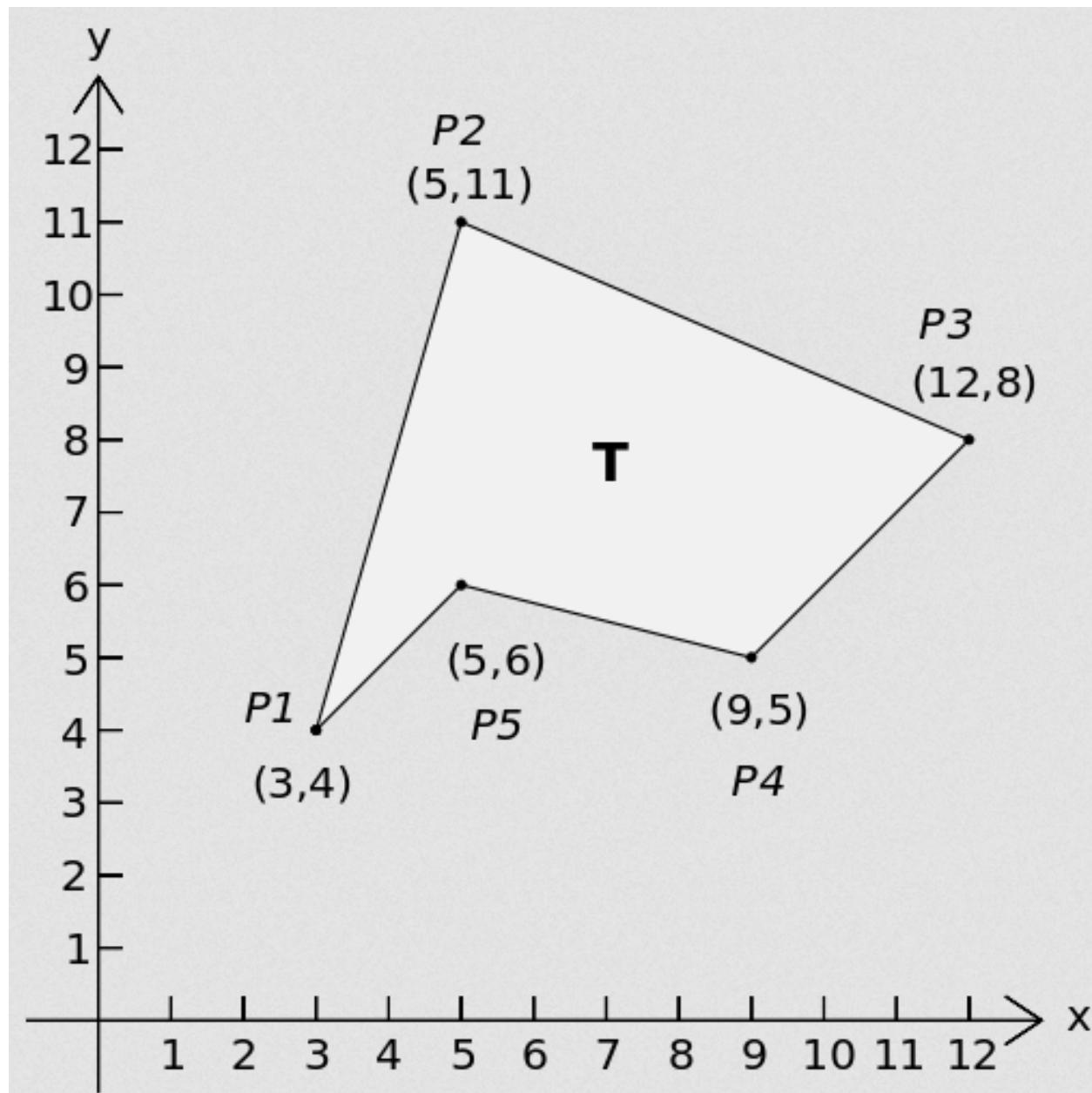
```
/160 | 0.342 | -0.940 | -0.839 |
/170 | 0.174 | -0.985 | -0.916 |
=====
"""

```

5. Un método para calcular el área de un polígono (no necesariamente regular) que se conoce como fórmula del área de Gauss o fórmula de la Lazada (*shoelace formula*) consiste en describir al polígono por sus puntos en un sistema de coordenadas. Cada punto se describe como un par (x, y) y la fórmula del área está dada mediante la suma de la multiplicación de los valores en una diagonal a los que se le resta los valores en la otra diagonal, como muestra la figura



$$2A = (x_1y_2 + x_2y_3 + x_3y_4 + \dots) - (x_2y_1 + x_3y_2 + x_4y_3 + \dots)$$



- Utilizando una descripción adecuada del polígono, implementar la fórmula de Gauss para calcular su área y aplicarla al ejemplo de la figura.
 - Verificar que el resultado no depende del punto de inicio.
6. 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

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

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:

$$\begin{aligned} A(x_1, \dots, x_n) &= \bar{x} = \frac{x_1 + \dots + x_n}{n} \\ G(x_1, \dots, x_n) &= \sqrt[n]{x_1 \cdots x_n} \\ H(x_1, \dots, x_n) &= \frac{n}{\frac{1}{x_1} + \dots + \frac{1}{x_n}} \end{aligned}$$

- 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 la siguiente lista 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]
```

8. 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]
```

18.4 Ejercicios de Clase 04

1. 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
```

2. 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
```

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

4. 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.

5. **PARA ENTREGAR.** Describimos una grilla de **sudoku** como un string de nueve líneas, cada una con 9 números, con números entre 1 y 9. Escribir un conjunto de funciones que permitan chequear si una grilla de sudoku es correcta. Para que una grilla sea correcta deben cumplirse las siguientes condiciones

- Los números están entre 1 y 9
- En cada fila no deben repetirse
- En cada columna no deben repetirse
- En todas las regiones de 3x3 que no se solapan, empezando de cualquier esquina, no deben repetirse

1. Escribir una función que convierta un string con formato a una lista bidimensional. El string estará dado con nueve números por línea, de la siguiente manera (los espacios en blanco en cada línea pueden variar):

```
sudoku = """145327698
839654127
672918543
496185372
218473956
753296481
367542819
984761235
521839764"""
```

2. Escribir una función `check_repetidos()` que tome por argumento una lista (unidimensional) y devuelva verdadero si la lista tiene elementos repetidos y falso en caso contrario (puede ser conveniente explorar el uso de `set`).
3. Escribir la función `check_sudoku()` que toma como argumento una grilla (como una lista bidimensional de 9x9) y devuelva verdadero si los números corresponden a la resolución correcta del Sudoku y falso en caso contrario. Note que debe verificar que los números no se repiten en filas, ni en columnas ni en recuadros de 3x3. Para obtener la posición de los recuadros, puede investigar que hacen las líneas de código:

```
j, k = (i // 3) * 3, (i % 3) * 3
r = [grid[a][b] for a in range(j, j+3) for b in range(k, k+3)]
```

suponiendo que `grid` es el nombre de nuestra lista bidimensional, cuando `i` toma valores entre 0 y 8.

18.5 Ejercicios de Clase 05

- 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), ...]
```

- 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

18.6 Ejercicios de Clase 06

1. Implemente los métodos `__add__`, `producto` y `abs`

- `__add__()` 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 con el valor del producto interno
- `abs` toma como argumentos el propio objeto y retorna el número real correspondiente

Su uso será el siguiente:

```
v1 = Vector(1,2,3)  
v2 = Vector(3,2,1)  
v = v1 + 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):  
        return f"Punto en el espacio con coordenadas: x = {self.x}, y = {self.y}, z =  
↪{self.z}"
```

(continúe en la próxima página)

(provine de la página anterior)

```

def __repr__(self):
    return f"Punto(x = {self.x}, y = {self.y}, z = {self.z})"

def __call__(self):
    return self.__str__()

@classmethod
def total(cls):
    "Imprime el número total de puntos"
    print(f"En total hay {cls.num_puntos} puntos definidos")

```

Complete la implementación de la clase Vector con los métodos pedidos

```

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 __mul__(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
```

- Un método `from_string` (**si puede**) 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

```

- Escriba los métodos llamados `__mul__(self, value)` y `__rmul__(self, value)`, que devuelvan el producto de un polinomio por un valor constante, llamando al método `mul` definido anteriormente. Eso permitirá usar la operación producto en la forma:

```

>>> p1 = Polinomio([1,2,3,4])
>>> k = 3.5
>>> p1 * k
>>> k * p1

```

18.7 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))
```

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.
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:

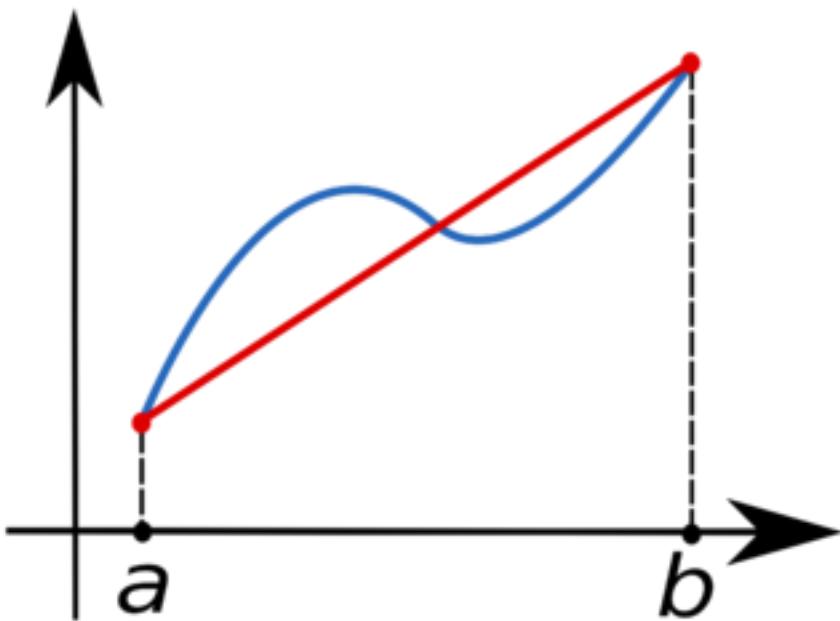
- `-v vel` o, equivalentemente `--velocidad=vel`, donde `vel` es el número dando la velocidad inicial en m/s. El valor por defecto será 0.
- `-a alt` o, equivalentemente `--altura=alt`, donde `alt` es un número dando la altura inicial en metros. El valor por defecto será 1000. La altura inicial debe 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` o, 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` o, 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 **08_Suapellido.py** en un adjunto por correo electrónico, con asunto: **08_Suapellido**, antes del día miércoles 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:

$$\text{Li}(t) = \int_2^t \frac{1}{\ln x} dx$$

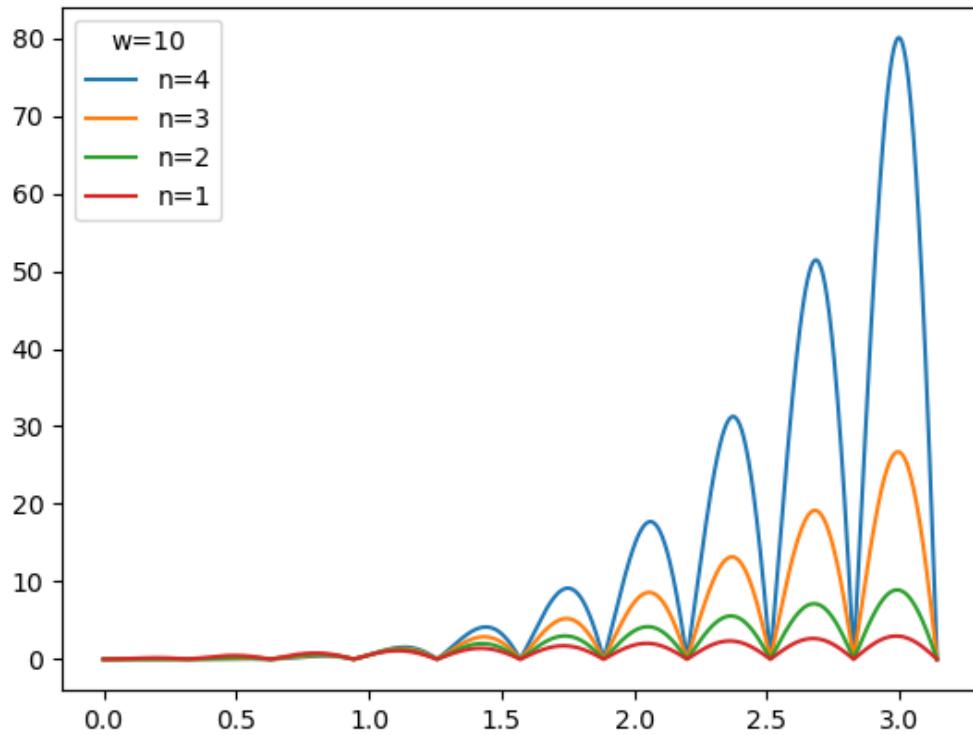
usando la función 'trapzf' para valores de `npts=10, 20, 30, 40, 50, 60`

18.8 Ejercicios de Clase 09

1. Realizar un programa para visualizar la función

$$f(x, n, w) = x^n |\sin(wx)|$$

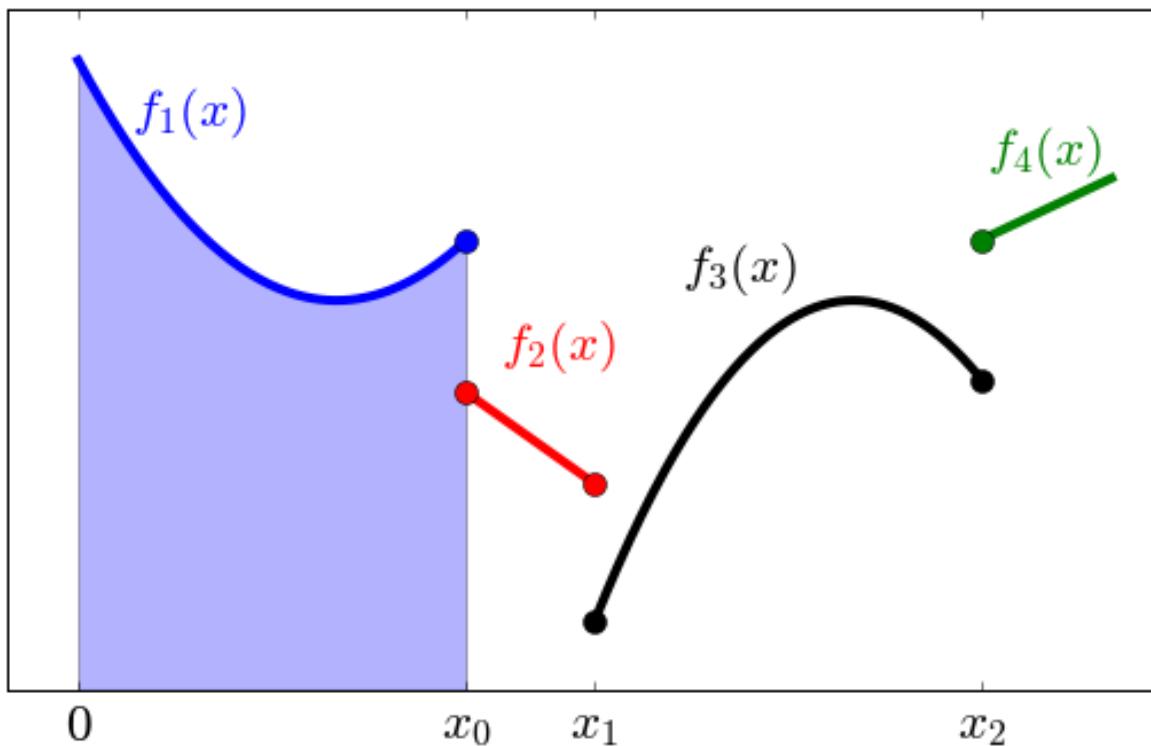
El programa debe realizar el gráfico para $w = 10$, con cuatro curvas para $n = 1, 2, 3, 4$, similar al que se muestra en la siguiente figura



2. Para la función definida a trozos:

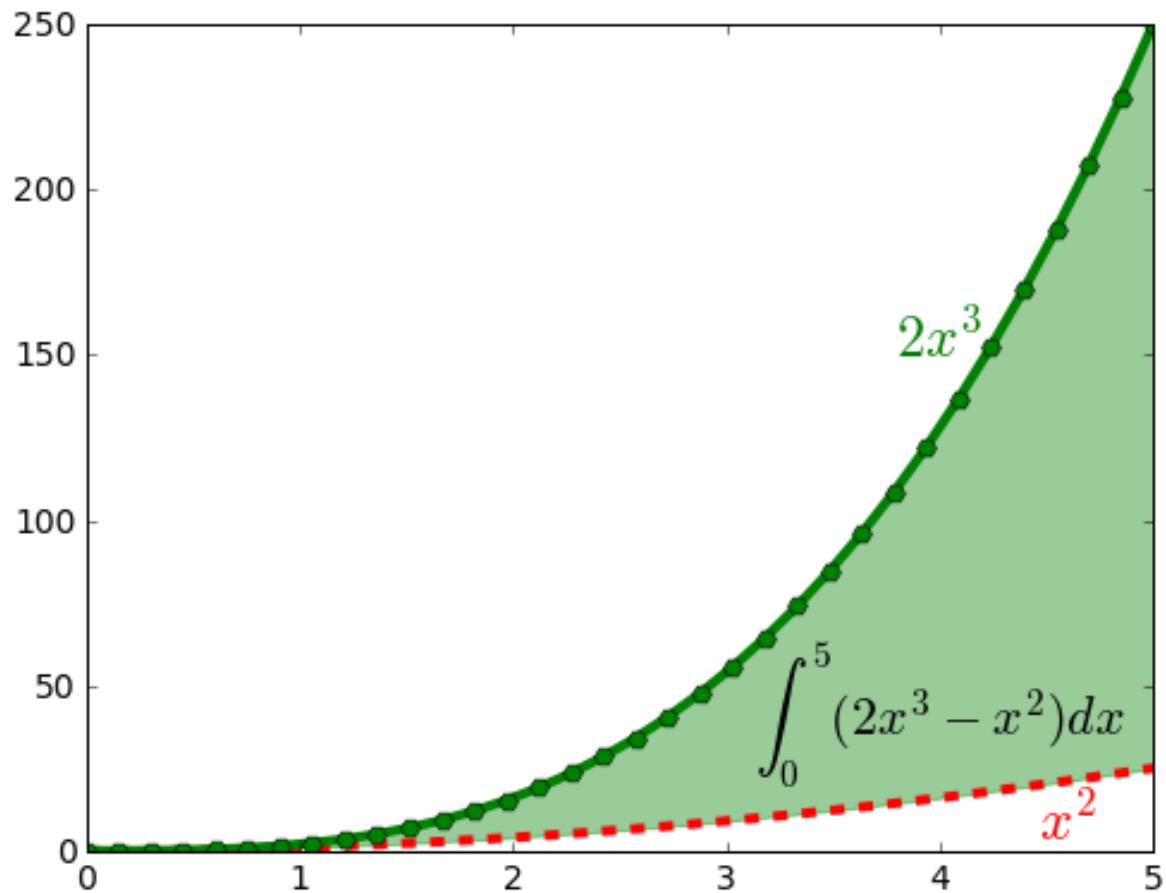
$$f(x) = \begin{cases} f_1(x) = x^2/8 & -\pi < x \leq \pi/2 \\ f_2(x) = -0,3x & \pi/2 < x < \pi \\ f_3(x) = -(x - 2\pi)^2/6 & \pi \leq x \leq 5\pi/2 \\ f_4(x) = (x - 2\pi)/5 & 5\pi/2 < x \leq 3\pi \end{cases}$$

realizar la siguiente figura de la manera más fiel posible.

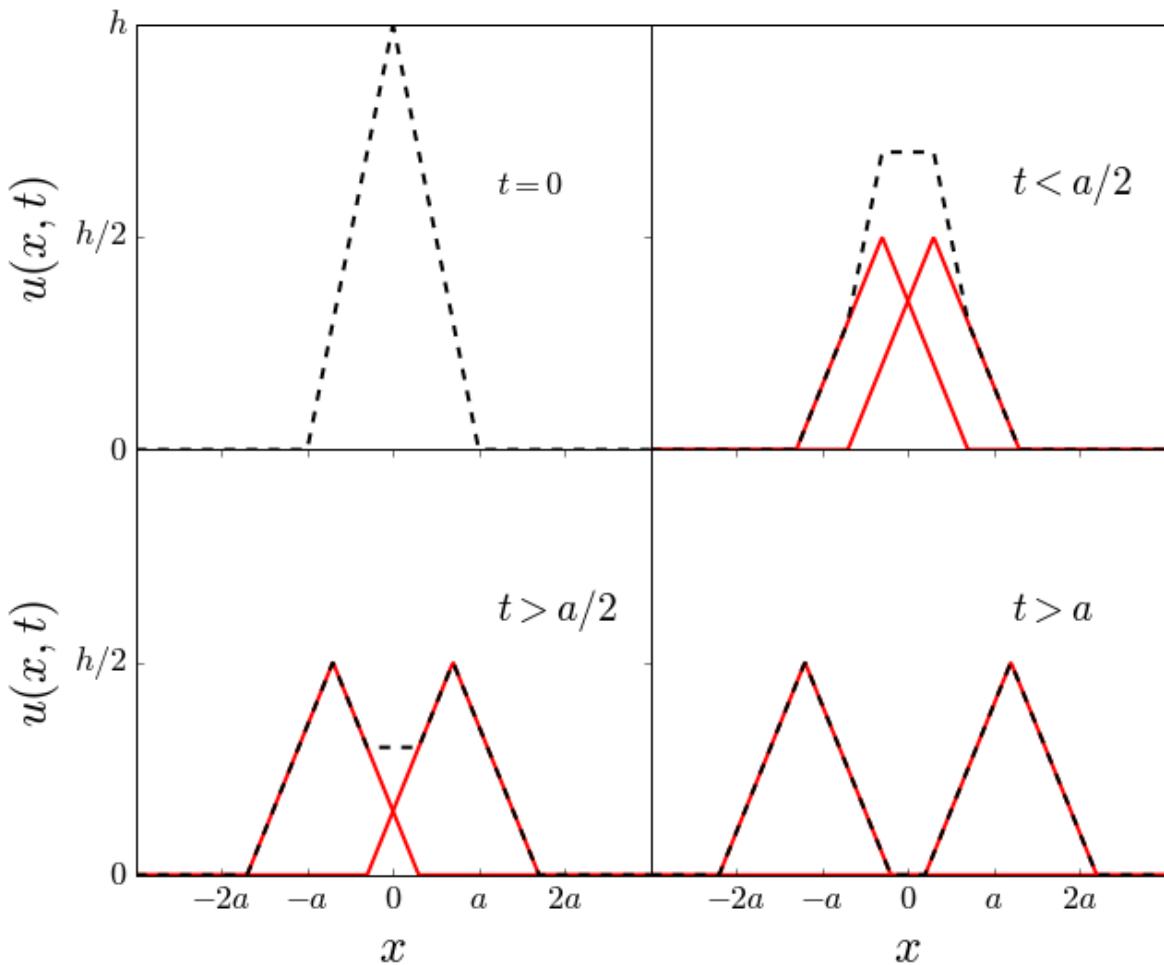


Pistas: Buscar información sobre `plt.fill_between()` y sobre `plt.xticks` y `plt.yticks`.

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).

18.9 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.

- 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:

- 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:

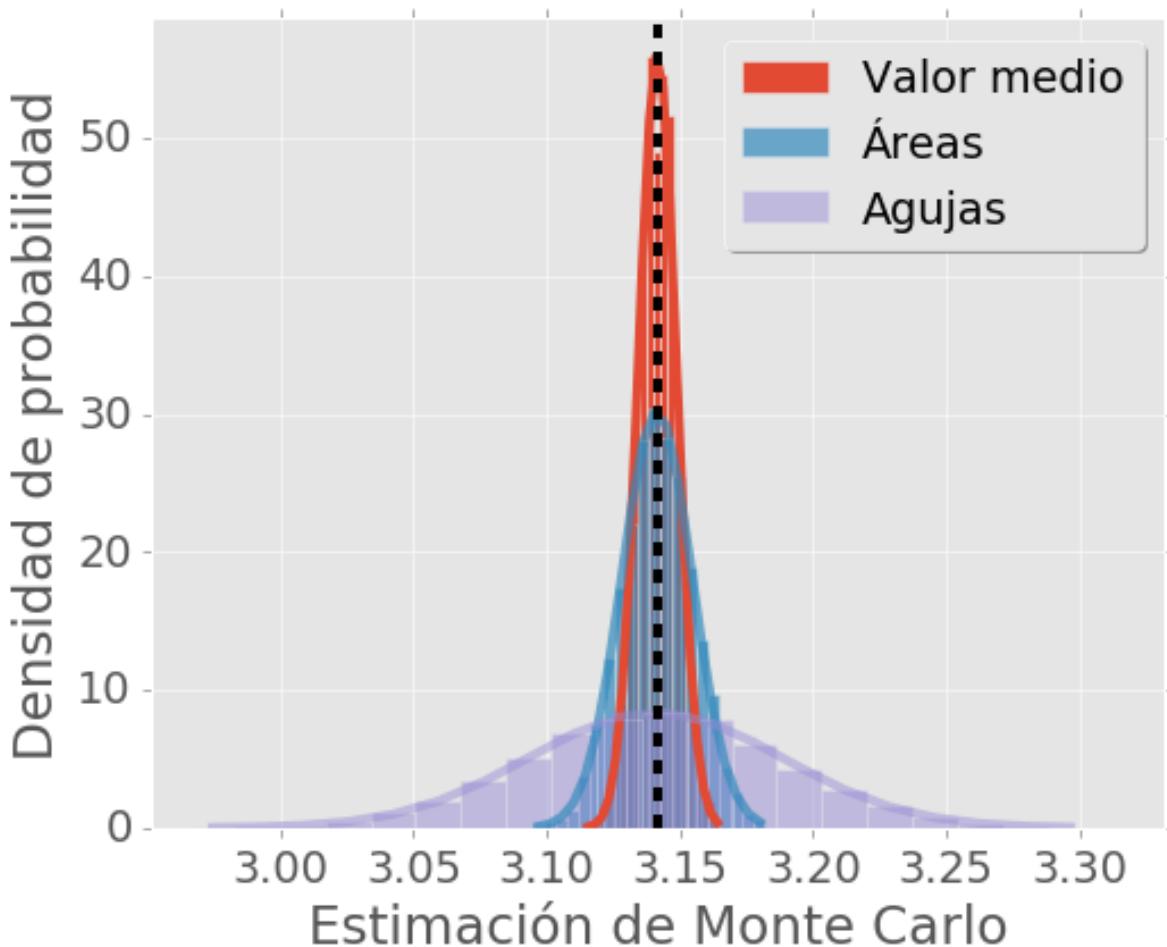
$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) dx$$

Tomando la función particular $f(x) = \sqrt{1-x^2}$ entre $x = 0$ y $x = 1$, obtenemos:

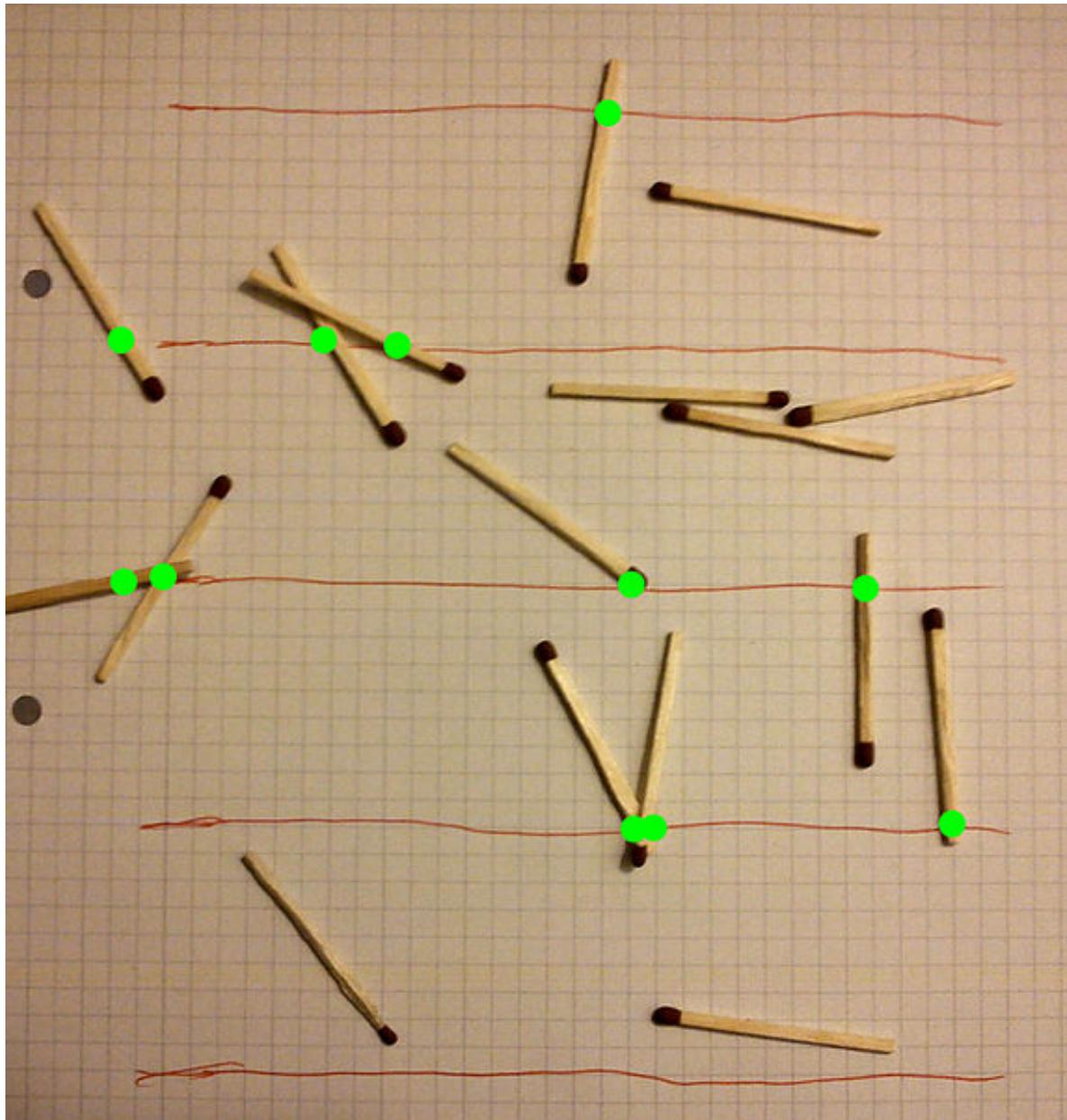
$$\langle f \rangle = \int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$

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

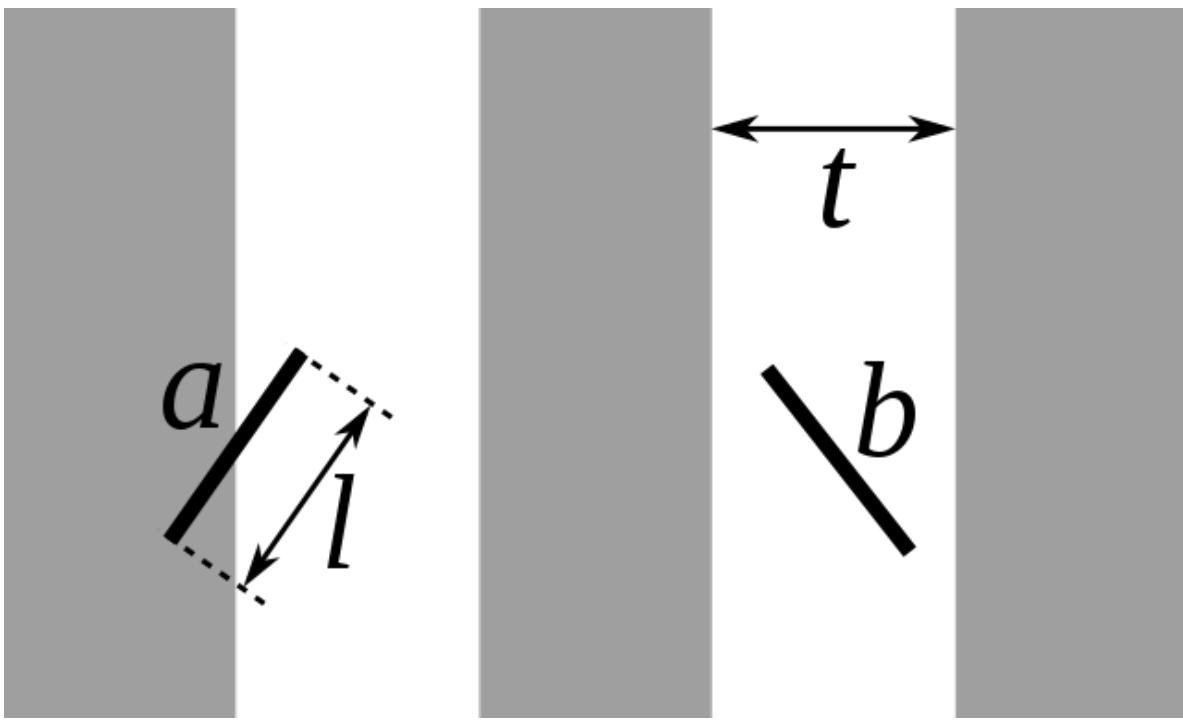
- Generar puntos aleatoriamente en el intervalo $[0, 1]$
 - Calcular el valor medio de $f(x)$ para los puntos aleatorios x .
 - El resultado va a ser igual al valor de la integral, y por lo tanto a $\pi/4$.
3. Utilizar las funciones anteriores con diferentes valores para el número total de puntos N . 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.

18.10 Ejercicios de Clase 11

1. Graficar para valores de $k = 1, 2, 5, 10$ y como función del límite superior L , el valor de la integral:

$$I(k, L) = \int_0^L x^k e^{-kx/2} \sin(kx) dx$$

con rango de variación de L entre 0 y 2π .

2. 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.
3. 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').
4. 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.
5. 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:

6. la distribución de ocurrencias del primer parámetro.
7. la distribución de ocurrencias de energías de la tercera partícula.
8. la distribución de ocurrencias de ángulos de la cuarta partícula, medido respecto al tercer eje.
9. 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`.

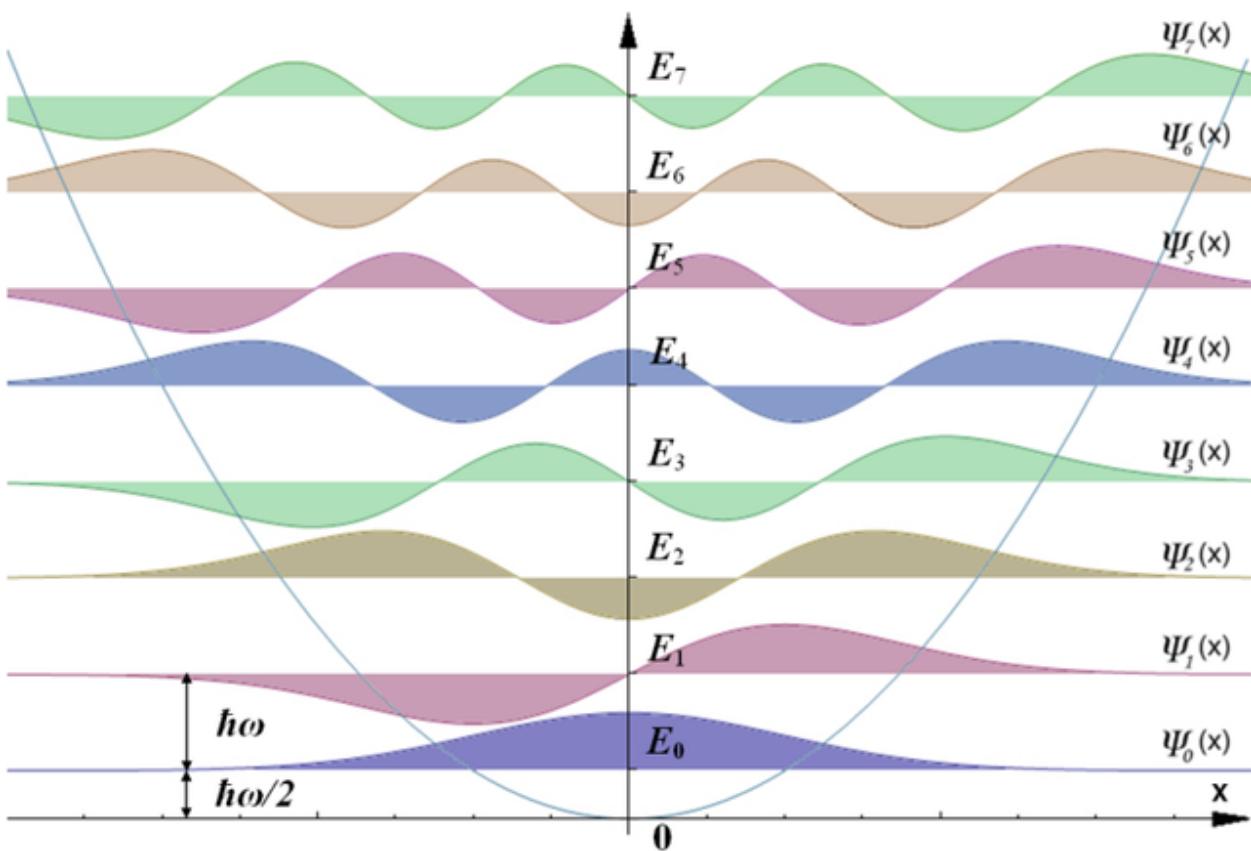
7. Usando **Scipy** y **Matplotlib** grafique las funciones de onda del oscilador armónico unidimensional para las cuatro energías más bajas ($n = 1, 2, 3, 4$), en el intervalo $[-5, 5]$. Asegúrese de que están correctamente normalizados.

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)



18.11 Ejercicios de Clase 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.

18.12 Ejercicios de Clase 14

1. Utilizando **Matplotlib**:

- Hacer un gráfico donde dibuje una parábola $y = x^2$ en el rango $[-5, 5]$.
- En el mismo gráfico, agregar un círculo en $x = -5$.
- El círculo debe moverse siguiendo la curva, como se muestra en la figura

2. **PARA ENTREGAR. Caída libre 2:** Modificar el ejercicio de la clase 8 de caída libre que entregó, para aceptar dos nuevas opciones:

- La opción `--vx` permite dar una velocidad inicial en la dirección horizontal.
- La opción `--animate`, tal que cuando se utilice, el programa muestre una animación de la trayectoria.
- La animación tiene que tener un cartel indicando el tiempo, y la velocidad y altura correspondiente a ese tiempo.
- Agregue una “cola fantasma” a la partícula, que muestre posiciones anteriores.

Envíe el programa llamado **14_Suapellido.py** en un adjunto por correo electrónico, con asunto: **14_Suapellido**

3. 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.
4. 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 19

Material extra

19.1 Programa Detallado

Autor Juan Fiol

Version Revision: 2022

Copyright Libre

19.1.1 Clase 1: Introducción al lenguaje

- Cómo empezar: Instalación y uso
 - Instalación
 - Documentación y ayudas
 - Uso de Python: Interactivo o no
 - Notebooks de Jupyter
- Comandos de Ipython
 - Comandos de Navegación
 - Algunos de los comandos mágicos
 - Comandos de Shell
- Conceptos básicos de Python
 - Características generales del lenguaje
 - Tipos de variables

19.1.2 Clase 2: Tipos de datos y control

- Tipos simples: Números
- Tipos compuestos
- Strings: Secuencias de caracteres
 - Operaciones
 - Iteración y Métodos de Strings
 - Formato de strings
- Conversión de tipos
- Tipos contenedores: Listas
 - Operaciones sobre listas
 - Tuplas
 - Rangos
 - Comprensión de Listas
- Módulos
 - Módulo math
 - Módulo cmath
 - Adicionales

19.1.3 Clase 3: Tipos complejos y control de flujo

- Diccionarios
 - Creación
 - Selección de elementos
 - Acceso a claves y valores
 - Modificación o adición de campos
- Conjuntos
 - Operaciones entre conjuntos
 - Modificar conjuntos
- Control de flujo
 - if/elif/else
 - Iteraciones
- Técnicas de iteración
 - Iteración sobre conjuntos (*set*)
 - Iteración sobre elementos de dos listas
 - Iteraciones sobre diccionarios

19.1.4 Clase 4: Funciones

- Las funciones son objetos
- Definición básica de funciones
- Argumentos de las funciones
 - Ámbito de las variables en los argumentos
 - Funciones con argumentos opcionales
 - Tipos mutables en argumentos opcionales
 - Número variable de argumentos y argumentos *keywords*
- Empacar y desempacar secuencias o diccionarios
- Funciones que devuelven funciones
- Funciones que toman como argumento una función
- Aplicacion 1: Ordenamiento de listas
- Funciones anónimas
- Ejemplo 1: Integración numérica
 - Uso de funciones anónimas
- Ejemplo 2: Polinomio interpolador

19.1.5 Clase 5: Entrada y salida, decoradores, y errores

- Funciones que aceptan y devuelven funciones (Decoradores)
 - Notación para decoradores
 - Algunos usos de decoradores
- Atrapar y administrar errores
 - Administración de excepciones
 - “Crear” excepciones
- Escritura y lectura a archivos
 - Ejemplos
- Archivos comprimidos

19.1.6 Clase 6: Programación Orientada a Objetos

- Breve introducción a Programación Orientada a Objetos
- Clases y Objetos
 - Métodos especiales
- Herencia
- Atributos de clases y de instancias
- Algunos métodos “especiales”

- Método `__del__()`
- Métodos `__str__` y `__repr__`
- Método `__call__`
- Métodos `__add__`, `__mul__`

19.1.7 Clase 7: Control de versiones y biblioteca standard

- ¿Qué es y para qué sirve el control de versiones?
 - Cambios en paralelo
 - Historia completa
- Instalación y uso: Una versión breve
 - Instalación
 - Interfaces gráficas
 - Documentación
 - Configuración básica
 - Creación de un nuevo repositorio
 - Clonación de un repositorio existente
 - Ver el estado actual
 - Creación de nuevos archivos y modificación de existentes
 - Actualización de un repositorio remoto
 - Puntos importantes
- Algunos módulos (biblioteca standard)
 - Módulo `sys`
 - Módulo `os`
 - Módulo `subprocess`
 - Módulo `glob`
 - Módulo `pathlib`
 - Módulo `Argparse`
 - Módulo `re`

19.1.8 Clase 8: Introducción a Numpy

- Algunos ejemplos
 - Graficación de datos de archivos
 - Comparación de listas y *arrays*
 - Generación de datos equiespaciados
- Características de *arrays* en **Numpy**
 - Uso de memoria de listas y arrays

- Velocidad de **Numpy**
- Creación de *arrays* en **Numpy**
 - Creación de *Arrays* unidimensionales
 - Arrays multidimensionales
 - Otras formas de creación
- Acceso a los elementos
- Propiedades de **Numpy** arrays
 - Propiedades básicas
 - Otras propiedades y métodos de los *array*
- Operaciones sobre arrays
 - Operaciones básicas
 - Comparaciones
 - Funciones definidas en **Numpy**
 - Lectura y escritura de datos a archivos

19.1.9 Clase 9: Visualización

- Interactividad
 - Trabajo con ventanas emergentes
 - Trabajo sobre notebooks
- Gráficos simples
- Formato de las curvas
 - Líneas, símbolos y colores
 - Nombres de ejes y leyendas
- Escalas y límites de graficación (vista)
- Exportar las figuras
- Dos gráficos en la misma figura
- Personalizando el modo de visualización
 - Archivo de configuración
 - Hojas de estilo
 - Modificación de parámetros dentro de programas

19.1.10 Clase 10: Más información sobre Numpy

- Creación y operación sobre **Numpy** arrays
 - Funciones para crear arrays
 - Funciones que actúan sobre arrays
 - Productos entre arrays y productos vectoriales
 - Comparaciones entre arrays
- Atributos de *arrays*
 - reshape
 - transpose
 - min, max
 - argmin, argmax
 - sum, prod, mean, std
 - cumsum, cumprod, trapz
 - nonzero
- Conveniencias con arrays
 - Convertir un array a unidimensional (ravel)
 - Enumerate para `ndarrays`
 - Vectorización de funciones escalares
- Copias de arrays y vistas
- Indexado avanzado
 - Indexado con secuencias de índices
 - Índices de arrays multidimensionales
 - Indexado con condiciones
 - Función where
- Extensión de las dimensiones (*Broadcasting*)
- Unir (o concatenar) *arrays*
 - Apilamiento vertical
 - Apilamiento horizontal
- Generación de números aleatorios
 - Distribución uniforme
 - Distribución normal (Gaussiana)
 - Histogramas
 - Distribución binomial

19.1.11 Clase 11: Introducción al paquete Scipy

- Una mirada rápida a Scipy
- Funciones especiales
 - Funciones de Bessel
 - Función Error
 - Evaluación de polinomios ortogonales
 - Factorial, permutaciones y combinaciones
- Integración numérica
 - Ejemplo de función fuertemente oscilatoria
 - Funciones de más de una variable
- Álgebra lineal
 - Productos y normas
 - Aplicación a la resolución de sistemas de ecuaciones
 - Descomposición de matrices
 - Autovalores y autovectores
 - Rutinas de resolución de ecuaciones lineales
- Entrada y salida de datos
 - Entrada/salida con *Numpy*
 - Ejemplo de análisis de palabras
 - Entrada y salida en Scipy

19.1.12 Clase 12: Un poco de graficación 3D

- Gráficos y procesamiento sencillo en 2D
 - Histogramas en 2D
 - Gráficos de contornos
 - Superficies y contornos

19.1.13 Clase 13: Interpolación y ajuste de curvas (fiteo)

- Interpolación
 - Interpolación con polinomios
 - Splines
 - B-Splines
 - Lines are guides to the eyes
 - Cantidades derivadas de *splines*
- Interpolación en dos dimensiones

- Interpolación sobre datos no estructurados
- Fiteos de datos
 - Ajuste con polinomios
- Fiteos con funciones arbitrarias
 - Ejemplo: Fiteo de picos

19.1.14 Clase 14: Animaciones e interactividad

- Animaciones con **Matplotlib**
 - Una animación simple en pocos pasos
 - Segundo ejemplo simple: Quiver
 - Tercer ejemplo
- Trabajo simple con imágenes
 - Análisis de la imagen
- Gráficos interactivos (“widgets”)
 - Cursor
 - Manejo de eventos
 - Ejemplos integrados

CAPÍTULO 20

Material adicional

- Puede descargar las Clases en formato pdf