
Clases de Python

Versión 2020.1

Juan Fiol

26 de febrero de 2020

1. Clase 0: Introducción al lenguaje Python orientado a Ingenierías y Física	1
2. Clase 1: Introducción al lenguaje	13
3. Clase 2: Tipos de datos y control	29
4. Clase 3: Control de flujo y tipos complejos	45
5. Clase 4: Algunas Técnicas de iteración	67
6. Clase 5: Funciones	85
7. Clase 6: Programación Orientada a Objetos	101
8. Ejercicios de Clase 1	117
9. Ejercicios de Clase 2	119
10. Ejercicios de Clase 3	123
11. Ejercicios de Clase 4	127
12. Ejercicios de Clase 5	129
13. Ejercicios de Clase 6	131
14. Programa completo	135

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

1.1 Autor

Juan Fiol

1.2 Licencia



Esta obra está bajo una [Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/).

1.3 Python y su uso en ingenierías y ciencias

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

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

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

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

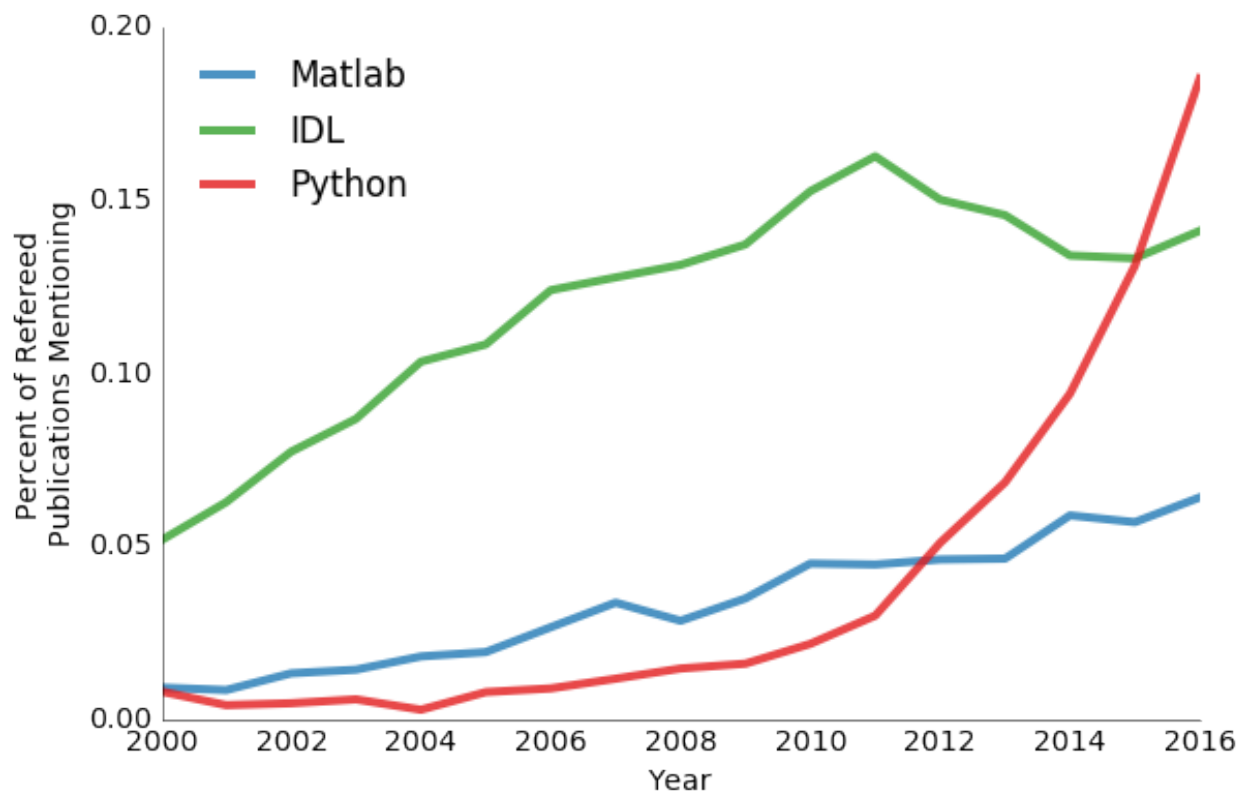
en la bolsa, en interacción con bases de datos, y en el procesamiento de gran número de datos como se encuentran en astronomía, biología, meteorología, etc.

En particular, Python encuentra un nicho de aplicación muy importante en varios aspectos muy distintos del trabajo de ingeniería, científico, o técnico. Por ejemplo, es un 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 inhabilite 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 esta razones el tiempo de desarrollo: desde la idea original hasta una versión que funciona correctamente puede ser mucho menor que en otros lenguajes.

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



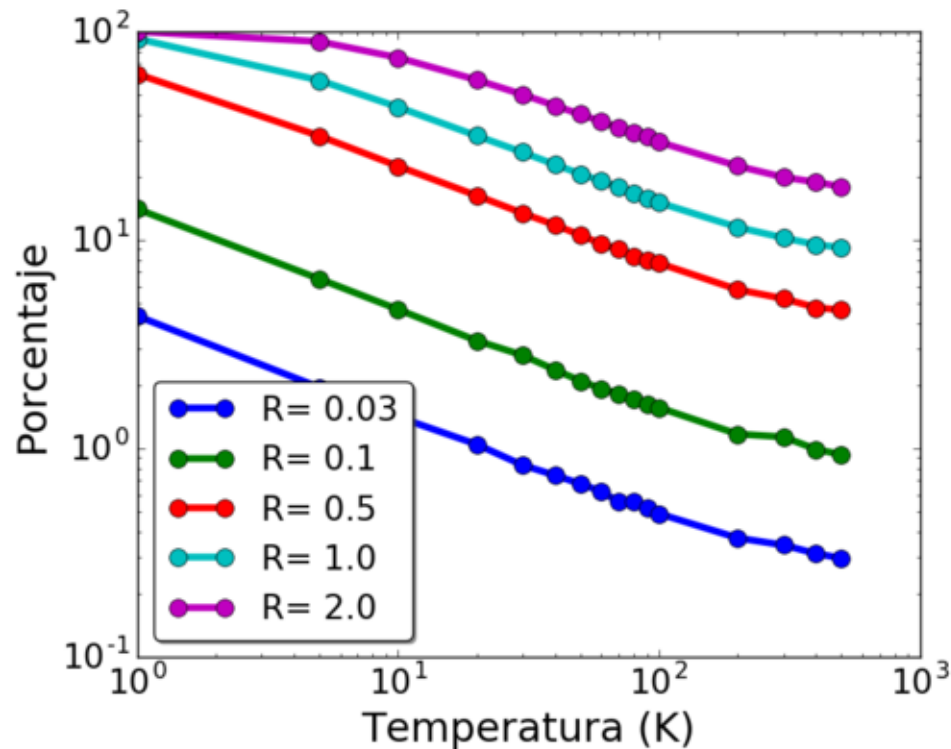
Un último punto que no puede dejar de mencionarse es que Python es libre (y gratis). Esto significa que cada versión nueva puede simplemente descargarse e instalarse sin limitaciones, sin licencias. Además, al estar disponible el código

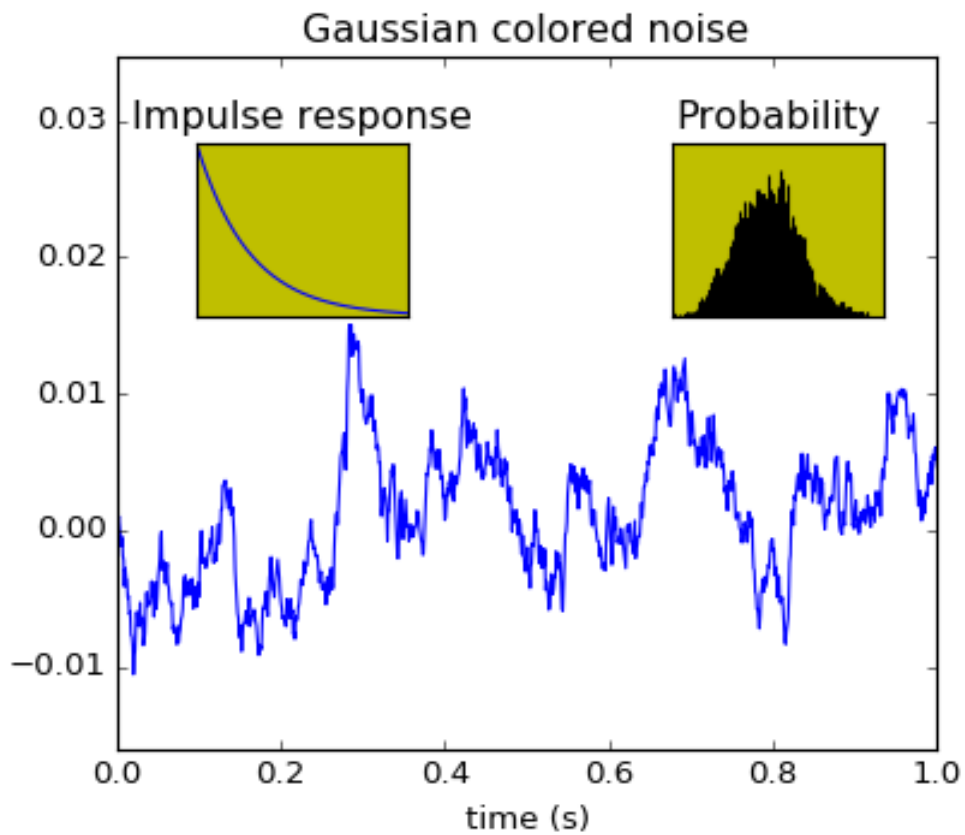
fuelle uno podría modificar el lenguaje –una situación que no es muy probable que ocurra– o podría mirar cómo está implementada alguna función –un escenario bastante más probable– para copiar (o tomar inspiración en) alguna funcionalidad que necesitamos en nuestro código.

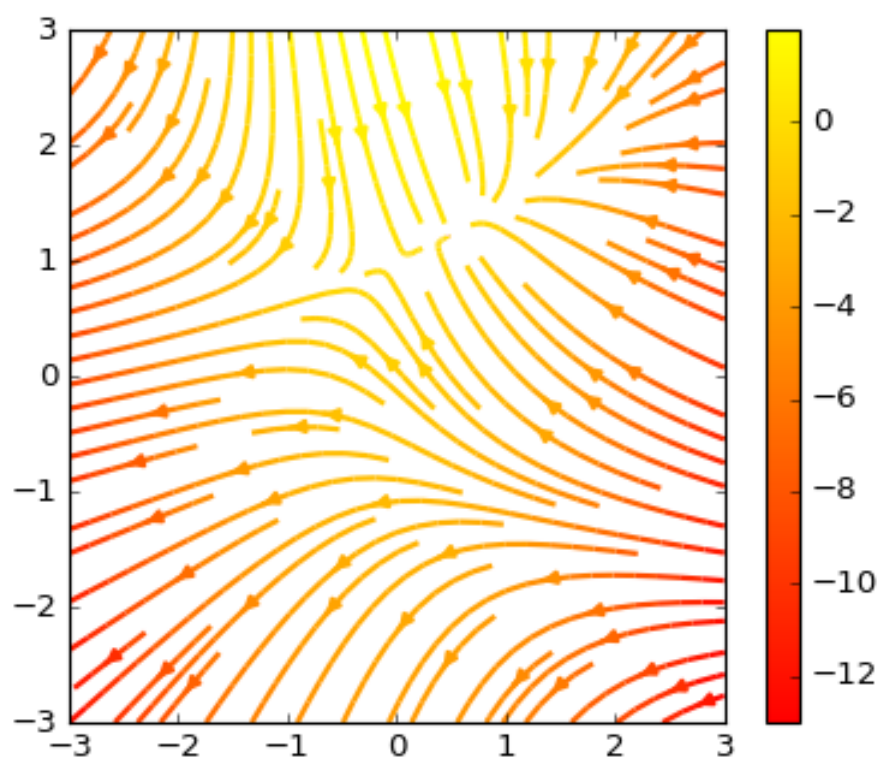
1.4 Visita y excursión a aplicaciones de Python

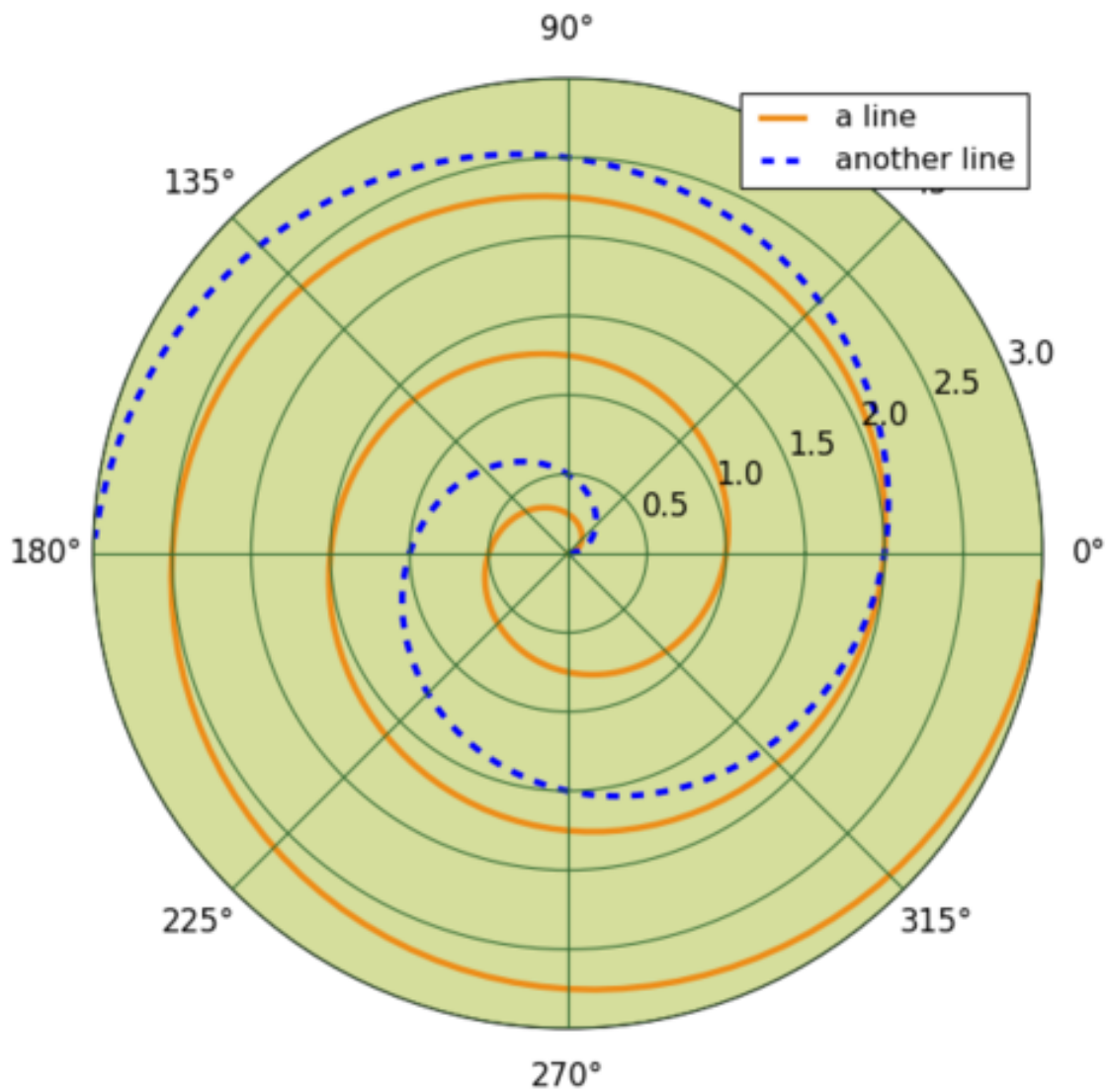
1.4.1 Graficación científica en 2D

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



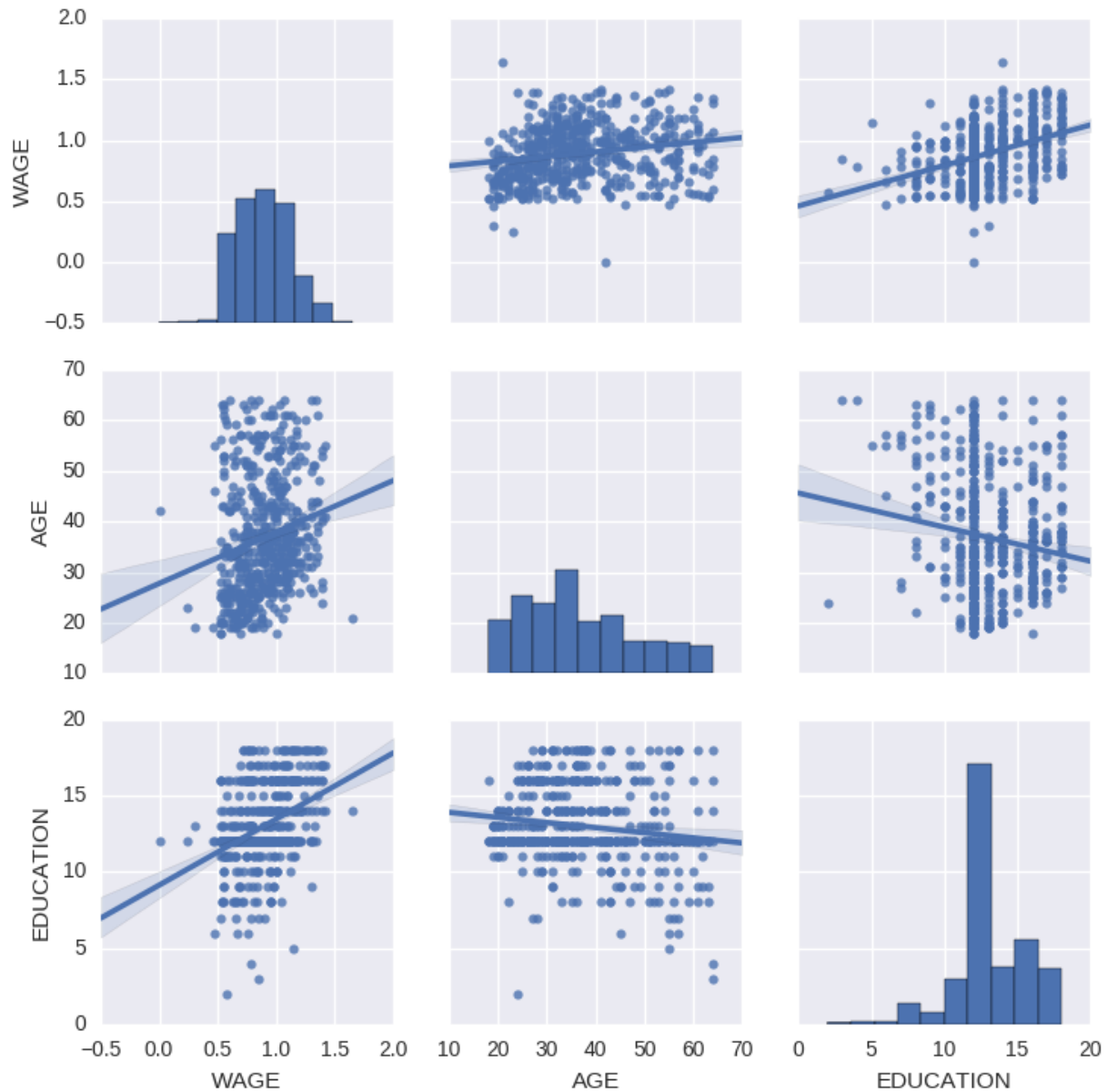






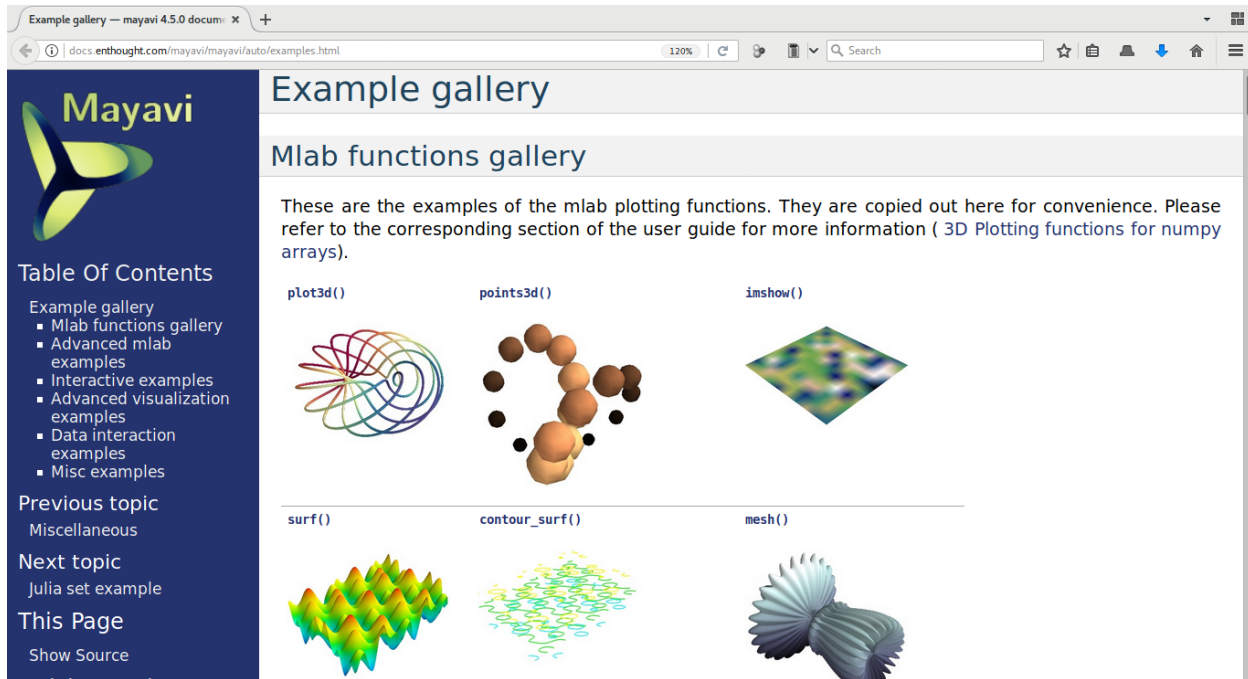
El mejor lugar para un acercamiento es posiblemente la [Galería de matplotlib](#)

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



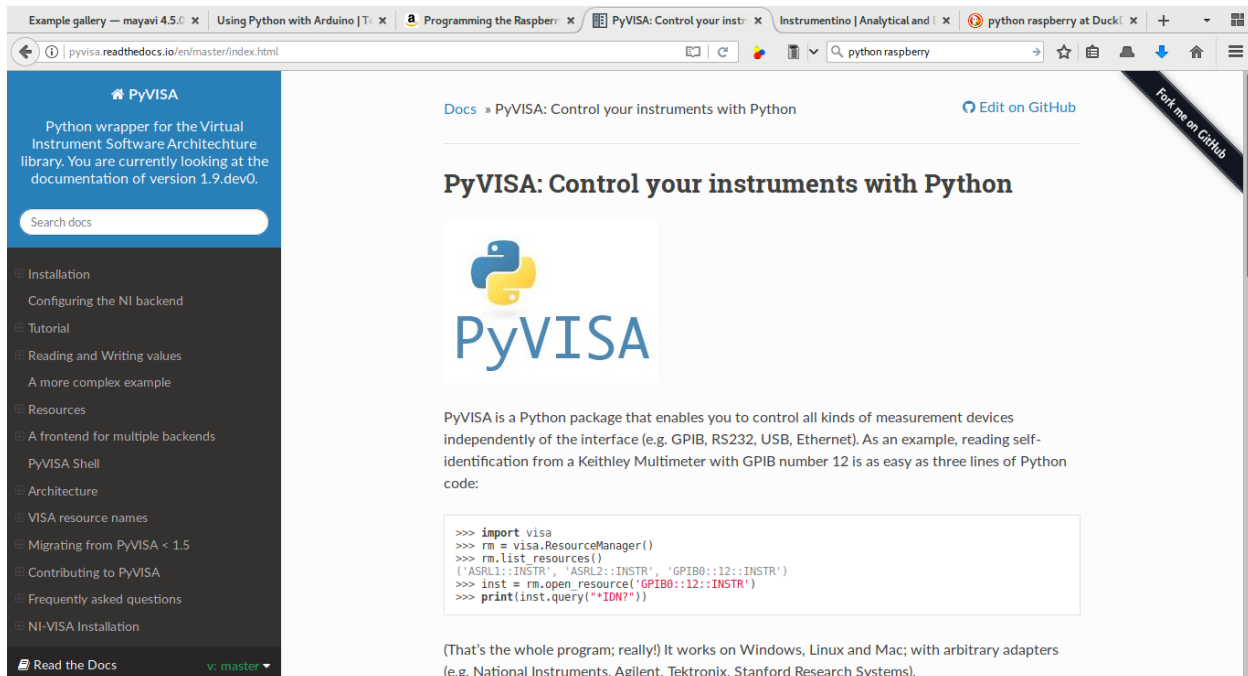
1.4.2 Graficación en 3D

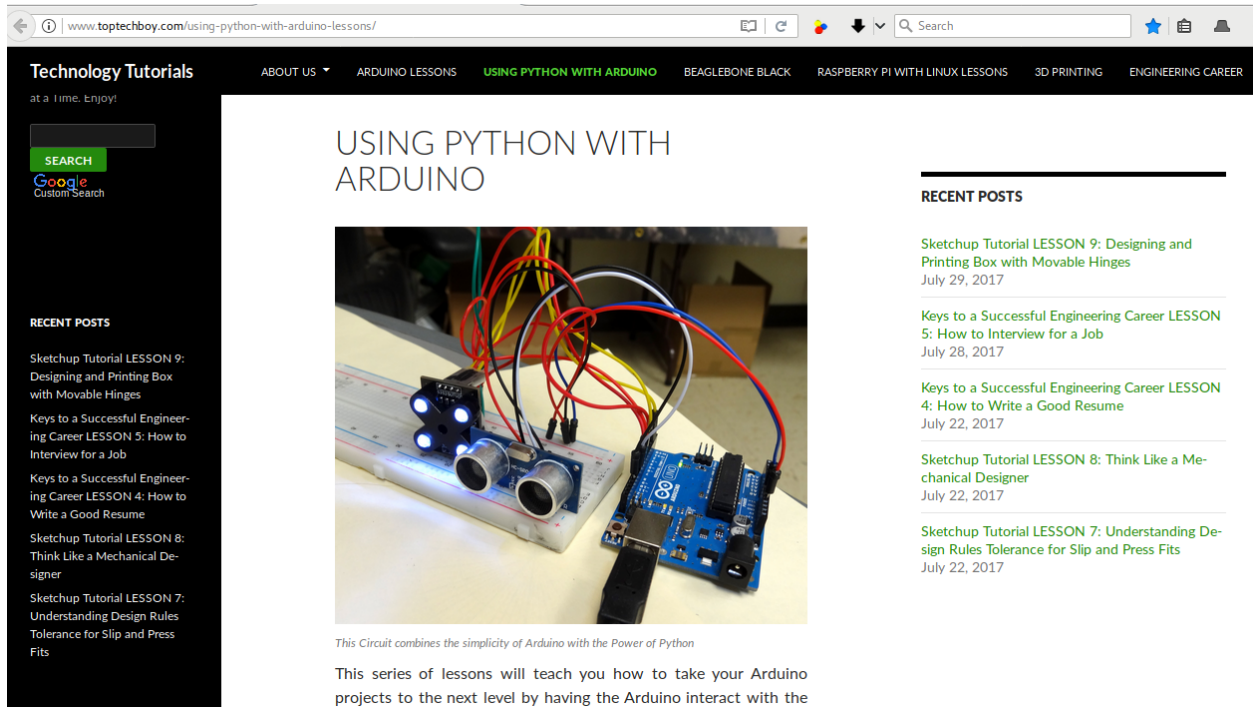
Matplotlib tiene capacidades para realizar algunos gráficos en 3D, si no son demasiado complejos. Para realizar gráficos de mayor complejidad, una de las más convenientes y populares bibliotecas/entornos es Mayavi:



1.4.3 Programación de dispositivos e instrumentos

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





1.4.4 Otras aplicaciones

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

1.5 Aplicaciones científicas

Vamos a aprender a programar en Python, y a utilizar un conjunto de bibliotecas creadas para uso científico y técnico: En este curso vamos a trabajar principalmente con Ipython y Jupyter, y los paquetes científicos [Numpy](#), [Scipy](#) y [Matplotlib](#).



Figura 1: Herramientas

1.6 Bibliografía

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

1.6.1 Accesible en línea

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

1.6.2 Libros

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

1.7 Otras referencias de interés

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

- Videos de Curso para Científicos e Ingenieros

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.

Versiones: Hay dos versiones activas del lenguaje Python.

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

2.1.1 Instalación

En este curso utilizaremos **Python 3**

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

- **Anaconda**. (Linux, Windows, MacOS)
- **Canopy**. (Linux, Windows, MacOS)
- **Winpython**. (Windows)
- **Python(x,y)**. (Windows, no actualizado desde 2015)

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

```
`sudo apt-get install ipython3 ipython3-notebook spyder python3-matplotlib python3-  
↳numpy python3-scipy`
```

o, en Fedora 28, en adelante:

```
`sudo dnf install python3-ipython python3-notebook python3-matplotlib python3-numpy_  
↳python3-scipy`
```

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

2.1.2 Documentación y ayudas

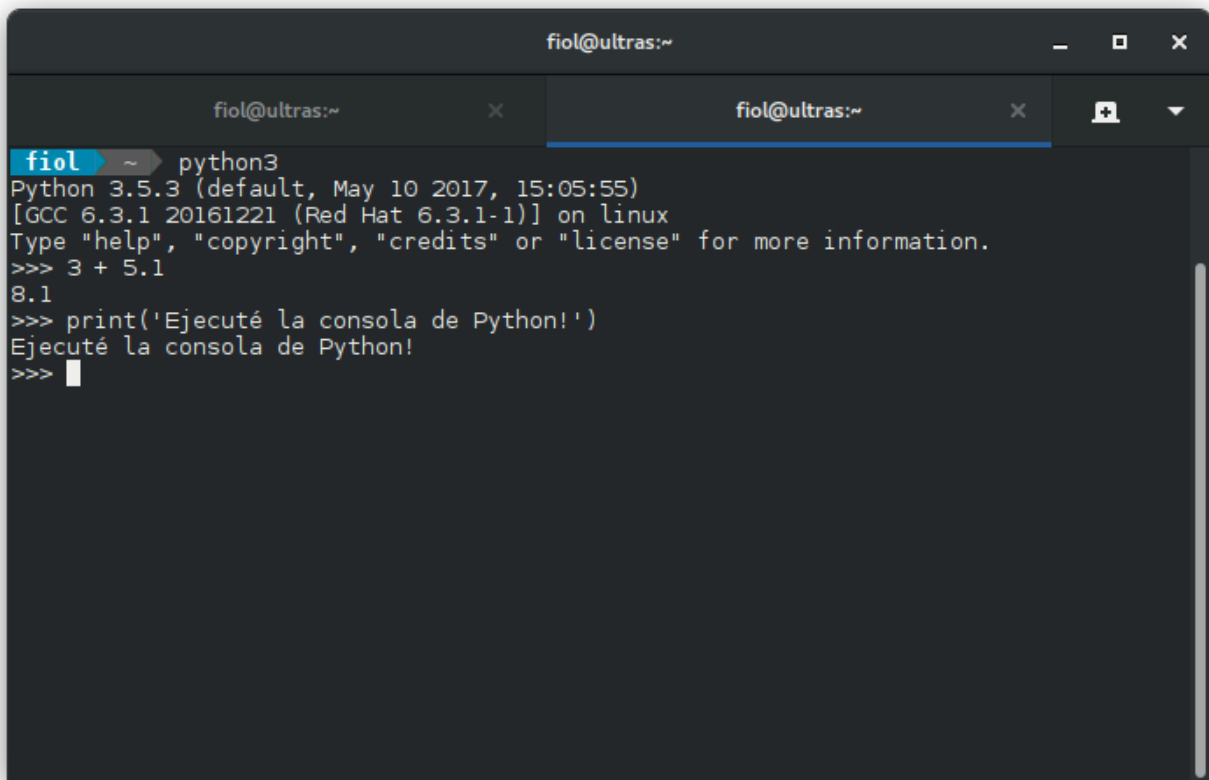
Algunas fuentes de ayuda *constante* son:

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

2.1.3 Uso de Python: Interactivo o no

Interfaces interactivas (consolas/terminales, notebooks)

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

A screenshot of a terminal window titled 'fiol@ultras:~'. The terminal shows a session where the user runs 'python3'. The output displays the Python version (3.5.3), the date and time (May 10 2017, 15:05:55), and the platform (linux). The user then enters '3 + 5.1' and the output is '8.1'. Finally, the user enters 'print('Ejecuté la consola de Python!')' and the output is 'Ejecuté la consola de Python!'. The prompt 'fiol ~' is visible at the top left of the terminal window.

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

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

```

fiol@ultras:~/trabajo/clases/pythons/clases-python/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

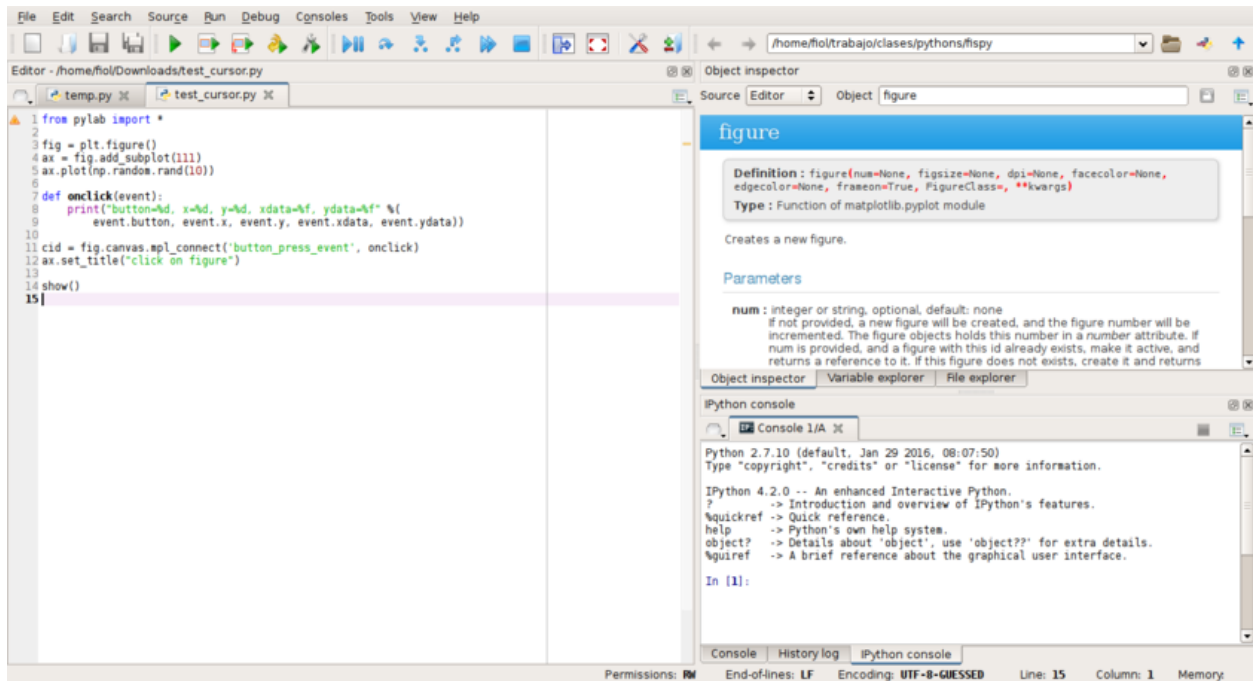
Una forma muy común/poderosa de usar Python es en forma *no interactiva*, escribiendo *programas* o *scripts*. Esto es, escribir nuestro código en un archivo con extensión *.py* para luego ejecutarlo con el intérprete. Por ejemplo, podemos crear un archivo *hello.py* (al que se le llama *módulo*) con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

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

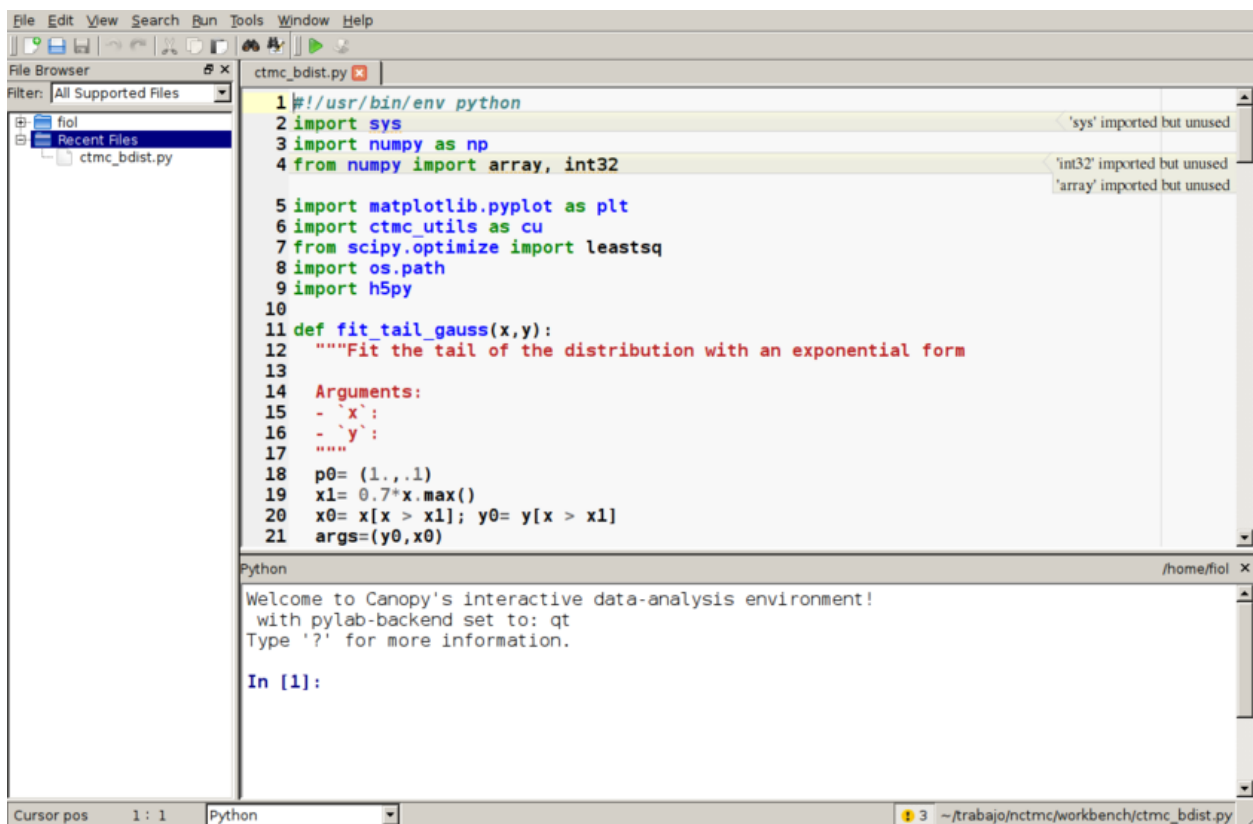
Si ejecutamos `python hello.py` o `ipython hello.py` se ejecutará el intérprete Python y obtendremos el resultado esperado (impresión por pantalla de *Hola Mundo!*, sin las comillas)

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 inscruados en la misma pantalla o en ventanas separadas. Además se puede escribir texto con formato (como este que estamos viendo) con secciones, títulos. Estos archivos se guardan con extensión *.ipynb*, que pueden exportarse en distintos formatos tales como html (estáticos), en formato PDF, LaTeX, o como código python puro. (.py)

2.2 Ejercicios 01 (a)

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

Ejecute el programa.

2.3 Comandos de IPython

2.3.1 Comandos de Navegación

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

```
%lsmagic
```

Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cat
↪%cd %clear %colors %conda %config %connect_info %cp %debug %dhist %dirs
↪%doctest_mode %ed %edit %env %gui %hist %history %killbgscripts %ldir
↪%less %lf %lk %ll %load %load_ext %loadpy %logoff %logon %logstart
↪%logstate %logstop %ls %lsmagic %lx %macro %magic %man %matplotlib %mkdir
↪%more %mv %notebook %page %pastebin %pdb %pdef %pdoc %pfile %pinfo
↪%pinf2 %pip %popd %pprint %precision %prun %psearch %psource %pushd %pwd
↪%pycat %pylab %qtconsole %quickref %recall %rehashx %reload_ext %rep %rerun
↪%reset %reset_selective %rm %rmdir %run %save %sc %set_env %store %sx
↪%system %tb %time %timeit %unalias %unload_ext %who %who_ls %whos %xdel
↪%xmode
```

Available cell magics:

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

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

2.3.2 Algunos de los comandos mágicos

Algunos de los comandos mágicos más importantes son:

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

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

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

```
%cd ~
```

```
/home/fiol
```

```
%pwd
```

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

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

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

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

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

```
%cd
```

```
/home/fiol
```

```
%mkdir scripts
```

```
%cd scripts
```

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

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

```
%cat prog1.py
```

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

```
%run prog1.py
```

```
hola y chau
```

```
%hist
```

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

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

```
%magic
```

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

```
%quickref
```


2.3.3 Comandos de Shell

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

comandos

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

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

```
%cat prog1.py
```

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

```
%run prog1.py
```

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

```
!date
```

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

2.4 Ejercicios 01 (b)

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

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

2.5 Conceptos básicos de Python

2.5.1 Características generales del lenguaje

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

- Es un lenguaje interpretado (no se compila separadamente)
- Provee tanto un entorno interactivo como de programas separados
- Las funciones, bloques, ámbitos están definidos por la indentación
- Tiene una estructura altamente modular, permitiendo su reusabilidad
- Es un lenguaje de *tipeado dinámico*, no tenemos que declarar el tipo de variable antes de usarla.

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

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

2.5.2 Tipos de variables

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

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

Tipos simples: Números

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
0.5
6.666666666666667
```

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

```
0
6
```

Nota: La función `print`

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

```
print?
```

```
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
```

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

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

Números complejos

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

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

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

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

```
1.5j * z2 + z1 = (-0.18599999999999994+4j)
z2^2 = (-0.5113760000000003+8.496j)
conj(z1) = (3-1j)
```

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

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

Operaciones

Las operaciones aritméticas básicas son:

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

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

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

Nota: Las funciones matemáticas están incluidas en el lenguaje.

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

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

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

Tipos simples: Booleanos

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

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

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

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

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

Hay un tipo *especial*, el elemento None.

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

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

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

```
d = 1
```

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

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

Operadores lógicos

Los operadores lógicos en Python son muy explícitos:

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

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

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

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

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

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

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

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

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

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

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

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

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

```
11 22 False
```

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

Esto sigue valiendo para otros números:

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

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

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

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

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

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

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

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

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

2.6 Ejercicios 01 (c)

7. Para el número complejo $z = 1 + 0,5i$

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

Clase 2: Tipos de datos y control

3.1 Escenas del capítulo anterior:

En la clase anterior preparamos la infraestructura:

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

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

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

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

3.2 Tipos de variables

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

Declaración, definición y asignación de valores a variables

3.2.1 Tipos simples

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

3.2.2 Disgresión: Objetos

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

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

Además, pueden tener: - componentes - métodos

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

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

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

```
2
```

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

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

```
14
```

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

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

```
0b11
0b11000000011000
```

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

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

```
True
```

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

```
False
```

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

```
c.as_integer_ratio()
```

```
(569, 4)
```

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

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

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

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

Entonces 0x1.1c8p+7 corresponde a:

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

```
142.25
```

Veamos como último ejemplo, los números complejos

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

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

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

3.2.3 Strings: Secuencias de caracteres

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

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

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

Operaciones

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

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

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

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

```
otro
```

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

```
dijo
```

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

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

```
mas
```

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

```
otromas = 'pSS€→"\'oó@ñ'
```

```
otromas
```

```
'pSS€→"\'oó@ñ'
```

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

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

Podemos imprimir los strings

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


- Tienen métodos (funciones que se aplican a su *dueño*)

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

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

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

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

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

Veamos cómo se puede operar sobre un string:

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

```
La mar estaba serena!
```

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

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

```
La mar astaba sarana!
```

o separar las palabras:

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

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

Estos son métodos que tienen definidos los *strings*.

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

Veamos algunos ejemplos más:

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

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

-----

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

Buscar y reemplazar cosas en un string:

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

```
6
```

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

```
12
```

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

```
12
```

```
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!
```

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

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

Formato de strings

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

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

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

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

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

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

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

3.3 Ejercicios 02 (a)

1. Centrado manual de frases
 - 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

3.4 Conversión de tipos

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

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


```

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

```

```

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

```

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

```

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

<ipython-input-43-9b48e3080d7d> in <module>
----> 1 print (1+'1')

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

```

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

```

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

```

```

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

```

3.5 Ejercicios 02 (b)

4. Para la cadena de caracteres:

```

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

```

* 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.

(continúe en la próxima página)

(proviene de la página anterior)

```

* Forme un nuevo string que contenga los 10 caracteres centrales de `s` (utilizando
↳ un método que pueda aplicarse a otros strings también). Imprima por pantalla el
↳ nuevo string.

* Cambie todas las letras "m" por "n" y todas las letras "n" por "m" en `s`.
↳ Imprima el resultado por pantalla.

* Debe entregar un programa llamado `02_SuApellido.py` (con su apellido, no la
↳ palabra "SuApellido"). El programa al correrlo con el comando `python3 SuApellido_
↳ 02.py` debe imprimir:

```

Nombre Apellido Clase 2 Distinguiendo: 2 5 1 2 Sin distinguir: 2 5 2 4 Que el hombre que lo desvela : longitud=28
 Aquí uela. desvela Un Aquí ne pongo a cantar Al compás de la vigüela, Que el hombre que lo desvela Una pema
 extraordinaria Cono la ave solitaria Com el cantar se consuela. ““

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]
```

```
cuadrados[3]
```

```
cuadrados[-1]
```

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

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

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

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

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

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

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

```
a is cuadrados
```

```
b = cuadrados.copy()
print (b)
print (cuadrados)
cuadrados[0] = -2
print (b)
print (cuadrados)
```

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
```

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

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

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

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

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

```
help(list)
```

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

```
print(L)
```

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

```
print(L)
```

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

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)
```

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

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

En las listas podemos sobrescribir uno o más elementos

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

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

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

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

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

```
range(2)
```

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

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

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

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

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
```

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
```

```
sum(L)
```

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

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 que 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 (y cmath)

El módulo **math** contiene las funciones más comunes (trigonómicas, 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 conviene importar sólo lo que vamos a usar, usando la notación:

```
from xxx import yyy
```

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

```
10.0
3.963327297606011 3
```

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

```
1.7888543819998317
```

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

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

<ipython-input-47-1158665f2c67> in <module>
      1 import math
----> 2 print(math.sqrt(-1))

ValueError: math domain error
```

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

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

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
```

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

3. Manejos de listas:

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

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

5. Operación rara sobre una lista:

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

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

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

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

- Obtenga una lista **L1** donde cada elemento sea una palabra.
- Cuente la cantidad de palabras en `s1` (utilizando `python`).
- Ordene la lista **L1** en orden alfabético.
- Ordene la lista **L1** tal que las palabras más cortas estén primero.
- Ordene la lista **L1** tal que las palabras más largas estén primero.
- Construya un string **s2** con la lista del resultado del punto anterior.
- Encuentre la palabra más larga y la más corta de la frase.

7. Escriba un script que encuentre las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$. Los valores de los parámetros defínalos en el mismo script, un poco más arriba.

8. Considere un polígono regular de N lados inscripto en un círculo de radio unidad:
- Calcule el ángulo interior del polígono regular de N lados (por ejemplo el de un triángulo es 60 grados, de un cuadrado es 90 grados, y de un pentágono es 108 grados). Exprese el resultado en grados y en radianes para valores de $N = 3, 5, 6, 8, 9, 10, 12$.
 - ¿Puede calcular la longitud del lado de los polígonos regulares si se encuentran inscriptos en un círculo de radio unidad?
9. Escriba un *script* (llamado `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:

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

3.7.2 Adicionales

11. Calcular la suma:

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

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

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

Clase 3: Control de flujo y tipos complejos

4.1 Control de flujo

4.1.1 if/elif/else

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

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

y en Python es muy parecido!

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

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

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

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

(continué en la próxima página)

(proviene de la página anterior)

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

```
Bueno, al menos aprobó!
```

4.1.2 Iteraciones

Sentencia for

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

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

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

Veamos otro ejemplo:

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

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

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

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

(continué en la próxima página)

(proviene de la página anterior)

```

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

```

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

```
suma += elemento
```

es equivalente a:

```
suma = suma + elemento
```

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

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

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

```
55
```

Loops: for, enumerate, continue, break, else

Veamos otros ejemplos de uso del bloque for.

```

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

```

```

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

```

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

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

Puntos a notar:

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

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

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

```
L
```

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

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

```

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

```

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

```

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

```

```

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

```

Veamos otro ejemplo, que puede encontrarse en la [documentación oficial](#):

```

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

```

```

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

```

Puntos a notar:

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

While

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

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

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

4.2 Ejercicios 03 (a)

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

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

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

4.3 Tipos complejos de datos

4.3.1 Listas y tuples

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

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

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

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

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

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

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

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

Veamos algunas definiciones y ejemplos utilizando listas y tuples:

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

```
'6:30'
```

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

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

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

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

```
l3 is t2?  True
l5 is t2?  False
(1, 2, 3, 'manzanas')
[1, 2, 3, 'manzanas']
```

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

```
l5 == t2?  False
l5 == list(t2)?  True
```

Mutabilidad

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

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

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

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

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

```
-----

TypeError                                Traceback (most recent call last)

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

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

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

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

```
-----

TypeError                                Traceback (most recent call last)

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

(continué en la próxima página)

(proviene de la página anterior)

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

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

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

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

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

4.3.2 Diccionarios

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

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

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

```
123
```

```
d0
```

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

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

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

```
Nombre: Juan
```

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

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

```
8400
```

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

```
d1
```

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

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

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

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

```
1213
1213
```

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

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

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

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

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

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

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

```
dict(it)
```

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

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

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

```
dict(it)
```

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

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

TypeError: unhashable type: 'list'
```

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

```
+++++
      Datos:
Juan García
```

(continué en la próxima página)

(proviene de la página anterior)

```
Av Bustillo 9500,
Bariloche

+++++
```

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

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

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

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

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

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

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

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

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

```
True
False
```

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

```
d1
```

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

```
d1[1]
```

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

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

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

```
d1
```

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

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

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

4.3.3 Conjuntos

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

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

```
mamiferos
```

```
{'gato', 'león', 'perro'}
```

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

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

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

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

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

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

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

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

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

```
{'bandurria', 'chimango', 'cóndor'}
```

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

```
{'bandurria', 'chimango', 'cóndor'}
```

```
domesticos - aves
```

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

```
l1= list(aves.difference(domesticos))  
print (l1)  
l1.sort(reverse=True)  
print(l1)  
print (l1[1])
```

```
['chimango', 'bandurria', 'cóndor']  
['cóndor', 'chimango', 'bandurria']  
chimango
```

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

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

Modificar conjuntos

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

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

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

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

```
c
```

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

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

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

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

```
c
```

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

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

```
c
```

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

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

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

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

KeyError: 2
```

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

```
c
```

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

4.4 Ejercicios 03 (b)

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

4.5 Escritura y lectura a archivos

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

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

```
f
```

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

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

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

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

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

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

4.5.1 Ejemplos

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

Ejemplo 03-1

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

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



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

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

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

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

print('\n')
if minlen == maxlen:
    mensaje = 'Encontramos {} palabras que tienen {} letras'.format(n, minlen)
else:
    mensaje = 'Encontramos {0} palabras que tienen entre {1} y {2} letras'\
        .format(n, minlen, maxlen)

print(mensaje)
```

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

Encontramos 420 palabras que tienen entre 3 y 4 letras

Hemos utilizado aquí:

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

La apertura de archivos se realiza utilizando la función `open` (este es un buen momento para mirar su documentación) con dos argumentos: el primero es el nombre del archivo y el segundo el modo en que queremos abrirlo (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,_
↪(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,_
↪(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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Encontramos un total de 45 palabras capicúa que tienen 3 letras
```

Aquí en lugar de leer todas las líneas e iterar sobre las líneas resultantes, iteramos directamente sobre el archivo abierto.

Además incluimos un string al principio del archivo, que servirá de documentación, y puede accederse mediante los mecanismos usuales de ayuda de Python.

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

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

```
!cat tmp.txt
```

```
Hola
 y chau
```

4.6 Ejercicios 03 (c)

2. 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), ...]
```

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

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

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

Para calcular la función de Bessel de orden N , se empieza con un valor de $M \gg N$, y utilizando los valores iniciales $J_M = 1$, $J_{M+1} = 0$ se utiliza la primera relación para calcular todos los valores de $n < M$. Luego, utilizando la segunda relación se normalizan todos los valores. **Nota:** Estas relaciones son válidas si $M \gg x$ (use algún valor estimado, como por ejemplo $M = N + 20$).

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

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

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

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

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

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

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

- 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 las siguientes listas de números enteros:

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

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

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

Clase 4: Algunas Técnicas de iteración

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

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

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

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

```
cs omHgtaánlei,
```

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

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

```
cs omHgtaánlei,
```

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

```
-----  
ValueError                                Traceback (most recent call last)  
  
<ipython-input-8-6170f6c3b697> in <module>  
----> 1 for j, t1, t2 in enumerate(zip(temp_min, temp_max)):  
      2     print('El día {} la temperatura mínima fue {} y la máxima fue {}'.  
→format(1+j, t[0], t[1]))  
  
ValueError: not enough values to unpack (expected 3, got 2)
```

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

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

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

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



```

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

```

```

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

```

```

1 3
2 4
3 5

```

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

```

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

```

```

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

```

```

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

```

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

También podemos escribirlo en forma más compacta usando comprensiones de listas

```

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

```

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

5.2 Más sobre diccionarios

5.2.1 Creación

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

```
d0
```

```
{}
```

```
d1
```

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

```
d2 == d1
```

```
True
```

```
d2 is d1
```

```
False
```

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

```
True
False
```

```
d3 == d1
```

```
True
```

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

```
d4
```

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

```
d5
```

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

¿Qué espera que produzca el siguiente código?

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

```
temps
```

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

5.2.2 Iteraciones sobre diccionarios

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

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

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

```
temps.keys()
```

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

```
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())
```

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

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

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

Si queremos iterar sobre los valores podemos utilizar simplemente:

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

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

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

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

```
1
```

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

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

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

La secuencia anterior puede escribirse en forma más compacta como

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

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

```
list(temps)
```

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

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

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

5.3 Ejercicios 04 (a)

1. Realice un programa para:

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

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

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

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

Note que la masa sólo debe contener 5 números decimales

5.4 Funciones

5.5 Las funciones son objetos

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

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

```
19.0
```

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

```
19.0
```

```
sum
```

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

```
print
```

```
<function print>
```

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

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

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

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

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

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

```
print('¿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, 'minimo': min, 'maximo': max}
```

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

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

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

```
19.0
```

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

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

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

5.6 Definición básica de funciones

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

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

```
fib(100)
```

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

```
fib
```

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

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

```
help(fib)
```

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

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

```
fib?
```

```
fib.__doc__
```

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

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

5.7 Argumentos de las funciones

5.7.1 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('Originalmente x vale', x)
func1(x)
print('Ahora x vale', x)
```

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

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

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

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

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

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

Consideremos estas variantes:

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

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

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

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

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

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

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

5.7.2 Funciones con argumentos opcionales

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
```

```
# 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))
```

```
(-139.99999426000002, 8.199999820135417e-05)
```

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

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

```
help(caida_libre)
```

```
Help on function caida_libre in module __main__:

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

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

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

Tipos mutables en argumentos opcionales

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

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

```
func2b(1)
```

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

```
func2b(2)
```

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

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

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

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

5.7.3 Número variable de argumentos y argumentos *keywords*

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

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

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

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

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

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

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

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

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

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

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

Exploremos otras variantes de llamadas a la función:

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

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

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

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

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

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

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

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

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

```
10
```

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

```
3240
```

5.8 Ejercicios 4 (b)

2. Escriba funciones para analizar la divisibilidad de enteros:

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

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

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

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


Clase 5: Funciones

6.1 Empacar y desempacar secuencias o diccionarios

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

Al definir un número variable de argumentos de la forma:

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

y luego utilizarla en alguna de las formas:

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

Python automáticamente convierte los argumentos en una única tupla:

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

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

Si definimos una función que recibe un número determinado de argumentos

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

y definimos una lista (o tupla)

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

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

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

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

```
t2 = *t1
```

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

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

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

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

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

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

```
(-52.920000000000001, 857.116)
```

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

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

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

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

En resumen:

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

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

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

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

6.2 Funciones que devuelven funciones

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

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

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

6.3 Ejercicios 05 (a)

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

Al evaluar la función f en un dado valor x debe dar el resultado: $f(x) = A \sin(wx)$ tal que se pueda utilizar de la siguiente manera: `python f = crear_sen(3, 1.5) f(2)` # Debería imprimir el resultado de $3 \sin(1.5 \cdot 2) = 0.4233600241796016$

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

6.4 Funciones que toman como argumento una función

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

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

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

```
t
```

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

El ejemplo anterior se podría escribir

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

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

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

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

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

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

Veamos otro ejemplo:

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

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

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

6.5 Aplicacion 1: Ordenamiento de listas

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

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

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

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

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

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

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

```
help(sorted)
```

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

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

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

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

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

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

Como vemos, los strings están ordenados adecuadamente. Si queremos ordenarlos por longitud de la palabra

```
sorted(s1, key=len)
```

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

Supongamos que queremos ordenarla alfabéticamente por la segunda letra

```
def segunda(a):
    return a[1]

sorted(s1, key=segunda)
```

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

6.6 Funciones anónimas

En ocasiones como esta suele ser más rápido (o conveniente) definir la función, que se va a utilizar una única vez, sin darle un nombre. Estas se llaman funciones *lambda*, y el ejemplo anterior se escribiría

```
sorted(s1, key=lambda a: a[1])
```

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

Si queremos ordenarla alfabéticamente empezando desde la última letra:

```
sorted(s1, key=lambda a: a[::-1])
```

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

6.7 Ejemplo 1: Integración numérica

Veamos en más detalle el caso de funciones que reciben como argumento otra función, estudiando un caso usual: una función de integración debe recibir como argumento al menos una función a integrar y los límites de integración:

```
# %load scripts/05_ejemplo_1.py
def integrate_simps(f, a, b, N=10):
    """Calcula numéricamente la integral de la función en el intervalo dado
    utilizando la regla de Simpson

    Keyword Arguments:
    f -- Función a integrar
    a -- Límite inferior
    b -- Límite superior
    N -- El intervalo se separa en 2*N intervalos
    """
    h = (b - a) / (2 * N)
    I = f(a) - f(b)
    for j in range(1, N + 1):
        x2j = a + 2 * j * h
        x2jm1 = a + (2 * j - 1) * h
        I += 2 * f(x2j) + 4 * f(x2jm1)
    return I * h / 3
```

En este ejemplo programamos la fórmula de integración de Simpson para obtener la integral de una función $f(x)$ provista por el usuario, en un dado intervalo:

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[f(x_0) + 2 \sum_{j=1}^{n/2} f(x_{2j}) + 4 \sum_{j=1}^{n/2} f(x_{2j-1}) - f(x_n) \right]$$

¿Cómo usamos la función de integración?

```
def potencia2(x):
    return x**2

integrate_simps(potencia2, 0, 3, 7)
```

```
9.0
```

Acá definimos una función, y se la pasamos como argumento a la función de integración.

6.7.1 Uso de funciones anónimas

Veamos como sería el uso de funciones anónimas en este contexto

```
integrate_simps(lambda x: x**2, 0, 3, 7)
```

```
9.0
```

La notación es un poco más corta, que es cómodo pero no muy relevante para un caso. Si queremos, por ejemplo, aplicar el integrador a una familia de funciones la notación se simplifica notablemente:

```
print('Integrales:')
a = 0
b = 3
for n in range(6):
    I = integrate_simps(lambda x: (n + 1) * x**n, a, b, 10)
    print('I ( {} x^{}, {}, {} ) = {:.5f}'.format(n + 1, n, a, b, I))
```

```
Integrales:
I ( 1 x^0, 0, 3 ) = 3.00000
I ( 2 x^1, 0, 3 ) = 9.00000
I ( 3 x^2, 0, 3 ) = 27.00000
I ( 4 x^3, 0, 3 ) = 81.00000
I ( 5 x^4, 0, 3 ) = 243.00101
I ( 6 x^5, 0, 3 ) = 729.00911
```

Este es un ejemplo de uso de las funciones anónimas `lambda`. Recordemos que la forma general de las funciones `lambda` es:

```
lambda x, y, z: expresión_de(x, y, z)
```

por ejemplo en el ejemplo anterior, para calcular $(n + 1)x^n$, hicimos:

```
lambda x: (n+1) * x**n
```

6.8 Ejemplo 2: Polinomio interpolador

Veamos ahora una función que retorna una función. Supongamos que tenemos una tabla de puntos (x, y) por los que pasan nuestros datos y queremos interpolar los datos con un polinomio.

Sabemos que dados N puntos, hay un único polinomio de grado N que pasa por todos los puntos. En este ejemplo utilizamos la fórmula de Lagrange para obtenerlo.

```
%load scripts/ejemplo_05_2.py
```

```
# %load scripts/ejemplo_05_2.py
def polinomio_interp(x, y):
    """Devuelve el polinomio interpolador que pasa por los puntos (x_i, y_i)

    Warning: La implementación es numéricamente inestable. Funciona para algunos
    puntos (menor a 20)

    Keyword Arguments:
    x -- Lista con los valores de x
```

(continué en la próxima página)

(proviene de la página anterior)

```

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

```

(continué en la próxima página)

(proviene de la página anterior)

```
f3 = polinomio_interp(x3, y3)

print('\n x      f2(x)      f3(x)\n' + 18 * '-')
for j in range(N):
    x = step * j
    print('{:.1f}   {:.2f}   {:.2f}'.format(x, f2(x), f3(x)))
```

x	f2(x)	f3(x)
0.0	0.00	0.00
0.2	0.04	0.02
0.4	0.16	0.13
0.6	0.36	0.43
0.8	0.64	1.02
1.0	1.00	2.00
1.2	1.44	3.46
1.4	1.96	5.49
1.6	2.56	8.19
1.8	3.24	11.66
2.0	4.00	16.00
2.2	4.84	21.30
2.4	5.76	27.65
2.6	6.76	35.15
2.8	7.84	43.90
3.0	9.00	54.00
3.2	10.24	65.54
3.4	11.56	78.61
3.6	12.96	93.31
3.8	14.44	109.74
4.0	16.00	128.00
4.2	17.64	148.18
4.4	19.36	170.37
4.6	21.16	194.67
4.8	23.04	221.18

6.9 Ejercicios 05 (b)

1. 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.

6.10 Funciones que aceptan y devuelven funciones (Decoradores)

Consideremos la siguiente función `mas_uno`, que toma como argumento una función y devuelve otra función.

```
def mas_uno(func):  
    "Devuelve una función"  
    def fun(args):  
        "Agrega 1 a cada uno de los elementos y luego aplica la función"  
        xx = [x+1 for x in args]  
        y= func(xx)  
        return y  
    return fun
```

```
ssum= mas_uno(sum)           # h es una función  
mmin= mas_uno(min)          # f es una función  
mmax= mas_uno(max)           # g es una función
```

```
a = [0, 1, 3.3, 5, 7.5, 2.2]  
print(a)  
print(sum(a), ssum(a))  
print(min(a), mmin(a))  
print(max(a), mmax(a))
```

6.10.1 Notación para decoradores

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

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

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

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

Notemos que al decorar una función estamos creando una enteramente nueva

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

```
parabola(a)
```

```
mparabola(a)
```

Algunas veces queremos eso, renombrar la función original (se lo llama decorar):

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

Son un montón de líneas, podemos simplificarlo:

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

Esta es una situación que puede darse frecuentemente en algunas áreas. Se decidió simplificar la notación, introduciendo el uso de @. Lo anterior puede escribirse como:

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

La única restricción para utilizar esta notación es que la línea con el decorador debe estar inmediatamente antes de la definición de la función a decorar

```
mi_parabola(a)
```

6.10.2 Algunos Usos de decoradores

```
mi_parabola(3)
```

El problema aquí es que definimos la función para tomar como argumentos una lista (o al menos un iterable de números) y estamos tratando de aplicarla a un número. Podemos definir un decorador para asegurarnos que el tipo es correcto (que no es del todo correcto)

```
def test_argumento_list_num(f):
    def check(v):
        if (type(v) == list):
            return f(v)
        else:
            print("Error: El argumento debe ser una lista")
    return check
```

```
mi_parabola = test_argumento_list_num(mi_parabola)
```

```
mi_parabola(3)
```

```
mi_parabola(a)
```

Supongamos que queremos simplemente extender la función para que sea válida también con argumentos escalares. Definimos una nueva función que utilizaremos como decorador

```
def hace_argumento_list(f):
    def check(v):
        "Corrige el argumento si no es una lista"
        if (type(v) == list):
            return f(v)
        else:
            return f([v])
    return check
```

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

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

6.11 Atrapar y administrar errores

Python tiene incorporado un mecanismo para atrapar errores de distintos tipos, así como para generar errores que den información al usuario sobre usos incorrectos del código.

En primer lugar consideremos lo que se llama un error de sintaxis

```
print("hola"))
```

```
File "<ipython-input-11-3e8c6f917d42>", line 1
    print("hola"))
            ^
SyntaxError: invalid syntax
```

El intérprete detecta el error y repite la línea donde lo identifica. Este tipo de errores debe corregirse para poder seguir con el programa.

Consideremos ahora el código siguiente, que es sintácticamente correcto pero igualmente causa un error

```
a = 1
b = 0
z = a / b
```

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

<ipython-input-10-c4a2a9b4b376> in <module>
----> 1 z = 1/0

ZeroDivisionError: division by zero
```

Cuando se encuentra un error, **Python** muestra el lugar en que ocurre y de qué tipo de error se trata.

```
print (hola)
```

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

<ipython-input-12-c6948929a301> in <module>
----> 1 print (hola)

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

Este mensaje da un tipo de error diferente. Ambos: `ZeroDivisionError` y `NameError` son tipos de errores (o excepciones). Hay una larga lista de tipos de errores que son parte del lenguaje y puede consultarse en la documentación de [Built-in Exceptions](#).

6.11.1 Administración de excepciones

Cuando nuestro programa aumenta en complejidad, aumenta la posibilidad de encontrar errores. Esto se incrementa si se tiene que interactuar con otros usuarios o con datos externos. Consideremos el siguiente ejemplo simple:

```
%cat ../data/ej_clase5.dat
```

```
1 2
2 6
3 9
4 12
5.5 30.25
```

```
with open("../data/ej_clase5.dat") as fi:
    for l in fi:
        t = l.split()
        print("t = {}".format(t))          # Línea sólo para inspección
        m = int(t[0])
        n = int(t[1])
        print("m = {}, n = {}, m x n = {}".format(m,n, m*n))
```

```
t = ['1', '2']
m = 1, n = 2, m x n = 2
t = ['2', '6']
m = 2, n = 6, m x n = 12
t = ['3', '9']
m = 3, n = 9, m x n = 27
t = ['4', '12']
m = 4, n = 12, m x n = 48
t = ['5.5', '30.25']
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-25-55192d3677b3> in <module>
      3     t = l.split()
      4     print("t = {}".format(t))          # Línea sólo para inspección
----> 5     m = int(t[0])
      6     n = int(t[1])
      7     print("m = {}, n = {}, m x n = {}".format(m,n, m*n))

ValueError: invalid literal for int() with base 10: '5.5'
```

En este caso se levanta una excepción del tipo `ValueError` debido a que este valor (5.5) no se puede convertir a `int`. Podemos modificar nuestro programa para manejar este error:

```
with open("../data/ej_clase5.dat") as fi:
    for l in fi:
        t = l.split()
        try:
            m = int(t[0])
            n = int(t[1])
            print("m = {}, n = {}, m x n = {}".format(m,n, m*n))
        except:
            print("Error: t = {} no puede convertirse a entero".format(t))
```

```
m = 1, n = 2, m x n = 2
m = 2, n = 6, m x n = 12
```

(continué en la próxima página)

(proviene de la página anterior)

```

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
Error: t = ['6'] 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

```

```

-----
IndexError                                Traceback (most recent call last)

<ipython-input-36-52fbc3b0e675> in <module>
      4     try:
      5         m = int(t[0])
----> 6         n = int(t[1])
      7         print("m = {}, n = {}, m x n = {}".format(m,n, m*n))
      8     except (ValueError):

IndexError: list index out of range

```

```

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 línea "{}" 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
Error: La línea "6" no contiene un par

```

La forma general

La declaración `try` funciona de la siguiente manera:

- Primero, se ejecuta el *bloque try* (el código entre las declaraciones `try` y `except`).
- Si no ocurre ninguna excepción, el *bloque except* se saltea y termina la ejecución de la declaración `try`.
- Si ocurre una excepción durante la ejecución del *bloque try*, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada `except`, se ejecuta el *bloque except*, y la ejecución continúa luego de la declaración `try`.
- Si ocurre una excepción que no coincide con la excepción nombrada en el `except`, esta se pasa a declaraciones `try` de más afuera; si no se encuentra nada que la maneje, es una *excepción no manejada*, y la ejecución se frena con un mensaje como los mostrados arriba.

El mecanismo es un poco más complejo, y permite un control más fino que lo descripto aquí.

6.11.2 Levantando excepciones

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

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

```
-----

Exception                                Traceback (most recent call last)

<ipython-input-39-c3c4a53042e7> in <module>
      1 x = 1
      2 if x > 0:
----> 3     raise Exception("x = {}, no debería ser positivo".format(x))

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

O podemos ser más específicos, y dar el tipo de error adecuado

```
x = 1
if x > 0:
    raise ValueError("x = {}, no debería ser positivo".format(x))
```

```
-----

ValueError                                Traceback (most recent call last)

<ipython-input-38-3c0ee90e0415> in <module>
      1 x = 1
      2 if x > 0:
----> 3     raise ValueError("x = {}, no debería ser positivo".format(x))

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

Clase 6: Programación Orientada a Objetos

7.1 Breve introducción a Programación Orientada a Objetos

Vimos como escribir funciones que realizan un trabajo específico y nos devuelven un resultado. La mayor parte de nuestros programas van a estar diseñados con un hilo conductor principal, que utiliza una serie de funciones para realizar el cálculo. De esta manera, el código es altamente reusable.

Hay otras maneras de organizar el código, particularmente útil cuando un conjunto de rutinas comparte un dado conjunto de datos. En ese caso, puede ser adecuado utilizar un esquema de programación orientada a objetos.

7.2 Clases y Objetos

Una Clase define características, que tienen los objetos de dicha clase. En general la clase tiene: un nombre y características (campos o atributos y métodos).

Un Objeto en programación puede pensarse como la representación de un objeto real, de una dada clase. Un objeto real tiene una composición y características, y además puede realizar un conjunto de actividades (tiene un comportamiento). Cuando programamos, las partes son los datos, y el comportamiento son los métodos.

Ejemplos de la vida diaria serían: Una clase *Bicicleta*, y muchos objetos del tipo bicicleta (mi bicicleta, la suya, etc). La definición de la clase debe contener la información de qué es una bicicleta (dos ruedas, manubrio, etc) y luego se realizan muchas copias del tipo bicicleta (los objetos).

Se dice que los **objetos** son instancias de una **clase**, por ejemplo ya vimos los números enteros. Cuando definimos: `a = 3` estamos diciendo que `a` es una instancia (objeto) de la clase `int`.

Los objetos pueden guardar datos (en este caso `a` guarda el valor 3). Las variables que contienen los datos de los objetos se llaman usualmente campos o atributos. Las acciones que tienen asociadas los objetos se realizan a través de funciones internas, que se llaman métodos.

Las clases se definen con la palabra reservada `class`, veamos un ejemplo simple:

```
class Punto:
    "Clase para describir un punto en el espacio"

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

```
P1 = Punto(0.5, 0.5, 0)
```

```
P1
```

```
<__main__.Punto at 0x7f95188cb610>
```

```
P1.x
```

```
0.5
```

Como vemos, acabamos de definir una clase de tipo Punto. A continuación definimos un *método* `__init__` que hace el trabajo de inicializar el objeto.

Algunos puntos a notar:

- La línea `P1 = Punto(0.5, 0.5, 0)` crea un nuevo objeto del tipo Punto. Notar que usamos paréntesis como cuando llamamos a una función pero Python sabe que estamos llamando a una clase.
- El método `__init__` es especial y es el Constructor de objetos de la clase. Es llamado automáticamente al definir un nuevo objeto de esa clase. Por esa razón, le pasamos los dos argumentos al crear el objeto.
- El primer argumento del método, `self`, debe estar presente en la definición de todos los métodos pero no lo pasamos como argumento cuando hacemos una llamada a la función. **Python** se encarga de pasarlo en forma automática. Lo único relevante de este argumento es que es el primero para todos los métodos, el nombre `self` puede cambiarse por cualquier otro **pero, por convención, no se hace**.

```
P2 = Punto()
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-5-b35d1ccc6ec0> in <module>
----> 1 P2 = Punto()

TypeError: __init__() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Por supuesto la creación del objeto falla si no le damos ningún argumento porque los argumentos de `__init__` no son opcionales. Modifiquemos eso y aprovechamos para definir algunos otros métodos que pueden ser útiles:

```
from math import atan2

class Punto:
    "Clase para describir un punto en el espacio"

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
```

(continué en la próxima página)

(proviene de la página anterior)

```

self.x = x
self.y = y
self.z = z
return None

def angulo_azimuthal(self):
    "Devuelve el ángulo que forma con el eje x, en radianes"
    return atan2(self.y, self.x)

```

```
P1 = Punto(0.5, 0.5)
```

```
P1.angulo_azimuthal()
```

```
0.7853981633974483
```

```
P2 = Punto()
```

```
P2.x
```

```
0
```

```
help(P1)
```

```

Help on Punto in module __main__ object:

class Punto(builtins.object)
|   Punto(x=0, y=0, z=0)
|
|   Clase para describir un punto en el espacio
|
|   Methods defined here:
|
|   __init__(self, x=0, y=0, z=0)
|       Inicializa un punto en el espacio
|
|   angulo_azimuthal(self)
|       Devuelve el ángulo que forma con el eje x, en radianes
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)

```

```
P1.__dict__
```

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

```

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

```

```
0.7853981633974483
0.7853981633974483
```

Vemos que al hacer la llamada a los métodos, omitimos el argumento `self`. El lenguaje traduce nuestro llamado: `P1.angulo_azimuthal()` como `Punto.angulo_azimuthal(P1)` ya que `self` se refiere al objeto que llama al método.

7.3 Herencia

Una de las características de la programación orientada a objetos es la alta reutilización de código. Uno de los mecanismos más importantes es a través de la herencia. Cuando definimos una nueva clase, podemos crearla a partir de un objeto que ya exista. Por ejemplo, utilizando la clase `Punto` podemos definir una nueva clase para describir un vector en el espacio:

```
class Vector(Punto):
    "Representa un vector en el espacio"

    def suma(self, v2):
        "Calcula un vector que contiene la suma de dos vectores"
        print("Aún no implementada la suma de dos vectores")
        # código calculando v = suma de self + v2
        # ...

    def producto(self, v2):
        "Calcula el producto interno entre dos vectores"
        print("Aún no implementado el producto interno de dos vectores")
        # código calculando el producto interno pr = v1 . v2

    def abs(self):
        "Devuelve la distancia del punto al origen"
        print("Aún no implementado la norma del vector")
        # código calculando el producto interno pr = v1 . v2
```

Acá hemos definido un nuevo tipo de objeto, llamado `Vector` que se deriva de la clase `Punto`. Veamos cómo funciona:

```
v1 = Vector(2,3.1)
v2 = Vector()
```

```
v1
```

```
<__main__.Vector at 0x7f951885e650>
```

```
v1.x
```

```
2
```

```
v1.angulo_azimuthal()
```

```
0.9978301839061905
```

```
v1.x, v1.y, v1.z
```

```
(2, 3.1, 0)
```

```
v2.x, v2.y, v2.z
```

```
(0, 0, 0)
```

```
v = v1.suma(v2)
```

Aún no implementada la suma de dos vectores

```
print(v)
```

```
None
```

```
class Vector(Punto):
    "Representa un vector en el espacio"

    def __add__(self, v2):
        "Calcula un vector que contiene la suma de dos vectores"
        print("Aún no implementada la suma de dos vectores")
        # código calculando v = suma de self + v2
        # ...

    def producto(self, v2):
        "Calcula el producto interno entre dos vectores"
        print("Aún no implementado el producto interno de dos vectores")
        # código calculando el producto interno pr = v1 . v2

    def abs(self):
        "Devuelve la distancia del punto al origen"
        print("Aún no implementado la norma del vector")
        # código calculando el producto interno pr = v1 . v2
```

```
v1 = Vector(1,2,3)
v2 = Vector(1,2,-3)
```

```
v1 + v2
```

Aún no implementada la suma de dos vectores

Los métodos que habíamos definido para los puntos del espacio, son accesibles para el nuevo objeto. Además podemos agregar (extender) el nuevo objeto con otros atributos y métodos.

Como vemos, aún no está implementado el cálculo de las distintas funciones, eso forma parte del siguiente

7.4 Ejercicios 06 (a)

1. Implemente los métodos suma, producto y abs

- `suma()` debe retornar un objeto del tipo `Vector` y contener en cada componente la suma de las componentes de los dos vectores que toma como argumento.

- `producto` toma como argumentos dos vectores y retorna un número real

- `abs` toma como argumentos el propio objeto y retorna un número real

Su uso será el siguiente:

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

7.5 Objetos y clases

7.6 Atributos de clases y de instancias

Las variables que hemos definido pertenecen a cada objeto. Por ejemplo cuando hacemos

```
class Punto:
    "Clase para describir un punto en el espacio"
    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        return None

    def angulo_azimuthal(self):
        "Devuelve el ángulo que forma con el eje x, en radianes"
        return atan2(self.y, self.x)
```

```
p1 = Punto(1,2,3)
p2 = Punto(4,5,6)
```

cada vez que creamos un objeto de una dada clase, tiene un dato que corresponde al objeto. En este caso tanto `p1` como `p2` tienen un atributo llamado `x`, y cada uno de ellos tiene su propio valor:

```
print(p1.x, p2.x)
```

```
1 4
```

De la misma manera, en la definición de la clase nos referimos a estas variables como `self.x`, indicando que pertenecen a una instancia de una clase (o, lo que es lo mismo: un objeto específico).

También existe la posibilidad de asociar variables (datos) con la clase y no con una instancia de esa clase (objeto). En el siguiente ejemplo, la variable `num_puntos` no pertenece a un punto en particular sino a la clase del tipo `Punto`

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0
    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None
```

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

```
Número de puntos: 0
<__main__.Punto object at 0x7f4d4884bdd0> <__main__.Punto object at 0x7f4d4884bf90>
Número de puntos: 2
```

```
p1.__dict__
```

```
{'x': 1, 'y': 1, 'z': 1}
```

Si estamos contando el número de puntos que tenemos, podemos crear métodos para acceder a ellos y/o manipularlos. Estos métodos no se refieren a una instancia en particular (p1 o p2 en este ejemplo) sino al tipo de objeto `Punto` (a la clase)

```
del p1
del p2
```

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

    def borrar(self):
        "Borra el punto"
        Punto.num_puntos -= 1

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

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

```
Número de puntos: 0
<__main__.Punto object at 0x7f4d488557d0> <__main__.Punto object at 0x7f4d488559d0>
En total hay 2 puntos definidos
```

```
p1.total()
```

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

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

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

Sin embargo, no estamos removiendo p1, sólo estamos actualizando el contador:

```
p1.x
```

```
1
```

7.7 Algunos métodos especiales

Hay algunos métodos que **Python** interpreta de manera especial. Ya vimos uno de ellos: `__init__`, que es llamado automáticamente cuando se crea una instancia de la clase.

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

```
del p1
del p2
```

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

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

(continué en la próxima página)

(proviene de la página anterior)

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

```
p1 = Punto(1,1,1)
p2 = Punto()
Punto.total()
del p2
Punto.total()
```

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

```
p1
```

```
<__main__.Punto at 0x7f4d4884b210>
```

```
p2
```

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

<ipython-input-17-32960d173fa8> in <module>
----> 1 p2

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

Como vemos, al borrar (con `del` en este caso) el objeto, automáticamente se actualiza el contador.

7.7.1 Métodos `__str__` y `__repr__`

El método `__str__` es especial, en el sentido en que puede ser utilizado aunque no lo llamemos explícitamente en nuestro código. En particular, es llamado cuando usamos expresiones del tipo `str(objeto)` o automáticamente cuando se utilizan las funciones `format` y `print()`. El objetivo de este método es que sea legible para los usuarios.

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

```
print(p1)
```

```
<__main__.Punto object at 0x7f4d22fc7e50>
```

Rehagamos la clase para definir vectores

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
```

(continué en la próxima página)

(proviene de la página anterior)

```

    "Inicializa un punto en el espacio"
    self.x = x
    self.y = y
    self.z = z
    Punto.num_puntos += 1
    return None

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

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

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

```

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

```
print(p1)
```

```
(x = 1, y = 1, z = 0)
```

```
ss = 'punto en el espacio: {}'.format(p1)
ss
```

```
'punto en el espacio: (x = 1, y = 1, z = 0)'
```

```
p1
```

```
<__main__.Punto at 0x7f4d22f6ea90>
```

Como vemos, si no usamos la función `print()` o `format()` sigue mostrándonos el objeto (que no es muy informativo). Esto puede remediarse agregando el método especial `__repr__`. Este método es el que se llama cuando queremos inspeccionar un objeto. El objetivo de este método es que de información sin ambigüedades.

```

class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

    def __del__(self):
        "Borra el punto y actualiza el contador"

```

(continué en la próxima página)

(proviene de la página anterior)

```

Punto.num_puntos -= 1

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

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

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

```

```
p2 = Punto(0.3, 0.3, 1)
```

```
p2
```

```
Punto(x=0.3, y=0.3, z=1)
```

```
p2.x = 5
p2
```

```
Punto(x=5, y=0.3, z=1)
```

Como vemos ahora tenemos una representación del objeto, que nos da información precisa.

7.7.2 Método `__call__`

Este método, si existe es ejecutado cuando llamamos al objeto. Si no existe, es un error llamar al objeto:

```
p2()
```

```

-----

TypeError                                Traceback (most recent call last)

<ipython-input-31-dd18cc1831f4> in <module>
----> 1 p2()

TypeError: 'Punto' object is not callable

```

```

class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y

```

(continúe en la próxima página)

(proviene de la página anterior)

```

        self.z = z
        Punto.num_puntos += 1
        return None

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

    def __str__(self):
        s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
        return s

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

    def __call__(self):
        return "Ejecuté el objeto: {}".format(self)
#     return str(self)
#     return "{}".format(self)

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

```

```

p3 = Punto(1,3,4)
p3

```

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

```
p3()
```

```
'Ejecuté el objeto: (x = 1, y = 3, z = 4)'
```

7.8 Ejercicios 06 (b)

1. 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):

```

(continúe en la próxima página)

(proviene de la página anterior)

```

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

def __str__(self):
    s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
    return s

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

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

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

```

Complete la implementación de la clase Vector con los métodos pedidos

```

class Vector(Punto):
    "Representa un vector en el espacio"

    def suma(self, v2):
        "Calcula un vector que contiene la suma de dos vectores"
        print("Aún no implementada la suma de dos vectores")
        # código calculando v = suma de self + v2
        # ...

    def producto(self, v2):
        "Calcula el producto interno entre dos vectores"
        print("Aún no implementado el producto interno de dos vectores")
        # código calculando el producto interno pr = v1 . v2

    def abs(self):
        "Devuelve la distancia del punto al origen"
        print("Aún no implementado la norma del vector")
        # código calculando la magnitud del vector

    def angulo_entre_vectores(self, v2):
        "Calcula el ángulo entre dos vectores"
        print("Aún no implementado el ángulo entre dos vectores")
        angulo = 0
        # código calculando angulo = arccos(v1 * v2 / (|v1||v2|))
        return angulo

    def coordenadas_cilindricas(self):
        "Devuelve las coordenadas cilíndricas del vector como una tupla (r, theta, z)"
        print("No implementada")

    def coordenadas_esfericas(self):
        "Devuelve las coordenadas esféricas del vector como una tupla (r, theta, phi)"
        print("No implementada")

```

3. **PARA ENTREGAR:** Cree una clase `Polinomio` para representar polinomios. La clase debe guardar los datos representando todos los coeficientes. El grado del polinomio será *menor o igual a 9* (un dígito).

- **NOTA:** Utilice el archivo `polinomio_06.py` en el directorio `data`, que renombrará de la forma usual

Apellido_06.py. Se le pide que programe:

- Un método de inicialización `__init__` que acepte una lista de coeficientes. Por ejemplo para el polinomio $4x^3 + 3x^2 + 2x + 1$ usaríamos:

```
>>> p = Polinomio([1, 2, 3, 4])
```

- Un método `grado` que devuelva el orden del polinomio

```
>>> p = Polinomio([1, 2, 3, 4])
>>> p.grado()
3
```

- Un método `get_coeficientes`, que devuelva una lista con los coeficientes:

```
>>> p.get_coeficientes()
[1, 2, 3, 4]
```

- Un método `set_coeficientes`, que fije los coeficientes de la lista:

```
>>> p1 = Polinomio()
>>> p1.set_coeficientes([1, 2, 3, 4])
>>> p1.get_coeficientes()
[1, 2, 3, 4]
```

- El método `suma_pol` que le sume otro polinomio y devuelva un polinomio (objeto del mismo tipo)
- El método `mul` que multiplique 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, 0.16666666666666666, 0.08333333333333333, 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
>>>
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> p.eval(x=0.5)
3.25
```

- (Si puede) Un método `from_string` que crea un polinomio desde un string en la forma:

```
>>> p = Polinomio()
>>> p.from_string('x^5 + 3x^3 - 2 x+x^2 + 3 - x')
>>> p.get_coeficientes()
[3, -3, 1, 3, 0, 1]
>>>
>>> p1 = Polinomio()
>>> p1.from_string('y^5 + 3y^3 - 2 y + y^2+3', var='y')
>>> p1.get_coeficientes()
[3, -2, 1, 3, 0, 1]
```

- Escriba un método llamado `__str__`, que devuelva un string (que define cómo se va a imprimir el polinomio). Un ejemplo de salida será:

```
>>> p = Polinomio([1,2.1,3,4])
>>> print(p)
4 x^3 + 3 x^2 + 2.1 x + 1
```

- Escriba un método llamado `__call__`, de manera tal que al llamar al objeto, evalúe el polinomio (use el método `eval` definido anteriormente)

```
>>> p = Polinomio([1,2,3,4])
>>> p(x=2)
49
>>>
>>> p(0.5)
3.25
```

- Escriba un método llamado `__add__` (`self, p`), que evalúe la suma de polinomios usando el método `suma_pol` definido anteriormente. Eso permitirá usar la operación de suma en la forma:

```
>>> p1 = Polinomio([1,2,3,4])
>>> p2 = Polinomio([1,2,3,4])
>>> p1 + p2
```

Ejercicios de Clase 1

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

Ejecute el programa.

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

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

7. Para el número complejo $z = 1 + 0,5i$
- Calcular z^2, z^3, z^4, z^5 .
 - Calcular los complejos conjugados de z, z^2 y z^3 .

Ejercicios de Clase 2

1. Centrado manual de frases

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

2. Para el número complejo $z = 1 - 0,5i$

- Escribir z , z^2 y z^3 en coordenadas polares.
- Escribir un programa, utilizando formato de strings, que escriba las frases:
 - El conjugado de $z=1-0.5j$ es $1+0.5j$
 - El conjugado de $z=(1+0.5j)^2$ es (con el valor correspondiente)

3. Para la cadena de caracteres:

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

- Cuente la cantidad de veces que aparecen los substrings `es`, `la`, `que`, `co`, en los siguientes dos casos: distinguiendo entre mayúsculas y minúsculas, y no distinguiendo. Imprima el resultado.
- Cree una lista, donde cada elemento es una línea del string `s` y encuentre la de mayor longitud. Imprima por pantalla la línea y su longitud.
- Forme un nuevo string de 10 caracteres que contenga los 5 primeros y los 5 últimos del string anterior `s`. Imprima por pantalla el nuevo string.
- Forme un nuevo string que contenga los 10 caracteres centrales de `s` (utilizando un método que pueda aplicarse a otros strings también). Imprima por pantalla el nuevo string.
- Cambie todas las letras `m` por `n` y todas las letras `n` por `m` en `s`. Imprima el resultado por pantalla.

- Debe entregar un programa llamado `02_SuApellido.py` (con su apellido, no la palabra SuApellido). El programa al correrlo con el comando `python3 SuApellido_02.py` debe imprimir:

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

3. Manejos de listas:

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

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

5. Operación rara sobre una lista:

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

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

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

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

- Obtenga una lista **L1** donde cada elemento sea una palabra.
- Cuente la cantidad de palabras en `s1` (utilizando `python`).
- Ordene la lista **L1** en orden alfabético.
- Ordene la lista **L1** tal que las palabras más cortas estén primero.
- Ordene la lista **L1** tal que las palabras más largas estén primero.
- Construya un string `s2` con la lista del resultado del punto anterior.

- Encuentre la palabra más larga y la más corta de la frase.
7. Escriba un script que encuentre las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$. Los valores de los parámetros defínalos en el mismo script, un poco más arriba.
 8. Considere un polígono regular de N lados inscripto en un círculo de radio unidad:
 - Calcule el ángulo interior del polígono regular de N lados (por ejemplo el de un triángulo es 60 grados, de un cuadrado es 90 grados, y de un pentágono es 108 grados). Expresé 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:

$$v = v_0 - gt$$

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

9.1 Adicionales

11. Calcular la suma:

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

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

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

CAPÍTULO 10

Ejercicios de Clase 3

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

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

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

3. 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), ...]
```

4. 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

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

5. 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
/=====
"""
```

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

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

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

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

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

- Calcular la desviación estándar:

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

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

Realizar los cálculos para las listas de números:

```
L1 = [6.41, 1.28, 11.54, 5.13, 8.97, 3.84, 10.26,
      14.1, 12.82, 16.67, 2.56, 17.95, 7.69, 15.39]
L2 = [4.79, 1.59, 2.13, 4.26, 3.72, 1.06, 6.92,
      3.19, 5.32, 2.66, 5.85, 6.39, 0.53]
```

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

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

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


Ejercicios de Clase 4

1. Realice un programa para:

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

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

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

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

Note que la masa sólo debe contener 5 números decimales

2. Escriba funciones para analizar la divisibilidad de enteros:

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

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

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

```

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

```

3. Realice un programa para:

- Leer el archivo **elementos.dat** en el directorio **data** y guardar los datos en un diccionario, cuyas claves serán los símbolos del elemento:

```
elementos.keys() = ['C', 'H', 'O', 'N', 'Na', 'Cl', 'Ca', 'Au']
```

y los valores serán diccionarios tal como se definieron en el ejercicio anterior. Por ejemplo:

```

elementos['H'] = {'S': 'H', 'Z':1, 'A':1, 'M': 1.00782503207, 'P': 0.999885,
↪'MS':1.00794}

```

- Imprimir todos los elementos, en un formato legible (y si le sale: agradable) para personas, ordenados en valores crecientes de masa.

4. PARA ENTREGAR: Adapte los programas realizados en el punto anterior para trabajar con funciones. Se requiere que escriba:

- Una función que lea un archivo (cuyo nombre es el argumento) y devuelva un diccionario donde cada clave es el símbolo del elemento.
- Una función que escriba en un string todos los elementos, ordenados alfabéticamente por clave, en una forma similar a

```

s = """
Elemento: C
Z = 6
A = 12
Masa = 12.0000000
Abundancia = 0.9893
Masa Promedio = 12.0107

Elemento: Ca
Z = 20
A = 40
Masa = 39.96259098
Abundancia = 0.96941
Masa Promedio = 40.078

...
"""

```

Esta función tendrá un argumento requerido que es el diccionario con los elementos y un argumento opcional `reverse` con valor por defecto `False`. Este argumento indica si los elementos se ordenan alfabéticamente de la manera natural (a,b,c,y,z) o inversa (z,y,x, b,a).

- Una función que reciba un nombre de archivo y un string y escriba el string en el archivo dado.
- Finalmente, escriba también el código llamando a las funciones anteriores para realizar el trabajo de lectura y escritura de los elementos en archivos.

Instrucciones de envío: Envíe el código en un archivo llamado `04_SuApellido.py` por correo electrónico con asunto (*subject*) `04_SuApellido`.

CAPÍTULO 12

Ejercicios de Clase 5

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

Al evaluar la función f en un dado valor x debe dar el resultado: $f(x) = A \sin(wx)$ tal que se pueda utilizar de la siguiente manera: “python `f = crear_sen(3, 1.5)` `f(2)` # Debería imprimir el resultado de `3*sin(1.5*2)=0.4233600241796016`”

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

2. Vamos a describir un **sudoku** como un array bidimensional de 9×9 números, cada uno de ellos entre 1 y 4.

Escribir una función que tome como argumento una grilla (Lista bidimensional de ``9x9``) y devuelva verdadero si los números corresponden a la resolución correcta y ``falso`` en caso contrario. Recordamos que para que sea válido debe cumplirse que:

- Los números están entre 1 y 9
- En cada fila no deben repetirse
- En cada columna no deben repetirse
- En todas las regiones de 3×3 que no se solapan, empezando de cualquier esquina, no deben repetirse.

Ejercicios de Clase 6

1. Para la Clase `Vector` dada en clase implemente los métodos `suma`, `producto` y `abs`
 - `suma` debe retornar un objeto del tipo `Vector` y contener en cada componente la suma de las componentes de los dos vectores que toma como argumento.
 - `producto` toma como argumentos dos vectores y retorna un número real
 - `abs` toma como argumentos el propio objeto y retorna un número real
2. Utilizando la definición de la clase `Punto`:

```
class Punto:
    "Clase para describir un punto en el espacio"

    num_puntos = 0

    def __init__(self, x=0, y=0, z=0):
        "Inicializa un punto en el espacio"
        self.x = x
        self.y = y
        self.z = z
        Punto.num_puntos += 1
        return None

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

    def __str__(self):
        s = "(x = {}, y = {}, z = {})".format(self.x, self.y, self.z)
        return s

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

    def __call__(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```

    return self.__str__()

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

```

Complete la implementación de la clase Vector con los métodos pedidos

```

class Vector(Punto):
    "Representa un vector en el espacio"

    def suma(self, v2):
        "Calcula un vector que contiene la suma de dos vectores"
        print("Aún no implementada la suma de dos vectores")
        # código calculando v = suma de self + v2
        # ...

    def producto(self, v2):
        "Calcula el producto interno entre dos vectores"
        print("Aún no implementado el producto interno de dos vectores")
        # código calculando el producto interno pr = v1 . v2

    def abs(self):
        "Devuelve la distancia del punto al origen"
        print("Aún no implementado la norma del vector")
        # código calculando la magnitud del vector

    def angulo_entre_vectores(self, v2):
        "Calcula el ángulo entre dos vectores"
        print("Aún no implementado el ángulo entre dos vectores")
        angulo = 0
        # código calculando angulo = arccos(v1 * v2 / (|v1||v2|))
        return angulo

    def coordenadas_cilindricas(self):
        "Devuelve las coordenadas cilíndricas del vector como una tupla (r, theta, z)"
        print("No implementada")

    def coordenadas_esfericas(self):
        "Devuelve las coordenadas esféricas del vector como una tupla (r, theta, phi)"
        print("No implementada")

```

3. **PARA ENTREGAR:** Cree una clase Polinomio para representar polinomios. La clase debe guardar los datos representando todos los coeficientes. El grado del polinomio será *menor o igual a 9* (un dígito).

NOTA: Utilice el archivo **polinomio_06.py** en el directorio **data**, que renombrará de la forma usual Apellido_06.py. Se le pide que programe:

- Un método de inicialización `__init__` que acepte una lista de coeficientes. Por ejemplo para el polinomio $4x^3 + 3x^2 + 2x + 1$ usaríamos:

```
>>> p = Polinomio([1, 2, 3, 4])
```

- Un método `grado` que devuelva el orden del polinomio


```
>>> p = Polinomio([1,2,3,4])
>>> p.grado()
3
```

- Un método `get_coeficientes`, que devuelva una lista con los coeficientes:

```
>>> p.get_coeficientes()
[1, 2, 3, 4]
```

- Un método `set_coeficientes`, que fije los coeficientes de la lista:

```
>>> p1 = Polinomio()
>>> p1.set_coeficientes([1, 2, 3, 4])
>>> p1.get_coeficientes()
[1, 2, 3, 4]
```

- El método `suma_pol` que le sume otro polinomio y devuelva un polinomio (objeto del mismo tipo)
- El método `mul` que multiplica al polinomio por una constante y devuelve un nuevo polinomio
- Un método, `derivada`, que devuelva la derivada de orden `n` del polinomio (otro polinomio):

```
>>> p1 = p.derivada()
>>> p1.get_coeficientes()
[2, 6, 12]
>>> p2 = p.derivada(n=2)
>>> p2.get_coeficientes()
[6, 24]
```

- Un método que devuelva la integral (antiderivada) del polinomio de orden `n`, con constante de integración `cte` (otro polinomio).

```
>>> p1 = p.integrada()
>>> p1.get_coeficientes()
[0, 1, 1, 1, 1]
>>>
>>> p2 = p.integrada(cte=2)
>>> p2.get_coeficientes()
[2, 1, 1, 1, 1]
>>>
>>> p3 = p.integrada(n=3, cte=1.5)
>>> p3.get_coeficientes()
[1.5, 1.5, 0.75, 0.16666666666666666, 0.08333333333333333, 0.05]
```

- El método `eval`, que evalúe el polinomio en un dado valor de `x`.

```
>>> p = Polinomio([1,2,3,4])
>>> p.eval(x=2)
49
>>>
>>> p.eval(x=0.5)
3.25
```

- (Si puede) Un método `from_string` que crea un polinomio desde un string en la forma:

```
>>> p = Polinomio()
>>> p.from_string('x^5 + 3x^3 - 2 x+x^2 + 3 - x')
>>> p.get_coeficientes()
```

(continué en la próxima página)

(proviene de la página anterior)

```
[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
```

CAPÍTULO 14

Programa completo

- Puede consultar el prog-detalle tentativo