
Clases de Python

Juan Fiol

2023

Dictado de las clases

| | |
|---|-----------|
| 1. Clase 0: Introducción al lenguaje Python orientado a Ingenierías y Física | 3 |
| 2. Clase 1: Introducción al lenguaje | 15 |
| 3. Clase 2: Tipos de datos y control | 33 |
| 4. Clase 3: Tipos complejos y control de flujo | 57 |
| 5. Ejercicios | 83 |
| 6. Información adicional | 89 |
| 7. Material adicional | 99 |

Institución

Instituto Balseiro - Univ. Nac. de Cuyo

Fecha

Febrero a abril de 2023

Docentes

Juan Fiol y Flavio Colavecchia

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

Autor: Juan Fiol

Licencia:

Esta obra está bajo una [Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional](#).

1.1 Python y su uso en ingenierías y ciencias

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

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

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

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

En particular, Python encuentra un nicho de aplicación muy importante en varios aspectos muy distintos del trabajo de ingeniería, científico, o técnico. Por ejemplo, es 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.

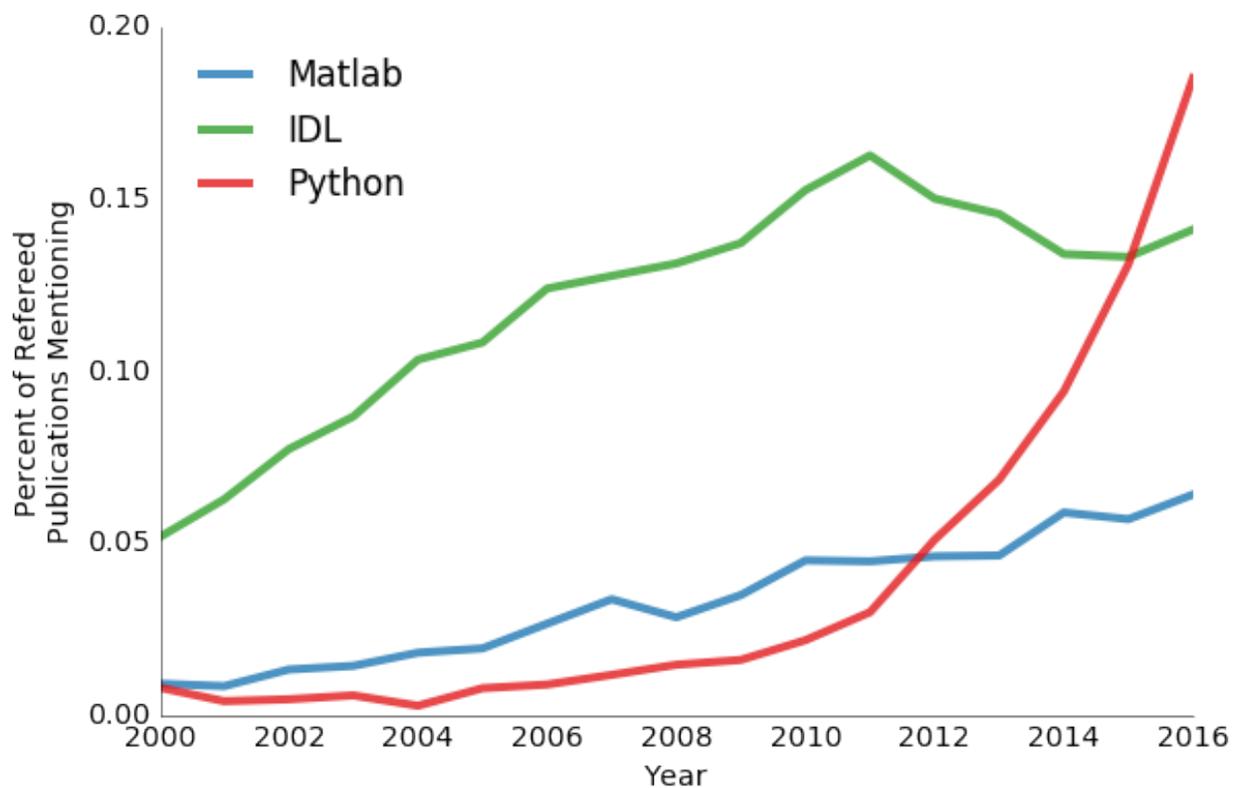
Python es un lenguaje interpretado, como Matlab o IDL, por lo que no necesita ser compilado. Esta característica trae aparejadas ventajas y desventajas. La mayor desventaja es que para algunas aplicaciones –como por ejemplo cálculo numérico intensivo– puede ser considerablemente más lento que los lenguajes 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. Alternativamente, se puede utilizar en forma interactiva, lo que permite modificar y ejecutar líneas específicas de un programa hasta obtener la versión correcta.

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 por C. Beaumont, correspondiente a la evolución hasta 2016 del uso de Python comparado con otros lenguajes/entornos en el ámbito de la Astronomía.

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

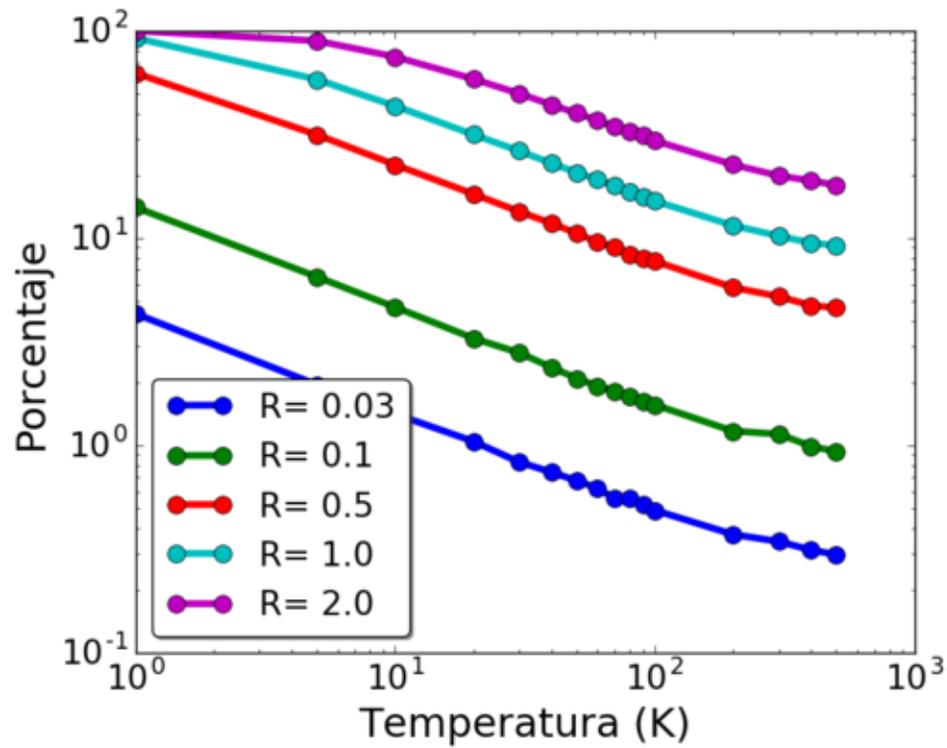


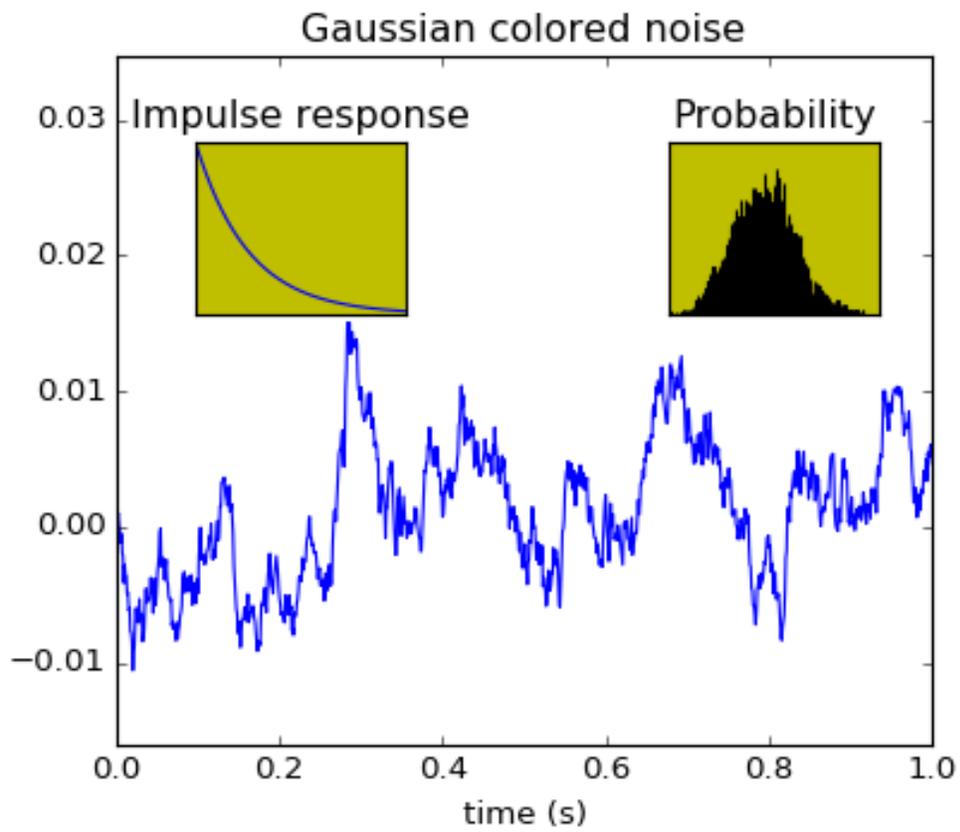
Un último punto que no puede dejar de mencionarse es que Python es libre (y gratis). Esto significa que cada versión nueva puede simplemente descargarse e instalarse sin limitaciones, sin licencias. Además, al estar disponible el código fuente uno podría modificar el lenguaje –una situación que no es muy probable que ocurra– o podría mirar cómo está implementada alguna función –un escenario bastante más probable– para copiar (o tomar inspiración en) alguna funcionalidad que necesitamos en nuestro código.

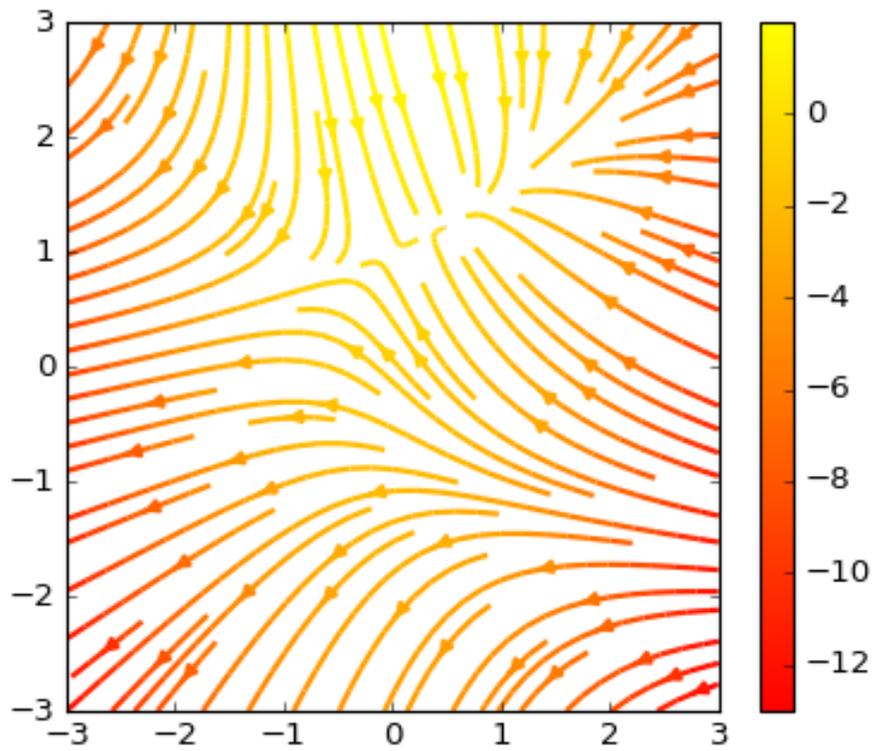
1.2 Visita y excursión a aplicaciones de Python

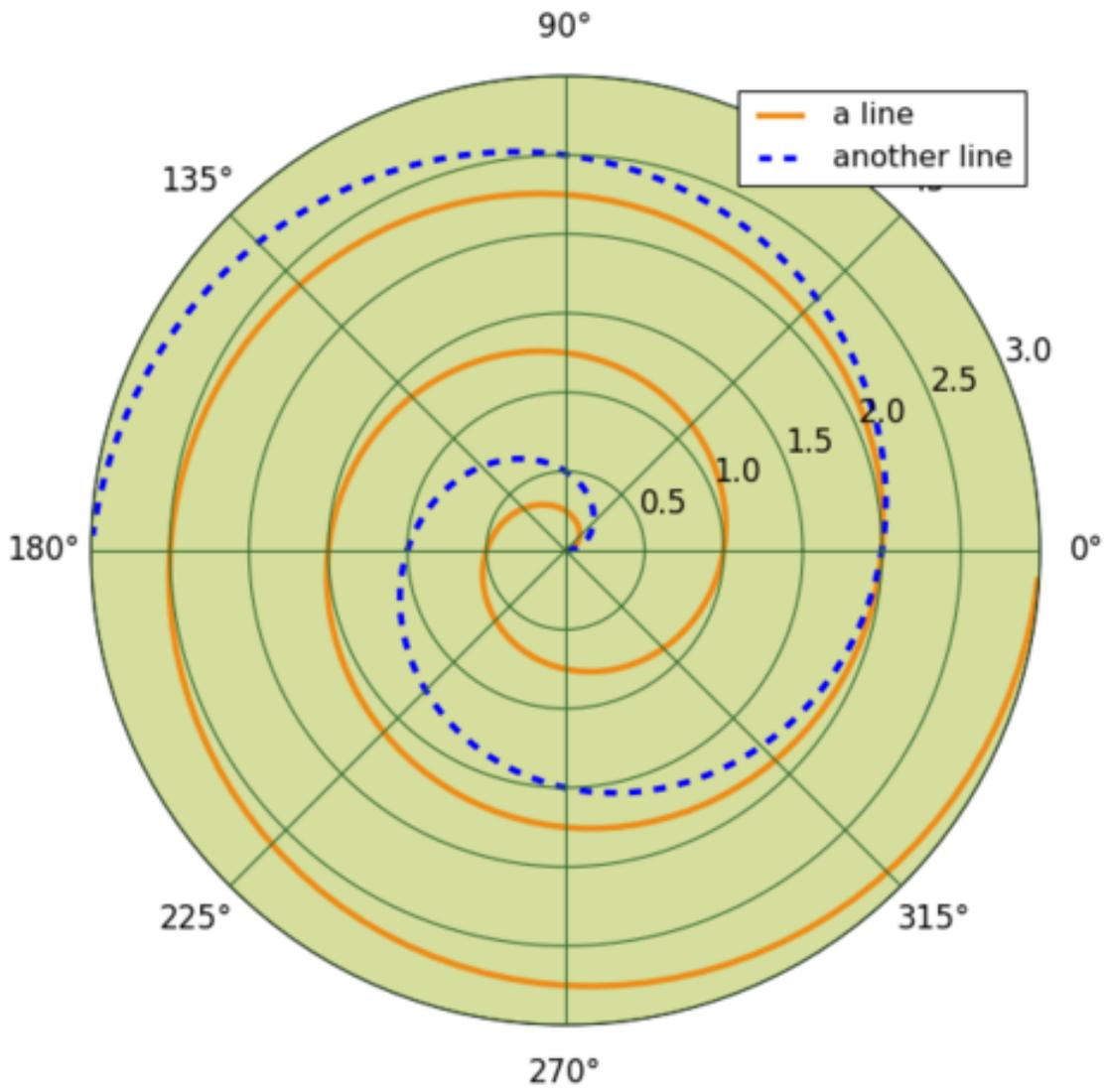
1.2.1 Graficación científica en 2D

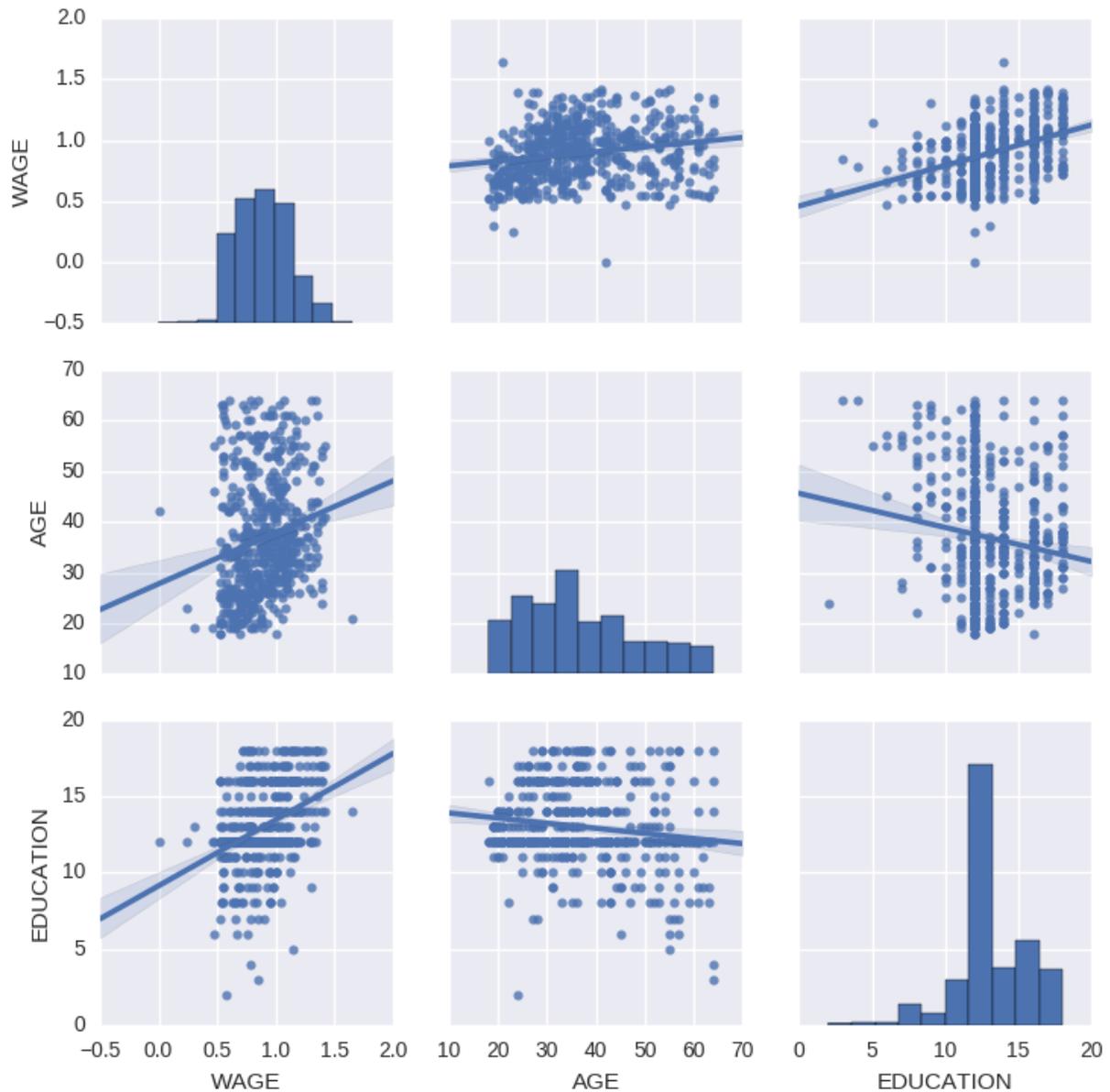
La biblioteca **matplotlib** es una de las mejores opciones para hacer gráficos en 2D, con algunas posibilidades para graficación 3D.









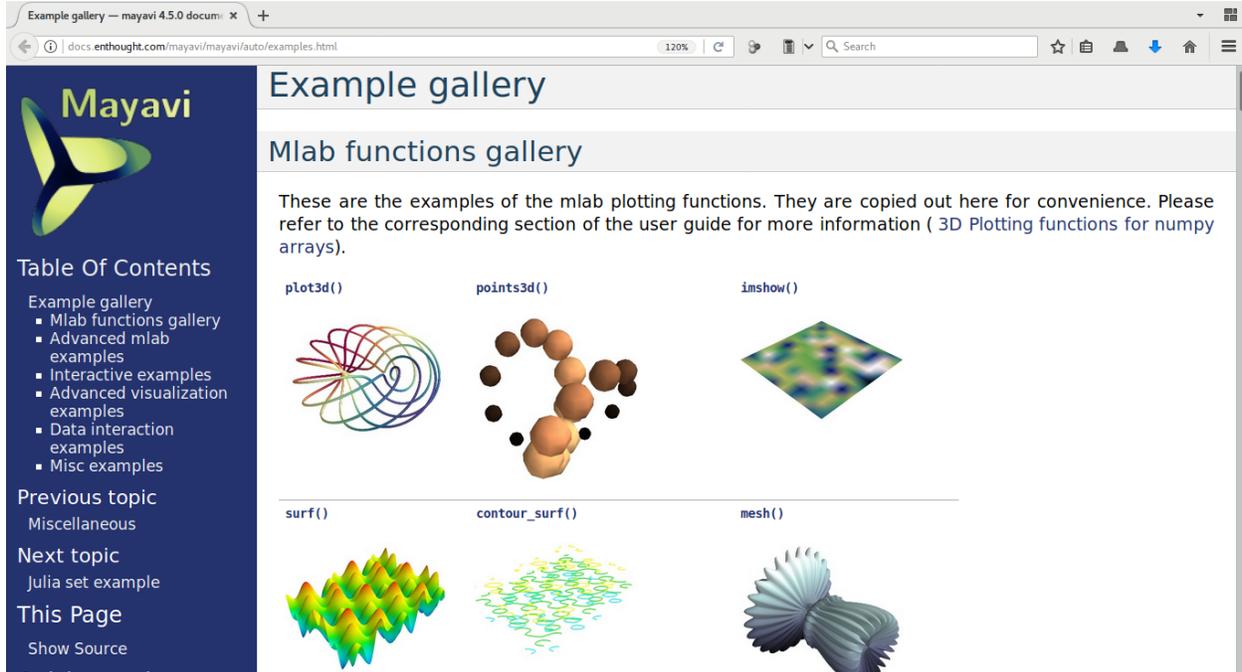


El mejor lugar para un acercamiento es posiblemente la [Galería de matplotlib](#)

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

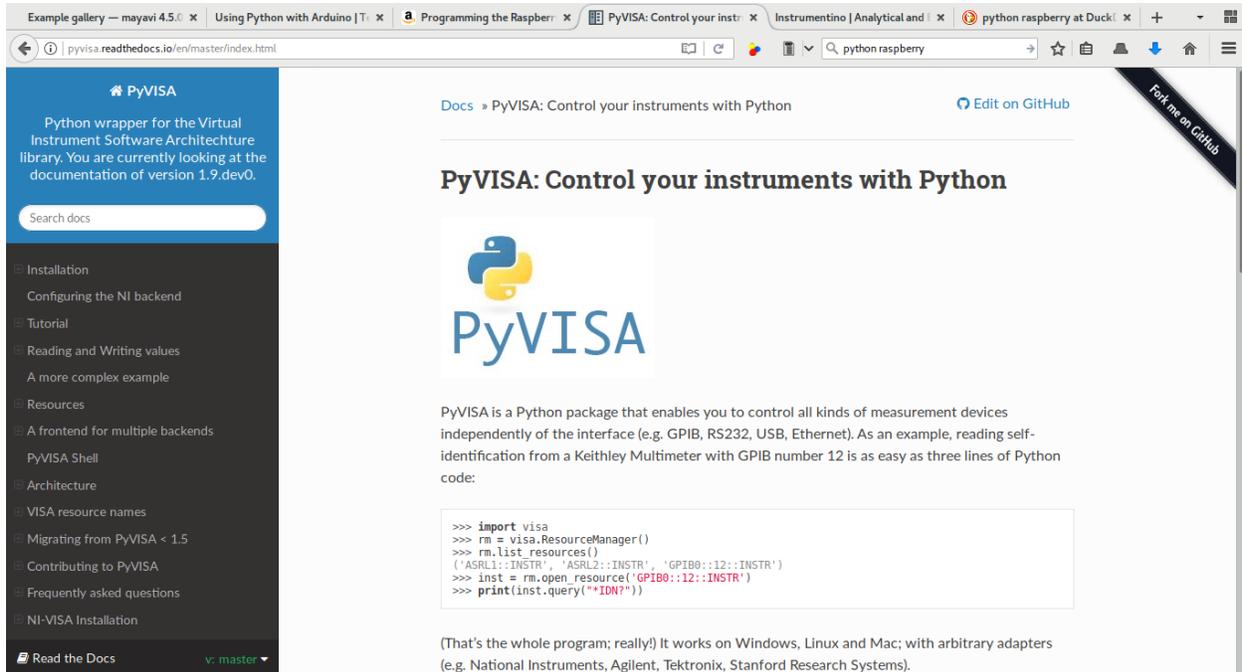
1.2.2 Graficación en 3D

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



1.2.3 Programación de dispositivos e instrumentos

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



Technology Tutorials
at a time. enjoy!

SEARCH
Google Custom Search

ABOUT US ARDUINO LESSONS **USING PYTHON WITH ARDUINO** BEAGLEBONE BLACK RASPBERRY PI WITH LINUX LESSONS 3D PRINTING ENGINEERING CAREER

USING PYTHON WITH ARDUINO

This Circuit combines the simplicity of Arduino with the Power of Python

This series of lessons will teach you how to take your Arduino projects to the next level by having the Arduino interact with the

RECENT POSTS

Sketchup Tutorial LESSON 9: Designing and Printing Box with Movable Hinges
July 29, 2017

Keys to a Successful Engineering Career LESSON 5: How to Interview for a Job
July 28, 2017

Keys to a Successful Engineering Career LESSON 4: How to Write a Good Resume
July 22, 2017

Sketchup Tutorial LESSON 8: Think Like a Mechanical Designer
July 22, 2017

Sketchup Tutorial LESSON 7: Understanding Design Rules Tolerance for Slip and Press Fits
July 22, 2017

RECENT POSTS

Sketchup Tutorial LESSON 9: Designing and Printing Box with Movable Hinges

Keys to a Successful Engineering Career LESSON 5: How to Interview for a Job

Keys to a Successful Engineering Career LESSON 4: How to Write a Good Resume

Sketchup Tutorial LESSON 8: Think Like a Mechanical Designer

Sketchup Tutorial LESSON 7: Understanding Design Rules Tolerance for Slip and Press Fits

1.2.4 Otras aplicaciones

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

1.3 Aplicaciones científicas

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



Figura 1: Herramientas

1.4 Bibliografía

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

1.4.1 Accesible en línea

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

1.4.2 Libros

- The Python Standard Library by Example de Doug Hellman, Addison-Wesley, 2017
- Python Cookbook de David Beazley, Brian K. Jones, OReilly Pub., 2013.
- Elegant Scipy de Harriet Dashnow, Stéfan 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,

1.5 Otras referencias de interés

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

- Videos de Curso para Científicos e Ingenieros

Clase 1: Introducción al lenguaje

2.1 Cómo empezar: Instalación y uso

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

Nota: Hay dos versiones activas del lenguaje Python.

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

2.1.1 Instalación

En este curso utilizaremos **Python 3**

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

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

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

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

o, en Fedora 28, en adelante:

```
`sudo dnf install python3-ipython python3-notebook python3-matplotlib python3-numpy`  
↪python3-scipy`
```

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

2.1.2 Documentación y ayudas

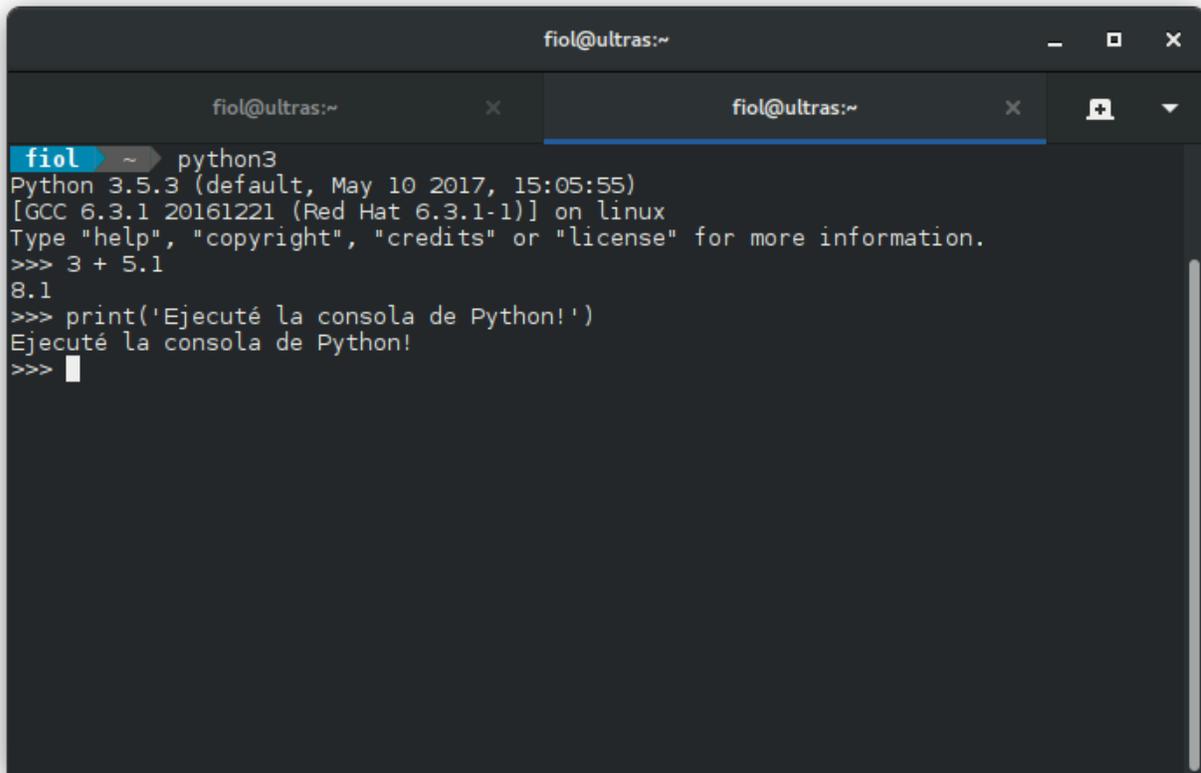
Algunas fuentes de ayuda *constante* son:

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

2.1.3 Uso de Python: Interactivo o no

Interfaces interactivas (consolas/terminales, notebooks)

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

A screenshot of a terminal window titled 'fiol@ultras:~'. The terminal shows a user prompt 'fiol ~' followed by the command 'python3'. The output displays the Python 3.5.3 version information, including the date and time of the installation, and the GCC version. The user then enters the expression '3 + 5.1', which is evaluated to '8.1'. Finally, the user enters a print statement: 'print('Ejecuté la consola de Python!')', which outputs 'Ejecuté la consola de Python!' to the terminal. The prompt '>>>' is visible at the end of the session.

```
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

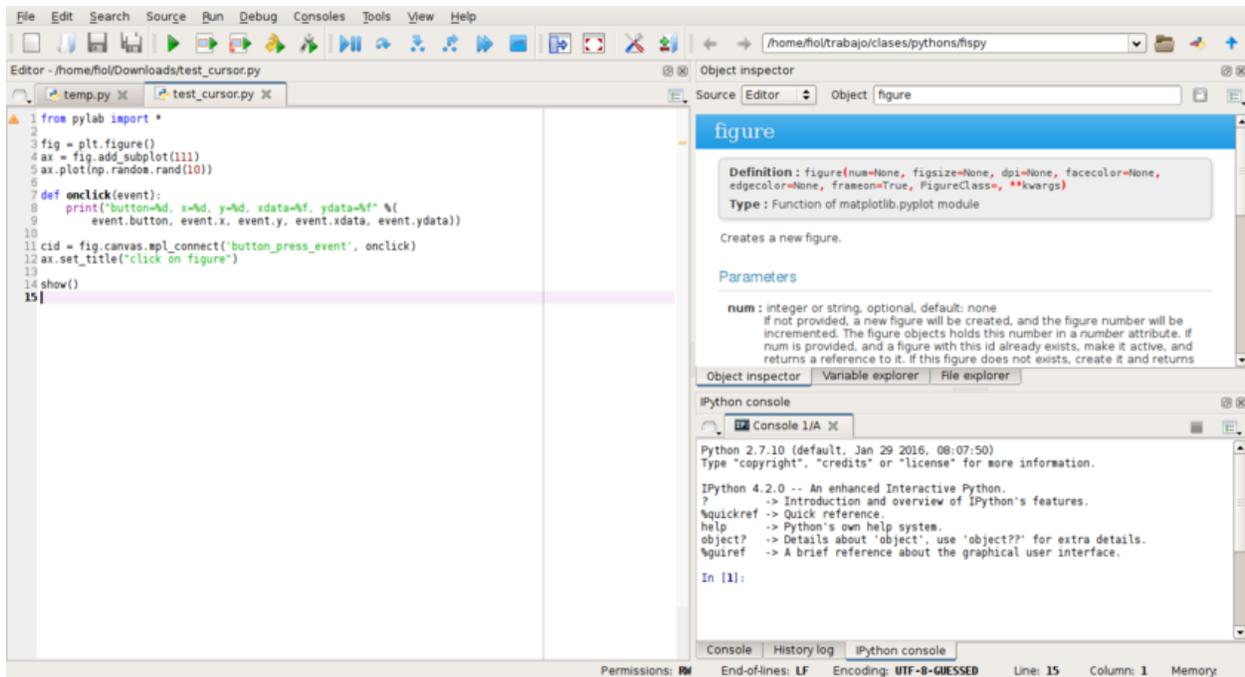
Una forma muy común/poderosa de usar Python es en forma *no interactiva*, escribiendo *programas* o *scripts*. Esto es, escribir nuestro código en un archivo con extensión *.py* para luego ejecutarlo con el intérprete. Por ejemplo, podemos crear un archivo *hello.py* (al que se le llama *módulo*) con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

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

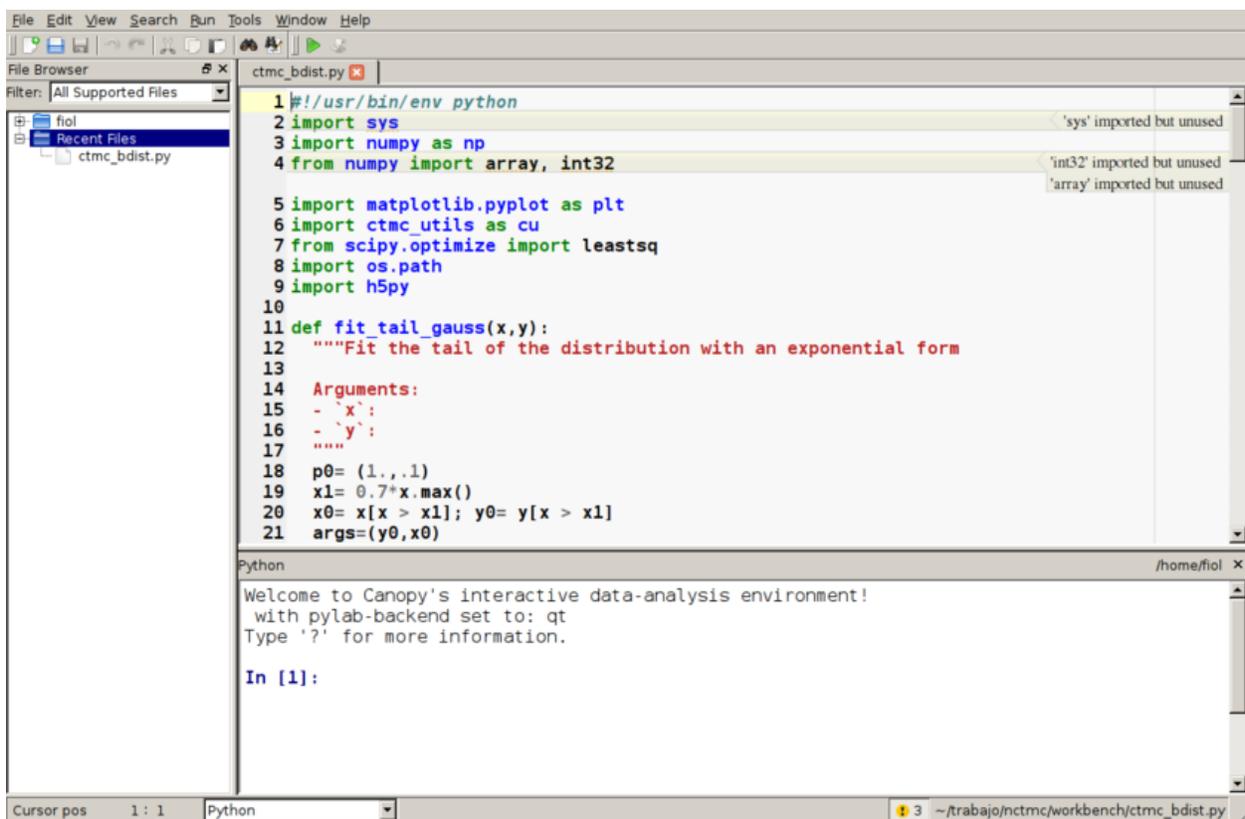
Si ejecutamos `python hello.py` o `ipython hello.py` se ejecutará el intérprete Python y obtendremos el resultado esperado (impresión por pantalla de *Hola Mundo!*, sin las comillas)

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**).

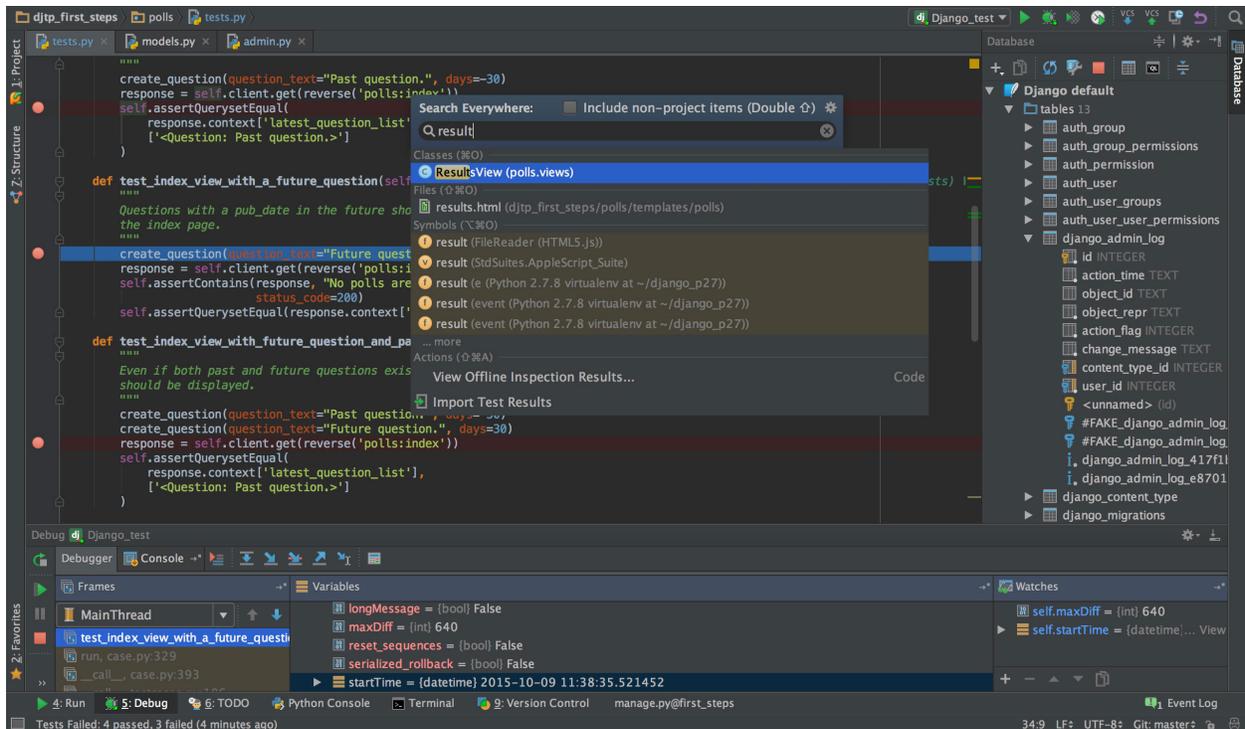
Un buen editor es **Spyder** que tiene características de IDE (entorno integrado: editor + ayuda + consola interactiva).



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



También Pycharm funciona muy bien



En todos los casos se puede ejecutar todo el código del archivo 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 Comandos de Ipython

2.2.1 Comandos de Navegación

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

```
%lsmagic
```

```
Available line magics:
```

```
%alias %alias_magic %autoawait %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 (comandos de navegación)  
↪ %pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint  
↪ %precision %prun %psearch %psource %pushd %pwd %pycat %pylab %qtconsole  
↪ %quickref %recall %rehashx %reload_ext %rep %rerun %reset %reset_selective %rm  
↪ %rmdir %run %save %sc %set_env %store %sx %system %tb %time %timeit  
↪ %unalias %unload_ext %who %who_ls %whos %xdel %xmode
```

(proviene de la página anterior)

Available cell magics:

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

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

2.2.2 Algunos de los comandos mágicos

Algunos de los comandos mágicos más importantes son:

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

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

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

```
%pwd
```

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

```
%cd ~
```

```
/home/fiol
```

```
%pwd
```

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

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

```
In [3]: %pwd
```

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

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

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

Clases de Python

```
%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")
```

```
%run prog1.py
```

```
hola
```

```
%hist
```

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

```
%magic
```

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

```
%quickref
```

2.2.3 Comandos de Shell

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

comandos

```
!echo "1+2" >> prog1.py
```

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

```
%cat prog1.py
```

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

```
%run prog1.py
```

```
hola  
hola otra vez
```

```
!date
```

```
jue 02 feb 2023 16:31:38 -03
```

2.3 Ejercicios 01 (a)

1. Abra una terminal (consola) o notebook y utilícela como una calculadora para realizar las siguientes acciones:
 - Suponiendo que, de las cuatro horas de clases, tomamos dos descansos de 15 minutos cada uno y nos distraemos otros 13 minutos, calcular cuántos minutos efectivos de trabajo tendremos en las 16 clases.
 - Para la cantidad de alumnos presentes en el aula: ¿cuántas horas-persona de trabajo hay involucradas en este curso?
2. Muestre en la consola de Ipython:
 - el nombre de su directorio actual
 - los archivos en su directorio actual
 - Cree un subdirectorio llamado `tmp`
 - si está usando linux, muestre la fecha y hora
 - Borre el subdirectorio `tmp`
3. Para cubos de lados de longitud $L = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
4. Para esferas de radios $r = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
5. Fíjese si alguno de los valores de $x = 2,05, x = 2,11, x = 2,21$ es un cero de la función $f(x) = x^2 + x/4 - 1/2$.

2.4 Conceptos básicos de Python

2.4.1 Características generales del lenguaje

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

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

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

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

2.4.2 Tipos de variables

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

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

Tipos simples: Números

Hay varios tipos de números en Python. Veamos un ejemplo donde definimos y asignamos valor a distintas variables:

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

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

Acá usamos la función `type()` que retorna el tipo de su argumento. Acá ilustramos una de las características salientes de Python: El tipo de variable se define en forma dinámica, al asignarle un valor.

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

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

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

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

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

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

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

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

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

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

Nota: La función `print`

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

```
help(print)
```

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

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

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

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

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

Se trata de un comando. Para escribir las sentencias anteriores en Python 2 sólo debemos omitir los paréntesis y separar la palabra `print` de sus argumentos con un espacio.

Nota: Disgresión: Objetos

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

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

Además, pueden tener:

- componentes
- métodos

Los *métodos* son funciones que pertenecen a un objeto y cuyo primer argumento es el objeto que la posee.

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

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 = (-0.51137600000000003+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
```

```
type(z1)
```

```
complex
```

```
help(z1)
```

Help on `complex` object:

```
class complex(object)
|   complex(real=0, imag=0)
|
|   Create a complex number from a real part and an optional imaginary part.
|
|   This is equivalent to (real + imag*1j) where imag defaults to 0.
|
|   Methods defined here:
|
|   __abs__(self, /)
|       abs(self)
|
|   __add__(self, value, /)
|       Return self+value.
|
|   __bool__(self, /)
|       True if self else False
|
|   __complex__(self, /)
|       Convert this value to exact type complex.
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __format__(self, format_spec, /)
|       Convert to a string according to format_spec.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattr__(self, name, /)
|       Return getattr(self, name).
|
|   __getnewargs__(self, /)
|
|   __gt__(self, value, /)
|       Return self>value.
|
|   __hash__(self, /)
|       Return hash(self).
|
|   __le__(self, value, /)
|       Return self<=value.
|
|   __lt__(self, value, /)
|       Return self<value.
|
|   __mul__(self, value, /)
```

(continué en la próxima página)

```
    Return self*value.

    __ne__(self, value, /)
        Return self!=value.

    __neg__(self, /)
        -self

    __pos__(self, /)
        +self

    __pow__(self, value, mod=None, /)
        Return pow(self, value, mod).

    __radd__(self, value, /)
        Return value+self.

    __repr__(self, /)
        Return repr(self).

    __rmul__(self, value, /)
        Return value*self.

    __rpow__(self, value, mod=None, /)
        Return pow(value, self, mod).

    __rsub__(self, value, /)
        Return value-self.

    __rtruediv__(self, value, /)
        Return value/self.

    __sub__(self, value, /)
        Return self-value.

    __truediv__(self, value, /)
        Return self/value.

    conjugate(self, /)
        Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j.

-----
Static methods defined here:

    __new__(*args, **kwargs) from builtins.type
        Create and return a new object. See help(type) for accurate signature.

-----
Data descriptors defined here:

    imag
        the imaginary part of a complex number
```

(continué en la próxima página)

(proviene de la página anterior)

```
|
| real
|     the real part of a complex number
```

Operaciones

Las operaciones aritméticas básicas son:

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

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

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

Nota: Las operaciones matemáticas están incluidas en el lenguaje.

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

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

```
división de 20/3:          6.666666666666667
parte entera de 20/3:      6
fracción restante de 20/3: 0.666666666666667
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('ft is True?', t == True)
print('ft is False?', t == False)
```

```
ft is True? False
ft is False? True
```

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

```
ft is True? False
<class 'bool'>
```

Hay un tipo *especial*, el elemento None.

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

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

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

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

```
b is a: False
```

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

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

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

Operadores lógicos

Los operadores lógicos en Python son muy explícitos:

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

y a todos los podemos combinar con not, que niega la condición. Veamos algunos ejemplos

```
print ('f20/3 == 6?', 20/3 == 6)
print ('f20//3 == 6?', 20//3 == 6)
print ('f20//3 >= 6?', 20//3 >= 6)
print ('f20//3 > 6?', 20//3 > 6)
```

```
f20/3 == 6? False
f20//3 == 6? True
f20//3 >= 6? True
f20//3 > 6? False
```

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

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

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

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

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

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

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

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

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

```
11 22 False
```

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

Esto sigue valiendo para otros números:

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

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

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

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

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

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

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

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

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

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

2.5 Ejercicios 01 (b)

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

Clase 2: Tipos de datos y control

Nota: Escenas del capítulo anterior:

En la clase anterior preparamos la infraestructura:

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

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

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

En este ejemplo simple, definimos algunas variables con los datos del problema (*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.

Otro punto importante de este ejemplo es el uso de comentarios. El caracter # inicia un comentario de línea, y el intérprete de python ignora todo lo que viene a continuación hasta que encuentra una nueva línea.

3.1 Tipos simples: Números

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

Se mencionó anteriormente que todas las entidades en Python son objetos, que tienen al menos tres atributos: tipo, valor e identidad. Pero además, puede tener otros atributos como datos o métodos. Por ejemplo los números enteros, uno de los tipos más simples que usaremos, tienen métodos que pueden resultar útiles en algunos contextos.

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

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

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

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

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

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

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

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

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

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

```
True
```

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

```
False
```

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

```
c.as_integer_ratio()
```

```
(569, 4)
```

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

```
0x1.1c80000000000p+7
```

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

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

Entonces 0x1.1c8p+7 corresponde a:

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

```
142.25
```

Recordemos, como último ejemplo, los números complejos:

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

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

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

3.2 Tipos compuestos

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

- **Strings**
Se pueden definir con comillas dobles (), comillas simples (), o tres comillas (simples o dobles). Comillas (dobles) y comillas simples producen el mismo resultado. Sólo debe asegurarse que se utiliza el mismo tipo para abrir y para cerrar el *string*
Ejemplo: `s = "abc"` (el elemento `s[0]` tiene el valor "a").
- **Listas**
Las listas son tipos que pueden contener más de un elemento de cualquier tipo. Los tipos de los elementos pueden ser diferentes. Las listas se definen separando los diferentes valores con comas, encerrados entre corchetes. Se puede referir a un elemento por su índice.
Ejemplo: `L = ["a", 1, 0.5 + 1j]` (el elemento `L[0]` es igual al *string* "a").
- **Tuplas**

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

- **Diccionarios**

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

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

3.3 Strings: Secuencias de caracteres

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

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

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

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

```
saludo, saludo2
```

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

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

```
otro= "that's all"
dijo = "'Cómo te va" dijo el murguista a la muchacha'
```

```
print(otro)
```

```
that's all
```

```
print(dijo)
```

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

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

```
print(respondio)
```

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

```
consimbolos = 'pSSE→"'oó@ñ'
```

```
consimbolos
```

```
'pSSE→"'oó@ñ'
```

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

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

Podemos imprimir los strings

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

Hola Mundo

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

that's all

Texto_largo

```
'Aquí me pongo a cantarn Al compás de la vigüela,nQue el hombre que lo desvelan Una
↪pena extraordinarianComo la ave solitarian Con el cantar se consuela.'
```

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

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

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

3.3.1 Operaciones

En **Python** ya hay definidas algunas operaciones que involucran *strings* como la suma (composición o concatenación) y el producto por enteros (repetición).

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


Indexado de *strings*

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

```
s = "0123456789"
print ('Primer caracter  :', s[0])
print ("Segundo caracter :", s[1])
```

```
Primer caracter  : 0
Segundo caracter : 1
```

Si queremos empezar desde el final utilizamos índices negativos. El índice -1 corresponde al último caracter.n

```
print ("El último caracter :", s[-1])
print ("El anteúltimo caracter :", s[-2])
```

```
El último caracter : 9
El anteúltimo caracter : 8
```

También podemos elegir un subconjunto de caracteres:

```
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])
```

```
Los tres primeros:      012
Todos a partir del tercero: 3456789
Los últimos dos:      89
Todos menos los últimos dos: 01234567
```

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

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

```
01289
```

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

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

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

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

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

```
0123456789
024
02468
9876543210
9630
```

Veamos algunas utilidades que se pueden aplicar 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 usando el método de *strings* `replace()`

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

```
La mar astaba sarana!
```

o separar las palabras:

```
print(a.split())
```

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

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

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

Veamos algunos ejemplos más:

```
a = 'Hola Mundo!'
b = "Somos los colectiveros que cumplimos nuestro deber!"
c = Texto_largo
print ('\nPrimer programa en cualquier lenguaje:\n\t\t' + a, 2*'\n')
print (80*'-')
print ('Otro texto:', b, sep='\n\t')
print ('Longitud del texto: ', len(b), 'caracteres')
```

```
Primer programa en cualquier lenguaje:
        Hola Mundo!
```

```
-----
Otro texto:
```

(continué en la próxima página)

(proviene de la página anterior)

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

Buscar y reemplazar cosas en un string:

```
b
```

```
'Somos los colectiveros que cumplimos nuestro deber!'
```

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

```
6
```

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

```
12
```

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

```
12
```

El método `find(sub[, start[, end]])` -> `int` busca el *substring* sub empezando con el índice `start` (argumento opcional) y finalizando en el índice `end` (argumento opcional, que sólo puede aparecer si también aparece `start`). Devuelve el índice donde inicial es *substring*.

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

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

3.3.3 Formato de strings

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

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

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

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

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

Más recientemente se ha implementado en Python una forma más directa de intercalar datos con caracteres literales, mediante *f-strings*, que permite una sintaxis más compacta. Comparemos las dos maneras:

```
pi = 3.141592653589793
s1 = "El valor de es {}".format(pi)
s2 = "El valor de con cuatro decimales es {:.4f}".format(pi)
print(s1)
print(s2)
print("El valor de 2 con seis decimales es {:.6f}".format(2*pi))
print("{:03d}".format(5))
print("{:3d}".format(5))
```

```
El valor de es 3.141592653589793
El valor de con cuatro decimales es 3.1416
El valor de 2 con seis decimales es 6.283185
005
 5
```

```
print(f"el valor de es {pi}")
print(f"El valor de con seis decimales es {pi:.4f}")
print(f"El valor de 2 con seis decimales es {2*pi:.6f}")
print(f"{5:03d}")
print(f"{5:3d}")
```

```
el valor de es 3.141592653589793
El valor de con seis decimales es 3.1416
El valor de 2 con seis decimales es 6.283185
005
 5
```

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).

```
a = 3 # a es entero
b = 3.1 # b es real
c = 3 + 0j # c es complejo
```

(continué en la próxima página)

(proviene de la página anterior)

```
print ("a es de tipo {0}\nb es de tipo {1}\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'>
```

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.

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

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

Input In [53], in <module>
----> 1 print (1+'1')

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

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

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

```
11
2
100001.0
```

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

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

Input In [55], in <module>
      1 # a menos que nosotros **nos equivoquemos explícitamente**
----> 2 print (1 + int('z'))

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

3.5 Ejercicios 02 (a)

1. Centrado manual de frases
 - a. Utilizando la función `len()` centre una frase corta en una pantalla de 80 caracteres. Utilice la frase: Primer ejercicio con caracteres
 - b. Agregue subrayado a la frase anterior
2. **PARA ENTREGAR.** Para la cadena de caracteres:

```
s = '''Aquí me pongo a cantar
Al compás de la vigüela,
Que el hombre que lo desvela
Una pena 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. (Posibles ayudas: busque información sobre funciones que aplican a *strings* y los métodos)
- 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`) por correo electrónico. El programa al correrlo con el comando `python3 02_SuApellido.py` debe imprimir:

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

```
1
```

```
cuadrados[3]
```

```
25
```

```
cuadrados[-1]
```

```
25
```

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

```
[1, 16]
```

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

```
[16, 25]
```

Los índices pueden ser positivos (empezando desde cero) o negativos empezando desde -1.

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

Nota: La asignación entre listas **no copia** todos los datos

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

```
True
```

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

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

```
a is cuadrados
```

```
True
```

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

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

3.6.1 Operaciones sobre listas

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

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

Empecemos concatenando dos listas, usando el operador suma

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

```
L = 2*L1
```

```
L
```

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

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

```
True
```

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

```
3
```

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

```
9
```

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

```
2
```

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

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

```
print(L)
```

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

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

```
print(L)
```

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

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

```
print(L)
```

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

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

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

```
print(L)
```

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

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

```
print(L)
```

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

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

```
print(L)
```

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

En las listas podemos sobrescribir uno o más elementos

```
L[0:3] = [2,3,4]
```

```
print(L)
```

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

```
L[-2:] = [0,1]
```

```
print(L)
```

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

```
L[-2:] = [7, "fin2"]
```

```
print(L)
```

```
[2, 4, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 7, 'fin2']
```

```
print(L)
L.remove(3)           # Remueve la primera aparición del valor 3
print(L)
```

```
[2, 4, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 7, 'fin2']
[2, 4, 2, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 7, 'fin2']
```

3.6.2 Tuplas

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

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

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

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

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

```
Tuples son inmutables
```

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

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

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

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

```
1 3 5
```

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

```
4 5 6
```

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

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

```
4 5
5 4
```

3.6.3 Rangos

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

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

```
range(2)
```

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

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

```
range
```

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

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

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

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

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

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

3.6.4 Comprensión de Listas

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

Podemos crear la lista `cuadrados` utilizando comprensiones de listas

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

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

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

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

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

```
sum(L)
```

```
1540
```

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

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

```
print(L)
```

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

```
L1 = list(reversed(L))
print(L1)
print(L)
```

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

```
L1.reverse()
```

```
print(L1)
```

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

Puede encontrarse más información en la Biblioteca de Python.

3.7 Módulos

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

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

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

Esto nos lleva a la primera cuestión: cómo saber ese nombre, y 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: `módulo.funcion()`.

3.7.1 Módulo `math`

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

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

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

A veces, sólo necesitamos unas pocas funciones de un módulo. Entonces para abreviar la notación 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)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Input In [62], in <module>
      1 import math
----> 2 print(math.sqrt(-1))
```

```
ValueError: math domain error
```

3.7.2 Módulo cmath

El módulo `math` no está diseñado para trabajar con números complejos, para ello existe el módulo `cmath`

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

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

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

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

```
0.4636476090008061
```

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

```
26.56505117707799
```

3.8 Ejercicios 02 (b)

3. Manejos de listas:

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

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

5. Operación rara sobre una lista:

- Defina la lista $L = [0, 1]$
- Realice la operación `L.append(L)`
- Ahora imprima `L`, e imprima el último elemento de `L`.

- Haga que una nueva lista L1 tenga el valor del último elemento de L y repita el inciso anterior.

6. Utilizando funciones y métodos de *strings* en la cadena de caracteres:

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

- Obtenga la cantidad de caracteres.
 - Imprima la frase anterior pero con cada palabra empezando en mayúsculas.
 - Cuente cuantas letras a tiene la frase, ¿cuántas vocales tiene?
 - Imprima el string s1 centrado en una línea de 80 caracteres, rodeado de guiones en la forma:
-En un lugar de la Mancha de cuyo nombre no quiero acordarme-
 - Obtenga una lista **L1** donde cada elemento sea una palabra de la oración.
 - Cuente la cantidad de palabras en s1 (utilizando python).
 - Ordene la lista **L1** en orden alfabético.
 - Ordene la lista **L1** tal que las palabras más cortas estén primero.
 - Ordene la lista **L1** tal que las palabras más largas estén primero.
 - Construya un string **s2** con la lista del resultado del punto anterior.
 - Encuentre la palabra más larga y la más corta de la frase.
7. Escriba un script que encuentre las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$. Los valores de los parámetros defínalos en el mismo script, un poco más arriba.
8. Considere un polígono regular de N lados inscrito 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 inscritos en un círculo de radio unidad?
9. Escriba un *script* (llamado `distancia1.py`) que defina las variables velocidad y posición inicial v_0, z_0 , la aceleración g , y la masa $m = 1$ kg a tiempo $t = 0$, y calcule e imprima la posición y velocidad a un tiempo posterior t . Ejecute el programa para varios valores de posición y velocidad inicial para $t = 2$ segundos. Recuerde que las ecuaciones de movimiento con aceleración constante son:

$$v = v_0 - gt$$
$$z = z_0 + v_0t - gt^2/2.$$

3.8.1 Adicionales

9. Calcular la suma:

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

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

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

Clase 3: Tipos complejos y control de flujo

4.1 Diccionarios

Los diccionarios son colecciones de objetos *no necesariamente homogéneos* que no están ordenados y no se pueden identificar mediante un índice (como L[3] para una lista) sino por un nombre o clave (llamado **key**). Las claves pueden ser cualquier objeto inmutable (cadenas de caracteres, números, tuplas) y los valores pueden ser cualquier tipo de objeto. Las claves no se pueden repetir pero los valores sí.

4.1.1 Creación

En la clase anterior vimos como definir listas

```
t01 = []
t02 = list()
t1 = list(range(1,11))
t2 = [2*i**2 for i in t1]
```

```
print(t01, t02)
print(t1)
print(t2)
```

```
[] []
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 8, 18, 32, 50, 72, 98, 128, 162, 200]
```

Creación de diccionarios

```
d01 = {}
d02 = dict()
d1 = {'S': 'Al', 'Z': 13, 'A': 27, 'M':26.98153863 }
```

(continué en la próxima página)

(proviene de la página anterior)

```
d2 = {'A': 27, 'M':26.98153863, 'S': 'Al', 'Z': 13 }
d3 = dict( [('S','Al'), ('A',27), ('Z',13), ('M',26.98153863)])
d4 = {n: n**2 for n in range(6)}
```

Acá estamos creando diccionarios de diferentes maneras:

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

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

```
{}
{}
```

```
print(d4)
```

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

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

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

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

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

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

```
True
True
```

```
d1 is d2
```

```
False
```

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

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

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

```
True
False
```

4.1.2 Selección de elementos

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

```
d1
```

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

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

```
27
```

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

```
26.98153863
```

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

```
'A1'
```

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

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

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

```
Nombre: Juan
```

```
Diccionario:
```

```
-----
```

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

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

```
8400
```

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

```
entrada
```

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

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

```
entrada
```

```
{'nombre': 'Juan',  
'apellido': 'García',  
'edad': 109,  
'dirección': 'Av Bustillo 9500,',  
'cod': 8400,  
'ciudad': 'Bariloche',  
1: [2, 3]}
```

4.1.3 Acceso a claves y valores

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

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

Keys:

```
['nombre', 'apellido', 'edad', 'dirección', 'cod', 'ciudad', 1]
```

Values:

```
['Juan', 'García', 109, 'Av Bustillo 9500,', 8400, 'Bariloche', [2, 3]]
```

Items:

```
[('nombre', 'Juan'), ('apellido', 'García'), ('edad', 109), ('dirección', 'Av Bustillo_  
→9500,'), ('cod', 8400), ('ciudad', 'Bariloche'), (1, [2, 3])]
```

```
entrada.items()
```

```
dict_items([('nombre', 'Juan'), ('apellido', 'García'), ('edad', 109), ('dirección', 'Av_
↳Bustillo 9500, '), ('cod', 8400), ('ciudad', 'Bariloche'), (1, [2, 3])])
```

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

```
[('nombre', 'Juan'),
 ('apellido', 'García'),
 ('edad', 109),
 ('dirección', 'Av Bustillo 9500, '),
 ('cod', 8400),
 ('ciudad', 'Bariloche'),
 (1, [2, 3])]
```

```
dict(it)
```

```
{'nombre': 'Juan',
 'apellido': 'García',
 'edad': 109,
 'dirección': 'Av Bustillo 9500,',
 'cod': 8400,
 'ciudad': 'Bariloche',
 1: [2, 3]}
```

4.1.4 Modificación o adición de campos

Si queremos modificar un campo o agregar uno nuevo simplemente asignamos un nuevo valor como lo haríamos para una variable.

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

```
entrada
```

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

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

```
1213
1213
```

En el siguiente ejemplo agregamos un nuevo campo indicando el país y modificamos el valor de la ciudad:

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

Datos:

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

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

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

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

entrada

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

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

Nuevo valor:

(continué en la próxima página)

(proviene de la página anterior)

```

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

```

```

# Para borrar un campo de un diccionario usamos `del`
print ("provincia' in entrada =", f"{'provincia' in entrada}")
del entrada['provincia']
print (f"{'provincia' in entrada} = ")

```

```

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

```

```

a = 1
print(f"{'a'}", "es lo mismo que", f"a={a}")

```

```

a=1 es lo mismo que a=1

```

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

```

entrada

```

```

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

```

```

entrada.pop(1)

```

```

[2, 3]

```

```

entrada

```

```

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

```

4.2 Conjuntos

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

```
mamiferos = {'perro', 'gato', 'león', 'perro'}
domesticos = {'perro', 'gato', 'gallina', 'ganso'}
aves = {"chimango", "bandurria", 'gallina', 'cóndor', 'ganso'}
otros = {'casa', 3}
```

```
otros
```

```
{3, 'casa'}
```

```
mamiferos
```

```
{'gato', 'león', 'perro'}
```

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

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

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

4.2.1 Operaciones entre conjuntos

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

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

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

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

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

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

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

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

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

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

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

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

```
domesticos - aves
```

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

4.2.2 Modificar conjuntos

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

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

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

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

```
c
```

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

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

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

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

```
c
```

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

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

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

```
c
```

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

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

```
-----
```

(continué en la próxima página)

(proviene de la página anterior)

```
KeyError                                Traceback (most recent call last)

Input In [157], in <module>
----> 1 c.remove(2)

KeyError: 2
```

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

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

```
c
```

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

4.3 Control de flujo

4.3.1 if/elif/else

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

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

y en Python es muy parecido!

```
if condición_1:
    bloque A
elif 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` (nulo), el número `0` (entero, real o complejo), cualquier secuencia (lista, tupla, string, conjunto o diccionario) 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:
    print ("Aprobó cómodo, felicitaciones!")
elif 6 <= Nota < 8:
    print ("Bueno, al menos aprobó!")
elif 4 <= Nota < 6 :
    print ("Bastante bien, pero no le alcanzó")
else:
    print("Siga participando!")

```

```
Bueno, al menos aprobó!
```

4.3.2 Iteraciones

Sentencia for

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

```

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

```

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

Veamos otro ejemplo, iterando sobre una lista:

```

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

```

```

auto
casa
perro
gato
árbol
lechuza
banana

```

La misma sintaxis se utiliza con otros tipos que se pueden iterar (*strings*, tuplas, conjuntos):

```
conj = set(Lista)
```

```
conj
```

```
{'auto', 'banana', 'casa', 'gato', 'lechuza', 'perro', 'árbol'}
```

```

for c in conj:
    print(c)

```

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

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

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

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

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

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

Nota: Acá utilizamos la función `max()` con un argumento requerido (`Lista`) que es la entidad sobre la que se va a encontrar el mayor valor. Notar que el mayor valor depende de como se defina la comparación entre dos elementos. La función `max()` permite un argumento opcional (`key`) que debe ser una función que se aplicará a cada elemento y luego se compararán los resultados de la aplicación de la función a los elementos. En este caso, a cada palabra se le calcula la longitud y esto es lo que se compara.

Otro ejemplo:

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

```
x=0, suma parcial=0
x=1, suma parcial=1
x=2, suma parcial=3
x=3, suma parcial=6
x=4, suma parcial=10
x=5, suma parcial=15
x=6, suma parcial=21
x=7, suma parcial=28
x=8, suma parcial=36
x=9, suma parcial=45
x=10, suma parcial=55
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
```

Veamos otras características del bloque for.

```
suma = 0
cuadrados = []
for i,elem in enumerate(range(3,30)):
    if elem % 2:          # Si resto (%) es diferente de cero -> Impares
        continue
    suma += elem**2
    cuadrados.append(elem**2)
    print (i, elem, elem**2, suma) # Imprimimos el índice y el elem al cuadrado
print ("sumatoria de números pares al cuadrado entre 3 y 20:", suma)
print ('cuadrados= ', cuadrados)
```

```
1 4 16 16
3 6 36 52
5 8 64 116
7 10 100 216
9 12 144 360
11 14 196 556
13 16 256 812
15 18 324 1136
17 20 400 1536
19 22 484 2020
21 24 576 2596
23 26 676 3272
25 28 784 4056
sumatoria de números pares al cuadrado entre 3 y 20: 4056
cuadrados= [16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784]
```

Puntos a notar:

- Inicializamos una variable entera en cero y una lista vacía
- range(3,30) nos da consecutivamente los números entre 3 y 29 en cada iteración.
- enumerate nos permite iterar sobre algo, agregando un contador automático.

Clases de Python

- 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.']
```

```
# En otros lenguajes...  
for j in range(len(L)):  
    print(f'Índice: {j} -> {L[j]} ({len(L[j])} caracteres)')
```

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

```
for j in range(len(L)):  
    print(f'La palabra "{L[j]}" tiene {len(L[j])} caracteres')
```

```
La palabra "I've" tiene 4 caracteres  
La palabra "had" tiene 3 caracteres  
La palabra "a" tiene 1 caracteres  
La palabra "perfectly" tiene 9 caracteres  
La palabra "wonderful" tiene 9 caracteres  
La palabra "evening." tiene 8 caracteres  
La palabra "But" tiene 3 caracteres  
La palabra "this" tiene 4 caracteres  
La palabra "wasn't" tiene 6 caracteres  
La palabra "it." tiene 3 caracteres
```

En python:

```
for j in L:
    print(f'La palabra "{j}" tiene {len(j)} caracteres')
```

```
La palabra "I've" tiene 4 caracteres
La palabra "had" tiene 3 caracteres
La palabra "a" tiene 1 caracteres
La palabra "perfectly" tiene 9 caracteres
La palabra "wonderful" tiene 9 caracteres
La palabra "evening." tiene 8 caracteres
La palabra "But" tiene 3 caracteres
La palabra "this" tiene 4 caracteres
La palabra "wasn't" tiene 6 caracteres
La palabra "it." tiene 3 caracteres
```

Hay ocasiones en que necesitamos conocer el índice. La solución de otros lenguajes nos lo provee (nos obliga a proveerlo). Python ofrece la función `enumerate()` que agrega un contador automático

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

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

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

```
for n in range(2, 20):
    for x in range(2, n):
        if n % x == 0:
            print( f'{n:2d} = {x} x {n//x}')
            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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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`)

```
list(range(2,2)), list(range(2,3))
```

```
([], [2])
```

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

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

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

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

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

4.4 Ejercicios 03 (a)

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

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

1. Incluyendo todos los valores de k
 2. Incluyendo únicamente los valores pares de k.
3. Calcule la suma

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

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

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

```

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

```

4.5 Técnicas de iteración

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

4.5.1 Iteración sobre elementos de dos listas

Consideremos las listas:

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

Queremos imprimir una lista que combine los dos datos:

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

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

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

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

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

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

```
[(-3.2, 13.2),
 (-2, 12),
```

(continué en la próxima página)

(proviene de la página anterior)

```
(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(f'El día {j} la temperatura mínima fue {t1} y la máxima fue {t2}')
```

```
El día 1 la temperatura mínima fue -3.2 y la máxima fue 13.2
El día 2 la temperatura mínima fue -2 y la máxima fue 12
El día 3 la temperatura mínima fue 0 y la máxima fue 13
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):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
tmedia.append((t1+t2)/2)
print(tmedia)
```

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

También podemos escribirlo en forma más compacta usando comprensiones de listas

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

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

4.5.2 Iteraciones sobre diccionarios

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

```
temps
```

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

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

```

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

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

```
temps.keys()
```

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

```
7 in temps
```

```
True
```

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

```
True
```

```
11 in temps
```

```
False
```

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

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

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

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

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

Si queremos iterar sobre los valores podemos utilizar simplemente:

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

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

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

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

```
l
```

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

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

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

La secuencia anterior puede escribirse en forma más compacta como

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

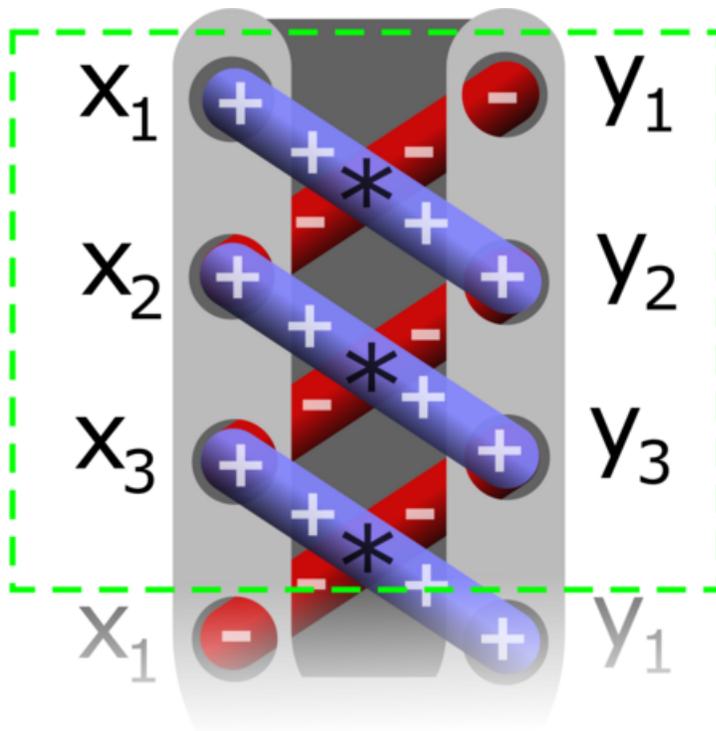
```
9 {'Tmin': 0, 'Tmax': 10}
8 {'Tmin': 4, 'Tmax': 10}
7 {'Tmin': 0, 'Tmax': 14}
6 {'Tmin': -2, 'Tmax': 11}
5 {'Tmin': -5, 'Tmax': 5}
4 {'Tmin': 4, 'Tmax': 18}
3 {'Tmin': -1, 'Tmax': 7}
2 {'Tmin': 0, 'Tmax': 13}
1 {'Tmin': -2, 'Tmax': 12}
0 {'Tmin': -3.2, 'Tmax': 13.2}
```

```
list(temps)
```

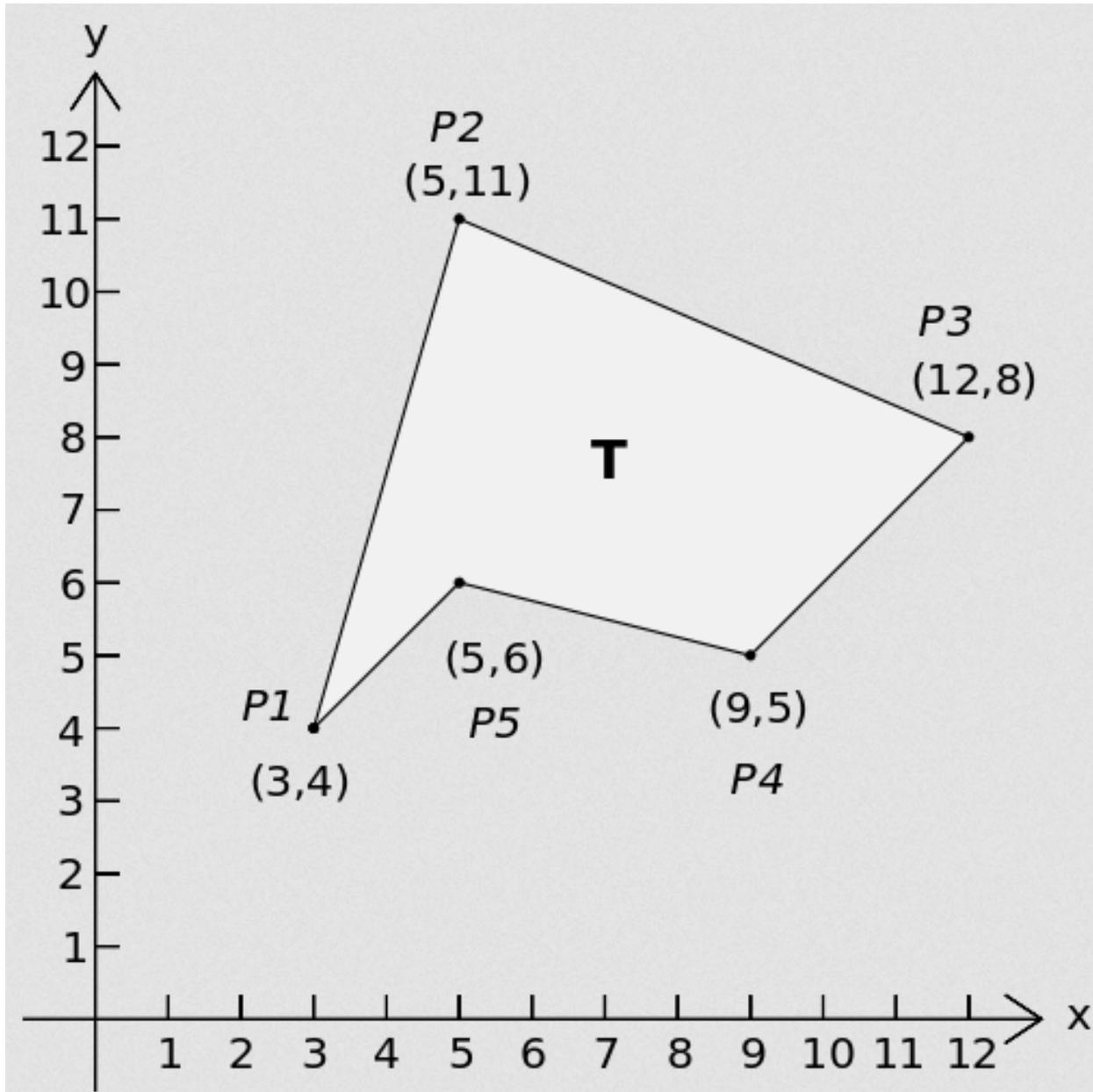
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.6 Ejercicios 03 (b)

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



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



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

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

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

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

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

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

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

$$J_3(2,5) = 0,21660$$

$$J_4(2,5) = 0,07378$$

$$J_7(2,5) = 0,00078$$

$$J_3(5,7) = 0,20228$$

$$J_4(5,7) = 0,38659$$

$$J_7(5,7) = 0,10270$$

$$J_3(10,0) = 0,05838$$

$$J_4(10,0) = -0,21960$$

$$J_7(10,0) = 0,21671$$

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

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

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

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

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

- Calcular la desviación estándar:

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

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

Realizar los cálculos para las listas de números:

L1 = [6.41, 1.28, 11.54, 5.13, 8.97, 3.84, 10.26, 14.1, 12.82, 16.67, 2.56, 17.95, 7.69, 15.39]

L2 = [4.79, 1.59, 2.13, 4.26, 3.72, 1.06, 6.92, 3.19, 5.32, 2.66, 5.85, 6.39, 0.53]

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

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

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

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

5.1 Ejercicios de Clase 01

1. Abra una terminal (consola) o notebook y utilícela como una calculadora para realizar las siguientes acciones:
 - Suponiendo que, de las cuatro horas de clases, tomamos dos descansos de 15 minutos cada uno y nos distraemos otros 13 minutos, calcular cuántos minutos efectivos de trabajo tendremos en las 16 clases.
 - Para la cantidad de alumnos presentes en el aula: ¿cuántas horas-persona de trabajo hay involucradas en este curso?
2. Muestre en la consola de Ipython:
 - el nombre de su directorio actual
 - los archivos en su directorio actual
 - Cree un subdirectorio llamado `tmp`
 - si está usando linux, muestre la fecha y hora
 - Borre el subdirectorio `tmp`
3. Para cubos de lados de longitud $L = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
4. Para esferas de radios $r = 1, 3, 5$ y 8 , calcule su superficie y su volumen.
5. Fíjese si alguno de los valores de $x = 2,05$, $x = 2,11$, $x = 2,21$ es un cero de la función $f(x) = x^2 + x/4 - 1/2$.
6. Para el número complejo $z = 1 + 0,5i$
 - Calcular z^2, z^3, z^4, z^5 .
 - Calcular los complejos conjugados de z, z^2 y z^3 .
 - Escribir un programa, utilizando formato de strings, que escriba las frases:
 - El conjugado de $z=1+0.5i$ es $1-0.5j$
 - El conjugado de $z=(1+0.5i)^2$ es (con el valor correspondiente)

5.2 Ejercicios de Clase 02

1. Centrado manual de frases
 - a. Utilizando la función `len()` centre una frase corta en una pantalla de 80 caracteres. Utilice la frase: Primer ejercicio con caracteres
 - b. Agregue subrayado a la frase anterior
2. **PARA ENTREGAR.** Para la cadena de caracteres:

```
s = '''Aquí me pongo a cantar
Al compás de la vigüela,
Que el hombre que lo desvela
Una pena 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. (Posibles ayudas: busque información sobre funciones que aplican a *strings* y los métodos)
- 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`) por correo electrónico. El programa al correrlo con el comando `python3 02_SuApellido.py` debe imprimir:

```
Nombre Apellido
Clase 2
Distinguiendo: 2 5 1 2
Sin distinguir: 2 5 2 4
Que el hombre que lo desvela : longitud=28
Aquí uela.
desvela
Un
Aquí ne pomgo a camtar
Al compás de la vigüela,
Que el hombre que lo desvela
Uma pema extraordinaria
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 $L = [1, 3, 5, \dots, 17, 19, 19, 17, \dots, 3, 1]$
 5. Operación rara sobre una lista:
 - Defina la lista $L = [0, 1]$
 - Realice la operación $L.append(L)$
 - Ahora imprima L , e imprima el último elemento de L .
 - Haga que una nueva lista $L1$ tenga el valor del último elemento de L y repita el inciso anterior.
 6. Utilizando funciones y métodos de *strings* en la cadena de caracteres:

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

- Obtenga la cantidad de caracteres.
 - Imprima la frase anterior pero con cada palabra empezando en mayúsculas.
 - Cuente cuantas letras a tiene la frase, ¿cuántas vocales tiene?
 - Imprima el string $s1$ centrado en una línea de 80 caracteres, rodeado de guiones en la forma:
-En un lugar de la Mancha de cuyo nombre no quiero acordarme-
 - Obtenga una lista **L1** donde cada elemento sea una palabra de la oración.
 - Cuente la cantidad de palabras en $s1$ (utilizando python).
 - Ordene la lista **L1** en orden alfabético.
 - Ordene la lista **L1** tal que las palabras más cortas estén primero.
 - Ordene la lista **L1** tal que las palabras más largas estén primero.
 - Construya un string **s2** con la lista del resultado del punto anterior.
 - Encuentre la palabra más larga y la más corta de la frase.
7. Escriba un script que encuentre las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$. Los valores de los parámetros defínalos en el mismo script, un poco más arriba.
 8. Considere un polígono regular de N lados inscrito 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 inscritos en un círculo de radio unidad?
 9. Escriba un *script* (llamado `distancial.py`) que defina las variables velocidad y posición inicial v_0, z_0 , la aceleración g , y la masa $m = 1$ kg a tiempo $t = 0$, y calcule e imprima la posición y velocidad a un tiempo posterior t . Ejecute el programa para varios valores de posición y velocidad inicial para $t = 2$ segundos. Recuerde que las ecuaciones de movimiento con aceleración constante son:

$$v = v_0 - gt$$

$$z = z_0 + v_0t - gt^2/2.$$

5.2.1 Adicionales

9. Calcular la suma:

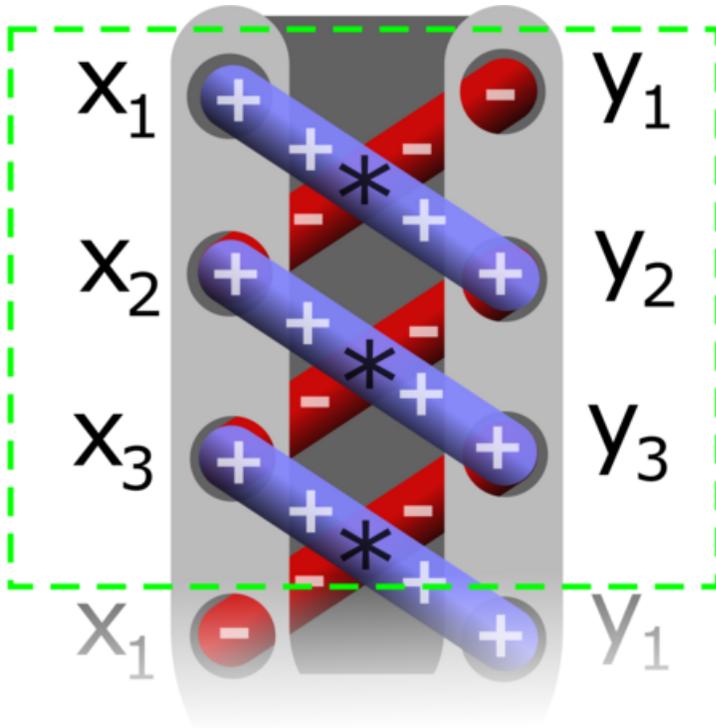
$$s_1 = \frac{1}{2} \left(\sum_{k=0}^{100} k \right)^{-1}$$

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

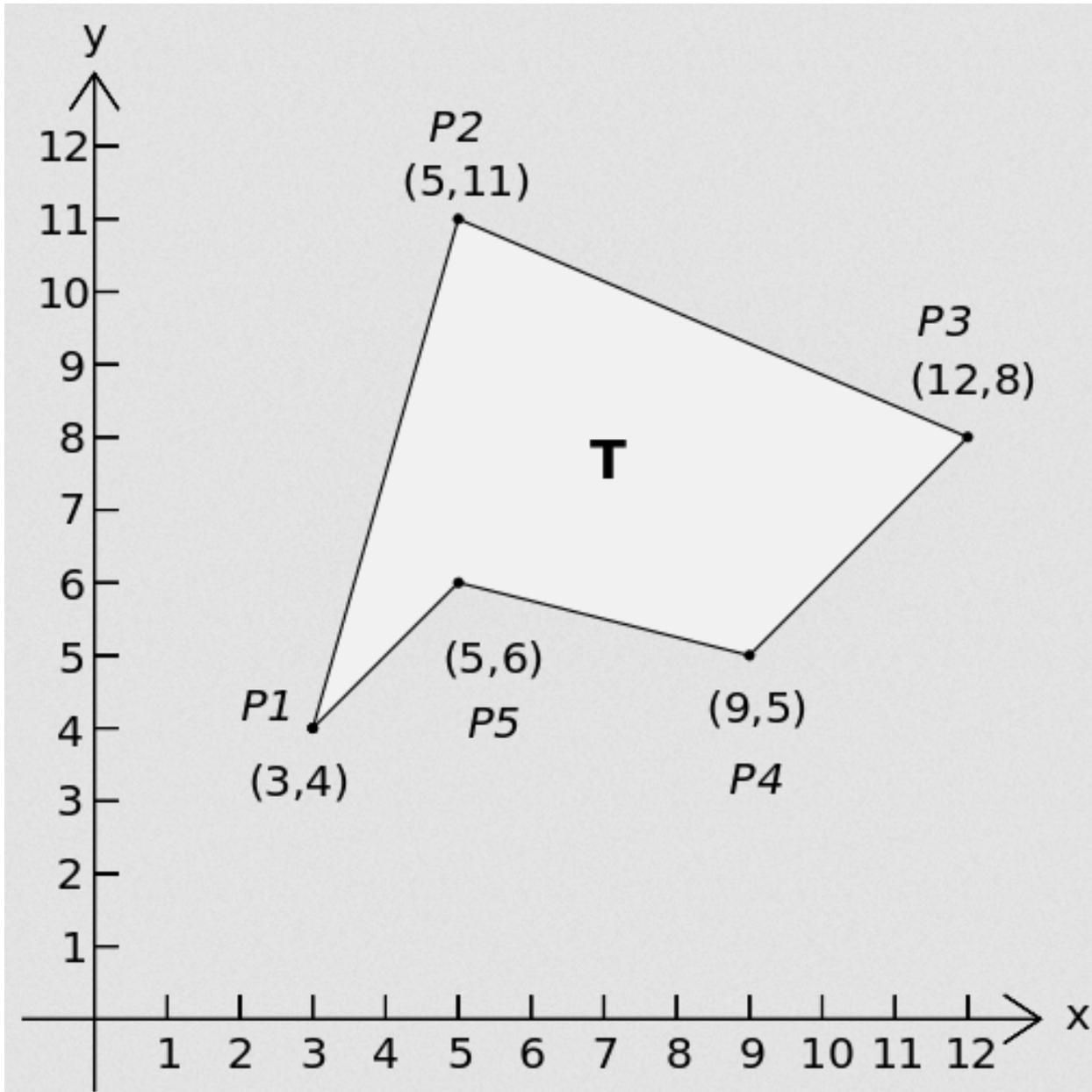
10. Construir una lista L2 con 2000 elementos, todos iguales a `0.0005`. Imprimir su suma utilizando la función `sum` y comparar con el resultado que arroja la función que existe en el módulo `math` para realizar suma de números de punto flotante.

5.3 Ejercicios de Clase 03

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



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



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

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

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

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

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

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

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

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

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

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

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

- Calcular la desviación estándar:

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

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

Realizar los cálculos para las listas de números:

```

L1 = [6.41, 1.28, 11.54, 5.13, 8.97, 3.84, 10.26, 14.1, 12.82, 16.67, 2.56, 17.95,
↪7.69, 15.39]
L2 = [4.79, 1.59, 2.13, 4.26, 3.72, 1.06, 6.92, 3.19, 5.32, 2.66, 5.85, 6.39, 0.53]
    
```

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

```

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

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

```

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

6.1 Programa Detallado

Autor

Juan Fiol

Version

Revision: 2022

Copyright

Libre

6.1.1 Clase 1: Introducción al lenguaje

- Cómo empezar: Instalación y uso
 - Instalación
 - Documentación y ayudas
 - Uso de Python: Interactivo o no
 - Notebooks de Jupyter
- Comandos de Ipython
 - Comandos de Navegación
 - Algunos de los comandos mágicos
 - Comandos de Shell
- Conceptos básicos de Python
 - Características generales del lenguaje
 - Tipos de variables

6.1.2 Clase 2: Tipos de datos y control

- Tipos simples: Números
- Tipos compuestos
- Strings: Secuencias de caracteres
 - Operaciones
 - Iteración y Métodos de Strings
 - Formato de strings
- Conversión de tipos
- Tipos contenedores: Listas
 - Operaciones sobre listas
 - Tuplas
 - Rangos
 - Comprensión de Listas
- Módulos
 - Módulo `math`
 - Módulo `cmath`
 - Adicionales

6.1.3 Clase 3: Tipos complejos y control de flujo

- Diccionarios
 - Creación
 - Selección de elementos
 - Acceso a claves y valores
 - Modificación o adición de campos
- Conjuntos
 - Operaciones entre conjuntos
 - Modificar conjuntos
- Control de flujo
 - `if/elif/else`
 - Iteraciones
- Técnicas de iteración
 - Iteración sobre conjuntos (*set*)
 - Iteración sobre elementos de dos listas
 - Iteraciones sobre diccionarios

6.1.4 Clase 4: Funciones

- Las funciones son objetos
- Definición básica de funciones
- Argumentos de las funciones
 - Ámbito de las variables en los argumentos
 - Funciones con argumentos opcionales
 - Tipos mutables en argumentos opcionales
 - Número variable de argumentos y argumentos *keywords*
- Empacar y desempacar secuencias o diccionarios
- Funciones que devuelven funciones
- Funciones que toman como argumento una función
- Aplicacion 1: Ordenamiento de listas
- Funciones anónimas
- Ejemplo 1: Integración numérica
 - Uso de funciones anónimas
- Ejemplo 2: Polinomio interpolador

6.1.5 Clase 5: Entrada y salida, decoradores, y errores

- Funciones que aceptan y devuelven funciones (Decoradores)
 - Notación para decoradores
 - Algunos usos de decoradores
- Atrapar y administrar errores
 - Administración de excepciones
 - Crear excepciones
- Escritura y lectura a archivos
 - Ejemplos
- Archivos comprimidos

6.1.6 Clase 6: Programación Orientada a Objetos

- Breve introducción a Programación Orientada a Objetos
- Clases y Objetos
 - Métodos especiales
- Herencia
- Atributos de clases y de instancias
- Algunos métodos especiales

- Método `__del__()`
- Métodos `__str__` y `__repr__`
- Método `__call__`
- Métodos `__add__`, `__mul__`

6.1.7 Clase 7: Control de versiones y biblioteca standard

- ¿Qué es y para qué sirve el control de versiones?
 - Cambios en paralelo
 - Historia completa
- Instalación y uso: Una versión breve
 - Instalación
 - Interfaces gráficas
 - Documentación
 - Configuración básica
 - Creación de un nuevo repositorio
 - Clonación de un repositorio existente
 - Ver el estado actual
 - Creación de nuevos archivos y modificación de existentes
 - Actualización de un repositorio remoto
 - Puntos importantes
- Algunos módulos (biblioteca standard)
 - Módulo `sys`
 - Módulo `os`
 - Módulo `subprocess`
 - Módulo `glob`
 - Módulo `pathlib`
 - Módulo `Argparse`
 - Módulo `re`

6.1.8 Clase 8: Introducción a Numpy

- Algunos ejemplos
 - Graficación de datos de archivos
 - Comparación de listas y *arrays*
 - Generación de datos equiespaciados
- Características de *arrays* en **Numpy**
 - Uso de memoria de listas y arrays

- Velocidad de **Numpy**
- Creación de *arrays* en **Numpy**
 - Creación de *Arrays* unidimensionales
 - *Arrays* multidimensionales
 - Otras formas de creación
- Acceso a los elementos
- Propiedades de **Numpy** arrays
 - Propiedades básicas
 - Otras propiedades y métodos de los *array*
- Operaciones sobre arrays
 - Operaciones básicas
 - Comparaciones
 - Funciones definidas en **Numpy**
 - Lectura y escritura de datos a archivos

6.1.9 Clase 9: Visualización

- Interactividad
 - Trabajo con ventanas emergentes
 - Trabajo sobre notebooks
- Gráficos simples
- Formato de las curvas
 - Líneas, símbolos y colores
 - Nombres de ejes y leyendas
- Escalas y límites de graficación (vista)
- Exportar las figuras
- Dos gráficos en la misma figura
- Personalizando el modo de visualización
 - Archivo de configuración
 - Hojas de estilo
 - Modificación de parámetros dentro de programas

6.1.10 Clase 10: Más información sobre Numpy

- Creación y operación sobre **Numpy** arrays
 - Funciones para crear arrays
 - Funciones que actúan sobre arrays
 - Productos entre arrays y productos vectoriales
 - Comparaciones entre arrays
- Atributos de *arrays*
 - reshape
 - transpose
 - min, max
 - argmin, argmax
 - sum, prod, mean, std
 - cumsum, cumprod, trapz
 - nonzero
- Conveniencias con arrays
 - Convertir un array a unidimensional (ravel)
 - Enumerate para ndarrays
 - Vectorización de funciones escalares
- Copias de arrays y vistas
- Indexado avanzado
 - Indexado con secuencias de índices
 - Índices de arrays multidimensionales
 - Indexado con condiciones
 - Función where
- Extensión de las dimensiones (*Broadcasting*)
- Unir (o concatenar) *arrays*
 - Apilamiento vertical
 - Apilamiento horizontal
- Generación de números aleatorios
 - Distribución uniforme
 - Distribución normal (Gaussiana)
 - Histogramas
 - Distribución binomial

6.1.11 Clase 11: Introducción al paquete Scipy

- Una mirada rápida a Scipy
- Funciones especiales
 - Funciones de Bessel
 - Función Error
 - Evaluación de polinomios ortogonales
 - Factorial, permutaciones y combinaciones
- Integración numérica
 - Ejemplo de función fuertemente oscilatoria
 - Funciones de más de una variable
- Álgebra lineal
 - Productos y normas
 - Aplicación a la resolución de sistemas de ecuaciones
 - Descomposición de matrices
 - Autovalores y autovectores
 - Rutinas de resolución de ecuaciones lineales
- Entrada y salida de datos
 - Entrada/salida con *Numpy*
 - Ejemplo de análisis de palabras
 - Entrada y salida en Scipy

6.1.12 Clase 12: Un poco de graficación 3D

- Gráficos y procesamiento sencillo en 2D
 - Histogramas en 2D
 - Gráficos de contornos
 - Superficies y contornos

6.1.13 Clase 13: Interpolación y ajuste de curvas (fiteo)

- Interpolación
 - Interpolación con polinomios
 - Splines
 - B-Splines
 - Lines are guides to the eyes
 - Cantidades derivadas de *splines*
- Interpolación en dos dimensiones

- Interpolación sobre datos no estructurados
- Fiteos de datos
 - Ajuste con polinomios
- Fiteos con funciones arbitrarias
 - Ejemplo: Fiteo de picos

6.1.14 Clase 14: Animaciones e interactividad

- Animaciones con **Matplotlib**
 - Una animación simple en pocos pasos
 - Segundo ejemplo simple: Quiver
 - Tercer ejemplo
- Trabajo simple con imágenes
 - Análisis de la imagen
- Gráficos interactivos (widgets)
 - Cursor
 - Manejo de eventos
 - Ejemplos integrados

6.1.15 Clase 15: Interfaces con otros lenguajes

- Interface con lenguaje C
 - Ejemplo 1: Problema a resolver
 - Interfaces con C
- Interface con lenguaje Fortran
 - Ejemplo 1: Problema a resolver
 - Interfaces con Fortran

6.1.16 Clase 16: Programación funcional con Python

- Los errores al programar
- Los errores en notebooks
- Mutabilidad
- Funciones
 - Funciones puras
 - Funciones de primer orden o primera clase
 - Funciones de orden superior
- Inmutabilidad

- No más loops

CAPÍTULO 7

Material adicional

- Repositorio de fuentes
- Puede descargar las Clases en formato pdf