Lenguaje Python
Para uso en Astronomía y Meteorología

Apuntes de Clase
Borrador

Carlos Feinstein
Cátedra de Computación
Facultad de Ciencias Astronómicas y Geofisicas
UNLP

Versión del 30 de abril de 2024

Índice general

Introducción

Este libro contiene los apuntes de las clases y tiene como fin ser el soporte para consultas de los alumnos de las carreras de Astronomía y Meteorología en la Facultad[1] en el área de la programación en lenguaje Python de la asignatura Computación. Este lenguaje se ha convertido en la actualidad en la base del análisis de datos y su principal ventaja es la facilidad de la interacción en *"Notebooks"* entre el usuario y sus datos. Si bien no es un lenguaje para realizar cálculos pesados ya que suele ser my lento para esta tarea. Pero como gran ventaja el Python se ha convertido en el "frontend" de bibliotecas como las usadas para ajustes de Machine Learning y otras tantas. Su uso en los análisis de datos científicos ya se ha vuelto indiscutible.

En este libro se han utilizado algunos ejemplos tomados del curso de posgrado que dictó el Dr. Christophe Morisset de la IA-UNAM en nuestra Facultad hace varios años. Más información puede encontrarse en su sitio web http://python-astro.blogspot.com/ . Ambos cursos son diferentes. El curso del Dr. Morisset fue pensado como de posgrado (en Python 2) con el fin último de calcular líneas de emisión en nebulosas. Mientras estos apuntes son diseñados para alumnos de grado (en Python 3), por lo cual se encontrarán muchas diferencias de orden y temas entre ambos cursos.

El libro es en sí un complemento detallado de los *notebooks Python* que se utilizan y comentan en las clases teóricas. Enlos distintos capítulos se discutiren las variables, su uso como objetos y estructuras de control del lenguaje Python, así como el uso de las bibliotecas más importantes como Pandas, Numpy, Matplotlib, etc. En el caso especial de la bibliotecas *Astropy*, esta es sólo de interés para los alumnos de astronomía ya que es una colección muy grande de rutinas de uso principalmente en esta ciencia.

En particular, el lenguaje Python evoluciona muy rápidamente así que la información en este apunte podría en algún caso tener una duración probablemente corta en el tiempo y por lo tanto fecha de vencimiento. En este texto haremos uso de la versión de Python 3. Es de esperar que funcione parcialmente en las versiones anteriores de este lenguaje, ya que en el Python no hay compatibilidad con versiones previas (backwards compatibility)

El curso se basa en el uso de python sobre un Notebook del entorno de trabajo Jupyter (aunque veremos que funciona bien en el entorno propietario de Google llamado Colaborative).

Estos apuntes actualmente están en proceso de escritura y por ello llevan claramente el cartel de *BORRADOR*, por lo cual es conveniente que si van a utilizar para estudio o consulta se vea de buscar la útilma la versión. Para distinguir versiones, la portada lleva la fecha de la versión correspondiente. La versión que se encuentra leyendo es del 30 de abril de 2024. Una versión web de estas notas se encuentra en http://astronomia.com.ar/Python/ y es actualizada periódicamente. Esta versión online proviene de los textos presentes y por lo tanto por ahora se considera en preparación e iría cambiando hasta que se llegue a la versión final.

[1] Facultad de Ciencias Astronómicas y Geofísicas - Universidad Nacional de La Plata

Los notebooks usados en los diferentes capítulos pueden obtenerse de astronomia.com.ar/computacion/python.html.

Se agradece por parte del autor la colaboración de los docentes Dres. Andrea Torres, Rodolfo Valverdú y el Lic. Sixto Giménez Benítez por su ayuda en la corrección de la redacción de este apunte.

CAPITULO 1

Introducción a Objetos

1.1. Objetos

En física, con un vector con 3 componentes podemos describir la posición de una partícula en el espacio y con otro vector de 3 componentes podemos describir su velocidad. En programación orientada a objetos, un objeto es algo parecido, pero su estructura de componentes es más compleja. Imaginémonos ahora los datos que tiene la Facultad de cada alumno del Observatorio. Esos datos dan una descripción como si fuese un vector que apunta a cada alumno. Es decir, figura el DNI, la dirección de su casa, una serie de datos personales, etc. Algunos de esos datos, a su vez tiene subestructura, por ejemplo la fecha de nacimiento que se divide en tres: día, mes y año, o datos no numéricos como el nombre y el apellido.

Es decir un objeto es un vector, bastante más complejo de los que estamos acostumbrados del álgebra y de la física. La idea principal es que la programación orientada a objetos conecta los datos, sus propiedades y distintas acciones que puedo hacer con ellos. Veamos la nomenclatura y conceptos:

- Clase: Se llama clase a la definición con la cual se crea el objeto. En nuestro ejemplo, la Universidad decidió qué datos son los que debería saber de sus alumnos, con esos datos creó la clase. Esta es una definición abstracta. Ya que la clase no es el objeto, sino la forma y la definición.

- Objeto: Es cuando se aplica la clase para crear una estructura de datos real, por ejemplo, creo un objeto con la definición de la clase en el que puedo cargar los datos de un alumno en particular. Crear el objeto se denomina como “la instancia".

- Atributos: Son datos que están en el objeto que nos describen características de este. Por ejemplo, si el objeto es un vector, un atributo normalmente, sería su tamaño. Esta característica al programar es una ventaja interesante, ya que no debo recordar por un lado el tamaño del vector, ya que me lo recuerda el atributo y por el otro el código puede escribirse de manera abstracta para cualquier tamaño. Estos atributos existen, porque fueron definidos con la clase con la que se creó el objeto.

- Métodos o Funciones: Los objetos tienen funciones predefinidas que se programaron en la clase. Por ejemplo, podría tener asociado a un vector una función que sume todos sus

elementos y otra función que calcule el módulo del vector. La funciones están definidas en la clase pero sólo se ejecutan con una orden del código del programa, en el momento en que se pide su activación, si no se la requiere no se ejecuta.

- Nomenclatura: Si tengo un vector A que es un objeto en mi programa, podría llamar a los elementos individuales como A[n], donde n sería el número de este elemento. Si el atributo de su tamaño está definido en la clase con el nombre "size", *A.size* me daría el tamaño. Si tuviese la función de la suma de componentes del vector definido en la clase con nombre sum, *A.sum()* calcularía la suma porque estoy llamando a esa función. El atributo siempre tiene un valor ya determinado, la función en un objeto se calcula en el momento que la pido. En Python las funciones además su nombre llevan los paréntesis indicando que es función. Hay excepciones, pero en general es la manera de distinguir entre atributos y funciones, ya que los atributos no llevan los paréntesis. Dentro de estos paréntesis se pueden escribir argumentos para modificar la ejecución de la función. Esta forma de acceder a atributos y funciones, se la llama notación punto o "dot.[en] inglés.

- Nomenclatura 2. Si tengo el objeto alumno, y dentro de este objeto existe como dato el “número de alumno” y se llama *número*, *alumno.número* será esa variable. Es decir que el objeto.algo me permite acceder a esa variable en particular. Siendo incluso tan complejo como lo necesite la estructura. Pudiendo tener, por ejemplo, *alumno.fecha.día*, *alumno.fecha.mes* y *alumno.fecha.año*

CAPITULO 2

Notebooks o cuadernos en Python

2.1. Notebooks

Se puede correr un programa escrito en Python de forma directa. Por ejemplo, si en una terminal escribo el siguiente comando:

python mi_programa.py

Esa orden funcionaría sin problemas. Pero no suele ser la manera de usar el lenguaje Python, al menos cuando se lo usa para análisis de datos. Veremos entonces que hay otras maneras de correr programas en este lenguaje.

La forma más útil de correr Python es de forma interactiva con un sistema donde el programa está dentro de lo que se llama un "Notebook" y veremos en este capítulo sobre las ventajas bastante apreciables de usar esta manera de trabajo. En el curso usaremos formas de Notebooks que provienen de diferentes orígenes, pero hay que indicar que muchas compañías han realizado sus propias versiones. En la sala de computación podremos usar la notebook de la empresa Anaconda (https://www.anaconda.com). Para activar la plataforma Anaconda sobre una terminal se debe correr el comando "conda_init" con el fin de que se seteen las variables de ambiente necesarias para su uso. Y si se quiere desactivar el sistema se puede con el comando "conda deactivate".

Dentro de la plataforma el Notebook de llama "Jupyter" (note la"y" y la falta de tilde). Este nombre proviene de que originalmente este sistema se diseñó no sólo para correr Python, si no también los lenguajes Julia y R. Es decir, que si juntamos **JU**lia, **PY**thon y R (que en inglés sonaría como **ER**) se puede leer JUPYTER y de ahí sale este nombre que no tiene nada que ver con el planeta.

Veamos como funciona este sistema. Con el comando:

Jupyter notebook $\rightarrow$ inicio un notebook nuevo

mientras que si quiero cargar un notebook ya creado y guardado en un archivo en mi directorio, ejecuto:

Jupyter notebook nombre_de_mi_notebook.ipynb $\rightarrow$ Vuelvo a abrir el archivo nombre_de_mi_notebook.ipynb

Cuando corro un notebook Jupyter, este se abre sobre una página web. Y esta página web es la que utilizo para escribir, editar y correr mi programa Python.

Hay que notar que la página web con Notebook es sólo una interface, en realidad mi programa está corriendo en un "Kernel". Dicho kernel es el intérprete que a su vez corre una versión interactiva de Python. Por lo cual en la página web del Notebook verán varios botones para controlar el Kernel. Por ejemplo, apagarlo, reiniciarlo, etc.

En un notebook hay celdas donde se escribe parte del código del programa. En una celda puede haber desde un renglón a varios conteniendo órdenes. El programa entonces es la suma de lo que está escrito en todas las celdas.

El programa se ejecuta corriendo todas las celdas en orden o puede ejecutarse cada celda individualmente. Esta última forma permite la interacción del programador con su código de una manera muy versátil. De esta forma se ahorra tiempo y permite una forma de trabajo muy creativa.

La gran ventaja de usar una página web y un kernel por separado es que se comunican entre ellos por internet (es decir comandos a través del sistema de red TCP/IP) y pueden entonces estar en computadoras diferentes. Por ejemplo, el Notebook y su página web podría estar en una Tablet donde el programador escribe sus órdenes, pero el Kernel podría estar en una supercomputadora a kilómetros de distancia.

Otro Notebook que podremos utilizar fue desarrollado por Google y se utiliza subiendo el código al Drive de Google (https://drive.google.com), y abriéndolo con el sistema llamado "Colaborative". Ver el apunte aparte de su uso. En este caso en particular el Kernel se encontraría en una computadora de Google. Pero es un entorno privado y si bien hay un tiempo gratis de uso este es limitado. Este sistema es pago si se lo quiere utilizar con todas sus funcionalidades, pero para proyectos chicos a medianos funciona muy bien. Si bien utilizaremos el sistema de notebooks Jupyter en este curso, hay muchos otros de estos sistemas tipo notebooks de uso público y privado disponibles.

Los programas escritos en un Notebook, pueden ser guardados. En el lenguaje Python (que no guardaría referencia de donde comienza y termina cada celda) o en un lenguaje propio donde se conservaría la información individual de cada celda. Estos archivos suelen tener extensión ".ipynb". Internamente el archivo está construido en un protocolo llamado "JSON". Este último suele usarse para guardar información de Objetos en forma de archivos.

CAPITULO 3

Variables

3.1. Comentarios - Markdown

En python como cualquier otro lenguaje, existen comandos que permiten poner comentarios que no afectan la corrida del programa pero que son muy útiles en la documentación del código realizado por el programador.

Veamos su uso:

A partir de un "#" todo lo que sigue es un comentario que se acaba cuando termina el renglón. Puede estar en la mitad de un comando.

En cambio tres " producen comentarios que pueden ocupar varios renglones. Tres comillas (3") lo inician al comentario y otras tres comillas lo terminan.

Ejemplo:

" " " Este en cambio es un comentario que ocupa muchas líneas " " " <— A este tipo de comentarios se los denomina docstrings

Por supuesto que estos tipos de comentarios son para celdas de código. Pero si utilizo un **Notebook** para escribir y correr un programa en Python tendré más opciones. En un notebook tenemos dos tipos de celdas. Un tipo para el escribir código y otra para comentarios que pueden llegar a ser muy sofisticados. Este último tipo de celda es para utilizar el lenguaje de texto Markdown. Este es una versión simplificada del **Latex** que permite incluso poner fórmulas utilizando este último lenguaje. Resumiendo tenemos dos tipos de celdas: las de código, donde escribo partes de mi programa y las de comentarios escritas en legunaje Markdown. Esto sólo es válido si desarrollan y corren programas en lenguaje Python en un Notebook.

3.2. Variables en Python y manejo de datos

3.2.1. Variables básicas

Las variables de las sentencias python tienen una diversidad muy grande, están las básicas del python original, pero que se complementan con otras que son agregadas por distintas librerías (numpy, pandas, astropy, etc..). Muchas de estas son de uso muy común en el análisis de datos. Dado esta diversidad, para evitar confusiones existen órdenes para preguntar a una variable de qué tipo es (por ejemplo: type()). Veremos cada una de ellas, empezando por las básicas que

son muy parecidas a las de los otros lenguajes y son las que vimos en el fortran determinadas por el Hardware de las computadoras. Pero siempre hay que tener presente que Python es un lenguaje orientado a objetos, así que las variables son objetos. Y que los objetos pueden ser creados por el usuario, es decir, uno puede crear variables con las propiedades que se necesiten creando una clase que incluya la definición adecuada.

Tipos de datos básicos:

- Enteros (Integers)
- Flotantes (Floats) incluye números complejos
- Textos (Strings)
- Lógicos (Boolean)
- Números complejos

Números

Veamos como asigno un valor a una variable, en este caso un "1" a la variable python "uno". Para probar que esto sucedió imprimiremos el número a continuación. Por lo cual la celda tiene dos comandos, la asignación del valor y la impresión de la variable.

[1]:
```
uno = 1
print("En la variable uno se asignó: ",uno)
```

```
En la variable uno se asignó:  1
```

Noten que no definí el tipo de variable, esto se hace automáticamente y se realiza en la asignación en forma automática. En este caso la variable "uno" se transforma en una variable que sólo contendrá números enteros.

En nuestro caso:

uno = 1

Así sería entero (ojo no es 1. -> no tiene el ".")

Resumiendo: una variable se crea cuando se le asigna un número, un texto, un resultado booleano. Esa característica de lo que se le asigna determina el tipo de variable que es de ahora en adelante. En este caso es un entero, entonces "uno" es una variable que guarda en sí números enteros.

El comando print() imprimió el resultado en la zona próxima, abajo de la celda. Resumiendo, tenemos las celdas con las órdenes y luego fuera de la celda los resultados de la corrida de esos comandos.

¿Qué hago si en mitad de un programa quiero preguntar cuál es el tipo de una variable en particular?

Para eso tengo la orden type .

En este caso es un entero. Hay que prestar atención a la orden print(), en la cual el texto puede ponerse con **"** (o también puede ser el símbolo '), para que se imprima un texto como tal cual es. Las distintas variables se separan con una coma (",") al igual que en Fortran.

[2]:
```
print('Esa variable es del tipo', type(uno))
```

```
Esa variable es del tipo <class 'int'>
```

[3]:
```
# es decir, puedo hacer lo siguente:

uno_f =1.
print(uno,uno_f)

# El 1 flotante es 1.0 (con el . y el 0, mientras que en el número
# entero esto no sucederá)

print('Esa variable es del tipo', type(uno_f))
```

```
1 1.0
Esa variable es del tipo <class 'float'>
```

Existen nombres que son prohibidos, no se permite su uso como variables o funciones, ya que identifican órdenes ya existentes del lenguaje Python. Este es el listado de esos nombres:

and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

True, False, None <– Estas últimas tres son las únicas en mayúsculas.

En los nombres de las variables **no pueden** estar estos símbolos: **!, @, #, $, %**

Mayúsculas y minúsculas son una letra diferente. Por ejemplo, Mag, mag y MAG son 3 variables diferentes.

El nombre de una variable no puede empezar con un número, pero sí tenerlo dentro del nombre. Se permiten usar variables con nombres largos, y a veces es cómodo separar las palabras con el símbolo (_) que puede ser usado como parte del nombre de la variable.

Incluso se pueden incluir caracteres no convencionales, aunque si para usarlos hay que apretar varias teclas a la vez, puede ser muy incómodo y cansador: Ejemplo:

[4]:
```
Que_feo_número_2 = 171717171.71717
print("El número feo es = ",Que_feo_número_2)
```

```
El número feo es =  171717171.71717
```

Sentencias multilínea

Hay que prestar atención a la sangría ("identation" en inglés) y a los símbolos: ( ), [ ] y { }. La sangría en Python significa que la sentencia está relacionada o englobada en la orden anterior. A diferencia de los demás lenguajes de programación su uso indica una acción, no es "inerte". Tiene significado funcional e indica una interacción con órdenes anteriores. Dicho de otra manera, es parte activa de una orden anterior junto con las otras sentencias que comparten el mismo tamaño de sangría. Puede haber dentro de un grupo de órdenes con sangría determinada, otras con un espaciado más largo de sangría. Estas últimas forman otro grupo de órdenes relacionadas en un entorno más interno. Dicho de otra manera una sangría determinada forma un bloque de sentencias, e incluso podría existir un bloque de sentencias más interno con una sangría mayor.

Esto lo veremos con detalle en los capítulos siguientes, pero lo importante que hay que recordar que en Python la sangría es una orden **activa**, no una simple estructura puesta por razones estéticas.

La barra inclinada "\" se puede usar para indicar que la orden sigue en el reglón siguiente. Veamos ejemplos:

[6]:
```
# con el símbolo \ puedo hacer lo siguiente:
a = 1 + 2 + 3 + \
     4 + 5 + 6 \
    + 7 + 8 + 9

# O Compartiendo la misma sangría:
b = (1 + 2 + 3
    + 4 + 5 + 6
    + 7 + 8 + 9)

print("El valor de a=",a,"y el valor de b=",b)
```

```
El valor de a= 45 y el valor de b= 45
```

También se pueden iniciar muchas variables en un sólo renglón.

[7]:
```
a = 1; b = 2; c = 3

x = y = z = "Lo mismo en las tres variables."

print(x,y)
```

```
Lo mismo en las tres variables. Lo mismo en las tres variables.
```

Variables de Caracteres (Strings)

Los strings son cadenas de caracteres de texto, y sirven para manejar justamente textos. Se les pueden aplicar muchos comandos y funciones para realizar distintas modificaciones. Es posible cortar o pegar textos, en la sección de operaciones las veremos con detalle.

[8]:
```
curso = 'Curso2022'

print(curso)
```

```
Curso2022
```

Puedo ir pidiendo las letras que forman la palabra de a una, pero tenemos que tener cuidado sobre cómo se numeran las componentes de una estructura en Python.

Veamos si pido el elemento "0" y NO el "1"

Fijense (muy importante que uso los corchetes y no paréntesis)

[9]:
```
curso[0]  # El primer elemento es el "0" !!!
```

[9]:
```
'C'
```

El primer elemento de un arreglo es el "0" <- ¡Cuidado con esto, es fuente constante de errores!

[10]:
```
# Pero si pido el elemento "1", veamos cuál es:

curso[1]
```

[10]:
```
'u'
```

[11]:
```
# También puedo pedir varios

curso[0:5]
```

[11]:
```
'Curso'
```

Ya que sería string[inicio:final:paso] <– Esta es una forma de notación muy general en Python. Recordar que el índice inicial es el "0" y el final es el n-1. Si el final dice "4" entonces el último elemento es el "3" (ojo que igual hay cuatro elementos porque el primero tiene como índice el valor "0").

Si pido un elemento del string con un número negativo, significa que se empezará a contar los elementos desde el último, sin importar cuántos elementos tenga. Es una forma de acceder a los elementos finales sin tener que averiguar antes cuántos elementos hay en total.

[12]:
```
curso[1:6:2]
```

[12]:
```
'us2'
```

[13]:
```
print('[0] =>', curso[0])
print('[0:1] =>', curso[0:1])
print('[0:2] =>', curso[0:2])
```

```
[0] => C
[0:1] => C
[0:2] => Cu
```

Note que para el uso del comando **print()** de un texto hemos usado indistintamente los caracteres " o '.

[14]:
```
print('[-1] =>', curso[-1])
print('[-2] =>', curso[-2])
print('[-1:-5:-1] =>', curso[-1:-5:-1])
```

```
[-1] => 2
[-2] => 2
[-1:-5:-1] => 2202
```

[15]:
```
# Y algo raro e impensado pero que tiene su lógica. Note que aplico
# la selección directamente al string. No hay variable.

print('[::-1] =>', 'Curso'[::-1])
```

```
[::-1] => osruC
```

Textos En el caso de los textos se permiten una variedad enorme de caracteres, incluso se pueden usar caracteres UNICODE.

[16]:
```
strings = "Esto es Python"
char = "C"
```

[17]:
```
# Imprimo el texto directo

print(strings)
print(char)
print("") # <---renglón en blanco.
```

```
Esto es Python
C
```

[18]:
```
multilinea_str = """Este es un string multilínea con más
de una línea de código."""

# En este caso tiene varios renglones al indicarse la asignación en
# la variable, pero se unen al asignarse. Esta forma de hacerlo
# se llama *docstring* y volveremos más adelante a discutirlo .

print(multilinea_str)
print("")
```

```
Este es un string multilínea con más
de una línea de código.
```

[19]:
```
crudo_str = r"Este está crudo  \n string" # <-- prestar atención al r
                                          # antes del símbolo ".
crudo_str2 = "Este no está crudo  \n string"

# Se puede hacer que el texto se haga literal (sin ejecutar el "\n" que␣
 ↪haría que
# se genere otro renglón). Esto se activa por el "r" antes del texto en la
# asignación.

print(crudo_str)
print(crudo_str2)
```

```
Este está crudo  \n string
Este no está crudo
 string
```

[20]:
```
# Se pueden escribir caracteres UNICODE.

unicode = 'Yo \u2665 la cursada de Computación'

#print(unicode)
```

```
#print("")
# Nota: los cuales no puedo pasar a este apunte porque el Latex no los␣
 ↪reconoce.
# Ver el notebook de la clase.
```

Variables Lógicas (Booleanas)

Sólo pueden guardar un "Verdadero" (True) o un "Falso" (False). Van con la primera letra en mayúscula y sin los puntos que vimos en FORTRAN.

[21]:
```
al = True
bl = False
print(al,bl)
print(type(al),type(bl))
```

```
True False
<class 'bool'> <class 'bool'>
```

Si bien el resultado es True o False, desde el punto de vista numérico hay un "1" en caso de una variable verdadera y un "0" en caso de que sea falso el valor guardado. Entonces me pueden servir para hacer cuentas. Por ejemplo, si multiplico una variable con un número por una booleana, en esta última habrá un 1 si es verdadera (y por lo tanto multiplicará por ese 1) o un 0 si es falsa.

Veamos esto:

[22]:
```
print('Uso a1 y b1 de la celda anterior')
print('Si multiplico un número por un booleano y era verdadero:', al*8)
print('Si multiplico un número por un booleano y era falso    :' ,bl*8)
```

```
Uso a1 y b1 de la celda anterior
Si multiplico un número por un booleano y era verdadero: 8
Si multiplico un número por un booleano y era falso    : 0
```

Convirtiendo tipos de Variables

USO las siguientes funciones, por eso llevan ()

- float() –> convierte a flotantes
- int() –> convierte a enteros
- str() –> convierte a strings

[23]:
```
a = True
print(a)
print(float(a))
print(int(a))
print(str(a))
print(int(13.97))
```

```
True
1.0
1
```

```
True
13
```

3.3. Operaciones con las variables

3.3.1. Operaciones matemáticas básicas

- Son iguales a FORTRAN
- Asignan los resultados de la misma manera a una tercera variable
- Hay operaciones no númericas
- Están las muy básicas y se usan exactamente igual que en Fortran, salvo una: +,-,*,/,**

- Y otras no tan convencionales, por ejemplo: //,%
- Existen asignaciones incrementales (o decrementales) sobre la misma variable (aditivas, multiplicativas)

[24]:
```
a = 2
b = 2
c = a + b
d = a - b
e = a * b
f = a ** b
g = a / b

print(c, 'es', a, '+', b)
print(d, 'es', a, '-', b)
print(e, 'es', a, '*', b)
print(f, 'es', a, '**', b)
print(g, 'es', a, '/', b, '¿Qué notan aquí?')
```

```
4 es 2 + 2
0 es 2 - 2
4 es 2 * 2
4 es 2 ** 2
1.0 es 2 / 2 ¿Qué notan aquí?
```

¡Las divisiones entre enteros generan un número real!

Esto es diferente a Fortran e incluso a las versiones anteriores de Python, incluyendo Python versión 2. Esta situación generó un quiebre muy fuerte entre las versiones de Python 2 y 3. Provocando que mucha gente conserve simultáneamente las dos versiones por la incompatibilidad del software.

[25]:
```
# ejemplo

124231/1134
```

[25]:
```
109.55114638447972
```

3.3.2. Asignaciones múltiples

Se pueden hacer asignaciones múltiples Es decir, que en una sentencia se asignan varios resultados.

El orden es determinante para entender a qué variable se le asigna cada dato.

[26]:
```
x,y = 25.4, 54.67788
print(x,y)

z1,z2= x*24.5,y*4567
print(z1,z2)
```

```
25.4 54.67788
622.3 249713.87796
```

3.3.3. Operaciones no convencionales

Veamos algunas de las operaciones no convecionales, pero que en sí son muy útiles: Por ejemplo, la **floor division** da como resultado la parte entera de una división. Para esta operación se usa el símbolo: //

[27]:
```
17//3
```

[27]: 5

Pero por otro lado tenemos la operación que nos da como resultado el resto de la división. El equivalente a la función módulo en Fortran. Se usa con el símbolo: %

[28]:
```
17%3
```

[28]: 2

Las operaciones incrementales son válidas en Python, pero ojo, no lo son las estilo C tipo i++.

[29]:
```
i=1

i+=10  #suma 10 a i  -> i=i+10
print(i)

i-=2  #resto 2 a la variable i -> i=i-2
print(i)

i*=4  #multiplico por 4 a i -> i=i*4
print(i)

i/=5  #divido por 5 a i -> i=i/5
print(i)
```

```
11
9
```

```
36
7.2
```

3.3.4. Operador Walrus

El operador walrus (en español es morsa), tiene un símbolo para su uso adecuado al animal que lleva su nombre **:=** .

Sirve para realizar una asignación dentro un cálculo. Es decir, puedo cargar un valor en una variable al momento de hacer un cálculo.

Ejemplo:

[30]:
```
a = 5
c = (b:= 3) + a
print(c)
```

```
8
```

Pueden ver que a la variable b se le asignó el valor 3 mientras se hacía la cuenta, que termina asignando el valor final a c.

3.3.5. Precisión de los cálculos

Es diferente a Fortran, los reales son siempre doble precisión (real*8) y los enteros tienen precisión arbitraria. Este término "precisión arbitraria" se utiliza para indicar que los enteros pueden utilizar todos los Bytes que sean necesarios y el largo del número puede ser considerable.

En contraposición recordemos que las variables en "Precisión doble" dependen del hardware pero en general los números válidos están entre 10^-308 y 10^308, con 16 dígitos significativos.

[31]:
```
numero=3242525225
z = numero**8
print(z)
```

```
1221987903028630696386077055895297398394480399555538550758197001968383789062 5
```

[32]:
```
a=1
b=100000000000000000000
print(a/b)
```

```
1e-21
```

Tengo la función de Python *type()* con la que puedo preguntar de qué tipo son las variables y puede ser muy útil en muchas ocasiones.

[33]:
```
print(type(2))
print(type(2.3))
```

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

Además puedo cortar (truncar) y redondear los números.

[34]:
```
print(int(0.8)) # truncando
print(round(0.88766)) # ahora el más cercano pero que sea entero.
```

```
0
1
```

Entendiendo el redondeo, en este caso se busca el valor par más cercano. Por ejemplo el valor 2.5 podría aproximarse a 2 o a 3, pero en este caso se prefiere el par entonces 2.5 sin decimales se convierte en 2 que es par y no en 3.

[35]:
```
print(round(0.3))
print(round(2.5))
```

```
0
2
```

3.4. Números complejos

Hay que recordar el concepto de objetos, ahora lo veremos en el caso de números complejos, pero esto se extiende a todos los tipos de variables. En Python la parte imaginaria se indica con la letra j, no con i como es la convención normal que se usa en matemática.

Por lo cual un número complejo sería el siguiente:

[36]:
```
a = 1.5 + 0.5j
```

Puedo hacer cuentas conservando las propiedades de los números complejos:

[37]:
```
a**2
```

[37]:
```
(2+1.5j)
```

Pero también puedo preguntar por el valor de la parte real e imaginaria por separado.

Ya que estos son atributos del *Objeto* número complejo

[38]:
```
a.real
```

[38]:
```
1.5
```

[39]:
```
a.imag
```

[39]:
```
0.5
```

Pero como también es un *Objeto* hay funciones asociadas dentro de este. Hay que recordar que a diferencia de los atributos es que ya están anotados, las funciones se calculan en el momento de ser llamadas.

[40]:
```
a.conjugate() # En este caso la función requiere los ()
```

[40]:
```
(1.5-0.5j)
```

[41]:
```
a*a.conjugate()
```

[41]:
```
(2.5+0j)
```

3.5. Operaciones Booleanas

[42]:
```
5 < 7
```

[42]:
```
True
```

[43]:
```
a = 4
b = 9
```

[44]:
```
b < a  #  ¿Es cierto?
```

[44]:
```
False
```

[45]:
```
c = 1
```

[46]:
```
c < a < b        # Sería ver si 1 < 4 < 9 es cierto
```

[46]:
```
True
```

Los operadores booleanos (o sea las operaciones lógicas) son **and, or, not** pero hay cosas como "is" (es un "and" literal)

[47]:
```
a < b and b < c
```

[47]:
```
False
```

[48]:
```
resultado = a < 8
print(resultado, type(resultado))
```

```
True <class 'bool'>
```

[49]:
```
print(resultado)
print(not resultado)
```

```
True
False
```

[50]:
```
not resultado is True
```

[50]:
```
False
```

También tenemos los comandos **is, is not**. El comando **is** también sirve para preguntar si el objeto es el mismo.

[51]:
```
a = 7
b = 7

print(a==b)
print(a is b)
```

```
True
True
```

[52]:
```
a = 22.4
resultado=22.4

print(resultado is a)
print(resultado==a)
```

```
False
True
```

3.6. Métodos de los Textos/Strings

[53]:
```
a = "Esto es un    string"
```

En Python tengo una cantidad muy importante de funciones intrínsecas, por ejemplo, puedo preguntar su tamaño con la función **len(a)**.

[54]:
```
len(a)
```

[54]: 20

Pero también la variable **a** tal como la hemos creado es ahora un **Objeto** de la **Clase** string (textos), por lo cuál tengo métodos o funciones que están asociadas al objeto. Puedo preguntar cuáles son las funciones o métodos de esa clase con la orden **dir()**

[55]:
```
print(dir(a))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',␣
 ↪'translate',
'upper', 'zfill']
```

Resumiendo: los strings son objetos (como todo en python). Por lo cual hay métodos (funciones) que están definidas en los objetos y las puedo usar en el momento que las necesite. Y como vimos son una cantidad considerable.

Tenemos dos strings B y C ¿Qué pasa si hacemos la operación B+C?

[56]:
```
B = "Primera Parte "
C = "Segunda Parte "

print(B+C)
```

```
Primera Parte Segunda Parte
```

Como se puede ver ambos strings se concatenan uno detrás del otro siguiendo el orden de la operación.

¿Y si hacemos D = B*2 ?

[57]:
```
D = B*2

print(D)
```

```
Primera Parte Primera Parte
```

Como generalización de la suma que vimos antes, se concatenan el mismo string la cantidad de veces que lo multiplico.

Métodos del objeto strings

Veamos algunas de las funciones que tengo en la clase "string" con más detalle:

[58]:
```
a.upper()
```

[58]:
```
'ESTO ES UN    STRING'
```

[59]:
```
a.title()
```

[59]:
```
'Esto Es Un    String'
```

[60]:
```
a.split()
```

[60]:
```
['Esto', 'es', 'un', 'string']
```

Esto que se generó se llama **lista**, aunque parece un vector, no lo es exactamente y veremos las "listas" en la clase que viene.

Veremos que las listas tienen elementos y estos están numerados. El primer elemento está numerado como "0".

[61]:
```
a.split()[1]
```

[61]:
```
'es'
```

[62]:
```
a = "Este es un string.   Con muchas sentencias."
```

Lo puedo cortar eligiendo el caracter, por ejemplo, pruebo con el "."

[63]:
```
a.split('.')
```

[63]:
```
['Este es un string', '   Con muchas sentencias', '']
```

Por lo que obtuve una lista de sólo dos elementos.

[64]:
```
a.split('.')[1].strip()

# Defino el caracter para el split .
# strip saca los blancos adelante del texto, que en este caso están demás.
# El default es el blanco, puede cambiarse a otro caracter
```

[64]:
```
'Con muchas sentencias'
```

[65]:
```
b = '   <-muchos espacios'
b.strip()
```

[65]:
```
'<-muchos espacios'
```

[66]:
```
a = 'tru'
b = 'la'
print(' '.join((a,b,b)))
print('-'.join((a,b,b)))
print(''.join((a,b,b)))
print(' & '.join((a,b,b)) + 'lo')
```

```
tru la la
tru-la-la
trulala
tru & la & lalo
```

Podemos hacer muchas cosas más con los strings. ¿Cómo se qué cosas y cuáles son sus comandos?

Como siempre hay que buscar y leer el manual.

3.7. Ingresos de datos por teclado

Se utiliza la función input()

[67]:
```
base = float(input("base: "))
altura = float(input("altura: "))

area=base*altura/2

print("El área es=",area)
```

```
base: 23
altura: 345
El área es= 3967.5
```

CAPITULO 4

Contenedores

4.1. Variables más complejas (Contenedores/Iterables)

En Python existen otros tipos de variables aparte de las que vimos en el capítulo anterior. Estas variables aprovechan la programación orientada a objetos y son estructuras que engloban a las variables básicas. Actúan no manejando datos individuales sino agrupaciones de ellos y se las conoce como contenedores. En palabras más simples, a los contenedores se los utiliza para manejar conjuntos de elementos de datos de cualquier tipo de los que vimos en el capítulo anterior. Los elementos pueden o no estar numerados, pueden o no estar repetidos y pueden o no ser inmutables. Con inmutables se indica que una vez cargados con un valor estas variables **NO** pueden ser modificadas durante el transcurso del programa. Si bien esto parece algo que no aporta y que obliga a consumir más recursos de la computadora, las variables inmutables son más "livianas" a la hora de ser usadas, dado que se trata de objetos sin mucha carga computacional. Los contenedores están pensados para que el usuario los utilice sobre grandes cantidades de datos.

En estas variables se debe prestar atención si se las está usando con paréntesis ( ), llaves { } o corchetes [ ] ya que estos son indicadores del tipo de contenedor.

Las variables guardadas en un contenedor se las considera **iterables** es decir, existen formas de recorrer con alguna operación a todos los elementos guardados en el contenedor.

No debe confundirse a los contenedores con variables aptas para álgebra vectorial, aunque se les parecen y podrían llegar a usarse, no son convenientes. Para el uso de cálculos en este campo deben utilizarse arreglos de la biblioteca *Numpy* que serán descriptos en un capítulo siguiente.

Los principales tipos de contenedores son los siguientes.

4.1.1. Tipos de datos Iterables:

- Listas
- Tuplas
- Sets
- Frozensets
- Diccionarios

4.2. Listas

- Son una colección de objetos, pueden ser de tipos diferentes. Tienen orden y este orden está numerado.
- Las listas pueden mutar, su contenido es modificable.
- Las listas se definen con una colección de datos entre corchetes, separados por “,”.
- Se utilizan los corchetes [ ] para indicar el conjunto de elementos de una lista.

Veamos un ejemplo:

[1]:
```
L = [1, '1', 1.4]
L
```

[1]:
```
[1, '1', 1.4]
```

Si escribo una operación o una variable sola, en otras palabras, cuyo valor no está asignándose a una variable, es decir, no es parte de ninguna operación, el notebook considera que hay una función **print()** implícita. Básicamente se usa la celda en modo calculadora. Pero esto sólo sirve para la última variable que se encuentre en la celda, las anteriores se ignoran y no se imprimen.

Veamos otra lista:

[2]:
```
L = ['azul','verde','rojo','violeta']
```

En este caso los elementos son una colección de variables de texto. Podemos preguntar qué tipo de variable es L. Recordemos que para averiguar el tipo de variable usamos la orden type()

[3]:
```
type(L) # Imprime el tipo de variable que es L
```

[3]:
```
list
```

Preguntemos por uno de los elementos de la lista, pero tenemos que recordar que: >* Los elementos están numerados. > >* El primer elemento tiene número **0** y el último es el **N-1** de la lista de **N** elementos en total.

[4]:
```
L[1]
```

[4]:
```
'verde'
```

[5]:
```
print(L[0])
```

```
azul
```

Pero también puedo preguntar con índices negativos, que significan que empiezo a contar por el final de la lista.

[6]:
```
L[-1] # Último elemento
```

[6]:
```
'violeta'
```

[7]:
```
L[-3]
```

[7]: 'verde'

La idea de que los números negativos sean indicación de que se empiece a contar por el final tiene sentido si no se conoce cuántos elementos contiene la lista. Y por otro lado tiene mucha utilidad si se necesita realizar alguna operación con los elementos que se encuentran en el final de la lista. En la vida real se utilizan a veces listas de una cantidad enorme de elementos.

Se pueden sumar listas, pero no es lo que uno se imagina, aunque tiene bastante sentido lo que sucede con lo que vimos en el capítulo anterior sobre la suma de *strings*.

[8]:
```
L=L+['rojo','amarillo']
```

Y si imprimo el resultado

[9]:
```
print(L)
```

```
['azul', 'verde', 'rojo', 'violeta', 'rojo', 'amarillo']
```

Vean que ahora la lista tiene los elementos originales con el agregado de los dos nuevos. En una lista los elementos pueden repetirse.

Bucle en listas

Por otro lado, hay maneras de utilizar un loop para tomar un rango de índices (inicio y fin) y un paso. Se lo construye de la siguiente manera:

Lista[Comienzo : final : paso] donde los elementos de índice i cumplen con *comienzo* <= *i* < *final* En este caso el final no está incluido.

Si no figura el comienzo se entiende que empieza la cuenta desde el inicio, es decir el elemento 0. Si no figura el final, es hasta el final de la lista. Si el paso no figura, se considera que es 1.

Un índice negativo se cuenta desde el final de la lista donde -1 es el último elemento, y un paso negativo implica que desde el final indicado voy avanzando hacia el comienzo de la lista.

"Slicing" (rebanar) es como se denomina al comando para extraer parte de una lista. Usaremos slicing para entender cómo se puede filtrar una lista a través de sus índices.

[10]:
```
L[1:3]
```

[10]: ['verde', 'rojo']

[11]:
```
L[2:]
```

[11]: ['rojo', 'violeta', 'rojo', 'amarillo']

[12]:
```
L[-2:]
```

[12]: ['rojo', 'amarillo']

[13]:
```
L[::2] # L[start:stop:step] cada dos elementos
```

[13]: ['azul', 'rojo', 'rojo']

[14]:
```
L[::-1]
```

[14]:
```
['amarillo', 'rojo', 'violeta', 'rojo', 'verde', 'azul']
```

[15]:
```
# Puedo modificar el contenido de la lista

L[2] = 'verde'
print(L)
```

```
['azul', 'verde', 'verde', 'violeta', 'rojo', 'amarillo']
```

Como la lista es un objeto usaremos algunas de las funciones o métodos ya programados en el objeto lista.

En este caso: “append( )”, “insert( )”, “extend( )”,“count( )” y “sort( )”

Muchos de estos métodos pueden utilizarse en la forma “punto” (“dot” en inglés) en la cual aplico el método haciendo:

Lista.método() donde dentro de los ( ) puedo agregar argumentos para casos especiales.

Veamos cómo los puedo usar:

append( ) → agrega un nuevo dato al final y lo uso en modo “dot”.

[16]:
```
L.append('rosa') # agregar un valor al final, note que a la lista L
                 # le corremos la función append(), que está construida
                 # dentro del objeto lista.
L
```

[16]:
```
['azul', 'verde', 'verde', 'violeta', 'rojo', 'amarillo', 'rosa']
```

[ ]:
```

```

insert( ) → agrega un nuevo dato en la posición indicada en esta función.

[17]:
```
L.insert(2, 'blue')    #  L.insert(índice, objeto) -- insertar un elemento
                       # en el lugar dado por el índice
L
```

[17]:
```
['azul', 'verde', 'blue', 'verde', 'violeta', 'rojo', 'amarillo', 'rosa']
```

extend( ) → agrega los elementos de otra lista.

[18]:
```
L.extend(['magenta', 'violeta'])
L
```

[18]:
```
['azul',
 'verde',
 'blue',
 'verde',
 'violeta',
 'rojo',
 'amarillo',
 'rosa',
```

```
 'magenta',
 'violeta']
```

sort( ) → ordena los elementos de la lista.

También puedo asignar un resultado de filtrar una lista a otra lista.

[19]:
```
L2 = L.copy()   # Notar que no se hizo L2=L ¿Cuál es la diferencia entre␣
 ↪ambas operaciones?
L2.sort()
print("Lista no ordenada", L)
print("")
print("Lista ordenada alfabéticamente",L2)
print("")
```

```
Lista no ordenada ['azul', 'verde', 'blue', 'verde', 'violeta', 'rojo',
'amarillo', 'rosa', 'magenta', 'violeta']

Lista ordenada alfabéticamente ['amarillo', 'azul', 'blue', 'magenta', 'rojo',
'rosa', 'verde', 'verde', 'violeta', 'violeta']
```

Pero si quiero ordenar la lista descendiendo de mayor a menor indico que "reverse" (al revés) sea verdadero. Aplicado este argumento provoca que si son números van del más grande al más chico, y si son textos van de la Z a la A.

[20]:
```
L2.sort(reverse=True)
print("Lista ordenada descendente", L2)
```

```
Lista ordenada descendente ['violeta', 'violeta', 'verde', 'verde', 'rosa',
'rojo', 'magenta', 'blue', 'azul', 'amarillo']
```

---

count() → cuenta la cantidad de veces que un tipo particular de elemento se encuentra en la lista. Notar que la búsqueda se hace indicando el elemento en cuestión como argumento de la función.

[21]:
```
print(L.count('yellow'))
```

```
0
```

[22]:
```
print(L.count('amarillo'))
```

```
1
```

Hay que recordar también que la lista puede contener cualquier variable como elemento, incluyendo otro iterable.

[23]:
```
L.append(2)
L
```

[23]:
```
['azul',
 'verde',
```

```
 'blue',
 'verde',
 'violeta',
 'rojo',
 'amarillo',
 'rosa',
 'magenta',
 'violeta',
 2]
```

[24]:
```
L = L[::-1] # Orden reverso
L
```

[24]:
```
[2,
 'violeta',
 'magenta',
 'rosa',
 'amarillo',
 'rojo',
 'violeta',
 'verde',
 'blue',
 'verde',
 'azul']
```

[25]:
```
L2 = L[:-3] # Elimina ya que corta los últimos 3 elementos
print(L)
print(L2)
```

```
[2, 'violeta', 'magenta', 'rosa', 'amarillo', 'rojo', 'violeta', 'verde',
'blue', 'verde', 'azul']
[2, 'violeta', 'magenta', 'rosa', 'amarillo', 'rojo', 'violeta', 'verde']
```

[26]:
```
#L[25] # Fuera del rango, da error
```

[27]:
```
print(L)
print(L[20:25]) # No hay error cuando rebano (slicing).
print(L[20:])
print(L[2:20])
```

```
[2, 'violeta', 'magenta', 'rosa', 'amarillo', 'rojo', 'violeta', 'verde',
'blue', 'verde', 'azul']
[]
[]
['magenta', 'rosa', 'amarillo', 'rojo', 'violeta', 'verde', 'blue', 'verde',
'azul']
```

Hasta ahora hemos visto una cantidad de funciones o métodos asociados al objeto *lista*. Usemos la función de Python dir() que me permite listar estos métodos. Note que varias de estas funciones ya las hemos usado en ejemplos anteriores.

[28]:
```
print(dir(L))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

[29]:
```
L1=L[2:]
L1.sort() # Se puede usar la tecla tab para ver los métodos o funciones
          # del objeto.
          # Esta puede variar según el origen del notebook utilizado
print(L1)
```

```
['amarillo', 'azul', 'blue', 'magenta', 'rojo', 'rosa', 'verde', 'verde',
'violeta']
```

[30]:
```
a = [1,2,3]
b = [10,20,30]
```

[31]:
```
print(a+b) # NO es lo que uno espera (o si?), pero tiene cierta lógica
```

```
[1, 2, 3, 10, 20, 30]
```

Lo que pasó fue que la suma concatenó ambas listas, es decir, pegó **b** después de **a**.

Y que pasará si hago:

c = a*3

[32]:
```
c = a*3
print(c)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Lo que sucedió fue que la lista **a** se concatenó 3 veces, y fue equivalente a hacer **c = a + a + a**

[33]:
```
#print(a*b) # No se pueden multiplicar listas
            # (Usar Numpy si son números
            # --> lo veremos la clase que viene)
```

4.2.1. Creando listas con comandos

Hay maneras en Python de crear listas cuando tienen patrones numéricos simples de una manera muy rápida y eficiente, veamos una forma simple de hacerlo con la función **range( )**

[34]:
```
L = list(range(4))
```

En el caso general *list(range(N))* crea una lista a partir de una definición numérica. El número es la cantidad de elementos
pero también indica que el último es el N-1 (3 en este caso) porque el primer elemento es el 0.

[35]:
```
L
```

[35]:
```
[0, 1, 2, 3]
```

Si en los arguementos de la función *range()* hay un solo valor(N) generara un conjunto de números enteros desde el "0" hasta N-1. Como se cuenta con el "0" incluido, el conjunto tendrá N valores.

En la forma más general es **range(inicio,final,paso)**, que generará la secuencia: [inicio, inicio+paso, inicio+2*paso,. . . ,final-paso].

Ejemplo: Y si quiero una lista de número pares:

[36]:
```
L = list(range(2, 20, 2)) # cada 2 enteros
L
```

[36]:
```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Puedo borrar elementos, en el ejemplo que sigue quito el quinto elemento pero en general se usa el comando **del( )** para remover el n+1-avo elemento. Es el n+1 porque hay que recordar que el primer elemento es el "0"

[37]:
```
L = list(range(0,20,2))
print(L)
del L[5]
print(L)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 12, 14, 16, 18]
```

4.3. Listas de listas

Es algo que se puede hacer, ya que una lista puede tener elementos de diferentes tipos de variables. También puede tener como elemento a otra lista.

[38]:
```
a = [[1, 2, 3], [10, 20, 30], [100, 200, 300]] # No es 2D (no es una MATRIZ,
                                                # es una lista de listas
print(a)
```

```
[[1, 2, 3], [10, 20, 30], [100, 200, 300]]
```

[39]:
```
print(a[0])
```

```
[1, 2, 3]
```

El elemento "0" es una lista.

[40]:
```
print(a[1][1])
```

```
20
```

Pero en este caso tomo la lista "1" y de ahí ahora tomo su elemento "1".

[41]:
```
# print(a[1,1]) # NO FUNCIONA, no son matrices.
```

[42]:
```
b = a[1]
print(b)
```

```
[10, 20, 30]
```

Lo que se hizo fue tomar la lista "1" y asignar la lista entera a la variable b. Y ahora pruebo que puedo cambiar un sólo valor en particular de esa lista, haciendo:

[43]:
```
b[1] = 999
print(b)
```

```
[10, 999, 30]
```

[44]:
```
print(a) # Cambiando b cambiamos a, Cuidado !!!
```

```
[[1, 2, 3], [10, 999, 30], [100, 200, 300]]
```

Los objetos cuando se asignan no se copian sino que se renombra el objeto ahora con otro nombre. Aclarando, no se crea un nuevo objeto, sino que con distintos nombres se puede acceder al mismo objeto o a parte de él como en este caso. Podemos preguntar con el operador **is** si es el mismo objeto.

El operador **is** ha generado alguna controversia recientemente sobre su uso con enteros, la discusión que es bastante compleja de este tema se puede leer acá: https://stackoverflow.com/questions/306313/is-operator-behaves-unexpectedly-with-integers.

[45]:
```
b[1] is a[1][1]
```

[45]:
```
True
```

[46]:
```
c = a[1][::] # copiar en vez de rebanar crea un nuevo objeto.
print(c)
c[0] = 77777
print(c)
print(a)
a[1] is c
```

```
[10, 999, 30]
[77777, 999, 30]
[[1, 2, 3], [10, 999, 30], [100, 200, 300]]
```

[46]:
```
False
```

Los métodos asociados a las listas son:

- sort( ): Ordena la lista en orden ascendente.
- type( ): Indica el tipo de Clase de un objeto.
- append( ): Agrega un elemento a la lista.
- extend( ): Agrega varios elementos a la lista.
- index( ): Retorna la primera aparición de un valor en particular.
- max( ): Retorna el item que tiene el valor máximo.
- min( ): Retorna el item que tiene el valor mínimo.

- len( ): Retorna la cantidad de elementos de una lista.
- clear( ): Borra la lista.
- insert( ): Inserta un elemento en la posición requerida.
- count( ): Retorna el número de veces que un elemento tiene el valor indicado.
- pop( ): Borra un elemento en la posición requerida y lo devuelve a una variable.
- remove( ): Borra la primera aparición de elemento con el valor indicado.
- reverse( ): Da vuelta el orden de los elementos de una lista.
- copy( ): Duplica la lista.

4.4. Tuplas

Son como las listas, pero sus valores son inmutables (no se pueden modificar durante la corrida del programa).

Se usan **( )** para indicar que es una tupla, a diferencia de las listas donde se emplean **[ ]** . Pero los **( )** en este caso son los que ya conocemos, utilizados para agrupar en aritmética. Una Tupla podría ser cargada con datos sin poner los paréntesis, pero por convención se los utliza.

En muchas maneras las tuplas son parecidas a las listas, pero la diferencia más importante es que las tuplas pueden usarse como índices (keys) en un diccionario y como elementos de un set, mientras las listas NO PUEDEN ser usadas para estas tareas. Veremos qué son diccionarios y sets a continuación.

La ventaja de las tuplas con respecto a las listas es que estas últimas tienen una gran cantidad de métodos o funciones programadas por lo cual son un objeto que necesita más recursos computacionales. Por eso se diseñaron las Tuplas para que sean más "livianas" y por lo tanto también más rápidas.

Dos funciones que tienen las tuplas pre-programadas son count( ) e index( ), la primera cuenta la cantidad de veces que cierto elemento se encuentra en una tupla y la segunda nos da el índice de la primera aparición de un elemento en particular.

Las funciones en Python que devuelven más de un valor retornan estos resultados en forma de una tupla.

Ejemplos:

[47]:
```
T = (1,2,3)
print(T.count(1))  # Cuento cuántas veces aparece el número "1"

T2 = (1,2,3,4,5,2,3,4,2,2)
print(T2.count(2))  # Cuento cuántas veces aparece el número "2"
```

```
1
4
```

Podría no poner los **( )** y funcionaría pero no es lo habitual. y podría generar alguna confusión para alguien que lee el programa. Note que al imprimir los paréntesis están donde se esperaría que estuviesen.

[48]:
```
T3 = 1,2,3,4,5
print(T3)
```

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

[49]:
```
T.index(2)  # Retorna el índice (sólo el primero) de dónde está el
            # valor buscado, en este caso un "2"
```

[49]: 1

Aunque es poco usado, si cargo varios elementos en una variable en una sola asignación, separados por comas, se presupone que la variable será una tupla:

[50]:
```
T2 = 1, 2, 3
print(T2)
type(T2)
```

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

[50]: tuple

[51]:
```
T[1]
```

[51]: 2

[52]:
```
### Las tuplas son inmutables
```

[53]:
```
#T(1) = 3 # No funciona
```

La única manera de borrar los elementos de una tupla es volver a escribirla entera.

[54]:
```
T=(3,2,1)  # la tengo que escribir toda de nuevo
T
```

[54]: (3, 2, 1)

Veamos una característica muy especial de las tuplas. Creo varias tuplas de tamaños diferentes y las imprimo:

[55]:
```
t3=(1,2,3)
t2=(1,2)
t1=(22)
t0=()

# imprimo los valores

print(t3)
print(t2)
print(t1)
print(t0)
```

```
(1, 2, 3)
(1, 2)
22
()
```

¿Qué paso aquí?

Por lo que se aprecia según el resultado de la impresión son todas tuplas ya que tienen los ( ), salvo cuando imprimo la que tiene un sólo elemento. Este aparece como un número sin los ( ) que caracterizan a la tupla. Es decir, es ahora un número y no una tupla. Note que incluso la que está vacía se imprimió con ( ), indicando que aún no teniendo ningún elemento sigue siendo una tupla.

¿Por qué se diseñaron las tuplas con esta característica?

La razón es para permitir que se pueda acceder rápidamente a los valores guardados para asignarlos a variables individuales en una sola sentencia.

Y esto se realiza de la siguiente forma:

var1,var2,var3,... = Tupla(val1,val2,val3,...)

donde la cantidad de variables debe ser igual a la cantidad de valores en la tupla.

Ejemplo:

Tengo la tuppla t_prueba que tiene los siguientes valores:

```
t_prueba = (1,2,3,4)
```

Si quisiera pasar esos valores a las variables a,b,c,d la manera más "clásica" sería:

```
a = t_prueba[0]
b = t_prueba[1]
c = t_prueba[2]
d = t_prueba[3]
```

Que es largo y lento, además en un caso real podría tener muchas más variables. Pero como es una tupla podría hacer:

[56]:
```
t_prueba = (1,2,3,4)

a,b,c,d = t_prueba

print(a)
print(b)
print(c)
print(d)

print("T_prueba es:",type(t_prueba)," a es:",type(a))
```

```
1
2
3
4
T_prueba es: <class 'tuple'>  a es: <class 'int'>
```

Para crear una tupla con un solo elemento debe indicarse con (elemento,). Note la ","

Cómo lo hacemos en el ejemplo siguiente:

[57]:
```
T=(22.3,)
print(T)
```

```
(22.3,)
```

4.5. Sets

Los **sets** son listas sin numerar, es decir los elementos no tienen una posición numérica con la cual el programador podría elegirlos en un orden determinado para una actividad.

Se distinguen de las listas por el uso de los { }.

No son muy usados, pero existen y están disponibles. Su principal ventaja es que al no estar numerados, son más simples y por lo tanto más eficientes en el momento de ser usados. Los elementos en un set no pueden estar repetidos.

Pero sí tienen una contraparte matemática clara, son la implementación en Python de la teoría de conjuntos ¿Recuerdan los diagrams de Venn?

Por lo cual, los elementos de un set serían equivalentes a los elementos de un conjunto y el set el conjunto.

Ejemplo:

[58]:
```
Mi_set={"gato","perro","loro"}
print(Mi_set)
print(type(Mi_set))
```

```
{'loro', 'perro', 'gato'}
<class 'set'>
```

El método **add( )** del contenedor set permite agregar nuevos elementos a un set ya existente.

[59]:
```
Mi_set.add("elefante")
print(Mi_set)
```

```
{'loro', 'elefante', 'perro', 'gato'}
```

Note que puede no haberse agregado al final, ya que este tipo de contenedor no tiene un orden establecido y este puede variar entre una computadora y otra cuando se ejecute este comando.

Y si vuelvo a agregar elementos que ya tengo:

[60]:
```
Mi_set.add("elefante")
Mi_set.add("tortuga")
Mi_set.add("gato")

print(Mi_set)
```

```
{'loro', 'elefante', 'perro', 'gato', 'tortuga'}
```

Tal como indicamos, de los elementos repetidos sólo se consideró uno como válido.

Los elementos de un set pueden ser removidos con los métodos “remove( )” o “discard( )”, el primero da error si el elemento no está y el segundo no (y el programa continua corriendo).

También existen métodos para realizar uniones o intersecciones entre dos sets de la forma en que se usan en teoría de Conjuntos.

[61]:
```
set1 = {'a1','a2','a3','a4','a5','a6'}
set2 = {'a2','a4','a6','a8','a10'}
```

```
set1.remove('a1')
print("Removí el 1 del set, ahora me quedan=", set1)
print("")

print("La unión de ambos sets sería:",set1.union(set2))
print("La intersección de ambos sets sería:",set1.intersection(set2))
```

```
Removí el 1 del set, ahora me quedan= {'a2', 'a6', 'a5', 'a4', 'a3'}

La unión de ambos sets sería: {'a2', 'a4', 'a8', 'a6', 'a5', 'a3', 'a10'}
La intersección de ambos sets sería: {'a2', 'a6', 'a4'}
```

4.6. Frozensets

Los **Frozen sets** o "sets congelados" son **sets** pero con la misma característica que las tuplas, son inmutables. En otras palabras son un set, pero no puedo reasignar con otros valores a las variables una vez creados.

Se utiliza la función frozenset() para crearlos a partir de una lista o una tupla. A diferencia de un set donde se usan{} para indicar su naturaleza, un frozen set se anota con doble sistema de símbolos de esta manera "frozenset({elementos})"

[62]:
```
ejemplo_de_una_lista = [1,2,3,4,5]
# Convierto la lista en un frozenset

frozen_set = frozenset(ejemplo_de_una_lista)

# Y me fijo la diferencia entre la lista y el frozen set
print(ejemplo_de_una_lista)
print(type(ejemplo_de_una_lista))
print()

print(frozen_set)
print(type(frozen_set))
```

```
[1, 2, 3, 4, 5]
<class 'list'>

frozenset({1, 2, 3, 4, 5})
<class 'frozenset'>
```

También puedo crearlos a partir de una tupla.

[63]:
```
ejemplo_tupla = (1,2,3,4,5)
frozen_set2= frozenset(ejemplo_tupla)

print(ejemplo_tupla)
print(type(ejemplo_tupla))
print()

print(frozen_set2)
print(type(frozen_set2))
```

```
(1, 2, 3, 4, 5)
<class 'tuple'>

frozenset({1, 2, 3, 4, 5})
<class 'frozenset'>
```

Resumiendo hasta ahora

Tipo	Mutabilidad	Elem. Repetidos	Elem. Numerados*	Símbolo
Listas	Si	Si	Si	[ ]
Tuplas	No	Si	Si	( )
Set	Si	No	No	{ }
Frozenset	No	No	No	frozenset({ })

* Indica que puedo acceder a un elemento en particular a través del número que señala su posición.

4.7. Diccionarios

Permiten organizar los datos en estructuras donde una palabra llave dispara otro dato guardado en el diccionario. Es decir relaciona una llave determinada con un valor particular. Pero este valor puede ser de cualquier tipo de variable, incluso una lista o una tupla.

Para distinguirlos de las listas y tuplas, los diccionaros utilizan "{ }" en su definición y uso.

[64]:
```
a_diccionario = {'uno' : 1.0,
                 'dos' : 2.0,
                 'una_lista' : ['esta', 'es', 'una', 'lista']
                 }
```

Si pregunto con "type( )" qué tipo de variable es a_diccionario:

[65]:
```
print(type(a_diccionario))
```

```
<class 'dict'>
```

Me contesta que es de la clase 'dict', un diccionario.

Para acceder a los datos:

[66]:
```
print(a_diccionario['uno'])
print()
print(a_diccionario['una_lista'])
```

```
1.0

['esta', 'es', 'una', 'lista']
```

Y de la misma manera puedo modificarlos o agregar nuevos valores:

[67]:
```
a_diccionario['otra_lista']=['esta','es','otra','lista']
a_diccionario['dos']= 'two'
```

```
# Y si los imprimo:
print(a_diccionario['otra_lista'])
print(a_diccionario['dos'])
```

```
['esta', 'es', 'otra', 'lista']
two
```

Otra manera de crear un diccionario es con la orden **dict( )** y entre los paréntesis escribo las llaves que apuntan a los valores de la siguiente forma:

dicc2 = dict(llave1='valor1', llave2='valor2', llave3='valor3,...)

Ejemplo:

[68]:
```
datos = dict(galaxia1='NGC2366', galaxia2='NGC1672', galaxia3='NGC7949',␣
 ↪galaxia4='NGC7552')

print(datos)
print("")
print(datos['galaxia3'])
```

```
{'galaxia1': 'NGC2366', 'galaxia2': 'NGC1672', 'galaxia3': 'NGC7949',
'galaxia4': 'NGC7552'}

NGC7949
```

Puedo listar todas las llaves de un diccionario en particular con la función keys(), veamos si la uso:

[69]:
```
b = a_diccionario.keys()
print("Las \'keys\' son:",b)
print("")
print("Y b es de tipo:",type(b))
```

```
Las 'keys' son: dict_keys(['uno', 'dos', 'una_lista', 'otra_lista'])

Y b es de tipo: <class 'dict_keys'>
```

CAPITULO 5

Archivos en Python

5.1. Tipos de archivos

Archivos de texto

Son archivos con textos editables con cualquier editor (por ejemplo, un programa en Python). Tienen renglones escritos con códigos Ascii o Unicode. Los finales de los renglones o líneas están indicados con un End of Line (EOL) que se puede escribir con la orden "\n" (newline).

Archivos binarios

En este tipo de archivo, no hay una indicación de final de línea y los datos están en una forma binaria sólo manejable por un program de computadora. Sirven para que una aplicación genere un resultado que luego será usado por otro programa. Tienen la ventaja de ser más rápidos para leer y además ocupan mucho menos espacio en el disco de la computadora.

5.2. Órdenes para el manejo de archivos

Para abrir un archivo, usamos la orden **OPEN()** que existe como función interna del Python. Esta orden relaciona el archivo en cuestión con un objeto. En otras palabras, creo un objeto que permite el acceso al archivo. Como mínimo esta función necesitaría el nombre y el camino al archivo. Se usa de la siguiente forma:

archivo1 = open('Mi_archivo.txt')

Cuando el archivo no se utiliza más se lo cierra con la sentencia **close()**.

En este caso que el archivo se indentificó con el objeto *archivo1*. La función que lo cierra y desactiva a este objeto, sería:

archivo1.close()

También para indicar más detalle se puede poner un identificar que indica que clase de actividad quiero realizar con este archivo, y se lo utiliza de esta manera:

Archivo2 = open("Nombre del archivo","Modo de acceso")

donde los modos de acceso principales son los siguentes:

Caracter

Significado

'r'

Abrir para leer, si el archivo no existe indica un error. Es el modo "default". Se comienza a leer por el principio del archivo.

'r+'

Abrir para leer y escribir, si el archivo no existe indica un error. Es el modo "default". Se comienza a leer por el principio del archivo.

'w'

Abrir para escribir, si el archivo existe se le escribe encima desde el principio. Se comienza a escribir por el principio del archivo. Crea el archivo si este no existe.

'w+'

Abrir para leer y escribir, si el archivo existe se le escribe encima desde el principio. Se comienza a escribir por el principio del archivo. Crea el archivo si este no existe.

'rb' o 'wb'

Abrir en binario, los datos son Bytes. 'rb' para leer, 'wb' para escribir.

'a'

Abrir el archivo para ser leído. Este es creado si no existe. Se ubica la nueva posición de escritura al final del archivo. Entonces is escribo nuevos datos irán al final, después de los datos existentes.

'a+'

Abrir el archivo para leer o escribir. Este es creado si no existe. Se ubica la nueva posición de escritura al final del archivo. Entonces si escribo nuevos datos estos irán al final, después de los datos existentes.

Por lo cual se podrían hacer las siguientes combinaciones:

open('archivo.txt')

open('archivo.txt', 'r')

open('archivo.txt', 'w')

5.3. Leer un archivo de texto

Para ello hay 3 comandos posibles:

5.3.1. read() o read(n)

Devuelve los Bytes leídos en forma de string. Lee *n* Bytes, si *n* no está especificado lee el archivo entero.

5.3.2. readline() o readline(n)

Lee cada línea del archivo y la devuelve en forma de string. Para un *n* específico, lee hasta *n* Bytes o hasta que se acabe el renglón. No importa si el *n* es más grande que el tamaño del renglón.

5.3.3. readlines() o readlines(n)

Lee todos los renglones del archivo y retorna cada línea como elemento de un string.

Ejemplos:

Creo un archivo y guardo unos datos en forma de strings. Para ello utilizo el comando de Jupyter %%writefile. Este es usado de la siguiente manera: "%%writefile Nombre_del_archivo", crea el "archivo" con toda la información que encuentra en la celda.

[1]:
```
%%writefile mi_archivo.txt
"Estoy acá
"Segunda línea"
"Tercera línea"
"Cuarta línea"
```

```
Overwriting mi_archivo.txt
```

Como el archivo creado tiene 4 renglones, probaremos leerlo primero con el comando read(). Por lo cual primero lo abro con un *open()*. Como no se indicaron la cantidad de Bytes se lee el archivo completo.

[2]:
```
archivo1 = open("mi_archivo.txt","r")

print("La salida de la lectura será: ")
print(archivo1.read())
print("")

archivo1.close()
```

```
La salida de la lectura será:
"Estoy acá
"Segunda línea"
"Tercera línea"
"Cuarta línea"
```

[3]:
```
archivo1 = open("mi_archivo.txt","r")
print( "La salida del comando readline es: ")
print(archivo1.readline())
```

```
La salida del comando readline es:
"Estoy acá
```

Como lee los renglones de a uno leyó el primero y si lo vuelvo a correr:

[4]:
```
print(archivo1.readline())
```

```
"Segunda línea"
```

Lee el segundo, ya que lee renglón por renglón en orden.

Ahora cierro el archivo:

[5]:
```
archivo1.close()
```

En cambio si uso el comand read() pero con cierta cantidad de Bytes por ejemplo 9 Bytes:

[6]:
```
archivo1 = open("mi_archivo.txt","r")
print("La salida de la función read(9) es: ")
print(archivo1.read(9))
print()

archivo1.close()
```

```
La salida de la función read(9) es:
"Estoy ac
```

En este caso **read()** no leyó todo el archivo, sólo los primeros 9 Bytes.

En cambio en la función **readline( )** si especifico el número de Bytes, lee esa cantidad de Bytes pero del renglón, ya no del archivo. En la próxima lectura seguirá con los Bytes no leídos. Veamos un ejemplo, donde pido menos Bytes que los que tiene el renglón:

[7]:
```
archivo1 = open("mi_archivo.txt","r")

print("La salida de la función readline() es:")
print(archivo1.readline(8))
print(archivo1.readline(11))
print(archivo1.readline(5))
print(archivo1.readline(5))

archivo1.close()
```

```
La salida de la función readline() es:
"Estoy a
cá

"Segu
nda l
```

En cambio *readlines()* lee todo el archivo y cada renglón se convierte en una componente de una lista de strings.

[8]:
```
archivo1 = open("mi_archivo.txt","r+")

print("La salida de la función es: ")
print(archivo1.readlines())
```

```
print()

# con la función seek() reseteo la lectura del
# archivo, y puedo volver a leer desde el comienzo

archivo1.seek(0)
todo=archivo1.readlines()

# imprimo el tercer elemento
print(todo[2])

archivo1.close()
```

```
La salida de la función es:
['"Estoy acá\n', '"Segunda línea"\n', '"Tercera línea"\n', '"Cuarta línea"
\n']

"Tercera línea"
```

5.4. Escribir un archivo de texto

Hay dos formas de escribir un archivo de texto, con el modo **write()** que escribe un renglón a la vez o con el modo **writelines** que permite escribir varios renglones.

Si abro un archivo para escritura:

[9]:
```
archivo1 = open("mi_archivo.txt","w")
```

Le puedo escribir un renglón haciendo:

[10]:
```
archivo1.write("Hola, aquí de nuevo\n")
```

[10]: 20

O varias líneas de una sola vez:

[11]:
```
# Creo una lista de strings
L = ["Segunda línea\n","Tercera línea\n","Cuarta línea\n", "Quinta línea\n"]

# las escribo en el archivo
archivo1.writelines(L)
```

Veo como quedó el archivo:

[12]:
```
archivo1 = open("mi_archivo.txt","r")
print(archivo1.read())
```

```
Hola, aquí de nuevo
Segunda línea
```

```
Tercera línea
Cuarta línea
Quinta línea
```

Estas son las órdenes originales de Python para el manejo de archivos. Más adelante veremos que varias librerías agregan otras formas de leer archivos, con fuciones mucho más poderosas de las que hemos visto en este capítulo e incluso más cómodas para realizar esa tarea.

CAPITULO 6

Estructuras de control

Son estructuras diseñadas para que cierto bloque de órdenes se repita una cantidad determinada de veces o que se ejecute mientras se cumplen ciertas condiciones. Los Bloques son definidos por "indentation" o sea la sangría. No hay una orden específica para indicar el final, este se asume con la desaparición de la sangría usada en esa estructura de control. Aunque puede continuar otra de una estructura superior a esta. Es decir, cuándo más interna es la estructura más larga es la sangría que determina la pertenencia de las órdenes.

En las estructuras de control tenemos las siguientes órdenes

- **for** para realizar bucles o loops.
- **if( )/elif/else** para hacer preguntas y tomar decisiones.
- **while( )** para realizar una tarea mientras el argumento sea verdadero.
- **"Comprehension list"** que son una versión compacta de los comandos anteriores y sirven para realizar todos los comandos anteriores en un sólo renglón. Sirven para volver más compacto el código.

6.1. FOR

Empecemos con la sentencia **for** para realizar "loops" o bucles.

Lo uso haciendo:

for ***variable*** **in** ***contenedor*****:**

> **Bloque determinado con una sangría similar que contiene las operaciones a realizar**

La *variable* es similar a como usamos la variable de un *DO* de Fortran. El contenedor puede ser cualquiera de los que vimos la clase anterior. Es de decir una Lista, Tupla, Set, etc.

Los "**:**" al final de la línea son obligatorios porque avisan que ahora viene el bloque de órdenes del bucle.

La idea es que la variable va tomando a todos los elementos del contenedor. En muchos textos a los elementos del contenedor se los considera como *iterables*. Dado que se itera sobre todos los elementos de ese conjunto. Por lo cual las operaciones que se especifiquen dentro del bloque del **for** serán aplicadas a todos los elementos del contenedor.

También hay que recordar que un contenedor puede tener como un elemento a otro contenedor, así que donde hablamos de *variable* podría ser algo que englobe más de una variable de

las básicas. Por ejemplo, podría tener una lista de tuplas, por lo cual el **for** actuaría de a una tupla por vez.

[1]:
```
for i in [1,2,3]:
      print(i+1)        # Prestar atención a la sangría
                        # ("indentation" en inglés)
```

```
2
3
4
```

Recordando que un contenedor puede tener distintos tipos de variables básicas, podríamos hacer lo siguiente:

[2]:
```
for cosa in [1,'ff',2]:
    print(cosa)
    print('end')
print('final end')    # El final de la sangría indica el final
                      # del bloque "for"
```

```
1
end
ff
end
2
end
final end
```

Por ejemplo, puedo recorrer un diccionario por sus *keys*.

[3]:
```
# Si defino un diccionario
ATOMIC_MASS = {}  # <-- Diccionario vacío

ATOMIC_MASS['H'] = 1
ATOMIC_MASS['He'] = 4
ATOMIC_MASS['C'] = 12
ATOMIC_MASS['N'] = 14
ATOMIC_MASS['O'] = 16
ATOMIC_MASS['Ne'] = 20
ATOMIC_MASS['Ar'] = 40
ATOMIC_MASS['S'] = 32
ATOMIC_MASS['Si'] = 28
ATOMIC_MASS['Fe'] = 55.8

# Puedo imprimir a partir de las keys, todos los valores del diccionario.
#  -> Ojo. La salida no está ordenada
# dict.keys() es el método que lista los "keys" del diccionario

print(ATOMIC_MASS.keys())

for key in ATOMIC_MASS.keys():
    print(key, ATOMIC_MASS[key])
```

```
dict_keys(['H', 'He', 'C', 'N', 'O', 'Ne', 'Ar', 'S', 'Si', 'Fe'])
H 1
He 4
C 12
N 14
O 16
Ne 20
Ar 40
S 32
Si 28
Fe 55.8
```

6.2. IF/ELIF/ELSE

Puedo hacer una preguna al estilo:

```
if(Condición):
    pasa esto si "_condición_" es verdadero
elif(otra Condición):
    pasa esto otro si "_otra condición_" es verdera y "algo" fue falso
else:
    Como las anteriores fueron falsas, hago esto que sigue acá
```

Note la sangría que ordena los inicios y finales en cada bloque de actividades.

La *Condición* puede ser una variable lógica que contiene un verdadero o un falso, o una comparación entre magnitudes realizada por un operador. A estos casos los veremos a continuación.

Se pueden utilizar las sentencias de Control con un estilo bastante parecido al Fortran salvo por la orden que en Fortran se llama “elseif()” y en Python “elif()”

Note que cada orden de la estructura del bloque lleva los “**:**” al final

6.3. Operadores condicionales

En el caso de que requiera comparar dos valores o más, deberá escribir la condición en el IF(), y esta se activará cuando esa condición sea verdadera. Las condiciones para activar o no un IF() tienen en python un muy rico lenguaje para detallar eventos que pueden ser verdaderos o falsos.

Operadores de Comparación: Se comparan números entre ellos, recordar que se puede poner una expresión que será evaluada antes de realizar la comparación.

Operador	Descripcion	sintaxis
>	Mayor	X > Y
<	Menor	X < Y
==	igual	X == Y
>=	Mayor o igual	X >= Y
<=	Menor o igual	X <= Y
!=	No es igual	X != Y

Operadores Lógicos:

Operador	Descripcion	sintaxis
and	Verdadero si ambos son verdaderos	X and Y
or	Verdadero si uno es verdadero	X or Y
not	Verdadero si es falso, y falso si es verdero	not X

Operadores de indentidad: Verifican que ocupen la misma memoria ram, con otras palabras, se pregunta si son el mismo objeto o parte de él.

Operador	Descripcion	sintaxis
is	x es igual a y	X is Y
is not	x no es igual a y	X is not Y

En el caso de los diccionarios se usa el operador '**in**' para preguntar si una 'key' en particular está definida. El modo de hacer estas operaciones sería el siguiente:

[4]:
```
num = 223.4

if num > 0:
    print("Es un número positivo")
elif num == 0:
    print("Cero")
else:
    print("Es un número negativo")
```

```
Es un número positivo
```

Puede tener estructuras de control una dentro de otra, pero las diferentes sangrías me indican los niveles en que cada estructura es válida.

[5]:
```
for i in range(10):
    if i > 5:
        print(i)

# Prestar atención a la doble sangría del if
```

```
6
7
8
9
```

[6]:
```
for i in range(10):
    if i > 5:
        print(i)
    else:
        print(i ,'es menor que cinco')
print('END')
```

```
0 es menor que cinco
1 es menor que cinco
2 es menor que cinco
3 es menor que cinco
4 es menor que cinco
5 es menor que cinco
6
7
8
9
END
```

También puedo usar estructuras de control en un SET aunque sus elementos no estén numerados

[7]:
```
este_set = {"frutilla", "banana", "cereza"}

for x in este_set:
  print(x)
```

```
frutilla
banana
cereza
```

En el caso de los diccionarios se usa el operador '**in**' para preguntar si una llave o 'key' en particular está definida. Veamos su modo de uso utilizando el diccionario que definimos anteriormente y preguntemos si 'mapa' nos dispara un resultado (spoiler: no está)

[8]:
```
a_diccionario = {'uno' : 1.0,
                 'dos' : 2.0,
                 'una_lista' : ['esta', 'es', 'una', 'lista']
                 }

if 'mapa' in a_diccionario:
     print(a_diccionario['mapa'])
else:
     print('Esa llave no existe')
```

```
Esa llave no existe
```

También para el caso de un diccionario se puede pedir que la orden **for**, recorra simultáneamente llaves y valores. Esto se puede hacer porque la función **items( )** de los diccionarios crea una tupla (llave, valor)

Esta orden puede ser muy confusa y se tarda en entenderla... <– **tomarlo con calma**

[9]:
```
print( a_diccionario.items())
```

```
dict_items([('uno', 1.0), ('dos', 2.0), ('una_lista', ['esta', 'es', 'una',
'lista'])])
```

Note que se imprimen una lista de tuplas, donde cada tupla es (llave, valor) y si es una lista puedo usar un for, pero descomponiendo la tupla en sus dos componentes por separado (recordar que esto es una propiedad de las tuplas).

[10]:
```
for llave, valor in a_diccionario.items():
    print(llave,'=', valor)
```

```
uno = 1.0
dos = 2.0
una_lista = ['esta', 'es', 'una', 'lista']
```

6.4. Try/Except

Estos comandos tienen una estructura perecida a un if()/else, pero diseñada para manejar errores.

Lo que escribo en el bloque del **try:** es probado y si es cierto ejecutado, pero si no lo es se activa el bloque del **except tipo_de_error**. Es decir, genera una excepción, avisa de ella y el programa sigue corriendo, no muere como debería haber pasado. Hay una cantidad muy importante de **tipos de errores** y en cada versión de python se han ido agregando más posibles errores. En la versión de Python que se esté utilizando tendrá un manual con un listado de ellos.

Probémoslo con el diccionario, y en este caso la excepción es 'KeyError'

[11]:
```
try:
  print(a_diccionario['dos'])
except KeyError:
  print('No estaba la llave')
```

```
2.0
```

[12]:
```
# y si pido una llave inexistente, se activa
# el except

try:
  print(a_diccionario['mapa'])
except KeyError:
  print('No estaba la llave')
```

```
No estaba la llave
```

6.5. WHILE( ), Break y Continue

El comando while(algo) funciona creando un loop mientras "algo" sea verdadero. En "algo" podemos construir un condicional con los operadores que hemos visto.

Por ejemplo:

[13]:
```
i = 1
while i < 4:
  print(i)
  i += 1
```

```
1
2
3
```

Podemos incluso poner otras estructuras de control y generar un **break** que rompe al while dada una condición, veamos como:

[14]:
```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

Entonces cuando i sea igual a 3 el loop terminará.

O con el comando **continue** se podrá evitar que cierto valor sea procesado, provocando que el programa siga con el próximo valor en la orden while( ).

[15]:
```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

Notar que el valor "3" no fue considerado en la secuencia.

6.6. Sentencias Match/Case

A partir del Python 3.10 se agregaró esta nueva sentencia de control, su uso es similar a clasificar en casos a las variantes de un IF(). Aunque también se podría verla como un sistema de clasificar patrones y con ellos generar diferentes resultados. La orden **case** es muy popular en varios lenguajes incluyendo Fortran 90/95.

Ejemplo, clasifiquemos errores del resultado de llamar a páginas web:

def http_error(status): match status: case 400: return "Bad request" case 404: return "Not found" case 418: return "I'm a teapot" case _: return "Something's wrong with the internet"

En esta función se puede apreciar que diferentes números de error generan diferentes textos de salida. Hay que notar que la última pregunta con la orden case (_:) actúa como el equivalente a un "else" de un IF() bloque, es decir, se activa cuando ninguno de los casos anteriores es válido. También se puede preguntar por varios casos en una sóla orden:

[16]:
```
status = 404
match status:
    case 401 | 403 | 404:
```

```
        print("No funcionó")
```

```
No funcionó
```

También se pueden hacer cosas más complicadas como utilizar de entrada una tupla con varias componentes. La sentencia **match** tiene la capacidad de comparar componente a componente de la tupla. En este ejemplo, tomado del tutorial de Guido Van Rossum, no sólo se comparan las componentes si no que se las copia al resultado. Indicando que componente del par ordenado ocupan.

[17]:
```
def prueba_match(point):
# point is an (x, y) tuple
    match point:
      case (0, 0):
        print("Origen")
      case (0, y):
        print(f"Y={y}")
      case (x, 0):
        print(f"X={x}")
      case (x, y):
        print(f"X={x}, Y={y}")
      case _:
        raise ValueError("No es un punto")

prueba_match((0,0))
prueba_match((0,4))
prueba_match((36,0))
prueba_match((3,4))
```

```
Origen
Y=4
X=36
X=3, Y=4
```

6.7. Listas de Comprensión (List Comprehension)

Estas son formas de estructuras de control extremamadamente compactas y bastante (¡o muy!) confusas cuando se las ve por primera vez. Se caracterizan por ahorrar muchas líneas de código.

Se basan en la idea de "iterar" sobre una estructura de control para generar sus valores, o bien, por ejemplo, convertir una lista en otra (por un cálculo, o un filtrado de sus elementos). Una variante de estas son las funciones generadoras que veremos en un capítulo siguiente.

Veamos un ejemplo de cómo funcionan:

[18]:
```
# creamos unas listas
animales = ['perro','gato','loro', 'tortuga','paloma']
números = [1,2,3,4,5]
```

```
# y las procesamos
# Por ejemplo, creo una nueva lista en la que sólo estén
# los animales que tengan una "a" en el nombre

nuevos_animales = [x for x in animales if "a" in x]
print("Nueva lista:",nuevos_animales)


# o calculo los cubos de los números de la lista "números"
nuevos_números= [x**3 for x in números]
print("Nuevos Números:", nuevos_números)
```

```
Nueva lista: ['gato', 'tortuga', 'paloma']
Nuevos Números: [1, 8, 27, 64, 125]
```

La sintaxis más general de una "list comprehension" es:

newlist = [expresión for item in colección if condición == True]

que en realidad se puede reducir a:

newlist = [expresión for item in colección]

Pero existe mucha libertad de cómo usar esta orden, por ejemplo podría haber más de un if() o más de una "collection de items"

newlist = [expresión for item in colección if test1 and test2]

o

newlist = [expresión for item in colección1 and item2 in colección2]

Por colección nos referimos a un iterable, y la definición exacta de un iterable la veremos más adelante, pero por ejemplo las listas, tuplas, etc., son iterables. Es decir puedo acceder uno por uno en un conjunto de elementos.

Note que la definición de un List Comprehensions va rodeada de [] (corchetes) como lo hace la definición de una lista en Python

Por ejemplo, puedo recorrer una lista haciendo:

[19]:
```
lista_nueva = [x for x in range(10)]
print(lista_nueva)
```

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

O recorrerla poniendo una condición:

[20]:
```
lista_nueva = [x for x in range(10) if x < 5]
print(lista_nueva)
```

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

O incluso aprovechar los métodos o funciones de los objetos:

[21]:
```
# la función upper pasa la letra a mayúscula
newlist = [x.upper() for x in animales]
print(newlist)
```

```
['PERRO', 'GATO', 'LORO', 'TORTUGA', 'PALOMA']
```

Al principio, estas listas de compresión son confusas, pero esta estructura puede entenderse mejor si se la piensa dividida en renglones donde cada uno de ellos desarrolla una actividad. Por ejemplo, si tengo

[x**2 for x in range(10) if x<5]

Expresión que puedo dividir en 3 líneas:

- la expresión matemática,
- la iteración y
- la condición para que se ejecute.

En este ejemplo:

x^2 es la expresión matemática

for x in range(10) es la iteración

if x < 5 es la condición

Separando en las 3 partes es más fácil de entender.

[22]:
```
[x**2 # expresión matemática
  for x in range(10) # iteración
  if x < 5] # condición
```

[22]:
```
[0, 1, 4, 9, 16]
```

CAPITULO 7

Funciones

7.1. Funciones en Python

En el lenguaje Python se pueden construir funciones para ser usadas reiteradas veces. Las funciones en Python tienen que estar definidas y por lo tanto cargadas antes de ser usadas, recordar que Python no es un lenguaje compilado. Por eso es muy común que todo código comience con la definición de varias funciones o con una carga de bibliotecas que contienen esas funciones.

Cómo en otros lenguajes la idea de función es escribir una sóla vez, cualquier estructura que se repita muchas veces en un cálculo. Pero esa idea a ido evolucionando a convertirse en un código que desarrolla una tarea en particular y que puede guardarse en forma separada y luego reutilizarse en otros programas. La idea entonces es que en el caso de resolver un problema en particular, guardar el código de las funciones utilizadas y que esta solución de software quede en disponibilidad para que uno u otros puedan acceder a ese código y reutilizarlo.

El éxito de Python se debe entonces a la existencia de una cantidad de bibliotecas especializadas en una cuestión particular que contienen funciones muy útiles sobre ese tema.

Para definir una función se usa la orden **def** seguido de su nombre y entre paréntesis las variables o los datos que se ingresan a la función. A estos últimos también se los llama argumentos. Al final del paréntesis tiene que ir el símbolo **:** para indicar al intérprete Python que las líneas siguientes con una sangría similar (un bloque) pertenecen a la función.

La sangría, al igual que en el caso de las estructuras de control, determina que órdenes son de esa función en particular, es decir determina el bloque de órdenes propias de la función. Como dentro de una función puede haber estructuras de control internas, estas tendrán más espacios de sangría que las que se indican como sentencias que son parte del bloque. Pero eso no es un problema, se siguen considerando dentro del bloque que forma la función.

Las funciónes suelen terminar con la orden **return**, que puede o no llevar los valores resultantes del cálculo, los cuales serán enviados al lugar donde la función fue llamada. La orden **return** es optativa, puede incluso no estar.

Veamos un ejemplo de una función que eleva al cubo valor que se ingresa:

[1]:
```
def cubo(x):
    print(x**3)
    return
```

```
cubo(5)
```

```
125
```

[2]:
```
# pero podría no haber puesto el return

def cubo(x):
    print(x**3)

cubo(5)
```

```
125
```

Notar que la función se dio por finalizada porque terminó la sangría, y no se usó la orden return. Este ejemplo demuestra la importancia de la sangría en Python para determinar inicio y fin de bloques de código.

Pero la pregunta es: ¿Devolvió la función un resultado?

[3]:
```
def func_cubo(x):
    """
    Retorno el cubo de un número.
    A este comentario se la llama docstring y puede accederse desde el
    notebook como una ayuda para recordar detalles del uso de la función
    como por ejemplo que variables ingresan y en que orden.
    Note las 3 comillas que se usan para indicarlo y cerrarlo
    """
    return(x**3)    # <-- El return puede hacer cuentas!!!

a = func_cubo(3)

print(a)
print("")

# pido el "help" para recordar detalles de la función y me
# devuelve el texto del docstring

help(func_cubo)
```

```
27

Help on function func_cubo in module __main__:

func_cubo(x)
    Retorno el cubo de un número.
    A este comentario se la llama docstring y puede accederse desde el
    notebook como una ayuda para recordar detalles del uso de la función
    como por ejemplo que variables ingresan y en que orden.
    Note las 3 comillas que se usan para indicarlo y cerrarlo
```

[4]:
```
# Otra manera hacer shift tab dentro del parentesis,
# aunque en algunas computadoras no funciona
# ¿Lo hace en la tuya?

func_cubo(3)
```

[4]: 27

Como en otros lenguajes puedo poner en los argumentos de la función variables o números

[5]:
```
a=3
print(a)

print(func_cubo(a))
```

```
3
27
```

Estos argumentos que hemos usado hasta ahora son requeridos y son obligatorios, ya que así están en la definición de la función. Debe llamarse a la función con estos argumentos cargados con valores numéricos o variables que los contengan. La falta de uno de estos argumentos en una llamada genera un error. Debido a que su orden importa se suelen llamar argumentos de posición o **positional arguments**.

Pero existen otros argumentos llamados de teclado o **keyword arguments** donde se indica un valor default (default = predeterminado) por si no se lo indica en el momento que se llama la función.

[6]:
```
def func_rara(x, y, z, a=0, b=1):
    """
    Esta función tiene 5 argumentos, pero dos de ellos
    tienen valores por default (entonces no son obligatorios)
    """
    return a + b * (x**2 + y**2 + z**2)**0.5


D = func_rara(3, 4, 5)
print(D)
```

```
7.0710678118654755
```

[7]:
```
E = func_rara(3, 4, 5, 10, 100)
print(E)
```

```
717.1067811865476
```

Veamos un ejemplo que es muy bueno para mostrar el uso de este tipo de argumentos.

Supongamos que queremos hacer un programa que pase medidas hechas en el sistema imperial, es decir pies y pulgadas a centímetros. la medida puede estar en pies, o en pulgadas o en los dos sistemas.

Recorar que 1 pulgada = 2.54 cm y 1 Pie = 12 pulgadas

[8]:
```
def cm(pies=0, pulgadas=0):
    pulgadas_a_cm = pulgadas * 2.54
    pies_a_cm = pies * 12 * 2.54
    return pulgadas_a_cm + pies_a_cm
```

En esta función pies y pulgadas tienen ya prefijado valores de inicio, es decir un default que es en ambos casos un "0".

Veamos como puede correr esta función y aprovechar los valores default de las argumentos de teclado.

[9]:
```
# Sólo tengo un valor en pies y nada en pulgadas

a = cm(pies=20)
print("20 pies son: ",a)

# Sólo tengo un valor en pulgadas y nada en pies

b = cm(pulgadas=22.5)
print("22 pulgadas son: ",b)

#Tengo parte de la medida en pies y otra parte en pulgadas

c = cm(pies=5, pulgadas=3.5)
print("5 pies y 3.5 pulgadas son: ",c)

# E incluso puedo invertir los argumentos y lo puedo hacer porque
# tiene nombres las variables

d = cm(pulgadas=3.5,pies=5)
print("3.5 pulgadas y 5 pies on: ",d)
```

```
20 pies son:  609.6
22 pulgadas son:  57.15
5 pies y 3.5 pulgadas son:  161.29000000000002
3.5 pulgadas y 5 pies on:  161.29000000000002
```

Puedo mezclar argumentos de ambos tipos, pero los de tipo "keyword" tienen que ir al final.

Pero estos últimos no tiene un orden si los escribo como **variable=dato**

También puedo asignar la función a una variable con cierto nombre y utilizar ese nombre para el cálculo. Veamos:

[10]:
```
X = func_cubo

print(X(5))
```

```
125
```

Aunque lo que sucedió a que el objeto que contiene la función tiene ahora más de un nombre. Es decir, no se copió.

7.2. *args y **kargs

Veamos el caso de una facilidad que tiene Python de hacer variable la cantidad de argumentos tanto los posicionales como los argumentos "Keywords" que envio a una función.

7.2.1. Caso de una cantidad arbitraria de argumentos posicionales (*args)

Si por ejemplo realizo un programa que utiliza una función que calcula el promedio por ejemplo para 3 números, lo haría de la siguente manera: Es decir le envio 3 argumentos

[11]:
```
def prom(x,y,z):
    prom = (x+y+z)/3
    return prom

print(prom(2,3,4))
```

```
3.0
```

De esta manera funcionaría, pero no me serviría para promediar más de 3 números o menos de 2. Existe una manera de universalizar para que el programa me sirva para una cantidad no determina de números. Es decir, que yo pueda mandar una cantidad de argumentos posicionales arbitraria.

Para realizar esta tarea tengo que indicar que la cantidad de argumentos no se conoce, sólo se sabrá al momento de la llamada a la función. Para eso uso ***args** como argumento en la defínción de la función.

Note el uso del símbolo * para indicar que se trata de una cantidad aleatoria de argumentos que se guardarán en la tupla con nombre **args**

Por ejemplo:

[12]:
```
# función con una cantidad variable de argumentos
def prom(*args):
    sum = 0
    for i in args:
        # calculo la suma total
        sum = sum + i
    # calculo el promedio
    promedio = sum / len(args)  # en vez len() podría haber puesto un␣
 ↪contador,
                                # pero usar len() es más rápido.
    print('Promedio =', promedio)


prom(156, 23, 23)
print()
prom(2,3,4,5,6)
```

```
Promedio = 67.33333333333333

Promedio = 4.0
```

7.2.2. Caso de una cantidad arbitraria de argumentos de teclado (**kwargs)

Si quisiera hacer la misma operación pero con una cantidad de argumentos de teclado, tendría que usar como argumento de la función, por ejemplo, el nombre ****kwargs**. El doble símbolo ** indica que se trata de argumentos de teclado. Los argumentos pasarán entonces a la función la cual los verá internamente como un diccionario. Por lo cual, los datos serán accesibles a través de un sistema llave-dato. Es decir para una llave determinada corresponde cierto dato.

Ejemplo:

[13]:
```
def porcentaje(**kwargs):
    sum = 0
    for sub in kwargs:
        # get argument name
        sub_nombre = sub
        # get argument value
        sub_nota = kwargs[sub]
        print(sub_nombre, "=", sub_nota)

# envio varios arguemtnos de teclado
porcentaje(matemática=56, física=61, química=73)
print()
```

```
matemática = 56
física = 61
química = 73
```

Por lo cual debemos considerar que hay 4 tipos de argumentos en la llamada de una función en Python:

- Argumentos default (los que indico al construir la función)
- Argumentos posicionales (se indentifican con la variable por la posición)
- Argumentos de Teclado (Son argumentos en los cuales modifico los valores indicados en el default)
- Argumentos que tienen una cantidad variable de elementos (*args y **kwargs)

Veamos ejemplos:

[14]:
```
def add(a,b=5,c=10):
    return (a+b+c)
```

Y podría hacer:

Por lo cual ahora los argumentos son posicionales, es decir identifico cuál es cuál por su posición.

O podría hacer lo siguiente:

[15]:
```
print(add(b=24,c=5,a=12))
```

```
41
```

Con lo cual ahora son argumentos de teclado, donde cada variable es indicada por nombre y valor.

Reglas para los argumentos

- Los argumentos default deben seguir después de los argumentos que no son default.
- Los argumentos de teclado deben seguir después de los agumentos posicionales, cuando se crea y se usa la función.
- Los argumentos de teclado deben indicar un nombre de una variable declarada en la definición de la función. Su orden no es importante.
- Ningúno de los argumentos puede recibir más de un valor.
- Los argumentos default son opcionales al momento de la llamada a la función.

Usando Objetos mutables como argumentos en Pyhton

Al poner un objeto en le arguemento este también queda como objeto del programa que llama a la función. Y esto vale aunque no envíe una variable a ese argumento en el llamado de la función.

Veamos un ejemplo:

[16]:
```
def appendItem(Nombre_del_item, Lista_de_items = []):
    Lista_de_items.append(Nombre_del_item)
    return Lista_de_items


print(appendItem('notebook'))
print(appendItem('PC'))
print(appendItem('Tableta'))
```

```
['notebook']
['notebook', 'PC']
['notebook', 'PC', 'Tableta']
```

La clara ventaja de hacer esto, es que la función **NO** se olvidó de las valores que recogió en las llamadas que se le hizo anteriormente.

7.2.3. El comando RETURN

Este podría devolver más de un valor. La construcción sería de esta forma, ya que la orden return debería indicar todos los valores a devolver

```
def algo():
    ...
    return(a,b)

x,y=algo()
```

Las funciones pueden devolver más de un resultado. En cierta manera violando la idea que uno ha aprendido en análisis matemático. Cuando son varios los resultados este es retornado en forma de tupla.

Por ejemplo:

[17]:
```
def cuadrado_cubo(x):
    x2 = x*x
```

```
    x3 = x2*x
    return(x2,x3)

print("Para x=3",cuadrado_cubo(3))
print("Para x=4",cuadrado_cubo(4))
print("Para x=5",cuadrado_cubo(5))
```

```
Para x=3 (9, 27)
Para x=4 (16, 64)
Para x=5 (25, 125)
```

Note que los resultados están englobados por ( ) indicando que es una tupla.

Las funciones creadas se vuelven métodos de mi programa Python y por lo tanto puedo en tiempo real preguntar por su existencia. Por ejemplo, con la orden dir() que me lista los métodos de un objeto y este comando sin argumentos me indica la composición de mi propio programa python visto como un objeto.

En este caso me sirve para listar las funciones cargadas en la memoria. En esa lista habrá funciones del sistema y las que agregué en las celdas anteriores.

[18]:
```
print(dir())
```

```
['D', 'E', 'In', 'Out', 'X', '_', '_4', '__', '___', '__builtin__',
'__builtins__', '__doc__', '__loader__', '__name__', '__package__',␣
 ↪'__spec__',
'_dh', '_i', '_i1', '_i10', '_i11', '_i12', '_i13', '_i14', '_i15', '_i16',
'_i17', '_i18', '_i2', '_i3', '_i4', '_i5', '_i6', '_i7', '_i8', '_i9', '_ih',
'_ii', '_iii', '_oh', 'a', 'add', 'appendItem', 'b', 'c', 'cm',␣
 ↪'cuadrado_cubo',
'cubo', 'd', 'exit', 'func_cubo', 'func_rara', 'get_ipython', 'open',
'porcentaje', 'prom', 'quit']
```

Puede verse que las funciones cubo, func_cubo, func_rara y X que creamos en este notebook hasta ahora, están como funciones en el sistema Python que estamos corriendo.

7.3. Función Lambda (Lambda expression)

La función lambda es para crear funciones de una línea, y son equivalentes en todos sus aspectos a la función sentencia de Fortran.

En Python se las considera como funciones sin nombre, pero que tienen mucha utilidad en algunos casos.

por ejemplo, si defino

[19]:
```
lambda x: x**2
```

[19]:
```
<function __main__.<lambda>(x)>
```

El sistema me dice que es una función, pero no tiene nombre. Para poder usarla tengo que asginarla a una variable.

[20]:
```
f = lambda x: x**2
# si la llamo con un argumento
print(f(2))
```

```
4
```

Funciona!!!

Y se la puede construir con muchas entradas de datos, no solo una. Por ejemplo:

[21]:
```
J = lambda x, y, z: (x**2 + y**2 + z**2)**0.5
J(1,2,3)
```

[21]:
```
3.7416573867739413
```

Entonces la forma general para crear una expresión Lambda sería:

lambda variables: ***expresión matematica en función de las variables que se ingresaron***

Y el caso más extremo sería ni siquiera darle una variable:

[22]:
```
f1 = lambda: "¿Para qué sirvo?"

print(f1())
```

```
¿Para qué sirvo?
```

7.4. Recursividad

La funciones en Python son recursivas, veamos el ejemplo del factorial.

$n! = n(n-1)! = n(n-1)(n-2)! = n(n-1)(n-2)....1$

[23]:
```
def fact(n):
    if n <= 0:
        return 1
    return n*fact(n-1)


print(fact(5))
print(fact(20))
print(fact(10))
```

```
120
2432902008176640000
3628800
```

7.5. Funciones que están en archivos

Quiero tener en un archivo que se llama ex1, la función siguiente

def f1(x):

```
"""
Este es un ejemplo de la función que devuelve: x**2    - parameter: x
"""
return x**2
```

Lo voy a crear con la celda siguente, ya que al correr la orden: % %writefile ex1.py graba la celda a un archivo.

[24]:
```
%%writefile ex1.py
# Lo que está en esta celda se salvará a un archivo con nombre "ex1.py"
def f1(x):
    """
    Este es un ejemplo de la función que devuelve: x**2
    y parámetro es x
    """
    return x**2
```

```
Overwriting ex1.py
```

[25]:
```
import ex1

# Esto importa un archivo que se llama ex1.py del directorio
# o de los directorios que indique poniendo el camino.
# El hecho de que muchos usuarios escriben y comparten funciones
# es lo que impulsó a Python en el área científica.

print(ex1.f1(4))  # ex1 es el archivo, f1() es la
                  # función que está dentro del archivo
```

```
16
```

[26]:
```
from ex1 import f1
print(f1(3))

#Cargo de ex1 SÓLO la función f1
```

```
9
```

[27]:
```
from ex1 import * # Aquí cargo todas las funciones. Lo cual es una
                  # mala idea se cargan todas las funciones con su
                  # nombre y pueden sobreescribir otros nombres
                  # que estaba usando para las funciones

print(f1(4))
```

```
16
```

[28]:
```
import ex1 as tt
print(tt.f1(10))

# Ahora cargo todas las funciones de ex1, pero las tengo
# que llamar con el prefijo tt en el nombre, es decir la
```

```
# función f1() es ahora tt.f1
```

```
100
```

Puedo tener más de una función en el archivo que importo, por ejemplo:

[29]:
```
%%writefile ex2.py
# Lo que está en esta celda se salvará a un archivo con nombre "ex3.py"

def f1(x):
    """
    Este es un ejemplo de la función que devuelve: x**2
    y parámetro es x
    """
    return x**2

def f2(x):
    """
    Este es un ejemplo de la función que devuelve: x**2
    y parámetro es x
    """
    return x**3
```

```
Overwriting ex2.py
```

[30]:
```
import ex2

print(ex2.f1(4))

print(ex2.f2(4))
```

```
16
64
```

7.6. Importando bibliotecas

No todas las funciones de Python que lo han hecho tan famoso y requerido están en el lenguaje, sino que muchas de las más usadas son externas a Python. Para usar estas funciones hay que "importarlas". Por lo cual hay que indicarle a nuestro programa que las cargue y de esta manera adquiera nuevas capacidades.

Incluso esto es válido para el caso de que se necesite usar funciones matemáticas trascedentes, ya que el Python en su versión original no las trae cargadas. Por ejemplo, Python no tiene funciones intrínsecas de matemática como el Fortran.

Por ejemplo:

calcular: print(sin(3.)) daría error, la funciones trigonoméricas no están construidas por default dentro de Python.

[31]:
```
#print(sin(3.))  # si se elimina el # y se corre la celda da error.
```

Pero en cambio si importo la funciones matemáticas que están en la bibliotecas **math** ya lo puedo usar. Lo haríamos de la siguiente manera:

[32]:
```
import math
print(math.sin(3.))
```

```
0.1411200080598672
```

[33]:
```
help(math.sin)
```

```
Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
```

[34]:
```
print(math.pi)
```

```
3.141592653589793
```

CAPITULO 8

Iteradores - Generadores

8.1. Iteradores

8.1.1. Definición

Cuando un proceso se debe repetir muchas veces conviene usar Iteradores.

Por ejemplo si hago:

for x in algo: **Lo que se repite va acá**

la variable **algo** es un iterable mientras **x** es un iterador.

Por ejemplo si tengo una lista:

Puedo iterar sobre la lista.

[1]:
```
lista = [1,2,3]
for elemento in lista:
    print(elemento)
```

```
1
2
3
```

Un secuencia es un iterable que tenga un orden, por lo cual puede ser una lista, una tupla, un set, un diccionario, un string o directamente bytes.

Un iterable es un objeto que puede recordar cual fue el último valor que entregó una vez que la iteración fue activada.

Por ejemplo, hago un loop sobre la variables que en este caso son una tupla:

[2]:
```
for elemento in ('azul','rojo', 'verde', 'amarillo'):
    print(elemento)
```

```
azul
rojo
verde
amarillo
```

O sobre las letras de una palabra:

[3]:
```
for letra in 'Computación':
    print(letra)
```

```
C
o
m
p
u
t
a
c
i
ó
n
```

Inluso podemos iterar sobre Bytes.

[4]:
```
for bytes in b'Binario':
    print(bytes)
```

```
66
105
110
97
114
105
111
```

Pregunta para el lector ¿Qué son esos números?

Pero no puedo iterar sobre los díigitos de un número entero, pero si lo convierto en un string si puedo.

[5]:
```
I = 1093647
I_texto = str(I)

for letra in I_texto:
    print(int(letra))

# otra forma de hacerlo mucho más compacta sería
print('')
digits = [int(d) for d in str(I)]
for digito in digits:
    print(digito)
```

```
1
0
9
3
6
4
7
```

```
1
0
9
3
6
4
7
```

Los *set´s* también son iterables, pero recordemos que son elementos no tienen orden, ni están numerados. Veamos el ejemplo anterior pero ahora con estructura de set:

[6]:
```
for elemento in {'azul','rojo', 'verde', 'amarillo'}:
    print(elemento)
```

```
azul
verde
rojo
amarillo
```

8.1.2. Desde el punto de vista del Objeto

iter dispara el proceso pero los items subsiguientes hay que pedirlos con el comando **next** hasta que los elementos se acaban y da error.

[7]:
```
lista = ['gato','perro','tortuga','canario']

i_pepe=iter(lista)   # <- activo la iteracción
print(next(i_pepe))
print(next(i_pepe))
print(next(i_pepe))
print(next(i_pepe))
```

```
gato
perro
tortuga
canario
```

También es posible crear una estructura de control usando **try/except** como vimos antes. Pero necesito usar en este caso las órdenes **iter/item** para generar el conjuto de los elementos en los cuales se itera.

Ejemplo:

[8]:
```
nums = [1,2,3,4]
i_nums = iter(nums)

while True:
    try:
        item = next(i_nums)
        print(item)
    except StopIteration:
        break
```

```
```

```
1
2
3
4
```

En este caso la activación del error al llamar a un valor cuando ya se acabaron activa el **except** y obliga a terminar la iteración.

8.2. Generadores

8.2.1. Definición

Pero como hacemos en el caso de que la iteración deba hacerse con datos que tengo en un archivo el cual por su tamaño no es posibile cargarlo en la memoria de la computadora? O en el caso de que se busque una opción posible dentro de infinitas posibilidades ¿cómo se maneja esta situación en Python?

Para ello se usan los generadores ya que un método for-loop no serviría.

Un generador es una función que actúa como los iteradores y "genera" los elementos para ser utilizados en el loop. Es decir que trabaja con una iteración donde los elementos de la iteración se solicitan uno a uno. Y de esta manera no consume tantos recursos de memoria. Con otras palabras, el generador me va dando de a uno los valores en la medida de que se los vaya pidiendo.

¿Cómo funcionan estos generadores? Como indicamos son funciones y van devolviendo los valores de la iteración en demanda. Veamos un ejemplo abstracto:

[9]:
```
def f():
    yield 1
    yield 2
    yield 3
```

Note que ahora la función no termina su ejecución con un "return", sino con "yield" y si la ejecuto:

[10]:
```
f()
```

[10]:
```
<generator object f at 0x7fa6a029fba0>
```

Me dice que la funcion f() es un generador, así que puedo iterar con ella:

[11]:
```
for x in f():
    print(x)
```

```
1
2
3
```

Veamos un ejemplo no tan abstracto. Calculemos los cuadrados de una lista, pero de uno a uno.

[12]:
```
# defino el generador
def cuadrados(numeros):
    for i in numeros:
        yield (i*i)

# armo la lista de números que voy a procesar
lista=[1,2,3,4]

# construyo la función que genera los números
# esta orden construye el generador, pero no la ejecuta!!!
mi_lista = cuadrados(lista)

print(mi_lista)
print("")

# y ahora la recorro
for numero in mi_lista:
    print(numero)
```

```
<generator object cuadrados at 0x7fa6a029fdd0>

1
4
9
16
```

[13]:
```
def numeros_primos():
    yield 2 # devuelvo 2 como el primer primo
    primo_cache = [2]

    for n in range(3,1000000):

        es_primo=True

        for p in primo_cache:
            if n%p == 0:
                es_primo = False
                break
        if es_primo:
            primo_cache.append(n)
            yield n

for p in numeros_primos():
    print(p)
    if p> 10:
        break
```

```
2
3
```

```
5
7
11
```

8.2.2. Expresión Generadora o "Generator Expression"

Son una manera más corta de construir generadores. Son un pariente de los List Comprehension y en cierta medida tiene notación de tupla, pero no tiene nada que ver con estas últimas.

Se escriben entre parentésis (las List comprehension entre corchetes) con una gramática similar.

Veamos una:

[14]:
```
def natural_numbers(i=1):
#  i = 0
  while True:
    yield i
    i += 1

#for num in natural_numbers():
#  print(num)
```

Utilizando las mismas reglas de un List Comprehension podria ponerle un final en su propia construcción asi:

[16]:
```
cubos = (x**3 for x in range(5))

for x in cubos:
    print(x)
```

```
0
1
8
27
64
```

CAPITULO 9

Arreglos en Python - Biblioteca Numpy

9.1. Numerical Python o NumPy

NumPy es una biblioteca para realizar cálculo en el campo del algebra lineal. Originalmente fue derivado de un paquete de rutinas hecho para Fortran, conocido como BLAS.

NumPy es un paquete fundamental para utilizar python en computación de cálculos científicos. Esta es una biblioteca que provee de objetos equivalentes a arreglos multidimensionales y todo un conjunto de rutinas para una operación rápida en los campos de la matemáticas, lógica, manipulación de formas, ordenamientos, entre/salida de datos, Trasformada discreta de Fourier, álgebra lineal básica, operaciones estadísticas básicas, simulaciones con números al azar y otros muchos temas.

Es una biblioteca más importante en el uso científico de python.

Se puede conseguir muchas mas información en www.numpy.org

9.2. Numpy en un notebook

9.2.1. Importando NumPy

Como toda biblioteca externa se debe cargar para que pueda ser usada por el notebook. El Python de por si no tiene esta biblioteca en memoria, por lo cual, debe ser el usuario el que decida utilizarla y para ello debe "importarla". Al hacerlo cargará no sólo los algoritmos que lleva programados, sino además las Clases con las cuales se definene las variables propias de la biblioteca. Los objetos que se usan en NumPy son básicametne arreglos que mantienen cierta similitud con los vectores y matrices del álgebra.

Primero tenemos que cargar la biblioteca, es bastante popular llamar "np" a la biblioteca numPy. Aunque no es obligatorio llamarla como "np" ya es un estandard reconocido.

[1]:
```
import numpy as np
```

Este comando carga la biblioteca NumPy, pero con un cambio de nombre, la traducción literal del comando sería "Carga NumPy como np". desde esta manera Numpy de ahora en adelante se llamará np en mi programa. Es normal renombrarla como "np", aunque nada evita que se pudiera haber elegido cualquier otro nombre.

9.2.2. La clase de los arreglos (array)

NumPy usa sus propias variables, es decir como Python es un lenguaje orientado a objetos se definió una clase para generar objetos con las propiedades necesarios para hacer álgebra vectorial. Es decir uso de vectores, matrices, cubos, etc, con mucha eficiencia y respetando propiedades del álgebra. Básicamente NumPy utiliza arreglos que son propios de su biblioteca.

Veamos el uso de la función array( ) que crea objetos NumPy a partir de variables de Python

Puedo crear un arreglo numpy tomándolo de una lista o tupla, o creándolos nuevos.

Veamos el ejemplo de usar una lista y con ella crear un arreglo NumPy.

Para ello uso la función array de NumPy, y como a NumPy lo llamé "np", a la función la tengo que llamar np.array( )

El arreglo NumPy creado ya no es una lista (aunque se le parece mucho), tiene otras propiedades.

[2]:
```
a_lista = [1,2,3,4,5,6]

a = np.array(a_lista)

print(a)
```

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

Notar que ahora no tenemos las "," típicas de una lista.

O podría hacer directamente

[3]:
```
a = np.array([1,2,3,4,5,6])
```

Si pregunto el tipo de variable que tengo ahora

[4]:
```
print("Es de la clase:", type(a))
```

```
Es de la clase: <class 'numpy.ndarray'>
```

El método np.array() funciona también conviertiendo tuplas a arreglos:

[5]:
```
b = np.array((1,2,3))

print(b)
print("Es de la clase:",type(b)) # Ya NO es una Tupla.... y además
                                 # desaparecen las comas de la lista o tupla
```

```
[1 2 3]
Es de la clase: <class 'numpy.ndarray'>
```

[6]:
```
# Creo una lista
L = [1, 2, 3, 4]
# Convierto mi lista a un arreglo NumPy
a = np.array(L)

print("La lista L es del tipo: ",type(L))
```

```
print("a es de tipo:",type(a))
print("El arreglo a es del tipo:",a.dtype)
print("")
print(L)
print(a)
```

```
La lista L es del tipo:  <class 'list'>
a es de tipo: <class 'numpy.ndarray'>
El arreglo a es del tipo: int64

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

El dtype es un comando de NumPy. Me indica que el arreglo "a" no es una lista de Python. Como "a" ahora es un objeto NumPy puedo utilizar las funciones de ese objeto y dtype es un método de esa clase. dtype me indica el tipo de arreglo dentro de las posibilidades de los arreglos de esta biblioteca. En este caso es tipo Integer*8 (64 bits).

A diferencia de las listas los arreglos NumPy engloban a variables que son del mismo tipo. Es decir que todo el arreglo Numpy es del mismo tipo de variable (Real*8, Int*8, complejo, etc)

Una vez definido, todos los agregados se modifican para que sean válidos en el tipo de arreglo final.

[7]:
```
Lista2=[1,2,22.] # <-- 1 y 2 enteros, 22. es real
Lista_numpy=np.array(Lista2)

print(Lista_numpy)
```

```
[ 1.  2. 22.]
```

Todo el arreglo terminó siendo real.

Los arreglos NumPy son objetos diseñados para que los cálculos son rápidos y eficientes.

Para ver la ventaja real de esta biblioteca usaremos el comando del notebook %timeit (que es una función del notebook, no es una orden de Python) que me permite tomar tiempos.

range es una función de python.

arange es una función que se agrega porque viene con numPy. Esta última crea los mismos valores de la lista, pero ahora en array NumPy

[8]:
```
L = range(1000)  # Crea una lista

%timeit for L in range(1000):    A=L**2

print('Pero usando NumPy')

A = np.arange(1000)  # Crea un arreglo Numpy

%timeit A2 = A**2
```

```
26 µs ± 857 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
Pero usando NumPy
591 ns ± 8.14 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Moraleja 1:

La idea que hay que tener presente cuando se usa NumPy, es que se va a trabajar con arreglos como variables. No hay que pensar en escribir código que apunte a algoritmos donde se trabaja con los elementos de a uno en un arreglo. El código se escribe pensando las operaciones matemáticas que se realizan sobre vectores y matrices. Veremos esto con detalle durante el curso.

Los arreglos NumPy llevan consigo al ser objetos, atributos y funciones. Estas funciones en particular son mucho más eficientes que las originales de Python. Probemos creando un arreglo usar primero la función que busca el mínimo valor de Python (min(a)) y luego lo que hace el mismo trabajo pero con la función que reside en el objeto NumPy (a.min()). Noten que tiene el mismo nombre pero que se las llama para su uso de forma diferente.

[9]:
```
# construyo un vector de 10.000.000 números con primer valor 1000 y último␣
 ↪0, con paso -0.00001

a = np.arange(1000,0,-0.0001)

# veo que tengo en a
print("Largo de a: ",len(a))

%timeit min(a)
%timeit a.min()
```

```
Largo de a:  10000000
301 ms ± 24.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.88 ms ± 274 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Moraleja 2:

Hay que usar las funciones que vienen en las biblioteca que son especifícas para resolver un tema particular, en el caso de álgebra vectorial (como en este último ejemplo) es prácticamente obligatorio usar las funciones de los objetos NumPy.

Pueden ser de varias dimensiones 1D, 2D, 3D, . . .

[10]:
```
a = np.array([1,2,3,4,5,6])
b = np.array([[1,2],[1,4]])
c=np.array([[[1,2], [2,3]], [[3,4], [4,5]]])

print("Imprimo el atributo shape")
print( a.shape, b.shape, c.shape)
print("")
print("Pero si me fijo el len()")
print(len(a),len(b), len(c))
```

```
Imprimo el atributo shape
(6,) (2, 2) (2, 2, 2)

Pero si me fijo el len()
6 2 2
```

El atributo shape me cuenta de las dimensiones del arreglo, en cambio la función len( ) me cuenta de su largo.

El comando shape me devuelve una **tupla** con la información, en cambio len( ) me devuelve un número.

Veamos como imprimimos el array c y sus componentes ya que este arreglo es trimensional.

[11]:
```
print(c[0])
```

```
[[1 2]
 [2 3]]
```

[12]:
```
print(c[0][0])
```

```
[1 2]
```

[13]:
```
print(c[0][0][0])  #<-- recién aquí me devuelve un número
```

```
1
```

Si tengo varias dimensiones y quiero saber la cantidad total de elementos es conveniente imprimir el atributo **size**.

[14]:
```
b.size
```

[14]: 4

Pero estrictamente la dimensión me la da el atributo ndim

[15]:
```
print (a.ndim, b.ndim, c.ndim )
```

```
1 2 3
```

La forma como NumPy numera los ejes, se conoce como el estilo "C" (por el lenguaje C que lo hace de esa manera) y es diferente al estilo Fortran. Aunque siempre es más cómodo visualizar las distintas dimensiones como *arreglos de arreglos de otros arreglos...*

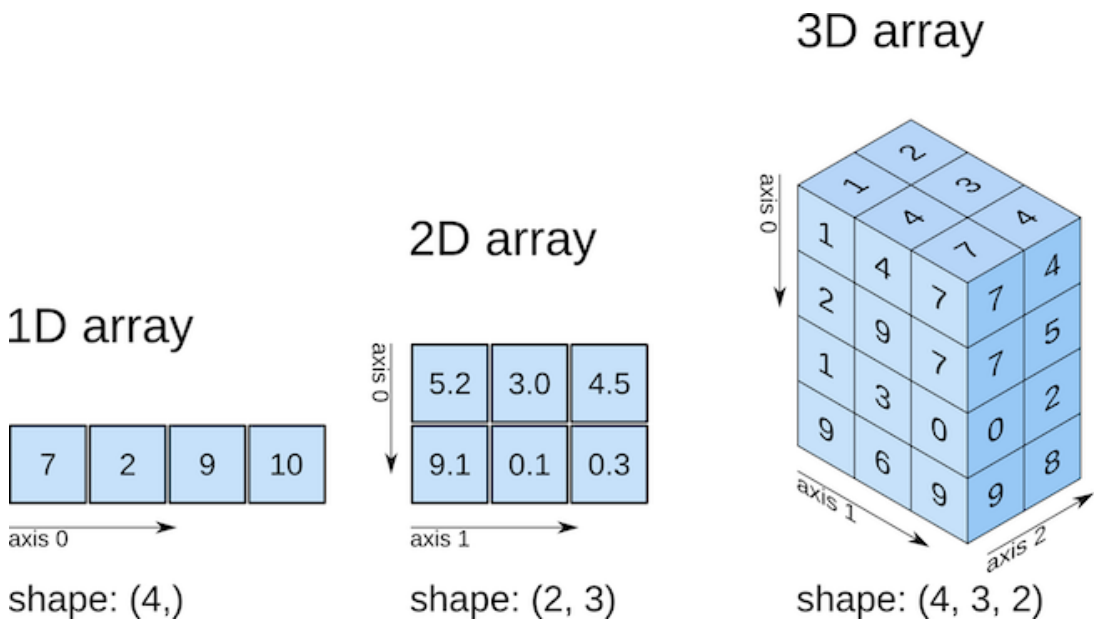


Figura 9.1: dimensiones

Crédito del dibujo:https://www.tomasbeuzen.com/python-programming-for-data-science/chapters/chapter5-numpy.html

---

La biblioteca NumPy trae funciones asociadas.

Las puedo buscar en el manual (www.numpy.org), veamos algunos ejemplos:

[16]:
```
a = np.array([1,2,3,4,5,6,7])
```

Por ejemplo, probemos con dos, la que calcula promedios de un array a.mean( ) y la que me encuentra el valor máximo a.max( )

[17]:
```
print("Los valores de a son:",a)
print("El promedio es:",a.mean())
print("El valor máximo es:",a.max())
```

```
Los valores de a son: [1 2 3 4 5 6 7]
El promedio es: 4.0
El valor máximo es: 7
```

mean( ) y max( ) son métodos (funciones) de la clase del arreglo, por eso se los usa con paréntesis ( ). Entre estos paréntesis es posible agregar argumentos, que se detallan en los manuales.

[18]:
```
print(a.mean) # Esto así imprime de que trata la funcion no su resultado
```

```
<built-in method mean of numpy.ndarray object at 0x108425050>
```

Veamos algunos de los argumentos que puedo solicitar para que el cálculo se realize con alguna variante. En este caso voy a especificar sobre que eje quiero el promedio.

[19]:
```
print ("Imprimo b: ",b)
```

```
# Promedio sobre todo el arreglo
print ("Imprimo el promedio de b: ",b.mean())

# Promedio sobre el primer eje (columnas)
print ("Imprimo el promedio de las columnas de b: ",b.mean(axis=0))

# Promedio sobre las filas, se sobreentiende que es "axis=1"
print ("Imprimo el promedio de las filas de b: ", b.mean(1))
```

```
Imprimo b:  [[1 2]
 [1 4]]
Imprimo el promedio de b:  2.0
Imprimo el promedio de las columnas de b:  [1. 3.]
Imprimo el promedio de las filas de b:  [1.5 2.5]
```

9.2.3. Creando arreglos usando sentencias específicas

La idea es que el arreglo reciba una cantidad de elementos creados en una forma automática y que cumplen reglas que se especifican como bucles. Es decir con un inicio, un final y un paso. Donde este último no tiene que ser obligtoriamente lineal. Por ejemplo, podríamos tener un paso geométrico, logarítmico, exponencial, etc.

También veremos que existen órdenes que copian la forma de otro arreglo, pero no sus elementos, sólo su forma, es decir su tamaño y dimensión.

[20]:
```
print(np.arange(10))
```

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

[21]:
```
print(np.linspace(0, 1, 10)) # Comienzo, final, y número de puntos
```

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

Probablemente estos puntos no fueron los que inicialmente se pensó, quizás debería llamarse al comando de esta manera.

[22]:
```
print(np.linspace(0, 1, 10, endpoint=False)) # No incluimos el punto final,␣
 ↪o lo hacemos hasta 11.
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

En este caso no incluímos el punto final entonces empieza en 0 y termina 0.9 con paso 0.1. También podríamos haber pedido que sean 11 números y hubiera dado este mismo resultado.

Para algún caso espcifico podría pedir que se genero un arreglo con "0" todos los elementos, para eso existe el comando np.zeros()

[23]:
```
print(np.zeros(2)) # Lleno con ceros
```

```
[0. 0.]
```

[24]:
```
print(np.zeros((2,2))) # es un arreglo de dos dimensiones lleno con ceros
```

```
[[0. 0.]
 [0. 0.]]
```

O pedir que mi arreglo sea similar en forma y tamaño con otro que ya existe pero que solo contenga al número "1" en todos sus elementos.

[25]:
```
print(np.ones_like(a)) # Esto es muy bueno, creo un arreglo (o tupla)
                       # usando las propiedades de otro que ya tengo
```

```
[1 1 1 1 1 1]
```

Además de las siguientes variantes:

[26]:
```
print(np.zeros_like(a)+3) # Me sirve para llenar con otro valor que no son␣
 ↪ceros o 1's
```

```
[3 3 3 3 3 3]
```

[27]:
```
print(np.ones_like([1,2,3]))
```

```
[1 1 1]
```

Puedo crear arreglos vacíos de números pero con la forma final que deben tomar.

[28]:
```
a22=np.empty([2, 2])
a11=np.empty([10])
```

[29]:
```
print(np.logspace(0, 2, 10)) # va de 10**comienzo hasta 10**final y calculo␣
 ↪10 valores
```

```
[  1.           1.66810054   2.7825594    4.64158883   7.74263683
  12.91549665  21.5443469   35.93813664  59.94842503 100.        ]
```

Una manera muy típica de trabajar en python es primero crea el array como unidimensional y luego darle forma con el comando **reshape( )**, que lo vimos en Fortran 90/95.

[30]:
```
a = np.array([1,2,3,4,5,6])
b = a.reshape((3,2)) # Este comando no cambia la "forma" de  a
print(a)
print('')
print(b)
```

```
[1 2 3 4 5 6]

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

Mientras que la función **ravel( )** lo vuelve unidimensional.

[31]:
```
print(b.ravel())
```

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

OJO - CUIDADO, los arreglos son objetos y cuando se reasignan no se copian, entonces apuntan a la misma memoria

[32]:
```
b = a.reshape((3,2))
c = a.reshape((3,2))
print(a.shape, b.shape)
```

```
(6,) (3, 2)
```

[33]:
```
print(b)
print("")
b[1,1] = 100 # modifico un valor del arreglo
print(b)
```

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

[[  1   2]
 [  3 100]
 [  5   6]]
```

[34]:
```
print(a) # !!! a y b son el mismo objeto comparten el mismo espacio de␣
 ↪memoria,
         # es decir apuntan a los mismos valores
```

```
[  1   2   3 100   5   6]
```

[35]:
```
print(b[1,1],a[3]) # el mismo valor
```

```
100 100
```

[36]:
```
c = a.reshape((2,3)).copy() # Esta es la solución
```

[37]:
```
print(a)
print('')
print(c)
```

```
[  1   2   3 100   5   6]

[[  1   2   3]
 [100   5   6]]
```

[38]:
```
c[0,0] = 8888
print(a)
print(c)
```

```
[  1   2   3 100   5   6]
[[8888    2    3]
 [ 100    5    6]]
```

Números al azar (random) en Numpy

Tengo funciones específicas ya programadas en NumPy para realizar la tarea.

[39]:
```
ran_uniform = np.random.rand(5) # Entre 0 y 1
ran_normal = np.random.randn(5) # Gausiana promedio 0 varianza 1
print("Números con distribución Uniforme: ",ran_uniform)
print ('')
print ("Números con distribución Gaussiana: ",ran_normal)
print ('')
ran_normal_2D = np.random.randn(5,5) # Gausiana promedio 0 varianza 1
print("Números con distribución Gaussiana, pero ahora en una matriz:␣
 ↪\n\n",ran_normal_2D)
```

```
Números con distribución Uniforme:  [0.82185235 0.1270517  0.02500347 0.
 ↪65405953
0.00564375]

Números con distribución Gaussiana:  [ 0.3330643   0.64555966  0.3741522
-0.19994844  0.46044796]

Números con distribución Gaussiana, pero ahora en una matriz:

 [[ 0.30121992 -1.07022067 -0.73436146 -1.20901705  0.92315628]
 [-0.80165365 -1.40490617 -0.22113263 -0.47695143 -0.93828187]
 [ 1.108599   -0.34993195 -0.4843905   0.95110066 -0.40953384]
 [ 1.33012431 -0.33259819  1.59435379 -0.18255226  0.86071945]
 [-0.44859864  0.47832475 -0.23067724 -0.63039662  0.52668981]]
```

Si en la celda siguiente elimino el # en la línea y corro la celda me da como resultado un "help" del comando. Estas ayudas que se agregan en las funciones son los comentarios que se agregaron en el código de la función como docstrings.

[40]:
```
#np.random.randn?
```

Slicing (cortando fetas)

Es tal como hemos visto con los comandos originales de Python. El rabanado de un objetos de Numpy se hace con los mismos comandos.

[41]:
```
a = np.arange(10)
print(a)
```

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

[42]:
```
print(a[1:8:3])
```

```
[1 4 7]
```

Usando arreglos y máscaras

Puedo en NumPy cambiar varios números en una sólo línea con un lista de estos elementos. Veamos un ejemplo:

[43]:
```
print(a)
a[[2,4,6]] = -999   # Puse como ínidices en el arreglo a, una lista de␣
 ↪elementos 2,4,6
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
[   0    1 -999    3 -999    5 -999    7    8    9]
```

Veamos como construir un arreglo NumPy donde sus elementos son variables lógicas

[44]:
```
a = np.random.randint(0, 100, 20) # min, max, N
print(a)
```

```
[10 85 44 72 22 22 64 39 43  3 94 32 21 28 89 93 27 33 70  4]
```

[45]:
```
a < 50
```

[45]:
```
array([ True, False,  True, False,  True,  True, False,  True,  True,
        True, False,  True,  True,  True, False, False,  True,  True,
       False,  True])
```

La respuesta al operador de relación fue un arreglo con valores booleanos, es decir es la respueta si es verdadera o falsa la pregunta que se realizó en cada elemento.

Veamos esto con más detalle y como usarlo en un caso real:

9.2.4. Operaciones con arreglos - Filtrando con máscaras

Utilizo el arreglo a del ejemplo anterior:

[46]:
```
a
```

[46]:
```
array([10, 85, 44, 72, 22, 22, 64, 39, 43,  3, 94, 32, 21, 28, 89, 93, 27,
       33, 70,  4])
```

Y lo uso para realizar operacion matemáticas simples:

[47]:
```
a + 1
```

[47]:
```
array([11, 86, 45, 73, 23, 23, 65, 40, 44,  4, 95, 33, 22, 29, 90, 94, 28,
       34, 71,  5])
```

Hay que recordar que ahora la variable para los cálculos es todo el arreglo. Y podemos hacer operaciones bastante más complejas, como la que sigue:

[48]:
```
a**2 + 3*a**3
```

[48]:
```
array([   3100, 1849600,  257488, 1124928,   32428,   32428,  790528,
        179478,  240370,      90, 2500588,   99328,   28224,   66640,
       2122828, 2421720,   59778,  108900, 1033900,     208])
```

Creo un arreglo más pequeño:

[49]:
```
a = np.arange(10)
print("a es:",a)
```

```
a es: [0 1 2 3 4 5 6 7 8 9]
```

Y realizo una operación más compleja y asigno el resultado a otro arreglo que llamo "b"

[50]:
```
b = a**2 + (a+1)**2
print("b es:",b)
```

```
b es: [  1   5  13  25  41  61  85 113 145 181]
```

Hago otra operación a la que asigno el resultado al arreglo "c"

[51]:
```
c = (a+2)**2
```

[52]:
```
print("b es:",b)
print("c es:",c)
```

```
b es: [  1   5  13  25  41  61  85 113 145 181]
c es: [  4   9  16  25  36  49  64  81 100 121]
```

Podría entonces preguntar que valores están repetidos en el mismo lugar en los arreglos b y c

[53]:
```
print(b == c)
```

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

Guardo el resultado en el arreglo "máscara"

[54]:
```
mascara = b==c
print("mascara es:", mascara)
```

```
mascara es: [False False False  True False False False False False False]
```

¿Y si la máscara la uso para filtrar el vector? ¿Cómo funciona eso?

Es decir pido que imprima b[mascara], pero mascara ahora funciona como un arreglo de los índices de b al poner de esa manera el comando. Sólo pasarán el filtro los elementos verdaderos.

[55]:
```
print("b es:",b)
print("b filtrado por la mascara es: ",b[mascara])
```

```
b es: [  1   5  13  25  41  61  85 113 145 181]
b filtrado por la mascara es:  [25]
```

También podría haber usado un expresión más compleja

[56]:
```
mascara= b < 15
print(mascara)
```

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

Podría entonces usar el arreglo máscara para filtrar los valores que cumplen con la condición solicitada.

[57]:
```
d=b[mascara]
print(d)
```

```
[ 1  5 13]
```

Incluso podría haber hecho todo esta operación en un sólo comando:

[58]:
```
print(b>c)
print(a[b>c])
```

```
[False False False False  True  True  True  True  True  True]
[4 5 6 7 8 9]
```

Cuidado que esto último lo puedo hacer porque a,b y c tienen el mismo tamaño y los elementos se corresponden en la numeración

NumPy maneja casi todas las expresiones matemáticas, log, trigonométricas, etc

[59]:
```
a = np.arange(18)
print("a es:",a)
print("")
print("El log10 de a es:",np.log10(a))
```

```
a es: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]

El log10 de a es: [      -inf 0.         0.30103    0.47712125 0.60205999
0.69897
 0.77815125 0.84509804 0.90308999 0.95424251 1.         1.04139269
 1.07918125 1.11394335 1.14612804 1.17609126 1.20411998 1.23044892]

/var/folders/3c/tj_0cb8x1xd_hwlkqgz3q1lr0000gn/T/ipykernel_76045/3913186439.
 ↪py:4
: RuntimeWarning: divide by zero encountered in log10
  print("El log10 de a es:",np.log10(a))
```

Notar que hubo un error al sacar el log10 de 0. Pero el programa siguió corriendo.

Ese error fue notificado en el cartel que sale con el "RuntimeWarning".

Podemos encontrarnos elementos que dieron errores y estos pueden ser: -inf, +inf, y NaN.

9.2.5. Operaciones con vectores y matrices

Multiplicación de Matrices

Utilizamos el método np.matmul(), que se usa np.matmul(primera matriz, segunda matriz), en donde multiplica la primera matriz por al segunda matriz.

[60]:
```
a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]

c = np.matmul(a, b)

print("c es:",c)
```

```
c es: [[4 1]
 [2 2]]
```

[61]:
```
a = [[1, 0], [0, 1]]
b = [1, 2]
c= np.matmul(a, b)

print(c)

c= np.matmul(b, a)
print(c)
```

```
[1 2]
[1 2]
```

[62]:
```
a = np.arange(1000000).reshape(1000,1000)
b = np.ones_like(a)
print(a)
print("")
print(b)
```

```
[[     0      1      2 ...    997    998    999]
 [  1000   1001   1002 ...   1997   1998   1999]
 [  2000   2001   2002 ...   2997   2998   2999]
 ...
 [997000 997001 997002 ... 997997 997998 997999]
 [998000 998001 998002 ... 998997 998998 998999]
 [999000 999001 999002 ... 999997 999998 999999]]

[[1 1 1 ... 1 1 1]
 [1 1 1 ... 1 1 1]
 [1 1 1 ... 1 1 1]
 ...
 [1 1 1 ... 1 1 1]
 [1 1 1 ... 1 1 1]
 [1 1 1 ... 1 1 1]]
```

[63]:
```
c=np.matmul(a,b)
print(c)
```

```
[[   499500    499500    499500 ...    499500    499500    499500]
 [  1499500   1499500   1499500 ...   1499500   1499500   1499500]
 [  2499500   2499500   2499500 ...   2499500   2499500   2499500]
 ...
 [997499500 997499500 997499500 ... 997499500 997499500 997499500]
 [998499500 998499500 998499500 ... 998499500 998499500 998499500]
 [999499500 999499500 999499500 ... 999499500 999499500 999499500]]
```

9.3. Solución de un sistema lineal de ecuaciones

A = floor(random.rand(4000,4000)*20-10) <– genera una matriz de 4000 x 4000 valores llena con números al azar

[64]:
```
A = np.random.rand(4000,4000)
print("Imprimo A")
print (A)

b = np.floor(np.random.rand(4000,1)*20-10)
print("")
print("Imprimo b")
print(b)

# Resolvamos Ax = b usando el comando NumPy para la tarea:
x = np.linalg.solve(A,b)

print("")
print("Imprimo el resultado")
print(x)
```

```
Imprimo A
[[0.10241764 0.06761673 0.79440504 ... 0.69839326 0.76543895 0.02907066]
 [0.78814087 0.23842683 0.26979685 ... 0.75872843 0.85152487 0.36487773]
 [0.38100369 0.83793953 0.9095295  ... 0.64496379 0.98972599 0.36759471]
 ...
 [0.76055677 0.71622225 0.75253981 ... 0.37226835 0.90336081 0.7316975 ]
 [0.78682505 0.19385855 0.4462108  ... 0.47510662 0.22346224 0.69229962]
 [0.78473259 0.93406629 0.11381045 ... 0.75953764 0.05511017 0.51590671]]

Imprimo b
[[ 4.]
 [ 0.]
 [ 0.]
 ...
 [ 8.]
 [ 0.]
 [-1.]]

Imprimo el resultado
[[ -54.62858949]
 [ -20.73700608]
 [  54.50332778]
 ...
 [  88.91263339]
 [-120.681328  ]
 [   7.0825793 ]]
```

9.4. Broadcasting

El broadcasting es la manera que tiene Python de compensar operaciones elemento a elemento entre arreglos de diferentes tamaños.
La situación más simple es cuando se realiza una operación entre un arreglo por un escalar. El escalar en virtualmente repetido todas las veces necesarias para repetir el tamaño del escalar.

Se suele considerar que el broadcasting es un fuerte consumidor de memoria de la computadora.

[65]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

[65]:
```
array([2., 4., 6.])
```

En este caso hubiese sido similar a que conviertiera b en b=np.array([2.0, 2.0, 2.0])

[66]:
```
a = np.array([1.0, 2.0, 3.0])
b = b=np.array([2.0, 2.0, 2.0])
a * b
```

[66]:
```
array([2., 4., 6.])
```

[67]:
```
a = np.array([[0.0], [10.0], [20.0], [30.0]])
b = np.array([1.0, 2.0, 3.0])
a+b
```

[67]:
```
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```

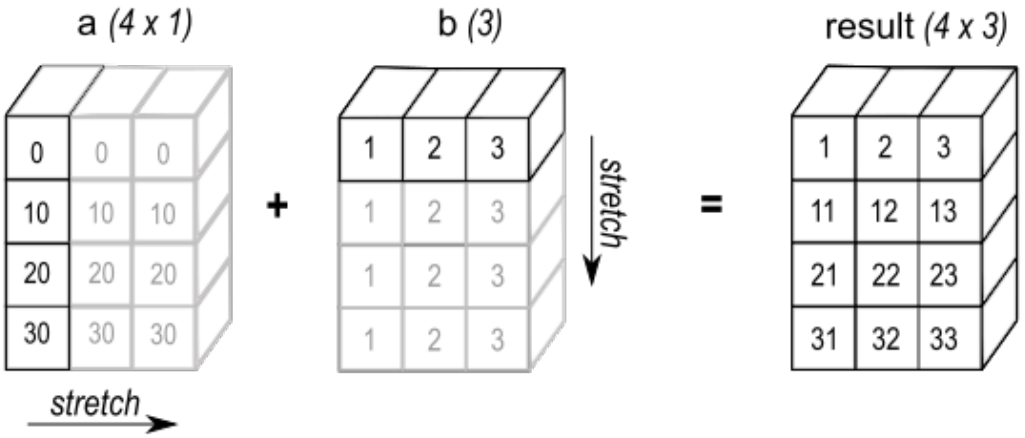


Figura 9.2: Arreglos

Moraleja: Los arreglos sirven para hacer álgebra vectorial, pero NO son vectores, ni matrices...

No tienen las mismas propiedades...

El broadcast sólo se puede realizar si:

- Los arreglos son iguales o
- uno de ellos tiene dimensión 1 arreglos son iguales

Forma de hacer las cosas con NumPy

Veamos como hay que razonar para usar NumPy y de paso calculemos PI (una vez más...)

Esta vez usando series de Taylor. En este caso arctan(1) como hizo en su momento Leibniz

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

Veamos un la forma de plantearlo en Python, y usando NumPy:

Por lo que calculo la serie de dos maneras, primero estilo Fortran: término por término de la serie mientras los sumo.

La segunda manera en modo NumPy, todos los términos al mismo tiempo, creando un vector de términos y que luego sumo.

La función pow(x, y) is igual a hacer el cálculo x^y, y funciona mejor que $x**y$, porque permite valores fracionarios y negativos de y.

[68]:
```
# Estilo Fortran
n = 10000000
total = 0
for k in range(n):
    total = total + pow(-1,k)/(2*k+1.0)
print("Cálculo un elemento y lo voy sumando              :",total*4)
```

```
Cálculo un elemento y lo voy sumando              : 3.1415925535897915
```

[69]:
```
#Estilo NumPy
k=np.arange(n)
term=pow(-1,k)/(2*k+1.0)
total=term.sum()
print("Todos los términos al mismo tiempo y sumo al final:", total*4)
```

```
Todos los términos al mismo tiempo y sumo al final: 3.1415925535897977
```

Noten que la forma NumPy es mucho más veloz.

9.5. Resumen de lo importante para usar NumPy

- Python con NumPy es muy bueno, pero hay que pensar en términos de matrices (o arreglos). Nadie usa Python sin usar NumPy en el área científica.
- Para usar NumPy en forma eficiente y que valga la pena, es necesario que al algoritmo a programar sea posible describirlo en términos de álgebra vectorial.
- Puede paralelizarse para soportar un hardware específico (GPUs o TPUs) en varias encarnaciones, aunque aún no hay una versión oficial.

- Queda claro que como existen una cantidad importante de órdenes, uno sólo necesita aprender los comandos básicos y para alguna tarea muy específica hay leer la documentación primero para verificar si el comando ya existe.
- En soporte a la idea anterior, existen demasiados comandos, puede ser una mala idea pensar que se pueden aprender todos.
- Se siguen hoy en día agregando y modificando comandos. A NumPy aún se lo continúa construyéndo.

CAPITULO 10

Gráficos - Biblioteca Matplotlib

10.1. Dibujos usando Matplotlib

Esta biblioteca es una de las varias que existen para realizar gráficos en Python. No es la única, pero es sin duda la más usada.

Ventajas: - Se pueden hacer dibujos con pocos comandos, pero llegado el caso muy sofisticados. - Tiene una forma dual, por comandos directos, o bien accediendo al objeto que describe la figura. Esta última opción es mucha más laboriosa pero muy útil en casos de gráficos muy sofisticadas o con detalles complejos.

Muchas información en https://matplotlib.org/

Y una cantidad impresionante de ejemplos (con el código para usarlo en un dibujo similar) en: https://matplotlib.org/stable/plot_types/index.html.

Y diagramas para referencias rápidas en: https://matplotlib.org/cheatsheets/.

Para usarlo hay que cargar **NumPy** de manera **Obligatoria**.

Matplotlib se construyó para su uso sobre NumPy.

Algunos de los ejemplos presentados en este capítulo fueron tomados de https://matplotlib.org/stable/gallery/.

10.1.1. Pyplot

Pyplot es la interfase a la biblioteca de matplotlib. Pyplot está diseñada siguiendo el estilo de un lenguaje llamado Matlab que fue muy exitoso en ingeniería y ciencia. Por lo cual a Matplotlib se accede a través de pyplot con el siguiente comando:

import matplotlib.pyplot as plt

Es decir, cargamos de matplotlib la parte de pyplot para usar esta biblioteca y sus funciones, veamos un ejemplo:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt # Esta es la biblioteca para graficar
```

Hay mucha información de como usar esta biblioteca en la página oficial: http://matplotlib.org/.

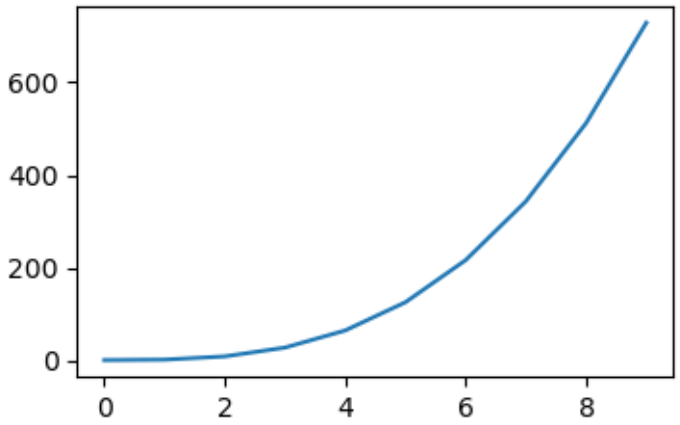
10.2. Uso del comando Plot()

Veamos que Matplotlib es muy fácil de usar y en este caso utilizaremos la función "plot()".

[3]:
```
x = np.arange(10) # defino un arreglo para tener un referencia
```

y dibujo $f(x) = x^3$ de una manera compacta.

[4]:
```
plt.plot(x, x**3); #  El ";" al final es para que no me salga una
                   #  descripción del objeto estilo
                   # [<matplotlib.lines.Line2D at 0x7fbd28b9b590>]
```



Cómo pueden ver en el ejemplo, este gráfico nos llevó sólo una línea de código.

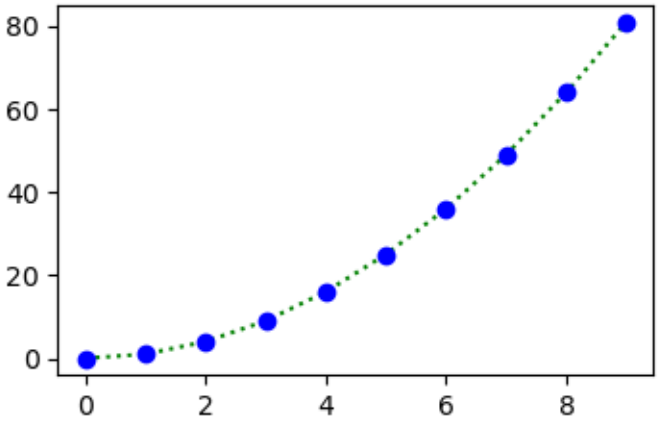
El método plot de matplotlib, necesitó la abscisa que es el arreglo NumPy **x** y la ordenada que es el resultado del cálculo de $f(x) = x^3$, que se realizó al vuelo mientras se corrió el código. Note que no se hizo indicación alguna del tipo de línea, color o punto en el dibujo. Es decir se cargaron valores "default" pre-programados.

10.2.1. Controlando los colores y los símbolos en el dibujo

Hay modos compactos de órdenes para modificar el gráfico, veamos algunas.

Y también podemos superponer gráficos

[5]:
```
plt.plot(x,x**2,':g')    #<--- determina la línea
plt.plot(x,x**2,'ob');   #<--- determina los puntos
```

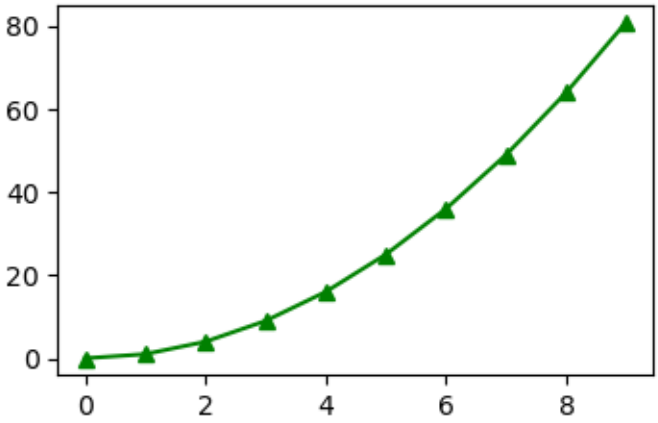


En este ejemplo, se generaron dos gráficos, uno por línea de código. En la primera línea, se agrego **':g'** que significa en un modo muy compacto ":" línea de puntos y "g" color verde (la g es de green). En la segundo línea ahora el "o" indica un punto lleno y "b" color azul (la b es de blue).

Note que la segunda orden no une puntos, y es entonces un gráfico de puntos discretos (no conectados o "scatter" en inglés).

También podría haber utilizado una orden no tan compacta y ser más descriptivo de lo que estoy programando.

[6]:
```
plt.plot(x, x**2, c='green', marker='^');
```



En este ejemplo pueden ver que el color y el símbolo se pueden poner también modos no tan comprimidos y más claros para el lector del código. "c" o "color" indicará el color y "marker" el símbolo para marcar los puntos en el dibujo.

10.2.2. Sobre escribiendo dibujos (Overplot)

[10]:
```
# Genero dos arreglos NumPy
x = np.linspace(0, 20, 100)  # 100 valores separados de 0 to 20
y = np.sin(x)

# y los dibujo, pero en el segundo realizo otro cálculo
# en los valores de la ordenada.
plt.plot(x, y)
plt.plot(x, y**2);
```

10.2.3. Modificando ejes y límites

Veamos este dibujo:

[11]:
```
plt.plot(x, y);
```

Pero resulta que quiero que sea entre 0, y 4π, o que la ordenada este en el rango (-1.5,1).

Tengo que fijar entonces estos límites en ambos ejes. para ello uso los siguientes comandos: xlim, e ylim.

Si no los uso, el gráfico se realizará con valores "default" determinados por el propio Python.

[12]:
```
plt.plot(x, y)
plt.xlim((0., 4*np.pi))
plt.ylim((-1.5,1.5))
```

[12]:
```
(-1.5, 1.5)
```

10.2.4. Multiples gráficos en una sola celda

¿Puedo poner varios gráficos por celda?

Probemos:

[13]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))

plt.plot(x, y)
plt.xlim((0., np.pi*3))
```

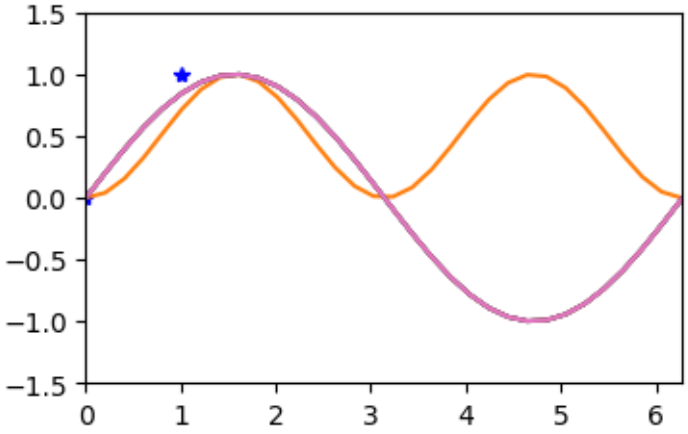
[13]:
```
(0.0, 9.42477796076938)
```

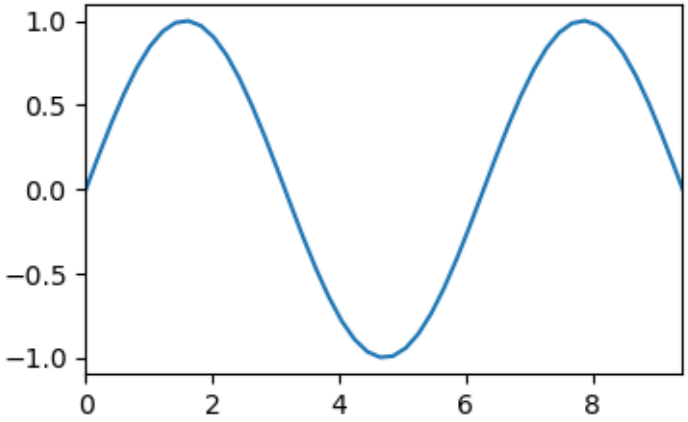
No funcionó, sólo veo el último dibujo. El primero se me perdió.

Si tengo más de una figura en una celda, tengo que "cerrar" el primero para que se grafique en la pantalla. Si no lo hago sólo veré el último de estos. Para cerrar el dibujo tengo que usar el comando **"plt.show( )"**.

[14]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))
plt.show()
```

```
plt.plot(x, y)
plt.xlim((0., np.pi*3))
plt.show()
```

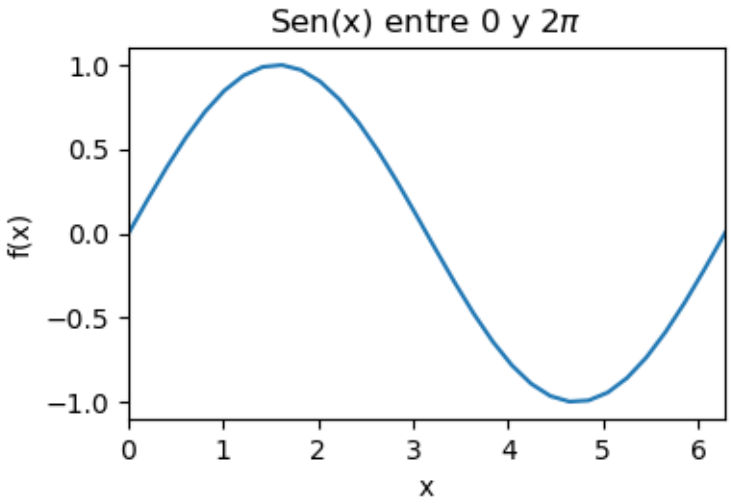




10.2.5. Nombre de los ejes y título de la figura

Para ello se utiliza el comando plt.title('texto') donde como string de texto se escribe el título. Los nombres de cada eje se indican con los comandos plt.xlabel('texto') y plt.ylabel('texto') para cada eje respectivamente.

[15]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))
plt.title('Sen(x) entre 0 y $2\pi$')
plt.xlabel('x')
plt.ylabel('f(x)');
```



10.3. Documentación en línea

[16]:
```
help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

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

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
    formatting like color, marker and linestyle. It's a shortcut string
    notation described in the *Notes* section below.

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses

    You can use `.Line2D` properties as keyword arguments for more
    control on the appearance. Line properties and *fmt* can be mixed.
    The following two calls yield identical results:

    >>> plot(x, y, 'go--', linewidth=2, markersize=12)
    >>> plot(x, y, color='green', marker='o', linestyle='dashed',
    ...      linewidth=2, markersize=12)
```

```
When conflicting with *fmt*, keyword arguments take precedence.


**Plotting labelled data**

There's a convenient way for plotting objects with labelled data (i.e.
data that can be accessed by index ``obj['y']``). Instead of giving
the data in *x* and *y*, you can provide the object in the *data*
parameter and just give the labels for *x* and *y*::

>>> plot('xlabel', 'ylabel', data=obj)

All indexable objects are supported. This could e.g. be a `dict`, a
`pandas.DataFrame` or a structured numpy array.


**Plotting multiple sets of data**

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call `plot` multiple times.
  Example:

  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')

- If *x* and/or *y* are 2D arrays a separate data set will be drawn
  for every column. If both *x* and *y* are 2D, they must have the
  same shape. If only one of them is 2D with shape (N, m) the other
  must have length N and will be used for every data set m.

  Example:

  >>> x = [1, 2, 3]
  >>> y = np.array([[1, 2], [3, 4], [5, 6]])
  >>> plot(x, y)

  is equivalent to:

  >>> for col in range(y.shape[1]):
  ...     plot(x, y[:, col])

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  >>> plot(x1, y1, 'g^', x2, y2, 'g-')

  In this case, any additional keyword argument applies to all
  datasets. Also, this syntax cannot be combined with the *data*
  parameter.
```

```
By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using
:rc:`axes.prop_cycle`.


Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to ``range(len(y))``.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.

    .. note::
        Technically there's a slight ambiguity in calls where the
        second label is a valid *fmt*. ``plot('n', 'o', data=obj)``
        could be ``plt(x, y)`` or ``plt(y, fmt)``. In such cases,
        the former interpretation is chosen, but a warning is issued.
        You may suppress the warning by adding an empty format string
        ``plot('n', 'o', '', data=obj)``.

Returns
-------
list of `.Line2D`
    A list of lines representing the plotted data.

Other Parameters
----------------
scalex, scaley : bool, default: True
```

```
        These parameters determine if the view limits are adapted to the
        data limits. The values are passed on to
        `~.axes.Axes.autoscale_view`.

    **kwargs : `~matplotlib.lines.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you specify multiple lines with one plot call, the kwargs apply
        to all those lines. In case the label object is iterable, each
        element is used as labels for each set of data.

        Here is a list of available `.Line2D` properties:

        Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array␣
 ↪and a
dpi value, and returns a (m, n, 3) array and two offsets from the bottom left
corner of the image
        alpha: scalar or None
        animated: bool
        antialiased or aa: bool
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        color or c: color
        dash_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
        dash_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
        dashes: sequence of floats (on/off ink in points) or (None, None)
        data: (2, N) array or two 1D arrays
        drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid',␣
 ↪'steps-
post'}, default: 'default'
        figure: `.Figure`
        fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
        gapcolor: color or None
        gid: str
        in_layout: bool
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
        linewidth or lw: float
        marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
        markeredgecolor or mec: color
        markeredgewidth or mew: float
        markerfacecolor or mfc: color
        markerfacecoloralt or mfcalt: color
        markersize or ms: float
```

```
        markevery: None or int or (int, int) or slice or list[int] or float or
(float, float) or list[bool]
        mouseover: bool
        path_effects: `.AbstractPathEffect`
        picker: float or callable[[Artist, Event], tuple[bool, dict]]
        pickradius: unknown
        rasterized: bool
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        solid_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
        solid_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
        transform: unknown
        url: str
        visible: bool
        xdata: 1D array
        ydata: 1D array
        zorder: float

    See Also
    --------
    scatter : XY scatter plot with markers of varying size and/or color (
        sometimes also called bubble chart).

    Notes
    -----
    **Format Strings**

    A format string consists of a part for color, marker and line::

        fmt = '[marker][line][color]'

    Each of them is optional. If not provided, the value from the style
    cycle is used. Exception: If ``line`` is given, but no ``marker``,
    the data will be a line without markers.

    Other combinations such as ``[color][marker][line]`` are also
    supported, but note that their parsing may be ambiguous.

    **Markers**

    =============   ===============================
    character       description
    =============   ===============================
    ``'.'``         point marker
    ``','``         pixel marker
    ``'o'``         circle marker
    ``'v'``         triangle_down marker
    ``'^'``         triangle_up marker
    ``'<'``         triangle_left marker
    ``'>'``         triangle_right marker
    ``'1'``         tri_down marker
```

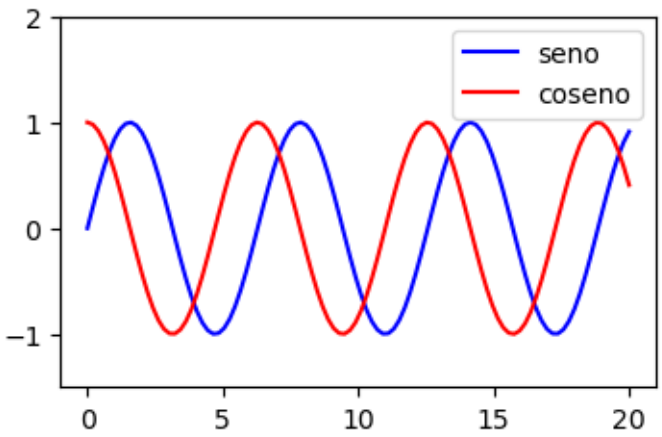
```
``'2'``          tri_up marker
``'3'``          tri_left marker
``'4'``          tri_right marker
``'8'``          octagon marker
``'s'``          square marker
``'p'``          pentagon marker
``'P'``          plus (filled) marker
``'*'``          star marker
``'h'``          hexagon1 marker
``'H'``          hexagon2 marker
``'+'``          plus marker
``'x'``          x marker
``'X'``          x (filled) marker
``'D'``          diamond marker
``'d'``          thin_diamond marker
``'|'``          vline marker
``'_'``          hline marker
=============   ==============================

**Line Styles**

=============    ==============================
character        description
=============    ==============================
``'-'``          solid line style
``'--'``         dashed line style
``'-.'``         dash-dot line style
``':'``          dotted line style
=============    ==============================

Example format strings::

    'b'    # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line

**Colors**

The supported color abbreviations are the single letter codes

=============    ==============================
character        color
=============    ==============================
``'b'``          blue
``'g'``          green
``'r'``          red
``'c'``          cyan
``'m'``          magenta
``'y'``          yellow
```

```
    ``'k'``          black
    ``'w'``          white
    =============    ==============================

    and the ``'CN'`` colors that index into the default property cycle.

    If the color is the only part of the format string, you can
    additionally use any  `matplotlib.colors` spec, e.g. full names
    (``'green'``) or hex strings (``'#008000'``).
```

10.3.1. Leyendas

A cada curva o puntos específicos dentro de una figura se los puede diferenciar con una leyenda que se escribe con el comando que los dibuja. Y luego al final, indicando el lugar de la figura donde se pondrá esta leyenda.

[17]:
```
x = np.linspace(0, 20, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, '-b', label='seno')
plt.plot(x, y2, '-r', label='coseno')
plt.legend(loc='upper right')
plt.ylim((-1.5, 2.0));
```



En este caso indicamos con rojo el $\sin(x)$ en azul y el $\cos(x)$ en rojo. Luego la leyenda la dibujamos arriba (“upper”) y a la derecha (“right”) con el comando plt.legend( ).

10.3.2. Nomenclatura de las distintas partes de un dibujo

Afortunamente la gente de Matplotlib a generado este dibujo que da nombre a las partes de un gráfico del tipo de los que se usan ciencia. Con estos nombres es muy fácil buscar (o adivinar) el comando que se necesita para su modificación para que el dibujo quede a gusto del usuario o bien de los requerimientos de las revistas o congresos.

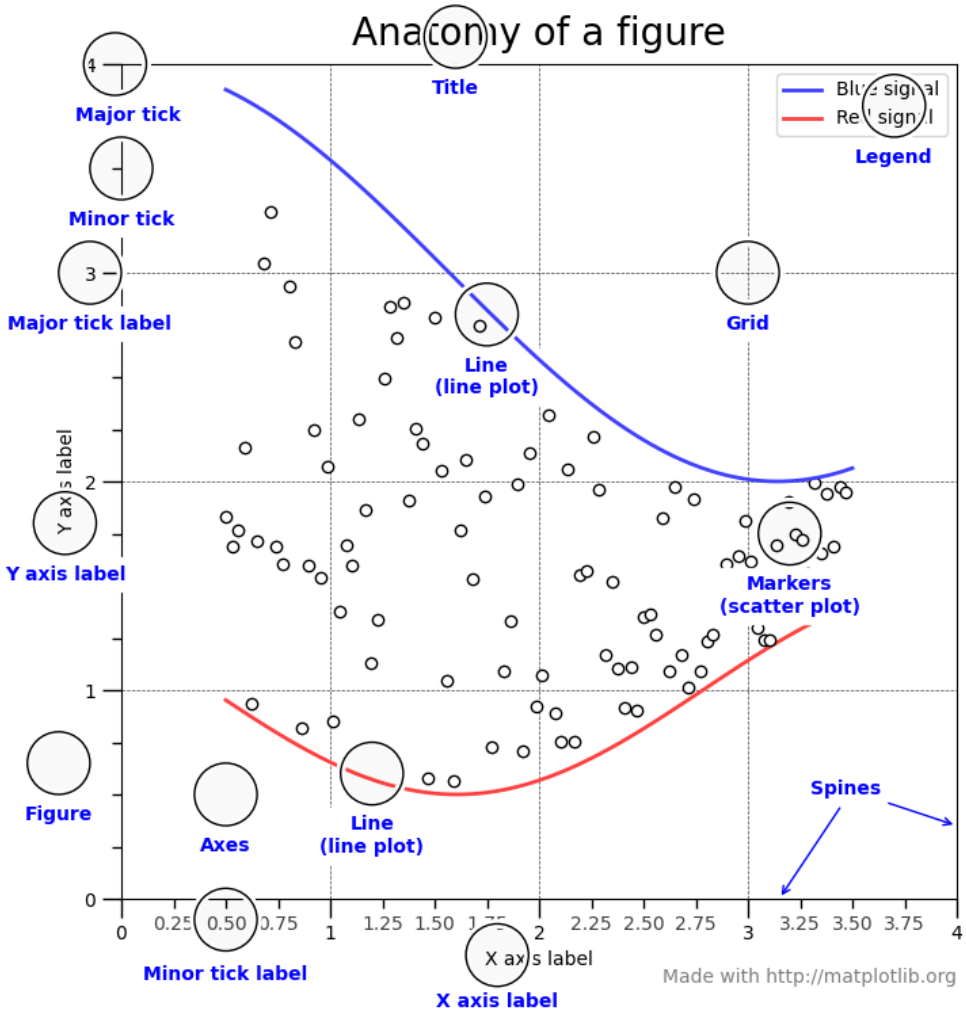


Figura 10.1: title

CAPITULO 11

Gráficos - Biblioteca Matplotlib II

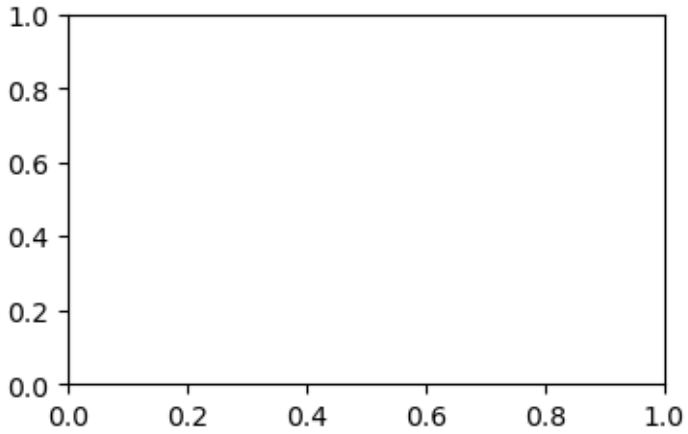
11.1. La figura como Objeto Python

Todo es un objeto en Python

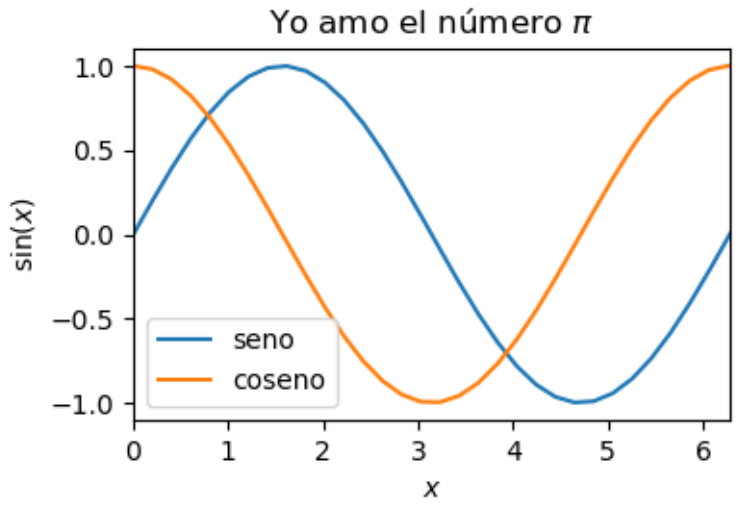
Puedo crear el objeto y luego modificar sus parámetros y agregar lo que quiero graficar. Suele ser más laborioso (no mucho) y las órdenes no son las mismas que las que vimos hasta ahora, pero muy parecidas. La forma de trabajo es que por un lado tengo la figura en sí, y por otro los ejes (ya que puede haber varios) y modifico cada uno con lo que necesito

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (4,2.5)  # <- Este comando es para que la
                                          # versión impresa de este notebook
                                          # tenga las dimensiones
                                          # correctas, el lector debe
                                          # ignorarlo.
```

[2]:
```
x = np.linspace(0, 20, 100)
y1 = np.sin(x)
y2 = np.cos(x)

fig = plt.figure()              # Genera un dibujo vacío
ax = fig.add_subplot(1, 1, 1)   # Especifico (número de filas, columnas,
                                # número de figura)
```



[3]:
```
fig, ax = plt.subplots()  # Creo la figura como dos objetos: figura
                          # en si y ejes
ax.plot(x, y1)
ax.plot(x, y2)
ax.set_xlim(0., 2*np.pi)
ax.legend(['seno', 'coseno'], loc='best')
ax.set_xlabel("$x$")
ax.set_ylabel("$\sin(x)$")
ax.set_title("Yo amo el número $\pi$");
```



Si se desea conocer más de este comando se puede poner "help(ax)" y generará una respuesta bastante extensa de los diversos comandos y métodos del objeto.

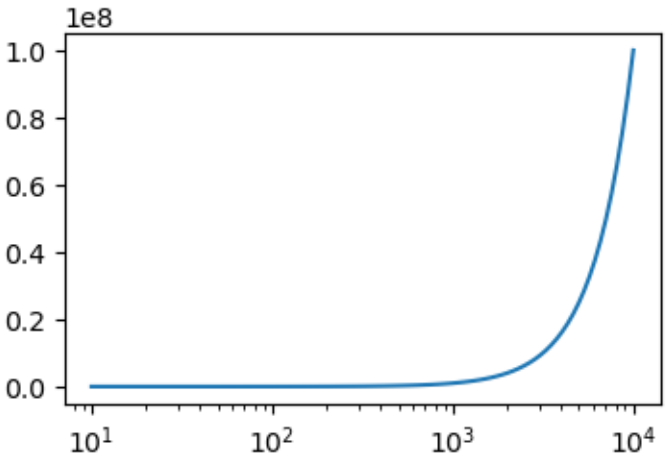
11.1.1. Plots Logarítmicos

Los gráficos logarítmicos se pueden hacer con un eje logaritmico y otro lineal o con ambos ejes logarítmicos

- Con un eje logarímtico se lo llama semilog"n"() donde n es x o y según sean las abscisas o las ordenadas
- Con los dos ejes logarítmico es loglog()

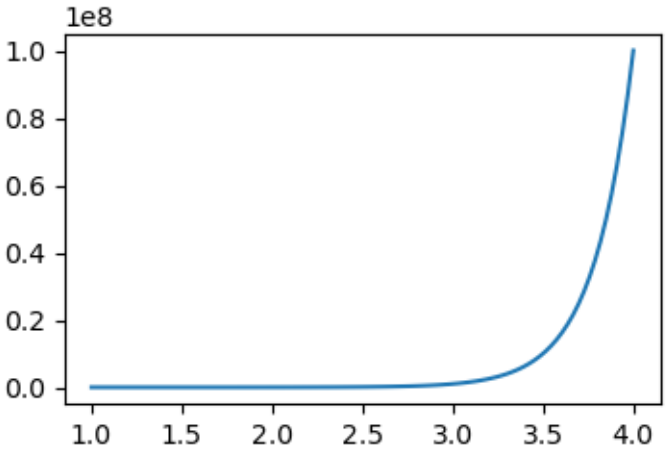
Si el eje X es el logarítmico puede hacer el gráfico así, donde queda muy claro su escala

[4]:
```
xl = np.logspace(1, 4, 100)
fig, ax = plt.subplots()
ax.semilogx(xl, xl**2);
```



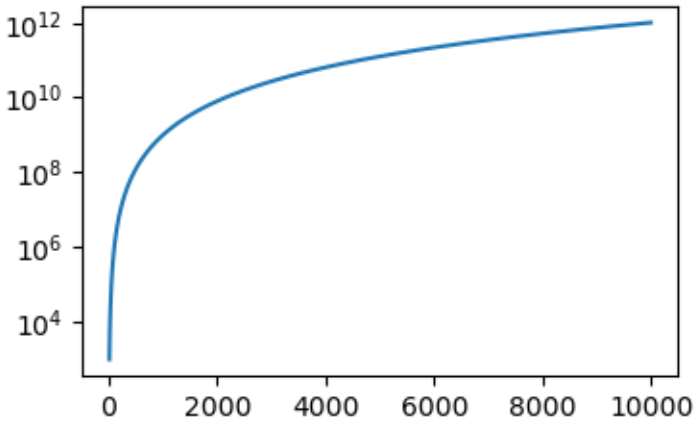
O dejarlo que ponga del valor del logaritmo en el eje directamente, auqnue en esta forma queda menos claro que clase de magnitud es.

[5]:
```
xl = np.logspace(1, 4, 100)
fig, ax = plt.subplots()
ax.plot(np.log10(xl), xl**2);
```

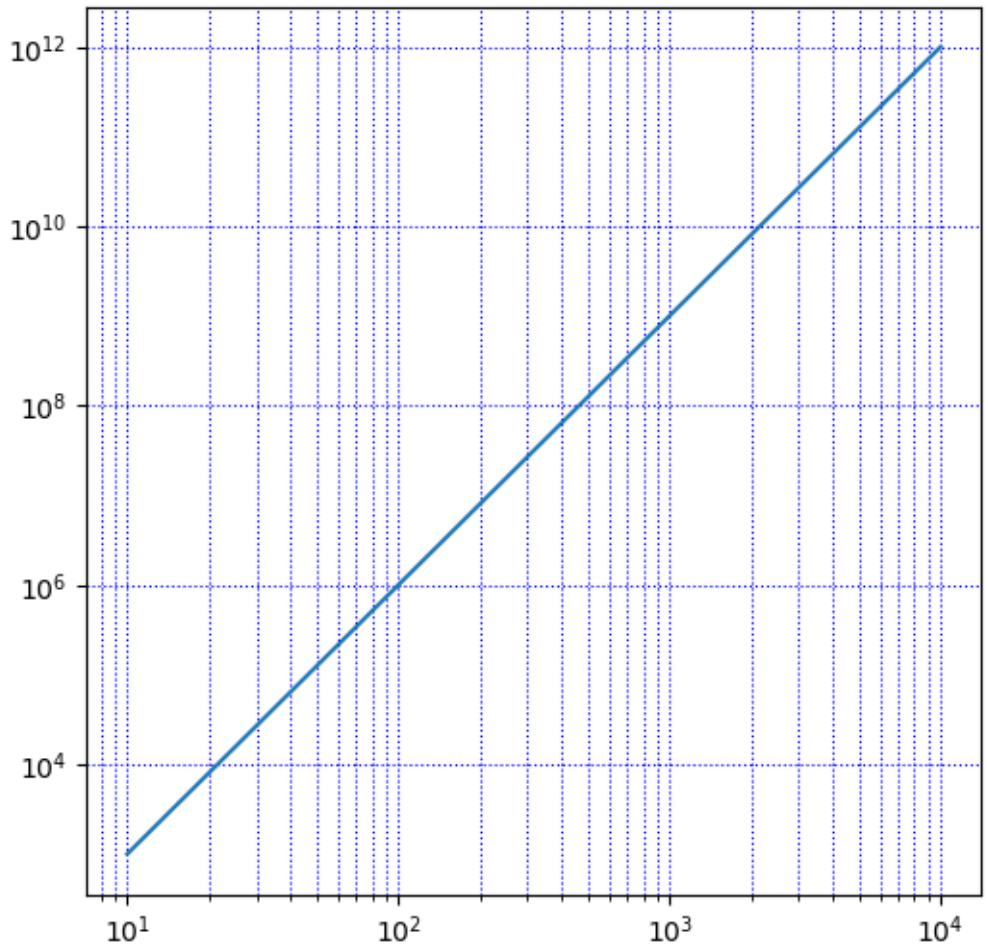


Puedo hacer lo mismo para el eje de las ordenadas (note que cambio entonces "plt.semilog**X**" por "plt.semilog**Y**")

[6]:
```
fig, ax = plt.subplots()
ax.semilogy(xl, xl**3);
```



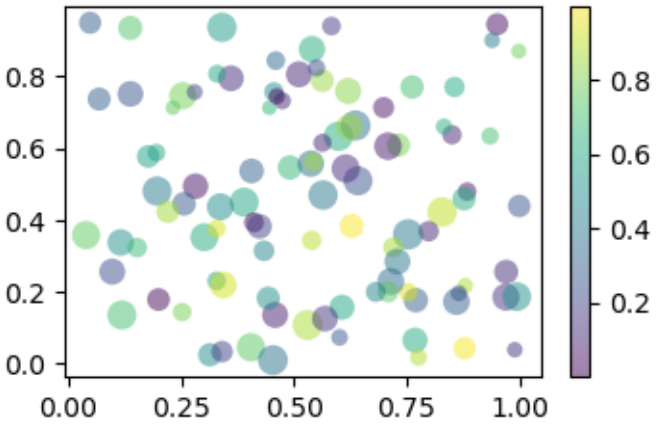
[7]:
```
fig, ax = plt.subplots(figsize=(6,6))
ax.loglog(xl, xl**3);
ax.grid(True,which="both",ls=":", c='blue')
```



11.1.2. Scatter o gráficos con puntos

Hay mucha libertad para elegir el símbolo para el punto, su tamaño y su color veamos un dibujo donde elijo las coordenadas, tamaño y color al azar.

[8]:
```
xr = np.random.rand(100)
yr = np.random.rand(100)
cr = np.random.rand(100)
sr = np.random.rand(100)

fig, ax = plt.subplots()
sc = ax.scatter(xr, yr, c=cr, s=30+sr*100, edgecolor='none', alpha=0.5) #␣
 ↪colores y tamaños dependen
                                                                          #␣
 ↪del valor de las variables
                                                                          #␣
 ↪que en este caso son números
                                                                          # al␣
 ↪azar
fig.colorbar(sc);
```



11.2. Calculando π

Utilizando el método de las piedras. Probaremos arrojando 5000 piedra.

Pero primero recodemos el algoritmo que ya describimos en el apunte de Fortran:

Recordemos que: se denomina frecuencia a la cantidad de veces de que un cierto estado se repita en un experimento frente a todos los experimentos que se han realizado. La frecuencia es algo que se puede medir. Por ejemplo, tirando una moneda y contando la cantidad de veces que sale cara frente a las veces que esa moneda fue arrojada.

La probabilidad de que ese estado en particular suceda es un resultado teórico que se obtendría de repetir infinitamente el experimento. La frecuencia entonces se convierte en probabilidad

cuando el número de experimentos tiende a infinito. En este caso vamos a asumir que cuando repito mucho veces un experimento y mido la frecuencia se parecerá bastante a la probabilidad. Es decir, puedo tirar una moneda muchas veces al aire, y medir la frecuencia de que salga cara y no seca. Pero sólo cuando se la tire infinitas veces obtendré la probabilidad del suceso.

Pero como la probabilidad es una definición: prob= casos favorables/casos totales, para una moneda perfecta tengo: una cara/dos posibilidades (cara mas seca). O sea que la probabilidad de obtener cara es $Prob = 0.50$.

Con esta idea de convertir la medida de la fracuencia en probabilidad vamos a calcular el valor de π utilizando las siguientes reglas:

- Tengo un cuadrado perfecto dibujado en el piso y le arrojo piedras. Si una piedra cae afuera la vuelvo a tirar.
- Tengo un contador que lleva la cuenta de todas las piedras que se arrojan.
- Dentro de ese cuadro dibujo un círculo centrado en el cuadrado y con el radio tal de que es tangente a todos los lados de cuadrado.
- Llevo la cuenta de todas piedras que caen en el círculo.
- Realizo este experimento utilizando la computadora. La posición de la piedra la determino generando dos números al azar con distribución uniforme. Es decir la coordenadas (x,y) del impacto de la piedra son dos números al azar.
- Repito el experimento muchas veces, con la idea de que cuando más veces mejor.
- Calculo la frecuencia de que la piedra caiga dentro del círculo. Es decir:

$frec = \frac{piedras\ en\ el\ círculo}{total\ de\ piedras}$

Viendo este experimento ¿Cuál es la probabilidad de que una piedra quede dentro del círculo?
\

$$Prob = \frac{Casos\ favorables}{Total\ de\ casos} = \frac{Área\ del\ círculo}{Área\ del\ cuadrado}$$

Reemplazando las áreas por su expresiones y haciendo las cuentas:

$$Prob = \frac{\pi R^2}{(2R)^2} = \frac{\pi}{4}$$

Si suponemos (aunque sabemos que se le parecen pero no son lo mismo) que la frecuencia medida es la probabilidad de que la piedra caiga en el círculo, obtenemos que:

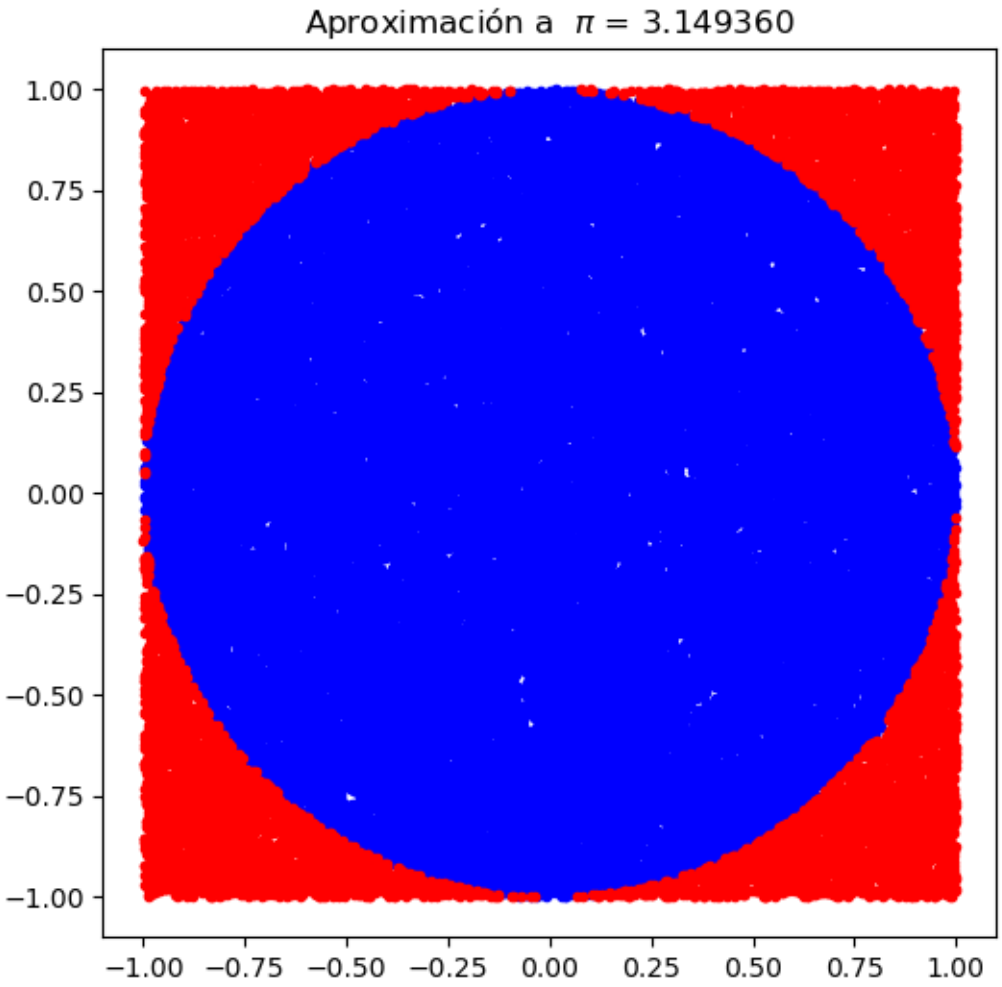
$frec \sim \frac{\pi}{4}$ y si despejo π obtengo $\pi \sim 4frec$ donde la frecuencia la obtengo a partir de la simulación. Pero para que esto funcione tenemos que hacer que la frecuencia realmente se parezca a la probabilidad y para que esto suceda tenemos que repetir el experimento muchas veces, en este caso lo haremos 5000 veces.

Veamos cómo sería un programa que realice toda esta tarea. Genere los dos números al azar, vea si estás coordenadas están dentro del círculo, actualice los contadores (piedras en el círculo y cantidad total de piedras), nos de un estimado del valor de π que obtuvimos hasta el momento y vuelva a repetir la operación una y otra vez.

[9]:
```
npts = 50000

xs = 2*np.random.rand(npts)-1
ys = 2*np.random.rand(npts)-1
```

```
r = xs**2+ys**2

ninside = (r<1).sum()

plt.figure(figsize=(6,6)) # Para que la figura sea cuadrada

plt.title("Aproximación a  $\pi$ = %f" % (4*ninside/float(npts)))
plt.plot(xs[r<1],ys[r<1],'b.')
plt.plot(xs[r>1],ys[r>1],'r.')


print("Pi da:", ninside/npts*4)
```

```
Pi da: 3.14936
```



Note que ahora usando la librería NumPy se "tiraron" las priedras no de a una, si no todas en un sólo comando. Por eso no hay ninguna orden tipo "bucle", a diferencia de como se realizó este mismo cálculo en el apunte de Fortran.

11.3. Parámetros del gráfico

Para dejar fijos algunos valores "domésticos" del dibujo tenemos dos formas de establecerlos:

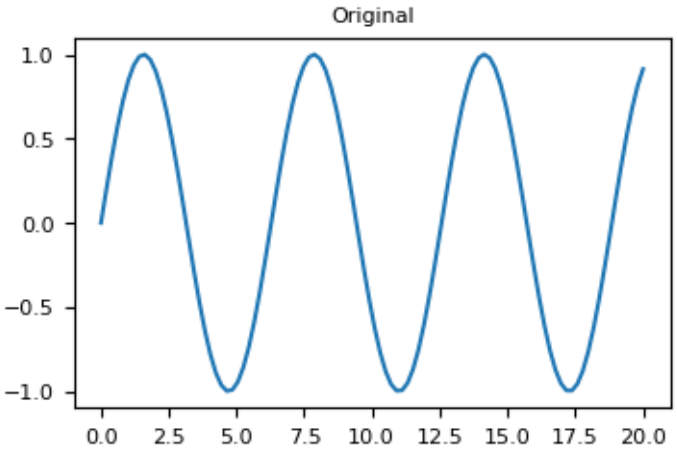
Con la orden **plc.rc('Lo que queremos cambiar', nuevo valor)**, donde nuevo valor puede ser una variable en la que está el valor del nuevo parámetro. O la orden:

import matplotlib # cargo TODA la matplotlib

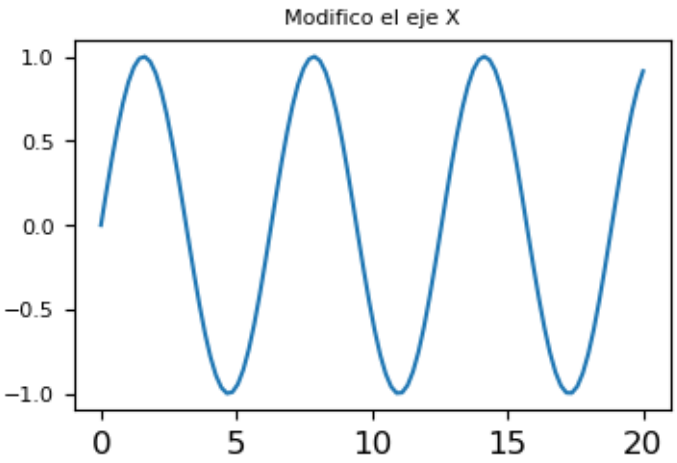
matplotlib.rc('Lo que queremos cambiar', nuevo valor) # modifico un parámetro

Veamos algunos ejemplos:

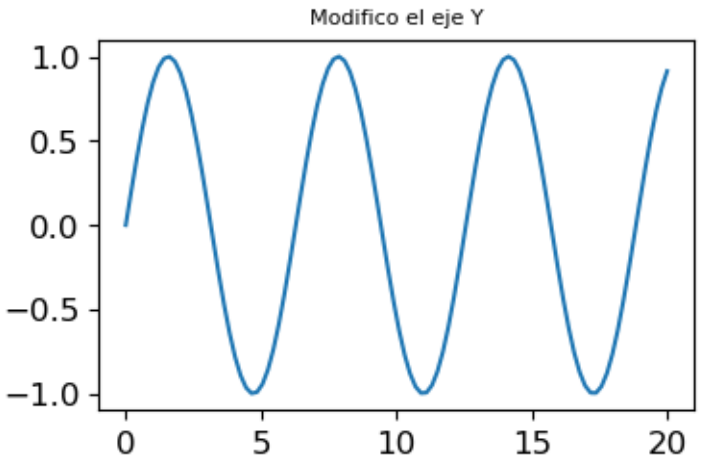
[10]:
```
SMALL_SIZE = 8
MEDIUM_SIZE = 10
BIGGER_SIZE = 12

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

# y  lo mismo se puede hacar para los demás parámetros del
# gráfico

## Ejemplo:
plt.plot(x,np.sin(x))
plt.title('Original')
plt.show()

print("")
print("Hago un cambio en los números de los ticks del eje X")
plt.rc('xtick', labelsize=BIGGER_SIZE)
plt.title('Modifico el eje X')
plt.plot(x,np.sin(x))
plt.show()

print("")
print("y si ahora cambio el eje Y")
plt.rc('ytick', labelsize=BIGGER_SIZE)
plt.title('Modifico el eje Y')
plt.plot(x,np.sin(x))
plt.show()

# O también lo podría hacer de la forma

import matplotlib
SMALL_SIZE = 8
matplotlib.rc('font', size=SMALL_SIZE)
matplotlib.rc('axes', titlesize=SMALL_SIZE)
```



Hago un cambio en los números de los ticks del eje X



y si ahora cambio el eje Y

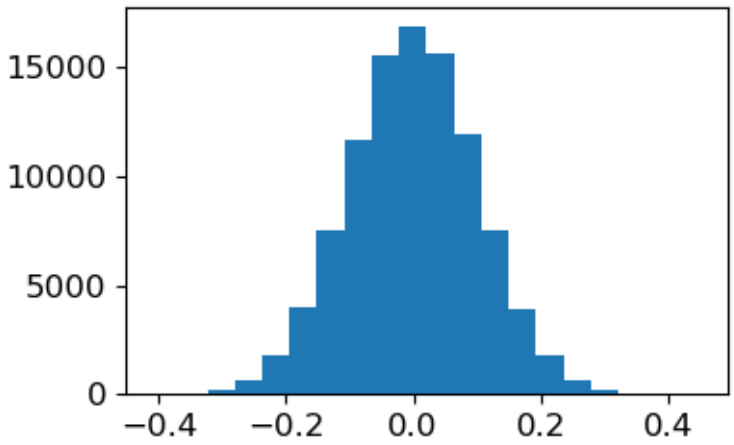


11.3.1. Cálculos y dibujos en Matplolib

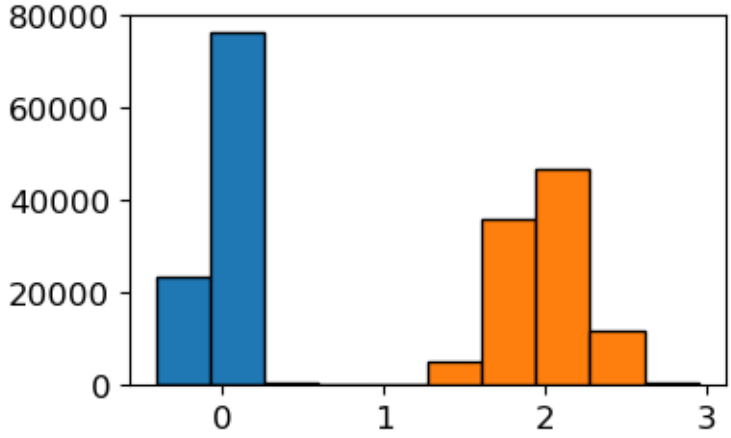
Matplotlib tiene también capacidad de hacer cálculos para dibujos específicos. Por ejemplo en el caso de calcular un histograma. Los histogramas son graficos de barras, donde cada barra es la cuenta de cuentos eventos se producen en un cierto intervalo.

Veamos como es esto:

[11]:
```
N_points = 100000
n_bins = 20

# Generemos dos distribuciones de números la azar
# gaussianas (Son los que tiene forma de campana)

dist1 = np.random.normal(0,0.1,N_points)
dist2 = 0.4 * np.random.normal(0,0.6,N_points) + 2

# Con esos datos hago el dibujo de un histograma

fig, axs = plt.subplots(1, 1, sharey=True, tight_layout=True)

axs.hist(dist1, bins=n_bins);
```



[12]:
```
bins = np.histogram(np.hstack((dist1, dist2)), bins=10)[1]
plt.hist(dist1, bins, edgecolor='black')
plt.hist(dist2, bins, edgecolor='black');
```

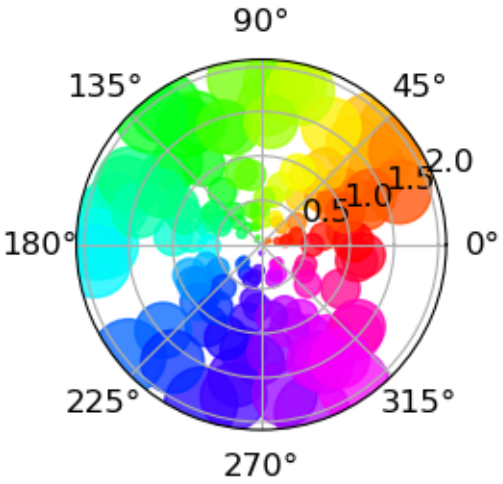


11.4. Gráficos con distribución de puntos en coordenadas Polares

Este es otro de los gráficos que podemos hacer Para este dibujo enero dos coordenadas r y tita con 150 números al azar cada una

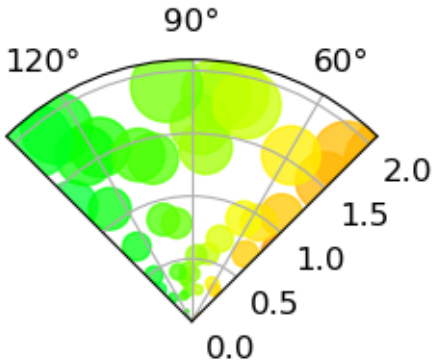
[13]:
```
N=150
r = 2* np.random.rand(N)
tita = 2 * np.pi * np.random.rand(N)

# Genero un área para cada punto en función de la coordenada r
area = 200* r**2

# Genero colores que los hago depender del la coordenada anfgular
color = tita
```

```
# Creo la figura
fig= plt.figure()

# Y en la figura creo un plot polar
ax = fig.add_subplot(projection='polar') #<-- Notar que uso la proyección␣
 ↪polar
                                         # para activar esta propiedad

# la dibujo

ax.scatter(tita, r, c=color, s=area, cmap='hsv', alpha=0.75);
```



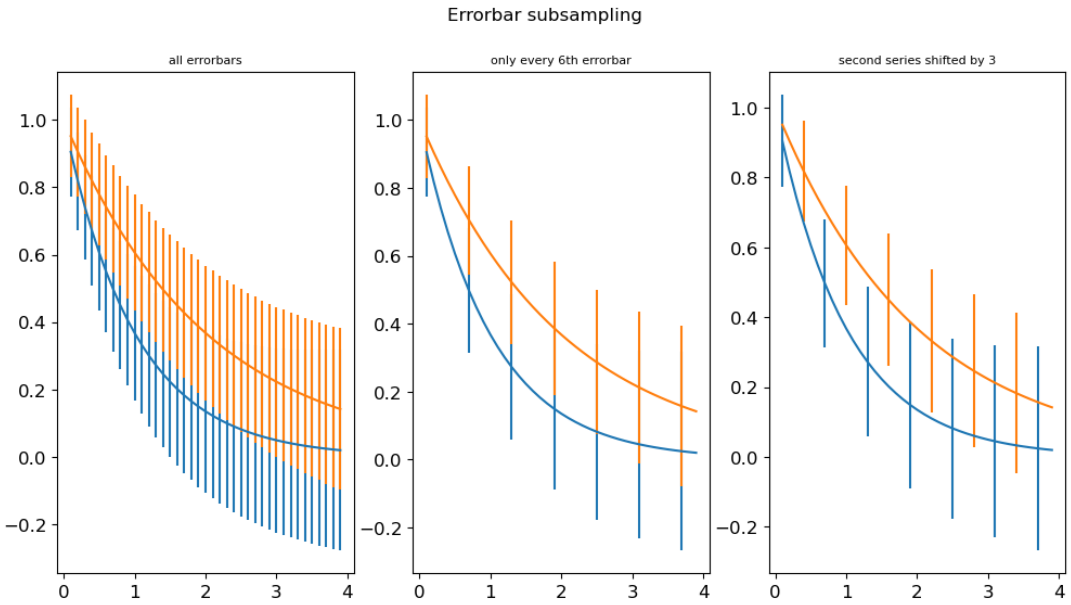
Incluso podría tomar una región determinado por el ángulo

[14]:
```
fig= plt.figure()
ax = fig.add_subplot(projection='polar')
ax.set_thetamin(45)
ax.set_thetamax(135)
ax.scatter(tita, r, c=color, s=area, cmap='hsv', alpha=0.75);
```



Gráficos múltiples usando un sólo Objeto

[15]:
```
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.1)
y1 = np.exp(-1.0 * x)
y2 = np.exp(-0.5 * x)

# example variable error bar values
y1err = 0.1 + 0.1 * np.sqrt(x)
y2err = 0.1 + 0.1 * np.sqrt(x/2)


fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3, sharex=True,
                                    figsize=(12, 6))

ax0.set_title('all errorbars')
ax0.errorbar(x, y1, yerr=y1err)
ax0.errorbar(x, y2, yerr=y2err)

ax1.set_title('only every 6th errorbar')
ax1.errorbar(x, y1, yerr=y1err, errorevery=6)
ax1.errorbar(x, y2, yerr=y2err, errorevery=6)

ax2.set_title('second series shifted by 3')
ax2.errorbar(x, y1, yerr=y1err, errorevery=(0, 6))
ax2.errorbar(x, y2, yerr=y2err, errorevery=(3, 6))

fig.suptitle('Errorbar subsampling')
plt.show()
```


Errorbar subsampling
all errorbars
only every 6th errorbar
second series shifted by 3
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
1
2
3
4

[ ]:

CAPITULO 12

Tablas en Python - Biblioteca Pandas

Pandas es una librería de Python pensada para manejar datos con la idea de que estos están en un formato de tabla. Es muy usada en análisis de grandes cantidades de datos o de series de valores que tengan una evolución temporal. Es un reemplazo muy eficiente de las planillas de cálculo. En el caso particular de la Astronomía, la librería AstroPy tiene integrada internamente un manejo de tablas similar a Pandas.

12.1. Cargando Pandas

Para usar Pandas primero importamos la libreria como se hace siempre con la orden **import** y en este caso sería:

[1]:
```
import pandas as pd
```

Es muy común que se lo importe con el nombre pd.

12.2. Tablas - DataFrame

Un **DataFrame** is un método de Pandas para cargar **toda una tabla** en un objeto Pandas, incluyendo el nombre de las columnas y el nombre de la filas. Este objeto dataframe es un arreglo diseñado para incluir en memoria una tabla entera.

Se lo utiliza de esta manera:

[2]:
```
pd.DataFrame({'Blanco': [22, 61], 'Negro': [148, 2], 'Rojo': [31,10]})
```

[2]:

	Blanco	Negro	Rojo
0	22	148	31
1	61	2	10

En este comando ya se pueden apreciar varias cosas que caracterizan al uso de Pandas:

- Pandas está diseñado para manejar tablas.
- Estas son de cualquier tamaño, incluso de un tamaño apreciablemente grande.
- Los datos en Panda tienen una estructura que describiremos a continuación, pero que conserva todoas las propiedades de la tabla original.

Ene le ejmeplo es importante notar que "Blanco", "Negro" y "Rojo" son nombres de la columnas, mientras que al no indicar nombre de las filas estas tienen un número de índice que comienza a contarse en "0"

Las tablas no se limitan sólo a números pueden contener cualquier variable básica de Python. Por ejemplo, textos:

[3]:
```
pd.DataFrame({'Roberto': ['Me gustó', 'Espantoso'], 'Ana': ['Muy Bueno',␣
↪'Regular']})
```

[3]:
```
      Roberto        Ana
0   Me gustó  Muy Bueno
1  Espantoso    Regular
```

Aparte de la estructura tipo diccionario que me sirve para cargar las columnas y darles nombre, también como se indicó anteriormente puedo darle nombre a las filas. Estos nombres de las filas se consideran como los índices de la tabla y por ello se los declara con el comando **index**. Este se usa como en el ejemplo siguiente:

[4]:
```
pd.DataFrame({'Roberto': ['Me gustó', 'Espantoso'],
              'Ana': ['Muy Bueno', 'Regular']},
              index=['Producto A', 'Producto B'])
```

[4]:
```
              Roberto        Ana
Producto A   Me gustó  Muy Bueno
Producto B  Espantoso    Regular
```

12.3. Series - pd.Series

Las **series** en comparación con los DataFrames son sólo una secuencia de valores, es decir, muy parecidos una lista de Python. En cierta manera vuelven a representar las variables "listas" del Python original, pero como veremos más adelante, son bastante más sofisticadas que las originales.

[5]:
```
pd.Series([1, 2, 3, 4, 5])
```

[5]:
```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Una serie en Panda se puede considerar como una columna individual de un DataFrame, entonces también tiene asignado un índice. Por ejemplo:

[6]:
```
pd.Series([30, 35, 40], index=['2015 Ventas', '2016 Ventas', '2017 ventas'],␣
↪name='Producto A')
```

[6]:
```
2015 Ventas    30
2016 Ventas    35
2017 ventas    40
```

```
Name: Producto A, dtype: int64
```

12.3.1. Leyendo Archivos

La gran ventaja de Pandas es la lectura de tablas enteras conservando los nombres de columnas e índices. Se pueden leer archivos en ascii, de planillas de cálculo y que se están volviendo muy comunes que son los CSV. Estos son archivos donde los datos están separados por comas, de ahí el nombre: **CSV** o "Comma Separated Values"

El comando para leer archivos es pd.read_csv en caso de un archivo CSV

Es decir se lo usaría de la siguente forma:

[7]:
```
NGC2366_datos = pd.read_csv("NGC2366_all_corregido.csv")
```

Donde ahora el archivo **se asigna en la variable** (es decir un DataFrame) en este caso en particular en NGC2366_datos

Notar que no indiqué el tamaño del archivo, por lo cual es podría haber sido **muy grande**.

Si se quiere ver que tan grande es la tabla que se cargó en memoria, puedo preguntarlo con el comando:

[8]:
```
NGC2366_datos.shape
```

[8]:
```
(635514, 7)
```

Que indica que el archivo tiene 635514 filas y 7 columnas.

Pero si quiero ver el nombre de las columnas, tengo que verlo preguntando por el encabezamiento es decir:

[9]:
```
NGC2366_datos.head()
```

[9]:

	raj2000	dej2000	F555W	e_555	F814W	e_814	id
1	112.119762	69.188531	19.147	0.001	18.725	0.001	1
2	112.149837	69.205247	19.186	0.001	19.050	0.001	2
3	112.173885	69.191933	19.420	0.001	19.460	0.001	3
4	112.178112	69.189439	19.544	0.001	20.552	0.002	4
5	112.177623	69.189439	19.576	0.001	20.473	0.002	5

También puedo pedir información de toda la tabla. Por ejemplo, columnas con su nombre, cantidad de datos y tipo de variable e incluso la cantidad de memoria ram utilizada.

[10]:
```
NGC2366_datos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 635514 entries, 1 to 635514
Data columns (total 7 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   raj2000  635514 non-null  float64
 1   dej2000  635514 non-null  float64
 2   F555W    635514 non-null  float64
 3   e_555    635514 non-null  float64
```

```
 4   F814W    635514 non-null  float64
 5   e_814    635514 non-null  float64
 6   id       635514 non-null  int64
dtypes: float64(6), int64(1)
memory usage: 38.8 MB
```

También se puede indicar al leer que cierta columna debe ser usada como índice, con el argumento "index_col".

12.3.2. Guardando archivos

Y si necesitara salvar al disco un tabla en particular, lo puedo hacer con el comando:

NGC2366_datos.to_csv('nombre del archivo')

El formato CSV es uno de los posibles, Pandas soporta muchos más (xlsx, json, zip, txt, xml, html, pdf, docs, etc), incluyendo el hecho que cargando otras librerías puede acceder a leer lenguajes tipo SQL de Bases de datos.

Para realizar ciertas tares, Pandas me permite extraer datos de una columna individual, accediendo a ella a través de su nombre, por ejemplo:

[11]:
```
F555W = NGC2366_datos['F555W']
F814W = NGC2366_datos['F814W']
print(F555W,type(F555W))
```

```
1         19.147
2         19.186
3         19.420
4         19.544
5         19.576
           ...
635510    28.887
635511    28.888
635512    28.891
635513    28.893
635514    28.919
Name: F555W, Length: 635514, dtype: float64 <class 'pandas.core.series.
 ↪Series'>
```

Pudiendo de esta manera realizar operaciones entre los datos extraidos. Y luego puedo agregar mis resultados a la tabla pandas, como una nueva columna.

[12]:
```
Color = F555W - F814W

NGC2366_datos['Color'] = Color

NGC2366_datos.head()
```

[12]:

	raj2000	dej2000	F555W	e_555	F814W	e_814	id	Color
1	112.119762	69.188531	19.147	0.001	18.725	0.001	1	0.422
2	112.149837	69.205247	19.186	0.001	19.050	0.001	2	0.136
3	112.173885	69.191933	19.420	0.001	19.460	0.001	3	-0.040
4	112.178112	69.189439	19.544	0.001	20.552	0.002	4	-1.008

```
5  112.177623  69.189439  19.576  0.001  20.473  0.002   5 -0.897
```

[13]:
```
NGC2366_datos.describe()
```

[13]:
```
             raj2000        dej2000          F555W          e_555  \
count  635514.000000  635514.000000  635514.000000  635514.000000
mean      112.222952      69.212076      27.253348       0.107779
std         0.051012       0.024674       0.949578       0.062213
min       112.096683      69.154641      19.147000       0.000700
25%       112.189167      69.193513      26.794000       0.058800
50%       112.223148      69.212395      27.458000       0.100000
75%       112.257305      69.230119      27.917000       0.147000
max       112.352558      69.263490      28.919000       0.275000

              F814W          e_814             id          Color
count  635514.000000  635514.000000  635514.000000  635514.000000
mean       26.613298       0.098408  317757.500000       0.640050
std         1.050428       0.065590  183457.233824       0.530134
min        18.367000       0.000700       1.000000      -4.098000
25%        26.101000       0.048000  158879.250000       0.230000
50%        26.802000       0.084000  317757.500000       0.745000
75%        27.321000       0.134000  476635.750000       1.022000
max        28.365000       0.275000  635514.000000       6.059000
```

Incluso puedo pedir una evaluación de parámetros estadísticos tipo promedio, dispersión, máximos y mínimos, etc, en un sólo comando

[14]:
```
NGC2366_datos.F555W.describe()
```

[14]:
```
count    635514.000000
mean         27.253348
std           0.949578
min          19.147000
25%          26.794000
50%          27.458000
75%          27.917000
max          28.919000
Name: F555W, dtype: float64
```

O pedir exactamente el análisis de una columna en particular, por ejemplo el promedio, y lo hacemos indicando la columna por su nombre.

[15]:
```
NGC2366_datos.F555W.mean()
```

[15]:
```
27.253347675267584
```

Cuyo resultado puedo asignarlo a una variable particular

[16]:
```
F555W_mean=NGC2366_datos.F555W.mean()
print(F555W_mean)
```

```
27.253347675267584
```

Se Puede hacer cálculos en Pandas, pero si quisiera aprovechar la potencia y la versatilidad de NumPy, tendría que extraer los datos pandaa y convertirlos a arreglos Numpy. Y esta tarea es muy sencilla.

Para esto utilizo la función array( ) de NumPy.

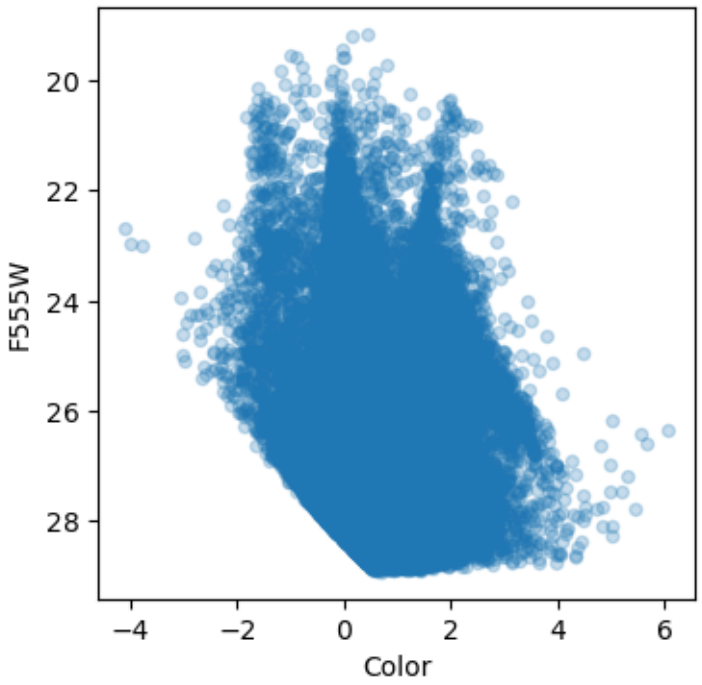
[17]:
```
import numpy as np
Color= np.array(NGC2366_datos['Color'])
print("Pandas")
print(NGC2366_datos['Color'])
print("")
print("NumPy")
print(Color)
```

```
Pandas
1         0.422
2         0.136
3        -0.040
4        -1.008
5        -0.897
           ...
635510    1.208
635511    0.625
635512    1.414
635513    0.922
635514    0.674
Name: Color, Length: 635514, dtype: float64

NumPy
[ 0.422  0.136 -0.04  ...  1.414  0.922  0.674]
```

Como vemos entonces para Pandas tengo una tabla y para NumPy un arreglo.

12.4. Gráficos en Pandas

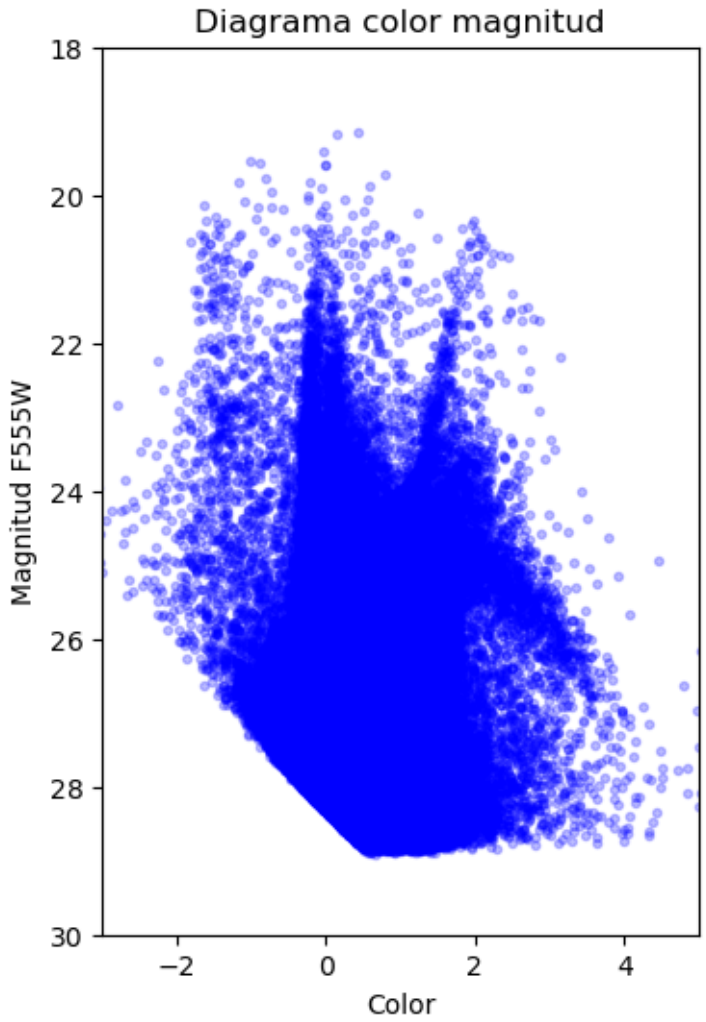
[18]:
```
NGC2366_datos.plot.scatter(x='Color',y='F555W',alpha=0.25,figsize=(4,4)).
↪invert_yaxis();
```



El método o función que agrego al final *.invert_yaxis()* es para que el eje F555W quede en una forma más conveniente para el uso astronómico, ya que menores valores de este filtro indican mayor brillo de la estrella.

Si lo hubiera realizado usando matplolib también hubiese sido posible, aunque es un poco más laborioso. Veámoslo:

[19]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(4,6))
plt.plot(NGC2366_datos['Color'], NGC2366_datos['F555W'], ".b",alpha=0.25)
plt.xlim([-3,5])
plt.ylim([30,18])  #note que por razones de interpretación invertí el eje al␣
 ↪igual que el gráfico anterior
plt.title("Diagrama color magnitud")
plt.xlabel("Color")
plt.ylabel("Magnitud F555W")
plt.show()
```

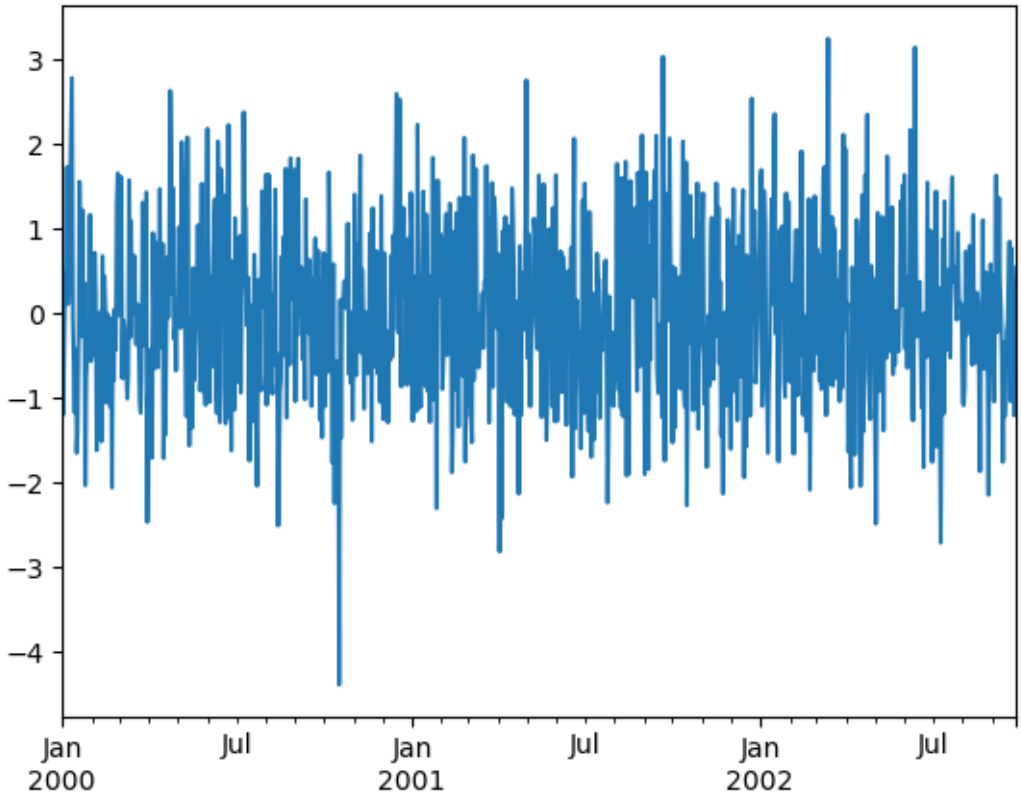


12.5. Series temporales

Pandas es muy útil para estudiar series temporales, es decir eventos que van sucediendo durante cierto intervalo de tiempo los cuales por ejemplo, tengo registrados en un archivo. Para ver como funciona Pandas con este tipo de situación generaremos una serie temporal con números al azar. Para hacerla más interesante crearemos una segunda serie, pero ahora de los valores de la primera acumulados. Es decir, el primer elemento es el primero de la segunda, pero el segundo elemento es primero mas el segundo y así continanos hasta el final.

Resumiendo: * Entonces creo la serie de valores al azar y la inicio en el 1/1/2000 y genero 1000 valores al azar y los correspondo a 1000 días diferentes (inicio= 1/1/2000, período=1000) * Acumulo esos valores en un segunda serie

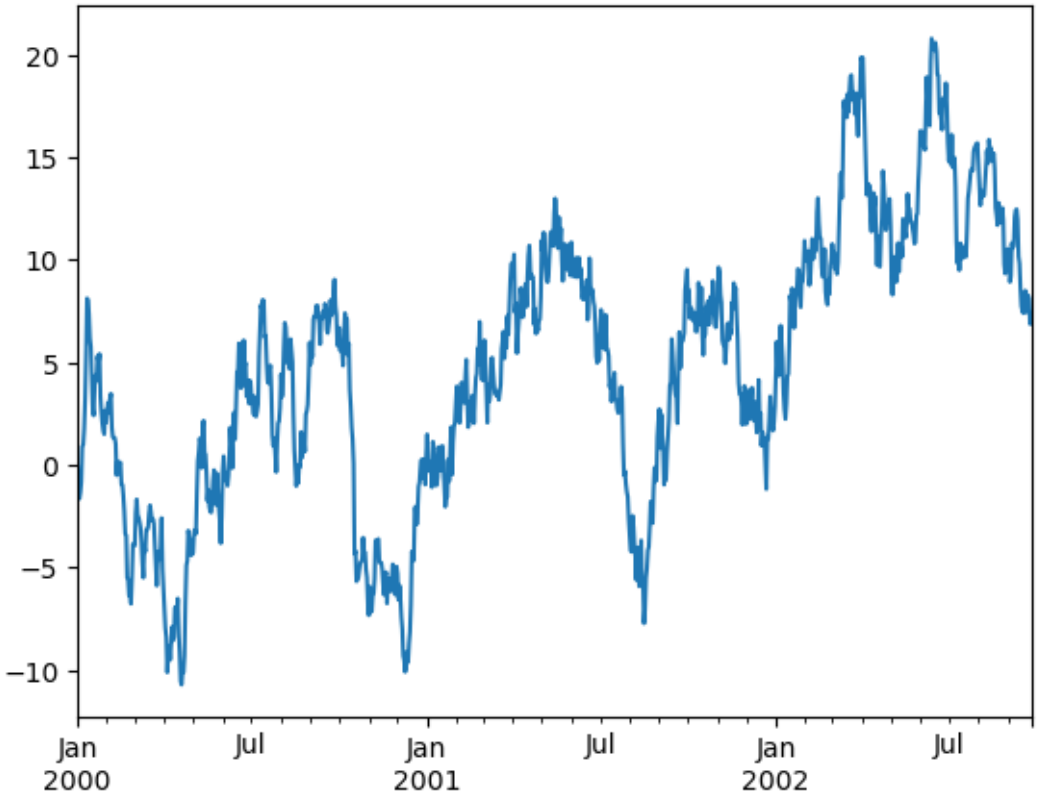
[20]:
```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000",␣
 ↪periods=1000))
ts.plot();
```



Como se le indicó periodo 1000 mil días, tenemos simulados un poco menos de 3 años de datos.

Genero la serie acumulada y para ahorrar memoria utilizo la misma variable.

[21]:
```
ts=ts.cumsum()
ts.plot();
```



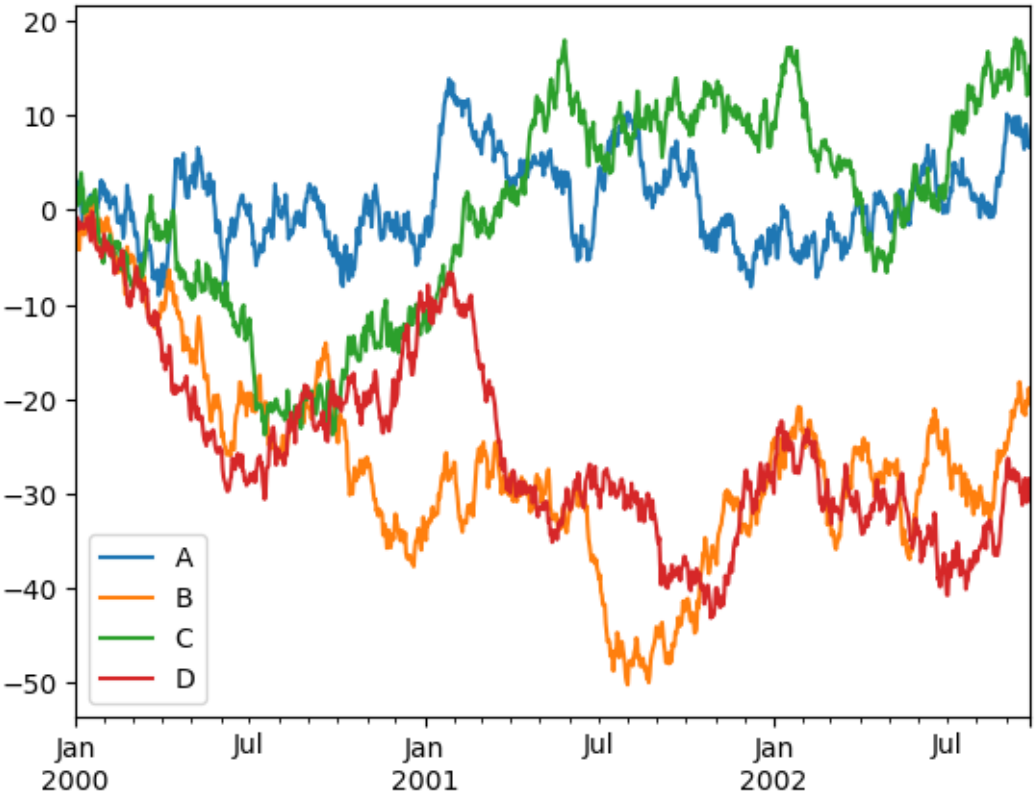
Describo sus propiedades

[22]:
```
ts.describe()
```

[22]:
```
count    1000.000000
mean        4.842470
std         6.620420
min       -10.727415
25%         0.120334
50%         5.210144
75%         9.481281
max        20.774382
dtype: float64
```

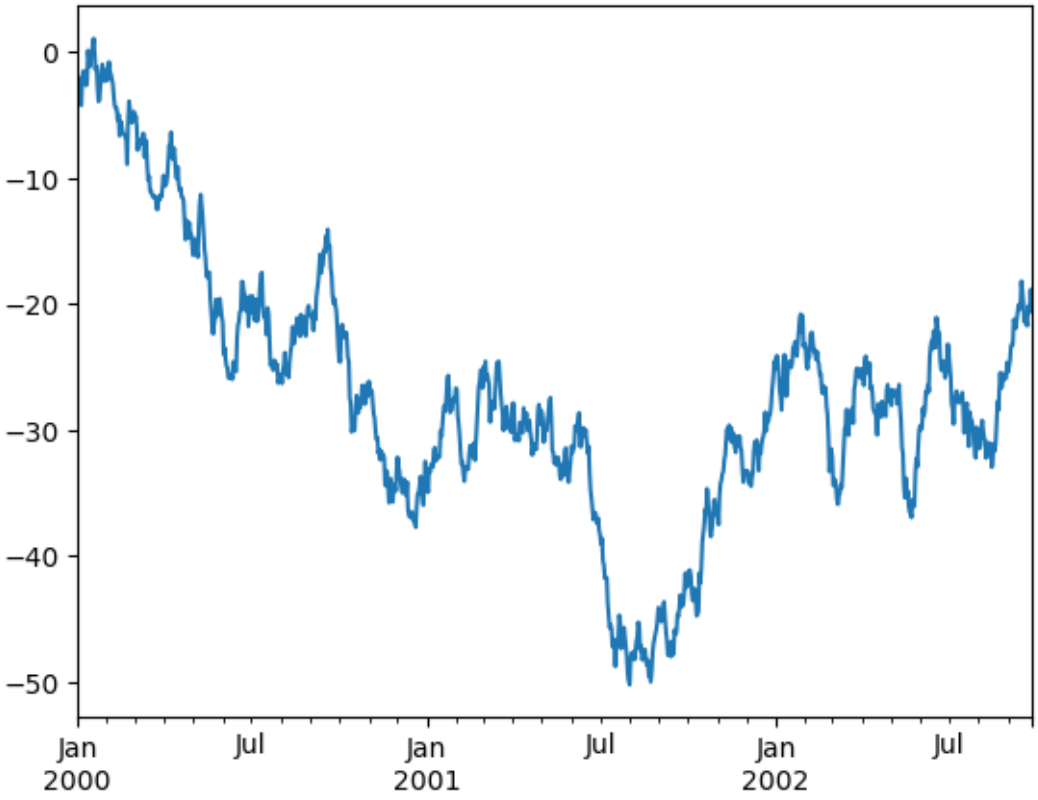
O en vez de una serie puedo crear varias que compartan el eje temporal y las guardo en un dataframe entero. Y todo esto en muy pocas órdenes:

[23]:
```
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,␣
 ↪columns=list("ABCD"))
df = df.cumsum()
df.plot();
```



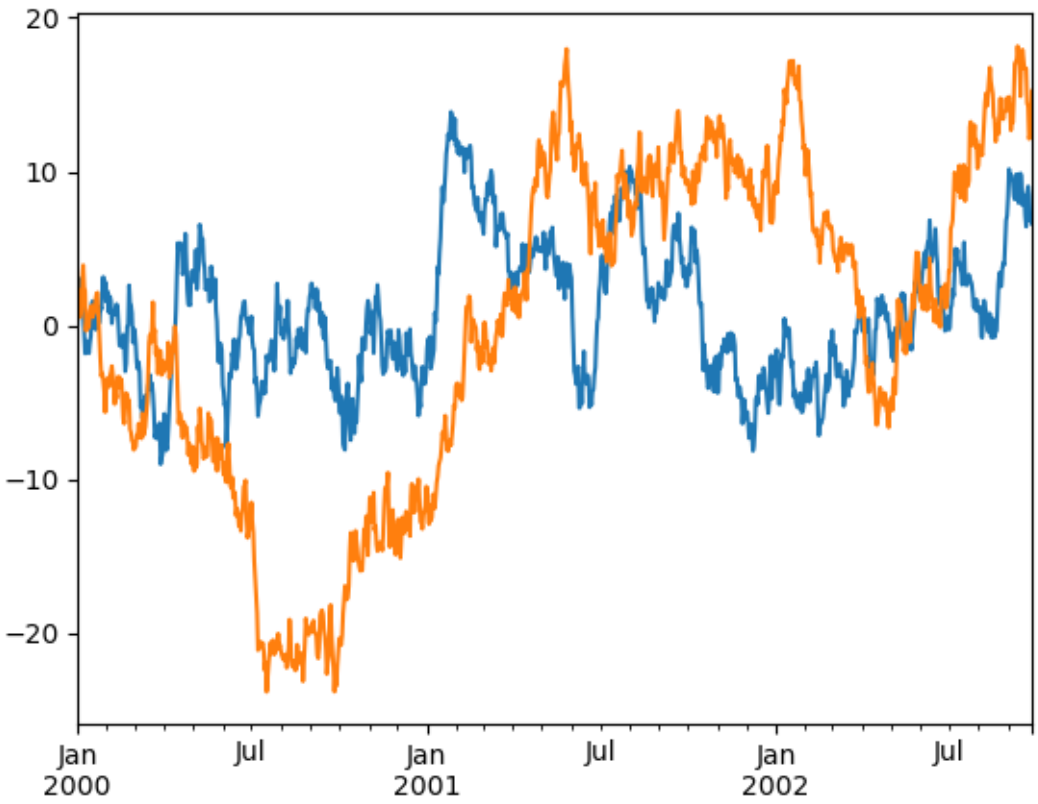
Pero si me interesa graficar sólo uno de las curvas, la pido por su nombre:

[24]:
```
df['B'].plot();
```



O puedo graficar de a dos, indicando cuales quiero como en el ejemplo anterior

[25]:
```
df['A'].plot()
df['C'].plot();
```



O graficar los puntos de 'A' contra los B, sin que se unan estos puntos con líneas (scatter plot in inglés). Alpha es la semi tranparencia, para que se vea la sobreposición de puntos.

[ ]:
```
df.plot.scatter(x='A',y='B', alpha=0.5);
```

También puedo hacer que se dibuje un gráfico tipo box, donde se indica el valor medio y el 20 % de todos los puntos, indicando con negro los puntos "outliers" (que son los que estadísticamente no parecen pertenecer a la muestra).

[ ]:
```
df.plot.box();
```

[ ]:
```
df=df.abs()  # hago esto para que todos los valores sean positivos
df.plot.area(figsize=(12,4),subplots=True);
```

Y podemos también hacer histogramas indicando el número de intervalos (el default es 10), pero acá pedimos que sean 20 intervalos (o "bins" en inglés)

[ ]:
```
df['A'].plot.hist(bins=20);
```

Pero también puedo hacerlo para todos los grupos al mismo tiempo:

[ ]:
```
df.plot.hist(bins=20, alpha=0.5);
```

CAPITULO 13

Astropy - Python en Astronomía

El proyecto AstroPy es un esfuerzo de la comunidad científica para desarrollar un paquete para uso astronómico utilizando Python como lenguaje. Sus principales metas son una mejora de usabilidad, interoperatibilidad aprovechando la colaboracion entre astronómos. La parte principal del software está pensado para el uso profesional de astrónomos y astrofísicos, pero podría ser utilizado por cualquiera desarrollando software para una aplicación astronómica o geodésica. El proyecto también incluye paquetes de software afiliados no programados por los desarrolladores principales pero que mantienen las metas que son similares a las de AstroPy y muchas veces incluso se basan en el software principal de AstroPy.

La página del proyecto con mucha información sobre este es: www.astropy.org Mientras que los paquetes de software afiliados se encuentran en affiliated.astropy.org

AstroPy se divide en varios temas en concreto en donde se ofrecen soluciones, estos son:

- Datos, sus estructuras y transformaciones
- Archivos, entrada/salida de datos y comunicación
- Cálculos y utilidades
- Sistema de AstroPy (documentación de como funciona - Nuts and Bolts)

Muchos de los ejemplos de este notebook son de la página www.astropy.org

Veremos sólo algunas de sus utilidades principales:

13.1. Constantes

Astropy.constants contiene un número muy importante de constantes físicas de uso astronómico. Cada una de estas constante es un objeto que contiene metadata describiendo su origen e incertezas.

Veamos algunos ejemplos:

[1]:
```
import numpy as np
import astropy
from astropy.constants import G
print(G)
```

```
  Name   = Gravitational constant
  Value  = 6.6743e-11
  Uncertainty  = 1.5e-15
```

```
  Unit  = m3 / (kg s2)
  Reference = CODATA 2018
```

[2]:
```
# también puedo hacer:
print(G.value)
# y utilizar este valor en un cálculo
```

```
6.6743e-11
```

Otra manera sería hacer un "import" todas las constantes y luego acceder a la que se quiere usar.

[3]:
```
from astropy import constants as const
print(const.G)
print("")
print(const.c)
```

```
  Name   = Gravitational constant
  Value  = 6.6743e-11
  Uncertainty  = 1.5e-15
  Unit  = m3 / (kg s2)
  Reference = CODATA 2018

  Name   = Speed of light in vacuum
  Value  = 299792458.0
  Uncertainty  = 0.0
  Unit  = m / s
  Reference = CODATA 2018
```

Las constantes están organizadas en módulos, y diferentes módulos pueden accederse en forma particular. Ejemplos de estos módulos son codata2010, codata2014 o codata2018 para constantes físicas. Para constantes astronómicas en posible acceder a constantes reguladas por la Union Astronómica Internacional (IAU) como iau2012 o iau2015. Esto se debe a que muchas constantes son remedidas constatemente y han cambiado su valor en la medida que se han obtenido mejores valores de estas.

Por ejemplo veamos el caso de la luminosidad solar:

[4]:
```
from astropy.constants import iau2012 as const
print(const.L_sun)
print("")
from astropy.constants import iau2015 as const
print(const.L_sun)
```

```
  Name   = Solar luminosity
  Value  = 3.846e+26
  Uncertainty  = 5e+22
  Unit  = W
  Reference = Allen's Astrophysical Quantities 4th Ed.

  Name   = Nominal solar luminosity
  Value  = 3.828e+26
  Uncertainty  = 0.0
```

```
Unit  = W
Reference = IAU 2015 Resolution B 3
```

13.2. Unidades

Todos sabemos la importancia de las unidades en cálculos, pero la astronomía mantiene dos sistemas unidades. El primero relacionado con las propias características de las observaciones astronómicas y el segundo MKS o CGS mucho más convencional y relacionado con parámetros de los laboratorios de física.

Astropy tiene la capacidad de manejar unidades, como atributos del objeto, pero con métodos que permiten el cálculo de transformación de unidades:

Para esta tarea, usamos el paquete **units** Si lo importo como **from astropy import units as u** todo lo que empieza con **u.algo** se considera que el valor está en unidades **algo**

En este ejemplo, que proviene de la página de AstroPy. En este caso quiero averiguar cuanto es la fuerza de atracción gravitatoria del Sol sobre un objeto de 100 kg a la distancia 2.2 unidades astronómicas. Pero cuando ejecute el comando print quiero que el resultado esté en Newtons.

$F = GM_{\odot}m_i/r^2$

[5]:
```
from astropy import constants as const
from astropy import units as u

F = (const.G  * const.M_sun * 100 * u.kg) / (2.2 * u.au) ** 2
print(F.to(u.N))    # <-- Preste atención al ".to"
```

```
0.12252238673869421 N
```

Pero si lo imprimo sin poner una unidad determinada

[6]:
```
print(F)
```

```
2.7419925619834707e+21 kg m3 / (AU2 s2)
```

También puedo convertir las constantes a la unidad que necesite. Por ejemplo, la velocidad de la luz en diferentes unidades.

[7]:
```
print(const.c)
print("")
print("La paso a kilómetros/segundo")
print(const.c.to('km/s'))
print("")
print("A Parsec/año")
print(const.c.to('pc/yr'))
print("")
print("O en CGS (Centímetro, Gramo, Segundo)")
print(const.c.cgs)
print("")
```

```
  Name   = Speed of light in vacuum
  Value  = 299792458.0
  Uncertainty  = 0.0
  Unit  = m / s
```

```
  Reference = CODATA 2018

La paso a kilómetros/segundo
299792.458 km / s

A Parsec/año
0.30660139378555057 pc / yr

O en CGS (Centímetro, Gramo, Segundo)
29979245800.0 cm / s
```

Como se puede ver las constantes tienen impresas sus unidades y como el sistema de unidades es bastante flexible, por ejemplo si saco la raíz cuadrada de la velocidad de la luz ($\sqrt{c}$)

[8]:
```
print(np.sqrt(const.c))
F2=F**2
print(F2**2)
```

```
17314.51581766005 m(1/2) / s(1/2)
5.6528191258897845e+85 kg4 m12 / (AU8 s8)
```

Para el caso de unidades redundantes es decir hay unidades del mismo tipo (distancia, tiempo, etc) en el numerador y en el denominador, la función decompose( ) puede ser usada para revizarlas y simplificarlas.

[9]:
```
t = 22.3 * u.kilometer / (130.51 * u.meter / u.second)

print(t)
print(t.decompose())
# o para F2
print(t.decompose())
```

```
0.17086813271013718 km s / m
170.8681327101372 s
170.8681327101372 s
```

También tengo unidades de longitud de onda, tanto en Ångstrom como nanometros (nm)

[10]:
```
lam = 5007 * u.angstrom

print(lam.to(u.nm))
print(lam.to(u.micron))
print(lam)
```

```
500.70000000000005 nm
0.5007000000000001 micron
5007.0 Angstrom
```

Aunque algunos cambios de unidadades necesitarían alguna información adicional. Por ejemplo, transformar electronvolts en unidades de longitud de onda.

[11]:
```
print(lam.to(u.eV, equivalencies=u.spectral()))
```

```
2.4762172644937133 eV
```

13.3. Coordenadas

AstroPy conoce todos los sistemas de coordenadas astronómicos y sus transformaciones

[12]:
```
from astropy.coordinates import SkyCoord
```

[13]:
```
# Escribo una coordenada de un astro
c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree)

print("RA:",c.ra)
```

```
RA: 10d41m04.488s
```

[14]:
```
print("La hora en RA:",c.ra.hour)
```

```
La hora en RA: 0.7123053333333335
```

[15]:
```
print("La Coordenadas en hms:",c.ra.hms)
```

```
La Coordenadas en hms: hms_tuple(h=0.0, m=42.0, s=44.299200000000525)
```

[16]:
```
print("La declinación:",c.dec)
```

```
La declinación: 41d16m09.012s
```

[17]:
```
print("La declinación en grados:",c.dec.degree)
```

```
La declinación en grados: 41.26917
```

[18]:
```
print("La declinación en radianes:",c.dec.radian)
```

```
La declinación en radianes: 0.7202828960652683
```

13.4. Transformaciones de Coordenadas

AstroPy conoce distintos sistemas de coordenadas y sus transformaciones

Cargo unas coordenadas

[19]:
```
c = SkyCoord(ra=14.68458*u.degree, dec=41.26917*u.degree)
```

[20]:
```
# A a coordenadas Galácticas
print(c.galactic)
```

```
<SkyCoord (Galactic): (l, b) in deg
    (124.40704063, -21.58166707)>
```

[21]:
```
# A grados, minutos y segundos
print(c.to_string('dms'))
```

```
14d41m04.488s 41d16m09.012s
```

[22]:
```
# A horas, minutos y segundos: Grados, minutos y segundos
print(c.to_string('hmsdms'))
```

```
00h58m44.2992s +41d16m09.012s
```

13.5. Tablas en Astropy

Proveen la capacidad de manejar tables heterogéneas con un modo de trabajo muy familiar para los usarios de las librerías Pandas y Numpy. De hecho son similares a una tabla Pandas básica (el rumor de pasillo es que usaron código de Pandas)

[23]:
```
from astropy.table import Table
```

[24]:
```
from astropy.table import QTable
import astropy.units as u
import numpy as np

a = np.array([1, 4, 5], dtype=np.int32)
b = [2.0, 5.0, 8.5]
c = ['x', 'y', 'z']
d = [10, 20, 30] * u.m / u.s

t = QTable([a, b, c, d],
           names=('a', 'b', 'c', 'd'),
           meta={'nombre': 'Primera tabla'})
```

[25]:
```
print(t)
```

```
 a   b   c    d
           m / s
--- --- --- -----
  1 2.0   x  10.0
  4 5.0   y  20.0
  5 8.5   z  30.0
```

Si no uso el comando print( ) la tabla se imprime en formato html (lenguaje de páginas web) y se activa dentro del notebook creando una tabla mucho más estilizada

[26]:
```
t
```

[26]:
```
<QTable length=3>
  a      b      c      d
                      m / s
int32 float64 str1 float64
----- ------- ---- -------
    1     2.0    x    10.0
    4     5.0    y    20.0
    5     8.5    z    30.0
```

[27]:
```
t.info
```

[27]:
```
<QTable length=3>
name  dtype   unit  class
---- ------- ----- --------
   a   int32         Column
   b float64         Column
   c    str1         Column
   d float64 m / s Quantity
```

[28]:
```
t.show_in_notebook()
```

[28]:
```
<IPython.core.display.HTML object>
```

Puedo ver la tabla sola en un tab de mi browser, con el siguente comando:

[29]:
```
 t.show_in_browser(jsviewer=True)
```

O se puede trabajar con una columna individual:

[30]:
```
t['a']
```

[30]:
```
<Column name='a' dtype='int32' length=3>
1
4
5
```

[31]:
```
P = t['a'] * 10
print(P)
```

```
 a
---
 10
 40
 50
```

E incluso podría agregar una nueva columna a la tabla.

[32]:
```
t['z'] = t['a'] * 20 +4

print(t)
```

```
 a   b   c    d    z
           m / s
--- --- --- ----- ---
  1 2.0   x  10.0  24
  4 5.0   y  20.0  84
  5 8.5   z  30.0 104
```