Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Formación Básica Departamento de Matemática

Introducción a

Autor: Jerónimo Gabriel Neder

Asignatura: Matemática Aplicada

Índice general

Capítulo 1

Introducción a Python

1.1. Preliminares

1.1.1. Algoritmo

Un algoritmo es en realidad un procedimiento por etapas. Es un conjunto de reglas que hay que seguir para realizar una tarea o resolver un problema en tiempo finito.

Mucho antes de la aparición de los ordenadores, los humanos ya utilizaban algoritmos. La receta de elevaración de una comida paticular, o las operaciones matemáticas para resolver un problema pueden considerarse algoritmos.

En el campo de la programación informática, los algoritmos son conjuntos de reglas que indican al ordenador cómo ejecutar una tarea. En realidad, un programa informático es un algoritmo que indica al ordenador qué pasos debe realizar y en qué orden para llevar a cabo una tarea específica. Se escriben utilizando un lenguaje de programación.

Existen distintos algoritmos para resolver un mismo problema. Las características de un algoritmo son:

- Un algoritmo debe ser preciso, es decir, debe indicar claramente, sin ambiguedades, cada uno de los pasos a seguir para conseguir el objetivo propuesto.
- Un algoritmo debe estar exacto, es decir, que si se sigue el algoritmo varias veces con el mismo juego de datos, los resultados obtenidos deben ser los mismos.
- Un algoritmo debe ser finito, de tiempo finito, su ejecución debe concluir en algún momento. ¿Quién ejecuta estas acciones? La respuesta es: un procesador. Un procesador es aquel sujeto o máquina que puede entender un enunciado y ejecutar el trabajo indicado en el mismo.

La sintaxis y semántica a utilizar para escribir un algoritmo depende del lenguaje que conoce el procesador, pues según cuál sea éste, habrá que describir las tareas con más o menos detalle.

Entonces, frente a un problema, sabiendo quien será el procesador y su lenguaje de comunicación, podremos describir la solución del problema mediante un algoritmo que pueda ser entendido y ejecutado sin ambigüedades.

1.1.2. Programa

Definiremos qué se entiende por el término programa. Un programa es un conjunto de acciones que puede entender y ejecutar una computadora. Otra definición podría ser: es un algoritmo traducido a algún lenguaje que pueda ser entendido por una computadora para poder ejecutarlo. Cada acción del programa se denomina instrucción.

Una instrucción es una combinación de palabras y símbolos que obedeciendo a la sintaxis propia de un lenguaje, son interpretados y utilizados por el computador para realizar una determinada acción.

Para llegar a hacer un programa es necesario el diseño previo del algoritmo. Esto es, representar la solución del problema en un lenguaje natural, en el cual se expresan las ideas diariamente.

Un lenguaje de programación es un conjunto de símbolos (sintaxis) y reglas (semántica) que permite expresar programas.

Los algoritmos deberían ser independientes tanto del lenguaje de programación en que se expresa el programa, como de la computadora que lo va a ejecutar.

1.1.3. Lenguajes de Programación

Los lenguajes de programación se pueden clasificar de la siguiente manera:

- Lenguaje de máquina
- Lenguaje de bajo nivel
- Lenguaje de alto nivel

Lenguaje de máquina

Es el sistema de códigos directamente interpretable por un circuito microprogramable, como el microprocesador de una computadora. Este lenguaje está compuesto por un conjunto de instrucciones que determinan acciones a ser tomadas por la máquina. Dicho de otra fomra, está escrito en un “idioma"que entiende el microprocesador (CPU, GPU). Las instrucciones son cadenas de 0 y 1 en código binario. Por ejemplo, la instrucción sumar los números 9 y 11 podría ser: 000111000111001110011011

Ventaja: la velocidad de ejecución de los programas es directa pues el microprocesador las entiende y ejecuta.

Desventaja: es un lenguaje de símbolos muy diferentes al nuestro, por lo tanto la codificación del programa se hace lenta y se pueden cometer muchos errores. Las instrucciones en código de máquina dependen del hardware de la computadora y por lo tanto difieren de una a otra.

Lenguaje de bajo nivel

Los lenguajes de bajo nivel por excelencia son los ENSAMBLADORES o Lenguaje ASSEMBLER. Las instrucciones son mnemotécnicos, como por ejemplo ADD, DIV, STR, etc. Una instrucción del ejemplo anterior sería ADD 9,11.

Ventaja: tienen mayor facilidad de escritura que el lenguaje anterior.

Desventaja: es un lenguaje dependiente del procesador (existe un lenguaje para cada tipo de procesador). La formación de los programadores es más compleja que la correspondiente a los programadores de al-

to nivel ya que exige no sólo técnicas de programación sino también el conocimiento del funcionamiento interno de la máquina. Se utiliza en la programación de PLC, control de procesos, aplicaciones de tiempo real, control de dispositivos. Un programa escrito en un lenguaje de este tipo no puede ser ejecutado directamente por el procesador, y debe ser traducido a código de máquina mediante un ensamblador. No se debe confundir, aunque en español adoptan el mismo nombre, el programa ensamblador ASSEMBLER encargado de efectuar la traducción del programa a código de máquina, con el lenguaje de programación ensamblador ASSEMBLY LENGUAGE.

Lenguaje de alto nivel

Está diseñado para que las personas escriban y entiendan los programas de modo mucho más natural, acercándose al lenguaje natural. Una instrucción del ejemplo anterior sería `9+11`. Los lenguajes de alto nivel son los más populares y existe una variedad muy grande. Algunos ejemplos son C,C++, Fortran, Java, Python, SQL.

Ventajas: son los más utilizados por los programadores. Son independientes de la máquina, las instrucciones de estos lenguajes no dependen del microprocesador, por lo cual se pueden correr en diferentes computadoras. Son transportables, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras. La escritura de los programas con este tipo de lenguajes se basa en reglas sintácticas similares a nuestro lenguaje, por lo tanto el aprestamiento de los programadores es mucho más rápida que en los lenguajes anteriormente nombrados.

Desventajas: Ocupan más lugar de memoria principal (RAM) que los anteriores. El tiempo de ejecución es mayor pues necesitan varias traducciones.

1.2. Python

¿Qué tipo de lenguaje de programación es Python?

- De alto nivel (sintaxis con una "lógica humana.[en] inglés, no pretende proveer control a las direcciones de memoria en la que se guardan datos y la manera de interactuar con el microprocesador (por ejemplo).
- Programación dinámica (ventaja y desventaja): no hay que compilar y podemos (a través de un intérprete) ver de inmediato el resultado del programa que vamos creando al programar pero al requerir la traducción del intérprete, la performance es en general mucho menor que los lenguajes que requieren compilación de los códigos previo a su ejecución.
- De propósito general: No tiene un propósito específico definido. Se puede usar para crear aplicaciones:
  1. para producir gráficos,
  2. científicas (como Fortran),
  3. web o para smartphones (Instagram es una aplicación que usa Python),
  4. de bajo nivel que interactúen de manera directa con hardware (robótica), etc.

¿Por qué Python?

- Es muy flexible por ser de propósito general pero a pesar de eso es relativamente sencillo de usar
- Existen muchas librerías de aplicaciones específicas que facilitan enormemente el desarrollo de aplicaciones

- Es software libre y fácil instalar en múltiples OS (Linux, MacOS, MS WIndows)
- Tiene una gran comunidad activa.
- Hay muchos temas que actualmente cuyas aplicaciones se programan mayormente en Python (Machine Learning, Deep Learning, etc.)

Además, el desarrollo de Python ha sido claramente influenciado por Java y C++, pero también muestra una notable similitud con **MATLAB** (otro lenguaje interpretado muy popular en computación científica). Python implementa los conceptos habituales de los lenguajes orientados a objetos, como clases, métodos, herencia, etc. Sin embargo, en este Curso no utilizaremos la programación orientada a objetos. El único tipo de datos que necesitamos es el array N-dimensional disponible en la librería Numpy. Para tener una idea de las similitudes y diferencias entre MATLAB y Python, veamos los códigos escritos en ambos lenguajes para la solución de ecuaciones simultáneas Ax = b mediante la eliminación gaussiana. Despreocupandonos del algoritmo en sí, veamos la semántica. Aquí está el código de la función escrita en MATLAB:

```
function x = gaussElimin(a,b)
n = length(b);
  for k = 1:n-1
    for i= k+1:n
      if a(i,k)  = 0
        lam = a(i,k)/a(k,k);
        a(i,k+1:n) = a(i,k+1:n) - lam*a(k,k+1:n);
        b(i)= b(i) - lam*b(k);
      end
    end
  end
for k = n:-1:1
  b(k) = (b(k) - a(k,k+1:n)*b(k+1:n))/a(k,k);
end
x = b;
```

La función equivalente en Python es:

```
from numpy import dot
def gaussElimin(a,b):
    n = len(b)
    for k in range(0,n-1):
        for i in range(k+1,n):
           if a[i,k] != 0.0:
                lam = a [i,k]/a[k,k]
                a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
                b[i] = b[i] - lam*b[k]
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - dot(a[k,k+1:n],b[k+1:n]))/a[k,k]
    return b
```

El comando `from numpy import dot` le indica al intérprete que cargue la función `dot` (que calcula el producto interno de dos vectores) desde la librería `numpy`. El operador de dos puntos (`:`), conocido como

operador de segmentación en Python, funciona de la misma manera que en MATLAB: define un segmento de un array.

La declaración `for k = 1:n-1` en MATLAB crea un bucle que se ejecuta con `k = 1, 2, ... , n  1`. El mismo bucle aparece en Python como `for k in range(n-1)`. En este caso, la función `range(n-1)` crea la secuencia `[0, 1, ... , n  2]`; luego, `k` itera sobre los elementos de la secuencia. Las diferencias en los rangos de `k` reflejan los desplazamientos nativos utilizados para los arrays. En Python el índice del primer elemento de la secuencia siempre es 0. En contraste, en MATLAB es 1.

También observe que Python no tiene declaraciones `end` para terminar bloques de código (bucles, subrutinas, etc.). El cuerpo de un bloque se define mediante su sangría (identación); por lo tanto, la sangría es una parte integral de la sintaxis de Python.

Al igual que MATLAB, Python distingue entre mayúsculas y minúsculas. Por lo tanto, los nombres `n` y `N` representan objetos diferentes.

En este archivo se puede ver una lista de comandos de MATLAB y su traducción a Python.

Enlaces útiles:

Sitio Oficial Python

Sitio oficial Numpy (Documentación)

Sitio oficial Spyder

Sitio Oficial JupyterLab

Sitio Oficial Anaconda (Incluye Spider y Jupyter Lab

Google Colab

1.2.1. Instalación y uso de Python

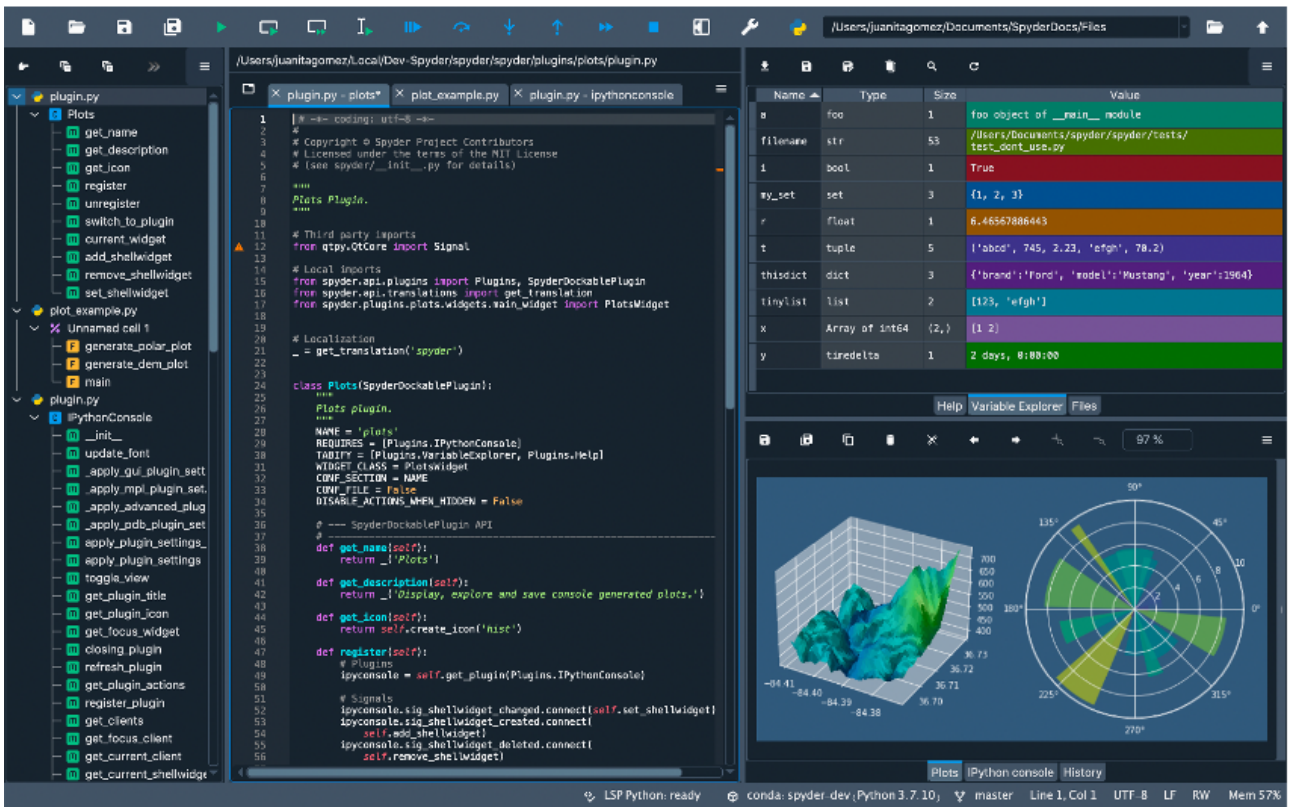
- Descargar del sitio oficial la última versión de Python (ver enlace arriba)

Para utilizar python en nuestra computadora, una ves instalado, descargamos el software Spyder (ver enlace arriba).

1.2.2. Spyder

Spyder es un entorno científico gratuito, de código abierto escrito en Python, para utilizar Python, diseñado por y para científicos, ingenieros y analistas de datos. Presenta una combinación única de funciones avanzadas de edición y visualización, como tambien una ejecución interactiva.

1.2.3. Entorno de Spyder



1.2.4. Qué es un script en Python?

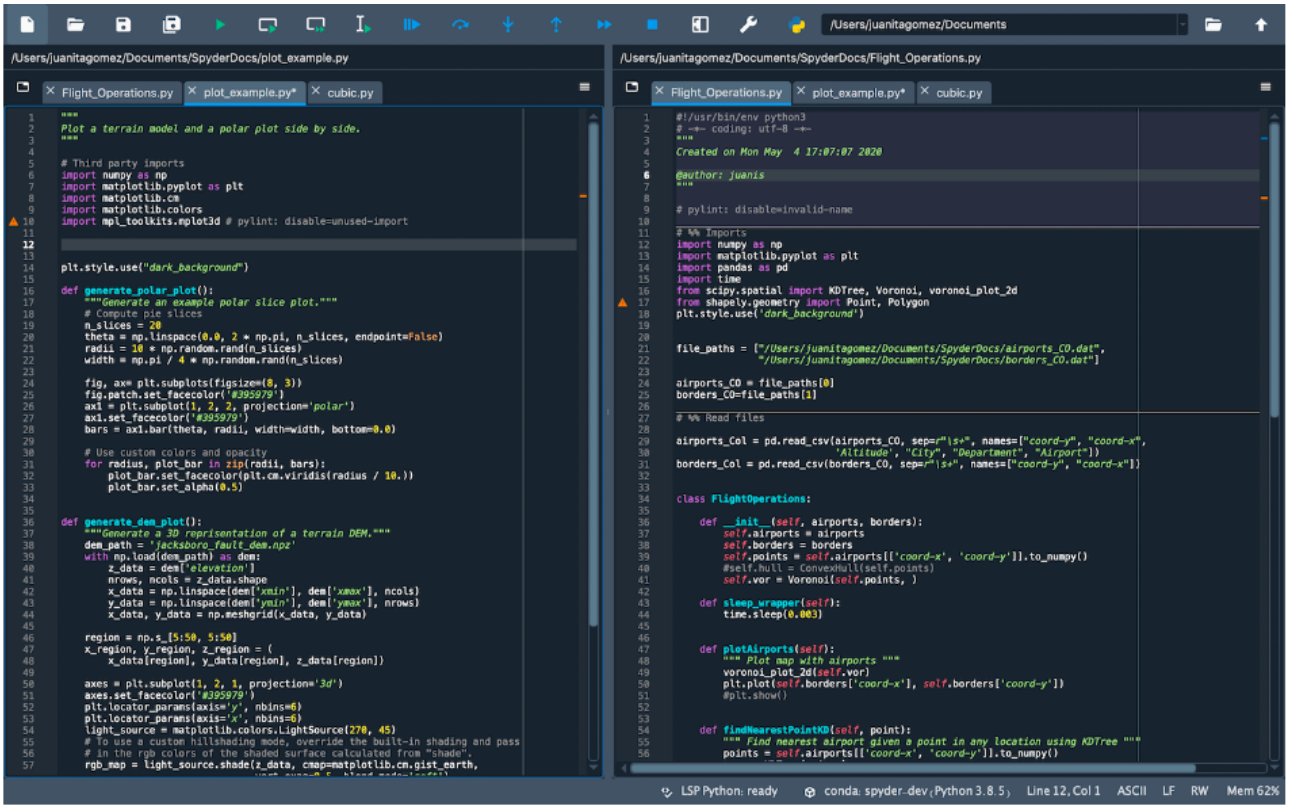
Un script es un código particularmente creado para resolver una tarea específica. Normalmente cumple una o varias de las siguientes características:

Se compone de un único módulo de python (un fichero de texto con extensión .py)

- Tiene un punto entrada/ejecución al final del mismo.
- Hace uso de librerías del sistem.
- Se lanza por consola.
- Define diferentes funciones simples.
- Permite el uso de argumentos desde la consola.

Crear un script en **Spyder**: Entre sus funciones Spyder contiene el editor, el cual nos permitirá crear nuestro archivos `.py`.

1.2.5. Entorno de Spyder (editor):



1.2.6. Anaconda

Anaconda es una distribución de Python. Está orientada a dar soporte integral al desarrollo científico con Python tanto analítico como gráfico. Entre sus características destacan:

1. Aplicación gráfica que integra las herramientas y versiones de Python orientadas al desarrollo científico.

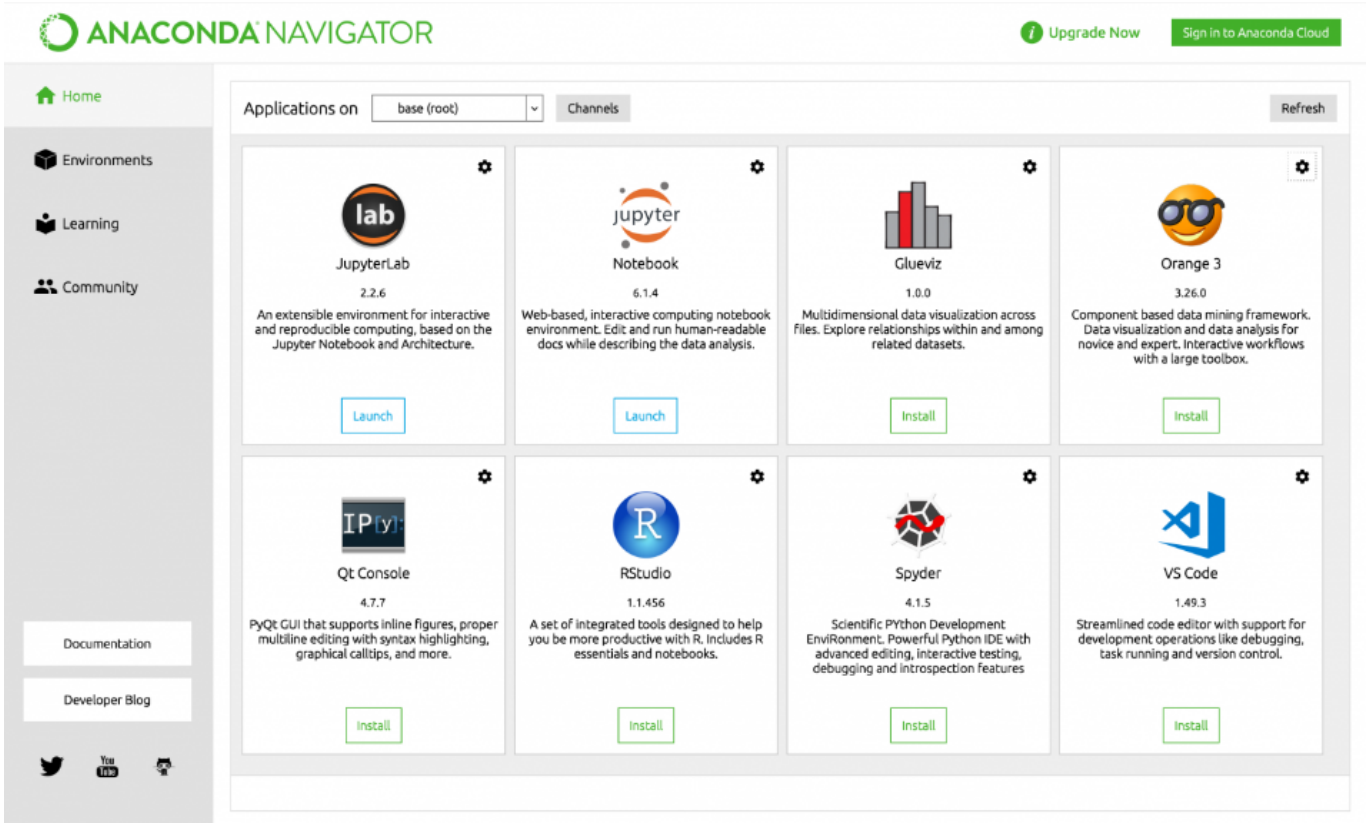
2. IDE para desarrollar fácilmente aplicaciones científicas, tanto en código como creación de gráficos.

3. Gestor de paquetes integrado (Conda). Integración avanzada con Jupyter Notebook. Soporte de entornos virtuales de Python.

4. Permite utilizar Python y R (Lenguaje de progamación) en el mismo entorno.

Anaconda es mantenida y desarrollada por la compañia https://anaconda.org/, que es la responsable de mantener y evaluar la herramienta y los paquetes disponibles del repositorio de Conda.

Aclaración: Spyder está incluido en Anaconda. Si se busca reducir la cantidad de memoria en disco (en uso) basta con usar Spyder independientemento de Anaconda, como lo mencionamos en la sección anterior.



Entorno Anaconda

1.2.7. Google Colab

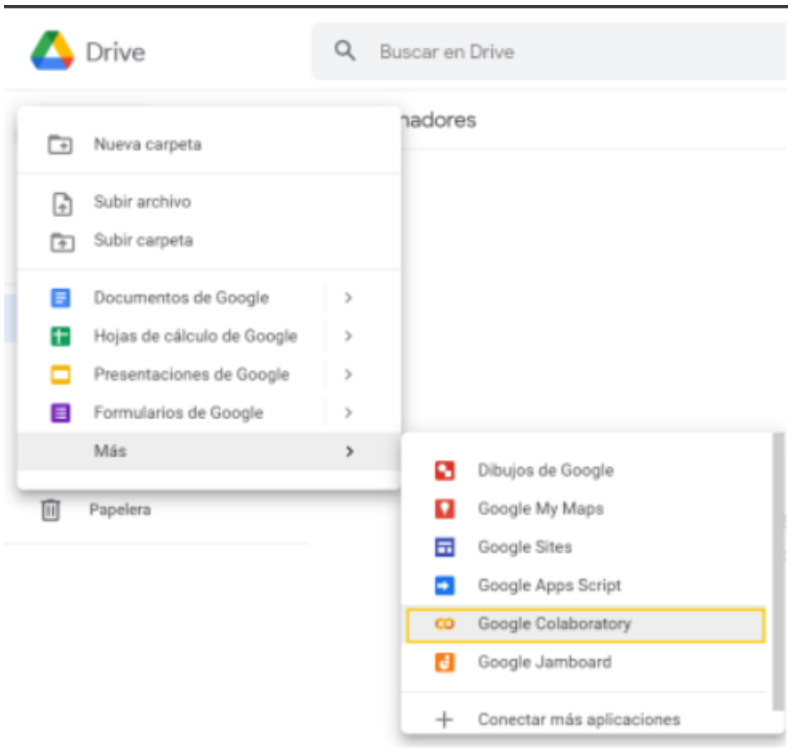
Google Colab (Ver enlace Arriba) es una representación en línea de Jupyter Notebook desarrollada por Google. Mientras que Jupyter Notebook requiere instalación en un ordenador y sólo puede utilizar los recursos locales de la máquina, Colab es una aplicación en la nube completa para Python codificación (requiere conexión a internet).

Podemos escribir códigos Python utilizando Colab en tus navegadores web Google Chrome o Mozilla Firefox. También podemos ejecutar esos códigos en el navegador sin necesidad de ningún entorno de ejecución o interfaz de línea de comandos.

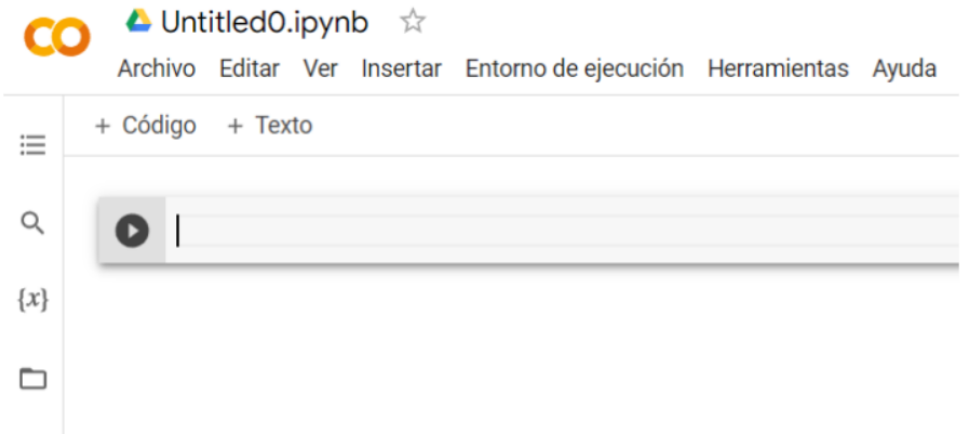
Además, nos permite dar a un Proyecto Python un aspecto más profesional añadiendo ecuaciones matemáticas, gráficos, tablas, imágenes y otros gráficos.

1.2.8. Entorno Colab

Colab forma parte de las aplicaciones de Google en la nube, por lo tanto se accede a través de Google Drive usando una cuenta personal. Para ingresar a un cuaderno Colab, ingresamos a Google Drive y a la izquierda de la pantalla hacemos click en +Nuevo. Allí nos encontraremos con lo siguiente:



Un cuaderno es entonces un documento que contiene c´odigo ejecutable (por ejemplo, Python) y tambi´en elementos de texto enriquecido (links, figuras, etc.). Es nuestro ((entorno de trabajo)) y se ve de la siguiente manera:



En la parte superior se encuentra el nombre del cuaderno (pudiendo cambiarlo) y su formato `.ipynb`, que viene de IPython Notebook y es un formato que nos permite ejecutar cuadernos tanto en IPython, como en Jupyter y Colab.

Un cuaderno está compuesto por celdas. Una celda es la unidad mínima de ejecución dentro de un cuaderno, es decir, es donde incluimos nuestro código y lo ejecutamos. Para ejecutar una celda podemos pulsar el botón con el icono de Play que se encuentra a la izquierda o pulsando Ctrl+Enter (ejecutar celda) o Shift+Enter (ejecutar celda y saltar a la siguiente). Tras la ejecución, debajo de la celda encontramos el resultado (si lo tiene).

Desde los botones de la parte superior o en el menú «Insertar» podemos añadir nuevas celdas, tanto específicas para código como para texto. La distinción que hace Colab sobre ellas es que las celdas de código son ejecutables, mientras que las de texto muestran directamente el texto que incluyamos.

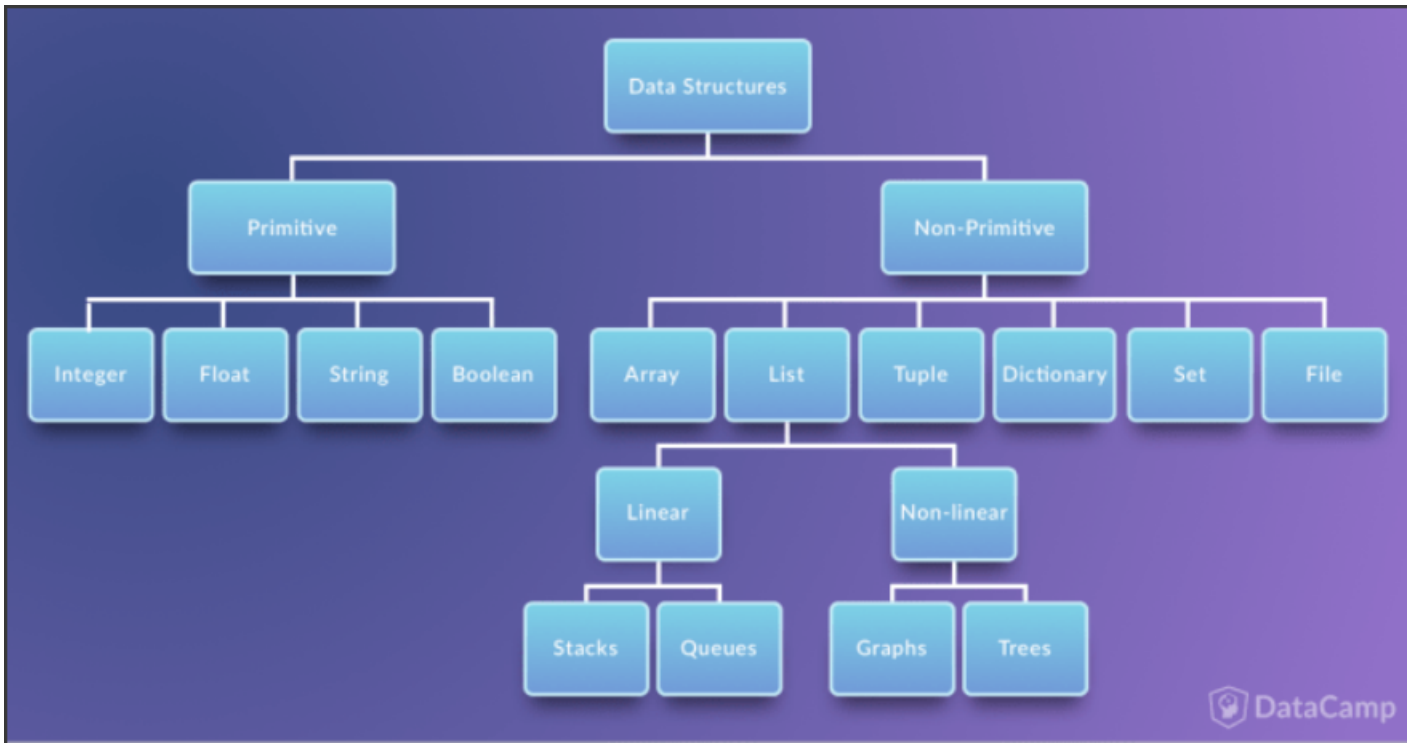
Una de las ventajas de Google Colab, aparte de poder trabajar en la nube y compartir las notebooks con otros usuarios, es que el entorno incorpora librerías Python sin necesidad de descargarlas.

Cada celda es independiente, pero todas las celdas en un cuaderno utilizan el mismo kernel. El estado del kernel persiste durante un tiempo, con lo que, aunque cada celda sea independiente, las variables declaradas en una celda se pueden utilizar en las celdas siguientes.

1.3. Variables y estructura de datos

1.3.1. Estructura de datos

Programar implica manipular datos de una manera sistemática. Comencemos por conocer los tipos de datos que el lenguaje permite manipular. Las principales estructuras de datos que maneja Python se ilustran en la siguiente figura:



NOTA 1: Además de Integer y Float, también existe el tipo de variable Complex similar a MATLAB (para manipular números complejos), a pesar de no aparecer en la figura.

NOTA 2: Si bien en esta figura, aparece Array como otro tipo de estructura de datos de Python, es importante aclarar que Python estándar no incluye soporte para arrays. Un array es similar a (aunque es menos menos general que) una lista y admite operaciones no definidas para las listas. Para trabajar con arrays es necesario importar librerías que incluyan procedimientos que operen con ellos como por ejemplo NumPy (Descargar en enlace arriba).

Las estructuras de datos y variables con las que más frecuentemente trabajaremos son:

1. String (texto: cadenas de caracteres, equivalente a las variables de tipo "String"de MATLAB)

2. Números:

   a) Integer (entera, equivalente a las variables de tipo ïnteger"de MATLAB)

   b) Float (de punto flotante, equivalente a las variables de tipo Float de MATLAB)

   c) Complex (compleja, equivalente a las variables çomplex"de MATLAB)

3. Boolean (variable booleana o lógica, equivalente a variables "lógicas"de MATLAB): True/False

4. None (variable nula

5. List (lista)

6. Tuple (tupla: lista inmutable)

7. Dictionary (diccionario)

Para saber de que tipo a quedado definida una variable, podemos usar la función intrínseca de Python: `type()` cuya utilización se muestra a continuación con un ejemplo. En el mismo usamos el hecho que el resultado de aplicar una función a una variable, puede ser usado a su vez como argumento de otra función de Python (como era predecible que sucediera) y por lo tanto podemos usar: `print(type(a))` para imprimir en pantalla, el tipo de la variable a.

```
a1 = 2 # "a1" es una variable tipo integer
a2 = 2.0 # "a2" es tipo float
a3 = 0+1j # "a3" es tipo complex
a4 = True # (o a4= False ) "a4" es tipo booleana
a5 = [1 ,2 ," hola "] # "a5" es una lista
a6 = (1 , 2, " hola ", 1j) # "a6" es una tupla
a7 = {" Nombre " : " Alumno ", " Materia ": "Matemática Aplicada","Carrera":"Ingeniería
a8 = None # "a8" es una variable sin tipo 'NoneType '

b1 = " Hola Matemática Aplicada "
print (b1, type (b1))
b2 = 2
print (b2, type (b2))
b3 = 2.0
print (b3, type (b3))
b4 = 3/2
print (b4, type (b4))
b5 = 1+1j
print (b5, type (b5))
b6 = (1+1j) **2
print (b6, type (b6))
b7= True
print (b7, type (b7))
b8=[1 ,2 ," hola "]
print (b8, type (b8))
b9=(1 ,2 ," hola " ,2+1j)
print (b9, type (b9))
b10 = {" Nombre " : " Juan ", " Apellido ": "Pedro","Altura":1.87}
print (b10, type (b10))
b11 = None
print (b11, type (b11))
 print(5+6)
print(8-1/2)
'Concatenar'+'Frase' #el operador cuando es aplicado a strings
#python automáticamente entiende que queremos concatenar
```

```
 Hola Matemática Aplicada  <class 'str'>
2 <class 'int'>
2.0 <class 'float'>
1.5 <class 'float'>
(1+1j) <class 'complex'>
2j <class 'complex'>
True <class 'bool'>
[1, 2, ' hola '] <class 'list'>
(1, 2, ' hola ', (2+1j)) <class 'tuple'>
{' Nombre ': ' Jerónimo ', ' Apellido ': 'Neder', 'Altura': 1.87} <class 'dict'>
```

```
None <class 'NoneType'>
```

1.3.2. Operaciones aritméticas

Ahora que sabemos un poco más como trabaja Python con distintos tipos de datos, veamos otras operaciones numéricas que tiene definidas:

- El operador + suma dos números,
- El operador - los resta,
- El operador * para multiplicar,
- El operador dividir con /,
- El operador // indica una division entera (por ejemplo, 10 // 8 daría 1, porque 10 entra solo una vez en 8 de forma entera).
- El operador ** es el encargado de .elevar a la".

```
print(5+6)
print(8-1/2)
'Concatenar'+'Frase' #el operador cuando es aplicado a strings python automáticamente
```

```
11
7.5
'ConcatenarFrase'
```

1.3.3. Comentarios en el código

Para realizar comentarios en el código de Python utilizamos y a continuación escribimos nuestro comentario. Tambien podemos hacer comentarios de varias lineas de codigo utilizando ''' o "

```
#Esto es un comentario

""" Esto es un comentario
    de varias
    líneas
"""

'''Esto es un comentario
    de varias
    líneas
'''
```

1.3.4. Variables

Hasta ahora solo estuvimos usando a Python para ejecutar programas de una sola linea, no mucho más de lo que podemos hacer con la calculadora. Si queremos hacer cosas que requieren más de una linea,

rápidamente nos vamos a dar cuenta que nos viene muy bien poder guardar los resultados anteriores.

Si queremos guardar un dato, le debemos asignar un nombre. Una vez hecho esto, Python guardará el dato en una variable con ese nombre.

Para asignar un dato a una variable, usamos el signo =. Por ejemplo, debajo le asignamos el entero 5 a la variable x.

```
x = 5
s = 'Cadena de caractéres'
```

Luego podemos acceder al número referenciando la variable x:

```
print(x)
print(s)

# y = x**2
# print(y)

# xx = "Universidad"
# print(xx)

# print(y)
```

```
5
Cadena de caractéres
```

1.3.5. Formato al texto

Vamos a ver un poco como dar formato a los strings. Esto no solo nos interesa para poder imprimir cosas en pantalla de una forma más elegante, sino que además es muy útil para manejar instrumentos en el laboratorio.

Para imprimir un texto junto a el valor de una variable en el ejemplo anterior, realizamos una linea de este estilo

```
print("La siguiente variable", nombre_variable, .es muy importante.")
```

que en principio no tiene nada de malo, pero es difícil de leer en el código y se puede poner peor si hay que poner más variables.

Dicho esto, una manera de reesrcribir esta linea de forma más elegante es usando algo que se conoce como un f-string (format string).

```
print(f"5 + 3 es igual a {5+3}") #esto es un format string
```

```
5 + 3 es igual a 8
```

1.3.6. Operaciones Booleanas (condicionales)

Tipo bool Como mencionamos antes, una variable booleana solo posee dos valores posibles: True y False. Son especialmente útiles para verificar si una condición se cumple (`True`) o no (`False`).

Estas variables se comportan bajo el álgebra de Bool, por lo que permiten hacer operaciones lógicas si se está familiarizado con esta. Además nos permiten verificar (des)igualdades.

```
# ¿2 es mayor que 4?
print(2 > 4)
# ¿3² es distinto de 2³?
print(3**2 != 2**3)
# ¿Se encuentra la letra 'o' en el string 'Euler'?
print('o' in 'Euler')
```

Los símbolos `>`, `!=` son operadores de comparación. Acá una lista de los más usados:

- a > b verifica si a es mayor que b
- a < b verifica si a es menor que b
- a >= b verifica si a es mayor o igual que b
- a <= b verifica si a es menor o igual que b
- a != b verifica si a es distinto que b
- a == b verifica si a es igual que b

Es importante destacar que para verificar la igualdad se usa == y no = ya que este último está reservado para asignar un valor.

Por otra parte, a in b verifica si a está contenido en la secuencia b. Esto resulta útil para muchas cosas además de verificar si una palabra contiene una letra.

En todos estos casos, los resultados de la operación son booleanos

1.3.7. Condicionales (`if, elif y else`)

La estructura de los `if` tiene la siguiente forma:

```
if Condición:
    |Sentencia1
    |Sentencia2
```

Importante: Las líneas que inician con if terminan con : (dos puntos). Además, en las líneas que siguen a los :, las barritas | marcan que esa parte de código está identada (tiene una sangría), lo que implica que es una parte que se va a ejecutar únicamente si la condición que está arriba (y sin identar) se cumple. O sea, podemos pensar que hay un bloque de código asociado a ese if, y todo en ese bloque debe poseer la misma indentación.

La estructura `if, else` es la siguiente:

```
if Condición:
    |Sentencia1
else
    |Sentencia2
```

Si la condición es cierta se ejecuta `Sentencia1`, si no (else) ocurre `Sentencia2`

Ejemplos:

```
#if

r=1

if r != 0:
    print((f'{r} es distinto de cero.'))

#doble if, para condiciones independientes
num = 5

if num > 0:
    print(f'{num} es positivo.')

if num <  10:
    print(f'{num} es menor que 10.')

num1 = -42

#if, else.

if num1 > 0:
    print(f'{num1} es positivo.')
else:
    print(f'{num1} es cero o negativo.')
```

```
1 es distinto de cero.
5 es positivo.
5 es menor que 10.
-42 es cero o negativo.
```

La herramienta fundamental, para elegir si se cumple una condición extra a una dada, es el `elif` (abreviatura de else if), que nos permite chequear condiciones extra si no se cumplen las anteriores. Esto es:

```
if bool_1:
    |esto se ejecuta
    |si bool_1 es True

elif bool_2:
    |esto se ejecuta
    |si bool_1 es False
    |pero bool_2 es True

elif bool_3:
    |esto se ejecuta
    |si bool_1 es False,
    |si bool_2 es False
    |pero bool_3 es True

...(elif bool_4 etc)

else:
      |esto se ejecuta
      |si todas las bool_n
      |son False
```

Nota sobre los bloques de código:

Como se mencionó arriba, las líneas de código asociadas a un mismo bloque deben poseer la misma cantidad de espacios en su indentación. Recomendamos usar 4 espacios para indentar (google colaboratory por defecto pone 2 espacios al apretar tab pero puede cambiarse en la configuración del editor).

1.3.8. Listas

Así como los strings se definen con abriendo y cerrando comillas, las listas se definen abriendo y cerrando corchetes. Los elementos se separan con comas.

```
verduras = ['Zanahoria', 'Lechuga', 'Cebolla', 'Papa']
```

Cada elemento en la lista tiene un índice que hace referencia a su posición en la lista.

¿Cuál es el primer elemento? ¿Cómo nos referimos a él?

En Python se empieza a numerar desde el 0. En este caso el primer elemento de la lista de verduras es "Zanahoria".

```
verduras[0]  # Pedimos el primer elemento de la lista
print(verduras[1])  # Pedimos el segundo elemento de la lista

#Tambien podemos empezar a contar de atrás para adelante. El último elemento de la lis
#Es decir, tenemos dos formas de referirnos a un elemento de la lista.

print(verduras[3])  # Pedimos el cuarto elemento de la lista

print(verduras[-1])  # Pedimos el último elemento de la lista
```

```
Lechuga
Papa
Papa
```

Es importante notar que las listas pueden tener elementos de distintos tipos adentro. También podemos guardar variables que hayamos definido antes.

```
Alumno = 'Jerónimo'

lista1 = [1.42, 23, 'Sofá', True, Alumno, 'Javier', "12345", 12.5, False]

print(lista1[-1])

print(lista1[4])
```

```
False
Jerónimo
```

1.4. Iteraciones

1.4.1. `For (loops)`

Un ciclo de tipo `for` permite ejecutar código una determinada cantidad de veces. Supongamos que necesitamos código que para un $n \in \mathbb{N}$ perteneciente a $[0, 3]$, muestre verduras$[n]$, de la lista definida arriba.

```
for n in range(0,4):  # range() sequencia de numeros de 0 a 3
    print(verduras[n])

#Además no es necesario especificar el inicio en 0, range lo hace por defecto.
#El número donde empieza el range es un parámetro opcional,
#si no le decimos nada, empieza directamente desde 0.

print(f'El tamaño de la lista de verduras es: {len(verduras)}  \n')

for n in range(len(verduras)):
    print(verduras[n])

#De la misma forma podemos llamar a range con tres argumentos para indicar,
#además del inicio y el final del intervalo, el paso de iteración:

for par in range(0, 10, 2):
    print(par)

#for impar in range(1, 10, 2):
    #print(impar)
```

```
Zanahoria
Lechuga
Cebolla
Papa
El tamaño de la lista de verduras es: 4

Zanahoria
Lechuga
Cebolla
Papa
0
2
4
6
8
```

A este proceso de recorrer los elementos de un objeto se lo llama **iterar**.

Es importante recordar la indentación, como ocurría con el `if` ; siempre que haya un `:` la línea siguiente debe estar indentada.

Otra manera de usar la iteración `for` es la siguiente,

```
for elemento in verduras:
    print(elemento)
```

```
Zanahoria
Lechuga
Cebolla
Papa
```

Ejemplo de uso:

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
suma = 0

for i in lista:
    suma = suma + i
    #suma += i # Esta es otra manera de sumar más compacta
print(suma)
```

```
55
```

Método `append()` Si queremos agregar un elemento a la lista y que lo haga el codigo podemos usar la función append, que agrega lo que le pedimos adentro de una lista.

```
lista3 = ['Esponja', 'Jabon', 'Shampoo']

lista3.append('Acondicionador')
print(lista3)

y1 = []

for x in range(0,4):
    y1.append(2*x+8)

print(y1)
```

```
['Esponja', 'Jabon', 'Shampoo', 'Acondicionador']
[8, 10, 12, 14]
```

1.4.2. `while loops`

Un bucle `while` o bucle indefinido es un conjunto de instrucciones que se repiten siempre que la expresión lógica asociada sea verdadera. La siguiente es la sintaxis abstracta de un bloque de bucle `while`.

```
    while <expresión lógica>:
      #Bloque de código que se repetirá hasta que la declaración lógica sea falsa
          bloque de código
```

Cuando **Python** llega a un bloque de bucle `while`, primero determina si la expresión lógica del bucle `while` es verdadera o falsa. Si la expresión es verdadera, el bloque de código se ejecutará y, una vez ejecutado, el

programa volverá a la expresión lógica al comienzo de la declaración `while`. Si es falso, entonces el ciclo `while` terminará.

Ejemplo: Determinar el número de veces que 8 se puede dividir entre 2 hasta que el resultado sea menor que 1.

```
i = 0
n = 8

while n >= 1:
    n /= 2
    i += 1

print(f'n = {n}, i = {i}')
```

Búcle infinito:

```
n = 0
while n > -1:
    n += 1
```

Dado que n siempre será mayor que 1 sin importar cuántas veces se ejecute el ciclo, el código anterior nunca terminará.

1.4.3. Comprensiones de listas

En Python, hay otras formas de realizar iteraciones; la comprensión de listas (diccionario, conjunto) es una forma importante y popular que se utilizará con frecuencia en nuestro trabajo. Las comprensiones permiten crear secuencias a partir de otras secuencias con una sintaxis muy compacta. Primero comencemos a ver la comprensión de la lista.

```
[Salida secuencia_de_entrada Condiciones]
```

Ejemplo: Si x = range(5), eleva al cuadrado cada número en x y guárdalo en una lista y.

Si no utilizamos la comprensión de listas, el código será algo como esto:

```
x = range(5)
y = []

for i in x:
    y.append(i**2)
print(y)
```

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

Con compresión de listas, nos queda:

```
x = range(5)

y = [i**2 for i in x]
print(y)
```

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

Si queremos los elementos pares:

```
y = [i**2 for i in x if i%2 == 0]
print(y)
```

```
[0, 4, 16]
```

1.5. Funciones

Algunas funciones, como `print()`, `range()`, `type()`, fueron presentadas en secciones anteriores.

Supongamos ahora que querems una función más general. Por ejemplo calcular el promedio entre números. Para esto utilizamos las palabra `def` para definir una nueva función y `return` para indicar que devuelva un cierto valor.

La estructura en general para una función en Python es la siguiente:

```
def nombre_de_la_funcion(par_1, par_2, par_3, ...):
  |bloque de código
  |return algun_valor
donde par_1, par_2, ... son los parámetros de la función.
```

Ejemplo: (Promedio de dos números)

```
def promediar_dos_numeros(num1, num2):
    suma_de_numeros = num1 + num2
    res = suma_de_numeros / 2
    return res
```

Ejemplo de uso de la funcion promediar:

```
a = 2
b = 6

promedio = promediar_dos_numeros(a, b)
print(f"El promedio de {a} y {b} es {promedio}")
```

```
El promedio de 2 y 6 es 4.0
```

1.5.1. Variables locales y globales

Una función en **Python** tiene su propio bloque de memoria reservado para las variables creadas dentro de esa función. Este bloque de memoria no se comparte con todo el bloque de memoria general, que almacena las variables que fueron creadas antes de definir la nueva función. Por lo tanto, una variable con un nombre determinado se puede asignar dentro de una función sin cambiar una variable con el mismo nombre fuera de la función. El bloque de memoria asociado a la función se abre cada vez que se utiliza una función.

Ejemplo:

```
def suma(a, b, c):
    out = a + b + c
    print(f'El valor de out dentro de la función suma es {out}')
    return out

out = 1
d = suma(1, 2, 3)
print(f'El valor de out fuera de la función suma es {out}')
```

```
El valor de out dentro de la función suma es 6
El valor de out fuera de la función suma es 1
```

1.5.2. Funciones Lambda

A veces, no queremos utilizar la forma definida antes para definir una función, especialmente si nuestra función es de solo una línea. En este caso, podemos usar una función anónima en **Python**, que es una función que se define sin nombre. Este tipo de funciones también se denomina función `labmda`, ya que se definen mediante la palabra clave `labmda`. Luego, una función lambda típica es de la forma:

```
    lambda arguments: expression
```

Ejemplos:

```
cuadrado = lambda x: x**2

print(cuadrado(2))
print(cuadrado(5))

suma1 = lambda x, y: x + y

print(suma1(2, 4))

f_1 = lambda x,y: x**2 + y**2

print(f_1(0,0))
```

```
25
6
0
```

1.5.3. Bibliotecas

Observemos que en lo anterior se utilizó en Python varias funciones intrínsecas, o "de fábrica", como help(), print() así también como operaciones básicas entre números como sumar, restar, etc; Además, vimos que podemos crear nustras propias funciones para que hagan lo que necesitemos usando el comando def nombre_funcion.

Supongamos que uno quiere calcular `cos(x)`. Python no viene por defecto con esa operación, uno debería crear un algoritmo (es decir, una serie de acciones) que calcule el valor del cos de x (cosa que puede ser no trivial). Vamos entonces a decirle a Python que, además de sus operaciones de fábrica, queremos ampliar nuestro abanico de operaciones matemáticas en todas las opciones que aparecen dentro de la biblioteca `"math"` (una biblioteca es, tal como sugiere el nombre, un archivo con un montón de funciones, objetos y demás).

```
import math  # Importo a mi programa todo lo que este contenido dentro del archivo "ma
```

Aclaración: No todas las bibliotecas vienen instaladas por defecto en Python. Las que vamos a utilizar en nuestro curso ya vienen instaladas por Google en Colab o fueron instaladas automaticamente por Anaconda (incluye Spyder) en su computadora. Si solo utilizamos Spyder sin el entorno de Anaconda, cuando queramos utilizar una biblioteca o librería nos solicitará que la instalemos.

Ahora que importamos en Python la biblioteca, podemos acceder a su contenido usando la sintaxis math.lo_que_haya_dentro_de_math por ejemplo, `math.cos()`

```
math.cos(0)
```

```
1.0
```

Ejemplo funciones `lambda`:

```
f_2 = lambda x: math.cos(x)

print(f_2(math.pi))
print(f_2(0))

f_3 = lambda x,y: math.exp(x**2 + y**2)

print(f_3(0,0))
print(f_3(1,0))
```

```
-1.0
1.0
1.0
2.718281828459045
```

1.5.4. Crear un script que accede a otros scripts

Para crear scripts que dentro de su código hagan uso de otros scripts ya realizados, o que contengan otra parte del código (útil para ordenar códigos largos), debemos utilizar el comando `import`

Supongamos que tenemos un script en el cual creamos una función suma, y lo guardamos como `operaciones.py`. Luego el contendio de dicha función es:

```
def suma(num1, num2):
    suma_de_numeros = num1 + num2
    return suma_de_numeros
```

Y luego, queremos crear en otro script una funcion que calcule el promedio de dos numeros, utilizando la función suma. Entonces, nos queda,

```
import operaciones

def prom(num1, num2):
    promedio = operaciones.suma(num1,num2)
    return promedio
```

Aclaración: Para hacer uso del comando `import` los scripts que van a ser utilizados deberán estar guardados en una misma carpeta de archivos.

Como alternativa a lo anterior, tambien es valido utiliar `from NombreArchivo import Nombrefunción`, por ejemplo,

```
from operaciones import suma

def prom(num1, num2):
    promedio = suma(num1,num2)
    return promedio
```

1.6. NumPy : vectores, matrices y tablas de datos

NumPy (NUMeric PYthon) es la biblioteca para operar sobre vectores de números. Además de contener un nuevo tipo de dato que nos va a ser muy útil para representar vectores y matrices, nos provee de un arsenal de funciones de todo tipo.

Vamos a empezar por importar la bliblioteca `numpy`. La sintaxis para lo anterior es `import biblio as nombre`:

```python
import numpy as np  # Acceso a las funciones de numpy a través de np.función()

# en ejemplo
print(f"El numero e = {np.e}")
print(f"El numero Pi = {np.pi}")

# Podemos calcular senos y cosenos de numeros igual que con math!
print(np.sin(np.pi))  # casi cero! tipo de dato float!
```

```
El numero e = 2.718281828459045
El numero Pi = 3.141592653589793
1.2246467991473532e-16
```

1.6.1. Arrays

Los arrays numéricos nos van a servir para representar vectores (los arrays no son vectores, aunque pueden comportarse como tales en muchos sentidos).

La idea es que es parecido a una lista: son muchos números juntos en la misma variable y están indexados (los puedo llamar de a uno dando la posición dentro de la variable). La gran diferencia con las listas de Python es que los arrays de numpy operan con las siguientes operaciones tradicionales:

- Si sumamos o restamos dos arrays, se suman componente a componente.
- Si multiplicamos o dividimos dos arrays, se multiplican o dividen componente a componente.

Veamos ejemplos usando la función `np.array` para crear arrays básicos.

```python
array_1 = np.array([1, 2, 3, 4])  # array toma como argumento un vector_like (lista, tu
array_2 = np.array([5, 6, 7, 8])

print("ARRAYS:")
print(f"{array_1} + {array_2} =  {array_1 + array_2}")  # aca los sumo
print(f"{array_1} * {array_2} = {array_1*array_2}")  # acá los multiplico

#Y al igual que con las listas, uno puede acceder a elementos específicos de un array

print(array_1[0], array_1[1], array_1[2], array_1[3])  # son 4 elementos, los indices v

# Además, podemos usar comandos similiares a los de listas de Python
print(array_2[-1])  # último elemento de array_2
print(array_2[0:3])  # desde el primero hasta el 3
```

```
ARRAYS:
[1 2 3 4] + [5 6 7 8] =  [ 6  8 10 12]
[1 2 3 4] * [5 6 7 8] = [ 5 12 21 32]
1 2 3 4
8
```

```
[5 6 7 8]
```

```
print("array_1:")
for num in array_1:  # Iteramos sobre los ELEMENTOS de array_1
    print(num)

print('\n')  # Esto es solo para que agregue una linea nueva y no queden pegados

print("array_2:")
for j in range(len(array_2)):  # Iteramos sobre los INDICES de array_2
    print(array_2[j])
```

```
array_1:
1
2
3
4
Hello
World!


array_2:
5
6
7
8
```

1.6.2. Matrices

Los arrays de dos dimensiones son utiles para representar matrices. Para crear arrays de dos dimensiones (o más), podemos aplicar la función `np.array` sobre una lista de listas:

```
c = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  # cada lista corresponde a

#una fila de la matriz
print(c)

# Aca hacer c[0] dara la primer FILA de la matrix.
# Si queremos sacar la primer componente habra que hacer c[0, 0]
```

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

```
identidad = np.identity(3)
print(f'Matriz identidad 3x3:\n{identidad}')

# Otras
ceros = np.zeros((4, 4))  # Todos ceros, matriz de 4x4
unos = np.ones((2,3))  # Todos unos, matriz de 2x3
I = np.eye(5, k=0) #Matríz identidad de tamaño 5
I1 = np.eye(5, k=1) # Unos en diagonal, como una identidad corrida, útil para cálculo

print()
print(f'Matriz de ceros:\n{ceros}')
print()
print(f'Matriz de unos:\n{unos}')
print()
print(f'Identidad corrida:\n{I}')
print()
print(f'Identidad corrida:\n{I1}')
```

```
Matriz identidad 3x3:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

Matriz de ceros:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

Matriz de unos:
[[1. 1. 1.]
 [1. 1. 1.]]

Identidad corrida:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]

Identidad corrida:
[[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]]
```

1.6.3. Particiones de intervalos

Un funcion de `numpy` muy útil, para crear un array entre dos valores fijos, es `linspace`. La función `np.linspace` genera un array NumPy formado por n números equiespaciados entre dos dados. Su sintaxis es:

```
np.linspace(valor-inicial, valor-final, tamaño de partición)
```

Otra función que nos permite crear un array NumPy es `np.arange`. Al igual que la función predefinida de Python range, genera un conjunto de números entre un valor de inicio y uno final, pudiendo especificar un incremento entre los valores, pero, al contrario de lo que ocurre con range, el resultado aquí es un array NumPy. Su sintaxis es:

```
np.arange(valor-inicial, valor-final, tamaño de paso)
```

Ejemplos:

```
# Equiespaciados - Elijo CANTIDAD DE PUNTOS en un dado intervalo [a, b]
equi_linspace = np.linspace(0, 1, 11)  # 11 números equiespaciados linealmente
#entre 0 y 1 (LINear SPACE = linspace)
print(f'Numero de puntos fijo: {equi_linspace}')  # N puntos en el rango [0, 1]
#con 1/(N-1) de espaciamiento

print('\n')

# Equiespaciados - Elijo EL PASO entre los puntos en un dado intervalo [a, b)
equi_arange = np.arange(0, 1.1, 1./10)  # como el range de Python pero
#puede venir con un paso en coma flotante
print(f'Paso entre puntos fijo: {equi_arange}')  # {0, paso, 2*paso, ... n*paso}
#siempre que n*paso<1
```

```
Numero de puntos fijo: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]


Paso entre puntos fijo: [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

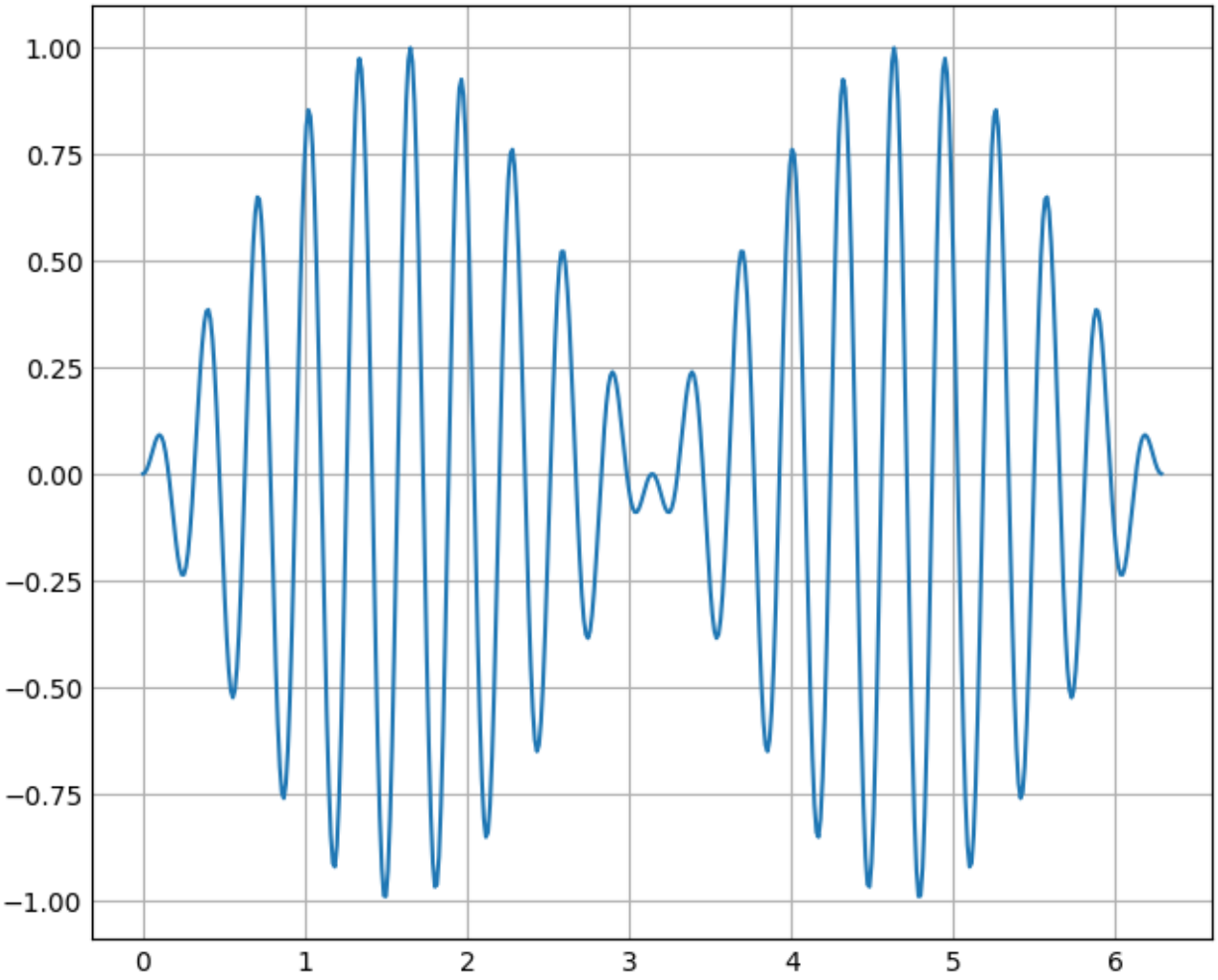
1.7. Gráfica de funciones (Ploteos)

La biblioteca `matplotlib` nos permitirá hacer ploteos. Esta librería es muy grande, por lo que tiene divididas sus funciones en distintos módulos. Utilizaremos la sublibrería `pyplot`. Una de las formas de importar esta librería es:

```
import matplotlib.pyplot as plt
```

Podemos usar esta librería en conjunto con `numpy` para graficar las funciones que querramos. Veamos un ejemplo, grafiquemos la función $f(x) = sin(x)sin(20x)$ entre 0 y 2π.

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

# Definimos la función
def f(x):
  return np.sin(x)*np.sin(20*x)

# Definimos el dominio y la imagen
x = np.linspace(0, 2*np.pi, 500) #array de 0 a 2pi con tamaño de paso (2pi-0)/499 y 500
y = f(x) #Aqui por defecto la función f se aplica a cada elemento del array x
         #por lo que y será un array de la misma dimensión que x

# Graficamos la función
plt.plot(x, y)
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(6, 5) #tamaño de gráfica (visor)
plt.show()
```



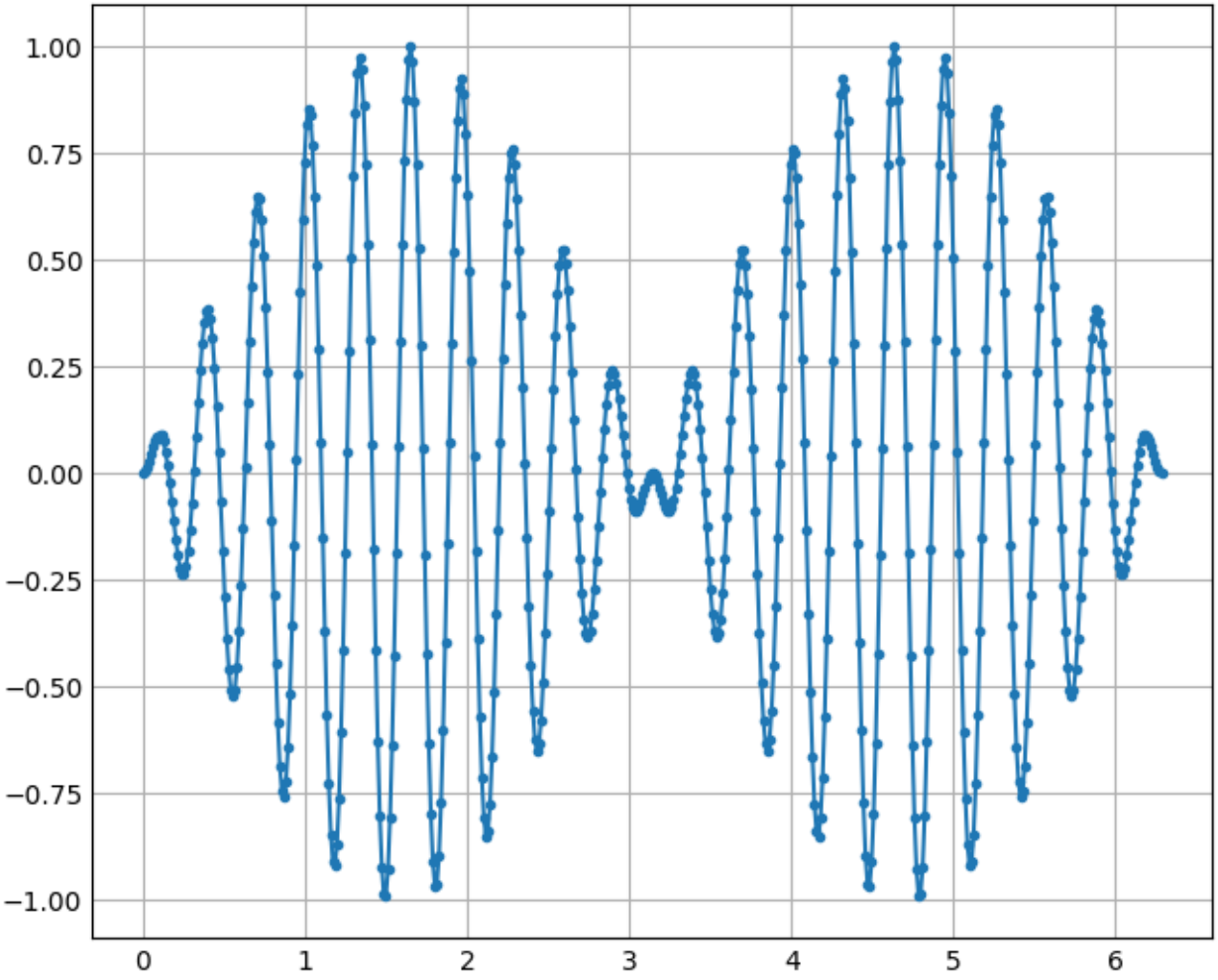
Información sobre el código que acabamos de escribir:

Primero escribimos la función que queremos graficar. Luego, definimos nuestro dominio `x` y nuestra imagen `y`. Y por último usamos la función de `plot` de `matplotlib.pyplot` para graficar y en función de `x`.

Observación 1: `plt.show()`: Si bien esta linea no es siempre necesaria en los notebooks (como google Colab), su propósito es çerrar"la figura y mostrarla.

Observación 2: y no es una función. Sino que es un array que contiene los números que resultan de aplicarle `f` a cada uno de los valores de `x`. Lo que vemos arriba parece ser una función continua pero en realidad es un conjunto de punto unidos con una poligonal. Para visualizar mejor esto podemos hacer un gráfico que muestre explícitamente los puntos que forman el gráfico:

```
plt.plot(x, y, '.-')
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(6, 5) #tamaño de gráfica (visor)
plt.show()
```

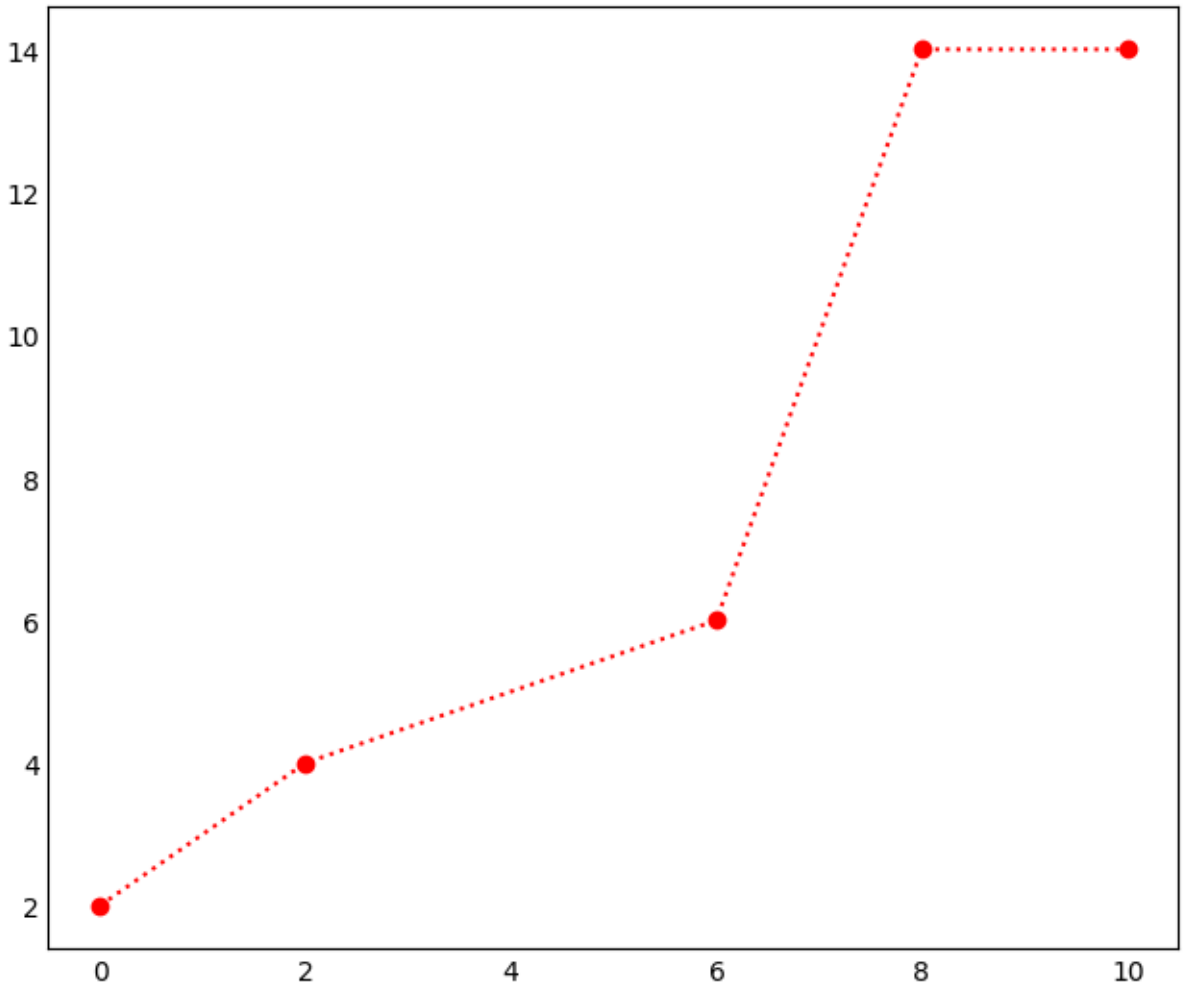


Función `plt.plot()`

Esta función toma los vectores que le damos como inputs (x e y) y grafica un punto en cada par de coordenadas (x,y). Además, por defecto une estos puntos con lineas rectas. Por lo tanto, para que la función funcione correctamente, es necesario que los array x e y tengan la misma longitud. Para dejar claro este

concepto veamos un ejemplo con pocos puntos:

```
coords_en_x = np.array([0, 2, 6, 8,  10])
coords_en_y = np.array([2, 4, 6, 14, 14])

plt.plot(coords_en_x, coords_en_y, 'o:r')
plt.grid()  # Esto me deja poner una grilla detrás!
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(6, 5) #tamaño de gráfica (visor)
plt.show()
```



1.7.1. Formato y estética en gráficos

En los gráficos anteriores cuando usamos plt.plot agregamos un término opcional para cambiar el color y la forma en la que se muestran los puntos que graficamos. Si usamos `help(plt.plot)` vemos que este argumento opcional es el `[fmt]` y que toma distintos strings con los que se puede cambiar la forma en que se ve el gráfico. La estructura general es

```
fmt = '[color][marker][line]'
```

donde marker hace referencia a la forma de los puntos y line hace referencia a la forma de la linea.

Los strings válidos son un montón y los pueden ver en la documentación usando el help o en la documentación online de la función.

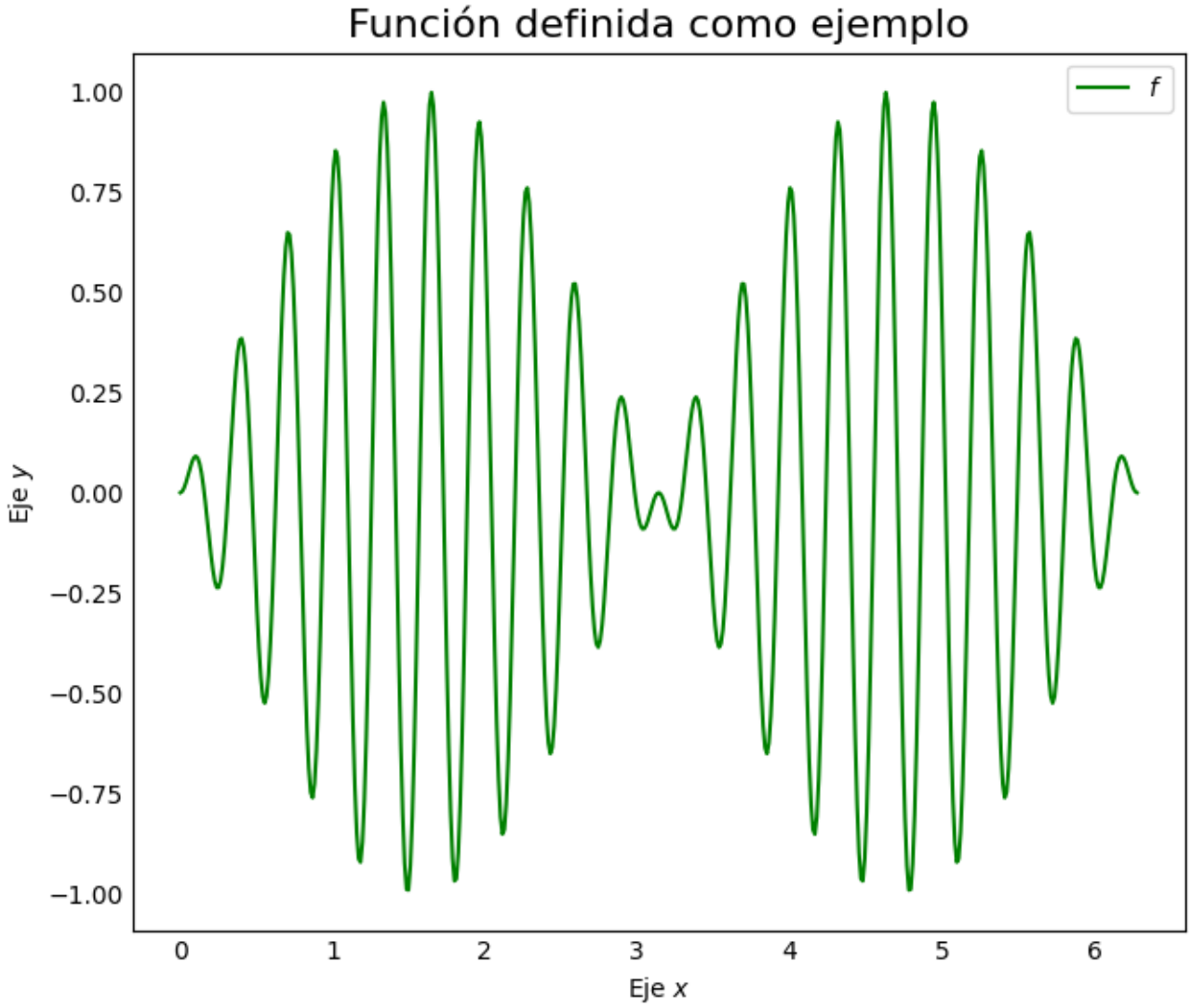
Algunos datos útiles:

- **color:** 'r' (rojo), 'b' (azul), 'g' (verde), 'm' (magenta), etc
- **marker:** '.' (puntito chiquito), 'o' (punto grande), '*' (estrella), etc
- **line:** '-' (sólida), '--' (de a rayas), ':' (de a puntos)

1.7.2. Título, labels y grid

Veamos también algunas funciones que nos permiten agregar información a la figura sobre la que estamos trabajando. Veamos un ejemplo con la función anterior.

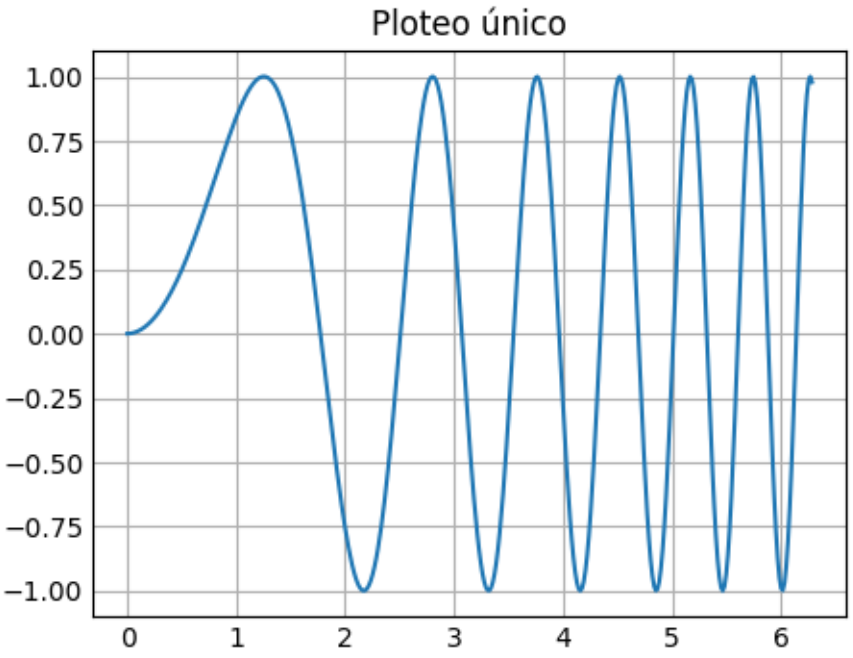
```
import numpy as np
import matplotlib.pyplot as plt

# Definimos la función
def f(x): return np.sin(x)*np.sin(20*x)

# Definimos el dominio y la imagen
x = np.linspace(0, 2*np.pi, 500)
y = f(x)

# Graficamos la función con linea sólida y le ponemos una etiqueta
plt.plot(x, y, 'g-', label = '$f$') #El simbolo monetario se utiliza para poder usar

#código latex en las etiquetas

# Agregamos título y etiquetamos los ejes
plt.title('Función definida como ejemplo', fontsize=16)
plt.xlabel('Eje $x$')
plt.ylabel('Eje $y$')

plt.grid()
plt.legend(loc = "upper right")
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(6, 5) #tamaño de gráfica (visor)
plt.show()
```


Función definida como ejemplo
f
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Eje y
0
1
2
3
4
5
6
Eje x

Varias gráficas en una misma imagen (`subplot`)

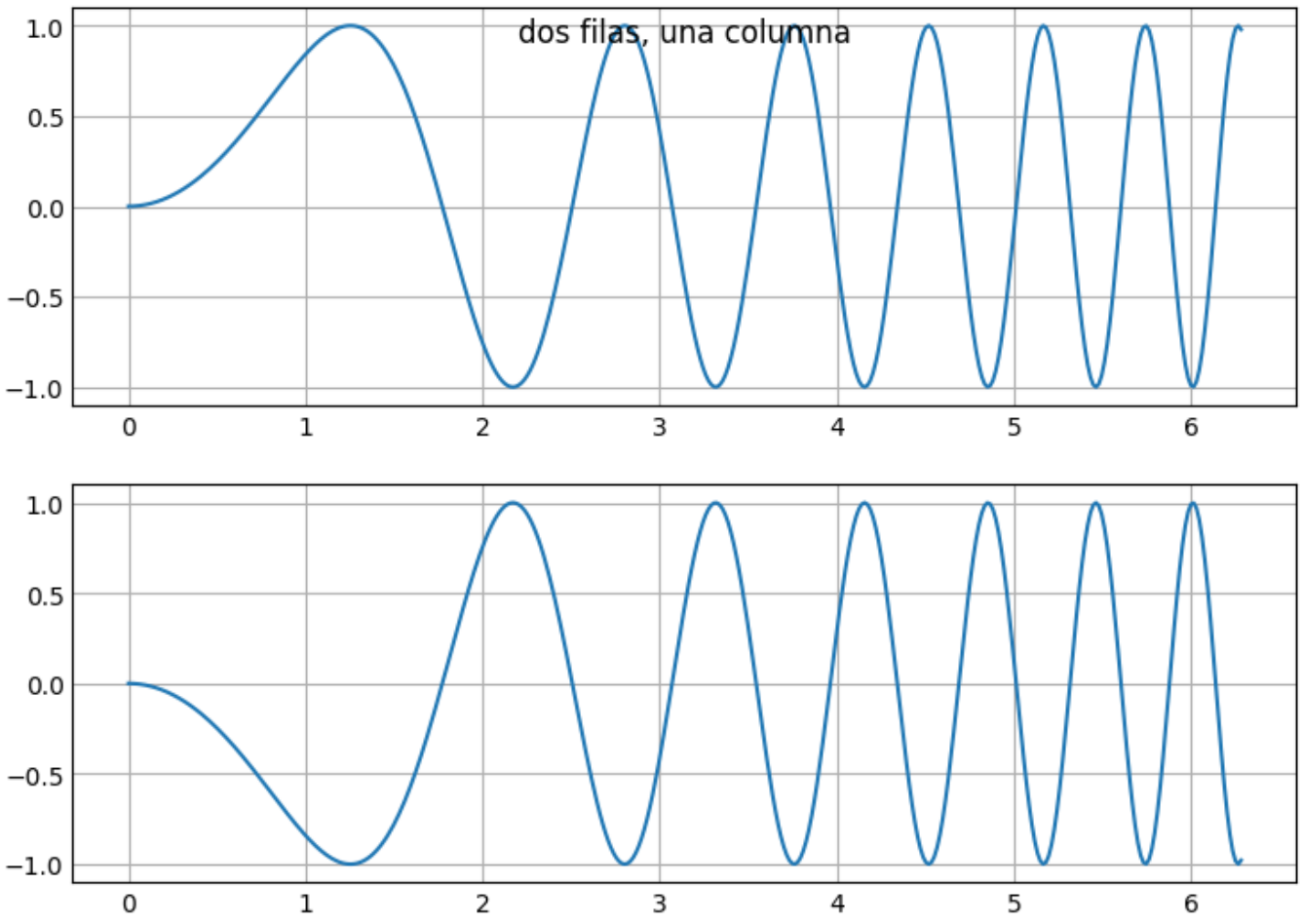
La función `matplotlib.pyplot.subplots` crea una figura y uno (o varios) conjunto de ejes, devolviendo una referencia a la figura y a los ejes. Por defecto -si no se especifica otra cosa- crea un único conjunto de ejes:

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

# Ejemplo 1
xq = np.linspace(0, 2 * np.pi, 400)
yq = np.sin(xq ** 2)
fig, ax = plt.subplots()
ax.plot(xq, yq)
ax.set_title('Ploteo único')
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(4, 3) #tamaño de gráfica (visor)
```

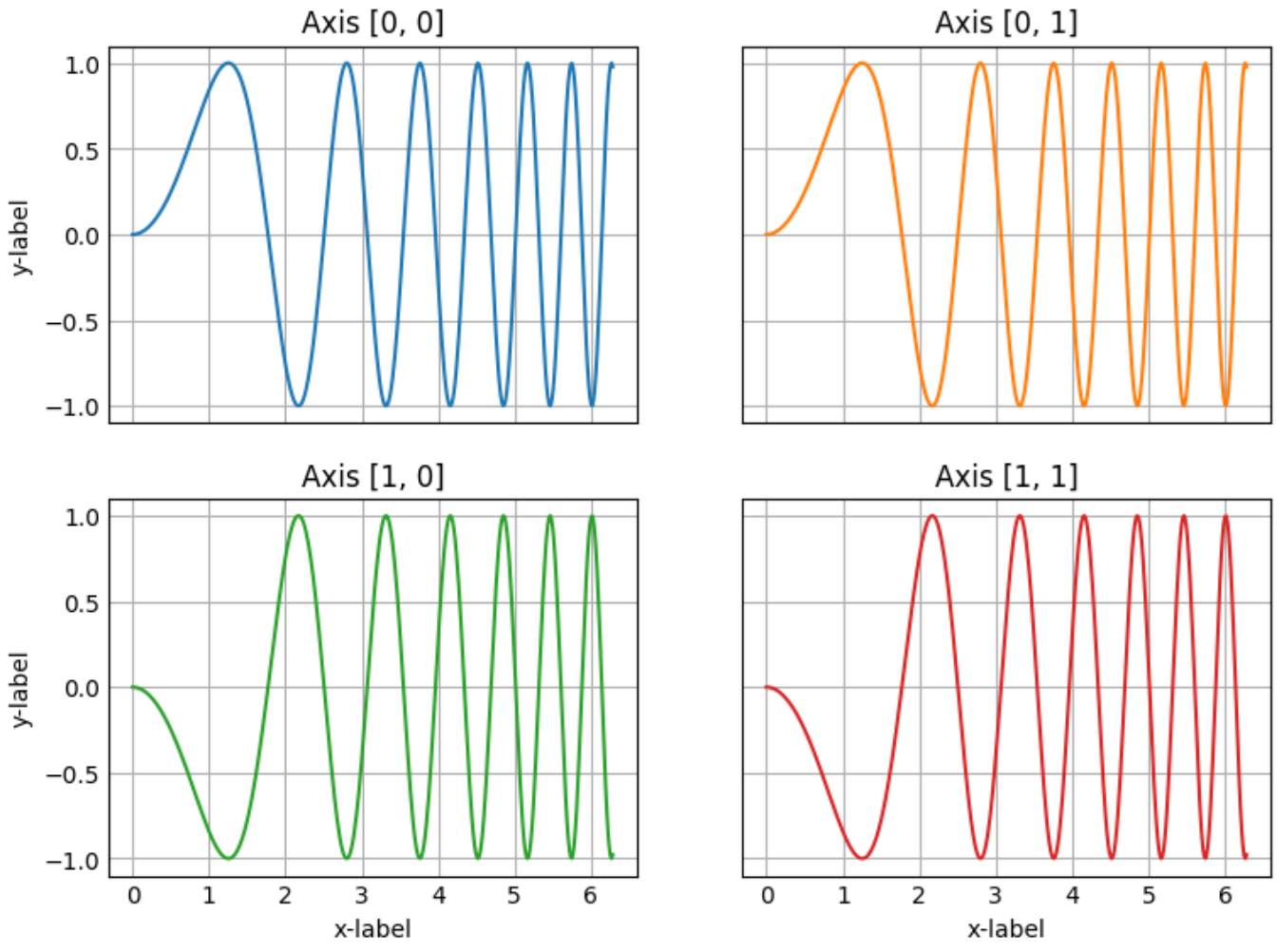


Si queremos crear una matriz de conjuntos de ejes de, por ejemplo, m filas y n columnas (es decir, m.n conjuntos de ejes repartidos de dicha forma), basta añadir estos valores como primeros argumentos de la función, esto es:

```
fig, axs = plt.subplots(2,1) #dos filas, una columna
fig.suptitle('dos filas, una columna')
axs[0].plot(xq, yq)
axs[1].plot(xq, -yq)
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(7, 5) #tamaño de gráfica (visor)
```



```
fig, axs = plt.subplots(2, 2) #2 filas, 2 columnas
axs[0, 0].plot(xq, yq)
axs[0, 0].set_title('Axis [0, 0]')
axs[0, 1].plot(xq, yq, 'tab:orange')
axs[0, 1].set_title('Axis [0, 1]')
axs[1, 0].plot(xq, -yq, 'tab:green')
axs[1, 0].set_title('Axis [1, 0]')
axs[1, 1].plot(xq, -yq, 'tab:red')
axs[1, 1].set_title('Axis [1, 1]')
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(7, 5) #tamaño de gráfica (visor)

for ax in axs.flat:
    ax.set(xlabel='x-label', ylabel='y-label')

# Marcar eje x y eje y solo en los bordes
for ax in axs.flat:
    ax.label_outer()
```



1.7.3. Anatomía de una figura con `matplotlib`

El siguiente código se obtuvo del siguiente enlace.

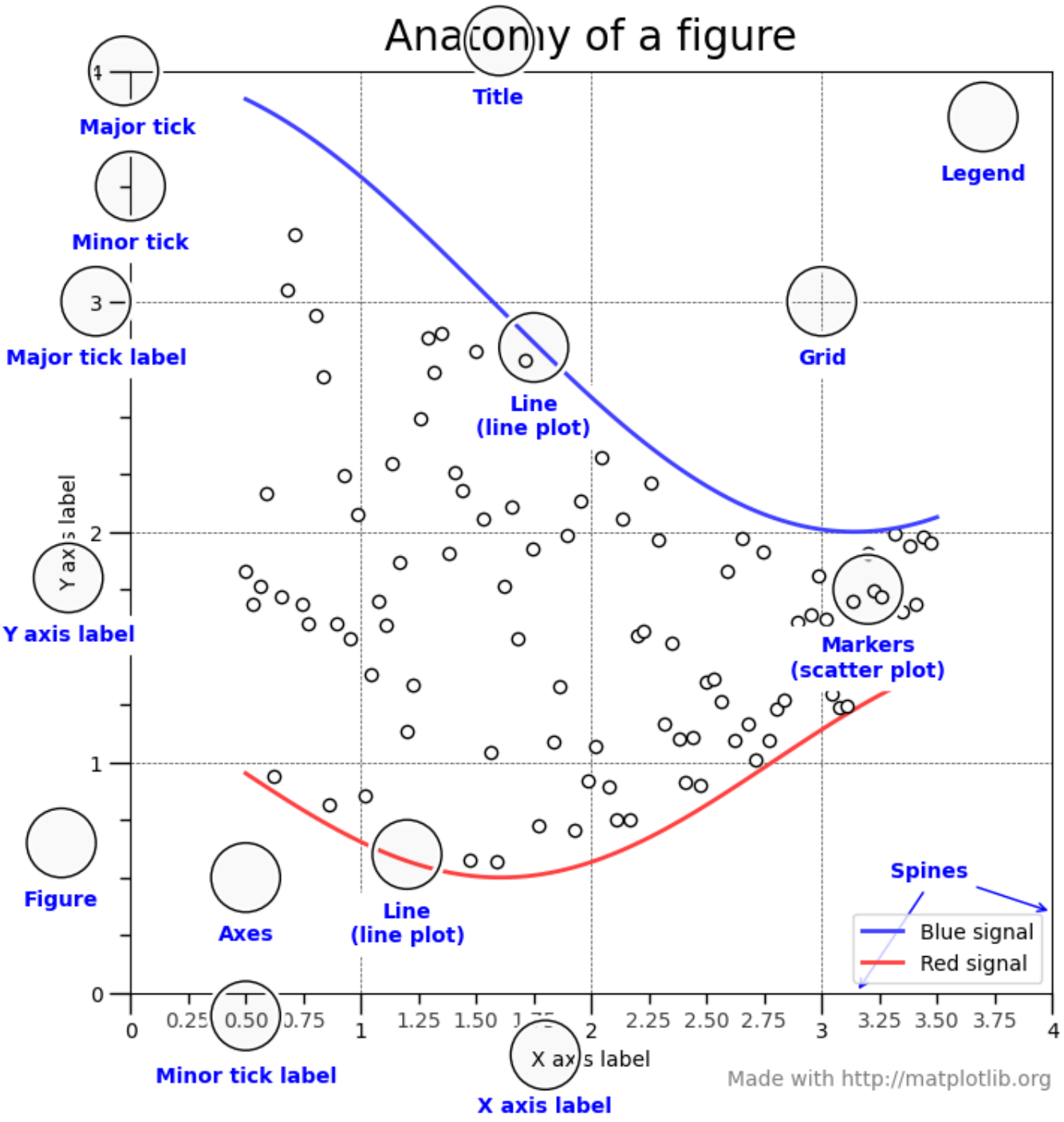
En el mismo sitio web se encuentran muchos ejemplos de uso de la librería `matplotlib`.

```
# This figure shows the name of several matplotlib elements composing a figure

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator, MultipleLocator, FuncFormatter


np.random.seed(19680801)

X = np.linspace(0.5, 3.5, 100)
Y1 = 3+np.cos(X)
Y2 = 1+np.cos(1+X/0.75)/2
Y3 = np.random.uniform(Y1, Y2, len(X))

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(1, 1, 1, aspect=1)


def minor_tick(x, pos):
    if not x % 1.0:
        return ""
    return "%.2f" % x

ax.xaxis.set_major_locator(MultipleLocator(1.000))
ax.xaxis.set_minor_locator(AutoMinorLocator(4))
ax.yaxis.set_major_locator(MultipleLocator(1.000))
ax.yaxis.set_minor_locator(AutoMinorLocator(4))
ax.xaxis.set_minor_formatter(FuncFormatter(minor_tick))

ax.set_xlim(0, 4)
ax.set_ylim(0, 4)

ax.tick_params(which='major', width=1.0)
ax.tick_params(which='major', length=10)
ax.tick_params(which='minor', width=1.0, labelsize=10)
ax.tick_params(which='minor', length=5, labelsize=10, labelcolor='0.25')

ax.grid(linestyle="--", linewidth=0.5, color='.25', zorder=-10)

ax.plot(X, Y1, c=(0.25, 0.25, 1.00), lw=2, label="Blue signal", zorder=10)
ax.plot(X, Y2, c=(1.00, 0.25, 0.25), lw=2, label="Red signal")
ax.plot(X, Y3, linewidth=0,
```

```
        marker='o', markerfacecolor='w', markeredgecolor='k')

ax.set_title("Anatomy of a figure", fontsize=20, verticalalignment='bottom')
ax.set_xlabel("X axis label")
ax.set_ylabel("Y axis label")

ax.legend()


def circle(x, y, radius=0.15):
    from matplotlib.patches import Circle
    from matplotlib.patheffects import withStroke
    circle = Circle((x, y), radius, clip_on=False, zorder=10, linewidth=1,
                    edgecolor='black', facecolor=(0, 0, 0, .0125),
                    path_effects=[withStroke(linewidth=5, foreground='w')])
    ax.add_artist(circle)


def text(x, y, text):
    ax.text(x, y, text, backgroundcolor="white",
            ha='center', va='top', weight='bold', color='blue')


# Minor tick
circle(0.50, -0.10)
text(0.50, -0.32, "Minor tick label")

# Major tick
circle(-0.03, 4.00)
text(0.03, 3.80, "Major tick")

# Minor tick
circle(0.00, 3.50)
text(0.00, 3.30, "Minor tick")

# Major tick label
circle(-0.15, 3.00)
text(-0.15, 2.80, "Major tick label")

# X Label
circle(1.80, -0.27)
text(1.80, -0.45, "X axis label")

# Y Label
circle(-0.27, 1.80)
text(-0.27, 1.6, "Y axis label")
```

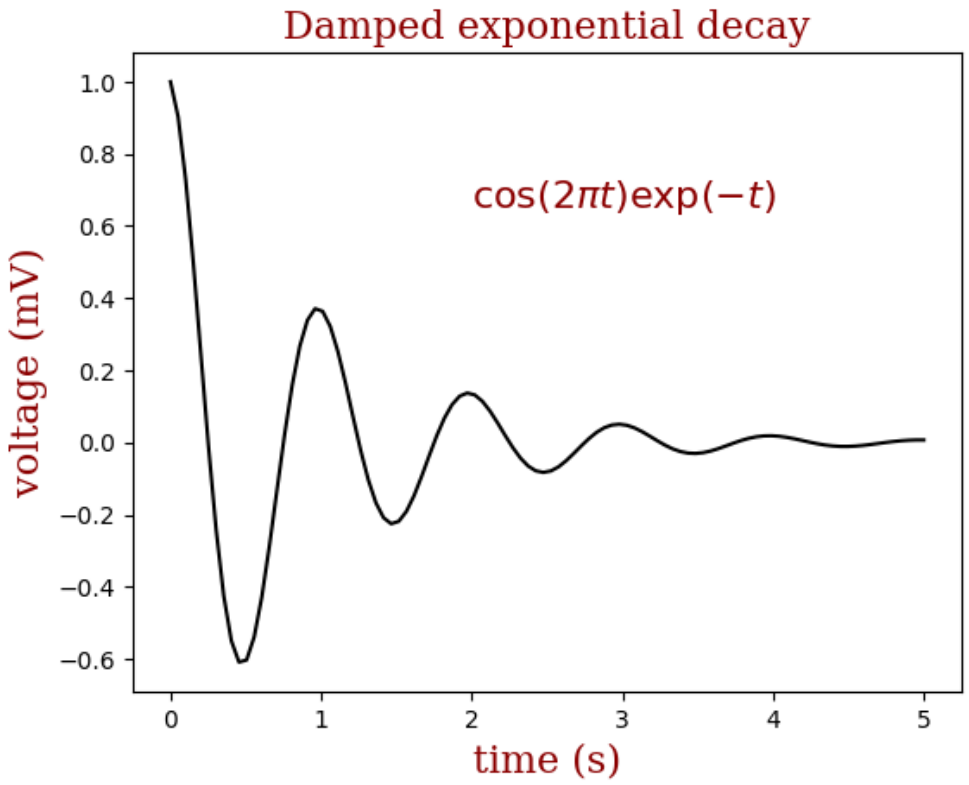
```
# Title
circle(1.60, 4.13)
text(1.60, 3.93, "Title")

# Blue plot
circle(1.75, 2.80)
text(1.75, 2.60, "Line\n(line plot)")

# Red plot
circle(1.20, 0.60)
text(1.20, 0.40, "Line\n(line plot)")

# Scatter plot
circle(3.20, 1.75)
text(3.20, 1.55, "Markers\n(scatter plot)")

# Grid
circle(3.00, 3.00)
text(3.00, 2.80, "Grid")

# Legend
circle(3.70, 3.80)
text(3.70, 3.60, "Legend")

# Axes
circle(0.5, 0.5)
text(0.5, 0.3, "Axes")

# Figure
circle(-0.3, 0.65)
text(-0.3, 0.45, "Figure")

color = 'blue'
ax.annotate('Spines', xy=(4.0, 0.35), xycoords='data',
            xytext=(3.3, 0.5), textcoords='data',
            weight='bold', color=color,
            arrowprops=dict(arrowstyle='->',
                            connectionstyle="arc3",
                            color=color))

ax.annotate('', xy=(3.15, 0.0), xycoords='data',
            xytext=(3.45, 0.45), textcoords='data',
            weight='bold', color=color,
            arrowprops=dict(arrowstyle='->',
                            connectionstyle="arc3",
```

```
                                color=color))

ax.text(4.0, -0.4, "Made with http://matplotlib.org",
        fontsize=10, ha="right", color='.5')

plt.show()
```



Ejemplo:

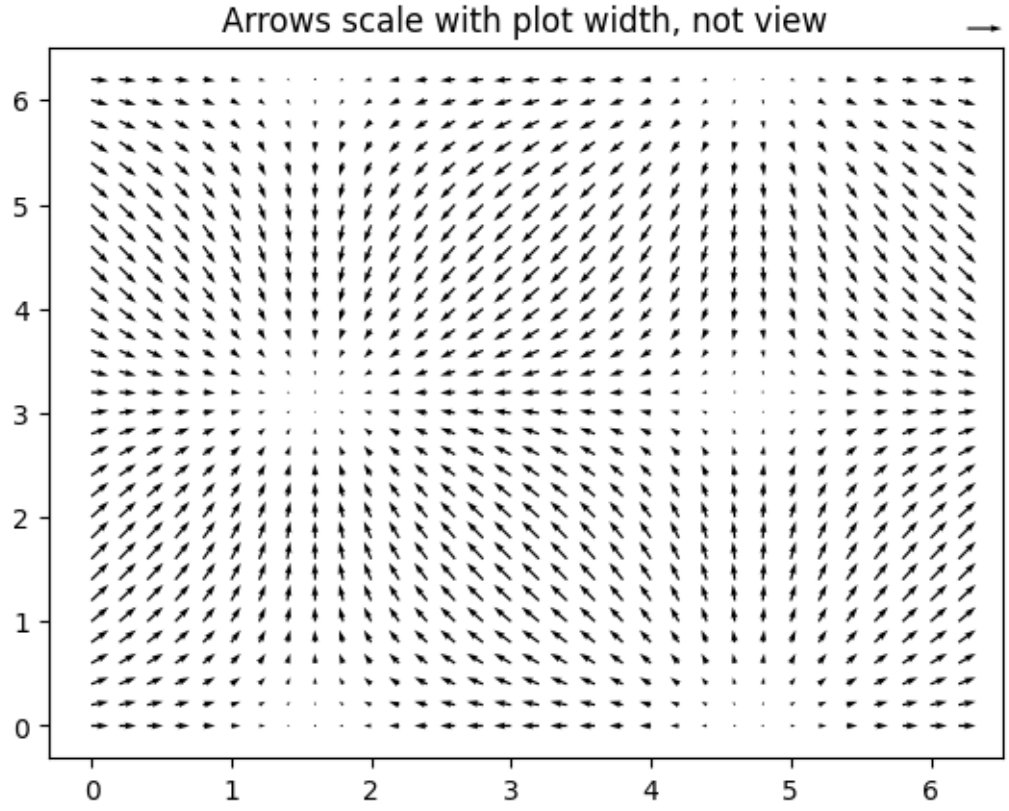
```
import numpy as np
import matplotlib.pyplot as plt


font = {'family': 'serif',
        'color':  'darkred',
        'weight': 'normal',
        'size': 16,
        }

x = np.linspace(0.0, 5.0, 100)
y = np.cos(2*np.pi*x) * np.exp(-x)

plt.plot(x, y, 'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$\cos(2 \pi t) \exp(-t)$', fontdict=font)
plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

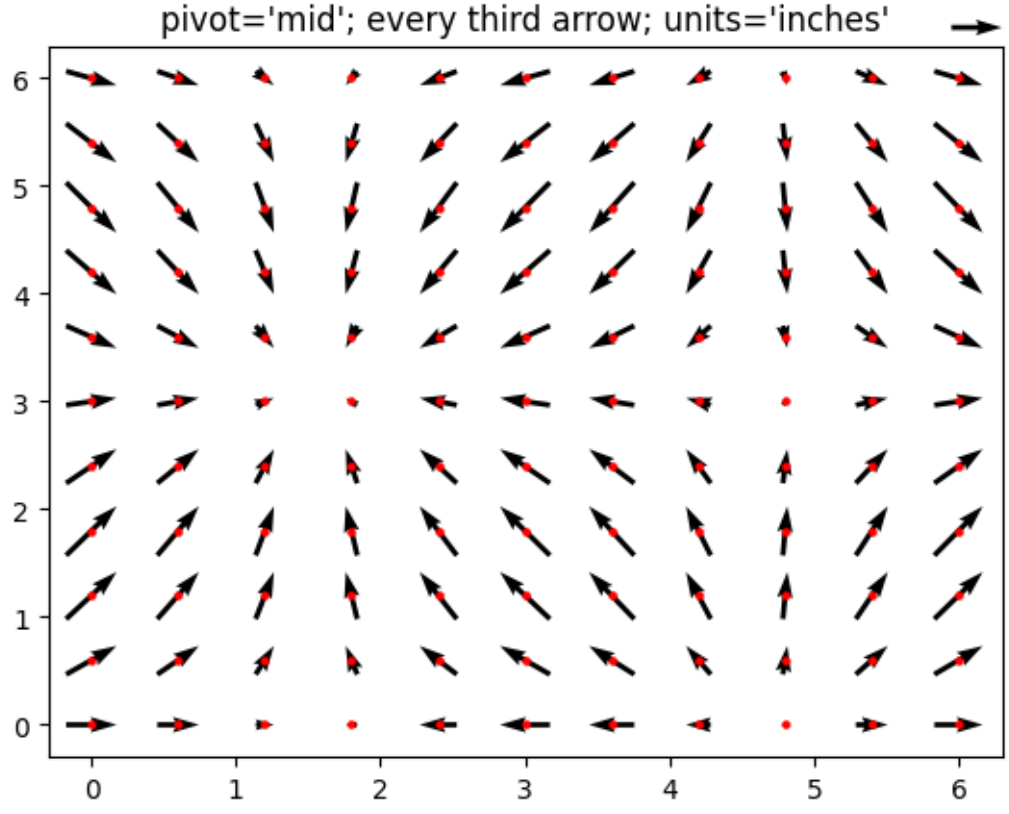


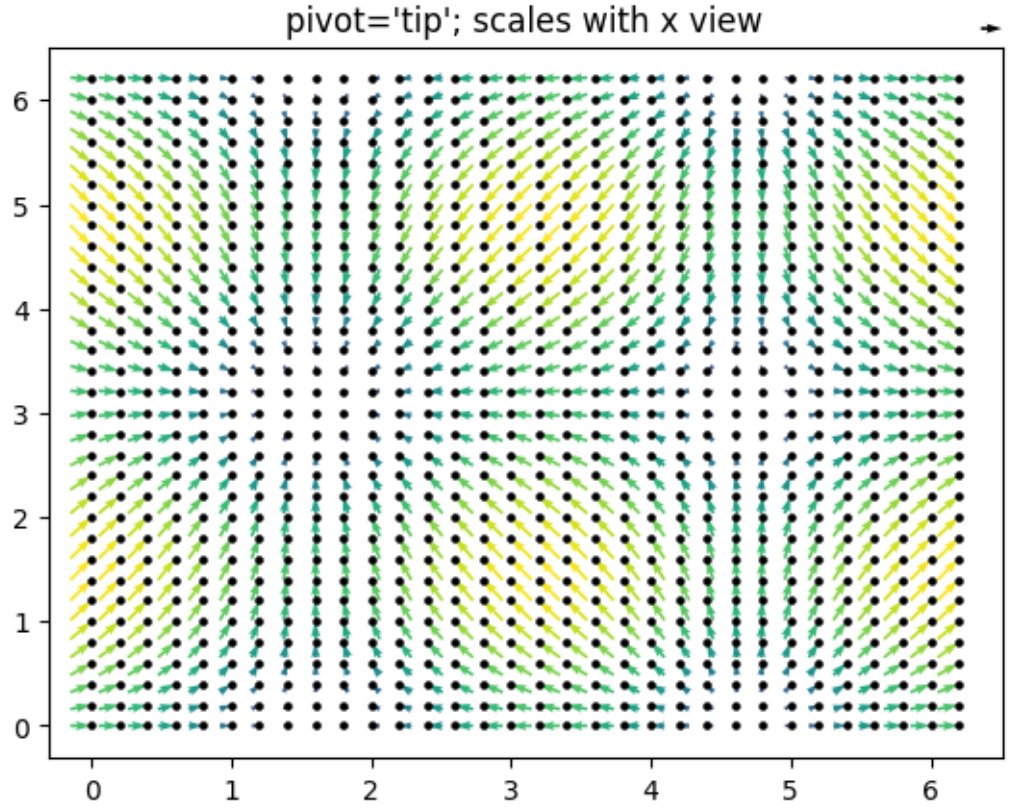
Otro Ejemplo:

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

X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))

U = np.cos(X)

V = np.sin(Y)

plt.figure()
plt.title('Arrows scale with plot width, not view')

Q = plt.quiver(X, Y, U, V, units='width')

qk = plt.quiverkey(Q, 0.9, 0.9, 2, r'2 \frac{m}{s}', labelpos='E',
                   coordinates='figure')
```

```
plt.figure()

plt.title("pivot='mid'; every third arrow; units='inches'")

Q = plt.quiver(X[::3, ::3], Y[::3, ::3], U[::3, ::3], V[::3, ::3],

qk = plt.quiverkey(Q, 0.9, 0.9, 1, r'1 \fracms', labelpos='E',
                   coordinates='figure')

plt.scatter(X[::3, ::3], Y[::3, ::3], color='r', s=5)

plt.figure()

plt.title("pivot='tip'; scales with x view")

M = np.hypot(U, V)


Q = plt.quiver(X, Y, U, V, M, units='x', pivot='tip', width=0.022,
               scale=1 / 0.15)


qk = plt.quiverkey(Q, 0.9, 0.9, 1, r'1 \fracms', labelpos='E',
                   coordinates='figure')

plt.scatter(X, Y, color='k', s=5)
plt.show()
```


Arrows scale with plot width, not view
0
1
2
3
4
5
6


pivot='mid'; every third arrow; units='inches'
0
1
2
3
4
5
6



#Ploteo de Funciones en varias variables Una manera simple de generar gráifcos 3D es haciendo uso de mplot3d toolkit. Para detalles sobre su uso, puede visitarse el sitio:

Página oficial para más información

Abajo, dejamos dos ejemplos para ver el potencial uso de esta herramienta:

1.7.4. La función meshgrid

La función `numpy.meshgrid` devuelve una lista de matrices de coordenadas a partir de vectores de coordenadas.

Veamos un ejemplo sencillo:

```
x0 = np.linspace(0, 3, 4)
y0 = np.linspace(0, 3, 4)

X0, Y0 = np.meshgrid(x0, y0)

print(X0,Y0)
```

```
[[0. 1. 2. 3.]
 [0. 1. 2. 3.]
 [0. 1. 2. 3.]
 [0. 1. 2. 3.]] [[0. 0. 0. 0.]
```

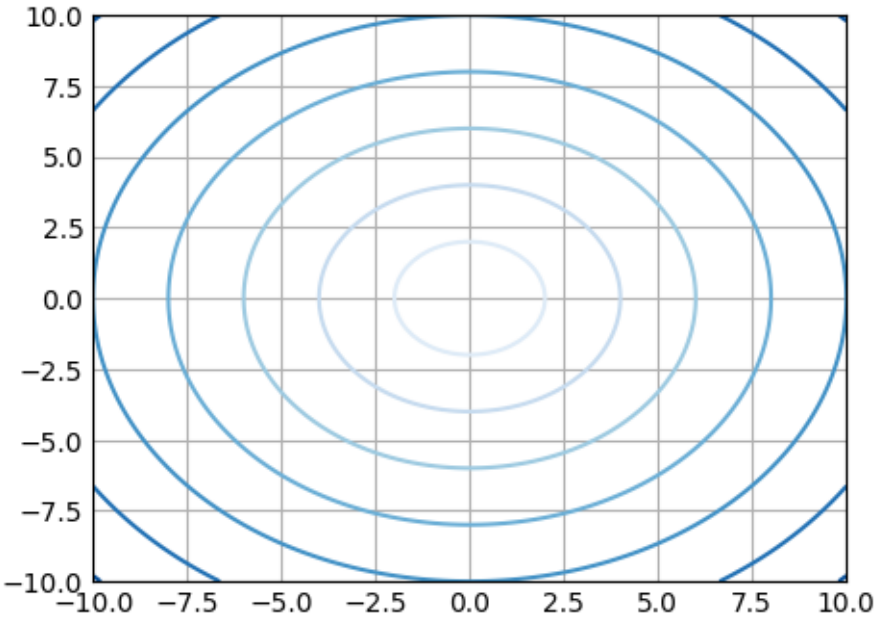
```
[1. 1. 1. 1.]
[2. 2. 2. 2.]
[3. 3. 3. 3.]]
```

Comando Countour (ploteo conjuntos de nivel)

La librería `matplotlib` proporciona la función `contour` para crear curvas de nivel.

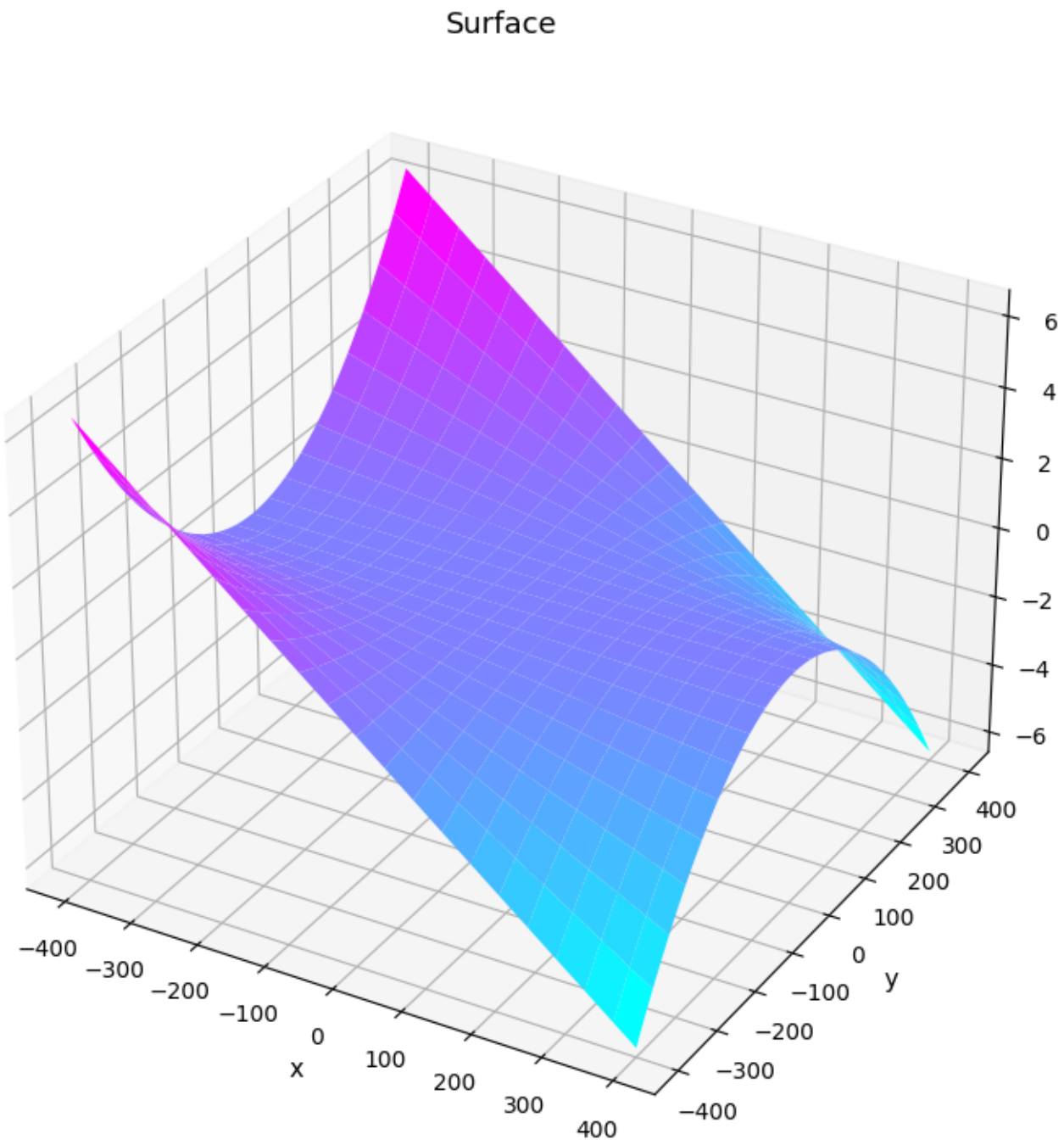
Ejemplo de uso:

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

# Datos
Xc, Yc = np.meshgrid(np.linspace(-10, 10, 100),
                     np.linspace(-10, 10, 100))
Zc = np.sqrt(Xc ** 2 + Yc ** 2)

# Contour
fig, ax = plt.subplots()
ax.contour(Xc, Yc, Zc)
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(4, 3) #tamaño de gráfica (visor)

plt.show()
```



1.7.5. Superficies

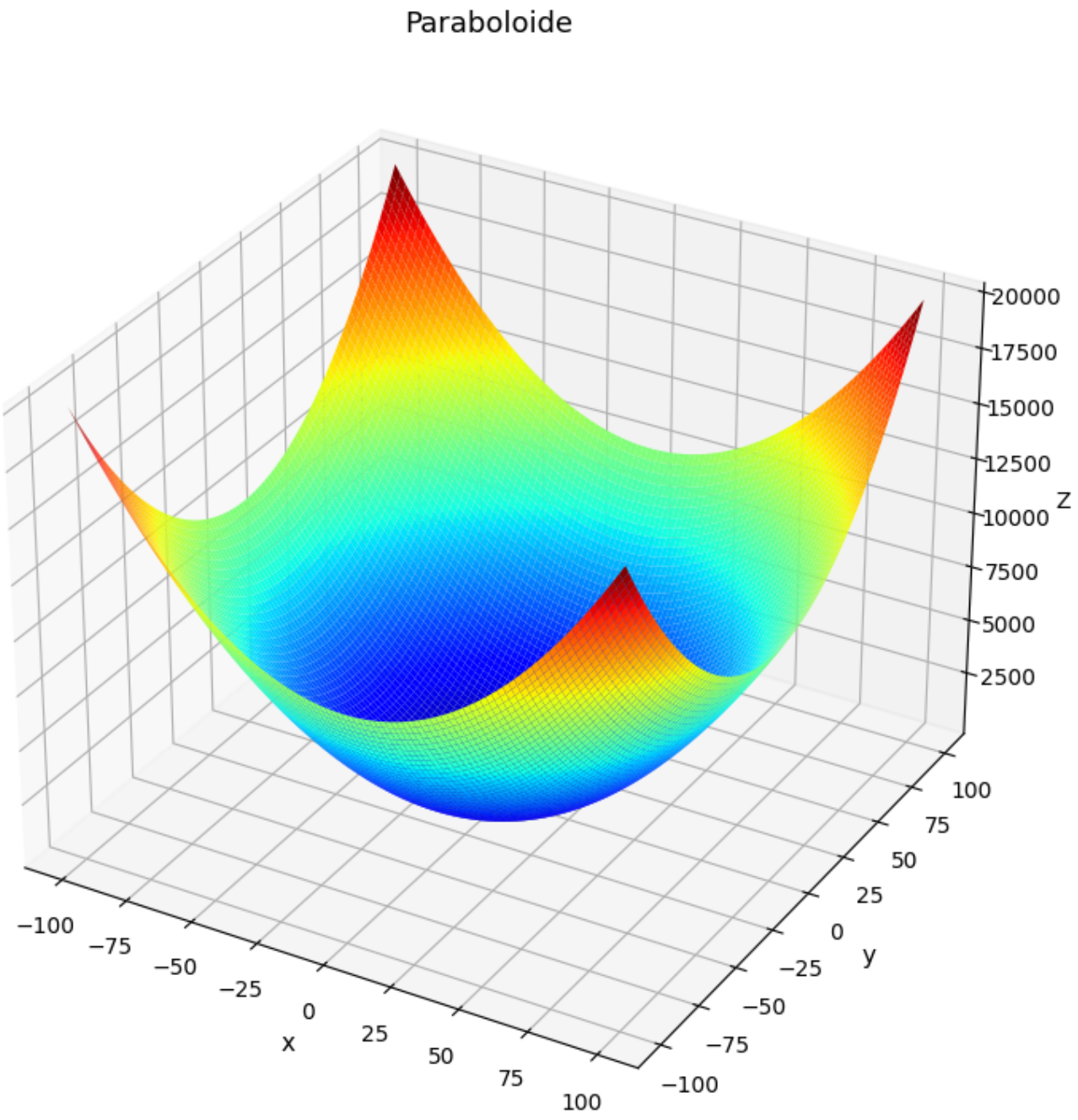
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def fonction1 (x, y):
    return 2*x**2 - x*y**2 + 2*y**2
x = np.linspace(-400, 400, 100)
y = np.linspace(-400, 400, 100)

X, Y = np.meshgrid(x, y)

Z = fonction1(X,Y)

fig = plt.figure(figsize = (10,7))
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=5, cstride=5,
                cmap='cool')

ax.set_title("Surface", fontsize = 13)
ax.set_xlabel('x', fontsize = 11)
ax.set_ylabel('y', fontsize = 11)
ax.set_zlabel('Z', fontsize = 11)
plt.show()
```


Surface
6
4
2
0
−2
−4
−6
−400
−300
−200
−100
0
100
200
300
400
x
400
300
200
100
0
−100
−200
−300
−400
y

1.7.6. Paraboloide

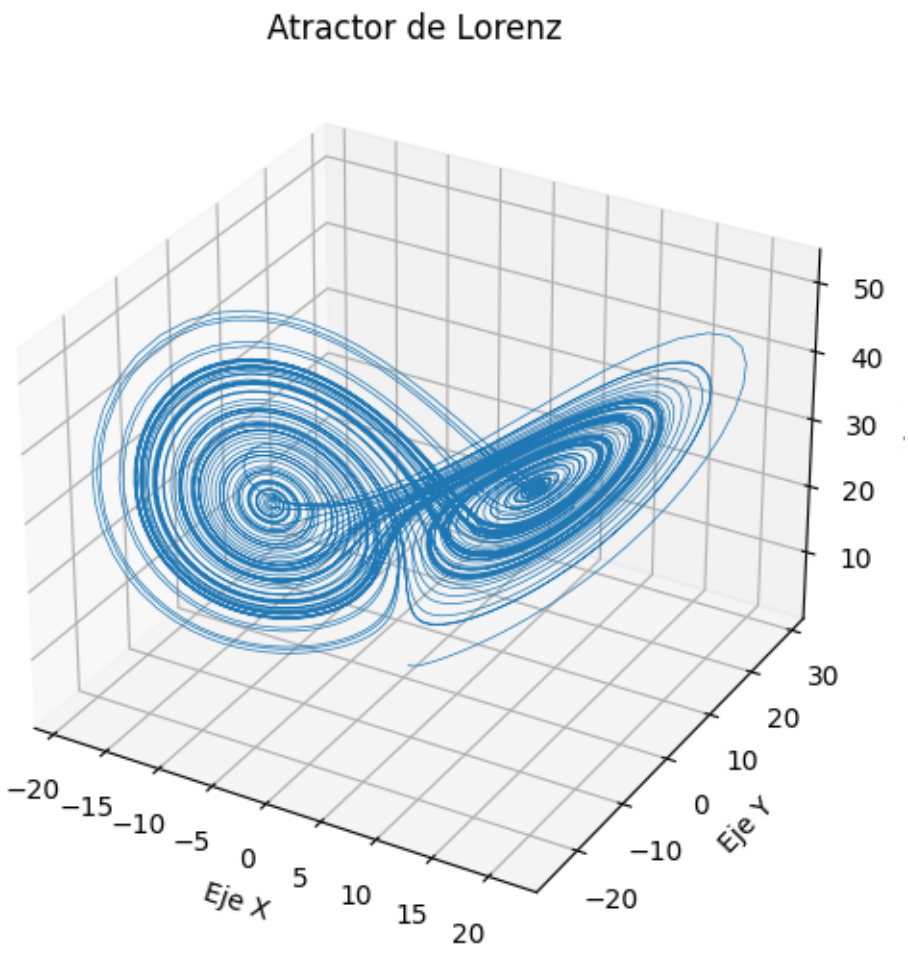
```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

def f2(x, y):
    return x**2 + y**2

x = np.linspace(-100, 100, 100)
y = np.linspace(-100, 100, 100)

X, Y = np.meshgrid(x, y)

Z = f2(X,Y)

fig = plt.figure(figsize = (10,7))
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='jet', edgecolor='none')

ax.set_title("Paraboloide", fontsize = 13)
ax.set_xlabel('x', fontsize = 11)
ax.set_ylabel('y', fontsize = 11)
ax.set_zlabel('Z', fontsize = 10)
```

```
Text(0.5, 0, 'Z')
```


Paraboloide
20000
17500
15000
12500
10000
7500
5000
2500
z
−100
−75
−50
−25
0
25
50
75
100
x
100
75
50
25
0
−25
−50
−75
−100
y

1.7.7. Atractor de Lorenz

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def lorenz(x, y, z, s=10, r=28, b=2.667):
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot


dt = 0.01
stepCnt = 10000

# Need one more for the initial values
xs = np.empty((stepCnt + 1,))
ys = np.empty((stepCnt + 1,))
zs = np.empty((stepCnt + 1,))

# Setting initial values
xs[0], ys[0], zs[0] = (0., 1., 1.05)

# Stepping through "time".
for i in range(stepCnt):
    # Derivatives of the X, Y, Z state
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.plot(xs, ys, zs, lw=0.5)
ax.set_xlabel("Eje X")
ax.set_ylabel("Eje Y")
ax.set_zlabel("Eje Z")
ax.set_title("Atractor de Lorenz")
fig = plt.gcf() #tamaño de gráfica (visor)
fig.set_size_inches(6, 7) #tamaño de gráfica (visor)

plt.show()
```



1.8. Algebra lineal con `NumPy`

1.8.1. Vectores

Creamos vectores fila y columna:

```
import numpy as np
vector_fila = np.array([[1, -5, 3, 2, 4]])
vector_columna = np.array([[1],
                           [2],
                           [3],
                           [4]])
print(vector_fila.shape) #shape (forma)
print(vector_columna.shape)
```

```
(1, 5)
(4, 1)
```

1.8.2. Calculo de Norma de un vector

```
# cargamos el módulo de algebra lineal
from numpy import linalg

v = np.array([1, 1, 1])
w = np.array([2, 2, 2])
z = np.array([1, 0, 1])

norma =linalg.norm(v)  # la norma 2 / módulo del vector v
print(norma)
print(np.sqrt(v[0]**2 + v[1]**2 + v[2]**2))  # calculado a mano == sqrt(3)
print(norma == np.sqrt(3))  # y numpy sabe que son lo mismo
```

```
1.7320508075688772
1.7320508075688772
True
```

```
from numpy.linalg import norm
new_vector = vector_fila.T # .T transpone vector
print(new_vector)
norm_1 = norm(new_vector, 1) #norma 1
norm_2 = norm(new_vector, 2) #norma euclidea
norm_inf = norm(new_vector, np.inf) #norma infinito
print('L_1 is: %.1f'%norm_1)
print('L_2 is: %.1f'%norm_2)
print('L_inf is: %.1f'%norm_inf)
```

```
[[ 1]
 [-5]
 [ 3]
 [ 2]
 [ 4]]
L_1 is: 15.0
L_2 is: 7.4
L_inf is: 5.0
```

1.8.3. Matrices

Usemos los vectores que creamos recién para crear una matríz y que `NumPy` calcule los autovectores y autovalores de esa matríz:

```
matriz = np.array([v, w, z], dtype=np.float64)
#eig devuelve una tupla de arrays con los autovalores en un array 1D y los autovec en
eigens = linalg.eig(matriz)

autvals, autvecs = eigens

print('Los autovalores:', autvals)
print()
print('Los autovectores:', autvecs)
```

Y para un sistema de ecuaciones del tipo $Ax = b$:

$$A = \begin{pmatrix} 1 & 2 & 5 \\ 2 & 5 & 8 \\ 4 & 0 & 8 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

```
from numpy import linalg

A1 = np.array([[1, 2, 5], [2, 5, 8], [4, 0, 8]], dtype=np.float64)
#print(A1)
b1 = np.array([1, 2, 3])
xs = linalg.solve(A1, b1)  # resuelve el sistema A*x = b
print(xs)
```

```
[0.67857143 0.07142857 0.03571429]
```

También se puede hacer producto de una matriz con vector, y se pueden calcular inversas de matrices cuadradas.

```
print(np.dot(A1,xs)) #producto matríz por vector (tambien se puede usar vector y vecto
print(linalg.inv(A1))
```

```
[1. 2. 3.]
[[-1.42857143  0.57142857  0.32142857]
 [-0.57142857  0.42857143 -0.07142857]
 [ 0.71428571 -0.28571429 -0.03571429]]
```

1.8.4. Determinante de matrices

```
from numpy.linalg import det

M = np.array([[0,2,1,3],
              [3,2,8,1],
              [1,0,0,3],
              [0,3,2,1]])
print('M:\n', M)

print('Determinant: %.1f'%det(M))
I = np.eye(4) #matríz identidad de tamaño 4x4
print('I:\n', I)
print('M*I:\n', np.dot(M, I))
```

```
M:
 [[0 2 1 3]
 [3 2 8 1]
 [1 0 0 3]
 [0 3 2 1]]
Determinant: -38.0
I:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
M*I:
 [[0. 2. 1. 3.]
 [3. 2. 8. 1.]
 [1. 0. 0. 3.]
 [0. 3. 2. 1.]]
```

Ejemplo: Sea la Matríz

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix},$$

calcule el número de condición y la clasificación de esta matriz.

Si $y = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$, obtenga la matriz aumentada $[A, y]$.

```
from numpy.linalg import \
             cond, matrix_rank

A = np.array([[1,1,0],
              [0,1,0],
              [1,0,1]])

print('Número de condición:\n', cond(A))
print('Rango:\n', matrix_rank(A))
y = np.array([[1], [2], [1]])
A_y = np.concatenate((A, y), axis = 1)
print('Matríz aumentada:\n', A_y)
```

```
Número de condición:
 4.048917339522305
Rango:
 3
Matríz aumentada:
 [[1 1 0 1]
 [0 1 0 2]
 [1 0 1 1]]
```

1.8.5. Resolver sistemas de ecuaciones lineales en *Python*

Analizaremos varios métodos para resolver sistemas de ecuaciones lineales, pero tambien mencionamos que es muy fácil hacerlo en *Python* con un comando intrínseco. En esta sección, usaremos *Python* para resolver sistemas de ecuaciones. La forma más sencilla de obtener una solución es mediante la función de resolución en `Numpy`.

Ejemplo: Use `numpy.linalg.solve` para resolver las siguientes ecuaciones:

$$\begin{cases} 4x_1 + 3x_2 - 5x_3 & = 2 \\ -2x_1 - 4x_2 + 5x_3 & = 5 \\ 8x_1 + 8x_2 & = -3 \end{cases}$$

```
import numpy as np

A = np.array([[4, 3, -5],
              [-2, -4, 5],
              [8, 8, 0]])
y = np.array([2, 5, -3])

x = np.linalg.solve(A, y)
print(x)
```

```
[ 2.20833333 -2.58333333 -0.18333333]
```

Ejemplo: Obtener con *Python* la descomposicón L y U de la matríz A definida antes.

```
from scipy.linalg import lu

P, L, U = lu(A)
print('P:\n', P) #P matríz de permutación
print('L:\n', L)
print('U:\n', U)
print('LU:\n',np.dot(L, U))
```

```
P:
 [[0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]]
L:
 [[ 1.    0.    0.  ]
 [-0.25  1.    0.  ]
 [ 0.5   0.5   1.  ]]
U:
 [[ 8.   8.   0. ]
 [ 0.  -2.   5. ]
 [ 0.   0.  -7.5]]
LU:
 [[ 8.  8.  0.]
 [-2. -4.  5.]
 [ 4.  3. -5.]]
```

1.9. Comandos útiles

1.9.1. Borrar todas las variables en memoria

```
reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

Bibliografía

[1] *Apunte de cátedra - Matemática Aplicada*, Ingeniería Eléctrica - Ingeniería Electrónica, Escuela de formación Básica, departamento de Matemática. Facultad de Ciencias Exactas, Ingenierıa y Agrimensura. Universidad Nacional de Rosario.

[2] *Apunte de cátedra - Taller de Fısica Computacional - Computacion Científica*, Licenciatura en Física. Facultad de Ciencias Exactas, Ingenierıa y Agrimensura. Universidad Nacional de Rosario.

[3] *Apunte de cátedra - Programación I*, Técnicatura en Inteligencia Artificial. Facultad de Ciencias Exactas, Ingenierıa y Agrimensura. Universidad Nacional de Rosario.

[4] Kiusalaas, J. , *Numerical Methods in Engineering with Python 3 (3rd ed.)*.
Cambridge: Cambridge University Press. 2013. doi:10.1017/CBO9781139523899

[5] Qingkai Kong, Timmy Siauw, Alexandre M. Bayen , *Python Programming and Numerical Methods A Guide for Engineers and Scientists* .
Elsevier: Academic Press is an imprint of Elsevier, 125 London Wall, London EC2Y 5AS, United Kingdom, 2021. ISBN: 978-0-12-819549-9

[6] Jiménez Bedolla, J. C., *Métodos numéricos usando Python con aplicaciones a la Ingeniería Química*.
Universidad Nacional Autónoma de México. Facultad de Química, Departamento de Matemáticas, 2022. D.R. © 2022 UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO. ISBN:978-607-30-5826-1