Introducción a Python

Alberto F. Hamilton Castro.

Dpto. de Ingeniería Informática y de Sistemas de la Universidad de La Laguna

27. octubre 2021

Introducción

Python es un lenguaje interpretado que ha experimentado una gran difusión en el ámbito científico en los últimos años. Los principales motivos son la sencillez de su sintaxis, el gran número de módulos y librerías disponibles y el hecho de que tanto el lenguaje como la mayoría de esas librerías son Software Libre.

Existen en la actualidad dos versiones disponibles Python2 (2.7) y la nueva versión Python3 (3.8). La versión 3, aunque es bastante compatible con la 2, tiene numerosas mejoras de rendimiento y en este momento es la recomendada.

Ejecución de Python

Para ejecutar Python tenemos varias formas:

- La forma más básica es invocando el ejecutable (`python` o mejor `python3`) desde la línea de comandos de nuestro intérprete de órdenes. Esto permite ir dando ordenes una por una. Es útil para hacer pequeñas pruebas de unas pocas líneas de código.
- Un intérprete mejorado es el IPython que se invoca con `ipython3`. Añade coloreado de sintaxis, asistente de completado y acceso a la documentación de los comandos.
- Se puede usar un editor de textos (que no un *procesador* de textos) para escribir un fichero de código fuente en Python que, por convenio, debe llevar la extensión `.py`. El programa se ejecutará después a través de la línea de comandos con la invocación

```
python3 ficheroPrograma.py
```

- Hay entornos de desarrollo o editores especializados (PyCharm, Spyder, etc.) que permiten ejecutar el código escrito directamente pulsando un botón en el GUI. Además añaden resaltado de sintaxis, visualización de variables, etc.
- Hace pocos años ha surgió el proyecto Jupyter que desarrolló las *Jupyter Notebooks*. Éstas permiten combinar código, resultados y textos informativos en distintas *celdas* de un único documento. Esto permite ver los resultados que genera un determinado código así como documentar el objetivo de dicho código o discutir los resultados obtenidos.

Jupyter Notebooks

Para utilizar las Jupyter Notebooks en nuestra máquina lo más conveniente es instalar la aplicación Anaconda, que además instala Python y cualquier librería que necesitemos de una manera fácil y organizada.

También las tenemos disponibles en la nube a través del servicio Google Colaboratory que guarda nuestros notebooks en nuestro Google Drive.

Lenguaje Python[1]

Introducción

La principal característica que distingue al lenguaje Python, de otros similares, es que el sagrado (número de espacios al principio de una línea) sirve para indicar el nivel de agrupamiento. De esta manera todas las estructuras de control (bucles, condicionales, definiciones de funciones, etc.) tienen un aspecto claro forzado por el lenguaje, lo que hace que los programas en Python sean mucho más legibles que en otros lenguajes que utilizan las llaves (`{}`) u otros símbolos para indicar los agrupamientos.

Los espacios en el resto de la línea no son significativos, pero es conveniente utilizar espacios para separa los distintos elemento: variables, operadores, palabras reservadas, etc.

```
x = 5 + (2 * w)
```

Por motivos de legibilidad, se recomienda que las líneas no superen los 79 caracteres. Si una instrucción supera esa longitud, se puede dividir en varias líneas usando el carácter contrabarra (`\`)

Los comentarios comienzan por el carácter almohadilla (#) y se extiende hasta final de línea.

Los identificadores (nombres de variables, funciones, etc.) pueden constar del letras, dígitos y y el carácter guion bajo (_). No pueden empezar por un dígito.

Se deben elegir **identificadores significativos** (que entienda el lector de nuestro código). En muchos casos es conveniente construirlos como concatenación de palabras, o bien con la aproximación *camelCase* (`volumenDeLaEsfera`), o bien la aproximación *snake_case* (`volumen_de_la_esfera`).

Otra característica, habitual en los lenguajes interpretados, es que no es necesario declarar las variables antes de usarlas y el tipo de datos que contienen puede cambiar durante la ejecución:

```
>>> x = 5
>>> x = 'Hola mundo'
```

Los tipos de datos básicos de Python son:

- números:
  - valores lógicos: `True` y `False`
  - enteros: `0 -10 800`

1 Algunas partes extraídas del tutorial de Bartolomé Sintes Marco

    - en punto flotante: `3.0 2.8e3 -1.102e-5`
    - complejos: `-3+2j 5.3+9.7j 1+1j`
- cadenas de caracteres: delimitadas por comillas simples (`'Hola'`), dobles (`"Hola"`) o triples comillas (`'''Puedo contener saltos de línea'''`).

Los operadores son los habituales de cualquier lenguaje (`+, -, *, /, >,<, ==, !=`). Destacar:

- la potencia con el doble asterisco `**` → `2**4`
- resto de la división entera con el tanto por ciento `%` → `5 % 2`
- cociente de la división entera con la doble barra `//` → `5 // 2`
- usar paréntesis para determinar la precedencia → `(4 + 3) * 5`
- los operadores lógicos son `or and not`

Estilo

Viene definido en PEP-0008. Programas *linter* verifican cumplimiento. Los más extendidos son PLint y Flake8 que pueden instalarse en editores como Atom y VSCode.

Secuencias

En Python están definidas las siguiente secuencias:

- **Lista**: Conjuntos de elementos con orden determinado. Pueden tener elementos del mismo o diferente tipo.
    - Se definen escribiendo los elementos entre corchetes y separados por comas → `[2, 9, 8 -1] ['Hola', 2.9, True]`
    - Se puede definir la lista vacía `[]` o con un elemento `[3]`
    - Se pueden concatenar dos listas con el operando suma: `[2, 'Hola'] + [1.5]`
- **Tupla**: es un conjunto ordenado e inmutable de elementos del mismo o diferente tipo
    - Se representan escribiendo los elementos entre paréntesis (no es siempre necesario) y separados por comas → `(0, 'hola', 2e-4)`.
    - Se puede definir la tupla vacía `()` y tupla con un solo elemento → `(3,)`
    - `Se puede desempaquetar (asignaciones múltiples) → x, y = (3, 5)`

```
>>> tup2 = (1, 2)
>>> x, y = tup2
>>> x
1
>>> y
2
```

- **Rango**: es una lista inmutable de números enteros en sucesión aritmética. Se definen a través de la función `range()`:
    - `range(n)` lista enteros consecutivos que empieza en 0 y acaba en `n`-1.
    - `range(m, n)` lista enteros consecutivos que empieza en `m` y acaba en `n`-1.
    - `range(m, n, p)` lista enteros que empieza en `m` y acaba justo antes de superar o igualar a `n`, aumentando los valores de `p` en `p`. `p` puede ser negativo. El valor `n`

nunca aparece.

- **Strings:** tienen las funcionalidades de una secuencia

Sobre estas secuencias tenemos las siguientes funciones:

- `len()` devuelve el número de elementos de la secuencia.
- obtener elementos individuales poniendo su índice entre corchetes a continuación del identificador de la lista. Los índices empiezan en 0.

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[0]
27
>>> fecha[1]
Octubre
>>> fecha[2]
1997
```

- se pueden usar números negativos para empezar a contar por el final.

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[-1]
1997
>>> fecha[-2]
Octubre
>>> fecha[-3]
27
```

- obtener subsecuencias utilizando la notación `nombreDeLista[inicio:límite]`

```
>>> dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"]
>>> dias[1:4] # Se extrae una lista con los valores 1, 2 y 3
['Martes', 'Miércoles', 'Jueves']
>>> dias[4:5] # Se extrae una lista con el valor 4
['Viernes']
>>> dias[4:4] # Se extrae una lista vacía
[]
>>> dias[:4]  # Se extrae una lista hasta el valor 4 (no incluido)
['Lunes', 'Martes', 'Miércoles', 'Jueves']
>>> dias[4:]  # Se extrae una lista desde el valor 4 (incluido)
['Viernes', 'Sábado', 'Domingo']
>>> dias[2:-1] # del tercer al penúltimo elementos
['Miércoles', 'Jueves', 'Viernes', 'Sábado']
>>> dias[:]   # Se extrae una lista con todos los valores
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
```

- con `del` se puede eliminar elementos, sublistas o la lista completa.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[4]   # Elimina la sublista ['E']
>>> letras
['A', 'B', 'C', 'D', 'F', 'G', 'H']
>>> del letras[1:4] # Elimina la sublista ['B', 'C', 'D']
>>> letras
['A', 'F', 'G', 'H']
>>> del letras      # Elimina completamente la lista
>>> letras
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    letras
  NameError: name 'letras' is not defined
```

- se puede recorrer su contenido con un bucle `for`.

```
>>> l = (1, "a", 3.14)
>>> l
(1, 'a', 3.14)
>>> for i in l:
...     print(i)
...
1
a
3.14
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

- otra funcionalidades:

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

lista.append(valor)
lista.insert(1,valor)
lista.pop()  # modifica lista, devuelve elemento emiminado
lista.pop(posicion)  # modifica lista, devuelve elemento emiminado

lista.remove(valor)

lista.count(valor)
lista.index(valor)

valor in lista  # devuelve True o False

listaCompleta = lista1 + lista2
lista1.extend(lista2)  # se modifica lista1
```

List comprehensions

(ver diapositiva)

Funciones sobre secuencias

(ver diapositiva)

Sobre cualquier secuencia se puede aplicar las siguientes funciones (típicas de programación funcional):

- `filter(funcion(x), sec)` selecciona elementos que cumplan función. Resultado puede tener menos elementos que `sec`. Si función es `None` se filtran los `False` de `sec`.
- `map(funcion(x) , sec)` aplica función a cada elemento y develve secuencia con los resultado (tendrá misma longitud que `sec`).
- `map(funcion(x,y), sec1, sec2)` similar a la anterior pero aplicando función a elementos correspondientes de ambas secuencias. Termina cuando se acabe una de las secuencias => resultado tendrá longitud de la más corta.
- `reduce(function(acumula,y) , sec[, initial])` aplica función al acumulador y cada uno de los elementos de la secuencia. Si se indica `initial` es el acumulador inicial, caso contrario será el primer elemento. Devuelve un escalar con el acumulador final. En Python 3 está en `functools.reduce`.

- `zip(sec1, sec2)` forma tuplas con elementos de cada secuencia. Termina cuando se acabe una de las secuencias.
- `enumerate(sec, start=0)` devuelve iterador con tuplas que contienen un contador y el elemento de la secuencia. Permite añadir índice a una secuencia:

```
>>> sec = [11, 12, 'hola', 33.4]
>>> for ind, valor in enumerate(sec):
...     print ind
...     print valor
...
0
11
1
12
2
hola
3
33.4
```

Diccionarios

```
dic = { key1: val1, key2: val2}

val1 = dic[key1]
val1 = dic.get(key1)
valD = dic.get(keyNo, default)  # devuelve default si keyNo en dic
valD = dic.setdefault(keyNo, default)  # si keyNo, devuelve su valor
        # caso constrario inserta default en dic y lo devuelve

dic.keys()  # iterador sobre keys
dic.values()  # iterador sobre valores

for key in dic:

for key in dic.keys():

for key, value in dict.iteritems():

for key, value in dict.items():  # en python3

{ k:k+2 for k in lista}  # Comprehension

if 'key' in dic:


d1.update(d2) # Mezclar diccionarios: Valores en d1 tienen preferencias sobre los de d1.
```

Sets

Elementos no se pueden repetir: diccionarios de solo keys.

```
{ 1, 2, 3 }
set([1,2,3])

{ x+2 for x in set1 }  # Comprehension
```

Dispone de las funciones matemáticas para conjuntos. Algunas de ellas con operadores equivalentes (| unión, & intersección, - diferencia, ^ diferencia simétrica)

(ver diapositiva)

Variables son punteros

Salvo que almacenen tipos básicos, las variables son punteros al objeto correspondiente. Asignar una variable a otra ⇒ dos punteros al mismo objeto ⇒ modificaciones en uno afecta a la otra variable.

Control de flujo

(ver diapositivas)

Clases

```
class DerivedClassName(Base1, Base2, Base3):

    def metodo(self, parametros ...)

    atributo = ...
```

No hay atributos ni métodos privados, si queremos se indican con `__nombre`.

Se puede sobrescribir métodos de clase base. Para llamarlos `Base1.metodo(self, param)`

Constructor

```
    def __init__(self):
        pass
```

Métodos estáticos precedidos por

- `@classmethod`: primer parámetro cls en vez de self
- `@staticmethod`: no llevan prime argumento

Atributos estáticos: se acceden con `NombreClase.Atributo`

A un objeto se le puede añadir atributo aunque no definido en la clase.

Excepciones

- Captura

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise #relanza la excepcion
```

```
else:
    # se ejecuta si no excepción
finally:
    # se ejecuta siempre (clean-up)
```

- Lanzamiento

```
raise Exception
```

Módulos

Potencialidad de Python el gran número de librerías/módulos.

Módulos que nos van a interesar:

- NumPy : librería fundamental para el cálculo científico en Python. Proporciona el manejo de matrices multidimensionales, polinomios y un largo etcétera.
- Matplotlib. : librería para crear representaciones gráficas en Python.

Podemos crear nuestros propios módulos poniendo definiciones de funciones, clases, etc en fichero e importándolo.

Módulos se pueden ejecutar directamente con la construcción:

```
if __name__ == "__main__":

    # código si se invoca directamente
```

Instalación de módulos

La herramienta principal es `pip` (y el correspondiente `pip3`), que hace instalación en carpeta de usuario de los módulos solicitados y de sus dependencias.

NumPy

Para poder utilizar las funcionalidades de NumPy es necesario, en primer lugar, importar la librería. La importación estándar es

```
>>> import numpy as np
```

a partir de este comando se accede a las funciones de NumPy con el prefijo `np.`

El tipo de datos básico que maneja la librería es el array (más correctamente *ndarray)* que es un colección multidimensional de elementos del mismo tipo. Puede haber arrays de una dimensión (*vectores*), dos dimensiones (*matrices*), tres o más dimensiones (*tensores*). La dimensión *principal* son las filas.

Creación

Las funciones más importantes para crear arrays son:

- `np.array()` se crea array a partir de listas u otros arrays:

- Si se pasa única lista → vector (una dimensión).
- Si se le pasa una lista de listas → genera una matriz (dos dimensiones)
- Así sucesivamente

```
>>> np.array([1, 3, 6])
array([1, 3, 6])
>>> np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

- `np.zeros(dim)` se crea array, de las dimensiones indicada por la tupla `dim`, con todos sus elementos a 0.

```
>>> np.zeros((3, 5))
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

- `np.zeros_like(array)` se crea con sus elementos a 0 y dimensiones como `array` pasado.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> np.zeros_like(a)
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

- `np.arange()` similar `range()` de Python pero puede usar decimales.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(10,20,0.5)
array([10. , 10.5, 11. , 11.5, 12. , 12.5, 13. , 13.5, 14. , 14.5, 15. ,
       15.5, 16. , 16.5, 17. , 17.5, 18. , 18.5, 19. , 19.5])
>>> np.arange(1.3,2.5,0.2)
array([1.3, 1.5, 1.7, 1.9, 2.1, 2.3])
```

- `np.ones()` y `np.ones_like()` son como `np.zeros()` y `np.zeros_like()` pero el array está lleno de unos.
- `np.identity(n)` devuelve matriz identidad de `n` filas y `n` columnas.

```
>>> np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

La generación de arrays de números aleatorios está en el sublibrería `np.random`:

- `np.random.random_sample(dim)` devuelve array, de las dimensiones indicada por la tupla `dim`, de números aleatorios obtenidos de una **distribución uniforme** en *[0, 1)*
- `np.random.standard_normal(dim)` devuelve array, de las dimensiones indicada por la tupla `dim`, de números aleatorios obtenidos de una **distribución normal** (media 0 y desviación estándar 1)

Atributos

Los array tienen atributos para saber, sus dimensiones su forma y su tipo de datos:

```
>>> ceros = np.zeros((2,4,3))
>>> ceros
array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]]])
>>> ceros.ndim  # Número de dimensiones
3
>>> ceros.dtype  # Tipo de datos del array
dtype('float64')
>>> ceros.shape  # tupla con el número de "elementos" en cada dimensión
(2, 4, 3)
```

Operaciones

Sobre los arrays están disponibles todos los operadores habituales (`+`, `-`, `/`, `*`, `**`) pero las operaciones se hacen **elemento a elemento** entre arrays de las mismas dimensiones.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b
array([[-1. ,  4. ,  0. ],
       [ 2. ,  8. , -5. ],
       [ 1.5,  9. ,  3.3]])
>>> a + b
array([[ 0. ,  6. ,  3. ],
       [ 6. , 13. ,  1. ],
       [ 8.5, 17. , 12.3]])
>>> b / a
array([[-1.        ,  2.        ,  0.        ],
       [ 0.5       ,  1.6       , -0.83333333],
       [ 0.21428571,  1.125     ,  0.36666667]])
>>> b ** a
array([[-1.00000000e+00,  1.60000000e+01,  0.00000000e+00],
       [ 1.60000000e+01,  3.27680000e+04,  1.56250000e+04],
       [ 1.70859375e+01,  4.30467210e+07,  4.64114844e+04]])
```

Si se opera con un escalar, se aplica el mismo escalar a todos los elementos de la matriz

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a * 3
array([[ 3,  6,  9],
       [12, 15, 18],
       [21, 24, 27]])
>>> 4 - a
array([[ 3,  2,  1],
       [ 0, -1, -2],
       [-3, -4, -5]])
>>> a ** 2
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

Para la multiplicación de matrices (como se define matemáticamente) se utiliza operador `@`

```
>>> a * b
array([[ -1. ,   8. ,   0. ],
       [  8. ,  40. , -30. ],
       [ 10.5,  72. ,  29.7]])
>>> a @ b
array([[ 7.50e+00,  4.70e+01, -1.00e-01],
       [ 1.50e+01,  1.10e+02, -5.20e+00],
       [ 2.25e+01,  1.73e+02, -1.03e+01]])
```

Otras operaciones matriciales (inversa, determinante, autovalores, etc.) están en el módulo `numpy.linalg`

```
>>> np.linalg.inv(a)
array([[-4.50359963e+15,  9.00719925e+15, -4.50359963e+15],
       [ 9.00719925e+15, -1.80143985e+16,  9.00719925e+15],
       [-4.50359963e+15,  9.00719925e+15, -4.50359963e+15]])
>>> np.linalg.det(a)
6.66133814775094e-16
```

Para obtener la traspuesta de una matriz se puede usar el atributo `T` o el método `transpose()`

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
>>> a.transpose()
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Los operadores de relación (`<, <=, ==, !=, >, >=`) operan también elemento a elemento (o con escalares) y devuelve un array de booleanos.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b
array([[-1. ,  4. ,  0. ],
       [ 2. ,  8. , -5. ],
       [ 1.5,  9. ,  3.3]])
>>> a > b
array([[ True, False,  True],
       [ True, False,  True],
       [ True, False,  True]])
>>> b != a
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> a == 5
array([[False, False, False],
       [False,  True, False],
       [False, False, False]])
>>> a < 6
array([[ True,  True,  True],
       [ True,  True, False],
       [False, False, False]])
```

Indexado

Se hace como las listas de Python.

```
>>> v = np.arange(10)
>>> v
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> v[4:8]
array([4, 5, 6, 7])
>>> v[8:]
array([8, 9])
>>> v[:-5]
array([0, 1, 2, 3, 4])
>>> v[:]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Para arrays multidimensionales se ponen las indexaciones separadas por coma:

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a[2,1]
8
>>> a[:,1]
array([2, 5, 8])
>>> a[2,:2]
array([7, 8])
>>> a[-1,-1]
9
```

Se puede hacer indexación mediante listas de enteros con `np.ix_()`

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a[np.ix_([0, 1], [1, 2])]
array([[2, 3],
       [5, 6]])
>>> a[np.ix_([0, 1], [2, 1])]
array([[3, 2],
       [6, 5]])
>>> a[np.ix_([0, 1, 0], [2, 1])]
array([[3, 2],
       [6, 5],
       [3, 2]])
```

Si la lista de enteros se va a utilizar sólo en una dimensión no hace falta `np.ix_()` y se puede usar la lista de enteros directamente

```
>>> a[:, [1, 2]]
array([[2, 3],
       [5, 6],
       [8, 9]])
>>> a[[2,2,0],1:]
array([[8, 9],
       [8, 9],
       [2, 3]])
```

También es posible utilizar vectores o arrays de booleanos para indexar

```
>>> l
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> l[l > 5]
array([6, 7, 8, 9])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a[:, -1] > 4
array([False,  True,  True])
>>> a[a[:, -1] > 4, :]
array([[4, 5, 6],
       [7, 8, 9]])
```

Reshape

El método `reshape()` permite cambiar las dimensiones de un array conservando los mismos elementos que tiene, por lo que las nuevas dimensiones tienen que ser congruentes.

```
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.reshape((3,2))
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Es posible dejar alguna de las dimensiones a `-1` para que se calcule su valor según corresponda al número de elementos.

```
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.reshape((-1,2))
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Los arrays que son vectores sólo tienen una dimensión y no son ni filas ni columnas. Por ello al trasponer un vector, da al mismo vector. Para convertirlo en vector columna habrá que usar `reshape()`:

```
>>> p=np.array([1,3,5,7])
>>> p
array([1, 3, 5, 7])
>>> p.T
array([1, 3, 5, 7])
>>> p.reshape(-1,1)
array([[1],
       [3],
       [5],
       [7]])
>>> p.T
array([1, 3, 5, 7])
```

Para convertir una matriz en un vector podemos usar `a.reshape(-1)`

Stack

Para crear arrays a partir de otros se pueden usar las funciones:

- `np.vstack()` para *apilar* matrices verticalmente
- `np.hstack()` para *apilar* matrices horizontalmente

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b
array([[11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
>>> np.hstack([a,b])
array([[ 1,  2,  3, 11, 12, 13],
       [ 4,  5,  6, 14, 15, 16],
       [ 7,  8,  9, 17, 18, 19]])
>>> np.vstack([a,b])
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
```

Agregaciones o reducciones

Son las funciones de análisis de datos: `mean`, `std`, `sum`, `cumsum`, `min`, `max`, etc. Se pueden invocar como métodos de los arrays o como funciones del módulo a las que se les pasa array como primer parámetro.

Si no se indica nada, se hace sobre todo el array. Se les puede pasar parámetro `axis` para indicar dimensión sobre la que trabajar: `axis = 0` por columnas, `axis = 1` por filas.

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a.max()
9
>>> np.max(a, axis=1)
array([3, 6, 9])
>>> a.max(axis=0)
array([7, 8, 9])
>>> a.mean()
5.0
>>> a.sum(axis=0)
array([12, 15, 18])
>>> a.cumsum(axis=1)
array([[ 1,  3,  6],
       [ 4,  9, 15],
       [ 7, 15, 24]])
```

Para saber índice del elemento donde se produjo el mínimo o máximo existen las funciones `argmin()` y `argmax()`

```
>>> r
array([[2.30531348, 8.15648592, 7.35277563, 9.07046516, 4.84514167],
       [6.73537854, 6.32334695, 4.52685555, 0.89912327, 2.94875055]])
>>> r.max()
9.070465156492052
>>> r.argmax()
3
>>> r.max(axis=0)
array([6.73537854, 8.15648592, 7.35277563, 9.07046516, 4.84514167])
>>> r.argmax(axis=0)
array([1, 0, 0, 0, 0])
```

Funciones que trabajan con valores lógicos:

- `any()` devuelve `True` si algún elemento del array es `True` (o-lógica)

- `all()` devuelve `True` solo si todos los elementos del array son `True` (y-lógica)

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a > 4
array([[False, False, False],
       [False,  True,  True],
       [ True,  True,  True]])
>>> np.all(a > 4)
False
>>> np.any(a > 4)
True
```

Si el array no es de booleanos, se considera `False` cualquier elemento igual a 0, y `True` a cualquier elemento distinto de 0 (convenio del lenguaje de programación C).

Se puede obtener los valores ordenados con la función `np.sort()`.

```
>>> r
array([[2.30531348, 8.15648592, 7.35277563, 9.07046516, 4.84514167],
       [6.73537854, 6.32334695, 4.52685555, 0.89912327, 2.94875055]])
>>> np.sort(r, axis=0)
array([[2.30531348, 6.32334695, 4.52685555, 0.89912327, 2.94875055],
       [6.73537854, 8.15648592, 7.35277563, 9.07046516, 4.84514167]])
>>> np.sort(r, axis=1)
array([[2.30531348, 4.84514167, 7.35277563, 8.15648592, 9.07046516],
       [0.89912327, 2.94875055, 4.52685555, 6.32334695, 6.73537854]])
```

Matplotlib

Librería que permite realizar distintos tipos de gráficas

La importación estándar es

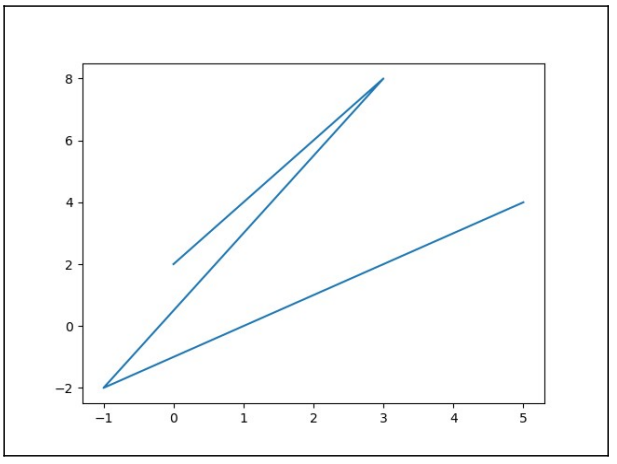
```
>>> import matplotlib.pyplot as plt
```

Graficas en 2 dimensiones

La función principal para generar una gráfica es `plt.plot()` que se puede invocar de distintas manera. Las principales son:

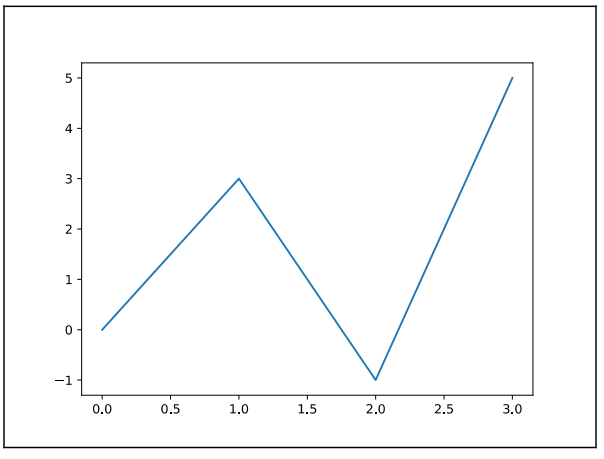
- `plot(x,y)` donde tanto `x` como `y` son vectores, se representarán los elementos de `y` frente a los de `x` , es decir, se representarán los puntos `(x[0]),y[0])`, `(x[1]),y[1]))`, `...` Las longitudes de `x` e `y` han de coincidir

```
>>> plt.plot([0, 3, -1, 5],[2, 8, -2, 4])
[<matplotlib.lines.Line2D object at 0x7fb9094db130>]
>>> plt.show()
```



- `plt.plot(y)` donde `y` es un vector, representa las componentes del vector frente a sus índices.

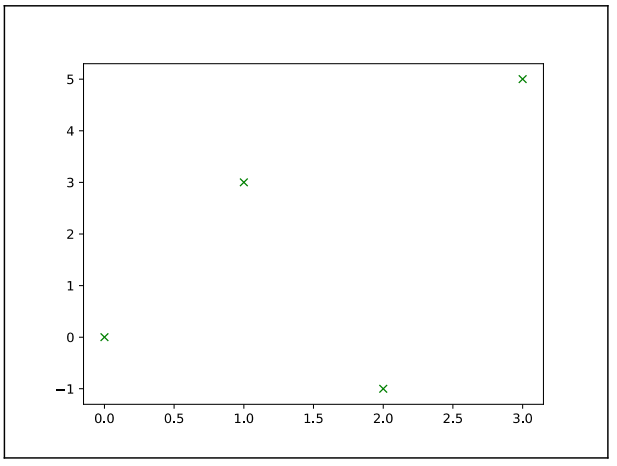
```
>>> plt.plot([0, 3, -1, 5])
[<matplotlib.lines.Line2D object at 0x7fb9094db130>]
>>> plt.show()
```



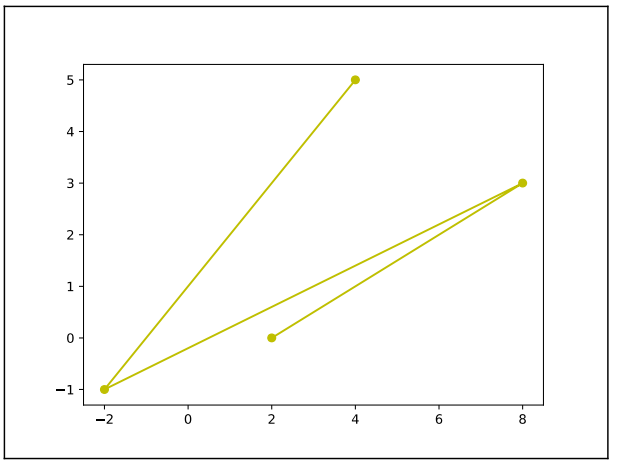
- `plt.plot(x, y, formato)` o `plt.plot(y, formato)` como las anteriores, pero en `formato` es una cadena de caracteres donde se puede indicar el color y el tipo de línea de la curva.
  - Para el tipo de línea, tenemos:
    - `'-'` segmento uniendo los datos, formato por defecto.
    - `'.'` puntos pequeños en cada dato.
    - `'x'` aspas en cada dato.
    - `'+'` cruces en cada dato.
    - `'o'` circulo en cada dato.
  - Los colores se indican con la inicial del nombre inglés del color:
    - `'r'` rojo

- `'g'` verde
- `'b'` azul
- `'m'` magenta
- `'c'` cian
- `'y'` amarillo
- `'k'` negro

```
>>> plt.plot([0, 3, -1, 5],'gx')
[<matplotlib.lines.Line2D object at 0x7febe2a5df40>]
>>> plt.show()
```

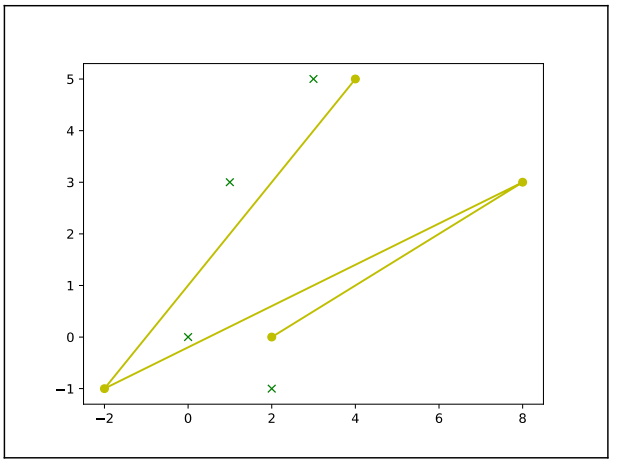


```
>>> plt.plot([2, 8, -2, 4],[0, 3, -1, 5],'yo-')
[<matplotlib.lines.Line2D object at 0x7febda51c2e0>]
>>> plt.show()
```



Para pintar varias curvas en una misma gráfica, se pueden hacer varios `plot()` antes de invocar el `show()`, o se puede componer un único `plot` con más parámetros

```
>>> plt.plot([0, 3, -1, 5],'gx', [2, 8, -2, 4],[0, 3, -1, 5],'yo-')
[<matplotlib.lines.Line2D object at 0x7febda4bd7c0>, <matplotlib.lines.Line2D object at
0x7febda4bd790>]
>>> plt.show()
```



Otras funciones similares a `plot` son:

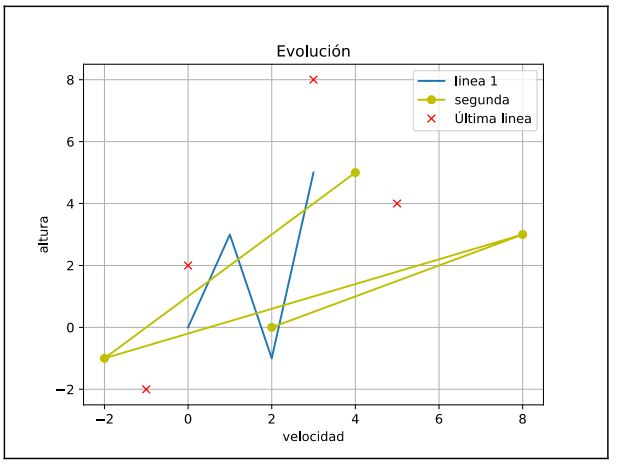
- `semilogx(arg)` recibe los mismos argumentos que `plot` pero utiliza una escala logarítmica en el eje x .
- `semilogy(arg)` recibe los mismos argumentos que `plot` pero utiliza una escala logarítmica en el eje
- `loglog(arg)` recibe los mismos argumentos que `plot` pero utiliza una escala logarítmica en los eje x e y
- `polar(angulo,modulo)` hace trazo bidimensional utilizando el ángulo y la distancia al origen para situar los puntos.

Para etiquetar la gráfica tenemos:

- `grid()` coloca rejilla en la gráfica.
- `title(string)` coloca el título a la gráfica.
- `xlabel(string)` `ylabel(string)` coloca las etiquetas en los distintos ejes.

Podemos poner nombre a cada curva con parámetro `label` y mostrarla con `legend()`

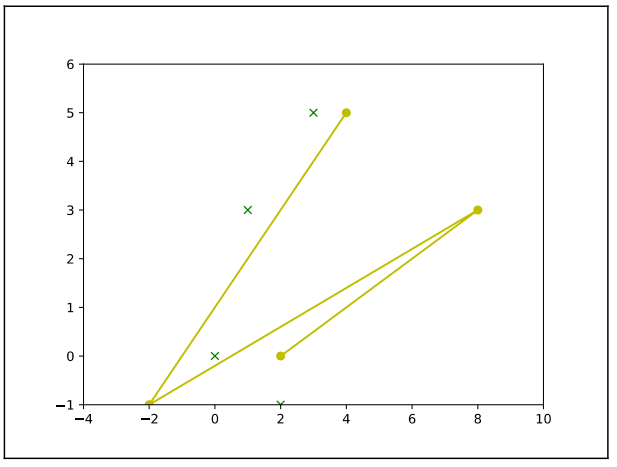
```
>>> plt.plot([0, 3, -1, 5],label='linea 1')
[<matplotlib.lines.Line2D object at 0x7f1d973e6430>]
>>> plt.plot( [2, 8, -2, 4],[0, 3, -1, 5],'yo-',label='segunda')
[<matplotlib.lines.Line2D object at 0x7f1d973e67f0>]
>>> plt.plot([0, 3, -1, 5],[2, 8, -2, 4],'rx',label='Última linea')
[<matplotlib.lines.Line2D object at 0x7f1d973e6b80>]
>>> plt.legend()
<matplotlib.legend.Legend object at 0x7f1d973d8340>
>>> plt.xlabel('velocidad')
Text(0.5, 0, 'velocidad')
>>> plt.ylabel('altura')
Text(0, 0.5, 'altura')
>>> plt.title('Evolución')
Text(0.5, 1.0, 'Evolución')
>>> plt.grid()
>>> plt.show()
```



El Matplotlib por defecto autoescala la gráfica. Para conocer o modificar el rango en cada eje está la función `axis()`:

- `axis([xmin, xmax, ymin, ymax])` fija el rango en cada eje.
- `xmin, xmax, ymin, ymax = axis()` devuelve el rango actual de cada uno de los ejes.

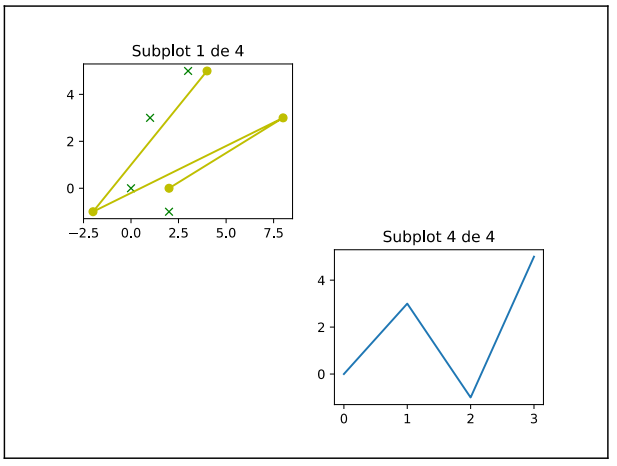
```
>>> plt.plot([0, 3, -1, 5],'gx', [2, 8, -2, 4],[0, 3, -1, 5],'yo-')
[<matplotlib.lines.Line2D object at 0x7febda4bd7c0>, <matplotlib.lines.Line2D object at
0x7febda4bd790>]
>>> plt.axis()
(-2.5, 8.5, -1.3, 5.3)
>>> plt.axis([-4, 10, -1, 6])
[-4, 10, -1, 6]
>>> plt.show()
```



Es posible tener varias subgráficas en la misma ventana:

- `subplot(f,c,a)` divide la ventana en `f` filas y `c` columnas de subgráficas y sitúa como gráfica actual la `a`-ésima.

```
>>> plt.subplot(2,2,1)
<matplotlib.axes._subplots.AxesSubplot object at 0x7fb0c02a9cd0>
>>> plt.plot([0, 3, -1, 5], 'gx', [2, 8, -2, 4], [0, 3, -1, 5], 'yo-')
[<matplotlib.lines.Line2D object at 0x7fb0c04ae730>, <matplotlib.lines.Line2D object at
0x7fb0c04ae850>]
>>> plt.title('Subplot 1 de 4')
Text(0.5, 1.0, 'Subplot 1 de 4')
>>> plt.subplot(2,2,4)
<matplotlib.axes._subplots.AxesSubplot object at 0x7fb0c04997c0>
>>> plt.plot([0, 3, -1, 5])
[<matplotlib.lines.Line2D object at 0x7fb0c03e3790>]
>>> plt.title('Subplot 4 de 4')
Text(0.5, 1.0, 'Subplot 4 de 4')
>>> plt.show()
```



Gráfica en 3 dimensiones

Para la representación de gráficas en 3D es necesario hacer uso del mplot3d Toolkit. Se debe añadir una nueva importación, por lo que las importaciones completas serían

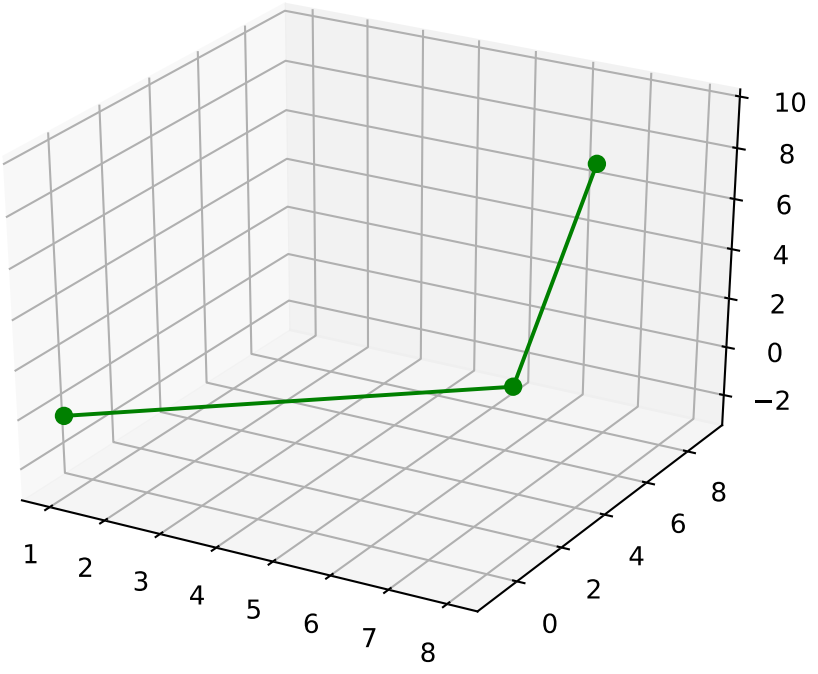
```
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
```

Es necesario crear una figura y dentro de ella un subplot con capacidades 3D

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

Sobre ese eje creado se invocará el método `plot()` con formas similares a las anteriores, pero serán necesarias tres parámetros con las coordenadas X, Y y Z de los puntos que forman las líneas

```
>>> ax.plot([1, 5, 8], [-1, 9, 4.5], [0, -3, 10],'go-')
[<mpl_toolkits.mplot3d.art3d.Line3D object at 0x7f13e87dae80>]
>>> plt.show()
```


10
8
6
4
2
0
−2
1
2
3
4
5
6
7
8
0
2
4
6
8