



Fundamentos de tratamiento de datos con el stack científico de Python



- ☰ I. Introducción
- ☰ II. Objetivos
- ☰ III. Gestión de matrices y cálculo estadístico con NumPy
- ☰ IV. Representación gráfica con Matplotlib
- ☰ V. Manipulación y análisis de datos con Pandas
- ☰ VI. Resumen
- ☰ VII. Caso práctico con solución
- ☰ VIII. Lecturas recomendadas
- ☰ IX. Glosario

≡ X. Bibliografía

≡ XI. Apéndice

I. Introducción



1.1. Introducción de la unidad

Recuerda:

Como se ha visto en las unidades anteriores, el analista y científico de datos debe enfrentarse al procesamiento de vastos volúmenes de datos, en formatos de lo más variopinto.

Sea cual sea la naturaleza de esos datos, **se requieren mecanismos de normalización** que permitan trabajar con los datos de forma homogénea y sin necesidad de adaptación a cada caso de uso.

Dentro de todos los posibles tipos de datos, una buena cantidad de ellos serán valores numéricos con los que habrá que realizar complejos cálculos matriciales.

Existen muchos lenguajes que ofrecen un amplio catálogo de funciones numéricas, como **R** o **Mathlab**, que son muy populares en el ámbito científico. No obstante, de nuevo el lenguaje de referencia para el procesamiento de datos numéricos va a ser **Python**.

A pesar de que Python es un lenguaje de propósito general, y que no está diseñado de forma específica para realizar análisis de datos, **se ha convertido en uno de los lenguajes más usados** gracias al extenso número de librerías científicas que la comunidad ha desarrollado para este fin.

A lo largo de esta unidad, el estudio se centrará en tres módulos concretos:

NumPy —

Es un módulo que incluye funciones para la manipulación de arrays multidimensionales, así como otras funciones para el cálculo estadístico. NumPy es popular no solo por facilitar la manipulación de matrices de datos, sino porque permite trabajar sobre grandes volúmenes de datos de forma eficiente.

Matplotlib —

Es un módulo que permite generar diferentes tipos de gráficas a partir de series de datos de menor o mayor complejidad. Matplotlib ofrece un catálogo de funciones de graficado similares a las ofrecidas por Matlab.

Pandas —

Inspirada en la interfaz de NumPy, Pandas es un módulo diseñado para el tratamiento de datos en formato tabular, de forma similar a los datos de un fichero CSV o en una base de datos relacional. Pandas define un conjunto de estructuras de datos y un catálogo de operaciones y hace posible manipular y filtrar los datos de forma sencilla.

 Estos tres módulos se pueden combinar de manera muy sencilla, de manera que no hace falta desarrollar mucho código para pintar datos de **Pandas** en una gráfica de **Matplotlib**, o para convertir una matriz de **NumPy** en un *dataframe* de **Pandas**.

Saber más

Cada uno de estos ejercicios se incluye en un cuaderno de Jupyter Notebook (.ipynb). Con el objetivo de familiarizarse con los distintos métodos y librerías, se recomienda descargarlos y ejecutarlos. Todos estos notebooks están en el archivo comprimido que se puede descargar en el siguiente enlace:



Ejercicios_ud_stack_python.zip

612.4 KB



II. Objetivos



2.1. Objetivos de la unidad

Los objetivos de esta unidad son:

1

Conocer la librería NumPy de Python y saber utilizarla para llevar a cabo la gestión de arrays y el cálculo estadístico.

2

Conocer la librería Matplotlib de Python y saber utilizarla para realizar representaciones gráficas de datos.

3

Conocer la librería Pandas de Python y saber utilizarla para manipular datos y analizarlos.

4

Saber utilizar de manera conjunta las librerías científicas para poder realizar un análisis de datos completo.

III. Gestión de matrices y cálculo estadístico con NumPy

El módulo NumPy (Numerical Python) es una extensión de Python que proporciona funciones y rutinas matemáticas para la manipulación de arrays y matrices de datos numéricos de una forma eficiente.

El elemento esencial de NumPy son unos objetos denominados ***ndarray*** (n-dimensional arrays), arrays multidimensionales donde todos sus elementos son del mismo tipo y están indexados por una tupla de números enteros.

Su principal ventaja es la eficiencia para manipular vectores y matrices. En este sentido, proporciona funciones que operan sobre *ndarrays*.

Cada *ndarray* tiene un conjunto de atributos que lo caracterizan:

- *ndarray.ndim*, número de dimensiones del array.
- *ndarray.dtype*, describe el tipo de elementos del array.
- *ndarray.shape*, dimensiones del array. Se trata de una tupla de enteros que indica el tamaño del array en cada dimensión. Por ejemplo, para una matriz de n filas y m columnas, sus dimensiones serían (n,m) y el número de dimensiones sería 2.

- *ndarray.size*, número total de elementos del array (producto de los elementos de la dimensión).
- *ndarray.itemsize*, el tamaño en bytes de cada elemento del array.
- *ndarray.data*, es el buffer con los elementos actuales del array. Normalmente no se usa este atributo, pues se accede a los elementos directamente mediante los índices.



Para usar NumPy, hay que importarlo. Normalmente se importa con un alias:

```
import numpy as np
```

CONTINUAR

Existen varias formas de crear un array:

Utilizando la función array

```
import numpy as np  
a = np.array( [2,3,4] )  
print (a)
```

```
[2 3 4]
```

```
a.dtype
```

```
dtype('int32')
```

Figura 1. Creación de un array usando la función array.

Fuente: elaboración propia.

La forma más sencilla de crear un array es utilizando la función **array** y una secuencia de valores (figura 1), notebook “Ejemplo_1.ipynb”. En este caso, el tipo del array resultante se deduce del tipo de elementos de las secuencias.

```
import numpy as np  
a = np.array( [[2,3,4], [5,6,7]] )  
a  
  
array([[2, 3, 4],  
       [5, 6, 7]])
```

Figura 2. Creación de un array con una secuencia de secuencias.

Fuente: elaboración propia.

Cuando a la función `array` se le pasa como argumento una secuencia de secuencias, genera un array de tantas dimensiones como secuencias (figura 2), notebook “Ejemplo_2.ipynb”.

```
import numpy as np
c = np.array( [[2,3], [5,6]], dtype=complex )
c

array([[2.+0.j, 3.+0.j],
       [5.+0.j, 6.+0.j]])
```

Figura 3. Especificación del array en el momento de su creación.

Fuente: elaboración propia.

El tipo del array puede especificarse en el momento de su creación (figura 3), notebook “Ejemplo_3.ipynb”.

Mediante la función arange()

```
import numpy as np  
a = np.arange(1,10,2)  
a
```

```
array([1, 3, 5, 7, 9])
```

Figura 4. Creación de un array con el método *arange*.

Fuente: elaboración propia.

Creación de un array mediante la función ***arange()***, que genera una secuencia de números (figura 4), notebook “Ejemplo_4.ipynb”. Toma como parámetros el rango de los números que ha de generar y la distancia entre ellos y genera un array unidimensional.

```
import numpy as np  
a = np.arange(2, 6, 0.3).reshape(2, 7)  
a  
  
array([[2. , 2.3, 2.6, 2.9, 3.2, 3.5, 3.8],  
       [4.1, 4.4, 4.7, 5. , 5.3, 5.6, 5.9]])
```

Figura 5. Redimensión de un array.

Fuente: elaboración propia.

Se puede redimensionar el array que genera `arange()` mediante el método `reshape()`, que toma como argumentos las dimensiones (figura 5), notebook “Ejemplo_5.ipynb”. Las dimensiones deben ser consistentes con el número de elementos generados.

```
import numpy as np
a = np.linspace(1,10,8)
a
array([ 1.        ,  2.28571429,  3.57142857,  4.85714286,  6.14285714,
       7.42857143,  8.71428571, 10.        ])
```

Figura 6. Uso de la función linspace.

Fuente: elaboración propia.

Cuando se usan números reales como argumentos de la función `arange()`, es difícil predecir el número de elementos del array resultante (debido a la imposibilidad de tener precisión infinita en un entorno computacional). Para esos casos es mejor usar la función `linspace` (figura 6), notebook “Ejemplo_6.ipynb”, que también genera una secuencia de números a partir de un rango dado para crear un array, pero recibe como tercer argumento el número de elementos en vez de la distancia entre ellos.

Función rand

```
import numpy as np
a = np.random.rand(10)
a

array([0.03136756, 0.56639071, 0.86969584, 0.58186369, 0.23643603,
       0.82588378, 0.52276312, 0.27644873, 0.31443282, 0.62791627])
```

Figura 7. Creación de un array con datos aleatorios.

Fuente: elaboración propia.

Creación de un array unidimensional con datos aleatorios mediante la función **rand** del módulo **Random** (figura 7), notebook “Ejemplo_7.ipynb”. La función **rand** devuelve un número aleatorio procedente de una distribución uniforme en el intervalo [0,1].

```
import numpy as np
a = np.random.rand(3,4) #valores aleatorios 3 filas y 4 columnas
a

array([[0.68768809, 0.1936327 , 0.32025235, 0.26777976],
       [0.12570141, 0.28699652, 0.90407167, 0.54861304],
       [0.52002797, 0.5257531 , 0.18119511, 0.07235105]])
```

Figura 8. Creación de un array multidimensional.

Fuente: elaboración propia.

También es posible generar arrays multidimensionales si se proporcionan las dimensiones como argumentos (figura 8), notebook “Ejemplo_8.ipynb”.

```

In [1]: import numpy as np
a = np.zeros(4)
a

Out[1]: array([0., 0., 0., 0.])

In [2]: b = np.ones ([1 ,2])
b

Out[2]: array([[1., 1.]))

In [3]: c = np.empty((4,4))
c

Out[3]: array([[6.23042070e-307, 4.67296746e-307, 1.69121096e-306,
   1.37961981e-306],
   [7.56587584e-307, 1.37961302e-306, 1.05699242e-307,
   8.01097889e-307],
   [1.78020169e-306, 7.56601165e-307, 1.02359984e-306,
   1.33510679e-306],
   [2.22522597e-306, 1.24611674e-306, 1.29061821e-306,
   2.29178686e-312]]))

```

Figura 9. Otras funciones de creación de arrays.

Fuente: elaboración propia.

Existen funciones que generan arrays especiales que solo requieren como argumento el tamaño del array (figura 9), notebook “Ejemplo_9.ipynb”:

- `zeros`: crea un array lleno de ceros.
- `ones`: crea un array lleno de unos.
- `empty`: crea un array con valores sin inicializar cuyo contenido es aleatorio.

En NumPy **se puede operar aritméticamente sobre los arrays como si se tratara de escalares**.

Esto significa que los operadores aritméticos actúan sobre cada elemento del array, produciendo un nuevo array resultante (figura 10), *notebook “Ejemplo_10.ipynb”*:

```
import numpy as np  
a = np.array([34,12,3,4])  
a
```

```
array([34, 12, 3, 4])
```

```
b = np.array([2,3,5,6])  
b
```

```
array([2, 3, 5, 6])
```

```
c=a-b  
c
```

```
array([32, 9, -2, -2])
```

```
d=a*10  
d
```

```
array([340, 120, 30, 40])
```

Figura 10. Operaciones aritméticas entre dos arrays.

Fuente: elaboración propia.

CONTINUAR



Cuando se opera con arrays de diferente tipo de elementos, **el tipo de los elementos del array resultante será el mismo que el del array con tipos de mayor precisión**. Esto se conoce como “upcasting”. Por ejemplo, si se suma un array de números enteros con un array de números reales, el array resultante contendrá números reales (figura 11), notebook “Ejemplo_11.ipynb”:

```
import numpy as np
a = np.array([2,3,6,7])
a
array([2, 3, 6, 7])

b=np.linspace(2,3,4)
b
array([2.          , 2.33333333, 2.66666667, 3.          ])

c=a+b
c
array([ 4.          ,  5.33333333,  8.66666667, 10.         ])

c.dtype
dtype('float64')
```

Figura 11. Operaciones entre arrays de diferente tipo.

Fuente: elaboración propia.



En NumPy existe un conjunto de funciones matemáticas denominadas **funciones universales**: *sin, cos, exp...* (figura 12), notebook “Ejemplo_12.ipynb”. Estas funciones operan elemento a elemento y generan como resultado un nuevo array.

```
import numpy as np
a = np.array([-7, 60, -9])
a

array([-7, 60, -9])

b=np.square(a)
b

array([ 49, 3600,    81], dtype=int32)

c=np.sqrt(np.abs(a))
c

array([2.64575131, 7.74596669, 3.        ])
```

Figura 12. Funciones universales sobre arrays.

Fuente: elaboración propia.

```
In [9]: import numpy as np  
a = np.array([[1, 0], [0, 1]])  
b = np.array([[4, 1], [2, 2]])
```

```
In [10]: c=np.dot(a,b)  
c
```

```
Out[10]: array([[4, 1],  
                 [2, 2]])
```

```
In [11]: c = a @ b  
c
```

```
Out[11]: array([[4, 1],  
                 [2, 2]])
```

Figura 13. Multiplicación matricial.

Fuente: elaboración propia.

En particular es muy útil la función `dot()`, que permite realizar la multiplicación matricial de dos arrays. No obstante, a partir de la versión 3.5 de Python se introdujo el operador `@`, que permite realizar esta operación (figura 13), notebook “Ejemplo_13.ipynb”.

CONTINUAR



Existen algunas operaciones implementadas como métodos de la clase *ndarray* y, por defecto, se aplican a todos los elementos del array. Algunas de las más importantes son:

1. **sum**: suma todos los elementos del array.
2. **min**: obtiene el menor valor del array.
3. **max**: obtiene el mayor valor del array.

Sin embargo, es posible especificar la dimensión sobre la que se quiere aplicar la operación mediante el argumento **axis** (figura 14), notebook “Ejemplo_14.ipynb”, que toma el valor 0 (columnas) o 1 (filas).

```
import numpy as np
a = np.random.rand(3,4)
a

array([[0.58584044, 0.62487763, 0.18664195, 0.18512665],
       [0.22869104, 0.80880337, 0.76676978, 0.84193634],
       [0.22640765, 0.48739203, 0.84777263, 0.8589042 ]])

a.sum()

6.6491636921061925

a.sum(axis=0)

array([1.04093912, 1.92107303, 1.80118436, 1.88596719])
```

Figura 14. Uso del argumento axis.

Fuente: elaboración propia.

CONTINUAR



Los operadores `+=` y `*=` modifican los arrays en vez de crear uno nuevo (figura 15), *notebook* "Ejemplo_15.ipynb":

El acceso a los arrays se realiza de forma indexada, de manera que se pueden seleccionar subrangos y se puede iterar sobre sus elementos. Téngase en cuenta lo siguiente:

- Cuando se accede a un array por índice, se indica un número por cada dimensión.
- Cuando se elige un subrango, se indica por cada dimensión `a:b:c`, donde `a` indica dónde se comienza, `b` dónde se termina y `c` el salto entre elementos.

```
import numpy as np  
a = np.array([2,3,5,6])  
a
```

```
array([2, 3, 5, 6])
```

```
a+=3
```

```
a
```

```
array([5, 6, 8, 9])
```

```
a*=a
```

```
a
```

```
array([25, 36, 64, 81])
```

Figura 15. Modificación de arrays.

Fuente: elaboración propia.

Cuando se trabaja con arrays de una dimensión, el acceso a los elementos se realiza de forma similar al de las listas o tuplas de elementos (figura 16), notebook “Ejemplo_16.ipynb”:

```
import numpy as np
arr = np.arange(2,20)
arr

array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
       19])

arr[0]

2

arr[1::3]

array([ 3,  6,  9, 12, 15, 18])
```

Figura 16. Acceso indexado a un array unidimensional.

Fuente: elaboración propia.

Una diferencia importante con las listas es que **las particiones de un ndarray mediante la notación [inicio:fin:paso] son vistas del array original**. Todos los cambios realizados en las vistas se reflejan en el array original (figura 17), notebook “Ejemplo_17.ipynb”:

```
import numpy as np
arr = np.arange(2,20)
arr

array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
       19])

b = arr[-2:]
b[::] = 0
arr

array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,  0,
       0])
```

Figura 17. Modificaciones de un array.

Fuente: elaboración propia.

El acceso a los elementos de un array bidimensional se realiza indicando los índices separados por una coma. Es importante recordar que los índices de los arrays en Python comienzan en la posición 0, de manera que un array de tres elementos permitirá el acceso a ellos mediante los índices 0, 1 y 2.

Asimismo, existe una diferencia importante en la notación para acceder a elementos de un array multidimensional en NumPy, con respecto a una lista multidimensional de Python. En una lista Python tradicional se accede al elemento de la posición x, y mediante la notación **[x][y]**. NumPy introduce una notación alternativa, más cercana al lenguaje matemático: **[x, y]**.

Dado que puede haber un número elevado de dimensiones, se puede utilizar una secuencia de tres puntos, “...”, para indicar todos los valores de cada dimensión a partir de una dada. Por ejemplo, **[1, ...]** accede al elemento en la posición **1** de la primera dimensión y todas sus subdimensiones (figura 18), *notebook “Ejemplo_18.ipynb”*:

```
In [7]: import numpy as np  
b = np.array([  
    [[ 0, 1, 2, 3],  
     [10,11,12,13]],  
    [[20,21,22,23],  
     [30,31,32,33]],  
    [[40,41,42,43],  
     [50,51,52,53]]  
])  
b
```

```
Out[7]: array([[ [ 0, 1, 2, 3],  
                  [10, 11, 12, 13]],  
  
                 [[20, 21, 22, 23],  
                  [30, 31, 32, 33]],  
  
                 [[40, 41, 42, 43],  
                  [50, 51, 52, 53]]])
```

```
In [11]: b[2,1] #Elemento de la fila 3 y columna 2
```

```
Out[11]: array([50, 51, 52, 53])
```

```
In [12]: b[0:2,1] #Elementos de todas las filas de la columna 2
```

```
Out[12]: array([[10, 11, 12, 13],  
                 [30, 31, 32, 33]])
```

```
In [19]: b[1,1,...] #Elementos de la fila 2, columna 2
```

```
Out[19]: array([30, 31, 32, 33])
```

Figura 18. Acceso de un array de varias dimensiones.

Fuente: elaboración propia.

```
b[0] #Toda la fila 1
```

```
array([0, 1, 2, 3])
```

Figura 19. Acceso a un array multidimensional indicando menos dimensiones.

Fuente: elaboración propia.

Cuando se accede a un array y se indican menos índices que el número de dimensiones, se consideran todos los valores de las dimensiones no indicadas (figura 19), *notebook* “Ejemplo_19.ipynb”.

Otra forma de acceso a partes de un array de NumPy es mediante un array de booleanos denominado máscara (figura 20), *notebook* “Ejemplo_20.ipynb”:

```
import numpy as np
n = np.arange(10000)
n
array([ 0, 1, 2, ..., 9997, 9998, 9999])

(n > 20) & (n < 44) #Comprobar si se cumple la condición en los elementos del array
array([False, False, False, ..., False, False, False])

mascara = (n > 20) & (n < 44) #Se crea la máscara con valor booleano
mascara
array([False, False, False, ..., False, False, False])

n[mascara] #Se usa máscara para recuperar los valores que cumplen con la condición
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
       38, 39, 40, 41, 42, 43])
```

Figura 20. Acceso a un array mediante una máscara.

Fuente: elaboración propia.

CONTINUAR



Hay varias formas de iterar sobre los elementos de un array:

Bucle for

```
In [1]: import numpy as np  
a = np.array([[1, 2, 3, 4],  
             [5, 6, 7, 8],  
             [9, 10, 11, 12]])  
a  
  
Out[1]: array([[ 1,  2,  3,  4],  
                 [ 5,  6,  7,  8],  
                 [ 9, 10, 11, 12]])  
  
In [2]: for i in a:  
        print (i)  
  
[1 2 3 4]  
[5 6 7 8]  
[ 9 10 11 12]
```

Figura 21. Recorrido de un array usando un for.

Fuente: elaboración propia.

Usando un bucle **for**, se puede iterar sobre la dimensión superior ($axis = 0$) (figura 21), notebook “Ejemplo_21.ipynb”.

Atributo flat

```
In [1]: import numpy as np  
a = np.array([[1, 2, 3, 4],  
             [5, 6, 7, 8],  
             [9, 10, 11, 12]])  
a  
  
Out[1]: array([[ 1,  2,  3,  4],  
                 [ 5,  6,  7,  8],  
                 [ 9, 10, 11, 12]])  
  
In [2]: for i in a.flat:  
         print (i)  
  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

Figura 22. Recorrido de un array usando el iterador flat.

Fuente: elaboración propia.

Usando el atributo **flat** de un array multidimensional, que permite iterar sobre todos los elementos del array: aplana el array y lo convierte en un array unidimensional (figura 22), *notebook* “Ejemplo_22.ipynb”:

Los arrays de NumPy ofrecen diferentes métodos y atributos que permiten realizar transformaciones. Los que se van a ver a continuación crean copias modificadas, dejando intacto el array original:

- **ravel()**: permite aplanar un array multidimensional en uno unidimensional.
- **reshape()**: permite cambiar la estructura del array.
- **T**: transpone el array.

```
In [6]: import numpy as np  
a = np.arange(12)  
a  
  
Out[6]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])  
  
In [7]: b = a.reshape(3,4)  
b  
  
Out[7]: array([[ 0,  1,  2,  3],  
                 [ 4,  5,  6,  7],  
                 [ 8,  9, 10, 11]])  
  
In [8]: a.ravel()  
  
Out[8]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])  
  
In [9]: b.T  
  
Out[9]: array([[ 0,  4,  8],  
                  [ 1,  5,  9],  
                  [ 2,  6, 10],  
                  [ 3,  7, 11]])
```

Figura 23. Cambios de estructura.

Fuente: elaboración propia.

```
import numpy as np
a = np.arange(4).reshape(2,2)
a
array([[0, 1],
       [2, 3]])

b = np.arange(5,9,1).reshape(2,2)
b
array([[5, 6],
       [7, 8]])

np.concatenate((a,b),axis=0) #Fusión vertical
array([[0, 1],
       [2, 3],
       [5, 6],
       [7, 8]])

np.concatenate((a,b),axis=1) #Fusión horizontal
array([[0, 1, 5, 6],
       [2, 3, 7, 8]])
```

Figura 24. Fusión de dos arrays.

Fuente: elaboración propia.

Fusionar dos arrays (figura 24), notebook “Ejemplo_24.ipynb”, mediante la función **concatenate()** verticalmente (atributo axis=0) o bien horizontalmente (atributo axis=1).

Atributo flat

```
import numpy as np
a = np.arange(12).reshape(2,6)
a

array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])

np.hsplit(a,3) #División de forma horizontal

[array([[0, 1],
       [6, 7]]),
 array([[2, 3],
       [8, 9]]),
 array([[ 4,  5],
       [10, 11]])]

np.vsplit(a,2) #División de forma vertical

[array([[0, 1, 2, 3, 4, 5]]), array([[ 6,  7,  8,  9, 10, 11]])]
```

Figura 25. División de un array.

Fuente: elaboración propia.

Dividir un array en partes iguales usando las funciones **hsplit()** y **vsplit()**, indicando el número de divisiones que se realizarán del array. Es importante resaltar que el número de divisiones debe ser divisible entre los elementos de las dimensiones del array (figura 25), *notebook* “Ejemplo_25.ipynb”.

```
import numpy as np
a = np.arange(10).reshape(2,5)
a

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

b,c=a #División vertical
b

array([0, 1, 2, 3, 4])

c

array([5, 6, 7, 8, 9])
```

Figura 26. División de un array por filas.

Fuente: elaboración propia.

Alternativamente se podría descomponer el array, lo cual se hace por filas (figura 26), notebook “Ejemplo_26.ipynb”.

```
import numpy as np
a = np.array( [ [1,2],[3,4] ] )

a
array([[1, 2],
       [3, 4]])

b = a
b[0][1] = 3000

a
array([[ 1, 3000],
       [ 3,    4]])

b
array([[ 1, 3000],
       [ 3,    4]])
```

Figura 27. Ejemplo de asignación.

Fuente: elaboración propia.

Las asignaciones y las llamadas a funciones no hacen copia del array ni de sus datos. Esto implica que si hay dos variables que apuntan al mismo array, si se hacen cambios en una de las variables, ese cambio será visible en la otra (figura 27), notebook “Ejemplo_27.ipynb”.

```
import numpy as np
a = np.array( [ [1,2],[3,4] ] )

a
array([[1, 2],
       [3, 4]])

c = a.view()
c
array([[1, 2],
       [3, 4]])

c[0][1] = 3000
c
array([[ 1, 3000],
       [ 3,     4]])
```

Figura 28. Uso del método `view()`.

Fuente: elaboración propia.

El método `view()` permite crear un nuevo array que toma los datos del array original, pero sin tratarse del mismo. Sin embargo, el array original se ve afectado por las operaciones que se hagan sobre la vista (figura 28), notebook “Ejemplo_28.ipynb”.

```
import numpy as np  
a = np.array( [ [1,2],[3,4] ] )  
  
d=a.copy()  
d  
  
array([[1, 2],  
       [3, 4]])  
  
d[1,1]=999  
d  
  
array([[ 1,   2],  
       [ 3, 999]])  
  
a  
  
array([[1, 2],  
       [3, 4]])
```

Figura 29. Copia de un array.

Fuente: elaboración propia.

El método **copy()** permite realizar una copia independiente del array original (figura 29), notebook “Ejemplo_29.ipynb”:

```
import numpy as np
a = np.array([9,3,5,2,5,1])
a
array([9, 3, 5, 2, 5, 1])

a.sort()
a
array([1, 2, 3, 5, 5, 9])

b = np.array([[4,1],[2,3]])
b
array([[4, 1],
       [2, 3]])

b.sort(0) #Ordenación por columnas
b
array([[2, 1],
       [4, 3]])
```

Figura 30. Ordenación de arrays.

Fuente: elaboración propia.

Es posible ordenar los arrays usando el método **sort()**. En el caso de arrays multidimensionales, hay que indicar la dimensión sobre la que se ordena (figura 30), notebook “Ejemplo_30.ipynb”.

```
import numpy as np  
b = np.array([False,True,False,False])  
b.any()
```

True

Figura 31. Operación sobre arrays.

Fuente: elaboración propia.

Se puede operar sobre arrays de booleanos mediante los métodos **any** y **all**. Permiten chequear si algún valor es cierto o si todos los valores son ciertos respectivamente (figura.31), *notebook* "Ejemplo_31.ipynb".

CONTINUAR

Atención:

Por último, es apropiado comentar que el módulo NumPy **proporciona métodos que permiten realizar otras operaciones matemáticas**, como obtener el mínimo elemento de un array, el máximo, álgebra lineal, operaciones sobre conjuntos... Se van a mostrar algunas de las operaciones estadísticas que facilita:



Suma, mínimo, máximo (figura 32), notebook "Ejemplo_32.ipynb":

```
import numpy as np
a = np.array([[100, 5, 6, 52],
              [80, 65, 4, 61]])
a

array([[100, 5, 6, 52],
       [80, 65, 4, 61]])

np.sum(a) #Suma todos los elementos
373

print(np.min(a), np.argmin(a)) #Valor mínimo y su posición
4 6

print(np.max(a), np.argmax(a)) #Valor máximo y su posición
100 0
```

Figura 32. Suma, máximo y mínimo.

Fuente: elaboración propia.



Media. Es la suma de todos los elementos dividida entre el número de elementos. En Python puede calcularse usando la función `numpy.mean` (figura 33), notebook "Ejemplo_33.ipynb":

```
import numpy as np
a = np.array([[100, 5, 6, 52],
              [80, 65, 4, 61]])
a
array([[100, 5, 6, 52],
       [80, 65, 4, 61]])

np.mean(a) #Media del array, considerando todos su valores
46.625

np.mean(a, axis=0) #Media por columnas
array([90., 35., 5., 56.5])
```

Figura 33. Media.

Fuente: elaboración propia.



Varianza. Es una medida de dispersión de una variable aleatoria definida como la esperanza del cuadrado de dicha variable respecto a su media. En Python puede calcularse usando la función `numpy.var` (figura 34), notebook “Ejemplo_34.ipynb”:

```
import numpy as np  
a = np.array([[100, 5, 6, 52],  
              [80, 65, 4, 61]])  
a
```

```
array([[100, 5, 6, 52],  
       [80, 65, 4, 61]])
```

```
np.var(a, axis=0) #Varianza por columnas
```

```
array([100. , 900. , 1. , 20.25])
```

Figura 34. Varianza.

Fuente: elaboración propia.



Desviación estándar. La desviación estándar es una medida de dispersión de una variable que se define como la raíz cuadrada de la varianza de la variable. En Python se puede calcular usando la función `numpy.std` (figura 35), notebook “Ejemplo_35.ipynb”:

```
In [1]: import numpy as np
a = np.array([[100, 5, 6, 52],
              [80, 65, 4, 61]])
a
```

```
Out[1]: array([[100, 5, 6, 52],
               [80, 65, 4, 61]])
```

```
In [2]: np.std(a, axis=0) #Desviación estandar (por columnas)
```

```
Out[2]: array([10., 30., 1., 4.5])
```

```
In [ ]:
```

Figura 35. Desviación estándar.

Fuente: elaboración propia.



Mediana. Representa el valor de la variable de posición central en un conjunto de datos ordenados. En Python puede calcularse usando la función *numpy.median* (figura 36), notebook “Ejemplo_36.ipynb”:

```
In [1]: 1 import numpy as np
        2 a = np.array([[1, 2], [3, 4]])
        3 a
```

```
Out[1]: array([[1, 2],
               [3, 4]])
```

```
In [2]: 1 np.median(a, axis=1) #Mediana por filas
```

```
Out[2]: array([1.5, 3.5])
```

Figura 36. Mediana.
Fuente: elaboración propia.

- **Correlación.** Dadas dos variables aleatorias, este coeficiente indica si están relacionadas o no. En Python puede calcularse usando `numpy.corrcoef`. La función devuelve los coeficientes de correlación (figura 37), notebook “Ejemplo_37.ipynb”:

```
"""
Se calcula el coeficiente de correlación teniendo
en cuenta que cada fila representa una variable
"""

import numpy as np
a = np.array([[2,3,4,5], [4,5,6,6]])
a

array([[2, 3, 4, 5],
       [4, 5, 6, 6]])

np.corrcoef(a) #correlación de la transpuesta

array([[1.          , 0.94387981],
       [0.94387981, 1.          ]])
```

Figura 37. Correlación.
Fuente: elaboración propia.

- **Covarianza.** Determina si existe una dependencia entre dos variables aleatorias. En Python se puede calcular usando `numpy.cov` (figura 38), notebook “Ejemplo_38.ipynb”:

```

import numpy as np
a = np.array([[2,3,4,5], [4,5,6,6]])
a

array([[2, 3, 4, 5],
       [4, 5, 6, 6]])

np.cov(a[:,0]) #covarianza de la primera columna

array(2.)

```

Figura 38. Covarianza.

Fuente: elaboración propia.



Métodos aleatorios (figura 39), notebook “Ejemplo_39.ipynb”:

```

import numpy.random as r
s = r.rand(10) #Genera los números aleatorios de una normal (0,1)
s

array([0.99878714, 0.75276533, 0.36091403, 0.58516899, 0.34515976,
       0.10511112, 0.08724998, 0.99505058, 0.30441745, 0.8198469 ])

t=r.normal(size=(5,1)) #Genera un array con números pertenecientes a una normal
t

array([[ 0.244925 ],
       [-0.06956181],
       [-0.14979204],
       [-0.51478657],
       [ 0.59603388]])

k=r.normal() #Genera un número perteneciente a una normal
k

-0.5558405645210815

```

Figura 39. Métodos aleatorios.

Fuente: elaboración propia.

IV. Representación gráfica con Matplotlib

Matplotlib es una librería de Python para realizar gráficos. Se caracteriza por que es fácil de usar, flexible y se puede configurar de múltiples maneras.

Para importar Matplotlib desde cualquier programa de Python, se hará de la siguiente manera: **import matplotlib.pyplot as plt**.

Y para usar un comando de Matplotlib se usará el estilo siguiente: `plt.comando()`.

El principal comando de Matplotlib es `plot()`, el cual permite representar gráficamente una lista de pares de valores (x, y) sobre un eje de coordenadas uniendo cada punto de acuerdo al orden en que aparecen. En el siguiente ejemplo, se representa la lista de pares de valores: $(1, -3), (2, 1), (3, 4), (0, 5)$ (figura 40), *notebook "Ejemplo_40.ipynb"*. **El comando %matplotlib inline permite ver un gráfico dentro de un notebook.**

```
%matplotlib inline  
import matplotlib.pyplot as plt  
plt.plot([1,2,3,0],[-3,1,4,5])  
plt.show()
```

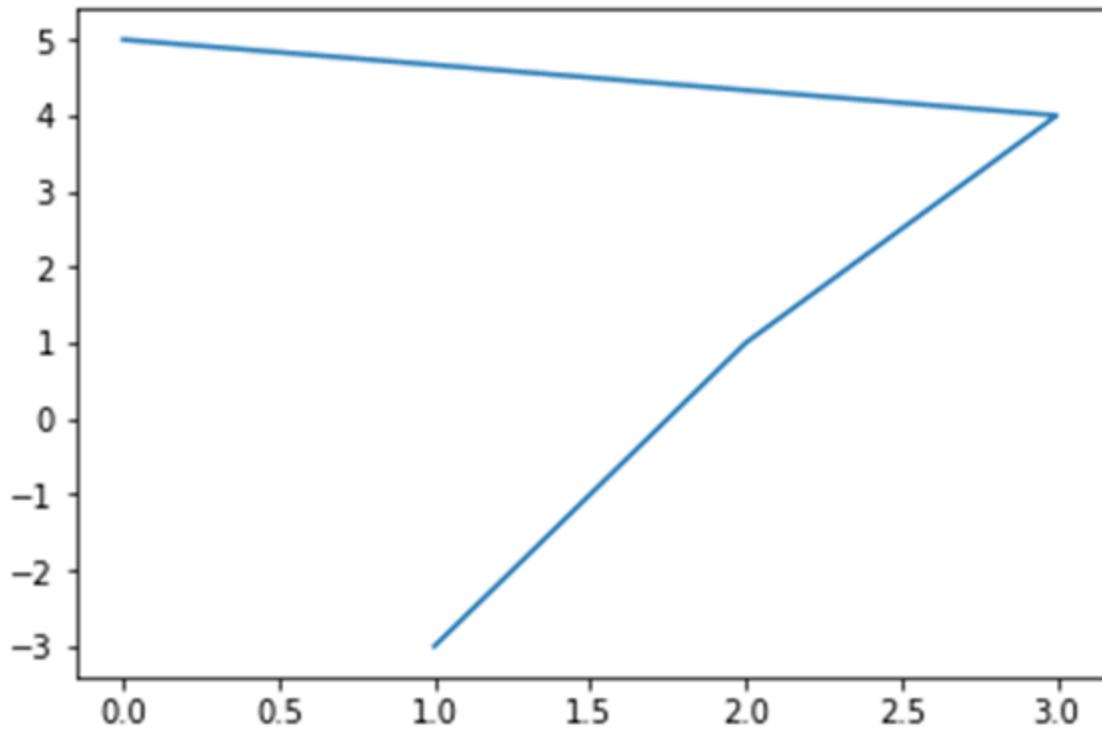


Figura 40. Uso de plot().

Fuente: elaboración propia.

```
%matplotlib inline  
import matplotlib.pyplot as plt  
plt.plot([-3,1,4,5])  
plt.show()
```

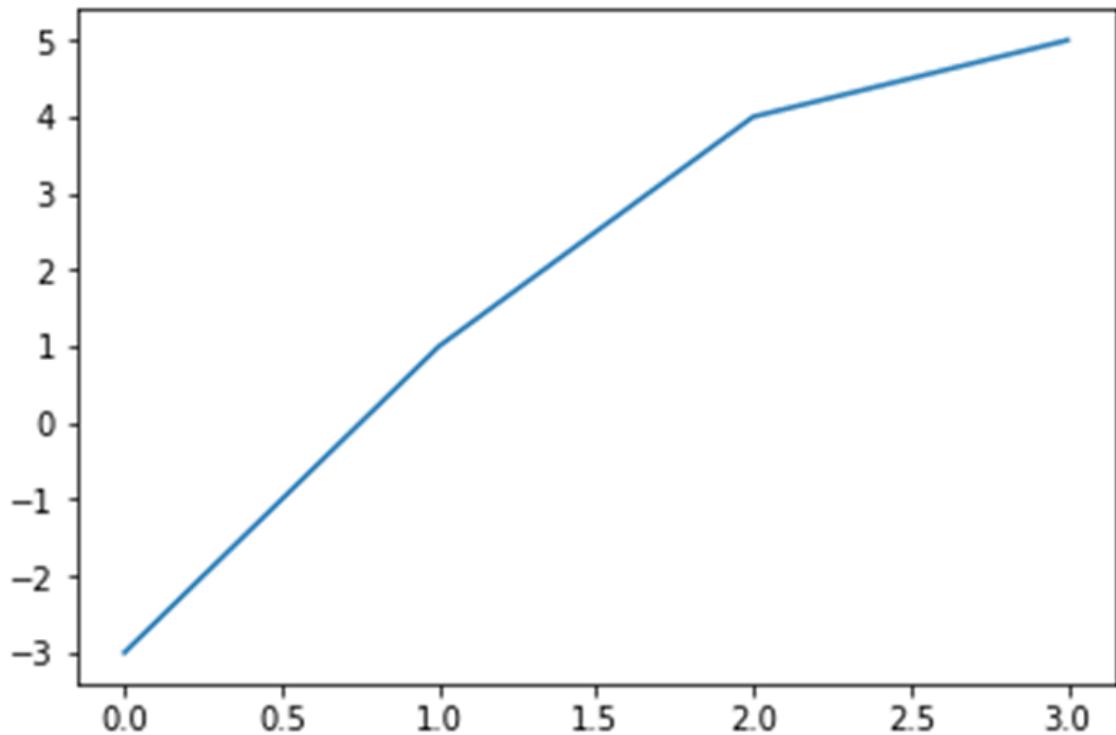


Figura 41. Representación omitiendo valores.

Fuente: elaboración propia.

Se puede omitir la lista de valores para el eje x, y. En este caso se usa como valores, por defecto, la lista de valores que van desde el 0 (primer valor) hasta el n-1, donde n es el número de valores proporcionados para el eje y. En el siguiente ejemplo se proporcionan únicamente valores para el eje y, por lo que se representa la secuencia de pares (0,-3), (1,1), (2,4), (3,5) (figura 41), notebook “Ejemplo_41.ipynb”.

Anotación:

En los ejemplos anteriores, se ha usado el comando `show()`, que sirve para abrir la ventana que contiene la imagen generada.

CONTINUAR

```
%matplotlib inline
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi**2 for xi in x])
plt.show()
```

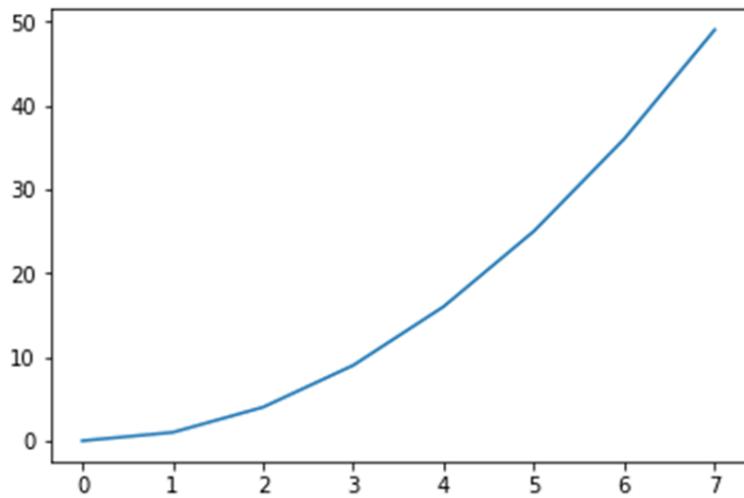


Figura 42. Representación usando range().

Fuente: elaboración propia.

Para representar datos, puede ser útil usar funciones que generen secuencias como *range(i,j,k)* o *arange (i, j, k)*. No obstante, la función *plot()* admite como datos de entrada tanto listas y tuplas Python como *numpy.arrays*. También admite el uso de *dataframes* de Pandas, que se verá después; pero los resultados pueden no ser los esperados y es recomendable la conversión de los *dataframes* en arrays de NumPy (figura 42), notebook “Ejemplo_42.ipynb”:

```
%matplotlib inline
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi**2 for xi in x])
plt.plot(x, [xi**3 for xi in x])
plt.plot(x, [xi**4 for xi in x])
plt.show()
```

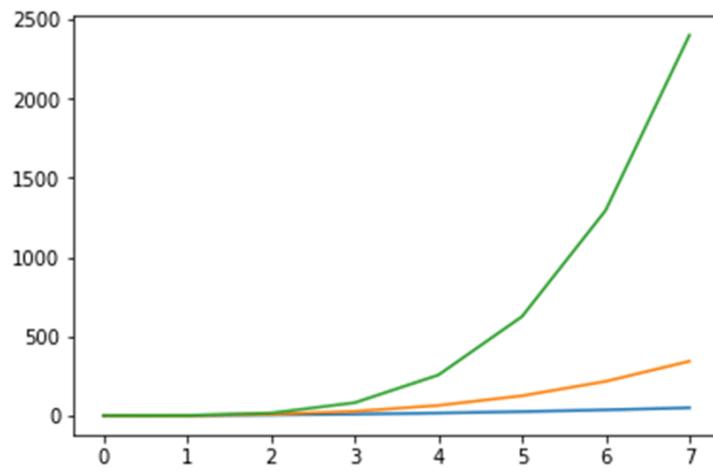


Figura 43. Representación de varias funciones en una misma gráfica.

Fuente: elaboración propia.

En algunas ocasiones, puede ser interesante mostrar varias representaciones sobre un mismo gráfico. Esto se puede realizar con el comando **plot()**, de formas distintas. La manera más fácil de representarlo es invocar **plot()** para que dibuje cada una de las funciones y, en último lugar, invocar el comando **show()** (figura 43), notebook “Ejemplo_43.ipynb”.

```
%matplotlib inline
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi**2 for xi in x], x, [xi**3 for xi in x], x, [xi**4 for xi in x])
plt.show()
```

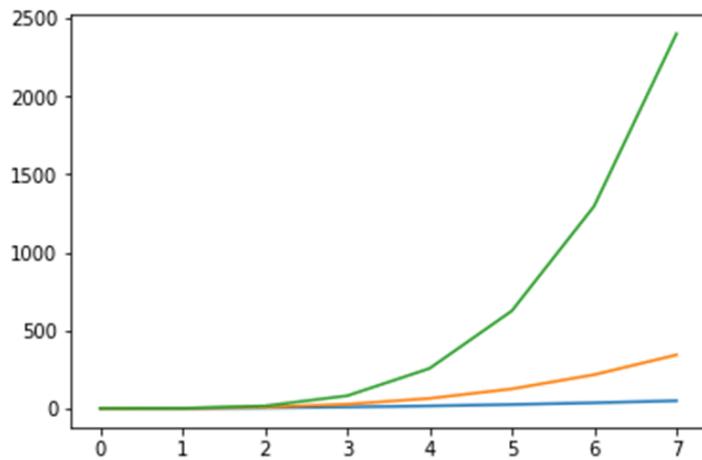


Figura 44. Representación de varias funciones en una misma gráfica.

Fuente: elaboración propia.

Otra forma de realizar la misma operación es incluir todas las listas y sus transformaciones en la misma función *plot*, de forma que queda una representación similar a las dimensiones de un array de NumPy (figura 44), notebook “Ejemplo_44.ipynb”:

Saber más

Obsérvese que, al superponer diferentes representaciones en la misma gráfica, Matplotlib utiliza automáticamente diferentes colores para cada representación.

CONTINUAR

A continuación, se va a mostrar **cómo añadir algunos elementos decorativos** que suelen aparecer en un gráfico:

Ejes de coordenadas

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(8.0)
plt.plot(x, [xi**2 for xi in x])
plt.axis()
```

(-0.3500000000000003, 7.35, -2.45, 51.45)

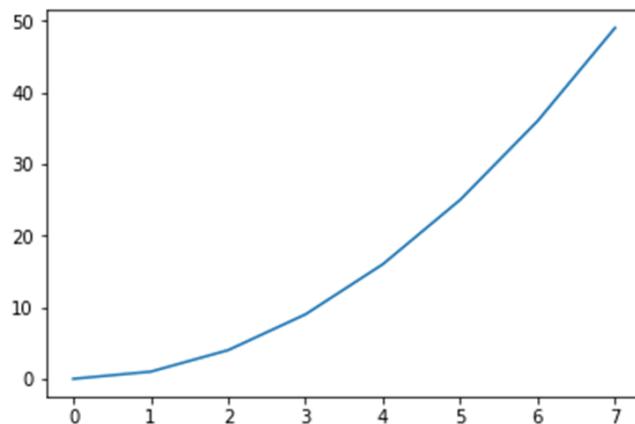


Figura 45. Función `axis()`.

Fuente: elaboración propia.

Por defecto, se establecen los valores que aparecen en los ejes de coordenadas, de manera que puedan mostrarse en la gráfica los puntos dibujados. Para configurar estos valores se usa la función `axis()`. Cuando se ejecuta sin parámetros, devuelve la escala actual utilizada en los ejes en forma de una tupla de cuatro valores que indican el límite inferior y superior de la escala de valores usada para el eje x y para el eje y, respectivamente (figura 45), *notebook* “Ejemplo_45.ipynb”.


```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(8.0)
plt.plot(x, [xi**2 for xi in x])
plt.axis([1,8,-3,8])
plt.show()
```

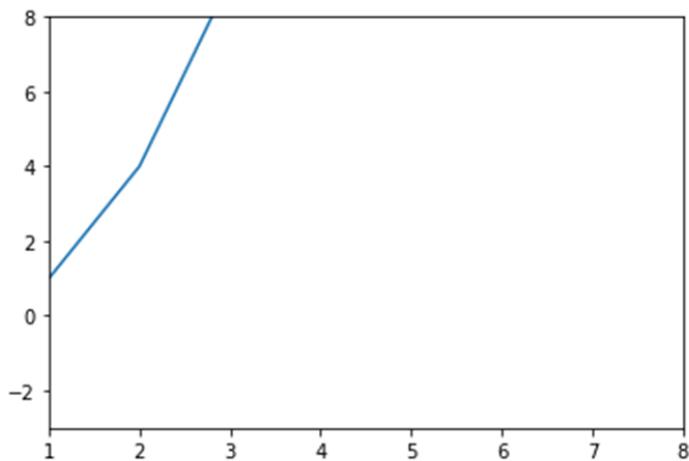


Figura 46. Cambio de escala.

Fuente: elaboración propia.

Para cambiar las escalas usadas, se invoca la función `axis()` mediante una lista de cuatro valores que representan el valor mínimo del eje x, el valor máximo del eje x, el valor mínimo del eje y, y el valor máximo del eje y (figura 46), *notebook* “Ejemplo_46.ipynb”.

```

1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import numpy as np
4 x = np.arange(8)
5 plt.plot(x, [xi**2 for xi in x])
6 plt.xticks(range(8),['a','b','c','d','e','f','g','h'])
7 plt.yticks(range(0,16,2))
8 plt.show()

```

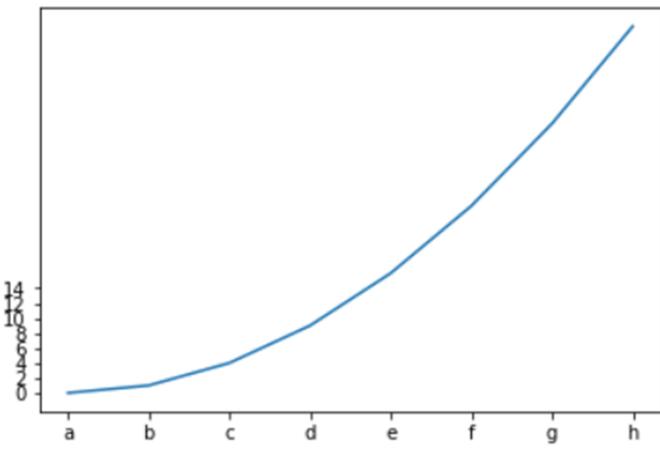


Figura 47. Valores de los ejes.

Fuente: elaboración propia.

La función `axis()` se puede invocar para configurar solo uno de los ejes. Para ello, habrá que proporcionarle los valores correspondientes de la escala utilizando la notación de parámetros argumentales. Por ejemplo, si se quiere modificar solo el eje y, de manera que varíe entre 4 y 10, se invocaría de la siguiente forma: `plt.axis(ymin=4,ymax=10)`.

Existen dos funciones, `xlim()` e `ylim()`, que permiten controlar las escalas de los ejes x e y respectivamente, de manera independiente. Se invocan con una lista de dos valores que representan respectivamente el límite inferior y superior de la escala de un eje.

Para gestionar los valores que se colocan sobre los ejes y la localización de ellas, se usan las funciones `plt.xticks()` y `plt.yticks()`, que permiten manipular estos elementos en los ejes x e y respectivamente. En ambos casos, toman como parámetros dos listas de la misma longitud que contienen las localizaciones de los valores en los ejes y, por otra parte, los valores que se desean colocar en esas posiciones. Esta última lista se puede omitir, de manera que se tomarían como valores los usados en las posiciones (figura 47), notebook "Ejemplo_47.ipynb".

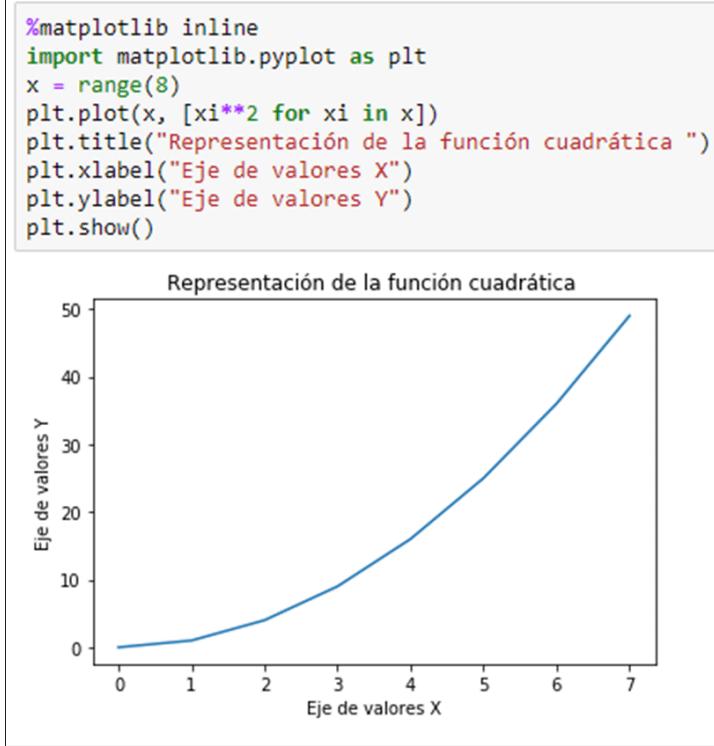


Figura 48. Gráfica con título y valores en los ejes.

Fuente: elaboración propia.

Para añadir etiquetas a los ejes x e y, se usan las funciones `plt.xlabel()` y `plt.ylabel()` respectivamente, que toman como parámetros los valores de las etiquetas que se quieran añadir a los ejes. Para añadir un título se utiliza la función `plt.title()`, que toma como parámetro el valor del título (figura 48), *notebook* “Ejemplo_48.ipynb”.

Cuadrícula

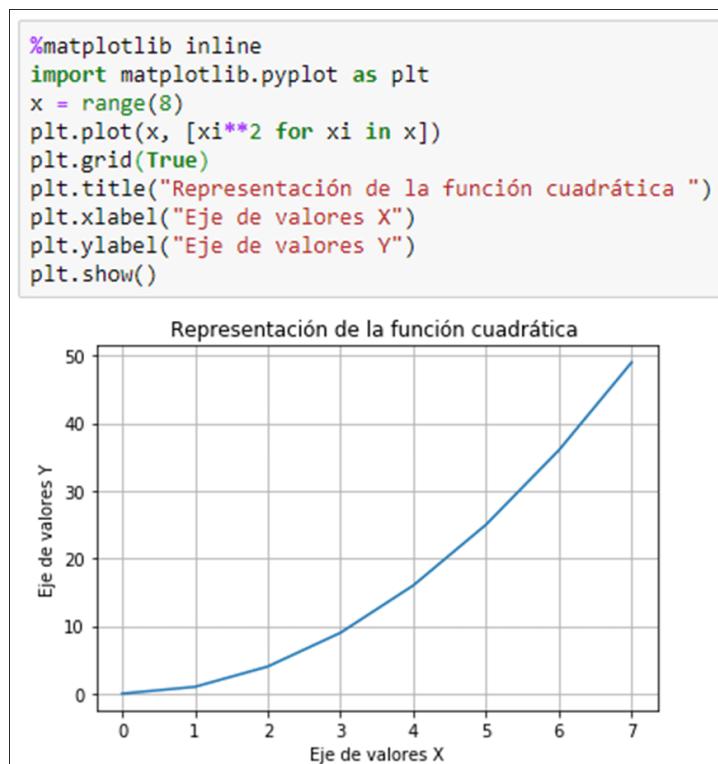


Figura 49. Gráfica con cuadrícula.

Fuente: elaboración propia.

Para añadir una cuadrícula a una gráfica, basta con usar la función `plt.grid()`, que toma como parámetro el valor `True` (habilitar la cuadrícula) o `False` (deshabilitar la cuadrícula). En la figura 49 aparece representada, *notebook* “Ejemplo_49.ipynb”:

Leyenda



Figura 50. Leyenda.

Fuente: elaboración propia.

Para añadir una leyenda a la representación de un conjunto de puntos, se invoca la función `plt.plot()`, que dibuja la secuencia con el argumento `label`, al cual se le asigna el valor que debe tomar la leyenda —esta asignación debe aparecer a continuación de los parámetros que aparecen sin formato de argumentos—. Por último, se invoca la función `plt.legend()` sin argumentos para que dibuje las leyendas (figura 50), *notebook* “Ejemplo_50.ipynb”:

Configurar el formato

Para configurar el formato para cada par de puntos x,y , se añade un tercer argumento adicional a la función `plt.plot()`. El formato es una cadena que contiene un código de tres o cuatro caracteres, que indican tres aspectos del formato: color, aspecto de la conexión entre puntos y aspecto de cada punto (tabla 1).

Código	Color
b	blue
c	cyan
g	green
k	black
m	magenta
r	red
w	white
y	yellow
Código	Tipo de línea
-	Línea sólida
--	Línea discontinua
-.	Línea discontinua con puntos
:	Línea de puntos
Código	Forma del punto
.	Punto
,	Píxel
O	Círculo
V	Triángulo hacia abajo
^	Triángulo hacia arriba
<	Triángulo hacia la izquierda
>	Triángulo hacia la derecha
1	Trípode hacia abajo
2	Trípode hacia arriba
3	Trípode hacia la izquierda
4	Trípode hacia la derecha

s	Cuadrado
p	Pentágono
*	Estrella
h	Hexágono
H	Hexágono rotado
+	Signo +
x	Cruz
D	Diamante
d	Diamante delgado
	Línea vertical
-	Línea horizontal

Tabla 1. Formato de los puntos representados.

Fuente: elaboración propia.

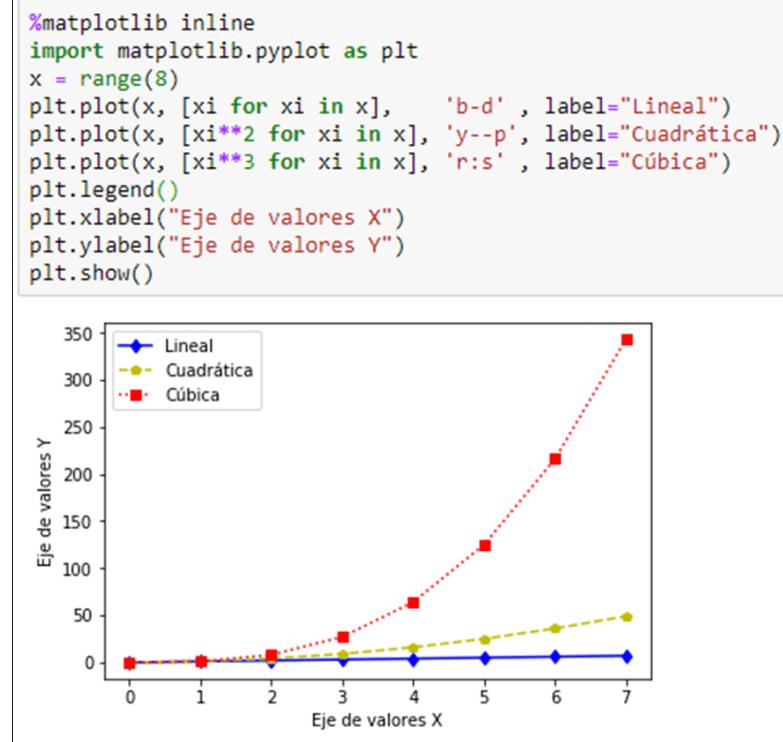


Figura 51. Formateado de varios puntos.

Fuente: elaboración propia.

En el ejemplo (figura 51), notebook “Ejemplo_51.ipynb”, se muestra un formateado de puntos.

Aparte del formato, existen otras propiedades de las líneas bidimensionales (`matplotlib.lines.Line2D`) que se pueden modificar. La lista completa se puede consultar en la documentación oficial de Matplotlib: <https://matplotlib.org/stable/tutorials/introductory/pyplot.html#controlling-line-properties>

Añadir texto

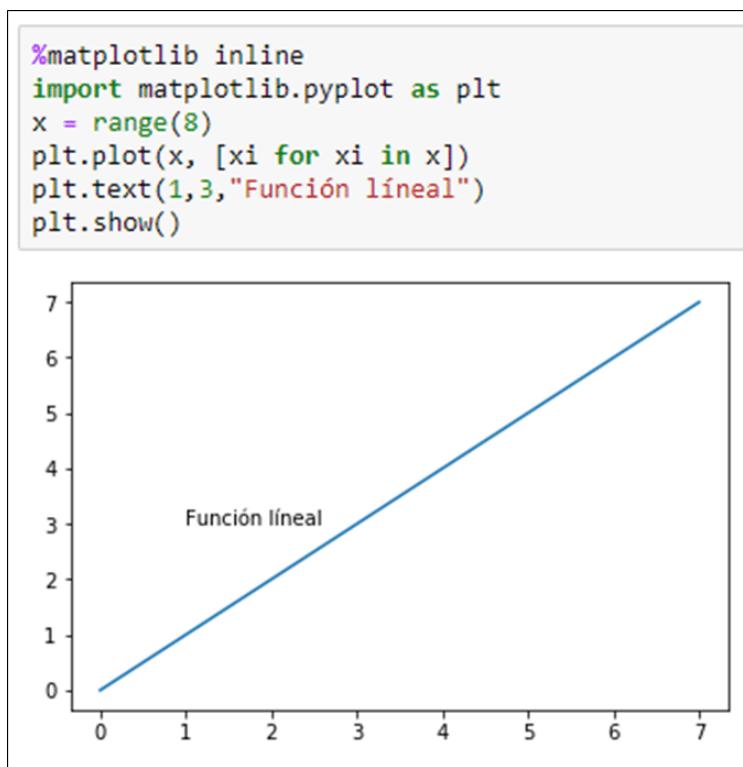


Figura 52. Texto sobre la gráfica.

Fuente: elaboración propia.

Para añadir texto, se puede usar la función `plt.text()`, que toma como parámetros un par de puntos que indican la posición donde se quiere escribir y la cadena que se quiere escribir. La posición es relativa a los ejes utilizados para representar los datos (figura 52), *notebook* “Ejemplo_52.ipynb”.

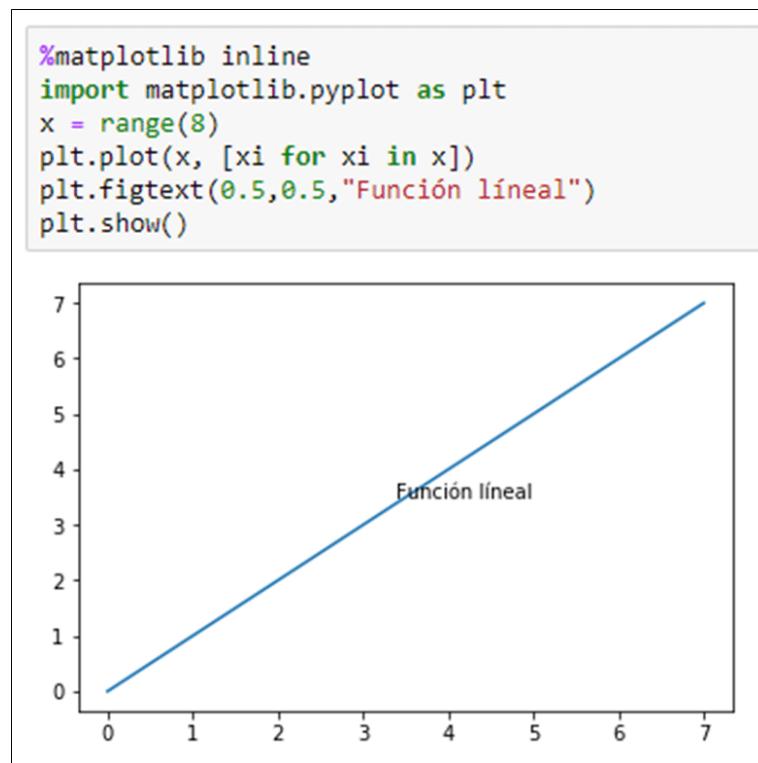


Figura 53. Situando texto sobre la gráfica.

Fuente: elaboración propia.

También es posible añadir texto a un dibujo con la función **plt.figtext()**, expresando las coordenadas de forma relativa a la figura dibujada. Las coordenadas varían entre 0 y 1 —la posición (0,0) representa la esquina inferior izquierda y (1,1) la esquina superior derecha—. Se muestra en la figura 53, notebook “Ejemplo_53.ipynb”.

Añadir anotaciones

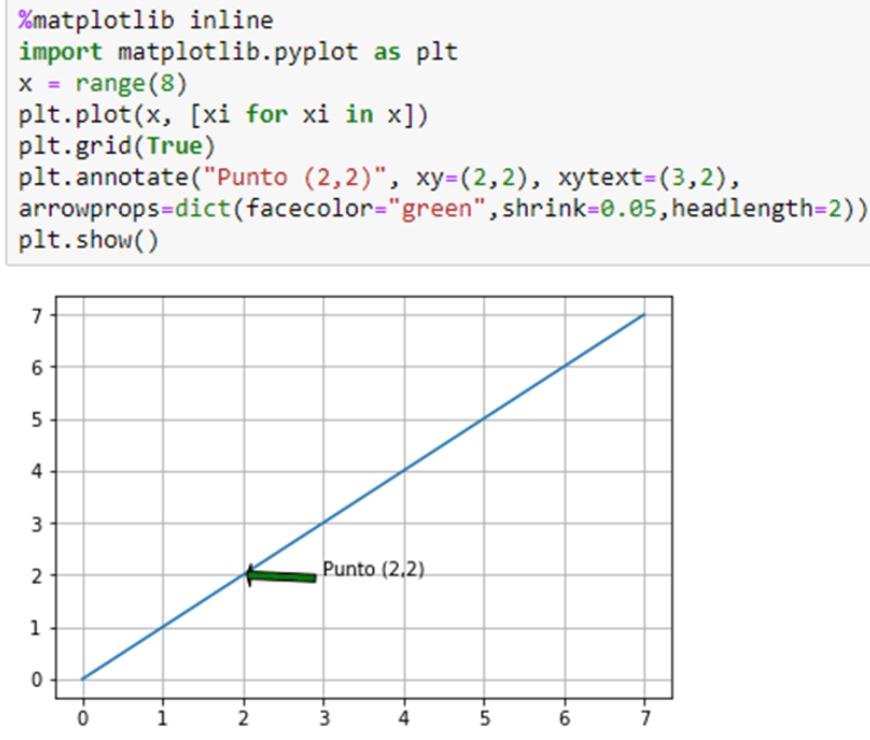


Figura 54. Anotación sobre la gráfica.

Fuente: elaboración propia.

Para añadir una anotación, se usa la función `plt.annotate()`, la cual toma como argumentos (figura 54), notebook “Ejemplo_54.ipynb”:

- Texto de la anotación.
- Argumento **xy**: posición de la función en la que se quiere añadir la anotación expresada en coordenadas respecto a los ejes.
- Argumento **xytext**: indica la posición de la anotación que se quiere añadir expresada en coordenadas respecto a los ejes.
- Argumento **arrowprops**: propiedades de la flecha que une la anotación con el punto anotado expresado como un diccionario con las claves **width** (ancho de la flecha), **headlength** (longitud de la flecha reservada a la cabeza de esta), **headwidth** (ancho de la base de la flecha), **facecolor** (color de la superficie de la flecha), **shrink** (desplazamiento de la flecha)

con respecto al punto anotado y la anotación expresado como un porcentaje).

CONTINUAR

A continuación, se van a revisar algunos de los tipos de gráficos más usados con Matplotlib:

Histogramas

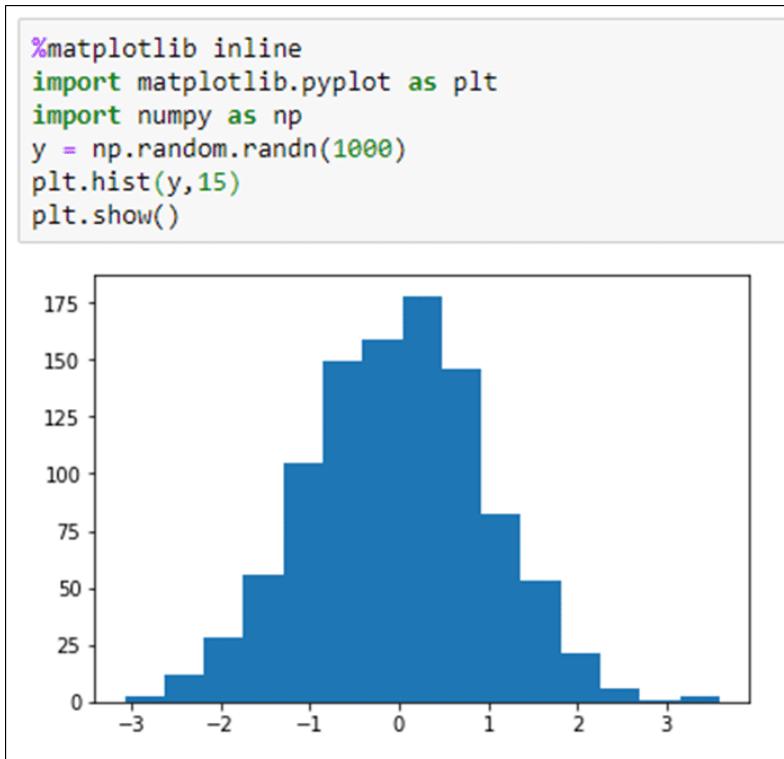


Figura 55. Histograma.

Fuente: elaboración propia.

Los histogramas son un tipo de gráficos que permiten visualizar las frecuencias de aparición de un conjunto de datos en forma de barras. La superficie de cada barra define una categoría que es proporcional a la frecuencia de los valores representados.¹

Para dibujar un histograma, se usa la función `plt.hist()`, que toma como argumentos los valores que se van a considerar para crear las categorías y el número de categorías que se quieren considerar. Si no se indica ningún valor para el número de categorías, toma por defecto el valor 10 (figura 55), notebook “Ejemplo_55.ipynb”:

¹[Histograms](#).

Diagramas de barras

```
%matplotlib inline
import matplotlib.pyplot as plt
left=range(5)
p1 = plt.bar(left, height=[20,35,30,35,27], width=0.4, color='green', yerr=[2,3,4,1,2])
p2 = plt.bar(left, height=[25,32,34,20,25], width=0.4, color='red', bottom=[20,35,30,35,27], yerr=[3,5,2,3,3])
plt.legend(["Secuencia 1","Secuencia 2"])
plt.xticks(range(5),["A","B","C","D","E"])
plt.show()
```

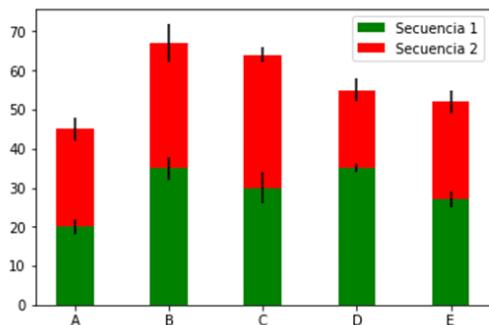


Figura 56. Diagrama de barras verticales.

Fuente: elaboración propia.

Los diagramas de barras son un tipo de gráficos que representan los datos como barras rectangulares, horizontales o verticales, cuyas dimensiones son proporcionales a los valores que representan los datos. Este tipo de gráficos permite comparar dos o más valores.²

Para dibujar un diagrama de barras se utiliza la función `plt.bar()`, que tiene como parámetros (figura 56), notebook “Ejemplo_56.ipynb”:

- Argumento **left**. Es una lista de las coordenadas del eje x donde quedará situada gráficamente la esquina izquierda de una barra.
- Argumento **height**. Es una lista con la altura de las barras.
- Argumento **width**. Representa el ancho de la barra, que por defecto es 0,8.
- Argumento **color**. Representa el color de las barras.
- Argumentos **xerr**, **yerr**. Representa barras de error sobre el diagrama de barras.

- Argumento **bottom**. Representa las coordenadas inferiores en el eje y de las barras, en el caso de que se quiera empezar a dibujar por encima del eje x.

[2Bar Diagrams.](#)

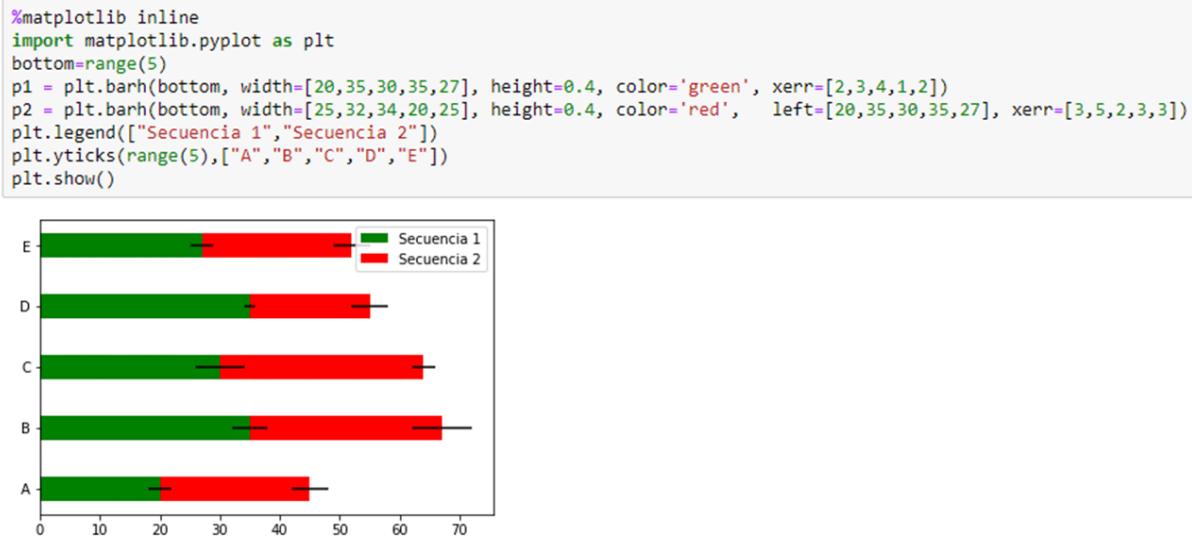


Figura 57. Diagrama de barras horizontal.

Fuente: elaboración propia.

Los mismos datos se podrían haber representado horizontalmente utilizando la función `plt.barh()`. Esta función tiene parámetros similares a la anterior (figura 57), notebook “Ejemplo_57.ipynb”:

- Argumento **bottom**. Es una lista de las coordenadas del eje y que indican dónde se situará gráficamente cada una de las barras del gráfico.
- Argumento **width**. Es una lista con la altura de las barras.
- Argumento **height**. Representa la altura de la barra, que por defecto es 0,8.
- Argumento **color**. Representa el color de las barras.
- Argumentos **xerr, yerr**. Representa las barras de error sobre el diagrama de barras.
- Argumento **left**. Representa las coordenadas inferiores en el eje x de las barras, en el caso de que se quiera empezar a dibujar por encima del eje y.

Diagramas circulares

```
%matplotlib inline
import matplotlib.pyplot as plt
x = [4,7,5,3,9,15]
labels = ['Madrid','Barcelona','Valencia','Zaragoza','Sevilla','Granada']
colors = ['yellow','blue','red','green','brown','orange']
explode = [0.2,0.2,0,0,0.1,0.1]
plt.pie(x,labels=labels,labeldistance=1.3,explode=explode, autopct='%1.1f%%',
         colors=colors, pctdistance=0.5, shadow=True);
plt.show()
```

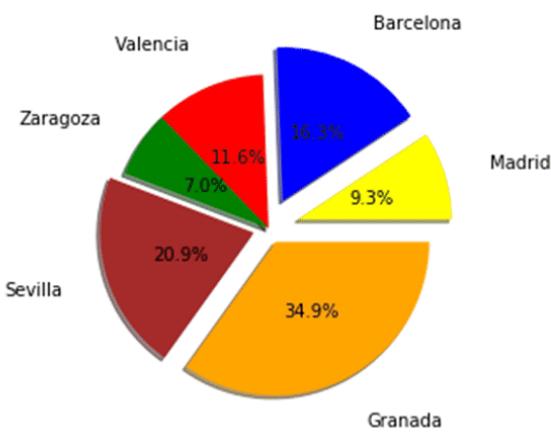


Figura 58. Diagrama circular.

Fuente: elaboración propia.

Se trata de un gráfico en forma de círculo que se encuentra dividido en sectores, de manera que la superficie ocupada por el sector representa proporcionalmente la cantidad descrita.³

Para representar gráficos circulares se utiliza la función `plt.pie()`, que toma como entrada un array X con valores. Así, para cada valor x del array X, la función genera sectores que son proporcionales a $x/Suma(X)$. Si la suma fuera menor que 1, se representan los valores directamente. Los sectores se dibujan empezando desde el eje horizontal, en el lado derecho y en sentido contrario a las manecillas del reloj. La función tiene los siguientes parámetros (figura 58), notebook “Ejemplo_58.ipynb”:

- **explode:** es un array de la misma longitud que el array de valores X, cuyos valores indican la fracción del radio que va a separar el sector del centro del círculo.

- **colors**: es la lista de colores que se usarán cíclicamente para los sectores. En caso de no indicarlos, sigue una progresión automática de azul, verde, rojo, cian, magenta...
- **labels**: es una lista de las etiquetas asociadas a cada sector.
- **labeldistance**: es la distancia respecto al centro del gráfico en la que se dibuja una etiqueta.
- **autopct**: esta función o cadena de formateo permite etiquetar los sectores con los valores numéricos que representan.
- **pctdistance**: es la distancia respecto al centro del gráfico donde se dibujan los valores numéricos.
- **shadow**: dibuja un sombreado en los sectores y el gráfico.

³[Pie Charts](#).

Diagramas de dispersión

Cadena	Código
o	Círculo
V	Triángulo hacia abajo
^	Triángulo hacia arriba
<	Triángulo hacia la izquierda
>	Triángulo hacia la derecha
s	Cuadrado
p	Pentágono
h	Hexágono
+	Signo +
x	Cruz
d	Diamante
8	Octógono

Tabla 2. Tabla de configuración.

Fuente: elaboración propia.

Un diagrama de dispersión dibuja los valores de dos conjuntos de datos como una colección de puntos desconectados. Las coordenadas de cada punto se obtienen de cada conjunto (la coordenada x del primer conjunto y la coordenada y del segundo conjunto). Este tipo de dibujos facilita el descubrimiento de correlaciones u otro tipo de relaciones entre los puntos considerados.

Para representar diagramas de dispersión se utiliza la función `plt.scatter()`, que toma como entrada dos arrays unidimensionales de la misma longitud. Los principales argumentos que admite la función son:⁴

- **s:** establece el tamaño de los puntos en pixel*pixel. Se puede indicar un único valor, en cuyo caso se aplicará a todos los puntos, o bien especificar un array de la misma longitud que los arrays de entrada. En este último caso, se especifica un tamaño particular para cada punto.

- **c**: color de los puntos. Se puede indicar un único valor, en cuyo caso se aplicará a todos los puntos, o bien especificar un array de la misma longitud que los arrays de entrada. En este último caso, se especifica un color particular para cada punto. Para especificar los colores se pueden usar los códigos de colores.
- **marker**: especifica el tipo de punto que se va a dibujar de acuerdo con la tabla 2.

⁴[Scatter Diagrams](#).

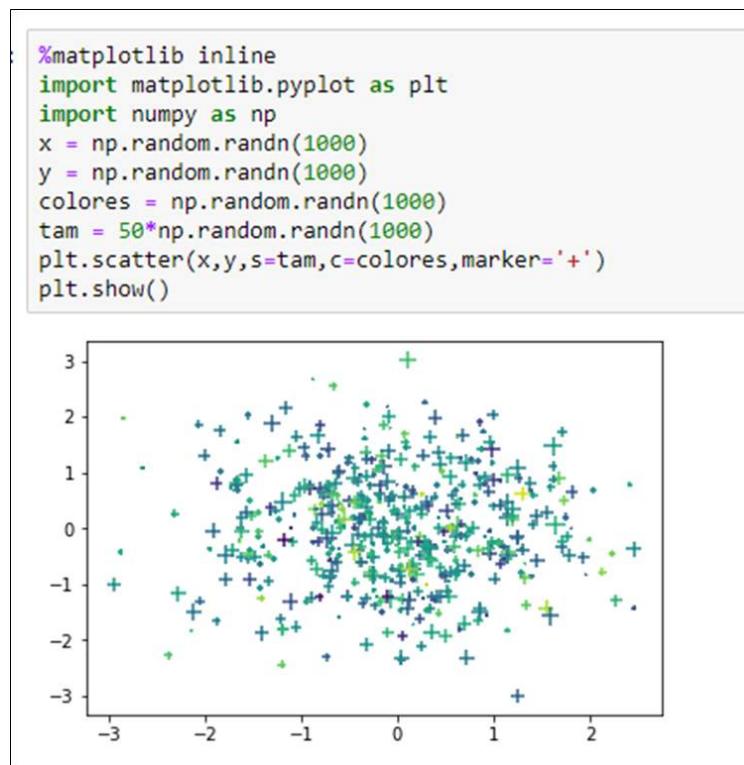


Figura 59. Diagrama de dispersión.

Fuente: elaboración propia.

En la figura 59, notebook “Ejemplo_59.ipynb”, se muestra un ejemplo de diagrama de dispersión.

Diagramas de contornos

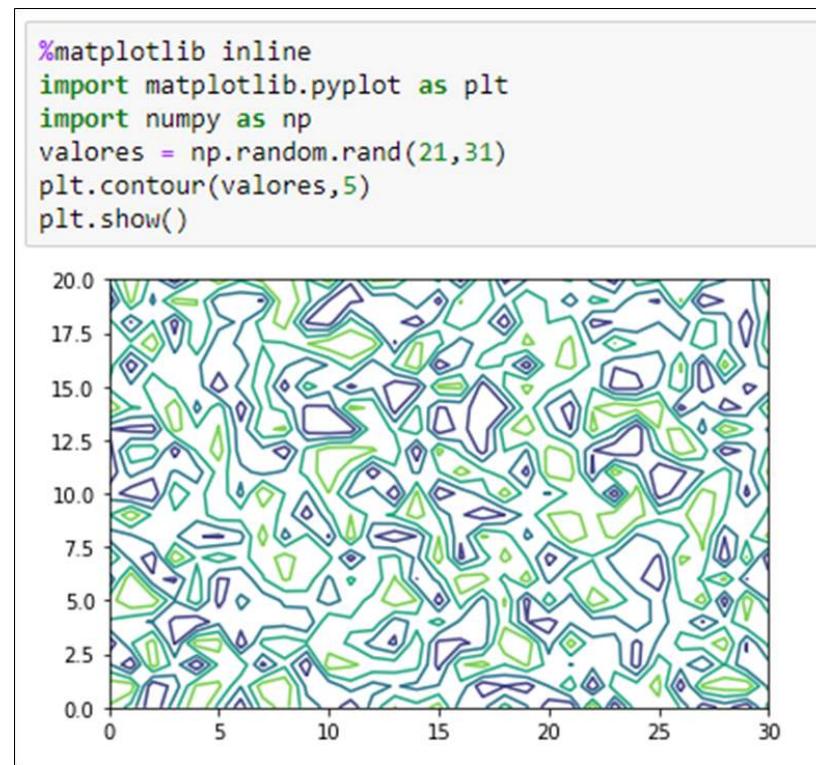


Figura 60. Diagrama de contorno.

Fuente: elaboración propia.

Las líneas de contorno para funciones de dos variables son curvas en las que la función toma un valor constante, es decir, $f(x,y)=C$, siendo C una constante. La densidad de las líneas indica la pendiente de la función.

Para dibujar un diagrama de contornos, existen las funciones `plt.contour()`,⁵ que toman como entrada un array de dos dimensiones y, opcionalmente, el número niveles de líneas de contorno (figura 60), notebook “Ejemplo_60.ipynb”. Las líneas se representan en diferentes colores, desde los valores más pequeños a los más grandes –desde el azul oscuro para áreas de bajo volumen hasta el rojo para áreas de alto volumen–.

⁵[Contour Diagrams](#).

Diagrama de caja

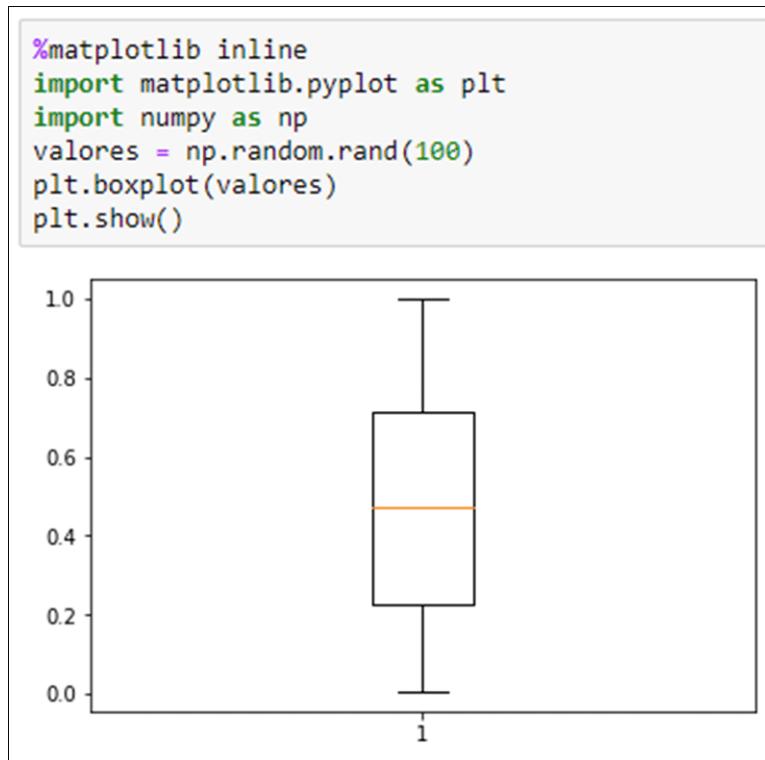


Figura 61. Diagrama de caja.

Fuente: elaboración propia.

Un diagrama de caja es un gráfico que representa los cuartiles de un conjunto de datos. Este tipo de diagramas se construyen mediante la función `plt.boxplot`⁶ (figura 61), notebook “Ejemplo_61.ipynb”:

⁶[Box Plots.](#)

Saber más

Se pueden consultar otros tipos de gráficos, con ejemplos, en la [web oficial de Matplotlib](#).

CONTINUAR

En Matplotlib existe la posibilidad de crear gráficas formadas por un conjunto de gráficos. Estos gráficos se denominan subgráficos. Para ello, se utiliza la función `plt.figure()`, la cual genera un objeto de tipo **Figure**, que es un contenedor donde se pueden añadir subgráficos utilizando el método `add_subplot()`. La invocación del método devuelve un objeto de tipo **axe**, que es un área donde se puede dibujar un gráfico independiente. Los parámetros del método permiten especificar de manera matricial dónde se insertarán los subgráficos:

- **numrows**: especifica el número de filas.
- **numcols**: especifica el número de columnas.
- **fignum**: especifica un número que varía entre 1 y `numrows*numcols`, que indica el subgráfico actual, avanzando de izquierda a derecha y de arriba abajo. Así, 1 indica la esquina superior izquierda y `numrows*numcols` indica la esquina inferior derecha.



Los objetos de tipo **axe** pueden invocar la función `plt.plot()` para representar puntos (figura 62), *notebook* “Ejemplo_62.ipynb”.

```
%matplotlib inline
import matplotlib.pyplot as plt
fig = plt.figure()
x = range (8)
ax1=fig.add_subplot(211)
ax1.plot(x, [xi for xi in x])
ax2=fig.add_subplot(212)
ax2.plot(x, [xi**2 for xi in x])
plt.show()
```

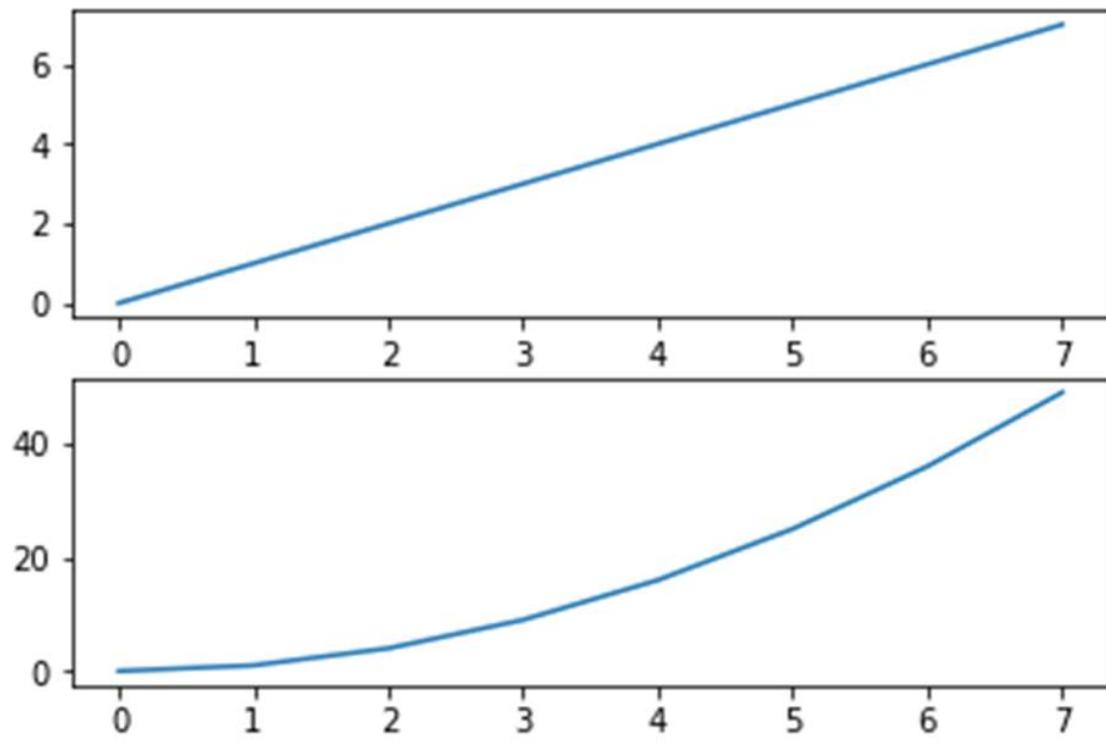


Figura 62. Ejemplo de subgráfico.

Fuente: elaboración propia.

V. Manipulación y análisis de datos con Pandas

Pandas es una librería construida sobre NumPy que ofrece estructuras de datos de alto nivel que facilitan el análisis de datos desde Python. Se van a estudiar dos estructuras de datos:

- Series.
- *Dataframes*.



Antes de trabajar con estas estructuras, se importan las librerías necesarias:

```
import numpy as np  
import pandas as pd  
from pandas import *
```

CONTINUAR

1. Series

Una serie es un array unidimensional que ofrece la posibilidad de indexar sus elementos a partir de una lista de etiquetas customizable.⁷

⁷[Pandas Series.](#)

Las series pueden construirse a partir de arrays, diccionarios, escalares, así como cualquier otro objeto iterable de Python:

ndarray

```
import numpy as np
import pandas as pd
from pandas import *
s=Series([1,2,3,4,5], index=['a','b','c','d','e'])
s

a    1
b    2
c    3
d    4
e    5
dtype: int64
```

Figura 63. Ejemplo de ndarray.

Fuente: elaboración propia.

Si el array de datos en un *ndarray*, entonces el índice debe ser de la misma longitud que el array de datos (figura 63), notebook “Ejemplo_63.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
s=Series([1,2,3,4,5])
s
```

0	1
1	2
2	3
3	4
4	5

dtype: int64

Figura 64. Ejemplo de ndarray sin índice.

Fuente: elaboración propia.

Si ningún índice es pasado, entonces se crea uno formado por valores que van desde [0, ..., len(data)-1] (figura 64), notebook “Ejemplo_64.ipynb”:

Diccionario

```
import numpy as np
import pandas as pd
from pandas import *
d={'a':0.,'b':1.,'c':2.}
s=Series(d, index=['b','c','d','a'])
s

b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Figura 65. Creación de un ndarray usando un diccionario.

Fuente: elaboración propia.

Si se pasa un índice, los valores del diccionario se asocian a los valores del índice (figura 65), notebook “Ejemplo_65.ipynb”:

```
import numpy as np
import pandas as pd
from pandas import *
d={'a':0.,'b':1.,'c':2.}
s=Series(d)
s
```



```
a    0.0
b    1.0
c    2.0
dtype: float64
```

Figura 66. Creación de ndarray usando un diccionario sin índice.

Fuente: elaboración propia.

Si no se proporciona un índice, entonces se construye a partir de las claves ordenadas del diccionario (figura 66), notebook “Ejemplo_66.ipynb”:

Valor escalar

```
import numpy as np
import pandas as pd
from pandas import *
s=Series(5., index=['a','b','c','d','e'])
s
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Figura 67. Creación de ndarray, valor escalar.

Fuente: elaboración propia.

En este caso, se debe proporcionar un índice y el valor será repetido tantas veces como la longitud del índice (figura 67), notebook “Ejemplo_67.ipynb”.

Algunas características de las series son:



Las series actúan de forma similar a *ndarray* y son un argumento válido de funciones de NumPy (figura 68), *notebook* “Ejemplo_68.ipynb”:

```
import numpy as np
import pandas as pd
from pandas import *
s=Series([1,2,3,4,5], index=['a','b','c','d','e'])
s[0]
```

1

```
s[s > s.median()]
```

```
d    4
e    5
dtype: int64
```

```
s[[4,3,1]]
```

```
e    5
d    4
b    2
dtype: int64
```

Figura 68. Uso de *ndarray* con NumPy.

Fuente: elaboración propia.



Las series actúan como un diccionario en el que se pueden gestionar los valores a través de los índices (figura 69), *notebook* “Ejemplo_69.ipynb”:

```
import numpy as np
import pandas as pd
from pandas import *
s=Series([1,2,3,4,5], index=['a','b','c','d','e'])
s['a']
```

```
1
```

```
s['e']
```

```
5
```

```
s
```

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

```
'e' in s
```

```
True
```

Figura 69. Uso de series como diccionarios.

Fuente: elaboración propia.



Sobre las series se pueden realizar operaciones vectoriales (figura 70), notebook “Ejemplo_70.ipynb”:

```
import numpy as np
import pandas as pd
from pandas import *
s=Series([1,2,3,4,5], index=['a','b','c','d','e'])
s+s
```

```
a    2
b    4
c    6
d    8
e   10
dtype: int64
```

Figura 70. Operaciones vectoriales sobre series.

Fuente: elaboración propia.

- La principal diferencia entre series de Pandas y *ndarrays* de NumPy es que las operaciones entre series alinean automáticamente los datos basados en las etiquetas, de forma que pueden realizarse cálculos sin tener en cuenta si las series sobre las que se operan tienen las mismas etiquetas.
- Obsérvese que el resultado de una operación entre series no alineadas será la unión de los índices. Si una etiqueta no se encuentra en una de las series, entonces se le asocia el valor nulo. Los datos y el índice de una serie tienen un atributo *name* que puede asociarle un nombre (figura 71), *notebook* “Ejemplo_71.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
s=Series([1,2,3,4,5], index=['a','b','c','d','e'])
s.name = "datos"
s
```



```
a    1
b    2
c    3
d    4
e    5
Name: datos, dtype: int64
```

Figura 71. Uso del atributo name.

Fuente: elaboración propia.

CONTINUAR

2. Dataframes

Un *dataframe* es una estructura que contiene una colección ordenada de columnas, cada una de las cuales puede tener valores de diferentes tipos. Está formado por datos, opcionalmente un índice (etiquetas de las filas) y un conjunto de columnas (etiquetas de las columnas). En caso de no existir un índice, se genera a partir de los datos.

Los datos de un *dataframe* pueden proceder de:

Diccionario de ndarrays

```
import numpy as np
import pandas as pd
from pandas import *
d = {'one':[1.,2.,3.,4.], 'two':[4.,3.,2.,1.]}
DataFrame(d)
```

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

Figura 72. Diccionario de ndarrays.

Fuente: elaboración propia.

Los arrays (figura 72), notebook “Ejemplo_72.ipynb”, deben ser de la misma longitud. En caso de existir un índice, este debe ser de la misma longitud que los arrays y, en caso de no existir, se genera como índice la secuencia de números $0 \dots \text{longitud}(\text{array})-1$.

Lista de diccionarios

```
import numpy as np
import pandas as pd
from pandas import *
data2 = [{‘a’:1,’b’:2},{‘a’:5,’b’:10,’c’:20}]
DataFrame(data2)
```

	a	b	c
0	1	2	NaN
1	5	10	20.0

```
DataFrame(data2, index=[‘first’,’second’])
```

	a	b	c
first	1	2	NaN
second	5	10	20.0

Figura 73. Lista de diccionarios.

Fuente: elaboración propia.

notebook “Ejemplo_73.ipynb”.

Diccionario de tuplas

```
In [3]: import numpy as np
import pandas as pd
from pandas import *

DataFrame({
    ('a','a1'): {('A','B'):1, ('A','C'):2},
    ('a','a2'): {('A','C'):3, ('A','B'):4},
    ('a','a3'): {('A','B'):5, ('A','C'):6},
    ('b','b1'): {('A','C'):7, ('A','B'):8},
    ('b','b2'): {('A','D'):9, ('A','B'):10}
})
```

Out[3]:

		a		b		
		a1	a2	a3	b1	b2
A	B	1.0	4.0	5.0	8.0	10.0
C	2.0	3.0	6.0	7.0	NaN	
D	NaN	NaN	NaN	NaN	9.0	

Figura 74. Diccionario de tuplas.

Fuente: elaboración propia.

notebook “Ejemplo_74.ipynb”.

Diccionario de series

```

import numpy as np
import pandas as pd
from pandas import *
d = {'one': Series([1.,2.,3.], index=['a','b','c']) ,
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df

```

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0


```
DataFrame(d, index=['d','b','a'])
```

	one	two
b	2.0	2.0
a	1.0	1.0
d	NaN	4.0

Figura 75. Diccionario de series.

Fuente: elaboración propia.

notebook “Ejemplo_75.ipynb”. El índice que resulta es la unión de los índices de las series. En caso de existir diccionarios anidados, primero se convierten en diccionarios. Además, si no se pasan columnas, se toma como tal la lista ordenada de las claves de los diccionarios.

Puede accederse a las etiquetas de las columnas y de las filas mediante los atributos *índice* y *columnas*. Téngase en cuenta que, cuando un conjunto particular de columnas se pasa como argumento con el diccionario, las columnas sobrescriben en las claves del diccionario.

Array estructurado

```
import numpy as np
import pandas as pd
from pandas import *
data = np.zeros((2,), dtype=[
    ('A','i4'), ('B','f4'), ('C','a10'),
])
data[:] = [(1,2.,'Hello'),(2,3.,"World")]
DataFrame(data)
```

	A	B	C
0	1	2.0	b'Hello'
1	2	3.0	b'World'

Figura 76. Array estructurado.

Fuente: elaboración propia.

notebook “Ejemplo_76.ipynb”.

CONTINUAR

3. Operaciones sobre *dataframes*

Manipulación de un *dataframe*

```
In [3]: import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)

Out[3]:
   one  two
a    1.0  1.0
b    2.0  2.0
c    3.0  3.0
d    NaN  4.0

In [4]: df['one']
Out[4]: a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

In [5]: df['three'] = df['one'] * df['two']

In [6]: df['flag'] = df['one'] > 2

In [7]: df
Out[7]:
   one  two  three  flag
a    1.0  1.0    1.0  False
b    2.0  2.0    4.0  False
c    3.0  3.0    9.0   True
d    NaN  4.0    NaN  False
```

Figura 77. Manipulación de un dataframe.

Fuente: elaboración propia.

Un dataframe es como un diccionario de series indexado, por lo que se pueden usar las mismas operaciones utilizadas con los diccionarios (figura 77), notebook “Ejemplo_77.ipynb”:

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df['one']

a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

df['three'] = df['one'] * df['two']

df['flag'] = df['one'] > 2

del df['two']

three = df.pop('three')

df

   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

Figura 78. Borrado de columnas.

Fuente: elaboración propia.

o bien ejecutando el método *pop()* del *dataframe* (figura 78), notebook “Ejemplo_78.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df['three'] = df['one'] * df['two']
df['flag'] = df['one'] > 2
del df['two']
three = df.pop('three')
df['foo'] = 'bar'
df
```

	one	flag	foo
a	1.0	False	bar
b	2.0	False	bar
c	3.0	True	bar
d	NaN	False	bar

Figura 79. Propagación a toda la columna.

Fuente: elaboración propia.

Cuando se inserta un valor escalar, entonces se propaga a toda la columna (figura 79), notebook "Ejemplo_79.ipynb".

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df['three'] = df['one'] * df['two']
df['flag'] = df['one'] > 2
del df['two']
three = df.pop('three')
df['foo'] = 'bar'
df['one_trunc'] = df['one'][:2]
df
```

	one	flag	foo	one_trunc
a	1.0	False	bar	1.0
b	2.0	False	bar	2.0
c	3.0	True	bar	NaN
d	NaN	False	bar	NaN

Figura 80. Inserción de una serie con distinto índice.

Fuente: elaboración propia.

Cuando se inserta una serie que no tiene el mismo índice, se crea el índice para el *dataframe* (figura 80), notebook “Ejemplo_80.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df['three'] = df['one'] * df['two']
df['flag'] = df['one'] > 2
del df['two']
three = df.pop('three')
df['foo'] = 'bar'
df['one_trunc'] = df['one'][:2]
df.insert(1, 'bar', df['one'])
df
```

	one	bar	flag	foo	one_trunc
a	1.0	1.0	False	bar	1.0
b	2.0	2.0	False	bar	2.0
c	3.0	3.0	True	bar	NaN
d	NaN	NaN	False	bar	NaN

Figura 81. Inserción de un dataframe.

Fuente: elaboración propia.

Las columnas, por defecto, se insertan al final; sin embargo, se puede elegir el lugar de inserción usando la función *insert* (figura 81), notebook “Ejemplo_81.ipynb”.

CONTINUAR

Indexación/selección

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df['one']

a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

Figura 82. Selección por columnas.

Fuente: elaboración propia.

Se puede seleccionar **por columnas** (figura 82), notebook “Ejemplo_82.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
```

```
df = DataFrame(d)
df.loc['b']
```

```
one    2.0
two    2.0
Name: b, dtype: float64
```

Figura 83. Selección por etiqueta.

Fuente: elaboración propia.

Se puede seleccionar **por etiqueta**, mediante el atributo *loc* del *dataframe* (figura 83), *notebook* “Ejemplo_83.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.], index=['a','b','c']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df.iloc[2]
```

one 3.0
two 3.0
Name: c, dtype: float64

Figura 84. Selección por entero.

Fuente: elaboración propia.

Selección de fila **por entero**, mediante el atributo *iloc* (figura 84), notebook "Ejemplo_84.ipynb".

```
In [3]: import numpy as np
import pandas as pd
from pandas import *
d = { 'one': Series([1.,2.,3.,4.,5.,6.,7.], index=['a','b','c','d','e','f','g']),
      'two': Series([1.,2.,3.,4.], index=['a','b','c','d'])}
df = DataFrame(d)
df.iloc[3:6]
```

Out[3]:

	one	two
d	4.0	4.0
e	5.0	NaN
f	6.0	NaN

Figura 85. Selección por rangos.

Fuente: elaboración propia.

Selección **por rangos** (figura 85), notebook “Ejemplo_85.ipynb”.

CONTINUAR

Alineamiento y aritmética

```
In [23]: import numpy as np
import pandas as pd
from pandas import *
df = DataFrame(np.ndindex(3,2,1,1), columns=['A','B','C','D'])
df2 = DataFrame(np.ndindex(1,2,3), columns=['A','B','C'])
df
Out[23]:
   A  B  C  D
0  0  0  0  0
1  0  1  0  0
2  1  0  0  0
3  1  1  0  0
4  2  0  0  0
5  2  1  0  0

In [24]: df2
Out[24]:
   A  B  C
0  0  0  0
1  0  0  1
2  0  0  2
3  0  1  0
4  0  1  1
5  0  1  2

In [25]: df+df2
Out[25]:
   A  B  C    D
0  0  0  0  NaN
1  0  1  1  NaN
2  1  0  2  NaN
3  1  2  0  NaN
4  2  1  1  NaN
5  2  2  2  NaN
```

Figura 86. Alineamiento.

Fuente: elaboración propia.

Cuando se realiza una operación aritmética entre dos *dataframes*, la combinación se hace **a nivel de fila y columna**, de manera que los elementos de la misma columna de un *dataframe* se combinan con los de la misma columna en el otro *dataframe*, elemento a elemento filas (figura 86), notebook “Ejemplo_86.ipynb”.

```
In [8]: import numpy as np
import pandas as pd
from pandas import *
df = DataFrame(np.ndindex(3,2,1,1), columns=['A','B','C','D'])
df

Out[8]:
   A  B  C  D
0  0  0  0  0
1  0  1  0  0
2  1  0  0  0
3  1  1  0  0
4  2  0  0  0
5  2  1  0  0

In [9]: df=df.iloc[3]
df

Out[9]:
   A  B  C  D
0 -1 -1  0  0
1 -1  0  0  0
2  0 -1  0  0
3  0  0  0  0
4  1 -1  0  0
5  1  0  0  0
```

Figura 87. Alineamiento de dataframe y serie.

Fuente: elaboración propia.

Cuando se opera con *dataframe* y series, se alinea el índice de las series sobre las columnas del *dataframe* (figura 87), notebook “Ejemplo_87.ipynb”.

```
In [7]: import numpy as np
import pandas as pd
from pandas import *
df = DataFrame(np.ndindex(3,1,2,1), columns=['A','B','C','D'])
Out[7]:
   A  B  C  D
0  0  0  0  0
1  0  0  1  0
2  1  0  0  0
3  1  0  1  0
4  2  0  0  0
5  2  0  1  0

In [8]: df*5+2
Out[8]:
   A  B  C  D
0  2  2  2  2
1  2  2  7  2
2  7  2  2  2
3  7  2  7  2
4 12  2  2  2
5 12  2  7  2
```

Figura 88. Operaciones con escalares.

Fuente: elaboración propia.

Se pueden hacer **operaciones con escalares** (figura 88), notebook “Ejemplo_88.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
randn = np.random.rand
df1 = DataFrame( {'a':[1,0,1], 'b':[0,1,1] }, dtype=bool )
df2 = DataFrame( {'a':[0,1,1], 'b':[1,1,0] }, dtype=bool )
df1 & df2
```

	a	b
0	False	False
1	False	True
2	True	False

Figura 89. Operaciones lógicas.

Fuente: elaboración propia.

Se pueden hacer **operaciones lógicas** (figura 89), notebook “Ejemplo_89.ipynb”.

CONTINUAR

Transposición

Para transponer, se utiliza el atributo *T* (figura 90), notebook “Ejemplo_90.ipynb”:

```
In [8]: import numpy as np
import pandas as pd
from pandas import *
df = DataFrame(np.ndindex(3,1,2,1), columns=['A','B','C','D'])
df
```

Out[8]:

	A	B	C	D
0	0	0	0	0
1	0	0	1	0
2	1	0	0	0
3	1	0	1	0
4	2	0	0	0
5	2	0	1	0

```
In [9]: df.T
```

Out[9]:

	0	1	2	3	4	5
A	0	0	1	1	2	2
B	0	0	0	0	0	0
C	0	1	0	1	0	1
D	0	0	0	0	0	0

Figura 90. Transposición.

Fuente: elaboración propia.

CONTINUAR

Visualización

Hay varias formas de visualizar la información de un *dataframe*:

	A	B	C	D
0	0.879912	0.754664	0.450893	0.548262
1	0.668827	0.016458	0.182358	0.061669
2	0.626958	0.423782	0.552083	0.869737
3	0.923197	0.531068	0.804448	0.987810
4	0.216840	0.254141	0.101713	0.054160
5	0.301915	0.617831	0.483386	0.660352
6	0.641980	0.237544	0.286541	0.680817
7	0.431369	0.572311	0.450401	0.731337
8	0.055978	0.860867	0.160450	0.012160
9	0.346314	0.077852	0.430650	0.481859

Figura 91. Datos de un *dataframe*.

Fuente: elaboración propia.

Todos los datos **a partir del nombre del *dataframe*** (figura 91), notebook “Ejemplo_91.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
randn = np.random.rand
df = DataFrame( randn(10,4), columns=['A','B','C','D'] )
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 A    10 non-null float64
 B    10 non-null float64
 C    10 non-null float64
 D    10 non-null float64
 dtypes: float64(4)
 memory usage: 400.0 bytes
```

Figura 92. Datos de un *dataframe* usando *info()*.

Fuente: elaboración propia.

Un resumen de la información contenida, usando el método *info()* (figura 92), *notebook* “Ejemplo_92.ipynb”.

```
import numpy as np
import pandas as pd
from pandas import *
randn = np.random.rand
df = DataFrame( randn(10,4), columns=['A','B','C','D'] )
print(df.to_string())
```

	A	B	C	D
0	0.034685	0.914251	0.946942	0.083720
1	0.972669	0.142089	0.788396	0.013883
2	0.325790	0.850953	0.380769	0.602521
3	0.039785	0.360459	0.619699	0.168697
4	0.416723	0.932684	0.873190	0.249169
5	0.855723	0.608578	0.062883	0.336972
6	0.836523	0.497623	0.205874	0.121346
7	0.904203	0.011100	0.847687	0.518168
8	0.352006	0.465943	0.377309	0.191254
9	0.280667	0.284980	0.545421	0.602264

Figura 93. Datos de un *dataframe* usando *to_string()*.

Fuente: elaboración propia.

Como una cadena usando el método *to_string()* (figura 93), notebook “Ejemplo_93.ipynb”.

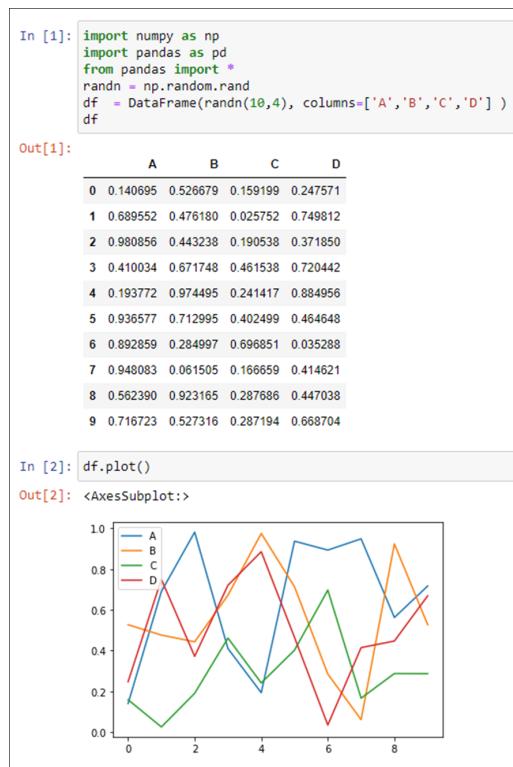
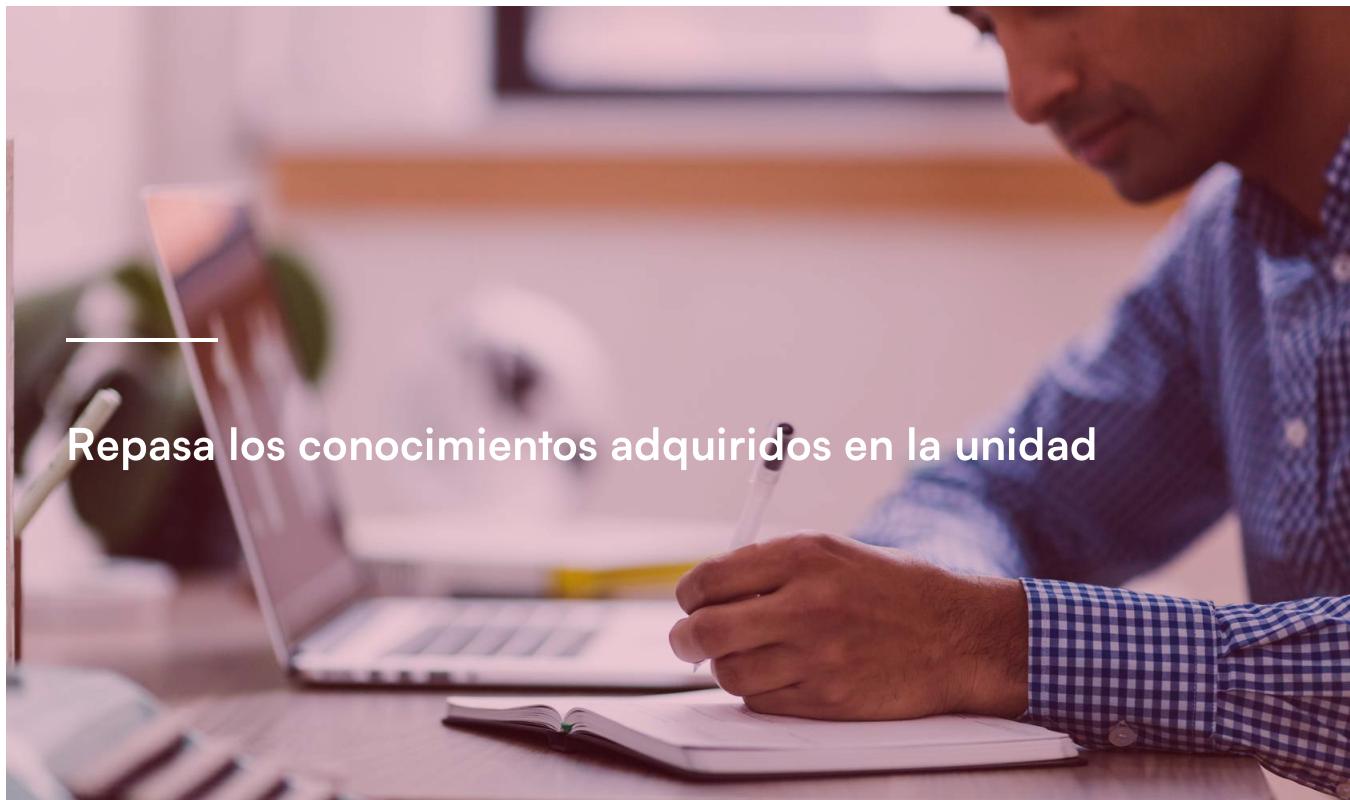


Figura 94. Pintando un dataframe.

Fuente: elaboración propia.

Se pueden **crear gráficas** a partir de los datos de un *dataframe*, usando el método *plot()*.

VI. Resumen



Repasa los conocimientos adquiridos en la unidad

En esta unidad se han presentado diferentes casos prácticos de situaciones reales en el ámbito del procesamiento de datos.

Asimismo, se han explorado las tres librerías Python más utilizadas para el procesamiento de datos en cada uno de esos casos.

La librería NumPy facilita la manipulación de los arrays de datos como si fueran tipos de datos básicos, lo que posibilita realizar operaciones con ellos sin tener que acudir a estructuras de datos más complejas.

Se han explorado las ventajas de usar NumPy con respecto a usar estructuras de datos tradicionales de Python:

- Expresividad de las operaciones. La sintaxis para manipular arrays ndimensionales se asemeja a la notación matemática, lo cual más sencilla la traslación de teoremas matemáticos a código.
- Las funciones permiten operar de forma implícita con matrices. Por ejemplo, transposición, multiplicación matricial, etc.

La librería Matplotlib permite la representación gráfica de puntos, que se pueden mostrar en diferentes tipos de gráficos. Asimismo, es posible utilizar diferentes tipos de gráficos para representar los datos. Los gráficos se pueden decorar con distintos elementos, como el título del gráfico o los valores de los ejes.

Matplotlib proporciona una interfaz de creación de gráficas muy similar a Matlab, lo cual facilita mucho a los usuarios de este último el familiarizarse con Matplotlib.

Por último, se ha visto que la librería **Pandas** proporciona mecanismos para la manipulación de datos en formato tabular, de forma similar a los datos de un fichero CSV o de una base de datos relacional. Pandas define un conjunto de estructuras de datos y un catálogo de operaciones sobre ellas y hace posible manipular y filtrar los datos de forma sencilla.

Saber más

Como información complementaria, se recomienda visualizar la siguiente clase magistral: “Introducción a dataframes con Python”, y descargar y ejecutar la mencionada clase magistral con los archivos que se facilitan. Para ello, en este enlace* se puede descargar una carpeta comprimida con los archivos que se necesitarán:

- **LMA_INTRO_DATAFRAMES PYTHON.ipynb** también disponible en **LMA_INTRO_DATAFRAMES PYTHON.html**

- Iris.csv
- Iris_merge.csv

Respecto a la clase, es posible verla en el siguiente enlace:

Juan Manuel Moreno. "[LMA BIG DATA: Introducción a dataframes con Phyton](#)".

IMF Business School, YouTube, 29/01/2019.

*

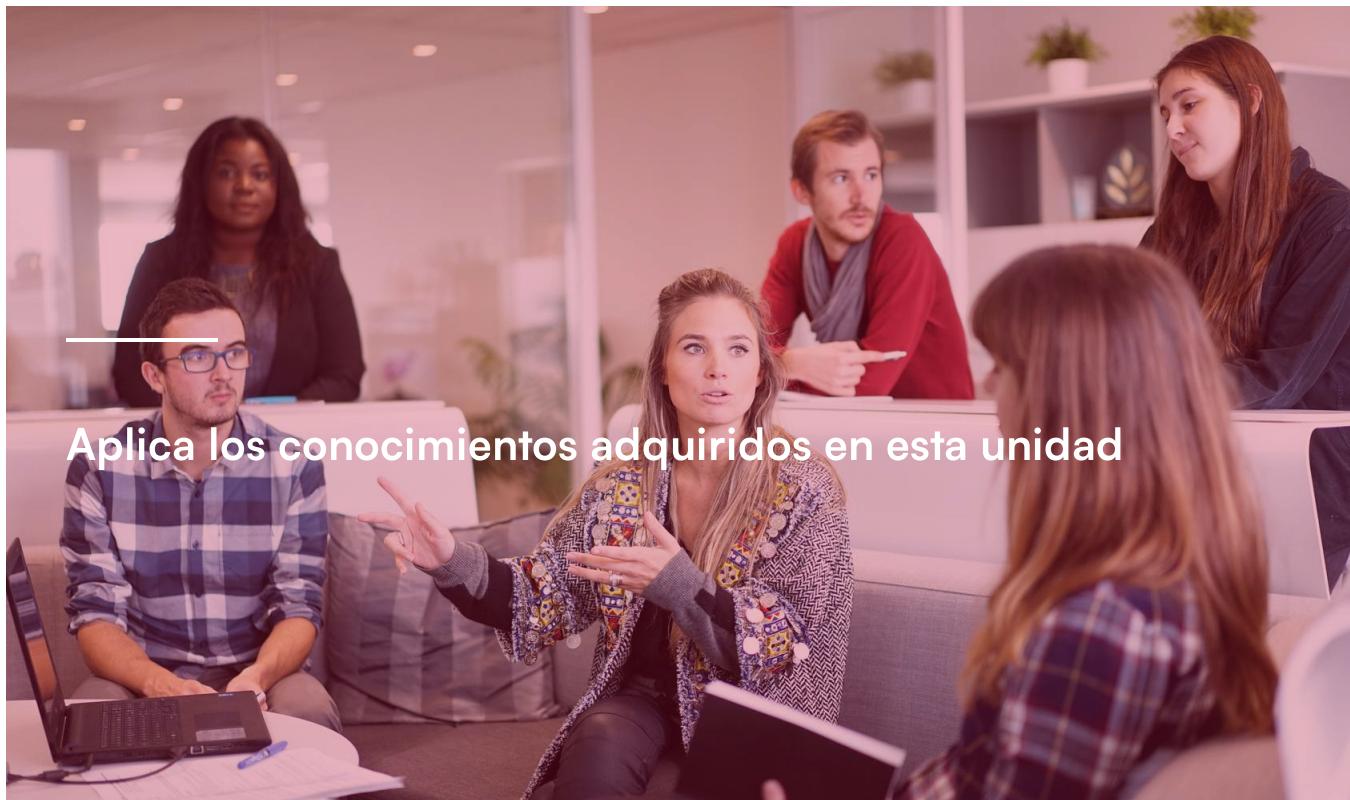


Archivos_LMA_Intro_Data_Frames_python.zip

4.5 KB



VII. Caso práctico con solución



Aplica los conocimientos adquiridos en esta unidad

DATOS

1. Crear una serie denominada "Sueldos_hombres" que contenga los datos [2500,1800,1900,2000,2100] y cuyos índices sean ["Euskadi", "Murcia", "Madrid", "Barcelona", "Zaragoza"], y otra denominada "Sueldos_mujeres" que contenga los datos [2300,1600,1980,1900,2150] y cuyos índices sean ["Euskadi", "Murcia", "Madrid", "Barcelona", "Zaragoza"]

"Córdoba"].

2. Usando las series anteriores, obtener otra serie que contenga la diferencia de sueldos entre mujeres y hombres.
3. Usando la serie anterior que contiene las diferencias de sueldos, obtener las ciudades donde la diferencia de sueldos es mayor de 100 euros.

VER SOLUCIÓN

SOLUCIÓN

La solución se puede descargar en el siguiente enlace:



[Caso_práctico.ipynb.zip](#)

1.3 KB



```
#Solucion A
import pandas as pd
sueldoshom = pd.Series( [2500,1800,1900,2000,2100],
                       index = ["Euskadi", "Murcia", "Madrid", "Barcelona", "Zaragoza"]
                      )
print(sueldoshom)
sueldosmuj = pd.Series( [2300,1600,1980,1900,2150],
                        index = ["Euskadi", "Murcia", "Madrid", "Barcelona", "Córdoba"]
)
print(sueldosmuj)

Euskadi      2500
Murcia       1800
Madrid        1900
Barcelona     2000
Zaragoza      2100
dtype: int64
Euskadi      2300
Murcia       1600
Madrid        1980
Barcelona     1900
Córdoba       2150
dtype: int64
```

Solución A

```
: #Solucion B
difsueldos = sueldoshom - sueldosmuj
difsueldos

: Barcelona      100.0
Córdoba          NaN
Euskadi         200.0
Madrid          -80.0
Murcia          200.0
Zaragoza          NaN
dtype: float64
```

```
#Solución C  
mascara = (difsueldos > 100)  
difsueldos[mascara]
```

```
Euskadi    200.0  
Murcia     200.0  
dtype: float64
```

VIII. Lecturas recomendadas

- [The Absolute guide for Beginners](#). NumPy.org.
- Maximov, L. [NumPy Illustrated: The Visual Guide to NumPy](#).
- [User's Guide](#). Matplotlib.org.

IX. Glosario



El glosario contiene términos destacados para la comprensión de la unidad

NumPy —

Extensión de Python que proporciona funciones y rutinas matemáticas para la manipulación de arrays y matrices de datos numéricos de una forma eficiente.

ndarray —

Arrays multidimensionales donde todos sus elementos son del mismo tipo y están indexados por una tupla de números positivos.

Matplotlib —

Librería de Python para realizar gráficos. Se caracteriza por que es fácil de usar, flexible y se puede configurar de múltiples maneras.

Pandas —

Librería construida sobre NumPy que ofrece estructuras de datos de alto nivel que facilitan el análisis de datos desde Python.

Dataframe —

Estructura que contiene una colección ordenada de columnas, cada una de las cuales puede tener valores de diferentes tipos. Está formada por datos y, opcionalmente, un índice (etiquetas de las filas) y un conjunto de columnas (etiquetas de las columnas). En caso de no existir un índice, se genera a partir de los datos.

Serie —

Objeto, como un array, que está formado por un array de datos y un array de etiquetas denominado índice.

Subgráfico —

Gráfico formado por un conjunto de gráficos.

axe —

Objeto que representa un área donde se puede dibujar.

X. Bibliografía

- [NumPy Quick Start.](#)
- [Matplotlib.](#)
- McKinney, W. *Python for Data Analysis: Data Wrangling with Pandas, NumPy and iPython*. O'Reilly; 2012.
- [Pandas: powerful Python data analysis toolkit.](#) v1.2.4 2021.
- [Python Data Analysis Library—pandas: Python Data Analysis Library.](#)
- Tosi, S. *Matplotlib for Python Developers*. PACKT; 2009.

XI. Apéndice

En unidades anteriores de este módulo se han estudiado las funcionalidades generales del lenguaje de programación Python con el fin de, posteriormente, orientarse a las librerías científicas para análisis de datos. En este apartado, se incluyen una serie de ejercicios complementarios en los que convergen diferentes técnicas como manejo de archivos, creación de funciones y desarrollo de diferentes estructuras de datos. Se recomienda descargar y ejecutar los cuadernos Jupyter Notebook, entre los que se incluye:

- Variables numéricas y de texto, además de operadores booleanos.
- Estructuras de control y ciclos.
- Tuplas, listas y diccionarios.
- Funciones, JSON y CSV.
- *Dataframe* y *Pyplot*.

Se pueden descargar todos los notebooks con los ejercicios complementarios en este enlace:

