Fundamentos de tratamiento de datos con el stack científico de Python © EDICIONES ROBLE, S.L.

campusformacion.imf.com © EDICIONES ROBLE, S.L.
ALEXIS GASTON VILLAGRA RAMIREZ
campusformacion.imf.com © EDICIONES ROBLE, S.L.
ALEXIS GASTON VILLAGRA RAMIREZ

Indice

I. Introducción

Actualmente, en el ámbito del análisis de datos, Python se ha convertido en uno de los lenguajes más utilizados. Python es un lenguaje de propósito general que no está diseñado de forma específica para realizar análisis de datos. Sin embargo, se ha definido un conjunto de librerías científicas para este fin que le dotan de toda la funcionalidad necesaria para realizar este tipo de tareas.

En esta unidad se van a estudiar las librerías científicas de Python. En primer lugar, se examinará la librería NumPy, que permite la gestión y manipulación de arrays de datos, así como el cálculo de operaciones estadísticas. La principal novedad de NumPy es la posibilidad de manejar arrays de datos como si fueran tipos de datos básicos.

También se estudiará la librería Matplotlib. Esta permite la representación gráfica de datos usando diferentes formatos de representación y tipos de gráficas. De una manera sencilla se pueden configurar las diferentes gráficas y datos que se representan.

Por último, se describirá la librería Pandas, la cual introduce un conjunto de estructuras de datos nuevas que permite realizar operaciones de manipulación de datos de una forma muy sencilla. Estas operaciones —como selección, limpieza o transformación de datos— suelen ser bastante útiles cuando se están preparando los datos para ser analizados.

II. Objetivos

Los objetivos que los alumnos alcanzarán tras el estudio de esta unidad son:

- Conocer la librería NumPy de Python y saber utilizarla para llevar a cabo la gestión de arrays y el cálculo estadístico.
- Conocer la librería Matplotlib de Python y saber utilizarla para realizar representaciones gráficas de datos.
- Conocer la librería Pandas de Python y saber utilizarla para manipular datos y analizarlos.
- 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, arrays multidimensionales donde todos sus elementos son del mismo tipo y están indexados por una tupla de números positivos.

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

Atributos de los ndarray

Cada ndarray tiene un conjunto de atributos que le 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 conteniendo los elementos actuales del array. Normalmente no se usa este atributo, pues se accede a los elementos directamente mediante los índices.

Importación de NumPy

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

```
import numpy as np
```

Creación de un array

Existen varias formas para crear un array:

La función array()

La forma más sencilla de crear un array es utilizando la función**array** y una secuencia de valores (figura 6.1.). 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] )
a
```

```
array([2, 3, 4])
```

```
a.dtype
```

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

Figura 6.1. Creación de un array usando la función array.

Cuando a la función array se le pasa como argumento una secuencia de secuencias, genera un array de tantas dimensiones como secuencias (figura 6.2.).

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

```
array([[2, 3, 4],
       [5, 6, 7]])
```

Figura 6.2. Creación de un array con una secuencia de secuencias.

El tipo del array puede ser especificado en el momento de la creación del mismo (figura 6.3.).

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

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

Figura 6.3. Especificación del array en el momento de su creación.

La función arange()

Creación de un array mediante la función **arange()**, que genera una secuencia de números (figura 6.4.). Toma como parámetros el rango de los números a generar, y la distancia entre ellos, y genera un array unidimensional.

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

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

Figura 6.4. Creación de un array con el método arange.

Redimensionar el array

Se puede redimensionar el array que genera **arange()** mediante el método **reshape()**, que toma como argumentos la dimensiones (figura 6.5.). Las dimensiones deben ser consistentes con el número de elementos generados.

```
b=arange(2,6,0.3).reshape(2,7)
b
```

```
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 6.5. Redimensión de un array.

Uso de argumentos reales con arange()

Cuando se usan argumentos reales con **arange()**, es mejor usar la función **linspace** (figura 6.6.), 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.


```
a=np.linspace(1,10,8)
a
```

```
array([  1.        ,   2.28571429,   3.57142857,   4.85714286,
         6.14285714,   7.42857143,   8.71428571,  10.        ])
```

Figura 6.6. Uso de la función linspace.

Array aleatorio con rand()

Creación de un array unidimensional con datos aleatorios mediante la función **rand** del módulo **Random** (figura 6.7.). La función **rand** devuelve un número aleatorio procedente de una distribución uniforme en el intervalo [0,1].

```
a = np.random.rand(10)     # genera 10 números aleatorios entre el 0 y el 1
a
```

```
array([ 0.59536657,  0.01584197,  0.42919441,  0.57029431,  0.94424139,
        0.31224204,  0.90745139,  0.31357952,  0.75333686,  0.8956221 ])
```

Figura 6.7. Creación de un array con datos aleatorios.

También es posible generar arrays multidimensionales si se proporcionan las dimensiones como argumentos (figura 6.8.).

```
b= np.random.rand(3, 4)      # valores aleatorios - 3 filas, 4 columnas
b
```

```
array([[ 0.59920063,  0.34591146,  0.36417726,  0.21562577],
       [ 0.04329456,  0.90805169,  0.9420666 ,  0.72941838],
       [ 0.88242129,  0.53053534,  0.57200571,  0.74450306]])
```

Figura 6.8. Creación de un array multidimensional.

Funciones especiales para la creación de arrays

Existen funciones que generan arrays especiales que solo requieren como argumento el tamaño del array (figura 6.9.):

- **zeros**: crea un array lleno de ceros.
- **ones**: crea un array lleno de unos.
- **empty**: crea un array cuyo contenido es aleatorio.

```
a=zeros(4)
a
```

```
array([ 0.,  0.,  0.,  0.])
```

```
b=ones([1,2])
b
```

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

```
c=empty((2,3))
c
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Figura 6.9. Funciones especiales.

Operaciones sobre arrays

En NumPy se puede operar aritméticamente sobre los arrays como si se tratara de escalares. Se caracteriza por:

Operaciones sobre la misma posición

Las operaciones actúan sobre los elementos de la misma posición y, como resultado, crean un nuevo array (figura 6.10.)

```
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])
```

```
c=a*10
c
```

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

Figura 6.10. Operaciones aritméticas entre dos arrays.

Arrays de diferente tipo

Cuando se opera con arrays de diferente tipo, el tipo del array resultante corresponde al más general (figura 6.11.)

```
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 6.11. Operaciones entre arrays de diferente tipo.

Funciones matemáticas universales

En NumPy existe un conjunto de funciones matemáticas denominadas**funciones universales**: sin, cos ,exp… (figura 6.12.). 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 6.12. Funciones universales sobre arrays.

En particular es muy útil la función**dot()**, que permite realizar la multiplicación matricial (figura 6.13.)

```
a=np.array([[2,3],[4,5]])
b=np.array([[1,2],[0,1]])
```

```
c=np.dot(a,b)
c
```

```
array([[ 2,  7],
       [ 4, 13]])
```

Figura 6.13. Multiplicación matricial.

El argumento axis

Existen algunas operaciones implementadas como métodos de la clase ndarray y, por defecto, se aplican a todos los elementos del array. Sin embargo, es posible especificar la dimensión sobre la que se quiere aplicar la operación mediante el argumento **axis** (figura 6.14.) que toma el valor 0 (columnas) o 1 (filas).

```
a=np.empty((3,4))
a
```

```
array([[  6.23042070e-307,   4.67296746e-307,   1.69121096e-306,
          1.78020848e-306],
       [  1.37961709e-306,   7.56599807e-307,   8.90104239e-307,
          1.24610383e-306],
       [  8.90092016e-307,   1.24611062e-306,   1.42410974e-306,
          1.24611266e-306]])
```

```
a.sum()
```

```
206
```

```
a.sum(axis=0)
```

```
206
```

Figura 6.14. Uso del argumento axis.

Operadores += y *=

Los operadores += y *= modifican los arrays en vez de crear uno nuevo (figura 6.15.).

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

```
a+=3
a
```

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

```
a*=a
a
```

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

Figura 6.15. Modificación de arrays.

Acceso a arrays

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.

Acceso en arrays unidimensionales

Cuando trabajamos con arrays de una dimensión, el acceso a los elementos se realiza de forma similar al de las listas o tuplas de elementos (figura 6.16.).

```
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 6.16. Acceso indexado a un array unidimensional

Diferencia con las listas

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 6.17.):

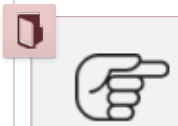
```
b = arr[-2:]
b[::] = 0
arr
```

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

Figura 6.17. Modificaciones de un array

Acceso a elementos en array multidimensional

El acceso a los elementos de un array bidimensional se realiza indicando los índices separados por una coma (figura 6.18.).



```
b =np.array([[ 0,  1,  2,  3],
             [10, 11, 12, 13],
             [20, 21, 22, 23],
             [30, 31, 32, 33],
             [40, 41, 42, 43]])
b
```

```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

```
b[2,3] #Acceso al elemento de la fila 3 y columna 4
```

```
23
```

```
b[0:5,1] #Acceso a los elementos de todas las filas de la columna 2
```

```
array([ 1, 11, 21, 31, 41])
```

Figura 6.18. Acceso de un array de varias dimensiones.

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 6.19.).

```
b[0] #Todas las columnas de la fila 1
```

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

Figura 6.19. Acceso a un array multidimensional indicando menos dimensione

Acceso mediante máscaras

Otra forma de acceso a partes de un array de NumPy es mediante un array de booleanos denominado máscara (figura 6.20.).

```
n = np.arange(10000)
n
```

```
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

```
( n > 20 ) & ( n < 44 )      # Se comprueba si se cumple la condición en los elementos del array
```

```
array([False, False, False, ..., False, False, False], dtype=bool)
```

```
mascara = ( n > 20 ) & ( n < 44 )    #Se crea la mascara con el array de booleanos
mascara
```

```
array([False, False, False, ..., False, False, False], dtype=bool)
```

```
n[mascara] #Se usa la mascara para recuperar los valores que cumplen 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 6.20. Acceso a un array mediante una máscara

Iteración sobre arrrays

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

Usando un bucle for

Usando un **for** (figura 6.21.).

```
a=np.arange(10)
a
```

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

```
for i in a:
    print (i)
```

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

Figura 6.21. Recorrido de un array usando un for.

Usando el iterador flat

Usando el iterador **flat**, que permite iterar sobre todos los elementos del array: aplana el array y lo convierte en un array unidimensional (figura 6.22.).

```
for i in a.flat:
    print (i)
```

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

Figura 6.22. Recorrido de un array usando el iterador flat.

Manipulaciones sobre un array

Sobre un array se pueden realizar diferentes tipos de manipulaciones:

Cambio de tamaño

Para ello se puede aplanar la matriz mediante la función**ravel()** y, a continuación, usar el método **shape()** o **resize()** para establecer las nuevas dimensiones (figura 6.23.).

```
a=np.arange(12).reshape(3,4)
a
```

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

```
a.ravel()
```

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

```
a.shape=(4,3)
a
```

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

Figura 6.23. Cambio de tamaño.

Fusión de dos arrays

Fusionar dos arrays (figura 6.24.) mediante la función**concatenate()** verticalmente (atributo axis=0) o bien horizontalmente (atributo axis=1).

```
a=np.arange(4).reshape(2,2)
b=np.arange(5,9,1).reshape(2,2)
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 6.24. Fusión de dos arrays.

División de un array

Dividir un array en partes iguales usando las funciones **hsplit()** y **vsplit()**, indicando la columna o fila por donde se divide dependiendo la función (figura 6.25.).

```
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 por la columna 2
```

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

Figura 6.25. División de un array.

Alternativamente se podría descomponer el array, lo cual se hace por filas (figura 6.26.).

```
import numpy as np
a= np.array([[1, 2, 3, 5],
             [4, 5, 6, 9]])
a
```

```
array([[1, 2, 3, 5],
       [4, 5, 6, 9]])
```

```
b,c=a  #División horizontal
b
```

```
array([1, 2, 3, 5])
```

```
c
```

```
array([4, 5, 6, 9])
```

Figura 6.26. División de un array por filas.

Asignaciones y copias

Las asignaciones y las llamadas a funciones no hacen copia del array o de sus datos (figura 6.27.).

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

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

```
b=a
b.shape=(1,4)
a
```

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

Figura 6.27. Ejemplo de asignación.

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 6.28.).

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

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

```
c=a.view()
c
```

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

```
c.shape=(1,4)
c
```

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

Figura 6.28. Uso del método view().

El método copy()

Permite realizar una copia independiente del array original (figura 6.29.)

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

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

```
d=a.copy()
d
```

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

```
d[1,1]=999
a
```

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

Figura 6.29. Copia de un array.

El método sort()

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 6.30.).

```
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 6.30. Ordenación de arrays.

Los métodos any() y all()

Se puede operar sobre arrays de booleanos mediante los métodos**any** and **all**. Permiten chequear si algún valor es cierto o si todos los valores son ciertos respectivamente (figura 3.31.).

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

```
True
```

Figura 6.31. Operación sobre arrays.

Operaciones estadísticas

Por último, es apropiado comentar que el módulo NumPy proporciona métodos que permiten realizar otras operaciones matemáticas, tales 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

Suma, mínimo, máximo (figura 6.32.).

```
import numpy as np
a=np.arange(8)
a
```

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

```
print( np.sum(a) ) #Suma de todos los elementos
print( np.min(a) , "--" , np.argmin(a) ) #Valor mínimo y su posición
print( np.max(a) , "--" , np.argmax(a) ) #Valor máximo y su posición
```

```
28
0 -- 0
7 -- 7
```

Figura 6.32. Suma, máximo y mínimo.

Media

Es la suma de todos los elementos dividida por el número de elementos. En Python puede calcularse usando la función numpy.mean (figura 6.33.).

```
np.mean(b) #Media del array b considerando todos sus valores
```

```
3.5
```

```
np.mean(b, axis=1) #Media por filas
```

```
array([ 0.5,  2.5,  4.5,  6.5])
```

Figura 6.33. Media.

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 6.34.).

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

```
array([ 5.,  5.])
```

Figura 6.34. Varianza.

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 6.35.)

```
np.std(b, axis = 0)  #Desviación estándar (se obtiene por columnas)
```

```
array([ 2.23606798,  2.23606798])
```

Figura 6.35. Desviación estándar.

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 6.36.)

```
np.median(b, axis=1) #Mediana por filas
```

```
array([ 0.5,  2.5,  4.5,  6.5])
```

Figura 6.36. Mediana.

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 6.37.).

```
"""
Se calcula el coef. de correlación teniendo en cuenta que cada fila representa una variable
"""
b=np.array([[2,3,4,5],[4,5,6,6]])
np.corrcoef(b)      # correlación de la transpuesta
```

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

Figura 6.37. Correlación.

Covarianza

Determina si existe una dependencia entre dos variables aleatorias. En Python se puede calcular usando numpy.cov (figura 6.38.)

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

```
array(2.0)
```

Figura 6.38. Covarianza.

Métodos aleatorios

Métodos aleatorios (figura 6.39.).

```
import numpy.random as r
s=r.rand(10) #Genera 1o números aleatorios de una normal (0,1)
print(s)
print()
t=r.normal(size=(5,1)) #Genera un array con números pertenecientes a una normal
print(t)
print()
k=r.normal(4,5) #Genera un número perteneciente a una normal (4,5)
print(k)
```

```
[ 0.89883229  0.90683388  0.68792669  0.49149395  0.67277533  0.79540743
  0.23956671  0.89474218  0.78273116  0.12295365]

[[ 0.32774169]
 [ 0.3672161 ]
 [-0.8507467 ]
 [-0.73066359]
 [-0.76842301]]

-3.025318953000859
```

Figura 6.39. Métodos aleatorios

IV. Representación gráfica con Matplotlib

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

Importación del módulo

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 comando plot()

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 6.40.).

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

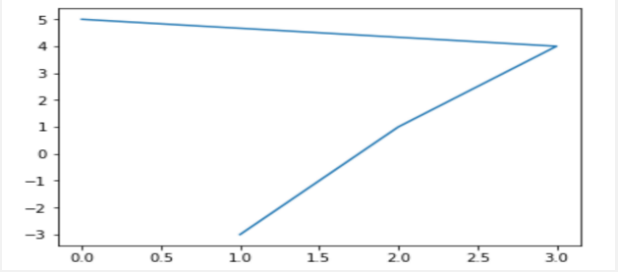


Figura 6.40. Uso de plot().

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 6.41.).

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

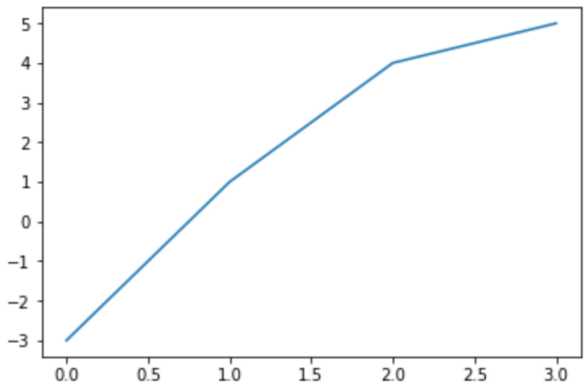


Figura 6.41. Representación omitiendo valores.

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

Presentar datos de secuencias

Secuencia única

Para representar datos, puede ser útil usar funciones que generen secuencias tales como **range(i,j,k)** o **arange (i, j, k)** (figura 6.42.).

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

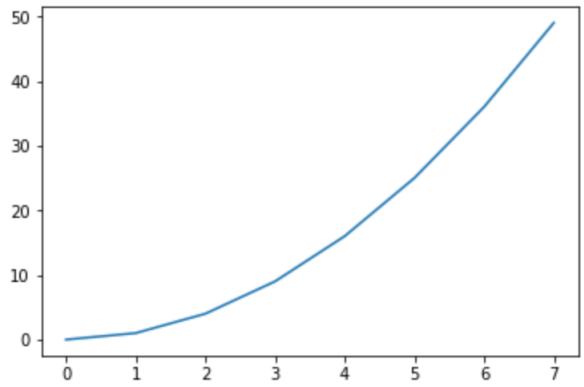


Figura 6.42. Representación usando range().

Secuencias múltiples

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 a **plot()** para que dibuje cada una de las funciones y, en último lugar, invocar al comando **show()** (figura 6.43.).

```
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()
```

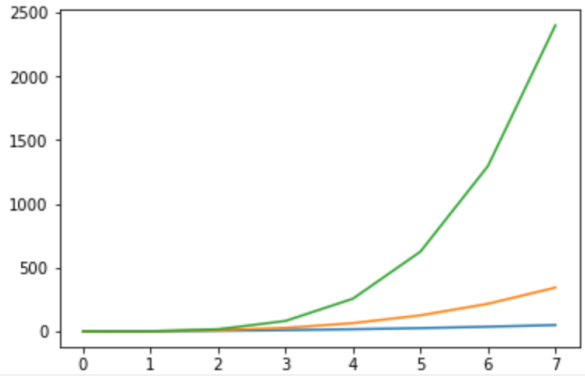


Figura 6.43. Representación de varias funciones en una misma gráfica.

Otra forma de realizar la misma operación es utilizando NumPy (figura 6.44.).

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

t = np.arange(8)

plt.plot(t, t**2, t, t**3, t, t**4)
plt.show()
```

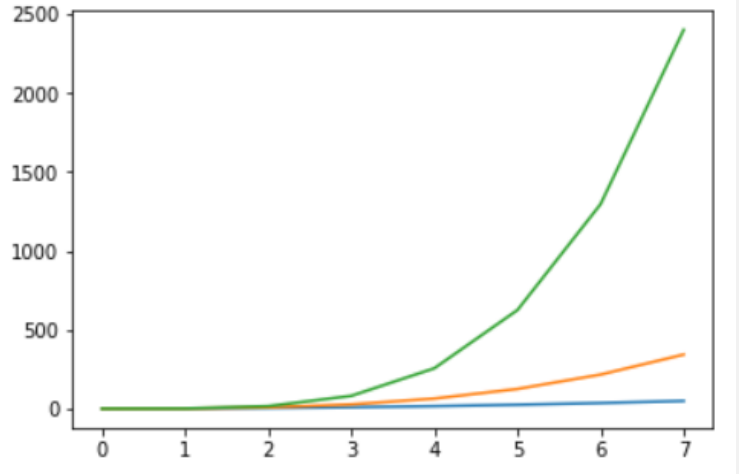


Figura 6.44. Representación de varias funciones en una misma gráfica.

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

Elementos decorativos

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

Ejes de coordenadas

Por defecto, se establecen los valores que aparecen en los ejes de coordenadas, de manera que puedan mostrarse en la gráfica los puntos que son dibujados. Para configurar estos valores se usa la función **axis()**.

axis() sin parámetros

Cuando se ejecuta sin parámetros, devuelve la escala actual utilizada en los ejes en forma de una tupla de 4 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 6.45.).

```
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()
```

```
(-0.35000000000000003,
 7.3499999999999996,
 -2.4500000000000002,
 51.450000000000003)
```

Figura 6.45. Función axis().

Cambio de escalas

Para cambiar las escalas usadas, se invoca la función**axis()** mediante una lista de 4 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 6.46.).

```
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()
```

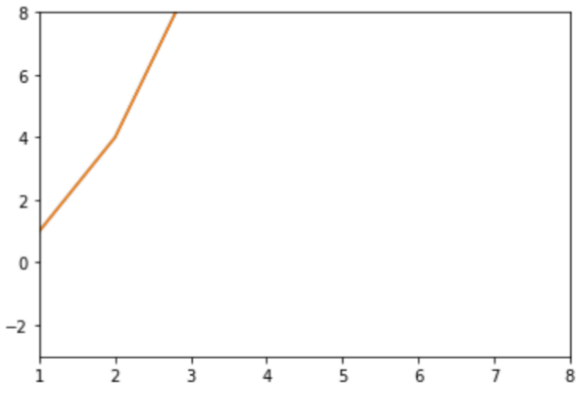


Figura 6.46. Cambio de escala.

Configuración de un solo eje

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 2 valores que representan respectivamente el límite inferior y superior de la escala de un eje.

Gestión de los valores sobre los ejes

Para gestionar los valores que se colocan sobre los ejes y la localización de ellas, se usan las funciones **plt.xticks()** e **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 6.47.).

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi for xi in x])
plt.xticks(range(8),['a','b','c','d','e','f','g','h'])
plt.yticks(range(0,16,2))
plt.show()
```

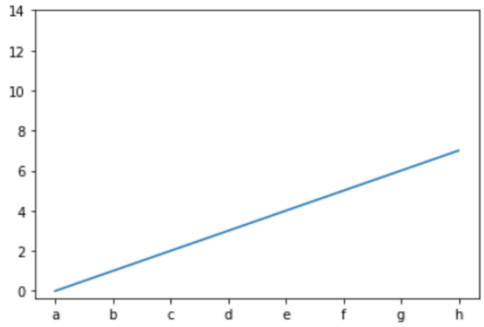


Figura 6.47. Valores de los ejes.

Añadir etiquetas a los ejes

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.

Título

Para añadir un título se utiliza la función **plt.title()**, que toma como parámetro el valor del título (figura 6.48.).

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi**2 for xi in x])
plt.title("Representación de la función cuadrática")
plt.xlabel("Eje de valores x")
plt.ylabel("Eje de valores y")
plt.show()
```

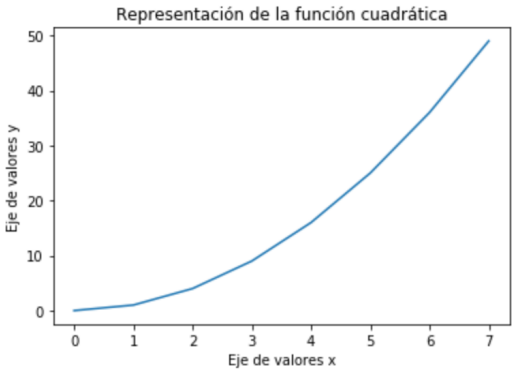


Figura 6.48. Gráfica con título y valores en los ejes.

Cuadrícula

Para añadir una cuadrícula a una gráfica basta con usar la función **plt.grid()**, la cual toma como parámetro el valor True (habilitar la cuadrícula) o False (deshabilitar la cuadrícula).

En la figura 6.49., aparece representada.

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi**2 for xi in x])
plt.grid(True)
plt.title("Representación de la función cuadrática")
plt.xlabel("Eje de valores x")
plt.ylabel("Eje de valores y")
plt.show()
```

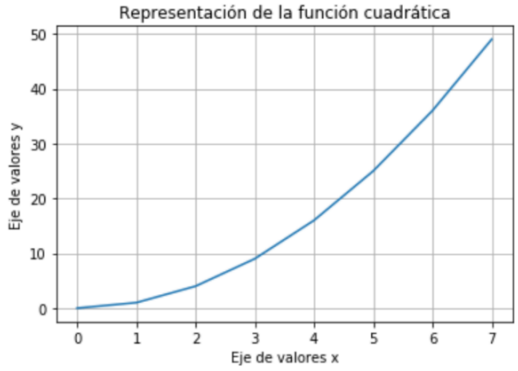


Figura 6.49. Gráfica con cuadrícula.

Leyenda

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 6.50.).

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi for xi in x], label="Lineal")
plt.plot(x, [xi**2 for xi in x], label="Cuadrática")
plt.plot(x, [xi**3 for xi in x], label="Cúbica")
plt.legend()
plt.show()
```

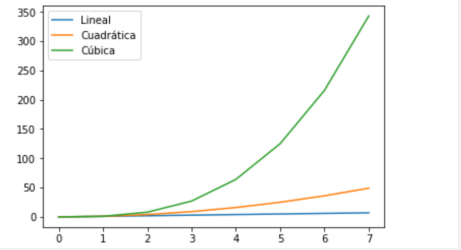


Figura 6.50. Leyenda.

Formato de puntos

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 se obtiene concatenando conjuntos de caracteres que representan el color, el estilo y la forma, respectivamente (tabla 6.1.).

Cadena	Código
blue	b
cyan	c
green	g
black	k
magenta	m
red	r
white	w
yellow	y

Código	Significado
-	Línea sólida
--	Línea discontinua
-.	Línea discontinua con puntos
:	Línea de puntos

Código	Significado
.	Punto
,	Píxel
0	Círculo
V	Triángulo hacia abajo
^	Triángulo haica arriba
<	Triángulo hacia la izquierda
>	Triángulo hacia la derecha
1	Trípode hacia abajo
2	Trípode hacia abajo
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 6.1. Formato de los puntos representados.

En el siguiente ejemplo (figura 6.51.), se muestra un formateado de puntos.

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi for xi in x], 'b-d',label="Lineal")
plt.plot(x, [xi**2 for xi in x], 'y--p', label="Cuadrática")
plt.plot(x, [xi**3 for xi in x], 'r:s',label="Cúbica")
plt.legend()
plt.show()
```

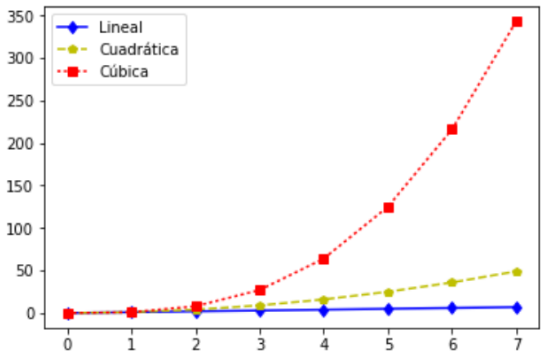


Figura 6.51. Formateado de varios puntos.

Texto

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 6.52.).

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi for xi in x])
plt.text(1,3,"Función lineal")
plt.show()
```

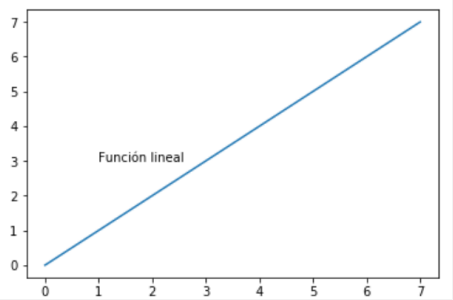


Figura 6.52. Texto sobre la gráfica.

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 6.53.

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi for xi in x])
plt.figtext(0.5,0.5,"Función lineal")
plt.show()
```

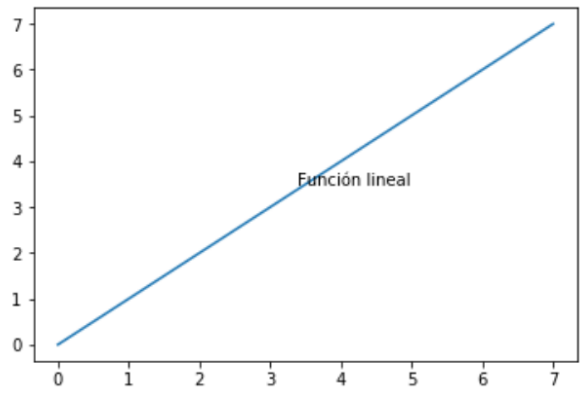


Figura 6.53. Situando texto sobre la gráfica.

Anotación

Para añadir una anotación, se usa la función **plt.annotate()**, la cual toma como argumentos (figura 6.54.):

- 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 width (ancho de la flecha), headlength (longitud de la flecha reservada a la cabeza de la misma), 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).

```
import matplotlib.pyplot as plt
x = range(8)
plt.plot(x, [xi for xi in x])
plt.grid(True)
plt.annotate("Punto (2,2)",xy=(2,2),xytext=(3,2),
arrowprops=dict(facecolor="green",shrink=0.05, headlength=2))
plt.show()
```

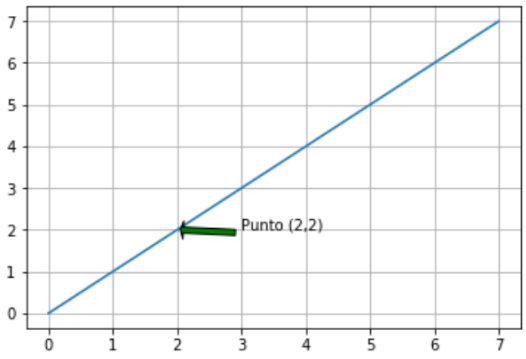


Figura 6.54. Anotación sobre la gráfica.

Tipos gráficos más usados con Matplotlib

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

Histogramas

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.

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 6.55.).

```
import matplotlib.pyplot as plt
import numpy as np
y = np.random.randn(1000)
plt.hist(y,15);
plt.show()
```

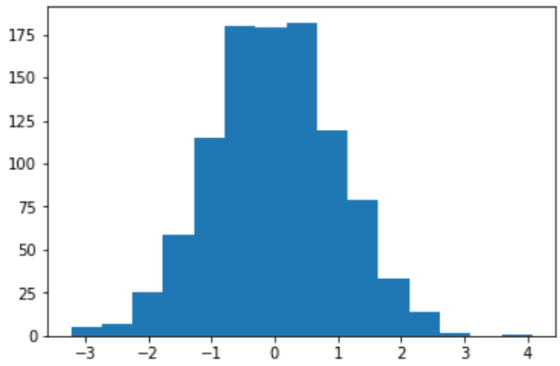


Figura 6.55. Historiograma

Diagramas de barras

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.

Diagrama de barras verticales

Para dibujar un diagrama de barras se utiliza la función **plt.bar()**, que tiene como parámetros (figura 6.56.):

- **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.
- **Argumento 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.

```
import matplotlib.pyplot as plt
p1 = plt.bar(left=range(5), height=[20, 35, 30, 35, 27],
width=0.4,color='green',yerr=[2, 3, 4, 1, 2])
p2 = plt.bar(left=range(5), 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()
```

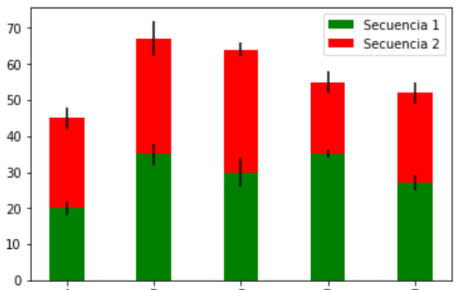


Figura 6.56. Diagrama de barras verticales.

Diagrama de barras horizontales

Los mismos datos se podrían haber representado horizontalmente utilizando la función **plt.barh()**. Esta función tiene parámetros similares a la anterior (figura 6.57.).

- **Argumento bottom.** Es una lista de las coordenadas del eje y que indican dónde se situarán 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.

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

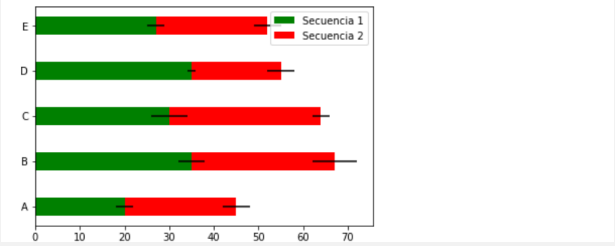


Figura 6.57. Diagrama de barras horizontal.

Diagramas circulares

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 a la cantidad descrita.

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.

Parámetros

La función tiene los siguientes parámetros (figura 6.58.):

- **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.

Ejemplo

```
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()
```

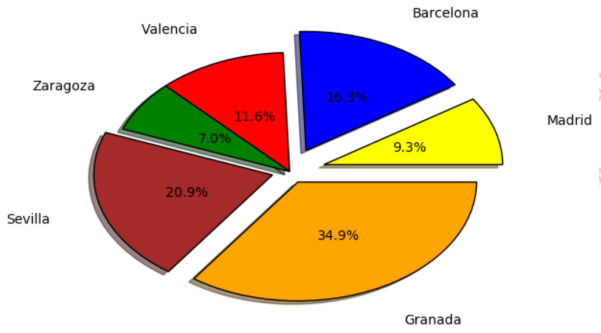


Figura 6.58. Diagrama circular.

Diagramas de dispersión

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 de 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.

Parámetros

Los principales argumentos que admite la función son:

- **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 a la tabla 6.2.:

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 6.2. Tabla de configuración.

Ejemplo

En la figura 6.59., se muestra un ejemplo de diagrama de dispersión.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000)
y = np.random.randn(1000)
colores=np.random.randn(1000)
tam=50*np.random.randn(1000)
plt.scatter(x, y, s=tam, c=colores, marker='+');
plt.show()
```

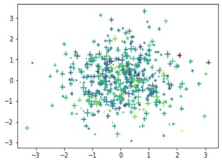


Figura 6.59. Diagrama de dispersión.

Diagramas de contornos

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()**, que toman como entrada un array de 2 dimensiones y, opcionalmente, el número niveles de líneas de contorno (figura 6.60.). Las líneas se representan en diferentes colores, de 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—.

```
import matplotlib.pyplot as plt
import numpy as np
valores = np.random.rand(21, 31)
plt.contour(valores,5)
plt.show()
```

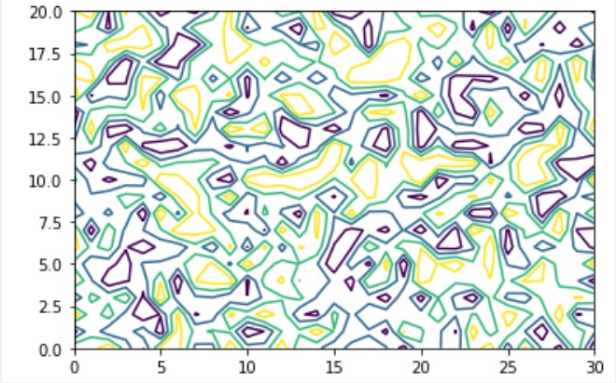


Figura 6.60. Diagrama de contorno.

Diagrama de caja

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** (figura 6.61.).

```
import matplotlib.pyplot as plt
import numpy as np
valores = np.random.randn(100)
plt.boxplot(valores)
plt.show()
```

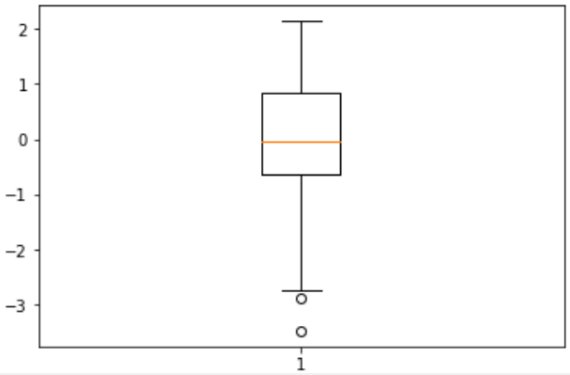


Figura 6.61. Diagrama de caja.

Conjunto de gráficos

En Matplotlib existe la posibilidad de crear gráficas formadas por un conjunto de gráficos. A estos gráficos se les denomina 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.

Parámetros

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 elsubgráfico 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.

Ejemplo

Los objetos de tipo **axe** pueden invocar la función **plt.plot()** para representar puntos (figura 6.62.).

```
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()
```

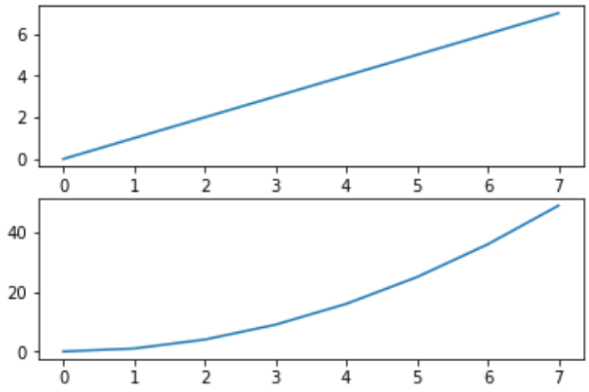


Figura 6.62. Ejemplo de subgráfico.

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.
- DataFrame.

Primero, importación de librerías

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

```
import numpy as np

import pandas as pd

from pandas import *
```

1. Series

Una serie es un objeto, como un array, que está formado por un array de datos y un array de etiquetas denominado índice.

Tipos de arrays de datos

El array de datos puede ser de 3 tipos:

ndarray

Si el array de datos en un ndarray, entonces el índice debe ser de la misma longitud que el array de datos (figura 6.63.).

```
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 6.63. Ejemplo de ndarray.

Si ningún índice es pasado, entonces se crea uno formado por valores que van desde [0, …,len(data)-1] (figura 6.64.).

```
Series([1,2,3,4,5])
```

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Figura 6.64. Ejemplo de ndarray sin índice.

Diccionario

Si se pasa un índice, los valores del diccionario son asociados a los valores del índice (figura 6.65.).

```
d = {'a' : 0., 'b' : 1., 'c' : 2.}
```

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

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

Figura 6.65. Creación de un ndarray usando un diccionario.

Si no se proporciona un índice, entonces se construye a partir de las claves ordenadas del diccionario (figura 6.66.).

```
d = {'a' : 0., 'b' : 1., 'c' : 2.
```

```
Series(d)
```

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

Figura 6.66. Creación de ndarray usando un diccionario sin índice.

Valor escalar

En este caso, se debe proporcionar un índice y el valor será repetido tantas veces como la longitud del índice (figura 6.68.).

```
Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

```
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Figura 6.67. Creación de ndarray un valor escalar.

Características de las series

Algunas características de las series son:

Similares a ndarray

Las series actúan de forma similar a ndarray, siendo un argumento válido de funciones de NumPy (figura 6.68.).

```
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 6.68. Uso de ndarray con NumPy.

Diccionario de tamaño fijo

Las series actúan como un diccionario de tamaño fijo en el que se pueden gestionar los valores a través de los índices (figura 6.69.).

```
s = Series([1,2,3,4,5], index=['a', 'b', 'c', 'd', 'e'])
```

```
s['a']
```

```
1
```

```
s['e'] = 12.
```

```
s
```

```
a     1
b     2
c     3
d     4
e    12
dtype: int64
```

```
'e' in s
```

```
True
```

Figura 6.69. Uso de series como diccionarios.

Permiten operaciones vectoriales

Sobre las series se pueden realizar operaciones vectoriales.

```
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 6.70. Operaciones vectoriales sobre series.

Alineamiento

La principal diferencia entre series y ndarray 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 6.71.).

```
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 6.71. Uso del atributo name.

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.

Datos en un dataframe

Los datos de un dataframe pueden proceder de:

Diccionario de ndarrays

Los arrays (figura 6.72.) 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… longitud(array)-1.

```
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

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

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

Figura 6.72. Diccionario de ndarrays.

Lista de diccionarios

Lista de diccionarios (figura 6.73.).

```
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 6.73. Lista de diccionarios.

Diccionario de tuplas

Diccionario de tuplas (figura 6.74.).

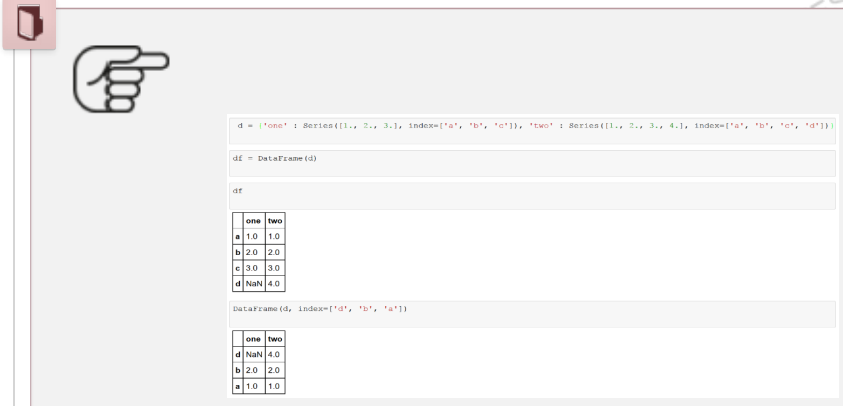
```
DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
           ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
           ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
           ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
           ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
```

		a			b	
		a	**b**	**c**	**a**	**b**
A	**B**	4.0	1.0	5.0	8.0	10.0
	C	3.0	2.0	6.0	7.0	NaN
	D	NaN	NaN	NaN	NaN	9.0

Figura 6.74. Diccionario de tuplas.

Diccionario de series

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 toman como tales la lista ordenada de las claves de los diccionarios.



```
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
d	NaN	4.0
b	2.0	2.0
a	1.0	1.0

Figura 6.75. Diccionario de series.

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

Array estructurado

Array estructurado (figura 6.76.).

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

In [2]: data = np.zeros((2,),dtype=[("A","i4"),("B","f4"),("C","U10")])

In [3]: data
Out[3]: array([(0,  0., ''), (0,  0., '')],
              dtype=[('A', '<i4'), ('B', '<f4'), ('C', '<U10')])

In [4]: data[:] = [(1,2.3,"Hola"),(2,3.4,"World")]

In [5]: import pandas as pd

In [6]: data = pd.DataFrame(data)

In [7]: data
Out[7]:
```

	A	B	C
0	1	2.3	Hola
1	2	3.4	World

```
In [8]: data.index = ["Primera_Fila", "Segunda_Fila"]

In [9]: data
Out[9]:
```

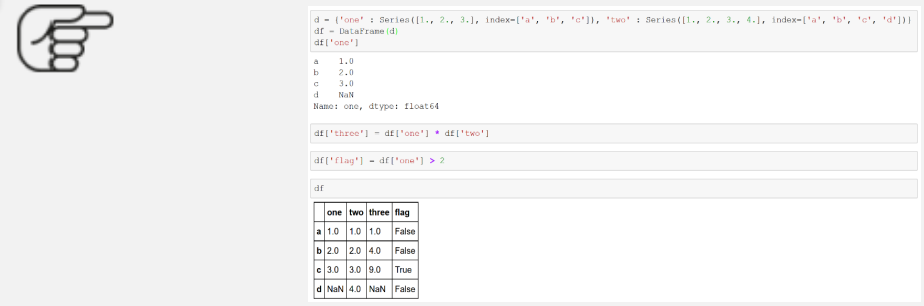
	A	B	C
Primera_Fila	1	2.3	Hola
Segunda_Fila	2	3.4	World

Figura 6.76. Array estructurado

Operaciones sobre dataframes

Manipulación de un dataframe

Un dataframe es como un diccionario de series indexado, por lo que se pueden usar las mismas operaciones utilizadas con los diccionarios (figura 6.77.).



```
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
```

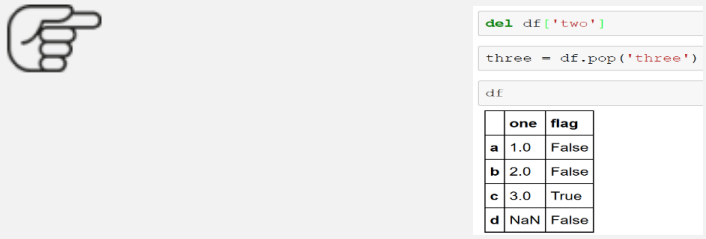
```
df
```

	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 6.77. Manipulación de un dataframe.

Borrado de columnas

Se pueden borrar columnas (figura 6.78.).



```
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 6.78. Borrado de columnas.

Inserción de valores

Cuando se inserta un valor escalar, entonces se propaga a toda la columna (figura 6.79.).

```
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 6.79. Propagación a toda la columna.

Creación de nuevo índice

Cuando se inserta una serie que no tiene el mismo índice, se crea el índice para el dataframe (figura 6.80.).

```
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 6.80. Inserción de una serie con distinto índice.

Selección del punto de inserción

Las columnas, por defecto, se insertan al final, sin embargo, se puede elegir el lugar de inserción usando la función **insert** (figura 6.81.).

```
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 6.81. Inserción de un dataframe.

Indexación/Selección

Selección por columnas

Se puede seleccionar por columnas (figura 6.82.).

```
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 6.82. Selección por columnas.

Selección por etiqueta

Se puede seleccionar por etiqueta (figura 6.83.).

```
df.loc['b']
```

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

Figura 6.83. Selección por etiqueta.

Selección por entero

Selección de fila por entero (figura 6.84.).

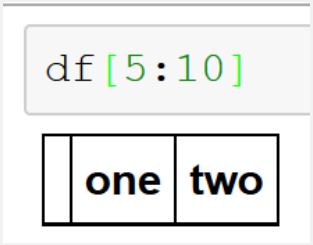
```
df.iloc[2]
```

```
one    3.0
two    3.0
Name: c, dtype: float64
```

Figura 6.84. Selección por entero.

Selección por rangos

Selección por rangos (figura 6.85.).



```
df[5:10]
```

	one	two

Figura 6.85. Selección por rangos.

Alineación y aritmética

Alineación

Cuando se alinea un dataframe, se realiza sobre las columnas y el índice, de manera que se hace la unión de las etiquetas de las columnas y de las filas (figura 6.86.).

```
randn = np.random.randn

df = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])

df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])

df + df2
```

	A	B	C	D
0	-1.193372	-1.091701	-0.241438	NaN
1	-0.322861	-0.219207	-1.502787	NaN
2	-0.716514	-0.474825	-1.195332	NaN
3	-1.460108	-0.822584	0.810185	NaN
4	0.105979	1.475265	-0.512214	NaN
5	1.354064	-0.936150	2.461633	NaN
6	-0.005034	-0.645340	-0.768346	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

Figura 6.86. Alineamiento.

Alineación de operación con dataframe y series

Cuando se operan con dataframe y series, se alinea el índice de las series sobre las columnas del dataframe (figura 6.87.).

```
df - df.iloc[0]
```

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	-0.776563	0.142339	-0.863939	-0.425021
2	-0.701542	0.733010	-0.393288	-1.261082
3	-0.821132	1.455670	1.164793	1.140483
4	0.340844	1.230268	-0.103345	1.719146
5	-0.895138	1.096608	3.161420	-0.162601
6	0.182207	-0.856004	-0.312982	-1.650914
7	-0.024389	-0.003078	1.244290	0.506915
8	-1.023797	1.869494	0.761191	0.539503
9	-1.341046	1.233889	2.306251	1.033930

Figura 6.87. Alineamiento de dataframe y serie.

Operaciones con escalares

Se pueden hacer operaciones con escalares (figura 6.88.).

```
df * 5 + 2
```

	A	B	C	D
0	3.702017	-0.462339	-1.211926	1.422500
1	-0.180798	0.249354	-5.531622	-0.702605
2	0.194305	3.202711	-3.178363	-4.882910
3	-0.403645	6.816011	4.612040	7.124916
4	5.406237	5.688999	-1.728652	10.018231
5	-0.773672	5.020698	14.595174	0.609496
6	4.613051	-4.742357	-2.776835	-6.832070
7	3.580070	-0.477730	5.009525	3.957074
8	-1.416970	8.885133	2.594030	4.120016
9	-3.003215	5.707104	10.319328	6.592150

Figura 6.88. Operaciones con escalares.

Operaciones lógicas

Se pueden hacer operaciones lógicas (figura 6.89.).

```
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 6.89. Operaciones lógicas.

Transposición

Para transponer, se utiliza el atributo T (figura 6.90.).

```
df[:5].T
```

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

Figura 6.90. Transposición.

Visualización

Hay varias formas de visualizar la información de un dataframe:

A partir del nombre

Todos los datos a partir del nombre del dataframe (figura 6.91.).

```
df
```

	A	B	C	D
0	0.340403	-0.492468	-0.642385	-0.115500
1	-0.436160	-0.350129	-1.506324	-0.540521
2	-0.361139	0.240542	-1.035673	-1.376582
3	-0.480729	0.963202	0.522408	1.024983
4	0.681247	0.737800	-0.745730	1.603646
5	-0.554734	0.604140	2.519035	-0.278101
6	0.522610	-1.348471	-0.955367	-1.766414
7	0.316014	-0.495546	0.601905	0.391415
8	-0.683394	1.377027	0.118806	0.424003
9	-1.000643	0.741421	1.663866	0.918430

Figura 6.91. Datos de un dataframe.

Resumen

Un resumen de la información contenida, usando el método info() (figura 6.92.).

```
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 6.92. Datos de un dataframe usando info().

Cadena

Como una cadena usando el método to_string() (figura 6.93.).

```
print(df.to_string())
```

```
          A         B         C         D
0  0.340403 -0.492468 -0.642385 -0.115500
1 -0.436160 -0.350129 -1.506324 -0.540521
2 -0.361139  0.240542 -1.035673 -1.376582
3 -0.480729  0.963202  0.522408  1.024983
4  0.681247  0.737800 -0.745730  1.603646
5 -0.554734  0.604140  2.519035 -0.278101
6  0.522610 -1.348471 -0.955367 -1.766414
7  0.316014 -0.495546  0.601905  0.391415
8 -0.683394  1.377027  0.118806  0.424003
9 -1.000643  0.741421  1.663866  0.918430
```

Figura 6.93. Datos de un dataframe usando to_string().

VI. Resumen

En esta unidad, se han presentado las principales librerías científicas de Python, las cuales se utilizan para realizar análisis de datos.

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

La librería Matplotlib permite la representación gráfica de puntos, pudiéndolos 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 son el título del gráfico o los valores de los ejes.

Por último, la librería Pandas permite manipular los datos en general de una manera simple, lo que facilita operaciones propias del análisis de datos tales como la limpieza o normalización de datos, o bien la selección de determinados datos.

VII. Caso práctico

Se pide:

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 otro denominada "Sueldos_mujeres" que contenga los datos [2300,1600,1980,1900,2150] y cuyos índices sean ["Euskadi", "Murcia", "Madrid", "Barcelona","Córdoba"]
2. Usando las series anteriores, obtén otra serie que contenga la diferencia de sueldos entre mujeres y hombres.
3. Usando la serie anterior que contiene las diferencias de sueldos, obtén las ciudades donde la diferencia de sueldos es mayor de 100 €.

Solución

1)

```
#Solución
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","Cordoba"])
print (sueldosmuj)
```

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

2)

```
#Solución
difsueldos = sueldoshom-sueldosmuj
print(difsueldos)
```

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

3)

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

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

Recursos

Bibliografía

- ***Pandas: powerful Python data analysis toolkit.*** **2017. [En línea] URL disponible en http://pandas.pydata.org/pandas-docs/stable/index.html** :
- ***Tentative Numpy Tutorial.*** **[En línea] URL disponible en http://wiki.scipy.org/Tentative_NumPy_Tutorial** :
- **Matplotlib. [En línea] URL disponible en http://matplotlib.org/** :
- **McKinney, W. *Python for Data Analysis.* EE. UU.: O'Reilly; 2012.** :

Glosario.

- **Axe**: Es un objeto que representa un área donde se puede dibujar.
- **Dataframe**: Es una 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.
- **Matplotlib**: Es una librería de Python para realizar gráficos. Se caracteriza porque es fácil de usar, flexible y se puede configurar de múltiples maneras.
- **Ndarray**: Son arrays multidimensionales donde todos sus elementos son del mismo tipo y están indexados por una tupla de números positivos.
- **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.
- **Pandas**: Es una librería construida sobre Numpy que ofrece estructuras de datos de alto nivel que facilitan el análisis de datos desde Python.
- **Series**: Una serie es un objeto, como un array, que está formado por un array de datos y un array de etiquetas denominado índice.
- **Subgráfico**: Son gráficas formadas por un conjunto de gráficos.