SEAS
CAMPUS SEAS

6 UNIDAD DIDACTICA

Python

6. Librerías de uso común en Python

ÍNDICE

OBJETIVOS

- Uso de tres paquetes de gran utilidad en multitud de aplicaciones.
- Aprender a crear vectores y matrices usando *Numpy*.
- Conocer las funciones disponibles de *Numpy*.
- Aprender a hacer representaciones gráficas de funciones.
- Aprender a crear series y *DataFrames* usando Pandas.
- Conocer las estructuras de datos disponibles en Pandas.
- Utilizar esas estructuras de datos en aplicaciones.

INTRODUCCIÓN

En esta Unidad se van a estudiar tres librerías de gran utilidad en numerosas aplicaciones: *NumPy*, *Matplotlib* y Pandas.

NumPy

Es un módulo básico para el cálculo científico en Python. Ofrece una gran capacidad para operar con matrices de N dimensiones, de álgebra lineal, transformadas de Fourier y cálculo con números aleatorios. Permite trabajar con un gran número de datos minimizando el consumo de memoria y utiliza algoritmos muy optimizados.

Matplotlib

Es una librería de trazado gráfico y visualización que es capaz de producir gráficos en 2D y 3D de alta calidad en diferentes formatos de salida, permitiendo formatos interactivos con el ratón.

Pandas.

Es un paquete de Python que proporciona estructuras de datos rápidas y flexibles. Posee una gran riqueza de funcionalidades para extraer, preparar y analizar datos. Trabaja sobre series y sobre *DataFrames* o tabla de datos.

6.1. Configuración para la importación de módulos

Para poder importar los módulos se debe seleccionar *"Preferences for New Projects"* del menú "*File*" y seleccionar el intérprete de Anaconda que contiene todos los paquetes, como se muestra en la figura 6.1. Se selecciona: Python 3.7 /anaconda3/bin/Python3:

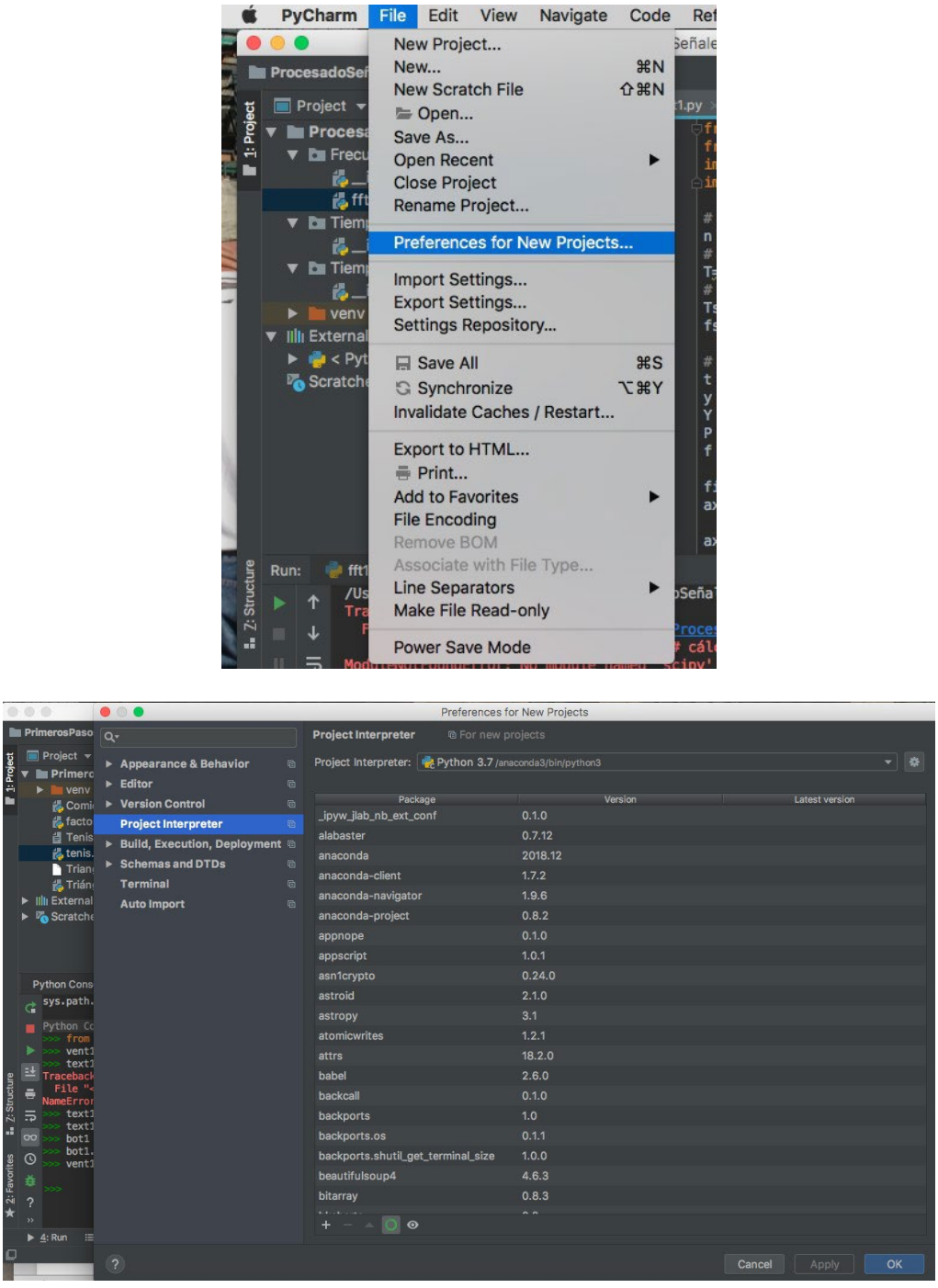

Figura 6.1. Selección del intérprete de Anaconda.

Otra forma de hacerlo consiste en seleccionar PyCharm →Preferences…, como muestra la figura 6.2

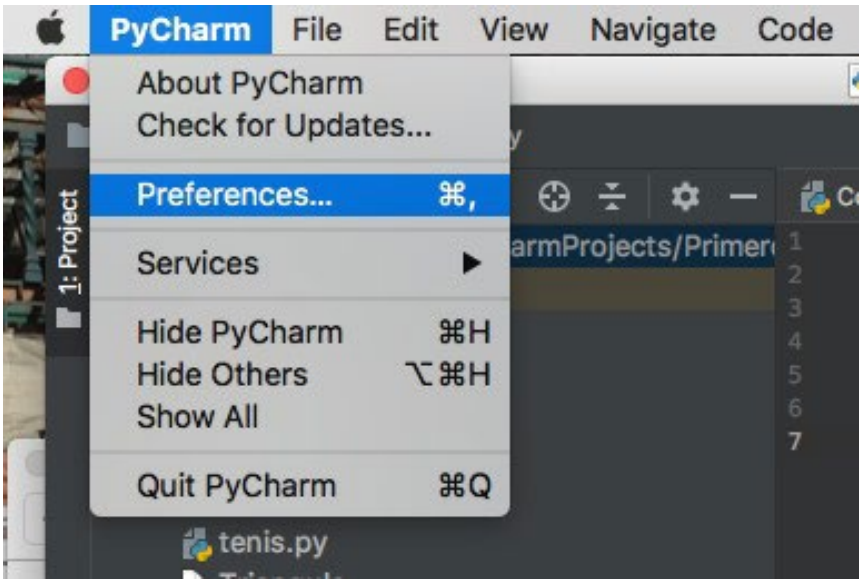

Figura 6.2. Selección de Preferences.

Tras realizar esa operación, nos aparece:

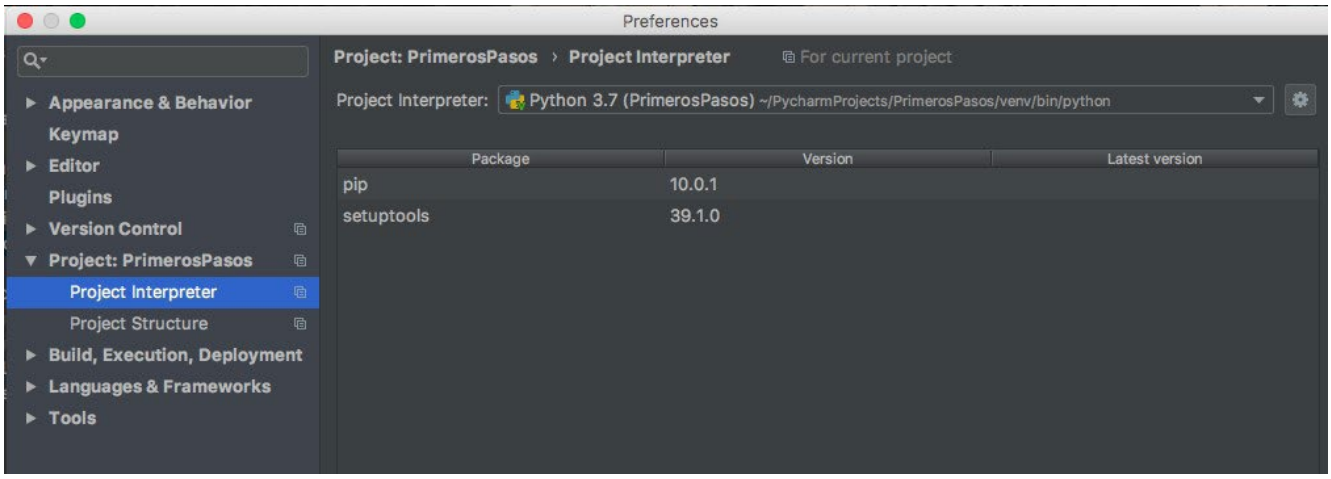

Figura 6.3. Project Interpreter.

Como aparece en la figura 6.4, se selecciona: Python 3.7 /anaconda3/bin/Python3:

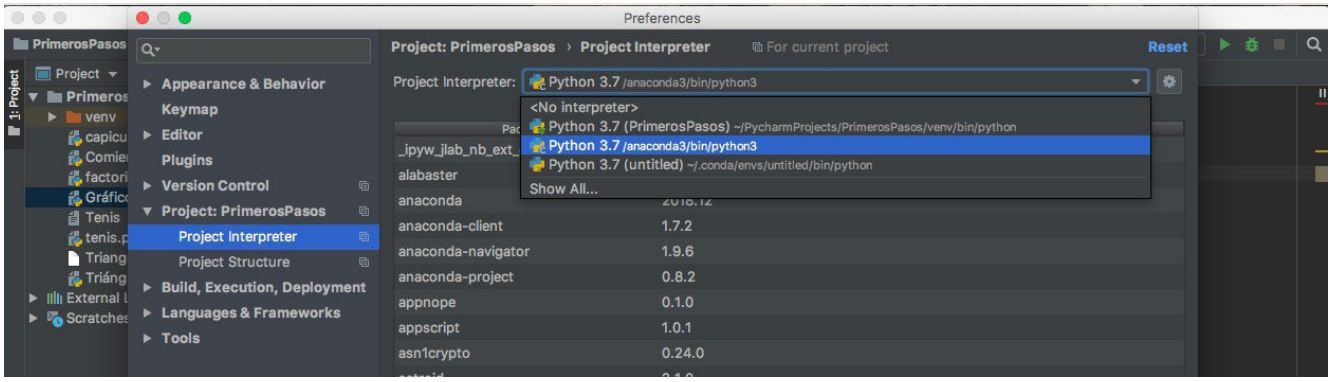

Figura 6.4. Selección de la versión que contiene los paquetes.

y finalmente se pulsa el botón *Apply*.

6.2. NUMPY

Vamos a comenzar con *Numpy* que es una librería usada para trabajar con vectores. La estructura de datos principal con el que trabaja *Numpy* es el vector multidimensional homogéneo (todos los componentes son del mismo tipo). En *Numpy* a las dimensiones se les llama ejes.

Como ya sabemos, lo primero que hay que hacer es importar *Numpy*. Normalmente se le suele cambiar el nombre y llamar con el alias np como se muestra en la figura 6.5 y siguientes.

```
import numpy as np
```

Figura 6.5. Importación de Numpy.

6.2.1. Creación de arrays

Hay varias formas de crear *array*s, es decir, vectores, matrices, etc. Una de ellas consiste en usar la función *array* a la que se le pasa una lista como argumento. El tipo del *array* resultante se deduce del tipo de los elementos que forman parte de la lista. Por ejemplo, en la figura 6.6 se muestra la creación de dos *array*s a (de enteros) y b (de reales).

```
import numpy as np
a = np.array([2, 3, 4])
print(a)
print(a.dtype)
b = np.array([1.2, 3.5, 5.1])
print(b)
print(b.dtype)
```

[2 3 4]

int64

[1.2 3.5 5.1]

float64

Figura 6.6. Uso de la función array:

También se puede pasar una lista de booleanos:

```
xb = np.array([True, False, True, False])
print(xb)
print(xb.dtype)
```

[True False True False]

bool

Figura 6.7. Creación de una lista de booleanos.

o de cadenas de caracteres. Por ejemplo:

```
xcad = np.array(['Ana', 'Pepe', 'Juan'])
print(xcad)
print(xcad.dtype)
```

['Ana' 'Pepe' 'Juan']

<U4

Figura 6.8. Creación de una lista de cadenas de caracteres.

La función **zeros**((filas, columnas)) crea un *array* relleno de ceros, y la función **ones**((filas, columnas)), relleno de unos:

```
z = np.zeros((3,5))      # no olvidar el doble paréntesis
print(z)
```

[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

Figura 6.9. Uso de la función zeros:

```
o = np.ones((3,5), dtype=np.int16)      # no olvidar el
doble paréntesis
print(o)
```

[[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]]

Figura 6.10. Uso de la función ones.

Se pueden aplicar a los vectores los operadores aritméticos de suma (+), resta (-), multiplicación (*) y división (/), operaciones que no trabajan con las matrices, sino con cada componente de cada matriz. Por ejemplo:

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
print('suma: ', x+y)
print('Resta: ', x-y)
print('Producto: ', x*y)       # componente a componente
print('Cociente: ', x/y)       # componente a componente
```

suma: [5 7 9]

Resta: [-3 -3 -3]

Producto: [4 10 18]

Cociente: [0.25 0.4 0.5]

Figura 6.11. Operadores aritméticos con arrays de numpy.

- *arange*()

  Para crear secuencias de números NumPy proporciona una función denominada *arange*, análoga a range, pero que devuelve arrays en lugar de listas. En el ejemplo que sigue se crea un array que comienza por 10, termina en 16 (excluído) con pasos de 2 en 2:

```
v1 = np.arange(10, 16, 2)
print(v1)
```

[10 12 14]

Figura 6.12. Generación de un array de enteros mediante la función arange.

Tiene la forma:

np.*arange*(inicial, final, paso)

genera un *array* de enteros en el intervalo semi-cerrado [inicial, final) a incrementos iguales al "paso".

En la figura 6.13 se muestra el código para generar una matriz de tres filas y cinco columnas en las que distribuye un rango del 0 al 14 (15 números enteros). La forma (filas,columnas) de la matriz se obtienen con la función de forma **shape**. La dimensión con *ndim* (en este caso es una matriz bidimensional), el tipo de los elementos de la matriz se obtiene con **dtype.name**, el número de elementos de la matriz mediante *size* y el tipo (la clase) a la que pertenece la matriz mediante *type*.

```
Import numpy as np
a = np.arange(15).reshape(3,5)
print(a)
print(a.shape)
print(a.ndim)
print(a.dtype.name)
print(a.itemsize)
print(a.size)
print(type(a))
b = np.array([6, 7, 8])
print(type(b))
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
(3, 5)
2
int64
8
15
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

Figura 6.13. Creación de una matriz con arange.

- linspace()

  Una función bastante similar es *linspace*, que tiene la sintaxis:

  np.linspace(valorInicial, valorFinal, numPuntos)

  Que genera un vector de puntos en el intervalo cerrado [valorInicial, valorFinal] con un total de nunPuntos.

  Por ejemplo:

```
v2 = np.linspace(0, 2, 9)
print(v2)
```

```
[0.   0.25  0.5   0.75   1.   1.25   1.5  1.75   2.]
```

Figura 6.14. Generación de un array mediante la función linspace.

NumPy tiene funciones para crear vectores con componentes pseudo-aleatorias. El generador se inicializa utilizando la función *seed* pasándole como argumento un valor cualquiera que queramos. Esto de hace cuando se quiere que no aparezcan siempre los mismos números. A su vez, la función *rand* genera números aleatorios en el intervalo (0, 1) con distribución uniforme. Finalmente, la función *randn* los genera con una distribución normal (gaussiana). Por ejemplo:

```
np.random.seed(123)
xu = np.random.rand(10)
xn = np.random.randn(10)
print(xu)
print(xn)
```

```
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897 0.42310646
 0.9807642  0.68482974 0.4809319  0.39211752]
[ 1.26593626 -0.8667404  -0.67888615 -0.09470897  1.49138963 -0.638902
 -0.44398196 -0.43435128  2.20593008  2.18678609]
```

Figura 6.15. Generación de arrays de números pseudo-aleatorios.

También se pueden generar *array*s de enteros pseudo-aleatorios. Para esto se usa la función ***randint***, que tiene la forma:

np.random.randint(valor menor, valor_mayor, número_de_valores)

Por ejemplo:

```
xe = np.random.randint(2, 10, 15)
print(xe)
```

[2 7 2 9 3 5 6 6 6 8 3 7 8 8 5]

Figura 6.16. Generación de arrays de números enteros pseudo-aleatorios.

Se pueden sustituir componentes como se muestra a continuación, en donde los componentes de xe que toman valores pares se sustituyen por -1.

```
for i in range(1, 15):
  xe[xe % 2 == 0] = -1
  print(xe)
```

[-1 7 -1 9 3 5 -1 -1 -1 -1 3 7 -1 -1 5]

Figura 6.17. Sustitución de componentes de un array.

Igualmente se pueden crear matrices (2D) o *arrays* de rango superior. Por ejemplo:

```
x2d = np.array([[1, 2, 3], [4, 5, 6]])
print(x2d)
print(x2d.ndim)
print(x2d.shape)
print(len(x2d))
```

[[1 2 3]

[4 5 6]]

2

(2, 3)

2

Figura 6.18. Generación de matrices.

Finalmente, se pueden extraer sub-vectores a partir de vectores utilizando el operador "**:**". Así, dado un vector x de *Numpy*, se pueden extraer:

```
x1 = x[inicial:paso;final]

x2 = x[:final]

x3 = x[inicial:]
```

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
x1 = x[3:7]
x2 = x[:4]
x3 = x[7:]
print(x1)
print(x2)
print(x3)
```

[4, 5, 6, 7]

[1, 2, 3, 4]

[8, 9, 10, 11]

Figura 6.19. Extracción de sub-vectores.

6.3. Matplotlib

Matplotlib es el paquete de Python más utilizado para representaciones gráficas de funciones matemáticas y señales. A su vez, matplotlib.pyplot es una colección de comandos que hacen que Matplotlib se parezca a Matlab. Para procesado de señales y representación de funciones será el paquete que importaremos. Cada función realiza una aportación a la representación de una gráfica. Por ejemplo, se crea una gráfica, posteriormente se crea un área de dibujo en la gráfica, se representan líneas en el área de dibujo, se decora la representación con etiquetas, etc.

```
import matplotlib.pyplot as plt
plt.plot([1, 5, 3, 9])
plt.ylabel('Lista de ordenadas')
plt.show()
```

Figura 6.20. Representación de cuatro puntos.

Se observa que al representar los cuatro puntos usando el comando plot sin más, los puntos se unen mediante un segmento rectilíneo.

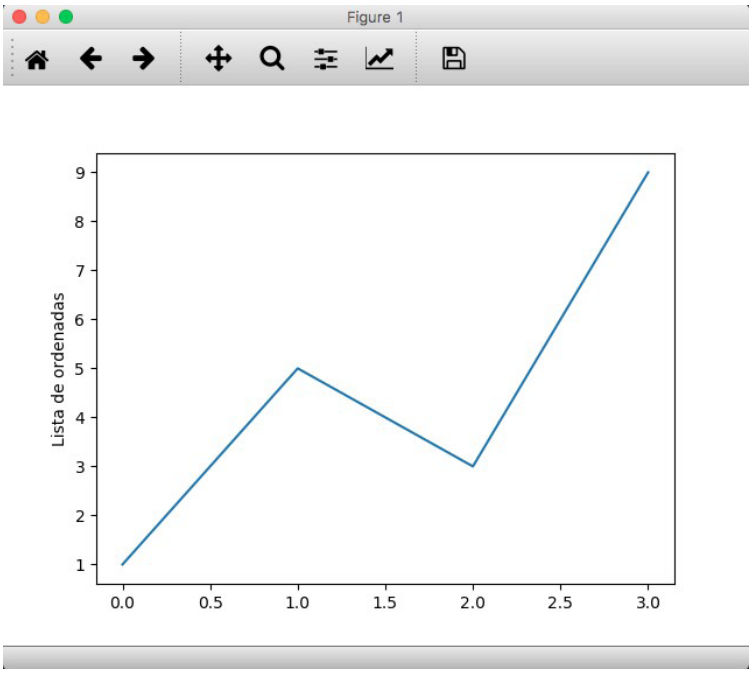

Figura 6.21. Representación de una lista de valores correspondientes a las ordenadas.

Se observa que en el eje x se representan cuatro valores numerados de 0 a 3 por defecto, mientras que los valores del eje y van de 1 a 9. Esto lo hace automáticamente Matplotlib cuando se asume que se representa una lista de valores sin especificar el eje de abcisas. El eje de ordenadas se adapta a los valores máximos y mínimos correspondientes a los valores que se van a representar.

plot es un comando muy versátil que puede tener un número arbitrario de argumentos. Por ejemplo, para representar una función en coordenadas cartesianas rectangulares, se pueden pasar sendas listas correspondientes a las coordenadas de los puntos que se van a representar, en la forma:

```
plt.plot(<lista_abcisas>, <lista_ordenadas>)
```

show() presenta en pantalla las figuras mediante una ventana sobre la que se puede interactuar. Este comando se debe escribir siempre que necesitemos ver la gráfica ya que si no, Python no representará la imagen. Se suele escribir al final para que afecte a todas las figuras del programa.

Por ejemplo:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 8, 27, 64])
plt.ylabel('Lista de ordenadas')
plt.xlabel('Lista de abcisas')
plt.show()
```

Figura 6.22. Código para representar puntos pasando listas de abcisas y ordenadas.

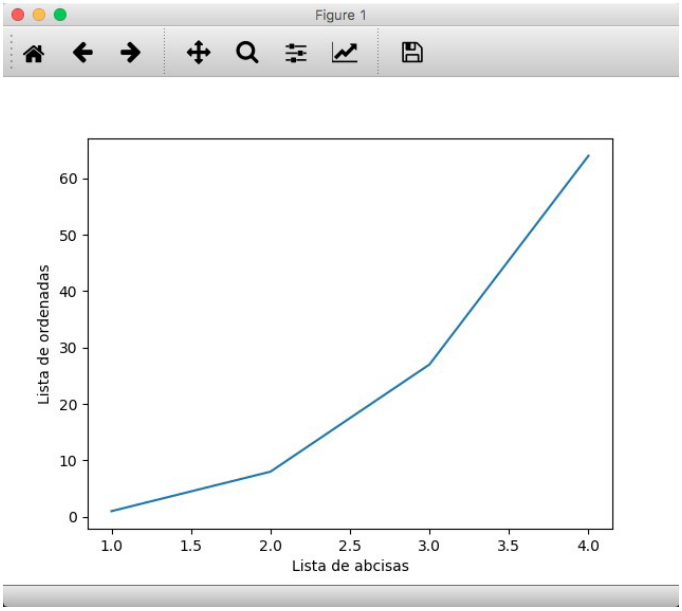

Figura 6.23. Representación de puntos.

Todavía se puede incluir un tercer parámetro que es una "cadena de formato", que especifica el color y el tipo de línea de la representación.

```
plt.plot(<lista_abcisas>, <lista_ordenadas>, <cadena _de_
formato>)
```

La cadena de formato por defecto es 'b-' que es una línea continua de color azul. La letra b indica el color (blue) y el carácter – indica que la representación es una línea continua que se obtiene uniendo mediante segmentos rectilíneos los puntos correspondientes a los valores de la representación.

Vamos a cambiar la representación dibujando ahora los puntos como puntos aislados de color rojo. Para esto se añadirá un nuevo parámetro 'ro', en donde la r indica el color y el carácter o es el símbolo que se usará en la representación de los puntos, en este caso un círculo.

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Figura 6.24. Código para representar puntos aislados y uso del comando axis.

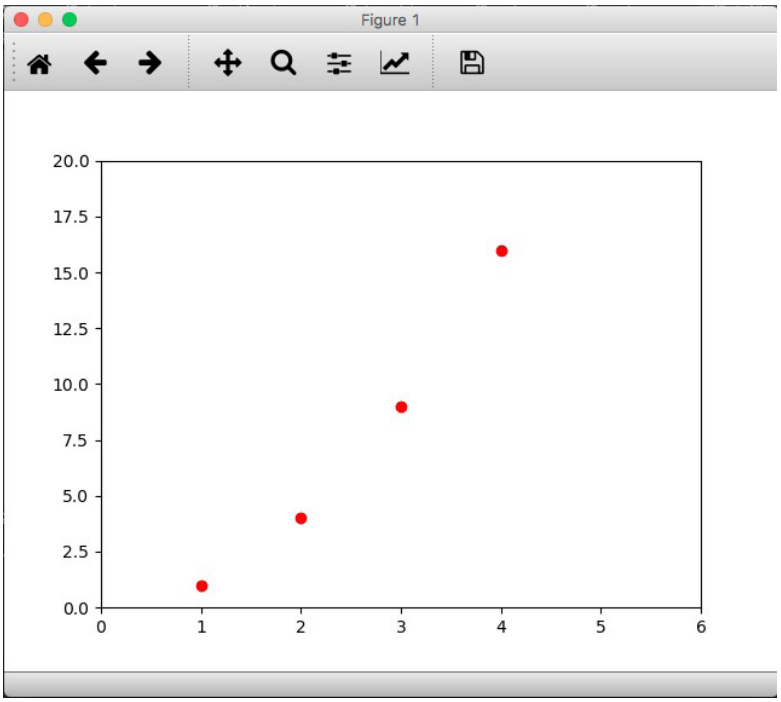

Figura 6.25. Representación de puntos aislados.

En el ejemplo anterior, el comando **axis()** toma como argumento una lista que delimita la región de valores en los que se va a realizar la representación: [xmin, xmax, ymin, ymax].

Como vemos, estas funciones vienen con un conjunto de valores predeterminados que permiten la personalización de todos los tipos de propiedades. En Matplotlib se pueden controlar los valores predeterminados de casi todas las propiedades: tamaño de figura y puntos por pulgada (*dpi, dots per inch*), grosor de línea, color y estilo, ejes y propiedades de cuadrícula, propiedades de texto y fuente, etc. Veamos un ejemplo:

```
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0., 2., 0.1)
plt.plot(t, t**2.0, 'r-', t, t**3, 'bs', t, t**4, 'g^')
plt.xlabel('tiempo (seg)')
plt.ylabel('amplitud')
plt.title('potencias de t')
plt.show()
```

Figura 6.26. Código representado en la figura 6.25.

En donde el carácter "s" representa cuadrados (Square) y el "^" representa triángulos. El primer carácter como siempre es el color. Los tipos más comunes son:

'-'	línea sólida.
'- -'	línea a rayas.
'-.'	línea con puntos y rayas.
':'	línea punteada.

En cuanto a los colores:

'b'	azul.
'g'	verde.
'r'	rojo.
'c'	cián.
'm'	magenta.
'y'	amarillo.
'k'	negro.
'w'	blanco.

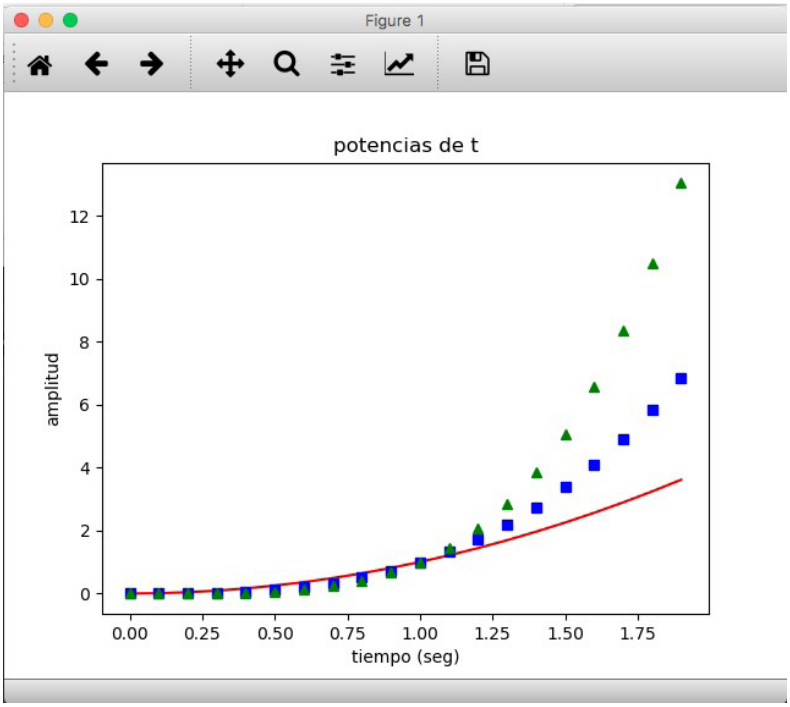

Figura 6.27. Representación del código de la figura 6.25.

- Propiedades de línea.

  También se puede controlar la anchura de la línea con el parámetro *linewidth*. Si repetimos la representación anterior poniendo una anchura de línea de 2.5, queda:

```
plt.plot(t, t**2.0, 'r-', t, t**3, 'bs', t, t**4, 'g^',
linewidth=2.5)
```

Figura 6.28. Cambio de la anchura de la línea.

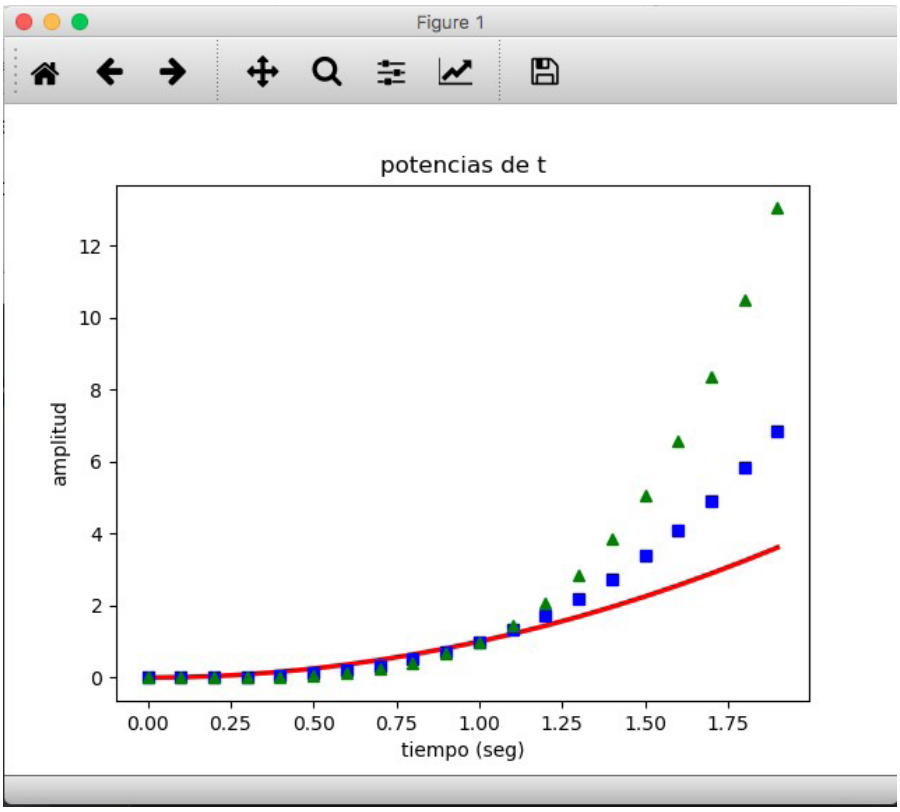

Figura 6.29. Representación con la nueva anchura de línea.

- Creación de figuras.

  La función:

  figure(num, figsize, dpi, color, facecolor, edgecolor, frameon), crea una nueva figura. Se puede utilizar sin argumentos y devuelve un identificador a la figura.

  - num es la numeración de la figura. Si num = None, se numeran automáticamente.
  - figsize = w, h son tuplas en pulgadas que dan el tamaño de la figura.
  - *dpi* es la resolución en puntos por pulgada.
  - *facecolor* es el color del rectángulo de la figura.
  - *edgecolor* es el color del perímetro de la figura.
  - *frameon* si es falso elimina el marco de la figura.

En la figura 6.30 se muestra el código para representar en una figura 1, tres períodos de una función de tipo coseno. Por tanto, el eje de abcisas irá de 0 a 6π y se van a tomar 300 puntos en total (100 por período).

```
from numpy import pi
import matplotlib.pyplot as plt
x = np.linspace(0, 6*pi, 300)
y = 5*np.cos(x)
plt.figure(1)
plt.plot(x, y)
plt.show()
```

Figura 6.30. Código para representar tres períodos coseno.

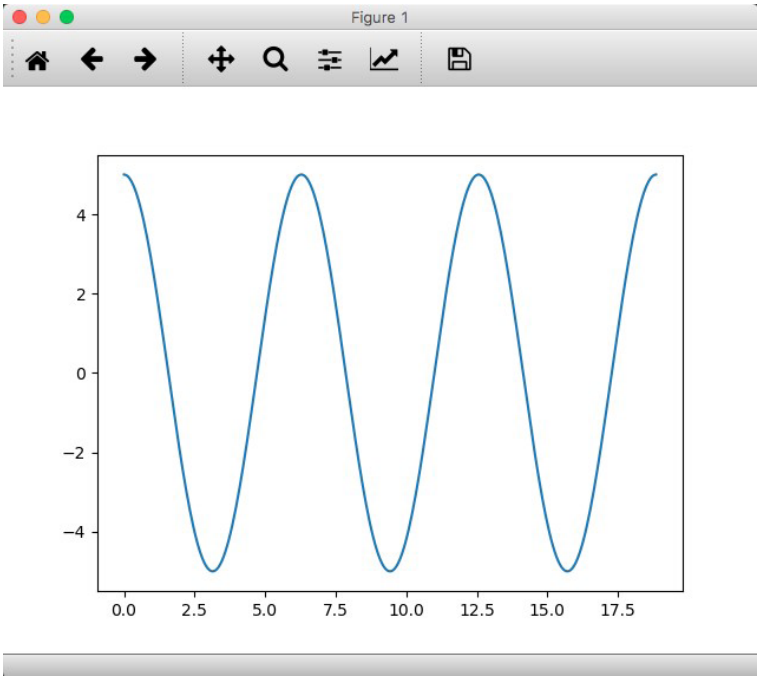

Figura 6.31. Representación de tres períodos coseno.

Si se quiere representar varias funciones en una misma figura, basta con representarlas en el contexto de figure(). Es una buena práctica cerrar las figuras con la función *close*.

```
t = np.linspace(-4, 4, 1000)
plt.figure()
x1 = 5*np.cos(2*np.pi*6*t)
x2 = 3*np.sin(2*np.pi*4*t)
x3 = np.cos(2*np.pi*2*t)
plt.plot(t, x1,'r-', t, x2, 'g-', t, x3, 'b-')
plt.show()
plt.close()
```

Figura 6.32. Código para representar tres funciones en una misma figura.

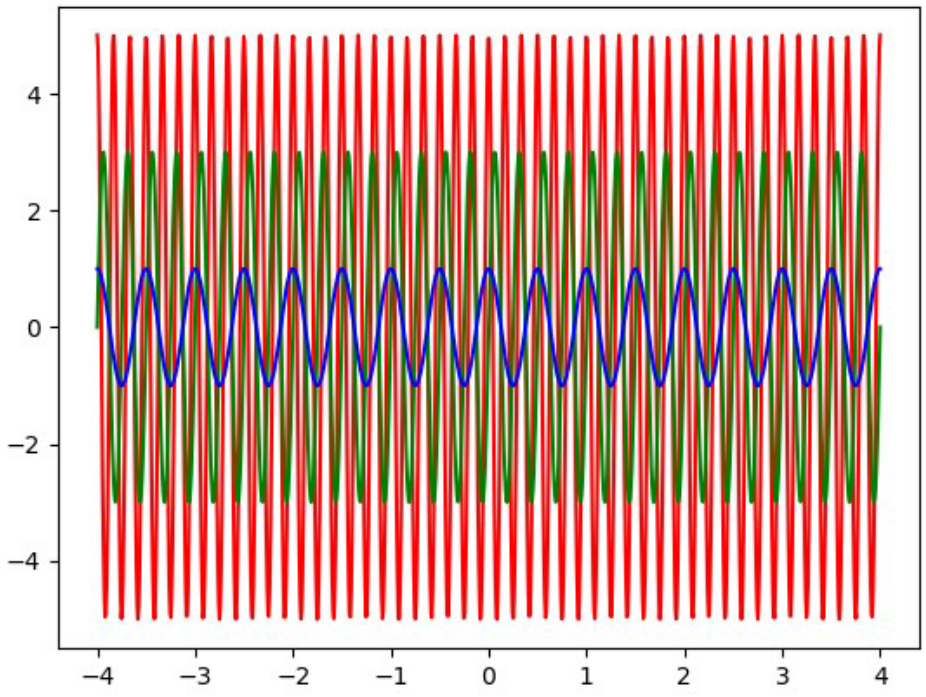

Figura 6.33. Representación de las tres gráficas en una misma figura.

6.3.1. Representación de múltiples figuras y ejes

Vamos a escribir una función "f" para generar un vector que represente una función sinusoidal amortiguada exponencialmente a la que se pasará como parámetro un vector de tiempos. Devolverá el vector cuyos valores correspondan a la función y su tamaño será el mismo que el del vector de tiempos pasado como parámetro de entrada. Además se van a definir dos ejes de tiempos t1 y t2 y se va a proceder a realizar distintas representaciones. Se usará la función *subplot* para especificar la disposición de las distintas gráficas que se van a representar y el número correspondiente a cada gráfica. Su sintaxis es:

subplot(filas, columnas, numGráfica)

Por ejemplo, subplot(2,1,1) prevé la representación de dos gráficas dispuestas en dos filas y una columna. Cuando el tercer parámetro valga la unidad, indica que se va a proceder a representar la primera gráfica que será la gráfica superior. Cuando tome valor 2 se representará la inferior.

```
import numpy as np
import matplotlib.pyplot as plt
def f(t):
  tao = 0.4
  f = 1/(tao)
  return np.exp(-t/tao)*np.cos(2*np.pi*f*t)

t1 = np.arange(0., 2., 0.02)
t2 = np.arange(0., 2., 0.05)

plt.figure(1)
plt.subplot(211)
plt.title('sinusoidal amortiguada (superior), sinusoidal
(inferior)')
plt.plot(t1, f(t1), 'bo', t2, f(t2),'k')
plt.ylabel('amplitud')
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r-')
plt.xlabel('tiempo (seg)')
plt.ylabel('amplitud')
plt.show()
```

Figura 6.34. Programa para representar dos funciones.

La función: plt.plot(t1, f(t1), ‘bo’, t2, f(t2),’k’) superpone dos representaciones de la función f(t). La primera usa el eje de tiempos t1 y se representa en forma de círculos de color azul. La segunda representa la misma función f(t) pero con el eje de tiempos t2, haciendo la representación en color negro.

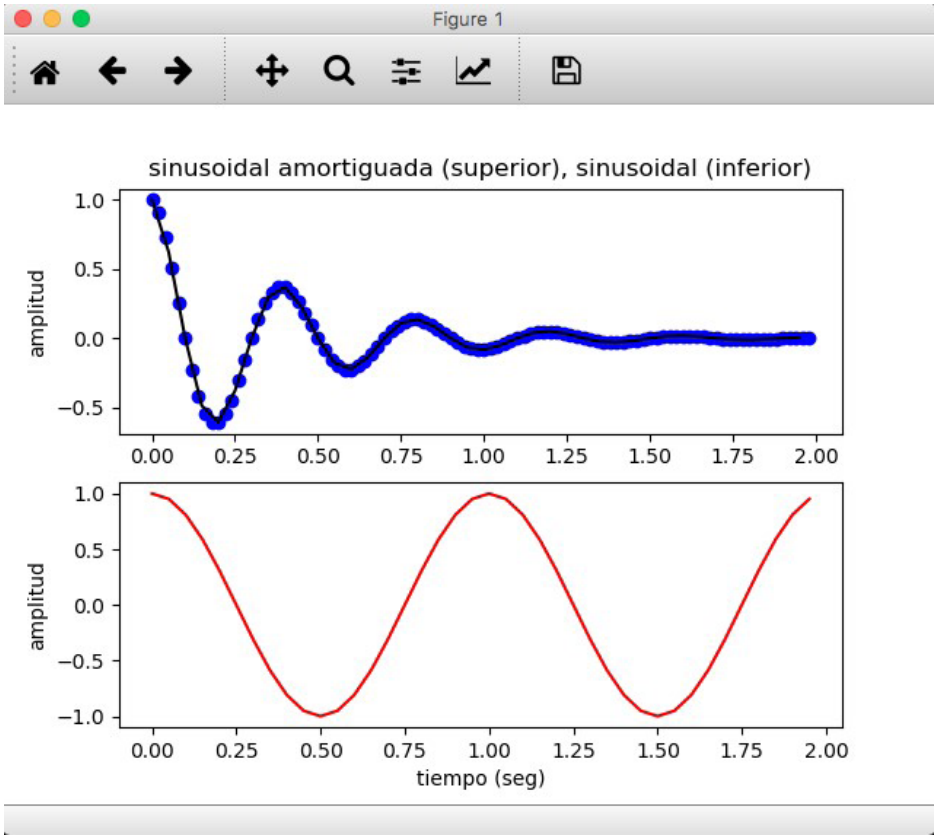

Figura 6.35. Representación de dos funciones.

La función f(t) genera una señal sinusoidal amortiguada exponencialmente. Se le pasa un vector t que representa un eje de tiempos.

En el código de la figura 6.36 únicamente se cambia el paso entre cada dos puntos consecutivos del eje de tiempos t2. La representación se ve en la figura 6.37.

```
import numpy as np
import matplotlib.pyplot as plt
def f(t):
  tao = 0.4
  f = 1/(1*tao)
  return np.exp(-t/tao)*np.cos(2*np.pi*f*t)

t1 = np.arange(0., 2., 0.02)    # de 0 a 2 a intervalos
de 0,02 en 0,02
t2 = np.arange(0., 2., 0.1)     # de 0 a 2 a intervalos
de 0,1 en 0,1

plt.figure(1)
plt.subplot(211)
plt.title('sinusoidal amortiguada (superior), sinusoidal
(inferior)')
plt.plot(t1, f(t1), 'bo', t2, f(t2),'k')
plt.ylabel('amplitud')
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r-')
plt.xlabel('tiempo (seg)')
plt.ylabel('amplitud')
plt.show()
```

Figura 6.36. Código modificando en paso t2.

En las gráficas en negro y rojo se observa claramente que la representación es mediante segmentos rectilíneos que unen cada punto con el siguiente. Si los puntos están muy próximos, da la sensación de continuidad.

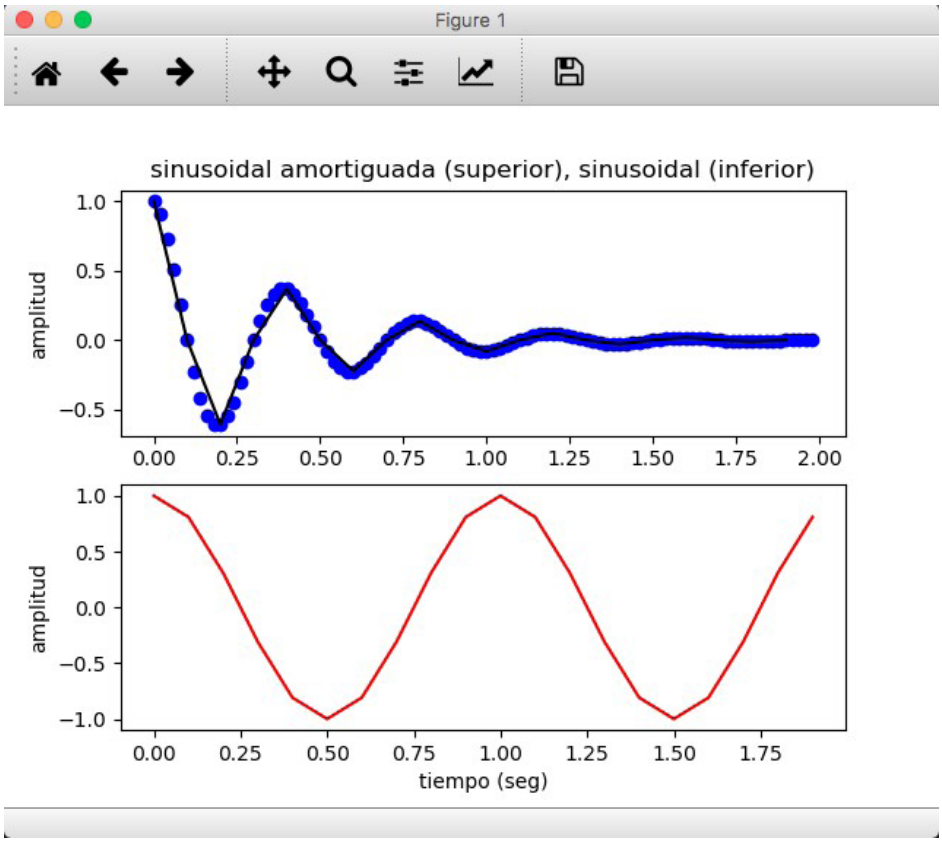

Figura 6.37. Representación con distinto paso.

Se puede crear un número arbitrario de gráficas y ejes en una misma figura. Los ejes se pueden colocar manualmente mediante el comando **axes**(), que permite especificar su posición (izquierda, inferior, ancho, alto)

Se pueden crear múltiples figuras usando múltiples llamadas a **figure**(). Cada figura puede contener ejes y gráficas. Por ejemplo:

```
import matplotlib.pyplot as plt
plt.figure(1)
plt.subplot(211)
plt.plot([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
plt.title('lineal (superior) y lineal a tramos
(inferior)')
plt.ylabel('ordenadas')
plt.subplot(212)
plt.plot([4, 5, 6, 9, 10, 11, 14, 15, 18, 20])
plt.xlabel('abcisas')
plt.ylabel('ordenadas')
plt.show()

plt.figure(2)
plt.plot([4, 5, 6, 8, 9, 10, 15, 16, 17, 22, 23])
plt.title('figura 2')
plt.xlabel('abcisas')
plt.ylabel('ordenadas')
plt.show()

plt.figure(1)
plt.subplot(211)
plt.plot([1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
plt.title('Figura múltiple')
plt.ylabel('ordenadas')
plt.subplot(212)
plt.plot([1., 1.2, 1.3, 0.4, 0.6, 1.6, 1.7, 1.8, 0.9,
2.0, 2.1, 2.2 ])
plt.xlabel('abcisas')
plt.ylabel('ordenadas')
plt.show()
```

Figura 6.38. Código para representar varias gráficas en una misma figura.

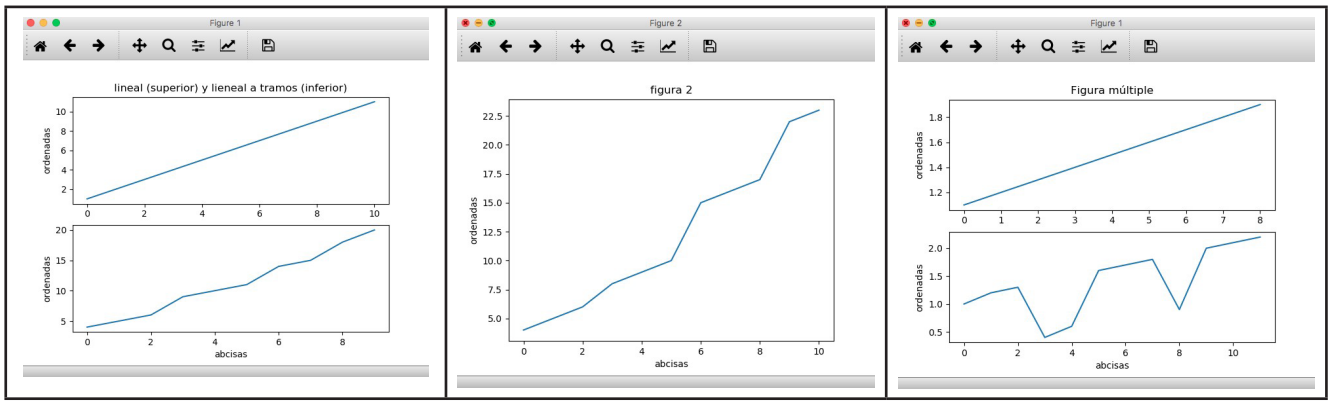

Figura 6.39. Representación de las gráficas del código de la figura 6.38.

Para que una figura deje de ser actual se usa el comando **clf**() y para que lo sean sus ejes **cla**(), de "*clear figure*" y "*clear axis*".

Si está haciendo muchas figuras, debe tener en cuenta una cosa más: la memoria requerida para una figura no se libera completamente hasta que la figura se cierra explícitamente con **close**(). Eliminar todas las referencias a la figura y/o usar el administrador de ventanas para eliminar la ventana en la que aparece la figura en la pantalla, no es suficiente, ya que Pyplot mantiene las referencias internas hasta que se llama a close().

6.3.2. Representación de texto

Se puede usar el comando **text**() para añadir texto en una posición arbitraria. A su vez, como ya hemos visto, los comandos **xlabel**(), **ylabel**() y **title**() se usan para añadir texto en las posiciones indicadas por el comando. En el ejemplo que sigue se van a generar 6000 números aleatorios con distribución normal y se va a representar su histograma, es decir el número de puntos que se generan dentro de cada uno de los intervalos en que se divide el eje de abcisas. La anchura de cada intervalo es 1/200=0,005. El valor de alpha es proporcional a la intensidad del color verde de la representación.

```
Import numpy as np
import matplotlib.pyplot as plt
np.random.seed(123)
mu, sigma = 50, 10      # valor medio y desviación típica
x = mu + sigma * np.random.randn(6000)   # distribución
normal
num_bins = 50       # número de subintervalos del eje de
abcisas
n,bins,patches=plt.hist(x,   num_bins,   density   =   1,
facecolor = 'g', alpha=0.75)
# gaussiana que mejor se adapta
g     =     ((1/(np.sqrt(2*np.pi)*sigma))*np.exp(-0.5*(1/
sigma*(bins – mu))**2))
plt.plot(bins, g, '–')
print('Valores de n: ', n)          # proporción de valores
generados en cada intervalo
print('Valores de bins', bins)      # intervalos generados
print('Valores de patches', patches)
plt.xlabel('Valores')
plt.ylabel('Probabilidad')
plt.title('Histograma')
m = max(n)
plt.text(20, .85*m, r'$\mu=50, sigma=10$')
plt.axis([10, 90, 0, 1.1*m])
plt.grid(True)
plt.show()
```

Figura 6.40. Uso de diversos comandos de texto:

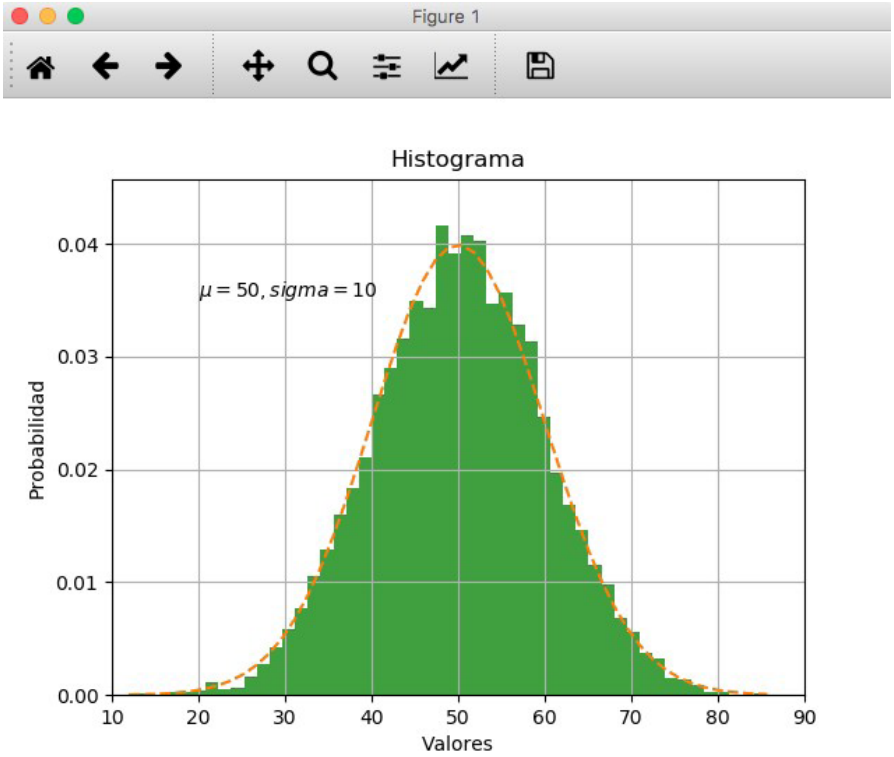

Figura 6.41. Gráfica correspondiente al código de la figura 6.40:

Se puede cambiar el tamaño y color de las etiquetas de los ejes de abcisas (igual de ordenadas mediante ylabel) mediante:

```
t=plt.xlabel('Valores', fontsize=14, color='red')
```

Se pueden colocar leyendas (etiquetas) en un gráfico con el comando *legend*:

```
legend(etiquetas, posición)
```

Cuando se representan varias curvas simultáneamente este comando identifica a cada una. Se puede utilizar sin argumentos si en cada función plot se ha incluido el argumento 'label', que es un texto para identificar la curva. Por ejemplo:

```
plt.legend('etiqueta1', 'etiqueta2', loc = 'upper left')
```

6.3.3. Uso de expresiones matemáticas en el texto

Matplotlib acepta expresiones de ecuaciones TeX en cualquier expresión textual. Por ejemplo, para escribir la expresión en el título, se puede escribir una expresión TeX entre símbolos $:

```
plt.title(r'$\sigma_i=15$')
```

indicando la r que figura al principio que es una cadena *raw* de forma que no se deben tratar las contra-barras como secuencias de escape.

Por ejemplo, vamos a representar dos señales sinusoidales amortiguadas exponencialmente, dibujadas con distinto color, en las que ponemos etiquetas identificativas.

```
frec = 2                                # frecuencia
t = np.linspace(0, 2, 1000)             # eje de tiempos
x1 = np.exp(-t)*np.sin(2*np.pi*frec*t)      # función a
representar
x2 = np.exp(-2*t)*np.sin(2*np.pi*2*frec*t)      # la otra
función
plt.figure()                            # creamos la figura
plt.plot(t, x1, 'k-', linewidth = 1.5, label = 'x1')
plt.plot(t, x2, 'r--', linewidth = 1, label = 'x2')
plt.legend(loc = 2)


# añadimos las etiquetas poniendo en Latex "mu" para
representar microegundos
plt.xlabel(r"$x (\mu seg) $", fontsize = 10, color = (1,
0, 0))
plt.ylabel('Amplitud')


# añadimos texto
plt.text(x = 1.5, y = 0.6, s = u'frec = 2', fontsize = 10)
plt.grid(True)
plt.grid(color = '0.5', linestyle = '--', linewidth = 1)
plt.axis('tight')
plt.title('Respuesta al impulso', fontsize = 10, color =
'0.75')
plt.savefig('RespuestaAlImpulso.png')
plt.show()
```

Figura 6.42. Código para representar gráficas con expresiones en Latex y con etiquetas.

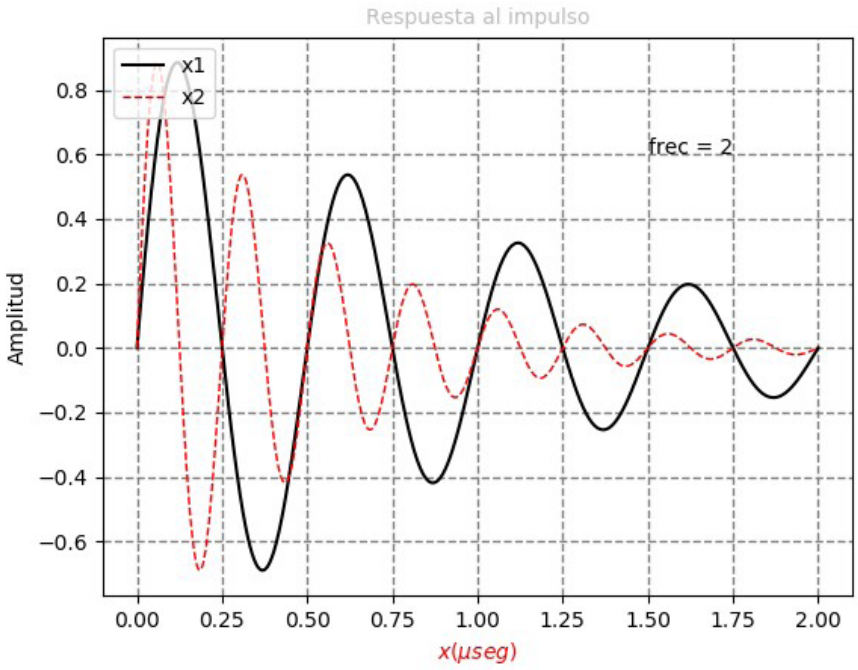

Figura 6.43. Gráfica correspondiente al código de la figura 6.42:

6.3.4. Representaciones con escalas no lineales

En algunas aplicaciones se prefiere usar ejes cuya escala no sea lineal. Por ejemplo, cuando se quiere graduar el eje de abcisas en una magnitud que tiene muchos órdenes de magnitud (valores pequeños y grandes simultáneamente) se prefiere usar representaciones semi-logarítmicas. Para esto se usa:

```
plt.xscale('log')
```

A continuación se muestra un ejemplo de datos representados en tres escalas diferentes: una lineal, otra semi-logarítmica (eje x logarítmico y eje y lineal) y finalmente, una logarítmica (ambos ejes logarítmicos).

```
# escalas no lineales
plt.figure(1)
x = np.arange(0, 100, 0.01)
y1 = 10 * x
y2 = 2*(x**2)

# representación lineal
plt.subplot(3,1,1)
plt.plot(x, y1)
plt.plot(x, y2)
plt.title('Lineal')
plt.grid()

# representación semilogarítmica
plt.subplot(3,1,2)
plt.xscale('log')
plt.plot(x, y1)
plt.plot(x, y2)
plt.title('Semilogarítmica')
plt.grid()

# representación logarítmica
plt.subplot(3,1,3)
plt.xscale('log')
plt.yscale('log')
plt.plot(x, y1)
plt.plot(x, y2)
plt.title('Logarítmica')
plt.grid()
plt.show()
```

Figura 6.44. Código para representar en tres escalas.

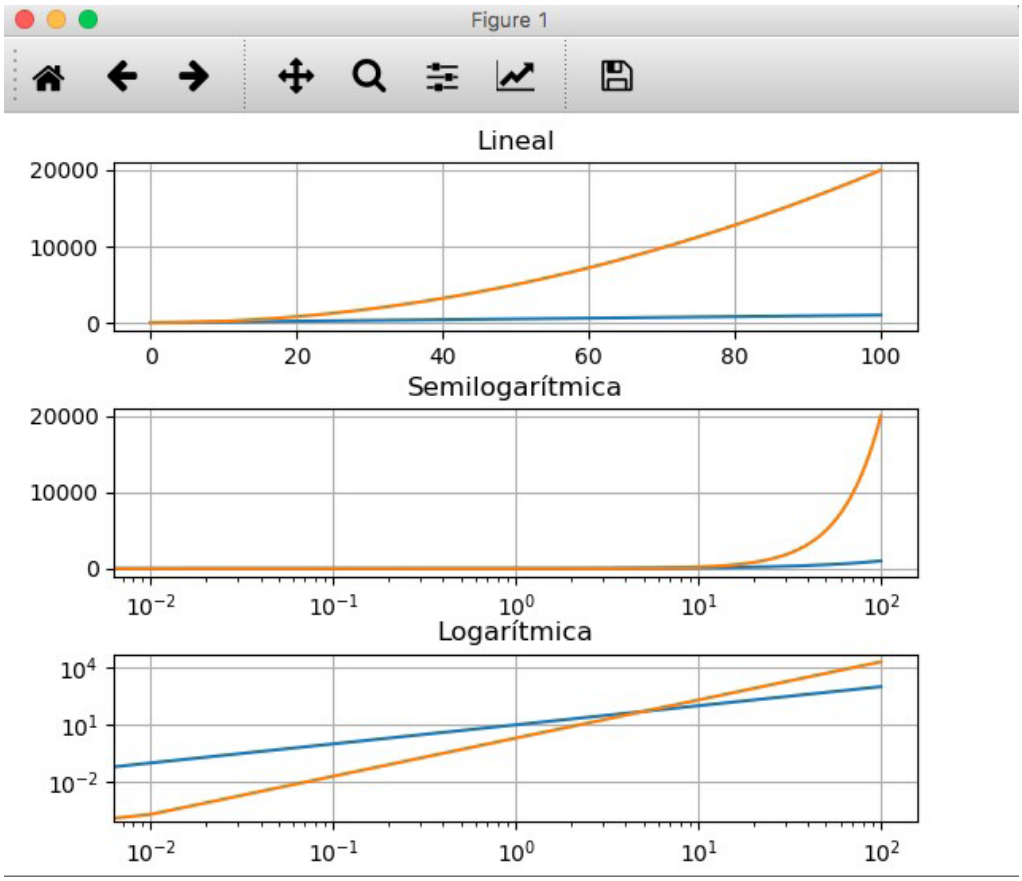

Figura 6.45. Representación en tres escalas: lineal, semi-logarítmica y logarítmica.

6.4. Pandas

Pandas es un paquete de Python que se usa para manipular y analizar datos. Pandas utiliza a *Numpy*. Contiene dos estructuras de datos y operaciones para operar con tablas numéricas y series temporales. Para utilizar pandas, hay que importarlo con la sentencia:

```
import pandas
```

Frecuentemente con objeto de abreviar se puede importar como:

```
import pandas as pd
```

En Pandas se utilizan dos tipos de datos:

- **Series,** que son vectores de valores indexados basados en los vectores de NumPy, pero más flexibles y potentes.
- ***DataFrames***, que son tablas bidimensionales (o multidimensionales) de datos organizados en forma de filas y columnas, indexados mediante etiquetas: "Etiquetas de Variables" (las columnas) y "Etiquetas de Individuos" (las líneas). Un *DataFrames* se puede comparar con una hoja de cálculo Excel.

En la figura se muestran la estructura de datos de Pandas. Cada columna equivale a una Serie, por lo que un *DataFrame*s se puede considerar como un conjunto de Series. Las filas se referencian por medio de un índice "Index".

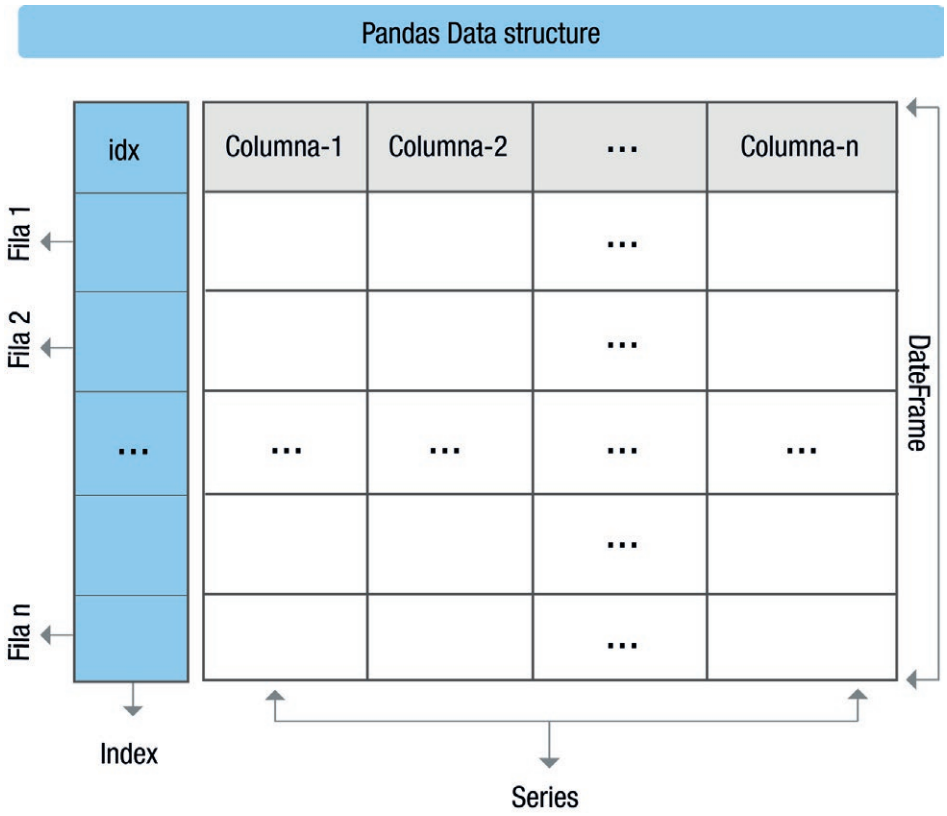

Figura 6.46. Estructura de los datos en Pandas.

A partir de esas estructuras se pueden leer los datos procedentes de un fichero tabulado o escribirlos en un fichero. También se pueden trazar gráficos utilizando la librería Matplotlib. En primer lugar vamos a ver las series que son parecidas a los *array*s y a continuación se verán los *dataframe*s. El manejo de ficheros se deja para la siguiente unidad.

6.4.1. Series

Este tipo de datos es el más básico de Pandas ya que es un *array* unidimensional (vector) cuyas componentes pueden ser de cualquier tipo (heterogéneo). Los índices no necesariamente tienen que ser enteros, por lo que se pueden ver como una estructura compuesta por dos *array*s: uno que hace de índice y el otro que contiene los datos.

6.4.1.1. Creación de las series

La forma de crear una serie es[1] mediante una llamada a la función **series** en la forma:

```
s1 = pd.Series(datos, index=índices)
```

Los datos son un nd*array* y, como es lógico, el índice debe tener la misma longitud que los datos. Si no se pasa ningún índice se crea un de enteros con valores desde 0 a len(data)-1. Los datos pueden ser un *array* de *Numpy*, una lista, o un diccionario.

Vamos a crear series a partir de determinados tipos de datos. En primer lugar vamos a partir de una lista.

Creación de una serie a partir de una lista de datos.

a. Serie cuyos datos son caracteres a partir de un *array* de *Numpy*:

```
# serie a partir de un array de numpy

s0 = pd.Series(np.array(['a', 'b', 'c', 'd', 'e']))
```

b. Serie numérica entera:

```
# serie numérica entera a partir de una lista

s1 = pd.Series([1, 2, 3, 4, 5])
```

c. Serie numérica de números reales:

```
# serie numérica de reales a partir de una lista

s2 = pd.Series([1.1, 2.2, 3.3, 4.4, 5.5])
```

d. Serie numérica especificando el tipos:

```
# serie de reales a partir de una lista, especificando
```

1 En realidad lo que se hace es crear un objeto de clase Series. Esto se verá más adelante al estudiar la Programación Orientada a Objetos.

```
el tipo

s3 = pd.Series([1, 2, 3, 4, 5], dtype=float)


# serie entera a partir de una lista especificando el tipo

s4 = pd.Series([1, 2, 3, 4, 5], dtype=np.int8)
```

e. Serie especificando índices:

```
# serie numérica especificando índices

s5 = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c',
'd', 'e'])
```

Se puede acceder a los datos de una serie a través de su posición o de los índices. En el siguiente ejemplo se muestra la creación de las series anteriores y el acceso a sus componentes.

f. Asociación de índice y serie.

También se puede crear una lista con los datos y otra con los índices. Posteriormente ambas listas se asocian. Por ejemplo, vamos a cambiar el índice de la serie s1.

```
# cambio de índice

# vamos a crear un índice distinto para s1

ind = ['impar1', 'par1, 'impar2', 'par2', 'impar3']

# asociamos el índice a la serie

s1.index = ind
```

g. Creación de una serie a partir de un diccionario.

```
# creación de una serie a partir de un diccionario

dic = {'d':4, 'c':3, 'b':2, 'a':1}

s6 = pd.Series(dic)
```

A continuación se muestra el código anterior junto con los resultados obtenidos.

```
import pandas as pd

import Numpy as np


# serie a partir de un array de Numpy
```

```
s0 = pd.Series(np.array(['a', 'b', 'c', 'd', 'e']))
print('Serie a partir de un array de Numpy')
print(s0)
```

Serie a partir de un *array* de *Numpy*:

0 a

1 b

2 c

3 d

4 e

dtype: object

```
# serie numérica entera a partir de una lista
s1 = pd.Series([1, 2, 3, 4, 5])
print('Serie numérica entera a partir de una lista')
print(s1)
```

Serie numérica entera a partir de una lista:

0 1

1 2

2 3

3 4

4 5

dtype: int64

```
# serie numérica de reales a partir de una lista
s2 = pd.Series([1.1, 2.2, 3.3, 4.4, 5.5])
print('Serie numérica de reales a partir de una lista')
print(s2)
```

Serie numérica de reales a partir de una lista:

0 1.1

1 2.2

2 3.3

3 4.4

4 5.5

dtype: float64

```
# serie numérica de reales a partir de una lista, especificando el tipo

s3 = pd.Series([1, 2, 3, 4, 5], dtype=float)

print('Serie de reales a partir de una lista, especificando el tipo')

print(s3)
```

Serie de reales a partir de una lista, especificando el tipo

0 1.0

1 2.0

2 3.0

3 4.0

4 5.0

dtype: float64

```
# serie numérica entera a partir de una lista especificando el tipo

s4 = pd.Series([1, 2, 3, 4, 5], dtype=np.int8)

print('Serie entera a partir de una lista especificando el tipo')

print(s4)
```

Serie entera a partir de una lista especificando el tipo:

0 1

1 2

2 3

3 4

4 5

dtype: int8

```
# serie numérica especificando índices

s5 = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd',
'e'])

print('serie numérica especificando índices')

print(s5)
```

serie numérica especificando índices

a 1

b 2

c 3

d 4

e 5

```
# cambio de índice

# vamos a crear un índice distinto para s1

ind = ['impar1', 'par1, 'impar2', 'par2', 'impar3']

# asociamos el índice a la serie

s1.index = ind

print('Cambiamos el índice')

print(s1)
```

Cambiamos el índice

impar1 1

par1 2

impar2 3

par2 4

impar3 5

dtype: int64

Creación de una serie a partir de un diccionario

```
# creación de una serie a partir de un diccionario
dic = {'d':4, 'c':3, 'b':2, 'a':1}
s6 = pd.Series(dic)
print('A partir de un diccionario')
print(s6)
```

A partir de un diccionario

d 4

c 3

b 2

a 1

dtype = int64

Figura 6.47. Código del ejemplo.

En el ejemplo anterior todos los componentes de las series eran del mismo tipo. Sin embargo, pueden ser de distintos tipos, solo que en este caso serán tratados como objetos por lo que de momento no vamos a verlo.

Creación de una serie con valores repetidos

Veamos cómo crear una serie con valores repetidos. Se puede hacer de dos formas utilizando la función *range*():

- pd.Series(0, index = range(5)) # crea una serie con 5 ceros
- pd.Series(0, range(5)) # crea una serie con 5 ceros

Como ya se puede suponer, una serie se comporta de forma similar a un *array* o a un diccionario.

6.4.1.2. Pertenencia de una componente a una serie

El operador **in** permite obtener si un índice determinado está en una serie. Por ejemplo:

```
      if 'c' in s5:    # devuelve True si el índice 'c'
pertenece a la serie s5.
```

Hay que tener en cuenta que el operador in no testea la pertenencia de un dato a una serie, sino a su índice. Por ejemplo:

```
'a' in s5                    # True

1 in s5                      # False

'a' on s6a.values    # False  COMPROBAR
```

6.4.1.3. Acceso a los datos de la serie

En el caso de los vectores de *Numpy* el índice sin números enteros que comienzan por el cero para la primera componente. Sin embargo, las series de Pandas son mucho más flexibles, ya que se pueden dar nombres a las componentes de las series. Eso hace que se pueda acceder a los datos de una serie a través de sus índices o por su posición. Así, en las series que se han definido en el código de la figura 6.45, el acceso a sus componentes podría ser (no olvidar que en la serie s1 se han modificado los índices:

```
print('Desde la serie s0: ', s0[2])                  #  tercer
componente de s0

print('Desde la serie s1: ', s1[2])                  #  tercer
componente de s1

print('Desde la serie s1: ', s1['impar2'])  # tercer componente
de s1

print(type(s1))

print('Desde la serie s2: ', s2[2])                  #  tercer
componente de s2

print('Desde la serie s3: ', s3[2])                  #  tercer
componente de s3

print('Desde la serie s4: ', s4[2])                  #  tercer
componente de s4

print('Desde la serie s5: ', s5['c'])      # tercer componente
de s5
```

Finalmente, la función **rank** devuelve el número de componente en que aparece cada valor en orden creciente. Si un número está repetido, da el valor medio de las componentes en que aparece.

6.4.1.4. Funciones que operan sobre series

Existen multitud de funciones que se pueden aplicar a las series. De hecho, en el ejemplo anterior se han aplicado a s1 las dos primeras: **len** y **size** que devuelven la longitud y el tamaño de las series respectivamente.

- **len**(s) # para obtener el tamaño de la serie s.
- -sa = s1.**copy**() # hace una copia de la serie.
- s.**describe**() # devuelve los valores estadísticos de la serie s.
- s.**value_counts**() # devuelve una serie con sus componentes no repetidas
- s.**index** # devuelve los índices de la serie s
- s.**values** # devuelve los valores de la serie s

Esta última función devuelve además el número de repeticiones de cada componente. Viene ordenada por este último número. Veamos un ejemplo:

```
import pandas as pd


s1 = pd.Series([1,5,2,7,3,2,5,8,1,4,7,2,5,3,6,1,2,7])

d1 = s1.describe()

print(d1)

print()

valCounts = s1.value_counts()

print(valCounts)

print()

ind1 = s1.index

print(ind1)

print()

val1 = s1.values

print(val1)

print()

print()
```

```
dic2 = {'1': [1], '2': [2, 3], '3': [3, 4, 5], '4': [4, 5, 6,
7]}

s2 = pd.Series(dic2)

d2 = s1.describe()

print(d2)

print()

valCounts = s2.value_counts()

print(valCounts)

print()

ind2 = s2.index

print(ind2)

print()

val2 = s2.values

print(val2)
```

```
count   18.000000
mean     3.944444
std      2.363254
min      1.000000
25%      2.000000
50%      3.500000
75%      5.750000
max      8.000000
dtype: float64

2  4
7  3
5  3
```

```
1    3
3    2
8    1
6    1
4    1
dtype: int64

RangeIndex(start=0, stop=18, step=1)

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

count    18.000000
mean      3.944444
std       2.363254
min       1.000000
25%       2.000000
50%       3.500000
75%       5.750000
max       8.000000
dtype: float64

[3, 4, 5]       1
[2, 3]          1
[1]             1
[4, 5, 6, 7]    1
dtype: int64

Index(['1', '2', '3', '4'], dtype='object')
```

[list([1]) list([2, 3]) list([3, 4, 5]) list([4, 5, 6, 7])]

6.4.1.5. Operaciones con las series

Se pueden realizar multitud de operaciones numéricas o booleanas con los datos de las series. Vamos a ver algunas de ellas.

Operadores numéricos

En cuanto a las operaciones de tipo numérico realizables con las series, se pueden utilizar los operadores aritméticos igual que se utilizan para los *array*s de *Numpy*. Por ejemplo:

- s + s # suma término a término.
- s * 2 # multiplica por 2 cada término.
- s + 1 # suma una unidad a cada término.

También se les pueden aplicar funciones numéricas de todo tipo. Para no ser exhaustivos, vamos a citar las funciones logarítmicas y las trigonométricas.

- np.log(s) # calcula el logaritmo de cada componente de la serie.
- np.cos(s) # calcula el coseno de cada componente de la serie

Operadores relacionales

Los datos de dos series se pueden comparar término a término utilizando los operadores relacionales. Por ejemplo:

```
s1 = pd.Series([1, 2, 3, 4])

s7 = pd.Series([2, 4, 6, 1])


iguales = s1 == s7

mayor = s1 > s7

menor = s1 < s7
```

Como ya hemos dicho, a comparación se realiza término a término. Por tanto, el resultado sería:

```
# operaciones con series

s7 = [2, 4, 6, 1]

s1 = [1, 2, 3, 4]

igual = s1 == s7
```

```
print(igual)

mayor = s1 > s7

print(mayor)

menor = s1 < s7

print(menor)
```

```
0    False
1    False
2    False
3    False
dtype: bool
0    False
1    False
2    False
3     True
dtype: bool
0     True
1     True
2     True
3    False
dtype: bool
```

6.4.1.6. Conversión entre series y listas o diccionarios

Las series se pueden convertir en listas o diccionarios y viceversa. Veamos algunos ejemplos trabajando con las series anteriores.

Conversión de series a listas

```
# conversión de series a listas

lst0 = s0.tolist()

print(lst0)


lst1 = s1.tolist()

print(lst1)
```

['a', 'b', 'c', 'd', 'e']

[1, 2, 3, 4]

Conversión de una serie en un diccionario

```
print()

print('Conversión de serie a diccionario')

d = s8.to_dict()

print(d)
```

Conversión de serie a diccionario

{'1': 10, '2': 20, '3': 30, '4': 40}

Conversión de listas que contienen listas a Series con sus componentes individuales

```
import pandas as pd


lst = [['uno', 'dos', 'tres'], ['cuatro', 'cinco'], ['seis'],
['siete']]
```

```
serie1 = pd.Series(lst)

print('Es una serie de listas')

print(serie1)


serie1=serie1.apply(pd.Series).stack().reset_index(drop=True)

print('Ahora se escribe una nueva serie con los elementos
originales de la serie anterior')

print(serie1)
```

Es una serie de listas

0 [uno, dos, tres]

1 [cuatro, cinco]

2 [seis]

3 [siete]

dtype: object

Ahora se escribe una nueva serie con los elementos originales de la serie anterior

0 uno

1 dos

2 tres

3 cuatro

4 cinco

5 seis

6 siete

dtype: object

Conversión de Series a diccionarios.

Se hará uso de la función to_dict. Por ejemplo:

```
# conversión de Series a Diccionarios

s = pd.Series([1, 6, 3, 1, 5, 3, 1, 7])
```

```
print('Imprimo la serie')
print(s)
print()

d = s.to_dict()
print('Convertida en diccionario', d)
print()

do = s.to_dict(OrderedDict)
print('Se crea una lista que contiene tuplas (clave, valor)')
print('Diccionario OrderedDict', do)
```

Imprimo la serie

0 1

1 6

2 3

3 1

4 5

5 3

6 1

7 7

dtype: int64

Convertida en diccionario {0: 1, 1: 6, 2: 3, 3: 1, 4: 5, 5: 3, 6: 1, 7: 7}

Se crea una lista que contiene tuplas (clave, valor)

Diccionario OrderedDict OrderedDict([(0, 1), (1, 6), (2, 3), (3, 1), (4, 5), (5, 3), (6, 1), (7, 7)])

6.4.1.7. Manipulación de las series

Modificación y filtrado de valores individuales

El filtrado es la extracción de aquella información contenida en una serie que cumple determinados criterios de búsqueda. Se puede aplicar tanto a series como a *DataFrame*s de Pandas.

Las series se pueden filtrar iterando a todos sus elementos y extrayendo aquellos que cumplan una cierta condición. Sin embargo, en Pandas se puede hacer de forma muy sencilla sin necesidad de programar ninguna iteración. Como siempre que se habla de condición, se deben usar predicados con los ya conocidos operadores relacionales.

Para realizar el filtrado de los valores individuales de una serie, modificando aquellos que cumplan una cierta condición, se utiliza la siguiente notación:

```
s[predicado] = dato
```

de forma que dada una serie s, se escribe entre corchetes un predicado, es decir, una condición lógica. Todos aquellos componentes de la serie cuyo valor cumpla esa condición se sustituyen por dato. Veamos algunos ejemplos.

Supongamos las series:

```
s7 = pd.Series([1, 2, 3, 4, 5 , 6, np.inf, np.nan])

      s7[s7 < 3] = 0    # pone a 0 los valores de s7 menores
que 3

      print(s7)
```

```
0       0.0
1       0.0
2       3.0
3       4.0
4       5.0
5       6.0
6       inf
7       NaN
dtype: float64
```

```
s7 = pd.Series([1, 2, 3, 4, 5 , 6, np.inf, np.nan])

      s7[s7 == np.inf] = 0   # pone a 100 los valores de s7 que
sean inf

      print(s7)
```

```
0       1.0
1       2.0
2       3.0
3       4.0
4       5.0
5       6.0
6       100
7       NaN
dtype: float64
```

```
s7 = pd.Series([1, 2, 3, 4, 5 , 6, np.inf, np.nan])

      s7[np.isnan(s7)] = 10   # pone a 10 los valores de s7 que
sean nan

      print(s7)
```

```
0       1.0
1       2.0
2       3.0
3       4.0
4       5.0
5       6.0
6       inf
7       10.0
dtype: float64
```

Sin embargo hay que aclarar que como inf (infinito) y nan (no es un número) no son números, no se pueden comparar consigo mismos. Por ejemplo:

```
numpy.nan == numpy.nan
```

es False.

Al igual que isnan se pueden utilizar isnull, notnan, notnull, min, max, mean, median, sum, mad, all, any, sort_values, idxmax. Por ejemplo:

```
(s1 == s2).all()       # mira si todos los valores son
iguales.

(s1 == s2).any()     # busca si hay alguna posición en la
que los valores son iguales.

s.sort_values().       # devuelve la serie ordenada por
valores.

s.idxmax()         # devuelve el índice correspondiente al
valor máximo
```

Formación de sub-series

Se pueden usar rangos para formar sub-series. Así:

- s7[2:4] devuelve una serie correspondiente a los valores de los índices del 2 al 4 EXCLUÍDO.
- s7[[0, 2, 3]] devuelve una serie con los valores de esos componentes.
- s7.iloc[2] devuelve una serie con el componente 2 (el primero es el 0)
- s7.iloc[2:4] lo mismo que la primera.

Algo que hay que tener el cuenta es que cuando se extrae una sub-serie, como por ejemplo, sb = s6a[s6a>3] los valores de los índices no cambian. s7[2] apunta siempre al mismo valor.

Operaciones sobre los índices.

Se pueden realizar operaciones con las series utilizando los índices. Por ejemplo:

```
si1 = pd.Series([1, 2, 3, 4], ['a', 'b', 'c', 'd'])

si2 = pd.Series([5, 6, 7, 8], ['a', 'd', 'e', 'f'])

si1 + si2
```

El resultado obtenido es:

a 6

b NaN

c NaN

d 10

e NaN

f NaN

ya que aquellos índices presentes solamente en una de las dos series dan como resultado NaN.

6.4.1.8. Funciones de filtrado

Existen muchas funciones que se pueden utilizar para el filtrado de series. Veamos algunas de ellas.

Función dropna

Devuelve una serie que contiene únicamente los datos que no son np.nan. Por ejemplo:

```
s1 = pd.Series([6, 1, 2, 7, 0, np.inf, 1, np.nan, np.nan])
print('Valores distintos de nan: ')
print(s1.dropna())
```

Valores distintos de nan:

0 6.0

1 1.0

2 2.0

3 7.0

4 0.0

5 inf

6 1.0

dtype: float64

Función fillna

También se pueden sustituir los np.nan por un número determinado usando la función **fillna**. Por ejemplo:

```
print('Vamos a sustituir los np.nan por unos: ')
print(s1.fillna(1))
```

Vamos a sustituir los np.nan por unos:

0 6.0

1 1.0

2 2.0

3 7.0

4 0.0

5 inf

6 1.0

7 1.0

8 1.0

dtype: float64

Otras formas de manipulación de datos son:

Función filter

La función Series.filter () de Pandas devuelve subconjuntos de datos del mismo tipo que los de entrada, de acuerdo con los criterios especificados. El filtro se aplica a las etiquetas del índice de la serie. Veamos con algún ejemplo su uso:

```
# importamos pandas como pd

import pandas as pd

import Numpy as np


# Creamos la Serie

sr = pd.Series([1, 2, 3, 5, 7, 9, np.nan])


# Creamos el índice

ind = ['impar1', 'uff, par', 'impar2', 'impar3, 'impar4',
'impar5', 'No_número']


# asociamos el índice a la serie

sr.index = ind
```

```
# escribimos la serie
print(sr)
```

```
impar1          1.0
Uff, par 2.0
impar2          3.0
impar3          5.0
impar4          7.0
impar5          9.0
No_número       NaN
dtype: float64
```

Ahora vamos a usar la función Series.filter () para filtrar los valores de la serie sr cuya etiqueta contiene un espacio en su nombre.

```
# filtramos valores
sal1 = sr.filter(regex = ' ') # buscamos en la columna datos
separados por un espacio

# escribimos el resultado
print(sal1)
```

```
Uff, par     2.0
dtype: float64
```

```
sal2 = sr.filter(ítems = ['impar2', 'imar4'])
print(sal2)
```

```
impar2            3.0
impar4            7.0
dtype: int64
```

```
sal3 = sr.filter(regex = '2')
print(sal3)
print()
```

```
impar2            3
dtype: float64
```

Aplicación de una función a una serie.

Aplicando la función **apply** se pueden modificar los valores de una serie. Basta con aplicar la función cuadrado a cada elemento para obtener una nueva serie.

```
def cuadrado(x):
    return x**2

sr1 = sr.apply(cuadrado)
print(sr1)
```

```
impar1            1.0
Uff, par  4.0
impar2            9.0
impar3           25.0
impar4           49.0
impar5           81.0
No_número        NaN
dtype: float64
```

Si se usa una expresión lambda:

```
sr2 = sr.apply(lambda x: x**3)
```

```
impar1          1.0
Uff, par    8.0
impar2         27.0
impar3        125.0
impar4        343.0
impar5        729.0
No_número     NaN
dtype: float64
```

6.4.2. *Dataframe*

Es una tabla bidimensional con índices de líneas y de columnas, pero también se puede ver como una lista de "Series" que comparten el mismo índice. Un *DataFrame* se comporta como un diccionario cuyas claves son los nombres de las columnas y los valores son series. Un *DataFrame* se puede crear a partir de un *array Numpy*, aunque no es muy práctico y el tipo de datos es el mismo para todas las columnas.

- Creación de un *DataFrame*.

```
ar = numpy.array([[1.5, 3, 3.4, 9], [3.4, 5, 3.6, 1.2],
[5.2, 1, 2.8, 3.9]])

df = pandas.DataFrame(ar, index = ['a1', 'a2', 'a3'],
colums = ['A', 'B', 'C', 'D'])
```

	A	B	C	D
a1	1.5	3	3.4	9
a2	3.4	5	3.6	1.2
a3	5.2	1	2.8	3.9

También se puede crear un *DataFrame* con un diccionario:

```
df = pandas.DataFrame({'A': [1.5, 3.4, 5.2], 'B': [3, 5,
1], 'C': [3.4, 3.6, 2.8], 'D': [9, 1.2, 3.9]}, index =
['a1', 'a2', 'a3'])
```

Se puede definir un DataFrame con las columnas en el orden que se quiera:

```
df = pandas.DataFrame({'A': [1.5, 3.4, 5.2], 'B': [3, 5, 1], 'C': [3.4, 3.6, 2.8], 'D': [9, 1.2, 3.9]}, columns = ['A', 'B', 'C', 'D'])
```

Mediante una lista de diccionarios:

```
df = pandas.DataFrame([{'A': 1.5, 'B': 3, 'C': 3.4, 'D': 9}, {'A': 3.4, 'B': 5, 'C': 3.6, 'D': 1.2}, {'A': 5.2, 'B': 1, 'C': 2.8, 'D': 3.9}])
```

Se pueden modificar los índices de un *DataFrame* cambiando el orden de las filas y/o de las columnas:

```
df.reindex(columns = ['C', 'A', 'B'], index = ['a2', 'a3'])
```

Los tipos de las diferentes columnas se obtienen: df.dtypes.

- Funciones que operan con los *DataFrames*:

```
df.info()
df.head(2)          # devuelve un dataframe con las dos primeras líneas
df.head(), df.tail()
df.columns          # los nombres de las columnas
df.columns.values
df.index            # los nombres de las filas
df.index.values
df.describe()               # devuelve valores estadísticos de la tabla
```

La dimensión de un dataframe se obtiene:

```
df.shape
len(df)
len(df.columns)
```

EJEMPLO

En la tabla que sigue se dan los valores de las acciones de cuatro empresas a 31 de diciembre de los años 2016 a 2018.

	2016	2017	2018
Santander	4,322	5,07	3,857
BBVA	5,717	6,618	4,5
Telefónica	7,796	7,502	7,14
Inditex	30,339	27,732	21,985

¿Cómo se implementaría esta tabla usando *DataFrame*?. Aplicar las funciones anteriormente descritas.

```
import pandas as pd
df1 = pd.DataFrame({'2016': [4.322, 5.717, 7.796, 30.339],
'2017': [5.07, 6.618, 7.502, 27.732], '2018': [3.857,
4.5, 7.14, 21.985]}, columns = ['2016', '2017', '2018'])
print(df1.info())
print()
print(df1.head(1))
print()
print(df1.columns)
print()
print(df1.columns.values)
print()
print(df1.first_valid_index)
print()
print(df1.index.values)
print()
print(df1.describe())
print()
print(df1.shape)
print()
print(len(df1))
print()
print(len(df1.columns))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
2016    4 non-null float64
2017    4 non-null float64
2018    4 non-null float64
dtypes: float64(3)
memory usage: 176.0 bytes
None
    2016  2017   2018
0  4.322  5.07  3.857
Index(['2016', '2017', '2018'], dtype='object')
['2016' '2017' '2018']
<bound method NDFrame.first_valid_index of       2016     2017
2018
0   4.322   5.070   3.857
1   5.717   6.618   4.500
2   7.796   7.502   7.140
3  30.339  27.732  21.985>
[0 1 2 3]
            2016       2017       2018
count   4.000000   4.000000   4.000000
mean   12.043500  11.730500   9.370500
std    12.280238  10.714914   8.528798
min     4.322000   5.070000   3.857000
25%     5.368250   6.231000   4.339250
50%     6.756500   7.060000   5.820000
75%    13.431750  12.559500  10.851250
```

```
max     30.339000  27.732000  21.985000

(4, 3)

4

3
```

Vemos que tal como han salido, los valores estadísticos no nos aportan demasiado, ya que nos interesaría calcular dichos valores sobre la evolución de cada acción sin mezclarlas entre si. Para ello, habría que cambiar filas y columnas. Como ejercicio, repetir todo efectuando ese cambio.

RESUMEN

- En esta unidad se estudian tres librerías de gran utilidad en numerosas aplicaciones: NumPy, Matplotlib y Pandas.
- NumPy es un módulo básico para el cálculo científico en Python.
- Matplotlib es una librería de trazado gráfico y visualización que es capaz de producir gráficos en 2D y 3D de alta calidad en diferentes formatos de salida.
- Pandas es un paquete de Python que proporciona estructuras de datos rápidas y flexibles.
- NumPy es la librería usada para trabajar con vectores.
- Trabaja con vectores multidimensionales heterogéneos ya que sus componentes pueden ser de distintos tipos.
- Para crear vectores (en adelante les llamaremos arrays) se puede usar la función *array* a la que se pasa una lista como argumento.
- Existen funciones para crear determinados arrays como: zeros (crea un *array* con ceros), ones (con unos), etc.
- Los operadores +, -, * y / operan elemento a elemento (no sobre matrices).
- Se pueden crear secuencias numéricas (rangos) con la función arange. Esta función devuelve un array, no una lista. Por ejemplo: arange(5, 9, 2) devuelve un array cuya primera componente es 5 y le sigue 7. Se toman de 2 en 2 pero se excluye el 9.
- A partir de un rango se puede crear una matriz usando la función reshape. Por ejemplo: arange(15).reshape(3,5) genera un rango de 0 a 14 y lo distribuye en una matriz de 3 filas por 5 columnas.
- linspace(inicial, final, numPuntos) genera un vector de puntos en el intervalo cerrado [inicial, final] con un total de numPuntos.
- numpy dispone de funciones para trabajar con números pseudo-aleatorios, tales como:
  - random.seed(num) para inicializar el generador.
  - random.rand(n) para generar n números aleatorios con distribución uniforme.
  - random.randn(n) para generarlos pero con distribución normal.
  - randint(inf, sup, n) para generar n enteros en el intervalo [inf, sup) sin llegar a tomar el valor sup.
- La función len(m) permite saber la dimensión de una matriz m.

- La función **shape** devuelve el tamaño de una matriz.
- Se pueden extraer sub-vectores utilizando el operador ":".
- **Matplotlib** es un paquete para representaciones gráficas.
- **matplotlib.pyplot** es una colección de comandos usados en las representaciones gráficas.
- La función **plot** de matplotlib se utiliza para crear gráficas.
- Las gráficas creadas se representan en pantalla usando la función **show**.
- A la función plot se le puede pasar una lista para que la represente. Si se le pasan dos listas separadas por coma, una corresponde a las abcisas y la otra a las ordenadas.
- La representación de la función plot se hace uniendo cada punto con el siguiente.
- La función **title** permite dar un título a la figura.
- **xlabel** e **ylabel** se usan para poner leyendas a los ejes coordenados.
- El comando **axis** toma como argumento una lista y se usa para introducir los límites correspondientes a los ejes de abcisas y ordenadas. Se introducen en la forma [xmin, xmax, ymin, ymax].
- Un tercer parámetro de la función **plot** puede ser una cadena de formato que indique la línea y su color. Se introducen entre comillas simples.
- La función **figure** crea una nueva figura. Las figuras se cierran con la función close.
- La función **subplot** permite definir e identificar la ventana en la que se va a representar cada figura.
- **text** permite añadir texto en una posición arbitraria.
- Se pueden definir escalas no lineales con el comando **xscale** pasando la escala como parámetro.
- **Pandas** permite manipular datos de forma sencilla.
- Puede trabajar con series (vectores indexados) y *DataFrames* (tablas de datos).
- Se pueden crear las series con el comando Series al que se le pasa una lista como parámetro.

- Sobre las series operan funciones: len(s), s.size, copy, value_counts, etc.
- En el caso de los vectores de Numpy el índice son números enteros que comienzan por el 0 para la primera componente. Sin embargo, las series de Pandas son mucho más flexibles para permitir trabajar con series más complejas.
  - Se puede utilizar un diccionario. sd = pandas.Series({'a':1, 'b':2, 'c':3, 'd':4}).
  - Se puede dar nombres a los componentes de una serie, especificando los nombres en una lista asignada a un parámetro index. Por ejemplo: sa = pandas.Series([1, 2, 3, 4], index = ['a', 'b', 'c', 'd']).
- Se pueden formar sub-series utilizando rangos. Por ejemplo: s[2:6].
- Se pueden realizar operaciones con las series utilizando los índices.
- Un *DataFrames* es una tabla bidimensional con índices de líneas y de columnas.
- Un *DataFrame* se comporta como un diccionario cuyas claves son los nombres de las columnas y los valores son series.
- Se puede crear a partir de un array numpy. También a partir de un diccionario o de una lista de diccionarios.
- Existen numerosas funciones que operan con *DataFrames* tales como: info, head, columns, index, etc.
- La dimensión de un *DataFrames* se btiene: df.shape, len(df) y len(df.columns).