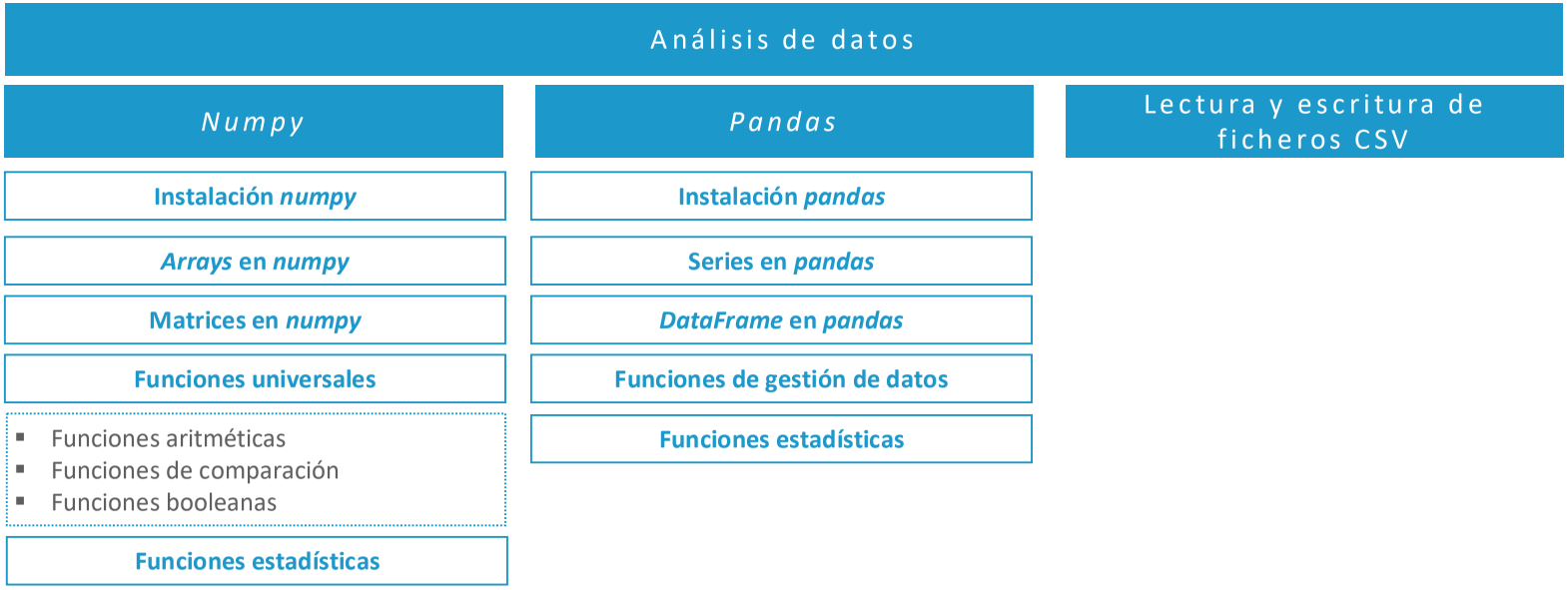
Introducción a la programación y al análisis de datos con Python

Análisis de datos

Índice

Esquema



Ideas clave

7.1. Introducción y objetivos

Al comienzo del curso enumeramos muchas de las áreas donde Python se ha convertido en uno de los lenguajes de programación más importantes. Una de esas áreas es el análisis de datos. En este tema, describiremos las principales funcionalidades de los dos módulos más importantes para manejar datos que tenemos en Python: `numpy` y `pandas`.

Al final de este tema, habrás logrado alcanzar los siguientes objetivos:

- Comprender las estructuras de datos básicas de `numpy`: *arrays* y matrices.
- Aprender a aplicar las funciones universales en `numpy`.
- Saber cómo aplicar funciones estadísticas en las estructuras de `numpy`.
- Conocer las estructuras de datos básicas de `pandas`: series y *dataframes*.
- Comprender las funciones de filtrado en las estructuras de `pandas`.
- Saber aplicar funciones estadísticas en `pandas`.
- Aprender a modificar los valores almacenados en las estructuras de datos de `pandas`.

7.2. Numpy

`numpy` es un módulo que podemos instalar en Python y está orientado a librerías científicas. Este módulo proporciona nuevas estructuras de datos, como son las matrices y las matrices multidimensionales, e incluye métodos muy potentes para trabajar con ellos.

Esta librería es la base de muchas de las librerías de análisis de datos y científicas que existen en Python. En este apartado veremos cómo se instala, las estructuras de *arrays* y matrices, y algunas de las funciones más interesantes.

Instalación de `numpy`

`numpy` no está incluido entre los módulos de la distribución básica de Python. En estos casos debemos instalar este módulo usando el gestor de módulo `pip`. La instrucción que debemos ejecutar en nuestra consola para instalar `numpy` es la siguiente:

```
pip install numpy
```

En el caso de que hayamos instalado Anaconda, la distribución recomendada para este curso, `numpy` vendrá instalado por defecto y no tendremos que hacer nada. Para utilizar `numpy` en nuestros proyectos es necesario importarlo previamente. Como norma general, cuando importamos `numpy` le asignamos un alias corto para que sea facil utilizarlo en nuestro código, este alias suele ser `np`. Para importar `numpy` en nuestro código, la primera instrucción debe ser la siguiente:

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

Una vez hecho esto, cada vez que hagamos referencia a elementos del módulo `numpy`, usaremos su alias `np`.

Arrays en `numpy`

Comenzaremos viendo la estructura más básica que existe dentro de `numpy`, los *arrays* de `numpy`. Esta estructura de datos es una secuencia de valores, los cuales tienen asignada una posición. Son muy parecidos a las listas de Python, pero los *arrays* son más rápidos y existen muchos cálculos que podemos hacer sobre todos los valores de un *array* de forma más rápida que con las listas en Python.

Para crear un *array* en `numpy`, usaremos la función `array()` pasando por parámetro la lista de valores que queremos incluir en el *array*. Por ejemplo, en el siguiente ejemplo crearemos un *array* con los valores de 1 a 5:

```
In [2]: array = np.array([1, 2, 3, 4, 5])
        array
Out[2]: array([1, 2, 3, 4, 5])
```

En este caso, hemos creado un *array* de una dimensión (`1-D Array`). Este tipo de *array* es una lista de valores y, si accedemos a una posición cualquiera del *array,* obtendríamos un único valor. Por ejemplo, si accedemos a la tercera posición obtendríamos el valor 3:

```
In [3]: array[2] # Recordar que las posiciones empiezan con 0
Out[3]: 3
```

Al igual que las listas, se pueden introducir elementos de diferente tipo en cada posición del *array.* Pero debemos tener en cuenta que las futuras operaciones nos pueden dar errores o resultados inesperados porque estamos operando con diferentes tipos de datos.

```
In [4]: array = np.array([1.0, 'Hola', True])
        array
Out[4]: array(['1.0', 'Hola', 'True'], dtype='<U32')
```

Matrices en `numpy`

Sin embargo, podemos crear *arrays* con más dimensiones. Cuando existe más de una dimensión en un *array,* estas estructuras tienen el nombre de matrices. La matriz más común es la que tiene 2 dimensiones. Para crear una matriz de dos dimensiones en `numpy`, usaremos una lista de listas. Por ejemplo, para crear una matriz donde la

primera fila tenga los valores `[1, 2, 3]` y la segunda fila tenga los valores `[4, 5, 6]`, lo haríamos de la siguiente manera:

```
In [5]: matriz = np.array([[1,2,3],[4,5,6]])
        matriz

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

Es importante que la longitud de las filas sean las mismas, ya que de lo contrario se creará un *array* de una dimensión donde cada elemento será una lista de Python.

```
In [6]: matriz = np.array([[1,2,3],[4,5]])
        matriz

Out[6]: array([list([1, 2, 3]), list([4, 5])], dtype=object)
```

Podemos crear matrices con más dimensiones, incluyendo más listas en las posiciones correspondientes. Esto hace que `numpy` sea una librería muy potente para crear matrices con `n` dimensiones.

Para entender por qué es importante utilizar `numpy`, vamos a hacer algunas comparaciones de rendimiento entre listas y *arrays*/matrices con `numpy`. En primer lugar, vamos a comparar la memoria que ocupa una lista de Python con respecto a la memoria que ocupa un *array* de `numpy`. Ambas estructuras tendrán 1000 valores:

```
In [7]: import sys # Nos permitirá preguntar por la memoria

        lista = range(1000)
        array = np.array(range(1000))

        '''
        La memoria de la lista la calculamos como lo que ocupa un número entero
        multiplicado por la longitud de la lista. El valor viene devuelto en bytes.
        '''
        print(sys.getsizeof(1) * len(lista))

        '''
        La memoría del array la calculamos con la multiplicación del tamaño del
        array (en número de elementos), por el tamaño de cada elemento en memoria
        (en bytes)
        '''
        print(array.size * array.itemsize)

        28000
        8000
```

Como se puede observar, los *arrays* implementados en numpy ocupan menos espacio en la memoria. En el caso del ejemplo, esta mejora es del 71 % aproximadamente.

También los *arrays* de numpy hacen que las operaciones sean mucho más eficientes. Vamos a comparar esto aplicando una operación entre dos listas y, a continuación, la misma operación usando *arrays.* En ambos casos el tamaño será el mismo (1 000 000 de elementos) y la operación será la resta. Veamos la comparación de tiempos:

```
In [9]: import time

        lista1 = range(1000000)
        lista2 = range(1000000)
        array1 = np.array(range(1000000))
        array2 = np.array(range(1000000))

        # Tiempo operación resta en listas
        comienzo = time.time()
        resultado = [x - y for x, y in zip(lista1, lista2)]
        final = time.time()

        print('Tiempo: ', final - comienzo)

        # Tiempo operación resta en arrays
        comienzo2 = time.time()
        resultado = array1 - array2
        final2 = time.time()

        print('Tiempo: ', final2 - comienzo2)
```

```
Tiempo:  0.08084297180175781
Tiempo:  0.005764007568359375
```

Como se puede observar, la misma operación es mucho más rápida usando los *arrays* de numpy. En el caso del ejemplo, tenemos una mejora de aproximadamente el 90 %. Estos factores hacen que los *arrays* de numpy sean más populares que las listas en Python, sobre todo cuando vamos a trabajar con grandes cantidades de datos. A continuación, veremos algunas de las funciones más útiles en numpy.

Funciones universales

Las funciones universales son aquellas que se aplican a cada uno de los elementos de un *array.* Es decir, que la función recorrerá cada uno de los elementos de un *array* en numpy y le aplicará la operación correspondiente. En caso de que esa operación se haga con dos *arrays,* por ejemplo, la resta de dos *arrays* elemento por elemento, es

necesario que ambos *arrays* tengan la misma longitud. Vamos a ver algunas de las funciones universales más importantes que están disponibles en el módulo `numpy`.

Funciones aritméticas

Veamos las operaciones aritméticas en las funciones universales de `numpy`:

- **`substract()`:** resta los elementos de dos *arrays* elemento por elemento.

```
In [10]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 10])
         np.subtract(array1, array2)
Out[10]: array([-2, 50,  5])
```

- **`add()`:** suma los valores de dos *arrays* elemento por elemento.

```
In [11]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 10])
         np.add(array1, array2)
Out[11]: array([ 10, 128,  25])
```

- **`multiply()`:** multiplica los valores de dos *arrays* elemento por elemento.

```
In [12]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 10])
         np.multiply(array1, array2)
Out[12]: array([  24, 3471,  150])
```

- **`divide()`:** divide los valores de dos *arrays* elemento por elemento.

```
In [13]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 10])
         np.divide(array1, array2)
Out[13]: array([0.66666667, 2.28205128, 1.5       ])
```

- **`power()`:** devuelve como resultado la potencia del elemento del primer *array* elevado al elemento del segundo *array*.

```
In [14]: array1 = np.array([2, 3, 5])
         array2 = np.array([2, 4, 10])
         np.power(array1, array2)
Out[14]: array([      4,      81, 9765625])
```

- **`sqrt()`:** devuelve la raíz cuadrada de cada uno de los elementos de un *array*.

```
In [15]: array1 = np.array([4, 89, 15])
         np.sqrt(array1)
Out[15]: array([2.        , 9.43398113, 3.87298335])
```

- **`square()`:** devuelve el cuadrado de cada uno de los elementos de un *array*.

```
In [16]: array1 = np.array([4, 89, 15])
         np.square(array1)
Out[16]: array([  16, 7921,  225])
```

- **`gcd()`:** devuelve el máximo común divisor de los elementos de dos *arrays*.

```
In [17]: array1 = np.array([2, 3, 5])
         array2 = np.array([2, 4, 10])
         np.gcd(array1, array2)
Out[17]: array([2, 1, 5])
```

- **`lcm()`:** devuelve el mínimo común múltiplo de los elementos de dos *arrays*.

```
In [18]: array1 = np.array([2, 3, 5])
         array2 = np.array([2, 4, 10])
         np.lcm(array1, array2)
Out[18]: array([ 2, 12, 10])
```

Funciones de comparación

Este conjunto de funciones permite comparar los valores de dos *arrays* elemento por elemento.

- **`greater()`:** realiza la operación `elem1>elem2` por cada uno de los valores de ambos *arrays.*

```
In [19]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 10])
         np.greater(array1, array2)
Out[19]: array([False,  True,  True])
```

- **`greater_equal()`:** realiza la operación `elem1>=elem2` por cada uno de los valores de ambos *arrays.*

```
In [20]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 15])
         np.greater_equal(array1, array2)
Out[20]: array([False,  True,  True])
```

- **less():** realiza la operación `elem1<elem2` por cada uno de los valores de ambos *arrays.*

```
In [21]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 15])
         np.less(array1, array2)
Out[21]: array([ True, False, False])
```

- **`less_equal()`:** realiza la operación `elem1<=elem2` por cada uno de los valores de ambos *arrays.*

```
In [22]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 15])
         np.less_equal(array1, array2)
Out[22]: array([ True, False,  True])
```

- **`equal()`:** realiza la operación `elem1==elem2` por cada uno de los valores de ambos *arrays.*

```
In [23]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 15])
         np.equal(array1, array2)
Out[23]: array([False, False,  True])
```

- **`not_equal()`:** realiza la operación `elem1!=elem2` por cada uno de los valores de ambos *arrays.*

```
In [24]: array1 = np.array([4, 89, 15])
         array2 = np.array([6, 39, 15])
         np.not_equal(array1, array2)
Out[24]: array([ True,  True, False])
```

Funciones booleanas

Funciones que aplican operaciones booleanas elemento por elemento. Los *arrays* sobre los que se realizan las operaciones deben contener valores booleanos.

- **`logical_and()`**: realiza la operación `and` sobre cada uno de los elementos de dos *arrays.*

```
In [25]: array1 = np.array([True, False, True])
         array2 = np.array([False, False, True])
         np.logical_and(array1, array2)
Out[25]: array([False, False,  True])
```

- **`logical_or`**(): realiza la operación `or` sobre cada uno de los elementos de dos *arrays.*

```
In [26]: array1 = np.array([True, False, True])
         array2 = np.array([False, False, True])
         np.logical_or(array1, array2)
Out[26]: array([ True, False,  True])
```

- **`logical_xor()`**: realiza la operación `xor` sobre cada uno de los elementos de dos *arrays.*

```
In [27]: array1 = np.array([True, False, True])
         array2 = np.array([False, False, True])
         np.logical_xor(array1, array2)

Out[27]: array([ True, False, False])
```

- **`logical_not()`**: realiza la operación `not` sobre cada uno de los elementos de un *array.*

```
In [28]: array1 = np.array([True, False, True])
         np.logical_not(array1)

Out[28]: array([False,  True, False])
```

Existen muchas otras funciones universales que se pueden consultar en los enlaces que hemos incluido en la sección A fondo.

Funciones estadísticas

Los *arrays* que hemos visto anteriormente son una forma muy buena de almacenar conjuntos de datos para, más adelante, estudiar algunas propiedades estadísticas como la distribución de los datos, la media, etc. En este epígrafe, vamos a enumerar las principales funciones estadísticas que podemos usar con los *arrays* de `numpy`.

- **`amin()`:** devuelve el valor mínimo de todos los elementos que existen en un *array.*

  También se puede aplicar sobre matrices.

```
In [29]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Mínimo: ", np.amin(array))

         matriz = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
         print("Mínimo: ", np.amin(matriz))

         Mínimo:  1
         Mínimo:  1
```

- **`amax()`:** devuelve el valor máximo de todos los elementos que existen en un *array*.

```
In [30]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Máximo: ", np.amax(array))

         matriz = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
         print("Máximo: ", np.amax(matriz))
```

```
Máximo:  10
Máximo:  8
```

- **`percentile()`:** devuelve el valor sobre el que se encuentra un porcentaje de un conjunto de observaciones ordenadas de mayor a menor. Para esta función debemos insertar el *array* o matriz de valores, el percentil que queremos observar (1-100) y, en el caso de matrices, el eje sobre el que queremos hacer la observación (0 sobre las filas, 1 sobre las columnas).

```
In [31]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Percentil 25: ", np.percentile(array, 25))
```

```
Percentil 25:  3.25
```

- **`median()`:** devuelve la mediana, es decir, el valor que separa el conjunto de observaciones ordenadas de mayor a menor en 2 mitades.

```
In [32]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Mediana: ", np.median(array))
```

```
Mediana:  5.5
```

- **`mean()`:** devuelve la media, es decir, el valor de tendencia central de un conjunto de observaciones. Se obtiene de la suma de todos los valores dividido por el número de observaciones.

```
In [33]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Media: ", np.mean(array))
```

```
Media:  5.5
```

- **`average()`:** devuelve la media ponderada de un conjunto de observaciones. En este caso, cada observación tiene un peso en la media (dado por otro *array*), el resultado será la suma de los valores de las observaciones multiplicado respectivamente por su peso y dividido por la suma de todos los pesos.

```
In [34]: pesos = np.array([0.2, 0.05, 0.05, 0.3, 0.0, 0.1, 0.05, 0.05, 0.1, 0.1])
         array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Media ponderada: ", np.average(array, weights=pesos))

         Media ponderada:  4.9
```

- **`std()`:** devuelve la desviación estándar de un conjunto de observaciones.

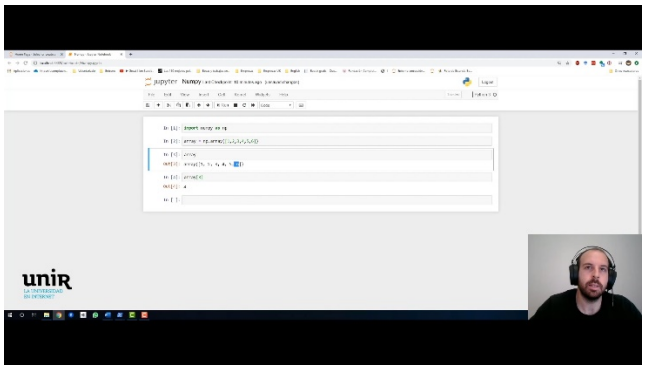
```
In [35]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Desviación estandar: ", np.std(array))

         Desviación estandar:  2.8722813232690143
```

- **`var()`:** devuelve la varianza de un conjunto de observaciones.

```
In [36]: array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         print("Varianza: ", np.var(array))

         Varianza:  8.25
```



Vídeo 1. Numpy.

Accede al vídeo a través del aula virtual

7.3. Pandas

pandas es otro módulo cuyo objetivo principal es la manipulación y el análisis de datos en Python. Este módulo es una extensión del módulo que hemos visto anteriormente, numpy. Este nuevo módulo incluye nuevas estructuras de datos y operaciones para manipular tablas de datos, como veremos más adelante.

En este apartado explicaremos cómo instalar pandas en nuestra distribución de Python, las estructuras más utilizadas (series y *dataframes*) y algunas de las funciones más utilizadas.

Instalación de pandas

Al igual que nos ocurría con numpy, pandas no está instalado en la distribución básica de Python, aunque si en la distribución de Anaconda. Para instalar pandas en nuestra distribución debemos ejecutar la siguiente instrucción en la consola:

```
pip install pandas
```

También, al igual que nos pasaba con numpy, si queremos utilizar pandas en nuestro proyecto es necesario que lo importemos al comienzo. En pandas también usaremos un alias que nos permita llamar a este módulo de forma rápida y sencilla. Normalmente, importamos pandas con el alias pd:

```
In [2]: import pandas as pd
```

Una vez hecho esto, usaremos el alias pd para llamar a los métodos y estructuras de datos de pandas.

Series en pandas

Las series son una estructura de datos de una dimensión, como las listas o los *arrays* en `numpy`. Lo que diferencia a las series es que el índice de cada elemento puede ser una etiqueta que asignemos nosotros, parecido a lo que se hace en los diccionarios. Para crear una serie utilizamos el método `Series()` donde añadiremos los valores y, si queremos, los índices personalizados:

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

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

Vemos como cada uno de los valores, en este caso valores enteros, tiene asignado una etiqueta como índice, en este caso, letras. Existen otras formas de crear series. Por ejemplo, podemos usar un *array* de `numpy` o un diccionario:

```
In [4]: serie1 = pd.Series(np.array([1, 2, 3, 4, 5]), index=['a', 'b', 'c', 'd', 'e'])
        print("Serie1:")
        print(serie1)

        diccionario = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
        serie2 = pd.Series(diccionario)
        print("Serie2:")
        print(serie2)

        Serie1:
        a    1
        b    2
        c    3
        d    4
        e    5
        dtype: int64
        Serie2:
        a    1
        b    2
        c    3
        d    4
        e    5
        dtype: int64
```

Las series actúan de forma similar a los *arrays* en `numpy`. Podemos acceder a una posición concreta usando la posición que ocupa un elemento o el índice asignado. Incluso podemos usar los rangos que hemos visto en las listas de Python.

```
In [5]: # Acceso por indice
        print(serie1['a'])

        # Acceso por posición
        print(serie1[3])

        # Acceso por rango
        print(serie1[2:])
```

```
1
4
c    3
d    4
e    5
dtype: int64
```

En las series existen 2 atributos que podemos necesitar con mucha frecuencia. El primero de ellos es el atributo `index` y nos devuelve los índices que hemos asignado a cada uno de los elementos de las series. Por otro lado, tenemos el atributo `values`, que devuelve un *array* con los valores de todos los elementos:

```
In [6]: # Obtener índices de una serie
        print(serie1.index)

        # Obtener valores de una serie
        print(serie1.values)
```

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
[1 2 3 4 5]
```

Las series están basadas en los *array* de `numpy`. Por este motivo, podemos aplicar operaciones vectorizadas de manera sencilla, al igual que hacíamos con los *arrays* en `numpy`. Sin embargo, es importante que ambas series tengan los mismos índices, ya que de lo contrario pueden devolver resultados no esperados.

```
In [7]: # Sumar los valores de 2 series (tienen que tener los mismos índices)
resultado = serie1 + serie2
print(resultado, "\n")

# Multiplicar todos los valores por 10
resultado = serie1 * 10
print(resultado, "\n")

# Obtener la raiz cuadrada de los valores de una serie
resultado = np.sqrt(serie1)
print(resultado, "\n")
```

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

a    10
b    20
c    30
d    40
e    50
dtype: int64

a    1.000000
b    1.414214
c    1.732051
d    2.000000
e    2.236068
dtype: float64
```

Dataframe en pandas

Los *dataframes* son la estructura más utilizada en pandas. Estos *dataframes* son una estructura de dos dimensiones de datos etiquetados, es decir, representan una tabla donde cada posición de dicha tabla tiene una etiqueta en la fila y otra etiqueta en la columna. Existen muchas formas de construir *dataframes.* Todas ellas usan la función `DataFrame()` de pandas. Por ejemplo, podemos construir un *dataframe* a partir de un diccionario que almacena dos series:

```
In [8]: # Construcción de un DataFrame a partir de un diccionario
diccionario = {'columna1': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
               'columna2': pd.Series([5, 6, 7, 8], index=['a', 'b', 'd', 'e'])}

dataframe = pd.DataFrame(diccionario)
dataframe
```

Out[8]:

	columna1	columna2
a	1.0	5.0
b	2.0	6.0
c	3.0	NaN
d	4.0	7.0
e	NaN	8.0

Como se puede observar, cada una de las series será una columna del *dataframe.* Los índices de las series serán las etiquetas de las columnas y las claves del diccionario, las etiquetas de las columnas. Además, podemos ver qué ocurre cuando unimos series con diferentes índices. En este caso, el *dataframe* usa todos los índices, pero a aquellas series que no tengan un valor en ese índice, le asignará NaN, que significa que no tiene valor. Otra posible forma de crear un *dataframe* es mediante una lista de diccionarios:

```
In [9]: lista = [{'a': 1, 'b': 2, 'c': 3}, {'a': 5, 'b': 8, 'd': 1}]

        dataframe = pd.DataFrame(lista)
        dataframe
```

Out[9]:

	a	b	c	d
0	1	2	3.0	NaN
1	5	8	NaN	1.0

En este caso, cada uno de los elementos de la lista ocupa una fila y las claves usadas en los diccionarios se usan para etiquetar las columnas. Es importante saber cómo se va a construir un *dataframe,* ya que dependiendo de qué estructura usemos, obtendremos *dataframes* diferentes. Por este motivo, es muy recomendable consultar la documentación de pandas.

Los *dataframes* nos permiten hacer diferentes operaciones con las columnas. La primera de ellas es poder consultar una columna. Para ello, usamos los corchetes ([]) y la etiqueta de la columna que contiene los valores que queremos consultar:

```
In [10]: # Creemos el dataframe
         diccionario = {'columna1': pd.Series([10, 2, 9, 8], index=['a', 'b', 'c', 'd']),
                        'columna2': pd.Series([5, 6, 7, 8], index=['a', 'b', 'd', 'e'])}

         dataframe = pd.DataFrame(diccionario)

         # Consulta columna 1
         dataframe['columna1']
```

```
Out[10]: a    10.0
         b     2.0
         c     9.0
         d     8.0
         e     NaN
         Name: columna1, dtype: float64
```

La forma en la que se nos devuelven los valores de una columna es a través de una serie. Otra operación que podemos hacer es agregar una nueva columna en el *dataframe.* Existen muchas maneras, pero la más común es agregar una nueva serie:

```
In [11]: serie = pd.Series([5, 19, 76, 22, 9], index=['a', 'b', 'c', 'd', 'e'])
         dataframe['columna3'] = serie
         dataframe
```

Out[11]:

	columna1	columna2	columna3
a	10.0	5.0	5
b	2.0	6.0	19
c	9.0	NaN	76
d	8.0	7.0	22
e	NaN	8.0	9

Por último, podemos eliminar una columna del *dataframe* usando la etiqueta de la columna y la sentencia `del`:

```
In [12]: del dataframe['columna3']
         dataframe
```

Out[12]:

	columna1	columna2
a	10.0	5.0
b	2.0	6.0
c	9.0	NaN
d	8.0	7.0
e	NaN	8.0

Un apartado importante dentro de los *dataframes* responde a cómo se puede acceder a los diferentes valores. Ya hemos visto cómo acceder a los valores de una columna. Pero también podemos usar las siguientes formas.

- **Seleccionar fila por etiqueta:** usando la propiedad `loc` del *dataframe* podemos acceder a los valores de una fila a través de su etiqueta:

```
In [13]: dataframe.loc['a']
Out[13]: columna1    10.0
         columna2     5.0
         Name: a, dtype: float64
```

- **Seleccionar fila por posición:** la propiedad `iloc` nos permite acceder a las filas por la posición que ocupan.

```
In [14]: dataframe.iloc[2:4]
```

Out[14]:

	columna1	columna2
c	9.0	NaN
d	8.0	7.0

- **Seleccionar filas por *array* booleano:** este método nos mostrará aquellas filas cuya posición tenga el valor `True` en un vector booleano. Este método es muy útil para filtrar elementos.

```
In [15]: # Creemos el dataframe
diccionario = {'columna1': pd.Series([10, 2, 9, 8], index=['a', 'b', 'c', 'd']),
               'columna2': pd.Series([5, 6, 7, 8], index=['a', 'b', 'd', 'e'])}

dataframe = pd.DataFrame(diccionario)

# Buscamos que posiciones de la columna1 tienen un valor mayor o igual al mismo elemento en la columna 2
filtro = np.greater_equal(dataframe['columna1'], dataframe['columna2'])

# Mostramos estos valores
dataframe[filtro]
```

Out[15]:

	columna1	columna2
a	10.0	5.0
d	8.0	7.0

Al igual que las series, podemos aplicar diferentes operadores de la misma manera que hacíamos en `numpy`. Sin embargo, en los *dataframes* hay que tener especial cuidado con los tamaños, ya que, si uno de los *dataframes* es más grande (en columnas o filas) que otro, el resultado devolverá valores `NaN` donde no haya podido hacer el cálculo:

```
In [16]: # Multiplicar todos los valores por 2
dataframe_doble = dataframe * 2
dataframe_doble
```

Out[16]:

	columna1	columna2
a	20.0	10.0
b	4.0	12.0
c	18.0	NaN
d	16.0	14.0
e	NaN	16.0

```
In [17]: # Sumar 2 dataframes
dataframe_suma = dataframe_doble + dataframe
dataframe_suma
```

Out[17]:

	columna1	columna2
a	30.0	15.0
b	6.0	18.0
c	27.0	NaN
d	24.0	21.0
e	NaN	24.0

```
In [18]: # Aplicar la raiz cuadrada de todos los valores de un DataFrame
np.sqrt(dataframe_suma)
```

Out[18]:

	columna1	columna2
a	5.477226	3.872983
b	2.449490	4.242641
c	5.196152	NaN
d	4.898979	4.582576
e	NaN	4.898979

Por último, vamos a ver otra propiedad que puede resultar muy útil en las operaciones que hagamos con *dataframe.* Esta propiedad es la transpuesta, es decir, cambiar la orientación del *dataframe* para que las columnas sean las filas y viceversa. Para ello solo tenemos que acceder al atributo `T` del *dataframe:*

```
In [19]: dataframe.T
```

Out[19]:

	a	b	c	d	e
columna1	10.0	2.0	9.0	8.0	NaN
columna2	5.0	6.0	NaN	7.0	8.0

Funciones de gestión de datos

En este apartado enumeraremos algunas de las funciones más importantes que nos permiten gestionar los datos que tenemos en un *dataframe.* Para ver estas funciones, usaremos el siguiente *dataframe* de ejemplo.

```
In [20]: diccionario = {'user3': pd.Series(['Juan', 'Barcelona', 32, 'Soltero', 'Hombre'], index=['nombre','localidad','edad','estado civil', 'sexo']),
          'user4': pd.Series(['Alicia', 'Santander', 47, 'Casado', 'Mujer'], index=['nombre','localidad','edad','estado civil', 'sexo']),
          'user2': pd.Series(['Marcos', 'Madrid', 31, 'Casado', 'Hombre'], index=['nombre','localidad','edad','estado civil', 'sexo']),
          'user1': pd.Series(['Isabel', 'Zaragoza', 70, 'Soltero', 'Mujer'], index=['nombre','localidad','edad','estado civil', 'sexo'])}

         dataframe = pd.DataFrame(diccionario).T
         dataframe
```

Out[20]:

	nombre	localidad	edad	estado civil	sexo
user3	Juan	Barcelona	32	Soltero	Hombre
user4	Alicia	Santander	47	Casado	Mujer
user2	Marcos	Madrid	31	Casado	Hombre
user1	Isabel	Zaragoza	70	Soltero	Mujer

- **Ordenar por valor:** podemos ordenar las filas de los *dataframes* siguiendo el orden de un valor. Para ello usaremos la función `sort_values()` y en el parámetro `by` pondremos el nombre de la columna por la que queremos ordenar. También podemos cambiar la forma de ordenarlo con el parámetro `ascending`.

```
In [21]: dataframe.sort_values(by='edad')
```

Out[21]:

	nombre	localidad	edad	estado civil	sexo
user2	Marcos	Madrid	31	Casado	Hombre
user3	Juan	Barcelona	32	Soltero	Hombre
user4	Alicia	Santander	47	Casado	Mujer
user1	Isabel	Zaragoza	70	Soltero	Mujer

```
In [22]: dataframe.sort_values(by='nombre', ascending=False)
```

Out[22]:

	nombre	localidad	edad	estado civil	sexo
user2	Marcos	Madrid	31	Casado	Hombre
user3	Juan	Barcelona	32	Soltero	Hombre
user1	Isabel	Zaragoza	70	Soltero	Mujer
user4	Alicia	Santander	47	Casado	Mujer

- **Ordenar por índice:** otra forma de ordenar un *dataframe* es hacerlo por el valor de sus índices. Esto lo haremos con la función `sort_index()`. Como en la función anterior, también podemos definir cómo ordenarlo con el parámetro `ascending`.

```
In [23]: dataframe.sort_index()
```

Out[23]:

	nombre	localidad	edad	estado civil	sexo
user1	Isabel	Zaragoza	70	Soltero	Mujer
user2	Marcos	Madrid	31	Casado	Hombre
user3	Juan	Barcelona	32	Soltero	Hombre
user4	Alicia	Santander	47	Casado	Mujer

```
In [24]: dataframe.sort_index(ascending=False)
```

Out[24]:

	nombre	localidad	edad	estado civil	sexo
user4	Alicia	Santander	47	Casado	Mujer
user3	Juan	Barcelona	32	Soltero	Hombre
user2	Marcos	Madrid	31	Casado	Hombre
user1	Isabel	Zaragoza	70	Soltero	Mujer

- **Agrupar por valores:** un recurso muy común es agrupar los datos por valores y después aplicar otra operación como: contar el número de filas en cada grupo o la media de un atributo, etc. Para ello usamos la función `groupby()` y, en el parámetro `by`, asignamos las etiquetas de la columna o columnas (en forma de lista) en la que queremos agrupar los elementos.

  La función `groupby` no devuelve un *dataframe* por defecto. Es necesario aplicarle una función que nos indique qué queremos visualizar de cada dato:

```
In [25]: # Agrupar por estado civil contanto el número de elementos de cada grupo
         dataframe.groupby(by='estado civil').count()
```

Out[25]:

	nombre	localidad	edad	sexo
estado civil				
Casado	2	2	2	2
Soltero	2	2	2	2

- **Aplicar funciones map:** en el tema de las funciones vimos como podíamos aplicar funciones anónimas para crear nuevas listas aplicando algunas operaciones. En los *dataframes* podemos hacer lo mismo con la función `apply`. Este nos permite aplicar una función anónima a todos los elementos de una fila o columna, devolviendo como resultado una serie con los nuevos valores.

```
In [26]: # Obtener que personas son Solteras y con menos de 35 años
         dataframe.apply(lambda item: item['edad'] < 35 and item['estado civil'] == 'Soltero', axis=1)

Out[26]: user3     True
         user4    False
         user2    False
         user1    False
         dtype: bool
```

Funciones estadísticas

Al igual que en las series, los *dataframes* cuentan con diferentes funciones estadísticas que nos permiten observar la distribución de los datos y algunos valores descriptivos. Una función muy útil para comenzar estudiando los datos es la función `describe()`, que nos devuelve varios atributos de cada columna como: cuántos valores existen o cuántos valores únicos existen. Dependiendo del tipo de datos que almacenamos en el *dataframe,* obtendremos unas u otras características:

```
In [27]: dataframe.describe()
```

Out[27]:

	nombre	localidad	edad	estado civil	sexo
count	4	4	4	4	4
unique	4	4	4	2	2
top	Alicia	Santander	31	Soltero	Mujer
freq	1	1	1	2	2

```
In [28]: # Al contener solo valores numéricos, devuelve valores como la media, la desvicación típica, etc.
         dataframe_doble.describe()
```

Out[28]:

	columna1	columna2
count	4.000000	4.000000
mean	14.500000	13.000000
std	7.187953	2.581989
min	4.000000	10.000000
25%	13.000000	11.500000
50%	17.000000	13.000000
75%	18.500000	14.500000
max	20.000000	16.000000

Además, podemos aplicar las funciones estadísticas que hemos visto en las series. Sin embargo, solo se aplicarán a las columnas que tengan un tipo de dato compatible. Por ejemplo, la mediana solo se calculará en la columna `edad` del *dataframe.*

```
In [29]: dataframe.median()

Out[29]: edad    39.5
         dtype: float64
```

7.4. Lectura y escritura de ficheros CSV

A estas alturas ya sabemos crear un *dataframe* e ir incorporando nuevos datos, consultar subconjuntos de datos o modificar valores. Sin embargo, normalmente no tenemos que crear un *dataframe* vacío en el que vamos introduciendo la información. Lo más normal es encontrarnos un fichero con extensión csv que incluye todos los datos con los que vamos a trabajar.

`pandas` incluye una función que permite cargar los datos de un fichero csv en un *dataframe.* Esta función es `read_csv()`. Debemos pasarle como argumentos la ruta donde se encuentra el fichero y, si los valores en el csv están separados por un valor distinto a comas (,), definiremos el nuevo separador con el argumento `sep=`. A continuación, vemos un ejemplo de cómo leer un fichero csv que contiene información sobre películas.

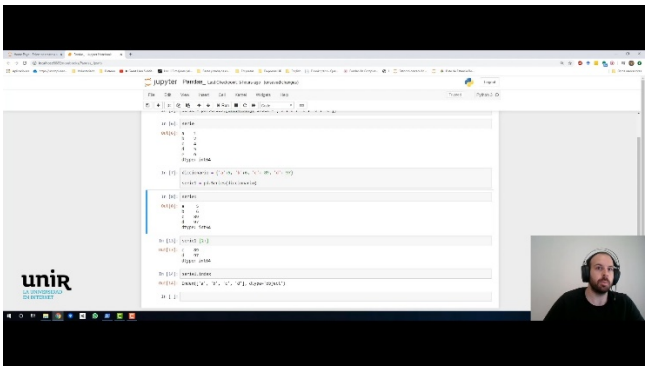
```
In [4]: peliculas = pd.read_csv('movies.csv')
        peliculas[:5]
```

Out[4]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Como podemos observar, esta función se encarga de incluir los datos correctamente en la estructura de un *dataframe.* También podemos guardar los *dataframes* que hagamos nosotros en un fichero csv. Para ello, solo tenemos que utilizar la instrucción `to_csv()` y pasar como argumento la ruta donde queremos almacenar este fichero.

```
In [5]: peliculas.to_csv('movies_2.csv')
```



Vídeo 2. Pandas.

Accede al vídeo a través del aula virtual