La librería *Pandas*

Pandas es una librería de Python para análisis y manipulación de datos. Permite explorar y procesar fácilmente datos, a través de dos tipos propios: las series (vectores de datos unidimensionales) y los *data frames* (tablas de datos bidimensionales). Es una capa por encima de la base proporcionada por *NumPy*, a la que añade una serie de funcionalidades nuevas para el procesamiento de los datos. Podríamos decir que *NumPy* es más apropiada para cálculos matemáticos, y *Pandas* para manipulación de datos. Aquí podemos encontrar más información, en su web oficial.

1. Primeros pasos

Lo primero que tendremos que hacer para trabajar con Pandas será instalar la librería con el correspondiente comando *pip*:

```
pip install pandas
```

Después, importaremos la librería en los ficheros Python que la necesiten. El *alias* habitual que se le suele poner (aunque no es obligatorio) es *pd*:

```
import pandas as pd
```

2. *Data frames*

Comenzaremos analizando los ***data frames***, que son las estructuras más habituales en Pandas. Como hemos comentado, son estructuras de datos bidimensionales, donde cada fila se identifica con un índice o etiqueta (*label*) que puede ser numérico o alfanumérico, y cada columna con otro índice o *label* que también puede ser numérico o alfanumérico. Esto permite que un *data frame* pueda representar cualquier tipo de información bidimensional en forma de tabla, y que podamos acceder a cada casilla en base a sus claves *fila - columna*. De hecho, podemos concebir un *data frame* como un conjunto de series agrupadas.

2.1. Creación de *data frames*

Veremos que existen distintas formas de crear un *data frame*, dependiendo de cómo conseguimos los datos.

2.1.1. Creación básica con datos directos

Para crear un *data frame* con Pandas, usamos su método `DataFrame`. Podemos pasarle como parámetro una lista o tabla bidimensional:

```
datos = [['Nacho', 44],['Juan', 70],['Ana', 40]]
dataFrame = pd.DataFrame(datos)
```

Automáticamente, Pandas asigna índices numéricos a las filas y las columnas. Si queremos especificar otras etiquetas alfanuméricas como columnas, podemos hacerlo usando un parámetro adicional llamado `columns`. Por ejemplo:

```
datosPersonas = pd.DataFrame(datos, columns=['Nombre', 'Edad'])
```

El resultado tras estas líneas de código será el siguiente *data frame*, que podemos imprimir por pantalla:

```
  Nombre  Edad
0  Nacho    44
1   Juan    70
2    Ana    40
```

También podemos crear *data frames* a partir de arrays de *NumPy*:

```
import numpy as np
import pandas as pd

datos = np.array([[1, 2], [3, 4], [5, 6]])
dataFrame = pd.DataFrame(datos)
```

2.1.2. Crear *data frames* a partir de diccionarios

Dado que un diccionario es una colección de datos que asocia una clave (típicamente alfanumérica) a un valor, podemos valernos de ellos para definir, a través de las claves, las etiquetas de un *data frame*. Los valores asociados a cada clave serán los valores de cada columna. En este caso, se supone que cada clave tiene asociada una secuencia de valores.

Veamos un ejemplo:

```
numeros = {}
numeros['pares'] = [2, 4, 6, 8]
numeros['impares'] = [1, 3, 5, 7]
# Alternativamente:
# numeros = { 'pares': [2, 4, 6, 8], 'impares': [1, 3, 5, 7]}
dataFrame = pd.DataFrame(numeros)
```

El resultado será el siguiente *data frame*:

```
   pares  impares
0      2        1
1      4        3
2      6        5
3      8        7
```

2.1.3. Crear *data frames* a partir de archivos CSV

Los archivos CSV son un medio bastante habitual de almacenar información textual, mediante datos separados por comas u otros separadores (punto y coma, etc). Por ejemplo:

```
nombre,edad,email,telefono
Nacho Iborra,44,nachoiborra@iessanvicente.com,611223344
Juan Pérez,70,juanp@gmail.com,699887766
May Calle,43, maycalle@iessanvicente.com,612345678
Mario,9,mario@hotmail.com,655443322
```

Podemos emplear la función `read_csv` de Pandas para cargar automáticamente la información de un archivo CSV en un *data frame*:

```
df = pd.read_csv('archivo.csv')
```

En el caso de que el archivo CSV utilice un separador diferente a la coma, podemos especificarlo con el parámetro `sep`. Por ejemplo:

```
df = pd.read_csv('archivo.csv', sep=';')
```

Existen otras formas de usar este método, que se pueden consultar en la documentación oficial. Por ejemplo, en el caso de que la primera fila del CSV no contenga los nombres de las columnas podemos indicar un

parámetro `header=None` para que se cuente esa primera fila como datos directamente:

```
df = pd.read_csv('archivo.csv', header=None)
```

> **NOTA:** si ejecutamos esta función desde un entorno en la nube como *Google Colab*, la ruta hacia el fichero dependerá de la ubicación de nuestro fichero de Google Colab. Si lo subimos a la misma carpeta donde se esté ejecutando remotamente nuestro proyecto, nos podría servir una instrucción como la anterior.

Podemos también exportar un *data frame* a formato CSV con la función `to_csv`. Normalmente conviene especificar un parámetro `index=False` si no queremos guardar los índices de fila (si son los numéricos predefinidos, no es necesario)

```
df.to_csv('otro_archivo.csv', index=False)
```

2.2. Acceso a los datos

Existen distintas formas de acceder a los datos de un *data frame* en Pandas, dependiendo de si queremos acceder a una casilla en concreto u obtener un rango de filas/columnas. Para ilustrar el ejemplo, partiremos de una tabla de datos como esta:

Nombre	Email	Edad	Telefono
Nacho	nacho@gmail.com	44	611223344
Juan	jperez@hotmail.com	70	699887766
Ana	anaib@gmail.com	40	619283746

Traducido a Pandas, quedaría algo así:

```
import pandas as pd

datos = { 'Nombre': ['Nacho', 'Juan', 'Ana'],
    'Email': ['nacho@gmail.com', 'jperez@hotmail.com',
    'anaib@gmail.com'], 'Edad': [44, 70, 40],
    'Telefono': ['611223344', '699887766', '619283746']}
dataFrame = pd.DataFrame(datos)
```

2.2.1. Acceso a casillas concretas

Para acceder a un dato concreto (casilla) de un *data frame* tenemos varias alternativas. Supongamos que queremos obtener el e-mail de la primera fila.

- Si utilizamos una nomenclatura similar a la usada en *NumPy* (`dataFrame[0, 1]`) o en las listas bidimensionales de Python (`dataFrame[0][1]`), no nos servirá, obtendremos un *Key error* porque no es la forma correcta de utilizar los índices en el *data frame*
- Disponemos de una propiedad llamada `loc` que permite indicar el índice de fila y el de columna, separados por comas. El índice de fila debe ser numérico, y el de columna deberá ser alfanumérico si las columnas tienen etiquetas (en otro caso, puede ser numérico).

```
email1 = dataFrame.loc[0, 1]        # Error
email2 = dataFrame.loc[0, 'Email']  # 'nacho@gmail.com'
```

- Alternativamente, tenemos la propiedad `iloc`, similar a la anterior pero especificando las posiciones numéricas de fila y columna.

```
email1 = dataFrame.iloc[0, 1]       # 'nacho@gmail.com'
```

- Finalmente, podemos emplear las propiedades `at` e `iat`, similares a las anteriores, para obtener el mismo resultado (usando índices alfanuméricos o numéricos para las columnas, respectivamente).

```
email1 = dataFrame.iat[0, 1]       # 'nacho@gmail.com'
email2 = dataFrame.at[0, 'Email']  # 'nacho@gmail.com'
```

2.2.2. Acceso a rangos de casillas

Podemos emplear las propiedades `loc` e `iloc` para obtener un rango de celdas, indicando la fila inicial y final, y la columna inicial y final (inclusive en el caso de *loc*, exclusive en el caso de *iloc*). También podemos indicar un conjunto separado por comas de filas o columnas que nos interesen.

```
# Columnas Nombre a Edad de las 4 primeras filas
celdas = dataFrame.loc[0:3, 'Nombre':'Edad']
# Columnas Nombre a Edad de las filas 5 y 9
celdas2 = dataFrame.loc[[5, 9], 'Nombre':'Edad']
# Columnas Nombre a Email de las filas 5 y 9 (Edad no se incluye)
celdas2 = dataFrame.iloc[[5, 9], 0:2]
```

Además, podemos seleccionar rangos de filas o columnas con los corchetes:

- Seleccionar un rango de filas indicando el número de fila inicial (inclusive), dos puntos y el número de fila final (exclusive).

```
# Nos quedamos con las filas 2, 3, 4
filasSeleccionadas = dataFrame[2:5]
```

- También podemos usar este operador de corchetes para quedarnos con un conjunto de columnas que nos interesen. Notar que, si sólo indicamos una columna, lo que obtenemos es una serie, no un *data frame*:

```
# Serie
nombres = dataFrame['Nombre']
# Data frame
nombreYEdad = dataFrame[['Nombre', 'Edad']]
```

En todos estos casos obtenemos como resultado un sub-*data frame* del original (o una serie, si sólo hemos seleccionado una columna)

2.2.3. Cambiar el índice de las filas

Hasta ahora, las filas de un *data frame* han sido índices numéricos a partir del 0. Podemos cambiar esto, y hacer que los índices de las filas sean los valores de alguna de las columnas. Por ejemplo, en el caso anterior podríamos hacer que los índices de filas fueran los distintos e-mails de los usuarios. Para ello, usaremos el método `set_index` del *data frame*, indicando el nombre de columna que queremos usar para indexar.

```
dataFrame = dataFrame.set_index('Email')
```

Esto hará que las filas ya no se identifiquen como la 0, 1, 2, sino como la fila de *nacho@gmail.com*, etc. Alternativamente, se puede utilizar esta segunda versión con el parámetro `inplace=True`, que actualiza los cambios sobre el *data frame* original, para no tener que reasignarlo, ya que la anterior opción genera una copia del original.

```
dataFrame.set_index('Email', inplace=True)
```

Esto permitirá que, a través de la instrucción `loc` vista antes, podamos acceder a los datos de una fila por este nuevo índice. Así obtendríamos, por ejemplo, la edad de *nacho@gmail.com*:

```
edad = dataFrame.loc['nacho@gmail.com', 'Edad']
```

En el caso de que queramos resetear el índice y volver a la numeración original de filas 0, 1, 2... usamos el método `reset_index`. Podemos indicar en el parámetro `drop=True` que queremos borrar el índice previo.

```
dataFrame.reset_index(drop=True, inplace=True)
```

> **NOTA**: el parámetro `drop=True` lo usaremos SOLO cuando queramos borrar la columna que hacía de índice, ya que de lo contrario perderemos esa información. En el ejemplo anterior, dejaríamos de tener disponible la columna `Email`, por lo que no es muy habitual hacerlo así.

2.2.4. Acceso a los nombres de columnas

En algunas ocasiones nos puede interesar acceder a los nombres de columnas. Por ejemplo, para recorrerlos y mostrar estadísticas o hacer operaciones secuencialmente con cada columna. Para ello accedemos a la propiedad `columns` del *data frame*. Si queremos obtenerlo en forma de lista, lo pasamos también por el método `tolist`:

```
columnas = dataFrame.columns.tolist()
```

2.3. Tipos de datos en pandas

Cuando trabajamos con un *data frame* cada columna puede ser de un tipo diferente. Pandas asigna automáticamente un tipo por defecto a cada columna, en función de la información que hay almacenada en ella. Podemos ver estos tipos con la propiedad `dtypes`, de forma similar a *NumPy*.

```
# Muestra un listado con los tipos de cada columna
print(df.dtypes)
```

Es posible que alguno de los tipos asignados no nos cuadre, y queramos cambiarlo. Por ejemplo, que haya asignado un valor real a datos que queremos que sean enteros. Para ello, debemos seleccionar la columna (o columnas) afectada(s) y aplicarles el cambio de tipo. Existen varias formas de hacerlo, pero podríamos hacerlo así, por ejemplo:

```
# Hacemos que 'Edad' sea entero de 32 bits
df = df.astype({'Edad': 'int32'})
```

> **NOTA**: es **IMPORTANTE** tener en cuenta que el cambio en los tipos de datos sólo se mantiene en tiempo de ejecución. Si guardamos los datos de nuevo a fichero y los volvemos a recuperar, se volverán a asignar tipos por defecto que no tienen por qué coincidir con los que hemos establecido en el código. Así, este paso se suele realizar para asegurarnos en la ejecución de que los datos son de un cierto tipo, o para ahorrar memoria (por ejemplo, transformar enteros de 64 bits en enteros de 32 bits).

> **Ejercicio 1:**
>
> Crea un programa llamado **VentasEmpresa.py** que cree un *data frame* con los datos de la siguiente tabla, y los muestre por pantalla:

Mes	Ventas	Gastos
Enero	20600	17900
Abril	22500	18500
Julio	15400	17600
Octubre	21100	18200

> **Ejercicio 2:**
>
> Crea un programa llamado **CasasRurales.py** que cargue en un *data frame* los datos de este archivo CSV de casas rurales de la provincia de Castellón. Queremos quedarnos sólo con las 4 primeras columnas (*id*, *localidad*, *codigo_postal* y *nombre*), transformando el *id* y el *codigo_postal* a enteros de 32 bits. Guarda el resultado en un archivo llamado *casas_rurales_resumen.csv*.
>
> **AYUDA:** vídeo con la solución del ejercicio

2.4. Algunas operaciones básicas

Veamos a continuación algunas funciones básicas de uso con *data frames*:

- Las funciones `head` y `tail` nos obtienen por defecto las primeras/últimas 5 filas del *data frame*. En caso de querer otra cantidad, debemos pasarla como parámetro:

```
primeros5 = df.head()
ultimos3 = df.tail(3)
```

- La función `describe` (sin parámetros) obtiene estadísticas de cada columna del *data frame* (valor máximo, mínimo, media...)
- La función `info` (sin parámetros) obtiene información sobre el tipo de dato de cada columna, y conteo de valores no nulos

```
print(df.describe())
df.info()
```

2.5. Filtrado

En ocasiones nos interesa obtener sólo los elementos o filas de un *data frame* que cumplan una cierta condición. Esto puede hacerse de varias formas. Por ejemplo, podemos establecer una condición que se aplique sobre una (o varias) columnas del *data frame*, y luego obtener un *data frame* alternativo con esa condición (filtrando las filas que la cumplan). El siguiente ejemplo se queda con las personas mayores de edad del *data frame* de un ejemplo anterior:

```
condicion = df['Edad'] >= 18
adultos = df[condicion]
```

Lo que hace la primera instrucción es crear un array de booleanos, poniendo a *False* los correspondientes a las filas que no pasan el filtro, y a *True* las que sí lo hacen. Luego, la segunda instrucción pasa este array de booleanos como parámetro a `df`, para filtrar los que sean *True*. Sería algo equivalente a hacer algo así, de forma manual:

```
adultos = df[[True, True, True, False]]
```

De forma adicional, podemos usar los operadores `&` y `|` para enlazar condiciones simples. Por ejemplo, así obtendríamos las personas entre 30 y 50 años:

```
condicion = (df['Edad'] >= 30) & (df['Edad'] <= 50)
rango = df[condicion]
```

Alternativamente, podemos pasar la condición (o condiciones enlazadas) como dato dentro de los corchetes, en lugar de crear la variable intermedia `condicion`:

```
rango = df[(df['Edad'] >= 30) & (df['Edad'] <= 50)]
```

2.6. Inserciones y borrados

Podemos **añadir filas** a nuestros *data frames* usando la instrucción `loc` vista antes para localizar celdas o rangos de celdas. Si la fila ya existe, se sobreescribe su contenido por el nuevo, y si no existe se crea. Hay que

tener en cuenta que el número de datos que pasemos debe ser igual que el número de columnas de nuestro *data frame*.

```
# Añadimos una nueva persona con su nombre, edad, email y teléfono
df.loc[6] = ['Pepe', 65, 'pepe123@gmail.com', '675849302']
```

Para **añadir columnas** en el *data frame*, ponemos el nuevo nombre de la columna entre corchetes, y le pasamos los valores para esa nueva columna (debe haber tantos valores como filas tenga nuestro *data frame*):

```
# Añadimos columna "localidad" al listado de personas
df['localidad'] = ['Alicante', 'Murcia', 'San Vicente']
```

A la hora de **borrar filas o columnas** de un *data frame*, usamos la instrucción `drop`, especificando:

- El número o etiqueta de fila / columna que queremos borrar
- Un parámetro `axis` que deberemos poner a 0 para indicar que queremos borrar una fila, y a 1 para una columna
- Un parámetro `inplace=True` para asegurarnos de alterar el *data frame* original (de lo contrario quedaría inalterado)

```
# Borramos fila 2
df.drop(2, axis=0, inplace=True)
# Borramos columna 'localidad'
df.drop('localidad', axis=1, inplace=True)
```

2.7. Reemplazos

La instrucción `replace` nos puede resultar muy útil para sustituir unos valores por otros en una(s) determinada(s) columna(s). La invocaremos sobre la columna donde queremos hacer el reemplazo y le podemos pasar dos vectores: uno con los valores que queremos reemplazar, y otro con los valores correspondientes del reemplazo.

El siguiente ejemplo actualiza la columna *activo* de un *data frame*, y reemplaza todos los valores 0 por "NO" y 1 por "SI":

```
datos['activo'] = datos['activo'].replace([0, 1], ['NO','SI'])
```

2.8. Ordenaciones

Podemos emplear la instrucción `sort_values` para ordenar los datos de un *data frame* respecto a una de sus columnas, especificada en el parámetro `by`. Esta instrucción ordena de forma ascendente por defecto, si queremos un orden descendente debemos especificar un parámetro `ascending=False` (por defecto es *True*).

El siguiente ejemplo ordena los datos del *data frame* anterior por la columna *edad*, de mayor a menor:

```
df = df.sort_values(by='Edad', ascending=False)
```

Cuando hacemos ordenaciones es posible que nos interese reindexar la colección de datos, ya que los índices numéricos antiguos acompañan a cada fila cuando se reordenan. Dicho de otro modo, si el elemento de la fila 112 del *data frame* pasa a ser el primero tras la ordenación, deberemos seguir refiriéndonos a él como el elemento 112. Puede que esto no interese, y queramos que ahora ése sea el elemento 0. Entonces tendremos que hacer algo como esto:

```
df.reset_index(drop=True, inplace=True)
```

2.9. Concatenaciones y agrupaciones

Podemos concatenar *dataframes* vertical u horizontalmente con la instrucción `concat` de Pandas.

```
# Concatenación vertical de df1 y df2
# (unas filas a continuación de otras)
resultado1 = pd.concat([df1, df2])
# Concatenación horizontal de df1 y df2
# (unas columnas a continuación de otras)
resultado2 = pd.concat([df1, df2], axis=1)
```

La instrucción `groupby` permite agrupar filas de un *data frame* por alguna de sus características (típicamente un valor de una columna), y así poder hacer operaciones específicas con esos grupos. Por ejemplo, esta instrucción calcula la media de edades de las personas agrupadas por su localidad:

```
datos = [['Alicante', 44],['Murcia', 70],
         ['Alicante', 40], ['Murcia', 55]]
df = pd.DataFrame(datos, columns=['localidad', 'edad'])
print(df.groupby(df['localidad']).mean())
```

2.10. Otras operaciones

Repasamos aquí brevemente otras operaciones disponibles con los *data frames* de Pandas:

- La instrucción `value_counts` permite contar cuántas muestras (filas) hay para cada uno de los valores de una columna categórica (nominal). Por ejemplo, podríamos ver cuántas personas hay de cada localidad:

```
df['localidad'].value_counts()
```

- La instrucción `cut` corta el *dataset* en varias secuencias, a partir de un conjunto de marcas (*bins*) para una columna dada. Por ejemplo, podríamos añadir una columna más al *dataset* anterior llamada *categoria_edad* que divida a los individuos según la franja de edad: de 0 a 12 años, de 12 a 18, de 18 a 65 y de más de 65, asignando a cada franja un valor de *Niño*, *Adolescente*, *Adulto*, *Anciano*:

```
df['categoria_edad'] = pd.cut(df['edad'],
    bins = [0, 12, 18, 65, np.inf],
    labels = ['Niño', 'Adolescente', 'Adulto', 'Anciano'])
```

> **Ejercicio 3:**
>
> Crea un programa llamado **ParquesEolicosProvincia.py** que utilice este archivo CSV sobre parques eólicos de la comunidad de Castilla y León. Nos interesa quedarnos con los parques de la provincia de Zamora que tengan más de 10 aerogeneradores. Muestra el listado resultante
>
> **AYUDA:** vídeo con la solución del ejercicio

> **Ejercicio 4:**
>
> Sobre el mismo archivo anterior, crea un nuevo programa **ParquesEolicosPotencia.py** que ordene los parques eólicos de mayor a menor potencia total, y se quede con el nombre, municipio, provincia y potencia de los 10 primeros.

> **Ejercicio 5:**
>
> Sobre el mismo archivo anterior, crea un programa **ParquesEolicosTotalesProvincia.py** que cuente cuántos parques eólicos hay en total para cada provincia de Castilla y León.

3. Series

Pasamos ahora a analizar las **series**, que son estructuras de datos unidimensionales. Constan de una única fila (o columna), donde podemos acceder a cada elemento por la posición que ocupa (comenzando por 0, como es habitual) o bien por una etiqueta que asociemos a cada casilla.

3.1. Creación de series

Para crear series en *Pandas* usamos el método `Series`, indicando el vector de datos con el que queremos crear la serie:

```
import pandas as pd

datos = [1, 5, 10]
serie = pd.Series(datos)
print(serie[0])         # 1
```

Como decimos, podemos especificar un conjunto de etiquetas adicional:

```
serie = pd.Series(datos, index=['a', 'b', 'c'])
print(serie['b'])       # 5
```

Alternativamente, podemos emplear un diccionario para especificar las etiquetas junto con sus valores asociados (que deberán ser valores simples, ya que de lo contrario sería un *data frame*):

```
datos = {'a': 1, 'b': 5, 'c': 10}
serie = pd.Series(datos)
print(serie['c'])       # 10
```

3.2. Otras operaciones

Podemos aplicar sobre las series muchas de las operaciones que se aplican sobre los *data frames*. Por ejemplo, obtener subrangos de una serie, con una sintaxis similar a la utilizada en *NumPy*:

```
# Dos primeros elementos de la serie
print(serie[:2])
```

Otras operaciones como `tail`, `head`, `loc` o `iloc` también están disponibles en las series, del mismo modo que las utilizamos en los *data frames*.

```
print(serie.head())     # 5 primeros elementos
print(serie.tail(3))    # 3 últimos elementos
```

> **Ejercicio 6:**

> Crea un programa llamado **SerieNotas.py** que cree una serie con las notas de unos alumnos (identificados por su código NIA), y luego muestre por pantalla las notas de los alumnos aprobados, ordenadas de mayor a menor.