TEMA 1 (parte 2)

PANDAS

Fundamentos

Pandas es una librería de Python especializada en el **manejo** y **análisis** de estructuras de **datos**. Sus características son:

- Define nuevas estructuras de datos basadas en los arrays de NumPy pero con nuevas funciones.
- Permite leer y escribir fácilmente ficheros en formato **CSV, Excel y bases de datos SQL**, entre otros.
- Permite acceder a los datos mediante **índices o nombres para filas y columnas**.
- Ofrece métodos para **reordenar, dividir y combinar** conjuntos de datos.
- Permite trabajar con series temporales (indexadas por un timestamp/marca temporal).
- Realiza todas estas operaciones de manera muy eficiente.

Veremos 2 tipos de estructuras de datos, las **Series y los DataFrame**

Fundamentos

Pandas es como una evolución de NumPy, ya que permite estructuras de datos y operaciones más complejas, pero la forma de trabajar es similar. La estructura de datos de NumPy es el ndarray, mientras que en Pandas existen 2, las Series y DataFrames.

Similitudes NumPy-Pandas:

- Ambas estructuras se pueden indexar de manera similar, hacer slicing e indexación booleana.
- Las funciones vectorizadas funcionan de manera similar.

Diferencias NumPy-Pandas:

La principal diferencia radica en la estructura de datos. Si en NumPy trabajamos con un ndarray (que puede ser multidimensional) que no es más que un array clásico y se puede indexar por su índice (que corresponde con la posición del dato) en Pandas trabajamos con Series, que es similar a un doble array, donde en uno se guardan los datos y en el otro se almacena el “nombre” de cada dato o un índice con nombre.

Un DataFrame es un conjunto de Series que comparten el mismo índice. Se puede ver como una tabla, donde cada Series es una columna con su propio nombre y tipo de datos. Todas las columnas comparten el mismo índice que puede ser numérico o alfanumérico.

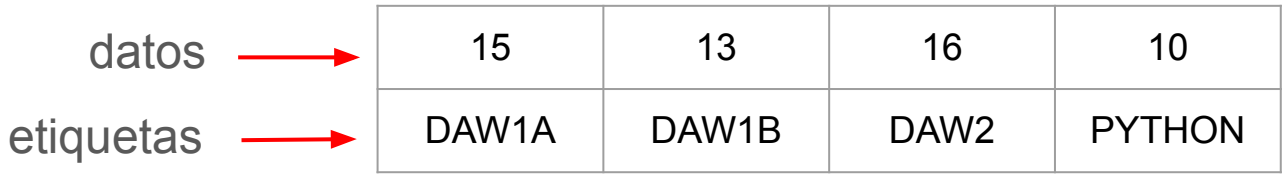
Series

Vamos a ver los 2 principales tipos estructurados que define Pandas para el manejo de datos:

- **Series**: Estructura unidimensional de datos indexados. Es como un ndarray de 1 dimensión de NumPy que tiene asociada una etiqueta a cada posición.

Por ejemplo, tenemos los datos de alumnos matriculados en cada curso. La etiqueta será nombre del curso. Podemos indexar los datos por posición o por etiqueta.

Las etiquetas se pueden repetir en la Series y son opcionales. Si no indicamos etiquetas se generan automáticamente de tipo entero al mismo número que el índice.



Series (creación y atributos)

`Creación de series:`

`Series(data=lista, index=índices, dtype=tipo)` : Devuelve un objeto de tipo Series con los datos de la lista `lista`, con un índice indicado en `índices` y el tipo de datos indicado en `tipo`. Si no se pasa la lista de índices se utilizan como índices los enteros del 0 a N-1, siendo N el tamaño de la serie. Si no se pasa el tipo de dato se infiere.

`Series(data=diccionario, index=índices)`: Devuelve un objeto de tipo Series con los valores del diccionario `diccionario` y las filas especificados en la lista `índices`. Si no se pasa la lista de índices se utilizan como índices las claves del diccionario.

Atributos:

- `s.size` : Devuelve el número de elementos de la serie.
- `s.index` : Devuelve una lista con los índices.
- `s.dtype` : Devuelve el tipo de datos de los elementos de la serie `s`.
- s.name : nombre de la serie

Series (creación)

```python
import pandas as pd

# Creamos la Series pasando los datos y las etiquetas
serie1 = pd.Series([15, 13, 18, 10], index=["DAW1A", "DAW1B", "DAW2", "PYTHON"])

# Podemos asignar un nombre a la serie
serie1.name = "Matriculados"
print(serie1)
```

```
DAW1A     15
DAW1B     13
DAW2      18
PYTHON    10
Name: Matriculados, dtype: int64
```

Series (indexación por posición o etiquetas)

Si bien se puede indexar una Seriesl igual que a una lista o array este método está **Obsoleto** y lanzará un aviso. Se recomienda usar iloc y loc para que no haya ambigüedad al indexar por posición o por etiqueta.

Acceso por posición: iloc[ ].

- `Series.iloc[i]` : Devuelve el elemento que ocupa la posición `i (comenzando en 0)` en la serie `s`.
- `Series.iloc[[posiciones]]`: Devuelve otra serie con los elementos que ocupan las posiciones de la lista `posiciones`.
- `Series.iloc[inicio:fin]:` Indexación mediante un slicing.

Acceso por etiqueta: loc[ ]

- `Series.loc[etiqueta]` : Devuelve el elemento con el nombre `nombre` en el índice.
- `Series.oc[[etiquetas]]` : Devuelve otra serie con los elementos correspondientes a las etiqeutas indicadas en la lista `etiquetas` en el índice.
- `Series.loc[inicio_etiqueta : final_etiqueta]:` Indexa los elementos entre ambas etiquetas (incluidas)

Series (indexación por posición)

```python
# Creamos la Series pasando los datos y las etiquetas
serie1 = pd.Series([15, 13, 18, 10], index=["DAW1A", "DAW1B", "DAW2", "PYTHON"])

# Acceso por posición única
print(serie1.iloc[0])
```

```
15
```

```python
# Acceso por lista de posiciones
print(serie1.iloc[[1, 2]])
```

```
DAW1B    13
DAW2     18
dtype: int64
```

```python
# Acceso por un slicing de posiciones
print(serie1.iloc[1:3])
```

```
DAW1B    13
DAW2     18
dtype: int64
```

Series (indexación por etiqueta)

```python
# Creamos la Series pasando los datos y las etiquetas
serie1 = pd.Series([15, 13, 18, 10], index=["DAW1A", "DAW1B", "DAW2", "PYTHON"])

# Acceso por etiqueta única
print(serie1.loc["PYTHON"])
```

```
10
```

```python
# Acceso por lista de etiquetas
print(serie1.loc[["PYTHON", "DAW1A"]])
```

```
PYTHON    10
DAW1A     15
dtype: int64
```

```python
# Acceso por rango de etiquetas
print(serie1.loc["DAW1A": "DAW2"])
```

```
DAW1A    15
DAW1B    13
DAW2     18
dtype: int64
```

Series (indexación booleana)

La indexación booleana funciona igual que en Numpy. Generamos una Serie booleana a través de una comparación que servirá para indexar los elementos que cumplen la condición.

```python
# Creamos la Series pasando los datos y las etiquetas
serie1 = pd.Series([15, 13, 18, 10], index=["DAW1A", "DAW1B", "DAW2", "PYTHON"])

# Indexación booleana
print(serie1[serie1 >= 15])
```

```
DAW1A    15
DAW2     18
dtype: int64
```

Resumen descriptivo de la serie

- `s.count()` : Devuelve el número de elementos que no son nulos ni `NaN` en la serie `s`.
- `s.sum()` : Devuelve la suma de los datos de la serie `s` cuando los datos son de un tipo numérico, o la concatenación de ellos cuando son del tipo cadena `str`.
- `s.cumsum()` : Devuelve una serie con la suma acumulada de los datos de la serie `s` cuando los datos son de un tipo numérico.
- `s.value_counts()` : Devuelve una serie con la frecuencia (número de repeticiones) de cada valor de la serie.
- `s.min()` : Devuelve el menor de los datos de la serie `s`.
- `s.max()` : Devuelve el mayor de los datos de la serie `s`.
- `s.mean()` : Devuelve la media de los datos de la serie `s` cuando los datos son de un tipo numérico.
- `s.var()` : Devuelve la varianza de los datos de la serie `s` cuando los datos son de un tipo numérico.
- `s.std()` : Devuelve la desviación típica de los datos de la serie `s` cuando los datos son de un tipo numérico.
- `s.describe()`: Devuelve una serie con un resumen descriptivo que incluye el número de datos, su suma, el mínimo, el máximo, la media, la desviación típica y los cuartiles

Series (operaciones)

Los operadores binarios (`+`, `*`, `/`, etc.) pueden utilizarse con una serie y devuelve otra serie con el resultado de aplicar la operación a cada elemento de la serie. **Igual que en NumPy**

También es posible aplicar una función a cada elemento de la serie mediante el siguiente método `Series.apply(function, args=())` : Devuelve una serie con el resultado de aplicar la función a cada uno de los elementos de la serie.

Ordenar una serie:

- `Series.sort_values(ascending=True, inplace=False)` : Devuelve la serie ordenada por los valores. Si argumento del parámetro `ascending` es `True (por defecto)` el orden es creciente y si es `False` decreciente. Si `inplace=True` no retorna nada y ordena el propio objeto.

- `Series.sort_index(ascending=True, inplace=False)` : En este caso la ordenación es por etiquetas.

Series (operaciones)

Eliminar los dados desconocidos en una serie: Los datos desconocidos representan en Pandas por `NaN` y los nulos por `None`.

Tanto unos como otros suelen ser un problema a la hora de realizar algunos análisis de datos, por lo que es habitual eliminarlos. Para eliminarlos de una serie se utiliza el siguiente método:

- `s.dropna()` : Devuelve una serie eliminando los datos desconocidos o nulos de la serie `s`.

DataFrame

- **DataFrame:** Puede entenderse como una tabla. Es una estructura bidimensional donde las columnas tienen un nombre y un tipo, y las filas poseen un índice y puede tener una etiqueta. Realmente un DataFrame es un conjunto de Series que comparten el mismo índice/etiquetas.

La forma más común de crear un DataFrame es mediante un diccionario, donde los datos son listas de igual tamaño: diccionario = {clave1: [datos1], clave2: [datos2], …}

Por ejemplo, quiero almacenar los matriculados de varios años:

```
datos = {2023: [15, 13, 18, 10], 2022: [12, 15, 13, 12], 2021: [16, 17, 11, 14]}
cursos=["DAW1A", "DAW1B", "DAW2", "PYTHON"]
```

DataFrame (creación)

- **Con diccionario**: DataFrame(data=diccionario, index=filas, columns=columnas, dtype=tipos)

  Menos el diccionario, el resto de parámetros es opcional. index es el nombre de las filas, si no se pasa será un entero, de 0 a N-1. Columnas son las columnas que tendrá el DataFrame, estas deben existir en el diccionario, sino, su valor estará a NaN.

- **Con una lista de listas**: DataFrame(data=listas, index=filas, columns=columnas, dtype=tipos)

  En la lista de listas, cada lista es una fila. columns es una lista con el nombre de las columnas.

  ```
  df = pd.DataFrame([ ['María', 18], ['Luis', 22], ['Carmen', 20] ], columns=['Nombre', 'Edad'])
  ```

- **ndArray NumPy**: DataFrame(data=ndarray, index=filas, columns=columnas, dtype=tipo)

  Similar a lista de listas, una matriz de datos, cada fila de la matriz es una fila del DataFrame.

DataFrame (creación a través de diccionario)

Las claves del diccionario serán los nombres de las columnas. El valor de cada elemento del diccionario es una lista que contiene los datos:

```
datos = {2023: [15, 13, 18, 10], 2022: [12, 15, 13, 12], 2021: [16, 17, 11, 14]}
cursos=["DAW1A", "DAW1B", "DAW2", "PYTHON"]
tabla = pd.DataFrame(datos)

# Asigno el índice a posteriori
tabla.index = cursos
tabla  # Sin el print se muestra así en jupyter
```

	2023	2022	2021
DAW1A	15	12	16
DAW1B	13	15	17
DAW2	18	13	11
PYTHON	10	12	14

Atributos

- `df.shape` : Devuelve una tupla con el número de filas y columnas del DataFrame `df`.
- `df.size` : Devuelve el número de elementos del DataFrame.

  `df.columns` : Devuelve una lista con los nombres de las columnas del DataFrame `df`.
- `df.index` : Devuelve una lista con los nombres de las filas del DataFrame `df`.
- `df.dtypes` : Devuelve una serie con los tipos de datos de las columnas del DataFrame `df`.
- `df.values` : Devuelve un ndarray con los datos del DataFrame (se recomienda usar función **df.to_numpy()**

Funciones básicas:

- `df.info()` : Devuelve información (número de filas, número de columnas, índices, tipo de las columnas y memoria usado) del `df`.
- `df.head(n)` : Devuelve las `n` primeras filas del DataFrame `df`.
- `df.tail(n)` : Devuelve las `n` últimas filas del DataFrame `df`.

Configurar índices de filas y nombres de columnas

`Seleccionar datos de 1 columna como índice:`

`df.set_index(keys = columnas, verify_integrity = bool)`: Devuelve el DataFrame que resulta de eliminar las columnas de la lista `columnas` y convertirlas en el nuevo índice. El parámetro `verify_integrity` recibe un booleano (`False` por defecto) para realizar una comprobación de si existen claves duplicadas.

Reordenar filas o columnas o introducir nuevas o filtrar:

`df.reindex(index=filas, columns=columnas, fill_value=relleno)` : Devuelve el DataFrame que resulta de tomar del DataFrame `df` las filas con nombres en la lista `filas` y las columnas con nombres en la lista `columnas`. Si alguno de los nombres indicados en `filas` o `columnas` no existía en el DataFrame `df`, se crean filas o columnas nuevas rellenas con el valor `relleno`.

DataFrame: indexación por columnas directamente

```
# Mostrar 1 columna
print(tabla[2023])
```

```
DAW1A     15
DAW1B     13
DAW2      18
PYTHON    10
Name: 2023, dtype: int64
```

```
# Mostrar varias, pasar una lista de columnas
print(tabla[[2023, 2021]])
```

```
         2023  2021
DAW1A      15    16
DAW1B      13    17
DAW2       18    11
PYTHON     10    14
```

DataFrame: indexación por filas

Se utilizan iloc y loc igual que con las Series.

Podemos indexar una fila, varias en una lista, un slicing por índice, un rango por etiquetas.

```python
# Acceso a filas por nombre
print(tabla.loc["PYTHON"])
```

```
2023    10
2022    12
2021    14
Name: PYTHON, dtype: int64
```

```python
# Acceso a filas por posición
print(tabla.iloc[3])
```

```
2023    10
2022    12
2021    14
Name: PYTHON, dtype: int64
```

DataFrame: indexación por filas / col (etiqueta o nº índice)

Indexar con loc e iloc indicando [fila, columna]

	2023	2022	2021
DAW1A	15	12	16
DAW1B	13	15	17
DAW2	18	13	11
PYTHON	10	12	14

```python
# Una fila y una columna por índice
print(df.iloc[0, 0])
```

```
15
```

```python
# Una fila y una columna por etiqueta
print(df.loc['DAW1A', 2023])
```

```
15
```

```python
# Slice de filas y columnas índices
df.iloc[0:2, 1:2]
```

DataFrame: indexación por filas / col (etiqueta o nº índice)

Incluso rangos o slicing

	2023	2022	2021
DAW1A	15	12	16
DAW1B	13	15	17
DAW2	18	13	11
PYTHON	10	12	14

```
# Slice de filas y columnas índices
df.iloc[0:2, 1:2]
```

	2022
DAW1A	12
DAW1B	15

```
# Rango etiquetas fila, columna
df.loc["DAW1B":"PYTHON", 2023:2022]
```

	2023	2022
DAW1B	13	15
DAW2	18	13
PYTHON	10	12

DataFrame: indexación encadenada

Podemos usar indexación mediante índices numéricos o indexación booleana y al resultado obtenido le aplicamos indexación por columna simple.

```
#  Deseamos ver las matrículas de los cursos con más de 15 titulados
# Indexación booleana para la condición df[df['Titulados'] > 13]
# Al resultado alterior indexamos por la columna matrículas
df[df['Titulados'] > 13]['Matriculas']
```

```
DAW1A    22
DAW1B    20
Name: Matriculas, dtype: int64
```

	Matriculas	Titulados
DAW1A	22	20
DAW1B	20	15
DAW2	25	13
PYTHON	15	12

Objeto index

Cada DataFrame tiene 2 objetos index que almacenan los nombres de las columnas y las etiquetas de las filas.

Los índices son inmutables (no se puede modificar el valor de una posición en concreto, como en las tuplas). Y también funcionan como conjuntos, soportando las típicas operaciones de intersección, unión y diferencia, aunque en los índices se permiten valores duplicados.

Métodos:

- index.append(ad_index) Concatena 'ad_index' a 'index' y retorna un nuevo índice
- index.difference(index, sort=None) Diferencia de conjuntos
- index.intersection(index, sort=False) Intersección de conjuntos
- index.union(index, sort=None) Unión de conjuntos
- index.isin(colección) Retorna un array booleano indicando si cada valor está contenido en la ‘colección’.
- delete() Compute new Index with element at Index i deleted
- drop() Compute new Index by deleting passed values
- insert() Compute new Index by inserting element at Index i
- is_monotonic Returns True if each element is greater than or equal to the previous element
- is_unique: Propiedad. Almacena True si el índice no tiene valores duplicados.
- unique() Compute the array of unique values in the Index

Además soporta el operador 'is' para comprobar la pertenencia al conjunto de 1 elemento en concreto.

Resumen descriptivo

Similar a la Series, mostrando información de cada columna

- `df.count()` : Devuelve una serie con el número de elementos que no son nulos ni `NaN` en cada columna del DataFrame `df`.
- `df.sum()` : Devuelve una serie con la suma de los datos de las columnas del DataFrame `df` cuando los datos son de un tipo numérico, o la concatenación de ellos cuando son del tipo cadena `str`.
- `df.cumsum()` : Devuelve un DataFrame con la suma acumulada de los datos de las columnas del DataFrame `df` cuando los datos son de un tipo numérico.
- `df.min()` : Devuelve una serie con los menores de los datos de las columnas del DataFrame `df`.
- `df.max()` : Devuelve una serie con los mayores de los datos de las columnas del DataFrame `df`.
- `df.mean()` : Devuelve una serie con las medias de los datos de las columnas numéricas del DataFrame `df`.
- `df.var()` : Devuelve una serie con las varianzas de los datos de las columnas numéricas del DataFrame `df`.
- `df.std()` : Devuelve una serie con las desviaciones típicas de los datos de las columnas numéricas del DataFrame `df`.
- `df.cov()` : Devuelve un DataFrame con las covarianzas de los datos de las columnas numéricas del DataFrame `df`.
- `df.corr()` : Devuelve un DataFrame con los coeficientes de correlación de los pares de datos de las columnas numéricas del DataFrame `df`.
- `df.describe(include = tipo)` : Devuelve un DataFrame con un resumen estadístico de las columnas del DataFrame `df` del tipo `tipo`. Para los datos numéricos (`number`) se calcula la media, la desviación típica, el mínimo, el máximo y los cuartiles. Para los datos no numéricos (`object`) se calcula el número de valores, el número de valores distintos, la moda y su frecuencia. Si no se indica el tipo solo se consideran las columnas numéricas.

Eliminar Columnas

Para eliminar columnas de un DataFrame se utilizan los siguientes métodos:

- `del d[nombre]` : Elimina la columna con nombre `nombre` del DataFrame `df`.

- `df.pop(nombre)` : Elimina la columna con nombre `nombre` del DataFrame `df` y la devuelve como una serie.

Métodos aritméticos y evitar valores NaN

Pandas admite operaciones vectorizadas al igual que los array de Numpy, pero además iguala las etiquetas filas. Es decir, si sumamos 2 Series con diferente orden pero etiquetas filas en común, las “alinea”. Para las etiquetas que no están en ambas Series el resultado será NaN. Para evitarlo podemos hacer operaciones aritméticas mediante los métodos que permiten rellenar los NaN con un valor.

En el ejemplo hacemos serie1 - serie2 y rellenamos los nulos que se generen con 0.

```
# Restamos a la Serie s1 los valores de s2 (s1 - s2)
s1.sub(s2, fill_value=0)
```

Los métodos son: add, sub, div, floordiv (división entera), mul, pow (potencia).

Además, las Series y DF se pueden pasar a las funciones vectorizadas de Numpy. Por ejemplo para calcular la raíz cuadrada de cada valor de la serie -> pandas.sqrt(serie)

Apply

El método Apply existe en el objeto Series y en DataFrame.

Al aplicarlo al DataFrame la función a aplicar puede recibir las columnas (se aplica en el eje vertical) o bien las filas (se aplica en el eje horizontal). En el ejemplo la función ‘función1’ devuelve el máximo de un array y se aplica en el eje horizontal (a cada fila).

```
# Busca en el eje horizontal (máximo de cada fila)
df.apply(funcion1, axis='columns') # columns o axis=1 son equivalentes
```

```
Utah      1.549430
Ohio      1.777915
Texas     0.196518
Oregon    2.132535
dtype: float64
```

	b	d	e
Utah	1.549430	-0.103965	-1.836008
Ohio	1.777915	0.681121	0.855245
Texas	0.196518	-1.066822	0.181010
Oregon	2.132535	-0.420770	-0.554330

Multi Index

DataFrame.set_index() nos permitía indicar una columna para que fuese la etiqueta de las filas, “setear un índice”.

Pero es posible indicar más de una columna, obteniendo índices de varios niveles. Si seleccionamos una columna que no es única en sus valores debemos tenerlo en cuenta, ya que tendremos un índice no único como filas. Lo normal es ordenar el DataFrame por el índice.

DataFrame original

	Ciudad	Año	Ventas	Gastos
0	Madrid	2023	100	70
1	Madrid	2024	120	80
2	Barcelona	2023	90	60
3	Barcelona	2024	110	75

```
# Agrupamos datos primero por Ciudad y luego por Año
df_multi = df.set_index(['Ciudad', 'Año'])
df_multi.head()
```

		Ventas	Gastos
Ciudad	**Año**		
Madrid	**2023**	100	70
	2024	120	80
Barcelona	**2023**	90	60
	2024	110	75

Indexación multi index

En estos casos debemos indexar usando las etiquetas de filas mediante una tupla con los elementos en orden. (etiqueta 1º nivel, etiqueta 2º nivel…).

		Ventas	Gastos
Ciudad	**Año**		
Madrid	**2023**	100	70
	2024	120	80
Barcelona	**2023**	90	60
	2024	110	75

```
# Indexación de filas concretas usando una tupla de índices
# Madrid 2023
df_multi.loc[('Madrid', 2023)]
```

```
Ventas    100
Gastos     70
Name: (Madrid, 2023), dtype: int64
```

Indexación multi index

No es necesario que indiquemos todos los niveles, pero debemos empezar por el primero y seguir en orden.

Ciudad	Año	Ventas	Gastos
Madrid	2023	100	70
	2024	120	80
Barcelona	2023	90	60
	2024	110	75

```
# Indexación en orden de jerarquía
# Todos los datos de madrid
df_multi.loc['Madrid']
```

Año	Ventas	Gastos
2023	100	70
2024	120	80

DataFrame (importación/exportación CSV)

CSV

read_csv(fichero, sep=separador, header=n, index_col=m, na_values=no-validos, decimal=separador-decimal)

- sep: carácter separador del csv.
- header: Fila del fichero que contiene nombre de las columnas para el DataFrame
- index_col: Columna del fichero que servirá para los nombres de las filas para DataFrame. Si no se indica será de 0 a N-1
- na_values: Lista con valores que serán convertidos a NaN en el DataFrame
- decimal: separador de parte decimal en números reales.

df.to_csv(fichero.csv, sep=separador, columns=booleano, index=booleano)

- sep: separador que se aplicará a los campos en el csv
- columns: True/False para exportar el nombre de las columnas.
- index: True/False para exportar los nombres de las filas como una columna más del csv.

DataFrame (importación/exportación excel)

EXCEL

read_excel(fich.xlsx, sheet_name=hoja, header=n, index_col=m, na_values=no-validos, decimal=separador-decimal)

df.to_excel(fichero.xlsx, sheet_name = hoja, columns=booleano, index=booleano)