Módulo 3: Transformación de Datos

Librerías:
import numpy as np

Arrays: un array o **matriz** es una estructura de datos que permite almacenar un conjunto de elementos del mismo tipo.
Los elementos están dispuestos en una secuencia ordenada y **se accede a ellos mediante un índice**, comienza en 0.

Métodos de arrays
.shape: devuelve una tupla que representa la forma (dimensión), y el **tamaño de una dimensión**. Si es **tridimensional dará 3 valores**: número de arrays pequeños, número de filas, número de columnas.
.ndim: número de índices para acceder a un elemento del array. Unidimensional 1 (filas), bidimensional 2 (filas, columnas), multidimensional 3 (arrays, filas, columnas).
.size: número total de elementos del array.
.dtype: **tipo** de datos

CREACIÓN:

np.array(lista): a partir de una lista o de una lista de listas.
np.empty((shape), dtype=float): un array vacío especificando la forma. **Por defecto es float**, no hace falta especificar type.
En shape bidimensional se especifican 2 valores: filas y columnas.
En shape tridimensional se especifican 3 valores: dimensiones, filas y columnas.
np.zeros(shape): *un array de ceros.*
En shape tridimensional se especifican 3 valores: dimensiones, filas y columnas.
np.ones(shape): un array de unos. En shape tridimensional se especifican 3 valores: dimensiones, filas y columnas.
np.arange(start, stop, step): un array con valores secuenciales. *Start opcional por defecto 0.* **Stop excluye el valor**. *Step opcional por defecto es 1.*

MÓDULO RANDOM Para simular datos aleatorios y realizar experimentos numéricos.

np.random.randint(low, high, (size)): low y high como start y stop. Size es la forma, por defecto es unidimensional. Dtype por defecto int.
array_random = np.random.randint(0, 50, (2,3)) números aleatorios entre el 0 y el 50 (no incluido) de 2 filas y 3 columnas.
np.random.rand(shape): números aleatorios en el rango [0, 1), 0 incluido y 1 excluido.
np.random.sample: Genera números aleatorios en un *array* en el rango [0, 1).

INDEXACIÓN: para acceder a un elemento se accede por su índice, al igual que en las listas.

Array unidimensional:
array(i) → i elemento
array[0] accede al primer elemento
array[-3] accede a los 3 últimos
array[2:5] accede a los elementos del 3 al 5
array[:11:2] accede a los elementos del primero al onceavo incluido saltando de 2 en 2.

Array bidimensional:
array(i, j) → i filas, j columnas. Lo que va a la izquierda de la , son filas a la derecha columnas. Funciona como start, stop, step.
array[0] accede a la primera fila
array[0][0] accede al elemento de la primera fila, primera columna
array[:2,:] accede a las dos primeras filas, todas las columnas
array[:,-3:] accede a todas las filas, tres últimas columnas

Array tridimensional:
array(i, j, k) → i array, j fila, k columna. Lo que va a la izquierda de la , son filas a la derecha columnas.
array[0, 1, 3] accede al primer array, fila 2, columna 4
array[0, 0, :] accede al primer array, primera fila, todas las columnas

FILTRADO
Paso 1: creamos una máscara booleana mask = array < 0.6
Paso 2: aplicamos la máscara array[mask] o array[array < 0.6] devuelve array unidimensional solo con los valores por debajo de 0.6
*Condiciones: **&** es como **and**, **|** es como **or**.
array [(array < 0.2) | (array >0.7)] devuelve los resultados menores de 0.2 o los mayores de 0.7.

Filtrado con np.where()

np.where(condición)
array = np.where(array > 0.8)
Devuelve un resultado con dos tuplas, la primera es el índice de las filas la segunda el índice de las columnas. Hay que combinar ambas tuplas.

np.where(condición, valor_si_verdadero, valor_si_falso)
array = np.where(array > 0.8, 'xxx', 'ooo')
Devuelve xxx cuando la condición es verdadera y ooo cuando es falsa.
np.where(array > 50 array + 500, array)
Devuelve el número+500 cuando es mayor de 50 y el número original cuando es menor de 50.

OPERACIONES ARITMÉTICAS

Suma elementos = np.sum() axis 0c/1f suma todos los elementos entre sí
np.add(): suma
np.substract(): resta
np.multiply(): multiplicación
np.divide(): división
np.power(): potencia
np.round(array, 2): redondea a 2 decimales los elementos del array
+, -, *, / también funcionan pero es más correcto los métodos NumPy
Escalar: array * 2 multiplicar cada elemento del array * 2
suma = np.add(array1, array2)
Suma los elementos del array1 con el correspondiente del array2.

Mínimo = np.min() el valor mínimo
Mínimo_columna = np.min(array, axis = 0)
Mínimo_fila = np.min(array, axis = 1)
Máximo = np.max() el valor máximo
Máximo_columna = np. max (array, axis = 0)
Máximo_fila = np. max (array, axis = 1)

FUNCIONES ESTADÍSTICAS

Media = np.mean() media de todos los elementos del array
media_columna = np.mean(array, axis = 0) axis 0 por columnas
media_fila = np.mean(array, axis = 1) axis 1 por filas

Varianza = np.var()
Indica cómo de dispersos están los *valores alrededor de su media*. Una varianza alta significa que los valores están dispersos, y baja que los valores están cercanos a la media. *Expresada en unidades al cuadrado.*
axis 0/1 por columnas/filas.

Desviación = np.std()
Indica cuánto varían los valores con respecto a la media. Una desviación alta significa que los valores están dispersos, y baja que los valores están cercanos a la media. *Expresada en las mismas unidades.*
axis 0/1 por columnas/filas.

OTROS MÉTODOS

np.sort(): ordena de menor a mayor por filas los elementos. Si queremos que ordene por columnas axis = 0.
Para **ordenar de mayor a menor** ponerle símbolo menos al método y al array → *ordenar = -np.sort(-array)*

np.transpose(array): bidimensional. Devuelve un nuevo array con las dimensiones intercambiadas, cambia las filas por las columnas.

np.transpose(arrays, filas, columnas) multidimensional. Devuelve un nuevo array con el número de dimensiones, filas y columnas de cada dimensión indicadas.

np.shape(array, (filas, columnas)): cambia la forma pero no los datos del array, cambia las filas por las columnas

np.swapaxes(array, axis1, axis2): intercambia los ejes especificados en axis1 y axis2.
.copy(): crea una copia en una nueva variable.
.flatten(): convierte un array multidimensional en uno unidimensional.

INTRO A PANDAS

Librerías:
```
import pandas as pd
```

SERIES:

Estructura de datos unidimensional.Contiene **datos de un solo tipo.**
Cada elemento en una Serie tiene una etiqueta de índice asociada.
Los índices pueden ser etiquetas personalizadas o valores numéricos generados automáticamente.

Creación de Series:

Serie vacía: serie_vacia = **pd.Series()**
Serie a partir de lista: serie = **pd.Series(lista)**
Serie a partir de diccionario: serie = **pd.Series(diccionario)**
Serie con índice personalizado: serie = **pd.Series(lista, index = [‘a’, ‘b’, ‘c’, ‘d’])**

Propiedades de las series:
serie.values: devuelve los valores
serie.index: devuelve los índices
serie.dtype: tipo de datos int64, float64, object, datetime64
serie.size: número de elementos de la serie
serie.shape: forma de la serie → (n,) n es el número de elementos

Indexación en las series:
Por posición: serie[0] accede al primer elemento
Por la etiqueta del índice: serie[‘etiqueta’]
Por rango: serie [0:3] star:stop, devuelve del primer al tercer elemento
Por lista de índices: serie [[0, 2, 3]] devuelve el primer, tercer y cuarto elemento

DATAFRAMES:
Estructura de datos bidimensional.Pueden contener diferentes tipos de datos.
Creación de DataFrames:
A partir de un diccionario de listas: cada key es una columna y cada elemento de su lista de values es un valor de esa columna. df = pd.DataFrame(diccionario)
A partir de una lista de diccionarios: las keys son las columnas, los values los elementos de esa columna. df = pd.DataFrame(lista_diccs)

Apertura de ficheros:
CSV:
pd.read_csv("../ruta/nombre_archivo.csv", sep = “;”, delimiter=None, header='infer', names=None, index_col=0, dtype=None)
• sep: si no devuelve DF especificar que están separadas por ;
• delimiter: lo mismo que sep
• header: encabezado de columna. None: sin nombres de columna, infer: nombres de columna del archivo, Número: indica el número de fila que será columna header=0.
• names: nombres de las columnas
• index_col: crear una columna índice
• dtype: para especificar el tipo de datos

Otros parámetros.
EXCEL:
pd.read_excel("ruta/archivo.xlsx", sheet_name=0, header=0, names=None, index_col=None, dtype=None)
• sheet_name: hoja del Excel que quieres leer. Leer varias hojas por su nombre =['Sheet1', 'Sheet2'].
• header: None: sin nombres de columna, 0: número de fila que será columna
• names: nombres de las columnas.
• index_col: qué columna será el índice. Un integer, nombre de columna o lista de columnas,
• dtype: para especificar el tipo de datos

JSON:
df = pd.read_json("nombre_archivo.json", orient=None, typ='frame', dtype=True, convert_axes=True, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding=None, lines=False)

• orient: colums: por defecto, formato columna. Index: formato índice. Records: formato de registros. Split: formato dividido.. values: valores, sin etiqueta de columna o ídice.
• Typ: frame: por defecto DataFrame. Series: crear un objeto Series
• dtype: para especificar el tipo de datos
• convert_axes: True indica si las etiquetas de los ejes deben convertirse en

índices o nombres de columna.
• convert_dates: True indica si se deben convertir las cadenas de fecha y hora en objetos de fecha y hora.
• keep_default_dates: True mantener las fechas predeterminadas.
• numpy: indica si los datos deben devolverse como una matriz NumPy en lugar de un objeto DataFrame. Por defecto, es False.
• precise_float: indica si se deben utilizar números de punto float precisos en lugar de valores de punto float nativos de Python. Por defecto, es False.
• date_unit: especifica la unidad de fecha y hora si se deben convertir las cadenas de fecha y hora. Puede ser 's' para segundos o 'ms' para milisegundos.
• encoding: permite especificar la codificación del archivo JSON si no se puede inferir automáticamente.
• lines: indica si el archivo JSON contiene múltiples objetos JSON en líneas separadas en lugar de un solo objeto JSON.

PICKLE
Archivo binario que se usa para serializar y deserializar objetos. Cuando guardas objetos en un archivo pickle, se puede almacenar o enviar.
Pkl = pd.read_pickle("ruta/archivo.pkl", compression='infer')
• compression: por defecto infer, la biblioteca intentará inferir automáticamente el tipo de compresión.
Puedes especificar un tipo de compresión explícitamente, como 'gzip' o 'bz2'

INDEXACIÓN:
LOC, ILOC
Métodos para acceder y manipular los datos en un DataFrame.

LOC: df.loc[filas, columnas]

Se puede indicar una etiqueta o una lista de etiquetas. Ej: **df.loc['Tues', 'Humidity']**
Filas por nombre del index, columna por su nombre
Para ver todos los valores de filas o columnas sustituir por : → **df.loc['Tues', :]**
Para dar una lista de valores de filas o columnas → **df.loc[['Mond', 'Tues'], :]**
*También se puede usar **start:stop:step**

ILOC: df.iloc[filas, columnas]

Para acceder utilizando integers de fila o columna, empiezan en 0. Ej: **df.iloc[1, 3]**
Para ver todos los valores de filas o columnas sustituir por : → **df.iloc[1, :]**
Para dar una lista de valores de filas o columnas → **df.iloc[[1, 2, 3], 2]**
*También se puede usar **start:stop:step**

POR CONDICIÓN
Se puede indicar el nombre de la columna **df['nombre_columna']** o lista columnas **df['columna1', 'columna2']**
Especificando una condición a la columna para que solo devuelva los valores que cumplen esa condición.

df1 = df.loc[df.Temperatura < 10, :] filtra para la columna Temperatura solo los valores menor de 10 y todas las demás columnas pero solo da las filas que coincidan que temperatura es menor de 10.

Con iloc se sustituye el nombre de la columna por su índice pero es necesario que sea en formato lista df.iloc[list(df[1] < 10), :]

Varias condiciones

LOC

```
df_dos_condiciones_loc = df.loc[(df.Wind > 20) & (df.Weather == 'Sunny'),
['Temperature', 'Wind']]
```

ILOC

```
df_dos_condiciones_iloc = df.iloc[list((df.Wind > 20) & (df.Weather == 'Sunny')),
[1,2]]
```

CREAR COLUMNAS

Asignación directa:

df1 = df[‘nueva_columna’] = (df[‘columa_operacion’] * 12) Devuelve una nueva_columna cuyos valores son los de una columna ya existente *12

Método .assign()

df1 = df.assign(nueva_columna=df[‘columa_operacion’] * 12)

Método .insert()

df1 = df.insert(loc, column, value, allow_duplicates=False)

Ej: df.insert(0, "indice", range(1,8))

Inserta la columna índice en la posición 0, con un rango de números del 1 al 7.

EDA

Librerías:

import pandas as pd

MÉTODOS:

pd.set_option('display.max_columns', None): visualizar el DataFrame con todas sus columnas

df.head(): muestra 5 primeras filas, o el número indicado entre los paréntesis

df.tail(): muestra 5 últimas filas

df.sample(6): muestra 6 de forma aleatoria

df.info(): información del DataFrame

df.duplicated().sum(): indica los valores duplicados, filas con la misma info

df.describe().T: aporta **datos estadísticos de la columnas numéricas** como la media, la desviación estándar, los valores mínimo y máximo, los percentiles y más:

- count: número de valores no nulos
- mean: media
- std: desviación estándar, dispersión de los datos
- min: valor mínimo
- 25%: valor por debajo del que se encuentran el 25% del valor de la columna
- 50%: mediana, divide al conjunto de daos en dos mitades iguales
- 75%: valor por debajo del que se encuentran el 75% del valor de la columna
- max: valor máximo

df.describe(include = "object").T: aporta **datos** sobre las **columnas categóricas**:

- count: número de valores no nulos
- unique: cantidad valores únicos
- top: valor más común de la columna
- freq: frecuencia del valor más común

df.shape[0]: número de **filas**

df.shape[1]: número de **columnas**

df.columns: nombres de las columnas

df[‘columna’]: saca **todos los datos** de la columna

df[‘columna’].unique(): saca los **datos únicos de la columna**

df[‘columna’].value_counts(): **total de cada valor único** de la columna

df[‘columna’].info(): **información** de la **columna**

df.columns.get_loc("columna"): **posición** de la **columna**

df.select_dtypes(include=None, exclude=None): **seleccionar columnas** de un **DataFrame por su tipo de datos**
• include: tipos de datos a incluir
• exclude: tipos de datos a excluir
• *Son opcionales*
• **Tipos de datos:** 'int', 'float', 'object'

df.drop(labels, axis=0, inplace=False): para eliminar columnas de un DataFrame
• labels: nombres de las filas o columnas a eliminar, un valor o lista
• axis: 0 filas, 1 columnas
• inplace: True eliminación en el DataFrame original, False nuevo DataFrame con los cambios. Opcional.

Datos nulos:
df.isnull(): devuelve una serie booleana con datos nulos
df.isnull().sum(): número total de datos nulos en el DataFrame
df['columna'].isnull()
df['columna'].isnull().sum()
.isna(): igual que isnull
.notnull(): muestra los datos no nulos

Valores duplicados:
df.duplicated(): devuelve una serie booleana con datos duplicados
df.duplicated().sum(): número total de datos duplicados en el DataFrame
df.duplicated(subset = "columna").sum(): número de valores duplicados en una columna

UNION

Librerías:

```
import pandas as pd
import numpy as np
pd.set_option('display.max_columns', None)
```

CONCAT:
Une dos o más DataFrame de forma horizontal (por los mismos índices, donde no coincidan rellena filas con NaN) o vertical (por defecto)
df_concat = pd.concat(objs, axis=0, join='outer', ignore_index=False)
• objs: lista de DataFrames a unir
• axis: 0 por filas, en vertical. 1 por columnas, en horizontal
• ignore_index: True reestablece el índice

MERGE:
Combina dos DataFrame por una o más columnas comunes
df_merge = df_left.merge(df_right, how='inner'/'left' on=None/left_on=None, right_on=None)
• df_left: primer DataFrame
• df_right: segundo DataFrame
• how: (op)métodos de unión: por defecto inner
  -inner: filas comunes por las columnas de unión. Como inner join en MySQL.
  - left: se conservan todas las filas del DataFrame de la izquierda. Left join
• on: si las columnas de unión se llaman igual
• left_on: columna de unión del primer DataFrame
• right_on: columna de unión del segundo DataFrame

JOIN:

Combina dos DataFrame por las etiquetas de índice. El primer DataFrame tiene el índice y el segundo una columna igual que el índice del primero.

df_join = df_left.join(df_right, on='nombre', how='tipo_de_join', lsuffix='', rsuffix='')

- df_left: DataFrame del que coge índice
- df_right: DF con el que unir, usando columna igual que el índice del primero
- on: columna e índice comunes
- lsuffix: un alias para las columnas del DF de la izquierda
- rsuffix: un alias para las columnas del DF de la derecha

LIMPIEZA

df = df.rename(colums= {'key': 'value'}, index= {'key': 'value'}, inplace=False)

Renombrar columnas

- colums: key nombre actual, value nuevo nombre
- index: (opcional) etiqueta del índice, key actual, value nuevo
- inplace: (op) False nuevo DataFrame por defecto, no poner si se crea nueva variable. True sobreescribe

Modificar valores de una columna reemplazando símbolos o espacios:

df['columna'] = df['columna'].str.replace('.', ' ')

Modifica en todos los valores de la columna sustituyendo el . por espacio.

Dict comprehension para unificar nombres de todas las columnas del DataFrame:

1. nuevas_columnas = {columna: columna.lower().replace(".", "") for columna in df.columns}
2. df.rename(columns = nuevas_columnas, inplace = True)

Modifica todas las columnas poniendo sus nombres en minúsculas y sustituye el . por nada, para que el nombre de la columna vaya todo junto.

dataframe.set_index(keys, drop=True, inplace=False)

Establecer una columna o columnas como índice del DataFrame

- keys: columna o lista de columnas que serán índice
- drop: (op) True por defecto, las columnas utilizadas como índice se eliminarán de DataFrame True sobreescribe
- inplace: (op) False nuevo DataFrame por defecto, no poner si se crea nueva variable. True sobreescribe

Modificar valores de una columna:

Sintaxis:

df ["columna"].str.lower()

Métodos:

.str.lower(): minúsculas
.str.upper(): mayúsculas
.str.capitalize(): primera letra mayúscula
.str.strip(): elimina espacios en blanco del principio y final
.str.split("-"): output con una lista por fila con sus elementos separados donde el -

Para sobreescribir con los métodos:

df[["new_columna1", "new_columna2"]] = df["columna_modificar"].str.split("-", expand=True).get([1, 2]):

- new_columna: columnas que queremos crear
- columna_modificar: columna de la que crear las nuevas
- "-": elemento donde hacer la separación
- expand=True: sobreescribe el DataFrame
- get[1,2]: no de índice de los elementos de la columna actual con los que crear las nuevas

Cambiar tipo de valor de una columna.
Modificarlo a tipo fecha:
df[‘columna’] = pd.to_datetime(df[‘columna’])

FILTRADO

Operadores de comparación: >, <, >=, <=, ==, !=
Crear una condición y aplicarla a columnas.
Se pueden aplicar **diferentes condiciones con & o |.**
1. Crear condición: condición = df[‘columna’] == ‘x’
2. Aplicar condición df_nuevo = df[condición]
3. df_nuevo es igual pero con la columna de la condición modificada

isin(): seleccionar filas que contienen valores específicos en una columna.
Un valor o lista de valores
df[‘columna’].isin(valores)
1. Crear filtro: filtro = [‘valor1’, ‘valor2’]
2. Aplicar filtro df_nuevo = df[df[‘columna’].isin(filtro)]
3. df_nuevo es igual pero con las filas que contienen los valores del filtro

between(): filtrar por un rango
nuevo_df = df[df[‘columna’].between(inicio, fin, inclusive=both/left/right/neither)]
- both: incluye los valores de inicio y fin
- left: incluye inicio pero no fin
- right: incluye fin pero no inicio
- neither: no incluye ni inicio ni fin

Para **filtrar por rango de fechas:**
1. variables con la **fechas inicio** = **pd.to_datetime('2013-01-01') fin = pd.to_datetime('2013-01-31')**
2. filtrar con **between df_nuevo = df[df["columna"].between(inicio, fin, inclusive = "both")]**

str.contains(): filtrar por palabras. Devuelve un booleano.
df[‘columna’].str.contains(pat, case=True, na=nan, regex=True)
- pat: patrón de texto a buscar
- case: (op) True distingue mayúsculas y minúsculas
- na=nan: (op)
- regex: (op) True se interpreta como regex

GROUP BY
Sintaxis:
Para una operación
variable = df.groupby(columna)[columna_operacion].operacion
Para varias operaciones
variable = df.groupby(columna)[columna_operacion].agg([‘op1’, ‘op2’])

Operaciones de agregación:
.count(): número valores no nulos
.describe(): resumen de los principales estadísticos
.sum(): suma de todos los valores
.mean(): media de los valores
.median(): mediana. Ordenados de menor a mayor, valor que queda en la mitad
.min(): valor mínimo
.max(): valor máximo
.std(): desviación estándar. Cuánto se desvían los valores de la media.
.var(): varianza. Cuánto se desvían los valores de la media al cuadrado.

Métodos:
.reset_index(): nuevo DataFrame del resultado con índice en 0
df.groupby(columna)[columna_operacion].operacion(numeric_only=True): aplica la operación a todas columnas numéricas
variable_agrupacion.ngroups: grupos formados después de la agrupación

APPLY

Para aplicar una función a una columna. Se puede añadir a una nueva columna de o aplica a una columna ya existente para modificar el DataFrame

Sintaxis:
df['nueva_columna'] = df['columna'].apply(función)
df['columna_modificar'] = df['columna_modificar'].apply(función)
Para aplicar una función que debe recibir dos parámetros (dos columnas del DF) se hace con lambda.

Sintaxis:
df['nueva_columna'] = df.apply(lambda x: función(x['col1'], x['col2']), axis=1)

OTROS MÉTODOS DE LIMPIEZA

Para aplicar una transformación o reemplazo de los valores a un Serie o DataFrame
.map(): transformación de cada elemento de una Serie
Pasos:
1. Crear diccionario de mapeo, keys valores actuales, values valores por los que reemplazar
diccionario = {0: "No", 1: "Si"}
2. Aplicar diccionario a una columna:
df['col'] = df['col'].map(diccionario)

.replace(): reemplaza valores en un DataFrame o Serie por otros especificados
Sintaxis:
df['col'] = df['col'].replace(valor a reemplazar, nuevo valor)
Se utiliza para reemplazar un valor concreto de esa columna, no todos

GESTIÓN VALORES NULOS

Librerías:

```
import pandas as pd
import numpy as np

from sklearn.impute import SimpleImputer
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.impute import KNNImputer

import seaborn as sns
import matplotlib.pyplot as plt
pd.set_option('display.max_columns', None)
```

Tipos nulos y cómo sustituir, teniendo en cuenta el contexto:
Columnas categóricas:
• Si el % de nulos es pequeño y tenemos una moda representativa, cambiamos por la moda
• En caso contrario, categorizar como "unknown"
Columnas numéricas:
• La mediana es más robusta que la media

Eliminar filas con valores nulos en una columna:
`df.dropna(subset=['columna'], inplace=True)`
Subrayar filas con valores nulos:
`df.style.highlight_null(color='yellow')`

PASOS GESTIÓN DE NULOS:
1. Conocer el porcentaje de nulos por columna del DF
```
porc_nulos = (df.isnull().sum() / df.shape[0]) * 100
```
Visualizarlo en un DF solo con las columnas con nulos:
```
df_nulos = pd.DataFrame(porc_nulos, columns = ["%_nulos"])
df_nulos[df_nulos["%_nulos"] > 0]
```

2. Extraer columnas con valores nulos:
Categóricas:
```
nulos = df[df.columns[df.isnull().any()]].select_dtypes(include = "O").columns
```
Numéricas: `include = np.number`

3. Conocer la distribución de categorías de las columnas con nulos
```
for col in nulos:
display(df[col].value_counts() / df.shape[0] * 100)
```
Saca el porcentaje de cada valor único en las columnas con nulos del paso2

4. Reemplazarlos valores nulos: si hay un dato dominante, podremos sustituir los nulos por la moda, si no por 'unknown'.

Métodos:
`.fillna()`: rellena los valores nulos con uno específico. Sintaxis:
Modificar por moda:
```
moda = df['columna'].mode()[0]
df['columna] = df['columna'].fillna(moda)
```
Modificar por valor predeterminado:
```
df['columna] = df['columna'].fillna('unknown')
```

SimpleImputer: igual que fillna. **Permite imputar un mismo valor, media, mediana o moda o valor constante, a todos los nulos de una columna.**
Sintaxis:
1. Imputar con la media `imputer = SimpleImputer(strategy='mean')`
• mean, median, most_frequent, constant
2. Transformar los datos
```
df['columna'] = imputer.fit_transform(df[['columna']])
```

IterativeImputer: hace una *predicción del valor nulo basándose en los datos de toda la tabla.*

Sintaxis:

1. Crear una instancia del IterativeImputer;

```
imputer = IterativeImputer(max_iter 10, random_state 42)
```

• max_iter: no de iteraciones que usará para completar los datos nulos
• random_state: semilla para el generador de números aleatorios

2. Ajustar y transformar los datos

```
data_imputed = imputer.fit_transform(df[[‘col1’, ‘col2’, ‘col3’,]])
```

• datos: columna(s) que contienen los valores faltantes

3. Sustituir las columnas modificadas si es más de una. *Si es una solo hacer el paso2, cambiando data imputed por columna a modificar.*

```
df[[‘col1’, ‘col2’, ‘col3’,]] = data_imputed
```

KNNImputer: *hace una predicción para los valores nulos numéricos basándose en los valores vecinos, columnas que le pasamos que podría usar.*

1. **Crear una instancia del IterativeImputer,** recomendable vecinos 3-5

```
imputer = KNNImputer(n_neighbors = 3-5)
```

2. **Ajustar y transformar los datos**

```
knn_imputed = imputer_knn.fit_transform(df[[‘col1’, ‘col2’]])
```

• col1, col2: columnas con la que completa los nulos

3. **Añadir nuevas columnas al DataFrame y comprobar con .describe() que son datos realistas**

```
df[[‘col1’, ‘col2’]])= knn_imputed
```

4. **Eliminar columnas que están repetidas**