Introducción a la librería *pandas*

Autor: Rubén Sierra Serrano
Universidad Francisco de Vitoria

Tabla de Contenidos

Antes de realizar cualquier tipo de actividad sobre un *dataset*, es recomendable llevar a cabo un análisis y una limpieza de datos previo para comprender aspectos como el tipo de variables que representan los atributos (discretas, continuas), el tipo de datos que contienen (**entero**, **flotante**, **cadena**, etc.), el comportamiento estadístico de las variables (distribuciones que siguen, correlaciones con otras variables del *dataset*, la presencia de *outliers*, entre otros factores) y la existencia de valores faltantes (`NaNs`).

Python, al ser el lenguaje predilecto para *Machine Learning* y *Data Analysis* debido a sus librerías especializadas (como *TensorFlow*, *PyTorch* o *scikit-learn*), hace que resulte útil emplear otra librería de Python, como *pandas* (en minúsculas), para el análisis mencionado previamente. Puede visitar la página web de *pandas* haciendo clic aquí. Para importar *pandas*:

In [179…
```
import pandas as pd
```

Cabe resaltar que *pandas* está construida sobre otra librería llamada *NumPy*, ya que trabaja con datos tabulares y series temporales. Nótese que un dato tabular no es más que un array con filas y columnas, pero con la salvedad de que emplea etiquetas para identificar las filas y las columnas (además de los índices posicionales).

Otra diferencia importante con respecto a *NumPy* es que las columnas dentro de un mismo array pueden tener distintos tipos de datos, mientras que en *NumPy* deben ser homogéneas.

pandas tiene tres tipos de objetos principales:

- **Serie**: en esencia, es un array unidimensional.
- **DataFrame**: en esencia, es un array bidimensional que resulta en un dato tabular. Es una colección de **Series**.
- **Index**: se emplea para indexar los dos tipos de objetos previos, permitiendo el acceso a cada dato individual. Esto puede hacerse implícitamente por posición o explícitamente mediante las etiquetas asignadas.

Antes de continuar, resaltar que se esta empleando la versión 2.2.3 de *pandas* en esta introducción, en caso de que algún comportamiento anómalo no descrito en la guía ocurra, consulte la documentación oficial de *pandas* para localizar el error y entender su comportamiento. Para comprobar la versión que esta empleando:

In [180…
```
pd.__version__
```

Out[180…
```
'2.2.3'
```

Index

Los tipos de datos secuenciales como los **listas**, las **tuplas** o los **arrays** de *NumPy* tienen un orden natural, por tanto, sus elementes tienen una posición dentro de dichos datos, esto se conoce como el índice implícito de la secuencia

In [181…
```
l = ["a", "b", "c", "d"]
#     0    1    2    3

print(l[0])
print(l[0:4])
```

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

en pandas, como se ha mencionado previamente, podemos declarar un índice explícito a través de una etiqueta además del índice implícito.

El tipo de datos para los índices de *pandas* es `pd.Index`. Este es el tipo de índice más genérico (existen dentro de *pandas* otros más específicos, como **RangeIndex** o **CategoricalIndex**). Son tipos de datos que contienen elementos y están basados en los **arrays** de *NumPy*, por lo tanto, ellos mismos tienen un índice posicional implícito.

Podemos crear un objeto **Index** pasando un objeto secuencial unidimensional (como las **listas**, las **tuplas**, los **arrays** de *NumPy* o las **Series** de *Pandas*) al constructor `pd.Index` de la siguiente forma

In [182…
```
idx = pd.Index((10, 20, 30, 40))
```

una vez declarado el objeto podemos acceder a sus elementos de manera posicional

In [183…
```
idx[0]
```

Out[183…
```
np.int64(10)
```

que devuelve el elemento con el tipo de dato que tuviera previamente, en este caso, el número 10 en forma de **entero** de *NumPy* representado con 64 bits.

También podemos acceder de manera simultánea a varios elementos del objeto mediante técnicas como *slicing*, *fancy indexing* o una máscara booleana

In [184…
```
idx[1:4]
```

Out[184…
```
Index([20, 30, 40], dtype='int64')
```

In [185…
```
idx[[0,1]]
```

Out[185…
```
Index([10, 20], dtype='int64')
```

In [186…
```
idx[idx % 4 == 0]
```

Out[186…
```
Index([20, 40], dtype='int64')
```

que devuelven otro objeto **Index** con los elementos seleccionados en el tipo de datos que tenían de manera original.

Los **Index** son objetos inmutables, es decir, una vez que se han creado no se pueden modificar, en contraposición, si tratamos de modificar un **Index** saltará un **TypeError**:

In [187…
```
idx[1] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[187], line 1
----> 1 idx[1] = 100

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\indexes\base.py:5371, in Index.__setitem__(self, key, value)
   5369 @final
   5370 def __setitem__(self, key, value) -> None:
-> 5371     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Para demostrar lo anterior, emplearemos la función `id()` que devuelve un **id** único que se otorga a cada objeto una vez que es creado

In [188…
```
id(idx)
```

Out[188…
```
2574523287408
```

In [189…
```
idx = pd.Index((10, 100, 30, 40))
idx
```

Out[189…
```
Index([10, 100, 30, 40], dtype='int64')
```

In [190…
```
id(idx)
```

Out[190…
```
2574804337440
```

Podemos comprobar que los **id** no coinciden, indicando que, en efecto, son objetos diferentes en memoria.

Como se ha mencionado previamente, también existen otros índices más específicos. Estos índices especializados son clases hija de **Index**

Subtipo	Descripción
RangeIndex	Secuencias de enteros generados por `range`
CategoricalIndex	Índices de valores categóricos
IntervalIndex	Representar y trabajar con intervalos (ej: Rangos Numéricos)
DatetimeIndex	Índices que contengan fechas (representadas internamente en **int64**)
⋮	⋮

Nota Importante:

En la versión 1.4.0 de *pandas*, se deprecan los subtipos numéricos de **Index**, como **Int64Index** y **Float64Index**.
Esto significa que, para representar índices con valores numéricos, se utilizará **NumericIndex**, variando únicamente el tipo de dato de sus elementos (`dtype`). Aunque los subtipos aún pueden usarse en las versiones 1.x de *pandas*, a partir de la versión 2.0 ya no estarán disponibles.

Operaciones de conjunto en índices

Se le pueden aplicar operaciones de conjuntos a los índices, es decir, poseen comportamiento similar a los conjuntos matemáticos:

- `.intersection()` para la intersección (∩)
- `.union()` para la unión (∪)
- `in` para pertenece a (∈)

Esto es especialmente útil a la hora de juntar **DataFrames**.

In [191…
```
idx1 = pd.Index([10, 20, 30, 40])
idx2 = pd.Index([10, 50, 60, 70])
```

In [192…
```
idx1.intersection(idx2)
```

Out[192…
```
Index([10], dtype='int64')
```

In [193…
```
idx1.union(idx2)
```

Out[193…
```
Index([10, 20, 30, 40, 50, 60, 70], dtype='int64')
```

In [194…
```
print(10 in idx1)
```

```
True
```

Al hacer `.intersection()` y `.union()`, pandas tratará de emplear el tipo de dato más general para determinar el tipo de dato del índice generado:

In [195…
```
idx_float = pd.Index([10.0, 100.0, 200.0])
```

In [196…
```
idx1.union(idx_float)
```

Out[196…
```
Index([10.0, 20.0, 30.0, 40.0, 100.0, 200.0], dtype='float64')
```

In [197…
```
idx1.intersection(idx_float)
```

Out[197…
```
Index([10.0], dtype='float64')
```

In [198…
```
idx_string = pd.Index(["a", "b", "c"])
```

In [199…
```
idx1.union(idx_string)
```

Out[199…
```
Index([10, 20, 30, 40, 'a', 'b', 'c'], dtype='object')
```

Índices de cadenas, enteros y flotantes

Cuando se crea un índice a partir de una cadena (string) en Pandas, el tipo de dato resultante de los elementos del índice es `object`. Este tipo de dato es fundamental en Python y sirve como base para todos los demás, actuando como un tipo "universal" debido a la naturaleza orientada a objetos del lenguaje. Su versatilidad permite la creación de estructuras de datos tabulares heterogéneas en Pandas, ya que todas las clases (tanto las nativas de Python como las creadas por el usuario) heredan de este tipo de dato.

Sin embargo, el uso de `object` no es eficiente al aplicar funciones universales, ya que carece de un equivalente en C (el lenguaje subyacente de Python). Esto impide que aproveche las optimizaciones a bajo nivel que sí tienen otros tipos de datos.

In [200…
```
string_index = pd.Index(["a", "b", "c"])
string_index
```

Out[200…
```
Index(['a', 'b', 'c'], dtype='object')
```

En contraste, si se crea un índice a partir de enteros o flotantes, los objetos serán `int64` o `float64`, respectivamente

In [201…
```
int_index = pd.Index([10, 20, 30])
int_index
```

Out[201…
```
Index([10, 20, 30], dtype='int64')
```

In [202…
```
float_index = pd.Index([10.0, 20.0, 30.0])
float_index
```

Out[202…
```
Index([10.0, 20.0, 30.0], dtype='float64')
```

Cuando creamos un **Index** en pandas con elementos de distintos tipos de datos, pandas determina automáticamente el tipo de dato más general para representar todos los valores de manera consistente, tratando de escoger siempre el tipo de dato más eficiente posible.

In [203…
```
int_and_float_index = pd.Index([10, 20, 30, 10.0, 20.0, 30.0])
int_and_float_index
```

Out[203…
```
Index([10.0, 20.0, 30.0, 10.0, 20.0, 30.0], dtype='float64')
```

In [204…
```
misc_index = pd.Index(["a", 10, 10.0])
misc_index
```

Out[204…
```
Index(['a', 10, 10.0], dtype='object')
```

Índices `range`

Este tipo de objetos puede ser creado a partir de un objeto `range`. Esto es más eficiente que los generados de manera directa con `pd.Index`, ya que es un tipo de dato conocido como "perezoso". Es decir, el ordenador solo crea el siguiente objeto de la secuencia cuando el programa lo necesita.

Además, en términos de espacio, el objeto `range` solo está determinado por los tres números **`start`**, **`stop`** y **`step`**, mientras que un tipo de dato de secuencia (listas, tuplas, etc.) necesita un puntero para cada elemento dentro de él.

In [205…
```
pd.Index(range(1, 10, 2))
```

Out[205…
```
RangeIndex(start=1, stop=10, step=2)
```

también se puede emplear la clase `pd.RangeIndex` de manera directa

In [206…
```
pd.RangeIndex(start = 1, stop = 10, step = 2)
```

Out[206…
```
RangeIndex(start=1, stop=10, step=2)
```

No unicidad

Los valores de un índice no tienen por qué ser necesariamente únicos. Dado que estos índices funcionan como claves, al consultar qué elemento está asociado a una determinada clave en una **Serie**, se devolverán todos los elementos que la comparten.

In [207…
```
pd.Index([1,1,2,3])
```

Out[207…
```
Index([1, 1, 2, 3], dtype='int64')
```

Por lo tanto, los índices no se utilizan necesariamente para identificar filas únicas en los datos, sino para asignar un valor a cada fila. De este modo, al realizar una consulta, es posible obtener más de un valor asociado.

Esto ocurre en contraposición de los valores posicionales que, por su propia naturaleza, solo devuelven un dato cuando son consultados.

Series

Las **Series** de pandas son un ejemplo de vectores asociativos cuyos elementos tienen una posición definida en la colección (como las **tuplas** o las **listas**), es decir, tienen un orden determinado (a diferencia de los **sets** en Python). Esta posición se conoce como índice implícito.

Además, las Series también cuentan con un índice explícito (similar a las claves de los **diccionarios**), conocido como *labels* o etiquetas. Esto significa que una Serie presenta

dos formas de asociación: una que proviene del orden natural dentro de la secuencia (implícita) y otra que es definida por el programador mediante etiquetas (explícita) y que es opcional.

Para declarar una **Serie**, se utiliza el constructor `pd.Series`, al cual se le pasa una secuencia ordenada de datos (pueden ser **tuplas**, **listas**, **arrays** de *NumPy*, etc., siempre y cuando la estructura mantenga un orden definido). Para definir el índice explícito, se utiliza el parámetro `index`.

Nótese que, si se introduce un **diccionario** como argumento, las **llaves** del diccionario serán utilizadas automáticamente como los índices explícitos de la Serie.

In [208…
```
serie = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
serie
```

Out[208…
```
a    10
b    20
c    30
d    40
dtype: int64
```

In [209…
```
serie_dict = pd.Series({"a": 10, "b": 20, "c": 30, "d": 40})
serie_dict
```

Out[209…
```
a    10
b    20
c    30
d    40
dtype: int64
```

Podemos acceder a los elementos de una **Serie** utilizando ambos tipos de índices: el implícito (basado en la posición) y el explícito (basado en etiquetas). En ambos casos, es posible aplicar técnicas como *slicing* y *fancy indexing*. Además, cuando se utiliza el índice implícito, también se pueden emplear máscaras booleanas para realizar filtrados condicionales.

In [210…
```
serie[0:2]
```

Out[210…
```
a    10
b    20
dtype: int64
```

In [211…
```
serie["a":"b"]
```

Out[211…
```
a    10
b    20
dtype: int64
```

Se puede observar que, al aplicar *slicing*, el índice posicional excluye el último elemento, mientras que el índice explícito lo incluye.

En el caso de emplear *slicing* con el índice explícito también podemos determinar el paso

In [212…
```
serie["a": "d": 2]
```

Out[212…
```
a    10
c    30
dtype: int64
```

En el caso anterior, *pandas* sabe que índice aplicar debido a la diferencia del tipo de dato (**enteros** para el índice posicional y **cadenas** para el índice explícito), sin embargo, en el caso de que el índice explícito también sea definido con **enteros**, en caso de acceder a un elemento individual, empleará el índice explícito pero al hacer *slicing* aplicará el posicional

In [213…
```
serie_index = pd.Series([100, 200, 300, 400, 500], index = [2, 3, 4, 5, 6])
serie_index
```

Out[213…
```
2    100
3    200
4    300
5    400
6    500
dtype: int64
```

In [214…
```
serie_index[2]
```

Out[214…
```
np.int64(100)
```

In [215…
```
serie_index[2:4]
```

Out[215…
```
4    300
5    400
dtype: int64
```

In [216…
```
serie[[0, 1]]
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\2915015515.py:1: FutureWarning:
Series.__getitem__ treating keys as positions is deprecated. In a future version,
integer keys will always be treated as labels (consistent with DataFrame behavio
r). To access a value by position, use `ser.iloc[pos]`
  serie[[0, 1]]
```

Out[216…
```
a    10
b    20
dtype: int64
```

Nota Importante:

En la versión 2.1.0 de pandas, se ha depreciado el uso de enteros dentro de corchetes ([]) para acceder a elementos de una **Serie** cuando se interpretan como posiciones (índice implícito). En versiones futuras, los enteros dentro de [] serán tratados únicamente como etiquetas del índice explícito, y no como posiciones.

In [217…
```
serie[["a", "b"]]
```

Out[217…
```
a    10
b    20
dtype: int64
```

In [218…
```
serie[serie <= 20]
```

Out[218…
```
a    10
b    20
dtype: int64
```

Atributos `loc` e `iloc`

Para evitar cualquier tipo de confusión que pueda surgir del uso del operador de indexación `[]`, se recomienda utilizar los atributos (que no funciones) `.loc[]` e `.iloc[]`, correspondientes a *location* (ubicación por etiqueta) e *implicit location* (ubicación por posición), respectivamente.

Estas herramientas permiten indicar de manera específica e inequívoca cuál de los dos tipos de índice se desea utilizar al acceder a los elementos de una **Serie**:

- `.loc[]` accede a los elementos utilizando el **índice explícito** (etiquetas definidas por el usuario).
- `.iloc[]` accede a los elementos utilizando el **índice implícito** (la posición numérica dentro de la Serie).

Este enfoque elimina ambigüedades, especialmente en casos donde el índice explícito está compuesto por enteros, lo que podría confundirse con las posiciones de los elementos. Por esta razón, el uso de `.loc[]` y `.iloc[]` se considera la forma más clara y recomendada para acceder a los datos en pandas.

Recordemos el caso que podía causar confusión cuando hablabamos de *slicing*:

In [219…
```
serie_index = pd.Series([100, 200, 300, 400, 500], index = [2, 3, 4, 5, 6])
serie_index
```

Out[219…
```
2    100
3    200
4    300
5    400
6    500
dtype: int64
```

In [220…
```
serie_index.loc[2]
```

Out[220…
```
np.int64(100)
```

In [221…
```
serie_index.iloc[2]
```

Out[221…
```
np.int64(300)
```

Ahora ya no hay ningun tipo de ambigüedad a la hora de indexar los elementos ya que buscamos específicamente dentro del índice deseado y obviamos el otro.

De nuevo, con estos métodos podemos indexar empleando *slicing*, *fancy indexing* o máscaras booleanas.

In [222…
```
serie_index.loc[2:4]
```

Out[222…
```
2    100
3    200
4    300
dtype: int64
```

In [223…
```
serie_index.iloc[2:4]
```

Out[223…
```
4    300
5    400
dtype: int64
```

In [224…
```
serie_index.loc[[2, 4]]
```

Out[224…
```
2    100
4    300
dtype: int64
```

In [225…
```
serie_index.iloc[[2, 4]]
```

Out[225…
```
4    300
6    500
dtype: int64
```

Eliminar elementos

Para eliminar un elemento de una **Serie** de *pandas*, hay que eliminar el valor asociado en el **Index**. Sin embargo, este último es un objeto inmutable. Para lograrlo, se debe emplear el método `.drop()`, el cual devuelve una nueva **Serie** con el índice actualizado explícitamente (es decir, no modifica el objeto **Index** original, sino que crea uno nuevo).

In [226…
```
serie = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
print(f"El id del objeto Serie es: {hex(id(serie))}")
print(f"El id del objeto Index asociado a la Serie es: {hex(id(serie.index))}")
print(serie)
```

```
El id del objeto Serie es: 0x25768ed0aa0
El id del objeto Index asociado a la Serie es: 0x2577e77f6b0
a    10
b    20
c    30
d    40
dtype: int64
```

In [227…
```
serie = serie.drop(labels=["a", "b"])
print(serie)
print(serie.index)
```

```
c    30
d    40
dtype: int64
Index(['c', 'd'], dtype='object')
```

In [228…
```
print(f"El id del objeto Serie es: {hex(id(serie))}")
print(f"El id del nuevo objeto Index asociado a la Serie es: {hex(id(serie.index
```

```
El id del objeto Serie es: 0x2577e77c8f0
El id del nuevo objeto Index asociado a la Serie es: 0x2577e737860
```

El método `.drop()` posee un parámetro booleano llamado `inplace`, cuyo valor por defecto es `False`:

- Cuando `inplace=False`, se crea una copia del objeto **Serie** con la operación aplicada, y dicha copia es la que se devuelve (por lo que es necesario asignar el resultado a una nueva variable para poder utilizarla).
- Cuando `inplace=True`, la operación se realiza *in situ*, es decir, directamente sobre el mismo objeto **Serie**, sin devolver una nueva copia.

In [229…
```
serie = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
print(f"El id del objeto Serie es: {hex(id(serie))}")
print(f"El id del objeto Index asociado a la Serie es: {hex(id(serie.index))}")
print(serie)
```

```
El id del objeto Serie es: 0x25768ed0aa0
El id del objeto Index asociado a la Serie es: 0x2577e77f6b0
a    10
b    20
c    30
d    40
dtype: int64
```

In [230…
```
serie.drop(labels=["a", "b"], inplace=True)
print(serie)
print(serie.index)
```

```
c    30
d    40
dtype: int64
Index(['c', 'd'], dtype='object')
```

In [231…
```
print(f"El id del objeto Serie es: {hex(id(serie))}")
print(f"El id del nuevo objeto Index asociado a la Serie es: {hex(id(serie.index
```

```
El id del objeto Serie es: 0x25768ed0aa0
El id del nuevo objeto Index asociado a la Serie es: 0x2577e735370
```

Por tanto, los objetos **Serie** son mutables.

Otros atributos y métodos importantes de las Series

El atributo `.index` devuelve el objeto **Index** (índice explícito) de una **Serie**

In [232…
```
serie = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
serie.index
```

Out[232…
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

El atributo `.values` devuelve un **array** de *NumPy* con los valores de la **Serie**

In [233…
```
serie.values
```

Out[233…
```
array([10, 20, 30, 40])
```

El atributo `.name` devuelve el nombre de la **Serie**, puede especificarse durante su creación con el atributo `name` en el constructor `pd.Series()` o de la siguiente forma

In [234…
```
serie.name = "Valores"
serie
```

Out[234…
```
a    10
b    20
c    30
d    40
Name: Valores, dtype: int64
```

Esto será especialmente útil con los **DataFrames**.

El método `.items()` devuelve un objeto `zip` que asocia los valores del **array** con sus respectivos índices explícitos

In [235…
```
serie.items()
```

Out[235…
```
<zip at 0x2576aefbcc0>
```

In [236…
```
list(serie.items())
```

Out[236…
```
[('a', 10), ('b', 20), ('c', 30), ('d', 40)]
```

este tipo de objetos nos permite iterar sobre las **tuplas** que contienen los pares de índice y valor

In [237…
```
for index, value in serie.items():
    print(f"Índice explícito: {index}, Valor: {value}")
```

```
Índice explícito: a, Valor: 10
Índice explícito: b, Valor: 20
Índice explícito: c, Valor: 30
Índice explícito: d, Valor: 40
```

En este sentido, las **Series** tienen un comportamiento muy similar a los **diccionarios** con la salvedad de que se pueden tener índices repetidos (las llaves de un **diccionario** deben de ser únicas) como consecuencia de la propiedad de no unicidad de los objetos **Index**

In [238…
```
geo = pd.Series(["Madrid", "España", "París", "Francia"], index = ["Ciudad", "Pa
geo
```

Out[238…
```
Ciudad     Madrid
País       España
Ciudad      París
País      Francia
dtype: object
```

In [239…
```
for entidad, elemento in geo.items():
    print(f"Entidad geográfica: {elemento}, Tipo de entidad geográfica: {entidad
```

```
Entidad geográfica: Madrid, Tipo de entidad geográfica: Ciudad
Entidad geográfica: España, Tipo de entidad geográfica: País
Entidad geográfica: París, Tipo de entidad geográfica: Ciudad
Entidad geográfica: Francia, Tipo de entidad geográfica: País
```

In [240…
```
geo["Ciudad"]
```

Out[240…
```
Ciudad    Madrid
Ciudad     París
dtype: object
```

Como consecuencia de esto, si se desea cambiar el valor de un elemento dentro de la **Serie**, se recomienda emplear los índices posicionales o utilizar *slicing* o *fancy indexing* o máscaras booleanas con índices explícitos, ya que al usar etiquetas se podrían modificar por error más elementos además del deseado.

In [241…
```
geo["Ciudad"] = "Londres"
geo
```

Out[241…
```
Ciudad    Londres
País       España
Ciudad    Londres
País      Francia
dtype: object
```

DataFrames

Un **DataFrame** es una estructura de datos bidimensional que consiste en una colección de objetos **Series** que comparten un índice explícito común para las filas. Esto significa que tanto las filas como las columnas están formadas por **Series** alineadas (estructuras unidimensionales), lo que da lugar a una estructura bidimensional similar a una tabla. De tal forma que facilitan la manipulación, análisis y visualización de datos tabulares.

El índice explícito de las columnas normalmente corresponde al encabezado de estas y el de las filas será el identificador único de cada registro. Estos índices pueden ser numéricos, alfabéticos o de cualquier otro tipo de datos, proporcionando flexibilidad en la organización y acceso a los datos.

Construcción de **DataFrames**

Para construir un objeto **DataFrame**, disponemos de la función `pd.DataFrame`, a la que podemos pasar diferentes tipos de estructuras de datos, como una **lista** de **Series**, una **lista** de **listas**, una **lista** de **diccionarios**, un **diccionario** de **Series** o un **diccionario** de **diccionarios**. En algunos casos, los índices explícitos de las filas y las columnas se generan automáticamente como se espera, mientras que en otros es necesario definirlos manualmente para obtener la estructura deseada.

Construcción de un **DataFrame** desde una **lista** de **Series**

Primero de todo, voy a crear un objeto **Index** para las columnas:

In [242…
```
indice = pd.Index(["España", "Francia", "Alemania", "Italia"])
```

A continuación, se declaran las **Series** que conformarán las columnas del **DataFrame**. En cada una de ellas, se asigna el atributo `index` al objeto **Index** previamente definido, con el fin de identificar sus valores con las etiquetas correspondientes. Además, mediante el atributo `name` se establece el nombre de la columna, es decir, la etiqueta con la que identificamos cada columna:

In [243…
```
capitales = pd.Series(["Madrid", "París", "Berlín", "Roma"], index = indice, nam
poblacion = pd.Series([48.35, 68.29, 83.28, 58.99], index = indice, name = "Pobl
PIB = pd.Series([1.62, 2.3, 3.05, 4.52], index = indice, name = "PIB")
```

Por último, creamos el **DataFrame** con el constructor `pd.DataFrame()` al pasarle una **lista** con las **Series** creadas previamente:

In [244…
```
paises = pd.DataFrame([capitales, poblacion, PIB])
paises
```

Out[244…

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

Podemos observar que las columnas provienen del objeto **Index** que habíamos creado, mientras que las filas se generan a partir de las **Series** declaradas.

Podemos transponer el **DataFrame** de modo que las filas representen a los países y las columnas a sus atributos (Capital, Población y PIB), utilizando el método `transpose()` o el atributo `T`. Esta operación no modifica el **DataFrame** original, sino que devuelve uno nuevo:

In [245…
```
paises.transpose()
```

Out[245…

	Capital	Población	PIB
España	Madrid	48.35	1.62
Francia	París	68.29	2.3
Alemania	Berlín	83.28	3.05
Italia	Roma	58.99	4.52

Construcción de un **DataFrame** desde un **diccionario** de **Series**

Primero que todo, debemos declarar el diccionario con la información que queremos que contenga el **DataFrame**. En dicho diccionario, las llaves representarán los nombres de las columnas (es decir, su índice), y los valores serán las **Series** definidas previamente:

In [246…
```
d = {
    "Capital": capitales,
    "Población": poblacion,
    "PIB": PIB
}
```

Por último, creamos el **DataFrame** utilizando el constructor `pd.DataFrame()`, al cual le pasamos el **diccionario** que hemos definido previamente:

In [247…
```
paises = pd.DataFrame(d)
paises
```

Out[247…

	Capital	Población	PIB
España	Madrid	48.35	1.62
Francia	París	68.29	2.30
Alemania	Berlín	83.28	3.05
Italia	Roma	58.99	4.52

Podemos observar que al crear el **DataFrame** mediante un **diccionario**, las columnas son las llaves de este.

En caso de que no queremos que tenga esta forma el **DataFrame**, podemos transponerlo como en el ejemplo anterior:

In [248…
```
paises.transpose()
```

Out[248…

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

Construcción de un **DataFrame** desde un **diccionario** de **diccionarios**

Creamos un **diccionario** para cada una de las columnas del **DataFrame** final, donde las llaves representan las etiquetas de las filas (en este caso, los países) y los valores corresponden a la información asociada a cada fila dentro de la columna que representa dicho **diccionario**:

In [249…
```
capitales_d = {
    "España": "Madrid",
    "Francia": "París",
    "Alemania": "Berlín",
    "Italia": "Roma"
}

poblacion_d = {
```

```
    "España": 48.35,
    "Alemania": 83.28,
    "Italia": 58.99,
    "Francia": 68.29
}

PIB_d = {
    "España": 1.62,
    "Francia": 2.3,
    "Alemania": 3.05,
    "Italia": 4.52
}
```

Nótese que las llaves no tienen por qué estar en el mismo orden en todos los **diccionarios**. De hecho, algunas incluso pueden faltar; en ese caso, *pandas* rellenará automáticamente el hueco correspondiente con un valor `NaN`.

A continuación, declaramos nuestro **diccionario** que va a contener los **diccionarios** anteriores:

In [250…
```
d = {
    "Capital": capitales_d,
    "Población": poblacion_d,
    "PIB": PIB_d
}
d
```

Out[250…
```
{'Capital': {'España': 'Madrid',
  'Francia': 'París',
  'Alemania': 'Berlín',
  'Italia': 'Roma'},
 'Población': {'España': 48.35,
  'Alemania': 83.28,
  'Italia': 58.99,
  'Francia': 68.29},
 'PIB': {'España': 1.62, 'Francia': 2.3, 'Alemania': 3.05, 'Italia': 4.52}}
```

Por último, creamos el **DataFrame** utilizando el constructor `pd.DataFrame()`, al cual le pasamos el **diccionario** de **diccionarios**:

In [251…
```
paises = pd.DataFrame(d)
paises
```

Out[251…

	Capital	Población	PIB
España	Madrid	48.35	1.62
Francia	París	68.29	2.30
Alemania	Berlín	83.28	3.05
Italia	Roma	58.99	4.52

Construcción de un **DataFrame** desde una **lista** de **diccionarios**

A continuación, vamos a crear un **DataFrame** a partir de una lista de diccionarios. Para ello, utilizaremos los diccionarios definidos en el caso anterior:

In [252…
```
lista = [capitales_d, poblacion_d, PIB_d]
```

Pasamos la **lista** al constructor `pd.DataFrame()` para crear el **DataFrame**:

In [253…
```
paises = pd.DataFrame(lista)
paises
```

Out[253…

	España	Francia	Alemania	Italia
0	Madrid	París	Berlín	Roma
1	48.35	68.29	83.28	58.99
2	1.62	2.3	3.05	4.52

Como estamos pasando una **lista** y no un **diccionario**, *pandas* no puede inferir automáticamente el índice que queremos asociar a las filas (recordemos que, en el caso de un **diccionario**, las llaves se utilizan como índice). Por ello, es necesario declararlo manualmente mediante el parámetro opcional `index`, al cual le pasamos una **lista** con las etiquetas correspondientes:

In [254…
```
paises_index = pd.DataFrame(lista, index = ["Capital", "Población", "PIB"])
paises_index
```

Out[254…

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

Otra alternativa habría sido utilizar el método `rename()`, que permite renombrar el índice de las filas del **DataFrame** mediante el parámetro `index`. Este parámetro acepta un diccionario donde cada llave representa la etiqueta que se desea modificar, y su valor correspondiente es la nueva etiqueta:

In [255…
```
paises_rename = paises.rename(
    index = {
        0: "Capital",
        1: "Población",
        2: "PIB"
    }
)
paises_rename
```

Out[255...

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

No es necesario renombrar todas las etiquetas de las filas; podemos omitir aquellas que no deseamos modificar:

In [256...

```
paises_rename = paises.rename(
    index = {
        0: "Capital",
        2: "PIB"
    }
)
paises_rename
```

Out[256...

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
1	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

Construcción de un **DataFrame** desde una **lista** de **listas**

Primero de todo, vamos a generar las **listas** que corresponden a las filas del **DataFrame**:

In [257...

```
paises_lista = ["España", "Francia", "Alemania", "Italia"]
capitales_lista = ["Madrid", "París", "Berlín", "Roma"]
poblacion_lista = [48.35, 68.29, 83.28, 58.99]
pib_lista = [1.62, 2.3, 3.05, 4.52]
```

Una vez que tenemos las **listas** con la información que queremos almacenar, construimos el **DataFrame** utilizando el constructor `pd.DataFrame()`, al que le pasamos los argumentos opcionales `index`, para definir las etiquetas de las filas, y `columns`, para definir las etiquetas de las columnas:

In [258...

```
paises = pd.DataFrame(
    [capitales_lista, poblacion_lista, pib_lista],
    index = ["Capital", "Población", "PIB"],
    columns = [paises_lista]
)
paises
```

Out[258...

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

Métodos y propiedades de los **DataFrames**

El método `info()` imprime información relevante sobre el **DataFrame** como el número total de columnas, las etiquetas de cada columna, los tipos de datos que contienen, el uso de memoria, el rango del índice y la cantidad de valores no nulos en cada columna.

In [259...

```
paises.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3 entries, Capital to PIB
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   (España,)    3 non-null      object
 1   (Francia,)   3 non-null      object
 2   (Alemania,)  3 non-null      object
 3   (Italia,)    3 non-null      object
dtypes: object(4)
memory usage: 120.0+ bytes
```

El método `describe()` devuelve información estadística relativa al **DataFrame**, como la media, el valor mínimo, el valor máximo, los cuartiles y la desviación estándar. Por defecto, solo opera sobre las columnas numéricas del **DataFrame**.

Sin embargo, al declarar el parámetro opcional **`include`** y establecerlo como **`"all"`**, también se incluyen las columnas categóricas. En este caso, se muestran estadísticas como la cantidad de valores únicos, el valor más frecuente (*top*) y su frecuencia (*freq*):

In [260...

```
df = pd.DataFrame(
    {
    'Edad': [25, 22, 23, 25, 24],
    'Salario': [50000, 54000, 50000, 48500, 60000],
    'Sexo': ['Hombre', 'Mujer', 'Mujer', 'Hombre', 'Hombre']
    })
df
```

Out[260...

	Edad	Salario	Sexo
0	25	50000	Hombre
1	22	54000	Mujer
2	23	50000	Mujer
3	25	48500	Hombre
4	24	60000	Hombre

In [261…
```
df.describe()
```

Out[261…

	Edad	Salario
count	5.00000	5.000000
mean	23.80000	52500.000000
std	1.30384	4663.689527
min	22.00000	48500.000000
25%	23.00000	50000.000000
50%	24.00000	50000.000000
75%	25.00000	54000.000000
max	25.00000	60000.000000

In [262…
```
df.describe(include="all")
```

Out[262…

	Edad	Salario	Sexo
count	5.00000	5.000000	5
unique	NaN	NaN	2
top	NaN	NaN	Hombre
freq	NaN	NaN	3
mean	23.80000	52500.000000	NaN
std	1.30384	4663.689527	NaN
min	22.00000	48500.000000	NaN
25%	23.00000	50000.000000	NaN
50%	24.00000	50000.000000	NaN
75%	25.00000	54000.000000	NaN
max	25.00000	60000.000000	NaN

El método `nunique()` devuelve la cantidad de valores únicos en las columnas categóricas del **DataFrame**:

In [263…
```
df.nunique()
```

Out[263…
```
Edad       4
Salario    4
Sexo       2
dtype: int64
```

El método `unique()` devuelve un **array** con los valores únicos de una columna del **DataFrame**:

In [264…
```
df["Sexo"].unique()
```

Out[264…
```
array(['Hombre', 'Mujer'], dtype=object)
```

El método `value_counts()` devuelve una **Serie** con la frecuencia de cada valor único.

Si se aplica sobre todo el **DataFrame**, se considera un valor único aquel que sea igual en todas sus columnas (es decir, filas idénticas). En cambio, si se aplica sobre una columna específica, solo se tiene en cuenta dicha columna para calcular la frecuencia de sus valores.

La **Serie** resultante omite por defecto los valores `NaN`, aunque se pueden incluir estableciendo el parámetro `dropna=False`:

In [265…
```
df.value_counts()
```

Out[265…
```
Edad  Salario  Sexo
22    54000    Mujer     1
23    50000    Mujer     1
24    60000    Hombre    1
25    48500    Hombre    1
      50000    Hombre    1
Name: count, dtype: int64
```

In [266…
```
df["Edad"].value_counts()
```

Out[266…
```
Edad
25    2
22    1
23    1
24    1
Name: count, dtype: int64
```

El método `count()` devuelve la cantidad de observaciones no faltantes en un **DataFrame**:

In [267…
```
df.count()
```

Out[267…
```
Edad       5
Salario    5
Sexo       5
dtype: int64
```

También puede aplicarse a columnas individuales, en cuyo caso devuelve un único valor correspondiente al número de elementos no faltantes en esa columna:

In [268…
```
df["Edad"].count()
```

Out[268…
```
np.int64(5)
```

Existen métodos que permiten calcular propiedades estadísticas de las columnas del **DataFrame**, como:

- `mean()` — devuelve la media de cada columna.
- `std()` — devuelve la desviación estándar.

- `median()` — devuelve la mediana.
- `quantile()` — permite calcular un cuantil específico.

En el caso del método `quantile()`, hay que indicar el valor del cuantil mediante el parámetro opcional `q`, cuyo valor por defecto es 0.5. Este parámetro debe tomar un valor dentro del intervalo $0 \leq q \leq 1$:

In [269…
```
df["Salario"].mean()
```
Out[269…
```
np.float64(52500.0)
```

In [270…
```
df["Salario"].std()
```
Out[270…
```
np.float64(4663.689526544407)
```

In [271…
```
df["Salario"].median()
```
Out[271…
```
np.float64(50000.0)
```

In [272…
```
df["Salario"].quantile()
```
Out[272…
```
np.float64(50000.0)
```

In [273…
```
df["Salario"].quantile(q = 0.2)
```
Out[273…
```
np.float64(49700.0)
```

El método `head()` muestra las primeras n observaciones del **DataFrame**. El número de observaciones se indica mediante el parámetro opcional `n`, cuyo valor por defecto es 5:

In [274…
```
paises.head(n = 3)
```
Out[274…

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

El método `tail()` muestra las últimas n observaciones del **DataFrame**. El número de observaciones se indica mediante el parámetro opcional `n`, cuyo valor por defecto es 5:

In [275…
```
paises.tail(n = 3)
```
Out[275…

	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

El método `transpose()` transpone el **DataFrame** como si fuera una matriz. El atributo `T` es un *getter* que permite acceder al resultado del método `transpose()`.

El método `rename()` permite renombrar las etiquetas de los índices, tanto de las filas como de las columnas.

El método `set_index()` emplea una columna ya presente en el **DataFrame** como índice.

In [276…
```
paises = pd.DataFrame(
    [paises_lista, capitales_lista, poblacion_lista, pib_lista],
    index = ["Pais", "Capital", "Población", "PIB"],
)
paises
```

Out[276…

	0	1	2	3
Pais	España	Francia	Alemania	Italia
Capital	Madrid	París	Berlín	Roma
Población	48.35	68.29	83.28	58.99
PIB	1.62	2.3	3.05	4.52

Es importante destacar que este método tiene el atributo `drop`, que por defecto tiene el valor `True`. Este valor hace que se elimine la columna que se estaba utilizando como índice, reemplazándola por el nuevo índice. En este caso, no nos interesa perder la información relativa al país en caso de futuros cambios del índice, por lo que cambiamos el valor de `drop` a `False`. Nótese que, al hacerlo, tendremos dos columnas con la misma información: una que actúa como índice y otra que permanece como la columna `Pais`:

In [277…
```
paises = paises.transpose()
paises = paises.set_index("Pais", drop = False)
paises
```

Out[277…

	Pais	Capital	Población	PIB
Pais				
España	España	Madrid	48.35	1.62
Francia	Francia	París	68.29	2.3
Alemania	Alemania	Berlín	83.28	3.05
Italia	Italia	Roma	58.99	4.52

Este método es de gran utilidad, ya que nos permite filtrar según la propiedad que deseemos dentro del **DataFrame** utilizando el atributo `.loc[]` de las **Series**:

In [278…
```
paises.loc["España"]
```

Out[278…
```
Pais          España
Capital       Madrid
Población      48.35
PIB             1.62
Name: España, dtype: object
```

Podemos cambiar la propiedad según la información a la que queramos acceder. Por ejemplo, supongamos que ahora queremos buscar toda la información relacionada con la ciudad de Madrid:

In [279…
```
paises = paises.set_index("Capital", drop = False)
paises
```

Out[279…

Capital	Pais	Capital	Población	PIB
Madrid	España	Madrid	48.35	1.62
París	Francia	París	68.29	2.3
Berlín	Alemania	Berlín	83.28	3.05
Roma	Italia	Roma	58.99	4.52

In [280…
```
paises.loc["Madrid"]
```

Out[280…
```
Pais          España
Capital       Madrid
Población      48.35
PIB             1.62
Name: Madrid, dtype: object
```

Nótese que ahora no podríamos buscar por el país España, ya que ya no se encuentra en el índice, lo que provocaría un `KeyError`:

In [281…
```
paises.loc["España"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pan
das\core\indexes\base.py:3805, in Index.get_loc(self, key)
   3804 try:
-> 3805     return self._engine.get_loc(casted_key)
   3806 except KeyError as err:

File index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas\\_libs\\hashtable_class_helper.pxi:7081, in pandas._libs.hashtable.Py
ObjectHashTable.get_item()

File pandas\\_libs\\hashtable_class_helper.pxi:7089, in pandas._libs.hashtable.Py
ObjectHashTable.get_item()

KeyError: 'España'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[281], line 1
----> 1 paises.loc["España"]

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pan
das\core\indexing.py:1191, in _LocationIndexer.__getitem__(self, key)
   1189 maybe_callable = com.apply_if_callable(key, self.obj)
   1190 maybe_callable = self._check_deprecated_callable_usage(key, maybe_callabl
e)
-> 1191 return self._getitem_axis(maybe_callable, axis=axis)

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pan
das\core\indexing.py:1431, in _LocIndexer._getitem_axis(self, key, axis)
   1429 # fall thru to straight lookup
   1430 self._validate_key(key, axis)
-> 1431 return self._get_label(key, axis=axis)

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pan
das\core\indexing.py:1381, in _LocIndexer._get_label(self, label, axis)
   1379 def _get_label(self, label, axis: AxisInt):
   1380     # GH#5567 this will fail if the label is not present in the axis.
-> 1381     return self.obj.xs(label, axis=axis)

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pan
das\core\generic.py:4301, in NDFrame.xs(self, key, axis, level, drop_level)
   4299             new_index = index[loc]
   4300 else:
-> 4301     loc = index.get_loc(key)
   4303     if isinstance(loc, np.ndarray):
   4304         if loc.dtype == np.bool_:

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pan
das\core\indexes\base.py:3812, in Index.get_loc(self, key)
   3807     if isinstance(casted_key, slice) or (
   3808         isinstance(casted_key, abc.Iterable)
   3809         and any(isinstance(x, slice) for x in casted_key)
   3810     ):
   3811         raise InvalidIndexError(key)
```

```
-> 3812     raise KeyError(key) from err
   3813 except TypeError:
   3814     # If we have a listlike key, _check_indexing_error will raise
   3815     #  InvalidIndexError. Otherwise we fall through and re-raise
   3816     #  the TypeError.
   3817     self._check_indexing_error(key)

KeyError: 'España'
```

El atributo `index` es el objeto **Index** empleado para etiquetar las filas del **DataFrame**.

In [282…
```
paises.index
```

Out[282…
```
Index(['Madrid', 'París', 'Berlín', 'Roma'], dtype='object', name='Capital')
```

Podemos cambiar el nombre del índice utilizando este atributo de la siguiente manera:

In [283…
```
paises.index.name = "Ciudades"
paises
```

Out[283…

Ciudades	Pais	Capital	Población	PIB
Madrid	España	Madrid	48.35	1.62
París	Francia	París	68.29	2.3
Berlín	Alemania	Berlín	83.28	3.05
Roma	Italia	Roma	58.99	4.52

o, podemos directamente no ponerle ningún nombre:

In [284…
```
paises.index.name = None
paises
```

Out[284…

	Pais	Capital	Población	PIB
Madrid	España	Madrid	48.35	1.62
París	Francia	París	68.29	2.3
Berlín	Alemania	Berlín	83.28	3.05
Roma	Italia	Roma	58.99	4.52

El atributo `columns` es el objeto **Index** empleado para etiquetar las columnas del **DataFrame**.

In [285…
```
paises.columns
```

Out[285…
```
Index(['Pais', 'Capital', 'Población', 'PIB'], dtype='object')
```

El método `drop()` se emplea para eliminar filas o columnas de un **DataFrame**.

In [286…
```
paises = paises.set_index("Pais", drop = False)
```

In [287…
```
paises_sin_capital = paises.drop(columns = "Capital")
paises_sin_capital
```

Out[287…

Pais	Pais	Población	PIB
España	España	48.35	1.62
Francia	Francia	68.29	2.3
Alemania	Alemania	83.28	3.05
Italia	Italia	58.99	4.52

In [288…
```
paises_sin_capital_ni_pib = paises.drop(columns = ["Capital", "PIB"])
paises_sin_capital_ni_pib
```

Out[288…

Pais	Pais	Población
España	España	48.35
Francia	Francia	68.29
Alemania	Alemania	83.28
Italia	Italia	58.99

In [289…
```
paises_sin_francia = paises.drop(index = ["Francia"])
paises_sin_francia
```

Out[289…

Pais	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Alemania	Alemania	Berlín	83.28	3.05
Italia	Italia	Roma	58.99	4.52

In [290…
```
paises_sin_francia_ni_alemania = paises.drop(index = ["Francia", "Alemania"])
paises_sin_francia_ni_alemania
```

Out[290…

Pais	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Italia	Italia	Roma	58.99	4.52

Seleccionar Datos

In [291…
```
paises.index.name = None
```

Se ha mencionado previamente que un **DataFrame** es una **Serie** de **Series** en el que el eje 0 corresponde a las filas y el eje 1 a las columnas. Por tanto cada columna es una **Serie** de valores a la que podemos acceder mediente `[]`:

In [292…
```
paises["Capital"]
```

Out[292…
```
España      Madrid
Francia      París
Alemania    Berlín
Italia        Roma
Name: Capital, dtype: object
```

Cuando utilizamos `[]`, siempre se seleccionan las columnas mediante su índice explícito. En caso de intentar acceder a una fila de esta manera, se producirá un error del tipo `KeyError`:

In [293…
```
paises["Francia"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\indexes\base.py:3805, in Index.get_loc(self, key)
   3804 try:
-> 3805     return self._engine.get_loc(casted_key)
   3806 except KeyError as err:

File index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas\\_libs\\hashtable_class_helper.pxi:7081, in pandas._libs.hashtable.PyObjectHashTable.get_item()

File pandas\\_libs\\hashtable_class_helper.pxi:7089, in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'Francia'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[293], line 1
----> 1 paises["Francia"]

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\frame.py:4102, in DataFrame.__getitem__(self, key)
   4100 if self.columns.nlevels > 1:
   4101     return self._getitem_multilevel(key)
-> 4102 indexer = self.columns.get_loc(key)
   4103 if is_integer(indexer):
   4104     indexer = [indexer]

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\indexes\base.py:3812, in Index.get_loc(self, key)
   3807     if isinstance(casted_key, slice) or (
   3808         isinstance(casted_key, abc.Iterable)
   3809         and any(isinstance(x, slice) for x in casted_key)
   3810     ):
   3811         raise InvalidIndexError(key)
-> 3812     raise KeyError(key) from err
   3813 except TypeError:
   3814     # If we have a listlike key, _check_indexing_error will raise
   3815     #  InvalidIndexError. Otherwise we fall through and re-raise
   3816     #  the TypeError.
   3817     self._check_indexing_error(key)

KeyError: 'Francia'
```

Tampoco se pueden utilizar índices posicionales con `[]`; esto también generará un error del tipo `KeyError`:

In [294…

```
paises[1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\indexes\base.py:3805, in Index.get_loc(self, key)
   3804 try:
-> 3805     return self._engine.get_loc(casted_key)
   3806 except KeyError as err:

File index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas\\_libs\\hashtable_class_helper.pxi:7081, in pandas._libs.hashtable.PyObjectHashTable.get_item()

File pandas\\_libs\\hashtable_class_helper.pxi:7089, in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 1

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[294], line 1
----> 1 paises[1]

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\frame.py:4102, in DataFrame.__getitem__(self, key)
   4100 if self.columns.nlevels > 1:
   4101     return self._getitem_multilevel(key)
-> 4102 indexer = self.columns.get_loc(key)
   4103 if is_integer(indexer):
   4104     indexer = [indexer]

File c:\Users\Rubén\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\core\indexes\base.py:3812, in Index.get_loc(self, key)
   3807     if isinstance(casted_key, slice) or (
   3808         isinstance(casted_key, abc.Iterable)
   3809         and any(isinstance(x, slice) for x in casted_key)
   3810     ):
   3811         raise InvalidIndexError(key)
-> 3812     raise KeyError(key) from err
   3813 except TypeError:
   3814     # If we have a listlike key, _check_indexing_error will raise
   3815     #  InvalidIndexError. Otherwise we fall through and re-raise
   3816     #  the TypeError.
   3817     self._check_indexing_error(key)

KeyError: 1
```

Podemos emplear dos `[]` de manera consecutiva, el primero para indicar la columna y el segundo para indicar la fila, de la siguiente manera:

In [295…
```
paises["Capital"]["España"]
```

Out[295…
```
'Madrid'
```

Se pueden seleccionar celdas específicas mediante el índice posicional utilizando el atributo `values`, que devuelve los valores del **DataFrame** en forma de un **array** bidimensional de *NumPy*. A continuación, se pueden usar `[]` para seleccionar la fila y la columna deseadas:

In [296…
```
paises.values
```

Out[296…
```
array([['España', 'Madrid', 48.35, 1.62],
       ['Francia', 'París', 68.29, 2.3],
       ['Alemania', 'Berlín', 83.28, 3.05],
       ['Italia', 'Roma', 58.99, 4.52]], dtype=object)
```

In [297…
```
paises.values[0]
```

Out[297…
```
array(['España', 'Madrid', 48.35, 1.62], dtype=object)
```

Podemos seleccionar una celda concreta especificando la fila y la columna (de nuevo, el primer valor indica la fila y el segundo la columna):

In [298…
```
paises.values[0][1]
```

Out[298…
```
'Madrid'
```

también se puede hacer de la siguiente manera:

In [299…
```
paises.values[0, 1]
```

Out[299…
```
'Madrid'
```

Esto último funciona porque se esta trabajando sobre un **array** de *NumPy* y no sobre un **DataFrame** de *pandas*.

En general, lo más recomendable a la hora de seleccionar datos dentro de un **DataFrame** es utilizar `.loc` e `.iloc`, ya que resultan más claros y versátiles. Además, permiten seleccionar múltiples filas y/o columnas de manera simultánea y sencilla.

Funcionan exactamente igual que en las **Series**, con la única salvedad de que ahora se trabaja sobre dos ejes (el eje 0 corresponde a las filas y el eje 1 a las columnas):

In [300…
```
paises.loc["España", # Eje 0 -> filas
           "Capital" # Eje 1 -> columnas
          ]
```

Out[300…
```
'Madrid'
```

In [301…
```
paises.iloc[0,
            1]
```

Out[301…
```
'Madrid'
```

Para seleccionar simultáneamente varias filas y/o varias columnas, es necesario pasarlas dentro de una **lista**:

In [302…
```
paises.loc[["España", "Italia"],
           ["Capital", "Población"]]
```

Out[302…

	Capital	Población
España	Madrid	48.35
Italia	Roma	58.99

In [303…
```
paises.iloc[[0, 3],
            [1, 2]]
```

Out[303…

	Capital	Población
España	Madrid	48.35
Italia	Roma	58.99

Se puede emplear también *slicing* y *fancy indexing* de igual manera que en las **Series**:

In [304…
```
paises.iloc[0:3:2,
            1:3]
```

Out[304…

	Capital	Población
España	Madrid	48.35
Alemania	Berlín	83.28

In [305…
```
paises.iloc[[0, 2],
            [1, 2]]
```

Out[305…

	Capital	Población
España	Madrid	48.35
Alemania	Berlín	83.28

Reemplazar Datos

Se pueden reemplazar valores empleando el operador de asignación `=` que nos permite

- Reemplazar una sola celda dentro del **DataFrame**
- Reemplazar múltiples celdas (se puede emplear *slicing* o *fancy indexing*)

In [306…
```
paises_nuevo = paises.copy()
paises_nuevo.loc["Francia",
                 "Capital"] = "Calais"

paises_nuevo
```

Out[306...

	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Francia	Francia	Calais	68.29	2.3
Alemania	Alemania	Berlín	83.28	3.05
Italia	Italia	Roma	58.99	4.52

In [307...
```
paises_nuevo.iloc[1:3] = [["Grecia", "Atenas", "10.41", "0.253"],
                          ["Rusia", "Moscú", "143.8", "2.021"]]
paises_nuevo
```

Out[307...

	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Francia	Grecia	Atenas	10.41	0.253
Alemania	Rusia	Moscú	143.8	2.021
Italia	Italia	Roma	58.99	4.52

(Nótese que si falta algún valor, saltará un `ValueError`)

Para actualizar el índice en el ejemplo anterior, se puede hacer mediante el atributo `index` o mediante el método `set_index()`:

In [308...
```
paises_nuevo.index = ["España", "Grecia", "Rusia", "Italia"]
paises_nuevo
```

Out[308...

	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Grecia	Grecia	Atenas	10.41	0.253
Rusia	Rusia	Moscú	143.8	2.021
Italia	Italia	Roma	58.99	4.52

In [309...
```
paises_nuevo.set_index("Pais", drop = False)
```

Out[309...

	Pais	Capital	Población	PIB
Pais				
España	España	Madrid	48.35	1.62
Grecia	Grecia	Atenas	10.41	0.253
Rusia	Rusia	Moscú	143.8	2.021
Italia	Italia	Roma	58.99	4.52

In [310...
```
paises_nuevo.index.name = None
```

Nótese que se puede realizar una sustitución utilizando **DataFrames** en caso de querer modificar múltiples columnas o filas, o **Series** si se desea sustituir únicamente una fila o una columna:

In [311…
```
paises_nuevo.iloc[3] = pd.Series(["Inglaterra", "Londres", 56.48, 3.382],
                                 index = ["Pais", "Capital", "Población", "PIB"])

paises_nuevo = paises_nuevo.set_index("Pais", drop = False)
paises_nuevo.index.name = None
paises_nuevo
```

Out[311…

	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Grecia	Grecia	Atenas	10.41	0.253
Rusia	Rusia	Moscú	143.8	2.021
Inglaterra	Inglaterra	Londres	56.48	3.382

In [312…
```
paises_nuevo.iloc[1:3] = pd.DataFrame([
                                        ["Francia", "París", 68.29, 2.3],
                                        ["Alemania", "Berlín", 83.28, 3.05]
                                      ], columns=["Pais", "Capital", "Población"

paises_nuevo = paises_nuevo.set_index("Pais", drop = False)
paises_nuevo.index.name = None
paises_nuevo
```

Out[312…

	Pais	Capital	Población	PIB
España	España	Madrid	48.35	1.62
Francia	Francia	París	68.29	2.3
Alemania	Alemania	Berlín	83.28	3.05
Inglaterra	Inglaterra	Londres	56.48	3.382

In [313…
```
paises_nuevo.iloc[:,1] = pd.Series(["León", "Calais", "Bremen", "Manchester"])
paises_nuevo
```

Out[313…

	Pais	Capital	Población	PIB
España	España	León	48.35	1.62
Francia	Francia	Calais	68.29	2.3
Alemania	Alemania	Bremen	83.28	3.05
Inglaterra	Inglaterra	Manchester	56.48	3.382

In [314…
```
paises_nuevo.iloc[:, 0:2] = pd.DataFrame([
                                           ["Turquia", "Ankara"],
                                           ["Italia", "Roma"],
                                           ["Serbia", "Belgrado"],
                                           ["Argentina", "Buenos Aires"]
```

```
                                        ])
paises_nuevo
```

Out[314…

	Pais	Capital	Población	PIB
Turquia	Turquia	Ankara	48.35	1.62
Italia	Italia	Roma	68.29	2.3
Serbia	Serbia	Belgrado	83.28	3.05
Argentina	Argentina	Buenos Aires	56.48	3.382

Si introducimos un único valor, *pandas* lo utilizará para rellenar todas las celdas que deseamos modificar (esto se llama *broadcasting*):

In [315…

```
paises_nuevo.iloc[1:3] = "Grecia"
paises_nuevo
```

Out[315…

	Pais	Capital	Población	PIB
Turquia	Turquia	Ankara	48.35	1.62
Grecia	Grecia	Grecia	Grecia	Grecia
Grecia	Grecia	Grecia	Grecia	Grecia
Argentina	Argentina	Buenos Aires	56.48	3.382

Valores Faltantes

Es muy habitual trabajar con bases de datos que contienen valores faltantes, por lo que resulta fundamental saber cómo identificarlos, analizarlos y tratarlos adecuadamente.

El manejo correcto de los *missing values* es crucial para garantizar la calidad de los datos y la validez de los análisis posteriores. Afortunadamente, *pandas* proporciona múltiples herramientas que nos permiten detectar, eliminar o sustituir estos valores de manera eficiente.

En Python, se puede utilizar el objeto `None` para indicar que un valor es indefinido o faltante dentro de una secuencia:

In [316…

```
[1, 2, None, 4]
```

Out[316…

```
[1, 2, None, 4]
```

el caso anterior indica que no hay un tercer elemento, es decir, que falta.

El estándar IEEE para **números flotantes** incluye un concepto equivalente para representar valores faltantes: el `NaN` (*Not a Number*):

In [317…

```
float("nan")
```

Out[317… nan

Dicho objeto también se encuentra predefinido dentro del módulo *built-in math*:

In [318…
```
import math
math.nan
```

Out[318… nan

otras librerías como *NumPy* también lo tienen predefinido:

In [319…
```
import numpy as np
np.nan
```

Out[319… nan

Igualdad de NaN

Al comparar dos objetos `NaN`, siempre se devolverá `False`. Esto se debe a que no se puede comparar dos valores no definidos:

In [320…
```
float("nan") == float("nan")
```

Out[320… False

ocurre lo mismo con el operador de identidad:

In [321…
```
float("nan") is float("nan")
```

Out[321… False

Esto plantea la cuestión de cómo podemos determinar si un valor es `NaN`, dado que no podemos compararlo directamente con el objeto. Para ello, las librerías *math*, *NumPy* y *pandas* cuentan con funciones específicas para identificar valores `NaN`:

In [322…
```
math.isnan(float("nan"))
```

Out[322… True

In [323…
```
np.isnan(float("nan"))
```

Out[323… np.True_

(**np.True_** no es más que un tipo booleano **True** nativo de *NumPy*, diseñado para trabajar con **arrays** de manera más eficiente)

In [324…
```
pd.isna(float("nan"))
```

Out[324… True

En una **Serie** de *pandas*

pandas, al detectar la presencia de valores faltantes como `None` o cualquier variante de `NaN`, convierte automáticamente el tipo de toda la **Serie** a **flotante**, debido al estándar IEEE 754 sobre la aritmética de punto flotante:

In [325…
```
pd.Series([1, 2, None, float("nan"), math.nan, np.nan])
```

Out[325…
```
0    1.0
1    2.0
2    NaN
3    NaN
4    NaN
5    NaN
dtype: float64
```

Recordemos que el tipo de dato de las **Series** es el más general que satisface todos los elementos de la **Serie**. De esta forma, si introducimos un objeto de tipo **cadena** en la **Serie**, el tipo de dato de la **Serie** pasará a ser **object** en lugar de **flotante**. En una **Serie** de tipo **object**, los `None` no se convierten en `NaN`, sino que permanecen como `None`, ya que **object** puede almacenar cualquier tipo de objeto (recordemos que en Python, `None` es un objeto):

In [326…
```
pd.Series([1, 2, "hola", None, float("nan"), math.nan, np.nan])
```

Out[326…
```
0       1
1       2
2    hola
3    None
4     NaN
5     NaN
6     NaN
dtype: object
```

De tal forma que, al trabajar con **Series** de tipo **object**, sería necesario verificar tanto la presencia de datos `NaN` como la de datos `None`.

Comprobación de valores faltantes

pandas tiene las siguientes funciones que sirven para comprobar si hay valores faltantes:

- `isnull()` devuelve `True` si hay valores `Null` o `None`. Es una función universal (funciona tanto con **Series** como con **DataFrames** al estar vectorizada) y realiza una comparación elemento a elemento.
- `isnotnull()` es similar a `isnull()`, pero devuelve el resultado opuesto.
- `isna()` es un alias de `isnull()`.
- `notna()` es un alias de `isnotnull()`.

In [327…
```
pd.isnull(pd.Series([1, 2, "hola", None, float("nan"), math.nan, np.nan]))
```

Out[327…
```
0    False
1    False
2    False
3     True
4     True
5     True
6     True
dtype: bool
```

Reemplazar valores faltantes en una **Serie**

Una manera (poco práctica y costosa computacionalmente) de sustituir los valores faltantes por los deseados sería mediante un bucle. En este enfoque, se itera a lo largo de la **Serie** y se reemplazan manualmente los valores faltantes.

Sin embargo, *pandas* proporciona funciones especializadas para esta tarea:

- `fillna()` permite reemplazar los valores faltantes. Se le puede pasar un parámetro **`value`** que indica el valor específico por el que se desea sustituir el **`NaN`**. También admite el parámetro **`method`**, que especifica el método de imputación que se quiere aplicar (por ejemplo, **`ffill`** para rellenar con el valor anterior o **`bfill`** para usar el siguiente).

In [328…
```
pd.Series([1, 2, "hola", None, float("nan"), math.nan, np.nan]).fillna(0)
```

Out[328…
```
0       1
1       2
2    hola
3       0
4       0
5       0
6       0
dtype: object
```

In [329…
```
pd.Series([1, 2, "hola", None, float("nan"), math.nan, np.nan]).fillna(method =
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\531300991.py:1: FutureWarning:
Series.fillna with 'method' is deprecated and will raise in a future version. Use
obj.ffill() or obj.bfill() instead.
  pd.Series([1, 2, "hola", None, float("nan"), math.nan, np.nan]).fillna(method =
"ffill")
```

Out[329…
```
0       1
1       2
2    hola
3    hola
4    hola
5    hola
6    hola
dtype: object
```

Si el primer valor es **`Null`** (es decir, **`None`** o **`NaN`**), el método **`ffill`** no se aplicará, ya que no existe un valor anterior con el que pueda rellenarse:

In [330…
```
pd.Series([None, 1, 2, "hola", None, float("nan"), math.nan, np.nan]).fillna(met
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\2404110448.py:1: FutureWarning:
Series.fillna with 'method' is deprecated and will raise in a future version. Use
obj.ffill() or obj.bfill() instead.
  pd.Series([None, 1, 2, "hola", None, float("nan"), math.nan, np.nan]).fillna(me
thod = "ffill")
```

Out[330…
```
0    None
1       1
2       2
3    hola
4    hola
5    hola
6    hola
7    hola
dtype: object
```

In [331…
```
pd.Series([1, 2, "hola", None,float("nan"), math.nan, np.nan, 1]).fillna(method
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\2304750355.py:1: FutureWarning:
Series.fillna with 'method' is deprecated and will raise in a future version. Use
obj.ffill() or obj.bfill() instead.
  pd.Series([1, 2, "hola", None,float("nan"), math.nan, np.nan, 1]).fillna(method
= "bfill")
```

Out[331…
```
0       1
1       2
2    hola
3       1
4       1
5       1
6       1
7       1
dtype: object
```

De forma análoga, si el último valor es `Null`, el método `bfill` no se aplicará, ya que no hay un valor posterior que se pueda utilizar para completar el valor faltante:

In [332…
```
pd.Series([1, 2, "hola", None,float("nan"), math.nan,np.nan]).fillna(method = "b
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\4004917256.py:1: FutureWarning:
Series.fillna with 'method' is deprecated and will raise in a future version. Use
obj.ffill() or obj.bfill() instead.
  pd.Series([1, 2, "hola", None,float("nan"), math.nan,np.nan]).fillna(method =
"bfill")
```

Out[332…
```
0       1
1       2
2    hola
3     NaN
4     NaN
5     NaN
6     NaN
dtype: object
```

Reemplazar valores faltantes en un **DataFrame**

Funciona de igual manera que la sustitución en **Series**, con la salvedad de que en un **DataFrame** el eje sobre el que se actúa resulta relevante.

Por defecto, las operaciones como `fillna()` se aplican a lo largo del eje 0 (filas), es decir, se rellena cada columna de forma independiente. Si se desea aplicar la sustitución a lo largo de las columnas (es decir, fila por fila), es necesario especificar el parámetro `axis=1`.

In [333…
```
df = pd.DataFrame([[0.1, 0.5, -0.3, 4],
                   [9.1 , float("nan"), 1.3, 1.4],
                   [2.5 , 2.7, float("nan"), 2.4],
                   [3, 6, 1, 5.5]]
                  )
df
```

Out[333…

	0	1	2	3
0	0.1	0.5	-0.3	4.0
1	9.1	NaN	1.3	1.4
2	2.5	2.7	NaN	2.4
3	3.0	6.0	1.0	5.5

Podemos observar que aplicar el método `ffill` sobre el eje 0 (filas) devuelve un **DataFrame** diferente al que obtendríamos si lo aplicamos sobre el eje 1 (columnas).

Esto se debe a que el eje determina la dirección en la que se propagan los valores no nulos:

- Con `axis=0`, los valores se rellenan hacia abajo dentro de cada columna.
- Con `axis=1`, los valores se rellenan hacia la derecha dentro de cada fila.

In [334…
```
df.fillna(method = "ffill", axis = 0)
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\4141302941.py:1: FutureWarning:
DataFrame.fillna with 'method' is deprecated and will raise in a future version.
Use obj.ffill() or obj.bfill() instead.
  df.fillna(method = "ffill", axis = 0)
```

Out[334…

	0	1	2	3
0	0.1	0.5	-0.3	4.0
1	9.1	0.5	1.3	1.4
2	2.5	2.7	1.3	2.4
3	3.0	6.0	1.0	5.5

In [335…
```
df.fillna(method = "ffill", axis = 1)
```

```
C:\Users\Rubén\AppData\Local\Temp\ipykernel_14892\2054155988.py:1: FutureWarning:
DataFrame.fillna with 'method' is deprecated and will raise in a future version.
Use obj.ffill() or obj.bfill() instead.
  df.fillna(method = "ffill", axis = 1)
```

Out[335...

	0	1	2	3
0	0.1	0.5	-0.3	4.0
1	9.1	9.1	1.3	1.4
2	2.5	2.7	2.7	2.4
3	3.0	6.0	1.0	5.5

Interpolación de Valores Faltantes

pandas permite rellenar los valores faltantes empleando técnicas más avanzadas de interpolación mediante la función `interpolate()`. Esta función es muy versátil y admite distintos métodos de interpolación, como la interpolación lineal, polinómica o mediante *splines* de distintos órdenes, entre otras opciones.

Se recomienda visitar la documentación oficial haciendo click aquí para conocer todos los métodos disponibles.

In [336...
```
df.interpolate(method = "linear", axis = 0)
```

Out[336...

	0	1	2	3
0	0.1	0.5	-0.30	4.0
1	9.1	1.6	1.30	1.4
2	2.5	2.7	1.15	2.4
3	3.0	6.0	1.00	5.5

In [337...
```
df.interpolate(method = "polynomial", order = 2, axis = 0)
```

Out[337...

	0	1	2	3
0	0.1	0.500000	-0.300000	4.0
1	9.1	0.866667	1.300000	1.4
2	2.5	2.700000	1.733333	2.4
3	3.0	6.000000	1.000000	5.5

In [338...
```
df.interpolate(method = "spline", order = 1, axis = 0)
```

Out[338...

	0	1	2	3
0	0.1	0.500000	-0.3	4.0
1	9.1	1.914286	1.3	1.4
2	2.5	2.700000	0.9	2.4
3	3.0	6.000000	1.0	5.5

In [339…
```
df.interpolate(method = "spline", order = 2, axis = 0)
```

Out[339…

	0	1	2	3
0	0.1	0.500000	-0.300000	4.0
1	9.1	0.866667	1.300000	1.4
2	2.5	2.700000	1.733333	2.4
3	3.0	6.000000	1.000000	5.5

Cargar Datos

pandas tiene una amplia variedad de funciones *built-in* para cargar diferentes tipos de fuentes de datos, como CSV, Excel, SQL, JSON, SAS, SPSS, entre otros.

Fuente de datos	Función de pandas	Descripción
CSV	`read_csv()`	Carga archivos en formato CSV.
Excel	`read_excel()`	Carga archivos de Excel (`.xls`, `.xlsx`, entre otros).
JSON	`read_json()`	Carga archivos en formato JSON.
SQL	`read_sql()`	Ejecuta una consulta SQL y devuelve un DataFrame.
HTML	`read_html()`	Extrae tablas de archivos HTML.
Parquet	`read_parquet()`	Carga archivos en formato Parquet.
HDF5	`read_hdf()`	Lee datos almacenados en formato HDF5.
Stata (DTA)	`read_stata()`	Lee archivos de Stata (`.dta`).
SAS	`read_sas()`	Lee archivos en formato SAS (XPORT o SAS7BDAT).
SPSS (via pyreadstat)	`read_spss()`	Lee archivos de SPSS (`.sav`, `.zsav`).
Clipboard	`read_clipboard()`	Carga datos desde el portapapeles.
Feather	`read_feather()`	Carga archivos en formato Feather.

Carga de un archivo CSV

Cualquiera de las funciones anteriores funciona de manera básica de la siguiente forma:

pd.read_csv(nombre_archivo)

Nótese que, en caso de que el archivo `.py` o `.ipynb` sobre el que se está trabajando no se encuentre en el mismo directorio que el archivo que se quiere cargar, será necesario especificar el **path** completo de este.

En el caso de cargar un archivo CSV, la función correspondiente (`read_csv()`) tiene varios parámetros opcionales, algunos de los más importantes son:

- `sep` y `delimiter`: permiten definir el separador de campos y el delimitador del archivo para que se cargue correctamente en un **DataFrame**. Funcionan de manera similar a los parámetros de la función del módulo *built-in csv* de Python.
- `header`: especifica la fila que se utilizará como cabecera (etiquetas de las columnas). Si no se indica, *pandas* la infiere automáticamente (por defecto, la primera fila).
- `usecols`: permite seleccionar qué columnas mantener en el **DataFrame**, ya sea mediante una lista de etiquetas o de índices posicionales.
- `names`: permite asignar nombres personalizados a las columnas. Suele usarse en combinación con `header=None` para evitar conflictos con la cabecera original del archivo.
- `index\_col`: especifica (por índice posicional o etiqueta) qué columna utilizar como **Index** del **DataFrame**.

Carga de un archivo Excel

pandas emplea librerías externas para poder cargar archivos de Excel, como *xlrd* para archivos `.xls` u *openpyxl* para archivos `.xlsx` y `.xlsm`, entre otros. Por tanto, es necesario instalar estas dependencias por separado si se desea trabajar con este tipo de archivos.

La función que se utiliza para cargar archivos de Excel es `read_excel()` . Funciona de manera similar a `read_csv()` y admite los mismos parámetros que mencionamos anteriormente para dicha función.

Además, posee el parámetro adicional `sheet\_name`, que indica la hoja específica que queremos importar. Este parámetro puede recibir el nombre de la hoja o su índice (basado en 0).

Filtrar y Ordenar

Nótese que tanto para filtrar como para ordenar, se podría hacer mediante iteraciones y comparaciones a lo largo del **DataFrame**. Sin embargo, esto resulta muy ineficiente y no aprovecha el potencial de las optimizaciones de *pandas*, como la paralelización. Por tanto, a continuación se revisarán algunas técnicas más eficientes para llevar a cabo dichas tareas.

Filtrado con máscaras booleanas

Para crear una máscara booleana, es necesario generar un **array** booleano que actúe como filtro para aplicar posteriormente al **DataFrame**.

Esta máscara se puede construir utilizando tanto los índices implícitos (posicionales) como los índices explícitos (etiquetas).

In [340…
```
df = pd.DataFrame([[0.1, 0.5, -0.3, 4],
                   [9.1 , -0.25, 1.3, 1.4],
                   [2.5 , 2.7, -9, 2.4],
                   [3, 6, 1, 5.5]],
                   index = ["Sergio", "Enrique", "Alba", "Maria"],
                   columns = ["a", "b", "c", "d"]
                  )
df
```

Out[340…

	a	b	c	d
Sergio	0.1	0.50	-0.3	4.0
Enrique	9.1	-0.25	1.3	1.4
Alba	2.5	2.70	-9.0	2.4
Maria	3.0	6.00	1.0	5.5

Por ejemplo, la siguiente máscara obtiene únicamente los valores estrictamente positivos de la columna `c`:

In [341…
```
mask = df["c"] > 0
mask
```

Out[341…
```
Sergio     False
Enrique     True
Alba       False
Maria       True
Name: c, dtype: bool
```

A continuación, podemos aplicar la máscara al **DataFrame** para eliminar las filas que no nos interesan, es decir, aquellas cuyos valores no sean estrictamente positivos:

In [342…
```
df[mask]
```

Out[342…

	a	b	c	d
Enrique	9.1	-0.25	1.3	1.4
Maria	3.0	6.00	1.0	5.5

También se puede emplear los atributos `loc` e `iloc` para declarar la máscara:

In [343…
```
mask = df.loc[:,"c"] > 0
mask
```

Out[343...
```
Sergio     False
Enrique     True
Alba       False
Maria       True
Name: c, dtype: bool
```

In [344...
```
df[mask]
```

Out[344...

	a	b	c	d
Enrique	9.1	-0.25	1.3	1.4
Maria	3.0	6.00	1.0	5.5

In [345...
```
mask = df.iloc[:,2] > 0
mask
```

Out[345...
```
Sergio     False
Enrique     True
Alba       False
Maria       True
Name: c, dtype: bool
```

In [346...
```
df[mask]
```

Out[346...

	a	b	c	d
Enrique	9.1	-0.25	1.3	1.4
Maria	3.0	6.00	1.0	5.5

Nótese que se puede adaptar el código para aplicar la máscara a las filas y no a las columnas de la siguiente forma:

In [347...
```
mask = df.iloc[2] > 0
mask
```

Out[347...
```
a     True
b     True
c    False
d     True
Name: Alba, dtype: bool
```

In [348...
```
df.loc[:, mask]
```

Out[348...

	a	b	d
Sergio	0.1	0.50	4.0
Enrique	9.1	-0.25	1.4
Alba	2.5	2.70	2.4
Maria	3.0	6.00	5.5

Ordenación

Podemos ordenar las filas del **DataFrame** tomando las etiquetas de las filas como referencia mediante la función `sort_index()`:

In [349…
```
df.sort_index()
```

Out[349…

	a	b	c	d
Alba	2.5	2.70	-9.0	2.4
Enrique	9.1	-0.25	1.3	1.4
Maria	3.0	6.00	1.0	5.5
Sergio	0.1	0.50	-0.3	4.0

Otra función para ordenar los datos en un **DataFrame** es `sort_values()`. Esta función permite reorganizar las filas o las columnas en función de los valores contenidos en una o varias columnas específicas.

El parámetro obligatorio `by` indica la columna (o columnas) que se usarán como referencia para realizar la ordenación.

Además, cuenta con un parámetro opcional llamado `axis` que define la dirección en la que se aplica el ordenamiento:

- Si `axis=0` (valor por defecto), se ordenan las *filas* según los valores de la(s) *columna(s)* indicada(s) en `by`.
- Si `axis=1`, se ordenan las *columnas* según los valores de la(s) *fila(s)* indicada(s) en `by`.

In [350…
```
df.sort_values("d")
```

Out[350…

	a	b	c	d
Enrique	9.1	-0.25	1.3	1.4
Alba	2.5	2.70	-9.0	2.4
Sergio	0.1	0.50	-0.3	4.0
Maria	3.0	6.00	1.0	5.5

In [351…
```
df.sort_values(by = "Sergio", axis = 1)
```

Out[351…

	c	a	b	d
Sergio	-0.3	0.1	0.50	4.0
Enrique	1.3	9.1	-0.25	1.4
Alba	-9.0	2.5	2.70	2.4
Maria	1.0	3.0	6.00	5.5

In [352…
```
df.sort_values(by = ["b", "d"])
```

Out[352...

	a	b	c	d
Enrique	9.1	-0.25	1.3	1.4
Sergio	0.1	0.50	-0.3	4.0
Alba	2.5	2.70	-9.0	2.4
Maria	3.0	6.00	1.0	5.5

In [353...

```
df.sort_values(by = ["Sergio", "Alba"], axis = 1)
```

Out[353...

	c	a	b	d
Sergio	-0.3	0.1	0.50	4.0
Enrique	1.3	9.1	-0.25	1.4
Alba	-9.0	2.5	2.70	2.4
Maria	1.0	3.0	6.00	5.5