02_Pandas

May 11, 2022

1 ¿Qué es el Data Science?

El Data Science es un campo interdisciplinario que involucra métodos científicos, procesos y sistemas para extraer conocimiento o un mejor entendimiento de datos en sus diferentes formas, ya sea estructurados o no estructurados, lo cual es una continuación de algunos campos de análisis de datos como la estadística, la minería de datos, el aprendizaje automático y la analítica predictiva.

El Data Science combina software, estadística, matemática, programación y visualización. Y su objetivo es extraer datos factibles de interpretarse e incluso crear nueva información. Las conclusiones que se obtienen permiten desarrollar productos demandados en el mercado o generar oportunidades de negocio de una empresa.

2 ¿Qué es pandas?

El paquete **pandas** es la herramienta más importante a disposición de los científicos y analistas de datos que trabajan en Python en la actualidad. Las poderosas herramientas de aprendizaje automático y visualización glamorosa pueden llamar toda la atención, pero pandas es la columna vertebral de la mayoría de los proyectos de datos.

En Computación y Ciencia de datos, pandas es una biblioteca de software escrita como extensión de Numpy para manipulación y análisis de datos para el lenguaje de programación Python. Esta librería ofrece dos de las estructuras más usadas en Data Science: la estructura Series y el DataFrame. En este curso veremos cómo crearlas, las herramientas básicas de uso y algunas de las funciones y métodos que nos permitirán extraer todo el potencial de ellas.

3 Las características de la biblioteca son:

- El tipo de datos DataFrame para manipulación de datos con indexación integrada. Tiene herramientas para leer y escribir datos entre estructuras de datos en memoria y formatos de archivos variados
- Permite la alineación de datos y manejo integrado de datos faltantes, la reestructuración y segmentación de conjuntos de datos, la segmentación vertical basada en etiquetas, indexación elegante, y segmentación horizontal de grandes conjuntos de datos, la inserción y eliminación de columnas en estructuras de datos.
- Puedes realizar cadenas de operaciones, dividir, aplicar y combinar sobre conjuntos de datos, la mezcla y unión de datos.
- Permite realizar indexación jerárquica de ejes para trabajar con datos de altas dimensiones en estructuras de datos de menor dimensión, la funcionalidad de series de tiempo: generación

de rangos de fechas y conversión de frecuencias, desplazamiento de ventanas estadísticas y de regresiones lineales, desplazamiento de fechas y retrasos.

4 Primeros pasos de Pandas

4.1 Instalar e importar

Pandas es un paquete fácil de instalar. Abra su programa de terminal (para usuarios de Mac) o línea de comandos (para usuarios de PC) e instálelo usando cualquiera de los siguientes comandos:

conda install pandas **o** pip install pandas

Dado que estamos trabajando sobre un cuaderno de Jupyter, puede ejecutar esta celda:

[ ]:
```
#! pip install pandas
#! conda install pandas
```

El ! al principio ejecuta las celdas como si estuvieran en una terminal.

Para importar pandas, generalmente lo importamos con un nombre más corto ya que se usa mucho:

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

5 Componentes principales de pandas: Series y DataFrames

Los dos componentes principales de los pandas son Series y DataFrame.

Una Series es esencialmente una columna y un DataFrame es una tabla multidimensional formada por una "colección de Series".

6 Series

Las series son estructuras unidimensionales conteniendo un array de datos (de cualquier tipo "soportado por NumPy") y un array de etiquetas que van asociadas a los datos, llamado índice (index en la literatura en inglés):

Clase pandas.Series (datos = Ninguno , índice = Ninguno , dtype = Ninguno , nombre = Ninguno , copia = Falso , fastpath = Falso)

Parámetros

** valor de tipo matriz de datos , iterable, dictado o escalar**

Contiene datos almacenados en Series. Si los datos son un dictado, se mantiene el orden de los argumentos.

** índice similar a una matriz o índice (1d) **

Los valores deben ser hash y tener la misma longitud que los datos . Se permiten valores de índice no únicos. El valor predeterminado será RangeIndex (0, 1, 2,..., n) si no se proporciona. Si los datos son similares a un dictado y el índice es Ninguno, las claves de los datos se utilizan como índice. Si el índice no es Ninguno, la Serie resultante se vuelve a indexar con los valores del índice.

** dtype str, numpy.dtype o ExtensionDtype, opcional**

Tipo de datos para la serie de salida. Si no se especifica, esto se deducirá de los datos . Consulte la guía del usuario para conocer más usos.

** nombre str, opcional**

El nombre para darle a la Serie.

** copy bool, predeterminado Falso**

Copie los datos de entrada.

[ ]:
```
import pandas as pd

d = {'a': 1, 'b': 2, 'c': 3}
ser = pd.Series(data=d, index=['a', 'b', 'c'])
ser
```

[ ]:
```
ventas1 = pd.Series([15,12,21]) # c/indice implicito
print(ventas1)
ventas = pd.Series([15,12,21], index = ["Ene", "Feb", "Mar"]) # c/indice␣
 ↪explicito
ventas
```

Los elementos de la serie pueden extraerse con el nombre de la serie y, entre corchetes, el índice (posición) del elemento:

[ ]:
```
ventas[0] #mostrar a traves del valor implicito (indice)
```

o con su etiqueta, si la tiene:

[ ]:
```
ventas["Ene"] #mostrar a traves del valor explicito (etiqueta)
```

Las etiquetas que forman el índice no necesitan ser diferentes. Pueden ser de cualquier tipo (numérico, textos, tuplas…) siempre que sea posible aplicar la función hash sobre ellas.

hash : https://www.interactivechaos.com/python/function/hash

Es de destacar que el lazo entre una etiqueta y un valor se mantendrá salvo que lo modifiquemos explícitamente. Esto quiere decir que filtrar una serie o eliminar un elemento de la serie, por ejemplo, no va a modificar las etiquetas asignadas a cada valor.

Otro comentario importante es al respecto de la inmutabilidad del índice de etiquetas: aun cuando es posible asignar a una serie un nuevo conjunto de etiquetas a través del atributo index, intentar modificar un único valor del index va a devolver un error.

Al igual que ocurre con el array NumPy, una serie pandas solo puede contener datos de un mismo tipo. En la imagen anterior puede apreciarse el índice a la izquierda (“Ene”, “Feb” y “Mar”) y los datos a la derecha (15, 12 y 21). El tipo de la serie, accesible a través del atributo dtype (Se muestra en la parte inferior: int64), coincide con el tipo de los datos que contiene:

[ ]:
```
ventas.dtype
```

Podemos acceder a los objetos que contienen los índices y los valores a través de los atributos **index** y **values** de la serie, respectivamente:

[ ]:
```
ventas.index
```

[ ]:
```
ventas.values
```

Puede apreciarse en el ejemplo que el índice es de tipo "objeto".

La serie tiene, además, un atributo name, atributo que también encontramos en el índice. Una vez los hemos fijado, se muestran junto con la estructura al imprimir la serie:

[ ]:
```
ventas.name = "Ventas 2020"
ventas.name
```

[ ]:
```
ventas
```

[ ]:
```
ventas.index.name = "Meses"
```

[ ]:
```
ventas
```

Obsérvese cómo, en este ultimo ejemplo, en la salida, tanto la serie como el índice se muestran con su nombre ("Ventas 2020" y "Meses", respectivamente).

El atributo **axes** nos da acceso a una lista con los ejes de la serie (solo contiene un elemento al tratarse de una estructura unidimensional):

[ ]:
```
ventas.axes
```

El atributo **shape** nos devuelve el tamaño de la serie:

[ ]:
```
ventas.shape
```

Para ver mas atributos de las Series se puede consultar la documentación de pandas: https://pandas.pydata.org/pandas-docs/stable/reference/series.html

7 DataFrames

Crear DataFrames directamente en Python y es bastante útil cuando se prueban nuevos métodos y funciones que se encuentran en los documentos de pandas.

Hay muchas formas de crear un DataFrame desde cero, pero una gran opción es usar un simple **dict**.

Digamos que tenemos un puesto de frutas que vende manzanas y naranjas. Queremos tener una columna para cada fruta y una fila para cada compra del cliente. Para organizar esto como un diccionario para pandas, podríamos hacer algo como:

8 class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)

Datos tabulares bidimensionales, de tamaño mutable y potencialmente heterogéneos.

La estructura de datos también contiene ejes etiquetados (filas y columnas). Las operaciones aritméticas se alinean en las etiquetas de fila y columna. Se puede considerar como un contenedor similar a un dict para los objetos de la serie. La estructura de datos primaria de los pandas.

Parámetros

data: matriz de datos (estructurada u homogénea), Iterable, dict o DataFrame

Dict puede contener series, matrices, constantes, clases de datos u objetos de tipo lista. Si los datos son un dictado, el orden de las columnas sigue el orden de inserción.

index: índice o como una matriz Índice que se utilizará para el fotograma resultante. Se establecerá de forma predeterminada en RangeIndex si no hay información de indexación como parte de los datos de entrada y no se proporciona un índice.

columns: índice o tipo matriz Etiquetas de columna que se utilizarán para el marco resultante. El valor predeterminado será RangeIndex (0, 1, 2,…, n) si no se proporcionan etiquetas de columna.

dtype: dtype, predeterminado Ninguno Tipo de datos a forzar. Solo se permite un solo tipo d. Si es Ninguno, infiera.

copy: bool, por defecto Falso Copie los datos de las entradas. Solo afecta la entrada DataFrame / 2d ndarray.

[ ]:
```
# Diccionario
datos = { 'manzanas' : [ 3 , 2 , 0 , 1 ], 'naranjas' : [ 0 , 3 , 7 , 2 ] }
```

Y luego páselo al constructor de Pandas DataFrame:

[ ]:
```
import pandas as pd

compras = pd.DataFrame( datos )
compras
```

8.0.1 ¿Cómo funcionó eso?

Cada elemento (clave, valor) en “datos” corresponde a una columna en el DataFrame resultante.

El índice de este DataFrame se nos dio en la creación como los números 0-3, pero también podríamos crear el nuestro cuando inicializamos el DataFrame.

Tengamos nombres de clientes como nuestro índice:

[ ]:
```
# Diccionario
datos = { 'manzanas' : [ 3 , 2 , 0 , 1 ], 'naranjas' : [ 0 , 3 , 7 , 2 ] }

compras = pd.DataFrame( datos , index = [ 'Luis' , 'Roberto' , 'Liliana' ,␣
↪'David' ])
```

```
compras
```

Las etiquetas de filas y de columnas -los índices- son accesibles a través de los atributos index y columns, respectivamente:

[ ]:
```
compras.index
```

[ ]:
```
compras.columns
```

La nomenclatura usada por pandas puede resultar un tanto confusa en lo que se refiere a los índices: tanto la estructura que contiene las etiquetas de filas como la que contiene las etiquetas de columnas son objetos de tipo Index ("índice", en español), pero, como se ha comentado, el índice de filas se denomina también index (aunque en minúsculas), y el de columna, columns.

Además, el nombre de "indice" se aplica normalmente a la referencia de un dato en una estructura según su posición. Por ejemplo, en la lista m = ["a", "b"], el índice del primer elemento es el número o valor que, añadido entre corchetes tras el nombre de la lista, nos permite acceder al elemento. Así, el índice del elemento "a" en la lista mencionada es 0, y el índice del elemento "b" es 1, lo que no es del todo coherente con el concepto de "índice" de una estructura pandas cuando lo especificamos explícitamente.

Para evitar esta confusión, hablaremos normalmente de "índices" (en plural) para referirnos a estas dos estructuras (de filas y columnas), de "índice" (en singular) para referirnos al índice de etiquetas del eje vertical, y de "índice de columnas" y de "índice de filas" siempre que sea necesario remarcar a cuál estamos refiriéndonos.

[ ]:
```
compras.index
```

[ ]:
```
compras.columns
```

El eje 0 es el correspondiente al índice de filas (eje vertical) y el eje 1 al índice de columnas (eje horizontal). Como puede verse en el ejemplo anterior ejemplo, ambos índices son de tipo "objeto" (ya se ha comentado que, concretamente, son objetos de tipo Index).

El atributo axes devuelve una lista con los ejes de la estructura (dos, al tratarse de una estructura bidimensional):

[ ]:
```
compras.axes
```

Al igual que ocurría con las series, los índices de filas y columnas son inmutables. Esto significa que, aunque podemos asignar un nuevo conjunto de datos (etiquetas) a ambas estructuras (index o columns), intentar modificar un único valor devolverá un error.

Tanto el índice de filas como el de columnas poseen el atributo name. Una vez fijado, se muestra al imprimir la estructura:

[ ]:
```
#         manzanas         naranjas
#Juno         3         0
#Robert         2         3
#Lily         0         7
```

```
#David        1        2

# Adiciona un nombre para el indice

compras.index.name = "Clientes"
```

[ ]:
```
#         manzanas         naranjas
#Juno         3         0
#Robert         2         3
#Lily         0         7
#David         1         2

# Adiciona un nombre para manzanas y naranjas

compras.columns.name = "Frutas"
```

[ ]:
```
compras
```

De forma semejante a como ocurría con las series, el atributo values de un dataframe nos permite acceder a los valores del dataframe, con formato array NumPy 2d:

[ ]:
```
compras.values
```

Este array tendrá un tipo u otro en función de los tipos de las columnas del dataframe, acomodándose de forma que englobe a todos ellos.

Y un dataframe también tiene un atributo shape que nos informa de su dimensionalidad y del número de elementos en cada dimensión. En el siguiente ejemplo Podemos ver que el dataframe compras tiene 4 filas y 2 columnas:

[ ]:
```
compras.shape
```

Para ver más información sobre DataFrame se puede consultar la documentación de pandas: https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

9 Creación de series

El constructor para la creación de una serie pandas es pandas.Series. Este constructor acepta tres parámetros principales:

- data: estructura de datos tipo array, iterable, diccionario o valor escalar que contendrá los valores a introducir en la serie.
- index: estructura tipo array con la misma longitud que los datos. Si este argumento no se añade al crear la serie, se agregará un índice por defecto formado por números enteros desde 0 hasta n-1, siendo n el número de datos.
- dtype: tipo de datos para la serie. Si no se especifica, se inferirá a partir de los datos.

Los valores del índice, como ya se ha comentado anteriormente, no tienen que ser necesariamente distintos aunque ciertas operaciones pueden generar un error si no soportan la posibilidad de tener

índices duplicados.

9.0.1 Utilizando una lista

```
[ ]: import pandas as pd
     s = pd.Series([7,5,3])
     s
```

Al no haberse especificado un índice, se asigna uno automáticamente con los valores 0, 1 y 2.

Si repetimos esta instrucción especificando un índice:

```
[ ]: s = pd.Series([7,5,3], index = ["Ene", "Feb", "Mar"])
     s
```

Aquí vemos cómo el índice por defecto ha sido sustituido por el indicado. En este caso, la longitud del índice deberá coincidir con el número de elementos de la lista.

Los mismos comentarios podrían hacerse si, en lugar de una lista, hubiésemos partido de un array NumPy para crear la serie.

9.0.2 Utilizando un diccionario

```
[ ]: d = {"Ene":7, "Feb":5, "Mar":3 }
     s = pd.Series(d)
     s
```

Aquí vemos cómo el constructor utiliza las claves como etiquetas del índice, y los valores del diccionario como valores de la serie.

Si incluimos el índice explícitamente en el constructor, los valores en la serie se tomarán en el orden en el que estén en el índice explícito. Además, si en éste hay valores que no pertenecen al conjunto de claves del diccionario, se añaden a la serie con un valor NaN:

```
[ ]: d = {"Ene":7, "Feb":5, "Mar":3 }
     s = pd.Series(d, index = ["Abr", "Mar", "Feb", "Ene"],dtype=int) #NaN
     s
```

En este ejemplo, hemos creado la serie especificando el índice que hemos formado dando la vuelta a las claves del diccionario (“Mar”, “Feb” y “Ene”) y hemos añadido a la lista de etiquetas el valor “Abr”, que no pertenece al conjunto de claves del diccionario. Se ha añadido a la serie, pero se le ha asignado el valor NaN. Es precisamente la presencia de este valor lo que modifica el tipo de la serie a float.

9.0.3 Utilizando un escalar

Si los datos se reducen a un escalar (no a una lista con un único elemento, sino a un sencillo escalar como 7 o 15.4) será necesario añadir el índice explícitamente. El número de elementos de la serie coincidirá con el número de elementos del índice, y el escalar será asignado como valor a todos ellos:

```
[ ]: s = pd.Series(7, index = ["Ene", "Feb", "Mar"])
     s
```

10 Creación de dataframes

El constructor de dataframes es "pandas.DataFrame". Acepta cuatro parámetros principales:

- data: estructura de datos ndarray (array NumPy), diccionario u otro dataframe
- index: índice a aplicar a las filas. Si no se especifica, se asignará uno por defecto formado por números enteros entre 0 y n-1, siendo n el número de filas del dataframe.
- columns: etiquetas a aplicar a las columnas. Al igual que ocurre con el índice de filas, si no se añade se asignará uno automático formado por números enteros entre 0 y n-1, siendo n el número de columnas.
- dtype: tipo a aplicar a los datos. Solo se permite uno. Si no se especifica, se infiere el tipo de cada columna a partir de los datos que contengan.

Los valores de los índices de filas y columnas no tienen por qué ser necesariamente distintos.

Veamos algunas de las estructuras a partir de las que es posible construir un dataframe:

10.0.1 Utilizando un diccionario

```
[ ]: import pandas as pd
     elementos = {
         "Numero atómico":[1, 6, 47, 88],
         "Masa atómica":[1.008, 12.011, 107.87, 226],
         "Familia":["No metal", "No metal", "Metal", "Metal"]
     }
     elementos
```

Y creamos el dataframe con él como primer argumento:

```
[ ]: tabla_periodica = pd.DataFrame(elementos)
     tabla_periodica
```

El dataframe se ha creado situando las claves del diccionario como etiquetas de columnas y las listas asociadas a cada clave como columnas del dataframe. Al no haber especificado un índice de filas, éste ha tomado valores por defecto (0, 1, 2 y 3).

A continuación repetimos la misma operación especificando las etiquetas tanto para filas como para columnas, utilizando los parámetros index y columns, respectivamente:

```
[ ]: tabla_periodica = pd.DataFrame(elementos,
                                    index = ["H", "C", "Ag", "Ra"],
                                    columns = ["Familia", "Numero atómico", "Masa␣
      ↪atómica"]
     )
     tabla_periodica
```

Recordemos que con el parámetro columns podemos especificar el orden en el que se mostrarán las columnas o incluso filtrar éstas (no incluyendo todas las etiquetas presentes en el diccionario como claves), pero no cambiar sus nombres. De hecho, ya se ha comentado que si alguna de las etiquetas incluidas en dicho argumento no apareciese en el conjunto de claves del diccionario, se crearía una columna con dicho nombre pero con todos sus valores fijados a NaN.

Si, en lugar de listas de datos como valores del diccionario, hubiesen sido arrays NumPy o series, el procedimiento habría sido exactamente el mismo.

10.0.2 Utilizando un array Numpy

En el caso de partir de un array NumPy, si no se especifican las etiquetas de filas y columnas, se asignan las etiquetas por defecto:

[ ]:
```
import numpy as np

unidades_Datos = np.array([[2, 5, 3, 2],
                           [4, 6, 7, 2],
                           [3, 2, 4, 1]])
unidades_Datos
```

[ ]:
```
unidades = pd.DataFrame(unidades_Datos)
unidades
```

Las filas del array NumPy siguen siendo interpretadas como filas del dataframe.

Si especificamos las etiquetas de filas y columnas, el resultado es diferente:

[ ]:
```
unidades = pd.DataFrame(unidades_Datos, index = [2015, 2016, 2017], columns =␣
↪["Ag", "Au", "Cu", "Pt"])
unidades
```

10.0.3 Utilizando diferentes diccionarios

También podemos partir de un conjunto de diccionarios, cada uno definiendo el contenido de lo que será una fila del dataframe:

[ ]:
```
unidades_2015 = {"Ag":2, "Au":5, "Cu":3, "Pt":2}
unidades_2016 = {"Ag":4, "Au":6, "Cu":7, "Pt":2}
unidades_2017 = {"Ag":3, "Au":2, "Cu":4, "Pt":1}

unidades = pd.DataFrame([unidades_2015, unidades_2016, unidades_2017],
                        index = [2015, 2016, 2017])
unidades
```

Los diccionarios deberán compartir el mismo conjunto de claves que se interpretarán como etiquetas de columnas. Si las etiquetas no coinciden, se crearán todas las columnas pero se asignarán NaN a los valores desconocidos:

[ ]:
```
unidades_2015 = {"Ag":2, "Au":5, "Cu":3, "Pt":2}
unidades_2016 = {"Ag":4, "Au":6, "Cu":7, "Pt":2}
unidades_2017 = {"Ag":3, "Pb":2, "Cu":4, "Pt":1}

unidades = pd.DataFrame([unidades_2015, unidades_2016, unidades_2017],
                        index = [2015, 2016, 2017])
unidades
```

En este ejemplo, el año 2017 tiene una clave, Pb, que no existe en los otros dos diccionarios. Y este mismo año carece de la clave Au que sí se encuentra en los otros dos. Vemos cómo los datos no coincidentes se han rellenado con NaN.

11 Otro metodos de contrucción

- pandas.DataFrame.from_dict, crea un dataframe a partir de un diccionario de diccionarios o de secuencias tipo array: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.from_dict.html
- pandas.DataFrame.from_records, que parte de una lista de tuplas o de arrays NumPy con un tipo estructurado: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.from_records.html

12 Inspección de la información

Normalmente, una vez hemos cargado un bloque de datos en una serie o un dataframe, lo primero que haremos será inspeccionarlo para confirmar que los datos cargados son los esperados y que la lectura se ha realizado correctamente. Para esto tenemos los métodos **head**, **tail** y **sample**, con un comportamiento semejante en series y dataframes, que nos muestran un subconjunto de los datos cargados. Además, los métodos **describe** e **info** nos proporcionan información adicional sobre los datos. Veamos estos métodos por separado.

13 El método head

Este método, pandas.Series.head para series y pandas.DataFrame.head para dataframes, devuelve los primeros elementos de la estructura (los primeros valores en el caso de una serie y las primeras filas en el caso de un dataframe). Por defecto, se trata de los 5 primeros elementos, pero podemos especificar el número que deseamos como argumento de la función. Por ejemplo, partamos de las siguientes estructuras:

[ ]:
```
import pandas as pd
entradas = pd.Series([11, 18, 12, 16, 9, 16, 22, 28, 31, 29, 30, 12],
                     index = ["ene", "feb", "mar", "abr", "may", "jun", "jul",␣
 ↪"ago",
                              "sep", "oct", "nov", "dic"])
entradas
```

[ ]:
```
salidas = pd.Series([9, 26, 18, 15, 6, 22, 19, 25, 34, 22, 21, 14],
                    index = ["ene", "feb", "mar", "abr", "may", "jun", "jul",␣
 ↪"ago",
                             "sep", "oct", "nov", "dic"])
salidas
```

[ ]:
```
almacen = pd.DataFrame({"entradas": entradas, "salidas": salidas})
almacen["neto"] = almacen.entradas - almacen.salidas
print(almacen)
print(almacen.shape)
```

En este ejemplo estamos mostrando todos los elementos de la estructura pues son apenas 12. En un caso real podemos estar hablando de miles o de millones.

Ahora, para mostrar apenas los primeros elementos de la estructura, ejecutamos el método head:

[ ]:
```
entradas.head()
```

[ ]:
```
almacen.head()
```

14 El método tail

Los métodos pandas.Series.tail (para series) y pandas.DataFrame.tail (para dataframes) son semejantes a los anteriores, pero muestran los últimos elementos de la estructura. Si no indicamos otra cosa como argumento, serán los 5 últimos elementos los que se muestren:

[ ]:
```
entradas.tail()
```

[ ]:
```
almacen.tail()
```

15 El método sample

Sin embargo, es frecuente que los datos que hayamos leído estén ordenados según algún criterio, y que el bloque de datos mostrado por los métodos head o tail estén formados por datos muy parecidos. Y en ocasiones nos puede convenir ver datos aleatorios de nuestra estructura. Para esto podemos utilizar los métodos "pandas.Series.sample" para series y "pandas.DataFrame.sample" para dataframes. Al contrario que head o tail, el número de elementos devueltos por defecto es uno, por lo que, si deseamos extraer una muestra mayor, tendremos que indicarlo como primer argumento:

[ ]:
```
entradas.sample()
```

[ ]:
```
almacen.sample(5)
```

16 El método describe

El método describe devuelve información estadística de los datos del dataframe o de la serie (de hecho, este método devuelve un dataframe). Esta información incluye el número de muestras, el valor medio, la desviación estándar, el valor mínimo, máximo, la mediana y los valores correspondientes a los percentiles 25% y 75%.

16.1 No Función Descripción

1 recuento () Número de observaciones no nulas

2 suma () Suma de valores

3 media () Media de valores

4 mediana () Mediana de valores

5 modo () Modo de valores

6 std () Desviación estándar de los valores

7 min () Valor mínimo

8 max () Valor máximo

9 abs () Valor absoluto

10 prod () Producto de valores

11 cumsum () Suma acumulada

12 cumprod () Producto acumulativo

Siguiendo con el ejemplo visto en la sección anterior:

[ ]:
```
import pandas as pd
entradas = pd.Series([11, 18, 12, 16, 9, 16, 22, 28, 31, 29, 30, 12],
                     index = ["ene", "feb", "mar", "abr", "may", "jun", "jul",␣
 ↪"ago",
                              "sep", "oct", "nov", "dic"])
salidas = pd.Series([9, 26, 18, 15, 6, 22, 19, 25, 34, 22, 21, 14],
                     index = ["ene", "feb", "mar", "abr", "may", "jun", "jul",␣
 ↪"ago",
                              "sep", "oct", "nov", "dic"])
almacen = pd.DataFrame({"entradas": entradas, "salidas": salidas})
almacen["neto"] = almacen.entradas - almacen.salidas

print(almacen.describe())
```

El método acepta el parámetro percentiles conteniendo una lista (o semejante) de los percentiles a mostrar. También acepta los parámetros include y exclude para especificar los tipos de las características a incluir o excluir del resultado.

17 El método info

El método info muestra un resumen de un dataframe, incluyendo información sobre el tipo de los índices de filas y columnas, los valores no nulos y la memoria usada:

[ ]:
```
almacen.info() #NaN
```

Solo los dataframes tienen implementado este método.

18 El método value_counts

Un método de las series pandas extremadamente útil es pandas.Series.value_counts. Este método devuelve una estructura conteniendo los valores presentes en la serie y el número de ocurrencias de cada uno. Estos valores se muestran en orden decreciente:

[ ]:
```
import numpy as np
s = pd.Series([3, 1, 2, 1, 1, 4, 1, 2, np.nan])
print(s)
s.value_counts()
```

Como puede apreciarse, por defecto no se incluyen los valores nulos. Este comportamiento puede modificarse haciendo uso del parámetro **dropna**:

[ ]:
```
s.value_counts(dropna = False)
```

En lugar de devolver los valores distintos y el número de ocurrencias, este método también puede agrupar los datos en "bins" y devolver una lista de bins (indicando sus márgenes) con el número de valores en cada uno de ellos. Por ejemplo, si quisiéramos agrupar los valores de la serie anterior en dos bins podríamos hacerlo de la siguiente forma:

[ ]:
```
s.value_counts(bins = 6)
```

Vemos que se han creados los dos bins, el primero conteniendo los valores entre 0.996 y 2.5 (intervalo abierto por la izquierda y cerrado por la derecha), bin en el que hay 6 valores, y el segundo conteniendo los valores entre 2.5 y 4 (intervalo también abierto por la izquierda y cerrado por la derecha), bin en el que hay 2 valores.

———————————- Selección de información ———————————-

- – Selección de datos en Series

Ya se ha comentado que una serie pandas consta de un array de datos y un array de etiquetas (el índice o index). Si al crear la serie no se ha especificado el índice, ya sabemos que se asignará uno implícito por defecto:

[ ]:
```
import pandas as pd

s = pd.Series([10, 20, 30, 40])
s
```

Podemos seleccionar los valores haciendo referencia al índice asignado con la misma notación que en un diccionario (la llamada "notación corchetes" o "square bracket notation"):

[ ]:
```
print(s[0])
print(s[2])
```

Usando esta sintaxis, si no se ha especificado un índice explícito, los índices negativos no están permitidos.

Si se asignan índices de forma explícita:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
s
```

podemos seleccionar los elementos usando el índice explícito o el implícito:

[ ]:
```
print(s["a"],s[0])
```

[ ]:
```
print(s["d"],s[3])
```

Con esta sintaxis, sí está permitido hacer uso de índices negativos para referirnos a los elementos desde el final de la estructura.

Si los índices asignados son números enteros (al igual que las etiquetas del índice implícito), el índice implícito queda desactivado:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = [3, 2, 1, 0])
s
```

[ ]:
```
s[3]
```

19 Uso de rangos

Es posible seleccionar rangos de valores. De esta forma, si usamos un rango numérico en una serie en la que hemos definido un índice explícito:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
s
```

[ ]:
```
s[1:3]
```

observamos que el rango se interpreta como haciendo referencia al índice implícito, y se incluyen los valores desde el primer índice incluido, hasta el último sin incluir.

Si no se incluye alguno de los límites, el comportamiento es el estándar en Python (si no se incluye el primer valor, se consideran todos los elementos desde el principio, y si no se incluye el último valor, se consideran todos los elementos hasta el final):

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
s[1:]
```

[ ]:
```
s[:3]
```

Si se utilizan los índices explícitos en el rango, el comportamiento es ligeramente diferente:

[ ]:
```
s["a":"c"]
```

[ ]:
```
s[:"c"]
```

[ ]:
```
s["b":]
```

Una posible fuente de confusión viene derivada del hecho de que, usando rangos, es posible hacer referencia tanto a las etiquetas como a los índices numéricos: si utilizamos etiquetas, hacemos referencia a las etiquetas (por supuesto):

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
```

[ ]:
```
s["b":"d"]
```

y, por tanto, si utilizamos números, hacemos referencia a los índices numéricos (¿por supuesto...?):

[ ]:
```
s[1:3]
```

¿Y qué ocurre si nuestras etiquetas son números? Pues que siempre que usemos rangos con números estaremos haciendo referencia a los índices numéricos: no es posible hacer referencia a las etiquetas:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = [3, 2, 1, 0])
s
```

[ ]:
```
s[1:3]
```

Sin embargo, algo como

[ ]:
```
s[1]
```

devolverá el valor cuya etiqueta es 1 (si existe):

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = [3, 2, 1, 0])
s
```

o el valor cuya posición es 1 si dicha etiqueta no existe y el índice explícito no es numérico:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
print(s)
s[1]
```

Es debido a esto que existen los métodos loc e iloc que veremos poco más adelante. Dichos métodos hacen una referencia explícita a etiquetas o posiciones, respectivamente, eliminando cualquier duda al respecto de su interpretación.

Al igual que con los array NumPy, es posible indicar, no un elemento simple ni un rango, sino una lista de valores. Por ejemplo:

```
#Serie
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
print(s)
s[[3, 1]]
```

En este ejemplo, la lista contiene los números 3 y 1, y son los valores correspondientes a estos índices -y en el orden especificado- los devueltos por la instrucción.

El resultado devuelto sigue siendo una serie pandas:

```
type(s[[3, 1]])
```

Con esta notación, en el caso de que la serie tenga un índice explícito numérico, los valores de la lista se interpretan como haciendo referencia al índice explícito.

También se puede utilizar el metodo **get**

```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
print(s)
s.get(1)
```

```
s.get("b")
```

20 Método loc

El método pandas.Series.loc permite seleccionar un grupo de elementos por etiquetas.

Como argumento de este método puede utilizarse una única etiqueta:

```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "b"])
s
```

```
s.loc["b"]
```

En este caso el argumento se interpreta siempre como etiqueta del índice, nunca como posición en dicho índice aun cuando se pase un número entero que no pertenece al conjunto de etiquetas y pueda representar una posición válida:

```
s.loc[0] #Error
```

También podemos pasar al método una lista de etiquetas, en cuyo caso se extraen los valores correspondientes a dichas etiquetas y en el orden en el que se incluyen en la lista:

```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
print(s)
s.loc[["d","a"]]
```

Otra opción es pasar al método un rango:

[ ]:
```
s["b":"d"]
```

En este caso es importante recalcar que, tal y como se ve en la imagen anterior, el método va a devolver todos los elementos entre los límites indicados ambos incluidos.

21 Método iloc

El método pandas.Series.iloc permite extraer datos de la serie a partir de los índices implícitos que éstos tienen asignados.

La opción más simple es utilizar como argumento un simple número entero (el primer elemento de la serie recibe el índice cero):

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
s
```

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[1]
```

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[0]
```

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[3]
```

Si el número es negativo, hace referencia al final de la serie (en este caso, el último elemento recibe el índice -1) -y esto tanto si se ha especificado un índice explícito como si no-:

[ ]:
```
#a    10
#b    20
#c    30
```

```
#d    40
#dtype: int64

s.iloc[-1]
```

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[-4]
```

Una segunda opción es pasar como argumento una lista o array de números, en cuyo caso se devuelven los elementos que ocupan dichas posiciones en el orden indicado en la lista o array:

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[[2, 0]]
```

También podemos incluir en esta lista números negativos, con la funcionalidad ya comentada:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[[-2, 0]]
```

Una tercera opción es usar como argumento un rango de números:

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[1:3]
```

Como vemos en el ejemplo anterior, si el rango tiene la forma a:b, se incluyen todos los elementos desde aquel cuyo índice es a (incluido) hasta el que tiene el índice b (sin incluir).

Si no se especifica el primer valor, se consideran todos los elementos desde el principio de la serie:

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[:3]
```

Y, si no se especifica el segundo valor, se consideran todos los elementos hasta el final de la serie:

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[2:]
```

También pueden usarse valores negativos para indicar el comienzo y/o el final del rango:

[ ]:
```
#a    10
#b    20
#c    30
#d    40
#dtype: int64

s.iloc[1:-1]
```

22 Uso de arrays booleanos

Una muy interesante opción para seleccionar elementos de una serie pandas es usar arrays booleanos. Por ejemplo, partimos de la siguiente serie:

[ ]:
```
s = pd.Series([5, 2, -3, 7, 8, 4])
s
```

Podemos seleccionar un conjunto de valores de la misma haciendo referencia al nombre de la serie y, entre los corchetes, una lista o array de booleanos (también puede ser una serie de booleanos, como veremos un poco más adelante):

[ ]:
```
#0    5
#1    2
#2   -3
#3    7
#4    8
#5    4
#dtype: int64
```

```
s[[True, False, False, True, True, False]]
```

En este caso hemos seleccionado los elementos cuyos índices son 0, 3 y 4, que son los índices que ocupan los booleanos True en la lista de booleanos usada (lista cuya longitud deberá ser igual a la longitud de la serie pues, de no ser así, se devuelve un error).

Esta lista o array de booleanos no tiene porqué ser especificada de forma explícita, puede ser el resultado de una expresión:

[ ]:
```
#0    5
#1    2
#2   -3
#3    7
#4    8
#5    4
#dtype: int64

print(type(s > 2))
s > 2
```

Aquí, hemos usado la expresión s > 2 para generar una serie pandas de booleanos, serie en la que los valores toman el valor True cuando el valor con el mismo índice de s toma un valor mayor estricto que 2.

Podemos entonces usar este resultado para extraer valores de la serie s (valores que serán aquellos mayores que 2):

[ ]:
```
import pandas as pd
s = pd.Series([5, 2, -3, 7, 8, 4])
s[s>2]
```

Este mismo enfoque puede ser usado con los métodos pandas.Series.loc y pandas.Series.iloc ya vistos en las secciones anteriores con algún matiz adicional:

El método loc puede ser usado tanto con un array explícito de booleanos:

[ ]:
```
s.loc[[True, False, False, True, True, True]]
```

como con una expresión que genera, por ejemplo, una serie pandas de booleanos:

[ ]:
```
s.loc[s>2]
```

Sin embargo, el método iloc tiene un comportamiento ligeramente diferente. Puede ser usados con arrays explícitos de booleanos:

[ ]:
```
s.iloc[[True, False, False, True, True, True]]
```

pero el uso de expresiones que generen una serie pandas de booleanos devuelve un error:

[ ]:
```
s.iloc[s>2]
```

Si el objeto que está generando la estructura de booleanos (s, en s > 2) fuese un array NumPy en lugar de tratarse de una serie pandas, sí sería posible usar el método .iloc. De esta forma, la expresión s > 2 genera, como hemos visto, una serie pandas, pero podemos extraer los valores con el atributo values, que genera un array numpy:

[ ]:
```
type((s>2).values)
```

[ ]:
```
(s>2).values
```

Si usamos esta expresión para realizar la selección en la serie original s, el resultado es ahora el correcto:

[ ]:
```
s.iloc[(s>2).values]
```

Es por ello que pandas recomienda usar el método loc cuando trabajemos con selección basada en booleanos.

23 Selección aleatoria

También podemos realizar una selección aleatoria a partir de una serie. El método pandas.Series.sample permite especificar o bien el número de elementos a extraer o bien la fracción del número total de elementos a extraer (parámetros **n** y **frac**, respectivamente), pudiendo especificar si la extracción se realiza con reemplazo o no (parámetro **replace**), los pesos a aplicar a cada elemento para realizar una extracción aleatoria ponderada (parámetro **weights**), y una semilla para el generador de números aleatorios que asegure la reproducibilidad de la extracción (parámetro **random_state**). Por ejemplo:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "s"])
```

[ ]:
```
s.sample(2, random_state = 18)
```

Hemos extraído 3 elementos, por defecto sin reemplazo, aplicando el valor 18 como semilla del generador de números aleatorios.

[ ]:
```
s.sample(frac = 0.6, random_state = 18)
```

En este otro ejemplo hemos extraído el 60% de los valores de la serie original haciendo uso del parámetro frac.

Si no hay reemplazo, el número máximo de elementos que podemos extraer coincide con la longitud de la serie. Pero si la extracción la realizamos con reemplazo, podemos especificar cualquier número de elementos:

[ ]:
```
s.sample(10, random_state = 18, replace = True)
```

24 Método pop

El método pandas.Series.pop extrae y elimina un elemento de una serie cuyo índice se indica como argumento ...devolviendo un error en caso de que no exista::

[ ]:
```
s = pd.Series([1, 2, 3, 4])
s
```

[ ]:
```
s.pop(1)
```

[ ]:
```
s
```

[ ]:
```
s = pd.Series([1, 2, 3, 4])
try:
    s.pop(2)
    print(s)
except:
    print("Error")
```

Si la serie tiene un índice explícito, el argumento de pop hará referencia a este índice:

[ ]:
```
s = pd.Series([10, 20, 30, 40], index = ["a", "b", "c", "d"])
s
```

[ ]:
```
s.pop("a")
```

[ ]:
```
s
```

y no al implícito, lo que devolvería un error:

[ ]:
```
try:
    s.pop()
except:
    print("Error")
```

25 - - Selección de datos en DataFrames

Desde un punto de vista semántico, un dataframe puede ser considerado semejante a un diccionario de series, en el que las claves son los nombres de las columnas y los valores, las columnas (que son series pandas). En este ejemplo:

[ ]:
```
ventas = pd.DataFrame({
    "Entradas":[41, 32, 56, 18],
    "Salidas":[17, 54, 6, 78],
    "Valoración": [66, 54, 49, 66],
    "Límite": ["No", "Si", "No", "No"],
    "Cambio": [1.43, 1.16, -0.67, 0.77]},
    index = ["Ene", "Feb", "Mar", "Abr"])
```

```
ventas
```

Podemos utilizar la sintaxis de los diccionarios para seleccionar la columna Entradas. Puede verse en el ejemplo siguiente cómo dicha columna es extraída con tipo de serie pandas:

[ ]:
```
print(type(ventas["Entradas"]))
```

[ ]:
```
ventas["Entradas"]
```

Esto significa que podemos realizar una selección en dicho resultado para, por ejemplo, extraer el valor correspondiente a febrero:

[ ]:
```
ventas["Entradas"]["Feb"]
```

Sin embargo, la más que razonable opción de eliminar los corchetes que separan ambos índices y sustituirlos por una coma no funciona:

[ ]:
```
ventas["Entradas","Feb"] #Error
```

Si, una vez seleccionada una columna, le asignamos una lista o array (o serie) de valores de la misma longitud, estamos modificando dicha columna del dataframe:

[ ]:
```
print(ventas.Entradas)
ventas["Entradas"] = [33, 25, 40, 12]
ventas
```

Si asignamos un único valor escalar, este se propaga por toda la columna:

[ ]:
```
ventas["Salidas"] = 1
ventas
```

Si estuviésemos asignando un array cuya longitud no coincidiese con la de la columna (y no estuviésemos asignando un escalar), obtendríamos un error.

Si asignamos una serie pandas se consideran los índices del dataframe y de la serie, haciendo coincidir los valores cuyos índices sean los mismos en ambas estructuras (si dicha columna no existe, se crea). En el caso de que haya valores en la serie con índices que no se encuentren en el dataframe, se descartan. Y en el caso de que haya índices en el dataframe que no se encuentren en la serie, se asigna un valor NaN.

Así, en el siguiente ejemplo, estamos añadiendo una serie cuyos índices son “Ene”, “Mar”, “Abr” y “May”. Es decir, la serie no tiene un valor para el índice “Feb” que sí se encuentra en el dataframe (se asigna un NaN), e incluye el índice “May” que no se encuentra en el dataframe y se descarta:

[ ]:
```
ventas = pd.DataFrame({
    "Entradas":[41, 32, 56, 18],
    "Salidas":[17, 54, 6, 78],
    "Valoración": [66, 54, 49, 66],
    "Límite": ["No", "Si", "No", "No"],
    "Cambio": [1.43, 1.16, -0.67, 0.77]},
```

```
    index = ["Ene", "Feb", "Mar", "Abr"])
ventas

ventas["Pérdidas"] = pd.Series([5, 4, 6, 8], index = ["Ene", "Mar", "Abr",␣
↪"May"])
ventas
```

Los valores asignados pueden proceder del propio dataframe:

[ ]:
```
ventas["Ganancias"] = (ventas["Entradas"]*2) - (ventas["Valoración"]/10)
ventas
```

También podemos acceder a una columna con la llamada "notación punto":

[ ]:
```
ventas.Ganancias
```

Deberemos tener en cuenta que con esta notación no es posible crear nuevas columnas ni eliminarlas con la función del y que solo funcionará si el nombre de la columna no incluye espacios en blanco y no coincide con ninguna palabra reservada de Python.

25.0.1 Uso de rangos

El uso de un rango numérico entre los corchetes realiza una selección de filas, lo que puede parecer una cierta incoherencia:

[ ]:
```
#Crear un arreglo con indice, columnas y valores de 0 - 17
import numpy as np
import pandas as pd
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "d", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df[2:5]
```

El equipo de pandas lo justifica diciendo que esta sintaxis resulta extremadamente conveniente al tratarse de un tipo de selección frecuentemente usada. Esto es cierto, pero el hecho de que selecciones aparentemente semejantes (df[1,2], df[[1, 2]], df[1:3, 5], etc.) devuelvan un error no facilita su comprensión.

En todo caso, vemos en el ejemplo anterior que se devuelven las filas entre el primer valor del rango (incluido) y el último (sin incluir). También podríamos haber usado las etiquetas del índice:

[ ]:
```
df["b":"d"]
```

[ ]:
```
df[:2]
```

[ ]:
```
df[:"c"]
```

Si, al realizar la selección, situamos entre los corchetes una lista de etiquetas, estaremos seleccionando columnas en el orden en el que aparecen en la lista y con formato dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
print(type(df[["C", "A"]]))
df[["C","A"]]
```

También es posible extraer de forma segura una columna de un dataframe usando el método pandas.DataFrame.get. Éste extrae la columna indicada devolviendo un valor alternativo (por defecto None) si dicha columna no existe:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.get("A")
```

[ ]:
```
df.get("D")
```

26 El método LOC

Al igual que ocurre con las series, el método pandas.DataFrame.loc permite seleccionar un conjunto de filas y columnas por etiquetas. Este método acepta diferentes argumentos. Para probarlos, partamos del siguiente dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

26.0.1 Uso con etiqueta simple

El primer escenario lo encontramos cuando usamos este método indicando una única etiqueta. En este caso estamos seleccionando la fila cuya etiqueta se indique:

[ ]:
```
df.loc["c"]
```

El resultado es una serie pandas con las etiquetas de columnas del dataframe original como índice.

Es necesario mencionar que el argumento será siempre interpretado como etiqueta, aun cuando pueda estar representando un índice válido:

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = [1, 3, 0, 4],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.loc[0]
```

Por supuesto, si dicha etiqueta no existe, se devuelve un error (nuevamente, aun cuando la etiqueta sea un número que pueda estar representando un índice válido.)

Si pasamos a loc una lista de etiquetas, estaremos extrayendo las filas cuyas etiquetas se indican, y en el orden en el que aparezcan en la lista:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.loc[["c","a","e"]]
```

Al contrario de lo que ocurre cuando solo indicamos una etiqueta, el resultado es un dataframe. Y lo es aún cuando la lista contenga un único elemento:

[ ]:
```
print(type(df.loc[["c"]]))
df.loc[["c"]]
```

Otra opción es utilizar rangos limitados por etiquetas. De esta forma, si continuamos con el mismo ejemplo:

[ ]:
```
df.loc["b":"d"]
```

Obsérvese que la selección incluye todas las filas incluyendo las dos de los extremos del rango.

27 Extracción de filas y columnas

En los ejemplos vistos hasta ahora estamos extrayendo una o varias filas para todas las columnas. En posible, por supuesto, especificar qué filas y qué columnas exactas queremos extraer. Así, si utilizamos una única etiqueta para indicar la fila, y una única etiqueta para indicar la columna, separadas por una coma, estaremos extrayendo un único valor:

[ ]:
```
#        A        B        C
#a       0        1        2
#b       3        4        5
#c       6        7        8
#d       9        10       11
#e       12       13       14
#f       15       16       17
```

```
import numpy as np
import pandas as pd
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
print(df)

df.loc["a", "C"]
```

Podemos sustituir una de las dos etiquetas por el símbolo de dos puntos (:), lo que supondrá seleccionar todos los elementos de ese eje:

[ ]:
```
df.loc[: ,"A"]
```

Esto supone que, por ejemplo, las dos expresiones siguientes devuelven el mismo resultado:

[ ]:
```
df.loc["b"]
```

[ ]:
```
df.loc["b", :]
```

Los métodos vistos pueden combinarse. Podemos, por ejemplo, seleccionar la intersección de las filas e y c (en este orden) y la columna B:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
print(df)

df.loc[["e", "c"], "B"]
```

28 El método iloc

El método pandas.DataFrame.iloc permite realizar selecciones por posición. Tal y como cabría esperar, pueden utilizarse diferentes tipos de argumentos que determinan qué elementos se están extrayendo.

En este primer caso cuando hacemos uso un número entero estamos seleccionando la fila cuyo índice se indica:

[ ]:
```
df = pd.DataFrame(np.random.randint(0, 10, 18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.iloc[2]
```

El número indicado siempre será tratado como posición, y no como etiqueta:

[ ]:
```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = [3, 2, 1, 0],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.iloc[3]
```

Si el número es negativo, hace referencia al final del dataframe. Así, siguiendo con este último ejemplo:

[ ]:
```
df.iloc[-1]
```

Si utilizamos como argumento una lista o array de números, estamos extrayendo las filas cuyos índices son los elementos del mismo, y en el orden en el que aparecen en él:

[ ]:
```
df = pd.DataFrame(np.random.randint(0, 10, 18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.iloc[[3, 1]]
```

En el ejemplo anterior, estamos extrayendo las filas cuyos índices son 3 y 1, y extrayéndolas en este mismo orden.

Si alguno de los índices es negativo, hará referencia al final de la lista.

Si utilizamos un rango de números, como en el siguiente ejemplo en el que indicamos como argumento 2:4, estamos extrayendo las filas cuyos índices van de la primera cifra del rango incluida (2 en el ejemplo) hasta la última cifra sin incluir (4 en el ejemplo):

[ ]:
```
df = pd.DataFrame(np.random.randint(0, 10, 18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
print(df)

df.iloc[2:4]
```

Como suele ser habitual, si no se especifica el primer valor, se consideran las filas desde la primera. Y si no se especifica el último valor, se consideran las filas hasta la última (incluida):

[ ]:
```
df.iloc[:3]
```

[ ]:
```
df.iloc[4:]
```

También pueden usarse valores negativos para especificar el comienzo o el final del rango.

29 Extracción de filas y columnas

Si añadimos un segundo argumento, estaremos haciendo referencia al índice de columna:

[ ]:
```
df = pd.DataFrame(np.random.randint(0, 10, 18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
print(df)

df.iloc[3, 1] #[F,C]
```

En el ejemplo anterior estamos extrayendo el valor correspondiente a la fila cuyo índice es 3 y a la columna cuyo índice es 1.

30 Selección con índices y etiquetas simultáneamente

En ocasiones nos encontraremos con que resultaría de utilidad poder realizar selecciones mezclando etiquetas e índices, y los métodos vistos, loc e iloc, solo permiten el uso de etiquetas o de índices, respectivamente. Para poder mezclar ambos tipos de referencias podemos recurrir a los métodos pandas.Index.get_loc y pandas.Index.get_indexer, métodos asociados a los índices de un dataframe:

El primero, get_loc, devuelve el índice de la etiqueta que se adjunte como parámetro. El segundo, get_indexer, devuelve un array con los índices de las etiquetas que se adjunten en forma de lista como parámetro. Por ejemplo, partimos del siguiente dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

Si aplicamos los métodos comentados al índice de columnas haciendo referencia a etiquetas de columnas, obtenemos los siguientes resultados:

[ ]:
```
df.columns.get_loc("B")
```

[ ]:
```
df.columns.get_indexer(["A", "C"])
```

En el primer caso hemos pasado la etiqueta “B” y el método ha devuelto su índice (1). En el segundo caso hemos pasado una lista de etiquetas y hemos obtenido un array con sus índices.

Si ejecutamos estos métodos en el índice de filas:

[ ]:
```
df.index.get_loc("d")
```

[ ]:
```
df.index.get_indexer(["c", "e"])
```

obtenemos resultados semejantes.

Ahora que sabemos cómo convertir etiquetas en sus índices equivalentes, podemos seleccionar datos de un dataframe mezclando etiquetas e índices si convertimos las etiquetas y utilizamos el método iloc ya visto. Por ejemplo, si quisiéramos extraer del anterior dataframe el dato que ocupa la fila "c" y la columna de índice 2, podríamos conseguirlo del siguiente modo:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
print(df)
df.iloc[df.index.get_loc("c"), 2] #Obtener el dato la [fila "c", columna 2]␣
↪equivalente a df.iloc[2,2]
```

O si deseásemos obtener de las filas 5 y 3 (en este orden) los valores correspondientes a las columnas C y A (en este orden), podríamos hacerlo con la siguiente expresión:

[ ]:
```
print(df)
df.iloc[[5, 3], df.columns.get_indexer(["C", "A"])]
```

31 Uso de listas de booleanos

Otro método especialmente útil para la selección es el uso de listas de booleanos. Nuevamente puede parecer un tanto incoherente aunque, en este caso, su uso sí es extremadamente conveniente. Veamos por qué:

Si partimos del mismo dataframe usado en la sección anterior, podemos crear una lista de booleanos (que, por motivos puramente pedagógicos, asignamos a una variable, mask) y realizar la selección con ella entre los corchetes. Vemos a continuación que este método también selecciona filas del dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
mask = [True, False, True, False, False, True] #Muestra los indices que cumplen␣
↪con True
df[mask]
```

El vector de booleanos deberá tener la misma longitud que el índice de filas (es decir, un booleano por fila) y la selección devolverá aquellas filas para las que el elemento correspondiente del vector tome el valor True.

La verdadera potencia de este estilo de selección se pone de manifiesto cuando la máscara se genera a partir de los datos del propio dataframe. Por ejemplo, si queremos seleccionar las filas para las que el valor de la columna A sea mayor que 7:

[ ]:
```
df[df.A > 7] #Muestra los indices que cumplen con la condición
```

Este tipo de filtrados resultan muy frecuentes en entornos de análisis, de ahí que la posibilidad de realizarlos sin necesidad de recurrir a métodos adicionales (loc, iloc o get, por ejemplo) resulte tan conveniente.

Aun así, esta técnica también es compatible con los métodos loc e iloc, con algún matiz adicional:

Con loc podemos usar directamente una expresión de comparación como la vista:

[ ]:
```
df.loc[df.B > 6] #Muestra los indices que cumplen con la condición
```

Sin embargo, con iloc nos veremos obligados a extraer los valores del dataframe resultante de la comparación -tal y como ocurría con las series- pues, de otro modo, obtendremos un error:

[ ]:
```
df.iloc[(df.B > 6 ).values]
```

Evitamos problemas si, tal y como sugiere pandas, utilizamos siempre el método loc.

32 Selección aleatoria

Al igual que ocurre con las series, también los dataframes tienen un método que permite extraer elementos del mismo de forma aleatoria: pandas.DataFrame.sample. Este método permite especificar el número de elementos a extraer (o el porcentaje respecto del total, parámetros **n** y **frac**, respectivamente), si la extracción se realiza con reemplazo o no (parámetro **replace**), los pesos a aplicar a los elementos para realizar una extracción aleatoria ponderada (parámetro **weights**) y una semilla para el generador de números aleatorios que asegure la reproducibilidad de la extracción (parámetro **random_state**). También es posible indicar el eje a lo largo del cual se desea realizar la extracción (por defecto se extraen filas, correspondiente al eje 0).

Veamos un ejemplo. Si partimos del siguiente dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

podemos extraer 3 filas de forma aleatoria, sin reemplazo (opción por defecto) y fijando como semilla del generador de números aleatorios el número 18, de la siguiente forma:

[ ]:
```
df.sample(2, random_state = 18)
```

Si especificamos como eje el valor 1, estaremos extrayendo columnas:

[ ]:
```
df.sample(2, random_state = 18, axis = 1) #Columnas
```

Si hacemos uso del parámetro frac, podemos especificar el porcentaje de elementos a extraer:

[ ]:
```
df.sample(frac = 0.6, random_state = 18)
```

33 El método pop

Otra forma de extraer datos es la proporcionada por el método pandas.DataFrame.pop, que extrae y elimina una columna de un dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(15).reshape([3, 5]),
                  index = ["a", "b", "c"],
                  columns = ["A", "B", "C", "D", "E"])
df
```

[ ]:
```
columna = df.pop("B") #Elimina la columna "B"
columna
```

[ ]:
```
df
```

34 Resumen de tipos de selección

La cantidad de posibilidades que nos ofrece la librería pandas a la hora de extraer información de un dataframe puede resultar un tanto abrumadora. Veamos un resumen de las principales técnicas vistas y qué estamos extrayendo con cada una. En la siguiente lista e representa una etiqueta, i un índice y b un booleano:

No se incluyen en el listado anterior combinaciones de estos métodos ni los métodos para la selección usando índices y etiquetas simultáneamente.

Aunque, a primera vista, pueda parecer bastante confuso, es posible destacar algunas reglas básicas:

1. Cuando se usan los métodos loc o iloc, el primer argumento siempre hace referencia a filas y el segundo a columnas. Esto significa que si no se incluye el segundo argumento, siempre estaremos extrayendo filas.
2. El método get devuelve columnas.
3. Sin incluir los métodos loc, iloc y get, solo hay -en el listado anterior- dos formas de extraer columnas: usando como argumento una etiqueta y usando como argumento una lista de etiquetas.
4. Del punto anterior extraemos como corolario que cualquier otra notación va a devolver filas (el uso de rangos y el uso de listas de booleanos)

Y, de hecho, si de los cuatro puntos anteriores quitamos los dos primeros por obvios, nos quedan dos reglas muy simples:

1. El uso de una etiqueta o de una lista de etiquetas devuelve columnas
2. En otros casos se devuelven filas (rangos de números o de etiquetas, o listas de booleanos)

Y, al respecto del tipo de estructura devuelta (si es un escalar, una serie o un dataframe) también es posible identificar una regla: salvo el caso más obvio en el que estemos extrayendo un valor resultante de la intersección de una fila y una columna, si la nomenclatura que estamos usado permite extraer más de una fila o de una columna (aunque estemos extrayendo solo una en un momento dado) devolverá siempre dataframes y, en caso contrario, series.

Por ejemplo, el uso de rangos permite extraer más de una fila (o de una columna), de forma que su uso siempre devuelve un dataframe. Así, df[“e1”:“e2”] siempre devolverá un dataframe, aunque

en este ejemplo estemos usando el rango para extraer una única columna.

Otro ejemplo: si estamos usando la notación df.loc[:, "e"], estamos extrayendo una columna y solo una columna, y con esta notación (dos puntos para las filas y una etiqueta para columnas) nunca podríamos extraer más de una columna, de forma que el resultado de la extracción siempre será una serie.

35 Edición de series

Podemos modificar un valor de una serie usando la notación corchetes, y haciendo referencia a índices o a etiquetas:

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

[ ]:
```
s[0] = -1
```

[ ]:
```
s["b"] = -2
```

[ ]:
```
s
```

Podemos asignar un valor a un rango, definido éste por índices o por etiquetas, asignándose dicho valor a cada uno de los elementos involucrados en el rango:

[ ]:
```
s[1:3] = 0
s
```

[ ]:
```
s["b":"d"] = -2
s
```

Como ya hemos visto en más de una ocasión, si el rango está delimitado por números (haciendo referencia a la posición de los elementos), el último elemento del rango no se incluye. Por el contrario, si está delimitado por etiquetas, el último elemento sí se incluye.

Al rango podemos asignar también una lista de valores, aunque en este caso la lista deberá tener el mismo número de elementos que el rango en cuestión:

[ ]:
```
s[1:3] = [0, 1]
s
```

[ ]:
```
s = pd.Series([1, 2, 3, 4, 5], index = ["a", "b", "c", "d", "e"])
s
s["b":"d"] = [10, 11, 12]
s
```

Si asignamos un valor haciendo referencia a una etiqueta que no existe en el índice, se añade dicha etiqueta al índice y se le asigna el valor:

```
[ ]: s["f"] = 0 #Se añade "f"
     s
```

Esto solo funciona con etiquetas. Si utilizamos un índice y éste no existe en la serie, se devolverá un error.

Si usamos un rango con etiquetas y alguna de las etiquetas no existe, solo se asigna el valor a las existentes:

```
[ ]: s["f":"h"] = 99 #Solo cambia el valor de "f", "h" no existe
     s
```

Por último, también podemos usar en la selección una lista -tanto de índices como de etiquetas-, en cuyo caso ya sabemos que estamos seleccionando los valores indicados en el orden indicado. Por ejemplo, podemos usar la lista [“c”, “a”] para asignar a los elementos correspondientes los valores 1 y 2, respectivamente:

```
[ ]: s[["c", "d"]] = [100, 200] #Esto es una lista, no un indice
     s
```

Si utilizamos índices, el resultado es semejante:

```
[ ]: s[[1, 0]] = [2000, 3000] #Esto es una lista, no un indice
     s
```

36 Eliminación series método pop y drop

El método pop ya ha sido presentado por eso se presentará el método pandas.Series.drop el cual devuelve una copia de la serie tras eliminar el elemento cuya etiqueta se especifica como argumento:

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

```
[ ]: r = s.drop("b")
     r
```

En este ejemplo hemos pasado como único argumento la etiqueta del elemento a eliminar, y el método ha devuelto la serie sin dicho elemento. Si la etiqueta no se encontrase en la serie, se devolvería un error.

También podemos pasar como argumento no una etiqueta, sino una lista de etiquetas. En este caso se eliminarán todos los elementos con dichas etiquetas:

```
[ ]: r = s.drop(["d", "a"])
     r
```

Obsérvese que las etiquetas no tienen que estar en orden.

El argumento inplace = True realiza la eliminación inplace (modificando directamente la serie).

Este método exige el uso de etiquetas para seleccionar los elementos a eliminar. Esto significa que si en un momento dado necesitamos eliminar uno o más elementos por su índice, deberemos convertirlos en sus correspondientes etiquetas, lo que resulta extremadamente sencillo seleccionando los elementos adecuados del index. En el siguiente ejemplo, partimos del mismo ejemplo ya visto anteriormente:

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

Si quisiéramos eliminar los elementos cuyos índices son 1 y 3, bastaría recordar que el atributo index devuelve todas las etiquetas y que s.index[[1, 3]] devuelve las correspondientes a dichos índices:

[ ]:
```
s.index[[1, 3]]
```

Si pasamos esta expresión como argumento del método drop, obtendremos el resultado esperado:

[ ]:
```
s.drop(s.index[[1, 3]])
```

37 Método Where (Buscar - Buscar/Reemplazar)

El método pandas.Series.where permite filtrar los valores de una serie de forma que solo los que cumplan cierta condición se mantengan. Los valores que no la cumplan son sustituidos por un valor (NaN por defecto):

[ ]:
```
s = pd.Series(np.arange(0, 10))
s
```

Supongamos ahora que queremos filtrar los valores de s que sean pares:

[ ]:
```
s.where(s % 2 == 0) #Where devuelve los valores NaN para quienes no cumplen con␣
↪la condición
```

Comprobamos que los valores que no cumplen la condición son sustituidos por NaN. Podemos modificar este valor de reemplazo pasando al método como segundo argumento el valor que queremos fijar:

[ ]:
```
s.where(s % 2 == 0, -1) #Con el segundo argumento, reemplaza os valores
```

38 Edición de DataFrame

Hemos visto la gran variedad de formas que tenemos a nuestra disposición para seleccionar elementos o bloques de elementos de un dataframe, y cada una de estas selecciones puede ser utilizada para modificar los valores contenidos en el dataframe. Veamos algunos ejemplos:

Podemos modificar un valor concreto usando los métodos loc o iloc, en función de que queramos usar sus etiquetas o índices:

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C"])
df
```

[ ]:
```
df.iloc[1, 2] = -1
df
```

Podemos modificar una columna completa seleccionándola y asignándole, por ejemplo, una lista con los nuevos valores. Si partimos del mismo ejemplo que en el caso anterior...

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C"])
print(df)

df["A"] = [10, 20, 30, 40]
df
```

En este caso, la longitud de la lista conteniendo los valores a insertar deberá coincidir con la longitud de la columna, salvo que en lugar de una lista se esté asignando un único valor, en cuyo caso se propagará a toda la columna.

Si la selección es un bloque de datos de un tamaño arbitrario, nos encontramos en el mismo escenario: o bien insertamos datos con el mismo tamaño que la selección, o insertamos un único valor que se propagará a toda la selección. Veamos el primer caso:

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C"])
print(df)

df.loc["b":"c", "A":"B"] = [[-1, -2], [-3, -4]] # [F, C]
df
```

En este ejemplo hemos seleccionado un bloque de 2x2, y hemos insertado datos con una estructura de las mismas dimensiones.

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C"])
print(df)

df.loc["b":"c", "A":"B"] = -1
df
```

y en este segundo caso hemos asignado un único valor a la misma selección.

Nos hemos encontrado también con el caso de insertar datos en una columna o fila inexistente, en

cuyo caso se crea y se le asignan los valores en cuestión. En el primer caso (de columna inexistente):

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C"])
print(df)

df["D"] = [10, 20, 30, 40]
df
```

[ ]:
```
Y en el segundo (de fila inexistente):
```

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape([4, 3]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C"])
print(df)

df.loc["e"] = [10, 20, 30]
df
```

39 El método Where

De forma semejante a las series, el método de los dataframes where filtra los valores contenidos en el dataframe de forma que solo los que cumplan cierta condición se mantengan. El resto de valores son sustituidos por un valor que, por defecto, es NaN.

Por ejemplo, partimos del siguiente dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape(-1, 3), columns=["A", "B", "C"]) #-1 es␣
 ↪igual a 0
df
```

Si ahora queremos filtrar los valores múltiplos de 2, por ejemplo, podemos hacerlo de la siguiente forma:

[ ]:
```
df.where(df % 2 == 0)
```

Todos aquellos valores que no son múltiplo de 2 son sustituidos por NaN. Si, por ejemplo, quisiéramos cambiar de signo a los valores que no cumplen la condición impuesta, lo haríamos así:

[ ]:
```
df = pd.DataFrame(np.arange(12).reshape(-1, 3), columns=["A", "B", "C"]) #-1 es␣
 ↪igual a 0
print(df)

df.where(df % 2 == 0, -df) #Cambiamos el signo desde la matriz "df"
```

40 Eliminación de elementos

El método pandas.DataFrame.drop elimina las filas o columnas indicadas y devuelve el resultado, permitiéndose diferentes criterios para especificarlas.

El primer criterio consiste en indicar la lista de etiquetas a eliminar y el eje al que pertenecen. Partamos del siguiente dataframe:

[ ]:
```
df = pd.DataFrame(np.arange(16).reshape([4, 4]),
                  index = ["a", "b", "c", "d"],
                  columns = ["A", "B", "C", "D"])
df
```

Podemos eliminar, por ejemplo, las filas cuyas etiquetas son “a” y “c” con el siguiente código:

[ ]:
```
df.drop(["a", "c"],axis = 0)  #axis 0 Filas, axis 1 columnas
```

Obsérvese que lo que se muestra es el resultado de eliminar las filas indicadas del dataframe. Éste no se modifica salvo que utilicemos el argumento inplace = True.

Como el eje por defecto es el 0, la instrucción anterior es equivalente a:

[ ]:
```
df.drop(["a", "c"]) #axis 0 Filas, axis 1 columnas
```

Para eliminar columnas, habría que indicar el eje correspondiente:

[ ]:
```
df.drop(["B", "D"], axis = 1) #axis 0 Filas, axis 1 columnas
```

Otra alternativa para especificar si estamos eliminando filas o columnas es utilizar directamente los parámetros index y columns. Así, otra forma de eliminar las filas “a” y “c” sería la siguiente:

[ ]:
```
df.drop(index = ["a", "c"])
```

-el resultado es el mismo que antes, lógicamente-. Y para eliminar las columnas “B” y “D”:

[ ]:
```
df.drop(columns = ["B", "D"])
```

41 Unión de series y dataframes

Frecuentemente nos encontramos con que los datos a analizar están repartidos entre dos o más bloques de datos, lo que nos obliga a unirlos, bien concatenándolos, o bien realizando un “join” entre las estructuras (uniones del mismo tipo que las realizadas en bases de datos). Revisemos las funciones asociadas.

42 Unión de series

Comencemos revisando las opciones disponibles para las series pandas

43 La función concat

Un caso con el que nos encontramos con relativa frecuencia es aquel en el que queremos unir una serie a otra. Por ejemplo:

```
[ ]: s = pd.Series([1, 2, 3, 4, 5,], index = ["a", "b", "c", "d", "e"])
     r = pd.Series([10, 11, 12], index = ["f", "g", "h"])
```

Si deseamos unir r y s en una nueva serie, podemos usar la función pandas.concat. Esta función permite especificar el eje a lo largo del cual unir los diferentes objetos (pueden ser series o dataframes). Por defecto, la concatenación se realiza a lo largo del eje 0:

```
[ ]: t = pd.concat([s, r])
     print(type(t))
     t
```

Podemos ver en el ejemplo anterior que el resultado es una serie pandas.

Si especificamos como eje de concatenación el eje 1, pandas alineará los valores con idénticas etiquetas. En el siguiente ejemplo, las series a y b tienen algunas etiquetas comunes (y otras no). El resultado incluye todas las etiquetas asignando el valor NaN ("Not a Number") a aquellos valores desconocidos:

```
[ ]: a = pd.Series([1, 2, 3, 4, 5,], index = ["a", "b", "c", "d", "e"])
     b = pd.Series([10, 11, 12], index = ["a", "b", "f"])
     pd.concat([a, b], axis = 1, sort = True)
```

(se ha utilizado el argumento sort = True para ocultar cierto aviso al respecto de un cambio en la funcionalidad de esta función en versiones futuras de la librería pandas)

Como puede observarse, el resultado es un dataframe:

```
[ ]: type(pd.concat([a, b], axis = 1, sort = True))
```

Por otro lado, ya sabemos que las etiquetas del índice no tienen por qué ser diferentes, de forma que si estuviésemos concatenando series con etiquetas comunes en sus índices, el resultado sería equivalente a los vistos hasta ahora:

```
[ ]: s = pd.Series([1, 2, 3, 4], index = ["a", "b", "c", "d"])
     r = pd.Series([10, 11, 12], index = ["a", "c", "e"])
     pd.concat([s, r])
```

En este ejemplo hemos concatenado dos series que tienen dos etiquetas comunes ("a" y "c"), y vemos que las dos apariciones de cada una de ellas se incluyen en el resultado de la concatenación.

44 El método append

Otra alternativa es usar el método pandas.Series.append, versión simplificada de la función concat ya vista que devuelve la unión de la serie sobre la que se aplica con otra (u otras) series, pero solo a lo largo del eje 0. Veámoslo en funcionamiento:

[ ]:
```
#Version concat simplificada
a = pd.Series([1, 2, 3, 4, 5], index = ["a", "b", "c", "d", "e"])
b = pd.Series([10, 11, 12], index = ["f", "g", "h"])
```

[ ]:
```
c = a.append(b)
c
```

Si el argumento ignore_index toma el valor True, se ignoran las etiquetas de las series:

[ ]:
```
c = a.append(b, ignore_index = True) #Solo valores
c
```

45 Concatenación y unión de dataframes

Ésta es otra de las áreas en las que la variedad de opciones puede resultar confusa. A modo de resumen, digamos que pandas ofrece dos principales funciones con este objetivo: pandas.concat y pandas.merge.

- La función concat permite concatenar dataframes a lo largo de un determinado eje
- La función merge permite realizar uniones (joins) entre dataframes tal y como se realizan en bases de datos. Esta función también está disponible como método: pandas.DataFrame.merge

Hay una tercera función que está disponible solo como método: pandas.DataFrame.append. El método append ofrece una funcionalidad semejante a la de la función concat pero reducida. Así, por ejemplo, solo permite realizar concatenaciones a lo largo del eje 0 (es decir, verticalmente).

Veamos en las siguientes páginas algunos ejemplos de estas funciones.

46 La función concat

La función pandas.concat es la responsable de concatenar dos o más dataframes (y de todas las estructuras proveídas por pandas) a lo largo de un eje, con soporte a lógica de conjuntos a la hora de gestionar etiquetas en ejes no coincidentes. Veamos un primer caso, el más sencillo posible, para el que partimos de los siguientes dos dataframes:

[ ]:
```
import numpy as np

df1 = pd.DataFrame(np.arange(9).reshape([3, 3]),
                   index = ["a", "b", "d"],
                   columns = ["A", "B", "C"])
df1
```

[ ]:
```
df2 = pd.DataFrame(np.arange(12).reshape([4, 3]),
                   index = ["a", "b", "c", "e"],
                   columns = ["B", "C", "D"])
df2
```

Nota: tal y como ha ocurrido en secciones anteriores, en algunos de los ejemplos que se muestran a continuación se utiliza el argumento sort = False para evitar que se muestre un aviso al respecto de cierto cambio de funcionalidad que se producirá en futuras versiones de esta librería.

Si pasamos a la función concat ambos dataframes como primer argumento (en forma de lista), obtenemos el siguiente resultado:

[ ]:
```
pd.concat([df1, df2], sort = False)
```

Vemos cómo, por defecto, la concatenación se ha realizado a lo largo del eje 0 (eje vertical), uniendo los índices de fila de ambos dataframes, y alineando las columnas por su etiqueta. Los valores para los que no hay datos se han rellenado con NaN (opción correspondiente al argumento por defecto join: "outer").

Si especificamos que la concatenación se realice a lo largo del eje 1 (eje horizontal), el resultado es el siguiente:

[ ]:
```
pd.concat([df1, df2], axis = 1, sort = False) # axis 0 fila, axis 1 columnas,␣
↪por tanto, se agregan nuevas columnas
```

De modo semejante al primer ejemplo, se han introducido NaN's allí donde no había datos, y se han alineado las filas por su etiqueta.

Estos dos ejemplos vistos son tipo "Outer" (opción por defecto), considerando todas las etiquetas de los dos dataframes aun cuando no sean comunes a ambos. Pero si especificamos el argumento join = "Inner", los resultados pasan a considerar solo las etiquetas comunes. Así, para el primer ejemplo visto tenemos:

[ ]:
```
pd.concat([df1, df2], join = "inner") #Solo muestra etiquetas comunes, por␣
↪defecto outer.
```

Incluyendo solo las columnas B y C comunes a ambos dataframes. Y para el segundo ejemplo tenemos:

[ ]:
```
pd.concat([df1, df2], axis = 1, join = "inner") #Solo muestra etiquetas␣
↪comunes, por defecto outer.
```

...incluyendo solo las filas a y b comunes a ambos dataframes.

El parámetro ignore_index controla el índice a asignar al eje a lo largo del cuál se realiza la concatenación. Si este parámetro toma el valor False (por defecto), el eje de concatenación mantiene las etiquetas de los dataframes originales. Si toma el valor True, se ignoran dichas etiquetas y el resultado de la concatenación recibe un nuevo índice automático numérico. Por ejemplo, si añadimos al siguiente ejemplo el argumento ignore_index=True, obtenemos el siguiente resultado:

[ ]:
```
pd.concat([df1, df2], axis = 1, join = "inner", ignore_index = True)
```

47 El método append

Como se ha comentado, el método pandas.DataFrame.append es un atajo de la función concat que ofrece funcionalidad semejante pero limitada: no permite especificar el eje de concatenación (siempre es el eje 0) ni el tipo de "join" (siempre es tipo "Outer").

Si seguimos con los mismos dataframes de la sección anterior:

[ ]:
```
df1 = pd.DataFrame(np.arange(9).reshape([3, 3]),
                   index = ["a", "b", "d"],
                   columns = ["A", "B", "C"])
df1
```

[ ]:
```
df2 = pd.DataFrame(np.arange(12).reshape([4, 3]),
                   index = ["a", "b", "c", "e"],
                   columns = ["B", "C", "D"])
df2
```

...podemos ver cuál es el resultado de aplicar este método a df1:

[ ]:
```
df1.append(df2, sort = False)
```

Al igual que ocurría con la función concat, el parámetro ignore_index nos permite controlar las etiquetas que recibe el índice del resultado: las de los dataframes originales (con ignore_index = False, opción por defecto), o uno nuevo automático (con ignore_index = True).