Introducción al objeto Series

El primero de los objetos de datos Pandas que vamos a estudiar es el objeto Series.

Un objeto Series es una generalización de un array de una dimensión y su implementación está basada en el tio ndarray de Numpy.

El objeto Series, podemos utilizarlo para representar una función matemática o datos agrupados, como pueden ser las cotizaciones de empresas en bolsa.

Inicialización del entorno.

Como siempre, lo primero que necesitamos será importar el paquete Pandas con alias `pd`.

También vamos a visualizar la versión usada de Pandas que usamos ya que es un dato importante para consultar la documentación. En el momento de editar este notebook la versión de pandas es: 1.4.3

Además, para facilitar la realización de los ejercicios, también importamos el paquete Numpy con el alias `np`.

In [5]:
```
import pandas as pd
import numpy as np
np.set_printoptions(floatmode='fixed', precision=3)
print('Pandas versión: ', pd.__version__)
```

```
Pandas versión:  1.4.3
```

Uso básico del objeto Series.

La clase `pandas.Series` representa un array 1-D cuyo eje puede ser indexado con etiquetas tanto numéricas, como textuales o incluso temporales.

Pandas reutiliza los métodos de Numpy automáticamente excluyendo posibles valores perdidos (representados con Nan) e incluso alineando los ejes si los arrays no tienen el mismo número de elementos.

Hay varias formas de crear un objeto Series. La más básica es usar una lista python.

Ejercicio 01: Dada la lista `[1.0, -1.5, 2.3, 3.4]` crear un objeto Series.

La salida debería ser algo parecido a lo siguiente:

```
0    1.0
1   -1.5
2    2.3
3    3.4
dtype: float64
```

In [6]:
```
serie = 0
#Pon tu código aquí.
#Sugerencia: usa el constructor pd.Series(...)

#
print(serie)
```

```
0
```

Observa que al no indicar un objeto `index`, se crea utiliza un índice entero automáticamente.

También podemos crear un objeto Series desde un numpy.ndarray con la condición de que tenga una única dimensión.

Ejercicio 02: Generar un objeto serie usando un Numpy.ndarray.

La salida debería ser algo parecido a lo siguiente:

```
0    0.548814
1    0.715189
2    0.602763
3    0.544883
4    0.423655
dtype: float64
```

In [7]:
```
gen = np.random.seed(0)
a = np.random.rand(5)
serie = 0
#Pon tu código aquí.
#Sugerencia: usa el constructor pd.Series(...)

#
print(serie)
```

```
0
```

Como se puede ver, el objeto Series tiene dos atributos principales:

- Los datos accesible con Series.array.
- Un índice accesible con Series.index.

Cuando se crea un objeto Series si no se indica un `índice` este se genera de forma automática con un rango entero `np.arange(<num datos>)`

Ejercicio 03: Crear un objeto Series con los datos `[1, -1, 3, 5]` y usando como índice asociado `['a', 'b', 'c', 'd']`. Forzar que los datos sean tipo np.float32

La salida debería ser algo parecido a lo siguiente:

```
Serie.array: <PandasArray>
[1.0, -1.0, 3.0, 5.0]
Length: 4, dtype: float32
Serie.index :  Index(['a', 'b', 'c', 'd'], dtype='object')
Serie:
 a    1.0
b   -1.0
c    3.0
```

```
d    5.0
dtype: float32
```

In [8]:
```
datos = [1, -1, 3, 5]
index = ['a', 'b', 'c', 'd']
serie = pd.Series()
#Pon tu código aquí
#Sugerencia: hay tres argumentos que indicar.

#

print('Serie.array: ', serie.array)
print('Serie.index : ', serie.index)
print('Serie:\n', serie)
```

```
Serie.array:  <PandasArray>
[]
Length: 0, dtype: float64
Serie.index :  Index([], dtype='object')
Serie:
 Series([], dtype: float64)
```

```
/tmp/ipykernel_24814/4059156636.py:3: FutureWarning: The default dtype for empty Series
will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to
silence this warning.
  serie = pd.Series()
```

Alternativamente podemos crear un objeto Series a partir de un diccionario. En este caso las claves definen el índice y los valores de las entradas del diccionario los valores de la serie.

Ejercicio 04: Usando el diccionario `{'a':0, 'b':-1, 'c':3, 'd':0}` crear un objeto Series correspondiente.

La salida debería ser algo parecido a lo siguiente:

```
serie.array:  <PandasArray>
[0, -1, 3, 0]
Length: 4, dtype: int64
serie.index:  Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [9]:
```
d = {'a':0, 'b':-1, 'c':3, 'd':0}
serie = pd.Series()
#Pon tu código aquí.
#Sugerencia: crea el objeto Series con el diccionario
#como argumento.

#
print('serie.array: ', serie.array)
print('serie.index: ', serie.index)
```

```
serie.array:  <PandasArray>
[]
Length: 0, dtype: float64
serie.index:  Index([], dtype='object')
```

```
/tmp/ipykernel_24814/3394067010.py:2: FutureWarning: The default dtype for empty Series
will be 'object' instead of 'float64' in a future version. Specify a dtype explicitly to
silence this warning.
  serie = pd.Series()
```

Usando un objeto Series como un numpy.ndarray

El objeto Series se puede utilizar de forma muy similar a cómo usamos un numpy.ndarray de una dimensión.

Ejercicio 05: Dado un objeto Series queremos imprimir el tercer valor de la series representada.

La salida debería ser algo parecido a lo siguiente:

```
Tercer valor: 0.603
```

In [10]:
```
np.random.seed(0)#Para poder reproducir.
s = pd.Series(np.random.random(10))
valor = 0
#Pont tu código aquí.
#Sugerencia: utiliza el objeto Series como si fuera un ndarray.

#
print(f"Tercer valor: {valor:.3f}")
```

```
Tercer valor: 0.000
```

Al igual que en un ndarray, podemos especifir un rango `inincio:fin:paso` para indexar un objeto Series, sin embargo, no solo se indexarán los valores; también se indexará el Índice asociado.

Ejercicio 06: Dada una serie, imprimir los valores en posiciones de índice pares.

La salida debería ser algo como:

```
Valores en posiciones pares (método directo):
0    0.548814
2    0.602763
4    0.423655
dtype: float64
```

In [11]:
```
np.random.seed(0)#Para poder reproducir.
s = pd.Series(np.random.random(5))
print("Valores en posiciones pares (método directo):")
#Pont tu código aquí.
#Sugerencia: usar directamente el objeto Series como si fuera un ndarray.

#
```

```
Valores en posiciones pares (método directo):
```

En algunas ocasiones puede ser intersante acceder a una versión "ndarray" del objeto Series. Por ejemplo, ya sabemos que si en un 1-D ndarray `a` indicamos el índice `a[-1]`, el valor indexado será el último, pero indicar en objeto Series `s` el índice `s[-1]` generará en general una excepción `ValueError` porque el valor `-1` no es válido para el índice de la serie. En este caso podremos usar la método Series.to_numpy() para obtener una "vista" del objeto Series como un `np.ndarray`.

Ejercicio 07: Imprimir, usando índices negativos, el último y el penúltimo elemento de una serie.

La salida debería ser algo como:

```
Vista          : [0.549 0.715 0.603 0.545 0.424]
Último valor   : 0.424
Penúltimo valor: 0.545
```

In [12]:
```
np.random.seed(0)#Para poder reproducir.
s = pd.Series(np.random.random(5))
ult = 0
penult = 0
#Pont tu código aquí.
#Sugerencia: Prueba dos alternativas, primero usar directamente el
#objeto Series como un ndarray. La segunda es obtener un vista
#usando el método Series.to_numpy()

#
print(f"Vista          : {v}")
print(f"Último valor   : {ult:.3f}")
print(f"Penúltimo valor: {penult:.3f}")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [12], in <cell line: 11>()
      4 penult = 0
      5 #Pont tu código aquí.
      6 #Sugerencia: Prueba dos alternativas, primero usar directamente el
      7 #objeto Series como un ndarray. La segunda es obtener un vista
      8 #usando el método Series.to_numpy()
      9
     10 #
---> 11 print(f"Vista          : {v}")
     12 print(f"Último valor   : {ult:.3f}")
     13 print(f"Penúltimo valor: {penult:.3f}")

NameError: name 'v' is not defined
```

Usando el objeto Series como un diccionario.

Como ya hemos visto, un objeto Series tiene asociado un Índice y podemos utilizar los valores de este índice para obtener el valor asociado. Hasta ahora hemos usado series siendo el índice un rango entero `[0, <tamaño de la serie>)` pero ya sabemos que el índice también puede ser textual. En este caso la serie actua como un dicccionario.

Ejercicio 08: La serie `ibex` almacena las cotizaciones de para varias empresas que cotizan en el índice bulsátil "IBEX". En este caso los valores de índice será los identificadores de las empresas coticadas. Mostrar el valor de cotización de la empresa "VVBA".

La salida debería ser algo como:

```
Cotización de "VVBA":

2.892
```

In [ ]:
```
ibex = pd.Series({'BANQIA':0.88, 'VVBA':2.892, 'SANTAN':2.076,
                  'CAJANOR':1.763})
print('Cotización de "VVBA": \n')
#Pon tu código aquí.
#Sugerencia: usa el texto "VVBA" como índice.

#
```

Un inconveniente de usar la notación `[]` para acceder como diccionario es que si el valor de Ìndice no existe, se genera una excepción de tipo `KeyError`. Usando el método Series.get(), si el valor de índice dado no es válido, se devolverá un valor por defecto `None`.

Ejercicio 09: La una serie `ibex` almacena las cotizaciones de para varias empresas que cotizan en el índice bulsátil "IBEX". Captura la excepción que se produce al mostrar el valor de cotización asociado a la empresa inexisitente "DESCONOCIDA" usando el operador `[]`. En la gestión de la excepción utiliza el método `get()` y muestra finalmente el valor obteneido.

La salida debería ser algo como:

```
Cotización de "DESCONOCIDA":

Valor de cotización  None
```

In [ ]:
```
ibex = pd.Series({'BANQIA':0.88, 'VVBA':2.892, 'SANTAN':2.076,
            'CAJANOR':1.763})
print('Cotización de "DESCONOCIDA": \n')
cot = 0
try:
#Pon tu código aquí.
#Sugerencia: usa el operador []
    cot = 0
#

except KeyError:
#
#Pon tu código aquí
#Sugerencia: captura la excepción KeyError y usa el método get().
    cot = 0
#
print("Valor de cotización ", cot)
```

Otros atributos de un objeto Series.

El objeto series tiene además de los atributos `values` e `index` otros atributos.

Ejercicio 10: Se desea saber el tipo de los elementos almacenados en un objeto Serie.

La salida debería ser algo como:

```
Tipo de los elementos: float64
```

In [ ]:
```
ibex = pd.Series({'BANQIA':0.88, 'VVBA':2.892, 'SANTAN':2.076,
            'CAJANOR':1.763})
print('Tipo de los elementos: ', end='')
#Pon tu código aquí.
#Sugerencia: utiliza el atributo 'dtype'

#
```

Ejercicio 11: Se desea saber el total de elementos almacenados en la serie.

La salida debería ser algo como:

```
Número de elementos: 4
```

In [ ]:
```
ibex = pd.Series({'BANQIA':0.88, 'VVBA':2.892, 'SANTAN':2.076,
            'CAJANOR':1.763})
print('Número de elementos: ', end='')
#Pon tu código aquí.
```

```
#Sugerencia: utiliza el atributo 'size'

#
```

Ejercicio 12: Se desea saber si una serie tiene algún valor `np.nan`.

La salida debería ser algo como:

```
ibex1 tiene valores nan? : False
ibex2 tiene valores nan? : True
```

In [ ]:
```
ibex1 = pd.Series({'BANQIA':0.88, 'VVBA':2.892, 'SANTAN':2.076,
            'CAJANOR':1.763})
ibex2 = pd.Series({'BANQIA':np.nan, 'VVBA':2.892, 'SANTAN':2.076,
            'CAJANOR':1.763})
print('ibex1 tiene valores nan? : ', end='')
#Pon tu código aquí.
#Sugerencia: utiliza el atributo 'hasnans'

#
print('ibex2 tiene valores nan? : ', end='')
#Pon tu código aquí.
#Sugerencia: utiliza el atributo 'hasnans'

#
```

Ejercicio 13: Se desea saber el nombre de una series. Este valor se establece al crearlo con el argumento 'name'.

La salida debería ser algo como:

```
Nombre de la serie: Cotizaciones
```

In [ ]:
```
ibex = pd.Series({'BANQIA':0.88, 'VVBA':2.892, 'SANTAN':2.076,
            'CAJANOR':1.763}, name='Cotizaciones')
print('Nombre de la serie: ', end='')
#Pon tu código aquí.
#Sugerencia: utiliza el atributo 'name'

#
```

In [ ]:
```

```