**Numpy**

**Numpy** es una libreria para la ciencia de los datos en Python. Proporciona un objeto **matriz** multidimensional de alto rendimiento y herramientas para trabajar con estas matrices.

<https://numpy.org/>

**Matrices**

Una matriz numpy es una cuadrícula de valores, todos del mismo tipo, y está indexada por una tupla de enteros no negativos. El número de dimensiones es el rango de la matriz; la forma de una matriz es una tupla de números enteros que dan el tamaño de la matriz a lo largo de cada dimensión.

Podemos inicializar matrices numpy desde listas de Python anidadas y acceder a elementos usando corchetes:

In [40]:

**from** **numpy** **import** random **as** r

In [126]:

print(r.choice(['Andres','Juan','Pedro', 'Mateo'], size= r.choice([1,2],p=[0.1,0.9]) , p=[0.5,0.2,0.2,0.1], replace=**False**))

['Juan']

In [127]:

**import** **numpy** **as** **np**

a = np.array([34, 25, 7]) *# Crear una matriz de rango 1*

In [128]:

print(type(a)) *# Prints "<class 'numpy.ndarray'>"*

<class 'numpy.ndarray'>

In [129]:

print(a.shape) *# Prints "(3,)"*

(3,)

In [130]:

print(a[0], a[1], a[2]) *# Prints "1 2 3"*

34 25 7

In [131]:

a[0] = 5 *# Cambiar un elemento de la matriz*

In [132]:

print(a) *# Prints "[5, 2, 3]"*

[ 5 25 7]

In [133]:

b = np.array([[1,2,3],[4,5,6]]) *# Crear una matriz de rango 2*

In [25]:

print(b.shape) *# Prints "(2, 3)"*

(2, 3)

In [134]:

print(b[0, 0], b[0, 1], b[1, 0]) *# Prints "1 2 4"*

1 2 4

Numpy también proporciona muchas funciones para crear matrices:

In [137]:

**import** **numpy** **as** **np**

matriz = np.zeros((3,3)) *# Crear una matriz de todos los ceros*

print(matriz) *# Prints "[[ 0. 0.]*

*# [ 0. 0.]]"*

[[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]]

In [138]:

b = np.ones((1,2)) *# Crear un conjunto de todos ellos*

print(b) *# Prints "[[ 1. 1.]]"*

[[1. 1.]]

In [143]:

c = np.full((2,2), 7) *# Crear una matriz constante*

print(c) *# Prints "[[ 7. 7.]*

*# [ 7. 7.]]"*

[[7 7]

[7 7]]

In [13]:

d = np.eye(2) *# Crear una matriz de identidad 2x2*

print(d) *# Prints "[[ 1. 0.]*

*# [ 0. 1.]]"*

[[1. 0.]

[0. 1.]]

In [5]:

e = np.random.random((2,2)) *# Crear una matriz llena de valores aleatorios*

print(e) *# Podría imprimir "[[ 0.91940167 0.08143941]*

*# [ 0.68744134 0.87236687]]"*

[[0.00175892 0.12706911]

[0.03101381 0.71757546]]

Puede leer sobre otros métodos de creación de matrices en la documentación de Numpy

<https://numpy.org/doc/stable/user/basics.creation.html#arrays-creation>

**Indexación de matrices**

Numpy ofrece varias formas de indexar en matrices.

**Rebanar**

Similar a las listas de Python, las matrices numpy se pueden cortar. Dado que las matrices pueden ser multidimensionales, debe especificar un segmento para cada dimensión de la matriz:

In [165]:

**import** **numpy** **as** **np**

*# Crear la siguiente matriz de rango 2 con forma (3, 4)*

*# [[ 1 2 3]*

*# [ 5 6 7]*

*# [ 9 10 11 ]]*

a = np.array([[1,2,3], [5,6,7], [9,10,11]])

In [148]:

*# Usar el rebanado para sacar el subconjunto que consiste en las 2 primeras filas*

*# y las columnas 1 y 2; b es el siguiente conjunto de forma (2, 2):*

*# [[2 3]*

*# [6 7]]*

b = a[:2, 1:3]

print(b)

[[2 3]

[6 7]]

In [167]:

print(np.fliplr(a))

[[ 3 2 1]

[ 7 6 5]

[11 10 9]]

In [149]:

*# Una rebanada de una matriz es una vista en los mismos datos, por lo que modificarla*

*# modificará la matriz original.*

print(a[0, 1]) *# Prints "2"*

2

In [150]:

b[0, 0] = 77 *# b[0, 0] es la misma pieza de datos que a[0, 1]*

In [151]:

print(a[0, 1]) *# Prints "77"*

77

También puede mezclar la indexación de enteros con la indexación de sectores. Sin embargo, al hacerlo, se obtendrá una matriz de rango más bajo que la matriz original.

In [21]:

**import** **numpy** **as** **np**

*# Crear la siguiente matriz de rango 2 con forma (3, 4)*

*# [[ 1 2 3 4]*

*# [ 5 6 7 8]*

*# [ 9 10 11 12]]*

a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

In [ ]:

*# Dos formas de acceder a los datos de la fila del medio de la matriz.*

*# Mezclando la indexación de enteros con rebanadas se obtiene una matriz de rango inferior,*

*# mientras que usando sólo rebanadas se obtiene un conjunto del mismo rango que el*

*# La matriz original:*

row\_r1 = a[1, :] *# Rank 1 view of the second row of a*

row\_r2 = a[1:2, :] *# Rank 2 view of the second row of a*

In [22]:

print(row\_r1, row\_r1.shape) *# Prints "[5 6 7 8] (4,)"*

print(row\_r2, row\_r2.shape) *# Prints "[[5 6 7 8]] (1, 4)"*

[5 6 7 8] (4,)

[[5 6 7 8]] (1, 4)

In [23]:

*# Podemos hacer la misma distinción al acceder a las columnas de una matriz:*

col\_r1 = a[:, 1]

col\_r2 = a[:, 1:2]

print(col\_r1, col\_r1.shape) *# Prints "[ 2 6 10] (3,)"*

print(col\_r2, col\_r2.shape) *# Prints "[[ 2]*

*# [ 6]*

*# [10]] (3, 1)"*

[ 2 6 10] (3,)

[[ 2]

[ 6]

[10]] (3, 1)

**Indexación de matrices de enteros:**

Cuando indexa matrices de números utilizando la división, la vista de matriz resultante siempre será una submatriz de la matriz original. Por el contrario, la indexación de matrices de enteros le permite construir matrices arbitrarias utilizando los datos de otra matriz. Aquí hay un ejemplo:

In [6]:

**import** **numpy** **as** **np**

a = np.array([[1,2], [3, 4], [5, 6]])

In [8]:

*# Un ejemplo de indexación de arreglos enteros.*

*# La matriz devuelta tendrá forma (3,2) y*

print(a.shape)

print(a[[0, 1, 2], [0, 1, 0]]) *# Prints "[1 4 5]"*

(3, 2)

[1 4 5]

In [26]:

*# El ejemplo anterior de indexación de arreglos enteros es equivalente a esto:*

print(np.array([a[0, 0], a[1, 1], a[2, 0]])) *# Prints "[1 4 5]"*

[1 4 5]

In [27]:

*# Cuando se usa la indexación de arreglos enteros, se puede reutilizar el mismo*

*# elemento de la matriz de la fuente:*

print(a[[0, 0], [1, 1]]) *# Prints "[2 2]"*

[2 2]

In [28]:

*# Equivalente al ejemplo anterior de indexación de arreglos enteros*

print(np.array([a[0, 1], a[0, 1]])) *# Prints "[2 2]"*

[2 2]

Un truco útil con la indexación de matrices enteras es seleccionar o mutar un elemento de cada fila de una matriz:

In [29]:

**import** **numpy** **as** **np**

*# Crear una nueva matriz de la cual seleccionaremos elementos*

a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a) *# prints "array([[ 1, 2, 3],*

*# [ 4, 5, 6],*

*# [ 7, 8, 9],*

*# [10, 11, 12]])"*

[[ 1 2 3]

[ 4 5 6]

[ 7 8 9]

[10 11 12]]

In [31]:

*# Crear una serie de índices*

b = np.array([0, 2, 0, 1])

*# Selecciona un elemento de cada fila de a usando los índices de b*

print(a[np.arange(4), b]) *# Prints "[ 1 6 7 11]"*

[11 16 17 21]

In [32]:

*# Muta un elemento de cada fila de a usando los índices de b*

a[np.arange(4), b] += 10

print(a) *# prints "array([[11, 2, 3],*

*# [ 4, 5, 16],*

*# [17, 8, 9],*

*# [10, 21, 12]])*

[[21 2 3]

[ 4 5 26]

[27 8 9]

[10 31 12]]

**Indexación de matriz booleana:**

La indexación de matriz booleana le permite seleccionar elementos arbitrarios de una matriz. Con frecuencia, este tipo de indexación se utiliza para seleccionar los elementos de una matriz que satisfacen alguna condición. Aquí hay un ejemplo:

In [33]:

**import** **numpy** **as** **np**

a = np.array([[1,2], [3, 4], [5, 6]])

In [34]:

bool\_idx = (a > 2) *# Encuentra los elementos de a que son más grandes que 2;*

*# esto devuelve un conjunto numérico de Booleans de la misma*

*# forma que a, donde cada ranura de bool\_idx dice*

*# si ese elemento de a es > 2.*

In [35]:

print(bool\_idx) *# Prints "[[False False]*

*# [ True True]*

*# [ True True]]"*

[[False False]

[ True True]

[ True True]]

In [36]:

*# Usamos la indexación de arreglos booleanos para construir un arreglo de rango 1*

*# que consiste en los elementos de un correspondiente a los valores Verdaderos*

*# de bool\_idx*

print(a[bool\_idx]) *# Prints "[3 4 5 6]"*

[3 4 5 6]

In [37]:

*# Podemos hacer todo lo anterior en una sola declaración concisa:*

print(a[a > 2]) *# Prints "[3 4 5 6]"*

[3 4 5 6]

Por brevedad, hemos omitido muchos detalles sobre la indexación de matrices numpy; si quieres saber más debes leer la documentación .

<https://numpy.org/doc/stable/reference/arrays.indexing.html>

**Tipos de datos**

Cada matriz numpy es una cuadrícula de elementos del mismo tipo. Numpy proporciona un gran conjunto de tipos de datos numéricos que puede utilizar para construir matrices. Numpy intenta adivinar un tipo de datos cuando crea una matriz, pero las funciones que construyen matrices generalmente también incluyen un argumento opcional para especificar explícitamente el tipo de datos. Aquí hay un ejemplo:

In [38]:

**import** **numpy** **as** **np**

x = np.array([1, 2]) *# Dejar que numpy elija el tipo de datos*

print(x.dtype) *# Prints "int64"*

int32

In [39]:

x = np.array([1.0, 2.0]) *# Dejar que numpy elija el tipo de datos*

print(x.dtype) *# Prints "float64"*

float64

In [40]:

x = np.array([1, 2], dtype=np.int64) *# Forzar un tipo de datos en particular*

print(x.dtype) *# Prints "int64"*

int64

Puede leer todo sobre numerosos tipos de datos en la documentación .

<https://numpy.org/doc/stable/reference/arrays.dtypes.html>

**Matemáticas de matriz**

Las funciones matemáticas básicas operan por elementos en matrices y están disponibles como sobrecargas de operador y como funciones en el módulo numpy:

In [41]:

**import** **numpy** **as** **np**

x = np.array([[1,2],[3,4]], dtype=np.float64)

y = np.array([[5,6],[7,8]], dtype=np.float64)

In [42]:

*# Suma de elementos; ambos producen la matriz*

*# [[ 6.0 8.0]*

*# [10.0 12.0]]*

print(x + y)

print(np.add(x, y))

[[ 6. 8.]

[10. 12.]]

[[ 6. 8.]

[10. 12.]]

In [43]:

*# Diferencia de elementos (resta); ambos producen la matriz*

*# [[-4.0 -4.0]*

*# [-4.0 -4.0]]*

print(x - y)

print(np.subtract(x, y))

[[-4. -4.]

[-4. -4.]]

[[-4. -4.]

[-4. -4.]]

In [44]:

*# Producto de elementos; ambos producen la matriz*

*# [[ 5.0 12.0]*

*# [21.0 32.0]]*

print(x \* y)

print(np.multiply(x, y))

[[ 5. 12.]

[21. 32.]]

[[ 5. 12.]

[21. 32.]]

In [46]:

*# División de elemetos; ambos producen la matriz*

*# [[ 0.2 0.33333333]*

*# [ 0.42857143 0.5 ]]*

print(x / y)

print(np.divide(x, y))

[[0.2 0.33333333]

[0.42857143 0.5 ]]

[[0.2 0.33333333]

[0.42857143 0.5 ]]

In [47]:

*# Raíz cuadrada de elemtos; produce la matriz*

*# [[ 1. 1.41421356]*

*# [ 1.73205081 2. ]]*

print(np.sqrt(x))

[[1. 1.41421356]

[1.73205081 2. ]]

Tenga en cuenta que **es una multiplicación por elementos, no una multiplicación de matrices**. En cambio, usamos la dotfunción para calcular productos internos de vectores, para multiplicar un vector por una matriz y para multiplicar matrices. dot está disponible como función en el módulo numpy y como método de instancia de objetos de matriz:

In [48]:

**import** **numpy** **as** **np**

x = np.array([[1,2],[3,4]])

y = np.array([[5,6],[7,8]])

In [49]:

v = np.array([9,10])

w = np.array([11, 12])

In [50]:

*# Producto interno de los vectores; ambos producen 219*

print(v.dot(w))

print(np.dot(v, w))

219

219

In [51]:

*# Matriz / producto vectorial; ambos producen la matriz de rango 1 [29 67]*

print(x.dot(v))

print(np.dot(x, v))

[29 67]

[29 67]

In [52]:

*# Matriz / producto de la matriz; ambos producen la matriz de rango 2*

*# [[19 22]*

*# [43 50]]*

print(x.dot(y))

print(np.dot(x, y))

[[19 22]

[43 50]]

[[19 22]

[43 50]]

Numpy proporciona muchas funciones útiles para realizar cálculos en matrices; uno de los más útiles es sum:

In [53]:

**import** **numpy** **as** **np**

x = np.array([[1,2],[3,4]])

In [54]:

print(np.sum(x)) *# Calcular la suma de todos los elementos; imprime "10"*

print(np.sum(x, axis=0)) *# Calcula la suma de cada columna; imprime "[4 6]"*

print(np.sum(x, axis=1)) *# Calcula la suma de cada fila; imprime "[3 7]"*

10

[4 6]

[3 7]

Puede encontrar la lista completa de funciones matemáticas proporcionadas por numpy en la documentación.

<https://numpy.org/doc/stable/reference/routines.math.html>

Además de calcular funciones matemáticas utilizando matrices, con frecuencia necesitamos remodelar o manipular datos en matrices. El ejemplo más simple de este tipo de operación es la transposición de una matriz; para transponer una matriz, simplemente use el **T** atributo de un objeto de matriz:

In [55]:

**import** **numpy** **as** **np**

x = np.array([[1,2], [3,4]])

print(x) *# Prints "[[1 2]*

*# [3 4]]"*

print(x.T) *# Prints "[[1 3]*

*# [2 4]]"*

[[1 2]

[3 4]]

[[1 3]

[2 4]]

In [56]:

*# Note que tomar la transposición de una matriz de rango 1 no hace nada:*

v = np.array([1,2,3])

print(v) *# Prints "[1 2 3]"*

print(v.T) *# Prints "[1 2 3]"*

[1 2 3]

[1 2 3]

Numpy proporciona muchas más funciones para manipular matrices; puedes ver la lista completa en la documentación. <https://numpy.org/doc/stable/reference/routines.array-manipulation.html>

**Broadcasting**

El Broadcasting es un mecanismo poderoso que permite a numpy trabajar con matrices de diferentes formas al realizar operaciones aritméticas. Con frecuencia tenemos una matriz más pequeña y una matriz más grande, y queremos usar la matriz más pequeña varias veces para realizar alguna operación en la matriz más grande.

Por ejemplo, suponga que queremos agregar un vector constante a cada fila de una matriz. Podríamos hacerlo así:

In [57]:

**import** **numpy** **as** **np**

*# Añadiremos el vector v a cada fila de la matriz x,*

*# almacenando el resultado en la matriz y*

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

v = np.array([1, 0, 1])

y = np.empty\_like(x) *# Crear una matriz vacía con la misma forma que x*

*# Agrega el vector v a cada fila de la matriz x con un bucle explícito*

**for** i **in** range(4):

y[i, :] = x[i, :] + v

In [58]:

*# Ahora y es lo siguiente*

*# [[ 2 2 4]*

*# [ 5 5 7]*

*# [ 8 8 10]*

*# [11 11 13]]*

print(y)

[[ 2 2 4]

[ 5 5 7]

[ 8 8 10]

[11 11 13]]

Esto funciona; sin embargo, cuando la matriz x es muy grande, calcular un bucle explícito en Python podría ser lento. Tenga en cuenta que agregar el vector va cada fila de la matriz x es equivalente a formar una matriz vv apilando múltiples copias de v verticalmente, luego realizando la suma de elementos de x y vv. Podríamos implementar este enfoque de esta manera:

In [59]:

**import** **numpy** **as** **np**

*# Añadiremos el vector v a cada fila de la matriz x,*

*# almacenando el resultado en la matriz y*

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

v = np.array([1, 0, 1])

vv = np.tile(v, (4, 1)) *# Amontonar 4 copias de V una encima de la otra*

print(vv) *# Prints "[[1 0 1]*

*# [1 0 1]*

*# [1 0 1]*

*# [1 0 1]]"*

y = x + vv *# Agrega x y vv elementalmente*

print(y) *# Prints "[[ 2 2 4*

*# [ 5 5 7]*

*# [ 8 8 10]*

*# [11 11 13]]"*

[[1 0 1]

[1 0 1]

[1 0 1]

[1 0 1]]

[[ 2 2 4]

[ 5 5 7]

[ 8 8 10]

[11 11 13]]

El Broadcasting Numpy nos permite realizar este cálculo sin crear realmente múltiples copias de v. Considere esta versión, usando transmisión:

In [15]:

**import** **numpy** **as** **np**

*# We will add the vector v to each row of the matrix x,*

*# storing the result in the matrix y*

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

v = np.array([[1, 0],[3, 0]])

print(v)

print(3 **in** np.sum(v,axis=0))

*#y = x + v # Añada v a cada fila de x utilizando la radiodifusión*

*#print(y) # Prints "[[ 2 2 4]*

*# [ 5 5 7]*

*# [ 8 8 10]*

*# [11 11 13]]"*

[[1 0]

[3 0]]

False

La línea y = x + v funciona aunque x tiene forma (4, 3) y v tiene forma (3,) debido a la transmisión; esta línea funciona como si v realmente tuviera forma (4, 3), donde cada fila era una copia de v, y la suma se realizó por elementos.

**El Broadcasting de dos matrices juntas sigue estas reglas:**

1. Si las matrices no tienen el mismo rango, anteponga 1 a la forma de la matriz de rango inferior hasta que ambas formas tengan la misma longitud.
2. Se dice que las dos matrices son compatibles en una dimensión si tienen el mismo tamaño en la dimensión, o si una de las matrices tiene el tamaño 1 en esa dimensión.
3. Los arreglos se pueden transmitir juntos si son compatibles en todas las dimensiones.
4. Después de la transmisión, cada matriz se comporta como si tuviese una forma igual al máximo de formas de las dos matrices de entrada.
5. En cualquier dimensión donde una matriz tiene un tamaño 1 y la otra matriz tiene un tamaño mayor que 1, la primera matriz se comporta como si se hubiera copiado a lo largo de esa dimensión.

Ver

<https://numpy.org/doc/stable/user/basics.broadcasting.html> <http://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc>

Las funciones que apoyan el Broadcasting se conocen como funciones universales. Puede encontrar la lista de todas las funciones universales en la documentación.

<https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs>

Estas son algunas aplicaciones del Broadcasting:

In [61]:

**import** **numpy** **as** **np**

*# Calcular el producto exterior de los vectores*

v = np.array([1,2,3]) *# v tiene forma (3,)*

w = np.array([4,5]) *# w tiene forma (2,)*

*# Para calcular un producto exterior, primero reformamos v para que sea una columna*

*# vector de forma (3, 1); podemos entonces emitirlo contra w para rendir*

*# una salida de la forma (3, 2), que es el producto exterior de v y w:*

*# [[ 4 5]*

*# [ 8 10]*

*# [12 15]]*

print(np.reshape(v, (3, 1)) \* w)

[[ 4 5]

[ 8 10]

[12 15]]

In [62]:

*# Agregar un vector a cada fila de una matriz*

x = np.array([[1,2,3], [4,5,6]])

*# x tiene forma (2, 3) y v tiene forma (3,) por lo que transmiten a (2, 3),*

*# dando la siguiente matriz:*

*# [[2 4 6]*

*# [5 7 9]]*

print(x + v)

[[2 4 6]

[5 7 9]]

In [63]:

*# Agregar un vector a cada columna de una matriz*

*# La x tiene forma (2, 3) y la w tiene forma (2,).*

*# Si transponemos x entonces tiene forma (3, 2) y puede ser difundida*

*# contra w para obtener el resultado de la forma (3, 2); transponiendo este resultado*

*# produce el resultado final de la forma (2, 3) que es la matriz x con*

*# el vector w añadido a cada columna. Da la siguiente matriz:*

*# [[ 5 6 7]*

*# [ 9 10 11]]*

print((x.T + w).T)

[[ 5 6 7]

[ 9 10 11]]

In [64]:

*# Otra solución es remodelar la w para que sea un vector de forma de la columna (2, 1);*

*# podemos entonces emitirlo directamente contra x para producir la misma*

*# salida.*

print(x + np.reshape(w, (2, 1)))

[[ 5 6 7]

[ 9 10 11]]

In [65]:

*# Multiplica una matriz por una constante:*

*# x tiene forma (2, 3). Numpy trata los escalares como matrices de forma ();*

*# estos pueden ser emitidos juntos a la forma (2, 3), produciendo el*

*# Siguiendo la matriz:*

*# [[ 2 4 6]*

*# [ 8 10 12]]*

print(x \* 2)

[[ 2 4 6]

[ 8 10 12]]

El Broadcasting suele hacer que su código sea más conciso y rápido, por lo que debe esforzarse por utilizarlo siempre que sea posible.

Documentación Numpy Esta breve descripción general ha abordado muchas de las cosas importantes que necesita saber sobre numpy, pero está lejos de ser completa. Consulte la referencia de numpy para obtener más información sobre numpy.

<https://numpy.org/doc/stable/reference/>

**Ejercicio Triqui (tres en linea, juego de la vieja, tres en raya, etc.)**

In [ ]:

**import** **numpy** **as** **np**

**def** displayInicialTriqui():

**for** i **in** range(0,3):

**for** j **in** range(0,3):

print("|\_",end="")

print("|")

**def** displayTriqui(matriz):

**for** i **in** range(0,3):

**for** j **in** range(0,3):

**if** matriz[i,j]==1:

print("|X", end="")

**else**:

**if** matriz[i,j]==10:

print("|O",end="")

**else**:

print("|\_",end="")

print("|")

**def** estadoDelJuego(matriz)->int:

*#devuelve 1 si gana la X*

**if** 3 **in** np.sum(matriz, axis=0) **or** 3 **in** np.sum(matriz, axis=1) **or** np.sum(np.diagonal(matriz))==3 **or** np.sum(np.diagonal(np.fliplr(matriz)))==3:

salida = 1

**else**:

*#devuelve 2 si gana la Y*

**if** 30 **in** np.sum(matriz, axis=0) **or** 30 **in** np.sum(matriz, axis=1) **or** np.sum(np.diagonal(matriz))==30 **or** np.sum(np.diagonal(np.fliplr(matriz)))==30:

salida = 2

**else**:

*#devuelve 3 si no hay ganador*

**if** np.sum(matriz)==45 **or** np.sum(matriz)==54:

salida = 3

**else**:

*#devuelve 4 si aun hay juego*

salida = 4

**return** salida

**def** jugarTriqui():

*#seleccionar aleatoriamente un jugador*

turno=**False**

**if** np.random.rand()<0.5:

turno=**True**

*#inicializar las variables del juego*

displayInicialTriqui()

matriz = np.zeros((3,3)) *# Crear una matriz de todos los ceros*

estado=4

**while**(estado==4):

*#atrapar el movimiento de los jugadores*

tupla= tuple(input("Ingrese la casilla a jugar, ej: la casilla superior derecha es 1,3: "))

*#convertir la notacion de los jugadores a los indices de la matriz*

x=int(tupla[0])-1

y=int(tupla[2])-1

*#validar movimiento dentro del tablero*

**if** x>=0 **and** x<= 2 **and** y>=0 **and** y<=2:

*#validar movimiento no antes hecho*

**if** matriz[x,y]==0:

**if** turno:

matriz[x,y]=1

**else**:

matriz[x,y]=10

*#llamar la funcion del estado del juego*

estado = estadoDelJuego(matriz)

*#llamar el visualizador*

displayTriqui(matriz)

*#cambiar de turno*

turno=**not**(turno)

**else**:

print("Recuerde jugar en casillas vacias!")

**else**:

print("Recuerda que el tablero es de 3 por 3! ej: la casilla inferior izquierda es 3,1")

*# Publicar el resultado del juego*

**if** estado == 1 :

print("El jugador X ha ganado!")

**else**:

**if** estado == 2 :

print("El jugador O ha ganado!")

**else**:

print("No hubo ganadores!")

# ¿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.

# ¿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.

# 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.

# Primeros pasos de Pandas

## 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:

In [ ]:

! pip 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:

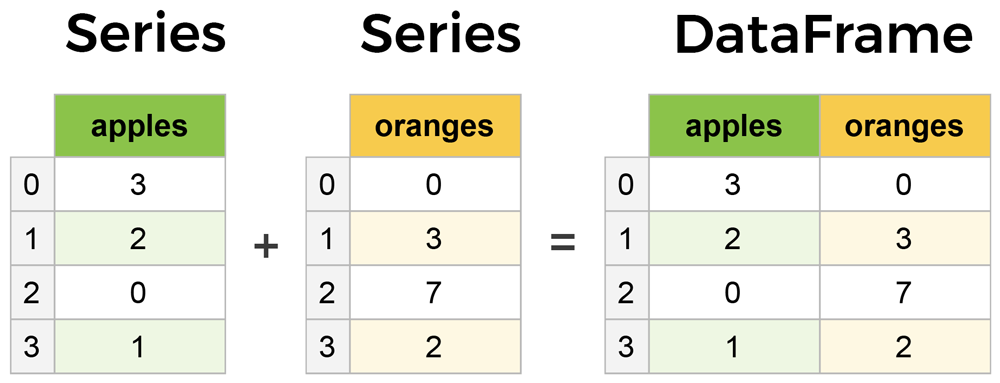
In [ ]:

**import** **pandas** **as** **pd**

# 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.



# 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):

In [ ]:

ventas = pd.Series([15,12,21], index = ["Ene", "Feb", "Mar"])

ventas

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

In [ ]:

ventas[0]

o con su etiqueta, si la tiene:

In [ ]:

ventas["Ene"]

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:

In [ ]:

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:

In [ ]:

ventas.index

In [ ]:

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:

In [ ]:

ventas.name = "Ventas 2020"

ventas.name

In [ ]:

ventas

In [ ]:

ventas.index.name = "Meses"

In [ ]:

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):

In [ ]:

ventas.axes

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

In [ ]:

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>

# 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:

In [ ]:

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

Y luego páselo al constructor de Pandas DataFrame:

In [ ]:

compras = pd.DataFrame( datos )

compras

### ¿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:

In [ ]:

compras = pd.DataFrame( datos , index = [ 'Juno' , 'Robert' , 'Lily' , 'David' ])

compras

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

In [ ]:

compras.index

In [ ]:

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.

In [ ]:

compras.index

In [ ]:

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):

In [ ]:

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:

In [ ]:

compras.index.name = "Clientes"

In [ ]:

compras.columns.name = "Frutas"

In [ ]:

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:

In [ ]:

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:

In [ ]:

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>

# 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.

### Utilizando una lista

In [ ]:

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:

In [ ]:

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 coindicir 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.

### Utilizando un diccionario

In [ ]:

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:

In [ ]:

d = {"Ene":7, "Feb":5, "Mar":3 }

s = pd.Series(d, index = ["Abr", "Mar", "Feb", "Ene"],dtype=int)

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.

### 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:

In [ ]:

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

s

# 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:

### Utilizando un diccionario

In [ ]:

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:

In [ ]:

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:

In [ ]:

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.

### 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:

In [ ]:

**import** **numpy** **as** **np**

unidades\_Datos = np.array([[2, 5, 3, 2],

[4, 6, 7, 2],

[3, 2, 4, 1]])

unidades\_Datos

In [ ]:

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:

In [ ]:

unidades = pd.DataFrame(unidades\_Datos, index = [2015, 2016, 2017], columns = ["Ag", "Au", "Cu", "Pt"])

unidades

### Utilizando diferentes diccionarios

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

In [ ]:

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:

In [ ]:

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.

# 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>

# 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.

# 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:

In [ ]:

**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

In [ ]:

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

In [ ]:

almacén = pd.DataFrame({"entradas": entradas, "salidas": salidas})

almacén["neto"] = almacén.entradas - almacén.salidas

almacén

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:

In [ ]:

entradas.head()

In [ ]:

almacén.head()

# 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:

In [ ]:

entradas.tail()

In [ ]:

almacén.tail()

# 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:

In [ ]:

entradas.sample()

In [ ]:

almacén.sample(5)

# 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%.

Siguiendo con el ejemplo visto en la sección anterior:

In [ ]:

almacén.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.

# 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:

In [ ]:

almacén.info()

Solo los dataframes tienen implementado este método.

# 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:

In [ ]:

**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**:

In [ ]:

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:

In [ ]:

s.value\_counts(bins = 2)

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:

In [ ]:

**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"):

In [ ]:

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:

In [ ]:

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

s

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

In [ ]:

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

In [ ]:

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:

In [ ]:

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

s

In [ ]:

s[0]

# 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:

In [ ]:

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

s

In [ ]:

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):

In [ ]:

s[1:]

In [ ]:

s[:3]

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

In [ ]:

s["a":"c"]

In [ ]:

s[:"c"]

In [ ]:

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):

In [ ]:

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

In [ ]:

s["b":"c"]

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

In [ ]:

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:

In [ ]:

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

In [ ]:

s[1:3]

Sin embargo, algo como

In [ ]:

s[1]

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

In [ ]:

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

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

In [ ]:

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

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:

In [ ]:

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:

In [ ]:

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**

In [ ]:

s.get(1)

In [ ]:

s.get("b")

# 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:

In [ ]:

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

s

In [ ]:

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:

In [ ]:

s.loc[0]

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:

In [ ]:

s.loc[["d", "a"]]

Otra opción es pasar al método un rango:

In [ ]:

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.

# 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):

In [ ]:

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

s

In [ ]:

s.iloc[1]

In [ ]:

s.iloc[0]

In [ ]:

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-:

In [ ]:

s.iloc[-1]

In [ ]:

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:

In [ ]:

s.iloc[[2, 0]]

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

In [ ]:

s.iloc[[-2, 0]]

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

In [ ]:

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:

In [ ]:

s.iloc[:3]

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

In [ ]:

s.iloc[2:]

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

In [ ]:

s.iloc[1:-1]

# 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:

In [ ]:

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

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):

In [ ]:

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:

In [ ]:

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):

In [ ]:

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:

In [ ]:

s.loc[[**True**, **False**, **False**, **True**, **True**, **True**]]

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

In [ ]:

s.loc[s>2]

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

In [ ]:

s.iloc[[**True**, **False**, **False**, **True**, **True**, **True**]]

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

In [ ]:

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:

In [ ]:

type((s>2).values)

In [ ]:

(s>2).values

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

In [ ]:

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

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

# 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:

In [ ]:

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

In [ ]:

s.sample(3, random\_state = 18)

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

In [ ]:

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:

In [ ]:

s.sample(10, random\_state = 18, replace = **True**)

# 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::

In [ ]:

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

s

In [ ]:

s.pop(1)

In [ ]:

s

In [ ]:

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

**try**:

s.pop(18)

**except**:

print("Error")

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

In [ ]:

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

s

In [ ]:

s.pop("a")

In [ ]:

s

y no al implícito, lo que devolvería un error:

In [ ]:

**try**:

s.pop()

**except**:

print("Error")

# - - 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:

In [ ]:

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:

In [ ]:

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

In [ ]:

ventas["Entradas"]

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

In [ ]:

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:

In [ ]:

ventas["Entradas","Feb"]

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:

In [ ]:

ventas["Entradas"] = [33, 25, 40, 12]

ventas

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

In [ ]:

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:

In [ ]:

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

ventas

Los valores asignados pueden proceder del propio dataframe:

In [ ]:

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

ventas

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

In [ ]:

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.

### Uso de rangos

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

In [ ]:

**import** **numpy** **as** **np**

df = pd.DataFrame(np.arange(18).reshape([6, 3]),

index = ["a", "b", "c", "d", "d", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

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:

In [ ]:

df["b":"d"]

aunque en este caso la selección incluye tanto la fila correspondiente a la primera etiqueta como la fila correspondiente a la segunda.

In [ ]:

df["b":"d"]

También podemos obviar la inclusión del primer o del segundo valor, considerándose las filas desde el comienzo/hasta el final del dataframe:

In [ ]:

df[:3]

In [ ]:

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:

In [ ]:

df = pd.DataFrame(np.arange(18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

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:

In [ ]:

df = pd.DataFrame(np.arange(18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

df.get("A")

In [ ]:

df.get("D")

# 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:

In [ ]:

df = pd.DataFrame(np.arange(18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

### 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:

In [ ]:

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:

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = [1, 3, 0, 4],

columns = ["A", "B", "C"])

df

In [ ]:

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:

In [ ]:

df = pd.DataFrame(np.arange(18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

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:

In [ ]:

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:

In [ ]:

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

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

# 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:

In [ ]:

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:

In [ ]:

df.loc[:, "A"]

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

In [ ]:

df.loc["b"]

In [ ]:

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:

In [ ]:

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

# 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:

In [ ]:

df = pd.DataFrame(np.random.randint(0, 10, 18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

df.iloc[2]

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

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = [3, 2, 1, 0],

columns = ["A", "B", "C"])

df

In [ ]:

df.iloc[3]

In [ ]:

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

In [ ]:

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:

In [ ]:

df = pd.DataFrame(np.random.randint(0, 10, 18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

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):

In [ ]:

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):

In [ ]:

df.iloc[:3]

In [ ]:

df.iloc[4:]

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

# Extracción de filas y columnas

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

In [ ]:

df.iloc[3, 1]

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

# 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:

In [ ]:

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:

In [ ]:

df.columns.get\_loc("B")

In [ ]:

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:

In [ ]:

df.index.get\_loc("d")

In [ ]:

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:

In [ ]:

df.iloc[df.index.get\_loc("c"), 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:

In [ ]:

df.iloc[[5, 3], df.columns.get\_indexer(["C", "A"])]

# 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:

In [ ]:

df = pd.DataFrame(np.arange(18).reshape([6, 3]),

index = ["a", "b", "c", "d", "e", "f"],

columns = ["A", "B", "C"])

df

In [ ]:

mask = [**True**, **False**, **True**, **False**, **False**, **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:

In [ ]:

df[df.A > 7]

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:

In [ ]:

df.loc[df.B > 6]

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:

In [ ]:

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

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

# 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:

In [ ]:

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:

In [ ]:

df.sample(3, random\_state = 18)

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

In [ ]:

df.sample(2, random\_state = 18, axis = 1)

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

In [ ]:

df.sample(frac = 0.6, random\_state = 18)

# 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:

In [ ]:

df = pd.DataFrame(np.arange(15).reshape([3, 5]),

index = ["a", "b", "c"],

columns = ["A", "B", "C", "D", "E"])

df

In [ ]:

columna = df.pop("B")

columna

In [ ]:

df

# 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.

# Edición de series

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

In [ ]:

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

s

In [ ]:

s[0] = -1

In [ ]:

s["b"] = -2

In [ ]:

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:

In [ ]:

s[1:3] = 0

s

In [ ]:

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:

In [ ]:

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

s

In [ ]:

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:

In [ ]:

s["f"] = 0

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:

In [ ]:

s["f":"h"] = 0

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:

In [ ]:

s[["c", "a"]] = [1, 2]

s

Si utilizamos índices, el resultado es semejante:

In [ ]:

s[[1, 0]] = [20, 21]

s

# 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:

In [ ]:

s = pd.Series([1, 2, 3, 4, 5],

index = ["a", "b", "c", "d", "e"])

s

In [ ]:

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:

In [ ]:

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:

In [ ]:

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:

In [ ]:

s.index[[1, 3]]

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

In [ ]:

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

# Método Where

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):

In [ ]:

s = pd.Series(np.arange(0, 10))

s

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

In [ ]:

s.where(s % 2 == 0)

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:

In [ ]:

s.where(s % 2 == 0, -1)

# 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:

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = ["a", "b", "c", "d"],

columns = ["A", "B", "C"])

df

In [ ]:

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...

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = ["a", "b", "c", "d"],

columns = ["A", "B", "C"])

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:

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = ["a", "b", "c", "d"],

columns = ["A", "B", "C"])

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

df

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

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = ["a", "b", "c", "d"],

columns = ["A", "B", "C"])

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):

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = ["a", "b", "c", "d"],

columns = ["A", "B", "C"])

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

df

In [ ]:

Y en el segundo (de fila inexistente):

In [ ]:

df = pd.DataFrame(np.arange(12).reshape([4, 3]),

index = ["a", "b", "c", "d"],

columns = ["A", "B", "C"])

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

df

# 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:

In [ ]:

df = pd.DataFrame(np.arange(12).reshape(-1, 3), columns=["A", "B", "C"])

df

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

In [ ]:

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:

In [ ]:

df.where(df % 2 == 0, -df)

# 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:

In [ ]:

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:

In [ ]:

df.drop(["a", "c"], axis = 0)

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:

In [ ]:

df.drop(["a", "c"])

Para eliminar columnas, habría que indicar el eje correspondiente:

In [ ]:

df.drop(["B", "D"], axis = 1)

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:

In [ ]:

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

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

In [ ]:

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

# 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.

# Unión de series

Comencemos revisando las opciones disponibles para las series pandas

# 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:

In [ ]:

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:

In [ ]:

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:

In [ ]:

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:

In [ ]:

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:

In [ ]:

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.

# 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:

In [ ]:

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

b = pd.Series([10, 11, 12], index = ["f", "g", "h"])

In [ ]:

c = a.append(b)

c

Si el argumento ignore\_index toma el valor True, se ignoran las etiquetas de las series:

In [ ]:

c = a.append(b, ignore\_index = **True**)

c

# 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.

# 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:

In [ ]:

**import** **numpy** **as** **np**

df1 = pd.DataFrame(np.arange(9).reshape([3, 3]),

index = ["a", "b", "d"],

columns = ["A", "B", "C"])

df1

In [ ]:

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:

In [ ]:

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:

In [ ]:

pd.concat([df1, df2], axis = 1, sort = **False**)

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:

In [ ]:

pd.concat([df1, df2], join = "inner")

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

In [ ]:

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

...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:

In [ ]:

pd.concat([df1, df2], axis = 1, join = "inner", ignore\_index = **True**)

# 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:

In [ ]:

df1 = pd.DataFrame(np.arange(9).reshape([3, 3]),

index = ["a", "b", "d"],

columns = ["A", "B", "C"])

df1

In [ ]:

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:

In [ ]:

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).

f