El futuro digital es de todos

MinTIC

# Pandas – Selección con índices y etiquetas

OPERADO POR:

Universidad Tecnológica de Pereira

Mision TIC 2022

RUTA DE APRENDIZAJE 1

# Pandas - Selección con índices y etiquetas

## 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 [105]: 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 [106]: s[[True, False, False, True, True, False]]

Out[106]: 0    5
          3    7
          4    8
          dtype: int64
```

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]
```

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

Out[168]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | 3 | 4 | 5 |
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

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

```
In [169]: df.columns.get_loc("B")
Out[169]: 1

In [170]: df.columns.get_indexer(["A", "C"])
Out[170]: array([0, 2], dtype=int64)
```

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 [171]: df.index.get_loc("d")
Out[171]: 3

In [172]: df.index.get_indexer(["c", "e"])
Out[172]: array([2, 4], dtype=int64)
```

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 [173]: df.iloc[df.index.get_loc("c"), 2]
Out[173]: 8
```

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

Out[174]:

| | C | A |
|---|---|---|
| **f** | 17 | 15 |
| **d** | 11 | 9 |

## 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 [175]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

In [175]:
```
df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                  index = ["a", "b", "c", "d", "e", "f"],
                  columns = ["A", "B", "C"])
df
```

Out[175]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | 3 | 4 | 5 |
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

In [176]:
```
mask = [True, False, True, False, False, True]
df[mask]
```

Out[176]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **c** | 6 | 7 | 8 |
| **f** | 15 | 16 | 17 |

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 [177]: df[df.A > 7]
```

Out[177]:

| | A | B | C |
|---|---|---|---|
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

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 [178]: df.loc[df.B > 6]
```

Out[178]:

| | A | B | C |
|---|---|---|---|
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

```
In [178]: df.loc[df.B > 6]
```

Out[178]:

| | A | B | C |
|---|---|---|---|
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

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

Out[179]:

| | A | B | C |
|---|---|---|---|
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

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.sampl**e. 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 [180]: df = pd.DataFrame(np.arange(18).reshape([6, 3]),
                            index = ["a", "b", "c", "d", "e", "f"],
                            columns = ["A", "B", "C"])
          df
```

Out[180]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | 3 | 4 | 5 |
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |
| **e** | 12 | 13 | 14 |
| **f** | 15 | 16 | 17 |

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 [181]: df.sample(3, random_state = 18)
```

Out[181]:

| | A | B | C |
|---|---|---|---|
| **f** | 15 | 16 | 17 |

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

```
In [182]: df.sample(2, random_state = 18, axis = 1)
```

Out[182]:

| | A | B |
|---|---|---|
| **a** | 0 | 1 |
| **b** | 3 | 4 |
| **c** | 6 | 7 |
| **d** | 9 | 10 |
| **e** | 12 | 13 |
| **f** | 15 | 16 |

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

```
In [183]: df.sample(frac = 0.6, random_state = 18)
```

Out[183]:

| | A | B | C |
|---|---|---|---|
| **f** | 15 | 16 | 17 |
| **e** | 12 | 13 | 14 |
| **b** | 3 | 4 | 5 |
| **a** | 0 | 1 | 2 |

## 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 [184]:
```
df = pd.DataFrame(np.arange(15).reshape([3, 5]),
                  index = ["a", "b", "c"],
                  columns = ["A", "B", "C", "D", "E"])
df
```

Out[184]:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **a** | 0 | 1 | 2 | 3 | 4 |
| **b** | 5 | 6 | 7 | 8 | 9 |
| **c** | 10 | 11 | 12 | 13 | 14 |

In [185]:
```
columna = df.pop("B")
columna
```

Out[185]:
```
a     1
b     6
c    11
Name: B, dtype: int32
```

In [186]:
```
df
```

Out[186]:

| | A | C | D | E |
|---|---|---|---|---|
| **a** | 0 | 2 | 3 | 4 |
| **b** | 5 | 7 | 8 | 9 |
| **c** | 10 | 12 | 13 | 14 |

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

Out[197]: a    1
          b    2
          c    3
          d    4
          e    5
          dtype: int64

In [198]: r = s.drop("b")
          r

Out[198]: a    1
          c    3
          d    4
          e    5
          dtype: int64
```

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

Out[199]: b    2
          c    3
          e    5
          dtype: int64
```

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

Out[200]: a    1
          b    2
          c    3
          d    4
          e    5
          dtype: int64
```

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

Out[203]: 0    0
          1    1
          2    2
          3    3
          4    4
          5    5
          6    6
          7    7
          8    8
          9    9
          dtype: int32
```

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

Out[206]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | 3 | 4 | 5 |
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |

```
In [207]: df.iloc[1, 2] = -1
          df
```

Out[207]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | 3 | 4 | -1 |
| **c** | 6 | 7 | 8 |
| **d** | 9 | 10 | 11 |

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 [208]: 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
```

Out[208]:

| | A | B | C |
|---|---|---|---|
| **a** | 10 | 1 | 2 |
| **b** | 20 | 4 | 5 |
| **c** | 30 | 7 | 8 |
| **d** | 40 | 10 | 11 |

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 [209]: 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
```

Out[209]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | -1 | -2 | 5 |
| **c** | -3 | -4 | 8 |
| **d** | 9 | 10 | 11 |

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

```
In [210]: 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
```

Out[210]:

| | A | B | C |
|---|---|---|---|
| **a** | 0 | 1 | 2 |
| **b** | -1 | -1 | 5 |
| **c** | -1 | -1 | 8 |
| **d** | 9 | 10 | 11 |

El futuro digital es de todos

MinTIC

# GRACIAS

OPERADO POR:

Universidad Tecnológica de Pereira

MisiónTIC2022