

Módulo 3. Ordenamiento de datos

Introducción

La ordenación de datos es un proceso fundamental dentro del procesamiento de datos.

Necesitamos establecer criterios de ordenamiento de los datos para darle un sentido y claridad a un gran volumen de datos iniciales.

Por ejemplo, organizamos los datos en orden alfabético o en un orden numérico ascendente o descendente o podemos ordenar una columna que enumere los valores de las ventas de un producto en orden descendente, para observar las ventas del producto de la más alta a la más baja.

En programación, se utilizan métodos de ordenamiento. Por ejemplo, la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo con un criterio de ordenamiento. El ordenamiento puede realizarse basándose en el valor de algún campo en un registro.

En Python, una forma que tenemos para ordenar datos es a través de Pandas, gracias a sus *DataFrames* y *series*.

Las principales características de dichas librerías son:

- Define nuevas estructuras de datos basadas en los *arrays* de la librería NumPy pero con nuevas funcionalidades.
- Permite leer y escribir fácilmente ficheros en formato CSV, Excel y bases de datos SQL.
- Permite acceder a los datos mediante índices o nombres para filas y columnas.
- Ofrece métodos para reordenar, dividir y combinar conjuntos de datos.
- Permite trabajar con series temporales.
- Realiza todas estas operaciones de manera muy eficiente. (Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>).

Video de inmersión

Unidad 1. Orden de datos

Tema 1. Ordenar datos con Pandas

Pandas es una librería fundamental para el ordenamiento, tratamiento y análisis de datos cuando usamos Python. Nos proporciona la clase *DataFrame*, la cual permite almacenar los datos en un objeto similar a una tabla de una base de datos.

Luego de haber practicado y entendido en el módulo anterior las distintas técnicas para empezar a manipular datos en las librerías NumPy y Pandas, en este módulo, abordaremos técnicas que nos permitirán ordenar los *dataset* con miras a fases posteriores de análisis más profundos.

Una vez importado un conjunto de datos en Pandas, debemos pensar en **darles un orden lógico que nos permita trabajarlos de forma adecuada**, por ejemplo, ordenarlos según los valores de una o varias columnas o en un orden ascendente o descendente. Por ejemplo, quizás necesitamos que las columnas de productos y ventas aparezcan en orden descendente, comenzando por los productos que más se venden.

Veamos un ejemplo sencillo de ordenamiento para entrar en materia. **Vamos a crear una lista que está originalmente en orden de mayor a menor** y, usando la función «*sort()*», vamos a generar una **nueva lista en orden de menor a mayor**.

Ingresamos el siguiente código en el intérprete *online*: <https://www.python.org/shell/>

```
>>> import pandas as pd  
>>> a = [5, 2, 3, 1, 4]  
>>> a.sort()  
>>> a  
[1, 2, 3, 4, 5]
```

Vemos como, **al usar la función «*sort()*», hemos ordenado la lista en orden ascendente**.

Veamos, ahora, otra forma muy sencilla de hacer un ordenamiento en una lista con la función «*sorted()*». Usaremos la misma lista anterior.

Ingresamos el siguiente código en el intérprete:

```
>>> sorted([5, 2, 3, 1, 4])
```

```
[1, 2, 3, 4, 5]
```

Figura 1: Funciones «sort()» y «sorted()»

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
>>> 
```

Fuente: captura de pantalla de Python (Python 3.10.5).

Vemos como la función «sorted()» devuelve una lista ordenada a partir de los elementos del iterable cedido como argumento. Ejemplos de iterables serían listas, cadenas y tuplas.

Función «sorted()»

Sintaxis: sorted(iterable, *, key=None, reverse=False)

Parámetros:

- **iterable**: cuyos elementos se quieren ordenar.
- **key** (opcional): especifica la función de un único argumento que será utilizada para determinar el criterio de ordenación de los elementos. El valor por defecto es *None*, lo que implica que los elementos serán comparados directamente.
- **reverse** (opcional): argumento booleano que determina si el orden será creciente o decreciente.

Veamos otro ejemplo aplicando la función «sorted()» a una lista de datos alfabéticos.

Ingresamos el siguiente código en el intérprete (el intérprete acepta la comilla simple 'k'):

```
>>> import pandas as pd
>>> print(sorted(['k','f','c','h']))
['c', 'f', 'h', 'k']
```

Vemos cómo el código `print(sorted(['k','f','c','h']))` en forma directa ordenó la lista en orden alfabético.

Aplicaremos la función a un diccionario, vemos que devuelve la lista de claves ordenada.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd<<<sorted()>>>
>>> d = dict([(3, "tres"), (1, "uno"), (2, "dos")])
>>> print(sorted(d))
[1, 2, 3]
```

Dos diferencias claves entre «sort()» y «sorted()» son las siguientes:

- «sorted()» retornará una nueva lista. «sorted()» acepta cualquier objeto iterable.
- «sort()» ordena la lista en su lugar. «sort()» acepta únicamente listas.

(Nota: en programación, un iterable es un conjunto de elementos que permite que se retornen sus elementos).

El método sort() puede tomar dos argumentos opcionales llamados key y reverse.

- **Key** tiene el valor de la función que será llamada en cada ítem de la lista.
- **Reverse** tiene un valor booleano de True o False. (Torres, 2021, <http://bit.ly/3Yoj5gr>)

Veamos un ejemplo del uso de «key».

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd
>>> nombres=['Carolina','Luis','Leo','Jackeline','Penelope']
>>> print('Sin orden: ', nombres)
Sin orden: ['Carolina', 'Luis', 'Leo', 'Jackeline', 'Penelope']
```

Ahora vamos a ordenarlos del nombre más corto al más largo.

Podemos usar la función **len()** como el valor para el argumento **key=key=len**, esto indicará al programa ordenar la lista de nombres por longitud, del más corto al más largo.

Ingresamos el siguiente código en el intérprete (basándonos en la lista anterior nombres):

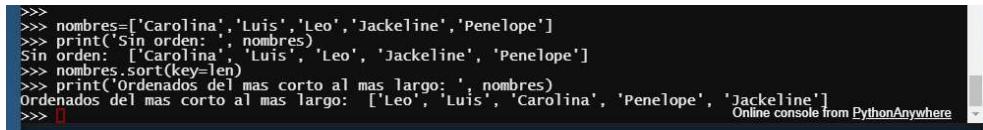
```
>>> nombres.sort(key=len)
```

```
>>> print('Ordenados del más corto al más largo: ', nombres)
```

Ordenados del más corto al más largo: ['Leo', 'Luis', 'Carolina', 'Penelope', 'Jackeline']

Vemos que los elementos de la lista nombres se han ordenado por la longitud de letras en el nombre. (Torres, 2021, <http://bit.ly/3Yoj5gr>)

Figura 2: Función «len()»



```
>>> nombres=['Carolina','Luis','Leo','Jackeline','Penelope']
>>> print('Sin orden: ', nombres)
Sin orden: ['Carolina', 'Luis', 'Leo', 'Jackeline', 'Penelope']
>>> nombres.sort(key=len)
>>> print('Ordenados del mas corto al mas largo: ', nombres)
Ordenados del mas corto al mas largo: ['Leo', 'Luis', 'Carolina', 'Penelope', 'Jackeline']
>>> 
```

Fuente: captura de pantalla de Python (Python 3.10.5).

El argumento «reverse» del método «sort()» tiene un valor booleano de «True» o «False».

Veamos un ejemplo del uso de reverse.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd
>>> nombres=['Carolina','Luis','Leo','Jackeline','Penelope']
>>> print('Sin orden: ', nombres)
Sin orden: ['Carolina', 'Luis', 'Leo', 'Jackeline', 'Penelope']
```

Ahora, «reverse=True» ordenará la lista nombres **en orden alfabético inverso**:

Ingresamos el siguiente código en el intérprete:

```
>>> nombres.sort(reverse=True)
>>> print('Ordenados en orden alfabético inverso: ', nombres)
Ordenados en orden alfabético inverso: ['Penelope', 'Luis', 'Leo', 'Jackeline', 'Carolina']
```

Vemos que los elementos de la lista nombres se han ordenado en orden alfabético inverso.

Con la función «sorted()» también podemos utilizar los argumentos «key» y «reverse».

Aplicación de la función «sorted()» en tuplas.

Podemos usar la función «sorted()» en tuplas: tuples. En Python, una tupla es un conjunto ordenado e inmutable de elementos del mismo o diferente tipo. Las tuplas se representan

escribiendo los elementos entre paréntesis y separados por comas.

Veamos un ejemplo de **aplicación de la función «sorted()» en tuplas**.

Tenemos una colección de tuplas que representan el nombre de estudiantes de aplicaciones de informática en un instituto, su edad y la aplicación que estudian.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd  
>>> Estudiantes=[('Carolina', 19, 'Word'), ('Jorge', 18, 'Excel'), ('Alberto', 20, 'Photoshop')]  
Estudiantes=[('Carolina', 19, 'Word'), ('Jorge', 18, 'Excel'), ('Alberto', 20, 'Photoshop')]
```

Podemos usar el método «sorted()» para ordenar estos datos por la edad del estudiante. La **«key» tiene el valor lambda de la función**, lo que le indica al programa ordenar los elementos en orden ascendente.

Las expresiones lambda en Python son una forma corta de declarar funciones pequeñas y anónimas (no es necesario proporcionar un nombre para las funciones lambda). Podemos definir este tipo de función usando la palabra reservada lambda.

Ingresamos el siguiente código en el intérprete:

```
>>> print(sorted(Estudiantes, key=lambda estudiante: estudiante[1]))  
[('Jorge', 18, 'Excel'), ('Carolina', 19, 'Word'), ('Alberto', 20, 'Photoshop')]
```

Vemos que los estudiantes aparecen ordenados por edad, de menor a mayor.

Ahora, utilicemos el **argumento booleano «reverse»**, que determina si el orden será creciente o decreciente, con la función «sorted()» para ordenar los elementos de la lista de estudiantes en orden alfabético inverso del nombre de las aplicaciones de informática que estudian.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd  
>>> Estudiantes=[('Carolina', 19, 'Word'), ('Jorge', 18, 'Excel'), ('Alberto', 20, 'Photoshop')]  
>>> print(sorted(Estudiantes, key=lambda estudiante: estudiante[2], reverse=True))  
[('Carolina', 19, 'Word'), ('Alberto', 20, 'Photoshop'), ('Jorge', 18, 'Excel')]
```

Vemos que los estudiantes aparecen **ordenados en orden alfabético inverso del nombre de las aplicaciones de informática** que estudian.

Ordenar un *DataFrame* de Pandas por una columna que contenga fechas usando la función «`sort_values()`».

Ingresamos el siguiente código en el intérprete.

Creamos el siguiente *DataFrame* de Pandas (recordemos mantener la indentación):

```
>>> import pandas as pd  
>>> df = pd.DataFrame ({'ventas': [4, 11, 13],  
                      'clientes': ['Castillo','Alvares','Quiroga'],  
                      'fecha': ['2022-10-23', '2022-01-20', '2022-07-22']})  
  
>>> df  
ventas clientes fecha  
0 4 Castillo 2022-10-23  
1 11 Alvares 2022-01-20  
2 13 Quiroga 2022-07-22
```

Primero, vamos a aplicar la función «`to_datetime()`» para convertir la columna fecha en un objeto de fecha y hora.

Ingresamos el siguiente código en el intérprete:

```
df['fecha']=pd.to_datetime(df['fecha'])
```

Ahora, vamos a ordenar el mismo DataFrame según la columna fecha usando la función «`sort_values()`».

Ingresamos el siguiente código en el intérprete:

```
>>> df.sort_values('fecha')  
ventas clientes fecha  
1 11 Alvares 2022-01-20  
2 13 Quiroga 2022-07-22  
0 4 Castillo 2022-10-23
```

Vemos que los clientes aparecen ordenados por fecha, de la más antigua a la más reciente.

Pregunta 1

Si ingresamos:

```
>>> sorted([25,13, 21, 33, 1, 14])
```

¿Cuál de las siguientes opciones será la salida correcta?

[25,13, 21, 33, 1, 14]

[1, 13, 14, 21, 25, 33]

[33, 25, 21, 13, 14, 1]

[33, 25, 21, 13, 14, 1]

Justificación

Tema 2. Agregaciones

Una parte esencial del análisis de conjuntos de *datasets* es poder obtener un resumen, computar agregaciones como «sum()», «mean()», «mediana()», «min()» y «max()», en las que un solo número tiene el poder de reflejar una idea de la naturaleza de un gran conjunto de datos.

En este tema, estudiaremos agregaciones en Pandas. Esta es un área muy amplia, ya que abarca desde operaciones simples similares a las que hemos visto en matrices NumPy hasta operaciones más sofisticadas. En este punto, solo podremos darnos una idea amplia sobre esto.

La función «pandas.DataFrame.aggregate()» agrega las columnas o filas de un **DataFrame**.

Las funciones de agregación más utilizadas son «min()», «max()» y «sum()».

Sintaxis de «pandas.DataFrame.aggregate()»:

```
DataFrame.aggregate(func,
```

```
    axis,
```

```
    *args,
```

```
    **kwargs)
```

Parámetros

«**Func**»: es la función de agregación que debe aplicarse. Por ejemplo, puede ser una lista de *string* o un diccionario.

axis: 0 por defecto. Si es 0 o *índex*, entonces, la función se aplica a las columnas individuales. Si es 1 o *columns*, entonces, la función se aplica a las filas individuales.

***args**: es un argumento posicional.

****kwargs**: es un argumento de palabras clave.

Esta función devuelve un escalar, Series o un *DataFrame*.

- Devuelve un escalar si se llama a una sola función con «Series.agg()».
- Devuelve una Series si una función simple es llamada con «DataFrame.agg()».
- Devuelve un *DataFrame* si se llaman múltiples funciones con «DataFrame.agg()».

Las operaciones de agregación se realizan siempre sobre un eje, ya sea el índice (por defecto) o el eje de la columna.

«**DataFrame.agg()**» es un alias de «**DataFrame.aggregate()**». Utilizaremos «**DataFrame.agg()**» en los códigos de los ejercicios.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd  
>>> dataframe=pd.DataFrame({'Asistencia':{0:60,1:100,2:80}, 'Nombre':{'0':'Jose', 1:  
'Pedro', 2:'Carmen'},  
    'Calificaciones': {0:90,1:75,2:82}})  
>>> print(dataframe)
```

Asistencia Nombre Calificaciones

0	60	José	90
1	100	Pedro	75
2	80	Carmen	82

Usaremos la función «DataFrame.agg()**» utilizando solo una función de agregación.**

Ingresamos el siguiente código:

```
>>> dataframe1 = dataframe.agg('sum')  
>>> print(dataframe1)  
Asistencia      240  
Nombre        JosePedroCarmen  
Calificaciones   247  
dtype: object
```

Vemos que la función de agregación «sum» se aplicó a las columnas individuales.

Para las columnas de tipo entero (asistencia y calificaciones), ha generado la suma y, para la columna de tipo *string* (nombre), ha concatenado las cadenas. El «*dtype:object*» muestra que se devuelve una Series.

Usaremos la función «DataFrame.agg()» utilizando dos funciones de agregación: «sum» y «min».

Ingresamos el siguiente código:

```
>>> dataframe1=dataframe.agg(['sum','min'])  
>>> print(dataframe1)  
Asistencia    Nombre Calificaciones  
sum    240 JosePedroCarmen  247  
min     60    Carmen     75
```

Vemos que la función de agregación «sum» se aplicó a las columnas individuales y que la función «min» generó los valores mínimos de las columnas de números.

Utilización de «DataFrame.aggregate()» con una columna especificada.

Siguiendo con el mismo *DataFrame* anterior:

```
>>> import pandas as pd  
>>>           dataframe=pd.DataFrame({'Asistencia':{0:60,1:100,2:80}, 'Nombre':{0:'Jose',1:  
'Pedro',2:'Carmen'},  
'Calificaciones': {0:90,1:75,2:82}})
```

Ingresamos el siguiente código en el intérprete:

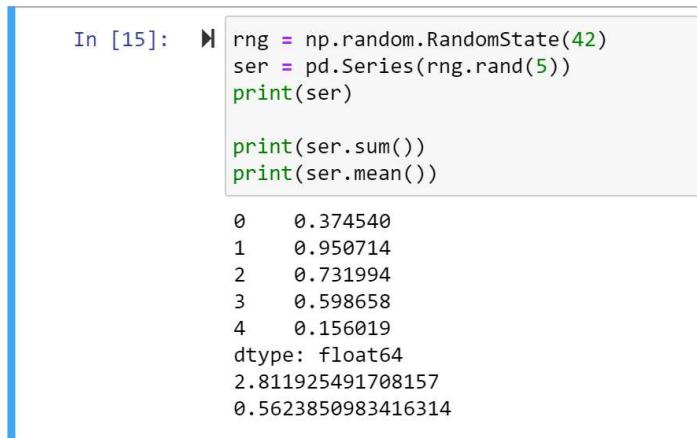
```
>>> import pandas as pd  
>>> dataframe1=dataframe.agg({'Calificaciones':['sum','max']})  
>>> print(dataframe1)  
Calificaciones  
sum    247  
max    90
```

Vemos que las funciones «sum» y «max» se aplicaron solo a la columna especificada 'calificaciones'.

Agregaciones simples con Pandas

Al igual que un arreglo unidimensional de NumPy, para una serie de Pandas, la agregación devuelve un valor único:

Figura 3: Agregación en Pandas, valor único



```
In [15]: rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
print(ser)

print(ser.sum())
print(ser.mean())
```

0 0.374540
1 0.950714
2 0.731994
3 0.598658
4 0.156019
dtype: float64
2.811925491708157
0.5623850983416314

Fuente: captura de pantalla de Python (Python 3.10.5).

Para un *DataFrame*, una agregación devuelve un resultado por columna.

Figura 4: Agregación para un *DataFrame*

```
In [21]: df = pd.DataFrame({'A': rng.rand(5),
   'B': rng.rand(5)})
print(df)

print("\nPromedio por columna: ")
df.mean()

          A      B
0  0.921874  0.388677
1  0.088493  0.271349
2  0.195983  0.828738
3  0.045227  0.356753
4  0.325330  0.280935

Promedio por columna:

Out[21]: A    0.315381
          B    0.425290
          dtype: float64
```

Fuente: captura de pantalla de Python (Python 3.10.5).

Función «describe()»

La función «**describe()**» de Pandas es muy útil y poderosa, ya que regresa estadísticas descriptivas entre las que se incluyen la **media, la mediana, el máximo, el mínimo, std y conteos para una columna en particular de los datos**. La función «**describe()**» solo regresa los valores de estas estadísticas para las columnas numéricas.

Vamos a ver ejemplos de su uso. Primero, debemos conocer su sintaxis.

Sintaxis de «pandas.DataFrame.describe()»

```
DataFrame.describe(percentiles=None,
                   include=None,
                   exclude=None,
                   datetime_is_numeric=False)
```

Parámetros

Percentiles: este parámetro indica los percentiles a incluir en la salida. Todos los valores deben estar entre 0 y 1. El valor por defecto es [.25, .5, .75], que devuelve los percentiles 25, 50 y 75.

Include: especifica los tipos de datos a incluir en la salida. Tiene opciones.

all: todas las columnas de la entrada se incluirán en la salida.

Una lista de tipos de datos: limita los resultados a los tipos de datos proporcionados.

None: el resultado incluirá todas las columnas numéricas.

Exclude: especifica los tipos de datos a excluir de la salida. Tiene dos opciones.

Una lista de tipos de datos: excluye del resultado los tipos de datos proporcionados.

None: el resultado no excluirá nada.

Datetime_is_numeric: un parámetro booleano. Indica si se deben tratar los tipos de datos *datetime* como numéricos. (Noor, 2023, <http://bit.ly/3mx79Mk>)

El retorno de la función «describe()» es: devuelve el resumen de estadísticas de la Series o DataFrame pasado.

Veamos un ejemplo del uso de la función «describe()» para encontrar las estadísticas de un DataFrame.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd  
>>> dataframe=pd.DataFrame({'Asistencia':{0:60,1:100,2:80}, 'Nombre':{'0':'Jose', 1:  
'Pedro',2:'Carmen'},  
    'Calificaciones': {0:90,1:75,2:82}})  
>>> print(dataframe)  
Asistencia Nombre Calificaciones  
0    60   Jose     90  
1   100  Pedro     75  
2    80 Carmen     82
```

Ahora, vamos a aplicar la función «describe()» ingresando el siguiente código en el intérprete:

```
>>> dataframe1=dataframe.describe()
```

```
>>> print('Estadísticas: \n')
```

```
>>> print(dataframe1)
```

Estadísticas:

Asistencia Calificaciones

count 3.0 3.000000

mean 80.0 82.333333

std 20.0 7.505553

min 60.0 75.000000

```
25%    70.0  78.500000
50%    80.0  82.000000
75%    90.0  86.000000
max    100.0 90.000000
```

Vemos que nos muestra las estadísticas de las dos columnas numéricas.

Veamos otro ejemplo del método «describe()» .

Figura 5: Función «describe()»

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Fuente: captura de pantalla de Python (Python 3.10.5).

Vemos como nos permitió contar, promediar, hacer la desviación estándar, sacar el mínimo y el máximo y los percentiles.

Al agregar «dropna()», se omiten los valores faltantes o *missing values*.

La función «pandas.DataFrame.dropna()» elimina los valores nulos (valores perdidos) del DataFrame y deja caer las filas o columnas que contienen los valores nulos.

NaN (no un número) y NaT (no un tiempo) representan los valores nulos. «DataFrame.dropna()» detecta estos valores y filtra el DataFrame en consecuencia.

Sintaxis de «pandas.DataFrame.dropna()»

```
DataFrame.dropna(axis,
                  how,
```

```
thresh,  
subset,  
inplace)
```

Funciones de este tipo nos ayudan a entender de forma rápida y a un alto nivel las principales características y propiedades del *dataset*.

Existen otros métodos de agregación, como los siguientes.

Tabla 1: Agregación y descripción

Agregación	Descripción
«count()»	Número total de ítems
«first()», «last()»	Primer y último ítem
«mean()», «median()»	Promedio y mediana
«min()», «max()»	Mínimo y máximo
«std()», «var()»	Desviación estándar y varianza
«mad()»	Desviación absoluta media
«prod()»	Producto de todos los ítems
«sum()»	Suma de todos los ítems

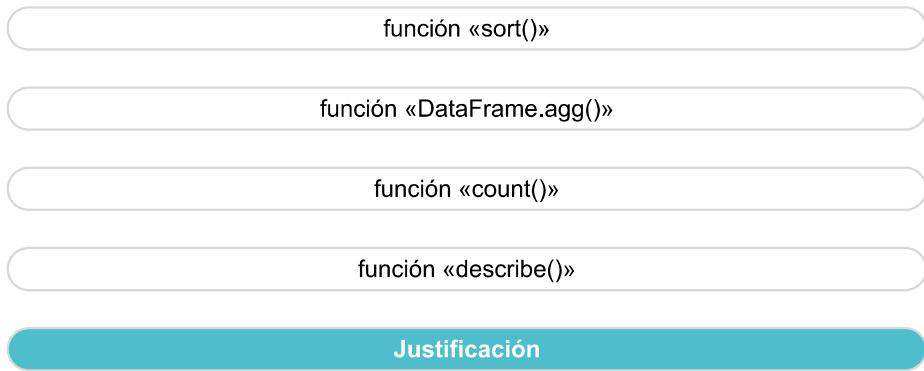
Fuente: elaboración propia.

De todas formas, en la práctica, muchas veces, las agregaciones simples quedan cortas a la hora de profundizar en los datos. Por esta razón, necesitamos pasar al siguiente nivel, el de las **agrupaciones**.

Pregunta 2

Seleccionar a qué opción corresponde la siguiente afirmación:

“Devuelve el resumen de estadísticas de la Series o Dataframe al cual se aplica”:



Tema 3. Agrupaciones

Las agrupaciones simples pueden darnos una idea de la naturaleza del *dataset*, pero es muy probable que **necesitemos agrupar los datos de un *DataFrame* condicionalmente según alguna etiqueta particular o índice, de acuerdo con los valores de una o varias columnas (categorías)**, como por ejemplo, el sexo o el país.

Figura 6: División de grupos por sexo

División en Grupos

Nombre	Sexo	Edad
Carmen	Mujer	22
Luis	Hombre	18
María	Mujer	25
Pedro	Hombre	30

Sexo = Hombre	
Nombre	Edad
Luis	18
Pedro	30

Sexo = Mujer	
Nombre	Edad
Carmen	22
María	25

Fuente: Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>

Función «groupby()»

Es un método muy importante para generar grupos resumiendo la información. Las agrupaciones se pueden realizar sobre el mismo *DataFrame* o crear otro nuevo.

«groupby()» viene de group by o agrupar por. Sirve para agrupar datos, según un determinado criterio, y aplicar operaciones sobre los elementos del conjunto conformado.

Los objetos Pandas se pueden dividir en cualquiera de sus ejes.

Veamos, primero, la sintaxis de la función «DataFrame.groupby()»

La sintaxis de «pandas.DataFrame.groupby()»

```
DataFrame.groupby(  
    by=None,  
    axis=0,  
    level=None,  
    as_index = True,  
    sort = True,  
    group_keys = True,  
    squeeze: bool = False,  
    observed: bool = False)
```

Parámetros

By: mapeo, función, cadena, *label* o iterable para agrupar elementos

Axis: agruparse junto con la fila (axis=0) o la fila (axis=1)

Level: Entero. Valor para agrupar por un nivel o niveles particulares

As_index: Booleana. Devuelve un objeto con etiquetas de grupo como índice

Sort: Booleana. Ordena las claves de grupo

Group_keys: Booleana. Añade claves de grupo a los índices para identificar las piezas

Squeeze: Booleana. Disminuye la dimensión del retorno cuando es posible

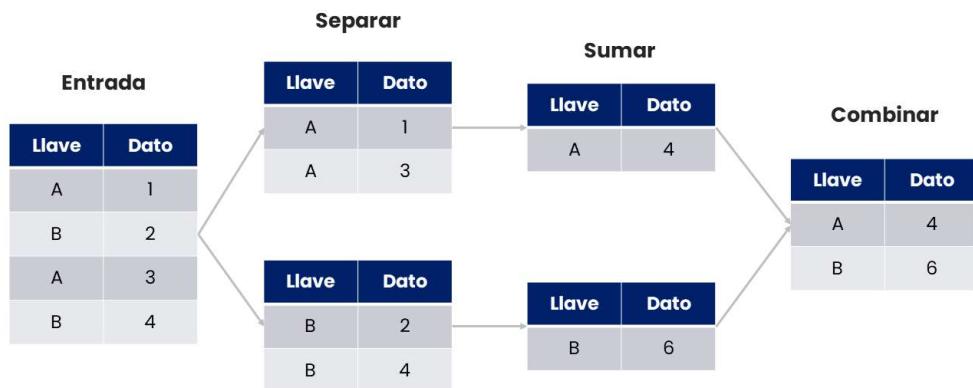
Observed: Booleana. Solo muestra valores observados si se establece en True. (Joshi, 2023, <http://bit.ly/3JjPHE2>)

La función «DataFrame.groupby()» devuelve un objeto «DataFrameGroupBy» que contiene la información agrupada.

La sintaxis de «groupby()» indica entre paréntesis el conjunto de datos que se quiere agrupar y, después, se puede seleccionar una serie entre corchetes para aplicar un método básico estadístico de Pandas como «sum()».

La función «`DataFrame.groupby()`» se puede interpretar como una secuencia de tres acciones por separado: separar, sumar y combinar, representado en el siguiente esquema.

Figura 7: Descomposición de la función «`groupby()`»



Fuente: elaboración propia.

Ahora que hemos comprendido el concepto de la función «`DataFrame.groupby()`», vamos a pasar a la práctica, haremos ejercicios en el intérprete *online*: <https://www.python.org/shell/>. Vamos, primero, a crear un *DataFrame* que contenga lo siguiente:

- una lista de frutas en un abasto
- su precio
- el dato de si están o no disponibles en inventario.

Al final, vamos a obtener un **listado solo de aquellas frutas que sí están en inventario**, usando la función «`DataFrame.groupby()`»

Usaremos # para escribir comentarios, que es la forma de hacerlo en Python.

Ingresamos el siguiente código en el intérprete para crear el *DataFrame*:

```
>>> import pandas as pd  
>>> # creamos la lista de frutas  
>>> lista_frutas = [('Mango',64, 'No') ,  
    ('Naranja', 14, 'Sí') ,  
    ('Banana', 20, 'No') ,  
    ('Manzana', 12, 'Sí') ,  
    ('Melón', 34, 'No') ,
```

```
('Kiwi', 84, 'Sí')]
```

```
>>> # creamos el DataFrame, a partir de la lista de frutas, indicando el nombre de las columnas
>>> df = pd.DataFrame(lista_frutas, columns = ['Fruta','Precio','En Inventario'])
>>> print(df)

  Fruta  Precio  En inventario
0  Mango     64        No
1  Naranja    14       Sí
2  Banana     20        No
3  Manzana    12       Sí
4  Melón      34        No
5   Kiwi      84       Sí
```

Vemos que «`print(df)`» nos muestra el contenido del *DataFrame* con la primera columna (índice), la columna fruta, la columna precio y la columna en inventario.

Es fácil observar que la naranja, la manzana y el kiwi sí están en inventario. Sin embargo, pensemos en **un DataFrame con una gran cantidad de productos** (centenares o miles), donde necesitamos saber rápidamente cuáles están en inventario. Si tomamos los datos de archivos de una empresa (ya no un abasto), aplicamos la función «`DataFrame.groupby()`»

Ahora, vamos a agrupar el *DataFrame* en grupos basados en los valores de la columna 'en inventario':

Para esto ingresamos el siguiente código:

```
>>> grouped_df = df.groupby('En Inventario')
>>> # solicitamos un listado de aquellas frutas que tienen el dato Sí en la columna en inventario
>>> print(grouped_df.get_group('Sí'))

  Fruta  Precio  En inventario
1  Naranja    14       Sí
3  Manzana    12       Sí
5   Kiwi      84       Sí
```

Nos muestra que solo la naranja, la manzana y el kiwi están en inventario.

Podríamos hacer lo mismo, pero para obtener **un listado de aquellas frutas que no están en**

inventario, simplemente ingresando lo siguiente:

```
>>> print(grouped_df.get_group('No'))
```

Fruta Precio En inventario

0	Mango	64	No
2	Banana	20	No
4	Melón	34	No

Vemos la potencia de la **función «DataFrame.groupby()»**. A través de un código simple, podemos obtener rápidamente un resultado agrupado según nuestra necesidad.

Pensemos en una empresa con centenares o miles de productos, están los datos en archivos y necesitamos obtener una diversidad de información rápidamente.

Aquí es donde comprendemos la potencia que tiene el lenguaje Python en el procesamiento de datos y, por esto, es uno de los lenguajes de programación que debemos manejar.

Veamos otro ejemplo. Tenemos el siguiente DataFrame:

Figura 8: Ejemplo de la función «DataFrame.groupby()»

```
In [36]: df = pd.DataFrame({'Llave': ['A', 'B', 'C', 'A', 'B', 'C'],  
                           'Dato': range(6)}, columns=['Llave', 'Dato'])  
df
```

	Llave	Dato
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

Fuente: elaboración propia.

Entonces aplicamos **«groupby()»** a la columna llave:

Figura 9: Resultado del ejemplo de la función «DataFrame.groupby()»

```
In [37]: print(df.groupby('Llave'))  
df.groupby('Llave').sum()
```

```
Out[37]:
```

Llave	Dato
A	3
B	5
C	7

Fuente: elaboración propia.

El `print` arroja como resultado un objeto del tipo «**DataFrameGroupBy**».

Vemos que agrupó los datos de la columna llave y sumó los valores contenidos en la columna dato para cada uno de esos grupos (A, B, C).

En el caso del ejemplo es «`sum()`», pueden obtenerse diversos agrupamientos, de acuerdo con lo que se requiera.

Se puede usar cualquiera de las funciones que hemos revisamos anteriormente.

En el siguiente ejemplo, se utiliza la función «`groupby()`» con «`median()`» para obtener la mediana o valor medio de diversos datos de astronomía (la mediana es el valor central de un grupo de números ordenados por tamaño).

Figura 10: Obtener la mediana

```
In [39]: planets.groupby('method')['orbital_period'].median()
```

method	orbital_period
Astrometry	631.180000
Eclipse Timing Variations	4343.500000
Imaging	27500.000000
Microlensing	3300.000000
Orbital Brightness Modulation	0.342887
Pulsar Timing	66.541900
Pulsation Timing Variations	1170.000000
Radial Velocity	360.200000
Transit	5.714932
Transit Timing Variations	57.011000
Name: orbital_period, dtype: float64	

Fuente: elaboración propia.

En el ejemplo anterior, se muestran valores de **período orbital** agrupados por método, obteniendo la mediana de los valores.

Vemos la potencia que tiene el lenguaje Python en el campo del procesamiento de datos y,

por esto, se utiliza cada vez más en áreas que manejan grandes volúmenes de datos, como es el caso de la ciencia.

Te invitamos a ver el siguiente video sobre agrupaciones que nos muestra el potencial de las herramientas de Python, como la función «groupby()» para generar grupos que ayudan a resumir y, así, generar información.

No olvidemos que, en el procesamiento de datos, debemos obtener información de los datos originales.

Video 1. agrupaciones

Fuente: OpenWebinars. (2020, 4 de noviembre). Agrupaciones de datos con Pandas [Archivo de video]. YouTube. <https://www.youtube.com/watch?v=hGQB9nAbjhI>

Pregunta 3

Seleccionar a qué opción corresponde la siguiente afirmación:

“La función «DataFrame.groupby()» se puede interpretar como una secuencia de tres acciones por separado”:

Separar, sumar y combinar.

Sumar, separar y combinar.

Combinar, sumar y separar.

Combinar, separar y sumar.

Justificación

Tema 4. Tablas pivot

Estudiaremos ahora las tablas *pivot* o tablas dinámicas. Quizás resulten familiares por las planillas de Excel u otros programas que operan con datos tabulares.

Una tabla dinámica (*pivot table* en inglés) es una tabla-resumen (datos dispuestos en filas y columnas) que agrupa datos procedentes de otra tabla o base de datos de mayor tamaño. Se usa como herramienta para el procesamiento de grandes cantidades de datos.

Toma datos de columnas simples como entrada y agrupa las entradas en una tabla bidimensional que proporciona un resumen multidimensional de los datos.

“La operación *pivot* permite construir un nuevo *DataFrame* a partir de otro, de forma que los valores de una columna pasen a ser nombres de columna del nuevo *DataFrame*” (Cursos informática, s. f., <http://bit.ly/3kP5tNM>).

Podemos generar una tabla dinámica con «groupby()» pero sería algo del tipo “hecho a mano”. Para resolver esto, Pandas cuenta con la función «pivot_table».

Esta función, al igual que en Excel, es extremadamente útil para **resumir conjuntos de datos de una forma rápida y eficaz**.

Antes de usarla, vamos a conocer su **sintaxis**:

```
pandas.pivot_table(data,  
                    values= None,  
                    index= None,  
                    columns= None,  
                    aggfunc= 'mean',  
                    fill_value= None,  
                    margins= False,  
                    dropna= True,  
                    margins_name= 'All',  
                    observed= False,  
                    sort= True)
```

Parámetros

(Los valores por defecto de todos los parámetros aparecen arriba).

Data: es el *DataFrame* del que queremos eliminar los datos repetidos.

Values: representa la columna para agregar.

Index: es una column, grouper, matriz, o una lista. Representa la columna de datos que queremos como índice, es decir, como filas.

Columns: es una column, grouper, matriz, o una lista. Representa la columna de datos que queremos como columnas en nuestra tabla pivote de salida.

aggfunc: es una función, una lista de funciones o un diccionario. Representa la función agregada que se aplicará a los datos. Si se pasa una lista de funciones agregadas, habrá

una columna para cada función agregada en el cuadro resultante con el nombre de la columna en la parte superior.

fill_value: es un escalar. Representa el valor que sustituirá a los valores que faltan en la tabla de salida

margins: es un valor booleano. Representa la fila y la columna generadas después de tomar la suma de la respectiva fila y columna.

dropna: es un valor booleano. Elimina las columnas cuyos valores son NaN de la tabla de salida.

margins_name: es una cuerda. Representa el nombre de la fila y la columna generada si el valor de los margins es True.

observed: es un valor booleano. Por defecto es False. Si es True, muestra los valores observados para los grupos categóricos.

sort: booleano, por defecto es True.

Devuelve el resumen del *DataFrame*. (Noor, 2023, <http://bit.ly/3SR5PQu>)

La principal función del «Pivot_table» son las agrupaciones de datos a las que se les suelen aplicar funciones matemáticas como sumatorias, promedios, etc. Si no indicamos en el parámetro «aggfunc» qué operación queremos hacer, por defecto, nos calculará la media de todas aquellas columnas que sean de tipo numérico.

La función «pandas.pivot_table()» evita la repetición de datos del *DataFrame*. Resume los datos y aplica diferentes funciones de agregación a los datos.

Veamos, a continuación, cómo aplicar esta función a un *DataFrame* para iniciarnos en la construcción de tablas dinámicas en Python con Pandas.

Primero, vamos a crear un *DataFrame* con:

nombre de estudiantes de JavaScript

fecha de la evaluación

puntuación en ejercicios en JavaScript

Ingresamos el siguiente código en el intérprete para crear el *DataFrame*:

```
>>> import pandas as pd
```

```
>>> dataframe = pd.DataFrame({
```

```
>>>     "Nombre":  
>>>     ["Ana",  
        "Ana",  
        "Ana",  
        "Ana",  
        "María",  
        "María",  
        "María",  
        "María"],  
  
    "Fecha":  
        ["03-06-2022",  
         "04-06-2022",  
         "03-06-2022",  
         "04-06-2022",  
         "03-06-2022",  
         "04-06-2022",  
         "03-06-2022",  
         "04-06-2022"],  
  
    "Puntuación en JavaScript":  
        [10,  
         2,  
         4,  
         6,  
         8,  
         9,  
         1,  
         10]  
    })  
  
>>> print(dataframe)  


|   | Nombre | Fecha      | Puntuación en JavaScript |
|---|--------|------------|--------------------------|
| 0 | Ana    | 03-06-2022 | 10                       |
| 1 | Ana    | 04-06-2022 | 2                        |
| 2 | Ana    | 03-06-2022 | 4                        |
| 3 | Ana    | 04-06-2022 | 6                        |
| 4 | María  | 03-06-2022 | 8                        |


```

5	María	04-06-2022	9
6	María	03-06-2022	1
7	María	04-06-2022	10

Vemos que hemos creado el *DataFrame* y «print(dataframe)» muestra sus columnas y los datos.

Ahora, vamos a aplicar la función «pandas.pivot_table()» para construir un nuevo *DataFrame* a partir del original, de forma que los valores de una columna pasen a ser nombres de columna del nuevo *DataFrame*, generando una tabla-resumen que agrupa datos procedentes de la tabla original.

Ingresamos el siguiente código:

```
>>> pivotTable=pd.pivot_table(dataframe,index="Nombre", columns= "Fecha")
>>> # hemos aplicado la función pivot_table() al dataframe original
>>> print(pivotTable)
```

Puntuación en JavaScript

Fecha 03-06-2022 04-06-2022

Nombre

Ana 7.0 4.0

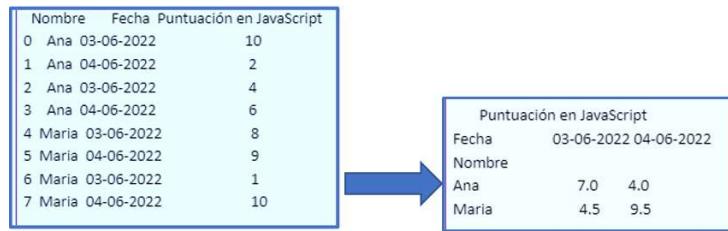
María 4.5 9.5

Vemos que el código «print(pivotTable)» muestra que, **como colocamos index="nombre", esa es la primera columna y, para cada nombre, aparece la media de la puntuación obtenida en cada una de las dos fechas.**

La función agregada por defecto «mean()» ha calculado la media de los valores.

Recordemos que el parámetro «aggfunc» de la función «pandas.pivot_table()», cuando no se le especifica algún elemento en particular, por defecto, muestra la media ('mean').

Figura 11: La base de la tabla *pivot* (tabla dinámica) y la tabla de la función «pandas.pivot_table()»



Fuente: elaboración propia.

Ahora, vamos a incluir valores en algunos parámetros de la función «`pandas.pivot_table()`», siguiendo este mismo ejercicio:

```
«index= "nombre"»
«columns= "fecha"»
«aggfunc= ["sum","count"]»
```

La función «`aggfunc`» al colocarle `aggfunc= ["sum","count"]` nos va a generar en el nuevo *DataFrame* de columnas para la suma (suma la puntuación del estudiante por fecha) y para el conteo (cuántos ejercicios de JavaScript hizo el estudiante en cada fecha).

Ingresamos el siguiente código:

```
>>> pivotTable = pd.pivot_table(dataframe,
>>>     index= "Nombre",
>>>     columns= "Fecha",
>>>     aggfunc= ["sum","count"])
>>> print(pivotTable)

      sum        count
      Puntuación en JavaScript    Puntuación en JavaScript
Fecha  03-06-2022 04-06-2022  03-06-2022 04-06-2022
Nombre
Ana      14     8         2     2
María     9    19         2     2
```

Vemos que el código `«print(pivotTable)»` muestra que, como colocamos `«índex="nombre"»`,

esa es la primera columna. Generó columnas para las fechas (así lo especificamos: «columns= "fecha"») y, para cada fecha, aparece la suma y el conteo separado por estudiante (así lo especificamos con el «aggfunc= ["sum", "count"]»).

Pueden colocarse en el «aggfunc» otros elementos estadísticos que nos interesen.

En un ejemplo como este con pocos datos, quizás no se ve el valor que esta función tiene, pero cuando se aplica la función «pandas.pivot_table()» adecuadamente a un gran volumen de datos (teniendo primero muy en claro los objetivos que se esperan lograr), nos muestra su gran potencial en el campo del procesamiento de datos, generando nuevos *DataFrames* que agrupan datos procedentes de otra tabla o de una base de datos de mayor tamaño, lo que constituye una herramienta muy útil para el procesamiento de grandes cantidades de datos (*Big Data*).

Pregunta 4

Seleccionar a qué opción corresponde la siguiente afirmación:

“La operación «pandas.pivot_table()» permite construir un nuevo *DataFrame* a partir de otro, de forma que”:

Los valores de una fila pasan a ser nombres de fila del nuevo *DataFrame*.

Los valores de una columna pasan a ser nombres de columna del nuevo *DataFrame*.

Justificación

Unidad 2. Metodología y características

Tema 1. Operaciones con *strings*

La clase *str* en Python se utiliza para representar texto, más conocido en el mundo de la programación como *string* o cadena de caracteres.

Una de las fortalezas de Python es la facilidad que brinda para trabajar datos del tipo *string* (cadena de texto). La librería Pandas proporciona un conjunto integral de operaciones de cadenas vectorizadas que se convierten en una pieza esencial del tipo de manipulación requerida cuando se trabaja con datos del mundo real.

La idea de las operaciones con *strings* o cadenas de texto es poder **ordenarlas o limpiarlas** para poder hacer mejores análisis.

Los cadenas (o *strings*) son un tipo de datos compuestos por secuencias de caracteres que representan texto. Estas cadenas de texto son de tipo *str* y se delimitan mediante el uso de comillas simples o dobles.

Recordemos:

- Los *strings* son cadenas de caracteres, es decir, un conjunto de caracteres que **pueden ser letras, números o símbolos**.
- Los *integers* son los números enteros.
- Los *float* son los números decimales.
- **Date** se refiere al dato asociado a la fecha u hora.

En Python, un carácter de guion bajo único (_) antes de un nombre se utiliza para especificar que el nombre **debe tratarse como una variable, función, método o clases "privado" o "interno"**.

En Python, cada letra de una palabra se puede identificar por su posición como en un vector, utilizando la función «len()».

Ingresamos el siguiente código al intérprete:

```
>>> palabra = 'manzana'  
>>> len(palabra)  
7  
>>> palabra[0]  
'm'  
>>> palabra[4]  
'a'
```

La función «len(palabra)» devolvió un valor entero que indica la cantidad de caracteres en la cadena 'manzana'.

La función «len()» devuelve la longitud de una cadena de caracteres o el número de elementos de una lista. El argumento de la función «len()» es la lista o cadena que queremos "medir".

>>> palabra[0] nos devolvió 'm' porque es el **primer carácter de la cadena, que comienza en la posición 0.**

>>> palabra[4] nos devolvió 'a' porque **es el carácter de la cadena que está en la posición 4.**

Recordemos: el primer carácter de la cadena comienza en la posición 0.

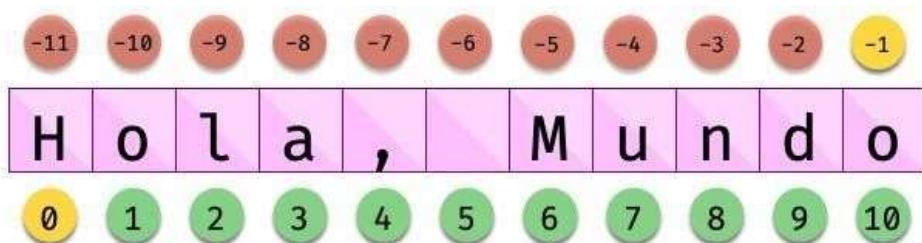
Las cadenas son iterables en Python y tienen cero-indexación. Los índices válidos para la longitud n de una cadena son 0,1,2,..,(n-1).

Podemos usar indexación negativa, donde el último elemento es un índice -1, el penúltimo elemento es un índice -2 y así sucesivamente.

“Los *strings* están indexados y cada carácter tiene su propia posición. Para obtener un único carácter dentro de una cadena de texto, es necesario especificar su índice dentro de corchetes [...]” (Delgado Quintero, 2020, <http://bit.ly/3YsIPbx>).

En la siguiente figura, vemos **arriba de la cadena la indexación negativa. Abajo de la cadena, vemos la indexación normal, iniciando en 0.**

Figura 12: Indexado de una cadena de texto



Fuente: Delgado Quintero, 2020, <http://bit.ly/3YsIPbx>

Las cadenas de texto en Python son inmutables. Por lo tanto, no podemos modificarlas.

Ingresamos el siguiente código al intérprete:

```
>>> mi_cadena = "escrito"
```

Intentemos cambiar esto a "**escritos**", agregando el carácter "s" al final.

Intentemos, entonces, asignar la letra "s". Mediante indexación negativa, sabemos que el índice del último carácter en «mi_cadena» es -1.

Ingresamos el siguiente código al intérprete:

```
>>> mi_cadena[-1]="s"
```

La salida será la siguiente:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

Como se muestra en el código de arriba, tenemos un *TypeError*. Esto es debido a que el objeto de cadena es inmutable y la tarea que intentamos ejecutar es inválida.

Sin embargo, puede que necesitemos modificar cadenas de texto. En este caso, los **métodos de cadena nos ayudarán**, ya que operan en cadenas existentes y retornan una nueva cadena modificada (hay que crear otro objeto cada vez que queramos modificar algo). **La cadena existente no es modificada.**

En el siguiente tema, aprenderemos a utilizar los métodos de cadena.

Pero, antes, **practiquemos algunas operaciones básicas con strings**. Saber cómo manipular cadenas de caracteres juega un papel fundamental en la mayoría de las tareas de procesamiento de texto.

En Python, para escribir cadenas de caracteres, delimitamos el texto con comillas simples o dobles.

Ingresamos el siguiente código al intérprete:

```
"esto es una cadena en Python"
```

Presionamos «Enter» y nos muestra:

```
'esto es una cadena en Python'
```

Observamos que no tuvimos que declarar previamente ninguna variable.

Ahora, vamos a asignar una cadena de caracteres a una variable, simplemente encerrando contenido entre comillas después de un signo de igual (=).

Ingresamos el siguiente código al intérprete:

```
>>> mensaje1 = 'Estoy aprendiendo operaciones con cadenas'
```

```
>>> print (mensaje1)
```

Estoy aprendiendo operaciones con cadenas

Operadores de cadenas de caracteres

Una forma de manipular cadenas de caracteres es utilizar operadores de cadenas de caracteres. Dichos operadores se representan con símbolos que asociamos a las matemáticas, como +, -, *, / y =. Estos signos realizan acciones similares a sus contrapartes matemáticas cuando se usan con las cadenas de caracteres, aunque no iguales.

Concatenar

Este término significa juntar cadenas de caracteres. **El proceso de concatenación se realiza mediante el operador de suma (+).** Debemos tener en cuenta que es necesario marcar explícitamente dónde queremos los espacios en blanco y colocarlos entre comillas.

Ingresamos el siguiente código al intérprete:

```
>>> mensaje1 = 'Estoy aprendiendo operaciones con cadenas'
```

```
>>> mensaje2 = 'en Python'
```

```
>>> frase = mensaje1 + mensaje2
```

```
>>> frase
```

'Estoy aprendiendo operaciones con cadenas en Python'

Observamos que muestra “cadenas en” sin separación. Tenemos que crear un espacio en blanco para que, al mostrar la frase, aparezcan separadas ambas cadenas.

Ingresamos el siguiente código al intérprete:

```
>>> mensaje1 = 'Estoy aprendiendo operaciones con cadenas'
```

```
>>> mensaje2 = 'en Python'
```

```
>>> # creamos un espacio entre ambas cadenas introduciendo ''
```

```
>>> frase = mensaje1 + '' + mensaje2
```

```
>>> frase
```

'Estoy aprendiendo operaciones con cadenas en Python'

Recordemos: concatenar dos o más *strings* con el **operador +** genera como resultado **un nuevo string.**

Multiplicar

Si necesitamos varias copias de una cadena de caracteres, utilizamos el **operador de multiplicación (*)**.

En este ejemplo, la cadena de caracteres mensaje2a lleva el contenido “Hola” tres veces, mientras que la cadena de caracteres mensaje2b tiene el contenido “Mundo”. Ordenemos imprimir las dos cadenas.

```
>>> mensaje2a = 'Hola ' * 3
```

```
>>> mensaje2b = 'Mundo'
```

```
>>> print(mensaje2a + mensaje2b)
```

Hola Hola Hola Mundo

Añadir

Si necesitamos añadir material de manera sucesiva al final de una cadena de caracteres, usamos el operador para esto que es compuesto (**+=**).

```
>>> mensaje3 = 'Hola'
```

```
>>> mensaje3 += ''
```

```
>>> mensaje3 += 'Mundo'
```

```
>>> print(mensaje3)
```

Hola Mundo

Comparar cadenas o *strings*

También es posible aplicar operaciones de comparación entre caracteres o cadenas.

```
>>> letra1 = 'd'  
>>> letra1 > 'a'  
True  
>>> letra1 > 'g'  
False  
>>> letra1 == 'D'  
False
```

El **símbolo** = es un operador de asignación.

El **símbolo** == en Python es un **operador de comparación**.

Un error habitual es utilizar un signo igual sencillo (=) en lugar del doble (==).

Al ingresar:

```
>>> letra1 == 'D'
```

Resultó:

False

Esto nos indica que los caracteres con mayúsculas son diferentes a los de minúsculas.

Mayúsculas y minúsculas de caracteres

Para convertir una frase o palabra a mayúsculas, se usa la instrucción: «palabra.upper()».

Para convertir una frase o palabra a minúsculas, se usa la instrucción: «palabra.lower()».

Mayúsculas: «upper()»

```
>>> mi_cadena = "escrito"  
>>> mi_cadena .upper()  
'ESCRITO'
```

Minúsculas: «lower()»

```
>>> mi_cadena = "escrito"  
>>> mi_cadena .lower()  
'escrito'
```

Pertenencia de un elemento

Si queremos comprobar que una determinada subcadena se encuentra en una cadena de texto utilizamos el operador **in**:

Se trata de una expresión que **tiene como resultado un valor «booleano» verdadero o falso**:

```
>>> proverb = 'Más vale malo conocido que bueno por conocer'
```

```
>>> 'malo' in proverb
```

```
True
```

```
>>> 'bueno' in proverb
```

```
True
```

```
>>> 'regular' in proverb
```

```
False
```

Nota: Prestemos atención al caso en el que buscamos descubrir si una subcadena **no está** en la cadena de texto:

```
>>> secuencia_ADN = 'ATGAAATTGAAATGGGA'
```

```
>>> 'C' not in secuencia_ADN
```

```
True. (Delgado Quintero, 2020, http://bit.ly/3YsIPbx)
```

Segmentar una cadena de texto

En algunos casos, puede ser útil dividir o separar un **string**. Para segmentar el texto por algún símbolo, se dispone de la instrucción **«split()»**.

Ingresamos el siguiente código al intérprete:

```
>>> texto = 'Estoy aprendiendo operaciones con cadenas'
```

```
>>> print(texto.split())
```

```
['Estoy', 'aprendiendo', 'operaciones', 'con', 'cadenas']
```

Separar los caracteres en una lista

Ingresamos el siguiente código al intérprete:

```
>>> list("Palabra")
```

Presionamos «Enter» y muestra la cadena separada en caracteres individuales, separados por comas:

```
[‘P’, ‘a’, ‘l’, ‘a’, ‘b’, ‘r’, ‘a’]
```

Cadena de caracteres de varias líneas

Para generar y definir un **string** largo, con varias líneas, se usan **tres comillas simples o dobles**.

Los saltos de línea también se tienen en cuenta a la hora de emitir la cadena.

Ingresamos el siguiente código al intérprete:

```
>>> x = """Estoy aprendiendo operaciones con cadenas en Python y aún hay mucha información.  
Estoy en el Módulo 3 y aun no lo hemos terminado,  
faltan otros temas por estudiar y esto aún no se termina  
ya que se amplía con temas que corresponden  
al Modulo 4 y finalmente allí termina con un punto y aparte."""
```

```
>>> print ( x )
```

```
Estoy aprendiendo operaciones con cadenas en Python y aún hay mucha información.  
Estoy en el Módulo 3 y aun no lo hemos terminado,  
faltan otros temas por estudiar y esto aún no se termina  
ya que se amplía con temas que corresponden  
al Modulo 4 y finalmente allí termina con un punto y aparte.
```

Vemos cómo al usar tres comillas dobles ("""") la cadena larga aparece de una forma adecuada.

Eliminar strings

Hemos visto anteriormente que un **string** en Python no puede ser modificado *a posteriori*. Tampoco se pueden borrar caracteres individuales de la cadena. El programa bloquea cualquier intento de modificación y muestra un mensaje de error.

Por lo tanto, puede ocurrir que debamos eliminar **strings** defectuosos del código que están asociados a una variable.

Ingresamos el siguiente código al intérprete:

```
>>> texto = "Este es un texto de ejemplo"
```

```
>>> print(texto)
```

Este es un texto de ejemplo

Ahora, ingresamos:

```
>>> del texto
```

```
>>> print (texto)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'texto' is not defined.

El sistema muestra un mensaje de error una vez que la variable que contenía ese texto fue eliminada.

Caracteres Unicode

Python trabaja por defecto con caracteres Unicode (<https://unicode-table.com/en/blocks/>).

Eso significa que tenemos acceso a la amplia carta de caracteres que nos ofrece este estándar de codificación.

Supongamos un ejemplo sobre el típico *emoji* de un cohete (*rocket*)

Representación Unicode del carácter *rocket*

```
>>> rocket_code = 0x1F680
```

```
>>> rocket = chr(rocket_code)
```

```
>>> rocket. (Delgado Quintero, 2020, http://bit.ly/3YsIPbx)
```

Pregunta 5

Si ingresamos en el intérprete de Python este código:

```
>>> palabra = 'procesamiento'
```

```
>>> len(palabra)
```

¿Cuál de las siguientes opciones será la salida correcta:

7

o

13

14

Justificación

Tema 2. Métodos de cadena de texto de Pandas

La mayor parte de la sintaxis de cadenas de texto de Pandas es fácil de comprender, ya que Python tiene muchos términos semejantes al inglés.

Python trae preinstalado (*built-in*) docenas de métodos que nos permiten hacer cosas con las cadenas de caracteres. Solos o en combinación, los métodos pueden hacer muchas operaciones con las cadenas de caracteres.

Podemos usar como referencia la lista de métodos de cadenas de caracteres (***String Methods***) en el sitio web de Python, que incluye información de cómo utilizar correctamente cada uno.

- <https://docs.python.org/2/library/stdtypes.html#string-methods>
- <https://docs.python.org/3/library/stdtypes.html>

A continuación, veremos algunos de estos métodos de cadena de texto de Pandas.

Cuando trabajamos con cadenas de texto en Python, podemos necesitar **buscar un patrón** en ellas o **reemplazar partes de una cadena** con otra subcadena.

Los métodos de cadenas operan en cadenas existentes y **retornan una nueva cadena modificada. La cadena existente no es modificada.**

Python posee dos métodos muy útiles que nos ayudan a ejecutar las tareas de procesar cadenas de texto:

- «**find()**» (find en inglés es encontrar)
- «**replace()**» (replace en inglés es reemplazar)

Método «**find()**»

Se usa el método «`find()`» para buscar patrones en cadenas de Python. Nos ayuda a buscar un patrón específico en una cadena.

La sintaxis general es la siguiente:

`<this_string>.find(<this_pattern>)`

La cadena **en donde vamos a buscar** está definida por **this_string**.

El patrón que vamos a **buscar** es **this_pattern**.

El **método `find()`** busca a través de **this_string** si existe la ocurrencia de **this_pattern**.

Si **this_pattern** existe, entonces retorna el índice inicial de la primera ocurrencia de **this_pattern**.

Si **this_pattern** no está en **this_string**, retorna -1.

Vamos a buscar una subcadena en una cadena de caracteres utilizando el método **find** y el intérprete nos indicará el **índice de inicio de la misma**.

Recordemos que los índices están numerados de izquierda a derecha y que el número en el que **se comienza a contar la posición es el 0**, no el 1. (Torres, 2021, <http://bit.ly/3F4B7O6>)

```
>>> mensaje5 = "Hola Mundo"  
>>> mensaje5a = mensaje5.find("Mundo")  
>>> print(mensaje5a)  
5
```

Nos muestra que **Mundo** comienza en la posición de índice 5.

Si la subcadena no está presente, el intérprete imprimirá el valor -1.

```
>>> mensaje6 = "Hola Mundo"  
>>> mensaje6a = mensaje6.find("perro")  
>>> print(mensaje6a)  
-1
```

Como la subcadena “**perro**” no está presente el intérprete, imprime el valor -1.

Método «`replace()`»

Si necesitamos reemplazar una **subcadena** de una cadena se puede utilizar el **método «`replace()`»**.

La sintaxis general es la siguiente:

<this_string>.replace(<this>, <with_this>)

El método replace() busca a través de this_string por el patrón this.

Si el patrón this está presente, retorna una nueva cadena de texto donde todas las ocurrencias del patrón this son reemplazadas con el patrón especificado en el artículo with_this.

Si el patrón this no es encontrado en this_string, la cadena de texto a retornar será la misma que this_string.

Si queremos reemplazar solo un cierto número de ocurrencias en vez de todas las ocurrencias en el patrón, podemos añadir un argumento opcional en la llamada al método que especifique cuántas ocurrencias del patrón queremos reemplazar.

<this_string>.replace(<this>, <with_this>, n_occurrences). (Torres, 2021,
<http://bit.ly/3F4B7O6>)

Veamos este ejemplo simple.

Ingresamos lo siguiente en el intérprete:

```
>>> mensaje = "HOLA MUNDO"  
# Vamos a reemplazar la L por pizza  
>>> mensaje1 = mensaje.replace("L", "pizza")  
>>> print(mensaje1)  
HOpizzaA MUNDO
```

Veamos otro ejemplo del método «replace()»

```
>>> mi_cadena= 'Yo disfruto codificando en C++.\\nC++ es fácil de aprender.\\nYo he estado  
codificando en C  
++ por 2 años.:)'  
# el carácter \\n introduce saltos de línea  
>>> print(mi_cadena)  
Yo disfruto codificando en C++.  
C++ es fácil de aprender.  
Yo he estado codificando en C++ por 2 años.:-)
```

Ahora, vamos usar el **método «replace()» para reemplazar la subcadena C++ por la subcadena Python.**

```
>>> print(mi_cadena.replace('C++','Python'))
```

Yo disfruto codificando en Python.

Python es fácil de aprender.

Yo he estado codificando en Python por 2 años.:)

Ahora, agregaremos el argumento adicional «n_occurrences» para que retorne una cadena donde solo la primera ocurrencia de ‘C++’ es reemplazada con ‘Python’:

```
>> print(mi_cadena.replace( 'C++','Python',1))
```

Yo disfruto codificando en Python.

C++ es fácil de aprender.

Yo he estado codificando en C++ por 2 años.:)

Método «index()»

Para buscar la ocurrencia de un patrón en una cadena, podemos usar también el **método «index()».**

Su sintaxis es la siguiente:

```
<this_string>.index(<this_pattern>)
```

El funcionamiento del método index() es muy similar al del método find().

Si this_pattern está presente en this_string, el método index() **retorna el índice inicial de la primera ocurrencia** de this_pattern.

Genera un ValueError si this_pattern no es encontrado en this_string.

Ingresamos el siguiente código en el intérprete:

```
>>> mi_cadena= 'Yo disfruto codificando en Python'
```

```
mi_cadena.index('Python')
```

27

Si contamos de izquierda a derecha en la cadena: Yo disfruto codificando en Python, **comenzando por la posición inicial de índice 0, la P de Python está en la posición 27.**

Método «capitalize()»

Pandas incluye funciones para abordar tanto la necesidad de operaciones de cadenas de texto vectorizadas como para **manejar correctamente los datos faltantes a través del atributo str** de los objetos Pandas *series*.

Por ejemplo, supongamos que creamos una serie Pandas con estos datos.

Figura 13: Pandas *series*

```
In [73]: import pandas as pd  
names = pd.Series(['peter', 'Paul', None, 'MARY', 'gUIDO'])  
names  
  
Out[73]: 0    peter  
1     Paul  
2     None  
3     MARY  
4    gUIDO  
dtype: object
```

Fuente: captura de pantalla de Python (Python 3.10.5).

Podemos usar el **método «capitalize()»** para trabajar todos los elementos de la serie de Pandas, incluyendo el valor faltante (*None*, en el índice 2).

Para convertir la primera letra de una cadena a mayúsculas, utilizamos el **método «capitalize()»**, que devuelve una copia de la cadena con la primera letra en mayúsculas.

Figura 14: Método «capitalize()»

```
In [74]: names.str.capitalize()  
  
Out[74]: 0    Peter  
1     Paul  
2     None  
3     Mary  
4    Guido  
dtype: object
```

Fuente: captura de pantalla de Python (Python 3.10.5).

Vemos que el método «**capitalize()**» devolvió una copia del *string*, donde el primer carácter de cada elemento de la cadena quedó en mayúsculas y el resto en minúsculas.

Son diversos los métodos de cadena de texto de Pandas.

A continuación, dejamos un listado de métodos para que podamos investigar y practicar:

«len()», «lower()», «translate()», «islower()», «ljust()», «upper()», «startswith()», «isupper()», «rjust()», «find()», «endswith()», «isnumeric()», «center()», «rfind()», «isalnum()», «isdecimal()», «zfill()», «index()», «isalpha()», «split()», «strip()», «rindex()», «isdigit()», «rsplit()», «rstrip()», «capitalize()», «isspace()», «partition()», «lstrip()», «swapcase()», «istitle()», «rpartition()».

Pregunta 6

Observando la sintaxis general del método «find()»: <this_string>.find(<this_pattern>

Seleccionar la opción correcta:

El patrón que vamos a buscar es this_string

La cadena en donde vamos a buscar está definida por this_pattern

El patrón que vamos a buscar es this_pattern

Justificación

Tema 3. Reestructurar un *DataFrame*

A menudo, la disposición de los datos en un *DataFrame* no es la adecuada para su tratamiento y es necesario reestructurar el *DataFrame*.

Los datos que contiene un *DataFrame* pueden organizarse en dos formatos: ancho y largo.

Figura 15: *DataFrame* de formatos ancho y largo

Formato ancho				Formato largo		
Nombre	Economía	Matemáticas	Programación	Nombre	Asignatura	Nota
Carmen	5.0	3.5	9.0	Carmen	Economía	5.0
Luis	6.5	7.0	4.0	Luis	Economía	6.5
Maria	8.0	8.5	6.5	Maria	Economía	8.0
				Carmen	Matemáticas	3.5
				Luis	Matemáticas	7.0
				Maria	Matemáticas	8.5
				Carmen	Programación	9.0
				Luis	Programación	4.0
				Maria	Programación	6.5

Fuente: Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>

Convertir un *DataFrame* a formato largo

Para convertir un *DataFrame* de formato ancho a formato largo (columnas a filas), se utiliza el siguiente método.

La función «`pandas.melt()`» remodela o transforma un *DataFrame* existente. Cambia la orientación del DataFrame de un formato ancho a uno largo.

Sintaxis de «`pandas.melt()`»

```
pandas.melt(dataframe,  
            id_vars,  
            value_vars,  
            var_name,  
            value_name,  
            col_level)
```

Parámetros

DataFrame: obligatorio. Es el *DataFrame* que queremos cambiar al formato largo.

Id_vars opcional: Puede ser un tuple, list, o una array N-dimensional. Es la columna utilizada para las variables de identificación. Puedes seleccionar más de una columna de identificadores.

Value_vars: opcional. Puede ser un tuple, list, o una array N-dimensional. Por defecto, las columnas no especificadas como variables identificadoras son variables de valor. También puedes seleccionarlas.

Var_name: opcional. Es una variable de tipo scalar. Es el nombre de la columna identificadora. Por defecto, es variable.

Value_name: opcional. Es una variable de tipo scalar. Es el nombre de la columna de no identificadores. Por defecto, es value.

Col_level: opcional. Es un entero o una cadena de caracteres. En el caso de las columnas de múltiples índices, podemos usar este parámetro para transformar nuestro *DataFrame*.

Devuelve un *DataFrame* transformado que contiene una o más columnas identificadoras y solo dos columnas no identificadoras llamadas variable y valor.

(Noor, 2023, <http://bit.ly/3L3yUX2>)

Veamos el siguiente ejemplo de uso de la función «`pandas.melt()`» para **cambiar la orientación del DataFrame de un formato ancho a uno largo**.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd  
>>> datos={'nombre':['María', 'Luis', 'Carmen'],  
>>>     'edad':[18, 22, 20],  
>>>     'Matemáticas':[8.5, 7, 3.5],  
>>>     'Economía':[8, 6.5, 5],  
>>>     'Programación':[6.5, 4, 9]}  
>>> df = pd.DataFrame(datos)  
>>> print(df)
```

Vemos que devuelve el DataFrame que hemos creado, de **formato ancho**:

	nombre	edad	Matemáticas	Economía	Programación
0	María	18	8.5	8.0	6.5
1	Luis	22	7.0	6.5	4.0
2	Carmen	20	3.5	5.0	9.0

	nombre	edad	Matemáticas	Economía	Programación
0	María	18	8.5	8.0	6.5
1	Luis	22	7.0	6.5	4.0
2	Carmen	20	3.5	5.0	9.0

```
>>> # Aplicamos la función pandas.melt()  
>>> df1 = df.melt(id_vars=['nombre', 'edad'], var_name='asignatura', value_name='nota')  
>>> print(df1)  
  
nombre edad asignatura nota  
0 María 18 Matemáticas 8.5  
1 Luis 22 Matemáticas 7.0  
2 Carmen 20 Matemáticas 3.5  
3 María 18 Economía 8.0  
4 Luis 22 Economía 6.5  
5 Carmen 20 Economía 5.0  
6 María 18 Programación 6.5  
7 Luis 22 Programación 4.0  
8 Carmen 20 Programación 9.0
```

Vemos que hemos convertido nuestro *DataFrame* de formato ancho a formato largo (**columnas a filas**), que las columnas de asignaturas quedaron ahora como filas.

Convertir un *DataFrame* a formato ancho

Para convertir un *DataFrame* de formato largo a formato ancho (**filas a columnas**) se utiliza el siguiente método:

Función `df.pivot`

```
df.pivot(index=filas, columns=columna, values=valores)
```

Devuelve el *DataFrame* que resulta de convertir el *DataFrame* df de formato largo a formato ancho. (Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>)

Se crean tantas columnas nuevas como valores distintos haya en la columna **columna**.

Los nombres de estas nuevas columnas son los valores de la columna **columna**, mientras que sus valores se toman de la columna **valores**.

Los nombres del índice del nuevo *DataFrame* se toman de los valores de la columna **filas**.

Veamos el siguiente ejemplo de uso de la **función «df.pivot()»**.

Continuando con el mismo código anterior en el que habíamos generado un *DataFrame* de formato largo (df1), ingresamos en el intérprete lo siguiente:

```
>>> print(df1.pivot(index='nombre', columns='asignatura', values='nota'))  
asignatura Economía Matemáticas Programación  
nombre  
Carmen    5.0    3.5    9.0  
Luis      6.5    7.0    4.0  
María     8.0    8.5    6.5. (Aprende con Alf, 2022, http://bit.ly/3SQ0HMi)
```

Devuelve el *DataFrame* que resulta de convertir el *DataFrame* (df1) de formato largo a formato ancho.

Pregunta 7

Seleccionar a qué opción corresponde la siguiente afirmación:

“La función «pandas.melt()» cambia la orientación del *DataFrame*”.

Seleccionar la opción correcta:

De un formato largo a uno ancho.

De un formato ancho a uno largo.

Justificación

Tema 4. Combinación y concatenación de *DataFrames*

Dos o más *DataFrames* pueden combinarse en otro *DataFrame*. La combinación puede ser de varias formas:

Concatenación: combinación de varios *DataFrames* concatenando sus filas o columnas.

Mezcla: combinación de varios *DataFrames* usando columnas o índices comunes.

Concatenación de *DataFrames*

Para concatenar dos o más *DataFrames*, se utiliza el siguiente método:

`df.concat(dataframes, axis = eje)`

Devuelve el *DataFrame* que resulta de concatenar los *DataFrames* de la lista *DataFrames*.

- Si el eje es 0 (valor por defecto), la concatenación se realiza por filas.
- Si el eje es 1, se realiza por columnas.

Si los *DataFrames* que se concatenan por filas no tienen el mismo índice de columnas, el *DataFrame* resultante incluirá todas las columnas existentes en los *DataFrames* y rellenará con valores NaN los datos no disponibles.

Si los *DataFrames* que se concatenan por columnas no tienen el mismo índice de filas, el *DataFrame* resultante incluirá todas las filas existentes en los *DataFrames* y rellenará con valores NaN los datos no disponibles.

Concatenación de filas. Las filas de los *DataFrames* se concatenan unas a continuación de las otras para formar el nuevo *DataFrame*. Para ello es necesario que los *DataFrames* que se combinen tengan el mismo índice de columnas. (Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>)

Figura 16: Concatenación de dos *DataFrame* por filas

Concatenación por filas		
Nombre	Sexo	Edad
Carmen	Mujer	22
Luis	Hombre	18
Nombre	Sexo	Edad
María	Mujer	25
Pedro	Hombre	30



Nombre	Sexo	Edad
Carmen	Mujer	22
Luis	Hombre	18
María	Mujer	25
Pedro	Hombre	30

Fuente: Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>

Veamos este ejemplo de concatenación de dos *DataFrame* por filas.

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd
>>> # creamos el primer DataFrame df1
>>> df1 = pd.DataFrame({"Nombre":["Carmen", "Luis"],
   "Sexo":["Mujer", "Hombre"], "Edad": [22, 18]}).set_index("Nombre")
>>> df1
   Sexo Edad
Nombre
Carmen Mujer 22
Luis Hombre 18

>>> # creamos el segundo df2
>>> df2 = pd.DataFrame({"Nombre":["María", "Pedro"],
   "Sexo":["Mujer", "Hombre"], "Edad": [25, 30]}).set_index("Nombre")
>>> df2
   Sexo Edad
Nombre
María Mujer 25
Pedro Hombre 30. ((Aprende con Alf, 2022, http://bit.ly/3SQ0HMi)).
```

Observamos que ambos *DataFrame* df1 y df2 tienen el mismo índice de columnas: nombre, sexo y edad.

Con el siguiente código, vamos a hacer la concatenación de los dos *DataFrame* por filas, con el método «pd.concat()»:

```
>>> df = pd.concat([df1, df2])
>>> df
   Sexo Edad
Nombre
Carmen Mujer 22
Luis Hombre 18
María Mujer 25
Pedro Hombre 30
```

Tenemos ahora el *DataFrame* resultante df, que incluye todas las columnas existentes en ambos *DataFrame* df1 y df2 y contiene los datos organizados de ambos *DataFrame* originales.

En este ejemplo, tenemos pocos datos, pocas filas y columnas, pero cuando se manejan

tablas con centenares o miles de datos y muchas filas y columnas, comprendemos el potencial y la utilidad de este método de Pandas.

"Concatenación de columnas. Las columnas de los *DataFrames* se concatenan unas a continuación de las otras para formar el nuevo *DataFrame*. Para ello, es necesario que los *DataFrames* que se combinen **tengan el mismo índice de filas**" (Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>).

Figura 17: Concatenación de dos *DataFrame* por columnas

Concatenación por columnas

Nombre	Sexo
Carmen	Mujer
Luis	Hombre
María	Mujer

Nombre	Edad
Carmen	22
Luis	18
María	25

→

Nombre	Sexo	Edad
Carmen	Mujer	22
Luis	Hombre	18
María	Mujer	25

Fuente: Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>

Veamos este ejemplo de **concatenación de dos *DataFrame* por columnas**

Ingresamos el siguiente código en el intérprete:

```
>>> import pandas as pd  
>>> # creamos el primer DataFrame df1  
>>> df1 = pd.DataFrame({"Nombre":["Carmen", "Luis", "María"],  
"Sexo":["Mujer", "Hombre", "Mujer"]}).set_index("Nombre")  
>>> df1  
Sexo  
Nombre  
Carmen Mujer  
Luis Hombre  
María Mujer  
  
>>> # creamos el segundo DataFrame df2  
>>> df2 = pd.DataFrame({"Nombre":["Carmen", "Luis", "María"],
```

```
"Edad":[22, 18, 25]).set_index("Nombre")
```

```
>>> df2
```

Edad

Nombre

Carmen 22

Luis 18

María 25

Observamos que ambos *DataFrame* df1 y df2 tienen **el mismo índice de filas**.

Con el siguiente código, vamos a hacer la concatenación de los dos *DataFrame* por columnas, con el método «pd.concat()», colocando axis = 1), ya que, si el eje es 1, se realiza por columnas:

```
>>> df = pd.concat([df1, df2], axis = 1)
```

```
>>> df
```

Sexo Edad

Nombre

Carmen Mujer 22

Luis Hombre 18

María Mujer 25

Tenemos ahora el *DataFrame* resultante df, que incluye todas las filas existentes en ambos *DataFrame* df1 y df2 y contiene los datos organizados de ambos *DataFrame* originales.

Mezcla de *DataFrames*

La mezcla de *DataFrames* permite integrar filas de dos *DataFrames* que contienen información en común en una o varias columnas o índices que se conocen como clave.

Para mezclar dos *DataFrames* se utiliza el siguiente método:

`df.merge(df1, df2, on = clave, how = tipo)`

Devuelve el *DataFrame* que resulta de mezclar el *DataFrame* df2 con el *DataFrame* df1, usando como claves las columnas de la lista clave y siguiendo el método de mezcla indicado por tipo.

El tipo de mezcla puede ser:

"inner"(por defecto): El DataFrame resultante sólo contiene las filas cuyos valores en la clave están en los dos DataFrames. Es equivalente a la intersección de conjuntos.

"outer": El DataFrame resultante contiene todas las filas de los dos DataFrames. Si una fila de un DataFrame no puede emparejarse con otra los mismos valores en la clave en el otro DataFrame, la fila se añade igualmente al DataFrame resultante rellenando las columnas del otro DataFrame con el valor NaN. Es equivalente a la unión de conjuntos.

"left": El DataFrame resultante contiene todas las filas del primer DataFrame y descarta las filas del segundo DataFrame que no pueden emparejarse con alguna fila del primer DataFrame a través de la clave.

"right": El DataFrame resultante contiene todas las filas del segundo DataFrame y descarta las filas del primer DataFrame que no pueden emparejarse con alguna fila del segundo DataFrame a través de la clave.

```
>>> import pandas as pd  
>>> df1 = pd.DataFrame({"Nombre":["Carmen", "Luis", "María"], "Sexo":["Mujer", "Hombre",  
"Mujer"]})  
>>> df2 = pd.DataFrame({"Nombre":["María", "Pedro", "Luis"], "Edad":[25, 30, 18]})  
>>> df = pd.merge(df1, df2, on="Nombre", how="outer")  
>>> print(df)  
   Nombre Sexo Edad  
0 Carmen Mujer NaN  
1 Luis Hombre 18.0  
2 María Mujer 25.0  
3 Pedro NaN 30.0
```

Los valores **NaN** aparecen cuando no pueden emparejarse en ambos *DataFrames* con los mismos valores, como se explicó anteriormente.

En la librería Pandas de Python, los **valores perdidos** se representan con **None** y **NaN** (**acrónimo de Not a Number**). (Aprende con Alf, 2022, <http://bit.ly/3SQ0HMi>)

Pregunta 8

Seleccionar a qué opción corresponde la siguiente afirmación:

'Las filas de los *DataFrames* se concatenan unas a continuación de las otras para formar el nuevo *DataFrame*. Para ello, es necesario que los *DataFrames* que se combinén tengan el mismo índice de columnas'.

Seleccionar la opción correcta:

Concatenación de filas.

Concatenación de columnas.

Justificación

Video de habilidades

Preguntas de habilidades

- Los datasets pueden combinarse con la función “merge()”, con dos alternativas posibles: “inner” y “outer”.

¿Qué afirmación responde mejor al siguiente caso?

```
In [46]: df1 = pd.DataFrame({  
    'Nombre': ['Juan', 'Francisca', 'Barbara', 'Tomas', 'Antonio'],  
    'Comuna': ["Providencia", "Macul", "Independencia", "La Reina", "La Florida"],  
    'Edad': [35, 21, 45, 38, 25],  
})  
  
df2 = pd.DataFrame({  
    'Empleado': ['Francisca', 'Juan', 'Tomas', 'Barbara'],  
    'Empresa': ['Indura', 'Latam', 'Ripley', 'Toyota'],  
    'Area': ['Contabilidad', 'Administracion', 'Ventas', 'RRHH'],  
})  
  
print(pd.merge(df1, df2, left_on="Nombre", right_on="Empleado", how="inner"))  
print()  
print(pd.merge(df1, df2, left_on="Nombre", right_on="Empleado", how="outer"))
```

	Nombre	Comuna	Edad	Empleado	Empresa	Area
0	Juan	Providencia	35	Juan	Latam	Administracion
1	Francisca	Macul	21	Francisca	Indura	Contabilidad
2	Barbara	Independencia	45	Barbara	Toyota	RRHH
3	Tomas	La Reina	38	Tomas	Ripley	Ventas

	Nombre	Comuna	Edad	Empleado	Empresa	Area
0	Juan	Providencia	35	Juan	Latam	Administracion
1	Francisca	Macul	21	Francisca	Indura	Contabilidad
2	Barbara	Independencia	45	Barbara	Toyota	RRHH
3	Tomas	La Reina	38	Tomas	Ripley	Ventas
4	Antonio	La Florida	25		NaN	NaN

Inner retorna solamente aquellos registros combinados que coinciden entre “Nombre” y “Empleado”. No se considera “Antonio”.

Outer retorna todos los registros de ambos datasets, considerando a “Antonio”.

Outer genera valores NaN para complementar valores faltantes.

Inner trae como resultado un dataframe con todas las filas de ambos datasets

Justificación

- En función de la siguiente imagen, ¿qué modificación identificás?

```
In [49]: df1 = pd.DataFrame({
    'Nombre': ['Juan', 'Francisca', 'Barbara', 'Tomas', 'Antonio'],
    'Comuna': ['Providencia', 'Macul', 'Independencia', 'La Reina', "La Florida"],
    'Edad': [35, 21, "45", 38, "25"],
})

df2 = pd.DataFrame({
    'Empleado': ['Francisca', 'Juan', 'Tomas', 'Barbara'],
    'Empresa': ['Indura', 'Latam', 'Ripley', 'Toyota'],
    'Area': ['Contabilidad', 'Administracion', 'Ventas', 'RRHH'],
})

print(pd.merge(df1, df2, left_on="Nombre", right_on="Empleado", how="inner").drop('Empleado', axis=1))
print()
print(pd.merge(df1, df2, left_on="Nombre", right_on="Empleado", how="outer").drop('Empleado', axis=1))

      Nombre      Comuna Edad Empresa   Area
0     Juan    Providencia   35   Latam Administracion
1 Francisca        Macul   21 Indura Contabilidad
2 Barbara  Independencia   45 Toyota       RRHH
3     Tomas      La Reina   38 Ripley      Ventas
4     Antonio     La Florida  25      NaN      NaN
```

Se eliminó una columna llamada "Empleado".

Se eliminó una columna llamada "Nombre".

Se agregó una nueva columna llamada "Empleado".

Justificación

3. ¿Cuál es la función de “how”?

Es una función que permite calcular la media de un DataFrame en pandas.

Es una función que permite especificar cómo se deben combinar los datos al unir dos DataFrames en pandas.

Es una función que permite contar el número de elementos no nulos en un DataFrame en pandas.

Justificación

4. Supongamos que tenemos dos DataFrames, uno con información sobre clientes y otro con información de sus compras. Queremos juntar ambos conjuntos de datos para tener un único DataFrame con toda la información. Creamos lo siguiente:

```

import pandas as pd
clientes = pd.DataFrame({
    'id_cliente': [1, 2, 3, 4],
    'nombre': ['Juan', 'Ana', 'Pedro', 'Luis'],
    'edad': [32, 28, 45, 19]
})

compras = pd.DataFrame({
    'id_cliente': [1, 2, 2, 3, 4, 4, 4],
    'producto': ['libro', 'pelota', 'cuaderno', 'guitarra', 'libreta', 'bolígrafo', 'mochila'],
    'precio': [20, 10, 5, 200, 3, 1, 50]
})

datos_cliente = pd.concat([clientes, compras], axis=1)

print(datos_cliente)

```

Utilizamos la función concat de Pandas para unir ambos DataFrames. ¿Cuál es el argumento que debemos utilizar para concatenar a lo largo de las filas?

axis=0

axis=1

No es posible concatenar a lo largo de las filas.

Justificación

Cierre

La ordenación de datos es un proceso fundamental dentro del procesamiento de datos, en donde hay que establecer criterios de ordenamiento de los datos para darle un sentido y claridad a un gran volumen de datos iniciales.

En este módulo 3, estudiamos una amplia variedad de herramientas de la librería Pandas, muy poderosas en el procesamiento de datos. Practicamos cómo ordenar datos y organizar los datos en orden alfabético o en un orden numérico ascendente o descendente y a arreglar los registros de una tabla en algún orden secuencial de acuerdo con un criterio de ordenamiento, etc.

Resumir la información es esencial cuando nos encontramos con una enorme cantidad de datos. Aprendimos sobre las agregaciones: tomando una serie de datos, subdividir los datos en grupos y agregar columnas o filas de un *DataFrame* y, sobre las agrupaciones, agrupar los datos según las categorías y aplicar una función a las categorías para generar grupos resumiendo la

información.

Estudiamos el potencial de las tablas dinámicas o tablas pivote como una herramienta muy útil en tareas de análisis de datos, en el procesamiento de grandes cantidades de datos.

Aprendimos la fortalezas de Python para trabajar datos del tipo *string* (cadena de texto), proporcionando un conjunto integral de operaciones de cadenas vectorizadas que se convierten en una pieza esencial del tipo de manipulación requerida cuando se trabaja con datos del mundo real.

Estudiamos los métodos de cadena de texto de Pandas, que Python trae preinstalados (*built-in*) con docenas de métodos que nos permiten hacer diversas operaciones con las cadenas de caracteres.

Finalmente, estudiamos cómo reestructurar un *DataFrame* y cómo realizar combinaciones, concatenación de *DataFrames* y mezcla de *DataFrames*.

A modo de cierre, invitamos a ver el siguiente video.

Video 2. Pandas

Fuente: Cctmexico. (2017, 23 de junio). *Pandas desde cero (Python): ¿Cómo hacer un Data frame? (Básico)*. [Archivo de video]. YouTube. <https://www.youtube.com/watch?v=JJ7BMoQotEY>

También, recomendamos leer el siguiente artículo en donde se ejemplifican distintas formas de ordenar datos.

Diferentes formas de ordenar dataframes en pandas

Figura 18: Pandas



Fuente: [Imagen sin título sobre Pandas], (s.f.), <https://images.app.goo.gl/nYJGaqpXUHm8EMdJ6>

Glosario

Referencias

[Imagen sin título sobre Pandas]. (s. f.).

<https://images.app.goo.gl/nYJGaqpXUHm8EMdJ6>

Aprende con Alf. (2022). La Librería Pandas.

<https://aprendeconalf.es/docencia/python/manual/pandas/>

Cctmexico. (2017, 23 de junio). Pandas desde cero (Python): ¿Cómo hacer un Data frame? (Básico). [Archivo de video]. YouTube. <https://www.youtube.com/watch?v=JJ7BMoQotEY>

Cursos de informática. (s. f.). Preparación y limpieza de los datos.

<https://cursosinformatica.ucm.es/trial/analisis/>

Delgado Quintero, S. (2020). Cadenas de texto. Aprende Python.

<https://aprendepython.es/core/datatypes/strings/>

Joshi, S. (2023, 30 de enero). Pandas DataFrame DataFrame.groupby() Función.

DelfStack. <https://www.delftstack.com/es/api/python-pandas/pandas-dataframe-dataframe.groupby-function/>

Noor, M. (2023, 30 de enero). Función Pandas DataFrame.describe(). DelfStack.

<https://www.delftstack.com/es/api/python-pandas/pandas-dataframe-dataframe.describe-function/#sintaxis-de-pandasdataframedescribe>

Noor, M. (2023, 30 de enero). Función Python Pandas pandas.pivot_table(). DelfStack.

https://www.delftstack.com/es/api/python-pandas/python-pandas-pandas.pivot_table-function/

Noor, M. (2023, 30 de enero). Función pandas pandas.melt(). *DelftStack*.
<https://www.delftstack.com/es/api/python-pandas/pandas-dataframe-dataframe.melt-function/>

OpenWebinars. (2020, 4 de noviembre). *Agrupaciones de datos con Pandas* [Archivo de video]. YouTube. <https://www.youtube.com/watch?v=hGQB9nAbjhI>

Rodríguez, D. (2018, 23 de noviembre). Tablas dinámicas en Python con pandas. *Analytics Lane*. <https://www.analyticslane.com/2018/11/23/tablas-dinamicas-en-python-con-pandas/>

Rodríguez, D. (2019, 29 de abril). Diferentes formas de ordenar *dataframes* en pandas. *Analytics Lane*. <https://www.analyticslane.com/2019/04/29/diferentes-formas-de-ordenar-dataframes-en-pandas/>

Torres, A. (2021). Ordenar listas en Python: Como ordenar por descendente o ascendente. *Free Code Camp*. <https://www.freecodecamp.org/espanol/news/ordenar-listas-en-python-como-ordenar-por-descendente-o-ascendente/>