

# SESSION 7-SUMMARY

Victor Miguel Terron Macias

25/2/2021

## SESION 7. FUNCIONES VECTORIZADAS Y LIMPIEZA DE DATOS

### OBJETIVOS

- Identificar las fuentes a las que se tiene que se tiene acceso para realizar proyectos.
- Practicar el proceso declarativo de transformación de datos y entender como funciona la aplicación de funciones a estructuras de datos de pandas.
- Realizar la limpieza de datos e identificar a este proceso como el primer paso a realizar en el procesamiento.

### PREWORK SESION 7

- Leer archivos en formato CSV
- Utilizar técnicas más avanzadas de exploración de datos
- Utilizar la exploración para encontrar inconsistencias, errores y redundancias
- Usar algunas técnicas básicas de limpieza de datos

### INTRODUCCIÓN

Ya sabemos cómo leer archivos con pandas y cómo realizar una exploración básica del contenido. En esta sesión vamos a aprender algunas técnicas más avanzadas de Exploración de Datos. También vamos a ver los principios de la Limpieza de Datos, que usamos para dejar nuestros conjuntos de datos listos para ser reestructurados, analizados y visualizados.

La Exploración y la Limpieza van totalmente de la mano. No puedes limpiar sin explorar primero, y gran parte de las técnicas de exploración que usamos tienen como objetivo justamente encontrar inconsistencias, errores, redundancias, etc, en nuestro conjunto de datos para poder deshacernos de ellas.

### LECTURA DE CSV's

La sesión pasada aprendimos a leer archivos en formato JSON. El formato JSON, que es muy parecido a los diccionarios de Python, es sólo uno de los tantos formatos con los que nos vamos a topar.

Los CSVs pertenecen a una clase de formatos donde las columnas de nuestra tabla se delimitan usando lo que se llama un separador. CSV significa Comma-Separated Values y como bien imaginarás significa que se usa una coma (,) para separar las columnas. Un CSV se ve así:

```
Suburb,Address,Rooms,Type,Price,Method,SellerG,Date,Distance,Postcode,Bedroom2,Bathroom,Car,Landsize,BuildingArea,YearBuilt,CouncilArea,Latitude,Longitude,Regionname,Propertycount
Abbotsford,68 Studley St,2,h,,SS,Jellis,3/09/2016,2.5,3067,2,1,1,126,,,Yarra,-37.8814,144.9958,Northern Metropolitan,4019
Abbotsford,85 Turner St,2,h,1480000,S,Biggin,3/12/2016,2.5,3067,2,1,1,282,,,Yarra,-37.7996,144.9984,Northern Metropolitan,4019
Abbotsford,25 Bloomburg St,2,h,1035000,S,Biggin,4/02/2016,2.5,3067,2,1,0,156,79,1900,Yarra,-37.8879,144.9936,Northern Metropolitan,4019
Abbotsford,18/659 Victoria St,3,u,,VB,Rounds,4/02/2016,2.5,3067,3,2,1,0,,,Yarra,-37.8114,145.0116,Northern Metropolitan,4019
Abbotsford,5 Charles St,3,h,1465000,SP,Biggin,4/03/2017,2.5,3067,3,2,0,134,150,1900,Yarra,-37.8893,144.9944,Northern Metropolitan,4019
Abbotsford,40 Federation La,3,h,850000,PI,Biggin,4/03/2017,2.5,3067,3,2,1,94,,,Yarra,-37.7969,144.9969,Northern Metropolitan,4019
Abbotsford,55a Park St,4,h,1600000,VB,Nelson,4/06/2016,2.5,3067,3,1,2,120,142,2014,Yarra,-37.8072,144.9941,Northern Metropolitan,4019
Abbotsford,16 Maugie St,4,h,,SN,Nelson,6/08/2016,2.5,3067,3,2,2,400,220,2006,Yarra,-37.7965,144.9965,Northern Metropolitan,4019
Abbotsford,53 Turner St,2,h,,S,Biggin,6/08/2016,2.5,3067,4,1,2,201,,1900,Yarra,-37.7995,144.9974,Northern Metropolitan,4019
```

Figure 1: IMG

En un archivo de texto donde cada fila de nuestra tabla tiene su propia línea y donde los valores de cada columna se delimitan usando una coma (,). Leer archivos .csv usando pandas es muy fácil. Lo único que tienes que hacer es lo siguiente:

```
In [2]: df = pd.read_csv('../Datasets/melbourne_housing-raw.csv', sep=',')
```

Figure 2: IMG

pandas tiene un muy conveniente método llamado `read_csv` que nos permite leer archivos .csv directamente. Ese mismo método también puede ayudarnos a leer otros formatos con columnas delimitadas por otros separadores. Por ejemplo, podemos leer .tsv ('tab-separated values'), que son archivos donde cada columna está delimitada por un tab (indentación). Sólo basta con llamar el método con el argumento `sep=`

¡Como ves, leer archivos tipo .csv es muy fácil!

## ANALISIS EXPLORATORIO DE DATOS

Muy bien, empecemos nuestra exploración con las técnicas que ya conocemos primero:

```
In [4]: df.head(5)
Out[4]:
```

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	..	Bathroom	Car	Landsize	BuildingArea	YearBuilt	Co
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	..	1.0	1.0	126.0	NaN	NaN	
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	..	1.0	1.0	202.0	NaN	NaN	
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	..	1.0	0.0	156.0	79.0	1900.0	
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	..	2.0	1.0	0.0	NaN	NaN	
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	..	2.0	0.0	134.0	150.0	1900.0	

5 rows x 21 columns

```
In [3]: df.shape
Out[3]: (19740, 21)

In [6]: df.columns
Out[6]: Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG', 'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Latitude', 'Longitude', 'Regionname', 'Propertycount'], dtype='object')
```

Figure 3: IMG

Ok, algunas cosas importantes:

- El dataset tiene 19740 filas y 21 columnas
- Podemos observar que la información que tiene el dataset es básicamente cierta descripción de las propiedades, datos de su ubicación y tipo y fecha de venta.

```

In [7]: df.dtypes
Out[7]: Suburb          object
        Address         object
        Rooms          int64
        Type           object
        Price          float64
        Method         object
        SellerG        object
        Date           object
        Distance       float64
        Postcode       float64
        Bedroom2       float64
        Bathroom       float64
        Car            float64
        Landsize       float64
        BuildingArea   float64
        YearBuilt      float64
        CouncilArea    object
        Lattitude      float64
        Longitude      float64
        Regionname     object
        Propertycount  float64
        dtype: object

```

Figure 4: IMG

- Hay varias columnas float64, rooms es int64, y el resto son object.
- Tenemos algunos Nans en el dataset.

¿NaNs? ¿Qué quiero decir con eso? Bueno, los NaNs son valores Not a Number que básicamente son valores nulos en nuestro dataset. Estos valores pueden causar muchos problemas, ya que son valores nulos en columnas numéricas pero no podemos realizar operaciones matemáticas con ellos (son valores Not a Number, así que las matemáticas están fuera de las posibilidades). En esta sesión vamos a aprender a lidiar con estos valores, pero antes tenemos que aprender algo llamado funciones vectorizadas, que nos ayudará mucho durante esta exploración.

## ARITMÉTICA CON SERIES Y FUNCIONES VECTORIZADAS

¿Recuerdas nuestras funciones map y filter? A esas funciones les pasábamos nosotros una función y una lista y nos regresaban una lista con los resultados de aplicarle la función a cada uno de los elementos en orden.

Las funciones vectorizadas funcionan muy parecido, pero están optimizadas para funcionar con arreglos de pandas y de numpy (si quieres saber más sobre numpy lee esto). Si tomas un arreglo de pandas (es decir, una Serie) y le aplicas una función vectorizada la función se aplica a todo el arreglo elemento por elemento y te regresa un arreglo del mismo tamaño con el resultado de la aplicación.

Aplicar funciones de manera vectorizada a una Serie de pandas es facilísimo, incluso más fácil que usar la función map. Primero veamos que podemos realizar una operación aritmética con una Serie y la operación se aplicará automáticamente a todo el arreglo “elemento por elemento”:

```

In [8]: serie = pd.Series([1, 2, 3, 4, 5])
        serie * 5
Out[8]: 0    5
        1   10
        2   15
        3   20
        4   25
        dtype: int64

```

Figure 5: IMG

Podemos realizar cualquier operación matemática y la aplicación se hará de la misma manera:

```

In [9]: serie * 10
Out[9]: 0    10
        1    20
        2    30
        3    40
        4    50
        dtype: int64

In [10]: (serie + 100) / 2
Out[10]: 0    50.5
         1    51.0
         2    51.5
         3    52.0
         4    52.5
         dtype: float64

```

Figure 6: IMG

¡Qué genial! No tenemos que usar map, ni que declarar una función, ni que usar lambda. Basta con una simple operación matemática. Pandas además está optimizado para funcionar de esta manera, así que la velocidad de aplicación es mucho mayor que la combinación de map con listas.

Otra manera de aplicar estas transformaciones es usando funciones vectorizadas. Por ejemplo, esto lo podemos hacer usando algunas funciones de numpy. Numpy es otra librería que es muy común entre los científicos de datos. Ofrece muchas herramientas para realizar cálculos numéricos a altas velocidades. No usaremos mucho esta librería en este módulo, pero es importante entender que se pueden utilizar funciones de numpy para aplicar funciones de manera vectorizada a Series de pandas. Para importar numpy hacemos lo siguiente:

```

In [11]: import numpy as np

```

Figure 7: IMG

Y por ejemplo, si quisiéramos elevar al cuadrado nuestra Serie, podríamos hacer algo como esto:

```

In [12]: np.power(serie, 2)
Out[12]: 0     1
         1     4
         2     9
         3    16
         4    25
         dtype: int64

```

Figure 8: IMG

También podemos sacar la raíz cuadrada de nuestra Serie:

## AGREGACIONES

Hay una variación de estas funciones vectorizadas llamadas agregaciones (o reducciones) que lo que hacen es tomar un arreglo, atravesarlo “elemento por elemento” y regresar un solo número que es un “resumen” del arreglo. Este “resumen” es justamente la agregación o reducción. Podemos aplicar estas funciones usando numpy o directamente desde una Serie o DataFrame de pandas. Para efectos prácticos, vamos a utilizar los métodos que vienen integrados directamente en pandas. Por ejemplo, podemos sumar todos los valores de una Serie de esta manera:

O podemos contar el número de elementos en una Serie así:

Podemos obtener el valor más pequeño de la Serie:

O el valor más grande:

¡Y eso no es todo!

```
In [13]: np.sqrt(serie)
Out[13]: 0    1.000000
         1    1.414214
         2    1.732051
         3    2.000000
         4    2.236068
         dtype: float64
```

Figure 9: IMG

```
In [15]: serie = pd.Series([1, 2, 3, 4, 5])
         serie.sum()
Out[15]: 15
```

Figure 10: IMG

```
In [17]: serie.count()
Out[17]: 5
```

Figure 11: IMG

```
In [18]: serie.min()
Out[18]: 1
```

Figure 12: IMG

```
In [19]: serie.max()
Out[19]: 5
```

Figure 13: IMG

## FUNCIONES VECTORIZADAS Y AGREGACIONES CON DATAFRAMES

Tanto las funciones vectorizadas como las agregaciones pueden ser aplicadas a DataFrames completos.

Veamos primero las agregaciones. Al aplicar una agregación a un DataFrame, lo que obtenemos de regreso es el resultado de aplicar la función a cada una de las columnas (que al final de cuentas son Series, ¿lo recuerdas?). Por ejemplo, tenemos este DataFrame:

```
In [27]: df
Out[27]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemonstar	24	3	3
Cegatron	54	6	45
Pikame Mucho	223	10	23
Lazarillo de Tormes	76	2	76
Stevia Wonder	56	5	24
Needle	12	45	6
El AgMcDuelo	34	2	2

Figure 14: IMG

Mira qué pasa cuando le aplicamos la agregación sum:

```
In [28]: df.sum()
Out[28]: precio      491
cantidad_en_stock    73
productos_vendidos   179
dtype: int64
```

Figure 15: IMG

pandas toma cada una de las columnas, suma todos los valores dentro de cada columna y nos regresa el resultado de las sumas en una nueva Serie, donde el índice son los nombres de las columnas en el DataFrame y los valores son las sumas.

También funciona con las demás agregaciones:

```
In [29]: df.min()
Out[29]: precio      12
cantidad_en_stock    2
productos_vendidos    2
dtype: int64

In [30]: df.max()
Out[30]: precio      223
cantidad_en_stock    45
productos_vendidos    76
dtype: int64
```

Figure 16: IMG

Interesante, ¿no lo crees?

Ahora veamos qué pasa cuando aplicamos funciones vectorizadas a nuestro DataFrame:

La función  $(x * 100)$  fue aplicada a cada uno de los elementos del DataFrame y obtuvimos un nuevo DataFrame con los resultados, ¿ves?

Cualquier función vectorizada que le apliquemos al DataFrame va a tener el mismo efecto:

Ahora sí, estamos listos para aprender a lidiar con valores nulos (NaNs).

```
In [33]: df * 100
```

```
Out[33]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	3400	300	300
Cegatron	5400	600	4500
Pikame Mucho	22300	1000	2300
Lazarillo de Tormes	7800	200	7600
Stevie Wonder	5600	500	2400
Needle	1200	4500	600
El AyMeDuele	3400	200	200

Figure 17: IMG

```
In [34]: df / 100
```

```
Out[34]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	0.34	0.03	0.03
Cegatron	0.54	0.06	0.45
Pikame Mucho	2.23	0.10	0.23
Lazarillo de Tormes	0.78	0.02	0.76
Stevie Wonder	0.56	0.05	0.24
Needle	0.12	0.45	0.06
El AyMeDuele	0.34	0.02	0.02

```
In [36]: np.power(df, 2)
```

```
Out[36]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	1156	9	9
Cegatron	2916	36	2025
Pikame Mucho	49729	100	529
Lazarillo de Tormes	6084	4	5776
Stevie Wonder	3136	25	576
Needle	144	2025	36
El AyMeDuele	1156	4	4

Figure 18: IMG

## NaNs NOT A NUMBER

Como ya dijimos, los NaNs (Not a Number) son valores nulos en nuestro conjunto de datos. Son valores que por alguna razón no se encuentran en nuestro dataset. A la hora de coleccionar nuestros datos, al momento de transcribirlos o de almacenarlos, algo pasó que algunos de esos datos faltan en el dataset final. Pandas está diseñado para lidiar con estos datos fácilmente.

Veamos un DataFrame con valores NaN (estamos usando el objeto de numpy `np.nan` para crear valores Nan):

```
In [59]: datos = {
    'precio': [34, 54, np.nan, np.nan, 56, 12, 34],
    'cantidad_en_stock': [3, 6, np.nan, np.nan, 5, 2, 10],
    'productos_vendidos': [3, 45, 23, np.nan, 24, 6, 23]
}

df = pd.DataFrame(datos, index=["Pokemaster", "Cegatron", "Pikame Mucho", "Lazarillo de Tormes", "Stevie Wonder", "Needle", "El AyMeDuele"])

In [60]: df
Out[60]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Pikame Mucho	NaN	NaN	23.0
Lazarillo de Tormes	NaN	NaN	NaN
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	23.0

Figure 19: IMG

Como puedes ver, a este DataFrame le faltan datos. Los valores NaN que contiene son valores nulos, y nos dicen nada útil, en realidad. Podríamos asumir cosas acerca de por qué faltan estos datos, pero no serían más que suposiciones.

Ahora, ¿qué pasa con estos datos nulos? Su presencia puede ser problemática para algunos de los análisis que queremos realizar.

En este caso nuestro dataset es muy pequeño y podemos visualizarlo todo de un solo vistazo. Pero vamos a imaginar que nuestro dataset es mucho más grande y que necesitamos saber si hay NaNs y cuántos. Para lograr esto podemos usar una función vectorizada llamada `isna` que chequea cada valor en nuestro DataFrame, lo transforma en `True` cuando el valor es igual a NaN y a `False` cuando no lo es:

```
In [61]: df.isna()
Out[61]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	False	False	False
Cegatron	False	False	False
Pikame Mucho	True	True	False
Lazarillo de Tormes	True	True	True
Stevie Wonder	False	False	False
Needle	False	False	False
El AyMeDuele	False	False	False

Figure 20: IMG

Ok, ¿y ahora qué hacemos con esto? Podemos usar la agregación `sum` para hacer un conteo de nuestros valores nulos. Si recuerdas, `sum` sumaba todos los valores de cada columna y regresaba el total por columna. Si sumas valores booleanos, los `Trues` cuentan como 1 y los `Falses` cuentan como 0. Esto significa que aplicando la función, obtendremos el total de valores nulos en cada columna:

De esta manera podemos saber si hay columnas que tienen demasiados valores nulos como para ser utilizadas.



```
In [65]: df.isna().sum()
Out[65]: precio                2
cantidad_en_stock            2
productos_vendidos           1
dtype: int64
```

Figure 21: IMG

```
In [61]: df.isna()
Out[61]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	False	False	False
Cegatron	False	False	False
Pikame Mucho	True	True	False
Lazarillo de Tormes	True	True	True
Stevie Wonder	False	False	False
Needle	False	False	False
El AyMeDuele	False	False	False

Figure 22: IMG

También podemos obtener el número de NaNs que hay en cada fila pasándole un argumento a sum. Podemos indicarle a sum el eje en el cual queremos realizar la operación. En el caso de pandas eje se refiere a la dimensión de la estructura de datos. A estos ejes se les llama axis. Vamos a entender mejor los ejes más adelante cuando veamos aritmética de Series; por el momento basta con saber que si le pasamos axis=1 a sum nos regresa la suma de NaNs por índice:

```
In [81]: df.isna().sum(axis=1)
Out[81]: Pokemaster                0
Cegatron                0
Pikame Mucho             2
Lazarillo de Tormes      3
Stevie Wonder            0
Needle                   0
El AyMeDuele             0
dtype: int64
```

Figure 23: IMG

La decisión de qué hacer con los NaNs depende mucho del contexto. Vamos a ver 3 cosas básicas que podemos realizar para limpiar estos datos indeseables:

- Eliminar filas con NaNs
- Eliminar columnas con NaNs
- Llenar los NaNs con algún valor.

## ELIMINAR FILAS CON NaNs

Usando el método dropna, la eliminación de filas con NaNs se vuelve muy sencillo. Sólo basta con llamar dropna y todas las filas que contienen NaNs son eliminadas:

Esta operación no modifica el DataFrame original, así que si queremos que el cambio persista tenemos que asignarlo a un nuevo DataFrame:

En caso de que queramos eliminar solamente las filas donde TODOS los valores sean NaN, podemos pasarle el argumento how='all':

En caso de que queramos eliminar solamente las filas donde TODOS los valores sean NaN, podemos pasarle el argumento how='all':

```
In [41]: df.dropna()
```

```
Out[41]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	23.0

Figure 24: IMG

```
In [42]: df_dropped = df.dropna()
df_dropped
```

```
Out[42]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	23.0

Figure 25: IMG

```
In [43]: df_dropped = df.dropna(how='all')
df_dropped
```

```
Out[43]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Pikame Mucho	NaN	NaN	23.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	23.0

Figure 26: IMG

```
In [43]: df_dropped = df.dropna(how='all')
df_dropped
```

```
Out[43]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Pikame Mucho	NaN	NaN	23.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	23.0

Figure 27: IMG

El valor default es `how='any'`, que elimina las filas donde haya mínimo un NaN.

## ELIMINAR COLUMNAS NaNs

Ahora, ¿qué pasa si quisiéramos eliminar NaNs por columna? Vamos a agregar una columna a nuestro DataFrame que contenga puros NaNs:

```
In [47]: df['descuento'] = np.nan
In [48]: df
Out[48]:
```

	precio	cantidad_en_stock	productos_vendidos	descuento
Pokemaster	34.0	3.0	3.0	NaN
Cogatron	54.0	8.0	45.0	NaN
Pikame Mucho	NaN	NaN	23.0	NaN
Lazarillo de Tormes	NaN	NaN	NaN	NaN
Stavie Wonder	56.0	5.0	24.0	NaN
Needle	12.0	2.0	6.0	NaN
El AyMeDuelo	34.0	10.0	23.0	NaN

Figure 28: IMG

Si te fijas, sólo bastó con llamar `df['descuento'] = np.nan` para conseguir una columna completa de NaNs. Esto tiene que ver con la aritmética de Series. Cuando asigno un solo valor a una Serie, automáticamente toda la Serie toma ese valor.

Para eliminar columnas donde haya NaNs, llamamos también `dropna` pero con el argumento `axis=1`:

```
In [49]: df.dropna(axis=1)
Out[49]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cogatron	54.0	8.0	45.0
Pikame Mucho	NaN	NaN	23.0
Lazarillo de Tormes	NaN	NaN	NaN
Stavie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuelo	34.0	10.0	23.0

Figure 29: IMG

Como aquí también el valor default es `how=any`, pandas elimina todas las columnas donde haya mínimo un NaN, que en este caso son todas. Para que elimine sólo las columnas donde todos los valores sean NaNs, hay que pasarle el argumento `how=all`:

## LLENAR VALORES NULOS CON UN VALOR

Digamos que tenemos ahora un DataFrame que se ve así:

Nuestra primera acción debería de ser eliminar las filas y columnas donde todos los valores sean NaN, porque no nos sirven de nada:

Ahora, ¿qué debemos hacer con valores nulos que nos quedan? Digamos que nuestro análisis más importante tiene que ver con la columna 'precio', entonces esa columna es muy importante que esté limpia. Pero digamos que nuestra 'productos\_vendidos' no es tan importante. Tal vez si hay un NaN en productos vendidos podemos asumir que no hay ningún producto vendido hasta ahora. En ese caso, podríamos llenar el/los NaN de la columna 'productos\_vendidos' con 0s. Eso se hace con el método `fillna`:

```
In [50]: df_dropped = df.dropna(axis=1, how='all')
df_dropped
```

```
Out[50]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Pikame Mucho	NaN	NaN	23.0
Lazarillo de Tormes	NaN	NaN	NaN
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	23.0

Figure 30: IMG

```
In [56]: df
```

```
Out[56]:
```

	precio	cantidad_en_stock	productos_vendidos	descuento
Pokemaster	34.0	3.0	3.0	NaN
Cegatron	54.0	6.0	45.0	NaN
Pikame Mucho	NaN	14.0	23.0	NaN
Lazarillo de Tormes	NaN	NaN	NaN	NaN
Stevie Wonder	56.0	5.0	24.0	NaN
Needle	12.0	2.0	6.0	NaN
El AyMeDuele	34.0	10.0	NaN	NaN

Figure 31: IMG

```
In [65]: df_dropped = df.dropna(axis=0, how='all')
df_dropped = df_dropped.dropna(axis=1, how='all')
df_dropped
```

```
Out[65]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Pikame Mucho	NaN	14.0	23.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	NaN

Figure 32: IMG

```
In [66]: df_dropped['productos_vendidos'].fillna(0)
```

```
Out[66]:
```

Pokemaster	3.0
Cegatron	45.0
Pikame Mucho	23.0
Stevie Wonder	24.0
Needle	6.0
El AyMeDuele	0.0

Name: productos\_vendidos, dtype: float64

Figure 33: IMG

Seleccionamos la columna donde queremos llenar los NaNs con 0 y llamamos el método `fillna(0)`. En este caso sólo estamos obteniendo de regreso la columna rellenada. Para tenerla en nuestro DataFrame podemos reasignarla a la misma columna:

```
In [67]: df_dropped['productos_vendidos'] = df_dropped['productos_vendidos'].fillna(0)
df_dropped
```

```
Out[67]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Pikame Mucho	NaN	14.0	23.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	0.0

Figure 34: IMG

¡Listo!

Ahora sí, podemos eliminar las filas que aún contengan NaNs porque sabemos que los NaNs que quedan son demasiados indeseables:

```
In [69]: df_cleaned = df_dropped.dropna()
df_cleaned
```

```
Out[69]:
```

	precio	cantidad_en_stock	productos_vendidos
Pokemaster	34.0	3.0	3.0
Cegatron	54.0	6.0	45.0
Stevie Wonder	56.0	5.0	24.0
Needle	12.0	2.0	6.0
El AyMeDuele	34.0	10.0	0.0

Figure 35: IMG

¡Y ya tenemos un dataset libre de valores nulos!

## APLICACIÓN EN NUESTRO DATASET ORIGINAL

Vamos a ver cómo funciona esto en nuestro dataset que teníamos al principio:

```
In [72]: df.head(5)
```

```
Out[72]:
```

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize	BuildingArea	YearBuilt	Co
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	...	1.0	1.0	126.0	NaN	NaN	
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	...	1.0	1.0	202.0	NaN	NaN	
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	...	1.0	0.0	156.0	79.0	1900.0	
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	...	2.0	1.0	0.0	NaN	NaN	
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	...	2.0	0.0	134.0	150.0	1900.0	

5 rows x 21 columns

Figure 36: IMG

Primero hacemos conteo de NaNs:

```
In [71]: df.isna().sum()
Out[71]: Suburb      0
Address      0
Rooms      0
Type      0
Price      4344
Method      0
SellerG      0
Date      0
Distance      8
Postcode      8
Bedroom2      4413
Bathroom      4413
Car      4413
Landsize      4796
BuildingArea      11123
YearBuilt      10389
CouncilArea      4444
Latitude      4292
Longitude      4292
Regionname      8
Propertycount      8
dtype: int64
```

Figure 37: IMG

Tenemos un total de filas de 19740, así que el hecho de que tengamos alrededor de 11000 valores NaNs en las columnas BuildingArea y YearBuilt no son una buena señal. Tal vez después cambiemos nuestra decisión, pero por el momento vamos a simplemente eliminarlas:

```
In [81]: df_2 = df.drop(columns=['BuildingArea', 'YearBuilt'])
df_2.isna().sum()
Out[81]: Suburb      0
Address      0
Rooms      0
Type      0
Price      4344
Method      0
SellerG      0
Date      0
Distance      8
Postcode      8
Bedroom2      4413
Bathroom      4413
Car      4413
Landsize      4796
CouncilArea      4444
Latitude      4292
Longitude      4292
Regionname      8
Propertycount      8
dtype: int64
```

Figure 38: IMG

Ok, ahora tenemos que decidir qué vamos a hacer con el resto de los NaNs. Presiento que no es muy grave no tener valores en 'Regionname', ya que es poco probable que usemos esa columna para nuestro análisis. Por lo tanto voy a llenar los NaNs con el valor Unknown:

El resto de las columnas voy a considerarlas esenciales, así que vamos a eliminar todas las filas donde todavía tengamos NaNs:

Y ahora vemos cuántas filas nos han quedado:

Si estas filas que nos quedan son suficientes o no, eso sólo lo sabremos continuando con el proceso.

## REINDEXANDO

Ahora, algo pasó con nuestro dataset después de eliminar los NaNs y es que nuestro índice ya no corresponde con el número de filas en nuestro dataset:

```
In [83]: df_2['Regionname'] = df_2['Regionname'].fillna('Unknown')
df_2.isna().sum()

Out[83]: Suburb      0
Address    0
Rooms      0
Type       0
Price      4344
Method     0
SellerG    0
Date       0
Distance   8
Postcode   8
Bedroom2   4413
Bathroom   4413
Car         4413
Landsize   4796
CouncilArea 4444
Latitude    4292
Longitude   4292
Regionname  0
Propertycount 8
dtype: int64
```

Figure 39: IMG

```
In [84]: df_dropped = df_2.dropna(axis=0, how='any')
df_dropped.isna().sum()

Out[84]: Suburb      0
Address    0
Rooms      0
Type       0
Price      0
Method     0
SellerG    0
Date       0
Distance   0
Postcode   0
Bedroom2   0
Bathroom   0
Car         0
Landsize   0
CouncilArea 0
Latitude    0
Longitude   0
Regionname  0
Propertycount 0
dtype: int64
```

Figure 40: IMG

```
In [85]: df_dropped.shape

Out[85]: (11646, 19)
```

Figure 41: IMG

19731	Whitlessa	30 Sherwin St	3	h	601000.0	S	Ray	29/07/2017	35.5	3757.0	3.0	2.0	2.0	1970.0	Marrin
19734	Williamstown	87 Pasco St	3	h	1285000.0	S	Jas	29/07/2017	6.8	3016.0	2.0	1.0	1.0	2010.0	White
19737	Yarraville	2 Adeney St	2	h	750000.0	SP	hockingstuart	29/07/2017	6.3	3013.0	3.0	2.0	2.0	1999.0	De
19738	Yarraville	54 Perland Rd	6	h	2450000.0	VR	Village	29/07/2017	6.3	3013.0	3.0	2.0	1.0	2011.0	i
19739	Yarraville	10/127 Somerville Rd	3	t	645000.0	SP	Jas	29/07/2017	6.3	3013.0	2.0	1.0	1.0	1980.0	i

11646 rows x 19 columns

Figure 42: IMG

Esto sucede porque eliminamos filas pero las filas que se mantuvieron siguen teniendo el mismo índice que antes. Hay veces que eso es lo que queremos (cuando nuestro índice son, nombres, etiquetas, letras, etc), pero en este caso, nos convendría que nuestro índice coincidiera con la posición de la fila en el dataset. Podemos corregir esto usando el método `reset_index` de la siguiente manera:

```
In [89]: df_dropped.reset_index().tail()
```

```
Out[89]:
```

	index	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize
11641	19731	Whittlesea	30 Sherwin St	3	h	601000.0	S	Ray	29/07/2017	35.5	3757.0	3.0	2.0	2.0	1970.0
11642	19734	Williamstown	87 Pasco St	3	h	1285000.0	S	Jas	29/07/2017	6.8	3016.0	2.0	1.0	1.0	2010.0
11643	19737	Yarraville	2 Adeney St	2	h	750000.0	SP	hockingstuart	29/07/2017	6.3	3013.0	3.0	2.0	2.0	1999.0
11644	19738	Yarraville	54 Pentland Pde	6	h	2450000.0	VB	Village	29/07/2017	6.3	3013.0	3.0	2.0	1.0	2011.0
11645	19739	Yarraville	10/127 Somerville Rd	3	t	645000.0	SP	Jas	29/07/2017	6.3	3013.0	2.0	1.0	1.0	1980.0

```
In [91]: df_dropped.shape
```

```
Out[91]: (11646, 19)
```

Figure 43: IMG

Como ves, el índice ahora corresponde con el número de filas que tenemos. El único problema es que ahora tenemos una columna llamada índice que contiene los índices anteriores. Una vez más, hay veces que queremos eso (cuando la información contenida ahí era relevante y no queremos deshacernos de ella), pero en este caso, en realidad no nos interesa mantener esa columna. Para resetear el índice y eliminarlo al mismo tiempo, usamos `reset_index(drop=True)`:

```
In [93]: df_dropped.reset_index(drop=True).tail()
```

```
Out[93]:
```

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize	Council
11641	Whittlesea	30 Sherwin St	3	h	601000.0	S	Ray	29/07/2017	35.5	3757.0	3.0	2.0	2.0	1970.0	Manning
11642	Williamstown	87 Pasco St	3	h	1285000.0	S	Jas	29/07/2017	6.8	3016.0	2.0	1.0	1.0	2010.0	Whitti
11643	Yarraville	2 Adeney St	2	h	750000.0	SP	hockingstuart	29/07/2017	6.3	3013.0	3.0	2.0	2.0	1999.0	Dai
11644	Yarraville	54 Pentland Pde	6	h	2450000.0	VB	Village	29/07/2017	6.3	3013.0	3.0	2.0	1.0	2011.0	H
11645	Yarraville	10/127 Somerville Rd	3	t	645000.0	SP	Jas	29/07/2017	6.3	3013.0	2.0	1.0	1.0	1980.0	H

Figure 44: IMG

¡Listo!

## RENOMBRANDO COLUMNAS

Además del índice, otros identificadores que tenemos que nos interesa mantener siempre limpios y claros son los nombres de nuestras columnas. En el caso de nuestro dataset, los nombres no son suficientemente homogéneos. Tenemos cosas como `Regionname` (segunda palabra con minúscula) y otras como `CouncilArea` (segunda palabra con mayúscula). Además, hay errores ortográficos (latitude, longitude). Y también que prefiero que los nombres sigan la convención de nombramiento de Python (snake\_case). Voy a renombrar mis columnas para que sigan las mismas convenciones.

Normalmente renombramos columnas cuando los nombres:



- No son lo suficientemente claros
- No representan la información que contiene esa columna.
- Tienen información basura.
- Tienen errores ortográficos.
- No siguen la convención que hemos decidido que deberían de tener.
- Son demasiado largos o difíciles de escribir.

Cambiamos entonces nuestros nombres de columnas. Primero creamos un mapa de los nombres viejos a los nombres nuevos:

```
In [62]: column_name_mapping = {
        'Suburb': 'suburb',
        'Address': 'address',
        'Rooms': 'rooms',
        'Type': 'type',
        'Price': 'price',
        'Method': 'method',
        'SellerG': 'seller_g',
        'Date': 'date',
        'Distance': 'distance',
        'Postcode': 'post_code',
        'Bedroom2': 'bedrooms',
        'Bathroom': 'bathroom',
        'Car': 'car',
        'Landsize': 'land_size',
        'CouncilArea': 'council_area',
        'Latitude': 'latitude',
        'Longitude': 'longitude',
        'Regionname': 'region_name',
        'Propertycount': 'property_count'
    }
```

Figure 45: IMG

Y ahora usamos el método rename para cambiar los nombres:

```
In [63]: df_renamed = df_dropped.rename(columns=column_name_mapping)
df_renamed.head()
```

```
Out[63]:
```

	suburb	address	rooms	type	price	method	seller_g	date	distance	post_code	bedrooms	bathroom	car	land_size	council_area	latitude
0	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	2.0	1.0	1.0	202.0	Yarra	-37.75
1	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	2.0	1.0	0.0	156.0	Yarra	-37.86
2	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	3.0	2.0	0.0	134.0	Yarra	-37.86
3	Abbotsford	40 Federation Ln	3	h	850000.0	Pt	Biggin	4/03/2017	2.5	3067.0	3.0	2.0	1.0	94.0	Yarra	-37.75
4	Abbotsford	55a Park St	4	h	1600000.0	VII	Nelson	4/06/2016	2.5	3067.0	3.0	1.0	2.0	120.0	Yarra	-37.86

Figure 46: IMG

Esto es todo por hoy. ¡Nos vemos en el Work para practicar todo lo que aprendimos!

## WORK SESION 5. FUNCIONES VECTORIZADAS Y LIMPIEZA DE DATOS

### OBJETIVOS

1. Identificar y utilizar las funciones vectorizadas.
2. Identificar agregaciones/reducciones.

3. Leer un CSV.
4. Encontrar y limpiar datos nulos.
5. Reindexar y cambiar el nombre de las columnas.

## CONTENIDO

### INTRODUCCIÓN

El día de hoy vamos a aprender a limpiar un poco nuestros datasets. Necesitamos limpiar nuestros datasets para facilitarnos los procesos posteriores de análisis y visualización. Trabajar con un dataset sucio es muy difícil y frustrante.

Vamos a aprender a encontrar valores nulos en nuestro dataset y limpiarlos.

Pero para poder hacer esto, primero vamos a aprender dos herramientas que se llaman funciones vectorizadas y agregaciones que expandirán tus posibilidades muchísimo.

## ARITMÉTICA CON SERIES Y FUNCIONES VECTORIZADAS

### EJEMPLO 1. FUNCIONES VECTORIZADAS CON SERIES

#### OBJETIVOS

- A prender cómo utilizar las funciones vectorizadas aplicadas a Series de pandas

#### DESARROLLO

Tenemos la siguiente serie:

```
import pandas as pd
serie_1 = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Recuerdas cómo utilizamos map para aplicar una función elemento por elemento a un arreglo. Podemos utilizar funciones vectorizadas para hacer esto mismo con Series y DataFrames de pandas. Esto resulta sumamente eficiente pues pandas está construido para funcionar de esta manera. Primero que nada, veamos cómo es posible aplicar operaciones aritméticas a Series de pandas y son aplicadas elemento por elemento. Por ejemplo:

```
serie_1 + 10
```

```
## 0    11
## 1    12
## 2    13
## 3    14
## 4    15
## 5    16
## 6    17
## 7    18
## 8    19
## 9    20
## dtype: int64
```

```
serie_1 * 10
```

```
## 0    10
## 1    20
## 2    30
## 3    40
## 4    50
## 5    60
## 6    70
## 7    80
## 8    90
## 9   100
## dtype: int64
```

```
(serie_1 + 10) * 100
```

```
## 0   1100
## 1   1200
## 2   1300
## 3   1400
## 4   1500
## 5   1600
## 6   1700
## 7   1800
## 8   1900
## 9   2000
## dtype: int64
```

```
serie_1 * 60 / 100
```

```
## 0    0.6
## 1    1.2
## 2    1.8
## 3    2.4
## 4    3.0
## 5    3.6
## 6    4.2
## 7    4.8
## 8    5.4
## 9    6.0
## dtype: float64
```

```
import numpy as np
np.power(serie_1, 2)
```

```
## 0     1
## 1     4
## 2     9
## 3    16
## 4    25
## 5    36
```

```
## 6      49
## 7      64
## 8      81
## 9     100
## dtype: int64
```

```
np.sqrt(serie_1)
```

```
## 0      1.000000
## 1      1.414214
## 2      1.732051
## 3      2.000000
## 4      2.236068
## 5      2.449490
## 6      2.645751
## 7      2.828427
## 8      3.000000
## 9      3.162278
## dtype: float64
```

¿Ves qué fácil resulta?

## EJEMPLO 2. AGREGACIONES

Las agregaciones son una variación de las funciones vectorizadas. Lo que hacen es tomar un arreglo (una Serie, por ejemplo), aplicar una operación a todos los elementos y regresar un resultado único que es la agregación o reducción del arreglo. Una agregación se ve así:

### OBJETIVOS

- Aprender cómo usar agregaciones para resumir o reducir un arreglo

### DESARROLLO

Las agregaciones entonces aplican una función a todo el arreglo entero y regresan un único valor que es la agregación o reducción del arreglo.

pandas ya tiene incluidas bastantes de éstas. Así que podemos llamarlas con tan sólo usar un método de nuestra Serie:

```
import pandas as pd
serie = pd.Series([1, 2, 3, 4, 5])
```

sum suman todos los elementos de nuestro arreglo:

```
serie.sum()
```

```
## 15
```

min y max nos dan el valor mínimo y máximo, respectivamente, de nuestro arreglo:

```
serie.min()
```

```
## 1
```

```
serie.max()
```

```
## 5
```

count nos da el conteo total del número de elementos en nuestro arreglo:

```
serie.count()
```

```
## 5
```

## EJEMPLO 3. FUNCIONES VECTORIZADAS Y AGREGACIONES CON DATAFRAMES

También podemos aplicar estas herramientas a DataFrames completos. Tanto las operaciones aritméticas, funciones vectorizadas y agregaciones funcionan con ligeras diferencias de procedimiento.

### OBJETIVO

- Aprender cómo usar Funciones vectorizadas y agregaciones aplicadas a DataFrames completos

### DESARROLLO

```
import pandas as pd
```

Tenemos el siguiente dataset:

```
datos = {
    'precio': [34, 54, 223, 78, 56, 12, 34],
    'cantidad_en_stock': [3, 6, 10, 2, 5, 45, 2],
    'productos_vendidos': [3, 45, 23, 76, 24, 6, 2]
}
```

```
df = pd.DataFrame(datos, index=["Pokemaster", "Cegatron", "Pikame Mucho", "Lazarillo de Tormes", "Stevie Wonder", "Needle", "El AyMeDuele"])
df
```

	precio	cantidad_en_stock	productos_vendidos
## Pokemaster	34	3	3
## Cegatron	54	6	45
## Pikame Mucho	223	10	23
## Lazarillo de Tormes	78	2	76
## Stevie Wonder	56	5	24
## Needle	12	45	6
## El AyMeDuele	34	2	2

Si aplicamos operaciones aritméticas a nuestro DataFrame la operación se aplicará elemento por elemento a nuestro DataFrame completo:

```
df * 100
```

	precio	cantidad_en_stock	productos_vendidos
##			
## Pokemaster	3400	300	300
## Cegatron	5400	600	4500
## Pikame Mucho	22300	1000	2300
## Lazarillo de Tormes	7800	200	7600
## Stevie Wonder	5600	500	2400
## Needle	1200	4500	600
## El AyMeDuele	3400	200	200

```
(df + 100) / 2
```

	precio	cantidad_en_stock	productos_vendidos
##			
## Pokemaster	67.0	51.5	51.5
## Cegatron	77.0	53.0	72.5
## Pikame Mucho	161.5	55.0	61.5
## Lazarillo de Tormes	89.0	51.0	88.0
## Stevie Wonder	78.0	52.5	62.0
## Needle	56.0	72.5	53.0
## El AyMeDuele	67.0	51.0	51.0

También podemos aplicar funciones vectorizadas con el mismo resultado:

```
import numpy as np
np.power(df, 2)
```

	precio	cantidad_en_stock	productos_vendidos
##			
## Pokemaster	1156	9	9
## Cegatron	2916	36	2025
## Pikame Mucho	49729	100	529
## Lazarillo de Tormes	6084	4	5776
## Stevie Wonder	3136	25	576
## Needle	144	2025	36
## El AyMeDuele	1156	4	4

```
np.sqrt(df)
```

	precio	cantidad_en_stock	productos_vendidos
##			
## Pokemaster	5.830952	1.732051	1.732051
## Cegatron	7.348469	2.449490	6.708204
## Pikame Mucho	14.933185	3.162278	4.795832
## Lazarillo de Tormes	8.831761	1.414214	8.717798
## Stevie Wonder	7.483315	2.236068	4.898979
## Needle	3.464102	6.708204	2.449490
## El AyMeDuele	5.830952	1.414214	1.414214

```
np.sin(df) + 100
```

```
##                precio  cantidad_en_stock  productos_vendidos
## Pokemaster        100.529083        100.141120        100.141120
## Cegatron          99.441211         99.720585        100.850904
## Pikame Mucho      100.053053         99.455979         99.153780
## Lazarillo de Tormes 100.513978        100.909297        100.566108
## Stevie Wonder     99.478449         99.041076         99.094422
## Needle            99.463427        100.850904         99.720585
## El AyMeDuele      100.529083        100.909297        100.909297
```

Si usamos agregaciones, las agregaciones se hacen de manera automática por columna:

```
df.sum()
```

```
## precio          491
## cantidad_en_stock    73
## productos_vendidos   179
## dtype: int64
```

Aunque podemos cambiar ese comportamiento usando `axis=1` para hacerlo por fila:

```
df.sum(axis=1)
```

```
## Pokemaster          40
## Cegatron            105
## Pikame Mucho        256
## Lazarillo de Tormes 156
## Stevie Wonder        85
## Needle              63
## El AyMeDuele        38
## dtype: int64
```

Todas las demás agregaciones funcionan también. El default (o `axis=0`) es hacerlo por columna, pero todas pueden funcionar por fila usando `axis=1`:

```
df.min()
```

```
## precio          12
## cantidad_en_stock    2
## productos_vendidos    2
## dtype: int64
```

```
df.min(axis=1)
```

```
## Pokemaster          3
## Cegatron            6
## Pikame Mucho        10
## Lazarillo de Tormes  2
## Stevie Wonder        5
## Needle              6
## El AyMeDuele        2
## dtype: int64
```

```
df.max()
```

```
## precio                223
## cantidad_en_stock     45
## productos_vendidos     76
## dtype: int64
```

```
df.max(axis=1)
```

```
## Pokemaster           34
## Cegatron             54
## Pikame Mucho         223
## Lazarillo de Tormes   78
## Stevie Wonder        56
## Needle               45
## El AyMeDuele         34
## dtype: int64
```

## EJEMPLO 4. IDENTIFICACION DE VALORES NAN O VALORES NULOS Y CONTEO

Como viste en tu Prework, los valores NaN (Not a Number) son bastante indeseables porque no podemos utilizarlos para realizar análisis estadístico u operaciones aritméticas. Es por eso que uno de los primeros pasos en la Limpieza de Datos suele ser la eliminación de estos valores.

### OBJETIVOS

- Aprender a identificar NaNs
- Aprender a realizar conteo de NaNs por fila y por columna

### DESARROLLO

Los NaNs se ven así:

```
import pandas as pd
import numpy as np
```

```
datos = {
    'precio': [34, 54, np.nan, np.nan, 56, 12, 34],
    'cantidad_en_stock': [3, 6, 14, np.nan, 5, 2, 10],
    'productos_vendidos': [3, 45, 23, np.nan, 24, 6, np.nan]
}
```

```
df = pd.DataFrame(datos, index=["Pokemaster", "Cegatron", "Pikame Mucho", "Lazarillo de Tormes", "Stevie Wonder"])
df
```

```
##                precio  cantidad_en_stock  productos_vendidos
## Pokemaster         34.0                3.0                3.0
```



## Cegatron	54.0	6.0	45.0
## Pikame Mucho	NaN	14.0	23.0
## Lazarillo de Tormes	NaN	NaN	NaN
## Stevie Wonder	56.0	5.0	24.0
## Needle	12.0	2.0	6.0
## El AyMeDuele	34.0	10.0	NaN

Para contarlos podemos usar una función vectorizada llamada `isna`, que nos regresa esto:

```
df.isna()
```

##	precio	cantidad_en_stock	productos_vendidos
## Pokemaster	False	False	False
## Cegatron	False	False	False
## Pikame Mucho	True	False	False
## Lazarillo de Tormes	True	True	True
## Stevie Wonder	False	False	False
## Needle	False	False	False
## El AyMeDuele	False	False	True

`isna` regresa `True` cuando encuentra un `NaN` y `False` cuando el valor es válido.

Después, podemos contar cuántos `NaNs` existen usando la agregación `sum`, que suma 1 por cada `True` y 0 por cada `False`:

```
df.isna().sum(axis=0)
```

```
## precio          2
## cantidad_en_stock  1
## productos_vendidos  2
## dtype: int64
```

Con `axis=0` nos regresa el conteo por columnas. Con `axis=1` nos regresa el conteo por filas:

```
df.isna().sum(axis=1)
```

```
## Pokemaster      0
## Cegatron        0
## Pikame Mucho    1
## Lazarillo de Tormes  3
## Stevie Wonder   0
## Needle          0
## El AyMeDuele    1
## dtype: int64
```

Practiquemos rápidamente esto antes de aprender a deshacernos de estos `NaNs`.

## EJEMPLO 5. LIMPIEZA DE NANS

Hay 3 operaciones básicas que podemos realizar para eliminar `NaNs` de nuestros datasets:

1. Eliminar filas con NaNs
2. Eliminar columnas con NaNs
3. Llenar los NaNs con algún valor.

Exploraremos las 3 opciones

## OBJETIVOS

- Aprender a limpiar NaNs por filas
- Aprender a limpiar NaNs por columnas
- Aprender a llenar NaNs con otros valores útiles

## DESARROLLO

### LIMPIANDO NANS POR FILAS

Tenemos el siguiente dataset

```
import pandas as pd
import numpy as np

datos = {
    'precio': [34, 54, np.nan, np.nan, 56, 12, 34],
    'cantidad_en_stock': [3, 6, 14, np.nan, 5, 2, 10],
    'productos_vendidos': [3, 45, 23, np.nan, 24, 6, np.nan]
}

df = pd.DataFrame(datos, index=["Pokemaster", "Cegatron", "Pikame Mucho", "Lazarillo de Tormes", "Stevie Wonder", "Needle", "El AyMeDuele"])

df
```

	precio	cantidad_en_stock	productos_vendidos
## Pokemaster	34.0	3.0	3.0
## Cegatron	54.0	6.0	45.0
## Pikame Mucho	NaN	14.0	23.0
## Lazarillo de Tormes	NaN	NaN	NaN
## Stevie Wonder	56.0	5.0	24.0
## Needle	12.0	2.0	6.0
## El AyMeDuele	34.0	10.0	NaN

Para limpiar las filas que tengan mínimo 1 valor NaN, se utiliza `dropna(axis=0, how='any')`:

```
df.dropna(axis=0, how='any')
```

	precio	cantidad_en_stock	productos_vendidos
## Pokemaster	34.0	3.0	3.0
## Cegatron	54.0	6.0	45.0
## Stevie Wonder	56.0	5.0	24.0
## Needle	12.0	2.0	6.0

Con el `axis=0` le estamos diciendo que queremos eliminar por filas. Con `how='any'` le decimos que queremos eliminar cualquier fila que tenga mínimo un NaN.

Si quisiéramos eliminar sólo las filas donde todos los valores sean NaN, podemos usar `axis='all'`:

```
df.dropna(axis=0, how='all')
```

```
##           precio  cantidad_en_stock  productos_vendidos
## Pokemaster    34.0             3.0             3.0
## Cegatron      54.0             6.0            45.0
## Pikame Mucho   NaN            14.0            23.0
## Stevie Wonder  56.0             5.0            24.0
## Needle        12.0             2.0             6.0
## El AyMeDuele  34.0            10.0             NaN
```

Estos resultados no se aplican directamente al DataFrame original. Si queremos que persistan tenemos que asignarlos a otra variable:

```
df_dropped = df.dropna(axis=0, how='all')
```

## LIMPIANDO NANS POR COLUMNAS

Vamos a agregar una columna:

```
df['descuento'] = np.nan
df
```

```
##           precio  cantidad_en_stock  productos_vendidos  descuento
## Pokemaster    34.0             3.0             3.0         NaN
## Cegatron      54.0             6.0            45.0         NaN
## Pikame Mucho   NaN            14.0            23.0         NaN
## Lazarillo de Tormes  NaN            NaN            NaN         NaN
## Stevie Wonder  56.0             5.0            24.0         NaN
## Needle        12.0             2.0             6.0         NaN
## El AyMeDuele  34.0            10.0             NaN         NaN
```

Al igual que por filas, eliminar NaNs por columna también se puede hacer usando `'any'` y `'all'`. La única diferencia es que ahora hay que usar `axis=1` para que se haga la eliminación por columnas:

```
df.dropna(axis=1, how='any')
```

```
## Empty DataFrame
## Columns: []
## Index: [Pokemaster, Cegatron, Pikame Mucho, Lazarillo de Tormes, Stevie Wonder, Needle, El AyMeDuele]
```

```
df_dropped = df.dropna(axis=1, how='all')
```

## LLENANDO NANS CON VALORES

Otra cosa que podemos hacer es llenar los valores NaN con algún otro valor.

Por ejemplo, digamos que tenemos este dataset:

```
df
```

```
##          precio  cantidad_en_stock  productos_vendidos  descuento
## Pokemaster    34.0             3.0             3.0         NaN
## Cegatron      54.0             6.0            45.0         NaN
## Pikame Mucho   NaN             14.0            23.0         NaN
## Lazarillo de Tormes   NaN             NaN             NaN         NaN
## Stevie Wonder  56.0             5.0            24.0         NaN
## Needle        12.0             2.0             6.0         NaN
## El AyMeDuele   34.0            10.0             NaN         NaN
```

Lo primero que hay que hacer es eliminar filas y columnas donde todos los valores sean NaN, puesto que no nos sirven de nada:

```
df_no_nans = df.dropna(axis=0, how='all')
df_no_nans = df_no_nans.dropna(axis=1, how='all')
```

```
df_no_nans
```

```
##          precio  cantidad_en_stock  productos_vendidos
## Pokemaster    34.0             3.0             3.0
## Cegatron      54.0             6.0            45.0
## Pikame Mucho   NaN             14.0            23.0
## Stevie Wonder  56.0             5.0            24.0
## Needle        12.0             2.0             6.0
## El AyMeDuele   34.0            10.0             NaN
```

Ahora, digamos que podemos asumir que si hay un valor NaN en “productos\_vendidos” es porque no ha sido vendido aún. En ese caso podemos rellenar ese NaN usando fillna:

```
df_no_nans['productos_vendidos'] = df_no_nans['productos_vendidos'].fillna(0)
```

```
df_no_nans
```

```
##          precio  cantidad_en_stock  productos_vendidos
## Pokemaster    34.0             3.0             3.0
## Cegatron      54.0             6.0            45.0
## Pikame Mucho   NaN             14.0            23.0
## Stevie Wonder  56.0             5.0            24.0
## Needle        12.0             2.0             6.0
## El AyMeDuele   34.0            10.0             0.0
```

Para finalizar, “precio” sí es una variable muy importante, así que nos deshacemos de las filas que aún tengan NaNs:

```
df_no_nans.dropna(axis=0)
```

```
##          precio  cantidad_en_stock  productos_vendidos
## Pokemaster    34.0             3.0             3.0
## Cegatron      54.0             6.0            45.0
## Stevie Wonder  56.0             5.0            24.0
## Needle        12.0             2.0             6.0
## El AyMeDuele   34.0            10.0             0.0
```

## EJEMPLO 6. APLICANDO LOS CONOCIMIENTOS A UN DATASET REAL-LIMPIEZA DE NANS EN UN DATASET REAL-

¡Vamos a ver un pequeño ejemplo donde vamos a aplicar lo que hemos visto el día de hoy a un dataset real!

Este dataset está en formato CSV, que quiere decir que cada columna está separada por una coma. Las líneas de nuestro archivo .csv son cada una las filas de nuestro dataset, y los datos en cada fila, separados por comas (,), conforman las columnas:

```
Suburb,Address,Rooms,Type,Price,Method,SellerG,Date,Distance,Postcode,Bedroom2,Bathroom,Car,Landsize,BuildingArea,YearBuilt,CouncilArea,Lattitude,Longitude,Regionname,Propertycount
Abbotsford,68 Studley St,2,h,,SS,Jellis,3/09/2016,2.5,3067,2,1,1,126,,Yarra,-37.8814,144.9958,Northern Metropolitan,4019
Abbotsford,85 Turner St,2,h,1480000,S,Biggin,3/12/2016,2.5,3067,2,1,1,282,,Yarra,-37.7996,144.9984,Northern Metropolitan,4019
Abbotsford,25 Bloomburg St,2,h,1035000,S,Biggin,4/02/2016,2.5,3067,2,1,0,156,79,1900,Yarra,-37.8879,144.9934,Northern Metropolitan,4019
Abbotsford,18/659 Victoria St,3,u,,VB,Rounds,4/02/2016,2.5,3067,3,2,1,0,,Yarra,-37.8114,145.0116,Northern Metropolitan,4019
Abbotsford,5 Charles St,3,h,1465000,SP,Biggin,4/03/2017,2.5,3067,3,2,0,134,150,1900,Yarra,-37.8893,144.9944,Northern Metropolitan,4019
Abbotsford,40 Federation La,3,h,850000,PI,Biggin,4/03/2017,2.5,3067,3,2,1,94,,Yarra,-37.7969,144.9969,Northern Metropolitan,4019
Abbotsford,55a Park St,4,h,1680000,VB,Nelson,4/06/2016,2.5,3067,3,1,2,120,142,2014,Yarra,-37.8072,144.9941,Northern Metropolitan,4019
Abbotsford,16 Maugie St,4,h,,SN,Nelson,6/08/2016,2.5,3067,3,2,2,400,220,2006,Yarra,-37.7965,144.9965,Northern Metropolitan,4019
Abbotsford,53 Turner St,2,h,,S,Biggin,6/08/2016,2.5,3067,4,1,2,201,,1900,Yarra,-37.7995,144.9974,Northern Metropolitan,4019
```

Figure 47: IMG

## DESARROLLO

```
import pandas as pd
```

Para leer un archivo .csv en pandas, usamos read\_csv y le indicamos que el separador (el signo que delimita las columnas en el archivo .csv) es una coma:

```
import pandas as pd
df=pd.read_csv('https://raw.githubusercontent.com/beduExpert/Procesamiento-de-Datos-con-Python-Santander')
df
```

```
##          Suburb          Address ...          Regionname Propertycount
## 0      Abbotsford      68 Studley St ... Northern Metropolitan      4019.0
## 1      Abbotsford      85 Turner St ... Northern Metropolitan      4019.0
## 2      Abbotsford      25 Bloomburg St ... Northern Metropolitan      4019.0
## 3      Abbotsford      18/659 Victoria St ... Northern Metropolitan      4019.0
## 4      Abbotsford          5 Charles St ... Northern Metropolitan      4019.0
## ...          ...          ...          ...          ...
## 19735      Windsor      201/152 Peel St ... Southern Metropolitan      4380.0
## 19736      Wollert      60 Saltlake Bvd ... Northern Metropolitan      2940.0
## 19737      Yarraville          2 Adeney St ... Western Metropolitan      6543.0
## 19738      Yarraville      54 Pentland Pde ... Western Metropolitan      6543.0
## 19739      Yarraville      10/127 Somerville Rd ... Western Metropolitan      6543.0
##
## [19740 rows x 21 columns]
```

```
df.shape
```

```
## (19740, 21)
```

```
df.head(5)
```

```
##          Suburb          Address ...          Regionname Propertycount
## 0  Abbotsford      68 Studley St ... Northern Metropolitan      4019.0
## 1  Abbotsford      85 Turner St  ... Northern Metropolitan      4019.0
## 2  Abbotsford     25 Bloomburg St ... Northern Metropolitan      4019.0
## 3  Abbotsford  18/659 Victoria St ... Northern Metropolitan      4019.0
## 4  Abbotsford       5 Charles St  ... Northern Metropolitan      4019.0
##
## [5 rows x 21 columns]
```

```
df.isna().sum()
```

```
## Suburb          0
## Address         0
## Rooms           0
## Type            0
## Price          4344
## Method          0
## SellerG         0
## Date            0
## Distance        8
## Postcode        8
## Bedroom2       4413
## Bathroom       4413
## Car            4413
## Landsize       4796
## BuildingArea   11123
## YearBuilt      10389
## CouncilArea    4444
## Lattitude      4292
## Longtitude     4292
## Regionname      8
## Propertycount   8
## dtype: int64
```

Éste es el número de columnas con el que nos quedamos:

```
df_dropped.shape
```

```
## (7, 3)
```

Guardemos el resultado:

```
df_dropped.to_csv('C:/Users/Victor Miguel Terron/Documents/PHASE2/DATA-SCIENCE-2PHASE/DATA PROCESSING A
```

Seguiremos trabajando este dataset en el último ejemplo.

## EJEMPLO 7. REINDEXANDO Y RENOMBRANDO COLUMNAS

Tenemos ahora un dataset que ha sido limpiado de NaNs. Tenemos ahora dos problemas. El primero es que nuestro índice no corresponde al número de filas que tenemos ahora:

19731	Whittlesea	30 Sherwin St	3	h	601000.0	S	Ray	29/07/2017	35.5	3757.0	3.0	2.0	2.0	1970.0	Mannin
19734	Williamstown	87 Pasco St	3	h	1285000.0	S	Jas	29/07/2017	6.8	3016.0	2.0	1.0	1.0	2010.0	Whitt
19737	Yarraville	2 Adeney St	2	h	750000.0	SP	hockingstuart	29/07/2017	6.3	3013.0	3.0	2.0	2.0	1999.0	De
19738	Yarraville	54 Pentland Pde	6	h	2450000.0	VB	Village	29/07/2017	6.3	3013.0	3.0	2.0	1.0	2011.0	I
19739	Yarraville	10/127 Somerville Rd	3	t	645000.0	SP	Jas	29/07/2017	6.3	3013.0	2.0	1.0	1.0	1980.0	I

11646 rows × 19 columns

Figure 48: IMG

```
In [73]: df_dropped.columns
Out[73]: Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG',
               'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car',
               'Landsize', 'CouncilArea', 'Latitude', 'Longitude', 'Regionname',
               'Propertycount'],
              dtype='object')
```

Figure 49: IMG

## OBJETIVOS

- Limpiar un poco más nuestro dataset asignándole un índice y nombres de columnas apropiadas

## DESARROLLO

Limpiemos nuestro dataset hasta que esté justo como lo dejamos en el Ejemplo pasado:

```
import pandas as pd
df = pd.read_csv('https://raw.githubusercontent.com/beduExpert/Procesamiento-de-Datos-con-Python-Santander/refs/heads/main/data/cleaned_data.csv')
df_2 = df.drop(columns=['BuildingArea', 'YearBuilt'])
df_2['Regionname'] = df_2['Regionname'].fillna('Unknown')
df_dropped = df_2.dropna(axis=0, how='any')
df_dropped
```

##	Suburb	Address	...	Regionname	Propertycount
## 1	Abbotsford	85 Turner St	...	Northern Metropolitan	4019.0
## 2	Abbotsford	25 Bloomburg St	...	Northern Metropolitan	4019.0
## 4	Abbotsford	5 Charles St	...	Northern Metropolitan	4019.0
## 5	Abbotsford	40 Federation La	...	Northern Metropolitan	4019.0
## 6	Abbotsford	55a Park St	...	Northern Metropolitan	4019.0
## ...	...	...	...	...	...
## 19731	Whittlesea	30 Sherwin St	...	Northern Victoria	2170.0
## 19734	Williamstown	87 Pasco St	...	Western Metropolitan	6380.0
## 19737	Yarraville	2 Adeney St	...	Western Metropolitan	6543.0

```
## 19738      Yarraville      54 Pentland Pde ... Western Metropolitan      6543.0
## 19739      Yarraville 10/127 Somerville Rd ... Western Metropolitan      6543.0
##
## [11646 rows x 19 columns]
```

Ahora, tenemos dos situaciones:

1. La primera es que nuestro índice no coincide con el número de filas que tenemos. En este caso, dado que nuestro índice es secuencial y numérico, y no tiene ningún significado además de eso, nos convendría que reflejara la cantidad de filas que tenemos en nuestro dataset.

Para lograr eso vamos a usar el método `reset_index`:

```
df_dropped.reset_index()
```

```
##          index      Suburb ...      Regionname  Propertycount
## 0           1  Abbotsford ... Northern Metropolitan      4019.0
## 1           2  Abbotsford ... Northern Metropolitan      4019.0
## 2           4  Abbotsford ... Northern Metropolitan      4019.0
## 3           5  Abbotsford ... Northern Metropolitan      4019.0
## 4           6  Abbotsford ... Northern Metropolitan      4019.0
## ...      ...      ...      ...      ...
## 11641  19731  Whittlesea ... Northern Victoria      2170.0
## 11642  19734  Williamstown ... Western Metropolitan      6380.0
## 11643  19737  Yarraville ... Western Metropolitan      6543.0
## 11644  19738  Yarraville ... Western Metropolitan      6543.0
## 11645  19739  Yarraville ... Western Metropolitan      6543.0
##
## [11646 rows x 20 columns]
```

Nuestro índice ya está correcto, pero ahora tenemos un columna llamada `index` que contiene el índice original. Como no queremos guardar esos datos, agregamos la opción `drop=True` para eliminar el índice anterior:

```
df_dropped.reset_index(drop=True)
```

```
##          Suburb      Address ...      Regionname  Propertycount
## 0  Abbotsford      85 Turner St ... Northern Metropolitan      4019.0
## 1  Abbotsford      25 Bloomburg St ... Northern Metropolitan      4019.0
## 2  Abbotsford       5 Charles St ... Northern Metropolitan      4019.0
## 3  Abbotsford     40 Federation La ... Northern Metropolitan      4019.0
## 4  Abbotsford      55a Park St ... Northern Metropolitan      4019.0
## ...      ...      ...      ...
## 11641  Whittlesea     30 Sherwin St ... Northern Victoria      2170.0
## 11642  Williamstown     87 Pasco St ... Western Metropolitan      6380.0
## 11643  Yarraville       2 Adeney St ... Western Metropolitan      6543.0
## 11644  Yarraville     54 Pentland Pde ... Western Metropolitan      6543.0
## 11645  Yarraville 10/127 Somerville Rd ... Western Metropolitan      6543.0
##
## [11646 rows x 19 columns]
```

Guardemos nuestros cambios:



```
df_dropped = df_dropped.reset_index(drop=True)
```

Ahora tenemos un problema con los nombres de las columnas: Tienen inconsistencias en la manera cómo están nombradas y algunas incluso tienen errores ortográficos. Vamos a cambiarles los nombres para tener consistencia:

```
column_name_mapping = {
    'Suburb': 'suburb',
    'Address': 'address',
    'Rooms': 'rooms',
    'Type': 'type',
    'Price': 'price',
    'Method': 'method',
    'SellerG': 'seller_g',
    'Date': 'date',
    'Distance': 'distance',
    'Postcode': 'post_code',
    'Bedroom2': 'bedrooms',
    'Bathroom': 'bathroom',
    'Car': 'car',
    'Landsize': 'land_size',
    'CouncilArea': 'council_area',
    'Lattitude': 'latitude',
    'Longtitude': 'longitude',
    'Regionname': 'region_name',
    'Propertycount': 'property_count'
}

df_renamed = df_dropped.rename(columns=column_name_mapping)

df_renamed
```

```
##           suburb ... property_count
## 0      Abbotsford ...          4019.0
## 1      Abbotsford ...          4019.0
## 2      Abbotsford ...          4019.0
## 3      Abbotsford ...          4019.0
## 4      Abbotsford ...          4019.0
## ...           ... ...              ...
## 11641  Whittlesea ...          2170.0
## 11642  Williamstown ...          6380.0
## 11643  Yarraville ...          6543.0
## 11644  Yarraville ...          6543.0
## 11645  Yarraville ...          6543.0
##
## [11646 rows x 19 columns]
```

¡Listo! Nuestro dataset va agarrando forma.

## RETO 1. FUNCIONES VECTORIZADAS

### PORCENTAJE TOTAL

Eres maestro en la H. Universidad de las Américas Unidas. Has realizado el examen final de la primera generación de estudiantes de la escuela. El conteo máximo de aciertos en el examen era de 68 (es decir, 68 aciertos equivale al 100% de las preguntas respondidas correctamente). La siguiente Serie reúne los aciertos obtenidos por los 25 alumnos de la generación:

```
import pandas as pd
aciertos = pd.Series([50, 55, 45, 65, 66, 46, 48, 53, 55, 56, 59, 68, 67, 60, 45, 56, 66, 64, 59, 55, 3
```

Tus calificaciones las das siempre en “porcentaje de aciertos”. Tu reto es convertir la Serie aciertos en la Serie porcentajes, que contiene cada valor de aciertos como un porcentaje del número de aciertos totales (68).

SÓLO puedes usar funciones vectorizadas de numpy para realizar tus cálculos. Aquí puedes encontrar las funciones que necesitas.

<https://www.interactivechaos.com/es/manual/tutorial-de-numpy/funciones-universales-matematicas>

```
## Realiza aquí tus cálculos
##
## ...
## ...

porcentajes =
```

## RETO 2. AGREGACIONES

### OBJETIVOS

- Usar funciones vectorizadas y agregaciones para computar la desviación estándar de un conjunto de datos

### DESARROLLO

#### DESVIACIÓN ESTANDAR

La desviación estándar es una medida que nos dice qué tan dispersos están los datos con respecto a la media. Es una de las medidas estadísticas más comunes e importantes. En este reto vamos a calcular la desviación estándar de un conjunto de datos usando funciones vectorizadas y agregaciones

Imagina que has realizado un censo en la H. Universidad de las Américas Unidas. Quieres saber qué tanta dispersión de edades hay en la universidad. Dada la naturaleza de la universidad, hay tanto alumnos extremadamente jóvenes (el más joven tiene 15 años) hasta alumnos bastante mayores (el alumno de más edad tiene 52 años). Para saber qué tan dispersas están las edades de los alumnos, vas a usar la desviación estándar.

El algoritmo para sacar la desviación estándar es el siguiente:

1. Saca el promedio de tu Serie. Esto se hace sumando todos tus datos y luego dividiéndolos entre la cantidad de datos (n)
2. Después toma tu Serie y réstale a cada elemento el promedio. De esta manera obtenemos una nueva Serie que contiene las diferencias entre cada dato y el promedio.
3. Después eleva tu Serie al cuadrado. Esto sirve para acentuar a los datos que están más alejados de tu promedio.

4. Ahora suma todos los elementos de tu Serie y divídelos entre la cantidad de datos de la Serie original menos 1 ( $n - 1$ ).
5. Por último, saca la raíz cuadrada del valor obtenido: Ésta es tu desviación estándar. Utiliza aritmética con Series, funciones vectorizadas y agregaciones para calcular esta estadística.

Asigna tu resultado final a la variable std.

```
import pandas as pd
#AGREGA OTRO IMPORT QUE NECESITES

edades = pd.Series([23, 24, 23, 34, 30, 17, 18, 24, 35, 28, 27, 27, 34, 32, 29, 16, 16, 17, 19, 34, 45,

## Realiza aquí tus cálculos
##
## ...
## ...

std =
```

## RETO 3. AGREGACIONES CON DATAFRAMES

### OBJETIVOS

- Aplicar agregaciones a DataFrames completos para obtener un análisis estadístico

### DESARROLLO

#### ANÁLISIS ESTADÍSTICO CON AGREGACIONES

Eres el Analista de Datos de EyePoker Inc. Te han pedido que realices ciertas agregaciones con un conjunto de datos para poder realizar un análisis estadístico básico de los datos que hay dentro.

El conjunto de datos es el siguiente:

```
import pandas as pd
#REALIZA CUALQUIER OTRA IMPORTACIÓN QUE NECESITES

datos = {
    'producto': ["Pokemaster", "Cegatron", "Pikame Mucho", "Lazarillo de Tormes", "Stevie Wonder", "Nee
    'precio': [12000, 5500, 2350, 4800, 8900, 6640, 1280, 1040, 23100, 16700, 15000, 13400, 19600],
    'cantidad_en_stock': [34, 54, 36, 78, 56, 12, 34, 4, 0, 18, 45, 23, 5],
    'cantidad_vendidos': [120, 34, 59, 9, 15, 51, 103, 72, 39, 23, 10, 62, 59]
}

df = pd.DataFrame(datos)
df

##                producto  precio  cantidad_en_stock  cantidad_vendidos
```

## 0	Pokemaster	12000	34	120
## 1	Cegatron	5500	54	34
## 2	Pikame Mucho	2350	36	59
## 3	Lazarillo de Tormes	4800	78	9
## 4	Stevie Wonder	8900	56	15
## 5	Needle	6640	12	51
## 6	El AyMeDuele	1280	34	103
## 7	El Desretinador	1040	4	72
## 8	Sacamel Ojocles	23100	0	39
## 9	Desojado	16700	18	23
## 10	Maribel Buenas Noches	15000	45	10
## 11	Cíclope	13400	23	62
## 12	El Cuatro Ojos	19600	5	59

Tu tarea es muy simple. Usando métodos de agregación, asigna las variables de la siguiente celda con los resultados de agregar nuestro DataFrame por columna usando cada una de las medidas estadísticas. Algunas de los métodos ya los conoces. Los que no, puedes encontrarlos en este link. Lo que queremos obtener es una Serie con los nombres de las columnas como índice y las agregaciones por columna como valores. Una de las columnas que tenemos en el DataFrame no se presta para realizar análisis numéricos, elimínala antes de realizar tu análisis y asigna el resultado a la variable `df_dropped`.

Sólo utiliza funciones de agregación para tu análisis. En este caso no requieres hacer ninguna operación aritmética.

```
df_dropped =

# El valor mínimo de cada columna
mins =

# El valor máximo de cada columna
maxs =

# El promedio por columna
means =

# La mediana por columna (El valor que se encuentra a la mitad de la secuencia ordenada de valores)
medians =

# La desviación estándar por columna
stds =
```

## RETO 4. IDENTIFICANDO Y LIMPIANDO NANS

### OBJETIVO

- Practicar la identificación de NaNs
- Practicar eliminar NaNs de un DataFrame usando diferentes técnicas

### DESARROLLO

#### Limpiando un dataset de NaNs

Eres el Data Wrangler de EyePoker Inc. Te han dado el siguiente dataset para que apliques algunas técnicas de procesamiento de datos:

```
import pandas as pd
import numpy as np

pd.options.mode.chained_assignment = None
#QUITA WARNINGS DE LA LIBREIRA PANDAS EN ALGUNOS CASOS
# LEER DOCUMENTACION

datos = {
    'precio': [12000, 5500, np.nan, 4800, 8900, np.nan, 1280, 1040, 23100, np.nan, 15000, 13400, np.nan],
    'cantidad_en_stock': [34, 54, np.nan, 78, 56, np.nan, 34, 4, 0, 18, 45, 23, 5],
    'cantidad_vendidos': [120, 34, np.nan, 9, 15, np.nan, 103, np.nan, np.nan, 23, 10, 62, 59],
    'descuentos': [np.nan] * 13
}

df = pd.DataFrame(datos, index=["Pokemaster", "Cegatron", "Pikame Mucho", "Lazarillo de Tormes", "Stevie Wonder", "Needle", "El AyMeDuele", "El Desretinador", "Sacamel Ojocles", "Desojado", "Maribel Buenas Noches", "Cíclope", "El Cuatro Ojos"])
df
```

```
##              precio  ... descuentos
## Pokemaster      12000.0  ...         NaN
## Cegatron         5500.0  ...         NaN
## Pikame Mucho         NaN  ...         NaN
## Lazarillo de Tormes  4800.0  ...         NaN
## Stevie Wonder      8900.0  ...         NaN
## Needle           NaN  ...         NaN
## El AyMeDuele       1280.0  ...         NaN
## El Desretinador     1040.0  ...         NaN
## Sacamel Ojocles    23100.0  ...         NaN
## Desojado          NaN  ...         NaN
## Maribel Buenas Noches 15000.0  ...         NaN
## Cíclope          13400.0  ...         NaN
## El Cuatro Ojos         NaN  ...         NaN
##
## [13 rows x 4 columns]
```

Para poder realizar los análisis y visualizaciones posteriores, te han pedido que elimines los NaNs del dataset. Realiza los siguientes pasos para limpiar tu dataset:

1. Has un conteo de cuántos NaNs hay en cada fila y en cada columna
2. Elimina las filas y columnas donde todos los valores sean NaN.
3. Dado que la columna cantidad\_vendidos no es tan importante, cambia los NaNs que haya en esa columna por 0.
4. Dado que la columna precio es muy importante, elimina las filas restantes que tengan algún NaN en dicha columna.

Realiza todas tus transformaciones usando el DataFrame df\_copy.

```
df_copy = df.copy()

## Realiza aquí tus transformaciones
```

```
##  
## ...  
## ...
```

## RETO 5. LIMPIANDO UN DATASET

### OBJETIVOS

- Aplicar todo lo que aprendimos el día de hoy a un dataset real

### DESARROLLO

#### Limpieza de datos en el mundo real

Hasta ahora hemos estado realizando ejercicios con datasets dummy (falsos). Ahora vamos a aplicar todo lo que hemos aprendido el día de hoy a un dataset real.

El dataset se encuentra en la carpeta Datasets en la raíz del repositorio. El nombre el dataset es 'melbourne\_housing-raw.csv'.

Lee el dataset usando pandas y realiza las siguientes tareas:

1. Ve a este link (<https://www.kaggle.com/anthonypino/melbourne-housing-market>) para conocer más sobre el dataset y los datos que contiene.
2. Explora tu dataset para entender su estructura
3. Identifica los NaNs en el dataset y dónde se encuentran
4. Elimina los NaNs de tu dataset
5. Resetea tu índice para que sea compatible con el nuevo dataset
6. Cambia los nombres de las columnas para que tengan consistencia y no haya errores ortográficos
7. Realiza agregaciones (min, max, mean, etc) de las siguientes filas para conocer mejor la distribución de tus datos: a) Price b) Distance c) Landsize
8. Si tienes dudas en algún momento, por favor pídele a la experta que te oriente. Todas las tareas que hay que realizar ya las hemos hecho en otros retos; puedes ir a revisar esos otros ejercicios para recordar.

¡Mucha suerte!