

ACÀMICA

¡Bienvenidos/as a Data Science!



Agenda

¿Cómo anduvieron?

Repaso: ¿Qué es programar? Puesta en común notebook

Explicación: Numpy

Break

¿Sabías que...?

Hands-on training

Cierre



¿Cómo anduvieron?



ACÁMICA



¿Alguna duda con estos canales?

¿Dónde estamos?



Cronograma

bloque

entrega

tiempo

ADQUISICIÓN Y EXPLORACIÓN

MODELADO

DEPLOY

Exploración de datos

Feature Engineering

Regresión

Optimización de parámetros

Procesam. del lenguaje natural

Sistema de recomendación

Publicación de modelos

SEM 1

SEM 5

SEM 8

SEM 12

SEM 13

SEM 18

SEM 23

SEM 2

SEM 6

SEM 9

SEM 14

SEM 19

SEM 24

SEM 3

SEM 7

SEM 10

SEM 15

SEM 20

SEM 4

SEM 11

SEM 16

SEM 21

SEM 17

SEM 22

APRENDIZAJE SUPERVISADO

APRENDIZAJE NO SUPERVISADO



BLOQUE 1

Exploración de datos

Semana 1

Introducción a Data Science
Python

Semana 2

Numpy
Probabilidad y Estadística – Pandas

Semana 3

Probabilidad y Estadística – Pandas
Matplotlib

Semana 4

Seaborn
Práctica integradora para el proyecto

Feature Engineering

Semana 5

Práctica integradora para el proyecto
Transformación de datos con Pandas

Semana 6

Clases y objetos
Scikit-Learn

Semana 7

¡DEMOS! (presentación de los trabajos de este
BLOQUE 1)



Repaso: Programación en Python



PRIMEROS PASOS CON PYTHON



UN LENGUAJE DE PROGRAMACIÓN VIENE CON...

tipos de datos

Números, texto, variables de verdad (bool), etc.

estructuras de datos

Podemos hacer “conjuntos” de cosas y agruparlas de formas específicas. ¡Y vienen con funcionalidades propias! Ejemplo: listas.

funciones propias

Ejemplo: `print()`, `type()`, etc.

Vimos, además, que podemos definir **Variables**.

PRIMEROS PASOS CON PYTHON



TIPOS DE DATOS

Enteros	Floats	Strings	Booleanos
Son los números que usamos para contar, el 0 y los negativos	Son los números "con coma" Se introducen usando puntos	Texto Se introducen entre comillas dobles, "", o simples, ' '.	Variables de "verdad": verdadero o Falso
-1 0 1 2	5.1 -1.3 1.0 10.0	"Hola Mundo" "A" 'Mi nombre es Esteban'	True False 1 == 2 1 == 1
[1]: <code>type(3)</code> [1]: <code>int</code>	[1]: <code>type(3.0)</code> [1]: <code>float</code>	[1]: <code>type("Hola")</code> [1]: <code>str</code>	[1]: <code>type(2==2)</code> [1]: <code>bool</code>

Operaciones con variables

Distintos **tipos de datos** permiten realizar distintas **tipos de operaciones**.

```
In [11]: b = 'Hola!'
c = ' Como estas?'
print(b + c)

Hola! Como estas?
```

```
In [12]: x = 5
y = 7
print(x+y)

12
```

```
In [13]: variable_1 = True
variable_2 = False
print(variable_1 or variable_2)

True
```

El resultado de estas **operaciones** dependen del **tipo de variable**:

```
In [14]: x = '5'
y = '7'
print(x+y)

57
```

```
In [17]: b = 'Hola!'
c = 8
print(b + c)

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-17-0721abbbb84d> in <module>()
      1 b = 'Hola!'
      2 c = 8
----> 3 print(b + c)

TypeError: must be str, not int
```

Operaciones básicas entre ENTEROS y FLOATS

```
In [42]: x = 3  
        y = 1.5  
        print(x/y)  
  
2.0
```

```
In [43]: x = 2  
        y = 3  
        print(x**y)  
  
8
```

```
In [44]: x = 10  
        y = 3  
        print(x%y)  
  
1
```

Operación	Operador	Ejemplo
Suma	+	3 + 5.5 = 8.5
Resta	-	4 - 1 = 3
Multiplicación	*	3 * 6 = 18
Potencia	**	3 ** 2 = 9
División (cociente)	/	15.0 / 2.0 = 7.5
División (parte entera)	//	15.0 // 2.0 = 7
División (resto)	%	7 % 2 = 1

Operaciones básicas con STRINGS

```
In [32]: txt_1 = 'Los textos'
         txt_2 = ' se concatenan.'
         print(txt_1 + txt_2)
```

Los textos se concatenan.

```
In [33]: b = 'Los textos'
         c = ' no se restan.'
         print(b - c)
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-33-def6c3f4c5e8> in <module>()
      1 b = 'Los textos'
      2 c = ' no se restan.'
```

```
----> 3 print(b - c)
```

TypeError: unsupported operand type(s) for -: 'str' and 'str'

```
In [34]: txt_3 = 'Los textos se multiplican. '
         print(txt_3 * 2)
```

Los textos se multiplican. Los textos se multiplican.

Operaciones lógicas

Un tipo importante de operación en programación son las **operaciones lógicas**. Estas pueden realizarse sobre **variables booleanas**.

```
In [27]: variable_1 = True  
         variable_2 = False  
         print(variable_1 or variable_2)
```

True

```
In [28]: print(not(variable_1))
```

False

El resultado es también una **variable booleana**.

A	B	A & B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

A	A!
False	True
True	False

Listas



Definición

Una estructura de dato muy importante en Python son las **listas**.
Una lista consiste en una serie de elementos ordenados:

```
In [47]: lista_1 = [2, 4.7, True, 'Texto']  
         type(lista_1)
```

```
Out[47]: list
```

```
In [49]: lista_2 = [0, lista_1, 'Mas texto']  
         print(lista_2)  
[0, [2, 4.7, True, 'Texto'], 'Mas texto']
```

→ Los elementos pueden ser de distintos tipos.

→ Incluso puede haber listas dentro de listas.

Las **listas** se definen con corchetes []

Operaciones con LISTAS

Las listas se pueden **sumar** entre sí (se **concatenan**). También se les puede agregar un elemento nuevo mediante el método **'append()'**

```
In [52]: lista_1 = [2, 4.7, True, 'Texto']  
         lista_2 = [42, 42]  
         lista_1 + lista_2
```

```
Out[52]: [2, 4.7, True, 'Texto', 42, 42]
```

```
In [53]: lista_1 = [2, 4.7, True, 'Texto']  
         lista_1.append('Un nuevo elemento')  
         lista_1
```

```
Out[53]: [2, 4.7, True, 'Texto', 'Un nuevo elemento']
```

Operaciones con LISTAS

```
In [55]: lista_1 = [2, 4.7, True, 'Texto']  
len(lista_1)
```

```
Out[55]: 4
```

```
In [56]: lista_2 = [0, lista_1, 'Mas texto']  
len(lista_2)
```

```
Out[56]: 3
```

```
In [59]: lista_vacia = []  
len(lista_vacia)
```

```
Out[59]: 0
```

```
In [60]: lista_vacia.append(42)  
lista_vacia.append('un segundo item')  
print(lista_vacia)
```

```
[42, 'un segundo item']
```

Las listas tienen un largo determinado por su cantidad de elementos. Se consulta mediante la función **len()**.

Se pueden generar listas vacías y luego ir agregándole elementos a medida que una lo precise.

Loops



LOOPS - For

Los **Loops** en programación son bloques de código que, dadas ciertas condiciones, se repiten una cierta cantidad de veces.

El **For** es un tipo de **Loop** que repite un bloque de código tantas veces como elementos haya en una **lista** dada:

```
In [62]: lista_1 = [0, 1, 2, 3]
         for item in lista_1:
             print('Hola.')
```

Hola.
Hola.
Hola.
Hola.

```
In [64]: lista_1 = [10, 20, 30,]
         for item in lista_1:
             doble = 2*item
             print(doble)
```

20
40
60

```
In [66]: lista_nombres = ['Ernesto', 'Camilo', 'Violeta']
         nueva_lista = []

         for item in lista_nombres:
             oracion = 'Mi nombre es ' + item
             nueva_lista.append(oracion)

         print(nueva_lista)
```

['Mi nombre es Ernesto', 'Mi nombre es Camilo', 'Mi nombre es Violeta']

Condicionales



CONDICIONALES - if

Los **condicionales** son bloques de código que se ejecutan únicamente si se cumple una condición. El resultado de esta condición debe ser un **Booleano** (True o False). Esto se logra mediante el condicional **if**.

[10]:

```
valor = 5
if valor > 10:
    print('El valor es mayor que 10')
```

5 > 10

False

No se cumple la condición.

[11]:

```
valor = 15
if valor > 10:
    print('El valor es mayor que 10')
```

15 > 10

True

El valor es mayor que 10

Se cumple la condición.

CONDICIONALES - if / else

Además uno puede agregar un código que se ejecute si la condición no se cumple. Para esto se utiliza el condicional **else**.

```
In [77]: nombre = 'Pedro'

if nombre == 'Juan':
    print('Esta persona se llama Juan')
else:
    print('Esta persona NO se llama Juan')
```

Esta persona NO se llama Juan

```
'Pedro' == 'Pedro'
```

True

```
'Juan' == 'Pedro'
```

False

La comparación entre strings también genera un booleano.

Nota: Para condicionales usamos doble igual ==, ya que nos reservamos el igual simple = para la asignación de variables.

CONDICIONALES - if / elif / else

Además del **if** y el **else**, uno puede agregar más condiciones a través de condicional **elif** (else if). De esta forma se puede agregar un número arbitrario de condiciones.

```
In [80]: edad = 20

if edad < 18:
    print('Esta persona tiene menos de 18 años')
elif edad > 18:
    print('Esta persona tiene mas de 18 años')
else:
    print('Esta persona tiene justo 18 años')
```

Esta persona tiene mas de 18 años

Combinando LOOPS y CONDICIONALES

Los distintos loops y condicionales que vimos se pueden combinar para generar procedimientos más complejos.

```
In [81]: lista_de_edades = [4,20,15,29,11,42,10,18]
        lista_mayores = []

        # Queremos armar una lista solo con las edades mayores o iguales a 18
        for edad in lista_de_edades:
            if edad >= 18:
                # Agremos a la lista de mayores
                lista_mayores.append(edad)

        print(lista_mayores)

[20, 29, 42, 18]
```

¿Y si Python no alcanza?



Numpy

Tenemos la lista con los primeros diez números naturales:

```
[ ]: primeros_10 = [0,1,2,3,4,5,6,7,8,9]
```

Y queremos sumarle un número, por ejemplo “2” a todos los elementos. ¿Qué pasará?

```
[ ]: primeros_10 + 2
```

```
[2]: primeros_10 + 2
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-2-8bd10d42dc1e> in <module>  
----> 1 primeros_10 + 2  
  
TypeError: can only concatenate list (not "int") to list
```

Numpy

¿Cómo se hace si queremos usar Python “puro”?

```
[3]: primeros_10_sumados = [x+2 for x in primeros_10]  
    primeros_10_sumados
```

```
[3]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Esto se llama “list
comprehension”

Numpy

¿Cómo se hace si queremos usar Python “puro”?

```
[3]: primeros_10_sumados = [x+2 for x in primeros_10]  
    primeros_10_sumados
```

Esto se llama “list
comprehension”

```
[3]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Pero es muy incómodo...

Numpy

A veces, las estructuras de datos que vienen con Python - y sus funcionalidades asociadas - no son suficientes. Para eso necesitamos usar **Librerías**.

Nuestra primera librería:



Numpy

- Fundamental para hacer cálculo numérico con Python
- Muy buena [documentación](#)
- Como muchas librerías, trae una estructura de datos propia: los **arrays** o arreglos.

array: a primer orden, es como una **lista**. De hecho, se pueden crear a partir de una lista.

Importamos la librería
(*numpy*) y le ponemos
un nombre (*np*)

```
[1]: import numpy as np
```

```
arreglo = np.array([1,2,3,4,5])  
arreglo
```

Es una lista

```
[1]: array([1, 2, 3, 4, 5])
```

```
[2]: print(arreglo)
```

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


Numpy

Si bien lo creamos a partir de una lista, tiene muchas más funcionalidades:

```
[1]: import numpy as np
```

```
arreglo = np.array([1,2,3,4,5])  
arreglo
```

```
[1]: array([1, 2, 3, 4, 5])
```

```
[2]: print(arreglo)
```

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

```
[3]: arreglo + 2
```

```
[3]: array([3, 4, 5, 6, 7])
```

¡Anduvo!



¿Qué aprendieron en los videos de la plataforma?

Numpy: Instalación

1. Activar el ambiente: *"conda activate datascience"*
2. Instalar NumPy: *"conda install numpy"*

Una vez que esté instalado, abrir con Jupyter Lab un notebook vacío.

Numpy

Formas de crear arreglos de numpy

- Ya vimos a partir de una lista

```
[1]: import numpy as np  
  
arreglo = np.array([1,2,3,4,5])  
arreglo
```

Numpy

Formas de crear arreglos de numpy

- Ya vimos a partir de una lista
- ¿Qué hace np.arange()?

```
[1]: import numpy as np  
  
arreglo = np.array([1,2,3,4,5])  
arreglo
```

Numpy

Formas de crear arreglos de numpy

- Ya vimos a partir de una lista

```
[1]: import numpy as np

arreglo = np.array([1,2,3,4,5])
arreglo
```

- ¿Qué hace np.arange()? Arreglo en un rango de valores, de "a saltos".

```
[4]: arreglo_1 = np.arange(2,9)
arreglo_1
```

```
[4]: array([2, 3, 4, 5, 6, 7, 8])
```

```
[6]: arreglo_2 = np.arange(2,9,2)
arreglo_2
```

```
[6]: array([2, 4, 6, 8])
```

Numpy

Formas de crear arreglos de numpy

- ¿Qué hace `np.linspace()`?

Numpy

Formas de crear arreglos de numpy

- ¿Qué hace `np.linspace()`? Arreglo equiespaciado

```
[10]: arreglo_3 = np.linspace(2,9,3)  
arreglo_3
```

```
[10]: array([2. , 5.5, 9. ])
```

```
[11]: arreglo_4 = np.linspace(2,9,20)  
arreglo_4
```

```
[11]: array([2.          , 2.36842105, 2.73684211, 3.10526316, 3.47368421,  
            3.84210526, 4.21052632, 4.57894737, 4.94736842, 5.31578947,  
            5.68421053, 6.05263158, 6.42105263, 6.78947368, 7.15789474,  
            7.52631579, 7.89473684, 8.26315789, 8.63157895, 9.          ])
```

Y algunas más que veremos más adelante.

Numpy

Seleccionando elementos de un arreglo:

- Si queremos ver una posición arbitraria:

```
[21]: arreglo = np.arange(2,20,4)
      arreglo
```

```
[21]: array([ 2,  6, 10, 14, 18])
```

```
[22]: print(arreglo[0], arreglo[2], arreglo[-1], arreglo[-4])
      2 10 18 6
```

Numpy

Seleccionando elementos de un arreglo:

- Si queremos ver una posición arbitraria:

```
[21]: arreglo = np.arange(2,20,4)  
      arreglo
```

```
[21]: array([ 2,  6, 10, 14, 18])
```

```
[22]: print(arreglo[0], arreglo[2], arreglo[-1], arreglo[-4])  
      2 10 18 6
```

- Y si queremos rangos:

Numpy

Seleccionando elementos de un arreglo:

- Si queremos ver una posición arbitraria:

```
[21]: arreglo = np.arange(2,20,4)
      arreglo
```

```
[21]: array([ 2,  6, 10, 14, 18])
```

```
[22]: print(arreglo[0], arreglo[2], arreglo[-1], arreglo[-4])
```

```
2 10 18 6
```

- Y si queremos rangos:

```
[32]: arreglo = np.arange(0,15)
      arreglo
```

```
[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
[33]: arreglo[2:12:2]
```

```
[33]: array([ 2,  4,  6,  8, 10])
```

comienzo

salto

final

Numpy

Seleccionando también podemos asignar:

```
[34]: arreglo = np.arange(0,15)  
arreglo
```

```
[34]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
[35]: arreglo[2:7] = 25  
arreglo
```

```
[35]: array([ 0,  1, 25, 25, 25, 25, 25,  7,  8,  9, 10, 11, 12, 13, 14])
```

Numpy

Arreglos multidimensionales

“Shape” y “axis” de los arreglos

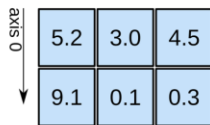
1D array



axis 0

shape: (4,)

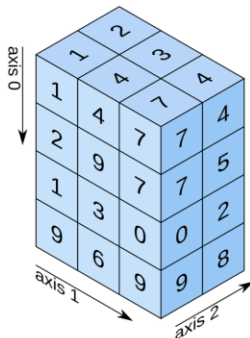
2D array



axis 1

shape: (2, 3)

3D array



axis 1

shape: (4, 3, 2)

No es la forma más cómoda de crearlo

```
[39]: arreglo2d = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])  
arreglo2d
```

```
[39]: array([[ 1,  2,  3,  4],  
           [ 5,  6,  7,  8],  
           [ 9, 10, 11, 12]])
```

```
[40]: arreglo2d.shape
```

```
[40]: (3, 4)
```

filas

columnas

Numpy

Arreglos multidimensionales

```
[42]: arreglo2d = np.arange(100).reshape(10,10)
      arreglo2d
```

```
[42]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
            [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
            [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
            [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
            [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
            [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
            [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
            [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
            [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
            [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
[43]: arreglo2d[2:5,::2]
```

```
[43]: array([[20, 22, 24, 26, 28],
            [30, 32, 34, 36, 38],
            [40, 42, 44, 46, 48]])
```

¿Qué está
haciendo?

Numpy

Filtros Booleanos/Máscaras

```
[66]: arreglo2d = np.arange(30).reshape(6,5)  
arreglo2d
```

```
[66]: array([[ 0,  1,  2,  3,  4],  
            [ 5,  6,  7,  8,  9],  
            [10, 11, 12, 13, 14],  
            [15, 16, 17, 18, 19],  
            [20, 21, 22, 23, 24],  
            [25, 26, 27, 28, 29]])
```

Numpy

Filtros Booleanos/Máscaras

```
[66]: arreglo2d = np.arange(30).reshape(6,5)  
arreglo2d
```

```
[66]: array([[ 0,  1,  2,  3,  4],  
            [ 5,  6,  7,  8,  9],  
            [10, 11, 12, 13, 14],  
            [15, 16, 17, 18, 19],  
            [20, 21, 22, 23, 24],  
            [25, 26, 27, 28, 29]])
```

→
Creamos la
máscara

```
[67]: mask = arreglo2d < 20  
mask
```

```
[67]: array([[ True,  True,  True,  True,  True],  
            [ True,  True,  True,  True,  True],  
            [ True,  True,  True,  True,  True],  
            [ True,  True,  True,  True,  True],  
            [False, False, False, False, False],  
            [False, False, False, False, False]])
```


Numpy

Filtros Booleanos/Máscaras

```
[66]: arreglo2d = np.arange(30).reshape(6,5)  
arreglo2d
```

```
[66]: array([[ 0,  1,  2,  3,  4],  
            [ 5,  6,  7,  8,  9],  
            [10, 11, 12, 13, 14],  
            [15, 16, 17, 18, 19],  
            [20, 21, 22, 23, 24],  
            [25, 26, 27, 28, 29]])
```

→
Creamos la
máscara

```
[67]: mask = arreglo2d < 20  
mask
```

```
[67]: array([[ True,  True,  True,  True,  True],  
            [ True,  True,  True,  True,  True],  
            [ True,  True,  True,  True,  True],  
            [ True,  True,  True,  True,  True],  
            [False, False, False, False, False],  
            [False, False, False, False, False]])
```

```
[68]: arreglo2d[mask]
```

```
[68]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
            17, 18, 19])
```

←
Y seleccionamos aquellos
elementos que cumplen la
condición que representa
la máscara

Numpy

Funciones de Numpy

Hay muchas funciones: vamos a mostrar un ejemplo, ya que la mayoría tiene una sintaxis similar

```
[52]: arreglo2d = np.arange(9).reshape(3,3)
      arreglo2d
```

```
[52]: array([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
[53]: arreglo2d.sum()
```

```
[53]: 36
```

```
[54]: arreglo2d.sum(axis = 0)
```

```
[54]: array([ 9, 12, 15])
```

```
[55]: arreglo2d.sum(axis = 1)
```

```
[55]: array([ 3, 12, 21])
```

¿Qué está haciendo?

Numpy

Funciones de Numpy

Hay muchas funciones: vamos a mostrar un ejemplo, ya que la mayoría tiene una sintaxis similar

```
[52]: arreglo2d = np.arange(9).reshape(3,3)
      arreglo2d
```

```
[52]: array([[0, 1, 2],
            [3, 4, 5],
            [6, 7, 8]])
```

```
[53]: arreglo2d.sum()
```

```
[53]: 36
```

```
[54]: arreglo2d.sum(axis = 0)
```

```
[54]: array([ 9, 12, 15])
```

```
[55]: arreglo2d.sum(axis = 1)
```

```
[55]: array([ 3, 12, 21])
```

Es equivalente a:

```
[56]: np.sum(arreglo2d)
```

```
[56]: 36
```

```
[57]: np.sum(arreglo2d, axis = 0)
```

```
[57]: array([ 9, 12, 15])
```

```
[58]: np.sum(arreglo2d, axis = 1)
```

```
[58]: array([ 3, 12, 21])
```

A close-up photograph of a white ceramic cup filled with a latte. The surface of the milk is decorated with intricate latte art, featuring a central heart shape surrounded by concentric, wavy lines. The cup is placed on a matching white saucer. In the background, a blurred white napkin and a silver fork are visible on a dark, textured surface. The overall lighting is soft and focused on the cup.

¡BREAK!

¿Sabías que...



Hands-on training



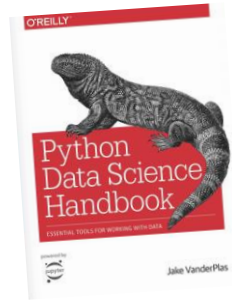


clase_03_numpy.ipynb

Recursos



Numpy



Capítulo 2, “Introduction to Numpy”, de [Python Data Science Handbook](#)



Recomendaciones para programar

- 1) Comentar el código en voz alta ayuda a aprender y a entender lo que estás haciendo.
- 2) No tengas miedo de hacer, romper y arreglar.
- 3) La frustración es una buena señal (“Get things done”).
- 4) Pedir la opinión de tus compañeros/as y mentores/as sobre tu código.
- 5) Busca crecer en comunidad (Medium, Github, Slack Stackoverflow, etc).
- 6) Pide ayuda a tu mejor amigo:

Google Search

I'm Feeling Lucky

Search by voice

Para la próxima

1. Comenzamos con estadística. Recomendamos mirar este recurso:
<https://seeing-theory.brown.edu/basic-probability/index.html>
2. Ver los videos de la plataforma “Biblioteca: Pandas”
3. Completar el notebook de hoy y el de Python si no lo hicieron



ACÀMICA