

Plan de Formación  
Profesional y Continua



**curso gratuito**

[Análisis de datos]

# Trabajo con Datos

# Introducción

¡Te damos la bienvenida al manual de Trabajo con Datos! 🙌.

Los objetivos de aprendizaje estarán orientados a incorporar conocimientos vinculados a las funcionalidades principales de la librería de código abierto.

## Aprendizaje esperado

Al finalizar este módulo serás capaz de:

- ✓ Comprender la matemática para data en python & numpy
- ✓ Utilización de la librería Pandas
- ✓ Trabajo con archivos de texto
- ✓ Procesamiento y limpieza de datos
- ✓ Acceder a datos y crear un dataframe.
- ✓ Conocer funcionalidades básicas como: creación y carga de datos. Creación desde escalar. Posicionamiento.
- ✓ Realizar operaciones esenciales como: Merge, grouping, Reshaping, Plotting.

# NumPy

NumPy, que significa "**Numerical Python**", es una **biblioteca de Python** que proporciona soporte para matrices y operaciones matemáticas de alto rendimiento. Fue creado para facilitar la manipulación eficiente de arreglos multidimensionales y matrices, así como para realizar operaciones matemáticas avanzadas sobre estos datos. NumPy es esencial para el cómputo científico y el análisis de datos en Python debido a su capacidad para realizar cálculos numéricos rápidos y eficientes.

**NumPy se integra estrechamente con Python** y es una de las bibliotecas fundamentales en el ecosistema de ciencia de datos de Python. Proporciona una interfaz para operaciones matriciales y algebraicas, lo que facilita la manipulación y el análisis de datos numéricos. Muchas otras bibliotecas populares para análisis de datos, como Pandas y Matplotlib, se construyen sobre NumPy, aprovechando su funcionalidad para realizar cálculos eficientes.

## Funcionalidades y Aplicaciones:

NumPy es utilizado en una variedad de aplicaciones, incluyendo:

- ↪ Manipulación de Datos: NumPy facilita la manipulación eficiente de datos en forma de matrices, lo que es esencial para tareas como filtrado, selección y transformación de datos.
- ↪ Cálculos Matemáticos: Ofrece una amplia gama de funciones matemáticas para realizar operaciones algebraicas, trigonométricas, estadísticas y otras operaciones numéricas esenciales.
- ↪ Operaciones de Álgebra Lineal: NumPy proporciona funciones para realizar operaciones de álgebra lineal, como la inversión de matrices, descomposiciones y soluciones de sistemas de ecuaciones lineales.
- ↪ Soporte para Números Aleatorios: Incluye funciones para generar números aleatorios, lo que es útil en simulaciones y experimentos estadísticos.

NumPy proporciona las herramientas necesarias para manipular datos de manera eficiente y realizar cálculos numéricos avanzados.

## Importar y usar NumPy

Para importar y usar NumPy en Python, primero debes asegurarte de tener NumPy instalado. Puedes instalar NumPy utilizando pip, que es el gestor de paquetes de Python.

Abre una terminal o línea de comandos y ejecuta el siguiente comando:

```
pip install numpy
```

Una vez que NumPy esté instalado, puedes importarlo en tu script o sesión de Python de la siguiente manera:

```
python  
  
import numpy as np
```

1. **Importar NumPy:** Utiliza la palabra clave import seguida de numpy para importar la librería.

puedes renombrar NumPy utilizando la palabra clave as seguida del nombre que desees utilizar para referirte a NumPy en tu código. La convención común es utilizar np como alias.

2. **Usar NumPy:** Una vez que hayas importado NumPy, puedes utilizar todas las funciones y herramientas que ofrece.

Por ejemplo, puedes crear arrays, matrices y realizar operaciones matemáticas utilizando funciones de NumPy.

Acá tienes un ejemplo sencillo de cómo usar NumPy para crear un array y calcular su suma:

```
python

import numpy as np

# Crear un array utilizando NumPy
mi_array = np.array([1, 2, 3, 4, 5])

# Calcular la suma de los elementos del array
suma = np.sum(mi_array)

print("Array:", mi_array)
print("Suma de los elementos del array:", suma)
```

Este código crea un array utilizando **NumPy**, calcula la suma de sus elementos utilizando la función **np.sum()** y luego imprime el array y el resultado de la suma.

Una vez que hayas **importado NumPy**, puedes acceder a todas las funciones y herramientas que ofrece para realizar operaciones de álgebra lineal, manipulaciones de arrays, estadísticas y mucho más. NumPy es una herramienta poderosa y versátil para el cálculo numérico en Python.

## Escalares, Vectores y Matrices en NumPy

NumPy es una librería de Python que proporciona estructuras de datos eficientes para trabajar con escalares, vectores y matrices.

- ↪ Los escalares en Python se pueden representar simplemente como variables numéricas.
- ↪ Los vectores se pueden representar utilizando arrays de una dimensión en NumPy.
- ↪ Las matrices se pueden representar utilizando arrays bidimensionales en NumPy.

## Operaciones de Álgebra Lineal con NumPy

- NumPy proporciona una amplia gama de funciones para realizar operaciones de álgebra lineal, como multiplicación de matrices, inversión de matrices, resolución de sistemas de ecuaciones lineales, descomposiciones, cálculo de valores y vectores propios, entre otros.

- Estas operaciones se pueden realizar de manera eficiente utilizando las funciones proporcionadas por NumPy, lo que hace que Python sea una opción poderosa para el álgebra lineal y el cálculo numérico.

### Librerías Adicionales para Álgebra Lineal:

- Además de NumPy, Python también tiene otras librerías como SciPy, que extienden las capacidades de NumPy para incluir más funciones y algoritmos avanzados de álgebra lineal y cálculo numérico.
- Estas librerías proporcionan una amplia gama de herramientas y algoritmos para resolver problemas complejos en álgebra lineal y campos relacionados.

### Trabajar con escalares, vectores y matrices utilizando NumPy en Python

#### 1. Escalares en Python:

```
python

# Definir un escalar como una variable numérica
escalar = 5
print("Escalar:", escalar)
```

Operaciones con Escalares:

```
python
```

```
import numpy as np

# Definir un escalar
escalar = 5

# Operaciones básicas con escalares
resultado_suma = escalar + 3
resultado_resta = escalar - 2
resultado_multiplicacion = escalar * 4
resultado_division = escalar / 2

print("Resultado de la suma:", resultado_suma)
print("Resultado de la resta:", resultado_resta)
print("Resultado de la multiplicación:", resultado_multiplicacion)
print("Resultado de la división:", resultado_division)
```

## 2. Vectores con NumPy:

```
python
```

```
import numpy as np

# Definir un vector como un array de una dimensión utilizando NumPy
vector = np.array([1, 2, 3])
print("Vector:", vector)
```

Operaciones con vectores:

```
python
```

```
import numpy as np

# Definir vectores utilizando NumPy
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])

# Suma de vectores
suma_vectores = vector1 + vector2

# Producto punto entre vectores
producto_punto = np.dot(vector1, vector2)

print("Suma de vectores:", suma_vectores)
print("Producto punto entre vectores:", producto_punto)
```

### 3. Matrices con NumPy:

```
python
```

```
import numpy as np

# Definir una matriz como un array bidimensional utilizando NumPy
matriz = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

print("Matriz:")
print(matriz)
```

Operaciones con Matrices:



```
python

import numpy as np

# Definir matrices utilizando NumPy
matriz1 = np.array([[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]])
matriz2 = np.array([[9, 8, 7],
                    [6, 5, 4],
                    [3, 2, 1]])

# Suma de matrices
suma_matrices = matriz1 + matriz2

# Multiplicación de matrices
producto_matrices = np.dot(matriz1, matriz2)

print("Suma de matrices:")
print(suma_matrices)
print("\nProducto de matrices:")
print(producto_matrices)
```

NumPy proporciona una amplia gama de funciones y métodos para realizar operaciones matemáticas y manipulaciones de datos de manera eficiente en Python.

## Ndarray & Documentación

### 1. ndarray en NumPy:

**ndarray** es la estructura de datos fundamental en NumPy. Es un array multidimensional que puede contener elementos del mismo tipo. Los arrays NumPy proporcionan una manera eficiente de almacenar y manipular datos numéricos en Python. Pueden ser de una, dos o más dimensiones, y ofrecen una amplia gama de funciones para realizar operaciones matemáticas y manipulaciones de datos.

## 2. Documentación de NumPy:

Puedes acceder a la documentación oficial de NumPy en su sitio web: [Documentación de NumPy](#). Acá encontrarás tutoriales, guías de referencia y una lista completa de funciones disponibles en NumPy, junto con ejemplos de uso.

### Crear un array NumPy:

```
python

import numpy as np

# Crear un array unidimensional con los números del 1 al 10
arr1d = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print("Array unidimensional:", arr1d)

# Crear una matriz de 3x3 con todos los elementos iguales a 0
matriz_zeros = np.zeros((3, 3))
print("Matriz de ceros:")
print(matriz_zeros)

# Crear una matriz identidad de 4x4
matriz_identidad = np.eye(4)
print("Matriz identidad:")
print(matriz_identidad)
```

## Operaciones con arrays:

```
python

import numpy as np

# Crear dos arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Sumar los elementos de los arrays
suma = arr1 + arr2
print("Suma:", suma)

# Multiplicar los elementos de los arrays
producto = arr1 * arr2
print("Producto:", producto)

# Calcular la media de los elementos de un array
arr3 = np.array([2, 4, 6, 8, 10])
media = np.mean(arr3)
print("Media:", media)
```

Estos son solo algunos ejercicios simples para comenzar a practicar con NumPy. Puedes explorar la documentación de NumPy para encontrar más funciones y ejemplos, y continuar practicando para familiarizarte con las capacidades de NumPy en Python.

# Introducción a Pandas y Dataframes

## ¿Qué es Pandas?

Pandas es una biblioteca de software enfocada en la manipulación y el análisis de datos rápidos y fáciles en Python. En particular, ofrece estructuras de datos de alto nivel (como DataFrame y Series) y métodos de datos para manipular y visualizar tablas numéricas y datos de series temporales. Está construido sobre NumPy y está altamente optimizado para el rendimiento (alrededor de 15 veces más rápido), con rutas de código críticas escritas en Python o C. La estructura de datos ndarray y las capacidades de transmisión de NumPy se usan mucho.

El creador de Pandas, Wes McKinney, comenzó a desarrollar la biblioteca en 2008 durante su mandato en AQR, una empresa de gestión de inversiones cuantitativas. Estaba motivado por un conjunto distinto de requisitos de análisis de datos que no estaban bien abordados por ninguna herramienta única a su disposición en ese momento.

## Características de pandas

La librería de código abierto pandas posee las siguientes particularidades:

- Estructuras de datos con ejes etiquetados que admiten la alineación automática o explícita de datos capaces de manejar datos de series temporales y no temporales.
- Capacidad para agregar y eliminar columnas sobre la marcha.
- Manejo flexible de datos faltantes.
- Combinación similar a SQL y otras operaciones relacionales.
- Herramientas para leer y escribir datos entre estructuras de datos en memoria y diferentes formatos de archivo (csv, xls, HDF5, bases de datos SQL).
- Remodelación y pivoteo de conjuntos de datos.

- Segmentación basada en etiquetas, indexación elegante y creación de subconjuntos de grandes conjuntos de datos.
- Agrupar por motor que permite operaciones de división, aplicación y combinación en conjuntos de datos.
- Funcionalidad de serie temporal: generación de rango de fechas y conversión de frecuencia, estadísticas de ventana móvil, regresiones lineales de ventana móvil, cambio de fecha y retraso.
- Indexación de ejes jerárquicos para trabajar con datos de alta dimensión en una estructura de datos de menor dimensión.

## ¿Qué es un Dataframe?

Los dataframes son estructuras contenedoras de datos, del tipo dataset, a través de los cuales se pueden manipular los datos. En distintos lenguajes los contenedores de datos pueden recibir distintos nombres como RecordSet, DataSet, DataTable, File, etc. En nuestro caso, y teniendo en cuenta que estamos utilizando Python como lenguaje de programación de modelos, utilizaremos el nombre de Dataframe al referirnos a los contenedores de datos debido a que la librería más utilizada en Python, Pandas, denomina dataframe a sus contenedores.

## Instalación de Pandas

Durante este recorrido utilizaremos Jupyter Notebook. El mismo lo hemos instalado en la Unidad “1-Entornos Virtuales”. Entonces, abrimos un notebook y ejecutamos el siguiente código:

```
In [ ]: pip install pandas
```

El sistema procederá a la instalación de pandas si no se encuentra instalado o informará lo siguiente para el caso que el mismo ya se encuentre instalado.

```
In [2]: pip install pandas

Requirement already satisfied: pandas in c:\users\ybarr\anaconda3\lib\site-packages (1.4.2)
Requirement already satisfied: numpy>=1.18.5 in c:\users\ybarr\anaconda3\lib\site-packages (from pandas) (1.21.5)
Requirement already satisfied: python-dateutil>=2.8.1 in c:\users\ybarr\anaconda3\lib\site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\ybarr\anaconda3\lib\site-packages (from pandas) (2021.3)
Requirement already satisfied: six>=1.5 in c:\users\ybarr\anaconda3\lib\site-packages (from python-dateutil>=2.8.1->pandas) (1.16.0)
Note: you may need to restart the kernel to use updated packages.

WARNING: There was an error checking the latest version of pip.
```

# Acceder a datos y crear un dataframe

## Accediendo a Datos



En este módulo, accederemos a datos contenidos en archivos en formato Texto y en formato Excel. A continuación detallamos la sintaxis a utilizar para cada uno de los casos:

### ■ Lectura de un archivo CSV

Leer un archivo separado por comas es tan simple como llamar a la función `read_csv`. De forma predeterminada, la función `read_csv` espera que el separador de columnas sea una coma, pero puede cambiarlo con el parámetro `sep`.

#### Sintaxis:

```
pd.read_csv(filepath, sep=, header=, names=, skiprows=,
na_values= ... )
```

 **Archivo de ayuda:** para obtener una explicación detallada de todos los parámetros, ejecuta: 

```
pd.read_csv?
```

Entonces para leer un CSV y mostrar su contenido por pantalla, ejecutamos el siguiente código:

```
import pandas as pd

df=pd.read_csv("C:/Users/ybarr/Desktop/salud/vira.csv",en
coding = "utf-8")

print(df)
```

Ahora bien, analicemos el código utilizado.

En la primera línea, importamos la librería pandas anteriormente instalada y la instanciamos en pd, para luego utilizar sus métodos.

En la línea 2, utilizamos el método read\_csv, utilizando la ruta como parámetro y un segundo modificador para establecer la codificación de los caracteres, como "utf-8". Asignamos el resultado de esa lectura, a una variable que por defecto será un dataframe.

Finalmente, en la tercera línea, solicitamos a Python, la impresión por pantalla de los datos existentes en el dataframe.

Entonces, al ejecutar el código, el notebook nos dará el siguiente resultado:

```
In [15]: import pandas as pd
df=pd.read_csv("C:/Users/ybarr/Desktop/salud/vira.csv",encoding = "utf-8")
print(df)
```

	departamento_id	departamento_nombre	provincia_id	provincia_nombre
0	760	SAN MIGUEL	6	Buenos Aires
1	760	SAN MIGUEL	6	Buenos Aires
2	14	CAPITAL	14	Córdoba
3	427	LA MATANZA	6	Buenos Aires
4	427	LA MATANZA	6	Buenos Aires
...	...	...	...	...
3458	274	FLORENCIO VARELA	6	Buenos Aires
3459	0	CIUDAD DE BUENOS AIRES	2	CABA
3460	0	CIUDAD DE BUENOS AIRES	2	CABA
3461	0	CIUDAD DE BUENOS AIRES	2	CABA
3462	0	CIUDAD DE BUENOS AIRES	2	CABA

	año	semanas_epidemiologicas	evento_nombre
0	2018	18	Bronquiolitis en menores de 2 años
1	2018	18	Bronquiolitis en menores de 2 años
2	2018	21	Enfermedad tipo influenza (ETI)
3	2018	19	Neumonía
4	2018	19	Bronquiolitis en menores de 2 años
...	...	...	...

## ■ Leer un archivo Excel

Pandas le permite leer y escribir en archivos de Excel, por lo que puede leer fácilmente desde Excel, escribir su código en Python y luego volver a escribir en Excel, sin necesidad de VBA. La lectura de archivos de Excel requiere la biblioteca xlrd. El mismo podemos instalarlo a través de pip, ejecutando el siguiente comando:

```
pip install xlrd
```

### La sintaxis de read\_excel es:

```
pd.read_excel('mi-excel.xlsx', 'hoja1')
```

Entonces, para leer un archivo Excel, ejecutamos el siguiente código:

```
import pandas as pd

df=pd.read_excel("C:/Users/ybarr/Desktop/salud/vdz.xls")

print(df)
```

Ahora bien, analicemos el código utilizado.

- En la primera línea, importamos la librería pandas anteriormente instalada y la instanciamos en pd, para luego utilizar sus métodos.
- En la línea 2, utilizamos el método read\_excel, utilizando solamente la ruta como parámetro. Asignamos el resultado de esa lectura, a una variable “df” que por defecto será un dataframe.
- Finalmente, en la tercera línea, solicitamos a Python, la impresión por pantalla de los datos existentes en el dataframe.

Entonces, al ejecutar el código, el notebook nos dará el siguiente resultado:

```
In [2]: import pandas as pd
df=pd.read_excel("C:/Users/ybarr/Desktop/salud/vdz.xls")
print(df)
```

	departamento_id	departamento_nombre	provincia_id	provincia_nombre	año
0	0	"sin dato"	6	Buenos Aires	2018
1	35	Avellaneda	6	Buenos Aires	2018
2	35	Avellaneda	6	Buenos Aires	2018
3	35	Avellaneda	6	Buenos Aires	2018
4	35	Avellaneda	6	Buenos Aires	2018
...	...	...	...	...	...
655	126	Orán	66	Salta	2018
656	126	Orán	66	Salta	2018
657	126	Orán	66	Salta	2018
658	126	Orán	66	Salta	2018
659	126	Orán	66	Salta	2018

	semanas_epidemiologicas	evento_nombre	grupo_edad_id
0	11	Dengue	10
1	7	Dengue	8
2	7	Dengue	10
3	9	Dengue	6
4	9	Dengue	8
...	...	...	...

## ■ Crear pdf desde un conjunto de Datos

Pandas le permite crear un dataframe desde una lista de valores. El mismo lo creamos con el metodo dataframe, enviando como parámetros, el conjunto de claves y sus respectivos valores para esas claves. Veamos un ejemplo:

👉 Creamos un nuevo dataframe con el siguiente código:



```
import pandas as pd

df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'], 'value': [1, 2, 3, 5]})

print(df1)
```

Ejecutamos en notebook y vemos el siguiente resultado:

```
In [2]: import pandas as pd
df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'], 'value': [1, 2, 3, 5]})
print(df1)
```

	lkey	value
0	foo	1
1	bar	2
2	baz	3
3	foo	5

## ■ Creación a partir de un escalar

Si los datos se reducen a un escalar (no a una lista con un único elemento, sino a un sencillo escalar como 7 o 15.4) será necesario añadir el índice explícitamente. El número de elementos de la serie coincidirá con el número de elementos del índice, y el escalar será asignado como valor a todos ellos. La sintaxis es la siguiente:

```
Df1 = pd.series (7, index = ["Enero","Febrero","Marzo"])
```

```
In [15]: df1 = pd.Series (7, index = ["Enero","Febrero","Marzo"])
print (df1)
```

	Enero	Febrero	Marzo
	7	7	7

dtype: int64

## ■ Creación desde un array

Suponiendo que contamos con el siguiente array:

```
mi_array = {'col1': 1.0, 'col2':2.0, 'col3': 3.0}
```

La **sintaxis** para crear un dataframe a partir del mismo e imprimir sus datos, es la siguiente:

```
mi_array = {'col1': 1.0, 'col2':2.0, 'col3': 3.0}

df = pd.DataFrame([mi_array])

print (df)
```

Veamos el resultado de su ejecución:

```
In [16]: mi_array = {'col1': 1.0, 'col2':2.0, 'col3': 3.0}
df = pd.DataFrame([mi_array])
print (df)

   col1  col2  col3
0    1.0    2.0    3.0
```

### ■ Acciones simples sobre un Dataframe

Siendo el dataframe la variable df1, a continuación profundizamos sobre algunas operaciones básicas:

Mostrar las primeras 10 filas del CSV:

```
print(df1.head(10))
```

Contar la cantidad de filas:

```
print(len(df1))
```

Contar la cantidad de filas 2:

```
print(«Cant. filas: %i» % len(df1))
```

→ #El %i indica que ahí va una variable de tipo integer.

→ #Al cerrar comillas dobles, el % indica qué es lo que debe reemplazar a %i

### Splicing de la data:

- ↪ print(df1[:10]) #muestra las primeras 10 filas
- ↪ print(df1[5:]) #muestra todo salvo las primeras 5 filas
- ↪ print(df1[-3:]) #muestra las últimas 3 filas
- ↪ print(df1[:-2]) #muestra todo salvo las últimas 2 filas

↪ `print(df1[-5:-2])` #muestra desde la 5ta desde el final hasta 2 desde el final

**Pasar el campo “keyword” a minúscula:**

```
Df1 = df1['Keyword'].str.lower()
```

**Cómo ver las columnas de un DataFrame en Pandas:**

```
print(df1.columns)
```

**Cómo ver el contenido de una columna en Pandas:**

```
print(df1['CTR']) #la columna se llama CTR
```

**Cómo sacar el promedio de los valores de una columna en Pandas:**

```
print(df1['CTR'].mean())
```

**Posicionamiento:**

Supongamos que deseamos mostrar el primer row completo, mostrando todas las columnas, del dataframe llamado df1. La sintaxis es la siguiente:

```
Df1.loc[0,:]
```

Usaremos la siguiente sintaxis para mostrar lo mismo que el caso anterior pero para solamente una columna:

```
Df1.loc[0,: 'nombre_columna']
```

# Operaciones esenciales

## Operaciones esenciales:

Una vez generado un dataframe con datos desde un archivo csv o Excel, podremos trabajar sobre el conjunto de datos. A continuación, nos enfocaremos en las siguientes operaciones: Merge, grouping, Reshaping, Plotting.

### Merge

En la operación de Merge necesitamos contar con dos dataframes. En este caso vamos a trabajar con 2 df generados a través de una lista de valores que generamos para entender su funcionamiento. Entonces generamos los mismos con el siguiente código:

```
import pandas as pd

df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
                    'value': [1, 2, 3, 5]})

df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
                    'value': [5, 6, 7, 8]})
```

Ahora bien, primero debemos entender las opciones de **Merge** que brinda pandas.

La **sintaxis de Merge** es la siguiente:

```
DataFrame.merge (right, how='inner', on=None,
                 left_on=None, right_on=None, left_index=False,
                 right_index=False, sort=False, suffixes=('_x', '_y'),
                 copy=True, indicator=False, validate=None)
```

En este sentido, solo repasaremos los parámetros más importantes del método. Nos enfocaremos puntualmente en el parámetro **“how”**. El mismo refiere a **cómo hacer el merge**. Sus posibles valores son:

- ➡ **left:** Usa solamente las claves del df izquierdo. Funciona de forma similar al SQL Left join. Se preserva el orden de las claves del df izquierdo. El resultado de la ejecución de Merge es el siguiente:

```
In [9]: df1.merge(df2, how='left', left_on='lkey', right_on='rkey', suffixes=('_left', '_right'))
```

Out[9]:

	lkey	value_left	rkey	value_right
0	foo	1	foo	5
1	foo	1	foo	8
2	bar	2	bar	6
3	baz	3	baz	7
4	foo	5	foo	5
5	foo	5	foo	8

- ➡ **right:** Usa solamente las claves del df derecho. Funciona de forma similar al SQL Right join. Se preserva el orden de las claves del dataframe derecho. El resultado de la ejecución de Merge es el siguiente:

```
In [10]: df1.merge(df2, how='right', left_on='lkey', right_on='rkey', suffixes=('_left', '_right'))
```

Out[10]:

	lkey	value_left	rkey	value_right
0	foo	1	foo	5
1	foo	5	foo	5
2	bar	2	bar	6
3	baz	3	baz	7
4	foo	1	foo	8
5	foo	5	foo	8

- ➡ **outer:** Usa la union de las claves de ambos df. Funciona de forma similar a un Full Outer Join. El resultado de la ejecución de Merge es el siguiente:

```
In [11]: df1.merge(df2, how='outer', left_on='lkey', right_on='rkey', suffixes=('_left', '_right'))
```

Out[11]:

	lkey	value_left	rkey	value_right
0	foo	1	foo	5
1	foo	1	foo	8
2	foo	5	foo	5
3	foo	5	foo	8
4	bar	2	bar	6
5	baz	3	baz	7

- ➡ **inner:** Usa la intersección de las claves de ambos df. Funciona de forma similar al SQL inner join. Se preserva el orden de las claves del df izquierdo. El resultado de la ejecución de Merge es el siguiente:

```
In [12]: df1.merge(df2, how='inner', left_on='lkey', right_on='rkey', suffixes=('_left', '_right'))
```

```
Out[12]:
```

	lkey	value_left	rkey	value_right
0	foo	1	foo	5
1	foo	1	foo	8
2	foo	5	foo	5
3	foo	5	foo	8
4	bar	2	bar	6
5	baz	3	baz	7

➡ **cross:** Funciona de forma similar a un producto cartesiano. Se preserva el orden de las claves del df izquierdo. El resultado de la ejecución de Merge es el siguiente:

```
In [15]: df1.merge(df2, how='cross', suffixes=('_left', '_right'))
```

```
Out[15]:
```

	lkey	value_left	rkey	value_right
0	foo	1	foo	5
1	foo	1	bar	6
2	foo	1	baz	7
3	foo	1	foo	8
4	bar	2	foo	5
5	bar	2	bar	6
6	bar	2	baz	7
7	bar	2	foo	8
8	baz	3	foo	5
9	baz	3	bar	6
10	baz	3	baz	7
11	baz	3	foo	8
12	foo	5	foo	5
13	foo	5	bar	6
14	foo	5	baz	7
15	foo	5	foo	8

📌 **Nota:** Los modificadores `left_on='lkey', right_on='rkey', suffixes=('_left', '_right')` nos permiten:

- ✓ Colocar un el nombre “lkey” a la columna resultante de la clave del df izquierdo
- ✓ Colocar un el nombre “rkey” a la columna resultante de la clave del df derecho
- ✓ Agregar el prefijo “\_left” y “\_right” a las columnas de valores resultantes

## Grouping

La **sintaxis de Group By Pandas** para un dataframe es la siguiente:

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

Una operación de agrupamiento, con Group By, implica combinar los valores iguales de uno o más campos. Esta operación se suele utilizar para agrupar grandes cantidades de datos y calcular operaciones agregadas en estos grupos, como por ejemplo, contar, sumar o realizar un promedio. Veamos un ejemplo.

Primero cargaremos en un dataframe un conjunto de datos que podamos agrupar para sumar los valores de ese campo en el dataframe. Entonces, ejecutamos el siguiente código para cargar un archivo CSV con un conjunto de valores:

```
In [36]: import pandas as pd
df=pd.read_csv("C:/Users/ybarr/Desktop/salud/vira.csv",encoding = "utf-8")
df2=df.drop(columns=["departamento_id", "departamento_nombre", "provincia_id", "año", "semanas_epidemiologicas", "evento_nombre",
"grupo_edad_id", "grupo_edad_desc"])
print(df2)
```

	provincia_nombre	cantidad_casos
0	Buenos Aires	2
1	Buenos Aires	3
2	Córdoba	1
3	Buenos Aires	2
4	Buenos Aires	2
...	...	...
3458	Buenos Aires	1
3459	CABA	1
3460	CABA	2
3461	CABA	1
3462	CABA	1

[3463 rows x 2 columns]

El dataframe contiene la cantidad de casos de una enfermedad por localidad en el tiempo.

En la línea 3 de nuestro código, eliminamos todas las columnas (con `df.drop(columna="X")`) que no queremos en nuestro dataframe y asignamos el dataframe resultante, a un nuevo dataframe, `df2`.

Luego, ejecutaremos el group by sobre el dataframe 2, para sumar la cantidad de casos existentes en el mismo por cada provincia. Para ello ejecutamos lo siguiente:

```
In [35]: df2.groupby(['provincia_nombre'])['cantidad_casos'].sum()

Out[35]: provincia_nombre
Buenos Aires      1525
CABA              1140
Catamarca         211
Chaco             728
Chubut            136
Corrientes        212
Córdoba           454
Entre Ríos        444
Formosa           102
Jujuy             476
La Pampa          49
La Rioja          297
Mendoza           357
Misiones           64
Neuquén           184
Río Negro         202
Salta             587
San Juan          325
San Luis          217
Santa Cruz        135
Santa Fe          41
Santiago del Estero 369
Tierra del Fuego  150
Tucumán           638
Name: cantidad_casos, dtype: int64
```

Vemos como resultado la suma de la cantidad de casos existentes, agrupados por provincia

Cabe resaltar que el agrupamiento podemos hacerlo por uno o más campos del dataframe. Su funcionamiento es similar al group by de SQL.

Por otro lado, las funciones que pueden aplicarse luego del agrupamiento pueden ser, count(), sum(), min(), max(), etc. Más funciones y detalles en el siguiente link (en la entrada “Aggregation”):

[https://pandas.pydata.org/pandas-docs/stable/user\\_guide/groupby.html#dataframe-column-selection-in-groupby](https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html#dataframe-column-selection-in-groupby)

## Reshaping

Utilizaremos esta funcionalidad para transponer un dataframe. Para este caso nos enfocaremos solamente en el método melt(), sabiendo que existen además otros métodos para poder realizar acciones similares. Entenderemos más claramente su funcionamiento, con el siguiente ejemplo. Supongamos que tenemos el siguiente dataframe df, creado con la siguiente sintaxis:

```
df = pd.DataFrame({'equipo': ['A', 'B', 'C', 'D'],
                    'goles': [100, 200, 300, 400],
                    'faltas': [1000, 190, 999, 888],
                    'amonestaciones': [232, 218, 310, 311]})
```



Al mostrar los datos por pantalla, vemos lo siguiente:

```
In [6]: df = pd.DataFrame({'equipo': ['A', 'B', 'C', 'D'],
                           'goles': [100, 200, 300, 400],
                           'faltas': [1000, 190, 999, 888],
                           'amonestaciones': [232, 218, 310, 311]})
df
```

```
Out[6]:
```

	equipo	goles	faltas	amonestaciones
0	A	100	1000	232
1	B	200	190	218
2	C	300	999	310
3	D	400	888	311

Aplicaremos el método `melt()` para hacer un reshape del dataframe `df1` y trasponerlo, con la siguiente sintaxis:

```
df = pd.melt(df, id_vars='equipo', value_vars=['goles',
'faltas', 'amonestaciones'])
```

Veamos el resultado en pantalla luego de ejecutar `melt()` sobre el dataframe `df1`:

```
In [7]: df = pd.melt(df, id_vars='equipo', value_vars=['goles', 'faltas', 'amonestaciones'])
df
```

```
Out[7]:
```

	equipo	variable	value
0	A	goles	100
1	B	goles	200
2	C	goles	300
3	D	goles	400
4	A	faltas	1000
5	B	faltas	190
6	C	faltas	999
7	D	faltas	888
8	A	amonestaciones	232
9	B	amonestaciones	218
10	C	amonestaciones	310
11	D	amonestaciones	311

De esta forma, vemos claramente como hemos modificado el dataframe `df1`.

## Plotting

El método `plot()`, lo usaremos para realizar algunos tipos de gráficos en pantalla en base a los datos contenidos en nuestro dataframe. Para trabajar con estos gráficos utilizaremos la librería `matplotlib`. La página oficial describe a la misma como:

“una biblioteca completa para crear visualizaciones estáticas, animadas e interactivas en Python.”

Para instalar la misma, la sintaxis es la siguiente:

```
pip install -U matplotlib
```

Una vez instalada la librería, avanzamos con plot(). La sintaxis de plot() es la siguiente:

```
dataFrame.plot (*args, **kwargs)
```

En las siguientes líneas mostraremos algunos ejemplos básicos para graficar dataframes.

- **Diagrama de Dispersión (Scatter Plot)**

Los diagramas de dispersión se utilizan para representar una relación entre dos variables. Este es el código para realizar un diagrama de dispersión usando Pandas.

```
import pandas as pd
import matplotlib.pyplot as plt

data = {'tasa_desempleo':
[6.1,5.8,5.7,5.7,5.8,5.6,5.5,5.3,5.2,5.2],
'indice_precio_stock':
[1500,1520,1525,1523,1515,1540,1545,1560,1555,1565]
      }

df =
pd.DataFrame(data,columns=['tasa_desempleo','indice_precio_stock'])
df.plot(x='tasa_desempleo', y='indice_precio_stock', kind =
'scatter')
plt.show()
```

Analicemos el código. 📌

- En las primeras líneas importamos pandas y matplotlib para graficar.
- Luego definimos dos dataframes.

→ Posteriormente, en el método plot(), definimos que nuestro gráfico será del tipo “scatter” y enviamos los valores para los ejes X e Y. Vemos el resultado en Notebook:



### ● Diagrama de líneas

Los gráficos de líneas se utilizan a menudo para mostrar tendencias a lo largo del tiempo. Este es el código para realizar un diagrama de líneas usando Pandas.

```

import pandas as pd
import matplotlib.pyplot as plt

data = {'anio':
[1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010],
        'tasa_desempleo':
[9.8, 12, 8, 7.2, 6.9, 7, 6.5, 6.2, 5.5, 6.3]}

df =
pd.DataFrame(data, columns=['anio', 'tasa_desempleo'])
df.plot(x='anio', y='tasa_desempleo', kind='line')
plt.show()

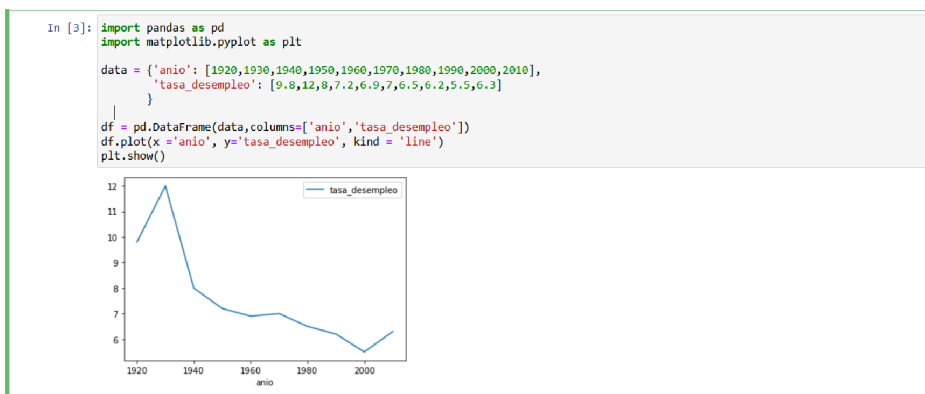
```

Analicemos un poco el código anterior. 🖱️

→ En las primeras líneas importamos pandas y matplotlib para graficar.

→ Luego definimos un data con dos columnas, tasa de desempleo y año.

→ Posteriormente, en el método plot(), definimos que nuestro gráfico será del tipo “línea” y enviamos los valores para los ejes X e Y con el fin de graficar la evolución de la tasa de desempleo por año. Vemos el resultado en Notebook:



### ● Diagrama de líneas

Los gráficos de barras se utilizan para mostrar datos categóricos. Veamos ahora cómo trazar un gráfico de barras usando Pandas. Este es el código para realizar un diagrama de barras usando Pandas.

```
import pandas as pd
import matplotlib.pyplot as plt

data = {'pais':
        ['USA', 'Canada', 'Germany', 'UK', 'France'],

        'PBI_per_capita':
        [45000, 42000, 52000, 49000, 47000]}

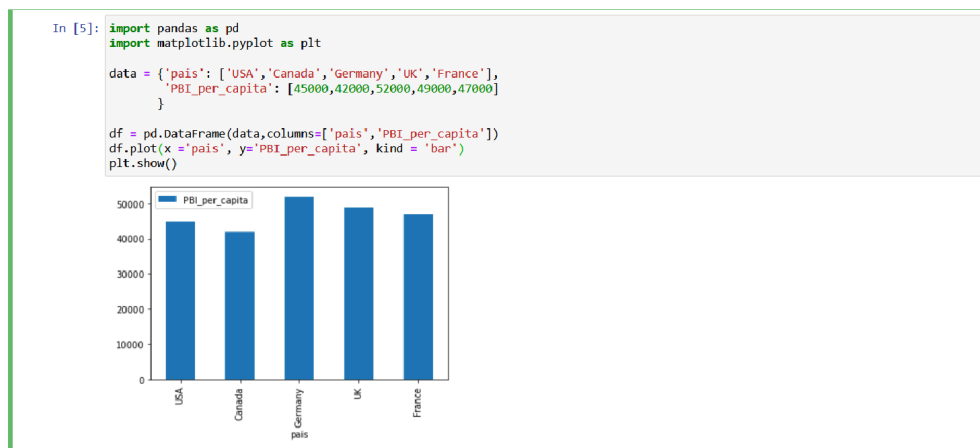
df =
pd.DataFrame(data, columns=['pais', 'PBI_per_capita'])
df.plot(x='pais', y='PBI_per_capita', kind='bar')

plt.show()
```

Analicemos un poco el código anterior.

→ En las primeras líneas importamos pandas y matplotlib para graficar.

- Luego definimos un data con dos columnas, País y PBI per cápita.
- Posteriormente, en el método plot(), definimos que nuestro gráfico será del tipo “Barras” y enviamos los valores para los ejes X e Y con el fin de graficar la evolución el PBI per cápita de acuerdo a cada país.
- Vemos el resultado en Notebook:



De esta forma cerramos el tema 3, Operaciones esenciales sobre dataframes.

# Trabajo con archivos, Read vs Parse.

Cuando se trabaja con archivos en Python, es importante entender la diferencia entre **"leer" (read)** y **"analizar" (parse)** el contenido del archivo. Aquí está la diferencia entre ambos conceptos:

## 1. Leer (Read) un archivo:

Leer un archivo significa simplemente cargar su contenido en la memoria del programa como una cadena de caracteres o bytes.

El contenido del archivo se guarda como una sola entidad y no se realiza ningún análisis o procesamiento adicional.

Esto es útil cuando solo necesitas acceder al contenido bruto del archivo y no necesitas procesar su estructura o realizar manipulaciones más avanzadas.

Ejemplo de lectura de un archivo de texto en Python:

```
python

with open('archivo.txt', 'r') as file:
    contenido = file.read()
```

Analizar un archivo implica interpretar y descomponer su contenido en estructuras de datos más útiles para realizar operaciones específicas.

Por lo general, implica dividir el contenido del archivo en partes significativas, como líneas o campos separados por algún delimitador (como comas en un archivo CSV).

El análisis permite manipular y trabajar con los datos de manera más eficiente y estructurada.

Ejemplo de análisis de un archivo CSV en Python:

python

```
import csv
with open('archivo.csv', 'r') as file:
    lector_csv = csv.reader(file)
    for fila in lector_csv:
        # Procesar cada fila del archivo CSV
        print(fila)
```

La lectura de archivos simplemente carga el contenido del archivo en la memoria, mientras que el análisis implica interpretar y procesar ese contenido para extraer información útil. Dependiendo de tus necesidades y del formato del archivo, puedes optar por leer el archivo en su totalidad o por analizar su contenido para trabajar con él de manera más eficiente.

### Leer Archivo:

Descarga el archivo de texto proporcionado (por ejemplo, un archivo .txt).

Utiliza Python para abrir y leer el contenido del archivo.

### Analizar Archivo:

Lee el archivo nuevamente, pero esta vez analiza su contenido línea por línea.

Para cada línea del archivo, realiza alguna operación, como contar palabras, buscar una cadena específica, o calcular estadísticas simples.

### Comparación:

Compara los resultados obtenidos al simplemente leer el archivo con los resultados obtenidos al analizar el archivo línea por línea.

Reflexiona sobre cuándo sería más adecuado simplemente leer el archivo y cuándo sería necesario analizar su contenido.

## Conexión a bases de datos con Python

Los sistemas de administración de bases de datos relacionales (RDBMS - Relational Database Management Systems) proporcionan una manera fácil de almacenar grandes cantidades de información. Además, con la ayuda de SQL podemos acceder, mantener y modificar los datos almacenados en ellos.

Si bien SQL nos permite realizar uniones o utilizar funciones de agregación, no proporciona los medios para realizar técnicas más avanzadas, como realizar pruebas estadísticas o crear modelos predictivos. Si deseamos realizar este tipo de operaciones, necesitaremos la ayuda de Python.

Esto plantea la pregunta: ¿cómo implementamos el código Python en bases de datos que responden a consultas SQL?

### Archivos de Texto (txt):

Para leer archivos de texto en Python, puedes usar la función `open()` para abrir el archivo y luego leer su contenido.

```
python

with open('archivo.txt', 'r') as file:
    contenido = file.read()
```

### Archivos CSV:

Los archivos CSV (valores separados por comas) son comunes en el análisis de datos y se pueden leer fácilmente en Python utilizando la librería `csv`.

```
python

import csv
with open('archivo.csv', 'r') as file:
    lector_csv = csv.reader(file)
    for fila in lector_csv:
        print(fila)
```



### Uso de CSV con Pandas:

Pandas proporciona una forma más conveniente de trabajar con datos CSV utilizando la función `read_csv()`.

```
python

import pandas as pd
datos = pd.read_csv('archivo.csv')
```

### Archivos Excel:

Para trabajar con archivos Excel en Python, puedes usar la librería pandas, que ofrece funciones como `read_excel()` para importar datos de hojas de cálculo de Excel.

```
python

import pandas as pd
datos_excel = pd.read_excel('archivo.xlsx', sheet_name='Hoja1')
```

### Importar Datos con NumPy:

NumPy también puede leer archivos de texto, aunque generalmente se utiliza para datos numéricos.

```
python

import numpy as np
datos_numpy = np.loadtxt('archivo.txt')
```

### Importar JSONs:

Para importar archivos JSON en Python, puedes usar la librería integrada `json`.

python

```
import json
with open('archivo.json', 'r') as file:
    datos_json = json.load(file)
```

Python ofrece diversas opciones para importar datos desde diferentes tipos de archivos. La elección de la librería adecuada dependerá del formato de los datos y de las operaciones que desees realizar con ellos. Para datos tabulares, Pandas suele ser la opción más conveniente, mientras que NumPy es útil para datos numéricos. La librería csv es útil para archivos CSV simples, y json es ideal para archivos JSON.

## Adaptador de base de datos

Para acceder a las bases de datos con Python, deberemos utilizar un adaptador de base de datos .

Python ofrece adaptadores de base de datos a través de sus módulos que permiten el acceso a las principales bases de datos como MySQL, PostgreSQL, SQL Server y SQLite.



Todos estos módulos se basan en la [PEP 249](#), especificación de API de base de datos de Python (DB-API) para administrar bases de datos. Como resultado, el código utilizado para administrar las bases de datos es coherente en todos los adaptadores de bases de datos.

Aprovechar los adaptadores de bases de datos requiere una comprensión de los objetos y métodos clave que se utilizarán para facilitar la interacción con la base de datos en cuestión.

## Maapeo relacional de objetos

El mapeo objeto relacional de objetos (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas ORM) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.

En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional que posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

## SQLAlchemy

SQLAlchemy es un ORM (Mapeador de relaciones de objetos), que está escrito en Python para trabajar con bases de datos. Ayuda a los programadores y desarrolladores de aplicaciones a tener una flexibilidad de control total sobre las herramientas SQL.

A menudo, al desarrollar aplicaciones en cualquier lenguaje de programación, nos encontramos con la necesidad de almacenar y leer datos de las bases de datos. Este módulo proporciona una forma Pythonic de crear y representar bases de datos relacionales desde dentro de los proyectos de Python. Una ventaja de trabajar con un módulo de este tipo es que no necesitamos recordar las diferencias sintácticas de las distintas bases de datos. El módulo hace todo el trabajo pesado por nosotros, mientras interactúa con todas las bases de datos de la misma manera.

## Conexión a sistemas de gestión de bases de datos relacionales

A continuación realizaremos una breve descripción de los sistemas de gestión de bases de datos relaciones que luego vamos a conectar utilizando Python.

### SQLite3

SQLite es una biblioteca C que proporciona una base de datos liviana basada en disco que no requiere un proceso de servidor separado y permite acceder a la base de datos utilizando una variante no estándar del lenguaje de consulta SQL.

Algunas aplicaciones pueden usar SQLite para el almacenamiento interno de datos. También es posible crear un prototipo de una aplicación usando SQLite y luego transferir el código a una base de datos más grande como PostgreSQL u Oracle.

SQLite viene integrado en la instalación de Python, por lo que no es necesario instalar ningún software adicional.

En el siguiente [link](#) vamos a encontrar un notebook con los pasos que debemos seguir para conectarnos a SQLite utilizando Python. Asimismo, utilizando esa conexión ejecutaremos algunas sentencias DDL y DML de SQL. Finalmente utilizaremos SQLAlchemy para crear un engine y trabajar con la base de datos utilizando Pandas

## SQLite3 + SQL Alchemy + Pandas

```
In [1]: # Import main Libraries
import sqlite3
import pandas as pd
import sqlalchemy as db
```

### Pasos generales para trabajar con SQLite 3

- 1. Crear una conexión a la base de datos:** utilizando el método `connect()` generamos un objeto de conexión a base de datos SQLite. Si es la primera vez que nos conectamos, se nos creará la base de datos en el mismo directorio donde se este ejecutando el notebook o archivo .py.
- 2. Crear un cursor:** El método `cursor()` se utiliza para realizar la conexión y ejecutar consultas SQL que nos permiten crear tablas, insertar datos, etc. Para crear un cursor solo necesitamos usar la conexión que ya hemos creado.
- 3. Ejecutar una sentencia SQL:** Una vez creado el cursor, podremos ejecutar las sentencias SQL utilizando el método `execute()`.
- 4. Realizar un commit:** El método `commit()` de SQLite se utiliza para guardar cualquier transacción de forma permanente en el sistema de base de datos. Todas las modificaciones de datos o del sistema realizadas por el comando COMMIT desde el comienzo de las transacciones son de naturaleza permanente y no se pueden deshacer ni revertir, ya que una operación COMMIT exitosa libera todos los recursos de transacción involucrados.
- 5. Desconectarnos de la base de datos:** El método `close()` de SQLite cierra la conexión a la base de datos.

### Configuraciones principales

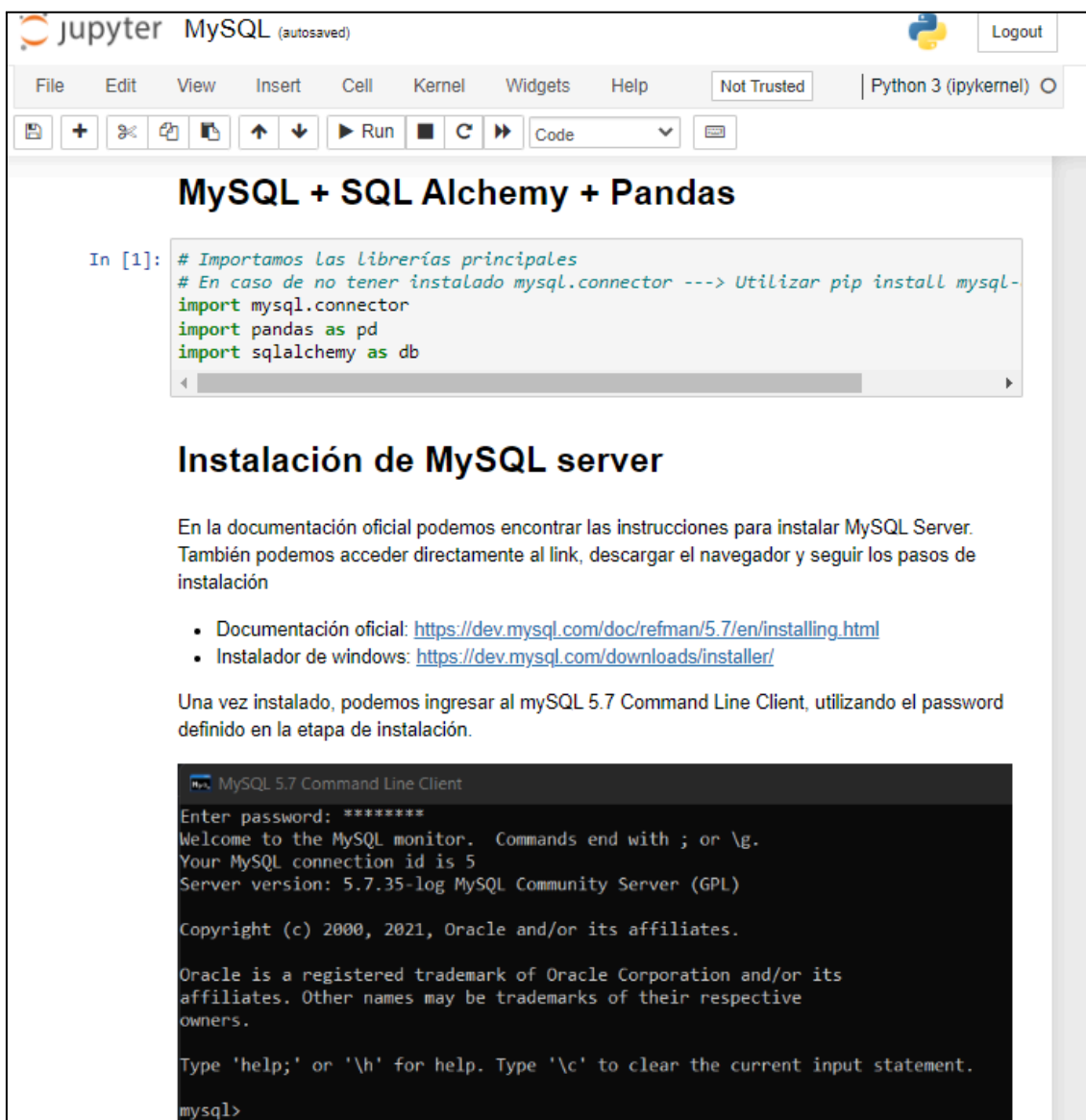
```
In [2]: # Crear una conexión a la base de datos
conn = sqlite3.connect('commerce.db')

# Crear un cursor
cursor = conn.cursor()
```

## MySQL

MySQL es uno de los sistemas de gestión de bases de datos relacionales (RDBMS) más populares del mercado actual. Ocupó el segundo lugar después de Oracle DBMS en el [DB-Engines Ranking](#) del año 2022 . Como la mayoría de las aplicaciones de software necesitan interactuar con los datos de alguna forma, los lenguajes de programación como Python proporcionan herramientas para almacenar y acceder a estas fuentes de datos.

En el siguiente [link](#) vamos a encontrar un notebook con los pasos que debemos seguir para conectarnos a MySQL utilizando Python. Asimismo, utilizando esa conexión ejecutaremos algunas sentencias DDL y DML de SQL. Finalmente utilizaremos SQLAlchemy para crear un engine y trabajar con la base de datos utilizando Pandas.



**MySQL + SQL Alchemy + Pandas**

```
In [1]: # Importamos las librerías principales
# En caso de no tener instalado mysql.connector ---> Utilizar pip install mysql-
import mysql.connector
import pandas as pd
import sqlalchemy as db
```

### Instalación de MySQL server

En la documentación oficial podemos encontrar las instrucciones para instalar MySQL Server. También podemos acceder directamente al link, descargar el navegador y seguir los pasos de instalación

- Documentación oficial: <https://dev.mysql.com/doc/refman/5.7/en/installing.html>
- Instalador de windows: <https://dev.mysql.com/downloads/installer/>

Una vez instalado, podemos ingresar al mySQL 5.7 Command Line Client, utilizando el password definido en la etapa de instalación.

```
MySQL 5.7 Command Line Client
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.7.35-log MySQL Community Server (GPL)

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

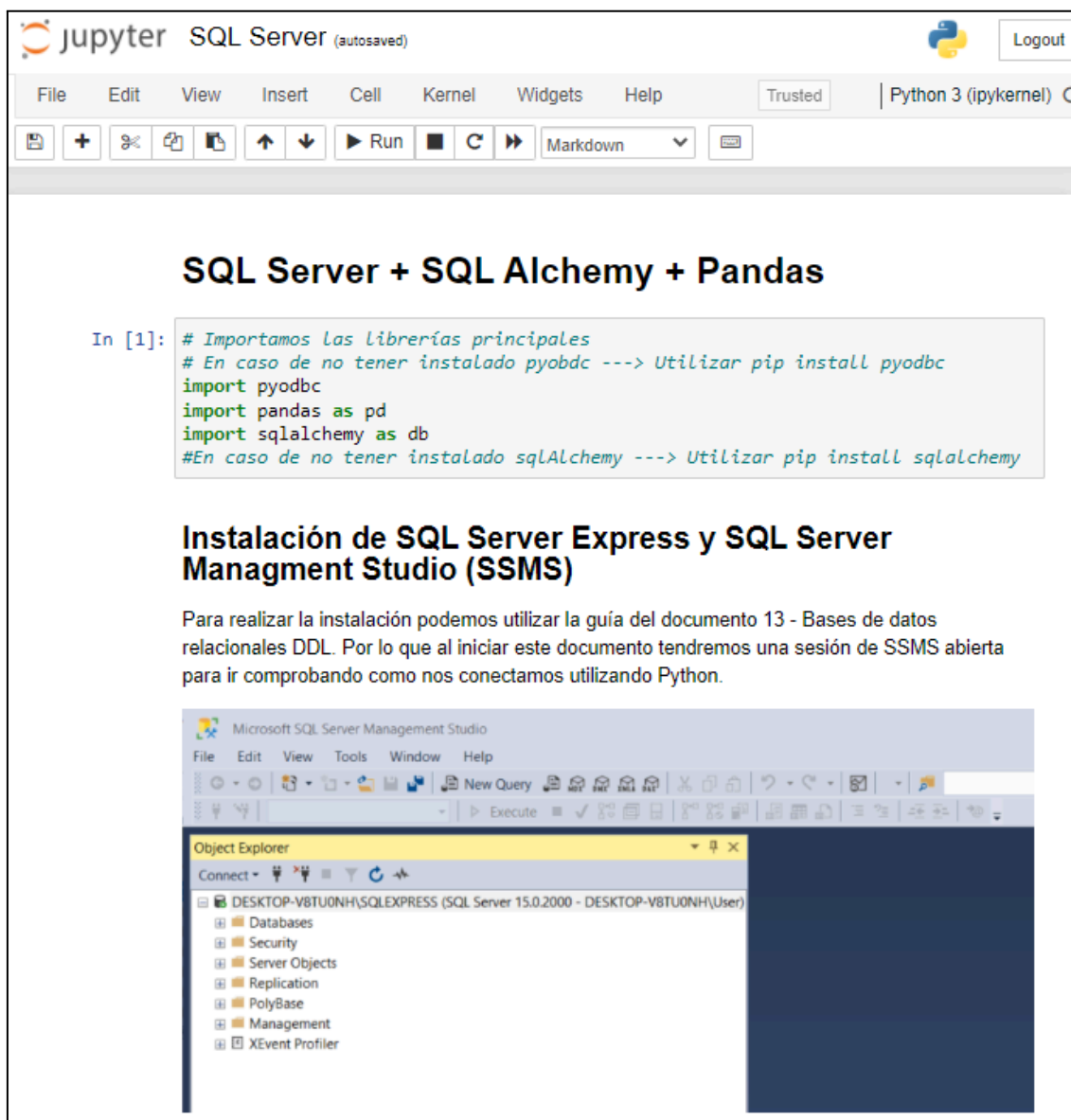
mysql>
```

## SQL Server

Microsoft SQL Server es un sistema de gestión de base de datos relacionales (RDBMS) desarrollado por la empresa Microsoft.

El lenguaje de desarrollo utilizado (por línea de comandos o mediante la interfaz gráfica de Management Studio) es Transact-SQL (TSQL), una implementación del estándar ANSI del lenguaje SQL, utilizado para manipular y recuperar datos (DML), crear tablas y definir relaciones entre ellas (DDL).

En el siguiente [link](#) vamos a encontrar un notebook con los pasos que debemos seguir para conectarnos a SQL Server utilizando Python. Asimismo, utilizando esa conexión ejecutaremos algunas sentencias DDL y DML de SQL. Finalmente utilizaremos SQLAlchemy para crear un engine y trabajar con la base de datos utilizando Pandas.



The screenshot shows a Jupyter Notebook interface with the title 'SQL Server (autosaved)'. The notebook contains a code cell with the following Python code:

```
In [1]: # Importamos Las librerías principales
# En caso de no tener instalado pyodbc ---> Utilizar pip install pyodbc
import pyodbc
import pandas as pd
import sqlalchemy as db
#En caso de no tener instalado sqlalchemy ---> Utilizar pip install sqlalchemy
```

Below the code cell, there is a section titled 'Instalación de SQL Server Express y SQL Server Management Studio (SSMS)'. The text reads: 'Para realizar la instalación podemos utilizar la guía del documento 13 - Bases de datos relacionales DDL. Por lo que al iniciar este documento tendremos una sesión de SSMS abierta para ir comprobando como nos conectamos utilizando Python.'

Below the text, there is a screenshot of the Microsoft SQL Server Management Studio (SSMS) interface. The 'Object Explorer' pane is visible, showing the following structure:

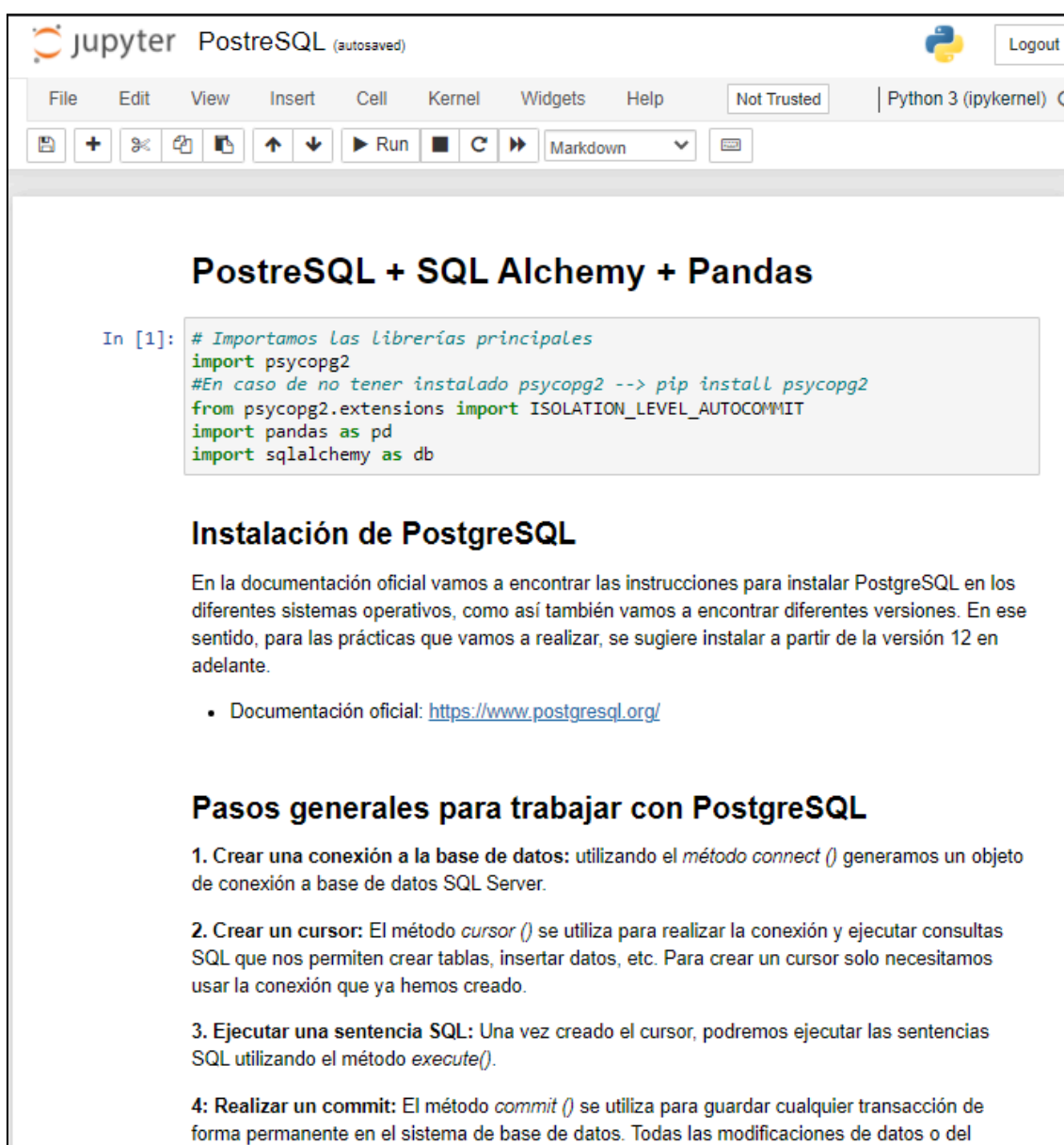
- DESKTOP-VBTU0NH\SQLEXPRESS (SQL Server 15.0.2000 - DESKTOP-VBTU0NH\User)
  - Databases
  - Security
  - Server Objects
  - Replication
  - PolyBase
  - Management
  - XEvent Profiler

## PostgreSQL

PostgreSQL es un sistema de gestión de base de datos relacionales (RDBMS) avanzado, de clase empresarial y de código abierto. Admite consultas SQL (relacionales) y JSON (no relacionales).

PostgreSQL es una base de datos altamente estable respaldada por más de 20 años de desarrollo por parte de la comunidad de código abierto. Utiliza como base de datos principal para muchas aplicaciones web, así como para aplicaciones móviles y de análisis.

En el siguiente [link](#) vamos a encontrar un notebook con los pasos que debemos seguir para conectarnos a MySQL utilizando Python. Asimismo, utilizando esa conexión ejecutaremos algunas sentencias DDL y DML de SQL. Finalmente utilizaremos SQLAlchemy para crear un engine y trabajar con la base de datos utilizando Pandas.



The screenshot shows a Jupyter Notebook interface with the title 'PostgreSQL (autosaved)'. The notebook contains a code cell with the following Python code:

```
In [1]: # Importamos las librerías principales
import psycopg2
#En caso de no tener instalado psycopg2 --> pip install psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT
import pandas as pd
import sqlalchemy as db
```

Below the code cell, the notebook has a section titled 'Instalación de PostgreSQL' with the following text:

En la documentación oficial vamos a encontrar las instrucciones para instalar PostgreSQL en los diferentes sistemas operativos, como así también vamos a encontrar diferentes versiones. En ese sentido, para las prácticas que vamos a realizar, se sugiere instalar a partir de la versión 12 en adelante.

- Documentación oficial: <https://www.postgresql.org/>

Below this, there is a section titled 'Pasos generales para trabajar con PostgreSQL' with the following steps:

1. Crear una conexión a la base de datos: utilizando el método `connect()` generamos un objeto de conexión a base de datos SQL Server.
2. Crear un cursor: El método `cursor()` se utiliza para realizar la conexión y ejecutar consultas SQL que nos permiten crear tablas, insertar datos, etc. Para crear un cursor solo necesitamos usar la conexión que ya hemos creado.
3. Ejecutar una sentencia SQL: Una vez creado el cursor, podremos ejecutar las sentencias SQL utilizando el método `execute()`.
4. Realizar un commit: El método `commit()` se utiliza para guardar cualquier transacción de forma permanente en el sistema de base de datos. Todas las modificaciones de datos o del



# Tratamiento de datos con Pandas

## Notebook práctico

En el siguiente [notebook](#) vamos a trabajar con algunos de los métodos más utilizados de la librería Pandas y luego vamos a enfocarnos en diferentes técnicas de filtrado y tratamiento de data frames, que nos será de mucha utilidad al momento de realizar un análisis exploratorio de datos.

**Datos con Pandas II**

Utilizar la librería Pandas para filtrar datos. Comprender conceptos de copias de dataframes utilizando el parámetro inplace

In [1]: `# Importamos la librería Pandas. En caso de no estar instalada, ejecutar --> pip install pandas`  
`import pandas as pd`

**Dataset a utilizar: Winter olympic medals**

Este dataset se encuentra disponible en la web y nos brinda información de los ganadores de medallas olímpicas entre 1924 y 2006

In [2]: `url = 'http://winterolympicsmedals.com/medals.csv'`  
`df = pd.read_csv(url)`  
`df`

Out[2]:

	Year	City	Sport	Discipline	NOC	Event	Event gender	Medal
0	1924	Chamonix	Skating	Figure skating	AUT	individual	M	Silver
1	1924	Chamonix	Skating	Figure skating	AUT	individual	W	Gold
2	1924	Chamonix	Skating	Figure skating	AUT	pairs	X	Gold
3	1924	Chamonix	Bobsleigh	Bobsleigh	BEL	four-man	M	Bronze
4	1924	Chamonix	Ice Hockey	Ice Hockey	CAN	ice hockey	M	Gold
...	...	...	...	...	...	...	...	...
2306	2006	Turin	Skiing	Snowboard	USA	Half-pipe	M	Silver
2307	2006	Turin	Skiing	Snowboard	USA	Half-pipe	W	Gold
2308	2006	Turin	Skiing	Snowboard	USA	Half-pipe	W	Silver
2309	2006	Turin	Skiing	Snowboard	USA	Snowboard Cross	M	Gold
2310	2006	Turin	Skiing	Snowboard	USA	Snowboard Cross	W	Silver

2311 rows × 8 columns

## Uso de loc e iloc en Pandas: Filtrado de Datos

La biblioteca Pandas proporciona las funciones `loc` e `iloc` para acceder y filtrar datos en un `DataFrame`. Aquí hay una explicación más detallada de cómo funcionan y cómo se utilizan para filtrar datos:

### **loc:** Acceso por Etiquetas

`loc` se utiliza para acceder a un grupo de filas y columnas utilizando etiquetas o una matriz booleana.

Puedes usar etiquetas para seleccionar filas o columnas específicas utilizando nombres de índices o nombres de columnas.

La sintaxis general es `df.loc[fila, columna]`, donde `fila` y `columna` pueden ser etiquetas individuales, listas de etiquetas o segmentos (por ejemplo, `inicio:fin`).

Ejemplo: `df.loc['fila1':'fila3', 'columna1':'columna3']` selecciona las filas de 'fila1' a 'fila3' y las columnas de 'columna1' a 'columna3'.

### **iloc:** Acceso por Posición

`iloc` se utiliza para la indexación basada en la posición, es decir, utilizando números enteros para seleccionar filas y columnas.

Puedes usar índices enteros para seleccionar filas o columnas específicas, similar a cómo se utiliza la indexación en las listas de Python.

La sintaxis general es `df.iloc[fila, columna]`, donde `fila` y `columna` pueden ser números enteros, listas de números enteros o segmentos.

Ejemplo: `df.iloc[0:3, 0:3]` selecciona las tres primeras filas y columnas del `DataFrame`.

### **Filtrado de Datos:**

Puedes filtrar datos en un `DataFrame` creando subconjuntos basados en condiciones lógicas.

Utilizas expresiones booleanas para especificar las condiciones de filtrado, y luego utilizas estas expresiones para seleccionar filas que cumplan con esas condiciones.

La sintaxis general es `df[condicion]`, donde `condicion` es una expresión booleana que devuelve una Serie de valores booleanos.

Ejemplo: `df[df['columna'] > 10]` selecciona todas las filas donde los valores en la columna 'columna' son mayores que 10.

Estos métodos son poderosos para acceder y filtrar datos en un DataFrame de Pandas. Son fundamentales para realizar análisis de datos y explorar conjuntos de datos de manera efectiva en Python.

### Implementar cada uno de los requerimientos utilizando Pandas:

```

py.py > ...
1 import pandas as pd
2
3 # Supongamos que tenemos un DataFrame df con columnas 'name', 'age' y 'city'
4
5 # Creamos sub_df_loc que obtiene las filas de df donde la edad es mayor a 30 y solo las columnas 'name' y 'city'
6 sub_df_loc = df.loc[df['age'] > 30, ['name', 'city']]
7
8 # Creamos sub_df_iloc que obtiene las filas en las posiciones 2 y 3 (excluyendo la posición 4) y las primeras dos columnas (índices 0 y 1)
9 sub_df_iloc = df.iloc[2:4, 0:2]
10
11 # Creamos df_chicago que contiene solo las filas donde la ciudad es 'Chicago'
12 df_chicago = df[df['city'] == 'Chicago']

```

Estos son ejemplos simples de cómo puedes utilizar `loc` e `iloc` para filtrar y seleccionar datos en un DataFrame de Pandas según ciertas condiciones y posiciones. Asegúrate de ajustar los nombres de las columnas y los valores de las condiciones según la estructura real de tu DataFrame.

## Uso de la Biblioteca Pandas para Limpieza y Procesamiento de Datos

La biblioteca Pandas es una herramienta poderosa en Python para trabajar con datos tabulares. Permite cargar, limpiar, procesar y analizar datos de una manera eficiente. A continuación, se detallan algunos conceptos clave sobre cómo utilizar Pandas para limpiar y procesar datos:

### DataFrame de Pandas:

Pandas proporciona una estructura de datos llamada DataFrame, que es una tabla bidimensional con filas y columnas etiquetadas.

Los datos en un DataFrame pueden ser de diferentes tipos, como números, cadenas, fechas, etc.

Es posible cargar datos en un DataFrame desde diferentes fuentes, como archivos CSV, Excel, bases de datos, etc.

- El DataFrame es la estructura de datos principal de Pandas y se utiliza para representar datos tabulares de manera bidimensional.
- Puedes crear un DataFrame desde diccionarios de Python, listas de listas, arrays de NumPy, archivos CSV, Excel, bases de datos SQL, etc.
- Un DataFrame consta de filas y columnas etiquetadas, lo que facilita la manipulación y el análisis de los datos.
- Puedes acceder a las filas y columnas del DataFrame utilizando etiquetas o índices numéricos.

### Exploración de Datos:

Una vez que los datos están cargados en un DataFrame, se puede utilizar una variedad de métodos y atributos para explorar los datos.

Algunos métodos útiles para explorar los datos incluyen `head()` para ver las primeras filas del DataFrame, `info()` para obtener información sobre los tipos de datos y los valores nulos, y `describe()` para obtener estadísticas descriptivas de los datos numéricos.

- El **método `head()`** muestra las primeras filas del DataFrame, lo que proporciona una vista previa rápida de los datos.
- El **método `info()`** proporciona información sobre el DataFrame, incluidos los tipos de datos de cada columna y el recuento de valores no nulos.
- El **método `describe()`** genera estadísticas descriptivas, como recuento, media, desviación estándar, mínimo, percentiles y máximo, para las columnas numéricas del DataFrame.

## Limpieza de Datos:

Pandas ofrece varias funciones para limpiar datos, como eliminar filas o columnas con valores nulos (`dropna()`), rellenar valores nulos con algún valor específico (`fillna()`), eliminar duplicados (`drop_duplicates()`), y más.

Es importante identificar y manejar valores nulos, valores atípicos y datos incorrectos para garantizar la calidad de los datos antes de realizar análisis adicionales.

- El **método `dropna()`** se utiliza para eliminar filas o columnas que contienen valores nulos.
- El **método `fillna()`** se utiliza para rellenar valores nulos con un valor específico, como la media, mediana o un valor constante.
- El **método `drop_duplicates()`** elimina filas duplicadas del DataFrame.
- Pandas también proporciona funciones para detectar y eliminar valores atípicos, como el cálculo del rango intercuartílico (IQR) y la eliminación de filas con valores fuera de cierto rango.

## Ordenamiento de Datos:

Pandas permite ordenar datos en un DataFrame según los valores de una o más columnas.

Esto se puede lograr utilizando el método `sort_values()` para ordenar los datos en orden ascendente o descendente según los valores de una columna específica.

- El **método `sort_values()`** ordena el DataFrame según los valores de una o más columnas.
- Puedes especificar la columna por la que deseas ordenar y si quieres que sea en orden ascendente o descendente.
- El ordenamiento de datos es útil para visualizar los datos ordenados o prepararlos para análisis posteriores.

## Separación de Datos:

A veces, es útil separar un DataFrame en conjuntos de datos más pequeños según ciertos criterios.

Esto se puede hacer utilizando la indexación booleana para filtrar filas según una condición específica o utilizando el método `groupby()` para agrupar los datos según los valores de una columna y luego iterar sobre los grupos resultantes.

- La indexación booleana se utiliza para filtrar filas según una condición específica. Por ejemplo, puedes seleccionar todas las filas donde el valor de una columna sea mayor que cierto umbral.
- El **método `groupby()`** se utiliza para agrupar los datos según los valores de una o más columnas.
- Puedes aplicar funciones de agregación a los grupos resultantes, como sumar, contar, calcular la media, etc.

### Visualización de Datos:

Pandas también ofrece capacidades básicas de visualización para ayudar a comprender los datos.

Se pueden generar gráficos simples, como histogramas, diagramas de dispersión y gráficos de barras, utilizando métodos como `plot()`.

Al comprender estos conceptos básicos y utilizar las funciones proporcionadas por Pandas de manera efectiva, puedes realizar una variedad de tareas de limpieza y procesamiento de datos de manera eficiente en Python.

- Pandas proporciona capacidades básicas de visualización a través del **método `plot()`**.
- Puedes generar diferentes tipos de gráficos, como histogramas, diagramas de dispersión, gráficos de líneas y de barras, directamente desde un DataFrame.

## **Método dropna()** (Eliminar datos NaN)

```
df1_clean = df1.dropna()
```

## **Método replace()**

```
DataFrame1 = DataFrame1.apply(lambda x:
x.str.replace("</a>", ""))
```

## **Expresiones Regulares (Regex)**

```
def correos(texto):
    regex =
    r'([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2
    ,4})'
    correos = re.findall(regex,texto)
    return correos
df_correos = df1.applymap(correos)
```

## Cierre 🖐️

¡Llegaste al final del manual de Pandas! 💪

A lo largo de este módulo de Python con Pandas, exploramos el concepto y las funcionalidades fundamentales de la **librería Pandas**, aprendimos a acceder a datos y a crear dataframes, centrándonos en las funciones básicas para su creación y carga. Además, repasamos operaciones esenciales como Merge, grouping, Reshaping y Plotting.

También trabajamos en la habilidad para conectarnos a diversos **sistemas de gestión de bases de datos relacionales** (RDBMS) en conjunto con Pandas. Conocimos los adaptadores disponibles para conectar: SQLite, MySQL, PostgreSQL y SQL Server y profundizamos en la ejecución de sentencias SQL para crear bases de datos, tablas, insertar registros y realizar consultas.

Hubo espacio para conocer cómo se realiza la integración de SQLAlchemy y ese aprendizaje nos permitió conectar Pandas con distintos RDBMS. Esta conexión posibilitó la consulta de tablas y la manipulación de datos mediante dataframes, así como la capacidad de escribir registros en la base de datos a partir de un dataframe.

Finalmente, exploramos técnicas para extraer subconjuntos de datos en dataframes, empleando diversas estrategias que facilitaron la extracción tanto de filas como de columnas.

**¡Excelente trabajo!** 🚀 Nos vemos en el siguiente manual o módulo.



# Referencias

- [Pandas dataframes](#)
- [Python Dataframes](#)
- [pandas.read\\_csv](#)
- [pandas.Dataframes.merge](#)
- [pandas.Dataframes.groupby](#)
- [pandas.melt](#)
- [Pandas.Dataframes.plot](#)
- <https://docs.sqlalchemy.org/en/14/core/engines.html>
- <https://www.sqlite.org/index.html>
- <https://www.postgresql.org/docs/>
- <https://dev.mysql.com/doc/refman/8.0/en>
- <https://www.microsoft.com/es-es/sql-server/>



# ¡Muchas gracias!

Nos vemos en la próxima clase