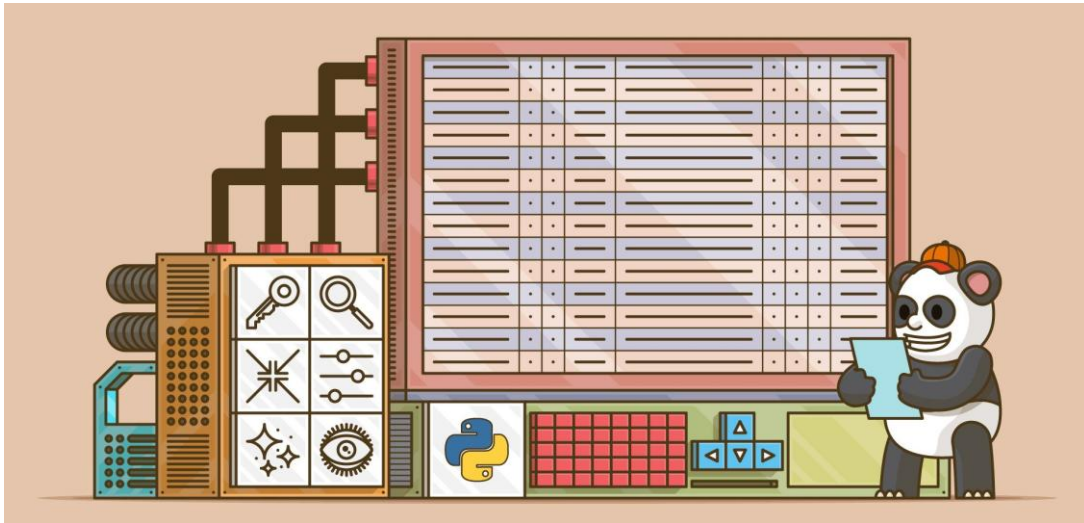


## Data Science Basics

# Pandas

# Pandas



Source: <https://realpython.com/>

Pandas es una librería de Python utilizada para trabajar con conjuntos de datos.

<https://pandas.pydata.org/>

Tiene funciones para **analizar, limpiar, explorar y manipular** datos.

El nombre "Pandas" hace referencia tanto a "Panel Data" como a "Python Data Analysis" y fue creado por Wes McKinney en 2008.

Pandas nos permite analizar big data y sacar conclusiones basadas en teorías estadísticas.

Pandas puede limpiar conjuntos de datos desordenados y; hacerlos legibles y relevantes.

# Instalar e importar

[https://pandas.pydata.org/pandas-docs/stable/getting\\_started/install.html](https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html)

Pandas es un paquete fácil de instalar. Abra su programa de terminal (shell o cmd) e instálelo usando cualquiera de los siguientes comandos:

Para los usuarios de jupyter notebook, ejecutar esta celda:

- El ! al principio ejecuta las celdas como si estuvieran en una terminal.

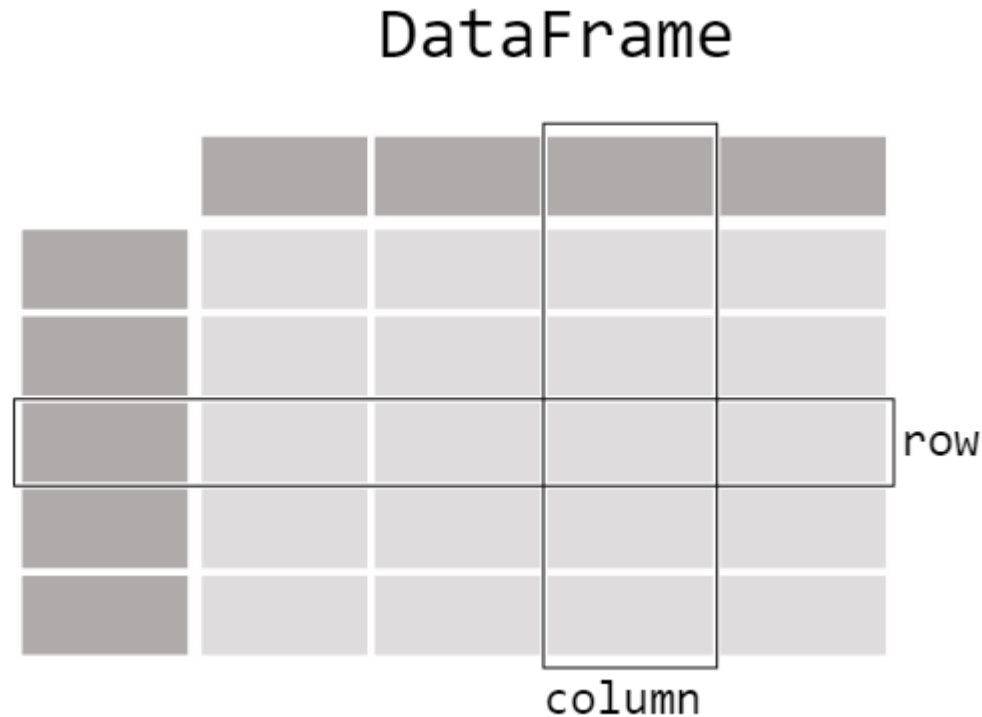
Para importar pandas:

```
$ conda install pandas
0
$ pip install pandas
```

```
!pip install pandas
```

```
import pandas as pd
```

# Representación de la tabla de datos



Un data frame es una estructura de datos bidimensional, es decir, los datos se alinean de forma tabular en filas y columnas.

<https://pandas.pydata.org/docs/reference/frame.html>

# Componentes principales de pandas: Series y DataFrames

| Series |   | Series  |   | DataFrame |         |
|--------|---|---------|---|-----------|---------|
| apples |   | oranges |   | apples    | oranges |
| 0      | 3 | 0       | 0 | 0         | 3       |
| 1      | 2 | 1       | 3 | 1         | 2       |
| 2      | 0 | 2       | 7 | 2         | 0       |
| 3      | 1 | 3       | 2 | 3         | 1       |

- Los dos componentes principales de pandas son la **Serie** y el **DataFrame**.
  - Una **serie** es esencialmente una columna.
  - Un **DataFrame** es una tabla multidimensional compuesta por una colección de Series.
- DataFrames y Series son bastante similares en el sentido de que muchas operaciones que se puede realizar con unas, se pueden realizar con las otras.
  - Como completar valores nulos y calcular la media.

## DataFrame:

- Potencialmente las columnas son de diferentes tipos.
- Tamaño: mutable
- Ejes etiquetados: filas y columnas
- Puede realizar operaciones aritméticas en filas y columnas

# Tipos de estructura de datos en Pandas

| Data Structure     | Dimensions | Description   |
|--------------------|------------|---|
| <b>Series</b>      | 1          | 1D labeled <u>homogeneous</u> array with immutable size   |
| <b>Data Frames</b> | 2          | General 2D labeled, size mutable tabular structure with potentially <u>heterogeneously</u> typed columns. |
| <b>Panel</b>       | 3          | General 3D labeled, size mutable array.   |

## Series

- La serie es una matriz unidimensional (matriz 1D) como estructura con datos homogéneos.

## DataFrame

- DataFrame es una matriz bidimensional (2D Array) con datos heterogéneos.

## Panel

- El panel es una estructura de datos tridimensional (3D Array) con datos heterogéneos.
- Es difícil representar el panel en representación gráfica.
- Pero un panel se puede ilustrar como un contenedor de DataFrame

# pandas.DataFrame

```
pandas.DataFrame(data, index , columns , dtype , copy )
```

- **data:** los datos toman varias formas como *ndarray*, *series*, *mapas*, *listas*, *dict*, constantes y también otro *DataFrame*.
- **index:** (opcional) para las *etiquetas de filas*, que se utilizarán para el frame resultante. El valor predeterminado es *np.arange(n)* si no se pasa ningún índice.
- **columns:** (opcional) para las *etiquetas de columnas*, la sintaxis predeterminada es - *np.arange(n)*. Esto solo es cierto si no se pasa ningún índice.
- **dtype:** tipo de dato de cada columna.
- **copy:** se usa para copiar datos.

## Crear un DataFrame

Se puede crear un DataFrame usando varias entradas como:

- list
- dict
- Series
- ndarrays
- DataFrame

# Crear un DataFrame desde cero

- Hay muchas formas de crear un DataFrame desde cero, pero una gran opción es simplemente usar un ***dict*** simple.
- Primero se debe **importar** pandas.
- Digamos que tenemos un puesto de frutas que vende manzanas y naranjas. Queremos tener una columna para cada fruta y una fila para cada compra del cliente.
- Para organizar estos datos como un **diccionario** para pandas:
- Y luego pasárselo al constructor pandas **DataFrame**:

```
import pandas as pd
```

```
data = { 'apples':[3, 2, 0, 1] , 'oranges':[0, 3, 7, 2] }
```

```
df = pd.DataFrame(data)
```

|   | apples | oranges |
|---|--------|---------|
| 0 | 3      | 0       |
| 1 | 2      | 3       |
| 2 | 0      | 7       |
| 3 | 1      | 2       |



# ¿Cómo funciona?

Cada elemento (clave, valor) en los datos corresponde a una columna en el DataFrame resultante.

- El **índice** (0-3) de este DataFrame fue generado en la creación. Pero también podemos crear el nuestro propio índice cuando inicializamos el DataFrame.

- P.ej. si se desea tener nombres de clientes como índice:

```
df = pd.DataFrame(data, index=['Luis', 'Ana', 'Juana', 'Pedro'])
```

|       | apples | oranges |
|-------|--------|---------|
| Luis  | 3      | 0       |
| Ana   | 2      | 3       |
| Juana | 0      | 7       |
| Pedro | 1      | 2       |

- Ahora podríamos **localizar** el pedido de un cliente usando sus nombres:

```
df.loc['Ana']
```

```
apples    2
oranges    3
Name: Ana, dtype: int64
```

# pandas.DataFrame.from\_dict

```
pandas.DataFrame.from_dict(data, orient='columns', dtype=None, columns=None)
```

- **data** : dict
  - De la forma **{field:array-like}** o **{field:dict}**.
- **orient** : **{'columns', 'index'}**, default **'columns'**
  - La "orientación" de los datos.
  - Si las claves de los datos deben ser las columnas del DataFrame resultante, usar 'columns' (predeterminado).
  - De lo contrario, si las claves deben ser filas, usar 'index'.
- **dtype** : **dtype**, default **None**
  - Tipo de datos a forzar, de lo contrario los infiere.
- **columns** : **list**, default **None**
  - Etiquetas de columna a usar cuando **orient='index'**.
  - Levanta una error **ValueError** si se usa con **orient='columns'**.

# Keyword: orient

```
data = {'col_1':[3, 2, 1, 0], 'col_2':['a','b','c','d']}  
pd.DataFrame.from_dict(data)
```



|   | col_1 | col_2 |
|---|-------|-------|
| 0 | 3     | a     |
| 1 | 2     | b     |
| 2 | 1     | c     |
| 3 | 0     | d     |

```
data = {'row_1':[3, 2, 1, 0], 'row_2':['a','b','c','d']}  
pd.DataFrame.from_dict(data, orient='index')
```



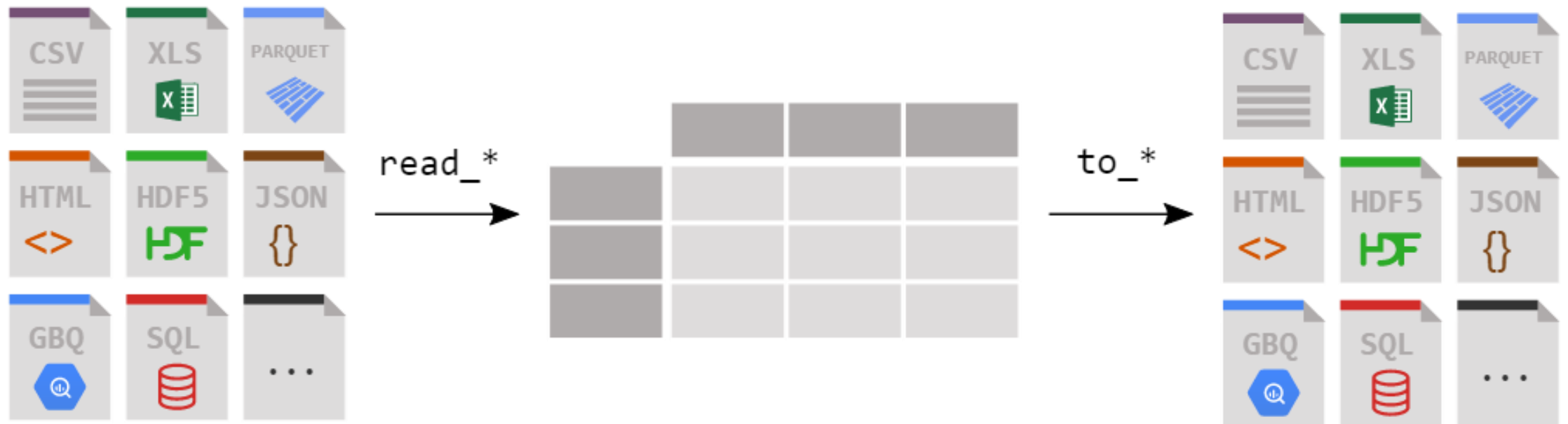
|       | 0 | 1 | 2 | 3 |
|-------|---|---|---|---|
| row_1 | 3 | 2 | 1 | 0 |
| row_2 | a | b | c | d |

```
data = {'row_1':[3, 2, 1, 0], 'row_2':['a','b','c','d']}  
pd.DataFrame.from_dict(data, orient = 'index', columns = ['A','B','C','D'])
```

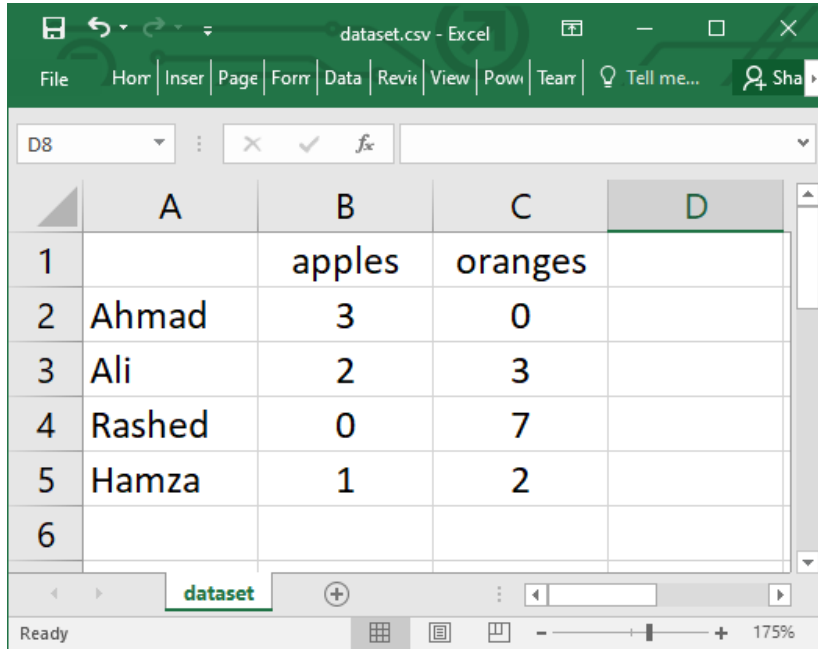


|       | A | B | C | D |
|-------|---|---|---|---|
| row_1 | 3 | 2 | 1 | 0 |
| row_2 | a | b | c | d |

# Cargar DataFrames desde archivos



# Leer datos de un CSV



|   | A      | B      | C       | D |
|---|--------|--------|---------|---|
| 1 |        | apples | oranges |   |
| 2 | Ahmad  | 3      | 0       |   |
| 3 | Ali    | 2      | 3       |   |
| 4 | Rashed | 0      | 7       |   |
| 5 | Hamza  | 1      | 2       |   |
| 6 |        |        |         |   |

```
File Edit Format Run Options Window Help
1 import pandas as pd
2
3 df = pd.read_csv('dataset.csv')
4 print(df)
5
6 # OR
7
8 df = pd.read_csv('dataset.csv', index_col=0)
9 print(df)
10
```

Ln: 6 Col: 0

# Leer datos de un CSV

- Con los archivos CSV, todo lo que se necesita es una sola línea para cargar los datos:

```
df = pd.read_csv('data/frutas.csv')
```

|   | Unnamed: 0 | apples | oranges |
|---|------------|--------|---------|
| 0 | Luis       | 3      | 0       |
| 1 | Ana        | 2      | 3       |
| 2 | Juana      | 0      | 7       |
| 3 | Pedro      | 1      | 2       |

- Los CSV **no tienen índices** como los DataFrames de Pandas, por lo que debemos designar el **index\_col** al leer el CSV:
  - Aquí estamos configurando el índice para que sea la columna cero.

```
dff = pd.read_csv('data/frutas.csv',  
index_col=0)
```

|       | apples | oranges |
|-------|--------|---------|
| Luis  | 3      | 0       |
| Ana   | 2      | 3       |
| Juana | 0      | 7       |
| Pedro | 1      | 2       |

# Leer datos de un JSON

- Un JSON es esencialmente un *dict* de Python:

```
df = pd.read_json('data/frutas.json')
```

- Tener en cuenta que esta vez nuestro **índice vino correctamente**, ya que el uso de JSON permitió que los índices funcionaran mediante anidamiento.
- Pandas intentará **descubrir cómo crear** un DataFrame analizando la estructura de su JSON y, a veces, no lo hace bien.
- A menudo, se deberá establecer el argumento de palabra clave **orient** según la estructura

# Ejemplo 1: Leer datos de un JSON

```
{  
  "apples" : { "Ahmad": 3, "Ali": 2, "Rashed": 0, "Hamza": 1 },  
  "oranges" : { "Ahmad": 0, "Ali": 3, "Rashed": 7, "Hamza": 2 }  
}
```



```
File Edit Format Run Options Window Help  
1 import pandas as pd  
2  
3 df = pd.read_json('dataset.json')  
4 print(df)  
Ln: 1 Col: 0
```



|        | apples | oranges |
|--------|--------|---------|
| Ahmad  | 3      | 0       |
| Ali    | 2      | 3       |
| Rashed | 0      | 7       |
| Hamza  | 1      | 2       |



## Ejemplo 2: Leer datos de un JSON

```
{  
  "Ahmad" : {"apples":3, "oranges":0},  
  "Ali" : {"apples":2, "oranges":3},  
  "Rashed" : {"apples":0, "oranges":7},  
  "Hamza" : {"apples":1, "oranges":2}  
}
```



```
File Edit Format Run Options Window Help  
1 import pandas as pd  
2  
3 df = pd.read_json('dataset.json')  
4 print(df)  
Ln: 1 Col: 0
```



|         | Ahmad | Ali | Rashed | Hamza |
|---------|-------|-----|--------|-------|
| apples  | 3     | 2   | 0      | 1     |
| oranges | 0     | 3   | 7      | 2     |

# Ejemplo 3: Leer datos de un JSON

```
{
  "Ahmad" : {"apples":3, "oranges":0},
  "Ali" : {"apples":2, "oranges":3},
  "Rashed" : {"apples":0, "oranges":7},
  "Hamza" : {"apples":1, "oranges":2}
}
```

```
File Edit Format Run Options Window Help
1 import pandas as pd
2
3 df = pd.read_json('dataset.json',
4                   orient='column')
5 print(df)
```

Ln: 6 Col: 0

|         | Ahmad | Ali | Rashed | Hamza |
|---------|-------|-----|--------|-------|
| apples  | 3     | 2   | 0      | 1     |
| oranges | 0     | 3   | 7      | 2     |

```
File Edit Format Run Options Window Help
1 import pandas as pd
2
3 df = pd.read_json('dataset.json',
4                   orient='index')
5 print(df)
```

Ln: 6 Col: 0

|        | apples | oranges |
|--------|--------|---------|
| Ahmad  | 3      | 0       |
| Ali    | 2      | 3       |
| Rashed | 0      | 7       |
| Hamza  | 1      | 2       |

# Convertir a CSV o JSON

- Después de un trabajo extenso en la limpieza de los datos, el siguiente paso será guardarlos como un archivo.
- De manera similar a las formas en que leemos los datos, pandas proporciona comandos intuitivos para guardarlos:
- Cuando guardamos archivos JSON y CSV, todo lo que tenemos que indicar es nuestro nombre de archivo deseado con la extensión de archivo adecuada.

```
df.to_csv('new_dataset.csv')  
df.to_json('new_dataset.json')  
df.to_sql('new_dataset', con)
```

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to\\_json.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html)

# Operaciones más importantes de DataFrame

Los DataFrames poseen cientos de métodos y otras operaciones que son cruciales para cualquier análisis.

Al menos se debe conocer las operaciones para:

- realizan **transformaciones simples** de los datos.
- proporcionan un **análisis estadístico** fundamental de los datos.

# Cargando un dataset de películas

- Cargamos el conjunto de datos desde un CSV y designamos los títulos de las películas para que sean nuestro índice.
- <https://grouplens.org/datasets/movielens/>

```
movies_df = pd.read_csv("data/movies.csv", index_col="title")
```

# Visualizar los datos

- Lo primero que debe hacer al abrir un nuevo conjunto de datos es imprimir algunas filas para tenerlas como referencia visual.
- Esto lo logramos con **.head()**:
  - `.head()` genera las primeras cinco filas de su DataFrame de forma predeterminada, pero también podríamos pasar un número: **`movies_df.head(10)`** generaría las diez primeras filas, por ejemplo.
- Para ver las últimas cinco filas, use **.tail()** que también acepta un número, y en este caso imprimimos las dos filas inferiores:

```
movies_df.head()
```

```
movies_df.tail(2)
```

# Obtener información sobre los datos

- **.info()** debe ser uno de los primeros comandos que ejecutemos después de cargar los datos.
- **.info()** proporciona los **detalles esenciales** sobre el conjunto de datos.
- Como: la cantidad de filas y columnas, la cantidad de valores no nulos, qué tipo de datos hay en cada columna y cuánta memoria está utilizando su DataFrame.
- **shape** nos muestra las dimensiones de nuestro Dataframe.

```
movies_df.info()
```

OUT:

```
<class 'pandas.core.frame.DataFrame'>  
Index: 1000 entries, Guardians of the Galaxy to Nine Lives  
Data columns (total 11 columns):  
Rank                1000 non-null int64  
Genre               1000 non-null object  
Description         1000 non-null object  
Director            1000 non-null object  
Actors              1000 non-null object  
Year               1000 non-null int64  
Runtime (Minutes)   1000 non-null int64  
Rating              1000 non-null float64  
Votes              1000 non-null int64  
Revenue (Millions)  872 non-null float64  
Metascore           936 non-null float64  
dtypes: float64(3), int64(4), object(4)  
memory usage: 93.8+ KB
```

```
movies_df.shape
```

OUT:

```
(1000, 11)
```

# Manejo de duplicados

- Este conjunto de datos no tiene filas duplicadas, pero siempre es importante verificar que no se estén agregando filas duplicadas.
  - Para demostrarlo, simplemente dupliquemos nuestro DataFrame de películas agregándolo a sí mismo:
  - El uso de **append()** [deprecated] o **concat()** devolverá una copia sin afectar el DataFrame original.
- La llamada **.shape** demuestra rápidamente que nuestras filas de DataFrame se han duplicado.

```
temp_df = movies_df.append(movies_df)
temp_df.shape
```

```
temp_df = pd.concat([movies_df, movies_df], ignore_index=True)
temp_df.shape
```

OUT:

(2000, 11)

- Ahora podemos intentar eliminar duplicados:

```
temp_df = temp_df.drop_duplicates()
temp_df.shape
```

OUT:

(1000, 11)



# Manejo de duplicados

- Al igual que `append()`, el método **`drop_duplicates()`** también devolverá una copia del DataFrame, pero esta vez con los duplicados eliminados.
  - Llamar a `shape()` confirma que volvimos a las 1000 filas de nuestro conjunto de datos original.
- Tener que re-asignar el DataFrame a la misma variable como en este ejemplo es un poco engorrosa.
- Por eso, pandas tiene el argumento de palabra clave *inplace* en muchos de sus métodos.
  - Usar **`inplace=True`** modificará el objeto DataFrame en su lugar:

```
temp_df.drop_duplicates(inplace=True)
```

- Otro argumento importante para `drop_duplicates()` es **`keep`**, que tiene tres opciones posibles:
  - **`first`**: (predeterminado) Elimina los duplicados excepto la primera aparición.
  - **`last`**: elimina los duplicados excepto la última aparición.
  - **`false`**: elimina todos los duplicados.

<https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/>

# Entender las variables

- Usar **describe()** en un DataFrame completo, podemos obtener un resumen de la distribución de variables continuas:

```
movies_df.describe()
```

OUT:

|       | rank        | year        | runtime     | rating      |      |
|-------|-------------|-------------|-------------|-------------|------|
| count | 1000.000000 | 1000.000000 | 1000.000000 | 1000.000000 | 1.00 |
| mean  | 500.500000  | 2012.783000 | 113.172000  | 6.723200    | 1.69 |
| std   | 288.819436  | 3.205962    | 18.810908   | 0.945429    | 1.88 |
| min   | 1.000000    | 2006.000000 | 66.000000   | 1.900000    | 6.10 |
| 25%   | 250.750000  | 2010.000000 | 100.000000  | 6.200000    | 3.6  |
| 50%   | 500.500000  | 2014.000000 | 111.000000  | 6.800000    | 1.10 |
| 75%   | 750.250000  | 2016.000000 | 123.000000  | 7.400000    | 2.3  |
| max   | 1000.000000 | 2016.000000 | 191.000000  | 9.000000    | 1.79 |

- `describe()` también se puede usar en una variable categórica para obtener el recuento de filas, el recuento único de categorías, la categoría superior y la frecuencia de la categoría superior:

```
movies_df['genres'].describe()
```

OUT:

```
count          1000
unique          207
top    Action,Adventure,Sci-Fi
freq           50
Name: genre, dtype: object
```

- Esto nos dice que la columna de género tiene 207 valores únicos, el valor superior es Acción/Aventura/Ciencia ficción, que aparece 50 veces (frecuencia).

# Más ejemplos

```
import pandas as pd
data = [1,2,3,10,20,30]
df = pd.DataFrame(data)
print(df)
```



|   | 0  |
|---|----|
| 0 | 1  |
| 1 | 2  |
| 2 | 3  |
| 3 | 10 |
| 4 | 20 |
| 5 | 30 |

```
import pandas as pd
data = {'Name' : ['AA', 'BB'], 'Age': [30,45]}
df = pd.DataFrame(data)
print(df)
```



|   | Name | Age |
|---|------|-----|
| 0 | AA   | 30  |
| 1 | BB   | 45  |

# Más ejemplos

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print(df)
```



|   | a | b  | c    |
|---|---|----|------|
| 0 | 1 | 2  | NaN  |
| 1 | 5 | 10 | 20.0 |

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print(df)
```



|        | a | b  | c    |
|--------|---|----|------|
| first  | 1 | 2  | NaN  |
| second | 5 | 10 | 20.0 |

# Más ejemplos

Este ejemplo muestra cómo crear un DataFrame con una lista de diccionarios, índices de fila e índices de columna.

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])

print(df1)
print('.....')
print(df2)
```



|        | a | b   |
|--------|---|-----|
| first  | 1 | 2   |
| second | 5 | 10  |
| .....  |   |     |
|        | a | b1  |
| first  | 1 | NaN |
| second | 5 | NaN |

# Más ejemplos:

## Crear un DataFrame a partir de un Dict de Series

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3] , index=['a', 'b', 'c']),
     'two' : pd.Series([1,2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df)
```



|   | one | two |
|---|-----|-----|
| a | 1.0 | 1   |
| b | 2.0 | 2   |
| c | 3.0 | 3   |
| d | NaN | 4   |

# Más ejemplos: Adición de columna

```
import pandas as pd

d = {'one':pd.Series([1,2,3], index=['a','b','c']),
     'two':pd.Series([1,2,3,4], index=['a','b','c','d'])
}
df = pd.DataFrame(d)
# Adding a new column to an existing DataFrame object
# with column label by passing new series

print("Adding a new column by passing as Series:")
df['three'] = pd.Series([10,20,30],index=['a','b','c'])
print(df)

print("Adding a column using an existing columns in DataFrame:")
df['four'] = df['one']+df['three']
print(df)
```



Adding a column using Series:

|   | one | two | three |
|---|-----|-----|-------|
| a | 1.0 | 1   | 10.0  |
| b | 2.0 | 2   | 20.0  |
| c | 3.0 | 3   | 30.0  |
| d | NaN | 4   | NaN   |

Adding a column using columns:

|   | one | two | three | four |
|---|-----|-----|-------|------|
| a | 1.0 | 1   | 10.0  | 11.0 |
| b | 2.0 | 2   | 20.0  | 22.0 |
| c | 3.0 | 3   | 30.0  | 33.0 |
| d | NaN | 4   | NaN   | NaN  |

# Más ejemplos: Borrado de columna

```
# Using the previous DataFrame, we will delete a column
# using del function
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
     'three' : pd.Series([10,20,30], index=['a','b','c'])
    }
df = pd.DataFrame(d)
print ("Our dataframe is:")
print(df)

# using del function
print("Deleting the first column using DEL function:")
del df['one']
print(df)

# using pop function
print("Deleting another column using POP function:")
df.pop('two')
print(df)
```



Our dataframe is:

|   | one | two | three |
|---|-----|-----|-------|
| a | 1.0 | 1   | 10.0  |
| b | 2.0 | 2   | 20.0  |
| c | 3.0 | 3   | 30.0  |
| d | NaN | 4   | NaN   |

Deleting the first column:

|   | two | three |
|---|-----|-------|
| a | 1   | 10.0  |
| b | 2   | 20.0  |
| c | 3   | 30.0  |
| d | 4   | NaN   |

Deleting another column:

|   |      |
|---|------|
| a | 10.0 |
| b | 20.0 |
| c | 30.0 |
| d | NaN  |



# Más ejemplos: Slicing en DataFrames

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df[2:4])
```



|   | one | two |
|---|-----|-----|
| c | 3.0 | 3   |
| d | NaN | 4   |

# Más ejemplos: Adición de filas

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df)

df2 = pd.DataFrame([[5,6], [7,8]], columns = ['a', 'b'])
df = df.append(df2 )
print(df)
```



|   | one | two |     |     |
|---|-----|-----|-----|-----|
| a | 1.0 | 1   |     |     |
| b | 2.0 | 2   |     |     |
| c | 3.0 | 3   |     |     |
| d | NaN | 4   |     |     |
|   | one | two | a   | b   |
| a | 1.0 | 1.0 | NaN | NaN |
| b | 2.0 | 2.0 | NaN | NaN |
| c | 3.0 | 3.0 | NaN | NaN |
| d | NaN | 4.0 | NaN | NaN |
| 0 | NaN | NaN | 5.0 | 6.0 |
| 1 | NaN | NaN | 7.0 | 8.0 |

# Más ejemplos: Borrado de filas

```
import pandas as pd

d = {'one':pd.Series([1, 2, 3], index=['a','b','c']),
     'two':pd.Series([1, 2, 3, 4], index=['a','b','c','d'])
}
df = pd.DataFrame(d)
print(df)

df2 = pd.DataFrame([[5,6], [7,8]], columns = ['a', 'b'])
df = df.append(df2 )
print(df)

df = df.drop(0)
print(df)
```

|   | one | two |     |     |
|---|-----|-----|-----|-----|
| a | 1.0 | 1   |     |     |
| b | 2.0 | 2   |     |     |
| c | 3.0 | 3   |     |     |
| d | NaN | 4   |     |     |
|   | one | two | a   | b   |
| a | 1.0 | 1.0 | NaN | NaN |
| b | 2.0 | 2.0 | NaN | NaN |
| c | 3.0 | 3.0 | NaN | NaN |
| d | NaN | 4.0 | NaN | NaN |
| 0 | NaN | NaN | 5.0 | 6.0 |
| 1 | NaN | NaN | 7.0 | 8.0 |
|   | one | two | a   | b   |
| a | 1.0 | 1.0 | NaN | NaN |
| b | 2.0 | 2.0 | NaN | NaN |
| c | 3.0 | 3.0 | NaN | NaN |
| d | NaN | 4.0 | NaN | NaN |
| 1 | NaN | NaN | 7.0 | 8.0 |

# Más ejemplos: Re-indexación

```
import pandas as pd

# Creating the first dataframe
df1 = pd.DataFrame({"A": [1, 5, 3, 4, 2],
                    "B": [3, 2, 4, 3, 4],
                    "C": [2, 2, 7, 3, 4],
                    "D": [4, 3, 6, 12, 7]},
                    index = ["A1", "A2", "A3", "A4", "A5"])

# Creating the second dataframe
df2 = pd.DataFrame({"A": [10, 11, 7, 8, 5],
                    "B": [21, 5, 32, 4, 6],
                    "C": [11, 21, 23, 7, 9],
                    "D": [1, 5, 3, 8, 6]},
                    index = ["A1", "A3", "A4", "A7", "A8"])

# Print the first dataframe
print(df1)
print(df2)
# find matching indexes
df1.reindex_like(df2)
```

- **dataframe.reindex\_like()** devuelve un objeto con índices coincidentes a mí mismo.
- Los índices que no coinciden se rellenan con valores NaN.

Out[72]:

|    | A   | B   | C   | D    |
|----|-----|-----|-----|------|
| A1 | 1.0 | 3.0 | 2.0 | 4.0  |
| A3 | 3.0 | 4.0 | 7.0 | 6.0  |
| A4 | 4.0 | 3.0 | 3.0 | 12.0 |
| A7 | NaN | NaN | NaN | NaN  |
| A8 | NaN | NaN | NaN | NaN  |

# Más ejemplos: Concatenar dataframes

```
import pandas as pd
df1 = pd.DataFrame({'Name':['A','B'], 'SSN':[10,20], 'marks':[90, 95] })
df2 = pd.DataFrame({'Name':['B','C'], 'SSN':[25,30], 'marks':[80, 97] })
df3 = pd.concat([df1, df2])
df3
```

Out[69]:

|   | Name | SSN | marks |
|---|------|-----|-------|
| 0 | A    | 10  | 90    |
| 1 | B    | 20  | 95    |
| 0 | B    | 25  | 80    |
| 1 | C    | 30  | 97    |

# Manejo de datos categóricos

- Hay muchos datos que son repetitivos, por ejemplo, el género, el país y los códigos postales siempre son repetitivos.
- Las **variables categóricas** sólo pueden asumir una cantidad limitada
- El tipo de datos categóricos es útil en los siguientes casos:
  - Una variable de cadena que consta de solo unos pocos valores diferentes. Convertir una variable de cadena de este tipo en una variable categórica ahorrará algo de memoria.
  - El orden léxico de una variable no es lo mismo que el orden lógico ("uno", "dos", "tres"). Al convertir a categórico y especificar un orden en las categorías, la clasificación y el mínimo/máximo usarán el orden lógico en lugar del orden léxico.
  - Como una señal para otras bibliotecas de Python de que esta columna debe tratarse como una variable categórica (por ejemplo, para usar métodos estadísticos adecuados o tipos de gráficos).

# Ejemplos

```
import pandas as pd

cat = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
print(cat)
```

```
import pandas as pd
import numpy as np

cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a", "c"])
df = pd.DataFrame({"cat": cat, "s": ["a", "c", "c", np.nan]})
print(df.describe())
print(df["cat"].describe())
```

# Lectura de datos de una base de datos SQL

- Si se está trabajando con datos de una base de datos SQL, primero debe establecer una conexión utilizando una biblioteca de Python adecuada y luego pasar una consulta a pandas.
- Usaremos **SQLite** para demostrarlo.
- Primero, necesitamos que **pysqlite3** esté instalado. Ejecutar este comando en el terminal:
- O, ejecutar en la celda:
- sqlite3 se usa para crear una conexión a una base de datos que luego podemos usar para generar un DataFrame a través de una consulta SELECT.
- Primero haremos una conexión a un archivo de base de datos SQLite:
- En esta base de datos SQLite tenemos una tabla llamada movies, y nuestro índice está en una columna llamada **"title"**.
- Al pasar una consulta SELECT y nuestra conexión, podemos leer de la tabla:

```
$ pip install pysqlite3
```

```
!pip install pysqlite3
```

```
import sqlite3  
con = sqlite3.connect("data/database.db")
```

```
df = pd.read_sql_query("SELECT * FROM movies", con)
```



# Lectura de datos de una base de datos SQL

- Al igual que con los CSV, podríamos pasar **index\_col='title'**, o podemos establecer un índice a posteriori:
- De hecho, podríamos usar **set\_index()** en cualquier DataFrame usando cualquier columna en cualquier momento.
- Indexar Series y DataFrames es una tarea muy común, y vale la pena recordar las diferentes formas de hacerlo.

```
df = pd.read_sql_query("SELECT * FROM movies", con,index_col='title')
```

```
df = df.set_index('title')
```

OUT:

|        | apples | oranges |
|--------|--------|---------|
| index  |        |         |
| June   | 3      | 0       |
| Robert | 2      | 3       |
| Lily   | 0      | 7       |
| David  | 1      | 2       |

# Guardar datos de una base de datos SQL

- Podemos guardar el Dataframe en la tabla.

```
movies_df.to_sql(name='movies', con=con)
```

# Otras funcionalidades interesantes:

Para probar estas funcionalidades usaremos el dataset iris.data:

<https://archive.ics.uci.edu/ml/datasets/iris>

```
import pandas as pd

df = pd.read_csv('data/iris.data',header=None)
```

| Funcionalidad         | Ejemplos   |
|-----------------------|--|
| Dar nombre a columnas | <pre>nombres = ['long_sepalo', 'ancho_sepalo', 'long_petalos', 'ancho_petalos', 'clase'] df.columns = nombres</pre>  |
| Índices               | <pre>df.index</pre>  |
| Conteo de valores     | <pre>df['clase'].value_counts()</pre>  |
| Uso de memoria        | <pre>df.memory_usage()</pre>   |
| Transposición         | <pre>df.T</pre>  |
| Ordenamiento          | <pre>df.sort_values('ancho_sepalo',ascending=False)</pre>  |
| Filtrado              | <pre>df[['long_sepalo', 'long_petalos']] df[:3] df.loc[[3,1,5],['ancho_sepalo']] df.iloc[:5,2:] df.iloc[[4,19],[0,2]] df[(df['long_sepalo']&gt;5) &amp; (df['long_petalos']&gt;2)]</pre> |

# Otras funcionalidades interesantes:

| Funcionalidad                   | Ejemplos   |
|---------------------------------|--|
| Operaciones con columnas        | <code>df['long_sepalo']-df['long_petalos']</code>  |
| Valores faltantes               | <code>df.isna()</code><br><code>df.isna().sum()</code><br><code>df.isna().sum().sum()</code><br><code>df['long_petalos'][:2]=np.nan</code>     |
| Completar los valores faltantes | <code>df['long_petalos'].fillna(2)</code><br><code>df['long_petalos']=df['long_petalos'].fillna(2)</code>                                      |
| Estadísticas básicas            | <code>df['long_petalos'].mean()</code><br><code>df['long_petalos'].median()</code>   |
| Agrupamiento                    | <code>df_group=df.groupby('clase')['ancho_petalos'].mean()</code><br><code>df_group.name='media_ancho_petalos'</code><br><code>df_group</code> |

# Entendiendo los datos con estadísticas

- Observar los datos en bruto **data.head()**
- Comprobación de las dimensiones de los datos con **data.shape**
- Obtener el tipo de datos de cada atributo con **data.dtypes**
  - <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dtypes.html>
- Resumen estadístico de datos con **data.describe()**
- Revisar la distribución de clases con **data.groupby()**
- Revisión de la correlación entre atributos con **data.corr(method='pearson')**
  - <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html>
- Revisar el sesgo de la distribución de atributos con **data.skew()**
  - <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.skew.html>

# Referencias

Documentación pandas:

- <https://pandas.pydata.org/pandas-docs/stable/index.html>

Input/output:

- <https://pandas.pydata.org/pandas-docs/stable/reference/io.html>

DataFrame:

- <https://pandas.pydata.org/pandas-docs/stable/reference/frame.html>

Series:

- <https://pandas.pydata.org/pandas-docs/stable/reference/series.html>