# Bibliotecas de Python

# Introducción

Python es un lenguaje de programación muy popular, poderoso y versátil que cuenta con una amplia gama de bibliotecas que ayudan a que la programación sea más fácil y eficiente. Pero, **¿qué son las bibliotecas?** Las bibliotecas son conjuntos de módulos que contienen funciones, clases y variables relacionadas, que permiten realizar tareas sin tener que escribir el código desde cero y este se puede reutilizar en múltiples programas y proyectos.

Entre las bibliotecas disponibles se encuentran las estándares, que se incluye con cada instalación de Python, y las de código abierto creadas por la gran comunidad de desarrolladores, que constantemente genera nuevas bibliotecas y mejora las existentes. Por ello es aconsejable que, al momento de utilizarlas, se verifique si existe alguna actualización en las guías de usuario.

Asimismo, estas bibliotecas se pueden clasificar según su aplicación y funcionalidad en: procesamiento de datos, visualización, aprendizaje automático, desarrollo web, procesamiento de lenguaje y de imágenes, entre otras. En este capítulo se analizarán tres de las bibliotecas más reconocidas y ampliamente utilizadas de Python: **Pandas** para manipulación de datos tabulares, **NumPy** para procesamiento numérico y **Matplotlib** para visualización.

### ¿Cómo se utilizan las bibliotecas?

Para acceder a una biblioteca y sus funciones, se debe instalar por única vez y luego, importar cada vez que la necesitemos.

En la parte superior de nuestro código debemos correr `import {nombre_de_biblioteca} as {nombre_corto_de_biblioteca}`. El alias o nombre corto de la biblioteca se suele agregar para lograr una mayor legibilidad del código, pero no es mandatorio.

```
import pandas as pd
```

> **Nota**
>
> En nuestro caso, la instalación no es necesaria, ya que vamos a utilizar Google Colab, pero en caso de usar otro IDE (como por ejemplo, Visual Studio Code), se realiza desde el símbolo del sistema (o en inglés: “Command Prompt”, o terminal o consola), corriendo: `pip install -nombre_de_biblioteca` o similares.

# Adaptaciones para Colab

A partir de esta guía es probable que debamos reutilizar ciertas cosas o adaptar la forma en la que venimos trabajando. Para esto, vamos a hacer una modificación:

Si tenemos variables o imports definidos en otra celda y no la ejecutamos, al intentar usarla nos va a dar error. Es importante entonces o correr todas las celdas, o que cada celda tenga la información necesaria para poder correrse de forma independiente. Acá vemos cómo, al querer ejecutar sólo la segunda celda, nos da error porque la variable `saludo` está definida en la celda anterior (que no se ejecutó).


## Pandas

Pandas es una biblioteca de código abierto diseñada específicamente para la manipulación y el análisis de datos en Python. Es una herramienta poderosa que puede ayudar a los usuarios a limpiar, transformar y analizar datos de una manera rápida y eficiente.

Se debe importar. Por convención:

```
import pandas as pd
```

Los datos a trabajar en Pandas van a tener, en general, una forma muy similar a las tablas:

| | Jurisdicción | Capital | Población (hab) | Superficie (km2) | PBI |
|---|---|---|---|---|---|
| 0 | Ciudad Autónoma de Buenos Aires | | 3075646 | 205.9 | 1.548638e+08 |
| 1 | Buenos Aires | La Plata | 17541141 | 305907.4 | 2.926899e+08 |
| 2 | Catamarca | San Fernando del Valle de Catamarca | 415438 | 101486.1 | 6.150949e+06 |
| 3 | Chaco | Resistencia | 1204541 | 99763.3 | 9.832643e+06 |
| 4 | Chubut | Rawson | 618994 | 224302.3 | 1.774785e+07 |

Figura 1: Ejemplo de tabla

Pandas es una biblioteca un poco más pesada que otras por todo lo que incluye, por lo que vamos a poder aprovechar mucho más Google Colab, que maneja grandes datos de forma rápida y eficiente.

## Serie

Pandas incorpora dos estructuras de datos nuevas llamadas: **Series** y **DataFrames**.

Una **serie** es un vector capaz de contener cualquier tipo de dato, como por ejemplo, números enteros o decimales, strings, objetos de Python, etc.

### Creación de una Serie

Para crearlas, se puede partir de un escalar, una lista, un diccionario, etc., utilizando `pd.Serie()`:

```
# Crear serie partiendo de una lista:
lista = [1, "a", 3.5]

pd.Series(lista)
```

```
0      1
1      a
2    3.5
dtype: object
```

Esto es parecido a las listas de Python.

Notemos que se ven dos líneas verticales de datos en el output. A la derecha se observa una columna con los elementos de la lista antes creada, y a la izquierda se encuentra el **índice**, formado por valores desde 0 a n-1, siendo n la cantidad de elementos. Este índice numérico es el predefinido, que nos muestra la posición de los elementos dentro de la Serie.
También podríamos cambiarlo si quisiéramos. En ese caso, se puede establecer utilizando `index`.

El índice es de vital importancia ya que permite acceder a los elementos de la serie (muy parecido a cómo las claves nos permiten acceden a los valores de un diccionario, o el índice a un elemento de una lista). Al elegir índices personalizados, tenemos que tener en cuenta que su longitud debe ser acorde al número de elementos de la Serie. De lo contrario, se mostrará un ValueError.

```
# Crear serie partiendo de una lista, indicando el índice
pd.Series(lista, index = ["x", "y", "z"])
```

```
x      1
y      a
z    3.5
dtype: object
```

También podemos crear Series utilizando diccionarios, y en ese caso, sus claves (o keys) pasan a formar el índice.

```
# Crear serie partiendo de un diccionario:
diccionario = {"x": 1, "y": "a", "z": 3.5}

a = pd.Series(diccionario)
a
```

```
x      1
y      a
z    3.5
dtype: object
```

#### Accediendo a un elemento

Como ya se debe estar imaginando, para acceder a un elemento de la serie, se debe indicar el valor del índice.

```
# Acceder al elemento de índice y:
a["y"]
```

```
'a'
```

Si los índices son numéricos, simplemente se le pasa entre corchetes un número.

### Operaciones con Series

Las Series no requieren recorrer valor por valor en un ciclo for si queremos realizar operaraciones. Podemos usar directamente operadores con Series.
Por ejemplo:

```
a + a
```

```
x      2
y     aa
z    7.0
dtype: object
```

```
a * 3
```

```
x       3
y     aaa
z    10.5
dtype: object
```

## DataFrame

Un **DataFrame** es una estructura de datos tabular **(bidimensional, en forma de tabla)**, compuesta por filas y columnas, que se parece mucho a una hoja de cálculo de Excel.

### De CSV a DataFrame

Cuando leemos un archivo CSV con `pd.read_csv()`, cada columna del CSV se convierte en una columna del DataFrame (y también en una Serie). Cada fila del CSV se convierte en una fila del DataFrame.

```
# Si tenemos un archivo "estudiantes.csv" con este contenido:
# nombre;edad;carrera
# Ana;20;Informática
# Juan;22;Mecánica

df = pd.read_csv('estudiantes.csv', sep=';')

# Ahora df tiene:
# - 2 filas (Ana y Juan)
```

```
# - 3 columnas (nombre, edad, carrera)
# - Cada columna es una Serie
```

### Creando DataFrames

Para crear un dataframe a partir de datos en vez de archivos, se utiliza `DataFrame()` y se ingresan diferentes estructuras como arrays, diccionarios, listas, series u otros dataframes.

En el siguiente ejemplo, se crea un Dataframe partiendo de un diccionario llamado `data` para las columnas y de una lista `label` para el índice:

```
data = {'columna_1': ['a', 'b', 'c', 'd', 'e', 'f'],
        'columna_2': [2.5, 3, 0.5, None, 5, None],
        'columna_3': [1, 3, 2, 3, 2, 3]}

labels = ['a1', 'a2', 'a3', 'a4', 'a5', 'a6']

df = pd.DataFrame(data, index=labels)
df
```

| | columna_1 | columna_2 | columna_3 |
|---|---|---|---|
| a1 | a | 2.5 | 1 |
| a2 | b | 3.0 | 3 |
| a3 | c | 0.5 | 2 |
| a4 | d | NaN | 3 |
| a5 | e | 5.0 | 2 |
| a6 | f | NaN | 3 |

Lo que vemos acá es el output de ejecutar el código de arriba en una celda: una tabla donde tenemos 3 columnas, y el índice de cada fila son los elementos de la lista `labels`.

En vez de tener un valor `None`, Pandas lo recibe y lo transforma en `NaN`, que significa "Not a Number". Esto es muy útil para trabajar con datos faltantes, ya que Pandas nos permite realizar operaciones con ellos sin que nos de error (a diferencia de Python).

### Atributos y descripción de un Dataframe

A continuación, se observa una tabla con métodos que nos permiten conocer las características de un determinado DataFrame.

| Método o atributo | Descripción |
|---|---|
| `.info()` | Resume la información del DataFrame |
| `.shape` | Devuelve una tupla con el número de filas y columnas |
| `.size` | Número de elementos, aunque también puede usarse `len` |
| `.columns` | Lista con los nombres de las columnas |
| `.index` | Lista con los nombres de las filas |
| `.head()` | Muestra las primeras filas |
| `.tail()` | Muestra las últimas filas |
| `.describe()` | Brinda métricas de las columnas numéricas |

Para ejemplificar los métodos y las funciones de Pandas, usaremos el **DataFrame df** definido en el siguiente bloque de código.

```
data = {'nombre': ['José Martínez', 'Rosa Díaz', 'Javier Garcíaz', 'Carmen López', 'Marisa C
                   'Pilar González', 'Pedro Tenorio', 'Santiago Manzano', 'Macarena Álvarez'
        'edad': [18, 32, 24, 35, 46, 68, 51, 22, 35, 46, 53, 58, 27, 20],
        'genero': ['H', 'M', 'H', 'M', 'X', 'H', 'H', 'M', 'H', 'X', 'M', 'H', 'H', 'M'],
        'peso': [85.0, 65.0, None, 65.0, 51.0, 66.0, 62.0, 60.0, 90.0, 75.0, 55.0, 78.0, 109
        'altura': [1.79, 1.73, 1.81, 1.7, 1.58, 1.74, 1.72, 1.66, 1.94, 1.85, 1.62, 1.87, 1.9
        'colesterol': [182.0, 232.0, 191.0, 200.0, 148.0, 249.0, 276.0, None, 241.0, 280.0, 1

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

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |
| 1 | Rosa Díaz | 32 | M | 65.0 | 1.73 | 232.0 |
| 2 | Javier Garcíaz | 24 | H | NaN | 1.81 | 191.0 |
| 3 | Carmen López | 35 | M | 65.0 | 1.70 | 200.0 |
| 4 | Marisa Collado | 46 | X | 51.0 | 1.58 | 148.0 |
| 5 | Antonio Ruiz | 68 | H | 66.0 | 1.74 | 249.0 |
| 6 | Antonio Fernández | 51 | H | 62.0 | 1.72 | 276.0 |
| 7 | Pilar González | 22 | M | 60.0 | 1.66 | NaN |
| 8 | Pedro Tenorio | 35 | H | 90.0 | 1.94 | 241.0 |
| 9 | Santiago Manzano | 46 | X | 75.0 | 1.85 | 280.0 |
| 10 | Macarena Álvarez | 53 | M | 55.0 | 1.62 | 262.0 |
| 11 | José Sanz | 58 | H | 78.0 | 1.87 | 198.0 |
| 12 | Miguel Gutiérrez | 27 | H | 109.0 | 1.98 | 210.0 |
| 13 | Carolina Moreno | 20 | M | 61.0 | 1.77 | 194.0 |

> **Nota**
>
> Muchas veces no vamos a querer modificar el dataframe original, pero sí vamos a querer manipularlo. En ese caso, podemos hacer una copia del dataframe original, de esta forma: `df_copy = df.copy()`

> **Importante**
>
> Editar un dataframe creado *a partir de un archivo*, no modifica el archivo en sí. Sólo los datos que tenemos en memoria.

### Info, dtypes, columns e index

Con `info()` se puede ver:

- el índice en la primera línea, que es un rango de 0 a 13
- el número total de columnas en la segunda línea
- el uso de la memoria en la última
- una tabla con los nombres de las columnas en **Column**, la cantidad de valores no nulos en **Non-Null Count** y el tipo de dato en **Dtype** para cada una de ellas.

```
df.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 14 entries, 0 to 13
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   nombre      14 non-null     str
 1   edad        14 non-null     int64
 2   genero      14 non-null     str
 3   peso        13 non-null     float64
 4   altura      14 non-null     float64
 5   colesterol  13 non-null     float64
dtypes: float64(3), int64(1), str(2)
memory usage: 804.0 bytes
```

Note que utilizando `columns` e `index` se obtiene parte de esta información:

```
# Nombre de cada columna
df.columns
```

```
Index(['nombre', 'edad', 'genero', 'peso', 'altura', 'colesterol'], dtype='str')
```

```
# índice
df.index
```

```
RangeIndex(start=0, stop=14, step=1)
```

### Shape y Size

La forma del DataFrame es de 14 filas y 6 columnas, por lo que contiene 84 elementos.

```
# Forma del DataFrame (filas, columnas)
df.shape
```

```
(14, 6)
```

```
# Número de elementos del DataFrame
df.size
```

```
84
```

### Head y Tail

Asimismo, cuando no conocemos un DataFrame, puede ser importante ver las primeras 5 filas con `head()` o las últimas con `tail()`. Si se quisiera observar un número determinado, sólo hay que especificarlo, por ejemplo:

```
# Mostrar las primeras 3 filas.
df.head(3)
```

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |
| 1 | Rosa Díaz | 32 | M | 65.0 | 1.73 | 232.0 |
| 2 | Javier Garcíaz | 24 | H | NaN | 1.81 | 191.0 |

```
# Mostrar las últimas 5 filas.
df.tail()
```

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 9 | Santiago Manzano | 46 | X | 75.0 | 1.85 | 280.0 |
| 10 | Macarena Álvarez | 53 | M | 55.0 | 1.62 | 262.0 |
| 11 | José Sanz | 58 | H | 78.0 | 1.87 | 198.0 |
| 12 | Miguel Gutiérrez | 27 | H | 109.0 | 1.98 | 210.0 |
| 13 | Carolina Moreno | 20 | M | 61.0 | 1.77 | 194.0 |

### Describe

Por otro lado, `describe()` devuelve un resumen descriptivo de las columnas de valores numéricos, como "edad", "peso", "altura" y "colesterol".

```
df.describe()
```

| | edad | peso | altura | colesterol |
|---|---|---|---|---|
| count | 14.000000 | 13.000000 | 14.000000 | 13.000000 |
| mean | 38.214286 | 70.923077 | 1.768571 | 220.230769 |
| std | 15.621379 | 16.126901 | 0.115016 | 39.847948 |
| min | 18.000000 | 51.000000 | 1.580000 | 148.000000 |
| 25 % | 24.750000 | 61.000000 | 1.705000 | 194.000000 |
| 50 % | 35.000000 | 65.000000 | 1.755000 | 210.000000 |
| 75 % | 49.750000 | 78.000000 | 1.840000 | 249.000000 |
| max | 68.000000 | 109.000000 | 1.980000 | 280.000000 |

Estas métricas podrían obtenerse de forma puntual (tanto para todo el DataFrame como para sólo una columna) utilizando funciones determinadas, como:

- `count()`: contabiliza los valores no nulos
- `mean()`: promedio
- `min()`: valor mínimo
- `max()`: valor máximo

Por ejemplo:

```
df.count()
```

```
nombre        14
edad          14
genero        14
peso          13
altura        14
colesterol    13
dtype: int64
```

```
df.min()
```

```
nombre        Antonio Fernández
edad                         18
genero                        H
peso                       51.0
altura                     1.58
colesterol                148.0
dtype: object
```

```
df.max()
```

```
nombre        Santiago Manzano
edad                        68
genero                       X
peso                     109.0
altura                    1.98
colesterol               280.0
dtype: object
```

### Accediendo a filas de un DataFrame

Finalmente, como ocurre con las series, para acceder a los elementos de un DataFrame se puede indicar la posición o el nombre de la fila o columna.

Para acceder a una fila en particular, utilizamos `iloc[]`. Podemos pasarle a `iloc` un entero, una lista de enteros, un rango de números (que indican las posiciones) o directamente el valor del índice.

```
# Mostrar la fila de posición 0, usando doble corchete [[]]
# Recibe una lista de elementos a mostrar (que contiene sólo al 0)
df.iloc[[0]]
```

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |

Si queremos mostrar un rango, lo hacemos así:

```
# Mostrar las filas de posición 0 y 3, usando doble corchete [[]]
# Recibe una lista de elementos a mostrar
df.iloc[[0, 3]]
```

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |
| 3 | Carmen López | 35 | M | 65.0 | 1.70 | 200.0 |

```
# Mostrar las filas de posiciones entre 0 hasta 3 (exclusive)
# Usa slices
df.iloc[:3]
```

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |
| 1 | Rosa Díaz | 32 | M | 65.0 | 1.73 | 232.0 |
| 2 | Javier Garcíaz | 24 | H | NaN | 1.81 | 191.0 |

```
# El equivalente a df.iloc[:3] es el uso de head(3)
df.head(3)
```

| | nombre | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |
| 1 | Rosa Díaz | 32 | M | 65.0 | 1.73 | 232.0 |
| 2 | Javier Garcíaz | 24 | H | NaN | 1.81 | 191.0 |

### Accediendo a columnas de un DataFrame

A veces no queremos sólo acceder a filas, sino a columnas. Por ejemplo, para calcular sumas, promedios, máximos o mínimos, etc.

Para acceder a una **columna** se pueden utilizar una lista con los nombres de las columnas que se quieren mostrar: `DataFrame[[columna1, columna2]]`.

```
# Mostrar la columna "nombre"
df[['nombre']]
```

| | nombre |
|---|---|
| 0 | José Martínez |
| 1 | Rosa Díaz |
| 2 | Javier Garcíaz |
| 3 | Carmen López |
| 4 | Marisa Collado |
| 5 | Antonio Ruiz |
| 6 | Antonio Fernández |
| 7 | Pilar González |
| 8 | Pedro Tenorio |
| 9 | Santiago Manzano |
| 10 | Macarena Álvarez |
| 11 | José Sanz |
| 12 | Miguel Gutiérrez |
| 13 | Carolina Moreno |

```
# Mostrar más de una columna: "nombre" y "edad":
df[['nombre', 'edad']]
```

| | nombre | edad |
|---|---|---|
| 0 | José Martínez | 18 |
| 1 | Rosa Díaz | 32 |

| | nombre | edad |
|---|---|---|
| 2 | Javier García | 24 |
| 3 | Carmen López | 35 |
| 4 | Marisa Collado | 46 |
| 5 | Antonio Ruiz | 68 |
| 6 | Antonio Fernández | 51 |
| 7 | Pilar González | 22 |
| 8 | Pedro Tenorio | 35 |
| 9 | Santiago Manzano | 46 |
| 10 | Macarena Álvarez | 53 |
| 11 | José Sanz | 58 |
| 12 | Miguel Gutiérrez | 27 |
| 13 | Carolina Moreno | 20 |

De esta forma, podríamos hacer algo así:

```
# calculamos el promedio para los valores de la columna 'edad'
df[['edad']].mean()
```

```
edad    38.214286
dtype: float64
```

### Modificar un Dataframe

A la hora de modificar un DataFrame, tenemos distintas posibilidades:

- Cambiar la estructura del mismo, como los nombres de las columnas y de los índices,
- Agregar una nueva filas o columna
- Reemplazar un dato en una determinada posición.

A continuación, se enumeran distintos métodos para llevar a cabo estos cambios.

| Método | Descripción |
|---|---|
| `rename()` | Renombra las columnas |
| `insert()` | Agrega columnas |
| `drop()` | Elimina columnas y filas |
| `loc[fila]` | Agrega una fila en un índice dado |
| `loc[fila, columna]` | Modifica un valor particular dado un índice y una columna |

| Método | Descripción |
|---|---|
| `map()` | Busca un valor dado en una columna y lo reemplaza |
| `replace()` | Reemplaza un valor dado en una columna |

### Rename, insert y drop

Para renombrar una columna, se utiliza un diccionario: `rename(columns={"nombre_columna": "nuevo_nombre_columna"})`

```
# Reemplazo "nombre" por "nombre y apellido"
df = df.rename(columns={"nombre": "nombre y apellido"})
df.head() # para que veamos el cambio de nombre en la columna
```

| | nombre y apellido | edad | genero | peso | altura | colesterol |
|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | 85.0 | 1.79 | 182.0 |
| 1 | Rosa Díaz | 32 | M | 65.0 | 1.73 | 232.0 |
| 2 | Javier Garcíaz | 24 | H | NaN | 1.81 | 191.0 |
| 3 | Carmen López | 35 | M | 65.0 | 1.70 | 200.0 |
| 4 | Marisa Collado | 46 | X | 51.0 | 1.58 | 148.0 |

Para agregar una nueva columna, existe el método `insert()`, que requiere indicar la posición de la nueva columna, el nombre de la nueva columna, y los valores de la misma.

Vamos a crear una lista llamada **direccion** con 14 valores, para cada una de las personas del DataFrame, y luego agregarla en la posición 3:

```
# Valores de la nueva columna
direccion = ["CABA", "Bs As", "Bs As", "Bs As", "CABA", "Bs As", "CABA", "CABA", "CABA", "CA

# Insertar la columna "direccion" en la posición 3 de columnas:
df.insert(3, "direccion", direccion)
df.head() # para que veamos que la columna nueva se agregó
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol |
|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | CABA | 85.0 | 1.79 | 182.0 |
| 1 | Rosa Díaz | 32 | M | Bs As | 65.0 | 1.73 | 232.0 |
| 2 | Javier Garcíaz | 24 | H | Bs As | NaN | 1.81 | 191.0 |

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol |
|---|---|---|---|---|---|---|---|
| 3 | Carmen López | 35 | M | Bs As | 65.0 | 1.70 | 200.0 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 |

Para agregar una nueva columna también podemos hacerlo directamente, como si fuera un diccionario: `df['nueva_columna'] = valores`.

Por ejemplo, supongamos que queremos ingresar una columna con el índice de masa corporal de las personas., que se calcula de la siguiente manera:

$$IMC = \frac{Peso(kg)}{Altura(m)^2}$$

Esto podemos hacerlo directamente trabajando sobre las columnas, como trabajábamos sobre los arrays de NumPy; y guardando el resultado en una nueva columna llamada "IMC".

```
# Crear la columna "IMC"
df["IMC"] = df["peso"] / df["altura"]**2
df.head()
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 1 | Rosa Díaz | 32 | M | Bs As | 65.0 | 1.73 | 232.0 | 21.718066 |
| 2 | Javier Garcíaz | 24 | H | Bs As | NaN | 1.81 | 191.0 | NaN |
| 3 | Carmen López | 35 | M | Bs As | 65.0 | 1.70 | 200.0 | 22.491349 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |

Esto lo que va a hacer es tomar todos los valores de peso y altura de cada fila, calcular el IMC y guardarlo en una nueva columna de la línea, bajo el nombre "IMC".

De manera similar, se puede crear la columna **dni** sin utilizar `insert()`, usando la lista **dni**:

```
df_copy = df.copy() # Hacemos una copia, para que no nos afecte el original
dni = [12345678, 23456789, 34567890, 45678901, 56789012, 67890123, 78901234, 89012345, 90123
df_copy["dni"] = dni
df_copy.head()
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC | dni |
|---|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | CABA | 85.0 | 1.79 | 182.0 | 26.528510 | 12345678 |
| 1 | Rosa Díaz | 32 | M | Bs As | 65.0 | 1.73 | 232.0 | 21.718066 | 23456789 |
| 2 | Javier Garcíaz | 24 | H | Bs As | NaN | 1.81 | 191.0 | NaN | 34567890 |
| 3 | Carmen López | 35 | M | Bs As | 65.0 | 1.70 | 200.0 | 22.491349 | 45678901 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 | 56789012 |

Por lo que vemos que no es indispensable usar `insert` para poder agregar una nueva columna, pero `insert` nos permite decir *en dónde* queremos agregarla.

Para eliminar una fila `(axis=0)` o columna `(axis=1)`, se utiliza `drop()`:

```
# Elimino una columna llamada "direccion". También podría hacer: `del df["direccion"]`
df = df.drop('direccion', axis=1)
```

```
# Elimino la fila 14
df = df.drop(14, axis=0)
```

### Insertar filas

Para agregar una nueva fila, se utiliza `loc[]`, que nos pide indicar el índice y los valores de la misma. Para ello, creamos una lista llamada **nueva_fila** con valores para cada columna del DataFrame.

```
# Valores de la nueva fila
nueva_fila = ['Carlos Rivas', 30, 'H', "CABA", 70.0, 1.75, 203.0, 22]

# Insertamos al final del DataFrame
largo = len(df.index) # o también: largo = df.shape[0], para obtener la cantidad de filas
df.loc[largo] = nueva_fila
df.tail() # Para ver que se agregó al final
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 10 | Macarena Álvarez | 53 | M | CABA | 55.0 | 1.62 | 262.0 | 20.957171 |
| 11 | José Sanz | 58 | H | Bs As | 78.0 | 1.87 | 198.0 | 22.305471 |
| 12 | Miguel Gutiérrez | 27 | H | CABA | 109.0 | 1.98 | 210.0 | 27.803285 |
| 13 | Carolina Moreno | 20 | M | CABA | 61.0 | 1.77 | 194.0 | 19.470778 |
| 14 | Carlos Rivas | 30 | H | CABA | 70.0 | 1.75 | 203.0 | 22.000000 |

> ¿Por qué usamos 'len' arriba?
>
> A diferencia de Python, podemos agregar una fila en una posición que no existe aún. Por eso arriba pudimos hacer `df.loc[largo]`.

### Modificar un valor

Finalmente, **para cambiar un valor determinado** también se utiliza `loc[]`, como por ejemplo, agregar el peso de Javier García (tercera fila), que antes tenía `NaN`:

```
df.loc[2, 'peso'] = 92
df.head()
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | H | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 1 | Rosa Díaz | 32 | M | Bs As | 65.0 | 1.73 | 232.0 | 21.718066 |
| 2 | Javier Garcíaz | 24 | H | Bs As | 92.0 | 1.81 | 191.0 | NaN |
| 3 | Carmen López | 35 | M | Bs As | 65.0 | 1.70 | 200.0 | 22.491349 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |

Para transformar los valores de una columna entera, podemos utilizar `map()` pasando un diccionario del estilo `{valor_viejo: valor_nuevo}`. Por ejemplo, modificar la columna "genero" reemplazando "H" por "M", "M" por "F":

```
df['genero'] = df['genero'].map({'H': 'M', 'M': 'F', 'X': 'X'})
df.head()
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | M | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 1 | Rosa Díaz | 32 | F | Bs As | 65.0 | 1.73 | 232.0 | 21.718066 |
| 2 | Javier Garcíaz | 24 | M | Bs As | 92.0 | 1.81 | 191.0 | NaN |
| 3 | Carmen López | 35 | F | Bs As | 65.0 | 1.70 | 200.0 | 22.491349 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |

Otra manera sería utilizando `replace()`. Por ejemplo, en la columna "direccion" vamos a modificar "Bs As" por "Buenos Aires".

```
df['direccion'] = df['direccion'].replace('Bs As', 'Buenos Aires')
df.head()
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | M | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 1 | Rosa Díaz | 32 | F | Buenos Aires | 65.0 | 1.73 | 232.0 | 21.718066 |
| 2 | Javier Garcíaz | 24 | M | Buenos Aires | 92.0 | 1.81 | 191.0 | NaN |
| 3 | Carmen López | 35 | F | Buenos Aires | 65.0 | 1.70 | 200.0 | 22.491349 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |

### Filtrar un Dataframe

Para filtrar los elementos de un DataFrame se suelen utilizar condiciones lógicas de la siguiente forma: `DataFrame[ condicion ]`.

Por ejemplo:

```
# Seleccionar aquellas personas menores de 40 años
# La condición es que la columna 'edad' del dataframe tenga valor menor a 40
df[ df['edad'] < 40 ]
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | M | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 1 | Rosa Díaz | 32 | F | Buenos Aires | 65.0 | 1.73 | 232.0 | 21.718066 |
| 2 | Javier Garcíaz | 24 | M | Buenos Aires | 92.0 | 1.81 | 191.0 | NaN |
| 3 | Carmen López | 35 | F | Buenos Aires | 65.0 | 1.70 | 200.0 | 22.491349 |
| 7 | Pilar González | 22 | F | CABA | 60.0 | 1.66 | NaN | 21.773842 |
| 8 | Pedro Tenorio | 35 | M | CABA | 90.0 | 1.94 | 241.0 | 23.913275 |
| 12 | Miguel Gutiérrez | 27 | M | CABA | 109.0 | 1.98 | 210.0 | 27.803285 |
| 13 | Carolina Moreno | 20 | F | CABA | 61.0 | 1.77 | 194.0 | 19.470778 |
| 14 | Carlos Rivas | 30 | M | CABA | 70.0 | 1.75 | 203.0 | 22.000000 |

Cuando se requieren múltiples condiciones, se puede adicionar usando símbolos como `&` para **and** y `|` para **or**. Por ejemplo:

```
# Seleccionar aquellas personas de genero femenino y menores de 40 años:
df[ (df['edad'] < 40) & (df['genero'] == 'F') ]
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 1 | Rosa Díaz | 32 | F | Buenos Aires | 65.0 | 1.73 | 232.0 | 21.718066 |
| 3 | Carmen López | 35 | F | Buenos Aires | 65.0 | 1.70 | 200.0 | 22.491349 |
| 7 | Pilar González | 22 | F | CABA | 60.0 | 1.66 | NaN | 21.773842 |
| 13 | Carolina Moreno | 20 | F | CABA | 61.0 | 1.77 | 194.0 | 19.470778 |

```
# Seleccionar aquellas personas cuyo peso es 60kg o 90kg:
df[(df['peso'] == 60.0) | (df['peso'] == 90.0)]
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 7 | Pilar González | 22 | F | CABA | 60.0 | 1.66 | NaN | 21.773842 |
| 8 | Pedro Tenorio | 35 | M | CABA | 90.0 | 1.94 | 241.0 | 23.913275 |

También puedo filtrar las fillas por el valor `NaN`.
Para eso, se utiliza la función `isnull()` que devuelve `True` si el valor de la columna es nulo o NaN. Por ejemplo:

```
df['IMC'].isnull()
```

```
0     False
1     False
2      True
3     False
4     False
5     False
6     False
7     False
8     False
9     False
10    False
11    False
12    False
13    False
14    False
Name: IMC, dtype: bool
```

Para visualizar aquellas filas donde el índice de masa corporal es nulo, filtramos:

```
df[df['IMC'].isnull()]
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 2 | Javier Garcíaz | 24 | M | Buenos Aires | 92.0 | 1.81 | 191.0 | NaN |

El método opuesto es `notnull()`, que devuelve `True` si el valor de la columna no es nulo o NaN. Por ejemplo:

```
df[df['IMC'].notnull()]
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 0 | José Martínez | 18 | M | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 1 | Rosa Díaz | 32 | F | Buenos Aires | 65.0 | 1.73 | 232.0 | 21.718066 |
| 3 | Carmen López | 35 | F | Buenos Aires | 65.0 | 1.70 | 200.0 | 22.491349 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |
| 5 | Antonio Ruiz | 68 | M | Buenos Aires | 66.0 | 1.74 | 249.0 | 21.799445 |
| 6 | Antonio Fernández | 51 | M | CABA | 62.0 | 1.72 | 276.0 | 20.957274 |
| 7 | Pilar González | 22 | F | CABA | 60.0 | 1.66 | NaN | 21.773842 |
| 8 | Pedro Tenorio | 35 | M | CABA | 90.0 | 1.94 | 241.0 | 23.913275 |
| 9 | Santiago Manzano | 46 | X | CABA | 75.0 | 1.85 | 280.0 | 21.913806 |
| 10 | Macarena Álvarez | 53 | F | CABA | 55.0 | 1.62 | 262.0 | 20.957171 |
| 11 | José Sanz | 58 | M | Buenos Aires | 78.0 | 1.87 | 198.0 | 22.305471 |
| 12 | Miguel Gutiérrez | 27 | M | CABA | 109.0 | 1.98 | 210.0 | 27.803285 |
| 13 | Carolina Moreno | 20 | F | CABA | 61.0 | 1.77 | 194.0 | 19.470778 |
| 14 | Carlos Rivas | 30 | M | CABA | 70.0 | 1.75 | 203.0 | 22.000000 |

### Ordenando y Contando

A continuación, se muestra una lista con métodos que resultan muy útiles a la hora de analizar datos:

| Método | Descripción |
|---|---|
| `sort_values(by, ascending)` | Ordena el DataFrame considerando los valores de la o las columnas determinadas y devuelve un DataFrame nuevo (no modifica el original) |

| Método | Descripción |
|---|---|
| `value_counts()` | Indica los valores únicos de una determinada columna y el número de veces que aparece en ella |

**Sort value:**

Para utilizar la función `sort_values(by, ascending)`, se debe indicar en el parámetro `by` una lista con las columnas para ordenar el DataFrame y en `ascending`, `True` si el orden deseado es creciente o `False` para decreciente.

En el siguiente ejemplo ordenamos por "nombre y apellido" en forma alfabética:

```
df.sort_values(by=['nombre y apellido'], ascending=[True])
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 6 | Antonio Fernández | 51 | M | CABA | 62.0 | 1.72 | 276.0 | 20.957274 |
| 5 | Antonio Ruiz | 68 | M | Buenos Aires | 66.0 | 1.74 | 249.0 | 21.799445 |
| 14 | Carlos Rivas | 30 | M | CABA | 70.0 | 1.75 | 203.0 | 22.000000 |
| 3 | Carmen López | 35 | F | Buenos Aires | 65.0 | 1.70 | 200.0 | 22.491349 |
| 13 | Carolina Moreno | 20 | F | CABA | 61.0 | 1.77 | 194.0 | 19.470778 |
| 2 | Javier Garcíaz | 24 | M | Buenos Aires | 92.0 | 1.81 | 191.0 | NaN |
| 0 | José Martínez | 18 | M | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 11 | José Sanz | 58 | M | Buenos Aires | 78.0 | 1.87 | 198.0 | 22.305471 |
| 10 | Macarena Álvarez | 53 | F | CABA | 55.0 | 1.62 | 262.0 | 20.957171 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |
| 12 | Miguel Gutiérrez | 27 | M | CABA | 109.0 | 1.98 | 210.0 | 27.803285 |
| 8 | Pedro Tenorio | 35 | M | CABA | 90.0 | 1.94 | 241.0 | 23.913275 |
| 7 | Pilar González | 22 | F | CABA | 60.0 | 1.66 | NaN | 21.773842 |
| 1 | Rosa Díaz | 32 | F | Buenos Aires | 65.0 | 1.73 | 232.0 | 21.718066 |
| 9 | Santiago Manzano | 46 | X | CABA | 75.0 | 1.85 | 280.0 | 21.913806 |

**¿Qué ocurre cuando ordenamos siguiendo varias columnas?** Los valores del DataFrame se ordenan siguiendo la primera columna en primer lugar, luego la segunda, y así sucesivamente.

```
df.sort_values(by=['genero', 'nombre y apellido'], ascending=[True, True])
```

| | nombre y apellido | edad | genero | direccion | peso | altura | colesterol | IMC |
|---|---|---|---|---|---|---|---|---|
| 3 | Carmen López | 35 | F | Buenos Aires | 65.0 | 1.70 | 200.0 | 22.491349 |
| 13 | Carolina Moreno | 20 | F | CABA | 61.0 | 1.77 | 194.0 | 19.470778 |
| 10 | Macarena Álvarez | 53 | F | CABA | 55.0 | 1.62 | 262.0 | 20.957171 |
| 7 | Pilar González | 22 | F | CABA | 60.0 | 1.66 | NaN | 21.773842 |
| 1 | Rosa Díaz | 32 | F | Buenos Aires | 65.0 | 1.73 | 232.0 | 21.718066 |
| 6 | Antonio Fernández | 51 | M | CABA | 62.0 | 1.72 | 276.0 | 20.957274 |
| 5 | Antonio Ruiz | 68 | M | Buenos Aires | 66.0 | 1.74 | 249.0 | 21.799445 |
| 14 | Carlos Rivas | 30 | M | CABA | 70.0 | 1.75 | 203.0 | 22.000000 |
| 2 | Javier Garcíaz | 24 | M | Buenos Aires | 92.0 | 1.81 | 191.0 | NaN |
| 0 | José Martínez | 18 | M | CABA | 85.0 | 1.79 | 182.0 | 26.528510 |
| 11 | José Sanz | 58 | M | Buenos Aires | 78.0 | 1.87 | 198.0 | 22.305471 |
| 12 | Miguel Gutiérrez | 27 | M | CABA | 109.0 | 1.98 | 210.0 | 27.803285 |
| 8 | Pedro Tenorio | 35 | M | CABA | 90.0 | 1.94 | 241.0 | 23.913275 |
| 4 | Marisa Collado | 46 | X | CABA | 51.0 | 1.58 | 148.0 | 20.429418 |
| 9 | Santiago Manzano | 46 | X | CABA | 75.0 | 1.85 | 280.0 | 21.913806 |

En el ejemplo de arriba, primero se ordena de manera creciente por genero, resultando en tres grupos: “genero” = “F”, “M” y “X”. Luego, cada uno de esos grupos se ordena por “nombre y apellido” de forma creciente.

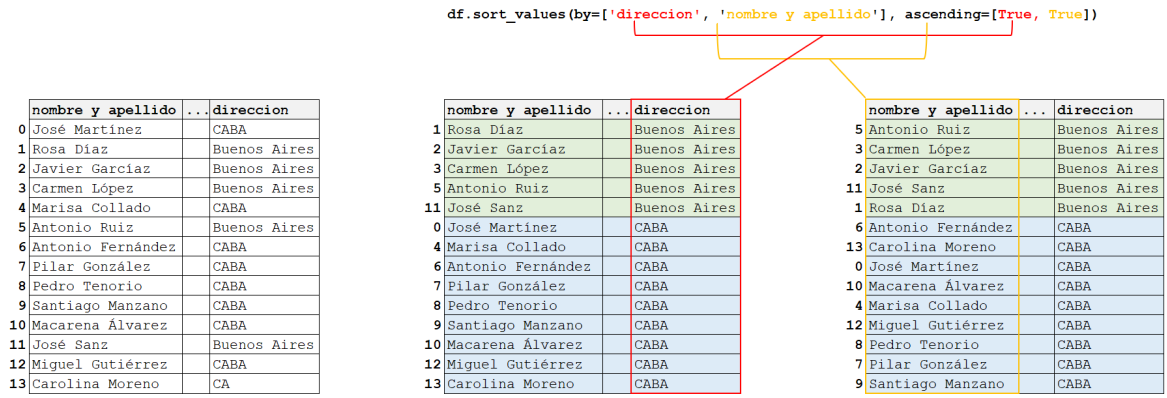



Figura 2: Ejemplo de sort_values()

### Value Count:

Utilizando `value_counts()` se pueden contar las filas en cada grupo según “direccion”. Es decir, cuenta cuántas apariciones hay de cada valor en una columna.

```python
df['direccion'].value_counts()
```

```
direccion
CABA            10
Buenos Aires     5
Name: count, dtype: int64
```

## Trabajando con archivos CSV

Una de las funcionalidades más útiles de Pandas es la capacidad de leer y escribir archivos CSV de forma muy sencilla. Como ya charlamos en la unidad 5, los archivos CSV (Comma Separated Values) son archivos de texto donde los datos están organizados en filas y columnas, separados por comas o punto y coma.

### Leyendo un archivo CSV

Para leer un archivo CSV usamos la función `read_csv()`:

```python
import pandas as pd

# Leer un archivo CSV
df = pd.read_csv('datos.csv')
```

Si el archivo usa punto y coma (`;`) como separador en lugar de coma, podemos aclararlo:

```python
df = pd.read_csv('datos.csv', sep=';')
```

> **!** ¿Y no necesitamos cerrar el archivo?
>
> A diferencia de cuando trabajamos con archivos de texto usando `open()`, con Pandas **no necesitamos cerrar el archivo manualmente**. Esto es porque `read_csv()` lee todo el contenido del archivo de una sola vez, lo carga en memoria como un DataFrame, y automáticamente cierra el archivo. El DataFrame queda en memoria y podemos trabajar con él sin mantener el archivo abierto.

> **⚠** ¿Qué beneficio le ves hasta el momento al manejo de archivos con Pandas?
>
> ¿Qué te llama la atención respecto de lo que vimos en la unidad 5?

### Escribiendo un archivo CSV

Para guardar un DataFrame como archivo CSV usamos `to_csv()`:

```
# Guardar el DataFrame en un archivo CSV
df.to_csv('nuevo_archivo.csv', index=False)
```

El parámetro `index=False` evita que se guarde el índice del DataFrame como una columna adicional dentro del archivo, algo que muchas veces no nos aporta información y no es relevante como para guardarlo.

### Pandas vs. Manipulación manual de archivos

Existen pros y contras de la manipulación de archivos con Pandas, en comparación con una manipulación manual.

| Aspecto | Archivos con `open()` | Archivos con Pandas |
|---|---|---|
| **Memoria** | Podemos leer línea por línea (bajo consumo) | Carga todo en memoria (alto consumo para archivos grandes) |
| **Cerrar archivo** | Debemos cerrar manualmente o usar `with` | Se cierra automáticamente |
| **Facilidad de uso** | Requiere parsear manualmente cada línea | Funciones listas para usar |
| **Manipulación de datos** | Debemos programar cada operación | Operaciones predefinidas (filtrar, ordenar, agrupar) |
| **Cuándo usar** | Archivos muy grandes o formato no estándar | Análisis de datos tabulares |

Recomendación

- Usá **Pandas** cuando necesites analizar y manipular datos tabulares (CSV, Excel).
- Usá **`open()` con lectura línea por línea** cuando el archivo sea muy grande y no entre en memoria, o cuando necesites un control más fino sobre el procesamiento.

Una vez creado nuestro "DataFrame" (en unos momentos definiremos qué es un DataFrame) a partir de un archivo CSV con Pandas, podemos simplemente manipularlo con la biblioteca.

## Manejo de errores en Pandas

Al trabajar con archivos CSV y DataFrames, pueden ocurrir varios tipos de errores. Es importante saber cómo manejarlos:

#### Archivo no encontrado

```
import pandas as pd

try:
    df = pd.read_csv('archivo_que_no_existe.csv')
except FileNotFoundError:
    print("El archivo no existe")
```

#### Columna no encontrada (KeyError)

```
try:
    valores = df['columna_inexistente']
except KeyError:
    print("La columna no existe en el DataFrame")
```

#### Operaciones con tipos incompatibles (TypeError)

```
# Si intentamos sumar una columna de texto con un número
try:
    resultado = df['nombre'] + 5
except TypeError:
    print("No se puede realizar esta operación con estos tipos de datos")
```

### Conclusiones

Pandas nos permite trabajar con datos de una manera muy sencilla y eficiente. Nos permite importar datos de distintas fuentes, limpiarlos, transformarlos y analizarlos. Además, nos permite visualizar los datos de una manera muy amigable, lo que nos va a permitir entender mejor los datos con los que estamos trabajando.

### Ejemplo integrador: Filtrar notas de un alumno

Veamos un ejercicio completo donde aplicamos lo aprendido de Pandas para resolver un problema real.

### Enunciado

Escribir un programa que reciba por parámetro:

1. La ruta de un archivo CSV con datos de alumnos (formato: `nombre;dni;materia;nota`)
2. Un número de DNI

El programa debe copiar todas las líneas correspondientes al alumno con ese DNI a un nuevo archivo llamado `<dni>.csv`.

Por ejemplo, si el archivo `alumnos.csv` contiene:

```
nombre;dni;materia;nota
Juan Pérez;12345678;Matemática;8
María García;23456789;Física;9
Juan Pérez;12345678;Física;7
Ana López;34567890;Matemática;10
Juan Pérez;12345678;Química;6
```

Y ejecutamos el programa con DNI `12345678`, se debe crear un archivo `12345678.csv` con:

```
nombre;dni;materia;nota
Juan Pérez;12345678;Matemática;8
Juan Pérez;12345678;Física;7
Juan Pérez;12345678;Química;6
```

### Resolución paso a paso

```python
import pandas as pd

def copiar_notas_alumno(archivo_entrada, dni_buscado):
    # Paso 1: Leer el archivo CSV con Pandas
    # Usamos sep=';' porque el separador es punto y coma
    df = pd.read_csv(archivo_entrada, sep=';')

    # Paso 2: Filtrar las filas donde el DNI coincide
    # Esto nos devuelve un nuevo DataFrame solo con las filas del alumno buscado
    df_filtrado = df[df['dni'] == dni_buscado]

    # Paso 3: Guardar el resultado en un nuevo archivo CSV
    # El nombre del archivo será el DNI con extensión .csv
```

```
    ruta_salida = f"{dni_buscado}.csv"
    df_filtrado.to_csv(ruta_salida, sep=';', index=False) # Esto es lo mismo que usar open(ru
```

Ejemplo de uso:

```
filtrar_notas_alumno('alumnos.csv', 12345678)
```

### Explicación de cada paso

#### Paso 1: Leer el archivo CSV

```
df = pd.read_csv(archivo_entrada, sep=';')
```

- `pd.read_csv()` lee el archivo y lo convierte en un DataFrame
- `sep=';'` indica que el separador de columnas es el punto y coma

#### Paso 2: Filtrar por DNI

```
df_filtrado = df[df['dni'] == dni_buscado]
```

Esta línea hace lo siguiente:

1. `df['dni']` accede a la columna `dni` del DataFrame
2. `df['dni'] == dni_buscado` compara cada valor con el DNI buscado, devolviendo `True` o `False` para cada fila
3. `df[...]` filtra el DataFrame, quedándose solo con las filas donde la condición es `True`

El resultado es un nuevo DataFrame con todas las filas del alumno.

#### Paso 3: Guardar en un nuevo archivo

```
ruta_salida = f"{dni_buscado}.csv"
df_filtrado.to_csv(ruta_salida, sep=';', index=False)
```

- Construimos el nombre/ruta del archivo de salida usando el DNI
- `to_csv()` guarda el DataFrame como archivo CSV
- `sep=';'` mantiene el mismo separador que el archivo original
- `index=False` evita que se agregue una columna extra con los índices

En este caso, usar `to_csv` nos ahorra usar `open(ruta_salida, "w")` para crear el archivo, y luego `writelines` para escribir en el mismo.

Ventajas de usar Pandas para este ejercicio

Comparado con la manipulación manual de archivos (usando `open()` y recorriendo línea por línea), con Pandas:

- No necesitamos parsear manualmente cada línea ni separar por `;`
- El filtrado se hace en una sola línea de código
- La escritura del archivo mantiene automáticamente el formato correcto
- El código es más legible y fácil de mantener

## NumPy

NumPy es una biblioteca de código abierto muy utilizada en el campo de la ciencia y la ingeniería. Permite trabajar con datos numéricos, matrices multidimensionales, funciones matemáticas y estadísticas avanzadas.

Como ya se mencionó anteriormente, para utilizarse se debe instalar e importar. Por convención, se suele importar como:

```
import numpy as np
```

NumPy incorpora una estructura de datos propia llamados **arrays** que es similar a la lista de Python, pero puede almacenar y operar con datos de manera mucho más eficiente: **el procesamiento de los arrays es hasta 50 veces más rápido**. Esta diferencia de velocidad se debe, en parte, a que **los arrays contienen datos homogéneos**, a diferencia de las listas que pueden contener distintos tipos de datos dentro.

### Arrays

Un **array** es un conjunto de elementos del mismo tipo, donde cada uno de ellos posee una posición y esta es única para cada elemento. Como dijimos arriba, en Python vimos las listas, que es lo más parecido a un Array.

Analicemos el siguiente ejemplo: si pensamos en una matriz, lo primero que nos viene a la mente es una tabla con valores ordenados en filas y columnas, donde una fila es la línea horizontal y una columna es la vertical. Es decir, una matriz es un conjunto de elementos que posee una posición o índice determinado por la fila y la columna, por lo que sería un array.

En este capítulo se trabajará principalmente con vectores y matrices, ya que consideramos que les será útil para aplicar los conocimientos de Numpy en otras materias.

### Creación de un Array

Un array se crea usando la función `array()` a partir de listas o tuplas. Por ejemplo:

```
a = np.array([1, 2, 3])
print(a)
```

```
[1 2 3]
```

También, se pueden crear arrays particulares, constituidos por ceros con `zeros()` o por unos con `ones()`:

```
# Creo un array de ceros con dos elementos
a_ceros = np.zeros(2)
print(a_ceros)
```

```
[0. 0.]
```

```
# Creo un array de unos con dos elementos
a_unos = np.ones(2)
print(a_unos)
```

```
[1. 1.]
```

Además, se pueden crear arrays con un rango de números, utilizando `arange()` o `linspace()`:

```
# Creo un array con un rango que empieza en 2 hasta 9 y va de 2 en 2.
a_rango = np.arange(2, 9, 2)
print(a_rango)
```

```
[2 4 6 8]
```

```
# Creo un array con un rango formado por 4 números
# que empieza en 2 hasta 10 (incluidos).
a_rango_2 = np.linspace(2, 10, num=4)
print(a_rango_2)
```

```
[ 2.          4.66666667  7.33333333 10.        ]
```

Esto es muy parecido a los rangos que ya vimos en Python, con la sutil diferencia de que el `final` del rango en este caso **sí se incluye**.

Finalmente, para crear arrays de más dimensiones, se utilizan varias listas:

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])

print(matriz)
```

```
[[1 2 3]
 [4 5 6]]
```

### Atributos de un array

#### Dimensión

Para caracterizar un array es necesario conocer sus dimensiones, utilizando `ndim`. De esta forma, se puede confirmar que el array llamado **matriz**, definido anteriormente, es bidimensional:

```
# Número de ejes o dimensiones de la matriz
matriz.ndim
```

```
2
```

#### Forma

Otra característica de interés es su forma o `shape`: para las matrices bidimensionales, se muestra una tupla (n, m) con el número de filas n y de columnas m:

```
# (n = filas, m = columnas)
matriz.shape
```

```
(2, 3)
```

#### Tamaño

El tamaño de un array es el número total de elementos que contiene, y se obtiene con `size`:

```
# Número total de elementos de la matriz: 2 filas x 3 columnas = 6 elementos
matriz.size
```

```
6
```

### Posiciones

Al elemento de una matriz A que se encuentra en la **fila i-ésima y la columna j-ésima** se llama $a_{ij}$. Así, para acceder a un elemento de un array se debe indicar primero la posición de la fila y luego, de la columna:

```
print('Elemento de la primera fila y segunda columna: ', matriz[0, 1])
```

```
Elemento de la primera fila y segunda columna:  2
```

Nótese la diferencia con las matrices (listas de listas) de Python, donde se accedía a un elemento por separado, primero a la fila y luego a la columna: `matriz[0][1]`.

También se puede elegir un rango de elementos en una fila o columna particular:

```
print('Los elementos de la primera fila, columnas 0 y 1: ', matriz[0, 0:2])
```

```
Los elementos de la primera fila, columnas 0 y 1:  [1 2]
```

```
print('Los elementos de la segunda columna, filas 0 y 1: ', matriz[0:2, 1])
```

```
Los elementos de la segunda columna, filas 0 y 1:  [2 5]
```

### Modificar arrays

De forma similar a lo aprendido con las listas de Python, se pueden modificar los arrays utilizando ciertas funciones. Para entender y aplicar las mismas, definamos un vector llamado a:

```
a = np.array([2, 1, 5, 3, 7, 4, 6, 8])

print(a)
```

```
[2 1 5 3 7 4 6 8]
```

### Insert

Se puede insertar una fila (`axis = 0`) o una columna (`axis = 1`) en una determinada posición. Para este ejemplo, primero creamos una matriz:

```
matriz_ejemplo = np.array([[2, 1, 5, 3], [7, 4, 6, 8]])
print(matriz_ejemplo)
```

```
[[2 1 5 3]
 [7 4 6 8]]
```

```
# Agregar fila de cincos en posición 1:
print(np.insert(matriz_ejemplo, 1, 5, axis=0))
```

```
[[2 1 5 3]
 [5 5 5 5]
 [7 4 6 8]]
```

A la función `insert()`, se le debe indicar:

- El array que se desea modificar
- La posición de la fila o columna que se desea agregar
- Los valores a insertar. **¡Ojo con las dimensiones!** Para el ejemplo anterior, matriz_ejemplo tenía 2 filas, por lo que se debe agregar una columna con 2 elementos o una fila con 4.
- El eje que se agrega: una fila (axis = 0) o una columna (axis = 1)

```
# Agregar columna de cincos en posición 1:
print(np.insert(matriz_ejemplo, 1, 5, axis=1))
```

```
[[2 5 1 5 3]
 [7 5 4 6 8]]
```

O lo que es equivalente:

```
# Agregar columna de cincos en posición 1:
print(np.insert(matriz_ejemplo, 1, [5, 5], axis=1))
```

```
[[2 5 1 5 3]
 [7 5 4 6 8]]
```

### Append y Delete

También podríamos agregar una fila o una columna utilizando `append()` al final, como ocurría con las listas:

```
# Agregar una última fila
a_modificada = np.append(matriz_ejemplo, [[1, 2, 3, 4]], axis=0)
print(a_modificada)
```

```
[[2 1 5 3]
 [7 4 6 8]
 [1 2 3 4]]
```

O eliminarlas con `delete()`

```
# Eliminar la fila de la posición 2.
print(np.delete(a_modificada, 2, axis=0))
```

```
[[2 1 5 3]
 [7 4 6 8]]
```

### Concatenate y Sort

Finalmente, podemos concatenar arrays, como los siguientes:

```
a = np.array([2, 1, 5, 3])
b = np.array([7, 4, 6, 8])

# Concatenar a y b:
c = np.concatenate((a, b))
print(c)
```

```
[2 1 5 3 7 4 6 8]
```

Y ordenar los elementos de un array como numérico o alfabético, ascendente o descendente.

```
print(np.sort(c))
```

```
[1 2 3 4 5 6 7 8]
```

### Operaciones aritméticas utilizando array

Como se ha mencionado anteriormente, Numpy tiene un gran potencial para realizar operaciones, muy superior al de las listas de Python. Por ejemplo, si quisiéramos sumar dos listas de python necesitaríamos realizar un `for`:

```
# Definir listas
a = [2, 1, 5, 3]
b = [7, 4, 6, 8]
c = []

# Sumar el primer elemento de a con el primero de b, el segundo elemento de a con el segundo
for i in range(len(a)):
  c.append(a[i] + b[i])
print(c)
```

```
[9, 5, 11, 11]
```

Utilizando las funciones de Numpy, esto ya no es más necesario:

```
# add() para sumar elemento a elemento de a y b
c = np.add(a, b)
print(c)
```

```
[ 9  5 11 11]
```

También podemos realizar otras operaciones, como la resta, multiplicación y división. Usar un arreglo dentro de una ecuación nos devuelve otro arreglo, donde cada elemento es el resultado de aplicar la operación a los elementos correspondientes de los arreglos originales.

```
x = np.array([0,  1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y = 3 * x + 2
print(y)
```

```
[ 2  5  8 11 14 17 20 23 26 29 32]
```

De esta forma, podemos realizar operaciones aritméticas con arrays de Numpy de forma muy sencilla y rápida. Antes, en Python, para realizar estas operaciones debíamos recurrir a un ciclo `for` o a el uso de `map`. Numpy se ocupa de ahorrarnos el trabajo y calcular, para cada elemento del array, el resultado.

Además, tenemos operaciones básicas que vienen predefinidas por Numpy. Las vamos a ver a continuación.

#### Operaciones básicas:

A continuación se muestra una lista con las operaciones básicas junto con sus operadores asociados, funciones y ejemplos.

| Operación | Operador | Función |
|---|---|---|
| Suma | `+` | `add()` |
| Resta | `-` | `subtract()` |
| Multiplicación | `*` | `multiply()` |
| División | `/` | `divide()` |
| Potencia | `**` | `power()` |

Definimos los vectores a y b con los que operaremos y veremos ejemplos:

```
a = np.array([1, 3, 5, 7])
b = np.array([1, 1, 2, 2])
```

- Suma:

```
resultado_1 = a + b
print("Suma usando +:", resultado_1)

resultado_2 = np.add(a, b)
print("Suma usando add():", resultado_2)
```

```
Suma usando +: [2 4 7 9]
Suma usando add(): [2 4 7 9]
```

- Resta:

```
resultado_1 = a - b
print("Resta usando -:", resultado_1)

resultado_2 = np.subtract(a, b)
print("Resta usando subtract():", resultado_2)
```

```
Resta usando -: [0 2 3 5]
Resta usando subtract(): [0 2 3 5]
```

- Multiplicación:

```
resultado_1 = a * b
print("Multiplicación usando *:", resultado_1)

resultado_2 = np.multiply(a, b)
print("Multiplicación usando multiply():", resultado_2)
```

```
Multiplicación usando *: [ 1  3 10 14]
Multiplicación usando multiply(): [ 1  3 10 14]
```

- División:

```
resultado_1 = a / b
print("División usando /:", resultado_1)

resultado_2 = np.divide(a, b)
print("División usando divide():", resultado_2)
```

```
División usando /: [1.  3.  2.5 3.5]
División usando divide(): [1.  3.  2.5 3.5]
```

- Potencia:

```
resultado_1 = a ** b
print("Potencia usando **:", resultado_1)

resultado_2 = np.power(a, b)
print("Potencia usando power():", resultado_2)
```

```
Potencia usando **: [ 1  3 25 49]
Potencia usando power(): [ 1  3 25 49]
```

> **Nota**
>
> Note que si quisiéramos operar con un vector `b` de elementos iguales, podríamos utilizar un escalar.

```
b = np.array([2, 2, 2, 2])

resultado_1 = a * b
print("Usando un vector b = [2, 2, 2, 2]:", resultado_1)

resultado_2 = a * 2
print("Usando un escalar b = 2:", resultado_2)
```

```
Usando un vector b = [2, 2, 2, 2]: [ 2  6 10 14]
Usando un escalar b = 2: [ 2  6 10 14]
```

**Logaritmo:**

NumPy provee funciones para los logaritmos de base 2, 10 y e:

| Base | Función |
|---|---|
| 2 | `log2()` |
| 10 | `log10()` |
| e | `log()` |

Por ejemplo:

```
# Ejemplo log2()
print("Logaritmo base 2:", np.log2([2, 4, 8, 16]))
# Ejemplo log10()
print("Logaritmo base 10:", np.log10([10, 100, 1000, 10000]))
# Ejemplo log()
print("Logaritmo base e:", np.log([1, np.e, np.e**2]))
```

```
Logaritmo base 2: [1. 2. 3. 4.]
Logaritmo base 10: [1. 2. 3. 4.]
Logaritmo base e: [0. 1. 2.]
```

> **Nota**
>
> Note que el número de Euler o número e es una constante incluida en NumPy como: `np.e`

```
np.e
```

```
2.718281828459045
```

> **Funciones trigonométricas (opcional)**
>
> Esta parte del apunte es opcional, es decir, no se evalúa en los exámenes. A continuación, una lista con las funciones trigonométricas más utilizadas, que toman los valores en radianes:

| Función trigonométrica | Función |
|---|---|
| seno | `sin()` |
| coseno | `cos()` |
| tangente | `tan()` |
| arcoseno | `arcsin()` |
| arcocoseno | `arccos()` |
| arcotangente | `arctan()` |

Por ejemplo:

```
# Ejemplo de seno
print("Seno de   / 2:", np.sin(np.pi / 2))

# Ejemplo de arcoseno
print(np.arcsin(1))
```

```
Seno de   / 2: 1.0
1.5707963267948966
```

```
# Ejemplo de coseno
print("Coseno de  :", np.cos(np.pi))

# Ejemplo de arcocoseno
print("Arcoseno de -1:", np.arccos(-1))
```

```
Coseno de  : -1.0
Arcoseno de -1: 3.141592653589793
```

```
# Ejemplo de tangente:
print("Tangente de 0:", np.tan(0))

# Ejemplo de arcotangente:
print("Arcotangente de 0:", np.arctan(0))
```

```
Tangente de 0: 0.0
Arcotangente de 0: 0.0
```

**Note**

Note que el número  es una constante incluida en NumPy como: `np.pi`

```
np.pi
```

```
3.141592653589793
```

Para convertir los radianes a grados y viceversa, se utiliza `deg2rad()` y `rad2deg()` respectivamente:

```
print("De grados [90, 180, 270, 360] a radianes:",
      np.deg2rad([90, 180, 270, 360]))

print("De radianes [ /2,  , 1.5* , 2* ] a grados:",
      np.rad2deg([np.pi/2, np.pi, 1.5*np.pi, 2*np.pi]))
```

```
De grados [90, 180, 270, 360] a radianes: [1.57079633 3.14159265 4.71238898 6.28318531]
De radianes [ /2,  , 1.5* , 2* ] a grados: [ 90. 180. 270. 360.]
```

**Operaciones con matrices:**

A continuación, una lista con las operaciones que les pueden ser de interés mientras estudian álgebra matricial:

| Función | Descripción | Comentario |
|---|---|---|
| `dot()` | Producto escalar | Se utiliza para obtener el producto escalar entre dos vectores. El resultado es un número. |
| `dot()` | Producto vectorial | También se utiliza para multiplicar matrices. El resultado es una matriz |
| `transpose()` | Traspuesta | Cambia las filas por las columnas y viceversa |
| `linalg.inv()` | Inversa | Inversa de una matriz |
| `linalg.det()` | Determinante | Determinante de una matriz |
| `eye()` | Matriz identidad | Matriz cuadrada con unos en la diagonal principal y ceros en el resto |

Definimos los arreglos 1 y 2, y matrices 1 y 2 con los que operaremos y veremos ejemplos:

```
# Crear arreglos
arreglo_1 = np.array([1, 2])
arreglo_2 = np.array([3, 4])

# Crear matrices
matriz_1 = np.array([[1, 3], [5, 7]])
matriz_2 = np.array([[2, 6], [4, 8]])
```

```
print("Producto escalar entre el array 1 y 2: \n", np.dot(arreglo_1, arreglo_2))
```

```
Producto escalar entre el array 1 y 2:
 11
```

```
print("Producto vectorial entre la matriz 1 y 2: \n", np.dot(matriz_1, matriz_2))
```

```
Producto vectorial entre la matriz 1 y 2:
 [[14 30]
 [38 86]]
```

```
print("Traspuesta de la matriz 1: \n", np.transpose(matriz_1))
```

```
Traspuesta de la matriz 1:
 [[1 5]
 [3 7]]
```

```
print("Inversa de la matriz 1: \n", np.linalg.inv(matriz_1))
```

```
Inversa de la matriz 1:
 [[-0.875  0.375]
 [ 0.625 -0.125]]
```

```
print("Determinante de la matriz 1: \n", np.linalg.det(matriz_1))
```

```
Determinante de la matriz 1:
 -7.999999999999998
```

> **Nota**
>
> Note que así como existen constantes numéricas, existen las matrices particulares como las compuestas por ceros `np.zeros()`, por unos `np.ones()` y la matriz identidad `np.eyes`.

```
print("Matriz de identidad de 3x3: \n", np.eye(3))
```

```
Matriz de identidad de 3x3:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

**Más operaciones útiles:**

| Operaciones | Función | Descripción |
|---|---|---|
| Máximo | `max()` | Valor máximo del array o del eje indicado |
| Mínimo | `min()` | Valor mínimo del array o del eje indicado |
| Suma | `sum()` | Suma de todos los elementos o del eje indicado |
| Promedio | `mean()` | Promedio de todos los elementos o del eje indicado |

Utilizando la matriz **data** como ejemplo:

```
data = np.array([[1, 2], [5, 3], [4, 6]])
```

- Valor máximo

```
print("Valor máximo de todo el array: ", data.max())
print("Valores máximos de cada columna: ", data.max(axis=0))
```

```
Valor máximo de todo el array:  6
Valores máximos de cada columna:  [5 6]
```

- Valor mínimo

```
print("Valor mínimo de todo el array: ", data.min())
print("Valores mínimos de cada fila: ", data.min(axis=1))
```

```
Valor mínimo de todo el array:  1
Valores mínimos de cada fila:  [1 3 4]
```

- Suma de elementos:

```
print("Suma de todos los elementos del array: ", data.sum())
print("Suma de los elementos de cada fila: ", data.sum(axis=1))
```

```
Suma de todos los elementos del array:  21
Suma de los elementos de cada fila:  [ 3  8 10]
```

- Promedio:

```
print("Promedio de todos los elementos del array: ", data.mean())
print("Promedio de los elementos de cada columna: ", data.mean(axis=0))
```

```
Promedio de todos los elementos del array:  3.5
Promedio de los elementos de cada columna:  [3.33333333 3.66666667]
```

> **Nota**
>
> Numpy te va a ser muy útil cuando curses materias como Análisis Matemático, Álgebra, Física, Estadística, entre otras. Te va a permitir realizar operaciones de manera rápida y eficiente, y te va a ayudar a entender mejor los conceptos.

## Matplotlib

> **Nota**
>
> Para Matplotlib también vamos a usar Google Colab

Matplotlib es probablemente la biblioteca de Python más usada para crear gráficos, también llamados **plots**. Esta biblioteca provee una forma rápida de graficar datos en varios formatos de alta calidad que pueden ser compartidos y/o publicados.

El primer paso es importar la biblioteca. Por convención:

```
import matplotlib.pyplot as plt
```

### Introducción

Para crear un gráfico con matplotlib, se deben seguir los siguientes pasos:

1. **Crear la figura** que contendrá el gráfico, utilizando las funciones `subplots()` o `figure()`. Se recomienda la primera, es la que va a utilizarse en la materia.
2. **Graficar los datos**, utilizando distintas funciones dependiendo del tipo de gráfico que se desea realizar:

| Función | Tipo de Gráfico |
|---|---|

| Función | Tipo de Gráfico |
|---|---|
| `plot()` | Gráfico de línea |
| `scatter()` | Gráfico de puntos |
| `bar()` | Gráfico de barras verticales |
| `barh()` | Gráfico de barras horizontales |
| `pie()` | Gráfico de torta |

3. **Personalizar el gráfico**. Este paso es muy recomendado para lograr un mejor entendimiento de la visualización. Esto incluye agregar etiquetas a los ejes, títulos, leyendas, etc. También podemos modificar el aspecto de las líneas, los puntos, los colores, y más (esto se deja en el final del apunte, y es opcional).
4. **Mostrar el gráfico**, utilizando la función `show()`

```
plt.show()
```

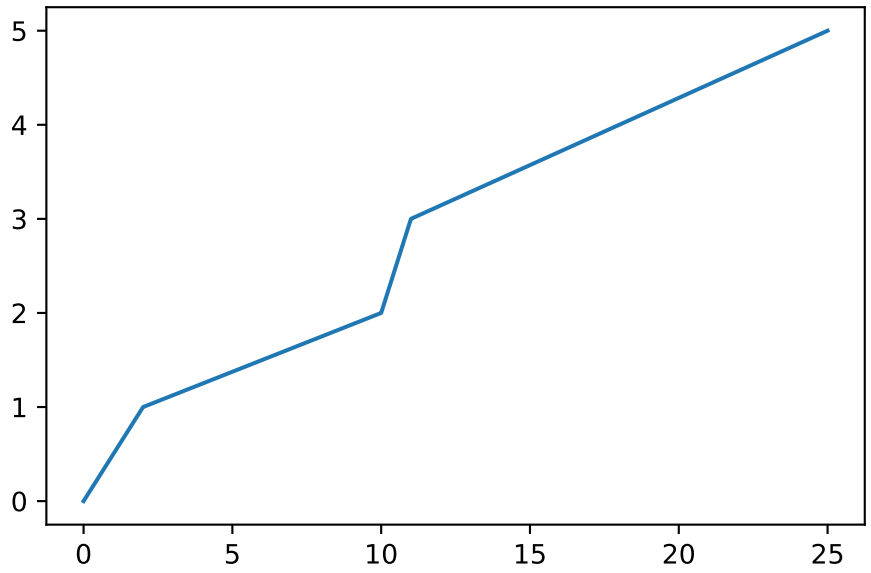
## Creando una figura

Para crear una figura, la función de gráfico recibe los datos a graficar y los parámetros necesarios para personalizarlo.

### Gráfico de línea

El gráfico de línea permite visualizar cambios en los valores lo largo de un rango continuo (tendencias), como puede ser el tiempo o la distancia.
Para crear un gráfico de línea, se utiliza la función `plot()`.

```
# Grafico elemental
x = [0,2,10,11,18,25]
y = [0,1,2,3,4,5]

fig, ax = plt.subplots()

# Gráfico de línea
ax.plot(x, y)
plt.show()
```

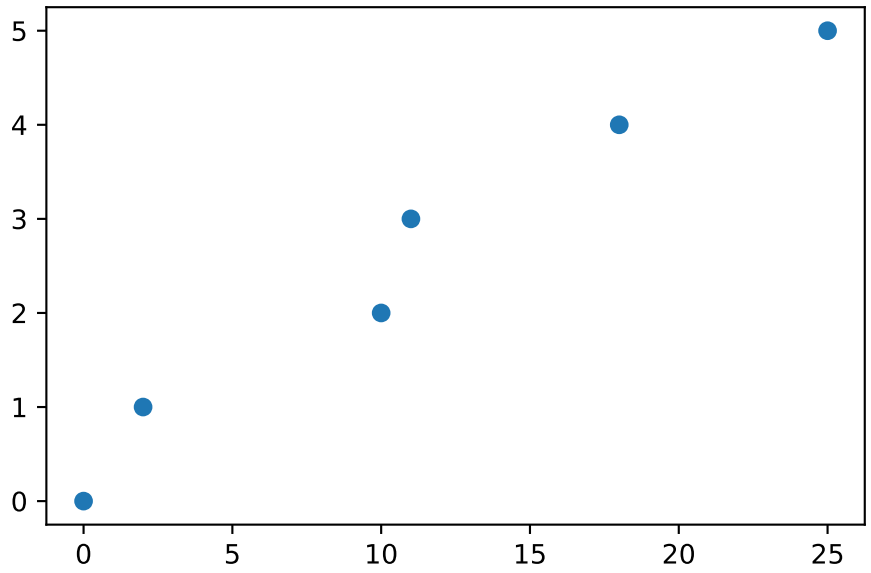



En este caso, se creó un gráfico de línea con los valores de **x** e **y**, tal que los puntos (0,0), (2,1), (10,2), (11,3), (18,4) y (25,5) están unidos por una línea recta.

### Gráfico de puntos

El gráfico de dispersión o puntos permite visualizar la relación entre las variables.
Para crearlo, se utiliza la función `scatter()`:

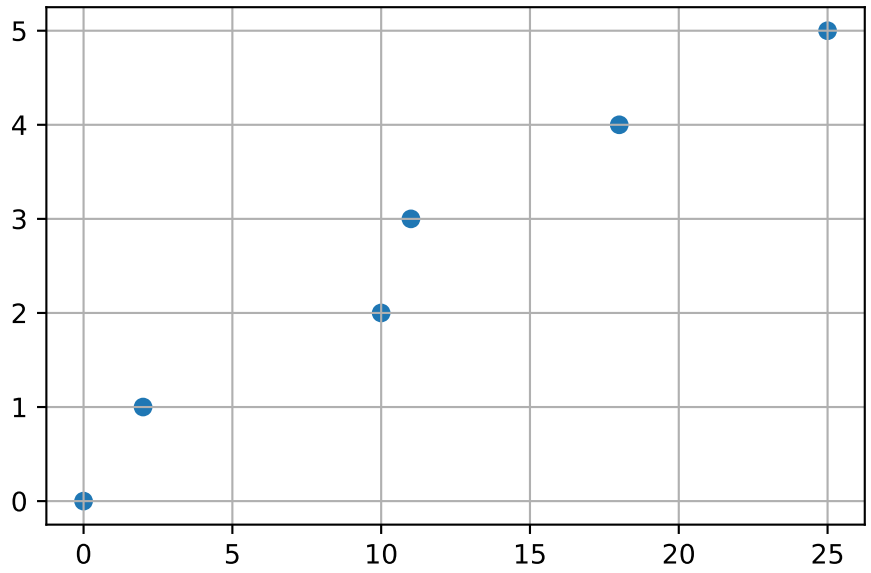
```
# Gráfico de puntos
fig2, ax2 = plt.subplots()
ax2.scatter(x, y)
plt.show()
```




**!** Grillas

¿Ves cómo en el gráfico de puntos quizás no se entiende bien la ubicación de cada punto? Esto es porque no tenemos una guía que nos ayude. Para eso, vamos a agregar una grilla. La grilla se puede agregar con al función `grid()`, y es una buena forma de darle legibilidad a un gráfico como puede ser el de línea y el de puntos.

```
# Gráfico de puntos
fig, ax = plt.subplots()
ax.scatter(x, y)
ax.grid()
plt.show()
```

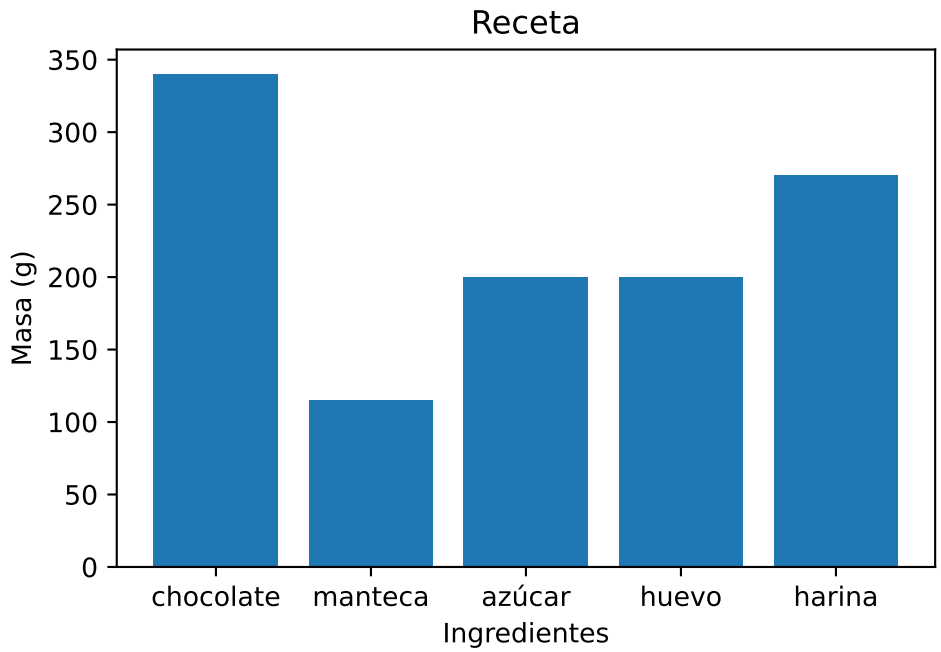



### Gráficos de Barras

El gráfico de barras permite visualizar proporciones, comparando dos o más valores entre sí. Para crearlo, se utiliza la función `bar()`, la cual primero recibe, en primer lugar, las etiquetas de las barras que se van a mostrar y en segundo lugar, la altura correspondiente a cada una de estas barras.

En el caso de este tipo de gráfico, **no hace falta que los valores de las etiquetas sean numéricos**.

```
peso = [340, 115, 200, 200, 270]
ingredientes = ['chocolate', 'manteca', 'azúcar', 'huevo', 'harina']

fig, ax = plt.subplots()

ax.bar(ingredientes, peso) # Acá podría usarse también barh

ax.set_xlabel('Ingredientes')
ax.set_ylabel('Masa (g)')

ax.set_title("Receta")

plt.show()
```

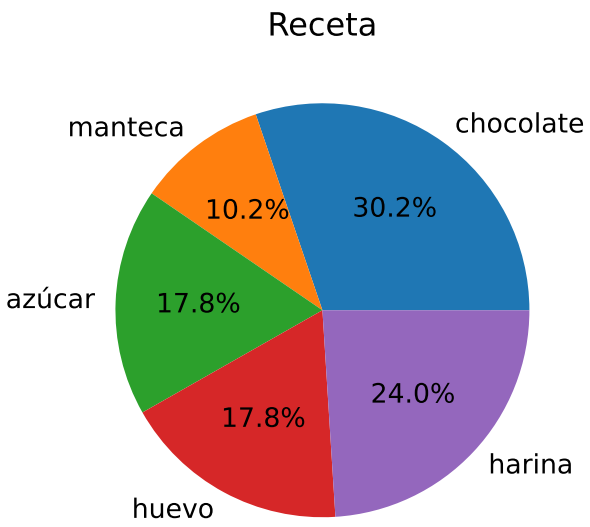



### Gráfico de torta

Finalmente, tenemos el gráfico de torta. El gráfico de torta, como el de barras, permite visualizar y comparar proporciones pero de manera circular y como partes de un todo.

Para crearlo, se utiliza la función `pie()`, que recibe los valores de las porciones y las etiquetas de cada una. Para las etiquetas, se debe indicar la variable `label`, y si además queremos que se muestren los porcentajes, se puede utilizar `autopct='%1.1f%%'`. `'%1.1f%%'` significa que se mostrará un decimal y el símbolo de porcentaje.

```
peso = [340, 115, 200, 200, 270]
ingredientes = ['chocolate', 'manteca', 'azúcar', 'huevo', 'harina']

fig, ax = plt.subplots()

ax.pie(peso, labels= ingredientes, autopct='%1.1f%%')

ax.set_title("Receta")

plt.show()
```

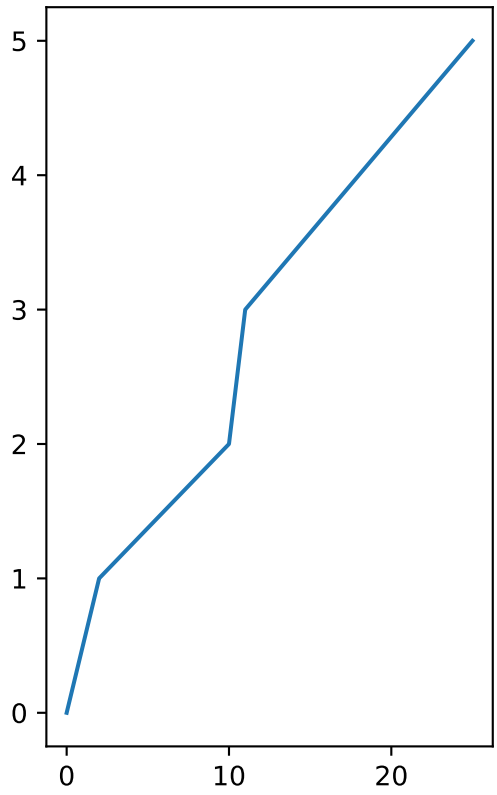



### Cambio de Tamaño

Podemos establecer el tamaño de la figura con el parámetro `figsize` dentro de la función `subplots()`. Este parámetro recibe una tupla con dos valores: el ancho y el alto de la figura, en pulgadas.

Veamos este ejemplo: **x** es el tiempo medido en minutos e **y** una distancia en metros, entonces:

```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)

fig, ax = plt.subplots(figsize=(3, 5))

ax.plot(x, y)

plt.show()
```




### Títulos

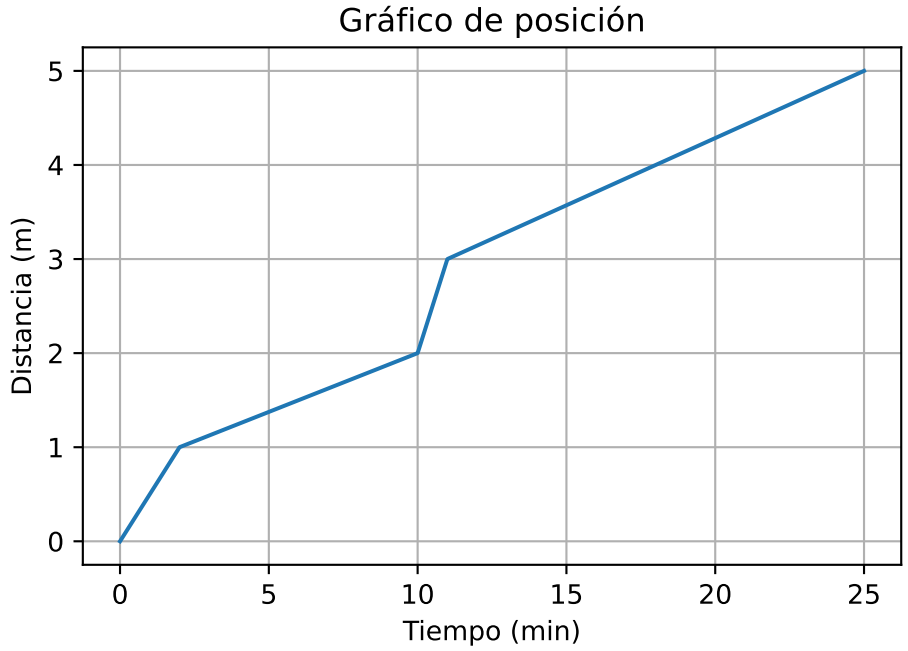
Así como los nombres de las variables en nuestro código, es importante que nuestro gráfico tenga un Título descriptivo que nos ayude a entender qué es lo que estamos viendo. Lo mismo pasa con los ejes: queremos poder entender qué representa cada uno.

Para setear un título, vamos a usar la función `set_title()`. Para los ejes, usaremos `set_xlabel()` y `set_ylabel()`. Cada una recibe un string que se usará como etiqueta del eje X, etiqueta del eje Y o título, respectivamente.

Siguiendo el ejemplo anterior, vamos a agregar títulos a los ejes y al gráfico:

```
x = [0,2,10,11,18,25]   # Tiempo (min)
y = [0,1,2,3,4,5]       # Distancia (m)

fig, ax = plt.subplots()

ax.plot(x, y)
```

```
# Mostrar el título del gráfico
ax.set_title("Gráfico de posición")

# Mostrar el título de los ejes
ax.set_xlabel('Tiempo (min)')
ax.set_ylabel('Distancia (m)')


ax.grid()
plt.show()
```

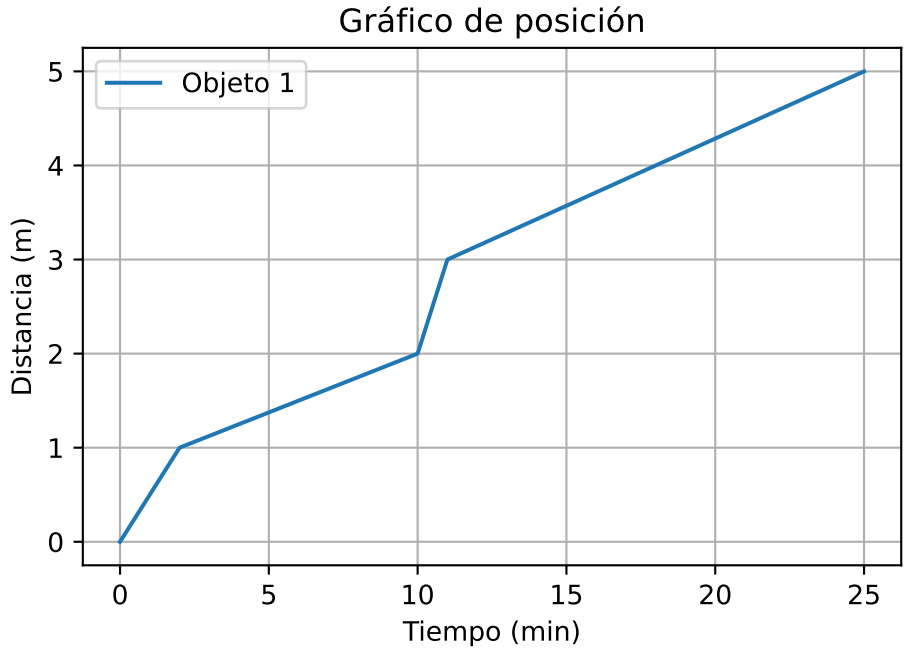



### Referencias

El gráfico con el que estamos trabajando sólo tiene una línea, pero si contara con más de una (lo vamos a ver más adelante en este apunte), el uso de referencias sería muy importante para lograr el entendimiento del mismo. Para rotular las líneas, dentro de `plot()` se debe definir la referencia como `label`. Luego se coloca `legend()`

```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)
```

```python
fig, ax = plt.subplots()

ax.plot(x, y, label='Objeto 1') # Agregar el label

ax.set_title("Gráfico de posición")
ax.set_xlabel('Tiempo (min)')
ax.set_ylabel('Distancia (m)')

# Agregar la refencia
ax.legend()

ax.grid()
plt.show()
```



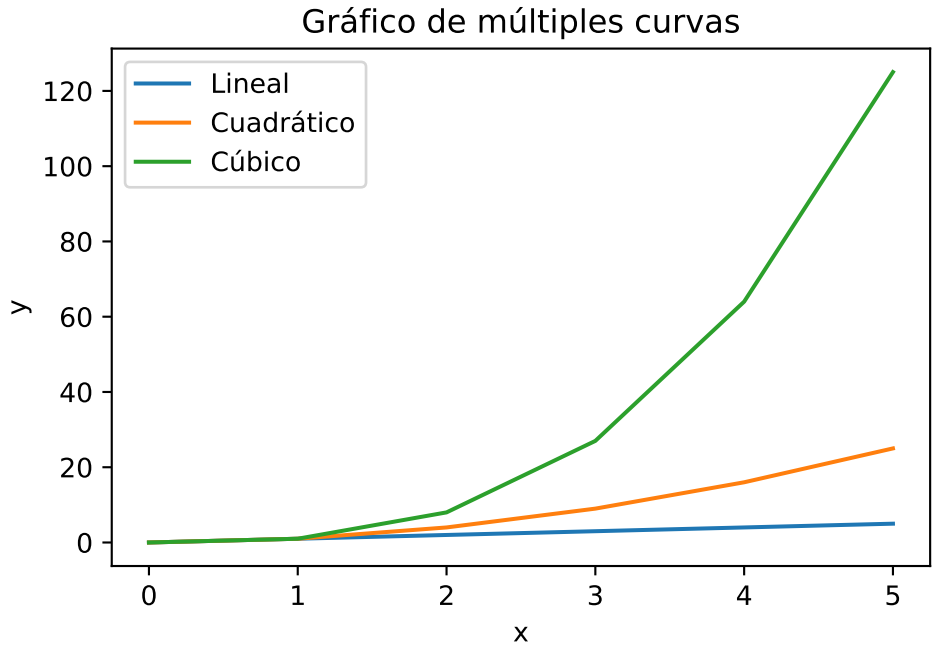

### Gráficos múltiples

En los casos anteriores, creamos siempre un sólo gráfico con una curva, en una figura. Pero **También podríamos graficar varias curvas en un mismo gráfico**.

Para esto vamos a seguir el mismo procedimiento que antes, pero vamos a agregar más de un `plot()` al mismo axes. También vamos a darles un label a cada uno, y luego vamos a agregar

una leyenda. Automáticamente, al estar en el mismo axes, matplotlib los va a agregar al mismo gráfico con distintos colores.

```
# Valores que se desean graficar
x = [0, 1, 2, 3, 4, 5]
y_linear = [0, 1, 2, 3, 4, 5]
y_quadratic = [0, 1, 4, 9, 16, 25]
y_cubic = [0, 1, 8, 27, 64, 125]

fig, ax = plt.subplots()

# Usamos distintos tipos de y, con distintas labels
ax.plot(x, y_linear, label='Lineal')
ax.plot(x, y_quadratic, label='Cuadrático')
ax.plot(x, y_cubic, label='Cúbico')

ax.set_title("Gráfico de múltiples curvas")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()

plt.show()
```



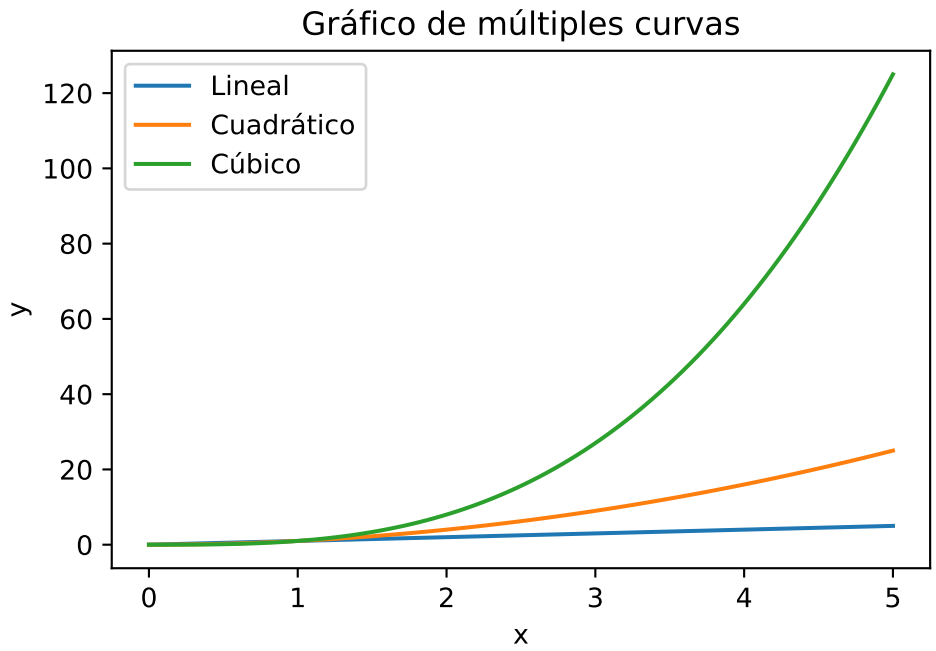
Note que se agregan nuevos datos al mismo axes, por lo que siempre usamos `plot()` pero con distintos valores de `y`. Asimismo, se estableció un tamaño de la figura con `figsize=(width, height)`

## Uniendo Bibliotecas

### Matplotlib y Numpy

Podemos usar arrays de numpy para simplificarnos el trabajo.
Repetimos el gráfico anterior pero usando numpy, y vamos a ver que pudimos obtener muchos más valores sin tener que calcularlos nosotros y escribirlos en una lista de Python.

```
import numpy as np

x = np.linspace(0, 5, 100) # Creamos un array de 100 valores entre 0 y 5
y_linear = x # usamos x
y_quadratic = x**2 # usamos x al cuadrado
y_cubic = x**3 # usamos x al cubo

fig, ax = plt.subplots()

# Usamos distintos tipos de y
ax.plot(x, y_linear, label='Lineal')
ax.plot(x, y_quadratic, label='Cuadrático')
ax.plot(x, y_cubic, label='Cúbico')

ax.set_title("Gráfico de múltiples curvas")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()

plt.show()
```




### Matplotlib y Pandas

También podemos usar columnas de Pandas como datos para crear un gráfico.

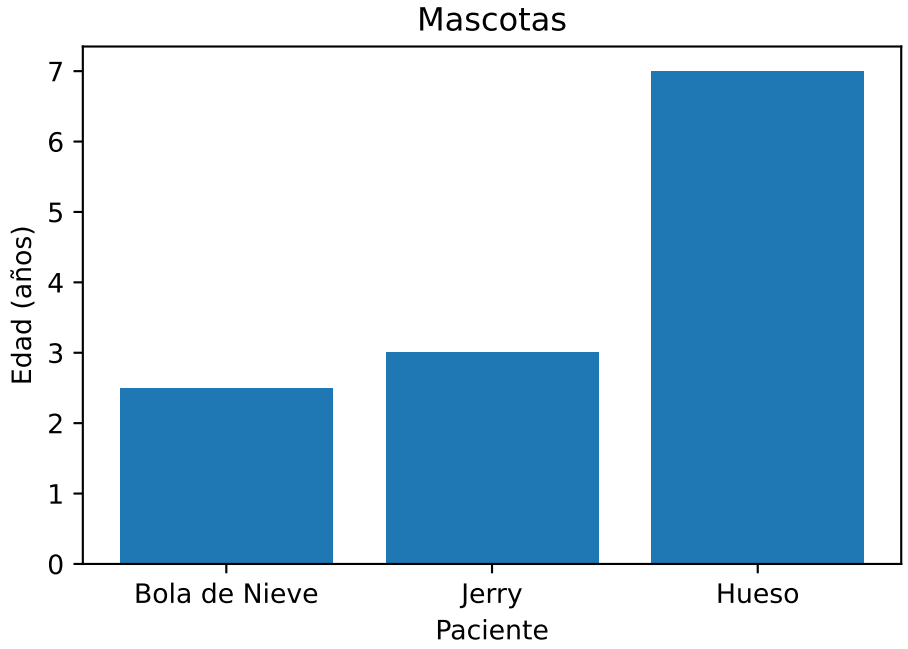
Supongamos el siguiente dataframe de los datos de las mascotas en una veterinaria:

```
data = { 'nombre': ['Bola de Nieve', 'Jerry', 'Hueso'],
         'especie': ['gato','chinchilla', 'perro'],
         'edad': [2.5, 3, 7],
         'visitas': [1, 3, 2],
         'prioridad': ['si', 'si', 'no']}

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

| | nombre | especie | edad | visitas | prioridad |
|---|---|---|---|---|---|
| 0 | Bola de Nieve | gato | 2.5 | 1 | si |
| 1 | Jerry | chinchilla | 3.0 | 3 | si |
| 2 | Hueso | perro | 7.0 | 2 | no |

Podemos ahora crear un gráfico usando las columnas del dataframe:

```python
# Determino las columnas del DataFrame que queremos graficar
x_values = df['nombre']
y_values = df['edad']

fig, ax = plt.subplots()

ax.bar(x_values, y_values)

ax.set_xlabel('Paciente')
ax.set_ylabel('Edad (años)')

ax.set_title("Mascotas")

plt.show()
```




**i** ¿Qué pasa si nuestros labels no llegan a verse? (opcional)

Puede pasar, sobre todo si tenemos muchos datos con nombres largos, que nuestros labels no lleguen a verse de forma correcta si los presentamos de forma horizontal.
Por ejemplo:
En esos casos, podemos pedirle a matplotlib que los presente de forma vertical, para que no se superpongan. Para eso, usamos `xticks()` y `rotation`: `plt.xticks(rotation=90)`.

El ángulo de rotación se mide en grados, por lo que `rotation=90` significa que se rotarán 90 grados. De esta forma, los labels se presentarán de forma vertical. Podríamos también usar otro ángulo, y se mostrarían los labels de forma inclinada.
Esta es la forma en que se visualizan los datos con los labels rotados:

### Bonus Track: Personalización (opcional)

> **i** ¿Qué significa opcional?
>
> Significa que esta parte del apunte no es obligatoria. Pero si quieren leerla, les va a permitir hacer gráficos más bonitos y personalizados. También les va a permitir llevarse el conocimiento de qué otras herramientas tienen disponibles, y volver a este apunte a buscar información en un futuro si es que la necesitan.

Esta imagen, fue obtenida de la referencia de matplotlib y resume de manera fácil y visual las modificaciones que podemos hacerla a las figuras creadas.

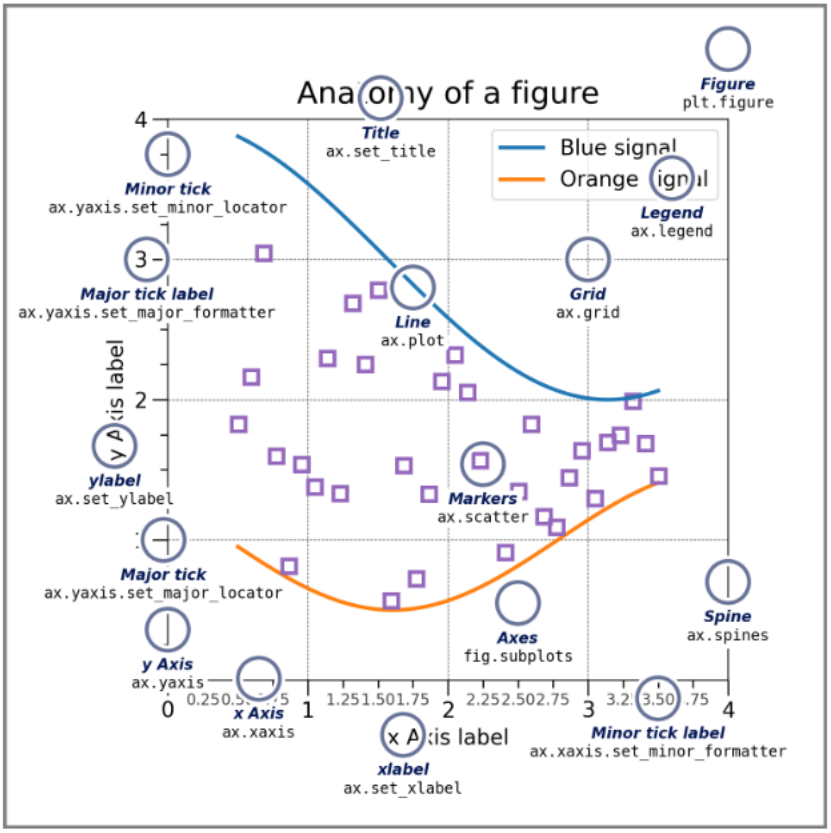



Figura 3: Partes de una Figura. Si querés conocer más detalle, podés ingresar a este link.

Como dijimos más arriba, las figuras pueden ser personalizadas de muchas maneras. Algunas son:

- Colores
- Estilos de línea
- Marcadores
- Grilla personalizada

¡y más!

### Girando los labels

No siempre nuestros labels van a tener todo el espacio disponible para mostrarse de forma correcta.

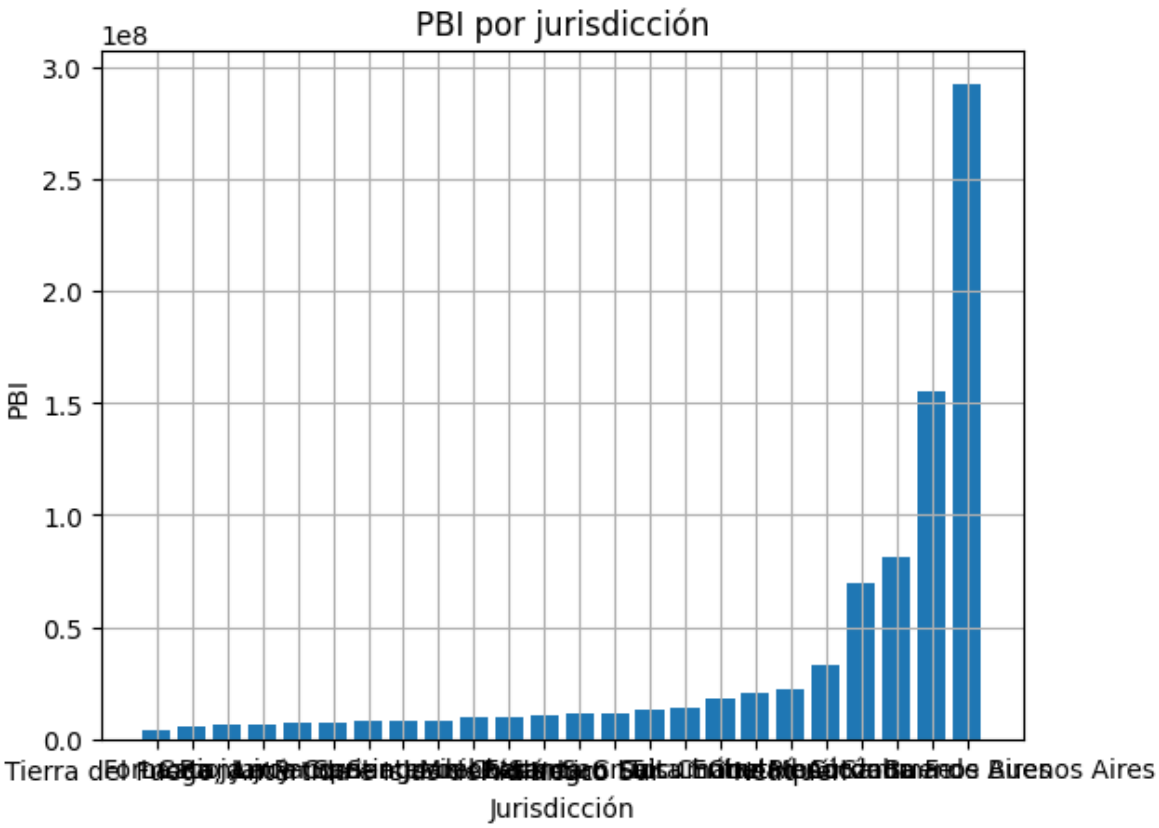



Figura 4: Los labels son muy grandes y no hay espacio para mostrarlos correctamente

Podemos darle un *ángulo* a los labels usando `xticks`: `plt.xticks(rotation=90)`. El valor que le pasemos indica el ángulo de rotación que le vamos a dar a los labels. De esta forma, no se pisan.

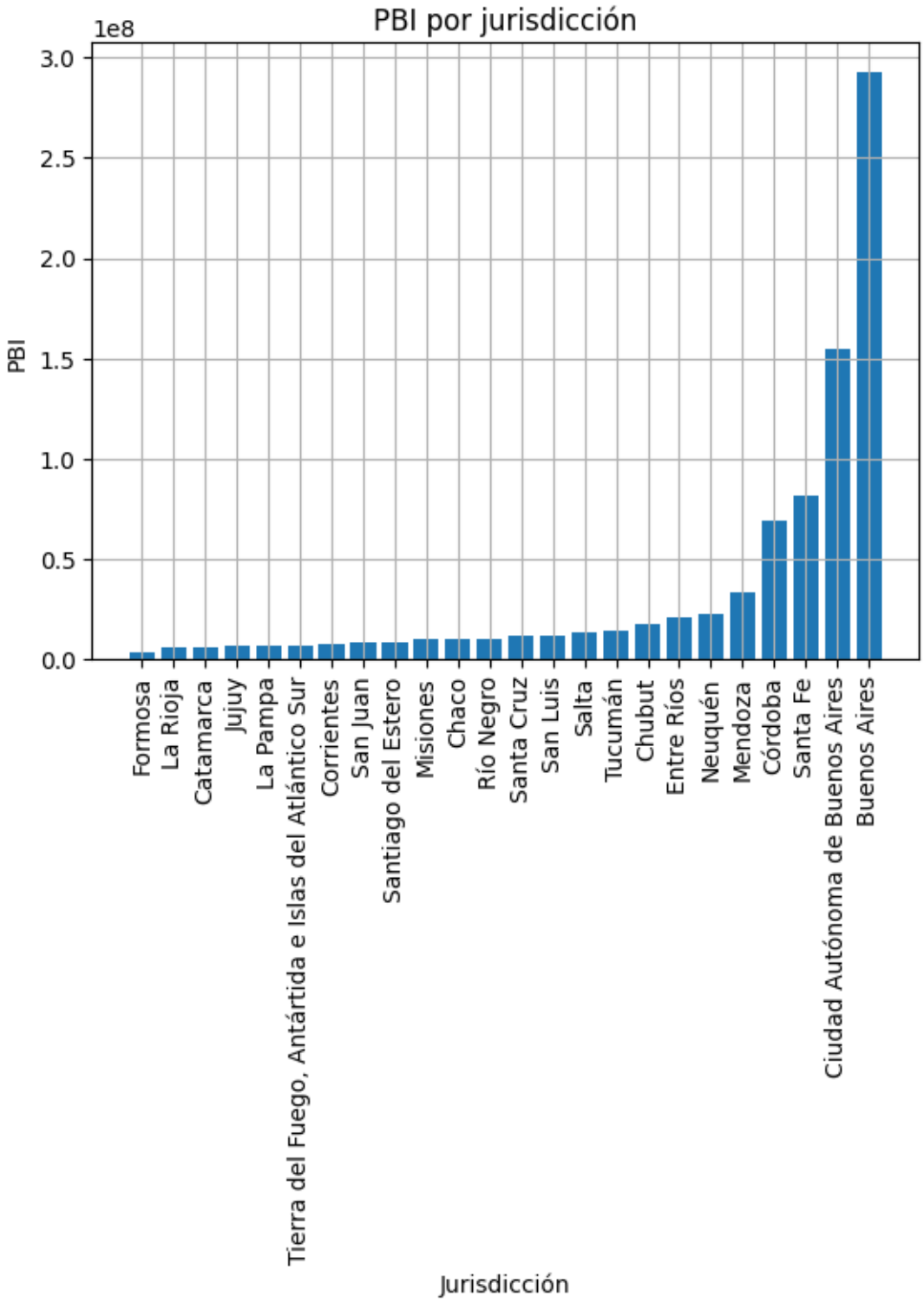



Figura 5: Los labels presentan un ángulo de 90 grados

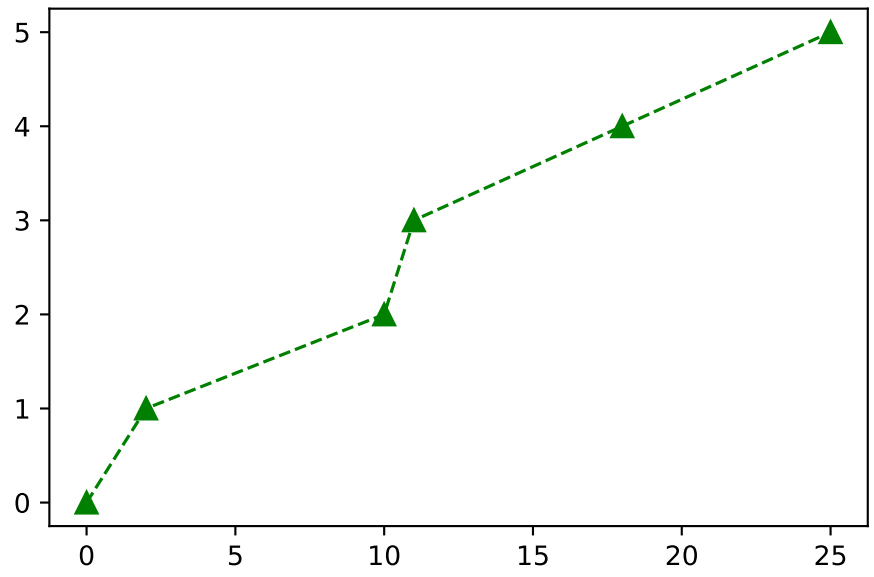
### Cambiando colores y estilos

Para cambiar los colores de los gráficos, podemos utilizar los parámetros `color`, `marker`, `linestyle`, `markersize` y `linewidth` dentro de la función `plot()`.

- **color** = nombre del color, por ejemplo: `'blue'`, `'green'`, `'red'`, etc.
- **marker** = forma de los puntos o marcadores, por ejemplo: `'^'`, `'o'`, `'v'`, etc.
- **linestyle** = estilo de línea, por ejemplo: `'solid'`, `'dashed'`, `'dotted'` o sus equivalentes:`'-'`, `'--'`, `':'`, entre otros.
- **markersize, linewidth** = con un número, establecemos el tamaño del marcador y el espesor de la línea respectivamente.

Si no le asignamos un valor, se establecen los predefinidos.

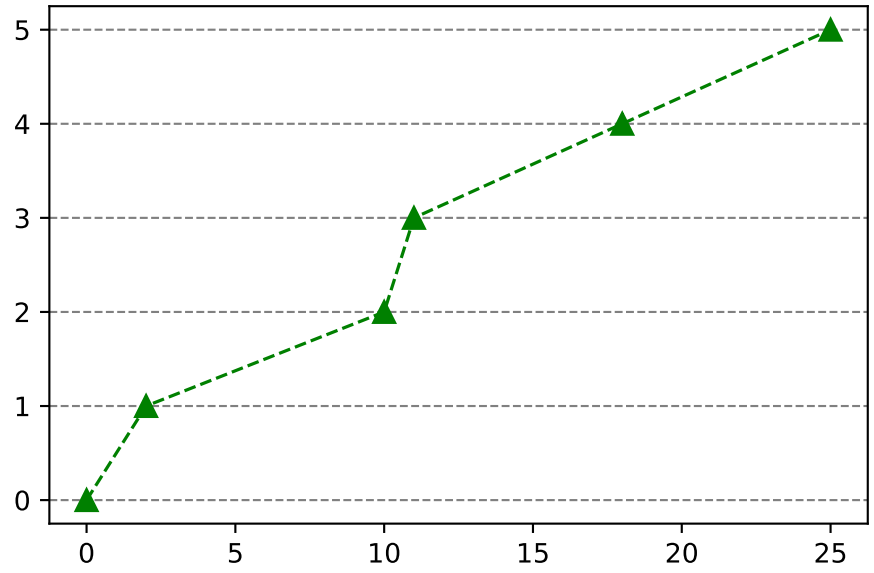
```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)

fig, ax = plt.subplots()

ax.plot(x, y, color='green', marker='^', linestyle='--', markersize=8, linewidth=1.2)
plt.show()
```



**Grilla personalizada:**

Si deseamos modificarle a una grilla, por ejemplo, el color, el estilo de línea, o sólo queremos ver uno de los ejes, podemos indicarlo utilizando parámetros muy similares a los vistos anteriormente pero en la funcion `grid()`.

```
x = [0,2,10,11,18,25]    # Tiempo (min)
y = [0,1,2,3,4,5]        # Distancia (m)

fig, ax = plt.subplots()

ax.plot(x, y, color='green', marker='^', linestyle='--', markersize=8, linewidth=1.2)

#Grilla modificada
ax.grid(axis = 'y', color = 'gray', linestyle = 'dashed')
plt.show()
```




Con esto finalizamos los temas de la materia.
¡Esperamos que hayas disfrutado de este recorrido!
Recordá que si querés dejarnos feedback podés hacerlo en el formulario que está en la parte de Contacto.

Si te interesa ser docente de la materia, podés ver los requisitos y escribirnos en la sección Ser Docente.

¡Muchos éxitos!