

# **Geoinformática**

Pablo López-Ramírez

1/4/23

# Table of contents

|   |               |
|---|---------------|
| <b>Prefacio</b>   | <b>4</b>      |
| <b>Introducción</b>   | <b>5</b>      |
| Organización del libro . . . . .                                      | 5             |
| <br><b>I Geoinformática: las herramientas básicas</b>                 | <br><b>6</b>  |
| <b>1 Transformación de datos</b>                                      | <b>8</b>      |
| 1.1 Limpieza de los datos . . . . .                                   | 10            |
| 1.2 Descripciones de los datos . . . . .                              | 14            |
| 1.3 Creación de variables . . . . .                                   | 15            |
| 1.3.1 Modificar valores . . . . .                                     | 16            |
| 1.3.2 Eliminar columnas . . . . .                                     | 17            |
| 1.3.3 Buscando datos . . . . .  | 18            |
| 1.4 Ordenar valores . . . . .   | 19            |
| 1.5 Exploración Visual . . . . .                                      | 20            |
| 1.6 Organizando los datos . . . . .                                   | 25            |
| 1.7 Agrupamiento, Transformación y Agregación . . . . .               | 29            |
| 1.8 Para Practicar . . . . .  | 34            |
| <br><b>2 Limpieza y transformación de datos de COVID-19 en México</b> | <br><b>36</b> |
| 2.1 Aplanado de datos . . . . .                                       | 38            |
| 2.2 Manejo de fechas . . . . .  | 47            |
| 2.3 Exportar datos . . . . .  | 50            |
| 2.3.1 Tarea . . . . .   | 51            |
| <br><b>3 Automatización</b>   | <br><b>52</b> |
| 3.1 Preproceso . . . . .  | 54            |
| 3.2 Preprocesar usando nuestras funciones . . . . .                   | 57            |
| 3.3 Guardando el resultado . . . . .                                  | 58            |
| <br><b>4 Curvas epidémicas</b>  | <br><b>59</b> |
| 4.0.1 Curva de casos confirmados . . . . .                            | 60            |
| 4.0.2 Curva de defunciones . . . . .                                  | 65            |
| 4.0.3 Combinando las dos gráficas . . . . .                           | 66            |

|                                   |           |
|-----------------------------------|-----------|
| 4.0.4 Hospitalizaciones . . . . . | 69        |
| <b>II Geoinformática en R</b>     | <b>72</b> |
| <b>5 Introducción a R</b>         | <b>74</b> |
| <b>6 Summary</b>                  | <b>75</b> |
| <b>References</b>                 | <b>76</b> |

# Prefacio

Para nosotros en CentroGeo, la computación no es sólo una herramienta para ayudarnos a resolver diferentes problemas geoespaciales; la computación es una parte integral del proceso de análisis y una forma de *pensar* en geografía. Ser capaz de programar nos permite liberarnos de los algoritmos, técnicas y configuraciones que se incluyen en el software (comercial o abierto) para el análisis de datos geográficos y pensar los problemas de forma diferente. Abrir la posibilidad de automatizar los procesos de análisis no sólo hace más eficiente nuestro trabajo, sino que también nos permite pensar en los problemas de forma diferente, de forma *computacional*. Con el fin de ayudar a nuestros estudiantes (y a estudiantes o profesionales de otras instituciones) a adquirir las herramientas técnicas básicas para poder llevar a cabo tareas de análisis de datos geoespaciales en Python o en R hemos creado este libro.

Este libro es (por lo pronto, pretende ser) una compilación de materiales educativos sobre Geoinformática. Busca funcionar como un apoyo para profesores interesados en impartir cursos relacionados con el uso de herramientas de programación para el análisis de datos geográficos o bien, para estudiantes independientes que busquen complementar su formación de manera autodidacta.

El libro y el material incluido se distribuye bajo una licencia Creative Commons, de forma que todo mundo es libre de utilizarlo y modificarlo de acuerdo a sus propias necesidades, siempre citando la fuente original.

Este libro fue creado con Quarto.

Para aprender más sobre quarto, visita: <https://quarto.org/docs/books>.

# Introducción

Este libro es una introducción a las principales herramientas de Python y R para el Análisis Espacial, está pensado como una primera aproximación y se centra en el uso de las herramientas más que en los conceptos de análisis.

El libro busca dar a los lectores un entendimiento intuitivo de las posibilidades, limitaciones y capacidades de las herramientas geoinformáticas y su aplicación en el Análisis Espacial, sin detallar extensivamente en el transfondo matemático o teórico de los métodos estudiados. En este sentido, el curso asume que los estudiantes cuentan con nociones básicas de programación y estadística y, preferentemente, de Análisis Espacial.

Los temas elegidos fueron escogidos para brindar a los estudiantes de un panorama amplio, mas no completo, de las herramientas, tanto tecnológicas como matemáticas de la geoinformática. La idea es que, aún si no se conocen todas las técnicas y métodos existentes, los estudiantes adquieran la formación necesaria para entender los conceptos básicos, el funcionamiento general de las herramientas computacionales y sean capaces de adaptarlos a sus propios proyectos. De esta forma el curso busca establecer una base para que, en un futuro, los estudiantes investiguen más a profundidad los temas que se relacionen con sus intereses particulares.

## Organización del libro

El libro está (estará) organizado en dos grandes secciones: una dedicada a Python y otra a R. Está organizado como un conjunto de talleres. En cada taller se revisarán algunas ideas detrás del análisis de datos geospaciales con énfasis en las herramientas y técnicas computacionales. Cada taller contiene todo el código necesario y las explicaciones básicas.

## **Part I**

# **Geoinformática: las herramientas básicas**

En esta parte del libro vamos a tratar de cubrir los fundamentos técnicos del procesamiento, análisis y visualización de datos geoespaciales con Python.

Para seguir los talleres vas a necesitar varios conjuntos de datos que puedes descargar de [aquí](#).

El libro está desarrollado a partir de [Notebooks de Jupyter](#), de forma que lo más natural es que vayas siguiendo el desarrollo del libro utilizando estos notebooks.

La forma más sencilla de instalar Jupyter y las librerías que estaremos utilizando es utilizando el gestor de paquetes [conda](#), que nos permite instalar fácilmente paquetes de Python sin preocuparnos por dependencias del sistema.

Existen varias formas de instalar y trabajar con conda. Para usuarios de Windows quizá lo más sencillo sea instalar el paquete de cómputo científico [Anaconda](#). Anaconda contiene, además del gestor de paquetes [conda](#), muchas librerías ya preinstaladas por lo que puede resultar un poco excesivo en tamaño.

Para trabajar de mejor forma en Python es recomendable crear *environments* de trabajo. Un *environment* es algo así como una instalación independiente de Python que contiene todo lo necesario para el desarrollo de un proyecto específico. A continuación les dejo un par de tutoriales en video para aprender a trabajar con *environments* de [conda](#):

[Anaconda Beginners Guide for Linux and Windows - Python Working Environments Tutorial](#)

[Master the basics of Conda environments in Python](#)

Finalmente, [conda](#) viene configurado por defecto para utilizar los repositorios de [Anaconda, Inc.](#). La empresa provee acceso a sus repositorios sin ningún costo, sin embargo en este repositorio no siempre se encuentran las versiones más actualizadas y completas que vamos a necesitar. Para evitar dificultades les recomiendo utilizar los repositorios de [conda-forge](#), acá les dejo un tutorial:

[Tutorial conda-forge](#)

# 1 Transformación de datos

Las bases de datos provenientes de situaciones reales son problemáticas; contienen datos faltantes, diferentes estructuras, etcétera. En general, para analizar un conjunto de datos primero es necesario transformarlos, ya sea para corregir errores o para ajustarlos a un formato que se adapte mejor a los métodos de análisis que queremos utilizar.

Dentro del análisis de datos en general y especialmente en el análisis de datos geoespaciales, una buena parte del tiempo y esfuerzo se consume en tareas relacionadas con la limpieza y transformación de datos. Dado lo extenuantes y relevantes que son estas tareas, es sorprendente encontrar que existen muy pocas publicaciones referentes a los patrones, técnicas y buenas prácticas existentes para una eficiente limpieza, manipulación y transformación de los datos.

En este taller nos vamos a enfocar en utilizar bases de datos provenientes del mundo real para aprender cómo leerlas usando bibliotecas de Python, de modo que puedan ser transformadas y manipuladas con el fin de ser eventualmente analizadas.

El primer paso es importar todas las librerías que vamos a utilizar:

```
import os
import pandas as pd
import seaborn as sns
import numpy as np
```

- `os` Provee de múltiples utilidades del sistema
- `pandas` La librería fundamental para la transformación de datos en Python
- `seaborn` Herramientas para generar gráficas estadísticas
- `numpy` la librería básica de análisis numérico y vectorial

---

Vamos a utilizar los datos del *Censo de Población y Vivienda 2020* de INEGI. Trabajaremos con los datos a nivel AGEB para la Ciudad de México. Una AGEB se define como un *Área Geográfica ocupada por un conjunto de manzanas perfectamente delimitadas por calles, avenidas, andadores o cualquier otro rasgo de fácil identificación en el terreno y cuyo uso de suelo es principalmente habitacional, industrial, de servicios, etc..* Las AGEB's son la unidad básica de representatividad del Marco Geoestadístico Nacional, son lo suficientemente pequeñas para



representar la variabilidad espacial, pero lo suficientemente grandes para mantener la privacidad de la población y disminuir efectos de ruido estadístico.

Los datos son [publicados por INEGI](#) en un archivo en formato csv que contiene diferentes agregaciones geográficas en el mismo archivo. Para entenderlo bien, vamos a abrirlo:

**i** Note

El archivo con los datos lo encuentras en la carpeta de datos del libro con el nombre `conjunto_de_datos_ageb_urbana_09_cpv2020.zip`

**⚠** Warning

Dentro de este libro, la convención es que los datos están guardados en la carpeta `datos/` relativa al notebook que se esté ejecutando.

```
db = pd.read_csv('datos/conjunto_de_datos_ageb_urbana_09_cpv2020.zip',
                 dtype={'ENTIDAD': object,
                       'MUN': object,
                       'LOC': object,
                       'AGEB': object})

db.head()
```

|   | ENTIDAD | NOM_ENT          | MUN | NOM_MUN                              | LOC  | NOM_LOC               |
|---|---------|------------------|-----|--------------------------------------|------|-----------------------|
| 0 | 09      | Ciudad de México | 000 | Total de la entidad Ciudad de México | 0000 | Total de la entidad   |
| 1 | 09      | Ciudad de México | 002 | Azcapotzalco                         | 0000 | Total del municipio   |
| 2 | 09      | Ciudad de México | 002 | Azcapotzalco                         | 0001 | Total de la localidad |
| 3 | 09      | Ciudad de México | 002 | Azcapotzalco                         | 0001 | Total AGEB urbana     |
| 4 | 09      | Ciudad de México | 002 | Azcapotzalco                         | 0001 | Azcapotzalco          |

La librería [Pandas](#) es la que provee la funcionalidad para trabajar con datos *tabulares* en Python. La estructura fundamental de Pandas es el [DataFrame](#), podemos pensar en los DataFrames como hojas de Excel, con columnas *nombradas* que funcionan como índices para las variables y filas para las observaciones.

Para leer el archivo utilizamos el método `read_csv()` de los DataFrames de Pandas. El parámetro `dtype` que le pasamos a la función nos asegura que ciertas columnas se lean con un tipo de datos especial, en este caso como `object`, para asegurarnos que no se lean como números y perdamos identificadores, vamos a regresar a esto más adelante.

La columna que nos interesa ahorita es `NOM_LOC`, esta nos ayuda a distinguir los datos que vienen en cada fila: las filas etiquetadas con `Total AGEB urbana` contienen los conteos para

cada AGEB de todas las variables, entonces, nuestra primera tarea es filtrar la base y quedarnos sólo con las columnas que en la columna `NOM_LOC` dice `Total AGEB urbana`.

```
db = db.loc[db['NOM_LOC'] == 'Total AGEB urbana']
db.head()
```

|     | ENTIDAD | NOM_ENT          | MUN | NOM_MUN      | LOC  | NOM_LOC           | AGEB | MZA |
|-----|---------|------------------|-----|--------------|------|-------------------|------|-----|
| 3   | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0010 | 0   |
| 30  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0025 | 0   |
| 82  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 003A | 0   |
| 116 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0044 | 0   |
| 163 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0097 | 0   |

Lo que hicimos aquí fue utilizar el selector `loc` de pandas para seleccionar las filas que queremos, pasándole el filtro que nos interesa, en este caso `db['NOM_LOC'] == 'Total AGEB urbana'`

## 1.1 Limpieza de los datos

Hasta aquí lo que tenemos es un `DataFrame` con todas las variables del censo agregadas por AGEB. Ahora, para poder realizar análisis a partir de esta base de datos, necesitamos asegurarnos de que los datos son del tipo correcto, es decir, si vamos a hacer cuentas, los datos deben ser de tipo `float` o `int`. Utilicemos entonces la *propiedad* `db.dtypes` para *preguntar* los tipos de datos.

```
db.dtypes
```

```
ENTIDAD      object
NOM_ENT      object
MUN          object
NOM_MUN      object
LOC          object
...
VPH_CVJ      object
VPH_SINRTV   object
VPH_SINLTC   object
VPH_SINCINT  object
VPH_SINTIC   object
Length: 230, dtype: object
```

Como podemos ver, no sólo las columnas que pedimos que leyera como `object` las leyó así, también las demás columnas. Esto se puede deber a que tienen codificados valores faltantes con caracteres especiales, por lo que pandas no pudo convertirlos automáticamente en números.

Para entender esto un poco mejor, vamos a leer el diccionario de datos del censo.

#### Note

También pueden explorar el archivo en excel, para verlo con más calma

```
diccionario = pd.read_csv('datos/diccionario_datos_ageb_urbana_09_cpv2020.csv', skiprows=3)
diccionario
```

|     | Núm. | Indicador   | Descripción                                      |
|-----|------|---|--|
| 0   | 1    | Clave de entidad federativa                       | Código que identifica a la entidad federativa    |
| 1   | 2    | Entidad federativa                                | Nombre oficial de la entidad federativa.         |
| 2   | 3    | Clave de municipio o demarcación territorial      | Código que identifica al municipio o demarcación |
| 3   | 4    | Municipio o demarcación territorial               | Nombre oficial del municipio o demarcación       |
| 4   | 5    | Clave de localidad                                | Código que identifica a la localidad al interi.  |
| ... | ...  | ...   | ...  |
| 225 | 218  | Viviendas particulares habitadas que disponen ... | Viviendas particulares habitadas que tienen      |
| 226 | 219  | Viviendas particulares habitadas sin radio ni ... | Viviendas particulares habitadas que no cue      |
| 227 | 220  | Viviendas particulares habitadas sin línea tel... | Viviendas particulares habitadas que no cue      |
| 228 | 221  | Viviendas particulares habitadas sin computado... | Viviendas particulares habitadas que no cue      |
| 229 | 222  | Viviendas particulares habitadas sin tecnológi... | Viviendas particulares habitadas que no cue      |

#### Warning

Fíjense como pasamos `skiprows=3` para leer el diccionario del censo. Esto le dice a pandas que el header (los nombres de las columnas), vienen en el cuarto renglón.

A partir de este diccionario podemos ver que hay varias formas de codificar valores faltantes: '999999999', '99999999', '\*' y 'N/D'.

Para poder convertir todas estas columnas en numéricas tenemos que reemplazar todos esos valores por la forma en la que se expresan los datos faltantes en Pandas, utilizando el valor *Not a Number* de numpy. Para hacer este reemplazo vamos a usar la función `replace` de Pandas, que toma como argumento el valor que queremos reemplazar y el valor por el cual lo queremos reemplazar:

```
db = (db
      .replace('999999999', np.nan)
      .replace('99999999', np.nan)
      .replace('*', np.nan)
      .replace('N/D', np.nan))
```

¡Esta fue una instrucción complicada!

Pero no es realmente difícil. Como hemos visto hasta aquí, los métodos de los DataFrames en general regresan otros DataFrames con el resultado de la operación, esto nos permite *encadenar* métodos, de forma que cuando hacemos `db.replace('999999999', np.nan).replace('99999999', np.nan)`, el segundo `replace` opera sobre el resultado del primero y así sucesivamente. Este encadenamiento de métodos nos ayuda a escribir código más fácil de leer.

Ahora ya tenemos todos los valores faltantes codificados adecuadamente, sin embargo aún nos falta convertirlos a números ¿verdad?

```
db.dtypes
```

```
ENTIDAD      object
NOM_ENT      object
MUN          object
NOM_MUN      object
LOC          object
...
VPH_CVJ      object
VPH_SINRTV   object
VPH_SINLTC   object
VPH_SINCINT  object
VPH_SINTIC   object
Length: 230, dtype: object
```

La forma normal de cambiar el tipo de datos de una columna es utilizar el método `astype`

```
db['VPH_CVJ'].astype('float').dtypes
```

```
dtype('float64')
```

### Note

Aquí no estamos asignando el resultado de la operación a ninguna variable, el resultado de esta operación no modifica el valor de los datos.

Así podríamos ir cambiando columna por columna, pero como estamos programando ¡nos gusta hacer las cosas en bruto!

En el diccionario de datos tenemos los nombres de todas las variables, entonces podemos utilizar estos nombres para seleccionar todas las columnas que contienen datos numéricos y cambiar su tipo en el DataFrame. Fíjense que las primeras 8 filas del diccionario contienen los identificadores geográficos:

```
diccionario.head(8)
```

|   | Núm. | Indicador                                    | Descripción                                       |
|---|------|--|---|
| 0 | 1    | Clave de entidad federativa                  | Código que identifica a la entidad federativa.... |
| 1 | 2    | Entidad federativa                           | Nombre oficial de la entidad federativa.          |
| 2 | 3    | Clave de municipio o demarcación territorial | Código que identifica al municipio o demarcaci... |
| 3 | 4    | Municipio o demarcación territorial          | Nombre oficial del municipio o demarcación ter... |
| 4 | 5    | Clave de localidad                           | Código que identifica a la localidad al interi... |
| 5 | 6    | Localidad                                    | Nombre con el que se reconoce a la localidad d... |
| 6 | 7    | Clave del AGEB                               | Clave que identifica al AGEB urbana, al interi... |
| 7 | 8    | Clave de manzana                             | Clave que identifica a la manzana, al interior... |

Las demás filas contienen los nombres (y descripciones) de las variables del Censo.

```
campos_datos = diccionario.loc[8:,] ['Mnemónico']
campos_datos
```

```
8      POBTOT
9      POBFEM
10     POBMAS
11     P_OA2
12     P_OA2_F
...
225    VPH_CVJ
226    VPH_SINRTV
227    VPH_SINLTC
228    VPH_SINCINT
```

```
229      VPH_SINTIC
Name: Mnemónico, Length: 222, dtype: object
```

Aquí utilizamos una vez más el método `loc` para seleccionar filas en nuestros datos. En esta ocasión seleccionamos las filas por *índice* (en este momento nuestro índice es simplemente el número de fila, más adelante usaremos índices diferentes), la selección `loc[8:,]` simplemente quiere decir *todas las columnas para las filas de la 9 en adelante*.

También estamos seleccionando una única columna al hacer `['Mnemónico']`, el resultado de esta selección ya no es un `DataFrame`, es una [Serie](#). Las series son las estructuras que usa Pandas para guardar una sólo columna (o fila).

Las Series se pueden utilizar (igual que las listas) para seleccionar columnas de un `DataFrame`, entoinces, ahora sí podemos cambiar todos los tipos de datos de una sola vez.

```
db[campos_datos] = db[campos_datos].astype('float')
db.dtypes
```

```
ENTIDAD      object
NOM_ENT      object
MUN          object
NOM_MUN      object
LOC          object
...
VPH_CVJ      float64
VPH_SINRTV   float64
VPH_SINLTC   float64
VPH_SINCINT   float64
VPH_SINTIC   float64
Length: 230, dtype: object
```

## 1.2 Descripciones de los datos

Pandas nos provee una serie de métodos para obtener descripciones generales de la tabla. Podemos usar el método `info` para obtener una descripción general de la estructura de la tabla y el espacio que ocupa en la memoria:

```
db.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2433 entries, 3 to 68915
Columns: 230 entries, ENTIDAD to VPH_SINTIC
dtypes: float64(222), int64(1), object(7)
memory usage: 4.3+ MB
```

Para obtener las estadísticas descriptivas podemos usar el método `describe`:

```
db.describe()
```

|       | MZA    | POBTOT       | POBFEM       | POBMAS      | P_0A2       | P_0A2_F     | P_0A2_M     | P_3  |
|-------|--------|--------------|--------------|-------------|-------------|-------------|-------------|------|
| count | 2433.0 | 2433.000000  | 2422.000000  | 2423.00000  | 2406.000000 | 2392.000000 | 2390.000000 | 2423 |
| mean  | 0.0    | 3758.993835  | 1970.647812  | 1804.64837  | 109.901912  | 54.471990   | 56.089121   | 3661 |
| std   | 0.0    | 2433.068753  | 1254.533102  | 1186.95856  | 85.636899   | 42.286817   | 43.908616   | 2347 |
| min   | 0.0    | 0.000000     | 0.000000     | 0.00000     | 0.000000    | 0.000000    | 0.000000    | 0.00 |
| 25%   | 0.0    | 2045.000000  | 1083.500000  | 974.00000   | 46.250000   | 23.000000   | 24.000000   | 2018 |
| 50%   | 0.0    | 3396.000000  | 1783.500000  | 1616.00000  | 91.000000   | 45.000000   | 46.000000   | 3304 |
| 75%   | 0.0    | 4992.000000  | 2617.500000  | 2391.00000  | 152.000000  | 75.000000   | 77.000000   | 4852 |
| max   | 0.0    | 21198.000000 | 11128.000000 | 10616.00000 | 709.000000  | 350.000000  | 393.000000  | 2053 |

### 1.3 Creación de variables

Muchas veces vamos a querer crear nuevas columnas a partir de las ya existentes. Por ejemplo, podemos estar interesados en el porcentaje de población femenina en cada AGEBA.

```
pct_fem = db['POBFEM'] / db['POBTOT']
pct_fem.head()
```

```
3      0.532516
30     0.521187
82     0.527037
116    0.535025
163    0.512408
dtype: float64
```

Fíjense cómo usamos `/` para dividir dos columnas. El resultado de la operación lo guardamos en la variable `pct_fem` ¿De qué tipo será esta variable?

```
pct_fem.info()
```

```
<class 'pandas.core.series.Series'>
Int64Index: 2433 entries, 3 to 68915
Series name: None
Non-Null Count  Dtype
-----
2405 non-null   float64
dtypes: float64(1)
memory usage: 38.0 KB
```

Es una serie, es decir una columna en nuestro caso. Como esta columna comparte el mismo *índice* que los datos originales (es resultado de una operación renglón por renglón), entonces la podemos agregar al DataFrame original facilmente:

```
db['pct_fem'] = pct_fem
db['pct_fem'].head()
```

```
/tmp/ipykernel_5237/2610780181.py:1: PerformanceWarning: DataFrame is highly fragmented.  This may cause some problems in the future. Try passing
db['pct_fem'] = pct_fem
```

```
3      0.532516
30     0.521187
82     0.527037
116    0.535025
163    0.512408
Name: pct_fem, dtype: float64
```

### 1.3.1 Modificar valores

De la misma forma que podemos agregar columnas (o filas) a nuestro DataFrame, podemos también modificar los valores existentes. Para explorar esto, vamos a crear una nueva columna y llenarla con valores *nulos*:

```
# Nueva columna llena de solamente el número 1
db['Nueva'] = None
db['Nueva'].head()
```



```
/tmp/ipykernel_5237/463547730.py:2: PerformanceWarning: DataFrame is highly fragmented.  This
  db['Nueva'] = None
```

```
3      None
30     None
82     None
116    None
163    None
Name: Nueva, dtype: object
```

Podemos fácilmente cambiar los valores de todas las filas:

```
db['Nueva'] = 1
db['Nueva'].head()
```

```
3      1
30     1
82     1
116    1
163    1
Name: Nueva, dtype: int64
```

O también cambiar el valor sólo para una fila específica:

```
db.loc[3, 'Nueva'] = 10
db['Nueva'].head()
```

```
3      10
30     1
82     1
116    1
163    1
Name: Nueva, dtype: int64
```

### 1.3.2 Eliminar columnas

Eliminar columnas es igualmente fácil usando el método `drop`:

```
db = db.drop(columns=['Nueva'])
'Nueva' in db.columns
```

False

¡Fíjense como *preguntamos* al final si ya habíamos eliminado la columna!

### 1.3.3 Buscando datos

Muchas veces queremos encontrar *observaciones* que cumplan con uno o más criterios. Una vez más, el método `loc` es nuestro amigo para seleccionar datos. Supongamos que queremos encontrar aquellas AGEs que tengan una población de ‘65 años o más’ mayor a 1,000 personas.

```
db_seleccion = db.loc[db['POB65_MAS'] > 1000, :]
db_seleccion.head()
```

|      | ENTIDAD | NOM_ENT          | MUN | NOM_MUN      | LOC  | NOM_LOC           | AGEB | MZA |
|------|---------|------------------|-----|--------------|------|-------------------|------|-----|
| 30   | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0025 | 0   |
| 444  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0186 | 0   |
| 3617 | 09      | Ciudad de México | 003 | Coyoacán     | 0001 | Total AGEB urbana | 0107 | 0   |
| 4075 | 09      | Ciudad de México | 003 | Coyoacán     | 0001 | Total AGEB urbana | 0287 | 0   |
| 4886 | 09      | Ciudad de México | 003 | Coyoacán     | 0001 | Total AGEB urbana | 0573 | 0   |

Simplemente pasamos la *condición* que nos interesa al selector y listo.

Los criterios de búsqueda pueden ser tan sofisticados como se requiera, por ejemplo, podemos seleccionar los AGEs en los cuales la población de 0 a 14 años sea menor a un cuarto de la población total:

```
db_seleccion = db.loc[(db['POB0_14'] / db['POBTOT']) < 0.25, :]
db_seleccion.head()
```

|     | ENTIDAD | NOM_ENT          | MUN | NOM_MUN      | LOC  | NOM_LOC           | AGEB | MZA |
|-----|---------|------------------|-----|--------------|------|-------------------|------|-----|
| 3   | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0010 | 0   |
| 30  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0025 | 0   |
| 82  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 003A | 0   |
| 116 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0044 | 0   |

|     | ENTIDAD | NOM_ENT          | MUN | NOM_MUN      | LOC  | NOM_LOC           | AGEB | MZA |
|-----|---------|------------------|-----|--------------|------|-------------------|------|-----|
| 163 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0097 | 0   |

Podemos hacer combinaciones arbitrarias de selectores utilizando los operadores lógicos & (and) y | (or). Por ejemplo, podemos combinar nuestras selecciones anteriores para encontrar las AGEBS con menos de 50% de mujeres y población de 0 a 14 años sea menor a un cuarto de la población total

```
db_seleccion = db.loc[(db['pct_fem'] < 0.5) &
                      ((db['POB0_14'] / db['POBTOT']) < 0.25), :]
db_seleccion.head()
```

|      | ENTIDAD | NOM_ENT          | MUN | NOM_MUN      | LOC  | NOM_LOC           | AGEB | MZA |
|------|---------|------------------|-----|--------------|------|-------------------|------|-----|
| 2342 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0877 | 0   |
| 3292 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 1165 | 0   |
| 5321 | 09      | Ciudad de México | 003 | Coyoacán     | 0001 | Total AGEB urbana | 0770 | 0   |
| 6016 | 09      | Ciudad de México | 003 | Coyoacán     | 0001 | Total AGEB urbana | 1092 | 0   |
| 7919 | 09      | Ciudad de México | 003 | Coyoacán     | 0001 | Total AGEB urbana | 1660 | 0   |

## 1.4 Ordenar valores

Finalmente, vamos a ver cómo ordenar los datos de acuerdo a los valores de un campo. Pensemos que queremos ver las 10 AGEBS más pobladas de la ciudad.

```
db.sort_values('POBTOT', ascending = False).head(10)
```

|       | ENTIDAD | NOM_ENT          | MUN | NOM_MUN               | LOC  | NOM_LOC           | AGEB | MZA |
|-------|---------|------------------|-----|-----------------------|------|-------------------|------|-----|
| 39932 | 09      | Ciudad de México | 010 | Álvaro Obregón        | 0001 | Total AGEB urbana | 0135 |     |
| 63316 | 09      | Ciudad de México | 016 | Miguel Hidalgo        | 0001 | Total AGEB urbana | 0444 |     |
| 65102 | 09      | Ciudad de México | 016 | Miguel Hidalgo        | 0001 | Total AGEB urbana | 1349 |     |
| 9394  | 09      | Ciudad de México | 004 | Cuajimalpa de Morelos | 0020 | Total AGEB urbana | 0316 |     |
| 9090  | 09      | Ciudad de México | 004 | Cuajimalpa de Morelos | 0001 | Total AGEB urbana | 0369 |     |
| 9190  | 09      | Ciudad de México | 004 | Cuajimalpa de Morelos | 0001 | Total AGEB urbana | 0373 |     |
| 6211  | 09      | Ciudad de México | 003 | Coyoacán              | 0001 | Total AGEB urbana | 1162 |     |
| 52537 | 09      | Ciudad de México | 012 | Tlalpan               | 0001 | Total AGEB urbana | 2121 |     |
| 26177 | 09      | Ciudad de México | 007 | Iztapalapa            | 0001 | Total AGEB urbana | 1994 |     |
| 42074 | 09      | Ciudad de México | 010 | Álvaro Obregón        | 0001 | Total AGEB urbana | 1171 |     |

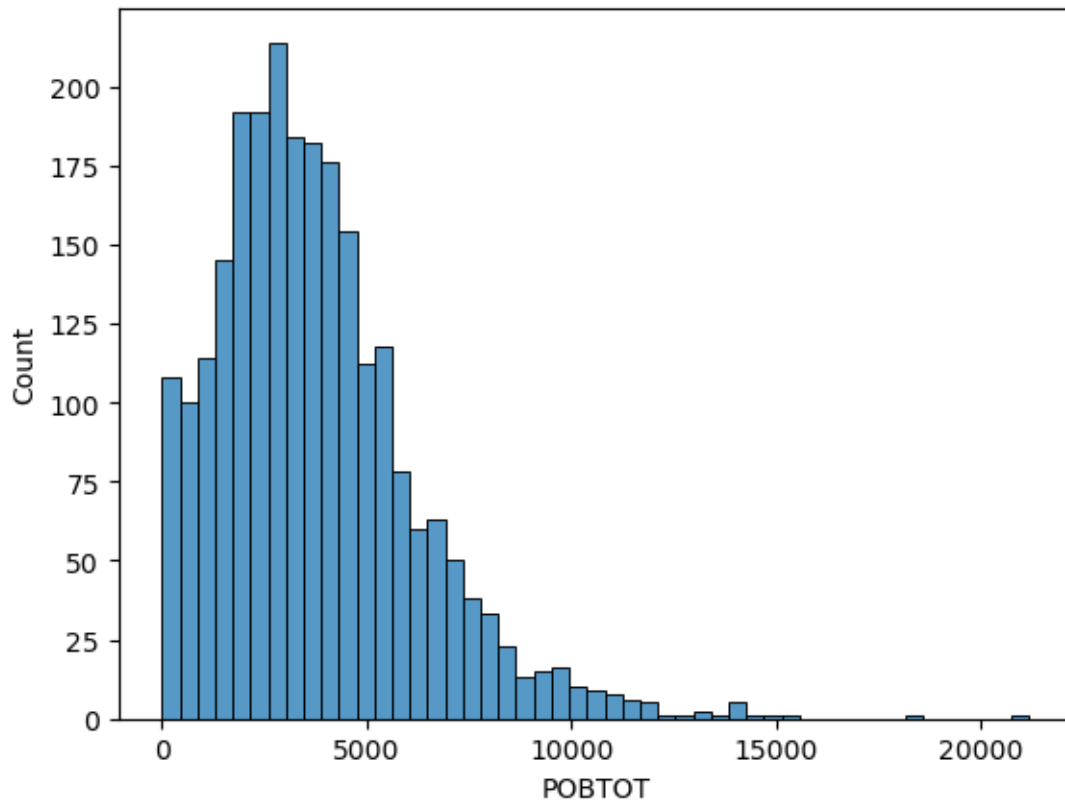
El método `sort_values` nos permite ordenar los datos de acuerdo al valor (o criterio) que queramos. El argumento `ascending = False` indica que los queremos ordenar de forma descendente.

## 1.5 Exploración Visual

Ya que nos empezamos a familiarizar con el manejo de datos usando Pandas, podemos empezar a hacer cosas más divertidas, por ejemplo, explorar visualmente los datos.

La librería `seaborn` nos ofrece una serie de herramientas para la exploración visual de los datos. Podemos comenzar con un histograma para ver la distribución de los valores de una columna.

```
_ = sns.histplot(db['POBTOT'], kde = False)
```



La función `histplot` de `seaborn` nos regresa el histograma, el argumento `kde=False` le dice que no queremos que ajuste una distribución empírica.

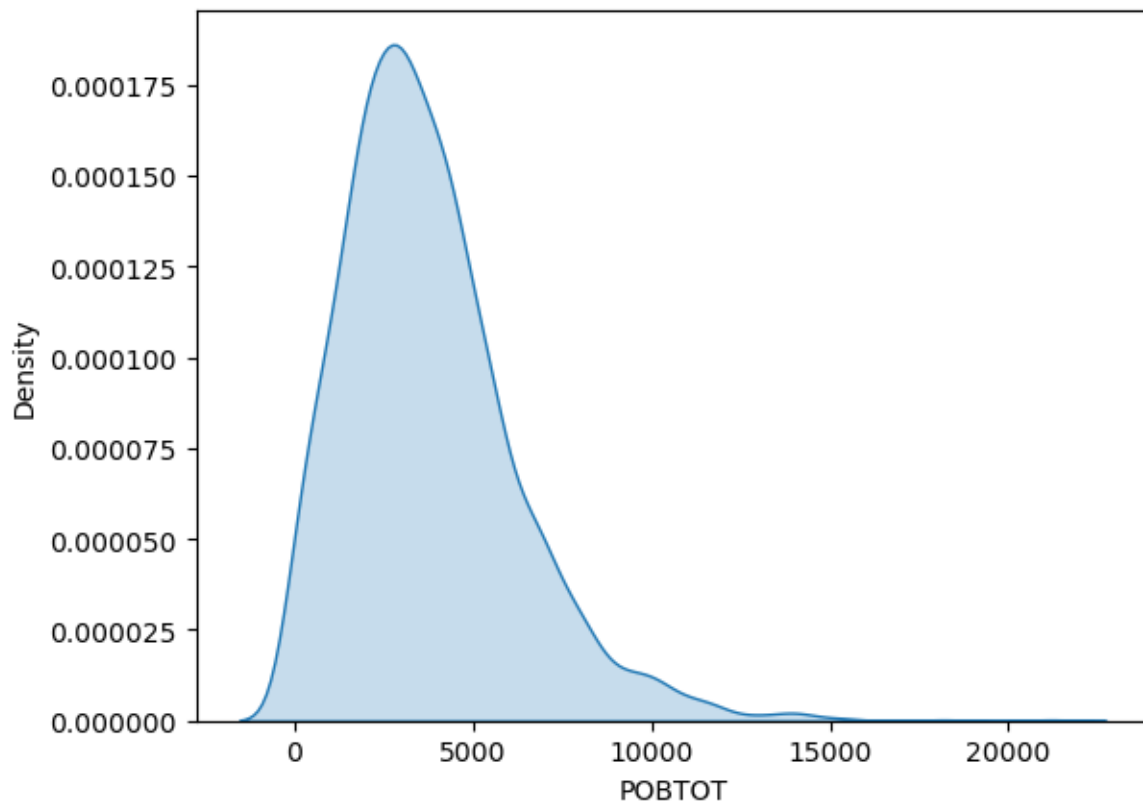
### Note

Cuando hicimos `_ = sns.histplot(db['POBTOT'], kde = False)` estamos asignando el resultado a la variable `_`, esto se hace comunmente cuando no queremos ya hacer nada más con ese resultado. Más adelante haremos operaciones sobre las gráficas.

#### 1.5.0.1 Densidad de Kernel

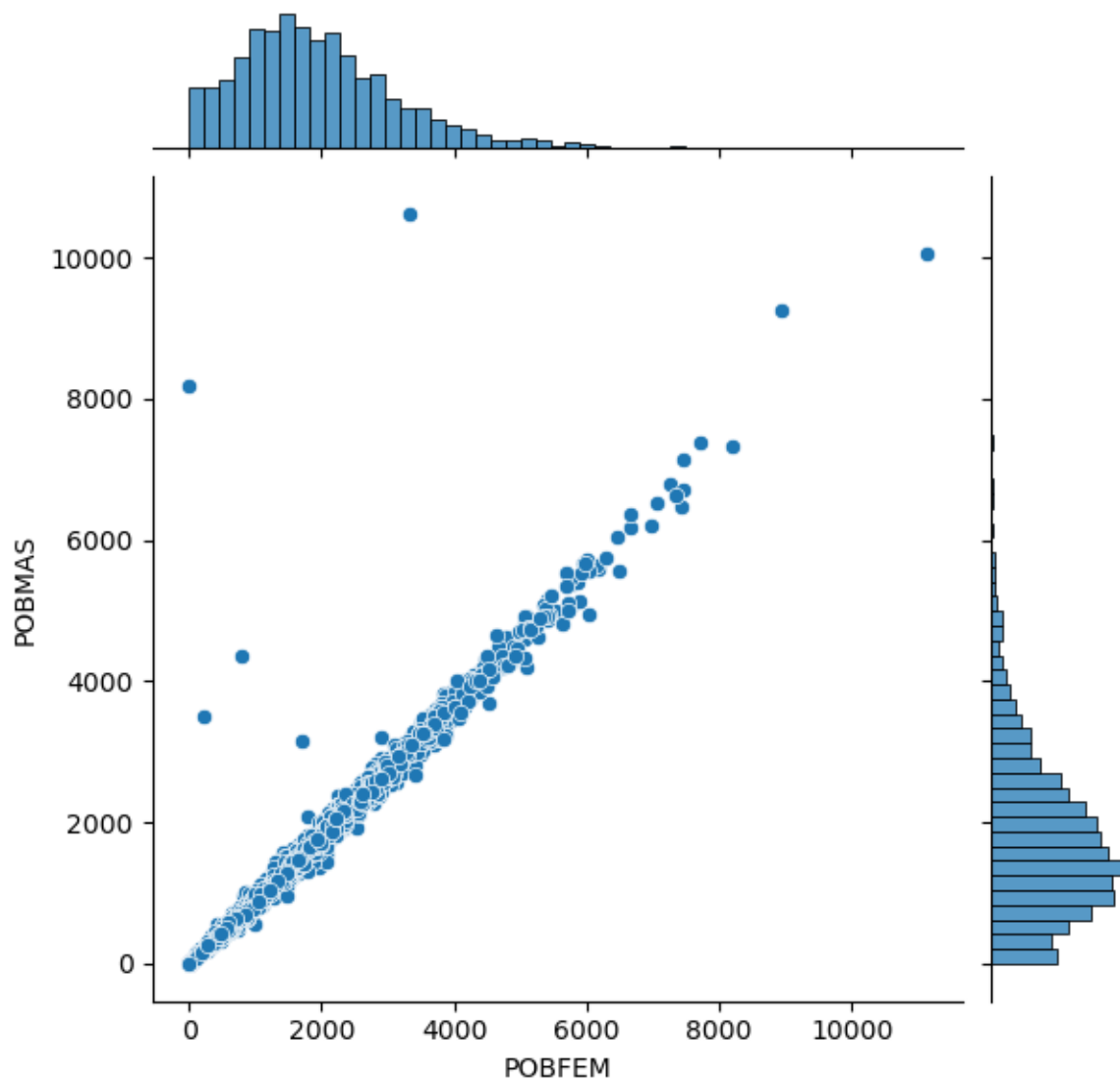
Otra forma de representar la distribución de una variable es ajustando una *densidad de kernel*, que estima una distribución (empírica) de probabilidad a partir de nuestras observaciones.

```
_ = sns.kdeplot(db['POBTOT'], fill = True)
```



Otra visualización muy útil es la de la *distribución conjunta* de dos variables. Por ejemplo, supongamos que queremos comparar las distribuciones de la población masculina y femenina.

```
_ = sns.jointplot(data=db, x='POBFEM', y='POBMAS')
```

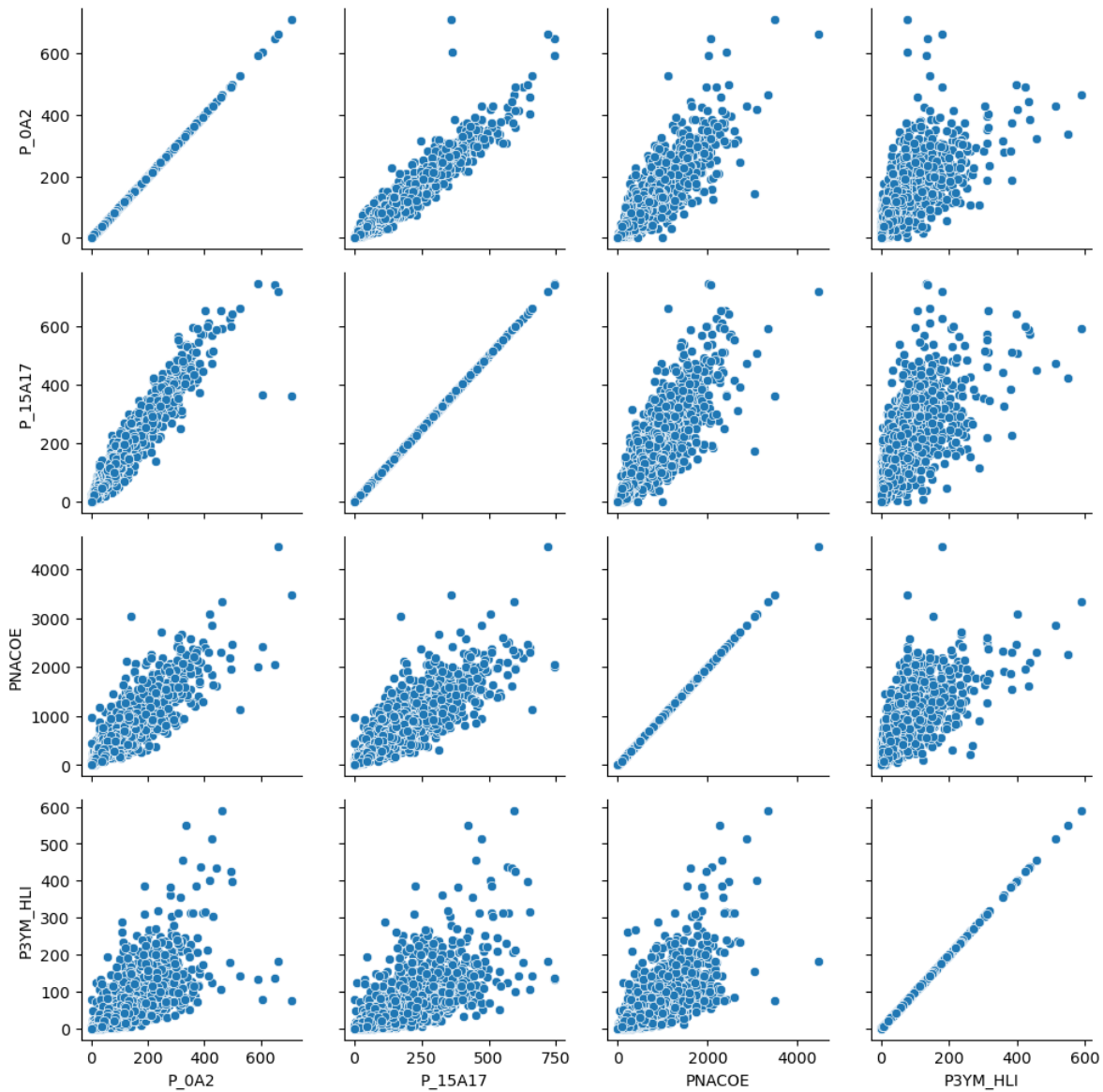


La relación, como es de esperarse, es casi perfectamente lineal, pero ver las distribuciones conjuntas nos permite identificar algunas AGEBS con poblaciones masculinas desproporcionadamente grandes ¿Qué serán?.

Muchas veces queremos visualizar la distribución conjunta de varias variables al mismo tiempo. Por ejemplo cuando queremos hacer ejercicios de regresión queremos explorar la correlación entre las covariables. Una forma de visualizar rápidamente estas distribuciones conjuntas es

con un `PairGrid`. Utlicemos uno sencillo para ver las distribuciones de algunas variables.

```
vars = ['P_OA2', 'P_15A17', 'PNACOE', 'P3YM_HLI']  
g = sns.PairGrid(db[vars])  
g = g.map(sns.scatterplot)
```



La función `PairPlot` sólo nos prepara la malla (un cuadrado del número de variables de los datos) y con el `map` llenamos esa malla con la gráfica que queramos, en nuestro caso un

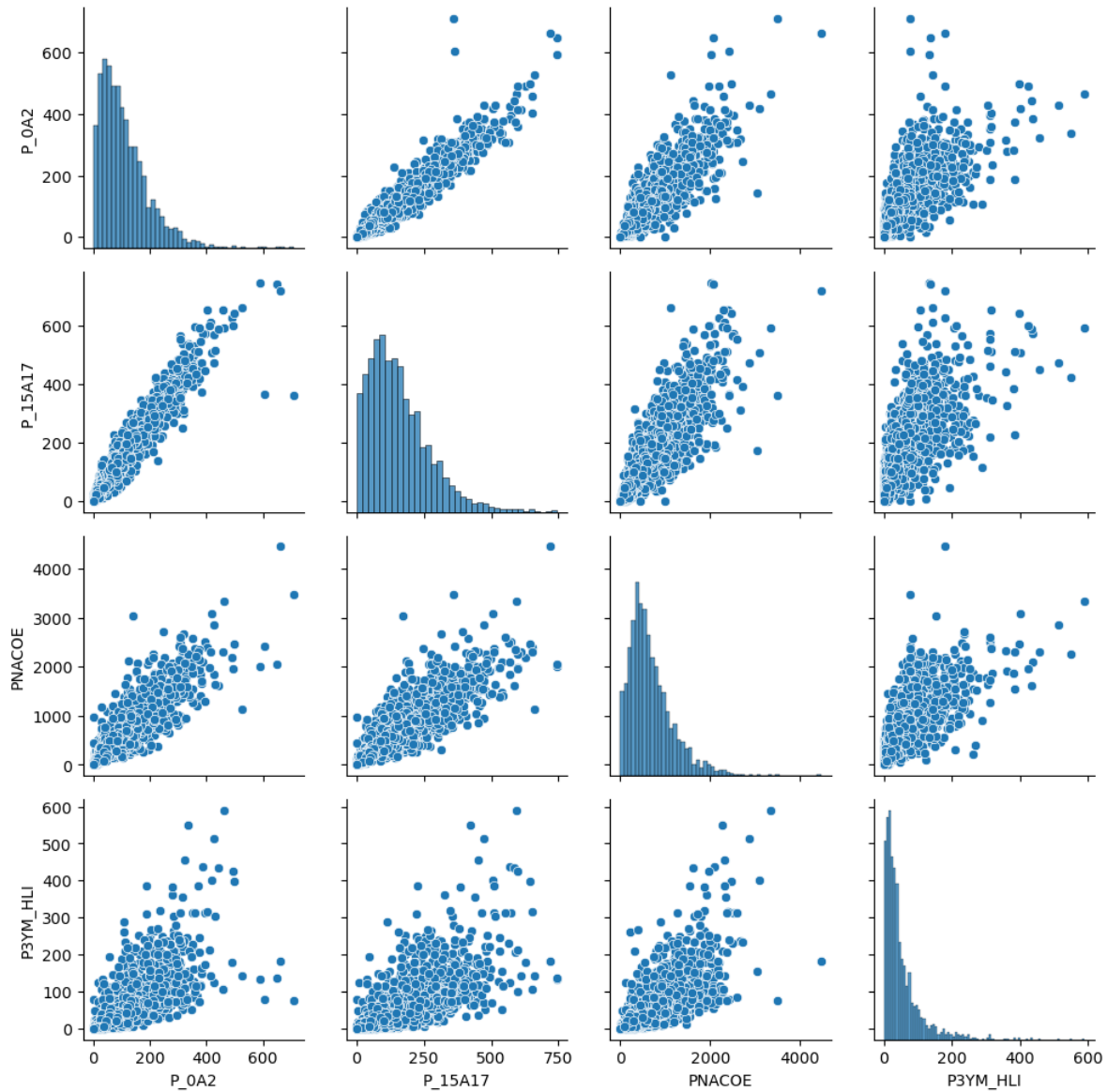
diagrama de dispersión.

En este caso la diagonal no es muy informativa, es un diagrama de dispersión de una variable consigo misma. `PairPlot` es muy flexible y nos permite *mapear* diferentes funciones para la diagonal y los demás elementos, por ejemplo:

```
g = sns.PairGrid(db[vars])
g.map_diag(sns.histplot)
g.map_offdiag(sns.scatterplot)
```

```
<seaborn.axisgrid.PairGrid at 0x7f70a19c4040>
```





---

## 1.6 Organizando los datos

Muchos flujos de análisis requieren organizar los datos en una estructura particular conocida como *Tidy Data* (algo así como *datos ordenados*). La idea es tener una estructura estandarizada con principios comunes de manipulación que sirva como entrada a diferentes tipos de análisis.

Las tres características fundamentales de un conjunto de datos *bien ordenado* de acuerdo a los principios *tidy* son:

1. Cada variable en una columna
2. Cada observación en una fila
3. Cada unidad de observación en una tabla

Para mayor información sobre el concepto de *Tidy Data*, puede consultarse el [Artículo Académico](#) original (de Acceso Libre), así como el [Repositorio Público](#) asociado a él.

Tratemos de aplicar el concepto de *Tidy Data* a los datos de la práctica. Primero, recordando su estructura:

```
db.head()
```

|     | ENTIDAD | NOM_ENT          | MUN | NOM_MUN      | LOC  | NOM_LOC           | AGEB | MZA |
|-----|---------|------------------|-----|--------------|------|-------------------|------|-----|
| 3   | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0010 | 0   |
| 30  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0025 | 0   |
| 82  | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 003A | 0   |
| 116 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0044 | 0   |
| 163 | 09      | Ciudad de México | 002 | Azcapotzalco | 0001 | Total AGEB urbana | 0097 | 0   |

Esta base de datos no cumple con las características *tidy*. En efecto, tenemos las variables en columnas (sin contar los identificadores), pero:

- Tenemos dos tipos de unidades: personas y viviendas. El principio *tidy* nos indica que necesitamos dos tablas para representar los datos.
- Para cada tipo de unidad tenemos en la misma fila tantas *observaciones* como variables (del mismo tipo). Por ejemplo, el valor de la población para cada grupo de edad en cada AGEB es una observación.

Entonces, vamos a trabajar en acomodar la tabla a los principios *tidy*. Para comenzar, trabajemos sólo con las variables que representan segmentos de edad de la población. Seleccionar sólo estas columnas puede ser engorroso, pero si nos fijamos en el diccionario, podemos observar que todas las variables que nos interesan empiezan con 'P\_'. Podemos usar esta observación para seleccionar, a partir de la lista de columnas, sólo las que nos interesan:

```
cols_pob = [c for c in db.columns if c.startswith('P_')]
print(cols_pob)
```

```
['P_OA2', 'P_OA2_F', 'P_OA2_M', 'P_3YMAS', 'P_3YMAS_F', 'P_3YMAS_M', 'P_5YMAS', 'P_5YMAS_F',
```

Ahora, vamos a construir un identificador único de AGEB para cada fila concatenando los identificadores de entidad, municipio, localidad y ageb:

```
db['AGEB_cvgeo'] = db['ENTIDAD'] + db['MUN'] + db['LOC'] + db['AGEB']
db['AGEB_cvgeo'].head()
```

```
3      0900200010010
30     0900200010025
82     090020001003A
116    0900200010044
163    0900200010097
Name: AGEN_cvgeo, dtype: object
```

Ya con este identificador, podemos eliminar de la tabla los identificadores que usamos para construirlo

```
db = db.drop(columns=['ENTIDAD', 'MUN', 'LOC', 'AGEB'])
```

Copiamos las columnas que nos interesan a una nueva tabla

```
rangos = db[['AGEB_cvgeo'] + cols_pob]
rangos.head()
```

|     | AGEB_cvgeo    | P_0A2 | P_0A2_F | P_0A2_M | P_3YMAS | P_3YMAS_F | P_3YMAS_M | P_3YMAS_T |
|-----|---------------|-------|---------|---------|---------|-----------|-----------|-----------|
| 3   | 0900200010010 | 60.0  | 32.0    | 28.0    | 3123.0  | 1663.0    | 1460.0    | 3046.0    |
| 30  | 0900200010025 | 122.0 | 58.0    | 64.0    | 5470.0  | 2856.0    | 2614.0    | 5340.0    |
| 82  | 090020001003A | 88.0  | 49.0    | 39.0    | 4147.0  | 2183.0    | 1964.0    | 4096.0    |
| 116 | 0900200010044 | 110.0 | 49.0    | 61.0    | 4658.0  | 2502.0    | 2156.0    | 4516.0    |
| 163 | 0900200010097 | 40.0  | 16.0    | 24.0    | 2136.0  | 1099.0    | 1037.0    | 2196.0    |

Ahora vamos a reorganizar la tabla de forma que cada grupo de edad corresponda a una fila en lugar de una columna, de esta forma tenemos las observaciones en filas, de acuerdo al principio *tidy*.

Para lograr esto lo que tenemos que hacer es la operación inversa de un pivote, es decir, un *stack*. El método `stack` hace justo lo que necesitamos, sólo tenemos que especificar el índice (lo que distingue a cada observación) que queremos utilizar para cada fila, en este caso `AGEB_cvgeo`.

```
rangos = rangos.set_index('AGEB_cvgeo').stack()
rangos
```

```

AGEB_cvgeo
0900200010010  P_0A2          60.0
                 P_0A2_F       32.0
                 P_0A2_M       28.0
                 P_3YMAS       3123.0
                 P_3YMAS_F     1663.0
                 ...
0901700011524  P_18A24_M      230.0
                 P_15A49_F     1111.0
                 P_60YMAS       706.0
                 P_60YMAS_F     394.0
                 P_60YMAS_M     312.0
Length: 96555, dtype: float64

```

Perfecto, eso se parece bastante a lo que buscamos, sólo que en lugar de un DataFrame lo que tenemos es una Serie. Fíjense que para cada valor del índice (`AGEB_cvgeo`), tenemos todos los valores de los grupos de población.

Para convertir esto en un DataFrame lo más sencillo es quitar el índice que creamos con la función `reset_index`:

```

rangos = rangos.reset_index()
rangos.head()

```

|   | AGEB_cvgeo    | level_1   | 0      |
|---|---------------|-----------|--------|
| 0 | 0900200010010 | P_0A2     | 60.0   |
| 1 | 0900200010010 | P_0A2_F   | 32.0   |
| 2 | 0900200010010 | P_0A2_M   | 28.0   |
| 3 | 0900200010010 | P_3YMAS   | 3123.0 |
| 4 | 0900200010010 | P_3YMAS_F | 1663.0 |

Ahora tenemos un DataFrame en el que el valor de la columna `AGEB_cvgeo` viene repetido para cada observación. Ya sólo necesitamos renombrar las columnas restantes para que nos indiquen más claramente su contenido:

```

rangos = rangos.rename(columns = {'level_1': 'Grupo', 0: 'Población'})
rangos.head()

```

|   | AGEB_cvgeo    | Grupo     | Población |
|---|---------------|-----------|-----------|
| 0 | 0900200010010 | P_0A2     | 60.0      |
| 1 | 0900200010010 | P_0A2_F   | 32.0      |
| 2 | 0900200010010 | P_0A2_M   | 28.0      |
| 3 | 0900200010010 | P_3YMAS   | 3123.0    |
| 4 | 0900200010010 | P_3YMAS_F | 1663.0    |

!Ahora tenemos nuestra tabla acomodada a los principios *tidy*!

## 1.7 Agrupamiento, Transformación y Agregación

Una ventaja de tener los datos estructurados de acuerdo a los principios *tidy* es la facilidad con la que podemos realizar procesos de transformación más sofisticados como agrupaciones y sumarios. Las agrupaciones consisten en *agrupar* observaciones en una tabla de acuerdo a sus valores (o expresiones) en una columna, a los datos agrupados se le pueden aplicar operaciones de agregación más o menos arbitrarias.

Digamos, por ejemplo, que queremos obtener los totales de población para cada grupo etario a través de todas las AGEBs. Para hacer esto tenemos que *agrupar* las observaciones por cada **Grupo** y después obtener el valor agregado por la suma. Vamos por partes.

```
grupos = rangos.groupby('Grupo')
grupos
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f70945509d0>
```

La función `groupby` nos permite agrupar los datos de acuerdo a una (o más) columnas. El resultado, como pueden ver, no es un DataFrame sino un objeto de la clase especial `pandas.core.groupby.generic.DataFrameGroupBy`. Esta clase sirve para representar DataFrames *agregados*, estos objetos nos permiten obtener de forma fácil los valores que corresponden a diferentes funciones de *agregación*. Por ejemplo, para obtener el total de población por cada grupo, podemos agregar nuestro objeto con la función `sum`:

```
grupos.sum(numeric_only=True)
```

| Grupo      | Población |
|------------|-----------|
| P_0A2      | 264424.0  |
| P_0A2_F    | 130297.0  |
| P_0A2_M    | 134053.0  |
| P_12A14    | 364225.0  |
| P_12A14_F  | 179955.0  |
| P_12A14_M  | 184240.0  |
| P_12YMAS   | 7864313.0 |
| P_12YMAS_F | 4141887.0 |
| P_12YMAS_M | 3722424.0 |
| P_15A17    | 377178.0  |
| P_15A17_F  | 185144.0  |
| P_15A17_M  | 191984.0  |
| P_15A49_F  | 2490275.0 |
| P_15YMAS   | 7500071.0 |
| P_15YMAS_F | 3961914.0 |
| P_15YMAS_M | 3538155.0 |
| P_18A24    | 975897.0  |
| P_18A24_F  | 483893.0  |
| P_18A24_M  | 491985.0  |
| P_18YMAS   | 7122878.0 |
| P_18YMAS_F | 3776738.0 |
| P_18YMAS_M | 3346138.0 |
| P_3A5      | 321650.0  |
| P_3A5_F    | 158674.0  |
| P_3A5_M    | 162933.0  |
| P_3YMAS    | 8871506.0 |
| P_3YMAS_F  | 4637724.0 |
| P_3YMAS_M  | 4233780.0 |
| P_5YMAS    | 8660874.0 |
| P_5YMAS_F  | 4533469.0 |
| P_5YMAS_M  | 4127403.0 |
| P_60YMAS   | 1487004.0 |
| P_60YMAS_F | 850901.0  |
| P_60YMAS_M | 636074.0  |
| P_6A11     | 685511.0  |
| P_6A11_F   | 337113.0  |
| P_6A11_M   | 348375.0  |
| P_8A14     | 829786.0  |
| P_8A14_F   | 408494.0  |

|          | Población |
|----------|-----------|
| Grupo    |           |
| P_8A14_M | 421280.0  |

Como ve, al usar un *agregador* sobre el objeto agrupado obtenemos un DataFrame con los valores que corresponden a la agregación que utilizamos.

#### **i** Note

El parámetro `numeric_only=True` le dice al agregador que sólo calcule el resultado para las columnas de tipo numérico.

En este caso la función que usamos para agregar los datos es la suma, sin embargo es posible utilizar cualquier función que opere sobre grupos de observaciones, por ejemplo, el promedio:

```
grupos.mean(numeric_only=True)
```

|            | Población   |
|------------|-------------|
| Grupo      |             |
| P_0A2      | 109.901912  |
| P_0A2_F    | 54.471990   |
| P_0A2_M    | 56.089121   |
| P_12A14    | 151.130705  |
| P_12A14_F  | 74.701121   |
| P_12A14_M  | 76.702748   |
| P_12YMAS   | 3245.692530 |
| P_12YMAS_F | 1710.110239 |
| P_12YMAS_M | 1536.287247 |
| P_15A17    | 156.310816  |
| P_15A17_F  | 77.014975   |
| P_15A17_M  | 79.993333   |
| P_15A49_F  | 1028.614209 |
| P_15YMAS   | 3095.365662 |
| P_15YMAS_F | 1635.802642 |
| P_15YMAS_M | 1460.237309 |
| P_18A24    | 403.596774  |
| P_18A24_F  | 200.038446  |
| P_18A24_M  | 204.143154  |
| P_18YMAS   | 2939.693768 |
| P_18YMAS_F | 1559.346821 |

| Grupo      | Población   |
|------------|-------------|
| P_18YMAS_M | 1380.989682 |
| P_3A5      | 133.298798  |
| P_3A5_F    | 65.949293   |
| P_3A5_M    | 67.775790   |
| P_3YMAS    | 3661.372678 |
| P_3YMAS_F  | 1914.832370 |
| P_3YMAS_M  | 1747.329757 |
| P_5YMAS    | 3574.442427 |
| P_5YMAS_F  | 1871.787366 |
| P_5YMAS_M  | 1703.426744 |
| P_60YMAS   | 615.481788  |
| P_60YMAS_F | 353.511010  |
| P_60YMAS_M | 264.040681  |
| P_6A11     | 284.326421  |
| P_6A11_F   | 140.055256  |
| P_6A11_M   | 144.734109  |
| P_8A14     | 343.739022  |
| P_8A14_F   | 169.218724  |
| P_8A14_M   | 174.587650  |

Las funciones que usamos para agregar (`sum` y `mean`) son funciones de `numpy` podemos utilizar cualquier [función de agregación](#). También es posible calcular diferentes agregaciones al mismo tiempo:

```
grupos.aggregate([np.sum, np.mean, np.std])
```

```
/tmp/ipykernel_5237/732611272.py:1: FutureWarning: ['AGEB_cvgeo'] did not aggregate successfully
grupos.aggregate([np.sum, np.mean, np.std])
```

| Grupo     | Población |            |            |
|-----------|-----------|------------|------------|
|           | sum       | mean       | std        |
| P_0A2     | 264424.0  | 109.901912 | 85.636899  |
| P_0A2_F   | 130297.0  | 54.471990  | 42.286817  |
| P_0A2_M   | 134053.0  | 56.089121  | 43.908616  |
| P_12A14   | 364225.0  | 151.130705 | 111.565262 |
| P_12A14_F | 179955.0  | 74.701121  | 55.572013  |



| Grupo      | Población |             |             |
|------------|-----------|-------------|-------------|
|            | sum       | mean        | std         |
| P_12A14_M  | 184240.0  | 76.702748   | 56.746606   |
| P_12YMAS   | 7864313.0 | 3245.692530 | 2056.644056 |
| P_12YMAS_F | 4141887.0 | 1710.110239 | 1073.566831 |
| P_12YMAS_M | 3722424.0 | 1536.287247 | 1001.153466 |
| P_15A17    | 377178.0  | 156.310816  | 113.532155  |
| P_15A17_F  | 185144.0  | 77.014975   | 56.107357   |
| P_15A17_M  | 191984.0  | 79.993333   | 57.996607   |
| P_15A49_F  | 2490275.0 | 1028.614209 | 692.206450  |
| P_15YMAS   | 7500071.0 | 3095.365662 | 1955.668987 |
| P_15YMAS_F | 3961914.0 | 1635.802642 | 1023.764553 |
| P_15YMAS_M | 3538155.0 | 1460.237309 | 950.879515  |
| P_18A24    | 975897.0  | 403.596774  | 279.378732  |
| P_18A24_F  | 483893.0  | 200.038446  | 138.590941  |
| P_18A24_M  | 491985.0  | 204.143154  | 143.125222  |
| P_18YMAS   | 7122878.0 | 2939.693768 | 1853.201763 |
| P_18YMAS_F | 3776738.0 | 1559.346821 | 973.233847  |
| P_18YMAS_M | 3346138.0 | 1380.989682 | 899.958704  |
| P_3A5      | 321650.0  | 133.298798  | 101.904268  |
| P_3A5_F    | 158674.0  | 65.949293   | 50.305709   |
| P_3A5_M    | 162933.0  | 67.775790   | 52.251282   |
| P_3YMAS    | 8871506.0 | 3661.372678 | 2347.050678 |
| P_3YMAS_F  | 4637724.0 | 1914.832370 | 1215.700184 |
| P_3YMAS_M  | 4233780.0 | 1747.329757 | 1147.281855 |
| P_5YMAS    | 8660874.0 | 3574.442427 | 2284.544513 |
| P_5YMAS_F  | 4533469.0 | 1871.787366 | 1185.089392 |
| P_5YMAS_M  | 4127403.0 | 1703.426744 | 1115.802146 |
| P_60YMAS   | 1487004.0 | 615.481788  | 358.110680  |
| P_60YMAS_F | 850901.0  | 353.511010  | 206.712937  |
| P_60YMAS_M | 636074.0  | 264.040681  | 152.406790  |
| P_6A11     | 685511.0  | 284.326421  | 213.690386  |
| P_6A11_F   | 337113.0  | 140.055256  | 105.214351  |
| P_6A11_M   | 348375.0  | 144.734109  | 109.164209  |
| P_8A14     | 829786.0  | 343.739022  | 255.780534  |
| P_8A14_F   | 408494.0  | 169.218724  | 126.379228  |
| P_8A14_M   | 421280.0  | 174.587650  | 130.222008  |

## 1.8 Para Practicar

La organización *Wikileaks* posee una Base de Datos pública en la cual se contiene, entre otras cosas, el número de casualidades existentes durante los primeros años de la *Guerra de Afganistán*, la cual puede ser consultada a través de la siguiente liga:

[https://docs.google.com/spreadsheets/d/1EAx8\\_ksSCmoWW\\_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit](https://docs.google.com/spreadsheets/d/1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit)

A partir de los datos, realiza los siguientes ejercicios: \* Descarga la tabla como un archivo de tipo `.csv` (Archivo -> Descargar como -> `.csv`, hoja actual). \* Importa los datos a un `DataFrame` de `Pandas`. \* Explora los datos generando estadísticas descriptivas y algunas gráficas. \* Examina qué tanto se ajusta a los principios del *Tidy Data* y ajústalo según creas conveniente \* Obten una cuenta total de las bajas por mes y genera una gráfica con dicho conteo.

|  |   |           |         |           |               |                                |                         |
|--|---|-----------|---------|-----------|---------------|--------------------------------|-------------------------|
| <div> <div>Wikileaks Afghanistan war logs analysis</div> <div> <div>☆</div> <div>🔗</div> <div>📄</div> </div> </div> <div> <div>Archivo</div> <div>Editar</div> <div>Ver</div> <div>Insertar</div> <div>Formato</div> <div>Datos</div> <div>Herramientas</div> <div>Extensiones</div> <div>Ayuda</div> </div> |   |           |         |           |               |                                |                         |
| <div> <div>🖨️</div> <div>🔍</div> <div>100%</div> <div>Solo lectura</div> </div>  |   |           |         |           |               |                                |                         |
| <div> <div>A1:D1</div> <div>fx</div> <div>Casualties detailed in the war logs, month by month</div> </div>   |   |           |         |           |               |                                |                         |
|  | A   | B         | C       | D         | E             | F                              | G                       |
| 1  | Casualties detailed in the war logs, month by month |           |         |           |               |                                |                         |
| 2  | Year  | Month     | Taliban | Civilians | Afghan forces | Nato (detailed in spreadsheet) | Nato - official figures |
| 3  | 2004  | January   | 15      | 51        | 23            |                                | 11                      |
| 4  | 2004  | February  |         | 7         | 4             | 5                              | 2                       |
| 5  | 2004  | March     | 19      | 2         |               | 2                              | 3                       |
| 6  | 2004  | April     | 5       | 3         | 19            |                                | 3                       |
| 7  | 2004  | May       | 18      | 29        | 56            | 6                              | 9                       |
| 8  | 2004  | June      | 163     | 32        | 14            | 2                              | 5                       |
| 9  | 2004  | July      | 21      | 19        | 19            |                                | 2                       |
| 10   | 2004  | August    | 20      | 26        | 10            | 3                              | 4                       |
| 11   | 2004  | September | 33      | 4         | 16            |                                | 4                       |
| 12   | 2004  | October   | 13      | 14        | 14            | 2                              | 8                       |
| 13   | 2004  | November  | 21      | 19        | 36            | 2                              | 7                       |
| 14   | 2004  | December  | 15      | 13        | 7             |                                | 2                       |
| 15   | 2005  | January   | 6       | 4         | 27            | 2                              | 2                       |
| 16   | 2005  | February  | 14      | 4         |               |                                | 3                       |
| 17   | 2005  | March     | 38      | 18        | 2             | 2                              | 6                       |
| 18   | 2005  | April     | 118     | 5         | 4             | 2                              | 19                      |
| 19   | 2005  | May       | 113     | 49        | 25            | 4                              | 4                       |
| 20   | 2005  | June      | 143     | 31        | 4             | 24                             | 29                      |
| 21   | 2005  | July      | 75      | 5         | 29            | 2                              | 2                       |
| 22   | 2005  | August    | 90      | 6         | 10            | 13                             | 33                      |
| 23   | 2005  | September | 89      | 12        | 29            | 12                             | 12                      |
| 24   | 2005  | October   | 122     | 15        | 10            | 3                              | 10                      |
| 25   | 2005  | November  | 36      | 13        | 24            | 4                              | 7                       |
| 26   | 2005  | December  | 48      | 16        | 16            | 3                              | 4                       |
| 27   | 2006  | January   | 26      | 93        | 14            | 1                              | 1                       |
| 28   | 2006  | February  | 12      | 54        | 106           | 12                             | 17                      |
| 29   | 2006  | March     | 36      | 37        | 61            | 15                             | 13                      |
| 30   | 2006  | April     | 52      | 33        | 43            | 5                              | 5                       |
| 31   | 2006  | May       | 85      | 54        | 50            | 16                             | 17                      |
| 32   | 2006  | June      | 295     | 71        | 39            | 19                             | 22                      |
| 33   | 2006  | July      | 220     | 64        | 36            | 13                             | 19                      |
| 34   | 2006  | August    | 161     | 101       | 42            | 18                             | 29                      |
| 35   | 2006  | September | 945     | 172       | 97            | 19                             | 38                      |
| 36   | 2006  | October   | 333     | 44        | 70            | 17                             | 17                      |
| 37   | 2006  | November  | 325     | 46        | 14            | 5                              | 9                       |
| 38   | 2006  | December  | 216     | 51        | 36            | 3                              | 4                       |
| 39   | 2007  | January   | 125     | 53        | 49            | 1                              | 2                       |
| 40   | 2007  | February  | 152     | 33        | 35            | 6                              | 18                      |
| 41   | 2007  | March     | 76      | 48        | 48            | 4                              | 10                      |
| 42   | 2007  | April     | 166     | 123       | 68            | 26                             | 20                      |
| 43   | 2007  | May       | 199     | 71        | 125           | 19                             | 25                      |
| 44   | 2007  | June      | 564     | 80        | 98            | 27                             | 24                      |
| 45   | 2007  | July      | 485     | 83        | 114           | 25                             | 29                      |
| 46   | 2007  | August    | 551     | 65        | 89            | 31                             | 34                      |
| 47   | 2007  | September | 917     | 64        | 144           | 20                             | 24                      |
| 48   | 2007  | October   | 462     | 46        | 75            | 11                             | 15                      |
| 49   | 2007  | November  | 270     | 60        | 67            | 11                             | 22                      |
| 50   | 2007  | December  | 172     | 37        | 55            | 5                              | 9                       |
| 51   | 2008  | January   | 86      | 43        | 48            | 8                              | 14                      |
| <div> <div>☰</div> <div>IED ATTACKS</div> <div>IED ATTACK CHART</div> <div>TOTAL CASUALTIES, ALL CAUSES</div> <div>CASUALTIES CHART</div> </div>   |   |           |         |           |               |                                |                         |

Figure 1.1: Wikileaks

## 2 Limpieza y transformación de datos de COVID-19 en México

En el capítulo anterior hicimos una introducción a las herramientas básicas de Python para manipular datos. Ahora, en este capítulo, vamos a trabajar con una base de datos más compleja y trataremos de seguir un flujo de trabajo real, basado en experiencias de trabajo con equipos de apoyo a la toma de decisiones.

La base de datos que vamos a utilizar son los datos abiertos sobre COVID-19 que publica diariamente la Dirección General de Epidemiología de la Secretaría de Salud. Esta es una base muy grande y compleja que incluye el seguimiento de todos los casos confirmados de COVID en México. El tratamiento que vamos a dar a los datos, es el usado en diferentes flujos de trabajo de los grupos que estuvieron trabajando con la Secretaría de Salud para generar productos de análisis durante los primeros dos años de atención de la pandemia.

Antes de empezar a analizar la base de datos, vamos a importar las librerías que utilizaremos en el taller.

```
import os # hablar con el sistema operativo
import glob # listar directorios y ese tipo de operaciones
import itertools # herramientas para iterar objetos
from pathlib import Path # manipular rutas a directorios
import zipfile # comprimir y descomprimir archivos
import numpy as np # operaciones vectorizadas
import pandas as pd # DataFrames
from datetime import timedelta, date, datetime # Manejar fechas
import openpyxl # leer/escribir archivos de excel
import requests # Hablar con direcciones web
import logging
```

Lo primero que vamos a hacer es explorar los datos publicados por la Secretaría de Salud para entender cómo están organizados. En la [carpeta de datos del libro](#) puedes encontrar un ejemplo de la base de datos para el 9 de enero de 2023 bajo el nombre `datos_abiertos_covid19.zip`.

Para leer los datos vamos a utilizar la función `read_csv()`, esta función (como pueden ver) acepta que el csv venga comprimido en un zip.

```
df = pd.read_csv('datos/datos_abiertos_covid19.zip', dtype=object, encoding='latin-1')
df.head()
```

|   | FECHA_ACTUALIZACION | ID_REGISTRO | ORIGEN | SECTOR | ENTIDAD_UM | SEXO | ENT |
|---|---------------------|-------------|--------|--------|------------|------|-----|
| 0 | 2023-01-03          | 01e27d      | 2      | 9      | 25         | 2    | 25  |
| 1 | 2023-01-03          | 180725      | 2      | 9      | 09         | 2    | 09  |
| 2 | 2023-01-03          | 06fce8      | 1      | 12     | 07         | 1    | 07  |
| 3 | 2023-01-03          | 1a4a8d      | 1      | 12     | 23         | 2    | 27  |
| 4 | 2023-01-03          | 1933c0      | 1      | 12     | 09         | 2    | 09  |

Cada renglón en la base de datos corresponde a un caso en seguimiento, el resultado de cada caso se puede actualizar en sucesivas publicaciones de la base de datos. Las columnas describen un conjunto de variables asociadas al seguimiento de cada uno de los casos. Las dos primeras columnas corresponden a la fecha en la que se actualizó el caso y a un id único para cada caso respectivamente, en este taller no vamos a usar esas dos columnas.

Luego vienen un conjunto de columnas que describen la unidad médica de reporte y, después, las columnas que nos interesan más, que son las que describen al paciente.

Para entender un poco mejor los datos, conviene leer el archivo de catálogo. Lo pueden descargar del sitio de datos abiertos o bien usar el que viene en la carpeta de datos del libro bajo el nombre 201128 Catalogos.xlsx. Como el catálogo es un archivo de excel con varias hojas, lo vamos a leer usando openpyxl que nos va a devolver un diccionario de DataFrames que relacionan el nombre de la hoja con los datos que contiene.

```
catalogos = 'datos/201128 Catalogos.xlsx'
nombres_catalogos = ['Catálogo de ENTIDADES', # Aquí están los nombres de las hojas del excel
                    'Catálogo MUNICIPIOS',
                    'Catálogo SI_NO',
                    'Catálogo TIPO_PACIENTE',
                    'Catálogo CLASIFICACION_FINAL',
                    'Catálogo RESULTADO_LAB'
                    ]

# read_excel nos regresa un diccionario que relaciona el nombre de cada hoja con
# el contenido de la hoja como DataFrame
dict_catalogos = pd.read_excel(catalogos,
                              nombres_catalogos,
                              dtype=str,
                              engine='openpyxl')

clasificacion_final = dict_catalogos['Catálogo CLASIFICACION_FINAL']
# Aquí le damos nombre a las columnas porque en el excel se saltan dos líneas
```

```
clasificacion_final.columns = ["CLAVE", "CLASIFICACIÓN", "DESCRIPCIÓN"]
clasificacion_final
```

|   | CLAVE | CLASIFICACIÓN                                     | DESCRIPCIÓN                  |
|---|-------|---|------------------------------|
| 0 | NaN   | NaN   | NaN                          |
| 1 | CLAVE | CLASIFICACIÓN                                     | DESCRIPCIÓN                  |
| 2 | 1     | CASO DE COVID-19 CONFIRMADO POR ASOCIACIÓN CLÍ... | Confirmado por asociación    |
| 3 | 2     | CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DIC...  | Confirmado por dictaminac    |
| 4 | 3     | CASO DE SARS-COV-2 CONFIRMADO                     | Confirmado aplica cuando:    |
| 5 | 4     | INVÁLIDO POR LABORATORIO                          | Inválido aplica cuando el ca |
| 6 | 5     | NO REALIZADO POR LABORATORIO                      | No realizado aplica cuando   |
| 7 | 6     | CASO SOSPECHOSO                                   | Sospechoso aplica cuando:    |
| 8 | 7     | NEGATIVO A SARS-COV-2                             | Negativo aplica cuando el c  |

Lo que estamos viendo aquí es el catálogo de datos de la columna `CLASIFICACION_FINAL`. Este catálogo relaciona el valor de la `CLAVE` con su significado. En particular, la columna `CLASIFICACION_FINAL` es la que nos permite identificar los casos positivos como veremos más adelante.

El resto de los catálogos funciona de la misma forma, en este momento sólo vamos a utilizar la clasificación de los pacientes, pero más adelante podemos utilizar algunas de las columnas restantes.

## 2.1 Aplanado de datos

Como acabamos de ver, de alguna forma la información viene *distribuida* en tres archivos, uno con los datos, otro con las categorías que usa y un tercero con sus descripciones. Para utilizar los datos más fácilmente, sobre todo para poder hablarle a las cosas *por su nombre* en lugar de referirnos a sus valores codificados, vamos a realizar un conjunto de operaciones para *aplanar* los datos.

En el bajo mundo del análisis de datos, *aplanar* una base de datos es la operación de substituir los valores codificados a partir de un diccionario. En este caso, los datos que leímos traen valores codificados, entonces la primera misión es substituir esos valores por sus equivalentes en el diccionario.

Como la base de datos es muy grande, vamos a trabajar sólo con un estado de la república, en este caso la Ciudad de México (pero ustedes podrían elegir otro cualquiera).

Para seleccionar un estado, tenemos que *elegir* las filas del DataFrame que contengan el valor que queremos en la columna `ENTIDAD`, para eso vamos a aprender a usar nuestro primer

operador de Pandas, el operador `loc` que nos permite seleccionar filas a partir de los valores de una o más columnas.

```
# el copy() nos asegura tener una copia de los datos en lugar de una referencia,
# con eso podemos liberar la memoria más fácil
df = df.loc[df['ENTIDAD_RES'] == '09'].copy()
df.head()
```

|    | FECHA_ACTUALIZACION | ID_REGISTRO | ORIGEN | SECTOR | ENTIDAD_UM | SEXO | EN |
|----|---------------------|-------------|--------|--------|------------|------|----|
| 1  | 2023-01-03          | 180725      | 2      | 9      | 09         | 2    | 09 |
| 4  | 2023-01-03          | 1933c0      | 1      | 12     | 09         | 2    | 09 |
| 8  | 2023-01-03          | 0741e4      | 2      | 6      | 09         | 2    | 09 |
| 13 | 2023-01-03          | 1c4d2e      | 2      | 9      | 09         | 1    | 09 |
| 15 | 2023-01-03          | 0a6cd6      | 2      | 6      | 09         | 1    | 18 |

Fíjense que lo que hicimos fue reescribir en la variable `df` el resultado de nuestra selección, de forma que `df` ahora sólo contiene resultados para la CDMX.

Ahora ya con los datos filtrados y, por lo tanto, con un tamaño más manejable, vamos a empezar a trabajarlos. Lo primero que vamos a hacer es cambiar los valores de la columna `MUNICIPIO_RES` por la concatenación de las claves de estado y municipio, esto porque nos hará más adelante más fácil el trabajo de unir los datos con las geometrías de los municipios y porque además así tendremos un identificador único para estos (claro que esto sólo tiene sentido al trabajar con varios estados al mismo tiempo).

```
df['MUNICIPIO_RES'] = df['ENTIDAD_RES'] + df['MUNICIPIO_RES']
df.head()
```

|    | FECHA_ACTUALIZACION | ID_REGISTRO | ORIGEN | SECTOR | ENTIDAD_UM | SEXO | EN |
|----|---------------------|-------------|--------|--------|------------|------|----|
| 1  | 2023-01-03          | 180725      | 2      | 9      | 09         | 2    | 09 |
| 4  | 2023-01-03          | 1933c0      | 1      | 12     | 09         | 2    | 09 |
| 8  | 2023-01-03          | 0741e4      | 2      | 6      | 09         | 2    | 09 |
| 13 | 2023-01-03          | 1c4d2e      | 2      | 9      | 09         | 1    | 09 |
| 15 | 2023-01-03          | 0a6cd6      | 2      | 6      | 09         | 1    | 18 |

Ahora vamos a corregir el nombre de una columna en la base de datos para que coincida con el nombre en el diccionario y después podamos buscar automáticamente. Para corregir el nombre de la columna vamos a utilizar la función `rename` de Pandas. Esta función nos sirve para renombrar filas (el índice del DataFrame, que vamos a ver más adelante) o columnas dependiendo de qué eje seleccionemos. El eje 0 son las filas y el 1 las columnas.

```
# Como estamos usando explícitamente el parámetro columns,
# no necesitamos especificar el eje
df = df.rename(columns={'OTRA_COM': 'OTRAS_COM'})
df.columns
```

```
Index(['FECHA_ACTUALIZACION', 'ID_REGISTRO', 'ORIGEN', 'SECTOR', 'ENTIDAD_UM',
      'SEXO', 'ENTIDAD_NAC', 'ENTIDAD_RES', 'MUNICIPIO_RES', 'TIPO_PACIENTE',
      'FECHA_INGRESO', 'FECHA_SINTOMAS', 'FECHA_DEF', 'INTUBADO', 'NEUMONIA',
      'EDAD', 'NACIONALIDAD', 'EMBARAZO', 'HABLA LENGUA INDIG', 'INDIGENA',
      'DIABETES', 'EPOC', 'ASMA', 'INMUSUPR', 'HIPERTENSION', 'OTRAS_COM',
      'CARDIOVASCULAR', 'OBESIDAD', 'RENAL_CRONICA', 'TABAQUISMO',
      'OTRO_CASO', 'TOMA_MUESTRA_LAB', 'RESULTADO_LAB',
      'TOMA_MUESTRA_ANTIGENO', 'RESULTADO_ANTIGENO', 'CLASIFICACION_FINAL',
      'MIGRANTE', 'PAIS_NACIONALIDAD', 'PAIS_ORIGEN', 'UCI'],
      dtype='object')
```

Fíjense cómo otra vez reescribimos la variable `df`. La mayor parte de las operaciones en Pandas regresan un DataFrame con el resultado de la operación y no modifican el DataFrame original, entonces para *guardar* los resultados, necesitamos reescribir la variable (o guardarla con otro nombre)

Ahora sí podemos empezar a *aplanar* los datos. Vamos a empezar por resolver las claves de resultado de las pruebas COVID. En los datos originales estos vienen codificados en la columna **RESULTADO\_LAB**, pero en el diccionario ese valor se llama **RESULTADO**, entonces otra vez vamos a empezar por renombrar una columna.

```
df = df.rename(columns={'RESULTADO_LAB': 'RESULTADO'})
```

Para sustituir los valores en nuestros datos originales vamos a usar la función `map` que toma una serie (una serie es una columna de un dataframe) y *mapea* sus valores de acuerdo a una correspondencia que podemos pasar como un diccionario. Veamos poco a poco cómo hacer lo que queremos.

Lo primero que necesitamos es un diccionario que relacione los valores en nuestros datos con los nombres en el diccionario. Recordemos cómo se ve el diccionario:

```
clasificacion_final
```

|   | CLAVE | CLASIFICACIÓN | DESCRIPCIÓN |
|---|-------|---------------|-------------|
| 0 | NaN   | NaN           | NaN         |
| 1 | CLAVE | CLASIFICACIÓN | DESCRIPCIÓN |



|   | CLAVE | CLASIFICACIÓN                                     | DESCRIPCIÓN  |
|---|-------|---|--|
| 2 | 1     | CASO DE COVID-19 CONFIRMADO POR ASOCIACIÓN CLÍ... | Confirmado por asociación                          |
| 3 | 2     | CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DIC...  | Confirmado por dictaminación                       |
| 4 | 3     | CASO DE SARS-COV-2 CONFIRMADO                     | Confirmado aplica cuando el caso es confirmado     |
| 5 | 4     | INVÁLIDO POR LABORATORIO                          | Inválido aplica cuando el caso es inválido         |
| 6 | 5     | NO REALIZADO POR LABORATORIO                      | No realizado aplica cuando el caso no es realizado |
| 7 | 6     | CASO SOSPECHOSO                                   | Sospechoso aplica cuando el caso es sospechoso     |
| 8 | 7     | NEGATIVO A SARS-COV-2                             | Negativo aplica cuando el caso es negativo         |

Necesitamos un diccionario {CLASIFICACION:CLAVE} (ya sé que hay unos valores espurios, pero no nos importan porque simplemente esos no los va a encontrar en nuestra base de datos).

Para construir este diccionario, vamos a empezar por construir la *tupla* que mantiene la relación que buscamos, para eso vamos a utilizar la función `zip` que toma dos *iteradores* como entrada y regresa un *iterador* que tiene por elementos las tuplas hechas elemento a elemento entre los dos iteradores de inicio. Veámoslo con calma:

```
l1 = ['a', 'b', 'c']
l2 = [1, 2, 3]
l3 = list(zip(l1,l2))
l3
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

Lo que nos regresa `zip` es un iterador con las tuplas formadas por los pares ordenados de los iteradores de entrada. En Python un iterador es cualquier cosa que se pueda *recorrer en orden*, a veces estos iteradores, como en el caso de `zip` no regresan todas las entradas sino, para ahorrar memoria, las generan conforme se recorren, por eso hay que hacer `list(zip)` para que se generen las entradas.

Ahora sí podemos entonces crear el diccionario con el que vamos a actualizar los datos:

```
clasificacion_final = dict(zip(clasificacion_final['CLAVE'], clasificacion_final['CLASIFICACION']))
clasificacion_final
```

```
{nan: nan,
 'CLAVE': 'CLASIFICACIÓN',
 '1': 'CASO DE COVID-19 CONFIRMADO POR ASOCIACIÓN CLÍNICA EPIDEMIOLÓGICA',
 '2': 'CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DICTAMINACIÓN',
 '3': 'CASO DE SARS-COV-2 CONFIRMADO',
 ...}
```

```
'4': 'INVÁLIDO POR LABORATORIO',
'5': 'NO REALIZADO POR LABORATORIO',
'6': 'CASO SOSPECHOSO',
'7': 'NEGATIVO A SARS-COV-2'}
```

Y entonces pasarlo como argumento a la función `map`. Hay un truco aquí, `map` toma como argumento una función que, para cada llave, regresa el valor correspondiente, entonces no es propiamente el diccionario lo que vamos a pasar, sino la función `get` del diccionario que hace justo lo que queremos. Esto nos revela una propiedad curiosa de Python, los argumentos de una función pueden ser funciones.

```
df['CLASIFICACION_FINAL'] = df['CLASIFICACION_FINAL'].map(clasificacion_final.get)
df['CLASIFICACION_FINAL'].head()
```

```
1          NEGATIVO A SARS-COV-2
4          NEGATIVO A SARS-COV-2
8    CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DIC...
13         CASO DE SARS-COV-2 CONFIRMADO
15         NEGATIVO A SARS-COV-2
Name: CLASIFICACION_FINAL, dtype: object
```

Ahora vamos a hacer una sustitución un poco más compleja, tenemos que encontrar todos los campos de tipo “SI - NO” y resolverlos (sustituir por valores que podamos manejar más fácil). Los campos que tienen este tipo de datos vienen en el excel de descriptores:

```
descriptores = pd.read_excel('datos/201128 Descriptores_.xlsx',
                             index_col='Nº',
                             engine='openpyxl')

descriptores
```

|    | NOMBRE DE VARIABLE  | DESCRIPCIÓN DE VARIABLE                           | FORMATO O F  |
|----|---------------------|---|--------------|
| Nº |                     |   |              |
| 1  | FECHA_ACTUALIZACION | La base de datos se alimenta diariamente, esta... | AAAA-MM-DD   |
| 2  | ID_REGISTRO         | Número identificador del caso                     | TEXTO        |
| 3  | ORIGEN              | La vigilancia centinela se realiza a través de... | CATÁLOGO: OI |
| 4  | SECTOR              | Identifica el tipo de institución del Sistema ... | CATÁLOGO: SE |
| 5  | ENTIDAD_UM          | Identifica la entidad donde se ubica la unidad... | CATALÓGO: EN |
| 6  | SEXO                | Identifica al sexo del paciente.                  | CATÁLOGO: SE |
| 7  | ENTIDAD_NAC         | Identifica la entidad de nacimiento del paciente. | CATALÓGO: EN |

| Nº | NOMBRE DE VARIABLE    | DESCRIPCIÓN DE VARIABLE                           | FORMATO O F   |
|----|-----------------------|---|---------------|
| 8  | ENTIDAD_RES           | Identifica la entidad de residencia del paciente. | CATÁLOGO: EN  |
| 9  | MUNICIPIO_RES         | Identifica el municipio de residencia del paci... | CATÁLOGO: M   |
| 10 | TIPO_PACIENTE         | Identifica el tipo de atención que recibió el ... | CATÁLOGO: TI  |
| 11 | FECHA_INGRESO         | Identifica la fecha de ingreso del paciente a ... | AAAA-MM-DD    |
| 12 | FECHA_SINTOMAS        | Identifica la fecha en que inició la sintomat...  | AAAA-MM-DD    |
| 13 | FECHA_DEF             | Identifica la fecha en que el paciente falleció.  | AAAA-MM-DD    |
| 14 | INTUBADO              | Identifica si el paciente requirió de intubación. | CATÁLOGO: SI  |
| 15 | NEUMONIA              | Identifica si al paciente se le diagnosticó co... | CATÁLOGO: SI  |
| 16 | EDAD                  | Identifica la edad del paciente.                  | NÚMERICA EN   |
| 17 | NACIONALIDAD          | Identifica si el paciente es mexicano o extran... | CATÁLOGO: NA  |
| 18 | EMBARAZO              | Identifica si la paciente está embarazada.        | CATÁLOGO: SI  |
| 19 | HABLA_LENGUA_INDIG    | Identifica si el paciente habla lengua indígena.  | CATÁLOGO: SI  |
| 20 | INDIGENA              | Identifica si el paciente se autoidentifica co... | CATÁLOGO: SI  |
| 21 | DIABETES              | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI  |
| 22 | EPOC                  | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI  |
| 23 | ASMA                  | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI  |
| 24 | INMUSUPR              | Identifica si el paciente presenta inmunosupre... | CATÁLOGO: SI  |
| 25 | HIPERTENSION          | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI  |
| 26 | OTRAS_COM             | Identifica si el paciente tiene diagnóstico de... | CATÁLOGO: SI  |
| 27 | CARDIOVASCULAR        | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI  |
| 28 | OBESIDAD              | Identifica si el paciente tiene diagnóstico de... | CATÁLOGO: SI  |
| 29 | RENAL_CRONICA         | Identifica si el paciente tiene diagnóstico de... | CATÁLOGO: SI  |
| 30 | TABAQUISMO            | Identifica si el paciente tiene hábito de taba... | CATÁLOGO: SI  |
| 31 | OTRO_CASO             | Identifica si el paciente tuvo contacto con al... | CATÁLOGO: SI  |
| 32 | TOMA_MUESTRA_LAB      | Identifica si al paciente se le tomó muestra d... | CATÁLOGO: SI  |
| 33 | RESULTADO_LAB         | Identifica el resultado del análisis de la mue... | CATÁLOGO: RE  |
| 34 | TOMA_MUESTRA_ANTIGENO | Identifica si al paciente se le tomó muestra d... | CATÁLOGO: SI  |
| 35 | RESULTADO_ANTIGENO    | Identifica el resultado del análisis de la mue... | CATÁLOGO: RE  |
| 36 | CLASIFICACION_FINAL   | Identifica si el paciente es un caso de COVID-... | CATÁLOGO: CI  |
| 37 | MIGRANTE              | Identifica si el paciente es una persona migra... | CATÁLOGO: SI  |
| 38 | PAIS_NACIONALIDAD     | Identifica la nacionalidad del paciente.          | TEXTO, 99= SE |
| 39 | PAIS_ORIGEN           | Identifica el país del que partió el paciente ... | TEXTO, 97= NO |
| 40 | UCI                   | Identifica si el paciente requirió ingresar a ... | CATÁLOGO: SI  |

Fíjense en alguno de estos campos en los datos:

```
df['OBESIDAD'].unique()
```

```
array(['2', '98', '1'], dtype=object)
```

Tenemos tres valores diferentes que corresponden (vean el diccionario) a SI, NO y NO ESPECIFICADO. Para todos los análisis que vamos a hacer en general sólo nos van a interesar los casos que **sabemos** que son SI, entonces lo que más nos conviene es codificar todos estos como binarios, es decir, sólo SI o NO. Además, podemos mejor decirles 1,0 respectivamente y así vamos a poder hacer cuentas mucho más fácil

De estos descriptores nos interesan los que tienen CATÁLOGO: SI\_ NO en el campo FORMATO O FUENTE. Para poder encontrar y sustituir de forma más sencilla y automática vamos a hacer un par de modificaciones a los datos:

- Reemplazar los espacios en los nombres de columnas por guiones bajos (para poder “hablarles” más fácil a las columnas)
- Quitar espacios al principio o al final de los valores de los campos (para asegurarnos de que siempre van a ser los mismos)

```
descriptores.columns = list(map(lambda col: col.replace(' ', '_'), descriptores.columns))
descriptores.head()
```

| Nº | NOMBRE_DE_VARIABLE  | DESCRIPCIÓN_DE_VARIABLE                           | FORMATO_O_FUENTE  |
|----|---------------------|---|-------------------|
| 1  | FECHA_ACTUALIZACION | La base de datos se alimenta diariamente, esta... | AAAA-MM-DD        |
| 2  | ID_REGISTRO         | Número identificador del caso                     | TEXTO             |
| 3  | ORIGEN              | La vigilancia centinela se realiza a través de... | CATÁLOGO: ORIGEN  |
| 4  | SECTOR              | Identifica el tipo de institución del Sistema ... | CATÁLOGO: SECTOR  |
| 5  | ENTIDAD_UM          | Identifica la entidad donde se ubica la unidad... | CATÁLOGO: ENTIDAD |

Poco a poco:

- `descriptores.columns` nos regresa (o les da valor, cuando está del lado izquierdo de un `=`) los nombres de las columnas del DataFrame
- `map(lambda col: col.replace(' ', '_'), descriptores.columns)` la función `map` regresa una asociación, como ya vimos. En este caso esta asociación se hace a través de una *función anónima* `lambda` que toma como argumento el nombre de una columna y regresa el mismo nombre pero con los espacios sustituidos por guines bajos

Al final, lo que hacemos es sustituir los nombres de las columnas por una lista hecha por nosotros, para que esto funcione la lista que pasamos debe ser de igual tamaño que la lista original de columnas.

Ahora vamos a hacer lo mismo pero con los valores de los campos:

```
descriptores['FORMATO_O_FUENTE'] = descriptores.FORMATO_O_FUENTE.str.strip()
descriptores['FORMATO_O_FUENTE'].head()
```

```
Nº
1          AAAA-MM-DD
2          TEXTO
3    CATÁLOGO: ORIGEN
4    CATÁLOGO: SECTOR
5    CATALÓGO: ENTIDADES
Name: FORMATO_O_FUENTE, dtype: object
```

Este fué más fácil. Fíjense cómo pedimos el campo del lado derecho: `descriptores.FORMATO_O_FUENTE`, esto es equivalente a `descriptores['FORMATO_O_FUENTE']` y los pueden usar indistintamente (claro, el primero sólo funciona si el nombre del campo no tiene espacios).

Filtremos ahora los descriptores para quedarnos sólo con los que nos interesan, para eso vamos a usar la función `query` de Pandas, que nos permite filtrar un DataFrame de forma conveniente usando una expresión *booleana*:

```
datos_si_no = descriptores.query('FORMATO_O_FUENTE == "CATÁLOGO: SI_NO"')
datos_si_no
```

| Nº | NOMBRE_DE_VARIABLE | DESCRIPCIÓN_DE_VARIABLE                           | FORMATO_O_FUENTE |
|----|--------------------|---|------------------|
| 14 | INTUBADO           | Identifica si el paciente requirió de intubación. | CATÁLOGO: SI_NO  |
| 15 | NEUMONIA           | Identifica si al paciente se le diagnosticó co... | CATÁLOGO: SI_NO  |
| 18 | EMBARAZO           | Identifica si la paciente está embarazada.        | CATÁLOGO: SI_NO  |
| 19 | HABLA_LENGUA_INDIG | Identifica si el paciente habla lengua indígena.  | CATÁLOGO: SI_NO  |
| 20 | INDIGENA           | Identifica si el paciente se autoidentifica co... | CATÁLOGO: SI_NO  |
| 21 | DIABETES           | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI_NO  |
| 22 | EPOC               | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI_NO  |
| 23 | ASMA               | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI_NO  |
| 24 | INMUSUPR           | Identifica si el paciente presenta inmunosupre... | CATÁLOGO: SI_NO  |
| 25 | HIPERTENSION       | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI_NO  |
| 26 | OTRAS_COM          | Identifica si el paciente tiene diagnóstico de... | CATÁLOGO: SI_NO  |
| 27 | CARDIOVASCULAR     | Identifica si el paciente tiene un diagnóstico... | CATÁLOGO: SI_NO  |
| 28 | OBESIDAD           | Identifica si el paciente tiene diagnóstico de... | CATÁLOGO: SI_NO  |
| 29 | RENAL_CRONICA      | Identifica si el paciente tiene diagnóstico de... | CATÁLOGO: SI_NO  |
| 30 | TABAQUISMO         | Identifica si el paciente tiene hábito de taba... | CATÁLOGO: SI_NO  |

| Nº | NOMBRE_DE_VARIABLE    | DESCRIPCIÓN_DE_VARIABLE                           | FORMATO_O_FUENTE |
|----|-----------------------|---|------------------|
| 31 | OTRO_CASO             | Identifica si el paciente tuvo contacto con al... | CATÁLOGO: SI     |
| 32 | TOMA_MUESTRA_LAB      | Identifica si al paciente se le tomó muestra d... | CATÁLOGO: SI     |
| 34 | TOMA_MUESTRA_ANTIGENO | Identifica si al paciente se le tomó muestra d... | CATÁLOGO: SI     |
| 37 | MIGRANTE              | Identifica si el paciente es una persona migra... | CATÁLOGO: SI     |
| 40 | UCI                   | Identifica si el paciente requirió ingresar a ... | CATÁLOGO: SI     |

Por si acaso, quitémosle también los espacios al campo FORMATO\_O\_FUENTE

```
descriptores['FORMATO_O_FUENTE'] = descriptores.FORMATO_O_FUENTE.str.strip()
```

Ahora sí, vamos a sustituir los valores como queremos en los datos originales. Para eso, lo primero que tenemos que hacer es fijarnos en el catálogo de estos campos:

```
cat_si_no = dict_catalogos['Catálogo SI_NO']
cat_si_no
```

|   | CLAVE | DESCRIPCIÓN     |
|---|-------|-----------------|
| 0 | 1     | SI              |
| 1 | 2     | NO              |
| 2 | 97    | NO APLICA       |
| 3 | 98    | SE IGNORA       |
| 4 | 99    | NO ESPECIFICADO |

Justo estos valores los queremos cambiar por claves binarias (acuérdense, para distinguirlos fácilmente). Entonces lo que necesitamos ahora es:

- Una lista de los nombres de los campos en donde vamos a hacer la sustitución
- Un mapeo de los valores con los que vamos a sustituir
- Hacer la sustitución primero en el diccionario y a partir de eso en los datos originales

```
# lista de los nombres de los campos
campos_si_no = datos_si_no.NOMBRE_DE_VARIABLE
# sustituimos en el catálogo de acuerdo a lo que nos interesa
cat_si_no['DESCRIPCIÓN'] = list(map(lambda val: 1 if val == 'SI' else 0, cat_si_no['DESCRIPCIÓN']))
# sustituimos en los datos originales
df[campos_si_no] = df[datos_si_no.NOMBRE_DE_VARIABLE].replace(
    to_replace=cat_si_no['CLAVE'].values,
```

```
df[campos_si_no]

value=cat_si_no['DESCRIPCIÓN'].values)
```

|         | INTUBADO | NEUMONIA | EMBARAZO | HABLA_LENGUA_INDIG | INDIGENA | DIABETES |
|---------|----------|----------|----------|--------------------|----------|----------|
| 1       | 0        | 0        | 0        | 0                  | 0        | 0        |
| 4       | 0        | 0        | 0        | 0                  | 0        | 0        |
| 8       | 0        | 0        | 0        | 0                  | 0        | 0        |
| 13      | 0        | 0        | 0        | 0                  | 0        | 0        |
| 15      | 0        | 0        | 0        | 0                  | 0        | 0        |
| ...     | ...      | ...      | ...      | ...                | ...      | ...      |
| 6393642 | 0        | 0        | 0        | 0                  | 0        | 0        |
| 6394417 | 0        | 0        | 0        | 0                  | 0        | 0        |
| 6394626 | 0        | 0        | 0        | 0                  | 0        | 0        |
| 6394988 | 0        | 0        | 0        | 0                  | 0        | 0        |
| 6395781 | 0        | 0        | 0        | 0                  | 0        | 0        |

Acá utilizamos para la última sustitución la función `replace` de Pandas que toma dos parámetros: la lista de valores a reemplazar y la lista de los valores de reemplazo. El reemplazo sucede elemento a elemento, es decir, se sustituye el primer elemento de la lista `to_replace` por el primer elemento de la lista `value` y así sucesivamente.

Hay más campos que podemos *aplanar* en la base de datos, como ejercicio pueden explorar algunos de ellos y sustituir como le hemos hecho aquí. Regresaremos a esto más adelante en el taller, pero por lo pronto nos vamos a mover a otra etapa del pre-procesamiento: el manejo de las fechas

## 2.2 Manejo de fechas

En Python las fechas son un tipo especial de datos, nosotros estamos acostumbrados a verlas como cadenas de caracteres: 20 de febrero de 2010, por ejemplo. Python puede hacer muchas cosas con las fechas, pero para eso tienen que estar codificados de la forma correcta.

En general el módulo `datetime` de Python provee las utilerías necesarias para manejar/transformar objetos del tipo fecha. Una de las cosas más útiles es transformar strings en objetos `datetime`:

```
datetime_object = datetime.strptime('Jun 1 2005 1:33PM', '%b %d %Y %I:%M%p')
datetime_object
```

```
datetime.datetime(2005, 6, 1, 13, 33)
```

Acá usamos un formato de fecha, '%b %d %Y %I:%M%p', para convertir el string 'Jun 1 2005 1:33PM'. De esa misma forma podemos especificar formatos diferentes:

```
datetime_object = datetime.strptime('06-01-2005 1:33PM', '%m-%d-%Y %I:%M%p')
datetime_object
```

```
datetime.datetime(2005, 6, 1, 13, 33)
```

Pandas tiene la *interfase* `to_datetime` para este tipo de operaciones que nos permite transformar campos de forma muy sencilla, por ejemplo, para transformar la columna `FECHA_INGRESO` de los datos originales en objetos de tipo `datetime` podemos hacer:

```
pd.to_datetime(df['FECHA_INGRESO'].head())
```

```
1    2022-01-19
4    2022-03-09
8    2022-02-20
13   2022-01-01
15   2022-06-28
Name: FECHA_INGRESO, dtype: datetime64[ns]
```

Veán la diferencia con el tipo de datos original:

```
df['FECHA_INGRESO'].head()
```

```
1    2022-01-19
4    2022-03-09
8    2022-02-20
13   2022-01-01
15   2022-06-28
Name: FECHA_INGRESO, dtype: object
```

Pandas *intenta* transformar los datos al tipo fecha usando formatos comunes. En general hace un buen trabajo, sin embargo, si nosotros conocemos el formato en el que están escritas las fechas, siempre es mejor ser explícito y usarlo para la transformación. En el caso de nuestros datos, el formato es: `%Y-%m-%d`, es decir, el año en cuatro caracteres, dos para el mes y dos para los días, separados por guiones medios. Para pasar el formato utilizamos la opción `format` de `pd.to_datetime()`



```
pd.to_datetime(df.FECHA_INGRESO, format="%Y-%m-%d")
```

```
1          2022-01-19
4          2022-03-09
8          2022-02-20
13         2022-01-01
15         2022-06-28
...
6393642    2022-01-23
6394417    2022-11-17
6394626    2022-11-16
6394988    2022-12-01
6395781    2022-12-16
Name: FECHA_INGRESO, Length: 1896084, dtype: datetime64[ns]
```

Aunque el resultado debería ser el mismo, ser explícito nos ayuda a entender mejor el código y a asegurarnos de que nuestros datos se comportan como nosotros esperamos. Por ejemplo, ¿qué sucedería si algún registro no se contiene datos en el formato que especificamos? Veamos el campo FECHA\_DEF que contiene registros *intencionalmente* inválidos.

```
pd.to_datetime(df['FECHA_DEF'], format="%Y-%m-%d")
```

```
ValueError: time data "9999-99-99" at position 0 doesn't match format specified
```

!Tenemos un ERROR! Pandas no puede transformar algunos datos utilizando el formato que le especificamos. En estos casos hay que especificar el comportamiento que *queremos* cuando Pandas encuentra una fecha que no se ajusta al formato. El comportamiento por defecto es arrojar una excepción, es decir, detenerse al encontrar un error y reportárnoslo. Eso puede resultar útil en algunos casos, sin embargo no en el nuestro en el que una fecha que no se ajusta al formato significa que el paciente no ha fallecido, es decir las fechas codificadas como 9999-99-99 corresponden a valores nulos en el campo. Para que pandas regrese un valor nulo cuando encuentre un error en la conversión de fechas, usamos la opción `coerce`:

```
pd.to_datetime(df['FECHA_DEF'], format="%Y-%m-%d", errors='coerce')
```

```
1          NaT
4          NaT
8         2022-02-21
```

```

13          NaT
15          NaT
...
6393642     NaT
6394417     NaT
6394626     NaT
6394988     NaT
6395781     NaT
Name: FECHA_DEF, Length: 1896084, dtype: datetime64[ns]

```

Ahora los registros que no se pueden convertir en fechas con el formato que especificamos regresan NaT (Not a Time) en lugar de error.

Una vez que entendimos las formas en las que queremos convertir las columnas con fechas, podemos transformar todas:

```

df['FECHA_INGRESO'] = pd.to_datetime(df['FECHA_INGRESO'], format="%Y-%m-%d")
df['FECHA_SINTOMAS'] = pd.to_datetime(df['FECHA_SINTOMAS'], format="%Y-%m-%d")
df['FECHA_DEF'] = pd.to_datetime(df['FECHA_DEF'], format="%Y-%m-%d", errors='coerce')
df[['FECHA_INGRESO', 'FECHA_SINTOMAS', 'FECHA_DEF']].head()

```

|    | FECHA_INGRESO | FECHA_SINTOMAS | FECHA_DEF  |
|----|---------------|----------------|------------|
| 1  | 2022-01-19    | 2022-01-17     | NaT        |
| 4  | 2022-03-09    | 2022-03-09     | NaT        |
| 8  | 2022-02-20    | 2022-02-13     | 2022-02-21 |
| 13 | 2022-01-01    | 2022-01-01     | NaT        |
| 15 | 2022-06-28    | 2022-06-28     | NaT        |

## 2.3 Exportar datos

Ya que tenemos procesados los datos, es muy posible que los queramos guardar para usarlos más adelante. La forma más sencilla de exportar los datos es guardarlos como un csv. Para esto Pandas tiene el método `to_csv`

```
df.to_csv("datos/covid_enero_2023_procesados.csv")
```

Hasta aquí hemos cubierto más o menos todo el pre-proceso de los datos. Claro no vimos todas las columnas, sólo nos fijamos en algunas, pero eso basta para darnos una buena idea de cómo se hacen las demás.

### **2.3.1 Tarea**

Sustituyan los valores de la columna TIPO\_PACIENTE por sus valores en el catálogo correspondiente

### 3 Automatización

En el taller anterior vimos toda una serie de pasos para preprocesar los datos de COVID-19. En esta actividad lo único que vamos a hacer es definir un par de funciones que realizan todo el flujo de preproceso. De esta forma podemos repetir todo el procedimiento de forma fácil.

```
import os # hablar con el sistema operativo
import glob # listar directorios y ese tipo de operaciones
import itertools # herramientas para iterar objetos
from pathlib import Path # manipular rutas a directorios
import zipfile # comprimir y descomprimir archivos
import numpy as np # operaciones vectorizadas
import pandas as pd # DataFrames
from datetime import timedelta, date, datetime # Manejar fechas
import openpyxl # leer/escribir archivos de excel
import requests # Hablar con direcciones web
import logging
```

En el taller anterior bajamos los datos directamente del sitio de la Secretaría de Salud, ahora vamos a automatizar el proceso de descarga de datos de forma que, desde Python, podamos descargar los datos y asegurarnos de que tenemos la última versión disponible.

Descargar y guardar archivos en Python es relativamente sencillo, vamos a usar tres módulos de la distribución base de Python:

- **os**. Este módulo provee herramientas para interactuar con el sistema operativo. La vamos a usar para construir los *paths* en donde vamos a guardar los datos y preguntar si el archivo ya existe.
- **requests**. Esta librería provee diferentes formas de interactuar con el protocolo HTTP. La vamos a usar para hacer las *peticiones* a la página y procesar la respuesta.
- **zipfile**. Esta librería sirve para trabajar con archivos comprimidos en formato *zip*. En nuestro caso la usaremos para descomprimir los diccionarios.

La parte complicada de entender es el uso de **requests** para comunicarse con la página en donde están los datos.

```
r = requests.get("https://www.centrogeo.org.mx/")
r.content[0:500]
```

```
b'\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n!DOCTYPE html>\r\n<html lang="es-es" dir="ltr" class=\'con
```

Como ven, una *petición* de tipo *get* simplemente nos regresa, a través de la propiedad *content*, el contenido de la respuesta del servidor. En el caso de la página de CentroGeo, el contenido es el HTML de la página (que podríamos ver mejor con un browser), pero en el caso de que la dirección apunte a un archivo de descarga, el contenido es el *stream* de datos del archivo. Este *stream* de datos lo podemos usar como entrada para escribir un archivo utilizando la función [open](#).

La función `open` va a tomar como entrada el *path* en donde queremos guardar el archivo, este *path* puede ser simplemente una cadena de caracteres, sin embargo esto haría que nuestro código no fuera interoperable entre sistemas operativos, entonces, en lugar de escribir el *path* como cadena de caracteres, vamos a escribirlo como un *objeto* de `os`:

```
os.path.join("datos", "datos_covid.zip")
```

```
'datos/datos_covid.zip'
```

Esta forma de construir el *path* nos asegura que va a funcionar en cualquier sistema operativo.

Ya con estas explicaciones, podemos escribir la función que descarga los datos:

```
def bajar_datos_salud(directorio_datos='data/'):
    """
    Descarga el ultimo archivo disponible en datos abiertos y los diccionarios correspondientes
    """
    fecha_descarga = datetime.now().date()
    url_datos = 'https://datosabiertos.salud.gob.mx/gobmx/salud/datos_abiertos/datos_abiertos/'
    archivo_nombre = f'{fecha_descarga.strftime("%y%m%d")}.COVID19MEXICO.csv.zip'
    archivo_ruta = os.path.join(directorio_datos, archivo_nombre)
    url_diccionario = 'https://datosabiertos.salud.gob.mx/gobmx/salud/datos_abiertos/diccionarios/'
    diccionario_ruta = os.path.join(directorio_datos, 'diccionario.zip')
    if os.path.exists(archivo_ruta):
        logging.debug(f'Ya existe {archivo_nombre}')
    else:
        print(f'Bajando datos...')
        r = requests.get(url_datos, allow_redirects=True)
```

```

open(archivo_ruta, 'wb').write(r.content)
r = requests.get(url_diccionario, allow_redirects=True)
open(diccionario_ruta, 'wb').write(r.content)
with zipfile.ZipFile(diccionario_ruta, 'r') as zip_ref:
    zip_ref.extractall(directorio_datos)

```

Para utilizar la función hacemos:

```
bajar_datos_salud('datos/')
```

Bajando datos...

### 3.1 Preproceso

Ahora, ya que tenemos los datos descargados, vamos a *empaquetar* en una función el flujo de preproceso que trabajamos en el taller anterior. Esta función va a tomar como entrada la carpeta en donde se encuentran los datos y diccionarios y el nombre del archivo de datos que queremos procesar. Toma dos parámetros adicionales, uno para decidir si queremos resolver o no las claves binarias y otro para definir la entidad que queremos procesar.

```

def carga_datos_covid19_MX(data_dir='datos/', archivo='datos_abiertos_covid19.zip', resolver=
"""
    Lee en un DataFrame el CSV con el reporte de casos de la Secretaría de Salud de Mé
    también lee el diccionario de datos que acompaña a estas publicaciones para prepar
    de generar columnas binarias para datos con valores 'SI', 'No'.

    **Nota 2**: Por las actualizaciones a los formatos de datos, esta función sólo va

    resolver_claves: 'sustitucion', 'agregar', 'si_no_binarias', 'solo_localidades'. R
    diccionario de datos y los catálogos. 'sustitucion' reemplaza los valores en las co
    crea nuevas columnas. 'si_no_binarias' cambia valores SI, NO, No Aplica, SE IGNORA

"""
    catalogo_nombre = '201128 Catalogos.xlsx'
    catalogo_path = os.path.join(data_dir, catalogo_nombre)
    descriptores_nombre = '201128 Descriptores.xlsx'
    descriptores_path = os.path.join(data_dir, descriptores_nombre)
    data_file = os.path.join(data_dir, archivo)
    print(data_file)

```

```

df = pd.read_csv(data_file, dtype=object, encoding='latin-1')
if entidad is not None:
    df = df[df['ENTIDAD_RES'] == entidad]
# Hay un error y el campo OTRA_COMP es OTRAS_COMP según los descriptores
df.rename(columns={'OTRA_COM': 'OTRAS_COM'}, inplace=True)
# Asignar clave única a municipios
df['MUNICIPIO_RES'] = df['ENTIDAD_RES'] + df['MUNICIPIO_RES']
df['CLAVE_MUNICIPIO_RES'] = df['MUNICIPIO_RES']
# Leer catálogos
nombres_catálogos = ['Catálogo de ENTIDADES',
                      'Catálogo MUNICIPIOS',
                      'Catálogo RESULTADO',
                      'Catálogo SI_NO',
                      'Catálogo TIPO_PACIENTE']
nombres_catálogos.append('Catálogo CLASIFICACION_FINAL')
nombres_catálogos[2] = 'Catálogo RESULTADO_LAB'

dict_catálogos = pd.read_excel(catalogo_path,
                               nombres_catálogos,
                               dtype=str,
                               engine='openpyxl')

entidades = dict_catálogos[nombres_catálogos[0]]
municipios = dict_catálogos[nombres_catálogos[1]]
tipo_resultado = dict_catálogos[nombres_catálogos[2]]
cat_si_no = dict_catálogos[nombres_catálogos[3]]
cat_tipo_pac = dict_catálogos[nombres_catálogos[4]]
# Arreglar los catálogos que tienen mal las primeras líneas
dict_catálogos[nombres_catálogos[2]].columns = ["CLAVE", "DESCRIPCIÓN"]
dict_catálogos[nombres_catálogos[5]].columns = ["CLAVE", "CLASIFICACIÓN", "DESCRIPCIÓN"]

clasificacion_final = dict_catálogos[nombres_catálogos[5]]

# Resolver códigos de entidad federal
cols_entidad = ['ENTIDAD_RES', 'ENTIDAD_UM', 'ENTIDAD_NAC']
df['CLAVE_ENTIDAD_RES'] = df['ENTIDAD_RES']
df[cols_entidad] = df[cols_entidad].replace(to_replace=entidades['CLAVE_ENTIDAD'].value,
                                           value=entidades['ENTIDAD_FEDERATIVA'].value)

```

```

# Construye clave unica de municipios de catálogo para resolver nombres de municipio
municipios['CLAVE_MUNICIPIO'] = municipios['CLAVE_ENTIDAD'] + municipios['CLAVE_MUNICIPIO']

# Resolver códigos de municipio
municipios_dict = dict(zip(municipios['CLAVE_MUNICIPIO'], municipios['MUNICIPIO']))
df['MUNICIPIO_RES'] = df['MUNICIPIO_RES'].map(municipios_dict.get)

# Resolver resultados

df.rename(columns={'RESULTADO_LAB': 'RESULTADO'}, inplace=True)
tipo_resultado['DESCRIPCIÓN'].replace({'POSITIVO A SARS-COV-2': 'Positivo SARS-CoV-2'})

tipo_resultado = dict(zip(tipo_resultado['CLAVE'], tipo_resultado['DESCRIPCIÓN']))
df['RESULTADO'] = df['RESULTADO'].map(tipo_resultado.get)
clasificacion_final = dict(zip(clasificacion_final['CLAVE'], clasificacion_final['CLASIFICACION_FINAL']))
df['CLASIFICACION_FINAL'] = df['CLASIFICACION_FINAL'].map(clasificacion_final.get)
# Resolver datos SI - NO

# Necesitamos encontrar todos los campos que tienen este tipo de dato y eso
# viene en los descriptores, en el campo FORMATO_O_FUENTE
descriptores = pd.read_excel(f'{data_dir}201128 Descriptores.xlsx',
                             index_col='Nº',
                             engine='openpyxl')
descriptores.columns = list(map(lambda col: col.replace(' ', '_'), descriptores.columns))
descriptores['FORMATO_O_FUENTE'] = descriptores.FORMATO_O_FUENTE.str.strip()

datos_si_no = descriptores.query('FORMATO_O_FUENTE == "CATÁLOGO: SI_ NO"')
cat_si_no['DESCRIPCIÓN'] = cat_si_no['DESCRIPCIÓN'].str.strip()

campos_si_no = datos_si_no.NOMBRE_DE_VARIABLE
nuevos_campos_si_no = campos_si_no

if resolver_claves == 'agregar':
    nuevos_campos_si_no = [nombre_var + '_NOM' for nombre_var in campos_si_no]
elif resolver_claves == 'si_no_binarias':
    nuevos_campos_si_no = [nombre_var + '_BIN' for nombre_var in campos_si_no]
    cat_si_no['DESCRIPCIÓN'] = list(map(lambda val: 1 if val == 'SI' else 0, cat_si_no['DESCRIPCIÓN']))

df[nuevos_campos_si_no] = df[datos_si_no.NOMBRE_DE_VARIABLE].replace(
    to_replace=cat_si_no['CLAVE'].values,
    value=cat_si_no['DESCRIPCIÓN'].values)

```



```

# Resolver tipos de paciente
cat_tipo_pac = dict(zip(cat_tipo_pac['CLAVE'], cat_tipo_pac['DESCRIPCIÓN']))
df['TIPO_PACIENTE'] = df['TIPO_PACIENTE'].map(cat_tipo_pac.get)

df = procesa_fechas(df)

return df

def procesa_fechas(covid_df):
    df = covid_df.copy()
    df['FECHA_INGRESO'] = pd.to_datetime(df['FECHA_INGRESO'], format="%Y-%m-%d")
    df['FECHA_SINTOMAS'] = pd.to_datetime(df['FECHA_SINTOMAS'], format="%Y-%m-%d")
    df['FECHA_DEF'] = pd.to_datetime(df['FECHA_DEF'], format="%Y-%m-%d", errors='coerce')
    df['DEFUNCION'] = (df['FECHA_DEF'].notna()).astype(int)
    df['EDAD'] = df['EDAD'].astype(int)
    return df

```

## 3.2 Preprocesar usando nuestras funciones

```

df = carga_datos_covid19_MX(entidad='09')
df

```

datos/datos\_abiertos\_covid19.zip

|         | FECHA_ACTUALIZACION | ID_REGISTRO | ORIGEN | SECTOR | ENTIDAD_UM       |
|---------|---------------------|-------------|--------|--------|------------------|
| 1       | 2023-01-03          | 180725      | 2      | 9      | CIUDAD DE MÉXICO |
| 4       | 2023-01-03          | 1933c0      | 1      | 12     | CIUDAD DE MÉXICO |
| 8       | 2023-01-03          | 0741e4      | 2      | 6      | CIUDAD DE MÉXICO |
| 13      | 2023-01-03          | 1c4d2e      | 2      | 9      | CIUDAD DE MÉXICO |
| 15      | 2023-01-03          | 0a6cd6      | 2      | 6      | CIUDAD DE MÉXICO |
| ...     | ...                 | ...         | ...    | ...    | ...              |
| 6393642 | 2023-01-03          | m1cd235     | 2      | 12     | MÉXICO           |
| 6394417 | 2023-01-03          | m0dbc4c     | 2      | 12     | MÉXICO           |
| 6394626 | 2023-01-03          | m13431e     | 2      | 12     | MÉXICO           |
| 6394988 | 2023-01-03          | m1493ea     | 2      | 12     | MÉXICO           |
| 6395781 | 2023-01-03          | m0a22b8     | 2      | 12     | MÉXICO           |

### 3.3 Guardando el resultado

Listo, con nuestras funciones tenemos ya nuestros datos preprocesados, ahora vamos a guardarlos para poder utilizarlos rápidamente en otros notebooks. En general tenemos muchas opciones para guardar los datos, csv, por ejemplo. En esta ocasión vamos a usar un formato nativo de Python el [pickle](#), que es una forma de *serializar* un objeto de Python. Pandas nos provee una función para guardar directamente un dataframe como pickle:

```
df.to_pickle("data/datos_covid_ene19.pkl")
```

En la documentación de [to\\_pickle](#) pueden ver las opciones completas.

```
df.to_csv("datos/covid_enero_2023_procesados.csv")
```

## 4 Curvas epidémicas

Ahora que ya tenemos idea de cómo procesar los datos y conocemos más o menos su contenido, podemos empezar a hacer algunas visualizaciones. En general, el flujo de análisis de datos va y viene entre el preproceso, la visualización y el análisis. El tipo de preproceso va ligado a las visualizaciones que queremos hacer y estas van mostrando el camino del análisis, por lo tanto en la vida real uno siempre está yendo y viniendo entre estas etapas.

En este taller vamos a explorar gráficamente los datos y empezarnos a hacer preguntas sobre ellos, preguntas que intentaremos ir respondiendo con nuevas visualizaciones. Para ello vamos a partir de la base de datos que construimos en el taller anterior.

Como siempre, lo primero es importar las librerías que vamos a utilizar

```
import plotly.express as px
import pandas as pd
import seaborn as sns
from datetime import timedelta, date, datetime
```

Vamos a utilizar los datos preprocesados que guardamos al final del taller anterior. Recordemos que los guardamos como csv, entonces los datos van a incluir el proceso que implique cambiar valores, pero no van a recordar los tipos de datos que cambiamos, entonces, después de leer los datos, vamos a ajustar los tipos de las columnas de fecha.

```
df = pd.read_csv("datos/covid_enero_2023_procesados.csv", index_col=0)
df['FECHA_INGRESO'] = pd.to_datetime(df['FECHA_INGRESO'], format="%Y-%m-%d")
df['FECHA_SINTOMAS'] = pd.to_datetime(df['FECHA_SINTOMAS'], format="%Y-%m-%d")
df['FECHA_DEF'] = pd.to_datetime(df['FECHA_DEF'], format="%Y-%m-%d", errors='coerce')
df
```

/tmp/ipykernel\_89594/2932843985.py:1: DtypeWarning:

Columns (13) have mixed types. Specify dtype option on import or set low\_memory=False.

|         | FECHA_ACTUALIZACION | ID_REGISTRO | ORIGEN | SECTOR | ENTIDAD_UM       |
|---------|---------------------|-------------|--------|--------|------------------|
| 1       | 2023-01-03          | 180725      | 2      | 9      | CIUDAD DE MÉXICO |
| 4       | 2023-01-03          | 1933c0      | 1      | 12     | CIUDAD DE MÉXICO |
| 8       | 2023-01-03          | 0741e4      | 2      | 6      | CIUDAD DE MÉXICO |
| 13      | 2023-01-03          | 1c4d2e      | 2      | 9      | CIUDAD DE MÉXICO |
| 15      | 2023-01-03          | 0a6cd6      | 2      | 6      | CIUDAD DE MÉXICO |
| ...     | ...                 | ...         | ...    | ...    | ...              |
| 6393642 | 2023-01-03          | m1cd235     | 2      | 12     | MÉXICO           |
| 6394417 | 2023-01-03          | m0dbc4c     | 2      | 12     | MÉXICO           |
| 6394626 | 2023-01-03          | m13431e     | 2      | 12     | MÉXICO           |
| 6394988 | 2023-01-03          | m1493ea     | 2      | 12     | MÉXICO           |
| 6395781 | 2023-01-03          | m0a22b8     | 2      | 12     | MÉXICO           |

#### Note

El parámetro `index_col=0` le dice al lector que la primera columna (que no tiene nombre en el csv) es el índice de los datos.

Lo primero que haremos será el desarrollo de *Curvas Epidémicas* es decir, la evolución temporal de los casos confirmados y las defunciones. Si consultamos los diccionarios de datos, podemos ver que los casos confirmados para COVID-19 corresponden a 3 categorías de la columna clasificación final:

1. CASO DE COVID-19 CONFIRMADO POR ASOCIACIÓN CLÍNICA EPIDEMIOLÓGICA
2. CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DICTAMINACIÓN
3. CASO DE SARS-COV-2 CONFIRMADO

mientras que las defunciones corresponden a todos aquellos registros que tengan una fecha de defunción válida, es decir, en nuestros datos preprocesados, todas las fechas válidas.

### 4.0.1 Curva de casos confirmados

El primer paso es extraer las filas que corresponden a casos confirmados

```
df.CLASIFICACION_FINAL.unique()
```

```
array(['NEGATIVO A SARS-COV-2',  
      'CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DICTAMINACIÓN',
```

```
'CASO DE SARS-COV-2 CONFIRMADO', 'CASO SOSPECHOSO',
'NO REALIZADO POR LABORATORIO',
'CASO DE COVID-19 CONFIRMADO POR ASOCIACIÓN CLÍNICA EPIDEMIOLÓGICA',
'INVÁLIDO POR LABORATORIO'], dtype=object)
```

A partir de estos valores podemos seleccionar las filas que queremos

```
valores_confirmados = ['CASO DE COVID-19 CONFIRMADO POR ASOCIACIÓN CLÍNICA EPIDEMIOLÓGICA',
                        'CASO DE COVID-19 CONFIRMADO POR COMITÉ DE DICTAMINACIÓN',
                        'CASO DE SARS-COV-2 CONFIRMADO']
confirmados = df.loc[df['CLASIFICACION_FINAL'].isin(valores_confirmados)]
confirmados.head()
```

|    | FECHA_ACTUALIZACION | ID_REGISTRO | ORIGEN | SECTOR | ENTIDAD_UM       | SEXO |
|----|---------------------|-------------|--------|--------|------------------|------|
| 8  | 2023-01-03          | 0741e4      | 2      | 6      | CIUDAD DE MÉXICO | 2    |
| 13 | 2023-01-03          | 1c4d2e      | 2      | 9      | CIUDAD DE MÉXICO | 1    |
| 25 | 2023-01-03          | 0a98b4      | 2      | 12     | CIUDAD DE MÉXICO | 1    |
| 42 | 2023-01-03          | 13cf10      | 2      | 9      | CIUDAD DE MÉXICO | 1    |
| 54 | 2023-01-03          | 0fef08      | 1      | 12     | CIUDAD DE MÉXICO | 2    |

Ahora tenemos una tabla con todos los casos confirmados, para hacer una curva epidémica, tenemos que agregar en una escala temporal. Lo más sencillo es primero agregar por día y a partir de ahí podemos construir agregados para cualquier intervalo que queramos.

Necesitamos decidir cuál fecha de todas las disponibles vamos a utilizar para agregar los casos. En este caso, la DGE sugiere utilizar la fecha de inicio de síntomas (FECHA\_SINTOMAS) para construir la curva de casos confirmados y la de defunción (FECHA\_DEF) para la curva de defunciones.

Entonces, para construir la curva de confirmados lo primero que tenemos que hacer es indexar el DataFrame por la fecha de inicio de síntomas

```
confirmados = confirmados.set_index('FECHA_SINTOMAS')
confirmados.index
```

```
DatetimeIndex(['2022-02-13', '2022-01-01', '2022-04-22', '2022-08-07',
               '2022-01-10', '2022-01-14', '2022-12-01', '2022-05-25',
               '2022-12-23', '2022-08-02',
               ...,
               '2022-06-23', '2022-08-16', '2022-08-19', '2022-08-01',
```

```
'2022-07-05', '2022-07-08', '2022-09-05', '2022-06-19',
'2022-06-20', '2022-09-23'],
dtype='datetime64[ns]', name='FECHA_SINTOMAS', length=769894, freq=None)
```

Ya con los datos indexados es fácil construir agregados diarios, sólo tenemos que seleccionar qué columnas queremos agregar. Por lo pronto hagamos un conteo sólo de casos confirmados. Para eso sólo tenemos que agrupar el índice usando una frecuencia diaria y tomar el tamaño de los grupos (de alguna columna, realmente no importa cual).

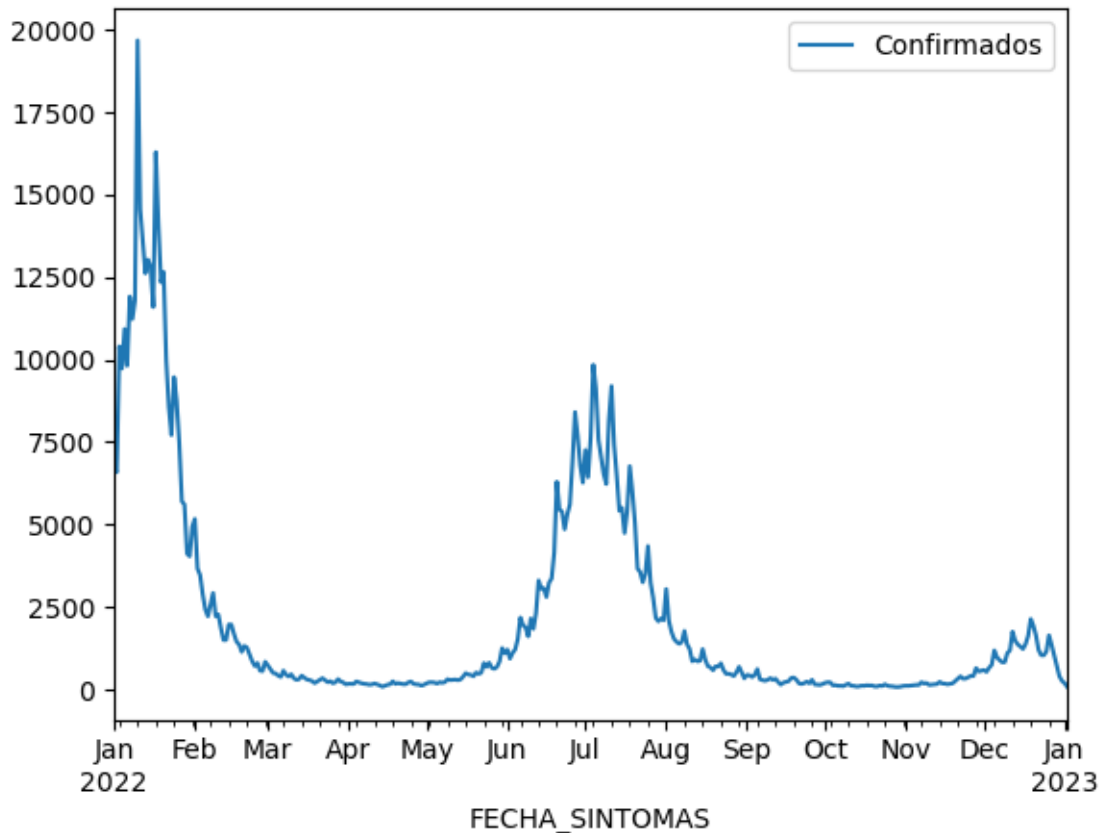
```
confirmados_diarios = (confirmados
                        .groupby(pd.Grouper(freq='D'))[['ID_REGISTRO']] # grupos por dia y
                        .size() # Calculamos el tamaño de cada grupo
                        .reset_index() # Convertimos el resultado (que es una serie) en Dat
                        .rename({0:'Confirmados'}, axis=1) # Le damos nombre a la columna q
                        )
confirmados_diarios
```

|     | FECHA_SINTOMAS | Confirmados |
|-----|----------------|-------------|
| 0   | 2022-01-01     | 6748        |
| 1   | 2022-01-02     | 6585        |
| 2   | 2022-01-03     | 10398       |
| 3   | 2022-01-04     | 9729        |
| 4   | 2022-01-05     | 10924       |
| ... | ...            | ...         |
| 362 | 2022-12-29     | 715         |
| 363 | 2022-12-30     | 399         |
| 364 | 2022-12-31     | 258         |
| 365 | 2023-01-01     | 187         |
| 366 | 2023-01-02     | 80          |

Hay muchas formas de visualizar estos datos, la primera y más sencilla es utilizar los métodos que provee Pandas, por ejemplo:

```
confirmados_diarios.set_index('FECHA_SINTOMAS').plot()
```

```
<AxesSubplot: xlabel='FECHA_SINTOMAS'>
```



Una alternativa que nos provee herramientas interactivas para visualizar los datos y que es muy fácil de usar es [Plotly](#). A través del módulo [Plotly express](#) podemos crear de forma muy simple gráficas que nos permitan interactuar con ellas.

```
fig = px.line(confirmados_diarios, x='FECHA_SINTOMAS', y="Confirmados")
fig.show()
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json

Como pueden ver, fue muy simple hacer una gráfica con herramientas para *pan* y *zoom*. Estas herramientas hacen más fácil ver que los datos de casos confirmados contienen la *mezcla* de dos señales: una de alta frecuencia que representa la variación diaria, con una especie de periodicidad semanal y una señal de baja frecuencia que contiene las *olas epidémicas*.

La señal de alta frecuencia contiene mucho *ruido* que corresponde a los ciclos de actualización de la información y que realmente nos dice poco de la tendencia de los datos. Una forma

sencilla de filtrar este ruido es utilizando la [media móvil](#). Para calcular este promedio, Pandas provee la función [rolling](#)

```
confirmados_diarios['Media Móvil'] = (confirmados_diarios
                                       .rolling(window=7)
                                       .mean())

confirmados_diarios.head(10)
```

/tmp/ipykernel\_89594/2896601829.py:3: FutureWarning:

Dropping of nuisance columns in rolling operations is deprecated; in a future version this w

|   | FECHA_SINTOMAS | Confirmados | Media Móvil  |
|---|----------------|-------------|--------------|
| 0 | 2022-01-01     | 6748        | NaN          |
| 1 | 2022-01-02     | 6585        | NaN          |
| 2 | 2022-01-03     | 10398       | NaN          |
| 3 | 2022-01-04     | 9729        | NaN          |
| 4 | 2022-01-05     | 10924       | NaN          |
| 5 | 2022-01-06     | 9816        | NaN          |
| 6 | 2022-01-07     | 11910       | 9444.285714  |
| 7 | 2022-01-08     | 11229       | 10084.428571 |
| 8 | 2022-01-09     | 11794       | 10828.571429 |
| 9 | 2022-01-10     | 19673       | 12153.571429 |

Y ahora la podemos graficar

```
fig = px.line(confirmados_diarios, x='FECHA_SINTOMAS', y='Media Móvil')
fig.show()
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json

Para graficar las dos series en la misma gráfica lo más sencillo es pasar los datos del formato ancho (en columnas) al formato largo (en filas con una columna que los distinga). Para esto vamos a usar la función [melt](#) de Pandas

```
confirmados_diarios = confirmados_diarios.melt(id_vars=['FECHA_SINTOMAS'], value_vars=['C', 'D'])
confirmados_diarios
```



|     | FECHA_SINTOMAS | variable    | value        |
|-----|----------------|-------------|--------------|
| 0   | 2022-01-01     | Confirmados | 6748.000000  |
| 1   | 2022-01-02     | Confirmados | 6585.000000  |
| 2   | 2022-01-03     | Confirmados | 10398.000000 |
| 3   | 2022-01-04     | Confirmados | 9729.000000  |
| 4   | 2022-01-05     | Confirmados | 10924.000000 |
| ... | ...            | ...         | ...          |
| 729 | 2022-12-29     | Media Móvil | 1137.142857  |
| 730 | 2022-12-30     | Media Móvil | 1045.571429  |
| 731 | 2022-12-31     | Media Móvil | 934.571429   |
| 732 | 2023-01-01     | Media Móvil | 796.714286   |
| 733 | 2023-01-02     | Media Móvil | 573.714286   |

Con los datos de esta forma, ahora podemos usar Plotly para graficar ambas variables utilizando como color la columna `variable`. El parámetro `color` nos permite separar dos series de datos cuando estas vienen en formato largo.

```
fig = px.line(confirmados_diarios, x='FECHA_SINTOMAS', y='value', color='variable')
fig.show()
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json

#### 4.0.2 Curva de defunciones

Ya que construimos la curva de casos confirmados, la de defunciones es exactamente igual, sólo necesitamos seleccionar al inicio del proceso los renglones que tengan una fecha de defunción válida e indexar por fecha de defunción

```
defunciones = confirmados.loc[confirmados['FECHA_DEF'].notnull()] # Seleccionamos los casos con fecha de defunción
defunciones = defunciones.set_index('FECHA_DEF') # indexamos por fecha de defunción
defunciones_diarios = (defunciones
    .groupby(pd.Grouper(freq='D'))[['ID_REGISTRO']] # grupos por día y variable
    .size() # Calculamos el tamaño de cada grupo
    .reset_index() # Convertimos el resultado (que es una serie) en DataFrame
    .rename({0:'Defunciones'}, axis=1) # Le damos nombre a la columna que representa el tamaño del grupo
)
defunciones_diarios['Media Móvil'] = defunciones_diarios.rolling(window=7).mean() # Calculamos la media móvil de 7 días
defunciones_diarios = defunciones_diarios.melt(id_vars=['FECHA_DEF'], value_vars=['Defunciones'])
fig = px.line(defunciones_diarios, x='FECHA_DEF', y='value', color='variable')
```

```
fig.show()
```

```
/tmp/ipykernel_89594/3052884713.py:9: FutureWarning:
```

```
Dropping of nuisance columns in rolling operations is deprecated; in a future version this w
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json
```

### 4.0.3 Combinando las dos gráficas

Para entender la evolución de la epidemia conviene poder ver las dos gráficas al mismo tiempo y explorarlas de forma conjunta. En general este tipo de combinaciones en las que dos o más gráficas comparten por lo menos un eje (en nuestro caso el tiempo) se llaman *Facetas*. Plotly nos permite crear este tipo de visualizaciones de forma muy sencilla, lo que necesitamos es combinar ambos datos (casos y defunciones) en un sólo DataFrame en formato largo y asegurarnos de que cada fila pueda distinguir a qué se refiere. En nuestros datos vamos a tener cuatro series diferentes: datos crudos y media móvil para casos y defunciones.

Comencemos con la serie de defunciones, lo primero que tenemos que hacer es agregar una columna con el tipo de serie, es decir, defunciones:

```
defunciones_diarios['Tipo'] = 'Defunciones'
defunciones_diarios
```

|     | FECHA_DEF  | variable    | value    | Tipo        |
|-----|------------|-------------|----------|-------------|
| 0   | 2022-01-03 | Defunciones | 1.000000 | Defunciones |
| 1   | 2022-01-04 | Defunciones | 2.000000 | Defunciones |
| 2   | 2022-01-05 | Defunciones | 1.000000 | Defunciones |
| 3   | 2022-01-06 | Defunciones | 3.000000 | Defunciones |
| 4   | 2022-01-07 | Defunciones | 8.000000 | Defunciones |
| ... | ...        | ...         | ...      | ...         |
| 721 | 2022-12-27 | Media Móvil | 4.714286 | Defunciones |
| 722 | 2022-12-28 | Media Móvil | 5.285714 | Defunciones |
| 723 | 2022-12-29 | Media Móvil | 5.285714 | Defunciones |
| 724 | 2022-12-30 | Media Móvil | 5.285714 | Defunciones |
| 725 | 2022-12-31 | Media Móvil | 4.571429 | Defunciones |

En la columna variable tenemos los valores **Defunciones** y **Media Móvil**, necesitamos cambiar el valor de **Defunciones** por algo que sea compatible con tener los casos confirmados en el

mismo DataFrame, pienses que al combinar ambas series queremos tener sólo dos valores diferentes en esta columna. Cambiemos entonces el valor de Defunciones por Conteo:

```
defunciones_diarios.loc[defunciones_diarios['variable'] == 'Defunciones', 'variable'] = 'Conteo'
defunciones_diarios
```

|     | FECHA_DEF  | variable    | value    | Tipo        |
|-----|------------|-------------|----------|-------------|
| 0   | 2022-01-03 | Conteo      | 1.000000 | Defunciones |
| 1   | 2022-01-04 | Conteo      | 2.000000 | Defunciones |
| 2   | 2022-01-05 | Conteo      | 1.000000 | Defunciones |
| 3   | 2022-01-06 | Conteo      | 3.000000 | Defunciones |
| 4   | 2022-01-07 | Conteo      | 8.000000 | Defunciones |
| ... | ...        | ...         | ...      | ...         |
| 721 | 2022-12-27 | Media Móvil | 4.714286 | Defunciones |
| 722 | 2022-12-28 | Media Móvil | 5.285714 | Defunciones |
| 723 | 2022-12-29 | Media Móvil | 5.285714 | Defunciones |
| 724 | 2022-12-30 | Media Móvil | 5.285714 | Defunciones |
| 725 | 2022-12-31 | Media Móvil | 4.571429 | Defunciones |

Ahora tenemos dos nombres diferentes para los campos con los que vamos a construir el eje de las X: FECHA\_DEF y FECHA\_SINTOMAS. Para poder combinar ambas series en una sola, necesitamos que esos campos tengan el mismo nombre en las dos series.

```
defunciones_diarios = defunciones_diarios.rename({'FECHA_DEF': 'Fecha'}, axis=1)
defunciones_diarios
```

|     | Fecha      | variable    | value    | Tipo        |
|-----|------------|-------------|----------|-------------|
| 0   | 2022-01-03 | Conteo      | 1.000000 | Defunciones |
| 1   | 2022-01-04 | Conteo      | 2.000000 | Defunciones |
| 2   | 2022-01-05 | Conteo      | 1.000000 | Defunciones |
| 3   | 2022-01-06 | Conteo      | 3.000000 | Defunciones |
| 4   | 2022-01-07 | Conteo      | 8.000000 | Defunciones |
| ... | ...        | ...         | ...      | ...         |
| 721 | 2022-12-27 | Media Móvil | 4.714286 | Defunciones |
| 722 | 2022-12-28 | Media Móvil | 5.285714 | Defunciones |
| 723 | 2022-12-29 | Media Móvil | 5.285714 | Defunciones |
| 724 | 2022-12-30 | Media Móvil | 5.285714 | Defunciones |
| 725 | 2022-12-31 | Media Móvil | 4.571429 | Defunciones |

Repetimos el proceso para los datos de casos confirmados:

```
confirmados_diarios['Tipo'] = 'Casos Confirmados'
confirmados_diarios.loc[confirmados_diarios['variable'] == 'Confirmados', 'variable'] = 'C
confirmados_diarios = confirmados_diarios.rename({'FECHA_SINTOMAS': 'Fecha'}, axis=1)
confirmados_diarios
```

|     | Fecha      | variable    | value        | Tipo              |
|-----|------------|-------------|--------------|-------------------|
| 0   | 2022-01-01 | Conteo      | 6748.000000  | Casos Confirmados |
| 1   | 2022-01-02 | Conteo      | 6585.000000  | Casos Confirmados |
| 2   | 2022-01-03 | Conteo      | 10398.000000 | Casos Confirmados |
| 3   | 2022-01-04 | Conteo      | 9729.000000  | Casos Confirmados |
| 4   | 2022-01-05 | Conteo      | 10924.000000 | Casos Confirmados |
| ... | ...        | ...         | ...          | ...               |
| 729 | 2022-12-29 | Media Móvil | 1137.142857  | Casos Confirmados |
| 730 | 2022-12-30 | Media Móvil | 1045.571429  | Casos Confirmados |
| 731 | 2022-12-31 | Media Móvil | 934.571429   | Casos Confirmados |
| 732 | 2023-01-01 | Media Móvil | 796.714286   | Casos Confirmados |
| 733 | 2023-01-02 | Media Móvil | 573.714286   | Casos Confirmados |

Ahora que ambas series tienen la misma *forma* y columnas que distinguen los cuatro casos que nos interesan, sólo resta combinar las series. En este caso, lo que queremos es *pegar* los datos de una abajo de la otra (el orden da igual). Para esto vamos a usar la función `concat` que toma una lista de DataFrames y regresa un DataFrame concatenado a lo largo del eje que queramos.

```
casos_defunciones = pd.concat([confirmados_diarios, defunciones_diarios], axis=0)
casos_defunciones
```

|     | Fecha      | variable    | value        | Tipo              |
|-----|------------|-------------|--------------|-------------------|
| 0   | 2022-01-01 | Conteo      | 6748.000000  | Casos Confirmados |
| 1   | 2022-01-02 | Conteo      | 6585.000000  | Casos Confirmados |
| 2   | 2022-01-03 | Conteo      | 10398.000000 | Casos Confirmados |
| 3   | 2022-01-04 | Conteo      | 9729.000000  | Casos Confirmados |
| 4   | 2022-01-05 | Conteo      | 10924.000000 | Casos Confirmados |
| ... | ...        | ...         | ...          | ...               |
| 721 | 2022-12-27 | Media Móvil | 4.714286     | Defunciones       |
| 722 | 2022-12-28 | Media Móvil | 5.285714     | Defunciones       |
| 723 | 2022-12-29 | Media Móvil | 5.285714     | Defunciones       |
| 724 | 2022-12-30 | Media Móvil | 5.285714     | Defunciones       |
| 725 | 2022-12-31 | Media Móvil | 4.571429     | Defunciones       |

Ya con la nueva serie como la queremos, podemos hacer nuestras *Facetas* utilizando el parámetro `facet_col` que le dice a Plotly qué columna usar para distinguir las dos series. Es importante decirle que no queremos que compartan el eje *y* porque las escalas son muy diferentes

```
fig = px.line(casos_defunciones, x='Fecha', y='value', color='variable', facet_col='Tipo',
fig.update_yaxes(matches=None)
fig.show()
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json

#### 4.0.4 Hospitalizaciones

Otra gráfica muy interesante para comprender la evolución de la epidemia es la de hospitalizaciones. Para obtener esta gráfica primero tenemos que seleccionar los pacientes confirmados como positivos a COVID-19 y que además fueron hospitalizados.

Los casos confirmados ya los tenemos calculados en la variable `confirmados`, entonces falta ver cómo obtener los pacientes hospitalizados

```
confirmados.TIPO_PACIENTE.unique()
```

```
array(['HOSPITALIZADO', 'AMBULATORIO'], dtype=object)
```

Gracias a nuestra base aplanada es muy fácil distinguirlos, entonces sólo los tenemos que seleccionar, agregar por día y podemos hacer una gráfica como las anteriores (incluyendo la media móvil). Recordemos que `confirmados` está indexado por fecha de inicio de síntomas, entonces nuestra curva de hospitalización estará indexada por la misma fecha

```
hospitalizados = confirmados[confirmados.TIPO_PACIENTE == 'HOSPITALIZADO']
hospitalizados_diarios = (hospitalizados
                           .groupby(pd.Grouper(freq='D'))[['ID_REGISTRO']] # grupos por día
                           .size() # Calculamos el tamaño de cada grupo
                           .reset_index() # Convertimos el resultado (que es una serie) en
                           .rename({0:'Hospitalizaciones'}, axis=1) # Le damos nombre a la
                           )
hospitalizados_diarios['Media Móvil'] = hospitalizados_diarios.rolling(window=7).mean()
hospitalizados_diarios = hospitalizados_diarios.melt(id_vars=['FECHA_SINTOMAS'], value_var
fig = px.line(hospitalizados_diarios, x='FECHA_SINTOMAS', y='value', color='variable')
fig.show()
```

/tmp/ipykernel\_89594/1651814779.py:8: FutureWarning:

Dropping of nuisance columns in rolling operations is deprecated; in a future version this will

Unable to display output for mime type(s): application/vnd.plotly.v1+json

Y, una vez más, para comparar vamos a poner las tres gráficas (casos confirmados, defunciones y hospitalizacones) en un Facet

```
hospitalizados_diarios['Tipo'] = 'Hospitalizaciones'
hospitalizados_diarios.loc[hospitalizados_diarios['variable'] == 'Hospitalizaciones', 'value'] = hospitalizados_diarios['value'].rolling(7).mean()
hospitalizados_diarios = hospitalizados_diarios.rename({'FECHA_SINTOMAS': 'Fecha'}, axis=1)
hospitalizados_diarios
```

|     | Fecha      | variable    | value      | Tipo              |
|-----|------------|-------------|------------|-------------------|
| 0   | 2022-01-01 | Conteo      | 117.000000 | Hospitalizaciones |
| 1   | 2022-01-02 | Conteo      | 91.000000  | Hospitalizaciones |
| 2   | 2022-01-03 | Conteo      | 124.000000 | Hospitalizaciones |
| 3   | 2022-01-04 | Conteo      | 120.000000 | Hospitalizaciones |
| 4   | 2022-01-05 | Conteo      | 144.000000 | Hospitalizaciones |
| ... | ...        | ...         | ...        | ...               |
| 729 | 2022-12-29 | Media Móvil | 35.285714  | Hospitalizaciones |
| 730 | 2022-12-30 | Media Móvil | 33.857143  | Hospitalizaciones |
| 731 | 2022-12-31 | Media Móvil | 31.285714  | Hospitalizaciones |
| 732 | 2023-01-01 | Media Móvil | 27.857143  | Hospitalizaciones |
| 733 | 2023-01-02 | Media Móvil | 21.571429  | Hospitalizaciones |

Combinamos con la serie de casos y defunciones

```
casos_defunciones_hospitalizaciones = pd.concat([hospitalizados_diarios, casos_defunciones], axis=1)
casos_defunciones_hospitalizaciones
```

|     | Fecha      | variable | value      | Tipo              |
|-----|------------|----------|------------|-------------------|
| 0   | 2022-01-01 | Conteo   | 117.000000 | Hospitalizaciones |
| 1   | 2022-01-02 | Conteo   | 91.000000  | Hospitalizaciones |
| 2   | 2022-01-03 | Conteo   | 124.000000 | Hospitalizaciones |
| 3   | 2022-01-04 | Conteo   | 120.000000 | Hospitalizaciones |
| 4   | 2022-01-05 | Conteo   | 144.000000 | Hospitalizaciones |
| ... | ...        | ...      | ...        | ...               |

|     | Fecha      | variable    | value    | Tipo        |
|-----|------------|-------------|----------|-------------|
| 721 | 2022-12-27 | Media Móvil | 4.714286 | Defunciones |
| 722 | 2022-12-28 | Media Móvil | 5.285714 | Defunciones |
| 723 | 2022-12-29 | Media Móvil | 5.285714 | Defunciones |
| 724 | 2022-12-30 | Media Móvil | 5.285714 | Defunciones |
| 725 | 2022-12-31 | Media Móvil | 4.571429 | Defunciones |

Graficamos las tres series

```
fig = px.line(casos_defunciones_hospitalizaciones, x='Fecha', y='value', color='variable',
fig.update_yaxes(matches=None)
fig.show()
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json

## **Part II**

# **Geoinformática en R**



Esta parte del libro cubre el manejo de datos espaciales utilizando R  
**EN CONSTRUCCIÓN**

# **5 Introducción a R**

EN CONSTRUCCIÓN

## 6 Summary

In summary, this book has no content whatsoever.

## References