

# **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
 <b>I Geoinformática en Python</b>	 <b>6</b>
<b>1 Transformación de datos</b>	<b>8</b>
1.1 Conjunto de Datos . . . . .	8
1.2 Limpieza de los datos . . . . .	10
1.3 Descripciones de los datos . . . . .	14
1.4 Creación de variables . . . . .	15
1.4.1 Modificar valores . . . . .	16
1.4.2 Eliminar columnas . . . . .	17
1.4.3 Buscando datos . . . . .	18
1.5 Ordenar valores . . . . .	19
1.6 Exploración Visual . . . . .	20
1.7 Organizando los datos . . . . .	25
1.8 Agrupamiento, Transformación y Agregación . . . . .	29
1.8.1 Para Practicar... . . . .	30
 <b>2 Ejemplo: datos COVID-19</b>	 <b>32</b>
2.1 Descargar datos . . . . .	33
2.2 Exploración del contenido . . . . .	34
2.3 Aplanado de datos . . . . .	36
2.4 Manejo de fechas . . . . .	45
2.4.1 Tarea . . . . .	47
 <b>3 Automatización</b>	 <b>48</b>
3.1 Función de preproceso . . . . .	48
3.2 Bajar y guardar datos . . . . .	48
3.3 Preprocesar . . . . .	49
3.4 Bajar y preprocesar usando nuestras funciones . . . . .	52
3.5 Guardando el resultado . . . . .	53

<b>II</b>	<b>Geoinformática en R</b>	<b>54</b>
<b>4</b>	<b>Introducción a R</b>	<b>56</b>
<b>5</b>	<b>Summary</b>	<b>57</b>
	<b>References</b>	<b>58</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 en Python**

Esta parte del libro cubre el manejo de datos espaciales utilizando Python.

Para los ejercicios de esta parte del libro vas a necesitar varios conjuntos de datos que puedes descargar de [aqui](#).

# 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

---

## 1.1 Conjunto de Datos

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 principamete 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:

#### Note

El archivo con los datos lo encuentras en la caropeta 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
1	09	Ciudad de México	002	Azcapotzalco		0000
2	09	Ciudad de México	002	Azcapotzalco		0001
3	09	Ciudad de México	002	Azcapotzalco		0001
4	09	Ciudad de México	002	Azcapotzalco		0001

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'`

## 1.2 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 tecnologí...	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 sola columna (o fila).

Las Series se pueden utilizar (igual que las listas) para seleccionar columnas de un `DataFrame`, entónces, 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

```

## 1.3 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

## 1.4 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_1585956/2610780181.py:1: PerformanceWarning: DataFrame is highly fragmented.
  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
```

### 1.4.1 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_1585956/463547730.py:2: PerformanceWarning: DataFrame is highly fragmented.
  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
```

### 1.4.2 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!

### 1.4.3 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

## 1.5 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
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

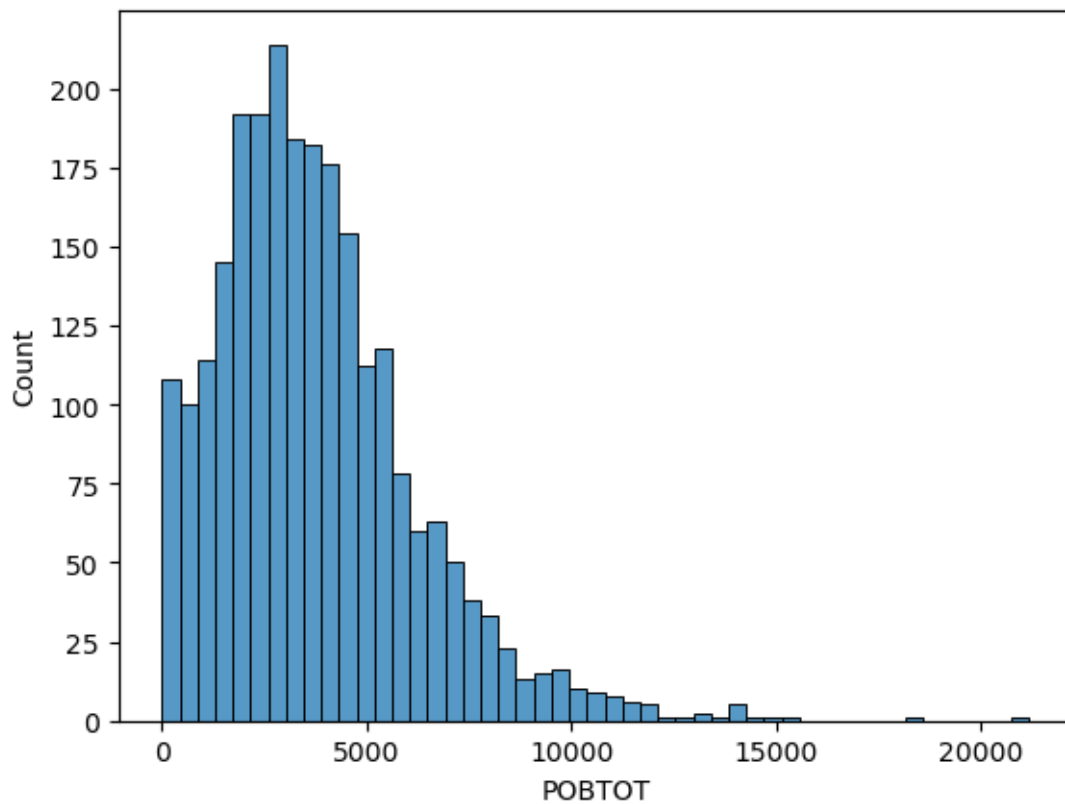
El método `sort_values` nos permite ordenar los datos de acuerdo al valor (o criterio) que queramos. El argumento `ascending = False` indica que los queremos ordenar de forma descendente.

## 1.6 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.

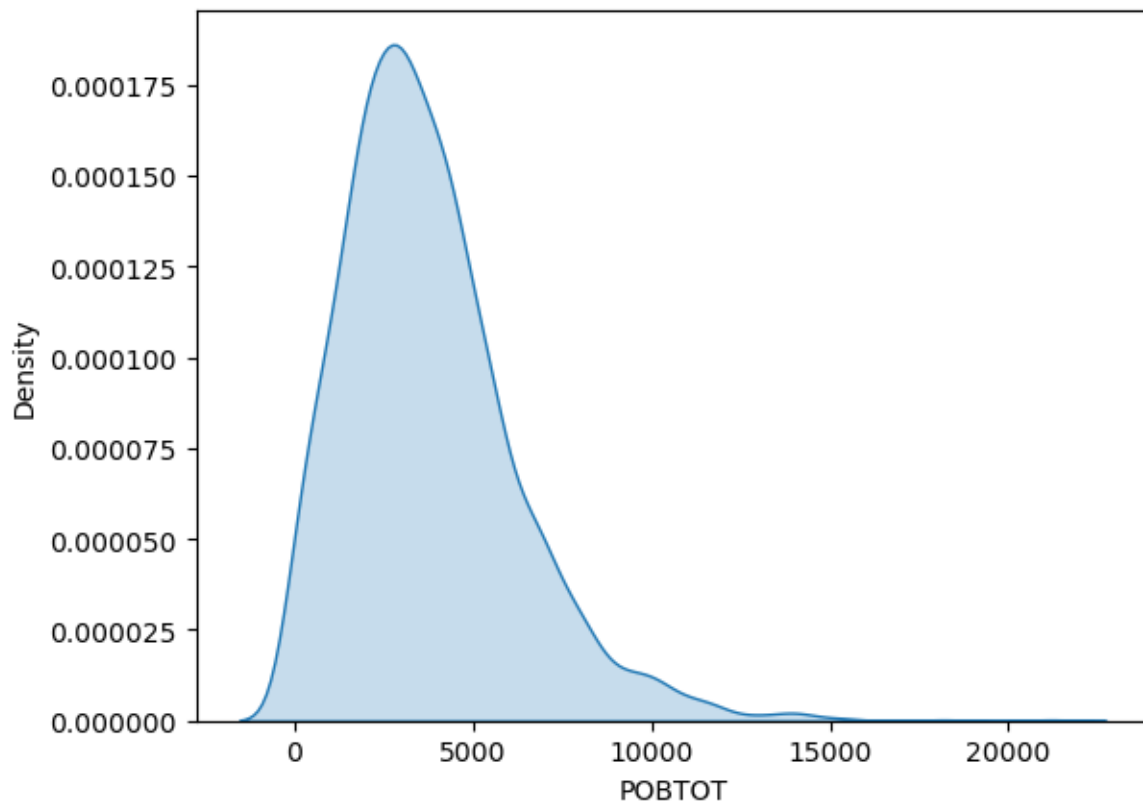
### **i** Note

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.

#### 1.6.0.1 Densidad de Kernel

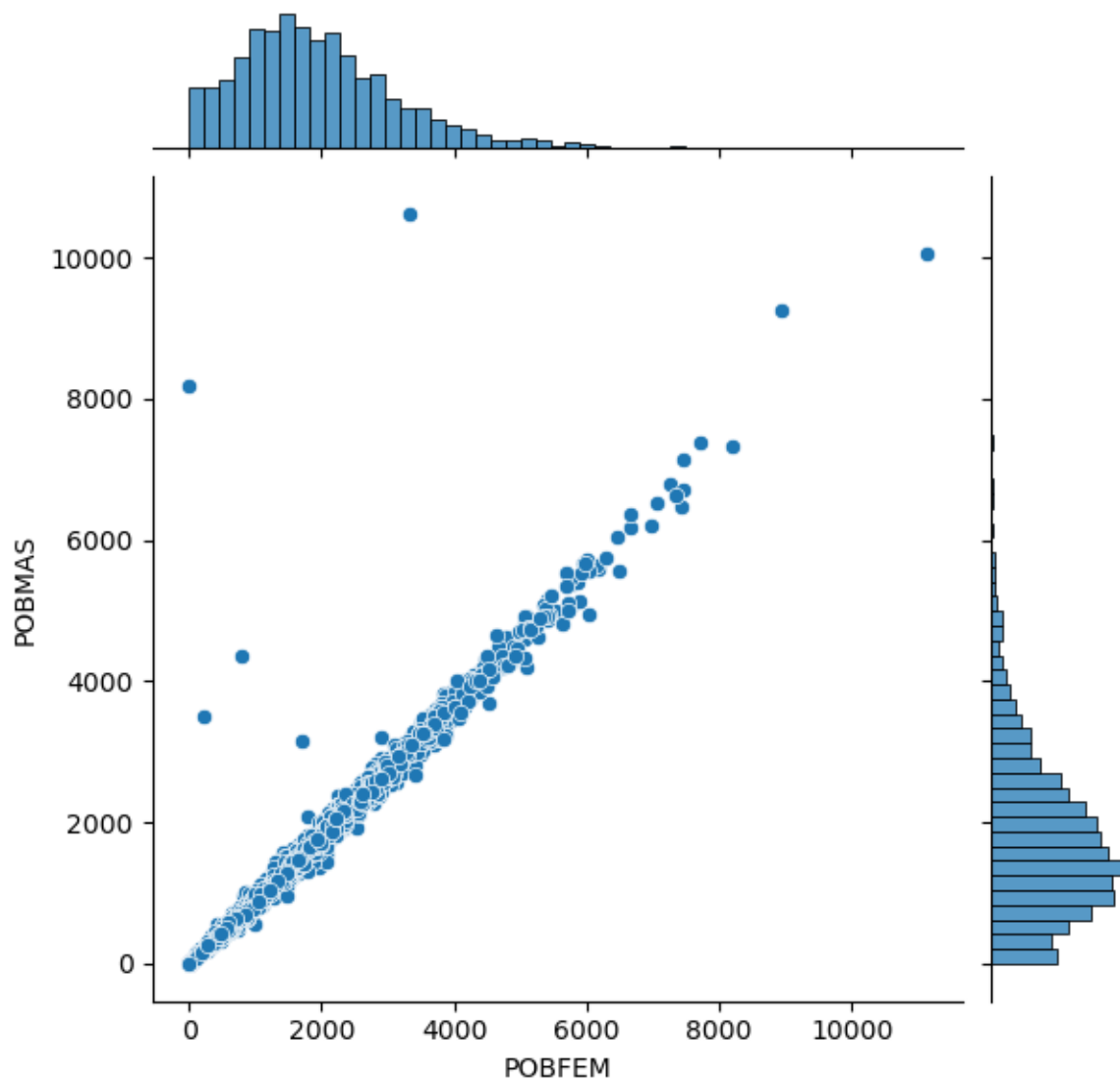
Otra forma de representar la distribución de una variable es ajustando una *densidad de kernel*, que estima una distribución (empírica) de probabilidad a partir de nuestras observaciones.

```
_ = 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')
```

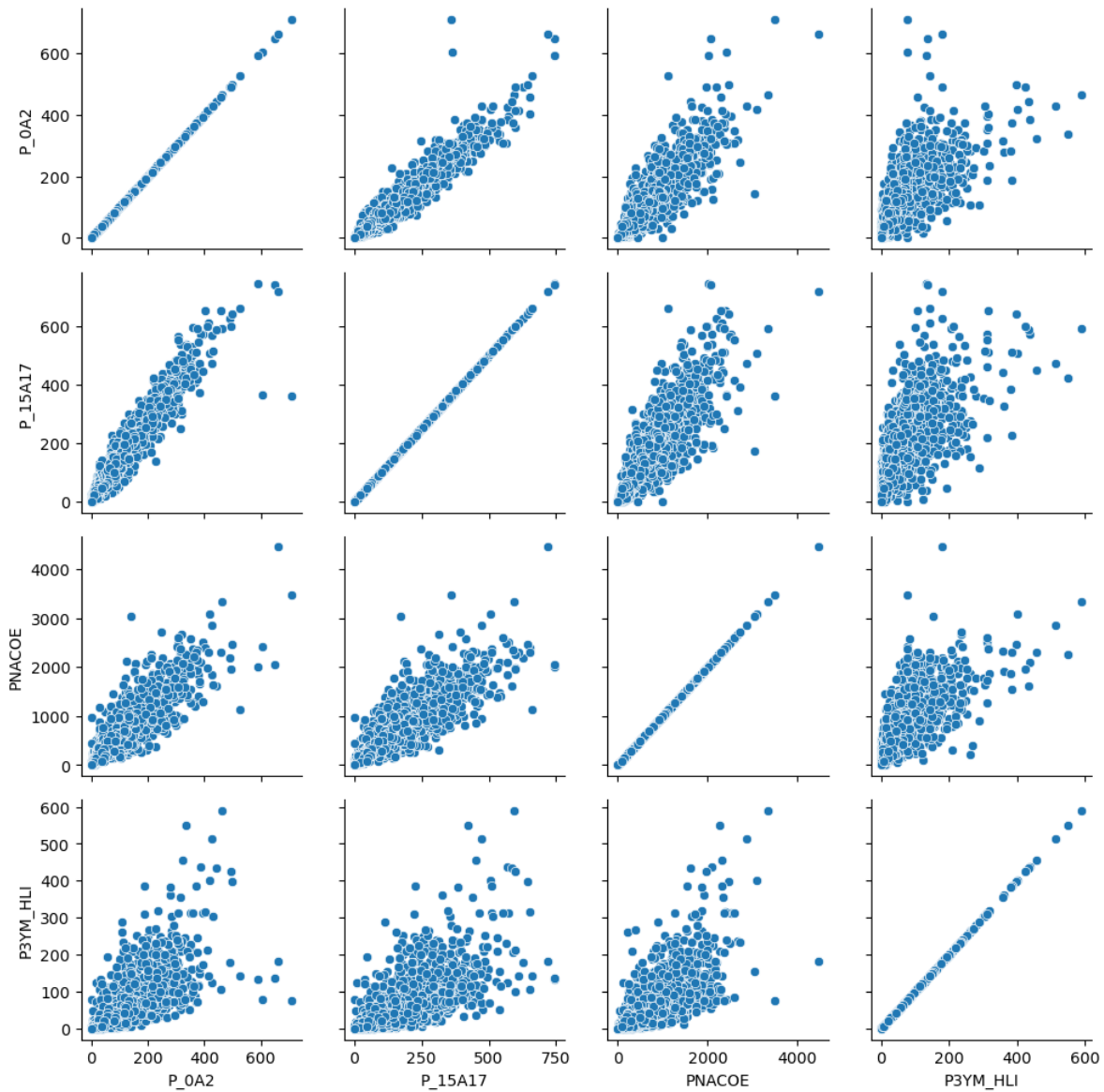


La relación, como es de esperarse, es casi perfectamente lineal, pero ver las distribuciones conjuntas nos permite identificar algunas AGEBS con poblaciones masculinas desproporcionadamente grandes ¿Qué serán?.

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

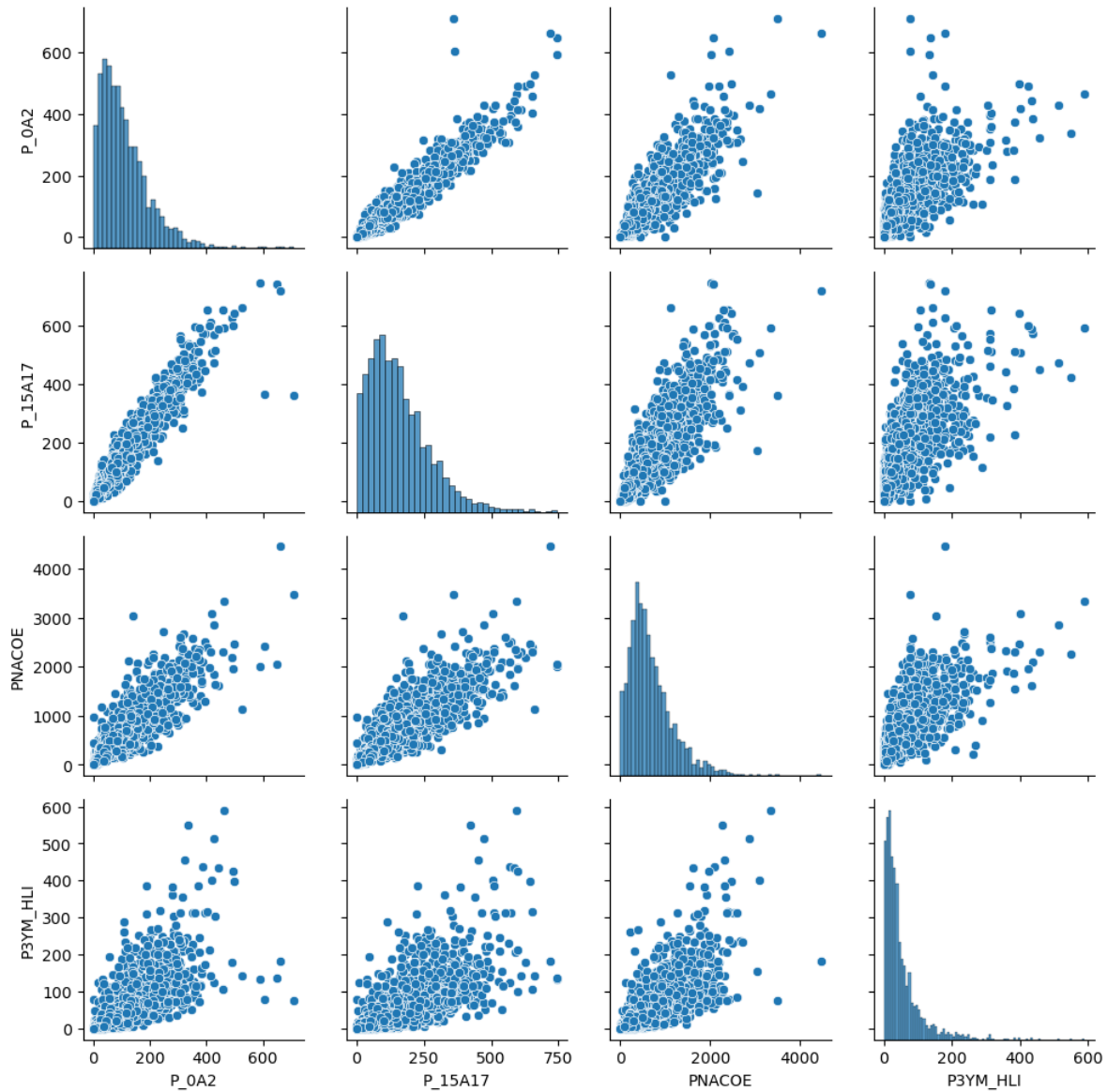
diagrama de dispersión.

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 0x7f62c77d80d0>
```





## 1.7 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: AGEB_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	4076.0
116	0900200010044	110.0	49.0	61.0	4658.0	2502.0	2156.0	4518.0
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`!

## 1.8 Agrupamiento, Transformación y Agregación

Una de las ventajas del proceso de limpieza anterior es que las tablas resultantes permiten realizar transformaciones avanzadas de forma directa; de las más comunes son las Operaciones de Agrupamiento (*Group-By*), las cuales permiten conjuntar observaciones utilizando sus etiquetas, índices o categorías, así como aplicar operaciones de carácter grupal.

Por ejemplo, si se quisiera sumar el total de la población perteneciente a cada subgrupo de edad, primero se necesita emplear uno de los operadores más populares y utilizados de **pandas**, y razón por la cual ha sido tan popularizado, llamado `.groupby()`:

```
pop_grupos = subgrupos_ordenDF.groupby('Subgrupo')
pop_grupos
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fc68c622668>
```

Aunque `pop_grupos` no genera un resultado inmediatamente visible, si almacena la manera en que se realiza la agrupación y, por ende, nos permite realizar una multitud de operaciones, como la suma buscada a través del método `.sum()`:

```
pop_grupos.sum()
```

	Frecuencia
Subgrupo	
pob_0a2	354934
pob_12a14	392333
pob_15a17	423462
pob_18a24	1042926
pob_25a29	720664

	Frecuencia
Subgrupo	
pob_30a49	2591465
pob_3a5	388398
pob_50a59	895259
pob_60ymas	1001547
pob_6a11	788875

Asímismo, puede generarse un resumen de cada subgroup utilizando `.describe()`

```
pop_grupos.describe()
```

	Frecuencia							
	count	mean	std	min	25%	50%	75%	max
Subgrupo								
pob_0a2	2432.0	145.943257	112.056808	0.0	61.00	119.0	209.25	743.0
pob_12a14	2432.0	161.321135	115.647911	0.0	74.00	138.5	226.25	772.0
pob_15a17	2432.0	174.120888	121.923333	0.0	82.00	151.5	241.00	760.0
pob_18a24	2432.0	428.834704	288.072840	0.0	218.00	382.5	580.25	2905.0
pob_25a29	2432.0	296.325658	201.741121	0.0	155.00	263.0	402.00	3099.0
pob_30a49	2432.0	1065.569490	693.264089	0.0	578.00	963.5	1417.25	8195.0
pob_3a5	2432.0	159.703125	121.498713	0.0	67.00	133.0	226.25	746.0
pob_50a59	2432.0	368.116365	217.976714	0.0	221.75	342.0	492.50	1401.0
pob_60ymas	2432.0	411.820312	250.572492	0.0	233.75	392.0	562.00	1525.0
pob_6a11	2432.0	324.372944	240.267120	0.0	144.00	273.0	461.00	1586.0

La versatilidad de `.groupby()` permite que sobre los grupos no sólo se puedan aplicar funciones genéricas, como `.sum()` o `.describe()`, sino también funciones generadas por el propio usuario, lo cual permite realizar casi cualquier tipo de transformación y agregación.

### 1.8.1 Para Practicar...

Resulta útil aplicar lo aprendido sobre otro tipo de Conjuntos de Datos. Por ejemplo, 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)

Desde Python es posible visualizar la apariencia de esta información utilizando los siguientes comandos:

```
# Importar la librería a utilizar
from IPython.display import IFrame

# Guardar en una variable la URL de la información
url = 'https://docs.google.com/spreadsheets/d/1EAx8_ksSCmoWW_SlhFyq2QrRn0FNNhcg1TtDFJzZRgc/edit'

# Mostrar como ventana lo que arrojaría la URL
IFrame(url, 700, 400)
```

<IPython.lib.display.IFrame at 0x7fc68e4cafd0>

Realiza lo siguiente con estos datos: \* Descarga la tabla como un archivo de tipo .csv (Archivo -> Descargar como -> .csv, hoja actual) \* Importa los datos a Python \* Explora generando algunas gráficas \* Examina qué tanto se ajusta a los principios del *Tidy Data* y conviértelo de forma que lo haga \* Obten una cuenta total de las casualidades por mes y genera una gráfica con ésta

## 2 Ejemplo: datos COVID-19

Las bases de datos provenientes de situaciones reales son problemáticas, no están limpias, pueden tener datos faltantes, codificar los datos de formas extrañas, combinar tipos de datos, etcétera.

Es bien sabido que dentro de la Ciencia de Datos, se tenga la componente espacial o no, el trabajo no sólo se enfoca a generar modelos sofisticados o al análisis mismo, sino también a tareas mucho más básicas y menos exóticas como obtener los datos, procesarlos y modificarlos de forma que puedan ser examinados y explorados para entender sus propiedades básicas.

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 una base de datos del mundo real para aprender, utilizando Python, técnicas para manipularlas, transformarlas y limpiarlas de forma que podamos hacer análisis.

La base de datos que vamos a utilizar son los datos abiertos 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.

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
```



## 2.1 Descargar datos

El primer paso es, evidentemente, descargar los datos. Es claro que podríamos ir a la [página](#) de la Dirección General de Epidemiología (DGE) y descargarlos, sin embargo, lo vamos a hacer por el camino difícil de bajarlos usando Python.

Para eso, vamos a definir una función que tome como argumento la fecha para la que queremos descargar los datos y el *path* en donde los vamos a guardar. La función es relativamente simple, sólo hace un *request* al archivo histórico de la DGE y guarda los datos en la ruta configurada.

Para trabajar vamos siempre a descargar tres archivos:

- Los datos COVID-19
- El catálogo de campos
- El archivo de descripción

El primer archivo contiene la serie de tiempo de seguimiento de casos hasta la fecha configurada, los dos archivos restantes sirven para entender la información contenida en los datos.

Entonces, primero definimos la función

```
def bajar_datos_salud(directorio_datos='data/', fecha='10-01-2022'):
    """
        Descarga el archivo de datos y los diccionarios para la fecha solicitada.
    """
    fecha = datetime.strptime(fecha, "%d-%m-%Y")
    url_salud_historicos = 'http://datosabiertos.salud.gob.mx/gobmx/salud/datos_abiertos/h
    archivo_nombre = f'{fecha.strftime("%y%m%d")}COVID19MEXICO.csv.zip'
    archivo_ruta = os.path.join(directorio_datos, archivo_nombre)
    url_diccionario = 'http://datosabiertos.salud.gob.mx/gobmx/salud/datos_abiertos/diccio
    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 {fecha.strftime("%d.%m.%Y")}')
        url_dia = "{}/{}/datos_abiertos_covid19-{}.zip".format(fecha.strftime('%Y'),
                                                                fecha.strftime('%m'),
                                                                fecha.strftime('%d.%m.%Y'))

        url = url_salud_historicos + url_dia
        r = requests.get(url, 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)
```

Con la función que acabamos de definir, podemos bajar los datos hasta la última fecha disponible

```
ayer = datetime.now() - timedelta(1)
bajar_datos_salud(fecha=ayer.strftime('%d-%m-%Y'))
```

## 2.2 Exploración del contenido

Antes de empezar a manipular los datos, lo primero que tenemos que hacer es explorarlos brevemente y entender cómo están organizados. Leamos los datos en un DataFrame.

Fíjense en la ruta en donde la función de arriba descarga los datos, con la barra exploradora del lado izquierdo pueden navegar hasta esa ruta y copiar el *path*.

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('data/220110COVID19MEXICO.csv.zip', dtype=object, encoding='latin-1')
df.head()
```

	FECHA_ACTUALIZACION	ID_REGISTRO	ORIGEN	SECTOR	ENTIDAD_UM	SEXO	ENT
0	2022-01-10	zz7202	1	12	16	2	16
1	2022-01-10	z58ed3	2	12	18	1	18
2	2022-01-10	z3be8c	2	12	07	1	07
3	2022-01-10	z526b3	2	12	09	1	09
4	2022-01-10	z3d1e2	2	12	09	1	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. Pueden descargarlo en el explorador de archivos del lado izquierdo, pero aquí vamos a abrirlo y explorarlo un poco

con Pandas. 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 = 'data/201128 Catalogos.xlsx'
nombres_catalogos = ['Catálogo de ENTIDADES', # Acá 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.

## 2.3 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
3	2022-01-10	z526b3	2	12	09	1
4	2022-01-10	z3d1e2	2	12	09	1
11	2022-01-10	z29dac	2	12	09	1
15	2022-01-10	z45dcb	2	12	09	2
22	2022-01-10	z23c2e	2	12	09	1
...	...	...	...	...	...	...
12731369	2022-01-10	m00073e	2	12	15	2
12731784	2022-01-10	m030623	2	12	15	2
12731821	2022-01-10	m049633	2	12	15	1
12732221	2022-01-10	m160d02	2	12	15	2
12732863	2022-01-10	m0da9ec	2	12	15	1

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
3	2022-01-10	z526b3	2	12	09	1	09
4	2022-01-10	z3d1e2	2	12	09	1	09
11	2022-01-10	z29dac	2	12	09	1	09
15	2022-01-10	z45dcb	2	12	09	2	09
22	2022-01-10	z23c2e	2	12	09	1	09

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
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

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 iterado 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_FINAL']))
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()
```

```

3     CASO DE SARS-COV-2  CONFIRMADO
4     CASO DE SARS-COV-2  CONFIRMADO
11    NO REALIZADO POR LABORATORIO
15        NEGATIVO A SARS-COV-2
22        CASO SOSPECHOSO
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('data/201128 Descriptores_.xlsx',
                             index_col='Nº',
                             engine='openpyxl')

descriptores

```

Nº	NOMBRE DE VARIABLE	DESCRIPCIÓN DE VARIABLE	FORMATO O F
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
8	ENTIDAD_RES	Identifica la entidad de residencia del paciente.	CATALÓGO: EN
9	MUNICIPIO_RES	Identifica el municipio de residencia del paci...	CATALÓGO: 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



Nº	NOMBRE DE VARIABLE	DESCRIPCIÓN DE VARIABLE	FORMATO O FUENTE
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(['1', '2', '98'], 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).

Perfecto, 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
31	OTRO_CASO	Identifica si el paciente tuvo contacto con al...	CATÁLOGO: SI_NO
32	TOMA_MUESTRA_LAB	Identifica si al paciente se le tomó muestra d...	CATÁLOGO: SI_NO
34	TOMA_MUESTRA_ANTIGENO	Identifica si al paciente se le tomó muestra d...	CATÁLOGO: SI_NO
37	MIGRANTE	Identifica si el paciente es una persona migra...	CATÁLOGO: SI_NO
40	UCI	Identifica si el paciente requirió ingresar a ...	CATÁLOGO: SI_NO

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'].values))
# sustituimos en los datos originales
df[campos_si_no] = df[datos_si_no.NOMBRE_DE_VARIABLE].replace(
    to_replace=cat_si_no['CLAVE'].values,
    value=cat_si_no['DESCRIPCIÓN'].values)

df[campos_si_no]
```

	INTUBADO	NEUMONIA	EMBARAZO	HABLA_LENGUA_INDIG	INDIGENA	DIABE
3	0	0	0	0	0	0
4	0	0	0	0	0	0
11	0	0	0	0	0	0
15	0	0	0	0	0	0
22	0	0	0	0	0	0
...	...	...	...	...	...	...
12731369	0	0	0	0	0	0
12731784	0	0	0	0	0	0
12731821	0	0	0	0	0	0
12732221	0	0	0	0	0	0
12732863	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

## 2.4 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())
```

```

3    2020-12-21
4    2020-04-22
11   2020-09-12
15   2020-08-18
22   2020-02-26
Name: FECHA_INGRESO, dtype: datetime64[ns]

```

Veán la diferencia con el tipo de datos original:

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

```

3    2020-12-21
4    2020-04-22
11   2020-09-12
15   2020-08-18
22   2020-02-26
Name: FECHA_INGRESO, dtype: object

```

Esto va a funcionar perfecto cuando el campo tiene sólo datos válidos (todos se ajustan a algún formato de fecha), pero ¿qué pasa en campos en donde no todos los datos son fechas válidas?

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

```
ParserError: month must be in 1..12: 9999-99-99
```

ERROR!!!! Efectivamente, pandas no puede transformar algunos datos en fechas verdaderas (porque no todos los registros tienen una fecha de defunción válida). Entonces hay que decirle a Pandas qué hacer con estos registros, lo que queremos en este caso es que los registros inválidos los regrese como nulos, para eso usamos la opción **coerce**

```
pd.to_datetime(df['FECHA_DEF'].head(), 'coerce')
```

```

3    NaT
4    NaT
11   NaT
15   NaT
22   NaT
Name: FECHA_DEF, dtype: datetime64[ns]

```

Listo, los registros que no se pueden convertir en fechas ahora regresan **NaT** (Not a Time) en lugar de error.

Con esto en realidad ya es muy simple convertir toda una columna en tipo fecha:

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

```
3    2020-12-21
4    2020-04-22
11   2020-09-12
15   2020-08-18
22   2020-02-26
Name: FECHA_INGRESO, dtype: datetime64[ns]
```

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.

### 2.4.1 Tarea

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

## 3 Automatización

### 3.1 Función de preproceso

En el taller anterior vimos toda una serie de pasos para preprocesar los datos de COVUD-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
```

### 3.2 Bajar y guardar datos

```
def bajar_datos_salud(directorio_datos='data/', fecha='10-01-2022'):
    '''
        Descarga el archivo de datos y los diccionarios para la fecha solicitada.
    '''
    fecha = datetime.strptime(fecha, "%d-%m-%Y")
    url_salud_historicos = 'http://datosabiertos.salud.gob.mx/gobmx/salud/datos_abiertos/h
    archivo_nombre = f'{fecha.strftime("%y%m%d")}-COVID19MEXICO.csv.zip'
    archivo_ruta = os.path.join(directorio_datos, archivo_nombre)
    url_diccionario = 'http://datosabiertos.salud.gob.mx/gobmx/salud/datos_abiertos/diccio
    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 {fecha.strftime("%d.%m.%Y")}')
        url_dia = "{}/{}/datos_abiertos_covid19_{}.zip".format(fecha.strftime('%Y'),
                                                                fecha.strftime('%m'),
                                                                fecha.strftime('%d.%m.%Y'))

        url = url_salud_historicos + url_dia
        r = requests.get(url, 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)

```

### 3.3 Preprocesar

```

def carga_datos_covid19_MX(data_dir = 'data/', fecha='210505', resolver_claves='si_no_binarias')
    """
    Lee en un DataFrame el CSV con el reporte de casos de la Secretaría de Salud de México
    también lee el diccionario de datos que acompaña a estas publicaciones para preparar
    de generar columnas binarias para datos con valores 'SI', 'No'.

    **Nota**: En esta versión la ruta esta y nombre de los archivos es fija. Asumimos
    donde se encuentran todos los archivos.

    **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

    """
    fecha_formato = '201128'
    nuevo_formato = True
    fecha_carga = pd.to_datetime(fecha, yearfirst=True)
    if fecha_carga < datetime.strptime('20-11-28', "%y-%m-%d"):
        raise ValueError('La fecha debe ser posterior a 20-11-28.')

    catalogos=f'{data_dir}{fecha_formato} Catalogos.xlsx'

```

```

descriptores=f'{data_dir}{fecha_formato} Descriptores.xlsx'
data_file = os.path.join(data_dir, f'{fecha}COVID19MEXICO.csv.zip')
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']

if nuevo_formato:
    nombres_catálogos.append('Catálogo CLASIFICACION_FINAL')
    nombres_catálogos[2] = 'Catálogo RESULTADO_LAB'

dict_catálogos = pd.read_excel(catálogos,
                               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"]

if nuevo_formato:
    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
if nuevo_formato:
    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['CLASIFICACIÓN']))
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'])
    df['FECHA_SINTOMAS'] = pd.to_datetime(df['FECHA_SINTOMAS'])
    df['FECHA_DEF'] = pd.to_datetime(df['FECHA_DEF'], 'coerce')
    df['DEFUNCION'] = (df['FECHA_DEF'].notna()).astype(int)
    df['EDAD'] = df['EDAD'].astype(int)

    df.set_index('FECHA_INGRESO', drop=False, inplace=True)
    df['AÑO_INGRESO'] = df.index.year
    df['MES_INGRESO'] = df.index.month
    df['DIA_SEMANA_INGRESO'] = df.index.weekday
    df['SEMANA_AÑO_INGRESO'] = df.index.week
    df['DIA_MES_INGRESO'] = df.index.day
    df['DIA_AÑO_INGRESO'] = df.index.dayofyear

    return df

```

### 3.4 Bajar y preprocesar usando nuestras funciones

```

ayer = datetime.now() - timedelta(2)
bajar_datos_salud(fecha=ayer.strftime('%d-%m-%Y'))
df = carga_datos_covid19_MX(fecha=ayer.strftime('%y%m%d'), entidad='09')
df

```

Bajando datos 18.01.2022

data/220118COVID19MEXICO.csv.zip

```
/tmp/ipykernel_3563/19327198.py:132: FutureWarning: weekofyear and week have been deprecated
  df['SEMANA_AÑO_INGRESO'] = df.index.week
```

FECHA_INGRESO	FECHA_ACTUALIZACION	ID_REGISTRO	ORIGEN	SECTOR	ENTIDAD_
2020-07-06	2022-01-18	z12d63	2	12	CIUDAD DE
2020-09-23	2022-01-18	z13788	1	12	CIUDAD DE
2020-06-15	2022-01-18	z2b144	2	12	CIUDAD DE
2020-12-21	2022-01-18	z526b3	2	12	CIUDAD DE
2020-04-22	2022-01-18	z3d1e2	2	12	CIUDAD DE
...	...	...	...	...	...
2021-10-11	2022-01-18	m00073e	2	12	MÉXICO
2021-10-13	2022-01-18	m030623	2	12	MÉXICO
2021-10-13	2022-01-18	m049633	2	12	MÉXICO
2021-10-13	2022-01-18	m160d02	2	12	MÉXICO
2021-10-14	2022-01-18	m0da9ec	2	12	MÉXICO

### 3.5 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.

# **Part II**

## **Geoinformática en R**

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

## 4 Introducción a R

EN CONSTRUCCIÓN



## 5 Summary

In summary, this book has no content whatsoever.

## References