Seminario de Lenguajes - Python

Cursada 2024

Más funciones de Pandas y mapas

💥 ¿Qué vimos en la clase anterior?

Modificación de datos

- renombar columna/s: **rename**
- renombrar todas las columnas: **columns** (igual cantidad)
- modificar datos: **replace**
- modificación en el momento: **inplace** (sólo en algunos casos)
- copia segura del dataset: **copy**
- crear una nueva variable
- cambio de tipo de dato:
  - datetime: **pd.to_datetime**
  - int, float..: **astype**
- eliminación de datos nulos:
  - de todas las filas: **df.dropna**
  - de una columna: **df.columna.dropna**
- generación de datos en función de datos del dataframe: **apply**

Crear nuestro Dataframe

- de un diccionario
- definición de los índices
- cambio de columna como índice

Agrupamientos

- **groupby**: definición de una operación de agregación
- una columna
- más de una columna
- operaciones:
  - **nunique**
  - **sum**
  - **count**
  - **value_counts**
  - otras

Gráficos

- matplotlib
  - simples: **df.plot**
  - más personalizados: **plt.funciones de cada gráfico**

- plotly
  - Gráficos simples: **plotly.express**
  - Gráficos más complejos: **plotly.graph_objects (go.Figure)**

Streamlit

- gráficos simples: **st.tipo_grafico**
- gráficos a través de librerías específicas: **st.librería(figura)**
- widget interactivos

Trabajemos con el dataset de aeropuertos otra vez

In [ ]:
```
import pandas as pd
from pathlib import Path
```

In [ ]:
```
file_route = Path('files')
file_data = 'ar-airports.csv'
df_airports = pd.read_csv(file_route/file_data)
```

🫀 Habíamos hecho algunas operaciones con los datos nulos

- eliminar datos nulos de la columna **elevation_ft** y cambiar a int.
- si no queremos perder esas filas, al crear una columna nueva en función de esa columna, ¿qué pasó?

In [ ]:
```
df_airports = df_airports.copy()

def define_height(value):
    if value < 131:
        return 'bajo'
    elif value < 903:
        return 'medio'
    else:
        return 'alto'
```

In [ ]:
```
df_airports['height'] = df_airports['elevation_ft'].apply(define_height)
df_airports[['name','elevation_ft','height']].head(4)
```

Pero ¿qué paso en los casos que era nulo?:

In [ ]:
```
df_airports.iloc[926]
```

Como el valor entra por el último else, le pone como **alto** en la columna creada, en las filas que la columna **elevation_ft** es nula.

Veamos algunas cuestiones para realizar en estos casos, que nos puede servir para hacer modificaciones.

📌 1.Comprobar si todos los valores float en la columna 'elevation_ft' no tienen decimales

In [ ]:
```
# Primero, ignorar los NaN para evitar errores de comparación
non_decimal_values = df_airports['elevation_ft'].dropna().apply(lambda x: x.is_intege
all_integers = non_decimal_values.all()
```

```
all_integers
```

- **df_airports['elevation_ft'].dropna()** devuelve una variable de tipo Series con los valores nulos eliminados
- **lambda x: x.is_integer()**: verifica que un valor sea igual a su parte entera, es decir comprobar que no tiene decimales
- **non_decimal_values.all()**: devuelve **True** si todos los valores son verdaderos

📌 2.Cambiar el tipo de dato que permita manejar valores nulos

In [ ]:
```
if all_integers:
    df_airports['elevation_ft'] = df_airports['elevation_ft'].astype(pd.Int64Dtype())
    print("Todos los valores fueron convertidos exitosamente a enteros.")
else:
    print("Algunos valores tienen decimales y no pueden ser convertidos a enteros sin
```

astype(pd.Int64Dtype()) convierte la columna a enteros (Int64) que permiten valores nulos o NaN representado como **NA**.

In [ ]:
```
df_airports.info()
```

In [ ]:
```
df_airports.iloc[926]
```

📌 3. Aplicar la función evaluando los valores nulos

In [ ]:
```
def define_height(value):
    if pd.isna(value):
        return 'desconocido'
    if value < 131:
        return 'bajo'
    elif value < 903:
        return 'medio'
    else:
        return 'alto'
```

In [ ]:
```
df_airports['elevation_ft'] = df_airports['elevation_ft'].astype('float').astype(pd.I
df_airports['altura'] = df_airports.elevation_ft.apply(define_height)
```

In [ ]:
```
df_airports.iloc[926]
```

¿Qué cambia?

- Podríamos usar **is_na** sin convertir la columna a **Int64Dtype**, pero de esta forma nos aseguramos que el valor es **NaN** en esos casos.
- De esta forma no eliminamos filas y podemos tener identificadas las alturas en los casos que esté presente.

¿Cuál es la cantidad y cuáles son filas con valores nulos?

¿Cómo puedo saber las filas que tienen en la columna **elevation_ft** valores nulos?

Para saber cuántos:

In [ ]:
```
df_airports.elevation_ft.isnull().sum()
```

Para saber cuáles:

In [ ]:
```
df_airports[df_airports['elevation_ft'].apply(lambda x: pd.isna(x))]
```

🦍 Agrupamientos con operaciones

- Vamos a trabajar con el dataset **felinos**
- Primero veamos algunas modificaciones para acomodar los datos

In [ ]:
```
felinos = pd.read_csv(file_route/'felinos_filtrado.csv')
felinos.stateProvince = felinos.stateProvince.replace('Neuquen','Neuquén')
felinos = felinos.dropna(subset=['year']).copy()
felinos.info()
```

Las columnas **day**, **month** y **year** se les asignó el tipo de dato **float**.

¿Con qué función le podemos cambiar el tipo de dato?

In [ ]:
```
felinos['year'] = felinos['year'].astype(int)
```

Para cambiar varias columnas al mismo tiempo podemos definir la lista de columnas y luego hacer el cambio:

In [ ]:
```
columns_to_convert = ['day','month']
felinos[columns_to_convert] = felinos[columns_to_convert].astype(int)
```

Definimos la lista de columnas que queremos cambiar a un mismo tipo de datos y luego aplicamos a todas la función que convierte **astype**.

Ahora veamos algunas operaciones y sus diferencias

¿Qué diferencia hay entre estas operaciones?

In [ ]:
```
felinos.groupby('genus')['stateProvince'].nunique()
```

In [ ]:
```
felinos.genus.value_counts()
```

In [ ]:
```
felinos.groupby('genus')['stateProvince'].count()
```

In [ ]:
```
felinos['stateProvince']
```

In [ ]:
```
felinos.groupby('genus')['stateProvince'].sum()
```

In [ ]:
```
felinos.groupby('genus')['year'].max()
```

In [ ]:
```
felinos.groupby('genus')['stateProvince'].value_counts()
```

In [ ]:
```
felinos.groupby(['genus','stateProvince'])['stateProvince'].count()
```

Ambos están contando la frecuencia de las combinaciones únicas de **genus** y **stateProvince**.

- **nunique**: cantidad de valores diferentes de una columna
- **value_counts**: la cantidad de valores de una columna
- **count**: cuenta la cantidad de valores no nulos

    - Para una Series: series.count()
    - Para un DataFrame: dataframe.count()
- **value_counts**: se utiliza para contar la frecuencia de los valores únicos en una Series.
    - Devuelve una Series que contiene los conteos de frecuencia de los valores únicos, ordenados en orden descendente.
- **sum**: devuelve los valores sumados, se uiliza principalmente en columnas con valores numéricos
- **max**: devuelve el valor máximo de una columna

📌 Gráfico usando agrupación

Hagamos un gráfico simple con las cantidades de provincias donde se han visto felinos

In [ ]:
```
felinos.groupby(['genus'])['stateProvince'].nunique().plot(kind='bar')
```

¿No quedaría mejor el gráfico con las barras ordendas por cantidad?

Más adelante veremos cómo graficar con los valores ordenamos por cantidad

🫀 Datos en mapas

📌 Folium

- librería de Python para la visualización de datos geoespaciales basada en **Leaflet.js**
- instalar:

```
pip install folium
```

- Generar mapas básicos **Map**
- Agregar puntos **Marker**
- Agregar polígonos **Polygon**
- Agregar áreas o multipolígonos usando GeoPandas y Folium

📌 Generar un mapa básico

In [ ]:
```
import folium
```

In [ ]:
```
folium.Map(
    location=(-33.457606, -65.346857),
    control_scale=True,
    zoom_start=5
)
```

- **location=(-33.457606, -65.346857)**: Indicamos el centro donde queremos generar el mapa.
- **control_scale=True**: muestra la barra de escala del mapa.
- **zoom_start=5**: establece un nivel de zoom que puede variar desde:
    - 0 (mundo entero) hasta
    - 18 (detalles de la calle), dependiendo de los datos disponibles.

¿Qué problemas tenemos con la identificación de los lugares, dado que estamos en Argentina?

Folium utiliza **OpenStreeMap**, dado que es abierto se puede adaptar la idenfitcación, el Miniterio de Defensa a través del Instituto Geográfico Nacional, pone a disposición una capa que adapta el mapa para utilizar:

In [ ]:
```
attr = (
    '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> '
    'contributors, &copy; <a href="https://cartodb.com/attributions">CartoDB</a>'
)

tiles = 'https://wms.ign.gob.ar/geoserver/gwc/service/tms/1.0.0/capabaseargenmap@EPSG%
m = folium.Map(
    location=(-33.457606, -65.346857),
    control_scale=True,
    zoom_start=5,
    name='es',
    tiles=tiles,
    attr=attr
)
m
```

Ahora que tenemos nuestro mapa con la identificación correcta, veamos cómo agregar datos

📌Agregar puntos o marker al mapa

Lo primero que necesitamos es identificar las columnas donde están están los datos de geolocalización, en el dataset de felinos

In [ ]:
```
felinos.columns
```

Los datos se encuentran en:

- decimalLatitude
- decimalLongitude

In [ ]:
```
felinos.loc[0,['decimalLatitude',       'decimalLongitude']]
```

Queremos mostrar en un mapa los avistajes de los diferentes felinos

- generar puntos con color que permitan diferenciar por tipo
- este dato se encuentra en la columna **genus**

- Definimos una función para generar el mapa con la identificación correcta: **generate_map**
- Definimos una función para que nos devuelva el color según el tipo: **get_color**
- Definimos una función para que agregue los puntos **Marker** al mapa: **add_marker**

In [ ]:
```
def generate_map():
    attr = (
    '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> '
    'contributors, &copy; <a href="https://cartodb.com/attributions">CartoDB</a>'
    )

    tiles = 'https://wms.ign.gob.ar/geoserver/gwc/service/tms/1.0.0/capabaseargenmap@E
    m = folium.Map(
        location=(-33.457606, -65.346857),
        control_scale=True,
        zoom_start=5,
        name='es',
        tiles=tiles,
```

```
        attr=attr
    )
    return m

def get_color(categoria):
    match categoria:
        case 'Puma':
            return 'blue'
        case 'Leopardus':
            return 'green'
        case _:
            return 'red'
def add_marker(row):
    color = get_color(row['genus'])
    folium.Marker(
        [row['decimalLatitude'], row['decimalLongitude']],
        popup=row['genus'],
        icon=folium.Icon(color=color)
    ).add_to(mapa)
mapa = generate_map()
felinos.apply(add_marker, axis=1)
mapa
```

Mostremos en el mapa los felinos vistos en un año específico

In [ ]:
```
felinos.year.unique()
```

Si queremos ver los datos ordenados, veamos cómo podemos utilizar las funciones de Pandas

🧰 Ordenamiento

📌 Ordenamiento por un criterio

In [ ]:
```
sorted_felinos = felinos.sort_values(by='year', ascending=False)
sorted_felinos.year
```

La función **sort_values** nos permite seleccionar varias opciones para ordenar, algunas de las más relevantes son:

- **by**: columna o lista de nombres de columnas por el cual ordenar
- **ascending**: por defecto de menor a mayor, si queremos cambiar el orden, **ascending=False**
- **inplace**: al igual que en muchas operaciones de pandas, la modificación puede ejecutarse solamente para ver el resultado, sin modificar el dataset original, con la opción **inplace=True**, se guarda el cambio en el momento en que se ejecuta.
- Con la instrucción **help** podemos consultar en el momento las opciones posibles:

In [ ]:
```
help(felinos.sort_values)
```

📌 Ordenamiento por dos criterios

Por ejemplo primero por **year** y luego por **month**:

In [ ]:
```
felinos.sort_values(by=['year','month'])[['genus','year','month']].tail(10)
```

Ordenemos por dos criterios pero de forma descendente el segundo:

In [ ]:
```
felinos.sort_values(by=['year','month','day'],ascending =[True , False , True])[['genu
```

Ordenar para mostrar en un gráfico

Habíamos hecho un gráfico simple de los diferentes felinos por provincia pero queremos que las barras se muestren ordenadas, entonces:

- ordenamos los datos agrupados
- luego hacemos el gráfico

In [ ]:
```
felinos.groupby(['genus'])['stateProvince'].nunique().sort_values(ascending=False).plo
```

Filtramos para graficar y mostrar en el mapa

- Grafiquemos los felinos vistos a partir del año **2010**
- **sorted_felinos**: es el dataframe con los datos ordenados por año

In [ ]:
```
sorted_felinos.year.unique()
```

In [ ]:
```
data_map  = sorted_felinos[sorted_felinos.year>2010]
data_map.shape
```

In [ ]:
```
data_map['year'].value_counts().plot(kind='bar')
```

In [ ]:
```
mapa = generate_map()
data_map.apply(add_marker, axis=1)
mapa
```

Usamos las funciones más arriba definidas para:

- generar el mapa
- definir los colores
- generar los puntos

Podemos graficar los felinos según los meses en que fueron vistos

In [ ]:
```
felinos.columns
```

In [ ]:
```
list_month = felinos.month.unique()
```

In [ ]:
```
list_month
```

In [ ]:
```
month_selected = 11
data_map = felinos[felinos.month==month_selected]
mapa = generate_map()
data_map.apply(add_marker, axis=1)
mapa
```

Veamos otros ejemplos, ahora con el dataset del **censo**

In [ ]:
```
censo_2022 = pd.read_csv(file_route/'c2022_tp_c_resumen_adaptado.csv')
```

In [ ]:
```
censo_2022.head()
```

Nos quedamos sin la primer fila, que tiene los datos totales de Argentina

In [ ]:
```
censo_2022 = censo_2022.iloc[1:]
```

```
censo_2022.head(3)
```

Podemos ver los datos **ordenados** con la cantidad de población de cada jurisdicción

In [ ]:
```
censo_2022.sort_values(by='Total de población', ascending=False).tail(3)
```

👉 Queremos mostrar en el mapa puntos en las capitales de las provincias

- generar los colores de los puntos según un rango de población.
- el dataset del **censo** no tiene datos de ubicación.
- vamos a **consultar** al dataset de **ar.csv**.
- ¿qué columnas son relevantes de cada dataset que nos permiten relacionar los datos que estamos buscando?

In [ ]:
```
arg_data = pd.read_csv(file_route/'ar.csv')
```

In [ ]:
```
censo_2022.columns
```

Censo

En el dataset **censo** en la columna:

- **Jurisdicción** está el nombre que debemos buscar en el dataset **arg_data**

In [ ]:
```
arg_data.columns
```

Geolocalizacón de ciudades de Argentina

En el dataset **arg_data** :

- los valores de la columna **capital** indican:
  - admin: es la ciudad capital de la provincia
  - primary: la capital de Argentina
- la columna **admin_name** contiene el nombre que tiene que coincidir con el que está en **Jurisdicción** de **censo**
- las columnas **lat** y **lng** contienen los datos para ubicar en el mapa

Del dataset **arg_data**, nos quedamos solamente con las capitales y CABA

In [ ]:
```
ref_admin = ['admin','primary']
arg_data_cap = arg_data[arg_data.capital.isin(ref_admin)]
arg_data_cap.tail(3)
```

Verificar las coincidencias

- verificar si todos los nombres del dataset de **censo** están en **arg_data**

In [ ]:
```
matches = censo_2022[censo_2022['Jurisdicción'].isin(arg_data_cap['admin_name'])]
print(f"cant de censo {censo_2022.shape[0]}")
print(f'cant de coincidencias {len(matches)}')
```

Encontrar los errores

Queremos saber cuáles son las filas del dataset **censo** que no encontraron coincidencias en **arg_data**

In [ ]:
```
no_matches = censo_2022[~censo_2022['Jurisdicción'].isin(arg_data_cap['admin_name'])]
no_matches
```

Encontremos cómo figura en **arg_data** para realizar el reemplazo en **censo**

In [ ]:
```
arg_data_cap[arg_data_cap['admin_name'].str.contains('Ciudad')]
```

Realizamos los dos reemplazos:

In [ ]:
```
replaces_values = {'Rio Negro':'Río Negro','Ciudad Autónoma de Buenos Aires': 'Buenos
censo_2022['Jurisdicción'] =censo_2022['Jurisdicción'].replace(replaces_values)
```

Realizar estas modificaciones de esta forma puede dar un **warning**, asegurémonos que las modificaciones se van a realizar de forma segura

In [ ]:
```
censo_2022= censo_2022.copy()
replaces_values = {'Rio Negro':'Río Negro','Ciudad Autónoma de Buenos Aires': 'Buenos
censo_2022['Jurisdicción'] =censo_2022['Jurisdicción'].replace(replaces_values)
```

In [ ]:
```
matches = censo_2022[censo_2022['Jurisdicción'].isin(arg_data_cap['admin_name'])]
print(f"cant de censo {censo_2022.shape[0]}")
print(f' cant de coincidencias {len(matches)}')
```

🧧 Agregar columnas de otro dataset

- vamos a agregar al dataset **censo** las columnas con geolocalización.
- veamos cómo agregar columnas en función de datos coincidentes en columnas de ambos dataset

¿Cómo lo hicieron sin pandas?

Para unir datos de dos datasets o más, usaremos la función **merge** de pandas.

🧧 Merge

In [ ]:
```
df_merged = censo_2022.merge(arg_data_cap, left_on='Jurisdicción', right_on='admin_nar
df_merged.head(4)
```

Vamos a ver cómo funciona y las opciones del **merge** con datos más simples

In [ ]:
```
# DataFrame 1: Información básica de personas
data1 = {
    'id': [1, 2, 3, 4],
    'name': ['Alicia', 'Gabriel', 'Justina', 'Leandro'],
    'department': ['Ing1', 'Objt1', 'Ing2', 'Objt2']
}
df1 = pd.DataFrame(data1)

# DataFrame 2: Información adicional sobre personas
data2 = {
    'id': [3, 4, 5, 6],
    'name': ['Justina', 'Leandro', 'Belén', 'Tomas'],
    'ranking': [70000, 80000, 90000, 100000],
    'prov': ['Entre Ríos', 'Formosa', 'Jujuy', 'Corrientes']
}
df2 = pd.DataFrame(data2)

print("DataFrame 1:")
print(df1)
```

```
print("\nDataFrame 2:")
print(df2)
```

Uso de merge

Tenemos diferentes criterios para unir datasets, son similares a las utilizadas en **sql**, veamos algunos:

- Inner
- Left
- Right
- Documentación

📌 1.En el caso de que las columnas para buscar las coincidencias se llamen iguales

- **Inner**: devuelve solo las filas que tienen valores coincidentes en ambas tablas.
- **Left**: devuelve todas las filas del DataFrame izquierdo y las filas coincidentes del DataFrame derecho. Si no hay coincidencia, los valores del DataFrame derecho serán NaN.
- **Right**: devuelve todas las filas del DataFrame derecho y las filas coincidentes del DataFrame izquierdo. Si no hay coincidencia, los valores del DataFrame izquierdo serán NaN.

Veamos con los dataframes creados

In [ ]:
```
df1
```

In [ ]:
```
df2
```

Inner

- **Inner**: devuelve solo las filas que tienen valores coincidentes en ambas tablas.

In [ ]:
```
inner_merged = pd.merge(df1, df2, on='id', how='inner')
print("\nResultado de Inner Join:")
inner_merged
```

Eliminar la columna utilizada para unir

In [ ]:
```
inner_merged = pd.merge(df1, df2, on='id', how='inner').drop(columns = ['name_y'])
print("\nResultado de Inner Join sacando la columna usada para unir:")
inner_merged
```

Left

- **Left**: devuelve todas las filas del DataFrame izquierdo y las filas coincidentes del DataFrame derecho. Si no hay coincidencia, los valores del DataFrame derecho serán NaN.

In [ ]:
```
left_merged = pd.merge(df1, df2, on='id', how='left')
print("\nResultado de Left Join:")
left_merged
```

Right

- **Right**: devuelve todas las filas del DataFrame derecho y las filas coincidentes del DataFrame izquierdo. Si no hay coincidencia, los valores del DataFrame izquierdo serán NaN.

In [ ]:
```
right_merged = pd.merge(df1, df2, on='id', how='right')
```

```
print("\nResultado Right Join:")
right_merged
```

📌 2.En el caso de que las columnas para buscar las coincidencias se llaman diferentes

Definamos nuevamente los dataframes con nombres diferentes en las columnas que usaremos para unir

In [ ]:
```
# DataFrame 1: Información básica de personas
data1 = {
    'id': [1, 2, 3, 4],
    'name': ['Alicia', 'Gabriel', 'Justina', 'Leandro'],
    'department': ['Ing1', 'Objt1', 'Ing2', 'Objt2']
}
df1 = pd.DataFrame(data1)

# DataFrame 2: Información adicional sobre personas
data2 = {
    'pers_id': [3, 4, 5, 6],
    'name': ['Justina', 'Leandro', 'Belén', 'Tomas'],
    'ranking': [70000, 80000, 90000, 100000],
    'prov': ['Entre Ríos', 'Formosa', 'Jujuy', 'Corrientes']
}
df2 = pd.DataFrame(data2)

print("DataFrame 1:")
print(df1)
print("\nDataFrame 2:")
print(df2)
```

Las opciones que debemos utilizar para estos casos son:

- **left_on='nombre'**: nombre de la columna del dataframe izquierdo.
- **right_on='nombre'**: nombre de la columna del dataframe derecho.

In [ ]:
```
# Inner Join with left_on and right_on
inner_merged_lr = pd.merge(df1, df2, left_on='id', right_on='pers_id', how='inner')
print("\nResultado de Inner Join con left_on y right_on:")
print(inner_merged_lr)

# Left Join with left_on and right_on
left_merged_lr = pd.merge(df1, df2, left_on='id', right_on='pers_id', how='left')
print("\nResultado de Left Join con left_on y right_on:")
print(left_merged_lr)

# Right Join with left_on and right_on
right_merged_lr = pd.merge(df1, df2, left_on='id', right_on='pers_id', how='right')
print("\nResultado de Right Join con left_on y right_on:")
print(right_merged_lr)
```

Unir dos dataframes:

- el del censo
- el que contiene datos geolocalizados

- del dataframe **arg_data** nos quedamos con las capitales y CABA: **arg_data_cap**
- filtramos las columnas que nos interesan
- luego hacemos el merge

In [ ]:
```
arg_data_cap = arg_data_cap[['admin_name','lat','lng']].copy()
arg_data_cap.head(3)
```

In [ ]:
```
df_merged = censo_2022.merge(arg_data_cap, left_on='Jurisdicción', right_on='admin_nar
df_merged.head(4)
```

Ya tenemos los datos unidos

📌 Agregar puntos en el mapa con color según criterio

Queremos cambiar los colores de los puntos en el mapa según la población

- ¿Cómo podemos definir los valores de los rangos a utilizar?
- ¿Qué función de pandas nos daba este tipo de información?

In [ ]:
```
censo_2022.describe()
```

Para hacer el mapa tenemos que realizar los siguientes pasos:

- generar al mapa base: **mapa = generate_map()**
- definir una función para asignar color según rango de población: **define_color_censo**
- definir una función, **add_marker** que realiza:
  - agrega los puntos indicando las columnas donde está la geolocalización
    - **lat**
    - **lng**
  - asigna el color usando la función **define_color_censo**
  - agrega el texto cuando se selecciona un punto, se utilizará la columna **Jurisdicción**.
- Aplicamos la función **add_marker** al dataframe: **apply**

In [ ]:
```
def define_color_censo(categoria):
    desc = df_merged.describe()
    percentile_25 = desc.loc['25%', 'Total de población']
    percentile_50 = desc.loc['50%', 'Total de población']
    percentile_75 = desc.loc['75%', 'Total de población']
    if categoria <= percentile_25:
        return 'blue'
    elif categoria <= percentile_50:
        return 'green'
    elif categoria <= percentile_75:
        return 'orange'
    else:
        return 'red'
def add_marker(row):
    color = define_color_censo(row['Total de población'])
    mark = folium.Marker(
        [row['lat'], row['lng']],
        popup=row['Jurisdicción'],
        icon=folium.Icon(color=color)
    )
    #print(mark.get_name())
    #print(mark.icon.render)
    mark.add_to(map)
map = generate_map()
df_merged.apply(add_marker, axis=1)
map
```

🫀 Mostrar multipolígonos en mapas

- Vimos cómo mostrar puntos ahora, veremos cómo mostrar áreas en el mapa.
- Trabajaremos con el dataset **area_protegida** que contiene en el dataset con estos datos.
- Utilizaremos **geopandas** para trabajar con los datos **MULTIPOLYGON**.
- Utilizaremos **shapely** para convertir datos string en datos conocidos para geopandas.

In [ ]:
```
import geopandas as gpd
from shapely import wkt
```

📌 Geopandas

- Es una liberría que permite manejar datos espaciales.
- instalar:
  `pip install geopandas`
- Las dos estructuras principales extienden dos clases de pandas:
  - geopandas.GeoDataFrame, subclase de pandas.DataFrame,
  - geopandas.GeoSeries, subclase de pandas.Series,
- Los datasets geoespaciales contienen información sobre ubicaciones geográficas.
- Pueden ser:
  - **puntos**: Representa ubicaciones individuales en un espacio 2D (o 3D).
  - **líneas**: Representa una serie de puntos conectados que forman una línea.
  - **polígonos**: Representa áreas cerradas, definidas por una serie de puntos conectados que forman un contorno cerrado.
  - **multipolígonos**: Representa múltiples polígonos en una sola entidad.
  - y más
- Al leer el dataset crea la columna **geometry** que es donde geopandas almacena estas geometrías.
- La columna geometry permite realizar operaciones geoespaciales como cálculos de distancias, uniones espaciales, intersecciones, etc.
- Geopandas puede detectar automáticamente la columna con datos geométricos, en el caso de datasets que tienen información con formato:
  - Shapefile
  - GeoJSON
  - otros formatos.

Y la convierte en una columna geometry del tipo **Geometry**.

- Documentación

Veamos un ejemplo de un dataset que tiene **multipolígonos**: Áreas protegidas

In [ ]:
```
geo_df = gpd.read_file(file_route/'area_protegida.csv')
geo_df.info()
```

In [ ]:
```
geo_df.geom.head()
```

Con la librería shapely convertimos los strings de la columna **geom** a formato geometry utilizando **wkt** (well Known text). El texto tiene que poder interpretarse para convertirse a tipo de datos de geopandas.

In [ ]:
```
geo_df['geometry'] = geo_df['geom'].apply(wkt.loads)
```

In [ ]:
```
geo_df.info()
```

Vamos a graficar sólos dos tipos de área protegida:

- Los tipos de reservas del dataset están en la columna **tap**:

In [ ]:
```
geo_df.tap.unique()
```

- **1**: Parque
- **2**: Reserva
- **3**: Monunmento Natural
- **0**: Información no disponible

In [ ]:
```
geo_df[geo_df.tap=='0']
```

In [ ]:
```
geo_df[geo_df.fna.str.contains('Puelo')]
```

Como ya vimos en varios ejemplos, los datos pueden contener errores, en algunos casos podemos arreglarlo, en otros casos, dependemos de un experto en el tema para definir la modificación

Convertimos a **int** los datos de la columna **tap**

In [ ]:
```
geo_df.tap = geo_df.tap.astype('int')
geo_df.tap.unique()
```

In [ ]:
```
geo_df.iloc[2:7]
```

Agreguemos al mapa las áreas correspondientes a:

- 1: Parque
- 2: Reserva

In [ ]:
```
list_types = [1,2]
geo_data_map = geo_df[geo_df.tap.isin(list_types)]
geo_data_map
```

In [ ]:
```
map = generate_map()
def add_multipoint(row):
    sim_geo = gpd.GeoSeries(row["geometry"]).simplify(tolerance=0.001)
    geo_j = sim_geo.to_json()
    geo_j = folium.GeoJson(data=geo_j, style_function=lambda x: {"fillColor": "orange"
    folium.Popup(row["nam"]).add_to(geo_j)
    geo_j.add_to(map)
```

Esta función realiza las siguientes acciones:

- genera un objeto de tipo **GeoSeries** de GeoPandas usando simplify.
- convertie estos datos al formato json que sabe interpretar folium.
- genera un objeto de tipo **GeoJson** en este caso con color **orange** para cada punto.
- le agrega una identificación a cad apunto, usando la columna **nam** de cada fila que le pasamos con **apply**.
- agrega al mapa cada área.

In [ ]:
```
geo_data_map.apply(add_multipoint, axis=1)

map
```

Agreguemos más datos a nuestro mapa

In [ ]:
```
lakes_arg = gpd.read_file(file_route/'lagos_arg.csv')
replace_col = {'Latitud en GD':'lat', 'Longitud en GD':'lng'}
lakes_arg = lakes_arg.rename(columns=replace_col)
lakes_arg.head(3)
```

In [ ]:
```
def add_marker_lake(row):
    folium.Marker(
        [row['lat'], row['lng']],
        popup=row['Nombre'],
        icon=folium.Icon(color='orange')
    ).add_to(map)
```

In [ ]:
```
def get_color(categoria):
    if categoria == 'Puma':
        return 'blue'
    elif categoria == 'Leopardus':
        return 'green'
    else:
        return 'red'
def add_marker_felinos(row):
    color = get_color(row['genus'])
    folium.Marker(
        [row['decimalLatitude'], row['decimalLongitude']],
        popup=row['genus'],
        icon=folium.Icon(color=color)
    ).add_to(map)
```

Ahora agregumos los felinos vistos, por ejemplo en el mes de **Noviembre**

In [ ]:
```
data_map_felinos = felinos[felinos.month==11]
```

In [ ]:
```
map = generate_map()
geo_data_map.apply(add_multipoint, axis=1)
data_map_felinos.apply(add_marker_felinos, axis=1)
lakes_arg.apply(add_marker_lake, axis=1)
map
```

Mapas en Streamlit

- Streamlit tiene una función para mostrar mapas usando **Mapbox**
- Otra opción es usar folium a través de la librería **streamlit_folium**, de la cual usaremos:

```
st_folium(map)
```

- instalar
  `streamlit_folium`

  * importamos
  ```python
  from streamlit_folium import  st_folium
  ```

- Documentación.

Agregamos las capas que queremos mostrar utilizando las funciones para agregar los puntos y luego mostramos el mapa en Streamlit con **st_folium(mapa)**

Iconos personalizados

In [ ]:
```
kw = {"prefix": "fa", "color": "green", "icon": "arrow-up"}
```

```
angle = 180
icon = folium.Icon(angle=angle, **kw)
folium.Marker(location=[-25.12, -58.18], icon=icon, tooltip=str(angle)).add_to(m)

angle = 45
icon = folium.Icon(angle=angle, **kw)
folium.Marker(location=[-25.17, -58.13], icon=icon, tooltip=str(angle)).add_to(m)

angle = 90
icon = folium.Icon(angle=angle, **kw)
folium.Marker([-31.88, -58.30], icon=icon, tooltip=str(angle)).add_to(m)
m
```

Para probar en casa

Gráfico con librería seaborn

- Es otra librería para visualización de datos.
- Instalación:

```
pip install seaborn
```

- Documentación

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
# Volver a calcular el número de localidades únicas para cada combinación de género y
num_localidades = felinos.groupby(['genus', 'stateProvince']).locality.nunique().reset

# Crear el gráfico de barras apiladas
plt.figure(figsize=(12, 7))
sns.barplot(x='stateProvince', y='locality', hue='genus', data=num_localidades, errorb
plt.title('Número de Localidades Únicas por Género y Provincia')
plt.ylabel('Número de Localidades Únicas')
plt.xlabel('Provincia')
plt.legend(title='Género')
plt.xticks(rotation=45)
plt.show()
```

In [ ]:
```
```