Seminario de Lenguajes - Python

Cursada 2024

Continuamos con el análisis de datos

- Pandas
- Matplotlib
- Mostramos todo en Streamlit

💢 ¿Qué vimos en la clase anterior?

En la clase pasada estuvimos viendo algunos conceptos sobre:

- Tipos de datos básicos de Pandas:

- Series
- Dataframe

Conocer el dataset:

- primeras o últimas filas
- cantidad de filas y columnas
- info de los datos y cálculos básicos estadísticos

- primeras: **head()**
- últimas: **tail()**
- filas y columnas: **shape**
- columnas: **columns**
- información de tipos de datos y cantidad de datos nulos: **info()**
- cálculos básicos de estadística: **describe()**

Acceso a los datos: posición numérica y por etiqueta

- posición numérica: **iloc**
- identificación por etiqueta: **loc**

Manipulación de Datos

- Filtrando los datos
- Modificación de nombres de columnas

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)
```

Reemplazos de valores en las celdas

In [ ]:
```
df_airports[df_airports.region_name.str.contains('Tierra del Fuego')]['region_name'].
```

¿Cómo reemplazamos?

In [ ]:
```
df_airports.region_name.replace('Tierra del Fuego Province', 'Tierra del Fuego, Antár
```

In [ ]:
```
df_airports[df_airports.region_name.str.contains('Tierra del Fuego')]['region_name'].
```

¿Qué significa **inplace==True**?

Realicemos una modificación de

- **'Río Negro Province'** -> **'Río Negro'**

In [ ]:
```
df_airports[df_airports.region_name.str.contains('Negro')]['region_name'].head(3)
```

In [ ]:
```
df_airports.region_name.replace('Río Negro Province', 'Río Negro')
```

¿Se realizó el cambio?

In [ ]:
```
df_airports[df_airports.region_name.str.contains('Negro')]['region_name'].head(3)
```

inplace hace el reemplazo sin tener que guardar el dataframe modificado en una nueva variable.

🧠 Renombrar las columnas

Renombrar la columna:

- **'name'** -> **'Nombre'**

Trabajemos con un dataframe más reducido.

In [ ]:
```
df_reduced =  df_airports[['type','name']]
```

¿Por qué usamos doble **[]**?

In [ ]:
```
df_reduced.rename(columns={'name': 'Nombre'}).head(3)
```

In [ ]:
```
df_reduced.columns
```

Usamos **inplace** si estamos seguros de los cambios que vamos a realizar

In [ ]:
```
df_reduced.rename(columns={'name': 'Nombre'}, inplace=True)
```

In [ ]:
```
df_reduced
```

Este **warning** nos indica que Pandas no puede asegurar que resultados sean los esperados, ¿cómo podemos asegurar que no haya problemas?

📌 Opción 1: Hacer una copia explícita, usando **copy()**

In [ ]:
```
df_reduced_copy = df_airports[['type', 'name']].copy()
df_reduced_copy.rename(columns={'name': 'Nombre'}, inplace=True)

df_reduced_copy.head(3)
```

📌 Opción 2: Asignar el resultado a una variable nueva

In [ ]:
```
df_reduced_var = df_airports[['type','name']]
df_reduced_var = df_reduced_var.rename(columns={'type': 'Tipo'})
df_reduced_var.head(3)
```

¿Y si queremos renombrar varias columnas?

In [ ]:
```
df_reduced_var = df_airports[['type', 'name', 'latitude_deg', 'longitude_deg']]
replace_columns = {'name':'Nombre', 'latitude_deg': 'Latitud', 'longitude_deg': 'Longi
df_reduced_var.rename(columns=replace_columns).head(3)
```

Y si queremos renombrar todas las columnas:

In [ ]:
```
df_reduced_var.columns = ['Tipo','Nombre','Latitud']
                            #, 'Longitud']
```

La cantidad de elementos de la **lista** debe coincidir con la cantidad de **columnas**

In [ ]:
```
df_reduced_var.columns = ['Tipo','Nombre','Latitud', 'Longitud']
df_reduced_var.head(3)
```

🧰 Modificación de tipo de datos

¿Y si queremos cambiar el tipo de dato para una columna?

Veamos la columna **last_updated**

In [ ]:
```
df_airports.last_updated
```

- Contiene datos de fechas, Pandas le asignó **object**, un tipo de datos general.
- Queremos verificar los años en que se realizaron actualizaciones.
  - ¿Con qué tipos de datos trabajamos en Python las fechas?

In [ ]:
```
date_updated = pd.to_datetime(df_airports.last_updated)
date_updated
```

In [ ]:
```
date_updated.dt.year.unique()
```

El atributo **.dt** en Pandas se usa para acceder a las propiedades de los datos datetime, como:

- años
- meses
- días
- horas
- minutos, etc.
- En este caso, **.dt.year** se utiliza para obtener el año de cada fecha en la variable que contiene fechas.

En el ejemplo anterior:

`date_updated = pd.to_datetime(df_airports.last_updated)`
Estamos trabajando con una variable **Series**, no con los datos propios del dataframe.

📌Modifiquemos el tipo de dato de la columna del dataframe

In [ ]:
```
df_airports.last_updated = pd.to_datetime(df_airports.last_updated)
df_airports[['name', 'last_updated']].head(3)
```

In [ ]:
```
df_airports.last_updated.head(3)
```

In [ ]:
```
df_airports.last_updated.dt.month.unique()
```

Se modificó una columna con tipo de dato **object** a **datetime** de Pandas usando:

- pd.to_datetime(columna)

¿Qué otras opciones tenemos?

In [ ]:
```
pd.to_
```

¿Y si queremos realizar otro tipo de modificación? Veamos la columna **elevation__ft**:

In [ ]:
```
df_airports.elevation_ft
```

- ¿A qué tipo de dato se podría cambiar?
- ¿Por qué haríamos el cambio?

Para cambiar a tipos de datos como int, float ...usamos

`df.column.astype(..: int, float, bool, str`
Veamos el código y qué pasa:

In [ ]:
```
df_airports.elevation_ft.astype(int)
```

- El problema es que tenemos valores nulos o infinitos, (Nan o Inf).
- Podemos asignarles un valor, por ejemplo 0, pero generará análisis erróneos cuando se involucre este valor.
- Podemos optar por:
  - eliminar las filas que tienen **non-finite values (NA or inf)**, ojo verificar que no perdemos información.
  - en el caso que se necesite trabajar con estos valores, nos podemos quedar con un subconjunto del dataset sin las filas que contienen valores nulos.
  - si no necesitamos evaluar los datos de esta columna, no la modificamos y trabajamos con todo el dataset .

Veamos la opción de sacar las filas que contienen valores nulos:

`df.dropna()`
Nos permite eliminar estos casos del dataframe.

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

Quedaron sólo 3 filas!!!!

📌 Eliminación de datos nulos

En realidad no se eliminaron porque no modificamos de forma definitiva el Dataframe.

`df.dropna()`
Elimina **todas** filas que contengan en **algunas** de las columnas valores nulos, y vimos que algunas columnas tienen muchos como:

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

Para eliminar las filas que contienen valores nulos **sólo de una columna**:

In [ ]:
```
df_airports_mod = df_airports.dropna(subset=['elevation_ft']).copy()
```

Para no perder los datos originales, realizamos una **copia** sacando los valores nulos de la columna **elevation_ft**

In [ ]:
```
df_airports_mod['elevation_ft'] = df_airports_mod.elevation_ft.astype(int)
df_airports_mod[['name','elevation_ft']].head(3)
```

¿Cuántos valores nulos hay en cada columna ahora?

In [ ]:
```
df_airports_mod.info(memory_usage='deep')
```

In [ ]:
```
df_airports.info(memory_usage='deep')
```

📌 Cálculos mátemáticos que nos pueden interesar...

- mean
- min
- max
- sum

In [ ]:
```
df_airports_mod.elevation_ft.min()
```

Queremos encontrar los aeropuertos cuya elevación sea menor al promedio.

In [ ]:
```
df_airports_mod[df_airports_mod.elevation_ft<df_airports.elevation_ft.mean()][['name'
```

¿Por qué usamos dos veces la variable del **dataframe**?

🎁 Crear nuestro dataframe

In [ ]:
```
recognized_women_info = {
    'Cecilia Berdichevsky': {'Area': 'Informática', 'Fecha_nacimiento': '1960-05-10'}
    'Noemí García': {'Area': 'Informática', 'Fecha_nacimiento': '1965-11-30'},
    'Cecilia Grierson': {'Area': 'Ciencia', 'Fecha_nacimiento': '1859-11-22'},
    'Julieta Lanteri': {'Area': 'Ciencia', 'Fecha_nacimiento': '1873-11-22'},
    'Jeanette Campbell': {'Area': 'Deporte', 'Fecha_nacimiento': '1965-03-20'},
    'Mary Terán': {'Area': 'Deporte', 'Fecha_nacimiento': '1936-01-23'},
    'Patricia Sosa': {'Area': 'Música', 'Fecha_nacimiento': '1956-01-23'},
    'Fabiana Cantilo': {'Area': 'Música', 'Fecha_nacimiento': '1959-03-03'}
}
```

- Convertir el diccionario a DataFrame

In [ ]:
```
women_df = pd.DataFrame.from_dict(recognized_women_info, orient='index')

women_df
```

orient=index: indica que las claves del diccionario deben sen interpretadas como índice del dataframe que se está

- Convertir el índice en columna como parte del Dataframe

In [ ]:
```
women_df.reset_index(inplace=True)
women_df.rename(columns={'index': 'Nombre'}, inplace=True)
women_df.index.name = 'index'
women_df
```

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

Accedemos a filas por número de fila y por etiquetas

In [ ]:
```
women_df.iloc[2]
```

In [ ]:
```
women_df.loc[2]
```

Generamos el Dataframe con etiquetas diferentes a números

In [ ]:
```
index_list = ['w' + str(i) for i in range(1, len(recognized_women_info) + 1)]
women_df = pd.DataFrame.from_dict(recognized_women_info, orient='index')
women_df.reset_index(inplace=True)
women_df.rename(columns={'index': 'Nombre'}, inplace=True)
women_df.index = index_list
women_df
```

Ahora el acceso con **iloc** y **loc** cambia

In [ ]:
```
women_df.iloc[0]
```

In [ ]:
```
women_df.loc[0]
```

Da error porque **0** no es un índice válido

In [ ]:
```
women_df.index
```

In [ ]:
```
women_df.loc['w1']
```

🚨 Desafío para resolver en casa

- Reemplazar en el dataset de Conectividad las celdas que contengan el carácter '-' con la palabra 'NO' usando Pandas

🧯 Agregar columna con datos de otras columnas

Queremos que agregar una columna **height** que contenga los valores:

- bajo

- medio
- alto

según los rangos de valores de la columna **elevation_ft**

Tenemos que **recorrer** el dataset y asignar el valor correspondiente, podemos hacerlo, recorriendo...usando **iterrrows**

In [ ]:
```
df_airports_height = df_airports.copy()
heights = []
for index, row in df_airports_height.iterrows():
    value = row['elevation_ft']
    if value < 131:
        height = 'bajo'
    elif value < 903:
        height ='medio'
    else:
        height = 'alto'

    # Añadir el resultado a la lista
    heights.append(height)
df_airports_height['height'] = heights
df_airports_height[['name','elevation_ft','height']].head(4)
```

¿Cómo nos queda el dataset con respecto a los valores nulos de cada columna?

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

Pero si hay algo que nos gusta de Pandas es no tener que **recorrer** manualmente, entonces ¿cómo hacemos?

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

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

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

Apply de Pandas permite aplicar una función a todas las filas de una columna

In [ ]:
```
df_airports_height.head(4)
```

De esta forma en lugar de usar **iterrows**, utilizamos **apply** con una función definida que genera el dato necesario.

🚨 Desafío para resolver en casa

Mejorar los cambios en el caso de valores **nan** y el cambio de tipo de datos

¿Qué pasó en estos casos?:

```
df_airports_height.iloc[926]
```

- ¿Está bien el valor correspondiete a la columna **'altura'**?
- Investigar:
  - la conversión al tipo de datos: pd.Int64Dtype()
  - la función **pd.isna** para consultar si un valor es **nan**

Cuando se convirtió a int, ¿se perdieron decimales?

- ¿Cómo podemos verifircarlo? Investigar:
  - **value.is_integer**

Abajo les dejo dos consultas de los errores que se generaron

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

In [ ]:
```
df_airports_height[df_airports_height.elevation_ft.isnull()][['name','elevation_ft','
```

Agregar columna en función de dos columnas del Dataframe

Queremos hacer la definición de los datos de la nueva columna en función de dos **columnas**

Hacemos una copia del dataframe y nos quedamos solamente con:

- 'large_airport'
- 'medium_airport'
- 'small_airport'

In [ ]:
```
columnas = ['large_airport', 'medium_airport', 'small_airport']
df_airports_height = df_airports[df_airports.type.isin(columnas)].copy()
```

In [ ]:
```
df_airports_height.type.unique()
```

In [ ]:
```
df_airports_height = df_airports[df_airports.type.isin(columnas)].copy()
def define_height(value):
    match value:
        case _ if value <= 131:
            return "bajo"
        case _ if value <= 903:
            return "medio"
        case _:
            return "alto"
def type_height(value, type_airport):
    text_value = define_height(value)
    match value:
        case _ if type_airport == 'large_airport':
            return 'large_'+text_value
        case _ if type_airport =='medium_airport':
            return 'medium_'+text_value
        case _:
            return 'small_'+text_value
```

In [ ]:
```
df_airports_height['height'] = df_airports_height.apply(lambda x: type_height(x.eleva
```

In [ ]:
```
df_airports_height[['name','type','elevation_ft','height']].head(4)
```

apply en este caso, no se especifica que se aplique a una sola columna porque necesitamos acceder a **dos columnas**.

```
df_airports_height.apply(lambda x: type_height(x.elevation_ft, x.type),
```

```
axis=1)
```

- **axis=1**: indica que se aplique la función a cada fila
- **axis=0**, la opción por **defecto** lo hace sobre una columna

Nuestros datos y gráficos en Streamlit

- Vamos a trabajar con un archivo que cuenta con un registro de avistajes de felinos en nuestro país
- Descargado del sitio de gbif y adaptado para poder trabajar estos datos

In [ ]:
```
felinos = pd.read_csv(file_route/'felinos_filtrado.csv')
```

Veamos que datos tiene

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

In [ ]:
```
felinos.head(3)
```

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

Queremos conocer en qué cantidad de provincias se avistaron los **diferentes** felinos

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

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

Arreglemos algunos problemas

In [ ]:
```
felinos.stateProvince = felinos.stateProvince.replace('Neuquen','Neuquén')
```

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

Necesito saber por cada tipo de felino, dato que tengo en **genus**, en cuántas provincias **diferentes** está.

🥊 Agrupamientos

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

groupby permite agrupar por una columna o más, pero necesita ir acompañado de alguna operación de agregación o transformación para poder visualizar el resultado correspondiente.

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

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

- **unique** indica los valores únicos de la agrupación que hicimos en función de la columna que indicamos
- **nunique** indica la cantidad de esa columna, en este caso las provincias. Podemos visualizar las cantidades de cada columna, no sólo de la provincia.

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

O podemos obtener qué cantidad de **diferentes tipos** de felinos se vieron en cada **provincia**.

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

Métodos de agregación

Dijimos que **groupby** tiene que estar acompañado de alguna operación de agregación, algunas de las que se puede utilizar son:

- sum(): Suma los valores de cada grupo.
- count(): Cuenta el número de elementos en cada grupo.
- mean(): Calcula la media de los valores en cada grupo.

Ejemplos usando **count** , que cuenta la cantidad según los valores de columna **genus**.

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

También podemos agrupar por dos columnas, y realizar la operación sobre una tercera columna que nos interesa, en este caso:

- agrupamos por **genus** primero.
- después por cada tipo de genus agrupa por **provincia**.
- y luego muestra la cantidad de valores únicos por localidad.
- Es decir la cantidad de localidades diferentes de cada provincia que está en el Dataframe, donde se vieron los tipos diferentes de felinos.

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

🎁 Graficamos

Vamos a ver algo más de Matplotlib y la librería Plotly

📌 Matplotlib

Grafiquemos las agrupaciones de datos hechas entre los tipos de felinos y las provincias.

In [ ]:
```
import matplotlib.pyplot as plt
```

In [ ]:
```
import numpy as np
genus_unique_province =felinos.groupby('genus')['stateProvince'].nunique()
province_unique_genus = felinos.groupby('stateProvince')['genus'].nunique()

# Configurar los datos para el gráfico de torta
labels_genus = genus_unique_province.index
sizes_genus = genus_unique_province.values

# Configurar los datos para el gráfico de barra
labels_province = province_unique_genus.index
sizes_province = province_unique_genus.values
posicion = np.arange(len(labels_province))
 # Crearsubplots
fig,ax = plt.subplots(1,2,figsize=(14 , 6))
paleta = plt.get_cmap('coolwarm')
ax[0].set_prop_cycle("color", [paleta(1. * i/len(labels_genus))
                               for i in range(len(labels_genus))])
ax[0].pie(sizes_genus , labels=labels_genus, autopct='%.1f%%', labeldistance=1.15, sta
```

```
ax[0].set_title('Porcentaje de provincias donde se vieron felinos ')

ax[1].bar(posicion , sizes_province, tick_label=labels_province)
ax[1].set_title('Cantidad de tipos de felinos diferentes vistos en cada provincia ')
ax[1].set_xticklabels(labels_province, rotation=45)

# Mostramos el gráfico
fig.tight_layout()
plt.show()
```

Algunas de las funciones mostradas:

```
fig,ax = plt.subplots(1,2,figsize=(14 , 6))
```

- **plt.subplots**: permite identificar la figura y el gráfico en sí, para poder personalizar cada uno. Podríamos tener varios gráficos en una misma figura y personalizar cada uno por separado.
  - variable **fig**: la figura que contiene ambos gráficos
  - **1** fila
  - **2** columnas
  - **figsize**: tamaño en pulgadas, 14 de ancho y 6 de alto
  - variable **ax**: un arreglo con la cantidad de gráficos especificados, ubicados en posición 0 y posición 1
- **plt.get_cmap**: define una paleta de colores específica
- **set_prop_cycle**: permite definir los colores de la paleta en función de la cantidad de porciones que se van a mostrar
- **labels**: las etiquetas para cada porción del gráfico de torta(pie)
- **labeldistance**: ubica las etiquetas a una distnacia específica desde el centro del círculo
- **tick_label**: las etiquetas para cada barra del gráfico (bar)
- **fig.tight_layout()**: ajusta los gráficos al tamaño de la figura

📌 Plotly

Lo que diferencia a Plotly de Matplotlib es que utiliza JavaScript, lo que permite una mayor interacción con los gráficos. Además, esta biblioteca está basada en la biblioteca d3.js y stack.gl.

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

Gráfico usando plotly.express

Hagamos un gráfico simple, de líneas de las cantidad de felinos vistos por año.

Primero vamos a modificar el tipo de datos de la columna **year**, como tiene valores nulos las eliminamos a esas filas.

In [ ]:
```
felinos_mod = felinos.dropna(subset=['year']).copy()
felinos_mod['year'] = felinos_mod['year'].astype(int)
dat_valores =felinos_mod.groupby('year')['genus'].count()
years = dat_valores.index
values = dat_valores.values
```

In [ ]:
```
import plotly.express as px

fig = px.line(felinos_mod, x=years, y=values, title='Eventos por Año')
fig.show()
```

Gráficos utilizando plotly.graph_objects (go.Figure)

Esta opción permite personalizar y generar gráficos con funcionalidades avanzadas.

Ejemplo de un **gráfico de torta** de los valores generados con las agrupaciones ya realizadas entre las columnas **genus** y **stateProvince**.

In [ ]:
```
import plotly.graph_objects as go
genus_unique_province = felinos.groupby('genus')['stateProvince'].nunique()
# Configurar los datos para el gráfico de torta
labels_genus = genus_unique_province.index
sizes_genus = genus_unique_province.values

fig_pie = go.Figure(data=[go.Pie(labels=labels_genus, values=sizes_genus, hole=0.3,
                                 hoverinfo="label+percent")] )
fig_pie.update_layout(title="Cantidad de provincias donde se vieron felinos")
fig_pie.show()
```

Veamos el significado de algunas de las opciones

- importamos las librerías:

```
import plotly.graph_objects as go
```

- **go.Figure()**: Crea una figura de Plotly que contiene data y diseño.
  - **data=**

```
go.Pie(labels=labels_genus, values=sizes_genus, hole=0.3,
                                 hoverinfo="label+percent"
```

- **go.Pie()** : genera una gráfico de torta.
  - labels: las etiquetas de los valores representados.
  - values: los valores par calcular los porcentajes.
  - hole: valor opcional que determina el tamaño de un agujero en el medio.
  - hoverinfo: lo que se va a mostrar cuando se pasa el mouse sobre cada porción. En este caso hace una combinación de la etiqueta más el porcentaje.
  - otras no utilizadas: **textinfo**, **textposition**, **marker**.
- **update_layout()**: permite la personalización.
- **fig.show()**: mostrar la figura.

Opciones de hoverinfo

- label: Muestra la etiqueta (nombre) de la porción.
- value: Muestra el valor numérico de la porción.
- percent: Muestra el porcentaje que representa la porción del total.
- name: Muestra el nombre de la traza.
- text: Muestra el texto asociado con la porción, si se ha definido.
- none: No muestra ninguna información de hover.
- all: Muestra toda la información disponible.

Gráfico de barras con Plotly

In [ ]:
```
fig_bar =  go.Figure(data=go.Bar(x=labels_province, y=sizes_province,
                    marker=dict(color='DarkSlateGrey'),
                    text=labels_province,
                    hoverinfo='x+y',
                    name='Cantidad de tipos de felinos diferentes vistos en cada pro
```

```
# Actualizar diseño y mostrar la figura
fig_bar.update_layout(title="Gráficos de felinos",
                  xaxis_title="Provincia",
                  yaxis_title="Cantidad",
                  barmode='group',
                  bargap=0.15,
                  bargroupgap=0.1)
```

Algunas de las opciones que cambian para un gráfico de barra

```
go.Bar(x=labels_province, y=sizes_province,
                     marker=dict(color='skyblue'),
                     text=labels_province,
                     hoverinfo='x+y',
                     name='Cantidad de tipos de felinos diferentes vistos en
cada provincia'))
```

- **x**: las etiquetas del eje x
- **y**: los tamaños de las barras
- **marker**: un diccionatio con opciones para personalizar la visualización como :
  - **color**
  - **size**
  - **line=** para un gráfico de línea: line=dict
- **name**: para definir el nombre de la traza, en caso que haya más de una, se identifican de forma individual.

```
fig_bar.update_layout(title="Gráficos de felinos",
                  xaxis_title="Provincia",
                  yaxis_title="Cantidad",
                  barmode='group',
                  bargap=0.15,
                  bargroupgap=0.1)
```

- **yaxis_title - xaxis_title**: título del eje **y** y **x**
- **barmode='group'**: indica la forma en que se van agrupar las barras en el gráfico
  - 'group': una al lado de la otra
  - 'stack': una arriba de la otra
- **bargap**: separación que hay entre las barras
- **bargroupgap**: separaciń entre grupos de barras.

Gráfico con Plotly con varias trazas

Les dejo el código y generación de gráfico que muestra la cantidad de los diferentes felinos vistos por cada mes del año, haciendo un gráfico de barra comparativo entre los 3 tipos de felinos

Agrupo por tipo de felino y por mes

In [ ]:
```
#primero conviero a int la columna month
felinos_month_int = felinos.dropna(subset='month').copy()
felinos_month_int.month = felinos_month_int.month.astype(int)

observations_by_month = felinos_month_int.groupby(['genus', 'month']).size().unstack(
observations_by_month
```

Genero cada barra con los valores de cada tipo de felino, indicando en **name** el nombre correspondiente

In [ ]:
```
fig = go.Figure()

for genus in observations_by_month.index:
    fig.add_trace(go.Bar(
        x=observations_by_month.columns,
        y=observations_by_month.loc[genus],
        name=genus,
        marker=dict(line=dict(width=1))
    ))

# Actualizar el diseño de la figura
fig.update_layout(
    title="Observaciones de Felinos por Mes",
    xaxis_title="Mes",
    yaxis_title="Cantidad de Observaciones",
    xaxis=dict(
        tickmode='array',
        tickvals=list(range(1, 13)),  # Asegurar que se muestran todos los meses del 
        ticktext=['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio', 'Ag
    ),
    barmode='group',  # Agrupar las barras una al lado de la otra
    bargap=0.15,      # Separación entre barras individuales
    bargroupgap=0.1   # Separación entre grupos de barras
)

# Mostrar el gráfico
fig.show()
```

unstack(fill_value=0): para separar los datos luego del groupby y la opción fill_value permite completar con un valor en los casos que no contengan datos, si no se pasa la opción se completa con NaN.

Veamos cómo cambian la disponibildiad de los datos sin usar **unstack**

In [ ]:
```
felinos_month_int.groupby(['genus', 'month']).size().unstack(fill_value=0)#
#felinos_month_int.groupby(['genus', 'month']).size()
```

Análisis y gráficos en Streamlit

Podemos generar diferentes tipos de gráficos en Streamlit:

- gráficos simples: le pasamos los datos directamente
  - **bar_chart**
  - **st.line_chart**
  - otros
- gráficos a través de librerías específicas: generamos la figura con la librería y luego la mostramos con las funciones de cada librería en Streamlit:
  - Matplotlib: **st.pyplot(figura)**
  - Plotly: **st.plotly_chart(figura)**
  - muchas más.
  - En el caso de usar las librerías específicas para graficar, la generación es igual que fuera de Streamlit, lo que necesitamos para verla es la función propia para cada librería.

Interacción con widgets

Los datos o gráficos que mostramos pueden generarse en función de elecciones que haga el usuario, algunos de los widgets que nos permiten esta interacción son:

- **st.multiselect**: da la opción de elegir varios valores de una lista
  `st.multiselect('Título', [datos])`
- **st.selectbox**: se puede seleccionar una sola opción
  `st.selectbox('Título', [datos])`
- **st.select_slider**: valores únicos
  `st.select_slider('Título', options=opciones)`
- **st.slider** rango de valores:(minimo, maximo) es un tupla que representa los valores seleccionados
  `st.slider('Rango de Años', minino, maximo, (minimo, maximo))`

In [ ]: