

Análisis de Datos con Python

Clase 06 - Ejercicio Práctico

✓ Actividad 1.1 - Filtrar pasajeros sobrevivientes

Consigna: Identificar y extraer a los pasajeros que **sobrevivieron** al naufragio.

Estrategia de resolución:

1. Cargar el dataset `titanic` desde `seaborn` en un `DataFrame` llamado `df_titanic`.
2. Explorar rápidamente la columna `survived` (valores 0 = no, 1 = sí) y verificar si hay valores faltantes.
3. Filtrar las filas donde `survived == 1` y guardar el resultado en `df_survivors`.
4. Comprobar tamaño y una vista preliminar para validar el filtrado.

Criterios de validación (checklist):

- `df_survivors["survived"].unique()` debe devolver solo `1`.
- La cantidad de filas de `df_survivors` debe ser menor o igual a la de `df_titanic`.
- No deben aparecer `NaN` en `survived` (si hubiera, se reporta).

```
# Importamos las librerías
import pandas as pd
import seaborn as sns

# 1) Cargar dataset en df_titanic
df_titanic = sns.load_dataset('titanic')
```

```
# 2) Exploración rápida de la columna 'survived'
print("Columnas del dataset:", list(df_titanic.columns), "\n")
print("Tipos de datos:\n", df_titanic.dtypes, "\n")

# Conteo de valores en 'survived' (incluye NaN si los hay)
print("Valores únicos en 'survived':", sorted(df_titanic['survived'].dropna().unique().tolist()))
print("Conteo de 'survived' (0=no, 1=sí):\n", df_titanic['survived'].value_counts(dropna=False), "\n")

# Chequeo de faltantes en 'survived'
na_survived = df_titanic['survived'].isna().sum()
print(f"Cantidad de valores faltantes en 'survived': {na_survived}\n")
```

```
# 3) Filtrado de sobrevivientes: survived == 1
df_survivors = df_titanic[df_titanic['survived'] == 1].copy()
```

De todas las filas que tiene el dataset, esta operación devuelve **solo aquellas en las que `survived` vale 1 (los que sobrevivieron)** y, además, **crea una copia independiente** de ese subconjunto (con las mismas columnas y el índice original), de modo que cualquier cambio posterior no afecte al `df_titanic` ni dispare advertencias.

```
# 4) Validaciones y vista preliminar
print("Shape original (df_titanic):", df_titanic.shape)
print("Shape filtrado (df_survivors):", df_survivors.shape, "\n")

# Validación de que solo quede el valor 1 en la columna 'survived'
valores_unicos_survivors = df_survivors['survived'].dropna().unique()
print("Valores únicos de 'survived' en df_survivors:", valores_unicos_survivors)

# Muestra rápida de las primeras filas de sobrevivientes
print("\nPrimeras 5 filas de df_survivors:")
print(df_survivors.head(5).to_string(index=False))
```

En la segunda parte del cuadro anterior, de todos los valores de la columna `survived` en `df_survivors`, se **descarta los `NaN`** y **devuelve solo los distintos** que aparecen (como un `ndarray` de NumPy).

El último `print()` muestra, de todas las filas que tiene `df_survivors`, solo las primeras 5 que ha convertido en una tabla de texto (no en un `DataFrame`), ocultando la columna de índice (`index=False`)

✓ Actividad 1.2 - Seleccionar columnas relevantes

Consigna: De los **sobrevivientes**, extraer solo las columnas **sex**, **age** y **fare**.

Estrategia de resolución:

1. Partimos de `df_survivors` (que filtramos hace 5 minutos).
2. Creamos un nuevo DataFrame `df_survivors_sel` con las tres columnas: `df_survivors[['sex', 'age', 'fare']]`.
3. Mantenemos los valores faltantes tal como están (especialmente en `age`, que suele tener `NaN`) y **no imputamos** en esta etapa.
4. Validamos estructura y coherencia:
 - Verificar `shape`, tipos de datos y conteo de nulos por columna.
 - Mostrar un vistazo de las primeras filas para confirmar el recorte.

Checklist de validación:

- `df_survivors_sel.columns.tolist()` debe ser `['sex', 'age', 'fare']`.
- `len(df_survivors_sel)` debe coincidir con `len(df_survivors)`.
- Reportar `df_survivors_sel.isna().sum()` para conocer faltantes antes de avanzar.

```
# 1) Selección de columnas
columnas_interes = ['sex', 'age', 'fare']
```

```
# Tomamos todas las filas y las columnas de la lista, y las copiamos
df_survivors_sel = df_survivors.loc[:, columnas_interes].copy()
```

```
# 2) Validaciones básicas
print("Columnas esperadas:", columnas_interes)
print("Columnas obtenidas:", df_survivors_sel.columns.tolist(), "\n")

print("Shape de df_survivors:", df_survivors.shape)
print("Shape de df_survivors_sel:", df_survivors_sel.shape)
print("¿Mismo número de filas?", len(df_survivors_sel) == len(df_survivors), "\n")

print("Tipos de datos en df_survivors_sel:\n", df_survivors_sel.dtypes, "\n")

print("Conteo de valores faltantes por columna:")
print(df_survivors_sel.isna().sum(), "\n")
```

```
# 3) Vista preliminar
print("Primeras 5 filas de df_survivors_sel:")
print(df_survivors_sel.head(5).to_string(index=False))
```

✓ Actividad 1.3 - Crear una columna categórica con la clase del pasajero

Consigna: Agregar al DataFrame de **sobrevivientes** una columna que clasifique a cada pasajero según la **clase en la que viajaba**, y **tiparla como categórica**.

Estrategia de resolución:

1. Partimos de `df_survivors_sel` (con `sex`, `age`, `fare`).
2. Tomamos la columna `class` desde `df_survivors` y la mapeamos a etiquetas en español:
 - `First -> "Primera"`, `Second -> "Segunda"`, `Third -> "Tercera"`.
3. Definimos un **orden** socioeconómico lógico en la categoría: `Tercera < Segunda < Primera`.
4. Añadimos la columna como **categórica ordenada** llamada `clase_cat` a `df_survivors_sel`.
5. Validamos:
 - `df_survivors_sel['clase_cat'].dtype` debe ser `category`.
 - `df_survivors_sel['clase_cat'].cat.categories` debe ser `['Tercera', 'Segunda', 'Primera']` (ordered).
 - `value_counts()` para revisar la distribución.

```
# 1) Definir el mapeo de etiquetas EN -> ES
mapa_clase = {
    'First': 'Primera',
    'Second': 'Segunda',
    'Third': 'Tercera'
}
```

```
}
```

```
# 2) Traer la columna 'class' desde df_survivors y mapear a español
# Usamos .map() para reemplazar etiquetas. Si hubiera algún valor no reconocido, quedaría NaN.
clase_es = df_survivors['class'].map(mapa_clase)

# Chequeo rápido de posibles faltantes tras el mapeo (no esperamos NaN en este dataset)
faltantes_mapeo = clase_es.isna().sum()
print(f"Faltantes en la clase luego del mapeo: {faltantes_mapeo}")
```

```
# 3) Crear una CATEGORÍA ORDENADA
# Definimos el orden socioeconómico: Tercera < Segunda < Primera
orden_categorias = ['Tercera', 'Segunda', 'Primera']

# Convierte clase_es en una variable categórica ordenada de pandas con esas
# categorías, habilitando comparaciones y ordenamientos coherentes por nivel
# socioeconómico.
clase_cat = pd.Categorical(clase_es, categories=orden_categorias, ordered=True)
```

Haz doble clic (o ingresa) para editar

```
# 4) Agregar la columna categórica al DataFrame seleccionado
df_survivors_sel['clase_cat'] = clase_cat
```

```
# 5) Validaciones y vista preliminar
print("\nValidaciones de 'clase_cat':")
print("dtype de 'clase_cat':", df_survivors_sel['clase_cat'].dtype)
print("Categorías:", df_survivors_sel['clase_cat'].cat.categories.tolist())

# Devuelve un booleano que indica si la columna categórica clase_cat
# está marcada como ordenada.
print("¿Es ordenada?", df_survivors_sel['clase_cat'].cat.ordered)

print("\nDistribución por clase (conteo):")
# sort=False respeta el orden categórico
print(df_survivors_sel['clase_cat'].value_counts(sort=False))

print("\nDistribución por clase (porcentaje):")
print((df_survivors_sel['clase_cat'].value_counts(normalize=True, sort=False) * 100).round(2).astype(str) + "%")

print("\nPrimeras 5 filas con 'clase_cat':\n")
print(df_survivors_sel.head(5))
```

En la *Distribución por clase (porcentaje)* se toma la columna `clase_cat` y se cuenta cuántos sobrevivientes hay en cada clase, pero en vez de dar números crudos, se calcula el porcentaje que representa cada clase dentro del total.

Como queremos respetar el orden de las categorías (Tercera, Segunda, Primera), no se reordenan los resultados. Luego, se convierten esos porcentajes a "porcientos": se multiplica por 100, se redondea a 2 decimales, se los pasa a texto y les pega el símbolo "%" al final. Por último, imprime esa serie. Todo en una línea! :)

Cierre de la Actividad 1

Lo que hicimos (idea de lo que puede ser un "reporte" o informe):

- Filtramos a quienes **sobrevivieron** (`survived == 1`) y validamos el resultado.
- Seleccionamos las columnas **sex**, **age** y **fare** para un foco inicial del análisis.
- Agregamos **clase_cat** como **categoría ordenada** (`Tercera < Segunda < Primera`) a partir de la columna original `class`.

Estado actual del DataFrame de trabajo: `df_survivors_sel` Columnas: `['sex', 'age', 'fare', 'clase_cat']`

Observaciones: `len(df_survivors_sel) == 342` (según vimos en la salida)

Nota: `age` conserva **faltantes** (52), lo cual es esperable y se tratará más adelante de ser necesario.

✓ Actividad 2.1 - Obtener los nombres de las columnas

Consigna: Listar los nombres de columnas del **DataFrame original** `df_titanic`.

Estrategia de resolución:

1. Trabajaremos **sobre** `df_titanic` (sin filtros).

2. Mostraremos los nombres de columnas con dos métodos equivalentes para que el alumno compare:

- `df_titanic.columns.tolist()`
- `df_titanic.keys().tolist()` (atajo)

3. Guardaremos esta lista en una variable (`cols_titanic`) para reutilizarla en los pasos siguientes.

Checklist de validación:

- Ver que la salida incluya columnas como `survived`, `pclass`, `sex`, `age`, `fare`, `deck`, etc.
- Confirmar que la lista es de tipo `list` y su longitud coincide con `df_titanic.shape[1]`.

```
# === Actividad 2.1 – Obtener los nombres de las columnas ===
# Objetivo: listar los nombres de columnas del DataFrame ORIGINAL df_titanic.
# Vamos por dos caminos equivalentes (.columns y .keys),
# validamos tipos/longitud y guardamos la lista en una variable para reutilizar
# en los próximos pasos.

# 1) Obtener lista de columnas (método estándar)
cols_titanic = df_titanic.columns.tolist()

# 2) Alternativa equivalente (algo menos usada)
# .keys() es básicamente un alias de .columns
cols_titanic_keys = df_titanic.keys().tolist()

# 3) Validaciones rápidas
print("Total de columnas según shape[1]:", df_titanic.shape[1])
print("Total de columnas (via .columns):", len(cols_titanic))
print("Total de columnas (via .keys): ", len(cols_titanic_keys))
print("¿Listas equivalentes?:", cols_titanic == cols_titanic_keys, "\n")

# 4) Mostrar los nombres de columnas de forma clara y numerada
print("Listado de columnas (enumerado):")
for i, c in enumerate(cols_titanic, start=1):
    print(f"{i:>2}. {c}")

# 5) (Opcional) Mostrar el tipo de objeto para reforzar la idea
print("\nTipo de 'cols_titanic':", type(cols_titanic))
```

✓ Actividad 2.2 - Eliminar la columna `deck`

Consigna: Quitar la columna `deck` del DataFrame **original** `df_titanic`.

Estrategia de resolución:

1. Confirmar si `deck` existe en `df_titanic.columns`.
2. Usar `df_titanic.drop(columns=['deck'], errors='ignore')` para evitar errores si no estuviera presente.
3. Guardar el resultado en `df_titanic_sin_deck` (mantenemos el original).
4. Validar:
 - Comparar `shape` antes y después (debe disminuir en 1 columna).
 - Verificar que `deck` ya no esté en `df_titanic_sin_deck.columns`.

Nota sobre nulos/advertencias: `deck` tiene muchos `NaN`, pero `drop` elimina la **columna completa**, no fila por fila; por lo tanto no realiza imputaciones ni genera conflictos por los faltantes. Si el entorno lanza advertencias, se deben a configuraciones de `pandas/display`, no al método en sí.

```
# 1) Información previa útil (opcional): cuántos faltantes tiene 'deck'
if 'deck' in df_titanic.columns:
    na_deck = df_titanic['deck'].isna().sum()
    print(f"Faltantes en 'deck' antes de eliminarla: {na_deck} de {len(df_titanic)} filas\n")
else:
    print("La columna 'deck' no existe en df_titanic (nada que eliminar).\n")
```

```
# 2) Guardar shape original para comparar
shape_original = df_titanic.shape
print("Shape original df_titanic:", shape_original)
```

```
# 3) Eliminar la columna (sin tocar el DataFrame original)
df_titanic_sin_deck = df_titanic.drop(columns=['deck'], errors='ignore').copy()

# 4) Validaciones después del drop
```

```
print("Shape resultante df_titanic_sin_deck:", df_titanic_sin_deck.shape)
print("¿Se redujo en 1 columna?", df_titanic_sin_deck.shape[1] == df_titanic.shape[1] - (1 if 'deck' in df_titanic
```

```
# 5) Confirmar que 'deck' ya no esté en las columnas
print("\n' deck ' en columnas resultantes?:", 'deck' in df_titanic_sin_deck.columns)
```

```
# 6) Mostrar las primeras columnas para inspección rápida
print("\nPrimeras columnas de df_titanic_sin_deck:")
print(df_titanic_sin_deck.columns.tolist()[:10])
```

✓ Actividad 2.3 - Reindexar el DataFrame

Consigna: Reindexar el DataFrame **después de eliminar** `deck`.

Estrategia de resolución:

1. Partimos de `df_titanic_sin_deck` (resultado del paso que hicimos recién).
2. **Reiniciamos el índice de filas** para que quede consecutivo desde 0 hasta `n-1` con `reset_index(drop=True)`.

Es buena práctica estandarizar el índice tras transformaciones.

3. Validamos que:
 - El índice sea un `RangeIndex` que empieza en 0 y termina en `len(df)-1`.
 - No haya duplicados de índice y su longitud coincida con el número de filas.

Checklist de validación:

- `type(df_titanic_reindexed.index) == RangeIndex`.
- `df_titanic_reindexed.index.min() == 0` y `df_titanic_reindexed.index.max() == len(df_titanic_reindexed)-1`.
- `df_titanic_reindexed.index.is_unique == True`.

```
# 1) Guardamos referencia del índice original para comparar (opcional)
idx_original = df_titanic_sin_deck.index
print("Tipo índice ORIGINAL:", type(idx_original))
print("Primeros 5 índices ORIGINALES:", idx_original[:5].tolist(), "\n")
```

El bloque anterior guarda el índice (las etiquetas de las filas) del DataFrame. Ese objeto es un `Index/RangeIndex` de pandas: una vista inmutable de los rótulos de fila.

```
# 2) Resetear el índice -> drop=True evita que el índice viejo pase a ser columna
df_titanic_reindexed = df_titanic_sin_deck.reset_index(drop=True).copy()

# 3) Validaciones del nuevo índice
idx_nuevo = df_titanic_reindexed.index
print("Tipo de índice NUEVO:", type(idx_nuevo))
print("RangeIndex esperado (0..n-1):")
print("  - Mínimo:", idx_nuevo.min())
print("  - Máximo:", idx_nuevo.max())
print("  - Largo :", len(idx_nuevo), "\n")

print("¿Índice único?:", df_titanic_reindexed.index.is_unique)
print("¿Coincide largo de índice con filas?:", len(idx_nuevo) == len(df_titanic_reindexed), "\n")
```

Esta operación reinicia el índice para que sea consecutivo 0..n-1 (`reset_index(drop=True)` descarta el índice viejo y evita que quede como columna) y crea una copia independiente del resultado con `.copy()`

```
# 4) Vistazo rápido para confirmar que NO apareció una columna 'index' accidental
print("Primeras columnas del DataFrame reindexado:")
print(df_titanic_reindexed.columns.tolist()[:10])
```

```
# 5) Muestra de las primeras filas (solo índice y 3-4 columnas) para inspección
cols_vista = ['survived', 'pclass', 'sex', 'age']
print("\nVista rápida (índice + columnas clave):")
print(df_titanic_reindexed.loc[:4, cols_vista].to_string())
```

✓ Cierre de la Actividad 2

Lo que hicimos:

- Listamos los **nombres de columnas** de `df_titanic`.
- Eliminamos la columna `deck` creando `df_titanic_sin_deck` (de 15 a 14 columnas).
- **Reindexamos** el resultado con `reset_index(drop=True)`. Tenemos un índice limpio `RangeIndex(0..n-1)` y sin columna `index` accidental.

Estado final de trabajo:

- DataFrame vigente para seguir: `df_titanic_reindexed`
- Columnas (principales): `survived, pclass, sex, age, sibsp, parch, fare, embarked, class, who, adult_male, embark_town, alive, alone` (sin `deck`).

Comienza a programar o [generar](#) con IA.

NOTAS Y ANEXOS FINALES:

Seaborn:

Seaborn es una librería de **visualización estadística** para Python, construida sobre Matplotlib y muy integrada con pandas. Te da una interfaz de alto nivel para hacer gráficos “listos para publicar” (mejores estilos por defecto, paletas, y funciones orientadas a análisis: boxplot, violinplot, catplot, pairplot, heatmaps, etc.).

¿Por qué trae el Titanic “adentro”? Porque Seaborn incluye una **colección de datasets de ejemplo** (iris, tips, penguins, flights, titanic, ...) pensados para:

- **Demostrar** cómo usar sus funciones en la documentación y tutoriales.
- **Practicar** rápidamente sin tener que buscar datos.
- **Estandarizar** ejemplos: todos vemos lo mismo, con los mismos tipos y nombres de columnas.

Técnicamente, cuando hacés `sns.load_dataset("titanic")` Seaborn:

1. **Descarga** ese CSV desde su repositorio público de datasets (si tenés internet) y
2. Lo **cachea** en tu máquina (normalmente en `~/ .cache/seaborn/`), así la próxima carga es local y más rápida. Si estás **offline** y nunca lo bajaste, fallará; si ya quedó en caché, cargará igual.

Tip útil:

- Ver la lista de datasets disponibles: `sns.get_dataset_names()`
 - El Titanic de Seaborn viene **limpio y tipado** (varias columnas ya son `category` o `bool`), ideal para ejemplos rápidos.
-