UT2: Manejo de datos con Pandas

Contenidos

1. Introducción

pandas es la librería estándar para manipulación de datos en Python.
Permite leer, limpiar, transformar y exportar datos fácilmente, y será muy útil para preparar la información que serviremos desde nuestras APIs.

Estructuras principales:

- **Series** → estructura 1D (como una lista con etiquetas).
- **DataFrame** → estructura 2D (tabla con filas y columnas).

🔧 **Instalación**

⚙️ *Nota:* Si usas **Google Colab**, **Jupyter Notebook** o **Anaconda**, pandas ya está instalado.
Si trabajas desde **VS Code**, la terminal o un entorno virtual, instálalo con:

```
pip install pandas
```

---

2. Jupyter Notebook

Jupyter Notebook: entorno interactivo para trabajar con datos

Jupyter Notebook es una herramienta que permite escribir y ejecutar código Python de forma interactiva, mezclando texto, código y resultados en un mismo documento.
Es el entorno más usado en ciencia de datos y análisis porque permite ir **probando paso a paso** el código y **ver directamente las salidas de pandas** en forma de tabla.

🔷 Ventajas principales

- Ejecución **celda a celda**, ideal para pruebas incrementales.
- Permite **documentar con texto y Markdown** el análisis.
- Muestra los resultados de **gráficos, tablas y outputs** directamente bajo cada celda.
- Perfecto para **experimentar, limpiar y explorar datos** antes de usarlos en una aplicación o API.

🔷 Estructura básica

Cada notebook se compone de **celdas**:

- Celdas de **código** (donde se escribe Python).
- Celdas de **texto Markdown** (para explicaciones, títulos, listas, etc.).

Ejemplo:

```
import pandas as pd

df = pd.DataFrame({
    "nombre": ["Ana", "Luis", "Marta"],
    "edad": [25, 30, 22]
})
df
```

💡 En un notebook, al ejecutar esa celda, la tabla DataFrame se muestra de forma visual y formateada (más agradable que en la consola).

🔷 Cómo instalar y ejecutar Jupyter Notebook

Si usas **Anaconda**, ya viene instalado.
Si no, puedes instalarlo manualmente con:

```
pip install notebook
```

Luego se lanza desde la terminal con:

```
jupyter notebook
```

Esto abrirá una pestaña en el navegador, normalmente en la dirección `http://localhost:8888`, desde donde puedes crear nuevos `notebooks.ipynb`

🔷 Alternativas modernas

- **JupyterLab** → versión más avanzada con pestañas, paneles, explorador de archivos, etc.
- **VS Code** → permite abrir y ejecutar notebooks directamente sin salir del editor.
- **Google Colab** → entorno online gratuito (no requiere instalación).

🔷 Usar Jupyter online (Google Colab)

Google Colab es una opción ideal para los alumnos que no quieren configurar nada localmente. Solo hace falta tener una cuenta de Google y acceder a:

👉 https://colab.research.google.com

Ventajas:

- No requiere instalación.
- Permite guardar los notebooks en **Google Drive**.
- Ya trae **pandas**, **NumPy** y **matplotlib** instalados.
- Se pueden importar datasets directamente desde URLs o desde el Drive.

Ejemplo de carga de CSV desde URL:

```
import pandas as pd

url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"
df = pd.read_csv(url)
df.head()
```

🔷 Guardar y compartir notebooks

Los notebooks se guardan con extensión **.ipynb**.
Pueden compartirse fácilmente:

- Por correo o Drive (como cualquier archivo).
- Exportándose a otros formatos: **HTML**, **PDF**, **Markdown**, etc.

```
jupyter nbconvert --to html mi_notebook.ipynb
```

⚙️ En resumen:

Usa **Jupyter Notebook o Colab** para explorar y limpiar datos.
Más adelante, cuando construyas las APIs, trabajarás con los mismos scripts .py pero ya listos para producción.

---

3. Crear y cargar datos

```
import pandas as pd

# Crear un DataFrame desde un diccionario
data = {
    "nombre": ["Ana", "Luis", "Marta"],
    "edad": [25, 30, 22],
    "ciudad": ["Madrid", "Sevilla", "Valencia"]
}
df = pd.DataFrame(data)
```

Cargar datos desde distintos formatos

```
# Desde un CSV
df_csv = pd.read_csv("personas.csv")

# Desde un Excel
df_excel = pd.read_excel("personas.xlsx", sheet_name="Hoja1")

# Desde un JSON
df_json = pd.read_json("personas.json")
```

---

4. Selección, filtrado y ordenación.

```
# Seleccionar una columna
nombres = df["nombre"]

# Filtrar por condición
mayores_25 = df[df["edad"] > 25]

# Seleccionar varias columnas
sub_df = df[["nombre", "ciudad"]]

# Seleccionar una fila por índice
fila_0 = df.loc[0]
```

Agrupaciones y ordenación

```
# Media de edad por ciudad
print(df.groupby("ciudad")["edad"].mean())

# Ordenar por edad descendente
df_ordenado = df.sort_values(by="edad", ascending=False)
```

5. Exploración y detección de valores anómalos

Exploración rápida

```
print(df.head())       # primeras filas
print(df.shape)        # dimensiones
print(df.columns)      # nombres de columnas
print(df.info())       # información general
print(df.describe())   # estadísticas básicas
```

Detección de valores anómalos

Antes de limpiar los datos, es recomendable **explorar su contenido** para detectar errores frecuentes:

- Valores duplicados o vacíos.
- Diferencias de formato (por ejemplo, mayúsculas/minúsculas o espacios).
- Abreviaturas o nombres inconsistentes.
- Outliers (valores numéricos fuera de rango).

Una función muy útil para inspeccionar el contenido de cada columna es **.unique():**

```
import pandas as pd

df = pd.read_csv("datos.csv")

# Mostrar los valores únicos de las columnas categóricas

print(df["ciudad"].unique())

print(df["nombre"].unique())
```

También se pueden obtener estadísticas rápidas con:

```
df.describe(include="all")
```

Esto muestra recuentos, medias, mínimos, máximos y frecuencias más comunes.
A partir de aquí, puedes decidir qué reglas aplicar en tu limpieza (por ejemplo, reemplazar abreviaturas o eliminar valores anómalos).

---

6. Limpieza de datos

```
# Comprobar valores nulos
df.isna().sum()

# Eliminar filas con valores nulos
df = df.dropna()

# Rellenar nulos con la media de la columna
df["edad"] = df["edad"].fillna(df["edad"].mean())

# Eliminar duplicados
df = df.drop_duplicates()

# Normalizar texto
df["nombre"] = df["nombre"].str.strip().str.title()

# Conversión de tipos
df["edad"] = pd.to_numeric(df["edad"], errors="coerce")
df["fecha"] = pd.to_datetime(df.get("fecha"), errors="coerce")

# Reemplazar valores
df["ciudad"] = df["ciudad"].replace({"Mad": "Madrid", "Sev": "Sevilla"})
```

Uso de expresiones regulares en la limpieza de datos

Las **expresiones regulares (regex)** son una herramienta muy potente para detectar o corregir patrones en texto.
En pandas se pueden usar directamente con métodos como .**str.contains()** o **.str.replace().**

Por ejemplo, para eliminar caracteres no alfabéticos de los nombres:

```
df["nombre"] = df["nombre"].str.replace(r"[^a-zA-ZáéíóúÁÉÍÓÚñÑ\s]", "", regex=True)
```

Para validar emails correctamente formados:

```
df = df[df["email"].str.contains(r"^[\w\.-]+@[\w\.-]+\.\w+$", na=False)]
```

O para detectar filas donde el número de teléfono no tenga 9 dígitos:

```
telefonos_erroneos = df[~df["telefono"].str.contains(r"^\d{9}$", na=False)]

print(telefonos_erroneos)
```

Las **regex** permiten automatizar muchas tareas de limpieza textual:

- Normalización de formatos (fechas, códigos postales, teléfonos).
- Detección de errores de escritura.
- Filtrado o eliminación de datos inválidos.

💡 Consejo: puedes probar tus expresiones regulares en sitios web como regex101.com antes de aplicarlas en tu código.

---

7. Guardar y exportar datos (CSV, Excel, JSON)

Una vez limpios los datos, es habitual guardarlos para su reutilización o para servirlos desde una API.

```
# Guardar como CSV
df.to_csv("personas_limpias.csv", index=False)

# Guardar como Excel
df.to_excel("personas_limpias.xlsx", index=False, sheet_name="Datos")

# Guardar como JSON
df.to_json("personas_limpias.json", orient="records", indent=4, force_ascii=False)
```

👉 En general:

- **CSV** → formato más universal y ligero.
- **Excel** → práctico para informes o colaboración.
- **JSON** → ideal para APIs.

---

8. Preparar datos para API

Las APIs suelen devolver datos en formato **JSON**. Podemos convertir fácilmente un DataFrame en JSON o en una lista de diccionarios.

```
# DataFrame → lista de diccionarios
data_api = df.to_dict(orient="records")

# DataFrame → cadena JSON
json_str = df.to_json(orient="records", indent=4, force_ascii=False)
```

Ejemplo práctico (usando FastAPI):

```
from fastapi import FastAPI
from fastapi.responses import JSONResponse
import pandas as pd

app = FastAPI()

@app.get("/personas")
def obtener_personas():
    df = pd.read_csv("personas_limpias.csv")
    data = df.to_dict(orient="records")
    return JSONResponse(content=data)
```

---

9. Pasos resumen del flujo para APIs

1. **Cargar datos** desde CSV, Excel o JSON.
2. **Explorar y limpiar**: eliminar duplicados, manejar nulos, normalizar texto y tipos.
3. **Filtrar o transformar** según las necesidades del endpoint.
4. **Exportar o convertir** a JSON para devolver desde la API.