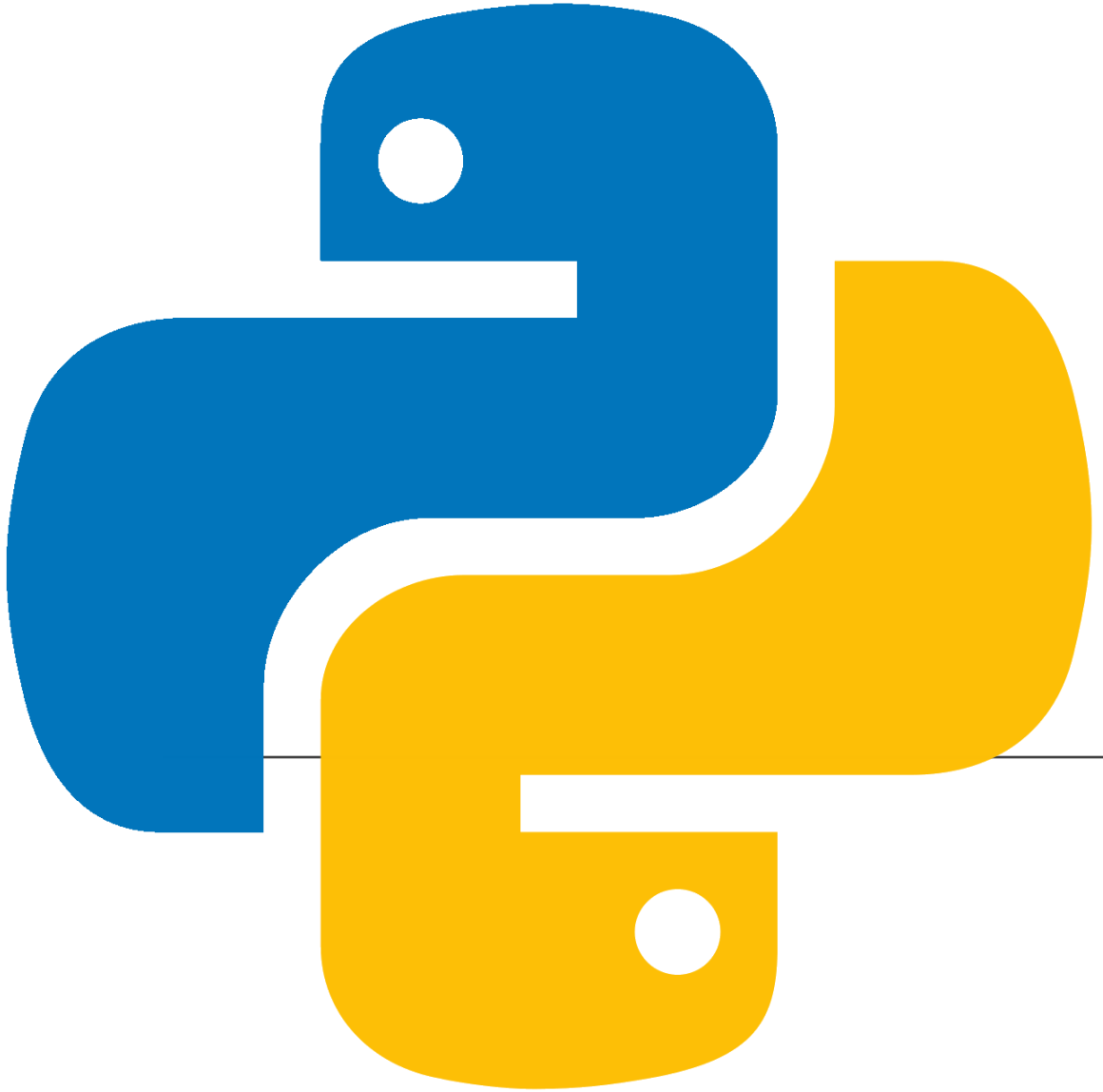


Python: Análisis de Datos y Ecosistemas Modernos de Analítica

ANÁLISIS DE DATOS

Recordando: Testing

El **testing** es esencial para asegurar que el código cumpla con los requisitos y evitar errores costosos.

Recordando: Ciclo de vida

El **análisis de datos** es el proceso de obtener **nueva información** a partir de métodos analíticos.

Estadística básica → promedios, medianas.

Modelos avanzados → regresiones, inferencias, predicciones.

Análisis visual → responder preguntas explorando gráficas o tablas de datos.

Fuentes de datos

Servicios web

- Forma de usar servidores es para **exponer servicios web**.
- Permiten acceder a funciones de software a través de la red.
- Utilizan los mismos **protocolos de la web** que se usan para acceder a páginas.

Servicios web

Un **servicio web** es un software accesible a través de la red que utiliza formatos estandarizados de transmisión de mensajes (como **JSON** o **XML**) para ejecutar funciones de manera remota.

Características:

- Se accede mediante una **red**.
- Expone una **interfaz bien definida** de funciones disponibles.
- Es **independiente del lenguaje** de programación usado en el cliente o en el servidor.

Interfaces de programación de aplicaciones (API)

Una **API** es la especificación que define cómo los componentes de software deben interactuar.

Permite que aplicaciones intercambien mensajes sin necesidad de conocer la implementación interna.

En un **servicio web**, es fundamental definir una API clara para acceder a sus funcionalidades.

Servicios REST

Usan el protocolo **HTTP** como medio de comunicación entre clientes y servidores.

Son **sin estado**: cada operación es independiente y debe incluir toda la información necesaria.

La información se transmite y recibe en formatos estandarizados como **JSON** o **XML**.

https://aws.amazon.com/es/what-is/restful-api/

¿Protocolos? ¿HTTP?

REST se basa en el uso del protocolo **HTTP** como medio de comunicación entre procesos.

Un HTTP, **un protocolo de red,** no es más que un lenguaje común y estandarizado para que distintos procesos puedan comunicarse.

Sus verbos principales (**GET, POST, PUT, DELETE**) permiten interactuar con los servicios REST.

REST

REST se basa en **recursos identificados por URLs** y en **acciones a través de métodos HTTP.**

En el análisis de datos es común **consumir servicios externos**

- **Conocer las limitaciones**
- **Comprender su interfaz**
- **Gestionar el acceso**

Método	Endpoint	Descripción
GET	`/ventas`	Todas las ventas
GET	`/ventas/{id}`	Detalle de una venta
GET	`/ventas/resumen`	Agregadas

Alternativas

SOAP es una alternativa tradicional a REST para construir servicios web. Define su propio protocolo, más complejo, y no depende únicamente de HTTP.

Al no ser nativo en muchos entornos web, ofrece **menos interoperabilidad**: por ejemplo, no es sencillo consumir un servicio SOAP desde un navegador.

Existen también opciones más modernas como **GraphQL**, que permiten consultas más flexibles y específicas sobre los datos.

Servicios en Python

A pesar de existir muchas liberías útiles, **requests** es una de las más sencillas y mejor documentadas.

Permite acceder a puntos HTTP, utilizar proxies, procesar códigos de respuesta, y mucho más.

```
py -m pip install requests
```

Usando requests

Podemos fácilmente utilizar los verbos HTTP mediante las funciones expuestas por la libería

```
import requests
respuesta = requests.get('https://xkcd.com/1906/')
codigo = respuesta.status_code
print(codigo)
contenido = respuesta.text
```

```
Out[17]: 200
```

Mediante el parámetro **text** podemos obtener el cuerpo de la respuesta, por ejemplo si esta es una página HTML, podemos posteriormente procesarla con ayuda de otras liberías.

Usando requests: descargas

Es muy fácil guardar el resultado de una solicitud, en caso de que por ejemplo sea un archivo.

```
import requests
resultado = requests.get("https://wwwnc.cdc.gov/travel/images/map-ecuador.png")
with open("alguna/ubicacion", 'wb') as f:
    f.write(resultado.content)
```

Recordemos que *open* nos permite trabajar con archivos de manera muy sencilla.

Usando requests: parámetros

Dentro del protocolo HTTP, las operaciones permiten proveer diferentes parámetros en la solicitud.

```
import requests
argumentos = {'argumento1':1,'argumento2':2}
r = requests.get('https://httpbin.org/get’, params=argumentos)
print(r.text)
```

En el caso de una solicitud GET, esto simplemente agregará los mismos a la URL

```
https://httpbin.org/get?argumento1=1&argumento2=2
```

Usando requests: POST

Como aprendimos anteriormente, existen diferentes verbos que podemos usar en HTTP, y cuando un servicio web soporta puntos de acceso POST, PUT o DELETE, los podemos acceder de manera similar.

```
import requests
argumentos = {'usuario':1,'clave':2}
r = requests.post('https://httpbin.org/post',params=argumentos)
```

A diferencia de GET, los parámetros serán enviados en el cuerpo de la solicitud, no en la URL.

Usando requests: JSON

Es posible que un servicio web responda directamente con información estructurada como objetos JSON y no texto plano. La siguiente función crea un diccionario a partir de ese JSON.

```
import requests
argumentos = {’username’: ’olivia’, ’password’: 123':}
respuesta = requests.post('https://httpbin.org/post',params=argumentos)

print(respuesta.json())
```

```
{'args': {}, 'data': '', 'files': {}, 'form': {'password': '123', 'username': 'olivia'}, 'headers': {'Accept': '*/*', 'Accept-E
ncoding': 'gzip, deflate', 'Content-Length': '28', 'Content-Type': 'application/x-www-form-urlencoded', 'Host': 'httpbin.org',
'User-Agent': 'python-requests/2.18.4'}, 'json': None, 'origin': '103.10.31.17, 103.10.31.17', 'url': 'https://httpbin.org/pos
t'}
```

Encuesta

¿Qué es REST?

Encuesta

¿Qué es REST?

Un estilo de arquitectura para servicios web basado

Fuentes libres

Existen diferentes **servicios libres** que facilitan el acceso a datasets:

Kaggle Datasets – múltiples temas (CSV, Parquet).

Google Dataset Search – buscador global de datasets.

Data.gov / Eurostat – datos abiertos (economía, sociedad).

Yahoo Finance / World Bank / IMF – finanzas y economía.

UCI ML Repo / NASA / NOAA – ciencia, clima, espacio.

CDC / WHO / Harvard Dataverse – salud y ciencias sociales.

NYC Taxi Data / AWS Open Data / MS Research – Big Data en CSV y Parquet.

ANALIZANDO EN PYTHON

Operaciones con Python

Lo que ya sabemos:

- Cómo trabajar con **archivos CSV y Parquet**.
- Concepto de **análisis de datos** en términos generales.
- Introducción a la conexión con **bases de datos**.

Ya sabemos dónde están los datos y cómo utilizarlos.

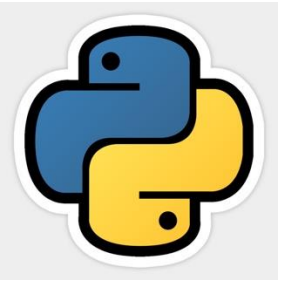
El Problema de la Eficiencia

¿Qué pasa si intentamos procesar millones de filas solo con Python nativo (listas, loops)?

- Procesos se vuelven **lentos**.
- El consumo de **memoria** escala muy mal.
- Se necesita mucho **código repetitivo** para operaciones simples.

Python no está optimizado para cálculos de gran volumen de datos por defecto.

numpy



Librería fundamental para **cálculo científico y numérico** en Python. Introduce la estructura de datos **ndarray** (arrays multidimensionales).

- Ofrece operaciones **vectorizadas**, eliminando la necesidad de bucles explícitos.
- Constituye la base sobre la que se construyen librerías como **pandas, scikit-learn y TensorFlow.**

numpy

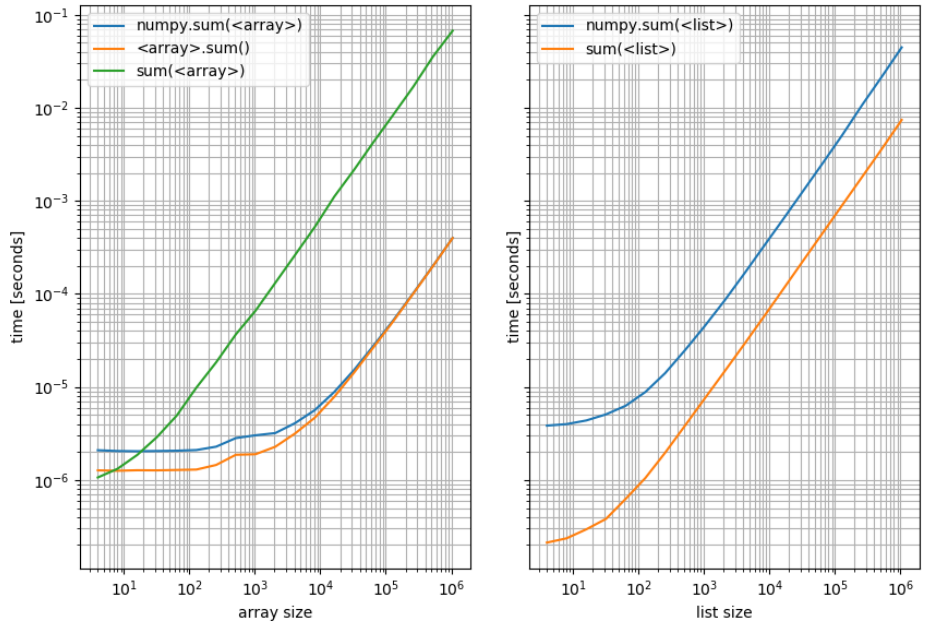
Gran conjunto de funciones matemáticas optimizadas.

- Velocidad: cálculos implementados en C, mucho más rápidos que los bucles en Python.
- Eficiencia en memoria: arrays más compactos que listas.
- Vectorización: aplicar operaciones a todos los elementos en una sola instrucción.

numpy

NumPy ejecuta operaciones de manera **más rápida y con menor uso de memoria**

```
import numpy as np

# Con listas
lista = [i for i in range(1_000_000)]
suma_lista = sum(lista)

# Con NumPy
arr = np.arange(1_000_000)
suma_numpy = arr.sum()
```



numpy

NumPy permite leer datos tabulares directamente desde archivos CSV.

```
import numpy as np

# Cargar archivo CSV (valores separados por coma)
data = np.loadtxt("datos.csv", delimiter=",")
print(data.shape)   # dimensiones del arreglo
print(data[:5])
```

NumPy trabaja mejor con **datos numéricos homogéneos**.
Para datos tabulares más complejos (mixtos: texto + números) existe **pandas**.

pandas

- Librería construida utilizando **NumPy**, pensada para análisis tabular.
- Introduce la estructura de datos **DataFrame** (similar a una tabla de base de datos).
- Facilita la carga, exploración, transformación y exportación de datos.
- Estándar de facto para análisis en Python

pandas

Un **DataFrame** es una estructura tabular con filas y columnas, similar a una hoja de cálculo o una tabla de base de datos, que permite analizar y manipular datos de forma eficiente.

Una **Series** es una estructura unidimensional, conceptualmente equivalente a una **columna de un DataFrame**, pero que puede existir de manera independiente.

Son **conceptos fundamentales** que se repiten en distintas librerías del ecosistema de datos.

Series

Una **Series** en pandas es una estructura de datos unidimensional que representa una **columna** con un valor asociado a cada fila, compuesta por dos partes:

- un **array de valores** (numéricos, texto, fechas, etc.)
- y un **índice** que etiqueta cada elemento.

Series

Es el bloque básico con el que se construyen los DataFrames.

```python
import pandas as pd

s = pd.Series([10, 20, 30, 40], index=["a", "b", "c", "d"])
print(s)
```

→

```
a    10
b    20
c    30
d    40
dtype: int64
```

DataFrame

Un **DataFrame** es una estructura de datos tabular con filas y columnas, similar a una hoja de cálculo o tabla de base de datos, que permite analizar y manipular información de forma eficiente.

- Estructura de datos **tabular**: filas (observaciones) y columnas (atributos).
- Similar a una **tabla de base de datos** o una **hoja de cálculo**.
- Cada columna puede tener un **tipo de dato distinto** (numérico, texto, fechas).
- Incluye un **índice** para identificar y acceder a las filas de forma eficiente.

DataFrame

Es la estructura **más usada** para análisis de datos en pandas.

Un **DataFrame** es un conjunto de Series organizadas en filas y columnas.

```
data = {
    "producto": ["A", "B", "C"],
    "precio": [100, 200, 300],
    "stock": [50, 20, 15]
}

df = pd.DataFrame(data)
print(df)
```

→

	producto	precio	stock
0	A	100	50
1	B	200	20
2	C	300	15

pandas

pandas facilita la **exploración de datos** con operaciones muy expresivas:

Selección de columnas, Filtrado de filas, Estadísticas rápidas (media, suma, conteo).

```python
# Selección de columna
print(df["precio"])

# Filtrado por condición
print(df[df["stock"] < 30])

# Promedio de precios
print(df["precio"].mean())
```

→

```
0    100
1    200
2    300
Name: precio, dtype: int64

  producto  precio  stock
1        B     200     20
2        C     300     15

200.0
```

pandas

En pandas podemos acceder a los datos de distintas maneras:

- loc selecciona por **etiqueta** (nombre de fila o columna).
- iloc selecciona por **posición** (índices numéricos).

```python
import pandas as pd

df = pd.DataFrame({
    "producto": ["A", "B", "C"],
    "precio": [100, 200, 300]
}, index=["x1", "x2", "x3"])

print(df.loc["x2"])   # por etiqueta
print(df.iloc[1])     # por posición
```

```python
# Selección por etiqueta
print(df.loc[0, "precio"])

# Selección por posición
print(df.iloc[1, 2])
```

Pandas: ¿Qué es el índice?

Cada **fila de un DataFrame o Series** está identificada por un **índice**.

El índice puede ser:

- Numérico (0, 1, 2, … por defecto).
- Personalizado (ej. códigos, fechas, etiquetas).

El índice permite **acceso eficiente**, **filtrado** y **alineación automática** de datos, **no es solo un número de fila**, es parte de la estructura.

Pandas: ¿Qué es el índice?

El índice también permite **selección por rango.**

Con loc ⟶ los **rangos son inclusivos.**
Con iloc ⟶ los **rangos son como en Python, excluyen el límite superior.**

```
# Rango por etiquetas (inclusivo)
print(df.loc["x1":"x2"])

# Rango por posición (exclusivo en el tope)
print(df.iloc[0:2])
```

Pandas: ¿Qué es el índice?

El índice puede tener varias propiedades:

- **Único o duplicado** (aunque lo recomendable es que sea único).
- **Ordenado o no ordenado** (afecta búsquedas y cortes por rangos).
- **Jerárquico (MultiIndex)** → útil para datos multidimensionales.

Pandas: MultiIndex

Un **MultiIndex** es un índice jerárquico: cada fila está identificada por **más de una clave.**

```
import pandas as pd

index = pd.MultiIndex.from_tuples([
    ("España", 2022),
    ("España", 2023),
    ("México", 2022),
    ("México", 2023)
], names=["pais", "anio"])

df = pd.DataFrame({"ventas": [100, 150, 200, 250]}, index=index)
print(df)
```

→

		ventas
pais	anio	
España	2022	100
	2023	150
México	2022	200
	2023	250

Pandas: MultiIndex

Facilita operaciones de **agregación y selección en varios niveles.**

Optimiza para que podemos filtrar por país, por año o por la combinación de ambos, lo que hace a pandas muy flexible para datos complejos.

Pandas: filtrado

El filtrado se hace aplicando expresiones lógicas sobre columnas, lo que devuelve subconjuntos de datos.

El índice en pandas: funciona como un **identificador único de las filas** y es la pieza clave para operaciones de filtrado, asi como groupby o merge.

Pandas: filtrado

El filtrado devuelve subconjuntos de filas cuyo índice se **mantiene**: pandas conserva la relación entre condición y filas.

Podemos filtrar filas de acuerdo a los valores de las columnas.

```
print(df[df["precio"] > 150])
```

	producto	precio
x2	B	200
x3	C	300

Pandas: Booleanos y filtrado

En pandas, las comparaciones sobre columnas generan una **Serie booleana** (valores True o False) con el mismo índice. El resultado es una **Serie** alineada al índice del DataFrame.

Esta Serie se puede usar para filtrar filas en un DataFrame.

```python
import pandas as pd

df = pd.DataFrame({
    "producto": ["A", "B", "C", "D"],
    "precio": [100, 200, 300, 150],
    "stock": [50, 20, 15, 0]
})

# Comparación booleana
print(df["precio"] > 150)
```

→

```
0    False
1     True
2     True
3    False
Name: precio, dtype: bool
```

Pandas: Booleanos y filtrado

Cuando aplicamos una Serie booleana al DataFrame, solo se devuelven las filas donde la condición es True.

El **índice se conserva,** lo que permite encadenar operaciones posteriores. Significa que el DataFrame resultante **no reinicia numeracióniinterna automáticamente**.

```
  producto  precio  stock
1        B     200     20
2        C     300     15
```

```python
# Pero si usamos loc con índice 0
print(filtro.loc[0])   # ERROR: no existe índice 0 en 'filtro'
```

Pandas: reset_index()

Cuando aplicamos una Serie booleana al DataFrame, solo se devuelven las filas donde la condición es True.

El **índice se conserva,** lo que permite encadenar operaciones posteriores. Significa que el DataFrame resultante **no reinicia numeracióniinterna automáticamente**.

Pandas: El índice

Conservar el índice original → útil cuando el índice tiene **significado** (ej. IDs, fechas).

Reiniciar el índice si queremos trabajar con subconjuntos como si fueran tablas “nuevas”.

```
filtro_reset = filtro.reset_index(drop=True)
print(filtro_reset)
```

	producto	precio	stock
1	B	200	20
2	C	300	15

→

	producto	precio	stock
0	B	200	20
1	C	300	15

Pandas: El índice

El **índice actúa como la “identidad” de cada fila** en pandas.

Filtrar por defecto **no cambia esa identidad**, solo oculta las filas que no cumplen la condición.

Pandas: El índice

El **índice actúa como la “identidad” de cada fila** en pandas.

Filtrar por defecto **no cambia esa identidad**, solo oculta las filas que no cumplen la condición.

Encuesta

Después de aplicar un filtro, ¿qué ocurre con el **índice** de un DataFrame?

Encuesta

Después de aplicar un filtro, ¿qué ocurre con el **índice** de un DataFrame?

Se **mantienen las etiquetas originales del índice**,

Pandas: Agrupaciones

groupby permite **agrupar filas** por los valores de una o más columnas.
Sobre cada grupo se aplican funciones de **agregación** (sumas, promedios, conteos, etc.).

El resultado utiliza la(s) columna(s) agrupada(s) como **nuevo índice**.

```python
import pandas as pd

df = pd.DataFrame({
    "categoria": ["A", "A", "B", "B", "B"],
    "ventas": [100, 200, 300, 150, 250]
})

print(df.groupby("categoria")["ventas"].sum())
```

→

```
categoria
A    300
B    700
Name: ventas, dtype: int64
```

Pandas: Agrupaciones

Podemos aplicar varias funciones a la vez con .agg().

```
print(df.groupby("categoria")["ventas"].agg(["count", "mean", "max"]))
```

categoria	count	mean	max
A	2	150	200
B	3	233	300

El MultiIndex organiza los resultados de cada grupo.

Pandas: Agrupaciones

Siempre al agrupar por más de una columna, se genera un **MultiIndex**.

```
df2 = pd.DataFrame({
    "categoria": ["A", "A", "B", "B", "B"],
    "region": ["Norte", "Sur", "Norte", "Norte", "Sur"],
    "ventas": [100, 200, 300, 150, 250]
})

print(df2.groupby(["categoria", "region"])["ventas"].sum())
```

```
categoria  region
A          Norte    100
           Sur      200
B          Norte    450
           Sur      250
Name: ventas, dtype: int64
```

Un **índice jerárquico** con las dos columnas agrupadas

Pandas: Agrupaciones

El filtrado se puede hacer antes o después del groupby.

```
# Filtrar ventas > 150 antes de agrupar
resultado = df2[df2["ventas"] > 150].groupby("categoria")["ventas"].mean()
print(resultado)
```

→

```
categoria
A    200.0
B    275.0
Name: ventas, dtype: float64
```

El **índice** conecta filtrado y agrupación: se conserva la categoría como referencia.

Pandas: Agrupaciones

A veces conviene volver la columna agrupada a datos planos en vez de índice:

```
resultado = df.groupby("categoria")["ventas"].sum().reset_index()
print(resultado)
```

→

	categoria	ventas
0	A	300
1	B	700

reset_index() es útil cuando se necesita un DataFrame “plano”

Pandas: Agrupaciones

Funciona de forma **similar a GROUP BY en SQL**, pero con la **flexibilidad de Python.**

El índice de salida refleja las columnas agrupadas.

Con **MultiIndex** se pueden representar dimensiones múltiples.

Pandas: Merge

¿Qué es un Merge en pandas?

Permite **combinar DataFrames** al estilo SQL (JOIN).

Se basa en **columnas clave** o en el **índice**.

Pandas: Merge

Devuelve un nuevo DataFrame, en el que se conservan únicamente las filas que cumplen las reglas definidas por el tipo de combinación (inner, left, right, outer).

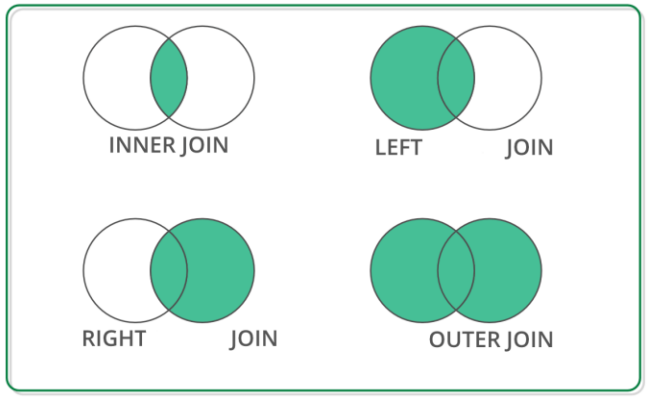
```
import pandas as pd

productos = pd.DataFrame({
    "id": [1, 2, 3],
    "nombre": ["X", "Y", "Z"]
})

ventas = pd.DataFrame({
    "id": [1, 2, 2, 4],
    "monto": [100, 200, 150, 300]
})

resultado = pd.merge(ventas, productos, on="id", how="inner")
print(resultado)
```

→

	id	monto	nombre
0	1	100	X
1	2	200	Y
2	2	150	Y

Pandas: Tipos de Merge

Devuelve un nuevo DataFrame, en el que se conservan únicamente las filas que cumplen las reglas definidas por el tipo de combinación (inner, left, right, outer).



Con outer, pandas conserva **todos los registros** de ambos DataFrames, completando con NaN cuando no hay correspondencia.

Pandas: El índice

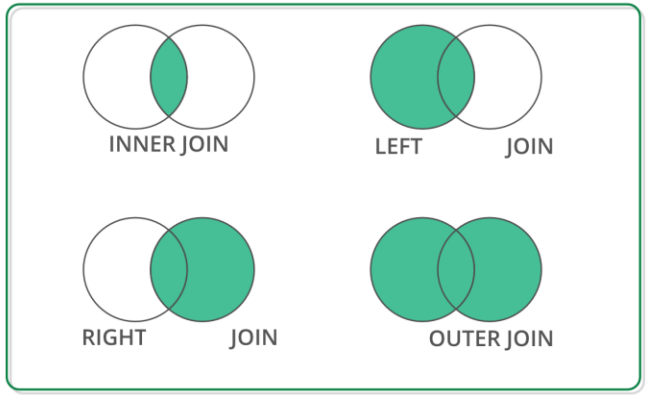
Podemos combinar DataFrames directamente por el índice.

```
productos = productos.set_index("id")

resultado = ventas.merge(productos, left_on="id", right_index=True)
print(resultado)
```

El índice en pandas actúa como clave primaria, muy parecido a SQL.

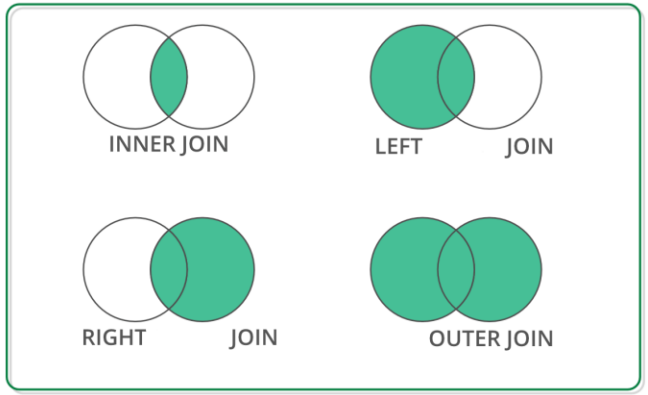
Encuesta

Si hacemos un merge entre dos DataFrames con how=”left", ¿qué filas se conservan en el resultado?



Encuesta

Si hacemos un merge entre dos DataFrames con how=”left", ¿qué filas se conservan en el resultado?



Todas las filas del DataFrame izquierdo, junto a las coincidencias encontradas en el derecho.

Pandas: Valores Nulos

Los valores nulos indican datos faltantes o no disponibles.

Representados como NaN (**Not a Number**) o None.

pandas tiene funciones específicas para **detectar, eliminar o reemplazar** estos valores.

Pandas: Valores Nulos

```python
import pandas as pd

df = pd.DataFrame({
    "producto": ["A", "B", "C"],
    "precio": [100, None, 300]
})

print(df.isnull())      # Detección booleana
print(df.notnull())     # Inverso
```

→

```
   producto  precio
0     False   False
1     False    True
2     False   False

   producto  precio
0      True    True
1      True   False
2      True    True
```

`isnull()` devuelve un DataFrame booleano alineado al índice.

Pandas: Valores Nulos

`dropna()` elimina filas con valores faltantes.

Útil para datos limpios rápidos, pero puede llevar a pérdida de información.

```
print(df.dropna())
```

`fillna()` permite definir reglas de imputación, desde un valor constante hasta funciones estadísticas.

```
print(df.fillna(0))        # Rellenar con un valor fijo
print(df.fillna(df.mean())) # Rellenar con promedio (solo numéricas)
```

Pandas: Valores Nulos

Los valores nulos se **ignoran automáticamente** en operaciones como mean, sum, etc.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({"valores": [10, np.nan, 20, 30]})
print("Promedio:", df["valores"].mean())
print("Suma:", df["valores"].sum())
```

```
Promedio: 20.0
Suma: 60.0
```

Algunas funciones permiten controlar este comportamiento con el argumento skipna.

```
print("Suma (skipna=False):", df["valores"].sum(skipna=False))
```

Pandas: Funciones propias

Aplicando Funciones con **apply**

- Permite usar funciones personalizadas sobre columnas o filas.
- Más flexible que las operaciones vectorizadas directas, **pero a mayor costo**.

```python
import pandas as pd

df = pd.DataFrame({"producto": ["A", "B", "C"], "precio": [100, 200, 300]})

# Aplicar una función a cada valor
df["precio_con_descuento"] = df["precio"].apply(lambda x: x * 0.9)
print(df)
```

→

	producto	precio	precio_con_descuento
0	A	100	90.0
1	B	200	180.0
2	C	300	270.0

Pandas: Fechas y tiempos

pandas tiene el tipo datetime64[ns] para trabajar con fechas.

- Se crean con pd.to_datetime.
- Ofrece atributos útiles como .dt.year, .dt.month, .dt.weekday.

```
df = pd.DataFrame({"fecha": ["2023-01-01", "2023-03-15", "2023-05-10"]})
df["fecha"] = pd.to_datetime(df["fecha"])

print(df["fecha"].dt.year)
print(df["fecha"].dt.month)
```

Pandas: Encadenado

¿Qué es Method Chaining en pandas?

Es el **encadenamiento de métodos** (.) para aplicar varias transformaciones en secuencia.

- Evita variables intermedias, hace el código más legible (“flujo de datos”).
- Funciona porque la mayoría de los métodos de pandas **devuelven un nuevo DataFrame**.

Pandas: Encadenado

```python
import pandas as pd

df = pd.DataFrame({
    "producto": ["A", "B", "C"],
    "precio": [100, 200, 300],
    "stock": [50, 20, 15]
})

resultado = (
    df[df["stock"] > 10]              # Filtrar
      .assign(total=lambda d: d["precio"] * d["stock"])  # Nueva columna
      .sort_values("total", ascending=False)             # Ordenar
)

print(resultado)
```

Pandas: Encadenado

`.pipe()` permite insertar funciones propias en una cadena.

```python
import pandas as pd

df = pd.DataFrame({
    "producto": ["A", "B", "C"],
    "precio": [100, 200, 300],
    "stock": [50, 20, 15]
})

# Función personalizada
def agregar_descuento(df, porcentaje):
    df = df.copy()
    df["precio_desc"] = df["precio"] * (1 - porcentaje)
    return df

resultado = (
    df.pipe(agregar_descuento, porcentaje=0.1)       # aplicar función custom
      .assign(valor=lambda d: d["precio_desc"] * d["stock"])
      .query("valor > 3000")                         # otra operación en cadena
)

print(resultado)
```

Pandas: Encadenado

Ventajas del Chaining con Funciones

- Mantener el estilo **flujo de datos** incluso con lógica compleja.
- Facilita reutilizar código en pipelines.
- Evita romper la lectura con funciones externas “sueltas”.
- Unidades de testeo.

Pandas: Creando columnas

Podemos crear columnas derivadas a partir de otras, pandas permite cálculos vectorizados directamente entre columnas:

```
df["valor_inventario"] = df["precio"] * df["stock"]
print(df)
```

```
  producto  precio  stock  valor_inventario
0        A     100     50              5000
1        B     200     20              4000
2        C     300     15              4500
```

Pandas: Creando columnas

assign crea columnas sin modificar el DataFrame original (útil en chaining).

Podemos encadenar varias asignaciones con lambdas.

```
nuevo = df.assign(
    precio_con_descuento=lambda d: d["precio"] * 0.9,
    inventario_valor=lambda d: d["precio"] * d["stock"]
)
```

Pandas: CSV

CSV = formato más común, pero pesado y sin tipos estrictos.

pandas facilita cargar y guardar con una línea de código.

```python
import pandas as pd

# Lectura
df = pd.read_csv("ventas.csv")
print(df.head())

# Escritura
df.to_csv("ventas_out.csv", index=False)
```

Pandas: Excel

Soportado a través de librerías externas (openpyxl, xlsxwriter).

Útil para interoperar con entornos no técnicos.

```
# Lectura
df_excel = pd.read_excel("ventas.xlsx")

# Escritura
df.to_excel("ventas_out.xlsx", sheet_name="Hoja1", index=False)
```

Pandas: Parquet

Formato moderno **columnar** (guarda datos por columnas). Muy rápido en lectura/escritura.

Ideal para datasets medianos y grandes. **Big Data y pipelines analíticos**.

```
# Escritura
df.to_parquet("ventas.parquet", engine="pyarrow")

# Lectura
df_parquet = pd.read_parquet("ventas.parquet")
print(df_parquet.head())
```

Pandas: Chunking

pandas puede leer archivos **en partes (chunks)** para no saturar memoria.

```
iterador = pd.read_csv("ventas_grandes.csv", chunksize=1000)

suma_total = 0
for chunk in iterador:
    suma_total += chunk["precio"].sum()

print("Suma total:", suma_total)
```

Pandas

Sirve como **puente entre datos crudos** y el **análisis avanzado / machine learning** en Python.

- **Estándar en Python** para trabajar con datos tabulares.
- Combina **facilidad de uso** con **alto poder analítico**.
- Se integra con las principales librerías del ecosistema: **NumPy, Matplotlib, scikit-learn, DuckDB.**

https://pandas.pydata.org/docs/getting_started/index.html

Encuesta

¿Qué estructura en pandas es equivalente a una columna en Excel?

Encuesta

¿Qué estructura en pandas es equivalente a una columna en Excel?

Series

DuckDB

Motor de base de datos **embebido y analítico**, pensado para trabajar con **datasets grandes**.

Similar a **SQLite o MySQL**, pero optimizado para **consultas analíticas**.

Se puede integrar con pandas, pero también puede correr de forma independiente.

https://duckdb.org/

DuckDB vs pandas

pandas:

- Excelente para manipulación de datos en memoria.
- Ideal en datasets pequeños a medianos.

DuckDB:

- Soporta datasets grandes en disco (CSV, Parquet).
- Usa **SQL familiar** para consultas.
- Puede integrarse con pandas para traer solo partes a la memoria.

DuckDB vs pandas

Cuando los datos **no caben en memoria**, pandas se vuelve lento o puede fallar.

DuckDB ofrece **SQL** con motor optimizado para **analítica columnar.**

Procesa directamente archivos **CSV/Parquet** sin necesidad de cargarlos por completo en memoria.

DuckDB vs pandas

```python
import duckdb
import pandas as pd

# pandas: carga todo el archivo
df = pd.read_csv("ventas_grandes.csv")
print(df["monto"].sum())

# DuckDB: procesa en disco con SQL optimizado
con = duckdb.connect()
print(con.execute("SELECT SUM(monto) FROM 'ventas_grandes.csv'").fetchone())
```

DuckDB

A diferencia de pandas que ofrece:

ofrece _to_csv(), _to_excel(), etc.

DuckDB utiliza el verbo **COPY** en la consulta

```python
import duckdb

# Conectar
con = duckdb.connect()

# Archivos de entrada
ventas = "ventas.csv"
productos = "productos.csv"

# Consulta: unir ventas y productos, calcular totales y promedios
query = f"""
    SELECT
        p.categoria,
        COUNT(*) AS num_ventas,
        SUM(v.monto) AS total_ventas,
        AVG(v.monto) AS promedio_venta
    FROM '{ventas}' v
    JOIN '{productos}' p
      ON v.id_producto = p.id
    WHERE v.fecha BETWEEN '2023-01-01' AND '2023-12-31'
    GROUP BY p.categoria
    ORDER BY total_ventas DESC
"""

# Ejecutar y exportar el reporte a CSV
con.execute(f"COPY ({query}) TO 'reporte_ventas.csv' (HEADER, DELIMITER ',');")

print("Reporte generado: reporte_ventas.csv")
```

Encuesta

DuckDB puede procesar directamente un archivo Parquet de millones de filas sin cargarlo por completo en memoria porque:

Encuesta

DuckDB puede procesar directamente un archivo Parquet de millones de filas sin cargarlo por completo en memoria porque:

Tiene una **arquitectura columnar y vectorizada** que lee solo lo necesario

CONTROL DE CALIDAD

Calidad de datos: Estructura

Podemos usar Python para describir entidades y modelos con clases,
y **validar datos de forma eficiente** aplicando reglas bien definidas.

Herramientas como **Pydantic** y **Pandera** permiten implementar este tipo de validaciones en distintos contextos de análisis.

Calidad de datos

El **Control de Datos** se enfoca en verificar no solo el **código**, sino también la **calidad de los datos**.

Garantiza que los datasets cumplen reglas de negocio y requisitos técnicos.

Sin validación, cualquier pipeline o análisis es una **caja negra poco confiable**.

Control de datos

Necesario con el fin de:

- Evitar errores en reportes o modelos.
- Detectar inconsistencias temprano.
- Asegurar confianza en los resultados.

Existen métricas que pueden calcularse en **Python o** mediante sistemas externos con el fin de realizar este control.

Métricas de Calidad

- **Completitud** → no hay valores faltantes en campos obligatorios.
- **Unicidad** → no hay duplicados en claves únicas.
- **Consistencia** → datos coherentes entre tablas/campos.
- **Validez** → valores dentro de rangos esperados (ej. precios ≥ 0).
- **Actualidad** → datos reflejan el estado más reciente.

Métricas simples

Pandas facilita producir estas **validaciones**, pero puede volverse complicado en pipelines grandes.

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "id": [1, 2, 2, 4],
    "precio": [100, -50, 200, np.nan],
    "fecha": pd.to_datetime(["2023-01-01", "2023-02-01", None, "2025-08-01"])
})

# Completitud
print("Valores nulos:", df.isnull().sum())

# Unicidad
print("Duplicados en id:", df["id"].duplicated().sum())

# Validez (precios >= 0)
print("Precios inválidos:", (df["precio"] < 0).sum())
```

pydantic

Librería para validación y gestión de datos en Python, basada en **anotaciones de tipo.**

Principales usos:

Validación de datos

Manejo seguro de **archivos de configuración**.

Modelado de entidades con reglas explícitas.

Welcome to Pydantic - Pydantic

pydantic: Un Modelo

```
from pydantic import BaseModel


class Usuario(BaseModel):
    id: int
    nombre: str
    edad: int


usuario = Usuario(id=1, nombre="Juan", edad=30)
print(usuario)
```

Welcome to Pydantic - Pydantic

pydantic: Validación

```
from pydantic import BaseModel, ValidationError


class Producto(BaseModel):
    nombre: str
    precio: float


try:
    producto = Producto(nombre="Laptop", precio="mil")
except ValidationError as e:
    print(“Datos no válidos”, e)
```

Welcome to Pydantic - Pydantic

pandera: Validación en pandas

Librería para **validación de datos en DataFrames de pandas.**

Permite definir **esquemas de validación** de columnas, tipos y reglas.

Integra validación dentro de los flujos de análisis → asegura **calidad y consistencia.**

https://pandera.readthedocs.io/en/stable/

pandera: Validación en pandas

```
import pandera as pa
from pandera import Column, DataFrameSchema

schema = DataFrameSchema({
    "id": Column(int, unique=True, nullable=False),
    "precio": Column(float, checks=pa.Check.ge(0)),
    "fecha": Column(pa.DateTime, nullable=False),
})

# Validación automática
schema.validate(MI_DATAFRAME)
```

https://pandera.readthedocs.io/en/stable/

pandera: Validación en pandas

Ofrece funcionalides para calidad de datos.

- Verificación de **tipos de datos** en columnas (int, float, datetime).
- Comprobación de **restricciones**: valores únicos, no nulos, rangos numéricos.
- Definición de **reglas de negocio** (ej. precios ≥ 0, fechas válidas).
- Validación dentro de **pipelines de ETL o Machine Learning**.

https://pandera.readthedocs.io/en/stable/

pandera: Validación en pandas

Ofrece funcionalides para calidad de datos.

- Verificación de **tipos de datos** en columnas (int, float, datetime).
- Comprobación de **restricciones**: valores únicos, no nulos, rangos numéricos.
- Definición de **reglas de negocio** (ej. precios ≥ 0, fechas válidas).
- Validación dentro de **pipelines de ETL o Machine Learning**.

https://pandera.readthedocs.io/en/stable/

Soda Core

En pipelines grandes, no basta con validación local: se necesitan **frameworks de monitoreo continuo**.

- Define checks declarativos en YAML o SQL.
- Corre validaciones en bases de datos o archivos grandes.
- Se integra con orquestadores (Airflow, dbt, Dagster).

https://docs.soda.io/overview-main

Soda Core

Los siguientes chequeos pueden por ejemplo aplicarse directamente a un DataFrame

```
checks for ventas:
  - row_count > 0
  - invalid_count(precio) = 0
  - duplicate_count(id) = 0
  - freshness(fecha) < 1d
```

Genera **reportes claros** para saber si los datos cumplen reglas de negocio.

Soda Core

```python
from soda.scan import Scan
import pandas as pd

# DataFrame de ejemplo
df = pd.DataFrame({
    "id": [1, 2, 2, 4],
    "precio": [100, -50, 200, None],
    "fecha": pd.to_datetime(["2023-01-01", "2023-02-01", None, "2025-08-01"])
})

# Definir checks directamente en código
scan = Scan()
scan.add_pandas_dataframe("ventas", df)

scan.add_check("row_count > 0", table_name="ventas")
scan.add_check("duplicate_count(id) = 0", table_name="ventas")
scan.add_check("invalid_count(precio) = 0", table_name="ventas")

# Ejecutar validación
scan.execute()

# Ver resultados
print(scan.get_logs_text())
```

→

```
{
  "row_count > 0": "pass",
  "duplicate_count(id) = 0": "fail",
  "invalid_count(precio) = 0": "fail"
}
```

Importancia del Control de Datos

Los datos crudos casi nunca son perfectos, tanto de entrada como salida.

- El Control previene costos mayores en producción.
- Cada herramienta **se adapta a un nivel distinto de madurez y escala**.

Orquestadores

La adquisición, validación y transformación son pasos dentro de **pipelines de datos.** A medida que crecen los flujos, necesitamos gestionar:

- **Dependencias de datos.**
- **Reintentos automáticos.**
- **Paralelismo.**
- Etc.

Los orquestradores permiten **combinar todo lo visto** (pandas, DuckDB, Pandera, Soda) en pipelines robustas y escalables.

Encuesta

¿Cuál de estas herramientas valida DataFrames estructuralmente en memoria dentro de Python?

Encuesta

¿Cuál de estas herramientas valida DataFrames estructuralmente en memoria dentro de Python?

pandera

Laboratorio

El Github incluye el segundo laboratorio para repasar los conceptos.

https://github.com/danoc93/ista-python-analisis-2025