IES SAN ANDRÉS

ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL Y BIG DATA

SISTEMAS DE BIG DATA

UT03: GESTIÓN DE LOS DATOS

ÍNDICE

1

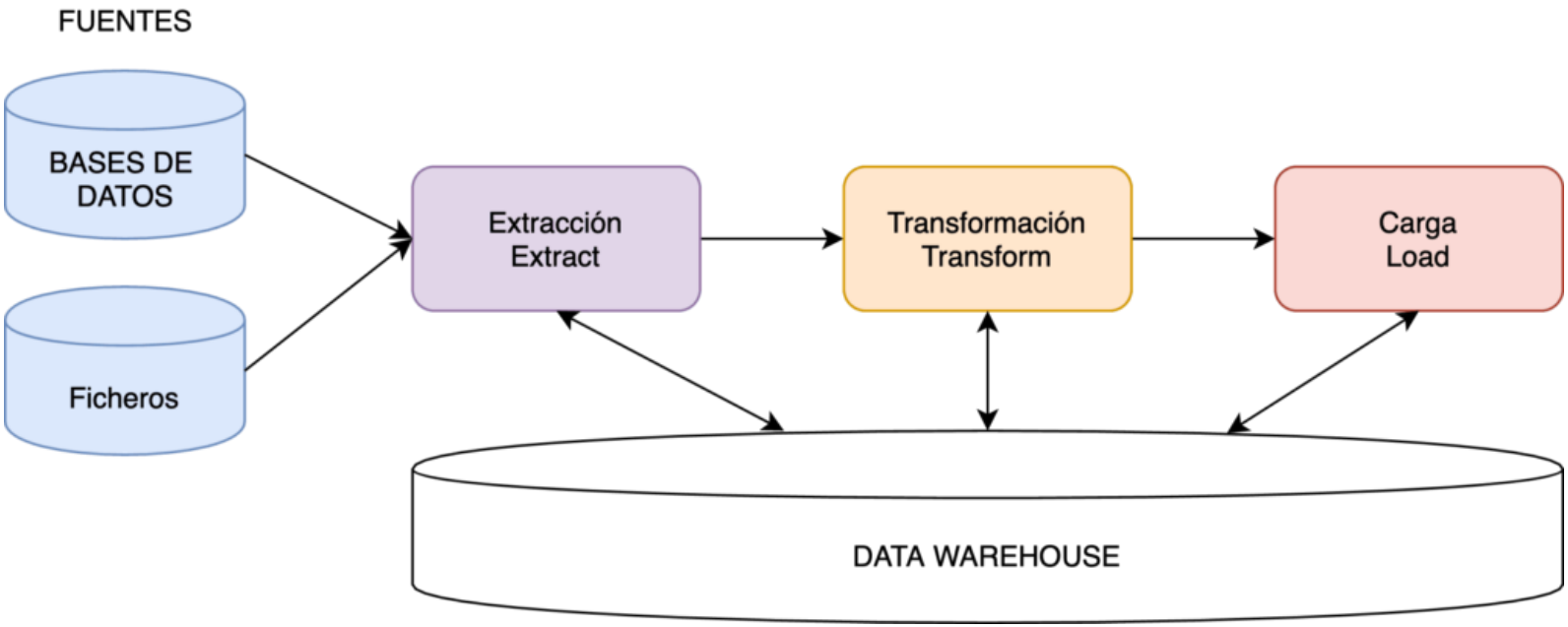
INTRODUCCIÓN A LOS PIPELINES DE DATOS

1 INTRODUCCIÓN A LOS PIPELINES DE DATOS

1.1 ¿QUÉ ES UN PIPELINE DE DATOS?

Un **pipeline de datos** es un conjunto de procesos estructurados que permiten **mover, transformar y almacenar datos** desde su origen hasta su destino final.

Este flujo suele automatizarse y definirse de forma secuencial.



El concepto no se refiere solo a mover los datos, sino también a **transformarlos y prepararlos** a lo largo del proceso, garantizando su calidad, integridad y utilidad.

Algunas etapas por las que pasa son:

- Limpieza
- Validación
- Enriquecimiento
- Combinación con otras fuentes

Un pipeline de datos bien diseñado:

- Permite automatizar tareas repetitivas
- Asegura trazabilidad y control de flujo de la información
- Facilita el escalado y mantenimiento de sistemas de datos
- Integra diferentes herramientas especializadas para cada fase del proceso

Un pipeline de datos se compone de varias **fases encadenadas**. Estas fases son:

- Ingesta de datos
- Procesamiento ETL y ELT
- Almacenamiento de datos
- Análisis de datos
- Consumo de datos

1. Ingesta de datos

La ingesta consiste en **capturar y recoger datos desde múltiples fuentes**, que pueden ser estructuradas (BBDD SQL), semiestructuradas (JSON, XML) o no estructuradas (log, imágenes, texto)

Ejemplos de fuentes:

- APIs (REST, SOAP)
- Bases de datos SQL y NoSQL
- Archivos planos (CSV, Excel, Parquet)
- Logs de servidores
- Sensores IoT
- Servicios de streaming (Kafka, MQTT)

La ingesta puede ser por lotes periódicos (**batch**) o en tiempo real (**streaming**)

2. Procesamiento: ETL y ELT

Esta fase se encarga de **preparar la información para su análisis y consumo**.

Hay dos enfoques diferentes:

- **ETL**: Extract, Transform, Load
- **ELT**: Extract, Load, Transform

E T L Extract – Transform - Load

Es el enfoque utilizado en arquitecturas de **Data Warehousing**

Los pasos son:

- **Extract**: se recopilan datos desde diferentes fuentes
- **Transform**: se **aplica lógica de negocio** a los datos. Por ejemplo:
  - Tareas de limpieza (eliminación de duplicados, nulos)
  - Conversión de tipos de datos
  - Normalización y estandarizaciones
  - Agregaciones y cálculos
  - Enriquecimiento
- **Load**: finalmente los datos se almacenan en un sistema destino **estructurado** como un Data Warehouse

E LT Extract – Load - Transform

Es el enfoque utilizado en arquitecturas de **Cloud Data Warehousing** y **Data Lakes**

Los pasos son:

- **Extract**: se recopilan datos desde diferentes fuentes
- **Load**: los datos se cargan inmediatamente en el sistema destino en su estado original (*raw*) sin modificaciones previas
- **Transform**: se aplica la lógica a los datos dentro del sistema destino, por ejemplo:
  - Ejecución de consultas SQL para limpiar y filtrar datos cargados
  - Modelado de datos y creación de vistas para análisis
  - Generación de tablas agregadas para reportes finales

1 INTRODUCCIÓN A LOS PIPELINES DE DATOS

1.2 INTRODUCCIÓN A PANDAS

¿Qué es Pandas?

Pandas es una biblioteca de software de código abierto escrita para el lenguaje de programación Python.

Su propósito fundamental es la **manipulación y el análisis de datos**.

Está construido sobre **Numpy**, otra librería de Python especializada en cálculos numéricos sobre arrays de datos.

Pandas, cuyo nombre viene PANel DAta, fue desarrollado en 2008 por **Wes McKinney** mientras trabajaba en AQR Capital Management.

En 2009 fue liberado como Open Source.

En la actualidad es el **estándar de facto** en la manipulación de datos programática.

2

EXTRACCIÓN: CONECTANDO CON LAS FUENTES

2 EXTRACCIÓN: CONECTANDO CON LAS FUENTES

2.1 FORMATOS DE ARCHIVOS COMUNES: CSV

Algunos ejemplos de archivos comunes que podemos encontrar son:

- Archivos de texto plano
- Ficheros CSV (*Comma Separated Values*)
- Documentos Excel
- Archivos JSON
- Formatos específicos: Parquet y Feather

Archivos CSV

Los archivos **CSV (Comma Separated Values)** son un formato universal para el intercambio de datos.

Son simples, legibles por humanos y soportados por casi cualquier sistema informático, desde mainframes de los años 80 hasta las nubes modernas.

Aunque comúnmente los llamamos CSV, en realidad son **Archivos de Texto Delimitados**, y tienen 3 componentes clave que debemos tratar:

- El delimitador
- La codificación
- El encabezado

El delimitador | La codificación | El encabezado

Es el carácter que separa una columna de la otra.

Puede ser:

- **Coma (,)**: el estándar oficial del CSV
- **Punto y coma (;):** muy común en Europa y Latinoamérica, ya que son lugares donde se usa la coma como separador decimal (10,5)
- **Tabulador(`\t`):** se llaman archivos TSV (Tab Separated Values). Son muy seguros porque es raro que un archivo contenga una tabulación accidentalmente.
- **Pipe (|)**: usado a menudo en sistemas antiguos o Unix
- **Delimitadores multicarácter**: a veces, un único carácter no es seguro porque el texto mismo podría contenerlo, por lo que en ocasiones podemos encontrar secuencias de dos o más caracteres como `::` o `~!~`

- **Espacios en blanco variables**: es el clásico en archivos de logs de servidores o salidas de comando de Linux, donde hay uno o más espacios de forma que se vean visualmente alineados.

```
IP           USER    STATUS
192.168.1.1  admin   OK
10.0.0.5     guest   ERROR
```

- **Caracteres invisibles (ASCII Control Characters):** en el mundo de Hadoop, Hive y Mainframes se suelen usar caracteres que no se pueden imprimir ni ver en un editor normal. Esto garantiza al 100% que el delimitador nunca aparezca dentro del texto de un usuario. Por ejemplo, `\x01` Start of Heading (es el delimitador por defecto de Hive) o `\0` (Null byte)

El delimitador | **La codificación** | El encabezado

Es la tabla de caracteres que usa el archivo para entender letras y símbolos. La elección de un sistema de codificación incorrecto mostrará caracteres extraños (`EspaÃ©a` en lugar de `España`).

Los sistemas más habituales son:

- **UTF-8**: el estándar moderno. Soporta emojis y todos los idiomas. Siempre intenta este primero.
- **Latin-1 (ISO-8859-1)**: el estándar antiguo de Windows/Europa Occidental. Muy común en archivos generados por Excel antiguos o sistemas *legacy* bancarios

Opcionalmente, la primera fila del archivo puede contener los nombres de las columnas.

Si el fichero no tiene encabezado, habrá que proporcionar los nombres manualmente durante la extracción.

pd.read_csv()

Carga un fichero CSV en un dataframe

```
import pandas as pd

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

print(df.head())
print(df.dtypes)
```

Pandas detecta automáticamente encabezados y separadores

```
     ID         Nombre      Departamento  Salario_Anual Fecha_Contratacion
0  1001      Ana García         Marketing          45000         2021-03-15
1  1002    Carlos Ruiz            Ventas          38500         2022-07-01
2  1003  Elena Vázquez        Desarrollo          52000         2020-11-20
3  1004   Jorge Méndez  Recursos Humanos          32000         2023-01-10
4  1005   Lucía Torres          Finanzas          47000         2019-05-25
ID                    int64
Nombre               object
Departamento         object
Salario_Anual         int64
Fecha_Contratacion   object
dtype: object
```

También reconoce los tipos de datos de las columnas

Algunos parámetros que puede tener son:

- **`sep`** o **`delimiter`**: el carácter que separa las columnas, por defecto coma
- **`header`**: fila que contiene los nombres de las columnas. `0` es la primera y `None` para indicar que no tiene.
- **`names`**: si `header=None` o se quieren renombrar. Es una lista con una cadena por cada campo.
- **`index_col:`** usa una columna del archivo como índice etiqueta de fila en lugar de números consecutivos. Ej.: `index_col='ID_Cliente'`
- **`use_cols:`** carga solo las columnas que se indiquen. Ahorra mucha RAM
- **`nrows:`** lee solo las N primeras filas. Útil para pruebas
- **`skiprows:`** salta filas al inicio (títulos, …)

- **`dtype`**: fuerza el tipo de dato de las columnas. Ej.: `dtype={'ID': str, 'Edad': 'int32}`
- **`parse_dates`**: intenta convertir columnas a objetos datetime automáticamente. Ej.: `parse_dates=['fecha_compra']`
- **`converters`**: aplica una función de Python a los datos mientras lee. Lento pero potente. Ej.: `converters={'precio': limpiar_moneda}`
- **`na_values`**: qué hacer si una fila tiene más columnas de las esperadas. Los posibles valores son `error` (opción por defecto), `skip` y `warn`
- **`encoding`**: codificación de caracteres, algunos valores pueden ser `utf-8`, `latín-1` o `cp1252`

Ejemplo:

```python
import pandas as pd

df = pd.read_csv(
    'datos.csv',
    sep=',',
    header=0,
    index_col='ID',
    encoding='utf-8',
    parse_dates=['Fecha_Contratacion'],
    dtype={'Departamento': 'category'}
)

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, 1001 to 1005
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Nombre             5 non-null      object
 1   Departamento       5 non-null      category
 2   Salario_Anual      5 non-null      int64
 3   Fecha_Contratacion 5 non-null      datetime64[ns]
dtypes: category(1), datetime64[ns](1), int64(1), object(1)
memory usage: 377.0+ bytes
None
```

Aunque la función read_csv() funciona muy bien, **en el mundo real lo habitual es encontrar archivos mal formados o corruptos** que hacen que el script lance un error fatal a mitad de proceso.

Veamos cómo gestionar algunos de los **errores más comunes** al leer archivos de texto con Pandas.

Error de estructura (`ParseError`)

Ocurre cuando una fila tiene más o menos columnas de las que debería. Por defecto, Pandas lanzará un `ParseError` y detendrá todo.

La solución es utilizar el parámetro `on_bad_lines='warn'`, que mostrará un aviso pero no interrumpirá la carga

Error de tipo de dato

Ocurre cuando una columna numérica tiene basura textual.

La estrategia es no forzar `dtype='int'`. Es mejor leer como `object` y limpiar en el paso de transformación.

	ID	Nombre	Departamento	Salario_Anual	Fecha_Contratacion
1	1001	Ana García	Marketing	45000	2021-03-15
2	1002	Carlos Ruiz	Ventas	38500	2022-07-01
3	1003	Elena Vázquez	Desarrollo	52000	2020-11-20
4	1004	Jorge Méndez	Recursos Humanos	ERROR	2023-01-10
5	1005	Lucía Torres	Finanzas	47000	2019-05-25

La estrategia es no forzar `dtype='int'` ya que lanzará error.

```
df = pd.read_csv(
    'datos.csv',
    sep=',',
    encoding='utf-8',
    dtype={'Salario_Anual': int}
)

print(df.head())
```

Si forzamos leerlo como entero lanzará error

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File parsers.pyx:1160, in pandas._libs.parsers.TextReader._convert_tokens()
```

Es mejor leer como `object` y limpiar en el paso de transformación.

Leemos todo como cadenas

Convertimos a valor numérico

```
df = pd.read_csv('datos.csv', dtype=str)

df['Salario_Anual'] = pd.to_numeric(df['Salario_Anual'], errors='coerce')

datos_limpios.head()
```

Las posibilidades son:
`raise`: si algo no es número lanza error
`ignore`: si algo no es número deja la columna como cadena.
`coerce`: fuerza la conversión, si algo no es número convierte a NaN

:

	ID	Nombre	Departamento	Salario_Anual	Fecha_Contratacion
0	1001	Ana García	Marketing	45000.0	2021-03-15
1	1002	Carlos Ruiz	Ventas	38500.0	2022-07-01
2	1003	Elena Vázquez	Desarrollo	52000.0	2020-11-20
3	1004	Jorge Méndez	Recursos Humanos	NaN	2023-01-10
4	1005	Lucía Torres	Finanzas	47000.0	2019-05-25

Por ejemplo, eliminando las columnas con NaN

```
errores = df[df['Salario_Anual'].isna()]
datos_limpios = df.dropna(subset=['Salario_Anual'])

datos_limpios.head()
```

	ID	Nombre	Departamento	Salario_Anual	Fecha_Contratacion
0	1001	Ana García	Marketing	45000.0	2021-03-15
1	1002	Carlos Ruiz	Ventas	38500.0	2022-07-01
2	1003	Elena Vázquez	Desarrollo	52000.0	2020-11-20
4	1005	Lucía Torres	Finanzas	47000.0	2019-05-25

Valores nulos no estándar

Por defecto, Pandas sabe que `NaN`, `null` o una celda vacía son nulos.

Pero podemos encontrar con datos de sistemas antiguos a veces usan cosas creativas como `?`, `-`, `SIN_DATO` o `9999`.

La solución es utilizar el parámetro `na_values` al que se le pasa una lista que define a qué se le asigna el valor NaN en el archivo.

```
df = pd.read_csv(
    'daots.csv',
    na_values=['?', 'SIN_RESPUESTA', '-', 'n/a']
)
```

El motor de lectura (`engine`)

A veces, el delimitador es complejo (ej: regex) o el archivo tiene caracteres muy extraños y el motor estándar de Pandas (escrito en C para velocidad) falla.

La solución es cambiar al motor de Python. Es más lento, pero mucho **más tolerante** y potente para casos extremos.

```
df = pd.read_csv(
    'archivo_raro.txt',
    sep='--',
    engine='python')
```

Por ejemplo, el motor C falla bastante con separadores multicarácter

2 EXTRACCIÓN: CONECTANDO CON LAS FUENTES

2.2 FORMATOS DE ARCHIVOS COMUNES: EXCEL

Leer archivos de Excel es diferente a leer CSV.

Mientras que CSV es texto plano simple, un archivo Excel (.xlsx) es técnicamente un contenedor comprimido (ZIP) lleno de XMLs, con estilos, múltiples hojas y metadatos.

	A	B	C	D	E	F	G
1	España						
2	Actualizado: jueves, 03 enero 2019						
3	Fecha: miércoles, 02 enero 2019						
4							
5	Estación	Provincia	Temperatura máxima (ºC)	Temperatura mínima (ºC)	Temperatura	Racha (km/h)	Velocidad máxima (km/h)
6	Estaca de Bares	A Coruña	11.1 (09:30)	9.9 (23:40)	10,5	105 (23:59)	63 (23:00)
7	As Pontes	A Coruña	11.4 (14:10)	-0.9 (04:30)	5,2		
8	A Coruña	A Coruña	12.3 (13:20)	6.0 (05:40)	9,2	37 (18:20)	24 (17:40)
9	A Coruña Aeropuerto	A Coruña	12.4 (15:50)	-2.0 (08:50)	5,2	21 (15:30)	13 (07:20)
10	Carballo, Depuradora	A Coruña	12.5 (15:30)	-2.0 (09:20)	5,3		
11	Cabo Vilan	A Coruña	11.3 (15:10)	7.2 (07:50)	9,3	54 (15:30)	43 (15:30)
12	Vimianzo	A Coruña	10.8 (15:40)	3.0 (01:00)	6,9		
13	Fisterra	A Coruña	10.9 (15:10)	5.7 (09:20)	8,3	58 (10:10)	44 (05:10)
14	Mazaricos	A Coruña	10.0 (16:10)	0.8 (04:00)	5,4		
15	Sobrado	A Coruña	10.3 (15:30)	-1.8 (03:40)	4,3		
16	Santiago de Compostela Aeropuerto	A Coruña	11.8 (15:50)	1.3 (04:00)	6,6	18 (13:10)	7 (13:20)
17	Noia	A Coruña	12.2 (16:10)	4.2 (07:50)	8,2	26 (12:20)	9 (12:40)
18	Boiro	A Coruña	15.9 (15:40)	0.9 (05:40)	8,4		
19	Padrón	A Coruña	13.8 (15:30)	-0.6 (07:40)	6,6	34 (12:20)	21 (12:20)
20	Santiago de Compostela	A Coruña	13.1 (16:10)	1.5 (07:30)	7,3	21 (20:10)	9 (01:40)

Esto hace que la extracción sea **más lenta** y requiera parámetros específicos para navegar dentro del archivo.

pd.read_excel()

Esta es la función que utilizaremos para leer archivos de Excel desde Pandas.

Pero requiere **instalar librerías** para poder hacerlo:

- Para **.xlsx** (Excel actual): `openpyxl`
- Para **.xls** (Excel 97-2003): `xlrd`

```
!pip install openpyxl
!pip install xlrd
```

```
Requirement already satisfied: openpyxl in /opt/conda/lib/python3.11/site-packages (3.1.2)
Requirement already satisfied: et-xmlfile in /opt/conda/lib/python3.11/site-packages (from
Requirement already satisfied: xlrd in /opt/conda/lib/python3.11/site-packages (2.0.1)
```

Una característica clave de Excel son las **hojas**, ya que un archivo de Excel es un libro que contiene varias hojas.

El parámetro que podemos utilizar la seleccionar la hoja a leer es **`sheet_name`**.

Si no indicamos este parámetro por defecto leerá la primera hoja.

```
# Leemos la primera hoja
df = pd.read_excel('./Aemet2019-01-02.xls')

# Leemos la hoja llamada Sevilla
df = pd.read_excel(
            './Aemet2019-01-02.xls',
            sheet_name="Sevilla")

# Leemos la tercera hoja pasando un entero
df = pd.read_excel(
            './Aemet2019-01-02.xls',
            sheet_name=2)
```

Si pasamos el parámetro **sheet_name=None**, leerá todas las hojas, pero devolverá un **diccionario de DataFrames** en lugar de un solo DataFrame.

```
dict_df = pd.read_excel(
            "Aemet2019-01-02.xls",
            sheet_name=None
)
print(type(dict_df))
```

```
<class 'dict'>
```

Normalmente los archivos Excel tienen líneas con logotipos, encabezados,.. que proporcionan información visual pero que no es relevante para extraer datos.

En estos casos podemos usar alguno de los siguientes parámetros:

- **`header`**: indica la fila en la que se encuentra el encabezado de los datos. Por ejemplo: `header=3`
- **`usecols`**: indica las columnas que hay que leer. Ejemplo: `usecols='A:F'`
- **`skiprows`**: salta las primeras filas antes de los datos. Ejemplo: `skiprows=5`
- **`skipfooter`**: salta las filas del final de la hoja. Ejemplo: `skipfooter=2`

Ejemplo de carga de datos de Excel:

```
df = pd.read_excel(
    'nomina_rrhh.xlsx',
    sheet_name='Plantilla_Activa',
    engine='openpyxl',
    header=2,
    usecols='A:E',
    dtype={'ID_Empleado': str}
)

print(df.head())
```

Cargamos solo una hoja

Indicamos explícitamente el motor. Opcional, pero recomendado

Los títulos están en la 3ª fila

Solo cargamos las columnas A hasta la E

En este ejemplo, indico que la columna ID_Empleado se leerá como cadena

2 EXTRACCIÓN: CONECTANDO CON LAS FUENTES

2.3 FORMATOS DE ARCHIVOS COMUNES: JSON

El formato **JSON (JavasScript Object Notation)** es el estándar actual para el intercambio de datos en la web (APIs) y bases de datos NoSQL.

A diferencia de CSV o Excel, JSON es **semi-estructurado y jerárquico** (anidado), lo que supone un reto único: hay que *aplanarlo* para que quepa en una tabla.

```json
{
    "count": 7,
    "items": ["socks", "pants", "shirts", "hats"],
    "manufacturer": {
        "name": "Molly's Seamstress Shop",
        "id": 39233,
        "location": {
            "address": "123 Pickleton Dr.",
            "city": "Tucson",
            "state": "AZ",
            "zip": 85705
        }
    },
    "total_price": "$393.23",
    "purchase_date": "2022-05-30",
    "country": "USA"
}
```

pd.read_json()

Con esta función podemos leer un archivo JSON

```
[
  {"id": 1, "producto": "Laptop", "precio": 1200},
  {"id": 2, "producto": "Mouse", "precio": 25}
]
```

```
# Pandas infiere que es una lista de registros
df = pd.read_json('datos.json')
print(df)
```

El problema es cuando encontramos un **JSON anidado**, es decir, tiene datos dentro de datos

```
[
  {
    "id": 1,
    "nombre": "Ana",
    "contacto": {
        "email": "ana@test.com",
        "telefono": "555-0001"
    }
  }
]
```

Si en este caso usamos read_json() obtendremos una columna llamada contacto que contiene diccionarios enteros, lo que no nos sirve.

Necesitamos **aplanar el JSON**.

pd.json_normalize()

Esta función *explota* las jerarquías

```python
import json

with open('usuarios.json') as f:
    data = json.load(f)

df = pd.json_normalize(data)

# RESULTADO VISUAL DEL DATAFRAME:
# id | nombre | contacto.email | contacto.telefono
# 1  | Ana    | ana@test.com   | 555-0001
```

Primero lo cargamos con la librería estándar json

Y luego lo aplanamos

Observa como nombra las columnas para reflejar la jerarquía del archivo JSON

Ejemplo:

```
df = pd.read_json('./components.json')
df
```

	id	nombre	especificaciones
0	101	Laptop Gamer	{'ram': '16GB', 'disco': '1TB SSD'}
1	102	Ratón Inalámbrico	{'ram': 'N/A', 'disco': 'N/A'}

```
with open('components.json') as f:
    data = json.load(f)

df = pd.json_normalize(data)
df
```

	id	nombre	especificaciones.ram	especificaciones.disco
0	101	Laptop Gamer	16GB	1TB SSD
1	102	Ratón Inalámbrico	N/A	N/A

```
[
  {
    "id": 101,
    "nombre": "Laptop Gamer",
    "especificaciones": {
      "ram": "16GB",
      "disco": "1TB SSD"
    }
  },
  {
    "id": 102,
    "nombre": "Ratón Inalámbrico",
    "especificaciones": {
      "ram": "N/A",
      "disco": "N/A"
    }
  }
]
```

El problema es que, si el JSON tiene más niveles de anidamiento no funcionará como esperamos:

```json
[
    {
        "ciclo": "DAM",
        "curso": "Primero",
        "estudiantes": [
            {"nombre": "Juan", "nota": 8},
            {"nombre": "Lucia", "nota": 9}
        ]
    },
    {
        "ciclo": "ASIR",
        "curso": "Segundo",
        "estudiantes": [
            {"nombre": "Pedro", "nota": 5}
        ]
    }
]
```

```python
with open('students.json') as f:
    data = json.load(f)

df = pd.json_normalize(data)
df
```

	ciclo	curso	estudiantes
0	DAM	Primero	[{'nombre': 'Juan', 'nota': 8}, {'nombre': 'Lu...
1	ASIR	Segundo	[{'nombre': 'Pedro', 'nota': 5}]

Para evitar esto, deberemos utilizar los parámetros:

- **`record_path`**: sirve para indicar a Pandas de dónde queremos sacar cada fila de la tabla final

```
df = pd.json_normalize(
    data,
    record_path=['estudiantes']
)
df
```

	nombre	nota
0	Juan	8
1	Lucia	9
2	Pedro	5

Observa que en este caso cargamos los datos que hay dentro de estudiantes, pero perdemos los datos de ciclo y curso.

- **`meta`**: con este parámetro indicamos qué datos del nivel superior queremos pegar al lado de cada estudiante.

```
df = pd.json_normalize(
    data,
    record_path=['estudiantes'],
    meta=['ciclo', 'curso']
)
df
```

	nombre	nota	ciclo	curso
0	Juan	8	DAM	Primero
1	Lucia	9	DAM	Primero
2	Pedro	5	ASIR	Segundo

En este caso conservo los datos del nodo padre

JSON Lines

Algo habitual es encontrar archivos con el formato **JSON Lines**, donde cada línea del archivo es un objeto JSON independiente en lugar de ser todo el archivo un solo objeto gigante.


```
{"id": "0", "question": "-5 *   -6", "answer": "30", "num_terms": 2, "num_digits": 1}
{"id": "1", "question": "30 * 44 =", "answer": "1320", "num_terms": 2, "num_digits": 2}
{"id": "2", "question": "621242   - 793732 = ?", "answer": "-172490", "num_terms": 2, "num_digits": 6}
{"id": "3", "question": "What is 97 + -41 + -19 - -31 + -45 * 84?", "answer": "-3712", "num_terms": 6, "num_digits": 2}
{"id": "4", "question": "( -71 + 75   ) - -1 - 36 =", "answer": "-31", "num_terms": 4, "num_digits": 2}
{"id": "5", "question": "  -( -(   -(   -5 -   8 ) * 1 ) ) - -6 - -2 = ?", "answer": "21", "num_terms": 5, "num_digits": 1}
{"id": "6", "question": "( 69 - 20 * -86 * -84 ) -   -72", "answer": "-144339", "num_terms": 5, "num_digits": 2}
{"id": "7", "question": "5 - -45", "answer": "50", "num_terms": 2, "num_digits": 2}
{"id": "8", "question": "What is   -( -( 330191 - -907543   ) +     617155 )?", "answer": "620579", "num_terms": 3, "num_digits": 6}
{"id": "9", "question": "-8 - -3 * -6 =", "answer": "-26", "num_terms": 3, "num_digits": 1}
{"id": "10", "question": "What is 89 - 6   * -31?", "answer": "275", "num_terms": 3, "num_digits": 2}
```


Para hacer más eficiente la carga de datos es necesario utilizar el parámetro `lines=True`.

```python
df = pd.read_json(
    'logs_servidor.json',
    lines=True)
```

2 EXTRACCIÓN: CONECTANDO CON LAS FUENTES

2.4 FORMATOS DE ARCHIVOS COMUNES: XML

Otro tipo de archivo que encontraremos habitualmente es **XML (Extensible Markup Languaje)**.

Al igual que pasa con JSON, XML es **jerárquico** mientras que Pandas es tabular, por lo que necesitaremos **aplanar** esa estructura.

```
<?xml version="1.0"?>
- <birds>
  - <owl id="1201">
      <species>Bubo bubo</species>
      <name>Eagle Owl</name>
      <region>Eurasia</region>
    </owl>
  - <owl id="1202">
      <species>Strix occidentalis</species>
      <name>Spotted Owl</name>
      <region>North America</region>
    </owl>
</birds>
```

Antes de leer un fichero XML deberemos tener instalada la librería `lxml`.

```
!pip install lxml
```

pd.read_xml()

Este es el método más sencillo.

Pandas intentará detectar automáticamente la estructura repetitiva (filas) y los datos (columnas)

```xml
<biblioteca>
    <libro categoria="ficcion">
        <titulo>El Quijote</titulo>
        <autor>Cervantes</autor>
        <precio>20.50</precio>
    </libro>
    <libro categoria="educacion">
        <titulo>Python Data Science</titulo>
        <autor>VanderPlas</autor>
        <precio>45.00</precio>
    </libro>
</biblioteca>
```

```
df = pd.read_xml('biblioteca.xml')

df
```

	categoria	titulo	autor	precio
0	ficcion	El Quijote	Cervantes	20.5
1	educacion	Python Data Science	VanderPlas	45.0

El problema es cuando hay niveles de **anidamiento**.

```xml
<empresa>
    <departamento nombre="IT">
        <empleados>
            <empleado id="1">Juan</empleado>
            <empleado id="2">Ana</empleado>
        </empleados>
    </departamento>
    <departamento nombre="HR"> ... </departamento>
</empresa>
```

```python
df = pd.read_xml('biblioteca.xml')
df
```

	nombre	empleados	departamento
0	IT	\n	None
1	HR	None	...

En esos casos debemos utilizar el parámetro **`xpath`** para indicar dónde debe buscar Pandas los datos.

```
df = pd.read_xml(
    'data.xml',
    xpath=".//empleado"
)
df
```

Le indicamos que busque la etiqueta llamada *empleado*

	id	empleado
0	1	Juan
1	2	Ana

```xml
<tienda>
    <seccion nombre="electronica">
        <producto id="A001">
            <nombre>Laptop</nombre>
            <precio moneda="USD">800</precio>
        </producto>
    </seccion>
    <seccion nombre="ropa">
        <producto id="B005">
            <nombre>Camiseta</nombre>
            <precio moneda="EUR">20</precio>
        </producto>
    </seccion>
</tienda>
```

Hay 4 formas de indicar la ruta:

- **Rutas absolutas**: comienza por barra inclinada simple (/) e indica la ruta desde la raíz hasta el elemento

```
df = pd.read_xml(
    'data.xml',
    xpath="/tienda/seccion/producto"
)
df
```

	id	nombre	precio
0	A001	Laptop	800
1	B005	Camiseta	20

- **Rutas relativas**: empieza con doble balla (//) y le dice al sistema: *busca este elemento donde sea que esté, sin importar la profundidad*.

```
df = pd.read_xml(
    'data.xml',
    xpath="//producto"
)
df
```

	id	nombre	precio
0	A001	Laptop	800
1	B005	Camiseta	20

- **Selección de atributos**: en XML, los datos a veces no están entre las etiquetas, sino en atributos. Para ello usaremos la arroba (@)

- **Predicados**: los corchetes funcionan como un WHERE de AQL, permiten seleccionar nodos basándonos en condiciones específicas. Se indican por corchetes. Ejemplos:
  - Por posición:
    - `//producto[1]`: selecciona el primer producto que encuentre
    - `//producto[last()]`: selecciona el último producto
  - Por atributo:
    - `//sección[@nombre='electrónica']/producto`: los productos que están dentro de una sección llamada `electrónica`.
  - Por valor hijo:
    - `//producto[precio>50]`: los productos cuyo `precio` sea mayor que 50

Otra situación que nos podemos encontrar es que el XML tenga la misma información duplicada en atributos y etiquetas