Extracción de Datos desde Múltiples Fuentes - Día 2

Change Status

pending · 40 min

Learning Objectives

1. Implementar estrategias de extracción desde archivos locales (CSV, Excel, JSON)
2. Establecer conexiones seguras a bases de datos SQL usando Python
3. Consumir datos desde APIs REST con manejo apropiado de errores
4. Gestionar configuraciones y credenciales de conexión de manera segura

Actividades y Aprendizajes

Aprende todo sobre funciones y módulos en Python con ejemplos prácticos.

Task 1: Extracción desde Archivos Estructurados (10 minutos)

Los archivos planos representan la **forma más común y accesible** de compartir datos, pero requieren técnicas específicas para una extracción robusta y eficiente.

CSV: El Formato Universal de Datos

CSV (Comma-Separated Values) es el "esperanto" de los datos - **universalmente comprendido** pero con **variaciones sutiles** que pueden causar problemas.

Estructura básica: Cada línea representa un registro, valores separados por delimitadores (coma, punto y coma, tab).

Variaciones problemáticas:

Delimitadores diferentes: Algunos usan `;` en lugar de `,`
Encoding de texto: UTF-8 vs ISO-8859-1 vs Windows-1252
Números con separadores: `1,234.56` vs `1.234,56`
Campos con comillas: `"Nombre, Apellido"` vs `Nombre, Apellido`
Líneas multilinea: Campos que contienen saltos de línea

Estrategias de lectura robusta:

```
# Lectura básica
df = pd.read_csv('datos.csv')

# Lectura avanzada con parámetros
df = pd.read_csv('datos.csv',
                 sep=';',                    # delimitador personalizado
                 encoding='utf-8',           # encoding específico
                 decimal=',',               # separador decimal
                 quotechar='"',             # carácter de cita
                 error_bad_lines=False)     # manejar líneas problemáticas
```

Detección automática: Pandas puede inferir automáticamente muchos parámetros, pero es mejor ser explícito para evitar sorpresas.

Excel: Datos Empresariales Complejos

Excel combina datos tabulares con **riqueza adicional** que puede ser tanto bendición como complicación.

Ventajas de Excel:

Múltiples hojas: Un archivo puede contener diferentes datasets relacionados
Metadatos ricos: Formatos, colores, fórmulas que pueden contener información valiosa
Familiaridad universal: Todos saben usar Excel

Desafíos técnicos:

Múltiples formatos: .xls vs .xlsx con diferentes capacidades
Hojas múltiples: ¿Cuál leer? ¿Todas o específica?
Tipos de datos mixtos: Excel permite mezclar texto y números en la misma columna
Fórmulas: ¿Leer valores calculados o las fórmulas mismas?

Lectura eficiente:

```
# Leer hoja específica
df = pd.read_excel('datos.xlsx', sheet_name='Ventas')

# Leer múltiples hojas
excel_data = pd.read_excel('datos.xlsx', sheet_name=['Ventas', 'Clientes'])
df_ventas = excel_data['Ventas']
df_clientes = excel_data['Clientes']

# Especificar tipos de datos
df = pd.read_excel('datos.xlsx', dtype={'id': int, 'precio': float})
```

JSON: Datos Semi-Estructurados

JSON representa datos **jerárquicos y flexibles** que requieren estrategias diferentes a las tablas planas.

Estructuras JSON comunes:

Array de objetos: `[{'nombre': 'Ana', 'edad': 25}, ...]`
Objeto anidado: `{'cliente': {'nombre': 'Ana', 'edad': 25}, 'productos': [...]}`

Lectura con Pandas:

```
# JSON plano (array de objetos)
df = pd.read_json('datos.json')

# JSON anidado - requiere procesamiento previo
import json
with open('datos.json') as f:
    data = json.load(f)

# Aplanar estructuras anidadas
df = pd.json_normalize(data, 'productos', ['cliente.nombre', 'cliente.ciudad'])
```

Task 2: Conexión a Bases de Datos SQL (10 minutos)

Las bases de datos SQL representan el **corazón de los sistemas empresariales**, requiriendo conexiones especializadas para extracción eficiente.

Arquitectura de Conexión

Componentes necesarios:

Driver de base de datos: Biblioteca específica para cada tipo de BD (sqlite3, psycopg2, pymysql)
Cadena de conexión: URL que especifica host, puerto, base de datos, credenciales
Credenciales seguras: Manejo de usuarios y passwords sin hardcodear

Patrones de conexión segura:

```
import sqlite3
import psycopg2
import pymysql

# SQLite (archivo local)
conn = sqlite3.connect('mi_base.db')

# PostgreSQL
conn = psycopg2.connect(
    host="localhost",
    database="mi_db",
    user="mi_usuario",
    password="mi_password"
)

# MySQL
conn = pymysql.connect(
    host='localhost',
    user='mi_usuario',
    password='mi_password',
    database='mi_db'
)
```

Estrategias de Extracción SQL

Lectura completa: Para tablas pequeñas o cuando se necesita todo el historial.

```
df = pd.read_sql('SELECT * FROM clientes', conn)
```

Lectura filtrada: Para datasets grandes, aplicar filtros en la base de datos.

```
query = """
SELECT c.nombre, c.ciudad, p.fecha_pedido, p.total
FROM clientes c
JOIN pedidos p ON c.id = p.cliente_id
```

```
WHERE p.fecha_pedido >= '2024-01-01'
"""
df = pd.read_sql(query, conn)
```

Lectura por chunks: Para tablas muy grandes, procesar en bloques.

```
chunk_size = 10000
for chunk in pd.read_sql('SELECT * FROM ventas_grandes', conn, chunksize=chunk_size):
    # Procesar chunk
    procesar_chunk(chunk)
```

Manejo de Conexiones

Context managers: Asegurar que conexiones se cierren automáticamente.

```
with sqlite3.connect('mi_base.db') as conn:
    df = pd.read_sql('SELECT * FROM tabla', conn)
# Conexión cerrada automáticamente
```

Connection pooling: Para aplicaciones con múltiples conexiones simultáneas.

Task 3: Consumo de APIs REST (10 minutos)

Las APIs REST representan **datos dinámicos y actualizados** que requieren técnicas específicas de consumo programático.

Fundamentos de HTTP para Datos

Métodos HTTP comunes:

GET: Recuperar datos (lectura)
POST: Enviar datos (creación)
PUT/PATCH: Actualizar datos
DELETE: Eliminar datos

Estructura de respuesta:

Status codes: 200 (OK), 404 (Not found), 500 (Server error)
Headers: Metadata sobre la respuesta (content-type, encoding)

Body: Datos reales en formatos JSON, XML, CSV
Consumo Básico con requests

```
import requests

# GET básico
response = requests.get('https://api.ejemplo.com/datos')
if response.status_code == 200:
    data = response.json()
    df = pd.DataFrame(data)
else:
    print(f"Error: {response.status_code}")

# GET con parámetros
params = {'fecha_desde': '2024-01-01', 'limit': 100}
response = requests.get('https://api.ejemplo.com/ventas', params=params)

# GET con headers (autenticación)
headers = {'Authorization': 'Bearer mi_token_api'}
response = requests.get('https://api.ejemplo.com/clientes', headers=headers)
```

Manejo de Errores y Edge Cases

Timeouts: Evitar esperas infinitas.

```
response = requests.get('https://api.ejemplo.com/datos', timeout=30)
```

Reintentos: Para APIs inestables.

```
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

retry_strategy = Retry(total=3, backoff_factor=1)
adapter = HTTPAdapter(max_retries=retry_strategy)

with requests.Session() as session:
    session.mount("https://", adapter)
    response = session.get('https://api.ejemplo.com/datos')
```

Rate limiting: Respetar límites de la API.

```
import time

# Esperar entre requests
time.sleep(1)  # 1 segundo entre llamadas
```

Autenticación en APIs

API Keys: En headers o parámetros.

```
headers = {'X-API-Key': 'mi_clave_secreta'}
response = requests.get('https://api.ejemplo.com/datos', headers=headers)
```

OAuth 2.0: Para APIs complejas.

```
# Requiere bibliotecas adicionales como requests-oauthlib
```