Implementación – APIs y Gestión de Datos
© Universitas Europaea IMF

Indice

Implementación – APIs y Gestión de Datos

Introducción a la Implementación de APIs y Gestión de Datos

1.1. ¿Qué es una API y cuál es su importancia?

Las **APIs (Application Programming Interfaces)** son conjuntos de reglas y protocolos que permiten la comunicación entre diferentes sistemas o aplicaciones. Actúan como intermediarios que facilitan la interacción entre software, permitiendo que una aplicación utilice funcionalidades de otra sin necesidad de conocer su implementación interna.

En el desarrollo de software moderno, las APIs juegan un papel crucial, ya que permiten la integración entre sistemas, la automatización de procesos y la escalabilidad de aplicaciones. Gracias a las APIs, es posible conectar aplicaciones web con bases de datos, consumir servicios de terceros (como pagos en línea o redes sociales) y desarrollar arquitecturas distribuidas y modulares.

Las **APIs** son esenciales en el desarrollo de software moderno, permitiendo la integración de aplicaciones y la creación de sistemas escalables. **REST** es el estándar más utilizado por su simplicidad, **SOAP** es ideal para aplicaciones que requieren alta seguridad, y **GraphQL** es la mejor opción cuando se necesita flexibilidad en la recuperación de datos.

La elección del tipo de API dependerá de los requerimientos específicos del proyecto y de la infraestructura disponible.

Definición y Propósito de las APIs en el Desarrollo de Software:

El propósito principal de una API es proporcionar un **método estructurado y seguro** para que las aplicaciones intercambien datos y servicios. Entre sus principales funciones se encuentran:

- **Facilitar la integración:** Permiten que diferentes aplicaciones se comuniquen sin importar su lenguaje de programación o plataforma.
- **Reutilización de funcionalidades:** Un servicio puede exponer una API para que otros desarrolladores la usen sin necesidad de replicar su lógica.
- **Automatización de procesos:** Muchas tareas pueden ser gestionadas automáticamente mediante llamadas a APIs, como la actualización de datos en tiempo real.
- **Escalabilidad:** Permiten que las aplicaciones crezcan de manera modular, conectando nuevos servicios conforme se necesiten.

Tipos de APIs: REST, SOAP y GraphQL:

Existen diferentes tipos de APIs, cada una con sus propias características y casos de uso.

1. APIs REST (Representational State Transfer)

REST es el estándar más utilizado en el desarrollo web debido a su simplicidad y eficiencia. Se basa en principios como:

- **Uso de métodos HTTP:**
  - **GET:** Obtener datos.
  - **POST:** Enviar datos para crear un nuevo recurso.
  - **PUT/PATCH:** Actualizar un recurso existente.
  - **DELETE:** Eliminar un recurso.
- **Formato de datos común:** REST generalmente usa **JSON** para el intercambio de información, facilitando la comunicación entre sistemas.
- **Independencia entre cliente y servidor:** El cliente puede consumir la API sin conocer detalles sobre la implementación del servidor.

Ejemplo de una API REST:

```
 GET https://api.tienda.com/productos
Respuesta:
{
   "id": 1,
   "nombre": "Laptop",
   "precio": 1200
}
```

REST es ideal para aplicaciones web y móviles que requieren comunicación rápida y escalable.

2. APIs SOAP (Simple Object Access Protocol)

SOAP es un protocolo más estructurado y seguro que REST, utilizado en entornos empresariales que requieren alta seguridad e integridad en los datos.

- **Formato XML:** SOAP usa XML en lugar de JSON, lo que lo hace más complejo pero más robusto.
- **Normas estrictas:** SOAP incluye estándares de seguridad como **WS-Security** para la autenticación de usuarios.
- **Basado en mensajes:** SOAP envuelve cada solicitud y respuesta en un sobre XML, lo que lo hace más pesado en términos de procesamiento.

Ejemplo de una solicitud SOAP:

```
 <soap:Envelope>
  <soap:Body>
    <GetProducto>
      <id>1</id>
    </GetProducto>
  </soap:Body>
</soap:Envelope>
```

SOAP es útil en servicios bancarios y aplicaciones empresariales críticas donde la seguridad es una prioridad.

3. APIs GraphQL

GraphQL es un estándar más moderno que permite a los clientes solicitar exactamente los datos que necesitan, reduciendo el uso innecesario de ancho de banda.

- **Flexibilidad en consultas:** Los clientes pueden definir qué campos desean obtener en la respuesta.
- **Eficiencia en la recuperación de datos:** Evita las múltiples llamadas a diferentes endpoints que ocurren en REST.
- **Formato basado en JSON:** Hace que sea fácil de integrar en aplicaciones web y móviles.

Ejemplo de consulta en GraphQL:

```
{
 producto(id: 1) {
   nombre
   precio
 }
}
```

GraphQL es ideal para aplicaciones que manejan grandes volúmenes de datos y requieren respuestas optimizadas.

1.2. Relación entre APIs y gestión de datos

Las **APIs (Application Programming Interfaces)** desempeñan un papel fundamental en la gestión de datos al permitir la comunicación entre aplicaciones y bases de datos. A través de una API, los sistemas pueden intercambiar información de manera estructurada, facilitando el acceso, manipulación y almacenamiento de datos en tiempo real. Las APIs y la gestión de datos están estrechamente relacionadas, ya que permiten un acceso seguro, eficiente y escalable a la información almacenada en bases de datos. Al conectar una API con una base de datos, las aplicaciones pueden ofrecer una **arquitectura flexible, distribuida y de alto rendimiento**, asegurando que los datos sean accesibles en cualquier momento y desde cualquier dispositivo.

Cómo las APIs Interactúan con Bases de Datos

Las APIs actúan como intermediarios entre los clientes (como aplicaciones web o móviles) y la base de datos, gestionando las solicitudes y respuestas. El proceso típico de interacción con una base de datos a través de una API sigue estos pasos:

1. **El cliente envía una solicitud HTTP** (GET, POST, PUT, DELETE) a la API.
2. **El controlador de la API recibe la solicitud** y valida los parámetros.
3. **La API consulta la base de datos** a través de una capa de acceso a datos (ORM o SQL).
4. **Los datos son procesados** y convertidos a un formato estructurado como JSON.
5. **La API devuelve la respuesta** al cliente, que puede ser una lista de datos, una confirmación de acción o un mensaje de error.

Ejemplo de interacción:

Cliente → API → Base de Datos → API → Cliente

- **Ejemplo de solicitud REST:**

```
GET /productos/1
```

- **Respuesta JSON:**

```
{
  "id": 1,
  "nombre": "Laptop",
  "precio": 1200
}
```

Las API permiten que las aplicaciones **accedan y manipulen datos de manera segura y estructurada**, sin necesidad de interactuar directamente con la base de datos.

Beneficios de Conectar APIs con Bases de Datos para la Escalabilidad:

El uso de APIs para gestionar bases de datos aporta múltiples ventajas en términos de **escalabilidad, flexibilidad y rendimiento**.

1. **Separación de responsabilidades:**
   - Las APIs desacoplan la lógica de negocio del almacenamiento de datos, permitiendo que múltiples clientes (web, móvil, IoT) accedan a la misma base de datos sin afectar su estructura.
2. **Escalabilidad horizontal:**
   - Las APIs permiten distribuir la carga entre múltiples servidores, facilitando el crecimiento de la aplicación sin comprometer el rendimiento.
3. **Manejo eficiente de grandes volúmenes de datos:**
   - Con consultas optimizadas y paginación, las APIs pueden gestionar datos en tiempo real sin afectar la velocidad de respuesta.
4. **Seguridad y control de acceso:**
   - A través de mecanismos como **tokens JWT y OAuth**, las APIs pueden gestionar permisos para restringir el acceso a la base de datos según los privilegios del usuario.
5. **Facilita la migración y la interoperabilidad:**
   - Si una empresa decide cambiar de base de datos (por ejemplo, de MySQL a MongoDB), solo necesita modificar la capa de acceso de la API sin afectar a los clientes que la consumen.

1.3. Herramientas y tecnologías clave 2t

- Frameworks en Python para APIs (Flask, Django REST Framework).
- Bibliotecas para manipulación de datos (SQLAlchemy, Django ORM, pandas).

2. Diseño e Implementación de APIs REST 4t

2.1. Fundamentos de una API REST 1t

- Principios de diseño RESTful.
- Diferencias entre APIs REST y otras arquitecturas.

2.2. Endpoints y métodos HTTP 1t

- Definición de endpoints y su estructura.

- Métodos HTTP y su uso en una API REST:
    - **GET**: Recuperar información.
    - **POST**: Crear un nuevo recurso.
    - **PUT/PATCH**: Actualizar datos.
    - **DELETE**: Eliminar un recurso.

2.3. Implementación de una API REST en Flask

Flask es un framework ligero en Python que permite desarrollar **APIs REST** de manera rápida y eficiente. Su flexibilidad lo hace ideal para aplicaciones pequeñas y medianas, facilitando la gestión de rutas, controladores y respuestas en formato JSON. La implementación de una **API REST en Flask** permite gestionar datos de manera eficiente, definiendo rutas claras y respuestas en formato JSON. Al manejar errores correctamente, se mejora la estabilidad y seguridad de la aplicación.

Creación de una API REST con Flask:

Para crear una API REST en Flask, es necesario instalar el framework y configurar una estructura básica:

1. **Instalar Flask (si no está instalado):**
2. pip install flask
3. **Estructura básica de la API:**
    - app.py: Contiene el código principal de la API.
    - routes.py: Define las rutas y controladores.
    - models.py: Define la estructura de datos.

Ejemplo de una API REST con Flask:

```
 from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def home():
    return jsonify({"mensaje": "Bienvenido a la API REST con Flask"})

if __name__ == '__main__':
    app.run(debug=True)
```

Manejo de Rutas y Controladores en Flask

Las rutas permiten definir **endpoints** que ejecutan funciones específicas en la API. Flask utiliza decoradores (@app.route) para asociar funciones a rutas.

Ejemplo de una API de gestión de productos:

```
productos = [
    {"id": 1, "nombre": "Laptop", "precio": 1200},
    {"id": 2, "nombre": "Teléfono", "precio": 800}
]

@app.route('/productos', methods=['GET'])
def obtener_productos():
    return jsonify(productos)

@app.route('/productos/<int:id>', methods=['GET'])
def obtener_producto(id):
    producto = next((p for p in productos if p["id"] == id), None)
    if producto:
        return jsonify(producto)
    return jsonify({"error": "Producto no encontrado"}), 404
```

Explicación:

- GET /productos: Devuelve la lista de productos.
- GET /productos/<id>: Devuelve un producto específico según su ID.

Respuesta en Formato JSON y Gestión de Errores

Flask facilita el envío de respuestas en **formato JSON**, lo que permite que las aplicaciones cliente (web, móviles) consuman datos de forma estructurada.

Para manejar errores, se pueden personalizar respuestas en caso de datos no encontrados o solicitudes incorrectas:

```
@app.errorhandler(404)
def recurso_no_encontrado(error):
    return jsonify({"error": "Recurso no encontrado"}), 404

@app.errorhandler(500)
def error_interno(error):
    return jsonify({"error": "Error interno del servidor"}), 500
```

Beneficios de manejar errores correctamente:

- Mejora la experiencia del usuario.
- Facilita la depuración y monitoreo de la API.
- Evita que se exponga información sensible en respuestas de error.

2.4. Pruebas y documentación de APIs

Las **APIs REST** deben ser probadas y documentadas correctamente para garantizar su funcionalidad y facilitar su uso por otros desarrolladores. Herramientas como **Postman** y **Swagger** permiten realizar pruebas de endpoints y generar documentación detallada de los servicios ofrecidos. **Postman** permite realizar pruebas interactivas de endpoints, mientras que **Swagger** documenta las APIs y facilita su exploración. Ambas herramientas mejoran la calidad del desarrollo de APIs REST y aseguran su correcto funcionamiento y mantenimiento.

Uso de Postman para Pruebas de Endpoints:

Postman es una herramienta ampliamente utilizada para probar APIs REST, permitiendo a los desarrolladores realizar solicitudes HTTP y verificar las respuestas de los endpoints.

Beneficios de usar Postman:

- Facilita la ejecución de pruebas **GET, POST, PUT y DELETE** sin necesidad de escribir código.
- Permite enviar datos en formato **JSON, XML o formulario**.
- Ofrece la posibilidad de guardar y compartir colecciones de solicitudes.
- Soporta la automatización de pruebas mediante **scripts en JavaScript**.

Ejemplo de prueba con Postman:

1. Ingresar la URL de la API (ejemplo: http://localhost:5000/productos).
2. Seleccionar el método HTTP (GET, POST, etc.).
3. Agregar parámetros o cuerpo de la solicitud si es necesario.
4. Hacer clic en "Send" y analizar la respuesta.

Uso de Swagger para Documentar APIs REST:

Swagger es una herramienta que permite documentar APIs REST y generar interfaces interactivas donde los desarrolladores pueden probar endpoints sin necesidad de usar Postman.

Ventajas de usar Swagger:

- Genera documentación en **formato OpenAPI** que describe los endpoints y parámetros.
- Permite probar la API directamente desde la interfaz web.
- Mejora la comunicación entre equipos de desarrollo y clientes.

Ejemplo de configuración en Flask con flasgger:

```
 from flask import Flask
from flasgger import Swagger

app = Flask(__name__)
Swagger(app)

@app.route('/productos', methods=['GET'])
def obtener_productos():
    """Lista todos los productos disponibles
    ---
    responses:
      200:
        description: Lista de productos en formato JSON
    """
    return {"productos": ["Laptop", "Teléfono", "Tablet"]}

if __name__ == '__main__':
    app.run(debug=True)
```

Al ejecutar esta API, Swagger genera automáticamente una interfaz accesible en http://localhost:5000/apidocs/, donde se pueden ver los endpoints y probarlos.

Postman permite realizar pruebas interactivas de endpoints, mientras que **Swagger** documenta las APIs y facilita su exploración. Ambas herramientas mejoran la calidad del desarrollo de APIs REST y aseguran su correcto funcionamiento y mantenimiento.

3. Bases de Datos y Python

3.1. Bases de datos en aplicaciones Python

Las **bases de datos** son un componente esencial en el desarrollo de aplicaciones en **Python**, especialmente en aquellas que requieren almacenamiento y gestión de datos persistentes. Existen dos tipos principales de bases de datos: **relacionales (SQL)** y **no relacionales (NoSQL)**, cada una con sus características y casos de uso específicos.

Bases de Datos Relacionales vs. No Relacionales:

1. **Bases de datos relacionales (SQL)**
    - Organizan la información en **tablas con filas y columnas**.
    - Utilizan **Structured Query Language (SQL)** para gestionar los datos.
    - Aplican **relaciones entre entidades** mediante claves primarias y foráneas.
    - Ejemplos: **MySQL, PostgreSQL, SQLite, SQL Server**.
2. **Bases de datos no relacionales (NoSQL)**
    - No siguen una estructura tabular rígida, sino que usan formatos más flexibles como **documentos, clave-valor, grafos o columnas**.
    - Diseñadas para **alta escalabilidad y velocidad de acceso**.
    - No requieren esquemas estrictos, lo que permite almacenar datos dinámicos.
    - Ejemplos: **MongoDB, Firebase, Redis, Cassandra**.

Cuándo Usar SQL vs. NoSQL en una API

La elección entre **SQL y NoSQL** depende de las necesidades de la API y del tipo de datos que se manejan:

- **Usar SQL cuando:**
    - Se requiere **integridad y consistencia** en los datos.
    - Hay **relaciones complejas** entre entidades (por ejemplo, un sistema de gestión de clientes y pedidos).
    - Se necesita realizar **consultas estructuradas** con JOINs y transacciones.
- **Usar NoSQL cuando:**
    - Se necesita **escalabilidad horizontal** y alta velocidad en consultas.
    - Los datos son **no estructurados o semiestructurados** (por ejemplo, almacenamiento de JSON en MongoDB).
    - Se manejan **grandes volúmenes de datos en tiempo real**, como en aplicaciones de redes sociales o IoT.

3.2. ORM con SQLAlchemy

El **Object-Relational Mapping (ORM)** es una técnica que permite interactuar con bases de datos utilizando **objetos en lugar de consultas SQL directas**. **SQLAlchemy** es una de las bibliotecas ORM más utilizadas en **Python**, permitiendo a los desarrolladores gestionar bases de datos relacionales de forma eficiente y flexible dentro de aplicaciones **Flask**. El uso de **SQLAlchemy** en Flask facilita la interacción con bases de datos relacionales, **automatizando la creación de tablas y la ejecución de consultas mediante objetos en Python**. Gracias a sus capacidades ORM, **permite un desarrollo más rápido, seguro y escalable**, reduciendo la necesidad de escribir consultas SQL manualmente.

¿Qué es un ORM y por qué usarlo?

Un **ORM** actúa como un puente entre el código en Python y la base de datos, permitiendo:

- **Evitar escribir consultas SQL manualmente**, reemplazándolas por métodos y clases en Python.
- **Facilitar la migración entre bases de datos**, ya que el código no depende de un motor específico.
- **Mejorar la seguridad**, evitando inyecciones SQL.
- **Hacer que el código sea más legible y mantenible**.

Instalación y Configuración de SQLAlchemy en Flask:

Para usar SQLAlchemy en una aplicación Flask, primero se debe instalar:

pip install flask-sqlalchemyLuego, se configura en la aplicación Flask:from flask import Flaskfrom flask_sqlalchemy import SQLAlchemyapp = Flask(__name__)app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db' # Base de datos SQLiteapp.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = Falsedb = SQLAlchemy(app)

Definición de Modelos y Relaciones entre Tablas:

En SQLAlchemy, las tablas de la base de datos se representan como **clases**.

Ejemplo de modelo de una tabla Usuario:

class Usuario(db.Model):   id = db.Column(db.Integer, primary_key=True)   nombre = db.Column(db.String(100), nullable=False)   email = db.Column(db.String(100), unique=True, nullable=False)   def __repr__(self):   return f"<Usuario {self.nombre}>"

Relaciones entre tablas (Uno a Muchos):

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    titulo = db.Column(db.String(200), nullable=False)
    usuario_id = db.Column(db.Integer, db.ForeignKey('usuario.id'), nullable=False)
    usuario = db.relationship('Usuario', backref='posts')
```

Aquí, **un usuario puede tener múltiples posts**.

Ejecución de Consultas y Operaciones CRUD

Crear la base de datos:

```
with app.app_context():
    db.create_all()
```

Operaciones CRUD:

- **Crear un usuario:**

```
nuevo_usuario = Usuario(nombre="Juan", email="juan@example.com")
db.session.add(nuevo_usuario)
db.session.commit()
```

- **Leer usuarios:**

```
usuarios = Usuario.query.all()
print(usuarios)
```

- **Actualizar un usuario:**

```
usuario = Usuario.query.get(1)
usuario.nombre = "Juan Pérez"
db.session.commit()
```

- **Eliminar un usuario:**

```
usuario = Usuario.query.get(1)
db.session.delete(usuario)
db.session.commit()
```

3.3. Django ORM y gestión de bases de datos

E l **Django ORM (Object-Relational Mapping)** es una herramienta integrada en Django que permite interactuar con bases de datos relacionales mediante código Python en lugar de escribir consultas SQL manualmente. Su uso facilita la creación, manipulación y consulta de bases de datos, asegurando un desarrollo más rápido y seguro. El **Django ORM** simplifica la gestión de bases de datos en Django, permitiendo definir modelos con clases de Python, sincronizar cambios mediante migraciones y realizar consultas avanzadas sin necesidad de SQL. Esto hace que el desarrollo sea más eficiente, seguro y escalable.

Definición de Modelos en Django ORM

En Django, las bases de datos se representan a través de **modelos**, que son clases de Python que definen la estructura de las tablas.

Ejemplo de modelo en Django:

```
from django.db import models

class Usuario(models.Model):
    nombre = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    fecha_registro = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.nombre
```

Explicación:

- CharField: Define un campo de texto con una longitud máxima.
- EmailField: Valida que los datos ingresados sean un correo electrónico.

- DateTimeField(auto_now_add=True): Guarda la fecha automáticamente cuando se crea un registro.
- __str__(): Método que devuelve una representación en texto del objeto.

Una vez definido el modelo, Django se encarga de generar la estructura en la base de datos sin necesidad de escribir SQL.

Migraciones y Sincronización con Bases de Datos

Cada vez que se define o modifica un modelo, se deben aplicar **migraciones** para sincronizar la estructura de la base de datos con los cambios en el código.

Pasos para aplicar migraciones en Django:

1. **Generar las migraciones:**

python manage.py makemigrations

2. **Aplicar las migraciones a la base de datos:**

python manage.py migrate

Esto crea o actualiza las tablas en la base de datos según la definición en los modelos.

Beneficios de las migraciones:

- Permiten modificar la estructura de la base de datos sin perder datos.
- Automatizan la gestión de cambios en los modelos.
- Facilitan la sincronización entre diferentes entornos de desarrollo.

Consultas Avanzadas con el ORM de Django

Django ORM permite realizar consultas de manera sencilla sin escribir SQL manualmente.

Ejemplo de consultas básicas:

- Obtener todos los usuarios:

```
usuarios = Usuario.objects.all()
```

- Filtrar por email:

```
usuario = Usuario.objects.filter(email="usuario@example.com").first()
```

- Obtener usuarios registrados después de una fecha específica:

```
from django.utils.timezone import now

usuarios_recientes = Usuario.objects.filter(fecha_registro__gte=now() - timedelta(days=30))
```

Consultas avanzadas:

- Ordenar resultados:

```
usuarios_ordenados = Usuario.objects.order_by('-fecha_registro')
```

- Contar registros:

```
total_usuarios = Usuario.objects.count()
```

- Relacionar modelos con **ForeignKey**:

```
class Pedido(models.Model):
  usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE)
   total = models.DecimalField(max_digits=10, decimal_places=2)
```

Esto permite obtener los pedidos de un usuario con:

```
pedidos = Pedido.objects.filter(usuario=usuario)
```

4. Manipulación de Datos con Pandas

4.1. Utilización de pandas

pandas es una biblioteca de Python diseñada para la manipulación y análisis de datos de manera eficiente y estructurada. Es ampliamente utilizada en ciencia de datos, aprendizaje automático y desarrollo de aplicaciones que requieren procesamiento de grandes volúmenes de información. pandas es una herramienta esencial para la manipulación de datos en Python, proporcionando una forma eficiente de gestionar información estructurada. Su facilidad de uso y compatibilidad con otras bibliotecas lo convierten en una opción ideal para análisis y procesamiento de datos en múltiples aplicaciones.

¿Qué es pandas y por qué es útil en la manipulación de datos?:

pandas proporciona estructuras de datos flexibles y herramientas para el manejo de datos tabulares, lo que facilita la manipulación, limpieza y análisis. Sus principales características incluyen:

- **Estructura de datos eficiente:** Utiliza **DataFrames** y **Series**, similares a las tablas de bases de datos o las hojas de cálculo de Excel.
- **Lectura y escritura de datos:** Permite importar y exportar archivos en múltiples formatos como **CSV, JSON, Excel y bases de datos SQL**.
- **Operaciones avanzadas:** Soporta filtrado, agrupamiento, transformación y agregación de datos.
- **Integración con otras bibliotecas:** Compatible con **NumPy, Matplotlib y scikit-learn**, lo que facilita su uso en análisis de datos y aprendizaje automático.

Instalación y Configuración de pandas en un Entorno Python:

Para instalar pandas en un entorno de Python, se usa el siguiente comando:

```
pip install pandas
```

Después de la instalación, se importa en un script de Python de la siguiente manera:

```
import pandas as pd
```

Para verificar que pandas está instalado correctamente, se puede ejecutar:

```
print(pd.__version__)
```

Si se necesita usar pandas en **Jupyter Notebook**, se recomienda instalarlo junto con las herramientas necesarias:

```
pip install jupyter pandas
```

4.2. Operaciones básicas con pandas

pandas es una biblioteca de Python que permite manipular datos de forma eficiente mediante estructuras como **DataFrames** y **Series**. Sus herramientas facilitan la carga, limpieza y análisis de datos provenientes de distintos formatos, como **CSV, JSON y bases de datos SQL** . Las operaciones básicas de **carga, limpieza y filtrado de datos** en pandas permiten gestionar grandes volúmenes de información de forma sencilla y eficiente. Gracias a su integración con archivos y bases de datos, es una herramienta esencial en análisis de datos y aplicaciones basadas en Python.

Carga y Lectura de Datos desde Archivos CSV, JSON y Bases de Datos:

pandas permite importar datos desde distintos formatos de archivo y bases de datos.

- **Leer datos desde un archivo CSV:**

```
import pandas as pd
df = pd.read_csv("datos.csv")
print(df.head())  # Muestra las primeras filas
```

- **Leer datos desde un archivo JSON:**

```
df_json = pd.read_json("datos.json")
```

- **Cargar datos desde una base de datos SQL:**

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///datos.db')  # Conexión a SQLite
df_sql = pd.read_sql("SELECT * FROM usuarios", con=engine)
```

Beneficios:

- Facilita la carga de datos desde diversas fuentes.
- Permite trabajar con grandes volúmenes de datos sin problemas de rendimiento.

Transformación y Limpieza de Datos:

Después de cargar los datos, es común realizar transformaciones para limpiarlos y prepararlos para el análisis.

- **Eliminar valores nulos:**

```
df = df.dropna()  # Elimina filas con valores nulos
```

- **Rellenar valores faltantes:**

```
df["edad"].fillna(df["edad"].mean(), inplace=True)  # ReemplazaNaN con la media
```

- **Renombrar columnas:**


```
df.rename(columns={"nombre": "Nombre_Completo"}, inplace=True)
```

- **Convertir tipos de datos:**

```
df["fecha"] = pd.to_datetime(df["fecha"])  # Convertir a formato fecha
```

Beneficios:

- Permite estandarizar datos para su análisis.
- Facilita la detección de inconsistencias y errores.

Selección y Filtrado de Información:

pandas ofrece métodos eficientes para seleccionar y filtrar datos dentro de un DataFrame.

- **Seleccionar columnas específicas:**

```
df_seleccion = df[["Nombre_Completo", "edad"]]
```

- **Filtrar datos según una condición:**

```
df_filtrado = df[df["edad"] > 30]  # Filtra registros donde edad > 30
```

- **Aplicar múltiples condiciones:**

```
df_filtrado = df[(df["edad"] > 30) & (df["ciudad"] == "Madrid")]
```

- **Ordenar datos:**

```
df_ordenado = df.sort_values(by="edad", ascending=False)
```

Beneficios:

- Facilita la extracción de datos relevantes sin modificar la estructura original.
- Mejora la eficiencia en el análisis de datos grandes.

4.3. Análisis de datos en una API

El análisis de datos en una API es una práctica común cuando se necesita procesar información obtenida de fuentes externas. **pandas**, una de las bibliotecas más potentes de Python, facilita la manipulación y análisis de estos datos al convertirlos en **DataFrames** para su posterior procesamiento, filtrado y generación de reportes. El uso de **pandas** para analizar datos de una API permite convertir información en **JSON a DataFrames**, procesar grandes volúmenes de datos y generar **estadísticas y reportes** de manera eficiente. Esto facilita la toma de decisiones basada en datos en múltiples aplicaciones.

Uso de pandas para Procesar Datos Obtenidos de una API:

Las APIs suelen devolver datos en **formato JSON**, lo que hace que pandas sea una herramienta ideal para estructurar y analizar esa información. Una API puede proporcionar datos financieros, meteorológicos, de redes sociales, entre otros, y pandas permite **transformar, limpiar y extraer información relevante**.

Para obtener datos desde una API y procesarlos con pandas, primero se usa la biblioteca **requests** para realizar la solicitud:

```
import requests
import pandas as pd

url = "https://api.example.com/datos"
respuesta = requests.get(url)

if respuesta.status_code == 200:
    datos_json = respuesta.json()
```

Aquí, los datos de la API son almacenados en formato JSON y listos para ser convertidos en un **DataFrame**.

Conversión de Datos en Formato JSON a DataFrames:

Una vez obtenidos los datos en formato JSON, se convierten en un **DataFrame de pandas** para facilitar su manipulación:

```
df = pd.DataFrame(datos_json)
print(df.head())  # Muestra las primeras filas del DataFrame
```

Si la API devuelve un JSON con estructuras anidadas, pandas permite normalizarlos:

```
from pandas import json_normalize

df = json_normalize(datos_json, "registros", ["id", "fecha"])
```

Esto permite manejar datos estructurados de manera más organizada dentro de un DataFrame.

Generación de Estadísticas y Reportes con pandas:

Una vez que los datos están en un DataFrame, se pueden generar estadísticas clave:

- **Resumen estadístico:**

```
print(df.describe())  # Muestra estadísticas como media, desviación estándar y percentiles
```

- **Conteo de valores únicos:**

```
print(df["categoria"].value_counts())  # Agrupar datos por categoría
```

- **Filtrado y análisis de datos específicos:**

```
df_filtrado = df[df["valor"] > 100]  # Filtrar registros donde el valor es mayor a 100
```

- **Exportar el reporte a CSV:**

```
df.to_csv("reporte.csv", index=False)
```

Esto facilita compartir los datos procesados con otros equipos o integrarlos en herramientas externas.

4.4. Integración de pandas con Flask y Django

La integración de **pandas** con frameworks como **Flask** y **Django** permite manipular y analizar datos dentro de APIs y aplicaciones web. pandas facilita la **transformación de datos** en controladores antes de enviarlos a la vista y la **exportación de información en formatos estructurados** como CSV, Excel y JSON. La integración de **pandas con Flask y Django** permite transformar datos antes de enviarlos a la vista y facilita la exportación de información en **CSV, Excel y JSON**. Esto mejora la flexibilidad de las APIs y la accesibilidad de los datos en distintos formatos.

Uso de pandas en Controladores para Transformar Datos Antes de Enviarlos a la Vista:

En una API basada en **Flask** o **Django**, los datos obtenidos de una base de datos o una API externa pueden requerir procesamiento antes de enviarse al cliente. **pandas** permite realizar transformaciones como filtrado, agregaciones y limpieza antes de que los datos sean renderizados o enviados como respuesta.

Ejemplo en Flask:

from flask import Flask, jsonifyimport pandas as pdapp = Flask(__name__)@app.route('/datos')def procesar_datos(): # Simulación de datos obtenidos de una API o base de datos datos = [{"id": 1, "nombre": "Juan", "edad": 25}, {"id": 2, "nombre": "María", "edad": 30}, {"id": 3, "nombre": "Carlos", "edad": 22}] # Convertir a DataFrame y procesar df = pd.DataFrame(datos) df["grupo_edad"] = df["edad"].apply(lambda x: "Joven" if x < 30 else "Adulto") return jsonify(df.to_dict(orient="records"))if __name__ == '__main__': app.run(debug=True)

Aquí, pandas se usa para **crear un nuevo campo** (grupo_edad) antes de enviar los datos procesados en formato JSON a la API.

Exportación de Datos en Formatos Estructurados (CSV, Excel, JSON) desde una API:

Muchas aplicaciones web requieren que los usuarios descarguen datos en formatos como **CSV, Excel o JSON**. pandas permite exportar datos procesados en estos formatos para facilitar su uso en otras herramientas.

Ejemplo en Django:

Exportar Datos en CSV y JSON

import pandas as pdfrom django.http import HttpResponse, JsonResponsefrom .models import Usuariodef exportar_csv(request): usuarios = Usuario.objects.all().values() df = pd.DataFrame(list(usuarios)) response = HttpResponse(content_type="text/csv") response["Content-Disposition"] = "attachment; filename=usuarios.csv" df.to_csv(path_or_buf=response, index=False) return responsedef exportar_json(request): usuarios = Usuario.objects.all().values() df = pd.DataFrame(list(usuarios)) return JsonResponse(df.to_dict(orient="records"), safe=False)

Explicación:

- exportar_csv(): Genera un archivo CSV descargable con los datos de la base de datos.
- exportar_json(): Convierte los datos a JSON y los devuelve en una respuesta HTTP.

Bibliografía y lecturas recomendadas:

- El tutorial de Python: https://docs.python.org/es/3/tutorial/
- Introducción a la Programación: https://mpru.github.io/introprog/introducci%C3%B3n-a-la-programaci%C3%B3n.html
- W3Schools (Python Tutorial)'
- GeeksforGeeks (Python Programming Language
- Python Crash Course" per Eric Matthes
- Automate the Boring Stuff with Python" per Al Sweigart

Actividades prácticas

Ejercicio 15. Diseño de una API REST para un Sistema de Reservas de Hoteles

Una cadena hotelera necesita desarrollar una API REST que permita a los clientes realizar reservas en sus hoteles de forma eficiente. El equipo de desarrollo ha decidido diseñar la API utilizando Flask o Django REST Framework (DRF) y gestionar los datos con una base de datos SQL. Además, la API deberá incluir funcionalidades para la exportación y análisis de datos mediante pandas.

1. Tareas a Resolver:

   Diseñar la estructura de la API:

   - Identificar los recursos principales que se gestionarán (habitaciones, reservas, clientes).
   - Definir los endpoints y métodos HTTP necesarios.
   - Determinar la base de datos a utilizar y si se aplicará un ORM.

   Definir el modelo de datos:

   - ¿Qué tablas y relaciones serán necesarias para almacenar reservas y disponibilidad de habitaciones?
   - ¿Cómo se garantizará la consistencia y escalabilidad del sistema?

   Exportación y análisis de datos con pandas:

   - ¿Qué métricas podrían analizarse a partir de los datos de la API?
   - ¿En qué formatos deberían exportarse los datos para su análisis (CSV, JSON, Excel)?

   Argumentar cómo se manejarán los siguientes aspectos:

   - Seguridad y autenticación de la API.
   - Manejo de errores y validación de datos.
   - Pruebas y documentación de la API.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos

Ejercicio 16. Diseño de una API REST para un Sistema de Gestión de Inventarios en un Almacén

Una empresa de logística necesita una API REST para gestionar su inventario de productos en múltiples almacenes. El equipo de desarrollo ha decidido diseñar la API utilizando Flask o Django REST Framework (DRF) con una base de datos SQL. Además, la API deberá ofrecer opciones de exportación de datos en CSV y JSON para su posterior análisis mediante pandas.

1. Tareas a Resolver:

Diseñar la estructura de la API:

- Identificar los recursos principales a gestionar (productos, almacenes, stock).
- Definir los endpoints y métodos HTTP.
- Seleccionar la base de datos y el ORM adecuado.

Definir el modelo de datos:

- Diseñar las tablas necesarias y sus relaciones.
- Asegurar que los datos sean consistentes y escalables.

Exportación y análisis de datos con pandas:

- ¿Qué métricas deberían analizarse (inventario disponible, productos más vendidos, etc.)?
- ¿En qué formatos deben exportarse los datos?

Argumentar las estrategias de seguridad, validación de datos y documentación de la API.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos