Universidad de los Andes | Educación Continua
Vicerrectoría Académica

Introducción al Backend y Arquitectura Cliente-Servidor

Curso: Desarrollo Backend con Django

Bases de datos con Python y Entornos Virtuales

Agenda

1

Fundamentos de Entornos Virtuales

Qué son, por qué usarlos y cómo implementarlos en tus proyectos

2

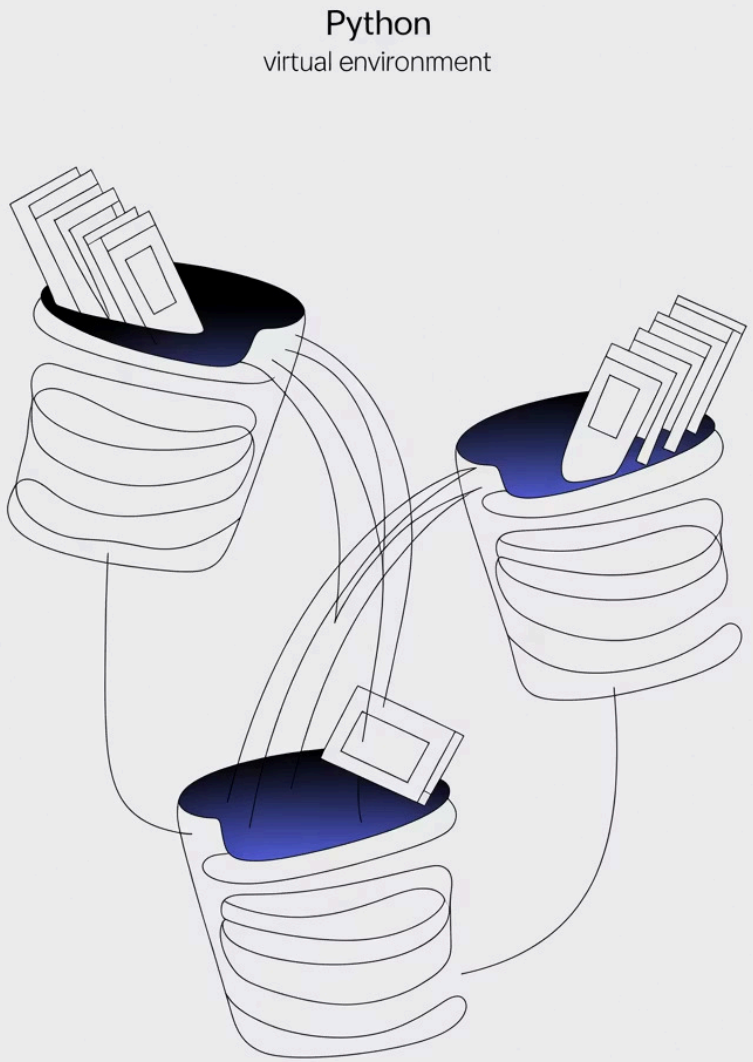
Conexión a Bases de Datos

SQLite3 y MySQL desde Python

3

Operaciones CRUD

Create, Read, Update y Delete con código Python



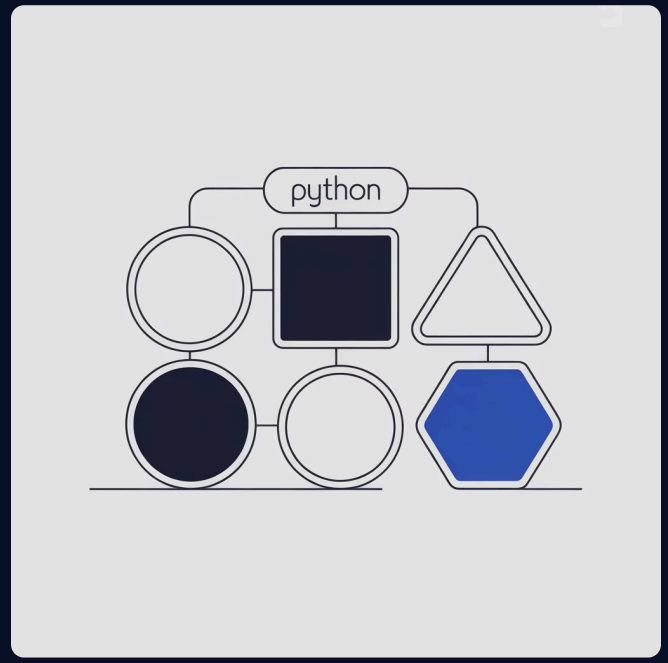
Entornos Virtuales en Python

Aislamiento y gestión de dependencias para proyectos profesionales

¿Qué son los Entornos Virtuales?

Un entorno virtual es un espacio aislado donde puedes instalar paquetes Python sin afectar a otros proyectos o al sistema.

- Cada proyecto puede tener su propio conjunto de dependencias
- Evita conflictos entre versiones de paquetes
- Facilita la reproducibilidad del proyecto
- Permite tener un archivo de requisitos (requirements.txt)



¿Por qué usar Entornos Virtuales?

Aislamiento

Cada proyecto puede tener sus propias dependencias sin interferir con otros

Reproducibilidad

Cualquier miembro del equipo puede recrear exactamente el mismo entorno

Seguridad

Evita actualizaciones accidentales que puedan romper compatibilidad

Portabilidad

Facilita mover el proyecto entre diferentes computadoras y servidores

Los entornos virtuales son **esenciales para el desarrollo profesional** y son altamente valorados en el mercado laboral colombiano.

Módulos para Entornos Virtuales

venv

Módulo integrado en Python para crear entornos virtuales ligeros

`python -m venv nombre_entorno`

virtualenv

Herramienta de terceros con más funcionalidades, compatible con Python 2 y 3

`pip install virtualenv`

`virtualenv nombre_entorno`

conda

Sistema de gestión de paquetes y entornos, popular en ciencia de datos

`conda create --name nombre_entorno`

En este curso nos enfocaremos en **venv**, el estándar oficial de Python.

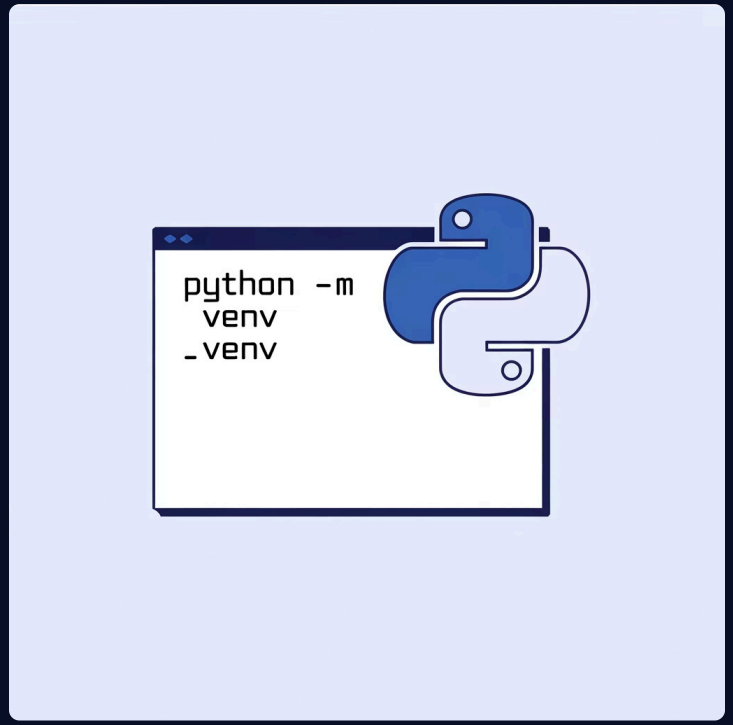
Creación de un Entorno Virtual

Para crear un entorno virtual con `venv`:

```
# En Windows
python -m venv .venv

# En Linux/macOS
python3 -m venv .venv
```

Esto creará una carpeta `.venv` en tu directorio actual con todos los archivos necesarios para el entorno virtual.



Usar `.venv` como nombre es una convención común en proyectos Python modernos. El punto inicial hace que sea una carpeta oculta en sistemas Unix.

Activación del Entorno Virtual

Windows (CMD)

```
.venv\Scripts\activate
```

Linux/macOS (Bash/Zsh)

```
source .venv/bin/activate
```

Windows (PowerShell)

```
.venv\Scripts\Activate.ps1
```

Verificación

```
which python # (Linux/macOS)
where python # (Windows)
```

Cuando el entorno está activo, verás `(.venv)` al inicio de tu prompt:

```
(.venv) PS C:\Users\usuario\mi_proyecto>
```

Instalación de Paquetes con pip

Una vez activado el entorno virtual, puedes instalar paquetes usando pip:

```
# Instalar un paquete específico
pip install mysql-connector-python

# Instalar varios paquetes
pip install requests pandas matplotlib

# Especificar versión
pip install flask==2.0.1
```

Los paquetes instalados son locales al entorno:

```
# Ver paquetes instalados
pip list

# Guardar dependencias
pip freeze > requirements.txt

# Instalar desde archivo
pip install -r requirements.txt
```

Desactivación y Eliminación

Desactivar el entorno

Para salir del entorno virtual, simplemente ejecuta:

```
deactivate
```

El prefijo (.venv) desaparecerá del prompt, indicando que has vuelto al entorno global de Python.

Eliminar el entorno

Si necesitas eliminar completamente un entorno virtual:

1. Asegúrate de que esté desactivado
2. Simplemente borra la carpeta del entorno:

```
# En Windows
rmdir /s /q .venv

# En Linux/macOS
rm -rf .venv
```

¡Cuidado! Eliminar el entorno borra todas las dependencias instaladas. Asegúrate de tener un requirements.txt actualizado.

Conexión a Bases de Datos

Integrando Python con SQLite y MySQL para aplicaciones reales

¿Por qué Conectar Python con Bases de Datos?

Persistencia

Almacenar datos de forma permanente entre ejecuciones del programa

Eficiencia

Optimización de consultas y almacenamiento para grandes volúmenes de datos

Integridad

Garantizar consistencia de datos mediante restricciones y transacciones

Escalabilidad

Capacidad para crecer y manejar más usuarios y datos

Las empresas colombianas buscan desarrolladores que puedan **integrar aplicaciones con bases de datos existentes** de forma eficiente y segura.

SQLite vs MySQL: ¿Cuál elegir?

SQLite

- Base de datos de archivo único
- No requiere servidor
- Ideal para aplicaciones pequeñas/medianas
- Perfecta para desarrollo y pruebas
- Incluida en la biblioteca estándar de Python
- Hasta ~1TB de datos

MySQL

- Sistema cliente-servidor
- Mayor rendimiento en concurrencia
- Mejor para aplicaciones web/empresariales
- Requiere instalación y configuración
- Necesita paquete adicional en Python
- Prácticamente ilimitado

Ambas son SQL y comparten sintaxis similar para consultas, pero difieren en implementación y casos de uso.

Conexión a SQLite con Python

```
import sqlite3

# Crear conexión (si la BD no existe, se crea)
conexion = sqlite3.connect('mi_base_datos.db')

# Crear un cursor para ejecutar comandos SQL
cursor = conexion.cursor()

# Crear una tabla
cursor.execute('''
CREATE TABLE IF NOT EXISTS usuarios (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    email TEXT UNIQUE,
    edad INTEGER
)
''')

# Guardar cambios y cerrar
conexion.commit()
conexion.close()
```

SQLite usa un archivo local para almacenar toda la base de datos, lo que lo hace ideal para comenzar.

Conexión a MySQL con Python

```
import mysql.connector

# Crear conexión
conexion = mysql.connector.connect(
    host="localhost",      # o dirección IP
    user="usuario",
    password="contraseña",
    database="mi_base_datos"
)

# Crear cursor
cursor = conexion.cursor()

# Ejecutar consulta
cursor.execute('''
CREATE TABLE IF NOT EXISTS usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    edad INT
)
''')

# Guardar y cerrar
conexion.commit()
conexion.close()
```

Requisitos previos:

1. Servidor MySQL instalado y funcionando
2. Credenciales de acceso válidas
3. Base de datos creada
4. Paquete instalado en el entorno virtual:

```
pip install mysql-connector-python
```

Manejo de Excepciones en Conexiones

```
import sqlite3

try:
    # Intentar conectar a la base de datos
    conexion = sqlite3.connect('mi_base_datos.db')
    cursor = conexion.cursor()

    # Operaciones con la base de datos...
    cursor.execute("SELECT * FROM tabla_que_no_existe")

except sqlite3.OperationalError as e:
    print(f"Error de operación: {e}")

except sqlite3.DatabaseError as e:
    print(f"Error de base de datos: {e}")

finally:
    # Este bloque siempre se ejecuta
    if 'conexion' in locals():
        conexion.close()
        print("Conexión cerrada")
```

Es **crucial manejar excepciones** para prevenir fallos en las aplicaciones cuando ocurren problemas con la base de datos.

Uso de Cursores para Consultas

¿Qué es un cursor?

Un cursor es un objeto que permite ejecutar comandos SQL y recuperar resultados fila por fila.

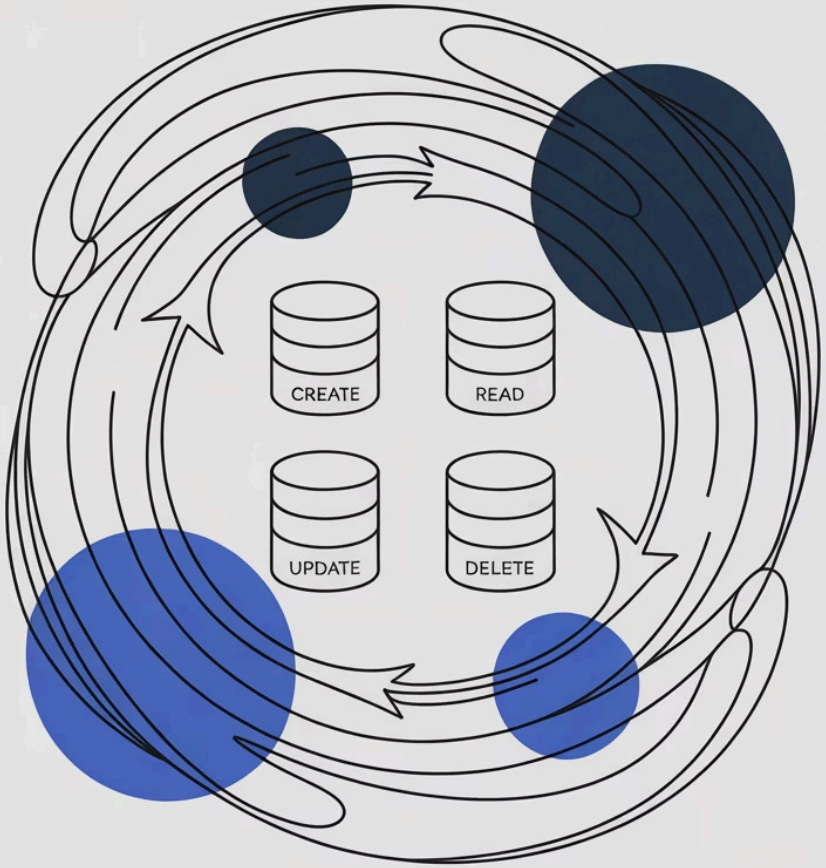
Ventajas:

- Control sobre la ejecución de consultas
- Manejo eficiente de grandes conjuntos de resultados
- Prevención de inyección SQL con parámetros
- Manejo de transacciones

Métodos principales:

- `cursor.execute()`: Ejecuta una consulta
- `cursor.executemany()`: Ejecuta una consulta con múltiples conjuntos de datos
- `cursor.fetchone()`: Obtiene la siguiente fila
- `cursor.fetchall()`: Obtiene todas las filas
- `cursor.fetchmany(size)`: Obtiene un número específico de filas
- `cursor.rowcount`: Número de filas afectadas

| Database



Operaciones CRUD

Create, Read, Update, Delete - El fundamento de la persistencia de datos

¿Qué es CRUD?

Create (Crear)

Insertar nuevos registros en la base de datos mediante INSERT

Read (Leer)

Consultar datos existentes mediante SELECT

Delete (Eliminar)

Borrar registros mediante DELETE

Update (Actualizar)

Modificar registros existentes mediante UPDATE

Estas cuatro operaciones básicas son el **fundamento de cualquier aplicación** que interactúe con bases de datos.

CREATE: Inserción de Datos

```
import mysql.connector

# Establecer la conexión con la base de datos MySQL
conexion = mysql.connector.connect(
    host='localhost',
    user='tu_usuario',
    password='tu_contraseña',
    database='biblioteca'
)
cursor = conexion.cursor()

# Insertar un solo registro
cursor.execute('''
INSERT INTO usuarios (nombre, correo, telefono)
VALUES (%s, %s, %s)
''', ('Juan Pérez', 'juan@ejemplo.co', 12345))

# Insertar múltiples registros
usuarios = [
    ('María López', 'maria@ejemplo.co', 344852),
    ('Carlos Gómez', 'carlos@ejemplo.co', 222321),
    ('Ana Torres', 'ana@ejemplo.co', 29456634)
]

cursor.executemany('''
INSERT INTO usuarios (nombre, correo, telefono)
VALUES (%s, %s, %s)
''', usuarios)

conexion.commit()
conexion.close()
```

Consejos importantes:

- Usa **parámetros con placeholders (%s, %s, %s)** para prevenir inyección SQL
- Usa `executemany()` para inserciones masivas
- Siempre haz `commit()` para guardar cambios
- Con MySQL, usa `%s` como placeholder

Última fila insertada:

```
print(f"ID generado: {cursor.lastrowid}")
```

READ: Consulta de Datos

```
import mysql.connector

conexion = mysql.connector.connect(
    host='localhost',
    user='tu_usuario',
    password='tu_contraseña',
    database='biblioteca'
)
cursor = conexion.cursor()

# Consulta básica - seleccionar todo
cursor.execute("SELECT * FROM usuarios")

# Recuperar todos los resultados
todos_usuarios = cursor.fetchall()
for usuario in todos_usuarios:
    print(usuario)  # Tupla con los datos

# Consulta con condiciones
cursor.execute("SELECT nombre, correo FROM usuarios")

# Recuperar un solo resultado
primer_usuario = cursor.fetchone()
print("Primer usuario: ", primer_usuario)

conexion.close()
```

Los resultados se devuelven como **tuplas** por defecto, donde cada valor corresponde a una columna.

Consultas con Diccionarios

```
import mysql.connector

conexion = mysql.connector.connect(
    host='localhost',
    user='tu_usuario',
    password='tu_contraseña',
    database='biblioteca'
)
# Configurar para que devuelva diccionarios
cursor = conexion.cursor(dictionary=True)

# Ejecutar consulta
cursor.execute('''
SELECT id, nombre, correo
FROM usuarios
''')

# Procesar resultados como diccionarios
for fila in cursor.fetchall():
    # Acceso por nombre de columna
    print(f"ID: {fila['id']}")
    print(f"Nombre: {fila['nombre']}")
    print(f"Email: {fila['correo']}")
    print("--------------------")

conexion.close()
```

Ventajas de usar diccionarios:

- Acceso a campos por nombre en lugar de posición
- Código más legible y mantenible
- Menos propenso a errores si cambia el orden de las columnas
- Facilita el trabajo con frameworks web como Flask o Django

En MySQL con `mysql-connector-python`, usa:

```
cursor = conexion.cursor(dictionary=True)
```

UPDATE: Actualización de Datos

```
import mysql.connector

conexion = mysql.connector.connect(
    host='localhost',
    user='tu_usuario',
    password='tu_contraseña',
    database='biblioteca'
)
cursor = conexion.cursor()

# Actualizar un solo registro
cursor.execute('''
UPDATE usuarios
SET correo = %s, telefono = %s
WHERE nombre = %s
''', ('juan.nuevo@ejemplo.co', 123456789, 'Juan Pérez'))

# Verificar cuántas filas se modificaron
filas_modificadas = cursor.rowcount
print(f"Filas modificadas: {filas_modificadas}")

conexion.commit()
conexion.close()
```

Buenas prácticas:

- Utiliza **WHERE** para evitar actualizar toda la tabla por error
- Verifica las filas afectadas con `rowcount`
- Siempre usa parámetros para valores variables
- Realiza actualizaciones dentro de transacciones para poder revertir en caso de error

Transacciones explícitas:

```
try:
 cursor.execute("BEGIN TRANSACTION")
 # Operaciones...
 conexion.commit()
except:
 conexion.rollback()
 raise
```

DELETE: Eliminación de Datos

```
import mysql.connector

conexion = mysql.connector.connect(
    host='localhost',
    user='tu_usuario',
    password='tu_contraseña',
    database='biblioteca'
)
cursor = conexion.cursor()

# Eliminar un registro específico
cursor.execute('''
DELETE FROM usuarios
WHERE id = %s
''', (9,))

# Eliminar múltiples registros
cursor.execute('''
DELETE FROM libros
WHERE anio_publicacion > %s
''', (2000,))

# Verificar cuántas filas se eliminaron
filas_eliminadas = cursor.rowcount
print(f"Filas eliminadas: {filas_eliminadas}")

conexion.commit()
conexion.close()
```

¡PRECAUCIÓN!

Nunca ejecutes `DELETE FROM tabla` sin una cláusula `WHERE`, a menos que realmente quieras eliminar todos los registros.

Alternativas a DELETE:

- **Soft Delete**: Añadir un campo "activo" o "eliminado" y actualizarlo en lugar de eliminar
- **Archivar datos**: Mover registros a una tabla de históricos antes de eliminarlos
- Usar transacciones para poder deshacer eliminaciones accidentales

Prevención de Inyección SQL

Incorrecto (peligroso):

```
# ¡NUNCA HAGAS ESTO!
nombre = "Robert'); DROP TABLE usuarios; --"
query = f"SELECT * FROM usuarios WHERE nombre = '{nombre}'"
cursor.execute(query)  # ¡PELIGRO!
```

Esto podría eliminar toda la tabla "usuarios"

Correcto (seguro):

```
# Siempre usa parámetros
nombre = "Robert'; DROP TABLE usuarios; --"
cursor.execute(
    "SELECT * FROM usuarios WHERE nombre = %s",
    (nombre,)
)  # Seguro
```

Los parámetros aseguran que los datos se traten como valores, no como código SQL

⚠ La inyección SQL es una de las vulnerabilidades más comunes y peligrosas. Nunca construyas consultas concatenando strings con datos de usuario.

Errores Comunes y Soluciones

OperationalError: no such table

La tabla no existe en la base de datos.

Solución: Verificar la ejecución del script que crea las tablas y comprobar el nombre de la base de datos.

IntegrityError: UNIQUE constraint failed

Intento de insertar un valor duplicado en un campo único.

Solución: Verificar si el registro ya existe antes de insertarlo o actualizar en lugar de insertar.

ProgrammingError: table already exists

Intento de crear una tabla que ya existe.

Solución: Usar `CREATE TABLE IF NOT EXISTS` o verificar antes si la tabla existe.

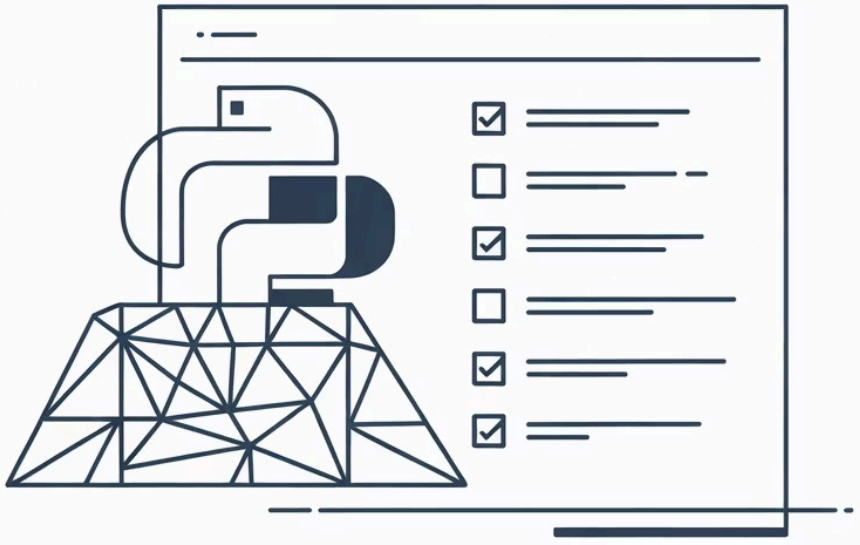
Error: no such module 'mysql.connector'

El paquete no está instalado en el entorno virtual.

Solución: Asegurarse de tener el entorno virtual activado y ejecutar `pip install mysql-connector-python`.

La mayoría de los errores se pueden resolver con un buen manejo de excepciones y validación previa de los datos.

Mejores Prácticas: Entornos Virtuales y Bases de Datos



Entornos Virtuales

- Nombrar con `.venv` o nombre descriptivo del proyecto
- Incluir siempre un `requirements.txt`
- No incluir la carpeta del entorno en control de versiones (agregar a `.gitignore`)
- Documentar la versión de Python utilizada
- Usar `pip-tools` para gestionar dependencias complejas

Bases de Datos

- Usar parámetros en consultas, nunca concatenar strings
- Cerrar conexiones explícitamente (`finally`)
- Manejar transacciones para operaciones múltiples
- Validar datos antes de enviar a la base de datos
- Evitar recuperar más datos de los necesarios
- Usar índices para consultas frecuentes

Pregunta de Reflexión

> ## ¿Qué ventajas ofrece usar un entorno virtual al trabajar con proyectos que requieren conexión a diferentes tipos de bases de datos?

Aislamiento de dependencias

Cada proyecto puede tener su propia versión de los conectores de bases de datos sin conflictos.

Seguridad

Las credenciales de conexión pueden manejarse de forma aislada para cada proyecto.

Portabilidad

Facilita migrar el proyecto a diferentes entornos o servidores manteniendo las mismas configuraciones.

Los entornos virtuales combinados con la capacidad de Python para conectarse a múltiples bases de datos crean un **ecosistema flexible y potente para el desarrollo de aplicaciones** en el contexto colombiano.

¡Gracias por tu atención!

¿Preguntas?

Universidad de los Andes | Educación Continua
Vicerrectoría Académica

#EDCOUNIANDES

https://educacioncontinua.uniandes.edu.co/

Contacto: educacion.continua@uniandes.edu.co

Universidad de los Andes | Vigilada Mineducación. Reconocimiento como Universidad: Decreto 1297 del 30 de mayo de 1964. Reconocimiento personería jurídica: Resolución 28 del 23 de febrero de 1949 MinJusticia.