Tema 4:

Gestión de entornos y librerías (6h)

4. Gestión de entornos y librerías (6h)

4.1. Concepto de entornos globales y virtuales

4.2. Creación y uso de entornos virtuales (venv, virtualenv)

4.3. Herramientas avanzadas: conda, pipenv, poetry

4.4. Instalación y actualización de librerías con pip

4.5. Buenas prácticas de gestión de dependencias

4.6. Parte práctica (ejemplos y ejercicios)

4.7. Distribución sugerida del tiempo

Profesor: Salvador Martínez Bolinches

Centro: IES Font de Sant Lluís

Año: 2025

Objetivos de aprendizaje

- Comprender la necesidad de aislar dependencias.
- Crear, activar y desactivar entornos virtuales.
- Gestionar dependencias con `pip` y `requirements.txt`.
- Conocer herramientas adicionales (venv, virtualenv, conda, poetry, pipenv).
- Aplicar buenas prácticas en proyectos Python.

4.1. Concepto de entornos globales y virtuales

En el ecosistema de Python, la gestión de dependencias y paquetes constituye un aspecto fundamental para el desarrollo de aplicaciones. Existen dos enfoques principales: el uso del **entorno global** y la creación de **entornos virtuales**. Ambos cumplen la función de permitir la instalación y ejecución de librerías, pero presentan diferencias notables en su alcance, control y aplicabilidad.

Entorno global

Características

El entorno global corresponde a la instalación principal de Python en el sistema operativo. Todas las librerías que se instalan mediante `pip` se registran de manera centralizada y quedan disponibles para cualquier proyecto.

Ejemplo:

```
pip install numpy
```

En este caso, el paquete `numpy` será accesible desde cualquier script en Python, independientemente de la carpeta de trabajo.

Ventajas

1. **Disponibilidad universal**: Los paquetes se instalan una sola vez y están disponibles para todos los proyectos.
2. **Simplicidad inicial**: Para principiantes, resulta más sencillo trabajar directamente en el entorno global sin gestionar entornos adicionales.
3. **Menor consumo de espacio**: Al no duplicar librerías para cada proyecto, se evita la redundancia de archivos en disco.

Desventajas

1. **Conflictos de versiones**: Si un proyecto requiere una versión de una librería distinta a la de otro, el entorno global genera incompatibilidades.
2. **Difícil replicabilidad**: Compartir un proyecto con otros desarrolladores o migrarlo a un servidor puede producir errores si no se documenta con precisión la versión de cada dependencia.
3. **Riesgo de inestabilidad**: Una actualización o desinstalación de paquetes puede afectar negativamente a proyectos previamente funcionales.

Analogía visual:

El entorno global puede compararse con una cocina compartida donde todos los proyectos utilizan los mismos ingredientes, corriendo el riesgo de confusión.

El entorno virtual, en cambio, se asemeja a cocinas privadas, cada una con su propia despensa organizada para garantizar que las recetas salgan siempre iguales.

Entornos virtuales

Características

Un entorno virtual es un espacio aislado dentro del sistema de archivos que contiene su propio intérprete de Python y un directorio específico de librerías. Se crea mediante herramientas como `venv` o `virtualenv`.

Ejemplo:

```
python -m venv mi_entorno
mi_entorno\Scripts\activate
pip install numpy==1.21
```

En este escenario, la versión de `numpy` instalada queda restringida únicamente al proyecto, sin interferir con el sistema global.

Ventajas

1. **Aislamiento de dependencias**: Cada proyecto mantiene versiones independientes de librerías, evitando conflictos entre proyectos.
2. **Reproducibilidad**: Al generar un archivo `requirements.txt`, es posible replicar el entorno exacto en otro equipo o servidor:

```
pip freeze > requirements.txt
pip install -r requirements.txt
```

Mejor práctica profesional: Facilita la colaboración en equipos de desarrollo y es estándar en proyectos de investigación y producción.

1. **Seguridad en actualizaciones**: Permite probar nuevas versiones de paquetes sin afectar a proyectos previos.

Desventajas

1. **Mayor consumo de espacio**: Cada entorno duplica las librerías necesarias, lo que puede ser costoso en proyectos numerosos.
2. **Curva de aprendizaje**: Para usuarios principiantes puede resultar más complejo comprender y manipular la activación y desactivación de entornos.
3. **Mantenimiento adicional**: Es necesario gestionar múltiples entornos, lo cual puede generar cierta sobrecarga administrativa.

Conclusión

El uso de entornos virtuales en Python constituye una práctica recomendada en la mayoría de los casos, especialmente en contextos académicos y profesionales donde la **reproducibilidad**, la **colaboración** y la **estabilidad** son requisitos esenciales. El entorno global resulta útil en fases de aprendizaje o para tareas simples, pero a medida que los proyectos se vuelven más complejos, los entornos virtuales ofrecen un marco más robusto y controlado.

4.2. Creación y uso de entornos virtuales (venv, virtualenv)

Con venv (incluido en Python 3)

Windows

1. Crear el entorno virtual

```
python -m venv mi_entorno
```

2. Activar el entorno virtual

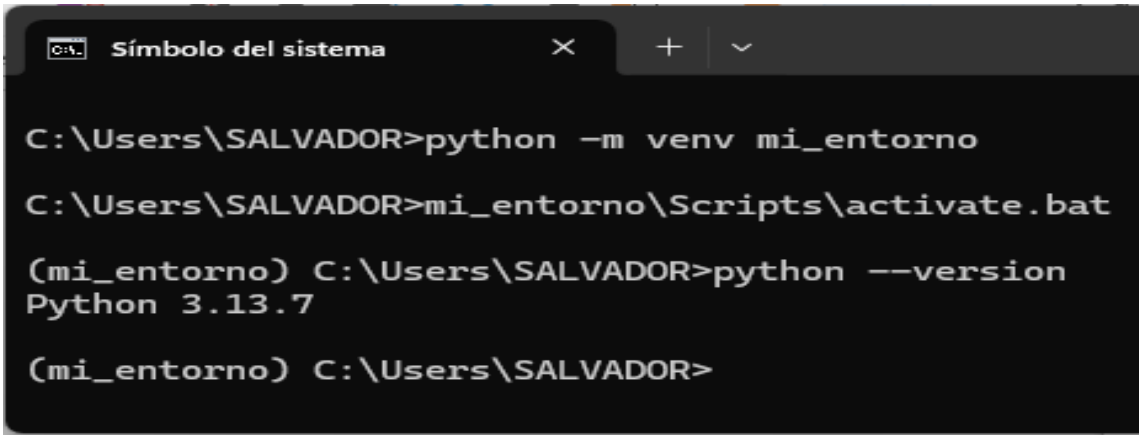
- En CMD: ***mi_entorno\Scripts\activate.bat***
- En PowerShell**:** ***.\nombre_del_entorno\Scripts\Activate.ps1***

  (Si te aparece un error de permisos, quizá tengas que habilitar la política de ejecución y luego volver a intentarlo.)

  Set-ExecutionPolicy RemoteSigned -Scope CurrentUser

3. Desactivar el entorno virtual

```
deactivate
```



Linux

```
# Crear un entorno virtual
python -m venv mi_entorno

# Activar en Linux/Mac
source mi_entorno/bin/activate

# Activar en Windows
mi_entorno\Scripts\activate

# Desactivar
deactivate
```

Con virtualenv

(alternativa más antigua)

```
pip install virtualenv
virtualenv mi_entorno
```

4.3. Herramientas avanzadas: `conda`, `pipenv`, `poetry`

- **Conda**:
  - Distribución de Python orientada a ciencia de datos.
  - Administra entornos y paquetes binarios.
  - Ejemplo:

```
conda create --name datos numpy pandas
conda activate datos
```

- **Pipenv**:
  - Combina `pip` y `virtualenv`.
  - Gestiona dependencias con un archivo `Pipfile`.
- **Poetry**:
  - Herramienta moderna para gestionar entornos, dependencias y publicación de paquetes.
  - Genera archivo `pyproject.toml`.

En este módulo trabajaremos principalmente con `venv` y `pip`, pero es importante conocer las alternativas.

Conda, Pipenv y Poetry son herramientas avanzadas para la gestión de entornos virtuales y dependencias en Python, cada una con sus enfoques y especialidades:

- **Conda** es un gestor de paquetes y entornos de propósito general, ideal para entornos científicos y de ciencia de datos, que gestiona tanto paquetes de Python como dependencias de otros lenguajes.
- **Pipenv** unifica Pip y Virtualenv, usando `Pipfile` y `Pipfile.lock` para gestionar dependencias y entornos de forma sencilla, aunque se enfoca más en aplicaciones que en librerías.
- **Poetry** es una herramienta moderna para la gestión de dependencias y el empaquetado, utilizando `pyproject.toml` y `poetry.lock` para un flujo de trabajo más integrado, similar al de npm en Node.js.

4.4. Instalación y actualización de librerías con `pip`

`pip` es el gestor de paquetes estándar de Python.

Comandos básicos

```
# Instalar
pip install requests

# Instalar versión específica
pip install requests==2.28.0

# Actualizar un paquete
pip install --upgrade requests

# Desinstalar
pip uninstall requests

# Ver paquetes instalados
pip list
```

4.5. Buenas prácticas de gestión de dependencias

- Usar siempre entornos virtuales.
- Registrar dependencias en un archivo `requirements.txt`:

```
pip freeze > requirements.txt
```

- Instalar dependencias de un proyecto:

```
pip install -r requirements.txt
```

- Evitar instalar paquetes globalmente salvo que sean de uso general.

4.6. Parte práctica (ejemplos y ejercicios)

Ejemplo 1: Creación de un entorno virtual y ejecución de un script

```
python -m venv proyecto1
source proyecto1/bin/activate   # Linux/Mac
proyecto1\Scripts\activate      # Windows
```

Crear un archivo `app.py`:

```
import requests

resp = requests.get("https://httpbin.org/get")
print(resp.json())
```

Ejecutar:

```
pip install requests
python app.py
```

Ejemplo 2: Uso de `requirements.txt`

1. Instalar un par de librerías:

   ```
   pip install requests flask
   ```

2. Guardar dependencias:

   ```
   pip freeze > requirements.txt
   ```

3. Comprobar contenido de `requirements.txt`.

4. Crear otro entorno vacío e instalar dependencias:

   ```
   python -m venv proyecto2
   source proyecto2/bin/activate
   pip install -r requirements.txt
   ```

Ejercicios propuestos

1. Crea un entorno virtual llamado `test_env`. Instala la librería `emoji`. Escribe un script que muestre un mensaje con un emoji.
2. Instalar una versión antigua de `requests`. Consultar la versión instalada con `pip show requests`. Actualizar a la última versión.
3. Escribe el comando para instalar todas las librerías listadas en un archivo `requirements.txt`.
4. ¿Qué información proporciona `pip show nombre_paquete`?
5. ¿En qué se diferencia `virtualenv` de `venv`?
6. Explica en 2–3 líneas qué es Conda y en qué contextos es más usada que venv/pip.
7. Explica qué pasaría si un proyecto requiere `Django 3.2` y otro `Django 4.1` y usas una única instalación global de Python.
8. Explica cómo podrías clonar el entorno de un compañero a partir de su `requirements.txt`.
9. Escribe un procedimiento breve para manejar dos proyectos con dependencias diferentes en la misma máquina.
10. ¿Qué herramienta puedes usar para tener múltiples versiones de Python en la misma máquina (ej. 3.7, 3.9, 3.11)?

4.7. Distribución sugerida del tiempo (6h)

- **Teoría (2h)** → entornos globales vs virtuales, `venv`, gestores alternativos, pip.
- **Ejemplos guiados (2h)** → creación de entornos, instalación de librerías, uso de `requirements.txt`.
- **Ejercicios prácticos (2h)** → trabajo individual y en parejas con proyectos simulados.