Adiós requirements.txt: uv revoluciona la gestión de dependencias en Python

12 de Enero de 2026

Adiós requirements.txt: uv revoluciona la gestión de dependencias en Python

Descubre cómo uv permite entornos reproducibles, lockfiles precisos y sincronización exacta entre máquinas y CI

9 October 2025

El **requirements.txt** ha sido una pieza clave en el desarrollo **Python** durante años. Ha resuelto problemas básicos y nos ha permitido compartir entornos de forma relativamente sencilla. Pero la realidad es que hay alternativas mejores para entornos reproducibles y rápidos. Llevo unas semanas probando y estudiando uv y en este post te explico cómo funciona, cuándo tiene sentido adoptarlo y cómo migrar un flujo clásico de requirements.in → lock → instalación.

Problemas del requirements.txt y cómo se hacía antes

El flujo clásico mínimo consistía en crear un entorno virtual, instalar las dependencias desde un requirements.txt y, cada vez que añadías una nueva librería durante el desarrollo, regenerar manualmente el archivo con pip freeze para reflejar el nuevo estado del entorno.

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


# durante el desarrollo
pip install nueva-lib
pip freeze > requirements.txt
```

Estas son las **limitaciones más comunes** del enfoque tradicional, que explican por qué **requirements.txt se ha quedado corto para proyectos medianos o grandes**:

- El requirements.txt generado con pip freeze suele listar dependencias transitorias que no quieres versionar explícitamente. Esto contamina el archivo y dificulta su mantenimiento.
- No diferencia entre las dependencias que instalas tú directamente (**top-level**) y las que esas librerías traen consigo (**transitive**), lo que complica actualizar paquetes o revisar qué usa realmente tu proyecto.
- En cada sistema operativo las dependencias pueden resolverse de forma diferente (por ejemplo, colorama solo se usa en Windows), así que con pip tradicional terminas necesitando un archivo distinto por plataforma.
- Finalmente, pip no tiene un flujo integrado de "compile → lock → sync" reproducible por defecto. Existen workarounds (constraints.txt, pip-tools) pero requieren pasos extra y no forman parte del flujo oficial.

Cómo mejora uv el flujo de desarrollo

Primero necesitamos instalar uv, ya que no viene incluido con Python. La instalación es sencilla y depende del sistema operativo que uses. Para macOS o Linux puedes instalarlo directamente desde la terminal con el script oficial de Astral: `curl -LsSf https://astral.sh/uv/install.sh | sh` .

Esto descargará la versión más reciente y la añadirá automáticamente a tu PATH. En Windows, el comando equivalente es: `PS> powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"` .

También puedes usar gestores de paquetes como Homebrew (brew install uv), Winget o Scoop, o incluso instalarlo con pipx. La documentación oficial con todas las opciones está aquí 👉 https://docs.astral.sh/uv/getting-started/installation/

Concepto básico: compile + sync

El flujo de trabajo de uv se basa en dos pasos principales: 'compilar' y sincronizar.

Primero, con `uv pip compile` generas un lockfile a partir de un **requirements.in**, resolviendo todas las versiones y dependencias necesarias de forma determinista. Este archivo (**requirements.lock**) guarda exactamente qué versión y qué hash se ha usado para cada paquete.

Después, con `uv pip sync` , instalas y ajustas el entorno para que coincida exactamente con lo que define ese lockfile: si hay paquetes de más, los elimina; si faltan, los instala.

El resultado es un entorno limpio, reproducible y consistente entre máquinas o entornos de CI.

Veamos ahora un ejemplo completo comparando cómo se hacía antes con pip y cómo se hace ahora con uv:

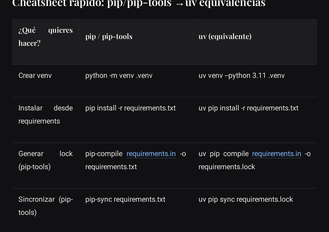
```
python -m venv .venv
source .venv/bin/activate
# definiamos top-level deps en requirements.in (o instalábamos y luego pip freeze)
pip-compile requirements.in -o requirements.txt
pip-sync requirements.txt
```

Ahora con uv el equivalente con todo lo que hemos comentado previamente sería:

```
uv venv --python 3.11 .venv          # crea venv (opcional: uv gestiona Python)
source .venv/bin/activate
# top-level deps en requirements.in
uv pip compile requirements.in -o requirements.lock   # lock reproducible
uv pip sync requirements.lock                          # instala exactamente lo loc
```

Diferencias y mejoras prácticas:

- **Resoluciones universales** con uv pip compile –universal: Permite generar un lockfile que funciona en múltiples sistemas operativos, evitando tener que crear un lock separado para cada plataforma. Esto simplifica proyectos multi-OS y ahorra mantenimiento.
- **Sincronización exacta** con uv pip sync: Garantiza que el entorno coincida exactamente con el lockfile, instalando lo que falta y eliminando paquetes sobrantes. Así no quedan dependencias huérfanas que puedan causar errores inesperados.
- **Flujo unificado**: reemplaza pip, pip-tools, virtualenv y ofrece comandos familiares (uv pip install, uv pip uninstall, uv pip freeze, uv venv). [8]
- **Rendimiento mejorado**: uv utiliza cache global de paquetes, hardlinks y copy-on-write, lo que hace que las reinstalaciones sean mucho más rápidas. Esto es especialmente útil en entornos de CI o cuando necesitas reconstruir entornos locales repetidamente.
- **Velocidad**: resolver e instalar son mucho más rápidos gracias a Rust, la paralelización y el cache global. Benchmarks oficiales muestran mejoras significativas usando warm cache como en el siguiente ejemplo instalando las dependencias de Trio.



Comparativa velocidad instalando dependencias de Trio

Nota: uv no es una réplica exacta de pip. Hay diferencias de comportamiento documentadas y casos límite a revisar antes de migrar proyectos legacy. Revisa la guía de compatibilidad. [8]

Cheatsheet rápido: pip/pip-tools → uv equivalencias

¿Qué quieres hacer?	pip / pip-tools	uv (equivalente)
Crear venv	python -m venv .venv	uv venv –python 3.11 .venv
Instalar desde requirements	pip install -r requirements.txt	uv pip install -r requirements.txt
Generar lock (pip-tools)	pip-compile requirements.in -o requirements.txt	uv pip compile requirements.in -o requirements.lock
Sincronizar (pip-tools)	pip-sync requirements.txt	uv pip sync requirements.lock
Listar freeze	pip freeze	uv pip freeze
Uninstall	pip uninstall <pkg>	uv pip uninstall <pkg>

uv no es una moda, es un replanteamiento del flujo de gestión de dependencias en Python. Aporta lockfiles reproducibles, sync estricto y un salto de rendimiento real en instalaciones y builds repetidos. Para proyectos nuevos o equipos que necesitan reproducibilidad y rapidez en CI tiene mucho sentido. Para proyectos legacy: prueba en un branch, revisa casos de compatibilidad y añade un job en CI que valide el lockfile antes de merge. Ten en cuenta que, como cualquier herramienta nueva, hay detalles y excepciones documentadas. Revisa la docu oficial antes de migrar en masa.

Nos vemos en la siguiente.

Abrazo,

Álvaro