

Python: Análisis de Datos y Ecosistemas Modernos de Analítica

ENTORNOS AISLADOS



Recordando

A medida que el programa crece en complejidad, es común empezar a crear funciones que puedan reutilizarse en distintos lugares del código.



Recordando Paquetes

Un paquete es una forma de agrupar módulos de manera estructurada.

En Python, un paquete se define a través de un árbol de carpetas que contiene módulos y, opcionalmente, otros subpaquetes.

[illegible]



Recordando Librerías

Una **librería** en Python es un conjunto de módulos y/o paquetes diseñados para ser reutilizados.

Pueden ser:

- **De la biblioteca estándar** (incluidas con Python, como os, math, datetime).
- **Externas**, instaladas con pip desde PyPI (como requests, numpy, pandas).



Recordando: Dependencias

Python cuenta con un **ecosistema muy amplio de paquetes de terceros**, lo que permite a los desarrolladores aprovechar funcionalidades avanzadas o especializadas sin tener que implementarlas desde cero.

Para instalar una librería desde el repositorio público de Python (PyPI), se utiliza:

```
python -m pip install NombreLibreria
```



Problema

A medida que un proyecto crece, el número de dependencias externas también aumenta.

Tratar de rastrear manualmente los paquetes utilizados dentro del código es ineficiente, especialmente si queremos reconstruir una aplicación desde cero.

¿Podemos mantener una lista de dependencias de mejor manera?



requirements

Los **gestores de paquetes en Python** permiten definir archivos que detallan la lista de dependencias de un proyecto, especificando las mismas características que usaríamos al instalarlas manualmente.

Convención:

requirements.txt → necesarias para ejecutar el proyecto.

requirements-dev.txt → usadas solo durante el desarrollo (por ejemplo, pruebas o linters).



requirements

requirements.txt

```
pandas  
numpy>0.5  
matplotlib  
algunalibreria==1.0.0
```

requirements-dev.txt

```
pytest  
black
```

Podemos definir las dependencias de un proyecto en un archivo e incluso especificar **rangos de versiones** compatibles usando la sintaxis de pip.

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




requirements

Incluye siempre los archivos de requisitos dentro del código fuente de tu proyecto.

Garantiza una gestión eficiente y reproducible de las dependencias, facilitando la instalación y el mantenimiento del proyecto en cualquier entorno.



requirements

Aunque definir un archivo de requisitos mejora la **reproducibilidad**, especialmente en proyectos complejos, surge un problema potencial:

Sin embargo, si no se indican las **versiones exactas** de las dependencias, cada entorno podría gestionarlas de forma distinta, lo que genera comportamientos inconsistentes o errores difíciles de rastrear.



requirements

Solución: Congelar explícitamente las versiones de las librerías.

Esto garantiza una mayor **reproducibilidad del entorno** y permite detectar regresiones al actualizar dependencias de forma controlada.

Esto se conoce como una **lockfile**.



Lockfile

Beneficios

- Capturan el **estado exacto** de todas las dependencias (incluyendo subdependencias).
- Garantizan **reproducibilidad total** del entorno.

Desventajas

- Los archivos generados pueden ser **muy largos y difíciles de mantener**.
- Requieren un proceso adicional de **renovación periódica** para no quedarse obsoletos.



Crear un Lockfile

Con **pip freeze** podemos congelar las versiones de **todas** las **librerías del entorno**, incluidas sus dependencias transitivas.

```
python -m pip freeze > requirements-frozen.txt
```

Esto asegura una **reproducibilidad total del entorno** y resulta especialmente útil para detectar regresiones cuando se actualizan versiones de librerías de forma controlada e independiente.



Compilación controlada

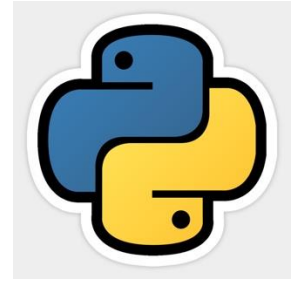
Con **pip-tools** podemos mantener un archivo requirements.in con las dependencias principales y los rangos de versiones aceptables.

```
python -m pip install pip-tools  
python -m piptools compile requirements.in
```

Una ventaja adicional es que, durante este proceso, pip-tools puede actualizar las versiones respetando las reglas definidas en el archivo de entrada.

```
python -m piptools compile requirements.in --upgrade
```

[pip-compile - pip-tools documentation v7.4.2.dev57](#)



requirements

requirements.in

```
pandas>2
```

requirements.txt

```
#  
# This file is autogenerated by pip-compile with Python 3.10  
# by the following command:  
#  
#   pip-compile requirements.in  
#  
numpy==2.1.1  
    # via pandas  
pandas==2.2.2  
    # via -r requirements.in  
python-dateutil==2.9.0.post0  
    # via pandas  
pytz==2024.2  
    # via pandas  
six==1.16.0  
    # via python-dateutil  
tzdata==2024.1  
    # via pandas
```



Encuesta

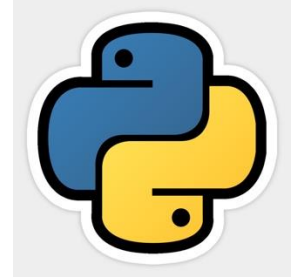
¿Cuál es el propósito de un archivo requirements.txt?



Encuesta

¿Cuál es el propósito de un archivo requirements.txt?

Definir una lista explícita de dependencias para la aplicación



Encuesta

¿Qué es mejor para un proyecto?:

1. Un archivo **requirements.txt** con dependencias declaradas, sin especificar versiones.
2. Un archivo **requirements.txt** creado con **pip compile** que incluye todas las versiones exactas.



Encuesta

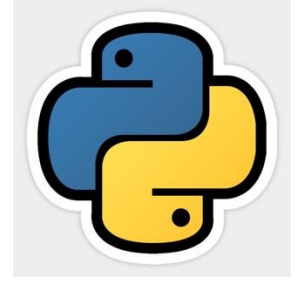
¿Qué es mejor para un proyecto?:

1. Un archivo **requirements.txt** con dependencias declaradas, sin especificar versiones.
2. Un archivo **requirements.txt** creado con **pip compile** que incluye todas las versiones exactas.

DEPENDEN

Flexibilidad → actualizar fácilmente, pero efectos inesperados.

Robustez → entornos reproducibles, costo de mantenimiento.



Recomendación

¡Siempre mantenerse al tanto de las versiones de nuestras librerías, prevenir la **deuda técnica** a medida que se publican nuevas versiones!

Prefiere **pip-tools compile** a partir de un archivo **requirements.in**



Otro Problema

Como vimos anteriormente, cuando instalamos dependencias con **pip**, estas se ubican en el directorio por defecto del sistema.

Una única versión de cada librería queda disponible para **todas las aplicaciones** que se ejecutan en ese sistema.

Problemas:

- Todas las aplicaciones quedan atadas a una misma versión de la librería.
- Se pierde control si alguien actualiza o cambia la dependencia global del sistema.
- Existe poca flexibilidad si no se tiene acceso al directorio centralizado de instalación.



Solución: Entornos aislados

Una solución es usar **entornos virtuales**, que permiten gestionar las dependencias de una aplicación en un contexto aislado e independiente del sistema.

De esta forma, varias aplicaciones pueden convivir en el mismo equipo, cada una con sus propias versiones de librerías, evitando conflictos y garantizando mayor consistencia.



Solución: Entornos aislados

Existen muchas formas de aislar dependencias; una de ellas es el uso de contenedores como **Docker**, aunque esto requiere conceptos externos que no se cubren en este curso.

La opción más común en Python son los **entornos virtuales**, creados fácilmente con la herramienta integrada **venv**.



venv: virtual environment

La herramienta **venv** viene incluida por defecto en las versiones recientes de Python.

Con el siguiente comando se crea un entorno virtual en el directorio actual:

```
python -m venv venv
```

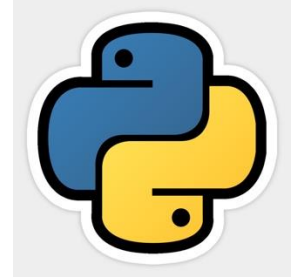
Esto generará una carpeta llamada **venv** que contendrá [todas las dependencias](#) del proyecto y una [copia aislada del intérprete de Python](#).



venv: virtual environment

Una vez **activado** el entorno virtual con **activate**, tanto la instalación de paquetes como la ejecución del código se realizan únicamente dentro de ese contexto.

Para **desactivar** el entorno y volver al sistema, basta con ejecutar **deactivate**



Importancia

Python recomienda el uso de **entornos virtuales** para la gestión de proyectos porque su sistema de dependencias no es el más robusto.

Evita generar conflictos entre proyectos.



Prevenir

- **Polución** de paquetes, incluso sobre utilidades base del sistema.
- **Conflictos de versiones** entre proyectos.
- **Falta de reproducibilidad**, dificultando replicar problemas o entornos.
- Necesidad de **privilegios de administrador** para ciertas instalaciones.



¿Cómo funciona?

Un **entorno virtual** es simplemente una carpeta que contiene una copia del intérprete de Python y un espacio donde se instalarán los paquetes externos.

Al **activarlo**, el terminal ajusta su contexto para usar ese intérprete y sus paquetes, ignorando por completo la instalación global del sistema.



Alternativas

Existen varias herramientas para gestionar entornos:

- **virtualenv**: el proyecto original que inspiró la creación de venv.
- **Pipenv**: inspirado en el manejo de dependencias de Node.js, combina entornos y lockfiles.
- **Conda**: solución más amplia, usada en múltiples lenguajes y entornos científicos.

Sin embargo, **venv** viene incluido por defecto en Python y suele ser suficiente.



Encuesta

¿Cuál es mayor ventaja de los entornos virtuales aislados de Python?



Encuesta

¿Cuál es mayor ventaja de los entornos virtuales aislados de Python?

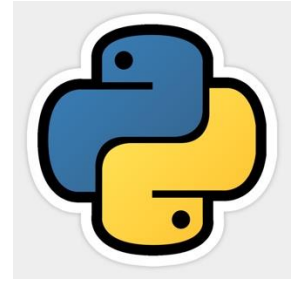
Aislar dependencias y su manejo controlado



Entornos virtuales

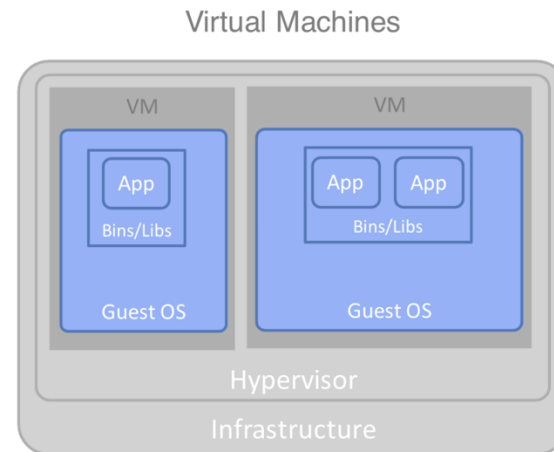
Los entornos virtuales aseguran consistencia en proyectos de Python, pero en aplicaciones más complejas presentan limitaciones.

La principal es que solo gestionan **paquetes y dependencias de Python**, dejando fuera librerías o servicios externos que también pueda necesitar la aplicación.



Máquinas virtuales

La virtualización completa de otro sistema operativo y sus recursos físicos. Una máquina virtual (VM) es un archivo de computadora, típicamente llamado imagen, que se comporta como una computadora real.





Máquinas virtuales

Ofrecen aislamiento completo, ya que requieren un **hypervisor**, encargado de virtualizar sistemas operativos enteros dentro de una misma máquina física.

Problemas:

- **Alto costo** en recursos y rendimiento.
- **Sobrecarga** por la virtualización completa a nivel de sistema operativo.
- **Mayor complejidad operativa**, especialmente al empaquetar y desplegar aplicaciones en entornos escalables.



Contenedores

Son **paquetes de software ligeros, autónomos y ejecutables** que incluyen todo lo necesario para ejecutar una aplicación: código, tiempo de ejecución, herramientas del sistema, bibliotecas y configuración.

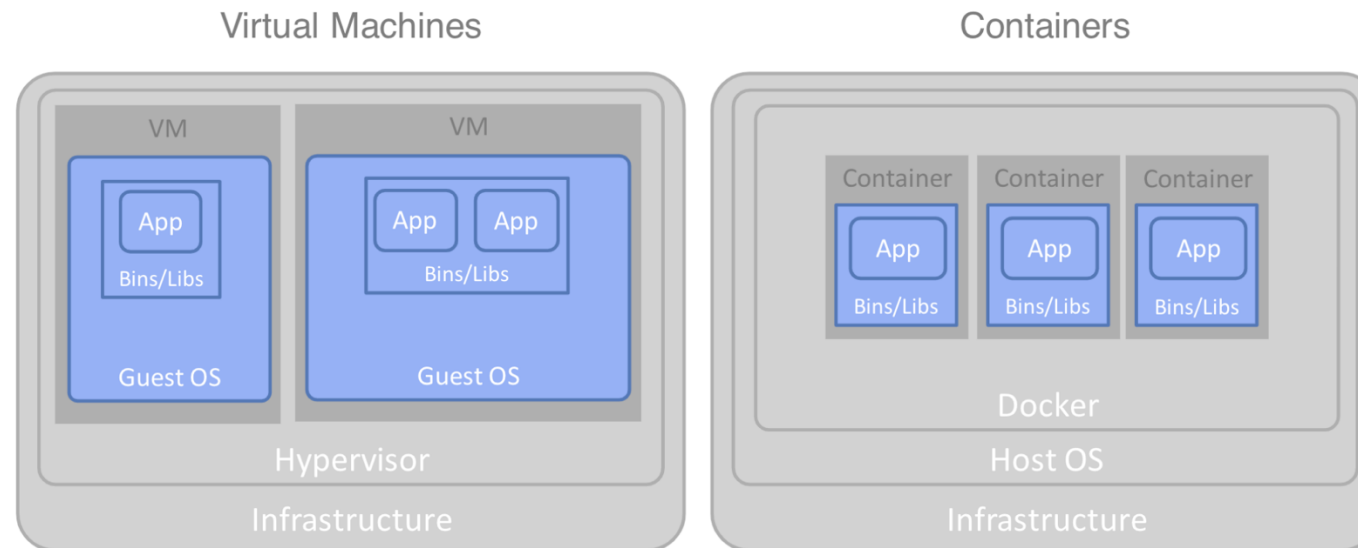
Los **contenedores** virtualizan el entorno de ejecución a nivel del sistema operativo, ofreciendo aislamiento sin necesidad de un sistema operativo completo como en las máquinas virtuales.

Ejemplo: Docker, un gestor de contenedores muy utilizado en sistemas Linux y otros entornos.



Contenedores

Virtualización del **sistema operativo** vs el **entorno de ejecución del sistema operativo**.





Contenedores

Uno de los beneficios de los **contenedores** es que solo virtualizan el entorno de ejecución del sistema operativo.

Esto permite crear **contextos aislados** con mayor alcance que un entorno virtual de Python, pero con un costo mucho menor que las máquinas virtuales, ya que comparten los recursos del mismo sistema operativo.



Contenedores

Los **contenedores** ofrecen un alcance más amplio que los entornos virtuales de Python, ya que no solo gestionan las dependencias del lenguaje, sino también las **dependencias del sistema** necesarias para la aplicación.

Además, simplifican el **despliegue en entornos de microservicios** y facilitan la ejecución en **sistemas escalables y multiplataforma**, garantizando consistencia y compatibilidad.



Trabajando con contenedores

La herramienta de contenedores más utilizada en la industria es **Docker**, aunque existen otras alternativas.

Docker se basa en el concepto de **imágenes**, que definen la configuración necesaria para crear y ejecutar un contenedor.



Dockerfile

Un **Dockerfile** es un archivo de texto con instrucciones que sirven como **plantilla para construir una imagen de Docker**. Esa imagen resultante puede usarse para crear uno o varios contenedores.

```
# Imagen base oficial de Python 3.12
FROM python:3.12-slim

# Establecer directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar los archivos de dependencias
COPY requirements.txt .

# Instalar las dependencias en el contenedor
RUN pip install --no-cache-dir -r requirements.txt

# Copiar el código de la aplicación
COPY . .

# Definir el comando por defecto para ejecutar la aplicación
CMD ["python", "app.py"]
```



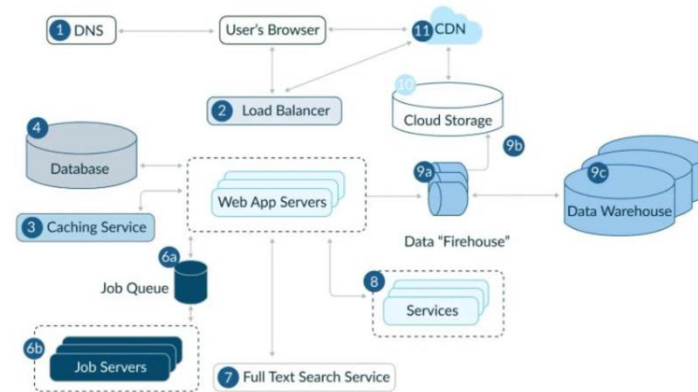

Contenedores

- **Más eficientes que las máquinas virtuales**, ya que comparten el sistema operativo y son más fáciles de gestionar.
- **Más completos que los entornos virtuales de Python**, al aislar también las dependencias y configuraciones del sistema.



Sistemas complejos

¡El concepto de **contenedores** no se limita a aplicaciones de Python!



Sus beneficios de consistencia y virtualización permiten aislar **todos los componentes** de una arquitectura: servicios, bases de datos, colas de mensajería, APIs y aplicaciones completas, garantizando que trabajen de forma coherente en cualquier entorno.



Docker compose

Herramienta para gestionar varios contenedores como parte de una **arquitectura unificada**. Permite definir, en un único archivo (docker-compose.yml), los **servicios**, **volúmenes** y **redes internas** que conectan los contenedores.

En resumen: combina múltiples servicios definidos con imágenes de Docker y los hace trabajar en conjunto.



Docker compose

```
version: "3.9"

services:
  app:
    build: .
    container_name: my_app
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    image: postgres:15
    container_name: my_db
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    volumes:
      - db_data:/var/lib/postgresql/data

volumes:
  db_data:
```



Docker compose

Simplifica el **aislamiento** y la **reproducibilidad** de todos los componentes de una aplicación, automatizando por completo el despliegue y la inicialización de un sistema completo en entornos de desarrollo.

```
docker-compose up
```

```
$ docker-compose ps
```

Name	Command	State	Ports
mi-aplicacion_backend_1	python app.py	Up	0.0.0.0:5000->5000/tcp
mi-aplicacion_basedatos_1	docker-entrypoint.sh	Up	5432/tcp
mi-aplicacion_frontend_1	sh -c "npm install && ...	Up	0.0.0.0:3000->3000/tcp

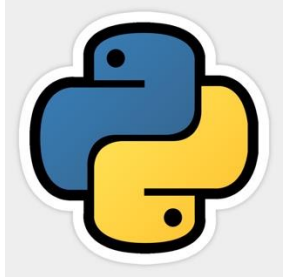


En la nube

En la nube, los contenedores habilitan entornos que serían imposibles de lograr con máquinas virtuales tradicionales por su peso y complejidad.

Gracias a su ligereza y portabilidad, los contenedores permiten crear **entornos de desarrollo bajo demanda**, que se inician en segundos y con la misma configuración para todos los usuarios.

Github Codespaces

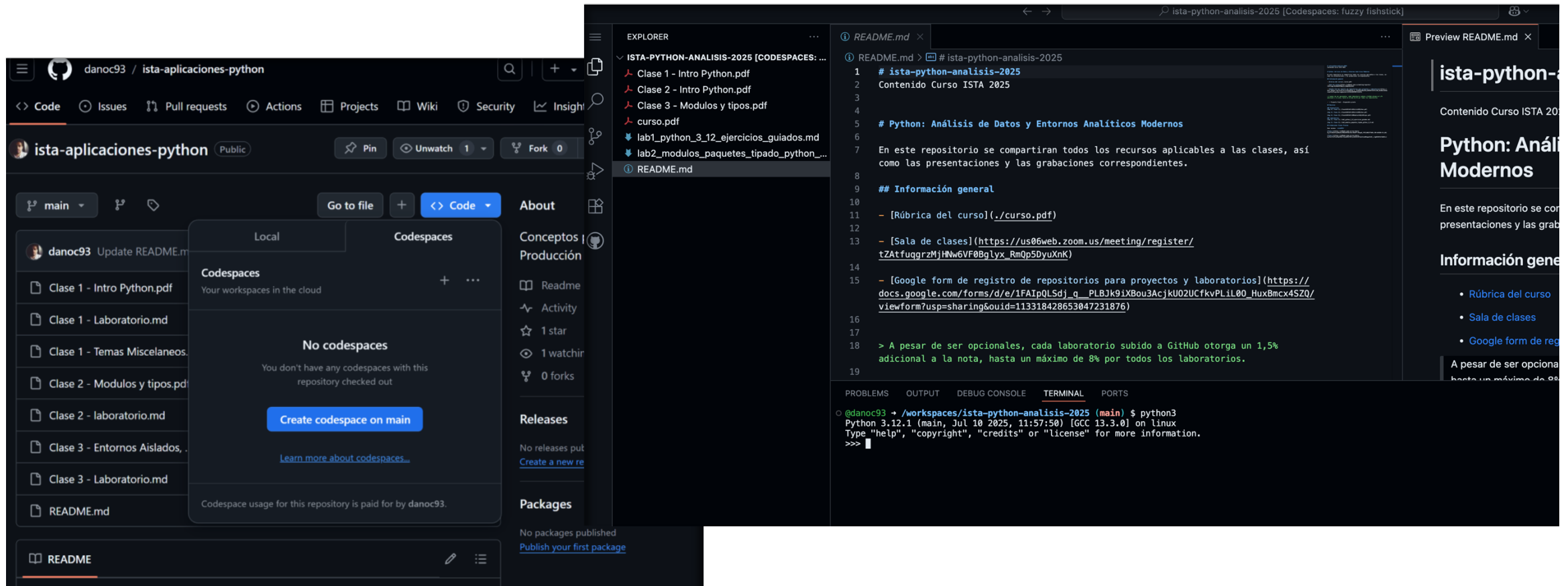
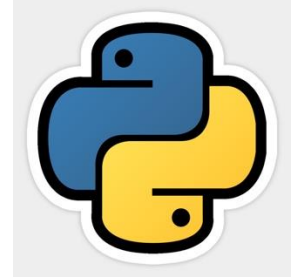


Un servicio que ofrece un entorno de desarrollo completo en la nube, basado en contenedores.

Esto permite a los desarrolladores abrir un repositorio y comenzar a trabajar de inmediato, sin preocuparse por instalaciones locales o dependencias, asegurando consistencia entre equipos y plataformas.

<https://github.com/features/codespaces>

Github Codespaces

The screenshot shows the GitHub web interface for the repository 'ista-aplicaciones-python' by user 'danoc93'. The repository is public and has 0 forks. The file explorer on the left shows a list of files: 'Clase 1 - Intro Python.pdf', 'Clase 2 - Intro Python.pdf', 'Clase 3 - Modulos y tipos.pdf', 'curso.pdf', 'lab1_python_3_12_ejercicios_guiados.md', and 'lab2_modulos_paquetes_tipado_python...'. The main content area displays the 'README.md' file, which contains the following text:

```
1 # ista-python-analisis-2025
2 Contenido Curso ISTA 2025
3
4
5 # Python: Análisis de Datos y Entornos Analíticos Modernos
6
7 En este repositorio se compartiran todos los recursos aplicables a las clases, así
8 como las presentaciones y las grabaciones correspondientes.
9
10 ## Información general
11
12 - [Rúbrica del curso](./curso.pdf)
13
14 - [Sala de clases](https://us06web.zoom.us/meeting/register/
15 tZAtfuggrzMjHw6VF0Bglyx_Rm0p5DyuXnK)
16
17 - [Google form de registro de repositorios para proyectos y laboratorios](https://
18 docs.google.com/forms/d/e/1FAIpQLSdj_q_PLBjK9iXBou3AcjKU02UCfkVPLiL00_HuxBmcx4SZQ/
19 viewform?usp=sharing&id=113318428653047231876)
20
21 > A pesar de ser opcionales, cada laboratorio subido a GitHub otorga un 1,5%
22 adicional a la nota, hasta un máximo de 8% por todos los laboratorios.
```

The right sidebar shows a preview of the README.md file. The bottom of the interface shows the terminal output of a command: '@danoc93 → /workspaces/ista-python-analisis-2025 (main) \$ python3'. The output shows the Python version (3.12.1) and the GCC version (13.3.0) on Linux.

**HERRAMIENTAS
DE CALIDAD Y
ROBUSTEZ**



linter

El **linting** es el proceso de señalar **errores sintácticos y estilísticos** en el código.

Permite identificar y corregir prácticas que pueden provocar errores o dificultar la mantenibilidad.

Ejemplos de problemas que detecta un linter:

- Variables no inicializadas.
- Llamadas a funciones no definidas.
- Paréntesis o signos de puntuación faltantes.
- Estilo inconsistente en el formato del código.



formatter

Un **formatter** ajusta automáticamente el **formato del código** para que mantenga consistencia.

Puede aplicar convenciones generales o seguir reglas específicas del proyecto.

Ejemplos:

- Indentación uniforme
- Espacios alrededor de operadores
- Ordenamiento de importaciones

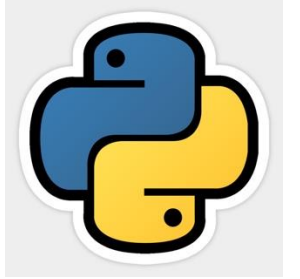


Importancia

Muchos **IDEs** modernos incluyen linters y formatters integrados, lo que ayuda a detectar problemas de inmediato mientras se escribe código.

Estas herramientas también pueden instalarse como librerías y ejecutarse en **entornos automatizados** (por ejemplo, en pipelines de CI/CD), garantizando verificaciones consistentes más allá del entorno local de cada desarrollador.

ruff



Es una herramienta **todo en uno** para Python:

- **Lint**er (detecta errores y malas prácticas).
- **Formateador** (ajusta el estilo del código automáticamente).

```
ruff check .
```

```
ruff format .
```

No es la única herramienta disponible, pero sí la más **rápida**.



Chequeo Estático

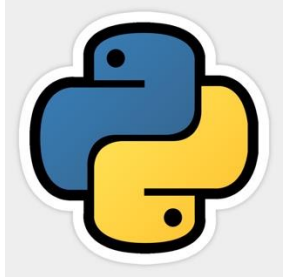
Recordemos:

En Python, las variables y funciones son **dinámicas**, lo que puede generar errores solo visibles en tiempo de ejecución.

El **chequeo estático** permite analizar el código sin ejecutarlo, detectando inconsistencias antes de que lleguen a producción.

Verifica que las anotaciones de tipo (int, str, list, etc.) se respeten en el código.

mypy



El chequeo estático basado en anotaciones de tipo ayuda a prevenir errores como **pasar argumentos incorrectos** o **retornar valores inesperados**, antes de ejecutar el programa.

```
mypy src/
```



Automatización de chequeos

Una **plataforma de CI/CD** permite ejecutar acciones automatizadas sobre un proyecto.

La **integración continua (CI)** automatiza la validación de cambios:

- Chequeos de calidad
- Pruebas
- Testing automatizado

La **entrega continua (CD)** facilita que el código validado esté siempre listo para desplegarse.

[¿En qué consiste la integración continua? | Atlassian](#)



Automatización de chequeos

En un flujo de **CI/CD**, podemos integrar las herramientas de calidad que ya conocemos para garantizar robustez en cada cambio:

Ruff → Linter y formateador rápido.

mypy → Chequeo estático de tipos.

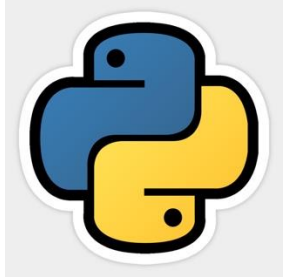
Pytest → Testeo (siguiente clase).

En librerías → Empaquetado, Verificación, Publicación.

En aplicaciones → Despliegue.

[¿En qué consiste la integración continua? | Atlassian](#)

Explorar



GitHub Actions

Plataforma de **automatización integrada en GitHub**, permite ejecutar flujos de trabajo (workflows) en cada *push*, *pull request* o evento definido.

CircleCI

Plataforma de **CI/CD en la nube**, con gran flexibilidad y soporte para múltiples lenguajes y entornos.



Laboratorio

<https://github.com/danoc93/ista-python-analysis-2025>