BlackCoffee: Tooling Python y Data Science

Mario García Márquez

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

España | digital 2026

UNIVERSIDAD DE GRANADA

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

repsol

Contenidos de la charla

La charla se divide en dos partes

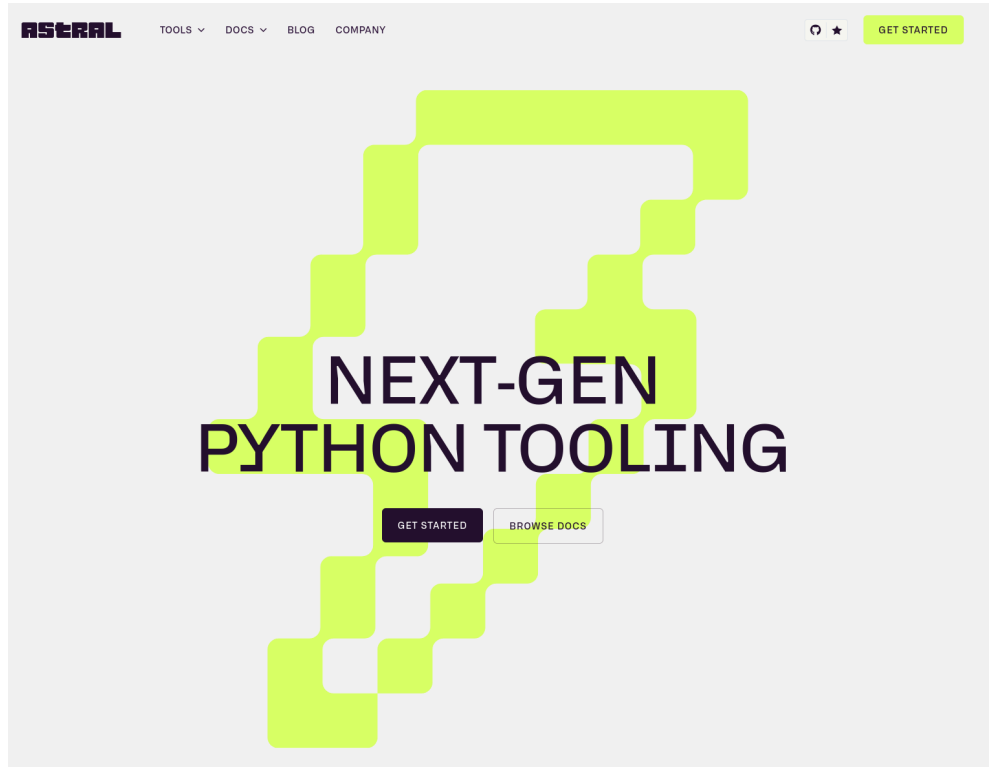
- Tooling
  - UV - Gestor de proyectos
  - Ruff - Linter
  - TY - Type Checker
- Librerías
  - Hydra - Gestión de configuración
  - WandB - Tracking de experimentos

¿Por qué tooling?

- Como científicos, gran parte de nuestra de investigación acaba generando código como resultado.
- Idealmente, este código tiene que ser claro, reproducible y fácil de mantener, para que nuestra investigación también lo sea.
- El tooling nos va a ayudar a esto

Contexto sobre el tooling

- Las herramientas que vamos a ver están desarrolladas por Astral.
- Se centran en hacer tooling para Python con foco en la eficiencia. Todo escrito en Rust.
- Comprado por OpenAI en marzo de este año.



UV - Gestor de proyectos

Esta herramienta nos va a permitir:

- Gestionar versiones de Python y venv
- Gestionar dependencias de proyectos
- Asegurar que las dependencias sean reproducibles
- Publicar paquetes en PyPi

Siendo x10-100 veces más rápido de pip en el proceso.

Instalación

Existen dos maneras de instalar UV:

La primera opción utiliza pip, recomiendan hacerlo en un entorno aislado (e.g.: entorno de conda):

```
pip install uv
```

La segunda opción te instala el binario en tu home (no require de sudo):

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Creando un proyecto

Primeros pasos

Para crear un proyecto usando UV, basta ejecutar el siguiente comando:

```
uv init  nombre del proyecto
```

Esto generará una carpeta con el nombre asignado y una estructura básica. UV también permite crear proyectos orientados a ser librerías y posteriormente ser publicados, para ello usaremos:

```
uv init --lib nombre del proyecto
```

Si no especificamos un nombre, el proyecto se creará en la carpeta en la que estemos y cogerá el nombre de esta.

Estructura del proyecto

Primeros pasos

La estructura resultante de nuestro proyecto sera la siguiente:

```
tree -L 1 -a -C
.
├── .git
├── .gitignore
├── main.py
├── pyproject.toml
├── .python-version
└── README.md
```

El archivo pyproject.toml

Primeros pasos

El contenido del archivo es el siguiente:

```
[project]
name = "blackcoffee"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = "⩾3.14"
dependencies = []
```

Ejecutando un programa

Primeros pasos

Ejecutaremos nuestro `main.py` con el siguiente comando:

```
uv run main.py
```

Al ejecutar esto se nos han creado nuevos directorios:

```
tree -L 1 -a -C
.
├── .git
├── .gitignore
├── main.py
├── pyproject.toml
├── .python-version
├── README.md
├── uv.lock
└── .venv
```

Instalando dependencias

Primeros pasos

Para instalar una librería, por ejemplo numpy, usaremos el siguiente comando:

```
uv add numpy
```

UV también cuenta con una interfaz equivalente a la de pip, por si no queréis aprender nada nuevo:

```
uv pip install numpy
```

Esto nos habrá modificado nuestro archivo `pyproject.toml`:

```
dependencies = [
    "numpy⩾2.4.6",
]
```

Se pueden especificar versiones concretas de paquetes con `uv add "numpy==2.0.0"`.

Gestión de dependencias

Primeros pasos

Obviamente, uv también nos permite eliminar dependencias:

```
uv remove numpy
```

o actualizarlas:

```
uv lock --upgrade
```

La actualización puede ser de un único paquete:

```
uv lock --upgrade-package numpy
```

Además, uv utiliza una cache en el sistema para almacenar paquetes, se acabó tener 40 copias de pytorch.

Locking de paquetes

Asegurando la consistencia

El archivo `uv.lock` guarda la información de todos los paquetes instalados en el proyecto, de manera que cuando alguien se baje este proyecto tenga exactamente las mismas dependencias.

Este archivo se actualiza de forma automática cada vez que instalamos algo. Para tener el entorno actualizado acorde a este archivo (sin ejecutar `uv run`) basta con ejecutar:

```
uv sync
```

Si hiciese falta, podemos exportar esta información a un `requirements.txt`:

```
uv export --format requirements.txt
```

Herramientas

No hace falta instalar para ejecutar

Muchas veces nos instalamos paquetes de pip que en realidad son ejecutables (mismamente UV). UV ofrece una herramienta para ejecutar estas herramientas sin necesidad de instalarlas con pip en ningún entorno. Para ello usaremos:

```
uv tool run paquete
```

O alternativamente

```
uvx paquete
```

Ruff - Linter

Para que tu código no esté guarro

Ruff fue la primera herramienta de Astral.

- Ofrece tanto un linter como un formateador
- x100 más rápido que flake8 (tarda solo 0.29s en hacer un pase por el repo de CPython)
- Sustituye a muchas herramientas como isort, black o flake8.
- Tiene extensión de VSCode (la mejor manera de usarlo)

Caso de uso

Proyecto para descargar

Para la demostración de como usar Ruff, lo mejor es hacerlo con algo de código ya escrito.

El proyecto se puede encontrar en github.com/mariogmarq/BlackCoffee.

Manos a la obra

Empezamos a limpiar el proyecto

El proyecto que os habéis descargado usa `uv`, asi que usad `uv sync` para instalaros las dependencias.

Ruff se instala con pip, pero vamos a aprovechar que tenemos ya `uv` instalado para ello. Ejecutaremos ruff con

```
uvx ruff
```

Manos a la obra

Empezamos a limpiar el proyecto

Primero de todo pasaremos el linter, para ello basta con ejecutar

```
uvx ruff check
```

Si quisieramos solo comprobar un archivo en concreto haríamos

```
uvx ruff check main.py
```

Ruff nos dice automaticamente que algunos errores se pueden arreglar automaticamente, para ello

```
uvx ruff check --fix main.py
```

Las reglas de Ruff

Un mundo de posibilidades

Ruff por defecto considera un subconjunto de normas para el linter, pero hay muchas que vienen desactivadas. Para ejecutar el linter con alguna norma en concreto usaremos el comando `--select`. Por ejemplo, para comprobar un orden correcto de imports

```
uvx ruff check --select I main.py
```

Este ajuste se puede hacer de manera permanente con el siguiente bloque dentro de `pyproject.toml`:

```
[tool.ruff.lint]
select = ["E", "F", "I"]
ignore = ["F401"]
```

A formatear el código

Hay que ponerse bonito

Que el código sea legible y consistente es importante, ruff por defecto sirve de sustituto del famoso formateador `black`. Para ello, basta con ejecutar:

```
uvx ruff format
```

Cualquier cambio de configuración se haría modificando nuestro archivo `pyproject.toml`:

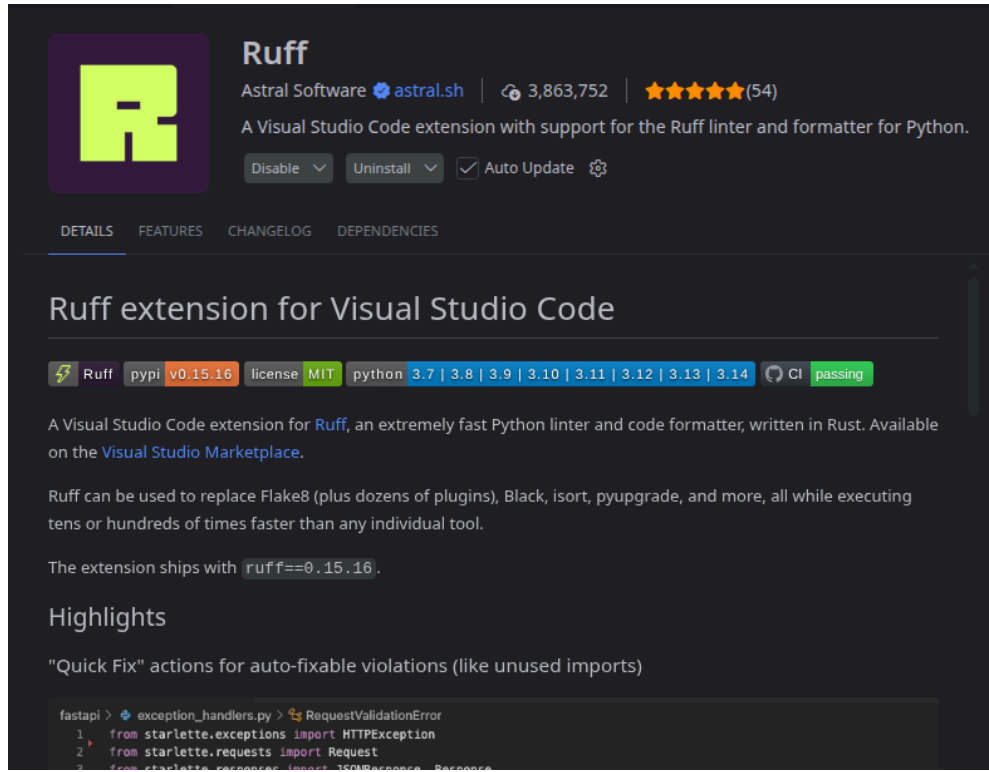
```
[tool.ruff]
line-length = 100

[tool.ruff.format]
quote-style = "single"
indent-style = "tab"
docstring-code-format = true
```

La extensión de VsCode

La joya de la corona

Os permite tener todo dentro del editor’

- Se puede configurar para que formatee el archivo al guardar
- Te permite ver los errores en el editor y corregirlos pulsando un solo botón
- Es la manera más cómoda de usarlo



TY - Type Checker

Tipado estático

- La última herramienta en la que está trabajando Astral.
- Permite comprobar que los tipos especificados en nuestro código estén bien.
- Más sofisticado y mucho más rápido que alternativas como Mypy.

Tipos en Python

Sí, esto existe.

Puede que algunos no lo sepáis pero Python, aunque sea un lenguaje interpretado y dinámico, permite incluir tipos a su código, por ejemplo:

```
from typing import List

def sum(l: List[int]) -> int:
  acc = 0
  for num in l:
    acc += num
  return acc
```

En ejecución, Python ignora estos tipos, unicamente son usados por LSPs y para información para el usuario. Para saber si los tipos están bien (y ahorraros bugs en vuestro código), tenéis que usar un programa externo.

Ejecutando TY

Comprobación de tipos

Para ejecutar TY, basta con ejecutar el siguiente comando:

```
uvx ty check
```

Al ejecutar este comando, nos detectará un error con uno de los argumentos de una función. Al igual que Ruff, TY se configura con reglas dentro de nuestro `pyproject.toml`.

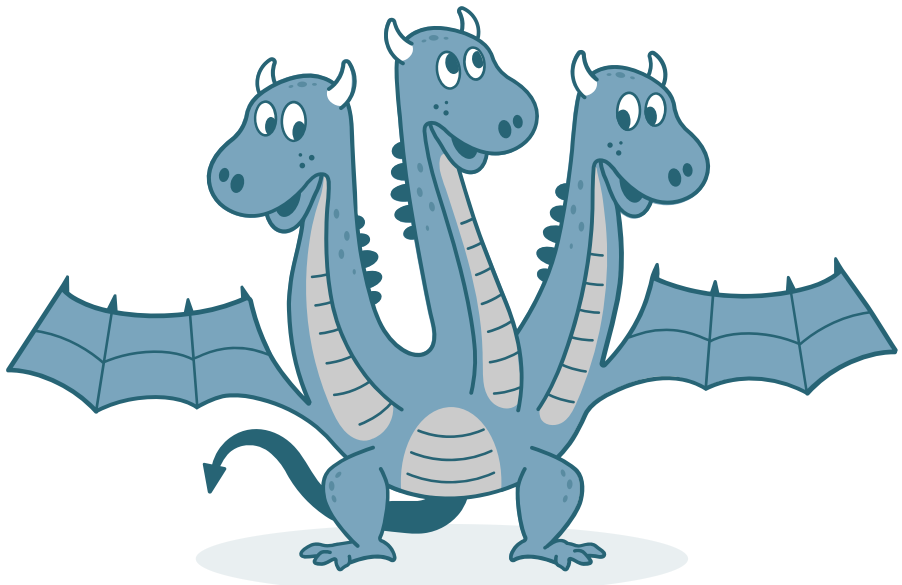
```
[tool.ty.rules]
redundant-cast = "ignore"
invalid-key = "warn"
missing-override-operator = "error"
```

También cuenta con su propia extensión de VsCode.

Hydra

Configurando experimentos

- Librería desarrollada por Meta.
- Permite la gestión de la configuración de los experimentos mediante archivos yaml.
- Permite que la configuración sea modular e intercambiable.
- La manera más sencilla de compartir los parámetros de tu baseline. 😎



Instalación

Más de lo mismo

Para instalar Hydra, nos bastará con ejecutar en nuestro proyecto el siguiente comando:

```
uv add hydra-core
```

Configuración básica

Primeros pasos

Para usar Hydra, nos bastará con decorar nuestra función `main`:

```
import hydra
from omegaconf import DictConfig

...

@hydra.main(version_base=None)
def main(cfg: DictConfig):
  print(cfg)
  ...
```

Podremos ejecutar nuestra aplicación con:

```
uv run main.py +batch_size=256
```

Moviendo la configuración a un archivo

Para que la cosa sea sostenible

Crearemos un archivo `conf/config.yaml` en nuestro proyecto con los contenidos deseados:

```yaml
training:
  batch_size: 256
  epochs: 10
```

y modificaremos nuestro `main.py` para indicarle que esa va a ser nuestra configuración:

```python
@hydra.main(version_base=None, config_path="conf", config_name="config")
def main(cfg: DictConfig):
  print(cfg)
  ...
```

Ejecutaremos con:

```
uv run main.py
uv run main.py training.batch_size=128
uv run main.py +training.dropout=false
```

Varias configuraciones

Modularización

Imaginemos la siguiente estructura:

```
conf/
├── config.yaml
└── training
    ├── beta.yaml
    └── gaussian.yaml
```

y el siguiente contenido de `config.yaml`

```yaml
defaults:
 - training: gaussian
```

Bastaría con ejecutar:

```
uv run main.py
uv run main.py training=beta
uv run main.py training.batch_size=128
```

Tipado estático con Hydra

Ya que sabemos usar TY, hay que aprovechar

La manera más rápida y sencilla de obtener tipado es mediante ducktyping. Definimos una dataclass conveniente:

```python
from dataclasses import dataclass

@dataclass
class TrainingConf:
  batch_size: int
  dropout: bool

@dataclass
class Config:
  training: TrainingConf

@hydra.main(version_base=None, config_path="conf", config_name="config")
def main(cfg: Config):
  print(cfg)
```

Tipado estático con Hydra

Más estricto

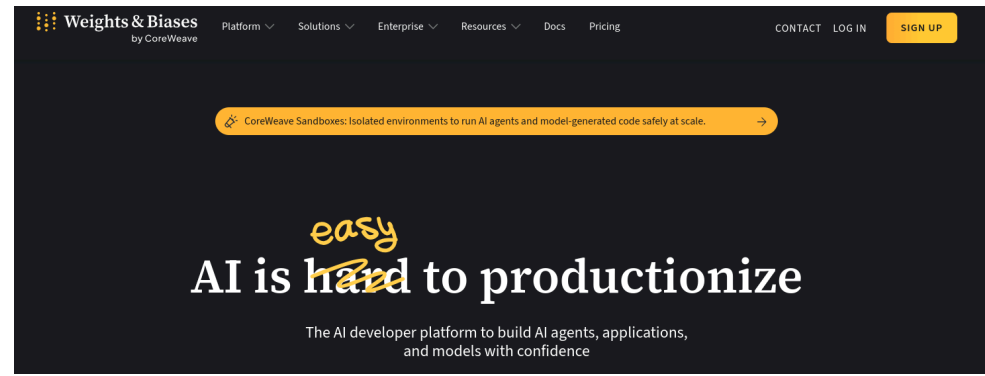
Si además quisieramos que Hydra nos capturase errores de tipos al pasar los argumentos escribiríamos lo siguiente:

```
from hydra.core.config_store import ConfigStore

cs = ConfigStore.instance() # Singleton pattern
cs.store(name="config_schema", node=Config)
```

Weight & Biases

Tracking de experimentos

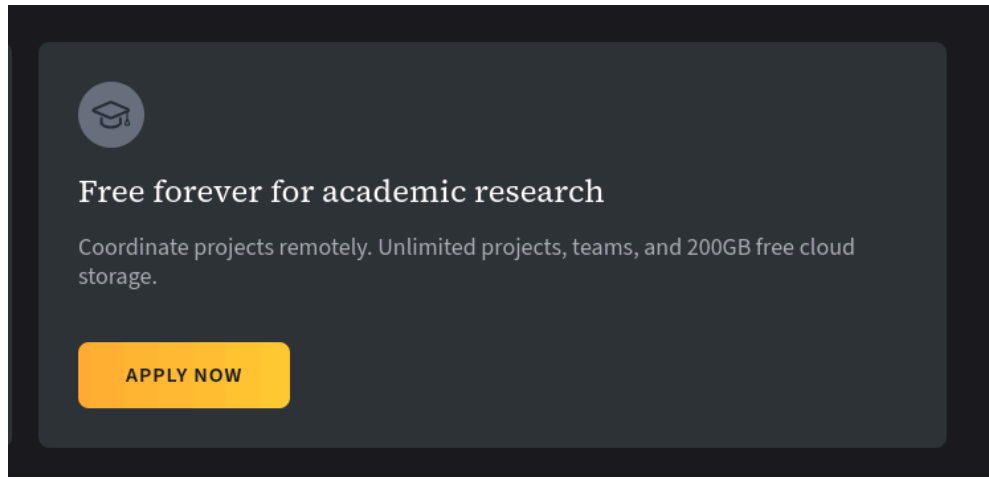
- Herramienta online para tracking de experimentos.
- Permite organizar las runs de manera limpia y ordenada.
- Fácil de compartir resultados.
- Permite almacenar modelos.
- MLFlow como alternativa OpenSource.



Creando una cuenta

Importante para la universidad

- Acceder a la web de WandB para haceros una cuenta.
- Dentro de la pestaña de pricing aparece para solicitar una cuenta académica.
- Permite tener más proyectos, equipos y almacenamiento para modelos.



Creando una API KEY

Autentificación

Para crear una API KEY de WandB hay que seguir los siguientes pasos:

- Pulsar en tu icono e ir a "User Settings".
- Pulsar en "Create new API key".
- Create.
- Copiar la API Key (solo aparece una vez).
- Esta key debería de ir en una variable de entorno llamada `WANDB_API_KEY`.

```
export WANDB_API_KEY=<your_api_key>
```

Configurando WandB en un proyecto

Primeros pasos

Como venimos acostumbrando el primer paso es instalar WandB en nuestro proyecto:

```
uv add wandb
```

Los cambios en el código son mínimos:

```
import wandb

def main(config: Config) -> None:
  ...
  with wandb.init(project="project-name", config=config, tags=["debug", "foo"]) as run:
    for epoch in range(config.epochs):
      ...
      run.log({"loss": loss, "epoch": epoch})
```

Cosas buenas de WandB

Aquí toca trastear y ver una demo

- Nos permite tener todo ordenado por proyectos, filtar, agregar runs, etc.
- Fácil de compartir con tu supervisor.
- Logging automático de métricas del sistema, para hacer un uso correcto de los servidores.
- Guarda la configuración y commits de las run, para máxima reproducibilidad.
- Guarda el log de la consola en un archivo para su fácil gestión

Más características

Un pelín más de uso

- Además del logging básico de métricas WandB también permite registrar otro tipo de contenido como imágenes.

```
wandb.log({"reconstructed_views": wandb.Image("views.png")}, step=epoch)
```

- También podemos guardar binarios o "artifacts":

```
artifact = wandb.Artifact(name="resnet-18", type="model")
artifact.add_file(local_path="path/to/resnet.pt")
run.log_artifact(artifact)
```

- Para usarlos más adelante:

```
artifact = run.use_artifact("resnet-18:latest")
directory = artifact.download()
```

- Para el caso exacto de modelos también existe `run.log_model` y `run.use_model`

Descargando las métricas para procesarlas

El último paso

Idealmente, al final de la experimentación nos gustaría procesar nuestras ejecuciones con el fin de preparar los datos para los papers!!

```
import wandb
import pandas as pd

api = wandb.Api()
runs = api.runs("usuario/proyecto")

for run in runs:
  if "debug" in run.tags or run.config.get("epochs") < 10:
    continue

  history = run.history() # pd.DataFrame
```

Muchas gracias por vuestra atención