# Clase 003 — Git y GitHub para data scientists

Parte: 0 — Prerrequisitos · Fuente: Pro Git (Chacon & Straub) — caps. 2 y 3 · GitHub docs. · Duración estimada: 120 min.

# Clase 003 — Git y GitHub para data scientists

> *Parte: 0 — Prerrequisitos · Fuente: Pro Git (Chacon & Straub) — caps. 2 y 3 · GitHub docs. · Duración estimada: 120 min.*

## Objetivo

Que el alumno use git no como "botón save" sino como un sistema serio de versionado: commits atómicos con mensajes útiles, branches por feature, PRs con review, y resolución de conflictos sin pánico. Adicionalmente: ignorar correctamente los archivos típicos de DS (datos pesados, notebooks con output, secrets).

## Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Inicializar un repo, hacer commits atómicos con mensajes en formato convencional.
2. Trabajar con branches: crear, cambiar, mergear y resolver un conflicto sin perder código.
3. Configurar .gitignore para un proyecto de DS (datos, .venv, secrets, outputs de notebooks).
4. Abrir y revisar un PR en GitHub desde la línea de comandos con gh.
5. Recuperar trabajo perdido con git reflog (la red de seguridad invisible).

## Temas

| # | Tema | Por qué importa |
|---|---|---|
| 1 | Modelo de git: working tree → staging → re | Sin este modelo mental, todo parece magia. |
| 2 | Commits atómicos + mensajes convencionale | Un commit = un cambio lógico revertible. |
| 3 | Branches y merge vs rebase | Cuándo usar cada uno; por qué no rebasear |
| 4 | .gitignore para data science | Datos, modelos, notebooks con output, .env |
| 5 | Conflictos: anatomía y resolución | <<<<<<<, =======, >>>>>>> y cómo no entrar |
| 6 | Pull Requests + review en GitHub | El review es donde se transfiere conocimie |
| 7 | git reflog — la red de seguridad | Aunque borres una rama, los commits viven |

## Definiciones y características

Repositorio (repo)

: Carpeta con un subdirectorio .git/ que guarda toda la historia. Características: contenido inmutable identificado por SHA-1, ramas son punteros móviles, todo cambio publicado es eterno (aunque borres el commit, vive en reflog 90 días).

Commit

: Snapshot inmutable del estado del repo en un momento. Tiene SHA-1, padre(s), autor, fecha, mensaje. Característica: atómico — debería poder revertirse solo sin romper nada.

Branch (rama)

: Puntero móvil a un commit. Mover el puntero es barato. HEAD apunta a la rama actual. La rama main no es especial; solo es la rama por defecto del proyecto.

Working tree / Staging / Repo / Remote

: Las 4 zonas: working tree (lo que editas) → staging area (lo preparado con git add) → repo local (lo commiteado) → remote (GitHub/GitLab). Cada git mueve cosas entre estas 4 zonas.

Merge vs Rebase

: Merge crea un commit nuevo que junta dos historias (preserva ambas). Rebase reescribe los commits de tu rama encima de otra (historia lineal pero modificada). Característica clave: nunca rebases ramas compartidas — reescribir SHAs rompe a tus compañeros.

Conventional Commits

: Convención que prescribe tipo(scope): descripción. Tipos: feat, fix, docs, refactor, test, chore, perf, style. Beneficio: changelogs y semver automáticos.

## Dataset / recursos

No requiere dataset. El "dataset" son los propios cambios que el alumno hace en archivos de prueba. Para el ejercicio del .gitignore, simulamos archivos típicos de DS (csv pesado, .env, .ipynb_checkpoints/).

## Ejercicios

1. Repo desde cero. git init, crea 3 archivos (README.md, data.csv, notebook.ipynb), haz 3 commits con mensajes en formato tipo: descripción (feat/fix/docs/chore).

2. Branch + conflicto. Crea rama feature/x, modifica una línea en README.md. Vuelve a main, modifica la misma línea distinto. Mergea, resuelve el conflicto a mano.

3. .gitignore profesional. Genera uno que ignore: .venv/, __pycache__/, .ipynb_checkpoints/, .csv en data/raw/, .env, models/.pkl. Verifica con git status que no aparecen.

4. PR desde la CLI. Crea repo en GitHub (con gh repo create), push, crea PR con gh pr create y descripción no trivial.

5. Recuperación. Borra una rama con commits. Recupera el HEAD con git reflog + git checkout <sha> + git switch -c rescate.

## Homework verificable

Repo público en GitHub con: 5+ commits en formato convencional, al menos 1 branch mergeada, un .gitignore de DS completo, README con badges (build status si aplica) y un PR cerrado.

Criterio de aceptación: El historial (git log --oneline) se lee como cambios atómicos coherentes. git status limpio después de un experimento. PR mergeado con descripción legible.

## Errores comunes

| Síntoma / mensaje | Causa y cómo arreglar |
|---|---|
| error: failed to push some refs to 'origin | El remote tiene commits que no tienes loca |

| | |
|---|---|
| fatal: refusing to merge unrelated histori | Estás juntando dos repos sin ancestro comú |
| Hice git reset --hard y perdí mi trabajo | Si fue local y no había commit: perdido. S |
| Please tell me who you are al hacer commit | Falta config global. Fix: git config --glo |
| Commit con archivo enorme; ahora git push | GitHub bloquea blobs >100 MB. Fix: NO bast |
| Mergeé un PR pero ahora hay conflictos en | Alguien mergeó algo antes y tu base local |
| .gitignore no funciona — el archivo sigue | Si el archivo ya estaba trackeado antes de |

## Preguntas frecuentes

¿Merge o rebase?

Regla simple: merge para todo lo público, rebase solo localmente antes de PR para limpiar tus propios commits. Nunca rebases una rama que alguien más usa.

¿Force push (git push -f) es siempre malo?

En main o ramas compartidas: catástrofe. En tu propia rama de feature después de rebase: aceptable. Mejor usar --force-with-lease que falla si alguien más empujó mientras.

¿Cómo deshago el último commit?

Si NO empujaste: git reset --soft HEAD~1 (mantiene cambios staged) o --hard (los borra). Si YA empujaste y quieres revertirlo sin reescribir historia: git revert HEAD (crea commit nuevo que deshace).

¿Squash o no squash al mergear?

Squash = un solo commit final con todo el PR. Bueno para mantener historia limpia en main. Pierdes el detalle de pasos intermedios. Política común: squash en PRs pequeños, merge commit en grandes.

¿Está bien commitear el .venv/ o el data/raw/customers.csv?

NO. .venv/ se reconstruye con requirements.txt. Datos grandes/sensibles van fuera del repo (DVC, S3, etc.) — ver clase 159 Parte 4.

Tengo 30 commits "wip" en mi rama, ¿qué hago antes del PR?

git rebase -i main para entrar al rebase interactivo. Cambia pick por squash (o fixup) en los commits intermedios; quedará un historial limpio.

## Referencias

- Pro Git book — cap. 2 Git Basics, cap. 3 Branching.
- Conventional Commits
- GitHub CLI manual

## Siguiente clase

Clase 004 — Estructura reproducible de proyecto (cookiecutter-data-science)

## Apéndice: notebook (primer bloque)

La mayoría de los ejercicios se hacen en terminal. Este notebook documenta los comandos y verifica el

estado del repo desde Python.

```
import subprocess
from pathlib import Path

def run(cmd):
    r = subprocess.run(cmd, shell=True, capture_output=True, text=True)
    return r.stdout.strip() or r.stderr.strip()

print('git version:', run('git --version'))
print('cwd        :', Path.cwd())
```

## Archivos complementarios

- notebook.ipynb