

# Git y GitHub

Todo lo que necesitás saber para nunca más perder código — desde cero, paso a paso

## EL PROBLEMA QUE RESUELVE

Sin Git, guardar versiones de código se parece a esto:

### LA REALIDAD SIN GIT

```
app.py → app_v2.py → app_BUENA.py → app_definitiva.py →  
app_definitiva_USARESTE.py
```

Y el problema no es solo el nombre de los archivos. Es que no sabés *qué cambió* entre versiones, no podés volver atrás de forma confiable si algo se rompe, dos personas no pueden trabajar en el mismo archivo sin pisarse, y si se rompe la computadora, todo se fue.

Git resuelve todo eso. Guarda **cada versión de tu proyecto** con su contexto: qué cambió, quién lo cambió, cuándo, y por qué.

## ¿QUÉ ES GIT?

Git es un **programa que instalás en tu computadora** — como Word, Chrome o VS Code. No tiene interfaz gráfica propia: se usa desde la terminal, con comandos de texto. Y hace una sola cosa, pero la hace perfectamente: **recordar el historial completo de un proyecto**.

### CÓMO FUNCIONA — LA ANALOGÍA DEL CUADERNO

Imaginá que llevás un cuaderno donde anotás el estado de tu proyecto cada vez que querés "marcar un punto". Git es ese cuaderno. Vos seguís trabajando en tus archivos normalmente. Cuando llegás a un punto importante — arreglaste un bug, terminaste una función, hiciste algo que funciona — le decís a Git "guardá este estado". Git toma una foto de todos los archivos y la archiva. Esas fotos se acumulan y forman una línea del tiempo completa de tu proyecto.

En cualquier momento podés:

- Ver cómo estaba el código en cualquier punto del pasado
- Comparar qué cambió entre dos versiones
- Volver a un estado anterior si algo se rompió
- Ver quién hizo cada cambio y por qué

Git funciona **100% offline** en tu computadora. No necesitás internet para usarlo. La historia de tu proyecto vive en una carpeta oculta llamada `.git/` dentro de tu proyecto. Lo creó Linus Torvalds (el mismo que creó Linux) en 2005, en 10 días.

## ¿QUÉ ES GITHUB?

GitHub es una **página web** (github.com) donde subís tu repositorio de Git a la nube. Es como tener una copia del historial de tu proyecto accesible desde cualquier computadora, compartible con cualquier persona.

Pero GitHub no es solo backup. Agrega herramientas que Git solo no tiene:

- **Interfaz visual:** ves el historial de commits, los cambios línea por línea, quién cambió qué — sin usar la terminal.
- **Pull Requests:** en equipo, no mezclás código directamente. Proponés cambios para que otros los revisen y aprueben antes de integrarlos.
- **Issues:** sistema para trackear bugs, tareas y features. Como un Trello dentro de tu repo.
- **GitHub Actions:** cuando hacés push, GitHub puede correr tests y deployar automáticamente.
- **Tu perfil:** es tu CV de programadora. Las empresas miran tu GitHub para ver qué construiste.

### GIT VS GITHUB — LA DIFERENCIA EN UNA SOLA FRASE

**Git** es el diario de tu proyecto que guardás en tu mochila. **GitHub** es la copia de ese diario guardada en la nube para que nadie lo pierda y todos puedan colaborar. Podés usar Git sin GitHub. No podés usar GitHub sin Git.

**Explorá github.com antes de la clase.** Buscá **python/cpython** (el lenguaje Python) o **microsoft/vscode** (VS Code). Vas a ver los archivos del proyecto, cada commit que se hizo, quién lo hizo y cuándo — exactamente la misma interfaz que vas a usar para tu propio código.

## ¿CÓMO EMPIEZO UN PROYECTO?

Hay dos formas de arrancar, dependiendo de si el proyecto ya existe o lo estás creando vos:

### GIT INIT — PROYECTO NUEVO DESDE CERO

El proyecto no existe todavía. Vos creás la carpeta, le hacés `git init`, y Git empieza a rastrear esa carpeta.

```
$ git init mi-proyecto
Initialized empty Git
repository
```

### GIT CLONE — PROYECTO QUE YA EXISTE EN GITHUB

El proyecto ya está en GitHub. Lo descargás entero — archivos, historial, y configuración de GitHub — con un solo comando.

```
$ git clone https://github.com/
mica/mi-proyecto.git
```

```
in mi-proyecto/.git/
```

```
Cloning into 'mi-proyecto'...  
done.
```

#### ⚠ DESPUÉS DE GIT INIT, TUS ARCHIVOS NO ESTÁN BAJO CONTROL DE GIT TODAVÍA

`git init` solo crea el contenedor. Git ve que los archivos existen pero los ignora — aparecen como "Untracked". Para que empiece a rastrearlos, necesitás el próximo paso: `git add`.

#### ✓ DESPUÉS DE GIT CLONE, PODÉS TRABAJAR DIRECTAMENTE

Entrás a la carpeta con `cd nombre-del-repo` y ya está. Git, el historial, y la conexión a GitHub ya vienen configurados. No necesitás hacer `git init` ni nada más.

## PASO 1 — DECIRLE A GIT QUÉ ARCHIVOS INCLUIR (GIT ADD)

Antes de guardar cualquier versión, tenés que **decirle a Git cuáles de tus archivos querés incluir**. Eso se hace con `git add`. Los archivos que "addeas" pasan a una zona de preparación llamada *staging area* — están listos para ser guardados.

```
bash  
  
# Ver el estado actual — qué sabe Git de mis archivos  
$ git status  
Untracked files:  
  app.py ← Git lo ve pero no lo rastrea todavía  
  utils.py  
  
# Agregar un archivo específico al staging  
$ git add app.py  
  
# O agregar TODOS los archivos modificados de una vez  
$ git add .  
  
# Verifico qué quedó en staging  
$ git status  
Changes to be committed:  
  new file: app.py ← ahora sí está listo para el commit  
  new file: utils.py
```

## PASO 2 — GUARDAR LA FOTO DEL MOMENTO (GIT COMMIT)

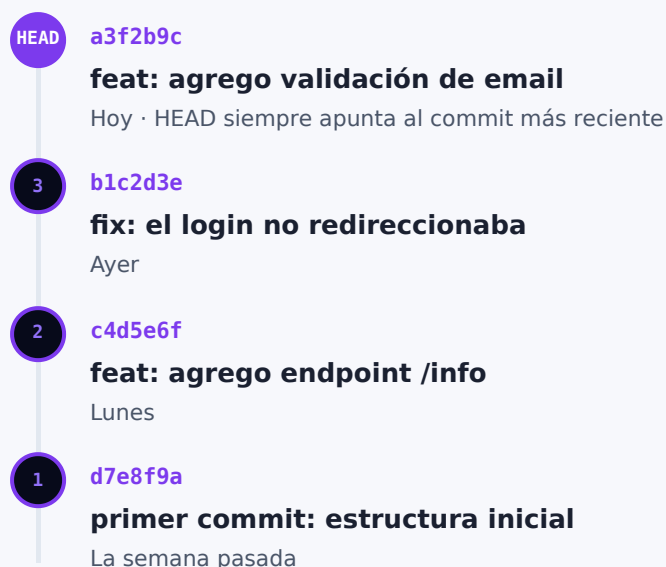
Una vez que tenés los archivos en staging, `git commit` toma esa foto y la guarda **de forma permanente** en el historial de tu proyecto. Cada commit queda guardado para siempre con su fecha, autor y mensaje.

Todo commit necesita un **mensaje** que explique qué hiciste y por qué. Ese mensaje es para vos en el futuro — en 3 meses, cuando mires el historial, vas a querer saber qué cambió sin tener que leer el código.

```
# Guardar el commit con un mensaje descriptivo
$ git commit -m "feat: agrego validación de email"
[main a3f2b9c] feat: agrego validación de email
 2 files changed, 45 insertions(+), 3 deletions(-)

# Ver el historial de commits – una línea por commit
$ git log --oneline
a3f2b9c feat: agrego validación de email ← HEAD, el más nuevo
b1c2d3e fix: el login no redireccionaba
c4d5e6f feat: agrego endpoint /info al backend
d7e8f9a primer commit: estructura inicial
```

#### ASÍ SE VE LA LÍNEA DEL TIEMPO QUE VAN FORMANDO LOS COMMITS



**Convención de mensajes:** Empezá con un tipo seguido de dos puntos: `feat:` para funcionalidad nueva, `fix:` para bugs, `docs:` para documentación, `refactor:` para mejoras sin cambio de comportamiento. Luego describí el QUÉ. Ejemplo: `fix: el formulario de login no redireccionaba después de autenticarse`.

#### ¿POR QUÉ DOS PASOS? EL ÁREA DE STAGING

Te debe haber parecido raro que haya que hacer `git add` Y `git commit` en vez de solo uno. La razón existe: el **área de staging** (también llamada "index") es una zona intermedia donde armás exactamente qué va en cada commit.

Imaginá que trabajaste todo el día y cambiaste 6 archivos: arreglaste un bug en `login.py` y también empezaste una feature nueva en `dashboard.py`. Son dos cambios *distintos* que no tienen nada que ver entre sí. Con el staging podés hacer **dos commits separados** — uno para el bug y otro para la feature — aunque los dos estén en la misma sesión de trabajo.

Sos la fotografía de un evento con 10 personas en la sala. Querés hacer distintas fotos de grupo — no siempre con todos. Con `git add` llamás a las personas que van en esta foto. Con `git commit` sacás la foto. Las que no convocaste siguen disponibles para la próxima foto.

#### UNTRACKED

**Sin rastrear.** El archivo existe pero Git lo ignora. Pasa justo después de `git init` o cuando creás un archivo nuevo. Git te dice "existe, pero no es mío todavía".

#### STAGED

**En el área de staging.** Hiciste `git add`. Este cambio está seleccionado y listo para el próximo `git commit`.

#### COMMITTED

**Guardado en el historial local.** Hiciste `git commit`. El cambio es parte permanente del repositorio local. Falta subirlo a GitHub con `git push`.

### PASO 3 — SUBIR MIS CAMBIOS A GITHUB (GIT PUSH)

Tus commits están guardados en tu máquina, pero *nadie más puede verlos todavía*. Para subirlos a GitHub usás `git push`. Git sabe a dónde subirlos porque tiene guardada la URL de GitHub bajo el nombre `origin` — ese alias se configura automáticamente cuando hacés `git clone`, o lo configurarás vos si empezaste con `git init`.

```
bash

# Ver a dónde apunta "origin" (la URL de GitHub)
$ git remote -v
origin https://github.com/mica/mi-proyecto.git (push)

# Subir los commits a GitHub
$ git push
To https://github.com/mica/mi-proyecto.git
   b1c2d3e..a3f2b9c main → main ← confirmación de qué se subió

# Primera vez que subís una rama nueva, necesitás -u
$ git push -u origin main

# Después de eso, con git push solo ya alcanza
```

### ¿QUÉ PASA SI OTROS HICIERON CAMBIOS? (GIT PULL)

Si estás trabajando en equipo — o vos misma committeaste desde otra computadora —, GitHub puede tener commits que vos todavía no tenés en tu máquina. `git pull` los baja y los une a tu rama local.

```
bash

# Bajar los cambios de GitHub y unirlos a tu carpeta
$ git pull
remote: Counting objects: 5, done.
Updating b1c2d3e..f4a5b6c
```

```
Fast-forward
app.py | 8 +++++---
```

```
# Si no hay nada nuevo, te lo dice
$ git pull
Already up to date.
```

**Regla práctica:** Hacé `git pull` antes de empezar a trabajar cada día. Si no bajás los cambios antes de pushear, Git te va a avisar que hay commits en GitHub que no tenés, y te va a pedir que primero los integres antes de subir los tuyos.

## EL FLUJO COMPLETO — DE PRINCIPIO A FIN

Este es el camino que recorre tu código desde que editás un archivo hasta que llega a GitHub:

|                         |                                                                              |
|-------------------------|------------------------------------------------------------------------------|
| <code>git add</code>    | Tu carpeta (Working Directory)<br>→ <b>Área de Staging</b>                   |
| <code>git commit</code> | Área de Staging<br>→ <b>Repositorio local (.git/ en tu máquina)</b>          |
| <code>git push</code>   | Repositorio local<br>→ <b>GitHub (la nube)</b>                               |
| <code>git pull</code>   | GitHub (la nube)<br>→ <b>Tu carpeta (baja Y une en un paso)</b>              |
| <code>git clone</code>  | GitHub (la nube)<br>→ <b>Tu carpeta (solo la primera vez, descarga todo)</b> |

## ¿QUÉ SON LAS RAMAS? (BRANCHES)

Hasta acá trabajamos en una sola línea del tiempo: cada commit va después del anterior. Pero ¿qué pasa cuando querés probar algo nuevo sin tocar lo que ya funciona? Para eso existen las **ramas** — líneas de tiempo paralelas de tu código.

### LA ANALOGÍA DEL INFORME DE WORD

Tenés el capítulo 3 de tu tesis. Funciona bien. Te piden que experimentes una versión completamente distinta, pero no querés perder la original. ¿Qué hacés? *Guardar como* → *capitulo3\_experimento.docx*. Trabajás en la copia. Si funciona, reemplazás el original. Si no, cerrás la copia y listo.

Las ramas en Git hacen exactamente eso — sin copiar archivos manualmente, con historial, y pudiendo trabajar en equipo en paralelo sin pisarse.

La rama principal se llama `main`. Cuando querés desarrollar algo nuevo, creás una rama propia, trabajás ahí, y cuando está lista, la unís de vuelta a `main`.

```
* a3f2b9c (HEAD → main) merge: uno feature/login con main
|\
| * f1e2d3c (feature/login) mejoro el formulario de login
| * e9f0a1b agrego validación de contraseña
|/
* b1c2d3e fix: el login no redireccionaba — punto donde nació la rama
* c4d5e6f feat: agrego endpoint /info
```

```
bash — flujo de trabajo con ramas

# Ver en qué rama estás (* = rama actual)
$ git branch
* main

# Crear una rama nueva y cambiarte a ella — todo en uno
$ git checkout -b feature/nueva-pantalla
Switched to a new branch 'feature/nueva-pantalla'

# Trabajás con total libertad: add, commit, push...
# Cuando terminás, volvés a main
$ git checkout main

# Unís tu rama a main (merge)
$ git merge feature/nueva-pantalla
Merge made by the 'ort' strategy.
pantalla.py | 34 ++++++
```

## ¿QUÉ ES UN PULL REQUEST?

En el trabajo real, cuando terminás de trabajar en tu rama no hacés el merge directamente — abrís un **Pull Request (PR)** en GitHub. Un PR es una propuesta formal de cambio: le decís al equipo "trabajé en esta rama, hice estos cambios — ¿los revisamos y los integramos a main?"

El equipo puede ver exactamente qué líneas cambiaron, comentar, pedir ajustes, y aprobar. Solo cuando el PR está aprobado se hace el merge. Esto evita que código con bugs entre directamente a la rama principal.

### LA ANALOGÍA

Es como entregar una tarea para corrección antes de que se publique la nota oficial. Vos hacés el trabajo en tu rama, lo subís, y abrís el PR para que lo revisen. Si está bien, se aprueba y se integra. Si hay algo para mejorar, te lo dicen antes de que afecte a todo el equipo.

```
bash — flujo completo con PR
```

```
# 1. Creás tu rama y trabajás
$ git checkout -b feature/nueva-pantalla
# (editás, git add, git commit...)

# 2. Subís la rama a GitHub
$ git push -u origin feature/nueva-pantalla
To https://github.com/mica/mi-proyecto.git
 * [new branch] feature/nueva-pantalla → feature/nueva-pantalla

# 3. En GitHub aparece un banner: "Compare & pull request" – hacés click
# 4. Escribís el título y descripción del PR
# 5. Tus compañeras revisan, comentan, aprueban
# 6. Alguien hace "Merge pull request" – tus cambios entran a main
```

## VOLVER A VERSIONES ANTERIORES

Esta es una de las cosas más poderosas de Git. Hay tres formas según qué necesitás:

### GIT CHECKOUT <HASH> — SOLO MIRAR, SIN CAMBIAR NADA

Querés **ver cómo estaba el código** en un commit anterior. Es como viajar en el tiempo para mirar — no modifica nada.

```
$ git checkout b1c2d3e
You are in 'detached HEAD' state. ← modo lectura, no te asustes

# Para volver al presente
$ git checkout main
```

### GIT REVERT <HASH> — DESHACER SIN BORRAR EL HISTORIAL (LA MÁS SEGURA)

Querés **deshacer los cambios de un commit** sin borrar el historial. Crea un nuevo commit que "cancela" el anterior. Usala cuando ya subiste el código a GitHub.

```
$ git revert a3f2b9c
[main e9f0a1b] Revert "feat: agrego validación de email"
Historia: ... → a3f2 → e9f0 (revert) – el historial queda intacto
```

### GIT RESET --HARD <HASH> — BORRAR COMO SI NUNCA HUBIERA PASADO

Querés **eliminar commits recientes** como si nunca hubieran existido. Reescribe la historia — solo para commits que todavía NO subiste a GitHub.

```
$ git reset --hard b1c2d3e
```

### ⚠ CUIDADO CON --HARD

`git reset --hard` borra permanentemente los cambios que no commiteaste y los commits que venían después del punto al que volviste. No se puede deshacer fácilmente. Usalo solo cuando estás completamente segura.

## REPOS PÚBLICOS, PRIVADOS, Y CÓMO AUTENTICARTE

Cuando creás un repositorio en GitHub podés elegir si es **público** (cualquiera puede verlo y clonarlo) o **privado** (solo lo ven las personas que vos invitás). Pero en ambos casos, para poder hacer `git push` necesitás que GitHub sepa que sos vos.

### REPOSITORIO PÚBLICO

Cualquiera puede **ver y clonar** el código. Pero para hacer `git push` (escribir) igualmente necesitás ser colaboradora o dueña del repo. No alcanza con que sea público.

### REPOSITORIO PRIVADO

Solo lo ven las personas invitadas. Para clonarlo, pushear y verlo en GitHub necesitás ser colaboradora — y autenticarte.

**¿Cómo agrego a alguien como colaboradora?** En el repo de GitHub: *Settings* → *Collaborators* → *Add people*. La persona recibe una invitación por email. Una vez que la acepta, puede clonar, pushear y abrir PRs.

Cuando hacés `git push` a un repo donde tenés permisos, GitHub necesita verificar tu identidad. Hay dos formas de autenticarte:

### OPCIÓN 1 — HTTPS CON TOKEN (LA MÁS FÁCIL PARA EMPEZAR)

Cuando clonás con una URL que empieza en `https://` y hacés push por primera vez, Git te pide usuario y contraseña. Pero GitHub ya no acepta la contraseña normal — necesitás un **Personal Access Token (PAT)**.

- Andá **github.com** → **tu avatar** → **Settings** → **Developer settings** → **Personal access tokens** → **Tokens (classic)**
- Hacé click **Generate new token**, dale un nombre, seleccioná el **repo**, y en **scope** generalo
- Copiá el token — solo aparece una vez. Guardalo en un lugar seguro.
- Cuando Git te pide la contraseña al hacer push, pegás el token en vez de tu contraseña

### OPCIÓN 2 — SSH (LA MÁS CÓMODA A LARGO PLAZO)

SSH usa un par de claves: una clave privada (que queda en tu máquina, nunca la compartís) y una clave pública (que subís a GitHub). Una vez configurado, **nunca más te pide contraseña**.

● ● ● bash — configurar SSH una sola vez

# 1. Generás el par de claves en tu máquina

```
$ ssh-keygen -t ed25519 -C "tu@email.com"
```

Generating public/private ed25519 key pair.

Enter file in which to save the key (~/.ssh/id\_ed25519):

→ Presioná Enter para aceptar el nombre por defecto

# 2. Mostrás la clave pública para copiarla

```
$ cat ~/.ssh/id_ed25519.pub
```

ssh-ed25519 AAAAC3Nza... tu@email.com ← copiás todo esto

# 3. En GitHub: Settings → SSH keys → New SSH key → pegás la clave

# 4. Ahora clonás con la URL SSH (no HTTPS)

```
$ git clone git@github.com:micadevops/mi-proyecto.git
```

Cloning into 'mi-proyecto'... ← sin pedir contraseña

**¿Cómo sé qué URL usar?** En GitHub, al clicar el botón verde "Code" para clonar, podés elegir entre la URL HTTPS y la SSH. Si ya tenés SSH configurado, usá la SSH — es más rápida y no te pide credenciales nunca. Si estás empezando, HTTPS con token es más simple para el primer día.

## REFERENCIA RÁPIDA — TODOS LOS COMANDOS

### 🚩 EMPEZAR

#### git init

Crea un repo en la carpeta actual. Los archivos quedan "untracked" hasta que hagas git add.

#### git clone <url>

Descarga un repo de GitHub con todo: archivos, historial, y origin ya configurado.

#### git config --global user.name "Mica"

```
git config --global  
user.email "mica@..."
```

Configura tu identidad. Hacelo una vez al instalar Git — se guarda en todos tus commits.

### 🕒 EL FLUJO DEL DÍA A DÍA

#### git status

Tu comando más importante. Muestra qué archivos cambiaron, cuáles están en staging, cuáles sin rastrear. Usalo constantemente.

#### git add .

```
git add archivo.py
```

Mueve cambios al staging area. El punto ( `.` ) agrega todo; podés especificar archivos individuales.

|                                            |                                                                                           |
|--------------------------------------------|-------------------------------------------------------------------------------------------|
| <b>git commit -m "mensaje"</b>             | Guarda lo que está en staging como un commit permanente en el historial local.            |
| <b>git push</b><br>git push -u origin main | Sube los commits a GitHub. Con <b>-u origin main</b> la primera vez que creás una rama.   |
| <b>git pull</b>                            | Baja los cambios de GitHub y los une a tu rama local. Hacelo antes de empezar a trabajar. |

#### VER HISTORIAL Y CAMBIOS

|                                        |                                                                                              |
|----------------------------------------|----------------------------------------------------------------------------------------------|
| <b>git log --oneline</b>               | Historial resumido: hash corto + mensaje. El más útil del día a día.                         |
| <b>git log --oneline --graph --all</b> | Como el anterior pero muestra el gráfico de ramas. Útil cuando hay branches.                 |
| <b>git diff</b><br>git diff --staged   | Cambios no en staging. Con <b>--staged</b> : lo que ya está en staging listo para commitear. |
| <b>git show &lt;hash&gt;</b>           | Muestra los cambios de un commit específico: quién, cuándo, y líneas exactas.                |

#### RAMAS

|                               |                                                                 |
|-------------------------------|-----------------------------------------------------------------|
| <b>git branch</b>             | Lista todas las ramas. La que tiene <b>*</b> es la rama actual. |
| <b>git checkout -b nombre</b> | Crea una rama nueva y te cambia a ella en un solo comando.      |
| <b>git checkout main</b>      | Cambia a la rama main (o a cualquier otra que especifiques).    |
| <b>git merge nombre</b>       | Une la rama indicada a la rama donde estás actualmente.         |

#### VOLVER ATRÁS

|                                      |                                                                                                              |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <b>git checkout &lt;hash&gt;</b>     | Viaja al estado de ese commit para verlo. No modifica la historia. Usá <b>git checkout main</b> para volver. |
| <b>git revert &lt;hash&gt;</b>       | Crea un commit que deshace el indicado. La forma segura cuando ya pusheaste a GitHub.                        |
| <b>git reset --hard &lt;hash&gt;</b> | Mueve al commit indicado borrando todo lo que vino después. Solo para commits no pusheados.                  |