Programación en Python

Clase 12: Git & GitHub

03/06/2026

Git, GitHub... ¿qué son?

Git, GitHub... ¿qué son?

Git es un sistema de control de versiones distribuido.

Git, GitHub... ¿qué son?

Git es un sistema de control de versiones distribuido.

GitHub es una plataforma en línea para alojar proyectos gestionados con Git.

Git, GitHub... ¿qué son?

Git es un sistema de control de versiones distribuido.

GitHub es una plataforma en línea para alojar proyectos gestionados con Git.

Git puede usarse sin GitHub.

Git, GitHub... ¿qué son?

Git es un sistema de control de versiones distribuido.

GitHub es una plataforma en línea para alojar proyectos gestionados con Git.

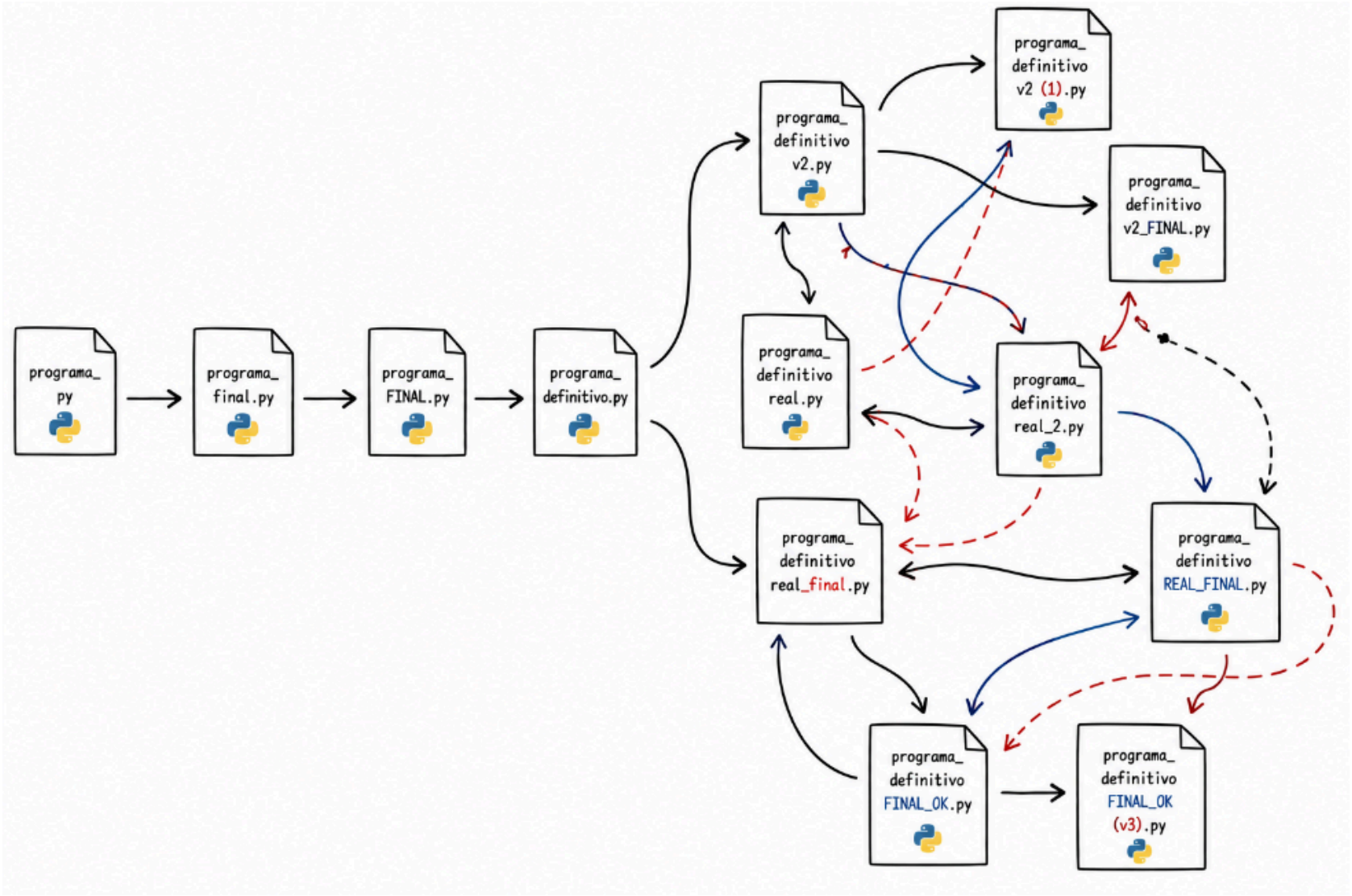
Git puede usarse sin GitHub.

GitHub no reemplaza a Git, lo complementa.

Git

A quién no le pasó

A quién no le pasó



Sistema de Control de Versiones (VCS)

Un **VCS** permite:

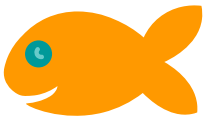
- Mantener un registro del historial de cambios.
- Documentar **qué** cambió y **cuándo** cambió.
- Volver a versiones anteriores si algo sale mal.
- Coordinar el trabajo cuando participan varias personas.

Sistema de Control de Versiones (VCS)

Un **VCS** permite:

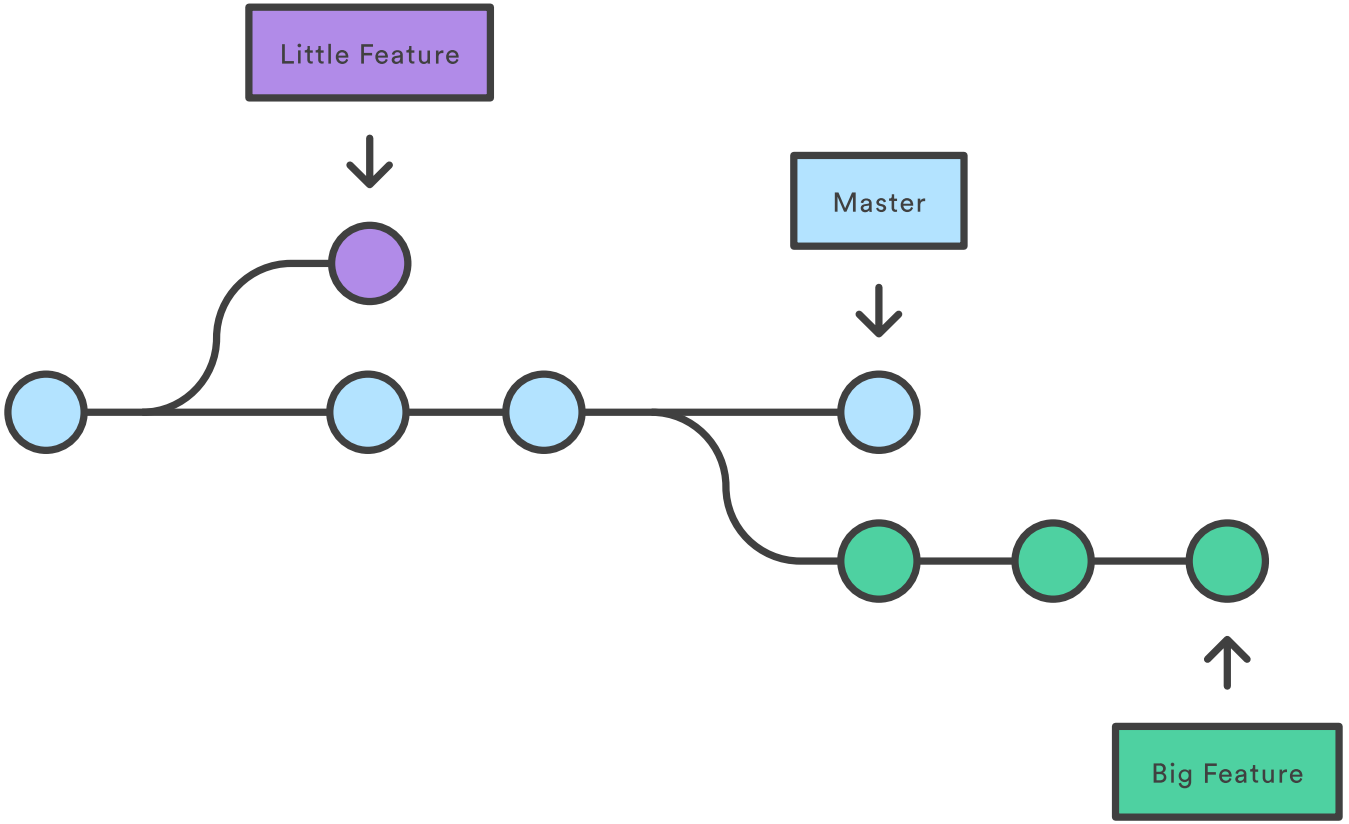
- Mantener un registro del historial de cambios.
- Documentar **qué** cambió y **cuándo** cambió.
- Volver a versiones anteriores si algo sale mal.
- Coordinar el trabajo cuando participan varias personas.

Algunos ejemplos:



El proyecto como historial

Git guarda “fotografías” del estado del proyecto en distintos momentos.



Cada punto representa un ***commit***: una instantánea de los cambios guardados.

Sobre Git

Existen distintos sistemas de control de versiones. Lo distintivo de Git es que es:

- **Distribuido**: cada persona tiene una copia completa del repositorio y de su historial.
- **Eficiente**: las operaciones suelen ser rápidas y el almacenamiento es liviano.
- **Completo**: permite registrar cambios, comparar versiones, crear ramas y fusionarlas.
- ***Open source***: es libre y sigue evolucionando desde su aparición en 2005.

Página oficial: https://git-scm.com

Instalación

Demostración en vivo

Git Bash

Git Bash es una consola para usar Git y comandos de terminal estilo Linux dentro de Windows.

Comandos básicos de Bash

```
ls
cd <directorio>
cd ..
pwd
mkdir <nombre>
touch <nombre>
rm <nombre>
cp <nombre> <directorio>
mv <nombre> <directorio>
```

Configuración mínima

Todas las acciones en Git necesitan un autor.

Se utiliza el comando `git config` de la siguiente manera:

```
git config --global user.name <nombre>
git config --global user.email <email>
```

Si se omite el *flag* `--global`, la configuración aplica solo al repositorio donde se realiza.

Para ver qué otras cosas se pueden hacer con `git config`:

```
git config -h
```

Repositorio

Un **repositorio** es un lugar donde se guarda un proyecto, incluyendo sus archivos, carpetas y el historial de cambios.

Repositorio

Un **repositorio** es un lugar donde se guarda un proyecto, incluyendo sus archivos, carpetas y el historial de cambios.

Git se encarga del seguimiento del historial de cambios y nos permite navegar entre versiones de nuestro proyecto.

Repositorio

Un **repositorio** es un lugar donde se guarda un proyecto, incluyendo sus archivos, carpetas y el historial de cambios.

Git se encarga del seguimiento del historial de cambios y nos permite navegar entre versiones de nuestro proyecto.

Las **ramas** hacen posible trabajar de forma colaborativa e independiente: se pueden realizar cambios, probar ideas o corregir errores sin afectar directamente el proyecto principal.

Repositorio

Un **repositorio** es un lugar donde se guarda un proyecto, incluyendo sus archivos, carpetas y el historial de cambios.

Git se encarga del seguimiento del historial de cambios y nos permite navegar entre versiones de nuestro proyecto.

Las **ramas** hacen posible trabajar de forma colaborativa e independiente: se pueden realizar cambios, probar ideas o corregir errores sin afectar directamente el proyecto principal.

Git se encarga de que la **fusión** de cambios sea amena y segura.

Inicialización de un repositorio

Para inicializar un repositorio se debe ejecutar `git init` en la raíz del proyecto:

```
git init
```

```
Initialized empty Git repository in /prueba/.git/
```

Inicialización de un repositorio

Para inicializar un repositorio se debe ejecutar `git init` en la raíz del proyecto:

```
git init
```

```
Initialized empty Git repository in /prueba/.git/
```

Se crea una carpeta oculta llamada `.git` que contiene todas las referencias asociadas al sistema de control de versiones.

```
ls .git
```

```
branches  config  description  HEAD  hooks  info  objects  refs
```

Guardado de cambios

En Git no alcanza con modificar archivos:

1. Hay que elegir qué cambios guardar.
2. Luego, hay que “guardarlos” explícitamente

Guardado de cambios

En Git no alcanza con modificar archivos:

1. Hay que elegir qué cambios guardar.
2. Luego, hay que “guardarlos” explícitamente

```
git status
git add <archivo>
git add .
git commit -m "<mensaje>"
```

- `git status` muestra el estado actual del repositorio.
- `git add` agrega cambios al área de *stage*.
- `git commit` crea la fotografía del estado preparado.
- El mensaje debe describir de manera clara qué se guardó.

El área de *stage*

El área de *stage* es una zona intermedia donde se preparan los cambios del próximo commit.

```
archivos modificados  ->  stage  ->  commit
```

Git solo guarda en el commit los cambios que fueron agregados con `git add`.

Consulta de estado

Para saber qué está pasando en el repositorio usamos principalmente:

```
git status
git log
```

- `git status` muestra archivos modificados, eliminados, agregados o no rastreados.
- También indica qué cambios ya están en *stage* y cuáles no.
- `git log` muestra la lista de commits realizados.
- Cada commit tiene un **hash** que lo identifica de manera única.

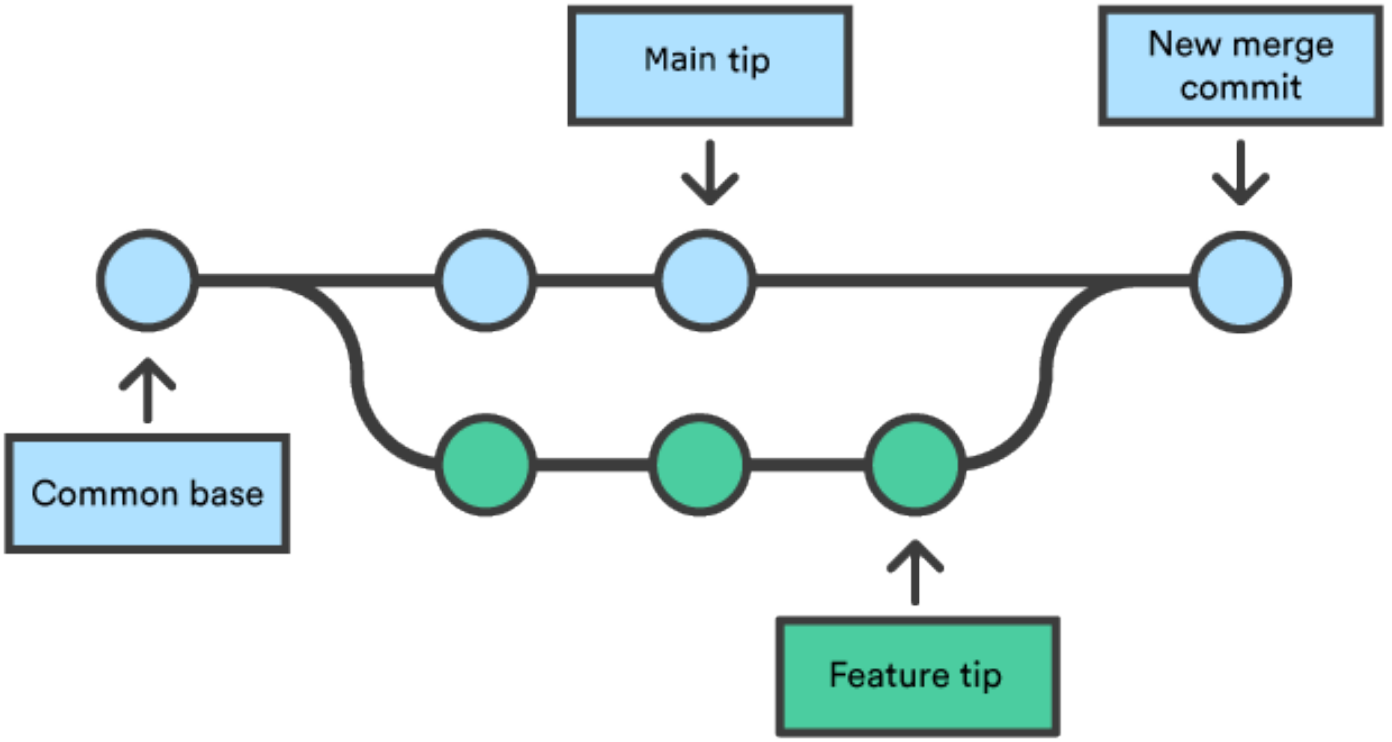
Ramas

Una **rama** es una línea de desarrollo independiente dentro del repositorio.

- Permite trabajar en una parte del proyecto sin afectar directamente a otras.
- Sirve para probar ideas, desarrollar funcionalidades o corregir errores.
- Cada rama tiene su propio conjunto de cambios y commits.
- Cuando el trabajo está listo, puede fusionarse con otra rama.

Ramas

Una rama parte de un *commit*, avanza de manera separada y luego puede integrarse.



Manipulación de ramas

Comandos básicos para crear y movernos entre ramas:

```
git branch
git branch <nombre_rama>
git switch <nombre_rama>
git switch -c <nombre_rama>
git checkout -b <nombre_rama>
```

- `git branch` lista las ramas existentes.
- `git branch <nombre_rama>` crea una rama nueva.
- `git switch <nombre_rama>` cambia a otra rama.
- `git switch -c` crea una rama y se mueve a ella.

Fusión de ramas

En algún momento necesitamos integrar el trabajo de una rama en otra.

Usaremos `git merge`.

```
git merge <nombre_rama>
```

- Hay que estar situados en la rama de destino.
- Git toma los cambios de la otra rama y los aplica sobre la actual.
- La fusión combina los historiales de ambas líneas de trabajo.
- Después se puede verificar el resultado con `git status` o `git log`.

Eliminación de ramas

Cuando una rama ya cumplió su objetivo, puede eliminarse:

```
git branch -d <nombre_rama>
git branch -D <nombre_rama>
```

- `-d` intenta borrar la rama de manera segura.
- `-D` fuerza la eliminación.
- Conviene borrar ramas cuando ya fueron integradas o dejaron de ser útiles.

El archivo `.gitignore`

A veces hay archivos que no queremos incluir en los commits:

- Información delicada que no debería versionarse.
- Archivos temporales.
- Resultados generados automáticamente.
- Configuraciones locales.

El archivo `.gitignore`

A veces hay archivos que no queremos incluir en los commits:

- Información delicada que no debería versionarse.
- Archivos temporales.
- Resultados generados automáticamente.
- Configuraciones locales.

El archivo `.gitignore` se ubica en la raíz del proyecto y contiene **reglas de exclusión**. Por ejemplo:

```
archivo_temporal.txt
carpeta_temporal/
*.log
**/temp
```

El archivo `.gitignore`

A veces hay archivos que no queremos incluir en los commits:

- Información delicada que no debería versionarse.
- Archivos temporales.
- Resultados generados automáticamente.
- Configuraciones locales.

El archivo `.gitignore` se ubica en la raíz del proyecto y contiene **reglas de exclusión**. Por ejemplo:

```
archivo_temporal.txt
carpeta_temporal/
*.log
**/temp
```

El archivo `.gitignore` sí debe agregarse al repositorio.

GitHub

GitHub: de local a remoto

Hasta ahora trabajamos con Git en nuestra computadora.

GitHub agrega un repositorio remoto: una copia del proyecto alojada en un servidor.

- Permite compartir código con otras personas.
- Sirve como punto común para sincronizar cambios.
- Conserva el historial de Git también en la nube.
- Es útil tanto para equipos como para proyectos individuales.

Repositorios remotos

En GitHub, los proyectos se guardan en repositorios.

- Un repositorio público puede ser visto por cualquier persona.
- Un repositorio privado solo es accesible para quienes tengan permiso.
- Cada repositorio tiene una URL propia.
- La URL suele seguir el formato `https://github.com/<usuario>/<repositorio>`.

GitHub también suma herramientas de colaboración: issues, pull requests, revisión de código y acciones automáticas.

Repositorio personal y repositorio de proyecto

GitHub permite crear distintos tipos de repositorios.

- El repositorio personal usa el mismo nombre que el usuario.
- Ese repositorio define la página principal del perfil.
- Un repositorio de proyecto guarda el código de una aplicación, análisis, paquete o trabajo específico.
- Para un proyecto existente en local, se puede crear un repositorio vacío en GitHub y vincularlo después.

Archivos habituales en GitHub

Al crear un repositorio, GitHub puede inicializar algunos archivos útiles.

- `README.md`: documentación inicial del proyecto.
- `.gitignore`: reglas para no versionar archivos innecesarios o sensibles.
- Licencia: condiciones de uso, distribución y colaboración.

`README.md` se escribe normalmente en Markdown, un formato simple para estructurar texto, enlaces, listas, tablas e imágenes.

Autenticación SSH

Para interactuar con GitHub desde la terminal necesitamos autenticarnos.

SSH utiliza dos claves:

- Una clave privada, que queda en nuestra computadora.
- Una clave pública, que agregamos a la cuenta de GitHub.

Con esa relación, GitHub puede reconocer que nuestra computadora está autorizada para acceder a los repositorios correspondientes.

https://docs.github.com/es/authentication/connecting-to-github-with-ssh

Conectar local y remoto

Si ya tenemos un proyecto con Git en local, debemos asociarlo al repositorio remoto.

```
git remote add origin git@github.com:<usuario>/<repositorio>.git
```

`origin` es el nombre habitual de la referencia remota.

La primera subida suele indicar también rama y remoto:

```
git push -u origin main
```

El `-u` deja configurado el destino para futuros `git push`.

Clonar un repositorio

Si el proyecto ya existe en GitHub, podemos traerlo a nuestra computadora.

```
git clone <URL>
```

Clonar no es lo mismo que descargar un `.zip`.

- Trae archivos, ramas, etiquetas e historial.
- Deja el proyecto listo para seguir usando Git.
- Puede hacerse con HTTPS o SSH.
- SSH suele ser más cómodo cuando ya configuramos autenticación.

Sincronización remota

En un proyecto compartido, el repositorio remoto puede cambiar mientras trabajamos en local.

```
git fetch
git pull
```

- `git fetch` descarga información del remoto, pero no aplica cambios sobre nuestros archivos.
- `git pull` descarga cambios y los fusiona con nuestra rama local.
- Si aparecen conflictos, hay que resolverlos antes de continuar.
- La primera vez puede ser necesario definir cómo combinar cambios:

```
git config pull.rebase false
```

Subir código a GitHub

El flujo local sigue siendo el mismo: modificar, preparar y crear commit.

```
git status
git add <archivo>
git commit -m "<mensaje>"
git push
```

`git push` envía los commits locales al repositorio remoto.

Si otras personas subieron cambios antes, Git puede rechazar el `push` hasta que sincronicemos con `pull`.

Forks

Un **fork** es una copia de un repositorio creada en nuestra cuenta de GitHub.

Se usa cuando:

- No tenemos permisos de escritura en el repositorio original.
- Queremos proponer cambios sin modificar directamente el proyecto base.
- Queremos experimentar o evolucionar el proyecto por nuestra cuenta.

Después de hacer el fork, podemos clonarlo, modificarlo, hacer commits y subir cambios a nuestra copia.

Pull Requests

Una **pull request** es una solicitud para que revisen e integren nuestros cambios.

En el flujo con fork:

- Hacemos un fork del repositorio original.
- Clonamos nuestra copia.
- Modificamos archivos y creamos commits.
- Subimos los cambios con `git push`.
- Abrimos una pull request hacia el repositorio original.

Quien mantiene el proyecto revisa la propuesta y, si corresponde, hace el merge.

Resumen GitHub

Proyecto propio:

```
crear repo remoto -> git remote add -> git push -> git pull / git fetch
```

Proyecto ajeno:

```
fork -> git clone -> cambios -> git add / commit / push -> pull request
```

Git sigue registrando el historial.

GitHub agrega el punto remoto donde se comparte, sincroniza y revisa el trabajo.