# Programación para Ciencia de Datos

Francisco Javier Mercader Martínez

# Índice

# Tema 1: Control de Versiones & Desarrollo basado en Test

## 1.1) Metodologías de Desarrollo de Software

### 1.1.1) Introducción

A lo largo de la historia, se han descrito una serie de metodologías para mejorar y facilitar el proceso de desarrollo software.
Cada metodología divide el trabajo en fases, cada una con un conjunto distinto de actividades.
El proceso de dividir el trabajo de desarrollo, normalmente en distintas fases, se conoce como **metodología de desarrollo de software.**
Estas diferentes fases de trabajo pueden incluir:

- Especificación de entregables o artefactos
- Desarrollo y verificación del código con respecto a la especificación
- Despliegue del código a sus clientes finales o entorno de producción.

### 1.1.2) Tipos de metodologías

- Metodología clásica - En cascada

Es un proceso de gestión de proyectos que hace hincapié en una progresión secuencial de una etapa del proceso a la siguiente.

Originario de las industrias manufacturera y de la construcción, y adoptado más tarde por la ingeniería de hardware.

Las etapas originales eran la especificación de requisitos, el diseño, la implementación, la integración, las pruebas, la instalación y el mantenimiento.

El progreso se visualizaba como un flujo de una etapa a otra (de ahí el nombre).

- Metodologías ágiles - Scrum

Es un grupo de metodologías diseñadas para ser más ligeras y flexibles que las metodologías clásicas.

Scrum utiliza ciclos de desarrollo predefinidos llamados *sprints*, normalmente de entre una semana y un mes de duración.

- Comienzan con una reunión de planificación del sprint para definir los objetivos y terminan con una revisión del sprint y una retrospectiva del sprint para discutir el progreso y cualquier problema que haya surgido durante ese sprint.

En estas reuniones, son facilitadas por el *Scrum master*.

Con dichas metodologías en mente, durante las fases de desarrollo e implantación de software surgen cuestiones que el equipo de desarrollo debe de manejar con cuidado.

- ¿Cómo se van a registrar los cambios realizados en el código fuente?
- ¿Cómo deshacer determinados cambios y volver a una versión anterior del proyecto?
- ¿Podemos llevar en paralelo diferentes versiones del proyecto?
- ¿Cómo podemos planificar las pruebas que hagamos a nuestro código?
- ¿Qué tipo de pruebas debemos de implementar? ¿A qué nivel?

- En el resto del tema veremos las herramientas que van a permitir responder y gestionar dichas herramientas en un proyecto de desarrollo del software.

## 1.2) Sistema de Control de Versiones

### 1.2.1) Introducción

Un Sistema de Control de Versiones (en adelante SCV), es un software que controla y organiza las distintas revisiones que se realicen sobre uno o varios documentos

Pero, ¿qué es una revisión?

- Una revisión es un cambio realizado sobre un documento, por ejemplo, añadir un párrafo, borrar un fragmento o algo similar.

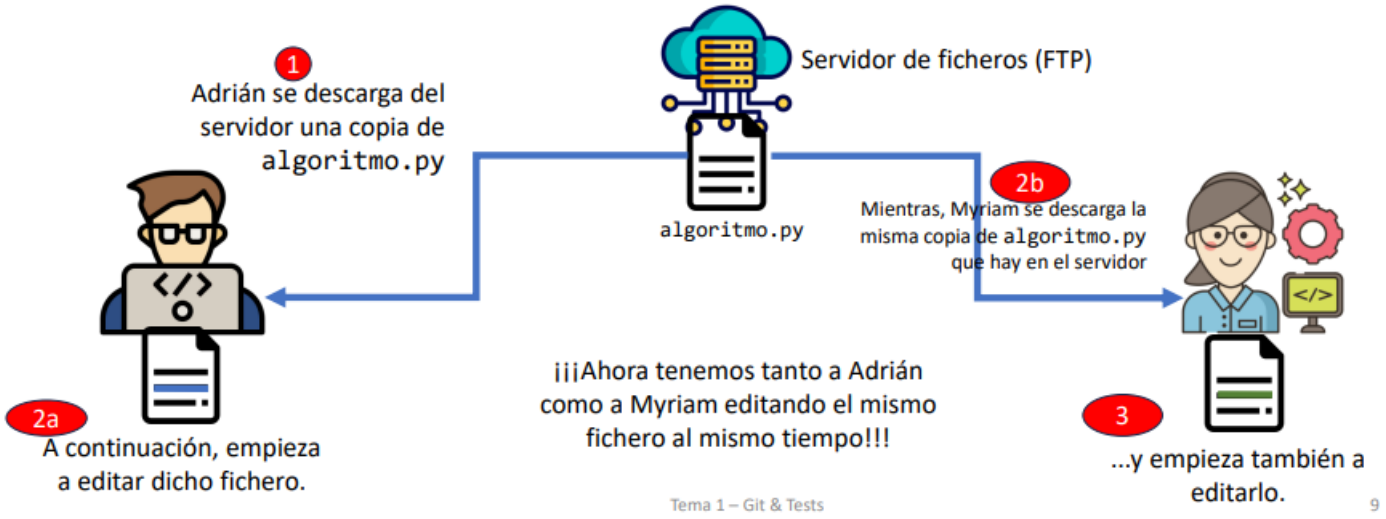

Un SVC guarda el historial de las distintas modificaciones sobre un fichero.

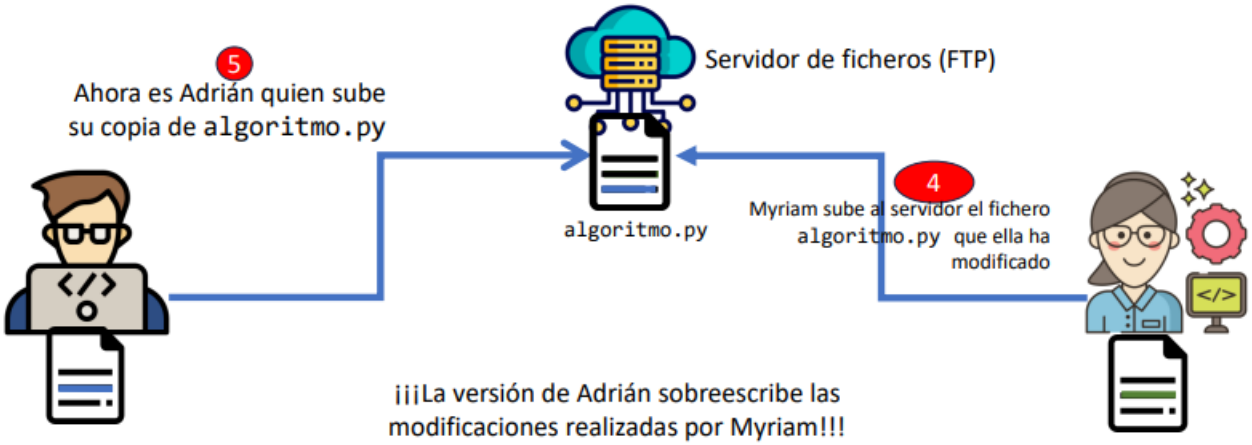
En cualquier momento podemos restaurar la revisión que queramos de un fichero.

Permite mantener una copia de seguridad de todas las modificaciones realizadas sobre un fichero, lo cual nos facilita la tarea de deshacer algo que esté mal.

Su mayor potencial surge cuando el desarrollo se hace en grupo.

- Riesgos del desarrollo en grupo sin SCV






### 1.2.2) Tipos

- Sistema Centralizado
  - Funcionan como un entorno clásico Cliente- Servidor
    - → Servidor: alojara el repositorio del proyecto, con toda la información de los cambios, ficheros binarios añadidos, ...
    - → El cliente trabaja con una "copia de trabajo" del servidor. Es una copia de cómo estaba el servidor en una revisión determinada.
  - El desarrollador hace cambios sobre esa copia de trabajo, y cuando considera que ha terminado con esa modificación la sube (`commit`) al servidor, el cual se encargará de fundir esos cambios en el repositorio.
  - Según la forma de controlar conflictos sobre un fichero podemos distinguir:
    - → Bloqueo de archivo: cuando alguien está trabajando con un archivo, se bloquea su acceso para el resto de usuarios.
    - → Fusión de versiones: controla qué líneas se han modificado por cada usuario.
- Sistema Distribuido
  - Similar a un sistema *Peer-to-Peer* (P2P).
  - La copia de trabajo de cada cliente es un repositorio en sí mismo, una rama nueva del proyecto central.
  - La sincronización de las distintas ramas se realiza intercambiando "parches" con otros clientes del proyecto.
  - No hay una copia original del código del proyecto, solo existen las distintas copias de trabajo.
  - Operaciones como los `commits`, no necesitan de una conexión con un servidor central.
    - → Dicha conexión solo es necesaria al "compartir" tu rama con otro cliente del sistema.
  - Cada copia de trabajo es una copia remota del código fuente y de la historia de cambios, dando una seguridad muy natural contra la pérdida de los datos.

### 1.2.3) Git

#### 1.2.3.1) Introducción

Escrito en C y en gran parte construido para trabajar en el kernel de Linux.

Su principal potencial es su modelo de ramas (lo veremos más adelante).

Diseñado para operar en local de forma rápida y sin conexión.

- Pocos comandos que accedan al servidor

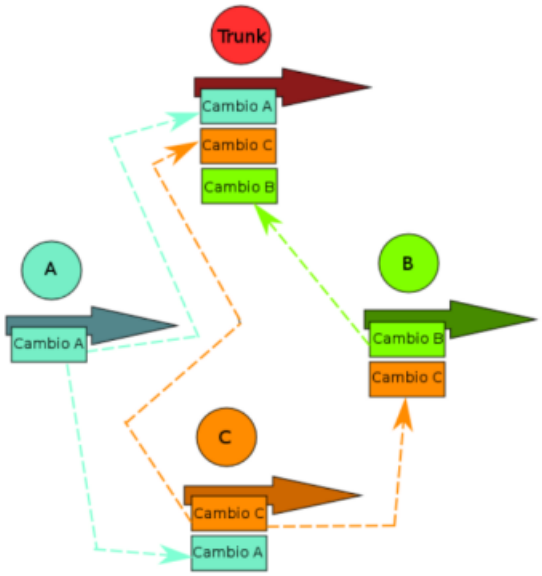
Los repositorios ocupan muy poco espacio.

### 1.2.3.2) Estructura general

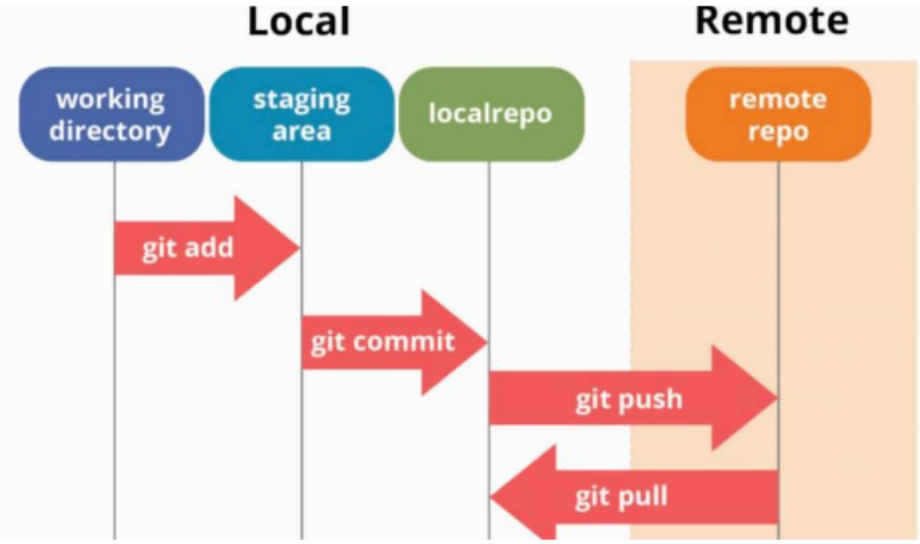
Cada desarrollador tiene su repositorio local.

Si un usuario realiza un cambio, lo hace localmente y cuando crea oportuno puede compartirlo con el resto de usuarios.

Existe una rama principal (Trunk) en la cual todos los usuarios registran los cambios realizados obteniendo de esta forma la versión final.




### 1.2.3.3) Git Workflow

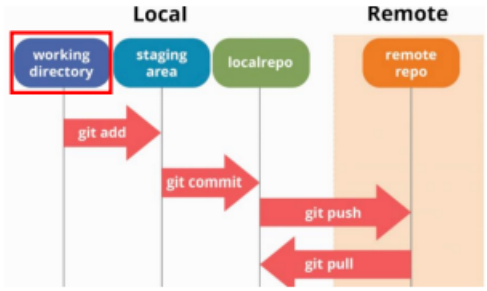


El primer paso es generar un *repositorio Git* asociado a nuestro directorio de trabajo (`working directory`) mediante el comando con `git init`.

- Esto creará dos directorios virtuales, un almacén de objetos (`.git`) y un área de montaje (o Index) que permitirán registrar los cambios realizados en el `working directory`.

En este punto hemos creado el repositorio Git, pero este se encuentra vacío.

- En la figura de la derecha, el fichero `index.html` está en el directorio de trabajo pero **no** en el repositorio...

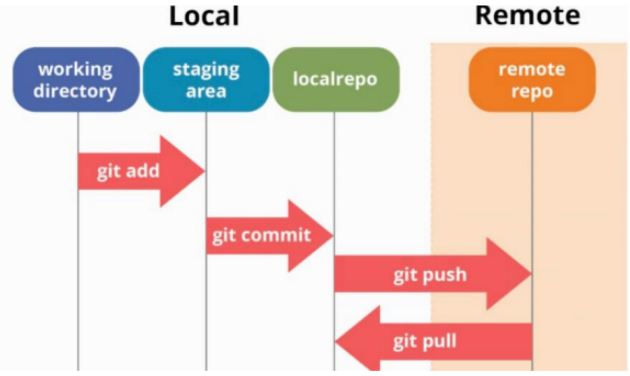




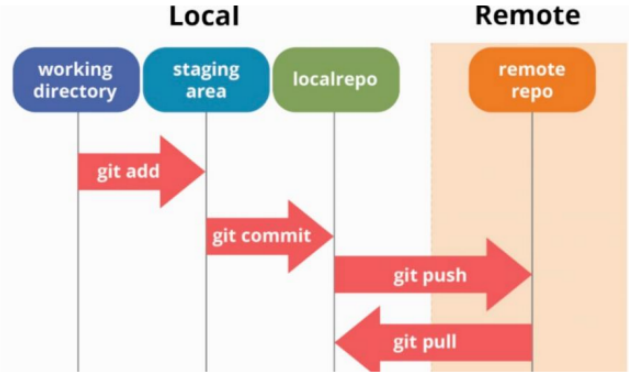
Con `git init` asociamos a nuestro directorio de trabajo donde tenemos todos los archivos de un proyecto:

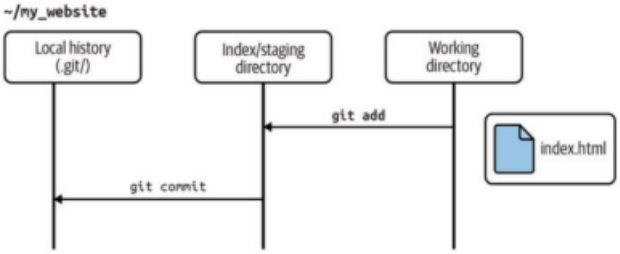
1) Nuestro repositorio local
2) Nuestro repositorio `git`

Para añadir los ficheros del `working directory` que queramos tener controlados dentro del repositorio Git, deberemos ejecutar el comando `git add` (Ej: `git add index.html`). Esto deja el archivo en una fase o área preparatoria (`staging area`).
Con esto indicamos a Git que queremos incluir la versión más reciente del fichero como parte de la revisión del repositorio.
De forma análoga, podemos indicar a git que queremos quitar un determinado fichero del repositorio mediante `git rm`.

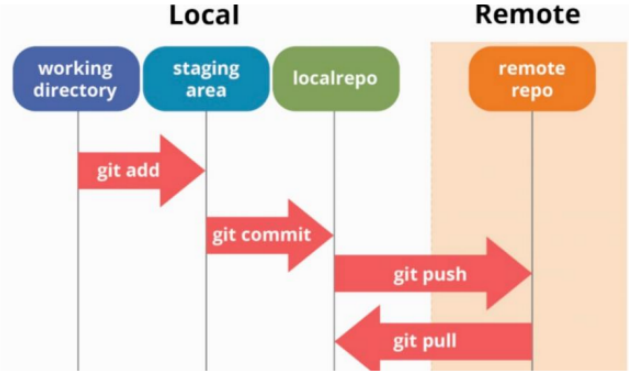



El siguiente paso lógico es consignar el nuevo fichero en el repositorio mediante el comando `git commit`.
Todos estos comandos lo que han hecho ha sido mover el fichero `index.html` del directorio de trabajo al directorio índice (`index`) y finalmente al subdirectorio con el historial local.



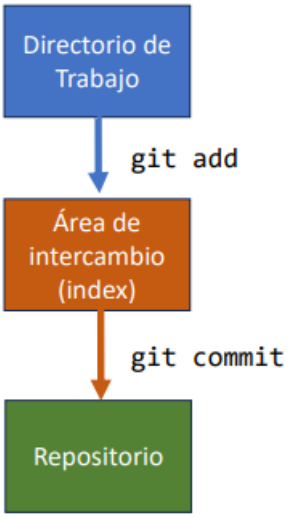
El *commit* que hemos realizado anteriormente es local, para actualizar la rama principal (Trunk) que se encuentra en un servidor deberemos usar el comando `git push`.
También, podríamos bajarnos los cambios que se realizado en dicha rama principal con `git pull`.



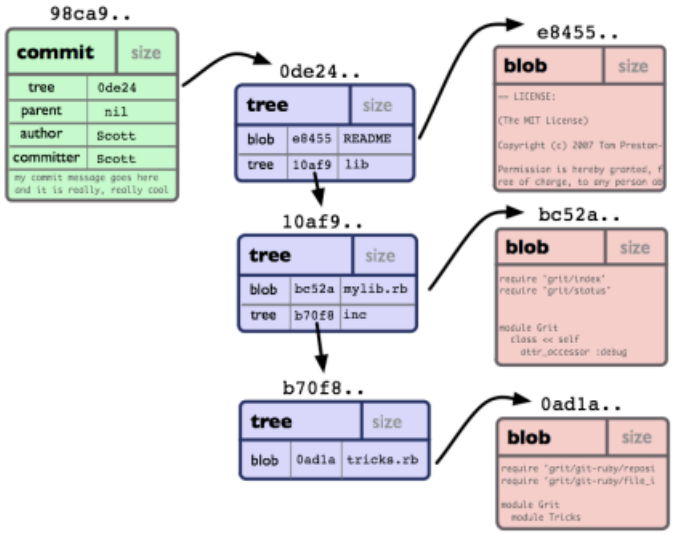

### 1.2.3.4) Componentes de un repositorio Git

Dentro de un repositorio Git nos encontramos con dos estructuras de datos

- El almacén de objetos: contiene los ficheros de datos originales y todos los mensajes de log, fechas y otra información necesaria para reconstruir el estado del proyecto en una determinada fecha.
  - Tiende a crecer de tamaño conforme editamos ⟶ Git comprime dicho almacén en *packfiles*.
  - Cada fichero se almacena como un *blob* y se indexa por su hash SHA-1 por lo que es muy difícil encontrar dos objetos con el mismo nombre.
  - Se copia a un nuevo repositorio cuando se ejecuta `git clone`.
- Área de intercambio o Índice: es el área intermedia donde se puede configurar el "aspecto" que tendrá la entrega antes de hacer `git commit`. Es privado del repositorio Git al cual pertenece.




```
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb
```




### 1.2.3.5) Gestión de ramas

Una rama es una línea de desarrollo que existe de forma independiente a otra, pero que comparte una historia común en algún punto temporal anterior.

Se puede decir que una rama siempre nace como una copia de algo y a partir de ahí pasa a generar su propia historia.


Git permite generar muchas ramas (branches) dentro de un proyecto (y por tanto múltiples líneas de desarrollo).

Una rama se implementa en git como un puntero a un commit ya existente.

Git lleva un control de los commits realizados sobre dicha rama de forma independiente.

Existen múltiples razones que justifican el uso de ramas dentro de un proyecto:

- Para representar diferentes estados en un proyecto (estable, en desarrollo, release candidate, etc.)
- Para representar una nueva versión del proyecto (porque sabes que determinados clientes van a querer seguir usando la versión anterior).
- Permite centrarse en una determinada funcionalidad del producto que estemos desarrollando.

Cuando se crea un repositorio, se crea una rama principal con nombre `master`.

Aunque un proyecto pueda contener múltiples ramas, un desarrollador trabajará sobre una rama en concreto en un momento determinado.

En cada momento, se tendrá una única rama activa.

- Inicialmente, al crear un repositorio, la rama master constituirá la rama activa.

Cada vez que ejecutemos `git commit` se aplicará sobre la rama activa.

El nombre de la rama hace en realidad referencia al `commit` más reciente realizado (`HEAD` en terminología `git`)

- En la figura de la derecha vemos una rama `development` que se ha fusionado con la rama `main`, de ahí que solo haya un `HEAD`.

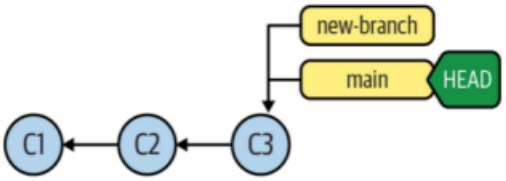
### 1.2.3.6) Creación de ramas

Para crear una rama derivada de la rama activa en la que nos encontremos debemos ejecutar el comando `git branch <nombre_de_la_rama> <punto_inicial>`.

- Por defecto `punto_inicial` hace referencia al HEAD de la rama activa.

Al ejecutar dicho comando simplemente hemos creado un nueva rama **pero** no nos hemos movido a la misma (el HEAD sigue apuntando a la rama origen)

En el ejemplo de la figura hemos creado una nueva rama `new-branch` pero la rama activa sigue siendo `main`.



Para poder movernos a una rama que hayamos creado previamente deberemos ejecutar
`git checkout <nombre_de_la_rama>`

Podemos obtener un listado de todas las ramas que tenemos en nuestro proyecto con el comando `git branch`.

Existe la posibilidad de crear una rama y moverse automáticamente a ella mediante el comando
`git checkout -b <nombre_de_la_rama>`

Para eliminar una rama podemos ejecutar `git branch -d <nombre_de_la_rama>`

- **No debe de ser la rama activa.**