Programación para Ciencia de Datos

Sesión 9: 25 de febrero de 2025

Git

- Git es un **sistema de control de versiones distribuido** (VCS, version control system)
- Fue creado por Linus Torvalds (sí... ese al que se le acuña la creación de LINUX) en 2005
- Permite rastrear cambios en el código durante el desarrollo de software
- Facilita la colaboración entre múltiples desarrolladores
- Mantiene **diferentes versiones** de un proyecto y **permite revertir cambios si es necesario**.
- Se usa ampliamente en el desarrollo de software, integrándose con plataformas como **GitHub, GitLab, Bitbucket y Git Kraken.**

Git

Instalación

- En Windows:
- https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
- En UBUNTU:
- sudo apt update
- sudo apt install git
- En MAC:
- git --version
- brew install git

Git

La principal función de Git es manejo de versiones

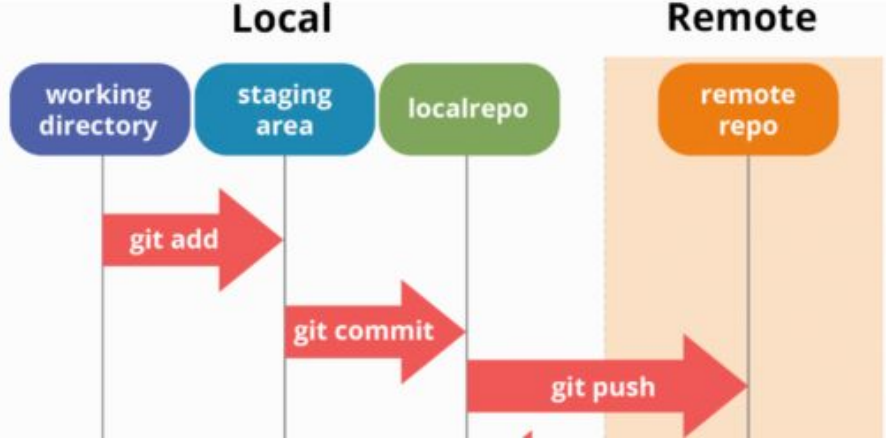
Es decir "guardar cambios"

Tiene 3 fases importantes:

1. status: Revisar qué archivos han sido modificados
2. add: Avisarle a Git que queremos que revise el archivo en cuestión
3. commit: Confirmar los cambios, i.e. guarda un snapshot.

Hay un cuarto paso opcional

push: Enviar los cambios al repositorio remoto



Git

Para verificar el estado de los archivos

```
git status
```

Archivos en **rojo** = No están rastreados (Git aún no los ha agregado).
Archivos en **verde** = Están en staging (listos para ser confirmados).

Para agregar un archivo específico:

```
git add nombre_del_archivo
```

Para agregar todos los archivos modificados:

```
git add .
```

Para confirmar los cambios (commit)

```
git commit -m "Mi mensaje fabuloso que me guía para saber qué cosas guardé"
```

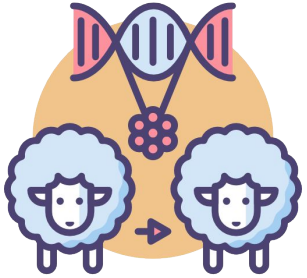
Git

- Pero bueno, bueno... es demasiado pretencioso que Git esté atento a los cambios de toda nuestra computadora
- Hay que acotarle lo que queremos que esté monitoreando
- Esto se hace a través de la creación de un repositorio
- Un repositorio no es más que una carpeta de archivos con la característica especial de que Git la está monitoreando

¿Cómo convertimos una carpeta “normalita” en un repositorio?

`git init` → Inicializa un nuevo repositorio Git en una carpeta.

Git

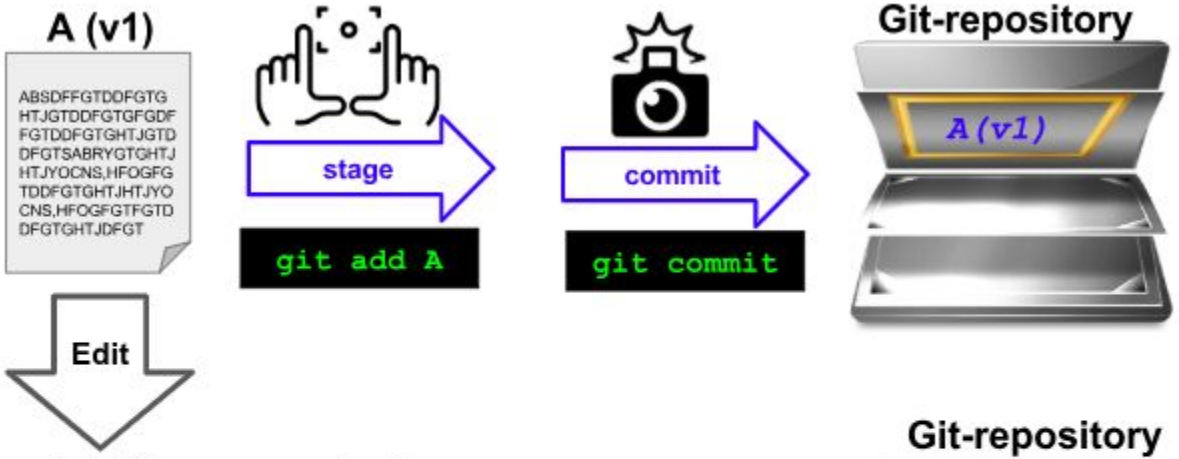


Pero también podemos usar un repositorio que ya existe en otro lado... Es decir queremos un **repositorio clon**

`git clone <URL_del_repositorio>` → Copia un repositorio existente desde una fuente remota.

Git

Mi archivo hoy



Guarda datos en el repo acerca de los cambios que hubo

Mi archivo mañana

Guarda datos en el repo acerca de los cambios que hubo

GitHub

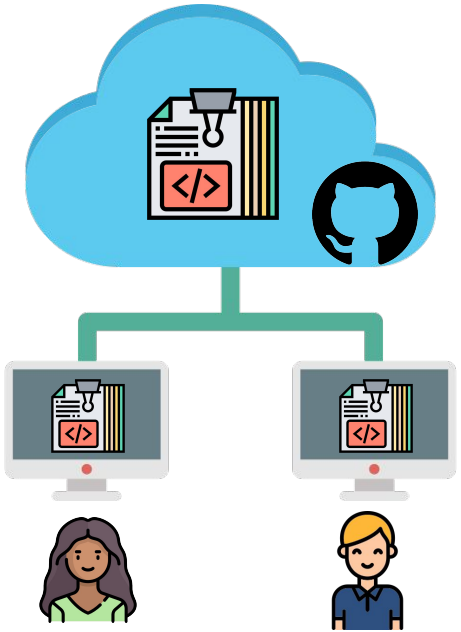
- GitHub es una plataforma en la nube para el **control de versiones** y **colaboración**, basada en Git
- Permite a los desarrolladores almacenar, administrar y rastrear cambios en su código
- GitHub también ofrece funciones para el seguimiento de problemas, revisión de código, administración de proyectos y automatización.

Git v.s. GitHub y sus primos

Control de versiones

Control de versiones basado en Git + Colaborar + Administrar + Desplegar

Colaboración: GitHub y sus primos



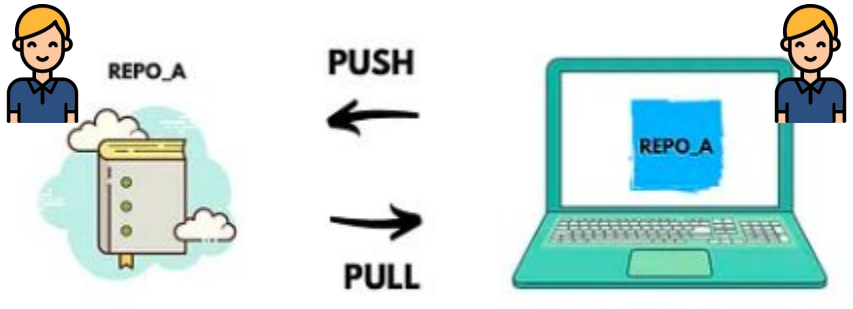
La colaboración tiene muchas ventajas
... pero también trae algunos inconvenientes

Básicamente el inconveniente mayor es ponernos de acuerdo en la manera cómo colaboramos

GitHub sirve para mantener ordenada la colaboración

Con el mega plus de que el código no está guardado en un solo lugar

GitHub



Conecta el repositorio local a un servidor remoto

`git remote add origin <URL_del_repositorio>`

Envía TU nueva versión con cambios al repositorio remoto

`git push origin <nombre rama>`

por ahora `git push origin main`

Descarga y fusiona los cambios del repositorio remoto

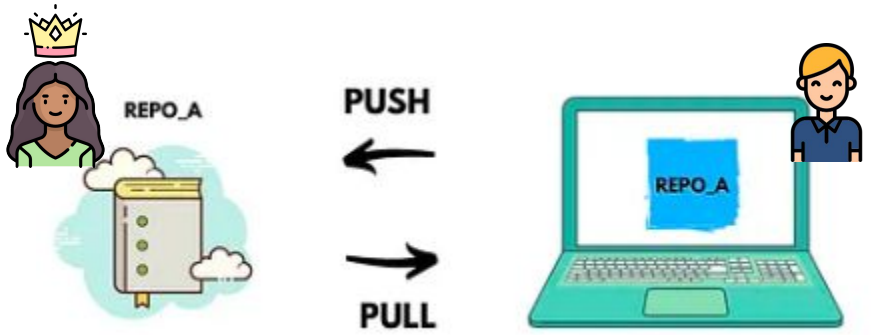
`git pull origin <nombre rama>`

por ahora `git pull origin main`

`git fetch` → **Descarga actualizaciones del repositorio remoto sin fusionarlas.**

GitHub

El repositorio remoto es un ente vivo, que está recibiendo modificaciones todo el tiempo



Que yo “empuje” mis cambios a un repo remoto NO significa que la dueña va a aceptar mis cambios

Necesitamos la versión del código más reciente

Descarga y fusiona los cambios del repositorio remoto

`git pull origin <nombre_rama>`
por ahora `git pull origin main`

Envía TU nueva versión con cambios al repositorio remoto

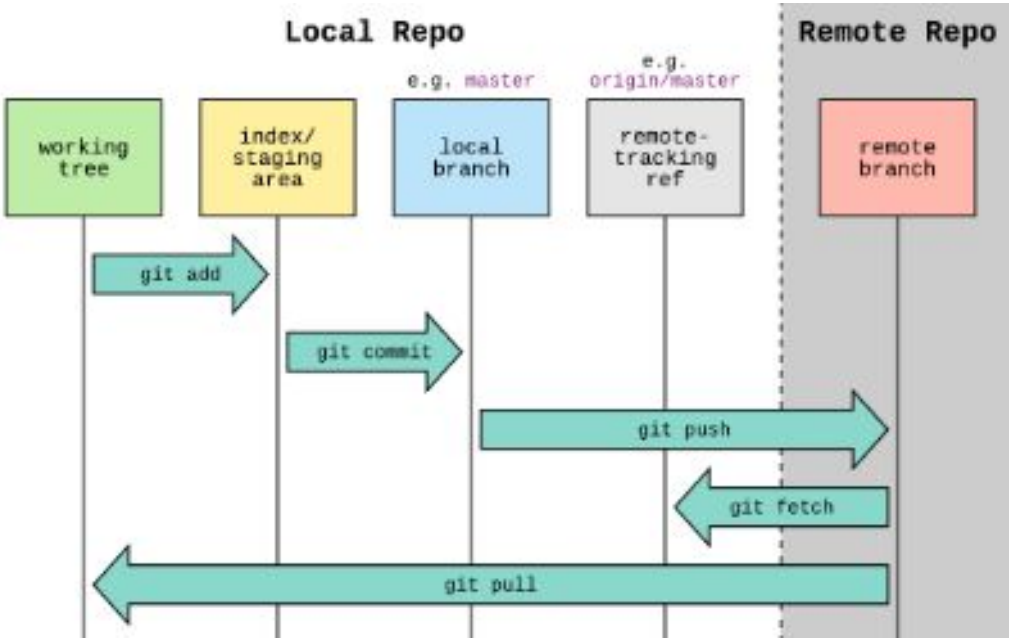
`git push origin <nombre rama>`
por ahora `git push origin main`

Crear un Pull Request (PR) en GitHub:

1. Ir al repositorio en GitHub.
2. Hacer clic en "Pull Requests" → "New Pull Request"
3. Seleccionar las ramas a comparar y revisar los cambios.
4. Hacer clic en "Create Pull Request" y agregar una descripción.

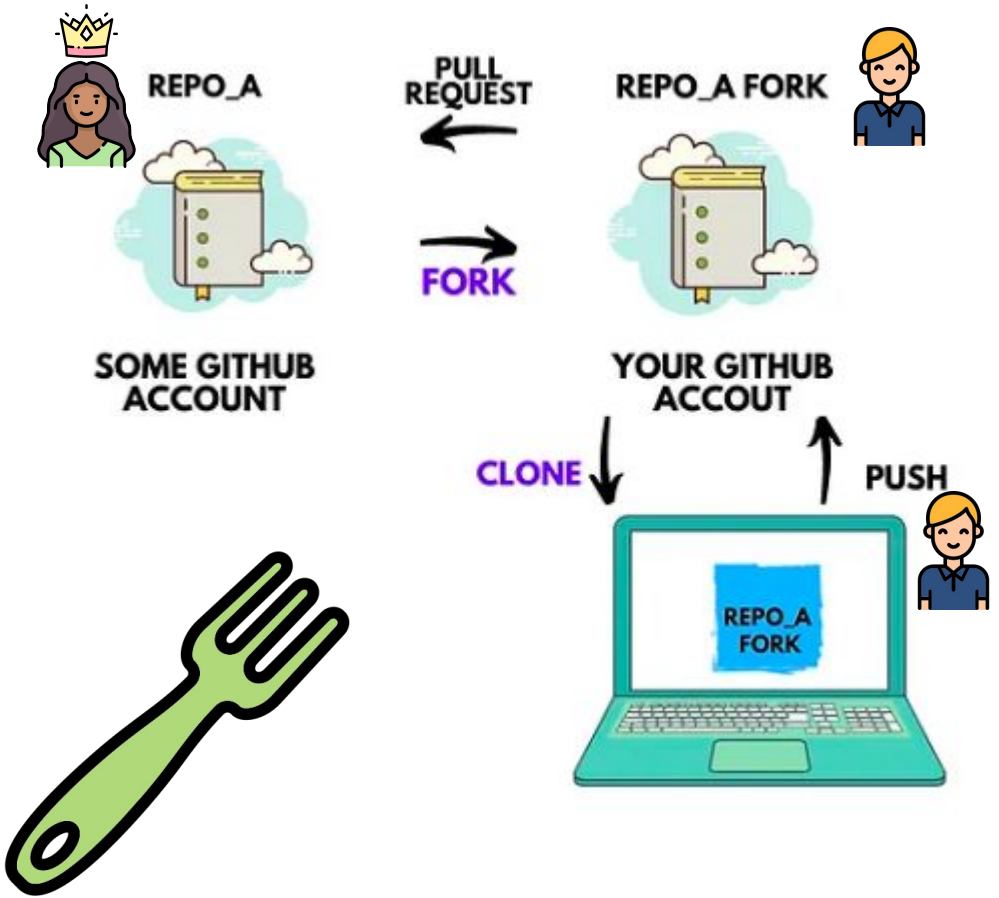
La persona “dueña” del repositorio remoto acepta o rechaza tu Pull Request

GitHub


Local Repo
Remote Repo
working tree
index/ staging area
local branch
e.g. master
remote-tracking ref
e.g. origin/master
remote branch
git add
git commit
git push
git fetch
git pull

GitHub

Cuando se hace un **fork** a un repositorio, se crea una copia del repositorio remoto original, pero el repositorio se copia en tu cuenta de GitHub, no en tu máquina local.



Creación de un proyecto con Git en RStudio

Caso 1: Inicializar un repositorio Git local nuevo

1. Ve a File → New Project
2. Selecciona New Directory → New Project
3. Marca la opción Create a Git repository
4. Haz clic en Create Project

Después de esto, RStudio creará un nuevo proyecto de R con Git habilitado. Deberías ver una pestaña de Git en el panel superior derecho.

Creación de un proyecto con Git en RStudio

Caso 2: Si deseas trabajar con un repositorio de GitHub ya existente, i.e. clonar un repositorio existente desde GitHub

Copia la URL del repositorio desde GitHub (ejemplo: https://github.com/user/repo.git).

En RStudio:

1. Ve a File → New Project → Version Control → Git
2. Pega la URL del repositorio
3. Selecciona una carpeta local donde guardar el proyecto
4. Haz clic en Create Project

Después de esto, RStudio creará un nuevo proyecto de R con Git habilitado. Deberías ver una pestaña de Git en el panel superior derecho.

Flujo de trabajo básico de Git en RStudio

Usa la pestaña de Git (en el panel superior derecho).

Paso A. Registrar y confirmar cambios (Commit)

1. Modifica o crea scripts de R (ejemplo: mi_cuaderno.Rmd)
2. Haz clic en la pestaña Git en RStudio
3. Selecciona los archivos a registrar (marca las casillas)
4. Haz clic en Commit
5. Escribe un mensaje de commit (ejemplo: "Agregado nuevo análisis")
6. Haz clic en Commit

Paso B. Subir cambios a GitHub
Haz clic en el botón Push (↑) para enviar los cambios a GitHub.

Paso C. Descargar cambios desde GitHub
Haz clic en el botón Pull (↓) para actualizar tu proyecto con los últimos cambios de GitHub.

R + Python

- Se puede ejecutar código de Python en R utilizando la librería de R **{reticulate}**
- Permite ejecutar scripts de Python, llamar funciones de Python y trabajar con objetos de Python directamente en R.

```
install.packages("reticulate")
library(reticulate)
```

- Las funciones `import_main()` e `import_builtins()` le dan acceso a la sesión de R al main de Python y a las funciones de funciones integradas de Python.

```
main <- import_main()
builtins <- import_builtins()
builtins$print('Hola Mundo!')
```

R + Python

- Se puede ejecutar código de Python directamente en un script de R con la función `py_run_string()`:

```
py_run_string("print('¡Hola desde Python!')")
```

- Para ejecutar un archivo de Python (`mi_script.py`):

```
py_run_file("script.py")
```

- Se puede definir funciones en Python y llamarlas desde R:

```
py_run_string("
def sumar(a, b):
    return a + b
")

# Se llama a la función de Python en R
py$sumar(5, 10)
```

R + Python

- Se puede ejecutar código de Python directamente en un script de R con la función `py_run_string()` y acceder a éste:

```
py_run_string("x = 10")
# acceder al main de python via el objeto 'py'
py$x
```

- Se puede importar y usar bibliotecas de Python directamente en R:

```
np <- import("numpy")  # se importa NumPy
np$array(c(1, 2, 3, 4, 5))  # se crea un array de NumPy
```

- Se puede especificar qué versión o entorno de Python usar:

```
use_python("/usr/bin/python3")  # Especificar una versión de Python
use_virtualenv("mi_entorno")  # Usar un entorno virtual
use_condaenv("mi_entorno_conda")  # Usar un entorno de Conda
```

R + Python

- De forma predeterminada, cuando los objetos Python se devuelven a R, se convierten a sus tipos de R equivalentes.

- Si prefiere hacer explícita la conversión de Python a R y tratar con objetos nativos de Python de forma predeterminada, se puede indicar `convert = FALSE` a la función de importación.

```
# importar numpy
np <- import("numpy", convert = FALSE)

# hacer una suma acumulada con NumPy
a <- np$array(c(1:4))
sumilla <- a$cumsum()

# convertir a R explícitamente
py_to_r(sumilla)
```

R + Python

- Se puede llamar a Python en scripts de R
- En este ejemplo, convertimos el dataframe de Palmer Penguins de R, en un dataframe de Pandas (de Python).
- Ya con este objeto python, se usará la función `pandas.crosstab`.

```
datos <- palmerpenguins::penguins

datos_en_pandas <- r_to_py(datos)

pd <- import("pandas", as = "pd", convert = FALSE)

tabla_frecuencias <- pd$crosstab(index = datos_en_pandas$species,
columns = datos_en_pandas$island)

print(tabla_frecuencias)
```

Algunos conceptos más profundos de Git

- Git es una base de datos distribuida en el corazón de su sistema de ingeniería.
    - Los datos se conservan en el disco.
    - Las consultas permiten a los usuarios solicitar información basada en esos datos.
    - El almacenamiento de datos está optimizado para estas consultas.
    - Los algoritmos de consulta están optimizados para aprovechar estas estructuras.
    - Los nodos distribuidos necesitan sincronizarse y "acordar" algún estado común.
- Si bien estos conceptos son comunes a todas las bases de datos, Git es particularmente especializado.
    - Git fue creado para almacenar archivos de código fuente de texto sin formato, donde la mayoría de los cambios son lo suficientemente pequeños como para leerlos de una sola vez, incluso si el código base contiene millones de líneas.
    - La gente usa Git para almacenar muchos otros tipos de datos, como documentación, páginas web o archivos de configuración.

Algunos conceptos más profundos de Git

- Git utiliza procesos de corta duración (a diferencia de otras bases de datos que utilizan procesos de larga duración con mucho almacenamiento en caché en memoria) y utiliza el sistema de archivos para persistir (conservar) los datos entre ejecuciones

- Los tipos de datos de Git son más restrictivos que los de una base de datos ”clásica”.

- Git no es una base de datos tradicional, pero tiene algunas características similares.

- Una base de datos “clásica” (i.e. como la que tenemos en mente) almacena datos estructurados en tablas (SQL) o documentos (NoSQL) para consultas y modificaciones eficientes.

Algunos conceptos más profundos de Git

- Git almacena archivos e historial de cambios en lugar de datos estructurados.
- Git rastrea cambios a lo largo del tiempo, en lugar de manejar consultas en tiempo real.
- Git usa un modelo distribuido, donde cada copia de un repositorio contiene todo el historial de versiones

¿En qué se parece Git a una base de datos “tradicional”?

- Almacenamiento basado en contenido: Git guarda datos como objetos identificados por un hash único (SHA-1), similar a cómo algunas bases de datos usan claves únicas.
- Control de versiones como almacenamiento de datos: Cada commit en Git puede verse como un "registro" en un historial.
- Recuperación eficiente de datos: Git puede recuperar versiones anteriores de archivos rápidamente, como una base de datos que consulta registros históricos.