Uso de git para trabajo colaborativo

Profesor: Joaquín Fontbona T.
Auxiliares: Pablo Zúñiga Rodríguez-Peña, Arie Wortsman Z., Camilo Carvajal Reyes

En este documento mostraremos las instrucciones para instalar y usar git, que es un sistema de control de versiones de código abierto ampliamente utilizado.

Índice de Contenidos

1. Instalación (Windows)

1.1. Instalación de *git* y *git bash*

Git viene integrado en las terminales de macOS y las distintas distribuciones de Linux. Para usar git en Windows usaremos una herramienta llamada *git bash*, que viene con la instalación de git que seguiremos a continuación.

1. Dirigirse al sitio de descargas de git para windows.
2. Seleccionar la opción *standalone installer* correspondiente a las características del sistema (64 o 32 bits). En la figura 8 podemos ver la página de instalación.

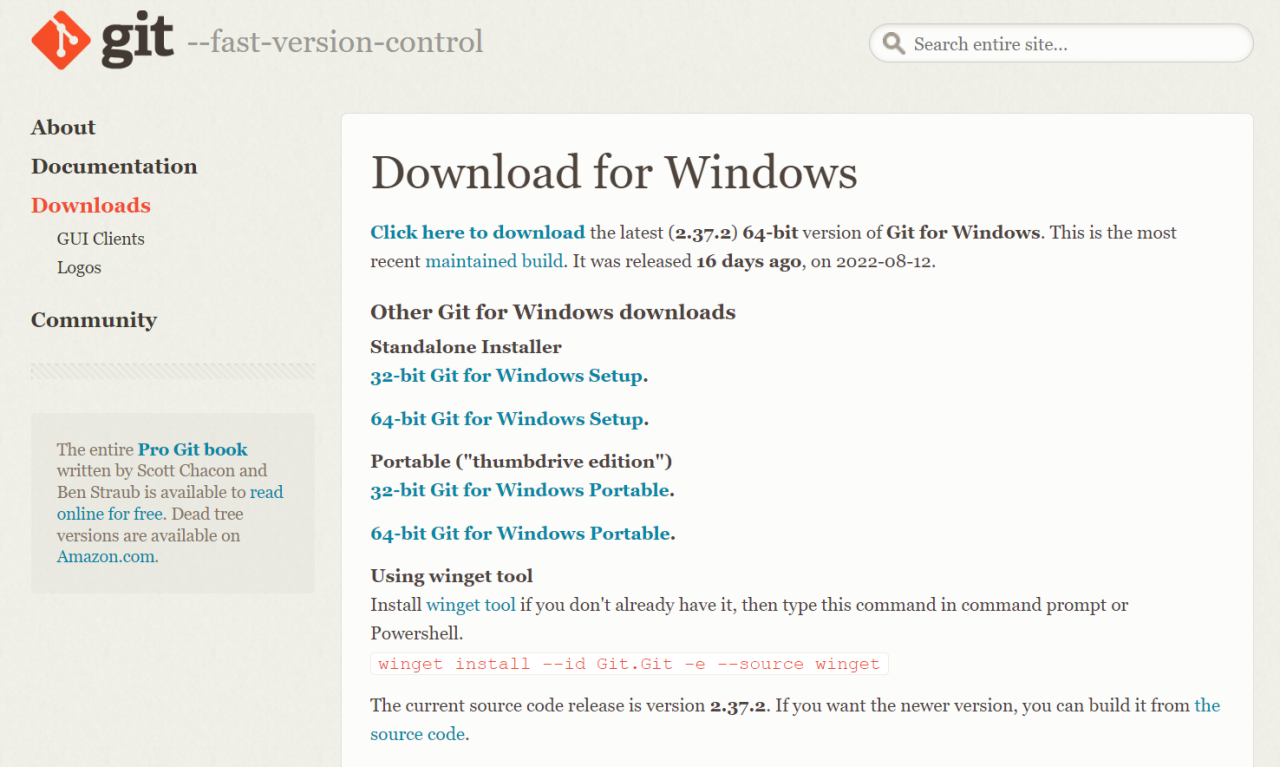


Figura 1: Descarga de instalación para Windows

.

3. Llevamos a cabo la instalación con los valores predeterminados. En particular debe estar seleccionada la casilla *git bash*. También es muy recomendable tener selecionada la instalación de *git lfs*, como es el caso de nuestra instalación de la figura 2

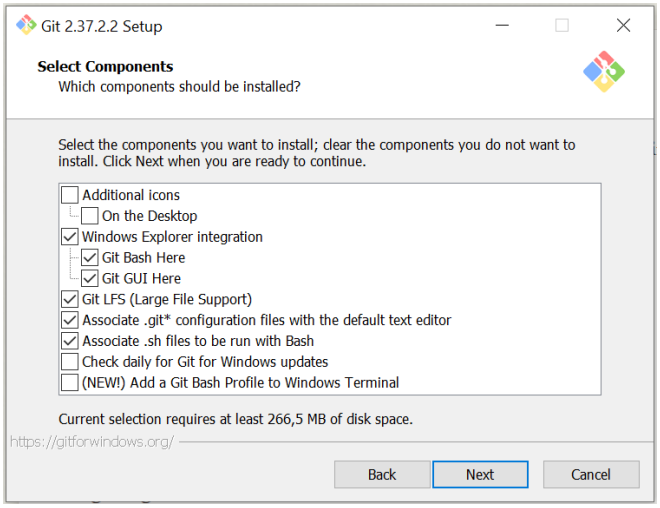


Figura 2: Selección de componentes en instalación para Windows

.

4. Una vez terminada la instalación, chequeamos que esta se haya realizado correctamente abriendo *git bash*. Para esto podemos escribir "bash" en el buscador de Windows, tal como en la figura 3.

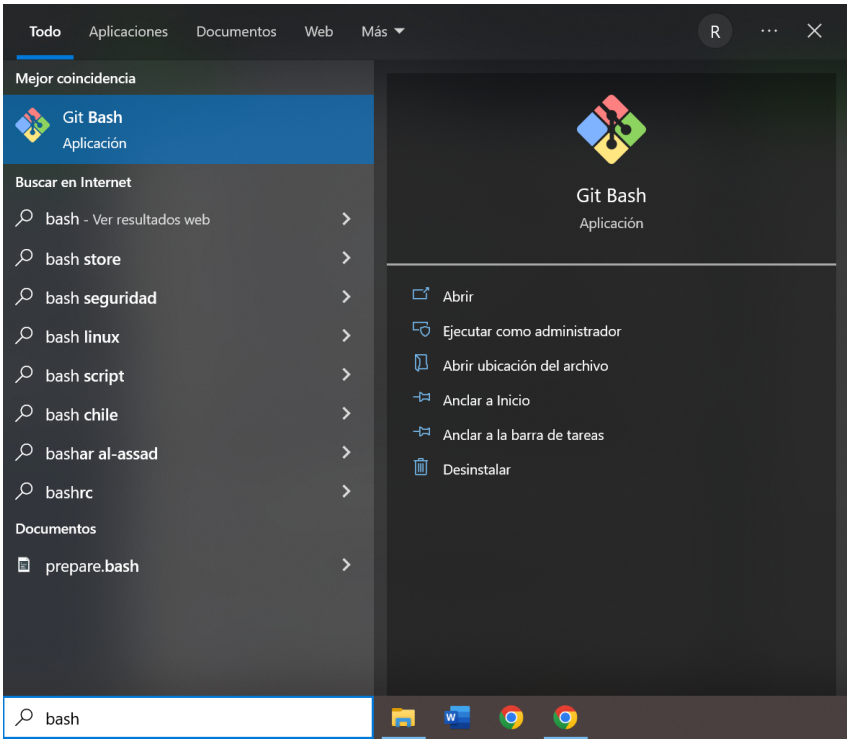


Figura 3: Abriendo *git bash*.

1.2. Configurar *git bash* para usar *conda*

OPCIONAL: Una vez que tengamos instalado *miniconda* y *git bash* podemos usar esta última para emular el uso de *Anaconda prompt* (y las terminales de otros sistemas operativos). De este modo podremos usar comandos de *conda* y gestión de archivos con *git* desde una misma línea de comando (fuente de esta sección).

1. Primero abrimos la carpeta de instalación de miniconda (o bien de anaconda) como muestra la figura 4.

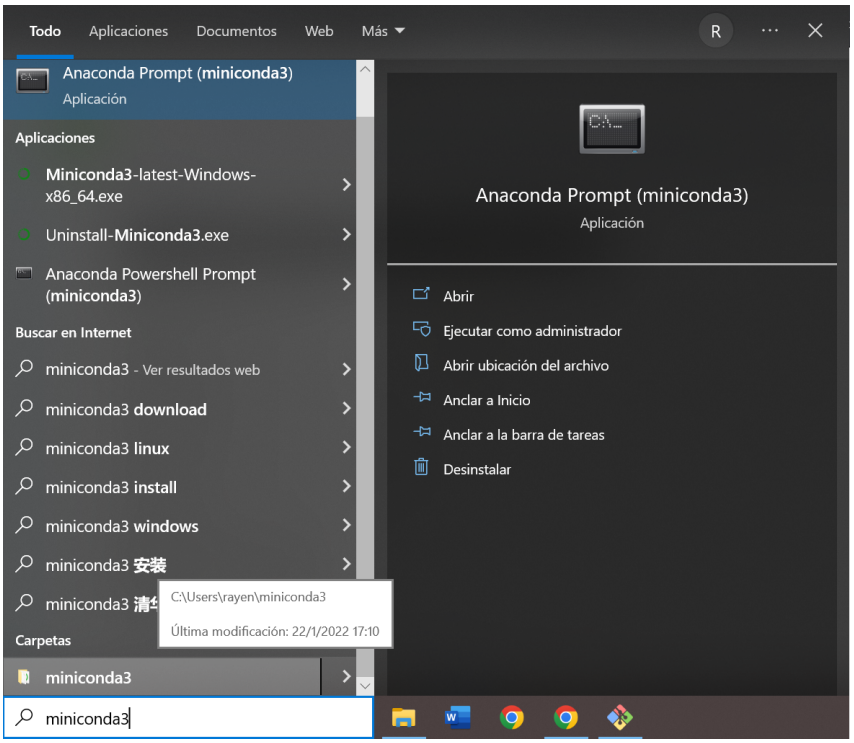


Figura 4: Abriendo el directorio de instalación de *miniconda*.

2. Cliqueamos en la carpeta *etc* y luego en la carpeta *profile.d*. En aquel momento deberíamos ver archivos como en la figura 5.

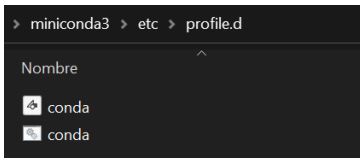


Figura 5: Abriendo el archivo de configuración de *miniconda*.

3. Hacemos click derecho en algun punto del explorador. Como bien es el caso de la figura 6, nos aparecerá la opción de abrir la terminal *git bash*. Hacemos click a esto.

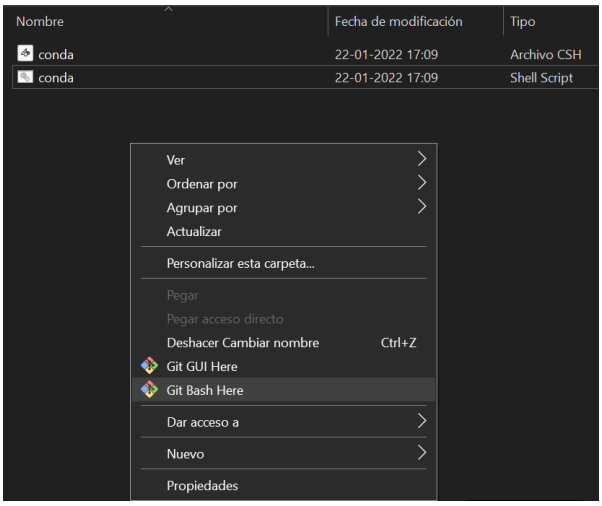


Figura 6: Abriendo *git bash* en el directorio de *miniconda*.

4. En la terminal que se abre desde aquel directorio ejectuamos el siguiente comando:

```
echo ". '${PWD}'/conda.sh" >> ~/.bashrc
```

5. Cerramos la terminal y volvemos a abrir *git bash*.

6. Intentamos activar algún ambiente conda que tengamos disponible con *conda activate env_name* (reemplazando env_name por el nombre del ambiente), o bien activar el ambiente de base sólo con *conda activate*. Esto debería mostrar el ambiente arriba del directorio actual de la terminal, como en la figura 7.

Figura 7: Activando *conda* en *git bash*.

2. Creación y gestión de un repositorio de manera local

En esta sección aprenderemos los comandos básicos de git para el uso en una linea de comandos. Esta puede ser los terminales de macOS o Linux, o bien *gitbash* en Windows. Partiremos creando un repositorio vacío y añadiremos archivos.

2.1. Inicializando un repositorio (init)

1. Desde la terminal (si no lo hemos hecho) debemos agregar nuestros datos ejectuando

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

donde reemplazamos “you@example.com” por el correo y “Your Name” por nuestro nombre.

2. Creamos un directorio para trabajar. Este puede ser creado de manera manual con un explorador de archivos, o bien con el comando *mkdir* desde una terminal.

3. Nos dirigimos a la carpeta desde la linea de comando usando

```
cd ruta_directorio
```

donde reemplazamos ruta_directorio por la ruta completa del directorio de trabajo.

4. Un repositorio consiste en un lugar (en este caso un directorio) donde se guardan y gestionan archivos. Un repositorio es manejado por git usando un archivo oculto *.git* que maneja las versiones de los archivos. Para inicializar un repositorio ejecutamos

```
git init
```

Esto debería verse como en la figura 8.

```
(base) camilo@camilo-G3-3590:~$ mkdir MA4402
(base) camilo@camilo-G3-3590:~$ cd MA4402
(base) camilo@camilo-G3-3590:~/MA4402$ git init
Initialized empty Git repository in /home/camilo/MA4402/.git/
(base) camilo@camilo-G3-3590:~/MA4402$
```

Figura 8: Inicializando un repositorio

.

2.2. Añadiendo cambios (add y commit)

1. Pogámonos en el caso de querer agregar un archivo a nuestro repositorio. Agregaremos un script de python llamado “test.py”, que simplemente imprime el string *hola mundo*. Haremos que git reconozca el archivo al ejecutar el comando

```
git add test.py
```

con lo cual el archivo se considerará montado (*staged* en inglés).

2. El ejecutar

```
git status
```

nos describirá el estado del directorio de trabajo. Veamos el resultado de ejecutarlo luego de haber montado el archivo test.py en la figura 9.

```
(base) camilo@camilo-G3-3590:~/MA4402$ git add test.py
(base) camilo@camilo-G3-3590:~/MA4402$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.py

(base) camilo@camilo-G3-3590:~/MA4402$
```

Figura 9: Git status al montar un archivo test.py

.

3. Ahora realicemos un commit. El commit es una operación que envía la última versión del código al repositorio. Este viene acompañado de un mensaje corto que describe los cambios ejecutados. Si los cambios no han sido montados estos no serán considerados por el commit. Realizamos el commit con el comando

```
git commit -m "mensaje del commit"
```

   donde lo que va después de -m es la descripción. En la figura 10 se ve el resultado de realizar un commit.

```
(base) camilo@camilo-G3-3590:~/MA4402$ git commit -m "add test.py"
[master (root-commit) 9380094] add test.py
 1 file changed, 3 insertions(+)
 create mode 100644 test.py
(base) camilo@camilo-G3-3590:~/MA4402$ git status
On branch master
nothing to commit, working tree clean
(base) camilo@camilo-G3-3590:~/MA4402$
```

Figura 10: Realizando un commit

4. Ahora veamos el caso en el que modificamos un archivo. En nuestro caso le agregamos el sampleo de una normal estándar a nuestro script. Para ver los cambios realizados a archivos que están siendo monitoreados por git ejecutamos lo siguiente:

```
git diff --name-only
```

   Si no agregamos "–name-only" entonces veremos el detalle de los cambios. Por otro lado, para ver los cambios que han sido montados con git add corremos el comando

```
git diff --name-only --staged
```

Para ver un ejemplo de esto agregamos un archivo README.md y lo montamos, además de la modificación que ya le hicimos a test.py, veremos lo que se muestra en figura 11.

```
(base) camilo@camilo-G3-3590:~/MA4402$ git add README.md
(base) camilo@camilo-G3-3590:~/MA4402$ git diff --name-only
test.py
(base) camilo@camilo-G3-3590:~/MA4402$ git diff --name-only --staged
README.md
(base) camilo@camilo-G3-3590:~/MA4402$ git diff
diff --git a/test.py b/test.py
index 1858da7..bf91da8 100644
--- a/test.py
+++ b/test.py
@@ -1,3 +1,6 @@
+import numpy as np
+
 if __name__=="__main__":
:...skipping...
diff --git a/test.py b/test.py
index 1858da7..bf91da8 100644
--- a/test.py
+++ b/test.py
@@ -1,3 +1,6 @@
+import numpy as np
+
 if __name__=="__main__":
-        print("Hola mundo")
+        print(np.random.normal())
+        print("Sampleo de una Gaussiana")
:...skipping...
diff --git a/test.py b/test.py
index 1858da7..bf91da8 100644
--- a/test.py
+++ b/test.py
@@ -1,3 +1,6 @@
+import numpy as np
+
 if __name__=="__main__":
-        print("Hola mundo")
+        print(np.random.normal())
+        print("Sampleo de una Gaussiana")

~
~
~
(base) camilo@camilo-G3-3590:~/MA4402$
```

Figura 11: Distintas formas de ejecución de git diff

2.3. Deshaciendo cambios (reset y restore)

A continuación tenemos otros comandos que nos permiten deshacer cambios en git:

```
# Para desmontar un archivo pero no deshacer los cambios
git reset -- README.md
```

```
# para deshacer cambios
git restore test.py
```

Visualizamos estos comando en figura 12.

```
(base) camilo@camilo-G3-3590:~/MA4402$ git reset -- README.md
Unstaged changes after reset:
M       test.py
(base) camilo@camilo-G3-3590:~/MA4402$ git restore test.py
(base) camilo@camilo-G3-3590:~/MA4402$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

Figura 12: Desmontando archivos y deshaciendo cambios

Si por otro lado queremos deshacer el último commit podemos hacerlo con

```
git reset HEAD^
```

HEAD^ denota el commit anterior a HEAD, que es el último commit. Por ende estaríamos volviendo al estado del primer commit, como se ve en la figura 13.

```
(base) camilo@camilo-G3-3590:~/MA4402$ git add README.md
(base) camilo@camilo-G3-3590:~/MA4402$ git commit -m "add README.md"
[master ca79fdb] add README.md
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
(base) camilo@camilo-G3-3590:~/MA4402$ git reset HEAD^
(base) camilo@camilo-G3-3590:~/MA4402$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present (use "git add" to track)
(base) camilo@camilo-G3-3590:~/MA4402$
```

Figura 13: Haciendo y deshaciendo un commit

3. Gestionando un repositorio remoto

3.1. Creando una versión remota de un repositorio local

Imaginemos que tenemos un directorio local con código que queremos compartir para trabajar con otras personas. En este caso podemos ir al directorio en cuestión, ejecutar

```
git init
```

```
git add --all
git commit -m "add all files"
```

y tendremos un repositorio git gestionable. Podemos subir con este repositorio a la web con la ayuda de github. Esto nos permite trabajar de manera colaborativa y compartir nuestros proyectos con la comunidad, además de contar con un respaldo para nuestro trabajo. A continuación usaremos github para tener una versión "remota" de nuestro repositorio. También existen alternativas como gitlab o Bitbucket, cuyo funcionamiento es análogo.
ADVERTENCIA: acá usamos "git add -all" a modo de ejemplo. Muchas veces podremos estar agregando archivos confidenciales o archivos pesados. No queremos que estos archivos estén en nuestro repositorio remoto. Veremos más adelante como arreglar este problema con un archivo .gitignore.

1. Creamos una cuenta en github.
2. Creamos un repositorio vacío en github/new, como en la figura 14.

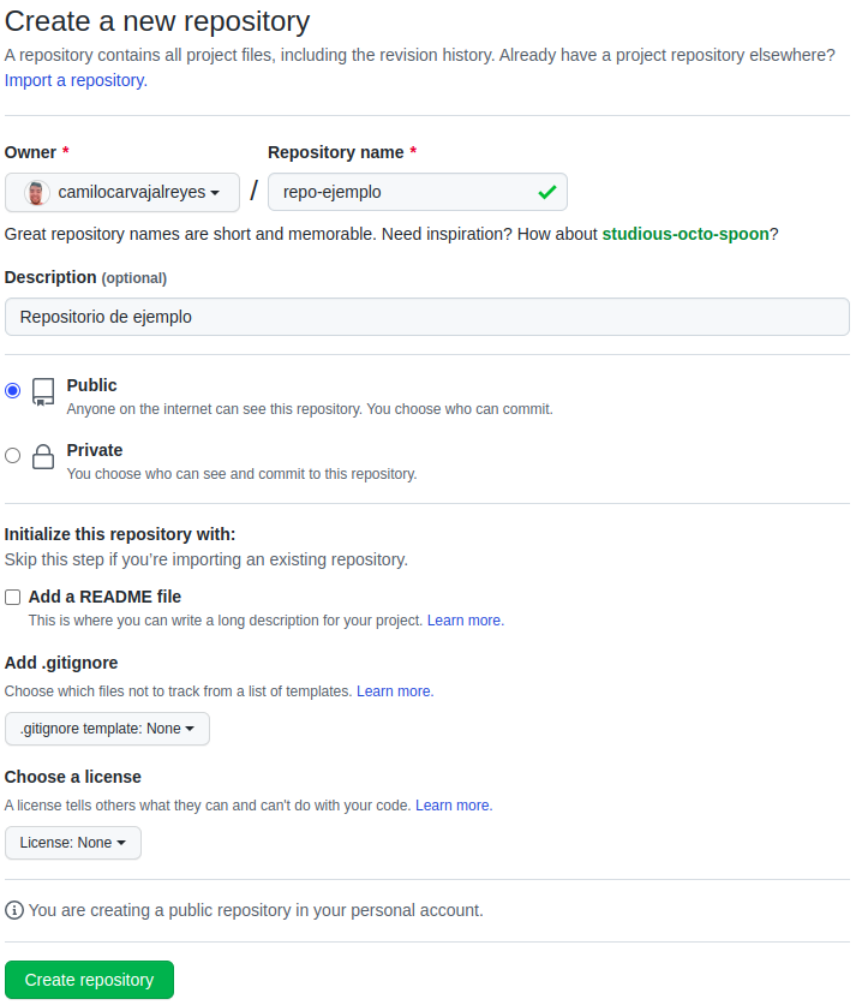


Figura 14: Creando un repositorio vacío en Github.

Acá aceptamos las opciones por defecto puesto que empujaremos un repositorio local. Notar que nuestro repositorio puede ser privado. En nuestro caso preferimos que esté público. Luego de apretar *Create repository* veremos una ventana como en la figura 15.

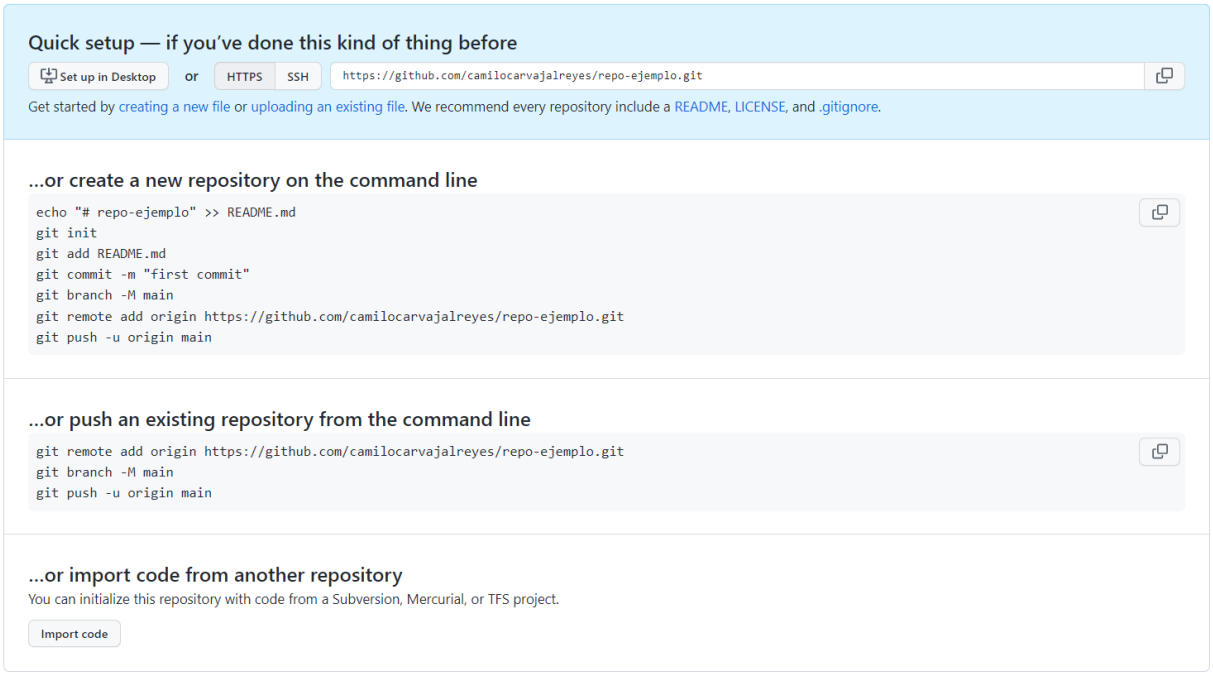


Figura 15: Creando un repositorio vacío en Github, parte dos.

3. En este caso seguiremos la segunda opción: empujando un repositorio desde la línea de comando. Para aquello hay que ejecutar los comandos siguientes, desde una terminal corriendo desde el directorio de nuestro proyecto (el mismo donde está la carpeta .git):

```
git remote add origin https://github.com/user/nombre-repo.git
git branch -M main
git push -u origin main
```

Debemos reemplazar *user* por nuestro usuario de github y *nombre-repo* por el nombre de nuestro repositorio. Luego de ejectuar el último comando tenemos que conectar nuestra cuenta github con nuestra instalación de git. En algunos casos se pedirán las credenciales directamente en la línea de comando. Para *git bash* (Windows), la terminal se mostrará como en la figura 16 al mismo tiempo que se abrirá la ventana de la figura 17.

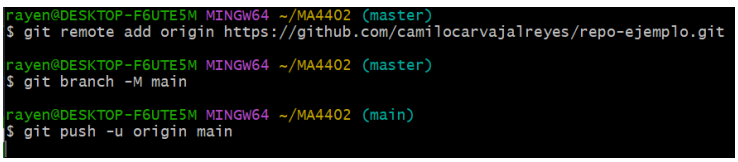


Figura 16: Agregando una dirección remota al repositorio.

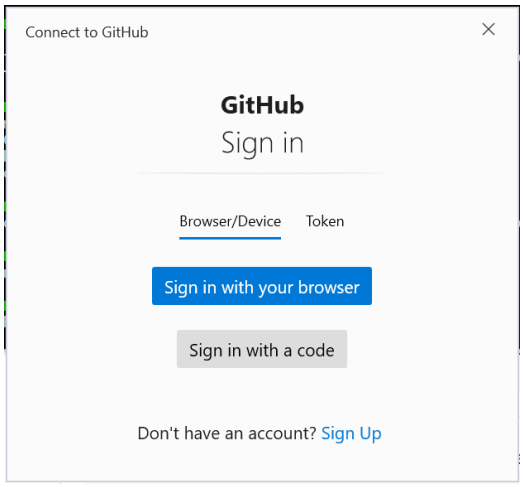


Figura 17: Conección a github desde la terminal

Al presionar *Sign in with your browser* se abrirá la ventana de la figura 18 en el navegador. Una vez que tengamos nuestra cuenta *github* abierta en ella debemos autorizar el uso de *GitCredentialManager* presionando el botón verde.

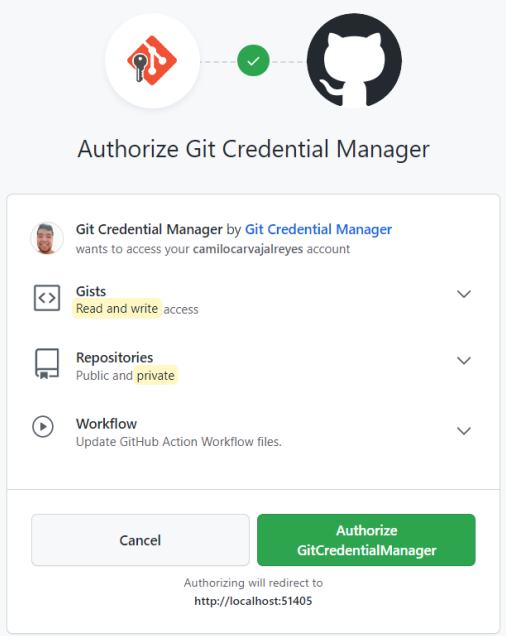


Figura 18: Autorizar el uso de github.

Luego de aquello, el comando de la terminal se terminará de ejectuar, como en la figura 19, llevando acabo el "empuje" de nuestro repositorio local a la versión remota.

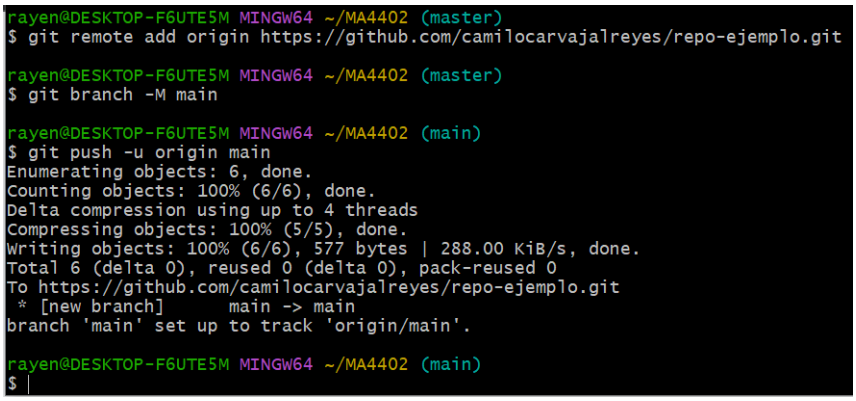


Figura 19: Agregando una dirección remota al repositorio luego de configurar github.

Veamos como se ve esto en el dominio *github*. En general la estructura de la url es "https://github.com/user/nombre-repo". Para este ejemplo el repositorio está disponible en https://github.com/camilocarvajalreyes/repo-ejemplo. Luego de nuestro primer push debería verse como en la figura 20.

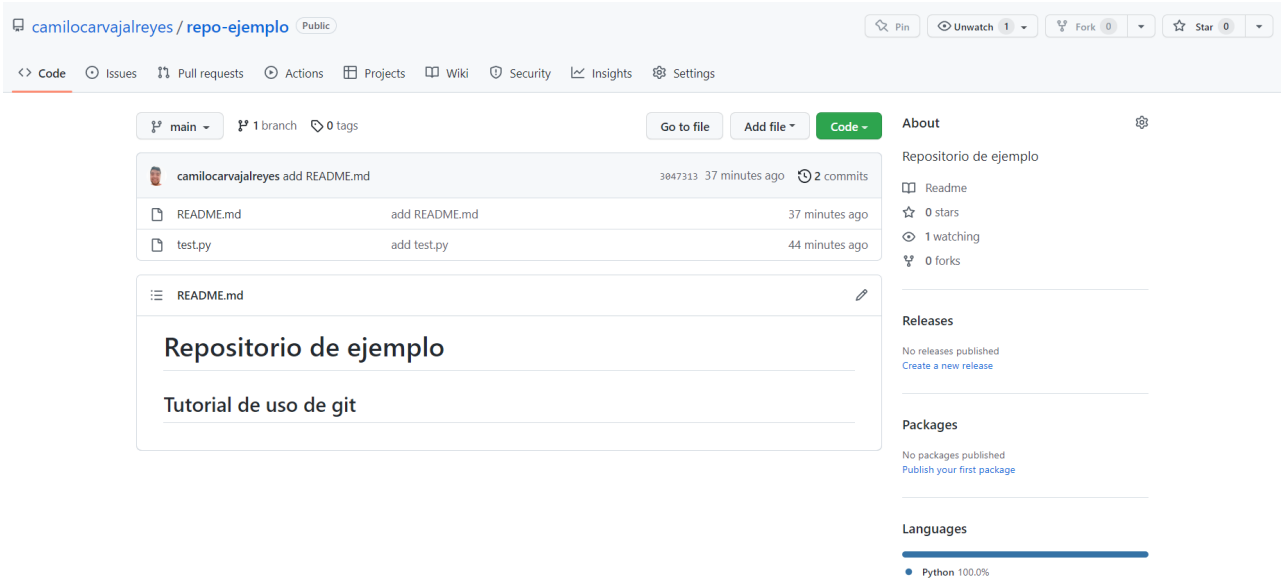


Figura 20: Visualización del repositorio de ejemplo en github.

3.2. Clonando un repositorio remoto (clone)

Cuando tenemos un repositorio remoto que queremos importar en nuestro computador lo que haremos será clonarlo. Encontraremos esto en las siguientes situaciones:

- Queremos trabajar con colegas pero desde los computadores personales, teniendo un sólo repositorio centralizado.
- Queremos ejectutar el proyecto de otra persona y eventualmente realizar nuestras propias modificaciones

- Queremos trabajar de manera personal pero usando distintos computadores (locales y/o virtuales)

Clonar un repositorio consiste en importar los archivos del proyecto remoto en cuestión, pero además crea de manera automática un puntero hacia el remoto. Esto nos permite importar actualizaciones del remoto a medida que estas se realicen (ver sección 3.4).

Para clonar un repositorio ejecturaremos

```
git pull direccion_repositorio
```

donde direccion_repositorio empieza con https y para *github* tiene la estructura "https://github.com/user/nombre-repo". A este método lo llamamos HTTPS, y mostramos un ejemplo de su uso en la figura 21. Existen también otros modos de clonar un repositorio que no cubriremos en este tutorial. Por ejemplo, el método SSH implica realizar ciertos pasos previos que se explican en este tutorial, y es una buena idea sobretodo si se quiere trabajar con repositorios privados.

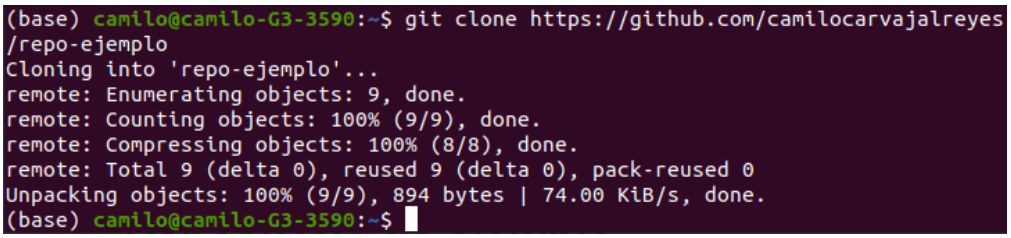


Figura 21: Clonando un repositorio.

3.3. Empujando un cambio (push)

Ahora haremos otro cambio a nuestro repositorio (ya sea clonado o creado localmente y habiéndole asignado una dirección remota). A modo de ejemplo, reemplazamos (en algún editor) el contenido de test.py por el código siguiente:

```
import numpy as np


if __name__=="__main__":
    print(np.random.normal())
```

Correr este código imprime la realización de una variable Gaussiana estándar unidimensional. Luego de montar los cambios (ejecutando git add test.py) haremos un commit (con git commit -m "mensaje"). Como ahora nos interesa que nuestra versión remota refleje los cambios locales le haremos "push" a este commit.

```
git push
```

En la figura 22 se ve la terminal luego de ejecutar el push. Podemos chequear que los cambios fueron efectivamente empujados dirigiendonos a la sección commits del repositorio. Podemos acceder a esta presionando el botón marcado en la figura 23 o bien agregando "/commits/main" a la URL del dominio github de nuestro proyecto Luego de acceder a esto veremos algo como en la figura 24. Podemos acceder a cada commit y ver el detalle de este.

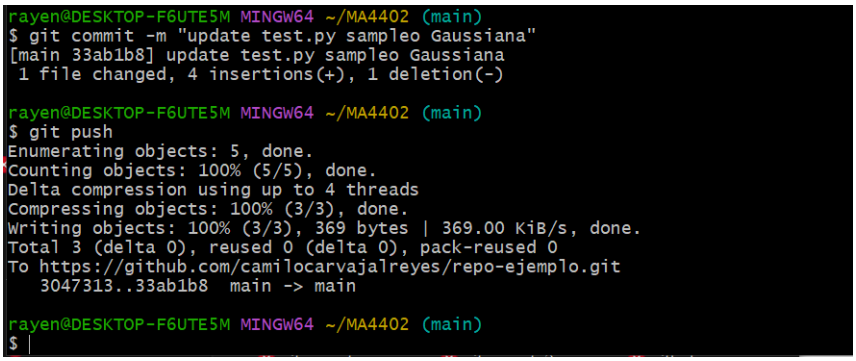


Figura 22: Realizando un commit y un push.

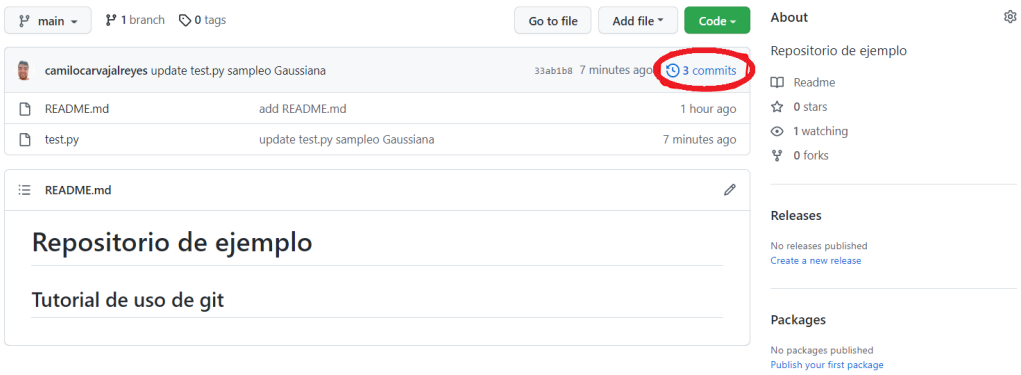


Figura 23: Dirigiendose a la sección commits de github.

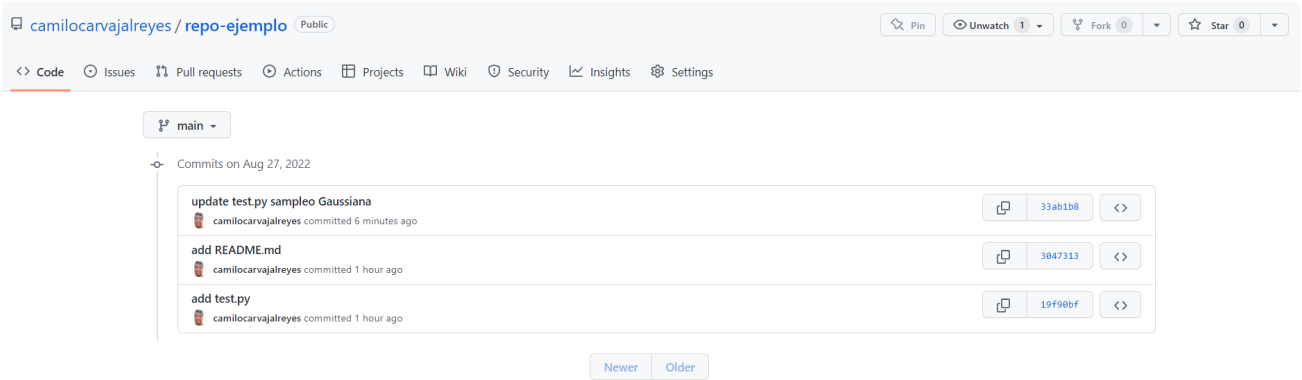


Figura 24: Vista de los últimos commits.

Muchas veces será útil visualizar los cambios hechos en un commit. Github nos muestra esto de manera amigable visualmente, como en el ejemplo de la figura 25.

Figura 25: Detalle de un commit.

IMPORTANTE: para poder efectivamente empujar los cambios a un repositorio remoto debemos tener los permisos para esto. El haber creado el remoto como lo hicimos en la sección 3.1 nos da automáticamente ese derecho. También es el caso de haber clonado un repositorio de nuestra propia cuenta y teniendo nuestras credenciales debidamente autorizadas (notar que podemos hacer nuestra propia copia remota de otro repositorio remoto usando *fork*). Por último, si queremos trabajar de modo colaborativo lo mejor es agregar distintas cuentas como *collaborators* (para github esto se hace donde se muestra en la figura 26). Esto compartirá los derechos de edición del remoto.

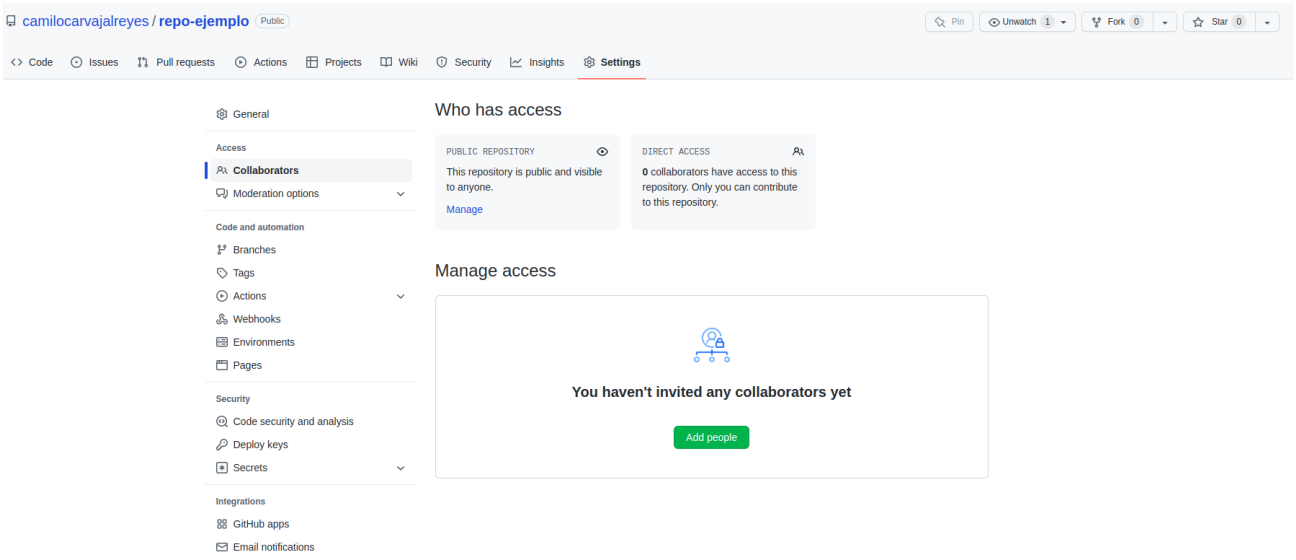


Figura 26: Agregar *collaborators* en github.

3.4. Importando cambios (pull)

Imaginemos que queremos importar cambios hechos por otra persona o en otro computador. Primero necesitamos que los cambios estén empujados (haciendo *push*), de modo que los commits estén disponibles en el repositorio remoto. Luego de aquello, otras versiones locales pueden descargar esos cambios usando:

```
git pull
```

A modo de ejemplo editaremos el README directamente desde github, lo cual hará que el commit esté en el remoto automáticamente (figura 27). Luego haremos git pull para que ese cambio esté de forma local (figura 28).

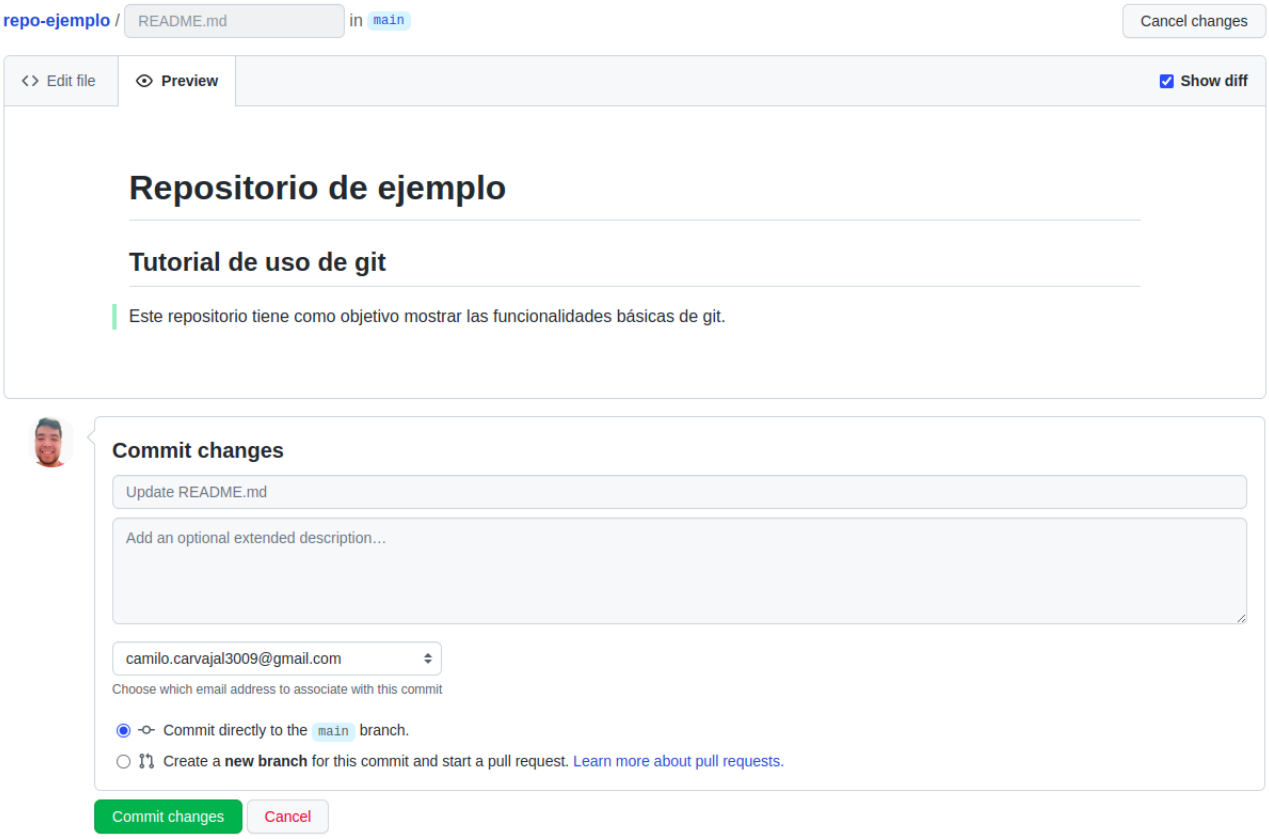


Figura 27: Actualizando el README directamente desde github.

```
(base) camilo@camilo-G3-3590:~/repo-ejemplo$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 763 bytes | 763.00 KiB/s, done.
From https://github.com/camilocarvajalreyes/repo-ejemplo
   e2844b2..15f9dc1  main       -> origin/main
Updating e2844b2..15f9dc1
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
(base) camilo@camilo-G3-3590:~/repo-ejemplo$
```

Figura 28: Realizando un pull para importar cambios del remoto.

Cabe señalar que el comando *git pull* es una simplificación de dos comandos: *git fetch* y *git merge*. Para el segundo necesitamos que no exista lo que se llama un "merge conflict". Esto sucede, por ejemplo, cuando algún commit en el remote edita un archivo que también ha sido editado de forma local. Se pueden resolver este tipo de choque de versiones con herramientas llamadas "merge conflict tools", de las cuales veremos un ejemplo en la sección 4.

3.5. Ignorando archivos (.gitignore)

Muchas veces tendremos archivos que no nos gustaría que aparezcan en una versión web pública. Lo recomendable en este caso es que tengamos un archivo ".gitignore". Podemos crear este archivo con cualquier editor (como el block de notas, VS Code, entre otros) y sigue ciertas reglas básicas de sintaxis. Mostramos un ejemplo con algunas reglas en figura 29.

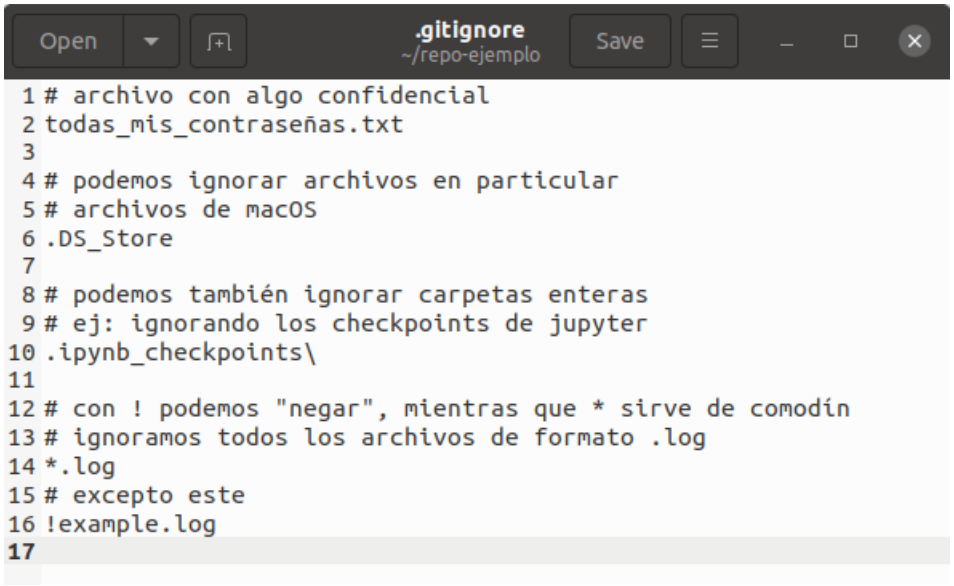


Figura 29: Creando un archivo .gitignore para ignorar archivos.

Notaremos que este archivo será oculto por los exploradores pues comienza por un punto. Para editarlo manualmente necesitamos entonces activar la opción que muestra archivos ocultos.

Los gestores de repositorios remotos como github y gitlab suelen tener limitaciones en cuanto al tamaño de los archivos a guardar (usualmente 100 MB). En este caso tenemos dos opciones, o bien los agregamos al .gitignore o bien usamos git lfs, que es una extensión que permite gestar archivos de gran tamaño.

4. Usando git con Visual Studio Code

Visual Studio Code es uno de los editores de código más usados. Posee soporte para numerosos lenguajes de programación, diversas funcionalidades añadidas (a través de extenciones) y una cómoda integración con git. Exploraremos las funcionalidades de esta última.

4.1. Acciones equivalentes a comandos básicos

Primero repasemos como hacer las operaciones básicas anteriores con la interfaz de VS Code. Para esto realizaremos una modificación a nuestro código usando el editor:

1. Abrir la carpeta correspondiente a un repositorio (i.e., el directorio con la carpeta oculta .git). Esto puede hacerse desde la barra superior o bien presionando sucesivamente los comandos ctrl + K y ctrl + O, para luego seleccionar la ruta correspondiente.

2. Desde el explorador de archivos seleccionamos un archivo a modificar. En nuestro caso le haremos un cambio a test.py, como bien se muestra en la figuras 30 y 30 (texto antes y después de editar respectivamente).

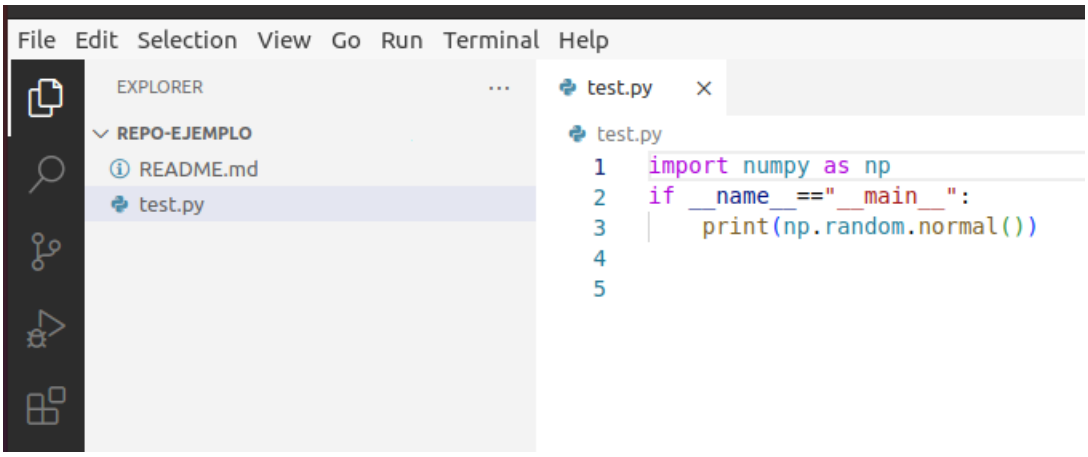


Figura 30: Editando un script con VS Code.

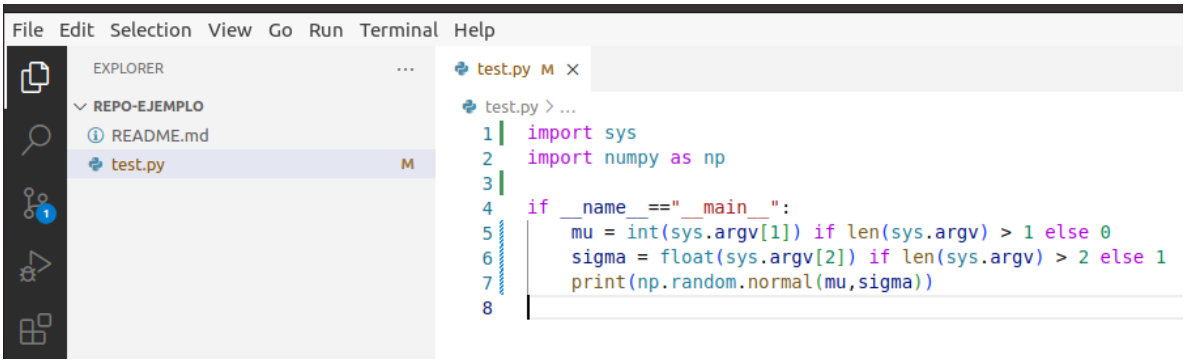


Figura 31: Editando un script con VS Code parte dos.

3. De ser necesario debemos instalar extensiones de VS Code que nos permitan la edición inteligente de código python (lo cual debería ser sugerido automáticamente con VS code).

Notemos que antes y después de la edición se notan ciertos cambios:

- En la barra izquierda tenemos un círculo azul con un número uno, sobre el ícono de Source Control.
- En el explorador de archivos tenemos una letra M marcando nuestro archivo editado.
- Las lineas de nuestro código están marcadas con colores. Verde corresponde a las lineas añadidas mientras que el celeste rallado corresponde a bloques de código editados.

Presionaremos el ícono Source Control (con el logo de git) y enseguida cliqueamos en el nombre del archivo editado. Se abrirá una ventana con las diferencias del archivo editado con el estado del último commit, como se visualiza en la figura 32.

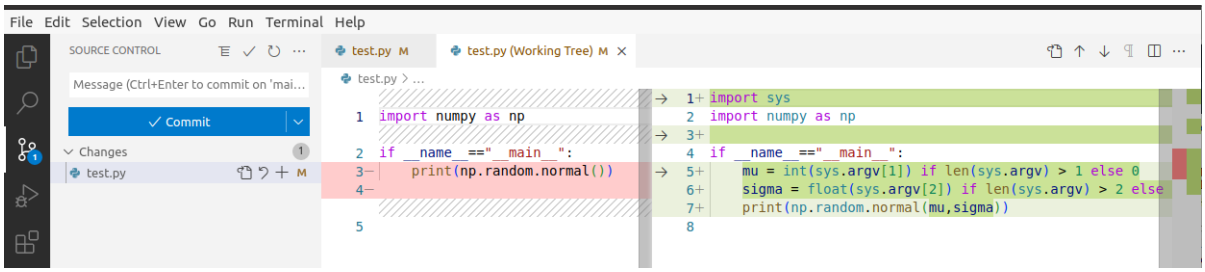
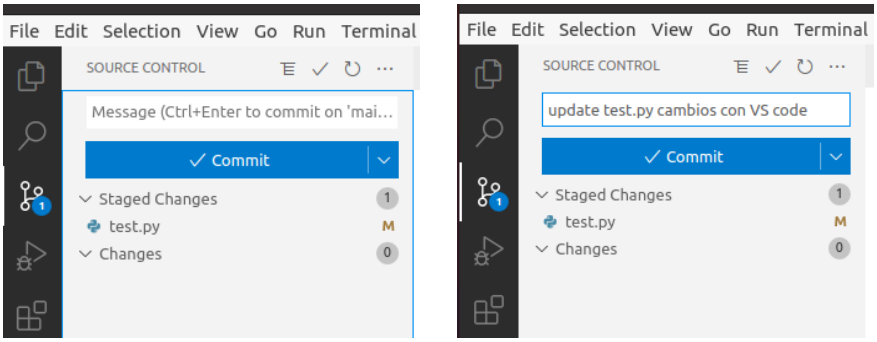


Figura 32: Viendo cambios en un script con VS Code source control.

Esto es parecido a lo que se muestra al ejecutar *git diff* en una terminal, pero de manera visualmente amigable. Ahora podemos montar y hacerle commit a los cambios desde Source Control, lo cual equivale a ejecutar *git add* y *git commit -m "mensaje"* desde una terminal:

1. Primero debemos pasar el puntero encima de los archivos a montar en sección Changes.
2. Veremos un signo + que debemos presionar. Ahora los archivos en cuestión aparecerán en la versión Staged Changes.
3. Agregamos un mensaje para el commit en la barra de escritura.
4. Presionamos Commit.

Podemos ver parte de estos pasos en la figura 33. Finalmente podemos empujar los commits al remoto presionando Push, que aparece en la barra de selección al presionar los tres puntos al costado de Source Control, como se ve en la figura 34. De modo similar podemos hacer un pull para importar cambios.



(a) Montando cambios (b) Mensaje para el commit

Figura 33: Añadiendo cambios con VS Code

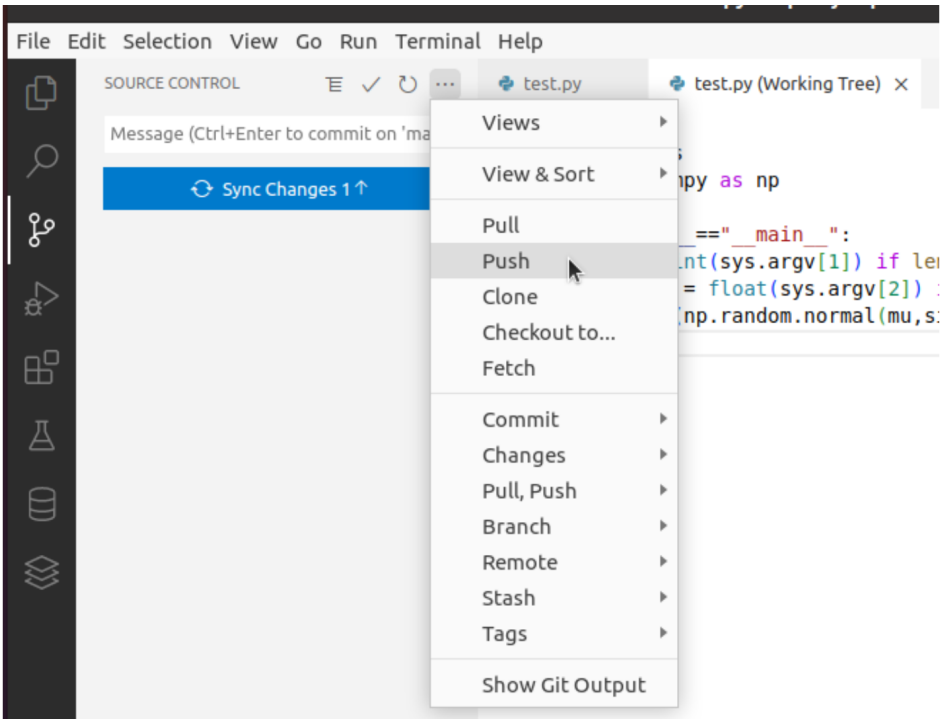


Figura 34: Empujando los cambios al remoto con VS Code.

4.2. Resolviendo un merge conflict

En esta subsección veremos como resolver un problema de versiones en un jupyter notebook con la ayuda de VS Code. Para usar jupyter dentro de VS Code debemos tener instalada la extensión correspondiente. El caso de uso es el siguiente partimos de un Notebook de base como el de la figura 35. Personas A y B tienen esta misma versión en sus repositorios locales.

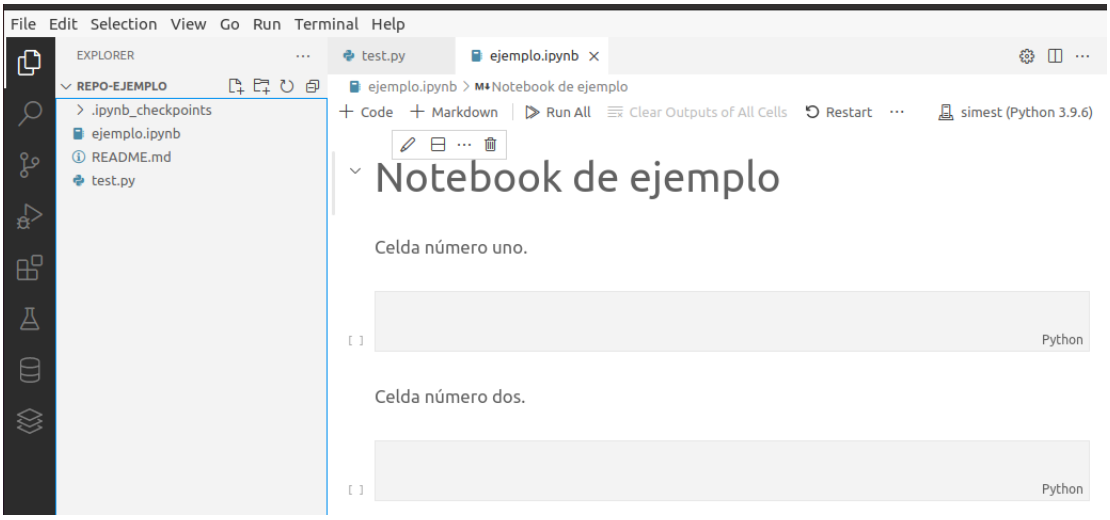


Figura 35: Notebook de base a editar.

Imaginemos lo siguiente:

- Persona A edita la celda número dos (figura 36).
- Persona B (en otro computador) edita la celda número uno, al mismo tiempo (figura 37).
- Persona A hace un commit y un push con sus cambios.
- Persona B hace un commit.

En este punto persona B no puede hacer un push, pues las versiones del mismo archivo son distintas.

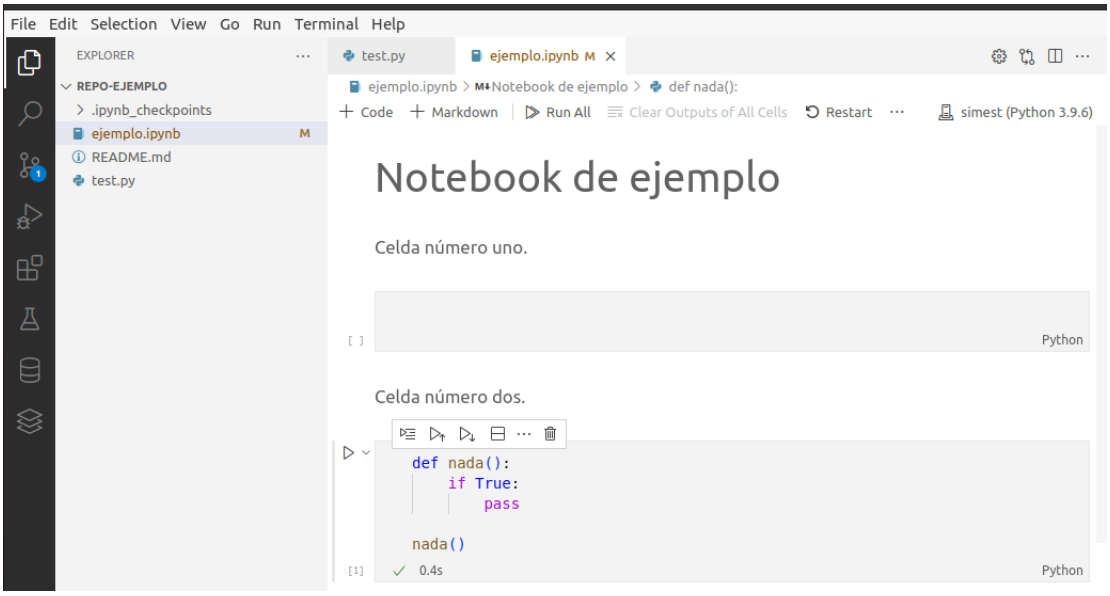


Figura 36: Notebook editado por persona A.

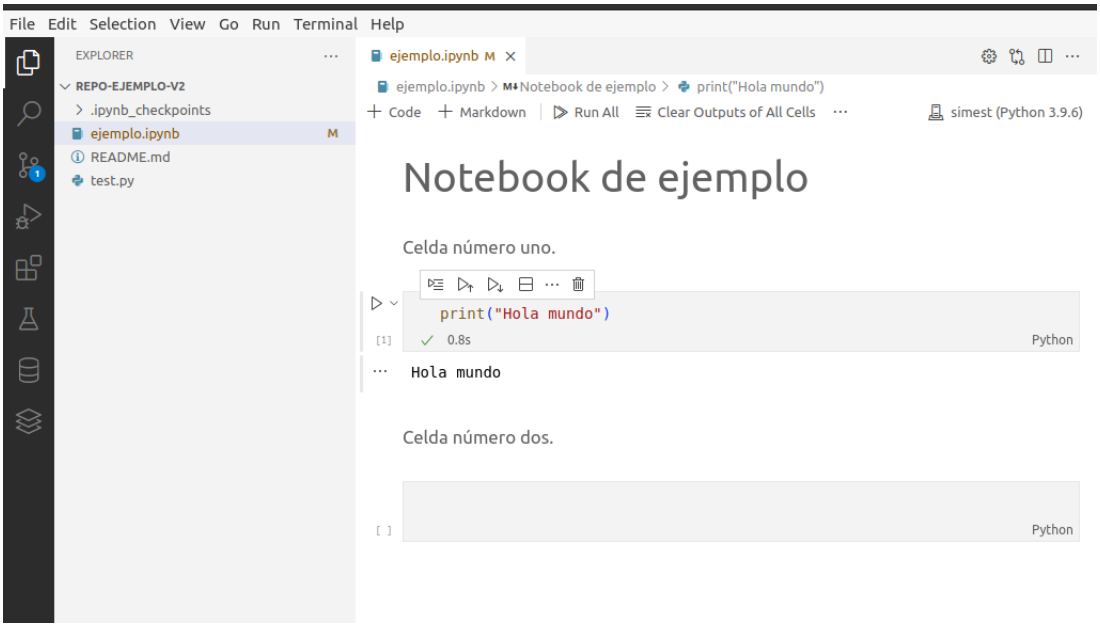


Figura 37: Notebook editado por persona B.

La persona B ahora ejectuta un pull. Esto generará un merge conflict, como se muestra en figura 38, que resolveremos usando VS Code.

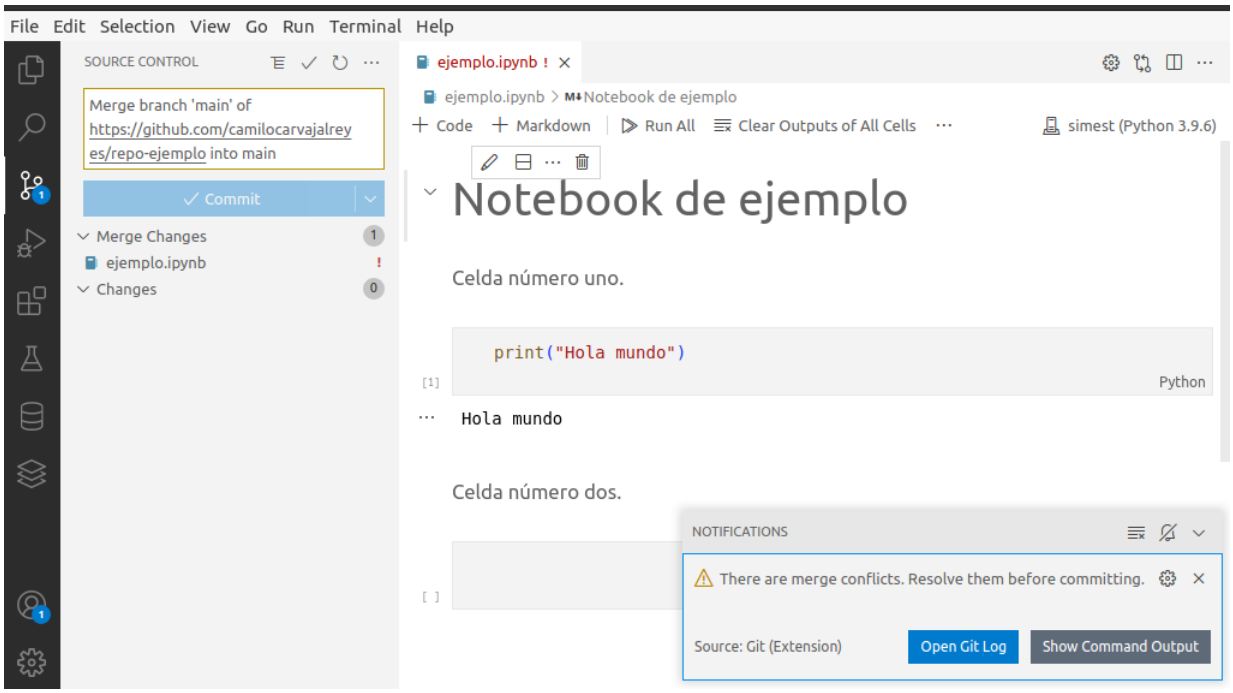


Figura 38: Notebook editado por persona B.

Haremos lo siguiente:

1. En la sección Source Control haremos click al nombre del archivo en conflicto, que aparecerá con un signo "!".
2. En una ventana se abrirá un editor que nos permitirá seleccionar que partes preferimos guardar en cada caso entre las versiones A y B.
3. Nos concentraremos en aquellas partes del código que representan contenido y output de celdas de jupyter. Primero nos fijamos que estemos seleccionando correctamente el código escrito por B (figura 39), lo cual hacemos presionando las casillas que aparecen a la izquierda del código.
4. Luego bajaremos a la celda posterior y seleccionamos el código escrito por persona A (figura 40).
5. Cliqueamos Accept Merge (botón azul inferior).
6. Presionamos Commit con el mensaje de merge que se coloca automáticamente.
7. Hacemos push.

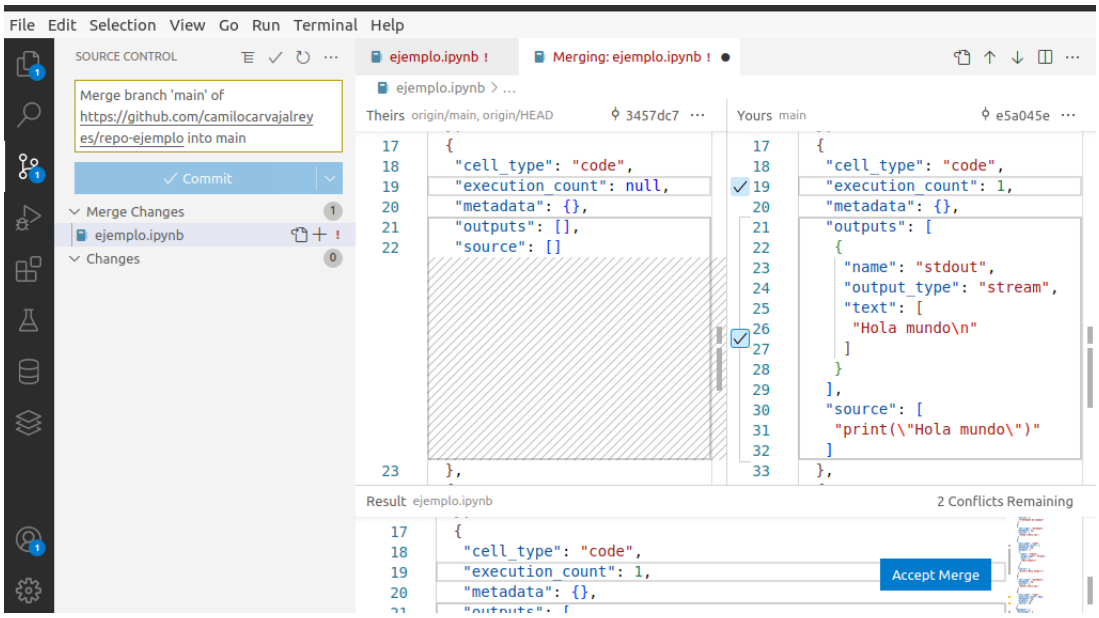


Figura 39: Aceptando cambios por persona B.

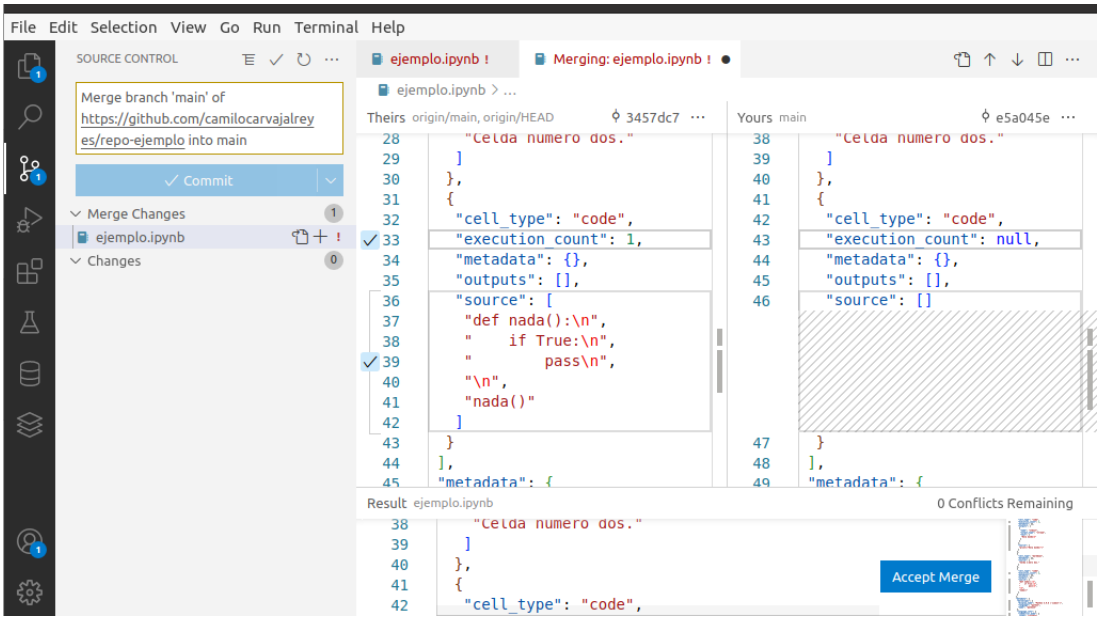


Figura 40: Aceptando cambios por persona A.

Ahora deberíamos tener una versión donde ambos cambios se vean al mismo tiempo. Podemos observar esto en el remoto de github, como en figura 41, en el cual es fácil visualizar el contenido de notebooks de jupyter. Persona A debe realizar un pull para finalizar el proceso.

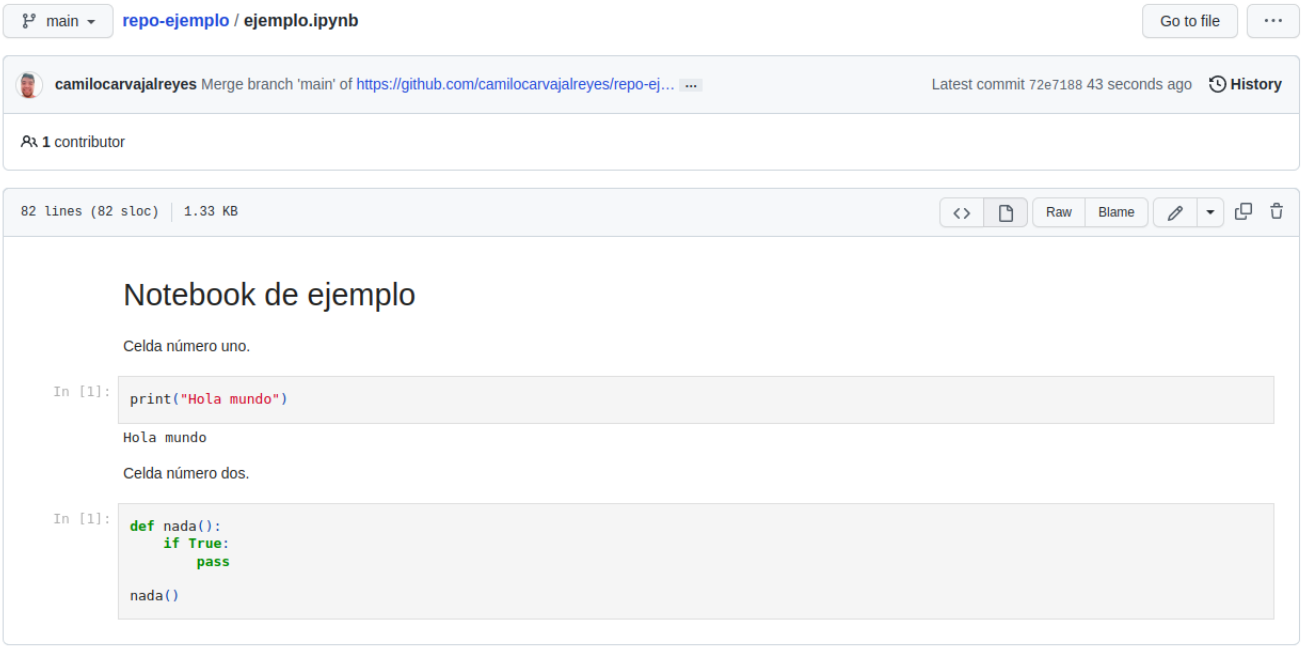


Figura 41: Notebook desde github luego del merge.