

Guía práctica para uso de git y GitHub

Para mancos

Por: Alarcon Lasagno Ramiro

Antes que nada:

Leer atentamente:

Esta guía es una recopilación ordenada de los más útiles comandos para manejarse en git que encontré. No son todos los comandos. Muchas de las definiciones las obtuve de esta página la cual contiene más comandos de los que voy a mostrar aquí, pero de una forma más desordenada.

<https://davidinformatico.com/apuntes-git/>

Por último, hay al menos dos maneras de leer esta guía:

- Para aquel que no sabe nada del tema le recomiendo leerla de inicio a fin sin saltarse nada los conceptos se explican de forma muy simple y entendible.
- Para aquellos mas experimentados con git que solo estén buscando responder algunas cuestiones o resolver algún problema en la pagina siguiente se encontrara un índice con un titulo seguido de una parte del código. Este índice está linkeado con la pagina para no perder tiempo. En cada capitulo se encontrará una tabla con los códigos para no tener que leer necesariamente el texto que lo acompaña.
- Por ultimo voy a aclarar que cuando se escribe <alguna cosa entre mayor y menor> para un comando, significa que el contenido es variable, pero en ningún comando se usan los simbolos (<>) o (><)

Índice:

<i>¿Qué es git?</i>	4
<i>¿Qué debemos saber para trabajar con git?</i>	5
Conceptos básicos:	5
• Working dir/Área de trabajo:.....	5
• Staging Area/Index/Área de ensayo.....	5
• El repositorio local	5
• Los repositorios remotos.....	5
• *Stash	5
• tree/snapshot	5
• rama	5
• rama master.....	5
• conflicto:.....	6
• HEAD	6
• tag:	6
<i>Descargar y configurar git:</i>	7
Manos a la obra (Un ejemplo práctico)	8
<i>Crear un repositorio vacío (init)</i>	9
Comandos usados:	11
<i>Cargar un archivo al stage (add)</i>	12
Comandos usados:	15
<i>Ignorar archivos:</i>	16
<i>Cargamos los cambios del stage en el repositorio (commit y status)</i>	18
El comando a usar:	18
A tener en cuenta:	18
Caso de ejemplo:	19
Comandos usados:	23
<i>Visualizar ramas y arboles (log)</i>	24
Comandos usados:.....	26
<i>Resumir comandos</i>	27
Un dato de color:	27
<i>Viajar por las ramas (checkout)</i>	28
Breve explicación:	28
Recordamos:	28
Importante	29
Viajando a la primera versión del programa:	30
Comandos usados:	35

Crear, eliminar y unir una rama:	36
Crear una rama:	36
Eliminar una rama:	36
Unir ramas:	36
Conflicto típico:	37
Comandos usados:	43
Comandos útiles:	44
Ver el historial de comandos:	44
Información sobre comando:	44
Comparar el ultimo commit con el directorio de trabajo:	44
Comparar commit actual con commit anterior:	44
Borrar archivo a través de ventana de comandos:	45
Etiquetas (tag)	46
Etiquetando un commit antiguo:	46
Etiquetar el HEAD	47
Ver todas las etiquetas:	47
Eliminar una etiqueta:	48
Ver contenido de etiqueta:	48
Repositorio remoto (git hub)	50
Crear un repositorio:	50
Cargar proyecto en repositorio:	52
Clonar repositorio de otra persona:	53
Fin	55
Comandos usados en toda la guía:	56

¿Qué es git?

Git es un método de guardado y archivado de proyectos (comúnmente programas de computadora) el cual tiene la característica de no solo salvar el programa sino también las diferentes modificaciones que se van haciendo a medida que se va desarrollando el mismo.

¿Para qué sirve?

Si se es un programador con experiencia uno tiende a reusar código. Mucho de este código se modifica o desaparece o incluso se rompe y deja de funcionar cuando uno mas lo necesita y como humanos tendemos a olvidar donde los guardamos si es que tenemos una gran cantidad archivador.

Aquí es donde git se vuelve útil. Por ejemplo:

- Supongamos que hacemos un trabajo donde debemos controlar una cámara para que este tome imágenes cada cierto tiempo.
- Meses después debemos usar un programa similar pero que en vez de tomar imágenes que haga reconocimiento de patrones
- Meses después un programa que controle una cámara para reconocer caras
- Y otra vez un programa para que de imágenes reconozca la cara de ciertas personas y las guarde en otra carpeta según que personas hay.

En todos estos programas se reúsan líneas de código. ¿Y que pasaría si sin querer se borran algunos programas, o si al modificar el primer programa para que reconozca patrones borro las líneas que le permiten almacenar imágenes?

Es cuando git aparece y te dice, podrías tener todos esos programas no solo en la misma carpeta, sino que además ocupando un mismo árbol de programas por el que puedas viajar recorriendo todas las modificaciones sin perder ningún programa.

No solo eso, podrías almacenar todo eso en la nube y descargarlo o compartirlo o modificarlo de forma grupal para proyectos grandes.

Pero antes de eso deberás de aprender algunos conceptos.

¿Qué debemos saber para trabajar con git?

Al trabajar con git se deben tener en cuenta que se estará trabajando con cinco áreas de trabajo.

Conceptos básicos:

- **Working dir/Área de trabajo:** Es el estado actual de nuestros ficheros, aquellos en los que programamos y que podemos visualizar en nuestro navegador de archivos.
- **Staging Area/Index/Área de ensayo:** Es una zona intermedia donde vamos almacenando los cambios que van a conformar un commit. Pueden ser ficheros completos o solo porciones concretas. Para “pasar” ficheros a esta zona usamos el comando `add`. Necesariamente hay que enviar los cambios al staging area antes de poder realizar un commit.
- **El repositorio local:** Es donde se almacenan los commits mediante el comando `commit`, una vez hemos seleccionado los cambios en el index. Toda su información se guarda en el *directorio .git*.

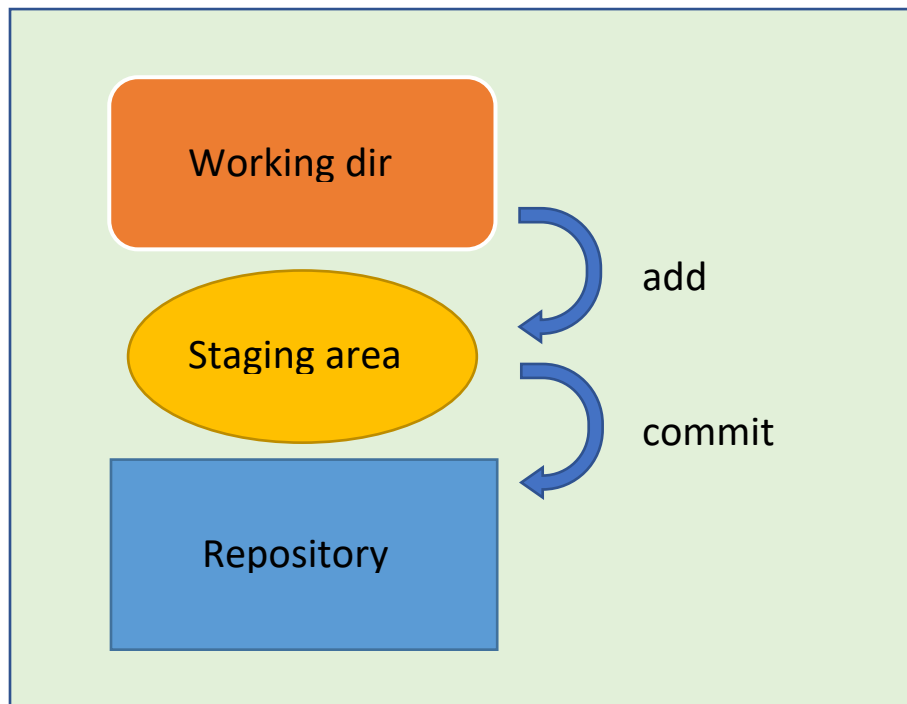
Mas tarde veremos otros dos:

- **Los repositorios remotos:** Puede ser uno solo (por defecto se llama origin), o podríamos tener varios. La comunicación entre el repositorio local y los remotos se realiza mediante los comandos `push`(enviar) y `pull`(descargar).
- ***Stash:** Es la zona de guardado rápido, una zona auxiliar para guardar los cambios en los que estamos trabajando cuando por algún motivo nos interrumpen y tenemos que cambiar de rama, pero aún no queremos hacer un commit porque es un commit a medias, sin acabar. Puede almacenar n estados y funciona como una pila, colocando siempre el primero los últimos cambios que salvemos.

También aprenderemos sobre:

- **tree/snapshot:** es una estructura de árbol que representa el estado de todos los ficheros del proyecto en un momento concreto en el tiempo. Cada commit guarda la referencia a un snapshot.
- **rama:** Una rama es una bifurcación en la línea de desarrollo del proyecto, que tendrá su propio historial de commit y estará separada por completo de la rama por defecto. Se usan para implementar nuevas funcionalidades sin afectar a la versión estable del proyecto.
Técnicamente una rama es un puntero al último commit de una línea de desarrollo. Cuando creamos una rama, simplemente estamos creando un nuevo puntero que podemos mover libremente respecto al resto de ramas.
- **rama master:** Es la rama creada por defecto.

- **conflicto:** Sucede cuando se unen dos ramas en las cuales se modifican los mismos trozos de un mismo fichero y git no sabe cuál de las dos elegir, por lo que somos nosotros quienes debemos tomar la decisión para resolverlo.
- **HEAD:** También es un puntero, que apunta al último commit de la rama en la que te encuentres trabajando actualmente, salvo que hayamos movido HEAD a propósito a un momento en el pasado.
- **tag:** Es un nombre con un puntero a un momento concreto en la historia del desarrollo. Se suelen usar para identificar los momentos en que se lanzan nuevas versiones. (Para cuando tenés demasiadas versiones. Un apodo común es colocar V1.0, V2.0, V3.2.1, etc...)



Como ya se explicó, el working dir es exactamente lo que tenemos en la computadora que podemos ver y editar, el archivo en sí. El staging área es una unidad intermedia que se dedica a mirar al working dir y a compararlo con el repositorio. Te avisa si modificaste el archivo o si hay algún archivo nuevo que esta en el working dir pero no en el repositorio, y también sirve de filtro para lo que vas a guardar en el repositorio. Ya que no tenés que guardar todo al mismo tiempo, podes guardar por separado un programa de varios archivos para que las modificaciones se tomen aparte. (eso es algo que con la practica puede ser muy útil)

Mas tarde se introducirán conceptos de ramas, árbol para viajar entre distintos repositorios y finalmente como guardar en la nube usando GitHub.

Descargar y configurar git:

Descargar:

<https://git-scm.com/downloads>

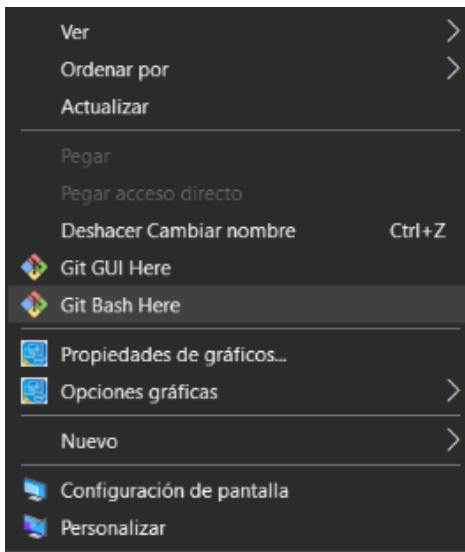
Descargamos git y lo instalamos con las opciones por defecto que trae.

Seguidamente en cualquier lugar de la pantalla o si queremos ya en la carpeta en la que vamos a trabajar comenzamos con la configuración:

Primeros pasos:

Fuente:

<https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Configurando-Git-por-primera-vez>



Seleccionamos Git Bash Here que es un cuadro emulador de Linux por lo que mientras más sepas de Linux mejor lo vas a usar.

Tu Identidad

Lo primero que deberás hacer cuando instales Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque los "commits" de Git usan esta información, y es introducida de manera **immutable** en los commits que envías:

Te recomiendo usar un nombre serio y una dirección seria de mail.

Deberás colocar el código siguiente colocando entre comillas tu nombre o el nombre con el que quieras que sea reconocido tu trabajo a nivel laboral.

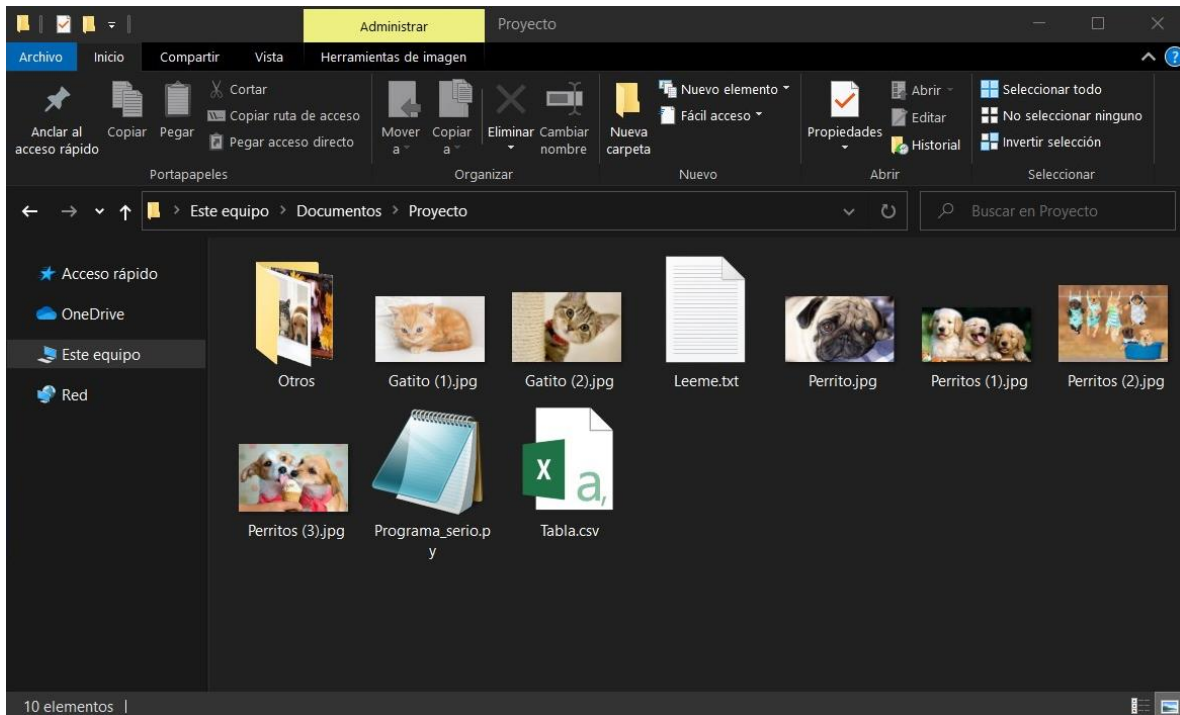
```
$ git config --global user.name "John Doe"
```

Seguidamente colocaras una dirección de mail al que será referenciado tu trabajo. Esto ahora no te será útil pero cuando quieras crear un portafolios online se vera su valor. (en este caso la dirección de correo electrónico va sin comillas).

```
$ git config --global user.email johndoe@example.com
```

Manos a la obra (Un ejemplo práctico)

Si es tu primera vez usando git puede que luego de instalar y configurar debas de crear tu primer repositorio. Para esto vas a ir a una carpeta donde tengas algún proyecto.



Vamos a suponer por un momento que soy un programador medio decente y que me propongo crear un programa en Python que, por ejemplo, que de una imagen me pueda reconocer si lo que hay en una foto es un perrito o un gatito.

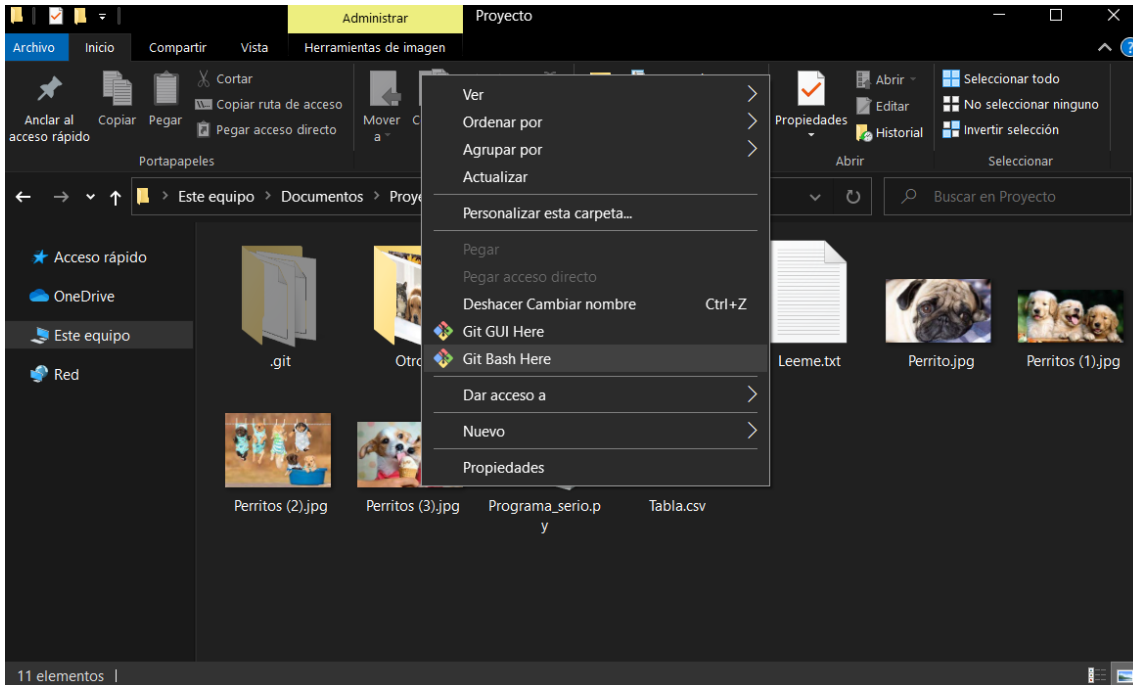
No tengo idea como hacer ese programa, pero asumo que quien lo cree va a contar con al menos un archivo (.py) un archivo para datos (.csv), algún archivo Leeme.txt para anotar detalles de su programa o destinado al usuario y fotos de perritos y gatitos a montones. Dentro de la carpeta Otros coloque también imágenes que contienen perritos y gatitos al mismo tiempo para probar al programa.

Claramente no voy a programar esto, aunque se podría. Pero la idea es mostrar:

- Como crear un repositorio vacío en esta carpeta.
- Como cargar el proyecto al stag (zona intermedia), tanto el programa como todo lo que este necesita para funcionar.
- Y como guardarlo de forma definitiva en un repositorio.

Crear un repositorio vacío (init)

Para esto debemos de ingresar en la carpeta del proyecto, botón derecho-git bash here



Aparecerá una ventana de comandos y comienzo escribiendo

- git status

Luego explicare su uso real cuando ingresemos los primeros archivos, pero de momento es para chequear si existe o no repositorio. Podemos ver que claramente no existe.

```
fatal: not a git repository (or any of the parent directories): .git
```

Seguidamente escribo el commando:

- git init

```
MINGW64:/c:/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto
$ git status
fatal: not a git repository (or any of the parent directories): .git

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto
$ git init
Initialized empty Git repository in C:/Users/ramir/Documents/Proyecto/.git/

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

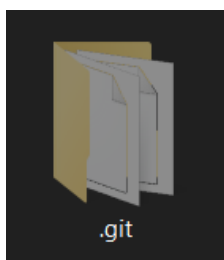
El cual nos devuelve:

Repositorio Git vacío inicializado en C: /Users/ramir/Documents/Proyecto/.git/

Importante: El antivirus o la seguridad de Windows puede que no permita que se genere o se trabaje con git, por lo que se deberá de desactivar la protección en tiempo real o generar excepciones que permitan usar el mismo.

Luego de eso ya tendremos un repositorio en nuestra carpeta.

Podremos notarlo porque en modo oculto aparecerá una carpeta (.git) la cual no debemos borrar ya que se borraría el repositorio. O haciendo "ls -la" en la línea de comandos:



Si borras esta carpeta se borrarán todas las modificaciones hechas y solo quedará lo que hay en tu espacio de trabajo en el momento en que borraste la carpeta.

Esto puede ser útil para aprender, ya que puedes probar códigos sin miedo. Pero no es aconsejable cuando ya estás trabajando seriamente porque no conviene perder todas las modificaciones si no se cuenta con un backup.

```
MINGW64:/c:/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ ls -la
total 260
drwxr-xr-x 1 ramir 197609  0 nov. 13 16:40 ./
drwxr-xr-x 1 ramir 197609  0 nov. 13 16:12 ../
drwxr-xr-x 1 ramir 197609  0 nov. 13 16:40 .git/
-rw-r--r-- 1 ramir 197609 4520 nov. 13 16:14 'Gatito (1).jpg'
-rw-r--r-- 1 ramir 197609 6975 nov. 13 16:14 'Gatito (2).jpg'
-rw-r--r-- 1 ramir 197609  0 nov. 13 16:15 Leeme.txt
drwxr-xr-x 1 ramir 197609  0 nov. 13 16:20 Otros/
-rw-r--r-- 1 ramir 197609 8334 nov. 13 16:12 Perrito.jpg
-rw-r--r-- 1 ramir 197609 8018 nov. 13 16:12 'Perritos (1).jpg'
-rw-r--r-- 1 ramir 197609 9563 nov. 13 16:13 'Perritos (2).jpg'
-rw-r--r-- 1 ramir 197609 7210 nov. 13 16:13 'Perritos (3).jpg'
-rw-r--r-- 1 ramir 197609  0 nov. 13 16:15 Programa_serio.py
-rw-r--r-- 1 ramir 197609 180224 nov. 13 16:16 Tabla.csv

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Ahora que ya creaste tu repositorio vamos a incluir los primeros archivos.

Comandos usados:

\$ git init
Crea un repositorio local vacío en el directorio actual. Este se almacena en el directorio “.git”.
\$ git status
Este es el más usado de todos. Nos dice la rama en la que nos encontramos y los ficheros que contiene el stage junto a su estado (new/modified/deleted). Además, avisa de ficheros sin seguimiento por git (untracked) o ficheros con conflictos.

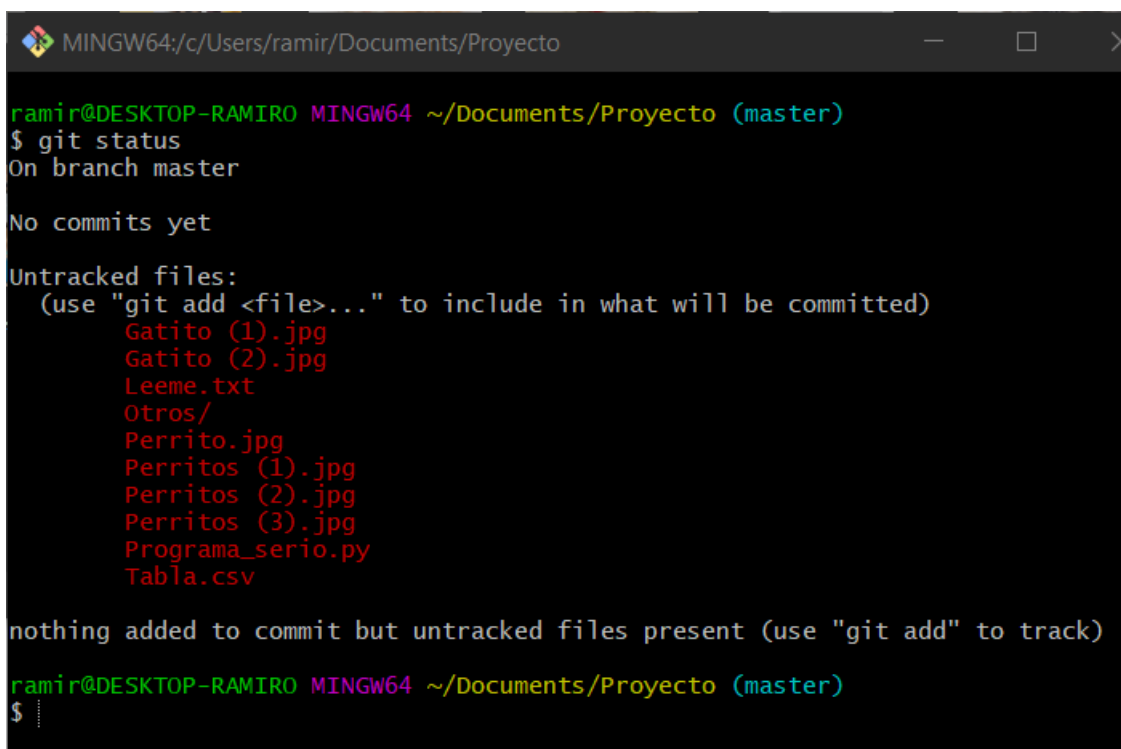
Cargar un archivo al stage (add)

Cargar un archivo en un repositorio local no es una tarea instantánea, pero dependiendo que desees cargar puede ocupar un mínimo de dos líneas de código o más según la cantidad y variedad de archivos que queramos cargar.

Como ya indicamos antes (y si soy recursivo en esto porque es importante) entre el directorio de trabajo y el repositorio existe una zona intermedia llamada área de ensayo o stage en la cual decidiremos cuales de todos los archivos que se encuentran en la carpeta de nuestro proyecto queremos cargar al repositorio para guardar.

Ya habiendo creado nuestro repositorio vamos a volver a utilizar

```
$ git status
```



```
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Gatito (1).jpg
    Gatito (2).jpg
    Leeme.txt
    Otros/
    Perrito.jpg
    Perritos (1).jpg
    Perritos (2).jpg
    Perritos (3).jpg
    Programa_serio.py
    Tabla.csv

nothing added to commit but untracked files present (use "git add" to track)
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Lo que vemos en Untracked files (archivos sin seguimiento) es a todos los archivos y carpetas que no fueron cargados a git y por ende sus cambios no fueron guardados.

Para cargar por ejemplo “Programa_serio.py”

```
$ git add Programa_serio.py
```

o

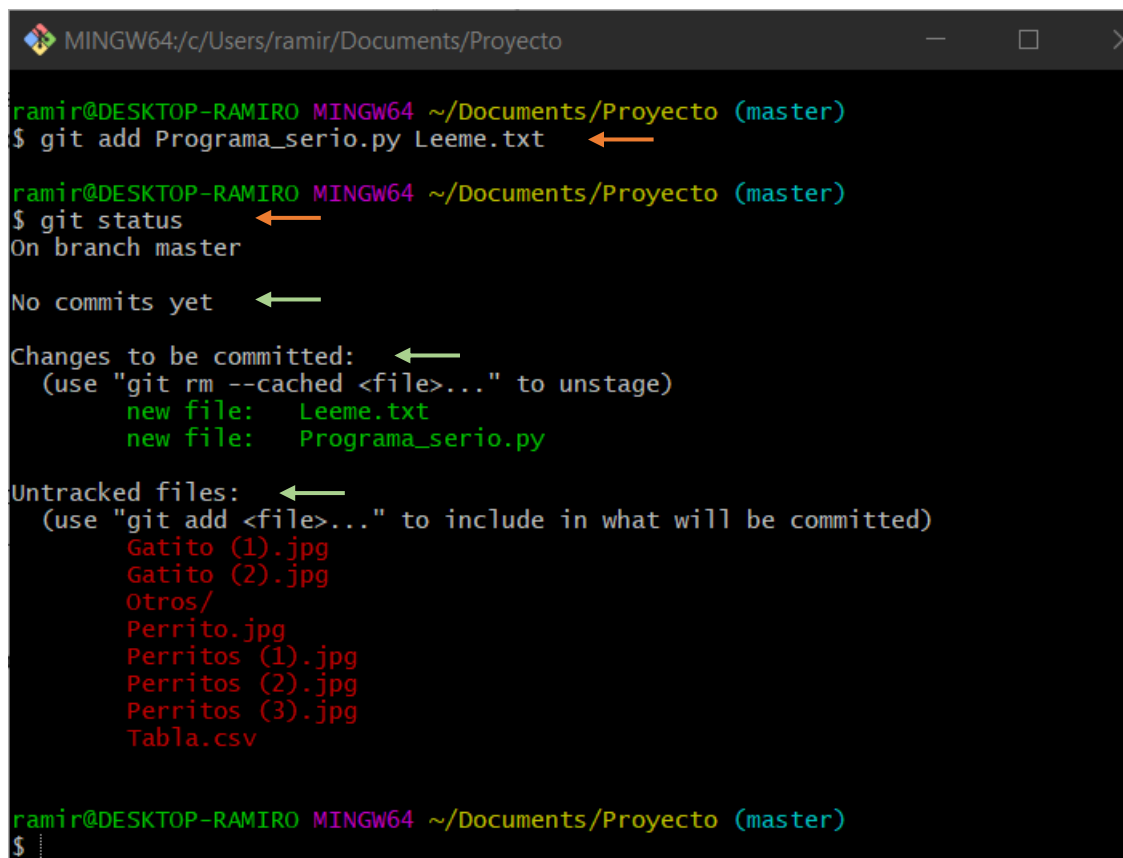
```
$ git add "Programa_serio.py"
```

Las comillas solo son necesarias en el caso de que el nombre del archivo tenga espacios, caso contrario pueden estar como no.

Si quisiera gargar por ejemplo tanto Programa_serio.py como Leeme.txt, podría usar una sola línea de comandos así:

```
$ git add Programa_serio.py Leeme.txt
```

Notar que Programa_serio.py y Leeme.txt fueron separados por un espacio.



```
MINGW64:/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git add Programa_serio.py Leeme.txt

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git status
On branch master

No commits yet

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

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Gatito (1).jpg
    Gatito (2).jpg
    Otros/
    Perrito.jpg
    Perritos (1).jpg
    Perritos (2).jpg
    Perritos (3).jpg
    Tabla.csv

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

En naranja los comandos.

Luego de cargar los archivos en el stag tanto si fuera junto o separado. Vamos a hacer

```
$ git status
```

Ahora se mostrara un cartel que indica que no hay comits aun, otro que indica los cargados para ser comprometidos o guardados en el repositorio (committed) y por ultimo los que aun no han sido cargados al stage

En el caso en que confundamos al cargar, dado que aun no guardamos en el repositorio, podemos arrepentirnos haciendo:

```
$ git reset
```

Este comando eliminara todo lo hecho con add siempre que no se haya guardado en el repositorio.

Otras formas de almacenar en el stage.

- \$ git add . (espacio y punto: coloca todos los archivos de la carpeta en el stage, incluso otras carpetas)
- \$ git add *.jpg (Útil para cargar solo un tipo de archivo, puede ser jpg como cualquier cosa.)
- \$ git add Otros/ ("Nombre_carpeta/" en nuestro ejemplo la carpeta se llama Otros, no olvidar el guion para indicar que es una carpeta)
- \$ git add -A (Similar a "git add ." revisa todo lo que esta dentro del directorio de en que creamos git y lo sube al stage)
- \$ git add Otros/*.jpg (Para el caso particular en que dentro de la carpeta otros haya otros archivos que no sean jpg y que no me interesen colocar en el stage.)

Notar en este ultimo caso que si colocamos un espacio entre "Otros/" y "*.jpg" va a cargar tanto los archivos en la carpeta "Otros" sin filtrar como los archivos ".jpg" dentro de la carpeta principal. Es por esto que es muy importante y útil seguir usando el comando.

- \$ git status

Siempre que se pueda. En caso de equivocación volvemos a usar.

- \$ git reset
- \$ git reset *.jpg
- \$ git reset Otros/
etc...

Comandos usados:

\$ git add <fichero>
Manda uno o varios ficheros separados por espacios o un directorio al staging area. Recién añadido un nuevo fichero al repositorio, este se encuentra en estado “untracked” y debemos hacer un primer add.
\$ git add .
Manda todos los ficheros del directorio al staging area.
\$ git add -A
Manda todos los ficheros del directorio al staging area.
\$ git add -all
Manda todos los ficheros del directorio al staging área.
\$ git add *.<formato>
*.doc; *.txt; *.py; etc...
\$ git add <NombreCarpeta>/
Manda todos los ficheros de una carpeta del directorio principal al staging area.
\$ git reset
Con estos comandos podemos sacar un fichero de la zona Index.
\$ git reset *<formato>
Variación para formatos
\$ git reset <NombreCarpeta>/
Variación para carpetas.
\$ git status
Este es el más usado de todos. Nos dice la rama en la que nos encontramos y los ficheros que contiene el stage junto a su estado (new/modified/deleted). Además, avisa de ficheros sin seguimiento por git (untracked) o ficheros con conflictos.

Ignorar archivos:

Suele pasar que uno tiende a trabajar con

```
$ git add .
```

Pero siempre hay algún archivo o carpeta que no pertenece a nuestro proyecto pero que tampoco (por alguna razón) podemos quitar de nuestra carpeta de proyecto. Para esos casos lo que podemos hacer es crear un archivo llamado “.gitignore” lo abriremos con el editor de texto y colocaremos el nombre de todos los archivos que deseemos ignorar.

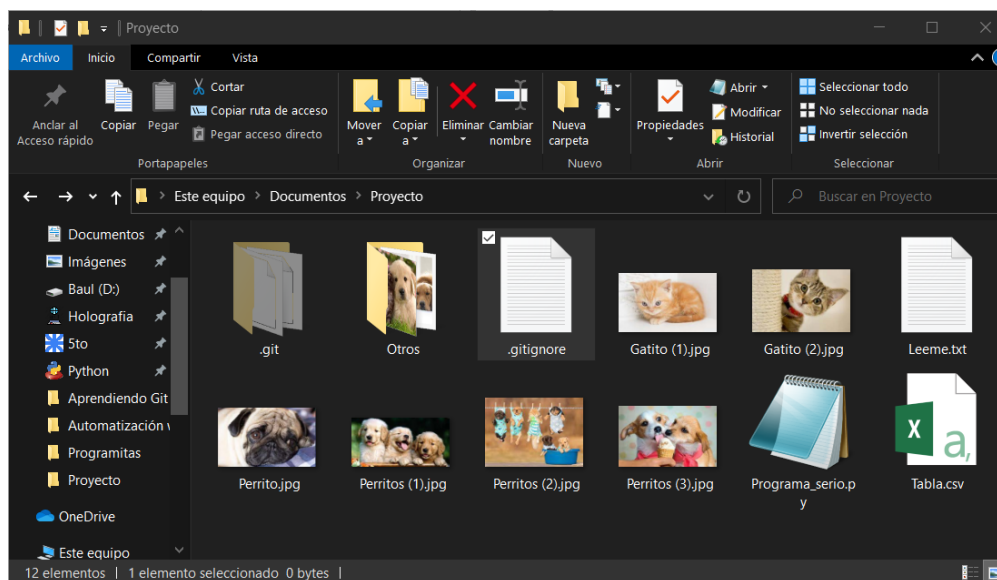
Ejemplo: Vamos a ignorar de git el archivo “Leeme.txt” y la carpeta “Otros”.

Primero vamos a verificar que git los reconoce.

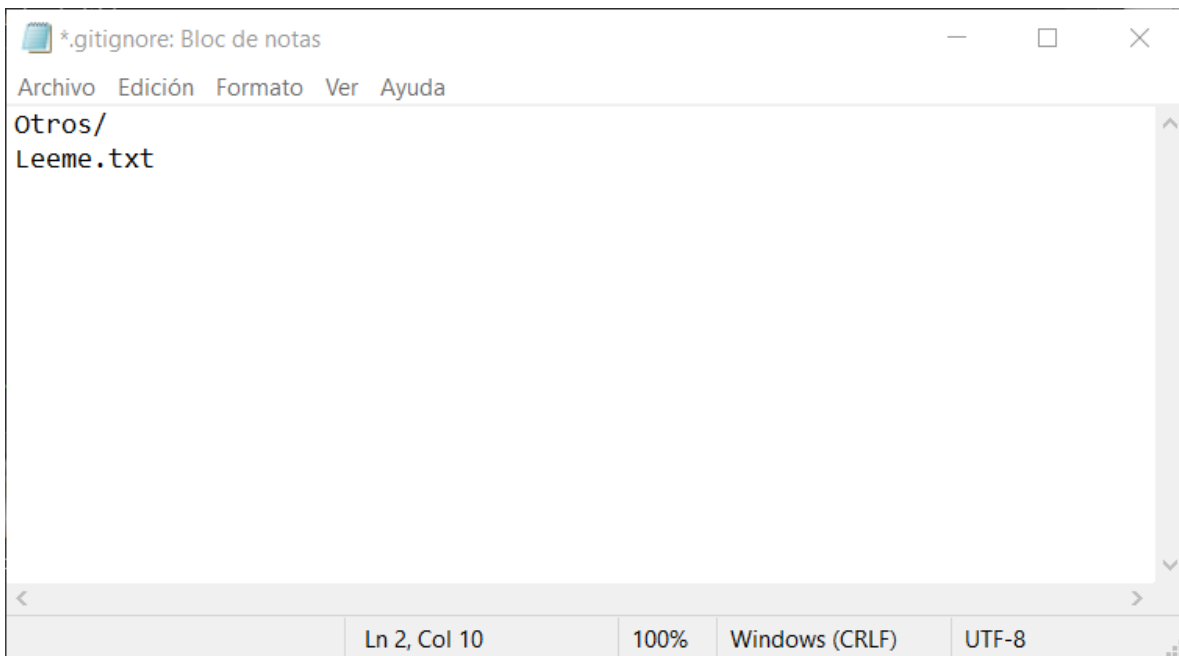
```
MINGW64: c:/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$ git status
HEAD detached at 161e105
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Gatito (1).jpg
    Gatito (2).jpg
    Leeme.txt
    Otros/
    Perrito.jpg
    Perritos (1).jpg
    Perritos (2).jpg
    Perritos (3).jpg
    Tabla.csv

nothing added to commit but untracked files present (use "git add" to track)
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$
```

Creamos el archivo “.gitignore”



Agregamos los archivos y carpetas que deseamos ignorar y guardamos



```
MINGW64:/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$ git status
HEAD detached at 161e105
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        Gatito (1).jpg
        Gatito (2).jpg
        Perrito.jpg
        Perritos (1).jpg
        Perritos (2).jpg
        Perritos (3).jpg
        Tabla.csv

nothing added to commit but untracked files present (use "git add" to track)
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$
```

Cargamos los cambios del stage en el repositorio (commit y status)

commit: Conjunto de cambios (changeset) sobre uno o varios ficheros del proyecto, que lleva asociado un breve mensaje descriptivo y un autor. Y que se identifican de forma única con una cadena del tipo “eda792...8bed6c” (41 caracteres), que no es más que un hash de tipo SHA-1 de la fecha, autor, mensaje, id del snapshot(árbol) en ese momento y el id del anterior commit (un commit guarda un enlace a su anterior commit o a varios commit si se trata de un merge de dos o más ramas). Como el id es un string muy largo, habitualmente se hace referencia al mismo usando solo los 7 primeros caracteres.

El comando a usar:

```
$ git commit -m “descripción breve”
```

Registra definitivamente los cambios que previamente estaban en el Stage. Un commit contiene:

- **id:** Identifica el cambio de forma única. Es el hash SHA-1 del resto de campos.
- **mensaje:** que habitualmente se trunca a 72 caracteres al visualizarlo en el log o github.
- **autor:** El autor del cambio identificado por su nombre y su email.
- **id del árbol o snapshot** en ese momento.
- **id del anterior commit.**

A tener en cuenta:

Generalmente

```
$ git add <fichero>
```

y

```
$ git commit -m “Descripcion de la modificacion”
```

Suelen ser dos comandos que se usan consecutivamente. La razón por la que no lo explique en el capítulo anterior es porque con commit podemos demostrar la utilidad de hacer add a archivos separados.

Es muy tentador cuando se trabaja hacer algo por el estilo:

```
$ git add .
```

```
$ git commit -m “Cargo archivos”
```

o

```
$ git add .
```

```
$ git commit -m "Nueva modificacion"
```

Pero se pierde la oportunidad de generar registros valiosos de las diferentes partes de un programa.

Caso de ejemplo:

Vamos a hacer un ejemplo con el Programas_serio.py

```
MINGW64:/c/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git add Programa_serio.py

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git commit -m "Primera version de programa de gatitos y perritos"
[master (root-commit) 85a10d9] Primera version de programa de gatitos y perritos
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Programa_serio.py

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Hacemos git status y obtenemos

```
MINGW64:/c/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Gatito (1).jpg
    Gatito (2).jpg
    Leeme.txt
    Otros/
    Perrito.jpg
    Perritos (1).jpg
    Perritos (2).jpg
    Perritos (3).jpg
    Tabla.csv

nothing added to commit but untracked files present (use "git add" to track)
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Observamos que ya no aparece información sobre los comits. Esto se da porque el comando status sirve para ver las modificaciones que suceden y que no están almacenadas en el repositorio. Como "Programa_serio.py" se encuentra en el repositorio, pero no esta modificado, no aparece nada sobre él.

Si luego hacemos hacemos

```
$ git add Leeme.txt
$ git status
```

```
MINGW64:/c:/Users/ramir/Documents/Proyecto

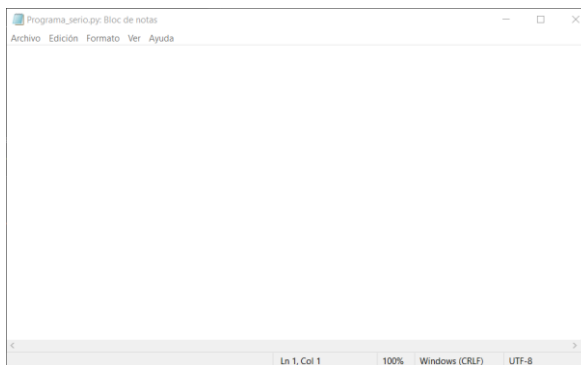
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git status
On branch master
Changes to be committed: ←
  (use "git restore --staged <file>..." to unstage)
    new file:   Leeme.txt

Untracked files: ←
  (use "git add <file>..." to include in what will be committed)
    Gatito (1).jpg
    Gatito (2).jpg
    Otros/
    Perrito.jpg
    Perritos (1).jpg
    Perritos (2).jpg
    Perritos (3).jpg
    Tabla.csv

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Por ultimo voy a modificar el archivo Programa_serio.py escribiendo cualquier tipo de código en él.

Antes



Después



Guardamos y volvemos a escribir el comando:

```
$ git status
```

```

MINGW64: c:/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   Leeme.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   Programa_serio.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Gatito (1).jpg
    Gatito (2).jpg
    Otros/
    Perrito.jpg
    Perritos (1).jpg
    Perritos (2).jpg
    Perritos (3).jpg
    Tabla.csv

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$

```

Ahora podemos ver toda la información que nos provee “git status”

- En la parte superior lo que se encuentra en el stage pero no fue guardado en el repositorio
- En el centro aquello guardado en el repositorio pero que sufrió modificaciones
- Por debajo aquello que no se encuentra ni en el repositorio ni en la zona intermedia.

Toda esta seguidilla de pasos es generalmente innecesaria. Pero es un buen método para ir recorriendo todos los cambios y entender la ventana de comandos.

Si en algún momento queremos limpiar la ventana podemos hacer uso del comando

```
$ clear
```

Este comando no deshará lo que hemos hecho, solo limpiará la ventana.

Ahora, para hacer “commit” a los nuevos cambios de “Programa_serio.py” debemos volver a cargarlo al stage. Sin embargo en el stage ya hay un archivo “Leeme.txt”.

Dependiendo de tu propia percepción de orden podrías hacerlo de la forma que quieras. Cada commit es como una versión o modificación del un proyecto. Si crees que todo tiene la misma importancia puedes hacer:

```

$ git add . (carga todos los archivos)
$ git commit -m "Cargamos todos los archivos con las modificaciones"

```

Si queremos que en este próximo commit solo se guarde la modificación de "Programa_serio.py" :

```
$ git reset  
$ git add Programa_serio.py  
$ git commit -m "Colocamos un print Hola mundo"
```

¿Qué hacemos si cuando hicimos la modificación nos equivocamos en algo, olvidamos un punto y coma o simplemente nos hubiese gustado agregar algo?.

Respuesta: Vamos a nuestro archivo, lo modificamos como queríamos y luego vamos a la ventana de comando y escribimos.

```
$ git commit -amend
```

Esto modifica el último commit con el contenido que tengamos en el Index, muy útil cuando se nos ha olvidado añadir algo en el commit o nos hemos confundido en la selección de cambios para incluir en el commit. *Realmente borra el commit y crea otro de nuevo. No funciona bien en git-scm ya que fue diseñado para Linux.

Y si por casualidad no fuimos precisos en la descripción breve del commit:

```
$ git commit --amend -m "corrección del mensaje"
```

También podría ocurrir que nos hayamos equivocado en el mensaje del commit.

Para hacer todo junto:

```
$ git commit -am "descripción"
```

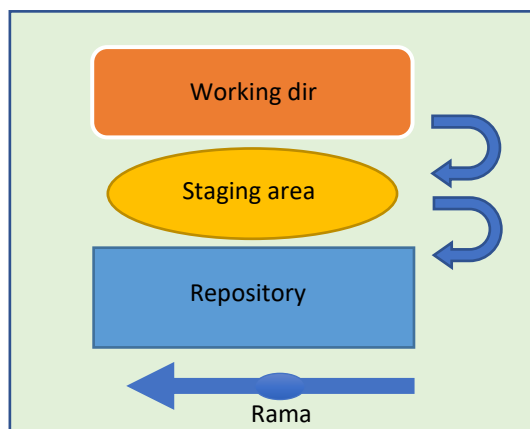
Hace un commit con todos los ficheros con cambios. Por tanto, esta opción a es para realizar el add previo y el commit de una sola vez. *Importante: solo funciona si el fichero ya está en seguimiento por git (tracked). (Solo para modificaciones)

Comandos usados:

\$ git add <fichero>
Manda uno o varios ficheros separados por espacios o un directorio al staging area. Recién añadido un nuevo fichero al repositorio, este se encuentra en estado “untracked” y debemos hacer un primer add.
\$ git commit -m "descripción breve"
Registra definitivamente los cambios que previamente estaban en el Stage. Un commit contiene
\$ git commit -am "descripción"
Hace un commit con todos los ficheros con cambios. Por tanto, esta opción a es para realizar el add previo y el commit de una sola vez. *Importante: solo funciona si el fichero ya está en seguimiento por git (tracked). (Solo para modificaciones)
\$ git commit --amend
Se modifica el último commit con el contenido que tengamos en el Index, muy útil cuando se nos ha olvidado añadir algo en el commit o nos hemos confundido en la selección de cambios para incluir en el commit. *Realmente borra el commit y crea otro de nuevo. (Solo recomendable en Linux)
\$ git commit --amend -m "corrección del mensaje"
Cuando nos hayamos equivocado en el mensaje del commit.
\$ git reset
Con estos comandos podemos sacar un fichero de la zona Index.
\$ git status
Este es el más usado de todos. Nos dice la rama en la que nos encontramos y los ficheros que contiene el stage junto a su estado (new/modified/deleted). Además, avisa de ficheros sin seguimiento por git (untracked) o ficheros con conflictos.

Visualizar ramas y arboles (log)

Para empezar a entender este tema debemos de dejar de pensar en una dimensión y pasar a pensar en dos dimensiones.



Ya explicamos como pasar de un directorio de trabajo a un área intermedia para poder finalmente pasarlo al repositorio. Pero ahora vamos a explicar:

- Como volver a una modificación anterior
- Como ramificar las modificaciones de un programa
- Como viajar por el árbol de modificaciones
- Conceptos de master y head

Como ya hemos explicado antes cada commit en el repositorio tiene asociado

- id: Identifica el cambio de forma única. Es el hash SHA-1 del resto de campos.
- mensaje: que habitualmente se trunca a 72 caracteres al visualizarlo en el log o github.
- autor: El autor del cambio identificado por su nombre y su email.
- id del árbol o snapshot en ese momento.
- id del anterior commit.

Para ver estos datos es que existen los comandos:

```
$ git log
$ git log --oneline
```

Importante: para salir de git log debes presionar la letra “q”

Sin embargo, log y log --oneline no dan mucha información acerca de a que rama pertenecen, y cuando las ramas se bifurcan es necesario al menos tener algún tipo de ayuda. Es por eso que otro comando muy útil es:

```
$ git log --oneline --decorate --graph
```

Con la opción graph podemos visualizar de forma “gráfica” las ramas y sus merges en el tiempo.

En el capítulo siguiente explicaremos mejor el tema de ramas.


```
MINGW64:/c:/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log
commit 12124debca3ff8720ea739e03960c0e739d19615 (HEAD -> master)
Author: Ramiro Alarcón Lasagno <ing.ramiro.a.l@gmail.com>
Date: Sat Nov 14 16:03:47 2020 -0300

    Colocamos un print Hola mundo

commit 85a10d90cde0b4c951a758f9383e6a7e913f4fc6
Author: Ramiro Alarcón Lasagno <ing.ramiro.a.l@gmail.com>
Date: Fri Nov 13 19:42:32 2020 -0300

    Primera version de programa de gatitos y perritos

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
12124de (HEAD -> master) Colocamos un print Hola mundo
85a10d9 Primera version de programa de gatitos y perritos

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Se usan por separado, “git log” nos da un registro detallado de los diferentes commit que generamos anteriores al “HEAD”

Por cada commit muestra el numero id, el autor de las modificaciones, la fecha en que se guardó el commit y la descripción breve con la que se registró.

En el caso de “git log –oneline” se nos da un registro reducido que solo nos muestra un id reducido que puede ser usado perfectamente para cualquier comando que requiera un id, un indicador de head, un indicador de la rama, en este caso la rama master y la descripción registrada.

Comandos usados:

\$ git log
Abre un listado que nos va a permitir ver el histórico completo de todos los commit que se han ido realizando en el repositorio a partir de HEAD hacia atrás, incluyendo los que nos traemos del remoto cuando hacemos un fetch. Funciona en modo interactivo y por defecto los últimos cambios aparecen los primeros.
\$ git log --oneline
Nos permite ver un commit por línea, mostrando solo el id y el mensaje.
\$ git log --oneline --decorate --graph
Con la opción graph podemos visualizar de forma “gráfica” las ramas y sus merges en el tiempo.

Resumir comandos

No te conviene saltearte esta sección.

Si llegaste al capítulo anterior habrás notado que muchos comandos son bastante largos y complicados de recordar.

Voy a darte un ejemplo de cómo darle un apodo a un comando para que no tengas que recordar los comandos mas largos.

El método es este:

```
$ git config --global alias.<apodo> "comando largo"
```

Voy a dar un ejemplo con un comando ya visto:

```
$ git config --global alias.lg "git log --oneline --decorate --graph"
```

Ahora solo debemos hacer

```
$ git lg
```

Si olvidamos que apodos creamos o estamos en la pc de alguien mas que tiene sus propios apodos podemos hacer:

```
$ git config --global -l
```

(Para leer)

```
$ git config --global -e
```

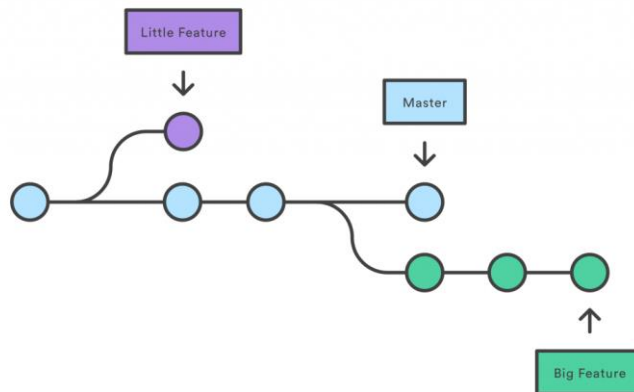
(Para modificar)

Un dato de color:

- 1 guion (-) se antepone a una letra
- 2 guiones (- -) se anteponen a una palabra

Viajar por las ramas (checkout)

Breve explicación:



Recordamos:

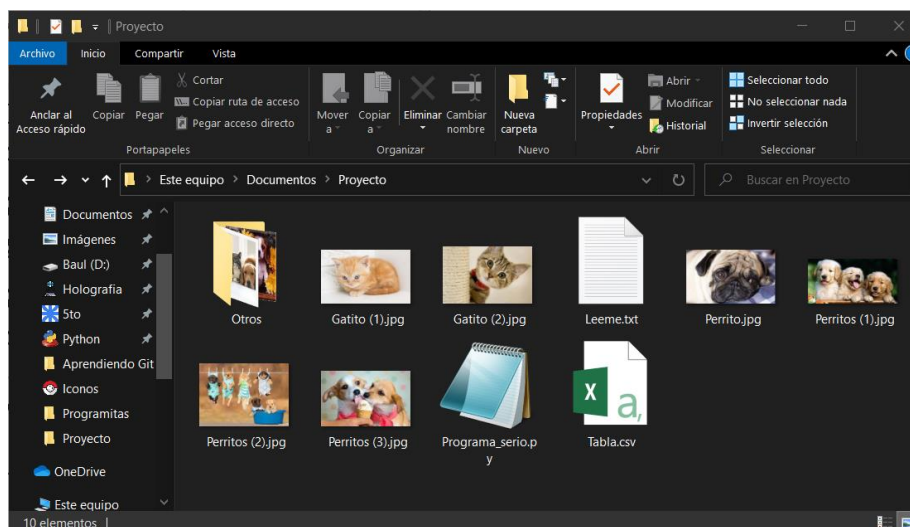
Una rama es una bifurcación en la línea de desarrollo del proyecto, que tendrá su propio historial de commit y estará separada por completo de la rama por defecto. Se usan para implementar nuevas funcionalidades sin afectar a la versión estable del proyecto.

Técnicamente una rama es un puntero al último commit de una línea de desarrollo. Cuando creamos una rama, simplemente estamos creando un nuevo puntero que podemos mover libremente respecto al resto de ramas.

Si llegaste a este punto ya debes saber como cargar un archivo al stage y de ahí guardarlo en el repositorio con su correspondiente descripción.

Si estas hiciste practica de todo lo anterior esta parte te va a gustar porque es todo práctica. Lo que vamos a hacer es borrar la carpeta “.git” y empezar de nuevo pero ya sin explicar nada. Vamos a crear nuestra rama master y luego usando “checkout” vamos a ramificarla.

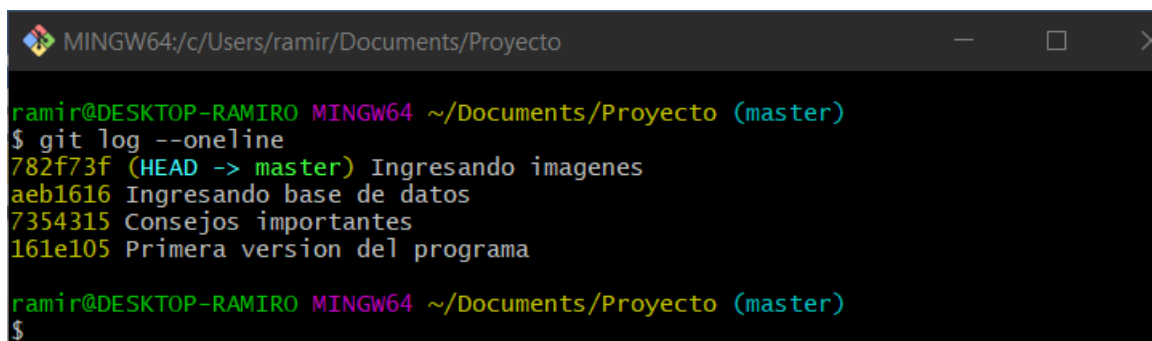
Recuerden que trabajamos con una carpeta como esta:



Luego de borrar la carpeta “.git” escribimos los comandos:

```
$ git init
$ git add Programa_serio.py
$ git commit -m "Primera versión del programa"
$ git add Leeme.txt
$ git commit -m "Consejos importantes"
$ git add Tabla.csv
$ git commit -m "Ingresando base de datos"
$ git add .
$ git commit -m "Ingresando imágenes"
$ git log --oneline
```

Deberá verse algo similar a esto.

A screenshot of a terminal window with a dark background. The title bar shows 'MINGW64:/c/Users/ramir/Documents/Proyecto'. The terminal text shows the user 'ramir@DESKTOP-RAMIRO' in the 'master' branch of the 'Proyecto' directory. They run 'git log --oneline', which displays four commit hashes and their messages: '782f73f (HEAD -> master) Ingresando imagenes', 'aeb1616 Ingresando base de datos', '7354315 Consejos importantes', and '161e105 Primera version del programa'. The prompt returns to '\$'.

Hasta ahora no hemos salido de la rama master. Por lo que vamos a movernos en la rama para crear una modificación y generar una rama alternativa.

Importante

Para eso utilizaremos “git checkout”

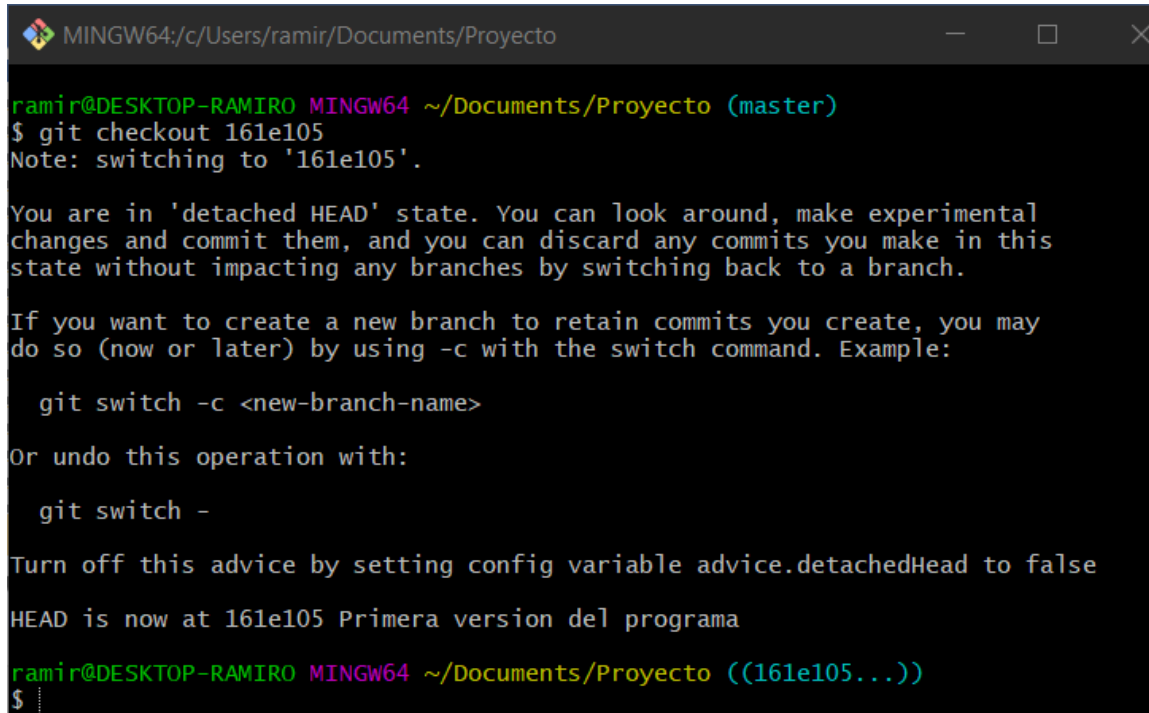
```
$ git checkout <id commit>
$ git checkout <nombre_rama>
```

Hay una cosa que se debe de tener en cuenta al usar checkout:

- Si te mueves a través de la rama **usando el id** de los commit, checkout hará dos cosas.
 - Ir a la modificación de ese id
 - Generar una nueva rama sin nombre en ese punto.
- ¿Por qué es importante saberlo? Porque si nos movemos a un punto de una rama y generamos modificaciones para luego hacer commit, esta modificación pertenecerá a una rama a la que habrá que nombrar (próximo capítulo) En todos los casos, cualquier modificación a la que hagas commit será en una rama aparte de la original.
- Si usas el nombre de la rama a la que quieres ir para volver al inicio entonces seguirás en la rama que elegiste y todas tus modificaciones.

Viajando a la primera versión del programa:

```
$ git checkout 161e105
```



```
MINGW64:/c:/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git checkout 161e105
Note: switching to '161e105'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false
HEAD is now at 161e105 Primera version del programa
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$
```

Estás en estado de 'HEAD separada'. Puede mirar a su alrededor, hacer cambios experimentales y confirmarlos, y puede descartar cualquier compromiso que realice en este estado sin afectar ninguna rama volviendo a cambiar a una rama.

Si desea crear una nueva rama para retener las confirmaciones que crea, puede

Hágalo (ahora o más tarde) usando -c con el comando switch. Ejemplo:

```
$ git switch -c <nombre-nuevo-branch>
```

O deshaga esta operación con:

```
$ git switch -
```

Como dijimos anteriormente si nos movemos con el id del commit debemos de crear una nueva rama, la ventana de comandos nos da la opción de colocarle nombre a esa nueva rama o deshacer el cambio lo que nos dejaría donde estábamos.

Si no deseamos generar una nueva rama podemos simplemente volver haciendo

- `git checkout master`
- `git checkout <nombre de la rama>`

Luego veremos de hacer la nueva rama, de momento solo haremos git log

```
$ git log
```

```
MINGW64/c/Users/ramir/Documents/Proyecto

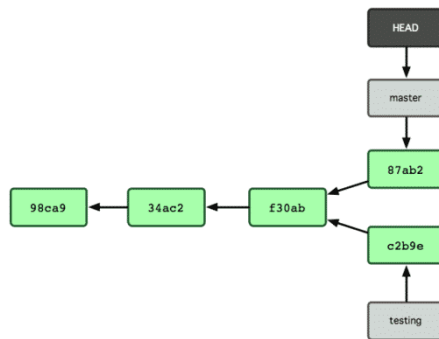
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$ git log
commit 161e105b379a56cc414d96d215dbd844eecdc3dc (HEAD)
Author: Ramiro Alarcón Lasagno <ing.ramiro.a.l@gmail.com>
Date: Sat Nov 14 17:51:24 2020 -0300

    Primera version del programa

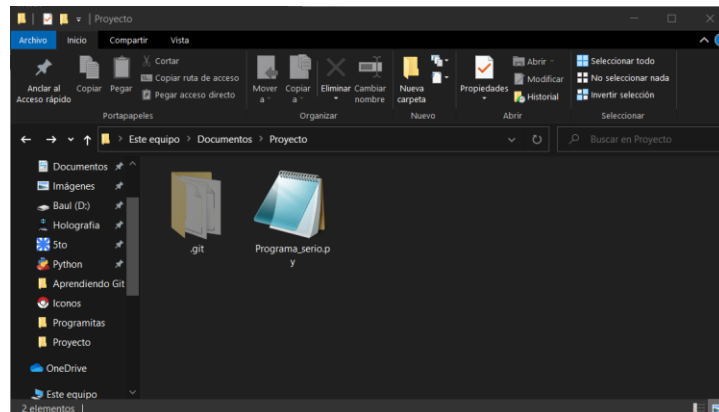
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$
```

Podemos ver tres cosas:

1. Que viajamos hacia el momento en que solo habíamos hecho commit del “Programa_serio.py” al que llamamos “Primera versión del programa”
2. Que al hacer log ya no podemos ver todos los commit que hicimos. Esto se debe a que log apunta a todos los commit anteriores al que se encuentra el HEAD que es donde estamos nosotros.



3. Al ver la carpeta del proyecto vamos a notar que solo se encuentra el archivo “Programa_serio.py”

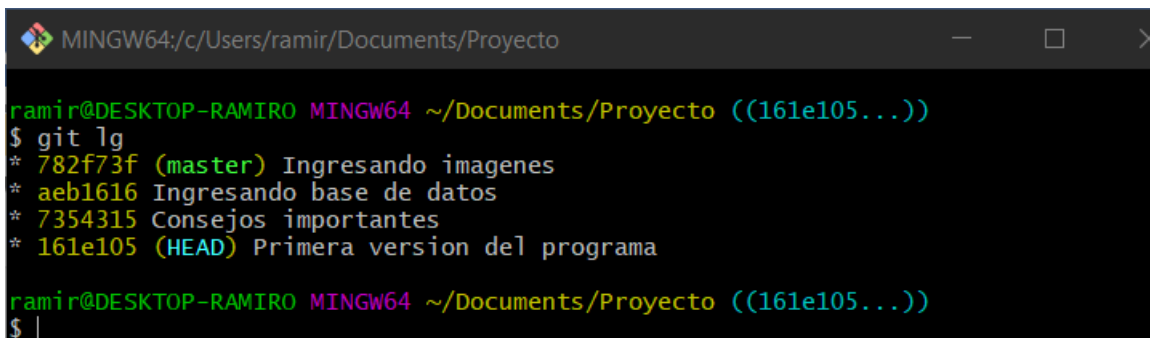


Independientemente de donde nos encontremos podemos ver las diferentes ramas usando:

```
$ git log --oneline --decorate --graph
```

O si ya aprendiste a resumir comandos:

```
$ git lg
```

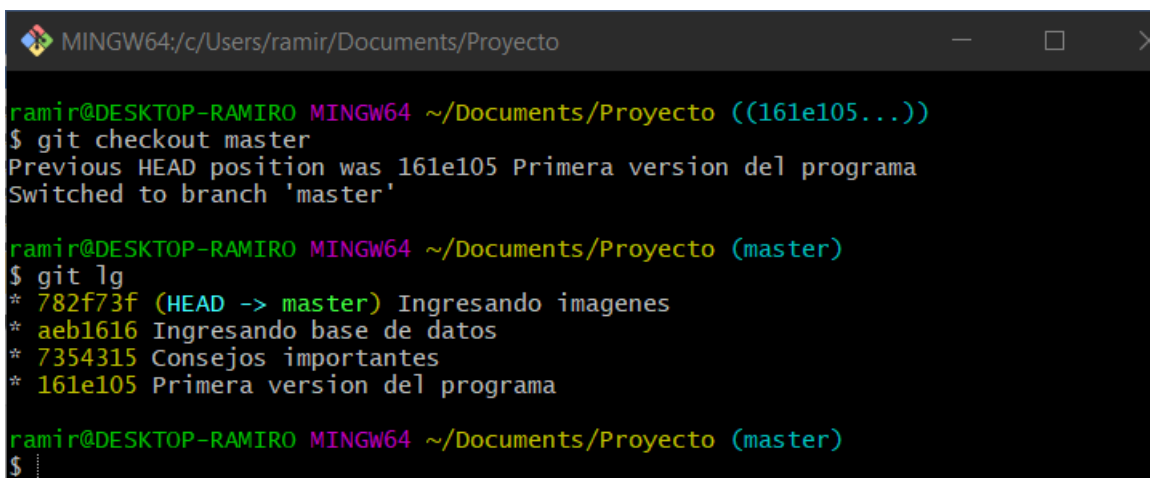
A terminal window titled 'MINGW64:/c/Users/ramir/Documents/Proyecto' showing the output of the 'git lg' command. The output lists four commits: 782f73f (master) for 'Ingresando imagenes', aeb1616 for 'Ingresando base de datos', 7354315 for 'Consejos importantes', and 161e105 (HEAD) for 'Primera version del programa'.

```
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$ git lg
* 782f73f (master) Ingresando imagenes
* aeb1616 Ingresando base de datos
* 7354315 Consejos importantes
* 161e105 (HEAD) Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$
```

Para volver al punto anterior podemos colocar uno de dos comandos.

- \$ git switch --
(deshace la rama sin nombre incluso si hiciste commit y vuelve a la rama master)
- \$ git checkout master
(Independientemente de donde se encuentre se dirige a seguir con la rama master desde el ultimo commit)
- \$ git checkout <nombre de cualquier otra rama>
(Igual que la segunda pero hacia una rama alternativa)

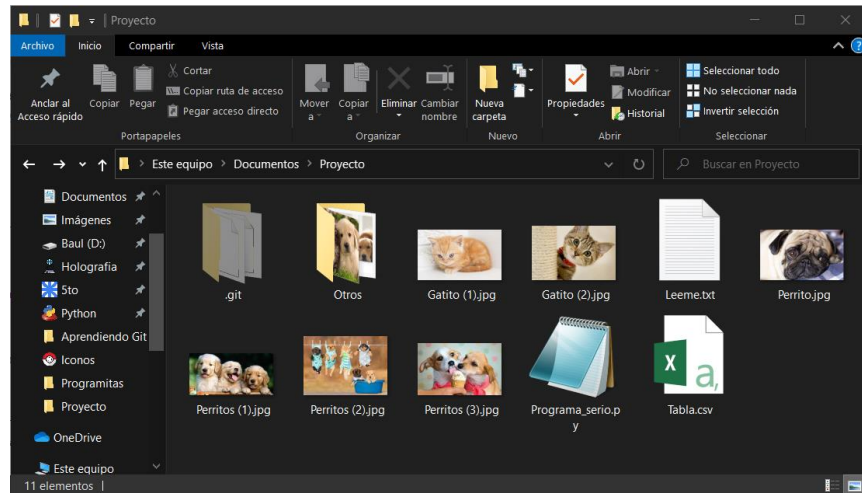
A terminal window titled 'MINGW64:/c/Users/ramir/Documents/Proyecto' showing the process of switching to the master branch. It first shows 'git checkout master' with a message about the previous HEAD position. Then it shows 'git lg' output, where the first commit is now marked as (HEAD -> master).

```
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((161e105...))
$ git checkout master
Previous HEAD position was 161e105 Primera version del programa
Switched to branch 'master'

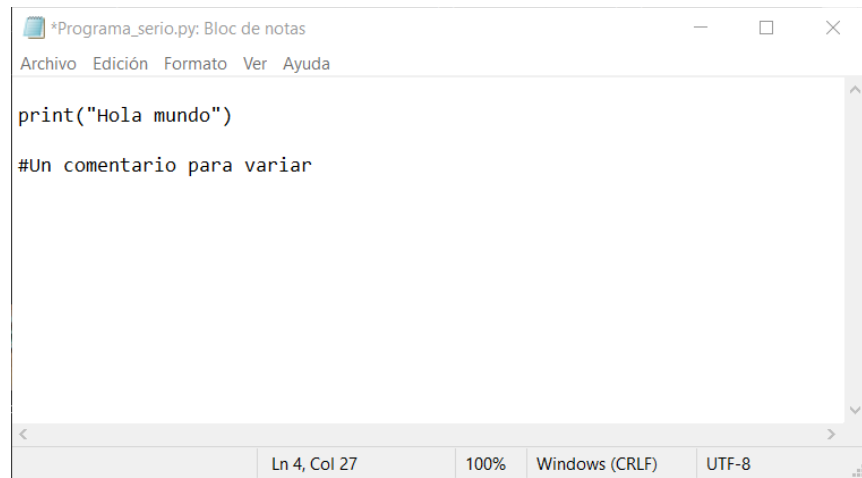
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git lg
* 782f73f (HEAD -> master) Ingresando imagenes
* aeb1616 Ingresando base de datos
* 7354315 Consejos importantes
* 161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Ahora que volvimos al ultimo commit volvemos a ver la carpeta del proyecto:

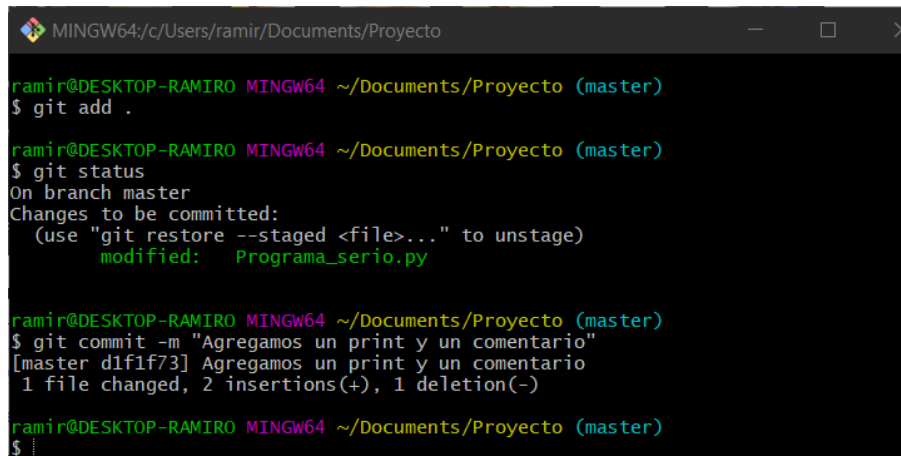


Elegimos un archivo cualquiera y lo modificamos:



Guardamos.

```
$ git add .  
$ git commit -m "Agregamos un print y un comentario"
```



```
MINGW64:/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
9f43a72 (HEAD -> master) Agregamos un print y un comentario
cdc1d9d Ingresando imagenes
aeb1616 Ingresando base de datos
7354315 Consejos importantes
161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Muchas veces nos sucede que realizamos un commit y después nos damos cuenta que no deberíamos haberlo hecho y necesitamos eliminarlo. ¿Como podemos eliminar el commit?

Solución si NO hemos subido el commit a nuestro repositorio remoto (no hemos realizado push, se explicará más tarde)

```
$ git reset --hard HEAD~1
```

--head: Con esta opción estamos indicando que retrocedemos a el commit HEAD~1 y perdemos todas las confirmaciones posteriores. HEAD~1 es un atajo para apuntar al commit anterior al que nos encontramos. CUIDADO, con la opción --head, ya que como he dicho se borran todos los commits posteriores al commit al que indicamos.

```
$ git reset --soft HEAD~1
```

--soft: con esta opción estamos indicando que retrocedemos a el commit HEAD~1 y no perdemos los cambios de los commits posteriores. Todos los cambios aparecerán como pendientes para realizar un commit.

```
MINGW64:/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git reset --soft HEAD~1

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
cdc1d9d (HEAD -> master) Ingresando imagenes
aeb1616 Ingresando base de datos
7354315 Consejos importantes
161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Comandos usados:

\$ git checkout <id commit>
Podemos usar checkout para visualizar el estado del proyecto en un commit concreto, es decir, en ese momento en el tiempo. Con esta operación lo que internamente estamos haciendo es apuntar a ese commit con el puntero HEAD.
\$ git checkout <nombre_rama>
Útil para volver a una rama generada previamente sin generar una rama previa.
\$ git switch -
Deshace la operación de checkout
\$ git checkout master
(Independientemente de donde se encuentre se dirige a seguir con la rama master desde el ultimo commit)
\$ git checkout <nombre de cualquier otra rama>
(Igual que la segunda pero hacia una rama alternativa)
\$ git reset --hard HEAD~1
--hard: Con esta opción estamos indicando que retrocedemos a el commit HEAD~1 y perdemos todas las confirmaciones posteriores. HEAD~1 es un atajo para apuntar al commit anterior al que nos encontramos. CUIDADO, con la opción -hard, ya que como he dicho se borran todos los commits posteriores al commit al que indicamos.
\$ git reset --soft HEAD~1
--soft: con esta opción estamos indicando que retrocedemos a el commit HEAD~1 y no perdemos los cambios de los commits posteriores. Todos los cambios aparecerán como pendientes para realizar un commit.

Crear, eliminar y unir una rama:

Para nuestro ejemplo vamos a hacer caso y generar nuestra nueva rama, existen varias formas:

Crear una rama:

Si queremos empezar una rama diferente a partir del punto en el que me encuentro pero seguir en la rama actual

```
$ git Branch <nombre_nueva_rama>
```

Si queremos empezar una rama diferente a partir del punto en el que me encuentro y ya comenzar a trabajar en esa nueva rama

```
$ git checkout -b <nombre_nueva_rama>
```

Si queremos empezar una rama diferente a partir de un id diferente.

```
$ git checkout <id commit>
```

```
$ git switch -c <Nombre_nueva_rama>  
(debe ser una sola palabra)
```

Eliminar una rama:

```
$ git branch -d <nombre_de_la_rama>
```

Unir ramas:

Suele pasar que muchas veces tenemos la rama principal funcionando y como no queremos romper lo que hemos creado pero necesitamos probar algunas variantes del programa que generamos una rama de desarrollo o una rama de pruebas. Tiempo después logramos con la rama prueba lo que estábamos buscando para mejorar la rama original. ¿Cómo coloco lo de la rama de desarrollo en la rama original?

Para eso debemos de ir a la rama original o a aquella que queremos cambiar y luego modificarla con el comando merge.

```
$ git checkout <rama_principal>
```

```
$ git merge <rama_secundaria>
```

Ejemplo:

```
$ git checkout master
```

```
$ git merge Desarrollo
```

Ahora la rama master genero un nuevo commit que es igual al ultimo commit de la rama Desarrollo. A partir de ahí la rama Desarrollo ya no existe porque es parte de la rama Master. Todo esto se puede visualizar con el comando.

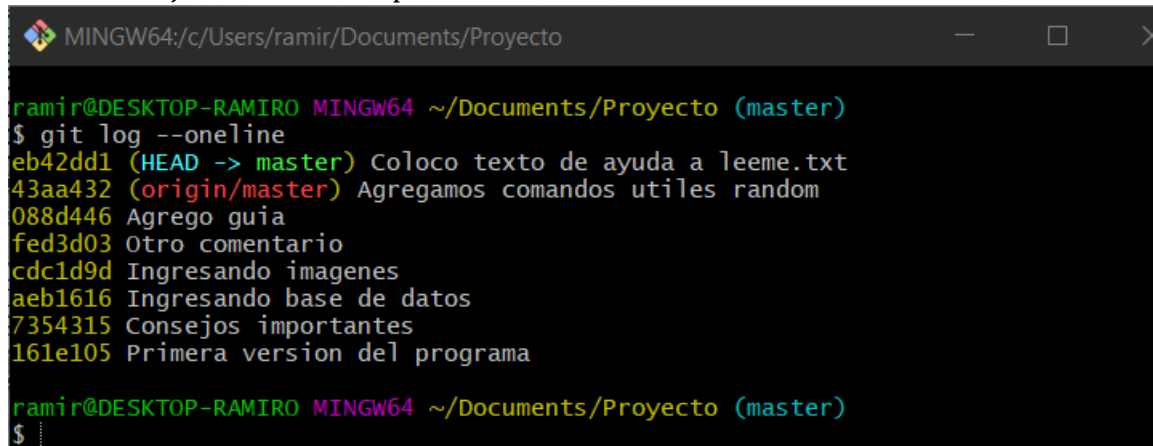
```
$ git log --oneline --decorate --graph
```

Conflicto típico:

Normalmente uno tiene dos o mas ramas que afectan al mismo archivo. Esto no es problema hasta que hacemos “Merge” Si queremos evitar conflictos el commit de la rama que queremos preservar debe de haber sido guardada ultima. Caso contrario se genera un conflicto donde no sabe cuales son los cambios que se desean guardar. Esto devuelve un archivo que muestra las diferentes modificaciones.

Ejemplo:

Vamos a viajar a un commit pasado



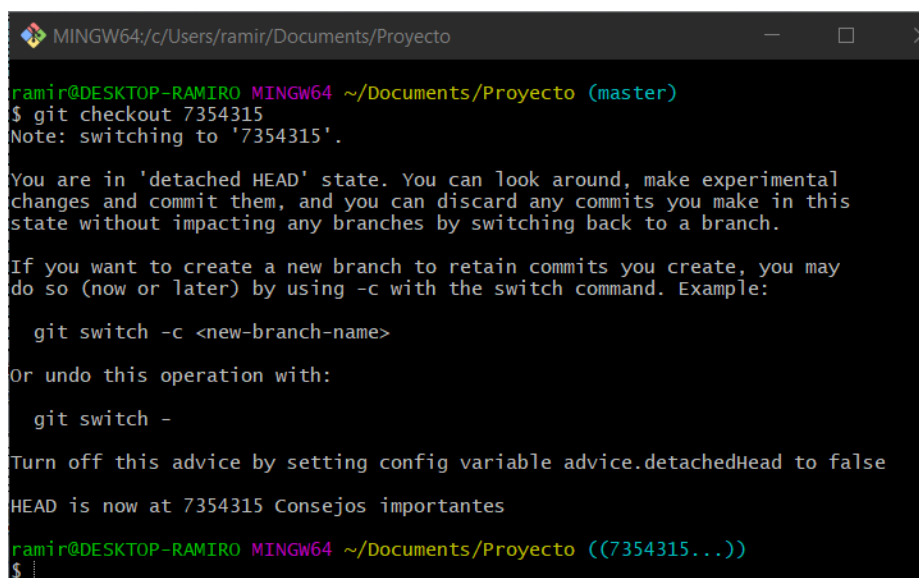
```
MINGW64:/c/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
eb42dd1 (HEAD -> master) Coloco texto de ayuda a leeme.txt
43aa432 (origin/master) Agregamos comandos utiles random
088d446 Agrego guia
fed3d03 Otro comentario
cdc1d9d Ingresando imagenes
aeb1616 Ingresando base de datos
7354315 Consejos importantes
161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

(Notaras que hice commits nuevos desde el ultimo ejemplo, fue para repasar algunos pasos, sin embargo, no afectan a este ejemplo asi que manos a la obra.)Elegiremos para este ejemplo el commit

“7354315 Consejos importantes”

```
$ git checkout 7354315
```



```
MINGW64:/c/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git checkout 7354315
Note: switching to '7354315'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

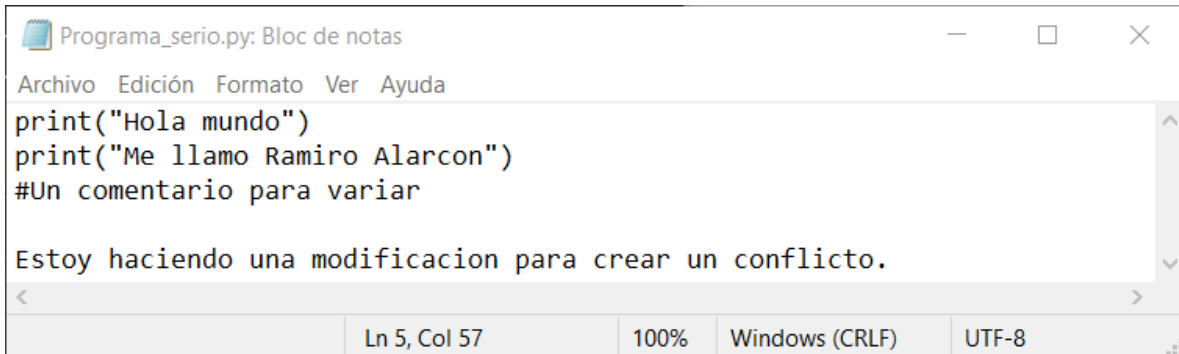
Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 7354315 Consejos importantes
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((7354315...))
$
```

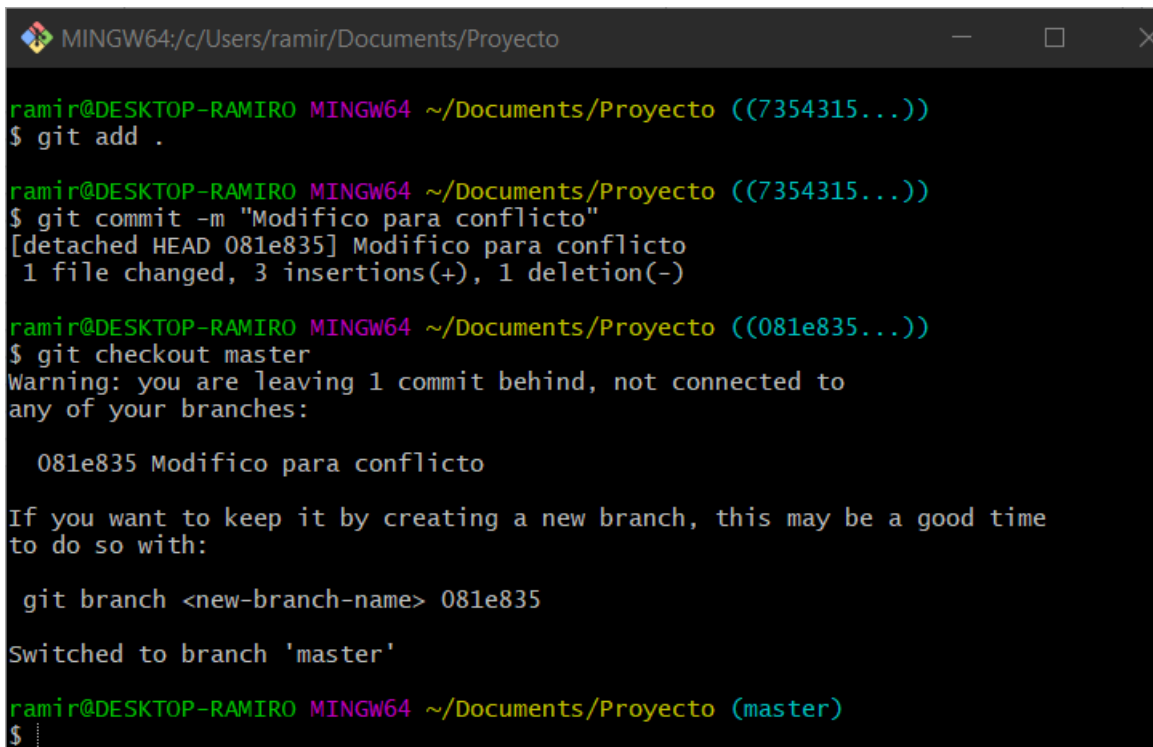
Si leímos el capítulo que trata sobre viajar a través de las ramas sabrás que ahora toda modificación se tomara como parte de una nueva rama. Procedemos con la modificación.



```
Programa_serio.py: Bloc de notas
Archivo Edición Formato Ver Ayuda
print("Hola mundo")
print("Me llamo Ramiro Alarcon")
#Un comentario para variar

Estoy haciendo una modificacion para crear un conflicto.
Ln 5, Col 57 100% Windows (CRLF) UTF-8
```

```
$ git add .
$ git commit -m "Modifico para conflicto"
$ git checkout master
```



```
MINGW64:/c/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((7354315...))
$ git add .

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((7354315...))
$ git commit -m "Modifico para conflicto"
[detached HEAD 081e835] Modifico para conflicto
1 file changed, 3 insertions(+), 1 deletion(-)

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto ((081e835...))
$ git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

    081e835 Modifico para conflicto

If you want to keep it by creating a new branch, this may be a good time
to do so with:

    git branch <new-branch-name> 081e835

Switched to branch 'master'

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Vemos que me ofrece darle un nombre a esa rama. Accedemos.

```
$ git branch Rama_conflicto 081e835
```

Haciendo git lg obtendremos esto:

```
MINGW64:/c:/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git lg
* 081e835 (Rama_conflicto) Modifico para conflicto
* eb42dd1 (HEAD -> master) Coloco texto de ayuda a leeme.txt
* 43aa432 (origin/master) Agregamos comandos utiles random
* 088d446 Agrego guia
* fed3d03 Otro comentario
* cdc1d9d Ingresando imagenes
* aeb1616 Ingresando base de datos
/
* 7354315 Consejos importantes
* 161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

git config --global alias.lg "git log --oneline --decorate --graph" = git lg
(resumir comandos)

Podemos ver la creación de la rama paralela.

Volvemos a master para crear una modificación

```
*Programa_serio.py: Bloc de notas
Archivo Edición Formato Ver Ayuda
print("Hola mundo")

#Un comentario para variar

Nueva modificacion (numero 2) para conflicto.
```

¿Que hicimos hasta ahora?

Creamos una rama para generar un cambio, luego volvimos a la rama original y generamos otro cambio. Cuando tratemos de hacer merge para que los cambios de la rama secundaria se sumen a la rama principal el programa notara que los cambios de la rama principal son mas recientes que los de la rama secundaria, eso creara un conflicto.

```
MINGW64:/c:/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git add .

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git commit -m "Ultima modificacion para conflicto"
[master a6047b3] Ultima modificacion para conflicto
2 files changed, 3 insertions(+), 1 deletion(-)
delete mode 100644 ~$a de git para mancos.docx

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Obtendremos algo como esto:

```
MINGW64/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git lg
* a6047b3 (HEAD -> master) Ultima modificacion para conflicto
* eb42dd1 Coloco texto de ayuda a leeme.txt
* 43aa432 (origin/master) Agregamos comandos utiles random
* 088d446 Agrego guia
* fed3d03 Otro comentario
* cdc1d9d Ingresando imagenes
* aeb1616 Ingresando base de datos
| * 081e835 (Rama_conflicto) Modifico para conflicto
...skipping...
* a6047b3 (HEAD -> master) Ultima modificacion para conflicto
* eb42dd1 Coloco texto de ayuda a leeme.txt
* 43aa432 (origin/master) Agregamos comandos utiles random
* 088d446 Agrego guia
* fed3d03 Otro comentario
* cdc1d9d Ingresando imagenes
* aeb1616 Ingresando base de datos
| * 081e835 (Rama_conflicto) Modifico para conflicto
|/
* 7354315 Consejos importantes
* 161e105 Primera version del programa
~
~
~
~

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Como ya estamos en la rama principal hacemos.

```
$ git merge Rama_conflicto
```

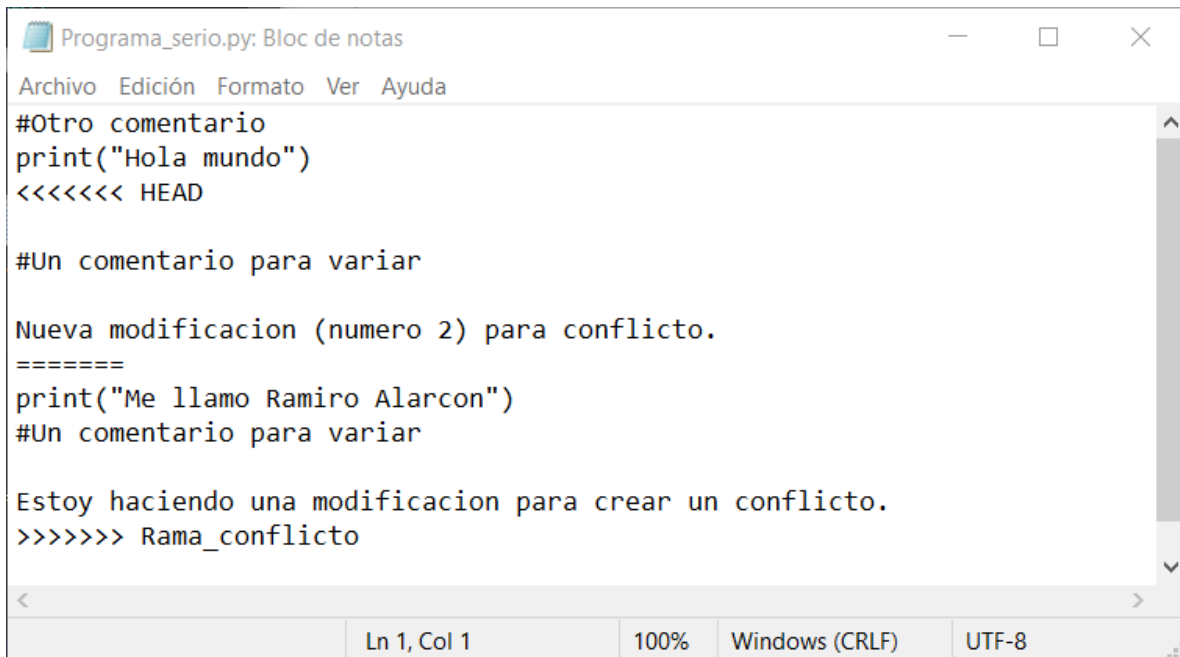
Finalmente obtenemos esto

```
MINGW64/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git merge Rama_conflicto
Auto-merging Programa_serio.py
CONFLICT (content): Merge conflict in Programa_serio.py
Automatic merge failed; fix conflicts and then commit the result.

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master|MERGING)
$
```

Y al revisar el archivo en cuestión obtenemos esto:



```
Programa_serio.py: Bloc de notas
Archivo Edición Formato Ver Ayuda
#Otro comentario
print("Hola mundo")
<<<<<<< HEAD

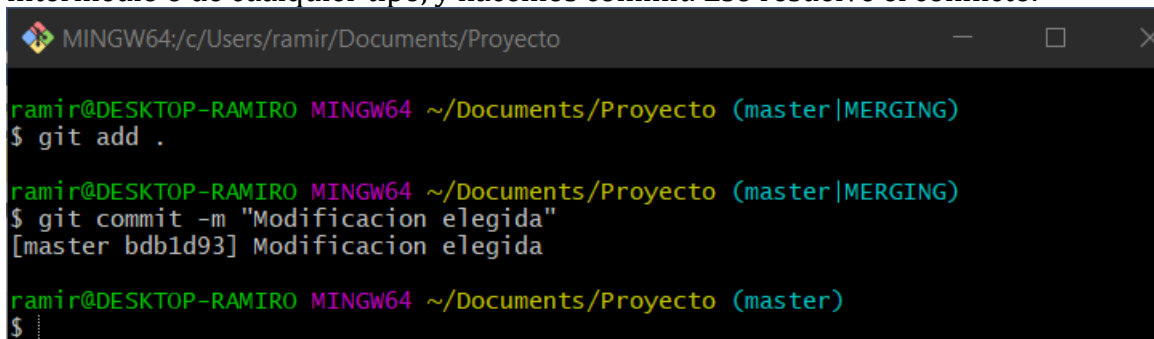
#Un comentario para variar

Nueva modificacion (numero 2) para conflicto.
=====
print("Me llamo Ramiro Alarcon")
#Un comentario para variar

Estoy haciendo una modificacion para crear un conflicto.
>>>>>>> Rama_conflicto

Ln 1, Col 1    100%    Windows (CRLF)    UTF-8
```

Por suerte el conflicto es más simple de resolver.
Solo debemos elegir cual es la modificación que queremos incluso puede ser algo intermedio o de cualquier tipo, y hacemos commit. Eso resuelve el conflicto.



```
MINGW64:/c/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master|MERGING)
$ git add .

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master|MERGING)
$ git commit -m "Modificacion elegida"
[master bdb1d93] Modificacion elegida

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

```
MINGW64:/c:/Users/ramir/Documents/Proyecto
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git lg
* bdb1d93 (HEAD -> master) Modificacion elegida
| \
| * 081e835 (Rama_conflicto) Modifico para conflicto
* | a6047b3 Ultima modificacion para conflicto
* | eb42dd1 Coloco texto de ayuda a leeme.txt
* | 43aa432 (origin/master) Agregamos comandos utiles random
* | 088d446 Agrego guia
* | fed3d03 Otro comentario
* | cdc1d9d Ingresando imagenes
* | aeb1616 Ingresando base de datos
| /
* 7354315 Consejos importantes
* 161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Conflicto solucionado.

Comandos usados

\$ git Branch <nombre_nueva_rama>
Empezar una rama diferente a partir del punto en el que me encuentro pero seguir en la rama actual
\$ git checkout -b <nombre_nueva_rama>
Empezar una rama diferente a partir del punto en el que me encuentro y ya comenzar a trabajar en esa nueva rama
\$ git checkout <id commit>
\$ git switch -c <Nombre_nueva_rama>
Empezar una rama diferente a partir de un id diferente.
\$ git branch -d <nombre_de_la_rama>
Eliminar una rama
\$ git checkout <rama_principal>
\$ git merge <rama_secundaria>
Unir ramas

Comandos útiles:

Ver el historial de comandos:

```
$ git reflog
```

(equivalente a `git reflog show HEAD`): ¿Qué pasa si queremos rehacer un cambio que hemos revertido con `git reset`? ¿Hemos perdido la posición de HEAD para siempre? La respuesta es NO. el comando `reflog` guarda un histórico de los últimos valores que ha ido tomando el puntero HEAD de la rama, y podemos rehacer un reset hacia atrás simplemente haciendo otro reset hacia el commit posterior que consultemos en el `reflog`. OJO!: `reflog` no guarda infinitos estados: hasta no tener 7000 objetos fuera sin empaquetar o más de 50 ficheros empaquetadores no se lanza el recolector de basura (`git gc`, que puede ser lanzado a propósito).

NOTA: Para hacer referencia a un objeto de `reflog` podemos usar su id o también la sintaxis `HEAD@{1}` indicando la posición en la pila de cambios, siendo 0 el estado actual.

El `reflog` no se salva al hacer un push, ya que contiene un log diferente en cada usuario local, por lo que no tendría sentido subirlo al repo. Sin embargo el `reflog` se almacena en el directorio `.git/logs`, que podemos salvar si lo creemos oportuno. Al fin y al cabo todos los objetos, aunque empaquetados, sí están respaldados en el remoto.

Información sobre comando:

```
$ git help <comando>
```

Comparar el ultimo commit con el directorio de trabajo:

```
$ git diff
```

(`git diff`: Permite ver las diferencias en el código entre el `working directory` y el `Index`.)

Comparar commit actual con commit anterior:

Para más información:

<https://git-scm.com/docs/git-show>

```
$ git show <opciones> <objeto>
```

- Muestra uno o más objetos (blobs, árboles, etiquetas y confirmaciones).

- Para confirmaciones, muestra el mensaje de registro y la diferencia textual. También presenta la confirmación de fusión en un formato especial producido por **git diff-tree --cc**.
- Para las etiquetas, muestra el mensaje de la etiqueta y los objetos referenciados.
- Para los árboles, muestra los nombres (equivalente a **git ls-tree** con **--name-only**).
- Para blobs simples, muestra el contenido simple.
- El comando toma opciones aplicables al comando **git diff-tree** para controlar cómo se muestran los cambios que introduce la confirmación.

Borrar archivo a través de ventana de comandos:

```
$ git rm <Nombre_archivo>
```

```
$ git commit -m "describimos que se borro y porque"
```

(Es igual que add, pero para cuando hemos borrado un fichero. Es decir, envía un fichero borrado al Index.)

Etiquetas (tag)

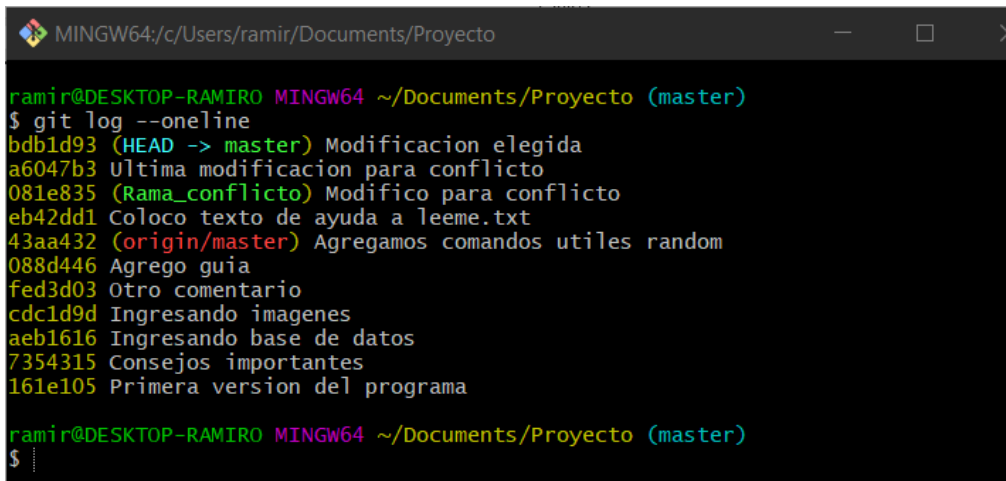
Para más información puede referenciarse a esta página:

<https://git-scm.com/book/en/v2/Git-Basics-Tagging>

Es un nombre con un puntero a un momento concreto en la historia del desarrollo. Se suelen usar para identificar los momentos en que se lanzan nuevas versiones.

Muchas veces se guardan demasiados commit con variaciones o correcciones muy pequeñas, es bueno tener una herramienta que nos ayuden a visualizar los saltos entre grandes versiones de un mismo programa.

La mejor forma de explicarlo es con un ejemplo:

A screenshot of a terminal window titled 'MINGW64: c:/Users/ramir/Documents/Proyecto'. The terminal shows the output of the command 'git log --oneline'. The output lists 11 commits with their hashes and descriptions. The current branch is 'master'.

```
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
bdb1d93 (HEAD -> master) Modificacion elegida
a6047b3 Ultima modificacion para conflicto
081e835 (Rama_conflicto) Modifico para conflicto
eb42dd1 Coloco texto de ayuda a leeme.txt
43aa432 (origin/master) Agregamos comandos utiles random
088d446 Agrego guia
fed3d03 Otro comentario
cdc1d9d Ingresando imagenes
aeb1616 Ingresando base de datos
7354315 Consejos importantes
161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Tomando como referencia esta lista de commits

Etiquetando un commit antiguo:

Para este caso en particular vamos a elegir el segundo commit “Consejos importantes”

```
$ git tag -a <versión> <commit id> -m “descripcion”
```

Para nuestro ejemplo:

```
$ git tag -a V0.0.1 7354315 -m “Version inicial sin imagenes”
```

Tambien creamos una version intermedia:

```
$ git tag -a V0.2.0 43aa432 -m “Detalles importantes”
```

```
MINGW64/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git tag -a v0.2.0 43aa432 -m "Detalles importantes"

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
bdb1d93 (HEAD -> master) Modificacion elegida
a6047b3 Ultima modificacion para conflicto
081e835 (rama_conflicto) Modifico para conflicto
eb42dd1 Coloco texto de ayuda a leeme.txt
43aa432 (tag: v0.2.0, origin/master) Agregamos comandos utiles random
088d446 Agrego guia
fed3d03 Otro comentario
cdc1d9d Ingresando imagenes
aeb1616 Ingresando base de datos
7354315 (tag: v0.0.1) Consejos importantes
161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Etiquetar el HEAD

Para crear una etiqueta en el commit donde ya nos encontramos solo debemos hacer:

```
$ git tag -a <versión> -m "descripción"
```

Para nuestro ejemplo:

```
$ git tag -a 2.0.0 -m "Version final para usuario"
```

```
MINGW64/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git log --oneline
bdb1d93 (HEAD -> master, tag: 2.0.0) Modificacion elegida
a6047b3 Ultima modificacion para conflicto
081e835 (rama_conflicto) Modifico para conflicto
eb42dd1 Coloco texto de ayuda a leeme.txt
43aa432 (tag: v0.2.0, origin/master) Agregamos comandos utiles random
088d446 Agrego guia
fed3d03 Otro comentario
cdc1d9d Ingresando imagenes
aeb1616 Ingresando base de datos
7354315 (tag: v0.0.1) Consejos importantes
161e105 Primera version del programa

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Ver todas las etiquetas:

Para eso utilizamos el comando:

```
$ git tag
```

```
MINGW64/c/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git tag
2.0.0
v0.0.1
v0.2.0

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Podemos ver que equivoque la versión a propósito, por ende la vamos a eliminar y volver a escribir.

[Eliminar una etiqueta:](#)

Para eliminar una etiqueta hacemos uso del comando:

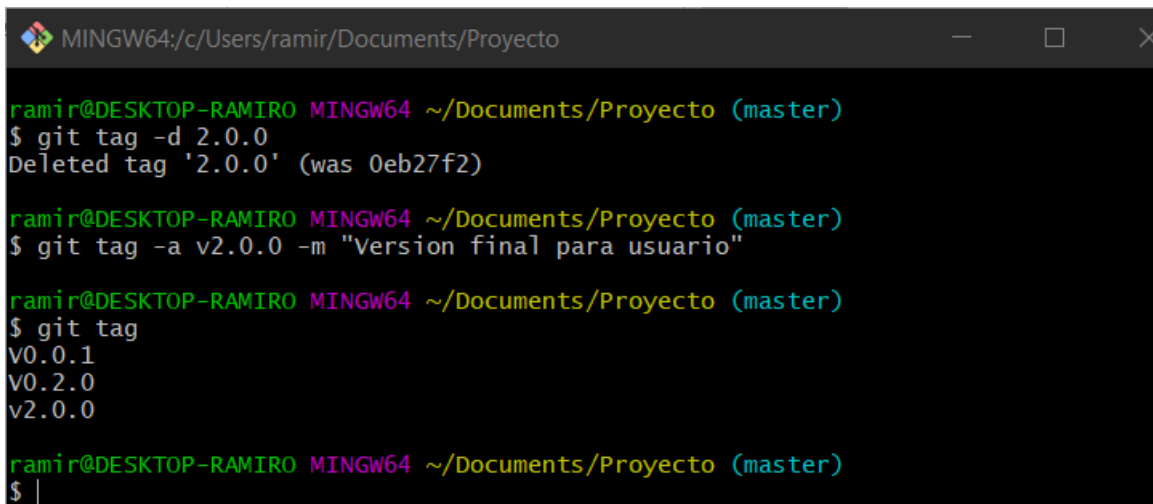
```
$ git tag -d <versión>
```

Para nuestro ejemplo:

```
$ git tag -d 2.0.0
```

Para luego reescribir ahora sin error:

```
$ git tag -a V2.0.0 -m "Version final para usuario"
```

A screenshot of a terminal window titled 'MINGW64:/c/Users/ramir/Documents/Proyecto'. The terminal shows the following sequence of commands and output:

```
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git tag -d 2.0.0
Deleted tag '2.0.0' (was 0eb27f2)

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git tag -a v2.0.0 -m "Version final para usuario"

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git tag
v0.0.1
v0.2.0
v2.0.0

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ |
```

El uso de la “v” antes de la versión es una elección personal, no esta ligado al programa ni a la forma de trabajar de git.

[Ver contenido de etiqueta.](#)

Para ver las características del commit asociado a una etiqueta utilizamos

```
$ git show <versión>
```



```
MINGW64:/c:/Users/ramir/Documents/Proyecto

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$ git show v2.0.0
tag v2.0.0
Tagger: Ramiro Alarcón Lasagno <ing.ramiro.a.l@gmail.com>
Date: Tue Nov 24 19:50:43 2020 -0300

Version final para usuario

commit bdb1d93b1081ed6bcaee2704a572a1006cb36a9d (HEAD -> master, tag: v2.0.0)
Merge: a6047b3 081e835
Author: Ramiro Alarcón Lasagno <ing.ramiro.a.l@gmail.com>
Date: Sat Nov 21 18:25:11 2020 -0300

    Modificacion elegida

diff --cc Programa_serio.py
index 34094e3,d5a9c90..1f51c61
--- a/Programa_serio.py
+++ b/Programa_serio.py
@@@ -1,6 -1,5 +1,5 @@@
+ #Otro comentario
+ print("Hola mundo")
- print("Me llamo Ramiro Alarcon")
+
+ #Un comentario para variar

- Nueva modificacion (numero 2) para conflicto.
- Estoy haciendo una modificacion para crear un conflicto.

ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Proyecto (master)
$
```

Repositorio remoto (git hub)

Generar una cuenta en GitHub, especialmente si ya tenés experiencia usando git puede ser altamente benéfico.

Te permite tener un repositorio remoto capaz de guardar todos tus proyectos con sus respectivos commit en la nube de forma publica o privada para compartir o para vender, y en el primer caso para aprender de los repositorios compartidos por otras personas.

Crear una cuenta en github:

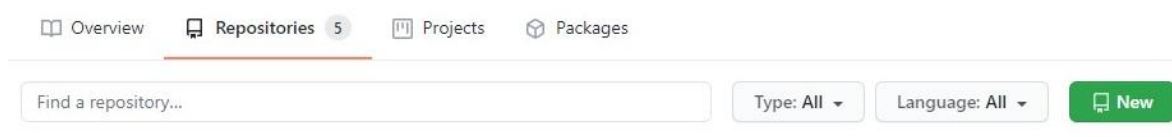
<https://github.com/>

Crear un repositorio:

Acto seguido con la cuenta abierta vamos a ir a la barra superior y dirigirnos a "Repositorios"



Seleccionamos "New"




- Nos pedirá un Nombre para el repositorio que será cualquiera que elijamos. Sin embargo, es de buenas practicas elegir un nombre que tenga que ver con el proyecto.
- Opcionalmente una breve descripción del proyecto.
- Nos dará la opción de hacerlo de acceso publico o privado
- Nos dará tres opciones para agregar al repositorio:
 - escribir una descripción larga de su proyecto
 - Elegir qué archivos no rastrear de una lista de plantillas (.gitignore)
 - Elegir una licencia que le dice a otros lo que pueden y no pueden hacer con su código


Owner * Repository name *

/

Great repository names are short and memorable. Need inspiration? How about **animated-robot**?

Description (optional)

☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☐ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)

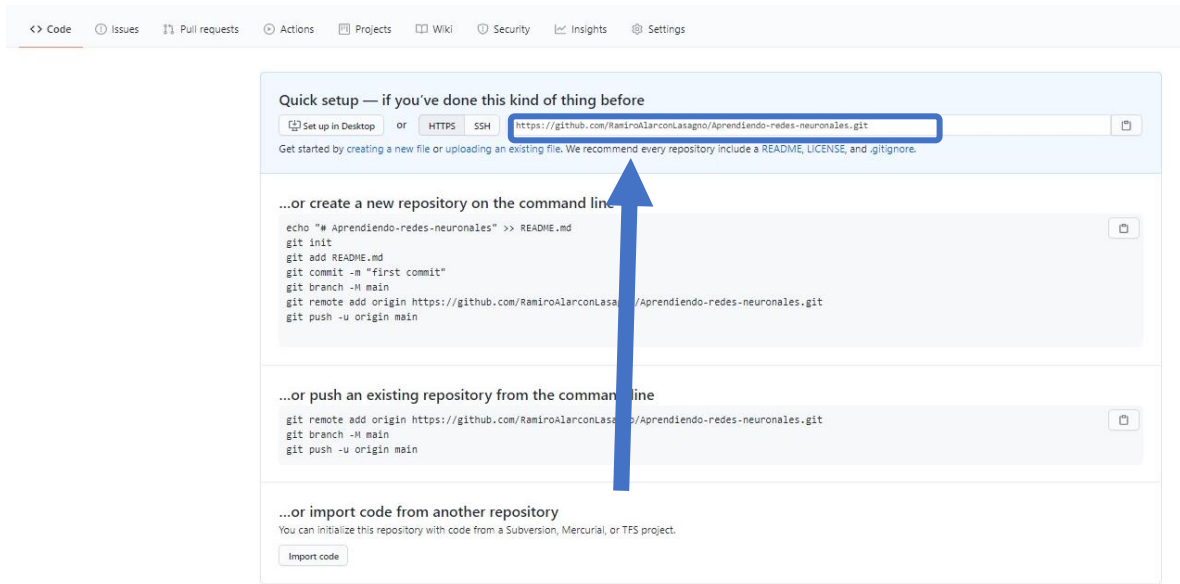
☐ **Add .gitignore**
Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**
A license tells others what they can and can't do with your code. [Learn more.](#)

[Create repository](#)

Y así creamos un repositorio

Esto nos va a devolver una pagina con un link que contiene la dirección de nuestro repositorio.



A partir de aquí volvemos a nuestra ventana de comandos situada en la carpeta de nuestro proyecto.

Cargar proyecto en repositorio:

Asociamos nuestro repositorio local de (.git) a nuestro repositorio remoto así:

```
$ git remote add origin <pegamos la dirección de repositorio remoto>  
$ git remote add origin https://github.com/...
```

Ahora que nuestro repositorio está asociado al repositorio remoto solo debemos de guardar los cambios en el repositorio remoto usando

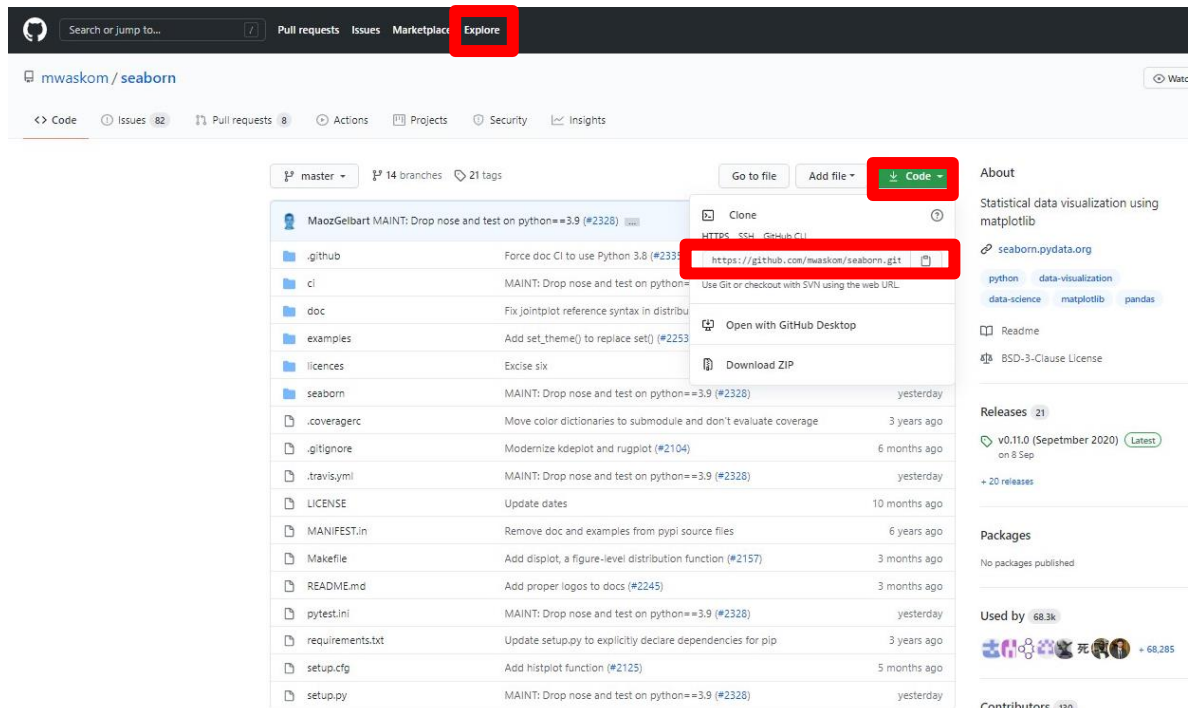
```
$ git push origin master
```

Y listo, nuestro repositorio tal como está en nuestra carpeta se encontrará en el repositorio remoto de github.

Clonar repositorio de otra persona:

Ya en github podemos explorar los repositorios públicos de otras personas desde la pestaña “Explore”

Ya habiendo encontrado un proyecto interesante nos dirigimos a code y tenemos dos opciones. Podemos descargar el proyecto en forma de .zip o clonar el repositorio.



Para clonar el repositorio copiamos el link apretando la carpeta al lado del link

Abrimos una carpeta nueva en nuestra computadora. Abrimos la ventana de comandos y escribimos:

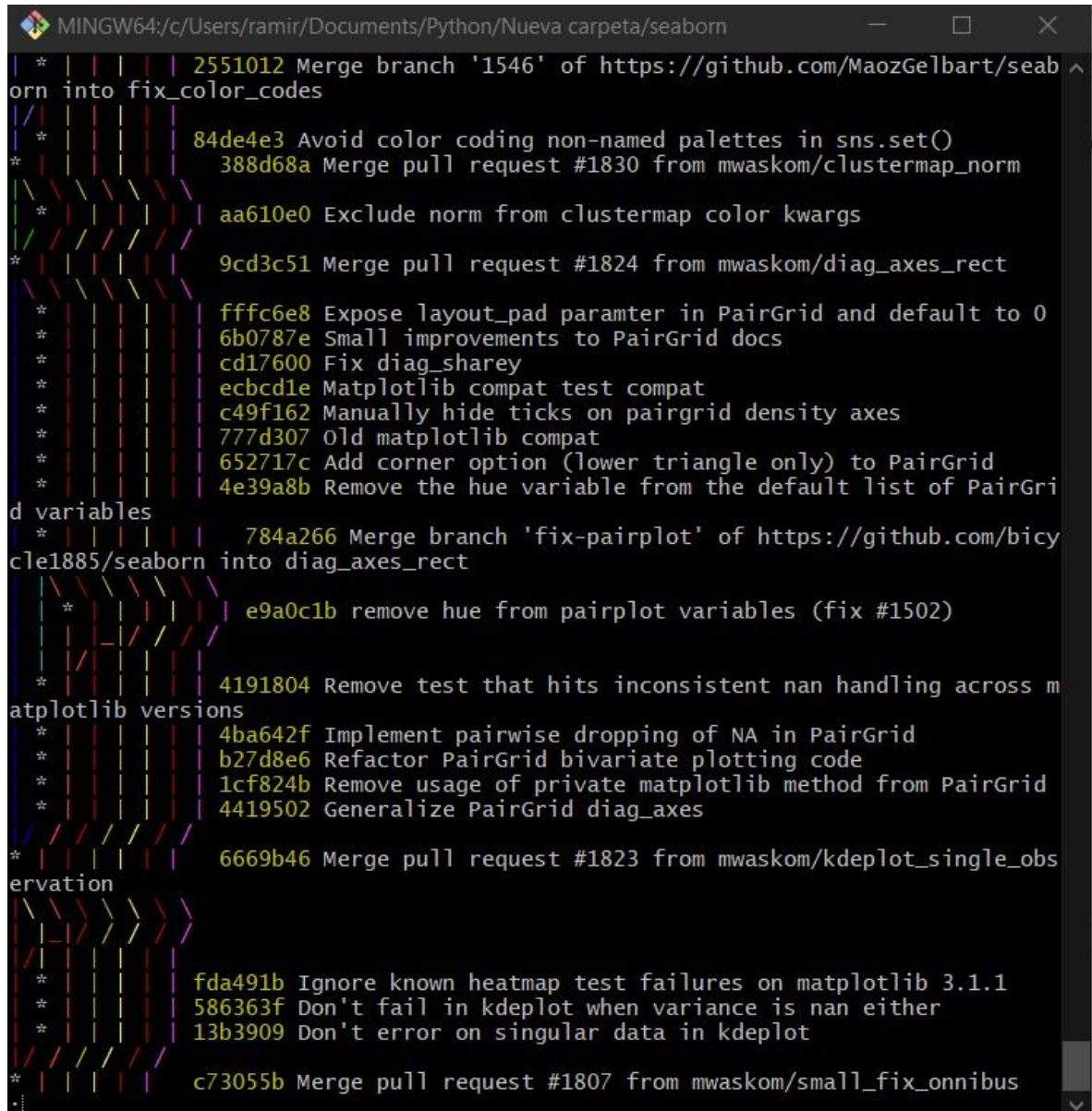
```
$ git clone <dirección>
```

En este caso:

```
$ git clone https://github.com/mwaskom/seaborn.git
```

```
MINGW64/c/Users/ramir/Documents/Python/Nueva carpeta
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Python/Nueva carpeta
$ git clone https://github.com/mwaskom/seaborn.git
Cloning into 'seaborn'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 13414 (delta 3), reused 0 (delta 0), pack-reused 13398
Receiving objects: 100% (13414/13414), 50.38 MiB | 4.68 MiB/s, done.
Resolving deltas: 100% (9598/9598), done.
ramir@DESKTOP-RAMIRO MINGW64 ~/Documents/Python/Nueva carpeta
$
```

Se creará una carpeta con el contenido del proyecto. Si abrimos la ventana de comandos dentro de esta carpeta y escribimos git lg:



```
MINGW64/c/Users/ramir/Documents/Python/Nueva carpeta/seaborn
* | | | | 2551012 Merge branch '1546' of https://github.com/MaozGelbart/seaborn into fix_color_codes
|/
* | | | | 84de4e3 Avoid color coding non-named palettes in sns.set()
* | | | | 388d68a Merge pull request #1830 from mwaskom/clustermap_norm
|/
* | | | | aa610e0 Exclude norm from clustermap color kwargs
|/
* | | | | 9cd3c51 Merge pull request #1824 from mwaskom/diag_axes_rect
|/
* | | | | fffc6e8 Expose layout_pad paramter in PairGrid and default to 0
* | | | | 6b0787e Small improvements to PairGrid docs
* | | | | cd17600 Fix diag_sharey
* | | | | ecbcd1e Matplotlib compat test compat
* | | | | c49f162 Manually hide ticks on pairgrid density axes
* | | | | 777d307 Old matplotlib compat
* | | | | 652717c Add corner option (lower triangle only) to PairGrid
* | | | | 4e39a8b Remove the hue variable from the default list of PairGrid variables
|/
* | | | | 784a266 Merge branch 'fix-pairplot' of https://github.com/bicycle1885/seaborn into diag_axes_rect
|/
* | | | | e9a0c1b remove hue from pairplot variables (fix #1502)
|/
* | | | | 4191804 Remove test that hits inconsistent nan handling across matplotlib versions
|/
* | | | | 4ba642f Implement pairwise dropping of NA in PairGrid
* | | | | b27d8e6 Refactor PairGrid bivariate plotting code
* | | | | 1cf824b Remove usage of private matplotlib method from PairGrid
* | | | | 4419502 Generalize PairGrid diag_axes
|/
* | | | | 6669b46 Merge pull request #1823 from mwaskom/kdeplot_single_observation
|/
|/
* | | | | fda491b Ignore known heatmap test failures on matplotlib 3.1.1
* | | | | 586363f Don't fail in kdeplot when variance is nan either
* | | | | 13b3909 Don't error on singular data in kdeplot
|/
* | | | | c73055b Merge pull request #1807 from mwaskom/small_fix_onnibus
```

Podremos ver la complejidad del proyecto e incluso dirigirnos a partes previas para hacer un estudio del desarrollo. O copiar código.

Fin

Esto es todo lo que contiene la guía pero no todo lo que hay que aprender sobre el tema. Sin embargo, este conocimiento es más que suficiente para manejarse de forma completa a través de git y github.

Espero te sirva.

Comandos usados en toda la guía:

\$ git init
Crea un repositorio local vacío en el directorio actual. Este se almacena en el directorio “.git”.
\$ git status
Este es el más usado de todos. Nos dice la rama en la que nos encontramos y los ficheros que contiene el stage junto a su estado (new/modified/deleted). Además, avisa de ficheros sin seguimiento por git (untracked) o ficheros con conflictos.
\$ git add <fichero>
Manda uno o varios ficheros separados por espacios o un directorio al staging area. Recién añadido un nuevo fichero al repositorio, este se encuentra en estado “untracked” y debemos hacer un primer add.
\$ git add .
Manda todos los ficheros del directorio al staging area.
\$ git add -A
Manda todos los ficheros del directorio al staging area.
\$ git add -all
Manda todos los ficheros del directorio al staging área.
\$ git add *.<formato>
*.doc; *.txt; *.py; etc...
\$ git add <NombreCarpeta>/
Manda todos los ficheros de una carpeta del directorio principal al staging area.
\$ git reset
Con estos comandos podemos sacar un fichero de la zona Index.
\$ git reset *<formato>
Variación para formatos
\$ git reset <NombreCarpeta>/
Variación para carpetas.
\$ git commit -m "descripción breve"
Registra definitivamente los cambios que previamente estaban en el Stage. Un commit contiene
\$ git commit -am "descripción"
Hace un commit con todos los ficheros con cambios. Por tanto, esta opción a es para realizar el add previo y el commit de una sola vez. *Importante: solo funciona si el fichero ya está en seguimiento por git (tracked). (Solo para modificaciones)
\$ git commit --amend
Se modifica el último commit con el contenido que tengamos en el Index, muy útil cuando se nos ha olvidado añadir algo en el commit o nos hemos confundido en la selección de cambios para incluir en el commit. *Realmente borra el commit y crea otro de nuevo. (Solo recomendable en Linux)

\$ git commit --amend -m "corrección del mensaje"
Cuando nos hayamos equivocado en el mensaje del commit.
\$ git log
Abre un listado que nos va a permitir ver el histórico completo de todos los commit que se han ido realizando en el repositorio a partir de HEAD hacia atrás, incluyendo los que nos traemos del remoto cuando hacemos un fetch. Funciona en modo interactivo y por defecto los últimos cambios aparecen los primeros.
\$ git log --oneline
Nos permite ver un commit por linea, mostrando solo el id y el mensaje.
\$ git log --oneline --decorate --graph
Con la opción graph podemos visualizar de forma “gráfica” las ramas y sus merges en el tiempo.
\$ git config --global alias.<apodo> “comando largo”
Resumir comando (pag 27)
\$ git config --global -l -e
Leer o escribir información del usuario (pag 27)
\$ git checkout <id commit>
Podemos usar checkout para visualizar el estado del proyecto en un commit concreto, es decir, en ese momento en el tiempo. Con esta operación lo que internamente estamos haciendo es apuntar a ese commit con el puntero HEAD.
\$ git checkout <nombre_rama>
Útil para volver a una rama generada previamente sin generar una rama previa.
\$ git switch -
Deshace la operación de checkout
\$ git checkout master
(Independientemente de donde se encuentre se dirige a seguir con la rama master desde el ultimo commit)
\$ git checkout <nombre de cualquier otra rama>
(Igual que la segunda pero hacia una rama alternativa)
\$ git reset --hard HEAD~1
--head: Con esta opción estamos indicando que retrocedemos a el commit HEAD~1 y perdemos todas las confirmaciones posteriores. HEAD~1 es un atajo para apuntar al commit anterior al que nos encontramos. CUIDADO, con la opción --head, ya que como he dicho se borran todos los commits posteriores al commit al que indicamos.
\$ git reset --soft HEAD~1
--soft: con esta opción estamos indicando que retrocedemos a el commit HEAD~1 y no perdemos los cambios de los commits posteriores. Todos los cambios aparecerán como pendientes para realizar un commit.
\$ git Branch <nombre_nueva_rama>
Empezar una rama diferente a partir del punto en el que me encuentro pero seguir en la rama actual

\$ git checkout -b <nombre_nueva_rama>
Empezar una rama diferente a partir del punto en el que me encuentro y ya comenzar a trabajar en esa nueva rama
\$ git checkout <id commit>
\$ git switch -c <Nombre_nueva_rama>
Empezar una rama diferente a partir de un id diferente.
\$ git branch -d <nombre_de_la_rama>
Eliminar una rama
\$ git checkout <rama_principal>
\$ git merge <rama_secundaria>
Unir ramas
\$ git reflog
Historial de comandos
\$ git help <comando>
Info de comandos
\$ git diff
Diferencias en el código entre el working directory y el Index
\$ git show <opciones> <objeto>
(Pag 44)
\$ git rm <Nombre_archivo>
\$ git commit -m "describimos que se borro y porque"
Borrar archivo por comando
\$ git tag -a <versión> <commit id> -m "descripcion"
Etiquetar commit antiguo
\$ git tag -a <versión> -m "descripción"
Etiquetar ultimo commit
\$ git tag
Ver etiquetas
\$ git tag -d <versión>
Eliminar etiqueta
\$ git remote add origin <pegamos la dirección de repositorio remoto>
Asociar repositorio local con repositorio remoto vacío.
\$ git push origin master
Cargar cambios del repositorio local al repositorio remoto
\$ git clone <dirección>
Clonar repositorio ajeno.