

```
1
2 Programming 'Language' = {
3
4     Introducción = [Python,
5                     Micropython]
6
7
8
9
10
11 }
12
13
14
```



```
1
2
3 Clase Extra 02 = {
4
5     Presentación = [Les
6                     damos la bienvenida al
7                     curso]
8
9
10
11
12 }
13
14
```



```
1 Clases = {  
2
```

```
3  
4  
5 Clase  
6 Extra 01
```

Bibliotecas Estándar y Externas. Introducción a las bibliotecas estándar de Python. Uso de bibliotecas populares (NumPy, matplotlib).



```
11  
12  
13  
14
```

**Clase
Extra 02**

Trabajando en equipo. Git y Github.



¿Qué es git? {

Git es un sistema de control de versiones distribuido, diseñado para manejar todo tipo de proyectos con velocidad y eficiencia. Fue creado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux. Git permite a múltiples desarrolladores trabajar en un proyecto de manera simultánea sin interferir en el trabajo de los demás.

Está optimizado para guardar cambios de forma incremental. Permite contar con un historial, regresar a una versión anterior y agregar funcionalidades. Además, es capaz de llevar un registro de los cambios que otras personas realicen en los archivos.

Para tener estas ventajas, es necesario tener un servidor de control de versiones propio o usar uno público. Nosotros vamos a usar **GitHub**, que es un servidor de control de versiones público, donde podemos almacenar y compartir nuestros proyectos de forma gratuita.

}



¿Qué es git? {

Con Git se obtiene una mayor eficiencia usando archivos de texto plano, ya que con archivos binarios no puede guardar solo los cambios, sino que debe volver a grabar el archivo completo ante cada modificación, por mínima que sea, lo que hace que incremente el tamaño del repositorio.

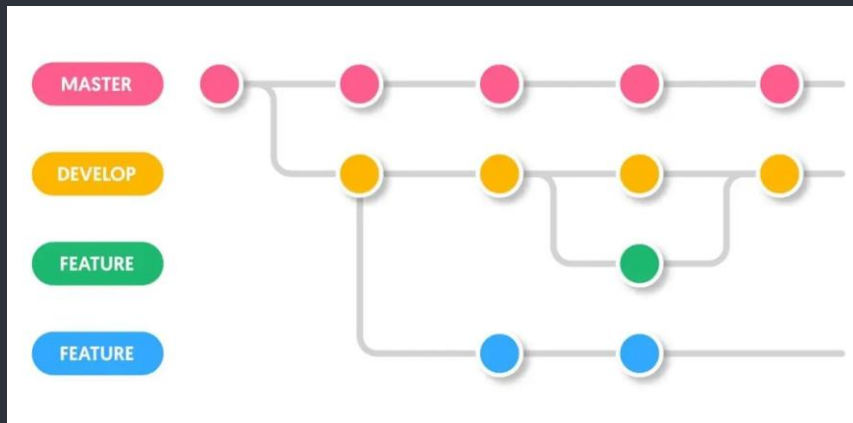
Gracias a git, podremos organizar el código, el historial y su evolución. Funciona como una máquina del tiempo que permite navegar a diferentes versiones del proyecto y si queremos agregar una funcionalidad nueva nos permite crear una rama (branch) para dejar intacta la versión estable y crear un ambiente de trabajo en el cual podemos trabajar en una nueva funcionalidad sin afectar la versión original. Por ello decimos que git nos permite:

- # Manejar distintas versiones o ramas del proyecto.
- # Guardar el historial o guardar todas las versiones de los archivos del proyecto.
- # Trabajar simultáneamente sobre un mismo proyecto.



¿Cómo funciona? {

Git almacena instantáneas de un mini sistema de archivos. Cada vez que confirmamos un cambio, Git toma una "foto" del aspecto del proyecto en ese momento y crea una referencia a esa instantánea. Si un archivo no cambió Git sólo crea un enlace a la imagen anterior idéntica que tiene almacenada.



Terminología {

Repositorio: es la carpeta principal donde se encuentran almacenados los archivos que componen el proyecto. El directorio contiene metadatos gestionados por Git, de manera que el proyecto es configurado como un repositorio local

Commit: un commit es el estado de un proyecto en un determinado momento de la historia del mismo, imaginemos esto como punto por punto cada uno de los cambios que van pasando. Depende de nosotros determinar cuántos y cuales archivos incluirá cada commit.

Rama (branch): una rama es una línea alterna del tiempo en la historia de nuestro repositorio. Funciona para crear features, arreglar bugs, experimentar, sin afectar la versión estable o principal del proyecto. La rama principal por defecto es master

}



Terminología {

Pull Request: en proyectos con un equipo de trabajo, cada persona puede trabajar en una rama distinta, pero llegado el momento puede pasar que dicha rama se tenga que unir a la rama principal. Para eso se crea un pull request donde comunicamos el código que incluye los cambios, es revisado, comentado y aprobado para darle merge. En el contexto de GIT, merge significa unir dos trabajos, en este caso la rama branch con la rama master.

Merge (mezclar o fusionar): es el proceso de combinar los cambios de dos ramas diferentes en una sola rama. Esto es útil cuando varios desarrolladores trabajan en diferentes características o correcciones de errores en paralelo y luego desean integrar sus cambios en una rama principal, como main o master.

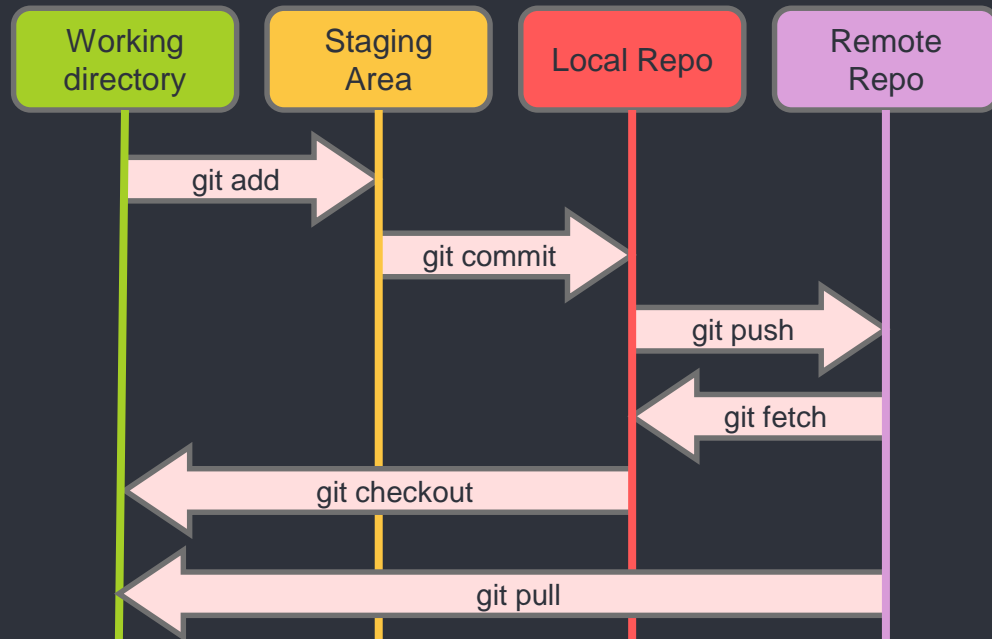
}



Flujo de trabajo {

```
# Git registra en
nuestro directorio local
los cambios que se
producen en los
archivos o código, cada
vez que se lo
indiquemos. De esta
forma podemos “viajar en
el tiempo” revirtiendo
cambios o restaurando
versiones de código.
```

```
# Esto puede hacerse
localmente o de forma
remota (servidor
externo).
```

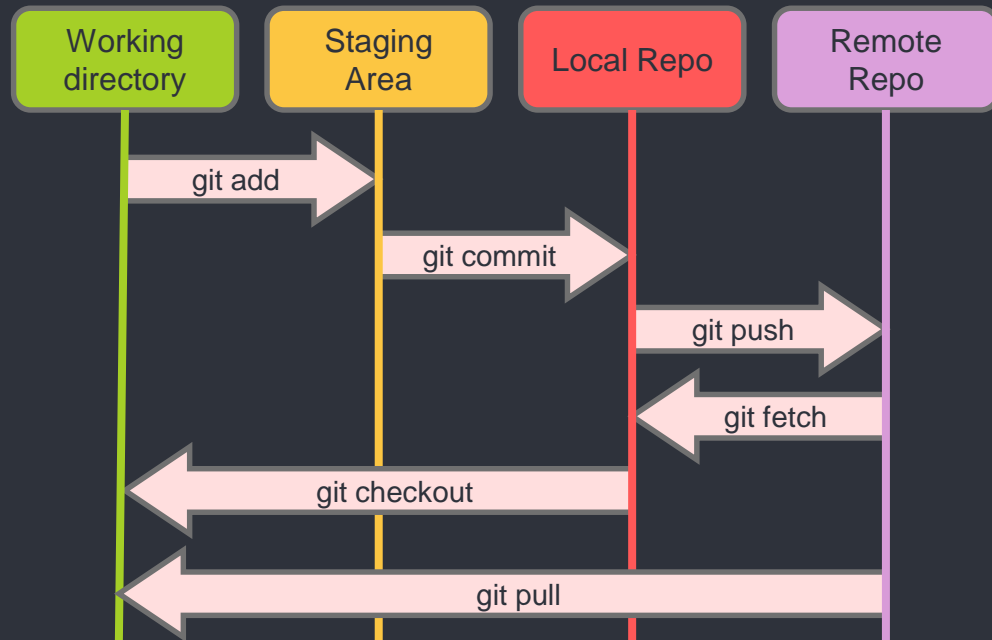


Flujo de trabajo {

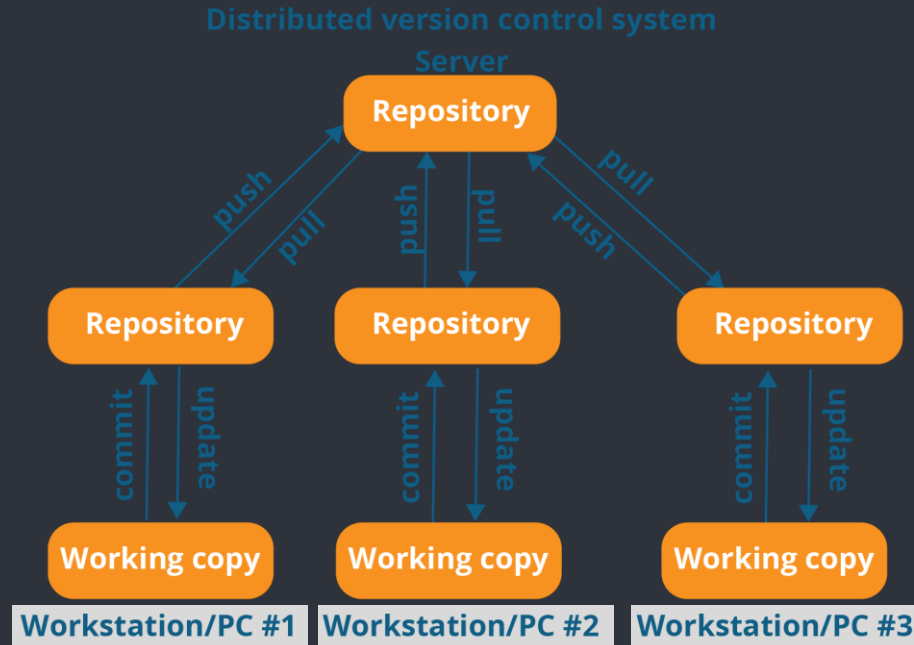
Working directory es nuestro directorio de trabajo. Cada vez que queremos agregar un archivo al staging area, usamos **git add**.

Luego, cuando queremos establecer un punto de restauración, ejecutamos **git commit**, y los archivos son actualizados en el repositorio (repository).

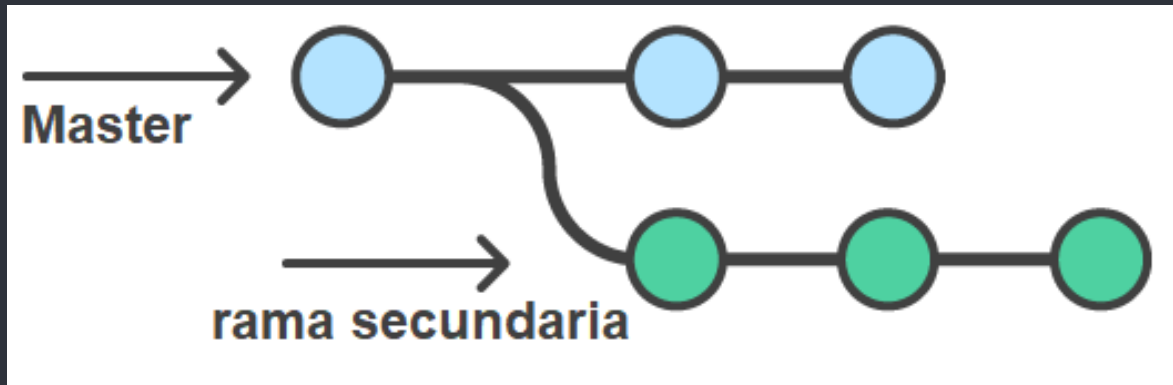
Diagrama de secuencia ->



Flujo de trabajo en equipo {



Branch {

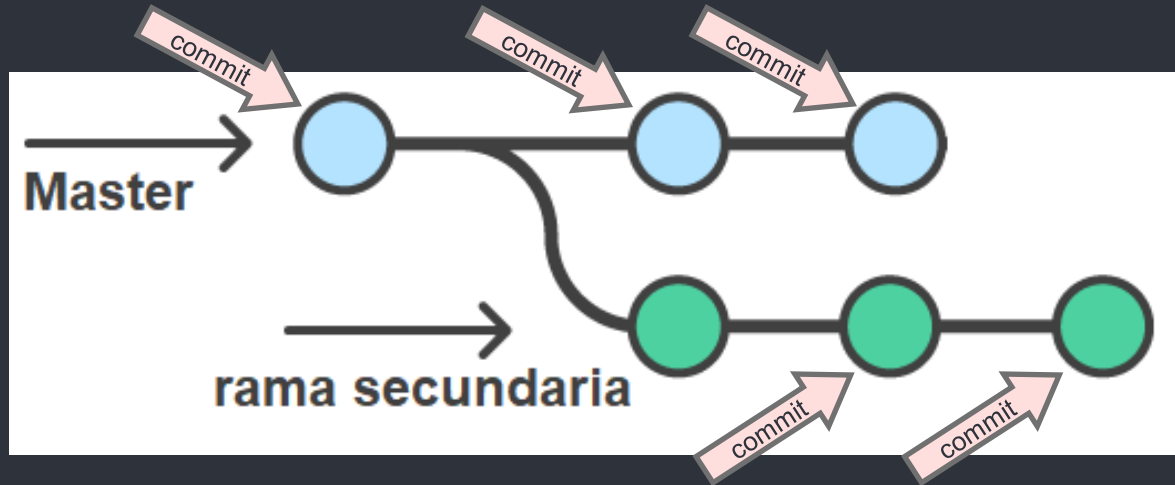


El trabajo principal suele llamarse Master, Main, Production, etc.

git branch rama_secundaria: con este comando vamos a crear una rama paralela a la rama Master, donde podremos hacer todos los cambios que deseemos. La rama Master no se modifica en absoluto, mientras que la nueva rama es la que va siendo modificada. **git branch** sin mas argumentos, nos muestra en que rama estamos parados.



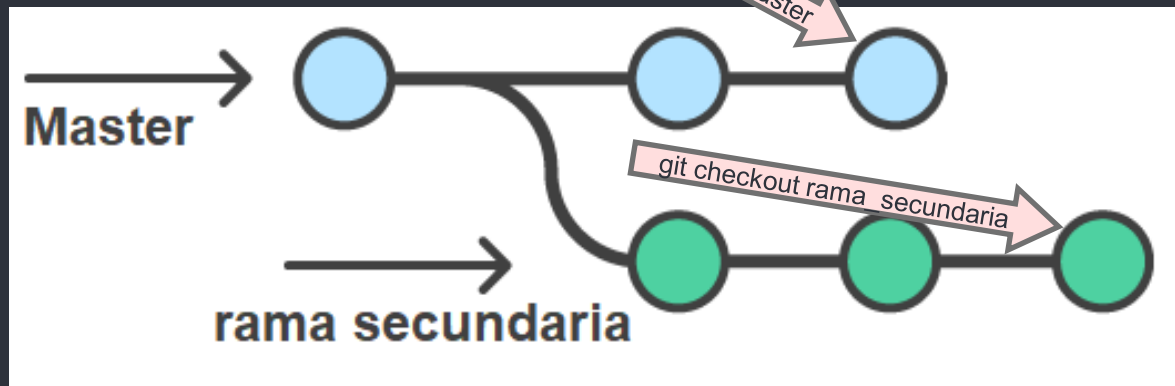
Commit {



Cada **commit** que hagamos, “fija” los cambios realizados, es como hacer click en guardar en un archivo de Word, con la diferencia que ahora vamos a tener un seguimiento de todos estos cambios. Cada círculo es un **commit**.



Checkout {

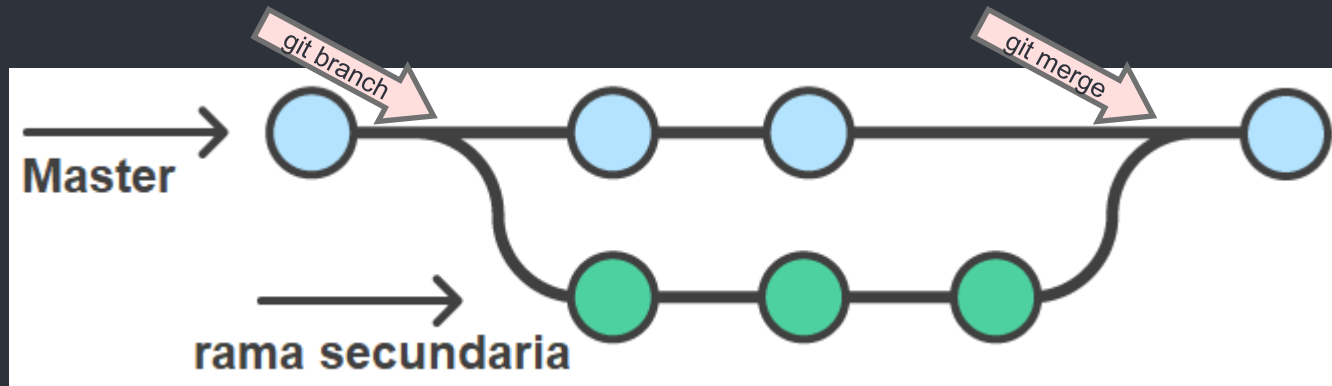


git checkout master: nos posiciona en el ultimo commit de la rama master.

git checkout rama_secundaria: nos posiciona en el ultimo commit de la rama rama_secundaria.



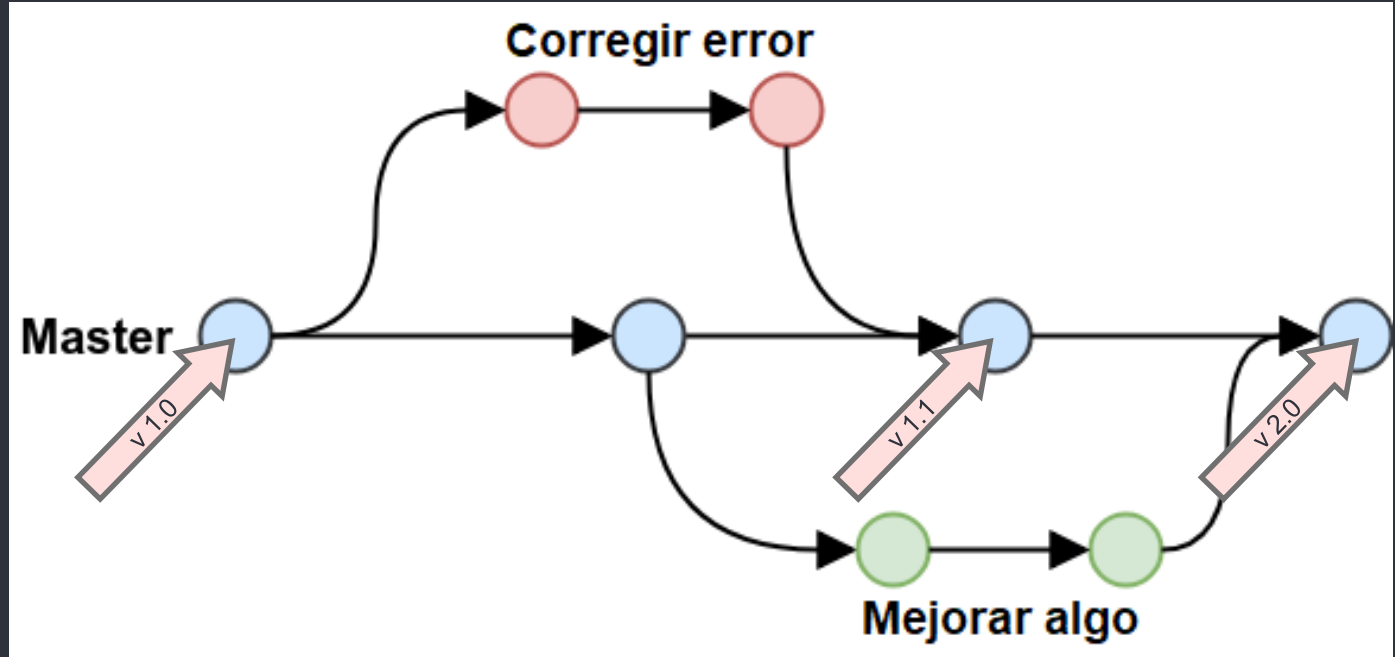
Merge {



Parados desde la rama de destino (en este caso la rama master **git checkout master**) hacemos **git merge rama_secundaria**. Con este comando vamos a mezclar, fusionar o “mergear” la rama paralela “rama_secundaria” a la rama Master.



Instalación {



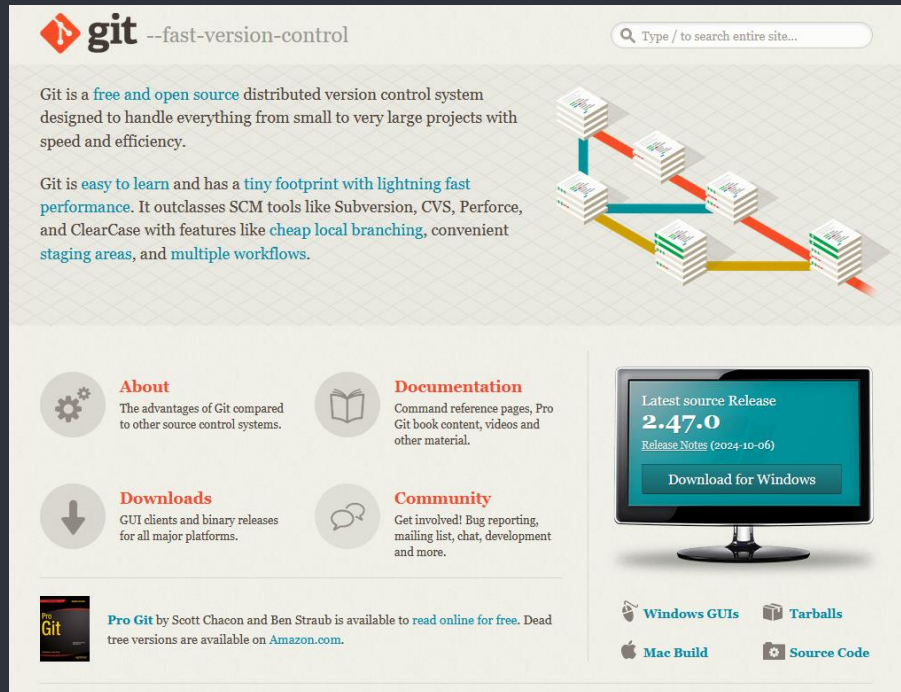
Instalación {

```
# Para descarga e instalar  
git, nos dirigimos a la  
siguiente pagina y  
descargamos el instalador
```

```
# https://git-scm.com/
```

```
# Procedemos luego a  
instalarlo como hacemos  
normalmente con todos los  
programas.
```

}



The screenshot shows the Git website homepage. At the top, there's a navigation bar with the Git logo and the tagline "--fast-version-control". Below this, a search bar is visible. The main content area features a large illustration of a branching model with multiple stacks of code blocks connected by lines. Text on the page describes Git as a "free and open source" distributed version control system designed for speed and efficiency. It also mentions that Git is "easy to learn" and has a "tiny footprint with lightning fast performance". Below the main text, there are four circular icons representing different sections: "About" (gears), "Documentation" (book), "Downloads" (download arrow), and "Community" (speech bubbles). Each section has a brief description. On the right side, there's a section for the "Latest source Release 2.47.0" with a "Download for Windows" button. At the bottom, there are links for "Windows GUIs", "Tarballs", "Mac Build", and "Source Code".



Primera configuración {

```
# Antes de realizar algunas de las operaciones más importantes de git,  
# necesitamos indicar cuál es nuestra dirección de correo y cuál es nuestro  
# nombre. Esto se hace con los comandos siguientes:
```

```
# Proporcionar la dirección de correo:
```

```
# git config --global user.email correodelusuario@dominio.com
```

```
# Proporcionar el nombre del propietario:
```

```
# git config --global user.name "NombreDelUsuario"
```

```
# Consultar los datos que tenemos registrados:
```

```
# git config --global -e
```

```
}
```



git init {

El primer paso para utilizar git en un proyecto consiste en inicializar la carpeta que lo contiene, convirtiéndola en un repositorio local. Para ello, utilizando los comandos provistos por el sistema operativo debemos ubicarnos en ella, y utilizar el comando **git init**.

Esto genera por defecto una rama llamada **"master"**.

}

```
MINGW64:/c/Users/Zek005/Desktop/practica_micropython

ZeK005@ZeK005-PC MINGW64 ~/Desktop
$ cd practica_micropython/

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython
$ git init
Initialized empty Git repository in C:/Users/Zek005/Desktop/practica_micropython/.git/

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (master)
$
```



git status {

Para ver el estado de la rama actual y su contenido utilizamos:

git status

Podemos renombrar el nombre de la rama con el comando

git branch -m <nombre>

Vamos a utilizar main para mantener los lineamientos de las plataformas.

}

```
MINGW64:/c/Users/ZeK005/Desktop/practica_micropython
ack)
ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (master)
$ git branch -m main

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (main)
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (main)
$
```



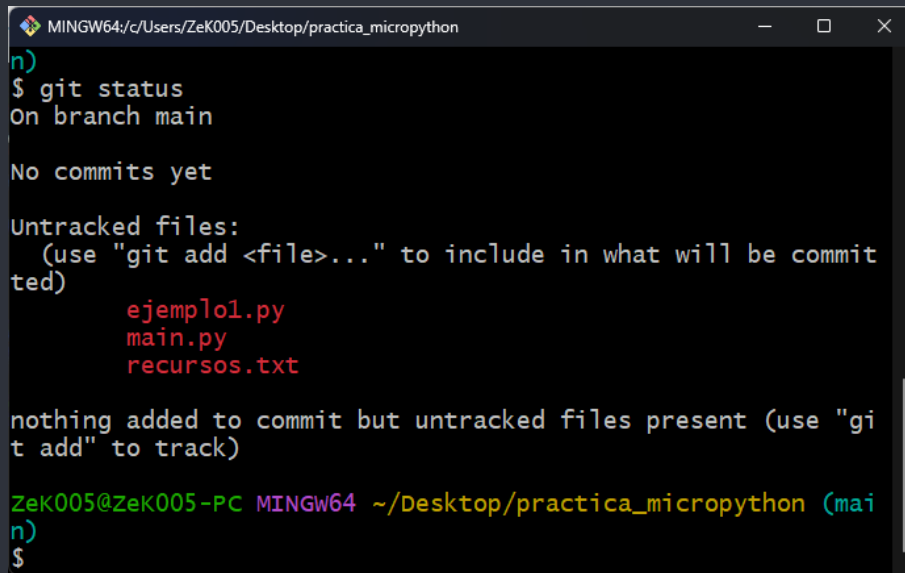
git status {

```
# Luego de trabajar dentro  
de la carpeta, hacemos  
nuevamente un status
```

git status

```
# Veremos en color rojo los  
archivos agregados o  
modificados que todavía no  
están siendo "trackeados",  
o sea, que no fueron  
agregados a la staging  
área.
```

```
}
```

A terminal window titled 'MINGW64: c:/Users/Zek005/Desktop/practica_micropython' showing the output of the 'git status' command. The output indicates the current branch is 'main', there are no commits yet, and lists three untracked files: 'ejemplo1.py', 'main.py', and 'recursos.txt' in red text. It also provides instructions on how to track these files using 'git add'.

```
MINGW64: c:/Users/Zek005/Desktop/practica_micropython  
n)  
$ git status  
On branch main  
  
No commits yet  
  
Untracked files:  
  (use "git add <file>..." to include in what will be commit  
ted)  
    ejemplo1.py  
    main.py  
    recursos.txt  
  
nothing added to commit but untracked files present (use "gi  
t add" to track)  
  
Zek005@Zek005-PC MINGW64 ~/Desktop/practica_micropython (mai  
n)  
$
```



git add {

Para incorporar los
archivos al área de staging
lo hacemos con el comando

git add <archivo>

Si queremos incorporar
todos los archivos a la vez
lo hacemos con el comando

git add .

Si hacemos un status
nuevamente, veremos los
archivos listos para commit
en verde.

}

```
MINGW64~/c/Users/ZeK005/Desktop/practica_micropython
$ git status
On branch main

No commits yet

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

Untracked files:
  (use "git add <file>..." to include in what will be commit
ted)
        ejemplo1.py
        recursos.txt

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (mai
n)
$
```



```
1 git commit {
```

```
2  
3  
4 # Los archivos del staging area se envían al repositorio local utilizando  
5 el comando git commit -m "comentario", donde "comentario" es una  
6 explicación de los cambios implicados. Es importante incluir una  
7 descripción relevante en cada commit, ya que será lo que git nos mostrará  
8 cuando veamos el "historial" de cambios realizados.
```

```
9  
10 # Cada vez que realizamos un commit, git genera un punto de restauración  
11 al cual es posible volver en cualquier momento.
```

```
12  
13 # Si no incluimos el comentario (git commit), y hemos configurado un  
14 editor de texto, git abre una ventana para que lo hagamos. Grabamos,  
15 cerramos, y el commit se habrá realizado.
```

```
16 }
```



```
1 git commit {
```

```
2  
3  
4 # git commit -m "comentario"
```

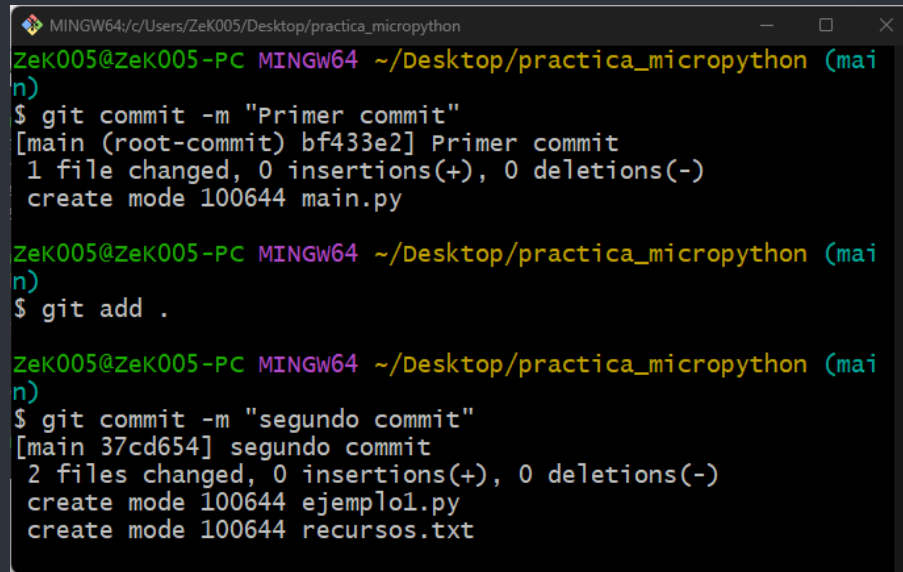
```
5  
6 # Vemos como se agrega un  
7 solo archivo al repositorio.
```

```
8 # Procedemos luego agregando  
9 el resto de archivos
```

```
10 # git add .
```

```
11 # git commit -m "comentario"
```

```
12  
13 }  
14
```



```
MINGW64/c/Users/ZeK005/Desktop/practica_micropython  
ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (mai  
n)  
$ git commit -m "Primer commit"  
[main (root-commit) bf433e2] Primer commit  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 main.py  
  
ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (mai  
n)  
$ git add .  
  
ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (mai  
n)  
$ git commit -m "segundo commit"  
[main 37cd654] segundo commit  
2 files changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 ejemplo1.py  
create mode 100644 recursos.txt
```



git log {

Luego de hacer el commit,
si queremos obtener un
historial de los cambios
realizados en los archivos
que integran nuestro
repositorio usamos **git log**

El ciclo de trabajo,
entonces, consiste en editar
los archivos, enviarlos al
staging area, y cuando
estamos listos, hacemos un
commit.

Repetimos este proceso las
veces que sea necesario.

}



```
MINGW64: c:/Users/ZeK005/Desktop/practica_micropython
ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (mai
n)
$ git log
commit 37cd6540c79cacb10235520064676e6f4742badf (HEAD -> mai
n)
Author: maxisimonazzi <maxisimonazzi@gmail.com>
Date: Mon Oct 28 22:53:21 2024 -0300

segundo commit

commit bf433e229fc4ac4e3d6a39022d1462658819671d
Author: maxisimonazzi <maxisimonazzi@gmail.com>
Date: Mon Oct 28 22:52:55 2024 -0300

Primer commit

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (mai
n)
$
```



git diff {

Una característica muy potente de git es la posibilidad de visualizar los cambios que se han producido en un archivo.

Con git diff podemos ver que líneas se agregaron (verde), eliminaron (rojo) o modificaron (amarillo) entre la versión actual del mismo y la del último commit:

git diff <archivo>

}

```
MINGW64/c/Users/Zek005/Desktop/practica_micropython
$ git diff main.py
diff --git a/main.py b/main.py
index 902b7d3..85aff9a 100644
--- a/main.py
+++ b/main.py
@@ -12,14 +12,8 @@ while True:
     verde.off()
     sleep(0.5)

- # -- Led Amarillo -- #
+ # -- Led Azul -- #
     amarillo.value(1)
     sleep(0.5)
     amarillo.value(0)
-     sleep(0.5)
-
- # -- Led Rojo -- #
-     rojo.value(True)
-     sleep(0.5)
```



Descartar cambios {

```
1
2
3
4
5     # Existen tres maneras de descartar cambios que hayamos realizado:
6
7     # git checkout <archivo>: descarta los cambios sobre el archivo y vuelve a
8     la versión que esté en el último commit del repositorio.
9
10    # git reset --hard: descarta todos los cambios no commiteados, sin
11    guardarlos. Vuelve a las versiones del último commit realizado.
12
13    # git stash: descarta todos los cambios no commiteados, guardándolos para
14    poder recuperarlos en un futuro.
```

```
}
```



Recuperar cambios descartados {

```
# Los cambios que han sido descartados con git stash pueden ser
recuperados.

# git stash list: lista todos los “puntos de restauración” que hemos
generado con “stash”. El más reciente tiene el índice 0 (cero).

# git stash show -p <stash-name>: Muestra los cambios que se encuentran
guardados en un stash en particular.

# git stash apply <stash-name>: Recupera los cambios desde un stash en
particular (no se elimina el punto de restauración).

# git stash drop <stash-name>: Elimina un “punto de restauración” de forma
definitiva, y la pila de cambios stasheados se reordenará. Esta acción es
irreversible.
```

}



.gitignore {

Cuando no necesitamos que todos los archivos de nuestro proyecto sean gestionados por git podemos hacer una lista con los archivos y/o carpetas a excluir, y guardarla en un archivo de texto que tenga el nombre **.gitignore**.

Se debe poner un nombre por línea, y todos los archivos allí listados serán ignorados por git.

}

```
MINGW64/c/Users/ZeK005/Desktop/practica_micropython
$ git status
On branch main
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)

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (main)
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

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



Ramas {

Podemos crear una nueva rama en nuestro proyecto, mediante estos comandos:

git branch: muestra la(s) ramas que componen el proyecto.

git branch <nombre de la rama>: crea una nueva rama con el nombre indicado

git checkout <nombre de la rama>: cambio a otra rama para trabajar en ella.

}

```
MINGW64~/c/Users/ZeK005/Desktop/practica_micropython
ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (main)
$ git branch
* main

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (main)
$ git branch rama_secundaria

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (main)
$ git checkout rama_secundaria
Switched to branch 'rama_secundaria'

ZeK005@ZeK005-PC MINGW64 ~/Desktop/practica_micropython (rama_secundaria)
$ |
```



GitHub {

GitHub es una plataforma de repositorios remotos. Además de permitir ver el código y descargar diferentes versiones de una aplicación, la plataforma también conecta desarrolladores para que puedan colaborar en un mismo proyecto.

Podemos sincronizar repositorios locales con repositorios remotos, clonar en nuestra PC repositorios públicos de terceros, utilizar la plataforma como un mecanismo de backup de nuestros repositorios locales. Para poder subir gratis los proyectos deberán ser de código abierto. Ofrece también una herramienta de revisión de código, en la que se pueden dejar anotaciones para mejorar y optimizar el código.

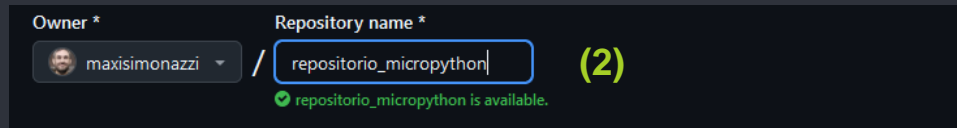
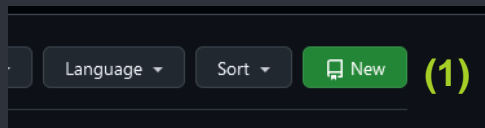
Link: <https://github.com/>

}



Crear repositorio remoto {

Creamos un repositorio (1), le damos un nombre (2) y lo enlazamos con nuestro repositorio local mediante el comando que nos muestra la plataforma (3).



...or create a new repository on the command line

```
echo "# repositorio_micropython" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/maxisimonazzi/repositorio_micropython.git
git push -u origin main
```

(3)

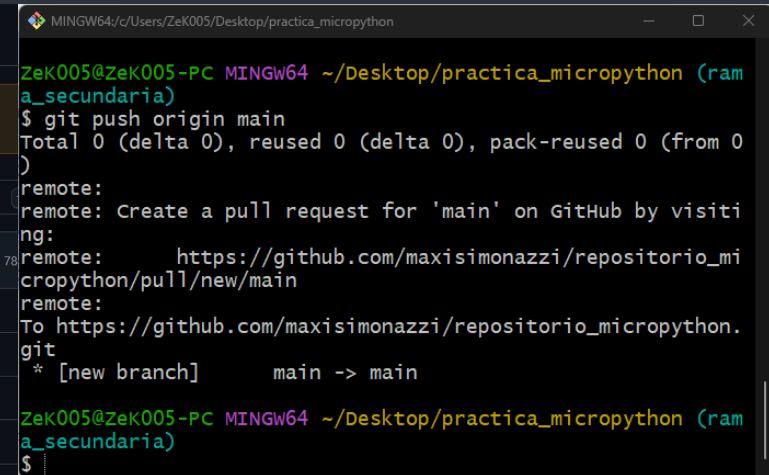
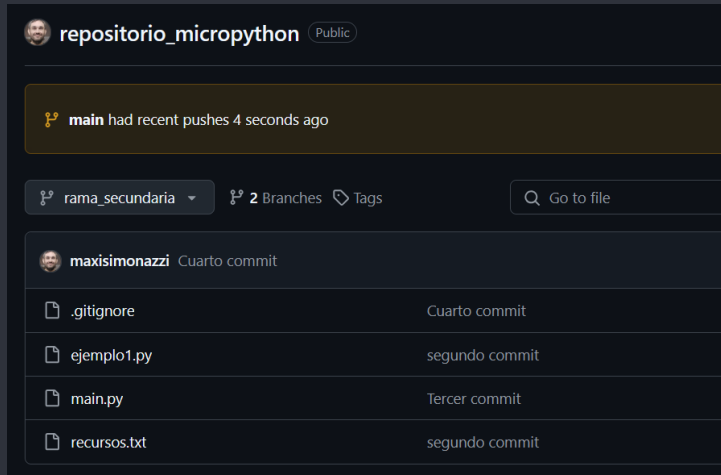
...or push an existing repository from the command line

```
git remote add origin https://github.com/maxisimonazzi/repositorio_micropython.git
git branch -M main
git push -u origin main
```



git push {

El comando **git push -u origin <rama>** sincroniza una rama del repositorio local con el repositorio remoto. Necesitamos el nombre de usuario en GitHub y un token que se obtiene desde **"Usuario -> Settings -> Developer settings -> Personal tokens"**.



git push {

git push es un comando de carga que permite subir los commits realizados en nuestro repositorio local a GitHub. Una vez allí, estos pueden ser descargados por el resto del equipo de trabajo.

Para crear una rama local en el repositorio de destino utilizamos:
git push <remote> <branch>

Si queremos enviar todas las ramas locales a una rama remota especificada.

git push <remote> --all

Una vez movidos los conjuntos de cambios se puede ejecutar un comando **git merge** en el destino para integrarlos.

}



git clone {

En caso de querer sincronizar nuestro trabajo con el de otro usuario, en forma local, podemos clonar su repositorio:

git clone <url repositorio externo>

Hacer los cambios necesarios, commitarlos, y luego, con push, enviarlos nuevamente al repositorio remoto. En este caso, en el push usaremos nuestro usuario y el token del propietario del repositorio.

}

```
MINGW64/c/Users/ZeK005/Desktop
ZeK005@ZeK005-PC MINGW64 ~/Desktop
$ git clone https://github.com/maxisimonazzi/repositorio_micropython.git
Cloning into 'repositorio_micropython'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 12 (delta 2), reused 11 (delta 2), pack-reused 0 (from 0)
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (2/2), done.
ZeK005@ZeK005-PC MINGW64 ~/Desktop
$
```



```
1 git pull {
```

```
2  
3  
4  
5 # El comando git pull se emplea para extraer y descargar contenido desde  
6 un repositorio remoto y actualizar al instante el repositorio local para  
7 reflejar ese contenido. El comando git pull es, en realidad, una  
combinación de dos comandos, git fetch seguido de git merge.
```

```
8  
9 # git pull <remote>: Recupera la copia del origen remoto especificado de  
la rama actual y la fusiona de inmediato en la copia local.
```

```
10  
11 # git pull --no-commit <remote> Recupera la copia del origen remoto, pero  
no crea una nueva conformación de fusión.
```

```
12  
13  
14 }
```



```
1
2
3 Aprender a programar
4
5 es aprender a pensar.
6
7
8
9
10
```

```
11 { Steve Jobs; }
12
13
14
```



```
1  
2  
3  
4  
5 { Nos vemos en la  
6 proxima clase }  
7  
8  
9  
10  
11  
12  
13  
14
```

