

Git y Github

Objetivos

- Mostrar el sistema de control de versiones *Git* junto con la plataforma de desarrollo colaborativo *Github* con el fin de que el estudiante conozca una nueva herramienta para el desarrollo colaborativo y ordenado de software.

Introducción

¿Cuántas veces no has guardado un proyecto en una carpeta con el nombre "Proyecto x (Este es el bueno)"? Git viene a solucionar este problema, además de ayudarnos a mejorar la manera de colaborar con otros miembros del equipo. Pero, ¿dónde puedo subir mi código para que no se quede en mi computadora y tenga que andarlo pasando por correo o USBs y se pueda utilizar de manera inmediata? Ese es el problema que Github resuelve. La llamada "red social de programadores" nos proporciona un lugar donde cada desarrollador pueda subir sus proyectos y desde cualquier parte del mundo un miembro del equipo pueda utilizar la nueva actualización los proyectos sin tanto esfuerzo. Durante este capítulo veremos estas tecnologías y la forma de mezclarlas con el fin de exprimir las distintas soluciones que cada una de éstas contribuye.

Git

¿Qué es Git?

¿Qué es un sistema controlador de versiones? Es un sistema que guarda los cambios dentro de un proyecto a un archivo o conjunto de archivos sobre el tiempo para que se pueda obtener versiones específicas después. Pongamos como ejemplo un lenguaje de programación

como Java, Java tiene distintas versiones lanzadas y tu puedes utilizar cualquiera de ellas. Existen distintos tipos de sistemas de controlador de versiones:

- **Sistema de Controlador de Versiones Local.** Se trata de un programa de vaya guardando todos los cambios de un archivo o archivos dentro de la computadora de todos los archivos dentro del proyecto. El problema con este tipo es que los registros de los cambios se quedan sólo dentro de la computadora.
- **Sistema de Controlador de Versiones Centralizado.** Se trata de un servidor que contiene el registro de todos los archivos versionados y entonces los clientes pueden obtener esos archivos. El problema con este tipo es que si el servidor se cae entonces ningún cliente puede obtener ni agregar cambios.
- **Sistema de Controlador de Versiones Distribuido.** Aquí cada cliente tiene todo el historial de los cambios. Si un servidor se cae, entonces se puede montar otro y cualquier cliente puede utilizar su historial para restaurarlo.

Git es una sistema de control de versiones distribuido libre y de código abierto diseñado para manejar todo desde pequeños a grandes proyectos con velocidad y eficiencia¹. Aunque *Git* no es único sistema controlador de versiones distribuido (también existen *Mercurial* ², *Bazaar* ³ o *Darcs* ⁴), sí es verdad que es el más popular y el más utilizado.

¹ pag_git

² mercurial

³ bazaar

⁴ darcs

¿Cómo es que funciona *Git*?

Al igual que muchos Sistemas Controladores de Versiones, *Git* guarda todos los cambios del proyecto en un directorio. Muchos otros Sistemas Controladores de Versiones guardan los cambios de cada archivo por cada una de las versiones. La forma en que *Git* guarda los cambios es con *snapshots* de cada uno de los archivos por cada versión. Para ser eficiente, si un archivo no cambia con respecto a la versión pasada entonces *Git* no vuelve a almacenar el archivo otra vez, en su lugar enlaza el archivo idéntico anterior que ya ha sido almacenado. Como se manejan las versiones en *Git* es mediante *commits*, por lo que con cada *commit* se guardará los *snapshots* de cada uno de los archivos del proyecto.

Veamos los distintos estados para cada archivo del proyecto:

- **Committed.** Los datos del archivo están guardados seguramente en la base de datos local del proyecto de *Git*.

- **Modified.** Se han cambiado los datos pero no se ha agregado a un *commit* ninguno de estos cambios a la base de datos del proyecto de *Git*.
- **Staged.** Se ha marcado una modificación del archivo en su actual version para agregarse al siguiente *snapshot* del siguiente *commit*.

Para terminar éste es el flujo de trabajo dentro de un proyecto de *Git*:

1. Se modifican algunos archivos dentro del proyecto de *Git*.
2. Se seleccionan los archivos a los que queremos que sean parte del siguiente *commit* para pasar al estado de **Staged**.
3. Se hace un *commit*, lo que hace que los archivos que estén en el estado de **Staged** se guarden en la base de datos del proyecto de *Git* permanentemente.⁵

⁵ [que_es_git](#)

Instalar Git

Instalar *Git* depende del sistema operativo. En este link se pueden ver las indicaciones oficiales para instalarlo:

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Aunque existen aplicaciones gráficas para utilizar los comandos de *Git*, la manera más popular de utilizar *Git* es mediante la terminal del sistema operativo. Durante el transcurso del manual se trabajarán todos los comandos en la terminal de un sistema UNIX.

Se verifica que tenemos instalado *Git* de manera exitosa si podemos ejecutar el siguiente comando en la terminal del respectivo sistema operativo:

```
-> git
```

Crear un repositorio de Git

Los proyectos de *Git* se hacen llamar **repositorios**. Para crear un repositorio ejecutamos siguiente comando:

```
-> git init <nombre del directorio del repositorio>
```

Esto nos creará un directorio del nombre que le pongamos en donde parecerá a primera vista que es un directorio vacío pero contendrá el directorio **.git** que es la base de datos donde *Git* guardará todos los commits e información referente al repositorio. Casi nunca se trabaja directamente con el directorio **.git**.

Agregar un commit

Entramos al directorio del repositorio y éste sólo tendrá el directorio (oculto) **.git**. Por el momento agregaremos un *Hello world!* escrito en *Python* como ejemplo para agregar a un *commit*. Creamos un archivo de python con el nombre **hello.py**

```
-> touch hello.py
```

Desde que creamos el archivo, el mismo archivo ya está en el estado de *Modified* ya que aún no lo movemos al área de *Staged*. Para verificar cuáles archivos están en el estado de *Modified* o de *Staged* se ejecuta el comando de **git status**⁶.

⁶ git status

```
-> git status
```

Este es un ejemplo de como aparecería para los cambios recién hechos:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will
   be committed)

    hello.py

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

Lo que nos está diciendo es que el archivo **hello.py** que acabamos de agregar aún no es rastreado por el repositorio ya que es nuevo. Así

como dice al momento de ejecutar **git status**, para pasarlo al estado de *Staged* ejecutamos el comando de **git add**⁷.

⁷ git add

```
-> git add hello.py
```

Ahora al momento de ejecutar el comando de **git status** obtendremos lo siguiente.

```
On branch master

No commits yet

Changes to be committed:
  (use "git tm --cached <file>..." to unstage)

        new file:   hello.py
```

Ahora tenemos nuestro archivo con los cambios (el cambio actual es que se creó el archivo) en el estado de *Staged*. Para por fin ponerlo en un *commit* (es decir crear una nueva versión) y guardar el cambio de forma permanente en la base de datos de *Git* ejecutamos el comando de **git commit**⁸.

⁸ git commit

```
-> git commit -m "Agregando el archivo hello.py"
```

Esto lo que hace es agregar todos los cambios a los archivos que están en el estado de *Staged* a un nuevo *commit* con el nombre de *Agregando el archivo hello.py*. Ahora al momento de ejecutar el comando de **git status** obtendremos lo siguiente.

```
On branch master
nothing to commit, working tree clean
```

Lo que nos quiere decir que no hay ningún archivo en el estado de *Modified* o en *Staged*. Para verificar que se creó el *commit* de manera exitosa ejecutamos el comando de **git log**⁹:

⁹ git log

```
-> git log
```

Lo que hace este comando es mostrar todos los *commits* que se han hecho. Este es la salida de ejecutar el comando con nuestros cambios actuales:

```
commit <hash> (HEAD -> master)
Author: <autor> <email>
Date:   <Fecha>

Agregando el archivo hello.py
```

Como podemos observar, cada *commit* es identificado con un hash. Además se muestra quién hizo cada *commit* y la fecha en que fue realizado. ¡Listo! Ya tenemos nuestro primer *commit*.

Ejercicio 0.0.1. Agrega el siguiente código al archivo de `hello.py`:

```
print("Hello World!")
```

y agrega los cambios a un nuevo *commit* con el nombre de "Agregando el *print*".

Regresar a un commit anterior

Hasta el momento, al momento de ejecutar `git log --oneline` tenemos los siguientes *commits*:

```
70f9c64 (HEAD -> master) Agregando el print
deae6aa Agregando el archivo hello.py
```

Por lo que hasta el momento tenemos 2 versiones de nuestro archivos. Supongamos que uno de nuestro desarrolladores no se dio cuenta que estaba en un archivo de *Python* y quiso agregar código *Java*. Agreguemos la siguiente línea a nuestro programa `hello.py`:

```
System.out.println("¡Hola Mundo!");
```

Y agregamos este cambio a un nuevo *commit* llamado Agregando *¡Hola mundo!*.

```
-> git add -A
-> git commit -m "Agregando ¡Hola Mundo!"
```

Ahora tenemos los siguiente *commits*:

```
d2afb9c (HEAD -> master) Agregando ¡Hola Mundo!
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Pero un desarrollador que sólo sabe *Python* no sabría que es lo que significa eso y por qué le sale error al momento de quererlo ejecutar. Uno podría sólo borrar esa línea y agregar un nuevo *commit* con la línea borrada pero ya tendríamos redundancia en los *commits* ya que 2 *commits* tendrían exactamente el mismo estado para cada archivo. Lo correcto sería regresar al *commit* donde teníamos todo correcto ya que tampoco queremos mantener el *commit* donde teníamos un error. Es justo lo que haremos. Existen varias maneras de regresar a un *commit* anterior. Para sólo regresar sin cambiar nada (digamos, sólo queremos regresar en modo lectura) podemos utilizar el comando **git checkout <hash de commit>**¹⁰. Se debe de poner el *hash* del *commit* para indicar a cuál queremos regresar. Nosotros queremos regresar al *commit* de *Agregando el print* cuyo *hash* es **70f9c64** entonces ejecutamos:

```
-> git checkout 70f9c64
```

Revisamos qué *commits* tenemos con **git log -oneline**:

```
70f9c64 (HEAD) Agregando el print
deae6aa Agregando el archivo hello.py
```

Así podemos ver el estado de nuestros archivos hasta ese *commit*. Para regresar hasta el último *commit*:

```
-> git checkout master
```

Esto nos regresara al último *commit* que tenemos en la *branch* (rama) *master*.

Ahora supongamos que queremos regresar a un *commit* pero no queremos perder todos los cambios que tenemos hasta ese momento. Para esto el comando **git reset**¹¹ con la opción **--soft**. En general el comando **git reset** nos ayuda a regresar a otros *commits*. Ejecutamos lo siguiente para regresar al *commit* de *Agregando el print*:

```
-> git reset --soft 70f9c64
```

Lo que hace esto es quitar todos los *commits* que había después del

Notemos que ahora no agregamos el archivo en específico al momento de hacer el **git add**. La opción **-A** (que viene de *All*) del comando **git add** agrega todos los archivos que se han cambiado desde el último *commit*. En este caso sólo se agregó el archivo *hello.py* porque es el único que hay pero en otros casos que se cambian múltiples archivos se agregarían todos al estado de *Staged*.

¹⁰ **git checkout**

Más adelante veremos que *checkout* también tiene otras funciones.

Después veremos qué son las ramas. Hasta el momento sólo hemos trabajado con la *branch* *master*

git reset también necesita el *hash* del *commit* al que se regresa.

¹¹ **git reset**

Siempre se tiene que tener cuidado de utilizar **git reset** ya que se eliminan todos los *commits* hasta el *commit* deseado.

commit al que regresamos y pone ese *commit* como el último que se ha hecho. Veamos los *commits* actuales:

```
70f9c64 (HEAD -> master) Agregando el print
deae6aa Agregando el archivo hello.py
```

Pero al momento de ejecutar **git status** nos dice:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   hello.py
```

Nos dice que hay cambios en el archivo de `hello.py` que son todos los cambios que había después del *commit* al que regresamos (Estos cambios están en el estado de *Staged*). Regresemos esos cambios al estado de *Modified* con:

Aquí podemos ver otro uso del comando **git reset**

```
-> git reset HEAD hello.py
```

Así, al momento de ejecutar **git status** tenemos:

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be
   committed)
  (use "git checkout -- <file>..." to discard changes
   in working directory)

    modified:   hello.py

no changes added to commit (use "git add" and/or "git
commit -a")
```

Hay un comando muy útil para verificar los cambios que están en el estado de *Modified*. Este comando es **git diff**¹². Al ejecutarlo nos muestra:

¹² **git diff**

```
diff --git a/hello.py b/hello.py
index f301245..80d1b3b 100644
--- a/hello.py
```



```
+++ b/hello.py
@@ -1,2 @@
 print("Hello World!")
+System.out.println("¡Hola Mundo!");
```

Ahí nos muestra cada línea de cada archivo que fue modificada con respecto al *commit* anterior. En este caso nos muestra que se agregó una línea de código en el archivo de `hello.py`, después del `print("Hello World!")`.

Para terminar, ahora supongamos que queremos regresar al *commit* de *Agregando el print* pero sin mantener los cambios que se hicieron después de este *commit*. Para esta acción ejecutamos **git reset** con la opción **--hard** y el *hash* del *commit* al que queremos regresar:

```
-> git reset --hard 70f9c64
```

En este caso no tendremos ningún cambio en el estado *Modified* o *Staged*, simplemente tendremos todos los archivos en el estado en que están en ese *commit*. Recordemos que utilizar **git reset** elimina todos nuestros *commits* que estaban después del *commit* al que regresamos.

¡Recuerden utilizar con cuidado **git reset**!

Con este último comando se pudo eliminar el error del otro programador de agregar código *Java* dentro de un archivo de *Python* pero también se muestran las otras distintas opciones que tenemos para poder regresar a un *commit* antiguo.

Ejercicio 0.0.2. Experimentemos. Agrega 2 nuevos *commits* en donde en cada uno agregues un archivo que desees. Tus *commits* se deben de ver así:

```
229a73a (HEAD -> master) Agregando archivo2
2240339 Agregando archivo1
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Después regresa al *commit* de *Agregando el print* sin eliminar los cambios con **git reset --soft** y agrega un nuevo *commit* con esos cambios con el nombre de *Agregando archivo1 y archivo2*. Los *commits* deben de quedar de la siguiente forma:

```
ca77f48 (HEAD -> master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Y se deben de conservar los 2 archivos que se crearon:

```
-> ls
archivo1.py archivo2.py hello.py
```

Branches

Supongamos que ya tenemos un gran proyecto y queremos agregar una nueva característica. Tu eres el encargado de esa característica y las demás personas están trabajando en otras características ajenas a la tuya. Todos los nuevos *commits* que realizarás no afectarán a las demás personas y por lo tanto no es necesario que ellos lo tengan. Si no hasta que se haga la integración de todas las características para tener un producto final. Para este problema existen las *branches* (o ramas en español).

Una *branch* lo que hace es encapsular todos los *commits* posteriores y conservará todos los *commits* anteriores a donde se creó la *branch*. Otra *branch* no podrá ver los *commits* que encapsulen las demás *branches* (a menos que se mezclen pero eso lo veremos más adelante). Desde el inicio de nuestro repositorio tenemos una *branch* ya creada, esta *branch* es la *master*. Veamos esto como un árbol. Digamos que la *branch master* es el tronco y todas las demás son ramas que salen de él (por eso el nombre de *branches*). Con la única diferencia que ahí las ramas sí se pueden volver a juntar con otras ramas, es decir, una *branch* puede volver a unirse con otra *branch*.

La manera de crear *branches* es con el comando de **git branch <nombre de la branch>**¹³. Vamos a crear una *branch* con el nombre de *Primera_branch*:

```
-> git branch Primera_branch
```

Podemos listar nuestras *branches* con **git branch**. Al ejecutarlo tenemos:

```
Primera_branch
* master
```

El asterisco indica en cual *branch* estamos trabajando. Para cambiarnos de *branch* utilizaremos el comando de **git checkout <nombre de la branch>**. Vamos a cambiarnos a la nueva *branch* que creamos:

Hasta el momento sólo hemos visto como guardar *commits* dentro de nuestra computadora. Más adelante veremos como colaborar con Git.

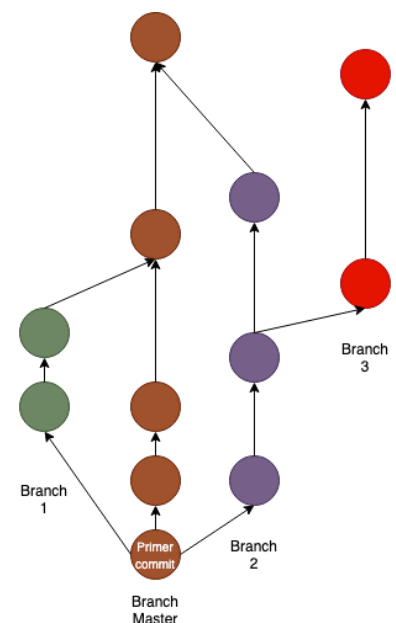


Figure 1: Ejemplo de *branches*. Cada círculo es un *commit*.

¹³ **git branch**

Crear una *branch* no nos lleva directamente a la *branch* creada.

```
-> git checkout Primera_branch
```

Cuando hacemos **git status** nos dice que ya estamos en *Primera_branch*:

```
On branch Primera_branch
nothing to commit, working tree clean
```

Ahora veamos los commits que tenemos en esta *branch* con **git log --oneline**:

```
ca77f48 (HEAD -> Primera_branch, master) Agregando
archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Podemos ver que tenemos los mismos *commits* que teníamos en la *branch master*, esto es debido a que la *branch* la creamos en *master*. Ahora crearemos un nuevo archivo llamado `archivo_otra_branch.py` y agregaremos un nuevo *commit* llamado *Agregando nuevo archivo en otra branch*. Ahora tenemos estos *commits*:

Cuando se crea una *branch* nueva, ésta toma todos los *commits* que tiene la *branch* de donde se está creando.

```
5fbfc55 (HEAD -> Primera_branch) Agregando nuevo
archivo en otra branch
ca77f48 (master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Si nos regresamos a la *branch master* podemos ver que ahora los *commits* no son los mismos y tampoco tenemos el archivo que se creo en la otra *branch*:

Recordemos que cambiamos de *branch* con **git checkout master**.

```
ca77f48 (HEAD -> master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
-> ls
archivo1.py archivo2.py hello.py
```

Regresemos a la *branch Primera_branch* y crearemos una nueva *branch* a partir de ahí:

```
-> git checkout Primera_branch
-> git checkout -b Segunda_branch
Switched to a new branch 'Segunda_branch'
```

Ahora esta nueva *branch* tiene los mismos *commits* que *Primera_branch*:

Veamos que con la opción **-b** de **git checkout** nos permite crear una nueva *branch* y movernos hacia ella.

```
5fbfc55 (HEAD -> Segunda_branch, Primera_branch) Agregando
nuevo archivo en otra branch
ca77f48 (master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Y si agregamos un nuevo *commit* en esta *branch* entonces ya no estaría al parejo ni con *master* ni con *Primera_branch*. Hagamos el ejemplo agregando un archivo *archivo_branch2.py* junto con su *commit* llamado *Agregando archivo en branch2*. Así está el estado actual de los *commits* en cada una de las *branches*:

Notemos que **git log** nos dice qué *branches* tienen como último *commit* de los *commits* mostrados

```
d1d59df (HEAD -> Segunda_branch) Agregando archivo en
branch2
5fbfc55 (Primera_branch) Agregando nuevo archivo en
otra branch
ca77f48 (master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
-> ls
archivo1.py archivo2.py archivo_branch2.py
archivo_otra_branch.py hello.py

5fbfc55 (HEAD -> Primera_branch) Agregando nuevo archivo
en otra branch
ca77f48 (master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
-> ls
archivo1.py archivo2.py archivo_otra_branch.py hello.py

ca77f48 (HEAD -> master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
-> ls
```

```
archivo1.py archivo2.py hello.py
```

Mezclando Branches

Una vez que se ha terminado una característica y quieres unir la rama de dicha característica con alguna otra entonces se hace una mezcla de *branches*. El comando que nos permite 2 *branches* es **git merge <nombre de la rama>**¹⁴. Lo que hace es poner todos los commits que estaban en la *branch* que se quiere unir a la otra. Hagamos el ejemplo uniendo nuestra *branch Segunda_branch* con *Primera_branch*. Como queremos la que quiere se mezclada es *Segunda_branch* entonces nos tenemos que primero parar en *Primera_branch* y de ahí unir *Segunda_branch*:

¹⁴ git merge

```
-> git checkout Primera_branch
Switched to branch 'Primera_branch'
-> git merge Segunda_branch
Updating 5fbfc55..d1d59df
Fast-forward
 archivo_branch2.py | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 archivo_branch2.py
```

Al ejecutar el comando nos dice cuáles son los archivos que se actualizarán. En este caso nos dijo que se agregó el `archivo_branch2.py` que no estaba en *Primera_branch*. Pero no todo es tan sencillo como parece ¿no?. En ocasiones puede haber problemas con las mezclas debido a que 2 *branches* modifican el mismo archivo. Actualmente *Primera_branch* tiene todos los commits de *Segunda_branch*. Modificaremos `archivo_branch2.py` en ambas *branches* y veremos que es lo que pasa. En *Primera_branch* agregaremos a `archivo_branch2.py` lo siguiente y lo guardaremos en un commit llamado *Agregando print en archivo branch 2*:

Cabe destacar que cuando mezclas una *branch* ésta no se borra.

En este primer ejemplo no hubo problemas porque ambas *branches* trabajaron con archivos distintos.

```
print("Este es el archivo branch 2")
```

En *Segunda_branch* agregaremos a `archivo_branch2.py` lo siguiente y lo guardaremos en un commit llamado *Agregando print en archivo branch 2*:

```
print("Este es el archivo_branch2")
```

Ahora queremos mezclar *Primera_branch* en *Segunda_branch*. Al momento de mezclar ¿Qué ocurrirá? ¿Con qué versión del archivo se quedará? Veamos:

```
-> git checkout Segunda_branch
Switched to branch 'Segunda_branch'
-> git merge Primera_branch
Auto-merging archivo_branch2.py
CONFLICT (content): Merge conflict in archivo_branch2.py
Automatic merge failed; fix conflicts and then commit the result.
```

Git trató de hacer una mezcla automática como lo hizo en la anterior pero no pudo porque ambos commits modificaron la misma línea de código en el mismo archivo. En su lugar nos dice que hay un conflicto en `archivo_branch2.py`. Cuando hacemos **git status** nos dice lo siguiente:

```
On branch Segunda_branch
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   archivo_branch2.py

no changes added to commit (use "git add" and/or
"git commit -a")
```

Podemos ver que hay cambios en `archivo_branch2.py` que estan en estado de *Modified*. Veamos esos cambios:

```
<<<<<<< HEAD
print("Este es el archivo_branch2")
=====
print("Este es el archivo branch 2")
>>>>>>> Primera_branch
```

Nos muestra los 2 cambios que se hicieron en la misma línea junto con unos signos extraños. Cuando hay conflictos al momento de

mezclar *branches* en *Git* esta es la forma en que nos lo muestra. Primera nos pone los cambios que están en HEAD junto con 7 "<" para delimitar donde inician y 7 "=" para delimitar donde inician y justo después nos pone los cambios que están en la otra *branch* que queremos mezclar hasta 7 ">" junto con el nombre de la *branch* que delimita donde terminan esos cambios. Si nuestro archivo que tiene conflictos tiene más de un conflicto entonces mostraría algo parecido por cada uno de los conflictos que tenga a lo largo del archivo. Para resolver el conflicto sólo basta modificar el archivo que lo tenga hasta que se quede satisfecho con el archivo y hacer *commit*. Nosotros borramos y modificamos el archivo hasta que nos quedamos con lo que necesitamos:

HEAD se refiere al último commit del actual *branch*.

```
print("Este es el archivo_branch2 o archivo branch 2")
```

Una vez que ya tenemos lo que necesitamos entonces hacemos *commit*:

```
-> git add -A
-> git commit -m "Resolviendo conflictos al mezclar con
Primera_branch"
```

Y los *commits* que tenemos son los siguientes:

```
44e0580 (HEAD -> Segunda_branch) Merge branch
'Primera_branch' into Segunda_branch
e544207 Agregando print en archivo branch 2
ad6c642 (Primera_branch) Agregando print en archivo
branch 2
d1d59df Agregando archivo en branch2
5fbfc55 Agregando nuevo archivo en otra branch
ca77f48 (master) Agregando archivo1 y archivo2
70f9c64 Agregando el print
deae6aa Agregando el archivo hello.py
```

Podemos ver que conserva los 2 *commits* de *Agregando print en archivo branch 2* y agrega un *commit* de mezcla que es donde se resolvieron los conflictos. Puede que esto resulte complicado en un principio pero al momento de trabajar colaborativamente los conflictos cuando se mezclan *branches* serán muy comunes por lo que poco a poco se hará normal y más sencillo.

Ejercicio 0.0.3. .

- Crea una nueva *branch* desde *Primera_branch* con el nombre de *Tercera_branch*.
- Modifica el print de `archivo_branch2.py` con lo siguiente y agrégalo a un commit con el nombre que quieras:

```
print("Modificando desde Tercera_branch")
```

- Haz una mezcla desde *Tercera_branch* con *Segunda_branch*, resuelve el conflicto como quieras y haz *commit* del conflicto resuelto.

Hasta ahora sólo hemos visto como trabajar con *Git* pero dentro de nuestra computadora sin compartir el proyecto con ninguna persona. Ahora veremos la forma de colaborar con *Git*.

Github

¿Qué es Github?

Github es un sitio de servicio de alojamiento web para proyectos de *Git*. Aunque hay otros servicios de alojamiento, *Github* provee control de acceso a otros desarrolladores y muchas características de colaboración tales como rastreo de errores, peticiones de características, manejo de tareas, y *wikis* para cada proyecto.¹⁵

Muchos lo consideran una "red social para desarrolladores" porque si tu proyecto es público entonces los demás desarrolladores pueden compartir ideas sobre el proyecto e incluso colaborar en él. También tiene opción para tener proyectos privados para que sólo ciertos desarrolladores puedan ver y colaborar. *Github*

tiene planes gratis en donde se pueden crear proyectos públicos ilimitados y algunos privados. Si se desea más características o más proyectos privados entonces tiene opciones de pago.

Conectar tu proyecto local *Git* a *Github*

Primero se debe de crear una cuenta en www.github.com. Una vez que fue creada y que se ha iniciado sesión, primero se debe de crear un repositorio al cuál asociar con nuestro repositorio local. Para crear un repositorio nos vamos al menú que está en la esquina superior derecha y damos click en tus repositorios (*Your repositories* en el caso en que esté en inglés.) Una vez que estamos en tus repositorios

Otros ejemplo son *Bitbucket* y *GitLab*.

¹⁵ <https://github.com/features>

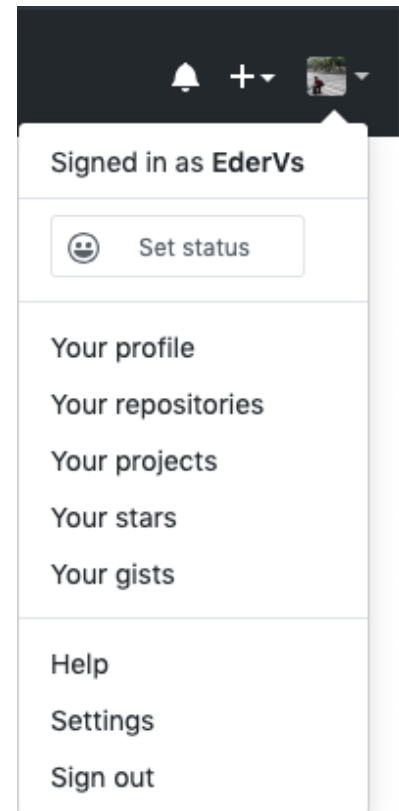



Figure 2: Donde se encuentra tus repositorios.

Overview **Repositories 62** Projects 0 Stars 2 Followers 53 Following 31

Find a repository... Type: All Language: All 

entonces damos click en *new*. Después nos muestra un formulario donde pondremos el nombre de nuestro repositorio, la descripción, si será público o privado, si queremos inicializar con un README nuestro repositorio, si queremos agregar gitignore (más adelante se verá eso) y el tipo de licencia que tendrá. El nombre de nuestro repositorio será *Repositorio Ejemplo*, lo pondremos público, no agregaremos README, gitignore ni licencia. Damos click en *Create repository*.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner **EderVs** / Repository name **Repositorio Ejemplo** ✓

Great repository name **Your new repository will be created as Repositorio-Ejemplo laughing-couscous?**

Description (optional)
Repositorio de ejemplo

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

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

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

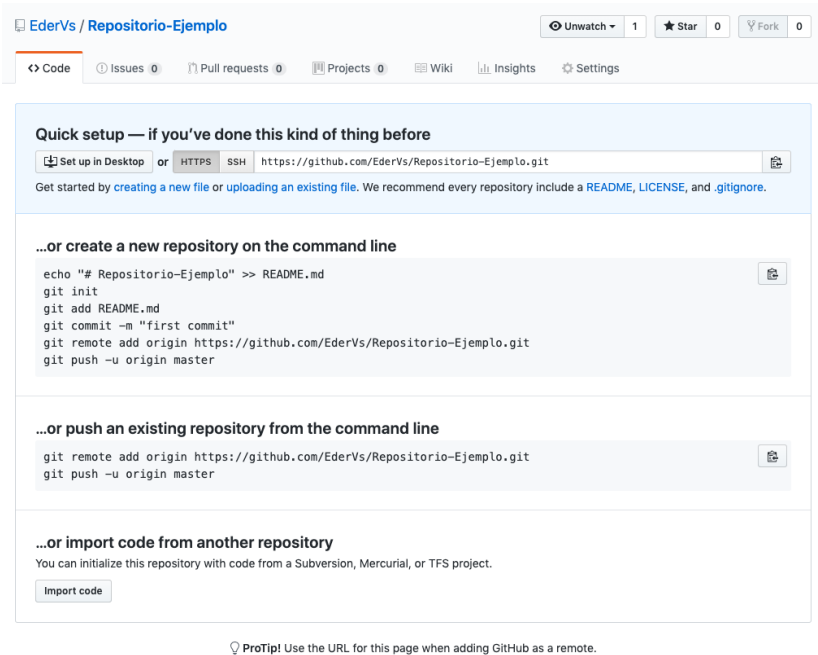
Add .gitignore: **None** Add a license: **None** ⓘ

Create repository

Una vez que se creo el repositorio nos direcciona a la página donde se encuentra nuestro repositorio y nos muestra las diferentes opciones para vincular nuestro repositorios locales:

Lo que nosotros haremos será agregar un repositorio existente desde la línea de comandos. Por lo que prácticamente haremos lo que nos explica hacer. Para vincular un repositorio local a cualquier sitio de alojamiento se utiliza el comando **git remote**¹⁶. Dentro de nuestro directorio del proyecto, en la línea de comandos, ejecutamos el siguiente comando como nos mostró *Github*:

¹⁶ **git remote**



```
-> git remote add origin
    https://github.com/EderVs/Repositorio-Ejemplo.git
```

Con esto vinculamos el repositorio con el que está en *GitHub* con el nombre de *origin*. Pero hasta ahora no tenemos todos los commits dentro de nuestro repositorio en *GitHub*. La forma de indicar a *Git* que se van a subir los cambios que tenemos localmente al alojamiento vinculado es con el comando **git push**¹⁷. Por lo que lo ejecutamos como nos dice *GitHub*:

¹⁷ **git push**

```
-> git push -u origin master
```

Esto lo que hace es subir los *commits* que no estén en el repositorio de *GitHub* de la *branch master* al repositorio que esté en *origin*.

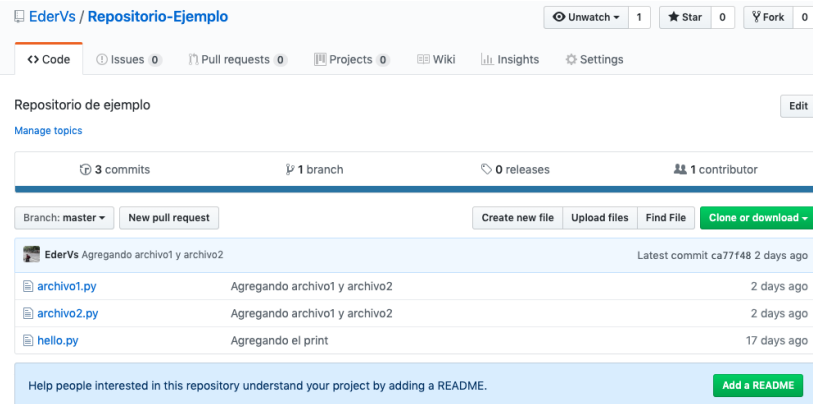


Figure 3: Ahora sí tenemos nuestros archivos dentro del repositorio de *GitHub*.

Ejercicio 0.0.4. Con lo hecho arriba sólo se subieron los *commits* que estaban en la *branch master*. Sube de la misma forma los commits que están en las otras *branches* que tenemos. Para verificar que se subieron, debes de tener tu repositorio así:

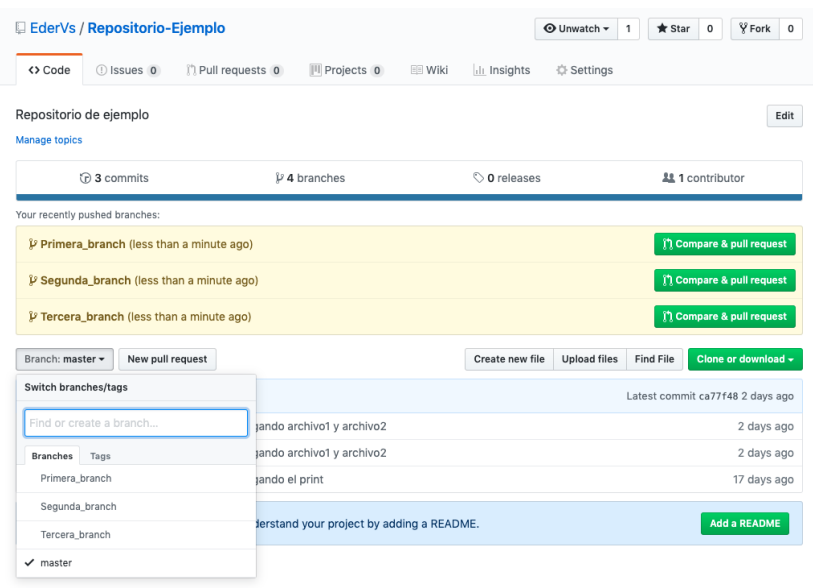


Figure 4: Así se muestran las *branches*

Teniendo nuestro proyecto dentro de Github ya es más fácil obtenerlo y empezar a trabajar con él. Para obtener un repositorio de Github y tenerlo de manera local utilizamos el comando **git clone <URL de repositorio>**¹⁸.

Hagamos el ejemplo de clonar el repositorio. Movamonos a otro directorio. Dentro de este directorio ejecutemos:

¹⁸ `git clone`

```
-> git clone <url de tu repositorio>
```

Llamemos éste repositorio que clonamos "Repositorio copia" y el que tenemos "Repositorio original".

Ejercicio 0.0.5. Dentro del "Repositorio Original" agrega un archivo `README.md` y que dentro tenga el nombre del repositorio de la siguiente manera:

```
// Repositorio ejemplo
```

Agrega un *commit* con ese cambio y con nombre *Agregando README* en la *branch master*. Después sube los cambios con **git push origin master** al repositorio de *Github*.

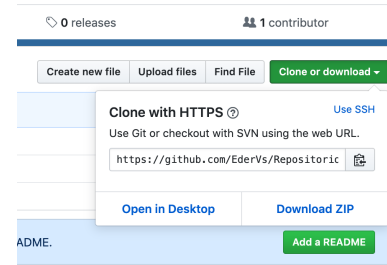


Figure 5: Dentro de nuestro repositorio de Github podemos ver cuál es la URL.

Recordemos que "Repositorio Original" y "Repositorio copia" están en directorios distintos.

Mantener actualizado tu repositorio local

Una vez que se han subido distintos *commits* desde distintos colaboradores tu repositorio local quedará desactualizado del repositorio que esté en *Github*. Mediante el anterior ejercicio nuestro "Repositorio Copia" quedó desactualizado de lo que hay en el repositorio de *Github*, por lo que lo actualizaremos. Hay 2 comandos que nos ayudarán a mantener actualizado nuestro repositorio, **git fetch**¹⁹ y **git pull**²⁰.

Git no sólo guarda los *commits* que tiene registrados en el repositorio local, si no también guarda los *commits* que hay en los repositorios remotos. . Al ejecutar **git fetch** lo que hace es actualizar los *commits* que tiene guardados del repositorio remoto (En este caso el repositorio de *Github*) pero no afecta los *commits* locales. Probemos ejecutándolo desde el "Repositorio Copia":

```
-> git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/EderVs/Repositorio-Ejemplo
ca77f48..e049fee master -> origin/master
```

¹⁹ **git fetch**

²⁰ **git pull**

Los Repositorios Remotos son los que los sitios de alojamiento mantienen.

En donde se actualizaron los *commits* es en `origin/master`. Así es como *Git* guarda los *commits* remotos: con el apodo del repositorio remoto, una diagonal y la *branch*. Probamos hacer un **git status**:

```
-> git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Nos dice que nuestra *branch* local está atrasada por 1 *commit* a lo que tiene en el repositorio local. En realidad `origin/master` es una *branch* por lo que podríamos hacer *merge*. Hagamos la prueba desde la *branch master* del "Repositorio copia":

```
-> git merge origin/master
Updating ca77f48..e049fee
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Y si ejecutamos **git status**:

```
-> git status:
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Entonces ya actualizamos nuestro repositorio local. Ahora probaremos con **git pull** pero antes configuraremos el mismo escenario.

Ejercicio 0.0.6. Dentro del "Repositorio copia" modifica el README agregando la descripción del repositorio de la siguiente manera:

```
// Repositorio ejemplo
Este es un repositorio ejemplo para aprender Git y Github
```

Agrega un *commit* con este cambio y con nombre *Agregando descripción en el README* en la *branch master*. Después sube los cambios con **git push origin master** al repositorio de *Github*.

Ahora tenemos "Repositorio original" desactualizado.

git pull lo que hace es aparte de actualizar los *commits* de los repositorios remotos, hace el *merge* automáticamente a la *branch* actual. Probemos ejecutando **git pull** en "Repositorio original":

```
-> git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0),
pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/EderVs/Repositorio-Ejemplo
   e049fee..5e467b7  master    -> origin/master
Updating e049fee..5e467b7
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

Y entonces ya tenemos actualizado nuestro repositorio local.

Cabe destacar que al momento de actualizar nuestro repositorio local entonces lo que se hace es *merge*, por lo que se debe estar preparado para posibles colisiones.

Python

Objetivos

- Crear programas realizados en el lenguaje de programación *Python* utilizando el paradigma Orientado a Objetos.
- Subir programas a su cuenta de *Github* para compartirlo al mundo y comparar resultados con otros desarrolladores.

Introducción

¿Qué es lo que hace a *Python* uno de los lenguajes de programación más populares del mundo? Se debe a su simpleza y a su comunidad que cada día está creando nuevas bibliotecas para muchos sectores. Python es un lenguaje de programación de alto nivel interpretado. Fue creado en 1991 por Guido van Rossum y actualmente está en su versión 3.7. A continuación presentamos algunas de sus características:

- Es un lenguaje de programación multi-paradigma. Generalmente se utiliza para Programación Orientada a Objetos y Programación Estructural pero también se pueden realizar funciones de Programación Funcional, e incluso con extensiones se puede realizar Programación Lógica.
- Tiene tipado dinámico y cuenta con recolector de basura.
- Tiene una filosofía de diseño que enfatiza la lectura de código usando muchos espacios en blanco para indentación y palabras reservadas sencillas. Además no utiliza puntos y comas (;) para separar sentencias.
- Fue diseñado para ser altamente extensible por lo que su núcleo es muy pequeño y todo se basa en sus bibliotecas estándar.

Los programas de *Python* tienen la terminación **.py**. Se pueden ejecutar los archivos utilizando el comando **python <nombre del archivo>**.

```
-> python hello.py
Hello World!
```

Debido a que *Python* es un lenguaje interpretado también se puede abrir la terminal de *Python* ejecutando el comando **python** y ahí ejecutar sentencia por sentencia.

```
-> python
Python 3.6.4 (default, Mar 1 2018, 18:36:50)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)]
on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> 5 + 5
10
>>>
```

Suponemos conocimiento previo de *Java* para tomarlo como punto de comparación con *Python*.

Instalación

Python se puede instalar descargando el lenguaje y siguiendo las indicaciones dependiendo del sistema operativo.

Desde <https://www.python.org/downloads/> se puede obtener la última versión.

Variables y tipos de datos

Uso de variables

El uso de las variables es el primer ejemplo de la simpleza y facilidad de *Python*. La forma de declarar una variable es la siguiente:

```
nombre_variable = valor
```

Los nombres de las variables pueden tener letras [a-zA-Z], dígitos [0-9] y guiones bajos [_]; sin embargo los nombres de las variables no

pueden empezar con un dígito. Como podemos ver, no se necesita el tipo de la variable para crearla.

Tipos de datos

Python utiliza el llamado *Duck Typing* para definir el tipo de una variable aunque es tipado fuertemente ya que si se quiere hacer una operación que no esté definida entre 2 tipos de datos entonces no deja que pase. Los tipos de datos que Python tiene en su núcleo son:

Si camina como pato y hace *quak* como pato, entonces debe ser un pato.

- *bool*. Valor Booleano. Ejemplo: **True** | **False**
- *bytearray*. Secuencia de *bytes*. Se puede obtener cada uno de los valores utilizando *brackets* (arreglo_byte[o]). Ejemplo: **bytearray(b'secuencia en ASCII')** | **bytearray(b"secuencia en ASCII")** | **bytearray([119, 105, 107, 105])**
- *bytes*. Es muy parecido a *bytearray* con la diferencia que es inmutable. Ejemplo: **b'secuencia en ASCII'** | **b"secuencia en ASCII"** | **bytes([119, 105, 107, 105])**
- *complex*. Números complejos. Ejemplo: **5.5+7i**
- *dict*. Diccionario o mejor conocido como *hash map*. Contiene llaves y valores que pueden ser de cualquier tipo con la condición de que las llaves deben de ser *hasheables*. Ejemplo: **{'llave': 1, 5: True, 5.5: 'valor'}**
- *float*. Número de representación de coma flotante. Ejemplo: **5.12345**
- *frozenset*. Lo mismo que *set* pero inmutable. Ejemplo: **frozenset([5, 'cadena', True])**
- *int*. Número entero con tamaño ilimitado. Ejemplo: **7**
- *list*. Lista. Puede contener distintos tipos en la misma lista. Se puede obtener cada uno de los valores utilizando *brackets* (lista[1]). Ejemplo **[1, 'cadena', 5.4, [5, 6]]**
- *set*. Conjunto no ordenado. No contiene duplicados. Puede contener distintos tipos pero deben poder ser *hasheables*. Ejemplo. **{5, 'cadena', True}**
- *str*. Cadena de caracteres. Ejemplos: **'Hola'**, **"Hola"**, **"""String de multiples líneas"""**.

- **tuple**. Como una lista inmutable. Se puede obtener cada uno de los valores utilizando *brackets* (tupla[2]). Ejemplos: (1, 'cadena', 5.4, [5, 6])

Operadores y funciones comunes

Muchos de los operadores que se utilizan en *Python* son muy parecidos a los de los demás lenguajes de programación. Estos son algunos de sus operadores y funciones comunes:

- **Operadores que regresan un bool.**

- **==**. Compara la igualdad entre 2 objetos. Ejemplo:

```
x == 5
```

- **!=**. Compara la desigualdad entre 2 objetos. Ejemplo:

```
y != x
```

- **not**. Operador *NOT* para booleanos. Ejemplo:

```
not True
```

- **and**. Operador *AND* para booleanos. Ejemplo:

```
x and y
```

- **or**. Operador *OR* para booleanos. Ejemplo:

```
x or y
```

- **<**. Operador de "menor que". Ejemplo:

```
x < 6
```

- <=. Operador de "menor o igual que". Ejemplo:

```
y <= 7
```

- >. Operador de "mayor que". Ejemplo:

```
x > 8
```

- >=. Operador de "mayor o igual que". Ejemplo:

```
y >= 9
```

- **Operadores aritméticos.**

- +. Operador de suma. Ejemplo:

```
7 + 8
```

- -. Operador de resta o de negación. Ejemplo:

```
7.1 + 8.9  
-8
```

- *. Operador de producto o multiplicación. Ejemplo:

```
x * 8
```

- /. Operador de división. Ejemplo:

```
10 / 2
```

- `//`. Operador de división entera. Ejemplo:

```
10 // 3 # Esto da 3
```

- `**`. Operador de exponenciación. Ejemplo:

```
2 ** 10
```

- `%`. Operador de módulo. Ejemplo:

```
35 % 2
```

- **Operadores de asignación.**

- `=`. Operador de asignación. ejemplo:

```
mi_variable = 2
```

- `+=`. Suma lo que se pone a lo que hay en la variable y actualiza el valor. Ejemplo:

```
mi_variable += 2 # Ahora mi_variable vale 4
```

- `-=`. Resta lo que se pone a lo que hay en la variable y actualiza el valor. Ejemplo:

```
mi_variable -= 1 # Ahora mi_variable vale 3
```

- ***=**. Multiplica lo que se pone a lo que hay en la variable y actualiza el valor. Ejemplo:

```
mi_variable *= 4 # Ahora mi_variable vale 12
```

Ahora algunas de las funciones y sentencias más utilizadas en python:

- **print()**. Imprime en terminal. Ejemplo:

```
print("Hola")  
print(x)  
print(True)
```

- **type()**. Nos indica el tipo de un objeto. Ejemplo:

```
type(5) # int  
type("Hola") # str
```

- **min()**. Nos regresa el valor más pequeño. Pueden enviarse una cantidad ilimitada de elementos o una lista. Ejemplo:

```
min(1,2,3,4) # 1  
min([5, 2, 0, 2]) # 0  
min(8, 2) # 2
```

- **max()**. Los mismo que min pero nos regresa el valor más grande.
- **abs()**. Nos regresa el valor absoluto. Ejemplo:

```
abs(2) # 2  
abs(-2) # 2
```

- **round()**. Nos regresa el valor redondeado. Ejemplo:

```
round(5.2) # 5  
round(5.6) # 6
```

- **sum()**. Nos regresa la suma de los valores dentro de una lista que le pasemos como argumento. Ejemplo:

```
sum([1,2,3,4]) # 10
```

- **sorted()**. Ordena una lista. Ejemplo:

```
sorted([5,2,3,1,4]) # [1,2,3,4,5]
```

- **len()**. Nos regresa el tamaño de una colección. Ejemplo:

```
len([1,2,3,4,5,5]) # 6
```

- **return**. Regresa un valor de una función. Ejemplo:

```
return 7
```

- **from import**. Importa bibliotecas. Ejemplo:

```
from random import randint  
# De esta manera se utiliza así:  
randint(5,10)  
# o  
import random.randint  
# De esta manera se utiliza así:  
random.randint(5,10)
```

- **with**

Estructuras de control

Desde aquí se ve la diferencia de *Python* con los demás lenguajes de programación. Para definir un bloque de código no se necesita *curly-braces*({}). Lo que se necesita son dos puntos (:) para definir donde empieza el bloque de código y todo lo que esté indentado a la derecha estará dentro del bloque de código. Para la indentación generalmente se utilizan *tabs* ó 4 espacios pero no ambos al mismo tiempo, esa es la única condición que pone *Python*. En los ejemplos de las estructuras de control se verá más claro.

Algunas de las estructuras de control en Python:

- **if.** Sentencia *if*. Se necesita una condición que regrese un *bool*. También se puede agregar un sentencia *else* y una sentencia *elif* que es la combinación de un *else* y un *if*. Ejemplo:

La condición no está entre paréntesis.

```
# Código
if x == 'a':
    y = 0
if x == 'b':
    y = 1
elif x == 'c':
    y = 2
else:
    y = 3
# Más código
```

- **while.** Sentencia *while*. Se necesita una condición que regrese un *bool*. Ejemplo:

```
# Código
while x < 7:
    x += 1
# Más código
```

- **for.** Sentencia *for*. Se necesita algo con que iterar. Se puede definir un iterador para una clase o utilizar algunos objetos o tipos que ya lo tienen. Un ejemplo son las listas. Otro muy utilizado es la función **range(a, b)** que regresa un iterador. También se necesita una variable que tendrá el valor de lo que itera. Ejemplo:

```
# Código
for x in range(25): # Esto equivale de 0 a 24
    y += 1
for x in range(2, 5): # Esto equivale de 2 a 4
    y += 1
for x in ['a', 2, [1,2]]:
    print(x)
# Más Código
```

- **def.** Sentencia para definir una función ó método. Los nombres de las funciones tienen las mismas restricciones que los nombres de las variables. Los argumentos de las funciones no necesitan que se defina el tipo al momento de definirlos ni tampoco se necesita definir el tipo que regresa la función. Ejemplo:

```
# Código
def fib(x):
    if x == 0 or x == 1:
        return 1
    return fib(x-1) + fib(x-2)
# Más Código
```

- **pass.** Sentencia que define que en un bloque no se está haciendo nada. Esta sentencia existe ya que *Python* manda un error cuando no hay nada en un bloque de código por la indentación. Ejemplo:

```
def alguna_funcion_sin_definir():
    pass
```

- **try, except y finally.** Sentencias para definir el flujo cuando un error sucede. Muy parecidas a las sentencias **try, catch y finally** en *Java*. Dentro de la sentencia **catch** se puede meter el tipo de error que espera. Ejemplo:

```
# Código
try:
```



```

    print(x)
except NameError: # Sólo obtiene el error NameError
    print("No existe la variable x")
finally:
    print("Ese era el valor de la variable x")
# Más Código
try:
    # Código
except: # Obtiene todos los posibles errores.
    # Código
# Más código

```

Ejercicio 0.0.7. Una matriz se puede representar como una arreglo de 2 dimensiones. En *Python* se puede representar como listas dentro de una lista. Se puede definir una matriz de la siguiente manera:

```

# Matriz de 3x3
matriz = [[1,1,0], [0,1,1], [1,0,1]]

```

Define una función que reciba una matriz y que regrese el espejo en vertical de esa matriz. El espejo del ejemplo anterior sería:

```

matriz_espejo = [[0,1,1], [1,1,0], [1,0,1]]

```

El código debe de estar dentro de una función y regresar una matriz que sea el espejo en vertical de la matriz que se recibe.

```

def matriz_espejo_vertical(matriz):
    # TODO: Código para regresar la matriz
    #         espejo en vertical
    return matriz_espejo

```

Programación Orientada a Objetos

La Programación Orientada a Objetos es muy parecida a la de Java. La definición de clases es la siguiente:

```
class Persona:
    # Métodos de la clase

# También se puede crear heredando otras clases
class Persona(SerVivo):
    # Métodos de la clase

# En Python está también la posibilidad de la multiherencia
class Persona(Carnivoro, Animal):
    # Métodos de la clase
```

Así como en *Java*, en *Python* se refiere al objeto de la clase dentro de la clase con una palabra reservada. La palabra reservada en *Python* es **self**. Todos los métodos en *Java* ya tienen la referencia del objeto de la clase pero en *Python* se lo tenemos que agregar en cada definición de los métodos dentro de la clase:

La palabra reservada en *Java* es **this**

```
class Persona:
    def respirar(self):
        # Acción de respirar
```

Para definir el constructor de la clase debemos de utilizar el nombre de **__init__** para el método.

```
class Persona:
    def __init__(self):
        # Constructor de la clase
```

Debemos definir los atributos de la clase dentro del constructor y debemos también tener que utilizar la palabra reservada **this**.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

No se pueden agregar más constructores a la clase debido a que

La única condición es que los argumentos no predeterminados deben de estar antes de los otros.

no tenemos sobrecarga de funciones pero podemos agregar valores predeterminados a los argumentos:

```
class Persona:
    def __init__(self, nombre, edad, planeta="Tierra",
                  edo_civil="Soltero"):
        self.nombre = nombre
        self.edad = edad
        self.planeta = planeta
        self.edo_civil = edo_civil
```

Todos los métodos de la clase que se quieran mandar a llamar dentro de la clase deben de estar después de **self**.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
        self.imprimir_nombre()

    def imprimir_nombre(self):
        print(self.nombre, "está vivo")
```

Para poder utilizar herencia se puede utilizar la palabra reservada *super* o directamente utilizar el nombre del padre en la herencia.

```
class Animal:
    def __init__(self, edad):
        self.edad = edad

    def respirar(self):
        print("Respirando")

class Persona(Animal):
    def __init__(self, nombre, edad):
        # Mandamos a llamar el constructor del padre pero
        # con la referencia del objeto de la clase
        Animal.__init__(self, edad)

        # Seguimos con el constructor normal
        self.nombre = nombre
        self.respirar()
```

```

        self.presentarse()

    def presentarse(self):
        print("Mi nombre es", self.nombre,
              "y tengo", self.edad)

# También podemos utilizar super() pero sólo cuando
# tenemos un padre
class Persona(Animal):
    def __init__(self, nombre, edad):
        # Mandamos a llamar el constructor del padre
        super().__init__(edad)

        # Seguimos con el constructor normal
        self.nombre = nombre
        self.respirar()
        self.presentarse()

    def presentarse(self):
        print("Mi nombre es", self.nombre,
              "y tengo", self.edad)

```

Para crear objetos de una clase no es necesario agregar una palabra reservada como **new** en *Java*, sólo es como si mandáramos a llamar a la clase:

```

class Animal:
    def __init__(self, edad):
        self.edad = edad

    def respirar(self):
        print("Respirando")

class Persona(Animal):
    def __init__(self, nombre, edad, planeta="Tierra"):
        Animal.__init__(self, edad)

        self.nombre = nombre
        self.respirar()
        self.presentarse()

    def presentarse(self):

```

```

        print("Mi nombre es", self.nombre,
              "tengo", self.edad, "y soy del planeta",
              self.planeta)

persona1 = Persona("Timmy", 10)
# Mandar a llamar un método de la clase de un objeto
persona1.respirar()

```

Existen algunos nombres de métodos reservados que se pueden sobrescribir. Un ejemplo es el método `__str__`.

```

class Animal:
    def __init__(self, edad):
        self.edad = edad

    def respirar(self):
        print("Respirando")

class Persona(Animal):
    def __init__(self, nombre, edad, planeta="Tierra"):
        Animal.__init__(self, edad)

        self.nombre = nombre
        self.respirar()
        self.presentarse()

    def presentarse(self):
        print("Mi nombre es", self.nombre,
              "tengo", self.edad, "y soy del planeta",
              self.planeta)

    def __str__(self):
        # Se debe de regresar un str
        return self.nombre + " de " + str(self.edad)

persona1 = Persona("Dexter", 10)
print(persona1)

```

Para complementar se dejan los siguientes *links*:

- <https://www.programiz.com/python-programming/object-oriented-programming>

- <https://python.swaroopch.com/oop.html>