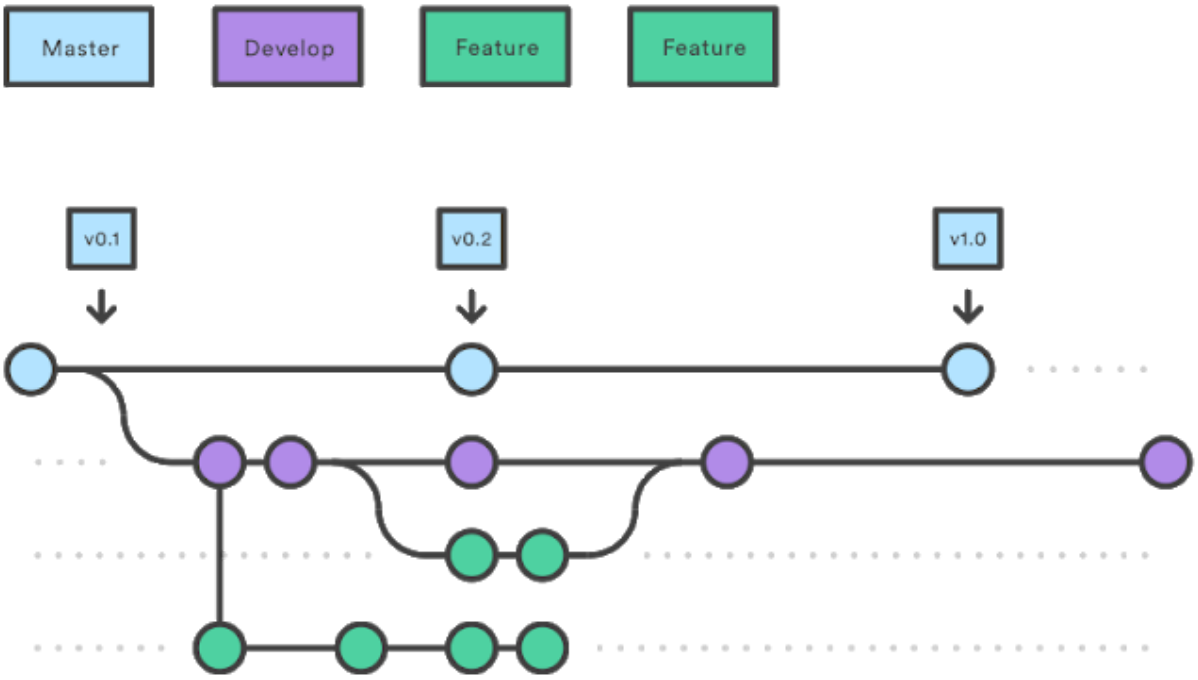
Guía de colaboración en Github

Introducción a Git y GitFlow



Git es un software para colaborar y versionar el desarrollo de código ampliamente usado en la industria y academia.

Git se basa en la idea de “ramas de desarrollo”. Típicamente, en la producción de software suele haber una versión del código que es la que se le presenta al usuario final. Se entiende que para llegar a esa instancia el producto debe contar con features completamente funcionales y testeados, para que, justamente, el usuario no se encuentre con nada que no funcione (como un botón que no responde) o que el usuario se tope con errores de código (por ejemplo, donde hace click y sale un cartel de error).

Si bien hay distintos patrones de manejo de Git y generalmente cada empresa lo adapta a sus necesidades el más conocido es GitFlow[1].

Si bien no hace falta extenderse sobre todas las etapas en GitFlow existen 5 tipos de ramas de desarrollo:

- Feature branches: donde cada developer desarrolla el feature en el que se está trabajando

[1] Acá pueden ver una buena explicación del flujo completo de GitFlow https://jeffkreeftmeijer.com/git-flow/

- Develop: donde se “juntan” los cambios en los nuevos features en los que cada programador o programadora trabaja.
- Release: donde se preparan los cambios antes de llegar al usuario final. Generalmente en esta instancia se toma el tiempo necesario para testear los features y ver cómo funcionan en consonancia con el desarrollo preexistente.
- Master: para mostrarle al usuario final el producto se usa por convención la rama master, lo que hay aquí es lo que el usuario final va a ver.
- Hotfix: si algo falla en producción (es decir, en el código que está en master) podemos hacer un cambio rápido usando una rama hotfix. De esta forma evitamos pasar por feature + develop + release.

En este ejemplo estamos desarrollando justamente cursos! Así que podemos simplificar el flujo anterior y colaborar de la siguiente manera:

- Feature branches: acá cada quien va a desarrollar su clase. Para ello vamos a emplear la nomenclatura: iniciales + 3 primeras letras del curso + número de clase. Por ejemplo: si quisiera preparar la clase 3 del curso introductorio a Python haría una rama lc_int_3.
- Develop: Una vez terminada nuestra clase podemos enviarla a develop y allí la revisaremos (en vez de en release)
- Master: acá estará el contenido disponible al público. Sólo llegará hasta acá el contenido revisado.

De develop a master se pide hacer un PR (pull request) con pedido de revisión. Después veremos qué significa exactamente ésto pero, dicho rápidamente, es hacer un pedido de llevar lo que hay en develop a master obligando a que alguien lo revise.

Manos a la obra

Antes que nada tenemos que asegurarnos que tenemos instalado Git[2]. Luego, debemos clonar el repositorio de interés, en este caso clonaremos institutohumai/cursos-python.
Para ello primero nos ubicamos donde queremos tener la carpeta para trabajar en el proyecto y luego ejecutamos:

```
git clone https://github.com/institutohumai/cursos-python.git
```

Al hacerlo notarán que se crea una carpeta con el nombre del proyecto conteniendo los archivos que corresponden.

Ahora, párense dentro de la carpeta y luego:

```
git status
```

[2] Se puede descargar desde la página oficial, donde también pueden encontrar tutoriales https://git-scm.com/downloads

Probablemente vean un mensaje como:

```
En la rama master
Tu rama está actualizada con 'origin/master'.

nada para hacer commit, el árbol de trabajo esta limpio
```

¿Qué es **origin**? **origin** es una forma de decirle al repositorio remoto desde donde descargamos el proyecto.
Se refiere a la url original https://github.com/institutohumai/cursos-python.git.

En términos más generales **remote** se refiere a cualquier servidor remoto donde esté hosteado el proyecto, lo contrario de local.

Ok, ahora vamos a acelerar un poco. A continuación vamos a seguir ejecutando los comandos desde la terminal pero vale remarcar que existen GUI que pueden ser muy útiles:

- En Windows y MAC se puede usar la aplicación de escritorio de Github: https://desktop.github.com/
- Desde Linux es muy recomendable la extensión de Git que provee VSCode,
- También pueden consultar esta página con muchos otros clientes. https://git-scm.com/downloads/guis

Veamos qué ramas tenemos en el proyecto:

```
git branch -a
```

Nos devolverá algo similar a:

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

En verde van a ver marcada la rama en la que está trabajando, es la rama activa. En rojo son las ramas remotas, y en blanco podrán ver las ramas locales. ¿Y develop? ¿Dónde está develop?
Para que nos aparezca localmente debemos hacer **checkout** hacia esa rama:

```
git checkout develop
Rama 'develop' configurada para hacer seguimiento a la rama remota
'develop' de 'origin'.
Cambiado a nueva rama 'develop'
```

checkout es el comando para pasar de una rama a la otra. Lean el mensaje, fíjense que dice que la rama **develop** va a seguir a la rama **develop** de **origin**. Si recordamos que origin es de donde descargamos el repositorio, lo que nos quiere decir el mensaje es que nuestra rama local develop va a "mirar" el estado de la rama develop del repositorio.
Ahora bien, ¿cómo puede estar seguro que está actualizada mi rama? Siempre que quieran traerse todos los cambios existentes en el repositorio **en la misma rama** deben, habiéndose parado (con checkout) en esa rama, ejecutar:

```
git pull origin
```

este comando buscan en origin, en la rama con el mismo nombre en la que se está, descarga los cambios y "mergea" con lo que tenemos localmente.

Ahora bien, listemos nuevamente las ramas.

```
git branch -a
```

```
* develop
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

Ahora en verde está develop, porque es donde nos ubicamos. Bueno… menos cháchara y más acción… Vamos a hacer un feature!!! Ok, digamos que vamos a hacer una clase para el curso Introducción. Primero, vamos a crearnos una rama para nosotros solos, ¿por qué haríamos eso? Bueno, básicamente si lo hacemos sobre develop, puede generarse un conflicto si dos personas editan el mismo archivo al mismo tiempo. Como las clases van a usar Jupyter Notebook esto sería muy propenso a ocurrir ya que el archivo crudo de las notebooks cambia con solo volver a ejecutarlas.
Para crear una rama y ubicarnos sobre ella al mismo tiempo se suele usar git checkout con el flag -b de branch. Siguiendo la convención explicada al comienzo y suponiendo que vamos a hacer algún cambio sobre la clase 5:

```
git checkout -b lc_int_5
```

Si listan las ramas van a notar que hay una nueva rama local que no existe en el remote.
Ahora vayamos a Introduccion/5_Poo_Proyecto y agreguemos un archivo test.txt.

Una vez hecho eso ejecutemos:

```
git status
```

Y obtenemos:

```
En la rama lc_int_5
Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será
confirmado)

      5_Poo_Proyecto/test.txt

no hay nada agregado al commit pero hay archivos sin seguimiento
presentes (usa "git add" para hacerles seguimiento)
```

Este comando nos dice si hay algún cambio en el proyecto que aún no ha sido notificado a git. Noten que lo menciona como “archivo sin seguimiento”.
git separa los cambios que nos interesa incluir en el proyecto de los que no. Los cambios “seguros” se agrupan en unidades llamadas **commits.** Un commit es básicamente una serie de cambios junto a un comentario indicando qué hicimos. Para agregar el cambio al commit debemos usar el comando git add, podemos usarlo para agregar un cambio a la vez con git add + el nombre del archivo, o usarlo para agregar todos los cambios pendientes de seguimiento empleando git add . (punto).

```
git add .
```

Veamos ahora el estado del proyecto con git status:

```
git status
En la rama lc_int_5
Cambios a ser confirmados:
  (usa "git reset HEAD <archivo>..." para sacar del área de stage)

      nuevo archivo:  5_Poo_Proyecto/test.txt
```

Noten que ahora este archivo está en verde en la lista de “Cambios a ser confirmados:”. ¿Qué falta para confirmar? Hacer el commit. Para ello vamos a ejecutar el comando con la forma:

git commit -m "acá va un mensaje explicativo"

En nuestro caso puede ser:

```
git commit -m "commit de prueba"
[lc_int_5 d0ebb5f] commit de prueba
 1 file changed, 1 insertion(+)
 create mode 100644 Introduccion/5_Poo_Proyecto/test.txt
```

Si hacen git status van a ver que no hay cambios sin seguimiento.
Bueno, ahora que ya tenemos nuestro cambio listo podemos enviarlo al repositorio remoto, a la rama lc_int_5.
Ok, así como pull sirve para traer los cambios que puedan existir, push es el comando para enviar desde mi proyecto local al remote. Para eso el comando git push origin nombre_de_rama va a buscar la rama nombre_de_rama en el remote y, si no existe, crearla, y enviar allí los cambios. Ejecutamos entonces:

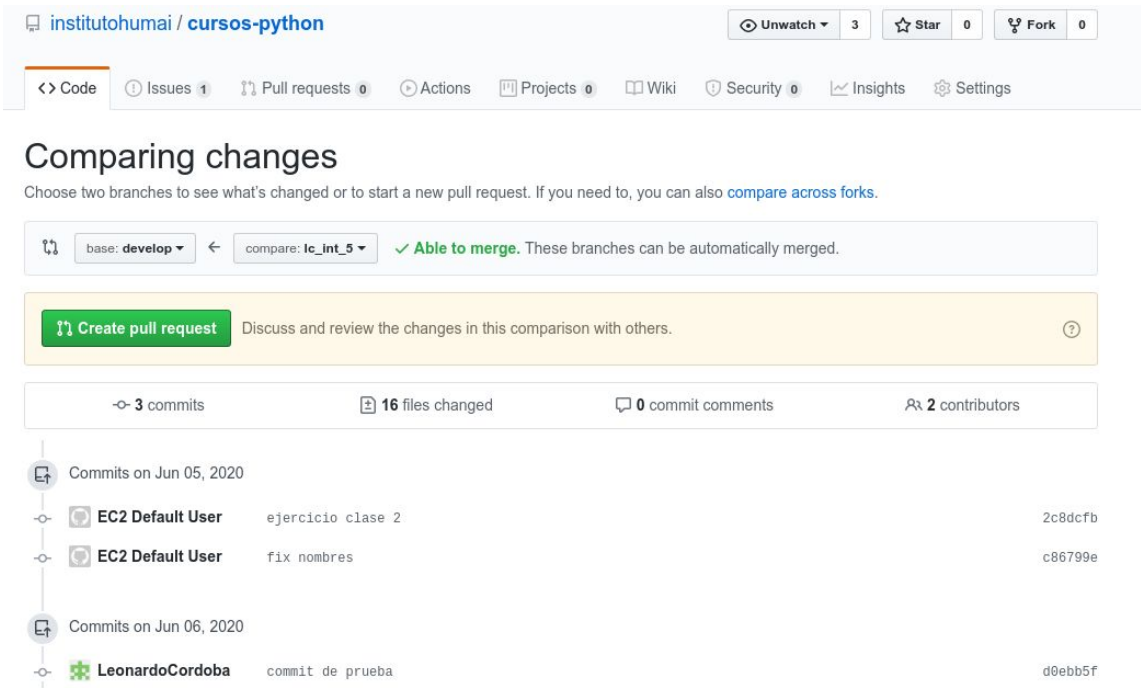
```
git push origin lc_int_5
```

Nos pide agregar las credenciales y obtenemos:

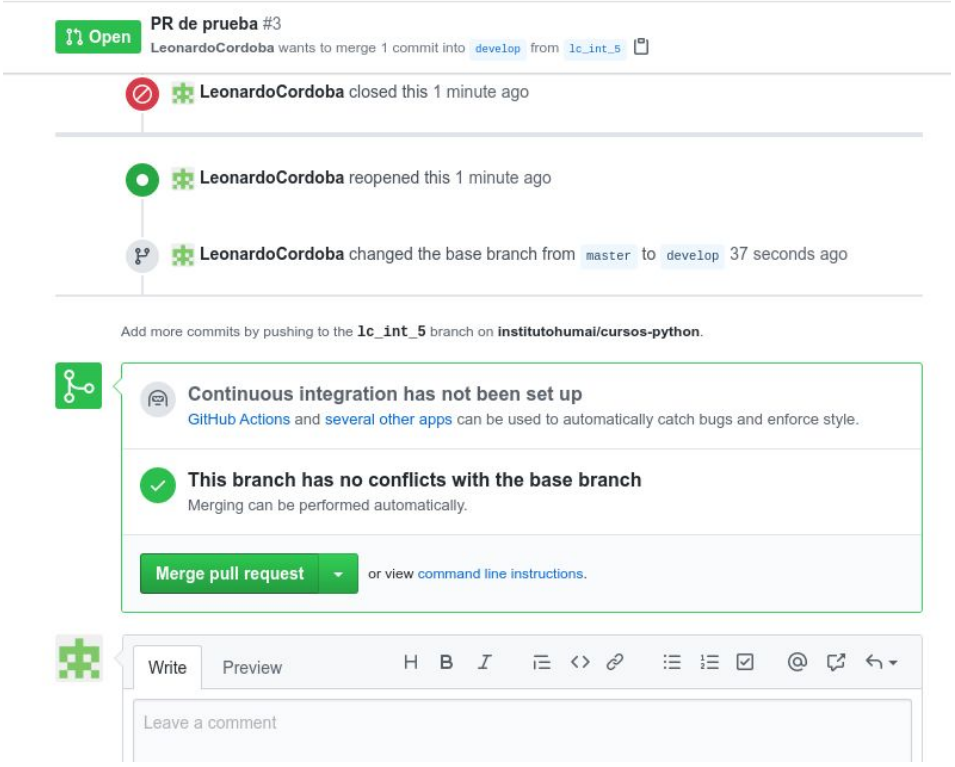
```
Contando objetos: 5, listo.
Delta compression using up to 4 threads.
Comprimiendo objetos: 100% (4/4), listo.
Escribiendo objetos: 100% (5/5), 438 bytes | 438.00 KiB/s, listo.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'lc_int_5' on GitHub by visiting:
remote:
https://github.com/institutohumai/cursos-python/pull/new/lc_int_5
remote:
To https://github.com/institutohumai/cursos-python.git
 * [new branch]      lc_int_5 -> lc_int_5
```

Ok, ya enviamos nuestro cambio a nuestra rama. Así podemos seguir desarrollando nuestra clases, cambios, etc. Una vez que nuestro trabajo esté listo debemos enviarlo a develop. Para ello vamos a github...

En la pestaña Pull Requests podemos crear un PR, que debe ir desde nuestra rama a develop. ¡Mucho cuidado con ésto! La idea es poder sumar nuestros cambios (por ejemplo, una clase) a develop, y una vez que esté todo listo (imaginemos que son muchas clases de un curso nuevo) enviarlo a master (que es lo que efectivamente se va a dictar en el curso).



El último paso es hacer merge (ojo, ¡no lo hagan con este ejemplo!):



¿Por qué hacemos primero un PR y luego un merge? En primer lugar, porque al hacer el PR Git chequea si hay conflictos y, en caso en que lo haya, nos avisará. En segundo lugar porque podemos incluir revisores, arriba a la derecha. Para enviar a develop no hacen falta revisores, pero desde develop a master sí es necesario. En este caso sugerimos que se nos incluya a LeonardoCordoba y a githubmg

Finalmente, para ver los cambios en las notebooks (ya que ver las diferencias en los JSONs es imposible) recomendamos usar https://www.reviewnb.com/, pueden probar accediendo a https://app.reviewnb.com/institutohumai/cursos-python/