

Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

1 de noviembre de 2021

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-???-???-?
© Facundo Batista & Manuel Carlevaro
Primera Edición (versión preliminar)
Escrito con X₃LaTeX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](#)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: <https://github.com/facundobatista/libro-pyciencia>

10 9 8 7 6 5 4 3 2 1

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científicos, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos" que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Prefacio	2
I Herramientas fundamentales	4
1. Versionado de código	5
1.1. Ramas	6
1.2. Git	7
1.3. Comenzando con Git	8
II Temas específicos	13
III Apéndices	14
A. Zen de Python	15

Parte I

Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Versionado de código

En un universo ideal, cuando queremos construir un proyecto de software, escribimos los sistemas de punta a punta, sin errores, los terminamos, funcionan perfecto, y no hay nada más que hacer.

Los universos ideales no existen.

En realidad, escribimos y reescribimos muchas veces cada programa, cada documento, cada archivo. Los creamos, los renombramos, quedan obsoletos y los borramos. Tenemos ideas que apuntan para un lado y las exploramos, luego nos damos cuenta que la decisión no era la más acertada, a veces corregimos un poco, a veces cambiamos totalmente la dirección.

Muchas veces, nos damos cuenta luego de modificar fuertemente un archivo, o incluso luego de borrarlo, que el estado anterior del mismo no estaba tan mal, y queremos volver a eso. Las primeras veces que nos pasa no tenemos alternativa, ya perdimos "lo viejo", pero luego empezamos a tratar de mantener algunas versiones de los archivos, antes de modificarlos, y empezamos generar copias con numeros o fechas en los nombres.

Y si a la situación le agregamos que podemos tener colaboradores en el proyecto, modificando algunos archivos sin que nos enteremos, o incluso modificando las mismas líneas de los mismos archivos que nosotros, todo se vuelve inmanejable.

Entran los Sistemas de Control de Versiones.

Un sistema de control de versiones es una herramienta que nos permite administrar los cambios que realizamos a los archivos de nuestros proyectos (no importa qué tipo de archivos sean), registrando qué cambios hizo qué persona en qué archivo, contribuyendo incluso a minimizar los conflictos por cambios simultáneos a las mismas líneas de un archivo.

En detalle, nos permite:

- grabar el estado del proyecto en un determinado momento ("sacar una foto" del estado de todos los archivos), pudiendo indicar anotaciones con respecto a ese estado
- revisar y comparar esos estados a través del tiempo, e incluso obtener el contenido de los archivos en esos distintos momentos
- trabajar en distintas ramas" de nuestro proyecto (manteniendo distintos cambios en paralelo, sin cruzarse); explicaremos este concepto más adelante [1.1](#)
- colaborar múltiples personas en el proyecto de forma distribuida

Notemos que algunas de estas características son provistas sólo en los sistemas modernos de versionado de código, como Git, Bazaar, o Mercurial. Sistemas más viejos, ya en desuso (Subversion, SourceSafe, CVS), no proveen todas estas características.



Aplicación	Versión
git	2.27.0

Debemos hacer énfasis en que es muy importante tener nuestros proyectos bajo algún sistema de versionado de código, incluso si trabajamos sólo nosotros en el proyecto. ¿Qué sistema elegir? Depende muchas veces del contexto y del grupo de trabajo, hoy en día el más difundido es Git, del cual hablaremos en detalle luego 1.2.

1.1. Ramas

¿Qué es una “rama” en este contexto? Vamos a enfocarnos un poco en este concepto, que es central para entender los sistemas de control de versiones modernos (pero al mismo tiempo tratando de no entrar en las especificidades de ninguna herramienta en particular).

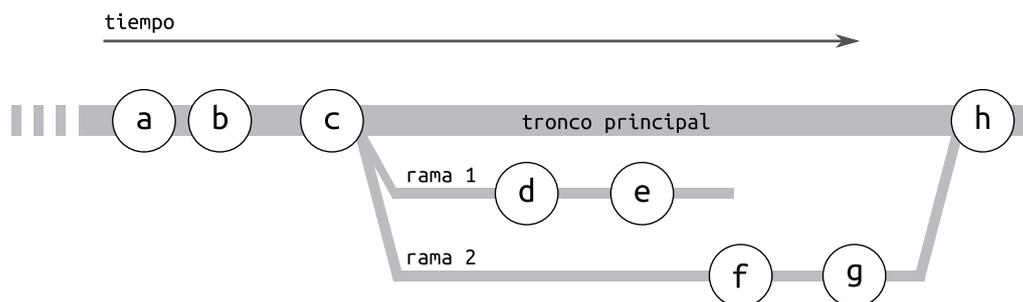
Es una técnica de desarrollo que surgió a partir de usar los sistemas de control de versiones, ya que sin ellos es imposible administrar los cambios. Nos permite explorar distintos caminos evolutivos de nuestro sistema de forma simple e intuitiva.

Supongamos que tenemos un proyecto con una veintena de archivos, estamos contentos con el estado actual, pero tenemos que implementar un cambio para el cual tenemos dos ideas distintas de cómo hacerlo. Decidimos ir por la primera idea y empezamos a implementarla, pero luego de unos días de trabajo nos damos cuenta que no es el mejor camino y decidimos probar la segunda idea.

El concepto de “rama” nos permite gestionar eso de forma muy limpia.

Tenemos que pensar en la evolución principal de nuestro proyecto como un “tronco” del cual salen estas ramas. En el momento en que queremos probar la primer idea, creamos una rama que bifurca el estado del proyecto. Trabajamos en esa rama, grabando los estados en distintos momentos, hasta que como dijimos antes, nos damos cuenta que no es lo que queremos. Ahí volvemos al estado del proyecto como estaba en el tronco principal, efectivamente “deshaciendo” todos los cambios que hicimos en la rama (pero no los perdemos, ¡quedan allí guardados!), y abrimos una segunda rama. Trabajamos en esta segunda rama por varios días, guardando los distintos estados en distintos momentos, hasta que terminamos el cambio. En ese momento integramos estos cambios en el tronco principal, y allí ya podemos pensar en el próximo cambio (el cual implementaremos en otra rama, y así...).

Veamos este proceso representado en el siguiente gráfico:



Allí vemos que en el tronco principal de desarrollo habíamos guardado los estados a, b y c. Luego, decidimos abrir la primer rama, desarrollamos, guardamos los estados d y e, pero decidimos abandonar esa línea de desarrollo. Entonces volvemos a abrir una rama desde el mismo punto anterior, guardamos los estados f y g, y como nos parece que está todo terminado correctamente, juntamos esa rama con el tronco principal, y grabamos el estado h antes de ponernos a trabajar con el siguiente cambio.

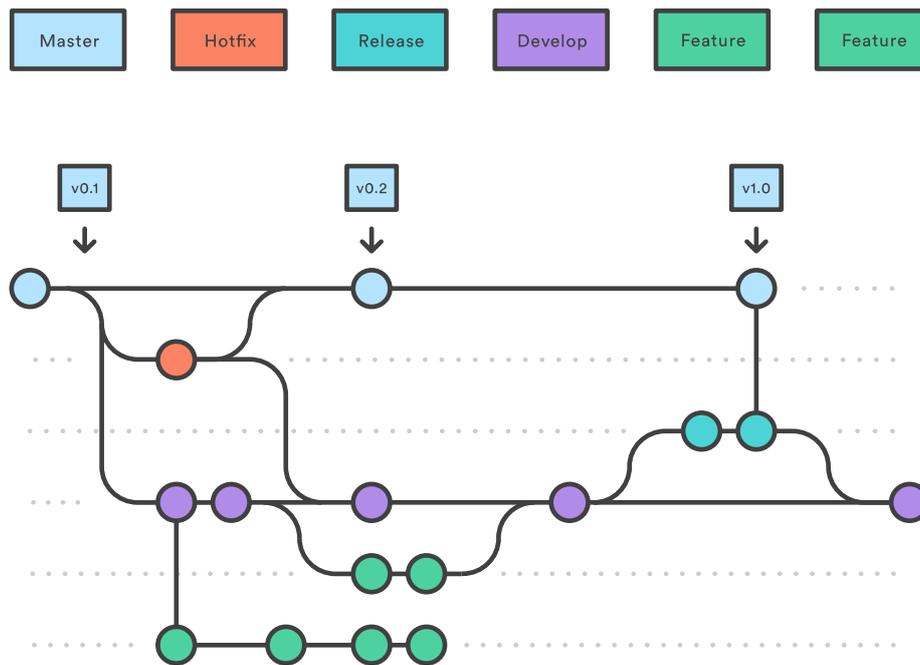


FIGURA 1.1: Flujo recomendado por Atlassian para trabajar con git [2]

1.2. Git

Git es el sistema de versionado de código más difundido en la actualidad.

Pensado originalmente para manejar proyectos distribuidos muy grandes, tuvo una gran difusión gracias al sitio Github que permite almacenar y gestionar los distintos proyectos, aunque hoy en día la mayoría de los sitios de manejo de proyectos (Launchpad, Gitlab, SourceForge, etc.) soportan git como mecanismo.

Aunque git es un sistema puramente distribuido, en la práctica el desarrollo se centraliza en una o más rama principales en algunos de esos sitios, definidas en función de cada proyecto.

Las bibliografías oficiales de Git detallan un mecanismo de trabajo llamado “gitflow”, donde recomiendan tener muchas ramas con funciones específicas, como se muestra en el siguiente gráfico:

Sin embargo en nuestra opinión y experiencia ese flujo de trabajo es demasiado complejo y se justifica solamente en proyectos y equipos muy grandes. En la gran mayoría de los casos es suficiente con un modelo mucho más simple de entender y ejecutar en el día a día, basado en las siguientes premisas:

- tenemos una rama principal, que siempre es “correcta”; es buena idea que no todos los colaboradores puedan escribir en esa rama principal, y la mayoría de los sitios de administración de proyectos proveen herramientas para que sólo se escriba allí luego de ejecutar pruebas y verificaciones que podemos definir por proyecto.
- de esa rama principal sacamos ramas de trabajo para cada cambio que necesitemos hacer en el proyecto, y a esa rama principal volvemos con código terminado, revisado y correcto.
- marcamos puntos temporales en esa rama principal usando etiquetas (por ejemplo, cuando liberamos o ponemos en servicio el código)

Hablamos sobre la dinámica de trabajo con ramas arriba, cuando las presentamos, pero entremos más en detalle sobre cómo utilizarlas en función del modelo que estamos explicando ahora.

Entonces, el “ciclo de vida” de una rama es:

- la creamos desde la rama principal para comenzar a trabajar en algún cambio (una nueva característica, solucionar algún problema, etc.)
- trabajamos en el código durante un tiempo (un rato, un día, una semana), muy posiblemente grabando el estado del código en varios momentos durante ese trabajo
- eventualmente, consideramos que el trabajo está hecho, entonces proponemos esa rama para que otros desarrolladores la evalúen y nos den feedback; mientras esto sucede podemos sacar una nueva rama desde la principal y comenzar otra tarea
- si el feedback que nos dan sobre nuestro trabajo implica que tenemos que realizar cambios en el mismo, volvemos a esa rama y hacemos las modificaciones necesarias y volvemos a proponer nuestro trabajo para que sea revisado
- en algún momento el equipo decidirá que el trabajo está completo de forma satisfactoria, entonces llevaremos todas las modificaciones de esa rama a la principal

En verdad, todo este flujo de trabajo es independiente de la herramienta. Se puede lograr con `git`, como explicaremos abajo, pero también con `bzr` (Bazaar), `hg` (Mercurial), o cualquier sistema de control de versiones más o menos moderno que permita trabajar con ramas de forma decente.

1.3. Comenzando con Git

En esta sección mostraremos lo básico de la utilización de `git` para lograr el flujo descrito arriba (y más, porque son operaciones genéricas que podemos combinar como querramos).

No es un tutorial completo sobre la herramienta, ni tampoco entra en detalles específicos sobre las estructuras internas de `git`. Para una utilización más avanzada de este sistema de control de versiones se deberán aprender otras opciones de la misma, y entender cómo funciona internamente en más detalle, pero creemos que dejando de lado muchas de esas especificidades permitimos un acercamiento más pedagógico a la utilización por primera vez de un control de versiones, sea `git` o el elegido en nuestro grupo de trabajo.

Entonces, el objetivo es bajar la barrera de entrada lo más posible para que todos podamos aprovechar esta herramienta, ya habrá tiempo para profundizar.

El primer paso es arrancar con un proyecto. Si el mismo ya está creado en otro lado, podemos clonarlo directamente:

```
$ git clone <la url que corresponda del proyecto>
```

Por ejemplo, para clonar el proyecto de este mismo libro, se puede hacer:

```
$ git clone https://github.com/facundobatista/libro-pyciencia.git
```

Si estamos arrancando un proyecto desde cero, a veces es mejor ir al sitio de administración que elijamos (Github, Launchpad, etc.), crear un repositorio vacío allí, y clonarlo, pero también podemos arrancar uno nuevo en nuestra máquina local:

```
$ mkdir miproyecto
$ cd miproyecto
$ git init -b trunk
```

En este procedimiento apareció una nueva palabra: “repositorio”. Un repositorio es el lugar donde el sistema de control de versiones almacena todos nuestros archivos más un montón de información extra para poder darnos las funcionalidades intrínsecas al control de versiones.

También cuando creamos el repositorio en nuestra máquina le dijimos a git que la rama principal se va a llamar `trunk`. Históricamente las ramas principales de Git se llamaron siempre `master`, pero en los últimos años surgieron y se afianzaron una serie de movimientos en el mundo que apuntan a evitar referenciar desigualdades históricas, principalmente raciales, reflejadas en muchos términos que se usan en la informática en el día a día, y en función de esto Github por ejemplo empezó a llamar `main` a la rama principal de un nuevo proyecto. Tanto “`main`” (*principal*) como “`trunk`” (*tronco*) son alternativas interesantes, porque en un caso hace referencia a que es la rama principal, y en el otro es de donde salen las otras ramas.

Sólo para arrancar, creemos un archivo en nuestro nuevo proyecto. Con algún editor de texto grabemos un archivo `saludo.txt` con algún contenido. Luego de eso, si hacemos `git status` veremos que Git nos reporta que ese archivo está “sin seguimiento”, o sea que no está siendo administrado/controlado por el sistema de versionado.

Le indicamos entonces a Git que queremos este archivo en nuestro proyecto (lo cual sólo tenemos que hacer la primera vez que creamos o traemos un archivo al proyecto):

```
$ git add saludo.txt
```

Ahora `git status` nos mostrará que tenemos un nuevo archivo, pero con cambios a ser confirmados. Es hora de usar uno de las órdenes de Git más utilizadas, y que es la responsable de “grabar el estado” de nuestro proyecto, cómo explicábamos arriba:

```
$ git commit -m "Archivo de prueba."
```

El argumento `-m` nos permite adjuntar un mensaje al `commit` que acabamos de realizar. Si omitimos esa opción, Git tratará de abrir un editor de texto para que escribamos dicho mensaje, lo cual funcionará o no ya dependiendo de cómo tenemos configurado nuestro sistema (si tenemos un editor por default, por ejemplo, y si el mismo se puede utilizar de la manera que Git lo está intentando usar).

Un nuevo `git status` nos mostrará que “el árbol de trabajo está limpio”. Estamos listos para agregar nuevos cambios a nuestro proyecto.

Un detalle importante que omitimos mencionar hasta ahora es que no estamos trabajando realmente con ramas como explicamos arriba, sino que hicimos el `commit` directamente a la rama principal. No está del todo mal, pero cuando el proyecto empieza a crecer en complejidad (y queremos explorar distintos cambios al mismo tiempo) o tenemos un equipo de trabajo con varios colaboradores, si se trabaja de esta manera realmente se pierden muchas de las funcionalidades que el sistema de control de versiones tiene para ofrecernos.

Entonces, creemos una nueva rama, y posicionémosnos para trabajar en ella:

```
$ git branch cambio-1
$ git checkout cambio-1
```

Podemos validar que estamos en dicha rama con `git status`. Ahora hagamos un cambio en nuestro único archivo (cambiémosle el contenido). Para grabar el nuevo estado del proyecto, primero deberíamos especificarle a git cuales archivos cambiados queremos registrar (con `add`) y luego efectivamente registrarlos (con `commit`), pero podemos hacer ambas acciones directamente:

```
$ git commit -am "Nuevo contenido de saludo."
```

Notemos que ahora al `commit` le pasamos dos opciones: `a` para agregar los cambios (por default, todos ellos), y `m` para el mensaje, que está escrito a continuación.

Realicemos el proceso de cambiar el archivo y volver a `commit` los cambios. Luego, para ver todo el historial de trabajo, usamos `git log`:

```
$ git log
commit 5a5f0369b89a6e5b0689f13d55671669c35ce9e4 (HEAD -> cambio-1)
Author: Facundo Batista <facundo@supermail.org>
Date: Mon Feb 8 17:05:01 2021 -0300
```

Más cambios en el saludo.

```
commit 4804a799ddaa714e1f9e4f301f2a668364e4e908
Author: Facundo Batista <facundo@supermail.org>
Date: Mon Feb 8 16:56:34 2021 -0300
```

Nuevo contenido de saludo.

```
commit a543a7406bca6ac08cbfda2bb291a7f25ec7cbc4 (trunk)
Author: Facundo Batista <facundo@supermail.org>
Date: Thu Feb 4 20:24:21 2021 -0300
```

Archivo de prueba.

Como vemos, tenemos un bloque de información por cada *commit* que hicimos. Ese bloque de información tiene dos partes, un encabezado con información del *commit* y el mensaje que nosotros incluimos (que en estos casos es corto, pero podrían ser varias líneas o párrafos).

Ese encabezado también tiene varias partes. El autor y la fecha son obvios, enfoquémonos en la primer línea: vemos que arranca con `commit` y luego hay una serie de letras y números: ese es el *hash* del `commit`, un identificador único que nos permite trabajar sobre ese `commit` o hacer referencia al mismo.

Finalmente, tenemos entre paréntesis el dato sobre las ramas y donde estamos parados. Vemos que el primer `commit` que hicimos originalmente en “trunk” menciona esa rama entre paréntesis: esto es porque la rama en la que estamos parados ahora “salió desde ese punto” de la rama de donde nació (en este caso, justamente, “trunk”, pero piensen que se puede sacar una rama desde cualquier otra rama). En el último `commit`, además de la indicación de que es el último `commit` de la rama `cambio-1`, vemos que dice `HEAD`: es la indicación de que estamos viendo el estado del proyecto en ese `commit`.

Hagamos la prueba de movernos de `commit` para explorar el pasado:

```
$ git checkout 4804a799ddaa714e1f9e4f301f2a668364e4e908
```

Al ejecutar eso recibimos un mensaje de Git diciéndonos que estamos en modo “detached HEAD” (HEAD desacoplada), que significa que no estamos actualmente dentro de ninguna rama, sino en ese `commit` específico. Podemos ver eso mismo haciendo `git log`, e incluso revisar el archivo de nuestro proyecto y ver que su contenido es el que “pertenece al pasado”.

Podemos volver a la rama en que estábamos haciendo `git checkout cambio-1`.

Y en este punto es donde dejamos de repetir que se puede utilizar tanto `git log` como `git status` para ver y entender cómo los distintos comandos afectan nuestro trabajo en el proyecto.

Ahora, siguiendo la idea del ejemplo original del capítulo, supongamos que nos arrepentimos de explorar este cambio en nuestro proyecto (digamos que entendimos que estuvo mal cambiar el contenido de `saludo.txt`, debíamos agregar un segundo archivo). Para arrancar limpios (y dejar intacto todo lo trabajado hasta ahora, que nos puede servir en el futuro), volvemos a `trunk` y arrancamos una nueva rama desde allí.

```
$ git checkout trunk
$ git checkout -b cambio-2
```

Atención al detalle de que, en lugar de crear la rama y después cambiar a ella, hacemos las dos operaciones en un sólo paso utilizando la opción `-b` del comando `checkout`.

Podemos ver las ramas que tenemos creadas en nuestro proyecto (Git nos indicará con un asterisco en cual estamos parados en este momento):

```
$ git branch
  cambio-1
* cambio-2
  trunk
```

Ahora podemos agregar otro archivo, incorporarlo al proyecto, hacer todos nuestros cambios, `commit`ear varias veces, revisar cómo nos queda, hasta estar contentos con nuestro trabajo. Una mímica de todo este proceso sería lo siguiente:

```
$ echo "hola hola" > saludo-extra.txt
$ git add saludo-extra.txt
$ git commit -am "Segundo archivo"
[cambio-2 1f779d7] Segundo archivo
 1 file changed, 1 insertion(+)
 create mode 100644 saludo-extra.txt
$ echo "hola hola chau" > saludo-extra.txt
$ git commit -am "Con despedida"
[cambio-2 df39361] Con despedida
 1 file changed, 1 insertion(+), 1 deletion(-)
$ echo ";hola!" > saludo.txt
$ git commit -am "Énfasis en el saludo original"
[cambio-2 524dd0a] Énfasis en el saludo original
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log
commit 524dd0afca1525f57658f81a0304e64112bbe977 (HEAD -> cambio-2)
Author: Facundo Batista <facundo@supermail.org>
Date:   Tue Feb 9 11:30:12 2021 -0300
```

Énfasis en el saludo original

```
commit df3936143e018d4e95d8149eddab9e29593765a7
Author: Facundo Batista <facundo@supermail.org>
Date:   Tue Feb 9 11:29:25 2021 -0300
```

Con despedida

```
commit 1f779d77cb748e7b3f95091f60ad9b50996233ac
Author: Facundo Batista <facundo@supermail.org>
Date:   Tue Feb 9 11:28:59 2021 -0300
```

Segundo archivo

```
commit a543a7406bca6ac08cbfda2bb291a7f25ec7cbc4 (trunk)
Author: Facundo Batista <facundo@supermail.org>
Date:   Thu Feb 4 20:24:21 2021 -0300
```

Archivo de prueba.

```
$
```

Vemos que esta segunda rama existe “en paralelo” con la otra, sin mezclarse la serie de cambios en cada caso.

¿Y ahora? Una vez que terminamos con nuestro cambio, es hora de llevarlo a la rama principal. Siempre es buena idea revisar cuál es la diferencia entre lo que tenemos y la rama a la que llevaremos los cambios:

```
$ git diff trunk
diff --git a/saludo-extra.txt b/saludo-extra.txt
new file mode 100644
index 0000000..e12be5d
--- /dev/null
+++ b/saludo-extra.txt
@@ -0,0 +1 @@
+holo hola chau
diff --git a/saludo.txt b/saludo.txt
index 5c1b149..8d1966f 100644
--- a/saludo.txt
+++ b/saludo.txt
@@ -1 +1 @@
-hola
+hola!
```

Es hora de llevar todo esto a la rama principal. En verdad no “llevamos” los cambios, sino que los “traemos”, por lo que primero tenemos que pararnos donde queremos los cambios, y luego indicamos que queremos “combinar” a la otra rama.

```
$ git checkout trunk
$ git merge cambio-2
```

Listo, tenemos todos los commits de cambio-2 también en trunk. Debemos mencionar que no siempre el “merge” funciona tan limpio, porque podemos tener conflictos, que aparecen cuando en distintas ramas se modificaron zonas similares de algún archivo.

Si en una rama tocamos un archivo y en la otra rama tocamos otro archivo, no hay problema. Incluso si tocamos el mismo archivo en las dos ramas, pero en zonas lo suficientemente separadas, tampoco es un inconveniente. Pero si en las distintas ramas modificamos las mismas líneas del mismo archivo, Git no podrá combinar ese archivo, y tendremos un conflicto que habrá que resolver.

Esta y muchas otras situaciones se pueden presentar en el día a día, ya que el manejo central de un sistema de control de versiones es bastante directo y entendible, pero tiene muchas puntas afiladas y zonas oscuras que implicará un aprendizaje continuo al usar la herramienta, al menos al principio. Lo importante es que arranquemos a utilizarla con los comandos aprendidos en este capítulo, el resto irá viniendo solo.

Para profundizar les dejamos dos guías en sistemas de control de versiones en general y git en particular: el Git Handbook [3] (parte de las Github Guides) es interesante y cortito, pero en inglés, y el tutorial de Atlassian [4] es muy completo, entra en muchos detalles y esta en castellano, pero no trabaja con Github sino con Bitbucket, lo cual puede ser confuso en algún punto, pero interesante para aprender variado.

Parte II

Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [5].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tenemos más de esos!

Bibliografía

- [1] URL: <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.
- [2] URL: <https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>.
- [3] URL: <https://guides.github.com/introduction/git-handbook/>.
- [4] URL: <https://www.atlassian.com/es/git>.
- [5] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.