Temas Selectos de Análisis Numérico y Computación Científica: Computo científico para el análisis de datos

Haydeé Peruyero

2024-03-01

Contents

1 Temas Selectos de Análisis Numérico y Computación Científica: Computo científico para el análisis de datos **7**

1.1 Temario 7

1.2 Referencias 8

1.3 Material interesante 8

1.4 DataCamp 8

2 Shell **11**

2.1 Navegar en archivos y directorios 11

2.2 Manipulación de archivos y directorios 14

2.3 Tuberías y filtros 18

2.4 Ciclos 21

2.5 Scripts 27

2.6 Buscando y encontrando cosas 31

2.7 if, while y for 34

2.8 Descarga y limpieza de bases de datos 37

3 Git y Github **39**

3.1 Repositorios 40

3.2 Rastrear cambios 41

3.3 Explorando el historial 45

3.4 Restore y reset 48

3.5 Ignorar archivos/carpetas 51

3.6 Github 54
3.7 Colaboradores 60
3.8 Conflictos 64
3.9 Trabajando con Ramas 68
3.10 Conectar con overleaf 70

4 Python **73**
4.1 Tipos de datos 73
4.2 Flujo de control 73
4.3 Visualización de datos 73
4.4 Manipulación de bases de datos 73
4.5 Análisis exploratorio de bases de datos 73
4.6 Funciones y scripts 73
4.7 Buenas practicas 73
4.8 Procesamiento de alto rendimiento 73
4.9 Programación en paralelo 73

5 SQL **75**
5.1 Bases de datos y manipulación 75
5.2 Explorar datos categóricos y texto no estructurado 75
5.3 Comparación con los otros programas 75
5.4 Valores faltantes 75
5.5 Combinar bases de datos 75

6 Power BI **77**
6.1 Introducción a Power BI 77
6.2 Transformando y visualizando datos 77
6.3 Manipulación de bases de datos 77
6.4 Análisis exploratorio de bases de datos 77
6.5 Variables categóricas y continuas 77

7 R **79**

7.1 Tipos de datos 79

7.2 Manipulación de bases de datos 79

7.3 Análisis exploratorio de bases de datos 79

7.4 Reportes con RMarkdown 79

7.5 Páginas web 79

Chapter 1

Temas Selectos de Análisis Numérico y Computación Científica: Computo científico para el análisis de datos

Curso del posgrado conjunto en Ciencias Matemáticas PCCM UNAM UMICH
2024-2

1.1 Temario

1. Git y Github
2. Shell
3. Python
4. SQL
5. Power BI
6. R
7. Estadística multivariada
8. Análisis de regresión

1.2 Referencias

[1] Arnold, Jeremey. Learning Microsoft Power BI, O'Reilly Media, Inc.

[2] Beaulieu, Alan. Learning SQL, O'Reilly Media, Inc., 2020

[3] Bruce, Peter, Bruce, Andrew and Gedeck, Peter. Practical Statistics for Data Scientists, O'Reilly Media, Inc., 2020.

[4] Crawley, Michael J. The R book. John Wiley & Sons, 2012.

[5] McKinney, Wes. Python for data analysis. O'Reilly Media, Inc., 2022.

[6] Nelli, Fabio. Python Data Analytics, Apress.

[7] Wade, Ryan. Advanced Analytics in Power BI with R and Python, Apress.

[8] Wickham, Hadley, and Garrett Grolemund. R for data science: import, tidy, transform, visualize, and model data. O Reilly Media, Inc., 2016.

[9] Zamora Saiz, Alfonso, et al. An Introduction to Data Analysis in R: Hands-on Coding, Data Mining, Visualization and Statistics from Scratch., Springer (2020).

[10] Software Carpentry, The Unix Shell, https://swcarpentry.github.io/shell-novice/

1.3 Material interesante

- Bookdown.
- Software Carpentry.
- Git
- Why Git
- R Markdown Cookbook
- STHDA
- YaRrr! The Pirate's Guide to R
- Learn ggplot2 Using Shiny App
- Ggplot2: Elegant Graphics for Data Analysis
  - Versión online
- Use R! Colección Springer
- Lattice: Multivariate Data Visualization with R
- R Graphics cookbook
- Cuenta pro de Github

1.4 DataCamp

Figure 1.1: DataCamp

Chapter 2

Shell

Descargar git bash para Windows o seguir las instrucciones de Software carpentries para otros sistemas operativos. Basado en la lección de The carpentries

2.1 Navegar en archivos y directorios

Cuando abrimos una terminal por primera vez, vamos a ver un **prompt** (usualmente es `$`) que nos indica que esta esperando los comandos. Después de teclear los comandos, debemos siempre presionar Enter.

```
$
```

Para listar lo que hay en un directorio usamos el comando `ls`.

```
$ ls
```

```
Documents  Downloads  Music  Pictures  Videos
```

Al comando `ls` le podemos agregar unos `adjetivos` para hacer más comprensible su lectura. Por ejemplo, la opción `-F` nos indica si es una carpeta, un archivo, un link, etc.

```
$ ls -F
```

```
ej.txt  Git/  PBI/  Python/  R/  Shell/  SQL/
```

Con la opción `--help` podemos acceder a la ayuda del comando.

```
$ ls --help
```

Otras opciones que nos ayudan a entender la información que tenemos en el archivo son `-lh`, nos muestra los permisos del archivo o carpeta, tamaño, propietario, fecha, nombre. La opción `ls -a` nos muestra los archivos ocultos.

Ejercicio: ¿Qué hace la opción `-l`? ¿Cómo podemos listar en orden de creación e inverso?

Con el comando `ls` también podemos listar los archivos de cualquier otro directorio, solo debemos indicarle el directorio después del comando:

```
$ ls - F Shell
```

El comando `ls` solo nos esta listando lo que hay en los directorios. Si nos queremos mover a otro directorio, lo podemos hacer con el comando `cd` y especificando el directorio.

```
$ cd Shell
$ ls
```

```
data/  ej1.txt  ejercicios/
```

El comando `pwd` nos da la ruta en la que estamos.

```
$ pwd
```

```
/d/Users/hayde/Documents/Curso_Comp_Cien/Shell
```

Para movernos a un directorio arriba, colocamos después de `cd` dos puntos.

```
$ cd ..
```

Si no colocamos los dos puntos, el comando `cd` nos lleva al `/home/`.

```
$ cd
```

Otra forma de movernos de un directorio a otro es especificando la *ruta absoluta*, es decir la ruta completa de a donde queremos movernos.

```
$ cd /d/Users/hayde/Documents/Curso_comp_Cien
```

Ejercicio: ¿Qué es una ruta relativa?

Otro opción útil que podemos usar con el comando `cd` es `-`.

Ejercicio: ¿A donde nos lleva `cd -`? y si volvemos a colocar `cd -` ¿a donde nos lleva? ¿Cuál es la diferencia entre `cd ..` y `cd -`?

La tilde `~`, shell la interpreta como el home del usuario, entonces si colocamos `cd ~/directorio` sería lo mismo que `/home/directorio`. Por ejemplo:

```
$ cd ~/Desktop
```

es lo mismo que

```
$ cd /c/Users/hayde/Desktop
```

Ejercicio: Supongamos que tenemos el siguiente árbol de datos en nuestra computadora y que estamos en `/Users/thing/`. ¿Si colocamos en la terminal `ls -F ../backup` que nos mostrará?

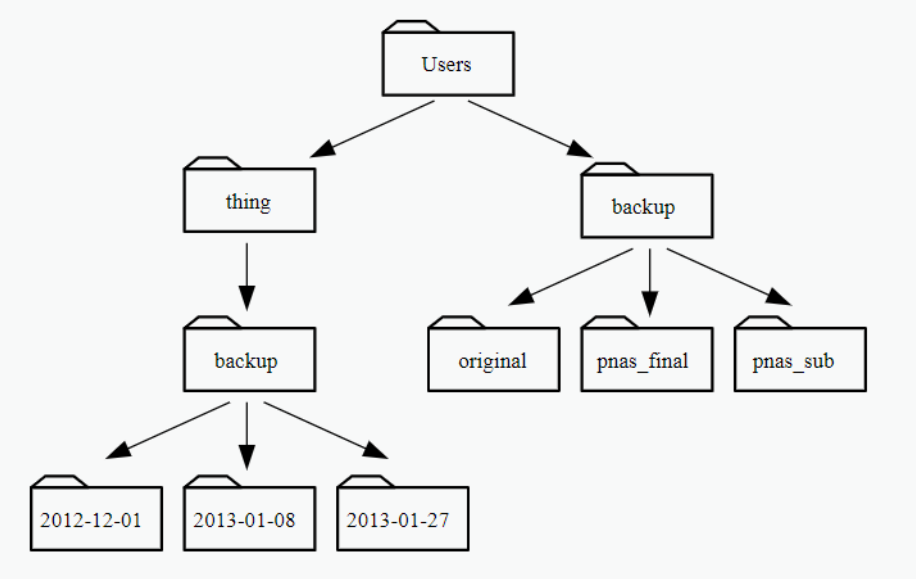


Figure 2.1: Ejercicio SC

Ejercicio: Explora las opciones `-s` y `-S`. ¿Hay diferencia entre mayúsculas y minúsculas?

La tecla Tab nos ayuda a completar los comando/rutas. Si la presionamos dos veces nos mostrará todas las posibles opciones.

2.2 Manipulación de archivos y directorios

Para crear directorios/carpetas desde la línea de comandos usamos el comando `mkdir nombre`. Vamos a crear la carpeta del curso. Es recomendable **no usar espacios en nombres de carpetas ni archivos**.

```
$ mkdir Curso_Comp_Cien
```

El comando `mkdir` nos permite crear más de un directorio y directorios anidados usando la opción `-p`.

```
$ mkdir -p Curso_Comp_Cien/Shell/data Curso_Comp_Cien/Shell/ejercicios
```

Para listar toda la estructura de nuestras carpetas podemos usar la opción `-FR` en el comando `ls`.

```
$ ls -FR
```

Otra opción útil para listar toda la estructura de nuestro directorio es la opción `tree`, no viene instalado por default en los sistemas operativos. En Windows, si se instalo git/bash se puede usar con `tree.com`.

```
$ tree.com
```

Nota: Para descargar `tree` y que podamos ver la estructura de árbol con archivos y carpetas hacer lo siguiente:

1) Ir a la página y descargar la versión que dice *binaries*.
2) Extraer lo que hay en la carpeta **bin**.
3) Copiar el archivo **tree.exe** a la carpeta `/c/Program_files/Git/usr/bin/`.
4) Para probarlo, solo colocar en la terminal `tree ruta`.

Para crear un archivo usando el editor nano (si fue el que configuraron), se usa el comando `nano nombre.extension`. Vamos a crear un archivo de prueba en la carpeta ejercicios y escribamos algo en el archivo.

```
$ cd Curso_Comp_Cien/
$ nano prueba.txt
```

Para guardar y salir del editor, usamos los comandos Ctrl+O o Ctrl+X seguido de `Yes` y enter.

Si en el archivo no guardaron nada, entonces no se creara. Una forma de crear archivos sin abrirlos es con el comando `touch`.

```
$ cd Shell/ejercicios
$ touch ../ej1.txt ej2.txt ../data/ej3.csv ../data/ej4.csv
```

Si checamos con `ls -l` los archivos o rutas que creamos, veremos que no tienen ningún tamaño.

Para mover archivos o renombrarlos, usamos el comando `mv` seguido del archivo que queremos mover/renombrar y la ruta a donde lo moveremos o el nuevo nombre del archivo.

```
$ pwd
$ mv ej2.txt ejercicio2.txt
```

Lo anterior esta cambiándole el nombre al archivo `ej2.txt` por `ejercicio2.txt`.

Si estamos en un directorio y queremos mover un archivo de otro directorio al directorio actual, podemos hacerlo especificando como primer argumento la ruta y nombre del archivo a mover y como segundo argumento un punto `..`

```
$ mv ../ej1.txt .
```

O especificando la ruta completa.

```
$ cd ..
$ mv data/ej3.csv ejercicios/.
```

Para copiar archivos, usamos el comando `cp` seguido por la ruta del archivo a copiar y la ruta del archivo a donde se copiará. Movámonos a la carpeta `Shell/ejercicios`.

```
$ pwd
$ cp ejercicio2.txt ../ej2.txt
$ ls -FR
```

Si usamos la opción `-r` (*recursivo*) en el comando `cp` podemos copiar un directorio completo y todos sus elementos.

```
$ cd ejercicios
$ cp -r ../data .
```

Si el último parámetro de `cp` es un directorio que ya existe, entonces copiará todos los archivos que se indiquen a ese directorio. Por ejemplo:

```
$ cp archivo1.txt archivo2.txt data
```

De igual forma para `mv`, si el último parámetro es un directorio movera todos los archivos a ese directorio.

```
$ mv archivo1.txt archivo2.txt data
```

Ejercicio: ¿Cual es el output de la siguiente colección de comandos?

```
$ pwd
```

```
/Users/haydee/Curso
```

```
$ ls -F
```

```
archivo.txt carpeta/
```

```
$ mkdir carpeta2
$ mv archivo.txt carpeta2/
$ cd carpeta2
$ cp archivo.txt ../carpeta/archivo_respaldo.txt
$ cd ..
$ ls -FR
```

Para borrar archivos usamos el comando `rm`, hay que tener cuidado cuando lo usemos ya que **borra definitivamente** los archivos o carpetas.

```
$ cd Shell
$ rm ejercicios/ejercicio2.txt
$ ls ejercicios/
```

Una forma segura de borrar archivos es usando la opción `-i`, con esto nos saldrá un mensaje preguntando si en verdad deseamos borrar el archivo. Para confirmar debemos colocar `y`.

```
$ rm -i /data/ej3.csv
```

```
rm: remove regular empty file 'ej3.txt'?
```

Si queremos borrar una carpeta, debemos hacerlo con la opción `-r`, de lo contrario obtendremos un error.

```
$ rm data
```

```
$ rm -r data
```

Otra opción para borrar directorios es `rmdir`.

Para mover/copiar/eliminar multiples archivos a la vez, podemos enumerarlos todos o usar comodines/patrones que sigan estos elementos. Supongamos que tenemos una lista de archivos todos con terminación `.txt`, entonces para borrarlos podemos usar `rm *.txt`. El `*` nos indica todo lo que este antes de `.txt`.

```
$ cd Shell/
$ touch prueba1.txt prueba2.txt prueba3.txt prueba4.txt prueba5.txt
$ rm *.txt
```

Otro comodín que podemos usar es `?`, pero este denota solo 1 espacio. Por ejemplo:

```
$ cd Shell/
$ touch prueba1.txt prueba2.txt prueba3.txt prueba4.txt prueba5.txt
$ rm prueba?.txt
```

```
$ cd Shell/
$ touch prueba1.txt prueba2.txt prueba3.txt prueba4.txt prueba5.txt
$ rm prue???.txt
```

Ejercicio: Supon que en el directorio `data` tienes dos archivos. ¿Cuál de los siguientes comandos te daría como resultado: `ethane.pdb` `methane.pdb`.

1) `ls *t*ane.pdb`
2) `ls *t?ne.*`
3) `ls *t??ne.pdb`
4) ls ethane.*

Dos comodines más que existen son los siguientes:

- `[...]`: busca coincidencias con exactamente cada caracter dentro de los corchetes, por ejemplo `[12]` coincidiria con `texto1.txt`, `texto2.tx` pero no con `texto3.txt`.
- `{...}`: busca coincidencias con cada uno de los elementos separados por comas dentro de las llaves, por ejemplo `{*.txt, *.csv}` buscaría todos los archivos con terminaciones `.txt` y `.csv` pero no con los que sean `.pdf`.

2.3 Tuberías y filtros

Vamos a usar los archivos de prueba de la lección de Shell de Software Carpentry. Descargarlos en el directorio que creamos que se llama Shell.

Vamos a explorar los archivos que están en la carpeta `excercise-data/alkanes`. Para contar cuantos palabras, líneas o caracteres tiene un archivo, usamos el comando `wc` que viene de `word count`.

```
$ ls
```

```
cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
```

```
$ wc cubane.pdb
```

```
20  156 1158 cubane.pdb
```

El primer número es el número de líneas del archivo, el segundo la cantidad de palabras y el tercero la cantidad de caracteres.

Si usamos alguna de los comodines, por ejemplo `*.pdb` con el comando `wc`, nos va a regresar la información de todos los archivos.

```
$ wc *.pdb
```

Notemos que en la última fila tenemos los totales de todos los archivos. Accedamos a la ayuda del comando con `help`.

```
$ wc --help
```

Ejercicio: ¿Cuál opción nos permite extraer solo la cantidad de líneas del archivo?

```
$ wc -l *.pdb
```

Si por error olvidamos colocar el nombre del archivo o cualquier otra cosa después del comando, la consola se quedará esperando una instrucción, para salir de esto basta presionar Ctrl+C.

Ya sabemos como extraer cierta información de nuestros archivos, pero supongamos que queremos guardarlo ahora en algún otro archivo para después analizarlo. El símbolo `>` redirige el resultado de los comandos usados a algún archivo.

```
$ wc -l *.pdb > lineas.txt
```

Para solo visualizar el contenido de un archivo sin entrar al editor de texto, podemos usar el comando `cat` seguido del nombre del archivo.

```
$ cat lineas.txt
```

Otro comando que puede resultar más útil para mostrar el contenido de un archivo es `less`, la diferencia con `cat` es que este último muestra todo el contenido en la pantalla, lo cual puede dificultar su lectura e inspección, mientras que `less` muestra una parte del contenido y de forma ordenada, si queremos seguir viendo el contenido podemos usar la tecla de espacio, b y para salir usamos la letra `q`.

Ya guardamos la información de la cantidad de líneas, pero supongamos que queremos saber cual archivo tiene la mayor cantidad de líneas o menor. Para hacer esto nos sirve el comando `sort`.

```
$ cd ..
$ sort numbers.txt
```

Si a `sort` le agregamos la opción `-n`, nos los ordena en numericamente en lugar de alfabeticamente.

```
$ sort -n numbers.txt
```

Ejercicio: De los archivos que están en la carpeta `alkane`, ¿cuál tiene la menor cantidad de líneas?

También podemos redirigir esta información a otro archivo y de ahí extraer la información.

```
$ sort -n lineas.txt > lineas_ordenadas.txt
```

El comando `head` nos ayuda a extraer las primeras `n` líneas de nuestro archivo. Por ejemplo, para extraer la primera línea del archivo `lineas_ordenadas.txt` y así saber cual archivo tenía la menor cantidad de líneas usaríamos `head -n 1`.

```
$ head -n 1 lineas_ordenadas.txt
```

El comando `echo` nos ayuda a imprimir en la consola caracteres.

```
$ echo Hola
```

Ejercicio: Realiza las siguientes instrucciones dos veces cada una. Explora las diferencias. ¿Qué hace el operador `>>`?

```
$ echo hola > test1.txt
```

```
$ echo hola >> test2.txt
```

El comando `tail` es similar al comando `head`, nos muestra las últimas `n` filas del archivo.

Ejercicio: Considera el archivo `/exercise-data/animal-counts/animals.csv`. Después de aplicar los siguientes dos comandos, ¿qué hay en el archivo `animals-subset.csv`?

```
$ head -n 3 animals.csv > animals-subset.csv
$ tail -n 2 animals.csv >> animals-subset.csv
```

2.3.1 Tuberías

Además de redirigir los output de los comandos que hemos ocupado, también podríamos anidarlos y al final mandarlo a un archivo. Para hacer esto se usan **tuberías** y su símbolo es `|`. Por ejemplo:

```
$ sort -n lineas.txt | head -n 1
```

En esta instrucción le estamos diciendo a la consola que primero nos ordene lo que hay en el archivo líneas en orden númerico y después que nos muestre la primera línea. De esta forma nos evitamos por ejemplo el haber creado el archivo `lineas_ordenadas.txt`.

Podemos anidar varias instrucciones a la vez. Por ejemplo, podríamos pedirle a la consola la cantidad de líneas de los archivos `*.pdb`, pedirle que las ordene numericamente y después que extraiga la primera línea.

```
$ wc -l *.pdb | sort -n | head -n 1
```

Ejercicio: De los archivos que están en la carpeta `alkanes`, obten los 3 archivos con la menor cantidad de líneas.

Ejercicio: Explora el archivo `exercise-data/animals-counts/animals.csv`. ¿Cuál será el resultado de la siguiente instrucción?

```
$ cat animals.csv | head -n 5 | tail -n 3 | sort -r > final.txt
```

El comando `cut` nos ayuda a extrar/cortar ciertas columnas de nuestros archivos. Por ejemplo, `cut -d , -f 2 archivo` nos está indicando que del archivo queremos cortar por caracteres `,` (eso hace `-d ,`) y que queremos extraer la segunda columna (`-f 2`).

```
$ cut -d , -f 2 animals.csv
```

Si quisieramos extraer los animales únicos de ese archivo, podemos usar el comando `uniq`.

```
$ cut -d , -f 2 animals.csv | sort | uniq
```

También se puede colocar por ejemplo `-f 2-5,8` para indicar que se deben seleccionar las columnas 2 a la 5 y la 8.

Ejercicio: ¿Porqué se necesita colocar el `sort` antes del `uniq`?

Ejercicio: Si quisiéramos ver cuantos animales hay de cada tipo, ¿que instrucción tendríamos que usar?

2.4 Ciclos

Los ciclos nos ayudan a repetir comandos o un conjunto de comandos para cada elemento de una lista. La estructura del ciclo `for` es como sigue:

```
for elemento in lista
do
  operacion/comando $elemento
done
```

La palabra `for` indica el comienzo del ciclo, la palabra `do` nos indica que es lo que se va a ejecutar y su comienzo y la palabra `done` indica el fin del ciclo.

Exploremos lo que hay en la carpeta `~/Shell/shell-lesson-data/exercise-data/creatures`. Listemos las primeras 5 filas de cada archivo.

```
$ head -n 5 basilisk.dat minotaur.dat unicorn.dat
```

Supongamos que queremos ver la clasificación de cada especie que se encuentra en la segunda línea de cada archivo. Una forma de hacerlo es con `head -n 2`, pero de esta forma también estamos viendo su nombre común, entonces vamos a hacerlo con un ciclo. Lo primero que tendríamos que hacer es usar justo `head -n 2` y al resultado de esto, si le pedimos la última línea ya solo veríamos la clasificación, entonces usamos un `tail -n 1`.

```
$ for filename in *.dat
> do
>   echo $filename
>   head -n 2 $filename | tail -n 1
> done
```

Notemos que cuando empezamos a teclear nuestro ciclo, el prompt cambia de `$` a `>`, esto indica que está esperando que continuemos el ciclo. También podemos usar `;` para continuar las instrucciones en una misma fila.

Dentro de los ciclos, las variables las mandamos a llamar con `$`, en el ejemplo, cuando ocupamos `$filename` estamos mandando a llamar la variable `filename` que definimos al inicio del ciclo. Es muy usual también encerrar entre llaves los nombres de las variables para delimitar el nombre, en el ejemplo, `$filename` sería equivalente a `${filename}`.

Ejercicio: Crea un ciclo que muestre en pantalla (echo) todos los números del 0 al 9.

Ejercicio: Ve a la carpeta `shell-lesson-data/exercise-data/alkanes` y lista lo que hay. 1) ¿Cuál es el output del siguiente código?

```
$ for datafile in *.pdb
> do
>     ls *.pdb
> done
```

2) ¿Y de este código?

```
$ for datafile in *.odb
> do
>     ls $datafile
> done
```

Explica las diferencias.

Ejercicio: En el directorio `shell-lesson-data/exercise-data/alkanes`, ¿cuál sería el output del siguiente código?

```
$ for filename in c*
> do
>     ls $filename
> done
```

Y si en lugar de `c*` usamos *c*?

Dentro de un ciclo también podemos pedir guardar archivos.

Ejercicio: Explora el siguiente código. ¿Cuál es el efecto de guardar en este ciclo?

```
$ for alkanes in *.pdb
> do
>     echo $alkanes
>     cat $alkanes > alkanes.pdb
> done
```

¿Cuál sería la diferencia si usamos ahora `>>`?

Ejercicio: Crea un ciclo que muestre las últimas 20 líneas de cada archivo en la carpeta `creatures`.

Como ya mencionamos, no es recomendable usar espacios ni caracteres especiales en nombres de archivos o carpetas. Si fuera el caso de que nuestros archivos tienen espacios, entonces deberíamos pasarlos al ciclo for encerrados entre comillas los nombres. Por ejemplo, supongamos que tenemos los archivos `archivo 1.txt` y `archivo 2.txt`. Para leerlos en el ciclo for tendríamos que usar la siguiente sintaxis.

```
$ for filename in "archivo 1.txt" "archivo 2.txt"
> do
>     head -n20 "$filename" | tail -n5
> done
```

Supongamos que queremos modificar nuestros archivos que se encuentran en la carpeta `creatures` pero que antes queremos respaldarlos en otros archivos llamados `original-basilisk.dat`, `original-unicorn.dat` y `original-minotaur.dat`. Una forma de hacerlo sería copiarlos a nuevos archivos con esos nombres, pero si no lo queremos hacer manualmente, ¿qué pasa si usamos el siguiente código?

```
$ cp *.dat original-*.dat
```

Esto nos va a dar un error porque el comando `cp` estaría esperando que `original-` sea una carpeta y no lo es, no existe esa carpeta. Entonces lo que podemos hacer es usar un ciclo.

Ejercicio: Crea un ciclo `for` que copie los dos archivos a dos nuevos archivos llamados `original-basilisk.dat`, `original-unicorn.dat` y `original-minotaur.dat`.

El comando `cp` no nos muestra en pantalla ningún output. Si quisieramos ver que en efecto se esta realizando la copia de estos archivos podemos usar el comando `echo` y pedir que nos diga como "se copio el archivo $filename". Usar `echo` de esta forma es una buena práctica de realizar lo que se llama **debugging**.

```
$ for filename in *.dat
> do
>     echo cp $filename original-$filename
> done
```

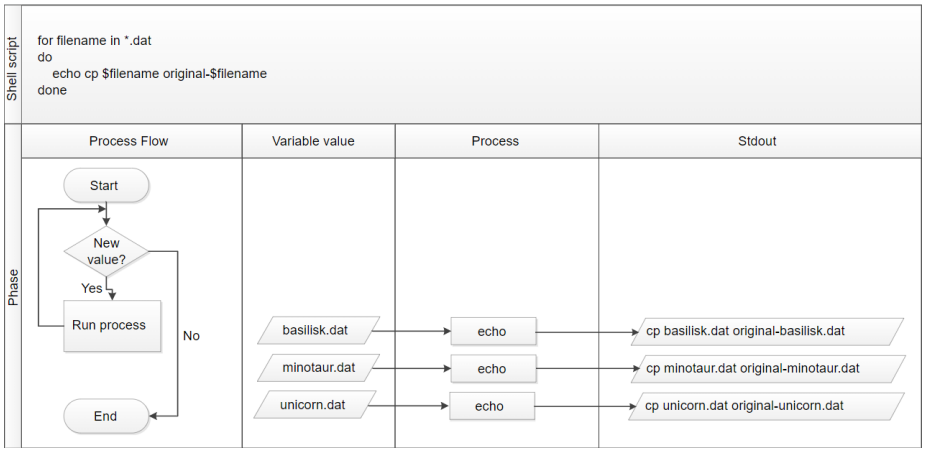


Figure 2.2: Debugging

Ejercicio: Supongamos que queremos previsualizar los comandos que el siguiente ciclo va a realizar en lugar de correrlo primero para asegurarnos de que está haciendo lo que queremos.

```
$ for filename in *.pdb
> do
>     cat $filename >> all.pdb
> done
```

¿Cuál de los siguientes dos códigos sería el correcto para revisar los comando a ejecutarse con el ciclo?

```
# Versión 1
$ for filename in *.pdb
> do
>     echo cat $filename >> all.pdb
> done
```

```
# Versión 2
$ for filename in *.pdb
> do
>    echo "cat $filename >> all.pdb"
> done
```

Corre los dos códigos y explora el contenido del archivo `all.pdb`.

Supongamos que queremos crear una estructura de directorios como sigue, para cada compuesto y cada temperatura queremos una carpeta para ir guardando ahí sus resultados, y que cada carpeta se llame `compuesto-temperatura`, ¿cómo podemos hacer esto? Una opción son los ciclos anidados.

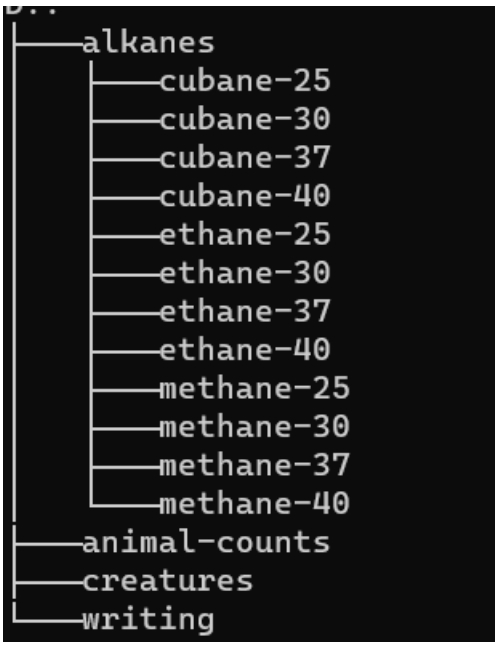


Figure 2.3: estructura-experimentos

```
$ for species in cubane ethane methane
> do
>     for temperature in 25 30 37 40
>     do
>         mkdir $species-$temperature
>     done
> done
```

Algunos comandos útiles para ver el historial de comadnos.

- `history` nos muestra el historial de comandos.
- Ctrl+R nos muestra la leyenda `reverse-i-search`, esto indica que está esperando que nosotros coloquemos una palabra y buscará por el último comando con esa palabra.
- `history` + `!123` nos repetirá el comando `!123` del historial.
- `!!` nos muestra el último comando usado.
- `!$` nos regresa la última palabra del último comando.

Ejercicio: En la carpeta `norht-pacific-gyre` se encuentran dos scripts (`.sh`) y una lista de archivos. Esta lista de archivos tiene terminaciones A, B y en el caso de que la terminación sea Z significa que el archivo está corrupto.

1) ¿Cómo podrías darte cuenta que los archivos con terminación Z están corruptos?

Supongamos que queremos ejecutar el script llamado `goostats.sh`, este script necesita recibir dos cosas, el archivo de entrada y el nombre del archivo de salida. Supongamos que queremos correr este script para todos los archivos con terminación A y B y que queremos que los archivos de salida se llamen `stats-$datafile`.

2) Crea un ciclo que te muestre en pantalla el nombre del archivo a usar como input.
3) Crea un ciclo que te muestre en pantalla el nombre del archivo de salida con el formato indicado. Pero quieres asegurarte que para el archivo input sea el nombre correcto el del archivo de salida.
4) Crea un ciclo que muestre los comandos a usarse para correr el script con los archivos de entrada y de salida del paso 2 y 3. Para correr un script como se indica, se usa el comando `bash nombre_archivo.sh input output`.
5) Agrega un `echo $datafile` para saber en que archivo va tu ciclo.

2.5 Scripts

Los scripts nos ayudan a repetir los comandos sobre listas de archivos. Supongamos que existen ciertos comando que siempre repetimos, vamos a guardarlos en un archivo para con un solo comando ejecutar esa lista de comandos.

Vamos al directorio `alkanes`. Supongamos que siempre queremos extraer las líneas de la 11 a la 15 de cada archivo. Por ejemplo, una forma en la que lo hacemos es:

```
$ head -n 15 octane.pdb | tail -n 5
```

Vamos a escribir eso en un archivo:

```
$ cd alkanes
$ nano middle.sh
```

Guardemos eso. Para ejecutarlo bastaría correr lo siguiente:

```
$ bash middle.sh
```

Supongamos que queremos las líneas de la 11 a la 15 pero de cualquier otro archivo. Vamos a modificar el archivo que creamos.

```
$ nano middle.sh
```

```
head -n 15 "$1" | tail -n 5
```

En el script, cuando colocamos `"$1"` se refiere al primer argumento/archivo en la línea de comandos, por ejemplo cuando colocamos en la línea de comandos:

```
$ bash middle.sh octane.pdb
```

Lo que estamos diciéndole a la consola es que reemplace dentro del script "$1" por el archivo `octane.pdb`. De esta forma nuestro script ahora lo podemos correr sobre cualquier archivo.

Nuestro script por el momento funciona solo con las líneas de la 11 a la 15. Supongamos que queremos modificar esto de tal forma que cuando vayamos a ejecutar el script le indiquemos las líneas que queremos extraer. Así como usamos `$1` para indicarle que era la primera variable en la línea de comandos, podemos usar las variables `$2` y `$3` para indicarle la segunda y tercera variable.

```
$ nano middle.sh
```

```
head -n "$2" "$1" | tail -n "$3"
```

Entonces podemos ejecutar el script como sigue:

```
$ bash middle.sh octane.pdb 15 5
```

Y podemos cambiar las líneas a mostrar, por ejemplo:

```
$ bash middle.sh octane.pdb 20 5
```

Lo único que falta en el script, es describir que hace, de esta forma cualquier otra persona (o nosotros más adelante), cuando queramos abrir el script podamos recordar y entender que argumentos pide y cual es su uso.

```
$ nano middle.sh
```

```
# Selecciona líneas intermedias de un archivo.
# Uso: bash middle.sh nombre_archivo linea_final linea_inicial
head -n "$2" "$1" | tail -n "$3"
```

Ahora, supongamos que queremos ordenar los archivos `.pdb` por cantidad de líneas. Sin un script eso lo hacemos así:

```
$ wc -l *.pdb | sort -n
```

Si queremos poner esto en un script pero queremos correrlo sobre varios tipos de archivos, digamos los `.pdb` y los `.dat`, no podemos colocar en nuestro script `*.pdb`, y si usamos como en los ejemplos anteriores `"$1"` o `"$2`, eso limitaría la cantidad de archivos que podemos pasarle después en la consola. Una forma de no depender de eso es con la variable `$@`, esto indica que pueden ser cualquier cantidad de argumentos en la línea de comandos.

```
$ nano sorted.sh
```

```
# Ordena archivos por su longitud
# Uso: bash sorted.sh uno_o_mas_archivos
wc -l "$@" | sort -n
```

```
$ bash sorted.sh *.pdb ../creatures/*.dat
```

Ejercicio: El archivo `animals.csv` ya vimos que es un archivo separado por comas que indica las especies y la cantidad de cada uno. Crea un script que se pueda aplicar a cualquier cantidad de archivos con ese formato y que te diga las especies únicas de cada archivo. Crea 3 archivos similares al `animals.csv` (copia y modifica) y prueba tu script.

Ejercicio: Corre el siguiente comando:

```
$ history | tail -n 5 > recientes.sh
```

¿Qué contiene ese archivo? ¿Observa la última línea del archivo? ¿Porqué guarda esa línea?

Ejercicio: En la carpeta `alkanes` supongamos que tenemos un `script.sh` que contiene lo siguiente:

```
$ head -n $2 $1
$ tail -n $3 $1
```

Dentro del directorio `alkanes`, corre lo siguiente:

```
$ bash script.sh '*.pdb' 1 1
```

¿Qué esperas obtener?

Ejercicio: Crea un script llamado `longest.sh` que reciba como argumentos un directorio y una extensión de archivos y que te devuelva el archivo en el directorio, que tenga esa extensión, con el mayor número de líneas.

Ejercicio: Considera los archivos que están en la carpeta `alkanes`. Explica que hace cada uno de los siguientes scripts al correrlos como `bash script1.sh *.pdb`, `bash script2.sh *.pdb` y `bash script3.sh *.pdb`.

```
# Script 1
echo *.*
```

```
# Script 2
for filename in $1 $2 $3
do
  cat $filename
done
```

```
# Script 3
echo $@.pdb
```

Ejercicio: (Debugging) Supongamos que tienen el siguiente script `do-errors.sh` en la carpeta `north-pacific-gyre`:

```
# Calcular estadisticas para los archivos
for datafile in "$@"
do
  echo $datafile
  bash goostats.sh $datfile stats-$datafile
done
```

Corre en la línea de comandos:

```
$ bash do-errors.sh NENE*A.txt NENE*B.txt
```

No muestra ninguna salida. Para ver porque, vamos a correrlo de nuevo con la opción -x:

```
$ bash -x do-errors.sh NENE*A.txt NENE*B.txt
```

¿Cuál es el output? ¿Cuál es la línea responsable del error?

Otra asignación de variables es `#@`, esto nos indica cuantos objetos hay de la variable.

Tres conceptos usados en bash son los siguientes:

- STDIN: estándar input
- STDOUT: estándar output
- STDERR: estándar error

Para redirigir el error y el output automáticamente a archivos, usamos las opciones:

```
$ 2> error.txt & 1> output.txt
```

O dentro de un script:

```
exec 1>> output.txt 2>> error.txt
```

Al inicio de un script se suele colocar `#!/usr/bash` para que el interprete sepa que es un script de bash y use el bash que se encuentra en `usr/bash`. Si no está ahí se coloca la ruta correspondiente, para saber donde está usamos `which bash`.

2.6 Buscando y encontrando cosas

La función `grep` (*global/regular expression/print*) nos ayuda a encontrar e imprimir líneas de archivos que contengan un patrón especificado.

Vamos a la ruta `excercise-data/writing`.

```
$ cat haiku.txt
```

Para buscar las líneas que contienen la palabra `not`, la instrucción sería la siguiente:

```
$ grep not haiku.txt
```

El comando `grep` es sensible a mayúsculas y minúsculas. Si buscáramos por ejemplo `Not` no nos encontraría ninguna coincidencia.

```
$ grep The haiku.txt
```

Observemos que ahora nos está regresando una palabra que contiene `The` en su estructura: `Thesis`. Para restringir a que solo sean coincidencias exactas, usamos la bandera `-w`.

```
$ grep -w The haiuku.txt
```

En el caso de querer buscar frases, debemos encerrarlas entre comillas.

```
$ grep -w "is not" haiku.txt
```

Si agregamos la opción `-n`, esto nos mostrará también la línea en la que se encuentra la coincidencia.

```
$ grep -n "it" haiku.txt
```

Las banderas se pueden combinar, por ejemplo `-nw` (`-wn` o `-n -w`) nos buscaría coincidencias exactas y los números de líneas.

```
$ grep -nw "it" haiku.txt
```

Para que no nos importe mayúsculas o minúsculas, usamos la opción `-i`.

```
$ grep -n -w -i "the" haiku.txt
```

La opción `-v` es para invertir nuestra búsqueda, es decir para mostrarnos las líneas que no contienen esa palabra/frase.

```
$ grep -n -w -v "the" haiku.txt
```

La opción `-r` busca recursivamente por la coincidencia en un conjunto de archivos indicados.

```
$ grep -r Yesterday .
```

Ejercicio: ¿Cómo obtendrían solo lo siguiente del archivo `haiku.txt`?

```
and the presence of absence.
```

Dentro del comando `grep` también podemos usar comodines (expresiones regulares), por ejemplo:

```
$ grep -E "^.o" haiku.txt
```

El `^` se refiere al inicio de la línea, el `.` se refiere a un carácter (análogo a `?`), entonces estaría buscando todas las líneas una palabra donde su segunda letra sea la letra `o`.

Ejercicio: El archivo que se encuentra en la carpeta `animal-counts/animals.csv` contiene una lista de animales, con su fecha de observación y cuantos animales se observaron.

```
2012-11-05,deer,5
2012-11-05,rabbit,22
2012-11-05,raccoon,7
2012-11-06,rabbit,19
2012-11-06,deer,2
2012-11-06,fox,4
2012-11-07,rabbit,16
2012-11-07,bear,1
```

Supongamos que queremos crear un script que tome como primer argumento la especie del animal y como segundo argumento el directorio. El script nos debe regresar un archivo llamado `<especie>.txt` que contenga una lista de fechas y el número de veces que se observo esa especie. Por ejemplo, `rabbit.txt` tendría que contener la siguiente información:

```
2012-11-05,22
2012-11-06,19
2012-11-07,16
```

Usa las opciones de ayuda de los comandos `cut` y `grep` (puedes usar `man grep` o `man cut` también para pedir ayuda de esos comandos, la palabra `man` se refiere a manual.)

Ejercicio: En la carpeta `exercise-data/writing` se encuentra el texto completo de Mujercitas `LittleWomen.txt`. Usando un `for` encuentra que hermana aparece más veces: *Jo*, *Meg*, *Beth*, *Amy*.

No solo podemos buscar una sola palabra, podríamos buscar dos palabras o buscar entre dos palabras:

```
$ grep -E "cadena1|cadena2" archivo
```

Ejercicio: ¿Cómo podrías mostrar en color lo que estás buscando? Explora la ayuda de `grep`.

Para buscar entre directorios, usamos el comando `find`. Por ejemplo, el siguiente comando nos encontrará todo lo que este en directorio actual

```
$ find .
```

La opción `-type d` nos mostrará solo carpetas.

```
$ find -type -d
```

Y con la opción `-type f` nos mostrará todos los archivos.

Si queremos encontrar algo que concuerde con algún nombre:

```
$ find . -name *.csv
```

Hay que tener cuidado con esta instrucción, `find` expande los comodines antes de correr los comandos, en el caso de tener más de un archivo veremos un mensaje de error. Para prevenir esto y que si busque todos los archivos, debemos encerrar el patrón entre comillas.

```
$ find . -name "*.txt"
```

Podemos hacer tuberías también con este comando, por ejemplo si quisieramos contar cuantas líneas tienen todos los archivos con terminación `txt`, podríamos hacer lo siguiente:

```
$ wc -l $(find . -name "*.txt")
```

Otro ejemplo:

```
$ grep "searching" $(find . -name "*.txt")
```

Ejercicio: La opción `-v` en `grep` busca todo lo que no concuerde con el patrón indicado. En la carpeta `creatures`, ¿cómo listarías todos los archivos que terminen en `.dat` menos el que se llama `unicorn`?

2.7 if, while y for

Ya vimos que la sintaxis del ciclo for es como sigue:

```
$ for file in list
> do
>     comands
> done
```

Una forma de crear una lista es como sigue:

```
$ {start..stop..increment}
```

Usando esta sintaxis, el código para listar todos los números del uno al 9 quedaría:

```
$ for numero in {1..9..1}
> do
>     echo $numero
> done
```

Otra forma de escribir la lista sería con la sintaxis `((x=1;x<=9;x+=1))`.

La estructura del ciclo `if` es como sigue:

```
$ if [ condicion ]; then
>     code
> else
>     other code
> fi
```

También se puede usar la estructura `((condicion))`.

Los operadores aritméticos en el condicional `if` que se pueden usar son los siguientes:

- >, <, ==, ! =
- `-eq`, `-ne`: igual y no igual a (equal to, not equal to).
- `-lt`, `-le`: para menor que o menor o igual que (less than,less than or equal to).
- `-gt`, `-ge`: para mayor que o mayor o igual que (greater than, greater than or equal to).

```
$ x=10
$ if [ $x -gt 5 ]; then
>     echo "$x es mayor que 5"
> fi
```

Otros banderas para los condicionales las pueden encontrar en el siguiente link.

La notación `&&` es para un **Y** y la notación `||` para un **O**, las cuales nos sirven para verificar más de una condición:

```
$ x=8
$ if [ $x -gt 5 ] && [ $x -lt 10]; then
>     echo "$x es mayor que 5 y menor que 10"
> fi
```

También podemos usar la siguiente estructura: `[[ $x -gt 5 && $x -lt 10]]`.

Dentro de un `if` podemos pedir que nos busque en algún archivo combinandolo con `grep`, dentro de la carpeta animal-count:

```
$ if grep -q "rabbit" animals.csv; then
>     echo "Rabbit esta en el archivo"
> fi
```

El mismo resultado lo podemos obtener como sigue:

```
$ if $(grep -q "rabbit" animals.csv); then
>     echo "rabbit esta en el archivo"
> fi
```

El ciclo `while` tiene la misma estructura del ciclo `for` y los operadores aritméticos son los mismos que en el `if`. Siempre recuerden poner un fin al ciclo para que no sea infinito.

```
$ x=1
$ while [ $x -le 5 ];
> do
>     echo $x
>     ((x+=1))
> done
```

Aparte de estos ciclos, existen los **CASE Statements**, los cuales pueden llegar a ser más útiles que los `if` cuando se tienen condiciones muy complicadas. Su estructura es la siguiente:

```
$ case 'String' in
>     patron1)
>     comando1;;
>     patron2)
>     comando2;;
>     *)
>     comando por default;;
> esac
```

Por ejemplo:

```
#!/bin/bash
#Control de flujo: case
echo "Escribe una frase"
read frase
case $frase in
  a*)
    echo "La frase empieza con a"
  ;;
  c*t)
    echo "La frase empieza con c y termina con t"
  ;;
  *com)
    echo "La frase termina con la cadena com"
  ;;
  *)
    echo "La frase no cumple con ninguna de las condiciones"
  ;;
esac
```

Si corremos este script en la terminal con `bash case_ejemplo.sh` veremos que dependiendo de la frase que coloquemos en la terminal, nos arrojará una de las opciones.

Ejercicio: Crea un case statement para adivinar tu edad. Debes pedirle al usuario que introduzca el número correspondiente a tu edad y que los casos o patrones obtengan por resultado una frase referente a si adivinaron o no su edad. Realiza lo mismo con un `if`.

2.8 Descarga y limpieza de bases de datos

Pendiente: Se verá después de SQL.

Chapter 3

Git y Github

Usar control de versiones es una forma de manejar proyectos, todo a lo que se hace `commit` se pierde, se queda un registro de todos los cambios y siempre es posible regresar a una versión anterior. Nos evitamos estar enviando y enviando correos con versiones finales para después comparar versiones. Se guarda el usuario que hizo el cambio y automáticamente obtenemos una notificación de si intentamos modificar lo mismo que un colaborador para revisar cual cambio guardar.

Pueden pensar en versión de control como una forma de `undo` ilimitado y de trabajar paralelamente con sus colaboradores.

Lo primero que vamos a hacer es configurar `Git` en nuestra computadora.

Vamos a abrir `Git bash` y configurar nuestro usuario y correo con la que vamos a enlazar más adelante Github.

```
$ git config --global user.name "Usuario"
$ git config --global user.email "email@domain.com"
```

Ahora, vamos a configurar los saltos de línea para no tener conflicto según el sistema operativo.

```
# Mac o Linux
$ git config --global core.autocrlf input
# Windows
$ git config --global core.autocrlf true
```

Para configurar el editor de texto por default:

```
$ git config --global core.editor "nano -w"
```

Por default, Git inicializa un repositorio con una rama llamada `master`, a partir del 2020, la mayoría de los servidores de Git cambiaron esto a que la rama principal fuera `main`, para configurar esto usaremos lo siguiente:

```
$ git config --global init.defaultBranch main
```

Los comandos anteriores solo se necesitan configurar una sola vez. Para ver la configuración que acabamos de realizar y probar cual es nuestro editor de texto usamos lo siguiente:

```
$ git config --global --edit
```

Y para revisar esta configuración sin entrar al editor:

```
$ git config --list
```

Si debieran hacer cambios en su usuario o correo o cualquier otra configuración lo pueden hacer ilimitadas veces con los comandos anteriores.

Para pedir ayuda nos sirve aún `git comando -h` o `git comando --help`, por ejemplo:

```
$ git config -h
# La siguiente nos abre en un navegador el manual completo
$ git config --help
```

O para ayuda general de Git: `git help`.

3.1 Repositorios

Un repositorio es donde se va a almacenar toda la información de nuestro proyecto, es donde vamos a tener toda la historia y registro de cambios y usuarios. Es recomendable tener un repositorio por proyecto y no multiples proyectos en un solo repositorio.

Vamos a movernos a la carpeta del curso y vamos a hacer una carpeta para trabajar con git.

```
$ cd Curso_Comp_cient
$ mkdir Mi_primer_repo
$ cd Mi_primer_repo
```

Para inicializar un repositorio usamos lo siguiente (dentro de la carperta).

```
$ git init
```

Al inicializar el repositorio, cualquier carpeta y archivo que se cree dentro de la carpeta quedará su registro, no es necesario inicializar las carpetas anidadas.

Si revisamos que tiene la carpeta solo con `ls` no vamos a notar ningún cambio pero si listamos con la opción `-a` veremos que contiene archivos ocultos. En el archivo `.git` se almacena **TODA** la información de nuestro repositorio, así que si lo borramos perderemos todo el historial del repositorio.

```
$ ls -a
```

Para cambiar manualmente la rama de nuestro repositorio si no es la main, lo podemos hacer como sigue.

```
$ git checkout -b main
```

Para preguntarle a git el estado de nuestro proyecto:

```
$ git status
```

Si dentro de una carpeta preguntamos `git status` y obtenemos el siguiente mensaje:

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

significa que si podemos inicializarlo como un repositorio.

Ejercicio: Dentro de la carpeta `Mi_primer_repo` crea una carpeta llamada `subproyecto1`. Si quieres llevar un registro de lo que hagas en ese subproyecto, ¿debes inicializarla? Inicializala. Ahora, ¿cómo borras el archivo `.git`?

3.2 Rastrear cambios

Vamos a crear un archivo de texto dentro de la carpeta `Mi_primer_repo`.

```
$ nano prueba.txt
```

Y escribamos algo en el archivo y guardemoslo.

```
Primer archivo en el que rastrearemos cambios.
```

Ahora, si preguntamos por el estado de nuestro proyecto vamos a obtener un mensaje de que hay algo nuevo:

```
$ git status
```

El mensaje que dice `untracked files` nos indica que hay cambios y que a Git no se le ha indicado que debe registrarlos. Para añadir estos cambios hacemos lo siguiente:

```
$ git add prueba.txt
```

Si revisamos el estado del proyecto vemos que ahora un mensaje diferente, ahora solo nos indica que no se ha realizado ningún commit pero que si se tiene registro de algo que cambio.

```
$ git status
```

Para hacer un commit:

```
$ git commit -m "Comenzando archivo de prueba y registro de cambios"
```

Este comando le dice a Git que tome todo lo que se añadió y que guarde una copia permanente dentro del directorio `.git`. Cada commit tiene un identificador único. Si no especificamos el mensaje, Git abrirá un editor de texto para colocar el mensaje. Los mensajes deben de reflejar lo que se está guardando para que sean útiles en el futuro.

Si ahora verificamos el estado del proyecto veremos que nos dice que no hay nada a lo que hacer commit ya que en el paso anterior añadimos todo y no hemos realizado ningún cambio.

```
$ git status
```

Para mostrar el historial del proyecto:

```
$ git log
```

Ahora, añadamos una línea nueva al archivo `prueba.txt`

```
$ nano prueba.txt
```

```
Primer archivo en el que rastrearemos cambios.
Segunda línea de cambios para continuar con el ejemplo.
```

Si revisamos el estado veremos de nuevo que nos devuelve el mensaje de que hay archivos sin rastrear. Para comparar las diferencias del archivo usamos:

```
$ git diff
```

El signo `+` nos está indicando cuales son los cambios en el archivo nuevo.

Vamos a hacer un commit de este cambio.

```
$ git commit -m "Añadimos la segunda linea al archivo"
```

¿Qué paso? Nos esta diciendo que no hemos añadido nada al `staged area` a lo que le podamos hacer un commit, recuerden añadir todo antes de hacer commit.

```
$ git add prueba.txt
$ git commit -m "Añadimos la segunda linea al archivo"
```

Añadir todo primero al área de preparación nos permite tener un mejor control de a que le estamos haciendo commit, por ejemplo podemos añadir y hacerle commit solo al archivo donde tenemos la bibliografía y no a todo el proyecto donde hay partes no completas.

Ejercicio: Añadamos una tercera línea al archivo y verifiquemos las diferencias en los archivos, después añadamoslo al área de preparación y revicemos las diferencias. ¿Qué sucede?

```
$ nano prueba.txt
$ git diff
$ git add prueba.txt
$ git diff
```

Al añadirlo al área de preparación lo estamos añadiendo permanentemente, entonces no hay ninguna diferencia. Si queremos las diferencias con lo último a lo que se le hizo commit podemos hacer lo siguiente:

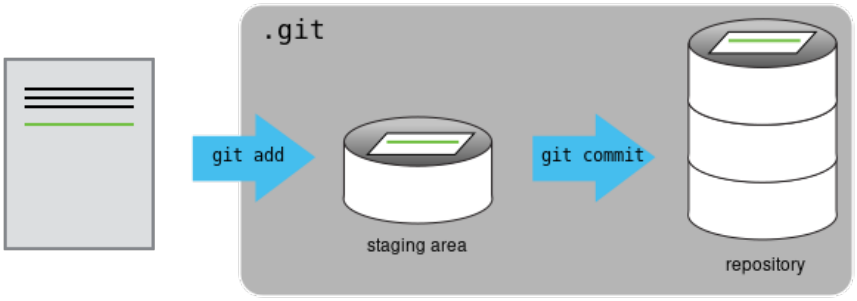


Figure 3.1: Software Carpentry - Version Control with Git

```
$ git diff --staged
```

Ahora hagamos el commit.

```
$ git commit -m "Tercera línea para probar que pasa cuando se añade al área de prepara
```

Si revisamos el historial, veremos ahora 3 commit diferentes:

```
$ git log
```

Cuando el historial es muy grande no nos va a mostrar todo el historial nuestra terminal, para ir avanzando debemos presionar Spacebar y para salir la letra Q. Al presionar / se puede buscar alguna palabra en los mensajes de los commits. Para limitar la cantidad de información que nos regresa `log` podemos especificar con la opción `-n` la cantidad de commits desde el más reciente. Para ver por ejemplo el último utilizariamos lo siguiente:

```
$ git log -1
```

Si queremos ver los mensajes en una sola línea usamos:

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

Git no guarda información de directorios vacíos. Por ejemplo:

```
$ mkdir dic_prueba
$ git status
$ git add dic_prueba
$ git status
```

Si creamos un directorio con archivos, entonces si podemos añadir todos los archivos a la vez y si quedará el registro del directorio también.

```
$ git touch dic_prueba/prueba1.txt dic_prueba/prueba2.txt dic_prueba/prueba3.txt
$ git status
$ git add dic_prueba
$ git status
$ git commit -m "Ejemplo de como realizar un registro de directorios con archivos"
```

En algunas ocasiones verán directorios vacíos con un archivo `.gitkeep`, este archivo es solo para que podamos añadir el repo a Git.

Ejercicio: Crea un archivo `mi_archivo.txt`, escribe algo en el y guárdalo en la ruta `Mi_primer_repo`. Añádelo a la historia de tu repo. ¿Cuáles son los pasos que debes realizar?

Ejercicio: Modifica el archivo `prueba.txt` añadiéndole una línea, ahora en el archivo `mi_archivo.txt` agrega algo y guárdalo. ¿Cómo añadirías los dos archivos al staging área? Añádelos y realiza el commit correspondiente.

Ejercicio: Crea un repositorio llamado `bio`. Escribe en un archivo llamado `me.txt` tres líneas de tu biografía, has un commit con tus cambios. Modifica una línea y agrega una cuarta línea. Muestra las diferencias entre el archivo en el staging área y el actual.

3.3 Explorando el historial

A los commits nos podemos referir a ellos con sus identificadores. Al último commit también nos podemos referir como `HEAD`. Añadamos una línea más al archivo `prueba.txt`.

```
$ nano prueba.txt
$ cat prueba.txt
```

Ahora, para ver el último cambio con el último commit:

```
$ git diff HEAD prueba.txt
```

Si quitamos el HEAD de esa última instrucción veremos lo mismo. Pero si colocamos un `~numero` vamos a ver que nos estamos refiriendo al commit anterior número n.

```
$ git diff HEAD~1 prueba.txt
```

Con `git show` vamos a ver los cambios con respecto a un commit anterior.

```
$ git show HEAD~1 prueba.txt
```

También podemos referirnos a los commit por su identificador de números y letras enorme o por los primeros 7 números o letras:

```
$ git diff 451b2ad469b96e13547e13dc0e718613acdc987c prueba.txt
$ git diff 451b2ad prueba.txt
```

Revisemos el estado:

```
$ git status
```

Con la siguiente instrucción podemos regresar las cosas a como estaban antes de hacer el último cambio.

```
$ git checkout HEAD prueba.txt
$ cat prueba.txt
```

O podríamos usar uno de los identificadores de commits:

```
$ git checkout 451b2ad prueba.txt
$ cat prueba.txt
$ git status
```

Y para regresarlo al último commit de nuevo:

```
$ git checkout HEAD prueba.txt
```

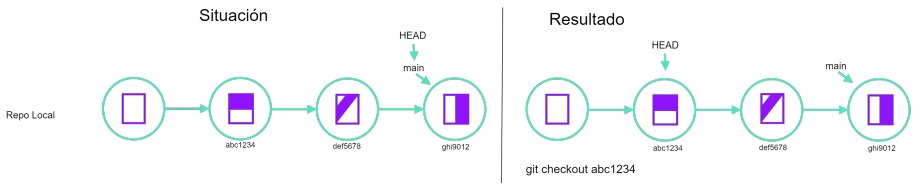


Figure 3.2: `git checkout <ID commit> <archivo>`, basdo en ASSP

El comando `git checkout` revierte los archivos a alguna versión anterior siempre que no lo hayamos añadido al staging área. Para revertir un commit usamos la instrucción `git revert [ID commit]`. Supongamos que tenemos un error en el archivo `prueba.txt` y que ya hicimos commit y queremos revertir al último cambio. Los pasos que haríamos serían los siguientes:

1) `git log` para identificar el ID del commit.
2) Copiar el ID del commit
3) `git revert [ID del commit]` para revertir a ese cambio.
4) Teclear el nuevo mensaje de commit.
5) Guardar y cerrar

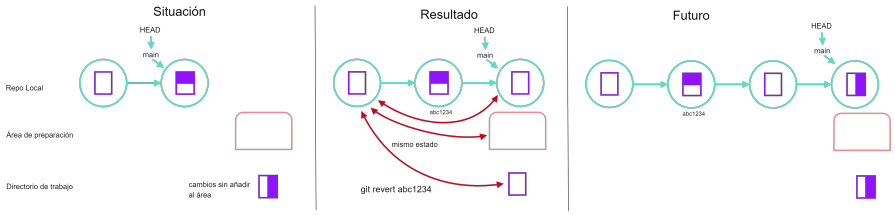


Figure 3.3: `git revert <ID commit>`, basado en APP

Si ya añadimos los cambios al área de preparación, ya no podemos usar git checkout simplemente.

Ejercicio: Realiza las siguientes instrucciones:

1) Crea una carpeta llamada `learn_revert`.
2) Muévete a la carpeta `learn_revert`
3) Inicializa el repositorio.
4) Crea un archivo llamado `first.txt` y añade una línea de texto.
5) Agrégalo al área de preparación y realiza el primer commit.
6) Crea el archivo `wrong.txt` y agrega una línea de texto.
7) Agrégalo al área de preparación y realiza un commit.
8) Agrega una segunda línea de texto al archivo `first.txt`, guárdalo, agrégalo al área de preparación y realiza un commit.
9) Agrega una tercera línea de texto al archivo `first.txt`, guárdalo, agrégalo al área de preparación y realiza un commit.

10) Queremos deshacer el commit realizado cuando se añadió el archivo `wrong.txt`. Como este commit fue el segundo de donde no estamos, podemos usar `git revert HEAD~2` (o podemos usar `git log` y encontrar el ID de ese commit).

¿Está el archivo `wrong.txt`? ¿Qué sucede con el historial de commits?

Otras opciones del historial del commit:

1) Para ver tanto las diferencias entre los archivos y los ID de los commits. Se puede colocar solo el nombre de un archivo y solo mostrara los commit que afectaban ese archivo o si no se coloca el nombre del archivo aplica sobre todo el historial de commits.

```
$ git log --patch prueba.txt
```

2) Para mostrar las descripciones detalladas de las modificaciones y archivos.

```
$ git log -p
```

3) Para mostrar los nombres de los archivos afectados en cada commit.

```
$ git log --name-only
```

4) Para mostrar los archivos afectados en cada commit con la leyenda de si fueron modificados (M) o añadidos (A) o eliminados.

```
$ git log --name-status
```

Consultar el siguiente link para ver más opciones y ejemplos.

3.4 Restore y reset

Otra forma de deshacer cambios es con `restore` y `reset`. Usualmente deshacer cambios se requiere para deshacer:

- Cambios antes de mandarlos al área de preparación.
- Cambios que ya se mandaron al área de preparación.
- Commits

Supongamos que hicimos un cambio en el archivo de prueba y lo guardamos y después decidimos que ya no queremos ese cambio, entonces usamos la opción:

```
$ git restore prueba.txt
```

Esto nos regresará a la versión del archivo del último commit. Esto no se puede deshacer, una vez echo esto no hay forma de recuperar los cambios que se habían realizado.

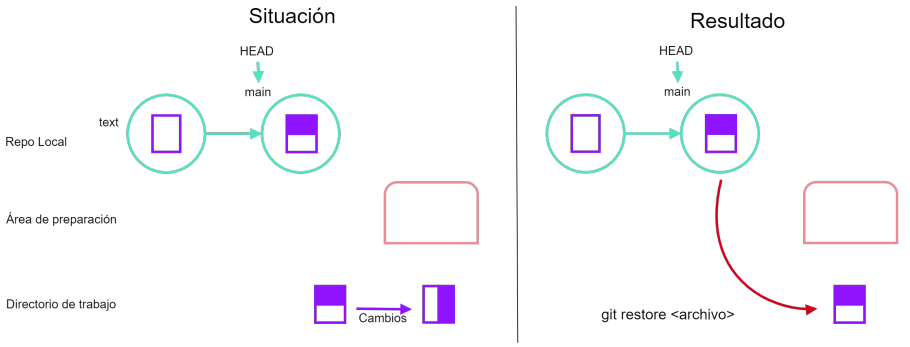


Figure 3.4: `git restore <archivo>`, basado en ASSP

Ahora, supongamos que hicimos un cambio y lo mandamos al área de preparación, entonces para sacarlo de esa área usamos:

```
$ git restore --staged prueba.txt
```

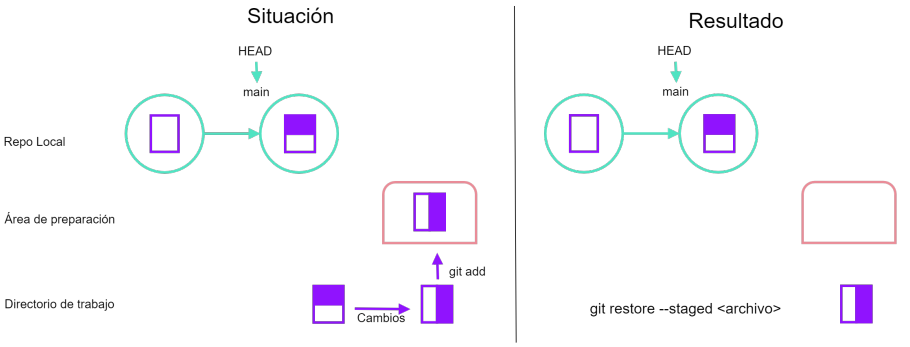


Figure 3.5: `git restore --staged <archivo>`, basado en ASSP

También se pueden restaurar todos los archivos de un proyecto con:

```
$ git restore .
```

Para restaurar a un commit anterior usamos el identificador del commit, por ejemplo:

```
$ git restore --source d3a9d6d prueba.txt
```

Si ahora revisamos el estado del proyecto veremos que si hay cambios.

Agreguemos una línea al archivo `prueba.txt` y añadamos el cambio al área de preparación. Usen `git checkout` para ver si podemos revertir el cambio. Veamos que nos dice el estado `git status`. Si usamos `git checkout -- prueba.txt` ya no veremos errores pero tampoco se restaurará el archivo.

Para hacerlo debemos usar `reset`:

```
$ git reset HEAD prueba.txt
```

Y si usamos ahora:

```
$ git status
```

Nos indica que ya podemos realizar la modificación con checkout:

```
$ git checkout -- prueba.txt
$ git status
```

Con `reset` tenemos tres posibles situaciones.

- Situación 1: `reset --soft HEAD~1`: si realizamos un cambio en nuestro archivo y lo añadimos al área de preparación, al realizar el `reset soft` al commit `HEAD~1` lo que estamos haciendo es como regresar a un commit anterior pero sin perder los cambios que ya tenemos en el área de preparación, entonces lo que va a resultar es que nuestro historial va a cambiar de un commit anterior hasta el cambio que tenemos ahora. Con `git status` vemos que no cambio nuestra área de preparación y después con `git log` podemos ver el cambio en el historial de commits.

- Situación 2: `reset --mixed HEAD~1`: si realizamos un cambio en nuestro archivo y lo añadimos al área de preparación, al realizar el `reset mixed` al commit `HEAD~1` lo que estamos haciendo es como regresar nuestra área de preparación a como estaba antes de ese commit, no perdemos el cambio realizado en el archivo pero nuestro historial cambiará, pasará del commit anterior hasta el próximo commit que realicemos. Con `git status` vemos que nuestra área de preparación si cambio y con `git log` vemos que nuestro último commit desapareció.

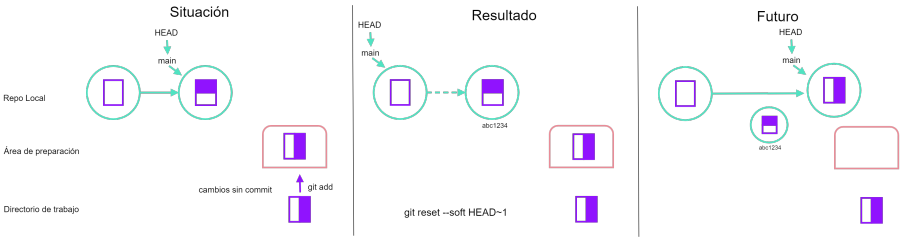
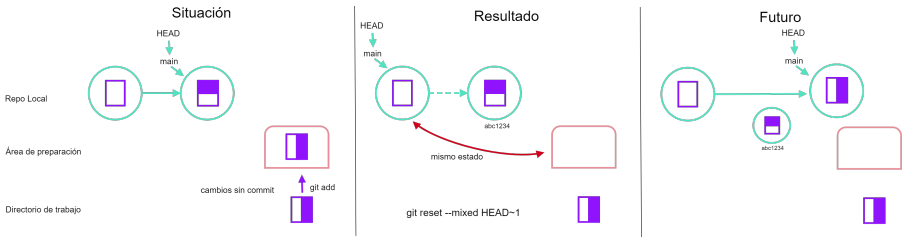


Figure 3.6: `git reset --soft HEAD~1`, basado en ASSP



- Situación 3: `reset --hard HEAD~1`: si realizamos un cambio en nuestro archivo y lo añadimos al área de preparación, al realizar el `reset hard` al commit `HEAD~1` lo que estamos haciendo es como regresar a un commit anterior pero perdiendo los cambios que ya tenemos en el área de preparación y en nuestro archivo actual, entonces lo que va a resultar es que estaríamos regresando hasta el commit anterior todo nuestro historial y a partir de ahí comenzarían nuestros cambios.

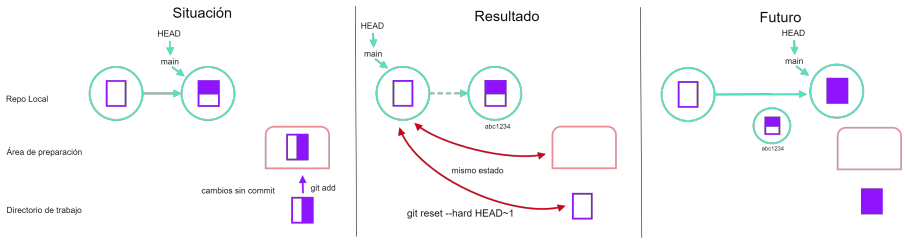


Figure 3.7: `git reset --hard HEAD~1`, basado en ASSP

3.5 Ignorar archivos/carpetas

Es muy usual tener un archivo llamado `.gitignore` donde se pueden colocar los nombres de archivos o carpetas que no queremos llevar registro.

Creemos unos archivos de prueba.

```
$ cd Mi_primer_repo
$ mkdir resultados
$ touch a.csv b.csv c.csv resultados/a.out resultados/b.out
```

Si preguntamos el estado veremos los cambios no registrados en el historial.

```
$ git status
```

Estos archivos por el momento no nos sirven de nada y guardarlos o registrarlos sería una perdida de tiempo/espacio. Para ignorarlos, creamos el archivo `.gitigno` y añadimos los nombres a ese archivo:

```
$ nano .gitignore
```

```
*.csv
resultados/
```

```
$ cat .gitignore
```

Estos patrones le están diciendo a Git que ignore todos los archivos `.csv` y todo lo que hay en la carpeta `resultados`, si después añadimos algo a la carpeta lo seguirá ignorando. Y si alguno de esos archivos ya se le dijo a Git que llevará su registro lo seguirá registrando.

```
$ git status
```

Si nos fijamos, el único documento que ahora nos menciona `Git` es el archivo `.gitignore`. Lo que nos falta es añadirlo y hacer el commit.

```
$ git add .gitignore
$ git commit -m "Creamos el archivo gitignore e ignoramos todo lo que hay en resultados
$ git status
```

El archivo `.gitignore` nos ayuda a no cometer el error de accidentalmente tratar de registrar y rastrear algo que se le dijo que no lo hiciera.

```
$ git add a.csv
```

Si realmente queremos agregarlo, tendríamos que usar la opción `-f`:

```
$ git add -f a.csv
```

Para ver el estado de los archivos ignorados usamos la siguiente instrucción:

```
$ git status --ignored
```

Ejercicio: Supongamos que tenemos las siguientes subcarpetas:

```
resultados/plots
resultados/datos
```

¿Qué tenemos que hacer si queremos ignorar solamente lo que hay en `datos` y no lo que hay en `plots`?

Para ignorar por ejemplo todos los archivos que terminan en `.csv` excepto uno en específico (`b.csv`) podemos indicarlo en el archivo `.gitignore` como:

```
*.csv  # ignoramos todos los csv
!b.csv # exepto el que se llama b.csv
```

Ejercicio: Supongamos ahora que tenemos la siguiente estructura de carpetas:

```
resultados/plots
resultados/datos
resultados/img
resultados/analisis
```

Y que queremos ignorar todo excepto lo que hay en datos. ¿Cómo lo harían?

Ejemplo: Supongamos que tenemos la siguiente estructura de archivos:

```
resultados/rdatos/a.csv
resultados/rdatos/b.csv
resultados/rdatos/c.csv
resultados/rdatos/info.txt
```

¿Cómo le indicas a Git que ignore todos los `csv` de la carpeta `rdatos` menos el que se llama `info.txt`?

Ejercicio: Supongamos que tenemos la siguiente estructura de datos:

```
resultados/a.csv
resultados/analisis1/b.csv
resultados/analisis2/c.csv
resultados/analisis2/sub_1/d.csv
```

¿Cómo le indicamos a Git que ignore todos los archivos `.csv` sin indicar manualmente todos los directorios?

Ejercicio: Si en el archivo `.gitignore` escribimos lo siguiente, ¿qué está ignorando?

```
*.csv
!*.csv
```

3.6 Github

El valor del control de versiones se hace evidente al comenzar a colaborar con otros. Contamos con la mayor parte de las herramientas necesarias para ello; lo único que resta es transferir cambios de un repositorio a otro.

Sistemas como Git posibilitan el traslado de trabajo entre cualquier par de repositorios. No obstante, en la práctica, resulta más conveniente utilizar una copia como punto central y mantenerla en la web en lugar de en la computadora portátil de alguien.

Vamos a comenzar por crear un repositorio remoto, pero para eso necesitamos configurar nuestra cuenta de Github también.

3.6.1 Paso 1: Crear un repositorio remoto

Lo primero que vamos a hacer es crear un repositorio remoto. Entra a tu cuenta de Github y dale click en Nuevo.

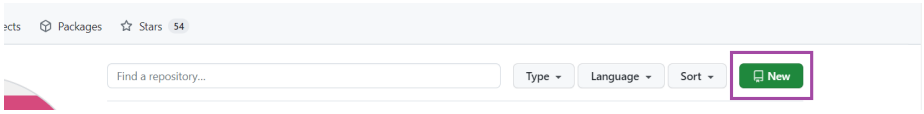


Figure 3.8: Crear repositorio nuevo

Ponle de nombre `Mi_primer_repo` (o el nombre que hayas usado en las secciones anteriores). Deja marcada la opción de público y no añadas un README ni una licencia.

Al darle click en crear repositorio, la página nos mostrará la siguiente información que es la que usaremos para configurar nuestro local con el remoto.

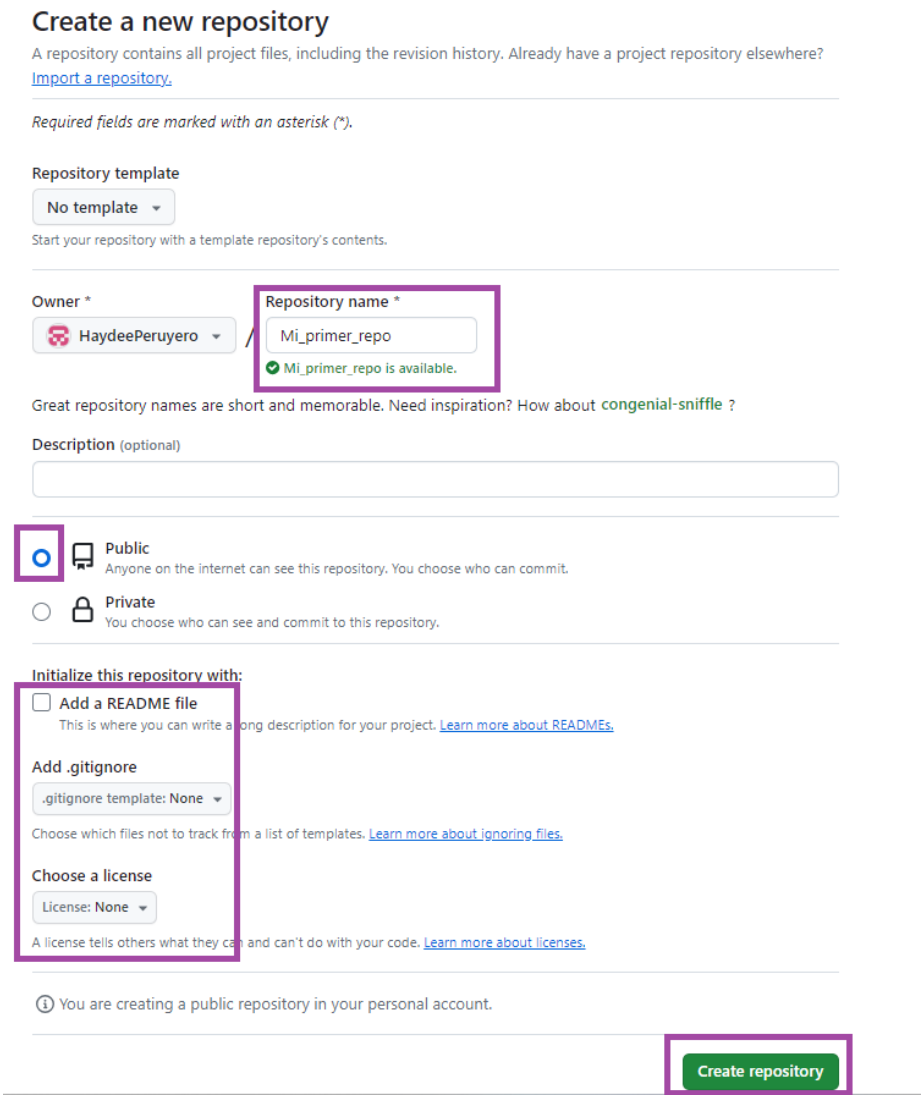
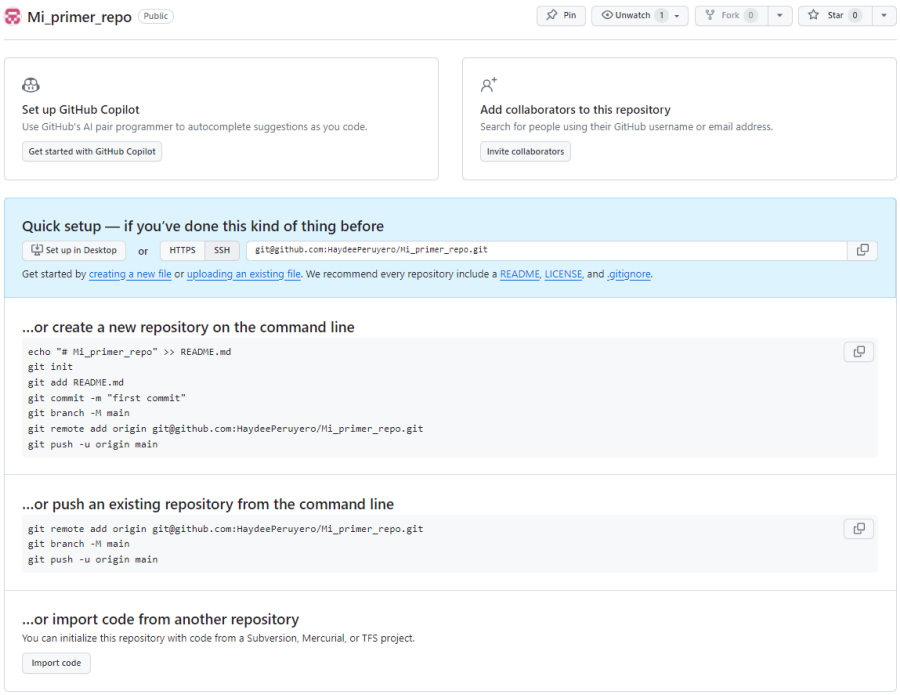


Figure 3.9: Crear repositorio vacío



Lo que acabamos de hacer es como si en nuestra terminal hubiéramos realizado lo siguiente:

```
$ mkdir Mi_primer_repo
$ cd Mi_primer_repo
$ git init
```

3.6.2 Paso 2: Conectar local a remoto

La página principal del repositorio remoto muestra una serie de información que necesitamos usar para conectar el repositorio remoto en Github con el repositorio local de nuestra computadora. Vamos a usar el protocolo de conexión SSH, da click en donde dice SSH y a continuación en el icono de copiar.

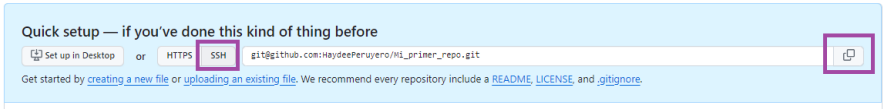


Figure 3.10: SSH link para clonar el repositorio

Ahora, dentro de nuestra carpeta del repositorio local, abrir una terminal y correr lo siguiente:

```
$ git remote add origin git@github.com:User/Mi_primer_repo.git
```

Para revisar que si se haya realizado correctamente procedemos a usar lo siguiente:

```
$ git remote -v
```

3.6.3 Paso 3: Conexión mediante SSH

Primero verificamos si ya tenemos algún par de llaves:

```
$ ls -al ~/.ssh
```

Si ya tienen algún par de llaves configuradas las van a ver listadas, si no tiene ninguna les saldrá una leyenda como la siguiente:

```
ls: cannot access '/c/Users/User/.ssh': No such file or directory
```

3.6.3.1 Paso 3.1: Crear un par de llaves SSH

Para crear el par de llaves usamos el siguiente comando, la opción `-t` se refiere al tipo de algoritmo usado y la opción `-C` indica un comentario para la llave, en este caso el comentario es nuestro correo.

```
$ ssh-keygen -t ed25519 -C "email@dominio.com"
```

Si tu sistema operativo no lo permite, usa `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`.

Como queremos usar el archivo default, solo damos Enter. Ahora nos pedirá una contraseña, tecleala, no vas a ver nada en la pantalla. Una vez creada verás en pantalla algo como lo siguiente:

```
Your identification has been saved in /c/Users/user/.ssh/id_ed25519
Your public key has been saved in /c/Users/user/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:SMSPIStNyA10KPxuYu94KpZg9AYjgt9g46A4kFy3g1o user@domain
The keys randomart image is:
+--[ED25519 256]--+
|^B== o.          |
|%*= *.+          |
|+=.E =.+         |
```

```
| .=.+.o..        |
|...   . S        |
|.+ o             |
|+ =              |
|.o.o             |
|oo+.             |
+----[SHA256]-----+
```

Lo que dice `identification` ser refiere a la llave privada la cual no debes compartir nunca y la cadena de caracteres que dice `fingerprint` se refiere a parte de tu llave pública.

Si repetimos el comando siguiente, verán ahora ya sus dos claves pública y privada.

```
$ ls -al ~/.ssh
```

Ahora que ya tenemos las claves, debemos decirle a GitHub cuales son.

```
$ cat ~/.ssh/id_ed25519.pub
```

Copia la cadena de caracteres, ve a la configuración de tu perfil de GitHub y da clic en “SSH and GPG Keys”.

Una vez ahí da clic en “Nueva llave SSH”.

Después coloca un título que te permita identificar que será la clave con la que usarás la computadora y pega tu llave pública.

Ahora solo falta revisar la conexión desde la terminal.

```
$ ssh -T git@github.com
```

Si vez un mensaje similar al siguiente, significa que quedo completa la autenticación.

```
$ Hi Name! Youve successfully authenticated, but GitHub does not provide shell access.
```

3.6.4 Paso 4: Push and pull

Una vez que ya tenemos configurado todo, solo falta enviar todo lo que tenemos en el repo local al remoto. Si se establecio la contraseña nos la va a pedir en la terminal o una ventana aparte.

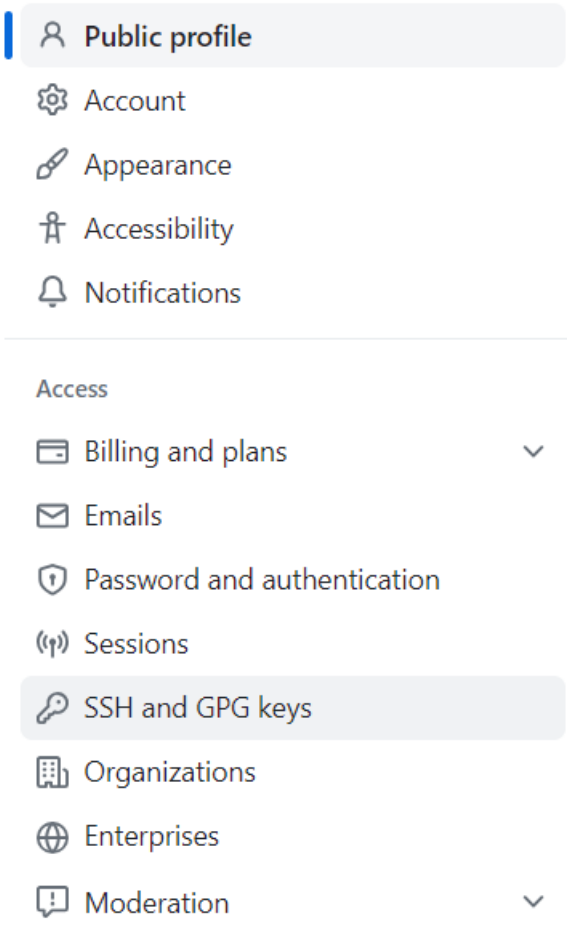


Figure 3.11: SSH and GPG Keys

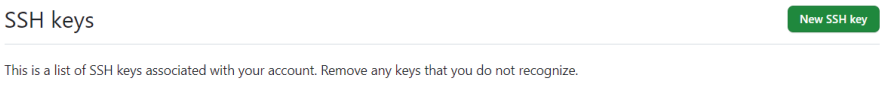


Figure 3.12: Nueva llave SSH

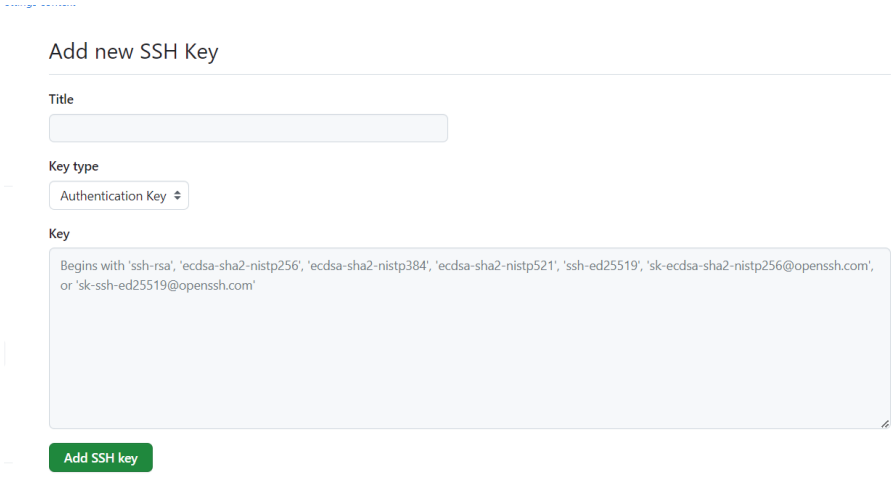


Figure 3.13: Llave pública

```
$ git push origin main
```

En esa instrucción, `origin` se refiere al repositorio remoto y `main` al local (las ramas que estamos intentando poner en el mismo contenido).

La situación en la que estamos es la siguiente:

Para actualizar nuestro repositorio local, lo que debemos hacer es lo siguiente:

```
$ git pull origin main
```

Como no hemos realizado ningún cambio en el remoto, no veremos nada nuevo en el local. En el remoto también podemos añadir archivos directamente.

Ejercicio: Añade un archivo nuevo desde el repositorio remoto y actualiza tus cambios en el local.

3.7 Colaboradores

Para esta parte, vamos a trabajar en parejas (si no es posible pueden abrir una segunda terminal para la realizar la parte de su equipo).

Vamos a ir a nuestro repositorio que creamos en GitHub y vamos a ir a la configuración y después en donde dice colaboradores.

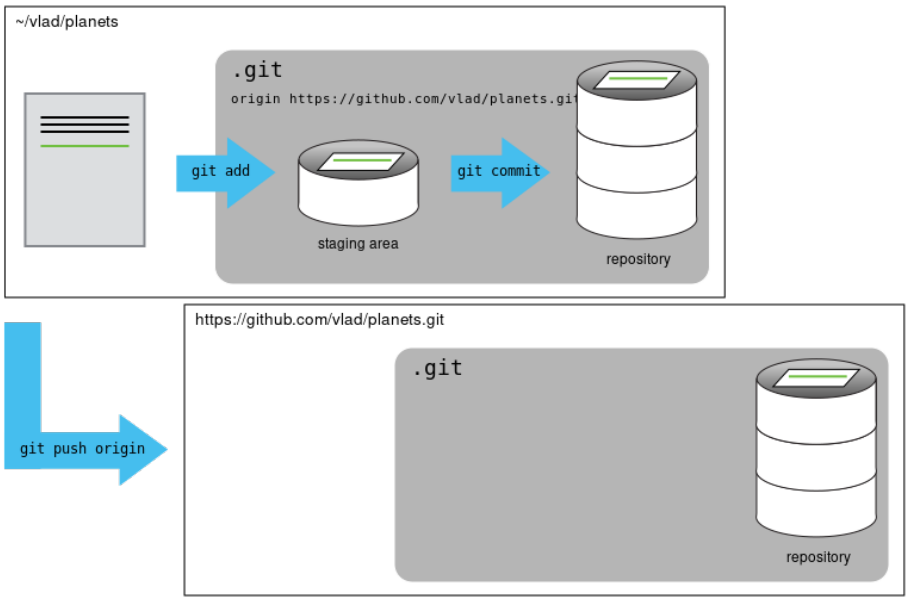


Figure 3.14: `git push origin main`

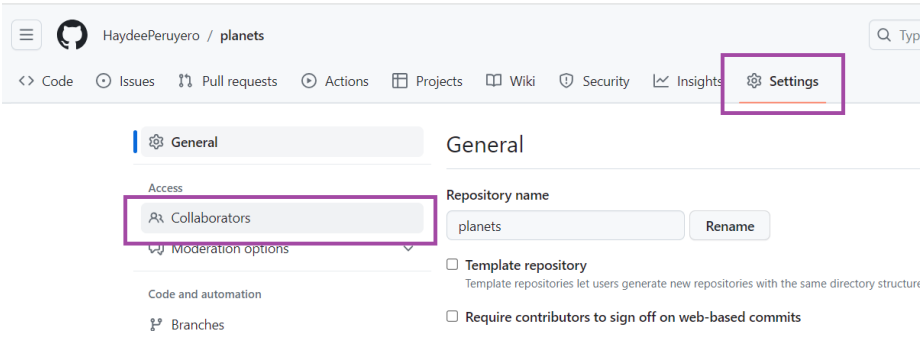


Figure 3.15: Configurar colaboradores

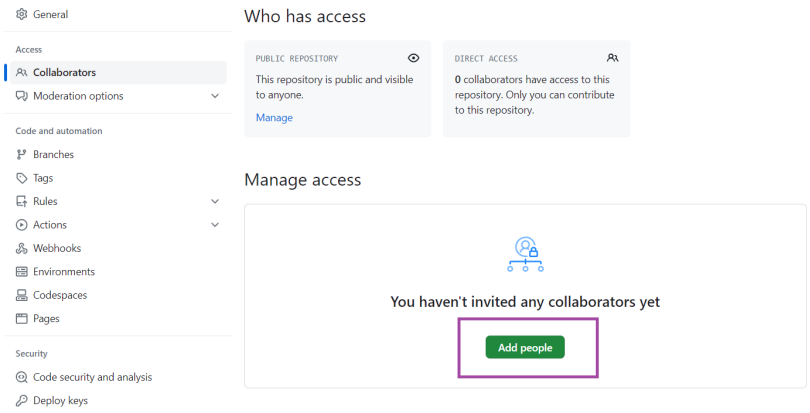


Figure 3.16: Añadir colaboradores

Una vez en esa ventana, den clic en añadir colaboradores. Van a buscar con el nombre de usuario de su compañero y dan enter.

En su correo o cuenta de github, deben ir a notificaciones y aceptar la invitación. También pueden usar el link.

Ahora, su colaborador debe descargar el repositorio a su computadora, a este paso se le llama clonar un repositorio. Para esto, abran una terminal y realicen lo siguiente:

```
$ git clone git@github.com:User/Mi_primer_repo.git ~/colaborador-Mi_primer_repo
```

La última parte de esa instrucción es la dirección de su computadora donde se clonara el repositorio de su colaborador.

Ahora, el colaborador realizará un cambio en el repositorio. Para esto, creará un archivo, lo añadira al área de preparación, hará el commit correspondiente y enviará los cambios al remoto.

```
$ nano notas.txt
$ git add notas.txt
$ git commit -m "Añadimos archivo de colaborador"
$ git push origin main
```

Si revisamos en la página de Github, veremos ahora un cambio en el repositorio, junto con el commit y quien lo realizo. Finalmente, actualizaremos el repositorio original local con los cambios del colaborador.

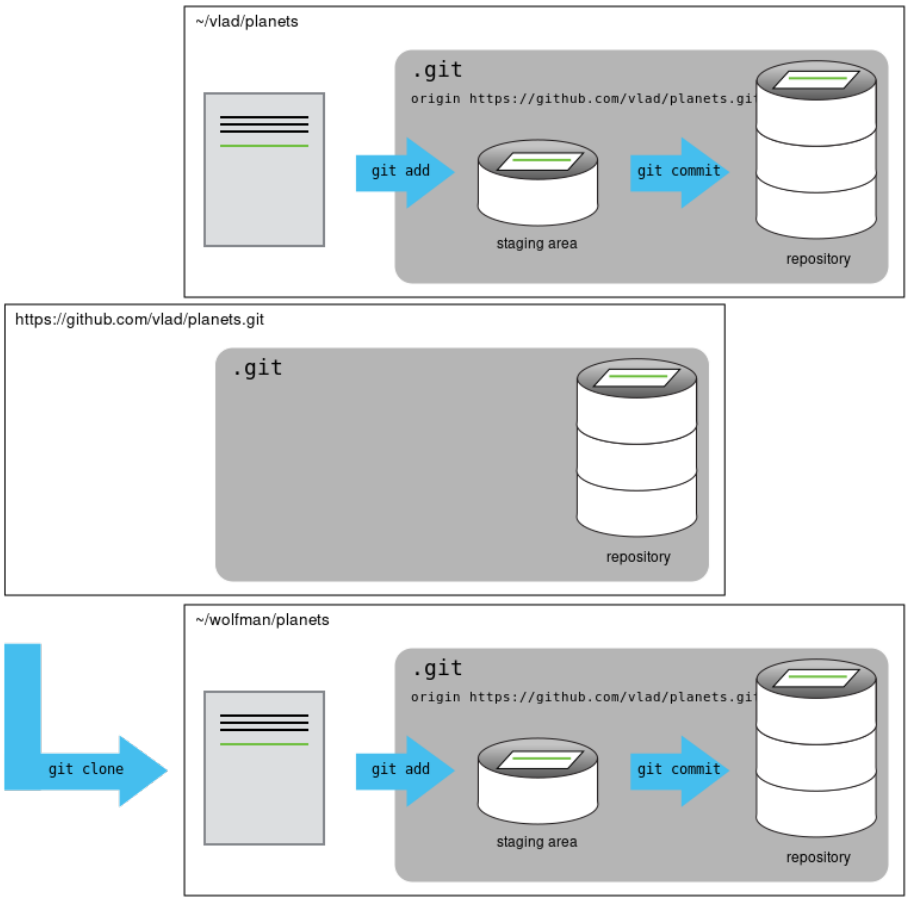


Figure 3.17: Estructura de los repositorios al clonarlos. Imagen de SC

```
$ git pull origin main
```

Una buena practica cuando se trabaja con colaboradores es realizar la siguiente serie de pasos:

1) Antes de comenzar a trabajar, siempre actualizar nuestro repositorio local con `git pull origin main`
2) Realizar cambios y añadirlos al área de preparación con `git add`
3) Realizar el commit con un mensaje apropiado que nos permita detectar que cambio se realizó.
4) Actualizar el repositorio remoto con los cambios usando `git push origin main`.

Otra buena practica es trabajar con ramas, este sería un paso antes del 2.

Ejercicio: Replicar lo que se hizo en esta sección cambiando roles de quien es el colaborador y quien el dueño del repositorio local.

3.8 Conflictos

Cuando comenzamos a trabajar con colaboradores, es usual que generemos conflictos si no se trabaja de forma adecuada.

Vamos a crear un conflicto para después resolverlo.

El colaborador va a modificar el `notas.txt` añadiendo algo. Luego lo añadirá al área de preparación, realizará el commit correspondiente y finalmente actualizará el repositorio remoto.

```
$ nano notas.txt
$ git add notas.txt
$ git commit -m "Modificamos el archivo notas para crear un conflicto"
$ git push origin main
```

Ahora, el dueño del repositorio realizará un cambio también al archivo notas (sin antes actualizar con los últimos cambios del colaborador) y realizará todos los pasos hasta poder actualizar el repositorio remoto.

```
$ nano notas.txt
$ git add notas.txt
$ git commit -m "Cambios en el archivo notas por el dueño del repositorio"
$ git push origin main
```

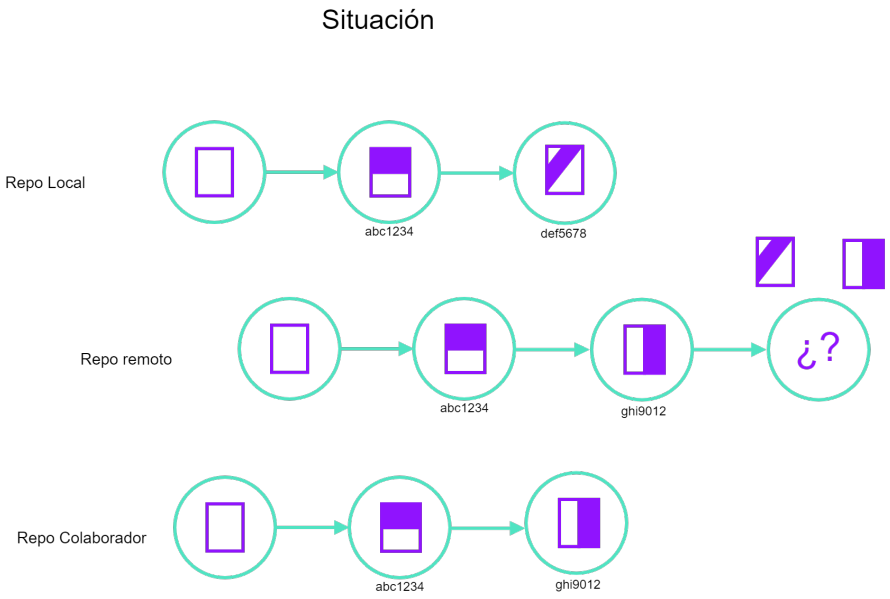


Figure 3.18: Conflicto generado cuando se intenta hacer un `git push`

Git no autorizo hacer el push porque había cambios en el remoto que no habíamos actualizado en el local, entonces vamos a actualizar el local primero, hacer un merge en la copia en la que estamos trabajando y después ya hacer un push.

```
$ git pull origin main
```

Vamos a ver un mensaje de que hay un merge que no pudo resolver porque se trabajo en la misma línea en el mismo archivo. Si abrimos el archivo vamos a ver algo como lo siguiente:

```
texto aqui
<<<<<<< HEAD
cambios locales aqui
=======
cambios del colaborador aqui
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Ahora lo que tenemos que hacer es decidir que cambios queremos conservar, para eso, podemos borrar nuestros cambios, los del colaborador, ambos, o cambiar todo por algo nuevo (y borrar los identificadores de los conflictos).

Una vez finalizado el merge/resolver conflicto, ya podemos añadirlo al área de preparación, realizar el commit y enviarlo al remoto.

```
$ git add notas.txt
$ git status
$ git commit -m "Resolvimos el conflicto en el archivo notas"
$ git push origin main
```

Cuando el colaborador intente hacer un `git pull` no verá ningún conflicto ya que git conservo el registro de que se resolvió y a la copia del colaborador se actualizará sin ningún problema.

```
$ git pull origin main
```

Una forma de evitar algunos conflictos es trabajando en ramas, o distribuyendo el trabajo y que cada quien trabaje sobre archivos diferentes.

También se puede dar el caso de que los conflictos sean por archivos con el mismo nombre pero con contenidos totalmente diferentes, por ejemplo con imágenes.

Vamos a crear de nuevo un conflicto con una imagen y vamos a tratar de resolverlo. Tanto el colaborador como el dueño del proyecto va a crear una imagen con el nombre imagen_prueba.jpg, guardará los cambios y realizará todo hasta mandarlo al repositorio remoto.

```
$ head -c 1024 /dev/urandom > imagen_prueba.jpg
$ ls -lh imagen_prueba.jpg
```

Lo añadimos al área de preparación, realizamos commit y lo enviamos al repositorio remoto.

```
$ git add imagen_prueba.jpg
$ git commit -m "Se creo imagen random para generar conflicto."
```

El colaborador realizará un push de su imagen al repositorio remoto.

```
$ git push origin main
```

Ahora el dueño del repositorio tratará de hacer un push también al repositorio.

```
$ git push origin main
```

Lo primero que nos va a pasar y decir git, es que no actualizamos nuestro repositorio con los cambios como en el ejercicio anterior. Vamos a hacer un pull.

```
$ git pull origin main
```

Nos va a marcar que hay conflictos y que no puede hacer auto merge y nos dirá adicionalmente un mensaje similar al siguiente:

```
warning: Cannot merge binary files: imagen_prueba.jpg (HEAD vs. 439dc8c08869c343538f6dc4a2b615b05
```

Esto se debe a que como es un archivo que no es de texto no puede empalmar los cambios. Entonces las opciones que tenemos es decidir quedarnos con solo una de las dos imágenes o renombrarlas para quedarnos con ambos.

1) Quedarnos con la imagen del dueño del repo:

La imagen del dueño del repo es el HEAD y la del colaborador tiene un id de commit. Entonces procedemos a lo siguiente.

```
$ git checkout HEAD imagen_prueba.jpg
$ git add imagen_prueba.jpg
$ git commit -m "Usar la imagen del dueño del repo en lugar de la del colaborador"
```

2) Quedarnos con la imagen del colaborador del repo:

La imagen del colaborador tiene un id de commit, buscarlo. Entonces procedemos a lo siguiente.

```
$ git checkout 439dc8c0 imagen_prueba.jpg
$ git add imagen_prueba.jpg
$ git commit -m "Usar la imagen del colaborador del repo en lugar de la del dueño"
```

3) Quedarnos con ambas imágenes:

```
git checkout HEAD imagen_prueba.jpg
$ git mv imagen_prueba.jpg imagen_prueba-dueño.jpg
$ git checkout 439dc8c0 imagen_prueba.jpg
$ mv imagen_prueba.jpg imagen_prueba-colaborador.jpg
```

Y finalmente para remover la imagen de prueba y añadir las dos nuevas versiones:

```
$ git rm imagen_prueba.jpg
$ git add imagen_prueba-dueño.jpg
$ git add imagen_prueba-colaborador.jpg
$ git commit -m "Use two images: dueño y colaborador"
$ git push origin main
```

3.9 Trabajando con Ramas

Trabajar con ramas en Git es una parte fundamental del flujo de trabajo colaborativo. Los siguientes pasos son los básicos para trabajar de esta manera.

3.9.1 1. Crear una rama

Cuando trabajas en un proyecto, es una buena práctica crear una rama separada para cada nueva función o corrección de errores que estés desarrollando. Para crear una nueva rama en Git, utiliza el comando:

```
$ git checkout -b nombre_de_la_rama
```

Este comando crea una nueva rama y te cambia a ella al mismo tiempo.

Otra opción es la siguiente:

```
$ git branch nombre_de_la_rama
$ git switch nombre_de_la_rama
```

3.9.2 2. Trabajar en la rama

Después de crear la rama, puedes comenzar a trabajar en tus cambios. Realiza tus modificaciones en los archivos como lo harías normalmente.

3.9.3 3. Agregar y confirmar cambios

Una vez que hayas realizado cambios que desees incluir en la rama, añádelos al área de preparación con:

```
$ git add nombre_del_archivo
```

Luego, confirma los cambios con un mensaje descriptivo:

```
$ git commit -m "Mensaje descriptivo de los cambios"
```

3.9.4 4. Empujar la rama al repositorio remoto

Si estás trabajando en un repositorio remoto compartido con otros colaboradores, es posible que desees compartir tus cambios. Para esto, hay dos formas de hacerlo. La primera es enviar los cambios a una rama remota y después confirmarlos y unirlos, para eso utiliza el comando:

```
$ git push origin nombre_de_la_rama
```

Esto enviará la nueva rama y los cambios asociados al repositorio remoto.

3.9.5 5. Fusionar cambios

Una vez que hayas completado tus cambios y estés listo para incorporarlos al proyecto principal, puedes fusionar tu rama con la rama principal (generalmente `main` o `master`). Para hacerlo, primero cámbiate a la rama principal:

```
$ git checkout main
```

Otra opción:

```
$ git switch main
```

Luego, fusiona tu rama con la rama principal:

```
$ git merge nombre_de_la_rama
```

NOTA: Los pasos 4 y 5 se pueden intercambiar de orden, es decir primero hacer el merge local cambiandonos a la rama principal y después enviando los cambios al remoto. Es importante primero hacer un `git pull` para actualizar nuestro local en la rama principal.

3.9.6 6. Resolver conflictos (si los hay)

Es posible que ocurran conflictos durante el proceso de fusión si otros colaboradores han realizado cambios en las mismas partes de los archivos. Git te indicará los conflictos y te permitirá resolverlos manualmente.

3.9.7 7. Eliminar la rama (opcional)

Una vez que hayas fusionado tus cambios en la rama principal y ya no necesites la rama de la función, puedes eliminarla:

```
$ git branch -d nombre_de_la_rama
```

3.9.8 8. Actualizar y sincronizar

Es importante mantener tu repositorio local actualizado con los cambios de otros colaboradores. Para hacerlo, utiliza:

```
$ git pull origin main
```

Esto traerá los últimos cambios de la rama principal del repositorio remoto y los fusionará con tu rama local.

Siguiendo estos pasos, podrás trabajar de manera efectiva con ramas en Git en un entorno colaborativo. Recuerda comunicarte con tus colaboradores y mantener un flujo de trabajo ordenado para evitar conflictos y errores.

3.10 Conectar con overleaf

Lo primero es crear el documento en Overleaf que nos interesa. Después se clona el archivo con la dirección que da Overleaf. Esa será la ruta de la carpeta. De manera local podemos cambiar el propio archivo *.tex* por uno que ya tengamos (parece que es más fácil hacer esto que intentar crear un nuevo documento en Overleaf a partir un repositorio existente, pero falta hacer más pruebas); también, es posible cambiar el nombre de la carpeta en la cual se generó el repositorio (parece que no hay problemas con Overleaf pero hay que hacer más pruebas).

A continuación los pasos para trabajar localmente una vez que ya se tiene el repositorio creado.

1) `cd ruta_de_la_carpeta`

2) `git branch nombre_rama_local`

3) `git switch nombre_rama_local` (o master)

4) `git status` (para ver cambios)

5) `git add .` (para añadir los cambios-todos)

6) ‘git commit -m “nombre del mensaje”“

7) `git switch master`

8) `git pull`

9) `git merge nombre_rama_local`

(a) `git switch nombre_rama_local` (b) `git merge master` (para actualizar ahora rama local) (c) seguir trabajando sobre la misma rama local (d) repetir de 4 a 9

10) `git push`

11) `git branch -d nombre_rama_local`

Chapter 4

Python

4.1 Tipos de datos

4.2 Flujo de control

4.3 Visualización de datos

4.4 Manipulación de bases de datos

4.5 Análisis exploratorio de bases de datos

4.6 Funciones y scripts

4.7 Buenas practicas

4.8 Procesamiento de alto rendimiento

4.9 Programación en paralelo

Chapter 5

SQL

5.1 Bases de datos y manipulación

5.2 Explorar datos categóricos y texto no estructurado

5.3 Comparación con los otros programas

5.4 Valores faltantes

5.5 Combinar bases de datos

Chapter 6

Power BI

6.1 Introducción a Power BI

6.2 Transformando y visualizando datos

6.3 Manipulación de bases de datos

6.4 Análisis exploratorio de bases de datos

6.5 Variables categóricas y continuas

Chapter 7

R

7.1 Tipos de datos

7.2 Manipulación de bases de datos

7.3 Análisis exploratorio de bases de datos

7.4 Reportes con RMarkdown

7.5 Páginas web