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

Haydeé Peruyero

2024-02-06

Contents

1 Temas Selectos de Análisis Numérico y Computación Científica: Computo científico para el análisis de datos **5**

1.1 Temario 5

1.2 Referencias 6

1.3 Material interesante 6

1.4 DataCamp 6

2 Shell **9**

2.1 Navegar en archivos y directorios 9

2.2 Manipulación de archivos y directorios 12

2.3 Tuberías y filtros 15

2.4 Ciclos 19

2.5 Scripts 24

2.6 Buscando y encontrando cosas 28

2.7 Descarga y limpieza de bases de datos 28

3 Git y Github **29**

3.1 Repositorios 29

3.2 Flujo de trabajo en Git 29

3.3 Comparando cambios 29

3.4 Crear Ramas 29

3.5 Actualizando ramas 29

3.6 Revertir cambios 29

3.7 Resolver conflictos 29

4 Python **31**

4.1 Tipos de datos 31
4.2 Flujo de control 31
4.3 Visualización de datos 31
4.4 Manipulación de bases de datos 31
4.5 Análisis exploratorio de bases de datos 31
4.6 Funciones y scripts 31
4.7 Buenas practicas 31
4.8 Procesamiento de alto rendimiento 31
4.9 Programación en paralelo 31

5 SQL **33**

5.1 Bases de datos y manipulación 33
5.2 Explorar datos categóricos y texto no estructurado 33
5.3 Comparación con los otros programas 33
5.4 Valores faltantes 33
5.5 Combinar bases de datos 33

6 Power BI **35**

6.1 Introducción a Power BI 35
6.2 Transformando y visualizando datos 35
6.3 Manipulación de bases de datos 35
6.4 Análisis exploratorio de bases de datos 35
6.5 Variables categóricas y continuas 35

7 R **37**

7.1 Tipos de datos 37
7.2 Manipulación de bases de datos 37
7.3 Análisis exploratorio de bases de datos 37
7.4 Reportes con RMarkdown 37
7.5 Páginas web 37

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

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?

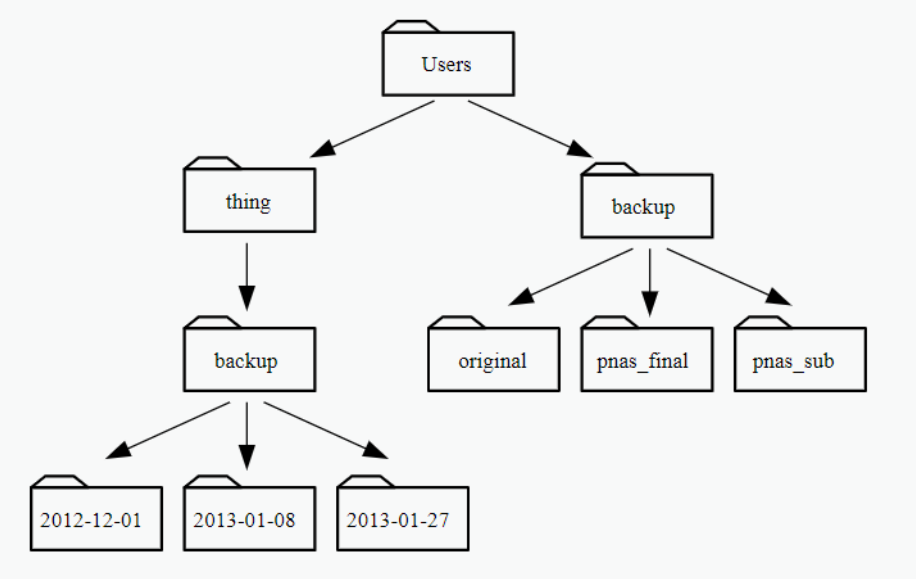

Figure 2.1: Ejercicio SC

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 .
```

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
```

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.*

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
```

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
```

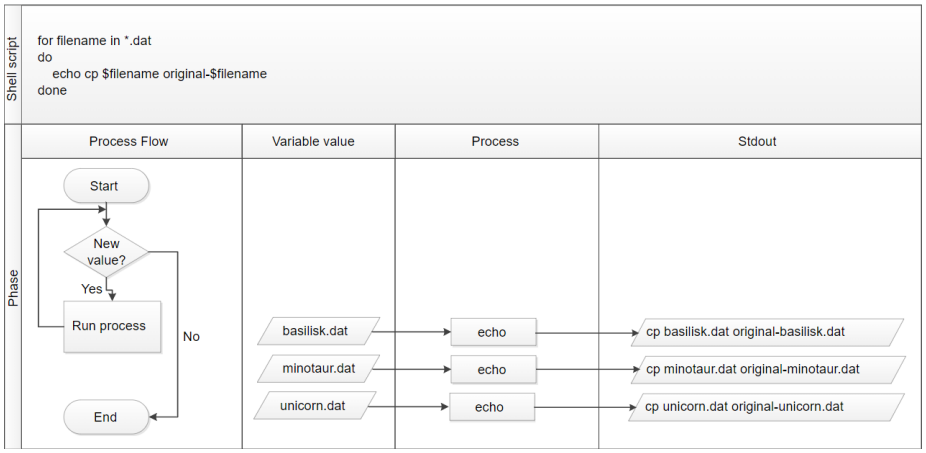

Figure 2.2: Debugging

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.

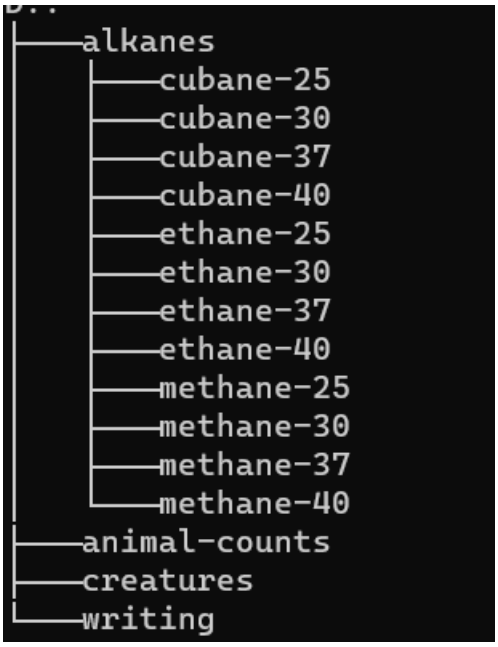

Figure 2.3: estructura-experimentos

```
$ 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?

2.6 Buscando y encontrando cosas

2.7 Descarga y limpieza de bases de datos

Chapter 3

Git y Github

3.1 Repositorios

3.2 Flujo de trabajo en Git

3.3 Comparando cambios

3.4 Crear Ramas

3.5 Actualizando ramas

3.6 Revertir cambios

3.7 Resolver conflictos

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