source:https://swcarpentry.github.io/shell-novice/

ubuntu- it is an linux distribution(complete linux OS) based on debian(GNU- GNU's Not Unix,used with a kernel called linux) and composed mostly of free and open source software. it is OFFICIALLY released in 3 editions-desktop,server,core of IOT devices and robots, these can be run on computer or virtual machine

unix shell-(the need for unix shell arises when we have some task that is repeated somany times that an individual will feel that it is better to use some commands and complete it as general procedure takes time and also is more prone to errors) A unix shell is a command line interface(a medium which will accept certain commands and let the work get completed) and a scripting language.

shell is a program where user can type and execute commands, tough commands(commands which will solve climate change) and easy commands (as creating an empty directory) and run other programs as well , the most popular unix shell is BASH(Bourne Again SHell- as it is derived from shell written by Stephen Bourne), it has high action to keystroke ratio,performs repetitive tasks faster by scripting,can access networked machines, it's disadvantages are textual in nature, cryptic commands and execution

prompt- a symbol that is displayed on the shell, by the shell which indicates that the command prompt is ready to take input command and execute(most probably the prompt will be a '$')

text cursor- this is the place where the command you type will be inserted, it can be a flashing or a solid block, in general a cursor will be an underscore'\_' or s pipe symbol '|'

the shell will execute the commands it already know off, any commands that you type miscorrectly or any commands that you type which are not there, it will throw a command not found error

file system- a part of Operating system, which is responsible for managing files(which hold information) and directories or folders(which hold files and/or other folders)

root directory holds everything else (by everything it means all files and folders)

ls -F : ls will list the files and F will give a character displayed at end of each item which will further specify what type of item it is

there are 3 types of end characters:

/ - indicates that it is a directory

@ - indicates that it is a link

\* - indicates that it is a executable

= - means socket.

| - means named pipe.

> - means door.

difference between man and help is that help will display content in same window, whereas man will display everything related to the command in different window

to navigate through man results:

you may use ↑ and ↓ to move line-by-line, or try B and Spacebar to skip up and down by a full page. To search for a character or word in the man pages, use / followed by the character or word you are searching for. Sometimes a search will result in multiple hits. If so, you can move between hits using N (for moving forward) and Shift+N (for moving backward).

To quit the man pages, press Q

a single dot(.) represents the current working directory and double dots(..) represents the parent directory

ls -F -a is equivalent to ls -Fa

files that are prefixed by a '.' are hidden files in linux file system, they are hidden to avoid the configuration files(files that start with .) while using ls, even though ls -a will display hidden files as well

an empty cd command will reset the (current working directory)cwd to home directory

-

1:saivinil\_pratap@TIGER02143:~$ cd tempo

2:saivinil\_pratap@TIGER02143:~/tempo$ cd tempor

3:saivinil\_pratap@TIGER02143:~/tempo/tempor$ cd

4:saivinil\_pratap@TIGER02143:~$ cd tempo/tempor

5:saivinil\_pratap@TIGER02143:~/tempo/tempor$ cd -

/home/saivinil\_pratap

in the above set of executions

step1:we moved to tempo

step2:we moved to tempor

step3: ON PRESSING cd YOU WILL RETURN TO HOME DIRECTORY.

step4:from home the command will move cwd to /tempo/tempor

step5: on pressing 'cd -' command it will move to prior directory before the last cd command happened(in step4 we used cd command, so before that cd command is executed the cwd is home/saivinil\_pratap so after step 5 it will again move to /home/saivinil\_pratap)

suppose we are working somewhere else deep down other directory and

1:we want to know which is home directory - pressing tilde (~) is enough

2:we want to move to home directory- on giving empty cd command it will move to home directory

3: 'cd ~' will also move the cwd to /home/saivinil\_pratap

4:let's suppose there is a folder "data" in /home/saivinil\_pratap/ and if you want to go to home, you can cd ~/data/../ will also move the cwd to /home/saivinil\_pratap/ i.e, home directory, it is complicated and unecessary, but it is a valid command

- '/' stands for home directory in most of the distributions

- $ ls -F /

here $ -> prompt

ls -> command

-F -> option

/ -> argument

-ls -a or ls --all

both commands are same

-list the files in order of time or order of date (sort files by time)

ls -lt

-$ ls -F /

this will list all the files along with its types in root directory

-The -p option allows mkdir to create a directory with nested subdirectories in a single operation:

$ mkdir -p project/data project/results

-Complicated names of files and directories can make your life painful when working on the command line. Here we provide a few useful tips for the names of your files and directories.

Don’t use spaces.

Spaces can make a name more meaningful, but since spaces are used to separate arguments on the command line it is better to avoid them in names of files and directories. You can use - or \_ instead (e.g. north-pacific-gyre/ rather than north pacific gyre/). To test this out, try typing mkdir north pacific gyre(this will create 3 different directories- north, pacific,gyre) and see what directory (or directories!) are made when you check with ls -F.

Don’t begin the name with - (dash).

Commands treat names starting with - as options.

Stick with letters, numbers, . (period or ‘full stop’), - (dash) and \_ (underscore).

Many other characters have special meanings on the command line. We will learn about some of these during this lesson. There are special characters that can cause your command to not work as expected and can even result in data loss.

If you need to refer to names of files or directories that have spaces or other special characters, you should surround the name in quotes ("").

-Create a text file

Let’s change our working directory to thesis using cd, then run a text editor called Nano to create a file called draft.txt:

$ cd thesis

$ nano draft.txt (if the file does not exit, a new file is created)

Which Editor?

When we say, ‘nano is a text editor(which runs within the shell)’ we really do mean ‘text’: it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because it is one of the least complex text editors. However, because of this trait, it may not be powerful enough or flexible enough for the work you need to do after this workshop. On Unix systems (such as Linux and macOS), many programmers use Emacs or Vim (both of which require more time to learn), or a graphical editor such as Gedit. On Windows, you may wish to use Notepad++. Windows also has a built-in editor called notepad that can be run from the command line in the same way as nano for the purposes of this lesson.

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer’s start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you ‘Save As…’

Let’s type in a few lines of text. Once we’re happy with our text, we can press Ctrl+O (press the Ctrl or Control key and, while holding it down, press the O key) to write our data to disk (we’ll be asked what file we want to save this to: press Return to accept the suggested default of draft.txt).

screenshot of nano text editor in action

Once our file is saved, we can use Ctrl+X to quit the editor and return to the shell.

Control, Ctrl, or ^ Key

The Control key is also called the ‘Ctrl’ key. There are various ways in which using the Control key may be described. For example, you may see an instruction to press the Control key and, while holding it down, press the X key, described as any of:

Control-X

Control+X

Ctrl-X

Ctrl+X

^X

C-x

In nano, along the bottom of the screen you’ll see ^G Get Help ^O WriteOut. This means that you can use Control-G to get help and Control-O to save your file.

nano doesn’t leave any output on the screen after it exits, but ls now shows that we have created a file called draft.txt:

command:

$ ls

draft.txt

-We have seen how to create text files using the nano editor. Now, try the following command:

$ touch my\_file.txt

What did the touch command do? When you look at your current directory using the GUI file explorer, does the file show up?

The touch command generates a new file called my\_file.txt in your current directory. You can observe this newly generated file by typing ls at the command line prompt. my\_file.txt can also be viewed in your GUI file explorer.

Use ls -l to inspect the files. How large is my\_file.txt?

When you inspect the file with ls -l, note that the size of my\_file.txt is 0 bytes. In other words, it contains no data. If you open my\_file.txt using your text editor it is blank.

When might you want to create a file this way?

Some programs do not generate output files themselves, but instead require that empty files have already been generated. When the program is run, it searches for an existing file to populate with its output. The touch command allows you to efficiently generate a blank text file to be used by such programs.

-mv command can be used to rename and move it to another location

since mv will silently overwrite any existing file with the same name, which could lead to data loss. An additional option, mv -i (or mv --interactive), can be used to make mv ask you for confirmation before overwriting

-mv text\_results/result1.txt .

this will move result1.txt file from test\_results folder to current directory as a single dot represents cwd

-mv /home/saivinil\_pratap/test1/quote.txt ./test2/

let us think there is a folder called test1 in which we have a folder test2 and a file quote.txt,the cwd is /home/saivinil\_pratap/test1/ and the above command will move the quote file from test1 to test2

-mv quote.txt ../../test2

this will move quote file in current working directory and moves to its grand parent directory and checks for test2 folder and if it exists it will move it to test2 folder

-$ ls thesis thesis\_backup

this will list files in thesis and thesis\_backup.

output:

thesis:

quotations.txt

thesis\_backup:

quotations.txt

- let's suppose that there is a file named quote.txt

command ls quote.txt will return quote.txt on terminal indicating that quote.txt is present in the current directory,if it is not present it will return an error

- rm :

it can remove a directory and all its contents if we use the recursive option -r, and it will do so without any confirmation prompts:

$ rm -r thesis

Given that there is no way to retrieve files deleted using the shell, rm -r should be used with great caution (you might consider adding the interactive option rm -r -i).

rm \*.txt (this will remove all the files with .txt extension in cwd)

rm \* .txt (THE SHELL WOULD EXPAND \* TO MATCH EVERYTHING IN THE CURRENT DIRECTORY, SO THE COMMAND WOULD TRY TO REMOVE ALL MATCHED FILES AND AN ADDITIONAL FILE CALLED .TXT)

-move or copy multiple files:

move command:

mv <relative path1 or absolute file path1> <relative path2 or absolute file path2> <relative path3 or absolute file path3> <target directory where the files are to be stored>

copy command:

cp <relative path1 or absolute file path1> <relative path2 or absolute file path2> <relative path3 or absolute file path3> <target directory where the files are to be stored>

NOTE THAT THE LAST ARGUMENT SHOULD ALWAYS BE DIRECTORY

wildcards:

\* is a wildcard, which matches zero or more characters. Let’s consider the shell-lesson-data/molecules directory: \*.pdb matches ethane.pdb, propane.pdb, and every file that ends with ‘.pdb’. On the other hand, p\*.pdb only matches pentane.pdb and propane.pdb, because the ‘p’ at the front only matches filenames that begin with the letter ‘p’.

? is also a wildcard, but it matches exactly one character. So ?ethane.pdb would match methane.pdb whereas \*ethane.pdb matches both ethane.pdb, and methane.pdb.

Wildcards can be used in combination with each other e.g. ???ane.pdb matches three characters followed by ane.pdb, giving cubane.pdb ethane.pdb octane.pdb.

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames before running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example, typing ls \*.pdf in the molecules directory (which contains only files with names ending with .pdb) results in an error message that there is no file called \*.pdf. However, generally commands like wc and ls see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that deals with expanding wildcards.

-The shell does not have a trash bin: once something is deleted, it’s really gone.

-wc:

wc is the ‘word count’ command: it counts the number of lines, words, and characters in files (from left to right, in that order).

If we run the command wc \*.pdb, the \* in \*.pdb matches zero or more characters, so the shell turns \*.pdb into a list of all .pdb files in the current directory:

$ wc \*.pdb

20 156 1158 cubane.pdb

12 84 622 ethane.pdb

9 57 422 methane.pdb

30 246 1828 octane.pdb

21 165 1226 pentane.pdb

15 111 825 propane.pdb

107 819 6081 total

wc -l <filename or wildcard pattern>-> the output shows number of lines per file

wc -m <filename or wildcard pattern>-> the output shows number of characters per file

wc -w <filename or wildcard pattern>-> the output shows number of words per file

note: wc -l : this will never execute as it does not have any filename or wildcard pattern as input.

we can redirect the output and write it in to a file, using the below command:

$ wc -l \*.pdb > lengths.txt

the above command is getting the line count of each file that has pdb extension and the output instead of being displayed on the screen will be redirected to lengths.txt file (which will be saved in the cwd)

-sort:

sorts the file in alphanumeric or numeric categories based on the options passed

1)if the file contains only numbers, sort command will sort in alphanumeric order

ex: sort <filename.extension>

2)if the file contains only numbers to sort it in numerical order you need to pass -n option

ex: sort -n <filename.extension>

3)if the file contains alphanumeric characters then the both above commands deliver same results , the results will be in numerical sort

$ sort -n lengths.txt > sorted-lengths.txt (this will sort the lengths file in numerical sort and the output is written to sorted-lengths.txt)

$ head -n 1 sorted-lengths.txt (Using -n 1 with head tells it that we only want the first line of the file; -n 20 would get the first 20, and so on, similarly tail will print lines from the end of the file, if you want to print nth record alone you have to select first n records using head and then use tail to have nth record, for example of 7th record we can have : head -n 7 | tail -1)

-$ sort -n lengths.txt > lengths.txt

sort -r sorted-lengths.txt (will sort the contents in reverse order)

> and >> operator difference:

> will write the contents after clearing the existing contents in the file

>> will add content in a new line to the existing content

pipe usage:

$ wc -l \*.pdb | sort -n (here the word count of files that end with .pdb will be passed to the next command after the pipe in ascending order)

$ wc -l \*.pdb | sort -n | head -n 1 (in addition to the above command it will return the first line)

cut:

$ cut -d , -f 2 animals.txt

The cut command is used to remove or ‘cut out’ certain sections of each line in the file, and cut expects the lines to be separated into columns by a Tab character. A character used in this way is a called a delimiter. In the example above we use the -d option to specify the comma as our delimiter character. We have also used the -f option to specify that we want to extract the second field (column).

uniq:

The uniq command filters out adjacent matching lines in a file. How could you extend this pipeline (a pipeline operation- [first] | [second] is a pipeline: the output of the first command is used as the input to the second) (using uniq and another command) to find out what animals the file contains

ex:$ cut -d , -f 2 animals.txt | sort | uniq

ex:cut -d, -f 2 animals.txt | sort | uniq -c ( shows the total count of each type of animal in the file,YOU HAVE TO USE SORT BEFORE UNIQ BECAUSE,Repeated lines in the input will not be detected if they are not adjacent)

loops:

Loops are a programming construct which allow us to repeat a command or set of commands for each item in a list. As such they are key to productivity improvements through automation. Similar to wildcards and tab completion, using loops also reduces the amount of typing required (and hence reduces the number of typing mistakes).

ex:$ head -n 5 basilisk.dat minotaur.dat unicorn.dat (here the first 5 lines of all the three files are displayed in the order they are typed)

syntax:

for thing in list\_of\_things

do

operation\_using $thing # Indentation within the loop is not required, but aids legibility

done

explanation:

When the shell sees the keyword for, it knows to repeat a command (or group of commands) once for each item in a list. Each time the loop runs (called an iteration), an item in the list is assigned in sequence to the variable, and the commands inside the loop are executed, before moving on to the next item in the list. Inside the loop, we call for the variable’s value by putting $ in front of it. The $ tells the shell interpreter to treat the variable as a variable name and substitute its value in its place, rather than treat it as text or an external command.

ex:

for file in \*.dat (or you can specify the list of files seperated by space- for file in basilisk.dat minotaur.dat)

> do

> head -n 2 $file | tail -n 1

> done

is same as

for file in \*.dat;do head -n 2 $file | tail -n 1; done

example explanation:

In this example, the list is three filenames: basilisk.dat, minotaur.dat, and unicorn.dat. Each time the loop iterates, it will assign a file name to the variable filename and run the head command. The first time through the loop, $filename is basilisk.dat. The interpreter runs the command head on basilisk.dat and pipes the first two lines to the tail command, which then prints the second line of basilisk.dat. For the second iteration, $filename becomes minotaur.dat. This time, the shell runs head on minotaur.dat and pipes the first two lines to the tail command, which then prints the second line of minotaur.dat. For the third iteration, $filename becomes unicorn.dat, so the shell runs the head command on that file, and tail on the output of that. Since the list was only three items, the shell exits the for loop.

-The shell prompt changes from $ to > and back again as we were typing in our loop. The second prompt, >, is different to remind us that we haven’t finished typing a complete command yet. A semicolon, ;, can be used to separate two commands written on a single line

note:wherever we use variable in a loop, we have to use $ before it

-let's suppose in the current directory we have unicorn .dat file compare and see the difference between below 2 codes

code 1:for filename in "red dragon.dat" "purple unicorn.dat";do head -n 100 "$filename" | tail -n 20;done

code 2:for filename in "red dragon.dat" "purple unicorn.dat";do head -n 100 $filename | tail -n 20;done

code 1 will throw an error:

head: cannot open ‘red dragon.dat’ for reading: No such file or directory

head: cannot open ‘purple unicorn.dat’ for reading: No such file or directory

code 2 will display output for unicorn.dat file as in the list of files that are to be iterated as "purplr unicorn.dat" is seperated by a space and in the do part filename is not covered by double quotes unicorn.dat is read seperately and the output will have

head: cannot open 'red' for reading: No such file or directory

head: cannot open 'dragon.dat' for reading: No such file or directory

head: cannot open 'purple' for reading: No such file or directory

CGGTACCGAA

AAGGGTCGCG

CAAGTGTTCC

CGGGACAATA...

ex:$ for filename in \*.dat

> do

> cp $filename original-$filename

> done

expl: the above example will make copies for each dat file in the cwd

-A loop is a way to do many things at once — or to make many mistakes at once if it does the wrong thing. One way to check what a loop would do is to echo the commands it would run instead of actually running them.

$ for datafile in NENE\*A.txt NENE\*B.txt

> do

> echo cp $datafile stats-$datafile

> done

check the below to loops to understand another issue that is plausible

# Version 1

$ for datafile in \*.pdb

> do

> echo cat $datafile >> all.pdb

> done

# Version 2

$ for datafile in \*.pdb

> do

> echo "cat $datafile >> all.pdb"

> done

explanation:

The second version is the one we want to run. This prints to screen everything enclosed in the quote marks, expanding the loop variable name because we have prefixed it with a dollar sign. It also does not modify nor create the file all.pdb, as the >> is treated literally as part of a string rather than as a redirection instruction.

The first version appends the output from the command echo cat $datafile to the file, all.pdb. This file will just contain the list; cat cubane.pdb, cat ethane.pdb, cat methane.pdb etc.

Try both versions for yourself to see the output! Be sure to open the all.pdb file to view its contents.

nested loops:

$ for species in cubane ethane methane

> do

> for temperature in 25 30 37 40

> do

> mkdir $species-$temperature

> done

> done

expl:

We have a nested loop, i.e. contained within another loop, so for each species in the outer loop, the inner loop (the nested loop) iterates over the list of temperatures, and creates a new directory for each combination.

history:

Another way to repeat previous work is to use the history command to get a list of the last few hundred commands that have been executed, and then to use !123 (where ‘123’ is replaced by the command number) to repeat one of those commands. For example, if Nelle types this:

$ history | tail -n 5

456 ls -l NENE0\*.txt

457 rm stats-NENE01729B.txt.txt

458 bash goostats.sh NENE01729B.txt stats-NENE01729B.txt

459 ls -l NENE0\*.txt

460 history

then she can re-run goostats.sh on NENE01729B.txt simply by typing !458.

few more commands:

Ctrl+R enters a history search mode ‘reverse-i-search’ and finds the most recent command in your history that matches the text you enter next. Press Ctrl+R one or more additional times to search for earlier matches. You can then use the left and right arrow keys to choose that line and edit it then hit Return to run the command.

!! retrieves the immediately preceding command (you may or may not find this more convenient than using ↑)

!$ retrieves the last word of the last command. That’s useful more often than you might expect: after bash goostats.sh NENE01729B.txt stats-NENE01729B.txt, you can type less !$ to look at the file stats-NENE01729B.txt, which is quicker than doing ↑ and editing the command-line.

$ history | tail -n 5 > recent.sh

the last command in the file is the history command itself, i.e., the shell has added history to the command log before actually running it. In fact, the shell always adds commands to the log before running them. Why do you think it does this?

reason:

If a command causes something to crash or hang, it might be useful to know what that command was, in order to investigate the problem. Were the command only be recorded after running it, we would not have a record of the last command run in the event of a crash.

shell scripts:

We are finally ready to see what makes the shell such a powerful programming environment. We are going to take the commands we repeat frequently and save them in files so that we can re-run all those operations again later by typing a single command. For historical reasons, a bunch of commands saved in a file is usually called a shell script, but make no mistake: these are actually small programs.

Not only will writing shell scripts make your work faster — you won’t have to retype the same commands over and over again — it will also make it more accurate (fewer chances for typos) and more reproducible. If you come back to your work later (or if someone else finds your work and wants to build on it) you will be able to reproduce the same results simply by running your script, rather than having to remember or retype a long list of commands.

Let’s start by going back to molecules/ and creating a new file, middle.sh which will become our shell script:

bash:

$ cd molecules

$ nano middle.sh

expl:

The command nano middle.sh opens the file middle.sh within the text editor ‘nano’ (which runs within the shell). If the file does not exist, it will be created. We can use the text editor to directly edit the file – we’ll simply insert the following line

code:

head -n 15 octane.pdb | tail -n 5'

expl:

This is a variation on the pipe we constructed earlier: it selects lines 11-15 of the file octane.pdb. Remember, we are not running it as a command just yet: we are putting the commands in a file.

Then we save the file (Ctrl-O in nano), and exit the text editor (Ctrl-X in nano). Check that the directory molecules now contains a file called middle.sh.

Once we have saved the file, we can ask the shell to execute the commands it contains. Our shell is called bash, so we run the following command:

bash:

$ bash middle.sh

output:

ATOM 9 H 1 -4.502 0.681 0.785 1.00 0.00

ATOM 10 H 1 -5.254 -0.243 -0.537 1.00 0.00

ATOM 11 H 1 -4.357 1.252 -0.895 1.00 0.00

ATOM 12 H 1 -3.009 -0.741 -1.467 1.00 0.00

ATOM 13 H 1 -3.172 -1.337 0.206 1.00 0.00

-passing parameters to a shell script:

What if we want to select lines from an arbitrary file? We could edit middle.sh each time to change the filename, but that would probably take longer than typing the command out again in the shell and executing it with a new file name. Instead, let’s edit middle.sh and make it more versatile:

shell:

$ nano middle.sh

Now, within “nano”, replace the text octane.pdb with the special variable called $1:

code:

head -n 15 "$1" | tail -n 5 (here the $1 refers to the first parameter that is passed along with bash command,double quotes are used here because we have to cover the case of the filename that happens to contain any spaces, ex:$ bash middle.sh octane.pdb, here the $1 is replaced by octane.pdb,)

Inside a shell script, $1 means ‘the first filename (or other argument) on the command line’. We can now run our script like this:

bash:

$ bash middle.sh octane.pdb

output:

ATOM 9 H 1 -4.502 0.681 0.785 1.00 0.00

ATOM 10 H 1 -5.254 -0.243 -0.537 1.00 0.00

ATOM 11 H 1 -4.357 1.252 -0.895 1.00 0.00

ATOM 12 H 1 -3.009 -0.741 -1.467 1.00 0.00

ATOM 13 H 1 -3.172 -1.337 0.206 1.00 0.00

or on a different file like this:

$ bash middle.sh pentane.pdb

output:

ATOM 9 H 1 1.324 0.350 -1.332 1.00 0.00

ATOM 10 H 1 1.271 1.378 0.122 1.00 0.00

ATOM 11 H 1 -0.074 -0.384 1.288 1.00 0.00

ATOM 12 H 1 -0.048 -1.362 -0.205 1.00 0.00

ATOM 13 H 1 -1.183 0.500 -1.412 1.00 0.00

Currently, we need to edit middle.sh each time we want to adjust the range of lines that is returned. Let’s fix that by configuring our script to instead use three command-line arguments. After the first command-line argument ($1), each additional argument that we provide will be accessible via the special variables $1, $2, $3, which refer to the first, second, third command-line arguments, respectively.

Knowing this, we can use additional arguments to define the range of lines to be passed to head and tail respectively:

bash:

$ nano middle.sh

code:

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

We can now run:

bash:

$ bash middle.sh pentane.pdb 15 5

output:

ATOM 9 H 1 1.324 0.350 -1.332 1.00 0.00

ATOM 10 H 1 1.271 1.378 0.122 1.00 0.00

ATOM 11 H 1 -0.074 -0.384 1.288 1.00 0.00

ATOM 12 H 1 -0.048 -1.362 -0.205 1.00 0.00

ATOM 13 H 1 -1.183 0.500 -1.412 1.00 0.00

By changing the arguments to our command we can change our script’s behaviour:

bash:

$ bash middle.sh pentane.pdb 20 5

output:

ATOM 14 H 1 -1.259 1.420 0.112 1.00 0.00

ATOM 15 H 1 -2.608 -0.407 1.130 1.00 0.00

ATOM 16 H 1 -2.540 -1.303 -0.404 1.00 0.00

ATOM 17 H 1 -3.393 0.254 -0.321 1.00 0.00

TER 18 1

This works, but it may take the next person who reads middle.sh a moment to figure out what it does. We can improve our script by adding some comments at the top:

Bash:

$ nano middle.sh

code:

# Select lines from the middle of a file.

# Usage: bash middle.sh filename end\_line num\_lines

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

A comment starts with a # character and runs to the end of the line. The computer ignores comments, but they’re invaluable for helping people (including your future self) understand and use scripts. The only caveat is that each time you modify the script, you should check that the comment is still accurate: an explanation that sends the reader in the wrong direction is worse than none at all

What if we want to process many files in a single pipeline? For example, if we want to sort our .pdb files by length, we would type:

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

(THE FIRST THING THAT HAPPENS HERE IS IT WILL EVALUATE \*.pdb AND WILL FORM A LIST OF ITEMS THAT SATISFY THIS CRITERIA)

because wc -l lists the number of lines in the files (recall that wc stands for ‘word count’, adding the -l option means ‘count lines’ instead) and sort -n sorts things numerically. We could put this in a file, but then it would only ever sort a list of .pdb files in the current directory. If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can’t use $1, $2, and so on because we don’t know how many files there are. Instead, we use the special variable $@, which means, ‘All of the command-line arguments to the shell script’. We also should put $@ inside double-quotes to handle the case of arguments containing spaces ("$@" is special syntax and is equivalent to "$1" "$2" …).

Here’s an example:

$ nano sorted.sh

# Sort files by their length.

# Usage: bash sorted.sh one\_or\_more\_filenames

wc -l "$@" | sort -n

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

9 methane.pdb

12 ethane.pdb

15 propane.pdb

20 cubane.pdb

21 pentane.pdb

30 octane.pdb

163 ../creatures/basilisk.dat

163 ../creatures/minotaur.dat

163 ../creatures/unicorn.dat

596 total

-debug mode

command:$ bash -x do-errors.sh NENE\*A.txt NENE\*B.txt

The -x option causes bash to run in debug mode. This prints out each command as it is run, which will help you to locate errors. In this example, we can see that echo isn’t printing anything. We have made a typo in the loop variable name, and the variable datfile doesn’t exist, hence returning an empty string.

key points:

Save commands in files (usually called shell scripts) for re-use.

bash [filename] runs the commands saved in a file.

$@ refers to all of a shell script’s command-line arguments.

$1, $2, etc., refer to the first command-line argument, the second command-line argument, etc.

Letting users decide what files to process is more flexible and more consistent with built-in Unix commands.

Place variables in quotes if the values might have spaces in them.

grep:

Unix programmers often use the word ‘grep’. ‘grep’ is a contraction of ‘global/regular expression/print’, a common sequence of operations in early Unix text editors. It is also the name of a very useful command-line program.

grep finds and prints lines in files that match a pattern. For our examples, we will use a file that contains three haiku taken from a 1998 competition in Salon magazine. For this set of examples, we’re going to be working in the writing subdirectory:

command:$ grep not haiku.txt

By default, grep searches for a pattern in a case-sensitive way. In addition, the search pattern we have selected does not have to form a complete word, as we will see in the next example.

Let’s search for the pattern: ‘The’.

command:

$ grep The haiku.txt

output:

The Tao that is seen

"My Thesis" not found.

This time, two lines that include the letters ‘The’ are outputted, one of which contained our search pattern within a larger word, ‘Thesis’.

To restrict matches to lines containing the word ‘The’ on its own, we can give grep with the -w option. This will limit matches to word boundaries.

Later in this lesson, we will also see how we can change the search behavior of grep with respect to its case sensitivity.

command:

$ grep -w The haiku.txt

output:

The Tao that is seen

Note that a ‘word boundary’ includes the start and end of a line, so not just letters surrounded by spaces. Sometimes we don’t want to search for a single word, but a phrase. This is also easy to do with grep by putting the phrase in quotes.

command:

$ grep -w "is not" haiku.txt

output:

Today it is not working

We’ve now seen that you don’t have to have quotes around single words, but it is useful to use quotes when searching for multiple words. It also helps to make it easier to distinguish between the search term or phrase and the file being searched. We will use quotes in the remaining examples.

Another useful option is -n, which numbers the lines that match:

command:

$ grep -n "it" haiku.txt

output:

5:With searching comes loss

9:Yesterday it worked

10:Today it is not working

Here, we can see that lines 5, 9, and 10 contain the letters ‘it’.

We can combine options (i.e. flags) as we do with other Unix commands. For example, let’s find the lines that contain the word ‘the’. We can combine the option -w to find the lines that contain the word ‘the’ and -n to number the lines that match:

command:

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

output:

2:Is not the true Tao, until

6:and the presence of absence:

Now we want to use the option -i to make our search case-insensitive:

command:

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

output:

1:The Tao that is seen

2:Is not the true Tao, until

6:and the presence of absence:

Now, we want to use the option -v to invert our search, i.e., we want to output the lines that do not contain the word ‘the’.

command:

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

output:

1:The Tao that is seen

3:You bring fresh toner.

4:

5:With searching comes loss

7:"My Thesis" not found.

8:

9:Yesterday it worked

10:Today it is not working

11:Software is like that.

If we use the -r (recursive) option, grep can search for a pattern recursively through a set of files in subdirectories.

Let’s search recursively for Yesterday in the shell-lesson-data/writing directory:

command:

$ grep -r Yesterday .

output:

data/LittleWomen.txt:"Yesterday, when Aunt was asleep and I was trying to be as still as a

data/LittleWomen.txt:Yesterday at dinner, when an Austrian officer stared at us and then

data/LittleWomen.txt:Yesterday was a quiet day spent in teaching, sewing, and writing in my

haiku.txt:Yesterday it worked

-wildcards:

grep’s real power doesn’t come from its options, though; it comes from the fact that patterns can include wildcards. (The technical name for these is regular expressions, which is what the ‘re’ in ‘grep’ stands for.) Regular expressions are both complex and powerful; if you want to do complex searches, please look at the lesson on our website. As a taster, we can find lines that have an ‘o’ in the second position like this:

command:

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

output:

You bring fresh toner.

Today it is not working

Software is like that.

We use the -E option and put the pattern in quotes to prevent the shell from trying to interpret it. (If the pattern contained a \*, for example, the shell would try to expand it before running grep.) The ^ in the pattern anchors the match to the start of the line. The . matches a single character (just like ? in the shell), while the o matches an actual ‘o’.

find: find command works recursively, finds files with specific properties that match patterns.

command:

$ find .

. on its own means the current working directory, which is where we want our search to start. find’s output is the names of every file and directory under the current working directory. This can seem useless at first but find has many options to filter the output and in this lesson we will discover some of them.

The first option in our list is -type d that means ‘things that are directories’.

Notice that the objects find finds are not listed in any particular order. If we change -type d to -type f, we get a listing of all the files instead:

command:

$ find . -name \*.txt

expl:

We expected it to find all the text files, but it only prints out ./haiku.txt. The problem is that the shell expands wildcard characters like \* before commands run. Since \*.txt in the current directory expands to haiku.txt, the command we actually ran was:

command:

$ find . -name "\*.txt"

expl:

To get what we want, let’s do what we did with grep: put \*.txt in quotes to prevent the shell from expanding the \* wildcard. this will get all the text files in current directory

command:

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

output:

11 ./haiku.txt

300 ./data/two.txt

21022 ./data/LittleWomen.txt

70 ./data/one.txt

21403 total

expl:

When the shell executes this command, the first thing it does is run whatever is inside the $(). It then replaces the $() expression with that command’s output. Since the output of find is the four filenames ./data/one.txt, ./data/LittleWomen.txt, ./data/two.txt, and ./haiku.txt, the shell constructs the command:

MATCHING AND SUBTRACTING:

The -v option to grep inverts pattern matching, so that only lines which do not match the pattern are printed. Given that, which of the following commands will find all files in /data whose names end in s.txt but whose names also do not contain the string net? (For example, animals.txt or amino-acids.txt but not planets.txt.) Once you have thought about your answer, you can test the commands in the shell-lesson-data directory.

command:

find data -name "\*s.txt" | grep -v net

expl: from current working directory it will go in to data directory and then search text files which ends with letter s and does not have substring "net" in it

notes:

-if you want to copy contents of a file to clipboard give:

command:clip.exe < filepath

-if you want to close an active host:

ps -A

it will show list of active ports that are available:

kill all the process using kill command

kill <process name>

-apt update:

lists all available packages and information about packages that can be upgraded

-apt list --upgradable

this command will list the packages for which updates ara available

-apt upgrade -y

upgrades all available packages that we can check using above command (apt list -u or apt list --upgradable)

-to run commands as root user (considering the situation where you know root user credentials)

su -

-if you dont know root user password, then you can use the following sudo command:

sudo -i

-if you want to come put of root user or exit out of root user simply type below command:

exit

-type the following command to check with which user you are running commands:

whoami

-less

this command only displays a portion of the data that is to be returned, if you want to see the remaining portion, then you can press space and continue to navigate unless you want to stop, press q to exit

- to get local host value in linux

ip a

the value you get after inet is your localhost

-curl command:

Linux curl command is used to download or upload data to a server via supported protocols such as HTTP, FTP, IMAP, SFTP, TFTP, IMAP, POP3, SCP, etc. It is a remote utility, so it works without user interaction. The data transfer from one place to another is one of the vital and most used tasks of a computer system.

-to unzip the tar file and save it in a directory use:

command: tar -zxvf <path of file> -C <destination path>

tar -zxvf mletrainings-vinil-0.0.7.tar.gz -C ../..

/temp/

-list biggest files or size order by size:

sudo du -a <path> | sort -n -r | head -n 20

centos:

which python -> to see which python is being used

ls -ls /usr/bin/python\* to check all python versions in centos

# misc:

make file is a simple bash script that is used to compile/run our code files

linux operating system represents everything in a file system, the hardware devices that we attach are also represented as a file. tty is short for teletype, which prints the file name of the standard input, standard input

commonly used commands:

to just print the output of an env variable: echo $<env variable> # here $ is just a sign of the shell prompt, means that shell is ready to accept commands, anything after $ is executed as command

cd ~ : will change cwd to home

.\* is a wild character which will match any character, any number of times. it is used in grep as well

cat /etc/\*-release - used to check which linux distribution are you using, ex: fedora, debian, centos

./ or ../ indicates that we are considering relative path, not absolute

to search for particular option in a particular command, i.e., for --all option in ls command, you can first give "man ls" and then type "/" and then enter "--all" it will show its search results with highlighted --all, similar to ctrl+f in chrome

to search in vi editor type "/" and then give search word

to untar tar.gz files : tar -zxvf <path of the file in quotes>

when using wsl (will already have a seperate space like we have c and d drive and it will be in the computer network afaik) and to go to c or d drive after opening WSL, you need to go to its parent directories (one at a time) untill you'll find mnt folder where you'll find c and d drives

to open wsl (to traverse through its folders ) you need to open file explorer and type \\wsl$\ to go to that path from there you need to traverse something like this : \\wsl$\Ubuntu-20.04\home\saivinil\_pratap

if the letters you are typing on the screen are not visible but terminal is accepting them invisibly, then type the command, stty sane

if a particular port is not running, but if terminal is saying, port is already in use, then you need to run the command- sudo fuser -k <port number>/tcp

## sudo -i

after you tun this command, you can run all the commands (that follow) with run the commands with root privilege (similar to running command prompt in admin mode). to let terminal, get back to normal (user level) mode, type the command *logout* or *exit* and then enter.

## adding or removing permission for a file using chmod

a -> admin or u -> user

g -> group

o -> others

r -> read

w -> write

a -> append

to remove write permission for a group users for a file named temp.txt in current directory.

sudo chmod g-w temp.txt

to add append permission for a group users for a file named temp.txt in current directory.

Sudo chmod g+a temp.txt