# Basics of Shell Scripting

github.com/grapheo12

## Introduction

## What is a Shell?

- Shell is a program that let's you run shell commands.
- A Shell is a command line interpreter, which runs your shell scripts.
- Shell **is not equal** to a Terminal.
- Examples:
    - tcsh
    - fish
    - Bourne - Again Shell (bash)
    - zsh etc.

## What is a Terminal then?

- A terminal emulator is a graphical program that is used to run the shell.
- It emulates, although in a small window and with lots of add-ons, the tty interface.
- Examples:
  - GNOME Terminal
  - Konsole
  - urxvt
  - Alacritty
  - XTerm
  - Termite etc.

## Wait! What is tty?

- tty (stands for Teletype) is text based interface to your operating system.
- On most Linux systems, you can use multiple teletype sessions and they can be accessed by hitting `Ctrl + Alt + F<1-12>` .
- What does a tty launch after you login into one? Yes, your default *shell*.

## Pipes and redirections i

- Everything in Linux is a file. `stdin` , `stdout` and other streams are also (kinda) files.
- Shell commands (we'll see a few in a moment) generally spit there outputs to `stdout` and in some mode take input from `stdin` .
- What this means, is that we can redirect the output of one command to the input of the other without creating an intermediate file.
- To do so we use the pipe ( `|` ).
- For example, suppose we write out something using `echo` and want to print the number of lines in it using `wc` . The command for that will be:

## Pipes and redirections ii

```
echo "Lorel ipsum dolor sit amet.\nSic mundus creatus es" | wc -l
```

- Operators for redirecting input/output from specific files / streams are:
  - `< inputfile` Redirects `stdin` to take input from a file.
  - `> outputfile` Redirects the output to a new file (Existing file is overwritten).
  - `>> outputfile` Appends the output to a file.
- An useful pattern is to redirect `stderr` to `/dev/null` (the black hole of Linux), so that it doesn't pollute your output: `2>/dev/null` .

## Some Basic Commands

## ls i

- This is used to list files.
- Usage: `ls -[Options] [path]`
- If a path is not given, current directory is assumed.
- Path can also contain wildcards. Example: `ls *.pdf` will list all the pdf files in the current directory.
- Options consists of one or a combination of character flags that invoke special functions:
  - `l` : List files with additional metadata
  - `a` : Show hidden files also. (Name begins with `.` )

## ls ii

  - `h` : Show file sizes in MBs and GBs instead of bytes.
  - `t` : Sort by date modified.
  - . . . and many more.
- Multiple options can be combined as: `ls -lah` .

*Exercise:* Open a shell and create a file with the list of all files (both hidden and visible) sorted by the date modified.

## echo and printf i

- These are used to print stuff (Obvious Lol!)
- `printf` supports formatted output like C. `echo` doesn't.
- `echo` is more common in use than `printf` .
- Usage: `echo "some text"` or `printf "format string" "parameters"` .

## echo and printf ii

### Single and double quotes in bash

In bash, both single and double quotes are allowed. However there is subtle difference in behaviour. Inside double quoted string, you can use sub-commands enclosed by `$()` . This is not possible with single quotes. Run:

```
echo "$(ls /bin)"
```

and

```
echo '$(ls /bin)'
```

to see the difference.

## cat, head and tail i

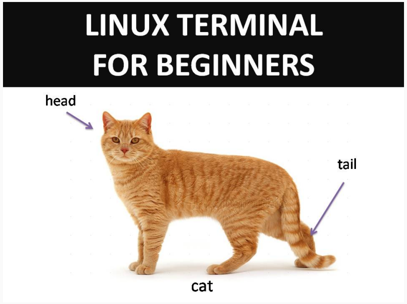

**Figure 1:** This is what these commands mean, literally!

- `cat` stands for `conCATenate`. `cat file1 file2 file3` will output all the 3 files combined in the given sequence.

## cat, head and tail ii

- However, in practice, people use `cat` to print out the 1 full file only.
- `head` prints out the first few lines of a file and `tail` prints out the last few lines of a file.
- Both accept a parameter `-n<Number>`. This limits the output to `Number` number of lines.

Example: to get the first 15 lines of a line, run:

```
head -n15 file
```

## find i

- `find` is quite powerful as a utility.
- Its basic task is to recursively print out all the files and directories from a given path.

```
$ find . # Search starts from current directory
$ find / # Search starts from root
```

- Although there are many filters and actions that find can perform.
- For example, `find -type f` finds only files. Change `f` to `d` and it will find only directories.

## find ii

- `find -name expr` will match `expr` to the file names. `expr` can be a string with wildcards.
- These filters can also be combined.
- The default action is `-print` .
- However `find -delete` will delete all the files it was supposed to print.
- Furthermore, `find -exec` will execute arbitrary command on the file names.

For example:

```
find -name "*.js" -exec rm {} \;
```

## find iii

will delete all js files. Here `{}` is a placeholder for the file name.

- Know more by running `man find` .

## grep i

- `grep` prints those lines in a given list of files that match a pattern.
- Usage: `grep pattern filename` .
- Another common usage is to pipe the output of some other command to grep. For example:

```
cat file | grep kharagpur
```

This will find all lines in a file that have the string “kharagpur” in it.

- `pattern` can be a Regular Expression too. For example:

## grep ii

```
whois google.com | grep [Cc]ountry
```

will fetch the whois record of Google.com using whois CLI (installed separately) and from that record, will find out all string which have either “country” or “Country” in it.

## which

- Every command that you run in the shell actually is an executable located somewhere in your PATH (it is an environment variable, more on that later).
- To find out which particular executable is being run, `which` is used.

```
$ which echo
/usr/bin/echo
```

**Resolving virtualenvs**

While working with multiple Python projects together, one might get confused as to what the current python binary is being used. At that time, running `which python` helps a lot.

## cp, mv and mkdir

- `cp some/path/file some/other/path` copies `file` from `some/path` to `some/other/path` .
- To recursively copy a folder and all its files and subdirectories, we use `cp -r` .
- The main job of `mv` is to move files and folders from one directory to the other.
- Althought `mv file newname` renames the file `file` to `newname` .
- Paths in cp and mv also support wildcards. For example, `cp yt-slides/*.pdf folder2/` copies only the pdf files.
- `mkdir` makes directories. Usage: `mkdir existing/path/new_directory_to_make` . This creates a new directory `new_directory_to_make` under the existing path `existing/path` .
- However, if the parent directory doesn't exist yet, we can create the whole hierarchy

## rm and rmdir

- `rmdir` removes empty directories.
- `rm` is a general command for removal of files and folders.
- To recursively delete, use the `-r` flag with `rm` .

**Warning**

NEVER RUN:

```
sudo rm -rf /
```

## wc, sort, shuf, . . .

These fall under the category of text manipulation programs.

- `wc` returns the newline, word and byte count for each of the files that are passed to it.
- We can get the individual newline, word or byte count by using `-l` , `-w` or `-c` flags respectively.
- `sort` sorts the lines of a document in lexicographical order. Although the ordering can be changed using appropriate flags.
- `sort -u` gives the unique lines in the document.
- `shuf` randomly selects a few lines from a file. The number of lines to take can be passed using `-n<Number>` flag.
- Other programs of this category are: `uniq` , `split` etc.

## wget and curl

- These programs are used to fetch resources from the internet.
- `wget` , as the name suggests, performs only GET requests.
- By default, wget saves the output to a file in the current directory. However, this can be changed using the `-o` flag.
- cURL is a more generic tool. It can be used to perform arbitrary HTTP requests.

For example, sending a POST request to an URL through curl is as follows:

```
curl -X POST -H 'Content-type: application/json'
 -d '{"message": "Hello"}' http://url/endpoint
```

`-X` defines request method, `-H` defines headers, `-b` defines Request body.

## Exercise

Look for the usage of these commands:

- man
- history
- sed
- awk
- top
- xargs
- cut
- time

## Variables and Control Flow

## Variables

- Variables here are not typed.
- All variables, when *USED* should be preceeded by **$** symbol.
- However, while declaring you should never use the $ symbol.

Example:

```
$ a=2      # Don't forget to put no space around equals
$ echo $a # Btw this is a comment.
```

## Environment Variables i

- These are variables that are picked up applications to modify their behaviour.
- Commands to set environment variables are `export` and `set` .
- Exported variables permeate to subshells, whereas set variables do not. It varies by shell, though.
- Environment variables can also be set for a particular program just by prepending it before the program name
- These variables are unset when the shell closes, unless you have specified them in your `.bashrc` .
- You can view all current environment variables by the `env` command.

## Environment Variables ii

Examples:

```
$ export HTTP_PROXY=172.16.2.30
$ set http_proxy=172.16.2.30
$ DRI_PRIME=1 ./android-studio
# DRI_PRIME is used to change video card
```

## Some Special Variables

- `$$` is the PID of the script.
- `$!` is the PID of most recently executed background pipeline.
- In a bash script, `$0` can be used to get the script name. `$i` for i >= 1 can be used to get the argument variables. (Compare with `sys.argv` of Python)
- `$PS1` controls the line shown at each prompt.
- `$PATH` contains a `:` separated list of

## Conditionals i

Bash doesn't use braces or indentation to mark the blocks.

The basic structure of an `if` block is as follows:

## Conditionals ii

```
if [condition1]
then
    # Block to execute
elif [condition2]
then
    # Block
else
    # Block
fi
```

- `[]` are a reference to the `test` command, which is run internally to check for the conditions.

## Conditionals iii

- Normal operators like `=` , `!=` apply to String comparison.
- Integer comparisons are done using `-eq` , `-gt` and `-lt` . (Guess their meaning!).
- `! Expr` negates the expression `Expr` .
- Some special comparisons:
  - `-n str` : Length of string is $> 0$.
  - `-z str` : Length is $== 0$.
  - `-d file` : `file` is an existing directory.
  - `-e file` : `file` exists.
  - `-r file` : `file` exists and the read permission is granted.
  - `-s file` : `file` exists and is not empty.

## Conditionals iv

- `-w file` : `file` exists and the write permission is granted.
- `-x file` : `file` exists and the execute permission is granted.

## for Loops i

- Bash's `for` loop is similar to the Python one.
- Although the concept of array is not present in bash.
- Common patterns:

```bash
for i in var1 var2 var3
do
    # Do something with $i
    # vari can be numbers also
done
```

## for Loops ii

```
for i in $(Command with multiple line output)
do
    # $i will contain one line at a time
done
```

```
for i in {1..5} # {START..STOP} range

for i in {1..5..2} # {START..STOP..STEP} range

for i in $(seq 1 100) # 1 to 100 sequence
```

## while Loops i

- `while` loops iterate while their conditions are true.
- Syntax:

```
while [condition]
do
    # Something to do
done
```

- A common pattern observed while using the `while` loop is incrementing the variables. This is done as shown: `x=$(( $x + 1 ))` .
- Another common usage of `while` is with the `read` command:

## while Loops ii

```
while read p
do
    # Something with $p
done
```

This reads from stdin line by line until EOF is received.

- `while` with no condition is an infinite loop.
- `break` and `continue` work as common sense predicts.
- Exercise: Read about `case` statement.

## Putting it all together

## .sh scripts i

- Apart from running from terminal, we can also put our commands in a script file.
- Files can be run as `sh script.sh` .
- To be able to run the script as an executable, we need to set executable flag on it. This is done by:

```
$ chmod +x script.sh
```

- But before that, we need to declare which shell to use to run it.
- This is done on the very first line of the script, by writing:

## .sh scripts ii

```
#!/bin/bash
```

- This is called the `Shebang` line.
- Following the previous 2 steps, one can then execute the script as:

```
$ ./script
```

## .bashrc

- When a shell is loaded on a terminal, you might want to run some commands before hand.
- For example, you might want to change the `PATH` variable so as to include your `java` compiler, or you might want to set the proxy variables.
- This can be achieved by putting the relevant commands in the bash configuration files.
- For user settings, we use `$HOME/.bashrc` .
- For root settings, the files are located in `/etc` .

## Exercises

## Practice is the key

As with any language, you can only know all the nuances once you get your hands dirty.

Teaching with slides merely does half the job.

Here are some problems to ponder.

## Problem 1

From user, take `n` as an input.

Then take `n` numbers as input.

Sort the numbers and display.

## Problem 2

Get the path of all `.py` files in your computer.

Then find out how many times in all these files the `os` module has been imported.