

Software Development in Engineering and Science (SDES) Using Linux Tools

FOSSEE team

(In particular: Puneet Chaganti, C. Madhusudan, Asokan Pichai,
2010)

Department of Aerospace Engineering
IIT Bombay

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

What is the Linux OS?

- Free Open Source Operating System

Free Free as in Free Speech, not Free Beer

Open-Source Permit modifications and redistribution of source code

- Unix-inspired
- Linux Kernel + Application software
- Runs on a variety of hardware
- Also called GNU/Linux

Why Linux?

- Free as in Free Beer
- Secure & versatile

Why Linux for Scientific Computing?

- Free as in Free Speech
- Can run for *ever*
- Libraries
- Parallel Computing

Outline

- 1 Introduction
- 2 Getting Started**
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

Logging in

- GNU/Linux does have a GUI
- Command Line for this module
- Hit `Ctrl + Alt + F1` (learn how to *come out* of that *first!*)
(Please note: this is keyboard dependent, and GNU/Linux distribution specific.)
- Login
- `logout` command logs you out

Where am I?

- Logged in. Where did we reach?
- `pwd` command gives the present working directory

```
$ pwd
/home/user
```

Think of a tree rooted at '/'

```
$
```

is called the 'bash prompt' (or shell prompt).

Type `command argument` at the prompt `$` : i.e.

```
$ command argument
```

You can change the prompt `$` (bash syntax: `$PS1`).

Some commands do not need an argument.

Almost all commands can be provided with additional options:

```
$ command -o1 -o2 arguments
```


What is in there?

- `ls` command lists contents of `pwd`

```
$ ls
```

```
jeeves.rst psmith.html blandings.html Music
```

- Can also pass directory as argument

```
$ ls Music
```

```
one.mp3 two.mp3 three.mp3
```

- The GNU/Linux world is case sensitive.

Commands, arguments, directory names: almost all.

There is a space between command, options, arguments:
some options can be combined.

Avoid spaces in general. In SDES course: spaces (and some more characters) are banned (from filenames)!

New folders

- `mkdir` creates new directories

```
$ mkdir sdes  
$ ls
```

- Special characters need to be escaped OR quoted

```
$ mkdir software\ engineering  
$ mkdir "software engg"
```

- Generally, use hyphens or underscores instead of spaces in names

Moving around

- `cd` command changes the `pwd`

```
$ cd sdes
```

```
$ pwd
```

```
/home/user/sdes/
```

- Alternately written as `cd ./sdes` (`.` : current)
- Specifying path relative to `pwd`
- `..` takes one level **up** the directory structure (`..` : 'parent')

```
$ cd ..
```

- We could use absolute path instead of relative

```
$ cd /home/user/sdes/
```

New files

- `touch` command creates a blank file

```
$ pwd  
/home/user  
$ cd sdes  
$ touch first  
$ ls  
first
```

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help**
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

What does a command do?

- `whatis` gives a quick description of a command

```
$ whatis touch  
touch (1) - change file timestamps
```

- `man` command gives more detailed description

```
$ man touch
```

- Shows all tasks that the command can perform
- Hit `q` to quit the `man` page. (This is syntax of 'less'.)
- For more, see the `man` page of `man`

```
$ man man
```

- `less` is more than `more`.

Using additional options

- `-h` or `-help` give summary of command usage

```
$ ls --help
```

- List out all files within a directory, recursively

```
$ ls -R
```

- Create a new directory along with parents, if required

```
$ pwd
```

```
/home/user/
```

```
$ ls sdes/
```

```
$ mkdir -p sdes/linux-tools/scripts
```

Searching for a command

- `apropos` searches commands based on their descriptions

```
$ apropos remove
```

- Returns a list of all commands that contain the search term
- In this case, we are interested in `rm`, `rmdir`

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling**
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

Removing files

- `rm` is used to delete files

```
$ rm foo
```

- `rm` works only for files; not directories
- Additional arguments required to remove a directory
- `-r` stands for recursive.
- Removes directory and all of it's content

```
$ rm -r bar
```

- `rmdir` can also be used; Explore

Copying Files

- `cp` copies files from one location to another

```
$ cp linux-tools/scripts/foo linux-tools/
```

- New file-name can be used at target location
- `foo` copied to new location with the name `bar`

```
$ cp linux-tools/scripts/foo linux-tools/bar
```

- `cp` overwrites files, unless explicitly asked not to
- To prevent this, use the `-i` flag

```
$ cp -i linux-tools/scripts/foo linux-tools/b  
cp: overwrite 'bar'?
```

Copying Directories

- `-r` is required to copy a directory and all its content
- Copying directories is similar to copying files

```
$ cd /home/user  
$ cp -ir sdes course
```

Moving Files

- `cp` and `rm` would be one way
- `mv` command does the job
- Also takes `-i` option to prompt before overwriting

```
$ cd /home/user
```

```
# Assume we have course directory already created
```

```
$ mv -i sdes/ course/
```

- No prompt! Why?

```
$ ls course
```

- `sdes` became a sub-directory of `course`
- `mv` command doesn't over-write directories
- `-i` option is useful when moving files around
- `mv` to rename — move to same location with new name

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership**
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

Linux File Hierarchy

- / is called the root directory
- It is the topmost level of the hierarchy
- For details `man hier`

Permissions and Access control

- In a multi-user environment, access control is vital
- Look at the output of `ls -l`

```
drwxr-xr-x    5 root users 4096 Jan 21 20:07
```

- The first column shows the permission information
- First character specifies type of the file
- Files have `-`; Directories have `d`
- 3 sets of 3 characters — for user, group and others
- `r`, `w`, `x` — for read, write, execute
- Either the corresponding character or `-` is present

Changing the permissions

- Permissions can be changed by owner of the file
- `chmod` command is used
- `-R` option to recursively change for all content of a directory
- Change permissions of `foo.sh` from `-rw-r--r--` to `-rwxr-xr--`

```
$ ls -l foo.sh
$ chmod ug+x foo.sh
$ ls -l foo.sh
```

Symbolic modes

Reference	Class	Description
u	user	the owner of the file
g	group	users who are members of the file's group
o	others	users who are not the owner of the file or members of
a	all	all three of the above; is the same as <i>ugo</i>

Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified

Mode	Name	Description
r	read	read a file or list a directory's contents
w	write	write to a file or directory
x	execute	execute a file or recurse a directory tree

Changing Ownership of Files

- `chown` changes the owner and group
- By default, the user who creates file is the owner
- The default group is set as the group of the file

```
$ chown alice:users wonderland.txt
```

- Did it work? **Not every user can change ownership**
- Super-user or `root` user alone is empowered

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files**
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

cat

- Displays the contents of files

```
$ cat foo.txt
```

- Concatenates the text of multiple files

```
$ cat foo.txt bar.txt
```

- Not-convenient to view long files

less

- View contents of a file one screen at a time

```
$ less wonderland.txt
```

- q: Quit
- Arrows/Page Up/Page Down/Home/End: Navigation
- ng: Jump to line number n
- /pattern: Search. Regular expressions can be used
- h: Help

- Statistical information about the file
- the number of lines in the file
- the number of words
- the number of characters

```
$ wc wonderland.txt
```

head & tail

- let you see parts of files, instead of the whole file
- `head` – start of a file; `tail` – end of a file
- show 10 lines by default

```
$ head wonderland.txt
```

- `-n` option to change the number of lines

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

- `tail` is commonly used to monitor files
- `-f` option to monitor the file
- `Ctrl-C` to interrupt

```
$ tail -f /var/log/dmesg
```


cut

- Allows you to view only certain sections of lines
- Let's take `/etc/passwd` as our example

```
root:x:0:0:root:/root:/bin/bash
```

- View only user names of all the users (first column)

```
$ cut -d : -f 1 /etc/passwd
```

- `-d` specifies delimiter between fields (default TAB)
- `-f` specifies the field number
- Multiple fields by separating field numbers with comma

```
$ cut -d : -f 1,5,7 /etc/passwd
```

cut

- Allows choosing on the basis of characters or bytes
- Example below gets first 4 characters of `/etc/passwd`

```
$ cut -c 1-4 /etc/passwd
```

- One of the limits of the range can be dropped
- Sensible defaults are assumed in such cases

```
$ cut -c -4 /etc/passwd
```

```
$ cut -c 10- /etc/passwd
```

paste

- Joins corresponding lines from two different files

<code>students.txt</code>	<code>marks.txt</code>
Hussain	89 92 85
Dilbert	98 47 67
Anne	67 82 76
Raul	78 97 60
Sven	67 68 69

```
$ paste students.txt marks.txt
```

```
$ paste -s students.txt marks.txt
```

- `-s` prints content, one below the other
- If first column of marks file had roll numbers? How do we get a combined file with the same output as above (i.e. without roll numbers). We need to use `cut` & `paste` together. But how?

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell**
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

Redirection and Piping

```
$ cut -d " " -f 2- marks1.txt \  
> /tmp/m_tmp.txt  
$ paste -d " " students.txt m_tmp.txt
```

or

```
$ cut -d " " -f 2- marks1.txt \  
| paste -d " " students.txt -
```

- The first solution used Redirection
- The second solution uses Piping

Redirection

- The standard output (stdout) stream goes to the display
- Not always, what we need
- First solution, redirects output to a file
- `>` states that output is redirected; It is followed by location to redirect

```
$ command > file1
```

- `>` creates a new file at specified location
- `»` appends to a file at specified location

Redirection ...

- Similarly, the standard input (stdin) can be redirected

```
$ command < file1
```

- input and the output redirection could be combined

```
$ command < infile > outfile
```

- Standard error (stderr) is the third standard stream
- All error messages come through this stream
- `stderr` can also be redirected

Redirection ...

- Following example shows `stderr` redirection
- Error is printed out in the first case
- Error message is redirected, in the second case

```
$ cut -d " " -c 2- marks1.txt \  
> /tmp/m_tmp.txt
```

```
$ cut -d " " -f 2- marks1.txt 1> \  
/tmp/m_tmp.txt 2> /tmp/m_err.txt
```

- `1>` redirects `stdout`; `2>` redirects `stderr`

```
$ paste -d " " students.txt m_tmp.txt
```


Piping

```
$ cut -d " " -f 2- marks1.txt \  
  | paste -d " " students.txt -
```

- – instead of FILE asks `paste` to read from `stdin`
- `cut` command is a normal command
- the `|` seems to be joining the two commands
- Redirects output of first command to `stdin`, which becomes input to the second command
- This is called piping; `|` is called a pipe

Piping

- Roughly same as – 2 redirects and a temporary file

```
$ command1 > tempfile
```

```
$ command2 < tempfile
```

```
$ rm tempfile
```

- Any number of commands can be piped together

Tab-completion

- Hit tab to complete an incompletely typed word
- Tab twice to list all possibilities when ambiguous completion
- Bash provides tab completion for the following.
 - 1 File Names
 - 2 Directory Names
 - 3 Executable Names
 - 4 User Names (when they are prefixed with a ~)
 - 5 Host Names (when they are prefixed with a @)
 - 6 Variable Names (when they are prefixed with a \$)

History

- Bash saves history of commands typed
- Up and down arrow keys allow to navigate history
- `Ctrl-r` searches for commands used

Shell Meta Characters

- “meta characters” are special command directives
- File-names shouldn't have meta-characters
- /<>!\$%^&*|{}[]"'`~;

```
$ ls file.*
```

- Lists `file.ext` files, where `ext` can be anything

```
$ ls file.?
```

- Lists `file.ext` files, where `ext` is only one character

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing**
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools

sort

- `sort` can be used to get sorted content
- Command below prints student marks, sorted by name

```
$ cut -d " " -f 2- marks1.txt \  
  | paste -d " " students.txt - \  
  | sort
```

- The default is sort based on the whole line
- `sort` can sort based on a particular field

sort ...

- The command below sorts based on marks in first subject

```
$ cut -d " " -f 2- marks1.txt \  
  | paste -d " " students.txt -\  
  | sort -t " " -k 2 -rn
```

- `-t` specifies the delimiter between fields
- `-k` specifies the field to use for sorting
- `-n` to choose numerical sorting
- `-r` for sorting in the reverse order

grep

- `grep` is a command line text search utility
- Command below searches & shows the marks of Anne alone

```
$ cut -d " " -f 2- marks1.txt \  
| paste -d " " students.txt - \  
| grep Anne
```

- `grep` is case-sensitive by default

grep ...

- `-i` for case-insensitive searches

```
$ cut -d " " -f 2- marks1.txt \  
| paste -d " " students.txt - \  
| grep -i Anne
```

- `-v` inverts the search
- To see everyone's marks except Anne's

```
$ cut -d " " -f 2- marks1.txt \  
| paste -d " " students.txt - \  
| grep -iv Anne
```

tr

- `tr` translates or deletes characters
- Reads from `stdin` and outputs to `stdout`
- Given, two sets of characters, replaces one with other
- The following, replaces all lower-case with upper-case

```
$ cat students.txt | tr a-z A-Z
```

- `-s` compresses sequences of identical adjacent characters in the output to a single one
- Following command removes empty newlines

```
$ tr -s '\n' '\n'
```

tr ...

- `-d` deletes all specified characters
- Only a single character set argument is required
- The following command removes carriage return characters (converting file in DOS/Windows format to the Unix format)

```
$ cat foo.txt | tr -d '\r' > bar.txt
```

- `-c` complements the first set of characters
- The following command removes all non-alphanumeric characters

```
$ tr -cd '[:alnum:]'
```

uniq

- `uniq` command removes duplicates from **sorted** input

```
$ sort items.txt | uniq
```

- `uniq -u` gives lines which do not have any duplicates
- `uniq -d` outputs only those lines which have duplicates
- `-c` displays the number of times each line occurs

```
$ sort items.txt | uniq -u  
$ sort items.txt | uniq -dc
```

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts**
- 10 Control structures and Operators
- 11 Miscellaneous Tools

Shell scripts

- Simply a sequence of shell commands in a file
- To save results of students in `results.txt` in `marks` dir

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

```
mkdir ~/marks
```

```
cut -d " " -f 2- marks1.txt \  
| paste -d " " students.txt - \  
| sort > ~/marks/results.txt
```

Shell scripts ...

- Save the script as `results.sh`
- Make file executable and then run

```
$ chmod u+x results.sh  
$ ./results.sh
```

- What does the first line in the script do?
- Specify the interpreter or shell which should be used to execute the script; in this case `bash`

Variables & Comments

```
$ name=FOSSEE  
$ count=`wc -l wonderland.txt`  
$ echo $count # Shows the value of count
```

- It is possible to create variables in shell scripts
- Variables can be assigned with the output of commands
- **NOTE:** There is no space around the = sign
- All text following the # is considered a comment

echo

- echo command prints out messages

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

```
mkdir ~/marks
```

```
cut -d " " -f 2- marks1.txt \  
| paste -d " " students.txt - \  
| sort > ~/marks/results.txt
```

```
echo "Results generated."
```

Command line arguments

- Shell scripts can be given command line arguments
- Following code allows to specify the results file

```
#!/bin/bash
mkdir ~/marks
cut -d " " -f 2- marks1.txt \
| paste -d " " students.txt - \
| sort > ~/marks/$1
echo "Results generated."
```

- \$1 corresponds to first command line argument
- \$n corresponds to *n*th command line argument
- It can be run as shown below

```
$ ./results.sh grades.txt
```

PATH

- The shell searches in a set of locations, for the command
- Locations are saved in “environment” variable called PATH
- `echo` can show the value of variables

```
$ echo $PATH
```

- Put `results.sh` in one of these locations
- It can then be run without `./`

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators**
- 11 Miscellaneous Tools

Control Structures

- `if-else`
- `for` loops
- `while` loops
- `test` command to test for conditions
- A whole range of tests that can be performed
 - `STRING1 = STRING2` – string equality
 - `INTEGER1 -eq INTEGER2` – integer equality
 - `-e FILE` – existence of `FILE`
- `man page of test` gives list of various tests

if

- Print message if directory exists in `pwd`

```
#!/bin/bash
if test -d $1
then
echo "Yes, the directory" \
$1 "is present"
fi
```

if-else

- Checks whether argument is negative or not

```
#!/bin/bash
if test $1 -lt 0
then
echo "number is negative"
else
echo "number is non-negative"
fi

$ ./sign.sh -11
```


[] - alias for test

- Square brackets ([]) can be used instead of `test`



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

```
if [ $1 -lt 0 ]
```

```
then
```

```
echo "number is negative"
```

```
else
```

```
echo "number is non-negative"
```

```
fi
```

- spacing is important, when using the square brackets

if-else

- An example script to greet the user, based on the time

```
#!/bin/sh  
# Script to greet the user  
# according to time of day  
hour=`date | cut -c12-13`  
now=`date +"%A, %d of %B, %Y (%r)"`  
if [ $hour -lt 12 ]  
then  
mess="Good Morning \  
$LOGNAME, Have a nice day!"  
fi
```

if-else ...

```
if [ $hour -gt 12 -a $hour -le 16 ]
then
mess="Good Afternoon $LOGNAME"
fi
if [ $hour -gt 16 -a $hour -le 18 ]
then
mess="Good Evening $LOGNAME"
fi
echo -e "$mess\nIt is $now"
```

- \$LOGNAME has login name (env. variable)
- backquotes store commands outputs into variables

for

Problem

Given a set of .mp3 files, that have names beginning with numbers followed by their names — 08 - Society.mp3 — rename the files to have just the names. Also replace any spaces in the name with hyphens.

- Loop over the list of files
- Process the names, to get new names
- Rename the files

for

- A simple example of the `for` loop

```
for animal in rat cat dog man  
do  
echo $animal  
done
```

- List of animals, each animal's name separated by a space
- Loop over the list; `animal` is a dummy variable
- Echo value of `animal` — each name in list

```
for i in {10..20}  
do  
echo $i  
done
```

for

- Let's start with echoing the names of the files

```
for i in `ls *.mp3`  
do  
echo "$i"  
done
```

- Spaces in names cause trouble!
- The following works better

```
for i in *.mp3  
do  
echo "$i"  
done
```

tr & cut

- Replace all spaces with hyphens using `tr -s`
- Use `cut` & keep only the text after the first hyphen

```
for i in *.mp3
do
echo $i|tr -s " " "-"|cut -d - -f 2-
done
```

Now `mv`, instead of just echoing

```
for i in *.mp3
do
mv $i `echo $i|tr -s " " "-" \
|cut -d - -f 2-`
done
```

while

- Continuously execute a block of commands until condition becomes false
- program that takes user input and prints it back, until the input is `quit`

```
while [ "$variable" != "quit" ]  
do  
  read variable  
  echo "Input - $variable"  
done  
exit 0
```


Environment Variables

- Pass information from shell to programs running in it
- Behavior of programs can change based on values of variables
- Environment variables vs. Shell variables
- Shell variables – only current instance of the shell
- Environment variables – valid for the whole session
- Convention – environment variables are UPPER CASE

```
$ echo $OSTYPE  
linux-gnu  
$ echo $HOME  
/home/user
```

Environment Variables ...

- The following commands show values of all the environment variables

```
$ printenv | less  
$ env
```

- Use `export` to change Environment variables
- The new value is available to all programs started from the shell

```
$ export PATH=$PATH:$HOME/bin
```

Outline

- 1 Introduction
- 2 Getting Started
- 3 Getting Help
- 4 Basic File Handling
- 5 Linux File Hierarchy, Permissions & Ownership
- 6 Looking at files
- 7 The Command Shell
 - Features of the Shell
- 8 More text processing
- 9 Simple Shell Scripts
- 10 Control structures and Operators
- 11 Miscellaneous Tools**

find

- Find files in a directory hierarchy
- Offers a very complex feature set
- Look at the `man` page!
- Find all `.pdf` files, in current dir and sub-dirs

```
$ find . -name '*'*.pdf'
```

- List all the directory and sub-directory names

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

cmp

- Compare two files

```
$ find . -name quick.c
./Desktop/programs/quick.c
./c-folder/quick.c
$ cmp Desktop/programs/quick.c \
c-folder/quick.c
```

- No output when the files are exactly the same
- Else, gives location where the first difference occurs

diff

- We know the files are different, but want exact differences

```
$ diff Desktop/programs/quick.c \  
c-folder/quick.c
```

- line by line difference between files
- > indicates content only in second file
- < indicates content only in first file

tar

- *tarball* – essentially a collection of files
- May or may not be compressed
- Eases the job of storing, backing-up & transporting files

Extracting an archive

```
$ mkdir extract  
$ cp allfiles.tar extract/  
$ cd extract  
$ tar -xvf allfiles.tar
```

- `-x` — Extract files within the archive
- `-f` — Specify the archive file
- `-v` — Be verbose

Creating an archive

```
$ tar -cvf newarchive.tar *.txt
```

- `-c` — Create archive
- Last argument is list of files to be added to archive

Compressed archives

- `tar` can create and extract compressed archives
- Supports compressions like `gzip`, `bzip2`, `lzma`, etc.
- Additional option to handle compressed archives

Compression	Option
<code>gzip</code>	<code>-z</code>
<code>bzip2</code>	<code>-j</code>
<code>lzma</code>	<code>--lzma</code>

```
$ tar -cvzf newarchive.tar.gz *.txt
```

Customizing your shell

- Bash reads `/etc/profile`, `~/.bash_profile`, `~/.bash_login`, and `~/.profile` in that order, when starting up as a login shell.
- `~/.bashrc` is read, when not a login shell
- Put any commands that you want to run when bash starts, in this file.