



WHY LINUX SHELL & COMMANDS?

Before graphical interfaces were commonplace, computer systems relied on text-based terminals to edit (text) files, and control program execution. Even though they may seem to be a thing from the past, they are very powerful because they allow us to create complex and highly customized applications at the cost of simpler and smaller ones. You can think of it like software building-blocks. Many advanced and high-level programming, or scripting languages, follow such an approach.

Throughout your degree at IST, and very likely while working as a professional in the future, you can be much more efficient in your day-to-day activities by becoming a power user. You'll no doubt be faced with the need to produce automated tasks, "fine-tuned" applications and other programming constructs for which you don't have (and don't need) a graphical interface. For such applications, shell programming is a very powerful and flexible tool that you should consider learning and using.

INTRODUCTION

The best way to learn the Linux command line is as a series of small, easy to follow steps. This Lab is organized as such, with each section building upon the knowledge and skills learned in the previous sections. The goal of this Lab is to work through as many sections as you can, reading the contents carefully and trying out the exercises with the help of your instructor, and subsequently conclude (on your own) whatever you did not have time to complete in class.

Each section is structured in the following format:

- An introduction outlining what you will learn in that section;
- Detailed material including examples;
- A set of activities to help you solidify your knowledge and skills.

The symbol ✂ represents an exercise you should do. Treat the activities as a starting point for exploration. The further you take them, the better you will do.

Unlike the remaining materials for this course, this tutorial was deliberately written in English, so that you get acquainted with the English terms related to the Linux shell.

SOME GENERAL SYNTAX:

- We will refer to Linux in the following pages, but this term actually generalizes to UNIX/Linux. Linux is an offshoot of UNIX and behaves pretty much exactly the same.
- Whenever you see **<something>**, this means that you should replace this with something useful. Replace the whole string of text (including the < and >). If you see something such as **<n>** then it usually means replace this with a number.
- Whenever you see **[something]** this usually means that this something is optional. When you run the command you may put in a string of text or leave it out.

PROBLEM SOLVING AND CREATIVE THINKING

If you wish to succeed with the Linux command line then there are two things you need: problem solving and creative thinking. Here are some basic pointers to help you along the way.

- **Explore and experiment.** Remember, you're learning about a set of building blocks and with them you can build very powerful tools. The examples you will find are intended to be an illustration of how they work, not the only thing you can do with them. We encourage you to tweak the examples and see how they behave. This will give you a much better understanding on how they work. You will have a lot of questions along the way of the form "What if....?" and "Can I ...?" to which we answer, "Give it a go and see what happens." The worst you normally get is an error message -- in which case you read the error message to understand why it didn't work, then have another go. Don't hold back!
- **Read carefully** and don't skip over the fine details. We can't stress this enough. The fine details are important and are often the difference between your command working and not working. If something isn't working then re-read the material carefully and look carefully over what you have typed to make sure you haven't made a typo.

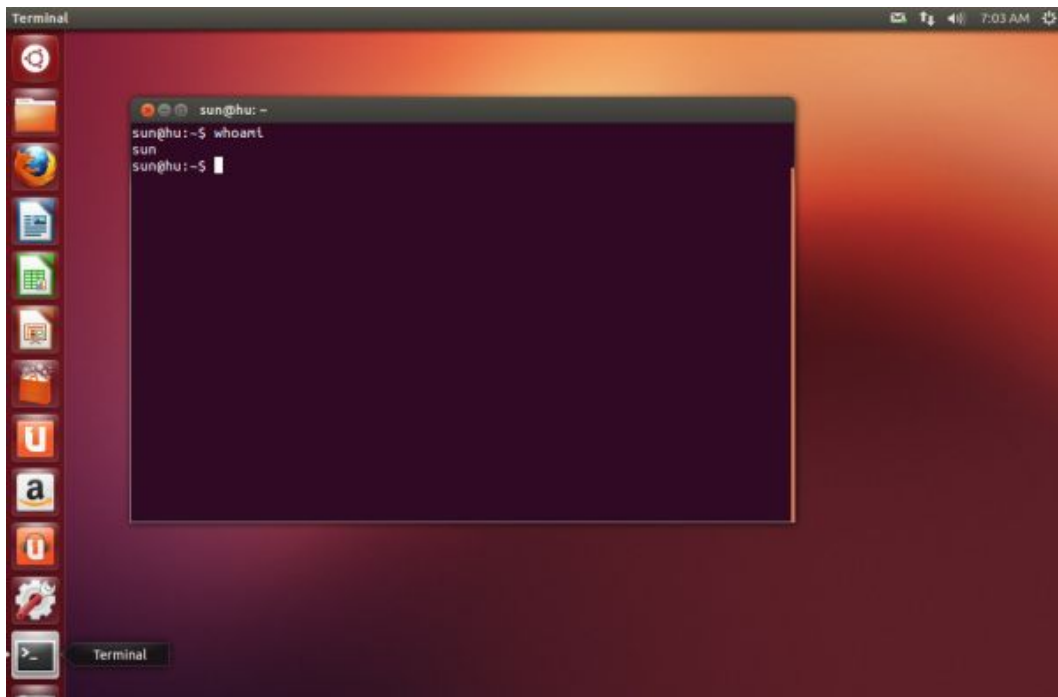
In summary, the general approach to this Lab and to learning about Shell is:

- Create a hypothesis.
- Run your command to test this hypothesis.
- Observe the output. If it is what you were expecting, great. If not then continue.
- Analyze the output and adjust your understanding accordingly.
- Repeat until you reach the desired outcome.

SHELLS

To put it simply, the shell is a program which allows you to run other programs. In each line, you specify the name of the program and its arguments and the shell runs it for you. Before there were GUI interfaces, this was the only way to run programs in computers! Nowadays, it is still very useful, since it allows its users to be more efficient and have greater control of their machines. In UNIX/Linux there are several different shells, and Windows has two common shells: command line (cmd) and the PowerShell. Throughout this tutorial, we will only consider Linux shells: the traditional Windows cmd shell is very limited in its capabilities and you will need to interact with Linux machines in this course. Hence, this tutorial is made considering you are using a Linux machine (eg: the lab computer). However, if you want to make it using your Windows machine, you should first connect through SSH to a computer at IST. In such a case, start by following the steps indicated in the SSH section.

You can access a computer's Shell after you login. Depending on your computer's configuration you have different ways to access it: if you're using a text-based login, either local or remote, once you login you'll be in the Shell; if you're using a graphical interface, you need to launch it manually, usually by running the **terminal** or command-line under accessories or system tools. Different Linux distributions may use different names. To log out of your terminal type exit.



Once the shell starts the first thing you will see is the *user_name@computer_name\$* (or a similar text), this is known as prompt. In front of this line you'll enter your commands interactively.

The behavior of the terminal interface will differ slightly depending on the *shell* program that is being used. Examples:

- Bourne shell (sh):
 - Bourne-Again Shell (bash)
 - Korn Shell (ksh)
 - Z Shell (zsh)
 - [Debian] Almquist Shell (dash/ash)
- C Shell (csh)
- Friendly Interactive Shell (fish)

Depending on the shell used, some extra behaviors can be quite nifty. You can find out what shell you are using by issuing the command: `echo $SHELL`.



Exercise: start a terminal/bash shell and issue the following commands:

```
echo $SHELL
```

The commando `echo` will print the text passed as argument on the terminal. In this example, `$SHELL` is a variable of the shell (called an "environment variable", as we will cover later in this Lab), which is used to hold information about the currently used shell.

Executing the command on your shell, you should get the following:

```
/bin/bash
```

LOOKING FOR HELP: THE `MAN` COMMAND

Even before we start practicing some shell commands, it is important to know where to look for help when things do not seem to work. Of course Google is a few keystrokes away, but UNIX and Linux systems come with their user manual built-in! Most of the commands have a **manual page** which gives useful, often detailed (however sometimes cryptic), descriptions of their usage.

Example:

```
man echo
```

Shows the manual page for the `echo` command.

Press `q` to exit the man page, `SPACE` to advance one terminal screen, and up/down arrows to move the text up and down, respectively. You can further investigate the functionality of man pages by running:

```
man man
```

A simpler explanation for each command is usually (but not always) available by issuing the command with the arguments `--help`. Example:

```
cat --help
```



Exercise:

1. Determine who wrote the `echo` command you're using.
2. What are the options of the `echo` command? What do they do?

Suggestion: consult the manual page for the `echo` command.

LISTING DIRECTORY CONTENTS:

The first set of commands that you're going to learn about are used to list the contents of a directory, navigate through the different directories of the filesystem, i.e., list files and directories stored in the hard drive or other storage devices available.

A **directory** is the Linux/UNIX equivalent to Windows folders. Hence, it is a place where you can store files (or other directories).


```
ls          list the contents of the current directory
ls -l       list a directory in long (detailed) format
ls -a       list the current directory including hidden files. Hidden files start with "."
ls -d       list directory entries instead of contents
ls -i       list inode of each file
ls -l       list one file per line
ls .*       list files begin with .
```

Example (don't worry too much about what everything means):

```
ls -l
drwxr-xr-x  4 cliff  user      1024 Jun 18 09:40 WAITRON_EARNINGS
-rw-r--r--  1 cliff  user      767392 Jun  6 14:28 scanlib.tar.gz
^  ^  ^  ^      ^  ^      ^      ^
|  |  |  |      |  |      |      |
|  |  |  |      | owner  group    size  date  time  name
|  |  |  |      number of links to file or directory contents
|  |  |  |      permissions for world
|  |  |  |      permissions for members of group
| permissions for owner of file: r = read, w = write, x = execute --no permission
type of file: - = normal file, d=directory, l = symbolic link, and others...
```

Open a shell and try `ls` using options in your current directory: `-ld`, `-u`, `-g`, `-r`, `-t`. Use `man ls` to investigate what these options do.

If you don't have/see any files on your current directory, before running the `ls` command, change the current directory by typing: `cd /etc`

 Exercise: List all the contents of your directory sorted by modification time, from the newest to oldest. What are the arguments of the ls command to produce such output?

-rw-----	1	ist14551	14551	1560	Sep 23 08:27	.bash_history
drwxr-xr-x	2	root	root	6144	Sep 21 08:14	..
drwx--x--x	15	ist14551	14551	2048	Sep 18 23:32	.
-rw-----	1	ist14551	14551	809	Sep 18 23:32	.viminfo
-rw-r--r--	1	ist14551	14551	36	Sep 18 22:56	.bashrc
drwxr-xr-x	3	ist14551	14551	2048	Sep 18 22:54	SO1617
drwxr-xr-x	9	ist14551	14551	2048	Jul 21 20:12	old
drwxr-xr-x	2	ist14551	14551	2048	Jul 19 13:58	Desktop
drwxr-xr-x	2	ist14551	14551	2048	Jul 19 13:58	Documents
drwxr-xr-x	2	ist14551	14551	2048	Jul 19 13:58	Downloads
drwxr-xr-x	2	ist14551	14551	2048	Jul 19 13:58	Music

DIRECTORIES:

The file system, which holds all the files and folders of your computer is organized as a tree.

File and directory paths in UNIX use the forward slash "/" to separate directory names in a **path**. Notice that this contrasts with Windows, which uses the backward slash "\" to separate folder names.


Also notice that while Windows organizes folders under the physical hard drives (e.g., C:\, D:\, etc), UNIX organizes all folders (including drives) under a general root directory simply named "/".

Examples:

/	"root" directory
/usr	directory usr (sub-directory of / "root" directory)
/usr/dl	is a subdirectory of /usr
/tmp	directory to store temporary files
/home	directory with home folders for all registered users
/home/ist100	home directory of user ist100; when the shell's user is ist100 running cd without arguments goes to this directory

MOVING AROUND THE FILE SYSTEM:

pwd	Show the "present working directory", or current directory.
cd	Change current directory to your HOME directory.
cd ~	Change current directory to your HOME directory.
cd /usr/IACLEIC	Change current directory to /usr/IACLEIC.
cd INIT	Change current directory to INIT which is a sub-directory of the current directory.
cd ..	Change current directory to the parent directory of the current directory.
cd ~jsmith	Change the current directory to the user jsmith's home directory (if you have permission).

 Exercise: navigate through your file system and list the contents of the following directories. Search online what is the purpose of a few of these directories:

opt	_____	etc	_____
bin	_____	lib	_____
sbin	_____	media	_____
usr	_____	root	_____
dev	_____	sys	_____
proc	_____	boot	_____
srv	_____	home	_____
var	_____	mnt	_____

run _____

tmp _____

hints: <http://www.thegeekstuff.com/2010/09/linux-file-system-structure/>

CREATING AND REMOVING DIRECTORIES:

<code>mkdir dir1 [dir2...]</code>	Create directories
<code>mkdir -p dir1/dir2</code>	Create the directory dir2, including all implied directories in the path.
<code>rmdir dir1 [dir2...]</code>	Remove an empty directory (fails to remove dir1 if it is non-empty)



Exercise:

1. Change the current directory to your home folder: `cd`
2. Create a new directory named IACLEIC: `mkdir IACLEIC`
3. Verify that your newly created directory has been created: `ls`
4. Change the current directory to the new folder: `cd IACLEIC`
5. Confirm you're back in your home directory: `pwd`
6. Create a new folder named Lab1 inside: `mkdir Lab1`
7. Because of the bad directory name, we need to delete it: `rmdir Lab1`
8. Return to your home directory using: `cd ..`
9. Confirm you're back in your home directory: `pwd`

MOVING, RENAMING, AND COPYING FILES:

A set of basic commands to help organizing the file system.

<code>cp file1 file2</code>	copy a file
<code>mv file1 newname</code>	move or rename a file
<code>mv file1 ~/IAC/</code>	move file1 into sub-directory IAC in your home directory.
<code>rm file1 [file2 ...]</code>	remove or delete a file
<code>rm -rf dir1 [dir2...]</code>	recursively remove a directory and all its contents. USE ONLY WITH EXTREME CAUTION!!

ATTENTION: In UNIX there is no trash bin holding deleted files and folders. Hence, when you remove a directory and its contents recursively there is no way to recover them. Moreover, if someone tells you to run `rm -rf /` **DON'T DO IT**. Depending on user permissions, this may erase **everything** in your computer, breaking your OS installation.



Exercise:

1. Copy the file from `/etc/hostname` to your new directory (this file contains the name of your computer):

`cp /etc/hostname ~`

2. Verify the file has been correctly copied using `ls -l`. What's the file size?

3. Rename the newly copied file `hostname` to `computer_name.txt`:

`mv hostname computer_name.txt`

4. Copy the file `/etc/shells` to your home directory, but naming it `shells.txt`:

`cp /etc/shells ~/shells.txt`

CHANGING FILE PERMISSIONS AND ATTRIBUTES

UNIX sets three permission levels for files and directories: (i) User/owner; (ii) group of users to which the owner belongs; (iii) everyone else. Permissions can be changed using command `chmod`, `chgrp`.

You can check each file's permission using `ls -l`.

```
ls -l
drwxr-xr-x    4 cliff    user      1024 Jun 18 09:40 WAITRON_EARNINGS
-rw-r--r--    1 cliff    user      767392 Jun  6 14:28 scanlib.tar.gz
^  ^  ^  ^      ^      ^
|  |  |  |      |      |
|  |  |  |      owner  group
|  |  |  |
|  |  |  | permissions for world
|  | permissions for members of group
| permissions for owner of file: r = read, w = write, x = execute --no permission
type of file: - = normal file, d=directory, l = symbolic link, and others...
```

The attribute value encodes the permissions using 3 digits (0-7) or a character (r=read /w=write/x=execute)

<code>chmod 755 <file></code>	Changes the permissions of file to be rwx for the owner, and rx for the group and the world. (7 = 111 binary = rwx ; 5 = 101 binary = r-x)
<code>chgrp cool kidz <file></code>	Makes file belong to the group <code>cool_kidz</code> .
<code>chown cliff <file></code>	Makes <code>cliff</code> the owner of file.
<code>chown -R cliff dir</code>	Makes <code>cliff</code> the owner of <code>dir</code> and everything in its directory tree

You must be the owner of the file/directory or be root before you can do any of these things. The execute attribute is particularly important to make scripts executable.



Exercise:

Change the attributes of `shells.txt` to be read-write by the owner and the group and read-only by everyone else:

```
ls -l
chmod 664 shells.txt
ls -l
```

INTERACTIVE HISTORY

A feature of `bash` and `tcsh` (and sometimes others) is that you can use the up-arrow keys to access your previous commands, edit them, and re-execute them.



Exercise: Press up-arrow key to retrieve your last entered command. It should show `ls -l` in your prompt. You can edit the command before re-running it again

```
ist14551@sigma01:~$ ls -la # this command shows hidden files
```

You can find your command history by typing the command "`history`", or by reading a file whose name varies according to the type of shell, e.g., `~/.bash_history`.

FILENAME COMPLETION

A feature of `bash` and others is that you can use the `TAB` key to complete a partially typed filename. For example, if you have a file named `introduction-to-computer-architectures.txt` in your directory and want to edit it, you can type `cat intro`, hit the `TAB` key, and the shell will fill in the rest of the name for you (provided the completion is unique).

BASH IS THE WAY COOL SHELL.

Bash will even complete the name of commands and environment variables. And if there are multiple completions, if you hit `TAB` twice bash will show you all the completions. Bash is the default user shell for most Linux systems.

Exercise:

Type: `ech + <TAB> + <SPACE> + $PA + <TAB> + <ENTER>`

REDIRECTION:

It is used to change the input source from the keyboard to a file, and the output from the console to a file. If there are no decisions to be made by the user, this is particularly useful to automate your scripts.

`echo string > newfile` Redirects the output of the echo command to a file 'newfile'

`echo string >> existfile` Appends the output of the echo command to the end of 'existfile'

Using `<` you can redirect the input to come from a file rather than the keyboard.

Exercise: create a file with assorted numbers and sort it:

```
echo 7 > numlist
echo 3 >> numlist
echo 1 >> numlist
echo 4 >> numlist
sort < numlist
```

and the sorted list will be output to the screen.

```
ist14551@sigma01:~$ sort < numlist
1
3
4
7
```

To output the sorted list to a file, type:

```
sort < numlist > sorted_numlist
```

The redirection directives, `>` and `>>` can be used on the output of most commands to direct their output to a file. Command `cat` will show the file's contents:

```
ist14551@sigma01:~$ cat sorted_numlist
1
3
4
7
```

PIPES:

The pipe symbol `"|"` is used to direct the output of one command to the input of another, without using files.

Exercise:

Lists the contents of `/etc` directory, one terminal screen at a time.

```
ls -l /etc | less
```

The `ls` command lists the contents of `/etc` directory and `less` is a pager: you can navigate with the up and down keys, using `q` to exit.

```
total 1412
drwxr-xr-x  3 root root    4096 Jul 27  2015 acpi
-rw-r--r--  1 root root    2986 Sep 14  2009 adduser.conf
-rw-r--r--  1 root root      45 Nov 25  2013 adjtime
drwxr-xr-x  2 root root   20480 Aug 17 13:30 alternatives
```



```
drwxr-xr-x  2 root root    4096 Jul 27  2015 anthy
drwxr-xr-x  3 root root    4096 Jul 27  2015 apache2
drwxr-xr-x  6 root root    4096 Jun  3  01:55 apt
```

Exercise:

Lists the largest files on the /etc directory

```
cd /etc
du -sc /etc/* | sort -n | tail
```

The command "du -sc" lists the sizes of all files and directories in the current working directory. That is piped through "sort -n" which orders the output from smallest to largest size. Finally, that output is piped through "tail" which displays only the last few (which just happen to be the largest) results.

```
ist14551@sigma01:~$ du -sc /etc/* | sort -n | tail
du: cannot read directory e/etc/audispf: Permission denied
du: cannot read directory e/etc/auditf: Permission denied
du: cannot read directory e/etc/lost+foundf: Permission denied
du: cannot read directory e/etc/mysql/.oldstufff: Permission denied
du: cannot read directory e/etc/polkit-1/localauthorityf: Permission denied
du: cannot read directory e/etc/ssl/privatef: Permission denied
224    /etc/texmf
232    /etc/ImageMagick-6
296    /etc/ssh
336    /etc/sane.d
348    /etc/X11
352    /etc/init.d
632    /etc/joe
732    /etc/mono
1336   /etc/ssl
10680  total
```

THE UNIX PHILOSOPHY

Ken Thompson, the creator of Unix, laid out a philosophy for creating minimalistic programs which **do one thing, and one thing well**. This way, each program is **simple** and easily understood. To achieve more complex functionality, it is possible to compose different programs (possibly using pipes).

This principle can be extended to most everything in software engineering. Remember this when building your own systems, programs, libraries, modules and functions!

ENVIRONMENT VARIABLES

You can have your shell to remember things for later, using environment variables via the command `export: export <environment variable>=<variable value>.`

DISCLAIMER: when trying out some of the commands in this section, you may inadvertently break the configuration of your shell session. If something seems wrong, try closing the shell and opening it again (this will make sure that the environment variables are reset to their default)



Exercise: If you are using the bash shell (this particular command varies from shell to shell) type the following:

```
export IAC=IACLEIC
```

The terminal won't produce any output or confirmation that the command was successfully completed. You can check its value using the echo command as shown above:

```
ist14551@sigma03:~$ echo $IAC
IACLEIC
```

By prefixing \$ to the variable name, you can evaluate it in any command.
To know the value of **all** environment variables, run the `env` command.

```
ist14551@sigma03:~$ env
TERM=xterm
SHELL=/bin/bash
SSH_CLIENT=146.193.44.91 51213 22
SSH_TTY=/dev/pts/0
USER=ist14551
IAC=IACLEIC
MAIL=/var/mail/ist14551
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
PWD=/afs/ist.utl.pt/users/5/1/ist14551
LANG=en_US.UTF-8
KRB5CCNAME=FILE:/tmp/krb5cc_14551_1kU9jJ
SHLVL=1
HOME=/afs/ist.utl.pt/users/5/1/ist14551
MATHEMATICA_HOME=/usr/local/Wolfram/Mathematica/10.2
LOGNAME=ist14551
```

One important environment variable is `PATH`. It allows for executing programs under different directories other than the current directory. All directories that hold the commands you want to execute need to be on the `$PATH` variable, separated by a colon (:). No spaces are allowed here unless they are preceded by the escape character \.



Exercise: List the first 9 paths available on your `PATH` variable:

_____	_____	_____
_____	_____	_____
_____	_____	_____

At this point we're not interested in changing the `PATH`, but here are examples on how to do it:

How to create a new `PATH`:

```
export PATH=/home/user/bin:/usr/games
```

How to append a new path to the existing `PATH` variable (to avoid deleting/replacing the existing ones):

```
export PATH=$PATH:/usr/games
export PATH=$PATH:/usr/more\ games
```

LET'S BUILD A SHELL SCRIPT!

Shell scripts are probably the main advantage of shells when compared to their graphical counterparts. A script is a file (by convention with the `.sh` extension) which contains a series of shell commands. This is very useful because now we can automate tedious tasks.

Let's begin by creating a directory to store our scripts:

```
cd ~
mkdir bin
cd bin
```

Let's create a toy script which accepts a name as a command line argument and prints out the name as James Bond does, waits for a bit, and then says farewell. Open file "bond_hello.sh" in your editor of choice.

```
#!/usr/bin/bash

echo "My name is $2, $1 $2"
sleep 5
echo "Farewell"
```

Let's break this down. The first line is what is called a [shebang](#). This tells the OS what program should be used to interpret the program. If you don't quite understand what this means, don't worry: just make sure that all your scripts begin with this line.

The next line is familiar, but they have the weird `$1` and `$2` symbols. This is because our `bond_hello` script requires two arguments: the name and surname of the user. `$1` means *the value of the first command line argument*. `$2` is the second, etc. This line outputs the famous Bond hello. Next, the script waits for 5 seconds, and then says "Farewell".

Now, if we want to run this, we should just type `./bond_hello.sh James Bond`!

However, if you type this in, you'll soon find this doesn't work (probably saying something like "Permission Denied"). If you run `ls -l` you can find that this file is **not executable**. Let's make it so, with `chmod`:

```
chmod +x bond_hello.sh
```

Now, when we run `./bond_hello.sh James Bond`, it does what we expected! Let's try to run it from another directory:

```
cd ~  
./bond_hello.sh
```

This will not work, since the `bond_hello.sh` file is not in the current directory. We can try to run it with its full path (`~/bin/bond_hello.sh`), but this is very cumbersome. What we have to do is place the `~/bin` directory in the `PATH`: this way, the OS knows to look there for our script.

```
export PATH=$PATH:~/bin  
bond_hello.sh PATH variable
```

Note that if you exit the shell and re-enter, the `PATH` variable will be reset. If you want to add the `~/bin` directory to the `PATH` permanently, add the `export` command to your `.bashrc` file.


There is a lot more to learn about shell scripting (this script is funny, but not very useful). You can do basically anything with scripts; after all, shell scripting is a full programming language! The best way to learn is by doing increasingly complex scripts: whenever you find a task you are doing over and over again, create a small script for it. If you run into a problem, Google probably has the answer.

ALIAS

This is used to replace a frequently used long command with a shorter one, to help you execute commands faster. If you're using bash shell, it may be worth editing the `.bashrc` in your home directory to include a set of aliases, so that they are loaded automatically when the shell starts.

```
alias name='command'
```

Assign a name to a command, or set of commands.

 **Exercise:** create an alias for the `"ls -la"` command, using only one character.

```
$ alias d='ls -la'  
  
$ d  
total 8  
drwxrwxrwt+ 1 rpd None 0 Sep 12 15:44 .  
drwxr-xr-x+ 1 rpd None 0 Sep 7 16:44 ..  
drwxr-xr-x+ 1 rpd None 0 Sep 12 15:44 rpd
```

VIEWING AND EDITING FILES ON-SCREEN:

<code>clear</code>	Clears the terminal screen
<code>cat filename</code>	Dumps a file to the screen in ascii.

<code>more filename</code>	Progressively dumps a file to the screen: ENTER = one line down SPACEBAR = page down q=quit
<code>less filename</code>	Like more, but you can use Page-Up too. Not by default on all systems.
<code>head filename</code>	Shows the first few lines of a file
<code>head -n filename</code>	Shows the first n lines of a file.
<code>tail filename</code>	Shows the last few lines of a file.
<code>tail -n filename</code>	Shows the last n lines of a file.
<code>vi filename</code>	Edits a file using the vi editor. All UNIX systems will have vi in some form.
<code>emacs filename</code>	Edits a file using the emacs editor. Not all systems will have emacs.
<code>file filename</code>	Determine type of FILES

SSH: ACCESSING REMOTE MACHINES

In this course, you may need to access a remote machine. A remote machine is any machine which isn't the one you are physically interacting with directly. At IST, there is a cluster (group of computers) in which every student has an account: sigma.

To log into a remote machine, you can use ssh: the secure shell. This opens a connection to the remote machine and gives you a shell to interact with it (note: you don't have a GUI to fall back on in this case!). The syntax is the following:

```
ssh username@machine_name
```

Let's log into sigma (note that you need to add the .tecnico.ulisboa.pt):

```
ssh istXXXXX@sigma.tecnico.ulisboa.pt
istXXXXX@sigma01:~$ # we now have a shell in sigma!
```

One important use-case for this is that the files in the sigma cluster are shared with all the computers in our labs. This means that if you add a file in one of the computers in the labs, if you log onto a different computer or in sigma, the file will be there. There is a command which goes hand in hand with ssh: scp (secure copy). The syntax is similar to a mixture of the cp command and the ssh command:

```
scp source destination
```

Either the source or the destination can be in the remote machine. Here is an example of sending a new file to the sigma cluster:

```
echo "I can send files over SSH" > message.txt
scp message.txt istXXXXX@sigma.tecnico.ulisboa.pt:~
```

Note that we specified the username and the name of the machine and then a colon (:) and a path. That is the path in the remote machine where the file will be stored.

NOTE: If you're on Windows, you can SSH from the PowerShell. The syntax is the same.



Exercise: ssh into sigma and check that the file you sent is there.

OTHER USEFUL COMMANDS

SEARCHING FOR STRINGS IN FILES

To search for strings in files there's a command named grep, and is used as follows:

```
grep string filename
```

It prints all the lines in a file that contain the string.

grep is one of many standard UNIX utilities. It searches files for specified words or patterns. The grep command is case sensitive; it distinguishes between Science and science.

Let's first create a file to grep into.

```
lscpu > cpu.txt # lscpu prints information about the CPU model
```

To ignore upper/lower case distinctions, use the -i option, i.e. type


```
grep -i cpu cpu.txt
```

To search for a phrase or pattern, you must enclose it in single quotes (the apostrophe symbol). For example to search for spinning top, type:

```
grep -i 'cpu mhz' cpu.txt
```

Some of the other options of grep are:


- v display those lines that do NOT match
- n precede each matching line with the line number
- c print only the total count of matched lines

 Exercise: Try some of them and see the different results. Don't forget, you can use more than one option at a time. For example, the number of lines without the words cpu, Cpu or CPU is

```
grep -ivc cpu cpu.txt
```

SEARCHING FOR FILES : THE *FIND* COMMAND

find search_path -name filename	
find . -name cpu.txt	Finds all the files named cpu.txt in the current directory or any subdirectory tree.
find / -name iac	Find all the files named iac anywhere on the system.
find /usr/local/games -name "*pacman*"	Find all files whose names contain the string 'pacman' and exist within the '/usr/local/games' directory tree.

 Exercise: Find all files named passwd inside /etc directory:

```
find /etc -name passwd
```

DIFF

Diff tells you the differences between files.

```
% diff file1 file2          (basic use of diff)
% diff -i file1 file2       (tell diff to ignore case)
```

Try options: -c, -e, -f, -h, -n

 Exercise: Identify the differences between two text files

1. Create a text file saying hello

```
echo "Good Morning, Vietnam" > file1.txt
```

2. And another saying goodbye.

```
echo "Good Evening, Jack" > file2.txt
```

3. Compare the two files

```
diff file1.txt file2.txt
```

You should get the identification of the lines on the new file that differ from the original one.

```
ist14551@sigma01:~$ diff file1.txt file2.txt
1c1
< Good Morning, Vietnam
```

```
---  
> Good Evening, Jack
```

Try to figure out what is the meaning of the numbers and letters before each line and the comparison signs (1c1, >, <)
Try:

```
diff file2.txt file1.txt
```


Compare with the previous execution.

ZIP, UNZIP

`zip` reduces the size of the given file while `unzip` reverses the effect and expands the compressed file back to its original form. The default extension `.zip` is used for compressed files. Here's an example on how compress and uncompress are used.

```
% zip file.zip file           (file.zip contains file)  
% unzip file.zip              (file.zip becomes file)  
  
% zip -r dir.zip dir          (directories need to be recursively zipped)  
% unzip dir.zip               (same as above)
```

There are other compression utilities such as `bzip2`, `gz`, `tar`. Investigate them.

 Exercise: Compress the largest file in /boot to your home directory:

Determine the largest file in /boot:

```
du -sc /boot/* | sort -n | tail -n 2
```

Compress it to your home folder

```
ist14551@sigma01:~$ zip boot.zip /boot/initrd.img-4.2.0-1-grsec-amd64
  adding: boot/initrd.img-4.2.0-1-grsec-amd64 (deflated 0%)
ist14551@sigma01:~$ ls -la boot.zip
-rw-r--r-- 1 ist14551 14551 17961758 Sep 23 14:06 boot.zip
```

Decompress it to your current folder (bear in mind that the directory will be reconstructed in your local folder.

```
ist14551@sigma01:~$ unzip boot.zip
Archive:  boot.zip
  inflating: boot/initrd.img-4.2.0-1-grsec-amd64
```

Now, let's compare the two files to see if they match. You're going to do this using MD5. MD5 is a checksum computed from a file's contents (think of it like the sum of bytes in the file), which is different for any two files even though they only differ in one bit!

Usage: md5sum file1 file2

```
ist14551@sigma01:~/boot$ md5sum initrd.img-4.2.0-1-grsec-amd64
/boot/initrd.img-4.2.0-1-grsec-amd64 0b86d34e6b2fb2c6804fe64277756d6b
0b86d34e6b2fb2c6804fe64277756d6b initrd.img-4.2.0-1-grsec-amd64
0b86d34e6b2fb2c6804fe64277756d6b /boot/initrd.img-4.2.0-1-grsec-amd64
```

Since the checksum is the same for both, the two files are identical.

COMMAND SUBSTITUTION

You can use the output of one command as an input to another command in another way called command substitution. Command substitution is invoked by enclosing the substituted command in backwards single quotes. For example:

```
cat `find . -name aaa.txt`
```

which will cat (dump to the screen) all the files named aaa.txt that exist in the current directory or in any subdirectory tree.

The shell is only one of the tools at your disposal to make your life easier and make you more efficient. There are many many other tools which will help you in both your academic and professional career. The following two are particularly useful:

- Editors: while coding, you'll be editing source code files most of the time. Mastering a good editor (i.e., getting to know its commands and shortcuts) is time well spent.
 - **Vim** (vi improved) is a text based editor (i.e., runs in the terminal) and one of the classics. It has a steep learning curve, but it is very much worth it! If you are logging into a remote machine and lacking a GUI, it is very useful. Even if you have a GUI, it is very comfortable to use. To learn `vim`, it's best to start with `vimtutor`, a program installed with `vim`, which is a first tutorial to the editor.
 - **Visual Studio Code** is a GUI based editor, and it is a free version of Microsoft's Visual Studio IDE. It has a familiar feel, and has become very popular in recent years.
- Version Control: ever had "project.final.doc", "project.final.final.doc", "project.final.final2.doc", "project.final.now_this_time_for_real.doc" files lying around? **git** is a version control system for projects. It allows you to go back and forward in the history of your project, including sharing the project with your peers. To learn git, it's best to start with the book: <https://git-scm.com/book/en/v2>

We cannot teach these (and many other things we would like to) in this course. However, MIT has a course only about this, with its classes and exercises publicly available at <https://missing.csail.mit.edu/>. We encourage you to check it out and learn more tools to help you in your career!

OTHER TUTORIAL SOURCES

1. <http://ryanstutorials.net/linuxtutorial/>
2. <http://freeengineer.org/learnUNIXin10minutes.html>
3. <http://www.cs.toronto.edu/~maclean/csc209/unixtools.html>
4. <http://www.doc.ic.ac.uk/~wik/UnixIntro/>