# COMP206-Basics of Linux

$ ls -l /bin

command line argument

Program Name/

Command

(always the first)

Second Argument

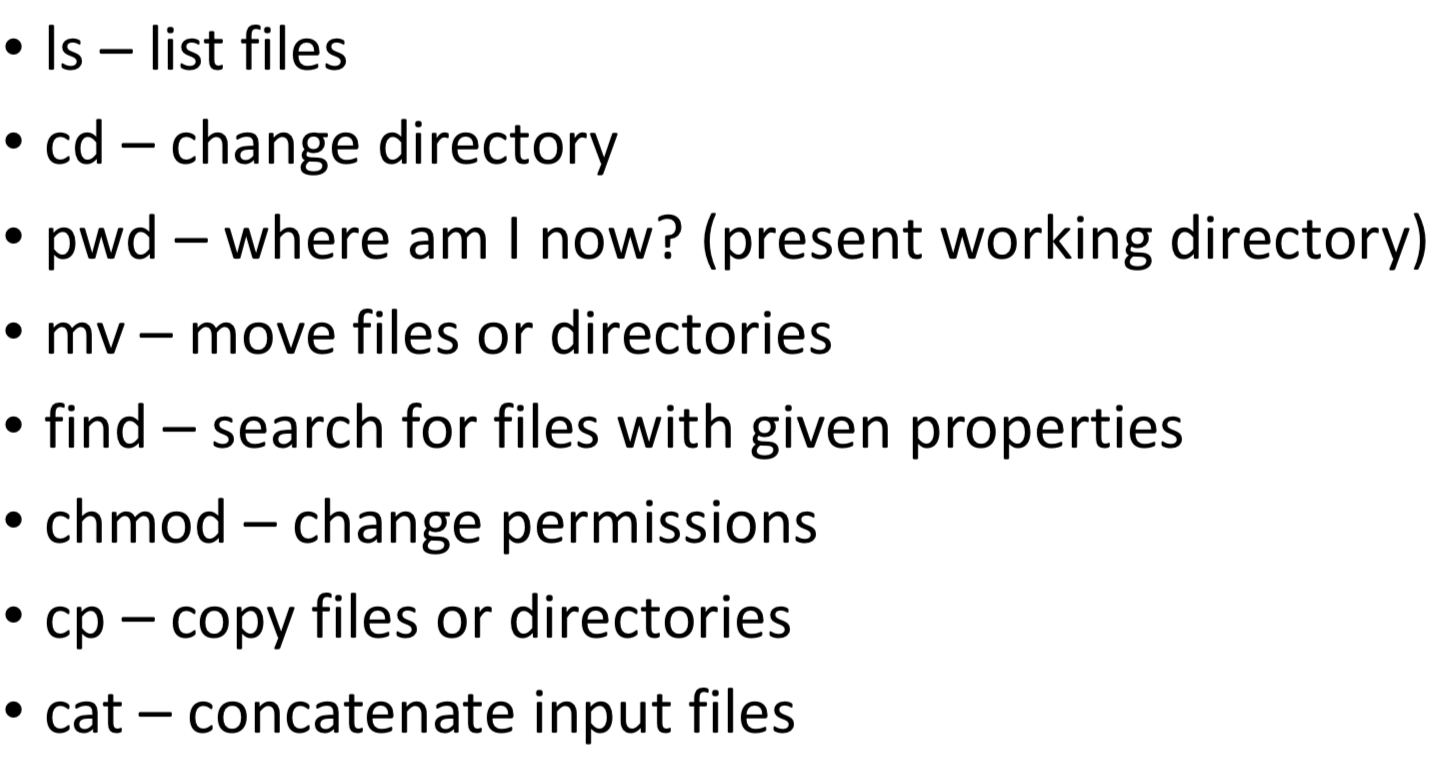
(this time a file name)

First Argument

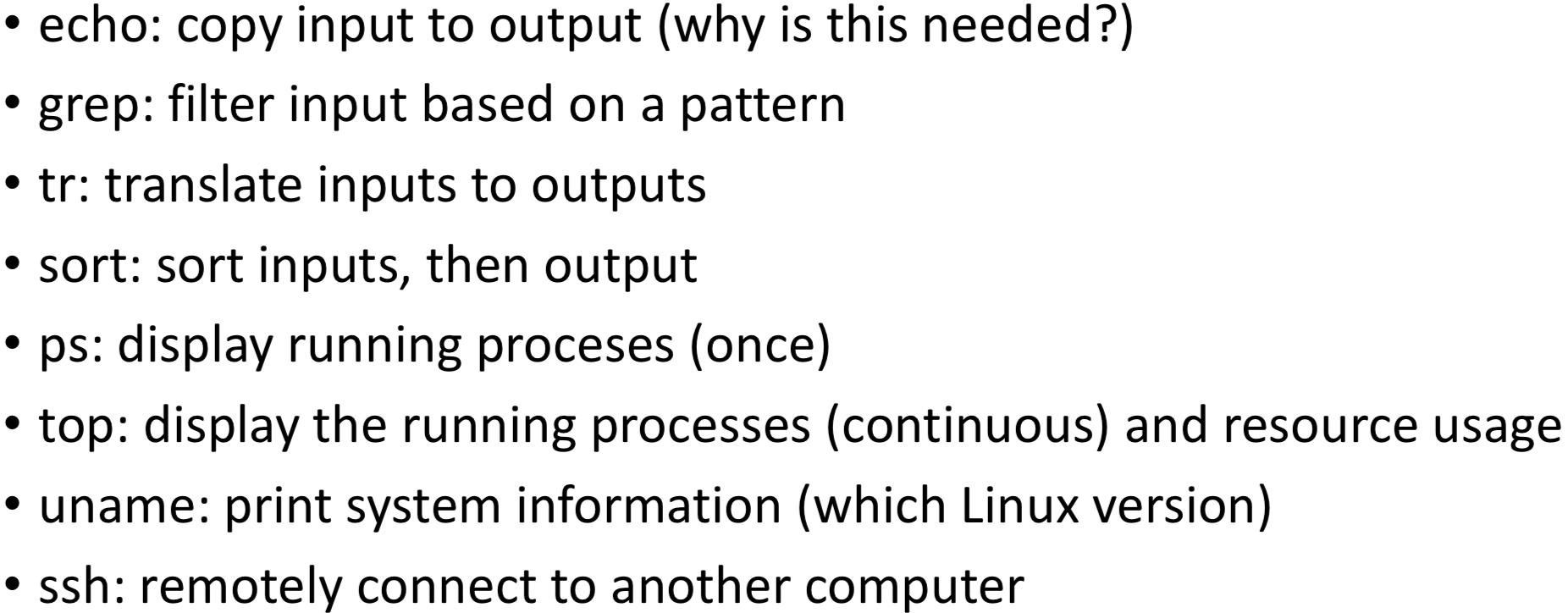
(this time a flag or option)

Note: Command line arguments are **separated by spaces**, as well as the command and the first command line argument.

[File related commands]



[More Commands]



Tools: **man command**– A linux command that shows the manual page for other commands!

**$ ls –help**, the "--help" argument.

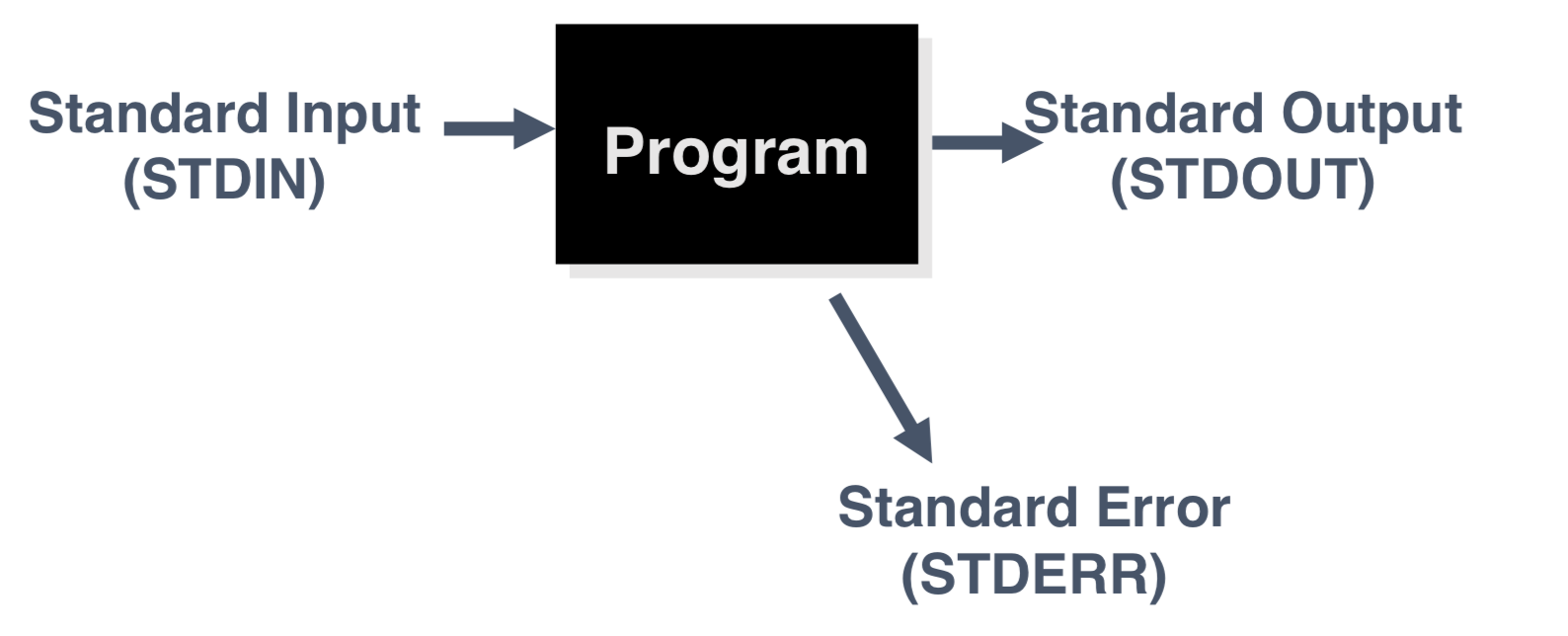
*When you enter commands, they are actually stored in a history. You can traverse this history using the up and down arrow keys. So don't bother re-typing out commands you have previously entered, you can usually just hit the up arrow a few times. You can also edit these commands using the left and right arrow keys to move the cursor where you want.*

Other Sources of Input—Standard I/O

*Arguments are given once, parsed by the program as it begins to modify its operation, but it is impossible to change or add to them once we press enter.*

Many programs require input during their operation. We saw this for tr, cut and sort. Other uses include e.g. a text-editor, the shell itself, a username/password entry dialog.

Linux calls this type of input **“standard input”** and gives us flexible options for where our programs get **standard input**. By changing these, we can start to build up more complex commands and programs.



Programs and Standard I/O

* Defaults for I/O

When a shell runs a program for you: standard input is your keyboard.

standard output is your screen/window.

standard error is your screen/window.

* Terminating Standard Input

If standard input is your keyboard, you can type stuff in that goes to a program. To end the input you press **Ctrl-D (^D)** on a line by itself, this ends the input stream.

What happens when you give the shell ^D?

* Input Redirection

The shell can attach things other than your keyboard to standard input.

**• A file, using the “<“ operator:**

E.g. “$ grep pattern < search\_file.txt”.

The contents of the file are fed to a program as if you typed it.

**• The output of another command, using the “|” operator:**

E.g. “$ ls | grep”

The output of another program is fed as input as if you typed it. Note that both programs can run simultaneously and continue to transmit information over their “pipe”.

* Output Redirection

The shell can attach things other than your screen to standard output (or stderr).

**• A file, using the “>” operator:**

E.g., $ ls > file\_info.txt

The output of a program is stored in file

**• The input of another program, using a pipe as we have seen on the previous slide**

Note: The command $ ls > foo.txt will create a new file named foo and **delete any existing file named foo**.

If you use >>, the output will be appended, leaving any contents that were in the file previously and adding the new output at the end (>> still creates the file if it wasn’t there previously):

E.g. $ ls /etc >> foo.txt

**You can do both!** E.g. $ sort < nums > sortednums

$ tr a-z A-Z < letter > rudeletter

* Errors

Sometimes we do not want them ending up in our re-direct file, so the default behavior of “>” is that errors stay on the terminal and only standard output enters the file.

• “1>” means to send standard output only (same as “>”)

• ”2>” means to send standard error only (std out might stay on terminal)

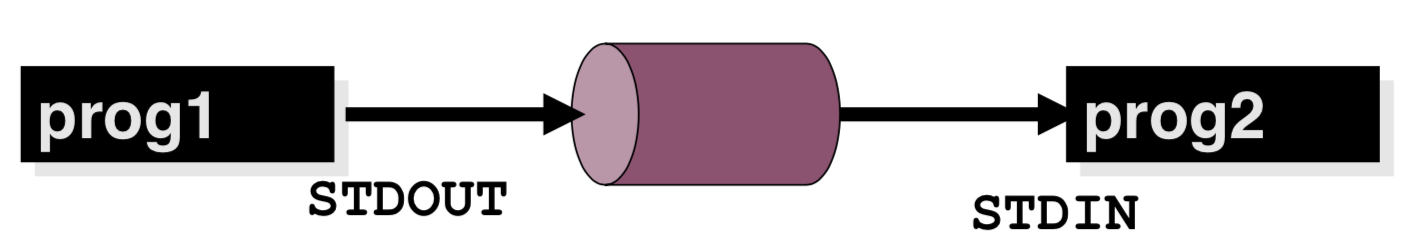
• “&>” means to send both standard error and output

Note: It’s OK to add both ”1>” and “2>” giving different files. Do not name the same file or you only get standard out.

* Pipe

A pipe is a holder for a stream of data.

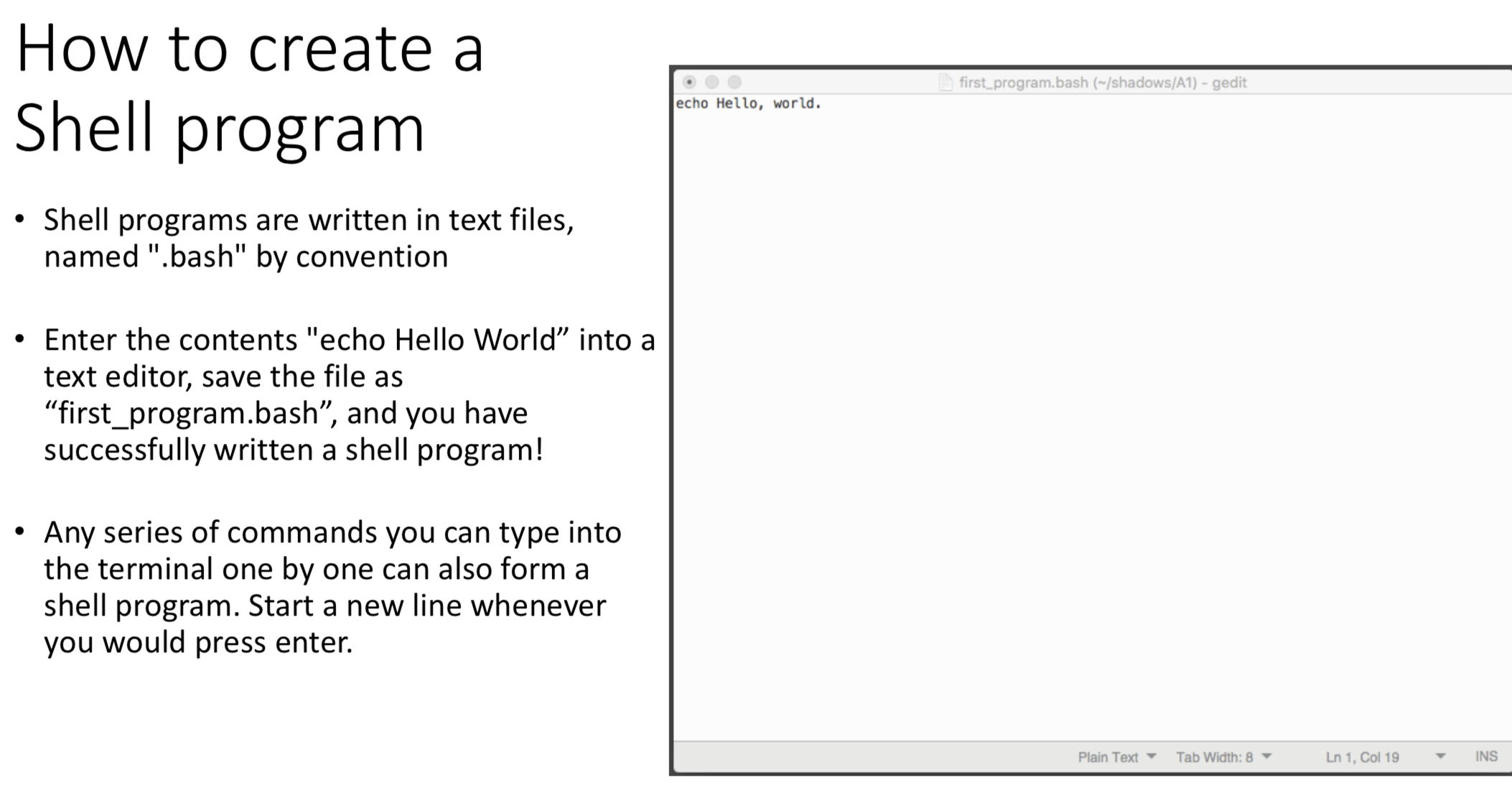
A pipe can be used to hold the output of one program and feed it to the input of another.



Separate 2 commands with the “|” character.

$ ls | sort

Shell Program



All shell programming can be done as single lines, using the “;”

operator, which is equivalent to pressing enter.

* How to run a shell program

$ bash first\_program.bash

$ . first\_program.bash

$ source first\_program.bash

* The first line of a bash program

It is proper shell programming practice to add a shabang (a.k.a. sharp- bang, hash-bang, pound-bang) sequence to indicate which shell we intend:

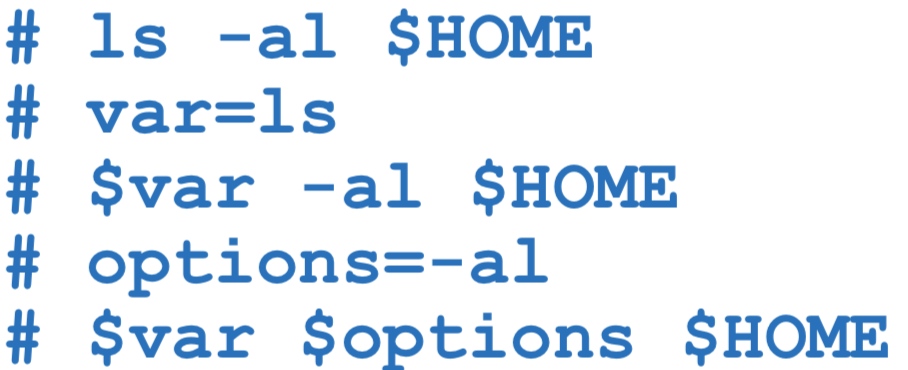
#!/bin/bash is the standard first line for every bash scrip

Shell Variables

Some of these are special parameters determine the behavior of the shell. Others are simply to be used to build program logic.

**Assignment with equals: #my\_var=Hello**

**Access with the dollar sign character: #echo $my\_var**

****

* Setting shell variables

Variable with an assignment command is a shell **builtin** command.

# HOME=/etc

# PATH=/usr/bin:/sdr/etc:/bin

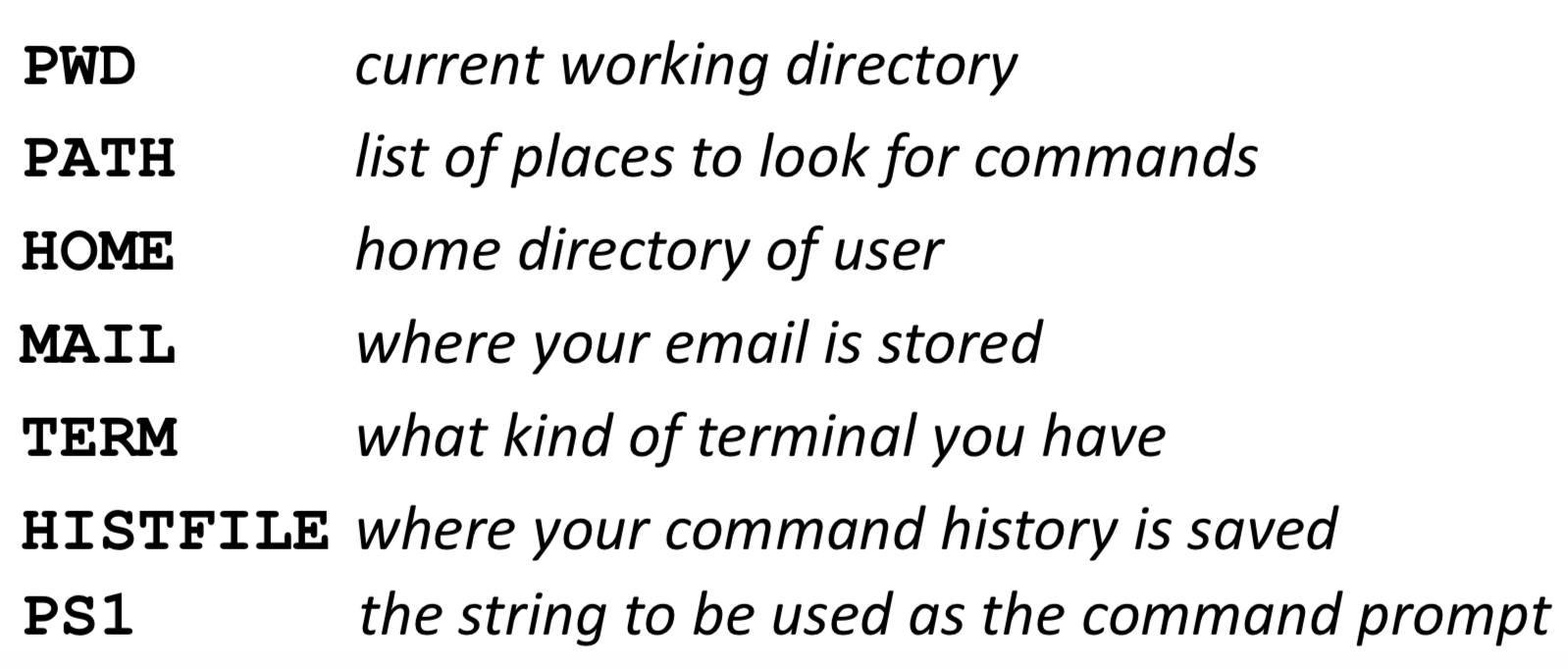
There cannot be any spaces in the variable name, between the variable name and the equals, between the equals and the value, or within the value.

However, it is possible to use values with spaces by enclosing them in quotes.

# var=“blah blah blah”

Note: The **set** command (shell builtin) without parameters will print out a list of all the shell variables.

[Shell Variables with Special Meaning]



E.g. The PS1 shell variable (a String) is your command line prompt and you can change the prompt by changing PS1.

#PS1= “Next command: ”

#PS1=“# ”

Note: Bash supports some fancy stuff in the prompt string

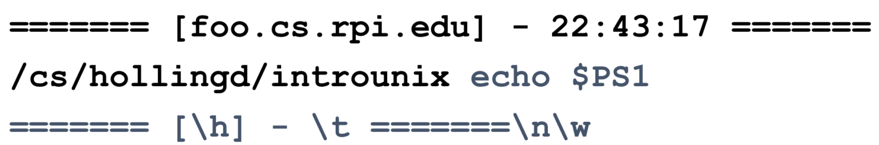
\t is replaced by the current time

\w is replaced by the current directory

\h is replaced by the hostname

\u is replaced by the username

\n is replaced by a newline



* Capturing a command output in a variable

*We saw that “>” stored the command’s output in a file. Sometimes we want to skip the filesystem (efficiency of memory vs disk) and re-use the output within our bash system.*

**# variable=`command`**

Anything that #command alone would output to the terminal is now stored as the value of variable. Access it with $variable.

A nearly equivalent syntax is **# variable=$(command)**

E.g. # var=`echo hello world`

# echo $var

hello world

Math

Enclose your computation in **$((computation))**

# a=$((3+5))

# echo There are $((60\*60)) seconds in an hour

Control Structures

* Shell Conditionals: if

*Shell conditionals are slightly different than other programming languages. They rely on a program to run and give an output code that can be evaluated.*

**“il” execute the body if the program returns an exit code of zero (success).**

Every process returns an integer value when it terminates. This is part of the Linux process specification. For example, "ls" returns 0 when the file you ask it to list was present, otherwise it returns 1(false). Check the return code from a program by accessing the special shell variable **"$?"**, which is set for every command in a shell program.

**if** *program*

**then**

*commands*

**elif** *otherpgm*

*commands*

**else**

*commands*

**fi**

Instead if using a junk file, use/dev/null, which is a special "file" specifically for the purpose of deleting whatever is put into it

E.g. if date | grep Mon > /dev/null

then

echo Another week starts.

fi

[Test Flag(s) Arguments]

test -r file: is the file readable?

test -w file

test arg1 = arg2: are the strings identical?

test arg1 != arg2: are the NOT equal?

-gt, -le, -eq etc: numerical tests( greater than, less than or equal to, equal, ...)

Test is so connected to shell conditionals that it has a special syntax **[[** *argument* **]]**:

if [[ -r my\_file.txt ]]

then

echo I can do something with the file!

fi

if [[ b = a ]]

then

echo hi fi

Note the spaces between each of the [[, each argument, and the ]]

* while

E.g. x=1

while test $x -lt 10

do

echo $x

x=`expr $x + 1`

done

**while** *program*

**do**

*list\_of\_command*

**done**

* for

this

is

a

test

E.g. for i in this is a test

do

echo $i

done

**for** *variables in wordlist*

**do**

*stuff*

**done**

Job Control

* Background Jobs

**If you follow a command line with "&", the shell will run the job in the background.**

You don't need to wait for the job to complete, you can type in a new command right away.

You can have a bunch of jobs running at once and do all this with a single terminal (window).



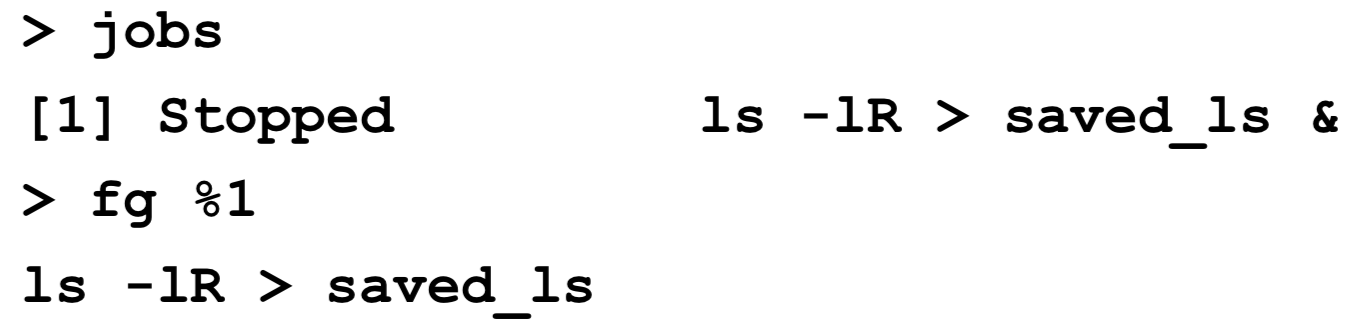
* Listing Jobs

The command **jobs** will list all background jobs**.**

* Moving a Job Back to the Foreground

**The fg command will move a job to the foreground.**

You give fg a job number (as reported by the **jobs** command) preceeded by a %.



Job number

* Suspending and Killing the Foreground Job

**You can suspend the foreground job by pressing ^Z (Ctrl-Z).**

Note: Suspend means the job is stopped, but nor dead. So the job will show up in the **jobs** output.

**You can kill the foreground job by pressing ^C (Ctrl-C).**

[Important Linux Paths]

“/” is the root of the file system. Every other file falls below “/” in the directory tree:

E.g., $ ls /

“~” is the current users home directory

E.g., $ ls ~/

“.” is means right here when it starts a path, and nothing if it occurs within a path (2nd case just a convenience for programming):

E.g., $ ls .

E.g. $ ls /usr/./bin

A screenshot of text

Description generated with very high confidence“..” means the parent directory

E.g. $ cd ..

<Practice>

"$ ls mtl10.jpg"

"$ ls "

"$ ls .."

"$ ls ~/A1\_rough/Q3/MontrealTest"

"$ ls gregs\_photos/../daves\_images/"

Find mtl jpg images starting with a 1 in the number

Find all jpgs

Find all directories that include the word "photos"

[Wildcards(metacharacters) for Filename Abbreviation]

**\* matches anything**.

If you give the shell \* by itself (as a command line argument) the shell will remove the \* and replace it with all the filenames in the current directory.

E.g. “a\*b” matches all files in the current directory that start with a and end with b.

E.g. $ echo \* = $ ls

E.g. ls \*

E.g. ls -al \*

**? matches any single character**

E.g. ls Test?.doc

**[abc…] matches any of the enclosed characters**

E.g. ls T[eE][sS][tT].doc

**[a-z] matches any character in a range**

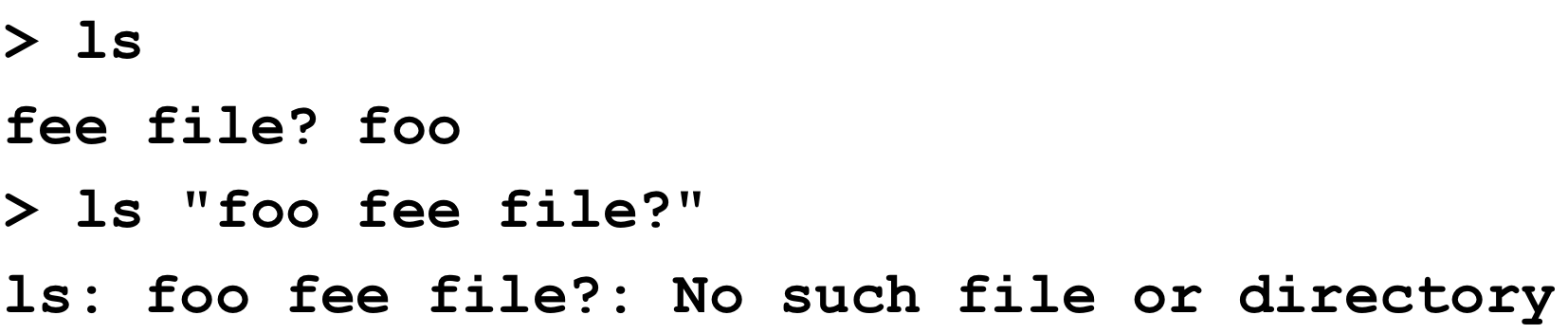
E.g. ls [a-zA-Z]\*

**[!abc…] matches any character except those listed.**

E.g. ls [!0-9]\*

[Quoting]

**“Double quote” around a string turn the string in to a single command line parameter** (turn off special meaning)



Note that some special characters are NOT ignored even if inside double quotes:

* $ (prefix for variable names)
* " the quote character itself
* \  slash is something special (\n)

you can use \$ to mean $ or \" to mean "

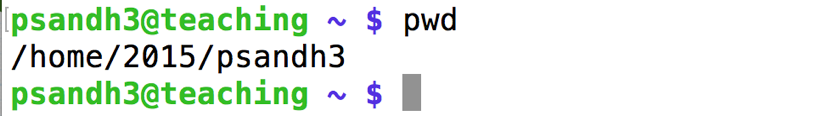
E.g. echo "This is a quote \" "

* Math in $((..)) is still evaluated
* Command-substitutions using $(…) or `..` are still evaluated

However, **‘single quote” is the stronger version**, that is nothing is interpreted as something other than its string values.

Or, you could use single quote just like double quote for syntax.

Present Working Directory **pwd**

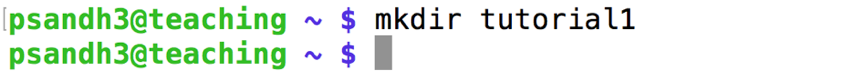


Make Directory **mkdir <dir\_name>**

current directory : .

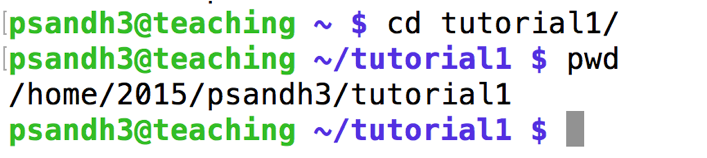
home directory : ~

root directory : /

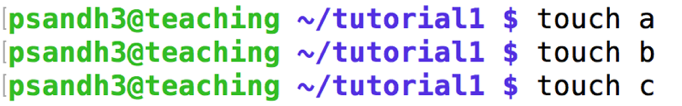


Change Directory **cd <dir\_name>**

Previous directory: ..



Create file **touch <filename>**



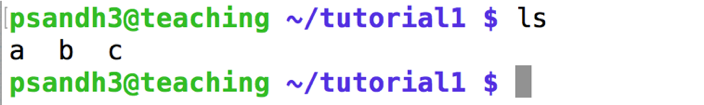
**echo “text” > <filename>**

page7image1832960

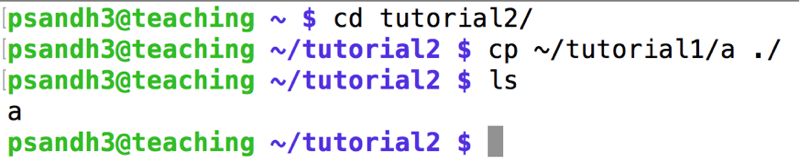
E.g. ying\_fall2018/comp206/tutorial/file1

Create a directory, go inside it and then create next. (Do it step by step!)

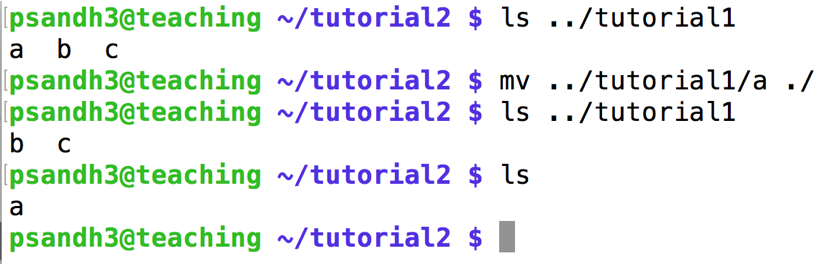
List Files **ls**



Copy the content **cp <source> <destination>**

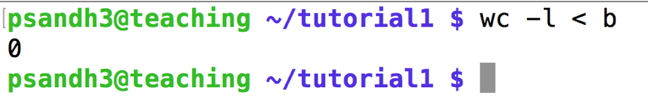


Move **mv <source> <destination>**



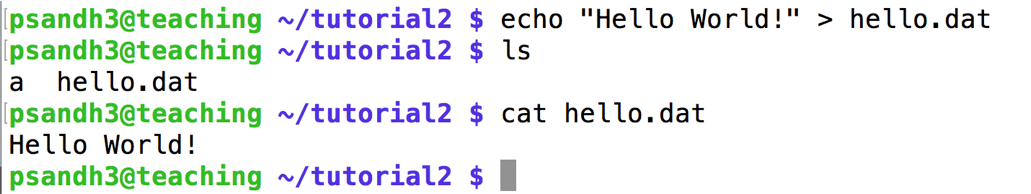
Print newline, word count for a given file **wc -l < <filename>**

Input redirection : <



Capture **cat <filename>**

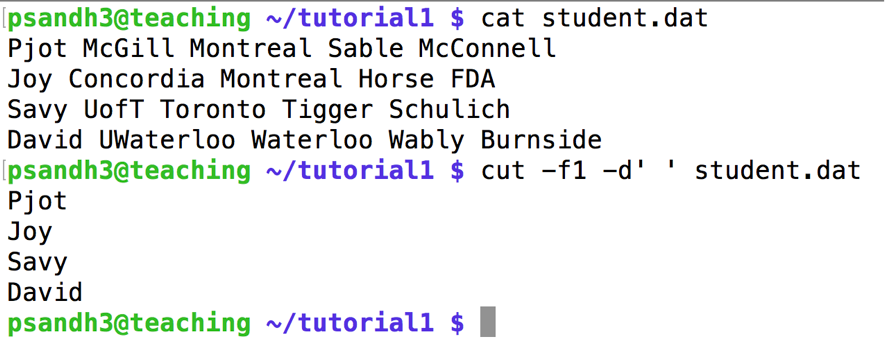
Output redirection: <



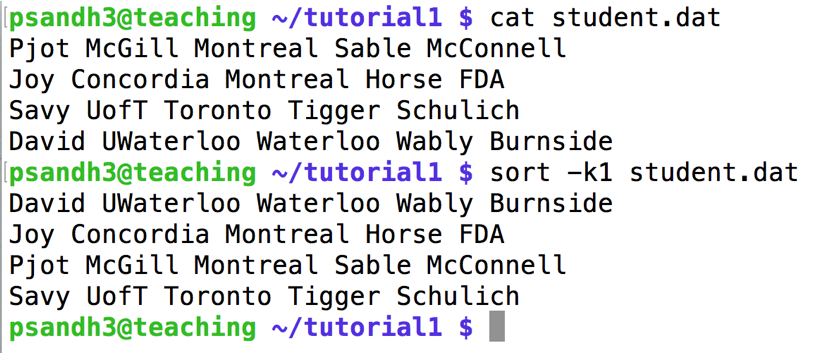
Manual page **man <command\_to\_know>**

press q to quit

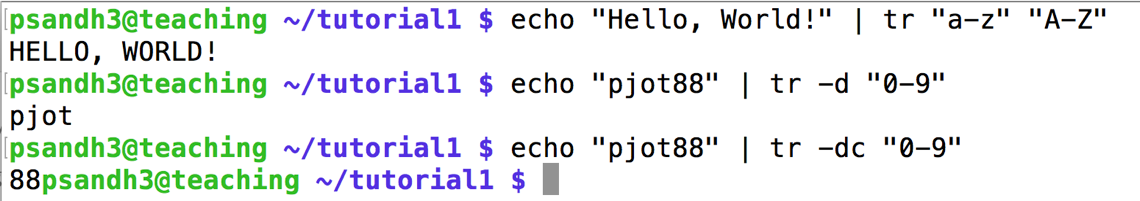
Cut **cut**



Sort **sort**



Translate **tr**



Grep **grep**

