Semester 1 2025
Astroinformatics I

Tutorial 4: Programming Intro

In this tutorial session, we will work with more complex `bash` scripts which include many characteristics of source code, such as conditional statements (`if ... else`) and loops.
To get started with our tutorial, first try to run the examples from the lecture.

1 While Loops

To get a better understanding of While loops, take a look at the following script which is a little more complex than what we saw in the lecture:

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

Question: Can you guess what happens here?

This loop lets the `echo` and `read` statements run indefinitely until you type `bye` when prompted.

Question: Could you think of an useful application of a loop following this scheme?

Another useful trick is the **while read loop**. The example uses the `case` statement, which is an useful alternative to the `if` statement when dealing with a lot of cases, like in a program menu.

```
#!/bin/sh
while read input_text
do
  case $input_text in
        hello)          echo English    ;;
        howdy)          echo American   ;;
        gday)           echo Australian ;;
        bonjour)        echo French     ;;
        "guten tag")    echo German     ;;
        *)              echo Unknown Language: $input_text
                ;;
   esac
done < myfile.txt
```

Question: Can you guess what happens here?

The script reads the file `myfile.txt`, one line at a time, into the variable `input_text`. The `case` statement then checks the value of `$input_text`. If what was read from myfile.txt was `hello` then it echoes the word `English`. If it was `gday` then it will echo `Australian`, and so on. If the line read from `myfile.txt` doesn't match any of the provided patterns, then the catch-all `*` default will display the message `Unknown Language: $input_text`, where `$input_text` is the value of the line that it read in from `myfile.txt`.
Let's say our `myfile.txt` file contains the following five lines:

```
hello
gday
bonjour
hola
```

(Note: Each line must end with a `LF` (newline) - if `myfile.txt` doesn't end with a blank line, that final line will not be processed.)

A sample run of the script using this file would produce:

```
$ ./while3.sh
English
Australian
French
Unknown Language: hola
```

2 Creating Files and Folders

We have seen that we can create folders using `mkdir`, and files using a variety of commands. You can create an empty text file using the `touch` command:

```
touch testfile.dat
```

Question: How can you check it is indeed empty?

A way to create a file and write to it is by using the `echo` and the `>` commands:

```
echo 'This is a test' > data.txt
echo 'yet another line' >> data.txt
cat > filename.txt
```

We can also interactively, i.e. on the console, write text to a file. For doing so, type the following command and then press Enter :

```
cat > myfile.txt
```

Now type your lines of text. For example:

```
Month, Profit
01/Apr/2004, $1500
01/May/2004, $1600
01/Jun/2004, $1450
01/Jul/2004, $1950
01/Aug/2004, $3950
01/Sep/2004, $2950
01/Oct/2004, $1750
01/Nov/2004, $1950
01/Dec/2004, $3250
```

When done and you need to save and exit, press Ctrl +D to return to the `bash` shell prompt. To view the file, use the `cat` command or the `more`/ `less` commands:

```
cat sales.txt
```

```
more sales.txt
```

2.1 A Shortcut for Creating Folders

This is a very handy shortcut for creating multiple folders following the same naming schema:

```
mkdir rc{0,1,2,3,4,5,6,S}folder
```

Which replaces:

```
for runlevel in 0 1 2 3 4 5 6 S
do
  mkdir rc${runlevel}folder
done
```

3 Programs

Often shell scripts call external programs. In addition to the built-in commands (such as `echo` and `which`), many useful commands are actually Unix utilities, such as `grep`, `cat`, `sed`, `awk`. In this section, we will cover a few examples.
Their range of application goes ways beyond what we cover here; it's good to know that these utilities exist - you will find many more, and often very specific, examples online.

3.1 `sed` (stream editor)

`sed` is a non-interactive editor. Instead of altering a file by moving the cursor on the screen, you use a script of editing instructions to `sed`, plus the name of the file to edit. You can also describe `sed` as a filter. Let's have a look at some examples to give you an idea of what `sed` can do:

```
$ sed 's/to_be_replaced/replaced/g' tmp/dummy
```

It replaces the string `to_be_replaced` with the string `replaced` and reads from the `tmp/dummy` file. The result will be sent to `stdout` (normally the console), but you can also add `> capture` to the end of the line above so that `sed` sends the output to the file `capture`.

```
$ sed 12,18d tmp/dummy
```

Here, `sed` shows all lines except lines 12 to 18. The original file is not altered by this command.

3.2 `awk`

`awk` (its name comes from the initials of its designers: Aho, Weinberger, and Kernighan) is suitable for pattern search and processing. The principle is simple: `awk` scans for a pattern, and for every matching pattern a action will be performed.
`awk` is capable of the following operations:

- Scanning files line by line
- Splitting each input line into fields
- Comparing input lines and fields to patterns
- Performing specified actions on matching lines

`awk` is generally used to change data files, as well as to change the formatting of data files.
We get started by preparing a dummy file `tmp/dummy` that contains the following lines:

```
test123
test
tteesstt
```

We want to print all lines containing `test`.
The pattern `awk` should look for is thus `test` and the action it performs when it found a line in the file `tmp/dummy` containing the string `test` is `print`:

```
$ awk '/test/ {print}' tmp/dummy
```

We get the following output:

```
test123
test
```

Try the following:

```
$ awk '/test/ {i=i+1} END {print i}' tmp/dummy
```

Question: What would you expect the above command does?

3.3 `grep` (global regular expression print)

Grep prints lines matching a search pattern.
Here is a typical `grep` command:

```
$ grep "error code" /var/log/messages -c
```

It counts how often the string `error code` can be found in the file `/var/log/messages`.
In many cases it would be useful to not only know whether a given string can be found and how often, but also what is the line number the string is found, and to get the line containing it displayed on the console.

Question: Using the online documentation, try to write a line of code for those two use cases: output the line number and the complete line where the string is found.

3.4 `wc` (word count)

`wc` is used to get the number of lines, word count, byte and characters count in the files specified in the file arguments. By default it displays four-columnar output, where the first column shows number of lines present in a file specified, the second column shows number of words present in the file, the third column shows number of characters present in file and the fourth column itself is the file name which are given as argument.
An example with one file:

```
$ wc tmp/dummyfile
```

An example with two files:

```
$ wc tmp/dummyfile tmp/dummyfile2
```

The following options exist:
`-l`: This option prints the number of lines present in a file.
`-w`: This option prints the number of words present in a file.
`-c`: This option displays count of bytes present in a file.
`-m`: This option counts the characters in a file.
`-L`: The `wc` command allows an argument `-L` (not to be mixed up with `-l`), it can be used to print out the length of longest (number of characters) line in a file.

In the following, we see some typical applications of the `wc` command:

Counting all files and folders in a directory
The `ls` command in unix is used to display all files and folders present in the directory. When it is piped with the `wc` command with the `-l` flag it display a count of all files and folders present in current directory.

```
$ ls testfolder | wc -l
```

Count the number of words in a file
We have seen that `wc -w` displays the number of words in a file, but indeed this command's output is two-columnar, giving the word count in the first column and the file name in the second column. We only want the word count here (imagine we might want to write it into a table later).
So to display 1st column only, pipe (`|`) the output of `wc -w` to the `cut` command with the `-c` flag.

```
$ wc -w  state.txt | cut -c1
```

Using redirection (`<`) works as well.

```
$ wc -w < state.txt
```

4 Some Useful Commands and Scripts

The following commands and scripts are especially useful when working with large tables as text files, which is very common in astronomy, We cover here especially use cases that are very common when preparing files that should be processed on a computing cluster, like changing file formatting (changing table delimiter, adding table header), modifying file names (including adding file extensions), removing or extracting columns, and similar operations. Other use cases are typical when tracking the progress of a program that processes such files, i.e. checking the number of files in directories including subdirectories, checking the length of files.

4.1 Empty all files

This empties all files ending with `.lc`. The files itself are not deleted.

```
truncate -s 0 *.lc
```

4.2 Empty a folder

This empties a folder, i.e. deletes the files it contains. The folder itself is not deleted.

```
rm -r myfolder/*
```

4.3 Add or rename file extensions

In case you have created files without a file extension (which is generally not posing a problem in Linux/Unix), but a program requires files to have an extension, you can automatically add the file extension by using the following loop:

```
for f in tmp/*; do mv "$f" "$f.txt"; done
```

In some cases it is necessary to rename the file extension without changing the underlying file type. This can be done the following way:´

```
find . -type f -name "*.avro.csv" -exec rename '.avro.csv' '.csv' '{}' \;
```

4.4 Get number of files in a `tar.gz` archive

Sometimes it is handy to know the number of files in a `tar.gz` archive before extracting it. The following command can do that:

```
tar -tzf /data_archives/lightcurves_public_20210411.tar.gz | wc -l
```

4.5 Count all files in subdirectories

Counting all files in subdirectories is a very common task, for example when you want to track the progress of a program that writes files into different subdirectories, or when you suspect that files are lost.
Counting all files in subdirectories is done by th following command:

```
find . -type f | wc -l
```

4.6 Get random files from a folder

In some cases, for example to create a smaller test subsample, it is necessary to get a random list of files from a folder. It is done by th following command:

```
ls |sort -R |tail -1000
```

4.7 Sort files by their size

Sorting files by their size is a common task that is easily carried out by a file system browser. If no GUI is available (you might work on a remote cluster or supercomputer), or you have a folder containing a huge amount of files (which is common for survey data) where a graphical file system browser likely becomes slow, using a file system browser might not be an option. Instead, use this command:

```
ls -al --si --sort=size
```

4.8 Remove double entries

Double entries in files can occur for various reasons. The following command removes double entries (i.e.: keeps only one version of a double entry) and saves the result into a new file.

```
cat old_file.txt | sort | uniq > new_file.txt
```

4.9 Delete column from file

Often tables contain a lot more columns than we need to read into a program. This is especially the case with files from astronomical surveys who nowadays often contain a range of columns for different science case. We can delete a column from a text file by doing the following:

```
cut -d\  -f2 --complement input_table.txt > output_table.txt
```

This deletes the second column. `-d ` is used for space as delimiter. For example, a comma as a delimiter would be specified by `-d,`.
If we rather want to specify which column to keep, we can do:

```
cut -d\  -f2 input_table.txt > output_table2.txt
```

This keeps only the second column.
Especially when working with large files it is always strongly recommended using command line utilities rather than trying to open the file with a graphical editor (spreadsheet or others). While the latter likely would crash or at least take a long time to load the file, the command line tool deals with less overhead and processes the file quickly.

4.10 Change column delimiter

Changing the column delimiter in a text file is just replacing text in file.
The following command replaces a comma with a semicolon:

```
sed 's/,/;/g' file
```

The following command replaces a comma with a space; see here the usage of `\` to escape the space:

```
sed 's/,/\ /g' file
```

4.11 Replace text for all files in folder

Sometimes we have to replace text, i.e. a file delimiter, for all files in a folder. It is done with the following command:

```
find my_folder -type f -exec sed -i -e 's/apple/orange/g' {} \;
```

4.12 Change column order

In some cases a program requires a certain order of columns, for example the first column being time, the second column being magnitude. In case this requirement isn't met in the input files, we can easily change the order of columns.
The following command outputs the first column of the input file as the second column of the output file, and the second column of the input file as the first column of the output file:

```
awk '{print $2" "$1}' < in.txt > out.txt
```

4.13 Add a line to a file

Opening a large file with a graphical text editor can produce a lot of overhead, taking a lot of time and slowing down the system. It is better to use a command line utility. Adding a line to a file as the first line, i.e. adding a column header, an be done the following way:

```
sed -i '1i col1,col2,col3' table.txt
```

4.14 Splitting up files

There are various reasons why one needs to split up a text file into several ones. One that is quite common is that a file contains a list of object IDs or file names to be processed by a program. One might want to split the file into smaller ones to pass each list to a different thread of a program.
Splitting up a file can be done by a) giving the number of files into which the input file should be splitted, b) giving the (maximum) number of lines each resulting file should have. In the following example, we want to split a file into several smaller files containing each 500 lines:

```
split -l 500 --numeric-suffixes=1 --suffix-length=3 input.dat output__
```

The individual arguments are:
`-l 500`: split into files containing each 500 lines
`--numeric_suffices=1 --suffix-length=3`: we want the files name with numeric suffixes three digits long, i.e. ending in 000, 001, ...
`input.dat`: this is our input file
`output__`: this is how the file names for our output files starts

We specify all the above as by default, the `split` command uses a very simple naming scheme. The file chunks will be named `xaa`, `xab`, `xac`, etc., and, presumably, if you break up a file that is sufficiently large, you might even get chunks named `xza` and `xzz`.

Exercise:
As an exercise, try the following: Create a folder to which you add several files (from the command line). Write a script that outputs you the file names of all files containing "astronomy".

5 Quick Reference

This quick reference guide summarizes the meaning of some of the less easily guessed commands and codes of shell scripts. These examples include process management, shell scripts arguments, and shell script test conditions.

Command	Description	Example
`&`	Run the previous command in the background	`ls &`
`&&`	Logical AND	`if [ "$foo" -ge "0" ] && [ "$foo" -le "9"]`
`||`	Logical OR	`if [ "$foo" -lt "0" ] \|\| [ "$foo" -gt "9" ]`
`^`	Start of line	`grep "^foo"`
`$`	End of line	`grep "foo$"`
`=`	String equality	`if [ "$foo" = "bar" ]`
`!`	Logical NOT	`if [ "$foo" != "bar" ]`
`$$`	PID of current shell	`echo "my PID = $$"`
`$!`	PID of last background command	`ls & echo "PID of ls = $!"`
`$?`	exit status of last command	`ls ; echo "ls returned code $?"`
`$0`	Name of current command (as called)	`echo "I am $0"`
`$1`	Name of current command's first parameter	`echo "My first argument is $1"`
`$9`	Name of current command's ninth parameter	`echo "My first argument is $9"`
`$@`	All of current command's parameters (preserving whitespace and quoting)	`echo "My arguments are $@"`
`$*`	All of current command's parameters (not preserving whitespace and quoting)	`echo "My arguments are $*"`
`-eq`	Numeric Equality	`if [ "$foo" -eq "9" ]`
`-ne`	Numeric Inquality	`if [ "$foo" -ne "9" ]`
`-lt`	Less Than	`if [ "$foo" -lt "9" ]`
`-le`	Less Than or Equal	`if [ "$foo" -le "9" ]`
`-gt`	Greater Than	`if [ "$foo" -gt "9" ]`
`-ge`	Greater Than or Equal	`if [ "$foo" -ge "9" ]`
`-d`	Is a Directory	`if [ -d /bin ]`
`-f`	Is a File	`if [ -f /bin/ls ]`
`-r`	Is a readable file	`if [ -r /bin/ls ]`
`-w`	Is a writable file	`if [ -w /bin/ls ]`
`-x`	Is an executable file	`if [ -x /bin/ls ]`