University of Reading

UNIX ADVANCED

A course building on the workshop course for Unix operating system.

University of Reading

OBJECTIVE

- The objective of this course is to introduce some more advanced techniques in using the Unix operating system, covering more powerful commands and operators for filtering and processing data files.

- Prerequisites – you should be fluent in the use of basic Unix commands, and familiar with the filesystem.

Resource download:

https://github.com/paulmheaton/Unix-Intermediate

Crib sheet:

https://devhints.io/bash

University of Reading

OVERVIEW

- In this workshop we develop the idea of using a shell script to process data.

- We will cover command line arguments to the shell script.
- Include optional switches
- Ask user for input.
- Use conditional branching
- Use looping to process many files.
- Dealing with errors.
- Using awk to process tabular data.
- Discuss output options.

University of Reading

THE BASIC SHELL SCRIPT

- Create a file called “process_data.sh”
- Change the permissions on the file to make it executable for all those in your group.
- Tell the user about the script and what it does.
- Run the script.

THE BASIC SHELL SCRIPT

- Your file should look something like this:

- #/bin/bash

echo This is a script to process tabular experimental data and output the results in a usable format.;

Echo DONE;

University of Reading

SHELL SCRIPT ARGUMENTS

- Add the following lines to your script after the first echo statement:

- echo The script is called: $0; # this is always the script file itself
- echo second argument is: $1;   # We will use this argument for a switch
- echo second argument is: $2;   # this will be the data to be processed

University of Reading

SHELL SCRIPT ARGUMENTS

- Save the script and launch it as follows:
- ./process_data.sh –c thermal_data.sh
- You should get an output telling you the script you are running, the switch you gave, and the file you wanted processing
- Notice how the $0 etc is interpreted within the echo text, to show the value of the argument on the command line.
- $0 is always the script itself, the remainder are allocated $1, $2 … etc

University of Reading

ERROR TRAPPING

- A BASH script will generally stop working if an error is found, but you may want a more controlled way of handling them.
- In our example we should check the file we intend to process actually exists! And that the switch we used is valid.
- If not, we should give the user a sensible message and exit gracefully.

Enter the following:

```
if [[ ! -e $2 ]]; then
    echo 'file does not exist!';
    exit;
fi
```

ERROR TRAPPING

- Refer to the crib sheet on the meaning of the above in the section marked “**File conditions”** for the *Standard Form* for the test conditions.

You will see that unix has some builtin expressions for handling files input.

Here we use the –e switch to detect if the file exists. We use the '!' symbol to represent 'not'.

So the expression reads: 'If the file does not exist then exit the script'

ERROR TRAPPING

- Refer to the crib sheet on the meaning of the above in the section marked “**File conditions”**.

Now you see how conditional branching works, let us add some help to our script:

```
if [[ $1 == "--help" ]]; then
    echo USAGE: process_data.sh [options] FILENAME;
    echo OPTIONS:;
    echo -c = Output cumulative data;
    exit;
fi
```

OPENING AND READING A FILE

Unix offers a few ways to open a file.

A file can be read in line by line:

```
echo  process command line argument directly;
while IFS= read -r line
do
        echo "$line";
done < "$2“
```

Or read in in one operation:

```
echo read in file to variable then process;
mydata= "$(cat $2)";
for line in $mydata; do
  echo $line
done
```

University of Reading

AWKWARD

- As you can see the above solution is not ideal: our data is in tabular form with the fields comma separated and each line is a record.
- We could use the first form to process each line, but we would have to split each string into an array then process the array which would require much coding.

- Fortunately for us a program has been written specifically to process tabular data. The program is called AWK after the initial letters of the authors surnames, and is standard on most Unix systems.

- AWK automatically *parses* files into a set of variables denoted by $n where n is the number of the column. By default AWK uses the spaces or tabs as a field separator, but you can change this.

University of Reading

AWK CONTINUED

- By default AWK reads files line by line using the newline character as the end of each set of data.
- The “Field Separator” FS tells AWK how to split the tabular data. Our file is comma separated so we set FS=“,”:
- `awk '{FS=","; print $3 "  " $4}' thermal_data_xxx.csv`
- This will print out columns 3&4 in a csv file.

MORE AWK

- To show more than one column simple add more variables to the print command, but ask awk to put some space between them.
- Let us use our data to show the temperature on the panel as a function of the data-time:
- Add the following lines to your process_data script:
- *awk '{FS=","; print "At " $1 " the temperature was " $3}' $2;*

- As you can see this makes filtering tabular data very easy.
- Also note the form of the AWK statement: *awk 'awkscript' filetoprocess*
- Awk followed by a series of statements inside apostrophes followed by filename. The text inside the quotes is the *AWK program*

AWK CONTINUED

- The time portion of our AWK output is a bit ugly. We do not need to show the date in this example, just the time of day.
- How do we eliminate this?
- Well we can further split the variables AWK has generated using an AWK function called “split”.
- Split has the form split(stringtosplit,arrayfor results,'field-separator')
- Change your script to read as follows :

- ```
  awk '{FS=",";
       datetime=$1;
       split(datetime,dtarr," ");
       print "At " dtarr[1] " the temperature was  " $3}' $2;
  ```
  ```

AWK CONTINUED

- The process of splitting variables down can be extended as much as needed.
- Let us just print out the timeofday instead of the whole date string.
- Change the script to read
- *awk '{FS=",";*
- *datetime=$1;*
- *split(datetime,dtarr," ");*
- *split(dtarr[2],timeofday,".");*
- *print "At " timeofday[1] " the temperature was " $3}' $2;*

- And run the script.

AWK CONTINUED

- As you can see AWK has it's own language, and can perform other operations on the data.

- What if you a summary at the end?
- The END statement tells awk what to do when it has finished processing. The format is END {…statements…}

AWK also has a BEGIN statement which performs operations before processing the file, and can be used to store “global” variables

The format is BEGIN {…statements…}

AWK CONTINUED

- Modify your AWK script as below:

```
awk 'BEGIN{
max=0
min=1000
FS=","
tot_heat_gain=0
}
```

University of Reading

AWK SCRIPT MAIN PROGRAM

```
{
    joules_gain=($2-$4)*4200*100
    tot_heat_gain+=joules_gain
    if($2<min){min=$2}
    if($2>max){max=$2}

    print "reading=",$2,"   max=",max,
 "min=",min,"   joules=",joules_gain
}
```

University of Reading

AWK SCRIPT END SECTION

- END { print "The maximum temperature was ", max
- print "\nThe minimum temperature was ", min
- print "\nThe total heat gain for the day was ", tot_heat_gain/24/60/60/1000 ,"KW" }
- ' $2;

University of Reading

EXERCISES WITH DATA

- Modify script to output cumulative data to file in format:
- Date Time Max Min HeatGain

- Create an AWK file and process data as awk –f filename.awk file-to-process.csv in the shell script.

- Process all the files in a directory and merge all the summary data into one file.

- Allow the user to alter the output filename

THE END!

- Happy Bashing! ☺

- Please give us your feedback by following this link below:

  https://forms.office.com/Pages/ResponsePage.aspx?id=xDv6T_zswEiQgPXkP_kOX7ArvOm3cbpHnixhCNWKRS9UNjFCNjg2V1E1NkhSTldFUUFORFBRRzlXUy4u