Module 1.8 Execution control

In this unit we will see how we can combine commands together, in order to control the flow of these commands. Specifically, we'll see how we can execute commands conditionally with an if or an else, and also how can create loops to iterate over files, or while a condition is true.

The "for" command allows us to iterate over a set of values. In order to see the time in all three European cities, of London, Paris and Madrid, I place them in a for loop.

The syntax starts with the iteration statement: for i in all these three timezones; do, continues with the commands we specify, which in our case is to: showtime the America/Los_Angeles time 11:45 in the $i timezone, and concludes with a done.

The i variable receives the values of London, then Paris, and lastly Madrid. I'm iterating with i through these values and according to the output: when the time in L.A. is 11:45, in London it is 7:45 PM, whereas in Paris and Madrid it is 8:45PM.

Apart from simple for loops, we can also define nested for loops. For instance, to extend the previous example, we can specify a set of timezones, and then convert time between any pair of them. To do that, I firstly set a variable TZONES to the timezones I'm interested in: Los Angeles, New York and London.

Then I iterate through the values of TZONES twice;the first for loop uses the timezones as input, and stores them in tzin, while the second for loop has them stored as output in tzout.

Using the function showtime, I now convert the time 11:45 for each tzin to each tzout. The result we see is a conversion of 11:45 between all timezones. The "if" command allows us to execute various other commands, based on the result of a condition; the result of whether another command succeeded or not. As a demonstration, I create two files: the sourcefile and the destfile, with a bit of delay, so that the destfile is clearly more recent than the sourcefile.

To refresh a file based on another, for example to refresh the destfile based on sourcefile,I run: if test sourcefile -nt destfile; using the test command with the nt option, to check whether the sourcefile is newer than the destfile. In case it is newer, sourcefile is copied to the destfile, and I echo that the destfile has been refreshed. Apparently, nothing happens because the sourcefile is older than the destfile, and, as a result, the destfile is not refreshed.

To verify the opposite case, I recreate the sourcefile to be newer than the destfile and run the command again, this time using an alternative syntax for the test command: a set of square brackets. Again I test whether the sourcefile is newer than the destfile, and repeat the same copy and echo commands.

We notice that the destfile is actually refreshed this time, because it is older than the sourcefile. The if command can also be combined with an else case.

In a similar example, I test whether the sourcefile is newer than the destfile, and after writing the same copy and echo commands, I add an else command. In case the condition is false, I echo that the destfile is up to date. As expected, the destfile is shown to be up to date.

The "while" command allows us to iterate while a condition holds. Let's now write a small script that counts the average number of characters per line, for all readable files in the etc directory.

First, I list the directory entries redirecting the output of ls, with a pipe to a while loop. While takes as an argument another command, read, in order to read the input directory entries line by line. Each filename is stored in a variable called name. Then for each filename, I test with an if command, if it is a regular readable file, passing the -f option to check it's a regular file, the -a option for and, and the -r option to check if it's readable.

In case the entry is an actual file and not a directory, and is readable, I display the filename without a newline, by passing the -n option to echo. Next, I run the expr command, to calculate the average number of characters per line in the particular file.

Specifically, I count the characters of the file using wc -c and the file as input, along with the number of its lines using wc -l, and divide the two. I complete the if command with fi and the while loop with done, and pipe the output to head to see the first few lines.

The "xargs" command is a magnificent tool for working with huge datasets. The command reads data from its standard input, and then supplies them as arguments to a specified command that it executes. It is needed, because the operating system, provides only a fixed amount of space for providing arguments to a command. Xargs overcomes this limit by executing the command repeatedly, with new batches of arguments chunked from its input.

Consider as an example, that we want to find how many lines exist, in all files under a specific directory, in my case the current directory. I run the find command in the current directory, specifying with -type f that I'm interested in files, and then apply cat through xargs on all files, in order to concatenate the files together.

Finally, I pipe the output of the wc -l command, to count the number of lines. Find doesn't always work as expected with xargs. For instance, let's say I want to find the oldest file, of the directory "Program Files" on a Windows machine. To do that I run find for "Program Files" and type file, I apply stat to get the modification time of these files, sort through the output numerically, and display the oldest file using head -1.

But, instead of the oldest file, we get various errors saying: cannot find Application or Verifier. The problem here is that the output of find, contains some lines with spaces, because on the Windows system, filenames often have their words separated by space.

Since xargs uses both a newline and a space to break up its arguments, some filenames containing spaces are incorrectly broken apart, leading to the creation and display of non-existing files.

To get around this, find offers the option -print0, which separates the output elements using a null character, instead of a newline, based on the fact that there are no nulls in filenames.

The output of find is then piped to xargs using a -0 option, to specify that the input is separated by null characters, rather than newlines or spaces. Again, I output the modification time and the name of each file, sorting the results numerically, and retrieving the oldest file, which is a LICENCE file of gsview.

By listing the file we verify that it was created 20 years ago, in the year 2000.

The "case" command allows us to run a specific command, based on pattern matching. To see this in action I create some command aliases, that depend on the operating system I currently run.

These are useful to make our commands work, independently of the system we use. The uname command displays the name of the current operating system, which in my case is Linux.

With a case statement I use the output of uname, and then insert an "in" and some patterns that specify what happens, depending on the value of uname. In case of Linux, I create a command alias s, which starts a document using gnome-open.

Then, I create a second alias cpt, that copies the current directory to the clipboard, using pwd to print the current directory, a pipe, and xsel --clipboard. With a double semicolon I end the first case.

I then start another one for Darwin, which is the output of uname on an Apple macOS machine. Similarly the alias s is set to "open" here, whereas cpt is set to "pwd | pbcopy".

On the Cygwin, Unix-like environment for Windows, the aliases vary accordingly. Finally, after ending the case command I run alias, and verify that the aliases cpt and s correspond to the Linux case. So cpt contains "pwd | xsel", while s is set to "gnome-open".

This concludes our foundations unit on execution control; stay with us!