Technische
Universität
Berlin

BASH advanced– Variables, scripting, tips & tricks

Training for Afghan System Administrators

Bash environment and config files

Bash has different config files, some of them global (in /etc/...) and some of them user-specific (in user's home dir).

- /etc/profile

  # the first config file that is being read when you login (changes in this file only become active once you re-login); holds a lot of environment variables, like $PATH
- /etc/bash.bashrc (on some distributions: /etc/bashrc)

  # This file is read every time you open a new shell. Often this file is being read by a statement in the ~/.bashrc file (user specific bashrc file); holds e.g. alias definitions
- ~/.bash_profile

  # like /etc/profile but for users; changes require to logout and then login again
- ~/.bashrc

  # like /etc/bash.bashrc; gets read every time you open a shell window

Working with shell variables

In Bash, declaring a variable is as easy as:

- a=5

  # we assign the variable “a” the value of 5; note how the variable name is “naked” here
- echo $a

  5

  # to print the value of a variable, we can use the echo command; note how we now use the $ sign to echo the *value of a*

Simple math in Bash:

- a=5
- b=6
- let c=$a+$b
- echo $c

  11

  # first we assigned a and b two values. Then we declare variable c with “let” (=makes sure c is arithmetic)

Variable scope

Variables that are declared like a=5 in Bash are called “shell variables”. They are NOT inhereted by other shells or sub-shells (a shell that was created by a script). If you want to make a variable available in a sub-shell, you have to export it:

- export a=5

  # if you now create a sub-shell (from within the shell where you declared the variable, then the variable exists and has the value we gave it.

Shell variables can only be exported to sub-shells, never to a “parent” or another “independent” shell.

Command substitution

It is possible to use the result of one command line as a string for another command. For example, the pwd command prints the current working dir as a text string to standard output. If you now want to use this text string, for example in an echo “ … “ statement, you can substitute it using to different ways (but identical in result):

Imagine you are in the directory /home/userX

- echo “The current working directory is $(pwd) ”
  OR
- echo “The current working directory is `pwd` ”
  # both commands will output “The current working directory is /home/userX”

Tests in Bash

If you want to test a condition, e.g. if a variable is equal to a value (or greater / lesser / etc...), then you use the test command of the Bash language.

- a=5
- test $a -gt 4
  # will evaluate to TRUE

There is also another way to write tests, with squared brackets: […]

- a=5
- [$ -gt 4]
  # will also evaluate to TRUE; spaces before and after the condition are required

There are also tests to check if a variable exists, if it is a file, etc. See *man test*

Shell scripting

A shell script is simply a list of shell command lines put together in one file (one command per line!).
Any shell script needs to start with “#!/bin/sh (called 'she-bang'), this tells the shell interpreter that this file is a) executable and b) that it is a shell script file (and e.g. not a perl script)

Useful variables when working with scripts:

- $0
  # name of the file (script) that is being executed
- $1, $2, $3, …
  # 1st, 2nd, 3rd … arguments that were passed on the command line to a script
- $@
  # all arguments as an array ($1, $2, $3, ….)
- $?
  # contains the exit value / error level of the LAST command run

While, for, if ...

There are different constructs, like in other programming languages that are very common. Instead of explaining how they work in words, please look at some example scripts and if in doubt, look into the “Advanced Bash Scripting Guide” (see Moodle or use Google).

The most common constructs and what they do:

- A “while loop” does something as long as a condition is fulfilled
- A “for loop” does something to a defined group of elements
- An “if … then” statement checks a condition and then does something accordingly (once)
- A “case” statement checks a variable against a list of possible values and then does something if value matches a given case.

Functions

If you repeatedly do the same operation / calculation / etc. in a Bash script, probably you can save time and space (less code) by using a function. A function is like a mini script inside a script. Every function has a name and consists of commands (one per line). For example look at this simple function to add 2 values (a and b)

-
  ```
  function addition () {
          let sum=$a+$b
          echo "The sum of $a + $b is $sum"
  }
  ```

- We can then call this function by just writing “addition” further down in our script.

BASH history commands

The BASH built-in history functions can help you make your life a LOT easier. It uses the file ~/.bash_history to look up previous commands.

- <ARROW UP/DOWN>
  # select command from history (one after another)
- <abc...><TAB>
  # auto completion for <abc...>; double press <TAB> to see a list of possible completions
- !!
  # runs the last command again (also called "bang bang")
- !n
  # run the n-th command; use the 'history' utility to see a numbered list
- !-n
  # run the command "n" before this one. so '!-1' is the last, '!-2' the second last, etc....
- !<abc...>
  # runs the last command that starts with abc
- !? <PATTERN>
  # runs the last command where <PATTERN> appeared in the text somewhere

Shell wildcard characters

Wildcards are useful, if you need to do the same action to a group of files that share a pattern. Since the wildcards are interpreted by the shell itself and not the individual utilities / commands, they will work the same way with all command line utilities!

- ls file*.pdf
  # * matches one or more characters → fileA.pdf, fileAbC.pdf, file1a2b.pdf
- ls file?.pdf
  # ? matches exactly one character → fileA.pdf, file2.pdf, but not fileAB.pdf, file123.pdf
- ls file[a-z].pdf
  # [a-z] matches any character in the given range → filea.pdf, fileb.pdf, but not fileA.pdf
- ls file[abc123].pdf
  # matches a, b, c, 1, 2, 3 → filea.pdf, file2.pdf, but not filed.pdf, file8.pdf
- ls file[!a-z].pdf
  # negates a-z → file1.pdf, fileA.pdf, but not filea.pdf, fileb.pdf, filez.pdf

More BASH tips & tricks

Exectuing multiple command lines at once:

- CMD1 ; CMD2

  # run CMD1, then run CMD2, etc....
- CMD1 && CMD2

  # run CMD1 then CMD2, but only if CMD1 exited without an error
- CMD1 || CMD2

  # only then run CMD2 if CMD1 produced an error before

Set an alias for commands you use often:

- alias ll = “ls -l”