Exercise Sheet 2

Writing good Bash scripts

08 October 2019

Exercise 1

Refresh your mind

As good start of the exercise session, you are encouraged to go through the material presented in the morning once more alone. Focus on the examples and try to understand all the details. Do not hesitate to try to copy the code there to your terminal and play around with it.

Exercise 2

Loops, loops everywhere

Consider the following tasks and write for each a short script.

1. Given two extensions, rename all files in the present folder having the first extension, changing it in the second provided one.
2. Suppose to have files in a folder with spaces in the name. Create a script to rename them, replacing each space with an underscore. Use the touch command with the brace-expansion feature in order to create a test environment to test your script.
3. Create an empty folder and move into it. Create some empty files via

   ```
   touch conf.{,{1000..200000..5000},save} prng.{,{1000..180000..3000},save}
   ```

   Implement a script that

   (a) deletes conf or prng files, if a prng or conf with the same number is not present;

   (b) find the largest number for which both a conf and a prng file exits.

Exercise 3

The power of arrays

Arrays play a very important role in Bash scripting and they are too often forgotten. To warm up and also explore the array notation think about

- how to find the longest entry of an array?
- how to find the maximum of an array with *numeric* entries?
- how to find only common entries in two arrays?
- how to sort an array?
- how to check if an array is sparse?

Once you feel comfortable with the array syntax, tackle the following problems.

1. Write a Bash script to make a report of the files in the present folder, counting them by extension. In order to test your script, create a test folder where you can create some files via

   ```
   for ext in jpg png eps pdf txt odt tex; do
       for ((index=1; index<=$(shuf -i 3-9 -n 1); index++)); do
           touch file_${index}.${ext}
       done
   done; unset -v 'ext' 'index'
   ```

2. Write a script that given integers on the command line parameters, calculates their **greatest common divisors** and their **least common multiple**, using the prime factorisations. Take a look to the `factor` command.

Exercise 4

The most flexible command line parser

Most Bash commands can be comfortably used thanks to command line options, which usually have a short and a long version. The `-h` or `--help` option is the starting point when you want to discover what a script does. Therefore, **any script** you will write **should have a `--help` option** to provide information about itself. In this exercise you will write by hand your command line parser, which you will use over and over again in all your scripts!

In general a command line parser is a block of code, which processes the command line parameters, doing some actions (typically setting global variables for the rest of the script). Here reasonable requirements you should fulfil.

1. Validate as often as possible the value of each option, checking for missing values as well.
2. By convention, `--` should separate script options from file(s) or, more in general, a list of "known objects".
3. If an option is not recognised, an error should be given.
4. Although it is nothing more than a common habit, short options have only one dash followed by one character only, while long options start with a double dash.
5. If the user asked for the help, all other (potentially wrong) options should be ignored.

In this exercise, just as practice example, you should implement the following options.

`-f, --file`
: An existing file to be stored in a variable.

`-b, --beta`
: A positive number with at maximum 4 decimal digits. The number without + and with exactly 4 decimal digits should be stored in a variable.

`-v, --verbose`
: An option without value which should be used to activate a verbose mode. If given it should set a global variable to `'TRUE'`, which otherwise should be set to `'FALSE'`.

`-s, --sizes`
: An option that can take an arbitrary number of positive integer numbers, which should be stored in an array.

`-a, --aspectRatios`
: Exactly the same as the `-s` option but values should be stored in a different array.

`-r, --range`
: An option to specify a lower and an upper bound to be stored in two variables.

`-w, --walltime`
: A string in the format `d-hh:mm:ss` which should be stored in a variable. The user can either specify it like that or specify integers followed by `d`, `h`, `m`, `s` (e.g. `1d4h`).

`--`
: If present, a double dash indicates that what follows should be a list of files. Store them in an array, without checking if they exist.

Requirement: The options `-s` and the option `-a` are mutually exclusive, i.e. cannot be specified at the same time. Approach the problem supposing to have a larger set of mutually exclusive options.

Bonus track: How would you allow the user to combine multiple short options? In this case the user might specify e.g. `-vs 7 13` instead of `-v -s 7 13`.

Exercise 5

A completely meaningless game (for humans)

For sure you will know the game in which you have to find the word not belonging to a given group of words. In this exercise we want to implement a variation of it… let's say for robots who cannot find a logic and just guess. The final game should ask the user interactively how many words should be in the group, display a group of random words and then ask which one does not belong to the group and repeat the question until the user guesses correctly. A couple of remarks:

- Draw your words randomly from the system dictionary which should be somewhere like `/usr/share/dict/words`.
- Use an appropriate Bash keyword to let the user choose the word in the group.

Exercise 6

A strange pipeline

Pipelines are handy and by now you will know them quite well. To practice a bit with redirections, try to think about how having a pipeline which redirect only standard error to the following command, while standard output is redirected to the terminal. You can use this compound command to test your solution

```
{ echo "I'm stdout"; echo "I'm stderr" >&2; }
```

and pipe to the `rev` command to see that the effect should have effect on the standard error only.