Exercise Sheet 3

Bash, challenge accepted

09 October 2019

Exercise 1

The world of functions

In this exercise we will write few functions to practice and understand how they can be useful.

1. In the exercise session from yesterday, you learnt how to provide your script with command line options and, then, how to parse them. Write a function that takes over this responsibility. How do you invoke this function? Take the opportunity to structure further more your code, extracting more functions to avoid code duplication.

2. Write a function which checks if an element is in a given array. How is input passed to this function? How would you signal to the caller the presence/absence of the element? Your function must work in general. Test it with

   ```
   array=("first element" $'to be\nfound' "something")
   string=$'to be\nfound'
   ```

   or with `string="notExisting"`.

3. Write a function that, given two integers, calculates their **greatest common divisors** using the Euclid's algorithm . This is a great opportunity to write a recursive function. Write another function which calculates the **least common multiple** of two integers, using the relation

$$\text{lcm}(a,b)\cdot\text{gcd}(a,b)=a\cdot b \quad .$$

Exercise 2

A (colourful) logger

Now that you are more confident with functions, try to implement some functionality to easily log information to the user. Often such a functionality is called logger. We will focus on a basic version, but with a bit of imagination you can improve it.

1. Start with two basic functions which should simply print what they receive to the output, prefixing it with a label and using a given colour. Using them via

   ```
   PrintInfo 'An informational message'
   PrintError 'An error message'
   ```

   should give

   ```
   INFO: An informational message
   ERROR: An error message
   ```

   where you can also put the label in **bold** if you wish. Write error messages to standard error.

2. Work on the functions in a way such that each parameter is printed on a new line, with a hanged indentation with respect to the label. For example,

   ```
   PrintError 'An error message' 'which spans two lines' 'or three'
   ```

   should give

   ```
   ERROR: An error message
          which spans two lines
          or three
   ```

3. You should have noticed that there is a lot of code duplication. Write a generic `Logger` function which take a label as first argument (either `INFO` or `ERROR`) and does what the other two functions where doing, depending on the label. Use the `Logger` in the `PrintInfo` and `PrintError`

functions. It is up to you if you want to use the `Logger` everywhere or having a wrapper per level.

4. Add a warning level to your logger, with the correspondent `PrintWarning` wrapper.
5. Additional optional level might be `FATAL`, `INTERNAL`, `DEBUG` and `TRACE`. A fatal error is an error which causes a script to terminate, while an error might not necessarily be cause of abort. Internal messages might be thought to be for developers. Debug and trace levels are thought for low level messages and they should be switched off in normal, production runs. Use an environment variable to set the logger level. Give a meaningful priority to your logger levels and make it such that only the levels more important than the chosen level are switched on.

Exercise 3

Using traps to clean up

The main purpose of this exercise is to understand how a trap works and how it can be useful. We will consider the most common case, cleaning up, but signal handling can be used for much more than that.

Suppose you have an analysis tool that takes a file with only one column in input and produces a file in output. Suppose as well that your tool produces temporary files which are auxiliary for the analysis. We will mimic the analysis tool via `sort -g` (redirecting the output) and the temporary files will be simply touched. Write a script to complete the following operations.

1. Split the data file into several files, each having one column of the original file.
2. For each single-column file
   - process it with the analysis tool (which will produce an output file);
   - create the auxiliary files, e.g. with `touch column_${index}_{0..9}.aux` (or similar);
   - sleep for one second;
   - remove the auxiliary files.

You can produce a fake data file via `seq 0 .001 1 | shuf | columns -c 11 -W 100` which has 11 columns, each with 91 lines.

3. Run the script (ideally in a new folder) and abort it with CTRL-C after few seconds. Your folder should have been polluted by a bunch of files.
4. Add a `trap` to your script to clean up on exit.

Exercise 4

Awk and sed warmup

Go through the few slides discussed in the lecture about `awk` and `sed`, focusing on the examples there. Once prepared an input file via

```
paste <(shuf -n 20 /usr/share/dict/ngerman) <(shuf -i 1-5000 -n 20)
```

achieve the following tasks in either of the tools.

1. Print the first 5 and last 10 lines.
2. Print every third line.
3. Display only lines for which the number on the second column is smaller than 1000.
4. Calculate the average of the second column.
5. Print lines starting by a vowel.
6. Print the first column word if the second column contains a number larger than 3000.

Exercise 5

A taste of awk and sed

Awk and sed are two incredible tools, but it is not automatically true that they are the *best tools* to solve your problem. In this exercise they can be used, but it might be you find your way without using them. This is true in general, indeed.

Here you can find a chunk of the standard output of a LQCD simulation, in particular measurements lines, only. Each measure reports in square brackets the simulation step (the so-called trajectory) at which it has been carried out. Unfortunately, there have been some I/O problems on the cluster where this file was generated and there are holes in the simulation. Observables of interest for the purpose of the exercise are `PLAQUETTE`, `fRECTANGLE` and `POLYAKOV`. Have a look to the file and then, playing in the terminal, figure out how to answer to the following questions.

1. How many times each observable has been measured?
2. Which is the first and last trajectory for each observable?
3. Where did the I/O problems occur for each observables?

You could set up a (long) command which would print

```
PLAQUETTE: Measures=2311 tr=[2000-4999] Holes=[2056-2287 3202-3433 4349-4579]
fRECTANGLE: Measures=2312 tr=[2000-4999] Holes=[2056-2286 3202-3433 4349-4579]
POLYAKOV: Measures=2311 tr=[2000-4998] Holes=[2056-2286 3202-3433 4349-4579]
```

Now that you understood the basic idea, write a Bash script in order to extract the observables in a file with 4 columns: the trajectory number and the three observables (`PLAQUETTE`, `fRECTANGLE` and `POLYAKOV`). Pay attention to the trajectories for which you do not have all observables! In such a cases, you should discard the whole trajectory (nothing to be added to the output data file) and print a warning to the user.

NOTE: In the whole exercise, you might dislike having to process several time the file, since it might be slow. Well, what does *slow* mean? How does the time you might save compare to the time you need to come up with a solution to avoid a file processing? Do you remember Donald Knuth's words? *"Premature Optimisation Is the Root of All Evil"*.