Exercise Sheet 3

Bash, challenge accepted

27 September 2023

Exercise 1

Discovering the redirection mechanism

From now on, you will bring your new bash scripts to a new level, e.g. always redirecting error messages to standard error.

Suppose to have to produce at some point in a larger script an input file for some application. This might require e.g. some logic to add different lines to a file. A minimal example might be the following, although you have to imagine to have tens of printing statements and more complex structures.

```
echo "THERMALIZE=1"
if (($1 % 4 == 0)); then echo "USE_HMC"; else echo "USE_RHMC"; fi
echo "DO_BACKUP=0"
```

Think of how redirecting the output of the block of code above to an external file

- using a compound command;
- using the `exec` builtin;
- using a here document (maybe not ideal).

Exercise 2

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 to create a pipeline which pipes only standard error into the standard input of 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 it to the `rev` command to see that the effect should have effect on the standard error only.

Exercise 3

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

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

Exercise 4

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 5

Writing a simple autocompletion script

Consider again the basic example discussed today in the lecture. We saw that the implemented autocompletion mechanism is not ideal, since the same command line option is displayed pressing `<TAB-TAB>` although already given. How would you fix this aspect of the script, supposing that each option of the `measure` script shall be given only once? You can create the autocompletion script in any folder (e.g. where you solve exercises) and source it in your terminal (after each modification).

Exercise 6

Rebinding the up and down arrows

If you did not know it already, today you discovered the GNU Redline library. You were also provided with some examples to bind differently keyboard shortcuts or single keystrokes. It is probably instructive to have a look to the output of the following couple of commands

```
bind -p | less
bind -P | less
```

with the help of section 8.4 the bash manual.

Afterwards, try to rebind the up and down arrow in your terminal as suggested in the lecture and explore how it works. Do you like it?