Exercise Sheet 1

Exploring the core of Bash

26 October 2020

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

Understanding quotes

Define a variable `a='apple'` in your terminal and try to figure out what happens (and why) when you pass the following expressions as argument to `echo`.

```
"$a"          '\''          "\'"          "$(echo hi)"
'$a'          "red$arocks"  "\""          '$(echo hi)'
"'$a'"        "redapple$"   "*"           $'$a\''
'"$a"'        '\"'          "\t\n"        "$'\t'"
```

At the end unset the variable `a`.

Exercise 3

The special parameters `*` and `@` and their quoted versions

Consider the following script.

```
1 #!/bin/bash
2 printf '\nScript run with %d argument(s)\n' "$#"
3 IFS=":${IFS}"
4 printf 'Using "$@":'
5 printf ' <%s>' "$@"   # or "$*" or $@ or $*
6 printf '\n\n'
```

Use the manual or the web to understand how the command `printf` works and to understand then the given script. Make the above script executable and complete it adding lines 4-6 for `"$*"`, for `$@` and for `$*`. Create a new temporary folder, move into it and `touch Day_{1..3}.dat` (what happens?). Run your script with the following arguments:

```
'*.dat' $(whoami) "Hello World"
```

Have you understood the difference between `"$@"`, `"$*"`, `$@` and `$*`? Are there differences between the unquoted `$@` and `$*`? Try removing quotes from `'*.dat'`.

Exercise 4

Arithmetic expansion

1. Using arithmetic expansion only in your terminal, find the largest integer which can be stored in a Bash variable.
2. How many bits are used in your opinion to store an integer value?
3. How would you explain that `echo $(( 2**64 ))` prints `0`? Try to change `64` with larger integers.
4. Read section 6.5 of the Bash manual v5.0 carefully.
   Is there any difference between `$(( 36#a ))` and `$(( 36#A ))`? And between `$(( 37#a ))` and `$(( 37#A ))`?
5. Can you explain the output of the following command?

   ```
   printf "%d %x %o\n" $(( 64#_ )) $(( 64#_ )) $(( 64#_ ))
   ```

Exercise 5

Understanding the power of parameter expansion

Go through the following list of tasks and explore the parameter expansion syntax. Feel free to simply play in the terminal or write one or more scripts.

1. The `EDITOR` environment variable is used to specify the user preferred text editor. Suppose to have a variable `filename` which store the name of a file. Then the command

   ```
   ${EDITOR} "${filename}"
   ```

   would open the file to edit. How would you make this command safer, considering the possibility that either of the variable (or both) is unset or null? Use a default value for `EDITOR` and abort if `filename` is unset.

2. The environment `PATH` variable contains the locations where the OS looks for commands as colon-separated list. Print the highest- and lowest-priority locations. How would you, instead, exclude the highest- and the lowest-priority locations?

3. It is common to have to deal with strings that contain some information separated by a delimiter. Consider for example the string `"b5.6789_s9876_thermalizeFromHot"` which identify a LQCD simulation. The characters `'b'` and `'s'` refer to a beta and a seed value. Such prefixes are alphabetic strings, whose length might vary. Assuming the beta and the seed values format are fixed, how would you extract

   - the beta value?
   - the seed value?
   - the postfix after the last `'_'`?
   - the beta and the seed prefixes?

   Test your code on different strings like `"beta6.0000_seed1111_continueWithNewChain"` or `"beta6.1234_s1234_thermalizeFromConf"`.

4. The `printf` builtin has the following interesting feature.

   > *The format is reused as necessary to consume all of the arguments. If the format requires more arguments than are supplied, the extra format specifications behave as if a zero value or null string, as appropriate, had been supplied.*

   It is then easily possible to concatenate strings using a delimiter.

   ```
   printf '%s_' First Second Third
   ```

   Use this command to assign the resulting string to a variable, getting rid of the trailing underscore. Run `help printf` to get inspired.

Exercise 6

Patterns gym

Create a new directory, move into it and run the command

```
touch file{1..20}{.{dat,png,txt},\ backup.dat,_bkp.png}
```

checking afterwards what happened using `ls`.

Think about how to achieve the following tasks, just using what you learnt today.

- List only files with the `.dat` extension.
- List only files with number 13 in the name.
- List only backup files and then all but backup files.
- List only files containing a space in the name.
- List all but files containing a space in the name.
- List files with a number that is multiple of 5 before the dot.

Now that you practised a bit with patterns, let us improve names of the files in this folder.

1. Rename the files containing a space replacing it by an underscore.
2. Change the `_bkp.png` suffix into `_backup.png`.
3. Add a leading 0 to numbers in files whose name contains a number smaller than 10.

Note, that you might be tempted to iterate over files, but the tasks above in this particular exercise can be achieved without using flow constructs, which will be discussed in detail tomorrow.

Bonus problem: Can you think of an easy way to print all bit strings that can build up 1 byte (8 bits)? The output of your command should look like the following.

```
00000000
00000001
00000010
00000011
  ....
11111100
11111101
11111110
11111111
```