Bash shell scripting: exercises day 2

Exercise list:

- Exercise 2.1 - FASTA to CSV converter: program logic
- Exercise 2.2 - Reusing standard input
- Exercise 2.3 - Getting user input from arguments
- Exercise 2.4 - a simple `for` loop
- Exercise 2.5 - FASTA to CSV converter: reading the input file line by line
- Exercise 2.6 - FASTA to CSV converter: testing for header lines

Notes:

- **Exercise material:** all exercise material is found in the `exercises/` directory of this Git repository. We suggest entering this directory when doing the exercises: `cd exercises/`.
- Some of the exercises build on previous exercises, so it's best if they are done in order.
- At the end of some exercises, you will find an **Additional Tasks** section. These sections contain additional tasks for you to complete, **if you still have the time after having completed the main exercise**. These sections will in principle not be corrected in class.
- **Exercise solutions:** all exercises have their solution embedded in this document. The solutions are hidden by default, but you can reveal them by clicking on the drop-down menu, like the one here-below. We encourage you to *not* look at the solution too quickly, and try to solve the exercise without it. Remember you can always ask the course teachers for help.

Exercise solution example

This would reveal the answer…

Exercise 2.1 - FASTA to CSV converter: program logic

Part 1 - description of the FASTA to CSV converter project

This is the first of a series of exercises where we will progressively build a script that converts an input in FASTA format (from a file or from standard input) into a line oriented format: TSV (`TAB` separated values).

In the input FASTA format, **each sequence starts with a header lines** (which can be recognized by its leading `>` character), followed by **a variable number of lines that contain the actual sequence** (generally the sequence contains nucleotides or amino-acids).
In the output TSC format, each sequence should be printed on **a single line**.

In other words, we want to turn something like…

```
>This is the first header
ATGCATGCGCGCGCGC
```

```
GCGCGCGCATGCATGC
GAGCGTTCGCACGAAC
>This is the second header
GCGCGCGCATGCATGC
GAGCGTTCGCACGAAC
ATGCATGCGCGCGCGC
```

…into (note: header and sequence are separated by a `TAB`):

```
>This is the first header	ATGCATGCGCGCGCGCGCGCGCGCATGCATGCGAGCGTTCGCACGAAC
>This is the second header	GCGCGCGCATGCATGCGAGCGTTCGCACGAACATGCATGCGCGCGCGC
```

Now that we are familiar with the task at hand, let's do the following:

- Create a new script file named `fasta2tsv.sh`.
- Make the script print a simple line, e.g. `Starting FASTA to TSV converter....`
- Add execution permissions to the script and test-run it.

Exercise solution: part 1

```
#!/usr/bin/env bash

# Print a test string, just to make sure the script works:
echo "Starting FASTA to TSV converter..."
```

Add execution permission and test run the script.

```
chmod a+x fasta2tsv.sh
./fasta2tsv.sh
# -> Starting FASTA to TSV converter...
```

Part 2 - write the program logic

It's generally a good idea to at least have a sketch of the program's behaviour before we start happily coding away.

In this exercise, your task is to:

- In the `fasta2tsv.sh` file, **draft the logic of the FASTA to CSV converter** in pseudocode. You can write the pseudocode as shell comments.

Exercise solution: part 2

```
#!/usr/bin/env bash

# Print a test string, just to make sure the script works:
echo "Starting FASTA to TSV converter..."

# Program logic in pseudo-code:

```

```
# Open the input file for reading.
# Loop through the file **one line at a time**, and for each line:

    # If line is a header line:
        # Start a new line.
        # Print line, followed by a `TAB`.

    # Else, if sequence: print the line.
```

Exercise 2.2 - Reusing standard input

In the `exercises/` directory, try to run the following command:

```
./reuse_stdin.sh data/Spo0A.msa
```

- Why doesn't the following work?

- How can we make it work?
- **Hint:** we will see this in more details in the next slides, but `$@` is a list of all input argument values passed to a script (or function).

For reference, here is the content of the `./reuse_stdin.sh` script:

```
#!/usr/bin/env bash

echo "The list of input arguments is: $@"
echo "And now we grep..."
grep Spo0
```

Exercise solution

- The problem is that the script `reuse_stdin.sh` expects to get user input on **the standard input**, but when we call the script with `./reuse_stdin.sh data/Spo0A.msa`, we are passing the input (in the form of a file name) **as an argument** (and indeed, you should see the file name `data/Spo0A.msa` printed in the list of arguments).

- To fix the problem, we should pass the content of the input file `data/Spo0A.msa` as standard input with the `<` redirection character:

  ```
  ./reuse_stdin.sh < data/Spo0A.msa
  ```

- Alternatively, we could also modify the `reuse_stdin.sh` script so that it accepts a file name as argument… this is what we will cover in the next lecture slides just now!

Exercise 2.3 - Getting user input from arguments

Write a short script named `hello_world.sh` that does the following:

- The script should **accept 2 input arguments**: `name` and `day` (in this order).
- The script should **print a sentence that contains the 2 values passed** by the user as the arguments `name` and `day`. For instance, if the user passes `John` as value for `name`, and `Wednesday` as value for `day`, the script could print something like: `Hello John, what a nice Wednesday`.
- If no or only 1 value is passed by the user, the script should **print an error message and exit** with an error code.
- Don't forget to **make the script executable** and run it with some test values to see if it works properly.

Here is how you would run this script:

```
./hello_world.sh John Wednesday
# -> Hello John, what a nice Wednesday
```

Hints:

- To exit a script, use the command `exit 0` (success) or `exit 1` (error code 1).
- To test whether a variable exists (and is non-empty), you can use `[ -z $variable ]` or `[[ -z $variable ]]` - the former works in most shells, while the latter only works in `bash` shells (we will see this in more details later in the course). These expressions **evaluate to true** if the variable is unset or empty.

Here is an example of where we test if `test_variable` is set and non-empty:

```
unset test_variable
[ -z $test_variable ] && echo "Variable 'test_variable' is not set..."
```

Exercise solution

```
#!/usr/bin/env bash

name=$1  # The 1st value passed to the script is assigned to variable "name".
day=$2   # The 2nd value passed to the script is assigned to variable "day".

[ -z "$day" ] && echo "please enter a name and day" &&  exit 1

echo "Hello $name, what a nice $day."
exit 0
```

We can then run the script with a couple of test values:

```
chmod a+x ./hello_world.sh John Wednesday
chmod a+x ./hello_world.sh
chmod a+x ./hello_world.sh John
chmod a+x ./hello_world.sh Wednesday
```

Additional tasks (if you have time):
Try to improve the script so that it can also detect when a single argument (instead of 2) is passed, and **provide a more custom error message** depending on whether a single argument or both arguments are missing.

Additional tasks solution

A simple solution is to detect whether both or only the 2nd value are missing from the input. Note that this assumes that the argument values are always passed in the correct order (first `name` and then `day`).

```
#!/bin/bash

name=$1
day=$2

[ -z "$name" ] && echo "please enter a name and day" &&  exit 1
[ -z "$day" ] && echo "please enter a day" &&  exit 1

echo "Hello $name, what a nice $day."
exit 0
```

We will see later in the course how you can make an interface that accepts named arguments, such as say `--name John --day Wednesday`.

Exercise 2.4 - a simple `for` loop

Part 1 - list files

Write a script named `list_files.sh` that:

- Accepts a **single input argument** that is a directory name.
- **Lists all files/directories located in that directory** (essentially, this script will do what `ls` does…).

Exercise solution: part 1

Script `list_files.sh`:

```
#!/usr/bin/env bash

directory=$1

for file in $directory/*; do
    echo $file
done
```

Part 2 - generate files

Write a script named `generate_empty_files.sh` that:

- **Generates a number of empty files** named `empty_file_<x>_test.txt`, where `<x>` is the number of the file (e.g. `empty_file_1_test.txt`, `empty_file_2_test.txt`, etc…).
- The script should **accept a single argument**: the number of files to generate.
- If no value is passed by the user, the script should **generate 5 files by default**.
- **Hints:**
  - To generate an empty file, you can use the `touch <file name>` command.
  - Remember the problem with expanding things like `for x in {1..$n}` that we saw in an exercise yesterday. In this exercise you will also need to get around this problem.

Example usage of `touch`:

```
touch a_new_file.txt  # Create a new (empty) file named "a_new_file.txt"
```

Exercise solution: part 2

Script `generate_empty_files.sh`:

```
#!/usr/bin/env bash

n=$1
[ -z $n ] && n=5

for ((i=1; i<=$n; i++)); do
    touch empty_file_${i}_test.txt
done
```

Alternatively, we can also use the `seq` command and **command substitution** `$( seq 1 $n )` to run our `for` loop:

```
for i in $( seq 1 $n ); do
    touch empty_file_${i}_test.txt
done
```

Additional Task: Part 3 - delete empty files

In the second part of this exercise, write a script that will automatically delete all files named `empty_file_<x>_test.txt`, but **only if they are empty**. Files that contain something should not be deleted, to avoid accidental data loss.

Hint:

- To test whether a file is empty or not, you can use `[ -s <file> ]` (or `[[ -s <file> ]]`, as long as you are in bash), which evaluates to **true** if the file is *not* empty.

- Example usage of `[ -s <file> ]`:

```
[ -s data/sample_01.dna ] && echo "file is not empty"
```

Important: when performing a **destructive** task in a loop, it's always best to run your script with prefixing the destructive command with `echo` as a test run, to make sure it works as expected!

- To be extra safe, you can also use `rm -i` instead of `rm` so that you are asked for confirmation to delete a file each time.

Additional Task solution: part 3

Script `delete_empty_files.sh`:

```
#!/usr/bin/env bash

for file in empty_file_*_test.txt; do
    [ -s "$file" ] && continue
    rm -i "$file"
done
```

Alternatively, we could also delete files when they are empty, instead of skipping non-empty ones as in the solution above. Note the usage of `!` as **logical not** (inverses the value of a boolean).

```
#!/usr/bin/env bash

for file in empty_file_*_test.txt; do
    [ ! -s "$file" ] && rm "$file"
done
```

Exercise 2.5 - FASTA to CSV converter: reading the input file line by line

Reuse the script **`fasta2tsv.sh`** that you created earlier (it contains the pseudocode for the FASTA to CSV converter) and modify it so that it does the following:

- **Reads** the content of the **standard input** line by line.
- **Prints** only the first character of each line.

Then test you script of the file `exercises/data/bee-18S.dna`.

Note: As we are here essentially re-inventing the command **`cut -c1`**, we can test that our script works properly by comparing its output to the output of the `cut -c1` command. This can be done using **`diff -s`**, as shown below. The output of the **`diff`** command should print that both (virtual) files are identical.

```
diff -s <(./fasta2tsv.sh < data/bee-18S.dna) <(cut -c1 data/bee-18S.dna)
```

Exercise solution

```
#!/usr/bin/env bash

while read line; do
    first_char=${line:0:1} # 1st char of line.
    echo "$first_char"
done
```

Now we can test our script with some input:

```
# This should print A, B and C on separate lines.
echo -e "A line\nB line\nC line" | ./fasta2tsv.sh

# This should print the first character of each line of the input file.
./fasta2tsv.sh < data/bee-18S.dna
```

Exercise 2.6 - FASTA to CSV converter: testing for header lines

Update your `fasta2tsv.sh` script so that it now prints the first character **only if the line starts with the character `>`**, which indicates a header line in FASTA format.
The output of your script should thus only print `>` characters (one per line).

- **Question:** How many sequences are there in the `exercises/data/bee-18S.dna` file?

Note: To test that your script works properly, you can run the following command, and it should tell you that the (virtual) files are identical.

```
diff -s <(./fasta2tsv.sh  < data/bee-18S.dna | wc -l) <(grep -c "^>" data/bee-18S.dna)
```

Exercise solution

```
while read line; do
    first_char=${line:0:1} # 1st char of line.
    [ $first_char == ">" ] && echo "$first_char"
done
```

The number of sequences in the `data/bee-18S.dna` file is simply the number of lines printed by our updated `fasta2tsv.sh` script (because each sequence starts with a header line):

```
./fasta2tsv.sh < data/bee-18S.dna | wc -l
```