Bash shell scripting: exercises day 3

Exercise list:

- Exercise 3.1 - Output formatting
- Exercise 3.2 - Add input argument support to `fasta2tsv`
- Exercise 3.3 - Introduction to functions
- Exercise 3.4 - Removing code duplication in `fasta2tsv`
- Exercise 3.5 - Final touches to `fasta2tsv`

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 3.1 - output formatting

In this exercise, **we continue to build the FASTA to TSV converter script** (`fasta2tsv.sh`), that we started earlier. At this point, your script should look roughly like this:

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

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

Part 1 - printing the sequence as a single line

Your first task is to modify your `fasta2tsv.sh` script so that it takes the correct action (i.e. does the correct printing) for each line of the input FASTA file, depending on whether it is a **header** or a **sequence line**.

Reminder: the objective of this script is to convert an input in FASTA format into a line oriented format, TSV (`TAB` separated values), where each sequence in the original FASTA sequence is 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     ATGCATGCGCGCGCGCGCGCGCATGCATGCGAGCGTTCGCACGAAC
>This is the second header    GCGCGCGCATGCATGCGAGCGTTCGCACGAACATGCATGCGCGCGCGC
```

Tip: To quickly test your script while developing it, you can run it only on a subset of the input file (this takes less space on your screen), for instance with:

```
head -n50 data/bee-18S.dna | ./fasta2tsv.sh
```

Exercise solution: part 1

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

while read line; do
    # Get the 1st char of the current line.
    first_char=${line:0:1}

    # If the line is a header, start a new line and skip the leading ">".
    # Otherwise, just append the line.
    if [[ '>' == "$first_char" ]] ; then
        printf '\n%s\t' "${line:1}"
    else
        printf "%s" "$line"
    fi
done
```

Part 2 - formatting details

If not already the case, make sure that your script:

- Does not print an unnecessary newline at the top of the output file.
- Ends the output file with a newline character.

Exercise solution: part 2

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

# Start by printing the first line of the file followed by a tab. This
# Assumes (rather reasonably) that the 1st line is a header.
read line
printf "%s\t" "${line:1}"

while read line; do
    # Get the 1st char of the current line.
    first_char=${line:0:1}

    # If the line is a header, start a new line and skip the leading ">".
    # Otherwise, just append the line.
    if [[ '>' == "$first_char" ]] ; then
        printf '\n%s\t' "${line:1}"
    else
        printf "%s" "$line"
    fi
done

# Print a newline at the end of the file.
printf "\n"
```

Exercise 3.2 - Add input argument support to `fasta2tsv`

Our `fasta2tsv.sh` script now does what it should, but it is not very versatile: it only accepts input from stdin and the separator of the output file is hard-coded to be a `TAB`.

Your task is to **improve the `fasta2tsv.sh` script so that it:**

- **Accepts both stdin and a filename argument as input**. This means that you should be able to run your script as both:

  ```
  ./fasta2tsv.sh < input_fasta_file
  ./fasta2tsv.sh input_fasta_file
  ```

- **Add support for an optional argument `-s` or `--separator`** that lets the user specify the field separator to be used in the output. If no separator is specified (i.e there is no `-s`/`--separator` passed on the command line), the script should default to use `TAB`.
  In other words, you should be able to run you script like this:

  ```
  ./fasta2tsv.sh -s "," input_fasta_file  # Output file will be comma delimited.
  ./fasta2tsv.sh input_fasta_file         # Output file will be TAB delimited.
  ```

Hints:

- When we discussed input (see course slides), we saw that the **`exec`** keyword can redirect I/O from within a script.
- The **`shift`** keyword can be used to remove arguments from the beginning of the argument list.
  - `shift` will remove a single argument, so that `$2` will become `$1`, `$3` becomes `$2`, etc.
  - `shift 2` removes 2 arguments, `shift 3` removes 3, etc.

Exercise solution

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

# If the first argument is `-s` or `--separator`, then the second argument is
# the separator.
FIELD_SEPARATOR=$'\t'           # ANSI quoting: understands \t, \n etc.
if [[ "$1" == "-s" ]] || [[ "$1" == "--separator" ]] ; then
    FIELD_SEPARATOR=$2
    # Remove the first 2 arguments in the list, so that the filename
    # argument (if specified) is now `$1`.
    shift 2
fi

# If there still is an argument, treat it as a filename and redirect standard
# input to that filename. This is like '<' on the command line, except it's
# done from within the script.
[[ "$1" ]] && exec < "$1"

# Assumes (rather reasonably) that the 1st line is a header.
read line
printf "%s%s" "${line:1}" "$FIELD_SEPARATOR"

while read line; do
    first_char=${line:0:1}  # Get the 1st char of the current line.
    if [[ '>' == "$first_char" ]] ; then
        printf '\n%s%s' "${line:1}" "$FIELD_SEPARATOR"
    else
        printf "%s" "$line"
    fi
done

printf "\n"
```

Note: `[[ "$1" ]] && exec < "$1"` is the short form for:

```
if [[ "$1" ]] ; then
    exec <"$1"
fi
```

The short form is often used where there is a single command to execute in a `if` block.

Exercise 3.3 - introduction to functions

Have a look at the script `exercises/compute_mean_values.sh`: it computes the mean values for 3 different arrays of numbers (`weights`, `lengths` and `ages`) and prints them to stdout.

In particular, you should note the usage of a function named `mean`, which computes the mean value of a sequence of integer numbers. As you can see, this function is called multiple times in the script, Removing code duplication.

If you try to run the script, you should get the following output (along with some error messages that you can ignore for now):

```
Age:    mean=94.00   # This is wrong: it should be 94.2
Weight: mean=20.00   # This is the correct value.
Length: mean=3.00    # This is wrong: it should be 15.12
```

The problem with this output is that only the mean value for `Weight` is correct: the values for `Age` and `Length` are wrong.
So let's try to fix this!

Part 1 - add support for floating point values

As we have seen from running `compute_mean_values.sh`, in its current form, the `mean` function only gives the correct result when all input values are integers, and when the mean value itself is also an integer.

Your first task in this exercise is therefore to **modify the `mean` function** so that:

- It supports floating point inputs.
- It can compute non-integer mean values.

Hints:

- One of the easiest ways to do floating point arithmetic in the shell is to use the **`bc` command**.
- The way that `bc` works is by passing it a string with the expression to evaluate. Example: `echo "1.1 + 2.2" | bc`.
- To perform floating point divisions, use `bc -l`. E.g. `echo "3 / 2" | bc -l`
- You can check your implementation by verifying that you get the following mean values:

```
Age:    mean=94.20
Weight: mean=20.00
Length: mean=15.12
```

Exercise solution: part 1

For the full script, see `exercises/solutions/compute_mean_values_part1_solution.sh`

```
# Function that computes the mean of an array of values, with support
# for floating point values.
function mean {
    local sum=0
    for n in "$@"; do
        sum=$(echo "$sum + $n" | bc -l)
    done
    echo "$sum / $#" | bc -l
}
```

Part 2 - add a `sum` function and refactor the script

We now would like to **add a function** that **computes the sum** of an array of values.
Specifically, you should do the following:

- **Write a `sum` function** that computes the sum of an array of values. It should support floating point values.
- **Modify `compute_mean_values.sh`** so that it prints the sum of each array, in addition of its mean.
- **Modify the `mean` function** in the script so that it uses the new `sum` function to compute the sum of values (i.e. no code duplication).

When implemented correctly, your script should output:

```
Age:    sum=471.00  mean=94.20
Weight: sum=100.00  mean=20.00
Length: sum=75.60   mean=15.12
```

Exercise solution: part 2

For the full script, see `exercises/solutions/compute_mean_values_part2_solution.sh`.

```
# Function that computes the sum of an array of values.
function sum {
    local sum=0
    for n in "$@"; do
        sum=$(echo "$sum + $n" | bc -l)
    done
    echo $sum
}

# Function that computes the mean of an array of values.
function mean {
    echo "$(sum $@) / $#" | bc -l
}
```

```
# Print sum and mean values.
printf "Age:\tsum=%.2f\tmean=%.2f\n" \
    $(sum "${ages[@]}") \
    $(mean "${ages[@]}")
printf "Weight:\tsum=%.2f\tmean=%.2f\n" \
    $(sum "${weights[@]}") \
    $(mean "${weights[@]}")
printf "Length:\tsum=%.2f\tmean=%.2f\n" \
    $(sum "${lengths[@]}") \
    $(mean "${lengths[@]}")
```

Bonus: actually, since we are now using `bc` to compute the sum, we could also rewrite the `sum` function in a more efficient (and elegant) way, avoiding having a `for` loop altogether...

```
# Function that computes the sum of an array of values.
function sum {
    echo $@ | tr " " "+" | bc -l
}
```

Additional Task (if you have time): Part 3 - add a `variance` function

Add a function that **computes the variance** of each array, and modify `compute_mean_values.sh` so that is additionally prints the variance values.

- **Hint:** The variance is computed as: `(x - mean(x))^2 / (n-1)`, where `n` is the total number of elements in the array.

When implemented correctly, your script should output:

```
Age:    sum=471.00  mean=94.20  var=250.70
Weight: sum=100.00  mean=20.00  var=62.50
Length: sum=75.60   mean=15.12  var=4.12
```

Exercise solution: part C

Here is an implementation of a function computing the variance of an array of values. For the full script, see `exercises/solutions/compute_mean_values_part3_solution.sh`.

```
# Function that calculates the sample variance of an array of values.
function variance {
    local mean_value=$(mean $@)

    # Compute the sum of squared difference to the mean.
    local squared_diff_sum=0
    for n in "$@"; do
        squared_diff=$( echo "($n - $mean_value)^2" | bc -l )
        squared_diff_sum=$( echo "$squared_diff_sum + $squared_diff" | bc -l )
```

```
    done

    # Return the sample variance.
    echo "$squared_diff_sum / ( $# - 1 )" | bc -l
}
```

Exercise 3.4 - Removing code duplication in `fasta2tsv`

Your task is to remove code duplication from your `fasta2tsv.sh` script (as of Exercise 3.2) by using **functions**.
Specifically, let's put the `printf` statements that print the sequence headers into a function that we can then call in our script.

Exercise solution

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

# Define a function that prints the header of a FASTA sequence, followed by
# the field separator.
function print_header {
    local header=$1
    printf '%s%s' "${header:1}" "$FIELD_SEPARATOR"
}

FIELD_SEPARATOR=$'\t' # ANSI quoting: understands \t, \n etc.
if [[ "$1" == '-s' ]] ; then
    FIELD_SEPARATOR=$2
    shift 2 # so that any filename argument is now $1
fi

# If a file name was passed, redirect its content to stdin.
[[ "$1" ]] && exec < "$1"

# Assumes (rather reasonably) that the 1st line is a header
read line
print_header "$line"

while read line; do
    # 1st char of line
    first_char=${line:0:1}
    if [[ '>' == "$first_char" ]] ; then
        printf '\n'
        print_header "$line"
    else
        printf "%s" "$line"
    fi
```

```
done

printf "\n"
```

Exercise 3.5 - Final touches to `fasta2tsv`

Let's add a few more options to our `fasta2tsv.sh` script:

- `-s`: to specify a **custom field separator** for the output. We already implemented this option in the previous exercise, but now we can refactor our code.
- `-h`: prints a **help message** explaining the usage of the script.
- `-t`: Adds a **header row** to the output (`-h` being already taken, we use `-t` as in "title").

Important:

- All arguments must be implemented as to be optional - the script should still work without them.
- The script should accept the optional arguments **in any order**. Only the input file name (if any) must obligatorily be passed in last position.
- To improve code readability, you should implement the printing of the "help" message in a separate function.

Coding tip: to do some debugging while writing loops, you can temporarily add a `printf` or `echo` command inside the loop the see the value of a specific variable inside the loop.

Exercise solution

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

# Function that displays the help information.
function show_help {
    cat <<END
"usage: fasta2csv [options] <fasta file>"

Options:
    -s: <character> specify separator (default: TAB)
    -t: output headers
    -h: this help
END
}

function print_header {
    local header=$1
    printf '%s%s' "${header:1}" "$SEPARATOR"
}
```

```
# Set default values: they may be overridden by options.
WITH_HEADER=    # Note: we could use WITH_HEADER=0, but then we should test differently.
SEPARATOR=$'\t' # ANSI quoting: understands \t, \n etc.

# Process input values passed by the user.
while [[ "$@" ]]; do

    # Note: while building our code, we might add a print statement to
    # see what is going on.
    #printf '$1: %s\n' "$1"

    case "$1" in
        -s )
            #printf "separator: '$2'\n"      # Used while debugging.
            SEPARATOR=$2
            shift 2
            ;;
        -t )
            #echo "Header flag is ON"        # Used while debugging.
            WITH_HEADER=1
            shift
            ;;

        -h )
            show_help
            exit 0     # Nothing more to do, so we exit the script early.
            ;;
        * )
            #echo "End of options detected"   # Used while debugging.
            break
            ;;
    esac
done

# If there still is a value in the input argument list, treat it as a filename
# and redirect standard input to that filename.
[[ "$1" ]] && exec < "$1"

# If requested, add a header line (title) to the output file.
[[ "$WITH_HEADER" ]] && printf "header%ssequence\n" "$SEPARATOR"

# Print the first line on the input FASTA file. This assumes that the 1st line
# of the file is a header.
read line
print_header "$line"
```

```
# Parse the remainder of the FASTA file.
while read line; do
    first_char=${line:0:1}
    if [[ '>' == "$first_char" ]] ; then
        printf '\n'
        print_header "$line"
    else
        printf "%s" "$line"
    fi
done

printf "\n"
```

Additional Tasks:

- Add the long form options `--separator` (for `-s`), `--help` (for `-h`), and `--title` (for `-t`) to the `fasta2tsv.sh` script. The idea is that the user can then use either the short or long option when running the command. For example, both `fasta2tsv.sh -h` and `fasta2tsv.sh --help` should display the help for the script.

Additional tasks solution

The solution is simply to match multiple patterns using `|`, the **OR operator**.

```
case "$1" in
        -s | --separator )
            SEPARATOR=$2
            shift 2
            ;;
        -t | --title )
            WITH_HEADER=1
            shift
            ;;

        -h | --help )
            show_help
            exit 0
            ;;
        * )
            break
            ;;
    esac
```