Exercises bash scripting

Download the files:

https://www.dropbox.com/s/2myg3k3mchx948g/bashScripting.zip

Work locally on your laptop

The following instructions have been tested on Ubuntu. Mac and Windows users can use Ubuntu in the Virtual Machine. All programs and packages are already installed within the Virtual Machine.

1. Copy the zipped data to your computer:
   ```
   wget https://www.dropbox.com/s/2myg3k3mchx948g/bashScripting.zip
   ```

2. Unzip the data:
   ```
   unzip bashScripting.zip
   ```

Exercise 1: Modify a pipeline for read quality check and trimming using bash scripting

We will work with paired-end data sets of different insertion sizes (200, 500). The reads come from two human samples from the 1000 genomes project. We will modify a bash pipeline, which will include read quality check and trimming. With this exercise you will learn step by step how pipelines are generalized using bash scripting.

The very basic bash script you can find in the file **script.sh** (included in the downloaded zip, data files can be found within the data_script folder). Try to understand what the script is doing.

To run the script, we first need to make the file executable:

```
chmod +x script.sh
./script.sh  #you don't need to run it at that stage
```

Open the file within a text editor and start to modify it according steps below:

a) Set up variables for every program path

This will allow you to quickly change the path to the programs if a new version is out (you just need to change it once in the script instead of multiple times).

- A variable can be defined as follows: name=hello
- And can be called by typing ${name}, eg: echo ${name}

So, the path to the program can be defined in a variable:

```
#!/bin/bash
pathFastqc=/home/student/APPL/FASTQC/FastQC_v0.11.7/fastqc
...
```

And the variable can be called at the point the program should be executed:

```
${pathFastqc} -t 4 -o fastQCResults ERR000064_200_1.fastq
```

 Do that for every program in the file

b) Set up variable for working directory

Most of the times you want to store all your results into one directory. Thus, it makes sense to define a variable pointing to this working directory. This will also allow you to quickly change the working directory.

```
#!/bin/bash
pathFastqc=/home/student/APPL/FASTQC/FastQC_v0.11.7/fastqc
...
~~cd /home/student/Documents/bashScripting~~
workPath=/home/student/Documents/bashScripting
mkdir ${workPath}  #creates working directory if it doesn't exist
cd ${workPath}     #change to working directory
...

#1. Quality check: FastQC
##########################
mkdir ${workPath}/fastQCResults
${pathFastqc} -t 4 -o ${workPath}/fastQCResults
/home/student/Documents/bashScripting/data_script/ERR000064_200_1.fastq
...
```

 Make sure that you write all the output of your pipeline into the working directory

c) Set up a variable for data directory

Define a variable giving the path to the directory where the original data is stored (in our case the fastq files)

```
...
dataPath=/home/student/Documents/bashScripting/data_script

#unzip files
gunzip ${dataPath}/*.fastq.gz

#1. Quality check: FastQC
#########################
mkdir ${workPath}/fastQCResults
${pathFastqc} -t 4 -o ${workPath}/fastQCResults
${dataPath}/ERR000064_200_1.fastq
...
```

 Apply it to the whole pipeline

d) Integrate a "for-loop" in the 1. Step (quality checking).

"for-loops" can be used to iterate over all files in a folder. For example:

```
for file in ${dataPath}/*.fastq
do
  echo $file
done
```

This code will print the filenames of all .fastq files within your data folder (the * means "anything", so `${dataPath}/*.fastq` gives a list of all files in the data folder that ends with .fastq):

```
ERR000064_200_1.fastq
ERR000064_200_2.fastq
ERR000061_500_1.fastq
ERR000061_500_2.fastq
```

 Try to integrate a "for-loop" into the 1. quality check step.

e) Create an array with file names

If you have a look at the script file, you will realize that we use the names of the input files all over again. Thus, we could define them in variables. However, maybe we have more input files in another data set, thus we would like to make the pipeline independent on the number of input files. This can be reached by putting the file names in an array (somehow a list of variables). Afterwards we can iterate over the array by using a "for-loop" to execute different commands (see step f). Thus, the commands are executed as many times as the size of the array.

An array is defined as follows:

```
names=(ERR000064_200 ERR000061_500)
```

The values of an array can be accessed by its index (attention: indexes start at 0!):

```
echo ${names[0]} #this will return ERR000064_200
```

If we now apply this to our pipeline it looks as follows:

```
names=(ERR000064_200 ERR000061_500)
...
#2. quality trimming: Trimmomatic
#################################
# - remove leading and trailing low quality basses (<3) or N
# - 4 base sliding window -> remove when average quality is < 15
# - remove reads which are shorter than 20 bp
java -jar ${pathTrimmomatic} PE -threads 4 ${dataPath}/${names[0]}_1.fastq
${dataPath}/${names[0]}_2.fastq ${names[0]}_1_trimPair.fastq
${names[0]}_1_trimUnpair.fastq ${names[0]}_2_trimPair.fastq
${names[0]}_2_trimUnpair.fastq LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15
MINLEN:20

java -jar ${pathTrimmomatic} PE -threads 4 ${dataPath}/${names[1]}_1.fastq
${dataPath}/${names[1]}_2.fastq ${names[1]}_1_trimPair.fastq
${names[1]}_1_trimUnpair.fastq ${names[1]}_2_trimPair.fastq
${names[1]}_2_trimUnpair.fastq LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15
MINLEN:20
...
```

 Apply it to the whole pipeline

f) Integrate a "for-loop" in the rest of the steps.

As mentioned in step e) we can now use the "for-loop" whenever the same function is applied to the files. The easiest way to iterate over an array is as follows:

```
for name in ${names[*]}
do
  echo $name
done
```

the command `${names[*]}` returns all items in the list, which can be accessed by $name within the for-loop.

 Try to integrate "for-loops" into the rest of the steps.

g) Set up a variable for the number of threads to use

A lot of the programs can use multiple cores. To define how many threads should be used by the programs we can define a variable giving the number of usable threads. This makes it easy to adapt the pipeline to the computational resources available.

```
Nthreads=4
```

 Apply it to the whole pipeline

➔ **Now the pipeline is set up in a way that it is applicable to any number of data files and different species. You just need to adapt the variables and arrays in the header (first part of the file).**

Exercise 2:

a) Write a script to unzip all files in the "Data" folder.

b) Write a for-loop to unzip all files ending in .fastq.gz in the "Data" folder.

c) Write a script which renames all unpaired files (samples with no R2 files) to "*_unpair.fastq". Hint:
   1. Create a for loop which iterates over all R1.fastq files
   2. Test within for loop if R2 file of corresponding R1 file exists. If not, rename R1 file. (`${file%R1.fastq}` will remove R1.fastq ending of the string saved in the file variable)

d) Merge all unpaired files into one file called "Unpair.fastq"

Exercise 3:

Write a script which checks if a folder name "compressed" exist within the Data folder and if not, create one. Then, compress all fastq files (keep original files, using `gzip –c file > newFile`) and move them to the folder named "compressed" (hint: use a for-loop).

Exercise 4:

Write a script to go through all fastq files in the folder and count the number of reads in the files (line number divided by 4 gives the number of reads) and write those to a file called "readNumber.txt"

Hints:

1. Write a for loop which iterates over all fastq files
2. Save the number of lines: use `wc –l file` → this will return 2 values (number of lines and file name), therefore we need to store it into an array: use `($(…))`
3. Divide the number of lines (first entry of the array) by 4 and you will get the number of reads: `$(($nbLines[0]}/4))`
4. Write the number of reads into a file called readNumber.txt

Exercise 5:

Write a script to convert the "ERR000061.1_R1.fastq" file to a FASTA file.

Hints:

1. Check if "ERR000061.1_R1.fa" file exists, if yes remove it
2. Write a while loop which iterates through all lines of a file (use google to find out how to do it: search terms "linux iterate lines in file")
3. Use a count variable to keep track at which line we are (see presentation)
4. If we are at line 1, write header to new .fa file; if we are at line 2, write sequence to new .fa file; if we are at line 3, do nothing; if we are at line 4, reset count variable to 0 (new read block starts)

Solutions:

- Exercise 1: script_mod.sh
- Exercise 2-5: BashScripting_solutions.sh