FriBash - Introduction to UNIX and Bash

Wegmann Lab - UniFR

2021-10-21

Contents

Overview **5**
0.1 Course organisation 5

1 Installing Bash **7**

2 Introduction **9**
2.1 BASH ?? Never heard! 9
2.2 BASH is awesome! 10
2.3 Using BASH and Getting Help 10
2.4 Basic Usage and Syntax 11

3 Files **13**
3.1 Create and change directory: mkdir, cd 13
3.2 List and create empty files:ls, touch 14
3.3 Copy and move files: cp, mv 15
3.4 Delete files and directories: rm 16

4 Redirection **19**
4.1 Input, Output and Redirection 19
4.2 Appending: "»" 21
4.3 Redirecting: "<" 21
4.4 Redirecting *STDOUT* to *STDIN*: "|" 22

5 Variables **25**
5.1 Assigning data to variables 26
5.2 Accessing data stored in variables 26
5.3 Creating variables from STDOUT 26

6 Conditions & Loops **29**
6.1 Conditionals 30
6.2 For Loops 31
6.3 While Loops 31
6.4 Evaluation of commands on the fly 32
6.5 First day 33

7 Bash Scripts **35**
7.1 Using `echo` . . . 36
7.2 Using `vim` . . . 36
7.3 Using a text editor with a GUI . . . 37
7.4 Execute a Bash script . . . 37
7.5 The Unix permission system . . . 38
7.6 Changing permissions . . . 40
7.7 Paths . . . 41
7.8 Passing arguments to a script . . . 43

8 Text files **45**
8.1 Encoding . . . 45
8.2 Newline issue . . . 48
8.3 Looking at files . . . 48

9 Zipping **53**
9.1 File compression . . . 53
9.2 Compressing multiple files . . . 54
9.3 Looking at zipped files . . . 55
9.4 Zipping from STDOUT . . . 56
9.5 Exercises . . . 56

10 Filtering and Editing **59**
10.1 `grep` . . . 59
10.2 `tr` . . . 61
10.3 `sed` . . . 62
10.4 Regular expressions . . . 65
10.5 Exercises Filtering and Editing . . . 68

11 Sorting and Unique lines **71**
11.1 Matching . . . 72
11.2 Exercises Sorting . . . 73

12 awk **75**
12.1 Basic structure . . . 75
12.2 Variables in awk . . . 77
12.3 Functions . . . 78
12.4 Extended matching . . . 79
12.5 If-Else . . . 80
12.6 Exercises awk . . . 81

13 Using R with BASH **83**
13.1 Exercises R and bash . . . 86

14 Solutions to Exercises **89**

Overview

This course **SBC.07110 - Introduction to UNIX and Bash** introduces the basics of shell script programming with Bash, the Bourne Again Shell. The course starts by introducing Bash, the standard command line tool on Unix, and by basic file and directory operations. Next, we discuss variables, loops and scripting in Bash, as well as manipulating text files. Finally, we will have a look at awk, a powerful tool for data extraction and reporting.

0.1 Course organisation

The course is organised as a self-studying tutorial, with all material and exercises provided through this website. You can follow the course in presence (see rooms here) or online. In any case, please add yourself to our Teams chat by following the instructions on moodle. If you have a students.unibe.ch address, we will have to add you with your eduroam nickname (which should look something like this: xw12q032@campus.unibe.ch). Please send this nickname to xenia.wietlisbach@unifr.ch instead. You can ask questions in the teams chat, and we will also post information about rooms/exams in there.

The exam will take place on **Friday, 22. October 2021 at 15.15 in room 001 (PER 17)**. It will consist of exercises just as those you will do during the course.

You will need the file BanthracisProteome.txt.gz to solve the exercises.

Chapter 1

Installing Bash

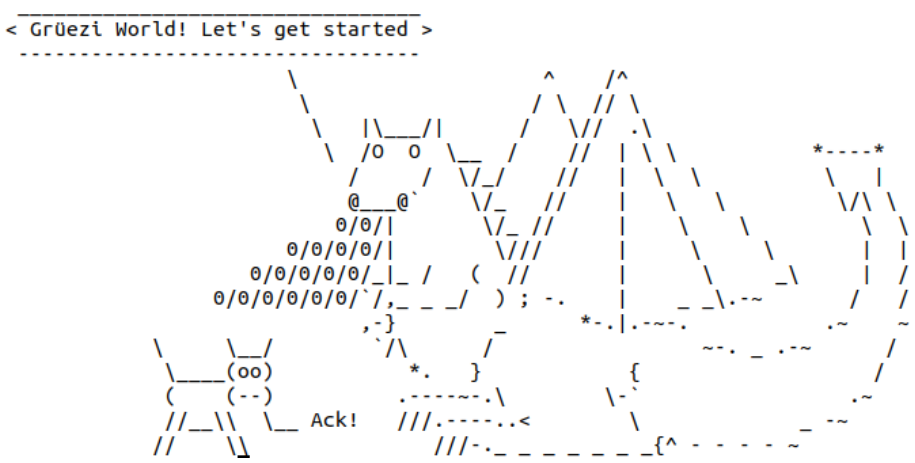
To follow this course and solve the exercises, you need to have Bash installed.

- If you run Linux on your computer, you are all set.
- If you have a Mac with OS X, you have something very similar to Bash already installed. This is sufficient for the course.
- If you have Windows, you will need to install Bash. You have three options:
    1. The recommended option is use the Ubuntu application, which is based on the Windows Subsystem for Linux (WSL). You can install it by following this tutorial. Note: This only works for Windows 10.
    2. Alternatively, you may install a virtual Linux on your computer using Virtualbox as explained in this tutorial. This also works for older Windows versions, but is a bit more complicated.
    3. You switch from Windows to Linux or set up a dual boot.

If you struggle installing Bash, let us know.

Chapter 2

Introduction



2.1 BASH ?? Never heard!

- **What is a shell?**
  - In computing, a shell is a user interface for access to an operating system's services. So, Shell is an environment in which we can run our commands, programs, and shell scripts.
  - Shells use either a command-line interface (CLI) so-called terminal or a graphical user interface (GUI) that uses icons, menus and a mouse.
  - There are different types of a shell, as there are different kind of operating systems. They share many commands among them but they also have their own set of functions.
  - Bash is a type of default shell in many Linux distributions (Ubuntu, Fedora, Debian,...). In newer MacOS versions, bash is not the default

shell but it is one of the many shells Mac users can use.

- **The Bourne Shell**
  - The original shell for Unix written by Steve Bourne at AT&T Bell Labs in 1979.
  - Small and simple with very few internal commands (called external programs for almost everything).
- **The Bourne Again Shell**
  - A reimplementation of the Bourne Shell for the GNU (Gnu is not Unix) free software movement in 1989.
  - A clone of the Bourne Shell with additional features.
  - Today, the default shell on all GNU/Linux.

2.2 BASH is awesome!

- **Why you should learn** *BASH*
  - With BASH you can use all functionalities of an OS using the keyboard.
  - With BASH you can use your computer remotely.
  - With BASH you can apply your tasks to a list of files automatically.
  - With BASH you can save your work flow (as a script) and use it again any time (repeatability).
  - With BASH you can share your work flow with others.
  - With BASH you can integrate many different programs, including R, Python, C++ etc.
- **What BASH is not**
  - BASH is not an efficient programming language. Write complicated tasks in another language and use BASH to embed it into your work flow.
  - BASH is not self explanatory.
  - BASH is a collection of external tools. Learning bash means learning many different tools.

2.3 Using BASH and Getting Help

- **How to start using** *BASH*
  - On a Unix machine, simply start a Terminal!
  - Alternatively, you can log into any Unix machine remotely using SSH.
- **Getting help**
  - Through the program man, BASH offers access to the manual pages of all tools for which manuals have been installed on the system.

```
$ man
$ man ls
```

```
- Press q to exit the man pages.
- man is only helpful if you know the name of the command... Google will likely be your best frie
```

2.4 Basic Usage and Syntax

Basic commands consist of the command, followed by arguments separated by spaces. Commands should be separated by either a semicolon ; or a new line.

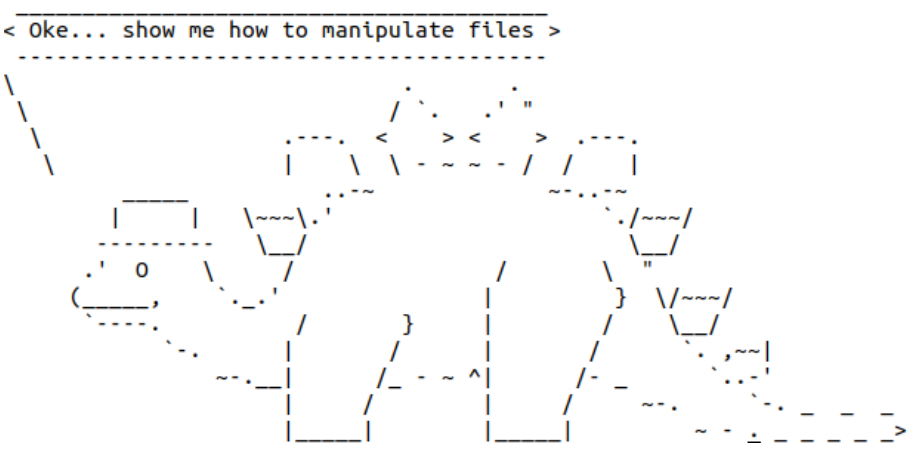
```
$ date
$ echo "Grüezi World"
Do 21 Okt 2021 07:46:50 CEST
Grüezi World
```

Chapter 3

Files

In this chapter, you will learn how to make use of the commands listed below:

```
mkdir
cd
ls
cp
touch
mv
rm
```



3.1 Create and change directory: mkdir, cd

Normally when we want to make a new directory, we would just go to our files and click on “New Folder”. The action of creating new directories and moving

through different levels can be replaced with the bash command “mkdir” and “cd”.

For example, if we want to create a new directory in our current location, we can do as follow:

```
mkdir  bashCourse
```

If we want to enter the directory we just created with mkdir, we can do that with “cd”:

```
cd bashCourse
```

If we want to go back to our previous location, we just need to type “cd ../” where “../” indicates that we want to go back to our previous position (“Go back” arrow in graphical interface).

```
mkdir  subdir
cd subdir
cd ../
```

In the example above, we first created a new subdirectory called subdir with “mkdir”, then we entered subdir with “cd” command, and finally went back to our initial position before entering subdir by typing “cd ../”

3.2 List and create empty files:ls, touch

To know what documents are located in our current folder, we can type “ls” to list the content. “ls” has many flags that help expand or specify the type of information you’re looking for. If we want to know the most common flags we can type “ls –help”. For example “ls -l” will print the content in a long list format:

```
ls
#subdir
ls -l
#total 4
#drwxrwxr-x 2 wegmannd wegmannd 4096 Sep 23 15:00 subdir
```

Sometimes we would like to create empty files for certain purpose. This can be done with the command “touch” follow by a file name or list of file names (you can type as many file names as you want separated by a space):

```
touch fileA.txt
ls
#fileA.txt subdir
touch fileB.txt fileC.txt
ls
#fileA.txt fileB.txt fileC.txt subdir
cd subdir
```

```
touch fileD.txt
ls
#fileD.txt
cd ..
```

Another cool function of "ls" is that we can list specific content by typing "ls" follow by the files or documents we want to look at.

```
ls *.txt
#fileA.txt fileB.txt fileC.txt
ls file[A,B]*
#fileA.txt fileB.txt
ls subdir
#fileD.txt
```

In the example above, * and [] are regular expressions used to represent any character or a list of specific one., In our case we are asking for any file that contains the suffix .txt or the prefix fileA or fileB (you will learn more about it later).

3.3 Copy and move files: cp, mv

Copying a file is possible with the command "cp". We can copy any file within the same directory as long as the file name of the copy differs from the original, for instance: "cp A B"; in this case we create a copy file called B from A.

```
cp fileA.txt copy.txt; ls
#copy.txt fileA.txt fileB.txt fileC.txt subdir
```

Note: Bash will send an error message if we try to make a copy with the same original file name and within the same directory. If we want to keep the same file name for some reason then, instead of providing a different file name, we have to provide another path directory, for example: "cp A subdir/"; in this case we copy file name A to subdir and keep the same original file name. There is no need to provide a file name after the path, unless we want to give a different name; for instance "cp A subdir/B"; we create a copy called B from A in the directory subdir.

Using only "cp" will not allow us to copy a full folder. In order to copy a directory, we have to use the "cp" with the flag "-r":

```
cp subdir newdir
#cp: omitting directory 'subdir'
cp -r subdir newdir; ls newdir
#fileD.txt
```

Note: the argument *-r* stands for recursive, meaning that a directory and all its content are copied.

In certain occasion, we have to move files and directories from one location to another. This can be done with the command "mv" follow by the file/directory we want to move as the first argument and the new path location where we will put the new files/directory as a second argument. For example: "mv A subdir/"; this command will move A to subdir folder, but we have to be cautious because if there is a file or directory called A inside subdir, that will be overwritten with the file or directory A that was just moved to subdir.

```
mv file[C,E].txt subdir; ls subdir
#fileC.txt fileD.txt fileE.txt
mv copy.txt fileE.txt; ls
#fileA.txt fileB.txt fileC.txt fileE.txt subdir
```

Note: "mv" can also be used to rename files and directories. As we can see from the example above, the file "copy.txt" was renamed to "fileE.txt". This occurs when we provide a file or directory name that doesn't exist as a second argument.

3.4 Delete files and directories: rm

Finally, another useful command is "rm", which delete files or directories (when providing the flag -r). We just need to type "rm" and the list of file or directories names we want to delete.

```
ls
#fileA.txt fileB.txt newdir subdir
rm fileB.txt; ls
#fileA.txt newdir subdir
```

Note: Keep in mind there is no way back! The command *rm* does not move files to the trash! files are completely deleted.

Removing a directory: rm -r

```
rm newdir
#rm: cannot remov 'newdir': Is a directory
rm -r newdir; ls
#fileA.txt subdir
rm -r *; ls
```

When providing a file name or directory, we need to be sure they are present in our current location, otherwise we have to write a path that take us to that file or directory. For example, suppose we are in our home directory that contains subdir, so if we want to copy or delete a file that is in subdir, we then have to type the path to subdir + the file name: "rm subdir/A" this will delete A within subdir.^

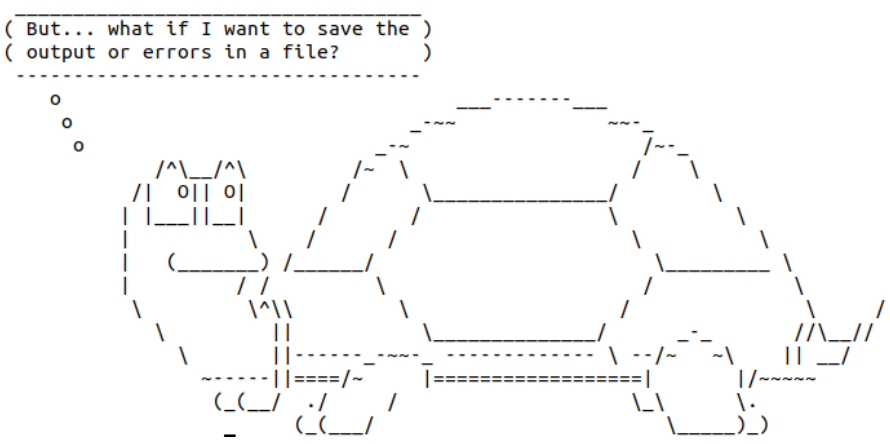
3.4.1 Exercises: Files and directories

See Section 14.0.1 for solutions.

1. Create a directory “bashExercises”. Enter the directory. In here, we will do all the exercises in order to keep track of our files.
2. Create two directories named ”human” and ”chimp”.
3. Enter directory ”human” and create three empty files named ”DNA.txt”, ”brain.txt” and ”food.txt”. Check if they are present.
4. Copy food.txt into the chimp directory.
5. Change into the chimp directory and verify that a copy of food.txt is there.
6. Move DNA.txt from the human directory to the chimp directory. Verify that human does not contain DNA.txt anymore, but that it is now in chimp.
7. Get the file sizes of all files in chimp directory and of all files in human directory while still in chimp directory.
8. Remove file brain.txt from human directory while still in chimp directory.
9. Remove the whole directory human while still in chimp directory..

Chapter 4

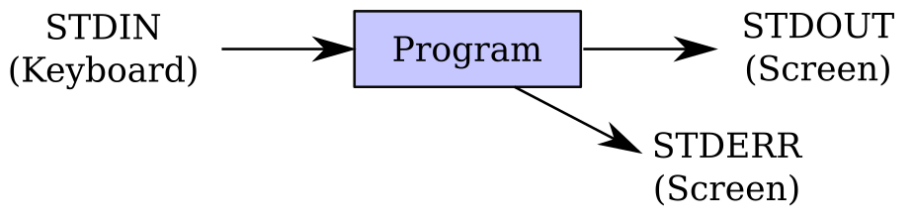
Redirection



4.1 Input, Output and Redirection

Standard in- and output

Each process has always these three "files" open: *STDIN* (input), *STDOUT* (output) and *STDERROR* (error). By default, *STDIN* is attached to the keyboard (typing the file name) and the others to the screen (displaying output and error).



Redirection

All three "files" may be redirected. For instance, *STDOUT* can be redirected to a file using the operator ">".

Redirecting STDOUT to a File (Example 1)

```
echo "Hello world!"
#Hello world!
echo "Hello world!" > out.txt
cat out.txt
#Hello world!
```

Redirecting STDOUT to a File (Example 2)

```
ls out.txt other.txt
#ls: cannot access other.txt: No such file or directory out.txt
ls out.txt other.txt > ls.txt
#ls: cannot access other.txt: No such file or directory
cat ls.txt
#out.txt
```

In the previous examples, the generated output of "echo" and "ls" is redirected using ">" (error message is not).

If we want to redirect *STDERR* to a file, we need to use "2>"

```
ls out.txt other.txt 2> ls.err
#out.txt
cat ls.err
#ls: cannot access other.txt: No such file or directory
```

We now save the error message in a file called ls.err by using "2>"

Of course, it can be annoying to type the same command twice to save the output and the error message in different files. So, we can simply type "2>" to redirect *STDERR* after redirecting *STDOUT* using ">" on the same line code.

```
ls out.txt other.txt > ls.txt 2> ls.err
ls out.txt other.txt > ls.all 2>&1
```

Note: 2>&1 means "redirect *STDERR* to the file where *STDOUT* goes", in our case it would go to ls.all

4.2 Appending: "»"

Append *STDOUT* or *STDERR* **to an existing file**

Keep in mind that using the operator ">" always overwrites the destination file, meaning that if we redirect *STDOUT* to an already existing file, the content of this file will be overwritten without warning!

If we want to continue appending the output to an existing file, we then have to use the operator "»" (two >).

Appending *STDOUT* to a file

```
echo "Line 1" > out.txt; cat out.txt
#Line 1
echo "Line 2" > out.txt; cat out.txt
#Line 2
echo "Line 3" >> out.txt; cat out.txt
#Line 2
#Line 3
```

Once we finish appending *STDOUT* to a file, that file will contain all previous outputs as shown in the example above.

We can also append *STDERR* and *STDOUT* to an existing file using "»" and the same operator as before "2>&1"

```
echo "Line 1" > out.txt
ls out.txt other.txt >> out.txt 2>&1
cat out.txt
#Line 1
#ls: cannot access other.txt: No such file or directory out.txt
```

4.3 Redirecting: "<"

Redirecting *STDIN*

Many standard commands of BASH like "head" are reading input from *STDIN* if no file is provided.

Using "head" on a file

```
ls > ls.txt
cat ls.txt
#ls.all
#ls.err
#ls.txt
#out.txt
head -n2 ls.txt
```

```
#ls.all
#ls.err
```

We can use head from *STDIN*

```
head -n2 #press Enter
#line 1
#line 1
#line 2
#line 2
```

In the example above, "head -n2" will print 2 lines in the screen but Enter is pressed without providing a file name, so we will have to type two new lines in order to validate the condition.

STDIN can be also redirected to a file using the operator "<". However, in many cases this is redundant.

```
head -n2 < ls.txt
#ls.all
#ls.err
```

4.4 Redirecting *STDOUT* to *STDIN*: "|"

Often it is handy to redirect the *STDOUT* of one command to the *STDIN* of another command. This can be done using the pipe operator |.

```
ls | head -n2
#ls.all
#ls.err
ls | head -n2 | sed 's/ls/blah/'
#blah.all
#blah.err
head -n2 < ls.txt | sed 's/l/L/'
#Ls.aLL
#Ls.err
echo "5 + 21 * 30" | bc
#635
```

We redirect the *STDOUT* of "ls" to the *STDIN* of "head" and so on. Basically, we pass the output of one command as the input for the next command and we can do it as long as the *STDIN* is valid.

4.4.1 Exercises: Redirection

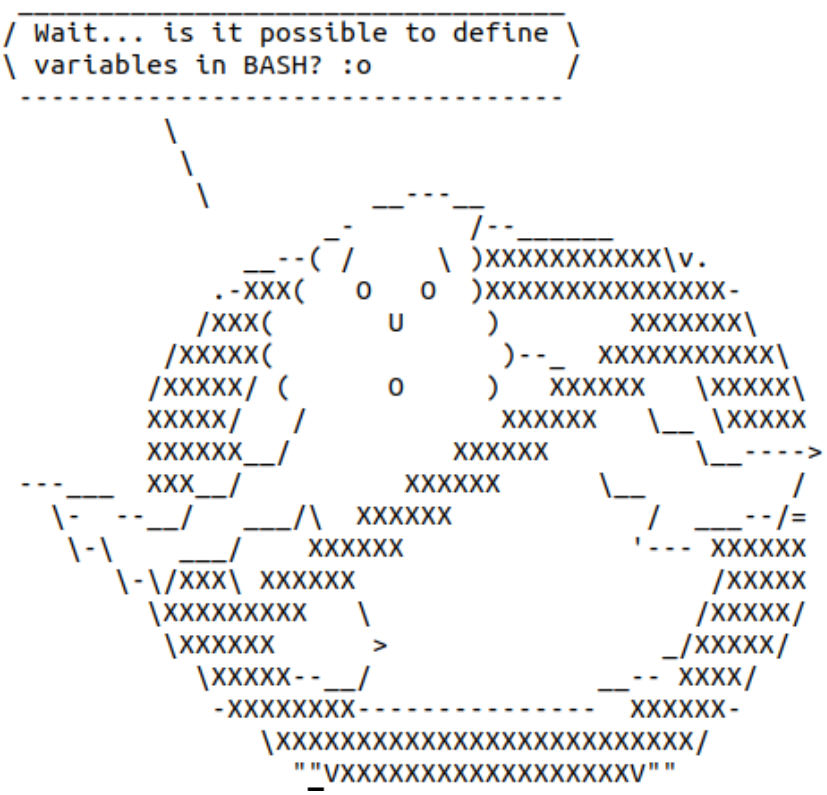
See Section 14.0.2 for solutions.

1. Create a directory "human" and enter it. Use echo to write "completely empty" to a file called brain.txt.

2. Print the content of brain.txt to the screen.
3. Append "... but still so smart" to "brain.txt".

Chapter 5

Variables



5.1 Assigning data to variables

Any string can be stored in a variable by using the operator = without spaces before and after, if spaces are added between the operator "=" an error will be prompt. Use quotes (") to include spaces and backslashes (\) to escape special characters like".

Declaring variables

```
a = 10
#a: command not found
a=10
b=Hello world!
#world!: command not found
b="Hello world!"
c="Be careful with "!"
c="Be careful with \"!"
```

5.2 Accessing data stored in variables

Before executing a command *BASH* is replacing all variables with its content. To indicate a string as a variable, put the character *$* in front and use *{* and *}* to avoid ambiguity.

```
echo $b
#Hello world!
echo "Dan says: $b"
#Dan says: Hello world!
echo "D${a}n"
#D10n
$a
#10: command not found
```

5.3 Creating variables from STDOUT

BASH offers an easy way to store the results of a command in a variable. Simple encompass a command within parenthesis and put a $ in front.

```
seq 1 10
#1 2 3 4 5 6 7 8 9 10
numbers=$(seq 1 10)
echo $numbers
#1 2 3 4 5 6 7 8 9 10
result=$(seq -s+ 1 10 | bc)
echo $result
#55
```

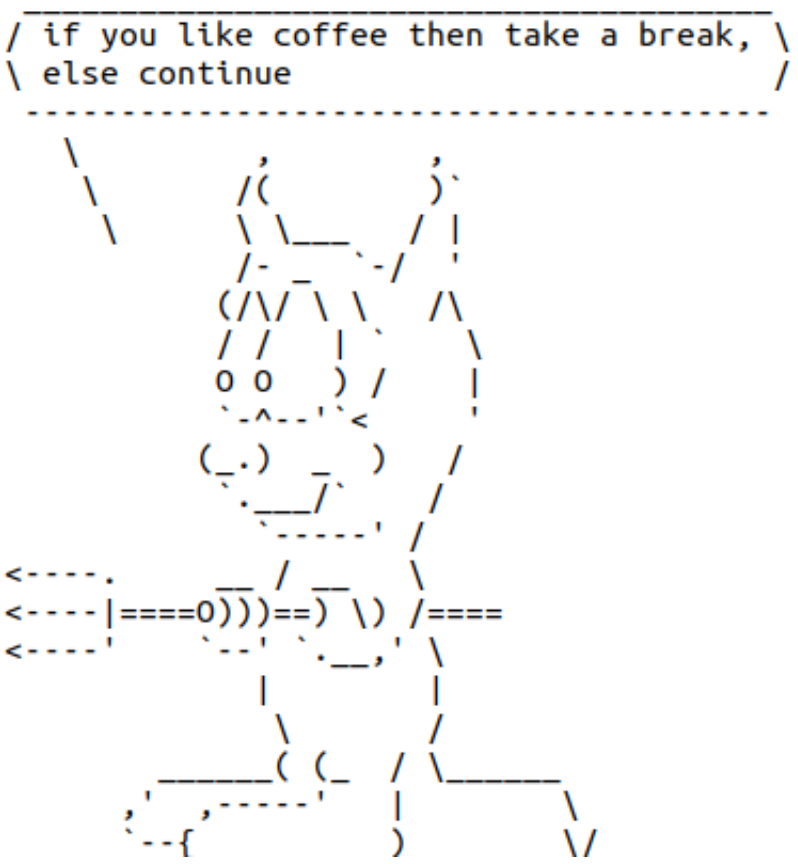
5.3.1 Exercises: Variable declaration

See Section 14.0.3 for solutions.

1. Declare a variable *course* that contains "Introduction to UNIX" and another variable *chapter* that contains "Chapter 4". Using the defined variables, print to the screen the following message: "Introduction to UNIX: Chapter 4!"

Chapter 6

Conditions & Loops



6.1 Conditionals

The if [..] .. then .. else .. fi **statement**

This construction allows to execute commands conditional on the evaluation of other expressions. They are to be written on different lines or separated by semicolons (;).

Declaring variables from output

```
if [ "foo" = "foo" ]; then echo "true!"; fi
#true!
if [ 1 > 20 ]; then echo "true!"; else echo "false!"; fi
#true!
if [ 1 -gt 20 ]; then echo "true!"; else echo "false!"; fi
#false!
```

By default, the expression is evaluating the arguments as strings. To force a numerical evaluation, **-lt** (<), **-gt** (>), **-le** (<=), **-ge** (>=), **-eq** (==) or **-ne** (!=).

File test operators

BASH offers built-in tests for the status of files.

- The most important are:
  - -e tests if a file / directory exists
  - -f tests if it is a regular file (no directory)
  - -d tests if it is a directory
  - -s tests if the size of a file is > zero

File test operators

```
if [ -f out.txt ]; then echo "true"; else echo "false"; fi
#true
if [ -d out.txt ]; then echo "true!"; else echo "false"; fi
#false!
```

Testing for not

To test for the opposite, simply put an exclamation mark ! before the operator.

File test operators

```
if [ ! -f out.txt ]; then echo "true"; else echo "false"; fi
#false
if [ ! 10 -gt 100 ]; then echo "true!"; else echo "false"; fi
#true!
```

6.2 For Loops

The *for* loop

The *for* loop let's you iterate over a series of 'words' within a string. The basic syntax consists the commands *for*, *do* and *done*.

for loops

```
for i in hello world; do echo $i; done
#hello
#world
x="one two three"
for y in $x
do echo $y | sed 's/t/T/'
done
#one
#Two
#Three
```

6.3 While Loops

The *while* loop

while executes code while the control expression is true and until it is false.

The while loop

```
COUNTER=0
while [ $COUNTER -lt 3 ]; do
echo "The counter is $COUNTER"
let COUNTER=COUNTER+1
done
#The counter is 0
#The counter is 1
#The counter is 2
```

We can also use a while loop to read over all lines of a file. For that we need to use "while read" follow by a variable name and use the operator "<" to redirect the *STDIN* to a file. Keep in mind, in this case a counter is not needed since the looping while be over once we reach the end of the file.

```
while read line; do
echo "Line: $line"
done < ls.txt
#Line: ls.all
#Line: ls.errl
#Line: ls.txt
#Line: out.txt
```

The variable line contains every time a new line from ls.txt (input is redirected with <)

6.4 Evaluation of commands on the fly

Capturing *STDOUT* as string

BASH offers an easy way to evaluate a specific command and then using the resulting *STDOUT* for further evaluation: simply put backticks ‘ or $() around the command.

Evaluate commands on the fly

```
for i in seq 1 3; do echo $i; done
#seq
#1
#3
for i in `seq 1 3`; do echo $i; done
#1
#2
#3
for i in $(seq 1 3); do echo $i; done
#1
#2
#3
echo "I like `echo "2+5" | bc`!"
#I like 7!
if [ 1 -gt $(echo "3 - 4" | bc) ]; then echo "true!"; else echo "false!"; fi
#true!
```

If we don’t provide the backstick or $(), the for loop will take the given command as a list of strings instead of evaluating it.

6.4.1 Exercises: Conditions and Loops

See Section 14.0.4 for solutions.

1. Go to directory chimp (previously created in Exercises Chapter 2), write a loop to add the words ”bananas”, ”mangos” and ”ants” to the file food.txt
2. Write a loop to add the word ”banana” 100 times to food.txt
3. Write a loop that writes 100 times ”ACGT” to DNA.txt, but all 100 times on the same line. Hint: check the man pages for echo to find a way!
4. Write a loop that reads each line of food.txt and prints it, starting with ”Line n = ” followed by the actual line. Here, n is the number of the line (Line 1 = , Line 2 = , . . .)

6.5 First day

Figure 6.1: ;)

Chapter 7

Bash Scripts

Until now, we have been executing all Bash commands on the command line. This is perfectly fine for small commands, however, for long and complicated Bash pipelines, we might want to save our work as a Bash script. A Bash script is a plain text file which contains a series of commands. Anything we can run on the command line can be put in a script, and the other way around.

Saving what we do as scripts allows us to reuse our work on new data sets, to share analysis pipelines with others and - most importantly - to remember later what we have actually done!

Writing scripts in very easy. Let's have a look at a simple example, a script that prints "Hello world!":

```
#!/bin/bash
echo 'Hello world!'
```

A bash script always starts with the line `#!/bin/bash`. The `#!` in here is called "shebang". It marks the beginning of a script. Right after the shebang is the path to the program (interpreter) that should be used to run the script. In our case, shebang tells the computer to use `/bin/bash` to run the script, which is simply the Bash shell. If our script contained python code instead, this line would need to be modified, such that the computer would interpret the code as python.

To wrap up: each Bash script starts with `#!/bin/bash` to tell the computer that the following code must be interpreted as Bash code.

The rest of the file contains the commands that Bash should execute, in our example printing "Hello world!". Nothing changes compared to the normal command line. If needed, comments can be added to the script. Everything after a `#` will be ignored by Bash (except for the shebang, of course). We can thus modify our script and add a comment:

```
#!/bin/bash

# This is a boring script that prints Hello World
echo 'Hello world!'
```

Let's now discuss how we can write such a Bash script. For this, we will review three common approaches.

7.1 Using `echo`

While tedious, we can write a Bash script line by line using `echo`. We simply use the operators `>` and `>>` to write and append to a file, just like before:

```
$ echo '#!/bin/bash' > test.sh
$ echo "echo 'Hello world'" >> test.sh
```

If you decide to write a bash script in this way, you will have to commend out all the special characters, as otherwise bash will interpret them already while writing and not only when executing the file. Note that it is good practice to use the extension ".sh" for files containing Bash scripts. However, Linux doesn't care about extensions and will also accept a file as a Bash script even if we call it "test.myFantasticFancyFileFormat".

7.2 Using `vim`

The command line text editor `vim` is installed on most systems and allows us to create or modify files even remotely.

Open a console and type

```
$ vim test.sh
```

`vim` offers multiple modes. By default, we enter the *command mode*. We now need to tell `vim` what we want to do:

- Go to *insert mode*: type `i`. We can now enter text. For example, write the two lines of code from the example above.
- To exit back to *command mode*, press the `Esc` button on the keyboard.
- When in *command mode*, we can give commands starting with a colon `:`, followed by the command letter, see below for examples. To confirm, press return (the `Enter` button on the keyboard).
  - `:w` saves the file
  - `:q` quits `vim`
  - `:wq` saves the file and quits `vim`
  - `:q!` quits `vim` without saving the file
  - `:s/x/y/g` replaces all occurrences of x with y

Quitting `vim` can be very confusing at first - in fact, the stackoverflow entry about this question has been visited more than 2.4 million times, which adds up to about 80 people an hour on workdays struggling to quit `vim`.

7.3 Using a text editor with a GUI

The most convenient option is to use a text editor with a graphical user interface (GUI). The standard editor on Gnome systems in `gedit`:

```
$ gedit test.sh
```

However, this only works when the connection allows X11 forwarding. X11 forwarding is referring to a mechanism that enables you to run a remote application (such as `gedit`), but forward the output display to your local computer. This is automatically the case when using the terminal on a Linux machine. However, when connecting to a remote computer (e.g. a cluster) with ssh, add the argument `ssh -X` to enable `gedit`. In some cases, this is not possible, so knowing how to use `vim` is still very handy.

7.3.1 Running a process in the background

By default, any program launched on the command line will become the main process - our command line is "blocked" and does not accept any further commands. Hence we can continue to use the command line only after we closed the program. To launch a program in the background (as a new process), put a `&` after the command:

```
$ gedit test.sh &
```

Try it out - we can now enter commands on the command line while having the text editor open in the background.

7.4 Execute a Bash script

We have discussed three approaches on how to write Bash code to a file. Let's now see on how to launch the Bash script!

A script can be executed by calling its name

```
$ ./test.sh
bash: ./test.sh: Permission denied
```

We get a "permission denied" error, since for security reasons, no one has the right to execute a file on Unix by default. We can see the permissions for a files (and directories) by typing

```
$ ls -l test.sh
-rw-rw-r-- 1 madleina madleina 31 Okt 21 07:46 test.sh
```

Nine columns appear. Those represent the following:

1. column: the file type and the file permissions.
2. column: the number of memory blocks.
3. column: the user (the one who has administrating power).
4. column: the group of the user.
5. column: the file size.
6. column: the month the file/directory was last modified.
7. column: the day the file/directory was last modified.
8. column: the time the file/directory was last modified.
9. column: the name of the file or the directory.

7.5 The Unix permission system

Let us have a closer look at the file permissions.

Linux allows us to do pretty anything we want. This of course comes along with a lot of dangers - we might delete directories we didn't intend to or uninstall essential programs that are needed for Linux to run. In addition, also malign user inputs (like viruses) can corrupt, change or remove crucial data. To prevent this, there is the Unix permission system that secures the filesystem. It divides authorization into two levels:

1. Ownership
2. Permission

7.5.1 Ownership

Every file and directory is assigned to three types of owners:

1. *User*: The user is the owner of the file. By default, this is the person who created the file.

2. *Group*: A group can contain multiple users. By default, the group has simply the same name as the user. However, imagine a project where a bunch of people need to access a file. Instead of manually assigning permissions to each user, we could add all users to a group, and then give group permissions to the file, such that all users of this group have the same permissions.

3. *Other*: Any other user who has access to the file. This person is neither user nor part of the group. Essentially, this boils down to "everybody else".

7.5.2 Permissions

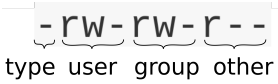
Linux defines permissions for each of the three owners described above. That way, Linux can e.g. allow me as a user to view my images, while preventing my colleague, who works on the same computer, to see them.

Every file and directory has the following three permissions defined:

1. *Read*: The read permission allows us to open and read a file, as well as to list the content of a directory. This permission is abbreviated with an `r`.
2. *Write*: The write permission allows us to modify the content of a file, as well as to add, remove or rename files in a directory. This permission is abbreviated with an `w`.
3. *Execute*: The execute permission allows us - big surprise - to execute a file. This permission is abbreviated with an `x`.

7.5.3 Viewing permissions

With this knowledge in mind, let's now inspect the first column from the `ls -l` command above.



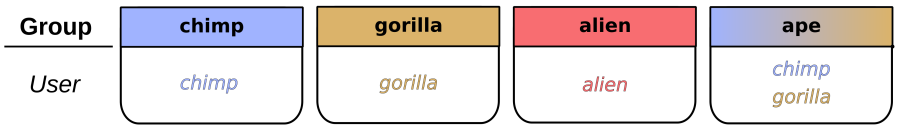
This column encodes the permissions given to the user, group and others

- The very first letter is the *file type*. The `-` implies that we have selected a file. A directory is encoded by a `d`.
- The next three letters are the permissions for the *user*. In our example, `rw-` means that the user can read and write, but not execute the file.
- The next three letters are the permissions for the *group*. As discussed above, Linux will by default add the user to a group with the same name as the user. Therefore, the group has by default also the same permissions as the user, in our example again `rw-`.
- The last three letters are the permissions for the *others*. In our example, `r--` means that the others can only read, but not write nor execute.

7.5.4 Exercises: Permissions

See Section 14.0.5 for solutions.

Consider the three users chimp, gorilla and alien. For each user, there is a group with the same name, i.e. chimp belongs to group chimp; gorilla belongs to group gorilla, and alien belongs to group alien. In addition, chimp and gorilla both belong to group ape.



For the following examples, list all users that have 1) read permissions, 2) write permissions and 3) execute permissions. Hint: the user and group permissions always supersede lower (i.e. other) permissions.

The first column represents permissions, the second column represents user, the third column represents group:

`Permissions User Group`

1) `-r--r----- chimp chimp`
2) `-rwxrw-r-- chimp ape`
3) `-r-xr--rwx alien ape`

7.6 Changing permissions

Permissions can be changed with the `chmod` command.

To add or remove a permission, use `+` or `-`, followed by the permission we would like to change (`r`, `w` or `x`).

For example, let's add the executing permission to all owners:

```
$ chmod +x test.sh
$ ls -l test.sh
-rwxrwxr-x 1 madleina madleina 31 Okt 21 07:46 test.sh
```

... and remove it again:

```
$ chmod -x test.sh
$ ls -l test.sh
-rw-rw-r-- 1 madleina madleina 31 Okt 21 07:46 test.sh
```

To change permissions of only one owner, add the owner in front (`u` for user, `g` for group, `o` for other).

For example, let's remove read permission from other:

```
$ chmod o-r test.sh
$ ls -l test.sh
-rw-rw---- 1 madleina madleina 31 Okt 21 07:46 test.sh
```

and remove write permissions from group:

```
$ chmod g-w test.sh
$ ls -l test.sh
-rw-r----- 1 madleina madleina 31 Okt 21 07:46 test.sh
```

Back to our original problem: How can we run `test.sh`? To run a script, the user (me) needs to have executing rights. Currently, we do not have this, but we can easily add it with:

```
$ chmod u+x test.sh
$ ls -l test.sh
-rwxr----- 1 madleina madleina 31 Okt 21 07:46 test.sh
```

Now, let's run the script!

```
$ ./test.sh
Hello world
```

7.6.1 Exercises: Bash scripts

See Section 14.0.6 for solutions.

Note: If you are on Windows and using the Ubuntu application, make sure to leave the directory that is under control of Windows, because Windows will not let you change permissions. Simply type `cd`, and it will jump to your Linux home directory. In there, you should be able to modify permissions.

1) Create a file "myPrecious.txt" and use `echo` to write the text "I and I alone!" into it. Make sure you are the only one allowed to read or write the file. Check the permissions using `ls`.
2) Change the permissions again to make sure that only the group permissions are set to read and write. Check that they are correct. Try to print the file content to the screen.
3) Remove the file myPrecious.txt
4) Create a directory "It's all yours!". Create a file "yours.txt" inside that directory. Check it is there.
5) Remove execution rights for yourself and try to enter the directory. Can you still see the files inside the directory?
6) Remove directory "It's all yours!"
7) Use `vim` to create a script "love.sh" that prints "I like Bash!" to STDOUT, and execute it 100 times.

7.7 Paths

You might have wondered why we write `./test.sh` instead of just `test.sh` for executing the script. The reason is simple: Bash wouldn't have found the file otherwise!

```
$ ./test.sh
Hello world
```

```
$ test.sh
Error in running command bash
```

But why?

When calling an executable such as test.sh, Bash looks for it in a list of directories that are stored in a variable called `$PATH`. This variable typically contains the following directories (separated by a colon `:`):

```
$ echo $PATH
/home/madleina/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/ga
```

Bash only looks for the executable in those directories and does not consider our current directory. Since `test.sh` is not located in one of these directories, Bash can not find it. We need to tell Bash the unique location of the file inside the filesystem - the so-called path. A path is how we refer to files and directories: it gives us the location of a file or a directory in the Linux directory system.

As a user, we need to know the path if we want to access a certain file or directory. We can specify the path as absolute or relative:

1. **Absolute path**: The absolute path specifies the location of a file from the root directory. Everything on a Linux system is located under the root directory. The root directory is the top-most directory of all other directories, like the trunk of a tree where all branches originate from. It is represented with a `/`. After the `/`, the absolute path lists all directories that need to be entered in order to get to the file, separated by `/`. We can get the absolute path with the command `pwd`:

```
$ pwd
/home/madleina/git/bash_lecture
```

We can run our script by pasting this path in front of its name:

```
$ /home/madleina/git/bash_lecture/test.sh
Hello world
```

2. **Relative path**: This is the path relative to the current location. For relative paths, we often use the following three symbols:

- `.` (a single dot): represents the directory we're in.
- `..` (a double dot): represents the parent directory (i.e. one level above).
- `~` (tilde): represents the home directory. This is the default directory that occurs after logging in. Most of our files will be located somewhere in here.

Let's make this clear with an example. In our current directory, create two new directories.

```
$ mkdir dir1 dir2
```

We can change directories using `cd`:

```
$ cd dir1
$ touch file1.txt
```

```
$ ls
file1.txt
```

Let's now go to dir2 (via the parent directory), and copy file1 from dir1 here:

```
$ cd ../dir2
$ cp ../dir1/file1.txt .
$ ls
file1.txt
```

Finally, move file1 from dir1 to the parent directory:

```
$ mv ../dir1/file1.txt ../movedFile.txt
$ ls ../dir1
```

We have only used relative paths in this example. They are very handy, since they make code much more readable: suppose we had to replace each relative path above with its absolute path, it would have been a mess!

In summary: The absolute path always starts with the root `/` directory and is independent of our current location. The relative path depends on our current location.

7.8 Passing arguments to a script

When running a script on the command line, we can pass arguments to it. Inside our script, we can use these arguments to e.g. write output to a specific directory or calculate something. There are pre-defined variables which we can use to access the command line arguments. `$0` refers to the first argument on the command line - which is simply the name of the Bash script we're executing. `$1` refers to the second argument, `$2` to the third, and so on.

For example, create a Bash script called "testArguments.sh" with the following content:

```
#!/bin/bash
echo "Argument 0: $0"
echo "Argument 1: $1"
echo "Argument 2: $2"
echo "Argument 3: $3"
```

and execute it:

```
$ chmod u+x testArguments.sh
$ ./testArguments.sh is awesome
Argument 0: ./testArguments.sh
Argument 1: is
Argument 2: awesome
Argument 3:
```

7.8.1 Exercises: Passing arguments to a script

See Section 14.0.7 for solutions.

1) Write a script called "dog.sh" that takes three arguments: the name, the age and the color of a dog. In your script, print "Here comes ..., a ... year old dog of ... color.", where you replace ... with the arguments above. Call your script with two of your favourite dogs.

2) Use `vim` to create a script "like.sh" that takes an argument and prints "I like ...!" where ... is the argument passed. Use a for loop to write "I like biology!", "I like computer science!" and "I like bioinformatics!".

3) Write a script called "reasons.sh" that takes an argument and appends it to a file called "whyILikeBash.txt". Call your script with three reasons why you like Bash (if you can't come up with a reason, you can also lists reasons why don't like it, and hopefully change your mind soon). Check the file.

4) Write a script called "append.sh" that takes two arguments: a string and a name of a file to which the string is to be appended. Use it to write the words "Bern", "Fribourg", "Lausanne" to a file called "words.txt". Check the file.

5) Write a script called "append2.sh" that takes two arguments: a string and a name of a file to which the string is to be appended. Your script should first test if the file (given by the second argument) exists. If it does not, the script should create it and write "Created by append.sh" to it. Call your script to write three words of your choice to a file called "moreWords.txt". Check the file.

6) Create a script "positive.sh" that takes a number as an argument. If the number is larger than zero, it should write the number to a file. The name of that file should be yourNumber.txt (for example, if your number is 5, the file is called 5.txt). Call your script with the numbers -10, 0 and 10, and check if only a file called "10.txt" has been written.

7) Create a script "explain.sh" that goes through all files and directories in the current directory and prints the name followed by "is a directory" or "is a file". Create some extra directories and files before running it.

8) Create a script "helper.sh" that 1) writes a script "print.sh" and 2) executes it. The script print.sh shall then take two arguments: a string and a number, and it should print the string number times to STDOUT. Finally, the script helper.sh should take the same arguments and forward them to print.sh when executing it. Run it to print 10 times "Gotcha!".

Chapter 8

Text files

There are generally two kinds of computer files:

1. **Text files**: Files that are structured as a sequence of lines of electronic text. They are readable without much processing.

2. **Binary files**: All files that are not text files. They thus require knowledge about their structure to be readable.

Text files are omnipresent because they are human readable and do not require meta data for processing. However, text files do not allow for formatting, and often use more space than clever binary files.

8.1 Encoding

Computers store 0s and 1s only. All data are thus represented in a binary sequence, that is, a sequence of 0s and 1s. The most basic unit of binary is a *bit*, which can be 0 or 1. With two bits, a computer can represent 4 different patterns: 00, 01, 10 and 11. With three bits, a computer can represent 8 different patterns: 000, 001, 010, 011, 100, 101, 110, 111. And so on.

The next larger unit is a *byte*, which consists of 8 bits. Guess: How many patterns can we represent with a byte? Well, we have eight combinations of zero or one, which results in $2^8 = 256$.

Since computers store 0s and 1s only, text needs to be encoded in 1s and 0s. There are different "maps" that encode how to get from a character (e.g. a letter, a number, a whitespace etc.) to a binary representation.

8.1.1 ASCII

ASCII stands for American Standard Code for Information Interchange. It is a very early standardized character-encoding scheme using 7 bits, originally developed for teletype machines. With 7 bits, we can store up to $2^7 = 128$ code points. Each such code point is mapped to a specific character. Below are some examples:

Binary	Decimal	Character
0000010	2	Start of text
0000100	4	End of transmission
0001001	9	Horizontal tab
0001010	10	Line feed
0001101	13	Carriage return
0110000	48	0
0110001	49	1
1000001	65	A
1000010	66	B
1100001	97	a
1100010	98	b

To form words or sentences, these bits are concatenated. For example, the sentence "ASCII is awesome!" results in the following binary code:

01000001 01010011 01000011 01001001 01001001 00100000 01101001 01110011 00100000 01100001 01110111 01100101 01110011 01101111 01101101 01100101 00100001

ASCII can represent the lower- and uppercase English alphabet (52 letters), as well as the digits 0-9, a couple of punctuation marks (.,!?), whitespaces, and other symbols (e.g. +-/#%&). The limit of 127 code points is therefore quickly reached, and many important characters, such as letters from non-English alphabets, do not have an ASCII representation.

8.1.2 UTF-8

UTF-8 stands for Universal Character Set + Transformation Format with 8 bit. UTF-8 is a more flexible encoding that can encode all possible characters in Unicode. Currently, these are 1'112'064 such characters (compared to 127 in ASCII)!

UTF-8 uses a set of one to four bytes, depending on the character. The first 256 characters - which include the characters from ASCII - are encoded by one byte. Characters that appear later are encoded as two bytes, three bytes and eventually four bytes.

If we have a sequence of bytes, like for the sentence “ASCII is awesome” above, the computer needs to know which bytes form one character. This was easy for ASCII, since each character was encoded by one byte. For UTF-8, there is a specific structure that encodes this:

Byte 1	Byte 2	Byte 3	Byte 4
Leading byte	Continuation byte		
0xxxxxxx			
110xxxxx	**10**xxxxxx		
1110xxxx	**10**xxxxxx	**10**xxxxxx	
11110xxx	**10**xxxxxx	**10**xxxxxx	**10**xxxxxx

The leading byte contains information on how many bytes are used. If only one byte is used, it starts with a 0. If two bytes are used in total, it starts with 110. And so on.

In contrast, every continuation byte starts with 10. Therefore, the computer can find the start of a coding block easily at any random position in the file by reading until a byte does not start with 10, since this is a leading byte.

The x in the representation above are then replaced with the bits of the actual character.

For example:

- The letter A is encoded by 1000001 (7 bits). UTF-8 adds a leading 0 to encode that a single byte is used, resulting in **0**1000001.

- The € sign is encoded by 1000001 0101100 (14 bits). Unfortunately, two bytes are not sufficient, since the total number of bytes must also be encoded. Two bytes use 5 out of 16 bits for the encoding of the number of bytes, therefore only leaving 9 bits for the encoding of the character. We therefore need 3 bytes. The encoding fills the bits from right to left:

  1. Fill 3^{rd} byte until it is full → **10**101100.
  2. Fill 2^{nd} byte until it is full → **10**000010.
  3. Fill 1^{st} byte until all bits were used. Then pad with zeros until byte is full → **1110**0010.

You might think that it would be a lot easier if UTF-8 would use four bytes for all characters, instead this complicated switching. The answer is easy: you save a lot of memory with smart UTF-8 encoding. A text only containing English letters would require four times more memory.

8.2 Newline issue

The end of a line has a historic legacy. In the teletype age, two different characters were required to end a line: a carriage return (CR) and a line feed (LF). Nowadays, on Unix systems, only the line feed is required. On Windows machines, both characters are used. And to make it more complicated, Mac systems used carriage return until version 9, and line feed since OS X. The important thing to realize is that depending on the system, a computer might insert different characters to end a line. This can be problematic when using the same file on a different system.

However, with a few Bash commands, we can easily replace the newline characters.

Let's consider the file test.txt, which uses both carriage return (`\r`) and line feed (`\n`) to end a line:

```
$ echo -e "Line 1\r\nLine 2" > test.txt
```

Here, `echo -e` will enable regular expression parsing, since `\n` is a regular expression for a line feed.

Let's have a look at the file. Using `cat -vE`, we can print non-visible characters using substitutes.

```
$ cat -vE test.txt
Line 1^M$
Line 2$
```

This option displays Unix line endings (`\n`) as `$` and Windows line endings (`\r\n`) as `^M$`.

If we want to get rid of the Windows line ending, we can replace the carriage return (`\r`) in the file using the following pipe:

```
$ cat test.txt | tr -d '\r' | cat -vE
Line 1$
Line 2$
```

We will learn more about `tr` later. But in this example, `tr -d` will delete all `\r` characters.

8.3 Looking at files

8.3.1 `more` and `less`

We have learned about `cat`, which prints the whole file to STDOUT. For large files, this will flood our screen extremely fast, such that it is cumbersome to scroll up to the part we wanted to see.

Thus, for large files, there are better options. Meet `more` and `less`!

```
$ more BanthracisProteome.txt
```

`more` shows one screen at a time, and allows us to scroll. For this, press space to get to the next page, and return to move down one line.

Less is more - one of many Linux jokes. `less` is an improved version of `more` that lets us navigate in both directions:

```
$ less BanthracisProteome.txt
```

Use the arrows to move up and down lines, and space to jump down one page.

Of course, we can also open a file in a graphical text editor, such as `gedit`. However, these tools usually need to read the whole file before displaying, which might take a long time for large files.

8.3.2 Line count

To count the number of lines in a file, we can use the command `wc`:

```
$ wc BanthracisProteome.txt
  515365  2987124 24285426 BanthracisProteome.txt
```

The three numbers represent the total line, word and character count, respectively. Usually, we are mostly interested in the number of lines in a file. For this, we can use the shortcut `wc -l`:

```
$ wc -l BanthracisProteome.txt
515365 BanthracisProteome.txt
```

8.3.3 Head and tail

The commands `head` and `tail` are useful when we want to extract parts of a file, for example individual lines or a set of lines. `head` will start at the beginning of the file. By default, the first 10 lines of a file are printed:

```
$ head BanthracisProteome.txt
ID   3MGH_BACAN              Reviewed;         205 AA.
AC   Q81UJ9; Q6I2S8; Q6KWK9;
DT   26-APR-2004, integrated into UniProtKB/Swiss-Prot.
DT   01-JUN-2003, sequence version 1.
DT   13-NOV-2013, entry version 69.
DE   RecName: Full=Putative 3-methyladenine DNA glycosylase;
DE            EC=3.2.2.-;
GN   OrderedLocusNames=BA_0869, GBAA_0869, BAS0826;
OS   Bacillus anthracis.
OC   Bacteria; Firmicutes; Bacilli; Bacillales; Bacillaceae; Bacillus;
```

With the flag `-n`, we can change the number of lines that are printed:

```
$ head -n 5 BanthracisProteome.txt
ID   3MGH_BACAN              Reviewed;         205 AA.
AC   Q81UJ9; Q6I2S8; Q6KWK9;
DT   26-APR-2004, integrated into UniProtKB/Swiss-Prot.
DT   01-JUN-2003, sequence version 1.
DT   13-NOV-2013, entry version 69.
```

```
$ head -n 5 BanthracisProteome.txt | wc -l
5
```

`tail`, on the other hand, starts at the end of the file. The same settings apply here:

```
$ tail BanthracisProteome.txt
DR   OMA; KKKHRVR; -.
DR   OrthoDB; EOG6X3WBN; -.
DR   ProtClustDB; CLSK687586; -.
DR   BioCyc; ANTHRA:GBAA_PXO1_0081-MONOMER; -.
DR   BioCyc; BANT261594:GJ7F-5672-MONOMER; -.
PE   4: Predicted;
KW   Complete proteome; Plasmid; Reference proteome.
SQ   SEQUENCE   53 AA;  6111 MW;  8BB41F35C12C0D07 CRC64;
     MDKKKKQRVR RAIFIGVIAM IVSLYIGNEL QDRNGKSYAP AKYFETGTKL ISY
//
```

```
$ tail -n 5 BanthracisProteome.txt
PE   4: Predicted;
KW   Complete proteome; Plasmid; Reference proteome.
SQ   SEQUENCE   53 AA;  6111 MW;  8BB41F35C12C0D07 CRC64;
     MDKKKKQRVR RAIFIGVIAM IVSLYIGNEL QDRNGKSYAP AKYFETGTKL ISY
//
```

We can chain these two commands to print e.g. lines 7-10 only:

```
$ head -n 10 BanthracisProteome.txt | tail -n 3
GN   OrderedLocusNames=BA_0869, GBAA_0869, BAS0826;
OS   Bacillus anthracis.
OC   Bacteria; Firmicutes; Bacilli; Bacillales; Bacillaceae; Bacillus;
```

In addition, `tail -n +x` prints the entire file, starting at line x. For example, if we want the entire file, except for the first two lines, we can use:

```
$ tail -n +2 BanthracisProteome.txt
```

8.3.4 Cut

`cut` is a command that divides a file into columns. Let's discuss its most important options on the first few lines of BanthracisProteome.txt, which look

like this:

```
$ head -n 2 BanthracisProteome.txt
ID   3MGH_BACAN              Reviewed;         205 AA.
AC   Q81UJ9; Q6I2S8; Q6KWK9;
```

The flag `-f` is used to specify the columns that should be extracted. This can be individual numbers (`-f 1`), ranges (`-f 1-3`), lists (`-f 1,5`) or a mixture of those (`-f 1,3-5`). For example, if we want to print the first column only, we can use:

```
$ head -n 2 BanthracisProteome.txt | cut -f1
ID   3MGH_BACAN              Reviewed;         205 AA.
AC   Q81UJ9; Q6I2S8; Q6KWK9;
```

Well... That didn't really work, did it? The output looks exactly the same as before. The reason is that `cut` does assume that the columns are separated by a tab `\t` by default, but this is not the case for our file. The first two columns are separated by three spaces. We use the `-d` flag to specify the delimiter:

```
$ head -n 2 BanthracisProteome.txt | cut -f1 -d ' '
ID
AC
```

... and this works nicely. However, if we take the second column instead:

```
$ head -n 2 BanthracisProteome.txt | cut -f2 -d ' '
```

we get an empty column. This is because cut will extract everything between the first and the second space - which is nothing! In such cases, it makes sense to use the command `tr` upfront. `tr -s` will remove multiple occurrences of a character, i.e. it will squeeze all double/triple/... spaces into a single space. This makes it much easier to parse the file with 'cut":

```
$ head -n 2 BanthracisProteome.txt | tr -s ' ' | cut -f2 -d ' '
3MGH_BACAN
Q81UJ9;
```

We will hear more about `tr` later.

Chapter 9

Zipping

The size of text files can usually be reduced through file compression. This is useful, or sometimes even necessary, when you want to backup your data, when you want to transfer files or simply to improve space efficiency on your computer.

Zipped files store information in fewer bits by removing redundant information. Most files are quite redundant, as they have the same information listed over and over again. File compression finds common patterns and replaces them with shortcuts to rid of the redundancy. Instead of listing a piece of information over and over again, a file compression program lists that information once and then refers back to it using a dictionary. For example, consider a text file containing the word "AAAABBBCCCCC". A simple compression would be to replace the each consecutive letter by the number of times it occurs: "A4B3C5". That is already much shorter than the original string, and, importantly, we didn't lose any information by compressing. Therefore, the more repetitive elements your file contains, the larger the relative gain by compression is.

9.1 File compression

You can use the commands `gzip` and `gunzip` to zip and unzip a file. When you zip a file using `gzip`, it will automatically receive the ending `.gz` and it will also appear dark red in the terminal, giving you a visual clue. When you unzip using `gunzip`, the `.gz` ending will disappear again.

```
$ gzip BanthracisProteome.txt
$ ls -sh BanthracisProteome.txt.gz
3.2M BanthracisProteome.txt.gz
```

```
$ gunzip -k BanthracisProteome.txt.gz
$ ls -sh BanthracisProteome.txt
24M BanthracisProteome.txt
```

You can see that as a zipped file, the `BanthracisProteome.txt.gz` occupies 3.2M, while the unzipped version occupies 24M, that's more than seven times less space when you zip it!

Note: `gzip` replaces the original file. Use `gzip -k` to keep the original file in addition to creating the zipped version. Don't worry if you forget the `-k` flag, you can simply unzip your file again, since no information is lost in the process.

The compression is most effective if the file is highly repetitive. You can test this out yourself by creating a file that simply states *"Zipping is great!"* 10000 times in a row.

```
$ for i in `seq 1 10000`; do echo "Zipping is great!" >> test.txt; done
$ ls -sh test.txt
176K test.txt
```

```
$ gzip test.txt
$ ls -sh test.txt.gz
4.0K test.txt.gz
```

As you can see, here the size of the file is reduced by more than 40 times!

9.2 Compressing multiple files

The `gzip` command that we learned in the previous section is only able to zip individual files. If you want to zip more than one file, you first have to put them in a so-called archive. The word archive is referring to the process of combining multiple files (and also directories if you want) into one single file. To archive files, you can use the command `tar -cf`. The option `-f` is to write the output to a **f**ile and not STDOUT (i.e. standard output), `-c` stands for **c**reate and `-z` **z**ips all files before putting them into the archive. So as you can see, you don't necessarily have to zip the files when you add them to an archive, but you have the option to do so if you want.

```
$ tar -cf all.tar BanthracisProteome.txt BanthracisProteome.txt.gz test.txt.gz
$ ls -sh all.tar BanthracisProteome.txt BanthracisProteome.txt.gz test.txt.gz
 27M all.tar
 24M BanthracisProteome.txt
3.2M BanthracisProteome.txt.gz
4.0K test.txt.gz
```

```
$ rm all.tar
$ tar -czf all.tar BanthracisProteome.txt BanthracisProteome.txt.gz test.txt.gz
$ ls -sh all.tar BanthracisProteome.txt BanthracisProteome.txt.gz test.txt.gz
6.4M all.tar
 24M BanthracisProteome.txt
3.2M BanthracisProteome.txt.gz
4.0K test.txt.gz
```

Again, you can see that the size of the archive is greatly reduced if you decide to add the `-z` flag that zips all the files that go into it.

Now you know how to create an archive, but what if you already have an archive and you want to extract the files in it? You can do this using the `tar -xf` command. The `-x` stands for extract and for the command to work, you also need to add the `-f`, otherwise no file will be written:

```
$ mkdir -p out
$ cd out
$ tar -xf ../all.tar
$ ls
BanthracisProteome.txt
BanthracisProteome.txt.gz
test.txt.gz
```

Remove the folder again:

```
$ rm -r out
```

9.3 Looking at zipped files

When you are dealing with a file that is zipped, usually you would unzip it first before performing any operation on it. Depending on the size of your file however, this can be a rather slow process and the bigger your file the slower it will be. Sometimes you just want to look at the content of your file without actually uncompressing it and bash offers the perfect command for this: `zcat`. It unzips the file line by line and prints it to standard output.

```
$ gzip -k BanthracisProteome.txt
$ zcat BanthracisProteome.txt.gz | head -n1
gzip: BanthracisProteome.txt.gz already exists; not overwritten
ID   3MGH_BACAN              Reviewed;         205 AA.
```

```
$ gunzip -c BanthracisProteome.txt.gz | head -n7 | wc -l
7
```

In the first example above, we first zipped the *BanthracisProteome.txt* file and kept the original file unchanged using the `-k` flag. Then we used `zcat` and `head` to look at the first line of the file while the actual file remained zipped. In the second example, you can see that we used the command `gunzip -c`. This is actually the same command as `zcat -` `zcat` is simply a short cut. You can use whichever command you prefer.

Some operations require to unzip the whole file regardless of which command you use. In the example below, we use `zcat` twice, once to look at the head of the file and then a second time to look at the tail. The command `time` prints out how long an operation took to execute.

```
$ time zcat BanthracisProteome.txt.gz | head > /dev/null

real    0m0.005s
user    0m0.005s
sys 0m0.002s
```

```
$ time zcat BanthracisProteome.txt.gz | tail > /dev/null

real    0m0.190s
user    0m0.169s
sys 0m0.053s
```

You can see that it takes longer to look at the tail than at the head, since `zcat` has to unzip the entire file line by line until it finally arrives at the bottom.

Note: `/dev/null` is a device file that immediately discards all data it receives, but it reports to you when the operation succeeds.

9.4 Zipping from STDOUT

You can combine the `cat`, `zcat` and `gzip` commands to optimize efficiency in your bash scripts. In the example below for instance, you can 'display' the content of a file using `cat` and then directly zip it using `gzip` and store the zipped file in 'test.gz':

```
$ cat test.txt | gzip > test.gz
```

Of course you can also go the other way around, start with a zipped file, and look at a specific line of that file by combining the `zcat` and the `head` command. Like this, you never actually have to unzip the file.

```
$ zcat test.gz | head -n1
Line 1
```

You can of course add as many intermediate steps as you wish:

```
$ zcat test.gz | sed 's/Zipping/Pipeing/' | gzip > new.gz
$ zcat new.gz | tail -n2
Pipeing is great!
Pipeing is great!
```

Note: We will learn more about `sed` later...

9.5 Exercises

9.5.1 Exercises: Zipping

See Section 14.0.8 for solutions.

1. Look at the file BanthracisProteome.txt using `more` and `less`
2. Create a new file 'short.txt' containing the first 200 lines of BanthracisProteome.txt. Then, zip the file
3. To the same thing as in exercise 2 without writing the file 'short.txt'
4. Extract the first 100 lines from the zipped file 'short.txt.gz' and store them in a new zipped file 'shorter.txt.gz', without ever unzipping 'short.txt.gz'
5. Create new files 'Line1.txt', 'Line2.txt', ..., each containing only one line of 'short.txt.gz' without unzipping the file (Hint: use a while loop)
6. Combine the 100 files 'Line1.txt', 'Line2.txt', ... to a zipped archive called 'all.tar'
7. Now zip all 'Line1.txt', 'Line2.txt', ... files individually and combine them to a new archive 'zipped.tar' without further zipping. Compare the sizes of the two archives.
8. Crate a directory 'all', copy the archive 'all.tar' there and extract it.
9. Clean up by deleting all Line* files, all archive files and the directory all with one command.

Chapter 10

Filtering and Editing

A filter is a program that reads standard input, performs an operation upon it and writes the results to standard output. It can therefore be used to process information in powerful ways such as restructuring output to generate reports, modifying text in files or other system administration tasks. The word filter here is defined in a broader context that what you might intuitively think, you actually came across a few such filters already, for example think of the commands `cat`, `head` or `wc`. In this chapter we will look at a few additional command line tools that can be used to filter and edit files and we will also have a short look at regular expressions. One particular command, `awk`, is quite powerful and it is actually its own language, we will therefore dedicate an entire chapter to it later in the tutorial.

10.1 `grep`

`grep` is an essential command in Bash, it is used to search text and strings in a given file. In other words, the grep command searches the given file for lines containing a match to the given pattern.

Random fact: `grep` is an acronym that stands for **G**lobal **R**egular **E**xpression **P**rint.

The syntax of `grep` is quite easy: you simply type the word `grep` followed by the pattern that you are looking for. If the pattern is present in the file that you specify, it will print out the entire line and highlight the pattern in red. If your pattern consists of only one expression (no spaces), you do not need to add double quotes behind it. However, you can also look for multiple expressions separated by spaces, then you will have to add the double quotes.

Let's look at an example: Using the `grep` command below, it will print out the lines of the specified file which contain the string 'Ba'. `grep` is case sensitive, so

it will not print out lines containing 'BA' or 'ba'.

```
$ head BanthracisProteome.txt | grep Ba
OS   Bacillus anthracis.
OC   Bacteria; Firmicutes; Bacilli; Bacillales; Bacillaceae; Bacillus;
```

If you are looking for the entire name 'Bacillus anthracis' you can use grep as well, but you need to add quotes:

```
$ head BanthracisProteome.txt | grep "Bacillus anthracis"
OS   Bacillus anthracis.
```

Often, we are not only interested whether a specific pattern is present in a file, but we want to know how many times (i.e. on how many lines in the file) the pattern occurs. We can use `grep` for this as well, since grep only prints out the lines that match the pattern, we can then simply count the number of lines:

```
$ grep ID BanthracisProteome.txt | wc -l
50532
```

Note: `-l` in the `wc` command specifies that we are only interested in the number of lines, without it, the `wc` command would print out the number of lines, the number of words and the number of bytes.

`grep` actually has an integrated line count option, it's the `-c` flag. So you can achieve the same result as in the example before using:

```
$ grep -c ID BanthracisProteome.txt
50532
```

There are many different options on how to use grep, a few are shown below. Remember you can always look at the manual by typing `man grep`.

If you want to keep the lines that do NOT match a pattern you can use `grep -v`.

```
$ wc -l BanthracisProteome.txt
515365 BanthracisProteome.txt
```

```
$ grep -cv ID BanthracisProteome.txt
464833
```

```
$ echo "515365 - 464833" | bc
50532
```

In the first command, we counted the total number of lines in the file, then in the second command we checked how many of these lines do not contain the pattern *ID*. Subtracting this number from the total number of lines obviously results in the number of lines that do contain the pattern *ID*.

If you are only interested in a specific number of lines, you can use `grep -m`. It will then only display the number of lines that you define:

```
$ grep -m3 Q[1-9] BanthracisProteome.txt
AC   Q81UJ9; Q6I2S8; Q6KWK9;
DR   ProteinModelPortal; Q81UJ9; -.
DR   IntAct; Q81UJ9; 2.
```

As mentioned before, `grep` is case-sensitive, however you can use the `-i` flag to make it case-insensitive

```
$ grep -m2 -i DAN BanthracisProteome.txt
RA   Escuyer V., Duflot E., Sezer O., Danchin A., Mock M.;
RA   Escuyer V., Duflot E., Mock M., Danchin A.;
```

10.2 tr

The command `tr` allows us to either *tr*anslate, delete or squeeze output character by character. There are many operations available when using the `tr` command such as searching and replacing text, transforming string from uppercase to lowercase (or vice versa), removing repeated characters from the string etc. Some examples are shown here:

Translating (=replacing) characters: `tr`

```
$ echo "tr is your    best friend..." | tr 'i' 'I'
tr Is your    best frIend...
```

Here we replaced all lower case i with upper case I.

```
$ echo "tr is your    best friend..." | tr 'ie' 'IE'
tr Is your    bEst frIEnd...
```

Here we replaced all lower case i or e with I and E.

Deleting characters: `tr -d`

```
$ echo "tr is your    best friend..." | tr -d ' '
trisyourbestfriend...
```

Here we got rid of all the white spaces.

```
$ echo "tr is your    best friend..." | tr -d 'ie'
tr s your    bst frnd...
```

Here we deleted every letter i and e.

You can also use `tr` to squeeze characters with the `-s` option. Squeezing means that you take a sequence of identical characters and you reduce their number to just 1.

```
$ echo "tr is your    best friend..." | tr -s '.'
tr is your    best friend.
```

```
$ echo "tr is your    best friend..." | tr -s '. '
tr is your best friend.
```

In the first example we squeezed the ... in the end into a single '.', and in the second example we added a space as well.

You can also use `tr` to convert lower case to upper case:

```
$ echo makeituppercase | tr [:lower:] [:upper:]
MAKEITUPPERCASE
```

10.3 `sed`

`sed` is a yet another powerful stream editor for filtering and transforming text. In contrast to `tr`, `sed` is applied line by line, so depending on the task that you perform it might make more sense to use one over the other.

The syntax of `sed` can feel weird at first, however it's such a powerful command that learning to use it will pay off very soon! `sed` is most often used to substitute (find and replace) characters or words in a line. The basic syntax for this is:

`sed 's/from/to/' inputFileName > outputFileName` or

`sed 's/from/to/g' inputFileName > outputFileName` or

`sed 'y/from/to/' inputFileName > outputFileName`

There are a couple of things going on here, first you notice that the entire command after `sed` is written in quotes `'`, this is part of the syntax and bash will through an error if you don't include them. Then, we either have the letter 's' or the letter 'y' in the beginning of the command. The 's' stands for **s**ubstitute and the 'y' stands for translating, sed will behave slightly differently depending on which one you use. So what is the difference? When you use the 's' option, the entire pattern that you provide will be replaced by your replacement string. Let's make an example:

```
$ for i in `seq 1 5`; do echo "hello there, line $i" >> sed.txt; done
$ sed 's/there/you/' sed.txt
hello you, line 1
hello you, line 2
hello you, line 3
hello you, line 4
hello you, line 5
```

If you do not provide an output file like in this example, it will just print into the terminal (standard output), which is perfect for demonstration purposes. You can see that it replaced every 'there' with 'you' as specified in the command. The 'y' option on the other hand translates individual characters (similar to `tr`):

```
$ sed 'y/eoi/XYZ/' sed.txt
hXllY thXrX, lZnX 1
hXllY thXrX, lZnX 2
hXllY thXrX, lZnX 3
hXllY thXrX, lZnX 4
hXllY thXrX, lZnX 5
```

Here, every 'e' has been translated (replaced) to an 'X', every 'o' to an 'Y' and every 'i' to a 'Z', so there is no need for 'eoi' to occur together as one string.

You probably noticed the 'g' at the end of one of the `sed` commands in the examples in the beginning of this section. The 'g' stands for global and you can chose to include it or not. Why would you want to include it? Since `sed` works line by line, it will read a line until it meets the pattern that you are looking for, replace that one occurrence and then move on to the next line. So, in case you want ALL occurrences replaced (not just the first one in the line), then you have to specify the global option.

Let's look at an example using the global option. Say we want to replace every small 'l' with a capital 'L' in our file:

```
$ sed 's/l/L/' sed.txt
heLlo there, line 1
heLlo there, line 2
heLlo there, line 3
heLlo there, line 4
heLlo there, line 5
```

As you see, if we do not provide the global option it will only replace the first instance of every line. Now, let's add the global option:

```
$ sed 's/l/L/g' sed.txt
heLLo there, Line 1
heLLo there, Line 2
heLLo there, Line 3
heLLo there, Line 4
heLLo there, Line 5
```

Perfect, now we successfully replaced all small l's with capital L's.

Note In this example, you could use the 'y' instead of the 's' option, then there would be no need to set the global option, since 'y' translates all characters that match the pattern.

Here are some more examples of `sed` using the file that you are already familiar with:

Translating characters: `sed 'y/from/to/'`

```
$ head -n1 BanthracisProteome.txt
ID   3MGH_BACAN              Reviewed;         205 AA.
```

```
$ head -n1 BanthracisProteome.txt | sed 'y/AI/ai/'
iD   3MGH_BaCaN              Reviewed;         205 aa.
```

```
$ head -n2 BanthracisProteome.txt | sed 'y/\n/ /'
ID   3MGH_BACAN              Reviewed;         205 AA.
AC   Q81UJ9; Q6I2S8; Q6KWK9;
```

In the example above you can see that `sed` can not replace newlines, since it is applied line by line.

Replacing text: `sed 's/from/to/`

```
$ head -n1 BanthracisProteome.txt | sed 's/BACAN/FRENCH FRIES/'
ID   3MGH_FRENCH FRIES              Reviewed;         205 AA.
```

```
$ head -n1 BanthracisProteome.txt | sed 's/A/a/'
ID   3MGH_BaCAN              Reviewed;         205 AA.
```

```
$ head -n1 BanthracisProteome.txt | sed 's/A/a/g'
ID   3MGH_BaCaN              Reviewed;         205 aa.
```

Note: Any character other than backslash or newline can be used instead of a slash to delimit the pattern and the replacement. Within the pattern and the replacement, the chosen delimiter itself can be used as a literal character, but you have to precede it with a backslash:

```
$ echo 'bununu' | sed 'saua\aag'
banana
```

In the example above, the letter 'a' was chosen as a delimiter, but at the same time we wanted to replace every 'u' with an 'a' as well, so we had to precede the replacement letter 'a' with a backslash, otherwise it wouldn't work.

`sed` also allows to apply commands to specific lines only:

```
$ head -2 BanthracisProteome.txt | sed '2 s/A/a/'
ID   3MGH_BACAN              Reviewed;         205 AA.
aC   Q81UJ9; Q6I2S8; Q6KWK9;
```

```
$ head -10 BanthracisProteome.txt | sed '2,10 s/A/a/'
ID   3MGH_BACAN              Reviewed;         205 AA.
aC   Q81UJ9; Q6I2S8; Q6KWK9;
DT   26-aPR-2004, integrated into UniProtKB/Swiss-Prot.
DT   01-JUN-2003, sequence version 1.
DT   13-NOV-2013, entry version 69.
DE   RecName: Full=Putative 3-methyladenine DNa glycosylase;
DE            EC=3.2.2.-;
```

```
GN   OrderedLocusNames=Ba_0869, GBAA_0869, BAS0826;
OS   Bacillus anthracis.
OC   Bacteria; Firmicutes; Bacilli; Bacillales; Bacillaceae; Bacillus;
```

Note: to indicate a range until the end of a file, use `$`: `sed '10,$'`. `sed` can do a lot more things such as deleting, replacing or adding lines. But all these tasks can also be achieved with `awk`, which has an easier syntax. A lot of the commands that you learned so far are able to execute similar if not the same tasks, it is up to you to find out which ones suit you best. Some commands will be more applicable or efficient for certain tasks, but ultimately there is no right or wrong.

10.4 Regular expressions

What is a regular expression?

Regular expressions (regex in short) are patterns that describe strings. They are super useful as they allow for complicated search patterns. Many tools and programming languages understand regex, including `grep` and `sed`. In fact, `grep` and `sed` interpret any pattern as regex.

Basic syntax

A regex consists of parts to be matched in a specific order. For example, the pattern `li` consists of two parts `l` and `i` that need to be matched in sequence.

A simple example:

```
$ echo 'I really like grep!' | grep 'li'
I really like grep!
```

Note: Even though it is not necessary for `grep`, it is good coding practice to encapsulate regex patterns in quotes.

Below you can find some of the most often used regular expressions as well as a few examples showing you how to use them.

The characters `^` and `$` indicate the start and end of a string:

```
$ echo 'blah blah blah' | grep 'blah'
$ echo 'blah blah blah' | grep '^blah'
$ echo 'blah blah blah' | grep 'blah$'
blah blah blah
blah blah blah
blah blah blah
```

When you execute these commands in the terminal you can see that with the first command, all three ‘blah’s are being highlighted, this is exactly what we expect from grep. In the second line, only the first one out of the three ’blah’s is being highlighted. This is because we added the `^` into the grep command, it

tells the program to only look for the pattern 'blah' at the beginning of a string. Similarly, when executing the third command, you will only receive the third of the 'blah's since the `$` sign tells the program to only look for the pattern 'blah' at the end of a string.

If you are looking for a pattern, but you are not quite sure exactly what it is like, or if there are multiple alternatives that differ maybe only by one character, then you can look for it either using `[ab]` or `(a|b)`. Both ways are valid, but the `(a|b)` syntax requires extended regex (option `-E`):

```
$ echo 'AUG CAG UAU UGG CAC AAC' | grep 'AU'
AUG CAG UAU UGG CAC AAC
```

```
$ echo 'AUG CAG UAU UGG CAC AAC' | grep 'A[UC]'
AUG CAG UAU UGG CAC AAC
```

Here, we are looking for a pattern that is either 'AU' or 'AC'

```
$ echo 'AUG CAG UAU UGG CAC AAC' | grep -E 'A(G|C)'
AUG CAG UAU UGG CAC AAC
```

Here, we are looking for a pattern that is either 'AG' or 'AC'

You can also provide a range of certain character class, and you can even combine multiple character classes in one statement: [A-C], [3-7], [f-m3-5].

```
$ echo 'abcdefghijklmnop012345678' | grep '[e-h]'
abcdefghijklmnop012345678
```

```
$ echo 'abcdefghijklmnop012345678' | grep '[l-w4-6]'
abcdefghijklmnop012345678
```

You can always use the square brackets to indicate ranges of characters that you would like to make, but for some frequently used classes, shortcuts have been established. For GNU tools, some helpful shortcuts include:

- `.` (the dot) matches any character - use carefully!
- `[[:alnum:]]` is equivalent to [a-zA-Z0-9]
- `[[:punct:]]` matches all punctuation characters
- `\w` matches all word characters, equivalent to [a-zA-Z0-9_]
- `\d` matches all digits, equivalent to [0-9]
- `\s` matches all white spaces (space, tab, new line, ...)
- `+` matches one or more occurrences
- `?` matches zero or one occurrence

It is often easier to define a class by exclusion using the negation operator `^`:

```
$ echo 'AUG CAG UAU UGG CAC AAC' | grep '[^A]'
AUG CAG UAU UGG CAC AAC
```

In case you are confused now, since just before we learned that `^` means it should match the character at the beginning of the string, and now we tell you it means

to match anything but the given character, don't worry, we can explain. The square bracket `[]` is used to match multiple characters and when you use the `^` inside of that square bracket, it means "match anything EXCEPT this", however if you use the `^` without square brackets like in the first example with the 'blah's, then it will look for matches at the beginning of the string. Both of these cases are quite useful, so it makes sense to know what they are about.

Special characters need to be escaped using the backslash:

```
$ echo 'file.txt file_txt.htm files.gz' | grep 'file.'
file.txt file_txt.htm files.gz
```

```
$ echo 'file.txt file_txt.htm files.gz' | grep 'file\.'
file.txt file_txt.htm files.gz
```

In the first example, all three files are highlighted, since the dot (.) stands for 'match any character'. If you actually want to match a dot, you will have to use a backslash.

You can use curly brackets to specify the number of occurrences that you are looking for in your pattern, they are called **quantifiers**. They are written after the character that you want to be repeated.

```
$ echo 'ATTACTTACCTTACCCTTACCCCTTACCCCCTT' | grep -E 'AC{2,3}T'
ATTACTTACCTTACCCTTACCCCTTACCCCCTT
```

```
$ echo 'ATTACTTACCTTACCCTTACCCCTTACCCCCTT' | grep -E 'AC{4}T'
ATTACTTACCTTACCCTTACCCCTTACCCCCTT
```

```
$ echo 'ATTACTTACCTTACCCTTACCCCTTACCCCCTT' | grep -E 'AC{3,}T'
ATTACTTACCTTACCCTTACCCCTTACCCCCTT
```

```
$ echo 'ATTACTTACCTTACCCTTACCCCTTACCCCCTT' | grep -E 'AC{,3}T'
ATTACTTACCTTACCCTTACCCCTTACCCCCTT
```

In the first example, we are looking for one A, followed by 2 OR 3 C's and then followed by one T. In the second example we only have one number in the curly bracket, this means you are looking for exactly this number of repetitions, in our case 4. `{min,}` and `{,max}` specify at least `min` and up to `max` occurrences, as illustrated in example 3 and 4.

In case you want to get fancy, there are again **shortcuts** for these expressions:

- `C*` is equivalent to C{0,}
- `C+` is equivalent to C{1,}
- `C?` is equivalent to C{0,1}

As mentioned before, the quantifiers are written after the target character. However, you can of course group multiple characters together, to do that you can simply use round brackets `()`.

```
$ echo 'AGTGTACCACAGTGTGTGTCACCAC' | grep -E '[AC](GT)+[AC]'
AGTGTACCACAGTGTGTGTCACCAC
```

In words, in the above example you are looking for either an `A` or a `C` (that's the `[AC]`), followed by one or more occurrences of `GT` (that's the `(GT)+`) and then again for either an `A` or a `C`, that's again the `[AC]`.

Well done, you finished this section! If you never heard of regular expressions before, that was probably quite challenging. Make sure to execute the code blocks in the tutorial, things will become more clear that way, and the exercises will hopefully help you implement some of the concepts you just learned about.

10.5 Exercises Filtering and Editing

10.5.1 Exercises: Filtering and Editing

See Section 14.0.9 for solutions.

1. Create a file DNA.txt, and add a random Sequence of at least 20 A, G, C and Ts using `vim`.
2. Print the content of DNA.txt to the screen, but replace all As with Gs on the fly.
3. Again replace all As with Gs on the fly, but add the output to DNA.txt as an extra line. Print the content of DNA.txt to the screen to check.
4. `grep`: extract all lines containg "ID" from BanthracisProteome.txt and look at those using `less`.
5. Extract the first 10 Lines containing "123" from BanthracisProteome.txt.
6. Extract the last 10 lines that do not contain "123" from BanthracisProteome.txt.
7. Count how many lines of the file BanthracisProteome.txt contain the pattern "out", but ignoring the first 100,000 lines.
8. Count how many lines of BanthracisProteome.txt contain the pattern "grep", regardless of the case (hence also allow for "Grep", "gREp", "GREP", ...).
9. Write a script "extract.sh" that extract the first and last 10 lines from a file (argument 1) that match a pattern (argument 2) and writes them to a new file (argument 3). Use this script to get the first and last 10 lines of BanthracisProteome.txt containing "ab" (case insensitive), followed by a number (e.g. ab7 or aB5) and store these lines in "abc.txt".
10. `tr` and `sed`: Extract the first 7 lines from BanthracisProteome.txt that contain the pattern "F*R", where * stands for any capital letter. Before

printing, replace all capital letters with minuscule ones. Find a solution with `tr` and one with `sed`.

11. Print the first 10 lines of BanthracisProteome.txt to screen, but squeeze all spaces (make sure that multiple spaces following each other are printed as a single one).

12. Extract all lines from BanthracisProteome.txt that contain 'Reviewed'^ and store them in a new file reviewed.txt. When doing so, delete all semicolons ';' from all lines and squeeze all spaces.

13. Replace all 'Reviewed' with 'tscheggt' inside the file 'reviewed.txt'.

14. Use sed to replace back 'tscheggt' with 'reviewed' inside 'reviewed.txt', but only on lines 5-9.

15. Regular Expressions: extract all lines from BanthracisProteome.txt that contain "SEQUENCE" followed by an arbitrary number of white spaces, a number, another white space, and "AA". And example Line would be "SQ SEQUENCE 205 AA; ... ".

16. Extract all lines from BanthracisProteome.txt with GO terms "GO:0009000" to "GO:0009999".

17. Extract all lines from BanthracisProteome.txt that contain a proper date of the form "01-JUN-2003".

18. Write a regex to extract all proper email address of the form "firstname.lastname@something.com". It should match "groovy.gorilla@jungle.com" or "hey.dude@cool.com" but not "secret@cia.com", "blah.blah.internet" or "no.domain@short".

Chapter 11

Sorting and Unique lines

In this section we are going to look at two essential commands of the Linux command line, `sort` and `uniq`. Both are quite simple to understand and they will make your coding life much easier further down the line. Let's start with `sort`, this command does exactly what you think it will do, it takes a text file as input and outputs it in a sorted way. The default is to sort alphabetically, not numerically. If you want to sort numbers however, you can simply add the flag `-n`. There are a few additional operations you can do such as randomizing file lines, removing duplicate lines, or checking whether a file is sorted.

```
$ for i in 7 5 10 1 5 100 5; do echo $i; done > numbers.txt
$ sort numbers.txt | tr '\n' ' '; echo
1 10 100 5 5 5 7
```

```
$ cat numbers.txt | sort -n | tr '\n' ' '; echo
1 5 5 5 7 10 100
```

In the first example the numbers were sorted alphabetically, so all the 1's come before the 5's etc. In the second example we used the `-n` to get a numerical sort. The `echo` at the end is just used to add a new line when printing the results to the screen, it is not absolutely needed.

If you have a file with a lot of duplicated lines and you want to know how many unique entries you have, you can use the command `uniq`. Again, the syntax is quite intuitive, but one thing you need to know is that only duplicated lines that are right next to each other will be detected. Sounds weird, but when you combine `uniq` with `sort`, then this will never be an issue.

```
$ uniq numbers.txt
7
5
10
1
```

```
5
100
5
```

```
$ sort numbers.txt | uniq
1
10
100
5
7
```

You can see, in the first example, no duplicates were removed, as soon as we sort it however it works!

You can use `uniq -d` to do the opposite and only keep duplicated lines.

```
$ sort numbers.txt | uniq -d
5
```

If you want to know how many times a specific line occurs in your file, you can use the `-c` flag.

```
$ sort numbers.txt | uniq -c
      1 1
      1 10
      1 100
      3 5
      1 7
```

You can also use `sort` and `uniq -d` to only keep entries that are present in two or more files:

```
$ for x in human chimp orangutan gorilla; do echo $x; done > primates.txt
$ head -2 primates.txt > primates.short
$ cat primates.txt primates.short | sort | uniq -d
chimp
human
```

The command `cat` simply pastes the contents of both files next to each other, so by then using the `-d` flag, you will receive the duplicated lines, i.e. the lines that are present in both files.

11.1 Matching

The last command that we are going to look at in this section is `join`, you can use it to match lines of two files based on a common field. For this to work, both files need to have identical join fields (columns). Per default, the join field is the first field delimited by a space.

```
$ echo -e "Mickey mammal\nDonald bird\nKarlo mammal" > systematics.txt
$ echo -e "Daisy female\nDonald male\nMickey male" > characters.txt
$ join characters.txt systematics.txt
join: systematics.txt:2: is not sorted: Donald bird
Mickey male mammal
```

This did not work, since the join fields (the name of the characters) are not identical. For the command to work, we have to sort the file first:

```
$ sort systematics.txt > systematics.sorted
$ join characters.txt systematics.sorted
Donald male bird
Mickey male mammal
```

If the join field is not the first field, you can specify which ones they are using `-1` and `-2`:

```
$ echo -e "fruit banana\nvegetable carrot\ngrain rice" > foodtypes.txt
$ echo -e "banana yellow\ncarrot orange\nrice white" > foodcolors.txt
$ join -1 2 -2 1 foodtypes.txt foodcolors.txt
banana fruit yellow
carrot vegetable orange
rice grain white
```

Here, we specified that for the first file (-1), the join column is the second one (2) and for the second file (2) the join column is the first one (1).

11.2 Exercises Sorting

11.2.1 Exercises: Sorting

See Section 14.0.10 for solutions.

1. The file "reviewed.txt" contains now the name and length of all proteins that have been manually reviewed. Use `cut` to extract all names, sort them and just look at the last 10.

2. Use this strategy to find the length of the longest and shortest of all reviewed proteins.

3. Check if there are any reviewed proteins of the exact same length. How many are they?

4. What is the most common length and how many reviewed proteins have this length?

5. Use `seq` to write the numbers from 1 to 100 to a file named 'numbers.txt'. Extract the third column of all lines in 'BanthracisProteome.txt' that

contain “KEGG” to a file names KEGG.txt. Paste these two files into a new file named ‘combined.txt’.

6. Get the content of combined.txt in shuffled order (use the tool shuff) and write the first 50 lines of it to a new file named ‘shuff.txt’. Join it with combined on the first column. Each line should then contain the same KEGG ID twice.

Chapter 12

awk

awk is a line-oriented language for text processing such as data extraction and reporting. It was written by Aho, Weinberger and Kernighan in 1977, but much improved since. Its syntax is quite different from the standard bash syntax, but once you get a hang on it you will see how powerful it can be for data manipulation.

12.1 Basic structure

awk commands are embedded in single quotes. They always consist of a *pattern* followed by an {action}, where the pattern specifies when the action should be performed.

In the following example, we are looking for the pattern "Dan" in the file BanthracisProteome.txt. By default each line containing the pattern is printed. Similar to the *grep* command in *bash*, the search is case sensitive (a search for "DAN" and "Dan" will give different results):

```
$ awk '/Dan/'  BanthracisProteome.txt
RA   Escuyer V., Duflot E., Sezer O., Danchin A., Mock M.;
RA   Escuyer V., Duflot E., Mock M., Danchin A.;
RA   Danchin A.;
```

Columns are addressed with the `$` sign. Where `$1`, `$2` and `$3` refer to the first, second and third column, respectively. Every string separated by any whitespace will be considered as a column. This has the advantage that you don't need to specify whether your field delimiter consists of a tab or multiple spaces. `$0` stands for the whole line. Hence, in the example below `{print $0}` is equivalent to the default, where no action is given:

```
$ awk '/Dan/ {print $0}' BanthracisProteome.txt
RA   Escuyer V., Duflot E., Sezer O., Danchin A., Mock M.;
RA   Escuyer V., Duflot E., Mock M., Danchin A.;
RA   Danchin A.;
```

```
$ awk '/Dan/ {print $1,$3}' BanthracisProteome.txt
RA V.,
RA V.,
RA A.;
```

awk understands the basic arithmetic operations: `+`,`-`, `*` and `/` (for addition, subtraction, multiplication and division) and `%` for modulo (also known as Euclidean division or division with remainder). To concatenate columns, you can simply add a space instead of a comma:

```
$ echo "1 2 3 4 5" | awk '{print $1, $2*$3, $3-$4, $5%$2, $1 $3}'
1 6 -1 1 13
```

As you can see, awk can not only read files, but you can also pipe the standard-output of your command-line to awk and you can pipe *awk*'s output to other bash commands:

```
$ echo "1 2 3 4 5" | awk '{print $1, $2*$3, $3-$4, $5%$2, $1 $3}' | sed 's/ / and /g'
1 and 6 and -1 and 1 and 13
```

awk allows to execute actions at the beginning and very end of a file / stream with the BEGIN and END blocks. As they are only executed once, they are good places to define variables (BEGIN) or to print a final calculation (END). Note that each print statement prints its own line, so for readability in this example we translate newlines to spaces in the end:

```
$ echo "sed awk" | awk 'BEGIN {print "The parrot"} {print} END {print "!"}' | tr '\n'
The parrot sed awk !
```

Note: `{print}` without specification prints the default, which is `$0` (the whole line).

You can also separate columns by other delimiters with the -F statement:

```
$ echo "I-don't-like-awk-exercises-even-though-they're-helpful." | awk -F '-' 'BEGIN {
But I like awk, though.
```

Or even multiple delimiters:

```
$ echo "Why-does_this-sentense-look_so_funny-?" | awk -F '[-_]' '{print $1, $3, $4, $7
Why this sentense funny ?
```

Also, you can sum over columns with the `+=` operator. In this example, we store the sum of all values in column 1 in the variable `sum` and print it in the end:

```
$ echo -e "1 2\n2 3\n5 6\n10 11"
$ echo " "
$ echo -e "1 2\n2 3\n5 6\n10 11" | awk '{sum+=$1}; END {print "sum of first column is " sum}'
1 2
2 3
5 6
10 11

sum of first column is 18
```

12.2 Variables in awk

You can define variables within the *awk* command. They can be dynamically defined and contain both numbers and strings. Note that the defined variable is only valid within that specific awk command and will be forgotten after the last single quote. Multiple actions can be performed in one command and are separated by a semicolon (;):

```
$ echo "5 7" | awk '{x=10; print $1*x; x=x+$2; print x}'
50
17
```

Variables can be increased by 1 for each entry via the `++` operator. This can easily be used to count lines by starting with `x=0` before the first line (using BEGIN), and increasing `x` with every line. In the following example, we count the lines that match the pattern "ID":

```
$ awk 'BEGIN {x=0} /ID/ {++x} END {print x, "Lines"}' BanthracisProteome.txt
50532 Lines
```

As mentioned above, the variables within *awk* are only valid within one command. Likewise, bash variables from outside of awk must first be introduced.

Attention: an undefined variable will not necessarily throw an error but have no value instead:

```
$ #trying to access an awk variable outside of awk
$ echo "5 7" | awk '{x=10; print $1*x; x=x+$2; print x}'
$ echo "this variable does not exist: $x"
50
17
this variable does not exist:
```

```
$ #trying to access a bash variable inside of awk
$ a=5
$ echo "5 7" | awk '{print $1+a, 10-a ; print a}'
5 10
```

Instead, bash variables should be passed to awk with the `-v` command before the first single quote:

```
$ a="awk"
$
$ echo "I like" | awk -v what="$a" '{print $0, what}'
I like awk
```

12.3 Functions

awk comes with a large array of built-in numeric functions, including `sqrt(x)` (square root of x), `log(x)`, `exp(x)` (exponential of x), `cos(x)`, `sin(x)` and `tan(x)`.

```
$ echo "10 100" | awk '{print log($1), sqrt($2)}'
2.30259 10
```

It also offers built-in functions for string manipulations, including `length()`, `substr()`, `sub()`, `tolower()` and `toupper()`:

length(): Printing the length of the line, including spaces.

```
$ echo "How long am I?" | awk '{print length($0)}'
14
```

substr(): the function substr(a,b,c) takes a string a, starts at position b inside that string, and returns everything inside c characters from there. If c is not given, the whole string is printed:

```
$ echo "first second third" | awk '{print substr($1,1,2)}'
$ echo "first second third" | awk '{print substr($1,1,3)}'
$ echo "first second third" | awk '{print substr($1,1,4)}'
$ echo "first second third" | awk '{print substr($1,1)}'
$ echo "first second third" | awk '{print substr($2,1)}'
$ echo "first second third" | awk '{print substr($2,3)}'
fi
fir
firs
first
second
cond
```

sub(): replace one string by another:

```
$ echo "This is awkward!" | awk '{sub("ward", "", $0); print "No,", $0}'
No, This is awk!
```

tolower() / toupper(): convert into lower or upper cases:

```
$ echo "This is awkward!" | awk '{sub("ward", "", $0); print "No,", tolower($1), $2, toupper($3)}
No, this is AWK!
```

Of course there are many more functions within *awk*. All available functions are explained in the (very long) man pages!

Random numbers with awk Random number generators calculate new random numbers based on the current one deterministically. To get different output, they need a different starting point, known as seed. By default, the `rand()` function will always start from the same seed within each started command - so if you repeat the following line multiple times, you will always start from the same point:

Note: There are different types of *awk*, namely `mawk`, `gawk` and base *awk*. `mawk` will always use a different seed, while `gwak` will always use the same seed. For *awk*, it depends on the specific version that you are using, try out the different commands in your command line and see how they behave.

```
$ echo -e "1\n2" | gawk '{print rand()}'
$ echo -e "1\n2" | gawk '{print rand()}'
0.924046
0.593909
0.924046
0.593909
```

If you want to be sure to use a different seed every time, you can simply use `srand()`, it will use time as a seed: `srand()` function. :

```
$ echo -e "1\n2" | awk 'BEGIN{srand()}{print rand()}'
$ echo -e "1\n2" | awk 'BEGIN{srand()}{print rand()}'
0.433822
0.0241092
0.0767769
0.504971
```

12.4 Extended matching

There are multiple ways of finding a pattern. Whether you want to extract lines where a substring matches anywhere in the line, or if you are looking for an exact match, these are the main basic options:

Return all lines that match the substring anywhere in the line:

```
$ awk '/ID/' BanthracisProteome.txt | wc -l
50532
```

Return the lines where the first column (or word) exactly matches the complete search pattern:

```
$ awk '$1=="ID"' BanthracisProteome.txt | wc -l
5493
```

Return the lines where the second column exactly matches the complete search pattern:

```
$ awk '$2=="ID"' BanthracisProteome.txt | wc -l
0
```

If you are looking for a pattern anywhere within a specific column, you can use the ~ sign. Be aware of these partial matches, as the search pattern "ID" can also be found within the string "NUCLEOT**ID**E", and the pattern "sample1" also matches the strings "**sample1**1", "**sample1**2", "**sample1**3", . . . :

```
$ awk '$2 ~ /ID/' BanthracisProteome.txt | wc -l
40676
```

To extract lines based on multiple seach patterns at once, you can use `&&` (and) and `||` (or).

This means, the following command returns all lines that contain either the exact pattern "ID" in the first column, **or** the string "ID" anywhere in the second column:

```
$ awk '$1 == "ID" || $2 ~ /ID/' BanthracisProteome.txt | wc -l
46163
```

While the next code returns all lines that contain the exact pattern "ID" in the first column, **and** contain the string "ID" anywhere in the second column:

```
$ awk '$1 == "ID" && $2 ~ /ID/' BanthracisProteome.txt | wc -l
6
```

You can also use regular expressions in your search patterns. Remember the tutorial about regular expressions (chapter 9.4) - the same results can be accomplished by matching for the regex pattern in awk:

```
$ echo -e 'TACACACTTTAGAGTTTACAGACTTT' | awk '$1 ~ /(A[CG])+/'
TACACACTTTAGAGTTTACAGACTTT
```

12.5 If-Else

To perform specific actions based on conditions, you can use if-else statements in the form of `if(<condition>){action}`, `if(<condition>){action} else {action}`, or if you want to add multiple else conditions you can extend the command to `if(<condition1>){action1} else if(<condition2>) {action2}`:

```
$ awk '{if ($1 == "ID") {++x} else {++y}} END {print x, "lines matched and", y, "lines
5493 lines matched and 509872 lines did not match out of 515365 lines
```

Note:: as we learned, undefined variables in *awk* do not have a value. Therefore we did not need to specify x and y in the beginning.

The if-command also makes it possible to store a variable until a condition is met. As en example imagine a file "zoo_inventory.txt" like that:

```
Species: Human
Daniel
Liam
Xenia
Species: Chimp
Betty
Tom
Species: Cat
Noobie
Nelson
```

So to print each individual with its corresponding species name, we can simply do:

```
$ awk '{if($1 ~ /Species/){sp=$2} else {print $1, "("sp")"}}' zoo_inventory.txt
Daniel (Human)
Liam (Human)
Xenia (Human)
Betty (Chimp)
Tom (Chimp)
Noobie (Cat)
Nelson (Cat)
```

12.6 Exercises awk

12.6.1 Exercises: awk

See Section 14.0.11 for solutions.

1. Extract all lines from BanthracisProteome.txt that contain "ID" in the first column and save them in a new file "prots.txt"
2. Use awk to calculate the percentage of them that are "Reviewed".
3. Write a BASH script that does the same thing using grep, wc and bc.
4. Use awk to get the total length of all proteins together in amino acids.
5. Use awk to print a file "len.txt" containing only the name of the protein and its length in amino acids (without the AA). Then, use awk to print a file "status.txt" containing only the name of the protein and its status (e.g. "Reviewed") for all proteins that contain only letters (no numbers) in their name. Finally, use join to create a file called len_status.txt, containing three columns: the protein name, its length and its status.

6. Use awk to create a new file "seq.txt" that contains one line per protein with three columns: 1) the name, 2) the length, 3) the amino acid sequence (as a single column without spaces). Hint: Remember, that awk reads your file line by line and use the opportunity to store values in a variable until a certain condition is met.

7. Make sure that the output of exercise 6 (seq.txt) is correct. To do so, test if the length of the protein sequence equals the number in column 2. If a line fails this sanity-check, print "ERR: lengths differ for <the failed line>". Use the END command to add a last line, either stating "file checked", or "ERR: something went wrong". (You can correct the length of failed proteins and repeat to see if your sanity-check works both-ways).

Chapter 13

Using R with BASH

Sometimes you will want to calculate something in R without actually entering the R environment manually. This is possible - and very handy - by passing STDIN (standard-input) to an R-script, and - if desired - getting the results past to STDOUT (standard-output).

But first you will have to specify the settings to be used. Some of them are compulsory and your attempt of using R in bash will fail if you don't specify them.

Now imagine the following R-script "example.r", which will print the correlation coefficient between x and y, as well as call the correlation coefficient between x^2 and y:

```
x <- 1:100
y <- x + rnorm(100)
print(cor(x,y))
cor(x^2, y)
```

To execute this script from the command-line, you run:

```
$ cat example.r | R --no-save

R version 4.1.1 (2021-08-10) -- "Kick Things"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> x <- 1:100
> y <- x + rnorm(100)
> print(cor(x,y))
[1] 0.9994745
> cor(x^2, y)
[1] 0.9685998
>
```

The `--no-save` or `--save` argument is compulsary and defines wether you want the work-space to be saved in the end or not (like the `q()` event when closing R). As you can see, this standard output is very bulky and large. It contains the copyright-information as well as all R-code. If you want to use only the output of the script, you will have to add `--silent` to your command:

```
$ cat example.r | R --silent --no-save
> x <- 1:100
> y <- x + rnorm(100)
> print(cor(x,y))
[1] 0.9993407
> cor(x^2, y)
[1] 0.9687322
>
```

This is already much better, but still contains the code. Instead of the `--no-save` argument you can also add `--vanilla` which adds some more options to make sure the execution is as clean as possible, and includes the `--no-save` option by default:

```
$ cat example.r | R --silent --vanilla
> x <- 1:100
> y <- x + rnorm(100)
> print(cor(x,y))
[1] 0.9994282
> cor(x^2, y)
[1] 0.9692636
>
```

Now to only get the bare output of your script, you can replace `--silent`and `--no-save` by the `--slave` command.

```
$ cat example.r | R --slave
[1] 0.9995182
[1] 0.9685607
```

Note: when calling R from the bash promt, it makes no difference if you actively *print* a code, or if you call it without the *print* command.

You can also directly pass the script to R via two ways:

```
$ R --slave < example.r
[1] 0.9993665
[1] 0.9671633
```

```
$ R --slave --file=example.r
[1] 0.9994708
[1] 0.9666766
```

The easiest way, though - and the most commonly used - is the `Rscript` command. `Rscript` is a shortcut for `R --slave --no-restore --file=`, where `--no-restore` implies `--no-save`:

```
$ Rscript example.r
[1] 0.9994355
[1] 0.9668596
```

Inside an R script, you have access to command-line arguments via the function `commandArgs`. You can store all arguments that come after your script in a list (here called "args"). Now you can access every element of that list in your downstream script:

```
#saving all arguments in the list "args"
args = commandArgs(trailingOnly=TRUE)
#accessing the first element as args[1], the second element as args[2], a.s.o.
x <- args[1]:args[2]
y <- x + rnorm(length(x))
paste('argument1:', args[1], '; argument2:', args[2])
print(cor(x,y))
cor(x^2, y)
```

```
$ Rscript example2.r 50 70
[1] "argument1: 50 ; argument2: 70"
[1] 0.9852944
[1] 0.9844357
```

You can also pass an R-command directly to R, and pass the output back to the shell:

```
$ echo "x <- 1:5 ; y <- x + rnorm(5); cat(y)" | R --slave | awk '{print $0}{print $1, $2+$3, $4*$
1.089707 1.71842 3.672731 3.232339 5.636937
1.089707 5.39115 18.2205
```

Note: `cat` in R removes the index [1] in the beginning.

All options can be found via `R --help`. Here once again the ones discussed in this tutorial:

–save or –no-save:

- compulsory
- specify weather or not the workspace has to be saved when quitting.

–silent:

- triggers R not to print the copyright info at the start

–slave:

- makes R run as quiatly as possible
- supresses the printing of commands
- implies *–no-save* and *–silent*

–vanilla:

- enables *–no-save* and other options that make sure the execution is as clean as possible.

13.1 Exercises R and bash

13.1.1 Exercises: R and bash

See Section 14.0.12 for solutions.

1. Use R to print 1000 normally distributed random numbers with specific mean μ=10 to STDOUT and use awk to calculate their mean.
2. Write a bash script that automatically downloads the current temperature at Swiss meteo stations and plots them as a function of the altitude at which they are measured. Use wget URL to download the data from here, then use cut to extract the relevant columns and R to plot to pdf. The script should further print the current temperature in Fribourg to the console and delete the downloaded file, as well as intermediate files.
3. Use awk and R to plot the protein length in amino acids (AA) against their molecular weight of all proteins in BanthracisProteome.txt (the info is given on lines starting with SQ).
4. Write a bash script that uses awk to extract the protein length (number of AA) of all proteins with a particular GO term (see lines starting with DR that have "GO;" in the second column) and R to plot it as a histogram to a pdf containing the GO Term as file name. Use a for loop to create such

a histogram for GO:0005886, GO:0005737, GO:0003677, GO:0005524 and GO:0016021.

Chapter 14

Solutions to Exercises

14.0.1 Files and directories

Corresponding exercises: see Section 3.4.1

1. ```
   mkdir bashExercises
   cd bashExercises
   ```

2. ```
   mkdir human chimp
   ```

3. ```
   cd human
   touch DNA.txt brain.txt food.txt
   ls
   ```

4. ```
   cp food.txt ../chimp
   ```

5. ```
   cd ../chimp
   ls
   ```

6. ```
   mv ../human/DNA.txt .
   ls ../human
   ls
   ```

7. ```
   ls -l
   ls -l ../human
   ```

8. ```
   rm ../human/brain.txt
   ```

9. ```
   rm -r ../human
   ```
   ```

14.0.2 Redirection

Corresponding exercises: see Section 4.4.1

1. ```
   echo "completely empty" > brain.txt
   ```
2. ```
   cat brain.txt
   ```
3. ```
   echo "...but still so smart" >> brain.txt
   ```

14.0.3 Variable declaration

Corresponding exercises: see Section 5.3.1

1. ```
   course="Introduction to UNIX"
   chapter="Chapter 4"
   echo "${course}: $chapter!"
   ```

14.0.4 Conditions and Loops

Corresponding exercises: see Section 6.4.1

1. ```
   for i in bananas mangos ants; do echo $i >> food.txt
   ```
2. ```
   for i in `seq 1 100`; do echo "banana" >> food.txt

   Can also be done using while:
   counter=0; while [ 100 -gt $counter ]; do echo "banana" >> food.txt; let counter=c
   ```
3. ```
   for i in ‘seq 1 100‘; do echo -n "ACGT"; done

   Can also be done using while:
   counter=0; while [ 100 -gt $counter ]; do echo "ACGT"; let counter=counter+1; done
   ```
4. ```
   num=1; while read line; do echo "Line $num = $line"; let num=num+1; done < food.tx
   ```

14.0.5 Permissions

Corresponding exercises: see Section 7.5.4

1. ```
   Read permissions: chimp
   Write permissions: None
   Execute permissions: None
   ```
   ```

2. ```
   Read permissions: chimp, gorilla, alien
   Write permissions: chimp, gorilla
   Execute permissions: chimp
   ```

3. ```
   Read permissions: chimp, gorilla, alien
   Write permissions: None
   Execute permissions: alien
   ```

14.0.6 Bash scripts

Corresponding exercises: see Section 7.6.1

1. ```
   echo "I and I alone!" > myPrecious.txt
   chmod -rw myPrecious.txt
   chmod u+rw myPrecious.txt
   ls -l
   ```

2. ```
   chmod -rw myPrecious.txt
   chmod g+rw myPrecious.txt
   ls -l
   cat myPrecious.txt
   # Note: you should get a "Permission denied" error, since you as a user do not have read per
   ```

3. ```
   rm myPrecious.txt
   # Since you do not have read and write rights but still are the owner of the file, you can d
   ```

4. ```
   mkdir "It's all yours!"
   touch It\'s\ all\ yours\!/yours.txt
   ls "It's all yours!"
   # Instead of using quotes, you can also escape the spaces and the exclamation mark. E.g.:
   ls It's\ all\ yours\!
   ```

5. ```
   chmod u-x "It's all yours!"
   cd "It's all yours!"
   # Note: you should get a "Permission denied" error.
   ls "It's all yours!"
   # Note: you should be able to see yours.txt, but also get an error that you can not access t
   ```

6. ```
   chmod u+x "It's all yours!"
   rm -r "It's all yours!"
   # Note: you need execution rights to recursively delete a directory! This is because you nee
   ```

7.
```
vim love.sh
#!/bin/bash
echo "I like Bash!"
# Leave vim by pressing Esc, then :wq
chmod u+x love.sh
for `seq 1 100`; do ./love.sh; done
```

14.0.7 Passing arguments to a script

Corresponding exercises: see Section 7.8.1

1.
```
vim dog.sh
#!/bin/bash
echo "Here comes $1, a $2 years old dog of $3 color."
# Leave vim by pressing Esc, then :wq
chmod u+x dog.sh
./dog.sh "Max" "15" "brown"
./dog.sh "Selma" "5" "blond"
```

2.
```
vim like.sh
#!/bin/bash
echo "I like $1!"
# Leave vim by pressing Esc, then :wq
chmod u+x like.sh
# you can call your script three times like this:
./like.sh "biology"
./like.sh "computer science"
./like.sh "bioinformatics"
# or alternatively, you write a for-loop:
for x in "biology" "computer science" "bioinformatics"; do ./like.sh "$x"; done
```

3.
```
vim reasons.sh
#!/bin/bash
echo $1 >> whyILikeBash.txt
# Leave vim by pressing Esc, then :wq
chmod u+x reasons.sh
# you can call your script three times like this:
./reasons.sh "powerful"
./reasons.sh "flexible"
./reasons.sh "fast"
# or alternatively, you write a for-loop:
for x in "powerful" "flexible" "flexible"; do ./reasons.sh "$x"; done
cat whyILikeBash.txt
```

4. ```
   vim append.sh
   #!/bin/bash
   echo $1 >> $2
   # Leave vim by pressing Esc, then :wq
   chmod u+x append.sh
   for n in Bern Fribourg Lausanne; do ./append.sh $n words.txt; done
   cat words.txt
   ```

5. ```
   vim append2.sh
   #!/bin/bash
   if [ ! -e $2 ]
   then echo "Created by append2.sh" > $2
   fi
   echo $1 >> $2
   # Leave vim by pressing Esc, then :wq
   chmod u+x append2.sh
   for n in hungry thirsty sleepy; do ./append2.sh $n moreWords.txt
   done
   cat moreWords.txt
   ```

6. ```
   vim positive.sh
   #!/bin/bash
   if [ $1 -gt 0 ]
   then echo $1 > "${1}.txt"
   fi
   # Leave vim by pressing Esc, then :wq
   chmod u+x positive.sh
   for n in "-10" "0" "10"; do ./positive.sh $n
   done
   ls
   ```

7. ```
   #!/bin/bash
   for name in `ls`
   do if [ -d $name ]
   then echo "$name is a directory"
   else echo "$name is a file"
   fi
   done
   # Leave vim by pressing Esc, then :wq
   chmod u+x explain.sh
   mkdir testdir1 testdir2
   touch testfile1 testfile2
   ./explain.sh
   ```

8.
```
vim helper.sh
#!/bin/bash
(
echo "#!/bin/bash"
echo "for i in \`seq 1 \$2\`"
echo "do echo \$1"
echo "done"
) > print.sh
chmod u+x print.sh
./print.sh $1 $2
# Leave vim by pressing Esc, then :wq

chmod u+x helper.sh
./helper.sh "Gotcha!" 10
```

14.0.8 Zipping

Corresponding exercises: see Section 9.5.1

1.
```
more BanthracisProteome.txt
less BanthracisProteome.txt
```

2.
```
head -n200 BanthracisProteome.txt > short.txt
gzip short.txt
```

3.
```
head -n200 BanthracisProteome.txt | gzip > short.txt.gz
```

4.
```
zcat short.txt.gz | head -n100 | gzip > shorter.txt.gz
```

5.
```
i=1
zcat shorter.txt.gz | head -n100 | while read line;
do echo $line > Line${i}.txt
let i=i+1
done
```

6.
```
tar -czf all.tar Line*.txt
```

7.
```
gzip Line*.txt
tar -cf zipped.tar Line*.txt.gz
ls -sh *.tar
```

8.
```
mkdir all
cd all
cp ../all.tar .
```

```
tar -xf all.tar
```

9. ```
   rm -r Line* *.tar all
   ```

14.0.9 Filtering and Editing

Corresponding exercises: see Section 10.5.1

1. ```
   # Press i to enter insert mode.
   # Then type AGCTTGCGACA...
   # Press Esc to enter command mode.
   # Type :w to save and :q to quit (shorter: type :wq).
   ```

2. ```
   cat DNA.txt | sed 's/A/G/g'
   ```

3. ```
   cat DNA.txt | sed 's/A/G/g' >> DNA.txt
   cat DNA.txt
   ```

4. ```
   grep ID BanthracisProteome.txt | less
   ```

5. ```
   grep -m 10 123 BanthracisProteome.txt
   ```

6. ```
   grep -v 123 BanthracisProteome.txt | tail
   ```

7. ```
   tail -n+100001 BanthracisProteome.txt | grep -c out
   ```

8. ```
   grep -ci grep BanthracisProteome.txt
   ```

9. ```
   #!/bin/bash
   grep -m10 -i $2 $1 > $3
   grep -i $2 $1 | tail >> $3
   chmod +x extract.sh
   ./extract.sh BanthracisProteome.txt ab[1-9] abc.txt
   # There is an option to combine head and tail in one grep call:
   grep -i $2 $1 | tee >(head > $3) | tail >> $3
   ```

10. ```
    grep -m7 F[A-Z]R BanthracisProteome.txt | tr 'A-Z' 'a-z'
    grep -m7 F[A-Z]R BanthracisProteome.txt | sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdef
    ghijklmnopqrstuvwxyz/'
    Of course there is a regular expression way with sed:
    sed -e 's/\(.*\)/\L\1/'
    ```
    ```

11. `head BanthracisProteome.txt | tr -s ' '`

12. `grep Reviewed BanthracisProteome.txt | tr -d ';' | tr -s ' '> reviewed.txt`

13. `sed -i 's/Reviewed/tscheggt/g'`

14. `sed -i '5,9 s/tschegt/reviewed/' reviewed.txt`

15. `grep -P "SEQUENCE\s+[0-9]+\s+AA" BanthracisProteome.txt`

16. `grep -E "GO:0009[0-9]{3}" BanthracisProteome.txt`

17. `grep -E "[0-9]{2}-[A-Z]{3}-[0-9]{4}" BanthracisProteome.txt`

18. ```
    # Write the email addresses in a file called emails.txt to see if your solution wo
    grep -E "[a-zA-Z]+\.[a-zA-Z]+@[a-zA-Z]+\.[a-zA-Z]{2,3}" emails.txt
    ```

14.0.10 Sorting

Corresponding exercises: see Section 11.2.1

1. `cut -f1 -d ' ' reviewed.txt | sort | tail`

2. ```
   cut -f4 -d ' ' reviewed.txt | sort -n | tail -n1
   cut -f4 -d ' ' reviewed.txt | sort -n | head -n1
   ```

3. `cut -f4 -d' ' reviewed.txt | sort -n | uniq -d | wc -l`

4. `cut -f4 -d' ' reviewed.txt | sort -n | uniq -c | sort | tail -n1`

5. ```
   seq 100 > numbers.txt
   grep 'KEGG' BanthracisProteome.txt | tr -s ' ' | cut -f3 -d' ' | head -n100 > KEGG
   paste numbers.txt KEGG.txt > combined.txt
   ```

6. ```
   shuff combined.txt | head -n50 > shuff.txt
   sort shuff.txt > shuff.sorted
   join shuff.sorted combined.txt
   ```

14.0.11 awk

Corresponding exercises: see Section 12.6.1

1. ```
   awk '$1 =="ID"' BanthracisProteome.txt > prots.txt
   ```

2. ```
   awk '{++tot; if($3=="Reviewed;") {++x}} END {print 100*(x/tot), "%"}' prots.txt
   ```

3. ```
   #!/bin/bash
   x=$(grep -c "Reviewed" prots.txt)
   tot=$(wc -l prots.txt | cut -f1 -d' ')
   percent=$(echo "scale=5; 100 * $x / $tot" | bc)
   echo "${percent}%"

   # solution: 13.36246%
   ```

4. ```
   awk '$1=="ID" {tot = tot+$4} END {print tot}' BanthracisProteome.txt

   # solution: 1439306
   ```

5. ```
   awk '$1=="ID" {print $2, $4}' BanthracisProteome.txt | sort > len.txt
   awk '$1=="ID" && $2!~/[0-9]/ {print $2, $2 }' BanthracisProteome.txt | sort > status.txt
   join len.txt status.txt > len_status.txt
   ```

6. ```
   # if 1st column is ID, store name \& len and empty the seq variable. If 1st column is SQ, se
   # The order of these commands is crucial. e.g. if the last two conditions are switched, you
   awk 'BEGIN {addseq=0}; {if ($1 == "ID") {name= $2; len=$4; seq="";} else {if ($1 == "SQ") {a
   ```

7. ```
   awk 'BEGIN {test=0}; {if (length($3) != $2) {print "ERR: lengths differ for " $0; test=1;}};
   ```

14.0.12 R and bash

Corresponding exercises: see Section 13.1.1

1. ```
   mu=10; echo "cat(rnorm(1000, $mu))" | R --slave | cut -d" " -f3- | tr " " "\n" | awk '{s+=$0
   ```

2. ```
   #!bin/bash
   wget https://data.geo.admin.ch/ch.meteoschweiz.messwerte-lufttemperatur-10min/ch.meteoschwei
   cut -d";" -f4,6 ch.meteoschweiz.messwerte-lufttemperatur-10min en.csv | tail -n+2 > cols.csv
   echo 'file <- read.csv("cols.csv", sep = ";"); pdf("altVsTemp.pdf"); plot(file[,2], file[,1]
   xlab = "altitude", ylab = "temperature", main = "altitude vs temperature"); dev.off()' | R
   --slave
   cat ch.meteoschweiz.messwerte-lufttemperatur-10min en.csv | grep Fribourg | cut -d";" -f4
   rm ch.meteoschweiz.messwerte-lufttemperatur-10min en.csv
   rm cols.csv
   ```
   ```

3. ```
   awk '$1 == "SQ" {print $3, $5}' BanthracisProteome.txt > lenWeight.txt
   echo 'lenWeight <- read.table("lenWeight.txt"); pdf("lenVsWeight.pdf"); plot(lenWe
   lenWeight[,2]); dev.off()' | R --slave
   rm lenWeight.txt
   ```

4. ```
   #!/bin/bash
   for go in GO:0005886 GO:0005737 GO:0003677 GO:0005524 GO:0016021; do
   #if $1=ID, save length and set "found" to 0;
   #if $1=DR and $2=GO (go-term found) set "found" to 1;
   #if the end of the protein is reached ($1=//), and a GO-term was found (found=1),
   awk -vGO=${go} '$1=="ID"{len=$4; found=0}; $1=="DR" && $2~"GO" {found=1}; $1=="//'
   echo "lengths <- read.table('${go}_length.txt'); pdf('${go}_length.pdf'); hist(len
   done
   ```

https://ryanstutorials.net/bash-scripting-tutorial/ https://linuxhandbook.com/
https://blog.hubspot.com/website/