TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 22/23

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)
http://db.in.tum.de/teaching/ws2223/foundationsde

Sheet Nr. 01

Exercise 1 Which tools are there in GNU coreutils and for what are they useful? Describe each tool in two sentences.

Coreutils have lots of configuration options, unfortunately often not with coherent command line flags. To learn more about utilities (and other facilities in Linux), use the man page as first source of help. For example type `man join` in the `command line` to learn more about how the join program works.

Solution:

- comm: compares two sorted files line by line
- join: join lines of two files on a common field
- sort: sort lines of text files by a key/column
- head / tail: output the first/last part of files
- uniq: report or ommit (subsequent!) repeated lines
- wc: print newline, word, and byte counts for each file
- seq: print a sequence of numbers

Not in core utils but useful:

- grep: print lines matching a pattern
- awk: pattern scanning and processing language
- sed: stream editor for filtering and transforming text

Exercise 2 Which means does bash offer to combine the coreutils (or any programs for that matter)?

Hint: use `man bash` and read DEFINITIONS, RESERVED WORDS and SHELL GRAMMAR. Also read the beginning of `man 7 pipe` for information on how pipes work in Linux.

A useful bash construct besides redirections (`<`,`>`,`2>1` etc.) and pipes (`|`) is process substitution (`<()`). Please read the Wikipedia article if you are not familiar with this.

Solution:
Pipelines The most important mean to combine programs with bash are pipelines. A pipeline is a sequence of one or more commands separated by one of the control operators | or |&. The format for a pipeline is:

```
[time [-p]] [ ! ] command [ [| | |&] command2 ... ]
```

Lists Pipelines can be combined using lists. A list is a sequence of one or more pipelines separated by one of the operators ;, &, &&, or ||, and optionally terminated by one of ;, &, or <newline>. In this example

```
command1 && command2
```

the operator && connects the two commands such that `command2` is only executed if `command1` returns an exit status of zero.
Control Constructs Besides pipelines and lists, bash offers other constructs like `if`, `for`, `while` etc.

Exercise 3 For which kinds of data sets are the gnu tools suitable?

Solution: The gnu utils work on line based text data sets. Many tools can handle entry delimiters, which enables them to understand the concept of records/tuples.
In contrast, the utils are not able to understand nested data structures like XML or JSON.

Exercise 4 What do we need a pager for even though `cat` can print to the terminal? Name an available pager on Ubuntu.

Solution: A pager does not need to consume the whole input. It is therefore able to manage large files quickly. Furthermore, features as searching are available. Pagers on Ubuntu are for example `less`, `more`, `head` and `tail`.

Exercise 5 Write a *regular expression* that matches all e-mail addresses in the file `mails_match.txt` and that does not match any of the lines in `mail_dont_match.txt`.

In case you are not familiar with regular expressions or want to refresh your skills, you can use for example the *quick start* or *tutorial* of this website.

For this task you can easily test your regular expressions using online tools like regex101.com. Copy the two `.txt` files to the online tool and try developing your regular expression.

Regular expressions can also be easily used in the command line like this:

```
egrep <your regular expression here> mails_match.txt
```

In a bash in Ubuntu, `egrep` will highlight the matched part per default. This may be helpful for debugging. Have a look at `man egrep` to learn about the usage of the command.

To determine whether all the mail addresses matched, you may want to check if the result of your regular expression application has the same number of lines as the original file. That means the following commands must return the same number:

```
wc -l mails_match.txt
egrep <your regular expression here> mails_match.txt | wc -l
```

Furthermore, the following command should produce no output at all:

```
egrep <your regular expression here> mails_dont_match.txt
```

Solution:

```
egrep ’^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$’ mails_match.txt
```

Beware: This regular expression by no means matches all possible e-mail addresses defined by RFC5332. It is only one possible solution for this exercise. If you care about the pros and cons of validation e-mail addresses using regular expressions, you might find this discussion interesting.

Exercise 6

It is often useful to divide regular expressions into *groups*. For example when searching for emails, name and domain are divided by the @ symbol. Groups provide the means within regular expression matching to match name and domain separately.

There are two different types of groups, *capturing* and *non-capturing* groups.

Capturing groups are used to store the matched string for later use like replacing. Non-capturing groups help to structure your regular expression.

Description with examples on this website.

Use *non-capturing groups* to find a short expression to match MAC addresses in the file `random-mac.txt`. Note that `grep` needs the flag `-E` to support groups.

Use *capturing groups* and replace to change the structure of the file `names.txt` from ”Firstname Lastname” to ”Lastname, Firstname”. You may notice that GNU utils are not the right tool for this task. Consider using a scripting language, e.g.:

```
perl -n -e’/.*/ && print $1’
```

You can learn about using regex in `perl` for example on this website.

Solution: A short solution to capture all mac addresses

```
(?:[0-9A-F]{2}:){5}[0-9A-F]{2}
```

A short solution to change the order of names:

```
^([A-z]+) ([A-z]+)$
```

Using it with perl:

```
cat random-names.txt |
perl -n -e’/^([A-z]+) ([A-z]+)$/ && print $2, ", ",$1, "\n"’
```

Exercise 7 Convert RGBA color codes from the hex form ”#FF0000FF” (red) to the rgba form ”rgb(255, 0, 0, 1.0)”.

In the rgba form the first three digits indicate how much of the additive light basic colors have to be added to generate the color. The fourth entry determines the alpha value, also known as opacity which ranges from 0 to 1.

In the hex form two hex digits represent the values of red, green, blue and alpha, all with a range from 0 to 255.

Use a scripting language, e.g. `perl`, to convert RBGA color codes from hex form to rbga form.

Solution:

```
echo "#FF0000F0" |
perl -n -e
'/#([0-9A-F]{2})([0-9A-F]{2})([0-9A-F]{2})([0-9A-F]{2})/
&& print "rgba(", hex($1), ",", hex($2),",", hex($3),",", hex($4)/255, ")"'
```