"Manipulation of Text Data Using Linux Tools"

Workshöppchen

Dr. Markus Kiefer

COS Heidelberg – Dpt. Biodiversity and Plant Systematics

26. September 2018

Centre for Organismal Studies Heidelberg

Overview

Basics

- Text. . .
- Shell work
- Quotation marks
- Variables
- Common commands
- Common tasks

What is text?

What is text?

- Everything that consists of readable characters that can be easily viewed by just displaying it in a terminal or printing with a typewriting device.
- Just characters, no formatting.
- No “binary” meaning, just plain old human-readable information.

ASCII

Old, relyable, simple but restricted to 127 chars.

```
    30 40 50 60 70 80 90 100 110 120
   ---------------------------------
  0:    (  2  <  F  P  Z  d   n   x
  1:    )  3  =  G  Q  [  e   o   y
  2:    *  4  >  H  R  \  f   p   z
  3: !  +  5  ?  I  S  ]  g   q   {
  4: "  ,  6  @  J  T  ^  h   r   |
  5: #  -  7  A  K  U  _  i   s   }
  6: $  .  8  B  L  V  `  j   t   ~
  7: %  /  9  C  M  W  a  k   u  DEL
  8: &  0  :  D  N  X  b  l   v
  9: '  1  ;  E  O  Y  c  m   w
```

Unicode and stuff

Unicode

A rather complicated encoding scheme using a variable number of bytes to encode text characters. Not restricted to a maximum amount of allowed chars. Can work with any language and alphabet.

UTF-8

Most popular Unicode-flavour. Nowadays, most terminals are set to UTF-8 encoding, so at least Umlauts and several other non-Ascii chars will “just work”. Theoretically restricted to approx. 2 billion possible chars, practically much less complete but very usable.

Text Examples

- Most sequence data: .fasta, .gb, .sam...
- Comma (or whatever)-separated tables: .csv, .txt
- Program source code and scripts: .sh, .py, .pl, .c...
- Structured data and markup documents: .xml, .html...

What is NOT text?

- (MS-)Office documents: .docx, .xlsx, .pptx. . .
- Executable programs.
- Packed data: .zip, .gz, .rar, .bam, .bcf. . .
- Multimedia: Graphics, videos, sound. . .
- Several other application specific data, including some of the more integrated bioinformatics tools (CLC, Geneious)

What's a shell anyway?

UI
A “user interface”. Anything that allows a user to conveniently interact with an operating system.

Bash
The “Bourne again shell”. Around since several decades, the most popular text-based shell, a derivative of the 1970s “Bourne shell”. Allows efficient file and folder manipulation, text data manipulation and limited programming.

How to connect to one?

Local

Open any terminal application. On Linux you'll probably see a Bash, on Windows you'll see DOS or Powershell (if you're lucky).

Remote

Use the SSH protocol to connect to a suitable server. Most popular SSH implementation on Windows is "putty".

How to work with a shell?

Commands

consist of a command “name”, followed by options and arguments:

```
    cp -r something somewhere
```

Helpers

Remember to use the history (up/down-keys) and text auto-completion (the Tab-key). Of course you can use your system’s clipboard to copy/paste text snippets.
Just in case your terminal seems stuck: Strg-c (Cancel) will unfreeze it in most cases. (Sometimes Strg-q is necessary.)

Quotes

Single quotes

Anything between singe quotes is handled “as is”, a text string without interpretation.

Double quotes

A text string, that belongs together (spaces included) but will be interpreted. E.g. Variables will be replaced by their content. Characters with special meaning have to be “escaped”.

Backticks

Anything in backticks will be interpreted as shell code and replaced by the result of that.

Reusing your code

Oneliners

You can type your commands – even complex ones – directly on the command line and execute them right away. If something goes wrong, use the cursor-up key and correct the errors.

Scripting

You can save your commands – especially complex ones – in a textfile and reuse them like small programs.

Named boxes

Variables

Variables are like named boxes of data. You can store something with a name tag of your choice and use it later.

```
a=something
echo $a
something
_
```

Named boxes

In Bash, you can just assign something to a variable with the assignment operator “=”. It will be created and filled if it doesn’t exist, yet.

Named boxes

In Bash, you can just assign something to a variable with the assignment operator “=”. It will be created and filled if it doesn’t exist, yet.
If you put a Dollar sign in front of the variable’s name, it will be “expanded” i. e. replaced by its content.

Caveats

- Bash is case sensitive!
  "raxml" is not the same as "RAxML".
- No questions asked!
  Bash assumes you know what you want. It will mercilessly overwrite any file you target a write operation on.
- Spaces are interpreted as delimiters!
  Be careful when using spaces: `rm a*` will delete everything beginning with an "a", while `rm a *` will delete "a" and **anything**! Use double quotes around variables that just *might* be expanded to something containing spaces, to avoid nasty surprises.

File commands

cp Copy something somewhere
Options: -r recursively including subdirectories

mv Move / rename something somewhere

rm Remove(=delete) file
Options: -r recursively (Careful here!)

touch Create empty file or update timestamp

cat Output file's content

head/tail Output first/last few lines
Options: -n Number of lines [10]

Folder commands

ls List content of folder
Options: -l long, detailed output; -a include hidden; -h human readable

mkdir Create folder
Options: -p OK if name already exists

rmdir Remove *empty* folder

cd Change into folder

pwd Show full path to current folder

Other commands

echo	Print out something to the terminal
clear	Empty visible part of terminal
history	What have I typed so far?
df	Show free space on disk(s) Options: -h human readable (kB, MB. . .)
chmod	Change access mode of file Arguments: e.g. a+w allow write access for all
man	Show manual pages for a command (exit by pressing“q”)
less	Conveniently display test files (“q” to exit)

File name “wildcards” et al.

When referring to file- and folder names, wildcards allow to address many files simultaneously. Be careful when using together with possibly deleterious commands like rm, cp. . .

`*` any number of arbitrary characters (including none at all)

`[a-c1234]` One of a given set of allowed characters, e.g. a, b, c, 1, 2, 3, 4

`?` Exactly one character (no matter which one)

`~` Shortcut for the user’s home folder

`.` “this folder”

`..` the “parent folder”

Moving

Move all FASTA files into new directory:

Moving

Move all FASTA files into new directory:

```
mkdir fastas
mv *.fa fastas/
```

Rename a file or folder:

Moving

Move all FASTA files into new directory:

```
mkdir fastas
mv *.fa fastas/
```

Rename a file or folder:

```
mv oldname newname$ mv *.fa fastas/
```

Move and rename a folder somewhere else:

Moving

Move all FASTA files into new directory:

```
mkdir fastas
mv *.fa fastas/
```

Rename a file or folder:

```
mv oldname newname$ mv *.fa fastas/
```

Move and rename a folder somewhere else:

```
mv folder1 /home/mkiefer/some/where/folder2
```

Copying

Copy all FASTA files into new directory:

Copying

Copy all FASTA files into new directory:

```
mkdir fastas
cp *.fa fastas/
```

Copying

Copy all FASTA files into new directory:

```
mkdir fastas
cp *.fa fastas/
```

Copy all SAM files beginning with “ax” or “ay” from somewhere else to here:

```
cp ../some/where/a[xy]*.sam .
```

Copying

Copy all FASTA files into new directory:

```
mkdir fastas
cp *.fa fastas/
```

Copy all SAM files beginning with “ax” or “ay” from somewhere else to here:

```
cp ../some/where/a[xy]*.sam .
```

Copy a folder tree somewhere else and go there:

```
mkdir -p any/where/other folder
cp -r any/ folder
cd folder/any/where/other
```

Other stuff

Get help about copy:

```
man cp
```

Other stuff

Get help about copy:

```
man cp
```

Get detailed listing of root folder:

```
ls -l /
```

Other stuff

Get help about copy:

```
man cp
```

Get detailed listing of root folder:

```
ls -l /
```

Where are we?

```
pwd
```

Overview

Datastreams

- Channels
- Redirection
- Pipes
- Common redirection tasks

Channels

Input and output channels are a common concept in Linux. Think about them as entrance and exit of programs. Most programs take in raw data on one side and spit out result data on the other.

Channels

Input and output channels are a common concept in Linux. Think about them as entrance and exit of programs. Most programs take in raw data on one side and spit out result data on the other.
Usually the “entrance” of a program is connected to your keyboard and the “exit” to your terminal.

std-Channels

stdout The standard channel for output (1), per default connected to your terminal

stderr The standard channel for error messages (2), per default also connected to the terminal

stdin The standard input channel, per default connected to your keyboard

Redirecting output

You can redirect any program’s output into a file, separately for normal output and error output:

```
ls -l > folderlisting 2> listerrors
```

This will create a file called “folderlisting” and fill it with the output of the ls command. Any possible errors will end up in “listerrors”.
“»” will redirect output and *append* it to a file, thus not overwriting it.

Redirecting input

You can also redirect any file’s content to the input channel of any program that is capable of accepting text input:

```
sort < some.txt
```

This will redirect the content of “some.txt” to the program “sort”, which will –unsurprisingly– print it out in alphabetical order.

Centre for Organismal Studies Heidelberg

Caveat!

Never try to simultaneously read from and write to a given file. You will end up with it being emptied!

The plumbing

In contrast to the redirection operators (>. . .), the “pipe” (|) is used to connect one program’s output to another one’s input channel.

```
ls -l | sort -r
```

This e.g. will give you a directory listing in reverse order.

Pipelines

Using redirects and pipes you can construct complex sequences of commands reading from different files and producing output in others.
Part of the GNU/Linux philosophy is to have small efficient programs dedicated to one given task each and to solve complex tasks by plugging together the right tools.

Redirection tasks

Create a single sequence file from separate sequences:

```
cat *.fasta > multi.fasta
```

Redirection tasks

Create a single sequence file from separate sequences:

```
cat *.fasta > multi.fasta
```

Sort file and remove duplicates:

```
cat file.txt | sort | uniq > sortedfile.txt
```

Redirection tasks

Create a single sequence file from separate sequences:

```
cat *.fasta > multi.fasta
```

Sort file and remove duplicates:

```
cat file.txt | sort | uniq > sortedfile.txt
```

Write something into a file:

```
echo 'Anfang' > newfile,txt
cat >> newfile # end input with Strg-D
```

Redirection tasks

Append lines 99 and 100 of file 1 to file2:

```
head -100 file1 | tail -2 >> file2
```

Redirection tasks

Append lines 99 and 100 of file 1 to file2:

```
head -100 file1 | tail -2 >> file2
```

Copy without using cp:

```
cat file1 > file2
```

Overview

Filtering

- Filtering lines
- Regular expressions
- Common filtering tasks

grep

grep is a versatile program designed to filter lines from textfiles.

Basic usage: "`grep pattern file`" will output every line from "file" with an occurrence of "pattern".

Patterns can be just keywords, parts of keywords or complex expressions.

grep's Options

-v Reverse search, everything that doesn't include...

-c Don't show hits, just count.

-B, -A Add *n* lines **b**efore or **a**fter the hit.

-f Read pattern(s) from file.

-i Case-insensitive search.

-o Show only matching parts of line.

What are regular expressions?

Regular expressions are rulesets that describe variable text patterns. They can be used to filter or automatically edit text files.
In the easiest case the “RegEx” could actually be just a keyword. In more complex cases it could match hundreds of different expressions.

What are regular expressions?

Regular expressions are rulesets that describe variable text patterns. They can be used to filter or automatically edit text files.
In the easiest case the “RegEx” could actually be just a keyword. In more complex cases it could match hundreds of different expressions.
The RegEx language is rather complicated and there are several dialects. But they have a couple of basic things in common.

The RegEx language

/RegEx/ Slashes traditionally enclose RegExs. Can be a pair of other characters if more convenient.

The RegEx language

/RegEx/	Slashes traditionally enclose RegExs. Can be a pair of other characters if more convenient.
.	One arbitrary character.

The RegEx language

/RegEx/ Slashes traditionally enclose RegExs. Can be a pair of other characters if more convenient.

. One arbitrary character.

[ ] One of a set of characters that can be described as an unseparated list of included chars and/or as char ranges like “a-z” or “0-9”. A literal dash would have to be the last char in the set.

The RegEx language

/RegEx/ Slashes traditionally enclose RegExs. Can be a pair of other characters if more convenient.

. One arbitrary character.

[ ] One of a set of characters that can be described as an unseparated list of included chars and/or as char ranges like “a-z” or “0-9”. A literal dash would have to be the last char in the set.

[^] One char not included in a set. ([^a-z] = not a lower case letter)

RegEx quantifiers

* Any number of the preceeding definition or none.

RegEx quantifiers

* Any number of the preceding definition or none.

{n,m} n to m of the preceding definition.

RegEx quantifiers

*	Any number of the preceeding definition or none.
{n,m}	n to m of the preceeding definition.
{n,}	at least n of the preceeding definition.

RegEx quantifiers

*	Any number of the preceeding definition or none.
{n,m}	n to m of the preceeding definition.
{n,}	at least n of the preceeding definition.
{,m}	m of the preceeding definition at maximum.

RegEx quantifiers

* Any number of the preceeding definition or none.

{n,m} n to m of the preceeding definition.

{n,} at least n of the preceeding definition.

{,m} m of the preceeding definition at maximum.

{n} exactly n of the preceeding definition.

Other RegEx components

$ The end of the line.

Other RegEx components

- $ The end of the line.
- ^ The beginning of the line.

Other RegEx components

$	The end of the line.
^	The beginning of the line.
(...)	Store a matched partial string for later use.

Other RegEx components

$	The end of the line.
^	The beginning of the line.
(...)	Store a matched partial string for later use.
|	"Or" operator to link two expressions.

Centre for Organismal Studies Heidelberg

Filtering tasks

Get all description lines from a FASTA file and sort them:

```
grep "^>" multi.fa | sort
```

Filtering tasks

Get all description lines from a FASTA file and sort them:

```
grep "^>" multi.fa | sort
```

Find all accession numbers for sequences from *A. thaliana* chromosomes 1 and 2 in an EMBL file:

```
grep -i -B 6 "^DE.*AT[12]G" multi.fa | grep "^AC "
```

Filter out “Notice” and “Warning” lines from the end of a log file:

```
tail -100 access.log | grep -i "notice\|warning" access.log
```

Mind the “escaped” “or” operator.

Filtering tasks

Find all occurences of at1g. . . in a FASTA file and get the ones that also appear in another file.

```
grep -i "at1g" multi1.fa | sort | uniq > filter.txt
grep -i -f filter.txt multi2.fa > both.fa
```

Filtering tasks

Find all occurences of at1g...in a FASTA file and get the ones that also appear in another file.

```
grep -i "at1g" multi1.fa | sort | uniq > filter.txt
grep -i -f filter.txt multi2.fa > both.fa
```

Filter all reverse reads from a combined FASTQ file:

```
cat comb.txt.gz | gunzip | grep -A 3 "-2 *$" | gzip > rv.tgz
```

Overview

Counting

- Counting tools
- Common counting tasks

Counting things

There are several tools to count things:

grep With the “-c” option grep will only count occurences.

wc The general purpose counter.
Options: -l for lines, -m for chars, -w for words. . .

uniq With the option “-c” uniq will report how many identical occurences of each unique line there were.

nl Will add line numbers.

Counting things

Lines

To count lines “grep -c” or piping grep’s output through “wc -l” will work. “nl” can help in a very descriptive way and “uniq -c” can be a useful special case.

Counting things

Lines

To count lines “`grep -c`” or piping grep’s output through “`wc -l`” will work. “`nl`” can help in a very descriptive way and “`uniq -c`” can be a useful special case.

Strings

Taking more than one occurrence per line into account is a little tricky sometimes. It can be done with “`grep -c -o`”.

test

Count the reads in a FASTQ file:

```
grep -c "^@" data.1.fastq
```

test

Count the reads in a FASTQ file:

```
grep -c "^@" data.1.fastq
```

Count how many “A” and “T” are in a sequence:

```
grep -v "^>" gen.fasta | grep -cio "A\|T"
```

test

Count the reads in a FASTQ file:

```
grep -c "^@" data.1.fastq
```

Count how many “A” and “T” are in a sequence:

```
grep -v "^>" gen.fasta | grep -cio "A\|T"
```

In a data file listing occurence on different species, which species was observed how often?

```
cat spec.txt | sort | uniq -c > specfreq.text
```

test

Count the reads in a FASTQ file:

```
grep -c "^@" data.1.fastq
```

Count how many "A" and "T" are in a sequence:

```
grep -v "^>" gen.fasta | grep -cio "A\|T"
```

In a data file listing occurence on different species, which species was observed how often?

```
cat spec.txt | sort | uniq -c > specfreq.text
```

How long are the reads in a FASTQ:

```
cat data.fq | grep -A 1 "@" | tail -1 | wc -m
```

Overview

- Working with CSV-like files
  - Tabular data
  - Common csv tasks

What are CSV-like files?

Basic structure

CSV means “comma separated values”, so originally those where tables organized in rows and columns, rows separated by newline characters, columns by commas. Optionally text values could be enclosed in double quotes. Optionally the first line could be declared as column headers.

Variants

Different separators are common (“;”, “Tab”. . .), use of newline chars is OS-dependent.

Problems with tabular data files

CSV text files are valuable data exchange and storage formats, but:

Problems with tabular data files

CSV text files are valuable data exchange and storage formats, but:

- "BOM" (Byteorder marks) are sometimes written to the start of the file as two unreadable bytes.

Problems with tabular data files

CSV text files are valuable data exchange and storage formats, but:

- “BOM” (Byteorder marks) are sometimes written to the start of the file as two unreadable bytes.
- Separator characters are part of the data and spoil the column order.

Problems with tabular data files

CSV text files are valuable data exchange and storage formats, but:

- “BOM” (Byteorder marks) are sometimes written to the start of the file as two unreadable bytes.
- Separator characters are part of the data and spoil the column order.
- Windows line endings are invisible troublemakers.

Problems with tabular data files

CSV text files are valuable data exchange and storage formats, but:

- "BOM" (Byteorder marks) are sometimes written to the start of the file as two unreadable bytes.
- Separator characters are part of the data and spoil the column order.
- Windows line endings are invisible troublemakers.
- Spaces and Tabs are easily mistaken.

Where do CSV-like files come from and where do they go?

Spreadsheet software

MS Excel, LibreOffice Calc, Freeoffice Planmaker and the whole Spreadsheet market can read and write CSV and other textual formats. This is also true for Statistics software, usually.

Where do CSV-like files come from and where do they go?

Spreadsheet software

MS Excel, LibreOffice Calc, Freeoffice Planmaker and the whole Spreadsheet market can read and write CSV and other textual formats. This is also true for Statistics software, usually.

Databases

Many database software can read and often also write those files.

Where do CSV-like files come from and where do they go?

Spreadsheet software

MS Excel, LibreOffice Calc, Freeoffice Planmaker and the whole Spreadsheet market can read and write CSV and other textual formats. This is also true for Statistics software, usually.

Databases

Many database software can read and often also write those files.

Others

Programming languages like Python, R, Perl, PHP etc. have libraries to deal with CSV & Co. in both ways.

Tools

The most important tool for CSV work is “cut”. It can cut out columns from tabular data. Options:

- -d x Specify the delimiter that is assumed to separate columns.
- -f n Specify which columns to output. Can be a comma-separated list and/or n-m expressions.

“csvtool” can be a more failsafe alternative if it is available.

CSV tasks

Get the first three columns from a spreadsheet:

```
cat data.csv | cut -d "," -f 1-3
```

CSV tasks

Get the first three columns from a spreadsheet:

```
cat data.csv | cut -d "," -f 1-3
```

Same with tab-delimited data:

```
cat data.csv | cut -f 1-3
```

Tab is the default delimiter.

CSV tasks

Get the first three columns from a spreadsheet:

```
cat data.csv | cut -d "," -f 1-3
```

Same with tab-delimited data:

```
cat data.csv | cut -f 1-3
```

Tab is the default delimiter.
Get read name and mapping position from a SAM file:

```
cat mapping.sam | cut -d " " -f 1,4
```

Overview

Editing

- Commandline-based Editors
- Common editing tasks

Interactive Editors

nano/pico A small editor that lives on the command line.
Options: -w suppresses line wrapping
Commands: See the two last lines. “^” means “Strg-”.

vim **The** commandline editor. Extremely useful, rather non-intuitive to work with
Get out with “`:q!`”

cat > file Calling cat without an input file will bind it to `stdin`.
Get out with “Strg-D”

Stream Editors

tr Quickly replaces all characters of one kind into another one. (E.g. `tr x u`)
Options: -d deletes characters

sed An *extremely* useful program. Will automatically edit files or data streams according to regular expressions. (E.g. `sed 's/X\+/U/g'`)
Options: -i will alter a file "in place", -n will suppress printing

Working with sed

sed rules generally look like this:

```
sed -opt '/addr/c/RegExMatch/RegExReplace/m' filename
```

addr	optional numeric or regular expression to define which lines to work on
c	a one letter command like “d” (delete) or “s” (substitute)
RegEx	rules what to look for and what to do to it
m	an optional modifier like “g” (global) or “i” (case insens.)

Things to do with tr

Replace Tabs with commas.

```
cat data.csv | tr "\t" ","
```

Things to do with tr

Replace Tabs with commas.

```
cat data.csv | tr "\t" ","
```

Set everything to upper case.

```
cat data.csv | tr "[:lower:] [:upper:]"
```

Things to do with tr

Replace Tabs with commas.

```
cat data.csv | tr "\t" ","
```

Set everything to upper case.

```
cat data.csv | tr "[:lower:] [:upper:]"
```

Delete all line endings

```
cat data.csv | tr -d "\n"
```

Things to do with sed

Replace all A and T with Y in a FASTA sequence.

```
sed '/^[^>]/s/[AT]/Y/gi' data.fa
```

Things to do with sed

Replace all A and T with Y in a FASTA sequence.

```
sed '/^[^>]/s/[AT]/Y/gi' data.fa
```

Change all dates from european to anglophone:

```
sed s/"\([0-9]\{1,2\}\)\.\([0-9]\{1,2\}\)\.\([12][0-9]\{3\}\)"/"\3-\2-\1"/ cal.txt
```

Things to do with sed

Replace all A and T with Y in a FASTA sequence.

```
sed '/^[^>]/s/[AT]/Y/gi' data.fa
```

Change all dates from european to anglophone:

```
sed s/"\([0-9]\{1,2\}\)\.\([0-9]\{1,2\}\)\.\([12][0-9]\{3\}\)"/"\3-\2-\1"/ cal.txt
```

Delete all FASTA descriptions:

```
cat data.fa | sed /"^>"/d
```

Common stream editing tasks

Get a certain sequence from a multi-FASTA:

```
cat multi.fa | sed '/^>/s/$/#/' | sed '/^>/s/^/#/' |
tr -d "\n" | tr "#" "\n" | grep -A 1 "X12345"
```

Common stream editing tasks

Get a certain sequence from a multi-FASTA:

```
cat multi.fa | sed '/^>/s/$/#/' | sed '/^>/s/^/#/' |
tr -d "\n" | tr "#" "\n" | grep -A 1 "X12345"
```

Insert a new codon after a given pattern:

```
cat data.fa | sed s/"\([AT].[GC][AGC]T[TG]\)"/"\1GGA"/
```

Overview

- Loops
  - for. . . in; do. . .
  - while. . . ;do. . .
  - Common tasks with loops

Loops

Loops are constructs that take care of repetitive tasks for you. They usually repeat something several times that is enclosed by the key words “do” and “done”. In most cases there is a helper variable involved that stores each element of some list in turn. Such constructs are elements of every programming language.

for Loops

For loops repeat things for every element in a list.

```
for i in $(seq 1 10); do echo $i ; done
```

seq will generate a series of numbers. The commands between “do” and “done” will be run for every number. The numbers are temporarily stored in the variable “i”.

while Loop

While loops repeat tasks as long as a certain condition is met.

```
while read a; do echo "$a" ; done < somefile.txt
```

read will get every line from “somefile.txt” and put it into the variable “a”. The while loop will be run as long as read can get new lines.

Tasks for loops

Make an alignment for every sample:

```
for s in $(cat samples.txt); do mafft "$s".fasta > "$s".aln.fasta ; done
```

Tasks for loops

Make an alignment for every sample:

```
for s in $(cat samples.txt); do mafft "$s".fasta > "$s".aln.fasta ; done
```

Make a tree for every group of sequences:

```
while read s; do mafft "$s".fasta > "$s".aln.fasta ;
sed -i /"^>"/s/"\(>.\{,10\}\)"/"\1"/ "$s".aln.fasta;
raxmlHPC8 -s "$s".aln.fasta -n "$s".tre -m GTRGAMMA ;done < samples.txt
```

Tasks for loops

Make an alignment for every sample:

```
for s in $(cat samples.txt); do mafft "$s".fasta > "$s".aln.fasta ; done
```

Make a tree for every group of sequences:

```
while read s; do mafft "$s".fasta > "$s".aln.fasta ;
sed -i /"^>"/s/"\(>.\{,10\}\)"/"\1"/ "$s".aln.fasta;
raxmlHPC8 -s "$s".aln.fasta -n "$s".tre -m GTRGAMMA ;done < samples.txt
```

Append a number to every FASTA description:

```
for i in $(seq 1 $(grep -c "^>" multi1.fasta )); do echo $i;
sed -i 0,/">.*[^0-9]$"/s/"\(^>.*[^0-9]\)$"/"\1$i"/ multi1.fasta; done
```

Overview

A little awk. . .

- Awk
- Common tasks in awk

awk

Awk is a programming language by itself. It is especially well suited for text manipulation. For small tasks you can envoke it like this:

```
cat data.csv | awk -F "," '{print "#Spalte 1: "$1"#, #Spalte 2: "$2"#"}' |
tr "#" "'" > data2.csv
```

AWK tasks

Write XML from CSV.

```
cat data.csv | awk -F "," 'BEGIN {print "<xml>"}
{print "<dataset>\n\t<art>"$1"</art>\n\t
<occurrences>"$2"</occurrences>\n</dataset>\n"}
END{print "</xml>"}'
```

Overview

EMBOSS

- European Molecular Biology Open Software Suite
- Emboss tools

"European Molecular Biology Open Software Suite"

Emboss is a free collection of small tools to accomplish common bioinformatics tasks published by the EBI. You will find many web applications to allow browser access to it. You can also use it for automated work in your own pipelines, though. The programs are designed to ask you for missing parameters interactively. You can change this behaviour by setting the option "`-filter`" which will allow you to pipe data through them.

Some EMBOSS tools

seqret	interconverts sequence formats (fasta to embl, genbank to nexus. . .)
transeq	translates into protein sequences
revseq	converts into the reverse (complemented) sequence
degapseq	removes gaps from an alignment
distmat	creates a distance matrix from an alignment

Emboss tasks

Reverse complement a set of sequences:

```
cat multi1.fa | revseq -filter > multi1.rev.fa
```

Overview

A little R. . .

- R
- Common tasks in R

R

R is a kind of hybrid between a statistics software and a programming language. As such it is especially well suited to do statistics on large datasets.

It is extremely versatile because there is a very active community providing function libraries for virtually every aspect of data analysis from descriptive statistics to NextGen-Sequences.

You can work with R by starting it: “`R`”.

This will bring you to R’s own shell where you can start using R functions. To quit just type “`q()`”.

R

R is a complex language with very special concepts of data handling and far beyond the scope of this little workshop. So just a few very basic examples...

R tasks

Read a csv file into a dataframe:

```
data <- read.csv(file="data.csv",header=FALSE, sep=",")
```

R tasks

Read a csv file into a dataframe:

```
data <- read.csv(file="data.csv",header=FALSE, sep=",")
```

Transpose it:

```
data2 <- t(data)
```

R tasks

Read a csv file into a dataframe:

```
data <- read.csv(file="data.csv",header=FALSE, sep=",")
```

Transpose it:

```
data2 <- t(data)
```

Multiply the second column by 2:

```
data[2]*2
```