EMBL Bio-IT
The Portal for Computational Biology

Intermediate Linux Course

Holger Dinkel, Frank Thommen & Toby Hodges

December 16, 2015

Contents

CHAPTER 1

More Commandline Tools

Here is a quick list of useful commandline tools which will be used throughout the rest of the document. Many of these tools have quite extensive functionality and only a very limited part can be discussed here, so the reader is encouraged to read more about these using the links given in the in the `links` section...

1.1 Commandline Tools

1.1.1 GZIP

gzip is a compression/decompression tool. When used on a file (without any parameters) it will compress it and replace the file by a compressed version with the extension '.gz' attached:

```
# ls textfile*
 textfile
# gzip textfile
# ls textfile*
 textfile.gz
```

To revert this / to uncompress, use the parameter `-d`:

```
# ls textfile*
 textfile.gz
# gzip -d textfile
# ls textfile*
 textfile
```

Note: As a convenience, on most Linux systems, a shellscript named `gunzip` exists which simply calls `gzip -d`

1.1.2 TAR

tar (tape archive) is a tool to handle archives. Initially it was created to combine multiple files/directories to be written onto tape, it is now the standard tool to collect files for distribution or archiving.

`tar` stores the permissions of the files within an archive and also copies special files (such as symlinks etc.), which makes it an ideal tool for archiving... Usually tar is used in conjunction with a compression tool such as gzip to create a compressed archive:

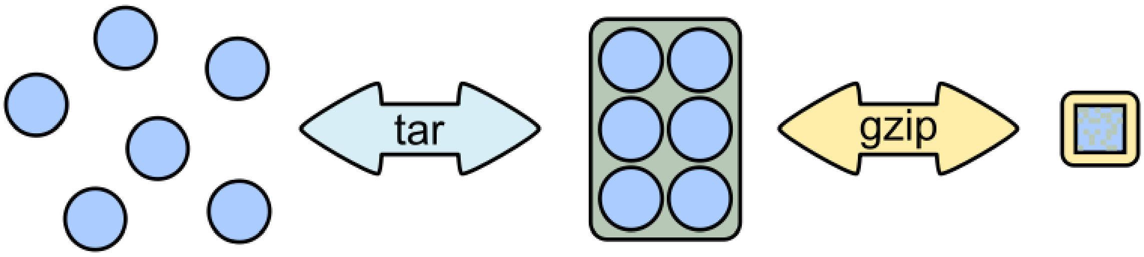


Fig. 1.1: source: Th0msn80 (Wikipedia)

The most common commandline switches are:

Option:	Effect:
-c	create an archive
-t	test an archive
-x	extract an archive
-z	use gzip compression
-f	filename filename of the archive

Note: Don't forget to specify the target filename. It needs to follow the `-f` parameter. Although you can combine options like such: `tar -czf archive.tar` the order matters, so `tar -cfz archive.tar` will *not* do what you want...

Creating an archive containing two files:

```
# tar -cf archive.tar textfile1 textfile2
```

Listing the contents of an archive:

```
# tar -tf archive.tar
 textfile1
 textfile2
```

Extracting an archive:

```
# tar -xf archive.tar
```

Creating and extracting a compressed archive containing two files:

```
# tar -czf archive.tar.gz textfile1 textfile2
# tar -xzf archive.tar.gz
```

Creating a backup (eg. before doing something dangerous?):

```
# tar -czf /folder/containing/the/BACKUP.tgz /folder/you/want/to/backup
```

1.1.3 GREP

grep finds lines matching a pattern in textfiles.

Usage: grep [options] pattern file(s)

```
# grep -i ensembl P04637.txt

 DR Ensembl; ENST00000269305; ENSP00000269305; ENSG00000141510.
 DR Ensembl; ENST00000359597; ENSP00000352610; ENSG00000141510.
 DR Ensembl; ENST00000419024; ENSP00000402130; ENSG00000141510.
 DR Ensembl; ENST00000420246; ENSP00000391127; ENSG00000141510.
 DR Ensembl; ENST00000445888; ENSP00000391478; ENSG00000141510.
 DR Ensembl; ENST00000455263; ENSP00000398846; ENSG00000141510.
```

Useful options:

Option:	Effect:
-v	Print lines that do not match
-i	Search case-insensitive
-l	List files with matching lines, not the lines itself
-L	List files without matches
-c	Print count of matching lines for each file

Count the number of fasta sequences (they start with a ">") in a file:

```
# grep -c '>' twofiles.fasta
 2
```

List all files containing the term "Ensembl":

```
# grep -l Ensembl *.txt
 P04062.txt
 P12931.txt
```

Search a file compressed with `gzip` using `zgrep`:

```
# zgrep -c '@M34567' IlluminaReads.fastq.gz
 34956188
```

1.1.4 SED

sed is a Stream EDitor, it modifies text (text can be a file or a pipe) on the fly.

Usage: ‘`sed command file`‘,

The most common usecases are:

Usecase	Command:
Substitute TEXT by REPLACEMENT:	‘s/TEXT/REPLACEMENT/’
Transliterate the characters x a, and y b:	‘y/xy/ab/’
Print lines containing PATTERN:	‘/PATTERN/p’
Delete lines containing PATTERN:	‘/PATTERN/d’

```
# echo "This is text." | sed 's/text/replaced stuff/'
 This is replaced stuff.
```

By default, text substitution are performed only once per line. You need to add a trailing 'g' option, to make the substitution 'global' ('s/TEXT/REPLACEMENT/g'), meaning all occurrences in a line are substituted (not just the first in each line). Note the difference:

```
# echo "ACCAAGCATTGGAGGAATATCGTAGGTAAA" | sed 's/A/_/'
 _CCAAGCATTGGAGGAATATCGTAGGTAAA

# echo "ACCAAGCATTGGAGGAATATCGTAGGTAAA" | sed 's/A/_/g'
 _CC__GC_TTGG_GG__T_TCGT_GGT___
```

You can use transliteration to replace all instances of a character with another character. For example, to switch Thymines to Uridines in a sequence:

```
# echo "AGTGGCTAAGTCCCTTTAATCAGG" | sed 'y/T/U/'
 AGUGGCUAAGUCCCUUUAAUCAGG
```

In the pattern specified in the `sed` command, each character in the first set is replaced with the character in the equivalent position in the second set. For example, to get the reverse transcript of a DNA sequence:

```
# echo "AGTGGCTAAGTCCCTTTAATCAGG" | sed 'y/ACGT/UGCA/'
 UCACCGAUUCAGGGAAAUUAGUCC
```

This is the complementary sequence, but we wanted the reverse complement, so we need to use the Linux command `rev` to reverse the output of the `sed` command:

```
# echo "AGTGGCTAAGTCCCTTTAATCAGG" | sed 'y/ACGT/UGCA/' | rev
 CCUGAUUAAAGGGACUUAGCCACU
```

When used on a file, sed prints the file to standard output, replacing text as it goes along:

```
# echo "This is text" > textfile
# echo "This is even more text" >> textfile
# sed 's/text/stuff/' textfile
 This is stuff
 This is even more stuff
```

sed can also be used to print certain lines (not replacing text) that match a pattern. For this you leave out the leading 's' and just provide a pattern: '/PATTERN/p'. The trailing letter determines, what sed should do with the text that matches the pattern ('p': print, 'd': delete)

```
# sed '/more/p' textfile
 This is text
 This is even more text
 This is even more text
```

As sed by default prints each line, you see the line that matched the pattern, printed twice. Use option '-n' to suppress default printing of lines.

```
# sed -n '/more/p' textfile
 This is even more text
```

Delete lines matching the pattern:

```
# sed '/more/d' textfile
 This is text
```

Multiple sed statements can be applied to the same input stream by prepending each by option '-e' (edit):

```
# sed -e 's/text/good stuff/' -e 's/This/That/' textfile
 That is good stuff
 That is even more good stuff
```

Normally, sed prints the text from a file to standard output. But you can also edit files in place. Be careful - this will change the file! The '-i' (in-place editing) won't print the output. As a safety measure, this option will ask for an extension that will be used to rename the original file to. For instance, the following option '-i.bak' will edit the file and rename the original file to textfile.bak:

```
# sed -i.bak 's/text/stuff/' textfile
# cat textfile
 This is stuff
 This is even more stuff
# cat textfile.bak
 This is text
 This is even more text
```

1.1.5 AWK

awk is more than just a command, it is a complete text processing language (the name is an abbreviation of the author's names). Each line of the input (file or pipe) is treated as a record and is broken into fields. Generally, awk commands are of the form:

```
awk condition { action }
```

where:

- condition is typically an expression
- action is a series of commands

If no condition is given, the action is applied to each line, otherwise just to the lines that match the condition.

```
# awk '{print}' textfile
 This is text
 This is even more text
```

```
# awk '/more/ {print}' textfile
 This is even more text
```

awk reads each line of input and automatically splits the line into columns. These columns can be addressed via $1, $2 and so on ($0 represents the whole line). So an easy way to print or rearrange columns of text is:

```
# echo "Bob likes Sue" | awk '{print $3, $2, $1}'
 Sue likes Bob

# echo "Master Obi-Wan has lost a planet" | awk '{print $4,$5,$6,$1,$2,$3}'
 lost a planet Master Obi-Wan has
```

awk splits text by default on whitespace (spaces or tabs), which might not be ideal in all situations. To change the field separator (FS), use option ‘-F’ (remember to quote the field separator):

```
# echo "field1,field2,field2" | awk -F',' '{print $2, $1}'
 field2 field1
```

Note two things here: First, the field separator is not printed, and second, if you want to have space between the output fields, you actually need to separate them by a comma or they will be concatenated together...

```
# echo "field1,field2,field2" | awk -F',' '{print $1 $2 $3}'
 field1field2field3
```

You can also combine the pattern matching and the column selection techniques, in this example we’ll print only the third column of the lines matching the pattern ‘PDBsum’ (case sensitive):

```
$ awk '/PDBsum/ {print $3}' P12931.txt
1A07;
1A08;
1A09;
1A1A;
...
```

awk really is powerful in filtering out columns, you can for instance print only certain columns of certain lines. Here we print the third column of those lines where the second column is ‘PDBsum’:

```
# awk '$2=="PDBsum;" {print $3}' P12931.txt
1A07;
1A08;
1A09;
1A1A;
...
```

Note the double equal signs “==” to check for equality and note the quotes around “PDBsum;”. If you want to match a field, but not exactly, you can use ‘~’ instead of ‘==’:

```
# awk '$4~"sum" {print $3}' P12931.txt
1A07;
1A08;
1A09;
1A1A;
...
```

1.2 I/O Redirection

Three IO "channels" are available by default:

- **Standard input (STDIN, Number: 0)**: The input for your program, normally your keyboard but can be an other program (when using pipes or IO redirection)
- **Standard output (STDOUT, Number: 1)**: Where your program writes its regular output to. Normally your terminal
- **Standard error (STDERR, Number: 2)**: Where your programs normally write their error message to. Normally your terminal

Input, output and error messages can be redirected from their default "targets" to others. If using the file descriptor numbers (0, 1, 2) in redirections, then there must be no whitespace between the numbers and the redirection operators.

Hint: Redirect to `/dev/null` to discard the output of any command

Write the output of *cmd* into *afile*. This will **overwrite** *afile*:

```
$ cmd > afile
```

Write the output of *cmd* into *afile*. This will **append** to *afile*:

```
$ cmd >> afile
```

Discard the output of *cmd*

```
$ cmd > /dev/null
```

Write the output of *cmd* into *afile* (overwriting *afile*!) and write STDERR to the same place:

```
$ cmd > afile 2>&1
```

Append the output and error messages of *cmd* to *afile*:

```
$ cmd >> afile 2>&1
```

Same as above:

```
$ cmd > afile 2> afile
```

Append the output of *cmd* to *afile* and discard error messages:

```
$ cmd >> afile 2>/dev/null
```

Three times the same: Discard output and error messages completely:

```
$ cmd > /dev/null 2>&1
$ cmd > /dev/null 2>/dev/null
$ cmd >& /dev/null
```

Use output of *cmd2* as standard input for *cmd1*:

```
$ cmd1 < cmd2
```

See also

- Bash One-Liners Explained, Part III: All about redirections [1]
- Bash Redirections Cheat Sheet [2]
- Redirection Tutorial [3]

1.3 Variables

The shell knows two types of variables: "Local" *shell* variables and "global" exported *environment* variables. By convention, environment variables are written in uppercase letters.

Shell variables are **only available to the current shell** and not inherited when you start an other shell or script from the commandline. Consequently, these variables will not be available for your shellscripts.

Environment variables are **passed on** to shells and scripts started from your current shell.

1.3.1 Setting, Exporting and Removing Variables

Variables are set (created) by simply assigning them a value

```
$ MYVAR=something
$
```

Note: There must be no whitespace surrounding the equal sign!

To create an environment variable, `export` is used. You can either export while assigning a value or in a separate step. Both of the following procedures are equivalent:

1.
   ```
   $ export MYGLOBALVAR="something else"
   $
   ```

[1] http://www.catonmat.net/blog/bash-one-liners-explained-part-three

[2] http://www.catonmat.net/blog/bash-redirections-cheat-sheet

[3] http://wiki.bash-hackers.org/howto/redirection_tutorial

2.
   ```
   $ MYGLOBALVAR=”something else”
   $ export MYGLOBALVAR
   $
   ```

Note: There is no `$` in front of the variable: To reference the variable itself (not its content) the name is used without `$`

Variables are removed with `unset`:

```
$ unset MYVAR
$
```

Note: Assigning a variable an empty value (i.e. `MYVAR=`) will *not* remove it but simply set its value to the empty string!

1.3.2 Listing Variables

You can list all your current environment variables with `env` and all shell variables with `set`. The list of shell variables will also contain all environment variables

```
$ set | more
BASH=/bin/bash
BASH_ARGC=()
BASH_VERSION='4.1.2(1)-release'
COLORS=/etc/DIR_COLORS.256color
COLUMNS=181
...
$
```

1.3.3 Variable Inheritance

Only environment variables will be available in shells and scripts started from your current shell. However in shell commands run in subshells (i.e. commands run within round brackets) also local (shell) variables of your current shell are available.

Examples

Consider the following small shellscript *vartest.sh*:

```
#!/bin/sh
echo $MYLOCALVAR
echo $MYGLOBALVAR
echo -----
```

We will use it in the following examples to illustrate the various variable inheritances:

1. Set the variables and run the script i.e. in a new shell:

```
$ export MYGLOBALVAR=”I am global”
$ MYLOCALVAR=”I am local”
$ ./vartest.sh
I am global
-----
$
```

2. “source” the script, i.e. run it within your current shell:

```
$ source ./vartest.sh
I am local
I am global
-----
$
```

3. Access the variables in a subshell:

```
$ (echo $MYGLOBALVAR; echo $MYLOCALVAR)
I am global
I am local
$
```

1.4 Tips and Tricks

1.4.1 Quoting

In Programming it is often necessary to “glue together” certain words. Usually, a program or the shell splits sentences by whitespace (space or tabulators) and treats each word individually. In order to tell the computer that certain words belong together, you need to “quote” them, using either single (‘) or double (”) quotes. The difference between these two is generally that within double quotes, variables will be expanded, while everything within single quotes is treated as string literal. When setting a variable, it doesn’t matter which quotes you use:

```
# MYVAR=This is set
 -bash: is: command not found

# MYVAR='This is set'
# echo $MYVAR
 This is set
# MYVAR="This is set"
# echo $MYVAR
 This is set
```

However, it does matter, when using (expanding) the variable: Double quotes:

```
# export MYVAR=123
# echo "the variable is $MYVAR"
 the variable is 123
```

```
# echo "the variable is set" | sed "s/set/$MYVAR/"
 the variable is 123
```

Single quotes:

```
# export MYVAR=123
# echo 'the variable is $MYVAR'
 the variable is $MYVAR
# echo "the variable is set" | sed 's/set/$MYVAR/'
 the variable is $MYVAR
```

Weird things can happen when parsing data/text that contains quote characters:

```
# MYVAR='Don't worry. It's ok.'; echo $MYVAR
 >
# you need to press Ctrl-C to abort
# MYVAR="Don't worry. It's ok."; echo $MYVAR
 Don't worry. It's ok.
```

Expanding and Escaping

You already learned how to expand a variable such that its value is used instead of its name:

```
# export MYVAR=123
# echo "the variable is $MYVAR"
 the variable is 123
```

"Escaping" a variable is the opposite, ensuring that the literal variable name is used instead of its value:

```
# export MYVAR=123

# echo "the \$MYVAR variable is $MYVAR"
 the $MYVAR variable is 123
```

Note: The "escape character" is usually the backslash "\".

1.4.2 Keyboard Shortcuts

When getting comfortable with working on the command line, it can be helpful to learn some tricks that can save you time, better manage your session, and help you to avoid annoying errors due to typos.

Tab-Completion: A Reminder

You're probably already aware of tab-completion, where you push the `tab` key to complete the name of a command, file, directory, etc. This is a huge time-saver and great tool for preventing the accidental inclusion of errors.

Move Quickly Through the Command Line

As well as tab-completion, you might be aware of CTRL-A to jump the cursor to the beginning of a line, and CTRL+E to jump to the end. On most systems, using the arrow keys while holding down the alt key will jump left or right by one word (or word-like string) at a time.

When editing a line, CTRL-W can be used to delete left from the current cursor position to the next beginning of a word. CTRL+U will delete left from the current cursor position to the beginning of the line.

Searchable Command History

You're probably aware of the command history, and that you can use the up and down arrow keys to scroll back and forth throughout that history. You can also use CTRL+R to search that command history. If you type CTRL+R and then the beginning of a command, you will see the most recent command in the history that matches that pattern (anywhere in the command). You can hit CTRL+R again to scroll backwards through the matches.

Job Management

Use CTRL+C to abort the current process, and CTRL+D to close the current shell.

If you don't want to abort, you might instead want to use CTRL+Z to suspend the current process. You can resume the most recently-suspended job with fg, to run it in the 'foreground' of the shell, or bg to run it in the 'background'. In the shell, a command running in the foreground is a job that will prevent the user from executing further commands until the job has finished. A job running in the background will continue to run while the user can carry on using the shell prompt to execute other commands. On a related note: to put a job in the background when you execute it, just add "&" to the end of the command.

If you have multiple jobs running/suspended at one time, you can view a list of these processes and their current status with jobs:

```
# sleep 250 &
[1] 19697
# sleep 100
^Z
[1]+  Stopped                 sleep 100
# jobs
[1]+  Stopped                 sleep 100
[2]-  Running                 sleep 250 &
```

As mentioned before, you can restart the most recently-suspended job with fg or bg. To restart another job in the list, you can refer to it with %1 for job number 1 in the list (sleep 100 in the example above), %2 for job 2, and so on. If, instead of restarting a job, you want to kill a suspended process, you can use the kill command and specify the job afterwards:

```
# jobs
[1]+  Stopped                 sleep 100
[2]-  Running                 sleep 250 &
# kill %2
[2]-  Terminated: 15          sleep 250
```

The jobs list contains details of all running or stopped tasks that were initiated within the current session. If you try to leave a session with exit while you still have a job running or suspended, you will receive a warning message. (Note that this is one of the rare occasions where the command line interface will ask you if you're sure before doing something that could be potentially bad for you.) Use exit a second time and the session will end, killing any remaining jobs as it does so.

CHAPTER 2

Commandline Exercises

2.1 TAR & GZIP

1. Use gzip to compress the file P12931.txt
2. Decompress the resulting file P12931.txt.gz (revert previous command)
3. Use tar to create an archive containing all fasta files in the current directory into an archive called "fastafiles.tar"
4. Use gzip to compress the archive "fastafiles.tar"
5. How can you achieve the two previous steps "using tar to create archive" and "gzip the archive" in one command?
6. Test (list the contents of) the compressed archive "fastafiles.tar.gz"
7. Download the compressed PDB file for entry 1Y57 from rcsb.org (eg. `wget "http://www.rcsb.org/pdb/files/1Y57.pdb.gz"`) and decompress it.

2.2 GREP

1. Which of the DNA files ENST0* contains "TATATCTAA" as part of the sequence?
2. List only the names of the DNA files ENST0* that contain "CAACAAA" as part of the sequence.
3. Considering the previous example, would you consider grep a suitable tool to perform motif searches? Why not? Try to find the pattern "CAACAAA" by manual inspection of the first three lines of each sequence.
4. Count the number of ATOMs in the file 1Y57.pdb.
5. Does this number agree with the annotated number of atoms? The PDB file has a comment which tells you how many atoms there are annotated in this file. This comment can be found by searching for the term "protein atoms" (use quotes and case insensitive search here!).

2.3 SED

1. Use sed to print only those lines that contain “version” in the files P05480.txt and P04062.txt
2. Use sed to change the text “sequence version 3” to “sequence version 4” in the files P05480.txt and P04062.txt (without actually changing the files, just printing)
3. Use sed to update the text “sequence version 3” to “sequence version 4” in the files P05480.txt and P04062.txt (this time, make the changes directly in the files)
4. Replace (transliterate) all occurrences of “r” by “l” and “l” by “r” (at the same time) in the file PROTEINS.txt (so that “structural” becomes “stluctular”)

2.4 AWK

1. Use awk to print only those lines that contain “version” in the files P12931.txt and P05480.txt and think about how this procedure is different to sed.
2. For all FASTA files that begin with “P” (“P*.fasta”) print only the second item of the header (split on “|”) eg. for “>sp|P12931|SRC_HUMAN Proto-oncogene”, print only “P12931”
3. The file “P12931.csv” contains phosphorylation sites in the protein P12931. (If the file “P12931.csv” does not exist, use `wget http://phospho.elm.eu.org/byAccession/P12931.csv` to download it).
   (a) Column three of this file lists the amino acid position of the phosphorylation site. You are only interested in position 17 of the protein. Try to use “grep” to filter out all these lines containing “17”.

   (b) Now use awk to show all lines containing “17”.

   (c) Next try show only those lines where column three equals 17 (Hint: The file is semicolon-separated...).

   (d) Finally print the PMIDs (column 6) of all lines that contain “17” in column 3.

2.5 Quoting and Escaping

1. Familiarize yourself with quoting and escaping.
   1. Run the following commands to see the difference between single and double quotes when expanding variables:

      ```
      $ echo "$HOSTNAME"
      ...
      $ echo '$HOSTNAME'
      ```

   2. Next, use ssh to login to a different machine to run the same command there, again using both quoting methods:

```
$ ssh pc-atcteach01 'echo $HOSTNAME'
...
$ ssh pc-atcteach01 "echo $HOSTNAME"
```

2. Closely inspect the results; is that what you were expecting? Discuss this with your neighbour.

CHAPTER 3

Basic Shell Scripting

3.1 What is a Script?

A script is nothing else than a number of shell command place together in a file. The simplest script is maybe just a complex oneliner that you don't want to type each time again. More complex scripts are seasoned with control elements (conditions and loops) which allow for a sophisticated command flow. scripts might allow for configuration and customization, thus allowing one script to be flexibly used in several different environments. Whatever you do in a script, you can also do on the commandline. This is also the first way to test your scripts step by step!

3.2 Script Naming and Organization

It is good practice - though not technically required - to give your scripts an extension which specifies their type. I.e. "*.sh*" for Bourne Shell and Bourne Again Shell scripts, "*.csh*" for C-Shell scripts. Sometimes "*.bash*" for Bourne Again Shell scripts is used.

We recommend to either store all scripts in one location (e.g. `~/bin`) and add this location to your `$PATH` variable (see *Variables* (page 8)) or to store the scripts together with the files that are processed by the script.

Hint: If you use scripts to process data, then the scripts should probably be archived together with the data files!

3.3 Running a Script

There are basically three ways to run a script:

1. the location to your script is not in your `$PATH` variable, then you have to specify the full path to the script:

   ```
   $ /here/is/my/script.sh
   [...]
   $
   ```

2. the location to the script is in the `$PATH` variable, then you can simply type its name:

```
$ script.sh
[...]
$
```

In both situations, the script will need to have execute permissions to be run. If for some reason you can only read but not execute the script, then it can still be run in the following way:

3. specifying the interpreter (i.e. the program required to run the script). For shellscripts this is the appropriate shell). The full path (relative or absolute) to the script has to be provided in this case, no matter whether the script location is already contained in `$PATH` or not:

```
$ /bin/sh /here/is/my/script.sh
[...]
$
```

3.3.1 Basic Structure of a Shellscript

Shellscripts have the following general structure:

- A line starting with “`#!`” which defines the interpreter. This line is called the *shebang line* and must be the first line in a script.
- A section where the configuration takes place, e.g. paths, options and commands are defined and it is made sure, that all prerequisites are met.
- A section where the actual processing is done. This includes error handling.
- A controlled exit sequence, which includes cleaning up all temporary files and returning a sensible exit status.

This is merely a recommendation to keep your scripts well structured. None of these sections are mandatory.

3.3.2 Readability and Documentation

Make your script easily readable. Use comments and whitespace and avoid super compact but hard to understand commandlines. Always take into account that not only the shell, but also human beings will probably have to read and understand your script. (see *Breaking up long lines* (page 34)) Even if your script is very simple - document it! This helps others understand what you did, but - most importantly - it helps you remember what you did, when you have to reuse the script in the future.

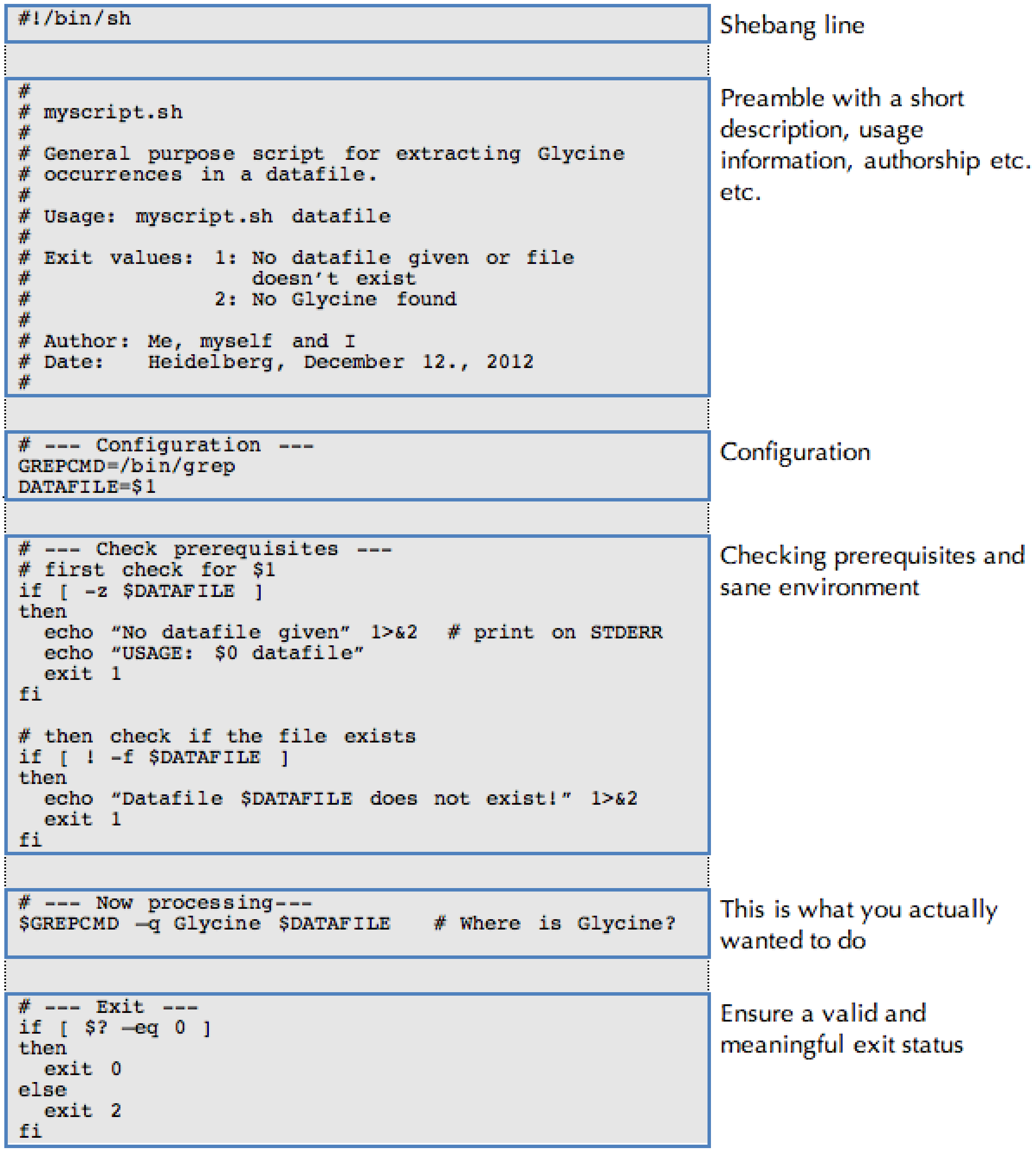
Documentation is done either by writing comments into the script or by creating a special documentation file (`README.txt` or similar). Documenting in the script can be done in several ways:

- A preamble in the script, outlining the purpose, parameters and variables of the script as well as some information about authorship and perhaps changes.
- Within the script as blocks of text or “End of line” comments.

To write comments, use the hash sign (“ #”). Everything after a “#” is ignored when executing a script.

3.3.3 Anatomy of a Shellscript

Let’s have a look at the following script, breaking it down into individual parts. First, the full script:



```
#!/bin/sh
```

```
#
# myscript.sh
#
# General purpose script for extracting Glycine
# occurrences in a datafile.
#
# Usage: myscript.sh datafile
#
# Exit values: 1: No datafile given or file
#                 doesn’t exist
#              2: No Glycine found
#
# Author: Me, myself and I
# Date:   Heidelberg, December 12., 2012
#
```

```
# --- Configuration ---
GREPCMD=/bin/grep
DATAFILE=$1
```

```
# --- Check prerequisites ---
# first check for $1
if [ -z $DATAFILE ]
then
   echo “No datafile given” 1>&2   # print on STDERR
   echo “USAGE: $0 datafile”
   exit 1
fi

# then check if the file exists
if [ ! -f $DATAFILE ]
then
   echo “Datafile $DATAFILE does not exist!” 1>&2
   exit 1
fi
```

```
# --- Now processing---
$GREPCMD –q Glycine $DATAFILE   # Where is Glycine?
```

```
# --- Exit ---
if [ $? –eq 0 ]
then
   exit 0
else
   exit 2
fi
```

Shebang line
Preamble with a short description, usage information, authorship etc. etc.
Configuration
Checking prerequisites and sane environment
This is what you actually wanted to do
Ensure a valid and meaningful exit status

You can see from this example, that very often the actual computation is only a small part of the code. The rest of the scripts deal with prerequisites, error handling, user dialogue, exit status etc. etc.

3.3.4 Reporting Success or Failure - The Exit Status

Commands report their success or failure by their exit status. An exit status of 0 (zero) indicates success(!), while any exit status greater then 0 indicates an error. Some commands report more than one error status. Refer to the respective manpages to see the meanings of the different exit stati. The exit status of a script is usually the exit status of the last executed command, which is reported by the environment variable $?:

Example: Displaying the exit status of the (successfully run) pwd command:

```
$ pwd
/home/fthommen
$ echo $?
0
$
```

Example: Displaying the exit status of the (unsuccessfully run) touch command:

```
$ touch /afile
touch: cannot touch ‘/afile’: Permission denied
$ echo $?
1
$
```

See *Ensuring a Sensible Exit Status* (page 32) about how to control the exit status of your script.

3.3.5 Command Grouping and Sequences

Commands can be concatenated to be executed one after the other unconditionally or based on the success of the respective previous command:

***cmd1; cmd2* –** Execute commands in sequence

Example: Create a directory and change into it:

```
$ pwd
/home/fthommen
$ mkdir a; cd a
$ pwd
/home/fthommen/a
$
```

***cmd1 && cmd2* –** Execute cmd2 only if cmd1 was successful:

Example: Create a directory and, if successful, change into it:

```
$ pwd
/home/fthommen
$ mkdir a && cd a
$ pwd
/home/fthommen/a
$
```

Example: Confirm that `/etc` exists:

```
$ cd /etc && echo "/etc exists"
/etc/exists
$
```

***cmd1* || *cmd2* –** Execute cmd2 only if cmd1 was not successful:

Example: Create a directory and, if not successful, print an error message:

```
$ mkdir /bin/a || echo "mkdir didn't work!"
mkdir: cannot create directory `/bin/a': Permission denied
mkdir didn't work!
$
```

Example: Decompress a gzipped file if it exists, or download it if not:

```
$ gzip -d 2W73.pdb || wget "http://www.rcsb.org/pdb/files/2W73.pdb.gz"
$
```

You can mix multiple `&&` and `||` controls into a single line.

Example: Create a directory and, if successful, change into it, if not successful, print an error message:

```
$ mkdir /bin/a && cd a || echo "Could not create directory a"
mkdir: cannot create directory `/bin/a': Permission denied
Could not create directory a
$
$ mkdir ~/a && cd ~/a || echo "Could not create directory a"
$ pwd
/home/fthommen/a
$
```

***(cmds)* –** Group commands to create one single output stream: The commands are run in a subshell (i.e. a new shell is opened to run them):

Example: Change into `/etc` and list content. You are still in the same directory as you were before:

```
$ pwd
/home/fthommen
$ (cd /etc; ls)
[... etc directory listing here ...]
$ pwd
/home/fthommen
$
```

***{ cmds; }* –** Group commands to create one single output stream: The commands are run in the current (!) shell.

Note: The opening "{" must be followed by a blank and the last command must be succeeded by a *semicolon* (';")

Example: Change into `/etc` and list its content. You are still in `/etc` after the bracketed expression (compare to the example above):

```
$ pwd
/home/fthommen
$ { cd /etc; ls; }
[... directory listing here ...]
$ pwd
/etc
$
```

3.4 Control Structures

The following syntax elements will be described for sh/bash *and* for csh/tcsh. However since this course is mainly about sh/bash, examples will only be given for sh/bash. Some notes about csh/tcsh specialities might be given in the text. This is only a selection of the most useful or most common elements. There are much more in the manpages. All shells offer myriads of possibilities which cannot possibly be demonstrated in this course. Some of the described features might be specific to bash and not be available in a classical Bourne Shell on other systems.

3.4.1 Conditional Statements

if - then - else

`if - then - else` is the most basic conditional statement: Do something depending on certain conditions. Its basic syntax is:

sh/bash	**csh/tcsh**

`if condition1`
`then`
`  commands`
`elif condition2`
`then`
`  more commands`
`[...]`
`else`
`  even more commands`
`fi`
	`if (condition) then`
	`  commands`
	`else if (condition2) then`
	`  more commands`
	`[...]`
	`else`
	`  even more commands`
	`endif`

Conditions can be either the **exit status of a command** or the **evaluation of a logical or arithmetic expression**:

1. Evaluating the exit status of a command: Simply use the command as condition. For example:

```
if grep -q root /etc/passwd
then
  echo root user found
else
```

```
  echo No root user found
fi
```

Note: In *csh/tcsh*

1. To evaluate the exit status of a command in it must be placed within curly brackets with blanks separating the brackets from the command: `if ({ grep -q root /etc/passwd }) then [...]`
2. Redirection of commands in conditions does not work

Hint: Redirect the output of the command to be evaluated to `/dev/null` if you are only interested in the exit status and if the command doesn't have a "quiet" option.

2. Evaluating of conditions or comparisons:

   Conditions and comparisons are evaluated using a special command `test` which is usually written as "`[`" (no joke!). As "`[`" is a command, it must be followed by a blank. As a speciality the "`[`" command must be ended with "`]`" (note the preceding blank here)

   Note: In csh/tcsh the `test` (or `[`) command is not needed. Conditions and comparisons are directly placed within the round braces.

sh/bash		**csh/tcsh**
	File condition	
-e *file*	*file* exists	-e *file*
-f *file*	*file* exists and is a regular *file*	-f *file*
-d *file*	*file* exists and is a directory	-d *file*
-r *file*	*file* exists and is readable	-r *file*
-w *file*	*file* exists and is writeable	-w *file*
-x *file*	*file* exists and is executable	-x *file*
-s *file*	*file* exists and has a size > 0	
	file exists and has zero size	-z *file*
	String Comparison	
-n s1	String s1 has non-zero length	
-z s1	String s1 has zero length	
s1 = s2	Strings s1 and s2 are identical	s1 == s2
s1 != s2	Strings s1 and s2 differ	s1 != s2
string	String string is not null	
	Integer Comparison	
n1 -eq n2	n1 equals n2	n1 == n2
n1 -ge n2	n1 is greater than or equal to n2	n1 >= n2
n1 -gt n2	n1 is greater than n2	n1 > n2
n1 -le n2	n1 is less than or equal to n2	n1 <= n2
n1 -lt n2	n1 is less than n2	n1 < n2
n1 -ne n2	n1 it not equal to n2	n1 != n2
	Combination of conditions	
! *cond*	True if condition *cond* is not true	! *cond*
cond1 -a *cond2*	True if conditions *cond1* and *cond2* are both true	*cond1* && *cond2*
cond1 -o *cond2*	True if conditions *cond1* or *cond2* is true	*cond1* || *cond2*

Examples: Test for the existence of the directory *sequence_files*:

```
if [ -e ./sequence_files ]
then
  ls -1 ./sequence_files/*.fasta
else
  echo no sequence_files directory here
fi
```

or:

```
if test -e ./sequence_files
then
  ls -1 ./sequence_files/*.fasta
else
  echo no sequence_files directory here
fi
```

Note: Bash supports another way of evaluating conditional expressions with `[[ expression ]]`. This syntax element allows for more readable expression combination and handles empty variables better. However it is not backwards compatible with the original Bourne Shell. See the bash manpage for more information

case

The `case` statement implements a more compact and better readable form of if - elif - elif - elif etc. Use this if your variable (you can *only* check for variables with case) can have a distinct number of valid values. A typical usage of case will follow later.

The basic syntax is:

sh/bash	**csh/tcsh**
`case variable in`	`switch (variable)`
`  pattern1)`	`  case pattern1:`
`    commands`	`    commands`
`    ;;`	`    breaksw`
`  pattern2)`	`  case pattern2:`
`    commands`	`    commands`
`    ;;`	`    breaksw`
`  *)`	`  default:`
`    commands`	`    commands`
`    ;;`	`endsw`
`esac`	

Note: for the patterns “`*`”, “`?`” and “`[...]`” can be used

Note: The “`*)`” (sh/bash) and “`default:`” (csh/tcsh) patterns are “catch-all” patterns which match everything not matched above. It is often used to detect invalid values of variable.

Note: Multiple patterns can be handled by separating them with “`|`” in sh/bash or by successive case statements in csh/tcsh.

Example: Check if /opt/ or /usr/ paths are contained in `$PATH`:

```
case $PATH in
 */opt/* )
    echo /opt/ paths found in \$PATH
    ;;
 */usr/* )
    echo /usr/ paths found in \$PATH
    ;;
 *)
    echo '/opt and /usr are not contained in $PATH'
    ;;
esac
```

or:

```
case $PATH in
 */opt/* | */usr/* )
    echo /opt/ or /usr/ paths found in \$PATH
    ;;
 *)
    echo '/opt and /usr are not contained in $PATH'
    ;;
esac
```

3.4.2 Loops

for / foreach

The for and foreach statements respectively will loop through a list of given values and run the given statements for reach run:

sh/bash	csh/tcsh

`for variable in list`
`do`
`  commands`
`done`
	`foreach variable (list)`
	`  commands`
	`end`

list is a list of strings, separated by whitespaces

Examples: List filenames and count number of sequences in every FASTA file in ./sequence_files:

```
for FILE in ./sequence_files/*.fasta
do
  echo " * $FILE"
  grep -c '\>' $FILE
done
or
for FILE in `ls ./sequence_files/*.fasta`
do
  echo " * $FILE"
  grep -c '\>' $FILE
done
```

while / until

The `while` and `until` loops execute your commands while (or until respectively) a certain condition is met:

sh/bash	**csh/tcsh**

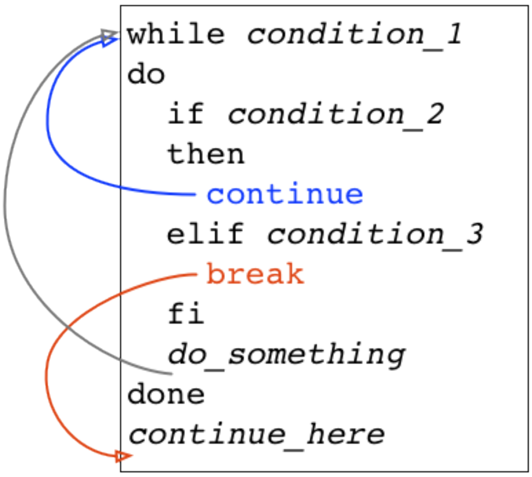
`while condition`
`do`
`  commands`
`done`
 	
`until condition`
`do`
`  commands`
`done`
	`while (condition)`
	`  commands`
	`end`

The conditions are constructed the same way as those used in if statements.

Note: The until statement is not available in csh/tcsh.

"Manual" loop control

Instead of (or additionally to) the built-in loop control in for/foreach, while and until loops, you can control exiting and continuing them with `break` and `continue`: `break` "breaks out" of the innermost loop (loops can be nested!) and continues after the end of the loop. `continue` skips the rest of the current (innermost) loop and starts the next iteration



```
while condition_1
do
  if condition_2
  then
    continue
  elif condition_3
    break
  fi
  do_something
done
continue_here
```

Fig. 3.1: Loop control

Symbol	
▬	Regular loop cycle
🟥	`break` due to *condition_2*
🟦	`continue` due *to condition_3*

3.5 Making Scripts Flexible

Scripts are most useful, if they can be reused. Copying scripts and changing them to fit the new situation is time-consuming and error-prone. Additionally if you add an improvement to the current script, then all previous versions will stay without it. Having one script with the possibility to configure it, is usually the better way. Customization of scripts can be achieved by either using variables or by adding the possibility to use your own commandline options and arguments.

3.5.1 Configurable Scripts

Using Variables

Any value - be it paths, commands or options - that is specific to individual applications or your script, should not be hardcoded (i.e. used literally within the script). Instead you should use variables to refer to them:

Bad example: You have to change two instances of the path each time you want to list another directory:

```
#!/bin/sh

echo "The directory /etc contains the following files:"
ls /etc
```

Good example: The path is now in a variable and only one instance has to be changed each time (less work, fewer errors):

```
#!/bin/sh

MYDIR=/etc

echo "The directory $MYDIR contains the following files:"
ls $MYDIR
```

Of course, you'll still have to modify the script each time you want to list the content of another directory. A more flexible way of customization would be to use a settings file.

Using a Settings File

Instead of having your configurable section within the script, it can be "outsourced" to its own file. This file is basically a shellscript which is run within the primary script. To run commands from a file within the current environment, the commands source (bash, csh/tcsh) or . (dot) (sh/bash) are used:

The settings file, e.g. settings.ini:

```
MYDIR=/etc
```

The script:

```
#!/bin/sh

. ./settings.ini

echo "The directory $MYDIR contains the following files:"
ls $MYDIR
```

3.5.2 Defining your own Commandline Options and Arguments

The best way to configure a script is to allow for your own commandline options and arguments. Commandline arguments are available within the script as so-called positional parameters `$1`, `$2`, `$3`: etc. `$0`: contains the name of the script. The variables important when dealing with commandline parameters are:

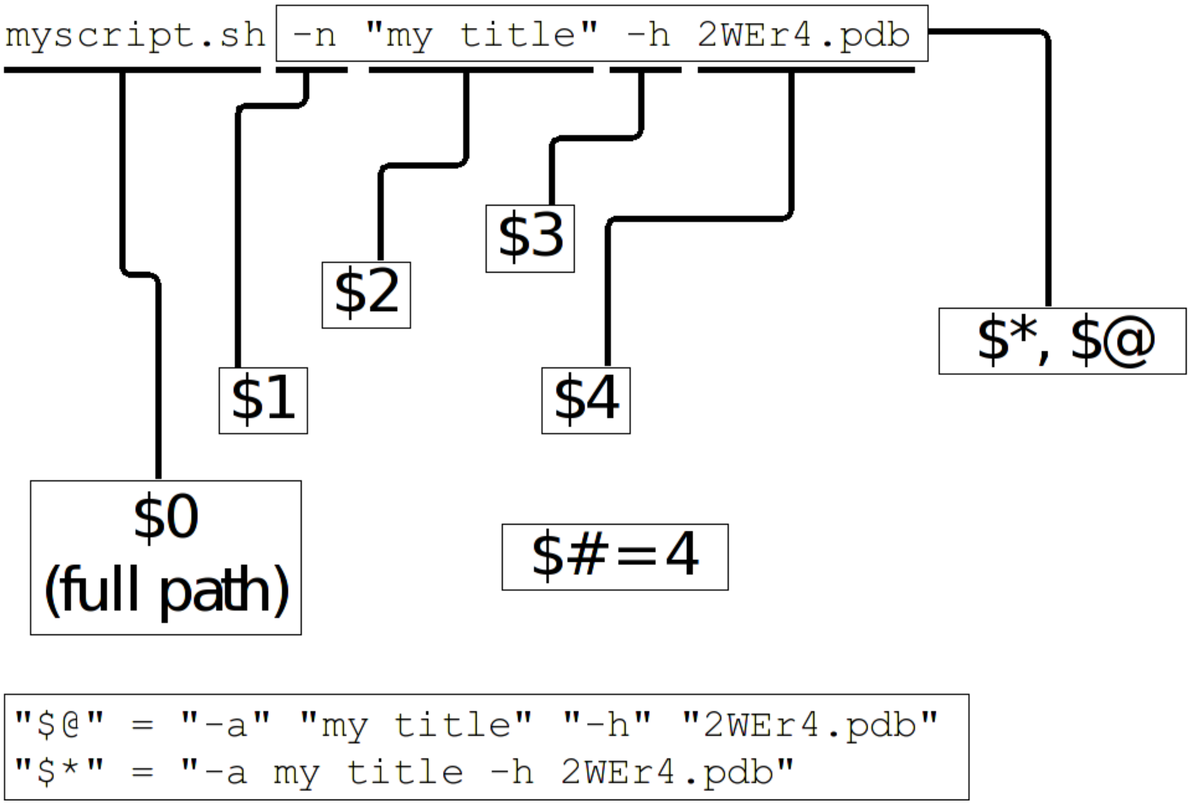
$0: path to the script. Either the path as you specified it or the full path if the script was executed through `$PATH`

$1, $2, $3, etc: Positional parameters (i.e. commandline arguments)

$#: Current number of positional parameters

$*: All positional parameters. If used within double quotes (“`$*`”), then it will expand to the list of all positional parameters, where the complete list is quoted

$@: All positional parameters. If used within double quotes (“`$@`”), then it will expand to the list of all positional parameters, where each parameter is individually quoted



```
"$@" = "-a" "my title" "-h" "2WEr4.pdb"
"$*" = "-a my title -h 2WEr4.pdb"
```

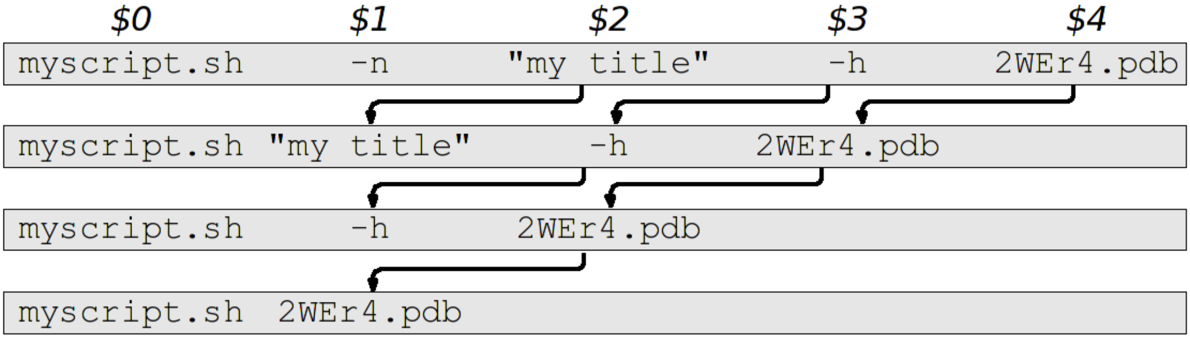
If you run the following script

```
#!/bin/sh
echo The script is $0
echo The first commandline option is $1
echo The second commandline option is $2
```

with two arguments, you’ll get the following output:

```
$ ./script.sh ABC DEF
The script is ./script.sh
The first commandline option is ABC
The second commandline option is DEF
$
```

In many cases you'll not know how many parameters are given on the commandline. In these cases you can use `shift` to loop through them. `shift` removes $1 and moves all other positional parameters one position to the right: $2 becomes $1, $3 becomes $2 etc.:


$0
$1
$2
$3
$4
myscript.sh -n "my title" -h 2WEr4.pdb
myscript.sh "my title" -h 2WEr4.pdb
myscript.sh -h 2WEr4.pdb
myscript.sh 2WEr4.pdb

With the help of "$#", "shift", "case" and the positional parameters we can now check all the commandline parameters:

```
while [ "$#" -gt 0 ]
do
  case $1 in
    -h) echo "Sorry, no help available!"  # not very helpful, is it?
        exit 1                            # exit with error
        ;;

    -v) VERBOSE=1                         # we may use $VERBOSE later
        ;;

    -f) shift
        FILE=$1                           # Aha, -f requires an
                                          # additional argument
        ;;

    *)  echo "Wrong parameter!"
        exit 1                            # exit with error
  esac
  shift
done
```

3.6 Ensuring a Sensible Exit Status

If you don't provide your own exit status, then the script will return the exit status of the last executed command (See *Reporting Success or Failure - The Exit Status* (page 22)). In many cases this might be what you want, but very often it isn't. Consider the following script which is a real example from real life and happened to me personally:

```
#!/bin/sh

[... do something that fails ...]
```

```
echo "End of the script"
```

This script will *always* succeed, as the `echo` command hardly ever fails. You will - from the exit status of the script - never be able to detect that something went wrong. Instead in such cases you should manually handle the exit codes of the commands that are run within the script.

With it's help we can keep track of the exit stati of all our important processing steps and finally return a sensible value:

```
#!/bin/sh
mystatus=0;

[... do something that might fail ...]
if [ $? -ne 0 ]
then
  mystatus=1
fi

[... do something else that might fail, too ...]
[ $? -ne 0 ] && mystatus=1         # same as above.  Do you understand
                                   # this?

echo "End of the script"
exit $mystatus
```

3.6.1 Why is the exit status important after all?

First when you use your script within other scripts, you'll probably need to be able to check, if it has succeeded. There might be other ways (e.g. checking outputfiles for certain strings, checking directly the textual output of the script etc.), but these ways are usually cumbersome and require lots of coding. Exit values are easy to check. Second: Other tools and systems might also use the exit status of your script. E.g. the cluster system uses your job's exit status to assess, if it has run successfully or not. Returning success even in case of failure will result in lots of complications in case a problem occurs. It took me several days to realize the bug above.

3.7 Tips and Tricks

3.7.1 Combining Variables with other Strings

When combining variables with other strings, then in some situations the variable name must be placed in curly brackets ("{}"):

```
$ A=Heidel
$ echo $Aberg

$ echo ${A}berg
Heidelberg
$
```

If you only want to work with part of the variable in the string, you can control this by operating on the value of the variable within the curly brackets. Use `%` to specify what you want to remove from the end of a string variable, and `#` to remove something from the beginning.

```
$ QUERYFILE=hg19_seqs.fasta
$ RESULTSFILE={$QUERYFILE}.blastn
$ echo $RESULTSFILE
hg19_seqs.fasta.blastn
$ echo ${QUERYFILE%.fasta}
hg19_seqs
$ RESULTSFILE=${QUERYFILE%.fasta}.blastn
$ echo $RESULTSFILE
hg19_seqs.blastn
$ echo ${QUERYFILE#hg19_}
seqs.fasta
$ echo ${QUERYFILE#*_}
seqs.fasta
```

Note from the above, that you specify what should be removed from the string with a pattern - meaning that you can use wildcards like `*` and `?` to make the operations flexible and perhaps to save you some typing.

3.7.2 Filenames and Paths

If possible, try to avoid any special characters (blanks, semicolons (”;”), colons (”:”), backslashes (“\”) etc.) in file and directory names. All these special characters can lead to problems in scripted processing. Instead, stick to alphanumeric characters (a-z, 0-9), dots (”.”), dashes (“-”) and underscores (“_”). Additionally sticking to lowercase characters helps avoiding mistypes and makes the automatic filename expansion easier.

3.7.3 Breaking up Long Code Lines

Code lines can become pretty long and unreadable, wrapping onto the next line etc. You can use the escape character (backslash, “\”) to break them up and enhance readability of your script. The escape character must immediately be followed by a newline (no intermediate blanks or other is allowed):

```
$ bsub -o output.log -e error.log -q clngnew -M 150000 -R "select[(mem > 15000)]" /g/so
```

becomes:

```
$ bsub -o output.log              \
       -e error.log               \
       -q clngnew                 \
       -M 150000                  \
       -R "select[(mem > 15000)]" \
    /g/software/bin/pymol -c -r pymol.pml
```

Which is way better to read and to maintain...

3.7.4 Script Debugging

sh/bash and csh/tcsh have both an option “-x” which helps debugging a script by echoing each command before executing it. This option can be set and unset during runtime with `set -x` / `set +x` (sh/bash) and `set echo` / `unset echo` (csh/tcsh).

3.7.5 Command Substitution

You can use the output of a command and assign it to a variable or use it right away as text string, by using the command substitution operator “‘” (backticks, backquotes) or “`$(...)`”. The backtick operator works in all shells, while $(...) only works in bash.

Three variants for the same (print out who you are in English text):

```
$ ME=`whoami`
$ echo I am $ME
I am fthommen
$

$ ME=$(whoami)
$ echo I am $ME
I am fthommen
$

$ echo I am `whoami`
I am fthommen
$
```

3.7.6 Create Temporary Files

You can create temporary files with mktemp. By default it will create a new file in /tmp and print its name:

```
$ mktemp
/tmp/tmp.Yaafh19370
$
```

You can take advantage of the fact that mktemp returns the name of the created file, to capture this file name and use it in your script.

3.7.7 Cleaning up Temporary Files

It is considered good practice and sometimes even important, to clean up temporary data before ending a script. A simple way - which will not cover all cases, though - could be to store all created temporary files in a variable and remove them all before exiting the script:

```
#! /bin/sh
ALL_TEMPFILES=""      # store a list of all temporary files here

TEMPFILE1=`mktemp`
ALL_TEMPFILES="$ALL_TEMPFILES $TEMPFILE1"
```

```
TEMPFILE2=`mktemp`
ALL_TEMPFILES="$ALL_TEMPFILES $TEMPFILE2"

[... process, process, process ...]

rm -f $ALL_TEMPFILES
exit
```

CHAPTER 4

Solutions to the Exercises

4.1 TAR & GZIP

1. Use gzip to compress the file P12931.txt

```
$ gzip P12931.txt
```

2. Decompress the resulting file P12931.txt.gz (revert previous command)

```
$ gunzip P12931.txt.gz
```

or

```
$ gzip -d P12931.txt.gz
```

3. Use tar to create an archive containing all fasta files in the current directory into an archive called “fastafiles.tar”

```
$ tar -c -f fastafiles.tar *.fasta
```

4. Use gzip to compress the archive “fastafiles.tar”

```
$ gzip fastafiles.tar
```

5. How can you achieve the two previous steps “using tar to create archive” and “gzip the archive” in one command?

```
$ tar -c -z -f fastafiles.tar.gz *.fasta
```

Note: Note the `-z`

6. Test (list the contents of) the compressed archive “fastafiles.tar.gz”

```
$ tar -tf fastafiles.tar.gz
```

7. Download the compressed PDB file for entry 1Y57 from rcsb.org (eg. `wget "http://www.rcsb.org/pdb/files/1Y57.pdb.gz"`) and decompress it.

```
$ wget "http://www.rcsb.org/pdb/files/1Y57.pdb.gz"
$ gunzip 1Y57.pdb.gz
```

4.2 GREP

1. Which of the DNA files ENST0* contains “TATATCTAA” as part of the sequence?

```
$ grep "TATATCTAA" ENST0*
ENST00000380152.fasta:ACGGAAGAATGTGAGAAAAATAAGCAGGACACAATTACAACTAAAAAATATATCTAA
ENST00000544455.fasta:ACGGAAGAATGTGAGAAAAATAAGCAGGACACAATTACAACTAAAAAATATATCTAA
```

2. List only the names of the DNA files ENST0* that contain “CAACAAA” as part of the sequence.

```
$ grep -l "CAACAAA" ENST0*
ENST00000380152.fasta
ENST00000544455.fasta
```

3. Considering the previous example, would you consider grep a suitable tool to perform motif searches? Why not? Try to find the pattern “CAACAAA” by manual inspection of the first three lines of each sequence.

Note: Answer: When using grep as a motif searching tool, you need to keep in mind that grep (like sed and awk) is line-oriented, meaning that by default it only searches for a given motif in a single line. In the given example, upon manual inspection you will find the given motif also in the file ENST00000530893.fasta (spanning multiple lines), which grep missed. You would need to think about how to do multi-line searches (eg. Removing line-breaks etc.)

4. Count the number of ‘ATOM’s in the file 1Y57.pdb

```
$ grep -c ATOM 1Y57.pdb
3632
```

5. Does this number agree with the annotated number of atoms? The PDB file has a comment which tells you how many atoms there are annotated in this file. This comment can be found by searching for the term “protein atoms” (use quotes and case insensitive search here!).

```
$ grep -i "protein atoms" 1Y57.pdb
REMARK   3   PROTEIN ATOMS            : 3600
```

This tells us that there are 3600 atoms annotated in this PDB file, however we initially counted 3632. This is because grep also counted any occurrence of “ATOM” within REMARKS. We can avoid this by either filtering out the remarks:

```
$ grep -v REMARK 1Y57.pdb | grep -c ATOM
3600
```

...or by telling grep to only count those lines that start with "ATOM":

```
$ grep -c ^ATOM 1Y57.pdb
3600
```

4.3 SED

1. Use sed to print only those lines that contain "version" in the files P05480.txt and P04062.txt

```
$ sed "/version/p"  P05480.txt P04062.txt
```

2. Use sed to change the text "sequence version 3" to "sequence version 4" in the files P05480.txt and P04062.txt (without actually changing the files, just printing)

```
$ sed "s/sequence version 3/sequence version 4/" P05480.txt P04062.txt
```

3. Use sed to update the text "sequence version 3" to "sequence version 4" in the files P05480.txt and P04062.txt (this time, make the changes directly in the files)

```
$ sed -i.bak "s/sequence version 3/sequence version 4/" P05480.txt P04062.txt
```

4. Replace (transliterate) all occurrences of "r" by "l" and "l" by "r" (at the same time) in the file PROTEINS.txt (so that "structural" becomes "stluctular")

```
$ sed "y/rRlL/lLrR/" PROTEINS.txt
```

4.4 AWK

1. Use awk to print only those lines that contain "version" in the files P12931.txt and P05480.txt and think about how this procedure is different to sed.

```
$ awk "/version/ {print}" P12931.txt P05480.txt
```

This is very similar to sed, you also have to use the slashes "/" to define the search pattern. However the sed notation is a little more concise...

2. For all FASTA files that begin with "P" ("P*.fasta") print only the second item of the header (split on "|") eg. for ">sp|P12931|SRC_HUMAN Proto-oncogene", print only "P12931"

```
$ awk -F"|" '/>/ {print $2}' P*.fasta
```

3. The file "P12931.csv" contains phosphorylation sites in the protein P12931. (If the file "P12931.csv" does not exist, use `wget http://phospho.elm.eu.org/byAccession/P12931.csv` to download it).

(a) Column three of this file lists the amino acid position of the phosphorylation site. You are only interested in position 17 of the protein. Try to use "grep" to filter out all these lines containing "17".

```
$ grep 17 P12931.csv
```

(b) Now use awk to show all lines containing "17".

```
$ awk "/17/ {print}" P12931.csv
```

(c) Next try show only those lines where column three equals 17 (Hint: The file is semicolon-separated...).

```
$ awk -F";" '$3==17 {print}' P12931.csv
```

(d) Finally print the PMIDs (column 6) of all lines that contain "17" in column 3.

```
$  awk -F';' '$3==17 {print $6}' P12931.csv
```

4.5 Quoting and Escaping

1. Familiarize yourself with quoting and escaping.

    1. Run the following commands to see the difference between single and double quotes when expanding variables:

    ```
    $ echo "$HOSTNAME"
    ...
    $ echo '$HOSTNAME'
    ```

    2. Next, use ssh to login to a different machine to run the same command there, again using both quoting methods:

    ```
    $ ssh pc-atcteach01 'echo $HOSTNAME'
    ...
    $ ssh pc-atcteach01 "echo $HOSTNAME"
    ```

2. Closely inspect the results; is that what you were expecting? Discuss this with your neighbour.

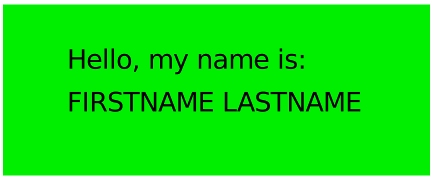
CHAPTER 5

Propositions for Scripting Exercises

Here are some ideas (not elaborated propositions) for some useful scripts which you might want to implement. Extend them to your liking. Thanks to Grischa Toedt & Chrysoula Pantzartzi for the ideas.

5.1 Replace Names in SVG

A SVG graphic is not a binary file (such as PNG or JPG image) but a textual description of objects. This allows to use commandline tools to manipulate SVG graphics. A simple example SVG like this:



looks like this:

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
    <rect x="10" y="10" height="80" width="200" style="fill: #00EE00"/>
    <text x="40" y="40">Hello, my name is:</text>
    <text x="40" y="60">FIRSTNAME LASTNAME</text>
</svg>
```

Save this text as a file called `badge.svg`, and write a script to replace ″FIRSTNAME″ and ″LASTNAME″ by a set of names from a CSV file (this file should just contain a list of names, one per line):

```
Firstname,Lastname
Holger,Dinkel
Frank,Thommen
...
```

Purpose: Search/Replace names in a SVG graphic by a list of names from a CSV file.

Usage Example:

```
$ replace_names.sh badge.svg names.csv
```

Required tools and commands: sed, for/while, etc.

Things to consider:

- Make sure to create a new SVG file for each entry in the names list.
- Consider how to name each SVG file. Just number them incrementally? Use Names?
- If it makes things easier, use two loops, one for replacing FIRSTNAME, then another one to replace LASTNAME.

5.2 General "Unpacker"

Several tools exist to pack/unpack files. Unfortunately they have different functionality and often different commandline parameters. This makes it difficult to work with them; it would be nice to have a tool which unpacks any given packed file, automatically recognizing which packing algorithm was used and just returns the unpacked file.

Purpose: Unpack a file or a number of files according to their packing or compression format (tar, gzip, zip, bzip, xz, see http://en.wikipedia.org/wiki/List_of_archive_formats for more ideas). This script can be used as a general wrapper around the various compression and packing tools with an uniform set of options

Usage Example:

```
$ ls -F
archive.tar.gz    text.txt    zuppfile
$ unpack *
archive.tar.gz is a gzip compressed tarfile ... uncompressed and unpacked
text.txt is not compressed or packed ... skipping
zuppfile is a zip compressed archive ... uncompressed and unpacked
$
```

Required tools and commands: file, tar, gzip/gunzip, zip/unzip, bzip2/bunzip2, xz, etc.

Things to consider:

- The type of a file is not necessarily deductible from its extension.
- What if the file doesn't have an extension at all?
- Depending on the tool and how the file has been archived, the unpacking/uncompressing can result in files being created in a subdirectory or directly in the current working directory. Is this what one wants/expects?
- What if the destination directory already exists?
- Some tools preserve the original archive, others don't... Maybe you can add consistency here?

Extendibility:

- Add option to keep/remove the original archives.

- Add option to unpack files into a separate directory.
- Add option to unpack files in directories named after the archive names. Check for already existing target directories!

5.3 Safety Backup Creator

One cannot have enough backups!!! However creating a backup can be tedious, so it's better to have a script which semi-automates this task.

Purpose: Create a backup copy of a directory/file in a defined location. E.g. as a safety copy/fallback before applying changes to a dataset etc.

Usage Example:

```
$ backup.sh datadir
datadir contains 12 files and is 12 MB in size
Copying datadir to /home/fthommen/safety_backups/datadir_20-MAY-2014 ... done
$ backup.sh datadir2
datadir contains 154 files and is 3 TB in size
Sorry, /home/fthommen/safety_backups/datadir2_20-MAY-2014 already exists ... aborting
$
```

Required tools and commands: cp, rsync, du, ls, date

Things to consider:

- Already existing safety backups should not be overwritten!
- Do you or don't you want to keep the full original path in some form? (dirname, basename)

Extendibility:

- Add option to pack/compress the data.

5.4 Column Chooser (advanced)

Purpose: Write a script, which takes a textfile with columnar layout and a header line and prints out only columns with the named headers of a textfile with columnar layout

Datafile:

```
NAME    FIRSTNAME       BIRTHDATE       STREET  NO
Meier   Daniel  30-MAY-1990     Meyerhofstrasse 12
Mueller Andreas 29-FEB-1960     Bahnhofstr.     1b
Schmid  Ariane  1-DEC-1990      Bahnhofstrasse  13
vonMyra Nikolaus        15-MAR-270      Dezemberstrasse 6
```

Usage Example:

```
$ columnchooser.sh FIRSTNAME NO
Daniela 12
Andreas 1b
```

```
Ariane 13
Nikolaus 6
$ columnchooser.sh CITY
Sorry, no column “CITY” found
$
```

Required tools and commands: awk, eval

Extendibility:

- Add options to define alternate column separators (awk -F).
- Add option to customize the concatenation of the printed fields.

CHAPTER 6

Appendix

6.1 Links and Further Information

6.1.1 Links

- A full 500 page book about the Linux commandline for free(!): LinuxCommand.org [1]
- Another nice introduction: "A beginner's guide to UNIX/Linux" [2]
- The "*commandline starter*" chapter of an O'Reilly book: Learning Debian GNU/Linux - Issuing Linux Commands [3]
- A nice introduction to Linux/UNIX file permissions: "chmod Tutorial" [4]
- Linux Cheatsheets [5]
- BioPieces [6] are a collection of bioinformatics tools that can be pieced together in a very easy and flexible manner to perform both simple and complex tasks.
- Google shell style guide [7]
- Useful bash one-liners for bioinformatics [8]
- Interactive explanation of your commandline: Explain Shell [9]
- Bash One-Liners Explained, Part III: All about redirections [10]
- Bash Redirections Cheat Sheet [11]
- Redirection Tutorial [12]

[1] http://linuxcommand.org/
[2] http://www.mn.uio.no/astro/english/services/it/help/basic-services/linux/guide.html
[3] http://www.oreilly.com/openbook/debian/book/ch04_01.html
[4] http://www.catcode.com/teachmod/
[5] http://www.cheat-sheets.org/#Linux
[6] http://code.google.com/p/biopieces
[7] https://code.google.com/p/google-styleguide
[8] https://github.com/stephenturner/oneliners
[9] http://www.explainshell.com
[10] http://www.catonmat.net/blog/bash-one-liners-explained-part-three
[11] http://www.catonmat.net/blog/bash-redirections-cheat-sheet
[12] http://wiki.bash-hackers.org/howto/redirection_tutorial

6.1.2 Command Line Mystery Game

CLMystery [13] is a game that you play on the commandline: There's been a murder in Terminal City, and TCPD needs your help to solve this crime *by using commandline tools only*!

To play the game, get the files from github and read the instructions:

```
wget https://github.com/veltman/clmystery/archive/master.zip
unzip master.zip
cd clmystery-master/
cat instructions
```

6.1.3 Recommended Reading: Real printed paper books

- Dietz, M., "*Praxiskurs Unix-Shell*", O'Reilly (highly recommended!, German language only)
- Herold, H., "*awk & sed*", Addison-Wesley
- Robbins, A., "*sed & awk Pocket Reference*", O'Reilly
- Robbins, A. and Beebe, N., "*Classic Shell Scripting*", O'Reilly
- Siever, E. et al., "*Linux in a Nutshell*", O'Reilly

6.1.4 Running Linux Commands in Mac

You can find the "Terminal" program in the "Utilities" folder of "Applications".

6.1.5 Running Linux Commands in Windows

Babun

The easiest way t oget a linux-like console on a Windows host is probably *babun <http://babun.github.io/>*!

Babun features the following: - Pre-configured Cygwin with a lot of addons - Command-line installer, no admin rights required - advanced package manager (like apt-get or yum) - color console - Auto update feature - "Open Babun Here" Explorer context menu entry

6.1.6 Live - CDs

A Live-CD is a complete bootable computer operating system which runs in the computer's memory, rather than loading from the hard disk drive. It allows users to experience and evaluate an operating system without installing it or making any changes to the existing operating system on the computer.

Just download an ISO-Image, burn it onto a CD/DVD and insert it into your DVD-Drive to boot your computer with Linux!

[13] https://github.com/veltman/clmystery

Fedora Live CD

This Live CD contains everything the Fedora [14] Linux operating system has to offer and it's everything you need to try out Fedora - you don't have to erase anything on your current system to try it out, and it won't put your files at risk. Take Fedora for a test drive, and if you like it, you can install Fedora directly to your hard drive straight from the Live Media desktop.

Knoppix

Knoppix [15] is an operating system based on Debian designed to be run directly from a CD / DVD or a USB flash drive, one of the first of its kind for any operating system. When starting a program, it is loaded from the removable medium and decompressed into a RAM drive. The decompression is transparent and on-the-fly. More than 1000 software packages are included on the CD edition and more than 2600 are included on the DVD edition. Up to 9 gigabytes can be stored on the DVD in compressed form.

BioKnoppix

Bioknoppix [16] is a customized distribution of Knoppix Linux Live CD. With this distribution you just boot from the CD and you have a fully functional Linux OS with open source applications targeted for the molecular biologist. Beside using RAM, Bioknoppix doesn't touch the host computer, being ideal for demonstrations, molecular biology students, workshops, etc.

Vigyaan

Vigyaan [17] is an electronic workbench for bioinformatics, computational biology and computational chemistry. It has been designed to meet the needs of both beginners and experts.

BioSlax

BioSLAX [18] is a live CD/DVD suite of bioinformatics tools that has been released by the resource team of the BioInformatics Center (BIC), National University of Singapore (NUS).

6.2 About Bio-IT

The Bio-IT Project aims to develop and strengthen the bioinformatics community at EMBL Heidelberg. It is made up of members across a range of disciplines in computational biology, in different Units and Core Facilities. The project aims to improve the standard of computational biology practised at EMBL Heidelberg, to encourage collaborations, and to provide a

[14] http://fedoraproject.org/wiki/FedoraLiveCD
[15] http://knopper.net/knoppix
[16] http://bioknoppix.hpcf.upr.edu
[17] http://www.vigyaancd.org
[18] http://www.bioslax.com

forum for discussion of issues and ideas relevant to bioinformatics here. The activities of the project include:

- the organisation and delivery of training courses such as this one
- the provision of one-to-one training and consultancy
- the organisation of social and networking events for the computational biology community
- regular meetings to discuss issues and ideas
- the development and maintenance of the Bio-IT Portal <http://bio-it.embl.de>

The Portal hosts information regarding upcoming courses and conferences/other events relevant to computational biology, resources to help with your work, and profiles of people involved in bioinformatics at EMBL. It is accessible from within the EMBL network (you must connect via VPN for off-site access).

6.2.1 Centres

EMBL Centres are 'horizontal', cross-departmental structures that promote innovative research projects across disciplines. All the EMBL Centres listed below have a strong computational component.

CBNA CSDA CBM

Biomolecular Network Analysis

The CBNA disseminates expertise, know-how and guidance in network integration and analysis throughout EMBL.

Statistical Data Analysis

The CSDA helps EMBL scientists to use adequate statistical methods for their specific technological or biological applications.

Modeling

The Centre for Biological Modeling (CBM) aims to support people to adopt mathematical modeling techniques into their everyday research.

6.3 Acknowledgements

Handouts provided by EMBL Heidelberg Photolab (Many thanks to Udo Ringeisen)

EMBL Logo © EMBL Heidelberg

Graphic of the Linux Filesystem taken from the SuSE 9.2 manual © Novell Inc.

All other graphics © Frank Thommen, EMBL Heidelberg, 2012

License: CC BY-SA 3.0

Special thanks go to contributors / helping hands (alphabetical order):

- Christian Arnold
- Jean-Karim Hériché
- Yan Ping Yuan
- Bora Uyar
- Thomas Zichner

Unix/Linux Command Reference

FOSSwire.com

File Commands

ls – directory listing

ls -al – formatted listing with hidden files

cd *dir* - change directory to *dir*

cd – change to home

pwd – show current directory

mkdir *dir* – create a directory *dir*

rm *file* – delete *file*

rm -r *dir* – delete directory *dir*

rm -f *file* – force remove *file*

rm -rf *dir* – force remove directory *dir* *

cp *file1 file2* – copy *file1* to *file2*

cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist

mv *file1 file2* – rename or move *file1* to *file2* if *file2* is an existing directory, moves *file1* into directory *file2*

ln -s *file link* – create symbolic link *link* to *file*

touch *file* – create or update *file*

cat > *file* – places standard input into *file*

more *file* – output the contents of *file*

head *file* – output the first 10 lines of *file*

tail *file* – output the last 10 lines of *file*

tail -f *file* – output the contents of *file* as it grows, starting with the last 10 lines

Process Management

ps – display your currently active processes

top – display all running processes

kill *pid* – kill process id *pid*

killall *proc* – kill all processes named *proc* *

bg – lists stopped or background jobs; resume a stopped job in the background

fg – brings the most recent job to foreground

fg *n* – brings job *n* to the foreground

File Permissions

chmod *octal file* – change the permissions of *file* to *octal*, which can be found separately for user, group, and world by adding:

- 4 – read (r)
- 2 – write (w)
- 1 – execute (x)

Examples:

chmod 777 – read, write, execute for all

chmod 755 – rwx for owner, rx for group and world

For more options, see **man chmod**.

SSH

ssh *user@host* – connect to *host* as *user*

ssh -p *port user@host* – connect to *host* on port *port* as *user*

ssh-copy-id *user@host* – add your key to *host* for *user* to enable a keyed or passwordless login

Searching

grep *pattern files* – search for *pattern* in *files*

grep -r *pattern dir* – search recursively for *pattern* in *dir*

command* | grep *pattern – search for *pattern* in the output of *command*

locate *file* – find all instances of *file*

System Info

date – show the current date and time

cal – show this month's calendar

uptime – show current uptime

w – display who is online

whoami – who you are logged in as

finger *user* – display information about *user*

uname -a – show kernel information

cat /proc/cpuinfo – cpu information

cat /proc/meminfo – memory information

man *command* – show the manual for *command*

df – show disk usage

du – show directory space usage

free – show memory and swap usage

whereis *app* – show possible locations of *app*

which *app* – show which *app* will be run by default

Compression

tar cf *file.tar files* – create a tar named *file.tar* containing *files*

tar xf *file.tar* – extract the files from *file.tar*

tar czf *file.tar.gz files* – create a tar with Gzip compression

tar xzf *file.tar.gz* – extract a tar using Gzip

tar cjf *file.tar.bz2* – create a tar with Bzip2 compression

tar xjf *file.tar.bz2* – extract a tar using Bzip2

gzip *file* – compresses *file* and renames it to *file.gz*

gzip -d *file.gz* – decompresses *file.gz* back to *file*

Network

ping *host* – ping *host* and output results

whois *domain* – get whois information for *domain*

dig *domain* – get DNS information for *domain*

dig -x *host* – reverse lookup *host*

wget *file* – download *file*

wget -c *file* – continue a stopped download

Installation

Install from source:

./configure

make

make install

dpkg -i *pkg.deb* – install a package (Debian)

rpm -Uvh *pkg.rpm* – install a package (RPM)

Shortcuts

Ctrl+C – halts the current command

Ctrl+Z – stops the current command, resume with **fg** in the foreground or **bg** in the background

Ctrl+D – log out of current session, similar to **exit**

Ctrl+W – erases one word in the current line

Ctrl+U – erases the whole line

Ctrl+R – type to bring up a recent command

!! - repeats the last command

exit – log out of current session

* use with extreme caution.