

Linux Course Documentation

Holger Dinkel, Frank Thommen & Thomas Zichner

February 23, 2015

Contents

1	Introduction to the Linux Commandline	1
1.1	Why Use the Commandline	1
1.2	General Remarks Regarding Using UNIX/Linux Systems	2
1.2.1	Absolute Paths / Relative Paths	3
1.3	General Structure of Linux Commands	3
1.4	A Journey Through the Commands	4
1.4.1	Useful Terminal Tools & Keyboard Shortcuts	4
1.4.2	Getting Help	6
1.4.3	Who am I, where am I	7
1.4.4	Moving Around	8
1.4.5	See What's Around	8
1.4.6	Organize Files and Folders	10
1.4.7	View Files	13
1.4.8	Extracting Informations from Files	15
1.4.9	Useful Filetools	16
1.4.10	Permissions	18
1.4.11	Remote access	19
1.4.12	IO and Redirections	21
1.4.13	Environment Variables	22
2	Exercises	25
2.1	Misc. file tools	25
2.2	Searching	25
2.3	Misc. terminal	25
2.4	Permissions	25
2.5	Remote access	26
2.6	IO and Redirections	26
2.7	Bioinformatics	27
3	Solutions to the Exercises	29
3.1	Misc. file tools	29
3.2	Searching	29
3.3	Misc. terminal	30
3.4	Permissions	30
3.5	Remote access	32
3.6	IO and Redirections	32
3.7	Bioinformatics	33
4	More Commandline Tools	35
4.1	Commandline Tools	35
4.1.1	GZIP	35

4.1.2	TAR	36
4.1.3	GREP	37
4.1.4	SED	37
4.1.5	AWK	39
4.2	I/O Redirection	40
4.3	Variables	42
4.3.1	Setting, Exporting and Removing Variables	42
4.3.2	Listing Variables	43
4.3.3	Variable Inheritance	43
4.4	Hints	44
5	I/O Redirection	47
6	Variables	49
6.1	Setting, Exporting and Removing Variables	49
6.2	Listing Variables	50
6.3	Variable Inheritance	50
6.3.1	Examples	50
7	Basic Shell Scripting	53
7.1	What is a Script?	53
7.2	Script Naming and Organization	53
7.3	Running a Script	53
7.3.1	Basic Structure of a Shellsript	54
7.3.2	Readability and Documentation	54
7.3.3	Anatomy of a Shellsript	55
7.3.4	Reporting Success or Failure - The Exit Status	56
7.3.5	Command Grouping and Sequences	57
7.4	Control Structures	58
7.4.1	Conditional Statements	59
7.4.2	Loops	62
7.5	Making Scripts Flexible	63
7.5.1	Configurable Scripts	63
7.5.2	Defining your own Commandline Options and Arguments	65
7.6	Ensuring a Sensible Exit Status	66
7.6.1	Why is the exit status important after all?	67
7.7	Tips and Tricks	67
7.7.1	Combining Variables with other Strings	67
7.7.2	Filenames and Paths	68
7.7.3	Breaking up Long Code Lines	68
7.7.4	Script Debugging	68
7.7.5	Command Substitution	68
7.7.6	Create Temporary Files	69
7.7.7	Cleaning up Temporary Files	69
8	Solutions to the Exercises	71
8.1	TAR & GZIP	71
8.2	GREP	72
8.3	SED	73
8.4	AWK	73
8.5	Quoting and Escaping	74

9 The Benefits of Version Control	75
10 git at a Glance	77
11 git Settings	79
11.1 setting your identity	79
11.1.1 Checking Your Settings	79
12 A Typical git Workflow	81
12.1 Creating a git Repository	81
12.2 Cloning a git Repository	81
12.3 Checking the Status	82
12.4 Adding files	82
12.5 Committing changes	83
12.6 Viewing the History	83
12.7 Pushing changes	83
12.8 Pulling changes	84
12.9 Undo local changes	84
12.10 Using centralized workflow	84
13 EMBL git server	85
14 Appendix	87
14.1 Links and Further Information	87
14.1.1 Links	87
14.1.2 Command Line Mystery Game	88
14.1.3 Real printed paper books	88
14.1.4 Live - CDs	88
14.2 About Bio-IT	89
14.2.1 Resources	90
14.2.2 Training and Outreach	90
14.2.3 Networking	90
14.2.4 Biocomputing expertise at EMBL	90
14.2.5 Centers	90
14.3 Acknowledgements	91
Index	93

Chapter 1

Introduction to the Linux Commandline

1.1 Why Use the Commandline

- It's **fast**. Productivity is a word that gets tossed around a lot by so-called power users, but the command line can really streamline your computer use, assuming you learn to use it right.
- It's **easier to get help**. The command line may not be the easiest thing to use, but it makes life a whole lot easier for people trying to help you and for yourself when looking for help, especially over the internet. Many times it's as simple as the helper posting a few commands and some instructions and the recipient copying and pasting those commands. Anyone who has spent hours listening to someone from tech support say something like, "OK, now click this, then this, then select this menu command" knows how frustrating the GUI alternative can be.
- It's nearly **universal**. There are hundreds of Linux distros out there, each with a slightly different graphical environment. Thankfully, the various distros do have one common element: the command line. There are distro-specific commands, but the bulk of commands will work on any Linux system.
- It's **powerful**. The companies behind those other operating systems try their best to stop a user from accidentally screwing up their computer. Doing this involves hiding a lot of the components and tools that could harm a computer away from novices. Linux is more of an open book, which is due in part to its prominent use of the command line.
- Many 'modern' bioinformatics tools (samtools, bamtools, ...) are written for the commandline in order to be run on clusters and to be incorporated in scripts.

1.2 General Remarks Regarding Using UNIX/Linux Systems

- **Test before run.** Anything written here has to be taken with a grain of salt. On another system - be it a different Linux distribution or another UNIXoid operating system - you might find the same command but without the support of some of the options taught here. It is even possible, that the same option has a different meaning on another system. With this in mind always make sure to test your commands (specially the “dangerous” ones which remove or modify files) when switching from one system to the other.
- **The Linux/UNIX environment.** The behaviour of many commands is influenced or controlled by the so-called “environment”. This environment is the sum of all your environment variables. Some of these environment variables will be shown towards the end of this course.
- **UPPERCASE, lowercase.** Don’t forget that everything is case-sensitive.
- **The Filesystem.** Linux filesystems start on top at the root directory (sic!) “/” which hierarchically broadens towards the ground. The separator between directories or directories and files in Linux is the slash (“/”).

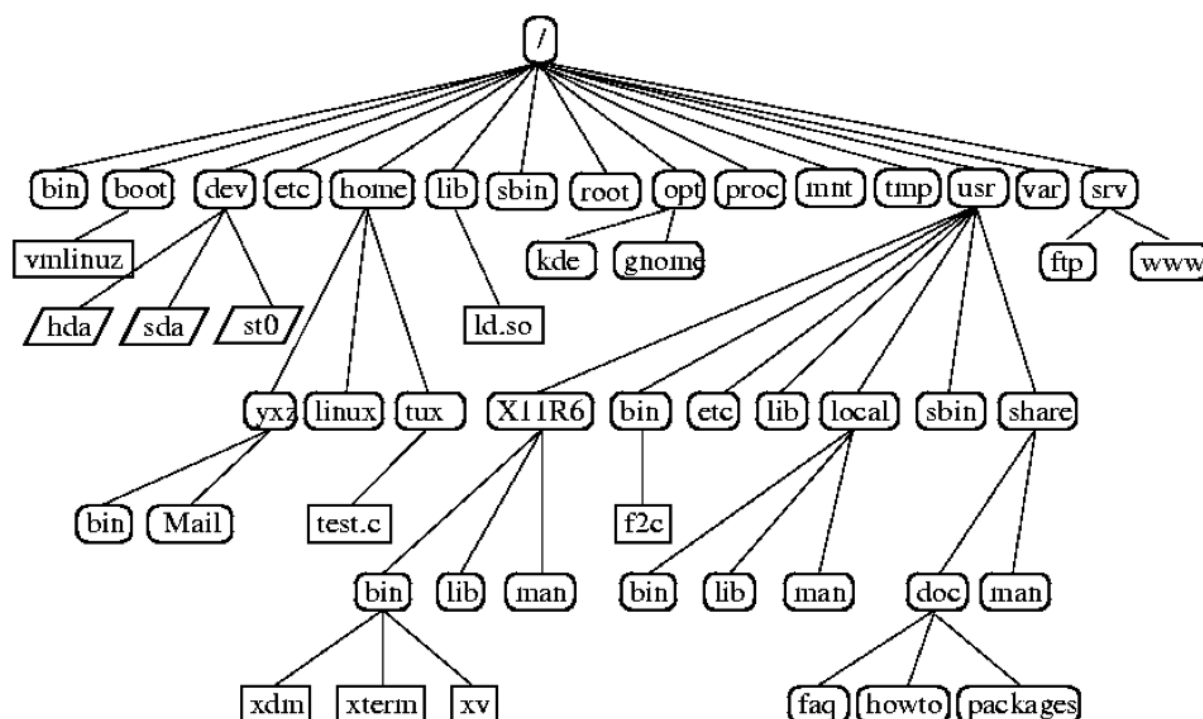


Figure 1.1: Depending on the Linux distribution you might or might not find all of above directories. Most important directories for you are `/bin` and `/usr/bin` (sometimes also `/usr/local/bin`) which contain the user software, `/home` which usually contains the users’ homedirectories and `/tmp` which can be used to store temporary data (beware: Its content is regularly removed!).

Note: The terms “directory” and “folder” are used interchangeably in this document.

1.2.1 Absolute Paths / Relative Paths

A path describes the the location of a file/folder in the filesystem: It is important to understand that there are basically two ways to describe such a path: Either by using an *absolute* pathname, or by using a *relative* pathname. The difference is that *absolute* paths always start with a “slash /”. This “slash” denotes the so called “root” of the filesystem (see below). *Relative* paths in contrast always starting with a directory name and denote the location of a file/folder *relative* to the current directory.

Note: When in doubt, it's best to use *absolute* filenames. Commands given with absolute pathname are more easily repeated later, as they can be run independent of the current working directory (unlike relative paths).

1.3 General Structure of Linux Commands

Many linux commands have options and accept arguments. Options are a set of switch-like parameters while arguments are usually free text input (such as a file-name).

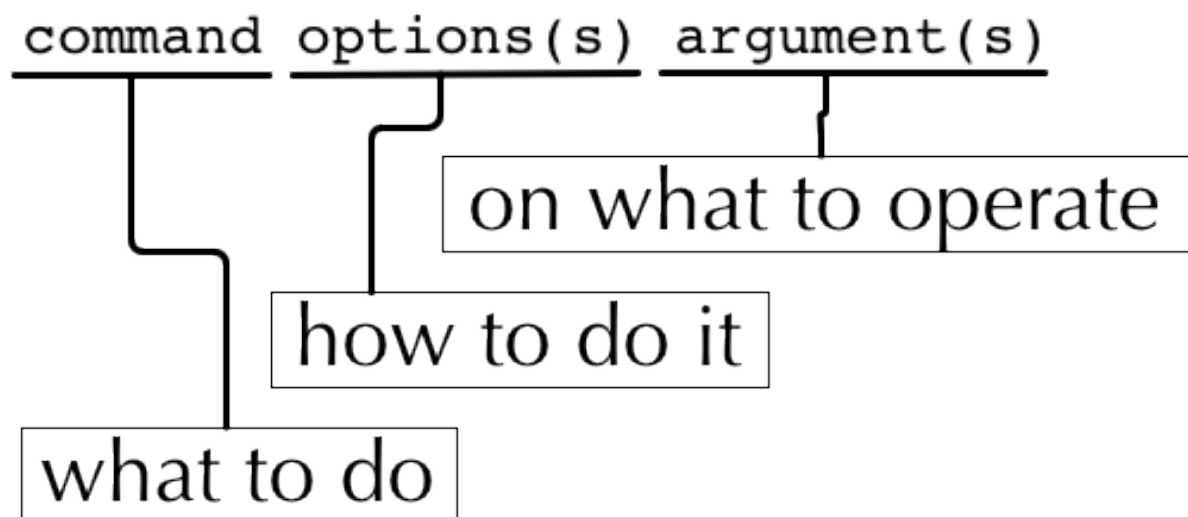


Figure 1.2: General structure of Linux commands.

For example, in the commandline `ls -l /usr/bin`, `ls` is the command, `-l` is an option and `/usr/bin` qualifies as an argument.

Commandline options (sometimes called comandline switches) commonly have one of the two following forms: The short form `-s` (just a single character) or the long form `--string`. E.g.

```
$ man -h
$ man --help
```

Short options are usually - though not always - concatenable:

```
$ ls -l -A -h
$ ls -lAh
```

Some options require an additional argument, which is added with a blank to the short form and with an equal sign to the long form:

```
$ ls -I "*.pdf"
$ ls --ignore="*.pdf"
```

Since Linux incorporates commands from different sources, options can be available in one or both forms and you'll also encounter options with no dash at all and all kinds of mixtures:

```
$ tar cf file.tar -C .. file/
$ ps auxgww
```

1.4 A Journey Through the Commands

Please note that all examples and usage instructions below are just a glimpse of what you can do and reflect our opinion on what's important and what's not. Most of these commands support many more options and different usages. Consult the manpages to find them.

Typographical conventions: Commands and examples are written in Courier. User Input is written in Courier bold and placeholders are generally written in *italic*.

1.4.1 Useful Terminal Tools & Keyboard Shortcuts

Navigating previous commands

You can use the ↑/↓ (up/down) arrow keys to navigate previously entered command and the ←/→ (left/right) keys to modify it before re-executing it.

Copying / Pasting using the mouse

On most Linux systems you can use the mouse to select text and then press the middle mouse button to paste that text at the position where your cursor is. This is especially useful for long directory or filenames.

Printing some text

To simply print some text in the console, use `echo`:

Usage: `echo`

```
$ echo "this is some text"
this is some text
$
```

It can also be used to print the content of a variable, see section [Environment Variables](#) (page 22)...

Interrupting commands

Whenever a program gets stuck or takes too long to finish, you can *interrupt* it with the shortcut CONTROL-C.

Leave the shell

To exit the shell/terminal, just type `exit` or press CONTROL-D.

clear - Clear the “screen”

Usage: `clear`

```
$ clear
$
```

In case the output of the terminal/screen gets cluttered, you can use `clear` to redraw the screen...

```
$ cat /bin/echo
$ ... (garbled output here)
$ clear
$
```

Note: If this doesn't work, you can use `reset` to perform a re-initialization of the terminal:

reset - Reset your terminal

Usage: `reset [options]`

```
$ reset
$
```

1.4.2 Getting Help

-h/--help option, no parameters

Many commands support a “help” option, either through `-h` or through `--help`. Other commands will show a help page or at least a short usage overview if you provide incorrect commandline options

man - show the manual page of a command

Usage: `man command or file`

```
$ man man
man(1)

NAME
  man - format and display the on-line manual pages

SYNOPSIS
  man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file]
  ...
$
```

For the navigation within a manpage see the [paragraph regarding less](#) (page 14).

Note: The behaviour of `man` is dependent of the `$PAGER` environment variable

apropos - list manpages containing a keyword in their description

Usage: `apropos keyword`

```
$ apropos who
...
who                (1)  - show who is logged on
who                (1)  - display who is on the system
whoami             (1)  - print effective userid
$
```

Use `apropos` to find candidates for specific tasks

/usr/share/doc/

The `/usr/share/doc/` directory in some Linux distributions contains additional documentation of installed software packages

1.4.3 Who am I, where am I

whoami - Print your username

Linux is a multi-User Operating System supporting thousands of users on the same machine. As usernames can differ between machines, it's important to know your username on any particular machine.

Usage: whoami

```
$ whoami
fthommen
$
```

hostname - Print the name of the computer

Each machine on the network has a unique name which is used to distinguish one from another.

Usage: hostname

```
$ hostname
pc-teach01
$
```

pwd - Print the current working directory

A Linux Filesystem contains countless directories with many subdirectories which makes it easy to get lost. It is good practice to check your position within the filesystem regularly.

Usage: pwd

```
$ pwd
/home/fthommen
$
```

date - Print current date and time

Usage: date

```
$ date
Tue Sep 25 19:57:50 CEST 2012
$
```

Note: The command `time` does something completely different from `date` and is *not* used to show the current time.

1.4.4 Moving Around

cd - Change the working directory

Usage: `cd [new_directory]`

```
$ pwd
/home/fthommen
$ cd /usr/bin
$ pwd
/usr/bin
$
```

Note: Using `cd` without a directory is equivalent to “`cd ~`” and changes into the users’s homedirectory

Note: Please note the difference between absolute paths (starting with “/”) and relative paths (starting with a directory name)

Special directories:

- “.”: The current working directory
- “/”: The root directory of this computer
- “..”: The parent directory of the current working directory
- “~”: Your homedirectory

```
$ pwd
/usr
$ cd /bin
$ pwd
/bin
```

```
$ pwd
/usr
$ cd
$ pwd
/home/fthommen
```

1.4.5 See What’s Around

ls - List directory contents

Usage: `ls [options] [file(s) or directory/ies]`

```
$ ls
/home/fthommen
$ ls -l aa.pdf
```

```
-rw-r--r-- 1 fthommen cmueller 0 Sep 24 10:59 aa.pdf
$
```

Useful options:

- l** Long listing with permissions, user, group and last modification date
- 1** Print listing in one column only
- a** Show all files (hidden, "." and "..")
- A** Show almost all files (hidden, but not "." and "..")
- F** Show filetypes (nothing = regular file, "/" = directory, "*" = executable file, "@" = symbolic link)
- d** Show directory information instead of directory content
- t** Sort listing by modification time (most recent on top)

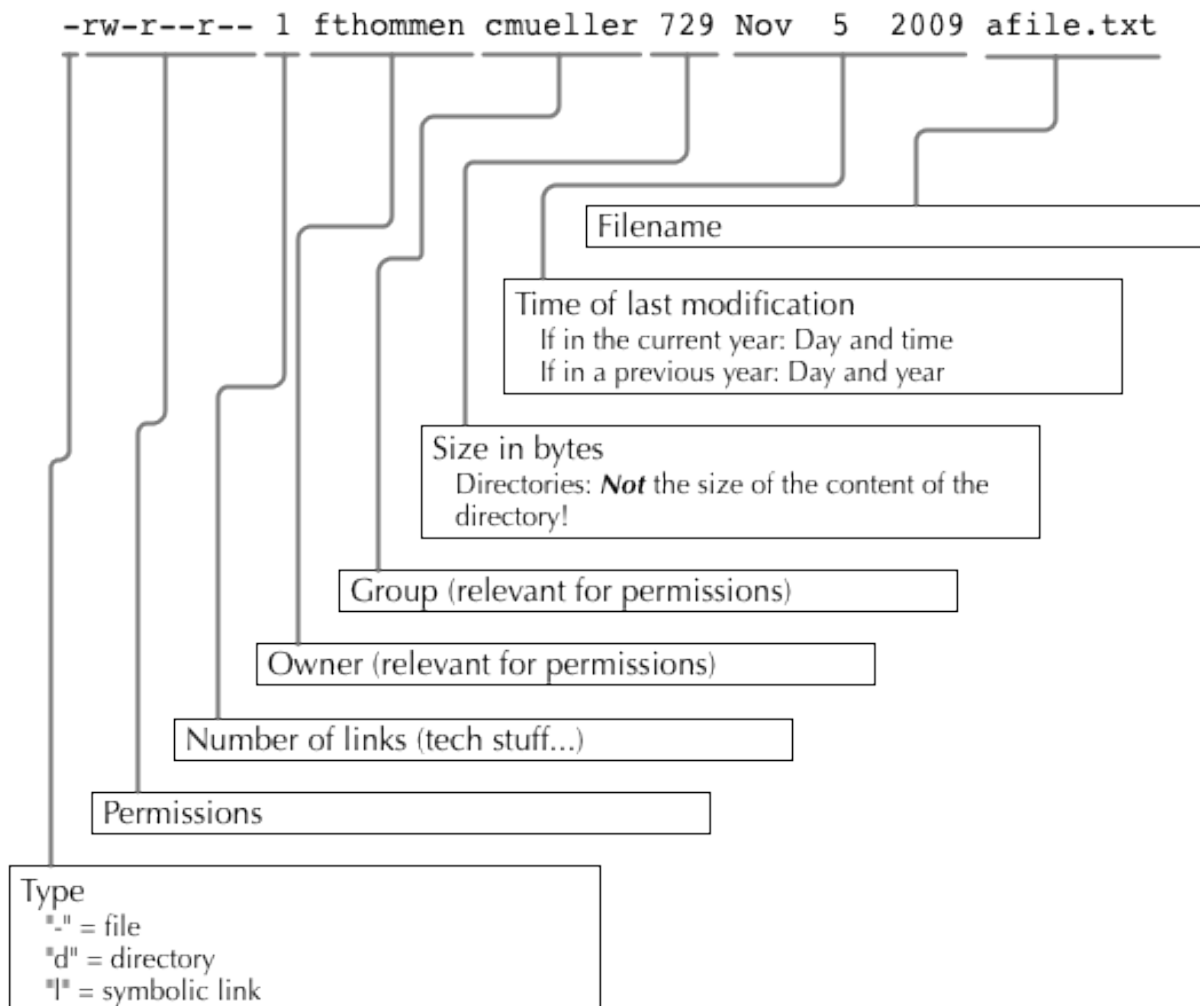


Figure 1.3: Elements of a long file listing (`ls -l`)

Digression: Shell globs

Files and folders can't only be referred to with their full name, but also with so-called "Shell Globs", which are a kind of simple pattern to address groups of files and folders. Instead of explicit names you can use the following placeholders:

- `?:` Any single character
- `*:` Any number of any character (including no character at all, but **not** matching a starting ".")
- `[...]:` One of the characters included in the brackets. Use "-" to define ranges of characters
- `{word1,word2}:` Each individual word is expanded

Examples:

- `*.pdf:` All files having the extension ".pdf"
- `? .jpg:` Jpeg file consisting of only one character
- `[0-9]*.txt:` All files starting with a number and having the extension ".txt"
- `*.???:` All files having a three-character extension
- `photo.{jpg,png}:` "photo.jpg" and "photo.png"

Note: The special directory "~" mentioned above is a shell glob, too.

1.4.6 Organize Files and Folders

cp - Copy files and folders

Usage: `cp [options] sourcefile destinationfile`

```
$ cp /usr/bin/less /tmp/backup_of_less
$
```

Useful options:

- | | |
|-----------|--|
| -r | Copy recursively |
| -i | Interactive operation, ask before overwriting an existing file |
| -p | Preserve owner, permissions and timestamp |

We copy a set of exercise files from the network share into our home directory:

```
$ cp -r /g/bio-it/courses/LSB/exercises ~/exercises
$
```


rsync - intelligently copying files and folders

Usage: `rsync [options] source target`

```
$ rsync -av /etc/ root@taperobot:/etc-backup
...
$
```

rsync allows you to copy files or folders locally or to wherever you have ssh access. You can have rsync have copying only newer files or only older files. If copy operation is interrupted, you can rerun rsync and it will only copy the missing files (in contrast to `cp` which will just copy everything again).

source and target can be local directories or have the form `user@remotehost:directory`, in which case you'll have to give your password for the remote host. This latter version will copy over the network.

Note: rsync is one of the few cases, where it effectively matters if a directory is written with an ending slash ("/") or not: If the source is a directory and ends with a slash, then the *content* of this directory will be copied into the target directory. If the source doesn't have an ending slash, then *a directory with the same name* will be created *within the target directory*

Useful option combinations:

- | | |
|------------|---|
| -av | Verbosely copies all source files which are different (different size, different age) or missing from the source. Beware: This will also copy files which are older on the source side |
| -au | Silently copies all source files which are different (different size, different age) or missing from the source. This combination will <i>not</i> overwrite newer files by older ones |

This should not copy any new files, as we previously copied these already:

```
$ rsync -av /g/bio-it/courses/LSB/exercises/ ~/exercises/
$
```

touch - Create a file or change last modification date of an existing file

Usage: `touch file(s) or directory/ies`

```
$ ls afile
ls: afile: No such file or directory
$ touch afile
$ ls afile
afile
$
```

```
$ ls -l aa.pdf
-rw-r--r-- 1 fthommen cmueller 0 Sep 24 10:59 aa.pdf
$ touch aa.pdf
$ ls -l aa.pdf
-rw-r--r-- 1 fthommen cmueller 0 Sep 25 22:01 aa.pdf
$
```

rm - Remove files and directories

Usage:

```
rm [options] file(s)
rm -r [options] directory/ies
```

```
$ ls afile
afile
$ rm afile
$ ls afile
ls: afile: No such file or directory
$
```

Useful options:

-i	Ask for confirmation of each removal
-r	Remove recursively
-f	Force the removal (no questions, no errors if a file doesn't exist)

Note: rm without the -i option will usually not ask you if you really want to remove the file or directory

mv - Move and rename files and folders

Usage:

```
mv [options] sourcefile destinationfile
mv [options] sourcefile(s) destinationdirectory
```

```
$ ls *.txt
a.txt
$ mv a.txt b.txt
$ ls *.txt
b.txt
$
```

Useful options:

-i	Ask for confirmation of each removal
-----------	--------------------------------------

Note: You cannot overwrite an existing directory by another one with `mv`

mkdir - Create a new directory

Usage: `mkdir [options] directory`

```
$ ls adir/
ls: adir/: No such file or directory
$ mkdir adir
$ ls adir
$
```

Useful options:

-p Create parent directories (when creating nested directories)

```
$ mkdir adir/bdir
mkdir: cannot create directory 'adir/bdir': No such file or directory
$ mkdir -p adir/bdir
$
```

rmdir - Remove an empty directory

Usage: `rmdir directory`

```
$ rmdir adir/
$
```

Note: If the directory is not empty, `rmdir` will complain and not remove it.

1.4.7 View Files

cat - Print files on terminal (concatenate)

Usage: `cat [options] file(s)`

```
$ cat P12931.fasta backup_of_P12931.fasta
...
$
```

Note: The command `cat` only makes sense for short files or for e.g. combining several files into one. See the redirection examples later.

head - Print first lines of a textfile

head is a program on Unix and Unix-like systems used to display the beginning of a text file or piped data.

Usage: head [options] file(s)

```
$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
$
```

Useful options:

-n num Print num lines (default is 10)

tail - Print last lines of a textfile

The tail utility displays the last few lines of a file or, by default, its standard input, to the standard output.

Usage: tail [options] file(s)

```
$ tail -n 3 /etc/passwd
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
gdm:x:42:42:./var/gdm:/sbin/nologin
sabayon:x:86:86:Sabayon user:/home/sabayon:/sbin/nologin
$
```

Useful options:

-n num Print num lines (default is 10)

-f “Follow” a file (print new lines as they are written to the file)

less - View and navigate files

Usage: less [options] file(s)

```
$ less P12931.fasta backup_of_P12931.fasta
...
$
```

Note: This is the default “pager” (a program for viewing files page by page, not an old-fashioned telecommunications device) for manpages under Linux unless you redefine your \$PAGER *environment variable* (page 22)

Navigation within less:

Key(s):	Effect:
up, down, right, left:	use cursor keys
top of document:	g
bottom of document:	G
search:	“/” + search-term
find next match:	n
find previous match:	N
quit:	q

1.4.8 Extracting Informations from Files

grep - Find lines matching a pattern in textfiles

grep is a command-line utility for searching plain-text data sets for lines matching a regular expression.

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:

-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

cut - extracting columns from textfiles

cut allows to get at individual columns in structured textfiles (for instance CSV files). By default, cut assumes the columns are TAB-separated.

Usage: cut [options] file(s)

Useful options:

- d DELIM** use DELIM instead of TAB for field delimiter. Make sure to use quotes here!
- f** select only these fields; this can either be a single field, multiple individual fields separated by comma or a range of startfield and endfield separated by dash '-'

Examples:

extract column six from the file `~/exercises/P12931.csv` (which is separated by semicolon ';'):

```
$ cut -d';' -f6 ~/exercises/P12931.csv
PMID
2136766
11804588
...
$
```

extract columns two, three, eight, nine and ten from the same file:

```
$ cut -d';' -f2,3,8-10 ~/exercises/P12931.csv
S; 12; 0.21; ; -
S; 17; 0.24; MOD_PKA_1; -
S; 17; 0.24; MOD_PKA_1; -
S; 17; 0.24; MOD_PKA_1; -
...
$
```

sort - sort a textfile

The sort utility is used to sort a textfile (alphabetically or numerically).

Usage: sort [options] file(s)

```
$ sort /etc/passwd
...
$
```

Useful options:

- f** fold lower case to upper case characters
- n** compare according to string numerical value
- b** ignore leading blanks
- r** reverse the result of comparisons

1.4.9 Useful Filetools

file - determine the filetype

Usage: file [options] file(s)

```
$ file /bin/date
/bin/date: ELF 32-bit LSB executable
$ file /bin
/bin: directory
$ file SRC_HUMAN.fasta
SRC_HUMAN.fasta: ASCII text
$
```

Note: The command `file` uses certain tests and some magic to determine the type of a file

which - find a (executable) command

Usage: `which [options] command(s)`

```
$ which date
/bin/date
$ which eclipse
/usr/bin/eclipse
$
```

find - search/find files in any given directory

Usage: `find [starting path(s)] [search filter]`

```
$ find /etc
/etc
/etc/printcap
/etc/protocols
/etc/xinetd.d
/etc/xinetd.d/ktalk
...
$
```

`find` is a powerful command with lots of possible search filters. Refer to the manpage for a complete list.

Examples:

- Find by name:

```
$ find . -name SRC_HUMAN.fasta
./SRC_HUMAN.fasta
$
```

- Find by size: (List those entries in the directory `/usr/bin` that are bigger than 500kBytes)

```
$ find /usr/bin -size +500k
/usr/bin/oparchive
/usr/bin/kiconedit
/usr/bin/opjitconv
...
$
```

- Find by type (d=directory, f=file, l=link)

```
$ find . -type d
.
./adir
$
```

1.4.10 Permissions

using `ls -l` to view entries of current directory:

```
$ ls -l
drwxr-xr-x 2 dinkel gibson 4096 Sep 17 10:46 adir
lrwxrwxrwx 1 dinkel gibson   15 Sep 17 10:45 H1.fasta -> H2.fasta
-rw-r--r-- 1 dinkel gibson  643 Sep 17 10:45 H2.fasta
$
```

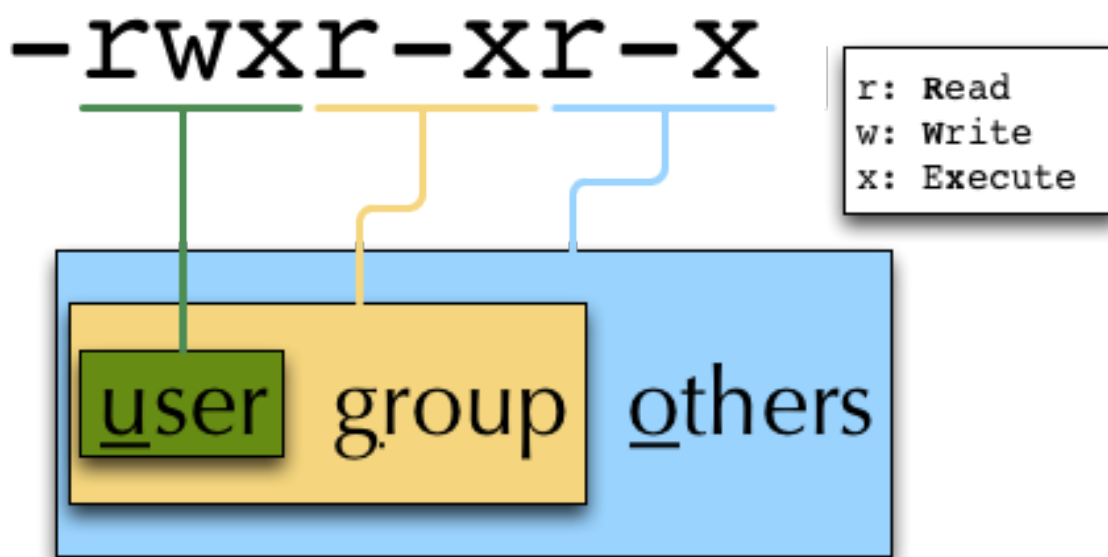


Figure 1.4: Linux file permissions

Changing Permissions

Permissions are set using the `chmod` (change mode) command.

Usage: `chmod [options] mode(s) files(s)`

```
$ ls -l adir
drwxr-xr-x 2 dinkel gibson 4096 Sep 17 10:46 adir
$ chmod u-w,o=w adir
$ ls -l adir
dr-xr-x-w- 2 dinkel gibson 4096 Sep 17 10:46 adir
$
```

The mode is composed of

Who		What		Which permission	
u:	user/owner	+:	add this permission	r:	read
g:	group	-:	remove this permission	w:	write
o:	other	=:	set exactly this permission	x:	execute
a:	all				

Add executable permission to the group:

```
$ chmod g+x file
$
```

Revoke this permission:

```
$ chmod g-x file
$
```

Allow all to read a directory:

```
$ chmod a+rx adir/
$
```

1.4.11 Remote access

To execute commands at a remote machine/server, you need to log in to this machine. This is done using the `ssh` command (secure shell). In its simplest form, it takes just the machinename as parameter (assuming the username on the local machine and remote machine are identical):

```
$ ssh remote_server
...
$
```

Note: Once logged in, use `hostname`, `whoami`, etc. to determine on which machine you are currently working and to get a feeling for your environment!

To use a different username, you can use either:

```
$ ssh -l username remote_server
...
$
```

or

```
$ ssh username@remote_server
...
$
```

When connecting to a machine for the first time, it might display a warning:

```
$ ssh submaster
The authenticity of host 'submaster (10.11.4.219)' can't be established.
RSA key fingerprint is a4:2c:c1:a6:34:49:a3:a9:b2:c3:52:f5:37:94:69:f5.
Are you sure you want to continue connecting (yes/no)?

...
$
```

Type *yes* here. If this message appears a second time, you should contact your IT specialist...

To disconnect from the remote machine, type:

```
$ exit
```

If setup correctly, you can even use *graphical tools* from the remote server on the local machine. For this to work, you need to start the ssh session with the `-X` parameter:

```
$ ssh -X remote_server
...
$
```

Copying files to and from remote computers can be done using `scp` (secure copy). The order of parameters is the same as in `cp`: first the name of the source, then the name of the destination. Either one can be the remote part.

```
$ scp localfile server:/remotefile

$ scp server:/remotefile localfile
```

An alternative username can be provided just as in `ssh`:

```
$ scp username@server:/remotefile localfile
```

1.4.12 IO and Redirections

Redirect

Redirect the output of one program into e.g. a file:

Inserting the current date into a new file:

```
$ date > file_containing_date
$
```

Warning: You can easily overwrite files by this!

Filtering lines containing the term “src” from FASTA files and inserting them into the file lines_with_src.txt:

```
$ cd ~/exercises/
$ grep -i "src" *.fasta > lines_with_src.txt
$
```

Append

Append something to a file (rather than overwriting it):

```
$ date >> file_containing_date
$
```

Pipe

Use the pipe symbol (|) to feed the output of one program into the next program. Here: use ls to show the directory contents and then use grep to only show those that contain fasta in their name:

```
$ cd ~/exercises
$ ls | grep fasta
EPSINS.fasta
FYN_HUMAN.fasta
P12931.fasta
SRC_HUMAN.fasta
$
```

1.4.13 Environment Variables

Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer.

\$HOME

Contains the location of the user's home directory. Although the current user's home directory can also be found out through the C functions `getpwuid` and `getuid`, `$HOME` is often used for convenience in various shell scripts (and other contexts).

Note: Do not change this variable unless you have a good reason and you know what you are doing!

\$PATH

`$PATH` contains a colon-separated (':') list of directories that the shell searches for commands that do not contain a slash in their name (commands with slashes are interpreted as file names to execute, and the shell attempts to execute the files directly). So if the directory `/usr/bin` is in `$PATH` (which it should), then the command `/usr/bin/less` can be accessed by simply typing `less` instead of `/usr/bin/less`. How convenient!

Warning: If you ever need to change this variable, you should always *append* to it, rather than overwriting it:

Overwriting (bad): `export PATH=/my/new/path;`

Appending (good): `export PATH=$PATH:/my/new/path`

\$PAGER

The `$PAGER` variable contains the path to the program used to list the contents of files through (such as `less` or `more`).

\$PWD

The `$PWD` variable points to the current directory. Equivalent to the output of the command `pwd` when called without arguments.

Displaying environment variables

Use `echo` to display individual variables *set* or `env` to view all at once:

```
$ echo $HOME
/localhome/teach01
$ set
```

```
...  
$ env  
...  
$
```

Setting an environment variable

Use `export` followed by the variable name and the value of the variable (separated by the equal sign) to set an environment variable:

```
$ export PAGER=/usr/bin/less  
$
```

Note: An environment variable is only valid for your current session. Once you logout of your current session, it is lost or reset.

Chapter 2

Exercises

2.1 Misc. file tools

1. Which tool can be used to determine the type of a file?
2. Use it on the following files/directories and compare the results:
 - (a) `/usr/bin/tail`
 - (b) `~`
 - (c) `~/exercises/SRC_HUMAN.fasta`

2.2 Searching

1. Which tool can be used to search for files or directories?
2. Use it to find all directories in the `~/exercises` directory
3. Search for the file `date` in the `/bin` directory
4. List those entries in the directory `/bin` that are bigger than 400kBytes

2.3 Misc. terminal

1. Which two tools can be used to redraw/empty the screen?

2.4 Permissions

1. Create a directory called `testpermissions`
2. Change your working directory to `testpermissions`
3. Create a directory called `adir`.
4. Use the command `which date` to find out where the `date` program is located.

5. Copy this date program into the directory `adir` and name it `'mydate'`.
6. Check the permissions of the copied program `'mydate'`
7. Change the permissions on `'mydate'` to remove the executable permissions.
8. Check the permissions of the program `'mydate'`
9. Change the permissions back so that the file is executable.
10. Try running it as `./mydate` or `adir/mydate` (depending on your current working directory)
11. Copy a textfile from a previous exercise into `adir`, then change the permissions, so you are not allowed to write to it. Test this by trying to read it via `cat`.
12. Then change the permissions so you can't read/cat it either. Test this by trying to read it via `cat`.
13. Change your working directory to `testpermissions`, and then try changing the permissions on the directory `adir` to non-executable.
14. What are the minimum permissions (on the directory) necessary for you to be able to execute `adir/mydate`?

2.5 Remote access

1. Login to machine `"sub-master.embl.de"` (using your own username)
2. Use `exit` to quit the remote shell (Beware to not exit your local shell)
3. Use `clear` to empty the screen after logout from the remote server
4. Use the following commands locally as well as on the remote machine to get a feeling for the different machines:
5. Copy the file `/etc/motd` from machine `sub-master.embl.de` into your local home directory (using `scp`)
6. Determine the filetype and the permissions of the file that you just copied
7. Login to your neighbor's machine (ask him for the hostname) using your own username

2.6 IO and Redirections

1. Use `date` in conjunction with the redirection to insert the current date into the (new) file `current_date` (in your homedirectory).
2. Inspect the file to make sure it contains (only a single line with) the date.
3. Use `date` again to append the current date into the same file.
4. Again, check that this file now contains two lines with dates.
5. Use `grep` to filter out lines containing the term `"TITLE"` from all PDB files in the `exercises` directory and use redirection to insert them into a new file `pdb_titles.txt`.

6. (OPTIONAL) Upon inspection of the file `pdb_titles.txt`, you see that it also contains the names of the files in which the term was found.
 - (a) Use either the `grep` manpage or `grep --help` to find out how you can suppress this behaviour.
 - (b) Redo the previous exercise such that the output file `pdb_titles.txt` only contains lines starting with `TITLE`.
7. The *third* column of the file `/etc/passwd` contains user IDs (numbers)
 - (a) Use `cut` to extract just the third column of this file (remember to specify the delimiter `:`)
 - (b) Next, use the *pipe* (page 21) symbol (`|`) and `sort` to sort this output *numerically*

2.7 Bioinformatics

Let's do some bioinformatics analysis! You can find the famous BLAST tool installed at `/g/software/bin/blastp`.

1. Typing the full path is too cumbersome, so let's append `/g/software/bin` to your `$PATH` variable and ensure that it works by calling `blastp`.
2. When you run `blastp -help`, you notice that it has a lot of options! Use redirections in conjunction with `grep` to find out which options you need to specify a *input_file* and *database_name*.
3. Now run `blastp` using the following values as options:

`database_name = /g/data/ncbi-blast/db/swissprot`

`input_file = suspect1.fasta`

Chapter 3

Solutions to the Exercises

3.1 Misc. file tools

1. Which tool can be used to determine the type of a file?

```
$ file
```

2. Use it on the following files/directories and compare the results:

(a) /usr/bin/tail

```
$ file /usr/bin/tail
/usr/bin/tail: binary executable
```

(b) ~

```
$ file ~
/home/dinkel: directory
```

(c) ~/exercises/SRC_HUMAN.fasta

```
$ file ~/exercises/SRC_HUMAN.fasta
~/exercises/SRC_HUMAN.fasta: ASCII text
```

3.2 Searching

1. Which tool can be used to search for files or directories?

```
$ find
```

2. Use it to find all directories in the ~/exercises directory

```
$ find ~/exercises -type d
```

3. Search for the file date in the /bin directory

```
$ find /bin -name date
```

4. List those entries in the directory /bin that are bigger than 400kBytes

```
$ find /bin -size +400k
```

3.3 Misc. terminal

1. Which two tools can be used to redraw/empty the screen?

```
$ clear
```

or:

```
$ reset
```

3.4 Permissions

1. Create a directory called testpermissions

```
$ mkdir testpermissions
```

2. Change your working directory to testpermissions:

```
$ cd testpermissions
```

3. Create a directory called adir.

```
$ mkdir adir
```

4. Use the command which date to find out where the date program is located.:

```
$ which date  
/bin/date
```

5. Copy this date program into the directory adir and name it 'mydate'.:

```
$ cp /bin/date adir/mydate
```

6. Check the permissions of the copied program 'mydate'

```
$ ls -lh adir/mydate  
-r-xr-xr-x  1 dinkel  staff    79K  9 Dec 13:47 mydate*
```

7. Change the permissions on 'mydate' to remove the executable permissions.:

```
$ chmod a-x adir/mydate
```

8. Check the permissions of the program 'mydate'

```
$ ls -lh adir/mydate
-r--r--r-- 1 dinkel staff 79K 9 Dec 13:47 mydate*
```

9. Try running it as ./mydate or adir/mydate (depending on your current working directory)

```
$ adir/mydate
permission denied
```

10. Change the permissions back so that the file is executable.

```
$ chmod a+x adir/mydate
```

11. Try running it as ./mydate or adir/mydate (depending on your current working directory)

```
$ adir/mydate
Mon Dec 9 13:50:12 CET 2013
```

12. Copy a textfile from a previous exercise into adir, then change the permissions, so you are not allowed to write to it. Test this by trying to read it via *cat*.

```
$ cp ~/exercises/SRC_HUMAN.fasta adir
$ chmod u-w adir/SRC_HUMAN.fasta
```

13. Then change the permissions so you can't read/cat it either. Test this by trying to read it via *cat*.

```
$ chmod u-r adir/SRC_HUMAN.fasta
```

14. Change your working directory to testpermissions, and then try changing the permissions on the directory adir to non-executable.

```
$ # no need to change directory,
$ # as we still are in the directory testpermissions
$ chmod a-x adir
```

15. What are the minimum permissions (on the directory) necessary for you to be able to execute adir/mydate?

```
$ chmod u+rx adir
```

3.5 Remote access

1. Login to machine “sub-master.embl.de” (using your own username)

```
$ ssh sub-master.embl.de -l username
```

2. Use exit to quit the remote shell (Beware to not exit your local shell)

```
$ exit
```

3. Use clear to empty the screen after logout from the remote server:

```
$ clear
```

4. Use the following commands locally as well as on the remote machine to get a feeling for the different machines:

```
A) ``hostname``  
B) ``whoami``  
C) ``ls -la ~/``
```

5. Copy the file /etc/motd from machine sub-master.embl.de into your local home directory (using scp):

```
$ scp sub-master.embl.de:/etc/motd ~/
```

6. Determine the filetype and the permissions of the file that you just copied:

```
$ file ~/motd  
~/motd: ASCII text
```

```
$ ls -l ~/motd
```

7. Login to your neighbor’s machine (ask him for the hostname) using your own username:

```
$ ssh hostname
```

3.6 IO and Redirections

1. Use date in conjunction with the redirection to insert the current date into the (new) file current_date (in your homedirectory).:

```
$ date > ~/current_date
```

2. Inspect the file to make sure it contains (only a single line with) the date.

```
$ cat ~/current_date
```

1. Use `date` again to append the current date into the same file.

```
$ date >> ~/current_date
```

2. Again, check that this file now contains two lines with dates.

```
$ cat ~/current_date
```

3. Use `grep` to filter out lines containing the term “TITLE” from all PDB files in the exercises directory and use redirection to insert them into a new file `pdb_titles.txt`:

```
$ grep TITLE ~/exercises/*.pdb > pdb_titles.txt
```

4. (OPTIONAL) Upon inspection of the file `pdb_titles.txt`, you see that it also contains the names of the files in which the term was found.

- (a) Use either the `grep` manpage or `grep --help` to find out how you can suppress this behaviour.

```
$ grep -h TITLE ~/exercises/*.pdb > pdb_titles.txt
```

- (b) Redo the previous exercise such that the output file `pdb_titles.txt` only contains lines starting with `TITLE`.

```
$ grep -h "^TITLE" ~/exercises/*.pdb > pdb_titles.txt
```

5. The *third* column of the file `/etc/passwd` contains user IDs (numbers)

- (a) Use `cut` to extract just the third column of this file (remember to specify the delimiter `:`):

```
$ cut -f3 -d':' /etc/passwd
```

- (b) Next, use the *pipe* (page 21) symbol (`|`) and `sort` to sort this output *numerically*:

```
$ cut -f3 -d':' /etc/passwd | sort -n
```

3.7 Bioinformatics

Let's do some bioinformatics analysis! You can find the famous BLAST tool installed at `/g/software/bin/blastp`.

1. Typing the full path is too cumbersome, so let's append `/g/software/bin` to your `$PATH` variable and ensure that it works by calling `blastp`.

```
$ export PATH=$PATH:/g/software/bin
$ blastp
```

2. When you run *blastp -help*, you notice that it has a lot of options! Use redirections in conjunction with *grep* to find out which options you need to specify a *input_file* and *database_name*.

```
$ blastp -help | grep input_file
[-subject subject_input_file] [-subject_loc range] [-query input_file]

$ blastp -help | grep database_name
search_strategy filename] [-task task_name] [-db database_name]
```

3. Now run *blastp* using the following values as options:

database_name = */g/data/ncbi-blast/db/swissprot*

input_file = *suspect1.fasta*

```
$ blastp -db /g/data/ncbi-blast/db/swissprot -query suspect1.fasta
```


Chapter 4

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...

4.1 Commandline Tools

4.1.1 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`

4.1.2 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:

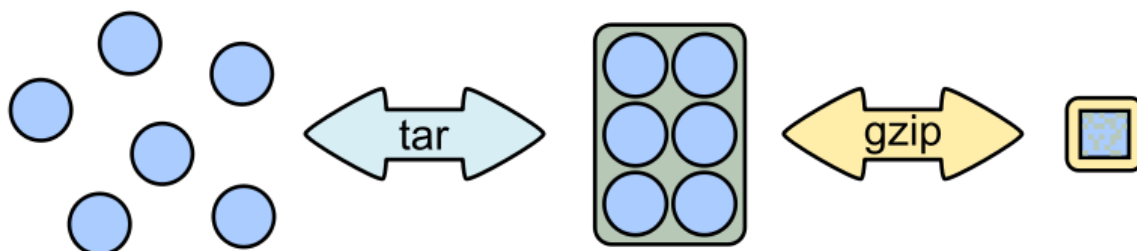


Figure 4.1: source: Th0msn80 (Wikipedia)

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
```

4.1.3 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
```

4.1.4 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____
```

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
```

4.1.5 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 catenated 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;
...
```

4.2 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

4.3 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.

4.3.1 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"
$
```

```
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:

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


```
$ 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!

4.3.2 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
...
$
```

4.3.3 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:

```
$ ./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
$
```

4.4 Hints

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.
```

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 “\”.

Chapter 5

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 <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

Chapter 6

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.

6.1 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"
$
```

```
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!

6.2 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
...
$
```

6.3 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.

6.3.1 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:

```
$ ./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
$
```


Chapter 7

Basic Shell Scripting

7.1 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!

7.2 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 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!

7.3 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 shells scripts 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
[...]  
$
```

7.3.1 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.

7.3.2 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 68)) Even if your script is very simple - document it! This helps others understand what you did, but - most important - 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 a comments use the hash sign (“#”). Everything after a “#” is ignored when executing a script.

7.3.3 Anatomy of a Shellscript

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

<pre>#!/bin/sh</pre>	Shebang line
<pre># # 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 #</pre>	Preamble with a short description, usage information, authorship etc. etc.
<pre># --- Configuration --- GREPCMD=/bin/grep DATAFILE=\$1</pre>	Configuration
<pre># --- 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</pre>	Checking prerequisites and sane environment
<pre># --- Now processing--- \$GREPCMD -q Glycine \$DATAFILE # Where is Glycine?</pre>	This is what you actually wanted to do
<pre># --- Exit --- if [\$? -eq 0] then exit 0 else exit 2 fi</pre>	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.

7.3.4 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 than 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 `$?`:

\$?: The exit status of the last run command

See [Ensuring a Sensible Exit Status](#) (page 66) about how to control the exit status of your script.

7.3.5 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 "Could not create directory a"
mkdir: cannot create directory '/bin/a': Permission denied
Could not create directory a
$
```

Example: Warn if a directory doesn't exist:

```
$ cd /etc || echo "/etc is missing!"
$ cd /nowhere >&/dev/null || echo "/nowhere does not exist"
/nowhere does not exist
$
```

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 ~/bin/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)
[... 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
$
```

7.4 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.

7.4.1 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	csch/tcsh
<pre>if condition1 then commands elif condition2 more commands [...] else even more commands fi</pre>	<pre>if (condition) then commands else if (condition2) then more commands [...] else even more commands endif</pre>

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 *csch/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 <i>file</i>	<i>file</i> exists	-e <i>file</i>
-f <i>file</i>	<i>file</i> exists and is a regular <i>file</i>	-f <i>file</i>
-d <i>file</i>	<i>file</i> exists and is a directory	-d <i>file</i>
-r <i>file</i>	<i>file</i> exists and is readable	-r <i>file</i>
-w <i>file</i>	<i>file</i> exists and is writeable	-w <i>file</i>
-x <i>file</i>	<i>file</i> exists and is executable	-x <i>file</i>
-s <i>file</i>	<i>file</i> exists and has a size > 0	
	<i>file</i> exists and has zero size	-z <i>file</i>
	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 is not equal to n2	n1 != n2
	Combination of conditions	
! <i>cond</i>	True if condition <i>cond</i> is not true	! <i>cond</i>
<i>cond1</i> -a <i>cond2</i>	True if conditions <i>cond1</i> and <i>cond2</i> are both true	<i>cond1</i> && <i>cond2</i>
<i>cond1</i> -o <i>cond2</i>	True if conditions <i>cond1</i> or <i>cond2</i> is true	<i>cond1</i> <i>cond2</i>

Examples: Test for the existence of /etc/passwd:

```
if [ -e /etc/passwd ]
then
    echo /etc/passwd exists
else
    echo /etc/passwd does NOT exist
fi
```

or:

```
if test -e /etc/passwd
then
    echo /etc/passwd exists
else
    echo /etc/passwd does NOT exist
```

```
fi
```

Note: Bash supports an additional 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
<pre>case variable in pattern1) commands ;; pattern2) commands ;; *) commands ;; esac</pre>	<pre>switch (variable) case pattern1: commands breaksw case pattern2: commands breaksw default: commands endsw</pre>

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

```
esac
```

7.4.2 Loops

for / foreach

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

sh/bash	cshtcsh
<pre>for variable in list do commands done</pre>	<pre>foreach variable (list) commands end</pre>

list is a list of strings, separated by whitespaces

Examples: List all files in /tmp in a bulleted list:

```
for FILE in /tmp/*
do
    echo " * $FILE"
done
or
for FILE in `ls /tmp`
do
    echo " * $FILE"
done
```

while / until

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

sh/bash	cshtcsh
<pre>while condition do commands done until condition do commands done</pre>	<pre>while (condition) commands end</pre>

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

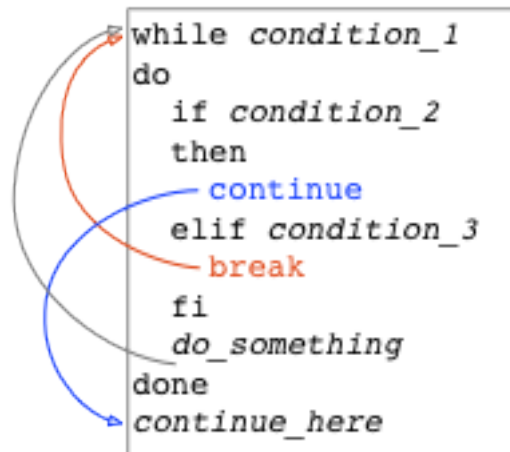





Figure 7.1: Loop control

Symbol	
	Regular loop cycle
	break due to <i>condition_2</i>
	continue due to <i>condition_3</i>

7.5 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.

7.5.1 Configurable Scripts

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) but assigned to variables:

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 an other 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, less 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 an other 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
```

7.5.2 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 to 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:

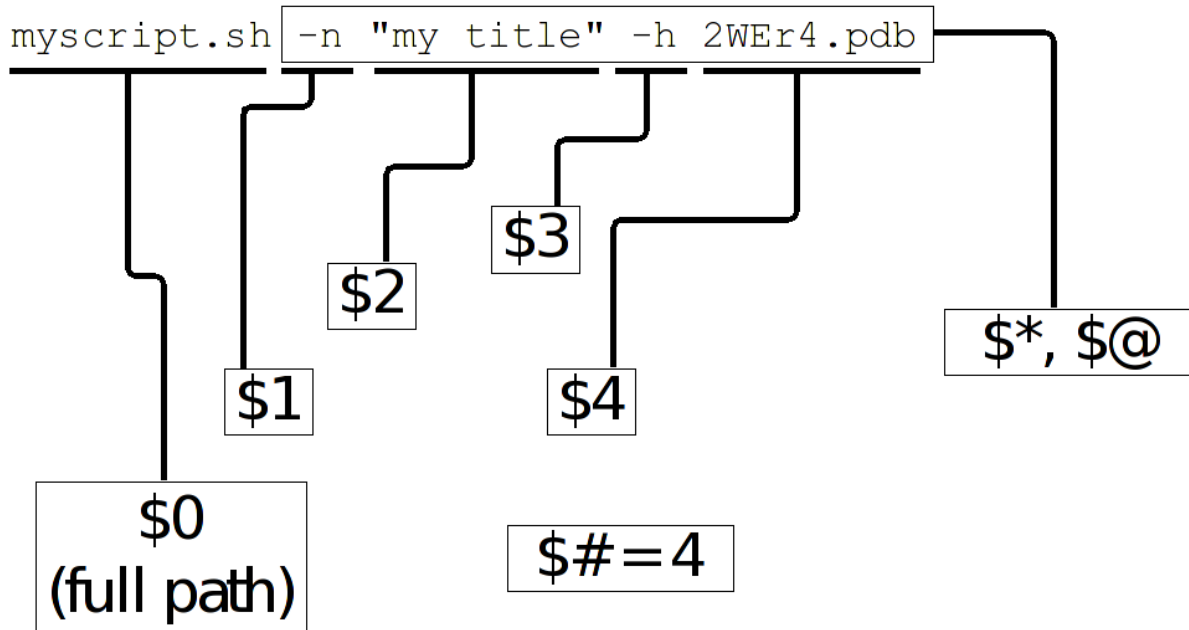
`$0`: path to the script. Either the path as you specified it or the full path if the script was executed through `$PATH`

`$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



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

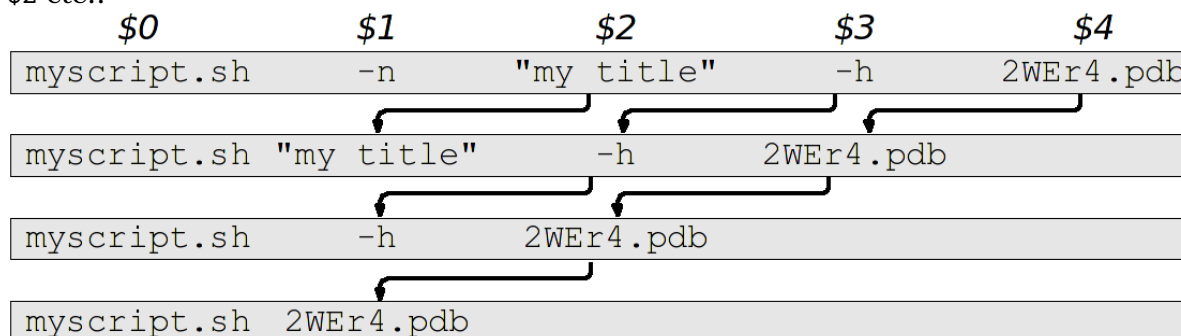
If you run the script

```
#!/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.:



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
```

7.6 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 56)). 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
```

7.6.1 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.

7.7 Tips and Tricks

7.7.1 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
$
```

7.7.2 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 lower-case characters helps avoiding mistypes and makes the automatic filename expansion easier.

7.7.3 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-1.4 -r -p < pymol.pml
```

Which is way better to read and to maintain

7.7.4 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).

7.7.5 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
$
```

7.7.6 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
$
```

7.7.7 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 8

Solutions to the Exercises

8.1 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
```

8.2 GREP

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

```
$ grep "TATATCTAA" ENST0*

ENST00000380152.fasta:ACGGAAGAATGTGAGAAAAATAAGCAGGACACAATTACAACATAAAAAATATATCTAA
ENST00000544455.fasta:ACGGAAGAATGTGAGAAAAATAAGCAGGACACAATTACAACATAAAAAATATATCTAA
```

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

```
$ grep "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 two 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, 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 ATOMs in the file 1Y57.pdb.
5. Does this number agree with the annotated number of atoms (Search the REMARKs for “protein atoms”)

```
$ grep -c "ATOM" 1Y57.pdb
3632
$ grep -i "protein atoms" 1Y57.pdb
REMARK      3      PROTEIN ATOMS                : 3600
```

This means there are 3600 atoms annotated in this PDB file, however we 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
```

8.3 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
```

8.4 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
```

8.5 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'
```

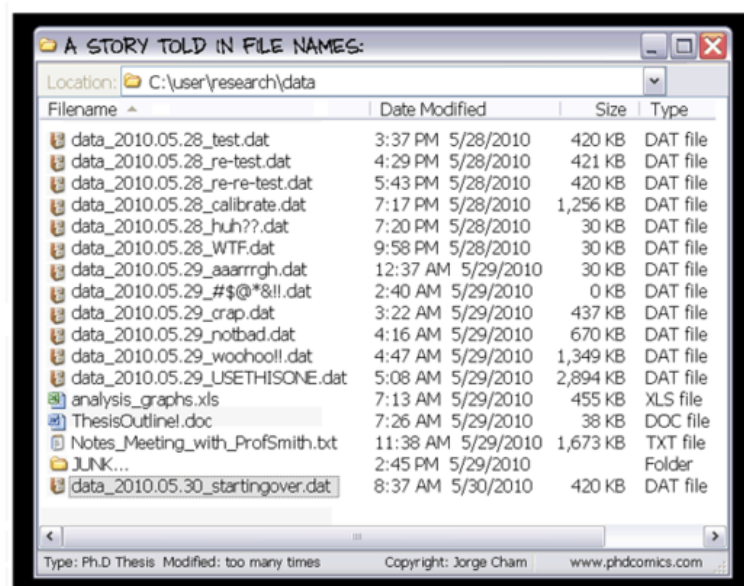
2. Next, use ssh to login to a different machine to run the same command there, again using both quoting methods:

```
$ 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 9

The Benefits of Version Control



Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. The benefits are at hand:

- **Track incremental backups and recover:** Every document can be backed up automatically and restored at a second's notice.
- **Track every change:** Every infinitesimal change can be recorded and can be used to revert a file to an earlier state.
- **Track writing experiments:** Writing experiments can be sandboxed to copies while keeping the main file intact.
- **Track co-authoring and collaboration:** Teams can work independently on their own files, but merge them into a latest common revision.
- **Track individual contributions:** Good VCS systems tag changes with authors who make them.

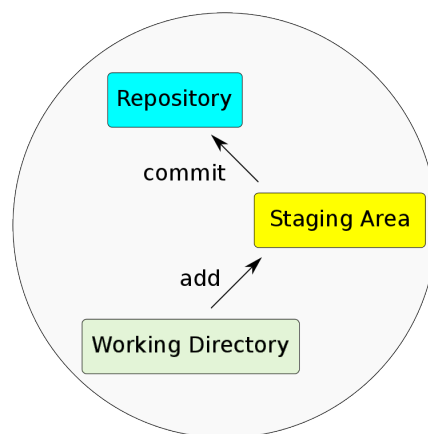


Figure 9.1: Files are *added* from the *working directory*, which always holds the current version of your files, to the *staging area*. *Staged* files will be stored into the repository in the next *commit*. The repository itself contains all previous versions of all files ever committed.

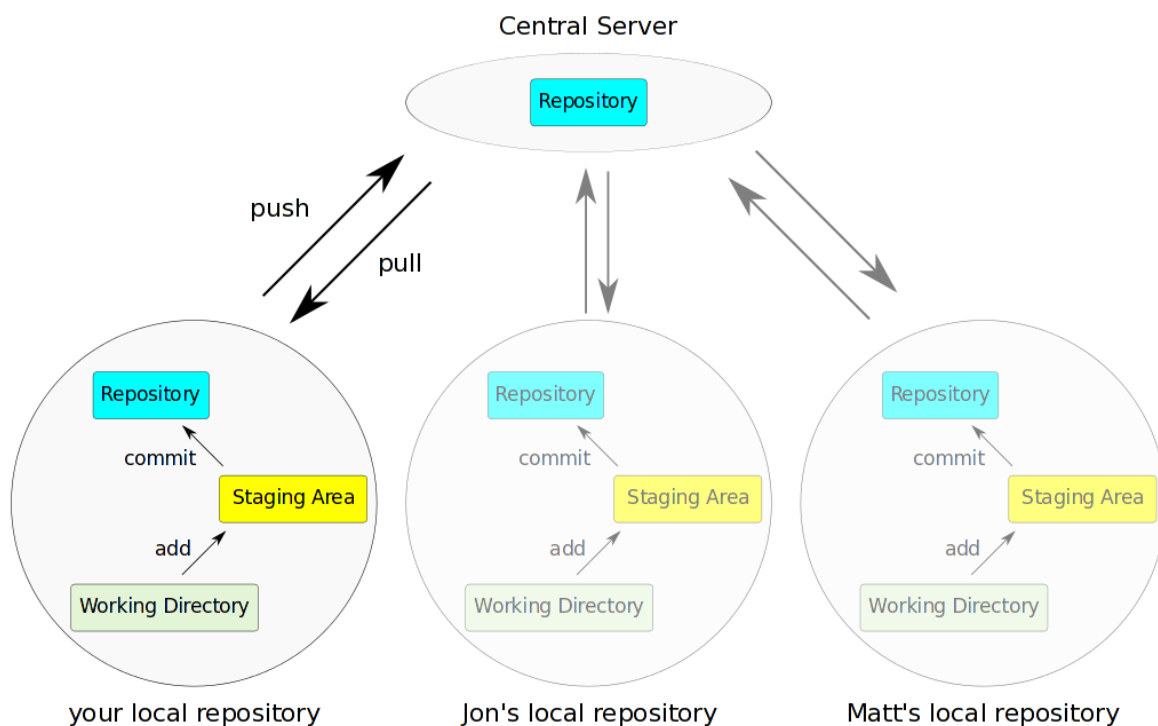


Figure 9.2: Distributed Workflow using a centralized repository. Here, three local copies of one central repository allow you, Jon and Matt to work on the same files and sync files between each other using the central server.

Chapter 10

git at a Glance

The git tool has many subcommands that can be invoked like *git <subcommand>* for instance *git status* to get the status of a repository.

The most important ones (and hence the ones we'll be focusing on) are:

init: initialize a repository

clone: clone a repository

status: get information about a repository

log: view the history and commit messages of the repository

add: add a file to the staging area

commit: commit your changes to your **local** repository

push: push changes to a **remote** repository

pull: pull changes from a **remote** repository

checkout: retrieve a specific version of a file

you can read more about each command by invoking the help:

```
git commit --help
git help commit
```


Chapter 11

git Settings

11.1 setting your identity

Before we start, we should set the user name and e-mail address. This is important because every git commit uses this information and it's also incredibly useful when looking at the history and commit log:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@embl.de
```

Other useful settings include your favorite editor as well as difftool:

```
git config --global core.editor vim
git config --global merge.tool meld
```

11.1.1 Checking Your Settings

You can use the *git config -list* command to list all your settings:

```
git config --list
user.name="John Doe"
user.email=johndoe@embl.de
core.editor=vim
merge.tool=meld
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```


Chapter 12

A Typical git Workflow

12.1 Creating a git Repository

Turning an existing directory into a git repository is as simple as changing into that directory and invoking *git init*. Here we first create an empty directory called *new_repository* and create a repository in there:

```
mkdir new_repository
cd new_repository
git init
```

Note: As a result, there should be a directory called *.git* in this directory...

12.2 Cloning a git Repository

Instead of creating a new directory, we can clone a repository. That *origin* repository can reside in a different folder on our computer, on a remote machine, or on a dedicated git server:

Local directory:

```
git clone ../other_directory
```

Remote directory:

```
git clone ssh://user@server/project.git
```

Remote git server:

```
git clone git@server:user/project
git clone git@git.embl.de:dinkel/linuxcommandline
```

12.3 Checking the Status

If you don't know in which state the current repository is in, it's always a good idea to check:

```
git status

# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

12.4 Adding files

First, we'll create a new file:

```
echo "First entry in first file!" > file1.txt

git status

# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       file1.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Now we'll add this file to the so called *staging area*:

```
git add file1.txt

git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   file1.txt
#
```

This tells us that the *file1.txt* has been added and can be committed to the repository.

12.5 Committing changes

It might be a bit confusing at first to find out that *git add* does **not** add a file to the repository. You need to *commit* the file/changes to do that:

```
git commit -m "message describing the changes you made"
```

Note: You **MUST** provide a commit message! *git* will ignore your attempt to commit if the message is empty. If you do not provide the *-m* parameter, *git* will open an editor in which you should write your commit message (can be multiple lines of text). Once you save/quit your editor, *git* will continue to commit...

After successfully committing, we can check the status again:

```
git status

# On branch master
nothing to commit, working directory clean
```

12.6 Viewing the History

You can use *git log* to view the history of a repository. All previous commits including details such as Name & Email-address of the committer, Date & Time of the commit as well as the actual commit message are shown:

```
git log

commit <some hash value identifying this commit>
Author: <your name and email address>
Date:   <the actual date of the commit>

message describing the changes you made
```

12.7 Pushing changes

If we had cloned this repository from a remote location, we probably want our changes to be propagated to that repository as well. To push all committed changes, simply type:

```
git push
```

Note: *git* “knows” from which location you had cloned this repository and will try to push to exactly that location (using the protocol you used to clone: *ssh*, *git*, etc)...

Warning: If you get a warning message, you probably ‘just’ need to pull others changes before you are allowed to push your own...

12.8 Pulling changes

To update your local repository with changes from others, you need to *pull* these changes. In a centralized workflow you actually **must** pull changes that other people have contributed, before you can submit your own.

```
git pull
```

Warning: Ideally, changes from others don’t conflict with yours, but whenever someone else has edited the same lines in the same files as you, you will receive an error message about a **merge conflict**. You will need to resolve this conflict manually, then add each resolved file (*git add*) and commit.

12.9 Undo local changes

One of the great features of using version control is that you can revert (undo) changes easily. If you want to undo all changes in a local file, you simply checkout the latest version of this file:

```
git checkout -- <filename>
```

Warning: You will loose all changes you made since the last commit!

12.10 Using centralized workflow

When you want to use one central repository, to which everybody can push/pull, you should initialize this repo like so: *git init -bare*. Basically what this does is create a repository which all the files from the *.git* directory in the working directory. This also means that you should never add/edit/delete files in this directory. Rather clone this directory in another folder/computer, edit files there and commit/push (see below)...

Chapter 13

EMBL git server

As part of the Bio-IT initiative, EMBL provides a central git server which can be used as a centralized resource to share and exchange data/code with collaborators:

<http://git.embl.de/>

The following rules apply:

- Repositories on the EMBL Git server are only granted to EMBL staff members.
- External users can be added as cooperators on a project, but the projects themselves have to be lead by someone with an active EMBL contract.
- Should the project leader leave EMBL, then the project has to be transferred to someone else or the complete repository will be removed.
- Repositories are always installed as sub-repositories of the project leader/repository responsible.
- By default, repositories are installed with only basic access permissions for the repository owner. He/she is then in charge of setting appropriate access permissions as described on the [Howto](#) page.

Basically, to use this server, you need to provide your full name, your EMBL email address and username, the name and a short description of the repository/project, along with your SSH public key to the admin and he will set things up so you are able to access your repository:

```
git clone git@git.embl.de:your_username/your_repository
```

Note: It's important to mention that the username for accessing the git.embl.de server is always **git**, not **your** username!

An SSH key can be generated using the command *ssh-keygen* (Windows users might want to use [putty](#)) like so:

```
ssh-keygen

Generating public/private rsa key pair.
Enter file in which to save the key (/home/username/.ssh/id_rsa):
Created directory '/home/username/.ssh'.
```

```
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /home/username/.ssh/id_rsa.  
Your public key has been saved in /home/username/.ssh/id_rsa.pub.  
The key fingerprint is: 2d:14:f5:d8:... username@hostname
```

This creates two files, in this case `/home/username/.ssh/id_rsa` and `/home/username/.ssh/id_rsa.pub`. The former is your **private** key and should **never** be handed out to anybody, while the latter one (ending in `.pub`) should be distributed to any server on which you intend to use it...

Chapter 14

Appendix

14.1 Links and Further Information

14.1.1 Links

- A full 500 page book about the Linux commandline for free(!): [LinuxCommand.org](http://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](http://oreilly.com/openbook/debian/book/ch04_01.html) ³
- A nice introduction to Linux/UNIX file permissions: “[chmod Tutorial](http://www.catcode.com/teachmod/)” ⁴
- [Linux Cheatsheets](http://www.cheat-sheets.org/#linux) ⁵
- For the technically interested: [Linux Filesystem Hierarchy Standard](http://www.pathname.com/fhs/) ⁶ and [Linux Standard Base](http://www.linuxfoundation.org/collaborate/workgroups/lsb) ⁷
- [Unix commands applied to bioinformatics](http://rous.mit.edu/index.php/Unix_commands_applied_to_bioinformatics) ⁸
- [BioPieces](http://code.google.com/p/biopieces) ⁹ 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](https://code.google.com/p/google-styleguide) ¹⁰
- [Useful bash one-liners for bioinformatics](https://github.com/stephenturner/oneliners) ¹¹
- Interactive explanation of your commandline: [Explain Shell](http://www.explainshell.com) ¹²

¹ <http://linuxcommand.org/>

² <http://www.mn.uio.no/astro/english/services/it/help/basic-services/linux/guide.html>

³ http://oreilly.com/openbook/debian/book/ch04_01.html

⁴ <http://www.catcode.com/teachmod/>

⁵ <http://www.cheat-sheets.org/#linux>

⁶ <http://www.pathname.com/fhs/>

⁷ <http://www.linuxfoundation.org/collaborate/workgroups/lsb>

⁸ http://rous.mit.edu/index.php/Unix_commands_applied_to_bioinformatics

⁹ <http://code.google.com/p/biopieces>

¹⁰ <https://code.google.com/p/google-styleguide>

¹¹ <https://github.com/stephenturner/oneliners>

¹² <http://www.explainshell.com>

14.1.2 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
```

14.1.3 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

14.1.4 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!

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.

¹³ <https://github.com/veltman/clmystery>

¹⁴ <http://fedoraproject.org/wiki/FedoraLiveCD>

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).

14.2 About Bio-IT

Bio-IT is a community project aiming to develop and strengthen the bioinformatics user community at EMBL Heidelberg. It is made up of members across the different EMBL Heidelberg units and core facilities. The project works to achieve these aims, firstly, by providing a forum for discussing and sharing information and ideas on computational biology and bioinformatics, focused on the **Bio-IT portal**. Secondly, we organise and participate in a range of different networking and social activities aiming to strengthen ties across the community.

¹⁵ <http://knopper.net/knoppix>

¹⁶ <http://bioknoppix.hpcf.upr.edu>

¹⁷ <http://www.vigyaan.cd.org>

¹⁸ <http://www.bioslax.com>

14.2.1 Resources

A list of biocomputing-related resources associated with the project, including, for example help provided for installing software on Linux computers at EMBL, instructions on using the Git versions control system server provided by EMBL, and various other kinds of information.

14.2.2 Training and Outreach

Bio-IT provides information on events (courses and conferences), both internal to EMBL and organised elsewhere by other organisations, that are related to biocomputing and bioinformatics.

14.2.3 Networking

Several different kinds of networking events for the Bio-IT community are being organised, including beer sessions for the EMBL community, and within-Heidelberg events for the larger Heidelberg biocomputing community.

14.2.4 Biocomputing expertise at EMBL

You can use the Bio-IT portal to search for people working at EMBL who have experience working with data or tools you might be interested in.

If you've not yet got a page up on the portal describing your own expertise, please do so. If you need any help doing this, you can read about this in the portal's FAQ section, or get in touch with one of the site administrators.

14.2.5 Centers

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.



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.

Molecular and Cellular Imaging

The [CMCI](#) makes your life in image processing/analysis easier and more fun.

Modeling

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

14.3 Acknowledgements

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

EMBL Logo © [EMBL Heidelberg](#)

Graphic of the [Linux Filesystem](#) (page 2) 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
- Jan Ping Yuan
- Bora Uyar
- Thomas Zichner

Index

Symbols

[59
#	55
\$?	56
\$HOME	22
\$PAGER	22
\$PATH	22
\$PWD	22
	21
>>	21
>	21

A

append	21
apropos	6
awk	39

B

backquote	68
backtick	68
break	63
breaksw	61

C

case	61, 66
cat	13, 32
cd	8, 30
chmod	18, 31
clear	5, 30, 32
clone	81
command	3
general structure	3
interrupt	5
switches	3
command substitution	68
comment	55
continue	63
cp	10, 30
cut	15, 33

D

date	7, 32
disconnect	20

E

echo	4, 22
elif	61
env	22, 43, 50
environment variables	22
display	22
set	23
escape character	68
exit	5, 20, 32
exit status	56, 66
export	23

F

file	16, 29, 32
append	21
overwrite	21
find	17, 29
for	62
foreach	62

G

grep	15, 21, 33, 37
gzip	35, 71

H

hash sign	55
head	14
hostname	7, 19, 32

I

if - then - else	59
init	81
interpreter	54

L

less	14, 22
ls	8, 30, 32

M

man	6
mkdir	13, 30
more	22
mv	12

O

options 3

P

pattern 61

Permissions 18

pipe 21

positional parameters 65

pwd 7

R

redirect 21

Remote access 19

reset 5, 30

rm 12

rmdir 13

rsync 11

S

scp 20, 32

secure copy 20

secure shell 19

sed 37

set 23, 43, 50, 68

shebang line 54

shift 66

sort 16

special variables: \$? 56

ssh 19, 32

T

tail 14

tar 36, 71

test 59

time 7

touch 11

U

unset 68

until 62, 63

V

variables

 environment variables 42, 49

 shell variables 42, 49

W

which 17, 30

while 62, 63

whoami 7, 19, 32