

Linux Course Documentation

Holger Dinkel, Frank Thommen & Toby Hodges

December 09, 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 Commandland	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	14
1.4.8	Extracting Informations from Files	15
1.4.9	Useful Filetools	18
1.4.10	Permissions	19
1.4.11	Remote access	20
1.4.12	IO and Redirections	22
1.4.13	Environment Variables	23
2	Exercises	25
2.1	Misc. file tools	25
2.2	Copying / Deleting Files & Folders	25
2.3	View Files	25
2.4	Searching	26
2.5	Misc. terminal	26
2.6	Permissions	26
2.7	Remote access	26
2.8	IO and Redirections	27
2.9	Putting it all together	27
2.10	Bioinformatics	28
3	Solutions to the Exercises	29
3.1	Misc. file tools	29
3.2	Copying / Deleting Files & Folders	29
3.3	View Files	30
3.4	Searching	31
3.5	Misc. terminal	31
3.6	Permissions	31
3.7	Remote access	33
3.8	IO and Redirections	34

3.9	Putting it all together	35
3.10	Bioinformatics	35
4	More Commandline Tools	37
4.1	Commandline Tools	37
4.1.1	GZIP	37
4.1.2	TAR	38
4.1.3	GREP	39
4.1.4	SED	39
4.1.5	AWK	41
4.2	I/O Redirection	42
4.3	Variables	44
4.3.1	Setting, Exporting and Removing Variables	44
4.3.2	Listing Variables	45
4.3.3	Variable Inheritance	45
4.4	Hints	46
4.4.1	Quoting	46
4.4.2	Keyboard Shortcuts	47
5	I/O Redirection	51
6	Variables	53
6.1	Setting, Exporting and Removing Variables	53
6.2	Listing Variables	54
6.3	Variable Inheritance	54
6.3.1	Examples	54
7	Commandline Exercises	57
7.1	TAR & GZIP	57
7.2	GREP	57
7.3	SED	58
7.4	AWK	58
7.5	Quoting and Escaping	58
8	Basic Shell Scripting	61
8.1	What is a Script?	61
8.2	Script Naming and Organization	61
8.3	Running a Script	61
8.3.1	Basic Structure of a Shellscript	62
8.3.2	Readability and Documentation	62
8.3.3	Anatomy of a Shellscript	63
8.3.4	Reporting Success or Failure - The Exit Status	64
8.3.5	Command Grouping and Sequences	65
8.4	Control Structures	67
8.4.1	Conditional Statements	67
8.4.2	Loops	71
8.5	Making Scripts Flexible	72
8.5.1	Configurable Scripts	73
8.5.2	Defining your own Commandline Options and Arguments	74
8.6	Ensuring a Sensible Exit Status	76
8.6.1	Why is the exit status important after all?	76
8.7	Tips and Tricks	77

8.7.1	Combining Variables with other Strings	77
8.7.2	Filenames and Paths	77
8.7.3	Breaking up Long Code Lines	77
8.7.4	Script Debugging	77
8.7.5	Command Substitution	78
8.7.6	Create Temporary Files	78
8.7.7	Cleaning up Temporary Files	78
9	Solutions to the Exercises	81
9.1	TAR & GZIP	81
9.2	GREP	82
9.3	SED	83
9.4	AWK	83
9.5	Quoting and Escaping	84
10	Propositions for Scripting Exercises	85
10.1	Replace Names in SVG	85
10.2	General “Unpacker”	86
10.3	Safety Backup Creator	87
10.4	Column Chooser (advanced)	87
11	The Benefits of Version Control	89
11.1	git at a Glance	90
11.1.1	git commands	90
11.1.2	git concepts	90
11.2	git settings	90
11.2.1	Setting your identity	90
11.2.2	Checking Your Settings	91
12 A	Typical git Workflow	93
12.1	Creating a git Repository	95
12.2	Cloning a git Repository	95
12.3	Checking the Status	96
12.4	Adding files	96
12.5	Committing changes	97
12.6	Viewing the History	97
12.6.1	Exercise	98
12.7	Pushing changes	98
12.7.1	Creating a second clone	99
12.8	Pulling changes	99
12.9	Solving conflicts	99
12.9.1	Manually merging a conflict	100
12.10	Undo local changes	101
13	Appendix	103
13.1	Links and Further Information	103
13.1.1	Links	103
13.1.2	Command Line Mystery Game	104
13.1.3	Recommended Reading: Real printed paper books	104
13.1.4	Running Linux Commands in Mac	104
13.1.5	Running Linux Commands in Windows	104
13.1.6	Live - CDs	104

13.2 About Bio-IT	106
13.2.1 Centres	106
13.3 Acknowledgements	107
Index	109

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 correctly.
- 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 distributions 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 (“/”).

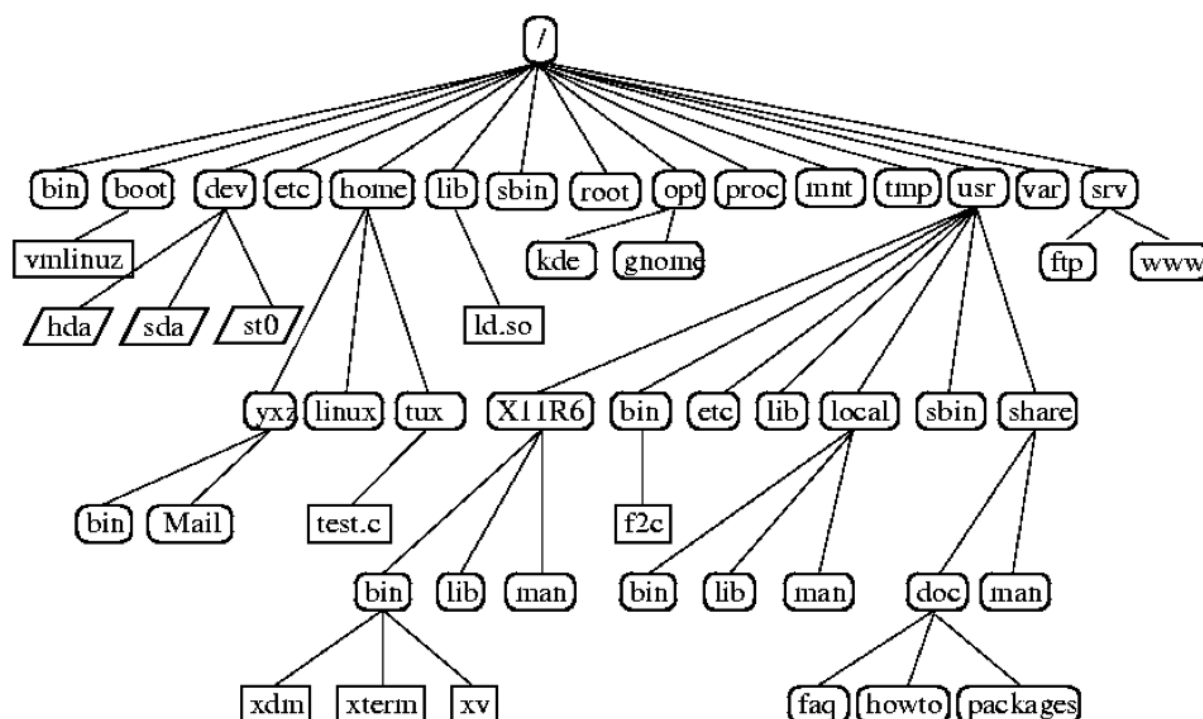


Fig. 1.1: Depending on the Linux distribution you might or might not find all of the above directories. Most important directories for you are /bin and /usr/bin (sometimes also /usr/local/bin) which contain the user software, /home which usually contains the users’ homedirectories and /tmp which can be used to store temporary data (be-ware: 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 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 start 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 regardless 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 filename).

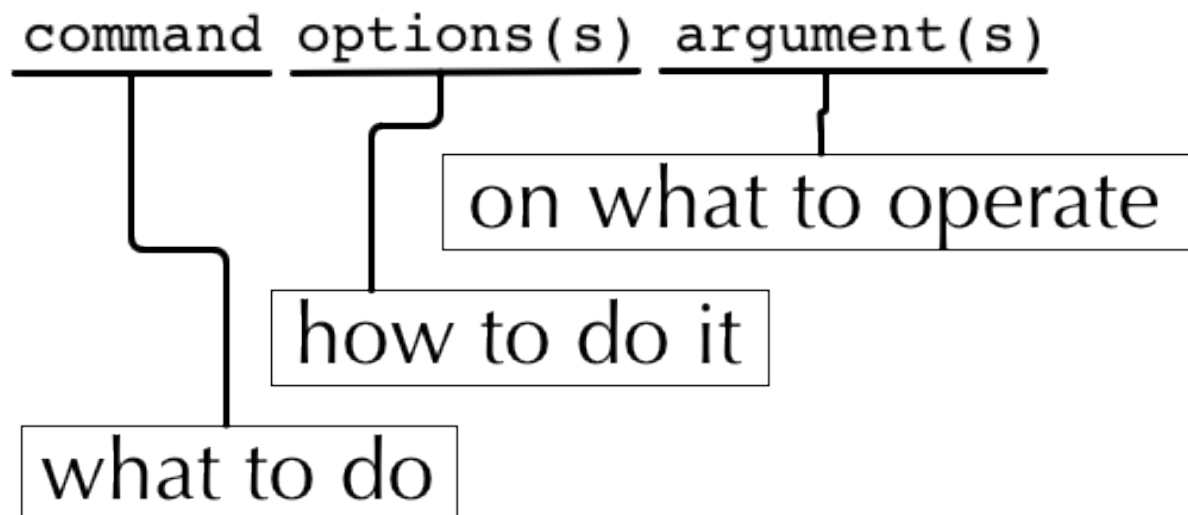


Fig. 1.2: General structure of Linux commands.

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 commandline 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 - concatenatable:

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

Some options require an additional argument, which is added after a blank to the short form and 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 Commandland

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.

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.

Saving time/avoiding typos with autocompletion

On most Linux systems you can autocomplete command names and filepaths by pressing TAB. This looks at the characters that you have entered so far and tries to predict what the rest of the command/path will be. This can save you from having to type out long command and file/directory names, and also reduces the likelihood of you accidentally spelling something incorrectly.

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 [Variables](#) (page 44)...

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 echo
echo(1)

NAME
    echo - display a line of text

SYNOPSIS
    echo [SHORT-OPTION]... [STRING]...
    echo LONG-OPTION
    ...
$
```

For the navigation within a manpage and how to exit the manpage, see the [paragraph regarding less](#) (page 15).

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)

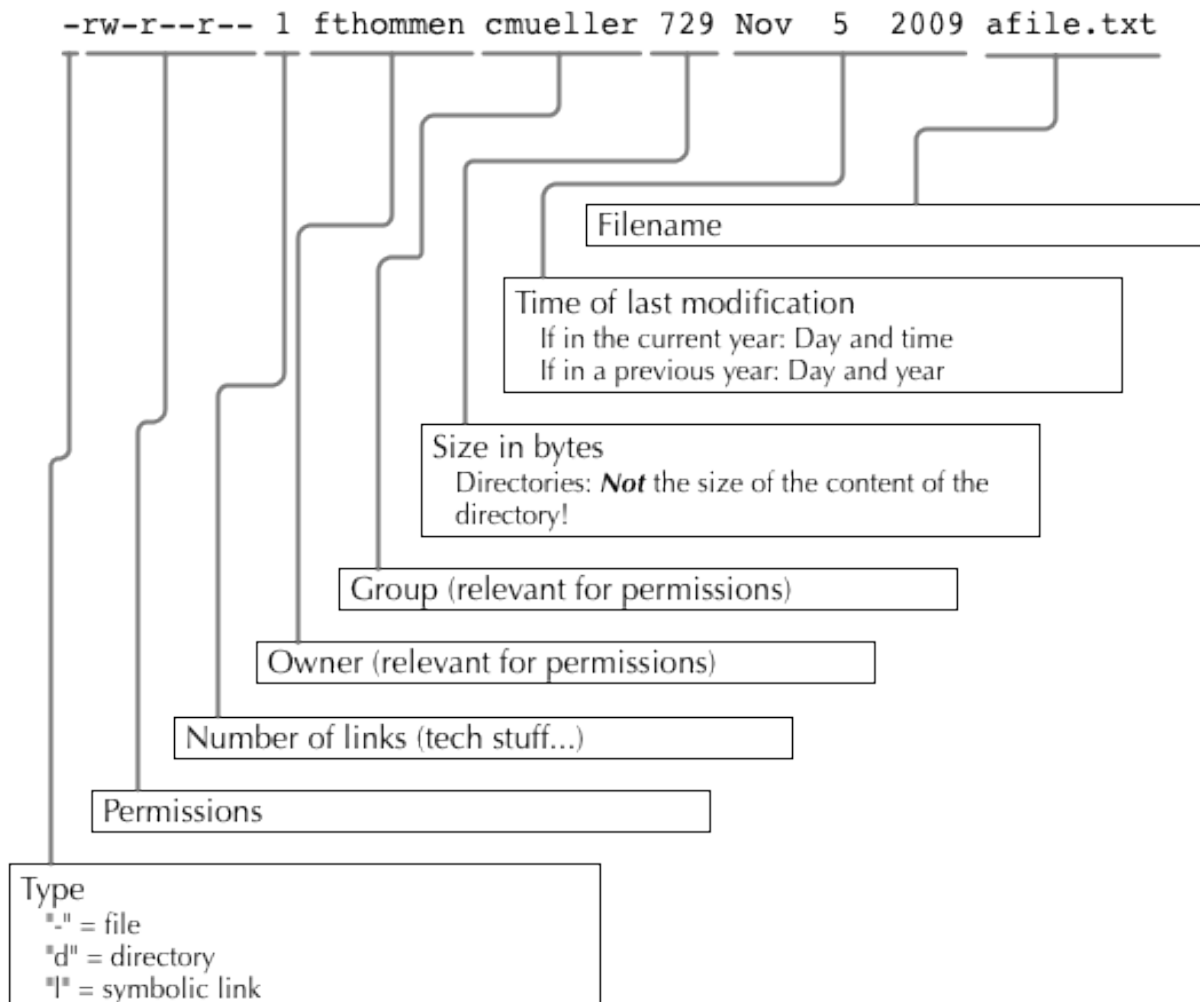


Fig. 1.3: Elements of a long file listing (`ls -l`)

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

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
$
```

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

Examples:

If the last filename given is nonexisting then the first file is copied as this new filename:

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

Be careful! If the last filename given does exist, this file will be overwritten and replaced with a copy of the first file.

If the last filename given is an (existing!) directory, then the file is copied into this directory:

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

This allows us to copy multiple files into the same directory at the same time:

```
$ cp /usr/bin/less /usr/bin/grep /usr/bin/tail /tmp/
$
```

To recursively copy files, we need to specify the `-r` option. Here, we copy a set of exercise files from the network share into our home directory:

```
$ cp -r /g/bio-it/courses/LSB ~/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` copy 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 destination
Beware: This will also copy files which are older on the destination side |
| -au | Silently copies all source files which are different (different size, different age) or missing from the destination 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/
$
```

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 44)

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
-A NUM	print NUM lines of trailing context (After)
-B NUM	print NUM lines of leading context (Before)
-C NUM	print NUM lines of output context (Context)

Examples:

- List all files in the current directory which contain the searchterm `Ensembl`:

```
$ grep -l Ensembl ./*  
P04637.txt  
P12931.txt
```

Note: You cannot combine the option `-v` and `-l` to find files which do not contain a certain searchterm. The reason is that `grep` works line-based and not really file-based... Therefore you should rather use the uppercase `-L` option!

- List all files in the current directory which **do not** contain the searchterm `Ensembl`:

```
$ grep -L Ensembl ./*  
1FMK.pdb  
3A4O.pdb  
...
```

- Count the number of occurrences (case insensitive!) of the term `atom` in all `pdb` files:

```
$ grep -ic atom ./*.pdb
```

- Find the term 'Homo sapiens' in the file `P04637.txt`, but also print two lines before the match:

```
$ grep -B2 'Homo sapiens' P04637.txt
```

- Find the term 'Homo sapiens' in the file `P04637.txt`, but also print the three lines following the match:

```
$ grep -A3 'Homo sapiens' P04637.txt
```

- Find the term 'Homo sapiens' in the file `P04637.txt`, but also print the surrounding five lines:

```
$ grep -C5 'Homo sapiens' P04637.txt
```

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 500 kBytes)

```
$ 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
$
```

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

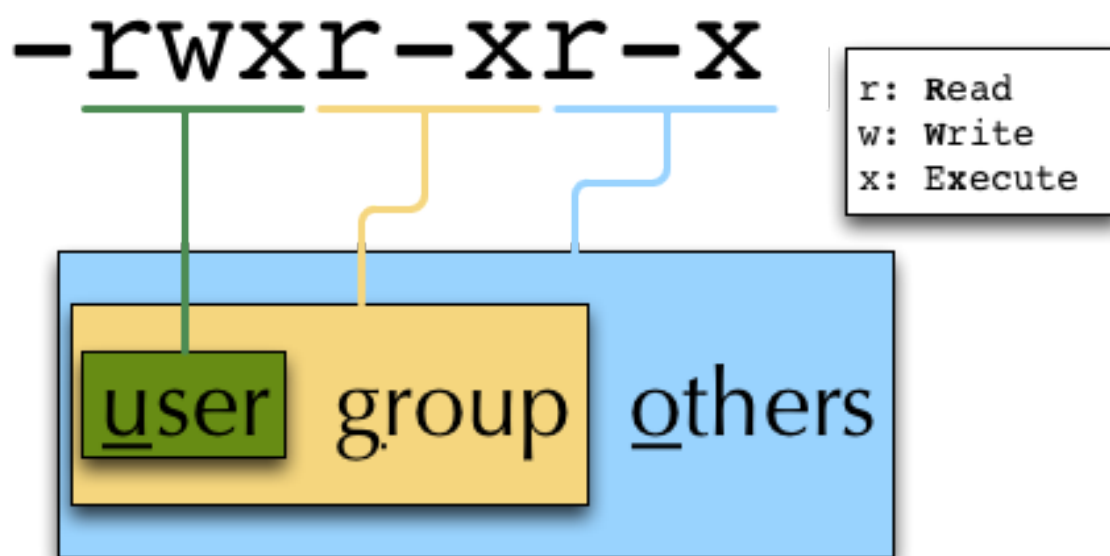


Fig. 1.4: Linux file permissions

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 Copying / Deleting Files & Folders

1. Navigate to your home directory
2. In your home directory, create a new directory named `new_dir`
3. Change into this directory, create a new empty file in there named `new_file`, and make sure that the file was created.
4. Duplicate this file by copying it as a new file named `another_file`
5. Delete the first file `new_file`
6. Also delete the directory (you are currently in) `~/new_dir`. Does it work?

2.3 View Files

1. Which tools can you use to see the first/last lines of the file `~/exercises/P12931.txt`?
2. How to only show the first/last three lines (of the same file)?
3. How do you print the whole file on the screen?

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

2.5 Misc. terminal

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

2.6 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 that you are still able to read the file 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`?

2.7 Remote access

1. Login to machine `"submaster.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 `submaster.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

2.8 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 22) symbol (`|`) and `sort` to sort this output *numerically*

2.9 Putting it all together

1. Create a new directory named `myscripts` in your homedirectory
2. Create an empty file named `mydate` in the newly created directory
3. Add the directory `~/myscripts` to your `PATH` environment variable
4. Use `echo` in combination with Redirection/Append to write "date" into the file `~/myscripts/mydate`

5. Change the permissions of the file `mydate` to be executable by you (and you only)
6. Run the file `mydate` (it should print the current date & time). Make sure you can run it from any directory (change to your homedirectory and just type `mydate`).

2.10 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`
4. Use either `less` or redirection to a file to manage the amount of information that `blastp` prints on your screen.

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/grep

```
$ file /usr/bin/grep
/usr/bin/grep: 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 Copying / Deleting Files & Folders

1. Navigate to your home directory

```
$ cd ~
```

or just

```
$ cd
```

2. In your homedirectory, create a new directory named `new_dir`

```
$ mkdir ~/new_dir
```

3. Change into this directory, create a new empty file in there named `new_file`, and make sure that the file was created:

```
$ cd ~/new_dir
$ touch new_file
$ ls new_file
```

4. Duplicate this file by copying it as a new file named `another_file`:

```
$ cp new_file another_file
```

5. Delete the first file `new_file`:

```
$ rm new_file
```

6. Also delete the directory (you are currently in) `~/new_dir`.

```
$ rmdir ~/new_dir
```

7. Did the deletion work? If not, try to remove all files from the directory first...:

```
$ rm ~/new_dir/*
$ rmdir ~/new_dir
```

3.3 View Files

1. Which tools can you use to see the first/last lines of the file `~/exercises/P12931.txt`?:

```
$ head ~/exercises/P12931.txt
$ tail ~/exercises/P12931.txt
```

2. How to only show the first/last three lines (of the same file)?:

```
$ head -n 3 ~/exercises/P12931.txt
$ tail -n 3 ~/exercises/P12931.txt
```

3. How do you print the whole file on the screen?:

```
$ cat ~/exercises/P12931.txt
```

or

```
$ less ~/exercises/P12931.txt
```

3.4 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 named date in the /bin directory

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

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

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

3.5 Misc. terminal

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

```
$ clear
```

or:

```
$ reset
```

3.6 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 that you are still able to read the file 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
```

3.7 Remote access

1. Login to machine "submaster.embl.de" (using your own username)

```
$ ssh submaster.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 submaster.embl.de into your local home directory (using scp):

```
$ scp submaster.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
```

3.8 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 22) symbol (`|`) and *sort* to sort this output *numerically*:

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

3.9 Putting it all together

1. Create a new directory named `myscripts` in your homedirectory:

```
$ mkdir ~/myscripts
```

2. Create an empty file named `mydate` in the newly created directory:

```
$ touch ~/myscripts/mydate
```

3. Add the directory `~/myscripts` to your `PATH` environment variable:

```
$ export PATH=$PATH:~/myscripts
```

4. Use `echo` in combination with Redirection/Append to write “date” into the file `~/myscripts/mydate`:

```
$ echo "date" >> ~/myscripts/mydate
```

5. Change the permissions of the file `mydate` to be executable by you (and you only):

```
$ chmod u+x ~/myscripts/mydate  
$ chmod go-x ~/myscripts/mydate
```

6. Run the file `mydate` (it should print the current date & time). Make sure you can run it from any directory (change to your homedirectory and just type `mydate`):

```
$ mydate
```

Congratulation, you’ve just created and run your first shell script!

3.10 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
```

4. Use either *less* or a redirection into a file to manage the amount of information that *blastp* prints on your screen.:

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

or:

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

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:

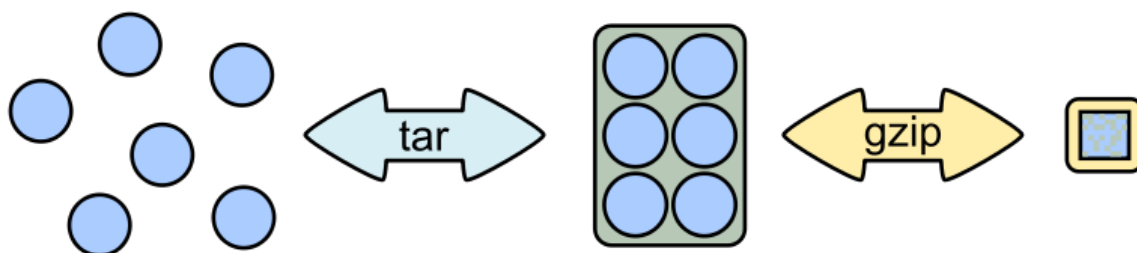


Fig. 4.1: source: Th0msn80 (Wikipedia)

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 concatenated together...

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

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

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

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

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

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

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

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 \$

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

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!

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:

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

3. Access the variables in a subshell:

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

4.4 Hints

4.4.1 Quoting

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

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

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

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

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

Single quotes:

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

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

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

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

Expanding and Escaping

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

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

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

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

Note: The “escape character” is usually the backslash “\”.

4.4.2 Keyboard Shortcuts

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

Tab-Completion: A Reminder

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

Move Quickly Through the Command Line

As well as tab-completion, you might be aware of CTRL-A to jump the cursor to the beginning of a line, and CTRL+E to jump to the end. On most systems, using the arrow

keys while holding down the `alt` key will jump left or right by one word (or word-like string) at a time.

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

Searchable Command History

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

Job Management

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

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

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

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

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

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

```
# kill %2  
[2]-  Terminated: 15          sleep 250
```

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

Chapter 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

Commandline Exercises

7.1 TAR & GZIP

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

7.2 GREP

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

7.3 SED

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

7.4 AWK

1. Use awk to print only those lines that contain “version” in the files P12931.txt and P05480.txt and think about how this procedure is different to sed.
2. For all FASTA files that begin with “P” (“P*.fasta”) print only the second item of the header (split on “|”) eg. for “>sp|P12931|SRC_HUMAN Proto-oncogene”, print only “P12931”
3. The file “P12931.csv” contains phosphorylation sites in the protein P12931. (If the file “P12931.csv” does not exist, use wget <http://phospho.elm.eu.org/byAccession/P12931.csv> to download it).
 - (a) Column three of this file lists the amino acid position of the phosphorylation site. You are only interested in position 17 of the protein. Try to use “grep” to filter out all these lines containing “17”.
 - (b) Now use awk to show all lines containing “17”.
 - (c) Next try show only those lines where column three equals 17 (Hint: The file is semicolon-separated...).
 - (d) Finally print the PMIDs (column 6) of all lines that contain “17” in column 3.

7.5 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 8

Basic Shell Scripting

8.1 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!

8.2 Script Naming and Organization

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

We recommend to either store all scripts in one location (e.g. ~/bin) and add this location to your \$PATH variable (see [Variables](#) (page 44)) 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!

8.3 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
[...]  
$
```

8.3.1 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.

8.3.2 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 77)) Even if your script is very simple - document it! This helps others understand what you did, but - most importantly - it helps you remember what you did, when you have to reuse the script in the future.

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

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

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

8.3.3 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.

8.3.4 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 status. The exit status of a script is usually the exit status of the last executed command, which is reported by the environment variable `?`:

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

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

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

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

See [Ensuring a Sensible Exit Status](#) (page 76) about how to control the exit status of your script.

8.3.5 Command Grouping and Sequences

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

`cmd1; cmd2` – Execute commands in sequence

Example: Create a directory and change into it:

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

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

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

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

Example: Confirm that `/etc` exists:

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

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

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

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

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

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

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

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

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

Example: Count the heterogens described in a gzipped PDB file or, if it doesn't exist, download the file:

```
$ gzip -c 4ZZN.pdb.gz && sed -n '/^HET /p' || wget "http://www.rcsb.org/pdb/files/4ZZN.pdb.gz"
$
```

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

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

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

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

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

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


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

8.4 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.

8.4.1 Conditional Statements

if - then - else

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

sh/bash	csh/tcsh
<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 *cs**h*/*tc**sh*

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 *cs**h*/*tc**sh* the `test` (or `[]`) command is not needed. Conditions and comparisons are directly placed within the round braces.

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

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

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

or:

```
if test -e ./sequence_files
then
    ls -l ./sequence_files/*.fasta
else
    echo no sequence_files directory here
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 $PATH in
  */opt/* )
    echo /opt/ paths found in \"$PATH\"
    ;;
  */etc/* )
    echo /etc/ paths found in \"$PATH\"
    ;;
  *)
    echo '/opt and /usr are not contained in $PATH'
    ;;
esac
```

or

case \$PATH in

```

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

```

8.4.2 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	csh/tcsh
<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 filenames and count number of sequences in every FASTA file in ./sequence_files:

```

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

```

while / until

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

sh/bash	csh/tcsh
<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

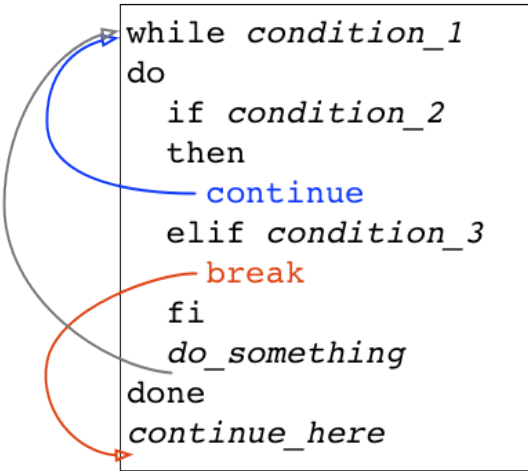





Fig. 8.1: Loop control

Symbol	
	Regular loop cycle
	break due to condition_2
	continue due to condition_3

8.5 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.

8.5.1 Configurable Scripts

Using Variables

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

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

```
#!/bin/sh

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

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

```
#!/bin/sh

MYDIR=/etc

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

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

Using a Settings File

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

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

```
MYDIR=/etc
```

The script:

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

```
. ./settings.ini

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

8.5.2 Defining your own Commandline Options and Arguments

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

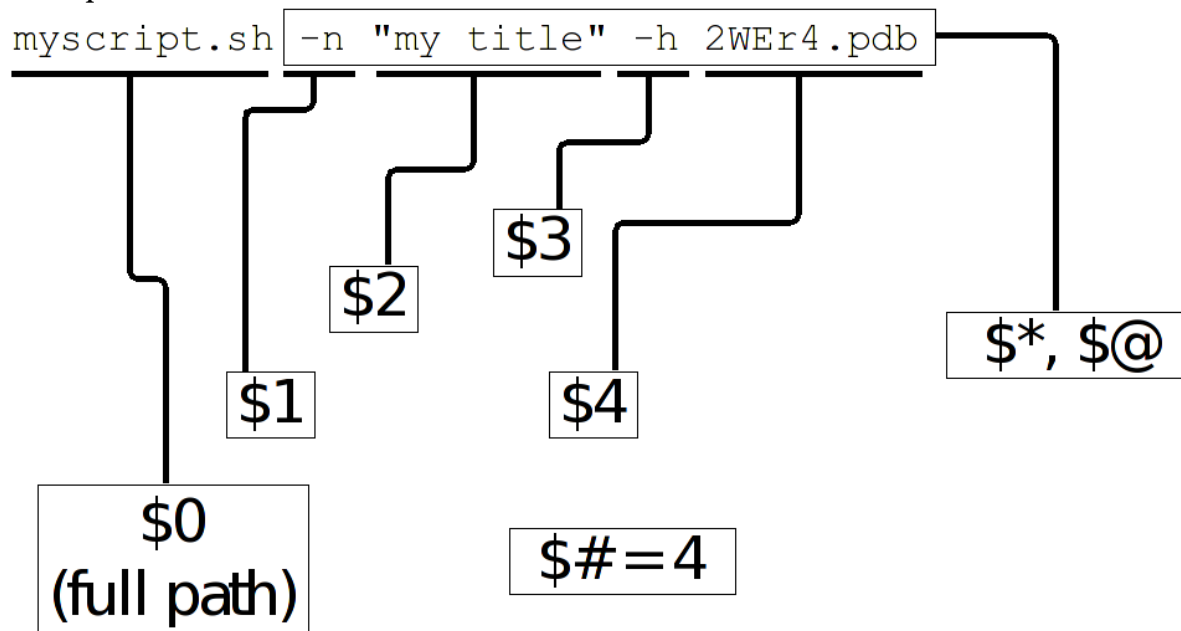
`$0`: path to the script. Either the path as you specified it or the full path if the script was executed through `$PATH`

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

`$#`: Current number of positional parameters

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

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



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

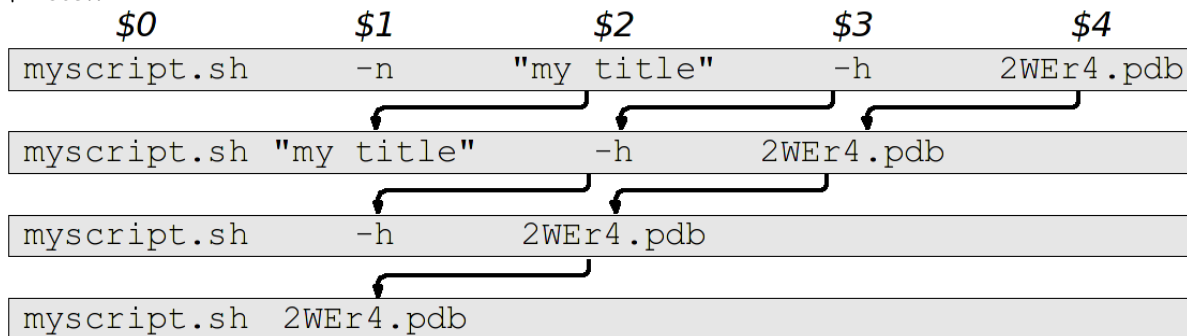
If you run the script


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

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

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

8.6 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 64)). 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 status 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
```

8.6.1 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.

8.7 Tips and Tricks

8.7.1 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
$
```

8.7.2 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.

8.7.3 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

8.7.4 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).

8.7.5 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
$
```

8.7.6 Create Temporary Files

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

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

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

8.7.7 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 9

Solutions to the Exercises

9.1 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
```

9.2 GREP

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

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

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

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

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

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

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

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

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

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

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


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

...or by telling `grep` to only count those lines that start with “ATOM”:

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

9.3 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
```

9.4 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
```

9.5 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.

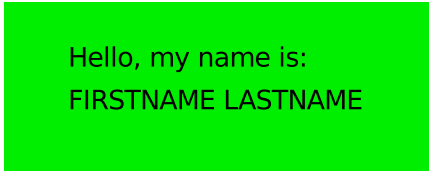
Chapter 10

Propositions for Scripting Exercises

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

10.1 Replace Names in SVG

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



Hello, my name is:
FIRSTNAME LASTNAME

looks like this:

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

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

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

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

Usage Example:

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

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

Things to consider:

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

10.2 General “Unpacker”

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

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

Usage Example:

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

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

Things to consider:

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

- Some tools preserve the original archive, others don't... Maybe you can add consistency here?

Extendibility:

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

10.3 Safety Backup Creator

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

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

Usage Example:

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

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

Things to consider:

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

Extendibility:

- Add option to pack/compress the data.

10.4 Column Chooser (advanced)

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

Datafile:

NAME	FIRSTNAME	BIRTHDATE	STREET	NO
Meier	Daniel	30-MAY-1990	Meyerhofstrasse	12
Mueller	Andreas	29-FEB-1960	Bahnhofstr.	1b

```
Schmid Ariane 1-DEC-1990 Bahnhofstrasse 13
vonMyra Nikolaus 15-MAR-270 Dezemberstrasse 6
```

Usage Example:

```
$ columnchooser.sh FIRSTNAME NO
Daniela 12
Andreas 1b
Ariane 13
Nikolaus 6
$ columnchooser.sh CITY
Sorry, no column "CITY" found
$
```

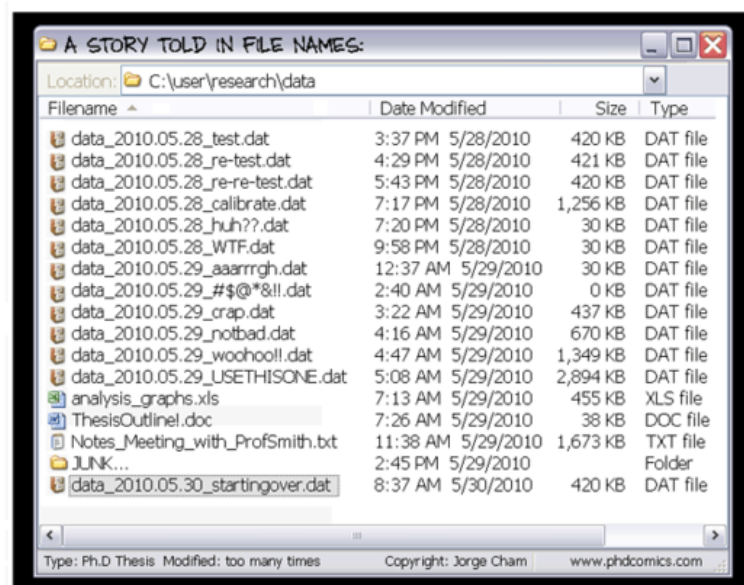
Required tools and commands: awk, eval

Extendibility:

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

Chapter 11

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.

11.1 git at a Glance

11.1.1 git commands

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
```

11.1.2 git concepts

commit

A commit is a recorded set of changes in your project's file(s). Try to group *logical* sets of changes together into one commit – don't mix changes which are unrelated.

repository

A repository is the history of all your project's commits.

11.2 git settings

11.2.1 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, enabling color output as well as `difftool`:

```
git config --global core.editor nano
git config --global color.ui auto
git config --global merge.tool kdiff3
```

11.2.2 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

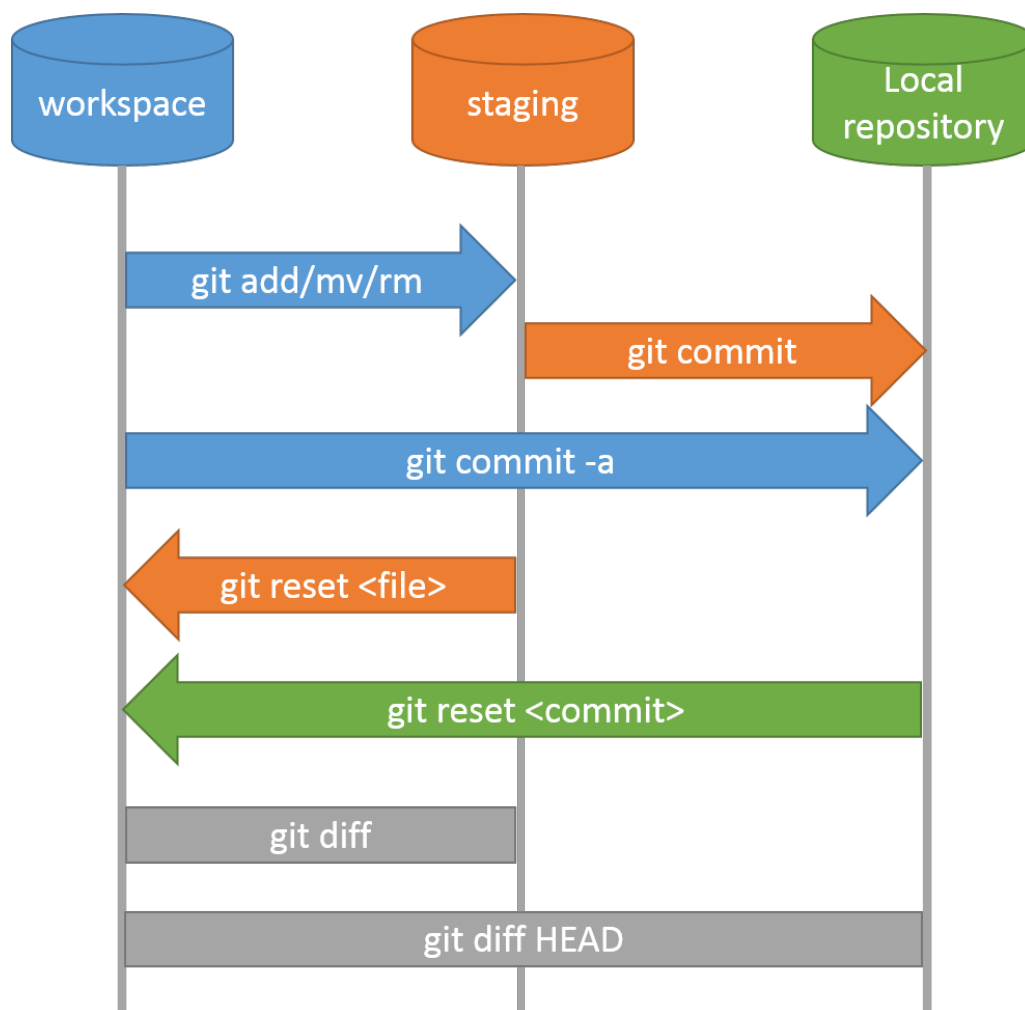


Fig. 12.1: Files are *added* from the *workspace*, which always holds the current version of your files, to the *staging area*. *Staged* files will be stored into the local repository in the next *commit*. The repository itself contains all previous versions of all files ever committed. (image courtesy of 'research bazaar' <https://raw.githubusercontent.com/resbaz/lessons/master/git/git-local.png>)

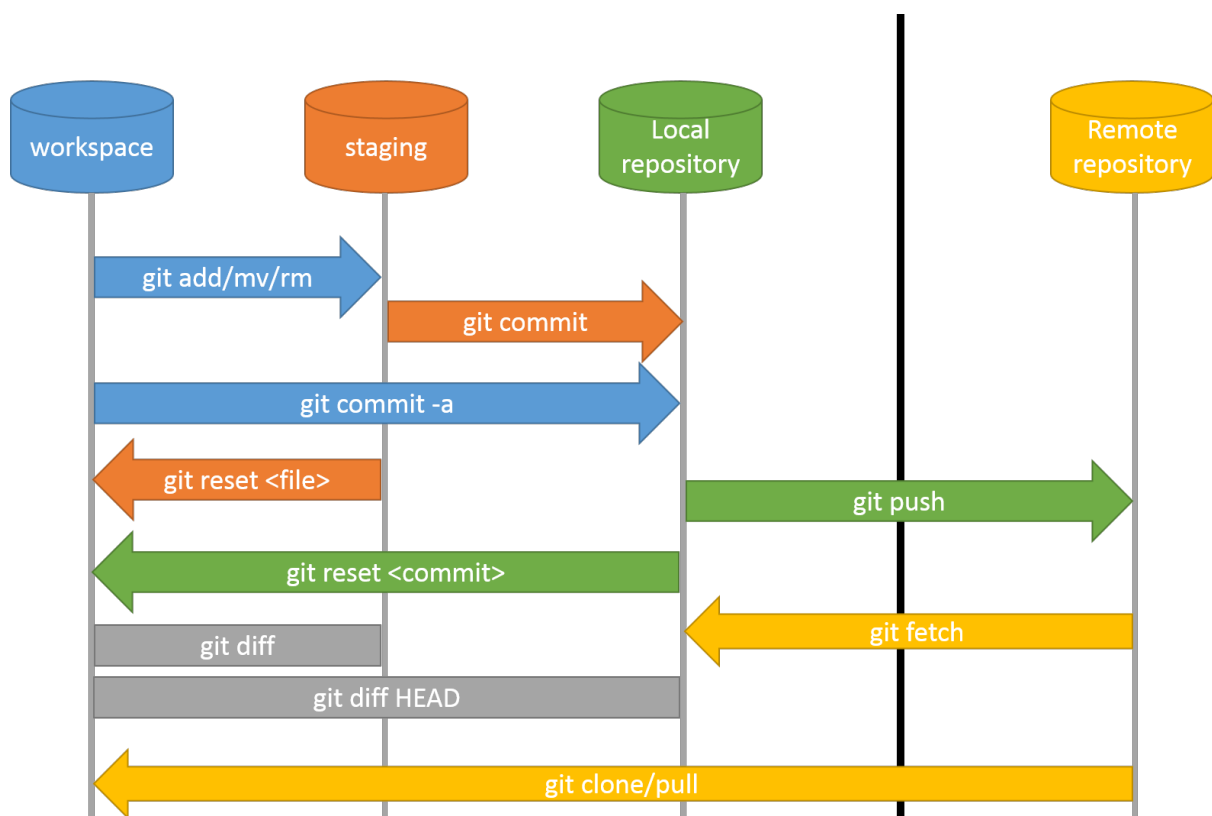


Fig. 12.2: Distributed workflow using a centralized repository. Here, you use *push* and *pull* to synchronize your local repository with a remote repository. (image courtesy of 'research bazaar' <https://raw.githubusercontent.com/resbaz/lessons/master/git/git-remote.png>)

12.1 Creating a git Repository

Turning an existing directory into a local git repository is as simple as changing into that directory and invoking *git init*. However, here we want to create one repository which we can use from multiple other folders to sync to/from, therefore in this case, we need to initialize it as a *bare* repository.

Note: Normally you do not need the *-bare*, but it's essential for this exercise...

So, here we first create an empty directory in our homedirectory called *repos* (this is meant to hold and serve all our repositories), and create a repository in there called *mythesis*:

```
mkdir ~/repos
cd ~/repos
mkdir mythesis
cd mythesis
git init --bare
```

Note: As a result, you should have the directory *~/repos/mythesis* and there should be a directory called *.git* in this directory...

12.2 Cloning a git Repository

Next, we can *clone* this repository into the *~/Documents/mythesis* folder.:

```
cd ~/Documents

git clone ~/repos/mythesis

Initialized empty Git repository in /localhome/training/Desktop/mythesis/.git/
warning: You appear to have cloned an empty repository.

cd mythesis
```

By *cloning*, we not only get the exact copy as the remote side, but we automatically tell git where we had got the data from, which allows us later to sync our changes back...

Note: You can clone from either a different folder on our computer, a remote machine (via ssh), or a dedicated git server:

Local directory:

```
git clone ~/repos/mythesis
```

Remote directory:

```
git clone ssh://remote_user@remote_server/mythesis.git
```

Remote git server:

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

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)
```

Here, everything is clear, not much going on (no news is good news).

Note: In fact, it's good practice, to use *git status* as often as possible!

12.4 Adding files

First, we'll create a new file:

```
echo "My first line towards a great paper!" > paper.txt

git status

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

Here, git tells us that there is a file, however it's *untracked*, meaning git does not know/care about it. We need to tell git first that it should keep track of it. So we'll add this file to the so called *staging area*:

```
git add paper.txt

git status

# On branch master
```

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

This tells us that the *paper.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>

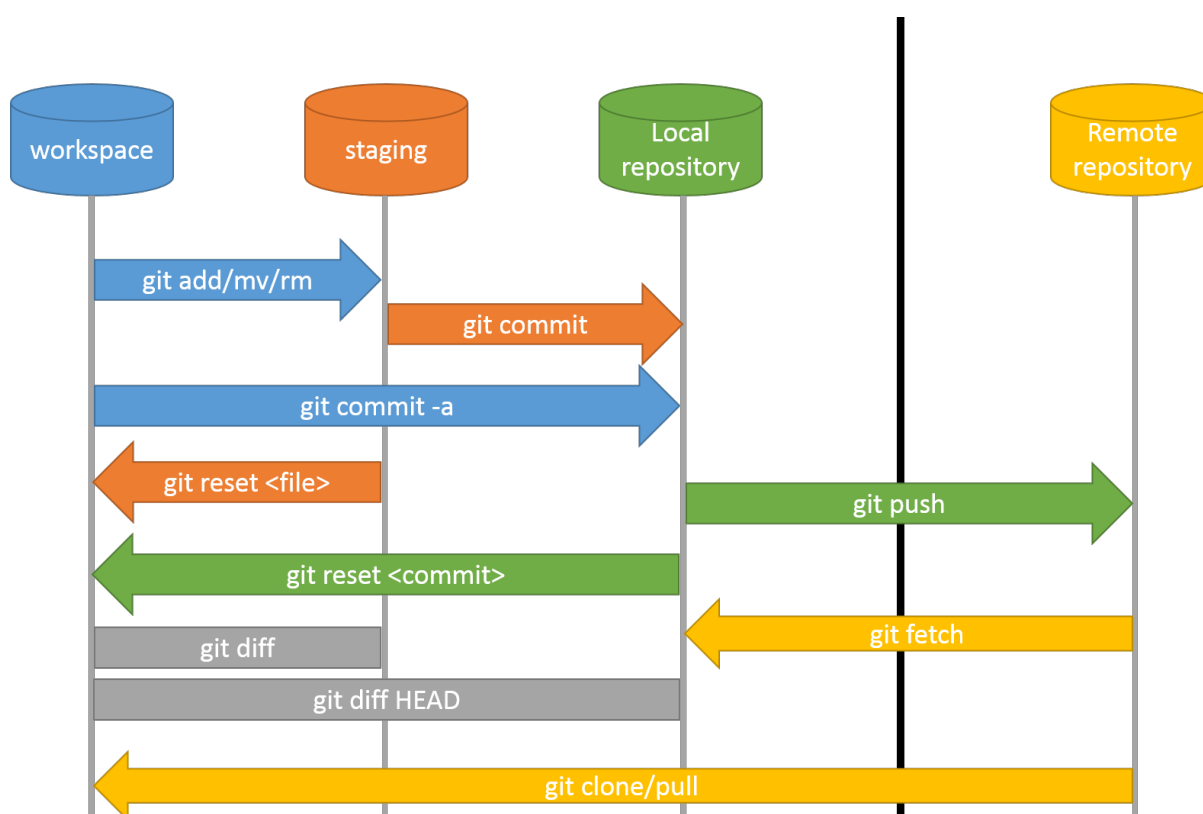
message describing the changes you made
```

12.6.1 Exercise

Repeat the add/commit procedures you just learned. Add more files, use an editor to add more content to the *paper.txt* file, commit your changes providing a meaningful commit message.

12.7 Pushing changes

In order to exchange/synchronize your changes with a remote repository, you use *git push*/*git pull*:



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, read it carefully! The most common error you get when trying to push are changes on the remote end which you first need to merge into your local repository before you are allowed to push your own...

12.7.1 Creating a second clone

In order to simulate contributing to our repository from another computer, we will again checkout the repository, but this time in a different folder named *mythesis-work*:

```
cd ~/Documents

git clone ~/repos/mythesis mythesis-work

cd ~/Documents/mythesis-work
```

This repository should contain all the changes you've pushed so far. Now we want to improve our *paper.txt* document. Use an editor to add more lines to this file:

```
echo "This line was contributed from work..." >> paper.txt
```

Again, *add*, *commit*, and *push* your changes.

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.

So we go back to the directory *~/Documents/mythesis* and (after checking the status) try to get the changes we've done in the *mythesis-work* directory:

```
cd ~/Documents/mythesis

git status

git pull
...
Auto-merging paper.txt
CONFLICT (content): Merge conflict in paper.txt
Automatic merge failed; fix conflicts and then commit the result.
```

12.9 Solving conflicts

When working collaboratively on a project, it is unavoidable that the same file gets changed by different contributors. This causes a conflict and needs to be dealt with.

Hint: It helps minimizing conflicts if you push/pull often!

To solve a merge conflict, you can either:

- manually merge the two files (see below)
- discard the remote file: `git checkout --ours conflicted_file.txt`
- discard the local file: `git checkout --theirs conflicted_file.txt`

12.9.1 Manually merging a conflict

To create a conflict, we change the same line in the file *paper.txt* in both directories (*mythesis* and *mythesis-work*) without pulling each others changes in between. Once we pull, git will tell us that a conflict has occurred.:

Automatic merge failed; fix conflicts and then commit the result.

When git encounters conflicts in files, it adds special markers <<<<<<, =====, >>>>>> into this file wrapping both conflicting changes. It is up to you to decide which of these changes to keep.:

```
...  
content of the file  
...  
<<<<<< HEAD:paper.txt  
your home changes  
=====  
your changes introduced at work  
>>>>>> 00000000000000000000000000000000000000000000000000000:paper.txt  
...  
rest of the file  
...  

```

Make sure to delete the lines that were introduced by git (otherwise you won't be able to commit changes). If you only wanted to keep your changes then you would delete everything except your changes:

```
...
content of the file
...
your home changes
...
rest of the file
...
```

Now, you need to add this file again to the staging area and commit to finish this conflicting merge. Use *git status* to see the status of the repository.

12.10 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!
--

If you want to checkout a specific version (revision) of a file, you need to specify the hash or name of the revision:

```
git checkout revision_name <filename>
```


Chapter 13

Appendix

13.1 Links and Further Information

13.1.1 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://www.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
- [BioPieces](http://code.google.com/p/biopieces) ⁶ are a collection of bioinformatics tools that can be pieced together in a very easy and flexible manner to perform both simple and complex tasks.
- [Google shell style guide](https://code.google.com/p/google-styleguide) ⁷
- [Useful bash one-liners for bioinformatics](https://github.com/stephenturner/oneliners) ⁸
- Interactive explanation of your commandline: [Explain Shell](http://www.explainshell.com) ⁹
- [Bash One-Liners Explained, Part III: All about redirections](http://www.catonmat.net/blog/bash-one-liners-explained-part-three) ¹⁰
- [Bash Redirections Cheat Sheet](http://www.catonmat.net/blog/bash-redirections-cheat-sheet) ¹¹
- [Redirection Tutorial](http://wiki.bash-hackers.org/howto/redirection_tutorial) ¹²

¹ <http://linuxcommand.org/>

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

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

⁴ <http://www.catcode.com/teachmod/>

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

⁶ <http://code.google.com/p/biopieces>

⁷ <https://code.google.com/p/google-styleguide>

⁸ <https://github.com/stephenturner/oneliners>

⁹ <http://www.explainshell.com>

¹⁰ <http://www.catonmat.net/blog/bash-one-liners-explained-part-three>

¹¹ <http://www.catonmat.net/blog/bash-redirections-cheat-sheet>

¹² http://wiki.bash-hackers.org/howto/redirection_tutorial

13.1.2 Command Line Mystery Game

CLMystery¹³ 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
```

13.1.3 Recommended Reading: Real printed paper books

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

13.1.4 Running Linux Commands in Mac

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

13.1.5 Running Linux Commands in Windows

Babun

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

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

13.1.6 Live - CDs

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

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

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.

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

¹⁵ <http://knopper.net/knoppix>

¹⁶ <http://bioknoppix.hpcf.upr.edu>

¹⁷ <http://www.vigyaan.cd.org>

¹⁸ <http://www.bioslax.com>

13.2 About Bio-IT

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

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

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

13.2.1 Centres

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



Biomolecular Network Analysis

The [CBNA](#) disseminates expertise, know-how and guidance in network integration and analysis throughout EMBL.

Statistical Data Analysis

The [CSDA](#) helps EMBL scientists to use adequate statistical methods for their specific technological or biological applications.

Modeling

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

13.3 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
- Yan Ping Yuan
- Bora Uyar
- Thomas Zichner

Index

Symbols

[..... 68
..... 63
\$? 64
\$HOME 23
\$PAGER 23
\$PATH 23
\$PWD 23
& 48
| 22
>> 22
> 22
] 68

A

append 22
apropos 6
awk 41, 88

B

backquote 78
backtick 78
bg 48
break 72
breaksw 70
bunzip2 86
bzip2 86

C

case 70, 75
cat 14, 34
cd 8, 31
chmod 19, 32
clear 5, 31, 33
command 3
 general structure 3
 interrupt 5
 switches 3
command substitution 78
comment 63
continue 72
cp 10, 32, 87
cut 17, 34

D

date 7, 34, 87
disconnect 21
du 87

E

echo 5, 23
elif 70
env 23, 45, 54
environment variables 23
 display 23
 set 24
escape 47
escape character 77
eval 88
exit 5, 21, 33
exit status 64, 76
export 24

F

fg 48
file 18, 29, 33, 86
 append 22
 overwrite 22
find 18, 31
for 71, 86
foreach 71

G

grep 15, 22, 34, 39
gunzip 86
gzip 37, 57, 81, 86

H

hash sign 63
head 14
hostname 7, 21, 33

I

if - then - else 67
interpreter 62

J

jobs 48

K

kill 48

L

less 15, 23

ls 8, 32, 33, 37, 87

M

man 6

mkdir 13, 31

more 23

mv 13

O

options 3

P

pattern 70

Permissions 19

pipe 22

positional parameters 74

pwd 7

Q

quoting 46

R

redirect 22

Remote access 20

reset 5, 31

rm 12

rmdir 13

rsync 11, 87

S

scp 21, 33

secure copy 21

secure shell 20

sed 39, 86

set 24, 45, 54, 77

shebang line 62

shift 75

sort 17

special variables: \$? 64

ssh 20, 33, 34

T

tail 14

tar 38, 57, 81, 86

test 68

time 7

touch 10

U

unset 77

until 71, 72

unzip 86

V

variables

 environment variables 44, 53

 shell variables 44, 53

W

which 18, 32

while 71, 72, 86

whoami 7, 21, 33

X

xz 86

Z

zip 86

File Commands	System Info
ls - directory listing	date - show the current date and time
ls -al - formatted listing with hidden files	cal - show this month's calendar
cd <i>dir</i> - change directory to <i>dir</i>	uptime - show current uptime
cd - change to home	w - display who is online
pwd - show current directory	whoami - who you are logged in as
mkdir <i>dir</i> - create a directory <i>dir</i>	finger <i>user</i> - display information about <i>user</i>
rm <i>file</i> - delete <i>file</i>	uname -a - show kernel information
rm -r <i>dir</i> - delete directory <i>dir</i>	cat /proc/cpuinfo - cpu information
rm -f <i>file</i> - force remove <i>file</i>	cat /proc/meminfo - memory information
rm -rf <i>dir</i> - force remove directory <i>dir</i> *	man <i>command</i> - show the manual for <i>command</i>
cp <i>file1 file2</i> - copy <i>file1</i> to <i>file2</i>	df - show disk usage
cp -r <i>dir1 dir2</i> - copy <i>dir1</i> to <i>dir2</i> ; create <i>dir2</i> if it doesn't exist	du - show directory space usage
mv <i>file1 file2</i> - rename or move <i>file1</i> to <i>file2</i> if <i>file2</i> is an existing directory, moves <i>file1</i> into directory <i>file2</i>	free - show memory and swap usage
ln -s <i>file link</i> - create symbolic link <i>link</i> to <i>file</i>	whereis <i>app</i> - show possible locations of <i>app</i>
touch <i>file</i> - create or update <i>file</i>	which <i>app</i> - show which <i>app</i> will be run by default
cat > <i>file</i> - places standard input into <i>file</i>	Compression
more <i>file</i> - output the contents of <i>file</i>	tar cf <i>file.tar files</i> - create a tar named <i>file.tar</i> containing <i>files</i>
head <i>file</i> - output the first 10 lines of <i>file</i>	tar xf <i>file.tar</i> - extract the files from <i>file.tar</i>
tail <i>file</i> - output the last 10 lines of <i>file</i>	tar czf <i>file.tar.gz files</i> - create a tar with Gzip compression
tail -f <i>file</i> - output the contents of <i>file</i> as it grows, starting with the last 10 lines	tar xzf <i>file.tar.gz</i> - extract a tar using Gzip
Process Management	tar cjf <i>file.tar.bz2</i> - create a tar with Bzip2 compression
ps - display your currently active processes	tar xjf <i>file.tar.bz2</i> - extract a tar using Bzip2
top - display all running processes	gzip <i>file</i> - compresses <i>file</i> and renames it to <i>file.gz</i>
kill <i>pid</i> - kill process id <i>pid</i>	gzip -d <i>file.gz</i> - decompresses <i>file.gz</i> back to <i>file</i>
killall <i>proc</i> - kill all processes named <i>proc</i> *	Network
bg - lists stopped or background jobs; resume a stopped job in the background	ping <i>host</i> - ping <i>host</i> and output results
fg - brings the most recent job to foreground	whois <i>domain</i> - get whois information for <i>domain</i>
fg <i>n</i> - brings job <i>n</i> to the foreground	dig <i>domain</i> - get DNS information for <i>domain</i>
File Permissions	dig -x <i>host</i> - reverse lookup <i>host</i>
chmod <i>octal file</i> - change the permissions of <i>file</i> to <i>octal</i> , which can be found separately for user, group, and world by adding:	wget <i>file</i> - download <i>file</i>
<ul style="list-style-type: none"> 4 - read (r) 2 - write (w) 1 - execute (x) 	wget -c <i>file</i> - continue a stopped download
Examples:	Installation
chmod 777 - read, write, execute for all	Install from source:
chmod 755 - rwx for owner, rx for group and world	./configure
For more options, see man chmod .	make
SSH	make install
ssh <i>user@host</i> - connect to <i>host</i> as <i>user</i>	dpkg -i <i>pkg.deb</i> - install a package (Debian)
ssh -p <i>port user@host</i> - connect to <i>host</i> on port <i>port</i> as <i>user</i>	rpm -Uvh <i>pkg.rpm</i> - install a package (RPM)
ssh-copy-id <i>user@host</i> - add your key to <i>host</i> for <i>user</i> to enable a keyed or passwordless login	Shortcuts
Searching	Ctrl+C - halts the current command
grep <i>pattern files</i> - search for <i>pattern</i> in <i>files</i>	Ctrl+Z - stops the current command, resume with fg in the foreground or bg in the background
grep -r <i>pattern dir</i> - search recursively for <i>pattern</i> in <i>dir</i>	Ctrl+D - log out of current session, similar to exit
command grep <i>pattern</i> - search for <i>pattern</i> in the output of <i>command</i>	Ctrl+W - erases one word in the current line
locate <i>file</i> - find all instances of <i>file</i>	Ctrl+U - erases the whole line
	Ctrl+R - type to bring up a recent command
	!! - repeats the last command
	exit - log out of current session
	* use with extreme caution.