Intermediate Linux Course

Release 1.1

Holger Dinkel & Frank Thommen

July 09, 2013

CONTENTS

CHAPTER 1

MORE COMMANDLINE TOOLS

1.1 Command-line Tools

1.1.1 GZIP

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

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

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

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

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

1.1.2 TAR

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

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

The most common command-line switches are:

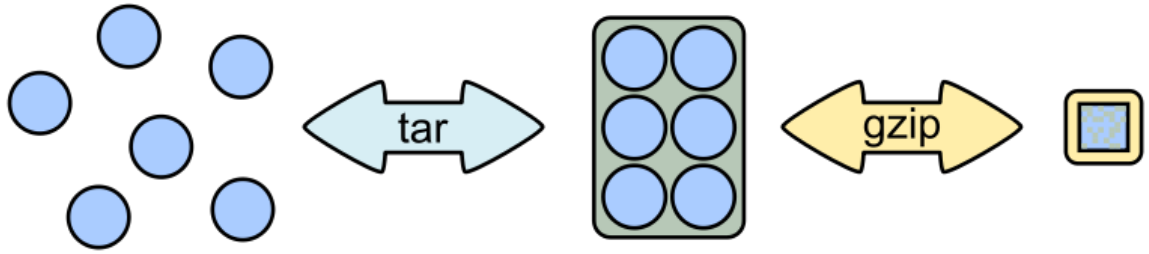

Figure 1.1: source: Th0msn80 (Wikipedia)

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

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
```

1.1.3 GREP

Find 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
```

1.1.4 SED

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

Usage: '`sed command file`',

The most common usecases are:

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

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

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

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

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

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

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

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

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

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

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

Delete lines matching the pattern:

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

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

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

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

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

1.1.5 AWK

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

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

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

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

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

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

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

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

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

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

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

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

You can also combine the pattern matching and the column selection techniques:

```
# awk '/more/ {print $3}' textfile
 even
```

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 fourth column is 'more':

```
# awk '$4=="more" {print $3}' textfile
 even
```

Note the double equal signs "==" to check for equality and note the quotes around "more". If you want to match a field, but not exactly, you can use '~' instead of '==':

```
# awk '$4~"ore" {print $3}' textfile
 even
```

1.2 Hints

1.2.1 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.
```

1.2.2 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
```

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

CHAPTER 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.

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 the file!) 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 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 inherited to shells and scripts started from your current.

3.1 Setting, Exporting and Removing Variables

Variables are set (created) by assigning them a value

```
# MYVAR=something::
```

There must be no whitespace around 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:

```
# export MYGLOBALVAR="something else"
```

```
# MYGLOBALVAR="something else"
# export MYGLOBALVAR
```

There is no `$` in front of the variable!

Variables are removed with unset:

```
# unset MYVAR
```

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

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

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.

3.4 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:

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
-----
```

"source" the script, i.e. run it within your current shell

```
# ./vartest.sh
I am local
I am global
-----
```

Access the variables in a subshell:

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

CHAPTER 4

BASIC SHELL SCRIPTING

4.1 What is a Script?

A script is nothing else than a number of shell command place together in a file. The sim- plest 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 customi- zation, 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! Script Naming and Organization It is good practice – though not technically required – to give your scripts an extension which specifies their type. I.e. ".sh" for Bourne Shell and Bourne Again Shell scripts, ".csh" for C-Shell scripts. Sometimes ".bash" for Bourne Again Shell scripts is used. We recommend to either store all scripts in one location (e.g. ~/bin) and add this location to your $PATH variable or to store the scripts together with the files that are processed by the script. If you use scripts to process data, then the scripts should probably be archived together with the data files Running a Script There are basically three ways to run a script: a) 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 by c) specifying the interpreter. The full path (relative or absolute) to script has to be provi- ded in this case, no matter wether the script location is already contained in $PATH or not:

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

4.1.1 Basic Structure of a Shellscript

Shellscripts have the following general structure:

1. A line starting with "#!" which defines the interpreter (i.e. the program used to run the script). This line is called the "shebang line" and must be the first line in a script
2. 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
3. A section where the actual processing is done. This includes error handling
4. 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.

4.1.2 Readability and Documentation

Make your script easily readable. Use comments and whitespace and avoid super compact but hardly understandable commandlines. Always take into account, that not only the shell, also human beings will probably have to read and understand your script. Even if your script is very simple – document it! This helps others understand what you did, but – most important – it helps you remember what you did, when you have to reuse the script in the future.

Documentation is done either by writing comments into the script or by creating a special documentation file (README.txt or similar). Documenting in the script can be done in several ways: #. A preamble in the script, outlining the purpose, parameters and variables of the script as well as some information about authorship and and perhaps changes #. Within the script as blocks of text or "End of line" comments

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

Shebang line:

```
#!/bin/sh
#
# myscript.sh
```

Preamble with a short description, usage information, authorship etc.:

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

Configuration:

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

Checking prerequisites and sane environment:

```
# --- 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
fi
```

This is what you actually wanted to do:

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

Ensure a valid and meaningful exit status:

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

Full script:

```
#!/bin/sh
#
# myscript.sh
# General purpose script for extracting Glycine
# occurrences in a datafile.
# Usage: myscript.sh datafile
# Exit values:
#    1: No datafile given or file doesn't exist
#    2: No Glycine found
#
# Author: Me, myself and I
# Date: Heidelberg, December 12., 2012
#
# --- Configuration ---
GREPCMD=/bin/grep
DATAFILE=$1
```

```
17 # --- Check prerequisites ---
18 # first check for $1
19 if [ -z $DATAFILE ]
20 then
21     echo "No datafile given" 1>&2 # print on STDERR
22     echo "USAGE: $0 datafile"
23     exit 1
24 fi
25 # then check if the file exists
26 if [ ! -f $DATAFILE ]
27 then
28     echo "Datafile $DATAFILE does not exist!" 1>&2
29     exit
30 fi
31 # --- Now processing---
32 $GREPCMD -q Glycine $DATAFILE # Where is Glycine?
33 # --- Exit ---
34 if [ $? -eq 0 ]
35 then
36     exit 0
37 else
38     exit 2
39 fi
```

CHAPTER 5

SOLUTIONS TO THE EXERCISES

5.1 Commandline tools

5.1.1 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 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
```

5.1.2 GREP

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

```
$ grep "TATATCTAA" ENST0*

ENST00000380152.fasta:ACGGAAGAATGTGAGAAAAATAAGCAGGACACAATTACAACTAAAAAATATATCTAA
ENST00000544455.fasta:ACGGAAGAATGTGAGAAAAATAAGCAGGACACAATTACAACTAAAAAATATATCTAA
```

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

```
$ grep "CAACAAA" ENST0*

ENST00000380152.fasta
ENST00000544455.fasta
```

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

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

4. Count the number of ATOMs (lines starting with "ATOM") in the file 1Y57.pdb.
5. Does this number agree with the annotated number of atoms (Search the REMARKs for "protein atoms")

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

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

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

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

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

5.1.3 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
```

5.1.4 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).

    1. 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
```

2. Now use awk to show all lines containing "17".

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

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

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

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

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

5.1.5 Quoting and Escaping

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"
```

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

CHAPTER 6

APPENDIX

6.1 Links and Further Informations

6.1.1 Links

- A full 500 page book about the Linux commandline for free(!): LinuxCommand.org [1]
- Another nice introduction: "A beginner's guide to UNIX/Linux" [2]
- The "commandline starter" chapter of an O'Reilly book: Learning Debian GNU/Linux - Issuing Linux Commands [3]
- A nice introduction to Linux/UNIX file permissions: "chmod Tutorial" [4]
- Linux Cheatsheets [5]
- For the technically interested: Linux Filesystem Hierarchy Standard [6] and Linux Standard Base [7]
- Unix commands applied to bioinformatics [8]
- BioPieces [9]

6.1.2 Real printed paper books:

- Dietz, M., "Praxiskurs Unix-Shell", O'Reilly (highly recommended!)
- 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

[1] http://linuxcommand.org/
[2] http://www.mn.uio.no/astro/english/services/it/help/basic-services/linux/guide.html
[3] http://oreilly.com/openbook/debian/book/ch04_01.html
[4] http://catcode.com/teachmod/
[5] http://www.cheat-sheets.org/#Linux
[6] http://www.pathname.com/fhs/
[7] http://www.linuxfoundation.org/collaborate/workgroups/lsb
[8] http://rous.mit.edu/index.php/Unix_commands_applied_to_bioinformatic
[9] http://code.google.com/p/biopieces

6.1.3 Live - CDs

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

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

Fedora Live CD

This Live CD contains everything the Fedora [10] 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 [11] 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 [12] 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 [13] 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 [14] 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).

[10] http://fedoraproject.org/wiki/FedoraLiveCD
[11] http://knopper.net/knoppix
[12] http://bioknoppix.hpcf.upr.edu
[13] http://www.vigyaancd.org
[14] http://www.bioslax.com

6.2 About Bio-IT

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

6.2.1 Resources

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

6.2.2 Training and Outreach

The "Training and Outreach" menu, bottom left, provides information on events (courses and conferences), both internal to EMBL and organised elsewhere by other organisations, that are related to biocomputing and bioinformatics

6.2.3 Networking

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

6.2.4 Biocomputing expertise at EMBL

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

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

6.3 Acknowledgements

EMBL Logo © EMBL Heidelberg

Bio-IT Logo © Bio-IT Project. EMBL Heidelberg

Graphic of the Linux Filesystem on page 3 from the SuSE 9.2 manual © Novell Inc.

All other graphics © Frank Thommen, EMBL Heidelberg, 2012

CHAPTER 7

INDICES AND TABLES

- *genindex*
- *search*

INDEX