Intermediate Linux Course: Commandline and Basic Scripting

Authors: Holger Dinkel & Frank Thommen
Structural and Computational Biology Unit
Version: 1.1b (May 2013)

Table of Content

Part I: More Commandline Tools

Command-line Tools

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
#
```

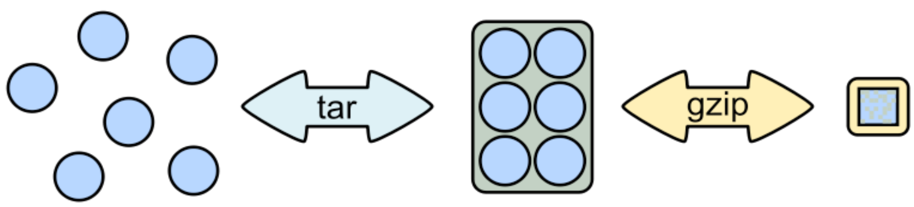
To revert this / to uncompress, use the parameter -d:

```
# 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`"

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:



source: Th0msn80 (Wikipedia)

The most common command-line switches are:

-c	**c**reate an archive
-t	**t**est an archive
-x	e**x**tract an archive
-z	use g**z**ip 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
#
```

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:
-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
#
```

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:

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**' (**i**n-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
#
```

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: **'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
#
```

Hints

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

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 “\”.

Part II: Basic Shell Scripting

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 and situations.

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
[...]
#
```

b) 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 the script has to be provided in this case, no matter wether the script location is already contained in `$PATH` or not:

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

Basic Structure of a Shellscript

Shellscripts have the following general structure:

- 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
- 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.

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, but also human beings will probably have to read and understand your script. See also Breaking up Long Code Lines on page 19.

Better readability means better maintainability!

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 (00README.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 comment use the hash sign (“#”). Everything after a “#” is ignored when executing a script.

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

Shebang line

```
#
# 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
#
```

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

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

Configuration

```
# --- 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
```

Checking prerequisites and sane environment

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

This is what you actually wanted to do

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

Ensure a valid and meaningful exit status

Reporting Success or Failure – The Exit Status

Commands report their success or failure by their exit status. An exit status of 0 (zero) indicates success, while any exit status greater then 0 indicates an error. Some commands report more than one error status. Refer to the respective manpages to see the meanings of the different exit stati.

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

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

See also *Ensuring a Sensible Exit Status* on page 18 about how to control the exit status of your script.

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

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

Confirm that `/etc` exists

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

`cmd1 || cmd2` – execute cmd2 only if cmd1 was ***not*** successful

Warn if a directory doesn't exist

```
# cd /etc || echo "/etc is missing!"

# cd /nowhere >&/dev/null || echo "/nowhere does not exist"
/nowhere does not exist
#
```

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

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

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

{ *cmds*; } – groups commands to create one single output stream. The commands are run in the current (!) shell. The opening "**{**" must be followed by a blank and the last command must be succeeded by a "**;**"

Change into `/etc` and list 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
#
```

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.

Conditional Statements

if – then – else

This is the most basic conditional statement: Do something depending on certain conditions. The basic syntax is

sh/bash

```
if condition1
then
  commands
elif condition2
  more commands
[…]
else
  even more commands
fi
```

csh/tcsh

```
if (condition) then
  commands
else if (condition2) then
  more commands
[…]
else
  even more commands
endif
```

Conditions can be a) the exit status of a command or b) the evaluation of a logical or arithmetic expression:

a) **Evaluating the exit status of a command**: Simply use the command as condition
Example

```
if grep –q root /etc/passwd
then
  echo root user found
else
  echo No root user found
fi
```

To evaluate the exit status of a command in csh/tcsh, it must be placed within curly brackets with blanks separating the brackets from the command: ***`if ({ grep –q root /etc/passwd }) then [...]`***

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.*

Note: Redirection of commands in conditions does not work for csh/tcsh

b) **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)

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

sh/bash *csh/tcsh*

File conditions

sh/bash		csh/tcsh
`-e` *file*	*file* exists	`-e` *file*
`-f` *file*	*file* exists and is a regular file	`-f` *file*
`-d` *file*	*file* exists and is a directory	`-d` *file*
`-r` *file*	*file* exists and is readable	`-r` *file*
`-w` *file*	*file* exists and is writeable	`-w` *file*
`-x` *file*	*file* exists and is executable	`-x` *file*
`-s` *file*	*file* exists and has a size > 0	
	file exists and has zero size	`-z` *file*

String Comparisons

sh/bash		csh/tcsh
`-n` *s1*	String *s1* has non-zero length	
`-z` *s1*	String *s1* has zero length	
s1 `=` *s2*	Strings *s1* and *s2* are identical	`s1 ==` *s2*
s1 `!=` *s2*	Strings *s1* and *s2* differ	*s1* `!=` *s2*
string	String *string* is not null	

Integer Comparisons

sh/bash		csh/tcsh
n1 `–eq` *n2*	*n1* equals *n2*	*n1* `==` *n2*
n1 `–ge` *n2*	*n1* is greater than or equal to *n2*	*n1* `>=` *n2*
n1 `–gt` *n2*	*n1* is greater than *n2*	*n1* `>` *n2*
n1 `–le` *n2*	*n1* is less than or equal to *n2*	*n1* `<=` *n2*
n1 `–lt` *n2*	*n1* is less than *n2*	*n1* `<` *n2*
n1 `–ne` *n2*	*n1* it not equal to *n2*	*n1* `!=` *n2*

Combination of conditions

sh/bash		csh/tcsh
`!` *cond*	True if condition *cond* is not true	`!` *cond*
cond1 `–a` *cond2*	True if conditions *cond1* and *cond2* are both true	*cond1* `&&` *cond2*
cond1 `–o` *cond2*	True if conditions *cond1* or *cond2* is true	*cond1* `||` *cond2*

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

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

or

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

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 (and 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

```
case variable in
  pattern1)
    commands
    ;;
  pattern2)
    commands
    ;;
  [...]
  *)
    commands
    ;;
esac
```

csh/tcsh

```
switch (variable)
  case pattern1:
    commands
    breaksw
  case pattern2:
    commands
    breaksw
  default:
    commands
endsw
```

"", "?" and "[...]" can be used for the patterns*

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.

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

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

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

Loops

for / foreach

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

sh/bash

```
for variable in list
do
  commands
done
```

csh/tcsh

```
foreach variable (list)
  commands
end
```

`list` is a list of strings, separated by whitespaces

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

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

or

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

while / until

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

sh/bash

```
while condition
do
  commands
done


until condition
do
  commands
done
```

csh/tcsh

```
while (condition)
  commands
end


N.A.
```

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

"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

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.

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) but assigned to variables:

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

```
#!/bin/sh

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

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

```
#!/bin/sh

MYDIR=/etc

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

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

Using a Settings File

Instead of having your configurable section within the script, it can be "outsourced" in 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
```

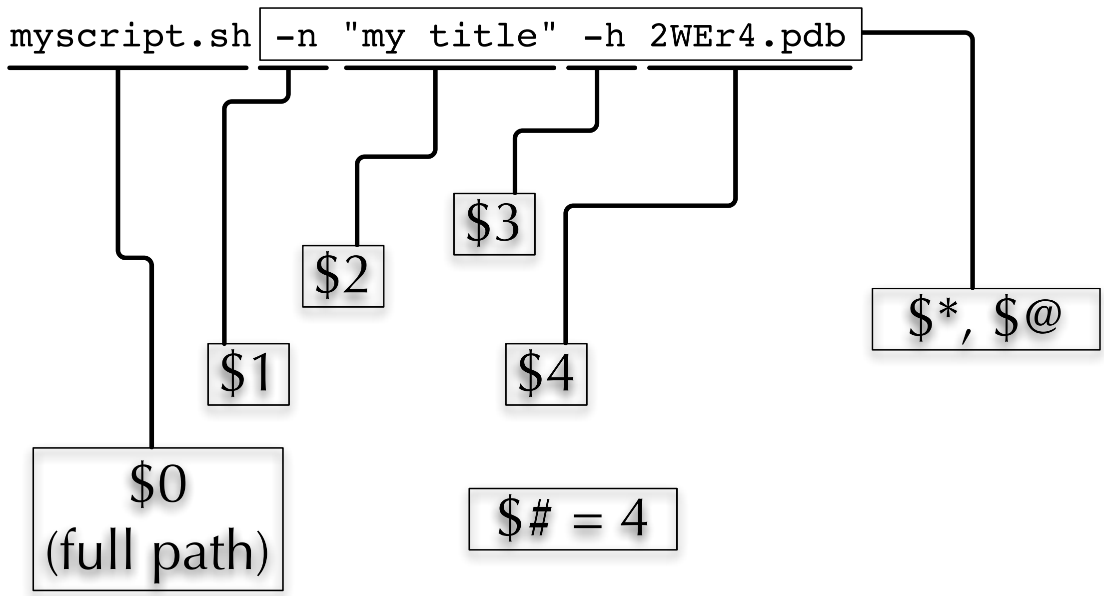
The script:

```
#!/bin/sh

. ./settings.ini

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

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 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"
```

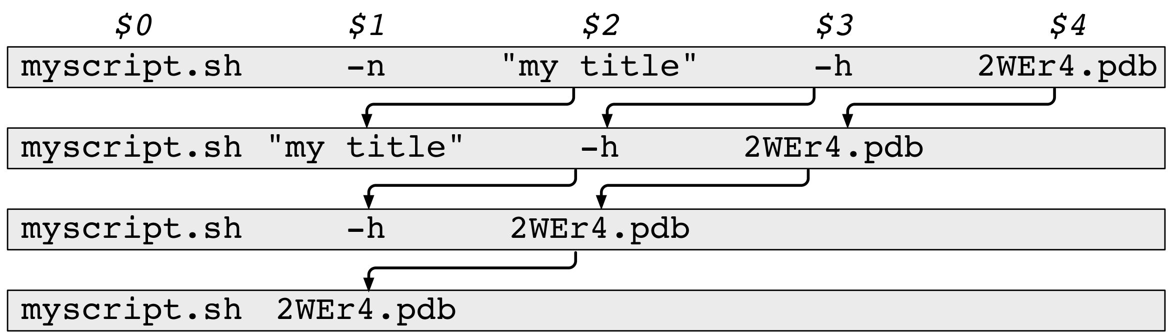
If you run the script

```
#!/bin/sh

echo The script is $0
echo The first commandline option is $1
echo The second commandline option is $2
```

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

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

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



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

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

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

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

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

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 on page 11). In many cases this might be what you want, but very often it isn’t. Consider the following script which is a real example from real life and happened to me personally:

```
#!/bin/sh

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

echo "End of the script"
```

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

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

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

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

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

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

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.

Tips and Tricks

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
#
```

Filenames and Pathes

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

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 enhanve readability of your script. The excape 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/software/bin/pymol-1.4 –r –p < pymol.pml
```

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

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).

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 operators "`" (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
#
```

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
#
```

Cleaning up Temporary Files

It is considerate, 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
```

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 http://bio-it.embl.de. Secondly, we organise and participate in a range of different networking and social activites aiming to strengthen ties across the community.

Links and Further Information

- A full 500 page book about the Linux commandline for free (!): LinuxCommand.org (http://linuxcommand.org/)
- Another nice introduction: "A beginner's guide to UNIX/Linux" (http://www.mn.uio.no/astro/english/services/it/help/basic-services/linux/guide.html)
- The "commandline starter" chapter of an O'Reilly book: Learning Debian GNU/Linux – Issuing Linux Commands (http://oreilly.com/openbook/debian/book/ch04_01.html)
- A nice introduction to Linux/UNIX file permissions: "chmod Tutorial" (http://catcode.com/teachmod/)
- Linux Cheatsheets (http://www.cheat-sheets.org/#Linux)
- For the technically interested:
  Linux Filesystem Hierarchy Standard (http://www.pathname.com/fhs/) and Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb)
- Unix commands applied to bioinformatics (http://rous.mit.edu/index.php/Unix_commands_applied_to_bioinformatics)
- BioPieces (http://code.google.com/p/biopieces/)

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

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 http://fedoraproject.org/wiki/FedoraLiveCD

This Live CD contains everything the Fedora 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 http://knopper.net/knoppix

Knoppix 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 http://bioknoppix.hpcf.upr.edu/

Bioknoppix 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 http://www.vigyaancd.org

Vigyaan 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 http://www.bioslax.com/

BioSLAX 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).

Acknowledgements

EMBL Logo © EMBL Heidelberg
Bio-IT Logo © Bio-IT Project. EMBL Heidelberg
All other graphics unless specifically declared: © Frank Thommen, EMBL Heidelberg

Index