# Unix

## Crash Course
## Exercise Solutions

This document is written as part of a course manual for the Technical Bioinformatics course in the Academic Medical Center in Amsterdam, the Netherlands.
Written by Rob Wolfram (`rsw@hamal.nl`), Unix systems administrator at the AMC.

The latest version of this document is available as both pdf and OpenDocument versions at `http://crash.hamal.nl/`.
The document should be considered an addendum to the syllabus, available at the same site.

The image on the title page originated from “The Unix-Haters Handbook” by Simson Garfinkel, Daniel Weise and Steven Strassman.

Command-line interaction is denoted in a non-proportional font like this:

```
$ echo foo

foo
```

The “dollar space” prefix denotes the “prompt”, i.e. where input from the user is expected.

# Exercises

The exercises presented here suppose that the reader has already familiarized herself with basic file management like copying, renaming, linking and deleting files and directories. The exercises combine many parts of the material and relevant man pages will have to be examined for the specific options. To set up the lab environment, do the following as an ordinary user on a Linux system:

```
wget http://crash.hamal.nl/exercises.sh -O /tmp/exercises.sh
sh /tmp/exercises.sh
```

**Exercise 1:**
What is the syntax of the substitute command to use with **`sed`** to replace all consecutive underscore characters (_) with the text “Mrs. Johnson” in the file “**`letter.txt`**”?

**Exercise 2:**
There is a text file named “**`secret.txt`**” in the directory “**`Deep Secret`**”. What is its content? What is the content of the other file in the same directory?

**Exercise 3:**
How do you delete the file with the name “**`-rf [a-z]*`**”?

**Exercise 4:**
If the binary for the **`ls`** command is unusable, how could you list the content of a directory?

**Exercise 5:**
What is the command line to delete all files in the “**`sizetest`**” directory tree where the files exceed 700 kB in size? What is the command line to move all the remaining files in the same directory tree that exceed 200kB in size to the directory “**`bigfiles`**”?

**Exercise 6:**
Write a script that will “unlink” all files in its arguments that are hard links, i.e. if the file has multiple links, copy it, delete the original and rename the copy to the original name. For all files with single names, print a message that it was already unlinked. Beware of limited disk space! Use files in the “**`dircmp0`**” directory for this exercise.

**Exercise 7:**
Write a script that expects two directory names as arguments. Then calculate the sum of the file sizes of all file names that are present in the first directory but not in the second (hint: look at the “**`-u`**” option in the manpage of **`uniq`**). Use the directories “**`dircmp0`**” and “**`dircmp1`**” for this exercise.

# Solutions

## Exercise 1:

The idea behind this exercise is to demonstrate the trap of *zero* or more characters when using the asterisk in regular expressions. If you would use the syntax:

```
$ sed "s/_*/Mrs. Johnson/g" letter.txt
```

you would get too many replacements. The first two lines would look like this:

```
Mrs. JohnsonDMrs. JohnsoneMrs. JohnsonaMrs. JohnsonrMrs. Johnson Mrs.↩
Johnson,Mrs. Johnson
Mrs. Johnson
```

You need *one* or more consecutive underlines. Basic regular expressions do not contain the **+** character so the proper way to achieve the correct result is by using two underscores followed by an asterisk:

```
$ sed "s/__*/Mrs. Johnson/g" letter.txt
```

## Exercise 2:

There are a number of gotchas in this exercise. The first is that that the directory name contains white space in its name. This should be quoted or escaped to prevent the shell of interpreting the name as two separate arguments:

```
$ ls Deep Secret
ls: cannot access Deep: No such file or directory
ls: cannot access Secret: No such file or directory
$ ls Deep\ Secret
secret.txt
$
```

We see the file but we can't see it's content:

```
$ cat Deep\ Secret/secret.txt
cat: Deep Secret/secret.txt: Permission denied
$
```

and even listing the file permissions is not permitted:

```
$ ls -l Deep\ Secret/secret.txt
ls: cannot access Deep Secret/secret.txt: Permission denied
$
```

The reason is that we can see the directory content but we cannot traverse (i.e. execute) the directory:

```
$ ls -ld Deep\ Secret
drw-r--r-- 2 user10 user10 4096 Nov 14 17:38 Deep Secret
$
```

If we make the directory executable we can verify the file permissions on the file and see its content.

```
$ chmod a+x Deep\ Secret
$ ls -l Deep\ Secret/secret.txt
-rw-r--r-- 1 user10 user10 26 Nov 14 17:38 Deep Secret/secret.txt
$ cat Deep\ Secret/secret.txt

And the answer is...
42

$
```

The second part is that there's another file in the directory. Both **`ls`** and *globbing* consider files that are preceded by a dot as “hidden”. The **`-a`** flag to **`ls`** is needed to see those files.

```
$ ls Deep\ Secret
secret.txt
$ echo Deep\ Secret/*
Deep Secret/secret.txt
$ ls -a Deep\ Secret
.  ..  .more_secrets.txt  secret.txt
$ cat Deep\ Secret/.more_secrets.txt

We know the answer is 42,
for Douglas Adams told us so.

$
```

## Exercise 3:

This is a really dangerous exercise. If you do not use quoting and just issue the command

```
rm -rf [a-z]*
```

every file and directory in your current directory that starts with a lowercase character will be recursively deleted.

```
$ ls
bigfiles  Deep Secret  dircmp0  dircmp1  letter.txt  -rf [a-z]* ↩
sizetest
$ rm -rf [a-z]*
$ ls
Deep Secret  -rf [a-z]*
$
```

If you made this error you can exit the current shell with the “**`exit`**” command and setup a new lab environment with the “**`sh /tmp/exercises.sh`**” command.

The solution to deleting this file is by first quoting or escaping the special characters to prevent the shell from *globbing* and interpreting the white space and then preventing the “**`rm`**” command of interpreting it all as a series op options since the file name starts with a dash. This can be achieved by either including the current directory as part of the file name:

```
$ rm "./-rf [a-z]*"
```

or telling the “**`rm`**” command that all remaining arguments are not options but should be considered directory entries by using the “**`--`**” option:

```
$ ls
bigfiles  Deep Secret  dircmp0  dircmp1  letter.txt  -rf [a-z]* ↩
sizetest
$ rm -- "-rf [a-z]*"
$ ls
bigfiles  Deep Secret  dircmp0  dircmp1  letter.txt  sizetest
$
```

## Exercise 4:

The two primary ways are using *globbing* or using a command like “**`find`**” to traverse the directory tree:

```
$ echo *
Deep Secret bigfiles dircmp0 dircmp1 letter.txt sizetest
$ find . -maxdepth 1 -print
.
./dircmp1
./Deep Secret
./bigfiles
./dircmp0
./letter.txt
./sizetest
$
```

## Exercise 5:

In this exercise we combine the output of the “**`find`**” command to list the files that qualify the constraint and pass that list as arguments to the “**`rm`**” command:
You can use “**`find`**” and “**`wc`**” to count the number of files before and after the action to verify that only a few files have been deleted:

```
$ find sizetest -type f -print | wc -l
50
$ find sizetest -type f -size +700k -print | xargs rm
$ find sizetest -type f -print | wc -l
30
$
```

In fact the second part is just as easy because modern versions of the “**`mv`**” command has a “**`-t`**” argument to specify the destination, so the second part could easily be achieved with:

```
$ find sizetest -type f -size +200k -print | xargs mv -t bigfiles
```

but with older versions of “**`mv`**” we would have to specify the destination *after* all the files. We can achieve that with grouping:

```
$ { find sizetest -type f -size +200k -print ; echo bigfiles ; } |
> xargs mv
$ find sizetest -type f -print | wc -l
9
$ find bigfiles -type f -print | wc -l
21
$
```

The “**>** ” on the second line is a secondary prompt and not part of the input. That is because the first line ended with a pipe, so more commands are expected.

## Exercise 6:

Example script:

```
#!/bin/sh
# Script to unlink the files in its arguments

for file in $@
do
    [ -f "$file" ] || \
        {
            echo "$file is not a regular file, skipping" 1>&2
            continue
        }
    CNT=`ls -l "$file" | awk '{print $2}'`
    if [ "$CNT" -gt 1 ]
    then
        cp "$file" "$file.$$"
        if [ $? -ne 0 ]
        then
            echo "Insufficient space, exiting" 1>&2
            rm "$file.$$"
            exit 1
        fi
        rm "$file"
        mv "$file.$$" "$file"
    else
        echo "$file was already unlinked" 1>&2
    fi
done
```

Some remarks on the script:

- Indenting is not mandatory but it increases readability by indicating the beginning and ending of program flow constructs.
- We use “**$@**” for the list of arguments to make sure that files with white space in the file name are properly processed.
- The “**1>&2**” construct makes the output of “**echo**” go to *stderr*.
- Within the square brackets, quoting is very important. If the variable is not quoted and empty, it leads to a syntax error.

## Exercise 7:

Example script:

```
#!/bin/sh
# Calculate the total of file sizes of files
# present in directory arg1 and absent in directory arg2

usage() {
    echo "Usage: $0 directory1 directory2" 1>&2
    exit
}

if [ $# -ne 2 ]
then
    echo "Incorrect number of arguments" 1>&2
    usage
elif [ \! -d "$1" -o \! -d "$2" ]
then
    echo "One or both arguments is not a directory." 1>&2
    usage
fi

{ ls "$1" ; ls "$2" ; ls "$2" ; } | sort | uniq -u | \
    ( cd "$1" ; xargs ls -ld ) | \
    awk '{sum += $5} END {print sum}'
```

Some remarks on the script:

- The actual work of the script is done by a *one-liner*, something you would type on the command line instead of using in a script. You could also opt for using intermediate output files but that is not necessary here.
- By listing the second directory twice, files that only exist in the second directory and not in the first are filtered out with the “**`uniq -u`**” command.
- By using the parentheses you force the **`cd`** and **`xargs`** commands into a sub-shell that becomes part of the pipe where the first command (**`cd`**) does not consume *stdin* but the second (**`xargs`**) does. Grouping would work here as well.