Using the Linux Virtual Machine

Update:
On my computer I had problems with screen resolution and the VM didn't resize with the window and take the full width when in fullscreen mode. This was fixed with installation of an additional package. In a console window (see below how to pen that), I gave this command:

```
sudo apt-get install virtualbox-guest-dkms
```

This asks for the password. That is "genomics". After restarting the VM, the screen resize works fine.

Some programs are found in the menu at the bottom-left corner.

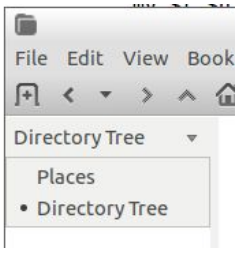
Many programs are command-line only and not in the menu. To use the command line, start the console by clicking the icon in the bottom panel.

Execute the following commands to create a new directory in your home folder "~/" for today's exercises:

```
mkdir ~/session_1
cd ~/session_1
```

Open the file browser to see what you did.

The Shared Folders is under /media. On the file browser you can change to directory view using the pull down menu on the top-left corner:



Click then "media" and "sf_evogeno".

Using RStudio

Click this link:
https://drive.google.com/open?id=0B3Cf0QL4k1-PTkZ5QVdEQ1psRUU
and select to open the file in RStudio. Save the file in the folder "session_1" that you created above.

The script requires two R libraries that are not installed. Install them by typing the following commands in the **R console** window:

```
install.packages("ggplot2")
install.packages("reshape")
```

Now you are ready to run the script e.g. by clicking the "Source" button in the top-right corner of the file editor window. If the script runs fine, the resulting plot is shown in the "Plots" tab in the right-bottom corner. We will get back to this particular script and plot later on the course.

Using BASH

Scripting

Bash is the command language used through the console. One can give bash commands directly in the console, or write them into a script file that is then executed. *Scripting* is a central part of this course and it is crucial to understand the basic commands used.

Type the following command in the console and press enter:
`seq 1 10`

Remember that you can learn more about any Unix command from the manual:
`man seq`
Press q to close the manual.

Do then the same as a script. Start the text editor geany:
```
cd ~/session_1
geany script1.sh &
```
write the command there, save the file and execute it with command:

```
source script1.sh
```

One central concept is a *loop* that does a certain thing multiple times. There are different ways of doing loops but knowing `for, do, done` is a good start. Open another script in geany:

```
geany script2.sh &
```

write the following script there, and execute it:

```
for num in 1 2 3 4 5; do
  echo "number" $num
done
```

We can replace the list of numbers (1 2 3 4 5) with the command `seq 1 5`. This has to be surrounded by backticks ` to execute the command and give its return value. In the VM, backticks are created by keeping Shift down and pressing ` key (next to Backspace in Finnish keyboard twice).

```
for num in `seq 1 5`; do
  echo "number:" $num
done
```

Often we need to loop across multiple files. Let's first create files:

```
for num in `seq 1 5`; do
  echo "number:" $num > file""$num.txt
done
```

Check that the files were created and they have the right content:

```
ls
cat file*.txt
```

Now we do something specifically for those files:

```
for f in `ls file*.txt`; do
  path=`readlink -m $f`
  echo "file name:" `basename $path`
  echo "directory:" `dirname $path`
  b=`basename $f .txt`
  n=$b.text
  mv $f $n
done
```

Check what has happened:

```
ls file*
```

You can reverse the change with command:

```
rename 's/text/txt/' file*text
```

Using *basename* (as above) to replace parts of strings is easier but it has its own limits and sometimes we need to use string operations:

```
for f in `ls file*.t{e,}xt`; do
  n=${f/file/data};
  echo $f $n
done
```

Another important concept is *condition*, for example testing if a file exists or not. Let's first make some files:

```
for num in `seq 1 5`; do
  echo "number:" $num > file""$num.txt
done
```

and then change them conditionally:

```
for num in `seq 1 5`; do
  s=file$num.txt
  l=`basename $s .txt`.text
  if [ -e "$s" ]; then
   mv $s $l
  elif [ -e "$l" ]; then
   mv $l $s
  fi
done
```

Check what happened with `ls file*` and run the script again.

When creating (or debugging) a script, it's useful to print out the commands using "echo":

```
for num in `seq 1 5`; do
  s=file$num.txt
  l=`basename $s .txt`.text
  echo -en "\nround $num: "
  if [ -e "$s" ]; then
   echo "mv $s $l"
   mv $s $l
  elif [ -e "$l" ]; then
   echo "mv $l $s"
   mv $l $s
  fi
done
```

Often it's useful to be able to use the same script e.g. for multiple data sets and be able to give arguments as input for the script. Re-using the first example, let's make a script that takes two arguments:

```
cd ~/session_1
geany script2.sh &
```

Write the commands below in the file and save it:

```
arg1=$1
arg2=$2
for num in `seq $arg1 $arg2`; do
  echo "number:" $num
done
```

Now execute the script with command:

```
source script2.sh 3 8
```

Stream processing

In addition to *scripts* that repeat certain things in a specified order, we use lots of commands that modify or process *text streams*. A central concept are pipes | that take the output of the left command and give as the input for the right command:

```
ls file* | wc
```

Let's get some real data:

```
wget -q --header='Content-type:text/x-gff3'
'http://rest.ensembl.org/overlap/region/human/7:140000000-14010000
0?feature=gene;feature=transcript;feature=cds;feature=exon' -O
genes.gff
```

We'll have a closer at the Ensembl REST later on the course. Now we just use this an example of genomic data file.

"less" is an amazing command. It can show the contents of many types of files, including simple pdfs and compressed files. The most important argument to us is -S that disables line wrapping: our files tend to have very long rows! Compare these two:

```
less genes.gff
```

```
less -S genes.gff
```

Remember that q quits many programs, including "less". Arrow keys can be used to scroll the screen up and down, and left and right.

A gff file is a table with columns separated by tabs. We can see this command:

```
cat -A genes.gff | less -S
```

We could select lines using keywords

```
grep CDS genes.gff | less -S
```

```
grep -e CDS -e gene genes.gff | less -S
```

but often we want to be more specific, e.g. we want lines where 3rd column is "CDS". For that we can use Awk:

```
cat genes.gff | awk '$3 ~ /CDS/ {print $0}' | less -S
```

or

```
awk '$3 ~ /CDS/ {print $0}' genes.gff | less -S
```

Awk is a scripting language of its own and worth learning the basics of. We mostly use it to select some lines and select/modify columns of that. We could write full programs, however, and execute them similarly to Bash programs. Start geany:

```
geany script1.awk &
```

and create and save the following script:

```
BEGIN {
  IFS="\t"
  OFS=":"
}
{
  if($3=="CDS") {c=c+1}
  if($3=="gene") {g=g+1}
  if($3=="transcript") {t=t+1}
}
END {
  print "csd",c
  print "gene",g
  print "transcript:",t
}
```

This script can now be executed as:

```
awk -f script1.awk genes.gff
```

An Awk script consists of three parts: BEGIN is done before anything, middle part is repeated for each line, and END is done in very last. Here we defined input and output field separators (IFS is by default tab so that was unnecessary) in BEGIN; count lines with specific words; and print the results in END.

Often we are lazy and do not write a proper program but a one-liner that does the same thing:

```
awk '{if($3=="CDS"){c=c+1} if($3=="gene"){g=g+1}
if($3=="transcript") {t=t+1}} END {print c,g,t}' genes.gff
```

Normally we are even lazier and use a combination of Bash programs to get the same information:

```
cut -f3 genes.gff | sort | uniq -c
```

On this course one has to understand some Awk but it's not strictly necessary to use Awk to do things. I'm so old that I learned Perl and could do the same using a one-liner:

```
perl -ne '(@r)=split /\t/,$_;if($r[2] eq "CDS"){$c+=1} if($r[2] eq
"gene"){$g+=1} if($r[2] eq "transcript"){$t+=1} END{print
"$c,$g,$t\n"}' genes.gff
```

If someone wants to use Python, that's fine but I won't be able to help.

As mentioned above, Awk has fields separators and then splits a line based on those. The value of first column is in $1, the second in $2 and so on. The full line is in $0. We can use regular expressions outside the main loop to select patterns or conditional statements within the loop. A typical task is to select certain lines and output a set of columns. Sometimes we may want to edit the fields before outputting them:

```
awk '$3 ~ /CDS/ {print $1,$4,$5}' genes.gff | head
```

```
awk '$3 ~ /CDS/ {if($7=="+"){$4-=1000} else {$5+=1000} print
$1,$4,$5}' genes.gff | head
```

Can you spot (and understand) the difference?

Finally, we often combine Awk commands with "sort":

```
grep -v "#" genes.gff | sort -k3,3 -k4,4 | awk '{print
$1,$3,$4,$5}' | less
```

Here, we drop the comment lines, sort the data first by the 3rd column an then by the 4th column. See `man sort` for more details.

Bash programs and commands amazingly powerful and one can't master all of them. If you want to do something, a good starting point is to believe that it can be done and then google for a potential solution. As an example, the following command helps reading large tables in a command-line window:

```
tail -n +3 genes.gff | column -t | less -#2 -N -S
```

Home work

If you great difficulties following these examples, you should immediately take some time and learn the basics of Linux command-line usage. This course will require fluent usage of Bash. Later on the course we will focus on the usage of real analysis programs and have no time to help with the elementary commands.

One can only learn the usage of Linux by using it and trying things. Some useful links to learn the basics are:

AWK community portal: http://awk.info/?Learn
BASH from basics to scripting: https://bash.cyberciti.biz/guide/Main_Page
BASH official beginner's guide: http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html
R-bloggers: http://www.r-bloggers.com/how-to-learn-r-2/
R graphical intro: http://tryr.codeschool.com/

Advanced topic: Using R Markdown with RStudio

Markdown is an easy format to write impressive-looking documents. R Markdown combines the Markdown language with embedded R code and graphics.

If you want to use R Markdown, you need to install additional R packages. These are automatically installed by RStudio if you select "File" -> "New File" -> "R Markdown". The installation takes a few minutes. When it is ready, RStudio automatically proposes to create an example document: fill in the details and select "html_document" as the output. The document is compiled using the "Knit" button in the top panel.

R Markdown becomes even better when documents are outputted as pdf. This requires LaTeX, however, and that takes ~1Gb of disk space. You can install the necessary packages with:

```
sudo apt-get install texlive texlive-latex-extra pandoc
```

You can then change the output format to "pdf_document" and recompile the document. An example of R Markdown document can be found here:
https://drive.google.com/open?id=0B3Cf0QL4k1-PNURxVEhKdUQ5RUk
The resulting pdf looks like this
https://drive.google.com/open?id=0B3Cf0QL4k1-PMXNGWjRaNzh6UkE

(The compilation requires additional steps that are either explained in the Rmd document or revealed by the error messages of Bash commands.)

More information about R Markdown can be found here
http://rmarkdown.rstudio.com/