Bioinformatics Lab Course, Seminar: Command Line Starter Kit

Asya Mendelevich
Skoltech, 30 Oct 2017

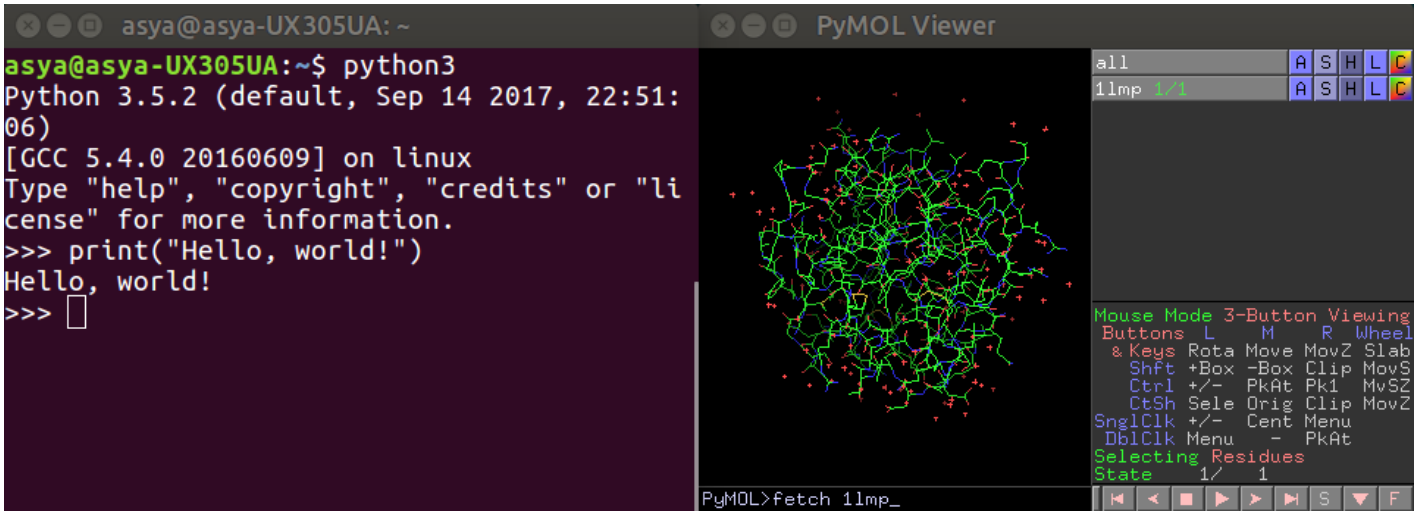
Contents

1 Get started

1.1 Command Line Concept

Where command line interface can be met

- Terminal;
- Programms and tools;
- Computer games (for example, Quake).



Permissions `sudo` – "substitute user and do", allows to run programs with the security privileges of superuser.

1.2 Safety Regulations

- avoid using unknown commands and blindly following instructions from the Internet;
- avoid making mistakes, could lead to irrevertible damage (confusing `sudo rm -r ./*` and `sudo rm -rf /*` may be dramatic);
- be careful with sudo, if a command needs sudo rights, it is dangerous enough;
- be careful with updates due to potential package incompatibility;

1.3 Installation

Linux and MAC Nothing to do, everything is already installed.

Windows Terminal is already here. But you need to install `putty` to be able to connect to a remote server:

1. download msi-file from https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
2. install, following the instructions of the installer

1.4 File System Structure

It is almost true that the file system structure is (almost) a tree, i.e. a hierarchical system of folders, and that the paths looks like:

`/home/user/Documents/folder1/folder2/myfile` or `/home/user/Documents/folder1/folder2/`,

lists of nested folders (directories) divided by slash. The last element may be either a file or a folder.

Absolute path The full path to the object, starting from root.
Relative path The path to the object, starting from current directory.

`/`	**Reference to the root directory**
`~`	**Alias for your home directory path**
`.`	**Reference to the current directory**
`..`	**The parent directory of the current directory**

2 Shortcuts

Helpful shortcuts for navigating in terminal:

`Tab`	**Automatically complete your current input, if possible**
`Ctrl+c`	**Stop the execution of the current command**
`Ctrl+d`	**Exit the shell (= `exit`)**
`UP`	**Previous command in history**
`DOWN`	**Next command in history**
`Ctrl+r`	**Search history**
`history`	**Show command history**
`<n>`	Number of last commands to show
`Ctrl+a, Home`	**Go to the start of the input line**
`Ctrl+e, End`	**Go to the end of the input line**
`Ctrl+u`	**Cut from start of line**
`Ctrl+k`	**Cut to end of line**
`Ctrl+l`	**Clear (= `clear`)**
`Ctrl+z`	**Sleep program**

3 Packages

A software package is an archive file containing a computer program as well as necessary metadata for its deployment. Packages may contain either a source code that needs to be compiled, or executable files for some specific purpose.

`apt-get install <package name>`	**Install package, may require sudo rights**
`apt-get remove <package name>`	**Remove package**
`apt-get update`	**Fetches the list of available updates**
`apt-get upgrade`	**Upgrade packages**

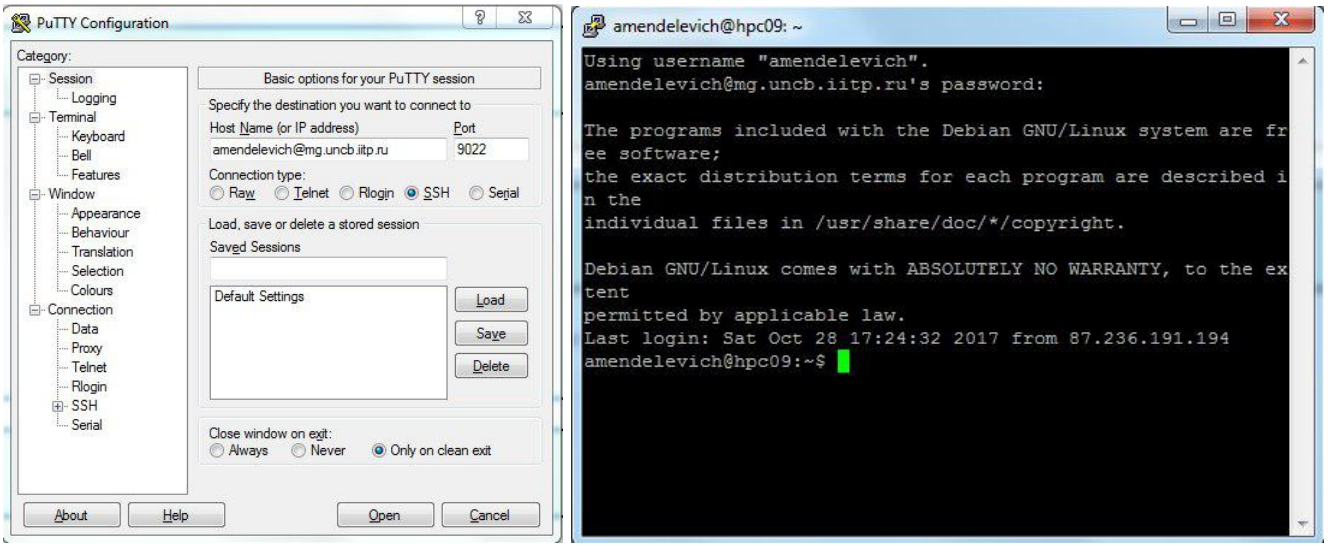
4 Remote work

A remote server is a distant machine, usually much more powerful than a personal computer, that can be reached by the means of remote access for performing various tasks. **Secure Shell (SSH)** is a cryptographic network protocol, the best known example application is for remote login to computer systems by users.

`ssh <user@host>`	**Connect to host as user**
`-p <port>`	Connect to host via a specific port
`scp`	**Transfer files securely over an SSH connection**
`<user@host:path> <local folder>`	From the server to the local machine
`<local path> <user@host:folder>`	From the local machine to a server folder
`-r`	Recursively (for folders)
`-P <port>`	Via a specific port

For our seminar:

- **Linux and MAC**: Open terminal and write `ssh <your_name>@mg.uncb.iitp.ru -p9022`
- **Windows**: Open Putty and fill two fields — Host Name (`<your_name>@mg.uncb.iitp.ru`) and Port (`9022`)



5 Commands and flags

The main purpose of the command line is to call external programs, e.g. `/usr/bin/ls`. You can reference most of them just by the name of the executable, in our example, just by typing `ls`. The output produced by the program is then translated into the console.

But sometimes you want to give a program some input. Different programs may use slightly different conventions for how to do that, but generally everything you give after the program name and until the end of line (or some special symbol like `>`) is treated as the arguments. For some programs, you can give an argument simply by appending something to the program name, after a space: `python mycode.py` (we give the name of the file we want `python` to interpret), `ping 8.8.8.8` (we want to `ping` Google Public DNS), `unzip ~/Downloads/how_to_get_rich_no_viruses.zip` (we want to extract an archive located in our Downloads directory).

If a program is more sophisticated, it may expect a number of arguments. Some of them take them one-by-one in some specific order, like `grep pattern outputfile.txt`. Often a program provides the names for the arguments, e.g. `-o` (often stands for output file), `-i`, and so on. Then you can call it like `iconv -f cp1251 -t utf8 inputfile` (we want to change the encoding of `inputfile` **f**rom cp1251 **t**o utf8).

Finally, *a flag* stands for a way to alter the default behaviout of a program somehow, e.g. `ls -a` shows **a**ll files, including the hidden ones.

To find out how you can interact with a particular program, you should check its manuals, or use Google.

`google`	**Just google it**
`man <cmd>`	**Manual for a command (uses `less` $\Rightarrow$ q key for quit)**
`<cmd> --help (-h)`	**help**
`<cmd> --version (-v)`	**version**

5.1 Disk usage

Helpful commands for disk space usage control:

`du`	**Estimate space usage**
`<dir>`	In directory `dir`
`-a`	For all files, not just directories
`-d <n>`	For n fewer levels
`-h`	Human-readable
`-s`	Summurise, total
`df -h`	**Report disk space usage**

```
df -h
du -sh
```

5.2 Processes

Helpful commands for managing running processes:

`top`	**Display processes**
`htop`	**Display processes, interactive**
`pgrep <name>`	**Display IDs for processes with name `name`**
`kill <pid>`	**Kill process with id `pid`**
`pkill`	**Kill process**
`<name>`	with name `name`
`-u <user>`	of a given user `user`

```
top
```

5.3 Time

`time <cmd>`	**Print command execution time**
`date`	**Print date and time**
`sleep <n>`	**Sleep for `n` seconds**

```
time du -sh ..
date
sleep 3
```

5.4 Files and directories

`echo <smth>`	**Print something**

```
echo Hello, world!
echo "Hello, world!" 179
```

`pwd`	**Show current directory**
`cd`	**Change directory**
`<dir>`	Change directory to `dir`
`..`	Parent directory
`-`	Previous directory
`/`	Root
`ls`	**List all files**
`<dir>`	In directory `dir`
`-l`	Long listing format, with information
`-a`	All, including hidden (starting with .)
`-t`	Sorted by modification time, newest first
`-S`	Sorted by file size
`-1`	One file per line
`mkdir <dir>`	**Create a directory**
`-p`	With all the parent directories

```
pwd
mkdir seminar
mkdir -p /home/<your_name>/seminar/f1/f2
ls
ls -l
cd ./seminar
cd -
cd /home/<your_name>/seminar
```

`locate <name>`	**Find all instances of file by name**
`find <dir>`	**Find file in directory** `dir`
`-name <name>`	With the filename being `name`
`-user <user>`	Owned by `user`
`whereis <app>`	**Show possible locations of app**
`which <app>`	**Show which app will be run by default**

```
ls /mnt/local/students/shared/seminar_command_line
find /mnt/local/students/shared/seminar_command_line -name interproscan.out
du -a /mnt/local/students/shared/seminar_command_line
find . -name f2
find /mnt/local/students/shared/ -name <your_name>
whereis ls
which cd
which pwd
which which
whereis which
```

`cp <source> <dest>`	**Copy the source file to the destination directory**
`-r`	Recursively (for folders)
`mv <source> <dest>`	**Move the source directory or file to the destination directory or file** `dest`
`ln -s <file> <link>`	**Create symbolic (soft) link** `link` **to** `file`
`rm <path>`	**Remove a file or directory**
`-r`	Recursively (for folders)
`-f`	Force
`touch <file>`	**Update if exists, otherwise create a blank file**

```
ln -s /mnt/local/students/shared/seminar_command_line/3/2/interproscan.out interlink
cp /mnt/local/students/shared/seminar_command_line/3/2/interproscan.out ~
mv ../interproscan.out .
ls -l ~
mv interproscan.out ./interpro
touch ~/new_file
rm new_file
rm ~/new_file
du -d 2 /mnt/local/students/shared/seminar_command_line/
cp /mnt/local/students/shared/seminar_command_line/4/5/GoatFishDog .
```

`file <file name>`	**Determine file type**
`wc`	**Print number of lines and words, and byte counts**
`-l`	Number of lines only
`-w`	Numder of words only
`head`	**Print the first 10 lines of a file**
`<n>`	..or first n lines
`tail`	**Print the last 10 lines of a file**
`<n>`	..or last n lines
`cat <f1> .. <fN>`	**Concatenate files and print them**
`less <file name>`	**View file content (q for quit)**

```
file interpro
file interlink
file GoatFishDog
head interpro
tail -5 interpro
head -n 2 interlink
wc -l interpro
wc -l GoatFishDog
less interpro
cat GoatFishDog
cp GoatFishDog ./GoatFishDog2
cat GoatFishDog GoatFishDog2
```

`sort`	**Sort lines alphabetically**
`-r`	Reverse sort
`-n`	Sort numerically
`uniq`	**Report repeated (sequentually) lines**
`-c`	With counts
`-u`	Only unique lines
`-d`	Only duplicates

```
sort GoatFishDog
uniq GoatFishDog
```

5.5 Pipes and Redirection

`<cmd> > <file>`	**Redirect stdout to a file**
`<cmd> >> <file>`	**Append stdout to the end of a file**
`<cmd> < <file>`	**Pass the contents of a file to a program as stdin**
`<cmd1> | <cmd2>`	**Feed the stdout of comd1 as stdin to cmd2**
`<cmd1> ; <cmd2>`	**Run cmd1 then cmd2**

```
sort GoatFishDog; uniq GoatFishDog
sort GoatFishDog | uniq
echo "Hello" >> redirect; cat redirect
echo "Hello" >> redirect; cat redirect
echo "..." > redirect
echo "Hello" >> redirect; cat redirect
```

5.6 Variables

`alias`	**Set short-name command variable**
`<short>=<long>`	
`<var name>=<expr>`	**Bind the expression value to a variable (use `$<var name>` to access the value)**

```
interpath='/mnt/local/students/shared/seminar_command_line/'
$interpath
alias f=find
f $interpath -name interproscan.out
```

5.7 Regular Expressions

Regular expressions help to create the mask that will fit the wanted names:

`*`	**Zero or more characters**
`?`	**Any character**
`[abc]`	**Range of characters, any of them**
`[^abc]`	**Not a range, a character that is not one of those**
`\`	**Removes a special meaning of a character**
`^`	**Start of the line**
`$`	**End of the line**
`<`	**The beginning of a word**
`>`	**The end of a word**

5.8 grep, sed, cut, awk

Note that `grep, sed` utilities are true monsters and `awk` is a programming language, so we will consider only small amount of functionality here.

All of them eat text file as an input (line by line) and returns stdout by default.

cut Tool used to extract fields from lines of text input.

`cut`	**Extract sections from each line of files**
`-c <list>`	Characters
`-f <list>`	Fields (default delimiter: `tab`)
`-d <list>`	Set delimiter
`--complement`	Complement

Where `list` is denoted as:

`<n>` – n-th element

`<n>-<m>` – elements from n to m

`-<n>` – remove elements till n, inclusively

`<n>-` – remove elements after n, inclusively

`<n>,<m>` – elements n and m

```
cut -f1,6 interproscan.out | head
cut --complement -f2-5 interproscan.out | head
```

sed Stream editor that parses, filters and transforms text files.

`sed`	**Filter and transform text file**
`-e`	Indicate that an expression follows
`-i`	In-place editing
`'s/<from>/<to>/'`	Substitution
`'s/<from>/<to>/g'`	Global (all matches in each line) Substitution
`'/<expr>/d'`	Delete lines that match a regular expression
`'/<expr>/!d'`	Delete lines that do not match a regular expression
`'<n>d'`	Delete line `n`

Where:

`*` – zero or more occurrences of the previous character

`<a>,<b>` – from `a` to `b`

```
sed 's/is/IS/' GoatFishDog
sed 's/is/IS/g' GoatFishDog
sed '/^ /d' GoatFishDog
sed '/^ *$/d' GoatFishDog
sed '/^ /d; s/my//g' GoatFishDog
sed '1,4d; $ d' GoatFishDog
sed -e '3d' -e '3d' GoatFishDog
```

grep (**g**lobally search a **r**egular **e**xpression and **p**rint) Utility for searching through texts for lines that match a regular expression.

`grep <pattern>`	**Print lines matching a pattern**
`-i`	Ignore case (ie uppercase, lowercase letters)
`-c`	Print a count of matching lines
`-v`	Return all lines which don't match the pattern
`-w`	Select only matches that form whole words
`-n`	Print the line number before each line that matches
`-r`	Recursive, read all files in given directory and subdirectories
`-l`	Print the name of each file which contains a match

```
grep -ciw "domain" interpro
grep -cw "domain" interpro
grep -c "domain" interpro
```

awk A programming language for text processing.

`awk`	
`-F'<delim>'`	Set a delimiter
`'print $<n>'`	Print `n`-th column (`print $0` will return whole file)
`'print $<n> <smth> '`	Print `n`-th column and `smth`
`'$<n>=<smth>'`	Replace all values by `smth`

```
awk '{print $5,4}' interpro | sort | uniq
awk -F'\t' '$2=" "' interpro | head
```

5.9 Archives

One of the way of compactification of data is archiving. But in the most of cases working directly with compressed files is impossible, so they need preliminary decompression before use.

`tar <file>`	**Archiving utility**
`c`	Create archive
`x`	Extract from archive
`f <name>`	Use archive file name `name`
`z`	gzip compressed tar
`j`	bzip2 compressed tar
`zip`	**Archiving utility**
`<arch> <file>`	Compress file
`-r <arch> <dir>`	Compress folder recursivelly
`unzip <arch name>`	**Uncompressing utility**
	Uncompress file to current directory
`-d <dir>`	Uncompress file to directory `dir`
`gzip`	**Compressing program**
`<file>`	Compress file
`-k <file>`	Compress file and keep uncompressed copy
`-r <dir>`	Compress folder recursivelly
`-<n>`	Set the compression level (from 1 to 9)
`-d <name>`	Decompress (the same with `gunzip <name>`)

```
zip redirect
gzip redirect
ls -l redirect*
gunzip redirect
ls -l redirect*
```

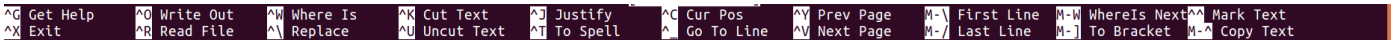
5.10 Networking

If some file can be received by the exact link, it can be downloaded with `wget`:

`wget <adress>`	**Download `adress` content**
`-c`	Continue a stopped download

6 nano

`nano` is a simple console editor, its main advantage is build-in help lines at the bottom of the window (^ means `Ctrl+`):



`nano` is so user-friendly that even asks you questions in uncertain situations. Arrows can be used for navigation.

`nano`		**Open an empty file**
	`<name>`	Open the file `name`
	`-v`	Open in read-only mode
`Ctrl+`		
	`G`	Get Help
	`O`	Write Out current file
	`X`	Exit
	`W`	Search for a string or a regular expression
	`K`	Cut the line
	`U`	Uncut Text
	`C`	Display the position
	`R`	Insert another file into the current one
	`Y`	Prev Page
	`V`	Next Page
	`^`	Mark text at the cursor position
	`\`	Replace a string or a regular expression
	`_`	Go to line and column number
`Alt+`		
	`\`	Go to first Line
	`/`	Go to last line
	`W`	Repeat last search
	`^`	Copy Text
	`U`	Undo
	`E`	Redo

7 Scripts

The command line you use is just an interpreter of `bash`. So, you can save a sequence of commands to a file and run it later. Such files are called scripts.

Imagine the following file `simplescript`:

```
echo "Hi!"
echo "A secret message" > secretfile
echo "Buy!"
```

As you can see, it is just the sequence of commands you can type into your command line. To run the commands one-by-one, we do `bash simplescript`. But that is tedious. Instead, in the beginning of `simplescript`, we add the following line:

```
#!/usr/bin/env bash
```

and then run it by `./simplescript`. The command line sees that in the first line we ask to run the script in `bash`.

One small detail: if we want to run the script in the latter way, it should have executable permissions:

`chmod`		**Change file mode**
	`+`	Add
	`-`	Remove
	`r`	Read
	`w`	Write
	`x`	Execute

Then it can be run by `<path to script>` command.

Another advantage is that we can specify any interpreter we want. For example, this line

```
#!/usr/bin/env python
```

asks to run the script in the `python` interpreter.

We will not be practicing real programming now, and consider only simple starter-kit stuff.

Parameters

- Are written space-separated after the script call.
- Use `$N` to access `N`-th parameter.

```
#!/bin/bash

echo "Hello, $1!"
```

Variables

- Set with `variablename=astring`.
- Use `variablename=$(acommand)` to capture the stdout of a command into the variable.
- Use `$variablename` to access the value.

if

- Runs a code if a condition is true
- Starts with `if <condition>; then` and ends with `fi`.
- If there is a set of conditions use `elif` and `else`.

```
if [ <cond> ]; then <smth>; elif [ <cond2> ]; then <smth2>; else <smth3>; fi
```

for

- Iterate over a series of words.
- Starts with `for <series>; do` and ends with `done`.

```
for i in <ser>; do <smth>; done
```

while

- Iterate while true.
- Starts with `while <condition>; do` and ends with `done`.

```
while [ <cond> ]; do <smth>; done
```

Note: may introduce an infinite cycle, be cautious!

`$[ <smth> ]` – result of `smth`.
`[[ <smth> ]]` – true-false result of `cond`.
`=, ==, -eq` – equality
`!=, -ne` – inequality
`<, -lt` – less
`<=, -le` – less or equal
`>, -gt` – greater
`>=, -ge` – greater or equal
`!` – logical negation
`&&, -a` – logical and
`||, -o` – logical or

Note: follow the spacing rules.

```
for i in {1..5}; do echo $i; done
END=5
for i in `seq 1 $END`; do echo $i; done
for i in `seq 1 $END`
do
echo $i
done

while [ $END != 10 ]; do echo $END; END=$[ $END + 1 ]; done

for word in $( cat GoatFishDog ); do
    if [ $word = 'is' ]; then echo 'si'
    elif [ $word = 'Adam' ]; then echo 'Eva'
    else echo "..."
    fi
done

for word in $( cat GoatFishDog ); do if [ $word = 'is' ]; then echo 'si'; elif [ $word = 'Adam.' ];
    then echo 'Eva'; else echo "..."; fi; done
```

8 TMUX

`tmux` is a terminal manager, especially helpful for creating persistent sessions on a remote server (if the connection is lost, you can log back and safely return to the stuff you were doing).

`tmux`	**Start new session**
`new -s <name>`	Set name of the session
`tmux a`	**Attach to the last session**
`-t <name>`	Attach to named session
`tmux ls`	**List of sessions**
`tmux kill-session`	**Kill session**
`-t <name or id>`	Kill session by name or id
`Ctrl+b`	**Common prefix for commands inside the session**
`d`	Detach
`t`	Big clock
`?`	List shortcuts
`s`	List of sessions
`c`	Create window
`w`	List windows
`n`	Next window
`p`	Previous window
`%`	Vertical pane split
`"`	Horizontal pane split
`ARROWS`	Switch between panes
`Ctrl+d`	**Kill pane or window or session**

9 GIT

A version control system for tracking changes in computer files and coordinating work on those files among multiple people.

`git`	**Revision control system**
`clone <adress>`	Clone a repository into a new directory
`pull`	Fetch from and integrate with another repository or a local branch
`add <file>`	Add file contents to the index
`commit -m <text>`	Record changes to the repository with message `text`
`push`	Update remote refs along with associated objects

```
git clone <repository>
cd <git_dir>; git push
```

Seminar Task

1. Create a new directory with the name `/seminar/task` in your home directory. It will be your working directory now.
2. Follow the link

   ftp://ftp.ncbi.nlm.nih.gov/pub/clinvar/vcf_GRCh38/

   Find `clinvar_20171002.vcf.gz` find. How to obtain the direct link to it?
   Download `clinvar_20171002.vcf.gz` to your new directory.
3. What is the size of the obtained archive? Unpack it. What is the size of the uncompressed vcf-file?
4. What is the structure of the file?
5. Count rows in the vcf-file that starts with `#` (header and info). Count content rows.
6. Save info-rows to a new file and remove them from `clinvar_20171002.vcf`.
7. Cut `CHROM, POS, REF` and `ALT` columns from `clinvar_20171002.vcf` and save somewhere.
8. Find the most mutable `REF` nucleotide in 15 chromosome.
9. Find the most prevalent nucleotide pair `REF-ALT` in 15 chromosome.

Homework

1. Clone GitHub repository https://github.com/Strausyatina/Skoltech_Seminar.git to your home directory.
2. Create a new directory `homework` in your home directory.
3. Make a symbolic to `SkHomeWork.bam`, located in `/mnt/local/students/shared/seminar_command_line/HW/`, in your `homework` directory.
4. Using `samtools`, convert bam-file to sam-file, sorted by read names (Check `samtools` version before looking for manual)
5. Get rid of 'chr' prefix in chromosome names in sam-file.
6. Write your own executable bash-script `RG_chr_counter.sh` that:
   - takes sam-file as single input;
   - streaming the content of tsv-file without header, with 3 columns:
     - chromosome number (with order: 1, $\cdots$,19, X,Y)
     - number of reads with Read Group ID: 1
     - number of reads with Read Group ID: 2

     to stdout.
7. Save the output of your script eating your sam-file to `RG_chr.tsv`.
8. Run the script from GitHub on your data (see help (`-h, --help`)). If the answer is 'ok' then:
9. Save the subset of your `history` that leads to victory to `hw_history`.
10. Clean your workspace, only the following should survive
    - sam-file
    - `RG_chr.tsv`
    - `RG_chr_counter.sh`
    - `hw_history`

    in your homework directory.