Linux shell tutorial

Linux Shell tutorial by Science IT at Aalto University.

This course consists of two parts: Linux Shell Basics and Linux Shell Scripting. The first one covers introductory level shell usage (which also is a backdoor introduction to Linux basics). The second one covers actual BASH scripting, using learning by doing.

Starred exercises (*) are for advanced users who would like further stimulation.

⚙ Prerequisites

- Part #1: A Linux/Mac computer or a Windows with SSH client installed to access any Linux server
- Part #2: Shell basics, know how to create a directory and edit a file from command line

Who is the course for?

- Part #1 for those who need a comprehensive dive into shell basics.
- Part #2 for people with intermediate/advanced level of Linux/Mac shell

Schedule

Ongoing. Optional lectures on average once per year with following timetable:

Linux Shell Basics	3 sessions x 3h
Linux Shell Scripting	4 sessions x 3h

References

Based on

- `man bash` v4.2 (Triton default version in Feb 2018)
- Advanced BASH scripting guide
- UNIX Power Tools, Shelley Powers etc, 3rd ed, O'Reilly
- common sense and 20+ years Linux experience
- see also other references in the text

Videos

Videos of previous instances:

- Part 1, 2023: Playlist
- Part 2, 2024: Playlist

Basic shell

First touch: getting a BASH shell

Set yourself up with a BASH shell. Connect to a server or open on your own computer. Examples and demos given during the lecture are done on a native Linux BASH on either Ubuntu or CentOS, though should work on all other Linux installations and Git BASH on Windows.

- Linux and Mac users: just open a terminal window
    - Ubuntu users hint: Ctrl + Alt + t
- Windows users:
    - (Priority #1) SSH to a remote native Linux installation. For aalto users [1].
        - Use built-in SSH client: open a Command Prompt (cmd) and launch a command 'ssh loginname@remote.server.name'
        - If no built-in client, install PuTTY [2].
    - (For easy testing) Git BASH [3] // Misses man pages, some utilities.
    - -or- VDI if such service available. Aalto user, see Remote desktop at [1].
    - -or- Windows Subsystem for Linux (WSL 2) and Ubuntu on Windows [4].
    - -or- Cygwin [5].

About the Linux Shell

- A *shell* is what you get when your terminal window is open. It is a command-line (CLI), an interface that interpreters and executes the commands.
- The name comes from being a "shell" (layer) around the operating system. It connects and binds all programs together.
- This is the basic, raw method of using UNIX-like systems. It may not be used everyday, but it's really good (necessary) for any type of automation and scripting - as is often needed in science, when connecting pieces together, or when using Triton.
- There are multiple shells. This talk is about bash, which is the most common one. zsh is another common shell which is somewhat similar but has some more powerful features. tcsh is another shell from a completely different family (the csh family), which has quite different syntax.
- `bash` is a "Bourne shell": the "bourne-again shell". An open source version of original Bourne shell.

- It may not be obvious, but the concepts here also apply to Windows programs and will help you understand them. They also apply more directly to Mac programs, because Mac is unix under the hood.

Basic shell operation

- You type things on the screen (standard input or stdin). The shell uses this to make a command.
- The shell takes the command, splits it into words, does a lot more preprocessing, and then runs it.
- When the command runs, the keyboard (still standard input) goes to the process, output (standard output) goes to the screen.

[1] (1, 2) https://scicomp.aalto.fi/aalto/remoteaccess/

[2] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

[3] https://gitforwindows.org/

[4] https://www.microsoft.com/en-us/p/ubuntu/9nblggh4msv6

[5] https://www.cygwin.com/

Starting out

Starter

```
du -hs * .[!.]* | sort -h
tar czf - $d | ssh kosh.aalto.fi 'cat > public_html/$d.tar.gz && chmod a+r $d.tar.gz'
find $d -type f \( ! -perm /g+w  -o -perm /o+w \) -exec chmod u+rwX,g+rwX,o-wx {} \;
```

A minimum to get started

Try them right away:

```
whoami (-or- id)
echo $SHELL
uname -a (-or- hostnamectl if available)
pwd
ls -lA
cd
date
grep searchword filename
cat filename
```

In combination with opeators > and | serves, plus text editor and a viewer like *less*, you should feel yourself safe already now.

For Aalto users: is your default shell a `/bin/bash`? Login to kosh/taltta and run `chsh -s /bin/bash`

Getting help in terminal

Before you Google for the command examples, try:

```
man command_name
```

Your best friend ever – `man` – collection of manuals. Type */search_word* for searching through the man page, navigating between matches with *n* and *shift + n*, *q* for the exit.

Additionally, many, but not all, commands have a usage summary if run with `... --help` or `... -h` options.

File viewing / editing

```
cat filename
less filename  # 'q' to exit
nano filename  # Ctrl-x to exit
```

The most common commands

The ones we use in this tutorial, subjective:

```
man, cd, pwd, echo, whoami, id, hostname, bg, fg, jobs, cat, less, ps, kill, top,
pgrep,
pstree, htop, du, cd, ls, mkdir, touch, ln, rm, cp, mv, mkdir, find, rsync, tar, scp,
ssh,
alias, set, umask, export, date, clear, head, tail, wc, grep, sort, uniq, tr, diff,
killall,
gzip, nano, xargs, chown, chmod, su, sudo, sleep, read, type, file, ping, ...
```

Plus shell programming language constructs adn control operators.

Important remark: not all of the external commands are available on all the systems. Even Linux distribution bundles may differ not speaking of the macOS setup and MS Windows packages.

Hint `type -a` to find binary location on the filesystem.

(*) Built-in and external commands

There are two types of commands:

- shell built-in: `cd`, `pwd`, `echo`, `alias`, `bg`, `set`, `umask` etc.
- external: `ls`, `date`, `less`, `lpr`, `cat`, etc.
- some can be both: e.g. `echo`. Options not always the same!
- For the most part, these behave similarly, which is a good thing! You don't have to tell which is which.
- **echo**: prints out `echo something to type` # types whatever you put after

Disable built-in command `enable -n echo`, after this */usr/bin/echo* becomes a default instead of built-in *echo*

Processes

What's a UNIX process?

- To understand a shell, let's first understand what processes are.
- All programs are a process: process is a program in action.
- Processes have:
  - Process ID or PID (integer)
  - Name (command being run)
  - Command line arguments/options
  - input and output: `stdin` (input, like from keyboard, another program, another device), `stdout` (output, like to screen, another program, file, device), `stderr` (like stdout but for errors)
  - Return code (integer) when complete
  - Working directory
  - environment variables: key-values which get inherited across processes.
- These concepts bind together *all* UNIX programs, even graphical ones.

Process listing commands (feel free to try, but we play more with them later):

```
ps auxw
top              # (q to quit)
htop             # (q to quit)
pstree
pstree $USER
pstree -pau $USER
```

Another way to find out what SHELL you are running:

```
ps -p $$
```

Working with processes

All processes are related, a command executed in shell is a child process of the shell. When child process is terminated it is reported back to parent process. When you log out all shell child processes terminated along with the shell. You can see the whole family tree with `ps af`. One can kill a process or make it “nicer”.

```
pgrep -af <name>
kill <PID>
pkill <name>
renice #priority <PID>
```

Making process “nicer”, `renice 19 <PID>`, means it will run only when nothing else in the system wants to. User can increase nice value from 0 (the base priority) up to 19. It is useful when you backup your data in background or alike.

Foreground and background processes

The shell has a concept of foreground and background processes: a foreground process is directly connected to your screen and keyboard. A background process doesn’t have input connected. There can only be one foreground at a time (obviously).

If you add `&` right after the command will send the process to background. Example: `firefox --no-remote &`, and same can be done with any terminal command/function, like `man pstree &`. In the big picture, the `&` serves the same role as `;` to separate commands, but backgrounds the first and goes straight to the next.

If you have already running process, you can background with Ctrl-z and then `bg`. Drawback: there is no easy way to redirect the running task output, so if it generates output it covers your screen.

List the jobs running in the background with `jobs -l` (show real PID as well), get a job back online with `fg` or `fg <job_number>`. There can be multiple background jobs.

Kill a foreground job: Ctrl-c

Hint: For running X Window apps while you logged in from other Linux / MacOS make sure you use `ssh -X ...` to log in. For Windows users, you need to install VcXsrv Windows X Server [1] on your workstation.

Hint: For immediate job-state change notifications, use `set notify`. To automatically stop background processes if they try writing to the screen `stty tostop`

Exit the shell and ‘screen’ utility

`logout` or Ctrl-d (if you don’t want Ctrl-d to quit, set `export IGNOREEOF=1` in *.bashrc*).

Of course, quitting your shell is annoying, since you have to start over. Luckily there are programs so that you don't have to do this. In order to keep your sessions running while you logged out, you should learn about the `screen` program.

- `screen` to start a session
- *Ctrl-a d* to detach the session while you are connected
- `screen -ls` to list currently running sessions
- `screen -rx <session_id>` to attach the session, one can use TAB for the autocompletion or skip the <session_id> if there is only one session running
- `tmux` is a newer program with the same style. It has some extra features and some missing features still.

Some people have their `screen` open forever, which just keeps running and never gets closed. Wherever they are, they ssh in, connect, and resume right where they left off.

Example: `irssi` on kosh / lyta

Exercises 1.1

✍ Exercise

- Find out which shell you are running, your user name, hostname, system name. For Aalto users: set your SHELL to BASH if you have not yet done so: `chsh -s /bin/bash` on kosh
- Find out with *man* how to use *top* / *ps* to list all the running processes that belong to you Tip: *top* has both command line options and hot keys.
- Find your shell session's PID, list the processes tree of all child processes that belong to your current session, with the command line, PID, user
- With pgrep list all bash processes, if you have SSH access, try both locally, and on a remote Linux server
- Run `nano filename`, send it to the background, and return back to the foreground``. Tip: quite `nano` with ``Ctrl-x`.
- Run `man htop`, send it to the backround, and then kill it with `kill`. Tip: one can do it by background job number or by PID.
- (*) Run `screen` session. Detach, close the seesion, open again and attach 'screen' back. Exit 'screen'.
- (*) Find out how to list a processes tree with `ps`, both all processes and only your own (but all your processes, associated with all terminals)
- (*) Try a use case: your current ssh session to a remote host "got stuck" and does not response. Open another ssh session to the same remote host and kill the first one. Tip: `echo $$` gives you current bash PID.
- (*) Get any X Window application (firefox, xterm, etc) to run on a remote Linux machine

[1] https://sourceforge.net/projects/vcxsrv/

Files and directories

Files contain data. They have a name, permissions, owner (user+group), contents, and some other metadata.

Filenames may contain any character except '/', which is reseved as a separator between directory and filenames. The special characters would require quotaion while dealing, with such filenames, though it makes sence to avoid them anyway.

Path can be absolute, starts with '/' or relative, that is related to the current directory.

`ls` is the standard way of getting information about files. By default it lists your current directory (i.e. *pwd*), but there are many options:

```
# list directory content
ls /usr/bin

# list directory files including dot files (i.e. hidden ones)
ls -A ~/directory1

# list all files and directories using long format (permissions, timestamps, etc)
ls -lA ../../directory2
```

Special notations and expanssions in BASH, can be used with any command:

```
./    stands for the current directory
../   parent directory
~     home directory
*     a wildcard, replaces any character(s) or none
?     only one character
[]    a group of characters, like [abc]
[!]   same as above but negated, i.e. all but those, like [!a-zA-Z]
{ab,cd,xyz}  expands by BASH as 'ab cd xyz'
{0..9}       expands as '0 1 2 3 4 5 6 7 8 9'
```

For the quotation:

```
'', "", \
```

Quotation matters `ls 'file name'` vs `ls file name` or `echo "$USER"` vs `echo '$USER'`

BASH first expand the expanssions and substitute the wildcards, and then execute the command. Could be as complex as:

```
ls -l ~/[!abc]???/dir{123,456}/filename*.{1..9}.txt
```

There are a variety of commands to manipulate files/directories:

```
cd, mkdir, cp, cp -r, rm, rm -r, mv, ln, touch
```

For file/directory meta information or content type:

```
ls, stat, file, type -a
```

Note that `cd` is a shell builtin which change's the shell's own working directory. This is the base from which all other commands work: `ls` by default tells you the current directory. `.` is the current directory, `..` is the parent directory, `~` is your HOME. This is inherited to other commands you run. `cd` with no options drops your to your $HOME.

```
# copy a directory preserving all the metadata to two levels up
cp -a dir1/ ../../

# move all files with the names like filename1.txt, filename_abc.txt etc to dir2/
mv filename*.txt dir2/

# remove a directories/files in the current dir without asking for the confirmation
rm -rf dir2/ dir1/ filename*

# create an empty file if doesn't exist or update its access/modification time
touch filename

# create several directories at once
mkdir dir3 dir4 dir5
# -or-
mkdir dir{3,4,5}

# make a link to a target file (hard link by default, -s for symlinks)
ln target_file ../link_name
```

Discover other ls features `ls -lX`, `ls -ltr`, `ls -Q`

You may also find useful `rename` utility implemented by Larry Wall.

File/directory permissions

- Permissions are one of the types of file metadata.
- They tell you if you can *read* a file, *write* a file, and *execute a file/list directory*

- Each of these for both *user*, *group*, and *others*
- Here is a typical permission bits for a file: `-rw-r--r--`
- In general, it is `rwxrwxrwx` – read, write, execute/search for user, group, others respectively
- `ls -l` gives you details on files.

Modifying permissions: the easy part

chmod/chown is what will work on all filesystems:

```
chmod u+rwx,g-rwx,o-rwx <files>   # u=user, g=group, o=others, a=all
# -or-
chmod 700 <files>   # r=4, w=2, x=1

# recursive, changing all the subdirectories and files at once
chmod -R <perm> <directory>

# changing group ownership (you must be a group member)
chgrp group_name <file or directory>
```

Extra permission bits:

- s-bit: setuid/setgid bit, preserves user and/or group IDs.
- t-bit: sticky bit, for directories it prevents from removing file by another user (example */tmp*)

Setting default access permissions: add to *.bashrc* `umask 027` [1]. The `umask` is what permissions are *removed* from any newly created file by default. So `umask 027` means "by default, g-w,o-rwx any newly created files". It's not really changing the permissions, just the default the operating system will create with.

Hint: even though file has a read access the top directory must be searchable before external user or group will be able to access it. Sometimes on Triton, people do `chmod -R o-rwx $WRKDIR; chmod o+x $WRKDIR`. Execute (`x`) without read (`r`) means that you can access files inside if you know the exact name, but not list the directory. The permissions of the files themselves still matter.

Modifying permissions: advanced (*)

Access Control Lists (ACLs) are advanced access permissions. They don't work everywhere, for example mostly do no work on NFS mounted directories. They are otherwise supported on ext4, lustre, etc (thus works on Triton $WRKDIR).

- In "normal" unix, files have only "owner" and "group", and permissions for owner/group/others. This can be rather limiting.

- Access control lists (ACLS) are an extension that allows an arbitrary number of users and groups to have access rights to files. The basic concept is that you have:
- ACLs don't show up in normal `ls -l` output, but there is an extra plus sign: `-rw-rwxr--+`. ACLs generally work well, but there are some programs that won't preserve them when you copy/move files, etc.
- POSIX (unx) ACLs are controlled with `getfacl` and `setfacl`
  - Allow read access for a user `setfacl -m u:<user>:r <file_or_dir>`
  - Allow read/write access for a group `setfacl -m g:<group>:rw <file_or_dir>`
  - Revoke granted access `setfacl -x u:<user> <file_or_dir>`
  - See current stage `getfacl <file_or_dir>`

File managers on Triton we have installed Midnight Commander – `mc`

Advanced file status to get file meta info `stat <file_or_dir>`

Exercise 1.2

✍ Exercise

- mkdir in the current direcotory, cd there and `touch` a file. Rename it. Make a copy and then remove the original. What does `touch` do?
- list all files in /usr/bin and /usr/sbin that start with non-letter characters with one `ls` command
- list with `ls` dot files/directories only (by default it lists all files/directories but not those that begin with `.`). "dotfiles" are a convention where filenames that begin with `.` such as `.bashrc` are considered "hidden".
- Explore `stat file` output. What metadata do you find? Try to stat files of different types (a regular file, directory, link, special device in /dev)
- create a directory, use `chmod` to allow user and any group members full access and no access for others
- (*) change that directory group ownership with `chown` or `chgrp` (any group that you belong to is fine), set s-bit for the group and apply t-bit to a directory, check that the upper directory has *o+x* bit set: now you should have a private working space for your group. Tip: see groups that you are a member of `id -Gn`
- `ls -ld` tells you that directory has permissions `rwxr-Sr--`. Do group members have access there?
- (*) create a directory (in /tmp if you are on a server with), use `setfacl` to set its permissions so that only you and some user/group of your choice would have access to it.
- (*) create a directory and a subdirectory in it and set their permissions to 700 with one command.

[1] https://www.computerhope.com/unix/uumask.htm

Find

- `find` is a very unixy program: it finds files, but in the most flexible way possible.
- It is a amazingly complicated program
- It is a number one in searching files in shell

With no options, just recursively lists all files starting in current directory:

```
find
```

The first option gives a starting directory:

```
find /etc/
```

Other search options: by modification/accessing time, by ownership, by access type, joint conditions, case-insensitive, that do not match, etc [1] [2]:

```
# -or-  'find ~ $WRKDIR -name file.txt' one can search more than one dir at once
find ~ -name file.txt

# look for jpeg files in the current dir only
find . -maxdepth 1 -name '*.jpg' -type f

# find all files of size more than 10M and less than 100M
find . -type -f -size +10M -size -100M

# find everything that does not belong to you
find ~ ! -user $USER | xargs ls -ld

# open all directories to group members
# tip: chmod applies x-bit to directories automatically
find . -type d -exec chmod g+rw {} \;

# find all s-bitted executable binaries
find /usr/{bin,sbin} -type f -perm -u+x,u+s

# find and remove all files older than 7 days
find path/dir -type f -mtime +7 -exec rm -f {} \;
```

Find syntax is actually an entire boolean logic language given on the command line: it is a single expression evaluated left to right with certain precedence. There are match expressions and action expressions. Thus, you can get amazingly complex if you want to. Take a look at the ‘EXAMPLES’ section in *man find* for the comprehensive list of examples and explanations.

find on Triton On Triton's WRKDIR you should use `lfs find`. This uses a raw lustre connection to make it more efficient than accessing every file. It has somewhat limited abilities as comparing to GNU find. For details `man lfs` on Triton.

Fast find – locate Another utility that you may find useful `locate <pattern>`, but on workstations only. This uses a cached database of all files, and just searches that database so it is much faster.

Too many arguments error solved with the `find ... | xargs`

[1] https://alvinalexander.com/unix/edu/examples/find.shtml

[2] http://www.softpanorama.org/Tools/Find/index.shtml

File archiving and transferring

File archiving

`tar` is the de-facto standard tool for saving many files or directories into a single archive file. Archive files may have extenssions *.tar*, *.tar.gz* etc depending on compression.

```
# create tar archive gzipped on the way
tar -caf arhive_name.tar.gz directory_to_be_archived/

# extract files
tar -xaf archive_name.tar.gz -C path/to/directory
```

Other command line options: *r* - append files to the end of an archive, *t* - list archive content. *f* is for the filename, and *a* selects the compression method based on the archive file suffix (in this example gzip, due to the .gz suffix. Without compression files/directories are simply packed as is.

```
# xz has better compression ratio than gzip, but is very slow
tar -caf archive_file.tar.xz dir1/ dir2/
```

Individual files can be compressed directly, e.g. with `gzip`:

```
# file.gz is created, file is removed in the process.
gzip file
# Uncompress
gunzip file.gz
```

Transferring files (+archiving on the fly)

As an Aalto user you may use remote directories [1]. Here we cover general use case, transferring from the command line.

For transferring single files or directories there is *scp*, though for FTP like functionality there is SFTP.

```
# transferring a file from the remote server to the current directory
sftp LOGIN@remote.server.fi:/path/to/file_to_copy .

# transferring files from your workstation to  remote server
sftp path/to/file_to_copy LOGIN@remote.server.fi:/path/to/destination/directory
```

Another use case, making a directory backup with `rsync`:

```
# sync two directories
rsync -vauW path/to/dir1/ LOGIN@remote.server.fi:/path/to/destination/dir1
```

(*) Transferring and archiving your data on the fly to some other place. The example will be in covered in details later.:

```
tar czf - path/to/dir | ssh LOGIN@remote.server.fi 'cat >
path/to/archive/dir/archive_file.tar.gz'
```

Exercise 1.3

✍ Exercise

- Find with `find` all the files in your $HOME that are readable or writable by everyone
  - (*) apply `chmod o-rwx` to all recently found files with `find`
- Make a tar.gz archive of any of your directory, when done list the archive content
- Extract only one particular file to a subdirectory from the archive
- Transfer just created archive using `sftp` (if still time, do the same with scp and rsync).
  - (*) Try ssh+tar combo to make transfer and archive on the fly.

[1] https://scicomp.aalto.fi/aalto/remoteaccess/#remote-mounting-of-network-filesystems

Command line utilities

Utilities: the building blocks of shell

- wide range of all kind of utilities available in Linux
- shell is a glue to bind them all together
- commandline is often a long list of those utilities joint into pipe that pass output of each other further

```
echo, pwd, id, hostname, uname, ps, top, pstree, bg/fg, jobs, kill, touch
ls, cd, cp, rm, mv, mkdir, ln, type, stat, file, du, chmod, chgrp (chown),
find, tar, gzip, sftp, rsync, man, nano (vim/emacs), less, ssh, ...
grep, cat, tr, cut, sort, head, tail, uniq, col, xargs,
date, wc, cal, nl, diff, alias, df, basename, w, split, tee,
sed, awk, paste, ...
```

Additional utilities for the software development, system administration etc

Coreutils by GNU You may find many other useful commands at
https://www.gnu.org/software/coreutils/manual/coreutils.html

Input and output: redirect and pipes

- stdout and stdin from the processes section, you remember right? each process has it
- by default *stdout* goes to the screen and *stdin* expects input from the keyboard
- we can change that on demand: pipes and redirections

Pipe: output of the first command as an input for the second one `command_a | command_b`:

```
# see what files/directories use the most space, including hidden ones
du -hs * .[!.]* | sort -h | tail -n 10

# count a number of logged in users
w -h | wc -l

# send man page to a default printer
man -t ls | lpr

# print all non-printable characters as well
ls -lA | cat -A
```

Redirects:

- Like pipes, but send data to/from files instead of other processes.
- Replace a file: `command > file.txt`
- Append to a file: `command >> file.txt` (be careful you do not mix them up!)
- Redirect file as STDIN: `command < file` (in case program accepts STDIN only)

```
echo Hello World > hello.txt

ls -lH >> current_dir_ls.txt

# create a few dummy files
echo 'a b c' > file1
echo 'x y z' > file2

# join two files into one
cat file1 file2 > file3

# go through file1 and replace spaces with a new line mark, then output to file2
tr -s ' ' '\n' < file1 > file4
# the same result but another approach: (and more readable format)
cat file2 | tr -s ' ' '\n' > file5

# join file1 and 2 lines one by one using : as a delimiter
paste -d : file4 file5 > file6

# get rid of output, 'null' is a special device
command > /dev/null
```

This is the unix philosophy and the true power of the shell. The unix philosophy is a lot of small, specialized, good programs which can be easily connected together. The beauty of the cli are elegant one-liners i.e. list of commands executed in one line.:

```
# tar and copy directory to a remote host
tar czf - path/to/dir | ssh LOGIN@remote.server.fi 'cat >
path/to/archive/dir/archive_file.tar.gz'

# to remove all carriage returns and Ctrl-z characters from a Windows file
cat win.txt | tr -d '\15\32' > unix.txt

# extract user names and store them to a file
getent passwd | cut -d: -f1,5 > users

# print the name of the newest file in the directory (non-dot)
ls -1tF | grep -v -E '*/|@' | head -1
```

http://www.bashoneliners.com/

Groupping commands

To dump output of all commands at once: group them.

```
{ command1; command2; } > filename  # commands run in the current shell  as a group
( command1; command2; ) > filename  # commands run in external shell as a group
```

stderr

A separate stream (=file descriptor), though we can direct it as well:

```
# redirect both stderr and stdout to /dev/null
ping -c 1 8.8.8.8 &> /dev/null

# piping both
command_a &| command_b
```

Advanced usage cases, like subshells, process substitution, other file descriptors than stdin/stderr/stdout etc will be covered in the Part 2 of this course.

Evaluations, separators and grep

Evaluations and separators: ; && ||

- With ; you can put several commands on the same line.

Chaining: `command_a ; command_b`: always runs both commands.

Remember exit codes? In shell, 0=success and anything 1-255=failure. Note that this is opposite of normal Boolean logic!

The `&&` and `||` are short-circuit (lazy) boolean operators. They can be used for quick conditionals.

- `command_a && command_b`
  - If `command_a` is successful, also run `command_b`
  - final exit code is last evaluated one, which has the role of Boolean *and*.
- `command_a || command_b`
  - If `command_a` is *not* successful, also run `command_b`
  - final exit code is that of the last evaluated command, which has the role of Boolean *or*.

Hint `command_a && command_b || command_c`

Try: `cd /nonexistent_dir && ls /nonexistent_dir` compare with `cd /nonexistent_dir; ls /nonexistent_dir`

Try: `ping -c 1 8.8.8.8 > /dev/null && echo online || echo offline`

grep

Later on you'll find out that `grep` is one of the most useful commands you ever discover on Linux (except for all the *other* most useful commands ever)

```
grep <pattern> <filename>  # grep lines that match <pattern>
 -or-
command | grep <pattern>  # grep lines from stdin
```

```
# search all the files in the dir/ and its subdirs, to match the word 'is', case
insensitive
grep -R -iw 'is' dir/

# grep all lines from *command* output, except those that have 'comment' in it
*command* | grep -v comment

# displaying 2 extra lines before and after the match (-A just after, -B just before)
grep -C 2 'search word' file

# counts the number of matches
grep -c <pattern> file(s)

# shows only the matched part of the string (by default grep shows whole line)
grep -o <pattern> file(s)

# accepts way more advanced regular expressions as a search pattern
grep -E <extended_regexpr> file(s)
```

For details on what <pattern> could be, look for REGULAR EXPRESSIONS at `man grep`.
Some examples:

```
# grep emails to a list
grep -Eio "\b[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,6}\b" file.txt

# grep currently running firefox processes
ps auxw | grep firefox

# grep H1 and H2 header lines out of HTML file
grep "<[Hh][12]>" file.html
```

Exercise 1.4

✍ Exercise

- make a pipe that counts number of files/directories (including dot files) in your directory
- grep directories out of `ls -l`
- grep all but blank lines of the 'man cut | grep ...'
- Using pipes and commands echo/tr/uniq, find doubled words out of `My Do Do list: Find a a Doubled Word`.
- If you are on a multiuser system, count unique logged in users. Tip: `w` or `users` gives you a list of all currently login users, many of them have several sessions open. Commands to discover: cut / sort / wc

- (*) Play with the commands grep, cut: find at least two ways to extract IP addresses out of /etc/hosts. Tip: *grep* has *-o* option, thus one can build a regular expression that will grab exactly what you need.

Initialization files and file editing

Initialization files and configuration

- When the shell first starts (when you login), it reads some files. These are normal shell files, and it evaluates normal shell commands to set configuration.
- You can always test things in your own shell and see if it works before putting it in the config files. Highly recommended!
- You customize your environment means setting or expanding aliases, variables, functions.
- The config files are:
  - `.bashrc` (when SSH) and
  - `.bash_profile` (interactive login to a workstation)
  - they are often a symlink from one to another
- To get an idea how complicated .bashrc can be take a look at <https://www.tldp.org/LDP/abs/html/sample-bashrc.html>

One of the things to play with: command line prompt defined in PS1 [1]

```
PS1="[\d \t \u@\h:\w ] $ "
```

For special characters see PROMPTING at `man bash`. To make it permanent, should be added to *.bashrc* like `export PS1`.

Creating/editing/viewing file

- A *text editor* edits files as ASCII. These are your best friend. In fact, text files are your best friend: rawest, most efficient, longest-lasting way of storing data.
- “pager” is a generic term for things that view files or data.

Linux command line *text editors* like:

- *nano* - simplest
- *vim* - minimal. To save&quit, `ESC :wq`
- *emacs* - or the simplest one *nano*. To save&quit: `Ctrl-x Ctrl-c`

To view contents of a file in a scrollable fashion: `less`

Quick look at the text file `cat filename.txt` (dumps everything to screen- beware of non-text binary files or large files!)

Other quick ways to add something to a file (no need for an editor)

```
echo 'Some sentence, or whatever else 1234567!-+>$#' > filename.txt
```

`cat > filename2.txt` to finish typing and write written to the file, press enter, then Ctrl-d.

The best text viewer ever `less -S` (to open a file in your EDITOR, hit *v*, to search through type */search_word*)

Watching files while they grow `tail -n 0 -f <file>`

Try: add above mentioned `export PS1` to *.bashrc*. Remember `source .bashrc` to enable changes

Exercise 1.5

✍ Exercise

- link *.bash_profile* to *.bashrc*. Tip: see `ln` command from the previous session.
- add `umask 027` to *.bashrc*, try creating files. Tip: `umask -S` prints your current setting.
- customize a prompt `$PS1` and add it to your *.bashrc*, make sure is has a current directory name and the hostname in it in the format *hostname:/path/to/current/dir*. Hint: save the original PS1 like `oldPS1=$PS1` to be able to recover it any time.

[1] https://www.ibm.com/developerworks/linux/library/l-tip-prompt/

How to make things faster: hotkeys

- Is it annoying to have to type everything in the shell? No, because we have hotkeys. In fact, it can become much more efficient and powerful to use the shell.
- Most important key: **TAB**: autocomplete. You should never be typing full filenames or command names. TAB can complete almost anything

Common hotkeys:

- TAB – autocomlpetion
- Home `or` Ctrl-a – start of the command line
- End `or` Ctrl-e – end
- Ctrl-left/right arrows `or` Alt-b/Alt-f - moving by one word there and back
- up/down arrows – command history
- Ctrl-l – clear the screen
- Ctrl-Shift-c – copy
- Ctrl-Shift-v – paste
- Ctrl-Shift– – undo the last changes on cli
- Alt-r – undo all changes made to this line

- Ctrl-r – command history search: backward (hit Ctrl-r, then start typing the search word, hit Ctrl-r again to go through commands that have the search word in it)
- Ctrl-s – search command history furtherword (for this to work one needs to disable default suspend keys `stty -ixon`)
- Ctrl-u – remove beginning of the line, from cursor
- Ctrl-k – remove end of the line, from cursor
- Ctrl-w – remove previous word

inputrc Check */etc/inpurc* for some default key bindings, more can be defined *~/.inputrc* (left as a home exercise)

CDPATH helps changing directories faster. When you type `cd dirname`, the shell tries to go to one of the local subdirectories and if it is not found shell will try the same command from every directory listed in the *$CDPATH*.

```
export CDPATH=$HOME:$WRKDIR:$WRKDIR/project
```

Quoting, substitutions, aliases

Last time, we focused on interactive things from the command line. Now, we build on that some and end up with making our own scripts.

Command line processing and quoting

So, shell is responsible for interpreting the commands you type. Executing commands might seem simple enough, but a lot happens between the time you press RETURN and time your computer actually does something.

- When you enter a command line, it is one string.
- When a program runs, it always takes an array of strings (the `argv` in C, `sys.argv` in Python, for example). How do you get from one string to an array of strings? Bash does a lot of processing.
- The simplest way of looking at it is everything separated by spaces, but actually there is more: variable substitution, command substitution, arithmetic evaluation, history evaluation, etc.

The partial order of operations is (don't worry about exact order: just realize that the shell does a lot of different things in same particular order):

- history expansion
- brace expansion (`{1..9}`)
- parameter and variable expansion (`$VAR`, `${VAR}`)
- command substitution (`$()`)
- arithmetic expansion (`$((1+1))`)

- word splitting
- pathname expansion (`*` , `?` , `[a,b]`)
- redirects and pipes

One thing we will start to see is shell quoting. There are several types of quoting (we will learn details of variables later):

```
# Double quotes: disable all other characters except $, ', \
echo "$SHELL"

# Single quotes: disable all special characters
echo '$SHELL'

# backslash disables the special meaning of the next character
ls name\ with\ space
```

By special characters we mean:

```
# & * ? [ ] ( ) { } = | ^ ; < > ` $ " ' \
```

There are different rules for embedding quoting in other quoting. Sometimes a command passes through multiple layers and you need to really be careful with multiple layers of quoting! This is advanced, but just remember it.

```
echo 'What's up? how much did you get $$?'      # wrong, ' can not be in between ''
echo "What's up? how much did you get $$?"      # wrong, $$ is a variable in this case
echo "What's up? how much did you get \$\$?"    # correct
echo "What's up? how much did you get "'$$'"?"  # correct
```

At the end of the line `\` removes the new line character, thus the command can continue to a next line:

```
ping -c 1 8.8.8.8 > /dev/null && \
echo online || \
echo offline
```

Substitute a command output

- Command substitutions execute a command, take its stdout, and place it on the command line in that place.

`$(command)` or alternatively `` `command` ``. Could be a command or a list of commands with pipes, redirections, grouping, variables inside. The `$()` is a modern way, supports nesting, works inside double quotes. To understand what is going on in these, run the inner command first.

```
# 'whoami' alternative
echo $(id -un):$(id -gn)@$(hostname -s)

# save current date to a variable
today=$(date +%Y-%m-%d)

# create a new file with current timestamp in the name (almost unique filename)
touch file.$(date +%Y-%m-%d-%H-%M-%S)

# archive current directory content, where new archive name is based on current path
and date
tar czf $(basename $(pwd)).$(date +%Y-%m-%d).tar.gz .
```

This is what makes BASH powerful!

Note: `$(command || exit 1)` will not have an effect you expect, command is executed in a subshell, exiting from inside a subshell, closes the subshell only not the parent script. Subshell can not modify its parent shell environment, though can give back exit code or signal it:

```
# this will not work, echo still will be executed
dir=nonexistent
echo $(ls -l $dir || exit 1)

# this will not work either, since || evaluates echo's exit code, not ls
echo $(ls -l $dir) || exit 1

# this will work, since assignment a comman substitution to a var returns exit
# code of the executed command
var=$(ls -l $dir) || exit 1
echo $var
```

More about redirection, piping and process substitution

STDIN, *STDOUT* and *STDERR*: reserved file descriptors *0*, *1* and *2*. They always there whatever process you run. But one can use other file descriptors as well.

File descriptor is a number that uniquely identifies an open file.

/dev/null file (actually special operating system device) that discards all data written to it.

```
# discards STDOUT only
command > /dev/null

# discards both STDOUT and STDERR
command &> /dev/null
command > /dev/null 2>&1    # same as above, old style notation

# redirects outputs to different files
command 1>file.out 2>file.err

# takes STDIN as an input and outputs STDOUT/STDERR to a file
command < input_file &> output_file
```

Note, that `&>` and `>&` will do the same, redirect both STDOUT and STDERR to the same place, but the former syntax is preferable.

```
# what happens if 8.8.8.8 is down? How to make the command more robust?
ping -c 1 8.8.8.8 > /dev/null && echo online || echo down

# takes a snapshot of the directory list and send it to email, then renames the file
ls -l > listing && { mail -s "ls -l $(pwd)" jussi.meikalainen@aalto.fi < listing; mv
listing listing.$(date +"%Y-%m-%d-%H-%M"); }

# a few ways to empty a file
> filename
cat /dev/null > filename

# read file to a variable
var=$(< path/to/file)

# extreme case, if you can't get the program to stop writing to the file...
ln -s /dev/null filename
```

Pipes are following the same rules with respect to standard output/error. In order to pipe both STDERR and STDOUT `|&`.

If `!` preceeds the command, the exit status is the logical negation.

tee in case you still want output to a terminal and to a file `command | tee filename`

`exec > output.txt` or `exec 2> errors.txt` executed in the script will send the output to the file, standard output or error output correspondingly. Opening other than standard file descriptors: **exec** causes the shell to hold the file descriptor until the shell dies or closes it.

```
# open input_file for reading into the file descriptor 3
exec 3< $input_file
# while open, any command can operate on the descriptor
read -n 3 var <&3
command <&3
# mind the file offset, one can read a line, or a few chars, if you have read the file
# to the end, to reset the offset, run another 'exec 3< ...'
# close the descriptor after you are done
exec 3>&-

# similar for writing
exec 5> $output_file; command > &5; ...; exec 5>&-
# or appending (keep in mind that you use >> only to open the file)
exec 5>> $output_file; command > &5; ...; exec 5>&-
# or writing and reading
exec 6<>$file; ... exec 6<>&-
# or use a name instead of the descriptor numeric value
exec {out}>$output_file; ... echo something >&$out; ...
# redirecting descriptor to another one
exec 3>&1
```

Opening a FD instead of using a file name multiple times may save you some IO. *Hint:* to monitor the file operations (system calls) one may employ **strace -f -c -e trace=write,openat your_script**.

But what if you need to pass to another program results of two commands at once? Or if command accepts file as an argument but not STDIN?

One can always do this in two steps, run commands and save results to file(s) and then use them with the another command. Though BASH helps to make even this part easier (or harder), the feature called *Process Substitution*, looks like `<(command)` or `>(command)`, no spaces in between parentheses and < signs. It emulates a file creation out of *command* output and place it on a command line. The *command* can be a pipe, pipeline etc.

The actual file paths substituted are */dev/fd/<n>*. The file paths can be passed as an argument to the another command or just redirected as usual.

```
# BASH creates a file that has an output of *command2* and pass it to *command1*
# file descriptor is passed as an argument, assuming command1 can handle it
command1 <(command2)

# same but redirected (like: cat < filename)
command1 < <(command2)

# in the same way one can substitute results of several commands or command groups
command1 <(command2) <(command3 | command4; command5)

# example: comparing listings of two directories
diff <(ls dir1) <(ls dir2)

# and vice versa, *command1* output is redirected as a file to *command2*
command1 > >(command2)

# essentially, in some cases pipe and process substituion do the same
ls -s | cat
cat <(ls -s)
```

Aliases

- Alias is nothing more than a shortcut to a long command sequence
- With alias one can redefine an existing command or name a new one
- Alias will be evaluated only when executed, thus it may have all the expansions and substitutions one normally has on the cli
- They are less flexible than functions which we will discuss next

```
# your own listing command
alias l='ls -lAF'

# shortcut for checking space usage
alias space='du -hs .[!.]* * | sort -h'

# prints in the compact way login:group
alias me='echo "$(id -un):$(id -gn)"'

# redefine rm
alias rm='rm -i'
alias rm='rm -rf'
```

Aliases go to *.bashrc* and available later by default (really, anywhere they can be read by the shell).

Exercise 2.1

[Lecturer's notes: about 40 mins joint hands-on session + break]

✍ Exercise

- Use command substitution to create an empty file with the date in the name, like `file.YYYY-MM-DD.out`. Tip: investigate `date +"..."` examples above and/or `man date`.

- Learn Brace expansions `echo {0..9} {a..z}`. Using it, create five directories (`mkdir`) in the current folder with the names like: DIR.NUMBER.CURRENT_YEAR, example mydir.1.2022, mydir.2.2022
- Make a command (so called one-liner) with `ls`, `echo`, redirections etc that takes a file path and says whether this file/directory exists or not. Redirect STDOUT/STDERR to /dev/null. Take `ping -c 8.8.8.8 ...` as an example.
- Use the example in the text above to send `du -hs * .[!.]* | sort -h` output to yourself via email.
- (*) Use any of the earlier created files to compare there modification times with `stat -c '%y' filename`, `diff` and the process substitution.
- (*) Using pipes and commands `echo`, `tr`, `uniq`, find doubled words out of `My Do Do list: Find a a Doubled Word.`
- (*) Join *find* and *grep* power and find all the files in /{usr/,}{bin,sbin} that have '#!/bin/bash' in it

Variables, functions, environment

Your ~/bin and PATH

The PATH is an environment variable. It is a colon delimited list of directories that your shell searches through when you enter a command. Binaries are at */bin*, */usr/bin*, */usr/local/bin* etc. The best place for your own is *~/bin.*:

```
# add to .bashrc
export PATH="$PATH:$HOME/bin"
# after you have your script written, set +x bit and run it
chmod +x ~/bin/script_name.sh
script_name.sh
```

You can find where a program is using `which` or `type -a`, we recommend the later one:

```
type -a ls      # a binary
type -a cd      # builtin
```

Other options:

```
# +x bit and ./
chmod +x script.sh
./script.sh   # that works if script.sh has #!/bin/bash as a first line
# with no x bit
bash script.sh  # this will work even without #!/bin/bash
```

Extension is optional note that *.sh* extension is optional, script may have any name

Variables

In shell, variables define your environment. Common practice is that environmental vars are written IN CAPITAL: $HOME, $SHELL, $PATH, $PS1, $RANDOM. To list all defined variables `printenv`. All variables can be used or even redefined. No error if you call an undefined var, it is just considered to be empty:

```
# assign a variable, note, no need for ; delimiter
var1=100 var2='some string'

# calling a variable is just putting a $ dollar sign in a front
echo "var1 is $var1"

# re-assign to another var
var3=$var1

# when appending a variable, it is considered to be a string
var+=<string>/<integer>
  var1+=50  # var1 is now 10050
  var2+=' more' # var2 is 'some string more'
# we come later to how to deal with the integers (Arithmetic Expanssions $(()) below)
```

There is no need to declare things in advance: there is flexible typing. In fact, you can access any variable, defined or not. However, you can still declare things to be of a certain type if you need to:

```
declare -r var=xyz   # read-only
declare -i var  # must be treated as an integer, 'man bash' for other declare options
```

BASH is smart enough to distinguish a variable inline without special quoting:

```
dir=$HOME/dir1 fname=file fext=xyz echo "$dir/$fname.$fext"
```

though if variable followed by a number or a letter, you have to explicitly separate it with the braces syntax:

```
echo ${dir}2/${file}abc.$fext
```

Built-in vars:

- $? exit status of the last command
- $$ current shell pid
- $# number of input parameters
- $0 running script name, full path
- $FUNCNAME function name being executed, [note: actually an array ${FUNCNAME[*]}]
- $1, $2 ... input parameter one by one (function/script)
- "$@" all input parameters as is in one line

```
f() { echo -e " number of input params: $#\n input params: $@\n shell process id: $$\n
script name: $0\n function name: $FUNCNAME"; return 1; }; f arg1 arg2; echo "exit code:
$?"
```

What if you assing a variable to a variable like:

```
var2='something'
var1=\$var2
echo $var1     # will return '$var2' literally

# BASH provides built-in 'eval' command that reads the string then re-evaluate it
# if variables etc found, they are given another chance to show themselves

eval echo $var1  # returns 'something'
```

In more realistic examples it is often used to compose a command string based on input parameters or some conditionals and then evaluate it at very end.

Magic of BASH variables

BASH provides wide abilities to work with the vars "on-the-fly" with `${var...}` like constructions. This lets you do simple text processing easily. These are nice, but are easy to forget so you will need to look them up when you need them.

- Assign a $var with default *value* if not defined: `${var:=value}`
- Returns $var value or a default *value* if not defined: `${var:-value}`
- Print an *error_message* if var empty: `${var:?error_message}`
- Extract a substring: `${var:offset:length}`, example `var=abcde; echo ${var:1:3}` returns 'bcd'
- Variable's length: `${#var}`
- Replace beginning part: `${var#prefix}`
- Replace trailing part: `${var%suffix}`
- Replace *pattern* with the *string*: `${var/pattern/string}`
- Modify the case of alphabetic characters: `${var,,}` for lower case or `${var^^}` for upper case

```
# will print default_value, which can be a variable
var=''; echo ${var:-default_value}
var1=another_value; var='';  echo ${var:-$var1}

# assign the var if it is not defined
# note that we use ':' no operation command, to avoid BASH's 'command not found' errors
: ${var:=default_value}

# will print 'not defined' in both cases
var='';  echo ${var:?not defined}
var=''; err='not defined'; echo ${var:?$err}

# will return 8, that is a number of characters
var='abcdefgh'; echo ${#var}

# returns file.ext
var=26_file.ext; echo ${var#[0-9][0-9]_}

# returns archive.tar.gz out of full path
fpath=/home/user/archive.tar.gz; echo ${fpath##*/}
# returns path with no file name
echo ${fpath%/*}

# in both cases returns photo
var=photo.jpeg; echo ${var%.jpeg}
var=26_file.ext; echo ${var%.[a-z][a-z][a-z]}

# returns 'I hate you'
var='I love you'; echo ${var/love/hate}
# other options for substitutions
var=' some text ';
echo ${var/# /}  # returns without the first space
echo ${var/% /}  # without the last space
echo ${var// /}  # without spaces at all
```

Except for the := the variable remains unchanged. If you want to redefine a variable:

```
var='I love you'; var=${var/love/hate}; echo $var  # returns 'I hate you'
```

BASH allows indirect referencing, consider:

```
var1='Hello' var2=var1
echo $var2  # returns text 'var1'
echo ${!var2}  # returns 'Hello' instead of 'var1'
```

To address special characters:

```
# replacing all tabs with the spaces in the var
var=${var//$'\t'/ }
```

Functions

Alias is a shortcut to a long command, while function is a piece of programming that has logic and can accept input parameters. Functions can be defined on-the-fly from the cli, or can go to a file. Let us set *~/bin/functions* and collect everything useful there.:

```
# whoami alternative turned into function
me() {
  $(id -un):$(id -gn)@$(hostname -s)
}

# turn check space usage into a function
spaceusage() {
  du -hs * .[!.]* | sort -h
}

# in one line, note spaces and ; delimiters
myfunction() { command; command; }
# -or- in a full format
function myfunction { command; command; }
```

Read functions into the current shell environment and run them:

```
source ~/bin/functions
me
spaceusage
```

The function refers to passed arguments by their position (not by name), that is $1, $2, and so forth:

```
func_name arg1 arg2 arg3  # will become $1 $2 $3

# advanced version of spaceusage using BASH variables magic
spaceusage() {
  du -hs ${1:-.}/* ${1:-.}/.[!.]* | sort -h;
}
```

Functions in BASH have `return` but it only returns the exit code. Useful in cases where you want to 'exit' the function and continue to the rest of the script. By default functions' variables are in the global space, once chaged in the function is seen everywhere else. `local` can be used to localize the vars. Compare:

```
var=2; f() { var=3; }; f; echo $var
var=2; f() { local var=3; }; f; echo $var

# get filename out of path
filename() {
  local fpath=${1:?path is missing} && \
  echo ${fpath##*/}
}

# filepath with no name
filepath() {
  local fpath=${1:?path is missing} && \
  echo ${fpath%/*}
}
```

If you happened to build a function in an alias way, redefining a command name while using that original command inside the function, you need to type *command* before the name of the command, like:

```
rm() { command rm -i "$@"; }
```

here you avoid internal loops (forkbombs).

Exporting a function with `export -f function_name` lets you pass a function to a sub-shell, by storing that function in a environment variable. Helpful when you want to use it within a command substitution, or any other case that launches a subshell, like `find ... -exec bash -c 'function_name {}' \;`.

Exercise 2.2

✍ Exercise

- Add `spaceusage()`, `filename()`, `filepath()`, `me()` to your *~/bin/functions* and play with them. Note: here and later, we suggest that all newly created functions would go to *~/bin/functions* file.
- Using `filename()` function make a `filebasename()` so that function would output a *filename* with no extension. Like `filebasename path/to/archive.tar.gz` would return *archive*.
- Using `find` utility, implement a *fast find* (=*ff*) function `ff word`. This function must return all the files and directories in the current folder which name contains *<word>*. Let it be case insensitive. Hint: `find . -iname ...`
- (*) Make an advanced version of `ff()` that would accept a directory name to search at as a second argument *($2)* and if it is missing then would use current. For a example `ff word path/to/`.
- (*) `:() { :|:&; };:` is a BASH fork-bomb [WARNING: Do not run it!]. Can you explain how it works it? & in this case sends process to the background.

- (*) On Triton write a function that `lfs find` all the dirs/files at $WRKDIR that do not belong to your group and fix the group ownership. Use `find ... | xargs`. Tip: on Triton at WRKDIR your username $USER and group name are the same. On any other filesystem, `$(id -gn)` returns your group name. One can
- (*) Expand the function above to set group's s-bit on all the $WRKDIR directories.

Conditionals

Tests: `[[ ]]`

- `[[ expression ]]` returns 0=true/success or 1=false/failure depending on the evaluation of the conditional *expression*.
- `[[ expression ]]` is a new upgraded variation on `test` (also known as `[ ... ]`), all the earlier examples with single brackets that one can find online will also work with double
- Inside the double brackets it performs tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal
- Conditional expressions can be used to test file attributes and perform string and arithmetic comparisons

Selected examples file attributes and variables testing:

- `-f file` true if is a file
- `-r file` true if file exists and readable
- `-d dir` true if is a directory
- `-e file` true if file/dir/etc exists in any form
- `-z string` true if the length of string is zero (always used to check that var is not empty)
- `-n string` true if the length of string is non-zero
- `file1 -nt file2` true if *file1* is newer (modification time)
- many more others

```
# check that directory does not exist before creating one
# where $dir is a variable
[[ -d $dir ]] || mkdir $dir

# checks that file exists
[[ -r $file ]] && mv $file ${file}.$(date +%Y-%m-%d) || echo $file is either non-
readable or does not exist
# in the script
[[ -r $file ]] && mv $file ${file}.$(date +%Y-%m-%d) || { echo $file does not exist;
exit 1; }

# Check if script/function is given an argument
[[ -z $1 ]] && { echo no argument; exit 1; }

# Often used alternative to ${var:-a_value} or ${var:?not defined}
[[ -n $var ]] || echo var is not defined
```

Note that integers have their own construction `(( expression ))` (we come back to this), though `[[ ]]` will work for them too. The following are more tests:

- `==` strings or integers are equal (`=` also works)
- `!=` strings or integers are not equal
- `string1 < string2` true if *string1* sorts before *string2* lexicographically
- `>` vice versa, for integers greater/less than
- `string =~ pattern` matches the pattern against the string
- `&&` logical AND, conditions can be combined
- `||` logical OR
- `!` negate the result of the evaluation
- `()` group conditional expressions

```
# If 'var' is ... then
[[ $var == 'some_value' ]] && ...

# If path is ... then
[[ $(pwd) == /some/path ]] && ...

# grouping () and booleans && || within the [[ ]]
[[ $(hostname -s) == kosh && ($(pwd) == $WORK || $(pwd) == $SCRATCH) ]] ...

# note that [[ ]] always require spaces before and after brackets (!)
```

In addition (old school), double brackets inherit several operands to work with integers mainly:

- `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge` equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal

```
# Some use cases for [[ ]]

# a popular way to check input arguments, if no input, exit (in functions
# 'return 1').  Remember, $# is special variable for number of arguments.
[[ $# -eq 0 ]] && { echo Usage: $0 arguments; exit 1; }

# if dir exists and is not empty, then archive it
d=path/to/dir; [[ -d $d && $(ls -A $d) ]] && tar caf $(basename $d).$(date +%Y-%m-
%d).tar.gz $d

# append PATH function
# as an exercise, we will re-implement this with the matching operator =~, see below
appendPATH() {
  local dpath=${1:?directory is missing} && \
  [[ -d $dpath && ! $(echo $PATH|grep $dpath) ]] && export PATH+=:$dpath
}
```

The matching operator `=~` brings more opportunities, because regular expressions come in play. Matched strings in parentheses assigned to *${BASH_REMATCH[]}* array elements.

```
# change shell on the Linux server if it is not BASH
[[ ! $SHELL =~ bash ]] && chsh -s /bin/bash
```

- Regular expressions (regexs) are basically a mini-language for searching within, matching, and replacing text in strings.
- They are extremely powerful and basically required knowledge in any type of text processing.
- Yet there is a famous quote by Jamie Zawinski: “Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.” This doesn’t mean regular expressions shouldn’t be used, but used carefully. When writing regexs, start with a small pattern and slowly build it up, testing the matching at each phase, or else you will end up with a giant thing that doesn’t work and you don’t know why and can’t debug it. There are also online regex testers which help build them.
- While the basics (below) are the same, there are different forms of regexs! For example, the `grep` program has regular regexs, but `grep -E` has extended. The difference is mainly in the special characters and quoting. Basically, check the docs for each language (Perl, Python, etc) you want to use regexs in.

Selected operators:

- `.` matches any single character
- `?` the preceding item is optional and will be matched, at most, once
- `*` the preceding item will be matched zero or more times
- `+` the preceding item will be matched one or more times
- `{N}` the preceding item is matched exactly N times
- `{N,}` the preceding item is matched N or more times
- `{N,M}` the preceding item is matched at least N times, but not more than M times
- `[abd]`, `[a-z]` a character or a range of characters/integers
- `^` beginning of a line
- `$` the end of a line
- `()` grouping items, this what comes to ${BASH_REMATCH[@]}

```
# match an email
email='jussi.meikalainen@aalto.fi'; regex='(.*)@(.*)'; [[ "$email" =~ $regex ]]; echo
${BASH_REMATCH[*]}

# extract a number out of the text
txt='A text with #1278 in it'; regex='#([0-9]+ )'; [[ "$txt" =~ $regex ]] && echo
${BASH_REMATCH[1]} || echo do not match

# case insensitive matching
var1=ABCD, var2=abcd; [[ ${var1,,} =~ ${var2,,} ]] && ...
```

For case insesitive matching, alternatively, in general, set `shopt -s nocasematch` (to disable it back `shopt -u nocasematch`)

Conditionals: if/elif/else

Yes, we have `[[ ]] && ... || ...` but scripting style is more logical with if/else construction:

```
if condition; then
  command1
elif condition; then
  command2
else
  command3
fi
```

At the *condition* place can be anything what returns an exit code, i.e. `[[ ]]`, command/function, an arithmetic expression `$(( ))`, or a command substitution.

```
# to compare two input strings/integers
if [[ "$1" == "$2" ]]
then
  echo The strings are the same
else
  echo The strings are different
fi

# checking command output
if ping -c 1 google.com &> /dev/null; then
  echo Online
elif ping -c 1 127.0.0.1 &> /dev/null; then
  echo Local interface is down
else
  echo No external connection
fi

# check input parameters
if [[ $# == 0 ]]; then
  echo Usage: $0 input_arg
  exit 1
fi
... the rest of the code
```

Expanding *tarit.sh* to a script

```
#!/bin/bash

# usage: tarit.sh <dirname>

dir=$1

# if directory name is given as an argument
if [[ -d $dir ]]; then
  archive=$(basename $dir).$(date +%Y-%m-%d).tar.gz

# if no argument, then the current directory
elif [[ -z $dir ]]; then
  dir='.'
  archive=$(basename $(pwd)).$(date +%Y-%m-%d).tar.gz

# otherwise error and exit
else
  echo $dir does not exist or empty
  exit 1
fi

# run tar
tar caf $archive $dir
```

case

Another option to handle flow, instead of nested *ifs*, is `case`.

```
read -p "Do you want to create a directory (y/n)? " yesno   # expects user input
case $yesno in
  y|yes)
    dir='dirname'
    echo Creating a new directory $dir
    mkdir $dir
    cd $dir
    ;;
  n|no)
    echo Proceeding in the current dir $(pwd)
    ;;
  *)
    echo Invalid response
    exit 1
    ;;
esac
# $yesno can be replaced with ${yesno,,} to convert to a lower case on the fly
```

In the example above, we introduce `read`, a built-in command that reads one line from the standard input or file descriptor.

`case` tries to match the variable against each pattern in turn. Understands patterns rules like `*, ?, [], |`.

Here is the *case* that could be used as an idea for your *~/.bashrc*

```
host=$(hostname)
case $host in
  myworkstation*)
    export PRINTER=mynearbyprinter
    # making your promt smiling when exit code is 0 :)
    PS1='$(if [[ $? == 0 ]]; then echo "\[\e[32m\]:)"; else echo "\[\e[31m\]:("; fi)\
[\e[0m\] \u@\h \w $ '
  ;;
  triton*)
    [[ -n $WRKDIR ]] && alias cwd="cd $WRKDIR" && cwd
  ;;
  kosh*|brute*|force*)
    PS1='\u@\h:\w\$'
    export IGNOREEOF=0
  ;&
  *.aalto.fi)
    kinit
  ;;
  *)
    echo 'Where are you?'
  ;;
esac
```

`;;` is important, if replaced with `;&`, execution will continue with the command associated with the next pattern, without testing it. `;;&` causes the shell to test next pattern. The default behaviour with `;;` is to stop matches after first pattern has been found.

```
# create a file 'cx'
case "$0" in
 *cx) chmod +x "$@" ;&
 *cw) chmod +w "$@" ;;
 *c-w) chmod -w "$@" ;;
 *) echo "$0: seems that file name is somewhat different"; exit 1 ;;
esac

# chmod +x cx
# ln cx cw
# ln cx c-w
# to make a file executable 'cx filename'
```

The following example is useful for Triton users: array jobs, where one handles array subtasks based on its index.

Exercise 2.3

✍ Exercise

- Re-implement the above mentioned example `... [[ -d $d && ! $(echo $PATH|grep $d) ]] ...` with the matching operator `=~`
- Improve the `tarit.sh` script we developed recently:
  - add check for the number of the given arguments. Hint: `$#` must be zero or one.
  - validate the given path like *path/to/file*. Hint: `[[ $d =~ regexpr ]]`, the path may have only alphanumeric symbols, dots, underscore and slashes as a directory delimiter.
- Expand `cx` script:
  - check that $@ not empty
  - add option for `cr` that would add read rights for all. Hint: `chmod a+r ...`
- (*) Write a function (add to *bin/functions*) that validates an IPv4 using `=~` matching operator. The function should fail incorrect IPs like 0.1.2.3d or 233.204.3.257. The problem should be solved with the regular expression only. Use `return` command to exit with the right exit code.

Loops

Arithmetic

BASH works with integers only (no floating point) but supports wide range of arithmetic operators using arithmetic expansion `$(( expression ))`.

- All tokens in the expression undergo parameter and variable expansion, command substitution, and quote removal. The result is treated as the arithmetic expression to be evaluated.
- Arithmetic expansion may be nested.
- Variables inside double parentheses can be without a $ sign.
- BASH has other options to work with the integers, like `let`, `expr`, `$[]`, and in older scripts/examples you may see them.

Available operators:

- `n++`, `n--`, `++n`, `--n` increments/decrements
- `+`, `-` plus minus
- `**` exponent
- `*`, `/`, `%` multiplication, (truncating) division, remainder
- `&&`, `||` logical AND, OR
- `expr?expr:expr` conditional operator (ternary)
- `==`, `!=`, `<`, `>`, `>=`, `<=` comparison
- `=`, `+=`, `-=`, `*=`, `/=`, `%=` assignment
- `()` sub-expressions in parentheses are evaluated first
- The full list includes bitwise operators, see `man bash` section *ARITHMETIC EVALUATION*.

```
# without dollar sing value is not returned, though 'n' has been incremented
n=10; ((n++))

# but if we need a value
n=10; m=3; q=$((n**m))

# here we need exit code only
if ((q%2)); then echo odd; fi
if ((n>=m)); then ...; fi

# condition ? integer_value_if_true : integer_value_if_false
n=2; m=3; echo $((n<m?10:100))

# checking number of input parameters, if $# is zero, then exit
# (though the alternative [[ $# == 0 ]] is more often used, and intuitively more clear)
if ! (($#)); then echo Usage: $0 argument; exit1; fi
```

```
# sum all numbers from 1..n, where n is a positive integer
# Gauss method, summing pairs
if (($#==1)); then
  n=$1
else
  read -p 'Give me a positive integer ' n
fi
echo Sum from 1..$n is $((n*(n+1)/2))
```

Left for the exercise: make a summation directly 1+2+3+...+n and compare performance with the above one.

For anything more mathematical than summing integers, one should use something else, one of the option is `bc`, often installed by default.

```
# bc -- an arbitrary precision calculator language
# compute Pi number
echo "scale=10; 4*a(1)" | bc -l
```

A way to get percentage with the floating point

```
done=5; total=12; printf "%.2f%%\n" "$((10**3*100*done/total))e-3"
```

For loops

BASH offers several options for iterating over the lists of elements. The options include

- Basic construction `for arg in item1 item2 item3 ...`
- C-style *for loop* `for ((i=1; i <= LIMIT ; i++))`
- while and until constructs

Simple loop over a list of items:

```
# note that if you put 'list' in quotes it will be considered as one item
for dir in dir1 dir2 dir3/subdir1; do
  echo "Archiving $dir ..."
  tar -caf ${dir//\/.}.tar.gz $dir && rm -rf $dir
done
```

If path expansions used (*, ?, [], etc), loop automatically lists current directory:

```
# example below uses ImageMagick's utlity to convert all *.jpg files
# in the current directory to *.png.
# i.e. '*.jpg' similar to 'ls *.jpg'
for f in *.jpg; do
 convert "$f" "${f/.jpg/.png}"   # quotes to avoid issues with the spaces in the name
done

# another command line example renames *.JPG and *.JPEG files to *.jpg
# note: in reality one must check that a newly created *.jpg file does not exist
for f in *.JPG *JPEG; do mv -i "$f" "${f/.*/.jpg}"; done

# do ... done in certain contexts, can be omitted by framing the command block within
curly brackets
# and certain for loop can be written in one line as well
for i in {1..10}; { echo i is $i; }
```

If *in list* omitted, *for* loop goes through script/function input parameters `$@`

```
# here is a loop to rename files which names are given as input parameters
# touch file{1..3}; ./newname file1 file2 file3
for old; do
  read -p "old name $old, new name: " new
  mv -i "$old" "$new"
done
```

Note: as side note, while working with the files/directories, you will find lots of examples where loops can be emulated by `find ... -print0 | xargs -0 ...` pipe.

Loop output can be piped or redirected:

```
# loop other all users (on a server) to find out who has logged in within last month
for u in $(getent group triton-users | cut -d: -f4 | tr ',' ' '); do
  echo $u: $(last -Rw -n 1 $u | head -1)
done | sort > filename
```

The *list* can be anything what produces a list, like Brace expansion *{1..10}*, command substitution etc.:

```
# on Triton, do something to all pending jobs based on squeue output
for jobid in $(squeue -h -u $USER -t PD -o %A); do
  scontrol update JobId=$jobid StartTime=now+5days
done

# using find to make a list of files to deal with; the benefit here is that you work
# with the filename as a variable, which gives you flexibility as comparing to
# 'find ... -exec {}' or 'find ... print0 | xargs -0 ...'
for f in $(find . -type f -name '*.sh'); do
  if ! bash -n $f &>/dev/null; then
    mv $f ${f/.sh/.fixme.sh}
  fi
done
```

C-style, expressions evaluated according to the arithmetic evaluation rules:

```
N=10
for ((i=1; i <= N ; i++))  # LIMIT with no $
do
  echo -n "$i "
done
```

Loops can be nested.

While/until loops

Other useful loop statement are `while` and `until`. Both execute continuously as long as the condition returns exit status zero/non-zero correspondingly.

```
while condition; do
  command1
  command2
  ...
done

# sum of all numbers 1..n; n expected as an argument
n=$1 i=1
until ((i > n)); do
  ((s+=i)); ((i++))
done
echo Sum of 1..$n is $s

# endless loop, can be stopped with Ctrl-c or killed
# drop an email every 10 minutes about running jobs on Triton
# can be used in combination with 'screen', and run in background
while true; do
  squeue -t R -u $USER | mail -s 'running jobs' mister.x@aalto.fi
  sleep 600
done

# with the help of 'read var' passes file line by line,
# IFS= variable before read command to prevent leading/trailing
# whitespace from being trimmed
input='/path/to/txt/file'
while IFS= read -r line; do
 echo $line
done < "$input"

# reading file fieldwise, IFS= is a delimiter, note quoting with \
file='/path/to/file.csv'
while IFS=\; read -r f1 f2 f3 f4; do
  printf 'Field1: %s, Field2: %s, Field4: %s\n' "$f1" "$f2" "$f4"
done < "$file"

# process substitution
while IFS= read -r line; do
  # do something with the lines
done < <(file -b *)
# instead, one can mistakenly try 'file -b * | while read line; do ... done'
# with pipe, 'while' body will be run in a subshell, and thus all variables
# used inside the loop will die when loop is over
```

All the things mentioned above for *for* loop applicable to `while` / `until` loops.

***printf* should be familiar to programmers, allows formatted output**

similar to C printf. [1]

Loop control

Normally *for* loop iterates until it has processed all its input arguments. *while* and *until* loops iterate until the loop control returns a certain status. But if needed, one can terminate loop or jump to a next iteration.

- `break` terminates the loop
- `continue` jump to a new iteration
- **`break n` will terminate *n* levels of loops if they are nested, otherwise terminated only**

    loop in which it is embedded. Same kind of behaviour for `continue n`.

Even though in most of the cases you can design the code to use conditionals or alike, *break* and *continue* certainly add the flexibility.

```
# here we expand an earlier example to avoid errors in case $f is missing/not accesible
for f in *.JPG *.JPEG; do
  [[ -r "$f" ]] || { echo "$f is missing on inaccessible"; continue; }
  mv -i "$f" "${f/.*/.jpg}"
done
```

Exercise 2.4

✍ Exercise

- Using `for d; do ... done` expand *tarit.sh* so that it would accept multiple directories. If no directory given, it still suppose to archive the current one.
- Using `for` loop rename all the files in the current directory tree with the *.txt* extension to *.fixed.txt*. Step #1: create dummy .txt files first with `mkdir d{1..3}; touch d{1..3}/{1..3}.txt`. Step #2: combine 'for' loop with 'find': `for f in $(find . -name '*.txt'); do ... done`.
- Use this page `while` example with *.cvs*, take the *demospace/Finnish_Univ_students_2018.csv* to count total number of students around Finland. Tip: add checking that the number of students field is a number `[[ $totalnmb =~ ^[0-9]+$ ]]`
- Using built-in arithmetic write a script *daystill.sh* that counts a number of days till a deadline (vacation/salary). Script should take a date as an argument, where date format is `days_till 2019-6-1`. Tip #1: `date -d GIVEN_DATE +"%s"` returns number of seconds till GIVEN_DATE since 1970-01-01 00:00:00 UTC. `date +%s` returns seconds till now. Tip #2: enough if you convert seconds to a number of days roughly `$(( .../60/60/24))`.
- Make script that takes a list of files and checks if there are files in there with the spaces in the name, and if there are, rename them by replacing spaces with the underscores. Use BASH's builtin functionality only.
    - As a study case, compare it against `find . -depth -name '* *' -execdir rename 's/ /_/g' {} \;`
- Write separate scripts that count a sum of any *1+2+3+4+..+n* sequence, both the Gauss version (see above) and summation with `for ((...))`. Where *n* is an argument, like *gauss.sh 1000*. Benchmark them with `time` for n=10000 or more.

    - (*) Implement both methods within one script as functions and benchmark them whithin the file
    - (*) For the direct summation one can avoid loops, how? Tip: discover `eval $(echo {1..$n})`
- (*) Get familiar with the `getent` and `cut` utilities. Join them with a loop construction to write a *mygetentgroup* script or just a oneliner that generates a list of users and their real names that belong to a given group. Like:

```
$ mygetentgroup group_name
meikalaj1: Jussi Meikäläinen
meikalam1: Maija Meikäläinen
...``
```

- (*) To Aalto users: on kosh/lyta run `net ads search samaccountname=$USER accountExpires 2>/dev/null` to get your account expiration date. It is a 18-digit timestamp, the number of 100-nanoseconds intervals since Jan 1, 1601 UTC. Implement a function that accept a user name, and if not given uses current user by default, and then converts it to the human readable time format. Tip: http://meinit.nl/convert-active-directory-lastlogon-time-to-unix-readable-time
    - Expand it to handle “Got 0 replies” response, i.e. account name not found.

[1] https://wiki.bash-hackers.org/commands/builtin/printf

Arrays, input, Here Documents

Arrays

BASH supports both indexed and associative one-dimensional arrays. Indexed array can be declared with `declare -a array_name`, or first assignment does it automatically (note: indexed arrays only):

```
arr=(my very first array)
arr=('Otakaari 1' Espoo 02150 [6]='PL 11000')
arr[5]=AALTO
```

To access array elements (the curly braces are required, unlike normal variable expansion):

```
# elements one by one
echo ${arr[0]} ${arr[1]}

# array values at once
${arr[@]}

# indexes at once
${!arr[@]}

# number of elements in the array
${#arr[@]}

# length of the element number 2
${#arr[2]}

# to append elements to the end of the array
arr+=(value)

# assign a command output to array
arr=($(command))

# emptying array
arr=()

# to destroy, delete an array
unset arr

# to unset a single array element
unset arr[6]

# sorting array
IFS=$'\n' sorted=($(sort <<<"${arr[*]}"))

# array element inside arithmetic expanssion requires no ${}
((arr[$i]++))

# split a string like 'one two three etc' or 'one,two,three,etc' to an array
# note that IFS=', ' means that separator is either space or comma, not a sequence of
them
IFS=', ' read -r -a arr <<< "$string"

# spliting a word to an array letter by letter
word=qwerty; arr=($(echo $word | grep -o .))
```

Loops through the indexed array:

```
for i in ${!arr[@]}; do
  echo arr[$i] is ${arr[$i]}
done
```

Negative index counts back from the end of the array, *[-1]* referencing to the last element.

Quick ways to print array with no loop:

```
# with keys, as is
declare -p arr

# indexes -- values
echo ${!arr[@]} -- ${arr[@]}

# array elements values one per line
printf "%s\n" "${arr[@]}"
```

Passing an array to a function as an argument could be the use case when you want to make it local:

```
f() {
  local arr=(${!1})    # pass $1 argument as a refence
  # do something to array elements
  echo ${arr[@]}
}

# invoke the function, huom that no changes have been done to the original arr[@]
arr=(....)
f arr[@]
```

BASH associative arrays (this type of array supported in BASH since version 4.2) needs to be declared first (!) `declare -A asarr`.

Both indexed arrays and associative can be declared as an array of integers, if all elements values are integers `declare -ia array` or `declare -iA`. This way element values are treated as integers always.

```
asarr=([university]='Aalto University' [city]=Espoo ['street address']='Otakaari 1')
asarr[post_index]=02150
```

Addressing is similar to indexed arrays:

```
for i in "${!asarr[@]}"; do
  echo asarr[$i] is ${asarr[$i]}
done
```

Even though key can have spaces in it, quoting can be omitted.

```
# use case: your command returns list of lines like: 'string1 string2'
# adding them to an assoative array like: [string1]=string2
declare -A arr
for i in $(command); do
  arr+=(["${i/ */}"]="${i/* /}")
done
```

Variable expanssions come out in the new light:

```
# this will return two elements of the array starting from number 1
${arr[@]:1:2}

# all elements without last one
${arr[@]:0:${#arr[@]}-1}

# parts replacement will be applied to all array elements
declare -A emails=([Vesa]=vesa@aalto.fi [Kimmo]=kimmo@helsinki.fi
[Anna]=anna@math.tut.fi)
echo ${emails[@]/@*/@gmail.com}
# returns: vesa@gmail.com anna@gmail.com kimmo@gmail.com
```

For a sake of demo: let us count unique users and their occurances (yes, one can do it with ‘uniq -c’ :)

```
# declare assoative array of integers
declare -iA arr

for i in $(w -h | cut -c1-8); do   # get list of currenly logged users into loop
  for u in ${!arr[@]}; do   # check that they are unique
    if [[ $i == $u ]]; then
      ((arr[$i]++))
      continue 2
    fi
  done
  arr[$i]=1  # if new, add a new array element
done

for j in ${!arr[@]}; do    # printing out
  echo ${arr[$j]} $j
done
```

Another working demo: script that automates backups or just makes a sync of data to a remote server. Same can be adapted to copy locally, to a usb drive or alike.

```
# array of directories to be backuped, to skip one, just comment with #
declare -A dirs
dirs[wlocal]=/l/$USER
dirs[xpproject]=/m/phys/extra/project/xp
dirs[homebin]=$HOME/bin

cmd='/usr/bin/rsync'                    # rsync
args="-auvW --delete --progress $@"     # accept extra args, like '-n' for the dryrun
first
serv='user@server:backups'              # copying to ~/backups that must exist

# array key is used for the remote dir name
for d in ${!dirs[@]}; do
  echo "Syncing ${dirs[$d]}..."
  $cmd $args ${dirs[$d]}/ $serv/$d
done
```

Exercise 2.5

✍ Exercise

- make a script/function that produces an array of random numbers, make sure that numbers are unique. Print the array nicely using `printf` for formating.
  - one version should use BASH functionality only (Tip: `$RANDOM`)
  - the other one can use `shuf`
- (*) Pick up the `ipvalid` function that we have developed earlier, implement IP matching regular expression as `^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$` and work with the ${BASH_REMATCH[*]} array to make sure that all numbers are in the range 0-255

Working with the input

User input can be given to a script in three ways:

- as command arguments, like `./script.sh arg1 arg2 ...`
- interactively from keyboard with `read` command
- as standard input, like `command | ./script`

Nothing stops from using a combination of them or all of the approaches in one script. Let us go through the last two first and then get back to command line arguments.

`read` can do both: read from keyboard or from STDIN

```
# the command prints the prompt, waits for the response, and then assigns it
# to variable(s)
read -p 'Your names: ' firstn lastn

# read into array, each word as a new array element ('arr' declared automatically)
read -a arr -p 'Your names: '
```

Given input must be checked (!) with a pattern, especially if script creates directories, removes files, sends emails based on the input.

```
# request a new directory name till correct one is given (interrupt with Ctrl-C)
regexp='^[a-zA-Z0-9/_-]+$'
until [[ "$newdir" =~ $regexp ]]; do
  read -p 'New directory: ' newdir
done
```

`read` selected options

- `-a <ARRAY>` read the data word-wise into the specified array <ARRAY> instead of normal variables
- `-N <NCHARS>` reads <NCHARS> characters of input, ignoring any delimiter, then quits
- `-p <PROMPT>` the prompt string <PROMPT> is output (without a trailing automatic newline) before the read is performed
- `-r` raw input - disables interpretion of backslash escapes and line-continuation in the read data
- `-s` secure input - don't echo input if on a terminal (passwords!)
- `-t <TIMEOUT>` wait for data <TIMEOUT> seconds, then quit (exit code 1)

`read` is capable of reading STDIN, case like `command | ./script`, with `while read var` it goes through the input line by line:

```
# IFS= is empty and echo argument in quotes to make sure we keep the format
# otherwise all spaces and new lines shrinked to one and leading/trailing whitespace
trimmed
while IFS= read -r line; do
  echo "line is $line"    # do something useful with $line
done

# To check current $IFS
cat -A <<<"$IFS"
```

Though in general, whatever comes from STDIN can be proceeded as:

```
# to check that STDIN is not empty
if [[ -p /dev/stdin ]]; then
  # passing STDIN to a pipeline  (/dev/stdin can be omitted)
  cat /dev/stdin | cut -d' ' -f 2,3 | sort
fi
```

Other STDIN tricks that one can use in the scripts:

```
# to read STDIN to a variable, both commands do the same
var=$(</dev/stdin)
var=$(cat)
```

In the simplest cases like `./script arg1 arg2 ...`, you check *$#* and then assign *$1*, *$2*, ... the way your script requires.

```
# here we require exactly two arguments
if (($#==2)); then
  var1=$1 var2=$2
  # ... do something useful
else
  echo 'Wrong amount of arguments'
  echo "Usage: ${0##*/} arg1 arg2"
  exit 1
fi
```

To work with all input arguments at once we have *$@*:

```
# $# is a number of arguments on the command line, must be non-zero
if (($#)); then
  for i; do
    echo "$i"
    # ... do something useful with each element of $@
    # note that 'for ...' uses $@ by default if no other list given with 'in ...'
  done
else
  echo 'No command line arguments given'
fi
```

As a use case, our *tarit.sh* script. The script can accept STDIN and arguments, so we check both:

```
# Usage: tarit.sh [dirname1 [dirname2 [dirname3 ...]]]
# or     command | tarit.sh

# by default no directories to archive. i.e. current
args=''

# checking for STDIN, if any, assigning STDIN to $args
[[ -p /dev/stdin ]] && args=$(</dev/stdin)

# if arguments are given, appending the $args with $@
(($#)) && args+=" $@"

# no arguments, no stdin, then it is a current dir
[[ -z "$args" ]] && args="$(pwd)"

# by now we should have a directory list in $args to archive
for d in $args; do
  # checking that directory exists, if so, archive it
  if [[ -d "$d" ]]; then
    echo Archiving $d ...
    tar caf ${d##*/}.$(date +%Y-%m-%d).tar.gz "$d"
  else
    echo "   $d does not exist, skipping."
  fi
done
```

Often, the above mentioned ways are more than enough for simple scripts. But what if options and arguments are like `./script [-f filename] [-z] [-b] [arg1 [arg2 [...]]]` or more complex? (common notaion: options in the square brackets are optional). What if you write a production ready script that will be used by many other as well?

It is were `getopt` offers a more efficient way of handling script's input options. In the simplest case `getopt` command (do not get confused with `getopts` built-in BASH function of similar kind) requires two parameters to work: first is a list of valid input options – sequence of letters and colons. If letter followed by a colon, the option requires an argument, if folowed by two colons, argument is optional. For example, the string `getopt "sdf:"` says that the options -s, -d and -f are valid and -f requires an argument, like *-f filename*. The second argument required by *getopt* is a list of input parameters (options + arguments) to check, i.e. just `$@`.

Let us use *cx* script as a demo:

```
# common usage function with the exit at the end
usage() {
  echo "Usage: $sname [options] file [file [file...]]"
  echo '       -a, gives access to all, like a+x, by default +x'
  echo '       -d <directory/path/bin>, path to the bin directory'
  echo "          can be used in 'cx' to copy a new script there"
  echo '       -a, gives access to all, like a+x, by default +x'
  echo '       -d <directory/path/bin>, path to the bin directory'
  echo "          can be used in 'cx' to copy a new script there"
  echo '       -v, verbose mode for chmod'
  echo '       -h, this help message'
  exit 1
}

# whole trick is in this part: getopt validates the input parameters,
# structures them by dividing options and arguments with --,
# and returns them to a variable
# then they are reassigned back to $@ with 'set --'
opts=$(getopt "avhd:" "$@") || usage
set -- $opts

# defining variables' default values
ALL=''
CMD='/usr/bin/chmod'
sname=${0##*/}  # the name this script was called by

# by now we have a well structured $@ which we can trust.
# to go through options one by one we start an endless 'while' loop
# with the nested 'case'. 'shift' makes another trick, every time
# it is invoked it is equal to 'unset $1', thus $@ arguments are
# "shifted down", $2 becomes $1, $3 becomes $2, etc
# 'getopt' adds -- to $@ which separates valid options and the rest
# that did not qualify, when it comes to '--' we 'break' the loop
while true; do
  case ${1} in
    -h) usage ;; # output help message and exit
    -a) ALL=a ;; # if -a is given we set ALL
    -v) CMD+=' -v' ;; # if verbose mode required
    -d) shift # shift to take next item as a directory path for -d
        BINDIR="$1"
        if [[ -z "$BINDIR" || ! -d "$BINDIR" ]]; then
          echo "ERROR: the directory does not exist"
          usage
        fi
     ;;
    --) shift; break ;;   # remove --
  esac
  shift
done

# script body

case "$sname" in
  cx*) $CMD ${ALL}+rx "$@" && \
       [[ -n "$BINDIR" ]] && cp -p $@ $BINDIR ;;
  cw*) $CMD ${ALL}+w "$@" ;;
  cr*) $CMD ${ALL}+r "$@" ;;
  c-w*) $CMD ${ALL}-w "$@" ;;
  *) echo "ERROR: no idea what $sname is supposed to do"; exit 1 ;;
esac
```

`getopt` can do way more, go for `man getopt` for details, as an example:

```
# here is getopt sets name with '-n' used while reporting errors: our script name
# accepts long options like '--filename myfile' along with '-f myfile'
getopt -n $(basename $0)  -o "hac::f:" --long "help,filename:,compress::"  -- "$@"
```

Exercise 2.6

✍ Exercise

- Using the latest *tarit.sh* (see lecture notes) version as an example, expand above *cx* script to accept STDIN, like `command | cx [options]`, where `command` produces a list of files. Example `find . -t file -name '*.sh' | cx -a -d /path/to/bin`.
- Using *cx* demo as an example, expand the latest version of our *tarit.sh* (see lecture notes) to make it accepting the following options and arguments: `tarit.sh -h -y -d <directory/with/backups> [dirname1 [dirname2 [dirname3 ...]]]`. By default, with no args, it still should make an archive of the current directory. `-h` returns usage info, `-d <directory/path/with/backups>` is a directory the tar archives will go to, your script has to check that directory exists, the script must also check whether a newly created archive already exist and if so, skip creating the archive with the corresponding warning message.
    - (*) `-y` should force overwriting already existing archive.
    - (*) `-s` should make script silent, so that no errors or other messages would come from any inline command.

Here Document, placeholders

A 'here document' and 'here string' take the line(s) following and send them to standard input. It's a way to send larger blocks to stdin.

```
# instead of 'echo $STRING | command ...'
command <<<$STRING

# instead of 'cat file | command ...'
command <<SomeMagicStopWord
The benefit is that one can use $var, $() etc in the text
The text ends with the Stop Word on a new line, the word can be any
SomeMagicStopWord
```

Often used for messaging, be it an email or dumping bunch of text to file.:

```
# NAME, SURNAME, EMAIL, DAYS are set earlier

mail -s 'Account expiration' $EMAIL<<END-OF-EMAIL
Dear $NAME $SURNAME,

your account is about to expire in $DAYS days.

$(date)

Best Regards,
Aalto ITS
END-OF-EMAIL
```

Or just outputting to a file (same can be done with echo commands):

```
cat <<EOF >filename
... text
EOF
```

One trick that is particularly useful is using this to make a long comment:

```
: <<\COMMENTS
here come text that is seen nowhere
there is no need to comment every single line with #
COMMENTS
```

Hint `<<\LimtiString` to turn off substitutions and place text as is with $ marks etc

In case you have a template file which contains variables as placeholders, replacing them:

```
# 'template' file like:
The name is $NAME, the email is $EMAIL

# command to substitute the placeholders and redirect to 'output' file
# the original 'template' file remains as is
NAME=Jussi EMAIL=jussi@gmail.com
cat template | while IFS= read -r line; do eval echo $line; done > output
# resulting file: The name is Jussi, the email is jussi@gmail.com
```

Traps, debugging, profiling

Catching kill signals: trap

What if your script generates temp file and you'd like to keep it clean even if script gets interrupted at the execution time?

The built-in `trap` command lets you tell the shell what to do if your script received signal to exit. It can catch all, but here listed most common by their numbers. Note that signals are one of the common ways of communicating with running processes in UNIX: you see these same numbers and names in programs like `kill`.

- 0 EXIT exit command
- 1 HUP when session disconnected
- 2 INT interrupt - often Ctrl-c
- 3 QUIT quit - often Ctrl-
- 9 KILL real kill command, it can't be caught
- 15 TERM termination, by `kill` command

```
# 'trap' catches listed signals only, others it silently ignores
# Usage: trap group_of_commands/function list_of_signals

trap 'echo Do something on exit' EXIT
```

Expanding the backup script from the Arrays section, this can be added to the very beginning:

```
interrupted() {
  echo 'Seems that backup has been interrupted in the middle'
  echo 'Rerun the script later to let rsync to finish its job'
  exit 1
}

trap interrupted 1 2 15
# ... the rest of the script
```

In other situation, instead of *echo*, one can come up with something else: removing temp files, put something to the log file or output a valuable error message to the screen.

Hint About signals see *Standard signals* section at `man 7 signal`. Like Ctrl-c is INT (aka SIGINT).

Debugging and profiling

BASH has no a debugger, but there are several ways to help with the debugging

Check for syntax errors without actual running it `bash -n script.sh`

Or echos each command and its results with `bash -xv script.sh`, or even adding options directly to the script. `-x` enables tracing during the execution, `-v` makes bash to be verbose. Both can be set directly from the command line as above or with `set -xv` inside the script.

```
#!/bin/bash -xv
```

To enable debugging for some parts of the code only:

```
set +x
... some code
set -x
```

If you want to check quickly a few commands, with respect to how variables or other substitutions look like, use DEBUG variable set to *echo*.

```
#!/bin/bash

$DEBUG command1 $arguments
command2

# call this script like 'DEBUG=echo ./script.sh' to see how *command1* looks like
# otherwise the script can be run as is.
```

One can also `trap` at the EXIT, this should be the very first lines in the script:

```
end() { echo Variable Listing: a = $a  b = $b; }
trap end EXIT  # will execute end() function on exit
```

For a sake of profiling one can use PS4 and `date` (GNU version that deals with nanoseconds). PS4 is a built in BASH variable which is printed before each command bash displays during an execution trace. The first character of PS4 is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is `+`. Add the lines below right after '#!/bin/bash'

```
# this will give you execution time of each command and its line number
# \011 is a tab
PS4='+\011$(date "+%s.%N")\011${LINENO}\011'
set -x
```

Optionally, if you want tracing output to be in a separate file:

```
PS4='+\011$(date "+%s.%N")\011${LINENO}\011'
exec 5> ${0##*/}.$$.x && BASH_XTRACEFD='5' && set -x
```

Or to get your script looking more professional, one can enable DEBUG, i.e. tracing only happens when you run as `DEBUG=profile ./script.sh`:

```
case $DEBUG in
  profile|PROFILE|p|P)
    PS4='+\011$(date "+%s.%N")\011${LINENO}\011'
    exec 5> ${0##*/}.$$.x && BASH_XTRACEFD='5' && set -x ;;
esac
```

For the larger scripts with loops and functions tracing output with the date stamps and line numbers can be summarized. For further discussion please take a look at [1]

[1] https://stackoverflow.com/questions/5014823/how-to-profile-a-bash-shell-script-slow-startup

Parallel, crontab, perl/awk/sed

Running in parallel with BASH

The shell doesn't do parallelzation in the HPC way (threads, MPI), but can run some simple commands at the same time without interaction.

The simplest way of parallelization is sending processes to a background and waiting in the script till they are done.:

```
# in the script body one may run several processes, like
command1 &
command2 &
command3 &
```

Here is an example that can be run as `time script` to demonstrate that execution takes 5 seconds, that is the timing of the longest chunk, and all the processes are run in parallel and finished before script's exit.:

```
# trap is optional, just to be on the safe side
# at the beginning of the script, to get child processes down on exit
trap 'killall $(jobs -p) 2>/dev/null' EXIT

# dummy sleep commands groupped with echo and sent to the background
for i in 1 3 5; do
  { sleep $i; echo sleep for $i s is over; } &
done

# 'wait' makes sure jobs are done before script is finished
# try to comment it to see the difference
wait
echo THE END
```

Putting `wait` at very end of the script makes it to wait till all the child processes are over and only then exit. Having `trap` at very beginning makes sure we kill all the process whatever happens to the script. Otherwise they may stay running on their own even if the script has exited.

Another way to run in parallel yet avoiding sending to the background is using `parallel`. This utility runs the specified command, passing it a single one of the specified arguments. This is repeated for each argument. Jobs may be run in parallel. The default is to run one job per CPU. If no command is specified before the `--`, the commands after it are instead run in parallel.

```
# normally the command is passed the argument at the end of its command line. With -i
# option, any instances of "{}" in the command are replaced with the argument.
parallel command {} -- arguments_list

# example of making a backup with parallel rsync
parallel -i rsync -auvW {}/ user@server:{}.backup -- dir1 dir2 dir3

# in case you want to run a command, say ten times, the arguments can be any dummy list
# normally parallel passes arguments at the end of the command, with '-i' they needs to
# be placed explicitly with '{}', or can be skipped, like here
parallel -i date -- {1..10}

#  if no command is specified before the --, the commands after it are instead run in
parallel,
parallel -- ls df "echo hi"
```

On Triton we have installed Tollef Fog Heen's version of parallel from *moreutils-parallel* CentOS' RPM. GNU project has its own though, with different syntax, but of exactly the same name, so do not get confused.

Crontab

Allows run tasks automatically in the background. Users may set their own crontabs. Once crontab task is set, it will run independently on whether you are logged in to the system or not.

Run `crontab -l` to list all your current cron jobs, `crontab -e` to start editor. When in, you may add one or several lines, then save what you have added and exit the editor normally.

```
# run 'script' daily at 23:30
30 23 * * * $HOME/bin/backup_script > /home/user/log/backup.log 2>&1

# every two hours on Mon-Wed,Fri
0 /2 * * 1-3,5 rm /path/to/my/tmp/dir/* >/dev/null 2>&1
```

The executable script could be a normal command, but crontab's shell has quite limited functionality, in case of anything more sofisticated than just a single command and a redirection you end up writing a separate script.

The first five positions corresponds to:

- minute (0-59)
- hour (0-23)
- day (1-31)
- month (1-12)
- day of week (0-7, 0 or 7 is Sunday)

Possible values are:

- `*` any value
- `,` value list separator
- `-` range of values
- `/` steps

You set your favorite editor: `export EDITOR=vim` (can be a part of ~/.bashrc).

As part of the crontab file you may set several environmet variables, like `MAILTO=name.surname@aalto.fi` to recieve an output from the script or any possible errors. If *MAILTO* is defined but empty (`MAILTO=""`), no mail is sent.

Perl, awk, sed

Powerful onliners. Please consult correspoding man pages and other docs for the details, here we provide some examples. As it was standed at very beginning of the course, shell, with all its functionality is only a glue in between all kind of utilities, like `grep` , `find` , etc. Perl, awk and sed are what makes terminal even more powerful. Even though Perl can do everything what can awk and sed, one still may find tons of examples with the later ones. Here we provide some of them.

Python is yet another alternative.

```
# set delimiter to : and prints the first field of each line of passwd file (user name)
awk -F: '{print $1}' /etc/passwd

# sort lines by length, several ways to do it
cat file | perl -e 'print sort { length($a) <=> length($b) } <>'
cat file | awk '{ print length, $0 }' | sort -n -s | cut -d" " -f2-

# placeholders replacement example above could be
NAME=Jussi EMAIL=jussi@gmail.com; sed -e "s/\$NAME/$NAME/" -e "s/\$EMAIL/$EMAIL/"
template

# inline word replacing in all files at once
perl -i -p -e "s/TKK/Aalto/g" *.html
```

About homework assignments

Available on Triton. See details in the *$course_directory*.

Contact

You can contact us via scip@aalto.fi.

About us

Linux shell tutorial is a course born out of Aalto University Science-IT.

To continue: course development ideas/topics

Additional topics:

- select command
- revise coreutils section, expand the examples and explanations, make it clear that BASH is about getting those small utilities to work together
- benchmark: C-code vs BASH, Python vs BASH, Perl vs BASH

Ideas for exercises

- function to find all broken links
- (homework?) Implement a profiler, that summarizes PS4/date output mentioned above
- (homework?) Script that makes ‘pe1 pe2 ... gpu32’ out of ‘pe[1-16],gpu11,gpu32’

In general, there could one script that one starts building from the first line up to a parallelization. Like backup script with rsync.

Git usage?

Bonus material

Parts that did not fit.

[FIXME: should be moved to another tutorial *SSH: beyond login*]

SSH keys and proxy (*bonus section*)

- SSH is the standard for connecting to remote computers: it is both powerful and secure.
- It is highly configurable, and doing some configuration will make your life much easier.

SSH keys and proxy jumping makes life way easier. For example, logging on to Triton from your Linux workstation or from kosh/lyta.

For PuTTY (Windows) SSH keys generation, please consult section “Using public keys for SSH authentication” at [1]

On Linux/Mac: generate a key on the client machine

```
ssh-keygen -o  # you will be prompted for a location to save the keys, and a passphrase for the keys. Make sure passphrase is strong (!)
ssh-copy-id aalto_login@triton.aalto.fi   # transfer file to a Triton, or/and any other host you want to login to
```

From now on you should be able to login with the SSH key instead of password. When SSH key added to the ssh-agent (once during the login to workstation), one can login automatically, passwordless.

Note that same key can be used on multiple different computers.

SSH proxy is yet another trick to make life easier: allows to jump through a node (in OpenSSH version 7.2 and earlier `-J` option is not supported yet, here is an old recipe that works on Ubuntu 16.04). By using this, you can directly connect to a system (Triton) through a jump host (kosh): On the client side, add to `~/.ssh/config` file (create it if does not exists and make it readable by you only):

```
Host triton triton.aalto.fi
    Hostname triton.aalto.fi
    ProxyCommand ssh YOUR_AALTO_LOGIN@kosh.aalto.fi -W %h:%p
```

[1] https://the.earth.li/~sgtatham/putty/0.70/htmldoc/

Instructor’s guide

Setting up instructions for the lecturer

Main terminal white&black with the enlarged font size. One small terminal at the top that shows commands to the learners.

- `export PROMPT_COMMAND='history -a'` # .bashrc or all the terminals one launches commands
- `tail -n 0 -F .bash_history`

Alternatively, `script` allows to follow the session even after sshing to a remote host plus command appear as soon as they are run. The regular expression can be adapted to the lecturer's PS1, this one assumes *]$ command*.

- `script -f demos.out` # action window
- `tail -n 1 -f demos.out | while read line; do [[ "$line" =~ \]\$\ ([^ ].+)$ ]] && echo ${BASH_REMATCH[1]}; done`