Vlaanderen
is supercomputing

Introduction to Linux

Ine Arts, Franky Backeljauw, Michele Pugno, Robin Verschoren

Version Spring 2026 – Part 1

VLAAMS
SUPERCOMPUTER
CENTRUM

Innovative Computing
for A Smarter Flanders

vscentrum.be

Overview

Part 1 – basics

- What is GNU/Linux?
  - available Linux-like environments
- The shell
  - what is the shell?
  - command-line basics
  - options & arguments
  - getting help
- The filesystem
  - navigating the filesystem
  - manipulating directories & files
  - using wildcard patterns
  - reading and editing text files
- Useful tools
  - download & extract files
  - comparing files and directories
  - processing text-formatted structured data
- Streams & pipelines
  - input and output streams & redirection
  - command pipelines
  - overview of frequently used commands

What is GNU/Linux?

- Unix-like computer **operating system** (OS)
  - free and open-source, worldwide community, active development
- Under the hood: **Linux kernel**
  - abstraction between hardware and software
  - device drivers, system calls, process and memory management, …
- Typically offers **GNU utilities and libraries**
  - basic tools to work with files, compile programs, …
  - e.g.: coreutils, binutils, Bash shell, …
- Comes in many flavours, called **distributions**
  - bundles desktop environments, applications, …

Red Hat Enterprise Linux fedora Rocky Linux openSUSE debian Canonical Ubuntu

Available Linux-like environments

Microsoft Windows

- Microsoft Subsystem for Linux (**WSL**)
  - by default, the installed Linux distribution will be **Ubuntu**
- Installation instructions *– for recent versions of Windows*
  - search for PowerShell, right-click on the icon and select "Run as administrator"
  - type this in the PowerShell window to install WSL

    ```
    wsl --install Ubuntu
    ```

  - restart your machine
- Optional: use Windows Terminal
  - use Command Prompt, PowerShell and `bash` (via WSL) from one application

Available Linux-like environments

Apple macOS

- Terminal app (built-in) or iTerm2
- note: macOS is based on BSD (Unix), thus offering BSD variants of commands
  - use **Homebrew** to install the GNU utilities – *first: run the one-line installation command*

    ```
    brew install coreutils findutils gnu-tar gnu-sed grep wget
    ```

  - the GNU variants of commands usually start with **g**
    - e.g., use (GNU) **g**sed instead of (BSD) sed
    - likewise for **g**ls, **g**grep, **g**tar, …
    - but not always, e.g., wget

Alternative options

- Use an **online terminal emulator** – e.g.: https://sandbox.bio/tutorials/playground

The shell – Part 1

Exploring the command line

What is the shell?

- A program that interprets commands and sends them to the OS

- Sometimes referred to as ”the terminal” or a “**Command-Line Interface**” (CLI)
  - waits for input and performs the requested tasks
  - the input language is a **scripting language** (variables, iterations, …)
  - provides access to 100s of commands/programs

- Different shell programs exist
  - on most Linux systems, the default shell is called **bash** (Bourne Again SHell)
  - note: on macOS, the default shell is zsh, but bash is also available

Command-line basics

bash

- $ and text preceding it is called the "**prompt**"
  - executing a command: type a command after the prompt and press the **Enter** key
  - autocompletion: type part of the command and press the **Tab** key (↹)

```
$ ls -l /etc/host↹
```

- **Linux systems are case and space sensitive**
  - files: `myfile` is not the same as `MyFile`
  - commands: spaces separate parts of commands

- Some keyboard shortcuts when using the Bash shell environment

Left ← and **Right** →	moving around the line	**Ctrl + a**	go to the beginning of the line
Up ↑ and **Down** ↓	browse the command history	**Ctrl + e**	go to the end of the line
Ctrl + r	backward history search	**Ctrl + l**	clear the screen

Hands-on

- Enter the following commands and try to interpret the output

```
$ echo Hello, world.   $ sleep 3
$ date                 $ time sleep 3
$ date --utc           $ who
$ whoami               $ echo $SHELL
$ hostname             $ echo -n Hello, world.
$ uptime               $ cal
$ clear                $ history
```

Hands-on

- Note: some commands may not be available yet
  - more software can be added by installing extra *packages*
  - installation instructions depend on OS and/or distributions

- **Linux** and **Windows WSL** (Ubuntu)
  - advice: first update the package list

    ```
    $ sudo apt update
    ```
  - when a command is missing, a message is shown

    ```
    $ cal
    Command 'cal' not found, but can be installed with:
    sudo apt install ncal
    ```
  - install the ncal package to make the cal command available

    ```
    $ sudo apt install ncal
    ```

Hands-on

- Note: some commands may not be available yet
  - more software can be added by installing extra *packages*
  - installation instructions depend on OS and/or distributions

- **macOS** (Homebrew)
  - note: on Homebrew packages are reffered to as *formulae*
  - optional: install a helper command (first time only)

    ```
    $ brew tap homebrew/command-not-found
    ```

    - this allows you to search for a missing command

      ```
      $ brew which-formula tree
      ```

  - install the tree formula to make the tree command available

    ```
    $ brew install tree
    ```

Anatomy of a command

- Single command: program that does one thing

```
$ command
```

- **Arguments** (parameters): provide the input/output that the command interacts with

```
$ command argument1 argument2 [...]
```

- **Options**: modify a command’s behavior (also called *flags*)

```
$ command -option
```

single dash + one letter (*short* form)

```
$ command --long-option
```

double dash + one word (*long* form)

- Generally, they compose as follows:

```
$ command [-o]... [--long-option]... [argument]...
```

Options & arguments

- Interpreted by the command itself → usage depends on the command
  - **convention : options first, non-option arguments last**
  - short options can be combined, the order often doesn't matter

    `$ date -R -u` = `$ date -Ru`

  - but for some commands, strict ordering rules apply

    ```
    $ find -maxdepth 2 -type f
    ```

  - non-option arguments often refer to a filename

    ```
    $ less myfile
    ```

  - but not always

    ```
    $ echo "This is an example"
    $ date +"%A %e %B"
    ```

Types of commands

type

- A command can be either:
  - any **program** (or script) on the system
    - use `which` to find out where the program is located/installed
  - a **built-in** shell command
    - get an overview with `man builtin`
  - an **alias** or (user-defined) shorthand for a more complex command
    - use `alias` to see the currently defined aliases
  - a (user-defined) shell **function**

Getting help

man

- Documentation for commands is available as online Linux man pages
  - *ask Google or ChatGPT for help – the web is your friend!*

- Or directly from the command-line itself
  - ask a command about its use with the `--help` or `-h` options (if available)

    ```
    $ ls --help
    ```

  - manual pages for commands – use **q** to quit

    ```
    $ man ls
    ```

  - more elaborate info manuals – use **q** to quit

    ```
    $ info ls
    ```

- Search man pages for keywords

  ```
  $ man -k <keyword>
  ```

Getting help

man

- Efficiently reading man pages

↓ / ↑ or `j` / `k`	scrolling up or down
`h`	help for the man page viewer
`q`	quit reading the man page

- Searching through man pages

`/ + ”word” + Enter`	search for the given word
`n`	find the *next* occurrence
`N`	find the *previous* occurrence

- Conventions for describing key combinations

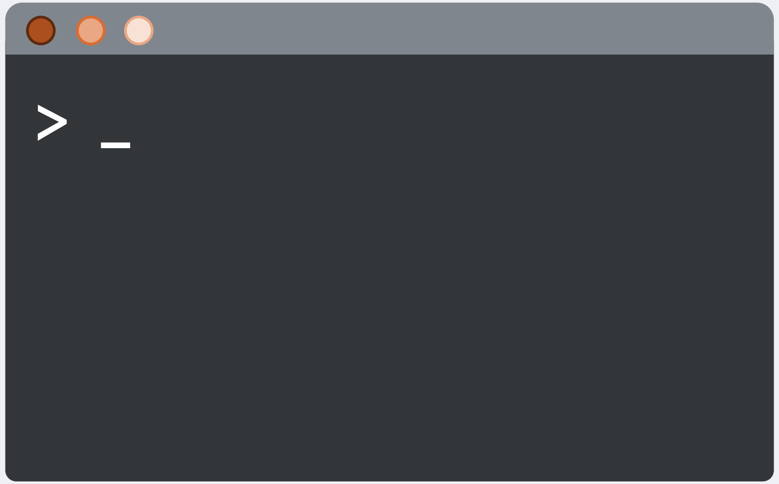
`^-<key>` = `Ctrl + <key>`	press Ctrl and the given key together
`C-<key>` = `Ctrl + <key>`	
`M-<key>` = `Alt + <key>`	M stands for “Meta” key (*note: Option on Apple keyboards*)

Hands-on

- Get help for some of the commands from the previous hands-on
  - try browsing through the man page
  - what are the options and which arguments does it accept?
  - try searching for some words in the man page

- For those using macOS
  - install the GNU coreutils: `brew install coreutils`
  - look at the man page of the commands `ls` and `gls`
  - what is the difference between these commands?

Summary – The shell



```
> _
```

- In your **terminal** window, you interact with the (bash) **shell**
- You type **commands** with **options** and **arguments**
  `$ command [--option]... [argument]...`
- You can not know *all* the options and exceptions:
  - use **man pages**
  - use the **web**

Vlaanderen
is supercomputing

The filesystem – Part 1

Navigating the filesystem

Manipulating files & directories

VLAAMS
SUPERCOMPUTER
CENTRUM

*Innovative Computing
for A Smarter Flanders*

vscentrum.be

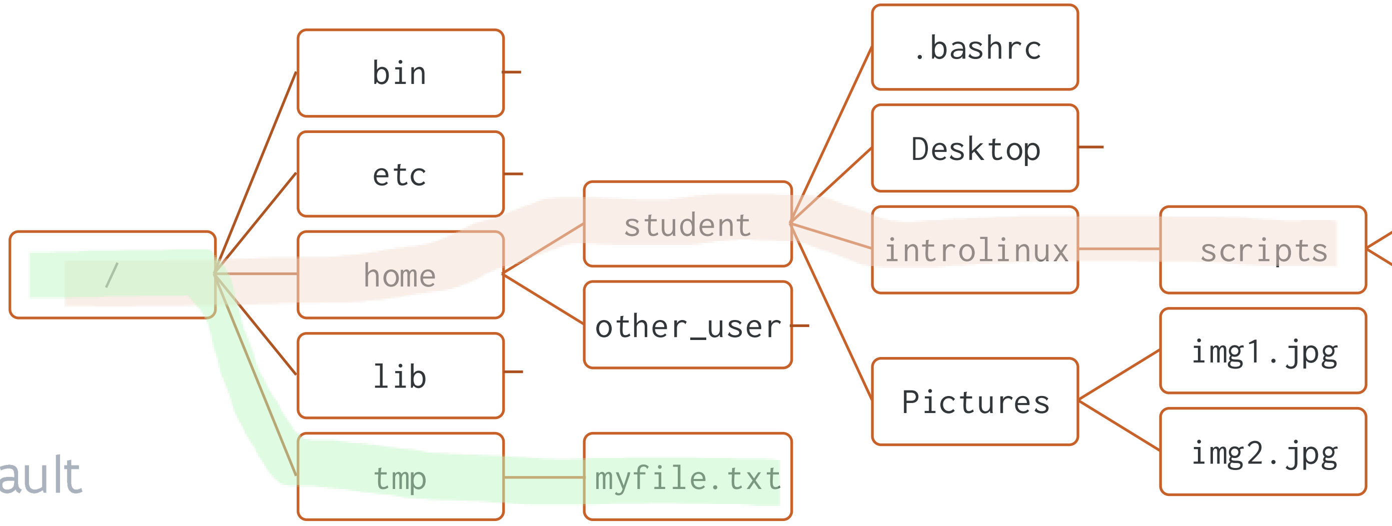
The filesystem – Directories and files

- Tree of **directories** and **files**

- **File name** describes the full location (also called *path*) in the file system
  - /home/student/introlinux/scripts
  - /tmp/myfile.txt
  - / is called the *root* directory

- Directories are separated by /

- The filesystem is **case sensitive**
  - note: macOS is case insensitive by default

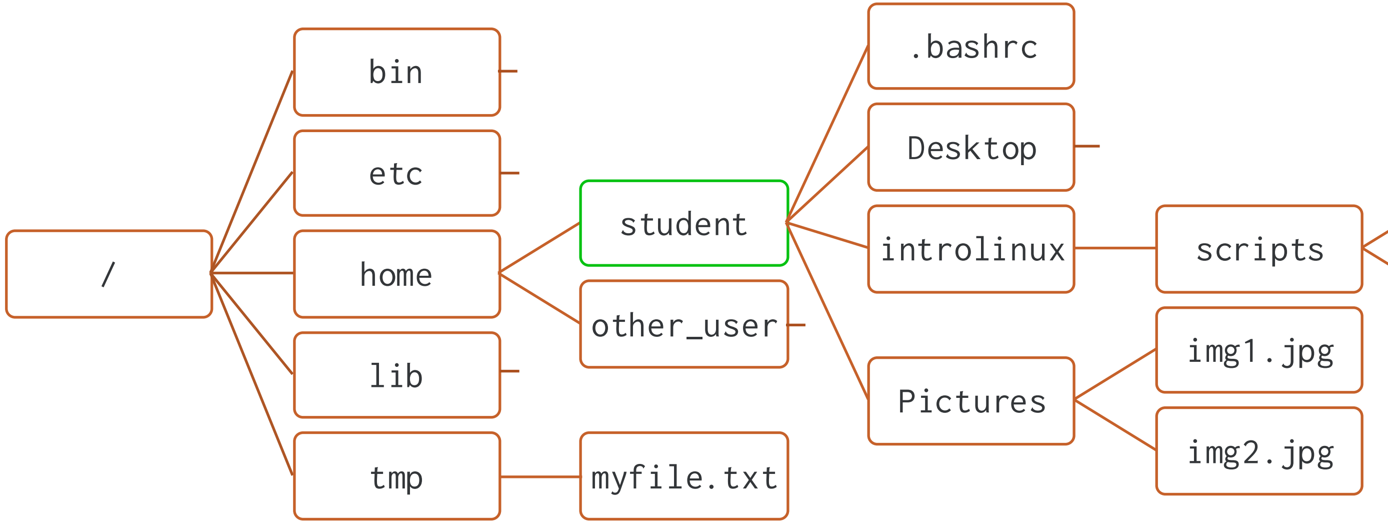


The filesystem – Absolute and relative path

pwd

- **Absolute** file name path starts from root /
- **Relative** file name starts from *current working directory*
- pwd prints the current working directory
  - at login, usually your home directory
- Use `..` to refer to a *parent directory*
- E.g., starting from `/home/student`

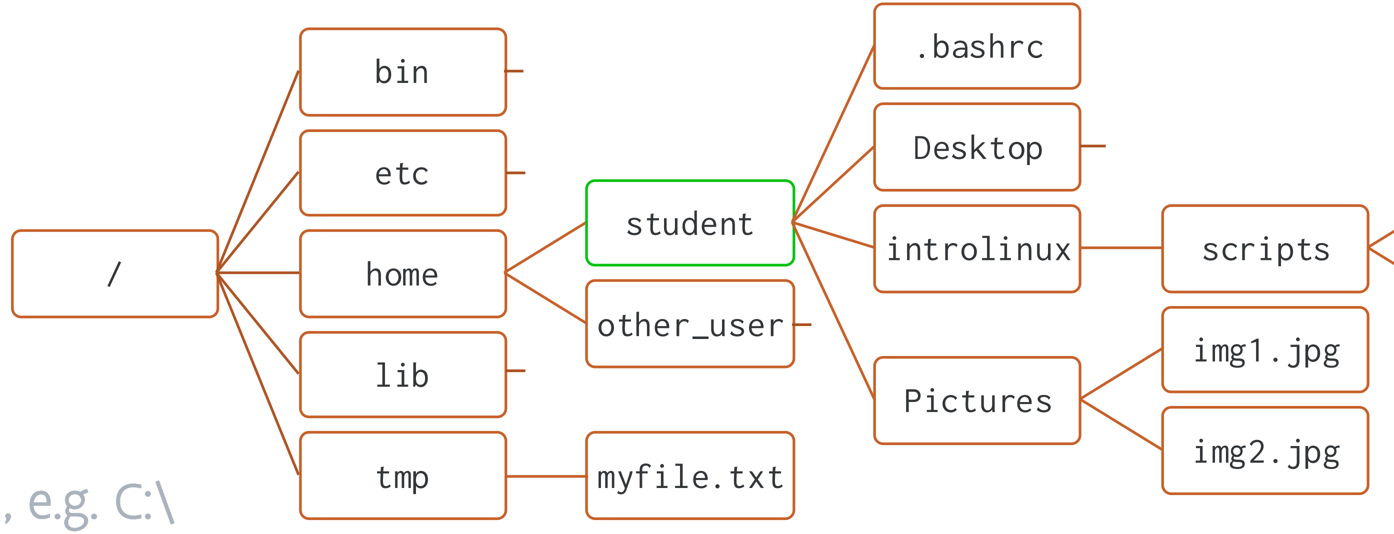
relative path	absolute path
`..`	`/home`
`../other_user`	`/home/other_user`
`../..`	`/`
`introlinux`	`/home/student/introlinux`



The filesystem – Absolute and relative path

pwd

- **Absolute** file name path starts from root /
- **Relative** file name starts from *current working directory*
- pwd prints the current working directory
  - at login, usually your home directory
- Use .. to refer to a *parent directory*
- note: on Windows
  - folders are separated by \
  - the filesystem is case insensitive
  - the root indicates a physical partition, e.g. C:\
  - there can be multiple (root) trees



Navigating the filesystem

cd
ls

- Use **cd** `<directory>` to change the current directory

```
$ cd Downloads
$ cd ../Documents
$ cd -          go back to the previous directory
$ cd            go to your home directory
```

- **ls** (without arguments) lists the current directory's contents

- **~** ("tilde") is a shorthand for the absolute path to your *home directory* (*)

```
$ cd ~ = $ cd /home/<username>
$ cd ~/Downloads = $ cd /home/<username>/Downloads
```

- A single **.** points to the *current directory*

```
$ cd ./Downloads = $ cd Downloads
```

(*) *note: on macOS, when using Belgian keyboard layout (AZERTY), use* `Option + n`

Hands-on

- Try out the following sequence of commands

```
$ cd
$ ls
$ cd Documents
$ pwd
$ cd ..
$ cd ./Documents
$ pwd
```

```
$ cd /bin
$ ls
$ pwd
$ cd ~
$ pwd
$ cd -
$ pwd
```

Manipulating directories and files

mkdir
rmdir

- **Warning: no “recycle bin” or undo!**
  - be *very* careful when moving/copying/removing files at the command-line!

- `mkdir` creates directories

```
$ mkdir dir1 dir2 dir3
```

  - create *nested* directories

```
$ mkdir -p topdir/subdir/subsubdir
```

- `rmdir` removes *empty* directories

```
$ rmdir dir1 dir2 dir3
```

- `touch` creates an empty file, or updates the timestamp of the file if it already exists
  - *note: commands to create real file content will follow later*

mv
cp
rm

Move, copy and remove

- `mv source target` *moves (renames)* files and directories
  - if `target` = existing file → overwrite
  - if `target` = existing directory → move inside it

```
$ mv source1 source2 ... target
```

move list of items into existing `target` directory

- `cp source target` *copies* files and directories
  - same rules as `mv`, except:

```
$ cp srcdir target
cp: -r not specified; omitting directory ‘srcdir’
```

  - **recursively** copy directories and their content:

```
$ cp -r srcdir target
```

- `rm file1 file2 ...` *removes (deletes)* files – *remember: no “recycle bin” or undo!*

```
$ rm -r mydir
```

recursively deletes directories *with* their contents

Using wildcards

- Wildcards help generate lists of filenames, e.g.:

```
$ mv file*.txt target
```

  - Bash replaces `file*.txt` by the list of matching files – called "*wildcard expansion*"

- `*` matches everything → `file*.txt` matches any filename which
  - starts with `file` and ends with `.txt`

- *Remember: no "recycle bin" or undo!*
  - *typing mistake can be dangerous!*

- *Safety first* for `cp`, `mv` and `rm`
  - using `-i` or `--interactive` asks for **confirmation** before overwriting or deleting
  - alternatively, use `echo` in front of the command to see the wildcard expansion

Wildcard patterns

- `*` any sequence of (0 or more) characters

  `file*.txt` → `file.txt file_copy.txt file1.txt ...`

- `?` any single character

  `file?.txt` → `file1.txt file2.txt ... files.txt`

- `[set of characters]` any single character from the given set

  `[fF]ile.txt` → `file.txt File.txt`

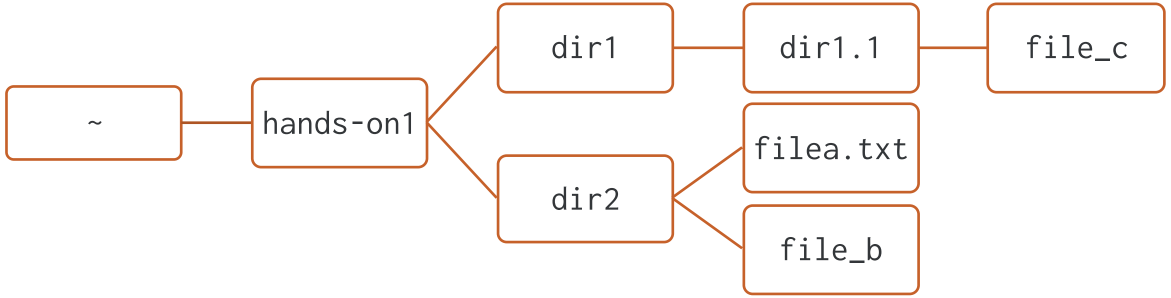
- `[!set of characters]` any single character *not* from the given set

  `file[!123].txt` → `file4.txt file5.txt ... files.txt`

- `[[:class:]]` use a predefined character class

Hands-on

- Create new directories and files in your home directory, according to the given diagram
  - use `touch file.txt` to create empty file
  - check your result with `tree ~/hands-on1`
  - *challenge yourself:* do this exercise from your home ~ without using cd



- Let's move things around
  - copy the files in `dir1.1` to its parent directory
  - rename `dir1` to `dir0`
  - copy `dir2` (including its contents) to `dir2_backup`
  - delete the files in `dir2` using wildcards
  - restore the backup directory

Hands-on

- Which names match the following patterns?

```
[abcdefghijk]*.pdf
backup.[0-9][0-9][123]
[Ff]ile?.*
file_[[:digit:]].txt
```

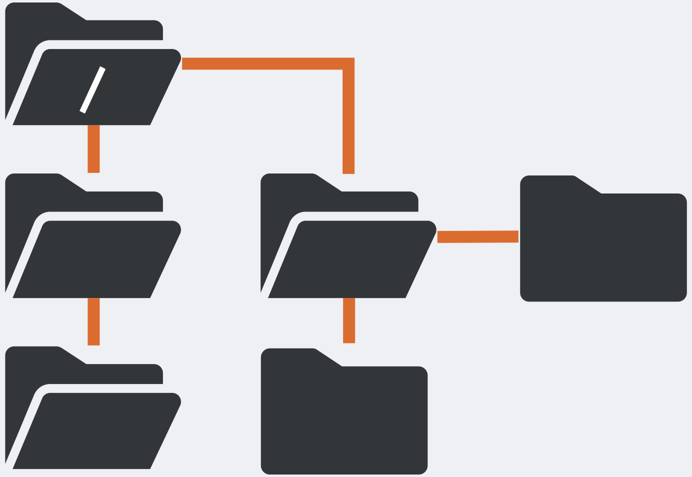
```
file_1.txt
cv.pdf
backup-001
backup.182
introLinux.pdf
A.pdf
File_C.docx
thesis.pdf
Filea.txt
backup.634
```

Reading and editing text files

- **Reading** (displaying) text files
  - `cat` → display the entire content of a text file
  - `less` → allows forward and backward navigation and searching
  - `head -n <x> or tail -n <x>` → print the first/last *x* lines of a file

- **Edit** text files using **text editors** that run inside the terminal
  - `nano` → simple and straightforward text editor
    - user-friendly, with an easy-to-use interface
  - `vi` → stands for visual interface, takes some practice
    - use “modes” for insert or commands
  - `emacs` → highly customizable, extensible text editor
    - powerful editing capabilities with, built-in support for multiple languages, with plugins
    - note: has a steep learning curve (e.g., lots of keybindings)

Summary – The filesystem



- All your **files** are ordered in a **directory tree**
- You refer to your file by its **path**
  `/example/of/an/absolute/path`
- You can navigate through your filesystem
- You can **move**, **remove** & **copy** files,
  but be careful when using:
  - wildcards `*`
  - recursive option `-r`
- You can **read** and **edit** files

Vlaanderen
is supercomputing

Useful tools – Part 1

Hands-on & examples

VLAAMS
SUPERCOMPUTER
CENTRUM

Innovative Computing
for A Smarter Flanders

Hands-on

- Scenario: a colleague sends you a link to a dataset (here: zip-file) and you want to know how many inputs there are in the file `squeue.txt`
  - *note: step by step instructions and commands are given*
    - *it is up to you to look up the correct usage*
  - download the file https://calcua.uantwerpen.be/courses/intro-linux/input.zip – use **`wget`**
  - extract (or unzip) the files – use **`unzip`**
    - *can you list the content of the zip file without unzipping it?*
  - locate the file named `squeue.txt` – use **`tree`** (*) and **`find`**
    - *which tool was better suited?*
  - count the number of lines in the file `squeue.txt`– use **`wc`**

(*) note: the `tree` command may not be installed yet

VLAAMS
SUPERCOMPUTER
CENTRUM

Hands-on

- Scenario: you download some scripts, and you quickly want to know the value of a parameter
  - *note: step by step instructions and commands are given*
    - *it is up to you to look up the correct usage*
  - download the files: https://calcua.uantwerpen.be/courses/intro-linux/pi_montecarlo.tar.gz
  - extract the files – use **`tar`**
    - *pay close attention to the options!*
  - you encounter two scripts with a similar name: `script01_new.py` and `script01_latest.py`
    - show the difference between these two files, but ignore white spaces – use **`diff`**
  - show the line where parameter `n_points` is assigned – use **`grep`**

Download & extract files

wget
unzip
tar

- Download files with **wget**

```
$ wget https://[...].zip
```

- **ZIP** file format

```
$ zip -r file.zip source_dir
$ unzip file.zip
```

- **TAR** / **TAR.GZ**

```
$ tar -czf file.tar.gz source_dir
$ tar -xzf file.tar.gz
```

  - TAR stands for Tape Archive – also called “*tarball*”
  - more common in Unix/Linux environments
  - preserves file permissions, ownership, and timestamps, making it more suitable for backups and archives

What I type:

```
$ tar czf data.tar.gz data
$ tar xzf data.tar.gz
```

What I say in my head:

"create ze file"

"extrakt ze file"

diff – Comparing files and directories

- Show differences between text files

`$ diff -i file1 file2`	ignore case
`$ diff -w file1 file2`	ignore all white space
`$ diff -y file1 file2`	output in two columns
`$ diff -r dir1 dir2`	recursively compare directories

Hands-on

- Scenario: you see that a colleague opens file `scores.csv` with *comma separated values* in Excel to sort the data by `Score`
  - *not on your watch – you use Miller like a pro!*
- Start by reading Miller in 10 minutes
- Install the Miller command **mlr**
  - on Linux and Windows WSL (Ubuntu) : `sudo apt install miller`
  - on macOS (with Homebrew) : `brew install miller`
- Try to “pretty-print” the `scores.csv` file
  - also, try to sort it by the values of the field `Score`

Processing text-formatted structured data

mlr

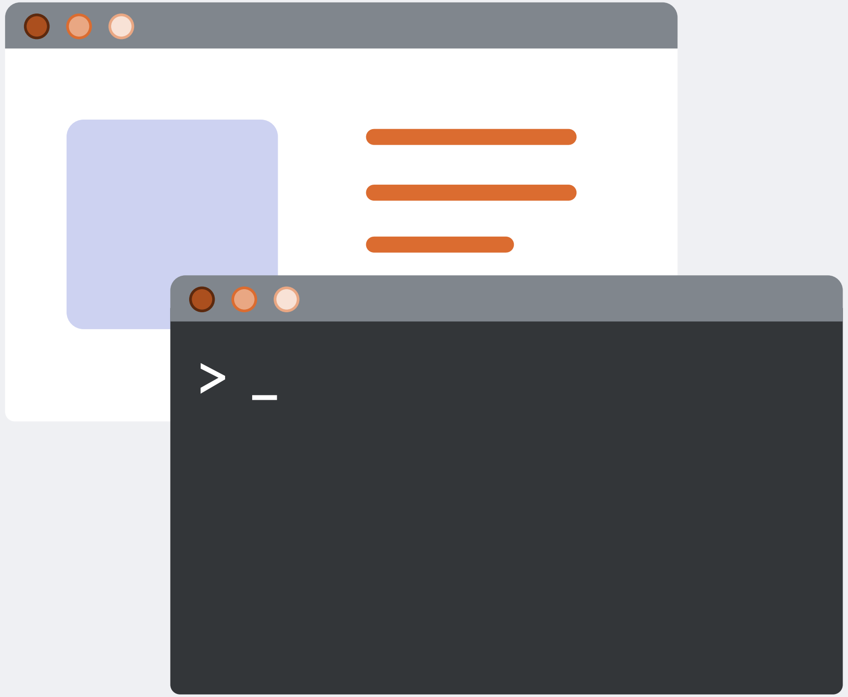
- Why our sysadmin 😍**Miller** *(obligatory slide!)*
  - easily query, shape and/or reformat CSV, TSV, JSON, … data files
  - pretty-print data files, convert between file formats
  - using compact verbs instead of a programming language

- Some examples

`$ mlr --icsv --ojson cat scores.csv`	convert scores.csv to JSON format
`$ mlr --c2j cat scores.csv`	using a keystroke-saver flag
`$ mlr --csv tail -n 4 scores.csv`	print header and last 4 lines
`$ mlr --c2p cut -f Name,Gender scores.csv`	pretty-print only fields Name and Gender

`$ mlr --c2p filter '$Course=="History"'`
`then cut -f Name,Score`
`then sort -r Score scores.csv`
	for the course History
	show Name and Score
	sorted in descending order

Summary – Useful tools



- Some things you used to do with a **graphical interface**
  - downloading files,
  - decompressing `.zip` or `.tar.gz` archives,
  - opening `.csv` in Excel, …
- However, it is easy to do with the command line & it can save you time *(automatization – see scripting)*
- You can discover and **install** new *shiny* tools

Vlaanderen
is supercomputing

Streams & pipelines

Input and output streams & redirection

Command pipelines

Input and output streams

- Output and input (**I/O**) of commands is managed using *streams* and *file descriptors*
  - **streams** provide an interface with powerful formatted input and output functions (high-level)
  - under the hood, streams use **file descriptors** (fd) to keep track of the I/O-resources (low-level)

stream	readable name	purpose	fd
stdout	standard output	for normal output	1
stderr	standard error	for printing warnings and errors	2
stdin	standard input	from which commands receive input	3

  - by default, “stdin” is read from the keyboard, while “stdout” and “stderr” are sent to the terminal

- We can **redirect** the output and input streams, to
  - write output to a file
  - send output from one command to input of another
  - read stdin from a file

Output redirection

- To redirect an output stream, use operator **`i>`** with its associated file descriptor (fd) **`i`**

- **Redirect standard output** (stdout)

```
$ ls > ls-output.txt = $ ls 1> ls-output.txt
```

  - the file `ls-output.txt` is created and contains the command's output
  - note: stderr is still shown in terminal

- **Redirect standard error** (stderr)

```
$ ls wrong-filename 2> ls-error.txt
```

- Redirect both stdout and stderr

```
$ ls *.txt *.jpg > ls-output.txt 2> ls-errors.txt
$ ls *.txt *.jpg > ls-output-and-errors.txt 2>&1
= $ ls *.txt *.jpg &> ls-output-and-errors.txt
```

to *different* files

to the *same* file

to the *same* file

Output redirection

>>

- **Hiding** a program's output

```
$ ls > /dev/null
```

  - `/dev/null` is a special "file" that discards everything written to it

- Warning: redirecting (>) creates a **new file**
  - if a file exists with the same name, it will be overwritten!
  - if the command produces no output, the file will be empty

- **Append** (>>) stdout and/or stderr to the end of a file *without erasing previous content*

```
$ date >> diary.txt
$ echo "Dear diary, today ..." >> diary.txt
$ ls notfound 2>> ls-errors.txt
$ ls *.txt *.jpg >> ls-output-and-errors.txt 2>&1
= $ ls *.txt *.jpg &>> ls-output-and-errors.txt
```

Input redirection

- Standard input (stdin) is by default read from the keyboard

- The **input redirection** operator `< filename` opens a file, and the program processes it as input
  - example, using the command-line calculator **bc**

    ```
    $ echo "2 * 17" > homework.txt
    $ bc < homework.txt
    34
    ```

  - useful for automating commands that normally require user input
  - or for reading from specific sources (devices) directly

- Redirecting both standard input and standard output

    ```
    $ bc < homework.txt > answers.txt
    ```

Command pipelines

- **Combine several commands** by chaining them using the “pipe” operator **|** (*)

```
$ command1 | command2 | command3 [| ...]
```

  - a *pipeline* creates a flow of data between commands
  - stdout from `command1` is directly sent to stdin of `command2` (etc)
  - the commands run in parallel, each command processes input as it becomes available

- Example: scrolling through the list of all processes with **`ps`** and `less`

```
$ ps aux | less
```

- **Create complex commands from simple building blocks**

```
$ who | cut -d' ' -f1 | sort | uniq > users
```

- note: to pipe stderr from a command, redirect it to stdout

```
$ command1 2>&1 | command2
```

(*) *note: on macOS, when using Belgian keyboard layout (AZERTY), use* `Shift + Option + l`

Hands-on

- Given the file `chemistry.txt`, how many courses are taught by Wouter Herrebout in the first semester?
  - *note: use pipelines whenever possible!*
  - investigate the file – use **cat**
  - print only the lines belonging to the first semester – use **grep**
  - of those lines, select the lines containing Wouter Herrebout – use **grep**
  - count the resulting number of lines – use **wc**
  - *challenge yourself: use* `mlr` *instead*

Hands-on

- Which are, in alphabetical order, the last 5 course codes starting with **1001WET**? Write them to a new file.
  - get the lines where the course code starts with *1001WET*
  - sort the lines in alphabetical order (by *course code*) – use `sort`
    - *is `sort` alphabetically or numerically by default?*
    - *how can you ignore cases?*
  - Of those lines, select the last 5 – use a pipe and `tail`
  - write the output to a new file
  - edit your pipeline to instead sort alphabetically by *course name*
    - *how do you specify the 'tab' delimiter? – search the web*

Hands-on

- Which course is listed more than once in the file chemistry.txt?
  - print each unique line of the file, with the number of times it occurred – use `uniq`
    - *carefully read the last line of DESCRIPTION in the man page*
  - print only the course(s) which are listed more than once, together with the number of times

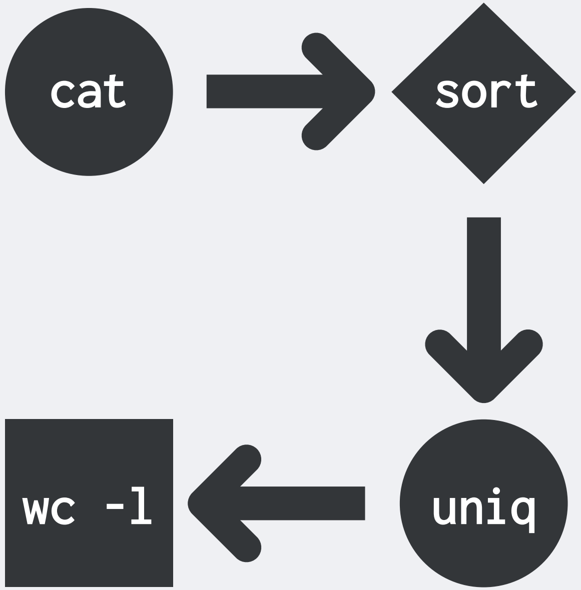
Overview of frequently used commands

- Typical commands for pipelines

`cat`	concatenate files (useful to print out file content)
`grep`	filter lines which match a given search pattern
`head` / `tail`	print first/last lines of input
`sort`	sort input alphabetically
`uniq`	report or leave out repeated lines
`wc`	print the number of lines, words and bytes of input
`sed`	transform input (pattern replacement and more)

- Find more commands in the GNU core utilities manual

Summary – Streams & pipelines



- You can **redirect** the output (> or >>) and input (<) flow of a command
  - *stdin*
  - *stdout*
  - *stderr*

- You can **pipe** (|) the output of one command to the input of another to build a **pipeline**

```
$ command1 | command2 | command3 ...
```

- When you get comfortable with frequently used commands, pipelines feel natural

Part 2 – Sneak preview – Shell scripts

bash

- **shell script** = text file containing a series of commands
- Example script "`myscript.sh`"

```
my_analysis input.data > my_results/science.txt
tar -cvzf my_results.tar.gz my_results
rm input.data
```

- Run (execute) the script

```
$ bash myscript.sh
```

- note:
  - commands are separated by newlines or by semicolons '**;**' (as in the terminal)
  - commands are executed one after the other, just as if you entered them manually

Vlaanderen
is supercomputing

Introduction to Linux

Ine Arts, Franky Backeljauw, Michele Pugno, Robin Verschoren

Version Spring 2026 – Part 2

Overview

Part 2 – diving deeper

- The environment
  - environment variables
  - aliases & environment startup
- The shell
  - variable and arithmetic expansions
  - command substitution
  - escaping special characters
- Useful tools
  - regular expressions
  - search and replace
- Bash scripting basics
  - writing and running shell scripts
  - using variables & command-line arguments
  - the `for` loop
- The filesystem
  - permissions & ownership
- Running programs
  - processes and threads
  - managing processes
- More bash scripting

The environment

Environment variables

Aliases & environment startup

Environment variables

export

- We can use variables in the shell

  `$ myvar=some_value` set the value for variable myvar

  `$ echo $myvar` get the current value of myvar – this is called “*variable expansion*”

  `$ set` display *all* (shell) variables (and functions)

  - no spaces around ‘=’
  - no spaces in `some_value` unless using quotes
  - these are “plain” variables – they only exist in the running shell itself

- **Environment variables** are special

  `$ export myvar` make myvar an environment variable

  `$ printenv` display (exported) *environment* variables

  - they are passed on to processes started from the shell
  - they can influence the behaviour of programs (e.g. `OMP_NUM_THREADS`, `PS1`)

Environment variables

- Some **standard environment variables**

PATH	a colon-separated list of directories that are searched when you enter the name of an executable program
HOME	the path name of your home directory (~)
USER	your user name
SHELL	the name of your shell program
PWD	the current working directory
TMPDIR	directory for temporary files (usually /tmp)

- Example: access an environment variable from within a Python script

```
$ python3 -c 'import os;
  print("hi there,", os.getenv("USER"), "!")'
```

Aliases

alias

- Substitute a string for a simple command
- `$ alias <name>=<value>` means that `<name>` will be replaced by `<value>`
- Handy to set default options and simplify your commands

`$ alias ls="ls -F --color=auto"`	append filetype indicator, colorize output
`$ alias lart="ls -Falrt --color=auto"`	show hidden files, recently modified first

- Removing (deleting) aliases *– in the current shell only*

`$ unalias <name>`	removes the alias for `<name>`
`$ unalias -a`	removes all aliases

Environment startup

- User-defined aliases, variables and functions are reset when restarting the shell

- Store the settings in a **startup file** so they are persistent for your environment
  - applied every time you start an interactive shell

    `~/.bashrc` *you can define your own aliases and functions here*

  - other files are applied for a login shell and when exiting a shell
    - see Bash Startup Files for more information

- note: on macOS, the default shell nowadays is **`zsh`** (Z shell)
  - for zsh, the startup file used for an interactive shell is named **`~/.zshrc`**
  - see Z Shell Startup Files for more information

Summary – The environment

~/.bashrc

```
export PATH="$HOME/bin:$PATH"
export EDITOR=nano

alias ll='ls -alF --color=auto'

echo "Welcome, $USER! Today is $(date)."
echo "Your preferred editor is $EDITOR."
```

- **Environment variables** are passed on to processes
- $PATH is a list of directories your shell searches for commands
- You can create an **alias** for long/complex commands you use often
- Add aliases, env. variables, functions, … to your **startup file** (~/.bashrc) to have them available every shell session

The shell – Part 2

Variable and arithmetic expansions

Command substitution

Shell expansions

- When you type a command-line and press **`Enter`**
  - the shell performs several processes on the text before it carries out your command
  - the process that makes this happen is called *expansion*

Variable expansion

- `$variable` → variable's current value

`$ echo $USER`	print the current value
`$ set`	display all variables
`$ echo $SUER`	what if variable doesn't exist?
`$ echo ${USER}_home`	use **{}** to disambiguate the variable name
`$ echo $USER_home`	doesn't work without {}
`$ myvar='Hello, world!'`	set a variable
`$ echo $myvar`	

Shell expansions

Arithmetic expansion

- `$((expression))` → result of expression

```
$ echo $((10 + 5 + 3))
```

  - arithmetic expression – *integers only!*
  - operators: `+`, `-`, `*`, `/` , `%` (remainder), `**` (exponentiation)
  - single parentheses may be used to group multiple subexpressions:

```
$ echo $(( (5**2) * (3*4) ))
```

Command substitution

- `$(command)` → output of command

```
$ echo We are now $(date)
$ echo I see $(ls -A | wc -l) files and subdirs
```

Shell expansions

Escaping special characters & using quotes

```
$ echo The total is $100.00 # ?!
```

- Use the **escape character** `\` for literal use of **special characters** (**$**, \, **`**, **{**, **}**, **(**, **)**, *****, **␣**)

```
$ echo The total is \$100.00
```

- Inside single quotes `''` special characters lose their meaning → no expansion at all

```
$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
```

- Inside double quotes `""` special characters lose their meaning *except* **$**, \, **`**

```
$ echo "$USER $((2+2)) $(cal)"
$ echo "The total is \$100.00"
```

Shell expansions

Other

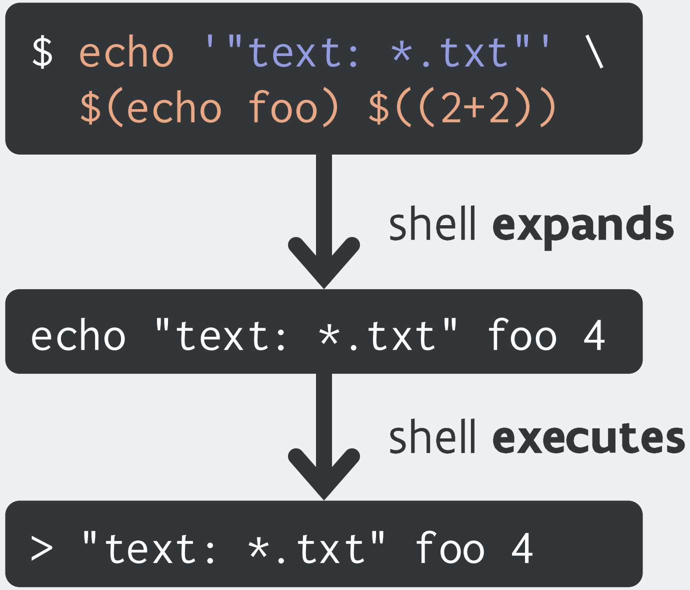
- Word splitting: words separated by space become separate arguments

```
$ touch "two words.txt"
$ ls -l two words.txt
$ ls -l "two words.txt"
$ ls -l two\ words.txt
$ ls -l two↹
```

- Quote removal: after all expansions, quotes are removed unless you escape or quote the quotes

```
$ echo "hello world"
$ echo \"hello\" '"world"'
```

Summary – The shell



```
$ echo '"text: *.txt"' \
  $(echo foo) $((2+2))
```

```
echo "text: *.txt" foo 4
```

```
> "text: *.txt" foo 4
```

- The shell interprets your command before running it, by **expanding**:
  - `$VARIABLES`
  - `$((arithmetic))`
  - `$(command output)`
- Ignore the meaning of special characters by using single quotes (`'`) or **escape** (`\`)

Vlaanderen
is supercomputing

Useful tools – Part 2

Regular expressions

Search and replace

Regular expressions

- Also called “*regex*”
  - symbolic notation used to **match text patterns**
  - similar to wildcards (`*`, [], ?), but more powerful

- Many programs and programming languages support regular expressions
  - `grep`, `sed`, …
  - Text editors, e.g. `emacs`
  - Python, Perl, Matlab…

- *Note: slight differences exist in notation and supported patterns*

Regular expressions

grep

- Example: counting animals in the Bible

```
$ grep -Eo ' (dragon|serpent|lion|eagle)s? ' bible.txt | sort | uniq -c
```

```
     10      dragon
      4      dragons
     10      eagle
      3      eagles
     43      lion
     13      lions
     14      serpent
      4      serpents
```

Regular expressions

grep -E
egrep

- Literal characters and digits

```
$ grep lion bible.txt
```

- "*Metacharacters*" are used for repetitions, grouping, alternatives, …
  - Two notations for metacharacters
    - **basic regular expressions** (BRE)

      `^ $ . [ ] * \( \) \{ \} \? \+ \|`

    - **extended regular expressions** (ERE)

      `^ $ . [ ] * ( ) { } ? + |`

- *Note: on the slides, we use ERE for readability*
  - using `$ grep -E` = `$ egrep`

VLAAMS
SUPERCOMPUTER
CENTRUM

Regular expressions – Metacharacters

- `.` Match any single character

```
$ grep '.word' words.txt
```

  - *Note: remark the difference with using wildcards*

```
$ touch .zip 1.zip 1zip 22.zip 2zip
$ ls *zip
$ ls *.zip
$ ls | grep .zip
```

- `^ $` Called "*anchors*", matches the beginning (`^`) or end (`$`) of a line

```
$ grep '^word' words.txt
$ grep 'word$' words.txt
$ grep '^word$' words.txt
```

Regular expressions – Character classes

- `[]` Character class

`[lw]ord`	matches `lord` and `word`
`[l-w]ord`	matches `lord`, `mord`, `nord`, ..., `word`
`[^lw]ord`	matches any `ord` not preceded by `l` or `w`
`[^l-w]ord`	matches any `ord` not preceded by `l`, ..., `w`
`^[A-Z]`	matches any uppercase letter at the beginning of a line
`^[-AZ]`	matches only the character `-`, `A` or `Z` at the beginning of a line

Regular expressions – Repetitions

- `?` Match preceding element zero or one time
- `*` Match preceding element zero or more times
- `+` Match preceding element one or more times
- `{}` Match preceding element a specific number of times

`{`*n*`}`	exactly *n* times
`{`*n*`,`*m*`}`	at least *n* times and/or at most *m* times – for or, drop *n* or *m*

- Some examples:

`A*`	matches <empty string>, A, AA, …
`.*`	matches any sequence of characters
`\$[1-9][0-9]{2,}`	match any amount of `$100` or more

Regular expressions – Sub-expressions, alternatives

- `()` sub-expression

`(tick)+`	matches 1 or more repetitions of (the word) `tick`

- `|` alternatives

`word|lord`	matches `word` and `lord`
`(w|l)ord`	matches `word` and `lord`, using grouping
`(w|l|sw)ord`	matches `word`, `lord` and `sword`

- `\`*n* reference to the *n*-th subexpression (used in find and replace)

Regular expressions – Cheat sheet

<table>
<tr><th></th><th></th><th></th></tr>
<tr><td>.</td><td>Match any</td><td>(BRE)</td></tr>
<tr><td>^ $</td><td>anchor beginning or end of line</td><td></td></tr>
<tr><td>[ ]</td><td>character classes</td><td></td></tr>
<tr><td>?</td><td>0 or 1 times</td><td>\?</td></tr>
<tr><td>*</td><td>0 or more times</td><td></td></tr>
<tr><td>+</td><td>1 or more times</td><td>\+</td></tr>
<tr><td>{<i>n</i>}</td><td><i>n</i> times</td><td>\{<i>n</i>\}</td></tr>
<tr><td>{<i>n</i>, <i>m</i>}</td><td>more than <i>n</i>, less than <i>m</i> times</td><td>\{<i>n</i>, <i>m</i>\}</td></tr>
<tr><td>( )</td><td>subexpression</td><td>\(\)</td></tr>
<tr><td>|</td><td>alternative</td><td>\|</td></tr>
<tr><td>\<i>n</i></td><td>reference to the <i>n</i>-th subexpression</td><td></td></tr>
</table>

- For a comprehensive overview, check this RegEX cheat sheet

Hands-on

- Use `grep -E` on the file `words.txt`
  - which words start with `chemi`?
  - which words contain both `are` and `be`? (answer using 1 regular expression)
  - which words start with a capital letter and contains two consecutive letters `a`?
  - how many five letter words do you find? (use a pipeline)

sed – Search and replace

- sed = **st**ream **ed**itor
  - works on standard input or a set of input files
  - perform text manipulations using regular expressions – *non-interactively*, using ‘commands’
  - powerful, but somewhat complex

- Typical usage: **search and replace**

```
$ sed 's/regexp/replacement/'
```

  - processes input line by line, prints the modified text to standard output
  - by default, replaces only the first occurrence on each line
  - by default, matching is done case sensitive
  - by default, uses BRE

- *For larger tasks, you might choose awk, Perl, Python, …*

sed – Search and replace

- **sed** [options] <script> <file>

-n	suppress automatic printing
-i	edit file in-place (instead of printing to standard output)
-E	use extended regex (ERE)

  - <script> = [line selection]<command>

n[,*m*]	line number *n* (until *m*)
$	refers to the last line
/regex/	lines that match regex

    - <command> performs an action on the (matched) text

s/regex/repl/	replace matches for regex by repl
d	delete the matched line(s)
*<command>*I	use case insensitive matching
*<command>*g	act on all matches on this line (global replacement)

sed – Search and replace

- Some examples:

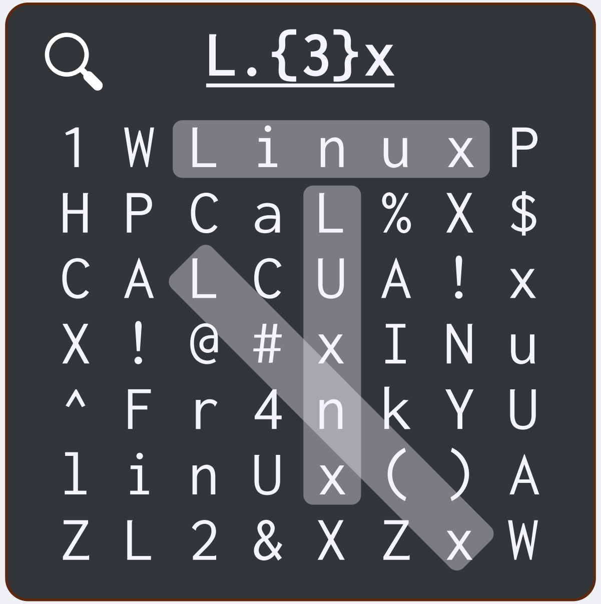
`$ sed -n '3,5p' distros.txt`	print only lines 3 to 5
`$ sed -i '1d' distros.txt`	delete the first line in the file
`$ sed '/Fedora/d' distros.txt`	equivalent of `grep -v Fedora`
`$ sed '/Fedora/a from Red Hat' distros.txt`	append the given text on a new line after the matched pattern
`$ sed 's/Fedora/& from Red Hat/' distros.txt`	use **&** to reference the whole matched pattern in the replacement string
`$ sed 's+/+-+g' distros.txt`	use another *delimiter* (**+** instead of **/**)

- Check the sed manual or this sed cheat sheet

Hands-on

- Find and replace all instances "`chemie`" by "`scheikunde`" in the file `chemistry.txt` and write the output to a new file.
  - make sure the replacement is case insensitive
  - do the replacement directly in the file
- In `distros/distrostab.txt`, rewrite `MM/DD/YYYY` as `YYYY-MM-DD`.
  - match the pattern `MM/DD/YYYY` using 3 subexpressions
  - construct the replacement by referring to the subexpressions

Summary – Useful tools



```
L.{3}x

1 W L i n u x P
H P C a L % X $
C A L C U A ! x
X ! @ # x I N u
^ F r 4 n k Y U
l i n U x ( ) A
Z L 2 & X Z x W
```

- **Regular expressions** are a powerful tool to match patterns of text
  - You can use **metacharacters** for:
    - wildcards,
    - repetitions,
    - groups, …
  - There *can* be subtle differences in syntax with different programming languages

- You can **search & replace** text

```
$ sed 's/old/new/g' file.txt
```

Bash scripting basics

Writing and running shell scripts

Shell scripts – Writing shell scripts

bash

- **shell script** = text file containing a series of commands
- Example script “`myscript.sh`”

```
my_analysis input.data > my_results/science.txt
tar -cvzf my_results.tar.gz my_results
rm input.data
```

```
$ bash myscripts.sh
```

run (execute) the script

- Note that
  - commands are separated by newlines or by semicolons ‘`;`’ – just as in the terminal
  - commands are executed one after the other – just as if you entered them manually
- Example scripts in https://calcua.uantwerpen.be/courses/intro-linux/scripts.zip

Shell scripts – Note about line endings

```
file
dos2unix
```

- Note about **line endings**
  - line endings are encoded differently under Windows and Unix/Linux
    - Windows style: carriage return + line feed (CRLF, `\r\n`)
    - Unix/Linux style: newline (`\n`)
  - this can introduce problems when running bash scripts

- Check which encoding is used:

```
$ file filename
```

- If needed, convert your “Windows style” file into a “Unix/Linux” style:

```
$ dos2unix -n inputfile outputfile
```

- *Note: any suitable text editor can do this as well*

Shell scripts – Running shell scripts

```
$ cat scripts/script01.sh
```

“shebang”

```
#! /bin/bash

# This is our first script.

echo 'Hello World!' # comment
```

`$ bash script01.sh`	call the interpreter (bash) ourselves
`$ chmod +x script01.sh`	
`$ script01.sh`	doesn’t work because work dir is not in PATH!
`$ ./script01.sh`	**the interpreter from the “shebang” is used**

Shell scripts – Running shell scripts

- #! is called "**shebang**" – it tells the system which **interpreter** should execute the script
  - for a bash script:

    `#!/bin/bash`
  - spaces (between parts) are optional:

    `#!/bin/bash` = `#! /bin/bash` = `#!              /bin/bash`

- Any scripting language interpreter can be used (not just bash)
  - example for Python:

    `#!/usr/bin/python3` uses that specific Python executable
  - or preferably:

    `#!/usr/bin/env python3` uses the first `python3` found in PATH

Shell scripts – Using variables

```
#!/bin/bash                                                     # script02.sh

currenttime=$(date +"%x %r %Z")
myname=$USER

echo "id: $myname, current time: $currenttime"
```

- Remember:
  - Setting a variable: without **$**, no spaces around **=** e.g., `myname=`**`some_value`**
  - Using a variable (variable expansion): with **$** e.g., `echo `**`$`**`myname`

- User variables can not start with a digit: $**1**, $**2**, ...
  - these are special variables, a.k.a. “command-line arguments”

Shell scripts – Command-line arguments

```
#!/bin/bash                                                        # script07.sh

echo "Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
...
\$9 = $9"
```

```
$ ./script07.sh these are four arguments
$ ./script07.sh "this is a single argument"
```

- Using command-line arguments in your script:
  - more than 9 args? → `${10}`, `${11}`, … or use `shift` (see next slide)
  - last arg → `${!#}` or `${@: -1}` (space is required) or `$BASH_ARGV` (bash only)
  - list of all command-line arguments: `$@`

Bash scripting – The for loop

`for variable in list; do commands; done`

```
#!/bin/bash                                                      # script09.sh
for i in A B C D; do
  echo $i
done
```

- `list` can be any bash expression resulting in a list, e.g.

  `for file in *.txt; do ... done`   loop over each `txt` file

- if “`in list`” is omitted, `for` loops over the command-line arguments

Bash scripting – The for loop

```
#!/bin/bash                                                     # script09b.sh


for i in $(seq 1 10); do
  echo $i
done


for i in $(seq 11 0.75 20); do
  echo $i
done


for i in {21..30}; do
  echo $i
done
```

Hands-on

- Write a script that adds up all command-line arguments
  - loop over all command-line arguments
  - add each argument to the total – use arithmetic expansion `$(( ))`
  - test your script with different inputs – make sure your script is executable

- What do you expect to happen when instead of integers you input:
    - text?
    - decimals?
  - *test your expectations!*

Hands-on

- Write a script that loops over each command-line argument and that
  - creates a directory `dir_<argument>` in the current location
  - copies a template file `input.txt` to `input_<argument>.txt` into this directory
  - replaces every occurrence of "`<param>`" in this file by the value of the argument
- *Challenge yourself!*
  - we want the name of the template file as the first command line argument
  - run previous script without changes, with this new argument – what happens?
  - try to fix what went wrong – look into the `shift` command

Hands-on

- Here is an example of a script with some more logic structures

  if

  while

  ~~case~~

  break / continue

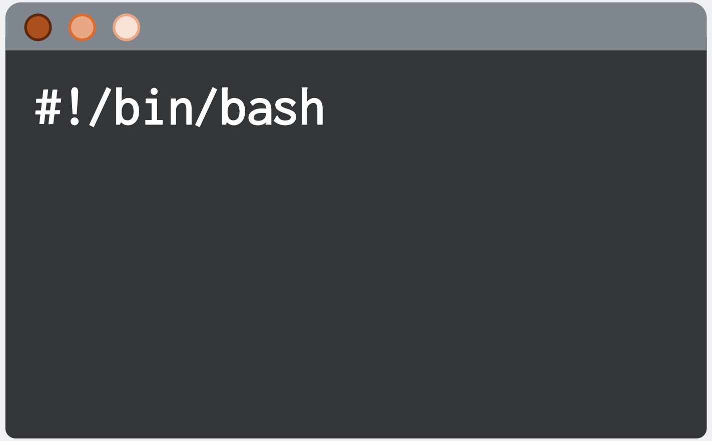
  ~~functions~~

  ...

  - *Try to figure out what it does*

```
#!/bin/bash                         # script12.sh
while echo -n "enter number: "; read NUM
do
  if [ $NUM -eq $NUM ] 2>/dev/null; then
    :
  else
    echo " $NUM is not a number"
    continue
  fi
  if [[ $(( $NUM % 2 )) -eq 0 ]]; then
    echo " $NUM is an even number"
    continue
  fi
  echo " $NUM is an odd number"
  break
done
```

Summary – Bash scripting basics



```
#!/bin/bash
```

- You can combine commands into a reusable **script**
- To run your script, you need to:
  - add a **shebang**
  - grant execute permission
- Up to now, you learned how to use:
  - variables,
  - arguments (`$1`)
  - for loops (`for ...; do ...; done`)

Vlaanderen
is supercomputing

The filesystem – Part 2

Permissions & ownership

VLAAMS
SUPERCOMPUTER
CENTRUM

Innovative Computing
for A Smarter Flanders

vscentrum.be

Permissions & ownership

`id`

- Every user has a unique **id** and (user)**name** and belongs to one or more **groups**

- To see your id, name and groups, run `id`

uid	your user id
gid	primary group id
groups	list of all groups you are a member of

- Every file or directory belongs to a user *and* a group with different **access permissions** for
  - **u**ser
  - **g**roup
  - **o**thers = all other users who are not a member of the file's group

Permissions & ownership

- Use `ls -l` to see **permissions** and **ownership**:

```
$ ls -l scripts
total 512
-rwxr-xr-x 1 vsc2xxxx antwerpenall   76 Feb  8 12:43 script01.sh
...
permissions     user     group     size modif.time   filename
```

- `-rwxrwxrwx` = three kinds of permissions for “user,” “group” and “others”

permission	file access	directory access
read	read file’s contents	list directory contents
write	modify file’s contents	create, remove & rename files *(also needs x)*
e**x**ecute	run file as a program	enter directory & access contents

Setting permissions

chmod

- chmod can change the permissions for files or directories

- Add/remove permissions using chmod + or chmod -

```
$ chmod +w file.txt          add write permission for all users
$ chmod g-w file.txt         remove write permission for the group the file belongs to
$ chmod ug+x,o-r file.txt
```

- Or using numbers instead, where 0=none, 1=x, 2=w, 3=wx, 4=r, 5=rx, 6=rw, 7=rwx

```
$ chmod 640 file.txt
```

- -R = Recursive, change permissions on a directory and all its contents

```
$ chmod -R go-xr my_private_dir
```

Changing ownership

chown

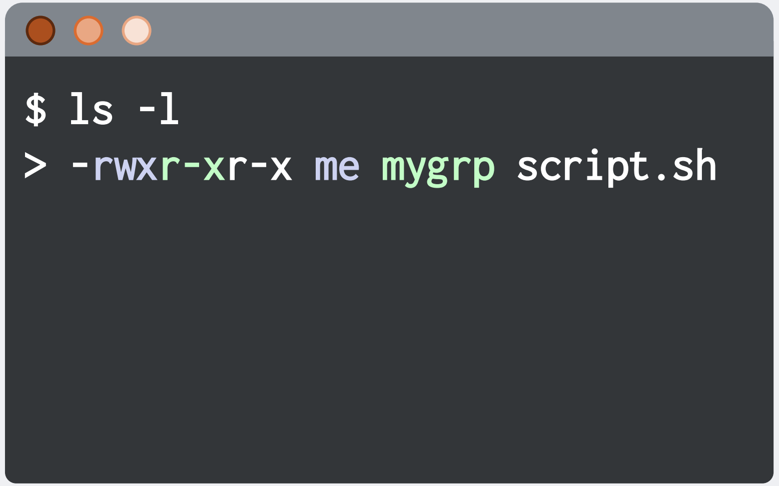
- chown can change the owner and/or group of files and directories

```
$ chown owner file.txt
$ chown owner:group file.txt
$ chown :group file.txt
```

  - -R = Recursive

```
$ chown -R owner:group my_dir
```

Summary – The filesystem



```
$ ls -l
> -rwxr-xr-x me mygrp script.sh
```

- Every file/directory has **permissions**:
  - read (**r**)
  - write (**w**)
  - execute (**x**)
- These permissions are separate for
  - owner (**u**)
  - group (**g**)
  - others (**o**)
- As file owner, you can modify the permissions and **ownership**

Running programs

Processes and threads

Managing processes

Processes and threads

- A **process** = running instance of a program
  - has a unique identifier or **PID** (**P**rocess **ID**)
  - can start other processes, its **child** processes
  - consists of one or more **threads**

- Threads **share** access to the process' memory
  - but processes cannot access other processes' memory!

- *Parallelization* on **multiple CPU cores**
  - multiple processes -> "*distributed memory parallelism*"
  - multiple threads in one process -> "*shared memory*"

Processes and threads – Looking at processes

ps
top
htop

- **ps** prints information on running processes.

`$ ps`	show processes *in the current shell*

```
  PID TTY          TIME CMD
 8627 pts/12   00:00:00 bash
19621 pts/12   00:00:00 ps
```

`$ ps x`	show **all** processes of current user
`$ ps ax`	show all processes of **all users**
`$ ps u`	show username, CPU and memory usage (can be combined with previous, e.g. `$ ps axu`)
`$ ps -u <user>`	show processes of the given user

- **top** or **htop** show processes together with CPU and memory usage *in real time*

Processes and threads – Managing processes

bg
fg

Foreground processes

- Example: run `xclock` with `$ xclock -update 1`
  - once the process is started, it is executing and occupying the terminal
  - you no longer have access to the prompt; it prevents you to run other commands
- To **terminate** the foreground process, press `Ctrl + c`
  - `xclock` disappears, the prompt returns
- To **stop** (pause) the foreground process, press `Ctrl + z`
  - the process is stopped, the prompt returns
  - the process can be restarted again using

`$ fg`	process resumes “in the foreground”
`$ bg`	process continues ”in the background”

Processes and threads – Managing processes

&
jobs

Background processes

- To start a process in the background, terminate the command by **&**

```
$ xclock -update 1 &
```

bash prints the job number and PID, e.g. `[1] 9582`

- When having multiple background processes, use `$ jobs` to see a list

```
$ xclock -update 1 &
[1] 9582
$ xclock -update 1 &
[2] 9588
$ jobs
[1]- Running xclock -update 1 &
[2]+ Running xclock -update 1 &
```

- Use the jobs' number to control the different processes, e.g.

```
$ fg %2
```

run job 2 in the foreground

Processes and threads – Managing processes

kill

Terminating processes

- *Reminder*: `Ctrl + c` terminates the **foreground** process

- Use the command **`kill`** `<PID>` to terminate any process (owned by you)

`$ kill 12345`	terminate process with PID 12345 *note: the process may belong to another shell*

- `kill %<jobnum>` terminates a **background** process

`$ kill %2`	terminate job 2, with time for cleanup
`$ kill -KILL %2`	terminate job 2 *immediately*

- Use `$ kill -STOP` and `$ kill -CONT` to pause/resume processes

Processes and threads – Looking at threads

top
htop

- The program sysbench is a multi-threaded *benchmark* tool

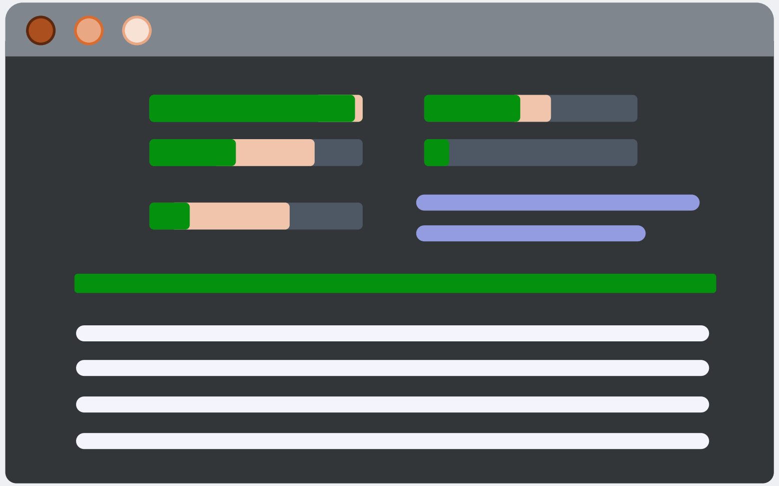
```
$ sysbench cpu --threads=3 --time=60 run
```

- `$ ps -T` displays each process' threads

```
$ ps -T $(pidof sysbench)
PID     SPID    TTY     STAT  TIME COMMAND
3172174 3172174 pts/1   Sl    0:00 sysbench cpu --threads=3 run
3172174 3172175 pts/1   Rl    0:09 sysbench cpu --threads=3 run
3172174 3172176 pts/1   Rl    0:09 sysbench cpu --threads=3 run
3172174 3172177 pts/1   Rl    0:09 sysbench cpu --threads=3 run
```

- `$ top -H` displays CPU usage for each thread *in real time*
  - when running `top`, hit **f** to display other info (e.g. CPU number)
  - alternative: use `htop` instead (has a nice text-graphics interface with colored output)

Summary – Running programs



- **Processes** are independent programs, each with a **PID**
- **Threads** are smaller units inside a process
- You can run commands as a **background process**, to keep interacting with your shell while it runs
- You can inspect running processes (`ps`, `top`) and terminate (`kill`) then

Vlaanderen
is supercomputing

More bash scripting

Conditional and looping constructs

Functions & debugging

Bash Scripting – `if` conditional construct

- **Generic form**

```
if test1; then commands1
elif test2; then commands2
elif ...
else commandsn
fi
```

```
#!/bin/bash                        # script04.sh

x=5
if [ $x -eq 5 ] ; then
  echo "x equals 5."
else
  echo "x does not equal 5."
fi
```

- Test syntax – different forms possible

`if test expression`

`if [ expression ]` equivalent form

`if [[ expression ]]` enhanced version – easier to use, e.g. in combination with variables

`if (( expression ))` for arithmetic expressions only

Bash Scripting – Test expressions

Tests with files

`file1 -nt file2`	file1 is newer than file2
`file1 -ot file2`	file1 is older than file2
`-d file`	file exists and is a directory
`-f file`	file exists and is a regular file
`-s file`	file exists and has size > 0
`-L file`	file exists and is a symbolic link
`-r file`	file exists and is readable
`-w file`	file exists and is writable
`-x file`	file exists and is executable

➢ Search for “bash file test operators” (or `man test`) to see more exotic ones…

Bash Scripting – Test expressions

Tests with strings

`-n string`	the length of the string > 0
`-z string`	the length of the string = 0
`string1 = string2`	strings are equal
`string1 != string2`	strings are not equal
`string1 > string2`	string1 sorts after string2
`string1 < string2`	string1 sorts before string2

Bash Scripting – Test expressions

Tests with integers

`int1 -eq int2`	int1 = int2
`int1 -ne int2`	int1 ≠ int2
`int1 -le int2`	int1 ≤ int2
`int1 -lt int2`	int1 < int2
`int1 -ge int2`	int1 ≥ int2
`int1 -gt int2`	int1 > int2

Bash Scripting – Test expressions

Combining test expressions

	[ ]	[[ ]]
AND	-a	&&
OR	-o	||
NOT	!	!

- Example:

```
if [[ $((x % 5)) -eq 0 && $((x % 2 )) -eq 0 ]]
then
        echo "$x is a multiple of 10"
fi
```

Bash Scripting – The while loop

- `while test; do commands; done`

```
#!/bin/bash                                                # script06.sh

count=1
while [ $count -le 5 ]; do
   echo $count
   count=$((count + 1))
done
echo "value of count: $count"

echo "Finished."
```

Bash Scripting – The while loop

```
#!/bin/bash                                                        # script06b.sh

while read jobid partition jobname user state rest; do
  echo $jobid $state
done < squeue.txt
```

- Alternatively, use this one-liner at the prompt:

```
$ cat squeue.txt | while read line; do ... done
```

- Combining `while` and `read` gives an easy (quick & dirty) way to process lines of output
  - *note: no worries about how many spaces separate fields*
- Note: `squeue.txt` can be found in the `input.zip` file

Bash Scripting – read input values

- Create variables and read their values from standard input

```
#!/bin/bash                                                      # script05.sh

echo -n "Please enter an integer -> "
read int

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "int = ${int}, var1 = ${var1}, ..."
```

- Remarks:
  - -n prevents echo from printing a new line
  - extended version: see `script05a.sh`

Bash Scripting – Command-line arguments

```
#!/bin/bash                                                     # script08.sh

echo "first argument in list: $1"
echo "last argument in list: ${@: -1}"

count=1
while [[ $# -gt 0 ]]; do
  echo "Nr of arguments left = $#"
  echo "Argument $count = $1"
  count=$((count + 1))
  shift
done
```

- Each time `shift` is executed, the value of **$#** is reduced by one, and the value of **$2** is moved to **$1**, the value of **$3** is moved to **$2**, etc.
  - *useful for looping over a variable number of arguments*

Bash Scripting – case conditional construct

```
case word in
  pattern1) commands1 ;;
  pattern2) commands2 ;;
  ...
esac
```

```
#!/bin/bash                                              # script11.sh

read -p "enter word > "
case $REPLY in
  [[:alpha:]])        echo "single alphabetic character." ;;
  [ABC][0-9])         echo "A, B, or C followed by digit." ;;
  ???)                echo "is three characters long." ;;
  *.txt)              echo "is a word ending in '.txt'" ;;
  *)                  echo "is something else." ;;
esac
```

Bash Scripting – break and continue

```
#!/bin/bash                                                         # script12.sh
while echo -n "enter number: "; read NUM
do
  if [ $NUM -eq $NUM ] 2>/dev/null; then
    :
  else
    echo " $NUM is not a number"
    continue
  fi
  if [[ $(( $NUM % 2 )) -eq 0 ]]; then
    echo " $NUM is an even number"
    continue
  fi
  echo " $NUM is an odd number"
  break
done
```

no-op

Bash scripting – Functions

```
#!/bin/bash                                                      # script03.sh
function func {              # shell function
  echo "use func for $1"
  return
}

echo "step 1"
func "step 2"
echo "step 3"
```

- Useful for sequence of commands that is often repeated
- Functions can also take arguments
- Example using functions defined in another file: `script03a.sh` and `script03b.sh`

Bash scripting – Debugging

- How to detect and handle errors in a script?

- Each finished command has an **exit status** – by convention:
  - success → exit status **0**
  - error → exit status **non-zero** *– the status values can differ for each command*

- The special variable $? holds the last process’ exit status:

```
$ ls existing_file
existing_file
$ echo $?
0
$ ls missing
ls: cannot access missing: No such file or directory
$ echo $?
2
```

Bash scripting – Debugging

- Debugging a script = inspecting the commands that it executes

- Put `set -x` at the beginning of your script
  - will print out all steps as they are executed
  - it’s a way to follow what’s going on if your script behaves unexpectedly

- Alternatively, use `set -e -u`
  - will stop the script if any command fails
  - or when an empty variable is used

- For more info info on debugging, check Debugging Bash scripts
- For other options to use with the `set` command, see Bash options

Summary – More bash scripting

- Other than *for loops*, *variables* and *arguments*, you can use:
  - conditionals (`if ...; then ...; fi`),
  - while loops (`while ...; do ...; done`),
  - case,
  - break & continue,
  - functions, …
- The **exit status** indicates success or failure
- `set -x` is great for **debugging**:
  it prints every step as it executes

The end

Course feedback

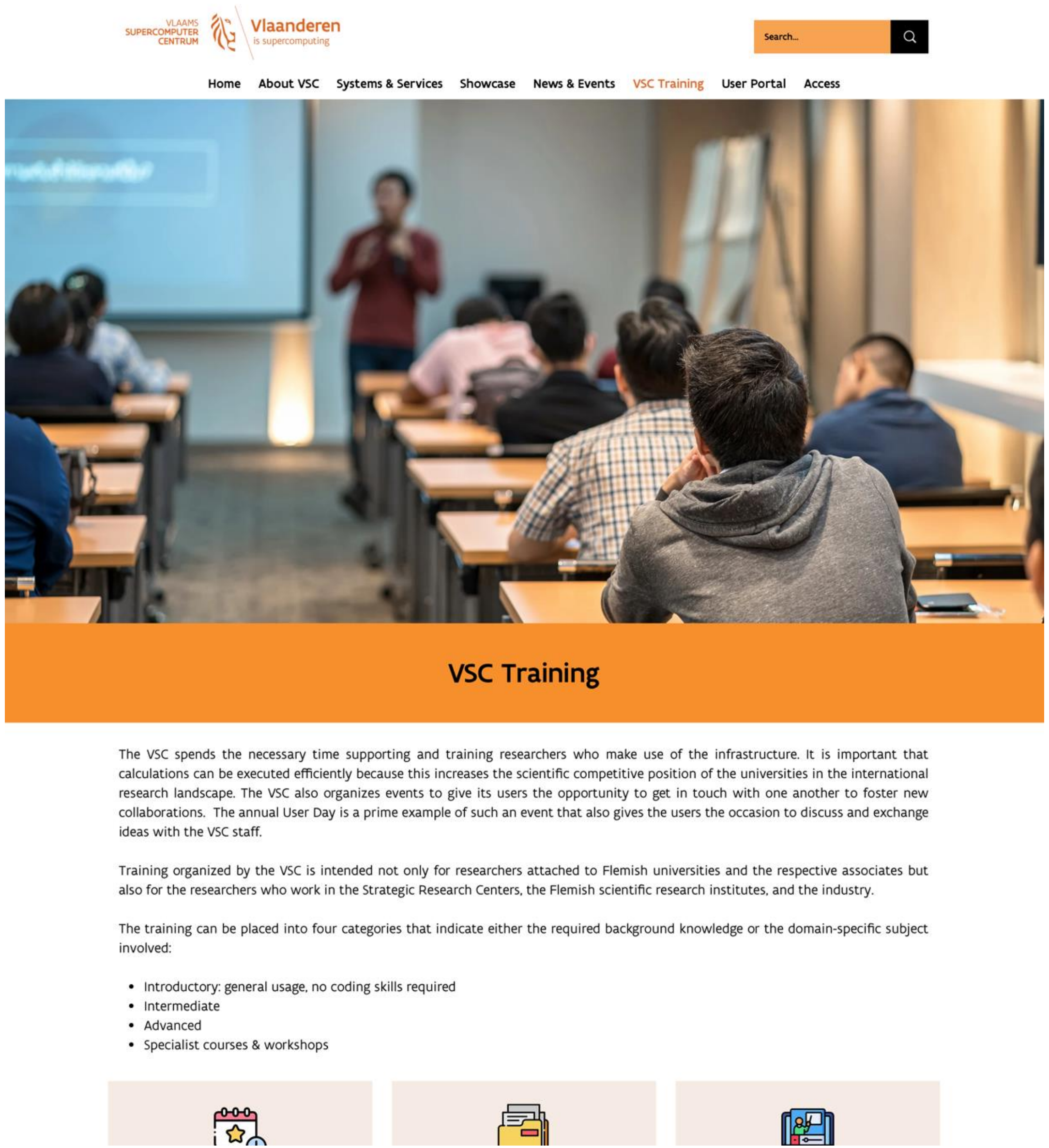
- Please fill in our short questionnaire before April 1st
- Let us know what you liked and how we can improve our courses
- *Thank you for your participation!*

Some links

- The Linux Command Line (downloadable book by William Shotts, 6th edition, 2024)
- Introduction to the GNU/Linux and UNIX command line (legacy)
- The (GNU) Bash Reference Manual
- Greycat's Wiki Bash Guide and FAQ and Reference Sheet
  - Bash Pitfalls – common mistakes made by bash users
- The Linux Documentation Project (TLDP)
  - Advanced Bash-Scripting Guide (by Mendel Cooper)
- Covering various Linux topics: Let's talk Linux @ How-To Geek, OMG! Ubuntu, …
- Cheat sheets via cheatsheets.zip or devhints.io

More training

- HPC core facility CalcUA
  - Introduction to Linux
  - HPC@UAntwerp introduction
  - Supercomputers for starters
- VSC Trainings
  - trainings organized by other VSC sites and abroad (including LUMI, PRACE, EUROCC)
- **Training sessions by Geert Jan Bex**



VSC Training

The VSC spends the necessary time supporting and training researchers who make use of the infrastructure. It is important that calculations can be executed efficiently because this increases the scientific competitive position of the universities in the international research landscape. The VSC also organizes events to give its users the opportunity to get in touch with one another to foster new collaborations. The annual User Day is a prime example of such an event that also gives the users the occasion to discuss and exchange ideas with the VSC staff.

Training organized by the VSC is intended not only for researchers attached to Flemish universities and the respective associates but also for the researchers who work in the Strategic Research Centers, the Flemish scientific research institutes, and the industry.

The training can be placed into four categories that indicate either the required background knowledge or the domain-specific subject involved:

- Introductory: general usage, no coding skills required
- Intermediate
- Advanced
- Specialist courses & workshops