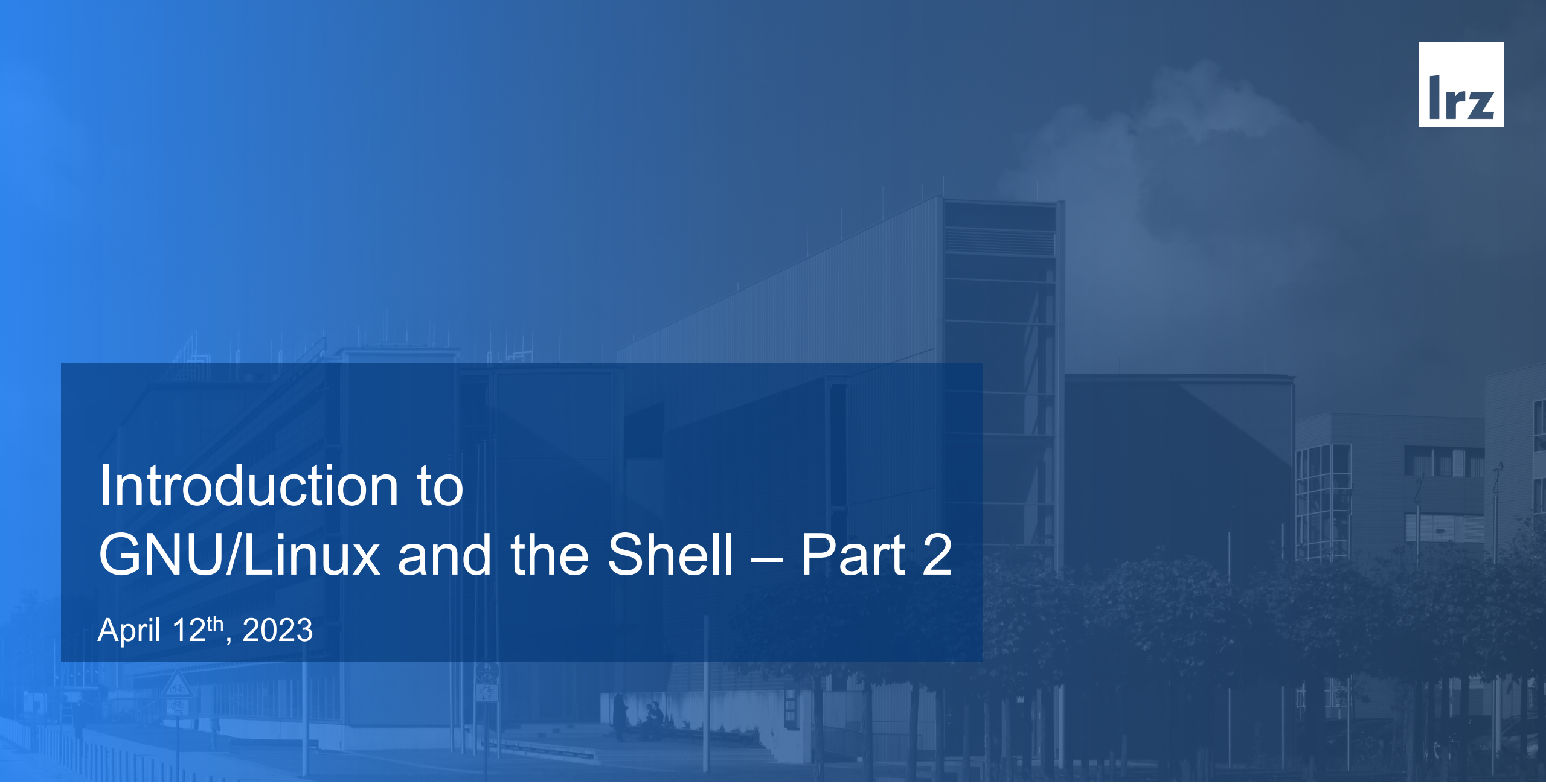
Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities

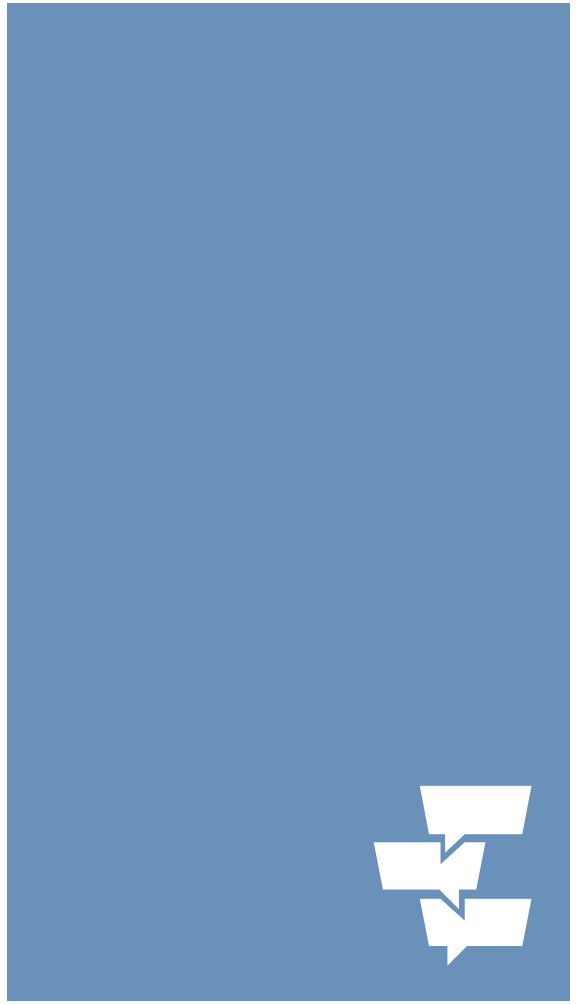


Introduction to GNU/Linux and the Shell – Part 2

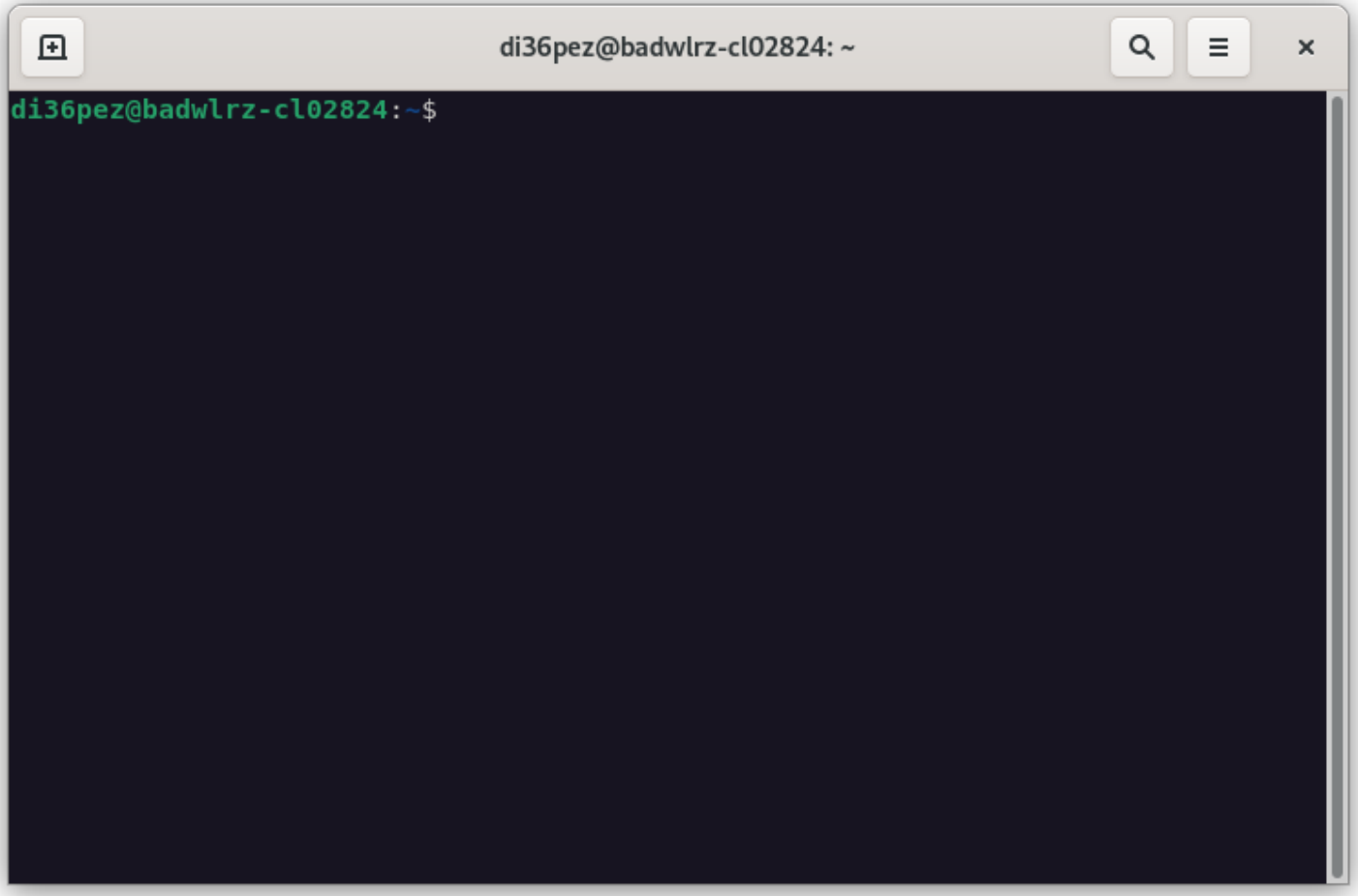
April 12th, 2023

Session Information

- The aim of this session is to provide an introduction to GNU/Linux and the Unix Shell
- You will probably benefit the most if you're not yet familiar with GNU/Linux and the Unix Shell, but if you plan to work on the AI, HPC and/or Compute Cloud infrastructure provided by LRZ
  -> by the end of this session, you should have the basic skills to successfully interact with GNU/Linux-based systems
- If you have questions, please ask at any time



A Unix-like Shell in a Terminal Application



File System Hierarchy Standard (FHS)

- On a Unix-like system (pretty much) everything is a file
- All files and directories appear (somewhere) under the root directory “/”, even if stored on different – possibly remote – devices. There are no drive letters like on other operating systems.
- Use `pwd` to get the name of the current working directory
- Use `ls` to list all files and directories in the current directory
- Use `ls /` to list all files and directories in the root directory
- Use `ls /any/other/dir` to list all files and directories in the specified directory

On Unix-like systems:
try the commands introduced on the left.

Exploring the File System

```
[root@localhost ~]# pwd
/root
[root@localhost ~]# ls
dos       hello.c
[root@localhost ~]# ls /
bin       etc       lib       linuxrc   mnt       proc      run       sys       usr
dev       home      lib32     media     opt       root      sbin      tmp       var
[root@localhost ~]#
```

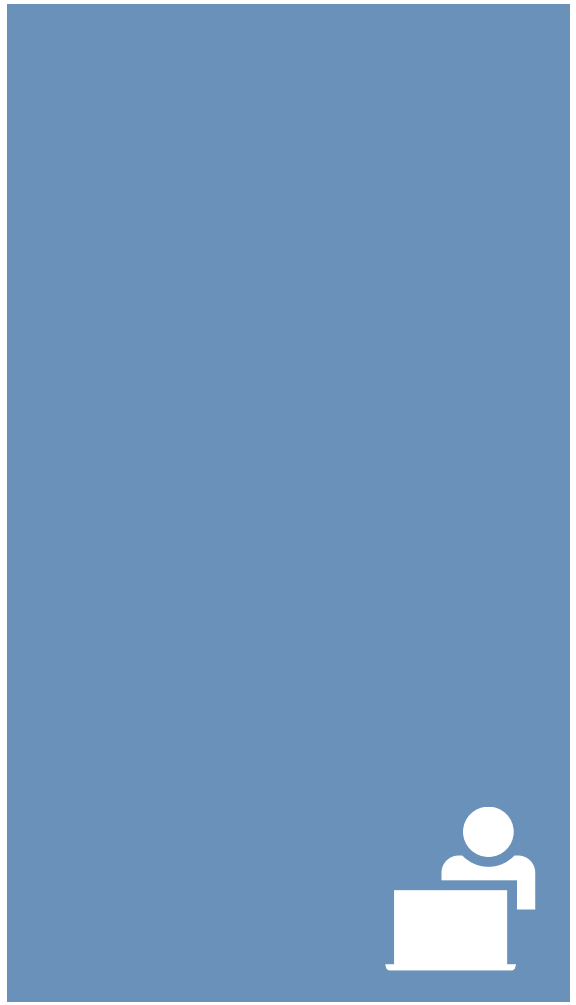
/bin*: command binaries (e.g. ls)
/etc: configuration files
/home: (regular) users' home directories
/lib*: libraries (for binaries in /bin et al.)
/media: mount points for removable media
/mnt: mounted filesystems
/root: home directory of the root user
/sbin*: system binaries
/usr: secondary hierarchy for read-only user data
/var: variable, i.e. changing files

- As an exercise, take a look at the contents of / and /usr. Can you spot the similar directory structure?
- Explore other directories. Are there any (regular) user home directories on your system?

* On modern systems, these (and /libXX) are only symlinks/shortcuts. Their former contents have been merged into their respective /usr/… counterparts, which they then point to.

Directories

- Create a new directory in your current (home) directory called „my_dir“:
  `$ mkdir my_dir`
- Change your current working directory to this folder:
  `$ cd my_dir`



Navigating Directories

```
[root@localhost ~]# mkdir my_dir
[root@localhost ~]# cd my_dir/
[root@localhost my_dir]# ls
[root@localhost my_dir]#
```

Notice the changing prompt…

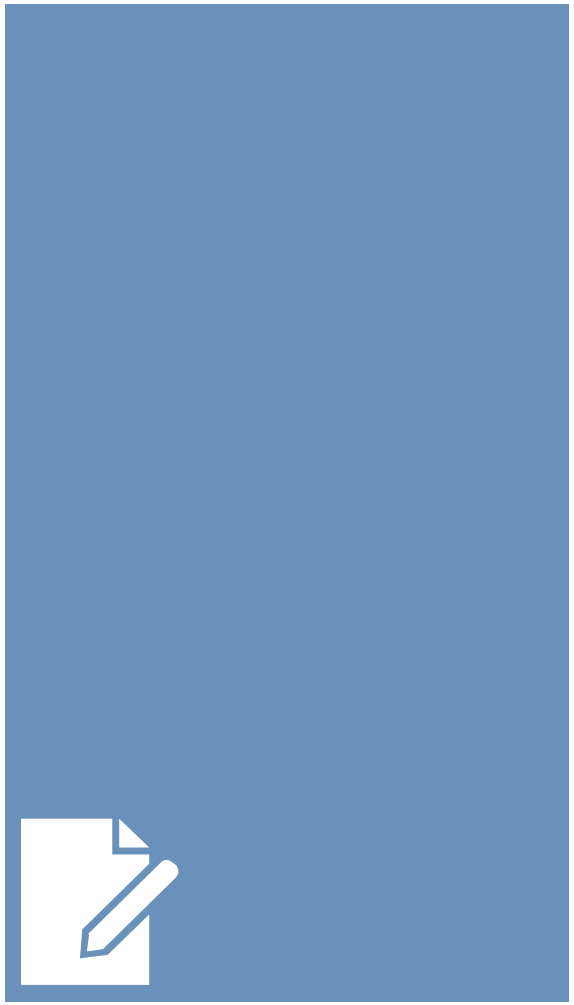
What does the “~” symbol represent?

You could use `cd ..` to move (back) to the parent folder.
A single dot `.` represents the current, two dots `..` the parent folder.

Absolute vs. relative paths:
specifying a location with a leading slash `/` indicates a start at the root of the file system (absolute), omitting it leads to an interpretation relative to the current directory

Tip: use the tab key for auto-completion!

File Manipulation



- Make sure you‘re located in the my_dir directory created earlier
- Create a new (text) file by “touching” it:
  ```
  $ touch my_file
  ```
- Can you spot the newly created file it in a file listing?
- What‘s the content of this new file? How can you tell?

File Manipulation

```
[root@localhost my_dir]# touch my_file
[root@localhost my_dir]# ls
my_file
[root@localhost my_dir]# ls -la
total 8
drwxr-xr-x    2 root     root              61 Feb 27 14:27 .
drwx------    4 root     root             158 Oct 24 19:34 ..
-rw-r--r--    1 root     root               0 Feb 27 14:27 my_file
[root@localhost my_dir]#
```

Can you spot the size of this (empty) file?

On most systems, you can use editors like e.g. nano, vi(m) or emacs to edit text files directly in the console.

Use nano to modify the existing file (write something to it):

```
$ module load nano
$ nano my_file
```

Note the shortcuts along the bottom of the *nano* screen; "^" represents the Control (CTRL) key

Use nano to create another file and write a couple of lines:

```
$ nano another_file.txt
```

Be aware of (missing) file extensions: In contrast to other operating systems, GNU/Linux does not rely on file extensions to specify the type of a file. For interoperability and clarity, file extensions can still be used, of course.

File Manipulation and Redirection

Refer to the *cat* manpage for additional information.

- There is a tool called cat. What does it do?
- “Concatenate FILE(s) to standard output.
  With no FILE, or when FILE is -, read standard input.”
- Use cat to display the contents of my_file
  ```
  $ cat my_file
  ```
- The shell allows for input/output redirection using > (and <)
- Use cat to write something to nice_file.txt and display it afterwards

```
$ cat > nice_file.txt
write something nice here
and add another line
CTRL+C
$ cat nice_file.txt
```

File Manipulation and Redirection

- Files can be appended using >>
```
$ echo “yet another line of text” >> nice_file.txt
$ cat nice_file.txt
```

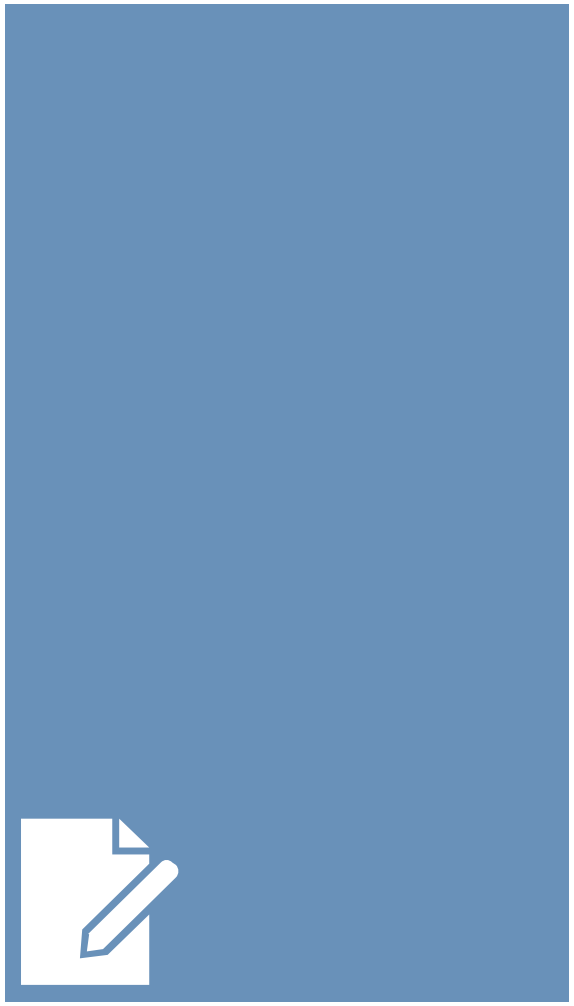
- Using << allows for the creation of here documents (input stream literals), the general format is:
```
command << delimiter  # (commonly EOF)
input stream
delimiter
```
- Try the following. Can you explain how the here document is used?
```
$ tr a-z A-Z << EOF
> all lower case
> o rly?
> EOF
```

Pipes

- Commands can be chained using | (the pipe). It will instruct the shell to use the output of one command directly as input for another command. Pipes can be used consecutively.

```
$ echo “some fancy words” | wc –l
$ echo “some fancy words” | tr “ ” “\n” | wc -l
```

File Manipulation

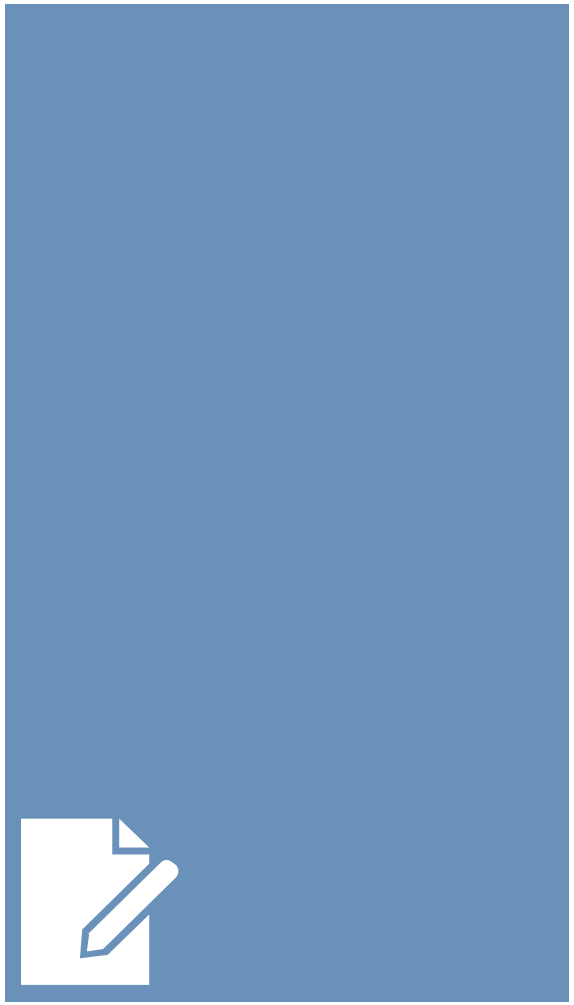


- Create a copy of “my_file” called “my_file1”:
  ```
  $ cp my_file my_file1
  ```
- Rename/move the copy “my_file1” to “new_file”:
  ```
  $ mv my_file1 new_file
  ```
- Delete the original file “my_file” :
  ```
  $ rm my_file
  ```
  Caution: there is no trash bin or undo!
- Take a look at the file listing. What is the expected output? Does it match?

File Manipulation

```
[root@localhost my_dir]# ls
my_file
[root@localhost my_dir]# cp my_file my_file1
[root@localhost my_dir]# ls
my_file   my_file1
[root@localhost my_dir]# mv my_file1 new_file
[root@localhost my_dir]# ls
my_file   new_file
[root@localhost my_dir]# rm my_file
[root@localhost my_dir]# ls
new_file
[root@localhost my_dir]#
```

File Manipulation

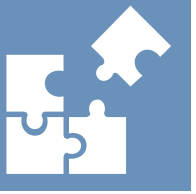


- Create two more copies of “new_file”, “01.bak” and “02.bak”
  ```
  $ cp new_file 01.bak
  $ cp new_file 02.bak
  ```
- Move to your home directory.
  ```
  $ cd ..
  ```
  alternatively:
  ```
  $ cd
  ```
  or
  ```
  $ cd /path/to/home/dir
  ```
- Copy “new_file” to your home directory.
  ```
  $ cp my_dir/new_file .
  ```
- Make a (full) copy of “my_dir” called “another_dir”.
  ```
  $ cp –r my_dir another_dir
  ```

Shell Wildcards

- Wildcards can be used flexibly for character matching in the shell:
  - Zero or more characters -> `*`
    ```
    $ ls -la my_dir/n*
    ```
  - Exactly one character -> `?`
    ```
    $ ls -la my_dir/0?.bak
    ```
- They can be combined in any way and are useful for operating on files and directories that contain certain patterns.
- Count the combined number of words in all (created text) files with a file extension:
  ```
  $ cat */*.??? | wc -w
  ```

Can you think of other patterns to match certain files or directories?



Searching: grep

- The grep command can be used to search for lines in text files that match given regular expressions (regex). Regex are patterns that match a set of strings.
- As basic building blocks (besides characters), they can include:
  - a boolean “or”, represented by the vertical bar or pipe |.
  - Parentheses () are used for grouping.
  - Placeholders (similar to shell wildcards) can be used for quantification. The ? represents zero or one occurrence of the preceding element, * represents zero or more occurrences of the preceding element and + represents one or more occurrence(s) of the preceding element.
  - The regular expression wildcard dot . matches any character and can also be combined with the quantifiers mentioned above.
- Consider options like
  -w: Select only those lines containing matches that form whole words
  -n: Prefix (file name and) line number to each match
  -i: Make search case-insensitive
  -v: Invert search, i.e. output non-matching lines
  and many more…
- Basic examples (without regex quantifiers, wildcards, etc.) are

```
$ grep something my_dir/nice_file.txt
$ grep -r “another line” .
```

Can you think of other regular expressions to match specific strings (but not others)?

Searching: find

- The `find` command can be used to search for files and directories, e.g.

```
$ find .
$ find . -type d
$ find . -name “*.txt”
$ find . -type f -name “a*”
```

Refer to the find manpage for additional information.

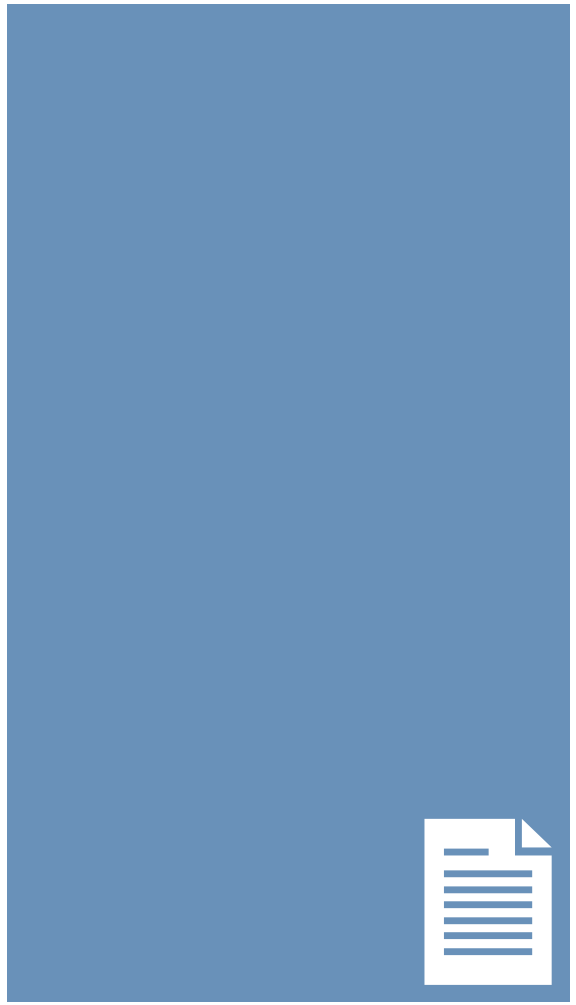
Shell Scripting

- Use shell scripts to save and re-use commands
- Create a new file myscript.sh containing the line

```
echo “This script is simple.”
```

- Once saved, you can run it explicitly (using the Bash shell)

```
$ bash myscript.sh
```



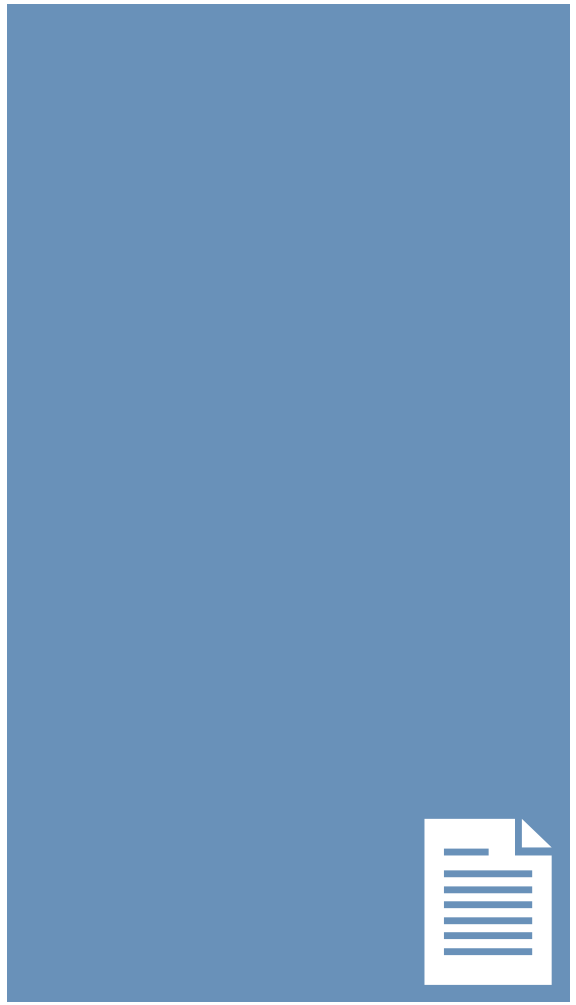
Shell Scripting

- Modify the script to allow for argument use. Add the line:

```
echo “This $1 is $2.”
```

- Provide the needed arguments when calling the script

```
$ bash myscript.sh “scripting” “getting somewhere”
```



Shell Scripting

- Add a shebang interpreter directive as the first line for direct execution:

```
#!/bin/bash
echo “This script is simple.”
```

- Afterwards, call the script directly

```
$ ./myscript.sh “attempt” “failing”
```

- What is going on?

Can you explain the unexpected outcome?

Ownership and Permissions

```
[root@localhost ~]# ls -la
total 20
drwx------    3 root     root          135 Oct 24 19:34 .
drwxrwxrwx   19 root     root          457 Feb 27 13:44 ..
-rw-------    1 root     root            0 Jul  8  2017 .Xauthority
-rwxr-xr-x    1 root     root           28 Jun 24  2017 .xsession
drwxr-xr-x    3 root     root          163 Aug 20  2011 dos
-rw-r--r--    1 root     root          242 Jul 15  2017 hello.c
[root@localhost ~]#
```

- Every file/directory is owned by a specific user
  (usually the original creator, but this can be changed)
- Every user is member of a (primary) group (and potentially additional ones)
- Notice the two “root” columns above:
  the first one is the owner of the respective file/directory (here, a user called root)
  the second one is the group assigned to the file/directory (here, a group called root)

Ownership and Permissions

```
[root@localhost ~]# ls -la
total 20
drwx------    3 root     root          135 Oct 24 19:34 .
drwxrwxrwx   19 root     root          457 Feb 27 13:44 ..
-rw-------    1 root     root            0 Jul  8  2017 .Xauthority
-rwxr-xr-x    1 root     root           28 Jun 24  2017 .xsession
drwxr-xr-x    3 root     root          163 Aug 20  2011 dos
-rw-r--r--    1 root     root          242 Jul 15  2017 hello.c
[root@localhost ~]#
```

- Permissions (access rights) for files and directories are managed in three different classes: user, group and others
- Three specific permissions apply to each class:
  - **r**ead (a file or the names of files in a directory)
  - **w**rite (modify a file or the entries of a directory)
  - e**x**ecute (a file or access file contents of a directory)

Ownership and Permissions

```
[root@localhost ~]# ls -la
total 20
drwx------    3 root     root          135 Oct 24 19:34 .
drwxrwxrwx   19 root     root          457 Feb 27 13:44 ..
-rw-------    1 root     root            0 Jul  8  2017 .Xauthority
-rwxr-xr-x    1 root     root           28 Jun 24  2017 .xsession
drwxr-xr-x    3 root     root          163 Aug 20  2011 dos
-rw-r--r--    1 root     root          242 Jul 15  2017 hello.c
[root@localhost ~]#
```

- The leftmost column represents these permissions as they apply to files and directories for each of these three classes

- Two examples from the output above:
  `dos:` `drwxr-xr-x` This is a **d**irectory. User (root) has rwx, (members of) group (root) rx and (all) other (users) rx permissions.
  `hello.c:` `-rw-r--r--` This is a file. User has rw, group r and other r permissions.

Shell Scripting

- In order to execute the previously created script file…
- …use chmod to change file permissions/mode bits
  ```
  $ chmod +x myscript.sh
  ```
- Afterwards, call the script directly again
  ```
  $ ./myscript.sh “time it” “executed directly”
  ```

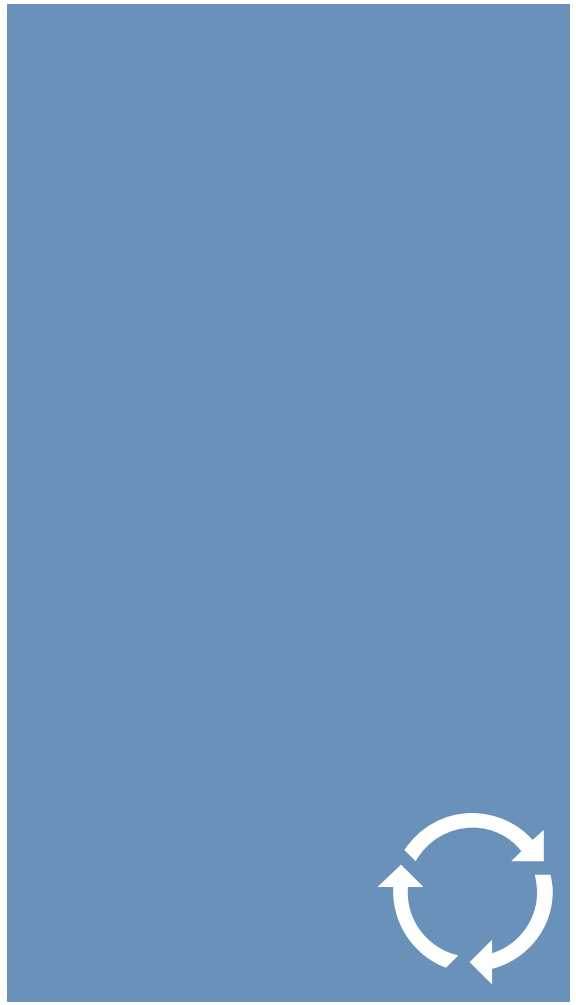
Refer to the chmod manpage for additional information.

Loops

- Loops are a programming construct which allow to repeat a command or set of commands for each item in a list
- To demonstrate the general syntax of a for-loop, create the file `myloop.sh` and execute if afterwards

```
#!/bin/bash

for VAR in first second third
do
    echo $VAR
done
```

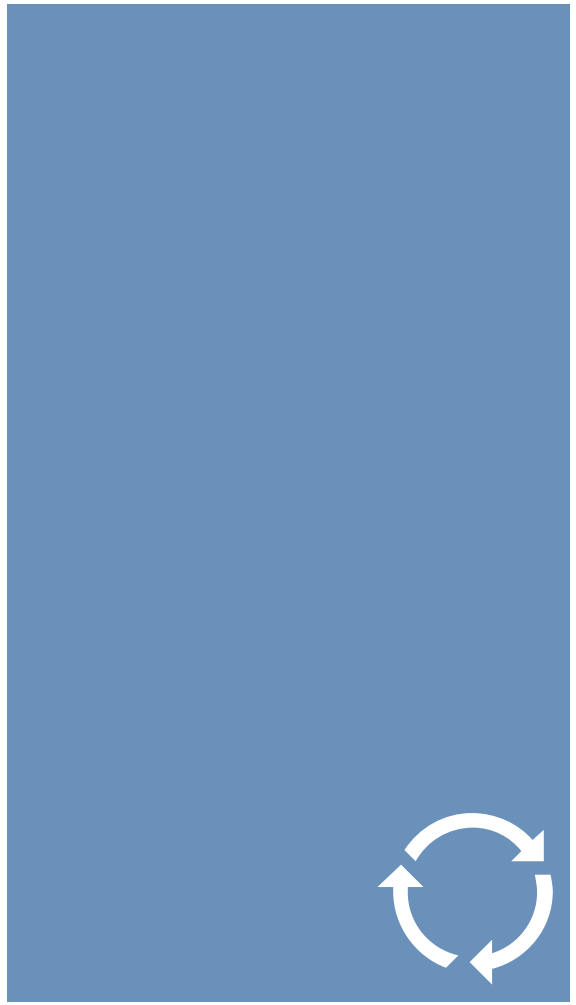


Loops

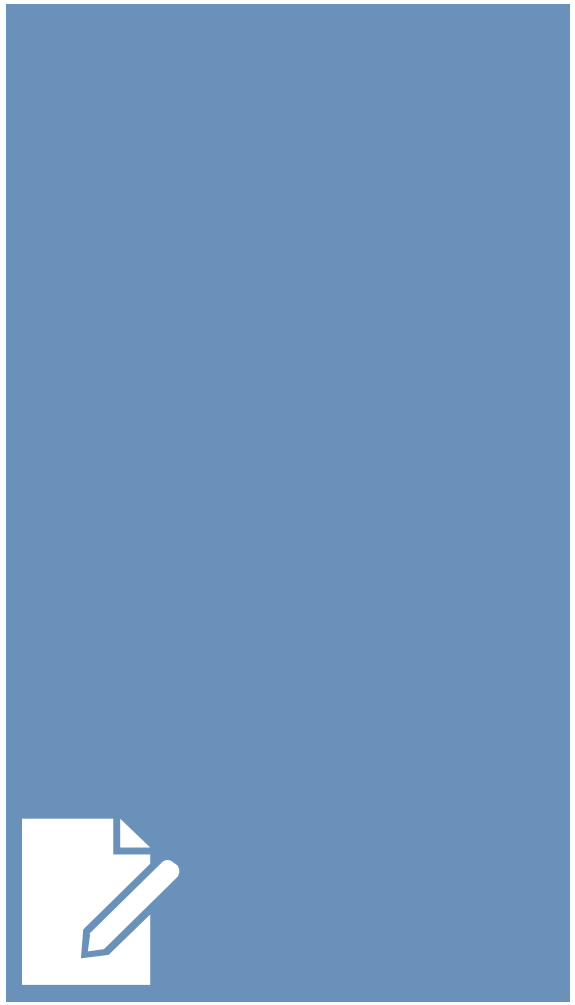
- Write a loop to print the number of words in every (text) file (created earlier) with a file extension!

```
#!/bin/bash

for TEXT in */*.???; do
    wc -w $TEXT
done
```



File Manipulation



- Finally, let‘s clean up: completely delete “another_dir”.

  ```
  $ rm –r another_dir
  ```

  Again, be cautious: there is no trash bin or undo!

- There is another command called `rmdir`? Does this also work?

Additional material

Visit https://linuxjourney.com/ for many more interactive tutorials!

One more thing: Environment Variables

- Environment variables are named values that can influence how programs are run in the shell environment (e.g. by providing context information)
- Use the command `env` to print these variables in the current environment
- To print a specific environment variable, use the `echo $VARNAME` command
  e.g. `echo $HOME`
- To set (or change) a specific environment variable,
  use the `export VARNAME=<value>` command
- On many LRZ systems, we provide advanced mechanisms to adjust these environment variables for user-specific modifications, e.g. on the high performance computing clusters a “module system” is available that (amongst other functionalities) allows for providing/running different versions of the same application (making changes to environment variables to do so).