Introduction to Linux for IMS students

This script is intended as an introduction to Linux and Linux shell commands in the IMS computer pool for students. I prepared this for the Methods in Computational Linguistics class in winter terms 2016-2021 and 2024.

Why will you need Linux shell commands at IMS? Because many classes at IMS will require you to have some knowledge of them – sometimes basic knowledge, sometimes more advanced knowledge. Classes where you will be using these commands include Text Technology, Syntax, Information Retrieval and Text Mining, the Computational Linguistics Team Labs, Speech Technology, Speech signal processing, just to name a few. It's also very likely that you'll need this for practical experiments with IMS tools during your thesis project.

This script is intended as a tutorial for self-studying, but I will offer four sessions during regular class times in the second week of the semester, where I will be present so that we can deal on an individual basis with whatever problems occur, address things that may not be explained clear enough in this tutorial, or where I can help you if you are stuck.

You don't need to have started working through this script before the first session, but I would advise you to do so. We will have only four sessions to get you through this tutorial, and keeping the pace will cost different amounts of self-study time depending on your previous knowledge. Please try to put in much self-study time right now if you don't have much computer experience, because even though you could continue to work your way through this tutorial on your own later in the year, it's much easier to do it while you can ask me in the sessions.

I hope you have fun doing this – I was intrigued when I first learned what you can do using Linux commands, so I hope you'll find them helpful, too. I am also grateful for any feedback on this tutorial – if you feel that something is missing, or would like more examples, please contact me at

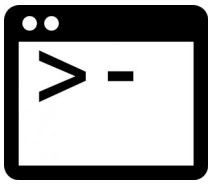
antje.schweitzer@ims.uni-stuttgart.de

Thanks! 10/24/24

Terminal and Shell

Before we start, here's a piece of advice: Instead of viewing this document in your browser, download it to your computer (right-click and save) and view it with a PDF viewer that shows the list of contents as clickable bookmarks next to the document itself. I've structured this tutorial into several chapters and I've tried to make the names telling enough. This way you can navigate through the document more easily.

A **terminal** used to be a device to get data into and from a computer. Originally, these were hardware devices which were hooked up to a "big" computer, and they looked like typewriters with a black screen. Nowadays, we still use terminals as one way of interacting with our computer; however, these terminals are just software applications which behave like a terminal: they are terminal *emulators*. But their icons are still reminiscent of the old terminal devices, they may look like this:



So how can you get a terminal? This depends on your operating system[1].

- Under **Linux**, you might find it in your dock, ready to be launched. If not, look for it in your Applications menu. You may also be able to search for Applications. Just search for an application called "Terminal", then launch it.

- Under **Mac OS X**, it's also easy: either look for it in a subfolder "Utilities" inside the Applications folder; or use the Spotlight Search (press <cmd> and <space> keys together), then start typing "Terminal"

- Under Windows, there usually is no Terminal. But on **older Windows** systems, you can install PuTTY. This is a Terminal emulator for remote connections under Windows. Once it is installed, you find it in the Start Menu; alternatively, you can type "PuTTY" in the search field right next to the Start menu. When you run it, a dialog box for configuring PuTTY pops up; we'll see what to enter in this box in the next section below.

- **Recent Windows** systems come with the PowerShell as a Terminal emulator; in this case you won't urgently need PuTTY. However PuTTY is

1 If you are not using your own computer, but are sitting in the computer pool, please skip to subsection "The window manager and the terminal on IMS computers" below.

still recommended for running applications that use a GUI (you'll learn about this later in this tutorial). If you don't want to install PuTTY, you can start the PowerShell by typing "PowerShell" in the search field next to the Start menu. If you did install PuTTY, type "PuTTY" in the search field.

Now that you've started a terminal, you can use it to interact with your operating system. However, the exact commands differ from operating system to operating system. Here we will look at how to interact with the Linux system at IMS. To this end, we'll first need to connect to IMS in your Terminal.[2]

Connecting to IMS

We will use the ssh protocol to connect to the IMS servers.

- If you are using a **Linux or Mac OS X terminal, or the PowerShell** (with PuTTY: see below), you simply type the following into the terminal window:

  ```
  ssh USERNAME@phoenix.ims.uni-stuttgart.de
  ```

  and hit the <ENTER> key. Please replace "USERNAME" in the command above by the user name you have been assigned for your IMS account. It should be 8 lower case letters; typically the first 6 letters of your last name followed by the first and last letter of your first name. If you don't have an account yet, contact me during the session, I have temporary guest accounts for this case.

  Probably you'll see a message like this

  ```
  The authenticity of host 'phoenix.ims.uni-stuttgart.de (141.58.127.109)'
  can't be established.

  ECDSA key fingerprint is
  SHA256:9HVMPTXnbAWiFhb3f4Hu5L8YFov7JqHuHwwUo0GZx2o.

  Are you sure you want to continue connecting (yes/no)?
  ```

  which is ok, so please type `yes`. Once you've confirmed, you'll probably see:

  ```
  Warning: Permanently added 'phoenix.ims.uni-stuttgart.de,141.58.127.109'
  (ECDSA) to the list of known hosts.
  ```

  You will be prompted for your password then. You will not be able to see what you're typing, but don't worry, that's for privacy reasons. Just type out the password, then hit <return>.

2 But keep in mind that you could also use the Terminal for interacting with your local computer - especially if you have a Linux system or a Mac OS - in this case, most commands introduced in this tutorial should work; however the organization of files in your file system will differ.

And then next time, your computer will remember this server and won't ask again.

- If you are using **PuTTY**, you need to enter the server details into the dialog box for configuring PuTTY:

  Click on category "Session", and enter

  ```
  phoenix.ims.uni-stuttgart.de
  ```

  as Host name. The port for ssh (22) should already be specified. If you want, give this configuration a name by typing a name into "Saved Sessions", then press "Save". Then, start the session by hitting the "Open" button. Next time, you can load that session before opening the connection.

  When you connect for the first time, you will see a box warning you that the authenticity of the server cannot be established ("The server's host key is not cached in the registry..."). Hit "Yes" anyway. You will then be prompted for your user name "Login as:". Type the username of your IMS account now (see explanation in the Linux section above for more info on your username). Finally, you will be asked for your password.

A short note regarding the server: I've suggested to use phoenix. This is a server intended for students which has 40 CPUs. However even then it can get "crowded" sometimes (this means it will be slow because many students are running processes on it at the same time). In this case, you can also try to use one of the other CPU servers for students, i.e. `nandu` or `kiwi`. The rest of the address is the same as with `phoenix`.

No matter which Terminal emulator you were using, or exactly which IMS server you were connecting to, you should now see a line that looks like this:

```
USERNAME@phoenix:~$
```

where instead of USERNAME you see your username, and instead of phoenix you may see a different server name. This is the so-called prompt, and it indicates that the **shell** is ready to take commands from you. Before we go on to work with the shell, here's how you get a shell if you are actually sitting in the IMS computer pool, at one of the student desktops there. If you are currently using your own computer, you can just skip that next subsection for now, but you may find it useful later during the semester in case you want to use the computer pool.

The window manager and the terminal on IMS computers

If you log into a computer in the IMS student pool using not a remote shell, as described above, but by physically sitting at that computer, using it as the local

host, here's what happens:

After logging in, the so-called window manager decides what your desktop looks like, which windows are opened, etc. The default window manager at IMS is Gnome, and this tutorial will assume that you are indeed using Gnome. You can select other window managers (or switch back to Gnome, in case you have changed it) after you have entered your user name: click on the little gear wheel next to the "Anmelden" ("Login") button, and select Gnome.

Once you see the desktop, hit the "Windows" key:

This will give you a dock for your "favorite" programs on the left of the screen, and a field for searching programs and settings at the top of the screen. If you start typing something now (no need to click into the field!), the characters will automatically appear in the search field, and Gnome will display a choice of programs that match your search.

You can start the programs that appear by left-clicking on them. You can also drag them to the dock of favorite programs permanently. To remove them from the dock, hit the Windows key to get the dock, then right-click on the program, select "Entfernen" ("Remove").

Let's make use of the search field a first time: If you want to **change the language of your desktop and all the menus**, you can do so by starting the Settings application: use the Windows button as described above to get the search field, then start to type "Einstellungen", which is German for "Settings". The icon of that application shows a gear wheel. Click on it, then select the Flag to change the language and region settings. You can then change the language by clicking on the first line "Sprache Deutsch" (which means "Language German"). A list of languages will come up; **I strongly recommend selecting English over your native language** (because it's easier to get help on English error messages, both from lecturers at IMS and from the internet)[3].

Once you've selected the language, Gnome will require to be restarted in order for the changes to take effect: there will be a small blue button "Neustart" ("Restart") that you should click. It'll tell you that you will be logged out, which you have to confirm. Log back on after that. When you do so, Gnome will ask you if you want to change the names of some folders that Gnome always creates for users: folders such as the Desktop, a folder for Documents, etc. So

3 If you don't like to use the search field for getting to the settings dialog, you can alternatively click on the "Power" button in the top right corner of your screen. This brings up a small dialog box, and in its bottom left corner there is that little gear wheel which opens the settings dialog.

these had been created with German names for you ("Schreibtisch" for "Desktop" for instance), and Gnome is now asking whether you want the English names instead. Make your choice, and then we can continue with our first Linux commands.

Hit the Windows button again to see the dock. It should usually contain the most important programs/applications – Firefox as a browser for the internet, Thunderbird as a mail client, etc. It should also contain an application called "**Terminal**". Its icon looks like a small black monitor:

If not, search for the Terminal application and drag it into your dock, then start it. You should now see a line that looks like this:

```
USERNAME@phoenix:~$
```

where instead of USERNAME you see your username, and instead of phoenix you will see a different computer name (in the pool, all computers are called something ending on -ente (which is German for "duck").

This is the so-called prompt, and it indicates that the **shell** is ready to take commands from you.

The shell

The shell is the program you use to interact with your operating system – in this case, Linux. If you log into an IMS server as described above, there will be a Linux shell running in the Terminal. You can interact with the system through this shell.

If the shell is ready to interact with you, it displays a **shell prompt** at the beginning of the line. This prompt often ends with a ">" sign (this is alluded to in the icon for the Terminal, see above!), or a "$" sign. Sometimes more info is displayed at the prompt, but we'll get to this later.

There are lots of single commands that you can use for interacting with the operating system – for instance, commands for listing files in a folder, for navigating through the hierarchy of folders on your computer, for displaying info on files, for manipulating and viewing files. You can even start all applications and programs by single commands in the shell instead of clicking on the icon.

It is also possible to write a sequence of commands into a file and then have the shell execute all commands in that file. This is called a **shell script** – a sequence of single shell commands to be executed in one go. They are very

helpful for automatizing typical sequences of commands.

There are different shells with slightly different syntax, the most wide-spread are probably tcsh (pronounce t-c-shell), csh (c-shell), bash (pronounce bash, or Bourne shell). Since the bash is the default shell for all IMS users, we will assume bash in this tutorial.

You can check the shell type by checking an **environment variable** called SHELL. This variable should hold the shell program itself. If you want to refer to the content of variables in the shell, you need to prepend a $ sign to the variable's name. Thus we can use a command called `echo` and type

```
> echo $SHELL
```

to get the content of the SHELL variable displayed. Please don't type the ">" symbol, throughout this tutorial it is just supposed to indicate that the above constitutes a command that is to be typed after the shell prompt. The above command should (usually…) display the user's preferred shell program itself, in our case hopefully:

```
/bin/bash
```

For the rest of this tutorial, we will use the following conventions: commands that you are supposed to type at the prompt are preceded by the ">" sign for the prompt, and they are set in the TT font used in the `> echo $SHELL` example above. I will display the shell prompt in front of the command to emphasize that this command should be typed at the prompt. **The prompt itself should of course not be typed in.** For displaying output of the shell, as the `/bin/bash` above, I'll use the same font, but no prompt symbol.

The `echo` command above is actually the very first shell command that we have used. In the above example, we have used the `echo` command with $SHELL as an **argument**. You can roughly think of commands as indicating to the system WHAT to do, and of the arguments as indicating WITH WHAT this should be done. So in our example, `echo` is a command that can print things into the terminal. By giving $SHELL as an argument, we specify that we want $SHELL to be printed. SHELL is a variable which is known to the system, and the $ sign at the beginning of $SHELL indicates that we want to get the content of that variable. So the above example worked because the $SHELL variable is set to /bin/bash in the IMS system, and this is what `echo` displayed.

If we were more playful, we might specify more than just one argument and type

```
> echo I love using a $SHELL shell
```

and this would simply print the sequence of arguments, replacing the variable by its content, but leaving the rest unchanged. Try it out.

The `echo` command was a nice first example, but you will probably be mostly using commands that manipulate files in some way, so let's have a look at how files are organized in Linux-like file systems.

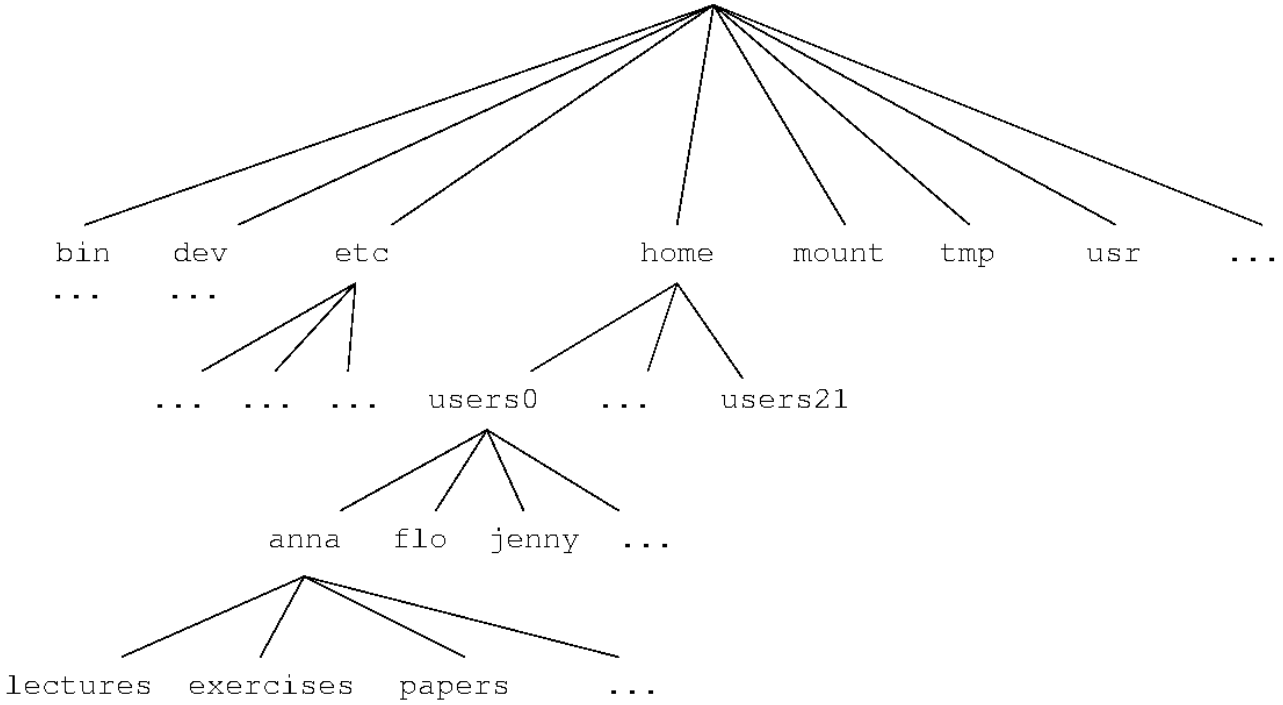
Directories and files

Case matters

Beware – in Linux and Unix, case matters!!! This means that `file.wav` and `File.wav` and `FILE.WAV` are all different files and may even co-exist in the same directory.

The directory hierarchy

In Linux, all directories (or "folders") are organized in one directory tree. The top node is called the **root node**. The symbol for the root node is /.



This is what the directory tree on a Linux machine could look like. Please note that this is an example, and that users anna, flo and jenny don't actually exist at IMS.

Anyway, every directory can then be addressed by its **path** along the directory tree, joining directory names by more "/" symbols. Here is an example: In the above hierarchy, you would address user anna's exercises directory as

/home/users0/anna/exercises

This means it can be reached by starting from the root node (the first "/" in the

path above), continuing into a directory called "home" into one called "users0" into directory "anna" into directory "exercises".

The working directory

It is important to understand that when you are running a bash shell (or any shell, for that matter), you are always inside some specific directory, which is called the **working directory**. Your working directory will always be some directory inside that tree.

The tree above is similar to the one at IMS, but I've made up the directories for anna, jenny, and flo, and left out a few others. So of course the tree is not exactly correct and also not complete, and it will certainly look a bit different on different machines and different Linux systems. But in any case, each user has one so-called **home directory** in which all their personal files can be written. And it is very common for Linux systems to have the home directories somewhere below /home (i.e. below a directory called "home" which is directly beneath the root node).

When you log onto the IMS servers, you will always get a shell in which you currently are in your own home directory, i.e. when you start, the working directory is your home directory. You can check the working directory with the following command:

```
> pwd
```

`pwd` is short for "print working directory". You can of course change the working directory in the course of a session. Since it's easy to forget in which directory you currently are, in many systems the shell prompt is configured to show your working directory. Conveniently, this is the case at IMS: we'll learn later that "~" is an abbreviation of your home directory, so if your prompt looks like this:

```
schweitt@phoenix:~$
```

the "~" in your prompt does indicate that your current working directory is your home directory. If the prompt is configured in some other may, you may have to use the `pwd` command frequently.

Navigating and listing the directories

The command to change the working directory is

```
> cd <DIRECTORY>
```

where `cd` is short for change directory, and <DIRECTORY> is the argument to the `cd` command. In the above notation, the <DIRECTORY> is supposed to be just a placeholder for an argument, with the <> brackets indicating that this is

a placeholder, which you should replace by something sensible (without the brackets then!!). I will continue to use this kind of notation when explaining the general use of commands. The placeholders will always be in upper case with these brackets, and I will try to use names that indicate what the intended arguments are – in this case here, they are directories.

So far, we have learned that you can refer to directories by their paths along the directory tree, starting from the root node. So in our example, we could change to the lectures directory beneath user anna's home directory by

```
> cd /home/users0/anna/lectures
```

or we could do this in many steps and first change to the root directory by

```
> cd /
```

followed by

```
> cd home
> cd users0
> cd anna
> cd lectures
```

Please note that if you don't specify an argument as in

```
> cd
```

you will end up back in your own home directory. (Very convenient sometimes!)

There's no fun in navigating the directories if you can't at least peek into them, so the next command we learn is

```
> ls
```

which is short for "list". In the above form it simply lists the contents of the working directory. Alternatively, specify a directory as an argument

```
> ls <DIRECTORY>
```

as in the following examples

```
> ls /home/users7/schweitt/lectures
> ls /home/users7/schweitt
> ls /
```

`ls` can even take more than one argument, so you might want to list several directories in one go as in

```
> ls <DIRECTORY1> <DIRECTORY2> … <DIRECTORYN>
```

Also, `ls` does not only work for directory arguments, you can also use it to list one or several files, i.e.

```
> ls <FILE>
> ls <FILE1> <FILE2> … <FILEN>
```

which does not give us a lot of information except that it does not work if the files don't exist – in this case, there will be an error; if the files exists, you'll get their names repeated back to you. However, we'll get to a bit more advanced uses of the `ls` command later, and then it will make sense for both directories and for files. Before that, we'll have a look at another way to refer to specific directories and files.

Relative and absolute paths

So far, we have referred to directories by specifying the full path from the root node to the directory. In the same way, we can refer to files by specifying the full path, for instance

```
> ls /home/users7/schweitt/lectures/document1.pdf
```

The full path from the top node down to the directory is called the **absolute path**. However, it is almost always more convenient to give a **relative path**, and relative means: relative to the working directory.

So relative paths only make sense in combination with some working directory, some "current position" in the directory tree. For instance, if you have changed to user schweitt's home directory

```
> cd /home/users7/schweitt/
```

and inside that home there is a folder called "lectures", as assumed above, you can list the contents of this folder by

```
> ls lectures
```

This is a relative path, and the shell will figure out that it has to prepend the path to your working directory to this relative name in order to get the absolute reference, e.g. /home/users7/schweitt/lectures. You can tell that **it's a relative path because there is no "/" symbol in front of the name.**

Depending on where you currently are in the directory tree, the above file might be referred to in a relative way as

```
> ls lectures/document1.pdf
> ls schweitt/lectures/document1.pdf
```

So now we know how to refer to files or directories that are somewhere beneath the working directory by using relative paths. The only thing missing

is: how to refer to files or directories that are higher up in the tree. For this, we use "..". This means "one directory up", and you've seen it many times when you navigated the directory trees using a graphical interface, both under Windows and under Linux: to get one directory up, you have to click on "..".

So if you're in your own home directory, you can refer to schweitt's document above in the following way:

```
> ls ../../users7/schweitt/lectures/document1.pdf
```

And if your home directory happens to be below users7, too, then you're lucky and can save one level of directories:

```
> ls ../schweitt/lectures/document1.pdf
```

Of course relative paths do not only work for the ls command, but for any command. For instance, if you are in your home directory, you could change into user schweitt's lectures folder by

```
> cd ../../users7/schweitt/lectures
```

or to the top directory by

```
> cd ../../..
```

but of course in the latter case, the absolute path would be shorter and more transparent:

```
> cd /
```

Finally, there is one convenient shortcut to refer to home directories, and that's by using the "~" symbol. User schweitt's home directory can be abbreviated as in the examples below:

```
> ls ~schweitt
> cd ~schweitt
```

If you use the "~" symbol without a specific user name, it refers to your own home directory, i.e. the command

```
> cd ~
```

is equivalent to

```
> cd ~schweitt
```

only if you are user schweitt.

Note that if you want to refer to your current working directory by a relative path, you need to use the "." symbol. So far we didn't need this, but if you

wanted to explicitly list your working directory without typing the absolute path, then you could use

```
> ls .
```

We didn't need this because if you use `ls` without an argument, it will by default list your working directory as shown above. However, later on you might need it for commands that take directories as argument, because most commands don't make this default assumption.

Please note that most users have their homes configured in a way that the contents are not accessible for other users. I've set up the `schweitt` directory so you can see its contents, but this is not typical. We'll learn later how to do this.

Creating and deleting directories

Now we know how to navigate the directories, and to display their contents. Before we go on, we'll quickly learn how to create directories, and how to delete them.

The command to create directories is called

```
> mkdir <DIRECTORY>
```

which is an abbreviation of "make directory".

Try it out: go to your home directory, and create a directory called "MethodsCL":

```
> cd
> mkdir MethodsCL
```

If such a directory exists already, don't worry, nothing will happen, and the shell will notify you that the directory existed already.

To delete it, use

```
> rmdir <DIRECTORY>
```

short for "remove directory". Again, no need to worry: rmdir only removes directories that are empty. So for the directory just created, you should be able to remove it without any problem:

```
> rmdir MethodsCL
```

Of course, the arguments to `mkdir` and `rmdir` can be specified using relative paths or absolute paths...

English, please :)

In case you have tried above to use `mkdir` twice, you have seen a message telling you that you cannot make the directory because it is already there. Unfortunately (at least unfortunate for non-Germans...) this message was probably in German. In order to get English as the system language, you can set yet a so-called environment variable: LANG. As you have learned above, you can check what it is currently set to by typing

```
> echo $LANG
```

which should currently give

```
de_DE.UTF-8
```

so we have German (de) as the language, and Germany (DE) as the country, and we will use UTF-8 as the character encoding.

Let's set this to English. I strongly recommend using English rather than your native language because it's much easier to find information on English error messages or notifications than on non-English messages. So the setting you might want is en_US.UTF-8 or en_GB.UTF-8.

To assign a variable a value, you use the `export` command as shown here:

```
export LANG="en_US.UTF-8"
```

Note that there should be no space before and after the = sign.

Unfortunately, this setting will only be remembered for the rest of this session, but we'll see below how we can make it permanent.

Exercises

1. Which of these paths are relative paths?

   a)/home/users3

   b)/home/users7/schweitt/lectures/

   c) ../schweitt/lectures/document1.pdf

   d)document1.pdf

   e)../../document1.pdf

   f) ~schweitt

2. Assume the directory hierarchy is as in the example tree above, and you are in user anna's home directory. Give three versions of a command to list the contents of user jenny's home directory (use relative and abolute paths).

3. Let's assume you are in a file system that has a different (unknown) directory structure and you want to change into user max's home directory. What command would you use?

4. How can you query the path to your current working directory?

5. Give a sequence of commands that goes to your home directory, creates a directory called "Exercise" there, and changes into this new directory.

The command line

I am a huge fan of using the command line for almost everything, even if you are actually sitting at the computer you are working on and could easily interact using graphical interfaces. That's because many things can be done in a very efficient way on the command line – especially when using the following tips and tricks.

No mouse pointer, but a history

Conveniently, the shell remembers what you've done in the past. It's very easy to "recycle" commands that you've used before. The easiest is to just use the arrow keys for up and down to go back and forth in your command history. Check it out: the arrow keys display preceding commands at the prompt. If you want to repeat one of those, just hit return, and you are done.

Maybe you want to adapt one of the last commands a bit – no problem, use the arrow keys to get the command back. Then use the left and right arrows in combination with backspace and delete to edit your command. Unfortunately you cannot use the mouse for jumping to specific positions within this command (however you can use the mouse to copy and paste stuff, see below). So you have to navigate inside the command line using the arrow keys, or the <Home>/<Pos1>[4] and <End>/<Ende> buttons. There are also keyboard shortcuts for navigation, and more, in case you want to keep your fingers on the keys – in the following, <ctrl>- means holding the <Ctrl>/<Strg> button while pressing the second key:

- <ctrl>-a (think "Anfang", or the A in the alphabet) is equivalent to <Home>: go to the beginning of the command line
- <ctrl>-d is equivalent to "Delete" and deletes the character where the cursor is
- <ctrl>-e is equivalent to <End> to go to the end of the command line
- <ctrl>-k deletes/kills anything from the cursor to the end of the line

In addition, you can use the mouse for selecting and copying text and use this in future commands. You should at least be able to copy and paste using either the right mouse button or alternatively the combination <ctrl>-c for copying and <ctrl>-p for pasting. There may be more convenient ways, but this may be different for different local operating systems, so you may have to experiment a bit... For instance, double clicking the left mouse button in my case selects only whole words. In addition, in my case what is selected is automatically copied to a buffer (no need to type <ctrl>-p) and gets pasted

4 the exact label on the key depends on your type of keyboard, hope I have covered the two most common versions here

when I click my mouse wheel. This buffer is different from the buffer in which things are explicitly copied by <ctrl>-c in my case! So I might even copy something using <ctrl>-c and select some text using the mouse, and then I can paste the first using <ctrl>-p and the second using the mouse wheel...

Note that **it is irrelevant where the mouse pointer currently is, pasted text will always be inserted at the position of your cursor** in the command line.

Starting applications from the command line

The shell has its own built-in commands, but in addition, it can execute any program that's executable – the shell just needs to know where the program is located in the file system. There are a number of directories that are by convention used to store executable programs that users might need, and each Linux distribution sees to it that the shell knows which these are (they are kept in another environment variable called PATH; if you are curious, google it – we will not deal with it in this tutorial).

So if you know the name of your application, it's usually sufficient to type its name just like any other shell command, and then the shell will know where to look for it and start it. Here's a few applications that you probably usually start by clicking on their icons, but you may as well type their names in the shell to start them:

```
> firefox
> thunderbird
> gedit
> evince
> oowriter
> ooimpress
```

However, all of the above are applications that have a graphical user interface (a GUI), and they open a "new" window on your desktop. You cannot open them if you are logged on remotely as described above. This is because Linux applications that open a GUI rely on a so-called X server running on the machine on which the window should pop up: the X server is responsible for controlling where the window is displayed, and for getting user input from the window. If you are running a Linux system on your local computer, all is well: you already have an X server running. The only thing left to do is to tell ssh that your local X server should be used. To this end, run the `ssh` command with the option -Y[5], i.e.:

```
ssh -Y USERNAME@phoenix.ims.uni-stuttgart.de
```

If you have Max OS X on your local machine, you can install XQuartz as an X Server. Once it is installed, you can run the `ssh` command with the -Y option as

5 Linux users should use -X instead of -Y as it's more secure. For Mac users -X would probably not work. But if you're concerned about security: google what you have to do as a Mac user to use -X too.

shown above.

If you have a Windows machine, and you are using PuTTY, you can install Xming from https://sourceforge.net/projects/xming/. During installation, you can specify that you don't need an ssh client (you already have PuTTY installed), so enable "Don't install an ssh client"). Once Xming is installed, there is only one modification to make in the configuration of PuTTY: In category "Connection", go to "SSH", open the subcategories (by clicking on the + symbol), select X11. Under X11 forwarding, select "Enable X11 forwarding". Save the configuration. Future connections should then have access to the X server.

If you have a Windows machine and are using the PowerShell, I don't have a solution - I did not get it to work, and I could not find an easy way around by doing a Google search. However, you can easily switch to PuTTY, and install Xming as described above; then everything should work.

If you don't want to install an X server, that's fine – you do not urgently need it. We will be using it below for setting up forwarding of your IMS mails and for configuring your shell prompt. This can however also be achieved even without opening a graphical window, as I'll explain below.

If you have an X server installed and are ready to open GUI applications – please be warned. These applications run at IMS, and all information about the graphical appearance has to be sent back and forth between your computer and the IMS server: this is going to be slow, even REALLY slow, depending on your connection. So be patient, and use it only if really needed. Opening GUI applications from the shell makes much more sense if you are physically sitting at the computer that the application is running on.

So what's the benefit of starting applications from the shell? Well, for the first two ones, the Firefox browser and the Thunderbird mail client – none, probably, and especially not if you are logged on remotely. However, for `gedit` (the standard text editor in Gnome) you can provide the name of the document as an argument, and then the application will start with your document opened already.

We will try this out and create a text document called text.txt. If you have done the above exercise, you should already have created a directory called Exercise. If not, create it now using `mkdir`. Then change into this new directory using `cd`.

Now start `gedit`, giving the name of the new file as an argument. Please put a & sign after the command, as shown below. This tells the shell to run that command "**in the background**", which means that the shell is not busy and can still interact with you while the command is running (i.e. while the `gedit` window is open). If you don't put the &, the command still works, but the shell will be busy until you close the `gedit` window.

```
> gedit text.txt &
```

This should open a `gedit` Window with the new (empty) file opened. (It might take a while, depending on your connection.) Type something, and save. Why is it convenient to give the name of the file as an argument? Well, if `gedit` already knows which file it is supposed to open and edit, you don't have to go through a file opening dialog, and also not through a file saving dialog when you're saving the file.

If you don't have an X server running on your local machine, the above won't work. In this case, please use `nano` as the editor[6]. It's a graphically much more simple editor. In this case, don't send it to the background, i.e. start it without the & sign. But do specify the name of the new file, i.e.

```
> nano text.txt
```

Your terminal will turn into a blank window, with a menu at the bottom. As in the `gedit` window, you can now type something. You can save using <ctrl>-o (write out) and exit using <ctrl>-x (see the labels at the bottom of the window). If you've set the system language to English before, as explained above, these labels should be English - if not, you'll see German (Speichern for write out, and Beenden for exit.

When you've saved the file, either using `gedit` or using `nano`, list the contents of your working directory – hopefully, the text.txt is there… Keep the file, we'll use it later. If all is well, close `gedit`.

Hidden files, and changing settings permanently

Now that we know how to use `gedit` or `nano` to create and edit files, we can use it to make the language change for the shell permanent. In order to do so, we need to look at hidden files first. Hidden files and directories in Linux have names that start with a "." symbol, and the convention is that they are usually not listed. So when we used the `ls` command above, we did not see them. Similarly, if you use a graphical interface for managing files, such as the Gnome Commander under Linux (this application is called "Dateien" in German), or the Finder on the Mac, they are usually not listed.

If you want to see them, you need to use

```
> ls -a
```

instead of just `ls`. (For memorizing: -a is for "all").

Go to your home directory, and try it out.

You should see several entries that start with a ".", and some of them are files,

6 If you're an experienced shell user, you can also use `vi` if you know how to use it.

and some are folders – for instance folders in which your settings for your browser or your mail client are stored, or settings for your gnome desktop. The file that we need now is called

```
.bash_profile
```

It can be used to add user-specific settings for environment variables etc. for the `bash`.

Open it using `gedit` or `nano`.

You will see that there is a line

```
# User specific environment and startup programs
```

The # is used for comments in the bash syntax, i.e. this specific line will be ignored by bash, it's just intended to give users an idea of where to put what.

If you don't have this file, you are probably not in your home directory; in this case, change to your home directory now and try again.

We can now put the definition of the LANG environment variable that we used above into the .bash_profile, below the above comment. This will then set the language to English for future sessions:

```
export LANG=en_US.UTF-8
```

In case you also want to use English number, date, and time formats etc., you can also set LC_ALL to en_US.UTF-8 – this would automatically include setting the language to English, but also use the other standard English formats. In this case, use instead of the above:

```
export LC_ALL=en_US.UTF-8
```

Save .bash_profile and close.

If you want to get the current bash to use this setting starting right now, you can type

```
> source .bash_profile
```

This will simply execute all commands in the file, including the newly added one. Otherwise, the change will come into effect next time you log into your account.

Forwarding your emails

This is somewhat unrelated to the shell, but now that we've discussed hidden

files, we can quickly check whether emails to your IMS email account[7] get directed to your st-account (this should be the case for IMS students). To do so, change to your home directory. There should be a file called .forward. Open it using `gedit` or `nano`

```
> gedit .forward &
```

or

```
> nano .forward
```

There should be your st-account email address in the first line. If you want, you could change this and enter your private email account here. If you want to forward to several addresses, add more lines – one line per address. Make sure you that there is a line break at the end of the last line (i.e. see if you can move the cursor to the line below the last address, if you can't, go to the end of the last line and hit the ↵ key on your keyboard). Once you're happy with the addresses you are forwarding to, save the file and close. Make sure that there is no typo.

Please don't ever (!) forward the mail from your private account to your IMS account if you are also forwarding your IMS mail to your private account. This creates a really nasty loop that will really, really annoy system administration.

Command-line completion

Let's return to the command line. Another convenient feature of the command line is that it will try to complete the commands for you if you hit the <Tab> key.

How does it do so? It will assume that the first thing that you type must be some command, since the syntax is always

```
> <COMMAND> <ARGUMENT1> <ARGUMENT2> … <ARGUMENTN>
```

If you start typing a command and then hit the <Tab> key once, the shell will try to complete that command. If there are several commands or applications that match what you have typed so far, the shell will display all possible

7 If you don't know what account I'm talking about: you should have got an info sheet about your IMS computer account. On that sheet it is explained that with your account you also got an email account (firstname.lastname@ims.uni-stuttgart.de) which you can use. On that sheet it is also explained how to retrieve messages from this account. This is an extra account, in addition to the student account you got when you registered as a student (which would be something like st1234@uni-stuttgart.de). We often send talk announcements and job ads or other interesting information to these IMS email accounts, but not to the st-accounts, because no mailing list exists that would reach exactly those student st-accounts that belong to IMS students. So it is recommended to make sure you get these messages. Also, it is easier for IMS teachers to figure out your IMS email address than your student email address, since the st addresses are only accessible to teachers of classes that you are registered for, or examiners of exams you registered for. So if you haven't registered for a class yet, your teacher will not be able to figure out your st address unless you tell them.

matches. If there's only one match left, it will automatically complete the whole command.

For instance, if you start to type "gedit" in a shell on the computers in the IMS computer pool, as soon as you have reached "ged" the only match left is gedit, and at this point, hitting <tab> will complete the name. Before this point, you get the matches displayed whenever you hit <tab> twice – fewer and fewer as you add more characters.

Once you have typed a command or application, the shell knows that commands and applications often can take one or several files or directories as arguments, and accordingly, when you type a blank after your command and then start to type a second string, the shell will try to match this string with a file or directory. So, to return to the above example with gedit (or nano): Go to the directory where you have created your text.txt file. Start typing gedit – at IMS, once you are at "ged", the command should be unambiguous. Hit <tab> and the shell completes to gedit. Then if you type a blank and then hit <tab>, the shell will assume that you will next specify some file in this directory, and if you really only have the text.txt in your directory and no other file, the shell will write it out for you even before you have typed its first letter… If you don't want to give a file in this directory as an argument, but one that's located elsewhere, start to type the path – no matter whether relative or absolute – and the shell will try to complete the path. Try and type "gedit ../" and then hit <tab> twice – you'll see that the shell will list everything in the directory above your current directory, and as you specify further letters, will narrow down the alternatives. Adding a second blank after your first argument will cause the shell to try to add another file or directory, in the same way as for the first one. This should work most shells, i.e. also in tcsh or csh for instance.[8]

File name expansion and globbing characters

One last helpful feature of the shell that we will discuss here is that it allows to refer to files and directories in a more general way by way of name patterns. This is done by means of so-called **globbing characters**, namely the * and the ? symbol. * can be used to stand for a sequence of 0 or more characters (except "/") in a file or directory name, and ? stands for exactly one character in such a name (again, except "/"). Here's an example:

```
> ls *.txt
```

8 You may have noticed already that bash is even more clever. For instance, it knows that the program xpdf (a viewer for PDF files) only makes sense with a .pdf file as an argument. So if you type xpdf and a blank, then hit <Tab>, it will only display files with the appropriate extension, but ignore files with other extensions such as .txt. If you like, try it out – get a PDF file from somewhere, save it in the same folder as the test.txt above, then see which filenames the autocompletion offers. This is because bash offers something called programmable completion. Using this it is possible for each command to specify which arguments are expected (and even more). For some very common commands, such as xpdf, our Linux distribution contains such specifications by default. If you want to learn more, google "programmable completion".

The shell will extend the string "*.txt" to all files or directories in your working directory that start with an unknown number of characters, followed by .txt. So if we had in the working directory three files text1.txt, text2.txt, and text99.txt, the shell would expand the above command to "ls text1.txt text2.txt text99.txt" (this expansion is done silently, you don't see it) and so you get the three file names listed:

```
text1.txt       text2.txt   text99.txt
```

This is one use of `ls` on files that actually gives valuable information: we now know that there are exactly three files matching the specified pattern. If you want to try this, change to the following directory:

```
> cd /mount/studenten/MethodsCL/2021/Linux/Globbing
```

I have created the three files there, so you can test the above command. Please note that you won't be allowed to modify these files, or add new ones in this directory. Don't be shy, this means that you don't need to worry about accidentally deleting or changing anything.

To illustrate the ? globbing symbol, we'll use

```
> ls text?.txt
```

In the above case, this would give the following output:

```
text1.txt text2.txt
```

Here, text99.txt does not match the pattern since there are two characters after the string "text", not only one.

For the sake of simplicity, the examples here assume that you have changed your working directory to the above directory. However, of course you can use globbing characters in longer paths, so you could also do this from any directory at IMS by

```
> ls /mount/studenten/MethodsCL/2019/Linux/Globbing/text?.txt
```

When could this command with globbing be helpful? Often, I hope ;) One scenario: assuming you have downloaded a database of image data with images belonging to different classes. For instance, you could have dog pictures (with extension .jpg for jpg format) and descriptions (in .txt format) in directories starting with dog_ and and the same for cats in directories starting with cat_ and maybe images of other animals in other directories. You could then list only the dog pictures by something like

```
> ls dog_*/*.jpg
```

or only the dog descriptions by

```
> ls dog_*/*.txt
```

Or you could list all .jpg images that you have in any subfolder by

```
> ls */*.jpg[9]
```

Note that in all the examples above, it is important to understand that "*" or "?" will not match the "/" symbol. This is why you need to specify "*/*.jpg" in the above example instead of only "*.jpg" – the "*" will not match something like "cat_01/image", but "*/*" will.

Another useful and very easy application: If you have a database of public domain texts, each text in a file with the filenames following the pattern author_genre_year.txt. You could then see which texts were by Goethe by

```
> ls goethe_*.txt
```

or see which dramas you have by

```
> ls *_drama_*.txt
```

Exercises

1. How can you start a program "in the background"?

2. Specify a pattern that matches all files in /home/users0/anna which have the extension .pdf (i.e. which end on "pdf").

3. Specify a pattern that would match the following file names:
   phonetics1.pdf
   phonetics1.txt
   phonetics21.pdf
   phonetics21.txt

   but not the following file names:

   phonetics2.pdf
   phonetics2.txt

4. The directory /resources/speech/corpora/TIMIT-1/timit/train/dr1 is part of the famous TIMIT-1 database and contains directories with recordings of speakers saying simple sentences. These directories correspond to speaker IDs. Speaker IDs start with f in case the speaker was female, and with m if the speaker was male (the TIMIT database is from the 90s).

9 For advanced users: in addition to using * as a , you can specify list of alternatives in curly brackets, like this, and get only cat and only dog descriptions in one command:
`> ls {cat,dog}_*/*.txt`

a) Specify a command to list the contents of all directories of female speakers. What is the output?

b) If you managed to specify the above command, you have seen that the directories contain files with different extensions, e.g. txt, wav, wrd, phn. These contain the text, the audio recording, the words line by line and the phoneme transcription (we'll discuss in the Speech part of the lecture what phonemes are). Specify a command that lists all txt files in speaker fvmh0's directory.

c) Specify a command that lists all text files in any female speaker's directory.

Manipulating files and directories

Copying and moving files

Copying and moving files works similarly in many cases: both can be used in two ways:

```
> mv <FILE> <DIRECTORY OR FILE>
> cp <FILE> <DIRECTORY OR FILE>
```

i.e., they take a file name as a first argument and a "target" directory or a "target" file as a second argument.

If the "target" is the name of an existing directory, then the file is moved/copied to the target directory, keeping its current file name. The only difference between `cp` and `mv` is that `cp` makes a copy and leaves the original file untouched, while `mv` results in really moving the file to the directory.

If on the other hand the "target" is not an existing directory, then it is assumed it specifies a file. In this case the first file will be moved/copied to a file with the specified name – i.e. the new file will have the name specified by the second argument, and the same content as the original file. Please note that if a file of that name existed already, this would result in overwriting the existing file, i.e. the original file will be lost. See below how you can avoid doing this accidentally.

Note that

```
> mv <FILE1> <FILE2>
```

consequently results in renaming the file to a new name, specified by the second argument. Actually, there is no other "rename" command in the shell; you **need** to use `mv` for renaming.

Note that moving or copying a file into a new directory only works if that directory exists – i.e., while both commands may create new files that didn't exist before, they cannot create new directories (how would the shell even be able to tell that your second argument was intended to be a directory instead of a new file name?). So **to move or copy files to directories that don't exist yet, you need to create these first using the mkdir command introduced above**.

Both commands can be used with more than two arguments, but only if the last argument specifies a directory:

```
> mv <FILE1> <FILE2> … <FILEN> <DIRECTORY>
> cp <FILE1> <FILE2> … <FILEN> <DIRECTORY>
```

This will result in copying or moving all N argument files to the target directory.

Remember that the way to refer to your current working directory in a relative way is to use ".". So for instance copying a file called experiment1.results from the directory above your current working directory to your working directory is achieved by simply typing

```
> cp ../experiment1.results .
```

The "." symbol works not only for the `cp` command, but for any command.

Since both the `cp` and the `mv` command can cause existing files to be overwritten, at IMS most user accounts are configured to use a more "careful" version of `cp` and `mv`. To understand this, we need to introduce **command options**.

Command options

Command options are a way to provide more fine-grained control over a command's behavior or to provide additional functionality. Options are typically specified right after the name of the command and before the arguments to the command. Some options even take their own arguments. Let's look at an example.

The above `mv` and `cp` commands by default don't care if they overwrite existing files. This is dangerous – files that are deleted that way are not stored in some Trash folder, instead, they are really gone forever. The same holds when you remove files using the command for removing, which we will introduce below.

So if you're new to this all, you might want to consider being a bit more cautious and to make these commands ask for confirmation if they would overwrite existing files. Actually, system administrators at IMS usually configure new user accounts to use an interactive version of `mv` and `cp`, which always asks. You would usually only get this interactive behavior if you use

```
> cp -i <FILE> <TARGET FILE>
> mv -i <FILE> <TARGET FILE>
```

In the above examples, "-i" is an option that specifies to use the interactive version of `cp` and `mv`.

So on new user accounts at IMS, if you type "cp <FILE> <TARGET FILE>", this usually effectively runs "cp -i <FILE> <TARGET FILE>", and you will have to confirm or reject the operation by hitting y or n. If you don't like this, there is a way out. Or, if your account is not configured in this way, there's a way to get exactly this behavior.

What the administrators might have done for your account was simply to define so-called **aliases.** In this case the aliases state that "cp" or "mv" are meant to run "cp -i" and "mv -i" instead. So the "cp" would be an alias (a shortcut) for typing "cp -i". You can check which aliases are defined for you by typing

```
> alias
```

This will list all defined aliases and could look like this:

```
alias cp='cp -i'
alias egrep='grep -E –color=auto'
alias fgrep='grep -F –color=auto'
alias grep='grep –color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l –color=auto'
alias ls='ls –color=auto'
alias mv='mv -i'
alias rm='rm -i'
```

You'll see that there are a lot of aliases, and among them, you might find the two for the `cp` and the `mv` command, as above. You can also check for specific aliases by giving the alias name as an argument:

```
> alias mv
```

If you have an alias defined for `mv`, this will return the command that is used for `mv`. So possibly if you do this, you might see

```
alias mv="mv -i"
```

If you don't have an alias for `mv`, nothing will be returned.

It's easy to remove aliases, just type

```
> unalias mv
> unalias cp
```

and this will remove the two aliases. Beware, after doing that, you will, for the rest of the session, be using the non-interactive versions of the two commands, which won't care if you're inadvertently overwriting stuff or not. However, you're spared typing y/n for each single file you might want to overwrite.

In case you want to re-define the aliases, or define them in case your account wasn't configured to have them, you can do so by

```
> alias mv="mv -i"
> alias cp="cp -i"
```

to get the more cautious versions.

Beware, both `alias` and `unalias` only affect the current session; in a new session, you would be back to the original behavior!! So if you want to enable or disable them permanently for your bash, you can again write this into a hidden file. This time it's recommend to add it to .bashrc in your home – this is also where system administration usually puts aliases for new users.
Open .bahsrc using `gedit` or `nano`, and put the specification of the aliases somewhere below the line

```
# User specific aliases and functions
```

Remember that if you want the changes you made to .bashrc to come into effect at once, you'll need to type

```
> source .bashrc
```

in your home directory after you've changed (and saved) the file. Otherwise they will only come into effect next time you start a shell.

Copying directories recursively

Copying empty directories works exactly as copying files. Copying non-empty directories is different: if you try to copy a directory that has files in it, `cp` will notify you that it left out the directory. If you do want to copy the whole thing recursively, i.e. directory including all contents, you'll need the -r option to the copy command, for recursively copying:

```
> cp -r <DIRECTORY> <TARGET DIRECTORY>
```

Note that moving directories doesn't need any such option; you can move (or rename) directories like this

```
> mv <DIRECTORY> <TARGET DIRECTORY>
```

this will move/rename the whole directory. If <TARGET DIRECTORY> exists, <DIRECTORY> will be moved inside that directory. If it does not exist, <DIRECTORY> will be renamed to the directory location and name you specify, leaving the names of files and other directories inside it unaffected.

Removing files and directories

The command to remove files is

```
> rm <FILE>
> rm <FILE1> <FILE2> … <FILEN>
```

For removing empty directories, you can use the `rmdir` command introduced above. If you want to remove non-empty directories recursively, you will again need an option to the `rm` command, in fact, the same as for the `cp` command above:

```
> rm -r <DIRECTORY>
> rm -r <DIRECTORY1> <DIRECTORY2> … <DIRECTORYN>
```

Note that this would also work for empty directories, so this is a more general **(but much more dangerous)** alternative to the `rmdir` command.

Again, there is a more cautious version of `rm`: if you specify `rm -i`, `rm` asks before removing files. You can again put an alias for this in your .bashrc file, as documented above for `mv` and `cp`.

Permissions and groups

We've learned how to copy, rename and delete files. If you have wondered whether you can manipulate arbitrary files in this way – no, you can't. The file system defines for each file and directory which users can access it in which ways. To see what people are permitted to do for a file, we need the `ls` command again: it provides an option "-l" for "long" listing, and if we use this option, we see much more information than just the file name. If you use it on the file text.txt created in the above exercise, you may see something like

```
> ls -l text.txt
-rw-r-----. 1 schweitt sem2324 13 18. Sep 10:23 text.txt
```

What does this mean? Let's look at the middle part first. The third column states the user name of the owner of the file – in the above case, the user is schweitt. The next column indicates a group of users – here, it's sem2324, which at IMS indicates the group of students who got their accounts in 2023/2024. Groups are useful because they allow to define permissions for a larger group of users which can be different from what everyone else on the file system is allowed to do, and different from what the owner herself/himself is allowed to do.

The next column indicates the size of the file (13 bytes in this case), then we have the date when this file was last changed, the time of change, and the file name.

The interesting part is the first column. It consists of 10 characters which indicate in this order:

- File type – a regular file (-), a directory (d), or something else which we won't discuss here
- Owner read – owner is allowed read access (r) or not (-)
- Owner write – owner is allowed write access (w) or not (-)
- Owner execute – owner is allowed to execute (x) or not (-)
- Group read – group is allowed read access (r) or not (-)

- Group write – group is allowed write access (w) or not (-)
- Group execute – group is allowed to execute (x) or not (-)
- Others read – others are allowed read access (r) or not (-)
- Others write – others are allowed write access (w) or not (-)
- Others execute – others are allowed to execute (x) or not (-)

So in the above example, we have a regular file; schweitt is allowed to read it (i.e. she can for instance copy the file, or look at its contents) and to write to it (i.e. she can modify or delete the file), but not to execute it (and actually, it wouldn't make sense to execute a .txt file – executing is for programs). The sem2324 group however is only allowed read access, and all other users don’t even have read access. The little dot at the end of this first column finally indicates that a so-called SELinux context is defined for that file. We won't go into detail here, if you're interested in more, google it. SELinux contexts are a means to control in a more fine-grained way what users and processes are allowed to do with which files, and they help to limit the damage users could do by inadvertently running malware programs.

There's one column I haven't mentioned – it's the one after the permissions. It states how many links there are. However, what counts as a link here is very complicated, and we will not make use of this information throughout the tutorial, so we will just ignore this column here.

If you look at permissions for a directory, these are indicated in the same way. But it's worth noting that in the case of directories, in order to list their content, permission to execute **and** to read is required.

Now that we know how to find out the permissions for a file, here is how to change them. The command is `chmod` (for change access mode). It can be used in several ways; here we will only discuss one possible way using what's called the symbolic notation: after the `chmod` command, one can specify for who we want to change the permissions (u for user, g for group, o for others, a for all three in one go), and then we can add or remove read, write and execute permissions by + and – signs. Here are some examples:

```
> chmod ug+rw text*.txt
```

This would add read and write permissions in all files matching the pattern, for the owner (abbreviated u, the user) and the group (g). It does not change anything in the permissions for others, i.e. these will remain unchanged.

```
> chmod o-rwx Exercises
```

This will make Exercises unreadable and unwritable and non-executable to

others.

It's possible to recursively change all files below some directory using the -R option:

```
> chmod -R o+rx,o-w <DIRECTORY>
```

would allow all other users to view and execute, but not to modify, all files below <DIRECTORY>.

Finally, a user might be in various groups, not only in sem2324, and might want to give another group permissions to some file. To find out the groups that a user belongs to, use

```
> groups <USERNAME>
```

For your own groups, it's sufficient to say

```
> groups
```

This will list all groups that you are in.

Now if a user is in group sem2324, but also in group xyz, then they could change the group of the file in the following way

```
> chgrp xyz text.txt
```

In general, the command is (either for a single file or directory, or recursively for a directory using option -R)

```
> chgrp <GROUP> <ONE OR SEVERAL FILES OR DIRECTORIES>
> chgrp -R <GROUP> <ONE OR SEVERAL DIRECTORIES>
```

But note that these commands are only allowed if the user who runs the command really is in that group. You cannot make a file belong to a specific group that you are not part of.

To conclude this section on permissions, one piece of advice: In my opinion users should give reading permissions as often as possible. It is often very, very helpful to be able to see each other's files, especially when working in groups, or when things don't work and you need help!

Exercises

1. Which command do you need to copy the file summary.txt from user schweitt's home to your working directory?

2. How could you copy user schweitt's directory "Exercise" recursively to a folder "Solutions" which exists in your own home directory?

3. What is the command to rename a file called "pirntout.pdf" to "printout.pdf"?

4. Which command would move all files in your working directory which have the extension ".txt" to an existing folder called "tmp"?

5. Which dangerous command would remove all files and directories in your folder ~/tmp (but not the directory tmp itself)? (Be VERY careful if trying this out!!!)

6. Remember the TIMIT-1 database form the last block of exercises? Some of its files are located in subdirectories beneath the directory /resources/speech/corpora/TIMIT-1/timit/train/dr1. These directories correspond to speaker IDs. Speaker IDs start with f in case the speaker was female, and with m if the speaker was male (the TIMIT database is from the 90s). Which command would copy all .txt files for the male speakers to a directory called timit inside the current working directory? (To avoid dealing with too many files, let's take only those speakers located in the dr1 portion of the TIMIT-1 database, i.e. below the directory specified above)

7. How can you check the permissions for your own folder ~/Exercise?

8. Which command would you use to take away your own write permissions for file "important.notes.txt" in your working directory? (This would be useful if you want to make sure that it won't be accidentally deleted or overwritten.)

9. Which command would you use to take away your own write permissions for everything beneath the directory "Notes" in your working directory?

10. Assume anna owns the directory which is listed here:

    --wx--x---. 2 anna sem1617 4096 21. Okt 09:24 Notes.txt

    What is strange about the permissions? Which command would you suggest to give more reasonable permissions?

Inspecting file contents

In many cases, in particular when working on tasks in computational linguistics, the files that you deal with are simple text files which contain only printable characters. This holds for most log files which may be output by some program, it holds for files containing source code, for most label files in phonetics, for most text corpora, etc.

Even though you can of course open such simple text files using a text editor like `gedit`, or even Microsoft Word, or Open Office Writer, if you just want to take a quick look into such a file it is generally much faster to use shell commands to do so.

One command to achieve this is `cat` (probably abbreviated from "concatenate and print"). It takes one or several files as arguments, and it simply prints everything that's in these files into your terminal – one file after the other:

```
> cat <FILE1> <FILE2> … <FILEN>
```

An interesting option to `cat` is the -n option, which will cause the lines in the output to be numbered (however if you supply multiple files, it will not reset the counter to 1 for each new file!).

The `cat` command is often useful; however, if files are long, one can easily get overwhelmed. In this case, the `less` command is helpful because it's true to its name and gives less output than `cat`: it displays the contents page by page, and it is possible to browse through the pages. It also offers a search function to find specific strings and patterns in the output. It can also take one or several files as arguments.

```
> less <FILE1> <FILE2> … <FILEN>
```

To try it out, we'll need a longer text file. You can find one here:

```
/mount/studenten/MethodsCL/2021/Linux/CasparHauser.txt
```

Copy it to your working directory (as an excercise for repeating the command for copying) and look at it using `less`. While less is running, the following commands are useful.

Moving around	
<space>	page forward
b	page backward
<return> or <arrow key down>	line forward
<arrow key up>	line backward

Searching	
/hallo<return>	search forward for string "hallo" (also works for regular expressions)
n	search for next occurrence of the string
N	search for preceding occurrence of the string
Switching between files	
:n	inspect next file (if several files were given)
:p	inspect preceding file (if several files were given)
Quitting	
q	quit

We can also use `less` to look at several files, for instance at the .txt files of female speakers in the TIMIT database (you've seen that database in the last two exercises):

```
> less /resources/speech/corpora/TIMIT-1/timit/train/dr1/f*/*.txt
```

If you run this command, you will mostly need :n and :p to go from one file to the next and back, as well as q for quit.

Character encodings

Depending on your configuration, you may have observed for the CasparHauser.txt file that some characters (the German Umlauts for instance) are not displayed properly. There are also some quote symbols that may be shown incorrectly.

If this happens, it's because the above file has its characters encoded in UTF-8 (which allows for a wider range of internationally used special characters by using more bytes for storing single characters). Formerly, in the Western hemisphere, ISO-8859-1 (sometimes called Latin 1) was widespread, and it could also encode Umlauts. It did so in a different way though, and that's why you need to know which encoding is actually used.

You will be able to see the Umlauts correctly only if you have configured both your local terminal emulator and your remote shell to use the correct encoding. In case of CasparHauser.txt, the encoding is UTF-8. If you have set the language of your shell as recommeded above:

```
export LANG=en_US.UTF-8
```

then you have specified that your shell expects UTF-8. If you still don't see the Umlauts in the file, then possibly your local Terminal emulator thinks it's displaying something other than UTF-8. In this case, you will need to find

where this can be set. Here's some hints - hopefully you don't need any of them anyway:

- In the Terminal application on Mac OS X, the encoding is hidden in the Preferences dialog: Go to "Profiles" (NOT to "Encodings"). In "Profiles", click on the "Advanced" tab and then select Unicode (UTF-8) under Text Encoding (at the bottom).

- For PuTTY under Windows, the encoding is set in the Category "Window", and there in "Translation". You have to specify the remote character set there. Make sure UTF-8 is selected.

- For the PowerShell, I honestly don't know - my impression is that it is invariably expecting UTF-8 anyway.

- Under Linux, it depends on which Terminal emulator you are using. The settings dialog could be somewhere in the main menu at the top of your desktop, or there could be a settings/properties dialog integrated in the window in which the Terminal appears. You might also get it by right-clicking on your Terminal.

A second problem related to the encoding that you might observe (I have this problem when logging into IMS from a Mac OS X terminal) is that even if you see special characters properly in your Terminal, you might not be able to type them when typing a search string while using `less`. I don't have a solution for this, but I hope that most of you will not have this problem.

The art of input and output redirection

Standard in, standard error, and standard out

Standard in, standard error, and standard out are "connections" used by programs for reading data, for giving error messages, and for giving back results.

Standard in, or **stdin** for short, is usually text or information that you type into the shell using your keyboard. Standard error (**stderr**) and standard out (**stdout**) are not always easy to distinguish since both connections are typically sent to your Terminal window (but you will see that they behave differently later).

Not all programs use all three "connections"; for instance, the `cp` command does not get its input from standard in; instead, it always reads directly from a file, and it doesn't write to standard out but to another file. However, in case of errors, it does write that error to standard out (such as "No such file or directory" etc.). The `ls` command on the other hand does write to standard out: the information on files provided by `ls` was always simply written into the Terminal window for us. And in case of error messages, `ls` also writes to stderr instead of stdout, just like `cp`. However `ls` also does not take input from stdin.

An example of a command that can read both from files and from standard in would be the `cat` command. In the examples of `cat` we have seen so far, we have always specified an input file as an argument. However, if you just invoke the command without any file arguments, as in

```
> cat
```

you will see that your shell doesn't display the prompt, as usual; instead it is waiting for input. You can provide input by typing something, then hitting the Enter key (↵). You will see that `cat` simply sends what you have typed to standard out (line by line if several lines). If you want to stop, use the <ctrl>-d key combination to terminate the input. Try it out! Similarly, `cat` with a file as an argument just sends the contents of the file to standard out – we have seen that earlier.

Redirecting standard out

Sometimes it is useful to save the output of a program in a new file. For instance, if we use the `-n` option to `cat` to insert line numbers in some file, we might want to keep this numbered version for the future. Or we might want to list all .txt files of female speakers in the TIMIT database and save it in a file. Or get the contents of all .txt files that we have in the TIMIT database into one file.

To achieve this, we simply use the ">" symbol for re-directing anything that the program writes to standard out into a file. This means the output will not appear in our Terminal window but will instead be redirected and written into the file we specify, as in these examples:

```
> cat -n /mount/studenten/MethodsCL.2021/Linux/CasparHauser.txt >
CasparHauser.LineNumbers.txt
> ls /resources/speech/corpora/TIMIT-1/timit/train/dr1/f*/*.txt >
timit.female.text.files.list
> cat /resources/speech/corpora/TIMIT-1/timit/train/dr1/*/*.txt >
timit.texts.list
```

Beware, you can easily overwrite files with these commands. By default, the files that you are redirecting to are either created (if they don't exist yet) or overwritten. Note however that some accounts might be configured in a way that the bash will refuse to overwrite existing files. The way to get this behavior is to set the noclobber option, which is a built-in protection mechanism in bash:

```
> set -o noclobber
```

This sets the noclobber option (clobbering means overwriting files by redirection). If the option is set, and you try to repeat the above command, which would then overwrite the CasparHauser.LineNumbers.txt, you get an error, and nothing happens:

```
 > cat -n /mount/studenten/MethodsCL.2021/Linux/CasparHauser.txt >
CasparHauser.LineNumbers.txt
bash: CasparHauser.LineNumbers.txt: cannot overwrite existing file
```

Try it out and run one of the above commands twice.

The command to switch this behavior off (i.e. to allow overwriting existing files by redirection) is

```
> set +o noclobber
```

So the +o switches the noclobber option off… Not very intuitive, I know. Here's a trick to remember which is which: -o is for no overwriting, +o is for overwriting.

If you don't want to set or unset `noclobber` for such cases, it's possible to specify explicitly that you want your redirection to possibly overwrite existing files, by appending a "|" sign to your ">" operator. So if you're more cautious, you might want to set `noclobber` so you won't overwrite files, and use these commands instead of the ones above in cases where you are sure you want to overwrite:

```
> cat -n /mount/studenten/MethodsCL.2021/Linux/CasparHauser.txt >|
CasparHauser.LineNumbers.txt
>  ls /resources/speech/corpora/TIMIT-1/timit/train/dr1/f*/*.txt >|
```

```
timit.female.text.files.list
```

Specifying ">|" instead of just ">" will overwrite existing files even if `noclobber` is set.

If you don't like the default behavior that you have in your account, you can again put the commands to set or unset `noclobber` in your personal configuration file, i.e. in .bashrc.

And another word of caution: if you try to redirect into a file that you are using as input, this will destroy the file. It's easy to try it out:

```
> echo hallo >| file
> cat file
hallo
> cat file >| file
> cat file
>
```

The sequence above first writes the word "hallo" into a file called "file", and then verifies that this was successful by executing `cat`. As you can see, this outputs "hallo", so the "hallo" was successfully written to the file. It then redirects the output of the `cat` command into the file. The next `cat` shows that the file is now empty…

Instead of overwriting files by redirecting standard out, it's also possible to append standard out to existing files:

```
> echo hallo1 >| file
> echo hallo2 >> file
```

These commands should first create a file, with content "hallo1", then append "hallo2" to that file. Check it out.

Redirecting stderr

The commands above did not redirect stderr. You can check this by the following sequence of commands:

```
> echo hallo >| testfile1
> ls testfile1 testfile2 >| output.txt
ls: cannot access 'testfile2': No such file or directory
```

This creates a file called testfile1 (with hallo in it, but that's not central here). Assuming that you do not have a file called testfile2 in your working directory, the following `ls` command should work only for testfile1, which you've just generated by the preceding command, i.e. it would write its name to standard out. However, for testfile2, you should get an error, and that should go to standard error.

In the example above, you can in fact only see the error, but not the result of the `ls` command on testfile1 (which listed its name) – and that's because that output went to standard out and was successfully written to a file called output.txt. This leaves only the message to standard error to be shown in your terminal. Check output.txt, the name of testfile1 should appear there.

More on redirection (only for advanced users)

If you want to redirect stderr, it maybe helps to know that the above operators for redirecting stdout can be more explicitly specified as redirecting everything that goes to "file descriptor 1" (which is stdout).

```
> echo hallo1 1>| output.txt
> echo hallo2 1>> output.txt
```

Now, standard error is equivalent to "file descriptor 2", and accordingly, assuming testfile1 exists, but testfile2 doesn't,

```
> ls testfile1 testfile2 2>| output.txt
testfile1
> cat output.txt
ls: cannot access 'testfile2': No such file or directory
```

leaves only the filename of testfile1 in your Terminal, and writes the error message to the file output.txt. This can be seen in the second command above: the `cat` command on output.txt shows the error message of the preceding `ls` command, which had successfully been written into output.txt. (Instead of overwriting the file called output.txt, we could also append to it using 2>> instead of 2>|)

So in this case the error message went to output.txt, while the name of testfile1 was listed in the terminal. If we want to have both stderr and stdout in one file, and we don't have `noclobber` set, we can do so by

```
>  ls testfile1 testfile2 &> output.txt
>  ls testfile1 testfile2 &>> output.txt
```

(Maybe it helps to think that & is an abbreviation for 1&2, so for both file descriptors.) However if we have set `noclobber` in a way that it refuses to overwrite existing files, we need a more complicated version in case we want to overwrite:

```
> ls testfile1 testfile2 >| output.txt 2>&1
```

This tells the shell to add stderr to stdout (2>&1) and to write stdout to the file output.txt (>|) even if the file exists. If you want to avoid this complex notation, use the much easier notation with only "&>", but unset the `noclobber` option, or make sure that no file of the name exists before you redirect into it. Sorry it's so complicated, not my fault... But as I said above, this is only intended for advanced users, so if this is overwhelming, just forget about it for

now. It can be helpful when you are running programs that take a long time to finish – you can then start them in the background, have them write all errors into a log file, and go home ;)

Pipes (for all users)

One of the coolest features in output redirection is that you can redirect output in a way that it is directly used as the input for the next command. We haven't seen many commands for which this is useful, but there will be many examples in the next section. Here, we'll illustrate pipes using the `ls` and `cat -n` commands, cooler stuff will be shown in the next section.

Assume you do not only want to list all files in your working directory, but you might also want to number them. We know that we can list files by `ls`, and that we can number lines in some input by `cat -n`. We can now glue the two together and send the output of `ls` as input to `cat` (we "pipe" the output of `ls` into `cat`). This works because `ls` writes to stdout and `cat` can read from stdin, and the | symbol causes the stdout of a command to be used as stdin for the next command. Try it out:

```
> ls | cat -n
```

We could have achieved the same by first writing the output of `ls` to a file, and then calling `cat -n` on that file:

```
> ls > list.of.files
> cat -n list.of.files
```

However the above is much quicker, and also saves us the effort of creating and later deleting a temporary file that we do not really need[10].

With what we've learned so far, we can not yet do super exciting stuff, so: sorry for the not very typical example in the footnote and also for the not very inspired exercises on the next page. We need a few more commands until we can see the potential of using pipes.

10 For advanced users: It is possible to use pipes even if the second command takes other arguments. So for instance, `cat` can take not only one argument, but arbitrarily many. If you want one of them to be the output from a preceding command, you need to tell `cat` where you want your input to go, so for instance

```
> echo "Content of list.of.files" > list.of.files
> ls | cat list.of.files -
```

This would first create a file that has "Content of list.of.files" as its first line. Then `ls` would list all files in that directory, and instead of displaying them in the terminal, the shell would pass them to the `cat` command. The `cat` command first outputs the contents of list.of.files (here, only the string "Content of list.of.files"), since that is its first argument, followed by the output of the earlier `ls` command, i.e. the files in that directory – since the position of the "-" in the arguments to `cat` indicates that `cat` should print this last.

Exercises

1. Give a command that would list all files in your working directory ending on ".txt" and write the output into a file called list.of.txt.files.

2. Use echo and output redirection to create a file called hello.txt which contains the string Hello.

3. Which command would move the file hello.txt from the previous question into a folder called Exercise, and write potential error messages into a file called move.log in your working directory? Check out if it works by running that command twice. Since the second time around, the file will already have been moved, this should cause an error in the second case, and this error should be written to your move.log.

Useful Linux commands for computational linguists

grep (and regular expressions)

One of the most useful commands IMO is the `grep` command. What it does is that it searches for lines containing certain strings in its input.

So an example of `grep` would be

```
> grep Apfel CasparHauser.txt
```

which, given the text example CasparHauser.txt introduced earlier, prints only those lines of the file which contain the string "Apfel". "Apfel", by the way, is German for "apple".

Anyway, you'll hopefully see that there are three occurrences of the word "Apfel", each in a separate line.

Now let's assume as a computational linguist you were interested in the occurrences of the *lemma* Apfel (i.e. the word in any form, singular or plural, and in any case (nominative, dative, etc.) – i.e., interested in any occurrence of either "Apfel" (nom., gen., dat., acc. singular) or "Äpfel" (nom., gen., acc. plural) or "Äpfeln" (dat. plural). This is where regular expressions come into play.

Regular expressions are a way to describe sets of strings. For instance,

```
[AÄ]pfeln\?
```

would describe all strings that start with either "A" or "Ä", then have the letters "pfel", and one ore zero occurrences of the letter "n". More generally, the square brackets indicate a set or range of possible letters, and the question mark with the backslash means: one or zero occurrences of the preceding letter.[11]

You can put this expression in quotes and give it as an argument to `grep`:

```
> grep "[AÄ]pfeln\?" CasparHauser.txt
```

Or you can find all occurrences of the word “go” in a version of Hamlet that I’ve put in the directory /mount/studenten/MethodsCL/2021/Linux:

```
> grep go /mount/studenten/MethodsCL/2021/Linux/Hamlet.txt
```

If you try this, you will see that you also find instances of “gone” or “good” etc.

11 Note that you can use egrep instead of grep, which has a slightly different syntax for some operators, and is more powerful. Here, we'll only cover the "normal" grep and basic regular expressions including extensions by the GNU implementation of grep. If you want more, google egrep.

By specifying a regular expression indicating that there should be whitespace (\s) before and after the go, we improve to only matches that really represent the word go:

```
> grep '\sgo\s' /mount/studenten/MethodsCL/2021/Linux/Hamlet.txt
```

However we now miss matches at the end of the line, where there is no character at all after "go", not even whitespace. So we can specify in the regular expression that we want either a whitespace or the end of the line after "go":

```
> grep '\sgo\(\s\|$\)' /mount/studenten/MethodsCL/2021/Linux/Hamlet.txt
```

Here's a list of meta-characters and operators when using `grep`. You should find everything we have used above and more in this list.

	Description	Example	Strings matching example
^	matches begin of line	^Der	Der (at begin of line)
$	matches end of line	Apfel$	Apfel (at end of line)
.	matches any character except end of line	.pfel[12]	Apfel, Ipfel, ypfel, #pfel, ...
*	repeat preceding token zero or more times	Äpfeln*	Äpfel, Äpfeln, Äpfelnn, Äpfelnnn, ...
\?	repeat preceding token zero or one times	Äpfeln\?	Äpfel, Äpfeln (and nothing else!)
\+	repeat preceding token one or more times	Äpfeln\+	Äpfeln, Äpfelnn, Äpfelnnn, ...
\(\)	brackets to group one or more characters to form a token	Sha\(la\)*	Sha, Sha la, Sha la la, Sha la la la, ...
[]	one of the characters in the brackets; ranges as in A-Z or A-S or 0-3 etc. are possible	[A-ZÄÖÜ]pfel	Apfel, Lpfel, Üpfel, ...
[^]	any character exept one of those specified in brackets; ranges are possible	[^ÄÖÜ]pfel	Apfel, Lpfel, Mpfel, ...
|	either the sequence of characters before the |,	doch|nicht	doch, nicht

12 Note that CasparHauser.txt is coded in UTF-8, and that Umlaut Ä is only interpreted as one character if you use the UTF-8 encoding in your Terminal. You can change (and query) the encoding of your terminal in the Terminal menu, under "Terminal". You can check the encoding of a file using "file -i", as in
> file -i CasparHauser.txt

	or the one after		
\s	some kind of space character	doch\snicht	doch nicht, doch<tab>nicht, …
\w	a letter character	Schoko\w*kuchen	Schokokuchen, Schokokirschkuchen, Schokonusskuchen, ...
\W	all non-letter characters (does not include numbers!)	Text\W*	Text, Text , Text., Text!!!, Text 123, ...
\	escape the special meaning of the metacharacters	100\$	100$ (at any position in the line, not necessarily the line end)

Finally, a very useful option to `grep` is `-v`. I'm not sure what the "v" stands for, but in any case `-v` **eliminates** all lines matching the pattern and displays only all others, instead of displaying only those that match. So the `-v` gives us just the opposite behavior – instead of listing all occurrences of the expression, it lists only lines where the expression does not occur. To eliminate all lines that contain digits numbers, we could use

```
> grep -v "[0-9]" CasparHauser.txt
```

sed

`sed` (short for "stream editor") is a really powerful command which can do all sorts of things. In this tutorial, we will only learn how to use it for substituting strings and regular expressions in its input.

The general syntax for substituting strings or expressions with `sed` is

```
> sed 's/EXPRESSION/REPLACEMENT/' <FILE>
```

or

```
> sed 's/EXPRESSION/REPLACEMENT/g' <FILE>
```

The stuff to be substituted is put between three identical characters – in this case, "/", but we could use any other character that does not appear in the expressions to be substituted. Before specifying these expressions, we put an "s" to indicate that we want to substitute. The two examples above differ in whether they have a "g" at the end of the substitution pattern. The first call replaces only the first occurrence in each line; the second call "greedily" (thus: g) replaces all occurrences.

Here's a more concrete example. We could use `sed` to replace all occurrences of

Äpfel or Apfel or Äpfeln in the CasparHauser.txt file by FRUIT:

```
sed 's/[ÄA]pfeln\?/FRUIT/g' CasparHauser.txt
```

By the way, if you would like to inspect the result without having to scroll for ages, you can pipe the output into a `less` command:

```
sed 's/[ÄA]pfeln\?/FRUIT/g' CasparHauser.txt | less
```

And then possibly use the search function in `less` to search for the first occurrence of FRUIT (i.e. type "/FRUIT").

A second useful application would be to use `sed` to tokenize a text (i.e., to separate the words at whitespace into a sequence of words, one per line).

```
> sed 's/\s\+/\n/g' CasparHauser.txt
```

Here, and elsewhere in the shell, "\n" is the new line symbol. The "\s" can stand for any space character (see regular expression chart above). So the above command replaces any sequence of one or more whitespace characters by a newline symbol.

If you check the output of the above command, you will see that it produces a lot of empty lines, in cases where there were several lines containing whitespace characters in a row. We could now pipe the output into a `grep` command and `grep` for lines that actually contain at least one character before the end of the line, which will leave only non-empty lines:

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "."
```

sort

Assume we want to do some statistics over the text in CasparHauser.txt. A useful command is to first sort all words in it alphabetically. To this end, we can use the `sort` command. Its syntax is

```
> sort <ONE OR SEVERAL FILES>
```

It sorts all lines in its input alphanumerically. The input can come either from one or several files, as above, or from standard in, for instance from a pipe.

If you use the -n option, it will sort numerically, and if you us the -r option, it will reverse the sorting order. Thus,

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "." | sort -r
```

will sort all words in CasparHauser.txt alphanumerically in descending order, while

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "^[0-9]" | sort -nr
```

will sort all tokens that start with a number numerically in descending order.

If you look at all words in CasparHauser.txt sorted alphanumerically, as in

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "." | sort
```

you'll notice that there are many identical lines that occur multiple times. If you use option -u (for "unique"), `sort` will print only one of each identical lines. However, if you want to know how many identical lines there were, we'll need to look at yet another simple command called `uniq` to this end.

uniq

uniq simply takes its input (from a file or from stdin) and keeps only one of several subsequent identical lines. Note that the identical lines have to be subsequent, i.e. if you have input like this

a
b
a
b
a

nothing will happen, because there are no two identical lines following each other. Thus it is often more interesting to use `uniq` on already sorted input, such as the output of the commands above:

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "." | sort | uniq
```

This should produce the same output as the `sort -u` above. However, `uniq` provides an option -c (for "count") which prints out the number of identical subsequent lines, resulting in output like this (I'm showing just the first few lines, the lines for words that aren't numbers are further down in the output):

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "." | sort | uniq -c
 1 &
15 *
 6 ***
 2 *****
 7 -
14 →
 2 …
 2 ]
–
 68 –
 1 ($1
```

This indicates that there were 7 lines with only a "-", 14 with a "->", 2 with "...", etc. Not very nice, I know, because we have left all these punctuation

mark symbols that were in the text. But you should be able to figure out how to delete these symbols using `sed` and \W (You'll be asked to do this in the Exercises section.)

wc

Finally, if you would like to know how many word tokens there were in CasparHauser.txt, you can use the `wc` command. `wc` counts the number of lines, words, and bytes in its input:

```
> wc CasparHauser.txt
 15941 134785 897580 CasparHauser.txt
```

The first number is the number of lines, the second the number of words, the third the number of bytes. To get only the number of lines, you can provide the -l option

```
> wc -l CasparHauser.txt
15941
```

The number of words is defined pretty much the way we've done it: it assumes that words are sequences of non-whitespace symbols which are delimited by whitespace symbols. Thus when we "manually" separate the words into lines at sequences of whitespace, using the sed command from above, and then count the lines, we get the same number as above, 134785:

```
> sed 's/\s\+/\n/g' CasparHauser.txt | grep "." | wc -l
134785
```

However, we could decide to "clean up" a bit first, and eliminate lines that don't look like reasonable text using `grep`, and then count again. This would of course give us a better estimate of the length of the text in words.

Exercises

1. Give a regular expression to be used by grep that describes the following sets of strings

   a) ab, abab, ababab, abababab, …

   b) aba, ababa, abababa, …

   c) a "gibberish" sentence, which looks like a text from some unknown language, with spaces and punctuation but no numbers or other symbols (something like: Ljsda jgj, qewlu blaj. – be inventive…)

2. Give examples of strings that match

a) [A-ZÄÖÜ][a-zäöüß]\+

b) [A-Za-z]*

3. Give a command that would find all lines in file CasparHauser.txt that contain Apfel either before a whitespace or before the end of the line.

4. Specify a command that would replace all occurrences of "king" by "queen" in the file /mount/studenten/MethodsCL/2021/Linux/Hamlet.txt.

5. Specify a command that would count the lines in the above Hamlet.txt.

6. Which combination of commands would give you the number of occurrences of the word go (i.e. not good!) in Hamlet.txt?

7. Above, we used the command

   sed 's/\s\+/\n/g' CasparHauser.txt | grep "." | sort | uniq -c

   to count the number of each word token in CasparHauser.txt. What would we have to add to delete all non-letter-characters using `sed` before sorting? What would we have to add if we then want to sort by the number of occurrences – i.e., have the most frequent tokens listed last? What is the most frequent word in CasparHauser.txt?

8. What ist the most frequent word in Hamlet.txt?

Using your home directory

Once you start using your IMS account, you might accumulate data there. However, there is a limit of how much data you are allowed to store. We'll briefly cover how you can check how much space you have left and what you can do if space gets tight.

The quota

Your quota determines how much data you can store in your home directory. To see it, type

```
> quota
```

You will see something like this:

```
Disk quotas for user schweitt (uid 1209):
     Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
schwarzmilan:/fs/users7
                  1256  4000000 6000000              38  150000  200000
```

This tells us that user `schweitt` has her home on server schwarzmilan in users7 (and that the device on which it is physically stored is called /fs/users7 there), that she is currently using 1256 kB of memory (see column "block"), while her quota is 4.000.000 kB, i.e. 4 GB (column "quota"). When she is over this quota, she will get a warning, but can still write more data onto the disk for some time, up to her absolute limit of 6 GB (see column "limit"). In this case she'd see in the output of the command above how long her grace period is: this is the time she is given to reduce the amount of data to be below her quota. Consequences of not being able to write to the disk after exceeding the quota for too long or after reaching the absolute limit may include not being able to log onto the computer using the window manager – in this case you can log in using the console or ssh and remove files to get into the accepted limits.

In addition to the quota on space, there is also a quota on number of files, and this is listed in the three next columns: `schweitt` has only 38 files in her home, while the limit is 150.000, with a hard limit at 200.000.

Disk usage by files: the du command

So once we know that we are using too much space, what can we do? First, it's not always easy to find out what exactly is taking that much space, and that's often because hidden files take up space, or files deeply hidden in some directory. A useful command in this case is the `du` command. In its default form, we just type

```
> du
```

and get a listing of all directories in the current working directory along with the size the take up. At the bottom we find the number for the working directory itself (i.e. the numbers for the subdirectories should sum up to that number). Since the output of this may get long, it's often helpful to use a pipe and sort by size like this:

```
> du | sort -n
```

This way you get the most problematic directories listed last. Then inspect those problematic directories (for instance, by changing into them and repeating the above command).

Once you have found unnecessary files or directories that lead to you being over quota, delete them. However, sometimes you won't want to delete, and we'll cover that in the next section.

Working with directories outside your home

One situation in which you will almost immediately go over quota is when you need to install so-called virtual environments for python. We will not cover in much detail what they are, I'll only say that for machine learning projects using specific advanced libraries in python, it's pretty tricky to come up with an environment in which all libraries are compatible with each other. Often you will need a specific version of a library to work with some other library, but that may not be compatible with the version that another library requires... This leads to the problem that you may need different combinations of library versions for different projects, and this is why each user ends up setting up their own environments for their own projects. This however means that each user is installing one or possibly many environments, and this may take up a lot of disk space. So this is one situation in which you easily go over quota, but you probably won't be able to simply delete environments that you are still using.

Another situation is that some programs (e.g. compilers) cache data for efficiency reasons, and the cached data are stored in hidden directories in your home that you might not be aware of. In this case you can configure the compiler to not cache anything and accept that recompiling will take longer.

In cases where you really need to keep the files, there is good news: Only files in your home directory and below count toward your quota, so what you can do is take care that your virtual environments and possibly your compiler caches are stored **outside** your home. The bad news is that you are usually not allowed to store data outside your home, so you will have to get permission from someone to store these elsewhere. For instance if you do work for some class, the class teacher might give write permissions below some designated directory such as

```
/mount/studenten/arbeitsdaten-studenten1/deeplearning
```

So ask your teacher if they can assign you some directory in which you are allowed to store data. Same if you are writing a thesis, ask your supervisor for disk space.

Please beware when working with big files: You might easily fill up a disk, and this is problematic, especially since you are sharing space with other users. Any file that you or someone else will try to safe after the disk is full will be empty. Even if it has been there and someone has just tried to make a tiny edit. So assume your friend has just finished a week-long programming assignment and only wants to insert last comment. But you have just downloaded something really big and now the disk is full. Your friend hits the <save> button after you have filled the disk and now their file cannot be saved any more and is therefore now empty and the week-long work is lost.

So two messages in this: check the space before you download big files. And don't do week-long projects without having a backup. Is the file really lost? Unfortunately it's definitely lost forever if it was somewhere below a directory with the string "arbeitsdaten" in it, as in the example above: per IMS conventions, this indicates that the directory is not in the nightly backup.

So how can you avoid this happening? Be aware that directories are distributed over different physical devices (disks). Use the following command to see how much space is left on the device on which some directory is stored:

```
> df <DIRECTORY>
```

This lists the device including space used and space available. Or try `df -h` to get a more human-readable output:

```
schweitt@phoenix:$ df -h /mount/studenten/arbeitsdaten-studenten1/deeplearning
Filesystem                          Size  Used Avail Use% Mounted on
/Dev/mapper/phoenix1-arbeitsdaten31  7.2T  6.6T  278G  97% /mount/arbeitsdaten31
```

You can see here that the `deeplearning` directory cited above is on a disk that's relatively full (at 97% capacity), although there is still available space (278 GB are not so bad...). For files in your home, you usually don't need to be concerned, thanks to having quotas established, they don't fill up so easily, and system administration is taking care that they don't reach their limits.

Also, in contrast to files below "arbeitsdaten", files in your home are backed up every night and can be restored by system administration. The same is true for most directories below `/mount/projekte`. Please use these directories with this in mind and don't put unnecessary huge files inside your home or into project directories that are in the backup. Especially if these files don't really need a backup (because it's just something you have downloaded from somewhere and that you could easily restore). This helps us to keep the backup process manageable (it's already close to the limits).

Getting Help/Learning more

If you want to learn more about a specific shell command, there is yet another shell command to do so, and that's the `man` command (short for: manual). So for instance, if you want to know more about the options that the `cp` command can take, you can look at the manual page for `cp` by the `man` command:

```
> man cp
```

This will display an (at first intimidating!) list of all arguments and options to `cp`, plus a description of all its options. Don't worry, most people using the shell are not familiar with all and every option, and it's also not a problem if you only understand a fraction of them. You'll still be able to get at least an idea of what is possible. Check it out some time.