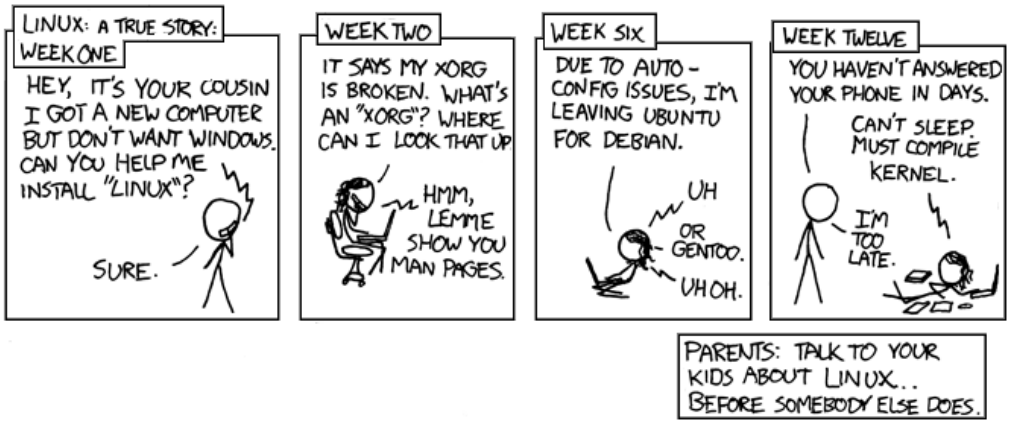
UNIX Workshop

Dimitri Robl



Version 0.1 September 23, 2018

Welcome to this three-day workshop which will teach you the basics of UNIX, (GNU/)Linux, and the command line! There is a lot to learn ahead of us, but first, you need to set up your working environment.

Contents

1 Installing a Virtual Machine

In order to make life easier for you, it won't be necessary that you install a Linux distribution or distro[1] on your harddrive. Instead, we will guide you through the installation of a virtual machine[2] running Debian GNU/Linux. If you already have a working installation of any Linux distro (either physical or virtual) you can skip this step.

1.1 Prerequisites

In order to use a virtual machine (VM), your physical machine should have at least two cores, 4GiB of main memory (i.e. RAM) and there should be at least 20GB free space on your disk. In this case more is always good.

1.1.1 Enable Virtualization

The first thing to check is whether you CPU supports virtualization, and has it activated. Follow these steps:

1. Boot your computer.
2. Depending on your motherboard press `ESC`, `DEL`, `ENTER`, `F2`, `F10`, or `F12`.
3. Look for an option which says "Advanced" or "CPU settings".
4. Depending on whether you use an Intel or an AMD CPU you will have to enable Intel Virtualization Technology, Intel VT, VT-x, AMD-V, AMD Virtualization or similar.
5. Check if this option is enabled and if not, enable it.
6. Save changes and Exit.

1.1.2 Install Oracle VirtualBox

Now you have to install the software to run virtual machines:

1. Download the installer from https://www.virtualbox.org/. It runs on Windows, OS X, Linux and Solaris hosts.
2. Start the installer and follow the instructions.

[1] You'll learn what exactly that means later in the course.
[2] If you are unfamiliar with this concept you can read the Wikipedia entry about it.

1.1.3 Getting the Installation Image

The final thing to obtain before the installation can begin is an image to install. Just download the netinst Debian image and store it somewhere you remember(!), you'll need it afterwards.

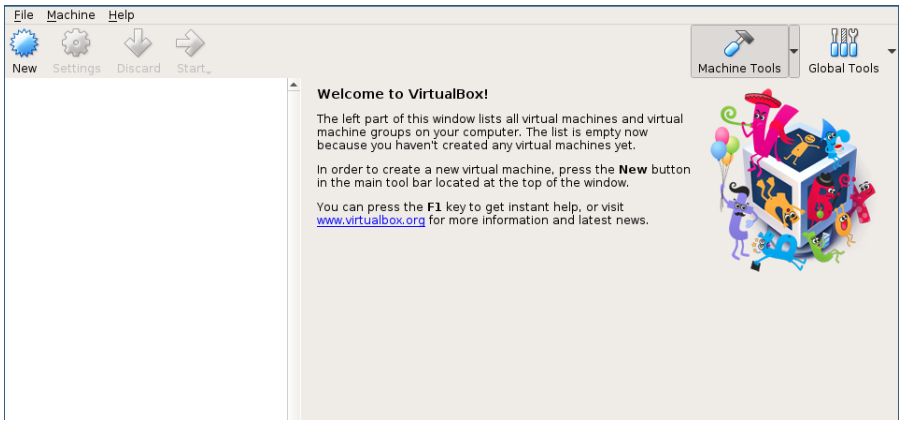
In case you don't have an x86_64 architecture [3], installation files for other architectures are available at https://www.debian.org/distrib/netinst. However, it is rather unlikely that you have a different architecture on a general purpose computer, unless you are still on a machine running 32-bit x86[4] or own one manufactured by Apple before 2006 which has a PowerPC CPU.

Now everything is ready to proceed to the installation!

1.2 Installing a Virtual Machine Running Debian GNU/Linux

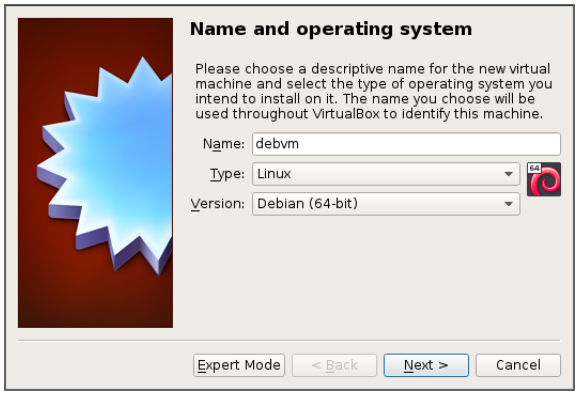
To install a VM running Debian GNU/Linux follow the instructions below:

1. Open VirtualBox.
2. Click on the "New" button in the upper left corner.

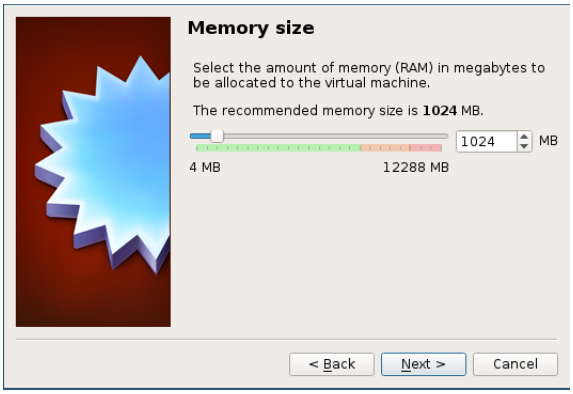


3. Enter a name for your VM, and if this is not automatically done select *Type:* → *Linux*, and *Version:* → *Debian (64-bit)*.[5] Press "Next".

[3] To make things less confusing, this architecture is also called x64, AMD64, and Intel 64.
[4] A.k.a. IA-32, i386.
[5] If you use a different rchitecture than x86_64 you have to specify this here as well.



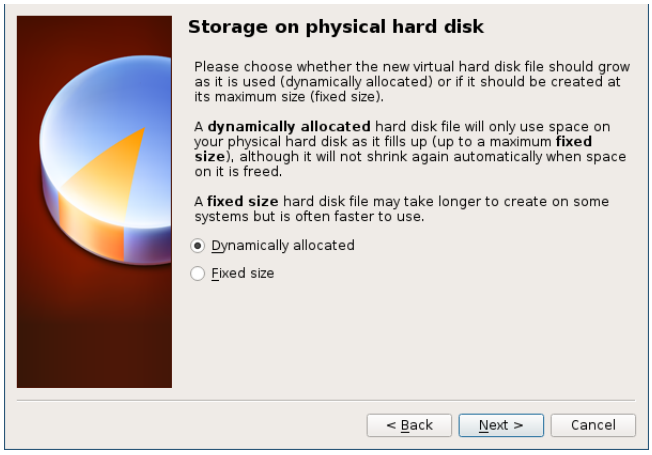
4. Select the amount of memory the VM will have. The best value here depends on what you want to do with the VM and how much RAM your physical machine has. The default value of 1024Mib (i.e. 1GiB) is sufficient for our purposes, but you can use more if you want and less if you are very low on physical RAM (i.e. 2GiB or less).



5. Choose how much disk space to give to your VM. This includes several steps:
   a) Select “Create a virtual hard disk now” and press “Create”.

   b) Leave “VDI (VirtualBox Disk image)” selected and press “Next”.

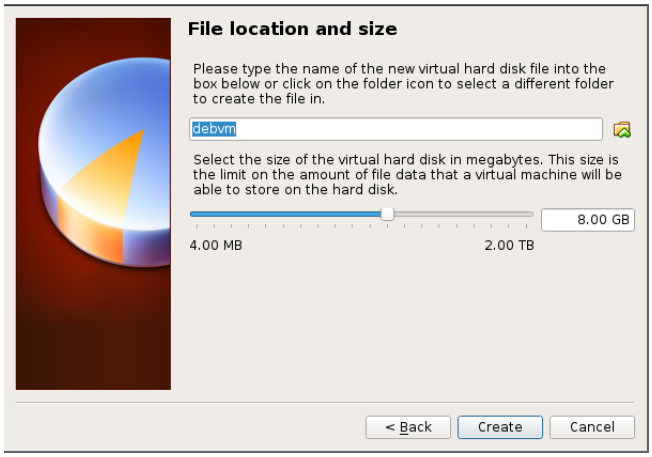
c) Again go with the default “Dynamically allocated” and and press “Next”.



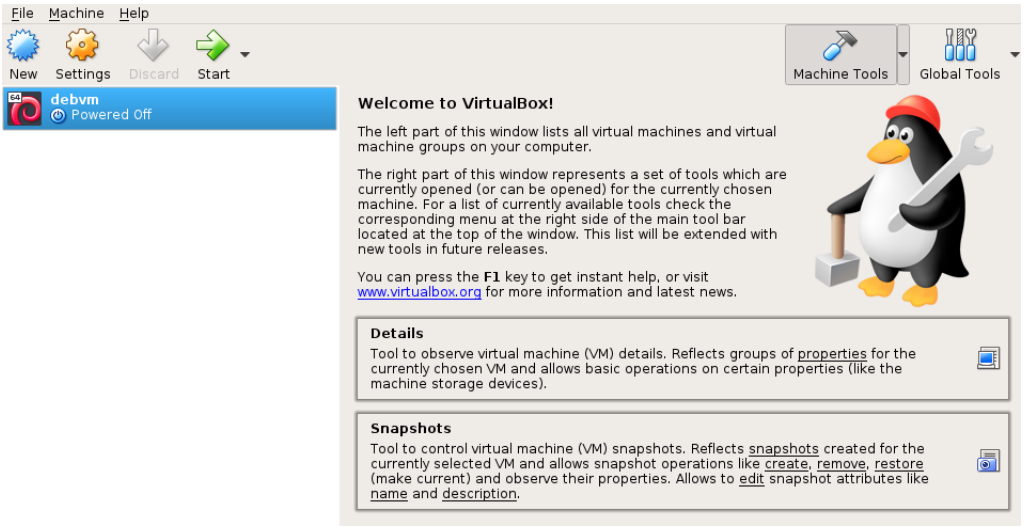
d) You can now change name and location of the file that will be the VM’s harddisk. If this sounds confusing: What the VM will believe is its disk will actually be nothing more than a file on the underlying operating system (OS) you run the VM on.

   A reasonable size also depends on your needs and the space you have. If you plan to really do anything with your VM, we recommend a minimum of 20GB, but if you just need it for this workshop the standard size of 8GB will suffice.

   After you are done, click on “Create”.



6. Now you're back to the main screen, which should look something like this:



Select you newly created VM and press "Start".

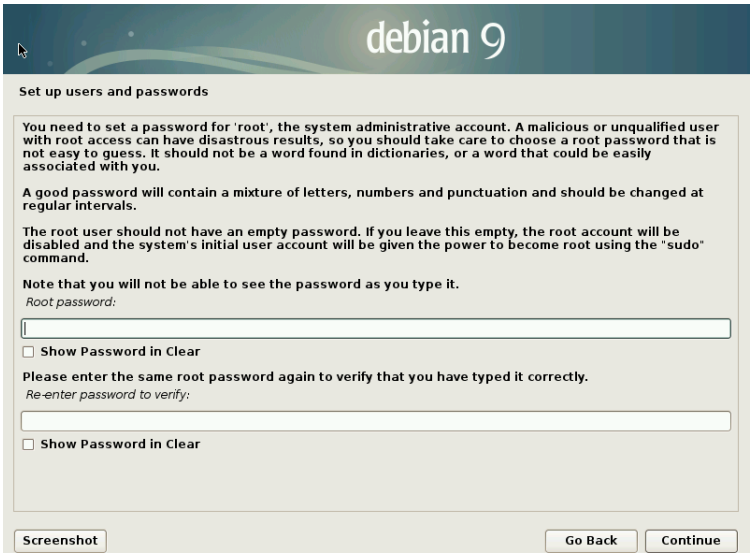
7. A pop-up should appear which asks you to select a start-up disk. Click on the folder icon and select the Debian GNU/Linux ISO-file you downloaded earlier (and whose download location you should remember ;)).

Press the “Start” button.

That’s it, a new window should pop up and the installation of your new virtual OS can begin. Select “Graphical Install” and follow the on-screen instructions. There are some things you should be careful about:

1. The selection of your locale (i.e. the language the system will use) and your keyboard layout – e.g. if you want to use an English system, but a German keyboard you have to tell the installer.

2. When asked to enter the `root` password, leave it blank to disable the `root` account and use `sudo` instead. We’ll discuss the details later.



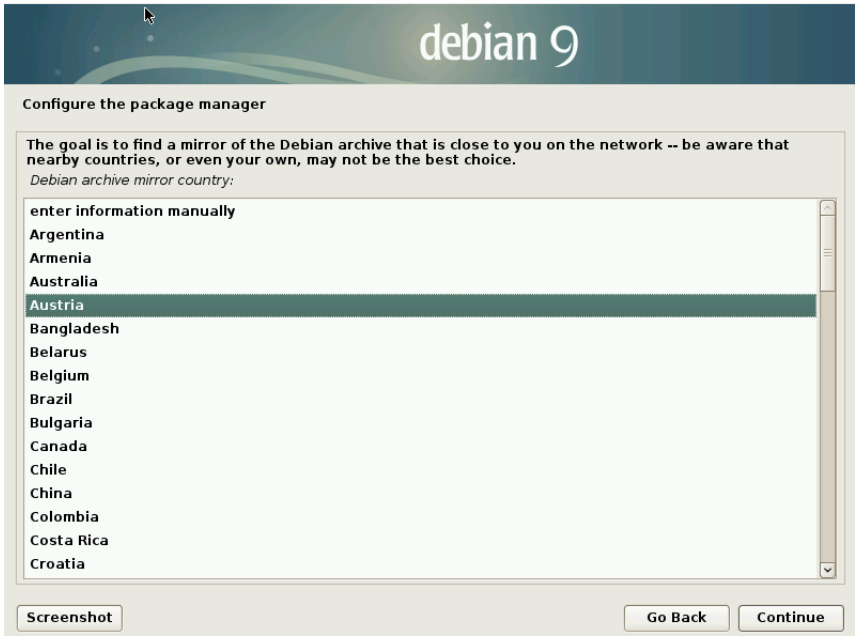
3. If you don’t know about disk partitioning leave the defaults intact. However, in case you install Debian GNU/Linux on real hardware on a portable device (e.g. a

laptop) and not as a VM, we *strongly* recommend using the "Guided - use entire disk and set up encrypted LVM" option. This will encrypt your harddrive which means that every time your PC or laptop starts you will have to enter a passphrase to decrypt the disk before the OS even boots. This has the huge advantage that in case your device is stolen, it will be almost impossible to access your data if the thief does not your password/passphrase – and no one should ever know that! Make sure to choose a strong password, where "strong" means "as long as possible". You don't necessarily have to care about special characters, length is always better. See here.

The only downside to this is the fact that if you forget or lose your password/passphrase you yourself also won't be able to access your data. This is one of the few cases in which it *may* be a good idea to write down your password/passphrase on a piece of paper and store it in some secure place.

Also, don't worry about the warnings when it says "Write changes to disk" if you set up a VM. The VM 'believes' it has a whole disk to write on and is not aware that it is actually just a file in the underlying OS. If you followed the instructions so far, it is not possible to destroy your real hard disk by installing the VM.

4. When you come to select mirror to download the packages, double-click on "Austria" and then select any mirror. As it says in the description `ftp.at.debian.org` is usually a good choice.



5. Once you come to the screen “Software selection” we recommend selecting “Xfce” or “LXDE” if the VM is on low memory ($<$ 2GiB), as these are very light weight GUIs. Otherwise you can stick with the default “Debian desktop environment”.

If you run into any problems, use your favorite search engine and look for something like “Debian installation tutorial”. Most likely you’ll find written instructions as well as videos on how to do it.

2 UNIX, GNU, and Linux

This section seeks to clarify terms you are very likely to encounter pretty soon when working with any (GNU/)Linux distro. Be prepared that people are *very* picky about words, and don't get discouraged in case you meet a particularly nasty partisan of one of these wordisms.

2.1 UNIX

In the beginning there was … the desire to play games! Yes, that's it! In the 1960ies two MIT students called Dennis Ritchie and Ken Thompson wanted to play the game "Space Travel" on their new, but underpowered minicomputer[1] PDP-7. To achieve their goal, they had to tweak the machine a bit and as they were researching operating systems (OS) at the time anyway, they ended up writing a whole new OS for the PDP-7 which came to be named UNIX.

UNIX, based on the even older Multics, introduced a lot of new concepts into OS-design, many of which are still in use today. UNIX was developed by Bell Labs which where prohibited from selling it under US-law, but in 1983 this decree was lifted and AT&T, the owner of Bell Labs promply tried to turn UNIX into a product, as it was already widely used among computer science students and companies they worked for. However, by this time there were already multiple versions of UNIX around, and one of the most important of them was developed by researchers in Berkeley; the Berkeley System Distribution (BSD).

BSD was sued by AT&T for copyright infringement despite the fact that large parts of AT&T's own UNIX implementation was based on code from Berkeley. After a long legal battle, BSD won its freedom and the base of modern FreeBSD, OpenBSD, NetBSD, etc. was laid out.

UNIX finally lost the battle for the PC against Apple and Microsoft, which was very likely at least in parts based on the law suits.

However, in the server market and in technical and scientific circles, UNIX and its derivatives continued to be state of the art and remain so today.

2.2 GNU

GNU (GNU's Not Unix) was founded in 1983 by Richard M. Stallman after he was fed up with the closed source philosophy of UNIX. He wanted to create an OS that is "free

[1] "Mini" here indicates that it didn't fill a whole room.

as in freedom", meaning it adheres to the four principles of free software, which state that users' have to be

0. able to run the program as they wish, for any purpose.

1. able to study how the program works and can change it according to their wishes.

2. able to redistribute copies to others, either free or for a fee.

3. able to distribute modified versions to others, again either for free or for a fee.

Many tools necessary for a complete OS were created by the GNU project, including `gcc` (GNU C Compiler), `g++` (GNU C++ Compiler), `gdb` (GNU Debugger), a shell, and others.

However, one essential part was incomplete: A working kernel. The kernel is the part of the OS which interacts directly with the hardware, so without a kernel, an OS can't run. Thus, the GNU tools were used on a number of UNIX systems, but a stand-alone OS was still wanting.

This changed when in 1991 a Finnish computer science student by the name of Linus Torvalds released a kernel he (eventually) called Linux.

2.3 Linux

Linus Torvalds originally released the Linux kernel under a license he created himself, but in 1992 he re-licensed it under the GPLv2 (General Public License, version 2) which is a license of the GNU project.

Technology-enthusiasts and academia quickly started using Linux as their kernel and as soon as it was GPL-ed there existed a full GNU OS with a working kernel. This is why you may read/hear the term GNU/Linux. Be careful here, some people are *very* picky about using Linux only when referring to the kernel and GNU/Linux when refering to a distro.

Linux' open source nature made it possible that its development proceeded very quickly with thousands of people contributing code and time. The current major release of Linux is version 4, the original 4.0 kernel was released in 2015. While Linux is often hailed as *the* example of a successful open source project with loads of people sacrificing their spare time to make the Linux kernel better, today this is obsolete in large parts. For the kernel releases between 4.8–4.13, 87.7% to 91.8% of the changes have been done by people who are paid for this by various companies.[2] This is not necessarily a bad thing, it just shows that the IT-industry heavily relies on Linux and therefore actively develops it and also that Linux today is very far from being a hobbyists' project but a mature OS kernel developed mostly by professionals.

[2] See here.

3 Basics

This section will introduce important concepts and words you'll have to work with, show you how to help yourself on a (GNU/)Linux system in case you get stuck and explain the filesystem.

To follow along, please download the training directory at
https://ct.cs.univie.ac.at/teaching/unixlinux-einfuehrungskurs/unixcourse.tgz
On the command line you can type (or just copy-paste – which can be done by just selecting text with your mouse to copy it and paste it by pressing the middle mouse button on most modern GUIs on (GNU/)Linux) the following to get the archive containing the folder:

```
wget https://ct.cs.univie.ac.at/teaching/unixlinux-einfuehrungskurs/unixcourse.tgz
tar -xvzf unixcourse.tgz
```

3.1 Shells, Terminals, and Commands

In the last chapter you've seen the concept of the kernel, the part of the OS which talks to the hardware. If you want to communicate with the kernel you need something, that wraps around it – a *shell*, for example.

Essentially, a shell is a user interface which allows you to access the services provided by the OS. You are most likely familiar with a graphical user interface (GUI) which allows you to access some services of the OS using a mouse and icons. A shell, on the other hand, requires text as input by providing you with a *command line*. While the GUI is often part of the OS you buy, i.e. you can't easily change the program that is your GUI if you use Microsoft Windows or Apple's OS X, a shell on a (GNU/)Linux system is just a program and you can choose from a wide variety of options. The shell we are going to work with here is the GNU `bash` (Bourne Again SHell), which is based on an older shell written by Stephen Bourne in 1979 and is obviously also a wordplay with the verb 'born'. We chose this one because it is currently the standard shell on all major (GNU/)Linux distros, but do play around with others if you like to.

The stuff you enter on the command line to your shell are *commands* and they work the same way as clicking an icon with your mouse does, e.g. you can either double-click on the Firefox icon, or just type the word "firefox" on the command line, followed by `ENTER`. Both actions will start a new instance of the Firefox webbrowser (if it is installed). One thing to keep in mind in the following sections is that most commands are actually stand-alone programs, even though they write their output to the terminal screen, while only a few of them are so-called *shell builtins*, i.e. keywords the shell itself understands and works on without starting a new program. This will be of spe-

cial importance in 3.3 where we talk about utilities that provide documentation for commands.

Many commands can take arguments and options. Their basic syntax is usually:

```
command [OPTION]... [ARGUMENT]...
```

The brackets around `OPTION` and `ARGUMENT` indicate that they are optional and the dots tell you that the preceding entity can be used multiple times, i.e. the line reads approximately as "Use `command` by typing `command`, possibly followed by one or more option(s), possibly followed by one or more argument(s)." Arguments tell the command e.g. on which file to operate, whereas options tell it what to do with the arguments. To separate these things, you have to put a space after the command, after each argument, and after each (group) of options.

Options come in two flavors: Short and long ones. Short options are usually indicated by a single, non-separated, preceding dash '`-`', while long options can be recognized by two non-separated preceding dashes '`--`'. A very simple example would be

```
$ firefox --new-window https://www.univie.ac.at
```

which, little surprisingly opens the homepage of the University of Vienna in a new window of the Firefox webbrowser.

In most cases options for a single dash consist of a single letter, while options preceeded by two dashes are words or phrases, e.g. let's say you use the `ls` program (to *list* directories and/or files) which has the options `-l`, `-a`, and `--human-readable` and want it to process argument `arg`. To do so, you could type:

```
$ ls -a -l --human-readable unixcourse
 contentdir          file3          onelinefile
 dir1                filetodelete   renamedfile
 dir2                grepfile       rmdirectory
 dir3                hugefile       rminteractive
'dir with spaces'    join1          sedfile
 errorfile           join2          sedfile.bak
 file1               linefile       sortfile
 file2               numericsort    tablefile
```

One thing to remember is that it is possible to combine short options behind a single dash, i.e. the following would be equivalent to the above:

```
$ ls -al --human-readable unixcourse
 contentdir          file3          onelinefile
 dir1                filetodelete   renamedfile
 dir2                grepfile       rmdirectory
 dir3                hugefile       rminteractive
'dir with spaces'    join1          sedfile
 errorfile           join2          sedfile.bak
 file1               linefile       sortfile
 file2               numericsort    tablefile
```

Long options cannot be combined, so you have to type each of them in full. Many programs have short and long options for the same operations and it's just a question of taste which one you use. A good approach is to start using long options because it is easier to remember what they do and start memorizing the short options for operations you use a lot.

But why bother to learn the names of programs if you can just click on an icon? Surely, that's much more intuitive and simple! Well, it depends. Imagine you commonly use about 50 programs, many of which interact with each other – a screen with 50 icons for programs seems rather overwhelming, don't you think? And how about moving stuff from one window to the other all the time – annoying, I'd say. Or opening the same 7 programs one after the other, hundreds of times and copying data from one to the other? Convenience is clearly not the word for this.

These are some of the tasks that can be solved easily on the command line. Most command line utilities follow the UNIX philosophy that each program should solve a single problem and solve it well. By combining the capabilities of many programs you get a very flexible toolset to solve your problem at hand. Make the easy things easy, and the hard things possible!

Figure 3.1: A DEC VT100 terminal. CC BY-SA ClickRick

Finally, a *terminal* is actually a piece of hardware (see 3.1), which can take input and display output. In this course and most likely in your whole life, you'll be confronted with terminal emulators, i.e. software which behaves like a terminal, like the on you see in 3.2.

When you open a terminal emulator your shell will be started and will provide you with a *prompt*, which usually looks a bit like

```
user@machine ~$
```

where `user` is your username, `machine` is the name of your computer and `~` is a shorthand for your home directory (see below for an explanation).[1] All your shell does now, is to wait for you to issue a command – let's do it the favor: type some random stuff and press `ENTER`. It should look something like this:

```
$ ajsdggasd
```

[1] From now on, we will abbreviate your prompt as just '$'.

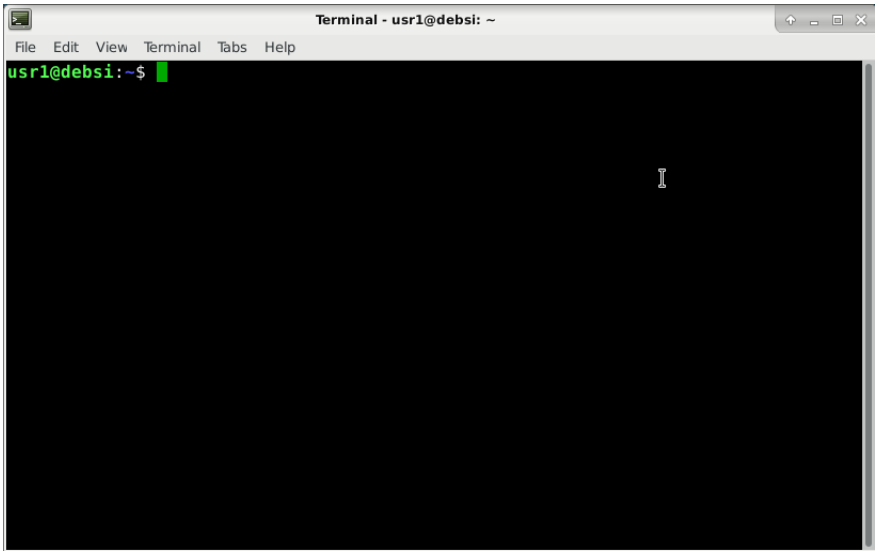


Figure 3.2: XFCE's terminal emulator

```
bash: ajsdggasd: command not found
$
```

Little surprisingly, this gibberish does not constitute a command. But fear not, we shall use existing commands starting in the next section!

3.2 The Filesystem

In order to use the filesystem efficiently it is essential to understand its basic structure and this is what this section wants to teach you.

The filesystem is software which manages how your files and directories are stored on you disk/pendrive/CD/etc. Usually there are at least three kinds of entities in a filesystem:

Files Some kind of data, possibly in a special format (e.g. PDF).

Directories These are places in which files are stored.

Links Links are references to either a file or a directory, i.e. they don't contain data, but just point to another place in the filesystem.

Filesystems in any UNIX system are usually structured as rooted trees.[2] The root of the filesystem is designated by the slash ' /'.

As you are logged in and opened a terminal already (if not, do so now), you can type

[2]For the mathematically interested: A rooted tree is an acyclic undirected graph in which one vertex has been designated the root.

```
$ pwd
/home/me
```

Of course, you'll get a slightly different output, but there are two important things here:

1. `pwd` is our first real command! It stands for 'print working directory' which prints you current working directory, i.e. tells you where in the filesystem you are.

2. The output, `/home/me` tells you, that you are in a directory `me` which is in a directory called `home`, which is in the root directory `/`.

Okay, now that you know where you are, you may want to know which files[3] are in this directory – you can list them using

```
$ ls
Documents Downloads Pictures Private
```

Again your output will differ slightly. So now, you used the program `ls` to show you the contents of the directory you are in. This is `ls`' standard behavior: If you don't provide any arguments it will display the content of you current working directory.

However, you can tell `ls` which directory it should list by providing a *path* as an argument. To show the contents of the root directory, type:

```
$ ls /
bin boot dev etc lib media mnt opt run sbin srv tmp usr var
```

The output provided here shows everything that is required in the root directory according to the Filesystem Hierarchy Standard (FHS), a document which defines what has to be in the root directory of any (GNU/)Linux distro.[4] But as it only defines the minimum, distros are free to add other things and you'll most likely also find at least `home`, `proc`, and `sys` in your root directory.

Now there is one more concept you have to understand regarding the filesystem: absolute and relative paths. In a rooted tree you can specify any point in the tree by starting at the root and listing every step on the way. This is what `pwd` does: Above it said that I was in `/home/me`, telling me my exact position in the filesystem in an absolute manner; therefore paths starting with the root directory are called *absolute paths*.

Unfortunately, absolute paths can be very inconvenient and long. Remember that we said `ls` can be provided with a path to list the contents of any directory in the filesystem, not just the one you are currently in and that in the above example the current directory contained `Documents`, `Downloads`, `Pictures`, and `Private`. Imagine you have a subdirectory in `Documents` which is called `university`. To list it, you could type

```
$ ls /home/me/Documents/university
```

[3] Here 'files' denotes data, directories, and links as it is the most generic concept.
[4] A short version of this can be found by typing `man 7 hier`.

An awful lot to type I'd say! To remedy this situation you can provide a path relative to your current working directory, i.e. a *relative path*. `pwd` told us that our current working directory is `/home/me`, so the relative path to `university` is just `Documents/university` without a leading '/'. Every path starting with a slash will be interpreted as an absolute path, while relative paths just start with the name of the next directory you hop into from your current directory.

That's it for filesystem theory, now we'll enable you to help yourself on a (GNU/)Linux system.

3.3 Help Utilities

Every (GNU/)Linux system comes with a lot of built-in documentation installed by default and of course the Internet hosts millions of HOW-TOs, tutorials, etc. We're going to focus on the offline documentation, but also provide some hints were you usually get reliable information online.

This section can be treated as a reference – reading it now in its entirety may cause you a bit of a headache as you may not be familiar enough with commands, options and arguments right now. For the beginning it will suffice if you skim through this section to get a basic idea about which facilities exist.

3.3.1 `man`

`man`, which is a shorthand for "manual page" is the traditional UNIX documentation and knowing how to use it will help you not only on (GNU/)Linux, but also on any other UNIX you'll encounter like the various BSDs. In case you ask someone who is in a hurry about a program and the only response is "RTFM", meaning "Read The Fucking[5] Manual", this program is usually what they want you to use.

If you want to access the manual page of the `ls` program, all you have to do is to type:

```
$ man ls
```

After this the man page for `ls` should appear. To scroll down press `SPACE` and to exit press `Q`.

We won't go into a lot of details here because your first task now is to type

```
$ man man
```

This will fire up the man page of `man`. Read it to learn how to use this program efficiently and make sure to read the "EXAMPLES" section!

The main things you should remember from this is how to move within the man pages, how to exit them, what sections are and how you can tell the program to only return entries from certain sections. Useful options you may want to remember are

[5] Or, "Fine", for the very pure.

`-k | --apropos` List the programs whose name or short description match the keyword you searched for. Note that this is the same thing the `apropos` program does.[6]

`-a | --all` Show all man pages from all sections which match the keyword you searched for.

If you read through `man`'s man page you'll understand what these options do ;)

3.3.2 `info`

While `man` is available on any UNIX system, the info pages were invented by the GNU project and are therefore not necessarily available. However, all widely used (GNU/)Linux distros have them installed by default.

To get the info page of `ls` type

```
$ info ls
```

You can again use `SPACE` to scroll forward and `Q` to get out of `info`.

Your next task is now to read

```
$ info info
```

As `info` is a bit more sophisticated than `man`, offering links, footnotes, menus, and other references it is essential to read this. Furthermore, `info` has a lot more keyboard shortcuts you can use, all of which are documented in its info page.

The difference between man pages and info pages is not a clearcut one. As a rule of thumb, man pages are often more concise and only useful if you already know what a program does in general, but just want to check available options, i.e. they are often very technical and offer few explanations.

The info pages, on the other hand, are often much more verbose, have sections for different kinds of options (while the man pages just list them one after the other) and at least try to be a bit more accessible to complete beginners.

Just compare some man pages and info pages for the same program, e.g. `man ls` vs. `info ls` to see the differences and get a feeling for them.

Note also, that all documentation is written by humans and some are more capable of explaining how stuff works than others. For this reason the quality of man pages and info pages can vary greatly from program to program. A good example can be found in this xkcd comic.

3.3.3 `help` and `type`

In 3.1 we mentioned that not every command you type is a full program, but some are just shell builtins. These are keywords understood by your shell. You can compare this with your GUI: Most icons you click on will start a new program, but some things are

[6] You guessed it: Use `man apropos` to find out more.

done by the GUI itself, e.g. if you click on the Windows button, it will display a list of programs you can start, but this list is just part of the GUI.

For these shell builtins no man pages or info pages exist. A command you'll use *very* often is `cd` ("change directory") which enables you to change your current working directory. But if you try to type

```
$ man cd
```

you'll get a man page titled "`BASH_BUILTINS(1)`" which will most likely be very confusing. To get help specifically for `cd` (or any other shell builtin for that matter) just type

```
$ help cd
```

and the help will be displayed on the terminal.

But, you may ask yourself, how am I supposed to know what is a program and what is a shell builtin?! Don't despair! This is where the `type` shell builtin comes into play. If you tried to access the man and/or info pages of a command and didn't get a satisfying result, try

```
$ type COMMAND_NAME
```

e.g.

```
$ type cd
cd is a shell builtin
```

And voilà, now you can be sure.

3.3.4 `--help` and `-h`

Sometimes you are just not sure about what was the right option to get what you want or you only forgot the correct spelling of the option. In these cases you may not want to read through the whole man or info page and most programs make this possible by providing a `--help` or `-h` option. So, typing `COMMAND --help` or `COMMAND -h` will in most cases print a short description of the command and its most common options to the screen.

Just try it out to see for yourself.

3.3.5 `whatis` and `whereis`

Many programs are continously developed and change from one release to another; functionality may be added, reduced, removed, deprecated, etc. And at some points you may not know whether you want the old or the new version. One way to solve this is to simply install both, use them and then decide which one you like more. On a GUI this may be indicated by two different icons for the different versions, but how can you tell the difference on the command line?

This is where the `which` and `whereis` utilities come in handy.

`which` tells you where in the filesystem the executable you are going to call is located and if you have installed two versions each of them will reside in a different place. Thus, if you type

```
$ which pwd
/bin/pwd
```

`which` tells you that if you call the `pwd` program, it will use the one in the `/bin` directory. So, in case you have a second version of `pwd` installed, which resides in `/usr/local/bin`, you'll have to use the absolute path to call it.

But sometimes you won't know if there are different versions of the same program on your machine, e.g. when you start working in a new place with your machine already set up for you. To find out your possibilities try

```
$ whereis COMMAND
```

which will show you where versions of `COMMAND` are installed, where their man pages are located and if the source code of the command is stored in a standard location on the system.

Attention: The following explanation will most likely make no sense at all if you haven't read about variables in 4.7. The actual difference between `whatis` and `whereis` is that the former looks through your `PATH` variable and returns the first executable it finds which actually is the one your shell would execute while the latter scans through your whole `PATH` as well as `MANPATH` plus some standard locations commonly used for programs on any (GNU/)Linux system, and returns everything it finds there, separated into executables, source files, and man pages.

3.3.6 The Internet Is Your Friend

If you have to do something real quick you may find yourself in a situation where you don't want to deal with man pages or info pages, but would prefer a ready-made command with all options you want already set. As it is very likely that other persons have had the problem at hand before you and many people post everything they achieve online, chances are that you can find the command of your dreams posted in some online forum, ready to be copy-pasted into your terminal emulator.

As you might expect, there are a myriad different sources where you can find information, but we will only deal with very few of them which offer reliable information in most cases.

Distros' Websites

Each distro has its website and often Wikis attached to them. Many have a lot of useful information, but keep in mind that these are distro-specific. It is also important to know how your distro is organized; in the case of Debian, you are faced with a community project where the whole content is provided by people spending their spare time on

providing information to others, but no one is responsible for doing so. This often leads to very helpful replies to questions, targeted at total beginners, but as no one is responsible for updating this content, sometimes you find very outdated information. Red Hat Enterprise Linux, on the other hand, is a distro aimed at corporations, providing a *lot* of information. However, as their material is intended for IT-professionals, you won't find a lot of explanation in many cases as they assume you already know what you are doing/want to do and just show you how you can do it with the tools Red Hat provides.

Thus, do a bit of research about each distro before using their documentation and how-tos.

Debian: Information about Debian can be found at:

- https://www.debian.org
- https://wiki.debian.org/
- https://debian-handbook.info/. This is the Debian Administrator's Handbook, the e-book version of which can be downloaded freely. However, you can also pay for the e-book version, and in case you start relying on it, it is definitely a good idea to give the authors something back for their hard work. The current (2018-09-12) version of the book if for Debien 8 Jessie, but the current Debian release is Debian 9 Stretch, so some information my be outdated.

Ubuntu: Ubuntu is an enterprise distribution with a desktop and a server version. They have a very active community as well which answers a lot of questions in the forums.

- https://www.ubuntu.com
- https://tutorials.ubuntu.com/. You can filter tutorials for different categories, e.g. server or desktop.
- https://help.ubuntu.com/
- https://ubuntuforums.org/
- https://askubuntu.com/ This is a searchable question and answer site.

Red Hat Enterprise Linux: As mentioned above, RHEL is an enterprise distribution and one of the most prominent examples showing how much money you can earn with free software. They have excellent, though rather technical guides for many topics, and some things can be only accessed if you have a support contract with Red Hat.

- https://www.redhat.com/ Their main page is very much like the website of other companies – a lot of bragging and ads, but it's still a good starting point to get a feeling about what Red Hat as a company is doing.

- https://access.redhat.com/ This is where you start out if you look for help. Browse a bit; they have a lot of documentation, guides, etc. and many things can be downloaded as PDF for free.

CentOS: CentOS, the "Community Enterprise OS" is the community version of RHEL. It is only rarely suited for home Desktop use and the software it ships is very outdated in many cases (as all the good stuff is reserved for the actual RHEL). However, if you want to get to know RHEL without paying for a license, CentOS is a good start and may be useful if you want to run small servers at home/in the cloud.

- https://centos.org/
- https://wiki.centos.org/. While this Wiki is still alive, it is updated rather sparsely and the current release CentOS 7 is not documented in any reasonable fashion. So much so, that https://wiki.centos.org/docs/ just tells you to visit the above mentioned https://access.redhat.com/.

Arch Linux: While Arch Linux is considered to be rather complicated and suited only for experienced users, it has a thriving community and a very well maintained Wiki with in-depth explanations for a lot of things. Highly recommended if you want to know something in detail.

- https://wiki.archlinux.org/

We'll stop here, but you can comb through all the other distros' websites as well, which is a very useful thing to do before you change distro.

If you are interested how many distros are out there and which ones are the most heavily used, you may find https://distrowatch.com/ (lists also non-Linux UNIX systems like the various BSDs) or https://static.lwn.net/Distributions/ useful.

Other Online Sources

However, there are also sites dedicated to Q&A, i.e. you can post a question and someone in the community replies with a hopefully correct answer. The ones you'll encounter over and over again are:

Stack Overflow: To be found at https://stackoverflow.com/. If you did some programming already or are going to do so, this site will also pop up many times. Whenever you research some problem you have with the command line it is almost certain that you are going to land on Stack Overflow after some time, regardless which search engine you use.

Stack Exchange for UNIX/Linux: Strictly speaking this is just a subsite of Stack Overflow and many topics specifical to (GNU/)Linux and/or UNIX in general will can be found here: https://unix.stackexchange.com/.

With these tools at your disposal you should be able to solve almost any problem your new shiny (GNU/)Linux system throws at you. **BUT**, beware! Not everything people post there is correct and if you are unlucky, someone may even put some malicious code in their reply, so **always** think before you copy-paste something onto your command line and press `ENTER`, especially if you have no idea what the command you want to use can do. Be even more careful if the line you copy starts with `sudo`. The good thing about UNIX systems is that you can do anything you like, but the bad thing is that you can do anything you type.

4 The Command-Line

Now, after so much theory, let's do something practical! The first thing to do is to start a terminal emulator if you haven't already done so.

Before we are going to start working with commands, we'll talk a bit about keyboard-shortcuts with which you can move quickly on a line. For this purpose just type a sentence on the command-line without pressing ENTER:

```
$ this is a beautiful test sentence
```

Your cursor will now be at the end of the line. If you made a typo somewhere in the line you can move there using the arrow keys, but this gets annoying pretty quickly as soon as the line gets long (your mouse won't work here). Thus, play around with the following shortcuts:

CTRL-A	Move to the beginning of the line.
CTRL-E	Move to the end of the line.
CTRL-K	Delete everything from your cursor to the end of the line.
CTRL-U	Delete everything from your cursor to the beginning of the line.
CTRL-T	Switch the sign currently below the cursor with the one before it and move the cursor one character to the right.
ALT-F	Move one word forward.
ALT-B	Move one word backwards.
ALT-T	Switch the word below the cursor with the one preceding it.

There are even more shortcuts, so look them up on your own if you're curious, but the ones above will suffice for the beginning. In case you don't see the use in some of them: That's ok, use cases will come ;)

And there is one more *vital* thing to know about moving aroung on the command-line: tab-completion. If you want to pass an argument to a program and you've already typed enough that the argument is uniquely identified, you can just hit TAB and the bash will automatically complete the argument. Why this is extremely useful will become clear once you start using commands and arguments, just keep this in mind. Additionally, if you press TAB twice, the bash will list all currently possible arguments. Try it out when following the instruction below!

4.1 Moving and Looking Around

Now, onwards to the command-line! For starters, let's find out where we currently are – you already know how to do this:

```
$ pwd
/home/me
```

Now, let's see what's in the directory we're currently in, which we can do with `ls(1)`, which "lists" directory entries:

```
$ ls
bin      documents  nikola    playground         unixcourse
Desktop  Downloads  personal  tor-browser_en-US
```

As you already know, you can pass options to `ls`. We'll start with the `-a | --all` option. This one shows you all files in the current directory:

```
$ ls -a
.               Downloads          .pki
..              .dvdcss            playground
.android        .F5Networks        .python_history
.ardentryst     .gconf             .ssh
.bash_history   .gnupg             tor-browser_en-US
.bash_logout    .gnuplot_history   .vim
.bash_profile   .gphoto            .viminfo
.bashrc         .ICEauthority      .vimrc
bin             .lesshst           .wget-hsts
.cache          .local             .Xauthority
.config         .mozilla           .xsession
Desktop         .neomutt           .xsession-errors
.dmrc           .xsession-errors.old
```

What?! Where did all these files come from?! Don't worry, they have been here all the time, but they are *hidden files*. You see that those files which have not been listed before, start with a dot '`.`' and this indicate that they are 'hidden'. This does not mean that you are not allowed to see them, but they are mostly configuration files which you just don't want to see every time you use `ls` as they just add a lot of useless output. For example, your browser will store your bookmarks, preferences, cookies, browsing history, etc. in a hidden directory. This way it does not confuse your profile with the one of another user who uses the same computer. Thus, a hidden file is just a file whose name starts with a dot, and programs like `ls` don't display them by default – nothing more to it. You can also create hidden files yourself if you start their name with a dot!

However, two hidden files are found in every directory: '`.`' and '`..`'. '`.`' is a pointer to the directory itself[1] and '`..`' is a pointer to the directory above this directory. This is a very convenient shortcut if you want to change to the directory above the one you are currently in, but you don't want to type the absolute path from the root directory. Changing directories? Great, we're already at the next command: `cd` which very surprisingly stands for 'change directory'. It takes as an argument a path (relative or absolute) and changes your current working directory:

[1] Why this is useful will become clear later.

```
$ cd Documents
$ pwd
/home/me/Documents
$ cd ..
$ pwd
/home/me
```

You can see here that we changed into the `Documents` directory and from there one directory "upwards" by using `..` as shortcut. Otherwise we would have had to type `cd /home/me` to change from `/home/me/Documents` to the `/home/me`.

Two more things regarding `cd`:

- As we've seen in 3.3.3, it is a shell-builtin, so if you want to know how it works, you have to type "`help cd`".
- If you just type `cd` on the command-line and press `ENTER` without providing an argument, `cd` will change to your home directory. The home directory is where all your personal stuff is stored and in the examples above, my home directory is `/home/me`. Thus, if you get lost in the filesystem or you are in a hurry and just want to go back to your home directory, type `cd`, press `ENTER` and you'll be right there.

Now that you've seen `cd`, let's go back to `ls`. Combining `ls`, `pwd`, and `cd` the filesystem is yours to explore and we'll let you do that in a minute, just one more option to `ls` and the associated concepts:

With the `-l` option `ls` provides you with "long listing":

```
$ ls -l
total 36
drwxr-xr-x  2 me me 4096 Aug 27 14:53 bin
drwxr-xr-x  2 me me 4096 Sep  4 17:49 Desktop
drwxr-xr-x 14 me me 4096 Sep  7 17:40 documents
drwxr-xr-x  2 me me 4096 Sep 10 21:43 Downloads
drwxr-xr-x  8 me me 4096 Aug 24 20:05 nikola
drwxr-xr-x  4 me me 4096 Aug 27 14:54 personal
drwxr-xr-x  3 me me 4096 Aug 26 20:43 playground
drwx------  3 me me 4096 Sep  9 14:36 tor-browser_en-US
drwxr-xr-x  2 me me 4096 Sep 17 11:48 unixcourse
```

As you can see there is a lot more information about each file. Actually, the output is now separated into fields, separated by a space. These fields are from left to right:

Permissions: See below, that's the new concept we're going to explain ;)

Hard Links: The number of hard links which point to this file. If you are interested in links, read it up, we won't cover them here as you don't need them for now.

Owner: The user to whom this file belongs.

Group: The group to which this file belongs.

Size: The size of the file in bytes.

Modification Time: The month, day and time give you the exact time when the file was last modified. If it was not modified in the current year, the year of the last modification is given as well.

Name: The name of the file.

Each file has an user owner which is usually the user who created the file in the first place as well as a group owner, which is usually one of the groups the user owner belongs to.[2] But what does that mean?

Well, it has to do with the first field which consists of a rather bewildering number of letters and dashes at first sight. However, it is actually a very simple system and tells you what you can do with a file:

The first character tells you *what kind of file* this is, e.g. `d` indicates a directory and - indicates a regular file. There are others as well, but these are the most common and important ones – if you want to know more look at the info page of `ls(1)`.

The next nine characters show the permissions of the file which in very short means who is allowed to do what with a file. These nine characters are actually 3 groups of 3 permission sets: The first three are the permissions of the owner[3], the second three are the permissions for the group, and the third permission set tells you what anybody else (often called "world" or "other") is allowed to do.

The reading of permissions is pretty easy: If there is a letter, the permission is present, if there is a dash it is absent. The postitions in each permission group from left to right tell you about:

`r` "Read permission". If this permission is set, it is allowed to read the file.

`w` "Write permission". If this permission is set, it is allowed to write something to the file, i.e. change it.

`x` "Execute permission". If this permission is set, it is allowd to execute this file – this is typically set for all programs, like `ls(1)`.

With this information at hand, can you decode this output:

```
-rwxr-x--x 1 nina hackers 2048 Jan 14 14:21 amazing-script
```

Try it, write down your solution and afterwards read the following explanation.

This is a regular file, which can be executed by the owner, the group, and everyone else, can be read by the owner and the group and written only by the owner. It belongs

[2] From now on we will just speak of "owner" to mean "user owner" and of "group" to mean "group owner".

[3] Yes, even owners are not necessarily allowed to do everything with the file!

to user `nina` and to the group `hackers`. It is 2048 bytes in size (i.e. 2 KB), was last modified on January 14th at 14:21, and is called "`amazing-script`".

As you've seen in 3.2, you can also pass absolute and relative paths as arguments to `ls` to tell it what to list. There is one thing you should be aware in this case: If the argument is a regular file, `ls` will list just the file, but if the argument is a directory, `ls` will show you the content of the directory:

```
$ ls file1
file1
$ ls contentdir
file_00  file_05  file_10  file_15   subdir04  subdir09
file_01  file_06  file_11  subdir00  subdir05  subdir10
file_02  file_07  file_12  subdir01  subdir06
file_03  file_08  file_13  subdir02  subdir07
file_04  file_09  file_14  subdir03  subdir08
```

Great, now you are ready for your tour through the filesystem! Use `ls`, `cd`, and `pwd` to look around – you are on a free software OS, there are no secrets (except if you don't have permission to look at something, of course ;)).

4.2 Working with Files

Up to this point we've just looked at files, examined their properties and moved around in the filesystem tree. This is all nice and fine, but maybe we want to change files' properties, or add stuff to or remove something from the filesystem. This is what this section is about. And for now, change in the `unixcourse` directory you've downloaded at the beginning of 3 to be able to follow all the examples below.

One of the things you may have already wondered about is: What if you want to have a directory for all of your lovely cat GIFs – how can you create such a directory? Well, the answer to this is `mkdir(1)` "make directory". In its simplest form it just takes a single argument: The name of the directory you want to create. Note, that this can be a relative or absolute path:

```
$ ls
 contentdir          file3          onelinefile
 dir1                filetodelete   renamedfile
 dir2                grepfile       rmdirectory
 dir3                hugefile       rminteractive
'dir with spaces'    join1          sedfile
 errorfile           join2          sedfile.bak
 file1               linefile       sortfile
 file2               numericsort    tablefile
$ mkdir newdir
$ ls
```

```
 contentdir          filetodelete   renamedfile
 dir1                grepfile       rmdirectory
 dir2                hugefile       rminteractive
 dir3                join1          sedfile
'dir with spaces'    join2          sedfile.bak
 errorfile           linefile       sortfile
 file1               newdir         tablefile
 file2               numericsort
 file3               onelinefile
$ cd newdir
$ pwd
/home/me/unixcourse/newdir
```

One thing about names: Whether you create directories as we do now, or other files which we'll do later, there are certain rules you should adhere to, to make your life easier: A file name (which includes directory names!) should consist of only letters of the English alphabet, underscores '_', dashes '-', and digits, and should always start with an underscore or a letter. Don't use special characters like ä, ß, ж, €, etc. and don't use spaces, tabs, newlines, etc.! This is just conventional, so you *can* use whatever you want, but it may make your life very inconvenient. In case you encounter filenames containing spaces, you will have to quote the name or escape the spaces using '
'. If you use `ls(1)` in the course directory you'll see that there is a directory called "`dir with spaces`". To change into it, you have three options:

```
$ pwd
/home/me/unixcourse
$ cd "dir with spaces"
$ pwd
/home/me/unixcourse/dir with spaces
$ cd ..
$ pwd
/home/me/unixcourse
$ cd 'dir with spaces'
$ pwd
/home/me/unixcourse/dir with spaces
$ cd ..
$ pwd
/home/me/unixcourse
$ cd dir\ with\ spaces
$ pwd
/home/me/unixcourse/dir with spaces
```

So you see, you can use either single or double quotes[4], or just escape each space with a prepended backslash. Again, we recommend that you don't do this because it may

[4] They are not the same in other contexts which we'll encounter later.

break some programs and scripts. Don't get us wrong – we don't think that it's great that you have to stick with these conventions, but as most people do, it will make it harder for you to use others' solutions to problems you encounter.

After this little detour about naming, let's return to creating directories. Per default `mkdir` assumes that the whole path of the directory you want to create exists, except for the last part of the path, i.e. the part behind the last '/'. If you try to create a whole directory hierarchy, this will fail:

```
$ ls dir1
$ mkdir dir1/subdir/subsubdir
mkdir: cannot create directory ‘dir1/subdir/subsubdir’: No such file or directory
```

The first command creates no output because there is nothing in `dir1` and when we try to create a subdirectory `subdir` with a sub-subdirectory `subsubdir` we get an error message (not a very helpful one in this case, but we know what the problem is anyway). But surely, there must be a way to create such hierarchies, right? Of course! You just have to pass the `-p | --parents` option to `mkdir`:

```
$ mkdir -p dir1/subdir/subsubdir
$ ls dir1
subdir
$ ls dir1/subdir
subsubdir
```

Voilà, this time the whole path was createdThe first command creates no output because there is nothing in `dir1` and when we try to create a subdirectory `subdir` with a sub-subdirectory `subsubdir` we get an error message (not a very helpful one in this case, but we know what the problem is anyway). But surely, there must be a way to create such hierarchies, right? Of course! You just have to pass the `-p | --parents` option to `mkdir`:

```
$ mkdir -p dir1/subdir/subsubdir
$ ls dir1
subdir
$ ls dir1/subdir
subsubdir
```

Voilà, this time the whole path was created.

Nothing persists forever, so from creation we move directly to destruction: Whenever you want to delete a directory, you can use `rmdir(1)` "remove directory". Like `mkdir` it takes one (or more) directories as arguments and deletes them. Ok, let's go!

```
$ rmdir dir1/subdir
rmdir: failed to remove 'dir1/subdir/': Directory not empty
```

Hm, that didn't work out as expected. But this is a good thing in this case. `rmdir` only removes directories if they are empty, i.e. there are no files (except '.' and '..') in it.

Why is this such a good thing? Well, now we're getting at something you should *really* remember: On the command-line *there is no undo*! If you delete something, it is gone for good and only people with a lot of training and expertise are able to recover files or directories you deleted.[5] Therefore, `rmdir` prevents you from mistakingly removing a lot of your files with a single line.

However, if you have a situation like we have, where we have `dir1` which contains only `subdir` which only contains `subsubdir` which contains nothing and you want to delete `subdir` and `subsubdir` `rmdir` offers the same option as `mkdir` does, to delete such a chain of empty directories, i.e. `-p | --parents`:

```
$ rmdir -p dir1/subdir/subsubdir
```

This deletes all three directories in the given path. Now, recreate the `dir1` directory and move on.

As you might have guessed, `rmdir` removes only directories – just try it out with the following file, conveniently placed in your course directory:

```
$ rmdir filetodelete
rmdir: failed to remove 'filetodelete': Not a directory
```

Now that wasn't surprising at all. But how can we remove files? The answer to this is `rm(1)` "remove". Thus:

```
$ ls
 contentdir          filetodelete    renamedfile
 dir1                grepfile        rmdirectory
 dir2                hugefile        rminteractive
 dir3                join1           sedfile
'dir with spaces'    join2           sedfile.bak
 errorfile           linefile        sortfile
 file1               newdir          tablefile
 file2               numericsort
 file3               onelinefile
$ rm filetodelete
 contentdir          file1       join2          rminteractive
 dir1                file2       linefile       sedfile
 dir2                file3       numericsort    sedfile.bak
 dir3                grepfile    onelinefile    sortfile
'dir with spaces'    hugefile    renamedfile    tablefile
 errorfile           join1       rmdirectory
```

This time, the file is gone. And remember what we said earlier about removing stuff? It's gone. Totally. Absolutely. No rebirth (without a lot of money). If you'd try to remove a directory with `rm`, this won't work out of the box:

[5] Their training is usually directly reflected in the rather huge amount of money you have to pay them to get your data back.

```
$ rm dir1
rm: cannot remove 'dir1': Is a directory
```

No surprises here. However, `rm` has some options we're going to talk about:

`-r | -R | --recursive` With this option, `rm` removes files and directories recursively, i.e. if you pass a directory as an argument with this option `rm` will remove this directory and *everything* within it. Again, no undo. *NO UNDO!* Sorry, just sayin'. For the sake of an example:

```
$ ls rmdirectory
file_01  file_03  file_05  file_07  file_09
file_02  file_04  file_06  file_08  file_10
$ rm -r rmdirectory
$ ls rmdirectory
ls: cannot access 'rmdirectory/': No such file or directory
```

Yes, the directory and all ten files it contained are gone now.

`-i` If you use this option, `rm` will ask you to confirm the deletion of each file, which can save your one file you forgot was in that directory, but *really* need.[6] Thus:

```
$ ls rminteractive
file_01  file_03  file_05  file_07  file_09
file_02  file_04  file_06  file_08  file_10
$ rm -ri rminteractive
rm: descend into directory 'rminteractive/'? y
rm: remove regular file 'rminteractive/file_01'? y
rm: remove regular file 'rminteractive/file_05'? y
rm: remove regular file 'rminteractive/file_04'? n
rm: remove regular file 'rminteractive/file_09'? y
rm: remove regular file 'rminteractive/file_07'? y
rm: remove regular file 'rminteractive/file_10'? y
rm: remove regular file 'rminteractive/file_02'? n
rm: remove regular file 'rminteractive/file_03'? y
rm: remove regular file 'rminteractive/file_06'? n
rm: remove regular file 'rminteractive/file_08'? y
rm: remove directory 'rminteractive/'? n
$ ls rminteractive
file_02  file_04  file_06
```

For each question `rm` asks you have to type '`y`' for "yes" or '`n`' for "no".

[6] Ever realized that "file" is just an anagram of "life"?

`-f | --force` Force `rm` to do what you told it remove everything without questions[7] `-r | -R | --recursive` was dangerous, *this* one is where the real bad stuff begins. Don't use it if you are not absolutely, 100%, utterly, totally, completely convinced that you know what you are doing. "Oh, come on", you may think, "I'm not *that* stupid, I know this by now. What can possibly go wrong?". Well, have a look at this, were someone accidentally wiped all data of a whole company (including the backup) with one single `rm -rf`. The author of these lines once removed all invoices of a company he worked for, but fortunately there was no `-r` option present, so it were just the invoices of the current year. There was no backup. I had to retype them by hand from printed versions – great fun, 10 out of 10, would recommend...

So, what else can you do with files? What about copying stuff from one place to somewhere else? This can be done with the `cp(1)` command. Its basic syntax is:

```
cp [OPTION]... SOURCE DESTINATION
```

i.e. after maybe passing one or more options to `cd` you first tell it *what* you want to copy and then you tell it *where* to copy it. Source and destination can be relative or absolute paths. As a very simple example, you can do:

```
$ ls
 contentdir          file1      join2          rminteractive
 dir1                file2      linefile       sedfile
 dir2                file3      numericsort    sedfile.bak
 dir3                grepfile   onelinefile    sortfile
'dir with spaces'    hugefile   renamedfile    tablefile
 errorfile           join1      rmdirectory
$ cp file1 file2
$ ls
 contentdir          file1      join2          rminteractive
 dir1                file2      linefile       sedfile
 dir2                file3      numericsort    sedfile.bak
 dir3                grepfile   onelinefile    sortfile
'dir with spaces'    hugefile   renamedfile    tablefile
 errorfile           join1      rmdirectory
```

However, there are some things about this simple scheme you have to take into account:

- If you want to copy a directory and all its content you have to use the `-r | -R | --recursive` option, like with `rm` when you wanted to delete a directory with its contents.
- If the destination is a directory, `cp` will copy the source into the destination directory, e.g.:

[7] In some cases `rm` asks you if you really want to delete a file even without the `-i` option present. See `man 1 rm`.

```
$ ls dir1
$ cp file1 dir1
$ ls dir1
file1
```

- If the destination is a directory, you can give multiple sources, i.e. you can copy multiple files at once into a directory by listing them all on the command-line and providing the directory you want to copy them into as the last argument:

  ```
  $ cp file1 file2 file3 dir1
  $ ls dir1
  file1 file2 file3
  ```

- But, wait, we already copied `file1` to `dir1`, so which version is now in `dir1`? As you may have guessed from the behaviour of `rm` and `rmdir`: The command-line is unforgiving. If there is already a file with the same name in a directory, this file is overwritten and `cp` won't tell you about it – it assumes that you know what you are doing. Fortunately, just like `rm` there is a `-i | --interactive` option which asks you every time a file would be overwritten. In addition to that, `cp` also offers the `-n | --no-clobber` option which tells the program to never overwrite existing files.

The last but one thing we're going to cover for working with files is ow to rename and move them around. This is done by the same utility, `mv(1)` "move". Its standard syntax is:

```
mv [OPTION]... SOURCE DESTINATION
```

Not only is its syntax the same as the one for `cp`, the same caveats about how to move multiple files and moving directories apply as well, with one exception: There is no `-r | -R | --recursive` or a similar option for `mv` – it always moves a whole directory if you pass one as source. If you just want to rename a file without moving it into another place in the filesystem, you just put the destination in the same directory:

```
$ ls
 contentdir          file1      join2          rminteractive
 dir1                file2      linefile       sedfile
 dir2                file3      numericsort    sedfile.bak
 dir3                grepfile   onelinefile    sortfile
'dir with spaces'    hugefile   renamedfile    tablefile
 errorfile           join1      rmdirectory
$ mv file3 renamedfile
$ ls
 contentdir          file1      join2          rminteractive
 dir1                file2      linefile       sedfile
```

```
 dir2               file3      numericsort   sedfile.bak
 dir3               grepfile   onelinefile   sortfile
'dir with spaces'   hugefile   renamedfile   tablefile
 errorfile          join1      rmdirectory
```

We'll end this section with something a bit more abstract again. In 4.1 we introduced you to the concept of permissions when you encountered the output of `ls -l`. Permissions were originally used in UNIX when many users sitting in front of terminals (the hardware devices you saw in 3.1) connected to central computers where all of them worked at the same time. To enable them to collaborate more easily they could set the permissions for group and others as they wanted to. This was done by the `chmod(1)` "change (file) mode". `chmod`'s basic syntax is:

```
chmod [OPTION]... MODE[,MODE]... FILE...
```

There are two ways to pass a set of permissions to `chmod`: By a symbolic representation and by an octal number. The general format of the symbolic mode is:

```
[ugoa...][[-+=][perms...]...]
```

The first part indicates for whom you want to change the permissions: `u` for "user", i.e. the user who owns the file, `g` for "group", `o` for "others" and `a` for "all". The default is `a`, i.e. to change the permissions for everyone, but if you write `u` you change only the user's permissions and if you write `ug` you change the permissions of the user and the group. As you may have noticed `ugo` and `a` are equivalent.

The second part means what you want to do: `+` to add the following permissions, `-` to remove the following permissions and `=` to set the permissions to exactly the following permissions.

The third field consists of the permissions you want to add/remove/set, i.e. the letters `r`, `w`, and `x`.[8]

Consider this example:

```
$ ls -l file1
-rw-r--r-- 1 me me 29 Sep 18 15:24 file1
$ chmod g+w file1
$ ls -l file1
-rw-rw-r-- 1 me me 29 Sep 18 15:24 file1
```

As you can see, we gave the group (`g`) additional (`+`) write permission (`w`), but left the other permissions intact.

Now it's your turn: Try do remove all rights from the group and the others. Is there more than one way to do this?

We're not going to explain the octal mode here: It's not hard, but it can be confusing at the beginning and can take a bit more time to grasp – read the manpage if you are interested :)

[8] If you look at the manpage of `chmod` you'll see that there are three more possible values, but for now, we'll ignore them.

4.3 Working with Content

At this point you know how to move around and how to change files, but you haven't learned how to work with the stuff that is *in* files, i.e. content. Many files on your system contain human readable[9] text which you can easily manipulate.

4.3.1 Looking at Content

But before we start writing stuff somewhere, we'll look at it first. The easiest way to see the contents of a file is to use the `cat(1)` command. This is an abbreviation for "concatenate", i.e. combining stuff. Its basic syntax is:

```
cat [OPTION]... [FILE]...
```

Now let's unravel the mysteries of the ominous `file1` with which we played around in the last section without ever knowing what's in there:

```
$ cat file1
A file with copiable content
```

Well, that wasn't very mysterious... It just contains a single line of text saying "A file with copiable content". So that is what `cat` does: It reads the contents from a file and prints them to the screen. What about another file:

```
$ cat onlinefile
This is a nice one liner
```

Ok, also nothing very interesting. But what about this concatenation thing? Have a look:

```
$ cat file1 onelinefile
A file with copiable content
This is a nice one liner
$ cat onelinefile file1
This is a nice one liner
A file with copiable content
```

You see, `cat` concatenates the content of the files in the order you pass them on the command-line. This is tremendously useful if you want to e.g. combine multiple files into one very long file without copy-pasting all the stuff. How to do this will be shown later.

As you may have realized, if files have a lot of content, i.e. many pages of test, `cat` will be rather inconvenient because it just prints everything to the screen and you can't read fast enough to see everything:

```
$ cat hugefile
```

[9] Well, let's say "readable by *some* humans" ;)

To read through such a file, you can use programs called "pagers" – they display text files one page at a time. The standard pager on (GNU/)Linux is `less(1)`.[10] So, let's view our file:

```
$ less hugefile
```

To move around scroll forward one page, use `SPACE`, for backwards use `B`. You can also move up and down one line with the arrow keys. To search for a pattern forward press '`/`' enter the pattern and press `ENTER`, to search for a pattern backwards, press '`?`' and press `ENTER`. When you finally want to quit the program, press `Q`.

But then again, how are you supposed to know beforehand how big a file is, i.e. whether it makes sense to use `cat` or `less`? There are different strategies and one you know already:

```
$ ls -l onelinefile hugefile
-rw-r--r-- 1 me me 431888 Sep 18 17:52 hugefile
-rw-r--r-- 1 me me     25 Sep 17 13:53 onelinefile
```

If you look at the size column you can see that there is a huge difference between these two files. Unfortunately, using bytes as a measuse is not very human-readable, especially because even `hugefile` could in fact consist of only a single line – a *very* long line, but it is possible. Therefore, it would be useful to be able to view the lines of a file. This can be done with `wc(1)` "word count":

```
$ wc onelinefile
 1  6 25 onelinefile
$ wc hugefile
 7654  70552 431888 hugefile
```

`wc` provides with some useful *metadata*, i.e. data about data, about a file. The first column is the number of lines in the file, the second column shows the number of words in the file and the third column tells you how many bytes the file contains. You can also list just one of these units using the `-l | --lines` for only the line count, `-w | --words` for only the word count, and `-c | --bytes`[11] for only the byte count.

4.3.2 Creating Content and I/O redirection

After so much information reception it is time to let you create something yourself! And the first tool for this is `echo(1)`. Very unexpectedly `echo` echoes what you pass to it:

[10] In the old days, the standard pager was `more(1)`. When someone wanted to replace it they thought that "more is less" and thus the name of `less`.

[11] "Why `-c`?" you may ask. This is because years ago characters were always encoded in ASCII (see the Wikipedia entry for an explanation) where each of them was exactly one byte in size and so the `-c` for **c**haracter came into being.

```
$ echo Hello command-line
Hello command-line
```

This seems easy enough. But don't be fooled, `echo` is an extremely useful tool and will introduce you to a lot of new concepts. Just as an aside: It is often a good idea to put the stuff you want `echo` to echo in double quotes. Read the man page to find out why this is so ;)

The first one is the concept of *I/O streams*[12]. A stream is a flow of data made available through time. While this sounds rather abstract, you've already worked with streams all the time in this course! Whenever you open a `bash` there are three standard streams opened for this `bash` as well:

`stdin` or "Standard Input", i.e. the stream into which you put data. This is usually your keyboard because this is where you input the stuff you write. The `bash` then reads those characters from the keyboard and displays them on the command-line.

`stdout` or "Standard Output", i.e. the stream into which the data goes. In most cases this is your screen or rather the terminal emulator on your screen. Yes, most of the time a program produces output, it writes it to `stdout`!

`stderr` or "Standard Error", i.e. the stream into which error messages are sent. This is also mostly your screen and you've already seen this in action: Remember when we tried to remove a non-empty directory with `rmdir`? There was an error message displayed and while you saw it on your screen, this was actually written to `stderr` and not to `stdout`.

But why make two stream which write to the same thing?! The reason is simple: They do so usually, but you can manipulate them indiviually. Manipulate? Oh yeah, you can play around with those streams until your head spins! This called I/O redirection.

It is time, to create your first file! You know now that whenever `echo` echoes something back to you, it actually reads input from `stdin` and puts it to `stdout`. But what if we *redirect* `stdout` to a file? See here:

```
$ ls
 contentdir          file1       join2          rminteractive
 dir1                file2       linefile       sedfile
 dir2                file3       numericsort    sedfile.bak
 dir3                grepfile    onelinefile    sortfile
'dir with spaces'    hugefile    renamedfile    tablefile
 errorfile           join1       rmdirectory
$ echo "I want this in a file" > echofile
$ ls
 contentdir          errorfile    join1          rmdirectory
 dir1                file1        join2          rminteractive
```

[12] For Input/Output stream.

```
 dir2               file2       linefile      sedfile
 dir3               file3       numericsort   sedfile.bak
'dir with spaces'   grepfile    onelinefile   sortfile
 echofile           hugefile    renamedfile   tablefile
$ cat echofile
I want this in a file
```

So now you know that by using '>' you can redirect `stdout` to a file. Let's try that again:

```
$ echo "More content" > echofile
$ cat echofile
More content
```

Ok, that was not expected. Where's the former line gone? Well, it's not here anymore. If you use '>' to redirect something to a file, the file will be *truncated*, i.e. totally emptied, and afterwards the new content will be written to the file. If you want to just add something to a file, without deleting what is already in there, you have to use ">>", which *appends* data to a file:

```
$ echo "And now a new line" >> echofile
$ cat echofile
More content
And now a new line
```

Note that there is no way to put something in the middle of a file using just the command-line. To do this you need a text editor which we'll discuss later.

You can also use this redirection on any other command, e.g. if you want to store its output so you can look at it later:

```
$ ls -l > lsfile
$ cat lsfile
total 516
drwxr-xr-x 13 me me   4096 Sep 19 10:05 contentdir
drwxr-xr-x  2 me me   4096 Sep 18 15:36 dir1
drwxr-xr-x  2 me me   4096 Sep 17 13:52 dir2
drwxr-xr-x  2 me me   4096 Sep 17 13:52 dir3
drwxr-xr-x  2 me me   4096 Sep 17 15:36 dir with spaces
-rw-r--r--  1 me me     22 Sep 23 12:29 echofile
-rw-r--r--  1 me me     63 Sep 19 11:07 errorfile
-rw-r--r--  1 me me     29 Sep 18 15:24 file1
-rw-r--r--  1 me me     29 Sep 18 15:33 file2
-rw-r--r--  1 me me     11 Sep 23 12:22 file3
-rw-r--r--  1 me me    472 Sep 23 09:49 grepfile
-rw-r--r--  1 me me 431888 Sep 18 17:52 hugefile
-rw-r--r--  1 me me     46 Sep 23 12:11 join1
-rw-r--r--  1 me me     48 Sep 23 12:11 join2
```

```
-rw-r--r--  1 me me    405 Sep 23 12:22 linefile
-rw-r--r--  1 me me      0 Sep 23 12:29 lsfile
-rw-r--r--  1 me me     26 Sep 23 12:22 numericsort
-rw-r--r--  1 me me     25 Sep 17 13:53 onelinefile
-rw-r--r--  1 me me     69 Sep 18 15:36 renamedfile
drwxr-xr-x  2 me me   4096 Sep 18 11:14 rmdirectory
drwxr-xr-x  2 me me   4096 Sep 18 15:18 rminteractive
-rw-r--r--  1 me me    331 Sep 23 11:02 sedfile
-rw-r--r--  1 me me    331 Sep 23 10:48 sedfile.bak
-rw-r--r--  1 me me     18 Sep 23 12:22 sortfile
-rw-r--r--  1 me me    747 Sep 23 11:25 tablefile
```

Your `ls` output has now been stored in a file.

Now, what if we want to do something with `stderr`?

```
$ ls nonexistentfile
ls: cannot access 'nonexistentfile': No such file or directory
$ ls nonexistentfile > errorfile
ls: cannot access 'nonexistentfile': No such file or directory
$ cat errorfile
$ ls nonexistentfile 2> errorfile
$ cat errorfile
ls: cannot access 'nonexistentfile': No such file or directory
```

You can see that if we use '>' to redirect the output, `errorfile` stays empty, because `ls` didn't print anything to `stdout`, just to `stderr` and because we just redirected `stdout` to the file, `stderr` is still printed to the command-line. The redirection for `stderr` is "2>" and you can see that using this results in the error message being not printed to the screen, but written to the file instead.

To be honest, we cheated a bit: Actually the streams are numbered, starting with zero: `stdin` is 0, `stdout` is 1 and `stderr` is 2. You can only write to `stdout` and `stderr` because they are output streams while `stdin` is an input stream you can't write to.[13] Thus, only output streams can be redirected using '>' and you can prepend the number of the output stream you want to redirect, i.e. `1>` or `2>`. If you don't specify anything, however, `1>` is implied because you much more often want to redirect `stdout` than `stderr` and this is why just '>' above worked as it did.

But we talked about "manipulating streams individually". What does that mean?

```
$ ls file1 nothere
ls: cannot access 'nothere': No such file or directory
file1
$ ls file1 nothere > lsfile 2> errorfile
$ cat lsfile
```

[13] This is pretty easy to understand if you think about the meaning of the streams: How would you try to write to your keyboard?

```
file1
$ cat errorfile
ls: cannot access 'nothere': No such file or directory
```

As you can see, it is possible to redirect `stdout` and `stderr` to different files. This can be very useful if you have a program with a lot of output and you want to keep normal output and error messages apart.

Note that if you want both of them to redirect to the same file you have to first redirect one stream to the file and then redirect the other one to the other stream:

```
$ ls file1 nothere > twostreams 2>&1
$ cat twostreams
ls: cannot access 'nothere': No such file or directory
file1
```

To redirect `stderr` to `stdout` you use `2>&1` which reads approximately as "redirect `stderr` to where `stdout` writes to". You could also do the following:

```
$ ls file1 nothere 2> twostreams 1>&2
$ cat twostreams
ls: cannot access 'nothere': No such file or directory
file1
```

The result is the same. Note, however, that the order is important: *First*, redirect one stream to a file, and *afterwards* redirect the other stream to it. This makes sense: If you read the sentence describing `2>&1` carefully, you'll notice that it says "where `stdout` writes to" and not "to `stdout`". Thus, if you redirect `stderr` to `stdout` first, and afterwards redirect `stdout`, `stderr` writes to where `stdout` wrote to before you redirected it.

To finish this, we'll give you a little shortcut: If you don't want to care about this first and second thing and consider "`> filename 2>&1`" an awful lot to type, you can use the short version: "`&> filename`". This is much more convenient, but introduces a new caveat: Not all shells are the same and this shortcut is something that works in `bash` but does not necessarily do so in other shells you may encounter. Just keep this in mind if you run into trouble ;)

In 4.3.1 when we introduced you to `cat` we mentioned that you can use it to combine contents of multiple files into a single one. Now you have all the necessary tools; play around with `cat` and I/O-redirection :)

4.4 Archiving and Compression

Sometimes it is useful to put files into a collection called an *archive*. An example would be if you want to send multiple files via email, but you don't want to append 10 files, but just one – this would be an archive.

Another thing that is useful in many cases is *compression*. Compressing data means to encode it in a way that uses less bits than the original data. This is possible by using algorithms which make use of redundant data – most data has such redundant elements because it is in certain file formats which demand structure, i.e. patterns.

4.4.1 Archiving with `tar`

To create an archive of multiple files on (GNU/)Linux and many other UNIX-like systems you can use the `tar(1)` "tape archive" utility. You can tell by its name that this program is pretty old, as it was created to work with tapes.

`tar` has an incredible amount of functionality (have a look at the manpage) and we will cover only the most basic stuff here. To create an archive you can use:

```
$ tar -cvf myfirstarchive.tar file1 file2 hugefile
file1
file2
hugefile
```

The explanation of the options is quite straightforward:

`-c | --create` tells `tar` to create an archive.

`-v | --verbose` tells `tar` to be verbose, i.e. to print each file it processes to `stdout`.

`-f | --file=ARCHIVE` tells `tar` how it should call the archive. It is conventional to add the `.tar` extension to `tar` archives, though not strictly necessary. However, do yourself a favour and use this extension ;)

After you told `tar` what you want and where to put it[14] you have to put all files you want to be in the archive on the command-line. If you pass a directory to `tar` it will automatically put all files in the directory in the archive as well, so you don't have to tell it explicitely to do that like you had to with `cp` or `rm`.

To check that everything worked correctly, you can check which files are in an archive by using the `-t | --list` option:

```
$ tar -tf myfirstarchive.tar
file1
file2
hugefile
```

This works essentially like `ls` but for archives.

But if you received an archive, the relevant thing you want to do is not just to list the files but to *extract* them on your system. To try this, create a directory, change into it and extract the files there using the `-x | --extract | --get` option:

[14] I.e. the name of the archive. This can be an absolute or a relative path.

```
$ mkdir tar-dir
$ cd tar-dir
$ ls
$ tar -xvf ../myfirstarchive.tar
file1
file2
hugefile
$ ls
file1  file2  hugefile
```

It is always a good idea to look into an archive you downloaded from the Internet before extracting it, to make sure that it really contains the files you want and not something else.

4.4.2 Compression Utilities

While archiving today is done mostly with `tar`, there are several utilities to compress files. Each of them uses a different algorithm to compress the data, but their usage is very similar. The only thing you should know about the algorithms used is how efficient they are in order to decide which one will be most appropriate for the problem you are facing.

One important fact you should know before starting to use compression is the following: Don't compress data which is already in a compressed format! If you do this you'll end up with a file that is *bigger* than the original one. Why is this? Well, as pointed out above, compression utilizes redundancies in data to store the same information in less bits. It also adds a bit of metadata to make recovering of the original data possible. If you have compressed data once and your compression algorithm is decent, there will be no redundant bits left to remove. Thus, if you now apply compression again, the algorithm will find nothing it can compress, but still adds its metadata, meaning you have more data than before.

For this reason, it is a good idea to check if the format of the files you want to compress already uses compression – well known examples are `.pdf`-files, `.odt`-files, and even `.docx`-files. You can try this out for yourself after you've learned how to use the utilities.

`gzip(1)`

The first tool we're going to look at is `gzip(1)`. Its basic syntax is:

```
$ gzip FILENAME...
```

By default `gzip` will do the following to each file you pass to it:

- Compress the data and store it in a file with the same name as the original plus the `.gz` extension.

- Remove the original file.

This especially the second point is important to keep in mind – if you want to keep your old file(s), use the `-k | --keep` flag.

To decompress a compressed file you have two options:

1. Use the `-d | --decompress | --uncompress` option.
2. Use the `gunzip` command which is more or less just a shortcut of the above.

By default this restores the uncompressed file and deletes the compressed one afterwards – again you can use `-k | --keep` flag to change this behaviour.

However, sometimes you just want to see what's in a compressed file without storing the decompressed version. To do this, you have two options again:

1. Use the `-c | --stdout | --to-stdout` flag which little surprisingly just prints the contents of the file to `stdout`.
2. Use the `zcat` utility which is again just a shortcut for the above.

Demo time![15]

```
$ ls -lh hugefile
-rw-r--r-- 1 me me 431888 Sep 18 17:52 hugefile
$ gzip hugefile
$ ls hugefile
ls: cannot access 'hugefile': No such file or directory
$ ls -lh hugefile.gz
-rw-r--r-- 1 me me 165943 Sep 18 17:52 hugefile.gz
$ gunzip hugefile.gz
$ ls hugefile.gz
ls: cannot access 'hugefile.gz': No such file or directory
$ ls -lh hugefile
-rw-r--r-- 1 me me 431888 Sep 18 17:52 hugefile
```

You can see that the original file was compressed to less than half of its size! However, you can do even better: Compressing data takes time and time is often precious. The better the compression the longer it takes, which can be a problem. Therefore `gzip` provides an option to adjust the speed vs. compression quality. You can pass it a number between 1 and 9 preceeded by a dash to indicate you want the program to be really fast (`-1`) or compress really good (`-9`). The default level is `-6`. Have a look:

```
$ ls -l hugefile
-rw-r--r-- 1 me me 431888 Sep 18 17:52 hugefile
$ gzip -1 hugefile
```

[15] The `-h | --human-readable` option to `ls(1)` makes the size output more readable for as by adjusting the units; e.g. 1024 byts will be shown as 1K instead to indicate that this is one KB.

```
$ ls -l hugefile.gz
-rw-r--r-- 1 me me 195743 Sep 18 17:52 hugefile.gz
$ gunzip hugefile.gz
$ gzip -9 hugefile
$ ls -l hugefile.gz
-rw-r--r-- 1 me me 165202 Sep 18 17:52 hugefile.gz
```

So the best compression produces a file which is about 15% smaller than what the worst compression creates. This may seem minor to you, but think about data center which have to store hundreds of thousands of terra bytes of data where this can make an enormous difference.

And now a quick demonstration of `zcat`:

```
$ cat file1
A file with copiable content
$ gzip file1
$ ls file1.gz
file1.gz
$ zcat file1.gz
A file with copiable content
$ ls file1.gz
file1.gz
```

You can see that in this case the compressed file stays intact and just its content is output to `stdout`.

Of the compression utilities we're going to discuss here, `gzip` is the "worst" one in terms of compression, but it is still widely used and "worst" in this case means still very good – and fast!

`bzip2(1)`

The `bzip2(1)` command has pretty much the same syntax and default behaviour as `gzip`, and there are a `bunzip2` as well as a `bzcat` utilities. Compressed files get the `.bz2` by default.

However, not all things are totally identical, so read the man page of `bzip2` and play around a bit to compare the compression rates of `bzip2` and `gzip`. In general, `bzip2` offers better rates than `gzip` but worse than the following utility.

`xz(1)`

Again, syntax and behaviour are very similar to the above and there are `unxz` and `xzcat` utilities offering the same functionality as with the other two programs. By default compressed files get the `.xz` extension.

This is the best and most versatile compression tool of the ones we discuss which will become apparent if you read its much longer man page and see what options it offers.

Again, play around to get a feeling for the program.

4.4.3 Combining Archiving and Compressing Data

You've seen that you can put multiple files into an archive which is a single file and you know that you can compress files – wouldn't it be convenient to combine these features, i.e. to create an archive with all the files you want and then compress the whole archive in one step? The good news is: You can. The better news is: `tar` has options to combine this into one single step.

`-z | --gzip` What this option does depends on the arguments you pass to `tar`. If you have given the `-c` option, `tar` will first create an archive and then compress the whole thing in one step:

```
$ tar -cvzf files.tgz file1 file2 file3
file1
file2
file3
$ ls files.tgz
files.tgz
$ tar -xvzf files.tgz
file1
file2
file3
```

Conventionally `gzip` compressed `tar`-archives use `.tar.gz` or less verbose `.tgz` as extension.

`-j | --bzip2` This compresses/decompresses a `tar` archive using the `bzip2` utility. Such archives usually get the `.tbz` or `.tbz2` extension. Again you can also use `.tar.bz2` if you want to be more verbose.

`-J | --xz` To your surprise, this will create a `xz`-compressed archive. The conventional extension is `.txz` in this case.

Note that if you use compression with `tar` the original files stay intact; just like when you use `tar` to create an archive.

4.5 Textfiles, Pipes, Wildcards, and a Little Bit of Regex

Here we enter the last big section about working with files and their content on the command-line, so take a deep breath and get ready to learn some incredibly useful concepts and tools!

All of them are mainly concerned with working with textfiles. You've already encountered `echo`, `cat`, `less`, and `wc`, but we'll add many more and find new applications for some of them as well.

4.5.1 Of Heads and Tails

Sometimes you want to look only at the beginning or the end of a file, e.g. if you do some C programming you may just want to see the `#includes` and comments at the top of a source file, or you've written a LaTeX file and forgot which packages you included, but you want them in a new project, so again, you only want the first few lines. To do this, there is the `head(1)` utility, whose basic syntax is:

```
$ head [OPTION]... [FILE]...
```

It prints the "head" of a file to `stdout`, which by default means the first 10 lines of each file you passed to it as an argument. However, 10 is sometimes not enough (or too much), so you can easily control the number of lines being printed with the `-n | lines=NUM` option:

```
$ head linefile
Line 02
Line 03
Line 04
Line 05
Line 06
Line 07
Line 08
Line 09
Line 10
$ head -n 5 linefile
Line 01
Line 02
Line 03
Line 04
Line 05
$ head -5 linefile
Line 01
Line 02
Line 03
Line 04
Line 05
```

As you can see, you can leave out the `-n` and write the desired number directly after the dash.

On the other hand, you may have added something to a file and don't remember what it was, or if you are running a web server or anything else that creates logs it is often useful to just look at the last lines of a textfile. The utility of choice in this case is `tail(1)`. Its syntax is equivalent to the one of `head` and by default it prints the last 10 lines of a file to `stdout`.

There is one common usecase for `tail`, however, which is not commonly found with `head`: We mentioned programs creating log files, i.e. files which log their activity. These files increase through time as the programs append more and more data to the end of the logs. If you want/have to monitor such programs, it is often useful to watch the logs while growing, but of course you only want to see the new entries at the end of the file. This can be done using `tail` with the `-f | --follow` option. To test this, do the following:

1. Type `tail -f linefile`.
2. Open a second terminal window and change to the `unixcourse` directory.
3. Append a new line to `linefile`:

   ```
   $ echo "newline" >> linefile
   ```

4. In the window which runs `tail -f linefile` you will see that a new line has been added.

Now try what happens, if you type '>' instead of ">>".

To stop `tail` from running, go to the terminal window it runs in and press `CTRL-C`. This will terminate the process. We'll explain this to you in 4.6, don't worry.

4.5.2 Sorting Text and Using Pipes

You'll encounter many situations where you want to sort the content of a file – e.g. a list of words you want to sort alphabetically or some numbers representing measurements and you want to quickly check their range. This is what the very fittingly named `sort(1)` program is good for:

```
$ cat sortfile
T
B
A
W
c
N
B
I
P
$ sort sortfile
A
B
B
c
I
```

```
N
P
T
W
```

That was easy enough! And now for numbers:

```
$ cat numericsort
9
253
85
76
0010
1000
$ sort numericsort
0010
1000
123
253
76
85
9
```

What?! Most likely this did not correspond to your expectations. How can `sort` think that 1000 is less than 9?? Of course, `sort` is not that stupid. The problem is caused by different expectations: `sort` sorts text alphabetically and numbers are just part of the alphabet: They come before all letters and there are exactly 10 of them: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Thus, `1000` and `9` are just words – `1000` starts with `1` which precedes `9` and must thus be placed first, like `abcd` precedes `z`. That is what alphabetically sorted means: you order everything according to their first letter, and only then you look at the second one to order those with the same first letter further, etc.

"OK", you may say, "this may be an answer, but it is still pretty annoying." Correct. And because this problem is very common, `sort` offers the `-n | --numeric-sort` option which tells it to sort the lines of a file numerically:

```
$ sort -n numericsort
9
0010
76
85
123
253
1000
```

Much better!

In the above output, which sorted the lines of `sortfile` alphabetically the letter `B` appeared twice. This is all nice and fine in many situation, but sometimes you want only unique the unique entries in a file, e.g. if you want to check how many different measurements you had or how many different users accessed your web server. If you do some research, you'll find that this is what the `uniq(1)` program thus. If you read its man page you'll find the following:

> Note: 'uniq' does not detect repeated lines unless they are adjacent.

What this essentially means is, that you have to be sure that all duplicate lines in a file are adjunct before passing the file to `uniq`. One possibility to achieve this is to sort the file first. Using what you've just learned and I/O redirection, you can do the following:

```
$ sort sortfile > tmp
$ uniq tmp
A
B
c
I
N
P
T
W
```

Great, there is only one line now starting with `B`. However, this workflow is rather clumsy. And worse: Imagine the file whose unique entries you want has 10 or even 100 GB – this approach forces you to create another file of the same size just for this little task.

Surely, there must be more intelligent way to do this! There is: Pipes.[16]

A pipe is a unidirectional data channel which allows you to "pipe" the output of one program as input to another program, i.e. `stdout` of one program will be sent to `stdin` of another program.[17]

To pipe the output of one program to another one, you use the vertical bar '`|`':

```
$ sort sortfile | uniq
A
B
c
I
N
P
T
W
```

[16] For those of you eagerly reading man pages who have already found the `-u | --unique` option for `sort`: Yes, this works perfectly fine, but for now, pretend you have no idea ;)

[17] Actually, pipes can do more than just this, but for our purposes, this is totally sufficient.

It does not look like much, but this is one of the most powerfull abilities the command-line offers because you can combine arbitrarily many commands in this manner, each one changing the output and passing it to the next. Try to find out what the following line does and what output you expect before running it:

```
$ sort sortfile | uniq | head -n 5 | tail -n 3
```

Or what about this:

```
$ head -n 200 hugefile | less
```

You can do an incredible lot of things with piping and we're going to encounter more usecases soon.

Note that you pipe `stdout` to `stdin`, leaving `stderr` alone. This means that if something in your pipe goes wrong, you'll see the error messages printed to the screen:

```
$ ls file1 file2 file3 file4 | head -2
ls: cannot access 'file4': No such file or directory
file1
file2
```

4.5.3 Wild Cards and Gentle Regexes

Until now you passed each argument you wanted a command to process directly to it, only using tab completion to increase your speed.

But, what if you have a directory with hundreds of files and you don't even know all of their names, but you do know that all of them contain some string or have a common number in their name? In this case you may want to use *wildcards*. Wildcards are characters the shell treats as special and which enable you to look for patterns. They are:

`*` The asterisk matches any string, including the null string, i.e. the empty string.

`?` The question mark matches any single character.

`[...]` This pattern matches any one of the enclosed characters.

But what does that mean? Let's look at some examples:

```
$ ls *
 contentdir         file1      linefile        sedfile
 dir1               file2      lsfile          sedfile.bak
 dir2               file3      numericsort     sortfile
 dir3               grepfile   onelinefile     tablefile
'dir with spaces'   hugefile   renamedfile
 echofile           join1      rmdirectory
 errorfile          join2      rminteractive
$ ls *file
```

```
echofile   hugefile  onelinefile  sortfile
errorfile  linefile  renamedfile  tablefile
grepfile   lsfile    sedfile
$ ls file*
file1 file10  file2  file3
$ ls *file*
echofile   file2        hugefile     renamedfile  tablefile
errorfile  file3        linefile     sedfile
file1      grepfile     lsfile       sedfile.bak
file10     hasfileinit  onelinefile  sortfile
$ ls file?
file1  file2  file3
$ ls [flsn]*
file1   file2  linefile  numericsort  sedfile.bak
file10  file3  lsfile    sedfile      sortfile
```

Try to explain the output yourself and only read on afterwards!

The first example just passes a '`*`' to `ls` and as this matches any pattern, all files in the directory are listed. The second example matches all files whose name ends in `file`. The third example matches all those whose name begins with `file`. The fourth example matches any file whose name contains the string `file` somewhere. The fifth example matches any file whose name starts with `file` followed by exactly one additional character. The last example matches any file whose name starts with `f`, `l`, `s`, or `n`.

Wildcards can be very handy to match only certain files but not others. However, they are restricted to the command-line and still very coarse grained filters. What if you want to check if a pattern exists e.g. at the end of lines in a file? This can be achieved using a *regular expression*, or regex for short.

Regular Expressions and `grep(1)`

First, let's shortly introduce you to the `grep(1)` utility. This is the program of choice if you want to "grab" some text patterns in a file. Using `less` and `cat` you know how to look at everything that's in a file and in combination with `sort`, `uniq`, `head`, and `tail` you've learned to remove some information, but until now you can't look for some specific patterns in a file, e.g. is there a measurement of exactly "3.14159265358979" or a sentence which contains the word "octothorpe" in the file?

With `grep` you're able to do this. Its basic syntax is:

```
grep [OPTION]... PATTERN [FILE]...
```

and the program looks for `PATTERN` in each file you pass to it. So, let's see if this measurement we talked about above is found:

```
$ grep "3.14159265358979" grepfile
```

```
$ grep "octothorpe" grepfile
but only here you'll find a wild octothorpe "#"
and for the curious: hash == hashtag == octothorpe
```

Now you know that there is no line containing the beginning of pi in this file, but two contain the word "octothorpe". If you also want to know which line(s) contain matching patterns, use the `-n | --line-number` option to `grep`.

Very well, but what about this regex stuff? The pattern you pass to `grep` can be a regex. Using regexes is similar to using wildcards, but it is *much* more powerful. To get the official definition for POSIX[18] regular expressions you can read `regex(7)`, but we'll do it a bit less formal.

As with wildcards, certain characters in regexes do not have their literal meaning. These are called *metacharacters*. These metacharacters are:

`^`	The caret matches the beginning of a line.
`$`	The dollar sign matches the end of a line.
`.`	The dot matches any single character (similar to the '?' wildcard).
`[ ]`	As with wildcards, a bracket-expression matches any single character within the brackets.
`[^ ]`	A bracket-expression starting with a caret matches any single character *not* found within the brackets, `[^abc]` matches anything except `a`, `b`, and `c`.
`*`	The asterisk in a regular expression matches the preceding element 0 or more times. Note that this is different from the meaning of the asterisk as a wildcard! Using wildcards, the expression `b*` matches everything that starts with the letter 'b', but as a regular expression it matches 0 or more 'b's!

With these new knowledge, try the following using `grep` with `grepfile`:

- Print all lines which contain a '#'.
- Print all lines which begin with a '#'.
- Print all lines which end in a punctuation mark.
- Print all lines which end in 'Z' or 'z'.
- Print all empty lines.
- Print all lines which contain no 'a'.

Check you results by looking at `grepfile` with `cat` or `less`. If you can solve all of them: Great! If not: Don't despair, it takes time to get used to the concept of regexes.

[18] "Portable Operating System Interface X". The 'X' was appended because it was intended for UNIX-like OSs.

There is also a very useful component to `grep` and regexes if you like to play crossword puzzles: On many distros, you can find at least one word list of the language your system runs on in `/usr/share/dict`. The most common one is just called "words". On my system it contains 102401 words. Suppose now that you are looking for a mountain range consisting of 12 letters, the second one is a 'p', and there fifth, fourth, and third to last letters are "hia":

```
$ grep '^.p.....hia..$' /usr/share/dict/words
Appalachia's
Appalachians
```

As an aside: If you ever want to look for one of the metacharacters as a literal, e.g. you want to know which lines contain a '$' you have to escape them with a backslash, i.e. you have to type "\$" in your pattern.

However, regexes don't end here. There are some more metacharacters than those we've see so far and we'll cover them now:

`( )` The expression between the parentheses is seen as a single element from the outside, e.g. `(abc)*` matches 0 or more occurences of the string "abc", e.g. it matches "abc", "abcabc".

`\`*n* Matches the n^{th} expression delimited by parenthesis, e.g. `(ab)(cd)..\2\1` matches any string that starts with "abcd" followed by two characters followed by "cd" followed by "ab".

`{`*m*`,`*n*`}` Matches any string that contains the preceding expression at least *m* and at most `n` times, e.g. `q{2,5}` matches 2, 3, 4, or 5 consecutive 'q's.
The version of `grep` that is installed on most (GNU/)Linux distros also allows some variations of this scheme: If you pass only one number *n* without a comma, the expression matches occurences of exactly *n* times the same character, e.g. `d{3}` matches exactly "ddd". You can also leave out one of the numbers but provide a comma to give just a lower or upper bound, e.g. `i{,10}` indicates "up to 10 consecutive 'i's" while `i{10,}` reads as "at least 10 consecutive 'i's".

`?` Matches the preceding expression zero or one times, e.g. `friends?` matches "friend" or "friends". Like with the asterisk, this differs from the question mark if it is used as a wildcard! The wildcard '`?`' is the same as the regex '`.`'.

`+` The plus sign matches one or more occurences of the preceding expression, e.g. `w+` matches 'w', "ww", "www", etc. The difference to the asterisk is that the plus sign does *not* match zero occurences.

`|` The vertical bar matches either the preceding or the following expression, e.g. `ab|Bc` matches "abc" and "aBc". Of course, this could also be done by `a[bB]c`, so why care about '`|`'? Well, if you combine it with parentheses or braces, it becomes clear: `(sw)|m|(gr)eet` matches "sweet", "meet", or "greet" or `(Ada)|(A\. Lovelave` matches "Ada Lovelace" and "A. Lovelace". Or think about something like `a{2,3}|e{1,5}`.

Unfortunately, with these metacharacters things are a bit confusing: While you had to use a backslash if you wanted the literal meaning of the metacharacters we talked about before, with this group it is the other way around: `grep` (and other tools) interpret them literally, *unless* you prepend a \'! For example, '+' is interpreted as a plus sign, you have to type `\+` to get the regex meaning. This can be pretty annoying, especially as this behaviour is not consistent: Some tools treat all metacharacters as metacharacters. And to make things really funny: If you pass the `-E | --extended-regexp` to `grep`, it will also interpret all metacharacters as metacharacters!

However, this will become less confusing as soon as you start using regexes regularly. Try to do the following, and play around with the `-E` switch:

- Print all lines which contain at least on 'Z' or 'z'.
- Print all lines which contain two expressions enclosed in parentheses. Then try to print all lines where there are two expressions in parentheses which follow directly after each other. And finally try to print only those lines where the parentheses-expression have some text between them.
- Print all lines containing the word "fox" or the word "bash".
- Print all lines which consist exclusively of 'a's.
- Print all lines which consist exclusively of 3 or 4 'a's.
- Print all lines which consist exclusively of at least 3 'a's.

Check you results by looking at `grepfile` with `cat` or `less`.

Now that you know some basic stuff about regexes and know the basics of `grep`, let's move on to two other programs which use regexes extensively:

`sed(1)` and `awk(1)`

These two utilities are much more powerful than those you encountered before, especially because `awk` is in fact a full-blown programming language, but we'll introduce only the very basics.

`sed` which is an acronym for "stream editor" is a tool with which you can edit a stream of textdata and print it to `stdout` (or redirect it to a file, of course). The usual way to invoke `sed` is:

```
sed [OPTON]... -e 'SCRIPT' [FILE]...
```

where `SCRIPT` is a command you pass to `sed` that tells it what to do. `sed` will then apply this command to each line of the input file. Sounds very confusing, but let's work through an example:

```
$ cat sedfile
This text talks about birds. Why? Because we all know that birds are
the best animals. All birds are much more beautiful and intelligent than
any other species (including humans). No one can ever doubt that birds
are superior. And one day, birds will take over the world! Thus, don't
ever mistreat a bird, it will get its revenge!
```

What utter nonsense! Birds?! Ridiculous! Surely it's moles who'll subdue the world in the near future! This text must be adjusted:

```
$ sed -e 's/bird\(s\)\?/mole\1/g' sedfile
This text talks about moles. Why? Because we all know that moles are
the best animals. All moles are much more beautiful and intelligent than
any other species (including humans). No one can ever doubt that moles
are superior. And one day, moles will take over the world! Thus, don't
ever mistreat a mole, it will get its revenge!
```

Much better! But how to dissect the command to `sed`?

`s` This is the command which means "substitute" whose syntax is

    `s/REGEX/REPLACEMENT/FLAGS`

    There are other commands, like `d` for delete or `a` for append something.

`/` The slash is used here as a delimiter character. This is just conventional and you can use any other character as a separator. Note, however, that in case you want your delimiter appear in the `REGEXP` or `REPLACEMENT` you have to escape it using a backslash.

`bird\(s\)\?` The pattern to match.[19]

`mole\1` The replacement. The `\1` refers to the first expression in parentheses in the pattern before, so if there is no 's', it will be empty and if there is, it will print one.

`g` This is a flag meaning "global". If you don't use it, `sed` only substitues the first match of the regex on a line and leaves the rest alone. Just repeat the command above but delete the `g` to see what happens.

Phew, that's a lot to congest. But once you get the hang of it, `sed` can be immensely useful.

A word of caution about redirection: Only in rare cases do you want to see the changes `sed` made just on the command-line; usually you want them to be stored somewhere and in many cases you want to replace the old file with the new content. Beware, the following will not work in the way you may expect:

[19] This regex is more complicated than it has to be, because we wanted to illustrate grouped expressions. Can you give a simpler version that works as well in this situation?

```
$ sed -e 's/bird\(s\)\?/mole\1/g' sedfile > sedfile
```

If you do this, `sedfile` will be empty afterwards. But why? When you invoke `sed` and tell it to redirect its output to `sedfile` the first thing it does is to open this file for writing and because you used the greater-than operator it will truncate the file (i.e. empty it) and only then starts working on the now empty file. If you used the >> operator, the changed text would be appended to the current file, but the old content would still be there – most likely also not, what you wanted. This leaves you with two possibilities:

1. Redirect the output to a temporary file and use `mv` afterwards. This has the big advantage that you can check your changes before making them final.[20]

2. Use `sed`'s `-i[SUFFIX] | --in-place[=SUFFIX]` option. This tells `sed` to edit files in place. If you supply a `SUFFIX`, `sed` will create a backup of your original file with this suffix, which you can use to restore the old file if something has gone wrong.

`sed` can also be used to print only certain lines from a file, regardless of any patterns. This can be done using the `p` command and the `-n` option: If a line doesn't match anything you specified in a command, by default `sed` will print this line unchanged. To suppress this behaviour and show you only those parts of the file that `sed` works on is what the `-n | --quiet | --silent` option is for. The `p` command is the "print" command and can be prepended by a line number, e.g.:

```
$ sed -n -e '1p' sedfile
This text talks about birds. Why? Because we all know that birds are
```

prints only the first line of `sedfile`. We'll use this in 4.5.4.

After `sed`, we'll have a quick glance at `awk`. The `awk` program is actually an implementation of the AWK programming language which owes its name to its creators Alfred Aho, Peter Weinberger, and Brian Kernighan, but after learning it, some people have concluded it must come from "awkward". We won't go into any details here, just be aware that it exists and look at the following example:

```
$ awk '{ print "line" NR ": " $0; }' sedfile
line1: This text talks about birds. Why? Because we all know that birds are
line2: the best animals. All birds are much more beautiful and intelligent than
line3: any other species (including humans). No one can ever doubt that birds
line4: are superior. And one day, birds will take over the world! Thus, don't
line5: ever mistreat a bird, it will get its revenge!
```

You can see that this command prepended the word "line", the line number, stored in the variable (see 4.7) `NR` and a space in front of each line. The current input line is stored in the variable `$0`.

If you are totally confused now: That's OK, `sed` and `awk` are rather advanced tools, but as they can be immensely useful it is great to know about them and what they do

[20] It is in general a good idea to first output `sed`'s results to `stdout` to check if the changes work as intended.

in general, so you can come back and learn them in-depth if you ever need to. There are whole books written just about these two programs and other books that just treat regexes, so there is *a lot* to learn, definitely more than we can cover in this introducory workshop.

4.5.4 Cutting and Joining

Before we leave you alone with textfiles, there are two more things to talk about. The first has to do with cutting stuff up and joining it together.

You'll often have files which consists of columns and lines, representing some kind of table. Have a look at `tablefile`:

```
$ cat tablefile
12.9 2.6 9.9 5.9 9.5 7.9 1.2
13.6 9.3 7.8 1.9 6.7 5.7 8.2
13.1 7.4 1.0 3.7 3.0 1.8 13.1
8.4 12.3 13.1 12.8 10.1 7.3 11.9
1.5 9.3 10.4 2.6 9.6 1.1 12.8
5.8 11.6 7.1 5.2 1.7 1.8 9.6
8.7 2.9 11.8 2.7 3.7 5.4 10.6
2.7 2.7 3.6 13.0 4.2 10.8 1.8
9.6 5.9 5.7 9.5 9.5 3.1 11.2
10.9 1.3 3.0 4.9 8.2 7.7 6.6
13.0 12.0 3.6 13.5 7.8 12.4 4.9
1.6 12.5 1.4 6.9 10.2 3.7 3.8
7.3 5.1 11.2 12.5 1.0 10.3 3.1
6.6 12.2 5.4 2.2 13.9 12.5 12.2
8.7 2.4 10.9 6.7 5.9 5.2 11.8
11.4 7.2 1.3 12.2 4.7 3.2 10.7
3.4 7.2 7.4 12.8 4.9 6.4 9.2
3.9 12.7 3.2 3.3 6.7 8.0 8.5
9.5 12.8 6.8 2.7 7.2 2.4 13.7
12.4 3.6 2.1 3.1 10.9 7.7 12.8
10.4 1.2 5.4 2.6 7.0 7.5 8.8
8.9 7.3 3.6 6.3 7.2 3.8 7.4
12.9 12.7 8.2 11.1 2.5 5.3 10.3
11.2 6.4 11.9 7.3 2.6 10.5 3.4
```

This file consists of 24 lines and each line has 7 colums. This could e.g. be the output of a program which took measurements for 1 week, every hour, e.g. the second value in the first column represents the measurement at 1am (we started at 00:00) on Monday. All nice and fine, but the file is almost unreadable for a human. This is, where `cut(1)` comes in: Its purpose is to cut out columns while leaving out others and its basic syntax is:

```
cut OPTION... [FILE]...
```

It has some very useful options:

-d | --delimiter=DELIM This tells `cut` by which character fields in the file are separated. In the case above they are separated by spaces, but it could be any other character as well.

-f | --fields=LIST This tells `cut` which fields it should output.

--output-delimiter=STRING If you want the output to be delimited differently than the input, you can use this option.

Thus, if you want just the the measurements of Wednesday, you can do:

```
$ cut -d" " -f3 tablefile
9.9
7.8
1.0
13.1
10.4
7.1
11.8
3.6
5.7
3.0
3.6
1.4
11.2
5.4
10.9
1.3
7.4
3.2
6.8
2.1
5.4
3.6
8.2
11.9
```

Or what about the files from Tuesday to Thursday:

```
$ cut -d" " -f2-4 --output-delimiter="^T|^T" tablefile
2.6     |       9.9     |       5.9
9.3     |       7.8     |       1.9
7.4     |       1.0     |       3.7
```

```
12.3    |       13.1    |       12.8
9.3     |       10.4    |       2.6
11.6    |       7.1     |       5.2
2.9     |       11.8    |       2.7
2.7     |       3.6     |       13.0
5.9     |       5.7     |       9.5
1.3     |       3.0     |       4.9
12.0    |       3.6     |       13.5
12.5    |       1.4     |       6.9
5.1     |       11.2    |       12.5
12.2    |       5.4     |       2.2
2.4     |       10.9    |       6.7
7.2     |       1.3     |       12.2
7.2     |       7.4     |       12.8
12.7    |       3.2     |       3.3
12.8    |       6.8     |       2.7
3.6     |       2.1     |       3.1
1.2     |       5.4     |       2.6
7.3     |       3.6     |       6.3
12.7    |       8.2     |       11.1
6.4     |       11.9    |       7.3
```

This looks much better![21] If you want to see only the measurements taken between 10am and 1pm, you can combine this with `sed` and pipes:

```
$ sed -n -e '11,14p' tablefile | cut -d" " -f2-4 --output-delimiter="^T|^T"
12.0    |       3.6     |       13.5
12.5    |       1.4     |       6.9
5.1     |       11.2    |       12.5
12.2    |       5.4     |       2.2
```

First we select the hours we need and then we cut out the columns for Tuesday, Wednesday, and Thursday.

Another thing you may want to do is joining two files which share a common field. To achieve this, you can use `join(1)`:

```
$ cat join1
Price ArtNr
10€ 1
20€ 2
100€ 3
400€ 4
$ cat join2
```

[21] The `^T` in the command-line above means a tab-character. You can reproduce it on the command-line by typing `CTRL-V` followed by a tab.

```
ArtNr Name
1 pendrive
2 cable
3 screen
4 laptop
$ join -1 2 -2 1 join1 join2
ArtNr Price Name
1 10€ pendrive
2 20€ cable
3 100€ screen
4 400€ laptop
```

So, you tell join that the second field in the first file (`-1 2`) and the first field in the second file (`-2 1`) are identical and it should use them to join the files. There are some other options and if you're interested play around a bit :)

4.5.5 Editors (a.k.a. Religions)

The last step on our journey through textfiles leads us into the fiercely disputed field of text editors. As the title of this section suggests, there are people who vehemtly insist that the editor they use is the only true one and all the others should be forbidden. It may well be editors about which the first flamewars were fought and they are fought until today with absolutely no progress or use.[22]

Our stance is: Taste differs and it's no good arguing about it, but we want you to be aware that you may encounter people who'll attack you just because you editor is not theirs.

One reason why editors may be such a emotional topic is that the really powerful ones take a considerable time to get used to and a lifetime to master. Thus, as soon as you learned one of those you may be really proud of you on the one side and feel like a total beginner if you encounter one of the others, which often leads to aversive reactions.

There are editor which can need a GUI while others can work directly within a terminal.

Common GUI editors on (GNU/)Linux systems are:

KWrite The KDE-Desktop's lightweight editor.

gedit The GNOME-Desktop's standard editor. If you left all the defaults intact when you installed the GUI of you VM at the beginning of the course, `gedit` will be installed on your system and just called "Text Editor".

There are gazillions of others – if you're interested look around and try some of them until you find something.

But as this course is about the command-line, we're emphazising on editors you can use in a terminal.

[22] You think we're joking? Have a look at this.

nano This very small and intuitive editor is now installed by default on many (GNU/)Linux systems.

vi(m) `vi` pronounced "Vee-Eye" and `vim` which stands for "Vi IMproved" and thus just pronounced "vim" are one of the main editors used by many programmers and sysadmins alike. They are incredibly powerful and have tons of features.

emacs The archrival to `vi(m)` for a long time, this giant of an editor is often described as "actually an operating system which happens to include a text editor". If you put in the effort to learn to use it efficiently, you'll be very productive when working with text.

Instead of talking about features of these editors, we'll let you learn them by their own means. First do the following:

```
$ sudo apt install vim emacs
```

This will install both editors. Now choose which one you want to try out first.
If it is `vim`, type:

```
$ vimtutor
```

And you will be in an interactive tutorial, showing you the basics of `vim`.
In case you chose emacs, type:

```
$ emacs
```

After the editor started, type `CTRL-H` followed by `T`. The tutorial will start.

We won't say anything else about editors; it is important to know how to use one of them, but which one doesn't really matter. Try a few, play around and stick with what you like most :)

4.6 Processes

After so much textfiles, we start with something completely different. You have worked a bit with commands by now, and maybe you wondered at some point, what they are. In very general terms we speak of a program as a executable binary that can be run on a CPU. Whenever you enter a command, the corresponding binary is loaded into memory and then each instruction is executed by the CPU. As soon as a program runs, it is called a *process*, i.e. a process is a program in execution. To make the difference a bit clearer: You have only one program found at `/bin/ls` but you can run it in different terminals at the same time, thus creating multiple processes from a single program.

If you want to see the processes currently running on your system the most widely used tools are `ps(1)` and `top(1)`. Their main difference is that `top` is interactive while `ps` is not. To view all proces that are currently running from the terminal you are working on, you can do:

```
$ ps
  PID TTY          TIME CMD
 2351 pts/4    00:00:00 bash
 4963 pts/4    00:00:00 ps
```

This tells you that there are just two processes associated with this terminal: your `bash` and the `ps` process which has already finished as soon as you can read the output. The displayed fields have the following meaning:

`PID` This is the "Process ID". Each process is assigned a number that is unique on the currently running system, to enable the system (and you) to refer to it unambiguously – the name alone would not be sufficient.

`TTY` This tells you with which terminal[23] the process is associated. As you are working in a terminal emulator and not on a real terminal the "pts" in the output means "pseudo-terminal".

`TIME` This tells you how many CPU time the process has used cumulatively.

`CMD` Rather obviously, this is the command's name.

To see all processes currently running on the system, use:

```
$ ps -elf
```

This creates a listing of all processes (`-e`) in a long (`-l`) and full (`-f`) format to show a lot of extra information.[24] We don't print it here, as the output is almost always more than 100 lines.

`ps` has a myriad of options and tweaks available, so it is highly recommended to read the man page. Additionally `ps` is a very old program and has acquired a lot of legacy-stuff during the time. Thus, there are many options which do almost-but-not-exactly the same, there are options with no dashes, one dash, or two dashes, etc. Therefore, reading the man page in this case also provides you with a glance of history.

`top` is also a very old program, but it is still very useful. If you start it the process will fill your screen and update periodically. The first will look something like this:

```
top - 14:11:33 up  4:48,  6 users,  load average: 0.02, 0.02, 0.00
```

This tells you the current time, how long the system is running, how many users are currently logged in and the load average. Note that a load average of 1 means one CPU is fully loaded, i.e. if your computer has more than one CPU or more than one core, a load of more than 1 is no problem.

By default, processes are sorted according to the `%CPU` column, which tells you how much CPU they are using.

[23] TTY stands for **t**ele**ty**pewriter, a rather old acronym.

[24] You'll find the options `aux` (without a dash!) doing something very similar in a lot of forums as these are the traditional options. However, `-elf` should be used today.

To quit the program type `Q`.

Again it is a good idea to read the man page to get acquainted with the many things you can do with `top`.

The processes you've seen so far run for only a very short time, e.g. the ouput of `ls` appears immediately and you can issue the next command. However, in 4.5.1 we already encountered a program which didn't return to command-line on its own, i.e. when we used `tail -f`. While this programs runs until you tell it to stop, there are other programs which just take a long time and it may be very annoying if they block your command-line all the time. Consider starting the Firefox web browser from the command-line: It would be pretty bad if you had to choose between browsing and working.

Thus, there are a lot of tools to control processes. First of all, there are keyboard shortcuts which send signal to processes. Signals tell a process what to do, e.g. `SIGTERM` tells the process to terminate. The keyboard shortcuts you can use are:

`CTRL-C`	Send an interrupt signal (`SIGINT`) to the process currently blocking the command-line. Usually, this leads to the process being terminated, as you learned in 4.5.1 when you stopped the `tail -f` process.
`CTRL-Z`	Sends a stop signal (`SIGTSTP`) to a process. A stopped process won't react to anything, unless you send it a `SIGCONT` (for "continue") signal.

Some processes just need a long time or even don't have a defined point in time when they stop, e.g. compiling a large program takes a long time and your web browser should only stop when you tell it to do so. How can you use such programs without sacrificing your command-line? The easiest way is to start them with an ampersand appended on the command-line, e.g.

```
$ firefox &
[1] 5133
$
[1]+  Done                    firefox
$
```

As you can see, Firefox is started, the shell tells you that it has job number 1 and PID 5133. The ampersand tells the shell to start the process in the background and the next time you press `ENTER` it will tell you that the job has finished. However, if you could e.g. start a long backup job, put it in the background and the shell will inform you as soon as its done, even if this takes hours or days (provided you don't shut down your computer or close the shell during this time as the job would be killed then).

The job number is also a unique number; not on the whole system, but just on the current shell. As it is usually much smaller than a PID it is also easier to remember for humans. To have a look how many jobs are currently associated with your `bash`, you can use the `jobs` shell-builtin.

4.6.1 Process Control

Now that you know how to find out which processes are running, either on the system or within your shell session and you know that you can start them in the background and are able to send two signals to them using the keyboard.

However, there is more you can do with processes. The `bash` offers some builtin functionality for this:

If you have a job running in the background, but you want it back on your shell, you can use the `fg` builtin, e.g.

```
$ xclock &
[1] 5257
$ jobs
[1]+  Running                 xclock &
$ fg 1
xclock
^Z
[1]+  Stopped                 xclock
$ bg 1
[1]+ xclock &
$ ps
  PID TTY          TIME CMD
 5130 pts/5    00:00:00 bash
 5257 pts/5    00:00:00 xclock
 5264 pts/5    00:00:00 ps
$ kill 5257
$
[1]+  Terminated              xclock
```

In this example you start an `xclock` in the background. Then you put it into the foreground using `fg 1` where 1 is the job ID of the process. After this, you lost access to your shell. The `^Z` in the output above indicates pressing `CTRL-Z`, i.e. sending a `SIGTSTP` to the process. After this you'll realise that `xclock` will no longer interact with you – the process is frozen. To let it continue in the background, you use the `bg` builtin. Afterwards you find out `xclock`'s PID and send it a signal using `kill`.[25]

`kill` is one of the most important tools when doing process control. Read its man page as well as `signal(7)` to understand what it does and how.

[25] This is a bit of a mess: There is a program called `kill(1)` which will be explained if you type "`man kill`". However, consider the following scenario: You have opened so many processes that you are not allowed to start new ones and some of them are no longer reacting, so you have to terminate them by sending them signals. However, `kill(1)` is a program and if you try to invoke it, this will be refused as you're not allowed to start any more processes. Therefore, there is a `bash`-builtin called `kill` which does exactly the same, but as it is just a part of `bash` it does not need to start a whole new process and will still be able to terminate the misbehaving ones. Thus, if you just type `kill` on the command-line the `bash`-builtin will be used and not the program `kill(1)`. This has the further advantage that you can use job IDs instead of PIDs. To tell `kill` that you're passing a job ID to it, prepend the number with '%', i.e. in the above example: `kill %1`.

4.7 Variables and Your Environment

This will be covered in the course and added to the script at a later point.