Introduction to Linux

Part I

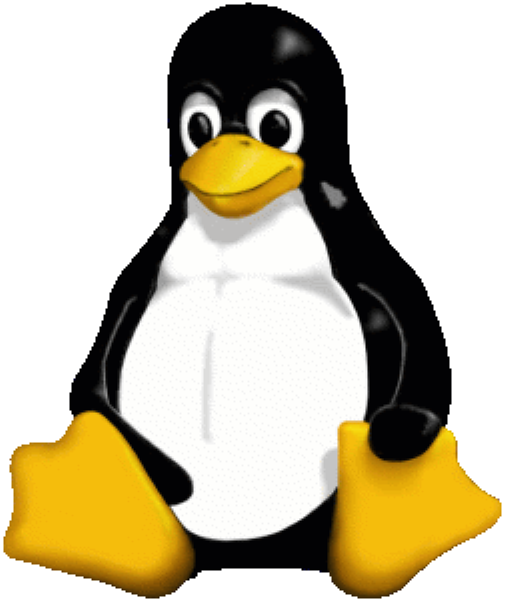
`https://goo.gl/Vg3iXW`

v.1.2 (2020-01-20)

Agenda

1. What is Linux?
2. Linux interface: GUI vs CLI
3. Connecting to a remote Linux system
4. Linux directory structure
5. Moving and looking around
6. Reading and writing files
7. Organizing files and folders
8. Using wildcards and braces

What is Linux?



What is Linux?

What is Linux?

The most common answer you'll hear is:

"Linux is an operating system"

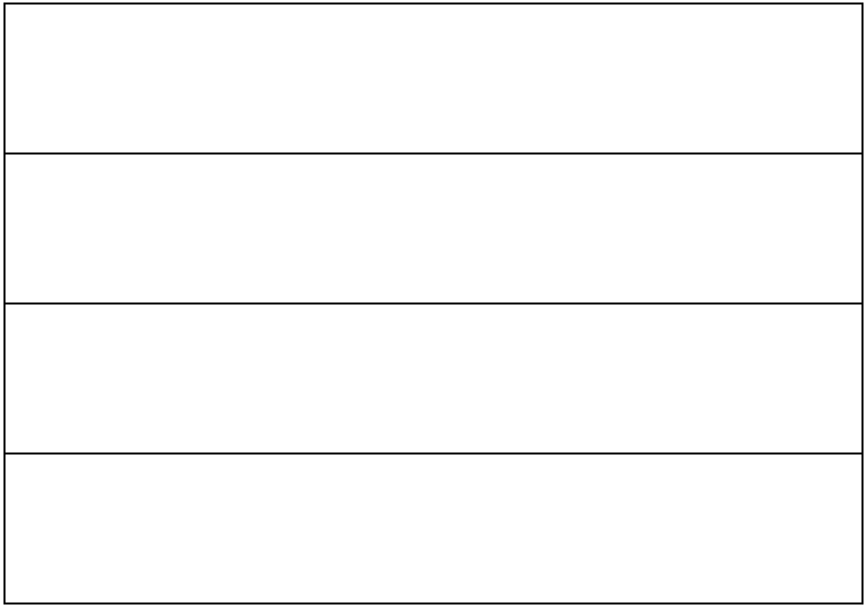
What is Linux?

The most common answer you'll hear is:

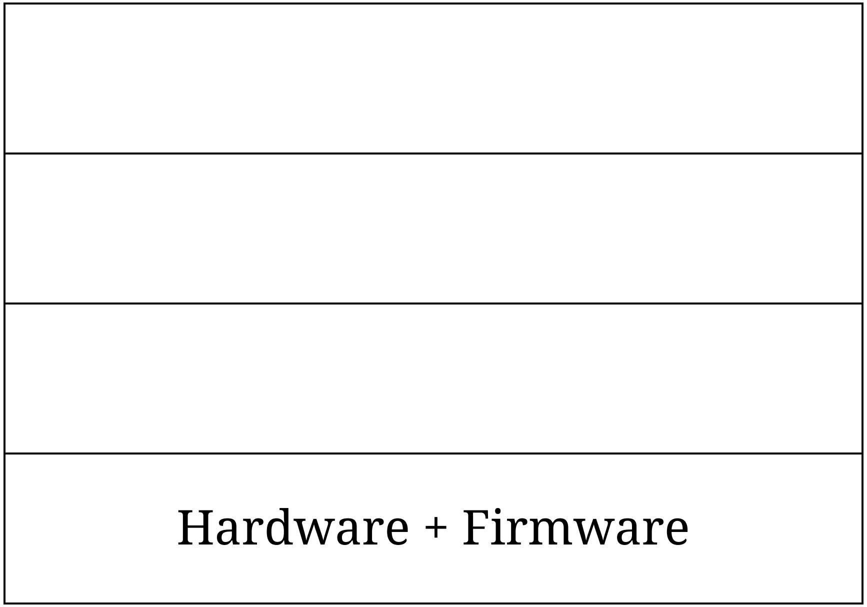
"Linux is an operating system"

But what does this mean?

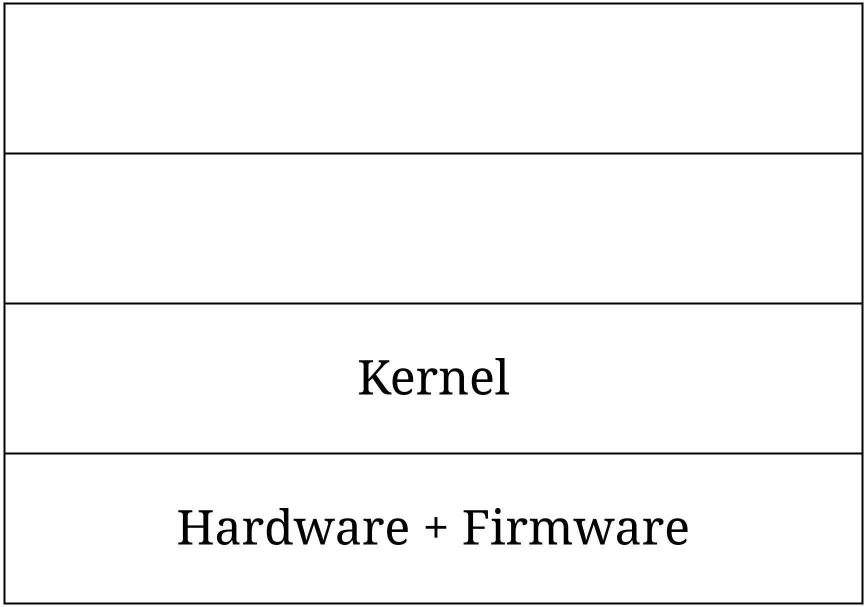
Operating systems



Operating systems



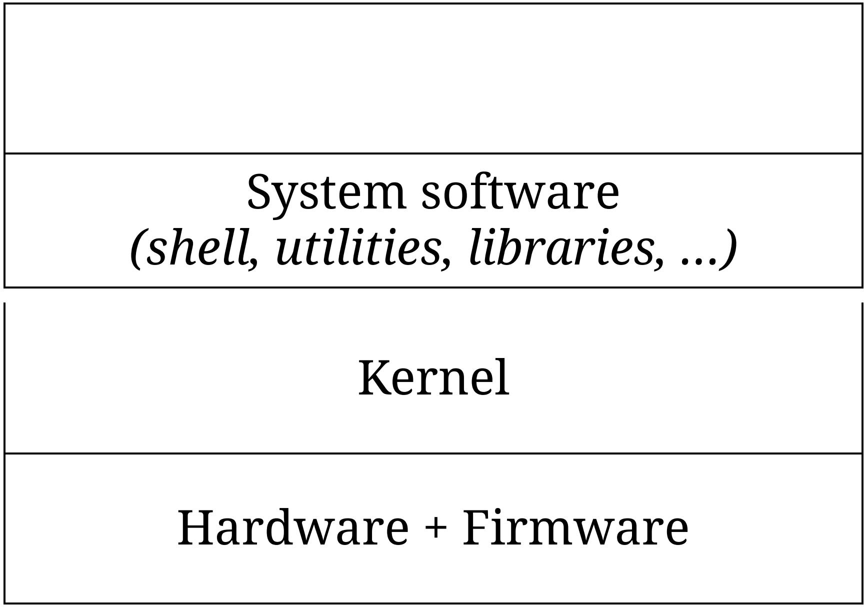
Operating systems



Kernel

Hardware + Firmware

Operating systems

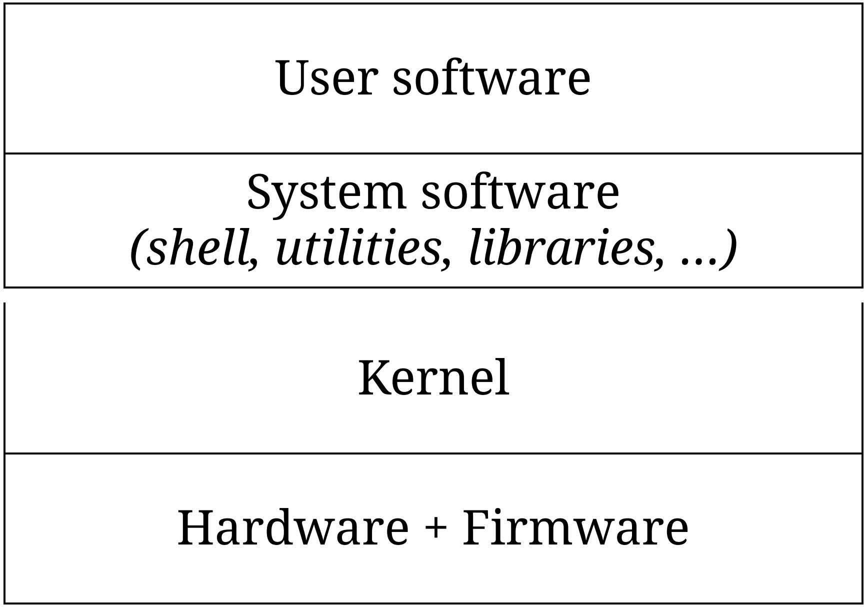


System software
(shell, utilities, libraries, …)

Kernel

Hardware + Firmware

Operating systems



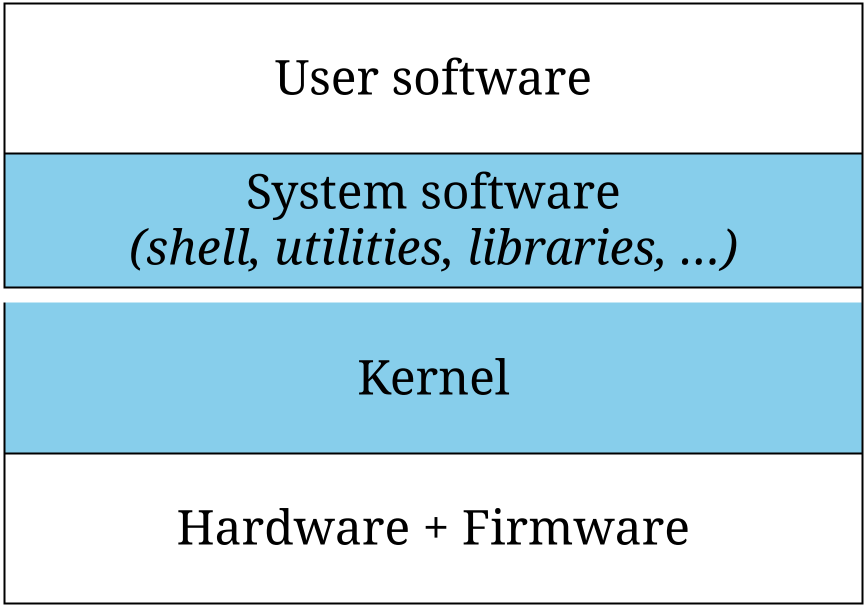
User software

System software
(shell, utilities, libraries, …)

Kernel

Hardware + Firmware

Operating systems



User software

System software

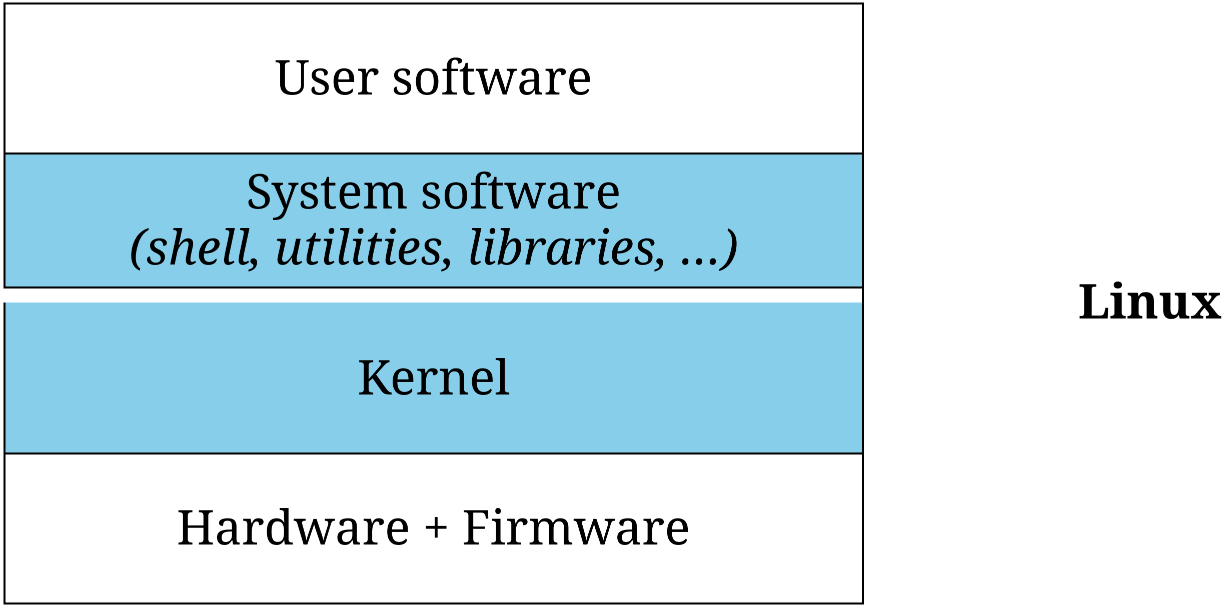
(shell, utilities, libraries, ...)

Kernel

Hardware + Firmware

Operating system
Windows, Linux,
MacOS, Android, ...

Operating systems



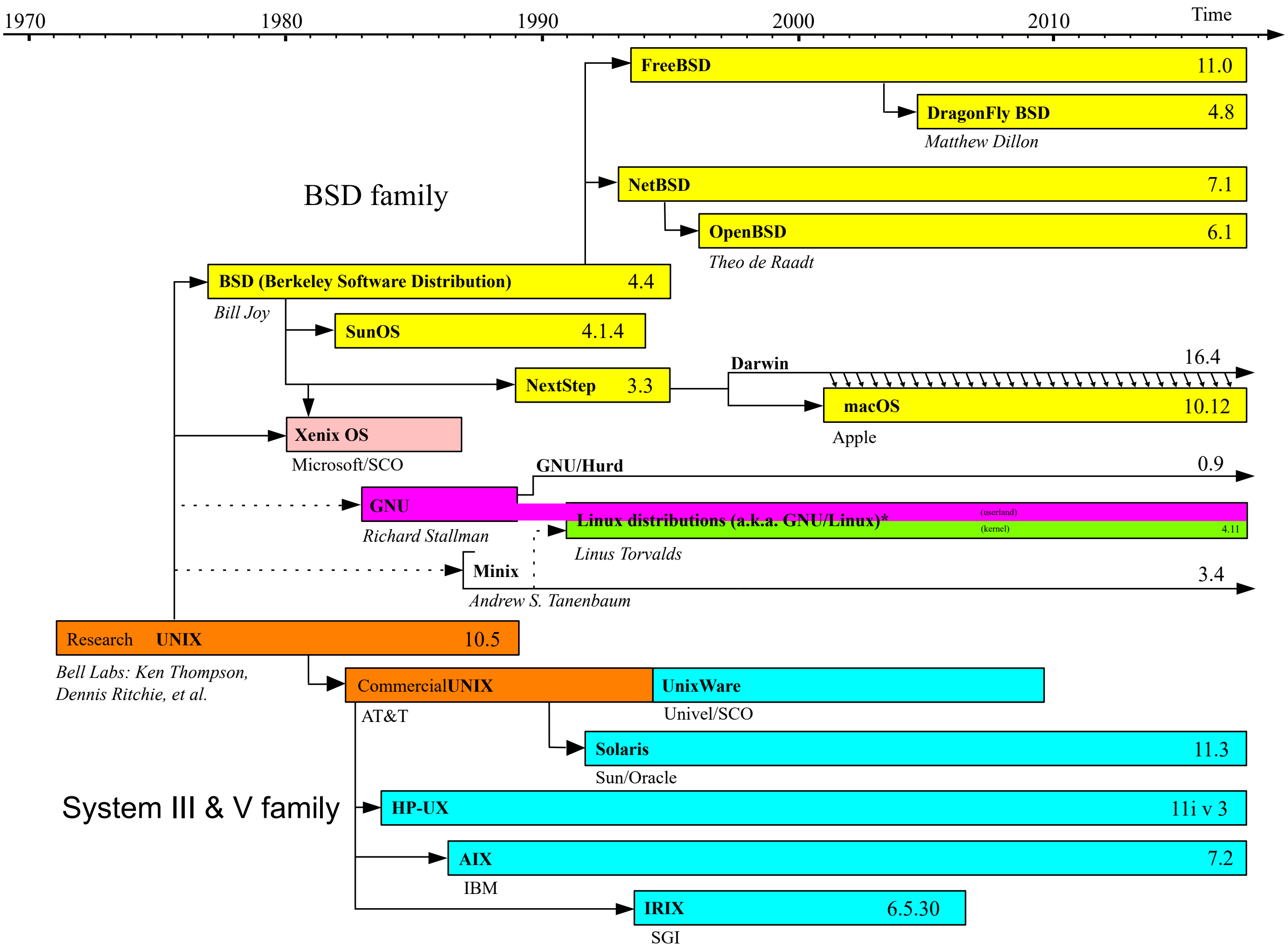
In practice, we call this part "Linux"

Linux? Wait, I also heard "UNIX"?

UNIX is the name of an operating system from 1970 that pioneered concepts that will form the basis of Linux (and other OSes) today.

More importantly, it introduced a set of conventions that its descendents follow. A system that follows them is called "UNIX-like".

Most of what you learn here will easily transfer to other UNIX-like OSes (e.g. macOS).



1970 1980 1990 2000 2010 Time

FreeBSD 11.0

DragonFly BSD 4.8

Matthew Dillon

BSD family

NetBSD 7.1

OpenBSD 6.1

Theo de Raadt

BSD (Berkeley Software Distribution) 4.4

Bill Joy

SunOS 4.1.4

Darwin 16.4

NextStep 3.3

macOS 10.12

Apple

Xenix OS

Microsoft/SCO

GNU/Hurd 0.9

GNU

Richard Stallman

Linux distributions (a.k.a. GNU/Linux)* (userland) (kernel) 4.11

Linus Torvalds

Minix 3.4

Andrew S. Tanenbaum

Research UNIX 10.5

Bell Labs: Ken Thompson, Dennis Ritchie, et al.

Commercial UNIX

AT&T

UnixWare

Univel/SCO

Solaris 11.3

Sun/Oracle

System III & V family

HP-UX 11i v 3

AIX 7.2

IBM

IRIX 6.5.30

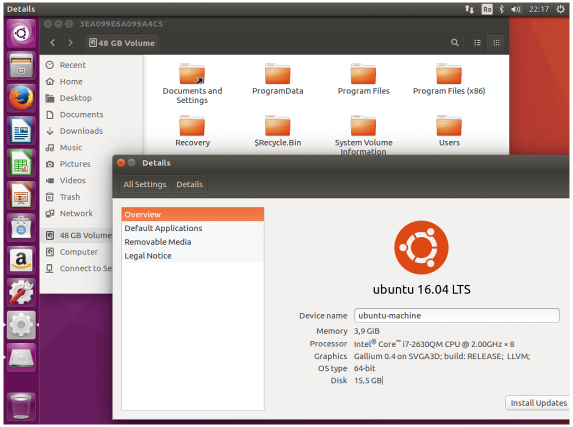
SGI

*The penetration of GNU utilities varies between distributions, some projects use GNU's implementation of the Linux kernel (Linux-libre). Some operating systems mentioned here include GNU utilities to a lesser degree.

User Interface

GUI

Graphical Interface

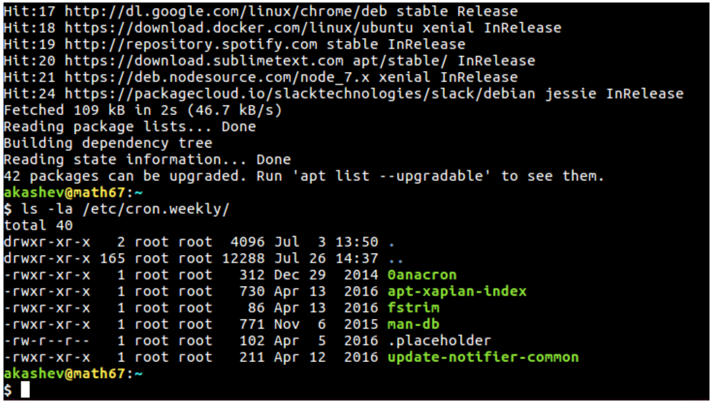


ubuntu 16.04 LTS

Device name ubuntu-machine
Memory 3,9 GiB
Processor Intel® Core™ i7-2630QM CPU @ 2.00GHz × 8
Graphics Gallium 0.4 on SVGA3D; build: RELEASE; LLVM;
OS type 64-bit
Disk 15,5 GB

CLI

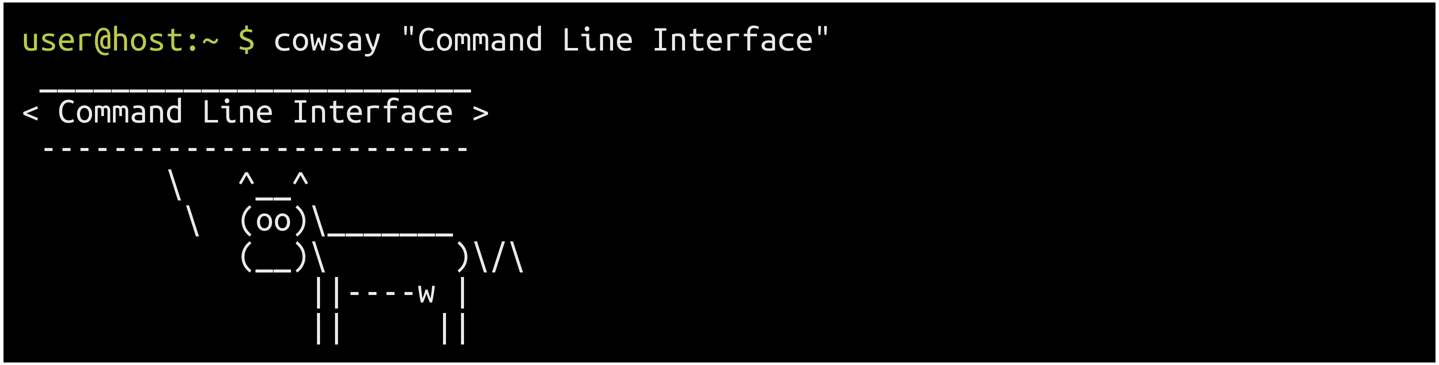
Command Line



```
Hit:17 http://dl.google.com/linux/chrome/deb stable Release
Hit:18 https://download.docker.com/linux/ubuntu xenial InRelease
Hit:19 http://repository.spotify.com stable InRelease
Hit:20 https://download.sublimetext.com apt/stable/ InRelease
Hit:21 https://deb.nodesource.com/node_7.x xenial InRelease
Hit:24 https://packagecloud.io/slacktechnologies/slack/debian jessie InRelease
Fetched 109 kB in 2s (46.7 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
42 packages can be upgraded. Run 'apt list --upgradable' to see them.
akashev@math67:~
$ ls -la /etc/cron.weekly/
total 40
drwxr-xr-x   2 root root  4096 Jul  3 13:50 .
drwxr-xr-x 165 root root 12288 Jul 26 14:37 ..
-rwxr-xr-x   1 root root   312 Dec 29  2014 0anacron
-rwxr-xr-x   1 root root   730 Apr 13  2016 apt-xapian-index
-rwxr-xr-x   1 root root    86 Apr 13  2016 fstrim
-rwxr-xr-x   1 root root   771 Nov  6  2015 man-db
-rw-r--r--   1 root root   102 Apr  5  2016 .placeholder
-rwxr-xr-x   1 root root   211 Apr 12  2016 update-notifier-common
akashev@math67:~
$ 
```

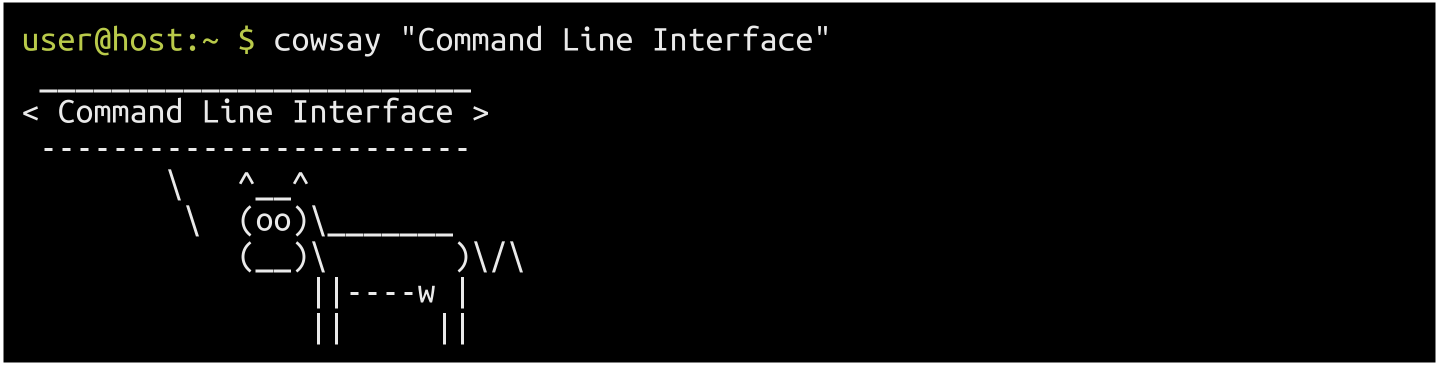
Some synonyms:
"Shell", "Terminal", "TTY"

Command Line Interface



```
user@host:~ $ cowsay "Command Line Interface"
 ________________________
< Command Line Interface >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Command Line Interface



```
user@host:~ $ cowsay "Command Line Interface"
 ________________________
< Command Line Interface >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

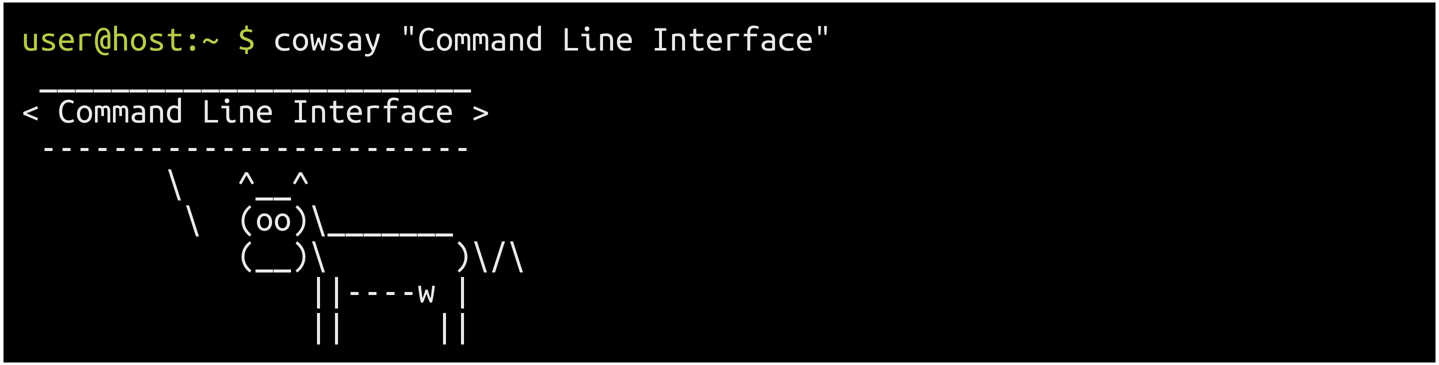
- Input, output, and commands are **text**.

Command Line Interface

```
user@host:~ $ cowsay "Command Line Interface"
 ________________________
< Command Line Interface >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- Input, output, and commands are **text**.
- Easy on the computer: can run on any hardware.

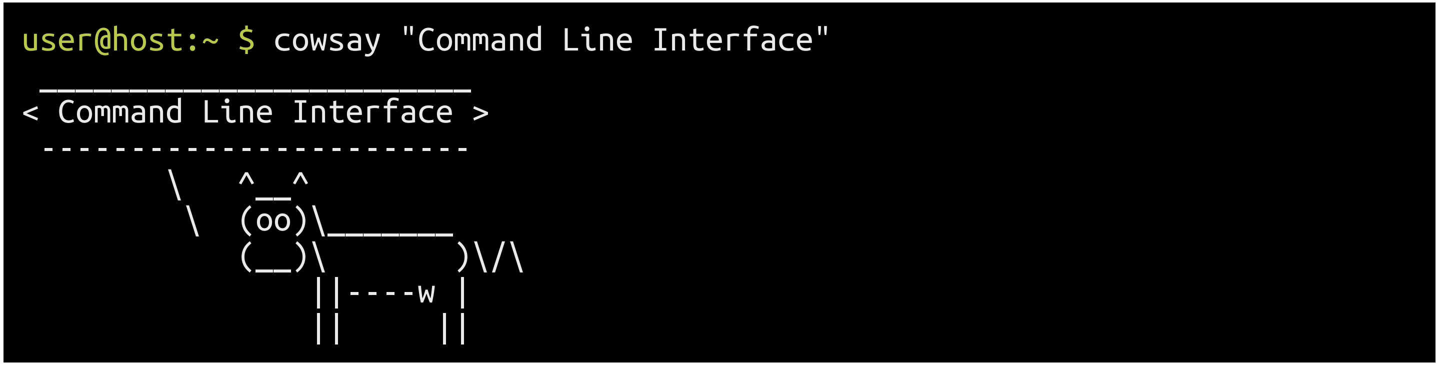
Command Line Interface



```
user@host:~ $ cowsay "Command Line Interface"
 ________________________
< Command Line Interface >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- Input, output, and commands are **text**.
- Easy on the computer: can run on any hardware.
- Network-friendly: a few bytes of text vs realtime stream of images / GUI updates => tool of choice for remote access.

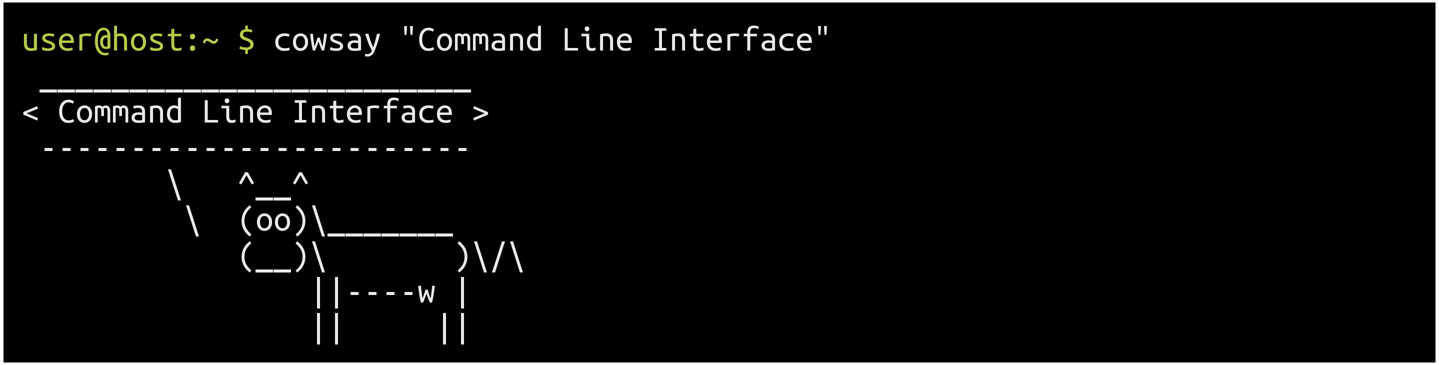
Command Line Interface



```
user@host:~ $ cowsay "Command Line Interface"
 ________________________
< Command Line Interface >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- Input, output, and commands are **text**.
- Easy on the computer: can run on any hardware.
- Network-friendly: a few bytes of text vs realtime stream of images / GUI updates => tool of choice for remote access.
- Scripting/automation-friendly: text is easier to manipulate.

Command Line Interface



```
user@host:~ $ cowsay "Command Line Interface"
 ________________________
< Command Line Interface >
 ------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- Input, output, and commands are **text**.
- Easy on the computer: can run on any hardware.
- Network-friendly: a few bytes of text vs realtime stream of images / GUI updates => tool of choice for remote access.
- Scripting/automation-friendly: text is easier to manipulate.
- Expert-friendly, but *beginner-unfriendly*.

Connecting to a remote Linux system

The standard tool to connect to a remote system is `ssh`.

Acronym:

SSH: **S**ecure **Sh**ell

It securely connects you to a remote system.
Communication is encrypted, both parties are authenticated.

First, you will need to log in to the system.

If your credentials are accepted, it creates a new shell for you.

It is then displayed on your screen and controlled by your keyboard, relayed over the network.

Connecting from MacOS / Linux / Win10:

Good news: you already have a terminal and `ssh` of your own!

Connecting from MacOS / Linux / Win10:

Good news: you already have a terminal and `ssh` of your own!

First, open the terminal:

- For MacOS, it's accessible from Launchpad, Utilities.
- For Linux GUI, usually look for a program called Terminal.
- For Windows 10, open Command Prompt.

Connecting from MacOS / Linux / Win10:

Good news: you already have a terminal and `ssh` of your own!

First, open the terminal:

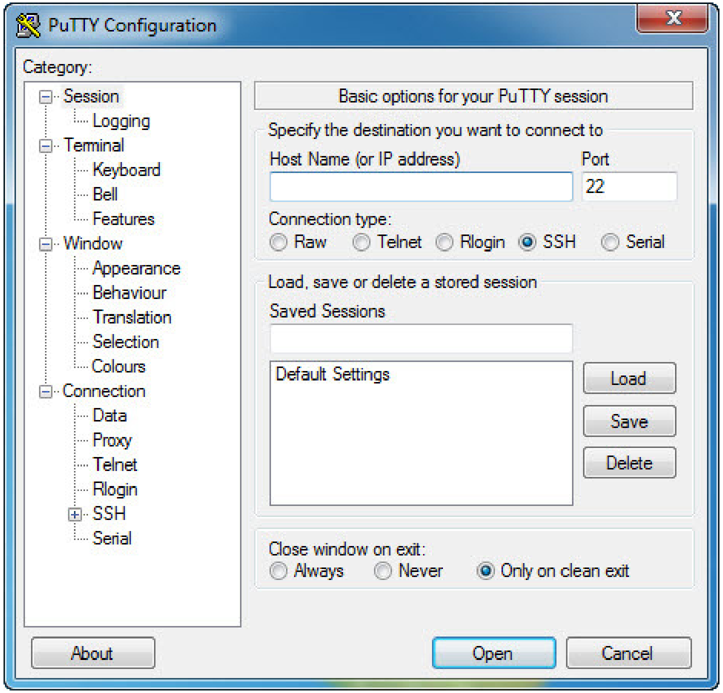
- For MacOS, it's accessible from Launchpad, Utilities.
- For Linux GUI, usually look for a program called Terminal.
- For Windows 10, open Command Prompt.

Then, you need to input the command to connect to a remote host:

```
local.user@local:~ $ ssh user@remote

[..some mutual* authentication later..]

user@remote:~ $
```

Connecting from Windows:

You will need an SSH client. Standard one: PuTTY



PuTTY Configuration

Category:

- Session
  - Logging
- Terminal
  - Keyboard
  - Bell
  - Features
- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection
  - Data
  - Proxy
  - Telnet
  - Rlogin
  - SSH
  - Serial

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address):

Port: 22

Connection type:
○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit:
○ Always ○ Never ● Only on clean exit

Download the appropriate installer: https://goo.gl/pHFReU

Connecting from Windows:

- Make sure "Connection type: SSH" is selected.
- Put the remote's host name / IP in the form.
- Select "Open"

A terminal window will open..

```
[..some mutual* authentication later..]

user@remote:~ $
```

Mutual authentication?

(and what's up with this side picture?)

Mutual authentication?

SSH authenticates both parties:

- Client to server
  - Username + password
  - Username + cryptographic key
  - Something else!
- Server to client
  - The server has a cryptographic key to prove its identity

Mutual authentication?

SSH authenticates both parties:

- Client to server
  - Username + password
  - Username + cryptographic key
  - Something else!
- Server to client
  - The server has a cryptographic key to prove its identity

The first time you connect, you need to explicitly say you trust the (previously unknown) server.

On subsequent connections, SSH will verify that you are still connecting to a server with the same key, and will warn you before login credentials are transmitted if you aren't.

Mutual authentication?

SSH authenticates both parties:

- Client to server
  - Username + password
  - Username + cryptographic key
  - Something else!
- Server to client
  - The server has a cryptographic key to prove its identity

The first time you connect, you need to explicitly say you trust the (previously unknown) server.

On subsequent connections, SSH will verify that you are still connecting to a server with the same key, and will warn you before login credentials are transmitted if you aren't.

This is called TOFU (**T**rust **O**n **F**irst **U**se).

Mutual authentication

So, the first time you connect to a new server, you should *expect* a warning you need to confirm:

In Linux/MacOS:

```
local.user@local:~ $ ssh user@remote
The authenticity of host 'remote (11.22.33.44)' can't be established.
ECDSA key fingerprint is SHA256:eQZbiUM4qV6ptjc0fN6/pFglj45qaNlXbLCULCTzSGM.
Are you sure you want to continue connecting (yes/no)?
```

Mutual authentication

So, the first time you connect to a new server, you should *expect* a warning you need to confirm:

In Linux/MacOS:

```
local.user@local:~ $ ssh user@remote
The authenticity of host 'remote (11.22.33.44)' can't be established.
ECDSA key fingerprint is SHA256:eQZbiUM4qV6ptjc0fN6/pFglj45qaNlXbLCULCTzSGM.
Are you sure you want to continue connecting (yes/no)? yes
```

Mutual authentication

So, the first time you connect to a new server, you should *expect* a warning you need to confirm:

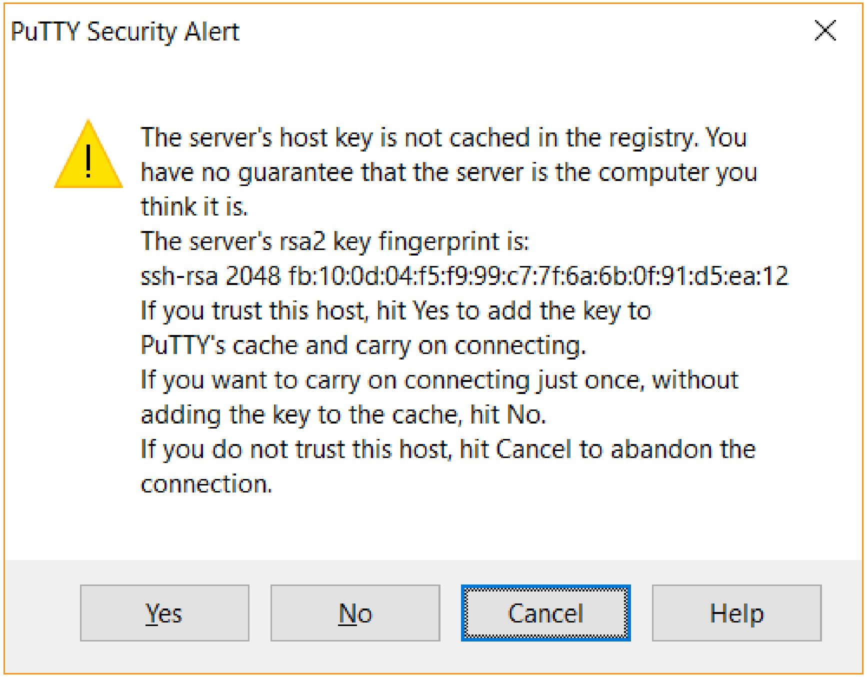
In Linux/MacOS:

```
local.user@local:~ $ ssh user@remote
The authenticity of host 'remote (11.22.33.44)' can't be established.
ECDSA key fingerprint is SHA256:eQZbiUM4qV6ptjc0fN6/pFglj45qaNlXbLCULCTzSGM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'remote,11.22.33.44' (ECDSA) to the list of known
hosts.

[..some client authentication later..]

user@remote:~ $
```

Mutual authentication

So, the first time you connect to a new server, you should *expect* a warning you need to confirm:

In Windows/PuTTY:



PuTTY Security Alert

The server's host key is not cached in the registry. You have no guarantee that the server is the computer you think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 fb:10:0d:04:f5:f9:99:c7:7f:6a:6b:0f:91:d5:ea:12
If you trust this host, hit Yes to add the key to PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without adding the key to the cache, hit No.
If you do not trust this host, hit Cancel to abandon the connection.

Hands-on time: Connect to a server

Using your Campus account username/password, use SSH/PuTTY to connect to UBELIX at `submit.unibe.ch`.

```
ssh user@submit.unibe.ch
```

If you're using the provided training VM, connect to it using the credentials given:

```
ssh trainingNN@12.34.56.78
```

(Substitute the user names / IP with real ones)

A reminder, PuTTY can be obtained from https://goo.gl/pHFReU

Greetings from a shell

```
username@remote:~ $
```

Greetings from a shell

After connecting, you will be greeted with something like this:

```
user@remote:~ $
```

Greetings from a shell

After connecting, you will be greeted with something like this:

```
user@remote:~ $
```

What you see is the interface of the **shell**: a text-based interface that allows you to launch other programs with commands.

Greetings from a shell

After connecting, you will be greeted with something like this:

```
user@remote:~ $
```

What you see is the interface of the **shell**: a text-based interface that allows you to launch other programs with commands.

Mnemonic:

It's called a shell **prompt** since it's **prompting** you to enter a command.

The prompt contains a short summary of current state of the shell.

Anatomy of a prompt

The prompt looks like this:

```
user@remote:~ $
```

This may vary slightly from system to system, and is fully configurable, but this is the typical form.

Anatomy of a prompt

The prompt looks like this:

```
user@remote:~ $
```

This may vary slightly from system to system, and is fully configurable, but this is the typical form.

This form of the prompt answers 3 questions:

- Who are you? **Username `user`**
- Where are you? **Hostname `remote`**
- Where in the filesystem are you? **`~`** (explained later)

Anatomy of a prompt

The prompt looks like this:

```
user@remote:~ $
```

This may vary slightly from system to system, and is fully configurable, but this is the typical form.

This form of the prompt answers 3 questions:

- Who are you? **Username `user`**
- Where are you? **Hostname `remote`**
- Where in the filesystem are you? `~` (explained later)

Terminating the prompt is (traditionally) a `$` character: it delimits where your input goes.

Taking command

The shell expects a textual command; most of the time you type the command and press [ENTER] to commit it.

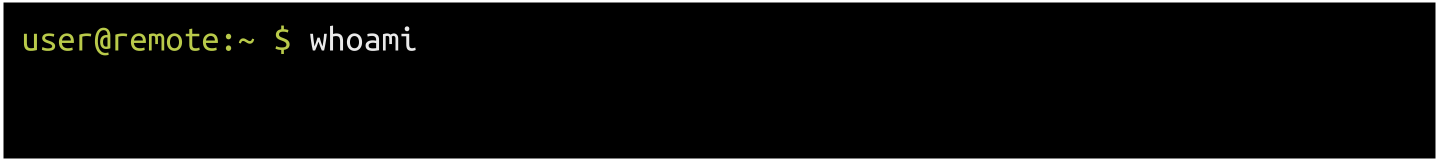
Let's try this (in slow motion)!

```
user@remote:~ $
```

Taking command

The shell expects a textual command; most of the time you type the command and press [ENTER] to commit it.

Let's try this (in slow motion)!



```
user@remote:~ $ whoami
```

1. Typing in "whoami" as the shell waits

Taking command

The shell expects a textual command; most of the time you type the command and press [ENTER] to commit it.

Let's try this (in slow motion)!

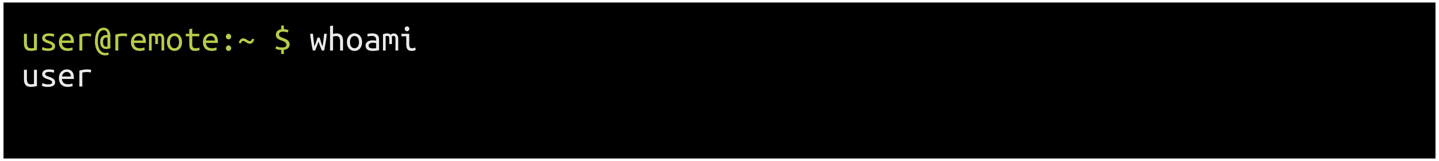
```
user@remote:~ $ whoami
```

1. Typing in "whoami" as the shell waits
2. Pressing [ENTER]. The shell will process the command (launch the program `whoami`)

Taking command

The shell expects a textual command; most of the time you type the command and press [ENTER] to commit it.

Let's try this (in slow motion)!



```
user@remote:~ $ whoami
user
```

1. Typing in "whoami" as the shell waits.
2. Pressing [ENTER]. The shell will process the command — launch the program `whoami`.
3. The program will take over input/output — in this case, it will output your username).

Taking command

The shell expects a textual command; most of the time you type the command and press [ENTER] to commit it.

Let's try this (in slow motion)!

```
user@remote:~ $ whoami
user
user@remote:~ $
```

1. Typing in "whoami" as the shell waits.
2. Pressing [ENTER]. The shell will process the command — launch the program `whoami`.
3. The program will take over input/output — in this case, it will output your username).
4. The program terminates, and control returns to the shell; it shows a new prompt.

Try it!

Here's a few commands for you to try:

```
whoami

echo Hello!

pwd

ls -l

date

sleep 3

clear

history 5
```

Each should do something and return you to the shell prompt. Can you guess what they do?

Note that you can use up/down arrows to access/repeat previous commands.

Safety first, or emergency exits!

So far every command we encountered automatically returned control back to the shell.

But what if a program is stuck, or expecting some input and you're not sure what to do?

Typical shortcuts to stop / quit a program:

- `Ctrl + C` (also called **interrupt**)
- `Esc` (from "**esc**ape")
- `q` (from "**q**uit")
- `Ctrl + D` (**end of input**, in case a program is waiting)

If you try those, usually you'll either exit the program or get some hint on how to do it.

`Ctrl` is sometimes denoted as `^`, e.g. `^C` for `Ctrl+C`.

Anatomy of a shell command

The shell expects input. What does it (typically) look like?

```
user@remote:~ $ program -f --option abc 123
```

Here, `program` is the **command** being executed, and the rest is the list of its **arguments**:

1. `-f`
2. `--option`
3. `abc`
4. `123`

Arguments that start with `-` or `--` are often called **flags** or **switches** and traditionally change some options of the command.

The Working Directory

One of the commands you executed, `pwd`, printed a directory path (of your home directory, by default):

```
akashev@submit01:~ $ pwd
/home/ubelix/math/akashev
```

The Working Directory

One of the commands you executed, `pwd`, printed a directory path (of your home directory, by default):

```
akashev@submit01:~ $ pwd
/home/ubelix/math/akashev
```

Mnemonic:

`pwd` stands for **P**rint **W**orking **D**irectory

The Working Directory

One of the commands you executed, `pwd`, printed a directory path (of your home directory, by default):

```
akashev@submit01:~ $ pwd
/home/ubelix/math/akashev
```

Mnemonic:

`pwd` stands for **P**rint **W**orking **D**irectory

Whenever you use the shell, there is a concept of the current (or "working") directory. This affects how commands search for files and how they interpret paths.

Think of it as of "where" you are: if a server is a building you're in, a working directory is the room you're in within that building.

The Working Directory

One of the commands you executed, `pwd`, printed a directory path (of your home directory, by default):

```
akashev@submit01:~ $ pwd
/home/ubelix/math/akashev
```

Usually, this information is printed in the shell prompt itself, to remind you of the current state.

The Working Directory

One of the commands you executed, `pwd`, printed a directory path (of your home directory, by default):

```
akashev@submit01:~ $ pwd
/home/ubelix/math/akashev
```

Usually, this information is printed in the shell prompt itself, to remind you of the current state.

In this example it's `~`, which represents the **home directory**.

The Working Directory

One of the commands you executed, `pwd`, printed a directory path (of your home directory, by default):

```
akashev@submit01:~ $ pwd
/home/ubelix/math/akashev
```

Usually, this information is printed in the shell prompt itself, to remind you of the current state.

In this example it's `~`, which represents the **home directory**.

Here's how it would look if you were somewhere else, for example in `/var/log`:

```
akashev@submit01:/var/log $
```

UNIX directory structure

If you're reading this tutorial, you likely already know that files are normally organized into nested "directories" (or "folders"). For example, on Windows you may have such a path:

```
C:\folder\subfolder\file
```

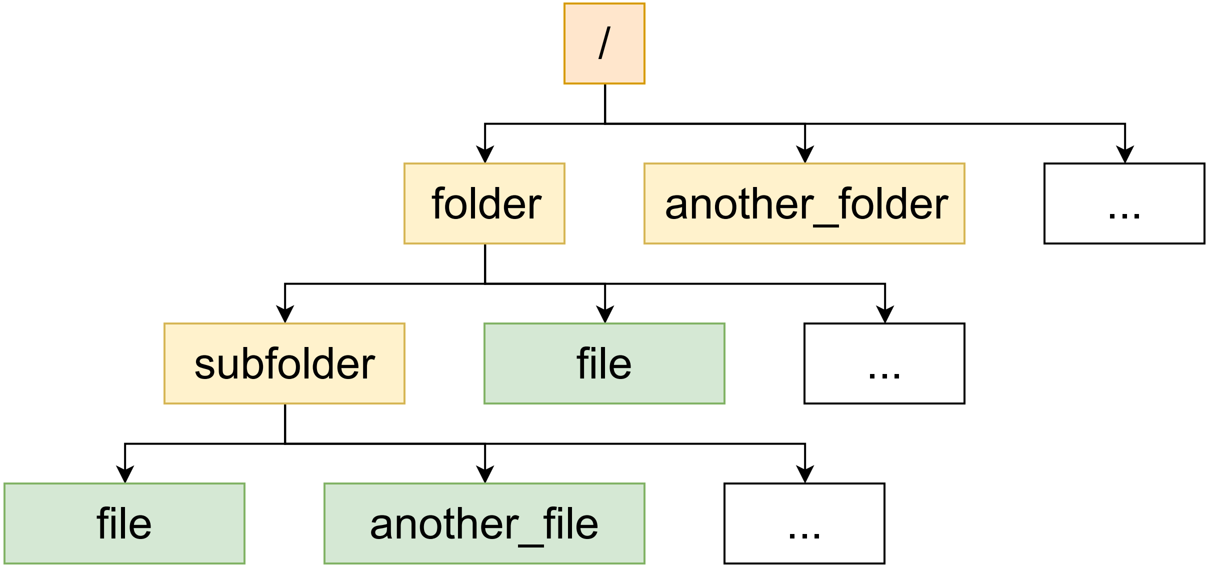
On Linux, paths looks similarly:

```
/home/user/folder/subfolder/file
```

UNIX directory structure

```
/folder/subfolder/file
```

- a file **`file`**
- inside a directory **`subfolder`**
- which is inside a directory **`folder`**
- which itself is inside the **root directory** /



The / as directory separator

Forward slashes (/) separate the folders in the path.
Using multiple is valid, so the following is the same file:

```
/home/user/folder/subfolder/file
///home/user///folder/subfolder//file
```

The / as directory separator

Forward slashes (/) separate the folders in the path.
Using multiple is valid, so the following is the same file:

```
/home/user/folder/subfolder/file
///home/user///folder/subfolder//file
```

A path to a regular file never ends in /, e.g. this is not valid:

```
/home/user/folder/subfolder/file/
```

The / as directory separator

Forward slashes (/) separate the folders in the path.
Using multiple is valid, so the following is the same file:

```
/home/user/folder/subfolder/file
///home/user///folder/subfolder//file
```

A path to a regular file never ends in /, e.g. this is not valid:

```
/home/user/folder/subfolder/file/
```

Directories can be referred to with or without the final /:

```
/home/user/folder/subfolder
/home/user/folder/subfolder/
```

The / as directory separator

Forward slashes (/) separate the folders in the path.
Using multiple is valid, so the following is the same file:

```
/home/user/folder/subfolder/file
///home/user///folder/subfolder//file
```

A path to a regular file never ends in `/`, e.g. this is not valid:

```
/home/user/folder/subfolder/file/
```

Directories can be referred to with or without the final `/`:

```
/home/user/folder/subfolder
/home/user/folder/subfolder/
```

Root directory is special: `/` is its only name.

Absolute and relative paths

If a path starts with `/`, it's an **absolute** path that starts at root:

```
/home/user/folder/subfolder/file
```

Absolute and relative paths

If a path starts with /, it's an **absolute** path that starts at root:

```
/home/user/folder/subfolder/file
```

If it does not, then it's a **relative** path that starts at the current working directory instead of /.

If the current working directory is

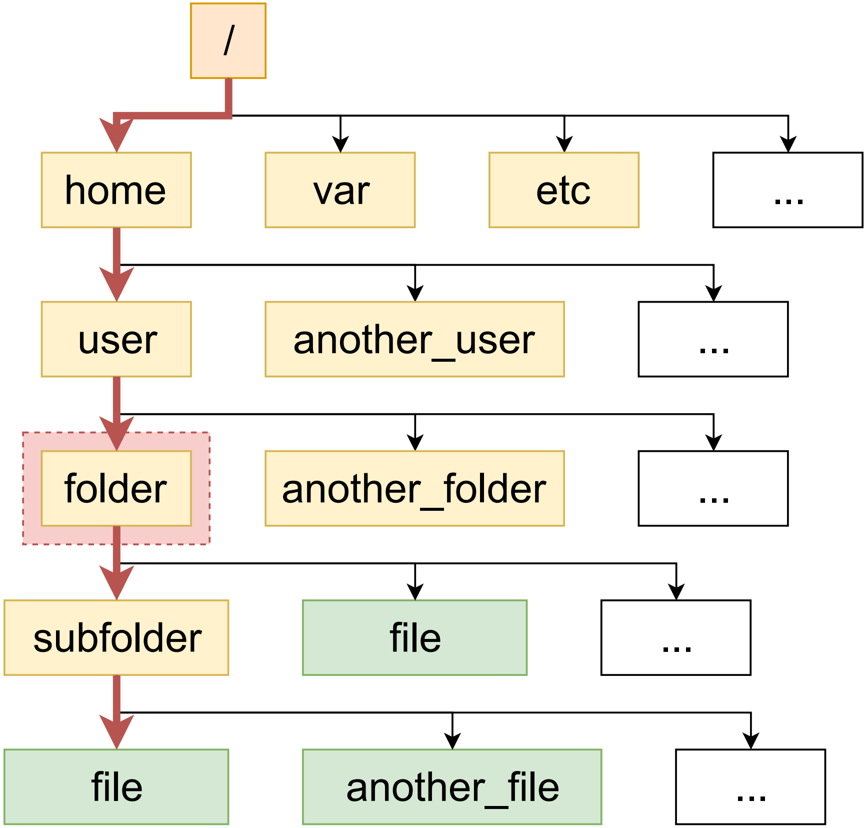
```
/home/user/folder
```

then the following paths point to the same file:

```
/home/user/folder/subfolder/file
subfolder/file
```

Absolute and relative paths

```
/home/user/folder/subfolder/file
subfolder/file
```



Special folders `.` and `..`

There are 2 special folders inside each folder: `.` and `..`

- `.` points to the folder itself.

```
/home/user/folder/subfolder/./file
```

Special folders `.` and `..`

There are 2 special folders inside each folder: `.` and `..`

- `.` points to the folder itself.

```
/home/user/folder/subfolder/./file
```

- `..` points to one folder "up" in the path. At root, it points to root itself.

```
/home/user/another_folder/../folder/file
/home/../../home/user/folder/file
```

Special folders . and ..

There are 2 special folders inside each folder: . and ..

- . points to the folder itself.

```
/home/user/folder/subfolder/./file
```

- .. points to one folder "up" in the path. At root, it points to root itself.

```
/home/user/another_folder/../folder/file
/home/../../home/user/folder/file
```

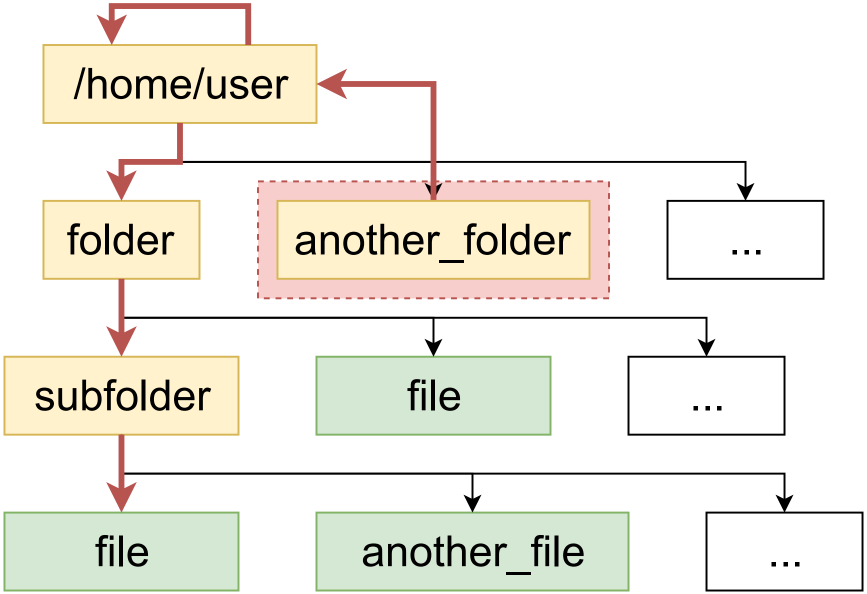
It's mostly important for relative paths:

```
# From /home/user/another_folder
../folder/file
```

Special folders . and . .

From `/home/user/another_folder`

```
/home/user/folder/subfolder/file
../folder/subfolder/file
```



Home directories

Each user has a **home directory** assigned.

It acts as your default working directory.

By convention, its path usually starts with `/home/` and ends with your username:

```
/home/<maybe something else>/username
```

It's frequently referred to as ~:

```
/home/username/folder/file
~/folder/file
```

You can even refer to others' home folder with `~username`:

```
/home/someone/file
~someone/file
```

Quiz time! [1/3]

Suppose the following:

```
Username:           userA
Home directory:     /home/userA

Working directory:  /scratch/folder/B
Target:             /scratch/folder/A/a
```

Which of those paths point to the target? (click to reveal)

- `~/../scratch/folder/A/a`
- `~userA/../../scratch/folder/A/a`
- `A/a`
- `../A/a`
- `/scratch/./folder/A/a`

Quiz time! [2/3]

Suppose the following:

```
Username:           userA
Home directory:     /home/userA

Working directory:  /home/userA/temp
Target:             ../../userB/folder/file
```

Which of those paths point to the target? (click to reveal)

`/home/userB/userB/folder/file`

`/home/userB/folder/file`

`~/folder/file`

`~/../userB/folder/file`

`~userB/folder/file`

Quiz time! [3/3]

Suppose the following:

```
Username:           userA
Home directory:     /home/userA

Working directory:  /home/userA/folder
Target:             /home/userA/folder/file
```

Which of those paths point to the target? (click to reveal)

`file`

`./file`

`~/file`

`~/folder/file`

`/home/userA/folder/subfolder/../file`

Preparing for training

Please execute the following command to add the exercises to your home folder:

```
$ curl https://scits.math.unibe.ch/script | bash
```

This should be the only time you don't understand what you're doing; by the end of Part II you should understand it.

Hint: On a Swiss German keyboard, | is `AltGr` + `7`

Moving around

Now that we know:

- Files and directories are organised in a tree.
- There's a "current"/working directory that we are in.

We need to learn to move around in that tree.

Moving around

Now that we know:

- Files and directories are organised in a tree.
- There's a "current"/working directory that we are in.

We need to learn to move around in that tree.

For that, we need the `cd` command:

```
user@remote:~ $ cd scits-training
user@remote:~/scits-training $ pwd
/home/username/scits-training
user@remote:~/scits-training $
```

Acronym:

`cd` stands for "**C**hange **D**irectory"

Moving around

The general format of the command is `cd DESTINATION`, where `DESTINATION` is a path (relative or absolute) to a directory.

```
user@remote:~ $ cd scits-training
user@remote:~/scits-training $ cd /usr/local/bin
user@remote:/usr/local/bin $
```

Moving around

The general format of the command is `cd DESTINATION`, where `DESTINATION` is a path (relative or absolute) to a directory.

```
user@remote:~ $ cd scits-training
user@remote:~/scits-training $ cd /usr/local/bin
user@remote:/usr/local/bin $
```

To go "back up", one uses the special `..` directory:

```
user@remote:/usr/local/bin $ cd ..
user@remote:/usr/local $ cd ../..
user@remote:/ $
```

Moving around

The general format of the command is `cd DESTINATION`, where `DESTINATION` is a path (relative or absolute) to a directory.

```
user@remote:~ $ cd scits-training
user@remote:~/scits-training $ cd /usr/local/bin
user@remote:/usr/local/bin $
```

To go "back up", one uses the special `..` directory:

```
user@remote:/usr/local/bin $ cd ..
user@remote:/usr/local $ cd ../..
user@remote:/ $
```

To go to your home directory, you can use `~`:

```
user@remote:/ $ cd ~
user@remote:~ $
```

cd shortcuts

There are two useful tricks when using `cd`:

"`cd -`" goes back to the previous directory you were in:

```
user@remote:~ $ cd -
user@remote:/ $
```

And "`cd`" without arguments goes to your home folder:

```
user@remote:/ $ cd
user@remote:~ $
```

Tab-completion

This is a good point to introduce a helpful CLI tool: **tab completion**

When entering a command, you can press the `[Tab]` key to suggest a command, or path, based on already entered input.

```
user@remote:~ $ cd scits-training/a
```

Pressing `[Tab]` now completes the name, since it's the only one that matches the beginning:

```
user@remote:~ $ cd scits-training/animals/
```

(continues on next slide)

Tab-completion

```
user@remote:~ $ cd scits-training/animals/
```

Pressing [Tab] once again won't change anything, since there are mutiple choices for completion; however, if it is pressed again, it shows possibilities:

```
user@remote:~ $ cd scits-training/animals/
Aardvark/ Badger/
user@remote:~ $ cd scits-training/animals/
```

Tab-completion

```
user@remote:~ $ cd scits-training/animals/
```

Pressing [Tab] once again won't change anything, since there are mutiple choices for completion; however, if it is pressed again, it shows possibilities:

```
user@remote:~ $ cd scits-training/animals/
Aardvark/ Badger/
user@remote:~ $ cd scits-training/animals/
```

The shell needs to know the next letter to proceed. So, we type only "A" and press Tab again:

```
user@remote:~ $ cd scits-training/animals/A
```

Tab-completion

```
user@remote:~ $ cd scits-training/animals/
```

Pressing [Tab] once again won't change anything, since there are mutiple choices for completion; however, if it is pressed again, it shows possibilities:

```
user@remote:~ $ cd scits-training/animals/
Aardvark/ Badger/
user@remote:~ $ cd scits-training/animals/
```

The shell needs to know the next letter to proceed. So, we type only "A" and press Tab again:

```
user@remote:~ $ cd scits-training/animals/Aardvark/
```

Tab-completion

```
user@remote:~ $ cd scits-training/animals/
```

Pressing `[Tab]` once again won't change anything, since there are mutiple choices for completion; however, if it is pressed again, it shows possibilities:

```
user@remote:~ $ cd scits-training/animals/
Aardvark/ Badger/
user@remote:~ $ cd scits-training/animals/
```

The shell needs to know the next letter to proceed. So, we type only "A" and press `Tab` again:

```
user@remote:~ $ cd scits-training/animals/Aardvark/
user@remote:~/scits-training/animals/Aardvark/ $
```

Looking around

To look around in a UNIX filesystem, you use the `ls` command:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls
big_file  description  empty_file  naming  subfolder
```

Mnemonic:

`ls` stands for **list**

This lists the names for contents of the working directory.

Looking around

To look around in a UNIX filesystem, you use the `ls` command:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls
big_file  description  empty_file  naming  subfolder
```

Mnemonic:

`ls` stands for **list**

This lists the names for contents of the working directory.

We can specify another folder to look at:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls ../Badger/
Arctonyx  Meles  Mellivora  Melogale  Mydaus
```

Looking around (in depth)

To show more information, we can use the `-l` (for **l**ong) flag:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls -l
total 25640
-rw-r--r-- 1 username groupname 26214400 Aug 28 18:20 big_file
-rw-r--r-- 1 username groupname      754 Aug 25 17:55 description
-rw-r--r-- 1 username groupname        0 Aug 28 16:51 empty_file
drwxr-xr-x 2 username groupname     4096 Aug 28 16:52 subfolder
```

Important information from this output:

- `-rw-r--r--` is called the **mode** (explained in Part II).
  - `d` denotes **directory** in this example.
  - `rw-r--r--` deals with permissions for the files.
- `username` and `groupname` are **owners** of the file.
- The number after `groupname` is the **size** (in bytes) of the file.
  - Important: for folders, it's not the size of all contents. You need `du` ("disk usage") to calculate that.
- The date/time after the size is the **modification date**.

Looking around (as puny humans)

One can use the flag `-h` (for **h**uman-readable) for more familiar size units:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls -l -h
total 26M
-rw-r--r-- 1 username groupname  25M Aug 28 18:20 big_file
-rw-r--r-- 1 username groupname  754 Aug 25 17:55 description
-rw-r--r-- 1 username groupname    0 Aug 28 16:51 empty_file
drwxr-xr-x 2 username groupname 4096 Aug 28 16:52 subfolder
```

Single-letter flags in commands can often be combined:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls -lh
total 26M
-rw-r--r-- 1 username groupname  25M Aug 28 18:20 big_file
-rw-r--r-- 1 username groupname  754 Aug 25 17:55 description
-rw-r--r-- 1 username groupname    0 Aug 28 16:51 empty_file
drwxr-xr-x 2 username groupname 4096 Aug 28 16:52 subfolder
```

Looking around (into hidden corners)

Another often-used flag is `-a` (for **a**ll): it lists contents with names that start with a dot `.` which are normally hidden in UNIX.

```
user@remote:~/scits-training/animals/Aardvark/ $ ls -a
.  ..  big_file  description  empty_file  .hidden  subfolder
```

As usual, it can be combined with others:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls -lah
total 26M
drwxr-xr-x 2 username groupname 4096 Aug 28 16:52 .
drwxr-xr-x 2 username groupname 4096 Aug 28 16:52 ..
-rw-r--r-- 1 username groupname  25M Aug 28 18:20 big_file
-rw-r--r-- 1 username groupname  754 Aug 25 17:55 description
-rw-r--r-- 1 username groupname    0 Aug 28 16:51 empty_file
drwxr-xr-x 2 username groupname 4096 Aug 28 16:52 subfolder
-rw-r--r-- 1 username groupname    0 Aug 28 16:51 .hidden
```

Looking around (in orderly fashion)

By default, files are ordered by name.

This behavior can be changed with flags; here are some examples:

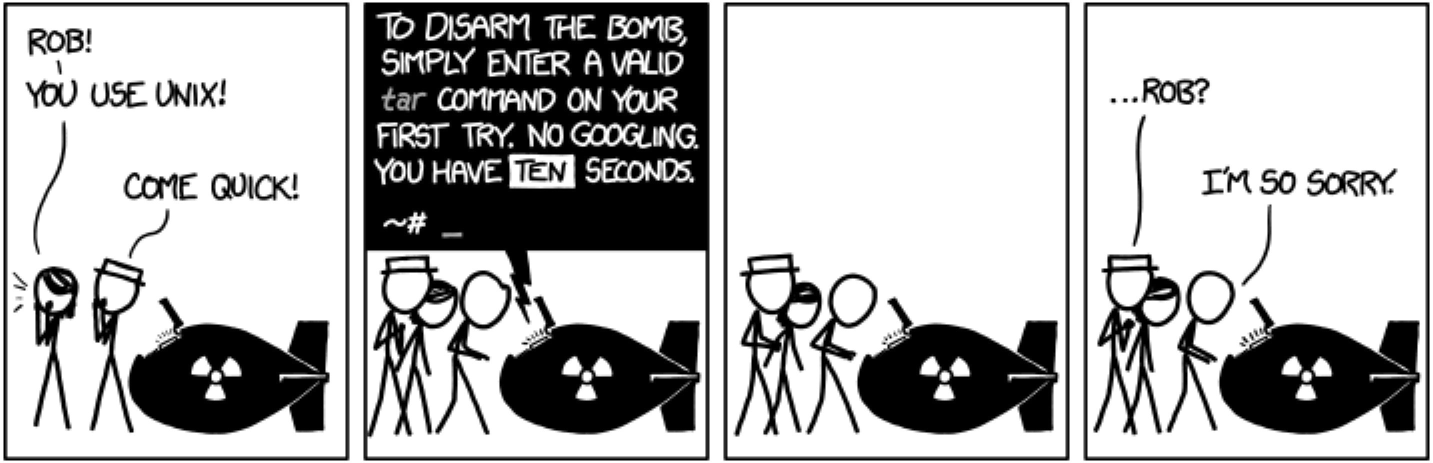
- `-r` **r**everses the sort order.
- `-S` sorts files by **s**ize.
- `-t` sorts files by modification **t**ime.
- `-X` sorts files by filename e**x**tension, e.g. `png` in `image.png`.

As usual, this can be combined with the previous ones.

Exercise:

List files in `Aardvark` by increasing size.

I'm never going to remember this!


ROB!
YOU USE UNIX!
COME QUICK!
TO DISARM THE BOMB, SIMPLY ENTER A VALID tar COMMAND ON YOUR FIRST TRY. NO GOOGLING. YOU HAVE TEN SECONDS.
~# _
...ROB?
I'M SO SORRY.

Good news: you don't have to.

As long as you remember the command's name, you can look up its correct usage from the terminal itself.

Image credit: https://xkcd.com/1168/

Getting help

Some common methods of getting help:

- Many programs support **`--help` flag** to print out their usage instructions:

```
user@remote:~/scits-training/animals/Aardvark/ $ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
[...]
```

Getting help

Some common methods of getting help:

- For most programs, you can look up their **manual file** with `man`:

```
user@remote:~/scits-training/animals/Aardvark/ $ man ls
```

Instead of just outputting the text and returning, you'll enter a mode for showing long files.

Look around using arrow keys and `PgDn`/`PgUp`.

Remember the hints on how to exit (here, it's `q`).

You can search a `man` page for "something" with `/something` and just / to go to the next find.

Getting help

Some common methods of getting help:

- Some commands are not separate programs, but are **built into the shell**, e.g. `cd`. For those, you can use `help`:

```
user@remote:~/scits-training/animals/Aardvark/ $ help cd
```

Getting help

Some common methods of getting help:

- Some commands are not separate programs, but are **built into the shell**, e.g. `cd`. For those, you can use `help`:

```
user@remote:~/scits-training/animals/Aardvark/ $ help cd
```

You can see what `help` can help with as well:

```
user@remote:~/scits-training/animals/Aardvark/ $ help
```

Try out `man`

Try opening the manual for `ls`:

```
user@remote:~/scits-training/animals/Aardvark/ $ man ls
```

Reminders:

- You can search a `man` page for "something" with `/something` and `n` to go to the next find.
- To exit, you can use `q`.

Exercise:

Try searching for the meaning of `-R` flag, and try to use it.

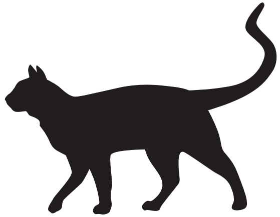
Reading files

We know how to look around the filesystem (with `ls`) and how to move around (with `cd`).

However, we still need to access the contents of files.

There are many ways to do that, I'll show a few more common ones.

Simple file reading



The simplest program to read the file is `cat`

```
user@remote:~/scits-training/animals/Aardvark/ $ ls
big_file  description  empty_file  naming  subfolder
user@remote:~/scits-training/animals/Aardvark/ $ cat description
The aardvark (ARD-vark; Orycteropus afer) is a medium-sized, burrowing,
[...]
```

Mnemonic:

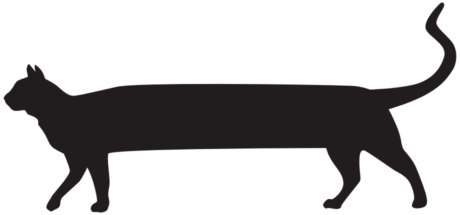
`cat` comes from the word "con**cat**enate",
which means joining things together in a series.

Exercise:

What happens if we call `cat` with two filenames?

```
cat description naming
```

File is too long!



Sometimes a file is too long to be comfortably read with `cat`

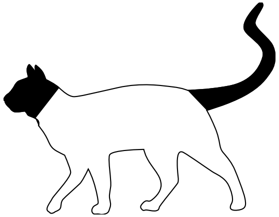
```
user@remote:~/scits-training/animals/Aardvark/ $ cd ../../numbers/
user@remote:~/scits-training/numbers/ $ cat hundred
1
2
[...]
99
100
```

A hundred lines is too much to fit into the terminal window.

While you can scroll to look through the output, sometimes files are much longer than that.

We can display only parts of the file, or use a program that allows to navigate a file.

Parts of a cat?



If a cat is too long, perhaps we only need to look at its beginning (`head`) or end (`tail`):

```
user@remote:~/scits-training/numbers/ $ head hundred
1
[...]
10
```

```
user@remote:~/scits-training/numbers/ $ tail hundred
91
[...]
100
```

Those commands display the first and last 10 lines of a file, respectively.

Mnemonic:

Remembering `cat` together with `head` and `tail` may help.

Self-help test

Of course, you can look up other options with the self-help methods like `man`.

Exercise:

Use one of the help methods (`man head` or `head --help`) to learn how to display 5 lines instead of 10 with `head`.

Hint: it will be a flag that should go before the filename.

Self-help test

Of course, you can look up other options with the self-help methods like `man`.

Exercise:

Use one of the help methods (`man head` or `head --help`) to learn how to display 5 lines instead of 10 with `head`.

Hint: it will be a flag that should go before the filename.

Answer: `-n 5`, `-n5` or `--lines=5`

```
user@remote:~/scits-training/numbers/ $ head -n 5 hundred
1
2
3
4
5
```

The file is too long, show less

One way to navigate a big file is `less`:

```
user@remote:~/scits-training/numbers/ $ less hundred
```

You will recognize this interface, since `man` also uses `less`.

Commands to try:

- **Arrow keys** to scroll line by line
- **`PgUp`** / **`PgDn`** to scroll screen by screen
- **`/something`** to search for "something"
- **`n`** to go to next found "something", **`N`** to go back
- **`>`** to go to the end of the file, **`<`** to go to the beginning
- **`h`** to show help
- **`q`** to quit

Modifying files

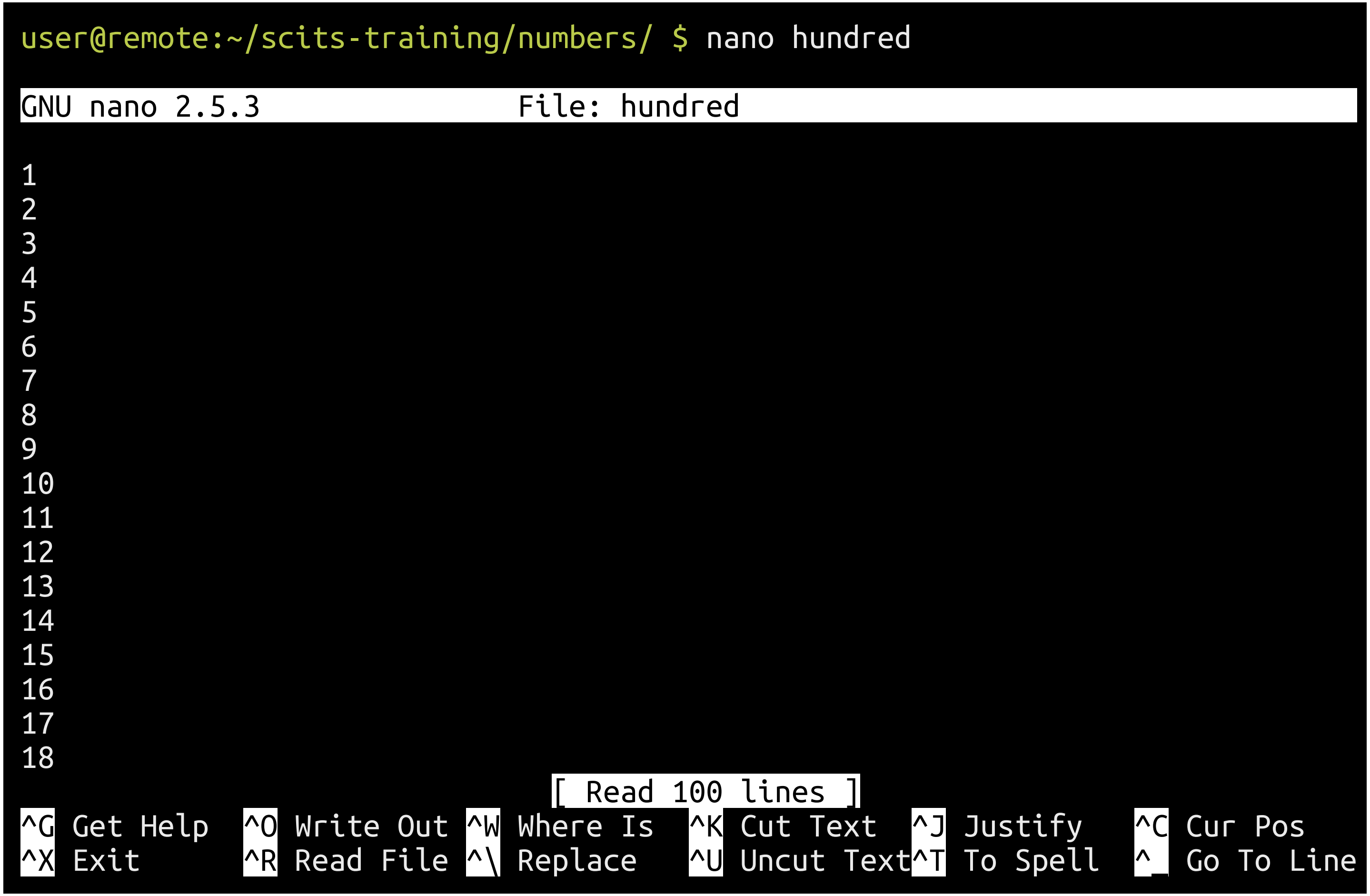
Besides reading, we need to be able to create and modify files.

There are many editors available, and which one is "best" can lead to hot debate.

We will mention and briefly explain two editors that are likely to be installed on any system you encounter nowadays.

- `nano`
- `vim`

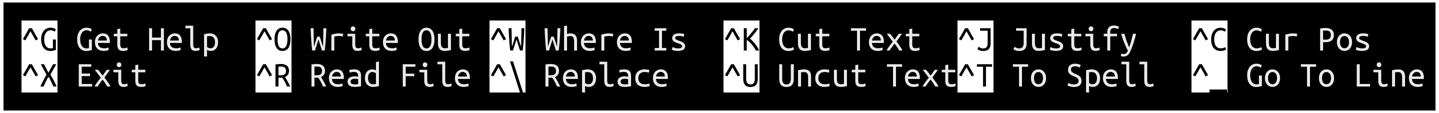
nano



```
user@remote:~/scits-training/numbers/ $ nano hundred

GNU nano 2.5.3                  File: hundred

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
                                  [ Read 100 lines ]
^G Get Help     ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
^X Exit         ^R Read File    ^\ Replace      ^U Uncut Text^T To Spell       ^_ Go To Line
```

nano

`nano` is a small and simple editor which helpfully shows its commands at the bottom (reminder, ^ means `Ctrl`):



```
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

You can use arrow keys to move around, input text as normal from where the cursor is.

Key commands:

- `Ctrl` + `W` "where is" for searching the file
- `Ctrl` + `O` "write out" to save changes
- `Ctrl` + `X` "exit" to get back to the shell

Try nano

Exercise:

1. Open a new file, `ten`, with `nano`:

   ```
   user@remote:~/scits-training/numbers/ $ nano ten
   ```

2. Add numbers from 1 to 10 to it, on separate lines
3. Save and exit `nano`
4. Verify what's in the file using `cat`

`vim`

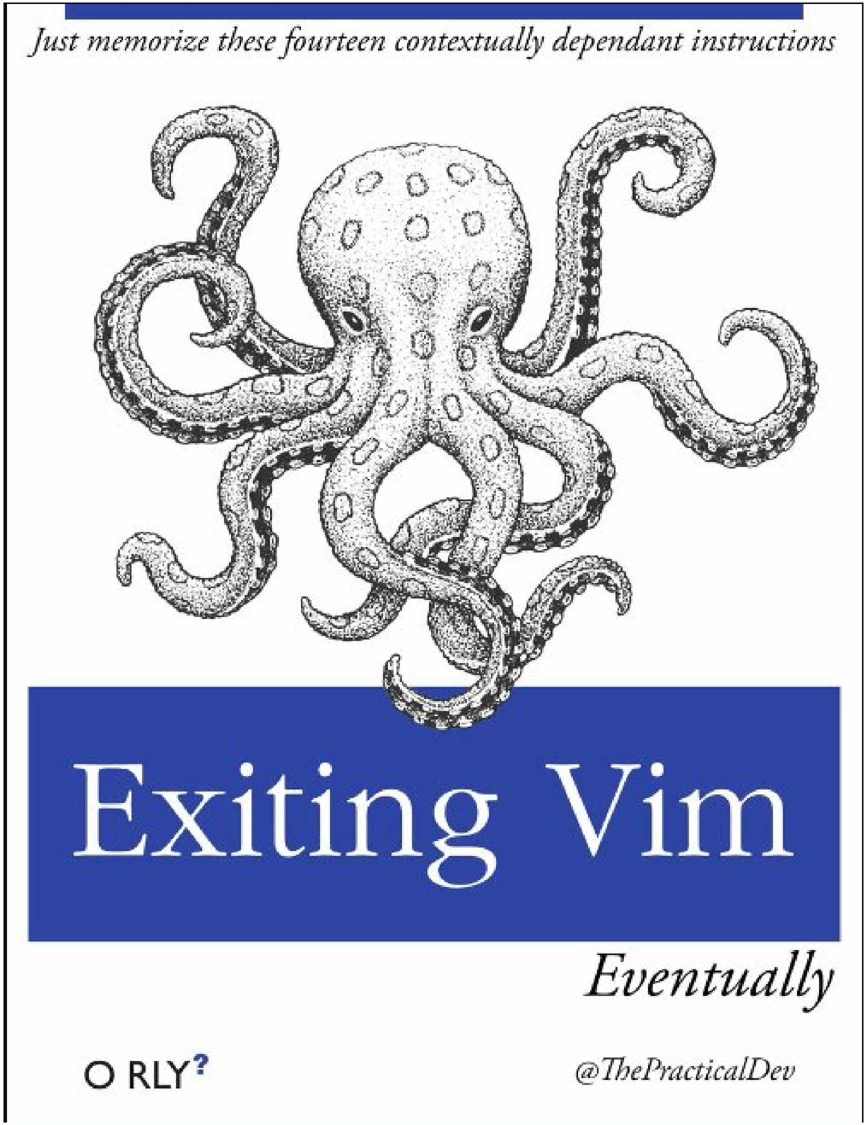
`vim` (or, technically, "Vi IMproved") is one of two "Swiss knife" editors that most Linux professionals prefer to use (the other one being `emacs`).

`vim` is available almost everywhere, and with proper configuration can do very sophisticated things.

With power comes complexity, but for basic editing one doesn't have to remember a lot.

If you wish to (later) explore `vim`, you can go through its built-in tutorial:

```
vimtutor
```



Just memorize these fourteen contextually dependant instructions

Exiting Vim

Eventually

O RLY?

@ThePracticalDev

Organizing files and folders

To recap, you should now be able to:

- Navigate the file tree (with `cd`)
- List folder contents (with `ls`)
- Read and write files (with `nano`)

Our goal now is:

- Make new folders
- To move and copy files and folders around
- Delete files and folders

Creating new folders

To create new folders, use the `mkdir` command:

```
user@remote:~/scits-training/numbers/ $ cd ..
user@remote:~/scits-training/ $ ls
animals  numbers
user@remote:~/scits-training/ $ mkdir new-folder
user@remote:~/scits-training/ $ ls
animals  new-folder  numbers
```

Mnemonic:

`mkdir` stands for **make dir**ectory

Exercise:

1. Create `new-folder` as shown above
2. Create directory `subfolder` inside it
3. Verify with `ls`

Creating new folders

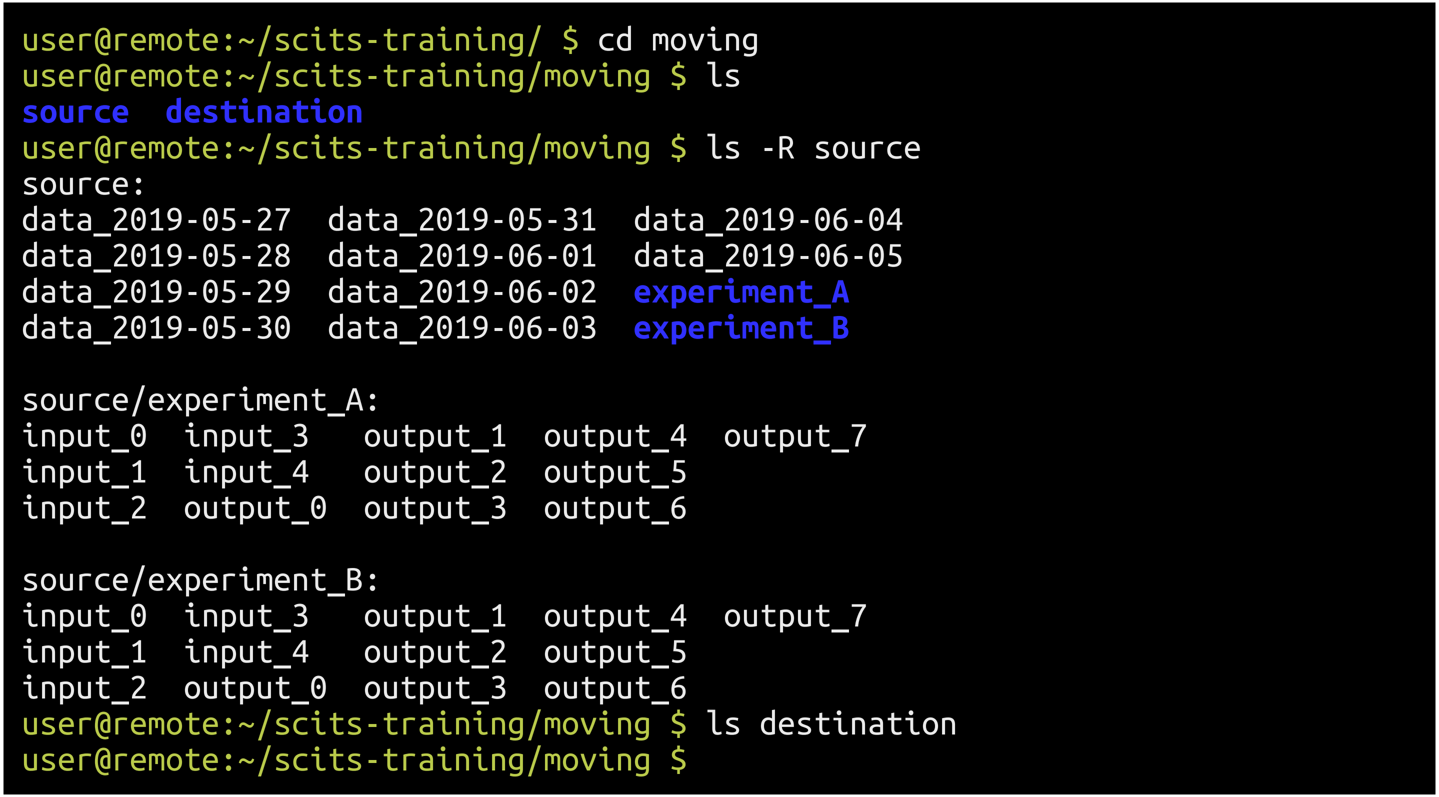
`mkdir` will fail if the folder already exists:

```
user@remote:~/scits-training/ $ mkdir new-folder
mkdir: cannot create directory ‘new-folder’: File exists
```

Using it with `-p` means "create if needed", and also works with chains of directories:

```
user@remote:~/scits-training/ $ mkdir -p new-folder/subfolder/subsubfolder
user@remote:~/scits-training/ $ ls -R new-folder
new-folder:
subfolder

new-folder/subfolder:
subsubfolder

new-folder/subfolder/subsubfolder:
user@remote:~/scits-training/ $
```

Preparing for moving exercises



```
user@remote:~/scits-training/ $ cd moving
user@remote:~/scits-training/moving $ ls
source  destination
user@remote:~/scits-training/moving $ ls -R source
source:
data_2019-05-27  data_2019-05-31  data_2019-06-04
data_2019-05-28  data_2019-06-01  data_2019-06-05
data_2019-05-29  data_2019-06-02  experiment_A
data_2019-05-30  data_2019-06-03  experiment_B

source/experiment_A:
input_0  input_3   output_1  output_4  output_7
input_1  input_4   output_2  output_5
input_2  output_0  output_3  output_6

source/experiment_B:
input_0  input_3   output_1  output_4  output_7
input_1  input_4   output_2  output_5
input_2  output_0  output_3  output_6
user@remote:~/scits-training/moving $ ls destination
user@remote:~/scits-training/moving $
```

Moving files

Move operations can be broken down into two cases:

1. Moving files and folders between folders:

   `folder1/something` → `folder2/something`

2. Renaming files and folders:

   `something` → `other`

   Technically, it's "moving" from old name to new.

Both cases are served with the `mv` command.

Mnemonic:

`mv` stands for **move**

Moving files

To move something to another folder: **`mv NAME DESTINATION`**, as long as the **`DESTINATION` is a directory that exists**.

```
user@remote:~/scits-training/moving $ mv source/data_2019-05-27 destination
```

You can specify multiple things to move at the same time, including folders:

```
$ mv source/data_2019-06-05 source/experiment_B destination
```

Moves both `source/data_2019-06-05` and `source/experiment_B` into `destination`.

Exercise:

Move `experiment_B` back into `source`

Renaming

To rename: `mv OLDNAME NEWNAME`, if `NEWNAME` is *not* an existing directory.

For example, let's rename `destination` to `dest`:

```
user@remote:~/scits-training/moving $ mv destination dest
```

If you're renaming something in another folder, you must specify the path twice:

```
$ mv dest/data_2019-06-05 dest/data_2019-07-05
```

Exercise:

Rename `dest` back into `destination`

Move + rename

Exercise:

Try the following from `~/scits-training/moving`:

```
$ mv source/data_2019-06-06 data_2019-07-06
```

Use `ls -R` to understand what happened

Move + rename

Exercise:

Try the following from `~/scits-training/moving`:

```
$ mv source/data_2019-06-06 data_2019-07-06
```

Use `ls -R` to understand what happened

Answer: Since there is no path for the second name, it moved into the current directory and got renamed:

`~/scits-training/moving/source/data_2019-06-02`

↓

`~/scits-training/moving/data_2019-07-02`

Copying

Copying is done with `cp`

Mnemonic:

`cp` stands for **copy**.

Syntax is the same:

- For copying to another directory, `cp NAME DESTINATION`
- For copying to another name, `cp OLDNAME NEWNAME`

Copying

Copying is done with `cp`

Mnemonic:

cp stands for **copy**.

Syntax is the same:

- For copying to another directory, `cp NAME DESTINATION`
- For copying to another name, `cp OLDNAME NEWNAME`

Exercise:

1. Copy `source/data_2019-06-03` and `source/data_2019-06-04` into `destination` (can you do it in one command?)
2. Copy and rename `source/data_2019-06-04` into `source/data_2019-07-04`

Copying folders

`cp`, unlike `mv`, will not copy directories by default:

```
$ cp source/experiment_B destination
cp: omitting directory 'source/experiment_B'
```

Copying folders

`cp`, unlike `mv`, will not copy directories by default:

```
$ cp source/experiment_B destination
cp: omitting directory 'source/experiment_B'
```

You need to use `-R` (or `-r`) to copy folders + content

```
$ cp -R source/experiment_B destination
```

Mnemonic:

`-R` stands for **r**ecursive

Deleting

To remove files or folders, use `rm`

Mnemonic:

`rm` stands for **rem**ove

- `rm NAME` to remove a file
- `rm -r FOLDER` to remove a folder

You can pass several names at once:

```
$ rm destination/experiment_B/input_0 destination/experiment_B/output_0
```

`rm` is unrecoverable!

When you delete files and folders with `rm`, you should be aware that there is no concept of "Trash".

Anything you delete (or overwrite) is lost with no easy way to recover.

You can use a flag `-i` to ask before any destructive operation.

```
user@remote:~/scits-training/moving $ cp -i -R source/experiment_B destination
cp: overwrite 'destination/experiment_B/input_1'?
```

On the other hand, sometimes you want to override those confirmations, especially for `rm` – you can do it with `-f`.

Mnemonic:

`-i` stands for **interactive**
`-f` stands for **force**

Wildcards

There are many similar-named files in `source`:

```
user@remote:~/scits-training/moving $ ls source
data_2019-05-27  data_2019-05-31  data_2019-06-04
data_2019-05-28  data_2019-06-01  data_2019-06-05
[...]
```

We may want to copy them all at once. We can use wildcards:

- **`*`** in a name means "any amount of any characters"
  - For example, **`A*`** can mean `A`, `A1` and `A10`
- **`?`** in a name means "any single character"
  - For example, **`A?`** can mean `A1`, `A6` but not `A10`

The wildcards will **not jump through directories**:

- **`*1`** can mean `A1`, `A11`, but not `subfolder/B1`
- **`*/*`** can match `subfolder/B1`

Wildcard quiz (1/3)

Which of the following names match the pattern `A*a*`

AAa
A/a
aaA
CBAcba
abcABC

Wildcard quiz (2/3)

Which of the following names match the pattern `A?a?`

AAa

AAaa

Aaaa

AAaaa

aAAaa

Wildcard quiz (3/3)

Which of the following patterns match the name `A110`

- `A*`
- `*`
- `*A*`
- `*A`
- `A???`

Using wildcards

Putting a name with a wildcard is equivalent to putting all names that match:

```
$ cp -r source/experiment_* destination
```

is equivalent to

```
$ cp -r source/experiment_A source/experiment_B destination
```

So, you can use wildcards in any command that expects multiple files.

Using wildcards

Putting a name with a wildcard is equivalent to putting all names that match:

```
$ cp -r source/experiment_* destination
```

is equivalent to

```
$ cp -r source/experiment_A source/experiment_B destination
```

So, you can use wildcards in any command that expects multiple files.

If no files match, the argument will be left as-is. Compare:

```
$ echo *
$ echo does_not_exist*
$ echo "*"
```

Try wildcards

From `~/scits-training/moving`, do the following:

Exercise:

1. List all data files from June inside `source`.
2. Copy all data files from July from `source` into `destination`.
3. Move all files starting with `input` from `source/experiment_A` into `destination`.
4. Delete all files with names ending with `1` from `destination`.

Use wildcards to do each point as one command.

Brace expansion

The bash shell (default on most systems) provides a useful mechanism called **braces**.

It allows to specify substitutions that get expanded to a list:

```
$ echo {1,2,3}
1 2 3
$ echo a{b,c,de,}f
abf acf adef af
```

Brace expansion

The bash shell (default on most systems) provides a useful mechanism called **braces**.

It allows to specify substitutions that get expanded to a list:

```
$ echo {1,2,3}
1 2 3
$ echo a{b,c,de,}f
abf acf adef af
```

Multiple braces can be combined:

```
$ echo {A,B,C}{1,2,3}
A1 A2 A3 B1 B2 B3 C1 C2 C3
```

You can also use ranges:

```
$ echo {1..12}
1 2 3 4 5 6 7 8 9 10 11 12
```

Brace expansion example

Braces are extremely useful for renaming to avoid repeating the path.

```
user@remote:~/scits-training/moving $ mv source/experiment_{B,C}
user@remote:~/scits-training/moving $ mv source/experiment_C/input_1{,.old}
```

Exercise:

Use braces to delete `input_2` and `output_2` inside `destination/experiment_B` in one command.

Introduction to Linux

Part II

`https://goo.gl/Vg3iXW#part2`

Agenda

1. Linux resources you can use
2. Moving data from/to a remote Linux system
3. Standard input/output and its redirection
4. Background processes
5. Durable sessions with `screen`
6. File ownership and permissions
7. Shell scripting basics
8. Environment and customization
9. (Extra credits)

What Linux resources can I use?

To do development and run light workloads:

- Your own computer may already run Linux.
- You can install Linux in a virtual machine.

  I recommend Virtualbox for personal use.
- If you're running Windows 10, you can install Windows Subsystem for Linux

What if it's not enough?

To create persistent services:

- Ask your group's sysadmin for servers/VM resources.
- UniBe Informatikdienste offers virtual machines.
- Cloud resources: SWITCHengines, other cloud services.

To run heavy calculations:

- UBELIX Linux cluster.
- Your group may have in-house infrastructure.
- Again, cloud services.

Moving data in and out

So far we have moved the data around on the system itself.

It doesn't help if you want to load external data or download the results of your programs.

Moving data in and out

So far we have moved the data around on the system itself.

It doesn't help if you want to load external data or download the results of your programs.

Perhaps, it's your own system and you have access to cloud storage or external storage devices.

Sometimes, you have a shared network folder between your computer and the target system.

But how to do it, if your only interface to the server is SSH?

Moving data in and out

So far we have moved the data around on the system itself.

It doesn't help if you want to load external data or download the results of your programs.

Perhaps, it's your own system and you have access to cloud storage or external storage devices.

Sometimes, you have a shared network folder between your computer and the target system.

But how to do it, if your only interface to the server is SSH?

We will cover two ways:

1. Downloading data from the Internet with `wget`

2. Copying data between computers with `scp`

Downloading from the shell

Sometimes, the data you need is a file on the Internet.

`wget` is the simplest-to-use tool for it:

```
user@remote:~/scits-training/moving $ cd ..
user@remote:~/scits-training/ $ wget https://example.com/
--2017-09-12 12:00:00--  https://example.com/
Resolving example.com (example.com)... 93.184.216.34
Connecting to example.com (example.com)|93.184.216.34|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1270 (1.2K) [text/html]
Saving to: ‘index.html’

100%[=======================================>] 1,270       --.-K/s   in 0s

2017-09-12 12:00:00 (36.2 MB/s) - ‘index.html’ saved [1270/1270]

user@remote:~/scits-training/ $ tail index.html
```

Mnemonic:

`wget` stands for **Web get**

Downloading from the shell

Notable option: renaming the file immediately.

- **-O** (for **o**utput) chooses a specific file to write to

```
$ wget https://tools.ietf.org/rfc/rfc1149.txt -O april.txt
[...]
$ less april.txt
```

As usual, use `man wget` to see more options.

It can work with HTTP/HTTPS/FTP-hosted files.

Downloading from the shell

Notable option: renaming the file immediately.

- **-O** (for **o**utput) chooses a specific file to write to

```
$ wget https://tools.ietf.org/rfc/rfc1149.txt -O april.txt
[...]
$ less april.txt
```

As usual, use `man wget` to see more options.

It can work with HTTP/HTTPS/FTP-hosted files.

`curl` is another option, which simply outputs the file.

```
$ curl https://scits.math.unibe.ch/script
```

We'll see how to actually use that in a little bit.

Transferring files between systems

To send files between two computers using SSH, the simplest command is `scp`.

Mnemonic:

`scp` stands for **secure copy**

`scp` behaves a lot like `cp`, but you can provide locations on other computers.

How to use `scp` on your own machine depends on the OS.

`scp` on Linux / MacOS (/ Win10)

From your **local** terminal, you can transfer a file from a remote system:

```
local.user@local:~ $ scp user@remote:~/scits-training/numbers/hundred .
[..some authentication..]
local.user@local:~ $ less hundred
```

`scp`'s parameters work similarly to `cp`, but you can refer to files on other systems by adding **`user@remote:`** to the path.

It works both ways, and can rename as well:

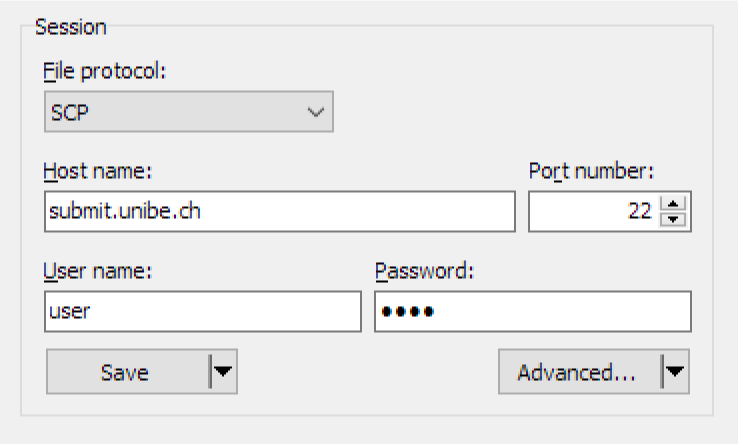
```
$ scp hundred user@submit.unibe.ch:~/scits-training/numbers/another_hundred
```

Graphical `scp` on Windows+PuTTY

While PuTTY includes a command-line client `pscp` with the same functions, it may be better to use a GUI client `WinSCP`.

It can be downloaded from https://winscp.net/

You can then connect using SCP (or SFTP) with your normal credentials and transfer files between your PC and the remote:



Session

File protocol: SCP

Host name: submit.unibe.ch

Port number: 22

User name: user

Password: ••••

Moving data in and out

Exercise:

1. Copy all `input`-files from `moving/source/experiment_A` to your computer with one command (use wildcards).
2. Copy some folder from your computer to the home folder of the remote system (hint: you'll use `-r`).

Hint: you'll need to run it in your local terminal!

Moving data in and out

Exercise:

1. Copy all `input`-files from `moving/source/experiment_A` to your computer with one command (use wildcards).
2. Copy some folder from your computer to the home folder of the remote system (hint: you'll use `-r`).

Hint: you'll need to run it in your local terminal!

In addition to `scp`, there's a command that works better for repeatedly copying large folders with small changes: **`rsync`**.

It will not be covered here, but look up information on it if it's your use case.

Processes and their input/output

When you're connected to a Linux system, what you normally see is the shell prompt, awaiting input:

```
user@host:~ $
```

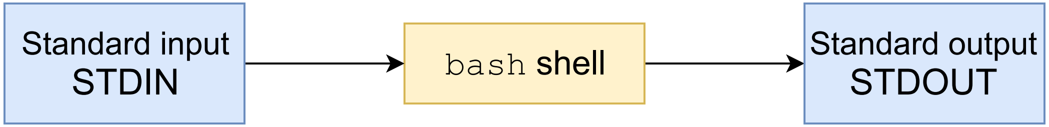
A **process** called shell is responsible for input/output at this moment.

By default, the input is either a keyboard connected to the system, or your keypresses being relayed over the network.

The output, by default, is the screen connected to the system, or text being relayed for display over the network.

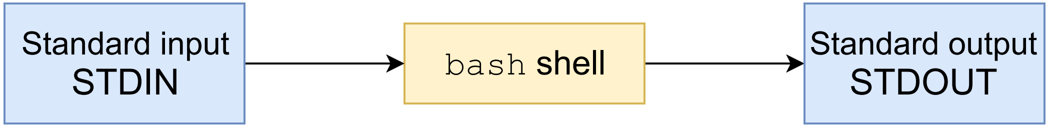
Processes and their input/output

Those are called **standard input** and **standard output**, or **STDIN/STDOUT**.

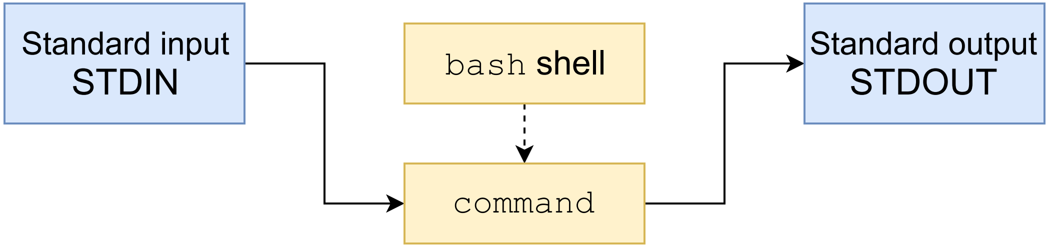


Processes and their input/output

Those are called **standard input** and **standard output**, or **STDIN/STDOUT**.



If the user issues a command that calls another program, the shell creates a **child process** and attaches the input/output to it.



The shell will wait until the program terminates, after which STDIN/STDOUT get reattached and a prompt is displayed.

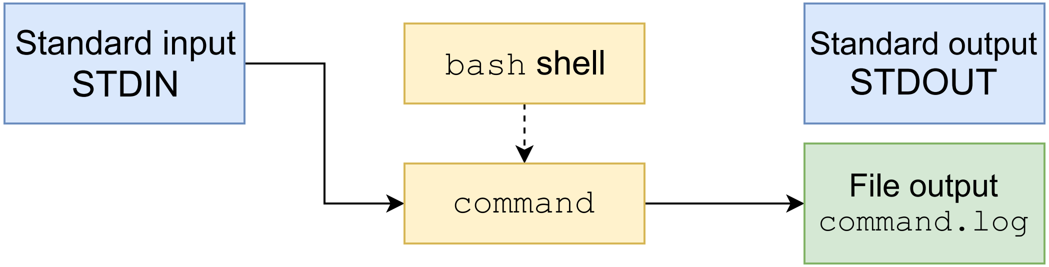
Redirecting output

Sometimes, we want to capture what a command is outputting to the screen.

For example, suppose that we want to save into a file the directory structure returned by `ls -R`:

```
user@host:~ $ ls -R ~/scits-training
scits-training:
animals  moving  numbers  io  scripts
[...]
user@host:~ $
```

We can instruct the shell to **redirect** the standard output:



Redirecting output

To redirect the output, we add **`> FILE`** to the command:

```
user@host:~ $ cd ~/scits-training/io
user@host:~/scits-training/io $ ls -R ~/scits-training > listing
user@host:~/scits-training/io $ cat listing
scits-training:
animals  moving  numbers  io  scripts
[...]
```

This will overwrite the contents of `FILE` (if any) with the output of the command.

The file will be created, if it does not exist yet.

Appending output

Sometimes we don't want to overwrite (sometimes called "clobber") the file with new contents and add them to the end instead.

To do that, use **`>> FILE`** instead of `> FILE`.

Exercise:

1. Try running the command `date` to see what it outputs.
2. Run `date` 3 times, appending the output to a file `date.log`.
3. Verify with `cat` that the file contains 3 records.

Error output

Try saving the output of a command with errors and you'll see that it still outputs to the screen:

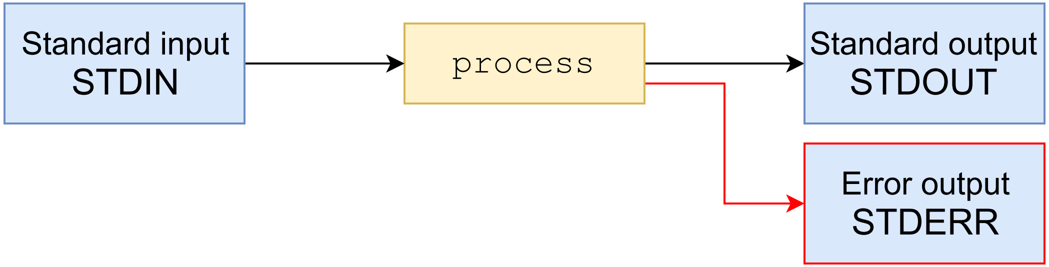
```
user@host:~/scits-training/io $ ls , > listing
ls: cannot access ,: No such file or directory
user@host:~/scits-training/io $
```

Error output

Try saving the output of a command with errors and you'll see that it still outputs to the screen:

```
user@host:~/scits-training/io $ ls , > listing
ls: cannot access ,: No such file or directory
user@host:~/scits-training/io $
```

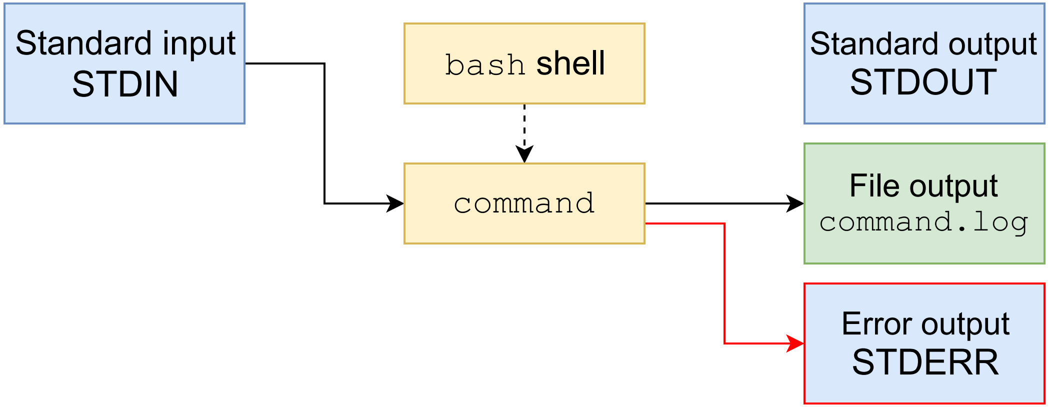
This is intentional: Linux actually has two output streams for its command line, STDOUT for normal data and **STDERR** for errors.



This simplifies debugging: errors are separate from data.

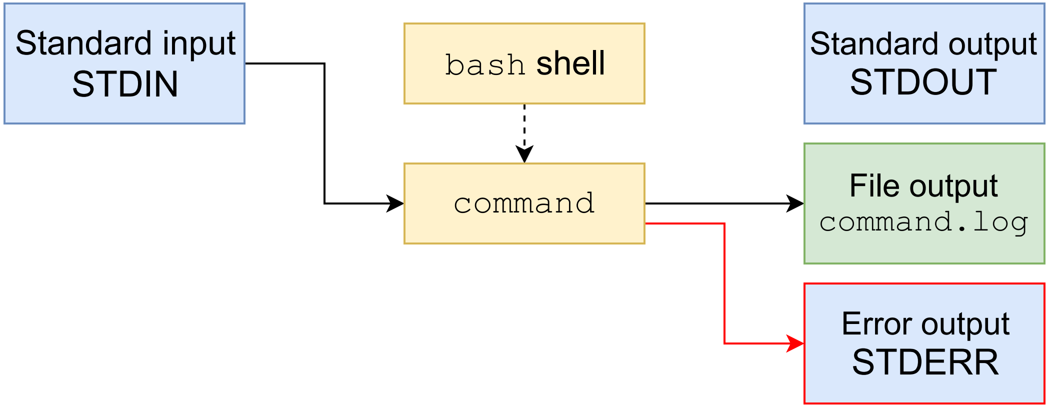
Error output

STDERR is not redirected when using > or >>:



Error output

STDERR is not redirected when using > or >>:



Standard input
STDIN

bash shell

command

Standard output
STDOUT

File output
command.log

Error output
STDERR

It's possible to redirect it as well with **2>** or **2>>**:

```
user@host:~/scits-training/io $ ls .. , > listing 2> errors
user@host:~/scits-training/io $ cat errors
ls: cannot access ,: No such file or directory
user@host:~/scits-training/io $ cat listing
..:
animals  io  moving  numbers  scripts
```

Discarding output

Sometimes we don't need output at all.

In this case, we can redirect it to a special file, **`/dev/null`**

It's a **device** that will accept any input, discarding it immediately.

Discarding output

Sometimes we don't need output at all.

In this case, we can redirect it to a special file, **`/dev/null`**

It's a **device** that will accept any input, discarding it immediately.

For example, one might want to silence errors:

```
user@host:~/scits-training/io $ ls .. , 2> /dev/null
..:
animals  io  moving  numbers  scripts
```

Interactive input

Most commands we've seen don't require any interactive input.

`tr` (for **translate**) is a command that transforms its input: it substitutes some characters with others.

For example, `tr 'a-z' 'A-Z'` would translate all lowercase letters into uppercase.

Interactive input

Most commands we've seen don't require any interactive input.

`tr` (for **translate**) is a command that transforms its input: it substitutes some characters with others.

For example, `tr 'a-z' 'A-Z'` would translate all lowercase letters into uppercase.

Let's try this:

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
```

Interactive input

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
But enter doesn't stop it!
BUT ENTER DOESN'T STOP IT!
```

A problem: text can contain many lines, and the program won't know when to stop.

Interactive input

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
But enter doesn't stop it!
BUT ENTER DOESN'T STOP IT!
```

A problem: text can contain many lines, and the program won't know when to stop.

We can terminate the program with Ctrl+C, but it actually expects an **end of input**.

We can signal end of input with Ctrl+D on an empty line (or pressing it twice).

Interactive input

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
But enter doesn't stop it!
BUT ENTER DOESN'T STOP IT!
```

A problem: text can contain many lines, and the program won't know when to stop.

We can terminate the program with Ctrl+C, but it actually expects an **end of input**.

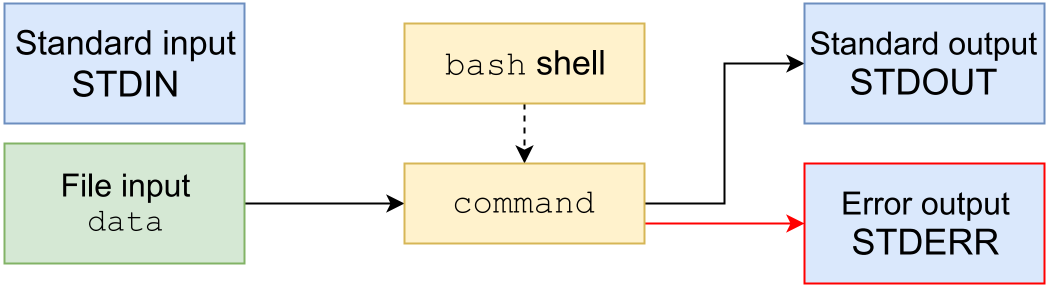
We can signal end of input with Ctrl+D on an empty line (or pressing it twice).

Exercise:

What happens if we press Ctrl+D while back at the shell prompt?

Redirecting input

What if we want to use a file as an input in a command that doesn't accept files as arguments, we need instruct the shell to use the file as the program's standard input:



This is done with adding `< FILE` to the command.

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z' < errors
LS: CANNOT ACCESS ,: NO SUCH FILE OR DIRECTORY
```

Redirects

Exercise:

1. Combine input and output redirection to save the output of last `tr` command into `errors.uppercase`
2. Use `cat` to verify the saved output.

Pipes

We have shown how to save outputs to a file, and further process files as inputs.

Sometimes, we don't need to save this intermediate representation. In that case, we can directly connect the output of one program to the input of another with **pipes**.

To do so, separate two commands with `|`:

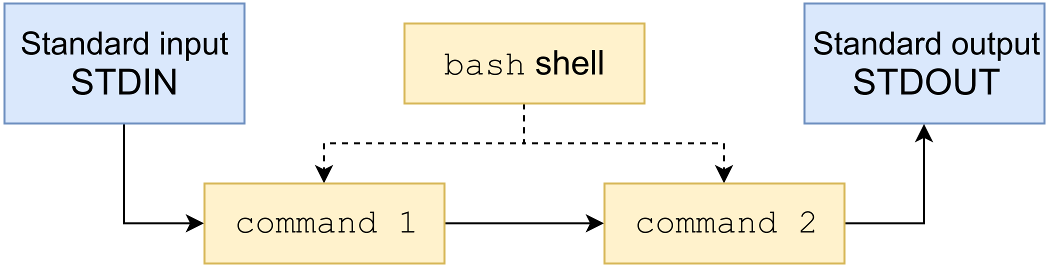
```
user@host:~/scits-training/io $ ls . | tr 'a-z' 'A-Z'
DATE.LOG
ERRORS
ERRORS.UPPERCASE
LISTING
```

Pipes

```
user@host:~/scits-training/io $ ls . | tr 'a-z' 'A-Z'
DATE.LOG
ERRORS
ERRORS.UPPERCASE
LISTING
```

Given this command, shell starts two processes in parallel and ties their respective output and input together.

Standard input/output is connected at the ends of the chain:

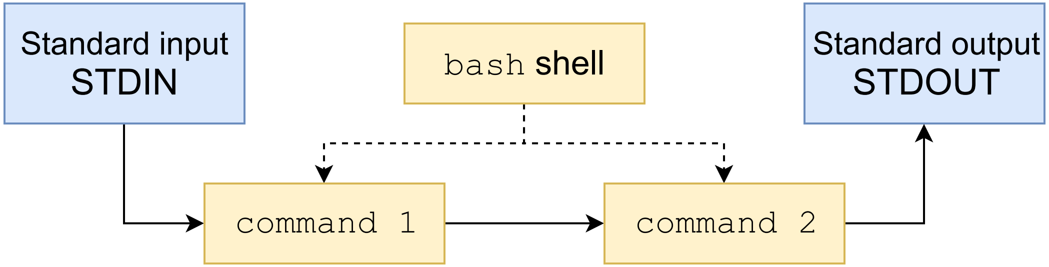


Pipes

```
user@host:~/scits-training/io $ ls . | tr 'a-z' 'A-Z'
DATE.LOG
ERRORS
ERRORS.UPPERCASE
LISTING
```

Given this command, shell starts two processes in parallel and ties their respective output and input together.

Standard input/output is connected at the ends of the chain:



Such **pipelines** can be longer than two commands, and can be combined with file redirects.

Example: Filtering output

One extremely useful command used in pipes is `grep`.

It allows to search for text patterns. Example:

```
user@host:~/scits-training/io $ ls .
date.log
errors
errors.uppercase
listing
user@host:~/scits-training/io $ ls . | grep log
date.log
```

`grep` is versatile:

- Can be used with **regular expression** patterns
- Can search for non-matching lines (with `-v`)
- Can search in files
- Can print where the match happened in a file

See `man grep` or Google for more examples.

Pipes and errors

You will notice that all errors are still output normally:

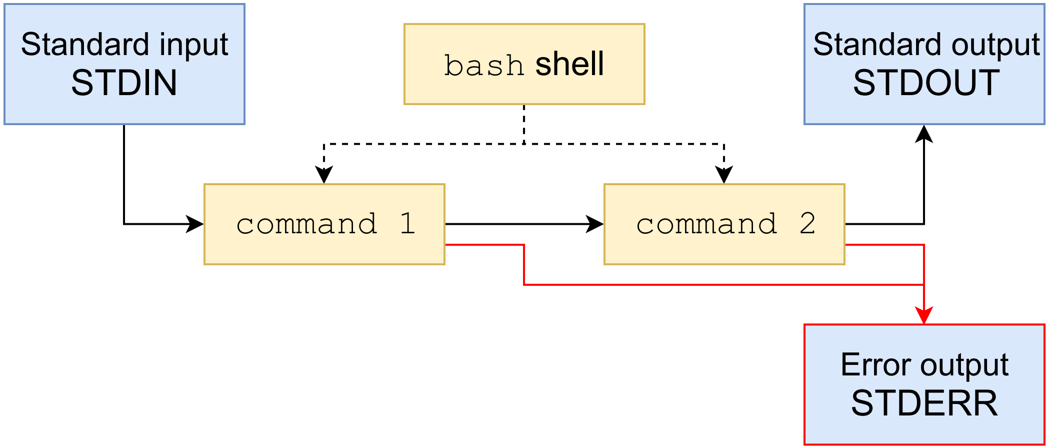
```
user@host:~/scits-training/io $ ls . , | tr 'a-z' 'A-Z'
ls: cannot access ,: No such file or directory
.:
DATE.LOG
ERRORS
ERRORS.UPPERCASE
LISTING
```

Pipes and errors

You will notice that all errors are still output normally:

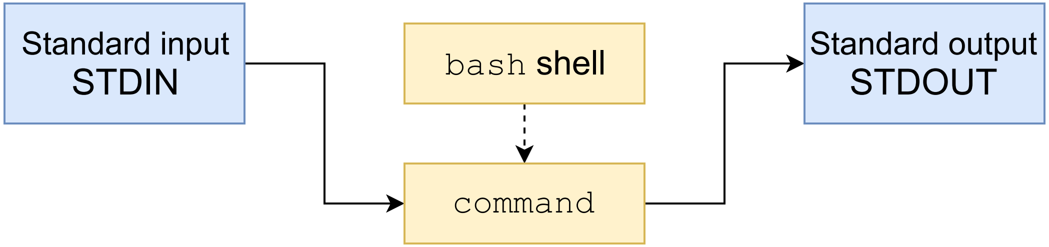
```
user@host:~/scits-training/io $ ls . , | tr 'a-z' 'A-Z'
ls: cannot access ,: No such file or directory
.:
DATE.LOG
ERRORS
ERRORS.UPPERCASE
LISTING
```

As before, errors are not normally redirected, and collected from all processes in the pipe:



Background jobs

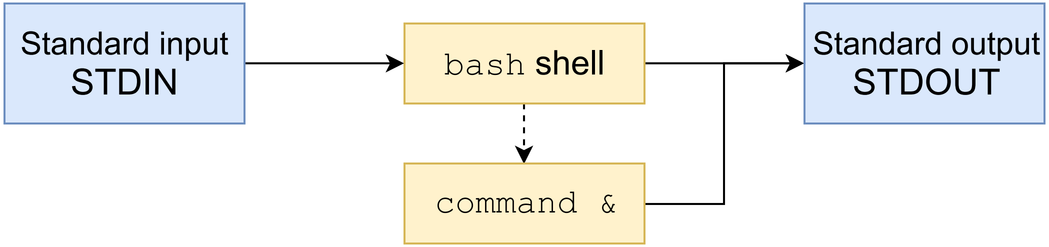
Recall that, when running a command, the shell waits until it is terminated: all input goes to the program (or nowhere).



Sometimes, we don't need to wait until the program terminates – we actually want it running in background.

Background jobs

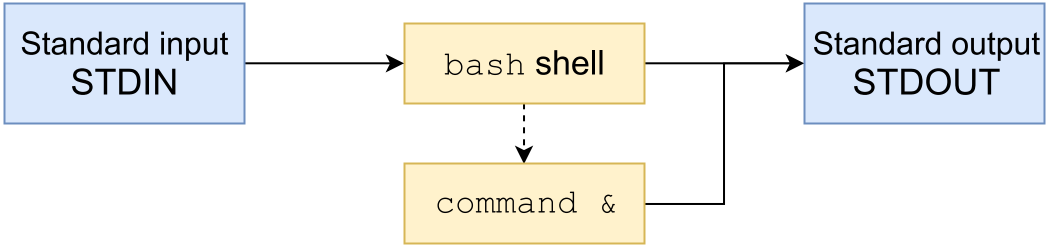
If you specify `&` at the end of the command, the shell will start it, but keep control of STDIN:



Instead of a **foreground** process, it becomes a **background** job.

Background jobs

If you specify `&` at the end of the command, the shell will start it, but keep control of STDIN:



Instead of a **foreground** process, it becomes a **background** job.

You are immediately returned to the shell and can run other commands while the job executes.

Note that both the shell and the background job are connected to STDOUT. Redirect output to prevent mix-ups.

Background jobs

Compare:

```
user@host:~/scits-training/io $ sleep 3
user@host:~/scits-training/io $ sleep 3 &
[1] 12231
user@host:~/scits-training/io $
```

Here, `[1]` is the **job number**, and `12231` is the **process ID**, or PID.

After 3 seconds and when another command finishes (you can just press Enter for an empty command), you'll be informed that the job terminated:

```
user@host:~/scits-training/io $
[1]+  Done                    sleep 3
user@host:~/scits-training/io $
```

Listing jobs

You can list running background jobs with **jobs**:

```
user@host:~/scits-training/io $ sleep 100 &
[1] 12232
user@host:~/scits-training/io $ sleep 0 &
[2] 12233
user@host:~/scits-training/io $ jobs
[1]-  Running                 sleep 100 &
[2]+  Done                    sleep 0
```

Terminating jobs

You can forcibly **terminate** a job with the `kill` command, which accepts either PID or job ID (with `%`):

```
user@host:~/scits-training/io $ sleep 100 &
[1] 12234
user@host:~/scits-training/io $ kill 12234
user@host:~/scits-training/io $ jobs
[1]+  Terminated              sleep 100

user@host:~/scits-training/io $ sleep 100 &
[1] 12235
user@host:~/scits-training/io $ kill %1
```

You can search for more process IDs to terminate with `ps ax`, in case something is misbehaving.

Stopped jobs

Background jobs have nothing connected to their standard input.

If a background job cannot continue without user input, it will **stop**, which the shell will signal to you:

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z' &
[1] 12236
user@host:~/scits-training/io $
[1]+  Stopped                 tr /a-z/ /A-Z/
user@host:~/scits-training/io $
```

Stopped jobs

Background jobs have nothing connected to their standard input.

If a background job cannot continue without user input, it will **stop**, which the shell will signal to you:

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z' &
[1] 12236
user@host:~/scits-training/io $
[1]+  Stopped                 tr /a-z/ /A-Z/
user@host:~/scits-training/io $
```

You can bring a job to **foreground** to pass STDIN from the shell to the running job with `fg` (or `fg %N` for a specific job number):

```
user@host:~/scits-training/io $ fg
tr /a-z/ /A-Z/
You are now talking to the job
YOU ARE NOW TALKING TO THE JOB
```

Stopping and resuming programs

You can stop most currently-running programs with `Ctrl+Z`:

```
user@host:~/scits-training/io $ sleep 100
^Z
[1]+  Stopped                 sleep 100
user@host:~/scits-training/io $
```

Stopping and resuming programs

You can stop most currently-running programs with `Ctrl+Z`:

```
user@host:~/scits-training/io $ sleep 100
^Z
[1]+  Stopped                 sleep 100
user@host:~/scits-training/io $
```

From there, you can use `fg` to resume normal execution of the program, or use **bg** to let it continue to run in the background.

```
user@host:~/scits-training/io $ bg
[1]+ sleep 100 &
user@host:~/scits-training/io $ jobs
[1]+  Running                 sleep 100 &
```

Background jobs are fragile

What will happen if you start a background job, and then close the terminal?

Background jobs are fragile

What will happen if you start a background job, and then close the terminal?

Closing the terminal (or disconnecting the SSH session) kills the shell you were talking to. Since the job was a **child process** of that shell, it will also be killed.

Background jobs are fragile

What will happen if you start a background job, and then close the terminal?

Closing the terminal (or disconnecting the SSH session) kills the shell you were talking to. Since the job was a **child process** of that shell, it will also be killed.

A minor inconvenience if you're working on your own machine (you can just leave the terminal open), but a much bigger problem with remote connections.

If the connection is broken, the shell is also terminated along with all processes launched from it.

How to protect against it?

`screen`

To protect your session, you can use **`screen`**.

`screen` starts a new shell that exists independently of your current one.

Even if the current shell dies (e.g. because you disconnected), the shell running in `screen` will continue together with all its child processes.

screen

To protect your session, you can use **screen**.

screen starts a new shell that exists independently of your current one.

Even if the current shell dies (e.g. because you disconnected), the shell running in screen will continue together with all its child processes.

Starting a new screen session is simple:

```
user@host:~/scits-training/io $ screen
[terminal screen is cleared]
user@host:~/scits-training/io $ echo "Hello, I'm in a screen"
Hello, I'm in a screen!
user@host:~/scits-training/io $
```

Reattaching to `screen`

Now suppose your connection was terminated.

Close the terminal where it is running to simulate that, then log in again.

Reattaching to `screen`

Now suppose your connection was terminated.

Close the terminal where it is running to simulate that, then log in again.

You can use **`screen -ls`** to list active sessions:

```
user@host:~ $ screen -ls
There is a screen on:
        13383.pts-2.host        (11/09/17 03:02:23)     (Detached)
1 Socket in /var/run/screen/S-user.
```

Reattaching to `screen`

Now suppose your connection was terminated.

Close the terminal where it is running to simulate that, then log in again.

You can use **`screen -ls`** to list active sessions:

```
user@host:~ $ screen -ls
There is a screen on:
        13383.pts-2.host        (11/09/17 03:02:23)     (Detached)
1 Socket in /var/run/screen/S-user.
```

You can attach to a screen session (possibly detaching it first, if it's being used somewhere) with **`-dR`** (for **detach, reattach**)

```
user@host:~ $ screen -dR
[terminal screen is cleared]
user@host:~/scits-training/io $ echo "Hello, I'm in a screen"
Hello, I'm in a screen!
user@host:~/scits-training/io $
```

Controlling `screen`

`screen` can be used for other things, such as having multiple parallel shell sessions open.

Controlling screen consists of pressing **`Ctrl+A`**, then a screen-specific command.

For example,

- **`c`** will **create** a new shell within screen
- **`n`** will switch to the **next** shell
- **`d`** will **detach** from `screen`, returning you to the original shell

Finally, you can use **`?`** to access built-in help, or use `man screen` for a more detailed manual.

Controlling `screen`

`screen` can be used for other things, such as having multiple parallel shell sessions open.

Controlling screen consists of pressing **`Ctrl+A`**, then a screen-specific command.

For example,

- **`c`** will **create** a new shell within screen
- **`n`** will switch to the **next** shell
- **`d`** will **detach** from `screen`, returning you to the original shell

Finally, you can use **`?`** to access built-in help, or use `man screen` for a more detailed manual.

Another popular alternative to `screen` is `tmux`. It will not be covered by this tutorial, but is worth looking into.

Users and groups

Before we discuss permissions, we need to understand users and groups in Linux.

A **user** is a unit of access control; it has a set of credentials to access the system and **owns** some files on it.

A **group** is a collection of users to facilitate shared access to resources. A user can belong to many groups but one group is considered primary.

You can use `id` to check your user and groups:

```
akashev@submit01:~ $ id
uid=7265(akashev) gid=1109(math) groups=1109(math),902(l_gaussian)
```

Here, `akashev` is my user, `math` is my primary group and `l_gaussian` is another group I belong to.

Permissions: `rwx`

Each file and directory in UNIX filesystems has 3 permissions (for a particular user).

Regular files:

- `r`, or **Read**, means that you can read the contents of a file.
- `w`, or **Write**, means that you can modify the file.
- `x`, or **eXecute**, means that the file may be launched as a program.

Directories:

- `r` means that you can read the list of files within the directory.
- `w` means that you can add or delete files from the directory.
- `x` means you can **traverse** the folder: enter it with `cd` and read the contents of its files.

Inspecting permissions

Try running `ls -la` to see permissions on files and folders:

```
$ ls -la
total 20
drwxrwxr-x 2 user group 4096 Sep 11 01:26 .
drwxrwxr-x 6 user group 4096 Sep 10 23:06 ..
-rw-rw-r-- 1 user group   90 Sep 10 23:08 date.log
-rw-rw-r-- 1 user group   47 Sep 11 00:50 errors
-rw-rw-r-- 1 user group   30 Sep 11 01:09 listing
```

We're interested in the first column: the cryptic `drwxrwxr-x` and `-rw-rwr--`, which are called **mode**.

Inspecting permissions

Try running `ls -la` to see permissions on files and folders:

```
$ ls -la
total 20
drwxrwxr-x 2 user group 4096 Sep 11 01:26 .
drwxrwxr-x 6 user group 4096 Sep 10 23:06 ..
-rw-rw-r-- 1 user group   90 Sep 10 23:08 date.log
-rw-rw-r-- 1 user group   47 Sep 11 00:50 errors
-rw-rw-r-- 1 user group   30 Sep 11 01:09 listing
```

We're interested in the first column: the cryptic `drwxrwxr-x` and `-rw-rwr--`, which are called **mode**.

- The first character denotes the **file type**.
  - `-` means "regular file".
  - `d` means "directory".
- The rest is divided in groups of three:
  - Access for the owner
  - Access for the group
  - Access for everyone else

File ownership

```
drwxrwxr-x 2 user group 4096 Sep 11 01:26 .
-rw-rw-r-- 1 user group   90 Sep 10 23:08 date.log
```

Each file in a UNIX filesystem has an **owner** and a **group** attached.

In the example above, `user` is the owner and `group` is the designated group.

Note that the user **doesn't have to be in the assigned group**.

Effective permissions

```
-rwxr-x--- 1 user group   90 Sep 10 23:08 script
```

To determine which permissions apply, the following is checked:

- If the user is the owner, the first set applies (`rwx`, full permissions)
- If the user is in the designated group, the second set applies (`r-x`, so cannot write)
- For all other users, the third set applies (`---`, so cannot do anything)

Effective permissions

```
-rwxr-x--- 1 user group   90 Sep 10 23:08 script
```

To determine which permissions apply, the following is checked:

- If the user is the owner, the first set applies (`rwx`, full permissions)
- If the user is in the designated group, the second set applies (`r-x`, so cannot write)
- For all other users, the third set applies (`---`, so cannot do anything)

A special user, **superuser** (normally called **root**), can completely disregard permissions and do anything to any file on the system.

Permissions: first match applies

Note that the system does not apply "best" permissions – only the first set that matches.

Let's reverse the situation:

```
----r-xrwx 1 user group   90 Sep 10 23:08 script
```

For this file, the owner cannot do anything to the file, anyone in `group` cannot modify it, but everyone else has full permissions.

Note: the owner can always change a file's permissions.

Modifying permissions

To modify a file's permissions, use `chmod CHANGES FILE`

Mnemonic:

`chmod` stands for **change mode**.

Possible changes:

- `+r`, `+w`, `+x` **add** permissions. Can combine: `+rw`
- `-r` **removes** permissions.
- `=r` sets pemissions to **exactly** `r--`.
- Prefix `u` changes permissions for the **user**, e.g. `u+r`.
- Prefix `g` changes permissions for the **group**, e.g. `g+rw`.
- Prefix `o` changes permissions for **others**, e.g. `o-w`.
- Prefix `a` or no prefix changes permissions for **all** three sets.
- An **octal number** (e.g. 750) sets permissions to a specific configuration (in this case, `rwxr-x---`).

Modifying permissions

Several changes can be applied at once, separated by commas:

```
user@host:~/scits-training/io $ ls -la date.log
-rwxrw-r-- 1 user group   90 Sep 10 23:08 date.log

user@host:~/scits-training/io $ chmod u+x,g=rx,o-r date.log

user@host:~/scits-training/io $ ls -la date.log
-rwxr-x--- 1 user group   90 Sep 10 23:08 date.log
```

Exercise:

Modify permissions on the file to be `r-xr--rw-`

Changing ownership

Similarly to `chmod`, the **`chown`** command allows changing a file's owner and group.

- `chown USER FILE` changes the owner
- `chown :GROUP FILE` changes the group
- `chown USER:GROUP FILE` changes both

Note: once the owner is changed, the old owner no longer can modify access to the file.

For this reason, only administrators can change the file owner, or assign a group the owner is not part of.

Exercise:

Use `groups` to list groups you belong to. Change any file's group to one of them.

Shell scripting

Shell is not just an interface to launch other programs; it comes with its own scripting language to automate complex tasks.

You can have variables, loops, conditionals – a full-featured programming language.

We will only show the very basics.

Exercise:

Navigate to `~/scits-training/scripts` and open `boom.sh` in your favourite editor (nano, vim)

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

The first line of the script is special:

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

It's called a **"shebang"** (for **shell** and "!" **bang**).

It tells the shell what to execute the rest of the script with. Since we're writing a `bash` shell script, we put there the path to `/bin/bash` itself.

Shell scripting

```bash
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

Other lines starting with **#** are **comments**

```bash
# I hope you get the reference
```

They are ignored by `bash` and are used to leave notes to yourself or others.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

echo command outputs its arguments to STDIN.

```
echo "Someone set up us the bomb."
```

Quotes are used to make text with spaces in it a single argument; here, they are optional.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

`for` designates a **loop**: a **variable `i`** will change from 5 to 1.

```
for i in {5..1}
do
  # something
done
```

The code in `# something` will repeat with `i` as 5, 4, 3, 2 and 1.

`do` and `done` delimit the bounds of the loop.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

One can use the variable in expressions **prefixed by `$`**, i.e. **`$i`**:

```
echo "$i.."
```

If there is ambiguity as to where a variable name ends, use braces: **`${i}`**, e.g. `"Sample ${i}A"` for "Sample 1A", etc.

Shell scripting

```bash
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

Variables can also be simply **assigned to**:

```bash
explosion="Boom!"
echo $explosion
```

The lack of spaces around = is **significant**.
Otherwise Bash will try to execute `explosion` as a command.

Running a script

OK, suppose we wrote the above script. How to execute it?

1. We need to make sure that it's allowed to execute:

   ```
   user@host:~/scits-training/scripts $ chmod +x boom.sh
   ```

2. For security reasons, the current directory is not automatically considered when starting other programs. We need to explicitly refer to it:

   ```
   user@host:~/scits-training/scripts $ ./boom.sh
   ```

Exercise:

1. Execute the script, saving its output to a file.
2. Modify the script to count down from 10.

Scripting, take two

The next script you will type out yourselves.

Open a new file `beer.sh` in your favourite editor.

We'll write a simple script to determine if a user is old enough to drink beer.

Scripting, take two

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

Any bash script should start with an appropriate shebang.

We want to ask the user for his/her age; we can use the `read` command.

```bash
# -n prevents a line break, and note the extra space
echo -n "What's your age? "
read age
```

This will display a promt for the user and wait for input. The result is then stored in the variable `$age`.

For simplicity, we will not check that the input is indeed a valid number.

Scripting, take two

```bash
#!/bin/bash
echo -n "What's your age? "
read age
```

We need to make a decision based on age; we need an if-then-else construct.

```bash
if [ $age -lt 16 ]
then
  echo "You're too young to drink!"
else
  echo "You're old enough, have a beer!"
fi
```

`fi` here is `if` reversed, to close the **`if`** statement.

Conditionals in bash are a bit clunky, but **`-lt`** here stands for **less than**. Again, the whitespace here is **significant**.

Scripting, take two

```bash
#!/bin/bash
echo -n "What's your age? "
read age
if [ $age -lt 16 ]
then
  echo "You're too young to drink!"
else
  echo "You're old enough, have a beer!"
fi
```

Exercise:

1. Save this script to `beer.sh`.
2. Change the file's mode to allow execution.
3. Test the script with different values.

Scripting improvements

Let's add a little personal touch.

`whoami` is a command that returns the username. Let's edit `beer.sh` to use it:

```
then
  echo "$(whoami), you're too young to drink!"
else
  echo "$(whoami), you're old enough, have a beer!"
fi
```

`$(something)` allows you to execute a command and substitute the result within another command.

Exercise:

Test the new additions.

Scripting improvements

Let's read the age from the command line arguments.

`bash` automatically populates **`$0`** with the **name of the executable**, and **`$1`**, **`$2`** and so on with **arguments**.

Let's use `$1` as age if it's defined:

```
if [ $1 ]
then
  age=$1
else
  echo -n "What's your age? "
  read age
fi
```

Exercise:

Test that `./beer.sh` now automatically gets the age from its first argument, and still asks if no argument is provided.

Return values

Whenever a program terminates, it returns a single integer to the shell that called it; it's called the **return value**.

By convention:

- **0** means "no error".
- any **non-zero value** means "some kind of error".

Let's return appropriate values:

```
then
  echo "$(whoami), you're too young to drink!"
  exit 1
else
  echo "$(whoami), you're old enough, have a beer!"
  exit 0
fi
```

Chaining commands

You can chain commands in shell with **;** or **&&**.

; will execute commands one by one, regardless of errors.

```
$ command1; command2
```

&& will only execute the next command only if the previous one returned 0, i.e. finished without errors.

```
$ command1 && command2
```

Exercise:

1. Apply the return value changes to `beer.sh`
2. Test it with `./beer.sh && echo 'Cheers!'`

Environment variables

Your profile files can set various **environment variables**: snippets of data inherited by programs running from shell.

You can see your current environment variables with:

```
$ env | less
```

Environment variables

Your profile files can set various **environment variables**: snippets of data inherited by programs running from shell.

You can see your current environment variables with:

```
$ env | less
```

Some programs rely on environmental variables to change their behavior. Example:

```
# Will replace the default editor with vim in some commands
export EDITOR=vim
```

Environment variables

Environment variables work similarly to variables in a script, except for the extra command **export**, which propagates this variable to child processes.

By convention, environment variables are UPPERCASE.

Environment variables

Environment variables work similarly to variables in a script, except for the extra command **export**, which propagates this variable to child processes.

By convention, environment variables are UPPERCASE.

Exercise:

1. Set an ordinary varable A (`A='Hello'`) and environment variable B (`export B='World'`).
2. Use `echo $A` and `echo $B` to display them.
3. Add `echo $A` and `echo $B` to a script and execute it. What changes?

$PATH variable

An important variable is `$PATH`.

It's a colon-separated list of directories which are searched when you try to run a program by name.

Notably, the current directory is not in `$PATH`.

If you have created some own scripts/programs and want them to be available by name from anywhere, you can put them in a folder (e.g. `~/bin`) and add it to `$PATH`:

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

$PATH variable, example

Exercise:

1. Try running `beer.sh` directly by name. It fails.
2. Add the folder that holds it to `$PATH`:

```
export PATH="$PATH:~/scits-training/scripts"
```

1. Try running `beer.sh` now.
2. Try going somewhere else (`cd ~`) and run it.

Aliases

If you use a certain command often, you can define a short name for it.

For example, if you want a shorter name for `ls -lh` because you always want to see human-readable sizes, you can make an alias:

```
$ alias lh="ls -lh"
$ lh
total 26M
-rw-r--r-- 1 user group 25M Sep 11 07:22 big_file
-rw-r--r-- 1 user group 735 Sep 11 07:22 description
-rw-r--r-- 1 user group   0 Sep 11 07:22 empty_file
-rw-r--r-- 1 user group 551 Sep 11 07:22 naming
drwxr-xr-x 0 user group 512 Sep 11 07:22 subfolder
```

Making customizations permanent

To make exports and aliases permanent, they need to be added either to `.bash_profile` or `.bashrc`.

Then they will apply on each opened shell.

- `.bash_profile` is **sourced** at most once. Put things there that shouldn't be called multiple times.
- `.bashrc` is sourced almost every time `bash` is called, except for initial SSH shell. To be safe, you can "include" `.bashrc` into `.bash_profile` like this:

```
# In .bash_profile
# -f tests that file exists
# source executes commands in the current shell
if [ -f ~/.bashrc ]; then
  source ~/.bashrc
fi
```

Extra credits

Let's look at our preparation script

Now you have enough knowledge to understand how we obtained `scits-training`:

```
$ curl https://scits.math.unibe.ch/script | bash
```

This downloads a file, and feeds it as input to the `bash` shell. What's in that file?

```
#!/bin/bash
echo "*** Downloading training archive.."
wget https://scits.math.unibe.ch/archive.tar.gz -O scits-training.tar.gz
echo "*** Deleting previous training folder, if any.."
rm -rf ~/scits-training
echo "*** Unpacking training folder.."
tar xzf scits-training.tar.gz
echo "*** All done!"
```

This is a script that gets executed and creates the folder.

Searching through history

There's a way to quickly search through previous commands.

`Ctrl+R` opens "reverse search" mode. Enter some pattern and the closest command in history that matches will be shown.

To look into older commands, press `Ctrl+R` again, or `Esc` to abort.

Exercise:

Try it:

1. Press `Ctrl+R`
2. Type in `boom`
3. Press `Ctrl+R` again to see previous commands

Recovering a stuck SSH session

Suppose you're in a terminal editor, and accidentally pressed `Ctrl+S` to save.

Now your terminal is frozen. How to recover it?

Recovering a stuck SSH session

Suppose you're in a terminal editor, and accidentally pressed `Ctrl+S` to save.

Now your terminal is frozen. How to recover it?

1. Specifically for `Ctrl+S`, you can press `Ctrl+Q`.

Recovering a stuck SSH session

Suppose you're in a terminal editor, and accidentally pressed `Ctrl+S` to save.

Now your terminal is frozen. How to recover it?

1. Specifically for `Ctrl+S`, you can press `Ctrl+Q`.
2. Generally, you can send commands to your SSH client by pressing `Enter`, then ~ (tilde), then a command.

Recovering a stuck SSH session

Suppose you're in a terminal editor, and accidentally pressed `Ctrl+S` to save.

Now your terminal is frozen. How to recover it?

1. Specifically for `Ctrl+S`, you can press `Ctrl+Q`.
2. Generally, you can send commands to your SSH client by pressing `Enter`, then ~ (tilde), then a command.
   - **`~.`** kills the SSH session - useful if stuck
   - **`~?`** prints help on other available commands

   This doesn't work on PuTTY, but you can control it from the window icon in the top left.

Custom shell prompt

The variable `$PS1` contains the format template for your shell prompt.

Throughout this training, you saw the following prompt:

```
user@host:~ $
```

You can customize it! For example:

```
user@host:~ $ export PS1="[\t] \u@\h:\w\\n\\$ "

[16:40:00] user@host:~
$
```

Want to control that precisely? Want to add color?

There's [a guide]() for that.

Finding files

The `find PATH` command looks through the filesystem at PATH to find files.

One can then filter the output with `grep`, or use `find`'s own keys for sophisticated filtering.

```
user@host:~ $ find ~/scits-training -name '*.sh'
/home/user/scits-training/scripts/boom.sh
/home/user/scits-training/scripts/beer.sh
```

xargs

The `xargs` command can be used to convert input into arguments of another command.

`xargs COMMAND` will take input and pass it as separate arguments after `COMMAND`:

```
user@host:~ $ find ~/scits-training -name '*.sh' | xargs cat
[contents of both .sh files]
```

This is equivalent to

```
user@host:~ $ cat /home/user/scits-training/scripts/boom.sh \
/home/user/scits-training/scripts/beer.sh
```

Public key authentication: theory

It can be useful to use key authentication instead of standard password authentication.

- Far more secure — suitable for internet-facing computers.
- May be required in cloud environments to set new VMs.
- Allows passwordless authentication for more convenience.

Public key authentication: theory

It can be useful to use key authentication instead of standard password authentication.

- Far more secure — suitable for internet-facing computers.
- May be required in cloud environments to set new VMs.
- Allows passwordless authentication for more convenience.

It is based on modern cryptography and consists of pairs of keys: **public**, which you can give to others, and **private**, that you keep yourself (preferably encrypted with a **passphrase**).

Having the private key allows you to prove that you own the keypair to anyone having your public key, without disclosing the private key itself.

Public key authentication: workflow

When setting up public key authentication on Linux, here's the workflow:

1. You generate a keypair: private key and public key files.

   On Linux/Mac, `ssh-keygen` is used. On Windows, PuTTYgen can be used.

2. You copy the public to the remote system.

3. You connect, instructing SSH to use the private key.

   If it's encrypted, you're asked for the passphrase (and may be cached in an SSH agent after that).

Public key authentication: workflow

On most Linux systems, it is sufficient to have your public key in `~/.ssh/authorized_keys` file.

To copy the keypair to the remote system, `ssh-copy-id` script can be used.

Public key authentication: workflow

On most Linux systems, it is sufficient to have your public key in `~/.ssh/authorized_keys` file.

To copy the keypair to the remote system, `ssh-copy-id` script can be used.

Otherwise, you need to create that file/folder yourself, and make sure they have proper permissions.

```
user@remote:~$ ls -l ~/.ssh
total 12
drwx------  2 user user 4096 Sep 13  2017 .
drwxr-xr-x 13 user user 4096 Sep  2 21:47 ..
-rw-------  1 user user 1159 Jan 21  2018 authorized_keys
user@remote:~$ cat .ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAA[...]j6aKfAUoXOE= some comment
```

Public key authentication: workflow

On most Linux systems, it is sufficient to have your public key in `~/.ssh/authorized_keys` file.

To copy the keypair to the remote system, `ssh-copy-id` script can be used.

Otherwise, you need to create that file/folder yourself, and make sure they have proper permissions.

```
user@remote:~$ ls -l ~/.ssh
total 12
drwx------  2 user user 4096 Sep 13  2017 .
drwxr-xr-x 13 user user 4096 Sep  2 21:47 ..
-rw-------  1 user user 1159 Jan 21  2018 authorized_keys
user@remote:~$ cat .ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAA[...]j6aKfAUoXOE= some comment
```

To use a key, one can use `-i PRIVATE_KEY_FILE` flag for `ssh`, or Pageant on Windows.

A tiny `vim` tutorial

`vim` modes: command mode

A central concept of `vim` is **modes**.

When you first open a file, if you try to start writing you'll (at best) not succeed, and at worst mangle your file and/or dig deeper into `vim`.

This is because `vim` is by default in **command mode**, that uses the entire keyboard for various command shortcuts and not actual text input.

You can always get back to command mode by **pressing `Esc` enough times**.

`vim` command mode: useful commands

From command mode, the following commands are very useful (confirm with Enter):

- `/something` searches for "something".
- `?something` does the same, but backwards.
- `/` or `n` goes to the next match.
- `?` or `N` goes to the previous match.

All of the above works in `less` as well.

- `:NUMBER` goes to line number `NUMBER` - very useful when debugging.

  E.g. `:50` goes to 50th line in the file.

`vim` modes: insert mode

After navigating around the file using arrow keys, `PgUp`/`PgDn` or other commands, you eventually want to edit the text.

Press `i` to switch to **insert mode**:

```
-- INSERT --                                                 1,1           Top
```

Here, you can use your keyboard as usual to edit/input text.

As a reminder, to get back to command mode you should press `Esc`.

Exiting `vim`, eventually

When you're done with editing, go back to command mode, and use one of the commands to exit:

- `:q` will **q**uit Vim if there are no unsaved changes to the file. It will stop you if there are any.
- `:w` **w**rites the changes to file.
- `:w filename` writes the changes to another file, `filename`. It is essentially the "Save As" command.
- `:wq` combines the two: saves current file and exits.
- `:q!` quits forcefully, discarding unsaved changes.

`vim`: minimal survival guide

The absolute minimum knowledge required to use `vim`:

- Use arrow keys to move around in any mode.
- Press `i` to start entering ("inserting") text.
- Press `Esc` before you do anything else.
- `:wq` to write your changes and quit, or
- `:q!` to discard your changes and quit.

Further `vim` help and training is available with `vimtutor` shell command.

Try vim

Exercise:

1. Open the file ten with vim:

```
user@remote:~/scits-training/numbers/ $ vim ten
```

2. Add numbers from 6 to 10 to the end, on separate lines
3. Save and exit vim (hint: write and quit)
4. Verify what's in the file using cat