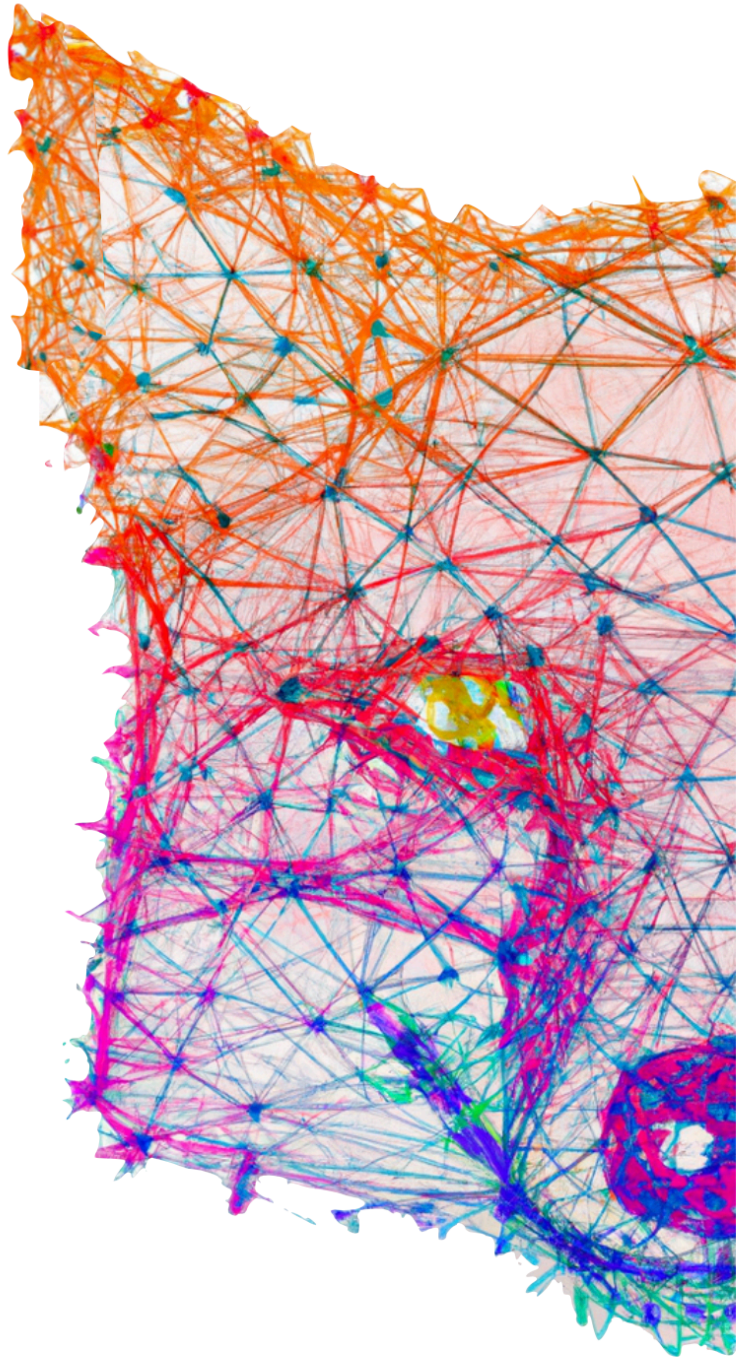
Scientific Computing Basics

What you'll need to make CHACAL a success

Louie Dartmoor Corpe (LPC Clermont)

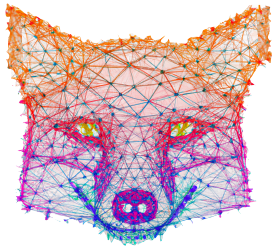
15 Jan 2024



CHACAL 2024

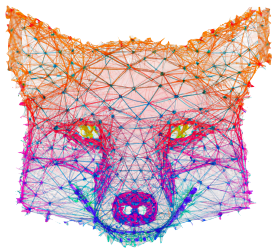
LPC Particules pLasmas Univers applications
Laboratoire de Physique de Clermont

Welcome!



- It's a pleasure to give you a warm **welcome to CHACAL24**
- The objective is to give you a **crash course in advanced computing methods** used in **particle physics**, specifically collider physics at the LHC.
- We'll cover three broad categories:
  - **Monte Carlo simulation**, a cornerstone of data analysis at the LHC
  - **Machine Learning,** one of the most powerful tools at our disposal for data analysis
  - **Quantum Computing** and alternative computing architectures, with an eye to the future
- The aim is that at the end of this school, you **come out with practical skills** to help you make use of these technologies in **your future research**... while forming links and strong cohorts along the way to help you **build your network**.

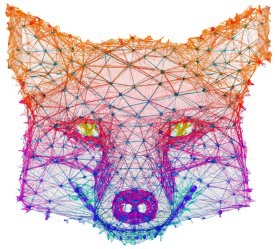
What we need from you



- You'll be getting **lectures (morning)** + **hands-on sessions (afternoon)**
- These will often be **led by world experts**... **make the most of it!**
  - Ask **questions**, be **proactive**, and **have fun**
  - Always be **respectful and patient** with each other and the other lecturers
  - Follow the CERN code of conduct* at all times: be excellent to each other
  - **Don't be shy:** there is no such thing as a dumb question, and we have **plenty of time for questions** and discussion

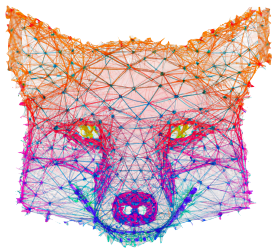
* https://cds.cern.ch/record/2240689/files/BrochureCodeofConductEN.pdf?

What tools do we need?



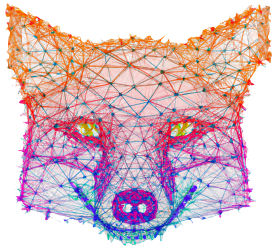
- Almost all the computing done at the LHC is done using **custom software**, developed using **open-source tools and platforms**
- That custom software is usually in three main languages:
  - **C++** (for heavy operations that need to happen fast and where we need precise control of the objects and classes we create. Usually more complicated to write, but executes fast because it's compiled.)
  - **Python** (for data analysis, and operations where time and memory are not an issue... easier to write but usually much slower than C++ as it's not compiled).
  - **Bash** (for navigating on the terminal, book-keeping, automation, etc.)

What tools do we need?



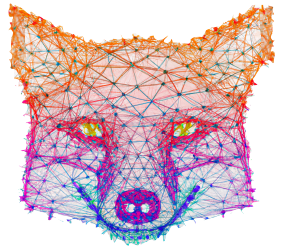
- There are also some other tools which you will need to know about this week:
  - **Jupyter notebooks**: provide an interactive and visual way to write python code. A lot of the tutorials will be done using such notebooks, either locally or via Google Collab (if you don't have one already, now is a good time to make a google account so you can follow those tutorials)
  - **Docker containers:** aka docker images. Containers are a way of taking a snapshot of a whole software stack (operating system and all), which can be run from within another machine. That way, instead of trying (and failing) to install a complicated programme with all its correct dependencies, you can download the docker container and just work inside that. Used heavily in industry, for example in web commerce: if there are more clients buying on your website, you can deploy resources eg on Amazon Webserver instantly by just sending the container with the website backend.

What will we cover today?

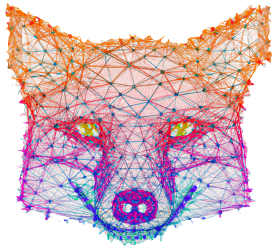


- Today I will review (in this order).
  - Brief introduction to **Docker containers**
  - Bash and the **basics of using a terminal**
  - **Python**: review of the basics, and the most common libraries we will need: numpy, matplotlib, pandas... as well as Jupyter notebooks

- This **afternoon we will get our hands dirty** with some of these skills in the tutorial. But if you have a laptop with you already, feel free to follow along!
- **Tomorrow, you'll learn all about C++** with Caterina.

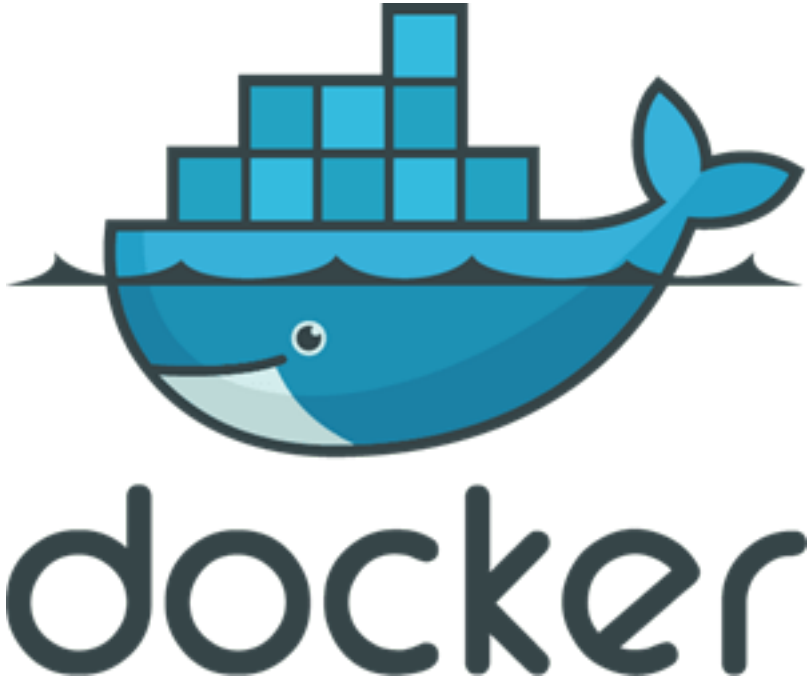


Let's get started!

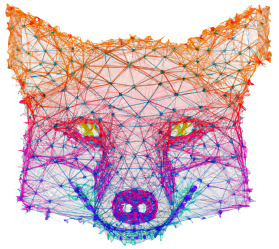


Docker

Software containers

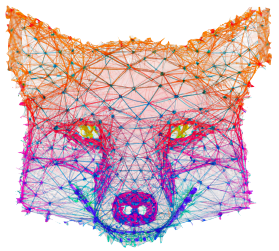


Why are we starting with Docker?



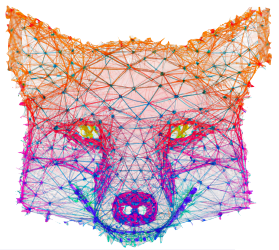
- As we will see in the next chapter, we tend to use **unix-based environments**, especially to interact with the terminal.
  Linux and Mac are unix-based, Windows is not.
- That means that **Windows users may struggle to follow the next section** "Bash and the terminal"... **unless they use a docker container!**
- We are staring with Docker to give a chance to **people with Windows machines to install docker** and load a simple linux-based image so they can follow along !

What is docker?



- **Docker** is an open platform which allows you to **open and run software containers**.
- Ok, but **what is a software container**???
  - A **software container** is a standard **unit of software that packages up code and all its dependencies** so the application **runs quickly and reliably from one computing environment to another**.
    - For example, so you can run a unix terminal tutorial on a windows machine ;)
  - In English: it's a box which has a **particular programme and all its dependencies** (down to the operating system) **compiled and runnable** from any operating system
- That means you can have a container for, for example **"Rivet installed on linux"** with versions x.y.z. You can **open that container on any machine** (windows, Mac, Fedora...) and it will be **"as if" you were running on the linux machine with Rivet installed**.

How to install it?



- First, you need a **docker engine**: see https://docs.docker.com/engine/install/
  - Suggest to use "Docker Desktop"
  - There are detailed installation instructions for all platforms
  - Probably best not to overload the wifi by all installing at once: Windows users please go first!

Manuals / Docker Engine / Install / Overview

Install Docker Engine

This section describes how to install Docker Engine on Linux, also known as Docker CE. Docker Engine is also available for Windows, macOS, and Linux, through Docker Desktop. For instructions on how to install Docker Desktop, see:

- Docker Desktop for Linux
- Docker Desktop for Mac (macOS)
- Docker Desktop for Windows

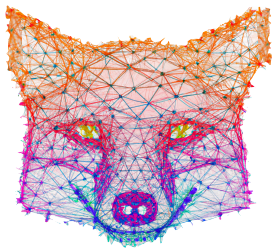
Install Docker Desktop on Windows

Install interactively | Install from the command line

1. Download the installer using the download button at the top of the page, or from the release notes.
2. Double-click `Docker Desktop Installer.exe` to run the installer.
3. When prompted, ensure the **Use WSL 2 instead of Hyper-V** option on the Configuration page is selected or not depending on your choice of backend.
   If your system only supports one of the two options, you will not be able to select which backend to use.
4. Follow the instructions on the installation wizard to authorize the installer and proceed with the install.
5. When the installation is successful, select **Close** to complete the installation process.

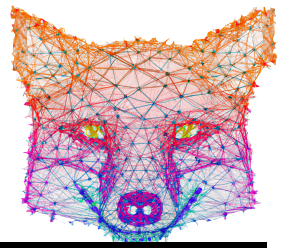
If your admin account is different to your user account, you must add the user to the **docker-users** group:

Container versus image



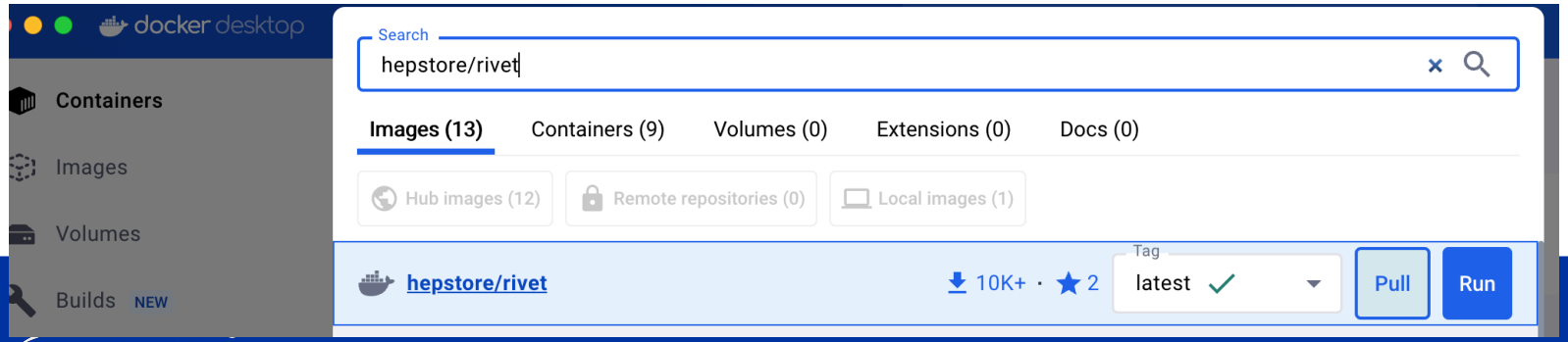
- Technically speaking:
  - A **software image** is the bundle of software and dependencies.
    You can see a list of available images at Docker hub
    https://hub.docker.com/
  - A **software container** is the instance of an image which you are running via docker.
- You can have multiple **containers** running with the same **image.**
- **In practice we use these terms interchangeably**

How to use it?

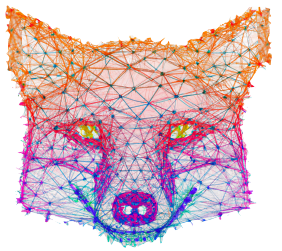


- We want to pick a simple image, for example with Rivet installed on ubuntu
- Thankfully there is one: https://hub.docker.com/r/hepstore/rivet
- **In a terminal**:
  `docker pull hepstore/rivet`
  will download the image
- **In docker desktop**: search and pull

```
louiecorpe_cern2@clratlmac02 ~ % docker pull hepstore/rivet
Using default tag: latest
latest: Pulling from hepstore/rivet
405f018f9d1d: Pull complete
32cb6d73f0f5: Pull complete
93b3370bbc3c: Pull complete
f617de0c3b09: Downloading [==================>
                  ]  66.24MB/156.6MB
3db6073e5257: Download complete
5ae4a5e6c533: Download complete
f617de0c3b09: Pull complete
3db6073e5257: Pull complete
5ae4a5e6c533: Pull complete
ecf46c197c08: Pull complete
bb4112a6518: Pull complete
3c2963256b9a: Pull complete
35cc04d89655: Pull complete
16e217af4e0e: Pull complete
58527630d45f: Pull complete
13c685819f40: Pull complete
a3f7acf4f8b9: Pull complete
316f21ee06a6: Pull complete
9f4590889b3c: Pull complete
793cbe07d8dc: Pull complete
50660d215dd: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:dfcee582fe4008509e0ce670c1e9d7b19e8a64c47645e
04f7aa9ec018e5bcfc7
Status: Downloaded newer image for hepstore/rivet:latest
docker.io/hepstore/rivet:latest
```



How to run it?

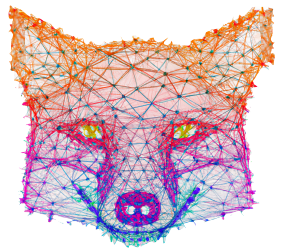


- **From the terminal**:
- You can then "enter" the image **`docker run -it hepstore/rivet`**
  and you can map to your local directories to access your local files and write new files out
- **`docker run -it -v $PWD/myfiles:/work/myfiles hepstore/rivet`**

Run the container in interactive mode Mapping this local folder to a folder in the container (-v for volume mount) Using this image

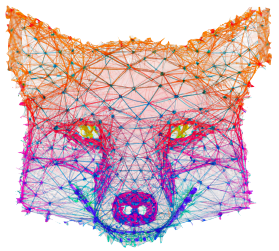
```
louiecorpe_cern2@clratlmac02 ~ % docker run -it -v $PWD/myfiles:/work/myfiles hepstore/rivet
ERROR: must "cd where/root/is" before calling ". bin/thisroot.sh" for this version of "qemu-x86_64"!
root@af6721063975:/work#
root@af6721063975:/work#
root@af6721063975:/work# ls
myfiles
root@af6721063975:/work# rivet --help
usage: rivet [-h] [--version] [-a ANA] [--list-analyses] [--list-keywords] [--list-used-analyses] [--show-analy
             [--analysis-path-append PATH] [--pwd] [-o HISTOFILE] [-p PRELOADFILES] [--no-histo-file] [-x XS] [
             [--unmatch-weights UNMATCH_WEIGHTS] [--nominal-weight NOMINAL_WEIGHT] [--weight-cap WEIGHT_CAP] [-
             [--event-timeout NSECS] [--run-timeout NSECS] [-l NATIVE_LOG_STRS] [-v] [-q]
             [ARGS ...]

Run Rivet analyses on HepMC events read from a file or Unix pipe Examples: rivet [options] <hepmcfile> [<hepmcf
[options] fifo.hepmc Environment variables: * RIVET_ANALYSIS_PATH: list of paths to be searched for plugin anal
for data files (defaults to use analysis path) * RIVET_REF_PATH: list of paths to be searched for reference dat
be searched for analysis info files (defaults to use data path) * RIVET_PLOT_PATH: list of paths to be searched
RIVET_ANALYSIS_PLUGINS: list of paths to be searched for data files (defaults to use data path) * RIVET_RANDOM_
RIVET_STRIP_HEPMC: reduce the complexity of HepMC events before analysis (experimental)
```

How to run it?



- **From the docker desktop**:
- You can easily "run" docker containers from the Desktop app, but these tend to close immediately because we are not asking them to do anything particular.
- So we can't easily run a docker container interactively via the Docker desktop unless you start it from the command line.
- If anyone knows how to die it without passing via the command line, let me know!

Once you are in...



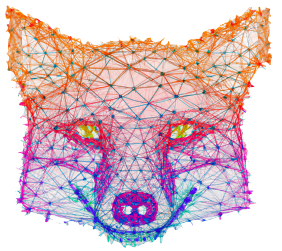
- Now your command line is no longer the one from your original machine (Windows, Mac, Linux..)
- But it's as if you were in a linux machine with rivet and all dependencies installed!
- Cool 😎

```
louiecorpe_cern2@clratlmac02 ~ % docker run -it -v $PWD/myfiles:/work/myfiles hepstore/rivet
ERROR: must "cd where/root/is" before calling ". bin/thisroot.sh" for this version of "qemu-x
86_64"!
root@503c60065bc8:/work#
root@503c60065bc8:/work#
root@503c60065bc8:/work# uname -r
6.5.11-linuxkit
root@503c60065bc8:/work# rivet --help
usage: rivet [-h] [--version] [-a ANA] [--list-analyses]                    d-analy
ses] [--show-analysis SHOW_ANALYSES] [--show-bibtex] [--
             [--analysis-path-append PATH] [--pwd] [-o H                    -no-his
to-file] [-x XS] [-n NUM] [--nskip NUM] [--skip-weights] [--match-weights MATCH_WEIGHTS]
             [--unmatch-weights UNMATCH_WEIGHTS] [--nominal-weight NOMINAL_WEIGHT] [--weight-
cap WEIGHT_CAP] [--nlo-smearing NLO_SMEARING] [--runname NAME] [--ignore-beams] [-d NUM]
             [--event-timeout NSECS] [--run-timeout NSECS] [-l NATIVE_LOG_STRS] [-v] [-q]
             [ARGS ...]

Run Rivet analyses on HepMC events read from a file or Unix pipe Examples: rivet [options] <h
epmcfile> [<hepmcfile2> ...] or mkfifo fifo.hepmc my_generator -o fifo.hepmc & rivet [options
```

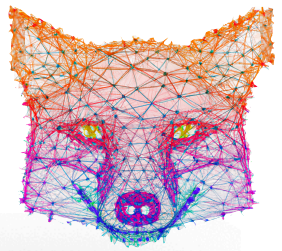
OS is linux ... on my Mac!

Rivet available

Recap on docker

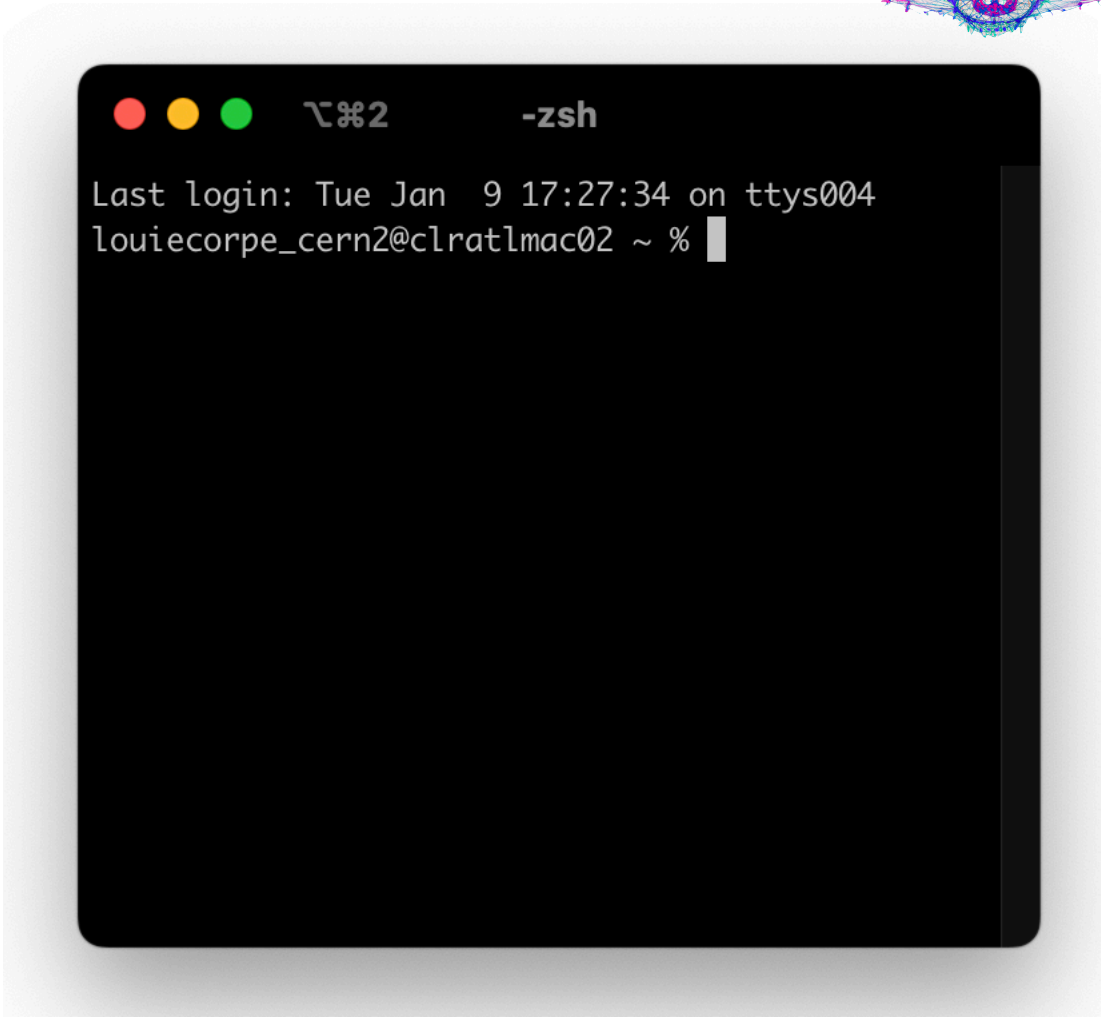


- **Docker images encapsulate a software and all it's dependencies** (including the OS)
- We can run download and run images in containers so that we can **use that software without worrying about how to install it on our machine**.
  - Almost any software can run on any platform
- We learnt how to **search for images** on Docker Hub
- We learnt how to **download those images** (= pull them from the hub)
- We learnt how to **execute them interactively** so that we can be "inside" the container

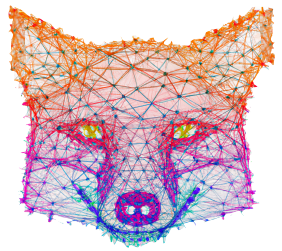


Bash

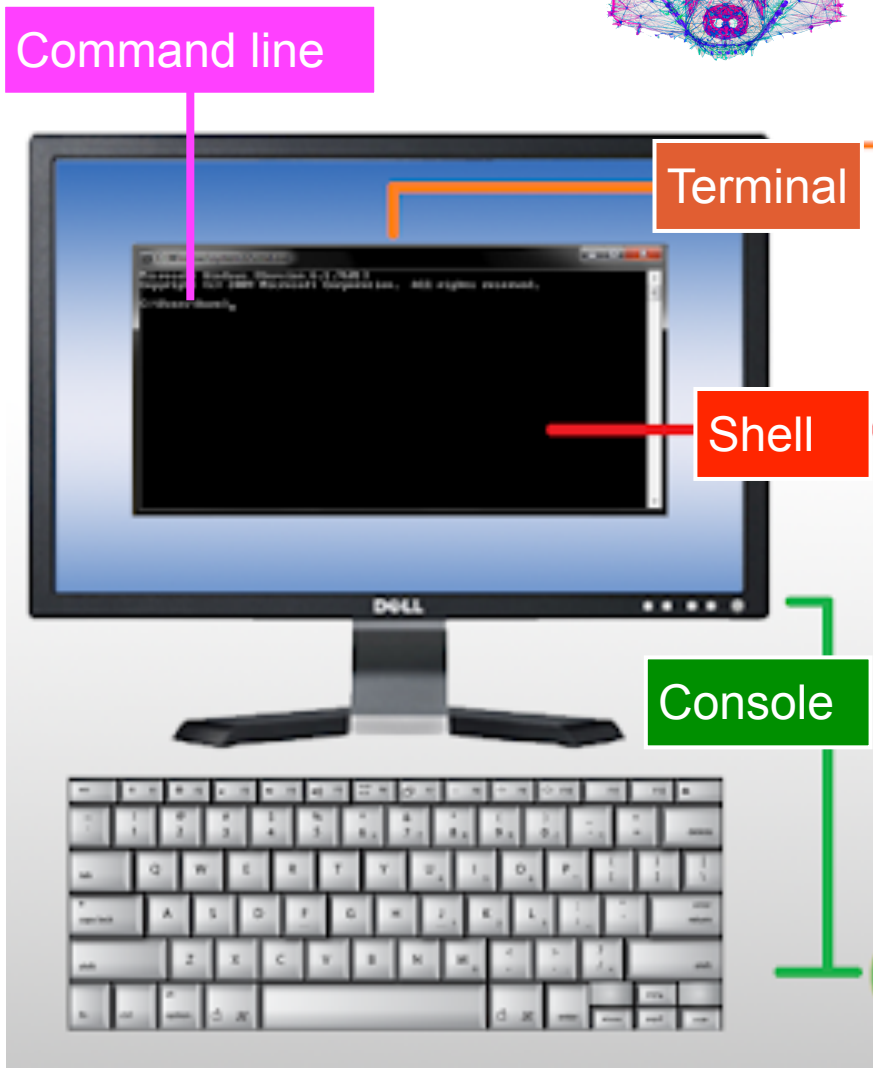
Or how I learned to stop worrying and love the terminal



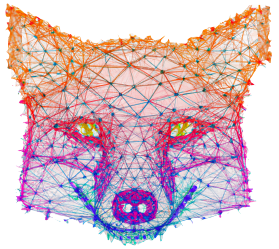
Terminals, consoles, shells...



- Technically speaking...
  - A **console** is a piece of physical hardware that allows direct interaction with a computer system. Includes things like keyboard and monitor...
  - A **terminal** is a device or programme that provides an interface to interact with the computer system. In the past this was a physical device of some sort, these days they are software-based.
  - A **shell** is a command-line textual interpreter that allows us to send and execute commands and view the output. There are lots of different kinds of shell, the most common being Bash on unix-like systems (Linux, Mac), Command Prompt on Windows.
  - A **command line** refers to the place where you write down the commands which are used to interact with the computer.
- **In practice all these terms are usually used interchangeably!**

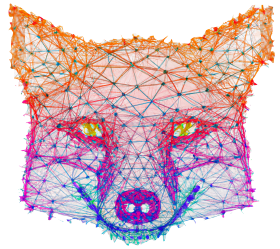


Operating systems, or unix vs windows



- **Windows uses a closed-source operating system**. How it executes the commands we run is not public information. It also means **we don't have full control**.
- In particle physics, we therefore opt for **operating systems based on "unix"**, which are **open-source at least with regards to the command-line interface.**
  - **Linux:** the whole operating system is open source, uses bash.
  - **Max OS:** the operating system is in general closed source, but the command line is unix-like.
- Since **bash (or similar) is the default on unix-like systems**, this is what we will use. Windows uses Command Prompt or PowerShell, which are not widely used in particle physics. So we won't cover those.

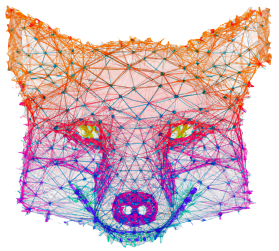
Your first command



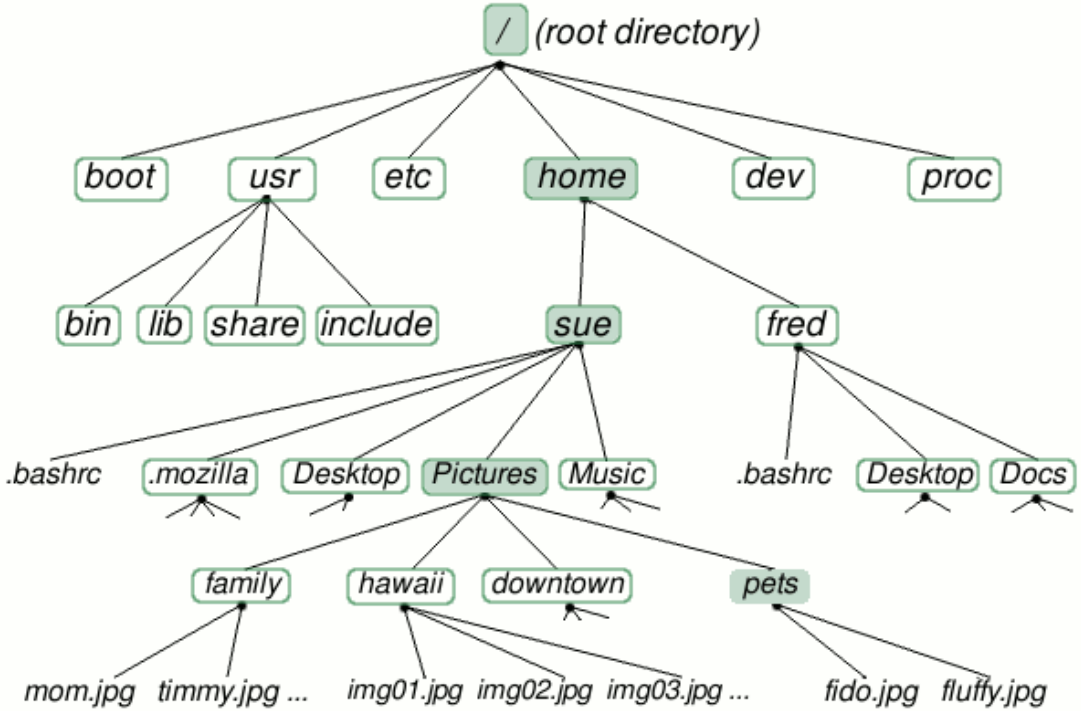
```
louiecorpe_cern2@clratlmac02 ~ % echo "hello world"
hello world
```

- `echo` is a command which just prints back the argument you specified.

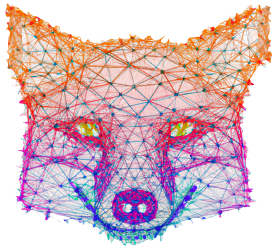
Unix file structure



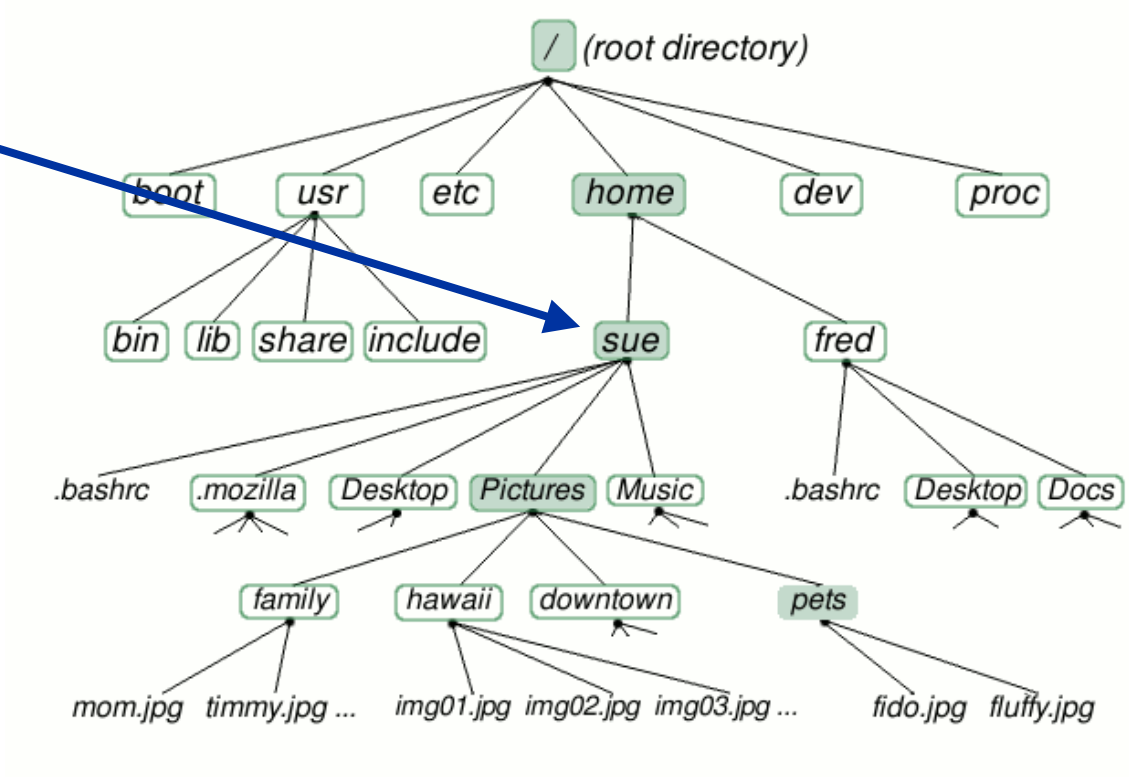
- Your computer's file system is organised as a tree of
  - **Directories** (=folders)
  - **Files** (.txt , .cc, .py, .jpg ...)
    - Some files (like text files, csv, code...) are **human readable** and can be edited directly ir a text editor
    - Other files are in **binary formats** (compiled code, root files, images...) that need to be decoded to be viewed.



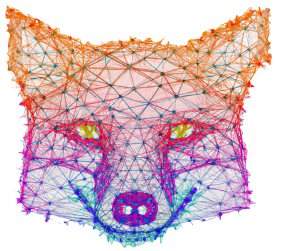
Basic navigation



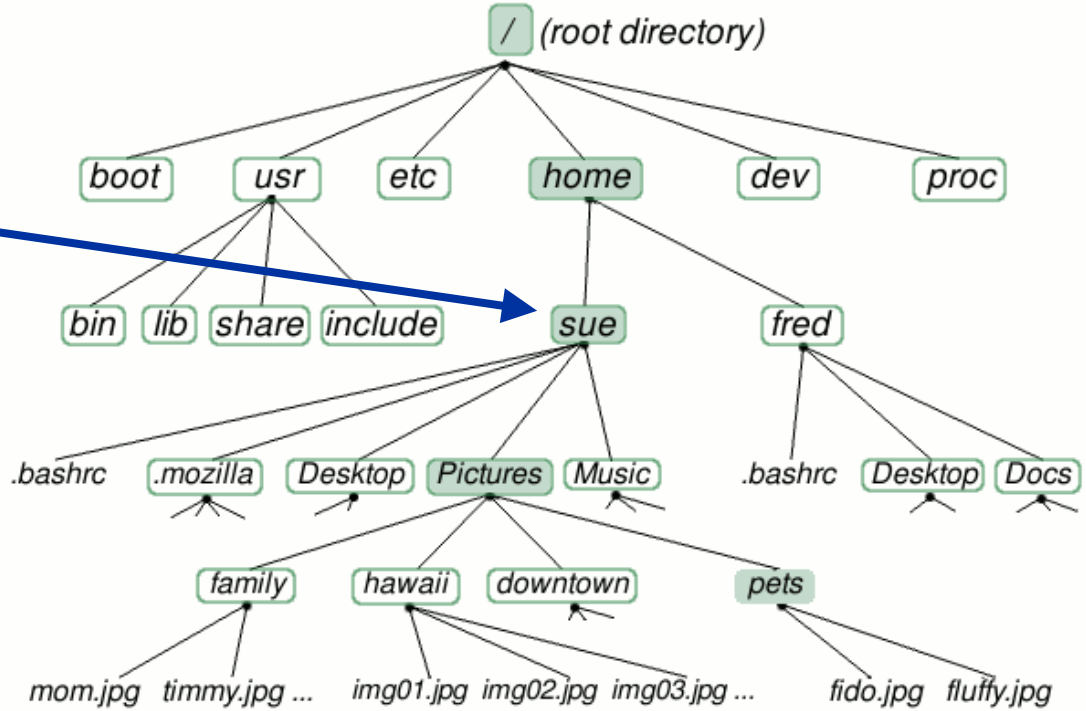
- When you log into your terminal, you start in your `home/<username>` directory



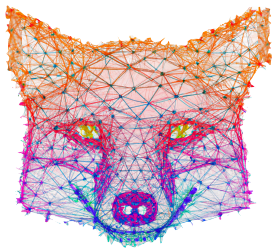
Basic navigation



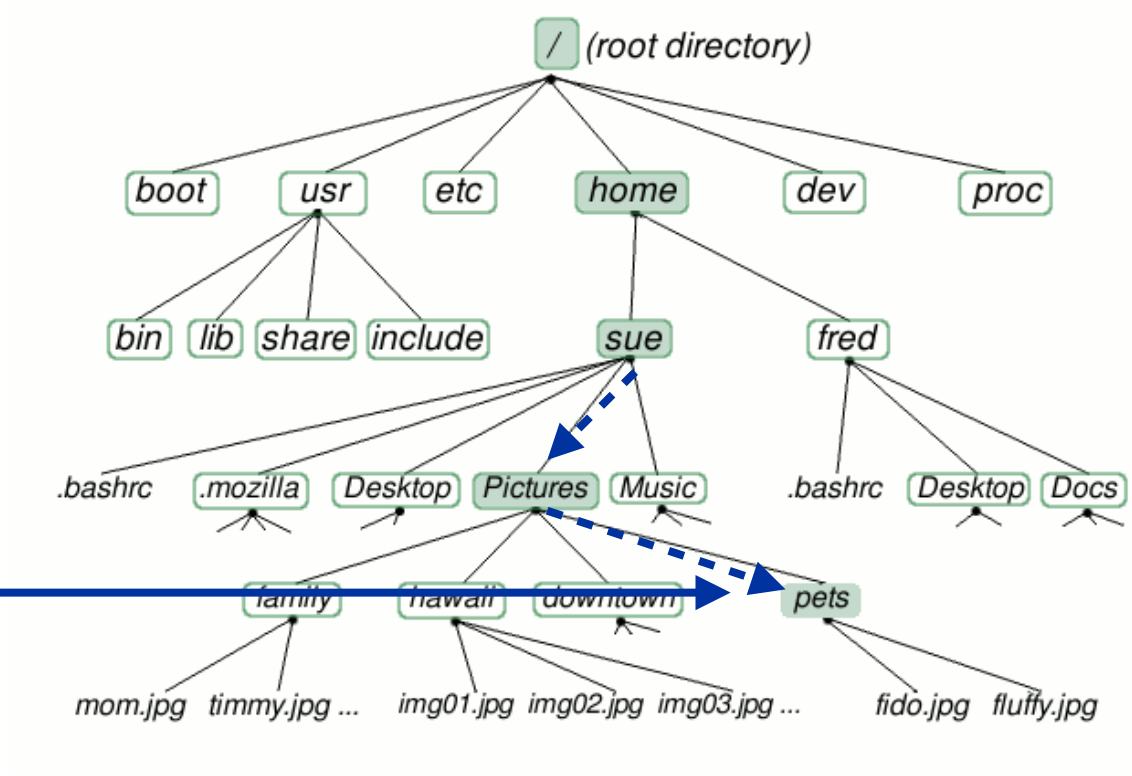
- When you log into your terminal, you start in your `home/<username>` directory
- Your location is given by a path, here: **`/home/sue`**
- you can check your current path at any time by typing **`pwd`** (print working directory)



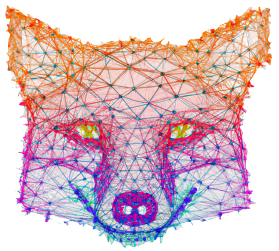
Basic navigation



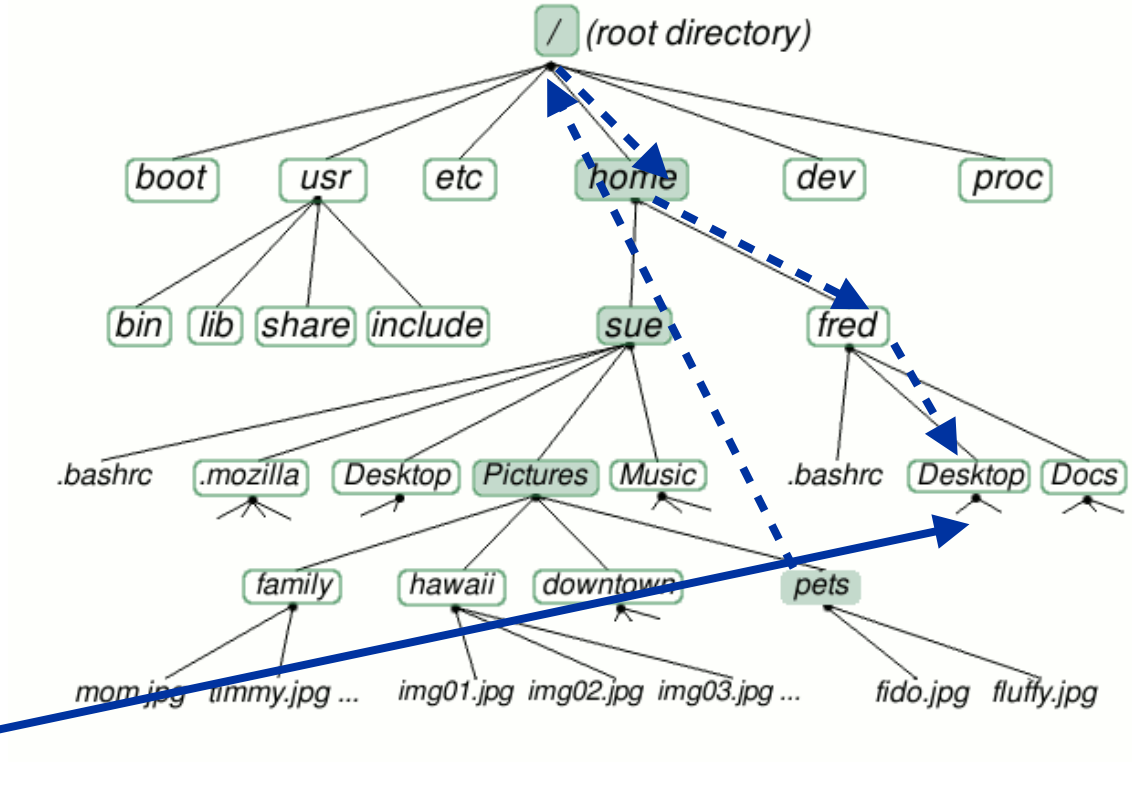
- When you log into your terminal, you start in your `home/<username>` directory
- Your location is given by a path, here: **`/home/sue`**
- you can check your current path at any time by typing **`pwd`** (print working directory)
- You can change your location using the **`cd`** (change directory) command either by specifying a relative path from your current location
  `cd Pictures/pets`



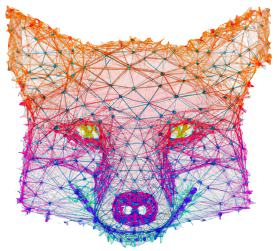
Basic navigation



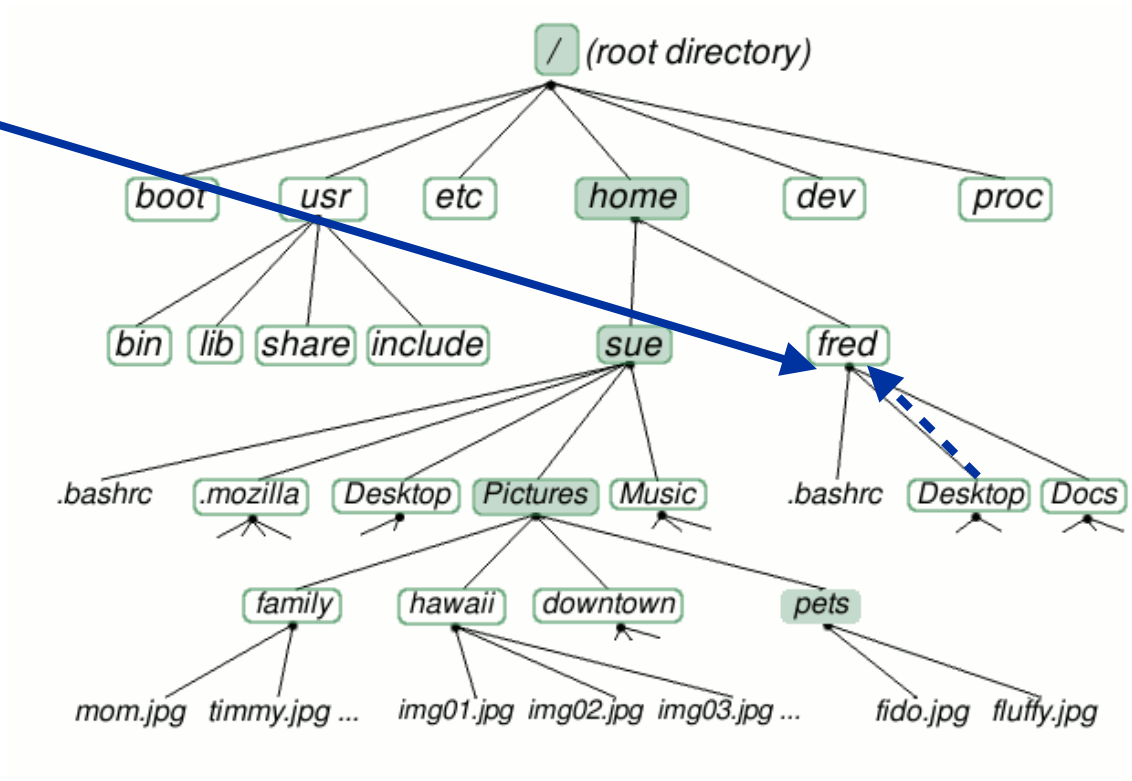
- When you log into your terminal, you start in your `home/<username>` directory
- Your location is given by a path, here: **`/home/sue`**
- you can check your current path at any time by typing **`pwd`** (print working directory)
- You can change your location using the **`cd`** (change directory) command either by specifying a *relative path* from your current location
  `cd Pictures/pets`
- or specifying an *absolute path* from the root directory.
  `cd /root/home/fred/Desktop`



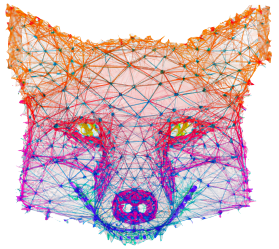
Basic navigation



- When specifying a relative path, you can use two dots "**..**" To indicate, "go back one directory towards root". A single dot means "here".
  `cd ..`

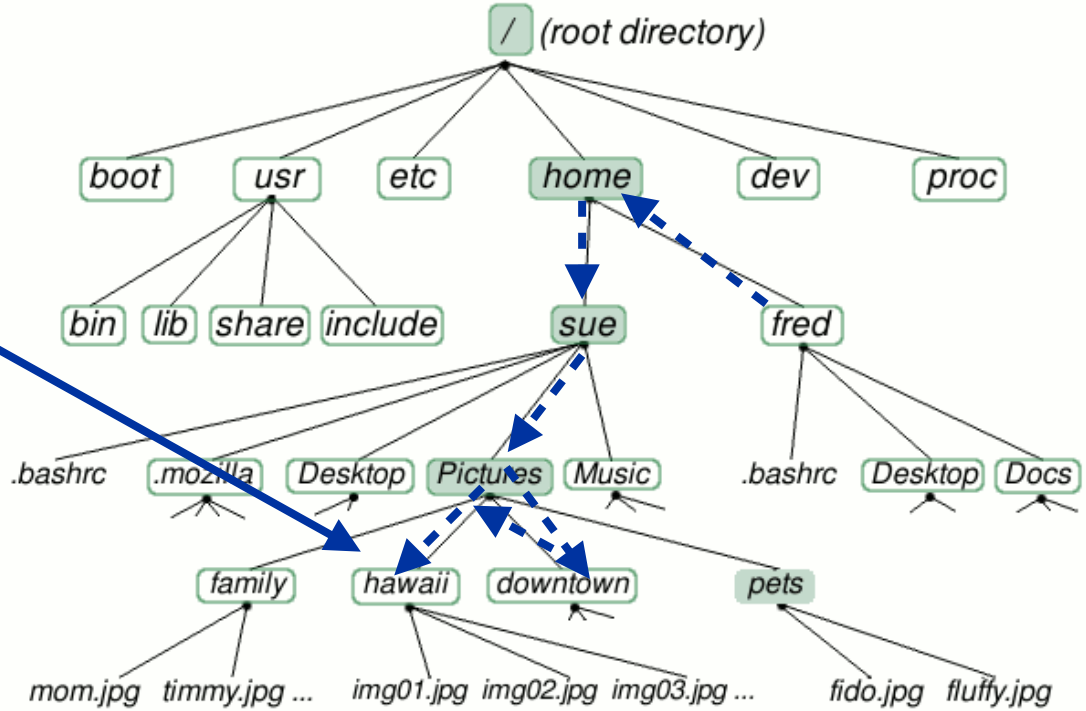


Basic navigation

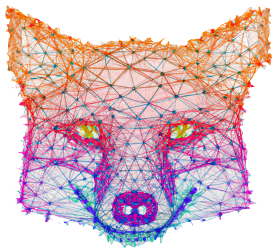


- When specifying a relative path, you can use two dots "**..**" To indicate, "go back one directory towards root". A single dot means "here".

```
cd ..
cd ../sue/Pictures/downtown/../hawaii
```

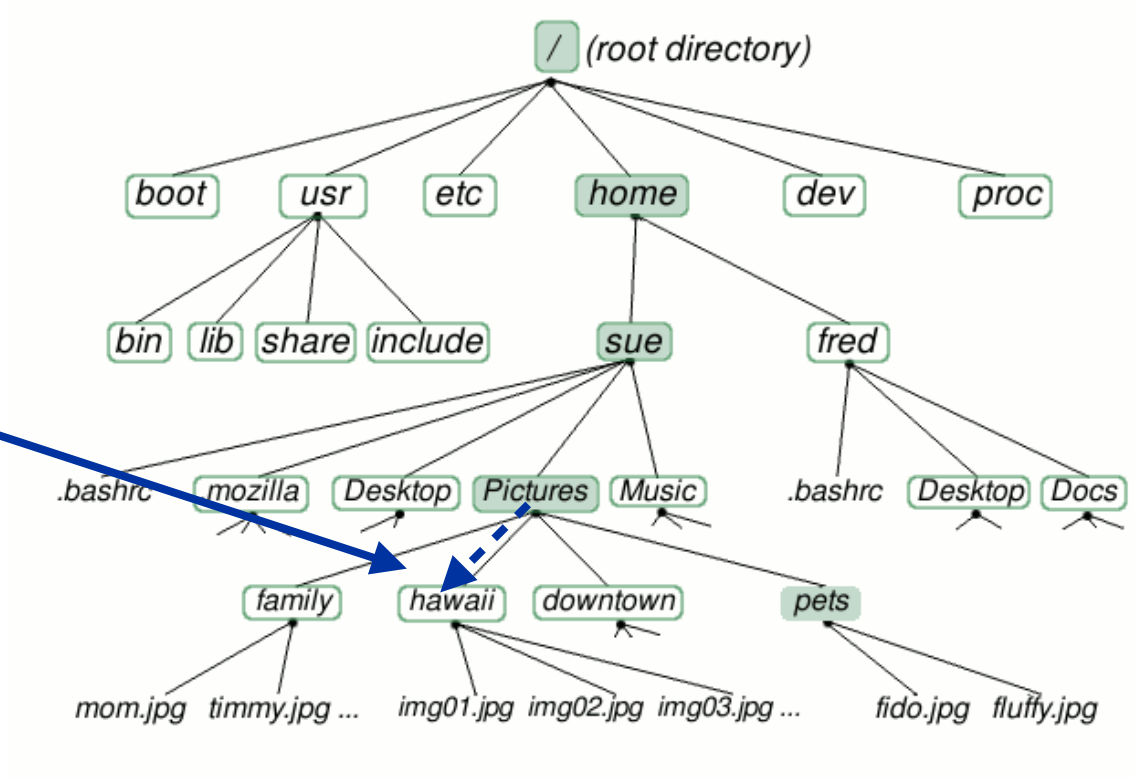


Basic navigation

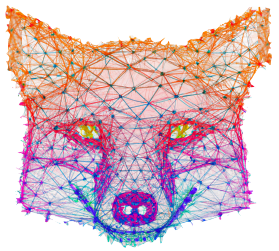


- When specifying a relative path, you can use two dots "**..**" To indicate, "go back one directory towards root". A single dot means "here".
  `cd ..`
  `cd ../sue/Pictures/downtown/../hawaii`

- To inspect the contents of a directory or get information about a file, use the **ls** (list) command
  `ls`
  (or ls `/home/sue/Pictures/hawaii`)
  > img01.jpg img02.jpg img03.jpg...

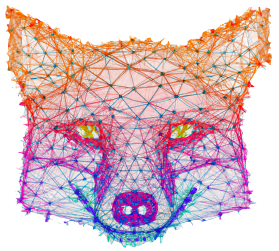


Creating and manipulating directories



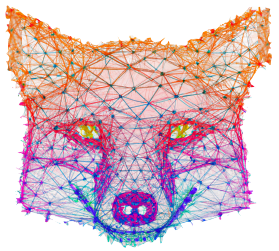
- You may want to create new file structures
  - `mkdir` (make directory). Use options -p if creating several nested directories at once
    eg: `mkdir -p /home/louie/path/to/my/project`
  - `mv` (move a directory or file). Format: mv <object to move> <where to move it to>
    Eg `mv chacal /home/louie/path/to/my/project/.`
    (what happens if `mv chacal /home/louie/path/to/my/project2` ?)
  - `cp` (copy a directory or file). Instead of updating the location, make a copy of a file or directory in a new location. Use `-r` (recursive) if copying a directory.
    eg `cp -r /home/louie/path/to/my/project2 /home/louie/path/to/my/project3`
  - `rm` (remove a directory or file). **Permanently** remove a file or directory. Use `-r` for directories. **USE WITH GREAT CAUTION, ALWAYS DOUBLE CHECK YOU ARE REMOVING WHAT YOU THINK YOU ARE.**

Files



- Inside your directories, you will have files.
  - **Text-based files** are eg code, configuration files, simple data formats like csv... these are usually designed to be written or edited by a human user.
  - **Binary files or custom formats,** which can't be opened by a text editor and need to be either decoded or opened with custom software. Human-readable is generally not an efficient format to store data in. That's why we designed more efficient formats, but which are no longer human-readable. Examples are images, sound files, databases, root files, compiled code...
  - We'll focus on human-readable text files in the following.

Common operations on text files



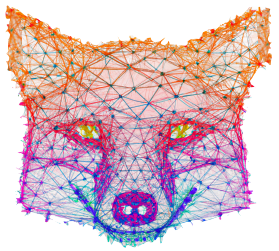
- **`>` or `>>`** (send or append). You can send or append something that would otherwise be printed to your screen towards a file.
  Eg **`echo "hello world" > file.txt`**

- **`ls -l`** (list, option "long") give info when the file was created, who created it, its size, and permissions. Eg **`ls -t file.txt`**
  `> -rw-r--r--  1 louiecorpe_cern2  staff  12 10 Jan 10:32 file.txt`

- **`more` or `less`** (see "more" of the file, ie a preview of file contents). This opens a basic interface to see what's in the file, which then disappears when you exit. You cannot edit the file. Eg **`less file.txt`**

- **cat** (concatenate) allows you to print the contents of the file to your screen.
  eg **`cat file.txt`**
  eg **`cat file.txt > file2.txt`**

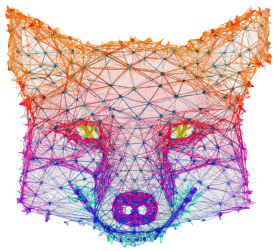
Text editors



- You won't get very far editing files by appending from the command line!
- To modify or write code or other files, you'll need a **text editor**.

- **There is an ancient and passionate argument** between those who use **vim** and those who use **emacs** as a their text editor.
  - I will give you a VERY BRIEF overview of how to use each.
  - I will try to be unbiased, but you will probably be able to tell which side of the argument I am on.
  - In reality, there are better tools, like VisualStudio, which real computer programmers use.
- If one of the other is not installed, you can usually easily do so via command line, eg for Ubuntu (Linux)
  `apt install vim` or `apt install emacs`
- The exact command to use to install a package depends on your operating system

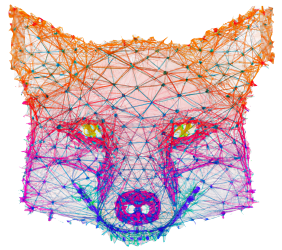
Vim in a nutshell

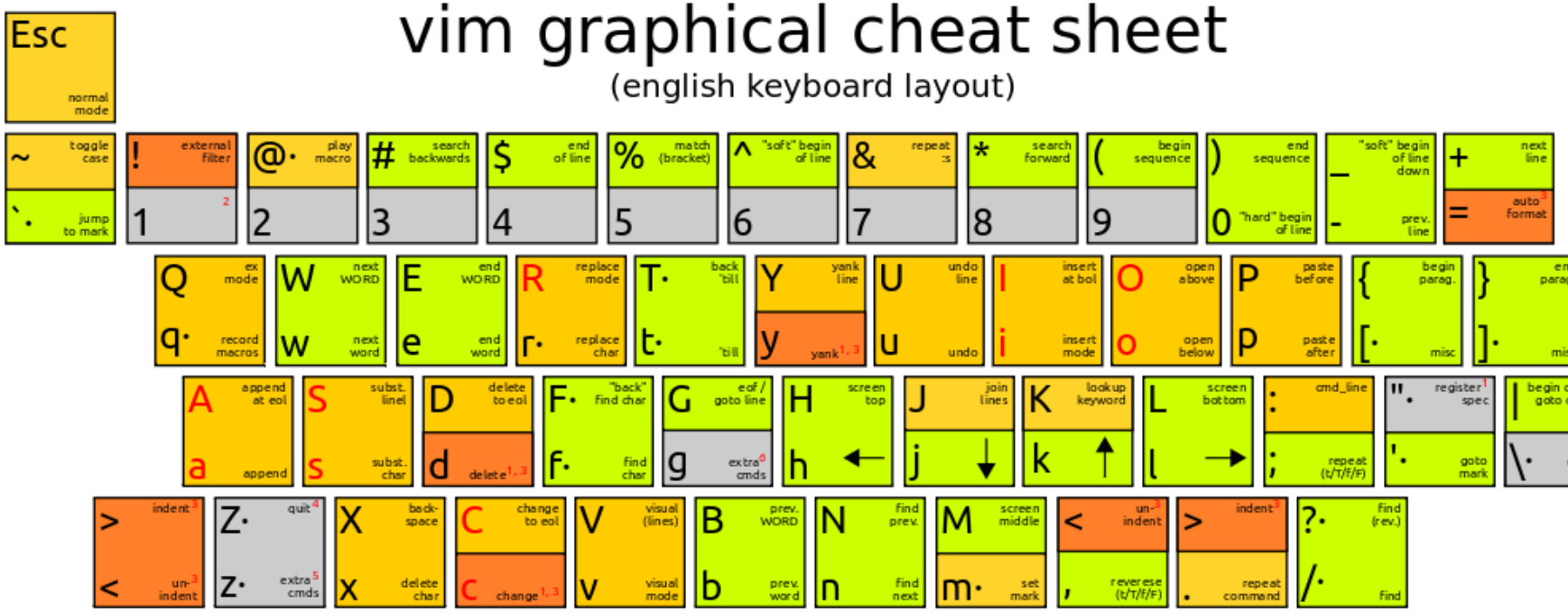


- Open a new or existing file like so **`vim my file.txt`**
- By default you are in command mode. This allows you to type certain commands to do operations like find and replace etc.
- By default, you navigate through the text via your keyboard.
  In **COMMAND mode**, type "**`:set mouse=a`**" which means your cursor will go to where you click.
- To **start editing text,** put your cursor to where you want to modify text and type "**`i`**". You are now in **INSERT mode**, and you can type and modify the text like in a "regular" text editor. When you are done, hit **esc** to return to COMMAND mode.
- You can **undo** your last change using "**`u`**"
- To **save your changes**, in COMMAND mode do "**`:w`**" (write). To quit do "**`:q`**".
- You may be asking... why is it so complicated?
  Vim becomes VERY powerful once you get the hang of it for rapid modifications to text.

vim graphical cheat sheet

(english keyboard layout)





motion	moves the cursor or defines the range for an operator
command	direct action cmd, if red it enters insert mode
operator	requires a motion afterwards, operates between cursor and destination
extra	special functions, requires extra input
q·	commands with a dot need a char argument afterwards

Main command line commands ("ex"):

:w [file] {save}
:q {quit}
:q! {quit without saving}
:wq {save & quit}
:e foo {open file foo}
:n {new file}
:sp {split window horizontally}
:vsp {split window vertically}
:reg {display content of named registers}
:Explore [dir] {open file-explorer}
:h {help}
:h holy-grail {list all commands}

Other important commands:

CTRL + R (redo)
CTRL + P / N (complete the current word)
CTRL + W (move cursor to next window)
[n] CTRL + 6 (toggle [n]th alternate file)
CTRL + F / B (page up / down)
CTRL + E / Y (scroll lin eup / down)
CTRL + V (block-visual mode)

Find and replace:

:%s/<RegExp>/<String>/g (replace <RegExp> by <String> filewide)
:s/<RegExp>/<String>/ (search current line and replace first match)

Vim 7.x only commands:

CTRL + X - CTRL + O (omni completition in insert mode)

:tabe [File] {edit [File] in a new tab}
:tabc [n] {close tab [n]}
:tabonly {close all other tab pages}
:tabmove [n] (move tab to position [n]}
:tab [cmd] {execute [cmd] and when it opens a new window open a new tab page instead, e.g :tab split opens current buffer in a new tabm :tab help gt opens tab page with help for gt}
:tabs {list all tab pages}
[n] gt {goto next or tab [n]}
gT {goto previous tab}

:undolist {list leafs in tree of change}
:earlier [n] [s/m/h] {goto older text state [n] times / sec / min / hours}
g- {goto older text state}
:later [n] [s/m/h] {goto newer txt state [n] times / sec / min / hours}
g+ {goto newer text state}

Notes:

[1] use "x before a yank/paste/del command to use that register ('clipboard') (x=a..x,*) (e.g.: "ay$ to copy rest of line to reg 'a')
[2] type in a number before any action to repeate it that number of times (e.g.: 2p, d2w, 5i, d4j)
[3] duplicate operator to act on current line (dd = delete line, >> = intent line)
[4] ZZ to save & quit, ZQ to quit w/o saving
[5] zt: scroll cursor to top, zb bottom, zz: center
[6] gg: top of file (vim only), gf: open file under cursor (vim only)

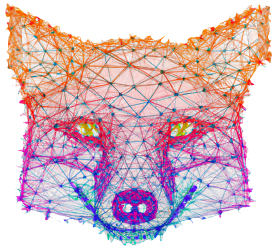
Visual mode:

Move around and type operator to act on the selected region.

Vim-Help navigation:

CTRL + ALT GR +] or :ta [:tag] (jump to subject using tags, CTRL + O to jump back)

Emacs in a nutshell

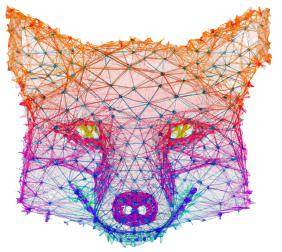


- emacs is in some ways more intuitive but (for reasons which are more philosophical than rational at this point) extremely frustrating for people who are used to vim.
- ~~First, uninstall emacs and use vim instead~~
- Open a file as **emacs file.txt.** Beware this opens a new window (why is this the default?!). You can get it to open directly in there terminal using **emacs -nw file.txt**
- Emacs then behaves more like a "normal" text editor, you can go ahead and start editing your text freely. Emacs will regularly auto-save.
- ~~Saving and quitting involve claw-like operations which emacs users pretend are perfectly normal.~~
- Emacs also has a sort of COMMAND more, which you access via **ctrl+x**
- Once in COMMAND mode you can save by hitting **ctrl+s**, and close using **ctrl+c**

HOW TO LEARN EMACS

A beginner's guide to Emacs 24 or later
Sacha Chua (@sachac) · sachachua.com/begin-emacs

Questions? I'd love to hear from you! 2013(v2)



STEP Ø (for geeks)

If you're a developer or sysad...

Learn Vim

→ the other text editor

It's okay. Learn the basics so that you can easily work on other people's computers. If you know your way around Vim, people won't give you as much grief over Emacs.

Bonus mini-cheatsheet!

Here's what you need to know in Vi:

i	insert mode ↔ <Esc> command mode
Long-time Vi user trying out Emacs?	:vimtutor
	:w write/save file
	:q quit
	:q! really quit

evil mode Emacs

Emacs? You're never installing that on my box

You will probably meet editor zealots. Be ambitextrous!

You can actually edit remote files in Emacs, but that's an intermediate topic. (Curious? See TRAMP)

server ↔ ssh/ftp ↔ emacs

Okay. Once you know the basics of Vim, you can get on with learning Emacs. ☺

Why Emacs?
- Amazingly customizable
- Endless room for growth

Most people start here!

STEP 1

Learn how to learn

There are some old books on Emacs, but the version differences can be rather confusing.

Start with the built-in tutorial instead.

Help → Emacs Tutorial ← repeat as many times as you need to

Other resources:

emacswiki.org
Lots of resources

planet.emacsen.org
Emacs-related blogs

IRC: irc.freenode.net → #emacs
Great for help and hanging out

Can't use the menu? Press Control-h t to start.

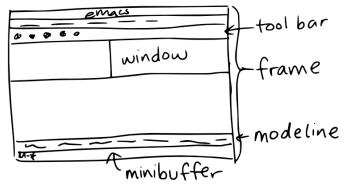
NOTE: The Emacs tutorial has lots of weird terms: "Meta key" "frame" "buffer"... This is because Emacs started a (long) time ago. Don't worry, you'll get the hang of it with practice.

Some things that might help:

C-x C-s: This is how keyboard shortcuts are written.
① press Control & x at the same time
② then let go of x and press Control & s

Tip: Since you're using Control for both keys, you can hold Control down instead of letting go between x and s.

M-x (Meta, usually Alt- or ⌘-) lets you call commands by name, which is great if you can't remember the keyboard shortcut. Ex: M-x help-with-tutorial RET (This means press Enter/Return)



windows show buffers, which could be
- files
- processes
- other info

Other good help commands:
C-h i manual
C-h k <keyboard shortcut> gives you help on it
C-h f describes commands/functions
C-h a searches for commands
C-h C-h shows help

STEP 2

Learn Emacs basics

C-x C-f open (find-file)
C-x C-s save
C-x C-c quit

NOTE: you don't need to quit Emacs after each file. Just leave it running and use C-x C-f to open the next.

How to select text
- Go to the start of your selection and press C-SPC (Control + space)
- Go to the end of your selection and run your command

C-w cut (kill)
C-y paste (yank)
M-y paste older things
C-/ undo

* Learn how to use Keyboard macros. They're awesome.
C-x (start macro
C-x) end macro
C-x e execute macro
e ... ...again

STEP 3

Extend & customize

M-x load-theme RET
Try out color themes → use daily to see the list and list-packages to add more

M-x customize-group RET
set common options

M-x customize-face RET
change background, foreground, etc.

M-x list-packages RET
install lots of modules... → want a prettier Emacs? Check out color themes! (Lots of people like color-theme-solarized)

and then...

editing your ~/.emacs.d/init.el file!

Huh? Where? Just use C-x C-f to create it!

(...(...))
(... ... ...)
(..... ...)
← initializes your Emacs, adds new functionality, and so on.

See emacswiki.org for lots of examples

Use M-x eval-buffer or restart Emacs to see the changes.

Broke your Emacs config? } emacs -q skips your customizations

STEP 4

Learn other handy tips

Buffer & window management

C-x b	switch buffer → even better with M-x ido-mode
C-x 2	split
C-x 3	split
C-x o	other window
C-x 0	get rid of current window
C-x 1	get rid of other windows

Navigation & Search

M-g M-g	go to line (hold Alt/⌘ and then press g twice)
C-s	Interactive search
C-r	Interactive search backward
M->	End of buffer
M-<	Beginning of buffer
M-x occur RET	Find lines

STEP 5

Explore!

Org-mode.org
organize your life in plain text

Narrowing/Widening focus.

TRAMP
remote access

Eshell / Term
command-line in Emacs

Calc
powerful calculator and converter

Writing & debugging Emacs Lisp (edebug)
(it sounds scary, but it's powerful ☺)

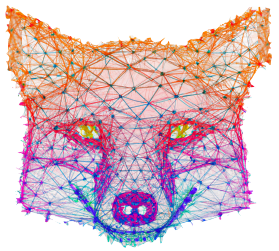
There's so much more!

Wouldn't it be awesome if my text editor could... Oh yeah, install.

Ask away, and discover more by exploring!

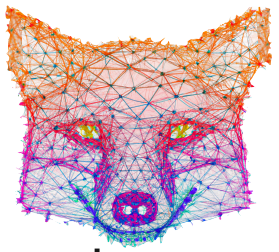
Sacha Chua

Environment variables



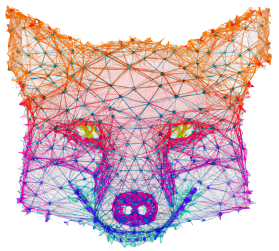
- In a terminal we frequently make use of **environment variables** to keep track of important paths, locations of libraries and files and executables, and to store all manner of information.
- Type **`env`** to **see all the environment variables** defined in your terminal session.
- Some important ones like `PATH` and `PYTHONPATH` are telling bash or python where to find executables or libraries.
- You can **create a new environment variable** like so **`export MYVAR="hello world"`**
- And you **can access its value** using **`${MYVAR}`**, eg **`echo "${MYVAR}"`**
- Environment variables become **very important for compilation** and maintaining a self-consistent set of packages. That's likely the main way you will use them this week.

Scripting



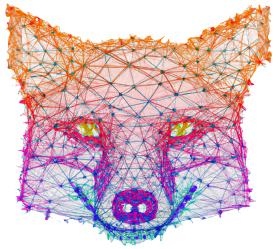
- Sometimes, we will want to **repeat a series of commands multiple times**, for example, when installing a software package.
- If we know a **sequence of commands** needs to be executed, and it's always the same, it makes sense to just **write them in a text file** and get the **shell to perform them one by one**.
  - Like the ***script* of a theatre production** - one line comes after the other!
- This is called **scripting**. We can make a new file "`myscrip.sh`", and tell it to do a certain number of things, like change directories, create new files, execute various commands.
  - Just write commands line by line as you would in the command line, then save the file.
  - After, you can do "`source myscript.sh`" to execute them sequentially.
- In principle you can also have if-else blocks, loops and other logical operations, but this bash is not a fun language to do these things in if you can avoid it. Logic should be reserved for your C++ or python files unless you know what you are doing.

Review of the terminal



- So after this part of the lecture you should know how to:
  - **Navigate a file structure**
  - **Create new directories** and files
  - **Copy or move files** and directories around
  - **Open a file in a text editor** (vim or emacs for example), make small changes, save and quit.
  - **Define or modify environment variables** and print their value
  - Write a simple **shell script**

- We will practice these skills in the hands-on session this afternoon!

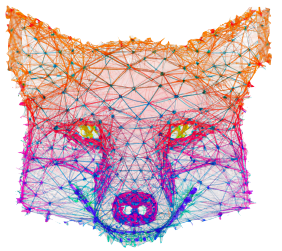


Python
And its various libraries...

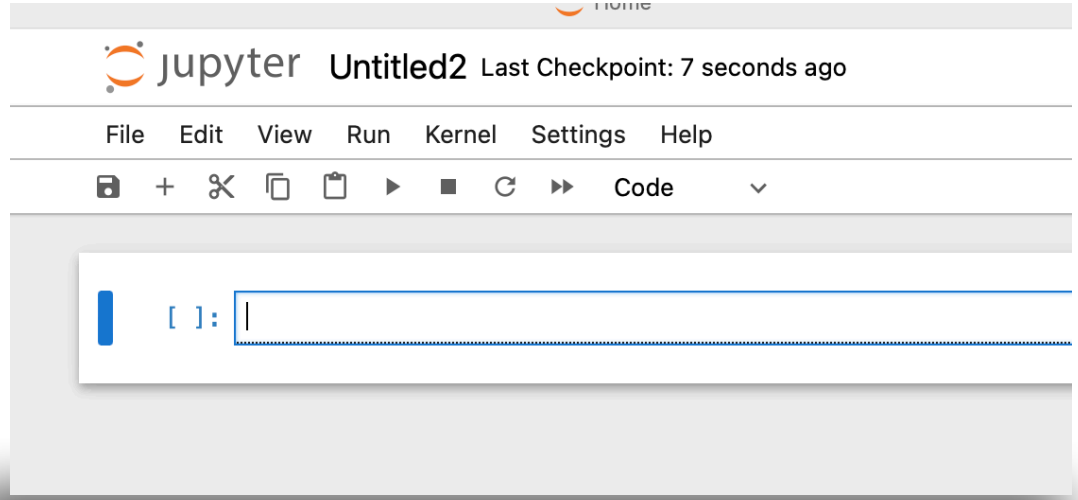
This part is based heavily on Romain Madar's "Introduction to Python for Data Analysis" course.
Find full materials here!
https://github.com/rmadar/lecture-python
And the Python Course PDF attached to indico.

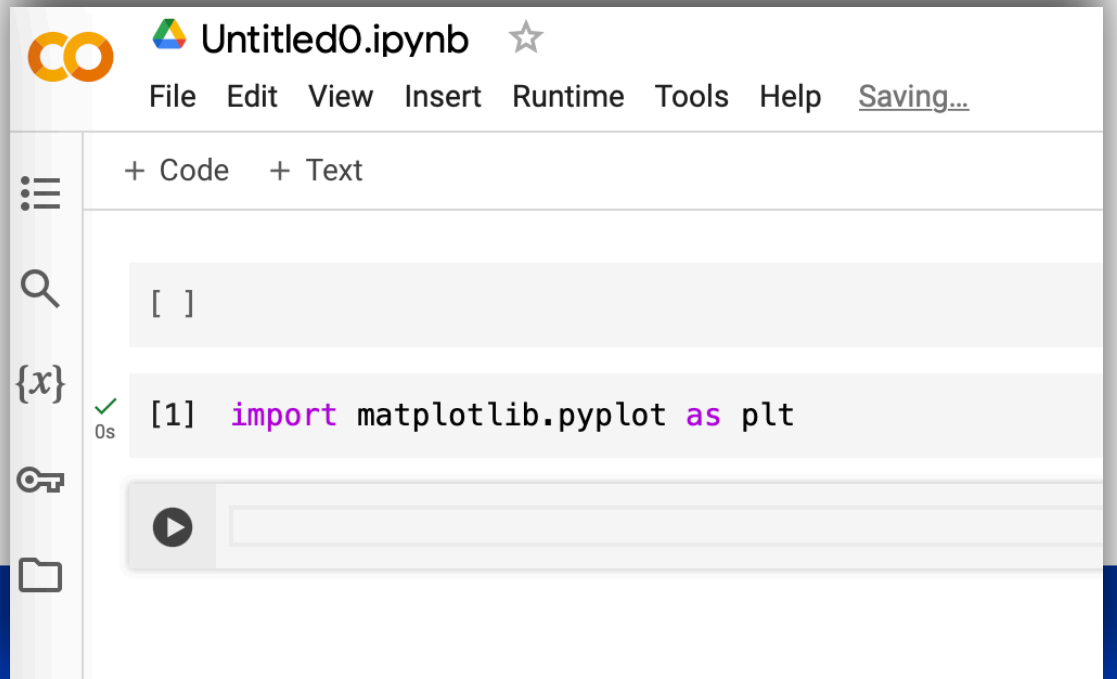
Python is a vast topic, I will only cover the basics to get everyone to a minimal level!

Jupyter notebooks

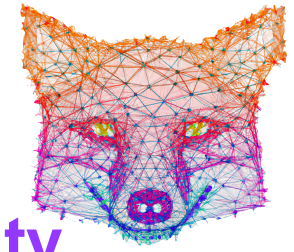


- **Jupyter notebooks** are used to create interactive notebook documents that can contain live code, equations, visualizations, media and other computational outputs.
- The simplest way to **install Jupyter** is on the command line via
  `pip install jupyter`
  https://jupyter.org/install
- You can also do it via **Google Colab**
  https://colab.google/
  that is how David's ML tutorials will be run
  - **Please create a Google account if you don't have one already!**





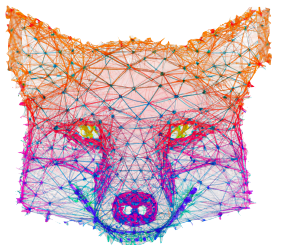
What is python ?



- Python is a very **popular coding language**, which has **many high quality libraries** for data analysis, plotting.
  - That's why its **used heavily in data science**, machine learning and many other domains
- Unlike C++, Python is not a compiled language.
  That means it's **easy to write but much slower to execute**.
  - Although some libraries like numpy use c++ under the hood so can be rather fast!
- Python supports classes (you can do object-oriented programming)
- One of the most important things to note: **logic in python is controlled using indentation instead of brackets (like in C++)**

Basic types and operations - numbers

Section 1.2 of the PDF



- There are three type of numbers: **int** (integer), **float** (floating point = decimal number) and **complex**.
- The **usual operations** (+, -, *, /) are available.
- In addition, there is also:
  - a**b (which means a^b),
  - a // b (divisor in integer division),
  - a % b (remainder in integer division),
- We also have **Booleans** (True/False)

```
# Basic numbers and operations
a = 2
b = 3.14
print(a+b)
print(a**b)
```

```
5.140000000000001
8.815240927012887
```

```
# Complex numbers and power
a = 1j
a**2
```

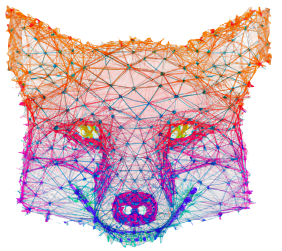
```
(-1+0j)
```

```
# Integer division example (// and % operators)
a , b = 10, 4
divisor, rest = a//b, a%b
print('{} = {}x{} + {}'.format(a, divisor, b, rest))
```

```
10 = 2x4 + 2
```

Basic types and operations - strings

Section 1.2 of the PDF



- **Strings** allow to manipulate words, sentence or even text with specific methods.
- String are also python lists (see next section) and list methods work as well
- The common and useful string manipulations can be counting the number of letters with `len(word)`
  or splitting a collection of words using `sentence.split(' ')`

```
w1 = 'hello'
print(w1, len(w1), w1[3])
```

```
hello 5 l
```

```
# Summing two strings is possible (all other operators dont work)
blank, w2 = ' ', 'world'
sentence = w1 + blank + w2
print(sentence)
```

```
hello world
```

```
# Multiplying a string by an integer is also possible
repetition = sentence*3
print(repetition)
```

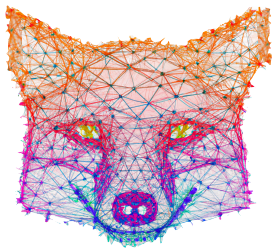
```
hello worldhello worldhello world
```

```
# Get a list of words from a sentence (cf. below for list objects)
s = 'It is rainy today'
list_words = s.split(' ')
print(list_words)
```

```
['It', 'is', 'rainy', 'today']
```

Object collections

Section 1.3 of the PDF

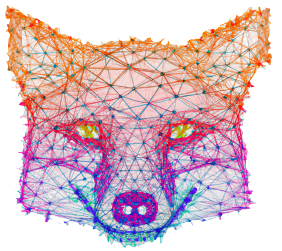


- There are four types of collection, which share several methods but differ from various aspects:
  - **Lists**, **dictionaries**, **sets** and **tuples**
- The most commonly used are the **lists** and **dictionary**.
- The specificy of the set is that it is unordered, while the specificity of the tuple is that it cannot be modified. We won't discuss them much more than that.

- number of items: `len(x)`
- loop over items with `for element in x:`
- check if a item is in the list: `element in x`

Object collections - lists

Section 1.3 of the PDF



- **Lists** are a *list of objects* with possibly different types.
- One can **search**, **loop**, **count** with lists. One can also add two lists or multiply a list by an integer, which makes a *concatenation* or a *duplication* (unlike for numpy arrays which we will see later).
- Lists can also be **indexed**. One can access the i^{th} element with my_list[i] or get a sub-list my_list[i:j]

```
# Defining a list and access basic information
my_list1 = [1, 3, 4, 'banana']
print('Second element is {}'.format(my_list1[1]))
print('Number of elements: {}'.format(len(my_list1)))
print('Is \'banana\' in the list? {}'.format('banana' in my_list1))
```

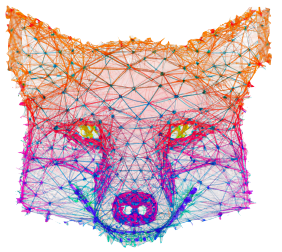
```
Second element is 3
Number of elements: 4
Is 'banana' in the list? True
```

```
# Sum of two lists
my_list2 = ['string', 1+3j, [100, 1000]]
my_list = my_list1 + my_list2
print(my_list)
```

```
[1, 3, 4, 'banana', 'string', (1+3j), [100, 1000]]
```

Object collections - dictionaries

Section 1.3 of the PDF



- **Dictionaries** are arrays of **(key, value)** pairs. Each value can be accessed via a unique key.
- The key must be a non-modifiable object, in practice string or integer.
- One can easily **loop**, **search**, **modify** a given key value, or even **add a new key** quite easily.

```
# dictionnary
person = {'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M'}
print(person)
```

```
{'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M'}
```

```
# Accessing value using the key
template = '{} ({}) is {} years old and is {} cm'
print(template.format(person['name'], person['gender'], person['age'], person['size']))
```

```
Charles (M) is 78 years old and is 173 cm
```

```
# Adding a key and its value
person['eyes'] = 'blue'
print(person)
```

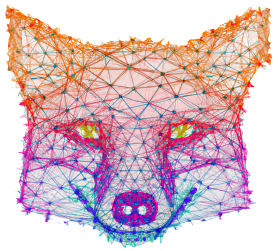
```
{'name': 'Charles', 'age': 78, 'size': 173, 'gender': 'M', 'eyes': 'blue'}
```

```
# Test if a key is present
print('name' in person)
print('brand' in person)
```

```
True
False
```

Looping in python

Section 1.4 of the PDF



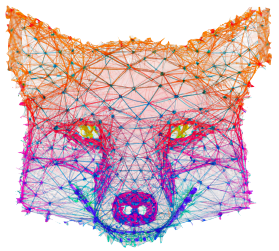
- **Loops** are at the core of programming.
- There are two way of repeating a instruction several times: the **for loop** and the **while loop**.
- Several instructions are common to both loops, such as
  - **continue** (skip instruction after and switch to the next element) or
  - **break** (stop the loop)

```python
# Define a dictionnary
students_marks = {
    'Jean': 12,
    'Chloe': 17,
    'Olivier': 8,
    'Helene': 10
    }

# Print the key, values items
for name, mark in students_marks.items():
    print(name, mark)
```

```
Jean 12
Chloe 17
Olivier 8
Helene 10
```

Looping in python

Section 1.4 of the PDF



- **Loops** are at the core of programming.
- There are two way of repeating a instruction several times: the **for loop** and the **while loop**.
- Several instructions are common to both loops, such as
  - **continue** (skip instruction after and switch to the next element) or
  - **break** (stop the loop)

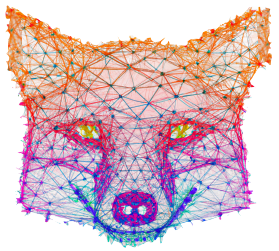
```python
# Compute sum(i^2) for i from 0 to 9
x = 0
for i in range(0, 10):
    x += i**2
print(x)
```

```python
# Loop over fruit via a set and print only ones with a 'p'
for fruit in s:
    if 'p' in fruit:
        print(fruit)
```

```
pineapple
apple
pear
prune
```

Looping in python

Section 1.4 of the PDF

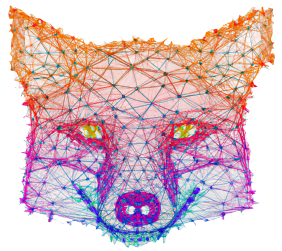


- **Loops** are at the core of programming.
- There are two way of repeating a instruction several times: the **for loop** and the **while loop**.
- Several instructions are common to both loops, such as
  - **continue** (skip instruction after and switch to the next element) or
  - **break** (stop the loop)

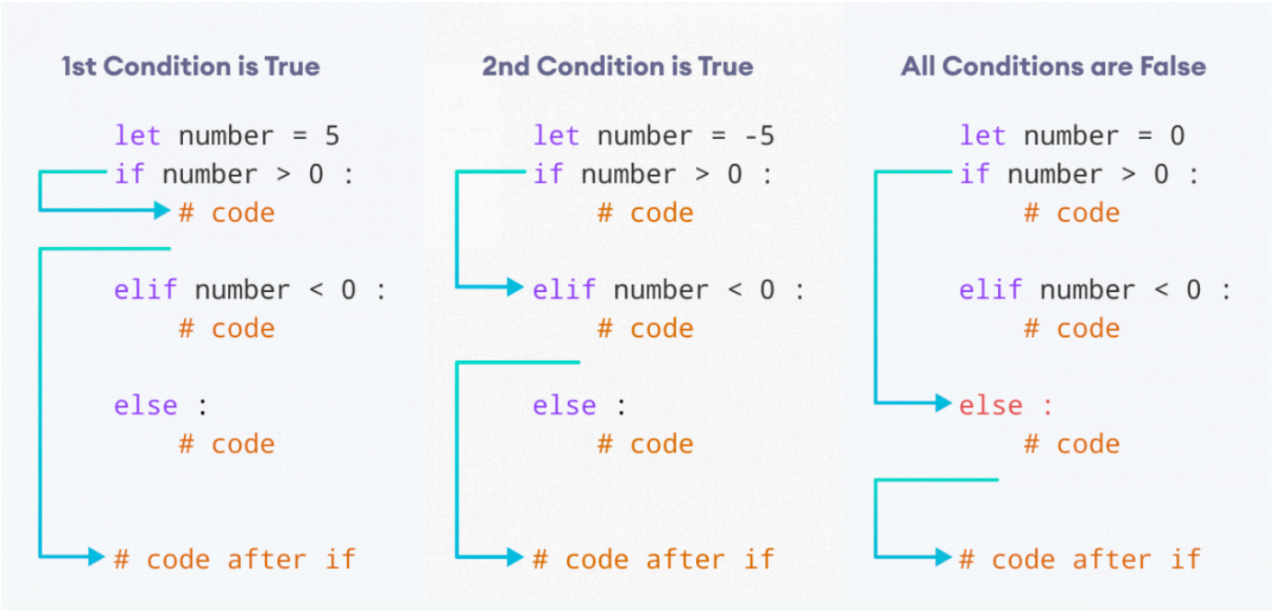
```
while len(my_list)>0:
    my_list.pop()
    print(my_list)
```

```
['orange', 'pineapple', 'apple', 'pear', 'prune']
['orange', 'pineapple', 'apple', 'pear']
['orange', 'pineapple', 'apple']
['orange', 'pineapple']
['orange']
[]
```

If-else blocks



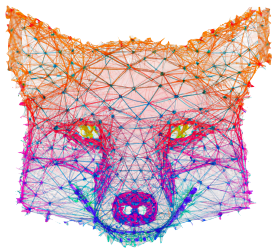
- **If-else** blocks allow you to control the logic flow of your code
- They work as you would expect, but beware of:
  - **elif**: if you want to add a third, fourth, etc option use elif (only one option in the block ever gets executed)
  - **Indentation**: make sure your logic does what you think it does by keeping related if/else keywords on the same indentation level!




```
# outer if statement
if condition1:
    # statement(s)

    # inner if statement
    if condition2:
        # statement(s)
```

List comprehension

Section 1.5 of the PDF



- **List comprehension** is the action of building a collection with one line of code. The comprehension syntax works for all collections, with conditions, or even nested loops (loops of loops).

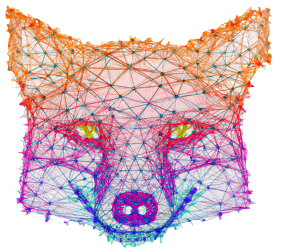
```python
# List
list_squares = [i**2 for i in range(1, 10)]
print(list_squares)

# Dictionnary
dict_squares = {i:i**2 for i in list_squares[0:5]}
print(dict_squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
{1: 1, 4: 16, 9: 81, 16: 256, 25: 625}
```

```python
# Comprehension list with a condition (e.g. keep only even numbers)
list_even = [i for i in range(0, 10) if i%2==0]
print(list_even)
```

```
[0, 2, 4, 6, 8]
```

Functions - *Section 1.6 of the PDF*



- **Functions** are defined as a set of instruction encapsulated into one object.
- This is particularly convenient when one has to the **same list of instructions several times**. **If you copy-paste the same piece of code more than two times, then make a function.**
- A function takes some **arguments**, perform some instructions and returns a **result**.
- The type of the argument is not fixed so the same instruction will be interpreted differently depending on the type. This is very different to C++ behaviour!

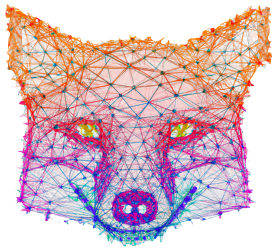
```
# Definition syntax
def function(argument):
    result = argument * 3
    return result

# Call syntax
function(2)
```

```
6
```

```
# Print the result for two types of arguements
print('function(10) = {}'.format(function(10)))
print('function(\'ouh\') = {}'.format(function('ouh')))
```

```
function(10) = 30
function('ouh') = ouhouhouh
```

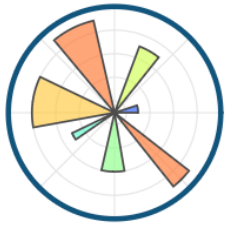
Libraries



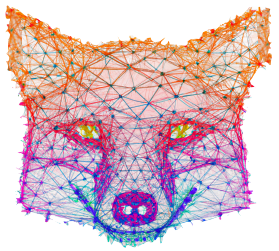
- One can write collections of functions and classes and maintain it. Some extremely powerful libraries are available this way:
  numpy, scipy, matplotlib, pandas...
- They can usually be installed easily on command line:
  `pip install matplotlib`
- And imported into your code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
```

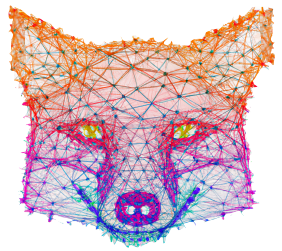
- I'll say a few words about the most important packages



Review of Python



- We reviewed:
  - Basic python **types** and **operations**
  - Common object collections: **lists and dictionaries**
  - **Looping** and **if-else statements**
  - Writing **Functions**
  - Review of **list comprehension**
  - **Packages/libraries** and **how to install them**
- But our python journey is not over! We will review now the core properties of the most important python library : **numpy**.
- And also common plotting and data manipulation libraries: **matplotib** and **pandas**

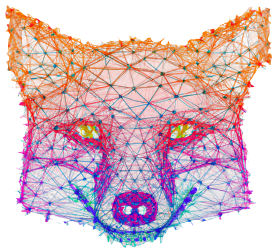


Numpy
And friends...

NumPy

This part is based heavily on Romain Madar's "Introduction to Python for Data Analysis" course.
Find full materials here!
https://github.com/rmadar/lecture-python
And the Python Course PDF attached to indico.

Numpy - *Chap 2 of the PDF*

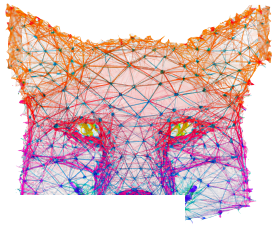


- Numpy (numerical python) is one of the most important packages in data science. Many other packages (scipy, pandas, scikitlearn) are based upon it.
- The core improvement wrt vanilla python is **numpy arrays.** It's like a python list, but with several critical improvements:
  - **Vectorised operations**
  - **Broadcasting**
  - **Fancy indexing and slicing.**
- These objects allow to **efficiently perform computations over large datasets** in a very concise way from the language point of view, and very fast from the processing time point of view.
- The price to pay is to **give up explicit *for* loops**. This lead to somehow a counter intuitive logic - at first.

```
import numpy as np
l1, l2 = [1, 2, 3], [3, 4, 5]
a1, a2 = np.array([1, 2, 3]), np.array([3, 4, 5])
```

Arrays can have arbitrary dimension,
But must all have the same data type.
Unlike lists which can contain any objects
Even of mixed types!

Vectorisation - *Chap 2.3.1 of the PDF*



- ***Vectorisation*** is a way to make computations on numpy array **without explicit loops**, which are very slow in python.
- In simple terms: you can do an operation on an array as if it was just a single item, and it will **implicitly do the operation on all items of the array.**
- The idea of vectorization is to compute a given operation *element-wise* while the operation is called on the array itself.

```
a = np.random.randint(low=1, high=100, size=100000)

def explicit_loop_for_inverse(array):
    res = []
    for a in array:
        res.append(1./a)
    return np.array(res)
```

```
# Using explicit loop
%timeit explicit_loop_for_inverse(a)
```

```
186 ms ± 13.8 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

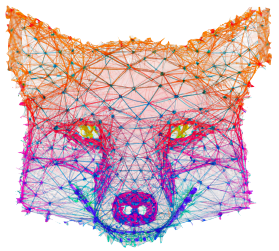
```
# Using list comprehension
%timeit [1./x for x in a]
```

```
150 ms ± 13.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
# Using vectorization
%timeit 1./a
```

```
106 µs ± 2.65 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Broadcasting - *Chap 2.3.2 of the PDF*



- ***Broadcasting*** is a way to compute operation between arrays of having different sizes in a implicit (and concise) manner.

```
# Translating 3 2D vectors by d0=(1,4)
points = np.random.normal(size=(3, 2))
d0 = np.array([1, 4])
print('points:\n {}\n'.format(points))
print('points+d0:\n {}'.format(points+d0))
```

```
points:
 [[ 0.24333406 -0.80020589]
 [ 1.06124037  0.11580338]
 [ 0.52607555 -1.94077365]]

points+d0:
 [[1.24333406 3.19979411]
 [2.06124037 4.11580338]
 [1.52607555 2.05922635]]
```

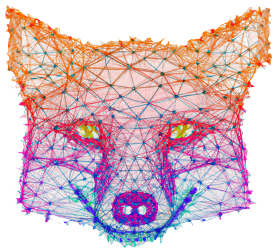
```
# operation between shape (3) and (1)
a = np.array([1, 2, 3])
b = np.array([5])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[6 7 8]
```

```
# operation between shape (3) and (1,2)
a = np.array([1, 2, 3])
b = np.array([
              [4],
              [5],
             ])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[[5 6 7]
 [6 7 8]]
```

Fancy indexing/slicing - *Chap 2.3.3 of the PDF*

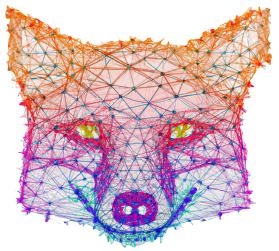


- ***Indexing*** in numpy takes on a whole new level compared to vanilla python
- You can still **access elements of an array using [i] or slices using [i:j]** as for lists
- But you can also **provide *arrays of indices*** saying which ones to pick, (which is not possible for lists)

```
a = np.random.randint(low=1, high=100, size=10)
print('a = {}'.format(a))
print('a[2] = {}'.format(a[2]))
print('a[-1] = {}'.format(a[-1]))
print('a[[1, 2, 5]] = {}'.format(a[[1, 2, 5]]))
```

```
a = [26 15 60 38 79 78 12 27 86 14]
a[2] = 60
a[-1] = 14
a[[1, 2, 5]] = [15 60 78]
```

Fancy indexing/slicing - *Chap 2.3.3 of the PDF*



- You can **access sub-arrays using the format [i:j:k]**, meaning take elements from i to j in steps of k. Negative k values go through the elements in reverse.
- This also works when you have **multi dimensional arrays**!

```
# 3D array
a = np.random.randint(low=0, high=100, size=(5, 2, 3))
print('a = \n{}'.format(a))
```

```
a =
[[[20  0  7]
  [93 24  4]]

 [[43 65 46]
  [97 59 57]]

 [[86  2 85]
  [31  0 57]]

 [[46  1 94]
  [98 65  6]]

 [[81 15  3]
  [88 47 90]]]
```

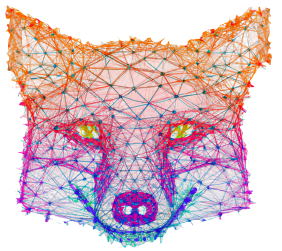
```
# Taking only the x,y values of the first vector for all observation:
print('a[:, 0, 0:2] =\n {}'.format(a[:, 0, 0:2]))
```

```
a[:, 0, 0:2] =
 [[20  0]
 [43 65]
 [86  2]
 [46  1]
 [81 15]]
```

```
# Reverse the order of the 2 vector for each observation:
print('a[:, ::-1, :] = \n{}'.format(a[:, ::-1, :]))
```

```
a[:, ::-1, :] =
[[[93 24  4]
  [20  0  7]]

 [[97 59 57]
  [43 65 46]]
```

Fancy indexing/slicing - *Chap 2.3.3 of the PDF*

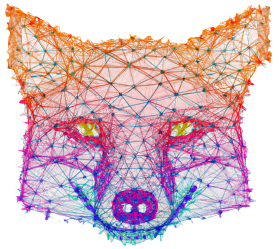


- Finally, in **numpy you can do "masking"**. If you provide a **list of True/False values** the same size and shape as your array, you will create a new list where only the elements with a "True" value are included.
- We can use this **to apply selections** on an array
- ~ can be used to negate a mask
- **Masking is a super-powerful tool in particle physics when applying selections!**

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
mask = a>0
print('a = \n{}'.format(a))
print('\nmask = \n {}'.format(mask))
```

```
a =
[[ 19 -52  77]
 [ 36 -68  88]
 [-61 -18 -71]
 [ 64 -46  58]
 [-83 -12  18]]

mask =
 [[ True False  True]
 [ True False  True]
 [False False False]
 [ True False  True]
 [False False  True]]

a[mask] =
 [19 77 36 88 64 58 18]

a[~mask] =
 [-52 -68 -61 -18 -71 -46 -83 -12]
```

Extra numpy tips - *Chap 2.4 of the PDF*



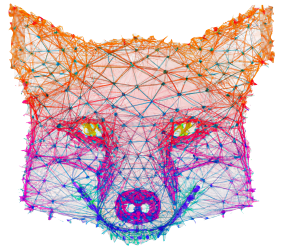
Dummy array initialization.

```
x = np.zeros(shape=(3, 2))                 # Only 0
x = np.ones(shape=(3, 2))                  # Only 1
x = np.full(shape=(3, 2), fill_value=10) # Only 10
x = np.eye(2)                              # Create identity matrix (only return 2D array)
```

Create sequence of numbers.

```
# Linear inteveral from 0 to 10
x = np.linspace(0, 10, 10) # 10 numbers between 0 and 1
x = np.arange(0, 10, 1.0)  # One number every 1.0 between 0 and 1
x = np.logspace(0, 10, 10) # 10 numbers between 10**0 and 10**10
```

Extra numpy tips - *Chap 2.4 of the PDF*



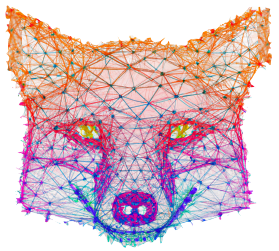
Shape-based manipulation of arrays.

```
a = np.arange(0, 18).reshape(3, 3, 2)
x = a.ravel()                       # Return a flat array
x = a.reshape(9, 2)                 # change the shape
x = a.T                             # transpose array: a.T[i, j, k] = a[k, j, i]
x = np.concatenate([a,a], axis=0) # concatenate arrays along a given axis: shape=(6, 3, 2)
x = np.stack([a, a], axis=0)        # group arrays along a given axis: shape=(2, 3, 3, 2)
```

Compare arrays.

```
# Making dummy arrays for comparisons
a = np.arange(-6, 6).reshape(3, 4)
b = np.abs(a)
c = np.append(b, [[1, 2, 3, 4]], axis=0)
```

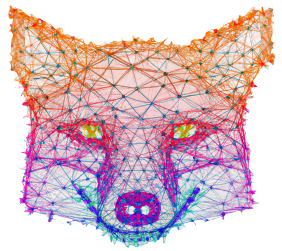
Plotting with matplotlib - *Chap 3.2 of the PDF*



- **Matplolib** is an extremely rich **library for data visualization** and there is no way to cover all its features here. The goal of this section is just to give short and practical examples to plot data.
- The main object of `matplotlib` is `matplotlib.pyplot` imported as `plt` here (and usually). The most common functions are then called on this objects, and **often takes numpy arrays in argument** (possibly with more than one dimension)

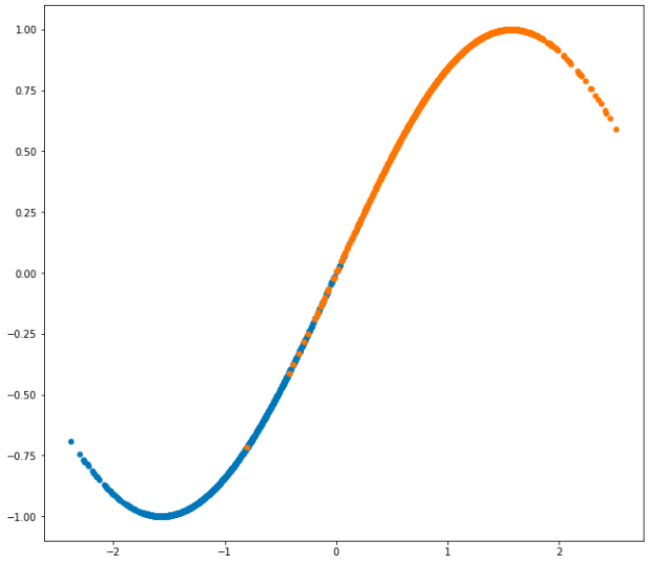
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

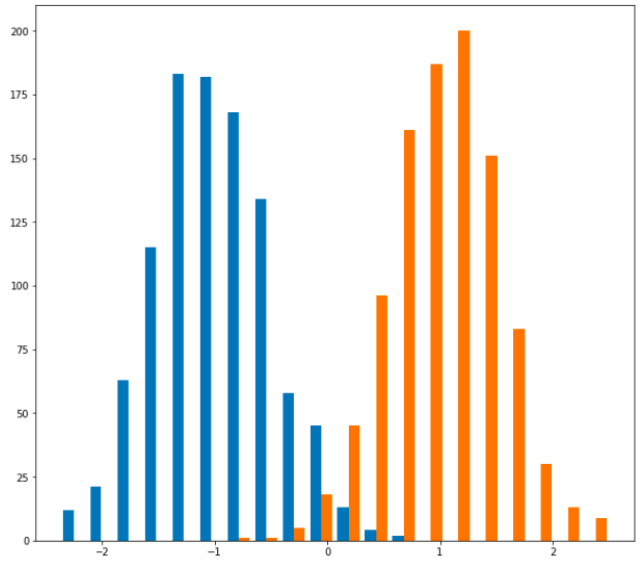
1D data plots - *Chap 3.2.1 of the PDF*



```
x = np.random.normal(loc=[-1, 1], scale=[0.5, 0.5], size=(1000,2))
y = np.sin(x)
```

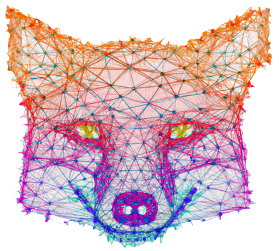
```
plt.figure(figsize=(24, 10))
plt.subplot(121) # 121 means 1 line, 2 column, 1st plot
plt.plot(x, y, marker='o', markersize=5, linewidth=0.0)
plt.subplot(122) # 122 means 1 line, 2 column, 2nd plot
plt.hist(x, bins=20);
```





- We can make use of **`plt.plot(...)`** or **`plt.hist(...)`** to plot one-dimensional data.

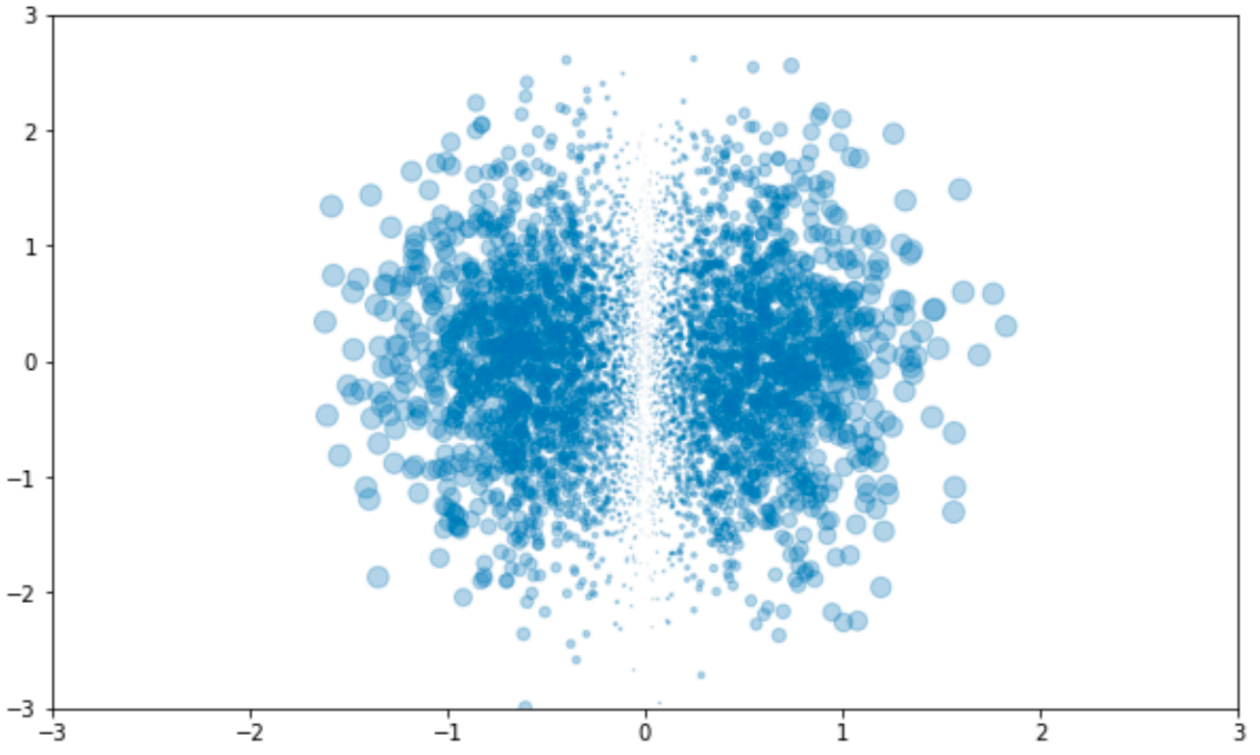
2D data plots - *Chap 3.2.2 of the PDF*



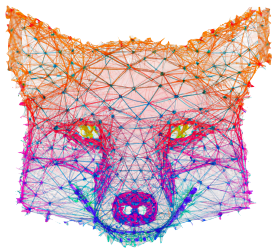
- Scatter is useful for 2D plotting
- For example we can use the size of markers as an extra dimension

```
points = np.random.normal(loc=[0, 0], scale=[0.5, 0.8], size=(5000,2))
x, y = points[:, 0], points[:, 1]
```

```
plt.figure(figsize=(10,6))
plt.scatter(x, y, s=100*(np.sin(x))**2, marker='o', alpha=0.3)
plt.xlim(-3, 3)
plt.ylim(-3, 3);
```



3D data plots - *Chap 3.2.3 of the PDF*

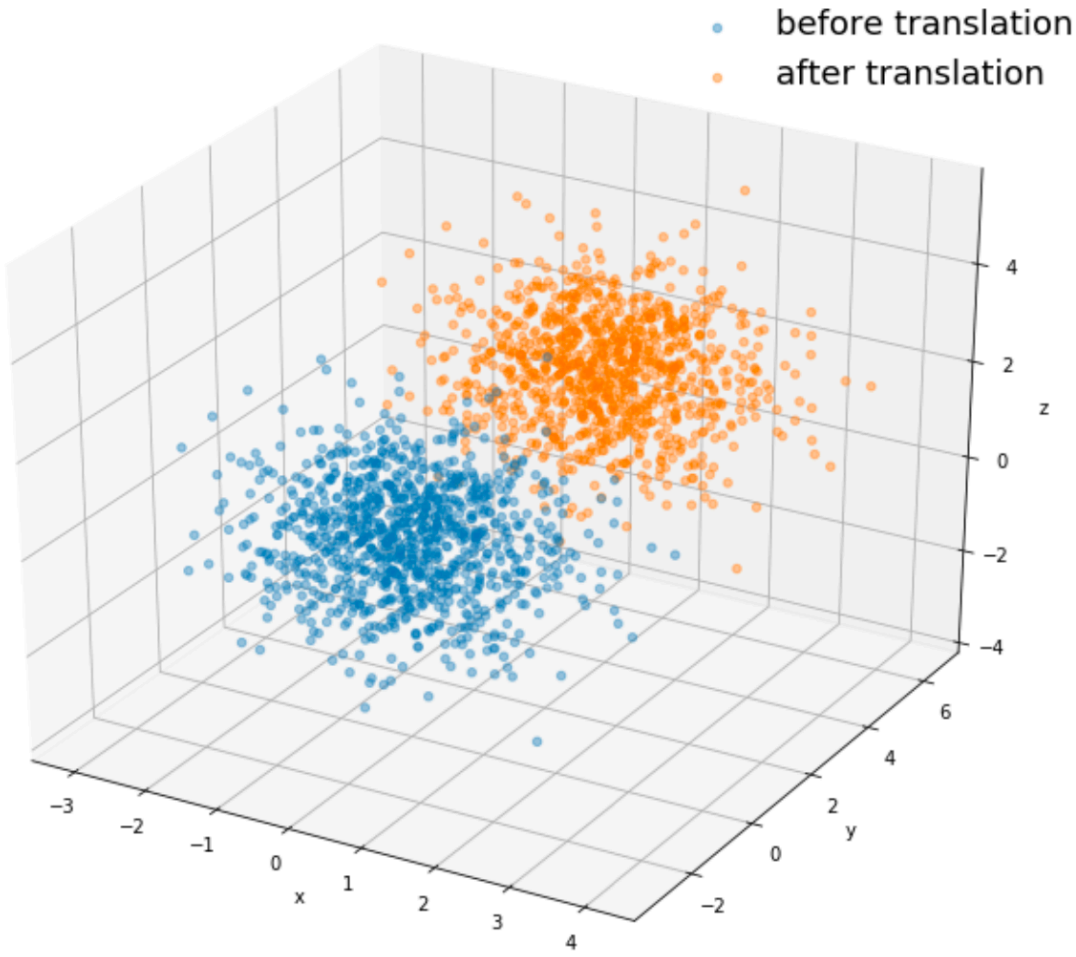


- We can also make 3-D representations

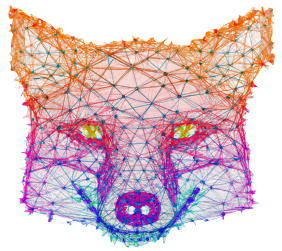
```
data = np.random.normal(size=(1000, 3))
r0 = np.array([1, 4, 2])
data_trans = data + r0
```

```
xi, yi, zi = data[:,0], data[:,1], data[:,2]
xf, yf, zf = data_trans[:,0], data_trans[:,1], data_trans[:,2]
```

```
from mpl_toolkits import mplot3d
plt.figure(figsize=(12,10))
ax = plt.axes(projection='3d')
ax.scatter3D(xi, yi, zi, alpha=0.4, label='before translation')
ax.scatter3D(xf, yf, zf, alpha=0.4, label='after translation')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend(frameon=False, fontsize=18);
```



3D data plots - *Chap 3.2.3 of the PDF*



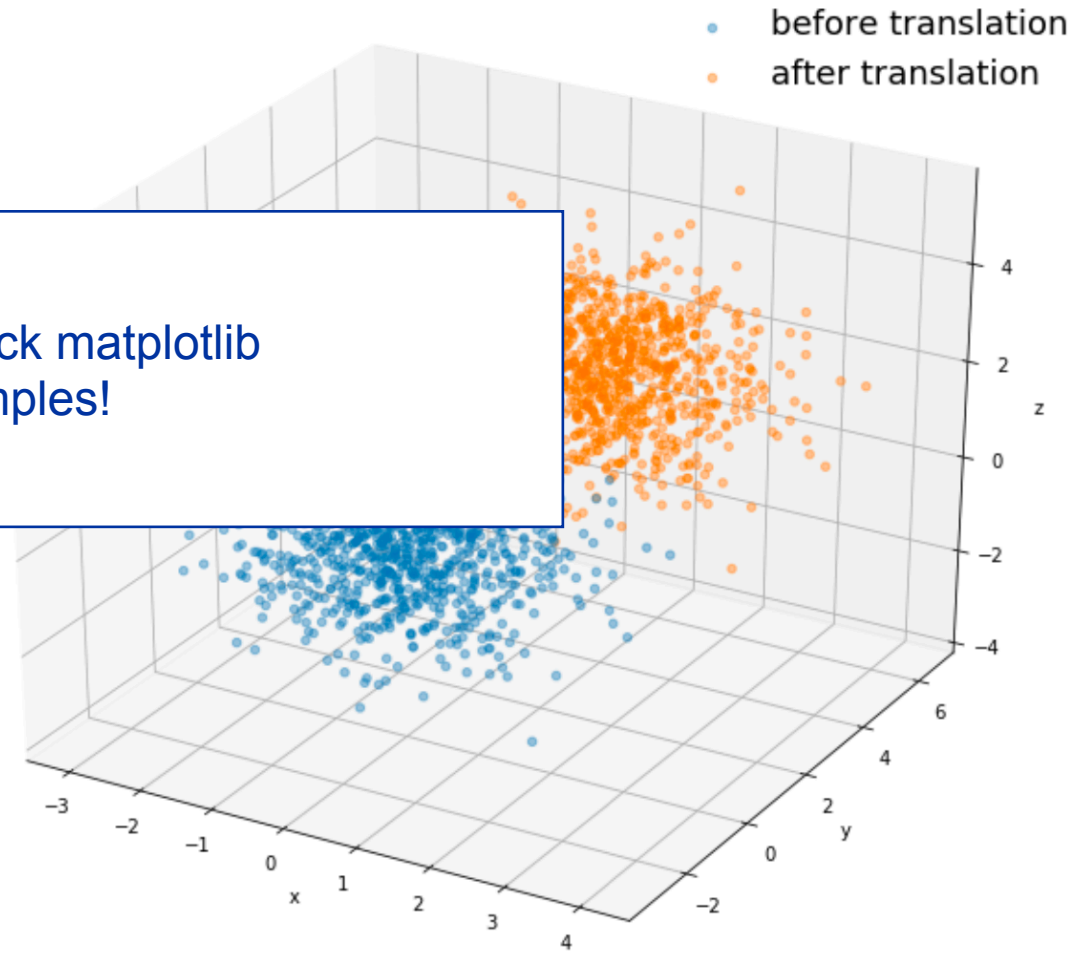
- We can also make 3-D representations

```
data = np.random.normal(size=(1000, 3))
r0 = np.array([1, 4, 2])
data_trans = data + r0
```

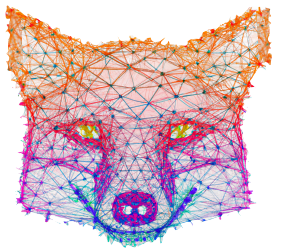
```
xi, yi, zi = data[:
xf, yf, zf = data_t
```

```
from mpl_toolkits i
plt.figure(figsize=
ax = plt.axes(projection='3d')
ax.scatter3D(xi, yi, zi, alpha=0.4, label='before translation')
ax.scatter3D(xf, yf, zf, alpha=0.4, label='after translation')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend(frameon=False, fontsize=18);
```

Many other kinds of plots are possible, check matplotlib documentation for comprehensive list examples!



Pandas dataframes- *Chap 3.3 of the PDF*



- **Pandas** is a package which is extremely useful to **manipulate datasets.**
- It can very quickly read in a dataset (for example as CSV file) into a so-called dataframe.
- **Dataframe objects are essentially dictionaries of numpy arrays**. You can access properties (= columns of the CSV) by name, add new columns, and manipulate the contents rapidly using dumpy-like vectorization, broadcasting and fancy indexing.

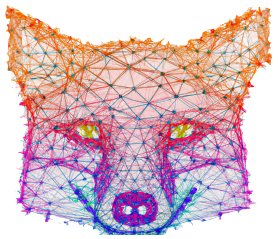
```
import pandas as pd
df = pd.read_csv('../data/WaveData.csv')
print(df.head())
```

	Date/Time	Hs	Hmax	Tz	Tp	Peak Direction	SST
0	01/01/2017 00:00	-99.900	-99.90	-99.900	-99.900	-99.9	-99.90
1	01/01/2017 00:30	0.875	1.39	4.421	4.506	-99.9	-99.90
2	01/01/2017 01:00	0.763	1.15	4.520	5.513	49.0	25.65
3	01/01/2017 01:30	0.770	1.41	4.582	5.647	75.0	25.50
4	01/01/2017 02:00	0.747	1.16	4.515	5.083	91.0	25.45

```
# Simply take value above -99
print(df[df>-99].head())
```

	date	height	heightMax	period	energy	direction	temperature
0	01/01/2017 00:00	NaN	NaN	NaN	NaN	NaN	NaN
1	01/01/2017 00:30	0.875	1.39	4.421	4.506	NaN	NaN
2	01/01/2017 01:00	0.763	1.15	4.520	5.513	49.0	25.65
3	01/01/2017 01:30	0.770	1.41	4.582	5.647	75.0	25.50
4	01/01/2017 02:00	0.747	1.16	4.515	5.083	91.0	25.45

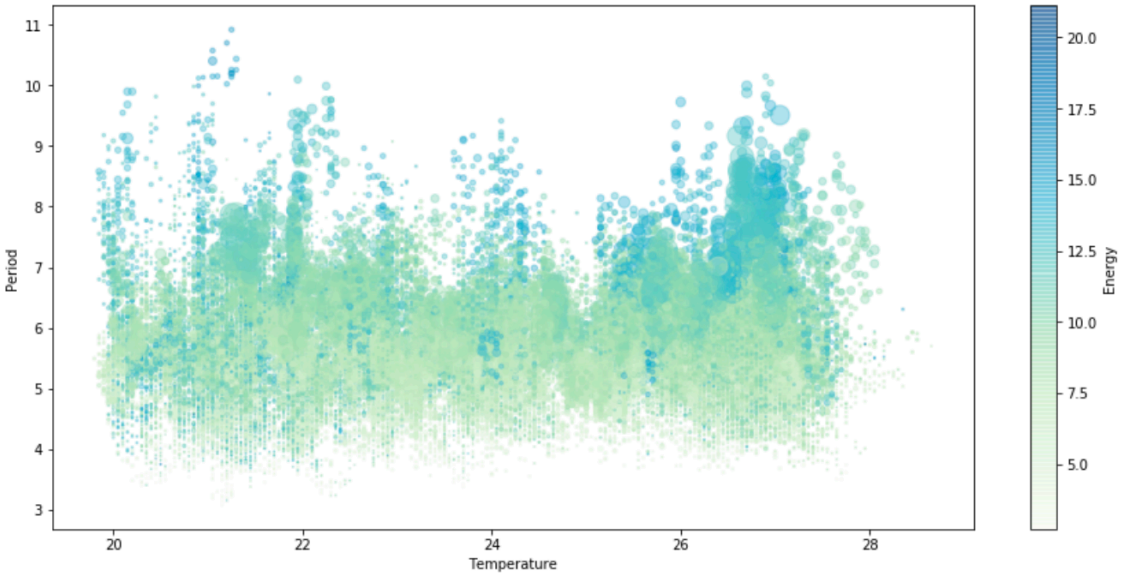
Pandas dataframes- *Chap 3.3 of the PDF*



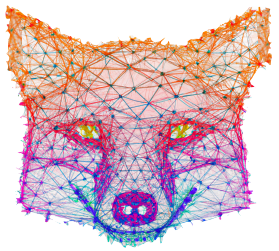
- **Pandas** is a package which is extremely useful to **manipulate datasets.**
- It can very quickly read in a dataset (for example as CSV file) into a so-called dataframe.
- **Dataframe objects are essentially dictionaries of numpy arrays**. You can access properties (= columns of the CSV) by name, add new columns, and manipulate the contents rapidly using dumpy-like vectorization, broadcasting and fancy indexing.

```
# Get numpy array for further manipulations
T = df['temperature'].values
P = df['period'].values
H = df['heightMax'].values
E = df['energy'].values
```

```
# Plot temperature vs period vs max_height vs energy
plt.figure(figsize=(15, 7))
plt.scatter(T, P , s=H**3, c=E, cmap='GnBu', alpha=0.4)
plt.colorbar(label='Energy')
plt.xlabel('Temperature')
plt.ylabel('Period');
```

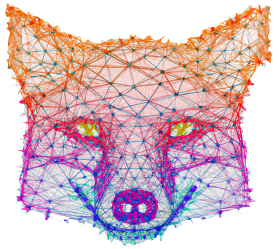


Review of Numpy and friends



- **Numpy** introduces a new collection called a **numpy array**. It's like a list but:
  - Can be of **arbitrary dimension**
  - Does **vectorised** operations
  - Supports **broadcasting**
  - Support **fancy indexing** and **masking**
- Many excellent packages are based upon numpy including:
  - **Matplotlib** for plotting (in particular plot(), hist(), scatter() types)
  - **Pandas dataframes**, which are basically dictionaries of numpy arrays for efficient dataset book-keeping and manipulation.

Conclusion



- We have covered a lot of ground this morning and I don't expect you to remember everything!
- After lunch, we will attempt some practical examples. That's the best way to learn !
- Thanks for your attention!
- Speak to me in the break if you have questions