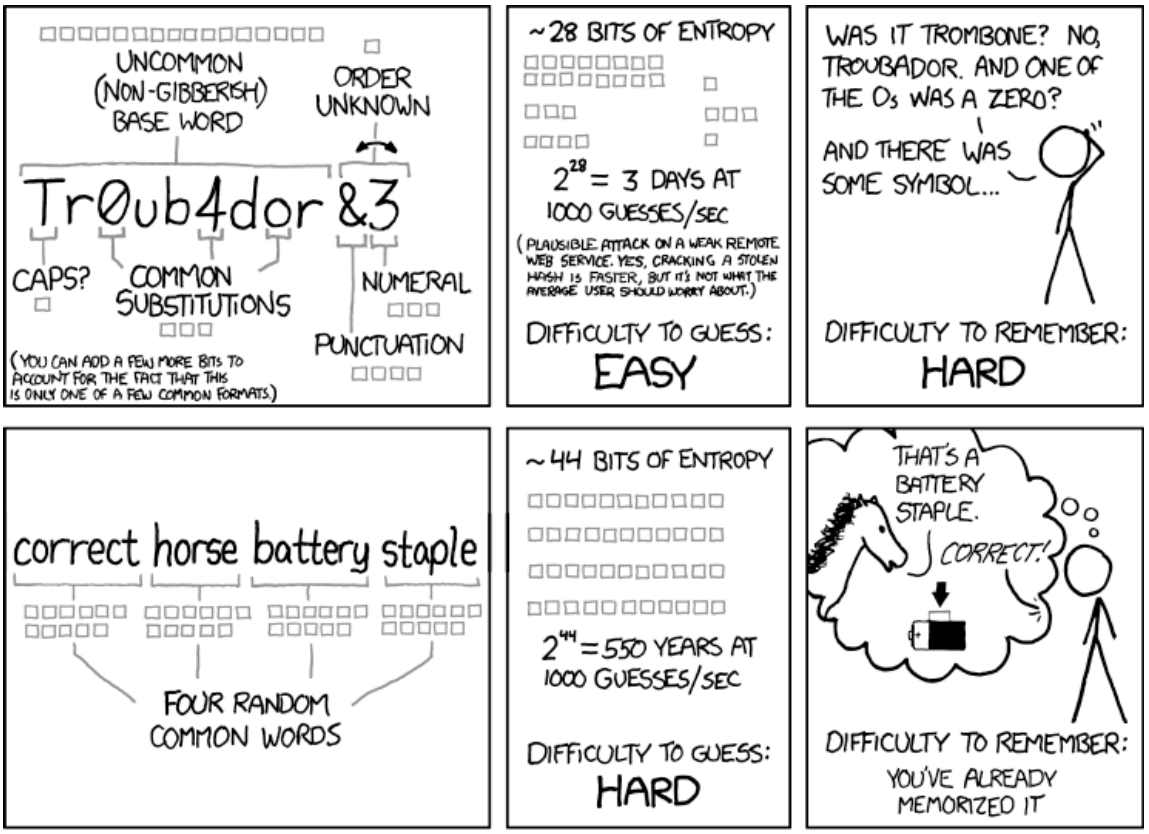
PhD Hacks Session -- KSETA Fellow Workshop 2023

Credits: All the pictures are taken from https://xkcd.com (https://xkcd.com)

Passwords!

In todays times, it is very important to use strong passwords. With fast computers, it gets more and more easy to crack easy passwords. Therefore, it is important to keep some points in mind when creating a password:

- avoid personal info! (birthday or so)
- a long password is way more save than a complicate one that is hard to remember
- at least 12 characters are recommended, more are better
- don't use words from dictionaries as they can easily be tested by algorithms
- if you use words, chain them to longer sentences. This makes the guessing harder.
- use unique passwords that usually don't go together in a "normal" sentence
- general recommendations from google: https://support.google.com/accounts/answer/32040?hl=en#zippy= (https://support.google.com/accounts/answer/32040?hl=en#zippy=)



And even better is to use a password manager to keep your passwords save, for example: https://wiki.ubuntuusers.de/Passwortmanager/ (https://wiki.ubuntuusers.de/Passwortmanager/)

Which OS should I use?

This is not easy to answer and depends on your projects and the hardware environment at your institutes. In general, I would recommend **Linux**, since it is free and many experiment software and general analysis tools are available.

- Ubuntu LTS (Linux): Free OS with a very good support for most hardware and beginner friendly. Can be easily installed (also alongside a windows installation). A lot of help can be found on the internet. Linux offers many helpful tools. Today, we will mainly focus on tools that are available as free software for linux based systems.
- MAC OS: Also a unix based OS, but the opposite of free :D . It can be recommended in general, since many of the tools etc can be installed with homebrew.
- Windows: Working on plane windows is not recommended from my side, since many many helpful things are not available for windows. In the end, each and everyone has to decide for him/herself. If your experiments software etc is only available for windows, I would recommend to keep that. But many data analysis tools etc are not available and therefore, you should decide carefully. As a tradeoff, you can also try `Windows Subsystem for Linux (WSL)`

If you want to try Linux, I am recommending `Ubuntu (LTS)` since it is the most beginner friendly, optionally with Gnome or KDE (more windows-like).
If you are already more experienced with unix-based systems, my recommendation is Fedora.

How to Use the Command Line

Bash

Bash is the standard shell (terminal) in most unix systems. A shortcut to open it on Ubuntu is `Ctrl+Alt+t` .

> Note: A great alternative is zsh, which is also the default shell for MacOS. However, I want to keep this tutorial as basic as possible, therefore, we will focus on bash.

The first important command we should get familiar with is `$ man <COMMAND>` .
It shows the ManPage for the chosen command (see: https://wiki.ubuntuusers.de/man/)\ (https://wiki.ubuntuusers.de/man/)%5C)

Basic commands:

- `ls` : list content of current directory
- `cd <directory>` : change directory to <directory>
- `rm <file>` : remove file; for directories, `-r` is necessary
- `cp <file> <destination>` : copy file to destination directory (the name can be changed); directories: again `-r`
- `mv <file> <destination>` : move file to destination; with this command, you also can rename files and directories and you can overwrite everything, so be careful!
- `mkdir` : create a directory (with `-p` , you can create a complete path)
- `passwd` : used to change your password
- `env` : prints all environment variables
- `echo` : with echo, you can print the content of a variable (e.g. `echo $PATH`)

> Excercies: list the current directory in long list format with human-readable sizes, sorted from oldest to newest entry without showing "." and "..".

To see hidden files, we need to add options to the `ls` command.
You can combine several options!

- `-a` : do not ignore hidden files
- `-d` : list the directory itself, not the content
- `-h` : show sizes as human-readable
- `--si` : likewise, but use powers of 1000 not 1024
- `-I=PATTERN` : igonre entries mathing shell pattern
- `-l` : long list format
- `-r` : reverse order
- `-R` : recursive; lists subdirectories
- `-S` : sort by file size

What also can be very useful are wildcards (*, [ ], ?):
For example, you can list all .txt files in a directory by typing:
`$ ls -la *.txt \`

- `*` : matches everything
- `?` : matches one character
- `[ ]` : matches a range, e.g.m `[0-9]`

If your directory has too many entries, it can be useful to "pipe" the content into `less` :
`$ ls <directory> | less`
(exit with `q`)
Here, you can also search by just entering `/<name> + Return` .
If you have multiple matches, you can jump to the next one with `n` .

Useful shortcuts:

- `Ctrl+a` : jump to position 1
- `Ctrl+e` : jump to end of line
- `Ctrl+w` : delete last word (until next blank space)
- `Ctrl+l` : clear screen
- `Ctrl+c` : stop running command/process
- `Ctrl+d` : close terminal
- `Ctrl+r` : search in history
- `Ctrl+y` : retrieves deleted or cut item.

.bashrc

The *.bashrc* is a configuration file that is queried when starting a new shell. Here, you can activate colors, export environment variables, define aliases and functions and so on.

The default bashrc in Ubuntu can befound at `/etc/skel/.bashrc` .
To use it, just copy it to your home directory and restart your terminal: `cp /etc/skel/.bashrc ~`

Some useful things:

- define aliases: `alias start_jupyter='source myvenv; jupyter notebook --no-browser --port=8080'`
- export variables: `export MyVar="importantThings"`
- define bash-functions: see below

But be careful not to overwrite an already existing command! To check, if a command exists, you can type in `$ which <command>` .

> Note: If the .bashrc is not loaded, try to create the file ~/.profile with the following content and restart your shell:

```
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi
```

> Important: Dont mess with your PATH variable!

find and grep

With `find` and `grep` we can search for things on our linux system.

find

`find` is mainly for searching files on your system. It is recommended to restrict the search to certain directories and avoiding searching on `/`, for example, as it can take for ever :D

To search for a file or directory within your current directory, just type:
`$ find <name>`
If a directory is searched and found, the content of the directory is printed.
Here, we also can use wildcards again:
`$ find a*` lists all files and all directories (+ their content below) starting with "a"

With `-name`, we can search for an exact filename, or all pdfs and such things:
`$ find -name "*.pdf"`

There are also additional options, for example `-delete` to delete all files that were found.

grep

With `grep` we can search for a pattern in files.
For example, while debugging, this can be very helpful.
`grep` has lots of cool features, but for most use-cases, the following command shoud be sufficient:

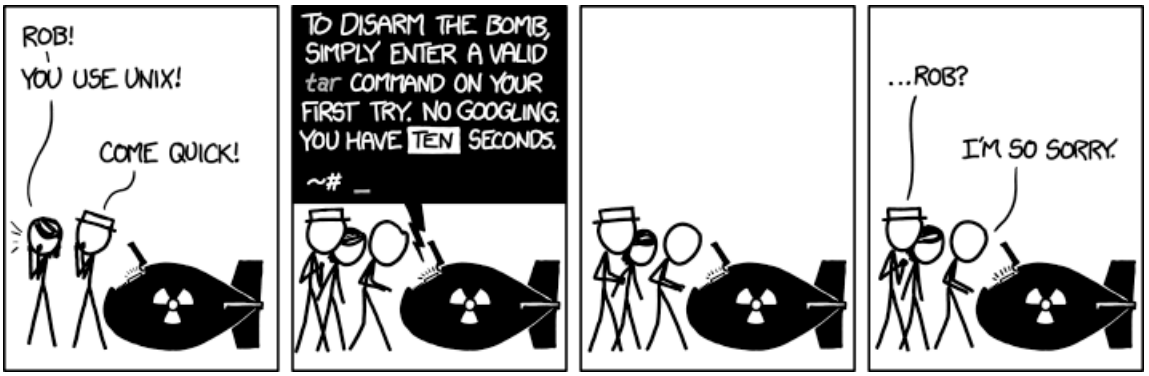
```
$ grep -nir "<pattern>"
```

With this, we are recursively searching for all occurences of the pattern in all files within the current directory and directories below (`-r`). Additionally, we are not case-sensitive (`-i`) and we show the line number of the occurance (`-n`).

If you only want to search one file, you have to specify the file:
`$ grep -ni "pattern" filename`

> Note: We can also combine this again with other commands by piping. For example $ ls -la | grep 'name' .

tar



`tar` is an archive tool that lets you combine several files to one archive (like zip/winrar /7zip/etc).
It contains access rigths. This means, if the files are only accessible to you, this is still the case after taring and untaring (on the contrary to `zip`).
This is very useful to, for example, save and backup your finished work in one file or to combine several small logfiles within one archive (#ceph ;-))

To pack an archive:

```
$ tar -cvf <archive_name> <directory or files>
```

> Note: The order of the input flags is essential! "-f" has to be always at the end since everything after it is interpreted as output and input strings!

Here, we also can use wildcards again:

```
$ tar -cvf plots.tar *.pdf
```

This tars all pdfs in the current directory to "plots.tar".

> Note: The files are not removed by this. If you want to delete them, you can do this when the tar ball is created.

We can also compress the archives by adding the option `-z`:

```
$ tar -czvf plots.tar.gz *.pdf
```

To have a look into an archive, we can use:

```
$ tar -tvf plots.tar.gz
```

To extract a tar ball, use (-xzvf for compressed archives):

```
$ tar -xvf plots.tar
```

This just extracts the content into the current directory (Attention: see warning below!)
You can also just extract a certain file from the archive by specifying it:

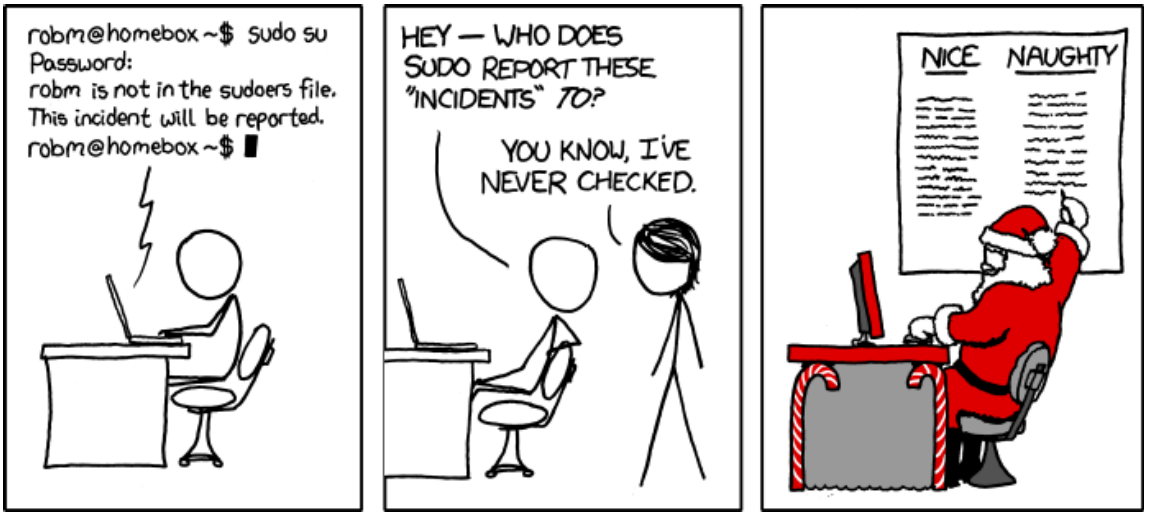
```
$ tar -xvf plots.tar <filename_in_archive>
```

To extract it to a certain folder, the folder has to exist. Then we can use:
`$ tar -xvf plots.tar -C /path/to/emptv/folder`

> Warning: tar overwrites on extract! So be careful...

How to Install Software on Linux

`apt` is a so-called package manager. Via apt, you can install all available software from the official ubuntu repositories (https://wiki.ubuntuusers.de/APT/ (https://wiki.ubuntuusers.de/APT/)). First, we need to update the package resources with (`sudo` is necessary to use sufficient privileges to install something):




```
$ sudo apt update
```

Then, we can install packages via:

```
$ sudo apt install <package>
```

> Note: To actually *update* an installed package, you have to use "$ sudo apt upgrade"

If you have a clean ubuntu installation, your local account should always have sudo rights. If a root account is configured, you need to login as `root` and add your local account to the sudoes group with:
`$ usermod -aG sudo username`
To check, who has sudo rights, you can check: `$ groups username`
or
`$ id username`
or have a look at `/etc/sudoers` .

SSH

SSH (secure shell) is a very powerful tool that is used to connect to servers/PCs via the internet. It is the de facto standard for secure internet connections.

If not installed, we first need to install it on our machine by typing:
`$ sudo apt install openssh-client`

If you want to connect to a computer, it needs also the `openssh-server` installed:
`$ sudo apt install openssh-server`
And eventually starting and enabeling it with:
`$ systemctl enable ssh --now`

If we now want to connect to a server, we simply need to type:
`$ ssh username@server`
And enter our credentials.
If you want to connect with a graphical interface, you can use the `-Y` option.\

If you encounter any problems, sometimes the verbose option is useful: `$ ssh user@server -vvv`
There are many many more CLI options, for example `-6` to enforce IPv6.
More info can be found here: https://linux.die.net/man/1/ssh (https://linux.die.net/man/1/ssh)

> Note: The standard ssh port is 22. Sometimes, servers change that to for example 24. Then, you can connect with "$ ssh -p 24 user@server".

Additional info: SSH connections time out at some point. You can change that on the server side by editing your `/etc/ssh/sshd_config`. Just add:

```
ClientAliveInterval 1200
ClientAliveCountMax 3
```

to the config and restart the sshd service with:
`$ sudo systemctl restart sshd`
With this, your clients will stay alive while ideling for 1 hour. (More info: https://www.tecmint.com/increase-ssh-connection-timeout/)\ (https://www.tecmint.com/increase-ssh-connection-timeout/)%5C)

An alternative is given in the `Byobu` section below, especially when you can't change the server settings.

SSH Keys

One of the most important (and secure) things about ssh are so-called `ssh keys`. They allow us to authenticate to a server by `public-key authentication` instead of a

password. This means, you have a private key that only you know (ideally password-protected!) and the server gets your public key. When you now try to connect to a server with your key, the server can identify you based on your key.
This increases the security immense compared to a password that can be cracked!

To generate a key pair, do the following:

1. go to `$ cd /your_home_directory/.ssh` or simply `$ cd ~/.ssh` \
2. `$ ssh-keygen -t ed25519 -C "<myname>@<server>"` \
3. enter the filename (you also can just press enter to get the default) \
4. enter a save password \
5. Finally, you get two files (default: `id_ed25519` and `id_ed25519.pub`) and an output like this:

```
The key's randomart image is:
+--[ED25519 256]--+
|              .++B|
|              oo +|
|         o.oo. o.|
|        E.B=.+. +|
|        S*+ *.+=.|
|           + O+ .|
|          o +.o. |
|         ..o.+ooo|
|          oo.+*+ |
+----[SHA256]-----+
```

Now, we can deploy our key to a server by typing:
`$ ssh-copy-id -i ~/.ssh/your_key.pub <user>@<server>`
Caution: Here, you have to give your public key (ending with .pub), as the command copies the content of your keyfile to the server. Never give the content of your private key file to anyone! And I recommend that you always protect your keys with a save password!

> Note: This only works, when passwort authentication is enabled! If not, your key has to be copied to the "autherized_keys" file at the server.

You can also create several keys for different servers. This is even more secure. You can specify the key you want to use with:
`$ ssh -i /path/to/your/private_key <username>@<server>`
(Here, you have to give your private key to the pub-key, you copied to the server. With this private key, you authenticate as the owner of the given pub-key)

Exercise

Lets try it out!

Create a key pair and copy it to "kseta@kseta-vm.etp.kit.edu". The password is "Kseta Bad-Herrenalb workshop2023". Then, login with your key pair. Be sure to copy your PUBLIC key!

SSH-Agents

In general, it is recommended to not use `agentforwarding` and instead type in your password each time. But most people are lazy and don't want to type in the password every time...
Therefore, we can use an `ssh-agent` to add our ID permanently to our ssh-client.

Note: This is only recommended for machines that you own or at least trust, because someone with root access could "steal" your identity by copying your agent! This you should not underestimate! Some popular hacks happend based on this. The official ssh man page says: Agent forwarding should be enabled with caution. Users with the ability to bypass file permissions on the remote host (for the agent's Unix-domain socket) can access the local agent through the forwarded connection. An attacker cannot obtain key material from the agent, however they can perform operations on the keys that enable them to authenticate using the identities loaded into the agent.

To activate an ssh agent, type in:
```
$ eval  ssh-agent
```
and then add your identity with:
```
$ ssh-add
```
or better with a timeout:
```
$ ssh-add -t 86400
```

This adds the key from `~/.ssh`

To list your identities, use:
```
$ ssh-add -l
```

You can add it to your *.bashrc* as:
```
alias start_agent='eval  ssh-agent ; ssh-add'
```

To delete all identities from an agent, use:
```
$ ssh-add -D
```
and to kill an agent: see below

And now comes the hacky part...
From a security point of view, I do not recommend using my function, but as I said, convenience is nice :D
In general, I recommend to kill an agent when finish working, so nobody can use it.

But if you do not kill the agent on logout, it remains in `/tmp/ssh-*`.
Therefore, it is at least better to reconnect to an existing agent then creating several permanent ones...
So feel free to add it to your *bashrc*

```
ssh-reagent () {
  if [ $HOSTNAME == "my_trusted_machine" ]
  then
    for agent in /tmp/ssh-*/agent.*; do
      export SSH_AUTH_SOCK=$agent
      if ssh-add -l 2>&1 > /dev/null; then
        echo Found SSH agent:
        ssh-add -l
        return
      fi
    done
    echo Cannot find ssh agent!
  fi
}
```

To destroy all agents you can access, you can use this short function:

```
destroy-agents () {
  agents="$(ps ax | grep [s]sh-agent | awk '{print $1}')"
  for agent in $agents; do
    export SSH_AGENT_PID=$agent
    echo $SSH_AGENT_PID
    ssh-agent -k # this just overrides the agent. The files
are still available in /tmp/ssh-*
  done
}
```

More on the topic: https://goteleport.com/blog/how-to-use-ssh-agent-safely/ (https://goteleport.com/blog/how-to-use-ssh-agent-safely/)

The better solution: ProxyJumps

From the security point, ProxyJumps are a better solution, since the ssh identity is not stored on the remote server. The bad part of it is that you have to type in your password every time... :D

To configure ProxyJumps, you have to create a config file in `~/.ssh` called `config` with such content:

```
Host jump_machine
    Hostname jump.etp.kit.edu
    Identityfile ~/.ssh/id_ed25519
    User <username>

Host target_server
    HostName target.etp.kit.edu
    User <username>
    Identityfile ~/.ssh/id_ed25519
    ProxyJump jump_machine
```

With this config, we can directly jump to the <target_server> via our jump machine using our given private key and do not deposite key information on the jump machine. Additionally, all other available ssh options can be specified in this config, like for example forcing IPv4 and so on.

> Note: ProxyJumps are very useful and sometimes the only possibility to connect to an (internal) network via a login node for example.

SSH Tunneling

There are two other useful things I want to show with SSH, both are somewhat related to ProxyJumps and can be summarized as `SSH tunneling`

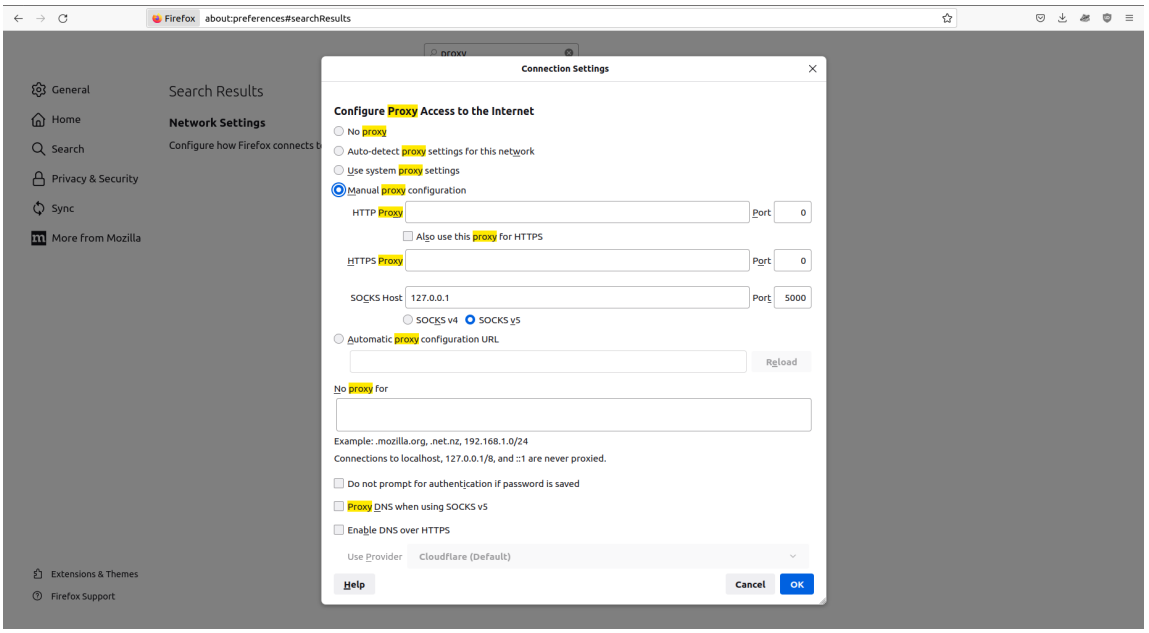
The first thing is the option `-D`. It enables a redirection of all network traffic over the remote machine. This is useful, if you want to access resources within a private network from home for example.

On a more technical level: ssh will act as a SOCKS server. Whenever a connection is made to this port, the connection is forwarded over the secure channel, and the application protocol is then used to determine where to connect to from the remote machine (see https://linux.die.net/man/1/ssh (https://linux.die.net/man/1/ssh)).

To enable the ssh tunnel, we have to decide for a port and type in:
`$ ssh -D <port> username@server`

Now, you are redirecting all network traffic over the remote machine.
An example where this can be used are the internal KIT websites that are only reachable from within the KIT network. In principle, the functionality is somewhat similar to a VPN, but you don't need additional software.
To access an internal website, you then just have to configure your browser to use a `SOCKS_v5` proxy.

> Note: ALL traffic is redirected. Hence, keep in mind to terminate the connection before doing secrete things like online banking etc!



Now, you can access internal websites with your browser!

If you don't want to redirect all your traffic, the `-L` option is the way to go. With this option, you can bind a specific port on your local machine to a specific port on the remote machine and only the traffic on this port is forwarded.
This option is useful, when you want to redirect specific applications as we will see in the `jupyter` section.
The syntax goes as follows:

```
$ ssh -L 8080:localhost:<PORT> <REMOTE_USER>@<REMOTE_HOST>
```

With this, we map our local port 8080 on localhost to a certain port of the remote host. A remote web application could then for example be accessed by typing `http://localhost:8080/` in your browser.

SCP

`Secure Copy` (SCP) is a tool to transfer files remotely using ssh.
The general syntax is:

```
$ scp <file> <username>@<IP address or hostname>:<Destination>
```

Like for `cp`, directories need the `-r` option.

Examples: (`-r` for directories)
Copy to remote:

```
$ scp (-r) <local file or directory> username@server:/path
/to/destination
```

Copy from remote:

```
$ scp (-r) username@server:/path/to/file_or_dir /local_path
```

`/to/destination`
To copy from remote to the current directory, just use `.` :
```
$ scp (-r) username@server:/path/to/file_or_dir .
```

Alternative: rsync

The better alternative is `rsync` . More information you can find for example at:
https://stackoverflow.com/questions/20244585/how-does-scp-differ-from-rsync (https://stackoverflow.com/questions/20244585/how-does-scp-differ-from-rsync)

SSHFS

Tool to mount a remote file system on your local machine via ssh (for example, a code repo for development, plots from plotting script etc etc)
Install:

```
$ sudo apt-get install sshfs
```

Now we create a local directory to mount the remote one:

```
$ mkdir remote_directory
```

and activate the sshfs connection:

```
$ sshfs user@server:/path_to_remote_folder/ ~/remote_directory
```

> Note: No relative paths for the remote dir (don't use "~")

To unmount the directory when you are finished (which is recommended), type:

```
fusermount -u ~/remote_directory
```

> Note: The unmounting can fail if you are inside the folder!

> WARNING: Deleting a mounted folder will delete the content of the folder!!! This means, you have to be very cautious and unmount a folder before deleting it from you local computer!

Alternative: ssh plugins for IDEs (see below)

BYOBU

Byobu is a terminal multiplexer based on `tmux` or `SCREEN`. In principle, it is just a wrapper config for the two multiplexers with many convenience functions. In general, tmux or screen are also fine to use, but in my opinion, some of the additions of byobu are very nice and increase productivity a lot without learning many shortcuts.

The idea behind such a terminal multiplexer is to run your tty as a separate process that is not bound to your actual terminal anymore. This has the advantage that you can "detach" from a session and reconnect again later. By this, you can keep your ssh connections open or scripts running etc without the need to keep your terminals open (and the risk of time outs, at least mostly).

More info: https://wiki.ubuntuusers.de/byobu/ (https://wiki.ubuntuusers.de/byobu/)

To install Byobu, type:

```
$ sudo apt install byobu
```

Now, it can be started by typing:

```
$ byobu
```

On first startup, you are asked to chose the byobu backend. I recommend `tmux`.

Most useful shortcuts:

- `Ctrl+a` : main control sequence
- `Ctrl+a -> d` : press first "control + a", then "d" to detach from a seesion
- `Ctrl+a -> <Number>` : jump to window with number <Number>
- `F2` : create a new window
- `Shift+F2` : create a new horizontal split
- `Ctrl+F2` : create a new vertical split
- `Ctrl+Shift+F2` : create a new session
- `F3` and `F4` : switch between windows
- `Ctrl+Shift+F3/F4` : move windows
- `Alt+arrow(left/right)` : switch between windows the better way
- `Shift+arrow(left/right/up/down): switch between splits
- `Alt+arrow(up/down)` : switch between sessions
- `F5` : reload profile
- `Ctrl+F5` : reconnect SSH sockets
- `F6` : detach from session
- `Ctrl+F6` : kill window or split
- `F7` : scroll mode (end with `Enter`)
- `F8` : rename window
- `F8` : rename window
- `Ctrl+Shift+F8` : rename window

- F9 : configuration Menu
- `Shift+F12` : deactivate/activate function keys (e.g. for htop)

Sometimes, it can happen, that parts of your local env get lost (lets say it like this, too technical else).
If you need a display set (for some software this is necessary), the you can export the following:

```
$ export DISPLAY=localhost:0.0
```

This can be added to your *.bashrc* or you can write a bash function that checks wether you are in a byobu/tmux session:

```
if { [ "$TERM" = "screen-256color" ] && [ -n "$TMUX" ]; } th
en
    echo "Set display"
    export DISPLAY=localhost:0.0
fi
```

Also matplotlib can have problems when no display is available. The easiest workaround is to change the mplbackend, e.g.:

```
if { [ "$TERM" = "screen-256color" ] && [ -n "$TMUX" ]; } th
en
    echo "MPL Backend in tmux/byobu session changed to 'Agg
'."
    export MPLBACKEND='Agg'·
fi
```

> Note: Those are no 100% fool proof ways of checking on the terminal but should be sufficient, if you don't mess around with the envs :D

Virtualenv

Virtual python environment that provides a python installation that is isolated from the system. This has multiple advantages:

- you do not need root permissions to install packages
- you don't have to clean up your system as it stays clean
- you can install different (e.g. older) versions of software that you need without affecting the main system
- you can have multiple setups for different tasks
- you can "freeze" your environment and save it or send it to other people that can easily replicate your software status

To set up a virtual environment, we do:

$ python3 -m venv <name>

> Note: Eventually, you have to install "venv"!

To activate your new env, you have to type:
`source <path-to-env>/bin/activate`

Now, we are in an isolated python environment, what you can see on the prompt.

To install python packages, you can use pip as usual. For example:

- `$ pip install numpy`
- `$ pip install pandas`
- `$ pip install matplotlib`
- `$ pip install seaborn`
- we can also specify a version with `$ pip install <package>==<version_number>`
- to update a package, use `$ pip install --upgrade <package>`

If you for example share your software setup with your collegues, you can easily use pip to do so:
`$ pip freeze > requirements.txt`
With this, we dump the env in a file named `requirements.txt` that contains the installed packages. For the installed virtual env above, it looks like this:

```
cycler==0.11.0
kiwisolver==1.3.1
matplotlib==3.3.4
numpy==1.19.5
pandas==1.1.5
Pillow==8.4.0
pyparsing==3.0.9
python-dateutil==2.8.2
pytz==2022.7.1
scipy==1.5.4
seaborn==0.11.2
six==1.16.0
```

To leave your virtual env, you just need to type:
`$ deactivate`

Another person (or you at another PC) now easily reproduce the environment by creating a new one as explained above and activate it. When it comes to installing the packages, we use the requirements.txt file to install the exact same software as before:
`$ pip install -r requirements.txt`

> Note: It is important to also use the same python version! You can check this with "$ which python3" or simply by entering the python CLI (just type "python3").

If you need other versions of python, for example python2, just use the setup command with python2:

```
$ python2 -m venv my_venv
```

Alternatively, you can also use the commandline tool without python to specify a certain installation of python directly (for example when you compiled it by yourself):

```
$ virtualenv --python="/usr/bin/my_special_python_version" "/path/to/new/virtualenv/"
```

More to read: https://www.freecodecamp.org/news/manage-multiple-python-versions-and-virtual-environments-venv-pyenv-pyvenv-a29fb00c296f/ (https://www.freecodecamp.org/news/manage-multiple-python-versions-and-virtual-environments-venv-pyenv-pyvenv-a29fb00c296f/)

Alternative: Conda

Conda is a more enhanced virtual environment tool. It also works with other languages then python, as it emulates a full environment. It actually ships python itself and installs a native package manager.

Which should i use then?

In general, I would recommend to use `conda`, as it has many more possibilities. But it also has a way more steep learning curve than `venv`. So if you aren't experienced and "just want it to run", `venv` would be the more easy approach.
If you insist on using `Windows`, (ana)conda is your best - if not only - chance to get scientific software running!

I usually adapt it to my usecase.
If venv is suitable for my purpose, I use it as it is fast and easy to setup. For example for a small programming project where I want my env and my code in one place. Else, if a more enhanced environment is necessary, I use `conda`.
Generally spoken, `venv` is more lightweight and focuses solely on python + you don't need any external sources for packages (This is an advantage and disadvantage...).
`conda` allows us to do way more complexe things. For example, if you need a `ROOT` installation with python bindings, you can simply install it with:

```
$ conda install -c conda-forge root
```

But this then obviously is way more heavy than a virtualenv...

So as written above, it really depends on your use-case!

Note: When working on different operating systems, conda is always to be preferred, as it is way more convenient by shipping python with it, not relying on the OS. Also, when you are using MacOS with M1/M2 (ARM).

To install, follow: https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html (https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html)

For python only, use Miniconda: https://docs.conda.io/projects/conda/en/latest/user-guide/install/rpm-debian.html (https://docs.conda.io/projects/conda/en/latest/user-guide/install/rpm-debian.html)

Many packages to instal can be found at: https://anaconda.org/conda-forge/ (https://anaconda.org/conda-forge/)

Further information: ***https://towardsdatascience.com/a-guide-to-conda-environments-bc6180fc533 (https://towardsdatascience.com/a-guide-to-conda-environments-bc6180fc533)*** **,** ***https://towardsdatascience.com/managing-project-specific-environments-with-conda-b8b50aa8be0e (https://towardsdatascience.com/managing-project-specific-environments-with-conda-b8b50aa8be0e)***

Cheat-Sheet: ***https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf (https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf)***

GIT

I really recommend to use git!

`git` is a tool for version control of your software. It is very useful for programming, gathering configs all together, etc. The special thing is that every so-called "commit" is tracked and you can revert, compare, and checkout certain commits.

`git` consits of two parts: your local working repository and the current "production state" remote.

> Note: git is for code and configs (small files). It is not recommended to push large files like .root or .hdf5 to a git repo!

Getting started

We will now only discuss the very basics. If you are not familiar with git and interested, try out this tutorial: https://githowto.com/ (https://githowto.com/)

First, install it if necessary:

```
$ sudo apt install git
```

Now first, we should configure some basic things:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your email"
$ git config --global init.defaultBranch main
```

Then, we need to create a github account: [www.github.com (http://www.github.com)](http://www.github.com)

> Note: git can also be used offline, but online is way more convenient and you can easily share your work with other people and between computers.

As a next step, we should create an ssh key (or use an existing one) and add it to github. Alternatively, you can create a `token` in the web interface, but as ever: I recommend using keys.

To actually use git, there are two possibilites. We can `git clone` an existing repo or we can `git init` a new one.

If we want to work on an existing repo, we can just clone it with:

```
$ git clone git@github.com:<username>/<repo>.git
```

If we want to start a new repo, we first have to create an empty repo at github.com. (There are hacky ways to do it from CLI, but I recommend to just do it in the web interface)

Therefore, we need to create a folder:

```
$ mkdir example_repo
```

and `cd` into it:

```
$ cd example_repo
```

To initialize our repo, we have to type in the directory:

```
$ git init
```

If we want a remote repository, we have to add the "origin" according to the created remote repo:

```
$ git remote add origin git@github.com:<username>/<repo_name>.git
```

To check the remote origin, you can type:

```
$ git remote -v
```

To check the current repo status, type in:

```
$ git status
```

The output should be something like this:\

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to tr
ack)
```

Now, we can create a file within the local repository, for example with vim or nano:
`$ vim example_file.txt`

If we check the `status` again, the file should now be listed under "Untracked files".

To actually add the file to the repository, we now have to "add" the changed files to a "commit" it with:
`$ git add <filename>`
and check again the status. The file now should be listed in: "Changes to be committed"

You can also add a complete directory or the current directory with all containing files with
`$ git add .`

If you have accidentally added a file, you can untrack (and keep it!) with:
`$ git reset <filename>`

To remove a file from a repo (and DELETE it!), use:
`$ git rm --cached <filename>`

After adding the files, we now prepare the commit to be "pushed" to github.com:

`$ git commit -m "First example file added to the repo."`

Additionally, we can add a `README` that is shown on the github page of the repo.

The `git status` now should be clean again.

As a last step, we now have to "push" our commits:
`$ git push --set-upstream origin main`

Now, we can work on our code and add files, change files etc and commit and push them again.

If we changed a file, we can review the changes with:
`$ git diff <filename>`

> If you do not have a git repo and want to compare changes, "vimdiff" is very usful!

To discard the current changes, we can "checkout" the remote file status:
`git checkout -- <filename>` (the "--" is no typo!)

To review our commit history, we can type:
`$ git log`

To compare the changes of a commit, we can also use `git diff` . Therefore, we need to get the commit hashes from `git log` and type:
`$ git diff <hash1>..<hash2>`

> Note: You don't need the full hash. 4 chars are sufficient, if unique!

> Note: git diff compares two endpoints and not a range! This means, if you revert changes in the next commit and compare it with the one before, you don't get any output, for example.

.gitignore

There is a possibility to exclude certain files, like for example *.pyc, in general with a `.gitignore` file. In principle, this can be done by two different approaches. You can blacklist certain file types/directories etc. Or you can blacklist everything and then whitelist all that you want to commit. This way round, you do not commit stuff randomly that you don't want in your repo. My recommendation is to do it the second way.

An example file could then look like this:

```
##### .gitignore file #####

# exclude everything
*
# include everything you want
!.gitignore
!*.py
# we can also include directories
!/directory
```

If you want to do it the first way, the syntax is like this:\

```
# Blocklist files/folders in same directory as the .gitignor
e file
/*

#ignore all kind of files
*.*

# Includelist some files
!.gitignore
!README.md

# Ignore all files named .DS_Store or ending with .log
**/.DS_Store
**.log

# Includelist folder/a/b1/ and folder/a/b2/
# trailing "/" is optional for folders, may match file thoug
h.
# "/" is NOT optional when followed by a *
!folder/
folder/*
!folder/a/
folder/a/*
!folder/a/b1/
!folder/a/b2/
!folder/a/file.txt
```

The above code would ignore all files except for .gitignore, README.md, folder/a/file.txt, folder/a/b1/ and folder/a/b2/ and everything contained in those last two folders. (And .DS_Store and *.log files would be ignored in those folders.)

Obviously I could do e.g. !/folder or !/.gitignore too. (taken from: [https://stackoverflow.com/questions/987142/make-gitignore-ignore-everything-except-a-few-files (https://stackoverflow.com/questions/987142/make-gitignore-ignore-everything-except-a-few-files)](https://stackoverflow.com/questions/987142/make-gitignore-ignore-everything-except-a-few-files))

This can also be configured globally for all of your local repos!
For the MacOS-people: This is very useful to generally ignore the wierd `.DS_Store` files :D

Therefore, we first need to create a global .gitignore file with a text editor:

```
$ vim ~/.gitignore
```

Now, we can set the global exclude file:

```
$ git config --global core.excludesFile '~/.gitignore'
```

Some general recommendations: %TODO

- use meaningful commit messages!
- It is better to split your progress in multiple commits than one big

> Note: The github.com search is total crap (I really don't know why...). I recommend cloning a repo and using "grep" to search for things, since it is way more efficient.

Git Alias

Git aliases are helpful to increase productivity as we do not need to type in more than necessary (remember: lazy is good! ;D)

To create an alias, we use the following syntax:

```
$ git config --global alias.<alias> <git-command>
```

For example:

```
$ git config --global alias.co checkout
```

Not very serious example ;)

Lets create a random commit message alias using the random commit message generator: https://whatthecommit.com (https://whatthecommit.com)!

First, we need to create a file called "wtc.sh" with the following content:

```
#!/bin/bash
message=$(curl --silent https://whatthecommit.com/ | sed 's
/<\/*[^>]*>//g' | tail | head -3 | tr -d '\n')
echo $message
git commit -m "$message"
```

Now, we need to make it executable with:

```
$ chmod +x wtc.sh
```

The next step is to create the alias:

```
$ git config --global alias.yolo '!sh ~/wtc.sh'
```

No we can commit with random messages by using:

```
$ git yolo :D
```

IDEs

For code developement, IDEs can be very helpful. Many people use vscode. It is free, easy to use, and has many extensions that enable powerful features. On top of that, you can use it for all languages. So it is a good recommendation overall.

To install it, just type:

```
$ sudo snap install --classic code
```

I am usually using `vim`, one of the default CLI text editors in linux, for smaller projects. It is very flexibel and you can add all functionality you want. But its quiet hacky and only recommended for people that are interested in configuring and hacking basic things together whithin linux (#nerds). And the learning curve is quiet steep. But if you spent the time, it can be very productive and powerful!

What I personally use for bigger projects are the IDEs from jetbrains. The pro versions are free for university members, which is nice. From my point of view, they are a little bit more ausgereift than vscode and provide in general a more professional workflow. But as I said, this is very subjective!

Here I recommend Pycharm (https://www.jetbrains.com/de-de/pycharm/ (https://www.jetbrains.com/de-de/pycharm/)) for python, CLion (https://www.jetbrains.com/de-de/clion/ (https://www.jetbrains.com/de-de/clion/)) for C/C++. There are also IDEs for web development, database development and even a lightweight vscode clone :D

Juypter

Another alternative for development (at least for small projects or visualizations and summarization of results -- like this tutorial here :)) is `jupyter`, as it provides a nice environment to easily test code. I especially use it for enhancing my plotting scripts, as it allows you to split your code into different code cells that can be executed individually and the results are cached. By this, it is not necessary to rerun the whole script every time. Meaning, I can for example read in the data in my first code cell, which is then cached, and only rerun the plotting steps while enhancing the plots. This saves a lot of time since I only have to read in the data once.
The bad thing about it is the RAM consumption...

> Warning: Since everything is cached in the RAM, it can be very demanding and memory consuming. So keep that in mind. This can kill your computer!

The easiest way to install jupyter is within a virtual python environment!

```
$ pip install jupyter
```

Now, we can set a password:
```
$ jupyter notebook password
```

To run a jupyter notebook, just type:
```
$ jupyter notebook
```
and follow the instructions given in the CLI.

Shortcuts

When working with jupyter, getting used to the shortcuts is really helpful and increases productivity. An overview is available when pressing `h`.

> Note: There are two modes within jupyter: the command mode and the edit mode. To enter the edit mode, just click in a cell. To get to command mode, press `Esc` or `Ctrl+M`

command mode:

- `Esc` : command mode
- `h` : help
- `m` : convert cell to markdown
- `c` : copy cell
- `x` : cut cell
- `v` : paste cell
- `Shift+Return` : execute cell
- `dd` : delete cell

edit mode:

- `Ctrl-Shift-Minus` : split cell at cursor(s)
- `Shift+Tab` : tooltip; in code cells: shows docstring of function

Themes

There are several packages that help to customize and personalize your jupyter experience. One is for example `jupyterthemes`. It can be installed with:
```
$ pip install jupyterthemes
```

To list the available themes, type in:
```
!jt -l
```
in a cell (see below)

In [37]:
```
!jt -l
```

```
Available Themes:
   chesterish
   grade3
   gruvboxd
   gruvboxl
   monokai
   oceans16
   onedork
   solarizedd
   solarizedl
```

Now we chose one with:

```
!jt -t chesterish
```

In [46]:
```
!jt -t grade3
```

more info: https://towardsdatascience.com/customize-your-jupyter-notebook-theme-in-2-lines-of-code-fc726cea1513 (https://towardsdatascience.com/customize-your-jupyter-notebook-theme-in-2-lines-of-code-fc726cea1513)

Working Example: python plotting

In [19]:
```
import pip

def install(package: str):
    """
    Function to install packages.

    Parameters
    ------
    package: str

    Return
    ------
    None
    """
    try:
        __import__(package)
    except ImportError:
        pip.main(['install', package])
        #!pip install package  # with this, we can do system calls fi
```

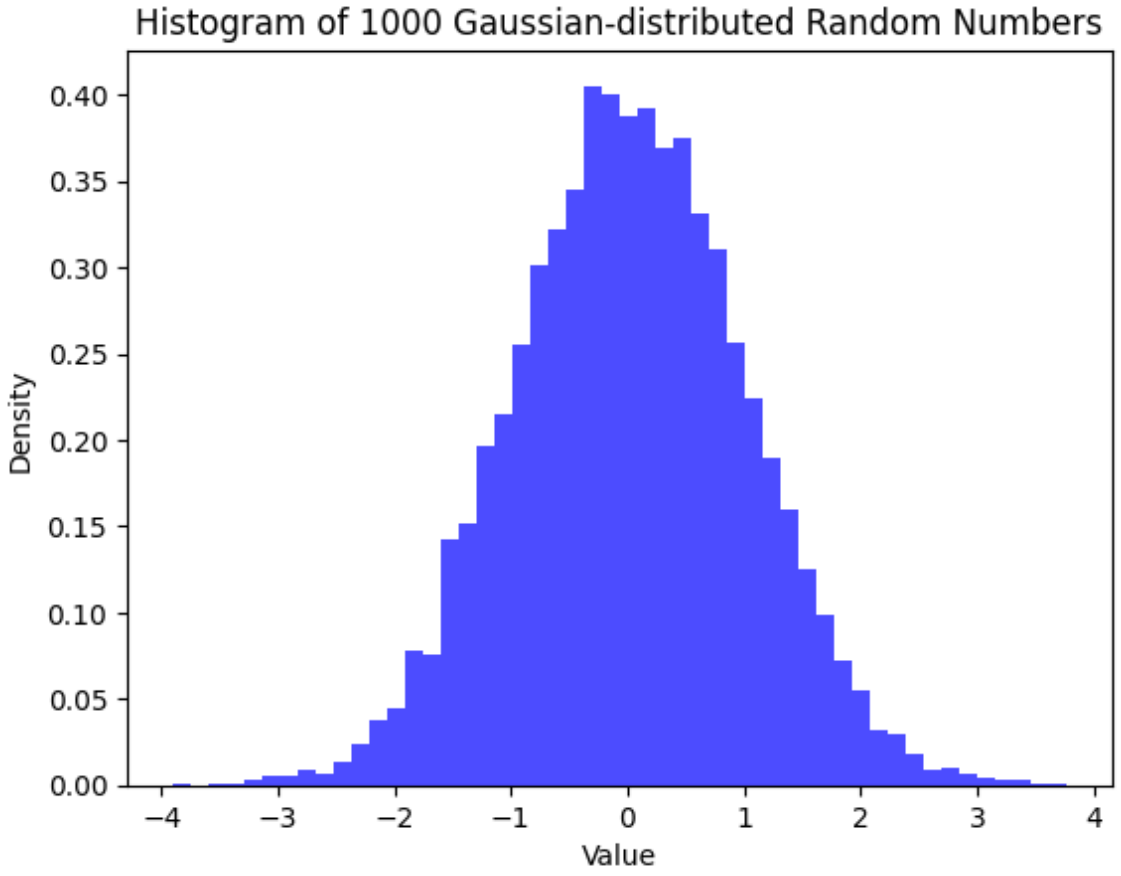
In [20]:
```
install("pandas")
install("matplotlib")
```

In [22]:
```
import numpy as np
from matplotlib import pyplot as plt
```

In [23]:
```
# this lets us create the plots inline within the jupyter notebook
%matplotlib inline
```

In [30]:
```
# Generate 1000 random numbers drawn from a Gaussian (normal) distri
data = np.random.normal(0, 1, 10000)
```

In [32]:
```
# Plot a histogram of the data
plt.hist(data, bins=50, density=True, alpha=0.7, color='blue')
plt.xlabel('Value')
plt.ylabel('Density')
plt.title('Histogram of 1000 Gaussian-distributed Random Numbers')
plt.show()
```



Remote Server

A cool thing is, we can spawn a permanent server on a remote machine and connect to it, whenever we want to use it. For this, we need to connect to the server via ssh and start a jupyter server there with:

`$ jupyter notebook --no-browser --port=<PORT>`

I usually use a port in the range of 8000-8100.

To connect to this server, we use a `ssh tunneling` :

`$ ssh -L PORT:localhost:PORT username@server`

With this, we redirect all traffic over PORT.

As a last step, we need to go to http://localhost:PORT (http://localhost:PORT)

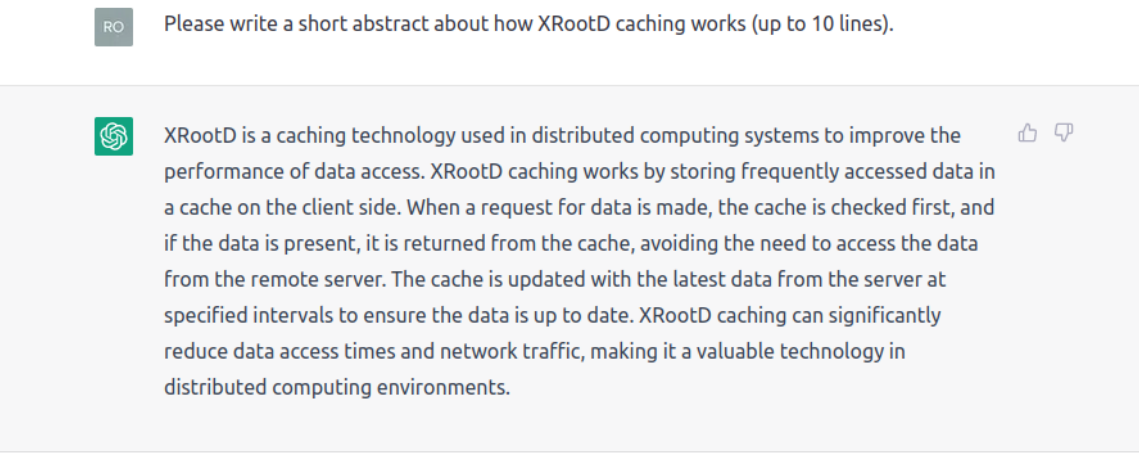
This server now can run for ever and you have your code available there without any additional sshfs/whatever tools.

Excercise

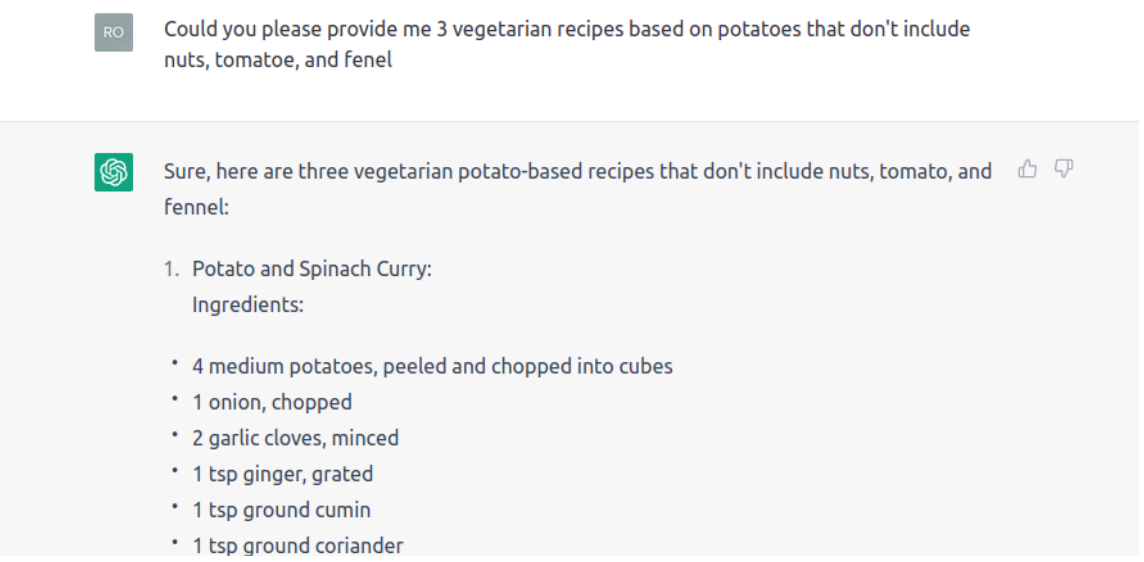
> Create a venv and spawn a jupyter server at kseta-vm.etp.kit.edu. Choose a port between 8001 and 8079. Then, connect to it from your laptop.

ChatGPT

You are sick of this shit and just want someone else to write your thesis?
Try out ChatGPT! :D



It can be really useful to explain basic terms. And many technical questions are answered very well (like for example: "how do I do X in Linux", or things like that). Furthermore, it is especially helpful for generating lists of suggestions instead of googeling things multiple times:



- 1/2 tsp turmeric
- 1/2 tsp paprika
- 1/4 tsp cayenne pepper
- 1 cup spinach leaves
- 1/2 cup vegetable broth
- Salt and pepper to taste
- 2 tbsp vegetable oil

Directions:

1. Heat the oil in a large pan over medium-high heat.
2. Add the onion and sauté until translucent.
3. Add the garlic and ginger and cook for

You even can let it generate small code snippets:

RO Please create me a short python script to read in columnar data from csv and plotting a histogram from the first column called "energy" with pandas and matplotlib

Sure, here's an example Python script that should do what you've described using pandas and matplotlib:

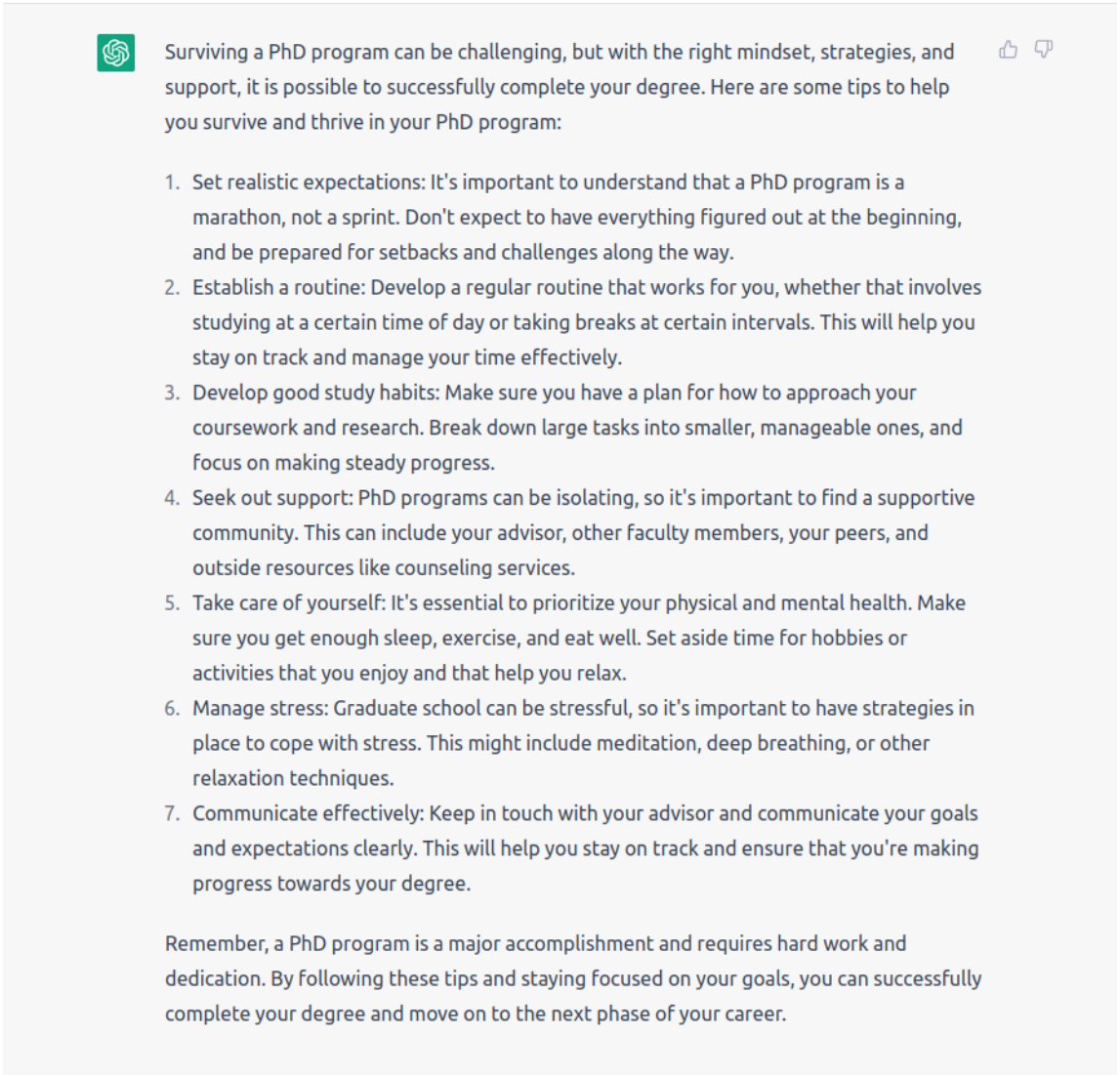
python Copy code

```python
import pandas as pd
import matplotlib.pyplot as plt

# Read in the CSV file using pandas
data = pd.read_csv('your_file.csv')

# Plot a histogram of the "energy" column using matplotlib
plt.hist(data['energy'], bins=10)

# Add axis labels and a title
plt.xlabel('Energy')
plt.ylabel('Count')
plt.title('Histogram of Energy')

# Show the plot
plt.show()
```

This script assumes that your CSV file is called "your_file.csv" and that the first column is named "energy". You can adjust the number of bins for the histogram by changing the value of the `bins` argument in the `plt.hist()` function call.

And it can also be useful to get you some general tips:

RO How to survive aPhD?



Surviving a PhD program can be challenging, but with the right mindset, strategies, and support, it is possible to successfully complete your degree. Here are some tips to help you survive and thrive in your PhD program:

1. Set realistic expectations: It's important to understand that a PhD program is a marathon, not a sprint. Don't expect to have everything figured out at the beginning, and be prepared for setbacks and challenges along the way.
2. Establish a routine: Develop a regular routine that works for you, whether that involves studying at a certain time of day or taking breaks at certain intervals. This will help you stay on track and manage your time effectively.
3. Develop good study habits: Make sure you have a plan for how to approach your coursework and research. Break down large tasks into smaller, manageable ones, and focus on making steady progress.
4. Seek out support: PhD programs can be isolating, so it's important to find a supportive community. This can include your advisor, other faculty members, your peers, and outside resources like counseling services.
5. Take care of yourself: It's essential to prioritize your physical and mental health. Make sure you get enough sleep, exercise, and eat well. Set aside time for hobbies or activities that you enjoy and that help you relax.
6. Manage stress: Graduate school can be stressful, so it's important to have strategies in place to cope with stress. This might include meditation, deep breathing, or other relaxation techniques.
7. Communicate effectively: Keep in touch with your advisor and communicate your goals and expectations clearly. This will help you stay on track and ensure that you're making progress towards your degree.

Remember, a PhD program is a major accomplishment and requires hard work and dedication. By following these tips and staying focused on your goals, you can successfully complete your degree and move on to the next phase of your career.

It can be a really helpful tool if used correctly. But keep in mind that this is a "stupid" language model and is not able to actually rate, if the provided information is correct. It also sometimes happens that it just tells you something and invents sources for that (no good scientific behaviour!! :D).
So be carefull and do not use for example generated code unreviewed!

And it is generally not reliable in calculating math things, or solving problems, etc. It is a language model and no real AI!