UPPSALA
UNIVERSITET

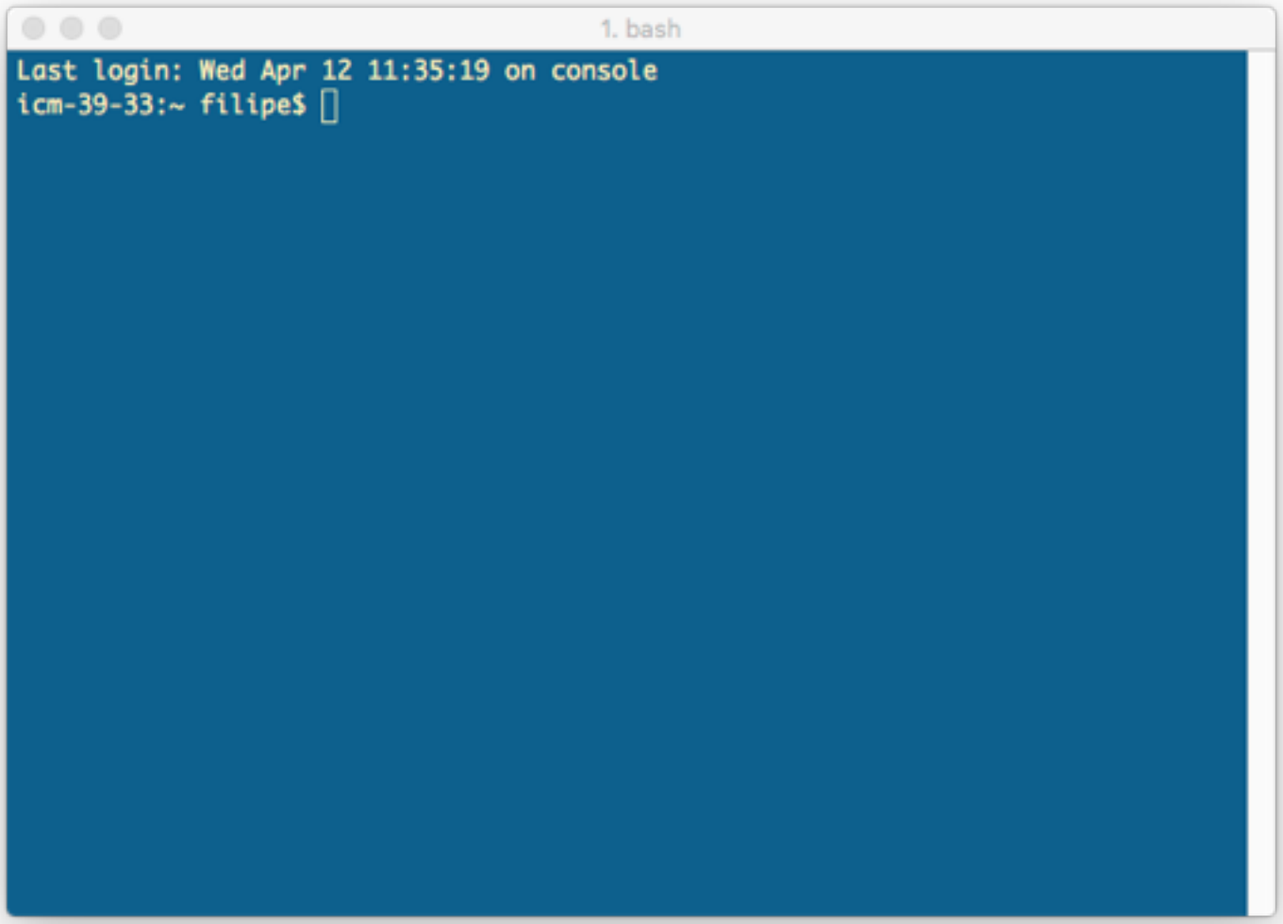
# An Introduction To The Unix Shell, Interactive Python And Git Repositories

2017-05-15

Advanced Scientific Programming with Python

# The Unix Shell

- Provides a command line interface (CLI) to the operating system
- Large variety of shells: `bash`, `tcsh`, `csh`, `ksh`, `zsh`
- We'll focus on `bash`, the default on most systems
- Documentation can be found by typing `man <command>`, e.g. `man bash`.




A shell on Mac OS X, a Unix system

# The Unix Shell

- On login the system executes:

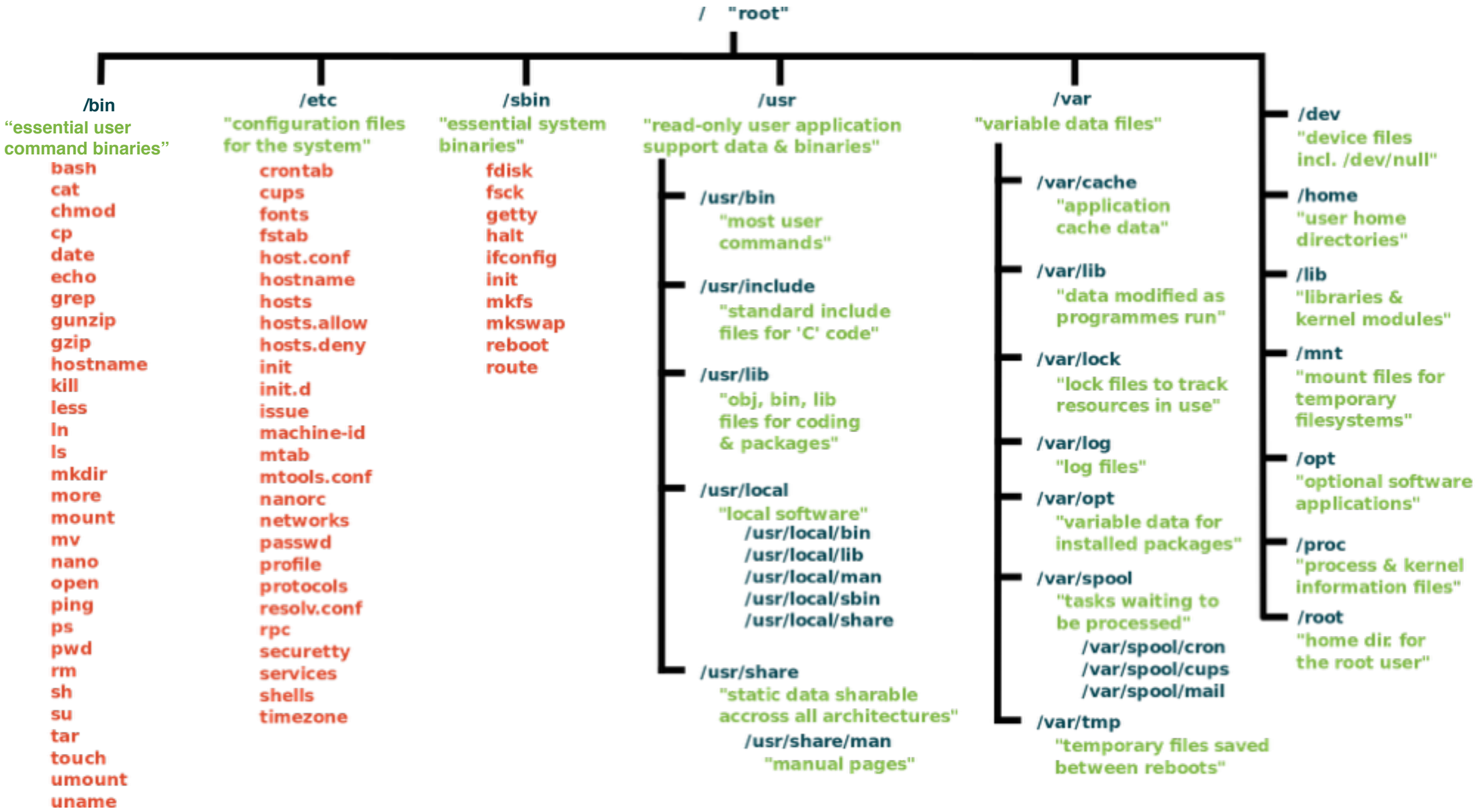

- To avoid complications it's recommended to have your `~/.bash_profile` as a symlink to `~/.bashrc`.

- The shell checks the first line of every program and if it finds `#!<interpreter>` it uses the the given interpreter to evaluate the file (e.g. `#!/usr/bin/env python`)

- bash is a very powerful shell that can be used for input redirection (<, >, &, etc…), job control (fg, bg, jobs), file globbing (*, [0-9], …).

- Check the **man page** for more!

# Standard Unix Filesystem

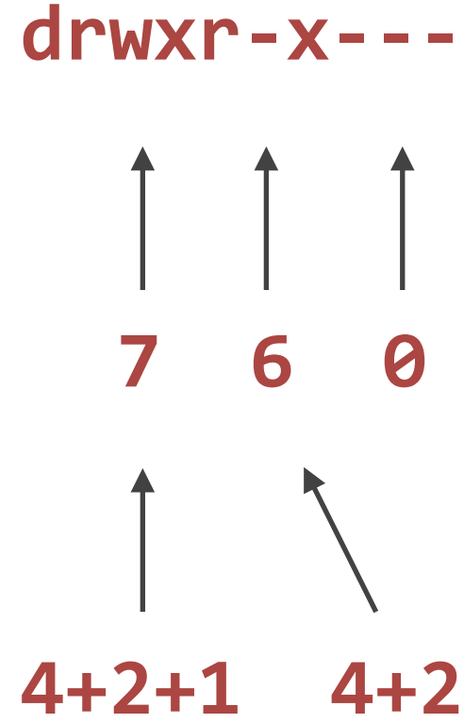


# Basic Unix Utilities

- `ls` - list directory contents. Example: `ls -ltr ~`
- `cd` - change working directory. Example: `cd ..`
- `cp` - copy files. Example: `cp -a ~/data/*.h5 /mnt/backup/data`
- `mv` - move files. Example: `mv * package/`
- `rm` - remove files. Example: `rm -rf ~/.*`
- `mkdir` - create directories. Example: `mkdir -p ~/data/run1/001/`
- `ln` - create links. Example: `ln -s /long_path_to_data/ data`
- `grep` - text filter. Example: `grep Result ~/data/log.txt`
- `more` - display file contents. Example: `more ~/data/log.txt`

# File Permissions

- In Unix systems most things are represented by files.
- All files have owner, group and other permissions.
- The basic 3 permissions are read (4), write(2) and execute(1).
- Permissions can be changed with **chmod**
- Owners can be changed with **chown**
- Permissions of newly created files are determined by the users' **umask**




Examples:

```
$ chmod 760 my_file
$ chown filipe.xray my_file
```

# File Globbing

Wildcards can specify sets of files. For example:

```
$ ls molecules
cubane.pdb  ethane.pdb  methane.pdb
cubane.txt  ethane.txt  methane.txt

$ rm molecules/*.txt
$ ls
cubane.pdb  ethane.pdb  methane.pdb
```

`*` matches 0 or more characters

`?` matches exactly 1 character

`[abc]` matches 1 character in the set

`[a-z]` matches 1 character in the range

For more details check `man 7 glob`

# Demo: Files And Permissions

# Pipes And Filters

- You can use `>` to redirect the output of a command to a file, e.g.:

```
$ wc -l *.pdb
12  cubane.pdb
20  ethane.pdb
9  methane.pdb
41  total
```

```
$ wc -l *.pdb > lengths.txt
$ cat lengths.txt
12  cubane.pdb
20  ethane.pdb
9  methane.pdb
41  total
```

- `>>` works similarly, but the output is appended to the file instead of replacing it.

# Pipes And Filters

- You can use `|` (called a pipe) to make the output of a command the input of the next, e.g.:

```
$ sort -n lengths.txt
9  methane.pdb
12  cubane.pdb
20  ethane.pdb
41  total
```

```
$ wc -l *.pdb | sort -n
9  methane.pdb
12  cubane.pdb
20  ethane.pdb
41  total
```

# Pipes And Filters

- You can use `<` to make the content of a file the input of a command, e.g.:

```
$ wc -l methane.pdb
9  methane.pdb
```

is the same as:

```
$ wc -l < methane.pdb
9  methane.pdb
```

- Most commands that read from standard input also accept files as arguments so this is not as useful.

# Standard File Descriptors

- Every process opens 3 numbered standard file descriptors, **`stdin`** (0), **`stdout`** (1), **`stderr`** (2).
- They correspond to the input (**`stdin`**), output (**`stdout`**) and error messages (**`stderr`**).
- You redirect the streams independently, e.g.:

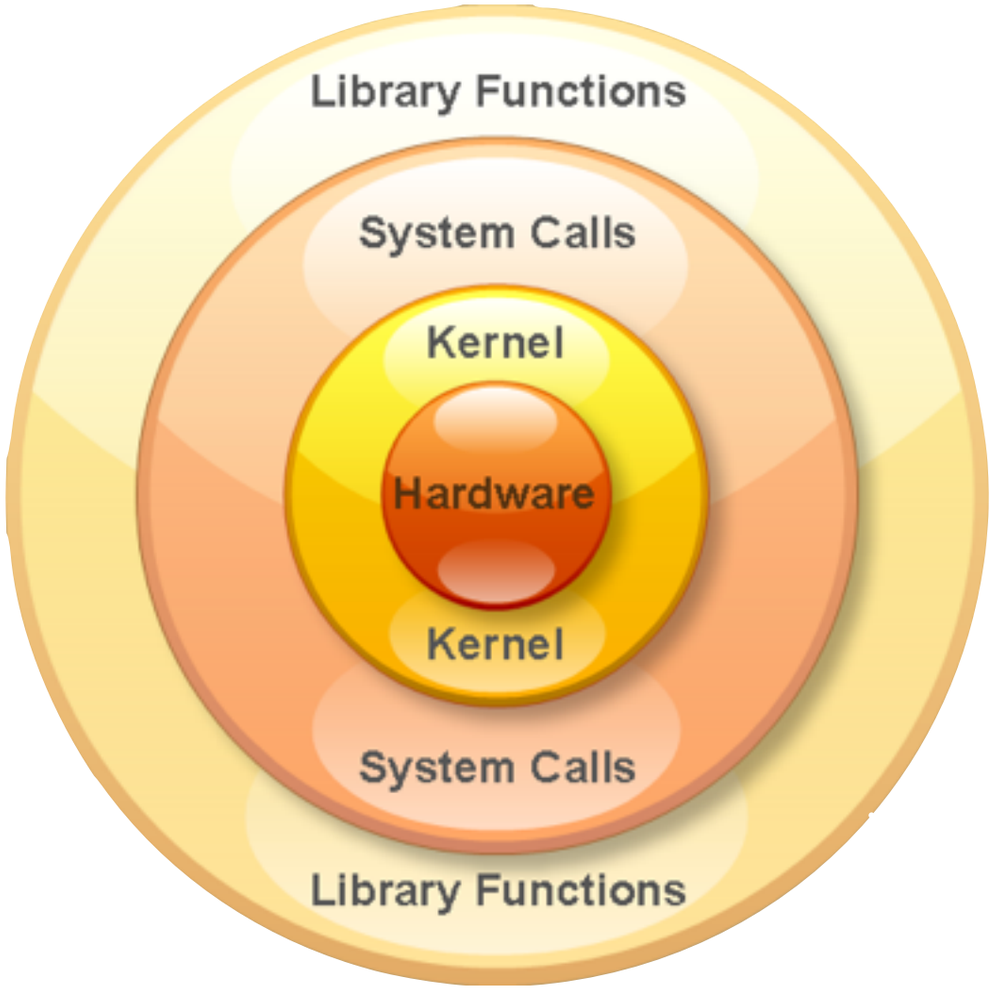
```
$ ls /bin/foo /bin/ls > /dev/null
ls: /bin/foo: No such file or directory

$ ls /bin/foo /bin/ls 2> /dev/null
/bin/ls
```

- You can combine streams, e.g.:

```
$ ls /bin/foo /bin/ls > /dev/null 2>&1
$ ls /bin/ls /bin/foo  2>&1 |wc -l
2
```

# Unix Processes

- A **process** is a running program and has a unique `pid` (process id).
- Each process has a parent. It inherits its **environment** from the parent.
- You can use `pstree` to check the process tree.
- Each process has its own memory address.
- Processes can have one or more **threads**.
- All threads in a process share the same memory space.



# Process Environment

- Each process has a parent process from which it inherits the environment.
- `PATH` is an environment variables used to search for executables. You can display it with `echo $PATH`.
- You can change variables using `export`, e.g. `export PATH=/bin`.
- You can use `which` to find out the full path of an executable (e.g. `which cp` returns `/bin/cp`).
- `LD_LIBRARY_PATH` is another, which lets the program know where to find dynamic libraries.
- You can use `ldd` to print out the libraries found by the system:

```
[root@login ~]# ldd /bin/cat
linux-vdso.so.1 =>  (0x00007ffc4b73c000)
libc.so.6 => /lib64/libc.so.6 (0x00007f8e0a978000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8e0ad28000)
```

- As usual, `man 7 environ` for more information.

# Job Control

- Signals can be used to control a process.
- `Ctrl+C (SIGINT)` asks the process to terminate.
- `Ctrl+Z (SIGSTP)` suspends the process. You can use `fg` to continue the process.
- You can also use the `kill` command to send signals.
- `kill 9 <pid>` sends the `SIGKILL` signal to the process causing it to terminate immediately.
- Certain processes (stuck inside kernel calls) cannot be killed.
- You can also use `nice` to control the priority of execution of the process.
- Check `man 7 signal` for more information.

# Demo: Pipes And Job Control

**If you need to refresh your knowledge check:**

**https://swcarpentry.github.io/shell-novice/**

# Some More Useful Programs

# SSH Tips And Tricks

- You can use ssh to login to a remote system:

```
ssh filipe@davinci.icm.uu.se
```

username host domain

- You can edit your ~/.ssh/config to save you some typing:

```
Host login davinci.icm.uu.se
     Hostname davinci.icm.uu.se
     ForwardAgent yes
     ForwardX11 yes
     User filipe
```

- Using ssh keys one does not need to retype your password

  `$ ssh-keygen` (this creates the key pair. Use a password!)

- This will create a `~/.ssh/id_rsa` (this is the secret!) and `~/.ssh/id_rsa.pub` (the public key).

- To use it copy the contents of the public key to the `~/.ssh/authorised_keys` in the target system. Make sure the file is `chmod 600.`

# Remote File Copy

- You can use `scp` to copy files:
  `scp -r my_dir davinci:`
- For larger transfers you can use `rsync`.
- It can be used to continue transfers and to synchronise two directories.

  `rsync -av my_images davinci:data`

  copies my_images inside the data directory

  `rsync -av my_images/ davinci:data`

  copies the files inside my_images inside the data directory
- Be careful with it as it can easily delete files in your own computer! Test with -n first!
- For very large transfers it's best to use Globus Online (globus.org).

# Working Remotely

- `screen` allows you to maintain a remote session even without a permanent connection. Example:

```
localhost$ ssh big_machine
big_machine$ screen
screen_session$ ./my_slow_script.py
```

- Now you can detach from the session (usually with `Ctrl+A+D`)

```
[detached]
big_machine$ exit
localhost$
```

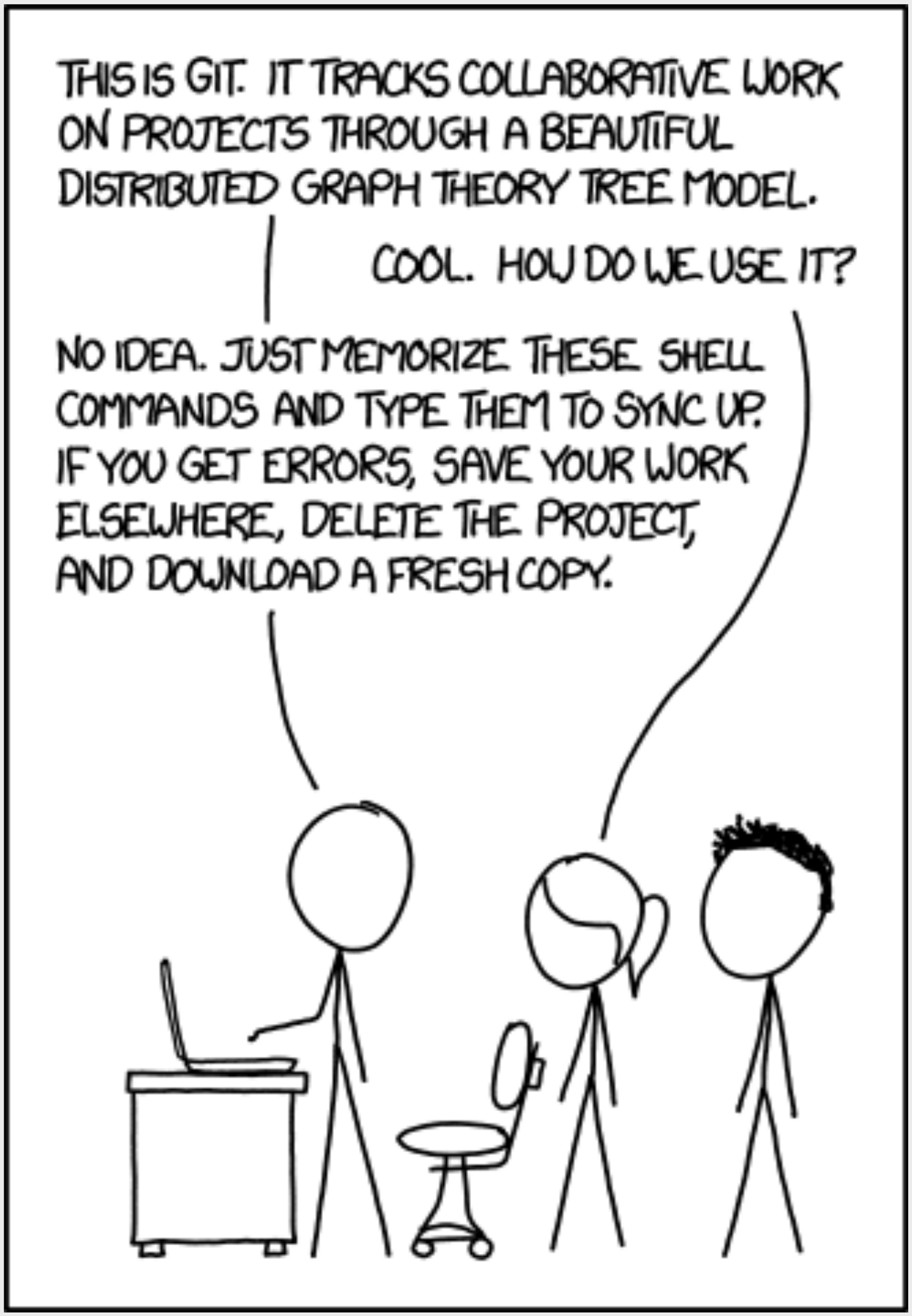
- To return to your session just login to `big_machine` again and do:

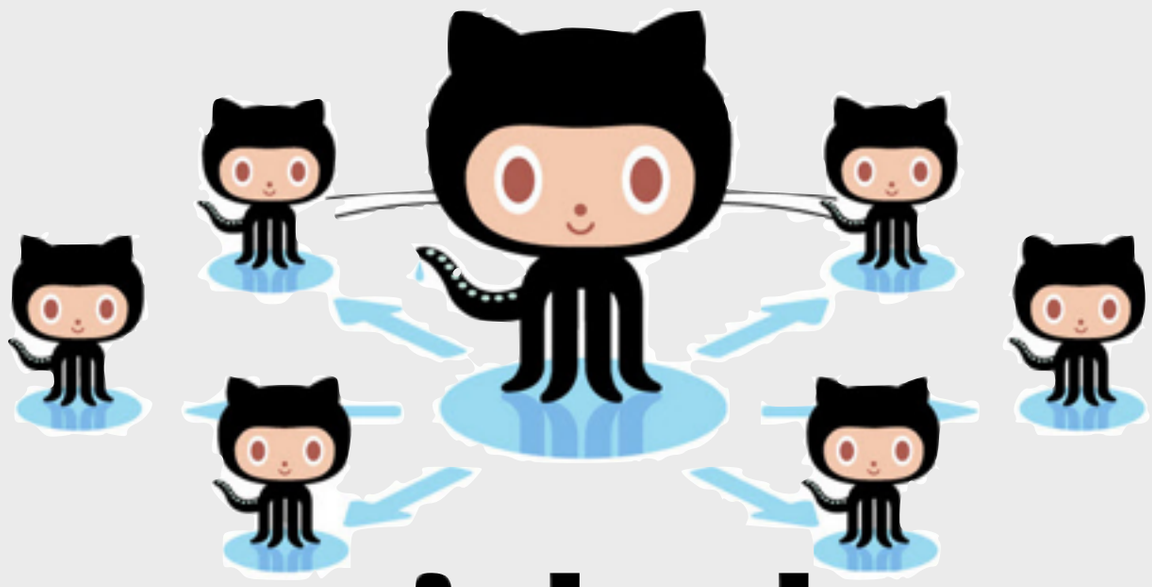
```
big_machine$ screen -r
screen_session$ ./my_slow_script.py
Result: 42
screen_session$ exit
[screen is terminating]
big_machine$
```

For more info check `man screen`

Also **do** create your own `.screenrc`

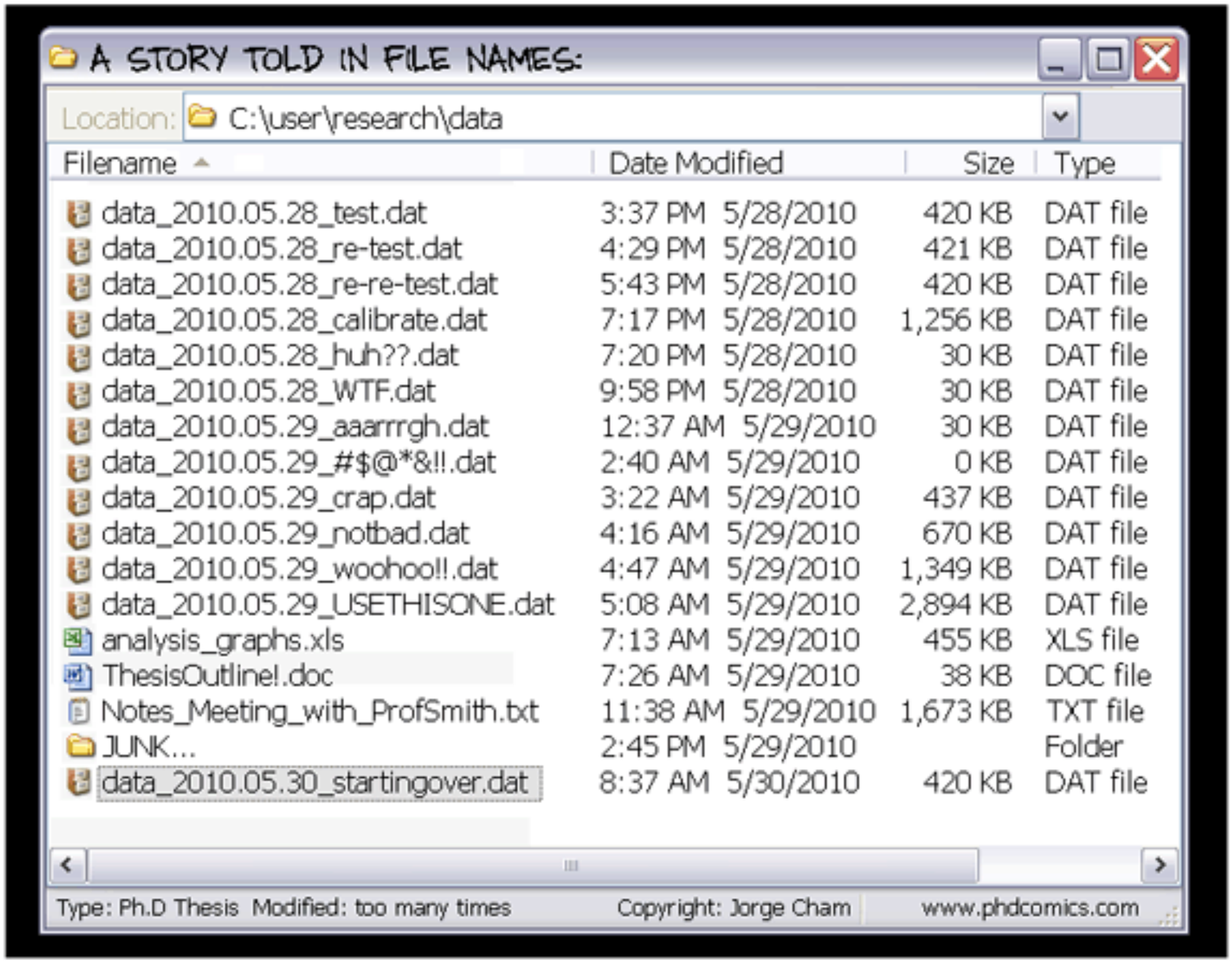
# An Introduction To Git



github
SOCIAL CODING

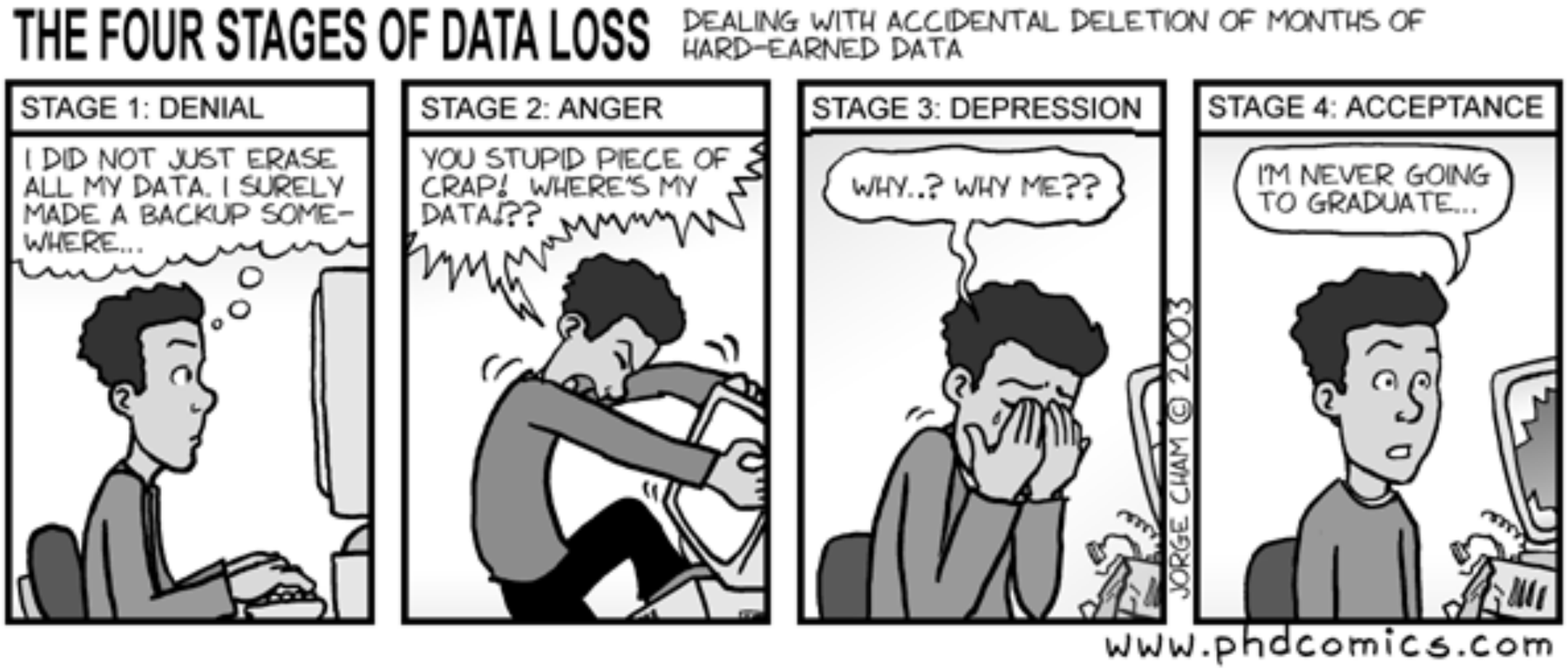
# Why Do I Need Version Control?




by Jorge Cham
www.phdcomics.com

# Why Do I Need Version Control?

- Your files are better organised
- You keep a history of all previous versions
- Your research is faster, more efficient and more reproducible
- Version control benefits collaborative work
- You always have a backup



# How Do I Use Git?


Creating a new project
```
$ git init
```

Cloning an existing project
```
$ git clone
https://github.com/…/project.git
```

Adding new files to be committed
```
$ git add README.md
```

Commit all new files
```
$ git commit -m “Useful message”
```

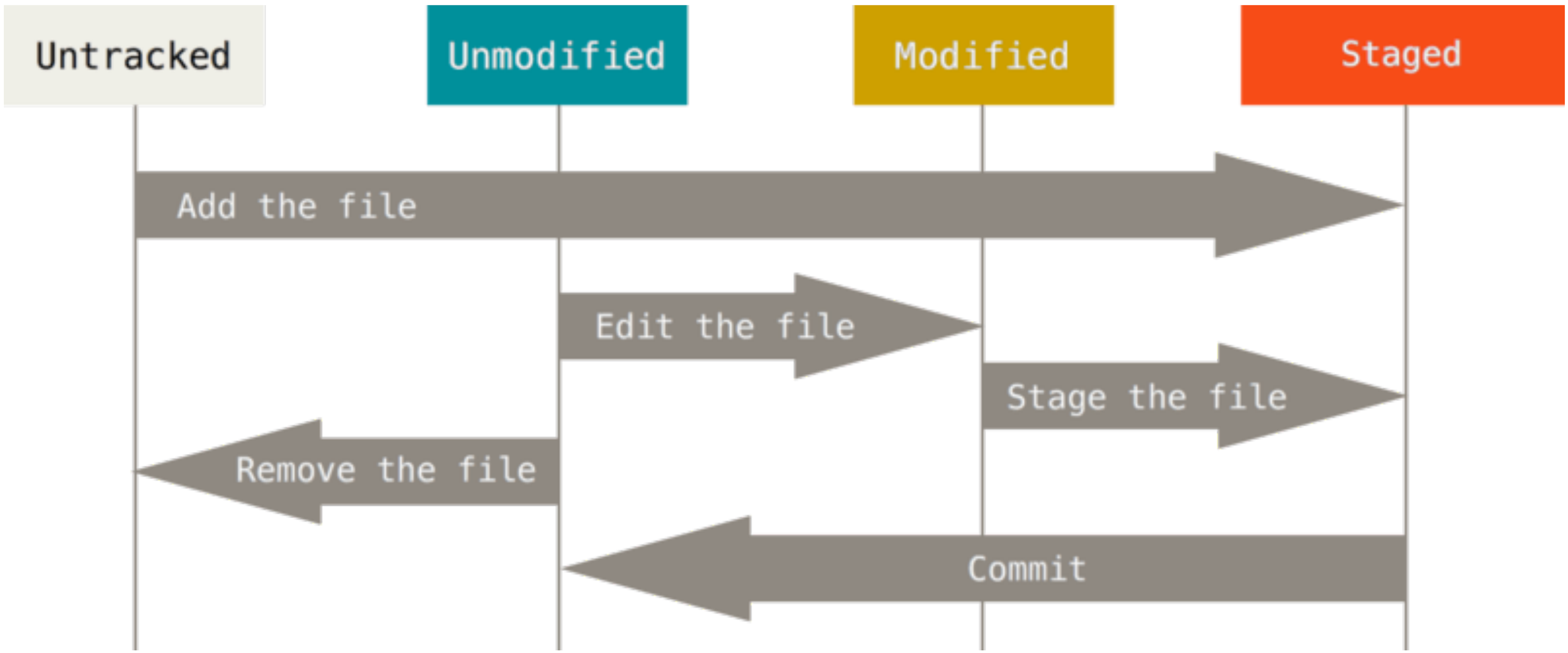
Updating the local copy (“pulling”)
```
$ git pull
```

Updating the remote (“pushing”)
```
$ git push
```

Current status of all files of repository
```
$ git status
```

Show the history (commit log)
```
$ git log
```

# The Lifecycle Of The Status Of Your Files




Pro Git Boot, by Scott Chacon: http://git-scm.com/book

# Setting Up Git

- Local configurations (only the current repository is affected)
  ```
  $ git config [options]
  ```
- Global configurations (only the user's configuration is modified)
  ```
  $ git config  --global [options]
  ```
- System configurations (all users are affected)
  ```
  $ git config  --system [options]
  ```
- Change your identity
  ```
  $ git config –-global user.name “Benedikt J. Daurer”
  $ git config –-global user.email “benedikt.daurer@icm.uu.se”
  ```
- Set your favourite editor (e.g. emacs or vim)
  ```
  $ git config –-global core.editor emacs
  ```
- Check your current settings
  ```
  $ git config –-list
  ```

# Demo: Basic Git Commands

# Getting Familiar With Basic Commands

1. Create a local copy (clone) of the following project:
   https://github.com/uu-python/participants

```
$ git clone https://github.com/uu-python/participants.git
```

2. Create a new file YOURNAME.md

```
$ $EDITOR filipe.md
```

3. Write something about yourself and add your file to the files tracked by git
4. Commit your changes and give a meaningful log message

```
$ git add filipe.md
$ git commit -m “Create new file”
```

5. Update your local repository by pulling from the remote

```
$ git pull
```

6. Update the remote repository by pushing your local changes

```
$ git push
```

# Deleting, Moving, Cancelling, Resetting

- Deleting a tracked file
  ```
  $ git rm FILE
  ```
- Deleting a tracked file (but keeping an untracked copy)
  ```
  $ git rm --cached FILE
  ```
- Moving a file (renaming)
  ```
  $ git mv FILE TARGET
  ```
- Unstaging a file
  ```
  $ git reset HEAD FILE
  ```
- Undo modifications of unstaged files
  ```
  $ git checkout -- FILE1 FILE2
  ```
- Checkout a previous version
  ```
  $ git checkout HASH
  ```

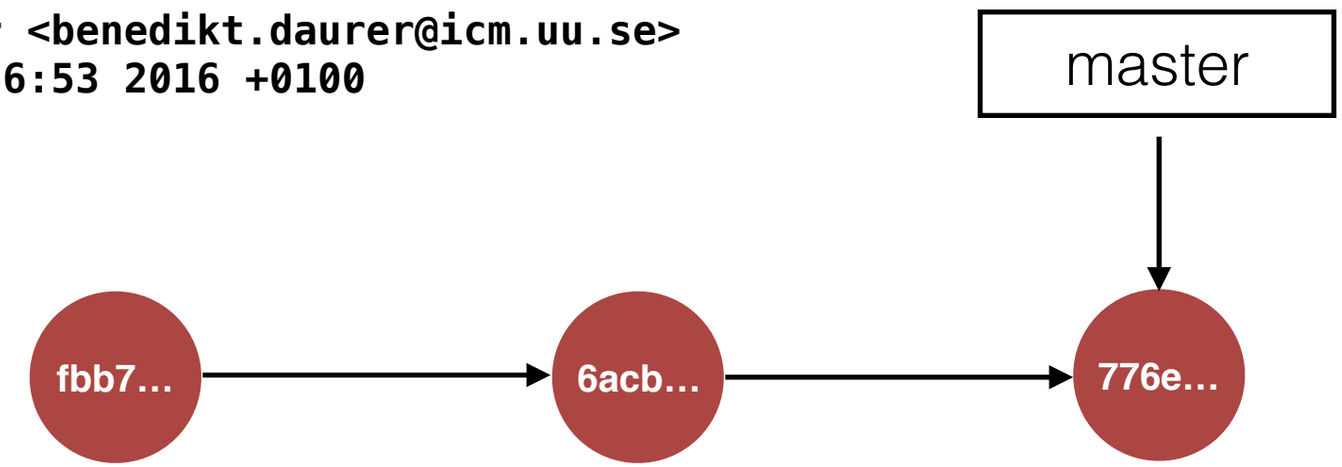
# Branching And Merging

## Check the history of the branch

```
benedikt@icm-241-135:~/particpants$ git log

commit 776e7c4f19493d88a85832dcef44ff2e569586bb
Author: Benedikt Daurer <benedikt.daurer@icm.uu.se>
Date:   Mon Nov 21 15:59:00 2016 +0100

    Fixed typo

commit 6acb9e49d9de91a8aa1536a659f3003f477e9c7d
Author: Benedikt Daurer <benedikt.daurer@icm.uu.se>
Date:   Mon Nov 21 15:57:53 2016 +0100

    Fixed link

commit fbb7ef51497d8aa77304280419c7cc4c67511626
Author: Benedikt Daurer <benedikt.daurer@icm.uu.se>
Date:   Mon Nov 21 15:56:53 2016 +0100

    Initial commit
```

## Check on the status of the branch

```
benedikt@icm-241-135:~/particpants$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```



# Branching And Merging

Create a new branch "testing"

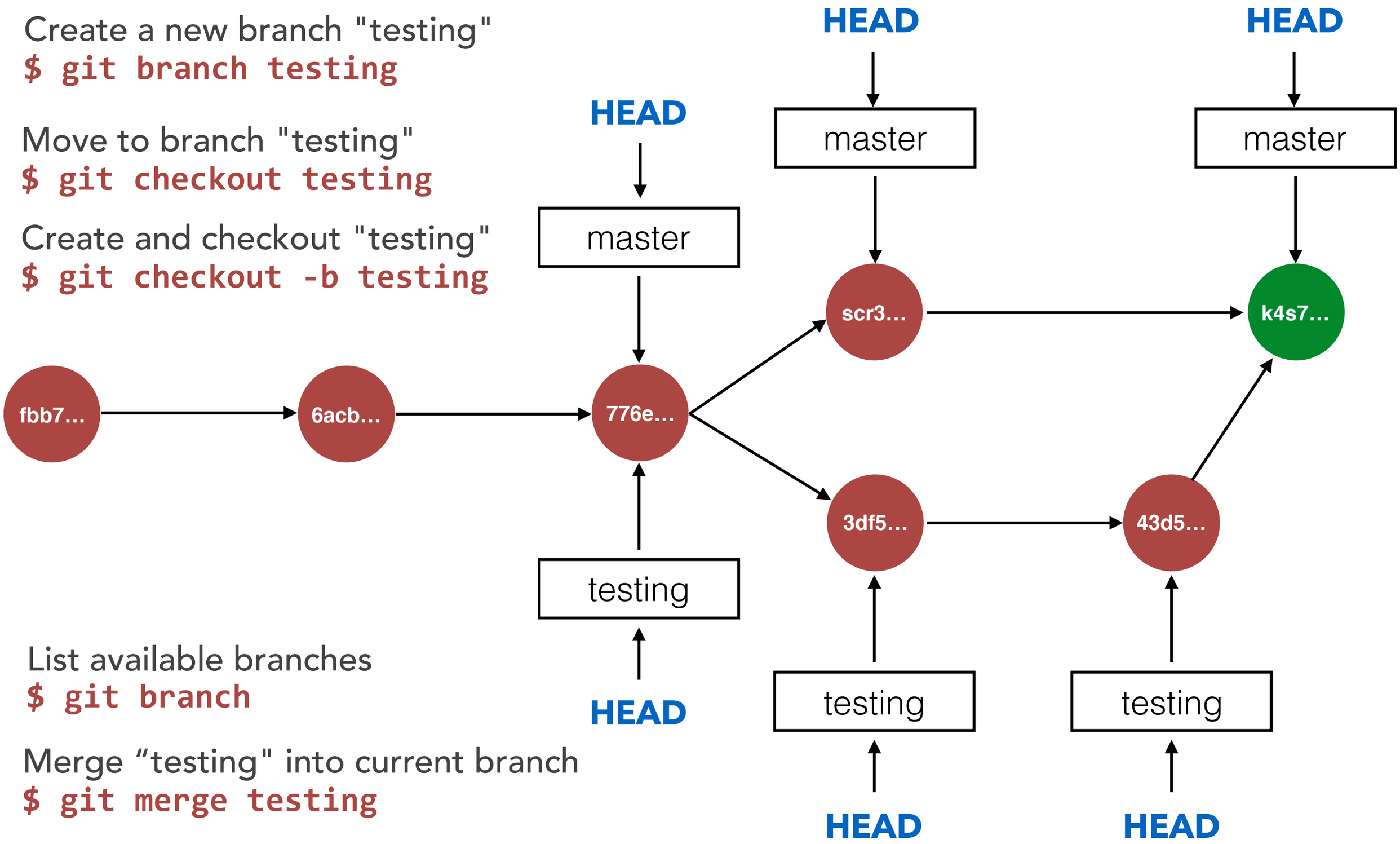
```
$ git branch testing
```

Move to branch "testing"

```
$ git checkout testing
```

Create and checkout "testing"

```
$ git checkout -b testing
```




List available branches

```
$ git branch
```

Merge “testing" into current branch

```
$ git merge testing
```

# Demo: Branching And Merging

# Contribute To A Collaborative Project

1. Create a local copy (clone) of the following project:
   https://github.com/uu-python/particpants

```
$ git clone https://github.com/uu-python/particpants
```

2. Create a new branch “yourname”

```
$ git checkout -b filipe
```

3. Edit the file of your neighbour or someone else (e.g. describe the person, write a message, ...)

4. Commit your changes and give a meaningful log message

```
$ git commit -m “Edited benedikt's file.”
```

5. Push your local to remote branch with the same name

```
$ git push --set-upstream origin filipe
```

6. Switch to the master branch and merge the branch “yourname" into master

```
$ git checkout master
$ git merge filipe
```

7. Update local master branch (pull)

```
$ git pull
```

8. Update remote master branch (push)

```
$ git push
```

# Take Home Messages

- Version control using git helps you better organize your work (e.g. code, documentation, manuscripts, thesis, webpages, ...)
- `git` (together with **github** or **bitbucket**) enables collaborative coding/writing
- You always have a backup and you can easily go back to previous versions

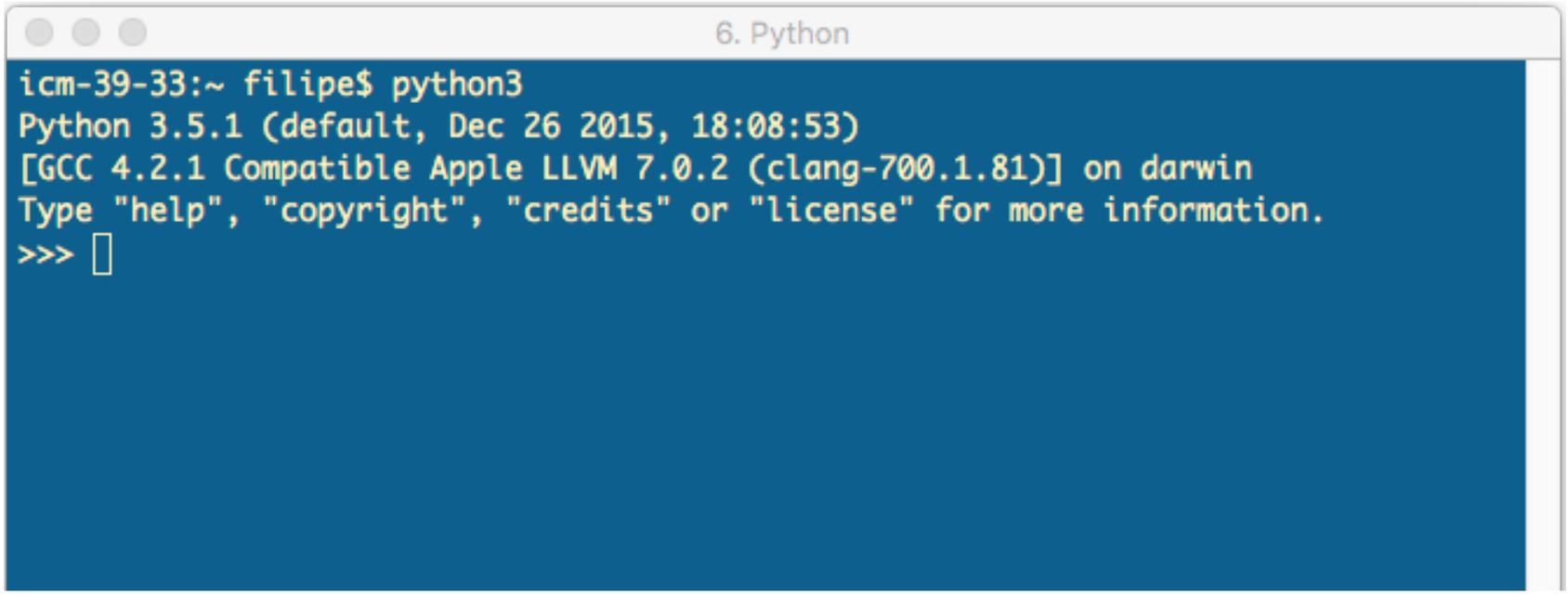
## Further Reading And Playing

- The Pro Git Book (https://git-scm.com/book/en/v2)
- Githug - a game for learning git (https://github.com/Gazler/githug)

# Interactive Python

# Standard Python Interpreter

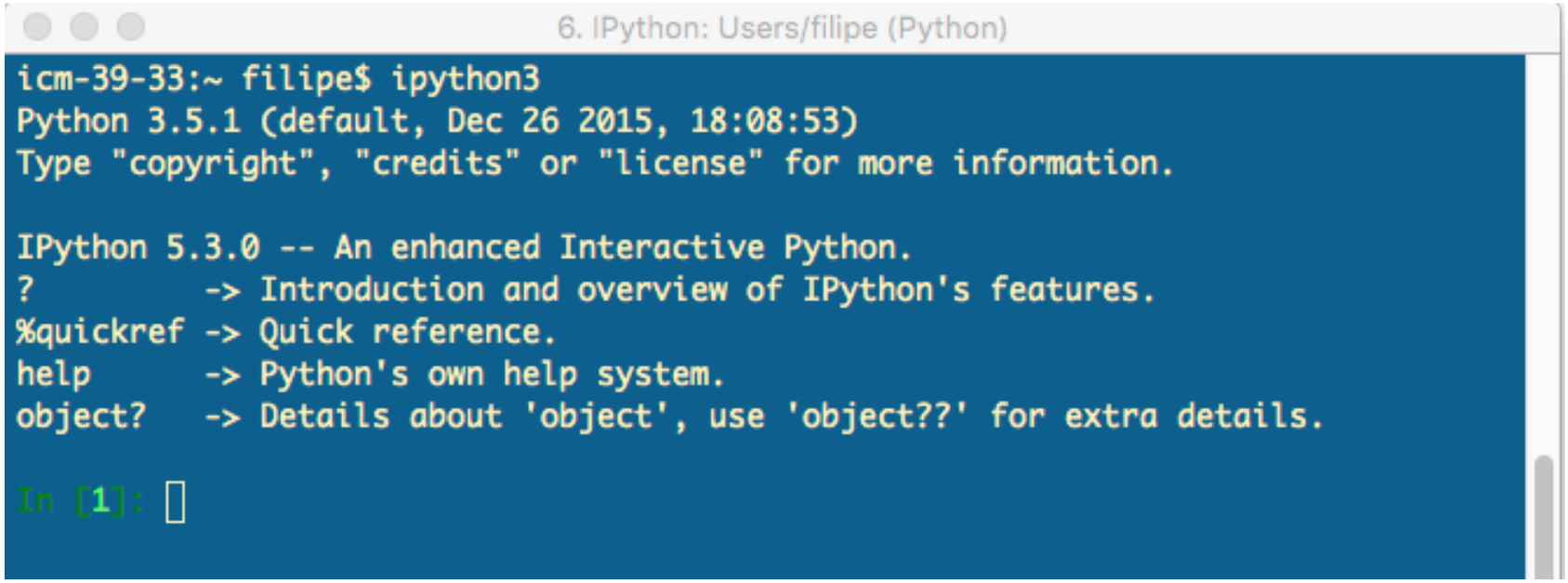
- The standard Python interpreter is `python`.
- For example, to run a script `my-program.py`:
  `$ python my-program.py`
- We can also start the interpreter by simply typing python at the command line, and interactively type Python code into the interpreter.




- The standard Python interpreter is not very convenient due to a number of limitations.

# IPython

- IPython addresses the limitation of the standard python interpreter
- A work-horse for scientific use of python. It provides an interactive prompt to the python interpreter with a greatly improved user-friendliness.




- It includes:
  Command history, using the up and down arrows.
  Tab auto-completion.
  In-line editing of code.
  Object introspection, and docstring extraction.
  Good interaction with operating system shell.

# Jupyter Notebook

- Open-source web application
- Allows you to create and share documents that contain live code, equations, visualisations and explanatory text
- Based on the IPython shell, but provides a cell-based environment with great interactivity
- Similar interface capabilities to Matlab.

- Calculations can be organised and documented in a structured way.
- Jupyter notebooks are usually run locally, from the same computer that run the browser.
- To start a new Jupiter notebook session, run the following command:
  `$ jupiter-notebook`

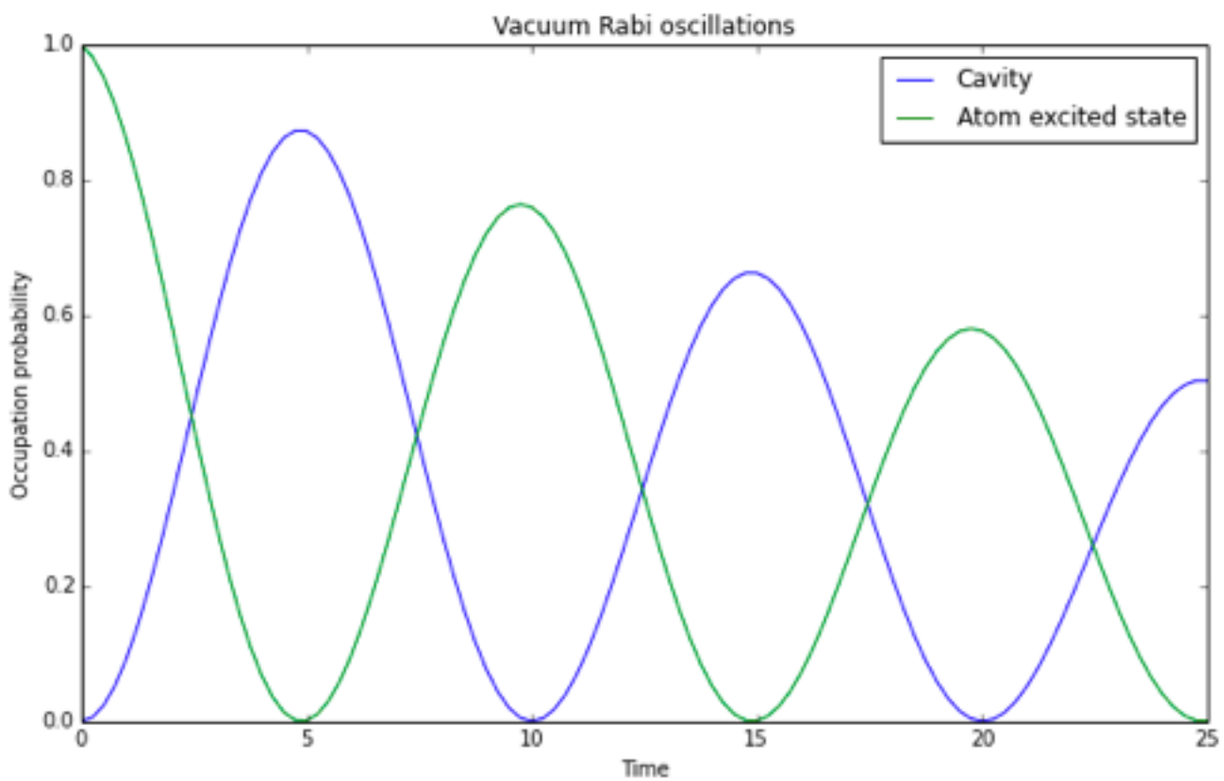
In [7]:
```
output = mesolve(H, psi0, tlist, c_ops, [a.dag() * a, sm.dag() * sm])
```

## Visualize the results

Here we plot the excitation probabilities of the cavity and the atom (these expectation values were calculated by the `mesolve` above). We can clearly see how energy is being coherently transferred back and forth between the cavity and the atom.

In [8]:
```
n_c = output.expect[0]
n_a = output.expect[1]

fig, axes = plt.subplots(1, 1, figsize=(10,6))

axes.plot(tlist, n_c, label="Cavity")
axes.plot(tlist, n_a, label="Atom excited state")
axes.legend(loc=0)
axes.set_xlabel('Time')
axes.set_ylabel('Occupation probability')
axes.set_title('Vacuum Rabi oscillations')
```

Out[8]: <matplotlib.text.Text at 0x7f8f0b8c3908>



# Demo: Jupyter Notebook