Introduction to version control (using Git)

Hans Fangohr

2019-06-17

Prerequisits for this session

- Have working `git` installation
- You can check for an existing installation using the command `git --version`, which should display some output similar to

  ```
  git version 2.22.0
  ```

- If this is not working, you need to install git (Download for example from `https://git-scm.com/downloads`)

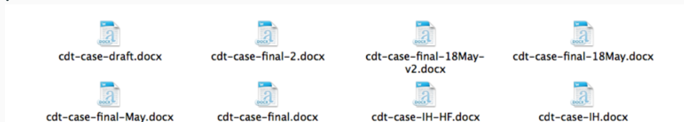
Why version control - Single user

- often we work on documents (or a set of files) for a long time (days, weeks, months, years)
- we (should) have backup copies in other places
- need to know:
    - regularly: what is the most recent version?
    - often: what where the changes introduced last?
    - sometimes: how did the project look like some time ago (say two weeks)

Why version control - Single user 2

- Common approaches:

```
file-1.doc
file-2.doc
file-2a.doc
file-3.doc
...
```



or or

```
gcm.f                          gcm2c-nonlin.f
gcm2.f                         gcm2c.f
gcm2b.f                        gcm2d-Dec2013.f
gcm2c-nonlin-NS.f              gcm2d.f
gcm2c-nonlin-after-AGM.f       gcm2e-Nov2013.f
gcm2f.f                        gcm-better.f
```

We can do much better using *version control* tools.

Why version control - in Team

- multiple people working on code
- may work on one file simultaneously
- need
    - tracking of versions
    - tracking of who did what change
    - merging of changes from different people

- Impossible to do manually -> Need version control tools

This session on Version Control and Git

- will introduce *idea* of version control together with
- *Git* which is a particular version control package
    - Homepage: `https://git-scm.com`

- we focus on the basics of the single user case

Terminology

Repository

something keeping track of all changes to the project for all the past (hidden in `.git`)

You can think of this as a (hidden) collection of the files `file-1.doc`, `file-2.doc`, `file-3.doc`, `file-4.doc`.

Working copy

the set of (visible) files (in the working directory), i.e. one copy of the project.

Typically, this will contain the most recent version of the file(s).

Getting started with git

- Suppose we need to write a Python program for a project called “project1“
- We create a directory `project1` and change into directory:

```
$ mkdir project1
$ cd project1
$
```

- At this stage, the directory is empty:

```
$ ls -l -a
total 0
drwxr-xr-x   2 fangohr  staff    64 Jun 15 20:25 .
drwx------+ 94 fangohr  staff  3008 Jun 15 20:25 ..
```

Initialise git repository

Need to do this only once for a given repository:

```
$ git init DIRECTORYNAME
```

Example

```
$ pwd
/Users/fangohr/Desktop/project1
$ git init .
Initialized empty Git repository in
/Users/fangohr/Desktop/project1/.git/
$ ls -la
total 0
drwxr-xr-x   3 fangohr  staff    96 Jun 15 20:32 .
drwx------+ 94 fangohr  staff  3008 Jun 15 20:25 ..
drwxr-xr-x   9 fangohr  staff   288 Jun 15 20:32 .git
```

Starting the project

- Suppose we create our first file `hello.py` in project1 directory:

```
def hello(msg):
    print(f"Hello world: {msg}")
```

- Example use

```
$ python
Python 3.6.7 [...]
>>> import hello
>>> hello.hello("a beautiful day")
Hello world: a beautiful day
```

Checking the status of files (`git status`)

- We can ask `git` about the `status` of our files in the project directory:

```
$ git status
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        hello.py

nothing added to commit but untracked files present
(use "git add" to track)
```

Checking the status of files (`git status`)

- Shorter output (one line per file): `git status -s`

  ```
  $ git status -s
  ?? test.txt
  ```

- `??` means ”untracked”

In-built help function

- Git has a fairly comprehensive help command:

```
$> git help
```

Or to get help for a particular command:

```
$ git help status
NAME
    git-status - Show the working tree status
SYNOPSIS
    git status [<options>...] [--] [<pathspec>...]

DESCRIPTION
    [ ... ]

OPTIONS
    -s, --short
        Give the output in the short-format.
```

First steps 1: Adding files (to staging area, `git add`)

- Add this file to the repository (i.e. tell `git` to track it):

```
$ git add hello.py
$
```

- At this stage, `git` knows that it should add the file. Let's check this (via the status command):

```
$ git status -s
A hello.py
```

  - `A` stands for Added (to the staging area)

First steps 1: Adding files - status

- `git status` produces a more verbose message than `git status -s`:

```
$ git status .
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   hello.py
```

First steps 2: Committing staged files (`git commit`)

- Commit changes (ask `git` to tag and record all changes in staging area):

```
$ git commit -m "first draft of hello-greeting function"
[master (root-commit) 6f2bb07] first draft of hello-greeting function
 1 file changed, 2 insertions(+)
 create mode 100644 hello.py
```

First steps 3: checking status (`git status`)

- Check status:

```
$ git status
On branch master
nothing to commit, working tree clean
```

- check status short:

```
$ git status -s
```

no news is good news, i.e. all files in the directory are up-to-date (=identical to last snap-shot)

- What file are in the directory?

```
$ ls -A -1
.git
hello.py
```

First steps 4: checking the history (`git log`)

- Study history of repository (the log):

```
$ git log
commit 6f2bb0750b84152f7ddb68bb8a5aff478e47ab92 (HEAD -> master)
Author: Hans Fangohr escape <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:14:45 2019 +0200

first draft of hello-greeting function
```

First steps 4: checking the history (`git log -p`)

- Show the history with the changes that have taken place (`-p` for `--patch`)

```
$ git log -p
commit 6f2bb0750b84152f7ddb68bb8a5aff478e47ab92 (HEAD -> master)
Author: Hans Fangohr escape <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:14:45 2019 +0200

    first draft of hello-greeting function

diff --git a/hello.py b/hello.py
new file mode 100644
index 0000000..60178c5
--- /dev/null
+++ b/hello.py
@@ -0,0 +1,2 @@
+def hello(msg):
+    print(f"Hello world: {msg}")
```

First steps 5: modifying the file

- extend programm `hello.py` to read:

```
def hello(msg):
    print(f"Hello world: {msg}")


if __name__ == "__main__":
    hello("from the main programme")
```

- Has `git` realised we have changed the file?:

```
$ $ git status -s
M hello.py
```

  - `M` stands for Modified

First steps 6: Review the changes (`git diff`)

- What is the `diff` erence (in comparison to the last snapshot):

```
$ git diff
diff --git a/hello.py b/hello.py
index 60178c5..564b54b 100644
--- a/hello.py
+++ b/hello.py
@@ -1,2 +1,5 @@
 def hello(msg):
     print(f"Hello world: {msg}")
+
+if __name__ == "__main__":
+    hello("from the main programme")
```

First steps 7: Stage and commit changes

- Suppose we are happy with this change, and want to take a snap-shot of the current files (i.e. *commit* the change):
    1. Stage the changes
    2. Commit the staged changes

```
$ git add hello.py
$ git commit -m "Added main program as demo"
```

First steps 8: The history (`git log`)

```
$ git log
commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749 (HEAD -> master)
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo

commit 6f2bb0750b84152f7ddb68bb8a5aff478e47ab92
Author: Hans Fangohr escape <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:14:45 2019 +0200

    first draft of hello-greeting function
(END)
```

First steps 8: The history (`git log -1`)

- Display only the last log entry:

```
$ git log -1
commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749 (HEAD -> master)
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo
```

First steps 8: The history (`git log -1 -p`)

- Display only the last log entry with the patch (=changes):

```
$ git log -1 -p
commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749 (HEAD -> master)
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo

diff --git a/hello.py b/hello.py
index 60178c5..564b54b 100644
--- a/hello.py
+++ b/hello.py
@@ -1,2 +1,5 @@
 def hello(msg):
     print(f"Hello world: {msg}")
+
+if __name__ == "__main__":
+    hello("from the main programme")
```

First steps 9: Adding another file

We create a new file `README.txt` which contains this line:

```
The hello.py program prints a friendly message.
```

Let's check the status, add, commit and check status:

```
$ git status -s
?? README.txt
$ git add README.txt
$ git status -s
A  README.txt
$ git commit -m "Adding README file"
[master 3dfabcf] Adding README file
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
$ git status -s
$
```

First steps 10: Study the history

- Using the `git log` command:

```
$ git log

commit 3dfabcff8e446cf24c884f3cd6fe2105bec7550e (HEAD -> master)
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:52:36 2019 +0200

    Adding README file

commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo

commit 6f2bb0750b84152f7ddb68bb8a5aff478e47ab92
Author: Hans Fangohr escape <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:14:45 2019 +0200

    first draft of hello-greeting function
(END)
```

Variations of `git log`

```
$ git log --oneline

3dfabcf (HEAD -> master) Adding README file
cf19828 Added main program as demo
6f2bb07 first draft of hello-greeting function
```

Variations of `git log`: `--stat`

```
$ git log --stat

commit 3dfabcff8e446cf24c884f3cd6fe2105bec7550e (HEAD -> master)
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:52:36 2019 +0200

    Adding README file

 README.txt | 1 +
 1 file changed, 1 insertion(+)

commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo

 hello.py | 3 +++
 1 file changed, 3 insertions(+)

commit 6f2bb0750b84152f7ddb68bb8a5aff478e47ab92
```

See which line was written when (and by whom!)

`git blame FILENAME`

```
$ git blame hello.py

^6f2bb07 (Hans Fangohr 2019-06-15 21:14:45 1) def hello(msg):
^6f2bb07 (Hans Fangohr 2019-06-15 21:14:45 2)    print(f"Hello world: {msg}")
cf19828a (Hans Fangohr 2019-06-15 21:45:02 3)
cf19828a (Hans Fangohr 2019-06-15 21:45:02 4) if __name__ == "__main__":
cf19828a (Hans Fangohr 2019-06-15 21:45:02 5)    hello("from the main programme")
```

Commit messages

Commit messages should be used to explain what has been changed in this commit, or what this commit provides / fixes / etc. For example:

- ”starting implementation of new feature X”
- ”as used for figure 3 in Nature paper”
- ”have added iterative method to solver suite”
- ”implemented suggestion from examiner in transfer viva”

The purpose of them is to (help) find a particular version of the code later. The messages are not limited to one line but can be very long if desired.

Typical cycle

While programming (or writing a report, creating a web page, etc), we tend to follow this cycle:

1. do the work (i.e. modify files)
2. commit changes with commit message
3. back to 1.

Only occasionally, we need to do special things:

- examine the history (partly shown)
- go back to an older snap shot (next topic -> "check out")

Going to particular commits (`git checkout`)

- the `checkout` command updates the files in the *working directory* (not the repository), and allows 'time travel'.
- Example:

```
$ ls
README.txt   hello.py
$ git log --oneline
3dfabcf (HEAD -> master) Adding README file
cf19828 Added main program as demo
6f2bb07 first draft of hello-greeting function
$ git checkout 6f2bb07
Note: checking out '6f2bb07'.
$ ls
hello.py
```

Going to the most recent commit (`git checkout`)

- If you have only one branch called `master` (this is the default), then to go to the most recent version in the repository use:

  ```
  $ git checkout master
  ```

- Otherwise, can use `git log --all` to see all revisions and choose first:

```
$ git log --oneline --all
3dfabcf (HEAD -> master) Adding README file
cf19828 Added main program as demo
6f2bb07 first draft of hello-greeting function
$ git checkout 3dfabcf
```

Which commit have we checked out?

- To know which commit we have checked out, we can use `git show HEAD`
- Example:

```
$ git checkout  cf19828a1e0a62
$ git show HEAD
commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749 (HEAD)
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo

diff --git a/hello.py b/hello.py
index 60178c5..564b54b 100644
--- a/hello.py
+++ b/hello.py
@@ -1,2 +1,5 @@
 def hello(msg):
     print(f"Hello world: {msg}")
+
+if __name__ == "__main__":
+    hello("from the main programme")
```

Show files at particular revision (`git show COMMIT:PATH`)

- to retrieve an older version of just one file, use `git show COMMIT:PATH`:
- Display available versions with hashes

```
$ git log --oneline
3dfabcf Adding README file
cf19828 Added main program as demo
6f2bb07 first draft of hello-greeting function
```

- Show `hello.py` for last commit:

```
$ git show 3dfabcf:hello.py

def hello(msg):
    print(f"Hello world: {msg}")

if __name__ == "__main__":
    hello("from the main programme")
```

Show files at particular revision (`git show COMMIT:PATH`) 2

- Show `hello.py` for first commit:

```
bash-3.2$ git show 6f2bb07:hello.py

def hello(msg):
    print(f"Hello world: {msg}")
```

- This command is useful to see (or retrieve and redirect into a file) an older version of one file.
- The `checkout` command will update all the files in the working repository and cannot operate on a single file.

What are revision specifiers?

- Revisions (=snap-shots=commits) are identified by
  - hashes (such as 7a6262cf0acf) and
  - user-defined tags
  - branch names (such as the default `master` branch)
- Hashes can be specified with less than 40 characters, as long as the sequence is unique for all hashes in the repository

Adding user-defined tags

- Version control is particularly important when maintaining software that is released to users (such as `v0.1`, `v0.2`, ...)
- Also useful if we need to remember particular revisions (such as `interim report`, `Masters thesis`, ...)

Git tags

- Tags are pointing to specific points in Git history
- Tagging is generally used to capture a point in history that is used for a marked version release (i.e. `v1.0`).
- An (annotated) tag can be created like this:

  ```
  $ git tag -a v1.4 -m "my version 1.4"
  ```

  where the tag is `v1.4` and the comment is =”my version 1.4”=
- To see a list of all tags, we can use `git tag`
- to checkout the version of a given, for example `v1.4`, we use `git checkout v1.4`
- Git provides ”tags” and ”annotated tags”.
  Recommendation: use ”annotated tags” as they carry more metadata.

Removing files from the repository (`git rm`)

- To remove a file from the repository (say `README.txt`), you can use:

```
$ git rm README.txt
```

- The status changes to Deleted:

```
$ git status -s
D  README.txt
```

- The following `git commit` actually commits the staged removal of the file from the working directory.

```
$ git commit -m "README.txt was not needed after all"
[master 1f8153e] README.txt was not needed after all
1 file changed, 1 deletion(-)
delete mode 100644 README.txt
```

- Note that the history of the file is not changed: we can always go back to earlier revisions and the file will be there.

Renaming files and directories (`git mv`)

- You can rename a file with `git`. Suppose we need to rename `README.txt` to `readme.org`:

  ```
  $ git mv README.txt README.org
  ```

  This (i) renames the file in the working directory and (ii) stages the change for the next commit.

  ```
  $ git mv README.txt readme.org
  $ git status -s
  R  README.txt -> readme.org
  ```

  In the output, `R` stands for Renamed.
- the command `git mv` comes from `move` which is a different term used for `rename` files.

Where is all the history stored?

- The whole repository lives in a hidden directory with name `.git`. (it is hidden due to the leading dot. On Linux/Unix/Mac OS X, you can display hidden directories with the `-a` switch:

```
$project1> ls -a
.git        hello.py   README.txt
```

- git manages its own (file-based) data base in this subdirecotry
- we should not normally change this (apart from config files)

Reverting changes

- Suppose you are working on file `hello.py`. You start with the most
  You make some changes to `hello.py`, and then realise that your changes were not useful, and you want to go back the last version from the repository.
- We can use `git checkout hello.py` to check out `hello.py` from the repository and replace our modified copy on the disk with it
- Can also be used if we deleted the file by accident.

Uncommitting the commit

- If you have committed something that you didn't want to commit, you can undo the last commit (but only the last commit!) with:

```
$project1> git reset HEAD~
```

- This will not change the file in the working directory (i.e. it still contains the changes).
- use `git checkout FILENAME` to revert the file back to the last committed version.

Referencing commits with HEAD

- `HEAD` is pointing to the commit we last checked out
  - can see the commit using `git show HEAD`
  - or (in file `.git/HEAD`)
- We can use `HEAD~` to refer to the commit before `HEAD`

```
$ git checkout 3dfabcf
$ git log --oneline
3dfabcf (HEAD) Adding README file
cf19828 Added main program as demo
6f2bb07 first draft of hello-greeting function
$ git show -s HEAD~
commit cf19828a1e0a628fd8bc581a52cb3b03a5a6e749
Author: Hans Fangohr grey laptop <hans.fangohr@xfel.eu>
Date:   Sat Jun 15 21:45:02 2019 +0200

    Added main program as demo
```

- `HEAD~2` is two commits before `HEAD`

Making copies (backups?) of the repository

You can either

- just copy the whole `project1` folder (which includes the hidden `.git` directory) to another place
- or (recommended): you can ask `git` to make the copy:

```
$project1> cd ..
$> git clone project1 project1-copy
```

Now you have an identical copy of the repository in directory `project1-copy`.

-> See also using "multiple repositories" (pull, push, merge)

Multiple repositories

- When working with several people, one can have multiple repositories in different places:
  - Developer A may be working on the graphical user interface while developer B is improving the numerical part of the code.
  - In a group project student A might be working on the introduction, student B on a results chapter and student C on the appendix of a large (LaTeX) document.
- Occasionally (maybe quite frequently), the changes in these repositories (or some of these) need to be combined (*merged*)
- We will cover this advanced topic only superficially for one particular setup (with one central master repository).

  There are many other ways in which (distributed) version control systems can be used.

Multiple repositories: one master repository 1 (`git push`)

1. Create the master repository, say:

```
$ mkdir master
$ git init master
```

2. Add any files that you have already, and commit.
3. Now individuals can clone from the master to carry out their work:

```
$> git clone master my-copy-A
$> cd my-copy-A
```

- Do the work here, and and modify files, commit as many times as you like (may need *pull*; see below).
- When you have completed your work, commit everything and
- `push` your changes to the master (may need `pull` first):

```
$my-copy-A> git push
```

Multiple repositories: one master repository 2 (`git pull`)

To import changes from the master repository (others could have done some work in the mean time) into `my-copy-A`, use the `pull` command:

```
$my-copy-A> git pull
```

If there have been changes on files that we have worked on as well, a ”merge” of the changes has to take place

- This is usually automatic when we use `git pull`
- Sometimes, one needs to edit a file manually during the merge. In that case `git` reports:

  ```
  CONFLICT (content): Merge conflict in hello.py
  Automatic merge failed; fix conflicts and then commit the result.
  ```
- After such a manual merge, we need to commit, typically like this:

  ```
  $my-copy-A> git add hello.py
  $my-copy-A> git commit -m "manual merge"
  ```
- Once we have merged, we may want to `git push` the merged version back to the master repository.

Location of multiple repositories, and github et al

The *cloning*, *pull* ing and *push* ing between repositories can happen:

- on the same file system/computer, where multiple repositories are hosted in different subdirectories (see example above)
- between computers using either
  - ssh or
  - the web server
- github, bitbucket, gitlab and others offer webhosting of repositories.
  - typically free for open source projects

Scientific truth and *reproducibility*

- we tend to believe the 'results' or 'findings' of a research group (or individual), if other individuals and groups can *reproduce* them
- this implies that individuals should be able to reproduce all of *their own* results

Use version control in computational science and data science

When using computer simulations or data processing, we should use version control to be able to reproduce any earlier results at any point in the future.

Summary of some important commands

`git add` (add files)

`git commit` (commit changes)

`git status` (show modified/missing/added/removed files)

`git log` (show log)

`git diff` (show diff to version of working directory)

`git show HEAD` (show last commit of working directory)

`git checkout master` (update working directory)

`git show COMMIT:FILE` (send version of file to stdout)

`git clone` (copy repository)

`git tag -a` (add tag)

`git tag` (list tags)

Configure git - author name

- the (user) configuration of git is in the `~/.gitconfig` file
- can edit this with text editor (respecting the syntax of the file)

```
$ cat ~/.gitconfig
[user]
        name = Hans Fangohr <escape>
        email = hans.fangohr@xfel.eu
```

- or use git to read and change entries:

```
$ git config --global user.name
Hans Fangohr <escape>
$ git config --global user.name "Hans Fangohr <grey laptop>"
$ git config --global user.name
Hans Fangohr <grey laptop>
```

What about Mercurial (hg)?

In a first approximation, git and mercurial provide very similar functionality.

- See Mercurial Tutorial: `http://hgbook.red-bean.com/`
- and other links on `http://www.selenic.com/mercurial`

Further reading

- Git book: `https://git-scm.com/book`
  - also German translation
- Various (subjective) summaries an tutorials, for example
  - `http://www.yolinux.com/TUTORIALS/Git-commands.html`
  - `https://education.github.com/git-cheat-sheet-education.pdf`
- Stackexchange
- Google

Further features

Useful topics to learn about

- cloud based storage of repositories (`https://github.com`, `https://bitbucket.org`)
- `.gitignore` – ignoring files in directories
- branches – allowing multiple lines of development in the same repository
- stashing – temporary place for hiding ongoing work
- `git add -p` – adding only some changes in a file to a commit
- if you use Emacs, must checkout `magit`.
- `~/.gitconfig`

Summary

- Nowadays, version control software (such as `git` and `hg`) is
  - easy to install
  - very easy to use (in single user mode), but sufficiently sophisticated to support very complex projects
- An effective way to:
  - keep track of the history of a project,
  - reliably retrieve earlier versions if required
  - recover from errors (such as accidental deletion of files, inability to retrieve working version)
  - always find the most recent version of a document and
  - have (versioned) backups