GIT Fundamentals

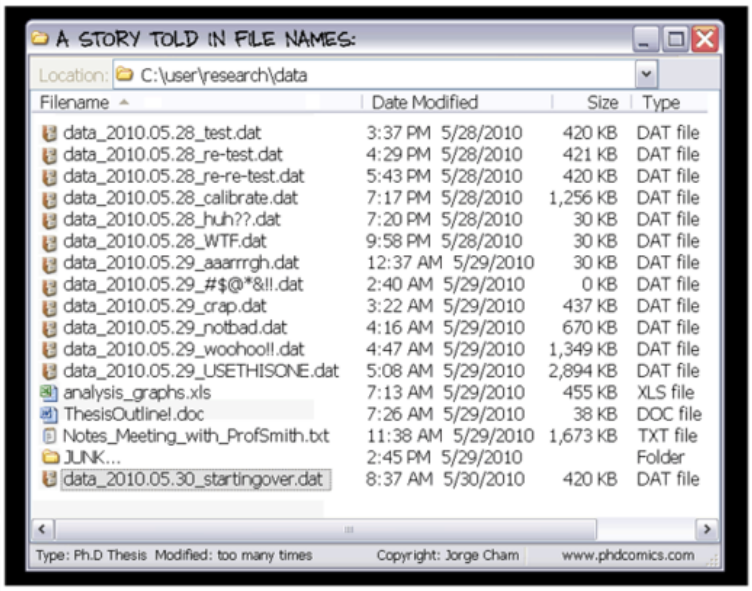
Holger Dinkel

Nov 09, 2016

Contents

Chapter 1

The Benefits of Version Control



Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. The benefits are at hand:

- **Track incremental backups and recover:** Every document can be backed up automatically and restored at a second's notice.
- **Track every change:** Every infinitesimal change can be recorded and can be used to revert a file to an earlier state.
- **Track writing experiments:** Writing experiments can be sandboxed to copies while keeping the main file intact.
- **Track co-authoring and collaboration:** Teams can work independently on their own files, but merge them into a latest common revision.
- **Track individual contributions:** Good VCS systems tag changes with authors who make them.

1.1 git at a Glance

1.1.1 git commands

The git tool has many subcommands that can be invoked like *git <subcommand>* for instance *git status* to get the status of a repository.

The most important ones (and hence the ones we'll be focusing on) are:

init: initialize a repository

clone: clone a repository

status: get information about a repository

log: view the history and commit messages of the repository

add: add a file to the staging area

commit: commit your changes to your **local** repository

push: push changes to a **remote** repository

pull: pull changes from a **remote** repository

checkout: retrieve a specific version of a file

you can read more about each command by invoking the help:

```
git commit --help
git help commit
```

1.1.2 git concepts

commit

A commit is a recorded set of changes in your project's file(s). Try to group *logical* sets of changes together into one commit – don't mix changes which are unrelated.

repository

A repository is the history of all your project's commits.

working copy

A working copy is a *local* version of the files of a repository. It is the set of files you are *working* on; they become part of a repository by *commiting* them.

1.2 git settings

1.2.1 Setting your identity

Before we start, we should set the user name and e-mail address. This is important because every git commit uses this information and it's also incredibly useful when looking at the history and commit log:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@embl.de
```

Other useful settings include your favorite editor, enabling color output as well as difftool:

```
git config --global core.editor nano
git config --global color.ui auto
git config --global merge.tool kdiff3
```

1.2.2 Checking Your Settings

You can use the *git config –list* command to list all your settings:

```
git config --list
user.name="John Doe"
user.email=johndoe@embl.de
core.editor=vim
merge.tool=meld
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Chapter 2

A Typical git Workflow

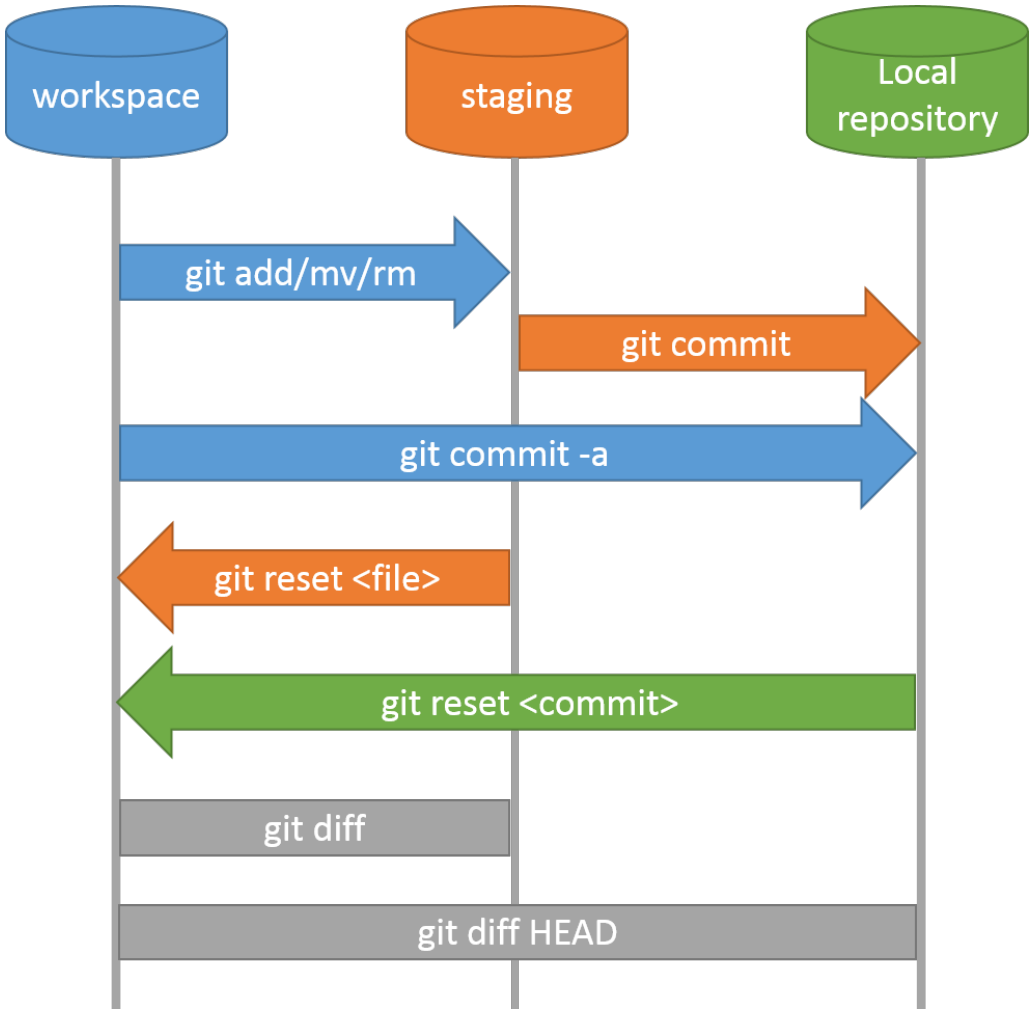


Fig. 2.1: Files are *added* from the *workspace*, which always holds the current version of your files, to the *staging area*. *Staged* files will be stored into the local repository in the next *commit*. The repository itself contains all previous versions of all files ever committed.

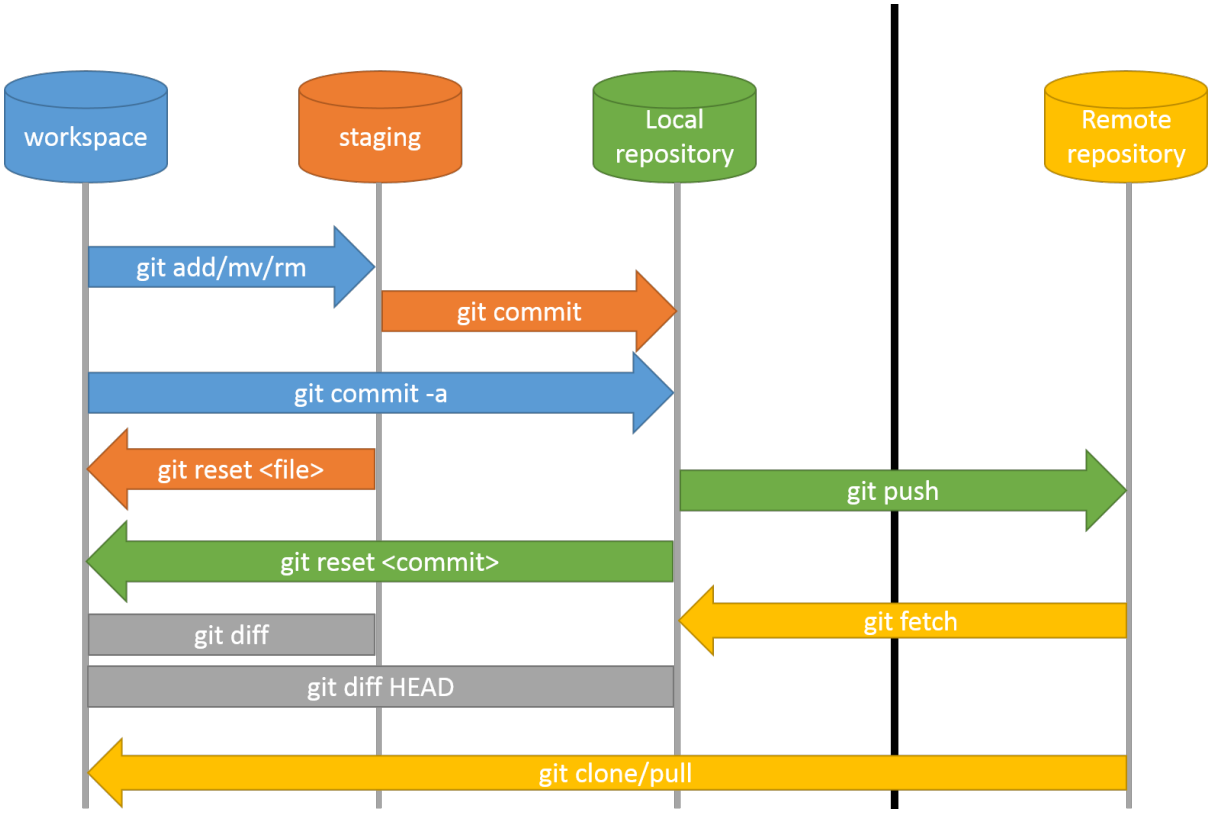


Fig. 2.2: Distributed workflow using a centralized repository. Here, you use *push* and *pull* to synchronize your local repository with a remote repository.

2.1 Creating a git Repository

Turning an existing directory into a local git repository is as simple as changing into that directory and invoking *git init*. However, here we want to create one repository which we can use from multiple other folders to sync to/from, therefore in this case, we need to initialize it as a *bare* repository.

Note: Normally you do not need the *–bare*, but it's essential for this exercise...

So, here we first create an empty directory in our homedirectory called *repos* (this is meant to hold and serve all our repositories), and create a repository in there called *mythesis*:

```
mkdir ~/repos
cd ~/repos
mkdir mythesis
cd mythesis
git init --bare
```

Note: As a result, you should have the directory *~/repos/mythesis* and there should be a directory called *.git* in this directory...

2.2 Cloning a git Repository

Next, we can *clone* this repository into the *~/Documents/mythesis* folder.:

```
cd ~/Documents

git clone ~/repos/mythesis

Initialized empty Git repository in /localhome/training/Desktop/mythesis/.
 ↪git/
warning: You appear to have cloned an empty repository.

cd mythesis
```

By *cloning*, we not only get the exact copy as the remote side, but we automatically tell git where we had got the data from, which allows us later to sync our changes back...

Note: You can clone from either a different folder on our computer, a remote machine (via ssh), or a dedicated git server:

Local directory:

```
git clone ~/repos/mythesis
```

Remote directory:

```
git clone ssh://remote_user@remote_server/mythesis.git
```

Remote git server:

```
git clone git@server:user/project
```

2.3 Checking the Status

If you don't know in which state the current repository is in, it's always a good idea to check:

```
git status

# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

Here, everything is clear, not much going on (no news is good news).

Note: In fact, it's good practice, to use *git status* as often as possible!

2.4 Adding files

First, we'll create a new file:

```
echo "My first line towards a great paper!" > paper.txt

git status

# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       paper.txt
nothing added to commit but untracked files present (use "git add" to␣
↪track)
```

Here, git tells us that there is a file, however it's *untracked*, meaning git does not know/care about it. We need to tell git first that it should keep track of it. So we'll add this file to the so called *staging area*:

```
git add paper.txt

git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   paper.txt
#
```

This tells us that the *paper.txt* has been added and can be committed to the repository.

2.5 Committing changes

It might be a bit confusing at first to find out that *git add* does **not** add a file to the repository. You need to *commit* the file/changes to do that:

```
git commit -m "message describing the changes you made"
```

Note: You **MUST** provide a commit message! git will ignore your attempt to commit if the message is empty. If you do not provide the *-m* parameter, git will open an editor in which you should write your commit message (can be multiple lines of text). Once you save/quit your editor, git will continue to commit...

After succesfully committing, we can check the status again:

```
git status

# On branch master
nothing to commit, working directory clean
```

2.6 Viewing the History

You can use *git log* to view the history of a repository. All previous commits including details such as Name & Email-address of the committer, Date & Time of the commit as well as the actual commit message are shown:

```
git log

commit  <some hash value identifying this commit>
Author: <your name and email address>
Date:   <the actual date of the commit>

message describing the changes you made
```

2.6.1 Exercise

Repeat the add/commit procedures you just learned. Add more files, use an editor to add more content to the *paper.txt* file, commit your changes providing a meaningful commit message.

2.7 Pushing changes

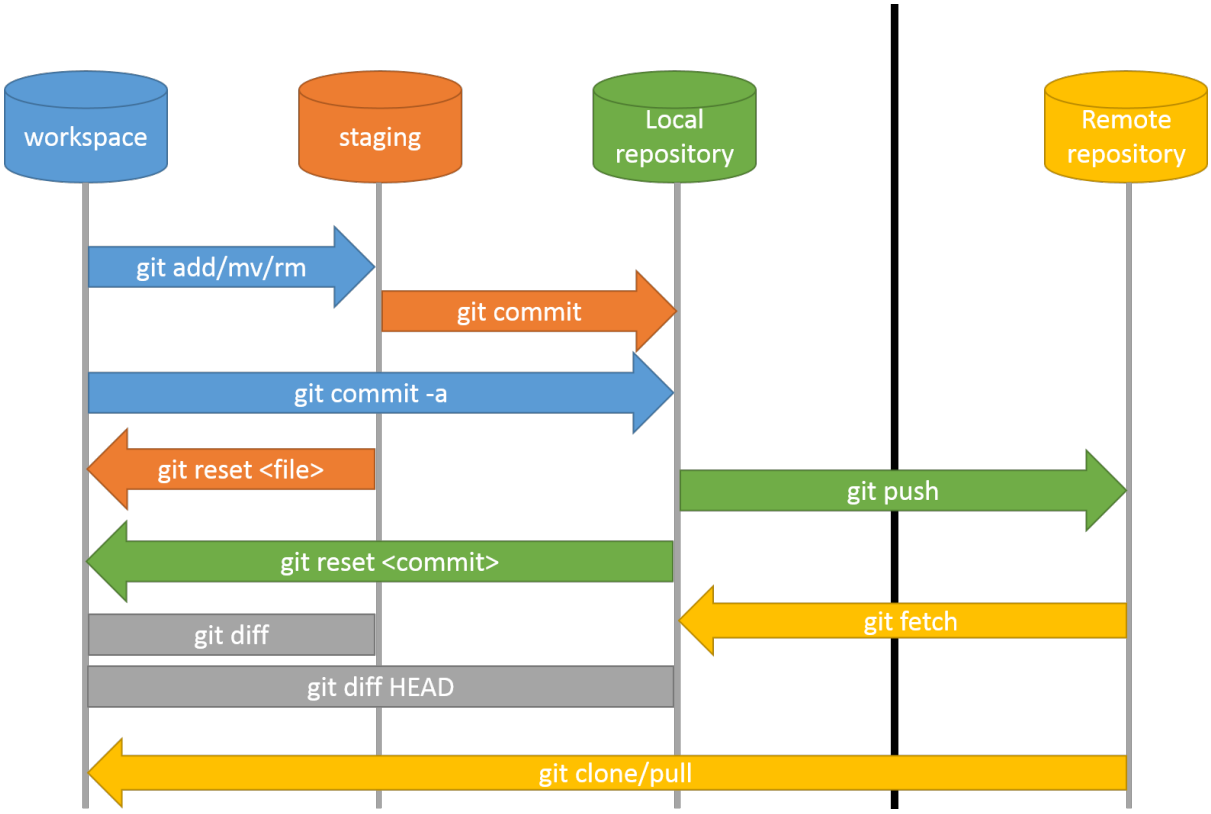
In order to exchange/synchronize your changes with a remote repository, you use *git push*/*git pull*:

To push all committed changes, simply type:

```
git push
```

Note: git "knows" from which location you had cloned this repository and will try to push to exactly that location (using the protocol you used to clone: ssh, git, etc)...

Warning: If you get a warning message, read it carefully! The most common error you get when trying to push are changes on the remote end which you first need to merge into your local repository before you are allowed to push your own...



2.7.1 Creating a second clone

In order to simulate contrubting to our repository from another computer, we will again checkout the repository, but this time in a different folder named *mythesis-work*:

```
cd ~/Documents

git clone ~/repos/mythesis mythesis-work

cd ~/Documents/mythesis-work
```

This repository should contain all the changes you've pushed so far. Now we want to improve our *paper.txt* document. Use an editor to add more lines to this file:

```
echo "This line was contributed from work..." >> paper.txt
```

Again, *add*, *commit*, and *push* your changes.

2.8 Pulling changes

To update your local repository with changes from others, you need to *pull* these changes. In a centralized workflow you actually **must** pull changes that other people have contributed, before you can submit your own.

```
git pull
```

Warning: Ideally, changes from others don't conflict with yours, but whenever someone else has edited the same lines in the same files as you, you will receive an error message about a **merge conflict**. You will need to resolve this conflict manually, then add each resolved file (*git add*) and commit.

So we go back to the directory *~/Documents/mythesis* and (after checking the status) try to get the changes we've done in the *mythesis-work* directory:

```
cd ~/Documents/mythesis

git status

git pull
...
Auto-merging paper.txt
CONFLICT (content): Merge conflict in paper.txt
Automatic merge failed; fix conflicts and then commit the result.
```

2.9 Solving conflicts

When working collaboratively on a project, it is unavoidable that the same file gets changed by different contributors. This causes a conflict and needs to be dealt with.

Hint: It helps minimizing conflicts if you push/pull often!

To solve a merge conflict, you can either:

- manually merge the two files (see below)
- discard the remote file: *git checkout –ours conflicted_file.txt*
- discard the local file: *git checkout –theirs conflicted_file.txt*

2.9.1 Manually merging a conflict

To create a conflict, we change the same line in the file *paper.txt* in both directories (*mythesis* and *mythesis-work*) without pulling each others changes in between. Once we pull, git will tell us that a conflict has occurred.:

```
Automatic merge failed; fix conflicts and then commit the result.
```

When git encounters conflicts in files, it adds special markers <<<<<<<, =======, >>>>>>> into this file wrapping both conflicting changes. It is up to you to decide which of these changes to keep.:

```
...
content of the file
...
<<<<<<< HEAD:paper.txt
```

```
your home changes
=======
your changes introduced at work
>>>>>>> 0000000000000000000000000000000000000000:paper.txt
...
rest of the file
...
```

Make sure to delete the lines that where introduced by git (otherwise you won't be able to commit changes. If you only wanted to keep your changes than you would delete everything except your changes:

```
...
content of the file
...
your home changes
...
rest of the file
...
```

Now, you need to add this file again to the staging area and commit to finish this conflicting merge. Use *git status* to see the status of the repository.

2.10 Undo local changes

One of the great features of using version control is that you can revert (undo) changes easily. If you want to undo all changes in a local file, you simply checkout the latest version of this file:

```
git checkout -- <filename>
```

Warning: You will loose all changes you made since the last commit!

If you want to checkout a specific version (revision) of a file, you need to specify the hash or name of the revision:

```
git checkout revision_name <filename>
```

2.11 Branching

Branches are deliberately lightweight in git, meaning you can very easily create (and delete) branches. So it's generally a good idea to use different branches...

To list all existing branches, you type::

```
git branch -a
```

To create a new branch you can type either::

```
git branch NewBranch
git checkout NewBranch
```

or, simpler::

```
git checkout -b NewBranch
```

Note: Remember you need to switch to new branches, by using `git checkout branchname`! Any new commit will now go into this branch, not affecting any other branch.

In order to take your changes from one branch into another, you need to merge the branches::

```
git merge master
```

You always merge **into** the currently checked out branch. In this example we merge the branch *master* into *NewBranch*. This will update *NewBranch* with all changes that have been committed to *master*. Since *NewBranch* is now up to date, we can also merge back it's changes into *master*::

```
git checkout master
git merge NewBranch
```

Finally, we can delete this branch::

```
git branch -d NewBranch
```

Chapter 3

Acknowledgements

3.1 Contributors

We are very grateful to the contributors who helped create and improve this document:

- Holger Dinkel
- Grischa Toedt
- Frank Thommen
- Toby Hodges
- Jean-Karim Hériché

3.2 Hardware/Software

The git server *riva* at fli is kindly maintained by Bernd Senf.

3.3 Copyright Material

- FLI Logo © FLI
- EMBL Logo © EMBL Heidelberg
- “A story told in filenames”: PhD Comics
- “Git local workflow”: Software Carpentry/Research Bazaar
- “Git remote workflow”: Software Carpentry/Research Bazaar
- “FINAL.DOC”: PhD Comics

3.4 License

Chapter 4

Links/References

the **git** program itself:

git for Windows [1], or for Mac [2]

Tools:

- SourceTree (a graphical user interface for git) [3]
- DiffMerge (a graphical merge tool) [4]
- Kdiff3 (another graphical merge tool) [5]
- githug - a game to learn git [6]

References:

- Try Git [7]
- A Visual Git Reference [8]
- A visual guide to version control [9]
- Version control for scientific research [10]
- Software Carpentry's introduction to git [11]

Scientific Articles About Git:

- Git can facilitate greater reproducibility & increased transparency in science [12]
- Improving the reuse of computational models through version control [13]

[1] http://www.git-scm.com/download/win
[2] http://www.git-scm.com/download/mac
[3] http://www.sourcetreeapp.com/download/
[4] http://www.sourcegear.com/diffmerge/
[5] http://kdiff3.sourceforge.net/
[6] https://github.com/gazler/githug
[7] http://try.github.io/levels/1/challenges/1
[8] http://marklodato.github.io/visual-git-guide/index-en.html
[9] http://betterexplained.com/articles/a-visual-guide-to-version-control
[10] http://blogs.biomedcentral.com/bmcblog/2013/02/28/version-control-for-scientific-research/
[11] https://github.com/swcarpentry/bc/blob/master/intermediate/git/01-conversational-git.md
[12] http://www.ncbi.nlm.nih.gov/pubmed/23448176
[13] http://www.ncbi.nlm.nih.gov/pubmed/23335018

GitHub
GIT CHEAT SHEET

V 1.1.1

Git is the open source distributed version control system that facilitates GitHub activities on your laptop or desktop. This cheat sheet summarizes commonly used Git command line instructions for quick reference.

INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Git for All Platforms
http://git-scm.com

CONFIGURE TOOLING

Configure user information for all local repositories

```
$ git config --global user.name "[name]"
```
Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```
Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```
Enables helpful colorization of command line output

CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

```
$ git init [project-name]
```
Creates a new local repository with the specified name

```
$ git clone [url]
```
Downloads a project and its entire version history

MAKE CHANGES

Review edits and craft a commit transaction

```
$ git status
```
Lists all new or modified files to be committed

```
$ git diff
```
Shows file differences not yet staged

```
$ git add [file]
```
Snapshots the file in preparation for versioning

```
$ git diff --staged
```
Shows file differences between staging and the last file version

```
$ git reset [file]
```
Unstages the file, but preserve its contents

```
$ git commit -m "[descriptive message]"
```
Records file snapshots permanently in version history

GROUP CHANGES

Name a series of commits and combine completed efforts

```
$ git branch
```
Lists all local branches in the current repository

```
$ git branch [branch-name]
```
Creates a new branch

```
$ git checkout [branch-name]
```
Switches to the specified branch and updates the working directory

```
$ git merge [branch]
```
Combines the specified branch's history into the current branch

```
$ git branch -d [branch-name]
```
Deletes the specified branch

GIT CHEAT SHEET

REFACTOR FILENAMES
Relocate and remove versioned files

```
$ git rm [file]
```
Deletes the file from the working directory and stages the deletion

```
$ git rm --cached [file]
```
Removes the file from version control but preserves the file locally

```
$ git mv [file-original] [file-renamed]
```
Changes the file name and prepares it for commit

SUPPRESS TRACKING
Exclude temporary files and paths

```
*.log
build/
temp-*
```
A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

```
$ git ls-files --other --ignored --exclude-standard
```
Lists all ignored files in this project

SAVE FRAGMENTS
Shelve and restore incomplete changes

```
$ git stash
```
Temporarily stores all modified tracked files

```
$ git stash pop
```
Restores the most recently stashed files

```
$ git stash list
```
Lists all stashed changesets

```
$ git stash drop
```
Discards the most recently stashed changeset

REVIEW HISTORY
Browse and inspect the evolution of project files

```
$ git log
```
Lists version history for the current branch

```
$ git log --follow [file]
```
Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```
Shows content differences between two branches

```
$ git show [commit]
```
Outputs metadata and content changes of the specified commit

REDO COMMITS
Erase mistakes and craft replacement history

```
$ git reset [commit]
```
Undoes all commits after `[commit]`, preserving changes locally

```
$ git reset --hard [commit]
```
Discards all history and changes back to the specified commit

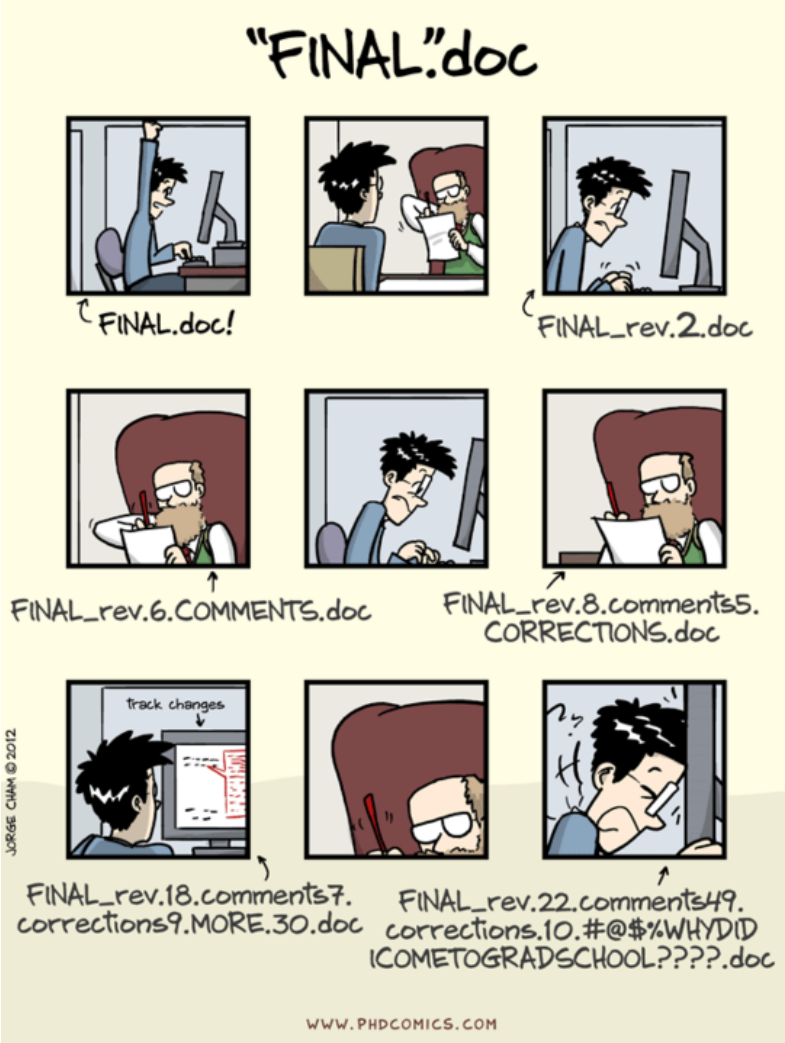
SYNCHRONIZE CHANGES
Register a repository bookmark and exchange version history

```
$ git fetch [bookmark]
```
Downloads all history from the repository bookmark

```
$ git merge [bookmark]/[branch]
```
Combines bookmark's branch into current local branch

```
$ git push [alias] [branch]
```
Uploads all local branch commits to GitHub

```
$ git pull
```
Downloads bookmark history and incorporates changes


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM