

gist / ˈdʒɪst/ *n* [AF, it lies, fr. *gisir* to lie, ultim. fr. L *jacere* — more at ADJACENT] (ca. 1711) **1** : the ground of a legal action **2** : the main point or part : ESSENCE <the ~ of an argument>

¹**git** /'git/ *n* [var. of *get*, term of abuse, fr. ²*get*] (1929) *Brit* : a foolish or worthless person

²**git** *dial* var of GET

git-go var of GET-GO

git·tern /'gi-tərn/ *n* [ME *giterne*, fr. MF *guiterne*, modif. of OSp *guitarra* guitar] (14c) : a medieval guitar

git essentials

Luca Formaggia

2024

A distributed VCS: git

Git is a *distributed* version control system (VCS) with an emphasis on speed. It was initially designed and developed by Linus Torvalds for Linux kernel development. Is now used by thousands of software developers worldwide and has in fact replaced previous VCSs.

Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server.

In git (almost) every command operates just on the **local** repository (the one you have on your PC). **You do not even need a remote repository to use git!**

What should git keep track of?

You will put in the git repo only the main files (sources, headers...), *i.e.* **files that are not created from other files.**

You should not put in the repo object files, executables, temporary files etc. etc., *i.e.* files that are in fact the product of a process (compilation, execution etc.).

You will see that there is a way to tell git to completely ignore certain files in your working area and reduce the risk of committing them by mistakes.

A snapshot of a git repository (using gitk)

The screenshot shows the gitk graphical user interface. At the top, a commit graph displays the history: a red dot for 'Local uncommitted changes, not checked in to index', followed by a yellow dot for 'master' with the message 'New examples', then two blue dots for 'Updated first lecture' and 'I forgot a file...', and finally an orange dot for 'remotes/bitbucket/master' with the message 'Fixed submodule Matelial'. The 'remotes/bitbucket/master' commit is selected, showing its SHA1 ID as 0000000000000000000000000000000000. Below the graph, a search bar and a commit list are visible. The commit list shows four entries by 'formaggia' with timestamps: 2019-01-20 23:19:00, 2019-01-20 23:18:44, 2019-01-18 13:49:51, and 2019-01-18 12:38:44. The main area displays the commit details for the selected commit, including the author 'Antonio Cervone and Luca Formaggia', the title 'An (maybe not so) gentle introduction to git', and the date '2019 V3'. The file 'GeneralCourses/git_intro/git_intro.tex' is highlighted, showing its index 590a06e..35354f9 100644. The file content is displayed in a diff view, showing the changes between the parent and the current commit.

File Edit View Help

Local uncommitted changes, not checked in to index

master New examples

Updated first lecture

I forgot a file...

remotes/bitbucket/master Fixed submodule Matelial

SHA1 ID: 0000000000000000000000000000000000

Find commit containing: Exact All fields

Search

Diff Old version New version Lines of context: 5 Ignore space change Line diff

Author:

Committer:

Parent: 40902a6fa69dd759a92bba60bec3c8e5ef043cab (New examples)

Branch:

Follows:

Precedes:

Local uncommitted changes, not checked in to index

----- GeneralCourses/git_intro/git_intro.tex -----

index 590a06e..35354f9 100644

@@ -20,11 +20,11 @@

%%%

\author{Antonio Cervone and Luca Formaggia}

\title{An (maybe not so) gentle introduction to git}

-\date{2017 V2}

+\date{2019 V3}

Patch Tree

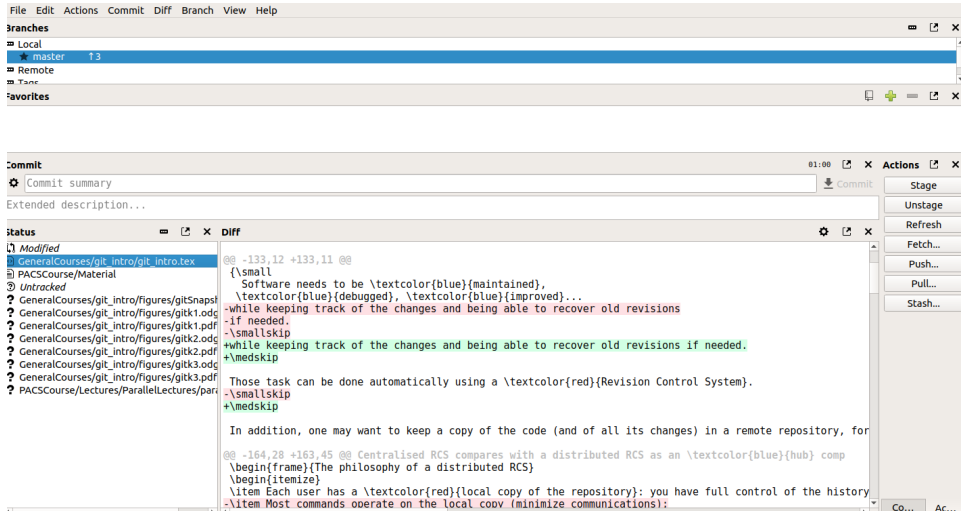
Comments

GeneralCourses/git_intro/git_intro.tex

PACSCourse/Material

Graphical Interfaces

There are plenty of graphical interfaces for git. I sometimes use **git-cola**, but you may find many others on the web. A list may be found **here**.



Useful references

On the web you can find plenty of resources on git!

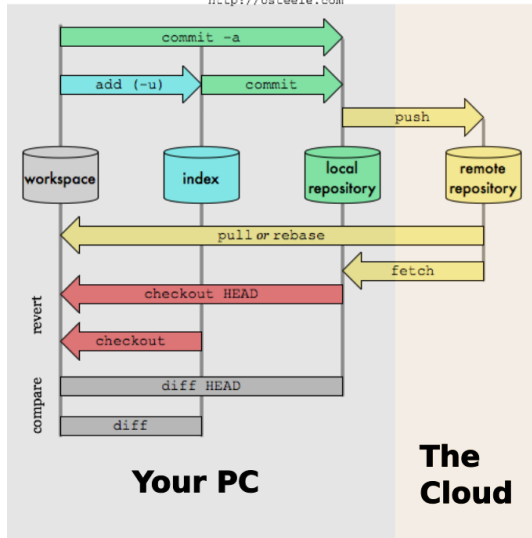
- ▶ Official documentation: <http://git-scm.com/docs>
- ▶ Official “Pro Git” book: <http://git-scm.com/book/>
- ▶ GitHub tutorials: <https://training.github.com/>,
<https://guides.github.com/>
- ▶ Useful Git patterns for real life:
<https://happygitwithr.com/workflows-intro.html>
- ▶ “Oh Sh*t, Git!?”: <https://ohshitgit.com/>
(without swears: <https://dangitgit.com/>), with the fantastic [zine](#).

And.... chatGPT is a git guru!

A visual overview of a typical git workflow

Git Data Transport Commands

<http://osteele.com>



Creating a new git repo

There are two possible ways to create a git repository:

1. You create the repository locally on your computer, and then, **if you wish**, you make a remote copy on the cloud, for instance on GitHub.
2. You create the repository on a remote site like GitHub, and then you **clone** it on your computer (or you work directly on GitHub, but less flexible)

In this course we will first consider the first option.

A few important terms of the git workspace

- **Working Directory**. The folders containing all files of your code. The root folder contains a hidden directory called `.git` with the local repo. **Don't touch it if you do not know what you are doing.** Files in the workspace may be **tracked** by git or **untracked**, or **ignored**.
- **Commit**. A snapshot of your project at a certain point in time. It records the changes you have made and allows you to revert to previous states if needed. Commits are uniquely identified by a SHA key.
- **Staging Area (Index)**. Before changes in the working directory are committed to the repository, they must be staged. The staging area contains the modified files you are ready to insert in the next commit.
- **Checkout**. The operations that extract a single file or an entire commit from the repo.
- **HEAD**. A "pointer" to the last checked out commit.
- **Branch**. A movable pointer to a commit that allows for parallel development and experimentation without affecting the main codebase. The default branch is normally called **master** or **main**.

First thing to do

If you have never used git on a computer you should first **let git know who you are** (so people can blame you for your wrongdoings or cherish you for your programming skill!)

```
$ git config --global user.name "Micky Mouse"
```

```
$ git config --global user.email "micky.mouse@gmail.com"
```

git config tells git to change some git configuration, the option global makes it to change parameters that applies to all git repos in your computer. Another thing you may want to change is the default editor

```
$ git config --global core.editor "gedit"
```

Need help?

Git has an integrated help. If you need help for the command `command` just do

```
$ git help command
```

Git uses a lot of terms that may confuse you at the beginning (and not only at the beginning, ...). A useful command is

```
$ git help glossary
```

You find more on the [Git book](#).

More help?

Other useful commands:

```
$ git help tutorial
```

A tutorial

```
$ git help tutorial-2
```

The second part of the tutorial.

Git is a very complete (and complex) version control system, **don't be scared**: in fact you will normally need just a few commands.

master or main, or something else

The default branch in git used to be called **master**. Recently, people objected that it is not a good name since the "master branch" is not mastering anything, it is just the branch you have chosen as your main branch. So, now GitHub names as `main` the default branch (unless you choose otherwise).

You can change how git names the default branch when creating a repo:

```
$ git config -global init.defaultBranch <name>
```

Create/clone a repo

Create a brand new local repository. In the directory where you want to create it type:

```
$ git init
```

Often you want to clone from an existing remote repository (on [GitHub](#) in this example). We will see that later on.
If you type

```
$ git status
```

you get something like

```
On branch master  
No commits yet  
nothing to commit (create/copy files and use "git add" to track)
```

Read git messages: they are useful!

Create a new file

Create a new file called `file1.cpp`
Show status of the repo:

```
$ git status
```

The output is:

```
On branch master.  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
    file1.cpp  
nothing added to commit but untracked files present  
(use "git add" to track)
```

Add a new file to the git index

The file `file1.cpp` is at the moment still **unknown to git** (it's a new file!). It is in the so called **untracked** state.

We need to tell git that we want to take track of it but putting it on stage:

```
$ git add file1.cpp
```

show again status of the repo

```
On branch master.  
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
new file:   file1.cpp
```

The file is ready to be committed in your (local) git repository!

The first commit

Now the file `file.cpp` is in the **staging area (index)**, ready to be committed in the repository. Maybe we may want to add other files to the commit, or maybe we just want to add just that file to the repository. In the latter case we just do

```
$ git commit
```

An editor will appear asking for a commit message. The message is compulsory. We can commit into the repository a newly created file without adding it beforehand, by using

```
$ git commit <file>
```

You can commit by giving a short message directly:

```
$ git commit -m "This is my first file"
```

The status after the commit

```
$ git commit -m "This is my first file"
```

```
[master aba508f] This is my first file  
1 file changed, 5 insertions(+)  
create mode 100644 file1.cpp
```

And if you do `git status`

```
On branch master  
(use "git push" to publish your local commits)  
nothing to commit, working tree clean
```

What does it mean?

The first message says that in the **branch** master (the only one we have at the moment) we have created a commit.

A commit a snapshot of your workspace at the moment of the commit, and is identified by an **hash key**, a long hexadecimal number whose first significant digits are shown (aba508f), and are normally enough to **identify the commit uniquely**.

You can at any time retrieve the situation of your files (those under git control) at a given commit using its hash key.

The message also says that in that commit you have added a file (the mode indicates just the file permission), which consists of 5 lines.

To add all modified file and commit in just one command:

```
$ git commit -a -m "A nice message"
```

What to put in a full commit message

If you do not use the `-m` option an editor is launched. You can put

A first line with a brief description (it is the one that is shown by some git commands)

Possibly followed by an empty line and a detailed description (possibly wrap lines to 72 characters)

Paragraphs are separated by empty lines.

- you may create lists using the `-` character

Modifying a file

Now we **modify** the content of file1.cpp

```
$ git status
On branch master Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified:   file1.cpp
no changes added to commit (use "git add" and/or "git commit -a")
```

If you want to commit the changes:

```
git add file1.cpp
git commit -m"a modification"
```

of, in just one command

```
git commit -a -m"a modification"
```

Going on the cloud: free remote repositories

There are many websites providing a nice place for a `git` repository and the possibility of sharing it with others, such as

- ▶ [GitHub](#)
- ▶ [Bitbucket](#)
- ▶ [GitLab](#)
- ▶ ...

Usually, with an educational/professional account, you can unlock extra features, such as unlimited private repositories

Create/clone a repo

If you want to use a remote repository there are two ways:

- 1) You already have a local repository. Then, you create an **empty remote repository** and add it to the **git remotes**
 - 2) You start by creating the remote repo on the cloud and then you **clone it**
- Let's first see the first option, since we have now a local repository.

Create an empty repo on GitHub

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).


Required fields are marked with an asterisk ().*

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner *

 lformaggia ▾

Repository name *

GitLesson

✔ GitLesson is available.

Great repository names are short and memorable. Need inspiration? How about **supreme-succotash** ?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Do not add files

Just follow the instructions

Quick setup — if you've done this kind of thing before

HTTPS

SSH

git@github.com:lformaggia/GitLesson.git

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository

...or create a new repository on the command line

```
echo "# GitLesson" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M master
git remote add origin git@github.com:lformaggia/GitLesson.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin git@github.com:lformaggia/GitLesson.git
git branch -M master
git push -u origin master
```

Explanation

```
$ git remote add origin git@github.com:lformaggia/GitLesson.git
```

adds a remote to your **local** git repo

```
$ git branch -M master
```

indicates the name of the default branch

```
$ git push -u origin master
```

copies the local repo to the origin remote and tells the remote to track the branch master. Now your branch master (the default one) is connected with the branch master of your remote repository on GitHub **and you have a remote branch remote/origin/master in your local repo.**

The message

```
Enumerating objects: 3, done.  
Counting objects: 100Writing objects: 100Total 3 (delta 0), reused 0 (delta 0),  
pack-reused 0  
To github.com:lformaggia/GitLesson.git  
* [new branch] master -> master  
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

The message that git writes on the terminal tells you that some object have been sent and regularly received, and that the branch master is now tracked on the remote repository.

How does git operates with the remote

Git allow you to do most operation **locally** Therefore it keeps in what are called **remote branches** a **copy** of the remote repository at the time of the last pull or fetch. You can check:

```
$ git branch -av
* master faa4c11 My first commit
remotes/origin/master faa4c11 My first commit
```

`git branch -av` lists all branches, local and remote alike, with some verbosity. The asterisk indicates the branch we are on (not many choices at the moment).

```
$ git status
On branch master
Your branch is up to date with 'origin/master'
...
```

`git status` tells that the local branch is aligned with the remote local branch.

If you have started from github

You need to clone from the existing remote repo:

With **HTTPS protocol** (simpler but more limited):

```
$ git clone https://github.com/username/mycode.git
```

With **SSH protocol** (you must have your public SSH key associated to your GitHub account):

```
$ git clone username@github.com:/mycode.git
```

Commands that communicate with a remote repository

Here you have the main three commands used by git to interact with a remote repository:

- ▶ **push** Send all committed changes to the remote.
- ▶ **pull** Get the changes from the remote and **merge** them with the local branch.
- ▶ **fetch** Get the changes from the remote, store them in the **local remote branch**.
You will need to merge manually.

In our example, `git pull` is equivalent to

```
$ git fetch
$ git merge /remote/origin/master
```

Using `fetch` you can have a look at the changes before merging them.

Interacting with the remote

Let's modify file1.cpp and add a new file file2.cpp to our repository

```
$ git add file2.cpp  
$ git commit -a -m"a modification and a new file"
```

Then,

```
$ git status  
On branch master  
Your branch is ahead of 'origin/master' by 2 commits.  
(use "git push" to publish your local commits)  
nothing to commit, working tree clean
```

I am **ahead** of the remote. Indeed the modifications are only local. I need to **push** them.

Interacting with the remote

```
$ git push
Enumerating objects: 8, done.
Counting objects: 100Delta compression using up to 8 threads
Compressing objects: 100Writing objects: 100Total 6 (delta 0), reused 0 (delta 0),
pack-reused 0
To github.com:lformaggia/GitLesson.git
faa4c11..2ffb31e master -> master
```

Then,

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

I just need `git push` and not the more complete form `git push origin master` (`git push <remote> <branch>`) since branch master is already tracked.

Push not possible because remote has been updated by others

It may happen that the push is blocked by git because one of your collaborators has pushed her commit to the remote. You receive a message like error: failed to push some refs.

You need to **pull** the changes from the repo. There are two strategies: merge and rebase. You can set the default behavior with

```
$ git config pull.rebase false
```

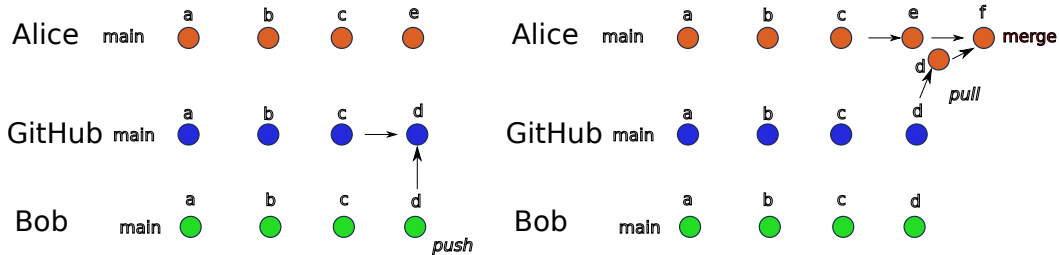
if you want merge to be the default, or

```
$ git config pull.rebase true
```

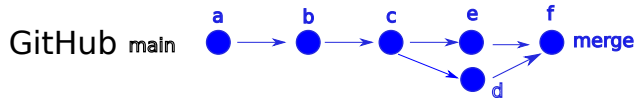
if you prefer rebasing. The difference are illustrated in the next slides.

Merge strategy

With the merge strategy the head commit in the remote, updated by Bob, is fetched and merged with the local head commit of Alice, the result is a commit with two parents: a **merge commit**

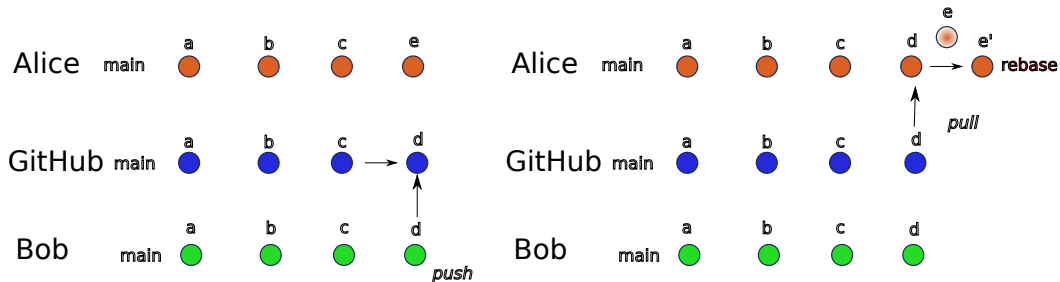


When Alice pushes to the remote the final situation on the remote is



Rebase strategy

With the merge strategy the commits created by Alice are "payed back" on top of the head commit in the remote, leaving a linear history



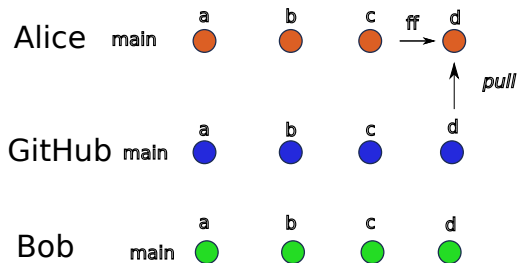
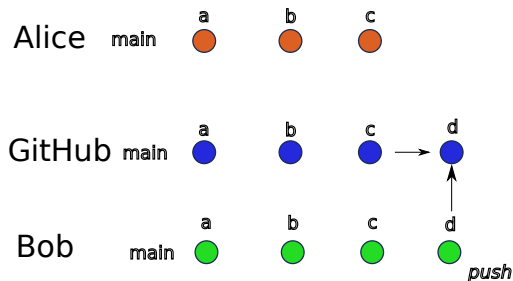
When Alice pushes to the remote the final situation on the remote is



With the rebase strategy the history of changes is altered to the benefit of a linear history

Fast forward

For completeness, we mention that if Alice had pulled the changes made by Bob without having added herself a new commit, we have a **fast forward** (unless we use `git pull -no-ff`)



Overriding default pull strategy

You have set the default pull strategy to *merge* but you want to pull with the *rebase* strategy?
Just do:

```
$ git pull --rebase=true <remote> <branch>
```

or, simply,

```
$ git pull --rebase <remote> <branch>
```

In the opposite situation:

```
$ git pull --rebase=true <remote> <branch>
```

A Note: When pulling from the remote it's better to rebase: you get a cleaner, linear, history.

Not possible to pull because of uncommitted changes

Another thing that may happen frequently is that you cannot pull from the remote repository because you have modified some files and not committed them. Git suggest what you have to do: either commit or stash. If you are not ready to commit your modification, you use `git stash`

```
$ git stash # stash your changes  
$ git pull #pull from the repo  
$ git stash pop #get your changes back
```

As simple as that (unless you have conflicts... see later)

Conflicts

Sometimes conflicts may arise!

```
CONFLICT (content): Merge conflict in file.cpp  
Automatic merge failed; fix conflicts and then commit the result.
```

It happens if Alice and Bob have modified the same lines of the same file! Git will tell you which are the conflicting files. (use `git status`) **What do I do now!!?**.

Since solving conflicts after a pull is identical to solving them when merging a branch, the solution is reported in the slides about branching.

Adding more files and directories

Now I create a directory `src` where I put some other files `file3.cpp` and `file4.cpp`. I do

```
$ git add src
```

Adding a directory means adding **all files in the directory**. Doing `git status` gives indeed:

```
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   src/file3.cpp
    new file:   src/file4.cpp
```

As expected!

Moving file

But now I realize that also file1.cpp and file2.cpp should be in src! The simplest thing to do is to use

```
$ git mv file1.cpp src/
```

git status now says

```
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
renamed:   file1.cpp -> src/file1.cpp
renamed:   file2.cpp -> src/file2.cpp
new file:   src/file3.cpp
new file:   src/file4.cpp
```

Great! I am ready for the next commit!

```
$ git commit -m "Added a few more files"
```

A note

You can launch git commands from any directory of the working tree! Non necessarily from the root.

A review: the possible states of a file in the working area

To summarize, a file in the working area can be in one of these 3 states:

- ▶ **untracked** The file is **not under git control**. Maybe it is just a temporary file. Or maybe you want to put it under control using `git add <file>`.
- ▶ **modified** The file has been modified since the last **commit** in the repository. Maybe you want to add it to the staging area with `git add <file>`.
- ▶ **staged** (in the index, or staging area). Ready to be **committed** into the repository with a `git commit`.

Checking-out a commit

```
$ git checkout <treeish>
```

Where `treeish` can be:

- ▶ The [hash key](#) of a commit, or a tag name;
- ▶ The name of a [branch](#): it will checkout the last commit (HEAD) of that branch;
- ▶ The [shorthand](#) name of a commit.

Some shorthand names for commits

Shorthand	Definition
HEAD	Last commit
HEAD^	One commit ago
HEAD^^	Two commits ago
HEAD~1	One commit ago
HEAD~3	Three commits ago

These shorthands can be used in all git commands that refer to a commit. **HEAD may be thought as a pointer to the last commit.**

```
$ git log HEAD~3..HEAD  
$ git checkout HEAD~2
```

The first command prints a brief log describing the last 4 commits. The second positions your working area to two commits ago.

Last commits

Use `git log` to have a look at the last commits

```
$ git log
commit fa1adfee063d73d110dc25e0299f7529857942c0 (HEAD -> master)
Author: formaggia <luca.formaggia@gmail.com>
Date: Sat Nov 9 19:07:54 2024 +0100
Added a few more files
commit 2ffb31e4f3d46da541b37f15338ebcb00ef7554c (origin/master)
Author: formaggia <luca.formaggia@gmail.com>
Date: Sat Nov 9 18:42:48 2024 +0100
...
```

Important

If you checkout a commit that is not at the tip of a branch (we will deal with branches soon!) then you are put to a **detached HEAD state**. You can modify files but not make commits! If you want to commit changes made while in detached HEAD state you have to create a branch:

```
$ git checkout HEAD~1
```

Note: checking out 'HEAD~1'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at 2ffb31e a modification and a new file

Delete and recover file

If you need to remove the file `file1.cpp` from your git working space you should consider two situations

1. The file is not under git control: just remove it;
2. The file is under git control. You can either

```
$ rm file1.cpp  
$ git add file1.cpp
```

or (better)

```
$ git rm file1.cpp
```

In both cases, the modification is staged for the next commit. To make it operative

```
$ git commit -m"I have deleted a file"
```


Recovering a deleted file

If I want to recover a deleted file (which is in the git history) again there are two possibilities

1. The file has been deleted but not staged nor committed.

```
$ git checkout -- <file>
```

2. The file deletion is staged. You need to unstage it first.

```
$ git restore --staged <file>  
$ git checkout -- <file>
```

3. The file deletion has already been committed (maybe time ago...). You need to search the last commit where the file was present

```
$ git log -- file1.cpp  
... commit fa1adfee063d73d110dc25e0299f7529857942c0 ...
```

and then use the commit hash in the command:

```
$ git checkout fa1adf -- file1.cpp
```

Important note

The fact that git allows to recover deleted files it means that removing a file from the working space **does not eliminate it from the git history**.

As a consequence, if you by mistake have committed a big temporary file that should not be put under git control, **git rm does not free space in the repository!**

Deleting a file from git history is not trivial, you should use a tool like **git-filter-repo**, **reading the instructions carefully!**.

In conclusion: **beware of not putting under git control big files that should not be under git control: zip files, executables, compiled libraries, data bases, big data files.... If you are not sure of what are you committing, use git status or a good graphical interface to check.**

Use .gitignore to select files you do not want to go in your git repo by mistake.

Fixing commits: amend reset and revert

Sometimes you want to correct your **last commit**, for instance to add some additional stuff, correcting the commit message, . . . , without creating a new commit

```
$ # do your additional work/add/rm  
$ git commit --amend
```

Never amend a commit that has been already pushed to a remote repo

Undo staging

`git reset` sets your HEAD to a specified commit. But it is a very versatile command

```
$ git reset <file>
```

Removes the specified file (all if `<file>` is empty) from the staging area, leaving the working directory unchanged.
It is the opposite of `git add`.

Eliminating commits but not the changes

```
$ git reset <commit>
```

Move back to <commit>, leaving the working directory alone: all changes made since <commit> will be in the working directory, not staged.

Hard Reset: bring history back!

```
$ git reset --hard <commit>
```

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. You are back at the situation at commit <commit>. **All modifications since <commit> are lost.**

```
$ git reset --hard
```

Equivalent to `git reset --hard HEAD`. In addition to unstaging files, Git will **overwrite all changes in the working directory**. **All uncommitted changes are obliterated**, and you go back to the situation at your last commit.

Reset and checkout

Note the difference between

```
$ git reset --hard <commit>
```

and

```
$ git checkout <commit>
```

With `checkout` the commits after <commit> (among which HEAD) **are still there**, that's why you are put in a *detached HEAD state*. `reset --hard <commit>` eliminates the commits following <commit> from the git history and sets HEAD to <commit>. **All changes since <commit> are deleted.**

A warning

`git reset --hard <commit>` may change the history.

Do not use it to “delete” commits that have already been pushed to a remote repository shared with collaborators (unless you want to lose friends).

Use `revert` in this case!

Revert

`git revert` reverts commits by **applying them backwards**. So it does not “delete” them. It is history safe.

```
$ git revert HEAD~3
```

reverts the changes specified by the fourth last commit in HEAD and creates a new commit with the reverted changes.

```
$ git revert HEAD~3..HEAD
```

reverts all changes introduced in the last four commits, creating a new commit with the reverted changes.

A pictorial view

HEAD



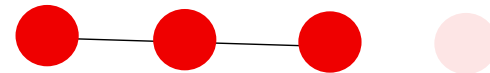
Initial condition

HEAD



git checkout HEAD¹

HEAD



git reset HEAD¹

HEAD



Cleaning up!

All commands seen in the previous slides (like almost all git commands) operate on file under git control. Sometimes you want to get rid of **untracked files** and have in your working area only the tracked one.

```
$ git clean [-d] [-f] [-x]
```

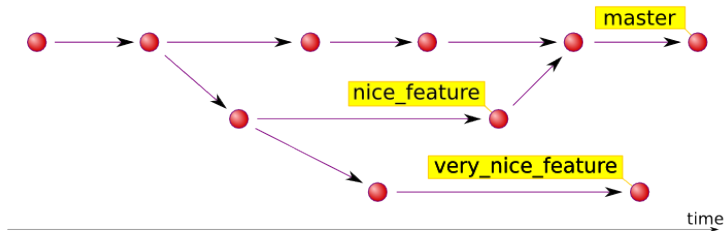
Option **-f** (force) is in fact **compulsory**. With just that option the command cleans all untracked files not listed in `.gitignore` (the files in `.gitignore` are untracked but considered as **known to git**.), with **-x** you get rid also of them. Directories emptied by the cleaning are kept unless you specify **-d**

There are also other options: e.g. **-i** = interactive mode

Some advice

- ▶ **Do atomic commits**: each commit should be related to a single “logical change” in your code: a bug fix, a new feature, ... **Avoid monster commits with a lot of modified files.**
- ▶ **Write significant commit messages**. Messages like “this is a commit” are useless! A good message helps you to remember what you have done, and others understand your work.
- ▶ **Git doesn't allow commits with empty messages.**

Branches



Branching is one of the most important feature of git. When you create a repo you create also the `master` branch (sometimes called `main`). But often you want to create a branch to work on a specific topic (or just to experiment something) without affecting the master branch.

Branches: Create

create new branch

```
$ git branch shiny_new_branch
```

Go to that branch.

```
$ git checkout shiny_new_branch
```

Or create and position yourself in a new branch with a single command

```
$ git checkout -b shiny_new_branch
```

Get other branches

list local branches

```
$ git branch  
  b1  
* b2  
  master
```

list remote branches (*i.e.* branches fetched from remote, see fetching later on)

```
$ git branch -r
```

list all branches (local and remote)

```
$ git branch -a
```

Branches: Delete

delete a branch

```
$ git checkout master  
$ git branch -d branch_to_delete
```

will cause an error if the branch was not merged with master!
To delete it without merging with master

```
$ git branch -D branch_to_delete
```

To delete a remote branch (see later for remotes)

```
$ git push origin :branch_to_delete
```


Branches: Merge

Often when you have finished working on a branch and you are happy of your work you want to merge the work on the master branch (or any other branch in fact).

To merge a branch into master:

```
$ git checkout master  
$ git merge branch_name
```

You can also **rebase** before merging

```
$ git rebase master #rebase current branch on master  
$ git checkout master  
$ git merge branch_name
```

Branches: Conflicts

Sometimes conflicts may arise!

```
CONFLICT (content): Merge conflict in file.cpp  
Automatic merge failed; fix conflicts and then commit the result.
```

What do I do now!!?.

Why conflicts?

Git tries to do its best to merge the work you have done in the branch, but maybe in the meantime you or one of your collaborators have modified the same lines of the same file on `master` (or the branch to which you are merging).

In that case you have a **conflict**, git stops the merge and it is up to you to sort out the mess.

If you want to cancel the merge and go back to the situation just before:

```
$ git merge --abort
```

Branches: Solve conflicts I

Manual solution. In the file with the conflict you have

```
<<<<<< HEAD
...portion in your HEAD commit
=====
...portion in the branch you are merging
>>>>>> branch_name
```

Use your preferred editor and modify the file. Eventually you should eliminate the <<<< and >>>> lines.

There are several graphic tools to help you fixing conflicts, such as **KDiff3** or **meld**, that can be activated using `git mergetool`. Or integrated in your IDE.

Branches: Solve conflicts II

Using a gui

```
$ git mergetool --tool=<tool>  
merge tool candidates:  meld opendiff kdiff3 tkdiff xxdiff tortoisemerge gvimdiff  
diffuse ecmerge p4merge araxis emerge vimdiff
```

You may use one of the tools indicated (you must have it installed). Some of them are rather sophisticated and provide a nice graphical interface.

Branches: Solve conflicts III

After you have fixed the conflicting files, you have to stage them

```
$ git add file.cpp
```

A commit is needed when all the conflicts are solved!

```
$ git commit
```

A special branch: the stash

Checkout with hanging modifications is forbidden!

```
$ git checkout master  
error: Your local changes to the following files would be overwritten by  
checkout: ...
```

Instead of creating a branch we can stash modifications

```
$ git stash save
```

and bring them back

```
$ git stash { pop | apply }
```

remember to clear the stash if you use apply, which does not delete the stash.

```
$ git stash clear
```

Additional information

Ignoring files!

Often in your working directory you have files (temporary results, executables, libraries, ...) that you **do not want to store in your repository**. You may just ignore them, but if you want to avoid making mistake (and stop `git status` indicating them as **untracked**) you can create a **.gitignore** file with the list of files to ignore (you can use wildcards)

`.gitignore` may be at the repo root level and in a sub-folder. In the latter case, it applies to files in the folders rooted in that sub-folder.

Remember to add and commit `.gitignore`, it must be in your repo!

Summary on commands operating on a remote

Share/retrieve a new branch with a remote called `repo_name`

```
$ git push repo_name branch_name  
$ git pull repo_name branch_name
```

Access a new branch in the repo after having fetched it

```
$ git checkout --track -b branch_local_name \  
    repo_name/branch_name
```

If you have just one repo it is sufficient to do

```
$ git checkout --track branch_name
```

A **tracking branch** is a branch that is linked to the corresponding branch in a remote repository. Indeed you may have **local branches** just for local modifications.

Summary on commands operating on a remote

Pull and adjourn submodules (if any)

```
$ git pull --recurse-submodules
```

Pull and do rebase instead of merging

```
$ git pull --rebase
```

Pull and do merge instead of rebase

```
$ git pull --rebase=false
```

Fetch instead of pulling (and merge/rebase later)

```
$ git fetch repo_name  
$ git merge|rebase remote/repo_name/branch_name
```

Get info

```
$ git log
```

Shows commit logs.

```
$ git shortlog
```

Summarizes log output, showing commit description from each contributor

```
$ git show <object>
```

Show some properties of the object (blob, commit, ...)

Use a graphical interface like `gitk` or `git-cola`, ...

Other useful commands

To show the differences with the last commit (you may specify a different commit and/or some specific files)

```
$ git diff branch files
```

To remove file(s) under git control from the working directory and stage it for removal in the next commit

```
$ git rm file(s)
```

To change the name of a file under git control (stages the change for the next commit)

```
$ git mv oldname newname
```

Frequently Asked Questions

Common issues - I

Q edited on master branch by mistake, I wanted to do in new_branch instead!

A a simple checkout will do it

```
$ git checkout -b new_branch
```

Q already committed to master branch

A create a new branch and delete the commit on master

```
$ git branch new_branch  
$ git reset --hard HEAD~1  
$ git checkout new_branch
```

Common issues - II

Q I want to discard my edit and get the latest version of the file!

A Simple

```
$ git checkout -- file
```

Q I want to get the file from another branch or another commit!

A It's the same command as before:

```
$ git checkout <branch|commit> -- file
```


Common issues - III

Q Get a commit from another branch

A cherry-pick it to the correct one

```
$ git checkout correct_branch  
$ git cherry-pick commit_hash
```

Q cannot get the branch of the local remote

A Try with a fetch

```
$ git fetch remote_name  
$ git remote show remote_name
```

Common issues - IV

Q the merge has gone bananas!

A give up the merge and try again

```
$ git merge branch_to_be_merged  
$ $!&#%$&^  
$ git reset --hard original_branch
```

or, more simply

```
git merge --abort
```

Q I want a file from that branch!

A get it with

```
$ git checkout branch files
```

A few (humorous) jokes and advices

- git pull a day keeps the conflicts away
- Commit early, commit often
- Be careful not to remove the branch you're standing on
- git: committed for life: once you go git, you never go back.
- It's not that hard, you just need to git good
- I git by with a little help from my friends

And remember...

In case of fire



1. `git commit`



2. `git push`



3. `leave building`