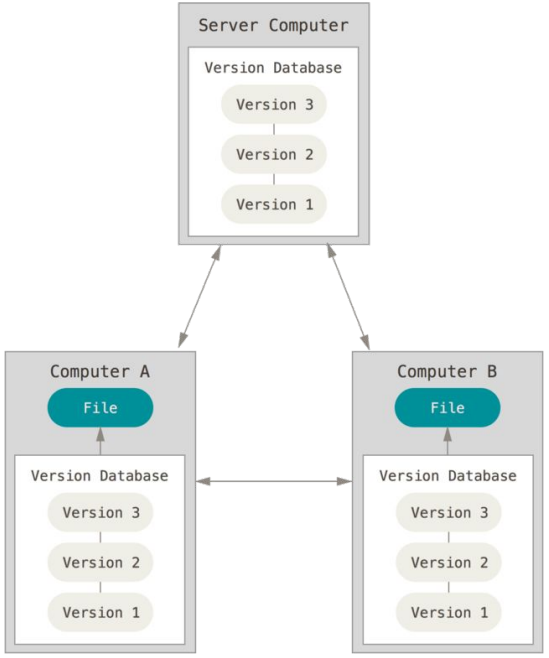
Contributing to IC

git basics

What is Git

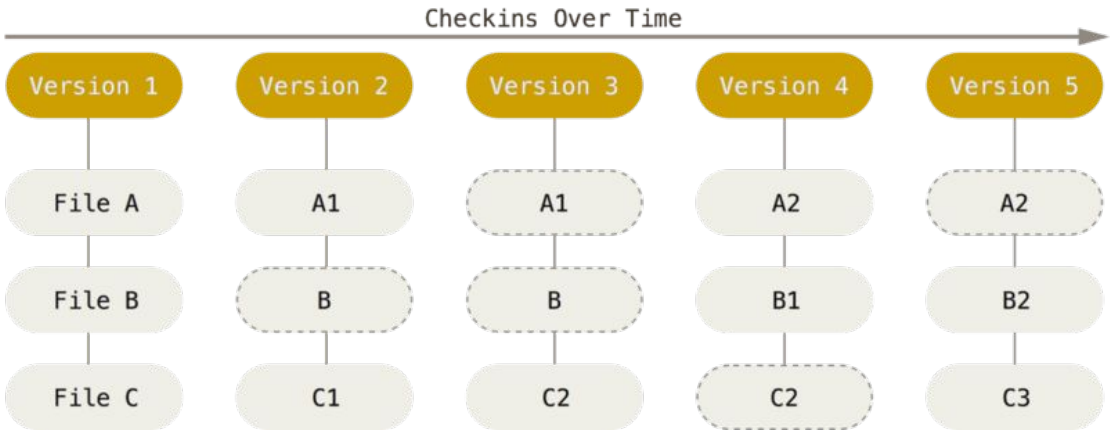
Git is a distributed Version Controlling System (VCS) : clients fully mirror the repository including its history



https://git-scm.com/book/en/v2

What is Git

Git takes a picture of what all your files look like at that moment and stores a reference to that snapshot.



Getting started

Make sure to configure your username and email.

$ git config --global user.name "username" # preferably something recognizable
$ git config --global user.email username@example.com

To start tracking a local repository use

$ git init

This will create .git folder in your repository. To stop versioning the repository just delete the folder

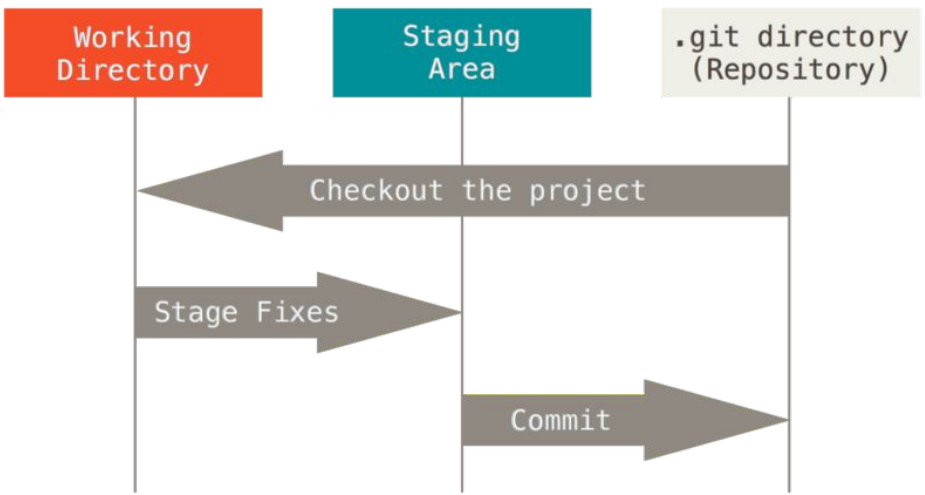
Exercise: create a local folder and add 2 text files (first_file.txt with line ‘First line’, and second_file.txt with ‘First line’) and initialize git

Git basics

Git has three main states that your files can reside in: modified, staged, and committed:

- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a file in its current version to go into your next commit snapshot (if you modify it again it will be partially staged).
- Committed means that the data is safely stored in your local database.

The additional files that are not being versioned by git appear as untracked.



Git basics

To check in what status are your files use
$ git status

To stage files use
$ git add filename1 filename2 … (git add -A adds everything)

To commit changes
$ git commit -m ‘commit message’ (-a flag will commit all modified files)

To see log history
$ git log

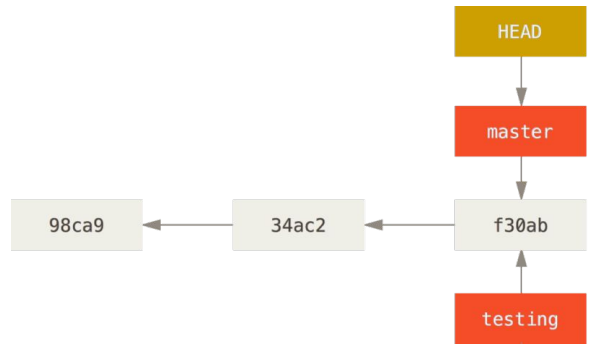
To see differences
$ git diff branch/commit (branch/commit)

To ignore file create .gitignore and add the filename(s) you don’t want to track. Wildcards are accepted (e.g. *.pyc)

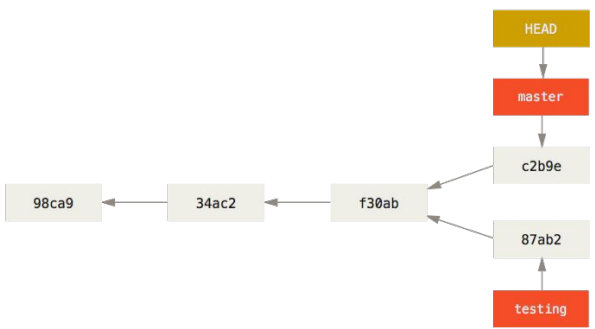
Exercise: in our folder stage first_file.txt, add ‘Another line’, stage and commit (‘First commit’)
Make second_file.txt be ignored. After every step run git status to understand the process.

Git branching

Branch is just a new pointer (points to the commit from where you created the branch). The default branch name in git is master



- When committing each commit refers the previous one.
- The new branch (testing) refers to the commit where it started
- HEAD points to the current branch you are in



We can make new commits in both master and testing by switching between the 2 branches

Git branching

To create a new branch starting at the last commit
$ git branch branch_name

To list all branches in your repo (starred one is the one you are on)
$ git branch

To switch to other branch do:
$ git checkout branch_name (or commit hash)

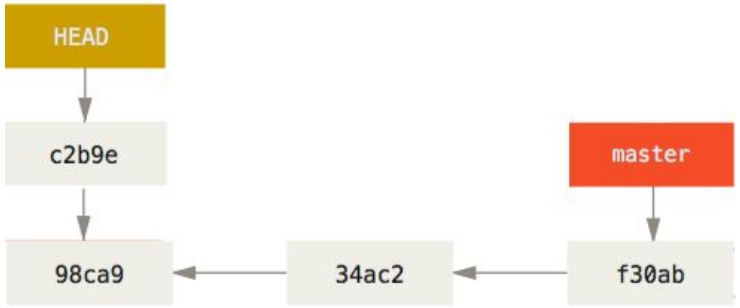
To delete a branch
$ git branch -d branch_name

Exercise: Remove .gitignore file and commit second_file.txt (‘Second commit’). Create a new dev1 and dev2 branches starting from Second_commit.
In dev1 add ‘Second line’ in second_file.txt, and commit ‘Add line in second_file’
In dev2 modify ‘Another line’ in first_file.txt to ‘Second line’, stage and commit ‘Modify first_file’
Check the branches logs

Git branching

DETACHED HEAD: not belonging to any branch (simply checking out one

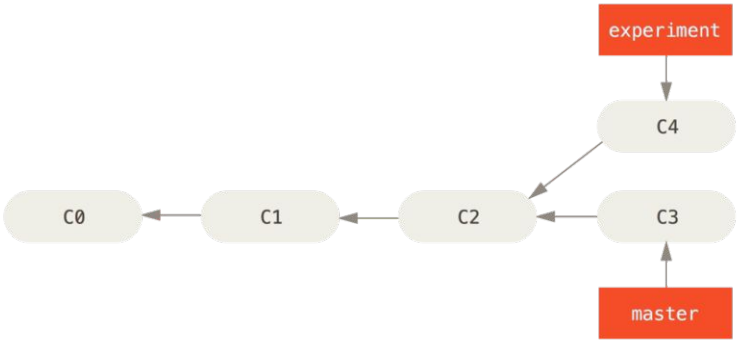
symbols ^ and ~ are used to point to relative position;
ie in this example HEAD is pointing at master^2

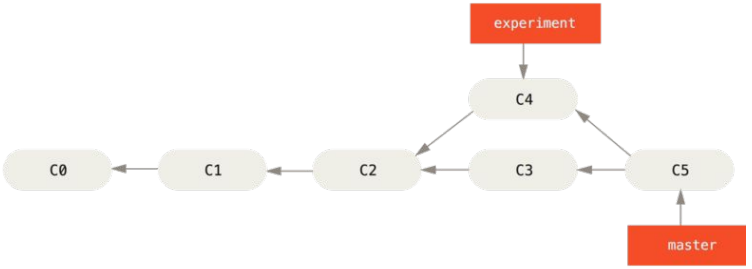


Commits made in detached head will not be saved because there is nothing referring to them!

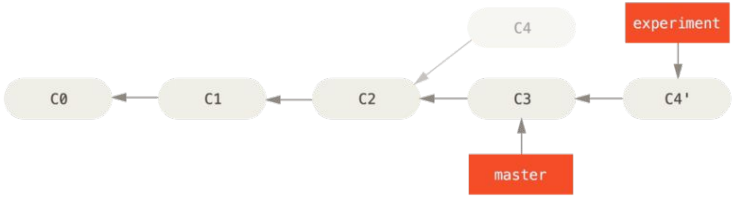
Git branching

2 ways to integrate changes to the main branch: merging and rebasing





Merging creates performs a three-way merge creating a new snapshot (and commit).



Rebase reapplies commits and creates linear history

Git branching

To merge current branch with another branch use:
$ git merge another_branch

To rebase the current branch onto another branch use:
$ git rebase another_branch

When joining branches there can be a merge conflict (if two branches made commits in the same lines). The conflicts will be displayed between <<<<<<<, =======, and >>>>>>>.
You should modify the conflicted file, stage it and continue merging/rebasing process
($ git rebase (merge) -continue)

Exercise: Checkout master and modify ‘Another line’ in first_file.txt to ‘2nd line’,
stage and commit ‘Modify first_file in master’
Checkout dev1 and merge it on master, check the history.
Checkout dev2 and rebase onto master, resolve merge conflicts keeping dev2 commit.

Magit

A simple(r) way to interact with git. Install emacs and follow instructions from:
https://magit.vc/manual/magit/Installing-from-Melpa.html#Installing-from-Melpa

Or easier (if it works):

wget https://raw.githubusercontent.com/nextic/IC/master/doc/init.el -O ~/.emacs.d/init.el
and (close and) open emacs

Shortcuts:
C-x g to start magit (it shows git status)
s/u to stage/unstage files
h will show all the options available

Exercise: Start magit. Compare dev1 and dev2 log history (notice difference between merge and rebase). Delete dev1.
In dev2 add ‘Third_line’ in first_file and commit. Create test_dev starting from dev2.

Some useful features

- Stashing (z) will save the snapshot of a changes in a repo without committing them.
  Exercise: In dev2 add ‘Fourth line’ in second_file.txt, stage and commit. Add ‘Not sure is good’ line to second_file.txt and stage it. Try to checkout test_dev. Stash change and checkout test_dev.

- Resetting (X) will reset the whole repo or the file to a given commit.
  Exercise: In test_dev reset first_file.txt to match master, commit changes

- Reverting (V) is undoing commits or changes
  Exercise: Revert changes in test_dev

- Commit options amend will add changes to last commit (extend if you dont want to change commit message)
  Exercise: Add ‘Second’ line in second_file.txt and commit. Modify ‘Second’ to ‘Second line’ and commit (amend or extend). Make sure it is only one commit.

Interactive rebase

Before merging your code you might want to rewrite the history. You can do it easily with interactive rebase option in magit. This process is **irreversible**.

**tip: when unsure about a rebase, create a branch in the top of the branch you are rebasing. This way, you will have a backup of your changes before rebasing*

Exercise: Add ‘Forth line’ to first_file and ‘Third line’ to second_file, stage and commit. Your history should look like this

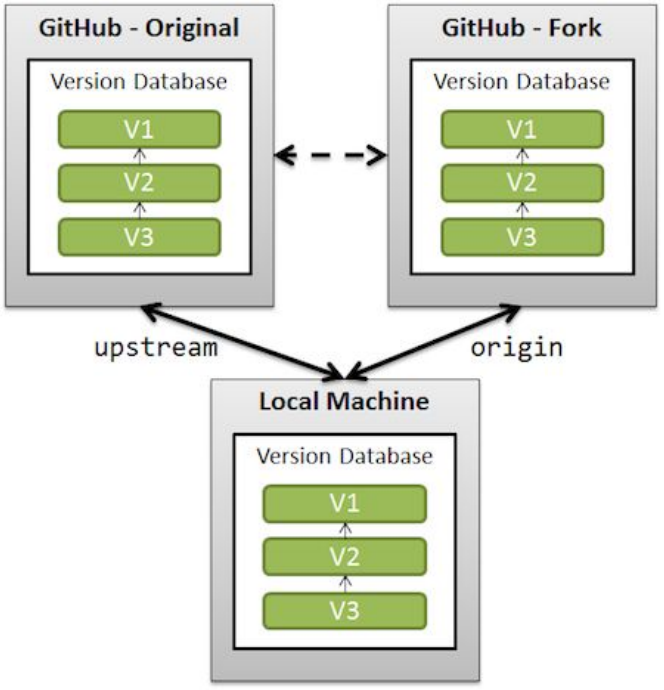
```
0406014 * test_dev Want to split this
555bace * Add second line in second file
1e8e1ca * Revert "Resseted first_file from master"
6ac7652 * Reset first_file from master
f85d30b  * Add third line in first_file
c18c267 * Modify first_file
1eee17f * master Modify first_file in master
01c888d * Second commit
baec38e * First commit
```

Run interactive rebase from ‘Modify first file’. Use instruction to:

- squash “Modify first_file” and “Add third line in first_file”;
- put “Add second line in second file” right after those commits
- kill “Resseted first_file from master” and Revert "Resseted first_file from master";
- Edit the last commit. Run rebase and follow instructions
- Check git status (you will see rebase in process). Reset (**soft**) to HEAD~ (or ^). Commit two files in two different commits. Continue rebase and check the log.

Remotes

Remote is just a version of the repository hosted on a server (GitHub, GitLab...)

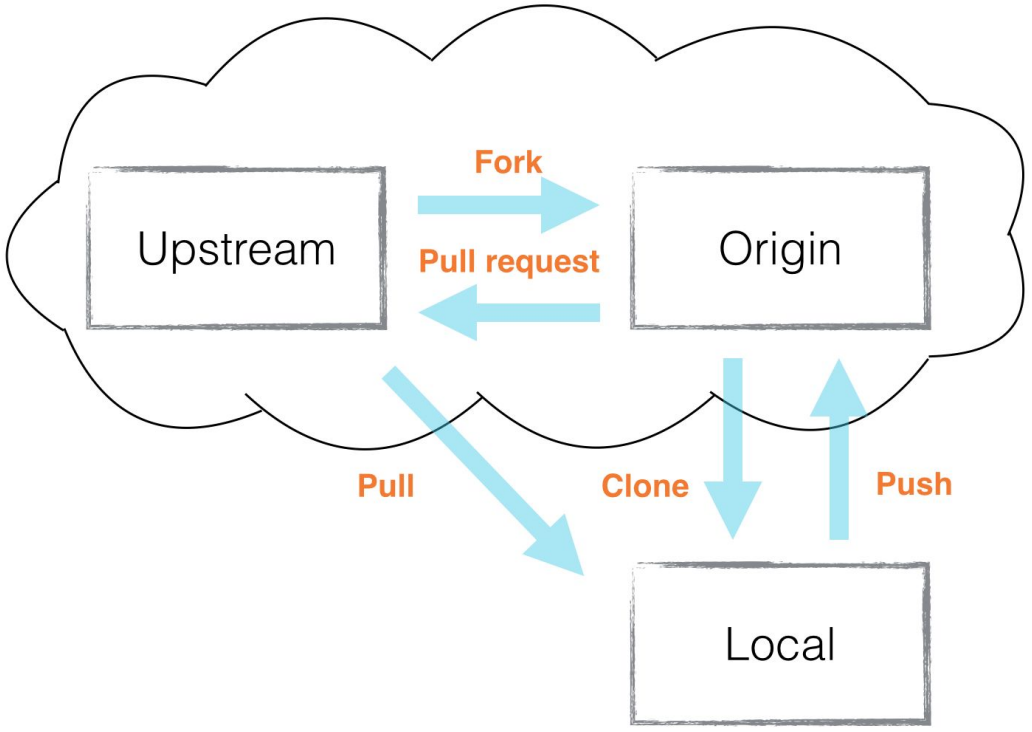


Remotes

Make your own fork of IC (hopefully everyone has it by now). Make sure your origin points to your github remote and upstream to nextic. in magit: M a <name> <link_to_remote.git>

Exercise: Add me (mmkekic) to your remotes. Fetch my changes (check that you can see my branches etc) Create silly_branch starting from master. Add silly_file.txt and commit it. Push changes to origin/silly_branch. Delete branch locally and remotely

Git (GitHub) workflow



- upstream/master, origin/master and local master should be always the same (dont change your master branch)
- Changes are made in a development branch, first created locally, then pushed to origin/dev_branch and then merged to upstream via a PR (GitHub feature)
- When/before PR approved the dev_branch is rebased onto upstream/master and merged (merging is done by designated persons)
- You can delete your dev_branch locally and remotely

IC best practice

- Style:
  - Align (‘from’, ‘import’, ‘as’ across imports; ‘=’, ‘<’, ‘>’, ‘,’, ‘.’ across lines…). Whatever enhances symmetry.
  - Blank spaces (always after comma, before and after ‘=’ unless is a one liner keyword argument)

- When developing:
  - Make commits sensible, while developing commit frequently but clean the history before making a PR
  - Make commit message clear (‘Add test_exact_results in esmeralda_test’; use second line if needed)
  - Use Type Hints
  - Write docstrings
  - Always opt for readability
  - Avoid modifying mutable inputs of your functions
  - If fixing a bug, first show the bug with appropriate test and then commit the fix
  - If the PR is very big, try prepare an explanation/tutorial notebook

- When reviewing (remember you are equally guilty if the bug is introduced by the developer):
  - Make sure to understand what the code does and why (if necessary checkout the PR locally)
  - If the PR is changing the existing functionalities try to understand how will that affect the global distribution of the data
  - If the PR is small go commit by commit, otherwise is easier to look at ‘Files changed’ tab
  - Check that the functions are tested
  - Check style