Advanced use of Git

Matthieu Moy

Matthieu.Moy@univ-lyon1.fr

https://matthieu-moy.fr/cours/formation-git/advanced-git-slides.pdf

Oct 2024

Goals of the presentation

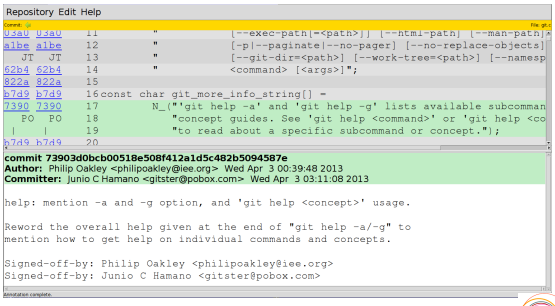
- Understand why Git is important, and what can be done with it
- Understand how Git works
- Motivate to read further documentation

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Merge branch "asdfasjkfdlas/alkdjf" into sdkjfls-final

Git blame: Who did that?

`git gui blame file`

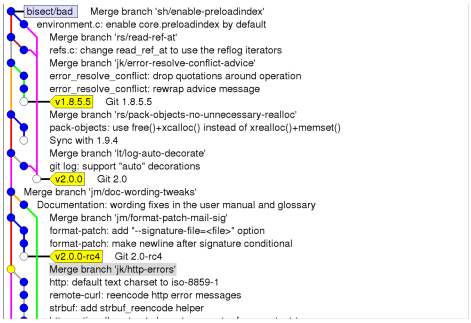


Bisect: Find regressions

```
$ git bisect start
$ git bisect bad
$ git bisect good v2.29.0
Bisecting: 607 revisions left to test after this (roughly 9 steps)
[8fe3ee67adcd2ee9372c7044fa311ce55eb285b4] Merge branch 'jx/i18n'
$ git bisect good
Bisecting: 299 revisions left to test after this (roughly 8 steps)
[aa4bffa23599e0c2e611be7012ecb5f596ef88b5] Merge branch 'jc/coding-guidelines'
$ git bisect good
Bisecting: 150 revisions left to test after this (roughly 7 steps)
[96b29bde9194f96cb711a00876700ea8dd9c0727] Merge branch 'sh/enable-preloadindex'
$ git bisect bad
Bisecting: 72 revisions left to test after this (roughly 6 steps)
[09e13ad5b0f0689418a723289dca7b3c72d538c4] Merge branch 'as/pretty-truncate'
...
$ git bisect good
60ed26438c909fd273528e67 is the first bad commit
commit 60ed26438c909fd273528e67b399ee6ca4028e1e
```

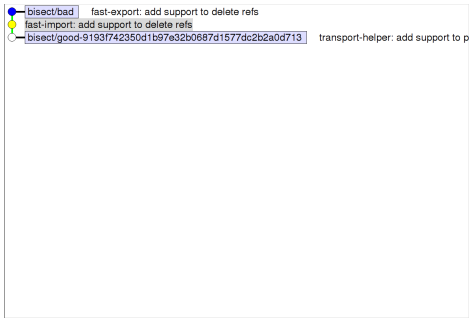

Bisect: Binary search

`git bisect visualize`



Bisect: Binary search

`git bisect visualize`



Then what?

`git blame` and `git bisect` point you to a commit, then ...

- Dream:
  - The commit is a 50-lines long patch
  - The commit message explains the intent of the programmer
- Nightmare 1:
  - The commit mixes a large reindentation, a bugfix and a real feature
  - The message says "I reindented, fixed a bug and added a feature"
- Nightmare 2:
  - The commit is a trivial fix for the previous commit
  - The message says "Oops, previous commit was stupid"
- Nightmare 3:
  - Bisect is not even applicable because most commits aren't compilable.

Which one do you prefer?
Clean history is **as** important **as comments**
for software maintainability

Two Approaches To Deal With History

Approach 1

“Mistakes are part of history.”

Approach 2

“History is a set of lies agreed upon.”[1]

[1] Napoleon Bonaparte

Approach 1: Mistakes are part of history

- ≈ the only option with Subversion/CVS/...
- History reflects the chronological order of events
- Pros:
  - Easy: just work and commit from time to time
  - Traceability
- But ...
  - Is the actual order of event what you want to remember?
  - When you write a draft of a document, and then a final version, does the final version reflect the mistakes you did in the draft?

Approach 2: History is a set of lies agreed upon

- Popular approach with modern VCS (Git, Mercurial...)
- History tries to show the best logical path from one point to another
- Pros:
  - See above: blame, bisect, ...
  - Code review
  - Claim that you are a better programmer than you really are!

Another View About Version Control

- 2 roles of version control:
  - For beginners: help the code reach upstream.
  - For advanced users: prevent bad code from reaching upstream.
- Several opportunities to reject bad code:
  - Before/during commit
  - Before push
  - Before merge

What is a clean history

- Each commit introduce small group of related changes (≈ 100 lines changed max, no minimum!)
- Each commit is compilable and passes all tests (“bisectable history”)
- “Good” commit messages

Reminder: good comments

- Bad: What? The code already tells

```
/*
 * Test if cmd is either --help or --version, and if so,
 * exit the current loop.
 */
if (!strcmp(cmd, "--help") || !strcmp(cmd, "--version"))
        break;
```

- Good (from `git.c`): Why? Usually the relevant question

```
/*
 * For legacy reasons, the "version" and "help"
 * commands can be written with "--" prepended
 * to make them look like flags.
 */
if (!strcmp(cmd, "--help") || !strcmp(cmd, "--version"))
        break;
```

Common rule: if your code isn't clear enough, rewrite it to make it clearer instead of adding comments.

Good commit messages

- Recommended format:

```
One-line description (< 50 characters)

Explain here why your change is good.
```

- Write your commit messages like an email: subject and body
- Imagine your commit message is an email sent to the maintainer, trying to convince him to merge your code[2]
- Don't use `git commit -m`

[2] Not just imagination, see `git send-email`

Good commit messages: examples

From Git's source code

https://github.com/git/git/commit/2939a1f70357d5b55232c2bf51e5ac32a4e7336c

```
mingw: bump the minimum Windows version to Vista

Quite some time ago, a last plea to the XP users out there who want to
see Windows XP support in Git for Windows, asking them to get engaged
and help, vanished into the depths of the universe.

We tried for a long time to play nice with the last remaining XP users
who somehow manage to build Git from source, but a recent update of
mingw-w64 (7.0.0.5233.e0c09544 -> 7.0.0.5245.edf66197) finally dropped
the last sign of XP support, and Git for Windows' SDK is no longer able
to build core Git's 'master' branch as a consequence. (Git for Windows'
'master' branch already bumped the minimum Windows version to Vista a
while ago, so it is fine.)

It is time to require Windows Vista or later to build Git from source.
This, incidentally, lets us use quite a few nice new APIs.

It also means that we no longer need the inet_pton() and inet_ntop()
emulation, which is nice.

Signed-off-by: Johannes Schindelin <johannes.schindelin@gmx.de>
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

Good commit messages: counter-example

GNU-style changelogs

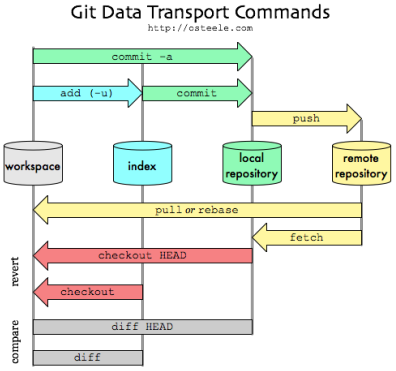
https://github.com/emacs-mirror/emacs/commit/bd013a448b152a84cff9b18292d8272faf265447

```
* lisp/replace.el (occur-garbage-collect-revert-args): New function

(occur-mode, occur-1): Use it.
(occur-region-start, occur-region-end, occur-region-start-line)
(occur-orig-line): Remove vars.
(occur-engine): Fix left over use of occur-region-start-line.
```

Nothing that the patch doesn't say already (5 lines, 0 bit of information), no idea what problem the commit is trying to solve.

Git Data Transport Commands



The index, or "Staging Area"

- "the index" is where the next commit is prepared
- Contains the list of files and their content
- `git commit` transforms the index into a commit
- `git commit -a` stages all changes in the worktree in the index before committing. You'll find it sloppy soon.

Dealing with the index

- Commit only 2 files:

```
git add file1.txt
git add file2.txt
git commit
```

- Commit only some patch hunks:

```
git add -p
(answer yes or no for each hunk)
git commit
```

`git add -p`: example

```
$ git add -p
@@ -1,7 +1,7 @@
 int main()
-       int i;
+       int i = 0;
        printf("Hello, ");
        i++;
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
@@ -5,6 +5,6 @@

-       printf("i is %s\n", i);
+       printf("i is %d\n", i);

Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
$ git commit -m "Initialize i properly"
[master c4ba68b] Initialize i properly
 1 file changed, 1 insertion(+), 1 deletion(-)
```

`git add -p`: dangers

- Commits created with `git add -p` do not correspond to what you have on disk
- You probably never tested these commits ...
- Solutions:
  - `git stash -k`: stash what's not in the index (`--keep-index`)
  - `git rebase --exec`: see later
  - (and code review)

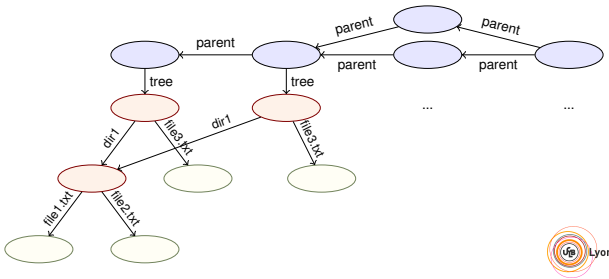
If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of "It's really pretty simple, just think of branches as..." and eventually you'll learn the commands that will fix everything.

Why do I need to learn about Git's internal?

- Beauty of Git: very simple data model
  (The tool is clever, the repository format is simple&stupid)
- Understand the model, and the 170+ commands will become simple!

Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



Git objects: On-disk format

```
$ git log
commit 7a7fb77be431c284f1b6d036ab9aebf646060271
Author: Matthieu Moy <Matthieu.Moy@univ-lyon1.fr>
Date:   Wed Jul 2 20:13:49 2014 +0200

    Initial commit
$ find .git/objects/
.git/objects/
.git/objects/fc
.git/objects/fc/264b697de62952c9ff763b54b5b11930c9cfec
.git/objects/a4
.git/objects/a4/7665ad8a70065b68fbcfb504d85e06551c3f4d
.git/objects/7a
.git/objects/7a/7fb77be431c284f1b6d036ab9aebf646060271
.git/objects/50
.git/objects/50/a345788a8df75e0f869103a8b49cecdf95a416
.git/objects/26
.git/objects/26/27a0555f9b58632be848fee8a4602a1d61a05f
```


Git objects: On-disk format

```
$ echo foo > README.txt; git add README.txt
$ git commit -m "add README.txt"
[master 5454e3b] add README.txt
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
$ find .git/objects/
.git/objects/
.git/objects/fc
.git/objects/fc/264b697de62952c9ff763b54b5b11930c9cfec
.git/objects/a4
.git/objects/a4/7665ad8a70065b68fbcfb504d85e06551c3f4d
.git/objects/59
.git/objects/59/802e9b115bc606b88df4e2a83958423661d8c4
.git/objects/7a
.git/objects/7a/7fb77be431c284f1b6d036ab9aebf646060271
.git/objects/25
.git/objects/25/7cc5642cb1a054f08cc83f2d943e56fd3ebe99
.git/objects/54
.git/objects/54/54e3b51e81d8d9b7e807f1fc21e618880c1ac9
...
```


Git objects: On-disk format

- By default, 1 object = 1 file
- Name of the file = object unique identifier content
- Content-addressed database:
  - Identifier computed as a hash of its content
  - Content accessible from the identifier
- Consequences:
  - Objects are immutable
  - Objects with the same content have the same identity (deduplication for free)
  - No known collision in SHA1 until recently, still very hard to find ⇒ SHA1 uniquely identifies objects (sha256 migration planned)
  - Acyclic (DAG = Directed Acyclic Graph)

On-disk format: Pack files

```
$ du -sh .git/objects/
68K     .git/objects/
$ git gc
...
$ du -sh .git/objects/
24K     .git/objects/
$ find .git/objects/
.git/objects/
.git/objects/pack
.git/objects/pack/pack-f9cbdc53005a4b500934625d...a3.idx
.git/objects/pack/pack-f9cbdc53005a4b500934625d...a3.pack
.git/objects/info
.git/objects/info/packs
$
```

⇝ More efficient format, no conceptual change (objects are still there)

Exploring the object database

- **`git cat-file -p`** : pretty-print the content of an object

```
$ git log --oneline
5454e3b add README.txt
7a7fb77 Initial commit
$ git cat-file -p 5454e3b
tree 59802e9b115bc606b88df4e2a83958423661d8c4
parent 7a7fb77be431c284f1b6d036ab9aebf646060271
author Matthieu Moy <Matthieu.Moy@univ-lyon1.fr> 1404388746 +0200
committer Matthieu Moy <Matthieu.Moy@univ-lyon1.fr> 1404388746 +0200

add README.txt
$ git cat-file -p 59802e9b115bc606b88df4e2a83958423661d8c4
100644 blob 257cc5642cb1a054f08cc83f2d943e56fd3ebe99  README.txt
040000 tree 2627a0555f9b58632be848fee8a4602a1d61a05f  sandbox
$ git cat-file -p 257cc5642cb1a054f08cc83f2d943e56fd3ebe99
foo
$ printf 'blob 4\0foo\n' | sha1sum
257cc5642cb1a054f08cc83f2d943e56fd3ebe99  -
```


Merge commits in the object database

```
$ git switch -c branch HEAD^
Switched to a new branch 'branch'
$ echo foo > file.txt; git add file.txt
$ git commit -m "add file.txt"
[branch f44e9ab] add file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt
$ git merge master
Merge made by the 'recursive' strategy.
 README.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```


Merge commits in the object database

```
$ git switch -c branch HEAD^
$ echo foo > file.txt; git add file.txt
$ git commit -m "add file.txt"
$ git merge master
$ git log --oneline --graph
*   1a7f9ae (HEAD, branch) Merge branch 'master' into branch
|\
| * 5454e3b (master) add README.txt
* | f44e9ab add file.txt
|/
* 7a7fb77 Initial commit
$ git cat-file -p 1a7f9ae
tree 896dbd61ffc617b89eb2380cdcaffcd7c7b3e183
parent f44e9abff8918f08e91c2a8fefe328dd9006e242
parent 5454e3b51e81d8d9b7e807f1fc21e618880c1ac9
author Matthieu Moy <Matthieu.Moy@univ-lyon1.fr> 1404390461 +0200
committer Matthieu Moy <Matthieu.Moy@univ-lyon1.fr> 1404390461 +0200

Merge branch 'master' into branch
```

Snapshot-oriented storage

- A commit represents exactly the state of the project
- A tree represents only the state of the project (where we are, not how we got there)
- Renames are not tracked, but re-detected on demand
- Diffs are computed on demand (e.g. `git diff HEAD HEAD^`)
- Physical storage still efficient

Branches, tags: references

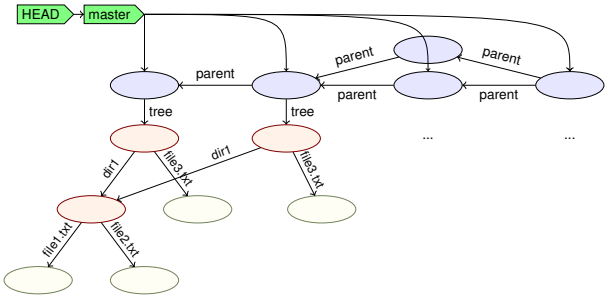
- In Java:

```
String s; // Reference named s
s = new String("foo"); // Object pointed to by s
String s2 = s; // Two refs for the same object
```

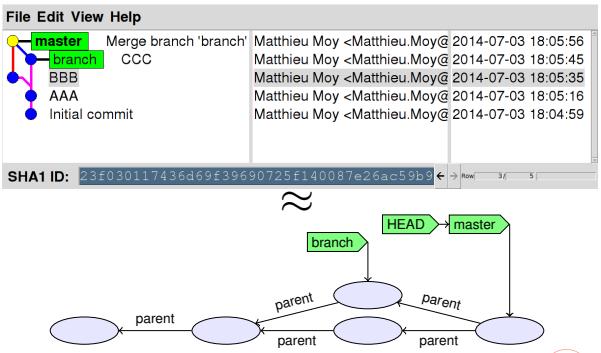
- In Git: likewise!

```
$ git log -oneline
5454e3b add README.txt
7a7fb77 Initial commit
$ cat .git/HEAD
ref: refs/heads/master
$ cat .git/refs/heads/master
5454e3b51e81d8d9b7e807f1fc21e618880c1ac9
$ git symbolic-ref HEAD
refs/heads/master
$ git rev-parse refs/heads/master
5454e3b51e81d8d9b7e807f1fc21e618880c1ac9
```


References (refs) and objects



Sounds Familiar?



≈

Branches, HEAD, tags

- A branch is a ref to a commit
- A lightweight tag is a ref (usually to a commit) (like a branch, but doesn't move)
- Annotated tags are objects containing a ref + a (signed) message
- HEAD is "where we currently are"
  - If HEAD points to a branch, the next commit will move the branch
  - If HEAD points directly to a commit (detached HEAD), the next commit creates a commit not in any branch (warning!)

Branches: Why and How

- 1 branch = 1 named ref to a commit
- Think of a branch as a set of commits
- Typical uses
  - maintenance branch (bugfix only, will lead to next minor release) vs development branch (new features, will lead to next major release)
  - Topic branch: 1 branch per feature
    - Create the branch
    - Work on it (`commit`)
    - Request a merge (`push` + pull-request, ...)
    - (Delete the branch when it's merged)

Branches and Tags in Practice

- Create a local branch and check it out:
  `git switch -c branch-name`[3]
- Switch to a branch:
  `git switch branch-name`
- List local branches:
  `git branch`
- List all branches (including remote-tracking):
  `git branch -a`
- Create a tag:
  `git tag tag-name`

[3] Old-timers like me still run `git checkout -b`.

Example

Implement `git clone -c var=value` : 9 preparation patches, 1 real (trivial) patch at the end!

`https://github.com/git/git/commits/84054f79de35015fc92f73ec4780102dd820e452`

Did the author actually write this in this order?

Git Rebase: TL; DR

`git rebase`,
`git rebase --interactive`:
⇝ Very powerful commands

(although a little dangerous)

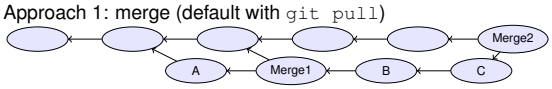
Omitted in this presentation

(slides kept for reference, read them offline, but we'll jump to 6)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)

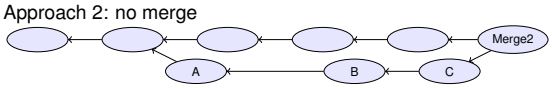


- Drawbacks:
  - Merge1 is not relevant, distracts reviewers (unlike Merge2).

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

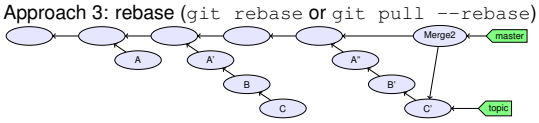


- Drawbacks:
  - In case of conflict, they have to be resolved by the developer merging into upstream (possibly after code review)
  - Not always applicable (e.g. "I need this new upstream feature to continue working")

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)



- Drawbacks: rewriting history implies:
  - A', B, C, A", B' probably haven't been tested (never existed on disk)
  - What if someone branched from A, A', B or C?
  - Basic rule: don't rewrite published history

Rewriting history with `rebase -i`

- `git rebase`: take all your commits, and re-apply them onto upstream
- `git rebase -i`: show all your commits, and asks you what to do when applying them onto upstream:

```
pick ca6ed7a Start feature A
pick e345d54 Bugfix found when implementing A
pick c03fffc Continue feature A
pick 5bdb132 Oops, previous commit was totally buggy

# Rebase 9f58864..5bdb132 onto 9f58864
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```


`git rebase -i` commands (1/2)

p, pick use commit (by default)

r, reword use commit, but edit the commit message
Fix a typo in a commit message

e, edit use commit, but stop for amending
- Once stopped, use `git add -p`, `git commit --amend`, ...

s, squash use commit, but meld into previous commit

f, fixup like "squash", but discard this commit's log message
- Very useful when polishing a set of commits (before or after review): make a bunch of short fixup patches, and squash them into the real commits. No one will know you did this mistake ;-).

`git rebase -i` commands (2/2)

x, exec run command (the rest of the line) using shell
- Example: `exec make check`. Run tests for this commit, stop if test fail.
- Use `git rebase -i --exec 'make check'`[4] to run `make check` for each rebased commit.

[4] Implemented by Ensimag students!

Git Reflog: TL; DR

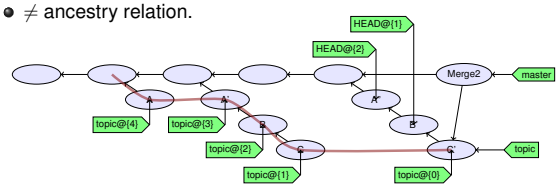
Git's reflog = detailed history

(makes `git rebase` less dangerous)

Good news:
if you don't know how to use it, a Git expert around you may do and recover data you thought was lost :-)

Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



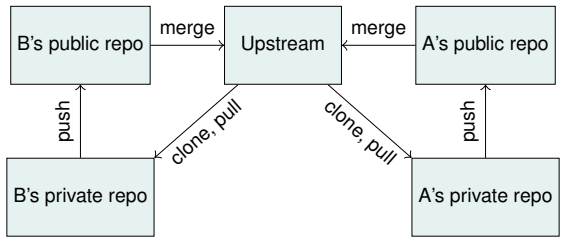
- `ref@{n}`: where *ref* was before the *n* last ref update.
- `ref~n`: the *n*-th generation ancestor of *ref*
- `ref^`: first parent of *ref*
- `git help revisions` for more

Centralized workflow

```
do {
   while (nothing_interesting())
      work();
   while (uncommited_changes()) {
      while (!happy) { // git diff --staged ?
         while (!enough) git add -p;
         while (too_much) git reset -p;
      }
      git commit; // no -a
      if (nothing_interesting())
         git stash;
   }
   while (!happy)
      git rebase -i;
} while (!done);
git push; // send code to central repository
```


Triangular Workflow with pull-requests

- Developers pull from upstream, and push to a "to be merged" location
- Someone else reviews the code and merges it upstream



Pull-requests in Practice

Contributor	create a branch, commit, push
Contributor	click "Create pull request" (GitHub, GitLab, BitBucket, ...), or `git request-pull`
Maintainer	receives an email
Maintainer	review, comment, ask changes
Maintainer	merge the pull-request

Code Review

- What we'd like:
  1. A writes code, commits, pushes
  2. B does a review
  3. B merges to upstream
- What usually happens:
  1. A writes code, commits, pushes
  2. B does a review
  3. B requests some changes
  4. ... then ?

Iterating Code Reviews

- At least 2 ways to deal with changes between reviews:
  1. Add more commits to the pull request and push them on top
  2. Rewrite commits (`rebase -i,...`) and overwrite the old pull request
     - ⋆ The resulting history is clean
     - ⋆ Much easier for reviewers joining the review effort at iteration 2
     - ⋆ e.g. On Git's mailing-list, 10 iterations is not uncommon.

Triangular Workflow: Advantages

- Beginners integration:
  - start committing on day 0
  - get reviewed later
- In general:
  - Do first
  - Ask permission after
- For Open-Source:
  - Anyone can contribute in good condition
  - "Who's the boss?" is a social convention

Tools ...

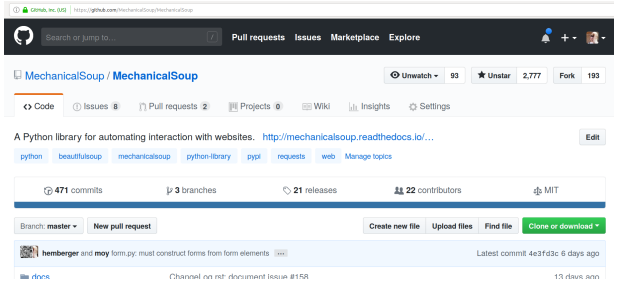
- Necessary for development (compiler, text editor)
- Catch mistakes early (tests, code analysis)
- Automate stuff (I'm lazy, too)

A Small Example: MechanicalSoup (Python library)

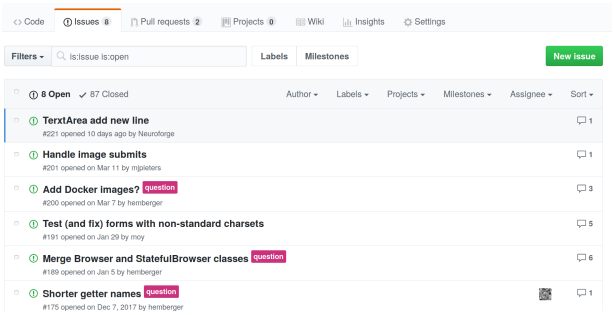
Disclaimer: I'm one of the authors

- Small library (400 LOC of Python + 1000 LOC of tests) for browsing websites
- Small, developed on free-time ⇒ no planning, no real methodology
- Tries to follow best practices and uses many fun tools
- Let's go trough a few of them... (you can do similar things with Lyon1's GitLab on `https://forge.univ-lyon1.fr/`)

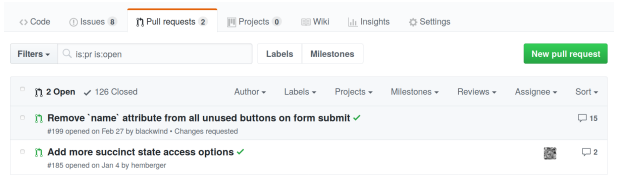
Hosting: Git, GitHub



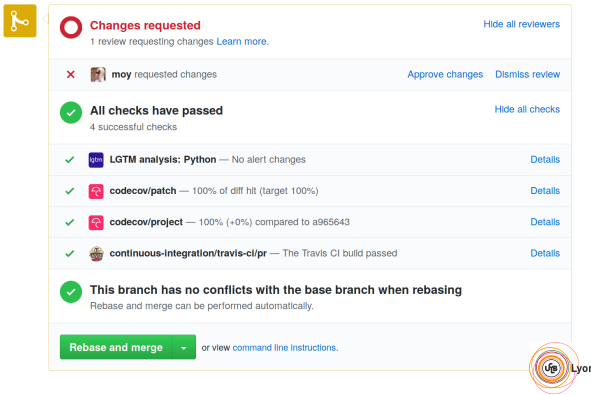
Report bugs, discuss future features: issue tracker



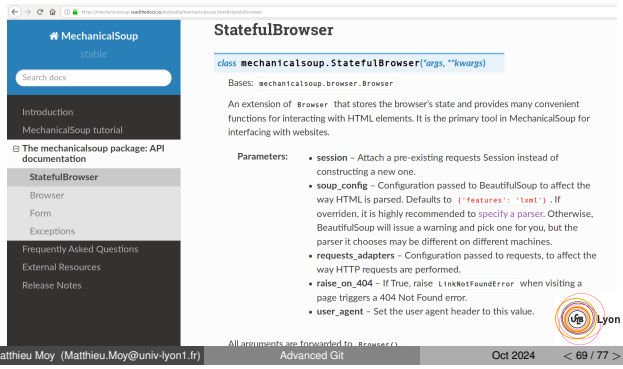
Submit code: pull-requests



Automated checks on pull-requests



Documentation automatically generated at each push (readthedocs)



About pull-requests and checks ...

- Anyone can submit a pull-request
- Pull-requests trigger build + tests + coverage check + style check + static analysis
  - Test failing ⇒ fail
  - Incompatibility with one supported version of Python ⇒ fail
  - Incorrect style (lines >80 characters, mis-placed space, ...) ⇒ fail
  - Line of code not covered by a test ⇒ fail
  - Bad pattern detected by code analysis ⇒ fail
- How we did all that? Mainly "use tools/services" and 30-lines long .travis.yml file.

Automated testing

(Because life it too short to spend time on manual testing)

- Code that tests code:

```
def test_no_404(httpbin):
    browser = mechanicalsoup.StatefulBrowser()
    resp = browser.open(httpbin + "/nosuchpage")
    assert resp.status_code == 404
```

- General form of automated tests:

```
def name_of_test_function():
    # given
    some_object = ...
    # when
    some_object.some_action(...)
    # then
    assert ...
```

Continuous Integration: example with GitLab-CI

https://gitlab.com/moy/gitlab-ci-demo

- Configuration (`.gitlab-ci.yml`):

```
before_script:
  - pip install flake8
  - pip install rstcheck

python_3_5:
  image: python:3.5
  script:
  - flake8 .
  - rstcheck *.rst
  - ./test.py

python_3_8:
  image: python:3.8
  script:
  - ./test.py
```

- Use: work as usual ;-)
  - Tests launched at each `git push`
  - Pass/failed indicator for each merge-request

Continuous Integration: example with GitHub and Travis-CI

https://github.com/moy/travis-demo

- Configuration (`.travis.yml`):

```
language: python
python:
  - "3.4"
  - "3.8"
install:
  - pip install pycodestyle
script:
  - pycodestyle main.py
  - ./test.py
```

- Use: work as usual ;-)
  - Tests launched at each `git push`
  - Pass/failed indicator for each pull-request

More Documentation

- `http://ensiwiki.ensimag.fr/index.php/Maintenir_un_historique_propre_avec_Git`
- `http://ensiwiki.ensimag.fr/index.php/Ecrire_de_bons_messages_de_commit_avec_Git`

Exercises

- Visit `https://github.com/moy/dumb-project.git`
- Fork it from the web interface (or just `git clone`)
- Clone it on your machine
- Repair the dirty history!