GIT tutorial

David Parsons, Matthieu Imbert, Soraya Arias, Thomas Calmant

May, 2018

Contents

1 Preamble **2**

2 First steps **2**

2.1 Basic configuration 2
2.2 Create your very first `git` repository 2
2.3 And now your first commit 3
2.4 Discarding changes 4
2.5 Unstage files 5
2.6 Playing with the index 6
2.7 Commit your work using git-gui 8

3 You are not alone **9**

3.1 Setup 9
3.1.1 Create your account 9
3.1.2 Fork the project 10
3.1.3 Clone the repo 10
3.2 Introduction 10
3.3 Simple commits / push / pull 11
3.4 Working copy and index / tracked and untracked files / .gitignore 13
3.5 Merge without conflict 16
3.6 Merge with conflict resolution 20
3.6.1 Manual conflict resolution by editing files 21
3.6.2 Manual conflict resolution with a merge tool 22
3.6.3 Pushing the resolved conflict 24
3.7 Branches 25

4 Conclusion **31**

1 Preamble

This practical work will illustrate `git` usage in 2 completely independent sections:

- In section 2, you will create your own `git` repository and manipulate the staging area to create your first commits.
- In section 3, you will learn how to collaborate through a central remote repository (push, fetch, merge), resolve conflicts and manipulate basic branches.

2 First steps

2.1 Basic configuration

Many things can be configured in `git`, but the most basic and almost mandatory configuration is to tell `git` who you are, *i.e.* your committer name and email address.
If you fail to do that, `git` will complain over and over about it.

```
$ git config --global user.name "<user name>"
$ git config --global user.email "<user email>"
```

You can also configure the default editor used by `git` for editing commit messages, if you don't like the default (`vi`). For example, one convenient editor on linux or mac is `nano`:

```
$ git config --global core.editor nano
```

On windows you can use `notepad`:

```
$ git config --global core.editor notepad
```

Note that when you run `git config` with the `--global` option, it sets the configuration globally (in your `~/.gitconfig`). Without this option, it will set it for the current `git` repository only (in `.git/config`).

2.2 Create your very first `git` repository

Make sure you create a directory for this practical work and go (`cd`) into it.
Now, initialize a new `git` repository and have a look at what happened:

```
$ git init
Initialized empty Git repository in ...
$ ls -A
.git
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

That's it, you have a (local) `git` repository at the ready !

2.3 And now your first commit

`Git` is often kind enough to hint you about what you should/could do next (depending on the current status). It just told you that you could: `create/copy files and use "git add" to track`, so let's do that:

```
$ echo "Hello World !" >> hello
$ git status
On branch master

Initial commit

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

  hello

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

So, we've created a file and `git` tells us it is *untracked*. Now it's telling us to: `use "git add" to track`.

```
$ git add hello
$ git status
On branch master

Initial commit

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

  new file:   hello
```

We've just *staged* file `hello`, that is, we've sent it to the commit preparation area (the staging area, a.k.a. the index).
We can now commit our work. This will create a new commit from the content of the staging area

```
$ git commit
```

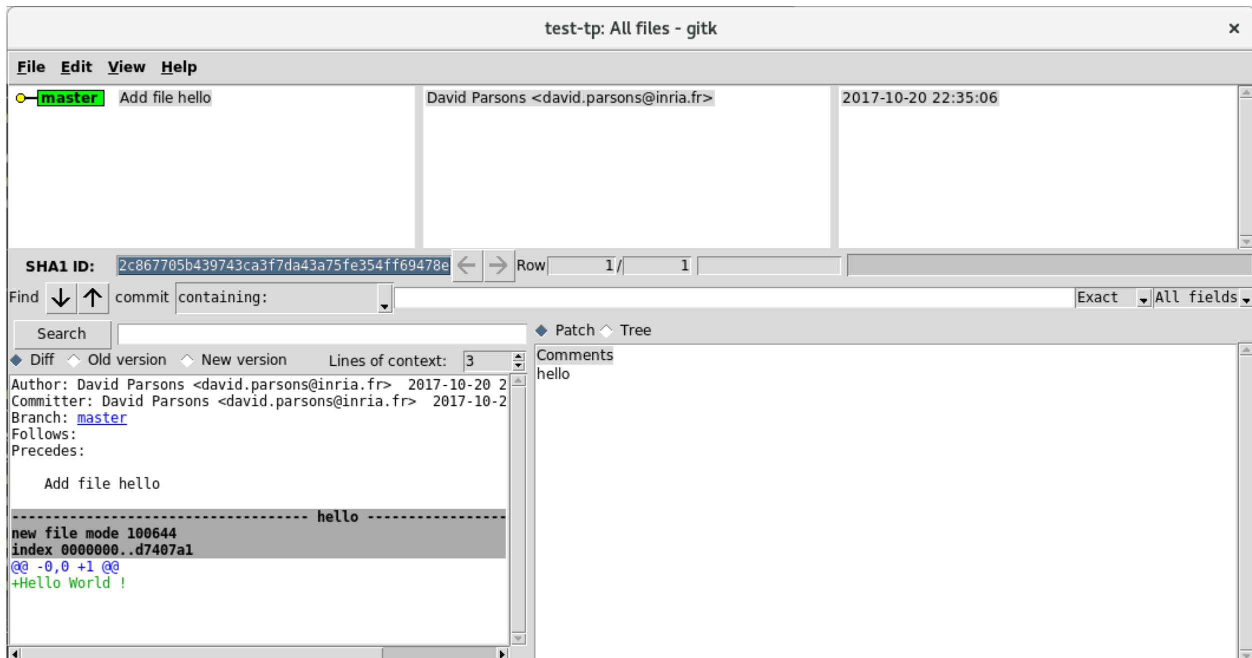
This opened the editor you have configured as `git`'s core editor and prompts you to *Please enter the commit message for your changes.*

A reasonable way to summarize our changes could be: *"Add file hello"*. Type this is your editor, then save and exit.

```
[master (root-commit) 2c86770] Add file hello
 1 file changed, 1 insertion(+)
 create mode 100644 hello
```

That's it, you've committed your first piece of work ! Now let's inspect the repository:

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



`Git` is telling you that your `working tree [is] clean`. That means that all your files are in the *unmodified* state, *i.e.* that you have no files in states *staged*, *modified* or *untracked*.

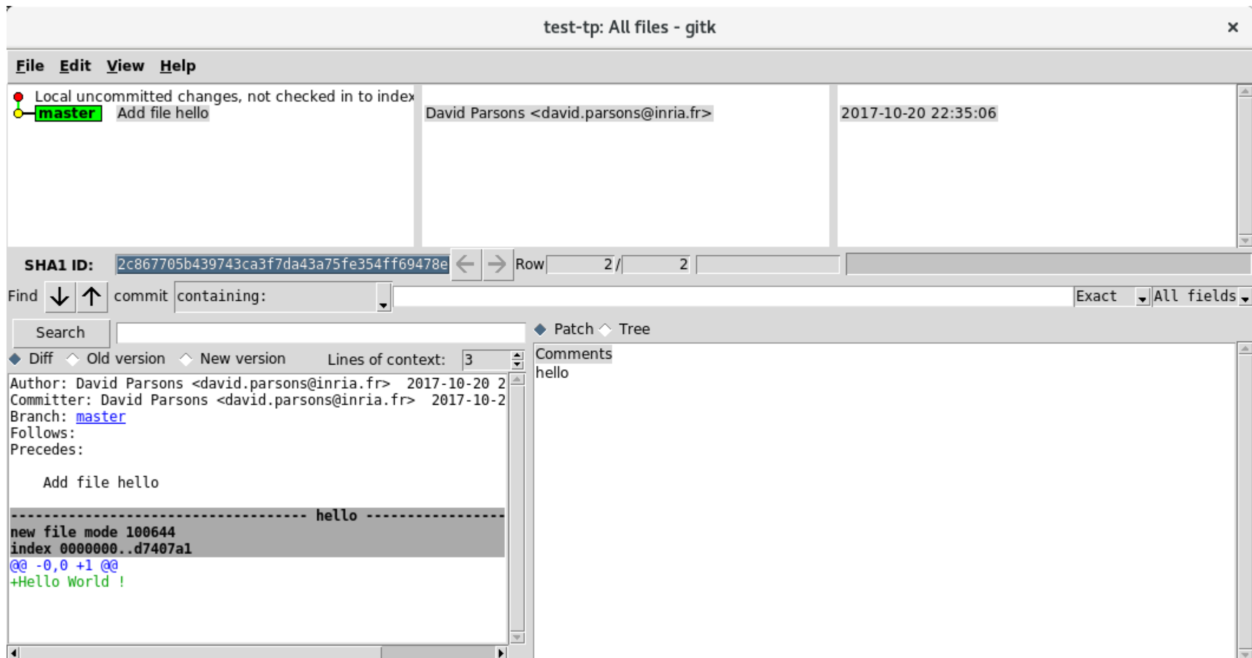
2.4 Discarding changes

Let's edit file `hello` and see what happens:

```
$ echo "Bonjour !" | cat >> hello
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

	modified:   hello

no changes added to commit (use "git add" and/or "git commit -a")
$ gitk&
```

`Git` tells you that you have `changes not staged for commit`: file `hello` state is now set to *modified*.

Tool `gitk` represents this by adding a red patch on top of your last commit. Note the message in place of the log message: `Local uncommitted changes, not checked into the index`.



Notice also that `git` prompts you to either `use "git add <file>..."` to update what will be committed or `use "git checkout -- <file>..."` to discard changes in working directory. Let's try the second option (i.e. discard changes):

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

`Git` has indeed discarded our changes, they are lost,... *forever*.

Most `git` commands are safe, this means that you can not lose anything you are tracking with `git`. But of course, there are exceptions.

`git checkout`, when used with file names, is one such exception. Here, you have explicitly told `git` to replace whatever was in your working directory's version of file `hello` with the last commit version of the same file. You have just what you've asked for.

2.5 Unstage files

Let's re-do our changes and, this time, stage them:

```
$ echo "Bonjour !" | cat >> hello
$ git stage hello
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   hello
```

`Git` prompts us with yet another option: *unstage*. But the syntax of the corresponding command is not the best one could expect !

`git reset` is a very powerful (and pretty evolved) tool that is not part of today's agenda. Actually, you should not use it in any other setting than the one above, i.e. to *unstage* files.
And, by the way, would you not prefer to have an `unstage` command to unstage files ?

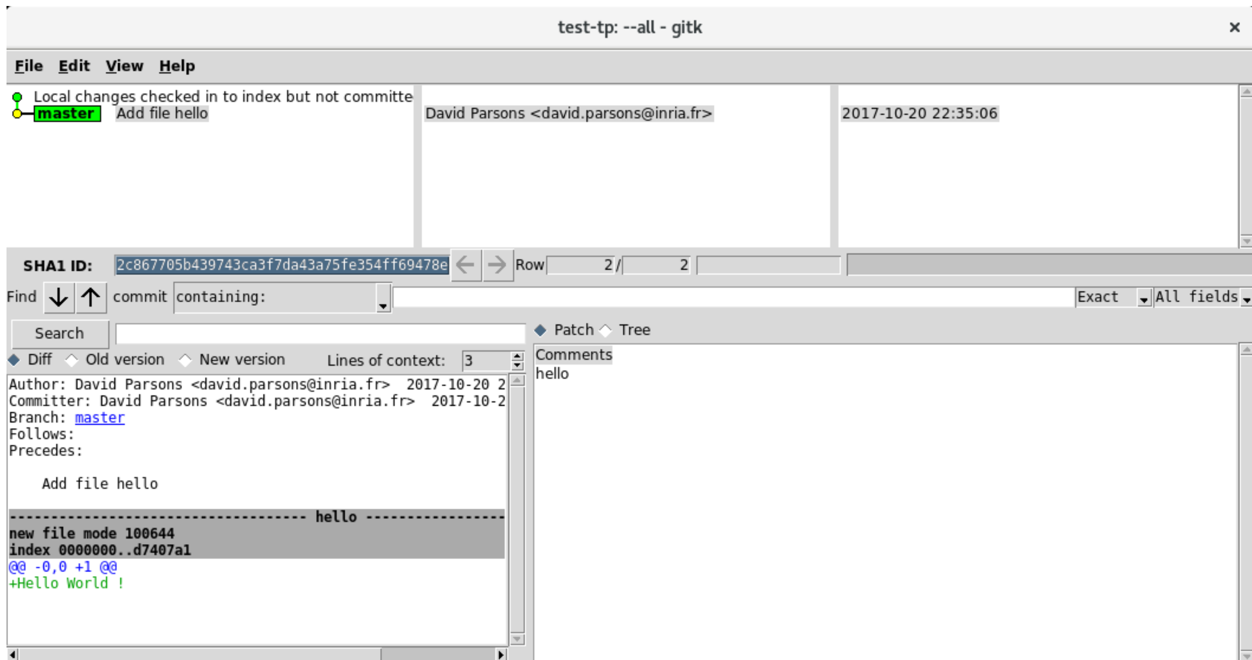
Let's create a new (alias) command and try it out:

```
$ git config --global alias.unstage "reset HEAD --"
$ git unstage hello
Unstaged changes after reset:
M hello
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   hello

  no changes added to commit (use "git add" and/or "git commit -a")
```

2.6 Playing with the index

Again, let's re-stage our changes:

```
$ git stage hello
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   hello

$ gitk&
```



In `gitk`, a set of staged changes is represented as a green patch with the message `Local changes checked in to index but not committed`.

We can continue to edit the same file:

```
$ echo "Buenos dias !" | cat >> hello
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   hello

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   hello
```

You now have 3 different versions of file `hello`: the first one is that of your last (and only) commit, the second one is in your staging area and the third one is in your working directory.

`git diff` allows you to visualize the differences between these versions. Try out the following:

```
$ git diff
diff --git a/hello b/hello
index bd6cb7c..d219b78 100644
--- a/hello
+++ b/hello
@@ -1,2 +1,3 @@
 Hello World !
 Bonjour !
+Buenos dias !
```

```
$ git diff HEAD
diff --git a/hello b/hello
index d7407a1..d219b78 100644
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 Hello World !
+Bonjour !
+Buenos dias !
$ git diff --staged
diff --git a/hello b/hello
index d7407a1..bd6cb7c 100644
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World !
+Bonjour !
```

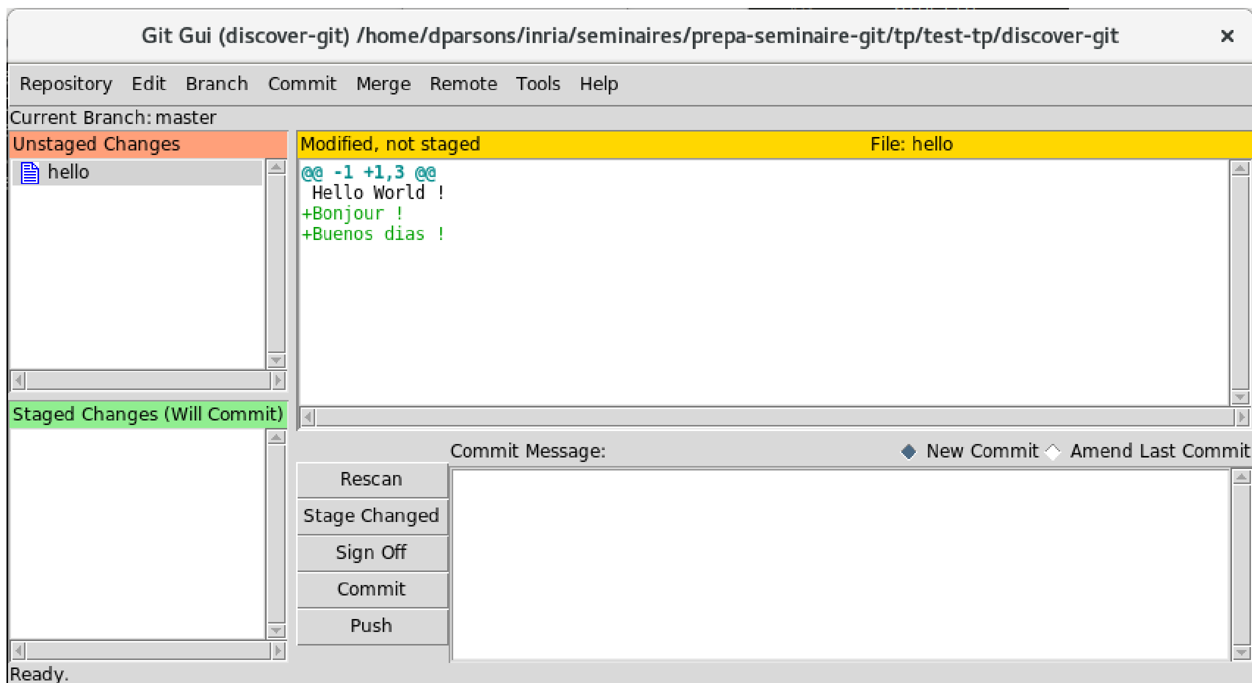
You could also visualize the same information with a graphical tool such as meld with the *difftool* command

2.7 Commit your work using git-gui

Tool `git-gui` is a nice lightweight graphical user interface you can use to prepare your commits and commit them.

First, launch `git-gui`:

```
$ git gui&
```



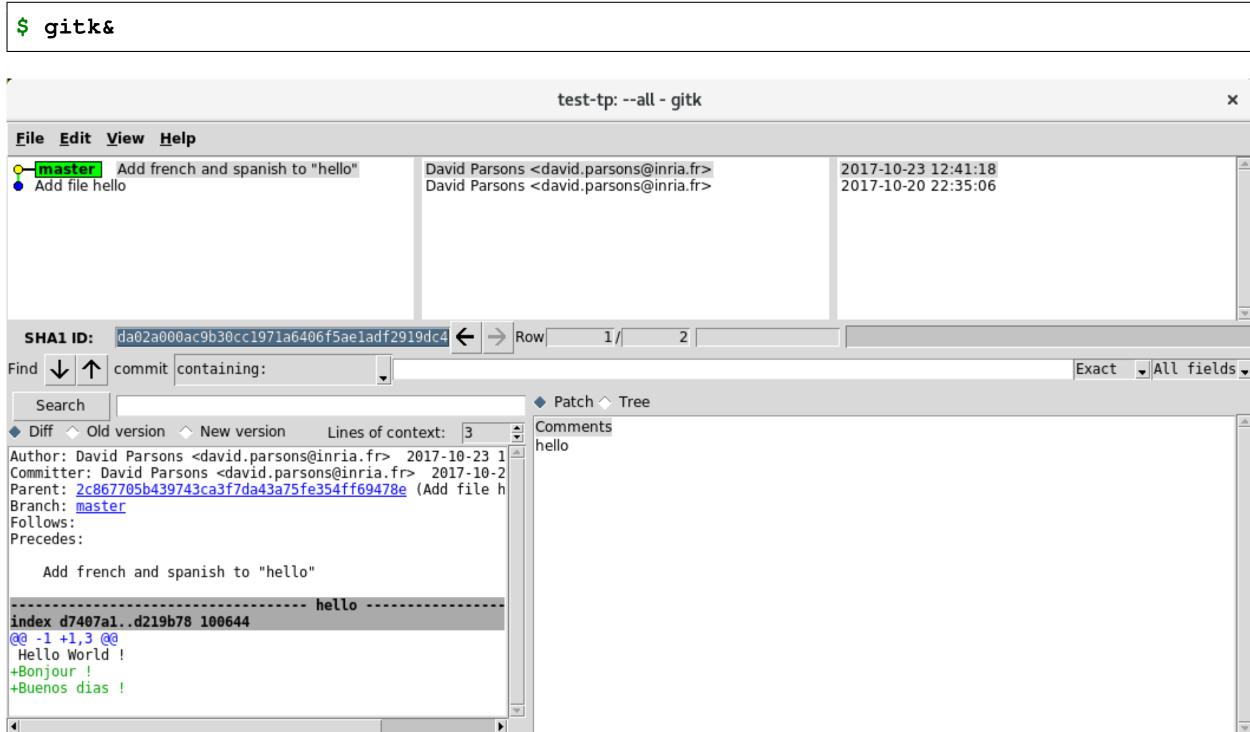
Now, prepare a commit with both the modifications we've made ("Bonjour" and "Buenos dias"). You can do that by clicking on the icon next to the filename `hello` in the upper-left pane (*Unstaged Changes*). This

is strictly equivalent to the command `git stage hello`. Note that file `hello` now shows in the bottom-left pane (*Staged Changes*). You can also visualize what happened using `git status`.

Finally, commit your changes: write your log message in the bottom-right text area and click the *Commit* button.

You should end up with the following result:

```
$ gitk&
```



3 You are not alone

In the first section (see section 2), you've created your own `git` repository and run some basic `git` commands, all of them having local effect only. Everything you did was strictly local to your computer.

Even though `git` does provide you with interesting functionalities when working locally, it is when you are contributing to a project involving more than one people that `git` shows its full potential.

The following series of exercises will illustrate the usage of `git` with a toy `python` development project. To reproduce the workflow of a real development project, attendees will need to work in groups of two: ***developer A*** (***Dev.A***) and ***developer B*** (***Dev.B***), two developers of the toy project.

3.1 Setup

This practical work can be carried out using either gitlab.inria.fr or github.com (inria members should prefer gitlab.inria.fr). However, the choice has to be made on a per-group basis since the chosen platform will be used to share ***Dev.A***'s and ***Dev.B***'s work.

3.1.1 Create your account

Gitlab.inria.fr

- Sign in using iLDAP (`https://gitlab.inria.fr`)
- Create an RSA key pair:

```
$ ssh-keygen -t rsa
```

- Upload your rsa public key (`https://gitlab.inria.fr/profile/keys`)

Github.com

- Create your GitHub personal account (`https://github.com/`)
- Create an RSA key pair:

```
$ ssh-keygen -t rsa
```

- Upload your rsa public key (`https://github.com/settings/keys` → *New SSH key*)

3.1.2 Fork the project

The base project you will build upon is located at `https://gitlab.inria.fr/sed-ral/tpgitsedra` or `https://github.com/david-parsons/tpgitsedra`.
However, you do not have write permission on either of these projects, you will hence need to *fork* one. Of course, since ***Dev.A*** and ***Dev.B*** will be collaborating to the same project, only one of them should create the fork for the group.

Dev.B: Fork the project by clicking on the *fork* button in the upper bar (gitlab) or upper right corner (github) of the project page.

Now ***Dev.B*** has his own copy of the project. In order to give ***Dev.A*** write permissions to the project, ***Dev.B*** needs to add ***Dev.A*** to the project's list of collaborators: *Settings* → *Members* (gitlab) or *Settings* → *Collaborators* (github).

3.1.3 Clone the repo

Finally, both ***Dev.A*** and ***Dev.B*** can *clone* the repository ***Dev.B*** has created.

NB: You will find the url of your git repository by clicking on *Clone or download* → *Clone with SSH*

```
$ git clone <your-git-repo-url>
Cloning into 'tpgitsedra'...
remote: Counting objects: 6, done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 6
Receiving objects: 100% (6/6), done.
$ cd tpgitsedra/
```

3.2 Introduction

The aim of this little toy project is to implement a trivial sphere class. The code is already prepared, as a skeleton with commented code. All you have to do is, when asked so, to uncomment lines of code by removing the character `#` at the beginning of the line. You need to follow the chronological order of the exercises and uncomment code with the corresponding (increasing) step number. There is one important thing you need to know about `python`: it relies on indentation to delimit blocks of code. So, take care to not mess up indentation: use 4 spaces as one level of indentation and do not mix spaces and tab characters, only use spaces.

The best setup to follow this tutorial is to have ***Dev.A*** and ***Dev.B*** next to each other, on two different workstations (it is also possible to share one workstation, though it's less convenient). Both ***Dev.A*** and ***Dev.B*** should have a terminal window opened, as well as the text editor of their choice. Be aware that

some of the **git** operations will change the files in your working copy. Some editors are aware of that and display a warning when trying to edit the obsolete content of a file which has changed on disk, while other editors may not. In any case, you should take care to always refresh the content of the edited files in the editor after **git** operations and before editing them.

3.3 Simple commits / push / pull

When you are ready and inside your freshly cloned **git** repository's working directory, look at the online help of the code:

```
$ ./runsphere -h
usage: runsphere [-h] [-v] [-s] [-d] [--load LOAD] [--save SAVE] [radius]

Sphere properties computation

positional arguments:
  radius         Sphere radius

optional arguments:
  -h, --help     show this help message and exit
  -v             get volume
  -s             get surface
  -d             get diameter
  --load LOAD    load from file
  --save SAVE    save to file
```

Let's instantiate a sphere of radius 1.5:

```
Dev.A$ ./runsphere 1.5
<sphere.sphere.Sphere object at 0x7f3ba927a190>
Traceback (most recent call last):
  File "./runsphere", line 25, in <module>
    print "  radius is %s" % my_sphere.radius
AttributeError: 'Sphere' object has no attribute 'radius'
```

Right now the sphere class is not implemented and doesn't even have a radius. Let's code a bit: ***Dev.A*** starts by uncommenting step 1 in **sphere/sphere.py**, adding a radius member variable to the sphere. Once done, ***Dev.A*** can check the state of his/her working copy:

```
Dev.A$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   sphere/sphere.py

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

        sphere/__init__.pyc
        sphere/sphere.pyc
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

`Git` tells us that we have one file modified in our working copy (`sphere.py`), and two untracked files with the `.pyc` extension, which are generated by `python`.

Dev.A adds the modified file to the `git` index, commits, and pushes the modification to the server repository:

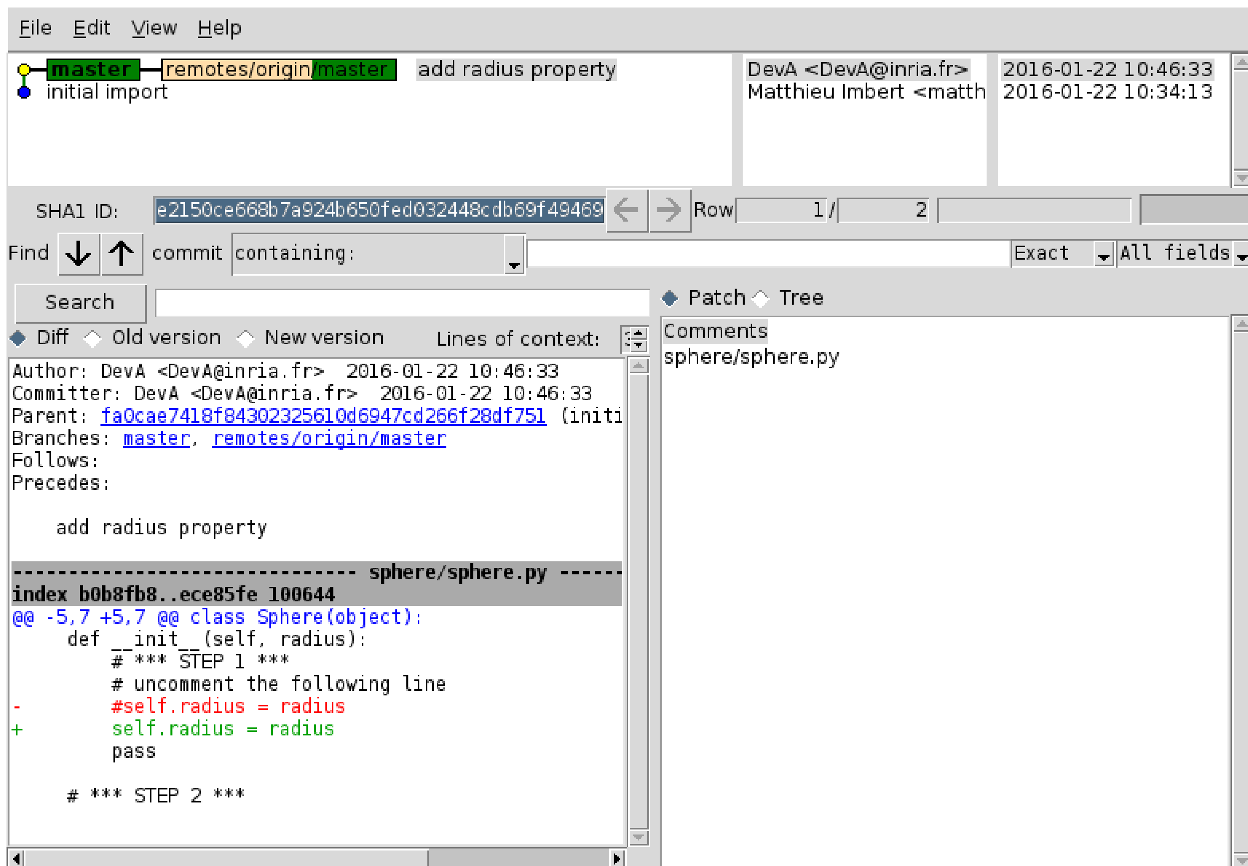
```
Dev.A$ git add sphere/sphere.py
Dev.A$ git commit -m "add radius property"
[master e2150ce] add radius property
 1 file changed, 1 insertion(+), 1 deletion(-)
Dev.A$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 380 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   fa0cae7..e2150ce  master -> master
```

Dev.A can inspect the situation with `git status`, as seen before, or with `git log`:

```
Dev.A$ git log
commit e2150ce668b7a924b650fed032448cdb69f49469
Author: DevA <DevA@inria.fr>
Date:   Fri Jan 22 10:46:33 2016 +0100

    add radius property

commit fa0cae7418f84302325610d6947cd266f28df751
Author: Matthieu Imbert <matthieu.imbert@inria.fr>
Date:   Fri Jan 22 10:34:13 2016 +0100

    initial import
```

Dev.A can also use the `gitk` graphical tool to look at the history. We use option `--all` because it will be easier later to see both our local branch(es) and the remote tracking one(s):

```
Dev.A$ gitk --all &
```



Don't close this `gitk`, we will use it during the whole tutorial session. You can continuously monitor the status of the local repository in `gitk` (hit <F5> to refresh). Let ***Dev.B*** open a `gitk` as well, and keep it open for the whole session:

```
Dev.B$ gitk --all &
```

Now let ***Dev.B*** get ***Dev.A***'s work into his local repository:

```
Dev.B$ git pull
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git+ssh://[...]/tpgitsedra
   fa0cae7..e2150ce  master     -> origin/master
Updating fa0cae7..e2150ce
Fast-forward
 sphere/sphere.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Dev.B hasn't done anything yet, so there's no need to merge concurrent commits, and `git` just needs to perform a `Fast-forward`.

3.4 Working copy and index / tracked and untracked files / .gitignore

Back to ***Dev.A***. When running `git status` we're continuously annoyed by the `.pyc` files. Let's tell `git` to ignore them. For this, add a `.gitignore` file at the root of the local `git` repository, with the following

content:

```
*.pyc
```

Now look at the state of our working copy:

```
Dev.A$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

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

We can see that the `.gitignore` file is already in action (no more mention of untracked `.pyc` files), but it is not yet being tracked by `git`.

Dev.A adds `.gitignore` to the index:

```
Dev.A$ git add .gitignore
```

But before commiting, we realize that we want to add a comment to the `.gitignore` file: Edit it and add a comment line:

```
# ignore python generated files
*.pyc
```

Now let's look at the situation:

```
Dev.A$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitignore

```

This shows us that:

- We have added a first version of `.gitignore` to the index (without the comment)
- We have a modified version of `.gitignore` in our working copy (with the added comment)

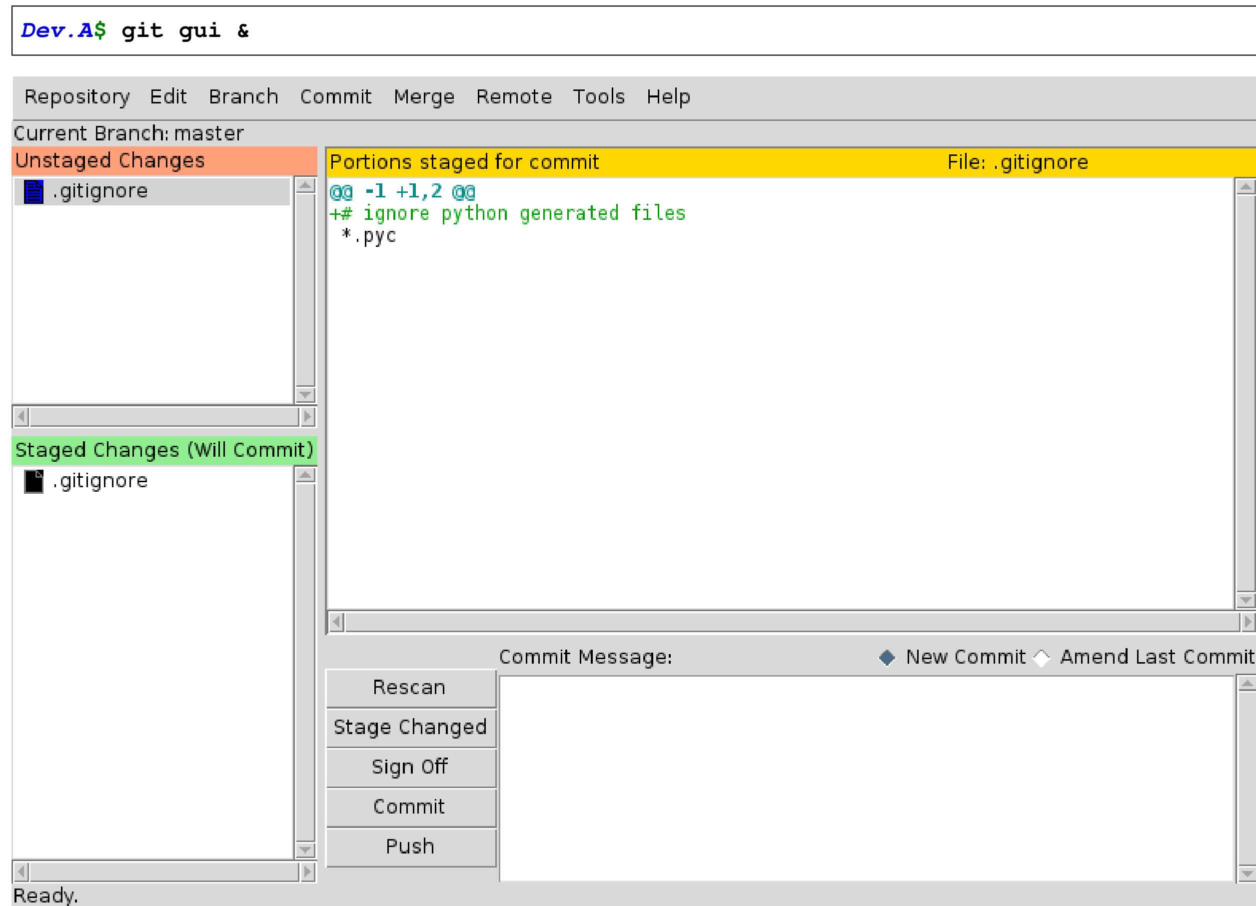
You can try:

- `git diff` to get the diff between the working copy and the index: the comment line.

- `git diff --staged` to get the diff between the index and the current branch HEAD: the `*.pyc` line.
- `git diff HEAD` or `git diff master` to get the diff between the working copy and the HEAD of branch master: the two lines of the `.gitignore` file.

Let's finish by adding ***Dev.A***'s last modification to the index and commit. We could do so the same way as before. Instead, this time, we will use the graphical tool `git gui` to stage our modifications and commit them:

```
Dev.A$ git gui &
```



In this tool, you can stage files / individual hunks / individual lines of codes to be commited. You can click on the `.gitignore` *filename* in both *unstaged changes* pane and *staged changes* pane, to see the diffs. Click on the `.gitignore` *icon* in the *unstaged changes* pane to add the file. Then by clicking on the `.gitignore` filename in the *staged changes*, you can review what will be commited. You can type the commit message "add a git ignore" in the *commit message* pane, and click the *commit* button. Do not click on the push button yet.

After this is done, we get a blank `git gui` window, all panes are empty, which shows us that there is nothing to commit. On the command line:

```
Dev.A$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

`git status` tells us that our working directory is clean, there's nothing to commit, and also that our local branch *master* is ahead of remote branch *origin/master* of the server repository by one commit.

3.5 Merge without conflict

Now, let ***Dev.A*** and ***Dev.B*** work concurrently to develop the Sphere code.

Dev.A implements an `__str__` method (which in `python` allows customizing the way objects print themselves) by uncommenting step 2. At the same time ***Dev.B*** implements the `surface` method by uncommenting step 3.

Dev.B develops, commits and pushes first:

```
Dev.B$ git commit sphere/sphere.py -m "add surface method"
[master 247f173] add surface method
 1 file changed, 1 insertion(+), 1 deletion(-)
Dev.B$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 381 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   e2150ce..247f173  master -> master
```

Dev.A also develops and commits:

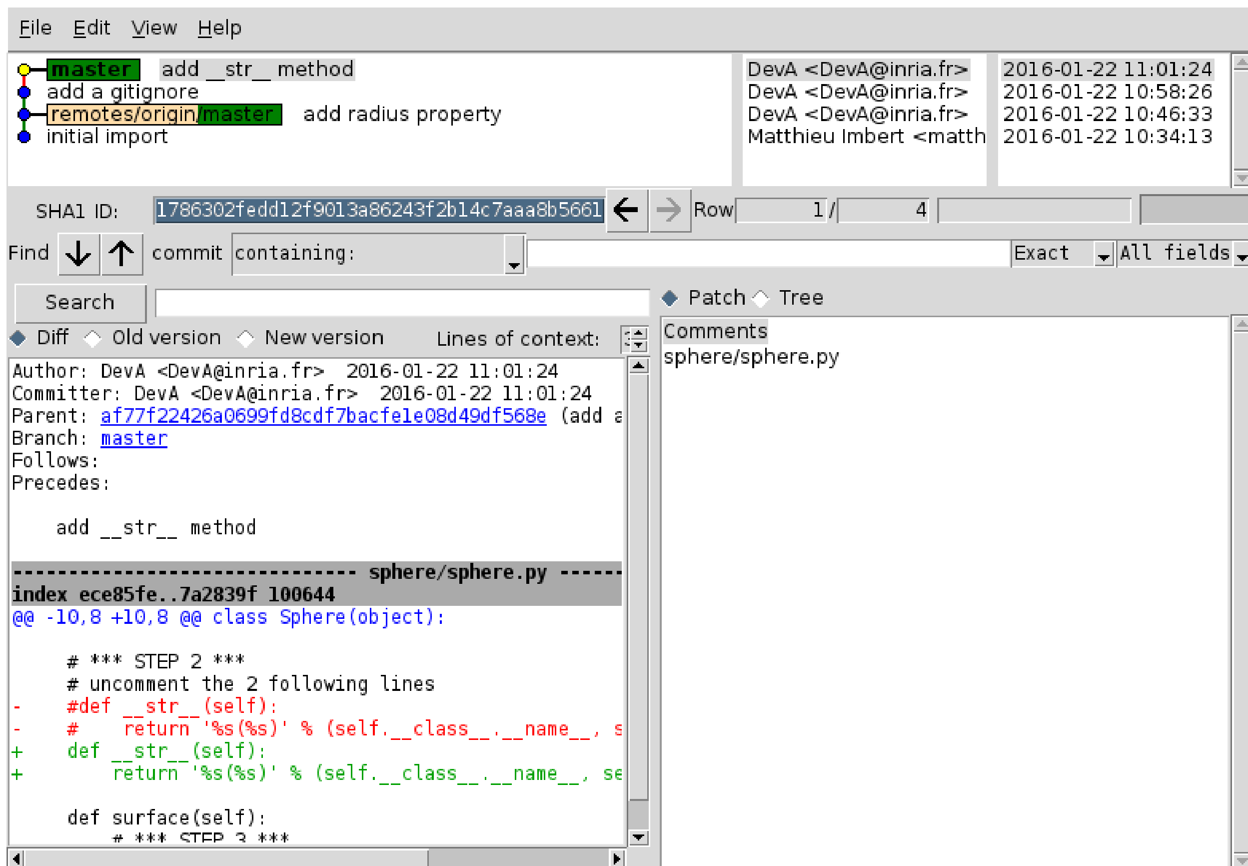
```
Dev.A$ git commit sphere/sphere.py -m "add __str__ method"
[master 1786302] add __str__ method
 1 file changed, 2 insertions(+), 2 deletions(-)
```

At this point ***Dev.A***'s local repository only knows of the state of the server repository at the time when ***Dev.A*** did his last sync operation (here, a `push`). It does not know about what ***Dev.B*** has just pushed. If ***Dev.A*** tries to push:

```
Dev.A$ git push
To git+ssh://[...]/tpgitsedra.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git+ssh://[...]/tpgitsedra.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

`git push` notices that the server repository has changed since ***Dev.A***'s last sync (because ***Dev.B*** pushed to it) and hence fails.

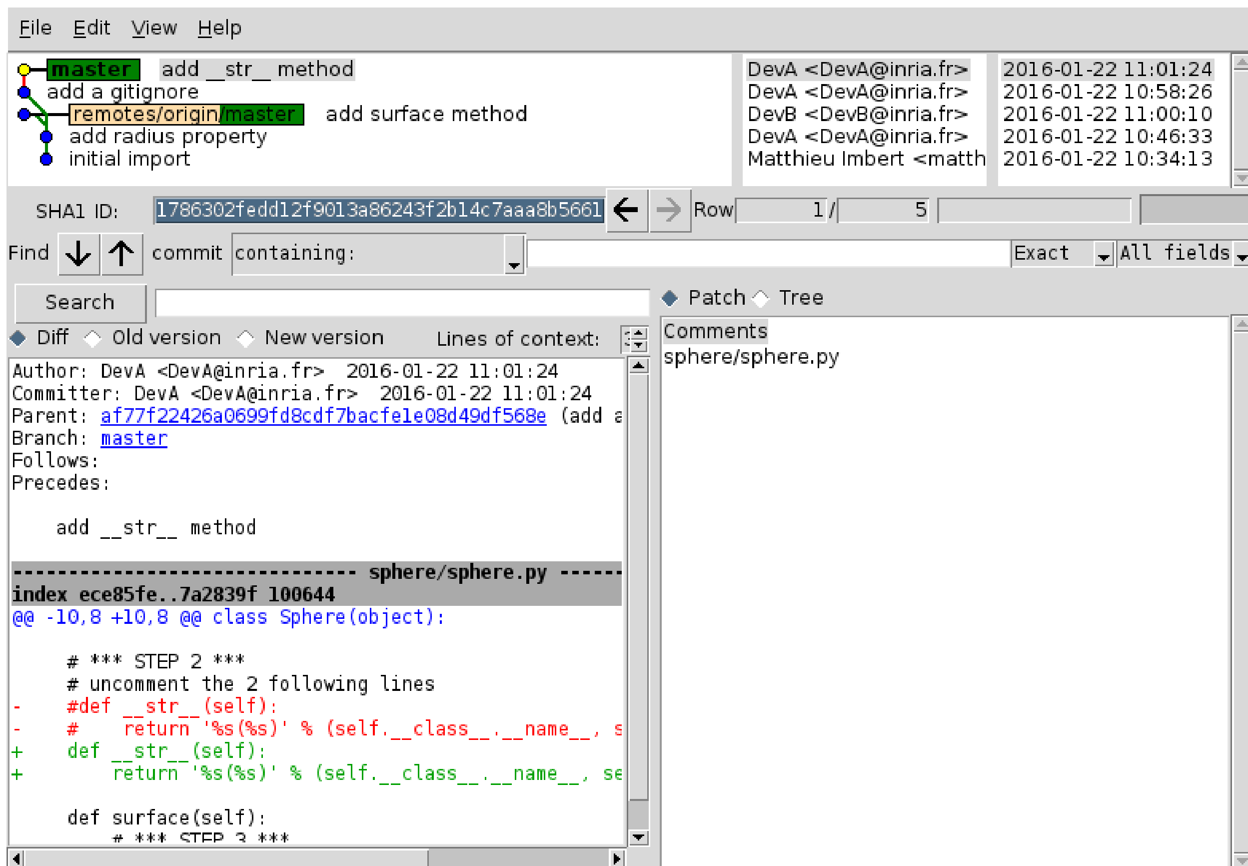
Let's look at the situation graphically: ***Dev.A*** presses key <F5> in `gitk`:



Dev.A can `fetch` the state of the remote branches (from the server repository):

```
Dev.A$ git fetch
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git+ssh://[...]/tpgitsedra
   e2150ce..247f173  master     -> origin/master
```

Now, again, ***Dev.A*** presses key <F5> in `gitk`:



We can see that there are two lines of development:

- branch *master*, our local line of development, not yet pushed to the server repository
- branch *origin/master*, the master branch of the server repository, which has diverged, since ***Dev.B*** has already pushed to the server repository

Dev.A needs to merge his local modifications with the remote ones:

```
Dev.A$ git merge
```

`Git` opens the editor configured as core.editor (default depends on operating system), with a boilerplate text:

```
Merge remote-tracking branch 'refs/remotes/origin/master'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

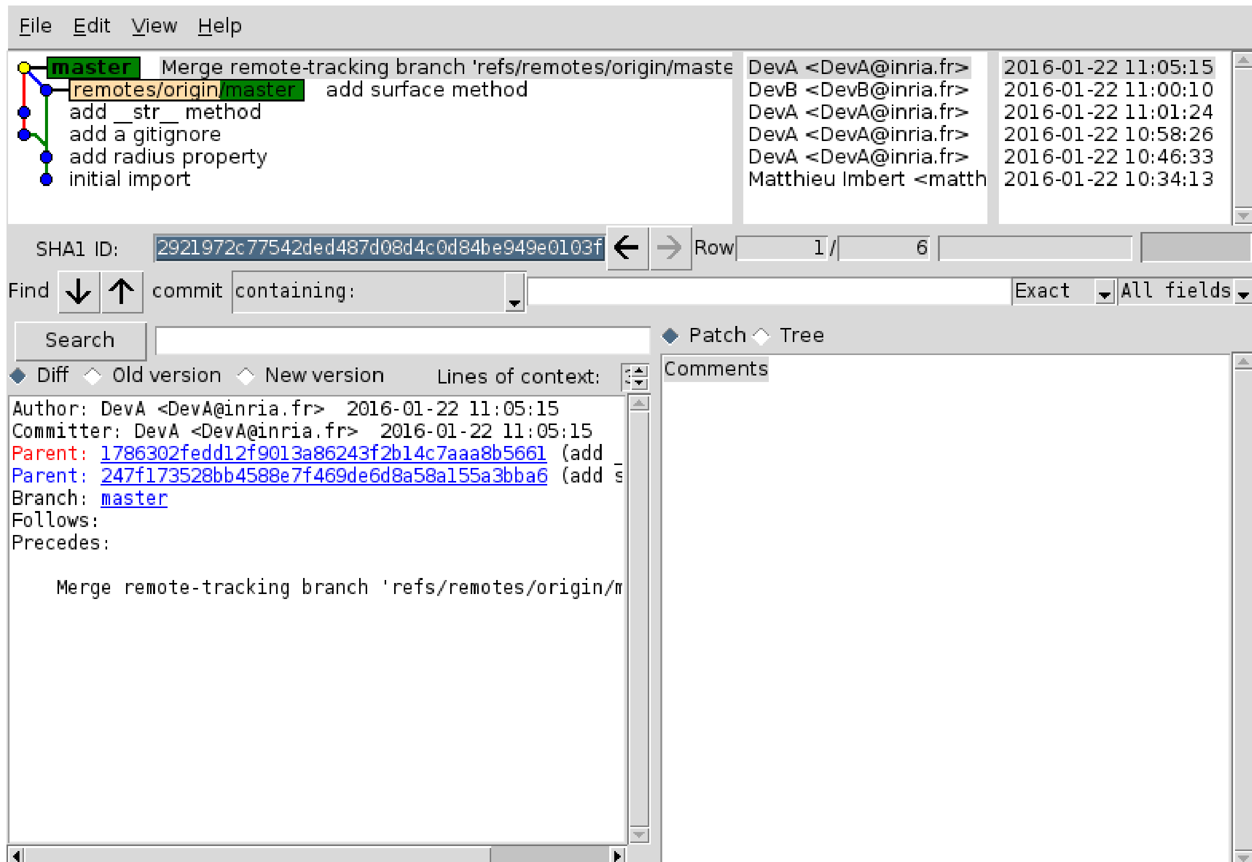
A merge results in a merge commit, so `git` proposes a default merge commit message that you can leave as is or customize. Once you've completed the message, save the comment and quit the editor:

```
Auto-merging sphere/sphere.py
Merge made by the 'recursive' strategy.
```

```
sphere/sphere.py | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

In this case, ***Dev.A*** and ***Dev.B*** modifications are in different parts of the file, so they can be merged automatically.

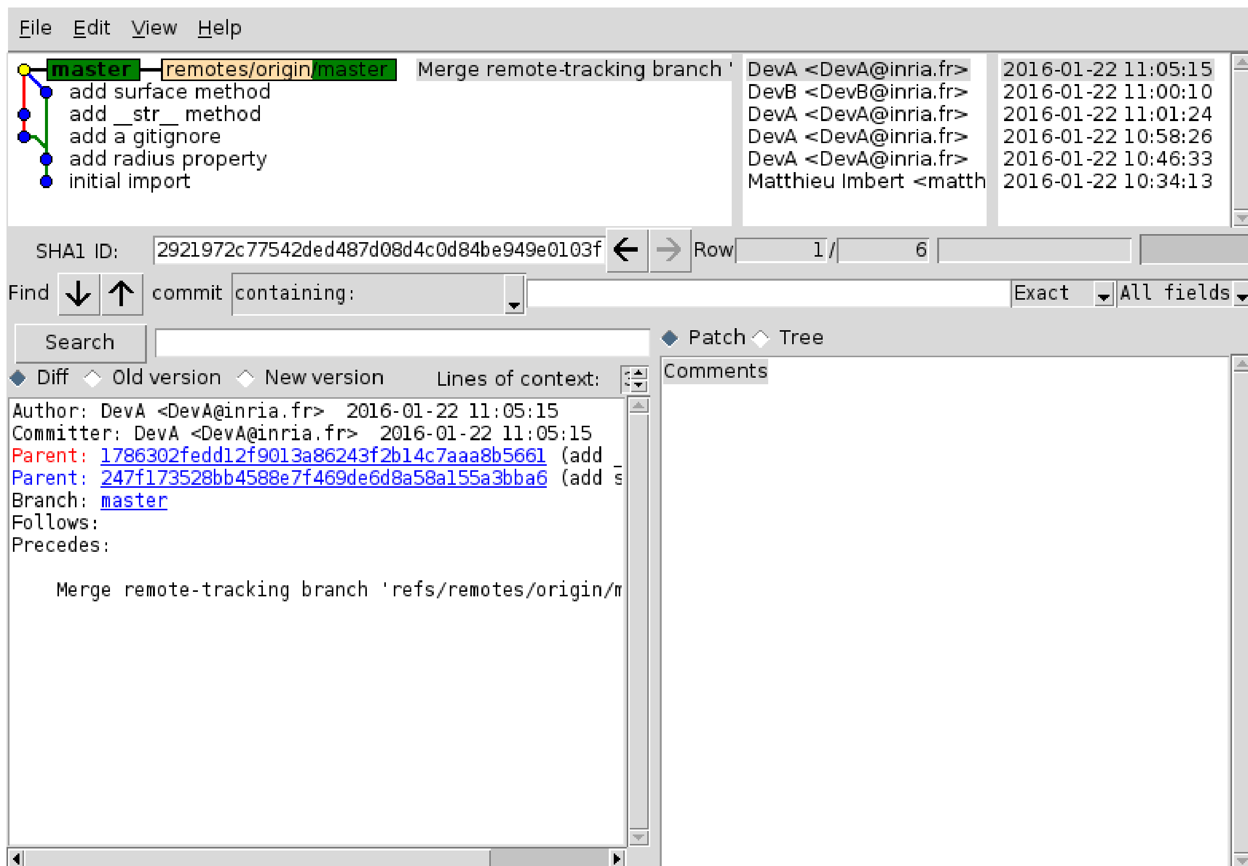
Pressing <F5> in `gitk`:



Now, it's time to push:

```
Dev.A$ git push
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 1.00 KiB | 0 bytes/s, done.
Total 11 (delta 4), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   247f173..2921972  master -> master
```

Pressing <F5> in `gitk`:



And for ***Dev.B***:

```
Dev.B$ git pull
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (11/11), done.
From git+ssh://[...]/tpgitsedra
   247f173..2921972  master     -> origin/master
Updating 247f173..2921972
Fast-forward
 .gitignore        | 2 ++
 sphere/sphere.py | 4 ++--
 2 files changed, 4 insertions(+), 2 deletions(-)
 create mode 100644 .gitignore
```

3.6 Merge with conflict resolution

Now both ***Dev.A*** and ***Dev.B*** start developing concurrently a `volume` method, by uncommenting step 4. This line has a bug: `4/3 * 3.1416` is wrong because from left to right, 4/3 is first computed, and as both operands are integer, the integer division is used, so the result is 1. ***Dev.A*** will use the buggy version while ***Dev.B*** detects this error and fixes it by replacing with `4.0/3.0 * 3.1416`.

Dev.A commits and pushes first

```
Dev.A$ git commit sphere/sphere.py -m "add volume method"
[master 25bcaab] add volume method
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)
Dev.A$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 361 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   2921972..25bcaab  master -> master
```

Dev.B commits, then tries to push, which fails because ***Dev.A*** has pushed meanwhile.

```
Dev.B$ git commit sphere/sphere.py -m "add volume method"
[master 39683f4] add volume method
 1 file changed, 1 insertion(+), 1 deletion(-)
Dev.B$ git push
To git+ssh://[...]/tpgitsedra.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git+ssh://[...]/tpgitsedra.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

So ***Dev.B*** needs to fetch and merge:

```
Dev.B$ git fetch
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git+ssh://[...]/tpgitsedra
   2921972..25bcaab  master     -> origin/master
Dev.B$ git merge
Auto-merging sphere/sphere.py
CONFLICT (content): Merge conflict in sphere/sphere.py
Automatic merge failed; fix conflicts and then commit the result.
```

The automatic merge that we have seen in the previous section is not possible here, since both developers have modified the same portion of code. So this merge needs manual conflict resolution.

There are two ways to manually solve the conflict.

3.6.1 Manual conflict resolution by editing files

The first way is to edit the conflicting file (`sphere/sphere.py`) directly. `Git` puts some markers at conflict(s) location(s):

```
        return 4/3 * 3.1416 * self.radius ** 3
```

- The line(s) between <<<<<<< `HEAD` and `=======` are ***Dev.B***'s version (HEAD on local branch *master*) before the merge

- The line(s) between `=======` and >>>>>>> `refs/remotes/origin/master` are the version in remote branch *origin/master* before the merge

Here, we keep ***Dev.B***'s version, save the file, add it and commit, but **do not push yet**:

```
Dev.B$ git add sphere/sphere.py
Dev.B$ git commit
[master e390720] Merge remote-tracking branch 'refs/remotes/origin/master'
```

3.6.2 Manual conflict resolution with a merge tool

The second way to manually solve the conflict is to use a merge tool, such as `meld`, `kdiff3`, `xxdiff` or `p4merge`. To try this second method, ***Dev.B*** needs to revert his local repository's state to the state it was in before the manual merge, i.e. before the last commit. This can be done with `git reset`. Be sure to check what happens at every step using `gitk --all` (<F5> to refresh) and don't forget the ^ after `HEAD`):

```
Dev.B$ git reset --hard HEAD^
HEAD is now at 39683f4 add volume method
```

Dev.B can see that the previous merge commit is still there, but is now *dangling*, no *ref* points to it. To suppress this dangling commit:

```
Dev.B$ git gc
Counting objects: 34, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (32/32), done.
Writing objects: 100% (34/34), done.
Total 34 (delta 10), reused 6 (delta 0)
```

Here, to really view the result, you need to press <shift-F5> in `gitk`, which completely reloads the *refs*, while <F5> updates the values of the refs, but still displays old refs even if they have disappeared.

Now ***Dev.B*** can try to merge once again:

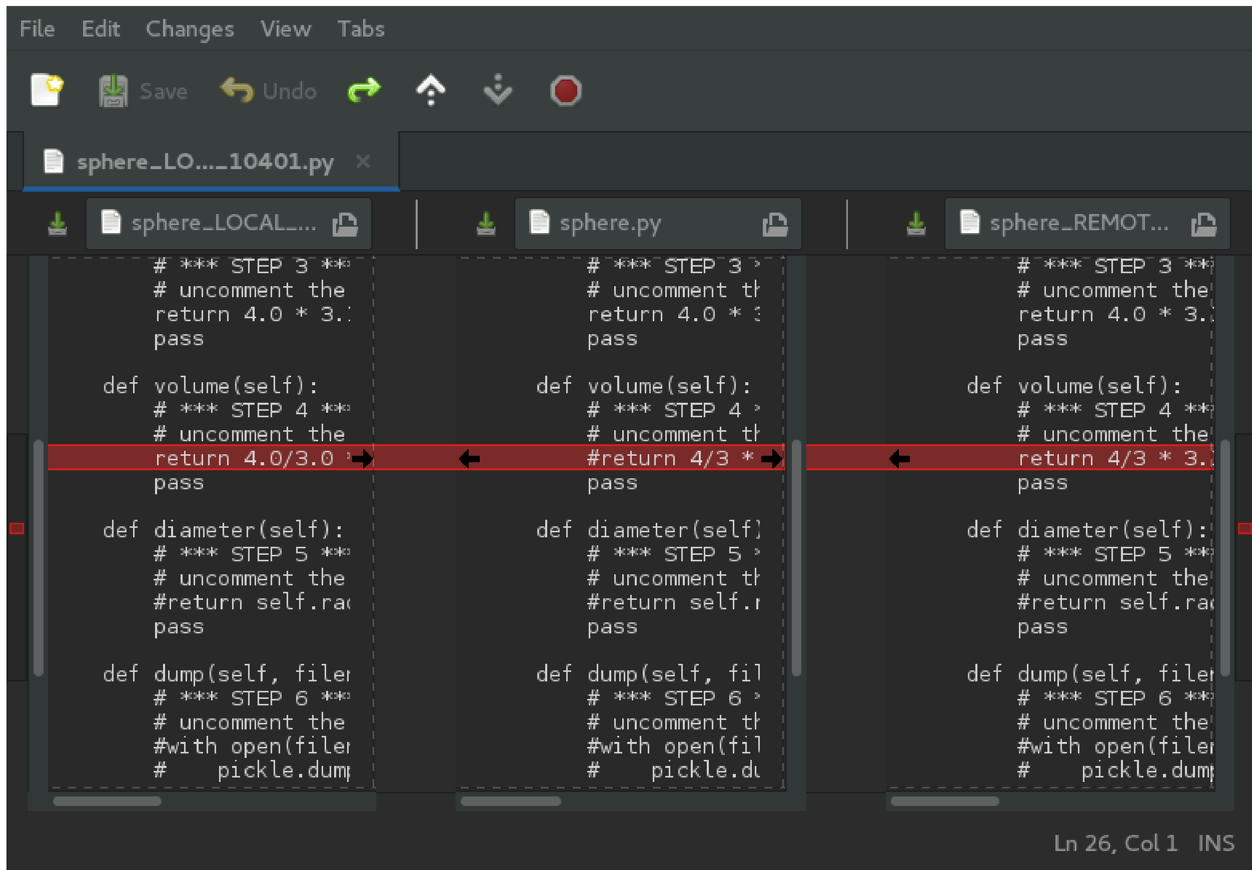
```
Dev.B$ git merge
Auto-merging sphere/sphere.py
CONFLICT (content): Merge conflict in sphere/sphere.py
Automatic merge failed; fix conflicts and then commit the result.
```

To run a graphical merge tool:

```
Dev.B$ git mergetool
```

When you run `git mergetool`, `git` will try to find a merge tool (among a list of supported ones) and automatically launch it for each conflicting file. Edit and save the result for every file.

Here is an example with `meld`:



The exact content of the 3 (or 4) columns of the merge tool depend on the merge tool used. In `meld` for example, with the default configuration, 3 versions of the file are shown:

- your local version on the left (opened read-only)
- the remote version on the right (opened read-only)
- the central column is initialized with the common ancestor of the conflicting file. When pressing save, the central column is saved as the merged file.

You can change this default configuration to suit your personal tastes, for example:

```
$ git config --global merge.tool meld
$ git config --global mergetool.meld.cmd 'meld $LOCAL $MERGED $REMOTE'
```

With this configuration, the center column will not be initialized with the common ancestor, but with the merged file.

Here's another configuration example, if you want to use p4merge (*e.g.* on windows):

```
$ git config --global merge.tool p4merge
$ git config --global mergetool.p4merge.path "C:/Program Files/Perforce/p4merge.exe"
```

`Git` automatically adds the resolved files to the index. This can also be changed with:

```
$ git config --global mergetool.meld.trustExitCode false
```

In this case, git will ask if the merge was successful after closing the merge tool.

Finally, you need to commit:

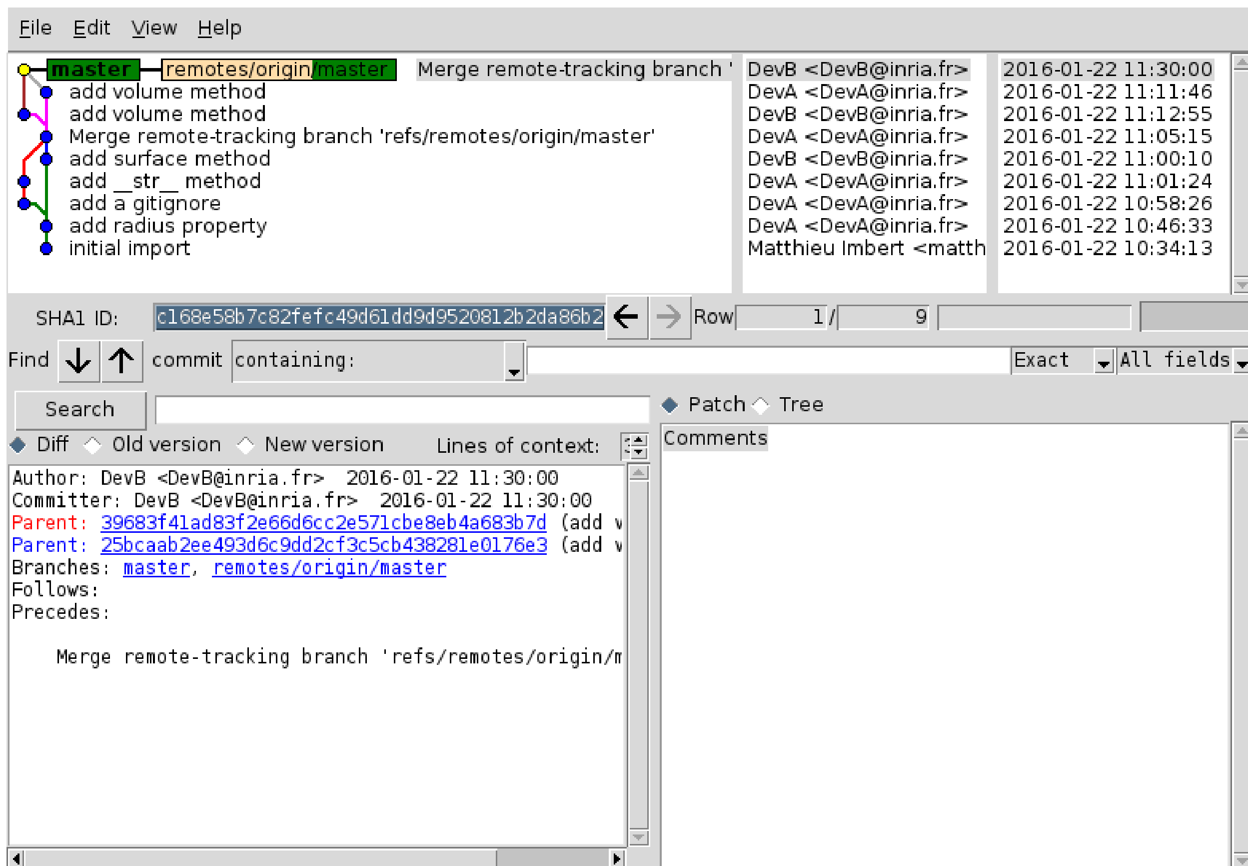
```
Dev.B$ git commit
[master c168e58] Merge remote-tracking branch 'refs/remotes/origin/master'
```

3.6.3 Pushing the resolved conflict

```
Dev.B$ git push
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 571 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 2 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   25bcaab..c168e58  master -> master
```

```
Dev.A$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From git+ssh://[...]/tpgitsedra
   25bcaab..c168e58  master     -> origin/master
Updating 25bcaab..c168e58
Fast-forward
 sphere/sphere.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Here's how it looks in ***Dev.A***'s `gitk` once ***Dev.B*** has pushed the merge and ***Dev.A*** has pulled it:



3.7 Branches

Dev.B wants to start a branch named `use_math_constants` to refactor the code to use the constant `math.pi` instead of 3.1416 everywhere in the code. Right now, ***Dev.B*** plans this branch to remain local. This is just an area for a short work which is planned to be merged in `master` and published once finished.

Dev.B can create the branch with `git branch use_math_constants` then switch to it with `git checkout use_math_constants`. Or do both in one command:

```
Dev.B$ git branch
* master
Dev.B$ git checkout -b use_math_constants
Switched to a new branch 'use_math_constants'
Dev.B$ git branch
  master
* use_math_constants
```

Now ***Dev.B*** replaces 3.1416 by `math.pi` in method `surface`, saves the file, and commits:

```
Dev.B$ git add sphere/sphere.py
Dev.B$ git commit -m "use math.pi in surface"
[use_math_constants 91e2897] use math.pi in surface
 1 file changed, 1 insertion(+), 1 deletion(-)
```

At the same time ***Dev.A*** implements method `diameter` by uncommenting step 5 in the code (The bug in the diameter computation is intentional, don't fix it yet). ***Dev.A*** then saves the file, commits, and pushes:

```
Dev.A$ git commit sphere/sphere.py -m "add method diameter"
[master 0af2d68] add method diameter
 1 file changed, 1 insertion(+), 1 deletion(-)
Dev.A$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 365 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   c168e58..0af2d68  master -> master
```

Dev.B can merge ***Dev.A***'s work in his `master` branch (a *fast-forward* in this case):

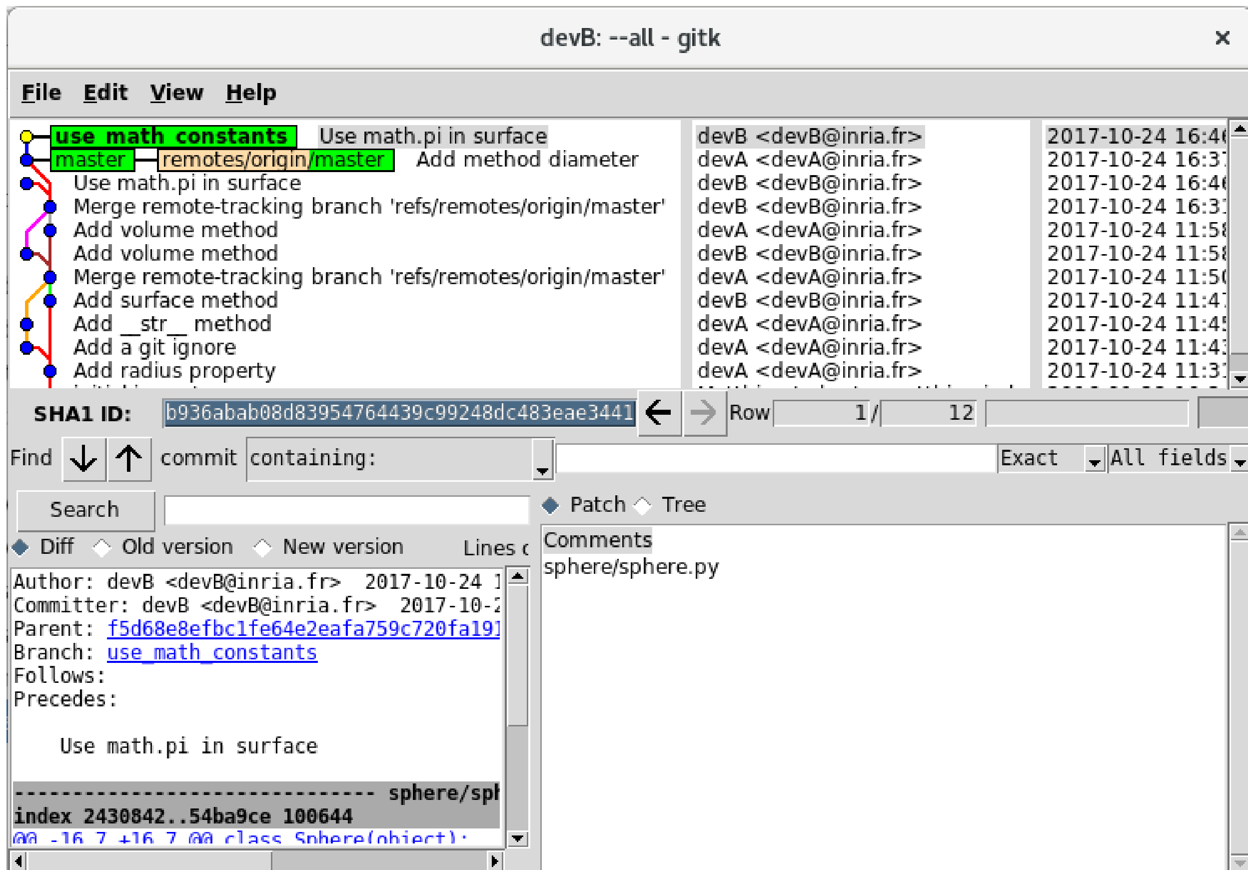
```
Dev.B$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
Dev.B$ git pull
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git+ssh://[...]/tpgitsedra
   c168e58..0af2d68  master     -> origin/master
Updating c168e58..0af2d68
Fast-forward
 sphere/sphere.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Then ***Dev.B*** wants to update his `git branch use_math_constants` with the new version: In this situation ***Dev.B*** can avoid a merge and rebase instead:

```
Dev.B$ git checkout use_math_constants
Switched to branch 'use_math_constants'
Dev.B$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: use math.pi in surface
```

Rebasing branch `use_math_constants` is a rewrite of its history. This can be done here with no harm because this is a local branch. This would be an issue on a published branch, so keep in mind that even though rebasing is a very useful tool, published commits should never be rebased.

In the following screenshot, you may notice that our original commit (third line) whose parent was the last merge commit is now dangling (no branch is pointing to it either directly or indirectly) and was replaced by a brand new commit with the same commit message but whose parent is the commit with message "Add method diameter"



Dev.B now replaces 3.1416 by `math.pi` in method `volume`, saves the file and commits:

```
Dev.B$ git commit sphere/sphere.py -m "use math.pi in volume"
[use_math_constants 76e5ff6] use math.pi in volume
 1 file changed, 1 insertion(+), 1 deletion(-)
```

At the same time ***Dev.A*** wants to start a branch named `serialization` to add support for saving / loading `Sphere` instances to disk. ***Dev.A*** doesn't know yet if it's a good idea, so this branch may just be a sandbox to explore a new idea. It may later be abandoned, remain local for a long time, be merged in `master`, or be published as a separate branch to allow for other developers to contribute to it. There is no need to decide it right now. This branch will stay local to ***Dev.A*** unless explicitly pushed to the server repository (see later).

```
Dev.A$ git checkout -b serialization
Switched to a new branch 'serialization'
Dev.A$ git branch
  master
* serialization
```

Dev.A implements method `dump` by uncommenting step 6. ***Dev.A*** then saves the file and commits:

```
Dev.A$ git commit sphere/sphere.py -m "add method dump"
[serialization 764feed] add method dump
 1 file changed, 2 insertions(+), 2 deletions(-)
```

At this point ***Dev.A*** and ***Dev.B*** share a cup of coffee and talk about their code. ***Dev.B*** thinks that his branch `use_math_constants` is ready to be merged in `master`. ***Dev.B*** also notices the bug in implementation of method `diameter` and opts to fix it. Finally, ***Dev.A*** and ***Dev.B*** agree to publish branch `serialization` on the server repository so that ***Dev.B*** can work on it (without merging it in `master` yet).

First ***Dev.A*** publishes his branch `serialization`

```
Dev.A$ git push --set-upstream origin serialization
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 360 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
 * [new branch]      serialization -> serialization
Branch serialization set up to track remote branch serialization from origin.
```

Dev.B fixes method `diameter`, by returning `self.radius * 2` and saves the file.

At that point in time, ***Dev.B*** realizes he is on branch `use_math_constants` while he intended to commit this fix on branch `master`. He thus tries to checkout master:

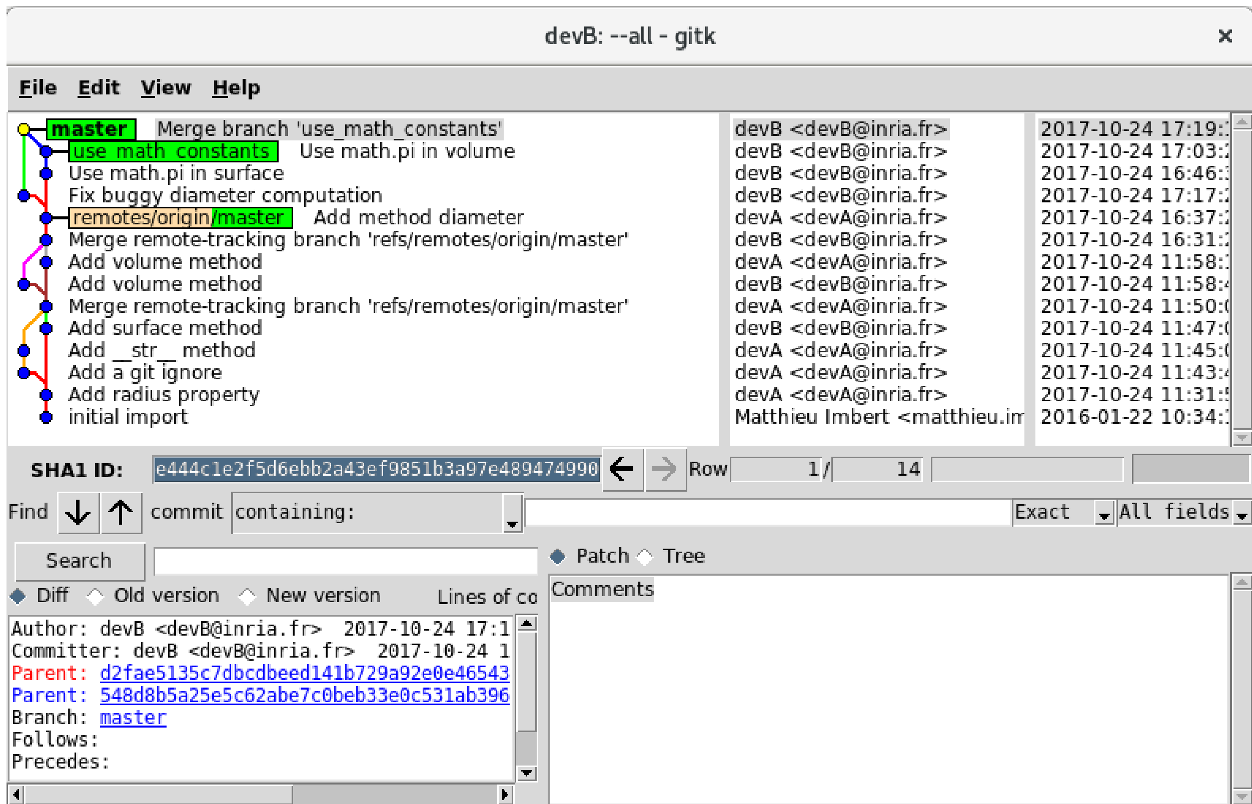
```
Dev.B$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
        sphere/sphere.py
Please, commit your changes or stash them before you can switch branches.
Aborting
```

This is a classic situation in `git`, when you have saved a file while being on a branch (`use_math_constants`) but what you really want is to apply the modification to another branch (`master`). As the message says, you can stash your modifications before switching branch, and then apply the stash. Or, more rapidly, you can use option `-m` to `checkout` the other branch while retaining the modifications in your working copy:

```
Dev.B$ git checkout -m master
M        sphere/sphere.py
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
Dev.B$ git commit sphere/sphere.py -m "fix buggy diameter computation"
[master 74fde24] fix buggy diameter computation
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Dev.B then wants to merge branch `use_math_constants` in `master`.

```
Dev.B$ git merge use_math_constants
Auto-merging sphere/sphere.py
Merge made by the 'recursive' strategy.
 sphere/sphere.py | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```



Dev.B now pushes:

```
Dev.B$ git push
Counting objects: 12, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (12/12), 1.07 KiB | 0 bytes/s, done.
Total 12 (delta 6), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   0af2d68..dbcd3e0  master -> master
```

Dev.B now wants to get branch `serialization`:

```
Dev.B$ git checkout serialization
error: pathspec 'serialization' did not match any file(s) known to git.
```

Dev.B needs to first fetch the server repository state:

```
Dev.B$ git fetch
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git+ssh://[...]/tpgitsedra
 * [new branch]      serialization -> origin/serialization
```

Now ***Dev.B*** can switch to branch `serialization`:

```
Dev.B$ git checkout serialization
Branch serialization set up to track remote branch serialization from origin.
Switched to a new branch 'serialization'
```

Dev.B adds function `loadSphere` by uncommenting step 7, saves the file, commits and pushes:

```
Dev.B$ git commit sphere/sphere.py -m "add function loadSphere"
[serialization a2db349] add function loadSphere
 1 file changed, 3 insertions(+), 3 deletions(-)
Dev.B$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 376 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git+ssh://[...]/tpgitsedra.git
   764feed..a2db349  serialization -> serialization
```

Dev.A fetches all branches:

```
Dev.A$ git fetch --all
Fetching origin
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 16 (delta 8), reused 0 (delta 0)
Unpacking objects: 100% (16/16), done.
From git+ssh://[...]/tpgitsedra
   764feed..a2db349  serialization -> origin/serialization
   0af2d68..dbcd3e0  master      -> origin/master
```

Here is the result in ***Dev.A***'s `gitk`:

devA: --all - gitk

File Edit View Help

remotes/origin/serialization Add function loadSphere	devB <devB@inria.fr>	2017-10-24 17:26:27
serialization Add method dump	devA <devA@inria.fr>	2017-10-24 17:05:00
remotes/origin/master Merge branch 'use_math_constants'	devB <devB@inria.fr>	2017-10-24 17:19:14
Use math.pi in volume	devB <devB@inria.fr>	2017-10-24 17:03:24
Use math.pi in surface	devB <devB@inria.fr>	2017-10-24 16:46:38
Fix buggy diameter computation	devB <devB@inria.fr>	2017-10-24 17:17:21
master Add method diameter	devA <devA@inria.fr>	2017-10-24 16:37:28
Merge remote-tracking branch 'refs/remotes/origin/master'	devB <devB@inria.fr>	2017-10-24 16:31:20
Add volume method	devA <devA@inria.fr>	2017-10-24 11:58:17
Add volume method	devB <devB@inria.fr>	2017-10-24 11:58:42
Merge remote-tracking branch 'refs/remotes/origin/master'	devA <devA@inria.fr>	2017-10-24 11:50:09
Add surface method	devB <devB@inria.fr>	2017-10-24 11:47:03

SHA1 ID: e93a32876ae10b3227cd2edf744678af567073ee Row 2 / 16

Find commit containing: Exact All fields

Search Patch Tree

Diff Old version New version Lines of cont

Comments

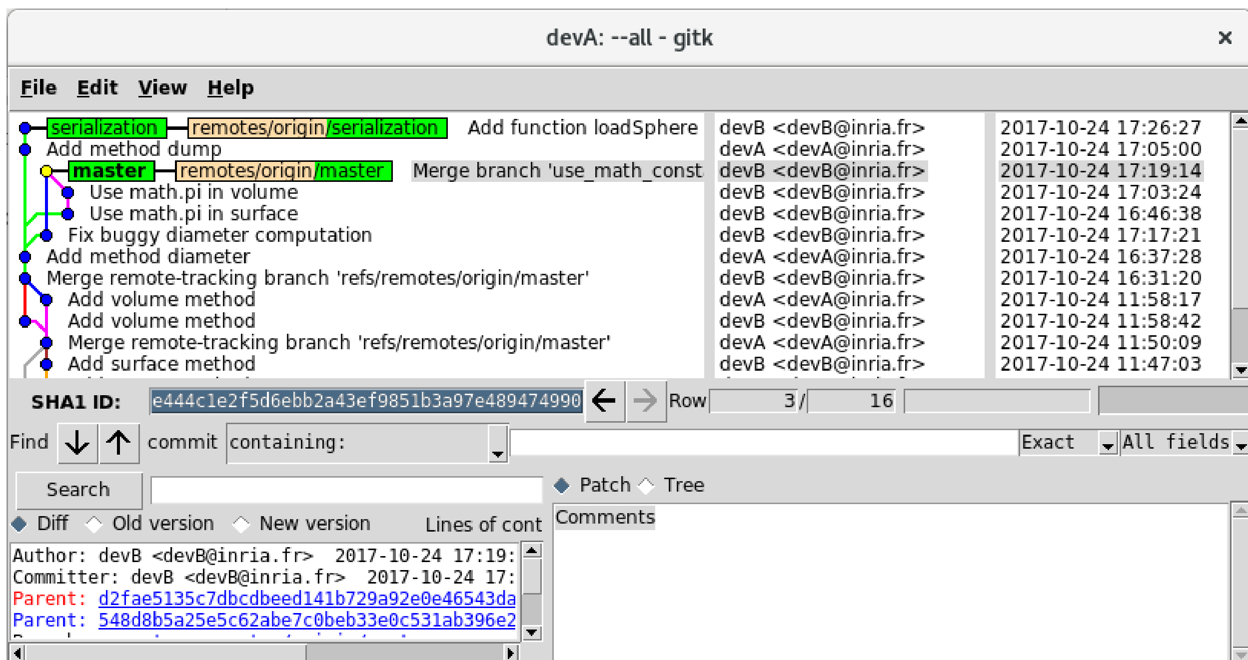
sphere/sphere.py

```
Author: devA <devA@inria.fr>  2017-10-24 17:05:
Committer: devA <devA@inria.fr>  2017-10-24 17:
Parent: f5d68e8efbc1fe64e2eafa759c720fa1910bd35
Child:  9550cf915497227e25a51aff23afb1bf7d18dbc
```

Dev.A can now update their branch `serialization` with the one from the server, and update their branch master with the one from the server:

```
Dev.A$ git branch
  master
* serialization
Dev.A$ git merge
Updating 764feed..a2db349
Fast-forward
 sphere/sphere.py | 6 +++---
 1 file changed, 3 insertions(+), 3 deletions(-)
Dev.A$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 3 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
Dev.A$ git merge
Updating 0af2d68..dbcd3e0
Fast-forward
 sphere/sphere.py | 6 +++---
 1 file changed, 3 insertions(+), 3 deletions(-)
```

Here is the final result in ***Dev.A***'s `gitk`:



4 Conclusion

You have performed a lot of basic and reasonably advanced `git` operations in a scenario with a central `git` repository.

To go further, you can study a few things that we did not cover here:

- Moving or removing files
- Stashing

- Tagging
- Cherry-Picking
- Rewriting history (interactive rebasing)
- Checkout an explicit commit (leading to a *detached head* situation)
- Bisecting