

LARICS git tutorial

Juraj Oršulić, Damjan Miklić

LARICS Lab
FER, University of Zagreb

December 2018



- 1 Preparation
- 2 The basic basics
- 3 Working with branches
- 4 Where to go from here?



A note on notation

Throughout the presentation, the following notation applies:

- Commands that you are supposed to type are displayed in monospace font preceded by a `>` symbol, such as
`> git --help`
- The `>` symbol only indicates the command prompt.
Do not type it in.
- When the command is too long to fit in one line, the line break will be escaped with a backslash (`\`). You do not need to type in the backslash and the line break.
- The text that you need to replace is given `<inside angle brackets>`, e.g.,
`> cd /home/<your username>`

When typing in the command with your replaced text, omit the brackets.



Preparing for the tutorial

- Open a [GitHub account](#) and set up an SSH key for passwordless login according to [these instructions on Github](#)
- Install the git command-line client and GUI tools

```
> sudo apt install git git-gui gitg
```
- Tell git who you are, and which editor to use:

```
> git config --global user.name "<Your Name>"
> git config --global user.email <youremail@fer.hr>
> git config --global core.editor \
    "gedit --wait --new-window"
```
- The code example in the exercise will be available in C++ and Python. If you will be using C++, install the build tools:

```
> sudo apt install build-essential cmake
```



Tutorial organization

- Work in pairs
- Pair up with someone using the same programming language (C++ or Python)
- When assigning tasks and describing how you will interact with each other through git, we will refer to you, the members of a pair, as *Mirko* and *Slavko*. Make an agreement about who will assume each role.



Version control system

- VCS - version control system

Basic features

- Keeps track of changes to our code
- Facilitates collaboration in software and documentation development

Additional requirements

- Work offline
- Support a distributed, decentralized workflow
- Compatibility with existing protocols (e.g. ssh, http)
- Data integrity protection
- Efficiency

About git

- Developed in 2005 by Linus Torvalds (in a single weekend!) to support Linux kernel development
- Very powerful, has a reputation of being hard to learn
 - We hope we will convince you otherwise :)
- Supports different workflows
 - We'll be using the GitHub workflow (more or less)

Meaning of the name *git*

git means "unpleasant person" in British slang. Linus: "I'm an egotistical bastard, and I name all my projects after myself."

From the readme:

- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "g*dd*mn idiotic truckload of sh*t": when it breaks

Basic terms: repository and commit

Repository (repo)

A place where your work is kept. It contains your code and its complete history, stored as a collection of commits.

Commit

A basic unit of work in a project. Contains a **snapshot** of the complete project, a reference to a previous snapshot (the *parent commit*), the commit message – a textual description of the changes in the commit (with respect to the parent commit), the commit author name, and the commit date.



Commit ID (SHA)

Commit ID

Every commit is marked with an alphanumeric identifier (SHA-1 hash) generated from the above information, which is used to uniquely identify the commit.

For example: `bdfa760c07d8f621ff603a2dc5d6de810cd62e88`

You can also use a prefix of the identifier to refer to this commit, usually 5 or 7 characters long, e.g. `bdfa760`.

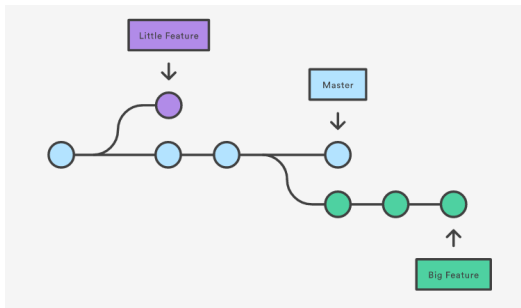


Basic terms: branch, master and head

Branch, master and head

A **branch** is simply a pointer to a commit. **Master** is usually the name of the main branch (but does not have to be). **HEAD** is a special pointer to the currently checked out commit.

The commits with their parent-child relationships form a directed acyclic graph (DAG).



Getting started on Github


To begin working on a project using Github, you have to create a repository. There are two ways to do this:

- Create an empty repository
 - When starting from scratch
- Fork another repository
 - When you wish to improve and build upon another repository
 - This creates your personal copy of the repository in which you can make your own commits




Making a Github fork and adding a collaborator

Task A [Mirko]: Fork the example repository

To fork a repository, simply click the  icon in the top right corner of the repository webpage. For this tutorial, you will need to fork the [larics/git-tutorial-code](#) repository.

Task B [Mirko]: Add a collaborator to your forked repository

In the GitHub user interface, on the  Settings tab of your forked repository, select the Collaborators and teams menu and add your partner as a collaborator with Write access.

Task C [Slavko]: Accept the collaboration invitation

Check your mail, or the list of notifications on Github. Accept the collaboration invitation.

Cloning the repository

Cloning

Cloning creates a local copy of the repository, which includes all the commits in the repository – the whole history.

```
> git clone git@github.com:<mirko>/git-tutorial-code.git
```

Task D [Mirko, Slavko]

Clone Mirko's repository onto your computer.

- NB – if you committed something, it is in your local repo, (almost) permanently. Do not make it a habit to delete your local repositories – if you "lose" a commit, it can be recovered (by its ID), even if you have deleted the branch!



Repository structure

What is contained in the repository directory?

```
> cd git-tutorial-code  
> ls -la  
> git status
```

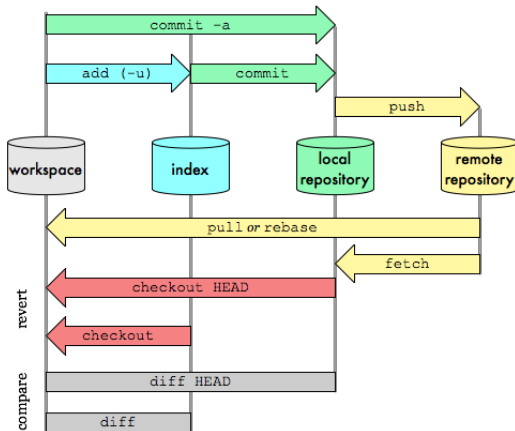
- The *working tree* – current version of files ("checked out")
- The hidden `.git` folder which contains repository metadata (all the commits)
- The `git status` command provides an overview of what is going on in the git repository.



Visualizing git operation

Git Data Transport Commands

<http://osteele.com>



Your first commit

Task E [Mirko, Slavko]

Open README.md and add the following lines, then save the file:

Maintainers:

<your name>

Run the following, and observe what is happening.

```
> git status
> git diff
> git add README.md
> git status
> git gui
> git commit -m "Add maintainer."
> git status
> git log
```


Making a commit - recap

- 1 Make a change in the working tree. For example:
 - Edit a file
 - Create a file
 - Delete a file
 - Move a file (git considers moving as deleting + creating a new file)
- 2 `git add` the change to the *staging area* ("index")
- 3 Perform a final inspection of the staged changes
 - `git gui` is handy for this
 - The staged changes should make a logically grouped set of changes
 - Feel free to make as many commits as you like
 - "Commit early and often"
- 4 `git commit` your changes
 - "50/72" rule - the title of the commit description should be ~50 characters, the body should be wrapped to 72 characters
 - Use the imperative form as the first word in the title – e.g.
"Add prime checking", "Implement saving to file", "Fix broken build"



Managing staged changes

- To get a list of staged and unstaged files, run `git status`.
- To unstage all changes, run `git reset`.
- To view the **unstaged** changes in the *diff* format, open `git gui`, or run `git diff`
- To view the **staged** changes in the *diff* format, you can also use `git gui`, or you can run `git diff --cached`
- You can finely tune what goes into the commit by staging/unstaging individual lines using `git gui`.
 - The shortcut for (un)staging all changes in a file is Ctrl-U
 - For those who prefer the command line, you can use the `-p` switch of `git add` and `git reset`.



Making some more commits

Task F [Mirko, Slavko]: Adding a license and .gitignore

- Add a license to the project. Create a LICENSE.txt file and copy the [Apache License 2.0 text](#) into this file.
- Create a .gitignore file, with the following two lines:
cpp/build/
*.pyc
- After modifying the files in the working tree, make two separate commits, one for the license, and the other for .gitignore.

Task G [Mirko]

Open README.md. Under the package description (the second line), add and commit something to annoy your colleague (but keep it friendly :)). For example:

Developed without any help from that guy Slavko!

Discarding unstaged changes

Task H [Mirko, Slavko]

- Delete a big chunk of text from README.md. Do not add or commit.
- To find out what is going on, use `git status`.
- Discard the changes by retrieving ("checking out") the last committed version of the file (can also be a directory) using the `checkout` command:

```
> git checkout README.md
```

Note that the checked out version will contain the staged changes.



Discarding many unwanted changes

- If you have made many changes in your working tree, spanning several files, you can discard all of them at once (this also includes the staged changes):

```
> git reset --hard
```
- This will restore all tracked files in the working tree to the most recently committed version (i.e. the HEAD commit).

Task I [Mirko, Slavko]: Discarding several unwanted changes

- Delete several files from the cloned example repository. Do not commit the changes.
- Check the output of `git status`.
- Discard all changes as shown above.

Going back in time

- A big point in using a version control system such as git is the ability to retrieve older versions of files in the project.
- Note that writing good commit messages is important, because it will make identifying the right version much easier.
- In the workflow we have described, the procedure for undoing mistakes is making additional commits which fix these mistakes.



Going back in time - 2

Task J [Mirko, Slavko]

- Delete a big chunk of text from `LICENSE.txt`.
- Commit your destruction of the license file.
- Git allows you to retrieve a version of a file from an earlier commit. To identify the desired commit, you can:
 - use the `gitg` tool
 - run `git log` (add `--oneline` for an abbreviated list of commits).
- Perform the retrieval by running:

```
> git checkout <commit ID> <file>
```

Git will place the checked out version of the file in the working tree, and will also stage the changes for you.
- Don't forget to specify the file, as `checkout` with a commit ID, but without a file, means something else!

Pushing the changes to the remote repository

- `commit` only saves changes **locally**!
- The command `git push` is used to upload the commits you have made to a branch in a remote repository (also known as just *remote*). In this case, the master branch on the origin remote:

```
> git push origin master
```

- Note that you can simply call `git push` if you have set up your local branch to be a tracking branch.
- This will be explained later; in most cases, git will have already automatically configured this for you.

Task K [Mirko, Slavko]

Try pushing the commits you have made to Mirko's repository.

Dealing with conflicts

Collaboration issues

What happens when somebody else pushed changes to the remote repository before us? In this case, git refuses to push. We must synchronize our local repository first!

Synchronizing the remote changes is a two-step process:

- 1 fetch the changes from the remote repository
- 2 merge the changes into your local branch

```
> git fetch
> git status
> git diff master origin/master
> git merge origin/master
```

Understanding diff

`git diff a b` shows changes that need to be applied to a to make it the same as b. a and b are references to any two commits.

Dealing with conflicts: merging

- When you call `merge`, git adds a special commit known as a *merge commit* to your local branch
- The merge commit has two parent commits: your local previous head commit, and the other side's commit (in this case, the *other side* is the same branch in the remote repository).

`fetch+merge=pull?`

- `pull` will do `fetch` and `merge` in a single step
- If the remote version has changed, and you have made local commits before pulling the remote changes, you might get yourself into trouble.^a
- When you are getting back to work on a branch that you share with someone else, pull the remote changes before making your own commits! This helps avoid unnecessary merge commits by fast-forwarding.

^aA really nice article advocating the use of `fetch+merge` can be found on [Mark's Blog](#)

Resolving conflicts

After merging, the file ends up in a conflicted state:

```
> git merge origin/master  
> git status  
> git gui
```

Conflict markers inside the file:

```
<<<<<<< HEAD
```

```
Code from the checked out branch ("local" in git gui)
```

```
=====
```

```
Code from the other branch ("remote" in git gui)
```

```
>>>>>>> origin/master
```

To resolve the conflict, manually edit the file, mark resolution with git add, commit and push:

```
> git add <file name>  
> git commit  
> git push origin master
```



Dealing with conflicts: merging

Task L [<the loser>]

- Synchronize the remote changes.
- Resolve the conflict (if any).
- Push your changes.

Task M [<the winner>]

- Synchronize the new remote changes that <the loser> has now managed to push.



Reverting previous commits

- Another benefit of splitting the work in different commits is the ability to undo them using `git revert`.
- To revert the offending commit:

```
> git revert <commit ID>
```
- Reverting generates a *revert commit*, which has the exactly opposite (inverse) set of changes.

Task N [Slavko]: Slavko gets his revenge

- Revert Mirko's annoying commit.



Amending the last commit

In the special case when you want to add additional changes to the *last commit* that you have made, **and if you have not yet pushed that commit**:

- 1 Stage the additional changes using `git add`
- 2 Inspect the changes, as previously described
- 3 Run `git commit --amend`
 - In case you only wish to amend the commit description, just run `git commit --amend`, without staging any changes.

Task O [Slavko]

Amend the description of the revert commit that you have just made:
Revert Mirko's rudeness. Push the revert commit.

Task P [Mirko]

Pull Slavko's changes.

Visualizing your repository

- The `gitg` tool can visualize your repository
- GitHub provides similar functionality with the Graphs → Network menu.

Task Q [Mirko, Slavko]

Visualize your repository with `gitg` and on GitHub. Notice how the history is not linear.



Binary files

Difference between text and binary files

- Changes to files are stored incrementally, as commit diffs, which is very space-efficient for text-files.
- For binary files, most of the time, the majority of the file is changed (e.g. when you edit a picture, or recompile an executable), which effectively means that the commit **contains a complete copy of the new version**.
- The old version **still persists** in the old commits, **even if you remove the file**, because git keeps the whole history.

How to handle binary files

Storing binary files (e.g. graphics) is acceptable if they are small and change infrequently. Otherwise, create a README file with instructions for downloading the files, or a script which downloads them into place.

Binary files - build output

Build output

- **Never** commit build output (even if it is text, e.g. documentation)! It can waste **huge** amounts of space and it will create unnecessary conflicts
- Add the appropriate entries to `.gitignore` so that git ignores the build output
 - This will also keep `git gui` and the output of `git status` free from clutter related to build files



Branches in git

- Branches are used for isolated experimenting (short-lived branches), and isolated feature development (long-lived branches)
- Branching often is encouraged
- Git is specifically designed for efficient work with branches
- A branch is **simply a pointer to a commit**
- The default branch name is **master**

When do I need a branch?

- When you are beginning to work on a new feature
- When you want to make experimental commits
- When you want to single out a change (can be several commits) for making a pull request
- Whenever you feel like it!

Creating a branch

Creating a branch:

```
> git branch <branch name>
```

Switching to a branch:

```
> git checkout <branch name>
```

Or, in one command:

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

Listing local branches:

```
> git branch -v
```

Two uses of checkout

- Notice that we have already used `checkout` for checking out an earlier version of a file. This does not affect the head commit in any way.
- The `checkout` command is also used for switching to another branch, which of course changes the head commit (and the files in the working tree)!

Working on a branch

Naming branches

Branches are usually named with dashes, e.g.

"pose-update-optimization", and can be prefixed with a name if the repository is a shared and one person is working on them (e.g. "matt-new-renderer").

Task R [Mirko]: Bugfix on a new branch

There is a bug in the `fact` function of the demo shell program.

- Create and checkout a new appropriately named branch.
- Fix the bug
 - C++: also change `short` to `long long` in a separate commit
- Commit.



Merging branches locally

After you are finished working on the bugfix, it's time to merge it back into the master branch.

```
> git checkout master  
> git merge <branch name>
```

After you have merged the branch, perform cleanup by deleting it:

```
> git branch -d <branch name>
```

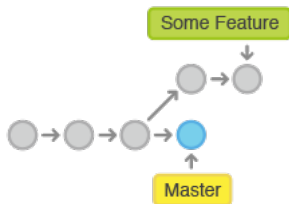
To delete an unmerged branch, you have to use the `-D` option.

Merge conflicts are handled in the same way as discussed before.

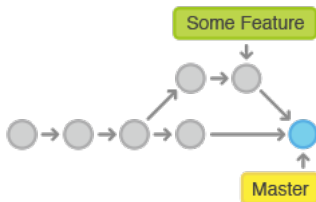
Remote branches at origin are also just special kinds of branches, so we have been working with branches all along :)



Before Merging



After a 3-way Merge



Task S [Mirko]: Merge the bugfix

Merge your bug fix to the `master` branch and then push the `master` branch. Have your colleague pull the changes before they continue working.



Pushing a new branch to a remote

- As with commits, newly created branches and the commits in them are (so far) stored only in the local repository
- Uploading them to the remote repository is performed by pushing.
- Newly created branches have to be pushed with the following command:

```
> git push -u <remote> <branch name>
```

This sets up the local branch as a *tracking branch* of the remote branch.

- `git status` will now show you how many new commits ahead the branches are one related to another
- `git push` will work without having to specify the remote
- When you `clone` a repository, this is automatically set up for you for the `master` branch, and for any existing branches that you check out



Pushing a new branch to a remote

Deleting remote branches

- Can be done from the Github interface;
run `git fetch --prune` afterwards to clean them up in your local copy of the remote repository
- To delete a remote branch using the command-line interface, run `git push --delete <remote> <branch_name>`

Task T [Slavko]: New feature on a remote branch

- Create a new branch in which you will implement the feature
- Implement a `square` command for the demo shell program, which computes the square of the provided argument
- After making the commits which implement the feature, push the feature branch to the `origin` remote.

Pull requests

Task [Slavko]: Pull request

- Look for your branch in the web interface of the GitHub repository and create a pull request for the master branch of your repository.
- Have Mirko review the pull request.
- If they find issues, correct them by pushing additional commits in the branch. The pull request will be automatically updated.
- Delete the branch after it has been merged.

Code review

Pull requests are an efficient and transparent code review mechanism. Code review is good. Pull requests are good. Use pull requests :)



Working with multiple remotes

Why would I need multiple remotes?

- We can get code changes from any repository, not just our remote repository (which is automatically set up as a remote named `origin` when cloning)
- A typical example is getting changes from the *upstream* repository, i.e., the repository that we forked.

Listing and adding remotes:

```
> git remote add <remote_name> git@<hostname>:<path to repo>
> git remote -v
```

We can now work with the new remote in the same way as with `origin` (except we can't push into it!), e.g.:

```
> git fetch <remote_name>
> git merge <remote_name>/<branch>
```



Merging upstream changes

Task: Merge upstream changes

- Set up a remote called `upstream` pointing to [the original repository you forked](#)
- `fetch` the `upstream` repository and compare its `master` branch with your `master` branch.
- Optional: for experimenting how the upstream changes will play with your changes, create a new branch.
- Merge the upstream `master` branch into your `master` branch (or first into your experimental branch, and then merge your experimental branch into your `master`).



Tips, further reading and useful links

- Install `zsh` and [oh-my-zsh](#), which provide awesome tab completion for `git` (branch names, commit IDs, `git` command options...)
- When you have mastered the merge workflow from this presentation, read up on the rebase workflow described at the [LARICS git tutorial](#), which enables you to produce much cleaner and linear commit history with fewer merges

Some additional literature:

- Every `git` command supports the `--help` option, e.g.,
 `> git help`
 `> git branch --help`
- [Stackoverflow](#) :)
- [The Pro Git book](#) (The `git` reference)
- [git - the simple guide](#) (sort of an online cheatsheet)
- [Git Magic](#) (tutorial)
- [Git Guys](#) (tutorial)



- Super cool simulation for practicing advanced git branching:
<https://learngitbranching.js.org>
- If you have any questions, feel free to send an email to juraj.orsulic@fer.hr, or drop by LARICS (C-XI-16).

