Introduction to Version Control with Git and GitHub

CoDaS-HEP school (2019) at Princeton University

David Luet

Princeton University
PICSciE, Research Computing/OIT, Geosciences

July 22, 2019

Outline

- Version Control Systems (VCS)
  - Definition
  - Distributed VCS
- Git locally
  - Git Concepts
  - Git Basic Usage
  - Introduction to Branching
  - Merging branches
  - Branch Management
  - Inspecting Changes
  - Working with Older Versions of the Repo: Undoing Changes
  - Conclusion
- Remote Repositories
  - Introduction
  - Creating a Remote Repository From `git_cheatsheet`
  - Working With Remote Repositories
  - Conclusion
- References
  - Git Remote Repository Hosting
  - Graphical User Interfaces

Outline

Version Control Systems (VCS)

Git locally

Remote Repositories

References

Appendices

Outline

Version Control Systems (VCS)
- Definition
- Distributed VCS

What does a version control system do?

A Version Control System (VCS) is a system that:

- records changes to a file or set of files over time.
- so that you can recall specific versions later.

But what can it do for me?

A VCS allows you to:

- revert specific files or the entire project back to a previous state.
- compare changes over time.
- see who made the changes.

It's not just for source code

- Typically, the files under version control are software source code.
- But you you can do this with nearly any type of file on a computer.
  - LaTeX files.
  - text files.
  - configuration files.
  - input files for a code.
- Word of caution: most VCSs don't handle binary files well.

Using VCS is liberating

- It makes it easy to recover from mistakes.
- It remembers what you did for you.
- Best of all: it’s not that hard to use.

I already have a system: I keep my files in time stamped directories

- Copy files into another directory or make a tarball.

```
src_implicit
src_explicit_works
src_explicit_fails
src_20170501.tgz
```

- This approach is very common because it is so simple.

But it’s incredibly error prone

- Too easy to accidentally overwrite or delete files.
- Complex to manage.
- Hard to collaborate with others.
- You are better off learning good habits now.

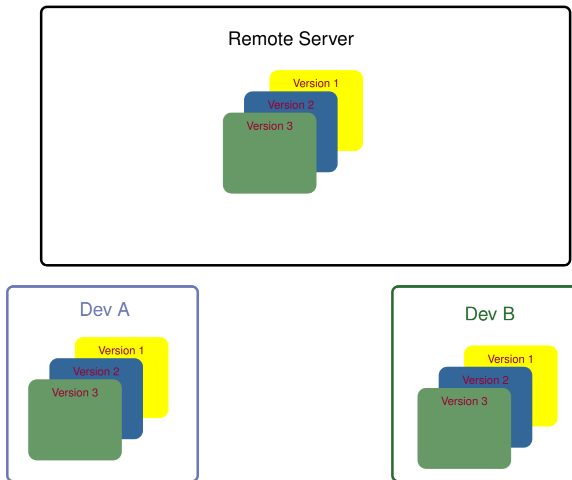
Outline

Version Control Systems (VCS)
- Definition
- Distributed VCS

Definition: repository

A repository is a set of files and directories that describes every version of the files under version control.

Distributed VCS (Git, Mercurial, Bazaar or Darcs)



- Developers have a fully mirror of the repository.
  - It contains all every versions of every files in the repository.
- Every clone is really a full backup of all the data.

Advantages of DVCS

- You don't need an internet connection to interact with the repo.
- Duplication: if any server dies any of the client repositories can be copied back up to the server to restore it.
- Supports multiple remote repositories:
  - so you can collaborate with different groups of people.
  - Differentiate between private and public work.

Outline

Version Control Systems (VCS)

Git locally

Remote Repositories

References

Appendices

Outline

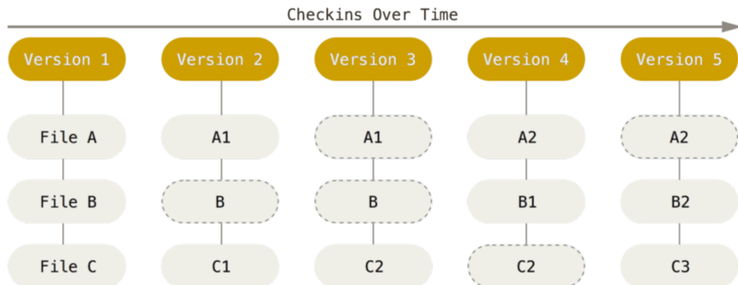
Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

Git history

Created in 2005 by Linus Torvalds, the creator of Linux, with the goals of:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Git stores snapshots of the files



Each snapshot is called a commit.

Nearly every operation is local

- Most operations in Git only need local files and resources to operate:
  - generally no information is needed from another computer on your network.
- You don't need an internet connection to:
  - browse the history of the project.
  - commit your changes.

Commit ID

- Git uses a checksum mechanism called `SHA-1` hash to differentiate and name the commits.
- Think of it as a unique label for a snapshot.
- The SHA-1 hashes are 40 characters long but you only need the first few (e.g. 7) to differentiate the commits uniquely.

```
$ git log --oneline
ed46b26 (HEAD -> master) Add adding_newfile to the index
5ba1206 Add new file and fix getting_repo.md
c1b0ca8 Initial commit
```

```
$ git show
commit ed46b26cd7cd2b2541c89f4a49799f2222f2c04b (HEAD ->
Author: David Luet <luet@princeton.edu>
Date:   Wed Apr 19 20:19:20 2017 -0400
Add adding_newfile to the index
```

Git generally only adds data

- When you do actions in Git, nearly all of them only add data to the Git database.
- It is hard to get the system to do anything that is not undoable or to make it erase data in any way.
- After you commit a snapshot into Git, it is very difficult to lose it, especially if you regularly push your database to another repository.
- You can experiment freely without the fear of screwing up.
- But you can lose or mess up changes you haven't committed yet.

States of files in the current directory

- Each file in your current directory can be in one of two states:
  - untracked.
  - tracked.
- Tracked files are files that were in the last snapshot then can be in three states:
  - unmodified or committed: latest version in Git.
  - modified: file has changed since last commit.
  - staged: marked to be introduced in the next commit (snapshot).

Stagging Area - A Photography Analogy

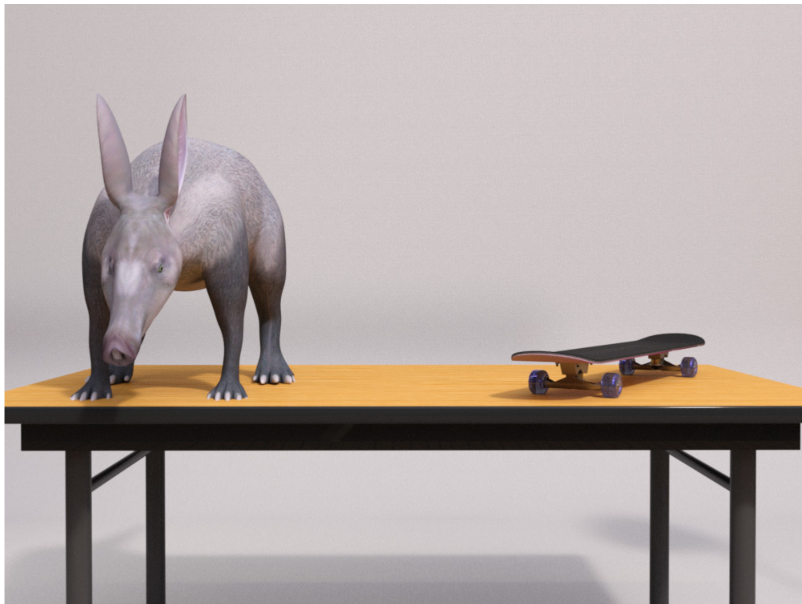
Stagging Area - Crafting Your Photography [commit]

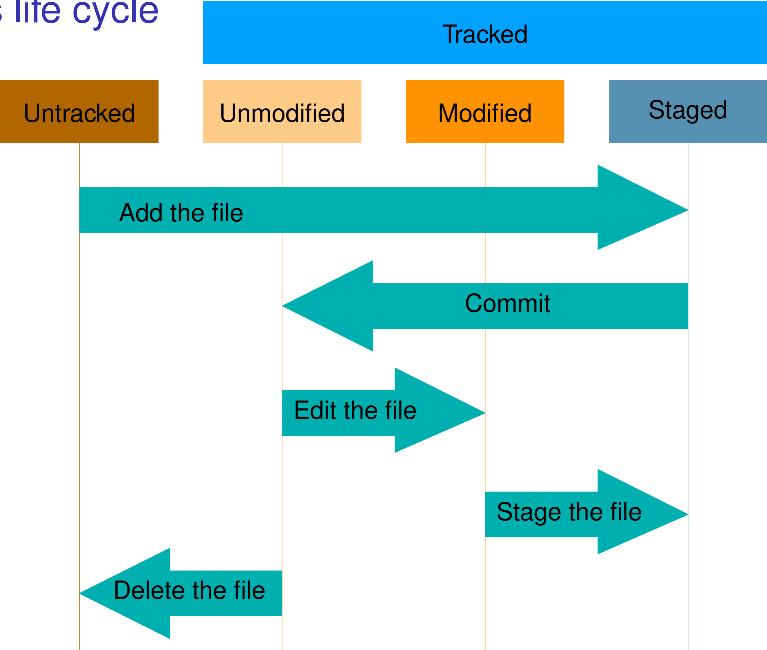
Stagging Area - Adding Objects [changes]

Stagging Area - Adding More Object [changes]

Stagging Area - Removing an Object [changes]

Stagging Area - Taking the Photo/Snapshot [commit]



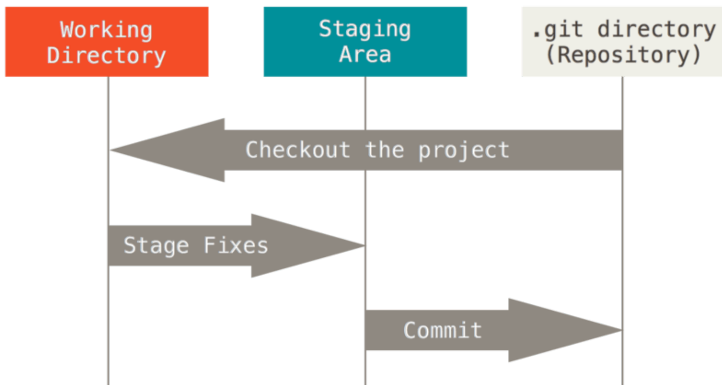

Files life cycle
Tracked
Untracked
Unmodified
Modified
Staged
Add the file
Commit
Edit the file
Stage the file
Delete the file

Three main sections of Git

- The Git directory where Git stores the meta data and object database for your project.
- The working directory is a single checkout (snapshot) of one version of the project.
- The staging area is a file that stores information about what will go into your next commit.
  - It's sometimes referred to as the "index", but it's also common to refer to it as the staging area.

Basic Git Workflow

- The first time you checkout the repository, you copy files from the `.git` directory to the Working directory.
- You modify files in your working directory.
- You stage the files, adding snapshots of them to your staging area.
- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.



Installing Git on your machine

The Git book has some explanation for different OS:
Getting Started - Installing Git

First time Git setup

- Set up your identity. Especially important when you work with people:

```
$ git config --global user.name "David Luet"
$ git config --global user.email luet@princeton.edu
```

- Your favorite editor and some colorful User Interface:

```
$ git config --global core.editor micro
$ git config --global color.ui auto
```

- Checking your settings:

```
$ git config --list
```

Getting help

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

For example:

```
$ git help config
$ git config --help
$ man git-config
```

Outline

Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

Micro text editor

- micro text editor:
  - open a file: `Ctrl+o`.
  - save: `Ctrl+s`.
  - quit: `Ctrl+q`.
  - help: `Ctrl+g`.
  - it supporst mouse selection and clipboard (copy/paste): `Ctrl+c`/`Ctrl+v`.
  - quick start.

Creating a Git repository: let's get some files first

You can download a tarball from the command line using the command `curl`:

```
curl -O -J https://luet.princeton.edu/git/01_git.tar
```

Creating a Git repository from an existing directory

```
# untar/uncompress tarball
$ tar xf 01_git.tar
# move to the new directory
$ cd 01_git_cheatsheet
# list what is in the directory
$ ls
getting_repo.md index.md tohtml.sh
# Create a Git repo
$ git init
Initialized empty Git repository in \
 /Users/luet/talks/PICSciEGit/cheatsheet/\
 git_cheatsheet/.git/
# list again to see what has changed
$ ls -a
.git getting_repo.md index.md tohtml.sh
```

Status of the repo

```
$ git status
On branch master

Initial commit

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

        getting_repo.md
        index.md
        tohtml.sh

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

Status

Untracked

Unmodified

Modified

Staged

getting_repo.md

index.md

tohtml.sh

Git tells you what to do next

```
$ git status
On branch master

Initial commit

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

       getting_repo.md
       index.md
       tohtml.sh

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

Adding files: staging files for commit

```
$ git add getting_repo.md index.md tohtml.sh
$ git status
On branch master

Initial commit

Changes to be committed: <= this means changes stagged
  (use "git rm --cached <file>..." to unstage)

        new file:   getting_repo.md
        new file:   index.md
        new file:   tohtml.sh
```

Status

Untracked

Unmodified

Modified

Staged

getting_repo.md

index.md

tohtml.sh

Committing files

```
$ git commit -m "Initial commit"
[master (root-commit) c1b0ca8] Initial commit
 3 files changed, 16 insertions(+)
 create mode 100644 getting_repo.md
 create mode 100644 index.md
 create mode 100755 tohtml.sh
```

Status

Untracked

Unmodified

Modified

Staged

getting_repo.md
index.md
tohtml.sh

Recording changes to the repository

- Let's assume that we have a Git repository and a checkout of the files for that project.
- We are going to:
  - make some changes.
  - commit snapshots of those changes into the repository.

Checking the status of your files

```
$ git status
On branch master
nothing to commit, working directory clean
```

- This means you have a clean working directory:
  - no tracked and modified files.
  - no untracked files
- The command tells you which branch you are on (`master`).

Adding a new file

- Add a new file (`micro` is a text editor):

```
# open the file with a text editor
$ micro adding_newfile.md
# Add whatever content you would like
# and save and quit (Ctrl+S, Ctrl+Q in micro)
```

- `git status` shows it as untracked and tells you what to do next.

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what \
      will be committed)

      adding_newfile.md

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

Status

Untracked	Unmodified	Modified	Staged
adding_newfile.md	getting_repo.md		
	index.md		
	tohtml.sh		

Untracted files

- Untracked basically means that Git sees a file you did not have in the previous snapshot (commit)
- Git won't start including it in your commit snapshots until you explicitly tell it to do so.
- It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include.

Tracking new files

- You do want to start including `adding_newfile.md`, so let's start tracking the file.

  `$ git add adding_newfile.md`

Status

Untracked	Unmodified	Modified	Staged
	getting_repo.md		
	index.md		
	tohtml.sh		
			adding_newfile.md

Looking at the status again

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

     new file:   adding_newfile.md
```

- You can tell that it's staged because it's under the `Changes to be committed` heading.
- If you commit at this point, the version of the file at the time you ran `git add` is what will be in the historical snapshot.
- The `git add` command takes a path name for either a file or a directory: if it's a directory, the command adds all the files in that directory recursively.

Modifying tracked files

- Let's change a file that was already tracked: `getting_repo.md`

```
$ micro getting_repo.md
# Modify the file and save and quit
# Ctrl+S, Ctrl+Q in micro
```

- and then run the `git status` command again

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

        new file:   adding_newfile.md

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:   getting_repo.md
```

Status

Untracked

Unmodified

Modified

Staged

getting_repo.md

index.md

tohtml.sh

adding_newfile.md

Staging modified files

- To stage the changes to `getting_repo.md`, you run the `git add` command.
- `git add` is a multipurpose command, you use it to:
  - begin tracking new files
  - to stage files
  - and to do other things like marking merge-conflicted files as resolved.
- It may be helpful to think of it:
  - more as "add this content to the next commit",
  - rather than "add this file to the project".

Staging modified files

Let's run `git add` now to stage the file `getting_repo.md`, and then run `git status` again:

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

    new file:   adding_newfile.md
    modified:   getting_repo.md
```

Status

Untracked

Unmodified

Modified

Staged

getting_repo.md

index.md

tohtml.sh

adding_newfile.md

Modifying staged files

At this point, suppose you remember one little change that you want to make in `getting_repo.md` before you commit it.

```
$ micro getting_repo.md # modify the file
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   adding_newfile.md
        modified:   getting_repo.md

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:   getting_repo.md
```

Status

Untracked

Unmodified

index.md
tohtml.sh

Modified

getting_repo.md

Staged

getting_repo.md

adding_newfile.md

Files can be both staged and unstaged

- Now `getting_repo.md` is listed as both staged and unstaged. How is that possible?
- It turns out that Git stages a file exactly as it is when you run the `git add` command.
- If you commit now, the version of `getting_repo.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`.

Files can be both staged and unstaged

If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add getting_repo.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
     new file:   adding_newfile.md
     modified:   getting_repo.md
```

Status

Untracked	Unmodified	Modified	Staged
	index.md tohtml.sh		getting_repo.md adding_newfile.md

Ignoring files

- Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked:
  - Objects files `*.o`
  - Latex generates a lot of log files `*.log`, `*.aux`, `*.bbl`.
  - micro generates backup files that end with a `~`.
- In such cases, you can create a file listing patterns to match them named `.gitignore`.
- Setting up a `.gitignore` file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

Adding `.gitignore`

- In our example, the script `tohtml.sh` reads the Markdown (`*.md`) and converts them into HTML. We don't want to save those HTML files since we can generate them easily.
- So our `.gitignore` file looks like that:

```
$ cat .gitignore
*~
*.html
```

- You need to add `.gitignore` to the Git repository

```
$ git add .gitignore
```

Status

Untracked

Unmodified

index.md
tohtml.sh

Modified

Staged

getting_repo.md

adding_newfile.md
.gitignore

Viewing your staged and unstaged changes

- The `git status` only tells you the status of a file.
- To know exactly what changes you have made to the file use `git diff`.
- You will probably use it most often to answer these two questions:
  1. What have you changed but not yet staged?

     `$ git diff`
  2. What have you staged but not yet committed?

     `$ git diff --cached`
- `git status` answers those questions very generally by listing the file names
- `git diff` shows you the exact lines added and removed.

git diff example

```
$ git diff adding_newfile.md
$ git diff --cached adding_newfile.md
diff --git a/adding_newfile.md b/adding_newfile.md
new file mode 100644
index 0000000..6fcccbd
--- /dev/null
+++ b/adding_newfile.md
@@ -0,0 +1,14 @@
+# Adding new files
+
+-   Check status
+
+       git status
+
+-   Stage changes
+
+       git add file_name
+
...
```

Committing your changes

- Now that your staging area is set up the way you want it, you can commit your changes.
- The simplest way to commit is to type `git commit`.
- Doing so launches your editor of choice.

Commit message

The editor displays the following text:

```
# Please enter the commit message for your changes. Line
# with '#' will be ignored, and an empty message aborts
# On branch master
#
# Changes to be committed:
# new file:   .gitignore
# new file:   adding_newfile.md
# modified:   getting_repo.md
#
~
```

- The lines starting with `#` are comments and won't be part of your message.
- You can see that the default commit message contains the latest output of the `git status` command commented out and one empty line on top.
- For an even more explicit reminder of what you have modified, you can pass the `-v` option to git commit.

Commit message

- Add the commit message, which a description of the changes:

```
# Please enter the commit message for your changes. Line
...
#
Add new file and fix getting_repo.md
```

- After adding the commit message you save (`Ctrl+S`) and exit (`Ctrl+Q`).
- `Git` gives you a summary of what has changed.

```
[master 5ba1206] Add new file and fix getting_repo.md
 3 files changed, 27 insertions(+), 4 deletions(-)
 create mode 100644 .gitignore
 create mode 100644 adding_newfile.md
```

Commit message

- Alternatively, you can type your commit message inline with the commit command by specifying it after a `-m` flag, like this:

```
$ git commit -m "Add new file and fix getting_repo.md"
 [master 5ba1206] Add new file and fix getting_repo.md
  3 files changed, 27 insertions(+), 4 deletions(-)
  create mode 100644 .gitignore
  create mode 100644 adding_newfile.md
```

- `git commit` outputs some information:
  - which branch you committed to (`master`),
  - the commit id (`5ba1206`),
  - how many files were changed,
  - statistics about lines added and removed in the commit.

Status

Untracked

Unmodified

Modified

Staged

getting_repo.md
index.md
tohtml.sh
adding_newfile.md
.gitignore

Skipping the staging area

- The staging area:
  - is amazingly useful for crafting commits exactly how you want them.
  - but sometimes it's a bit more complex than you need in your workflow.
- If you want to skip the staging area, Git provides a simple shortcut.
  - Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:

```
$ git commit -am "Your commit message"
```

    - This is convenient, but be careful: sometimes this flag will cause you to include unwanted changes.

A simple workflow

- make changes to `file1 file2 ....`
- commit directly (skip stagging area) with the commit message on the command line:

  ```
  git commit -m "Do something" file1 file2 ...
  ```

- or all the unstaged, uncommited changes to tracked files:

  ```
  git commit -am "Do something"
  ```

  Note: untracked files won't be part of the commit.

Viewing the commit history

- Looking back at what has happened: the history of the repo.
- The most basic and powerful tool to do this is the `git log` command.

Git log

```
$ git log
commit ed46b26cd7cd2b2541c89f4a49799f2222f2c04b
Author: David Luet <luet@princeton.edu>
Date:   Wed Apr 19 20:19:20 2017 -0400

    Add adding_newfile to the index

commit 5ba1206793de8e1818a387a8919281f47aeca523
Author: David Luet <luet@princeton.edu>
Date:   Wed Apr 19 16:58:14 2017 -0400

    Add new file and fix getting_repo.md

commit c1b0ca817eb513ab66c33f8e9008f7cd3664d22c
Author: David Luet <luet@princeton.edu>
Date:   Wed Apr 19 14:23:33 2017 -0400

    Initial commit
```

Git log

- One of the more helpful options is `-p`, which shows the difference introduced in each commit.
- You can also use `-2`, which limits the output to only the last two entries:

```
$ git log -p -2
commit ed46b26cd7cd2b2541c89f4a49799f2222f2c04b
Author: David Luet <luet@princeton.edu>
Date:   Wed Apr 19 20:19:20 2017 -0400

    Add adding_newfile to the index

diff --git a/index.md b/index.md
index 1e9032d..9604383 100644
--- a/index.md
+++ b/index.md
@@ -1 +1,2 @@
 -   [Getting a repo](./getting_repo.html)
+-   [Adding new file](./adding_newfile.html)

commit 5ba1206793de8e1818a387a8919281f47aeca523
Author: David Luet <luet@princeton.edu>
...
```

Git show

- See what one particular commit changed

  `git show commit_id`

```
$ git show 5ba1206  # your commit id will be different
commit 5ba1206793de8e1818a387a8919281f47aeca523
Author: David Luet <luet@princeton.edu>
Date:   Wed Apr 19 16:58:14 2017 -0400

    Add new file and fix getting_repo.md

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..4c3f7f3
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,2 @@
+*~
+*.html
diff --git a/adding_newfile.md b/adding_newfile.md
new file mode 100644
```

Removing files

- To remove a file from Git, you have to:
  1. remove it from your tracked files: tell Git to forget about it.
  2. then commit.
- The `git rm` command does that, and also removes the file from your working directory.
- If you simply remove the file from your working directory, it shows up under the `Changed but not updated` (that is, unstaged) area of your `git status` output.
- You may also want to keep the file on your hard drive but not have Git track it anymore.

  ```
  git rm --cached
  ```
- This is particularly useful if you forgot to add something to your `.gitignore` file and accidentally staged it.
- If you remove a file that has been committed, it will still be in the history of your repo.

Moving/renaming files

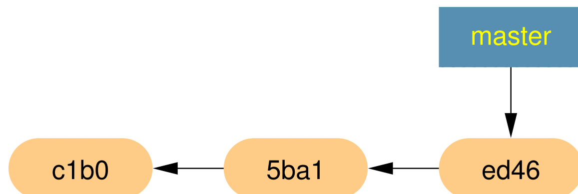
```
$ git mv README.md README
$ git status
On branch master
  (use "git reset HEAD <file>..." to unstage)
    renamed:    README.md -> README
```

Outline

Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

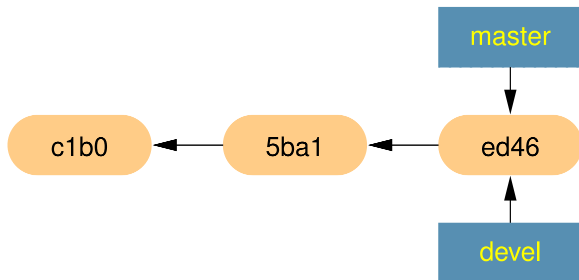
Branch definition

```
$ git log --oneline
ed46b26 Add adding_newfile to the index
5ba1206 Add new file and fix getting_repo.md
c1b0ca8 Initial commit
```

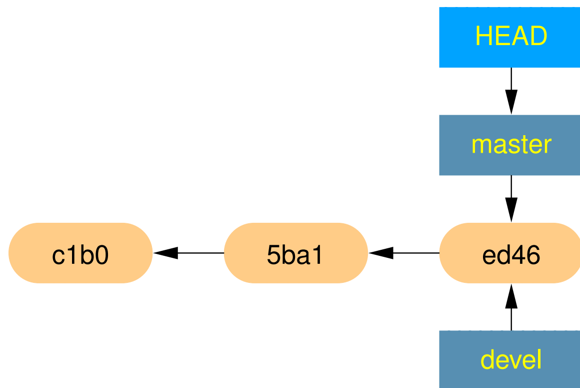


Creating a new branch

```
$ git branch devel
```



HEAD pointer

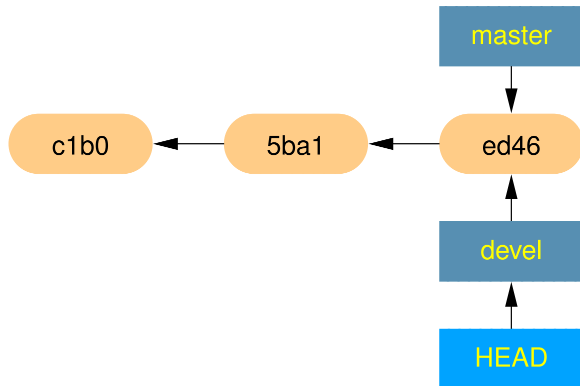



```
$ git log --oneline --decorate
ed46b26 (HEAD -> master, devel) Add adding_newfile to the index
5ba1206 Add new file and fix getting_repo.md
c1b0ca8 Initial commit
```

Switching branches

```
$ git checkout devel
Switched to branch 'devel'
```

Switching branches




```
$ git log --oneline --decorate
ed46b26 (HEAD -> devel, master) Add adding_newfile to the inde
5ba1206 Add new file and fix getting_repo.md
c1b0ca8 Initial commit
```

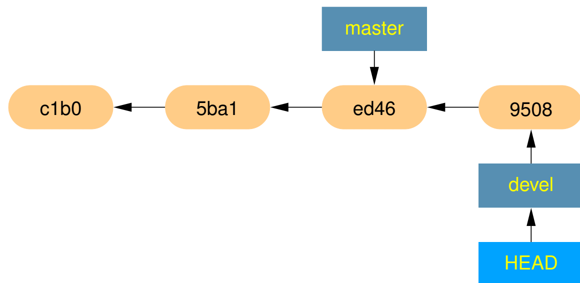
Outline

Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

Making changes to `devel` branch

```
$ ls -1
adding_newfile.md
getting_repo.md
index.md
tohtml.sh
$ micro showing_log.md
$ git add showing_log.md
$ git commit -m "Add showing_log"
$ ls -1
adding_newfile.md
getting_repo.md
index.md
showing_log.md
tohtml.sh
```

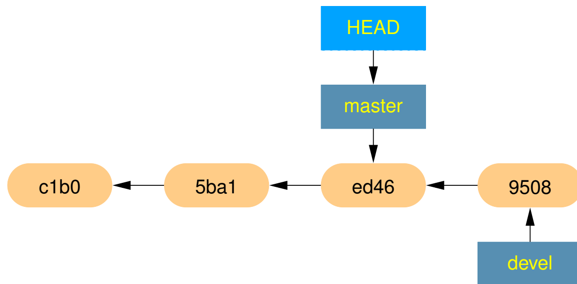
Making changes to `devel` branch




```
$ git log --oneline --decorate
9508bd5 (HEAD -> devel) Add showing_log
ed46b26 (master) Add adding_newfile to the index
5ba1206 Add new file and fix getting_repo.md
c1b0ca8 Initial commit
```

Switching back to master

```
$ git checkout master
Switched to branch ’master’
```



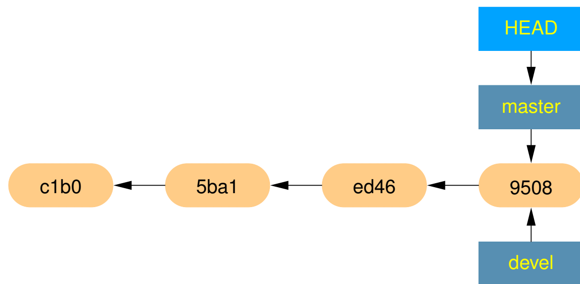
Working directory

```
$ ls -1
adding_newfile.md
getting_repo.md
index.md
tohtml.sh
```

The file `showing_log.md` is not in the working directory anymore.

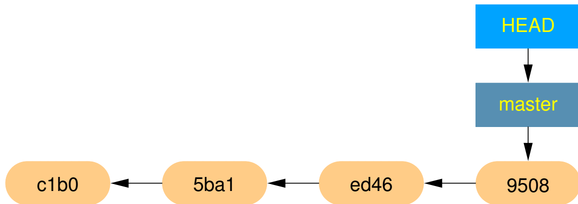
Merge `devel` into `master`: Fast-forward

```
$ git merge devel
Updating ed46b26..9508bd5
Fast-forward
 showing_log.md | 5 +++++
 1 file changed, 5 insertions(+)
 create mode 100644 showing_log.md
```

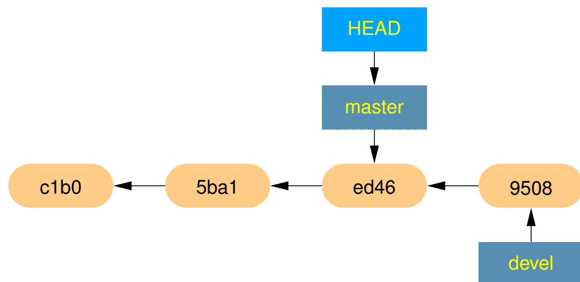


Delete branch `devel`

```
$ git branch -d devel
Deleted branch devel (was 9508bd5).
```



Another merging situation

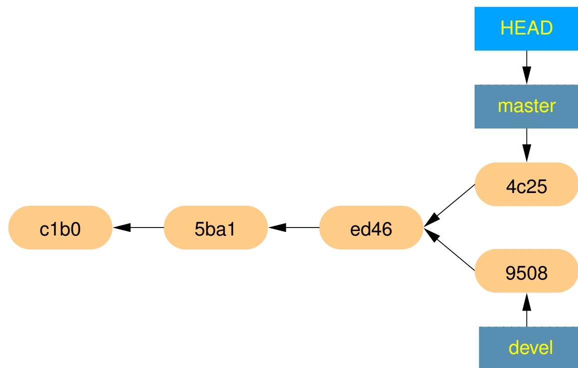



```
$ git log --all --decorate --oneline --graph
 * 9508bd5 (devel) Add showing_log
 * ed46b26 (HEAD -> master) Add adding_newfile to the index
 * 5ba1206 Add new file and fix getting_repo.md
 * c1b0ca8 Initial commit
```

Make changes in `master` branch

```
$ micro config.md
$ git add config.md
$ git commit -m "Add config.md"
[master 4c256d3] Add config.md
 1 file changed, 7 insertions(+)
 create mode 100644 config.md
```

Branches `master` and `devel` have diverged




```
$ git log --all --decorate --oneline --graph
 * 4c256d3 (HEAD -> master) Add config.md
 | * 9508bd5 (devel) Add showing_log
 |/
 * ed46b26 Add adding_newfile to the index
```

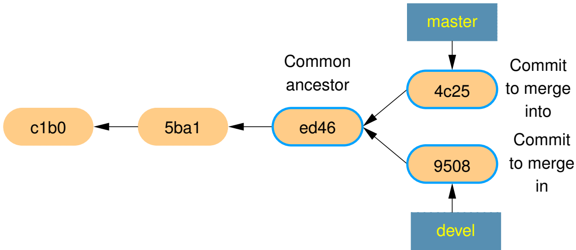
Merging `devel` into `master`: recursive strategy

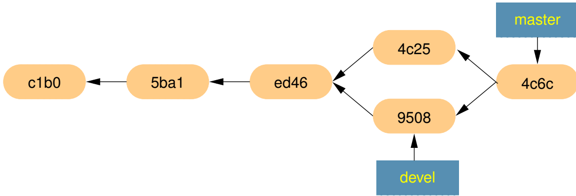
```
$ git merge devel

micro starts

Merge branch 'devel'

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

after editing, exit micro

Merge made by the 'recursive' strategy.
 showing_log.md | 5 +++++
 1 file changed, 5 insertions(+)
 create mode 100644 showing_log.md
```

Git created a merge commit — `git log`

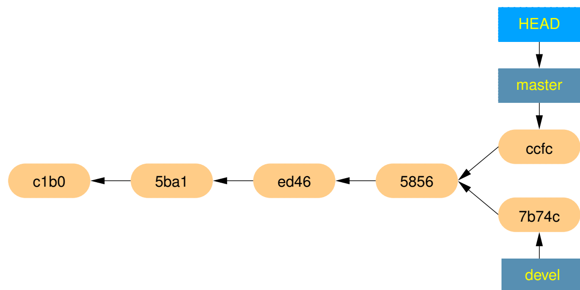
```
$ git log --all --decorate --oneline --graph
*   4c6cf1e (HEAD -> master) Merge branch 'devel'
|\
| * 9508bd5 (devel) Add showing_log
* | 4c256d3 Add config.md
|/
* ed46b26 Add adding_newfile to the index
* 5ba1206 Add new file and fix getting_repo.md
* c1b0ca8 Initial commit
```

Merge commit





Another situation—Let's start over again




```
$ git log --all --decorate --oneline --graph
 * ccfcc8b (HEAD -> master) Add initial set up to index
 | * 7b74cd2 (devel) Add showing logs
 |/
 * 5856ac1 Add config.md
 * ed46b26 Add adding_newfile to the index
 * 5ba1206 Add new file and fix getting_repo.md
 * c1b0ca8 Initial commit
```

Merge conflict

```
$ git checkout master
$ git merge devel
Auto-merging index.md
CONFLICT (content): Merge conflict in index.md
Automatic merge failed; fix conflicts and then commit the resu

$ git show ccfc --name-only
commit ccfcc8bc444289d90f4aeae1a8b819bbe8fe47e0
Author: David Luet <luet@princeton.edu>
Date:   Thu Apr 20 10:39:57 2017 -0400
    Add initial set up to index
index.md
$ git show 7b74c --name-only
commit 7b74cd2df0ed8177f28d740560938d1cb4f85729
Author: David Luet <luet@princeton.edu>
Date:   Thu Apr 20 10:37:58 2017 -0400
    Add showing logs
index.md
showing_log.md
```

Look at `index.md`

```
$ cat index.md
 -   [Getting a repo](./getting_repo.html)
 -   [Adding new file](./adding_newfile.html)
 <<<<<<< HEAD
 -   [Initial set up](./config.html)
 =======
 -   [Showing logs](./showing_log.html)
 >>>>>>> devel
```

Conflict resolution

- Edit `index.md` to resolve the conflict

```
$ micro index.md
-   [Getting a repo](./getting_repo.html)
-   [Adding new file](./adding_newfile.html)
-   [Initial set up](./config.html)
-   [Showing logs](./showing_log.html)
```

- Tell Git that you have resolved the conflict

```
$ git add index.md
```

Commit

- commit

```
$ git commit
```

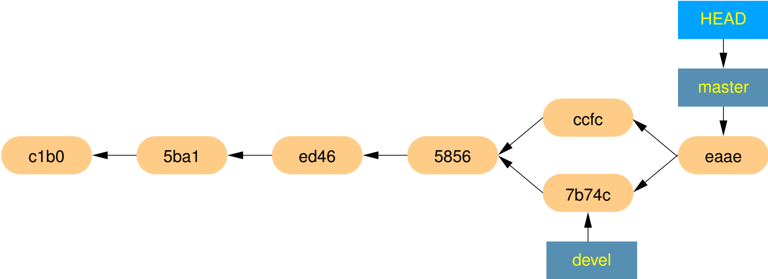
- An editor window opens with the message

```
Merge branch 'devel'

# Conflicts:
# index.md
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
# .git/MERGE_HEAD
# and try again.
...
```

- After you exit the editor, you get the message:

```
$ git commit
[master eaae48c] Merge branch 'devel'
```

Merge commit



Outline

Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

Listing all the local branches

```
$ git branch
  devel
* master
```

Listing all the local branches with last commit

```
$ git branch -v
  devel  7b74cd2 Add showing logs
* master eaae48c Merge branch 'devel'
```

Listing merged status of branches

- Merged branches

```
$ git branch --merged
   devel
 * master
```

- Branches that have not been merged

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
$ date > newfile.md
$ git add newfile.md
$ git commit -m "Add a newfile"
[new_branch 76666c5] Add a newfile
 1 file changed, 1 insertion(+)
 create mode 100644 newfile.md
$ git checkout master
Switched to branch 'master'
$ git branch --no-merged
  new_branch
```

Deleting unmerged branches

```
$ git branch -d new_branch
error: The branch 'new_branch' is not fully merged.
If you are sure you want to delete it, run \
    'git branch -D new_branch'.
$ git branch -D new_branch
Deleted branch new_branch (was 76666c5).
```

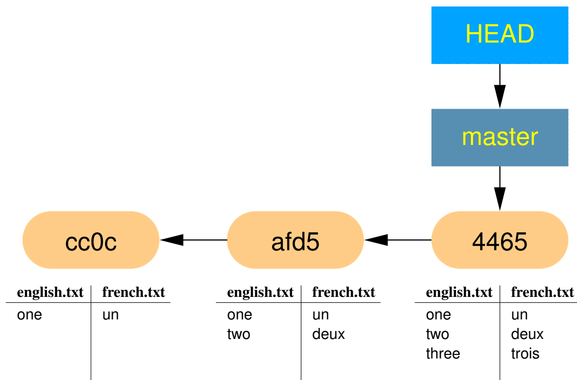
Outline

Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

New Repository

```
$ cd
$ git clone https://github.com/luet/05_git_changes.git
$ cd 05_git_changes
```

git diff




```
$ git log --oneline
4465751 (HEAD -> master) Add 3
afd544d Add 2
cc0c88c Add 1
```

Checking unstaged changes

Commit 4465 (latest)

```
$ cat english.txt
one
two
three
```

Unstaged changes

```
$ cat english.txt
one
three
four
two
```

```
$ git diff english.txt
index 4cb29ea..fb74366 100644
--- a/english.txt
+++ b/english.txt
@@ -1,3 +1,4 @@
 one
-two
 three
+four
+two
```

Comparing commits

- Syntax: `git diff old_commit..new_commit`
- The + and − signs refer to what has been added (+) and subtracted (−) between `old_commit` and `new_commit`.

```
$ git diff afd544d..4465751
diff --git a/english.txt b/english.txt
index 814f4a4..4cb29ea 100644
--- a/english.txt
+++ b/english.txt
@@ -1,2 +1,3 @@
 one
 two
+three
diff --git a/french.txt b/french.txt
index 25973d3..3d5dd13 100644
--- a/french.txt
+++ b/french.txt
@@ -1,2 +1,3 @@
 un
 deux
+trois
```

Other useful `git diff` variations

- Looking at the changes in one file between commits:

  `git diff old_commit..new_commit filename`

- Using `HEAD` to get the latest commit:

  `git diff 2896f02..HEAD`

git show: showing the changes in commit

```
$ git show afd544d
commit afd544d7dc5e9 [...]
Author: David Luet [...]
Date:   [...]

    Add 2

diff --git a/english.txt b/english.txt
index 5626abf..814f4a4 100644
--- a/english.txt
+++ b/english.txt
@@ -1 +1,2 @@
 one
+two
diff --git a/french.txt b/french.txt
index 49fd79f..25973d3 100644
--- a/french.txt
+++ b/french.txt
@@ -1 +1,2 @@
 un
+deux
```

```
$ git show afd544d english.txt
commit afd544d7dc5e9 [...]
Author: David Luet [...]
Date:   [...]

    Add 2

diff --git a/english.txt b/english.txt
index 5626abf..814f4a4 100644
--- a/english.txt
+++ b/english.txt
@@ -1 +1,2 @@
 one
+two
```

Outline

Git locally
- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

`git checkout <commit>`

- e.g.: `git checkout af7bedf`
- Reverts your entire working directory to the state it was in commit `af7bedf`.
- You are in a "detached HEAD" state, which means:
  - you can look at your files, even compile and run.
  - the `master` branch is not modified.
  - but none of the changes you make will be saved unless you create a new branch.
  - You can think of this commit as a read-only branches.

For more details see `git checkout <commit>` in Appendix.

git checkout <commit> <file>

- e.g.: `git checkout af7bedf french.txt`
- Resets the file `french.txt` to its version in commit `af7bedf`.
- Keeps the current version of the other files in the working directory.
- Git:
  - will modify the file in the current directory.
  - will stage the changes.
  - but will not commit the changes.

For more details see `git checkout <commit> <file>` in Appendix.

`git revert <commit>`

- e.g. `git revert 720242`.
- Revert (undo) commit `720242` by creating a new commit.
- Works well when you want to undo the last commit.
- But things get a little more complicated when you try to undo an earlier commit:
  - `git revert` uses `git merge`.
  - you often end up with merge conflicts.
- Often it's just easier to create a commit that fixes the problem by hand.

See `git revert` in Appendix.

git reset <commit>

- e.g. `git reset a92b610`.
- "rewinds" to commit `a92b610` by deleting the children commit.
  - but leaves the changes in your working directory.
- `git reset --hard a92b610`.
  - deletes all the changes after `a92b610`.

See `git reset` in Appendix.

`git revert` vs. `git reset`: why is `git reset` dangerous?

The big differences between `git revert` and `git reset` are:

- `git revert`
  - only reverts one commit.
  - creates a new commit to undo changes.
  - does not modify past history; it just adds to it.
- `git reset`:
  - does not create new commit, it just deletes old ones.
  - modifies the past.
  - that becomes a problem if you have already shared the deleted changes with others, for instance through a shared repository on GitHub.
  - `git reset` is dangerous use it with caution.

Never use `git reset` on commits that you have shared publicly: Others will not be able to push their changes after you do that.

Outline

Git locally

- Git Concepts
- Git Basic Usage
- Introduction to Branching
- Merging branches
- Branch Management
- Inspecting Changes
- Working with Older Versions of the Repo: Undoing Changes
- Conclusion

Git Locally: Conclusion

- Different states of a file:
  - untracked.
  - tracked:
    - committed.
    - modified.
    - staged.
- Three main sections:
  - `.git` directory, `Git` database.
  - working directory current version of your files.
  - stagging area: what will go into the next commit.
- Create a git repository from scratch: `git init`.
- Look at the status of the repo: `git status`.
- Add changes: `git add`.
- Commit: `git commit`.

Git Locally: Conclusion

- Looking at history of the repo:
  - `git log`, `git show`
  - `git diff`
- Important feature of `Git`: branches:
  - create, switch.
  - multiple scenarios for merging:
    - fast forward
    - recursive
    - merge conflicts.
- Working with old version of the repository and go back to some previous version with `git revert` and `git reset`.

Outline

Version Control Systems (VCS)

Git locally

Remote Repositories

References

Appendices

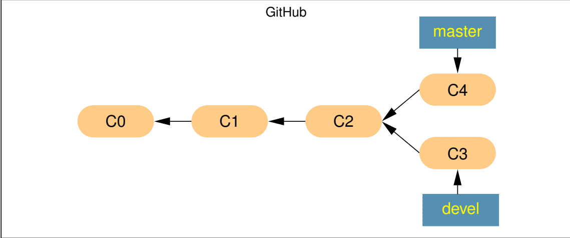
Outline

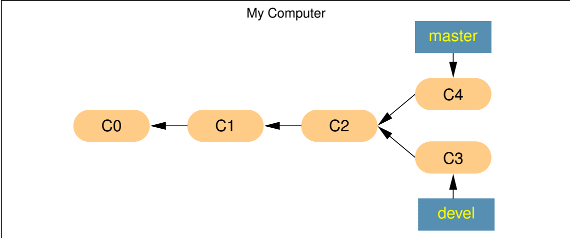
Remote Repositories
- Introduction
- Creating a Remote Repository From `git_cheatsheet`
- Working With Remote Repositories
- Conclusion

What is a remote repository?

- Remote repositories are versions of your project that are not on your computer.
- They usually include:
  - the history.
  - branches, called the remote branches.

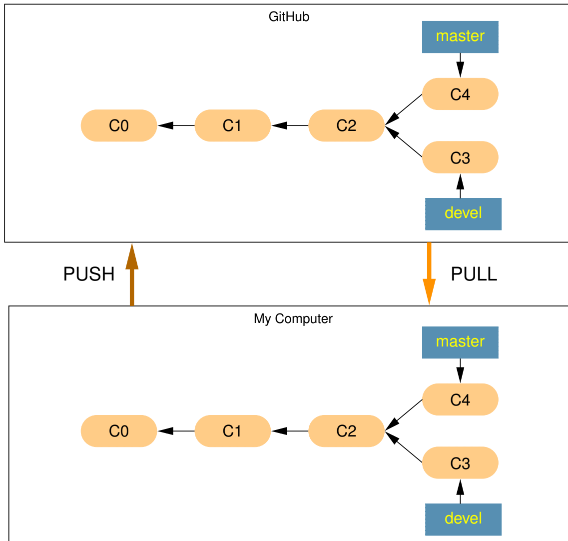
Remote repository on GitHub



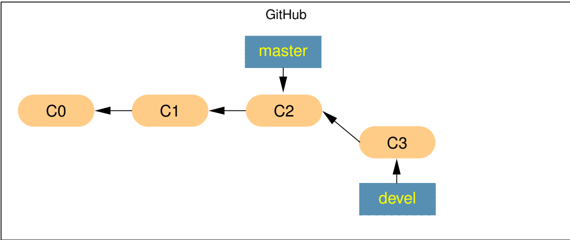


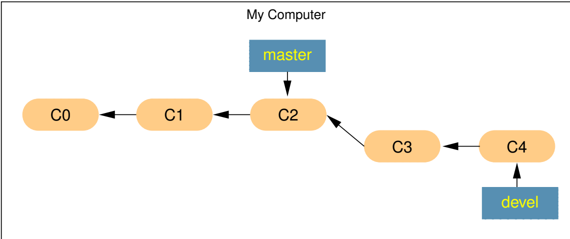
Interaction with remote repositories

- You push changes from your computer to the remote.
- You pull changes from the remote to your computer.

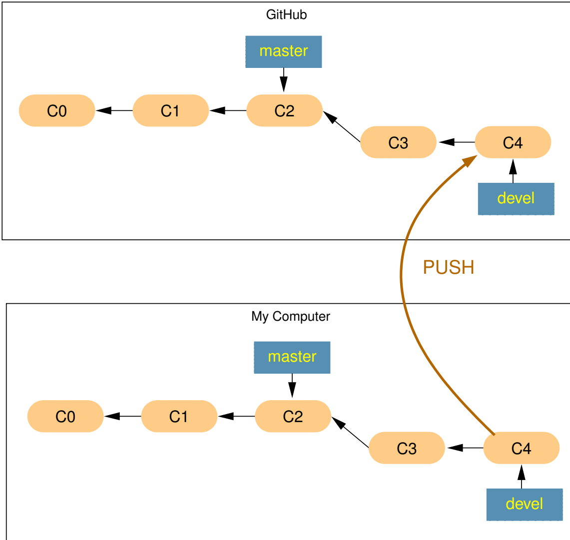


Pushing Commits to the Remote

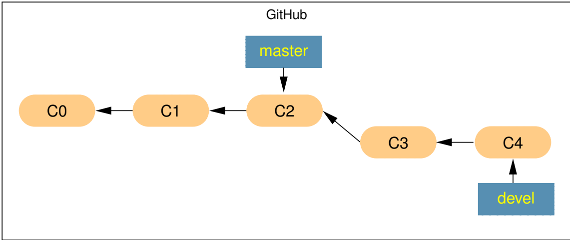


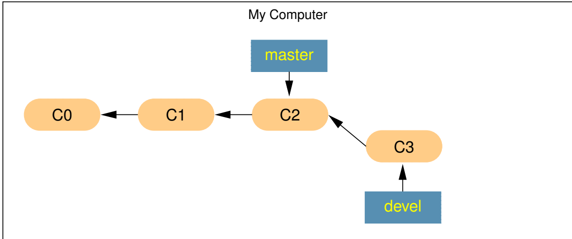


Pushing Commits to the Remote

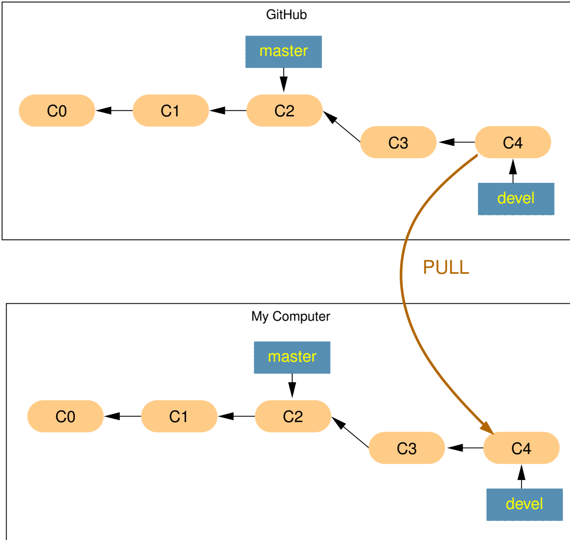


Pulling Commits from the Remote





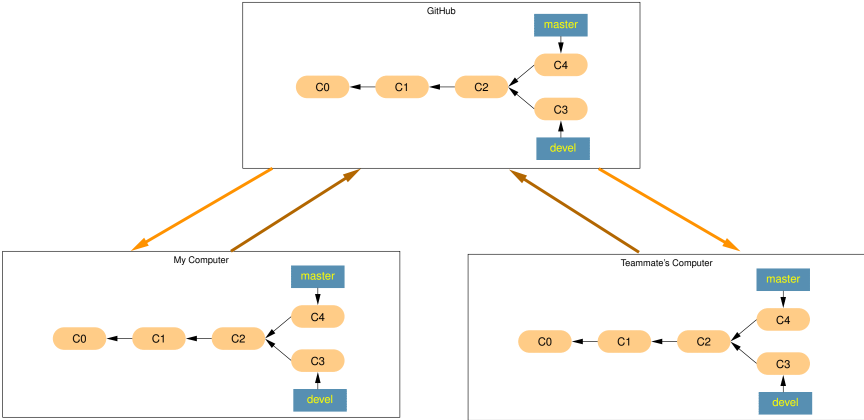
Pulling Commits from the Remote



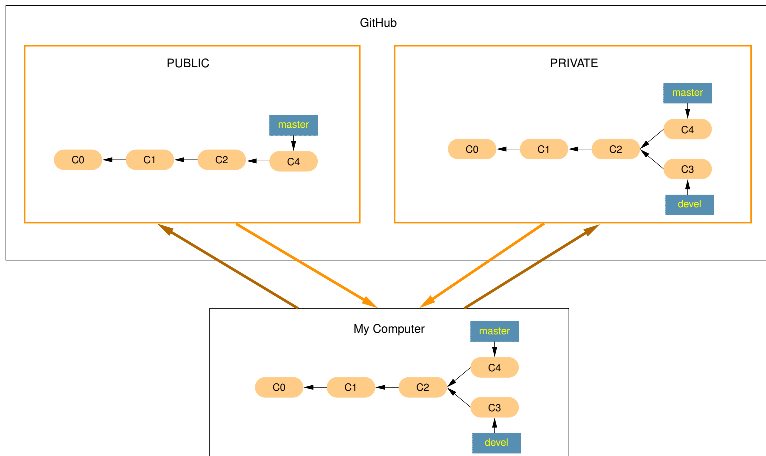
Why use remote repositories?

- backup your work.
- collaborate.
- you can have several remote repositories.

Remote for collaboration



You can have multiple remotes



Outline

Remote Repositories
Introduction
Creating a Remote Repository From `git_cheatsheet`
Working With Remote Repositories
Conclusion

Publishing your local repository on GitHub

- Now you want the publish the repository `git_cheatsheet` that we built earlier with `git init`.
- So that you can share it with your teammates.
- This is a 3 steps process:
  1. Create an empty repository on GitHub.
  2. Define a remote repository for your local repository.
  3. Push your local repository to the remote repository on GitHub.

GitHub empty repo: Start a project

GitHub

GitHub

GitHub, Inc. (US) https://github.com

Search

Search GitHub Pull requests Issues Gist

Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

Read the guide

Start a project

luet

14 hours ago

c-white commented on issue PrincetonUniversity/athena-public-version#3

Indeed the sign of ψ should be flipped in what I wrote above. As for the absolute value, you're right that changes nothing here. It's just a habit …

20 hours ago

leftaroundabout closed issue PrincetonUniversity/athena-public-version#3

Is MHD in spherical coordinates actually supported?

20 hours ago

leftaroundabout commented on issue PrincetonUniversity/athena-public-version#3

Thanks for the replies. I should have noticed that the initial magnetic field was not homogeneous. By properly transforming the initial field (left…

The GitHub Satellite schedule is here: save your seat

Save your seat for GitHub Satellite.

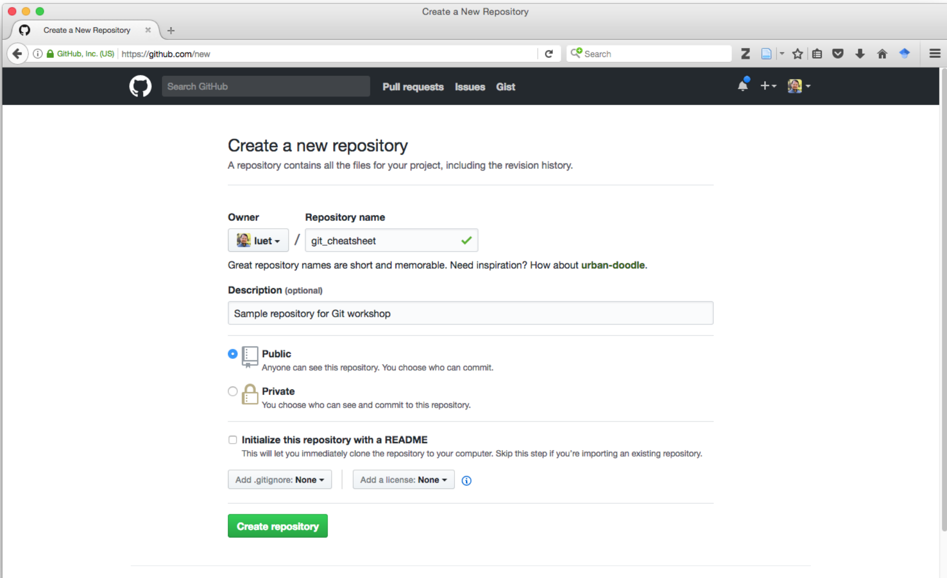
View 97 new broadcasts

Repositories you contribute to 6

- PrincetonUniv… /HelloWorld 0
- PrincetonUniv… /athena-publi… 10
- PrincetonUniv… /athena 2
- APC524/git_cheatsheet 0
- PrincetonUniv… /vonHoldt_chr… 0

Show 1 more repository…

GitHub empty repo: Create a new repository



GitHub empty repo: Pushing your local repo

luet/git_cheatsheet

luet / git_cheatsheet

Unwatch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Pulse | Graphs | Settings

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

Set up in Desktop or HTTPS SSH https://github.com/luet/git_cheatsheet.git

We recommend every repository include a README, LICENSE, and .gitignore.

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

```
echo "# git_cheatsheet" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/luet/git_cheatsheet.git
git push -u origin master
```

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

```
git remote add origin https://github.com/luet/git_cheatsheet.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

Add the remote definition to your local repository

Let's assume that you just created

`https://github.com/luet/git_cheatsheet`

```
$ pwd
/Users/luet/talks/CoDaS-HEP/git_cheatsheet
$ git remote -v
$ git remote add origin git@github.com:luet/\
                               git_cheatsheet.git
$ git remote -v
origin       git@github.com:luet/git_cheatsheet.git (fetch)
origin       git@github.com:luet/git_cheatsheet.git (push)
```

Pushing your repo to GitHub

```
$ git push --set-upstream origin master
Counting objects: 13, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (13/13), 1.42 KiB | 0 bytes/s, done.
Total 13 (delta 1), reused 5 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To git@github.com:luet/git_cheatsheet.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Outline

Remote Repositories
- Introduction
- Creating a Remote Repository From `git_cheatsheet`
- Working With Remote Repositories
- Conclusion

Pushing your changes to the remote

- Let’s say you have some commits that you have not pushed to the remote.
- `git status` tells you:

```
$ 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
```

Pushing your changes to the remote

It's time to push your work so that your colleagues can use it:

```
$ git push origin master
X11 forwarding request failed on channel 0
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 666 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@github.com:luet/git_cheatsheet.git
   c10216b..f4b087f  master -> master
```

Updating your local copy of the remote branches

- Now you want to check if someone has pushed some changes on GitHub.
- First you need to update your copy of the remote `origin`.

```
$ git fetch origin
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:APC524/git_cheatsheet
   f4b087f..f7aa4c4  master     -> origin/master
```

Has something changed?

```
$ git status
On branch master
Your branch is behind ’origin/master’ by 1 commit, \
  and can be fast-forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working directory clean
```

Look at what has changed

```
$ git diff origin/master
diff --git a/index.md b/index.md
index 70d3d1b..1e9032d 100644
--- a/index.md
+++ b/index.md
@@ -1,2 +1 @@
 -   [Getting a repo](./getting_repo.html)
--   [Adding a new file](./adding_newfile.html)
```

Merging the remote branch into your local branch

- So far, you have fetched the remote branch but the changes have not been merged.
- You merge them with

```
$ git merge origin/master
```

Pulling the changes

- Instead of fetching and merging you can use the command `git pull`:

`git pull` = `git fetch` + `git merge`

Showing remote repositories

- To see which remote servers you have configured, you can run

```
$ git remote -v
public https://github.com/luet/git_cheatsheet.git (fetch)
public https://github.com/luet/git_cheatsheet.git (push)
private https://github.com/PrincetonUniversity/\
        git_cheatsheet.git (fetch)
private https://github.com/PrincetonUniversity/\
        git_cheatsheet.git (push)
```

Outline

Remote Repositories
Introduction
Creating a Remote Repository From `git_cheatsheet`
Working With Remote Repositories
Conclusion

Conclusion

- Create and manage branches to develop your code while keeping a stable production version.
- Exchange commits between branches.
- Export your developments on GitHub for collaborative development and backup.

Outline

Version Control Systems (VCS)

Git locally

Remote Repositories

References

Appendices

Outline

References

Git Remote Repository Hosting

Graphical User Interfaces

Further Learning

Git Remote Repository Hosting

- GitHub:
  - Free unlimited public repo.
  - Free unlimited private repo (pricing):
    - limited to 3 collaborators. But as students, you can get a Student Pack.
  - You should setup some ssh keys, it will make your life easier: GitHub Help.
- GitLab: free, unlimited, supports Git LFS (Large File Storage).
- BitBucket:
  - free for everyone up to 5 users.
  - free academic subscription (`.edu` email).
- On your own server through ssh (more advanced).

Outline

References
- Git Remote Repository Hosting
- Graphical User Interfaces
- Further Learning

Graphical User Interfaces

- gitk:
  - simple but powerful.
  - almost always available on Linux machines (nobel).
  - the missing gitk documentation: this is a good, concise documentation.
- Atlassian SourceTree.
- GitHub Desktop.
- On Windows: TortoiseGit.

Outline

References
Git Remote Repository Hosting
Graphical User Interfaces
Further Learning

Further Learning

- The best way to learn Git is to start using it. Experiment with it on your local machine.
- Scott Chacon and Ben Straub Pro Git (excellent book):
  - Chapters 1,2 and 3 for basic Git usage.
  - Chapter 6 for GitHub.
- Git everyday
  - Individual Developer (Standalone)
  - Individual Developer (Participant)
- An interesting interactive Git cheatsheet
- GitHub Tutorials
- Chacon's Git tutorial:
  - Part 1
  - Part 2
- Atlassian Tutorials
- GitHub Guides

Outline

Version Control Systems (VCS)

Git locally

Remote Repositories

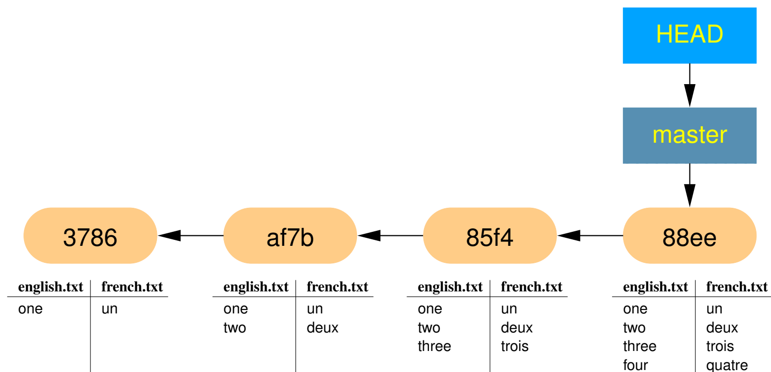
References

Appendices

Outline

Appendices
`git checkout`
`git revert`
`git reset`
`git submodule`

Checking out an old commit



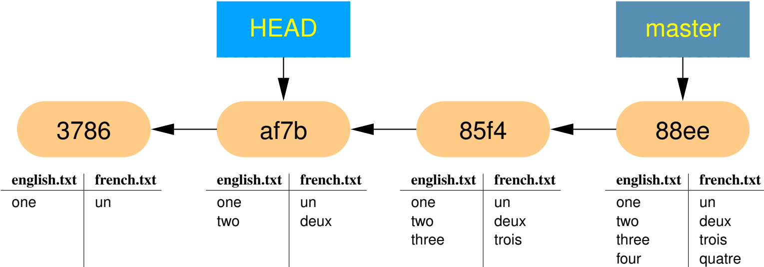

```
$ git log --oneline
596f954 (HEAD -> master, origin/master, origin/HEAD) Add 4
4465751 Add 3
afd544d Add 2
cc0c88c Add 1
```

git checkout <commit>

We want to look at commit `afd544d`

```
$ git checkout afd544d
Note: checking out 'afd544d'.

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

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

  git checkout -b <new-branch-name>

HEAD is now at afd544d... Add 2
```

Detached HEAD




```
$ git branch
 * (HEAD detached at afd544d)
   master
```

Detached HEAD

```
$ cat english.txt
one
two
```

```
$ cat french.txt
un
deux
```

```
$ git diff HEAD..master english.txt
diff --git a/english.txt b/english.txt
index 814f4a4..f384549 100644
--- a/english.txt
+++ b/english.txt
@@ -1,2 +1,4 @@
 one
 two
+three
+four
```

What can you do in detached HEAD state?

- Run the code.
- Compile the code.
- For instance, to go back in history to find out when a bug was introduced.
- You can commit.
  - those commits won't change the code in the branch `master`
  - but they won't be saved if you don't create a new branch.

Switching back to branch `master`

```
$ git branch
 * (HEAD detached at afd544d)
  master
$ git checkout master
Previous HEAD position was afd544d... Add 2
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
$ git branch
 * master
```

If you don't create a new branch, the detached `HEAD` state disappears.

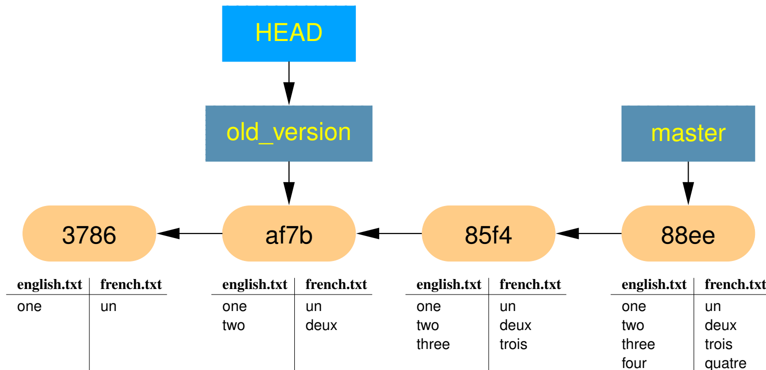
Creating a new branch from the detached HEAD state

- If you want to keep changes you make in the detached HEAD state, you need to create a new branch, otherwise your changes to the detached HEAD state will be lost.

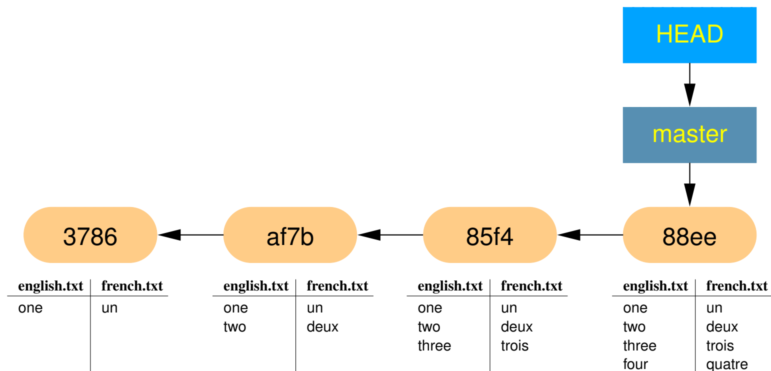
```
$ git checkout afd544d
...
$ git branch
 * (HEAD detached at afd544d)
   master
$ git checkout -b old_version
Switched to a new branch 'old_version'
$ git branch
   master
 * old_version
```

- Think of `git checkout <commit>` as a way to look at previous commits and see what has changed.

git checkout <commit> as an undo function



git checkout <commit> <file>



We want to:

- reset the file `french.txt` to its version in commit `afd544d`.
- keep the latest version of `english.txt`.

Reverting one file

```
$ git checkout afd544d french.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   french.txt
```

Git has:

- modified the file `french.txt`.
- staged the modifications.

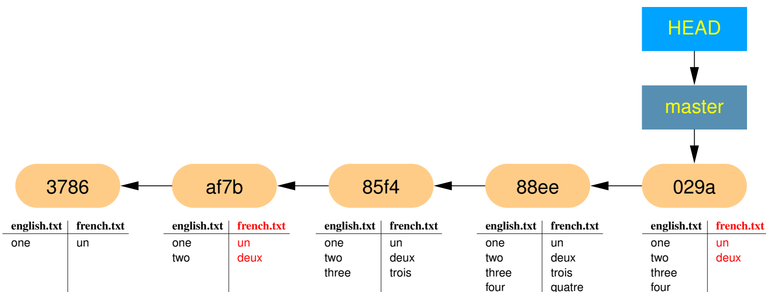
What has changed?

```
$ git diff --cached french.txt
diff --git a/french.txt b/french.txt
index 628ffa8..25973d3 100644
--- a/french.txt
+++ b/french.txt
@@ -1,4 +1,2 @@
 un
 deux
-trois
-quatre
```

Git modified the file `french.txt` so that it is the same as it was in commit `afd5`.

To keep the changes, you need to create a new commit

```
$ git commit -m "Revert french.txt to commit afd5"
[master 09919bb] Revert french.txt to commit afd5
 1 file changed, 2 deletions(-)
```



Outline

Appendices

- `git checkout`
- `git revert`
- `git reset`
- `git submodule`

Reverting a commit by creating a new commit: `git revert`

```
$ git log --oneline
596f954 (HEAD -> master) Add 4
4465751 Add 3
afd544d Add 2
cc0c88c Add 1
```

- We want to revert (undo) the last commit

```
$ git revert 596f954
[a window pops up]
```

git revert: reverting one commit

```
Revert "Add 4"

This reverts commit 596f9548c832ca6a75ca5ac81c165829f84f9fa7.

# Please enter the commit message for your changes. Lines star
# with '#' will be ignored, and an empty message aborts the co
#
# On branch master
# Your branch is up to date with 'origin/master'.
#
# Changes to be committed:
#       modified:   english.txt
#       modified:   french.txt
#
[exit the text editor]
[master 0138210] Revert "Add 4"
 2 files changed, 2 deletions(-)
```

`git revert` creates a new commit that is a "mirror" of the original commit

```
$ git diff 596f954^1..596f954
iff --git a/english.txt b/english.txt
index 4cb29ea..f384549 100644
--- a/english.txt
+++ b/english.txt
@@ -1,3 +1,4 @@
 one
 two
 three
+four
diff --git a/french.txt b/french.txt
index 3d5dd13..628ffa8 100644
--- a/french.txt
+++ b/french.txt
@@ -1,3 +1,4 @@
 un
 deux
 trois
+quatre
```

```
$ git diff 0138210^1..0138210
diff --git a/english.txt b/english.txt
index f384549..4cb29ea 100644
--- a/english.txt
+++ b/english.txt
@@ -1,4 +1,3 @@
 one
 two
 three
-four
diff --git a/french.txt b/french.txt
index 628ffa8..3d5dd13 100644
--- a/french.txt
+++ b/french.txt
@@ -1,4 +1,3 @@
 un
 deux
 trois
-quatre
```

Comments on `git revert`

- Works well when you want to undo the last commit.
- But things get a little more complicated when you try to undo an earlier commit:
  - `git revert` uses `git merge`.
  - you often end up with merge conflicts.
- Often it's just easier to create a commit that fixes the problem by hand.

Outline

Appendices
`git checkout`
`git revert`
`git reset`
`git submodule`

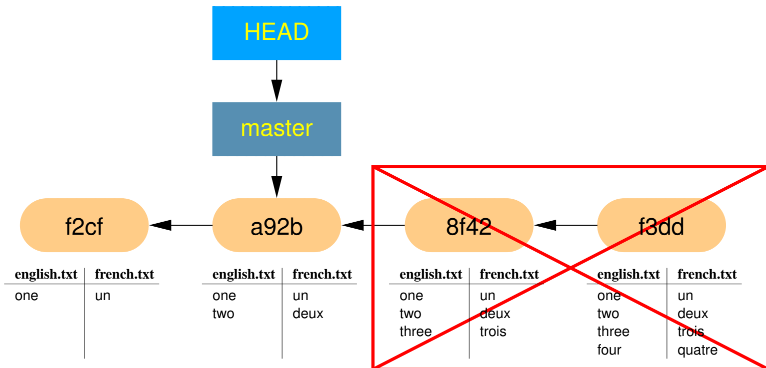
Reverting to a commit by deleting children commit: `git reset`

```
$ git log --oneline
596f954 (HEAD -> master) Add 4
4465751 Add 3
afd544d Add 2
cc0c88c Add 1
```

You want to reset the repo to commit `afd544d`.

```
$ git reset afd544d
Unstaged changes after reset:
M        english.txt
M        french.txt
```

`git reset` new history




```
$ git log --oneline
afd544d (HEAD -> master) Add 2
cc0c88c Add 1
```

What are those unstaged changes?

```
$ 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 worki

        modified:   english.txt
        modified:   french.txt

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

```
$ cat english.txt
one
two
three
four
```

```
$ cat french.txt
un
deux
trois
quatre
```

- `git reset` rewinds the commits but leaves the changes in your working directory.
- The default option is `git revert --mixed`, this is what we have been using here.

Removing the changes from the working directory

```
$ git log --oneline
596f954 (HEAD -> master) Add 4
4465751 Add 3
afd544d Add 2
cc0c88c Add 1
$ git reset --hard afd544d
HEAD is now at afd544d Add 2
$ git status
On branch master
nothing to commit, working tree clean
```

- All the changes from `4465` and `596f` are gone.
- From Git's point of view it's as if they had never existed: they have been deleted from the history of the repo.

`git revert` vs. `git reset`: why `git reset` is dangerous.

The big differences between `git revert` and `git reset` are:

- `git revert`
  - only reverts one commit.
  - creates a new commit to undo changes.
  - does not modify past history; it just adds to it.
- `git reset`:
  - does not create new commit, it just deletes old ones.
  - modifies the past.
  - that becomes a problem if you have already shared the deleted changes with others, for instance through a shared repository on GitHub.
  - `git reset` is dangerous use it with caution.

Never use `git reset` on commits that you have shared publicly: Others will not be able to push their changes after you do that.

Outline

Appendices
- `git checkout`
- `git revert`
- `git reset`
- `git submodule`

git submodule

- This is a good tutorial on `git submodule`: https://git.wiki.kernel.org/index.php/GitSubmoduleTutorial
- And this is the section on submodule in the Pro-Git book: https://git-scm.com/book/en/v2/Git-Tools-Submodules