Version control using Git & GitHub

Pablo Orviz <orviz@ifca.unican.es>

Master en Data Science
2017/18

Version Control Systems (VCS)

Version Control

“Systems that record changes to a/set of file/s over time so that you can recall specific versions later” (Chacon&Straub, *Pro Git*)

Version Control

“Systems that record changes to a/set of file/s over time so that you can recall specific versions later” (Chacon&Straub, *Pro Git*)

Allows to:

- Revert files back to a previous state
- Compare changes over time
- See who last modified something that is causing a problem
- Who introduced an issue and when
- ..

Version Control

“Systems that record changes to a/set of file/s over time so that you can recall specific versions later” (Chacon&Straub, *Pro Git*)

Allows to:

- Revert files back to a previous state
- Compare changes over time
- See who last modified something that is causing a problem
- Who introduced an issue and when
- ..

If you screw things up or lose files, you can easily recover

Version Control Systems (VCS)

Local Version Control Systems

Filesystem-based

- Duplicate files/directories
  - Hopefully time-stamped
  - Usually with prefix/suffix such as ‘v1’, ..

Version Control Systems (VCS)

Local Version Control Systems

Filesystem-based

- Duplicate files/directories
  - Hopefully time-stamped
  - Usually with ‘v2’
- Error-prone!

Version Control Systems (VCS)

Local Version Control Systems

Filesystem-based

- Duplicate files/directories
  - Hopefully time-stamped
  - Usually with ‘v2’
- Error-prone!

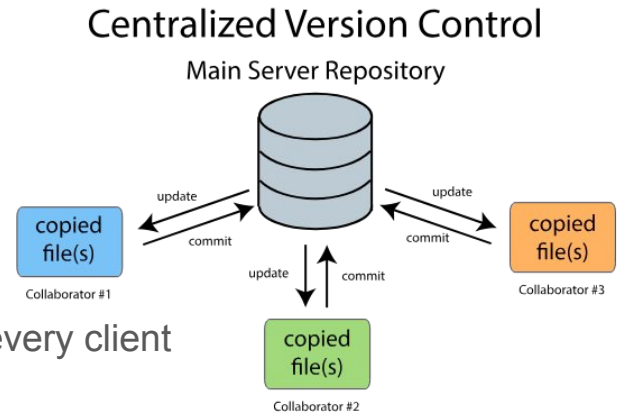
Database-based

- First approach to a real VCS
- Each new version is stored in the DB as a patch set (only diffs)
- Able to re-create any file by adding up the appropriate patches

Version Control Systems (VCS)

Centralized Version Control Systems (*aka CVCS*)

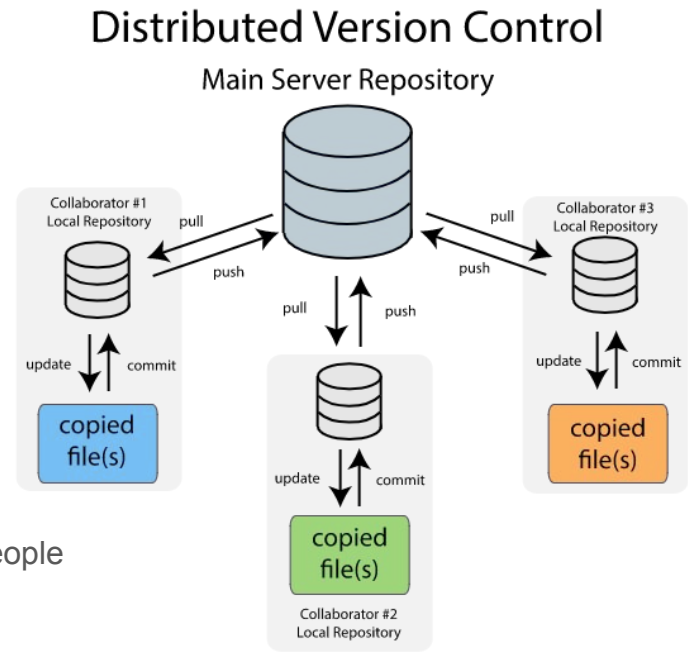
- Problem: Collaboration was needed (geographically distributed)
- Centralized solution:
  - **1 server** with all the versioned files
  - N clients that check out files from the server
- Pros:
  - Allows collaboration
  - Everyone knows what the others do (to a certain degree)
  - Facilitates administration when compared with local DBs on every client
- Cons:
  - **Single point of failure**
    - Network outage (temporary loss), disk failure/corrupted (~permanent loss)
- Examples: CVS, Subversion



Version Control Systems (VCS)

Distributed Version Control Systems (*aka DVCS*)

- Problem: Availability & reliability
- Distributed solution:
  - Each client **mirrors/clones the entire repository**
    - Full backup
    - Speed
- Pros:
  - High Availability
    - If *Main Server Repository* dies, it can be restored
      - From any client repository
  - Multiple collaborations
    - Through multiple remotes
      - Simultaneously collaborate with different groups of people
- Cons:
  - None, they are perfect! (almost)
- Examples: Git, Mercurial



If all you remember is..

- *A VCS offers a great added value when it comes to manage whatever type of files that evolve or change over time*

If all you remember is..

- *A VCS offers a great added value when it comes to manage whatever type of files that evolve or change over time*
  - Revert files back to a previous state
  - Compare changes over time
  - See who last modified something that is causing a problem
  - Who introduced an issue and when

If all you remember is..

- *A VCS offers a great added value when it comes to manage whatever type of files that evolve or change over time*
  - Revert files back to a previous state
  - Compare changes over time
  - See who last modified something that is causing a problem
  - Who introduced an issue and when
- *DVCS are the most mature VCS solutions*

If all you remember is..

- *A VCS offers a great added value when it comes to manage whatever type of files that evolve or change over time*
  - Revert files back to a previous state
  - Compare changes over time
  - See who last modified something that is causing a problem
  - Who introduced an issue and when
- *DVCS are the most mature VCS solutions*
  - High availability: they do not depended on a central server
  - Improve collaboration: handling multiple remotes
  - Speed: common operations are done faster as metadata is stored locally

If all you remember is..

- *A VCS offers a great added value when it comes to manage whatever type of files that evolve or change over time*
  - Revert files back to a previous state
  - Compare changes over time
  - See who last modified something that is causing a problem
  - Who introduced an issue and when
- *DVCS are the most mature VCS solutions*
  - High availability: they do not depended on a central server
  - Improve collaboration: handling multiple remotes
  - Speed: common operations are done faster as metadata is stored locally
- *Git is most widely-used DVCS tool*

Git and GitHub

GitHub

Git-repository hosting service
https://github.com

Social code: account-based
- Individual, e.g. https://github.com/orviz
- Organizations, e.g. https://github.com/masterdatascience-UIMP-UC

Extra functionality (on top of Git):
- Collaboration (social coding)
  - Forks
  - Pull requests
- Private/public repositories
- Integrations with other services (CI, monitoring, ..)
- Tools (wiki, issue tracking)
- Project webpage: GitHub Pages
- Reporting: graphs/stats
- ..

Alternatives: https://bitbucket.org, https://sourceforge.net

GitHub.com (https://octoverse.github.com/)

- 24 millions of users from 200 countries
- 1.5 millions of organizations
- 67 millions of repositories


MENU nature International journal of science

News & Comment Research

News Opinion Research Analysis Careers Books & Culture

NEWS · 15 JUNE 2018

Microsoft's purchase of GitHub leaves some scientists uneasy

They fear the online platform will become less open, but other researchers say the buyout could make GitHub more useful.

GitHub

Social coding

Science in GitHub.com: democratic databases

1. Ebola outbreak in West Africa, July 2014
2. PhD student wanted to model the outbreak spread
3. Every day
   - Downloaded PDF updates from the ministries of health of the affected countries
   - Converted the numbers into computer readable tables
   - Uploaded the files to a GitHub.com repository, thought may be useful for someone
4. Other researchers started to contribute in the project:
   - On some days, files were uploaded before her
   - Created programming scripts for error-checks on the data

https://www.nature.com/news/democratic-databases-science-on-github-1.20719

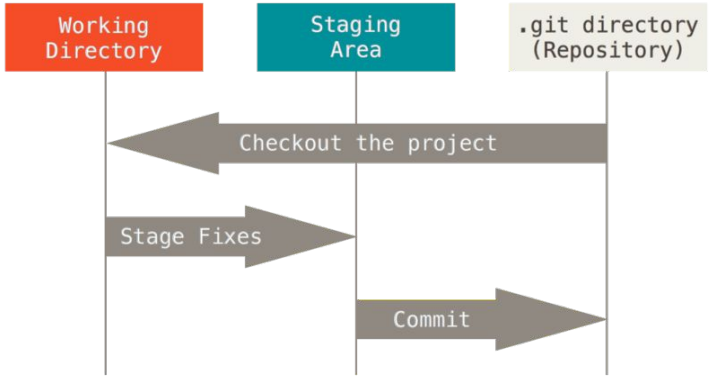
Git basics

- Uses **snapshots**, not differences
  - Every change (commit, save state, ..) triggers a snapshot of all the date and adds a reference to it
    - Different from the *delta-based*: stores the files changed and the changes themselves
- Most operations are **local**
  - Operations are *instant*
    - Checks local database, no network needed
    - Unlike CVCS, which adds network latency overhead
- The Three States
  - Committed: data safely stored in your local database
  - Modified: the file is changed but not committed
  - Staged: mark a modified file to go into your next commit snapshot

Git basics

The Three Sections of a Git project

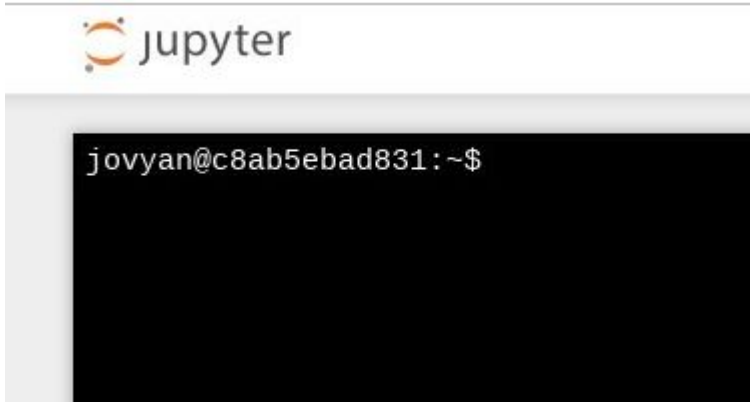
- *Git directory (.git)*, metadata and DB.
- *Working directory*: single checkout of one version of the project.
- *Staging area*: file that stores information about what will go into the next commit.



Workflow:

1. You modify files in your *working dir*.
2. You stage changes to go into the next commit.
    - Adds only these changes to the *staging area*.
3. You do a commit
    - Does a snapshot of the files as they are in the staging area to place them in the *Git directory*.

Getting started with Git



Command-line vs GUI

- *We will be using Git on the command line*
  - GUIs implement only a partial subset, e.g. https://desktop.github.com/
  - Hard way: if you master the command-line, GUI won't be any secret.
    - The opposite is not necessarily true.
- *Command line is available through https://datasciencehub.ifca.es*
  - Through the web browser, no need to install the Git client locally
  - Log in using your GitHub account.
    - If not already, create one in https://github.com
  - Sake of completeness: how to install Git

Getting started with Git

First-time Git Setup

- 3 Git configuration files on your system (from lowest to highest priority):
  1. `/etc/gitconfig` (system-wide)
  2. `$HOME/.gitconfig` or `$HOME/.config/git/config` (user space)
  3. `.git/config` in your Git directory (single repository)
- ***git config*** command line will modify the values in the files above for you
  - With ***--system*** option, reads/writes from `/etc/gitconfig`
  - With ***--global*** option, reads/writes from user space, aka `$HOME/.gitconfig`
  - With ***--local*** option, reads/writes from current repository, aka `.git/config`

Getting started with Git

First-time Git Setup

- Set your Git identity (globally)
  - **Important!** Name & Email will be used in each commit

```
$ git config --global user.name “John Doe”

$ git config --global user.email johndoe@example.com
```

- Additional configurations..
  - Set your preferred editor (**vim is not available by default in JupyterHub btw**)

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

Getting started with Git

Check your Settings

- List all of them:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
...
```

- Print specific configuration parameter:

```
$ git config user.name
John Doe
```

Ask Git for Help

- Quick reference of a command

```
# git <verb> -h
$ git config -h
```

- Manpage (full documentation)

```
# git help <verb>
$ git help config

# git <verb> --help
$ git config --help

# man git-<verb>
$ man git-config
```

If all you remember is..

- GitHub
  - *Not a VCS solution*, it hosts repositories that are managed with Git

If all you remember is..

- GitHub
  - *Not a VCS solution*, it hosts repositories that are managed with Git
  - Offers *additional functionality* on top of Git repositories, to foster:
    - Collaboration (Forks, Pull Requests)
    - Project Management (Issue tracking, Wiki, Project web page, ..)

If all you remember is..

- GitHub
  - *Not a VCS solution*, it hosts repositories that are managed with Git
  - Offers *additional functionality* on top of Git repositories, to foster:
    - Collaboration (Forks, Pull Requests)
    - Project Management (Issue tracking, Wiki, Project web page, ..)
- Git
  - States: Only cares about *tracked* files (modified, staged, committed)

If all you remember is..

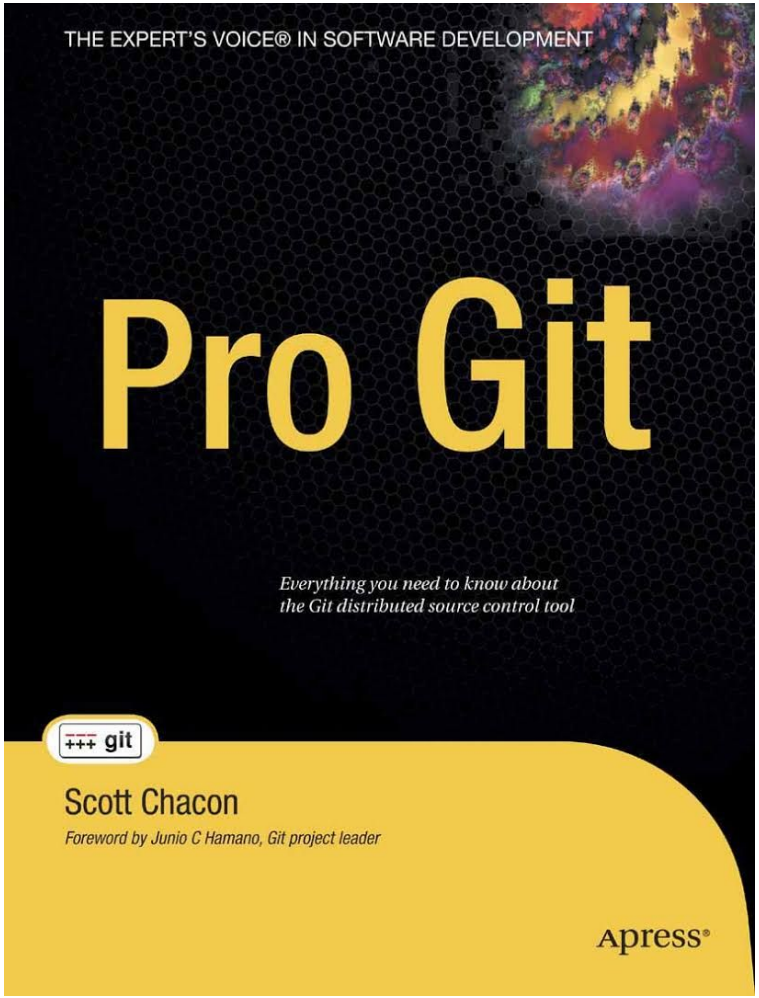
- GitHub
  - *Not a VCS solution*, it hosts repositories that are managed with Git
  - Offers *additional functionality* on top of Git repositories, to foster:
    - Collaboration (Forks, Pull Requests)
    - Project Management (Issue tracking, Wiki, Project web page, ..)
- Git
  - States: Only cares about *tracked* files (modified, staged, committed)
  - 3 Sections:
    - *Working directory*: current version, where you are working
    - *Staging area*: contains the file/s that will go in the next commit
    - *Git directory*: contains the data (local DB) for Git usage

Hands-on

Git Basics

*This tutorial is based on **Pro Git** by Scott Chacon*

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



Getting a Git repository in your system

1. Clone a remote (existing) repository

   With ***git clone*** == Full copy from remote

```
# Full copy (files, Git directory, ..) to ‘hellogitworld’ dir in current path
$ git clone https://github.com/masterdatascience-UIMP-UC/hellogitworld

# Full copy (files, Git directory, ..) to a different ‘myhellogitworld’ dir in current path
$ git clone https://github.com/masterdatascience-UIMP-UC/hellogitworld myhellogitworld
```

Getting a Git repository in your system

1. Clone a remote (existing) repository

   With ***git clone*** == Full copy from remote

```
# Full copy (files, Git directory, ..) to 'hellogitworld' dir in current path
$ git clone https://github.com/masterdatascience-UIMP-UC/hellogitworld

# Full copy (files, Git directory, ..) to a different 'myhellogitworld' dir in current path
$ git clone https://github.com/masterdatascience-UIMP-UC/hellogitworld myhellogitworld
```

- This command results in a copy of this repository in our system
  - Nothing yet in our Github account
- We will work locally on our copy
  - We may not have permissions for updating the remote repository

Getting a Git repository in your system

2. Initiate our Git repository from a directory in our system

With ***git init*** == Creates the .git directory

```
# Move to the directory meant to be the repository
$ cd /home/user/my_project

# Convert directory to Git repository (creates .git directory)
$ git init
```

Getting a Git repository in your system

2. Initiate our Git repository from a directory in our system

With ***git init*** == Creates the `.git` directory

```
# Move to the directory meant to be the repository
$ cd /home/user/my_project

# Convert directory to Git repository (creates .git directory)
$ git init
```

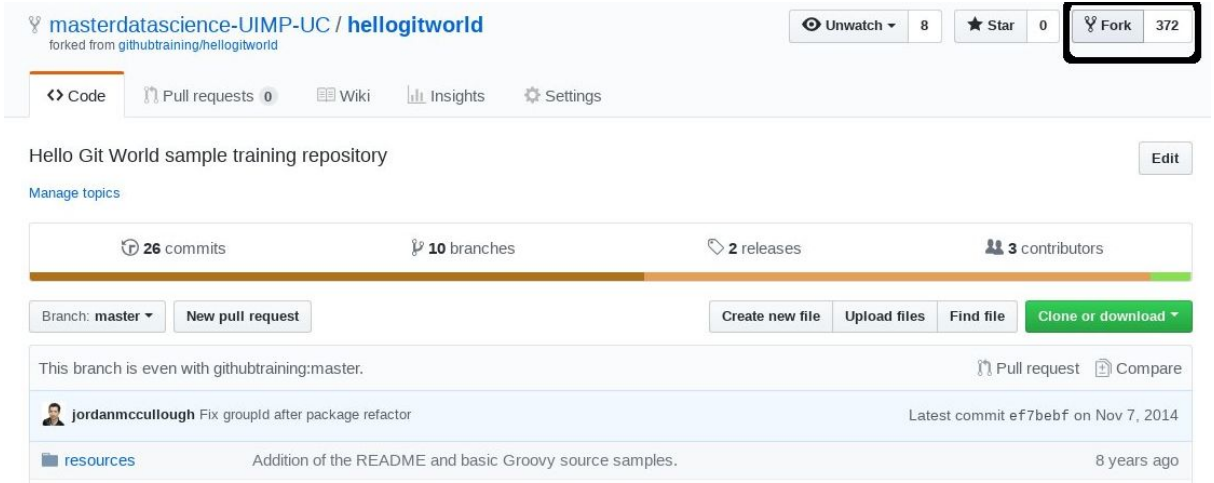
- Git does not track any file by default
  - Have to tell Git about the files we want to start tracking
- There is no remote repository in GitHub
  - We need to create one!

Creating a Git repository in GitHub

1. Create an empty repository
    - Go to `Repositories > New` and fill in the relevant information
    - Once done, we can either clone it (1) or push changes of an initialized one (2)

Creating a Git repository in GitHub

1. Create an empty repository
   - Go to `Repositories > New` and fill in the relevant information
   - Once done, we can either clone it (1) or push changes of an initialized one (2)
2. Fork a repository
   - A fork is a copy of a repository
   - Freely experiment with changes without affecting the original project
   - Go to the URL of the project to fork from and click on `Fork`



Example: Fork & Clone from GitHub

- Usual workflow for the exercises in the master..
  1. You will be given a GitHub repository URL with the exercises
  2. You will fork it to your GitHub personal account
  3. You will clone your fork to your system
  4. You will work on them, committing & pushing changes
  5. You will provide the fork URL to your teacher

Example: Fork & Clone from GitHub

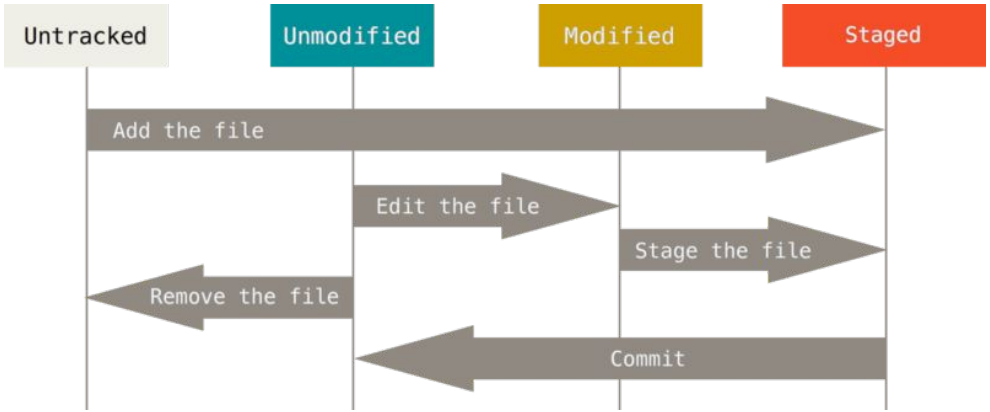
- Usual workflow for the exercises in the master..
  1. You will be given a GitHub repository URL with the exercises
  2. You will fork it to your GitHub personal account
  3. You will clone your fork to your system
  4. You will work on them, committing & pushing changes
  5. You will provide the fork URL to your teacher
- Let's do an example:
  1. (Web) Login to https://github.com
  2. (Web) Fork this repository: https://github.com/masterdatascience-UIMP-UC/hellogitworld
     i. Use your personal account as the target space
  3. (Terminal) Clone your fork:

```
# Full copy (files, Git directory, ..) to 'hellogitworld' dir in current path
$ git clone https://github.com/<your-account-name>/hellogitworld

# Enter in the 'hellogitworld' directory just created and list the files included
$ cd hellogitworld
$ ls -l
```

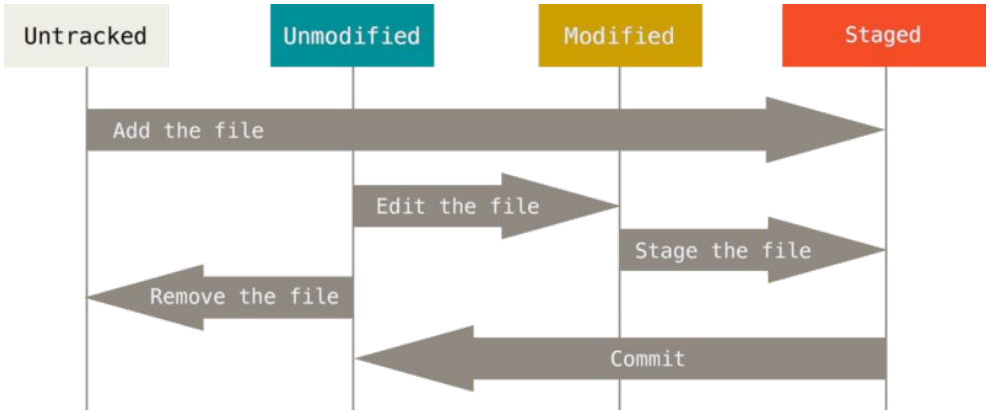
Checking the status of your files

- Status of files
  a. *Untracked*: Git does not know about these files
  b. *Tracked*: Git knows about
     i. *Unmodified*
     ii. *Modified*
     iii. *Staged*
- Workflow
  a. Edit files -> *Modified*
  b. Stage the required files -> *Staged*
  c. Commit them -> *Unmodified*



Checking the status of your files

- Status of files
  a. *Untracked*: Git does not know about these files
  b. *Tracked*: Git knows about
     i. *Unmodified*
     ii. *Modified*
     iii. *Staged*
- Workflow
  a. Edit files -> *Modified*
  b. Stage the required files -> *Staged*
  c. Commit them -> *Unmodified*
- ***git status*** command




```
# Inside ‘hellogitworld’ directory ==
$ git status

On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

A note on branches..

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Branching is more advanced topic that we will cover in a dedicated section of this tutorial

- For this introductory part, we will consider the default branch `master`

Tracking new files

```
# Create a new file called README
$ echo "My own README file" > README
```

Tracking new files

```
# Create a new file called README
$ echo "My own README file" > README

# Check the repository status
$ 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)

      README

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

Tracking new files

```
# Create a new file called README
$ echo "My own README file" > README

# Check the repository status
$ 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)

      README

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

- README is ***untracked*** (under *Untracked files*)
  - README was not in the previous snapshot (commit)
- Git just warns us, it won't do anything until we explicitly request so
  - Let's track the README file

Tracking new files

To track a new file we use ***git add***

```
# Start tracking README file
$ git add README
```

Tracking new files

To track a new file we use ***git add***

```
# Start tracking README file
$ git add README

# Check the repository status
$ 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: README
```

Tracking new files

To track a new file we use ***git add***

```
# Start tracking README file
$ git add README

# Check the repository status
$ 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: README
```

README is now ***staged*** (under *Changes to be committed*)

- Achieved by ***git add*** command
- At this point we could commit this change, resulting in the README file to be added to the repository content

Staging modified files (I)

Understanding **staging**..

```
# Same situation as before: README file was added
# We now modify an already tracked file --> fix.txt
$ echo “Fix #1 added -- You can safely remove this line --” >> fix.txt
```

Staging modified files (I)

Understanding **staging**..

```
# Same situation as before: README file was added
# We now modify an already tracked file --> fix.txt
$ echo “Fix #1 added -- You can safely remove this line --” >> fix.txt
$ 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:   README

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:   fix.txt
```

Staging modified files (I)

Understanding **staging**..

```
# Same situation as before: README file was added
# We now modify an already tracked file --> fix.txt
$ echo “Fix #1 added -- You can safely remove this line --” >> fix.txt
$ 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:   README

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:   fix.txt
```

fix.txt is ***unstaged*** (under *Changes not staged for commit*)

- == modified in the working directory but not staged
- Use ***git add*** command to stage it

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
```

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
# Check the status
$ 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:   README
    modified:   fix.txt
```

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
# Check the status
$ 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:   README
    modified:   fix.txt
```

Both README and fix.txt are ***staged*** and will go into the next commit

Staging modified files (II)

Understanding **staging**..

```
# Stage fix.txt file
$ git add fix.txt
# Check the status
$ 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:   README
    modified:   fix.txt
```

Both README and fix.txt are ***staged*** and will go into the next commit

..but you remember one little change that you want to make to fix.txt before you commit it..

Staging modified files (III)

Understanding **staging**..

```
# Add the last minute change to fix.txt
$ echo “Fix #2: Very important fix added -- You can safely remove this line --” >> fix.txt
```

Staging modified files (III)

Understanding **staging**..

```
# Add the last minute change to fix.txt
$ echo “Fix #2: Very important fix added -- You can safely remove this line --” >> fix.txt
# Check the status
$ 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:   README
    modified:   fix.txt

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:   fix.txt
```

Staging modified files (III)

Understanding **staging**..

```
# Add the last minute change to fix.txt
$ echo “Fix #2: Very important fix added -- You can safely remove this line --” >> fix.txt
# Check the status
$ 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:   README
    modified:   fix.txt

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:   fix.txt
```

fix.txt is ***unstaged*** and ***staged*** at the same time ?¿

- If you commit now, only the first version of fix.txt (“Fix #1..” as it was when we ran ***git add***) will be considered
- You have to ***git add*** fix.txt to stage it, so it goes in the next commit

Staging modified files (IV)

Understanding **staging**..

```
# Stage the last change of fix.txt == ‘Fix #2..’
$ git add fix.txt
```

Staging modified files (IV)

Understanding **staging**..

```
# Stage the last change of fix.txt == ‘Fix #2..’
$ git add fix.txt
# Check the status
$ 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:   README
    modified:   fix.txt
```

Committing your changes (I)

To commit a new change we use ***git commit***

Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit makes a snapshot of your project you can later on compare/revert to
  - A commit may involve multiple changes (although atomic ones are preferred)
  - Anything not staged won't be part of the next

Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit makes a snapshot of your project you can later on compare/revert to
  - A commit may involve multiple changes (although atomic ones are preferred)
  - Anything not staged won't be part of the next

```
# Ou changes are all staged, so we commit..
$ git commit
```

Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit makes a snapshot of your project you can later on compare/revert to
  - A commit may involve multiple changes (although atomic ones are preferred)
  - Anything not staged won't be part of the next

```
# Ou changes are all staged, so we commit..
$ git commit
```

With no options, git commit opens the default editor (core.editor from git config), with the content:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#     new file:   README
#     modified:   fix.txt
```

Committing your changes (I)

To commit a new change we use ***git commit***:

- A commit makes a snapshot of your project you can later on compare/revert to
  - A commit may involve multiple changes (although atomic ones are preferred)
  - Anything not staged won't be part of the next

```
# Ou changes are all staged, so we commit..
$ git commit
```

With no options, git commit opens the default editor (core.editor from git config), with the content:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#     new file:   README
#     modified:   fix.txt
```

- A **descriptive commit message** must be added to the blank line above the default text
- The default content can be left, just to remind in the future what files you have changed

Committing your changes (II)

A one-liner approach to git commit

```
# Our changes are all staged, so we commit..
$ git commit -m “New README and the two first fixes documented”
[master 1a90007] New README and the two first fixes documented
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

Committing your changes (II)

A one-liner approach to git commit

```
# Our changes are all staged, so we commit..
$ git commit -m “New README and the two first fixes documented”
[master 1a90007] New README and the two first fixes documented
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

Your first commit is done!

- To branch `master`
- With ID `1a90007`

Moving/renaming files

To move or rename tracked files we use ***git mv***:

```
# Let’s rename the README file..
$ git mv README README.to_delete
```

Moving/renaming files

To move or rename tracked files we use ***git mv***:

```
# Let’s rename the README file..
$ git mv README README.to_delete

$ 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)

    renamed:    README -> README.to_delete
```

Removing files

To remove a tracked file we use ***git rm***:

```
# Let’s remove README.to_delete file
$ git rm README.to_delete
error: the following file has changes staged in the index:
    README.to_delete
(use --cached to keep the file, or -f to force removal)
```

Removing files

To remove a tracked file we use ***git rm***:

```
# Let’s remove README.to_delete file
$ git rm README.to_delete
error: the following file has changes staged in the index:
    README.to_delete
(use --cached to keep the file, or -f to force removal)

$ git rm -f README.to_delete
rm ‘README.to_delete’

$ 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)

    deleted:    README
```

The next commit will eliminate the file and will be no longer tracked

Removing files

To remove a tracked file we use ***git rm***:

```
# Let’s remove README.to_delete file
$ git rm README.to_delete
error: the following file has changes staged in the index:
    README.to_delete
(use --cached to keep the file, or -f to force removal)

$ git rm -f README.to_delete
rm ‘README.to_delete’

$ 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)

    deleted:    README
```

The next commit will eliminate the file and will be no longer tracked

Viewing Commit History

To view the commit history of our project we use ***git log*** :

Viewing Commit History

To view the commit history of our project we use ***git log***:

```
$ git log
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> master)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

commit ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/master, origin/HEAD)
Author: Jordan McCullough <jordan@github.com>
Date:   Fri Nov 7 11:27:19 2014 -0700

    Fix groupId after package refactor

commit ebbbf773431ba07510251bb03f9525c7bab2b13a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 13:00:35 2014 -0700

    Update package name, directory

commit 45a30ea9afa413e226ca8614179c011d545ca883
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:59:55 2014 -0700

    Update package name, directory

commit 9805760644754c38d10a9f1522a54a4bdc00fa8a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:19:02 2014 -0700

    Fix YAML name-value pair missing space
```

Viewing Commit History

To view the commit history of our project we use ***git log***:

```
$ git log
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> master)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

commit ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/master, origin/HEAD)
Author: Jordan McCullough <jordan@github.com>
Date:   Fri Nov 7 11:27:19 2014 -0700

    Fix groupId after package refactor

commit ebbbf773431ba07510251bb03f9525c7bab2b13a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 13:00:35 2014 -0700

    Update package name, directory

commit 45a30ea9afa413e226ca8614179c011d545ca883
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:59:55 2014 -0700

    Update package name, directory

commit 9805760644754c38d10a9f1522a54a4bdc00fa8a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:19:02 2014 -0700

    Fix YAML name-value pair missing space
```

- Most recent commits show up first
- Info per commit:
  - commit ID
  - Author's name & email
  - Date
  - Author's commit message

Viewing Commit History

git log command has several interesting options:

Viewing Commit History

git log command has several interesting options:

- -p or --patch

```
$ git log -p -1
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> master)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

diff --git a/README b/README
new file mode 100644
index 0000000..29afdfa
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My own README file
diff --git a/fix.txt b/fix.txt
index e69de29..3e6b239 100644
--- a/fix.txt
+++ b/fix.txt
@@ -0,0 +1,2 @@
+Fix #1 added -- You can safely remove this line --
+Fix #2: Very important fix added -- You can safely remove this line --
```

Viewing Commit History

git log command has several interesting options:

- -p or --patch

```
$ git log -p -1
commit 1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> master)
Author: Pablo Orviz <orviz@ifca.unican.es>
Date:   Mon Oct 1 15:33:36 2018 +0000

    New README and the two first fixes documented.

diff --git a/README b/README
new file mode 100644
index 0000000..29afdfa
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My own README file
diff --git a/fix.txt b/fix.txt
index e69de29..3e6b239 100644
--- a/fix.txt
+++ b/fix.txt
@@ -0,0 +1,2 @@
+Fix #1 added -- You can safely remove this line --
+Fix #2: Very important fix added -- You can safely remove this line --
```

- --pretty

Options for --pretty:

- oneline
- short
- full
- Format (customizable)

```
$ git log --pretty=oneline
1a900077fdeffd73e0c7412a731949c540c78250 (HEAD -> master) New README and the two first fixes documented.
ef7bebf8bdb1919d947afe46ab4b2fb4278039b3 (origin/master, origin/HEAD) Fix groupId after package refactor
ebbbf773431ba07510251bb03f9525c7bab2b13a Update package name, directory
```

Working with remotes

So far we have been working locally on our computer..

Working with remotes

So far we have been working locally on our computer..

- Remote repositories are versions of your project that are hosted on the Internet (e.g. GitHub)
- Remote repositories could be either controlled by you, such as forks (write permissions), or collaborative (owned by others, read permissions)
- Collaboration == Manage remote repositories
  1. Add & Remove remotes: with ***git remote***
  2. Push & Pull changes to remote repositories, with ***git push*** and ***git pull***

Showing your remotes

```
# Lists shortnames of each remote
$ git remote
origin
```

Showing your remotes

```
# Lists shortnames of each remote
$ git remote
origin

# Lists shortnames of each remote together with their URLs
$ git remote -v
origin  https://github.com/orviz/hellogitworld (fetch)
origin  https://github.com/orviz/hellogitworld (push)
```

- `origin` is the default name Git gives to the remote repository you have cloned from
- ***`git clone`*** adds the `origin` remote for us

Adding remote repositories

Let's add the repository we forked from `https://github.com/masterdatascience-UIMP-UC/hellogitworld` as a remote repository

```
# Adds a remote repository named 'upstream'
$ git remote add upstream https://github.com/masterdatascience-UIMP-UC/hellogitworld
origin
```

Adding remote repositories

Let's add the repository we forked from `https://github.com/masterdatascience-UIMP-UC/hellogitworld` as a remote repository

```
# Adds a remote repository named 'upstream'
$ git remote add upstream https://github.com/masterdatascience-UIMP-UC/hellogitworld
origin

# List remotes
$ git remote -v
origin https://github.com/orviz/hellogitworld (fetch)
origin https://github.com/orviz/hellogitworld (push)
upstream  https://github.com/masterdatascience-UIMP-UC/hellogitworld (fetch)
upstream  https://github.com/masterdatascience-UIMP-UC/hellogitworld (push)
```

Now we can refer to `upstream` for any operation we may need to do with this remote repository

Fetch and pull from your remotes

git fetch allows us to retrieve data from remote repositories

- Format: ***git fetch <remote>***
- In collaborative scenarios, your local repository might be behind the last version of the remote repository
  - By fetching, Git gets any new work from the previous time you ran the command
- Only downloads the data to your local repository, ***git fetch*** does not merge it with your current version
  - You would need to merge it manually with ***git merge*** command

Fetch and pull from your remotes

git fetch allows us to retrieve data from remote repositories

- Format: ***git fetch <remote>***
- In collaborative scenarios, your local repository might be behind the last version of the remote repository
  - By fetching, Git gets any new work from the previous time you ran the command
- Only downloads the data to your local repository, ***git fetch*** does not merge it with your current version
  - You would need to merge it manually with ***git merge*** command

git pull can be seen as a combination of ***git fetch*** + ***git merge***

- Format: ***git pull <remote> <branch>***
- Will try to automatically fetch and merge the remote version into the current version you are currently working on (locally)
- More convenient, although may not work in all the scenarios

```
# Pulls from remote repository labelled as ‘origin’
$ git pull origin master
From https://github.com/orviz/hellogitworld
 * branch            master     -> FETCH_HEAD
Already up to date.
```

Push to your remotes

git push allows us to send our committed changes to a remote repository

- Format: ***git push <remote> <branch>***

Push to your remotes

git push allows us to send our committed changes to a remote repository

- Format: ***git push <remote> <branch>***

```
# Pushes changes to the repository labelled as ‘master’
$ git push origin master
```

This command will only work if we have write permissions to the remote repository, otherwise it will fail with a permission denied error.

Push to your remotes

git push allows us to send our committed changes to a remote repository

- Format: ***git push <remote> <branch>***

```
# Pushes changes to the repository labelled as ‘master’
$ git push origin master
```

This command will only work if we have write permissions to the remote repository, otherwise it will fail with a permission denied error.

Exercise (understand permissions):

- Compare the result of pushing to the ‘origin’ and ‘upstream’ remote repositories

Exercise with remotes

Handling more than one copy of a remote repository

In a real scenario you commonly do this from different locations (computers), but here we will use the same one with two different copies/directories: ***hellogitworld*** *and* ***hellogitworld-2***

1. Clone a second copy of your forked repository.
   - Name the second copy hellogitworld-2
   - ***git clone -h*** for getting summarized help
2. cd into the directory just created for the second copy
   - Make the following two changes **in two different commits**:
     i. Modify a tracked file
     ii. Add a new file
   - Push the changes to your remote (fork) repository => origin
3. cd into the directory of the first copy (aka hellogitworld)
   - Check with Git log the last commit done
   - Update (Pull) your working directory with last version of origin
   - Check with Git log the last commit done
     i. Has anything changed in the log?
   - Check that the files you modified with hellogitworld-2 are there

Useful commands:

```
$ git clone <github-repository>
<local-directory>
$ git add <file>
$ git commit -m “Put your commit msg here”
$ git remote -v
$ git push <remote-name> master
$ git log  (press ‘q’ to exit)
$ git pull <remote-name> master
```

If all you remember is..

- A **fork** is a GitHub feature that copies a remote repository in your local account
  - Forks will be the most common way to deliver work to teachers
- ***git clone*** creates a local copy of a remote repository
  - Sets the remote repository as origin
  - Sets the current branch as master
- ***git add*** is a multipurpose command:
  - Begin tracking new files
  - Stage files
- ***git commit*** gathers all the staged changes and creates a snapshot of the project
- ***git pull*** gets last updates from a remote repository and merges them with the current version in the local working directory
- ***git push*** sends committed changes to a remote repository

Hands-on

Collaborative work

GitHub Pull requests

GitHub feature that allows you to propose changes to others

- Primary source of collaboration
- *“Once a PR is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch”* (github.com)
- Owner/s of the repository are the one/s that eventually accept/reject the change
    - If accepted, the change will be typically merged into the owner’s `master` branch

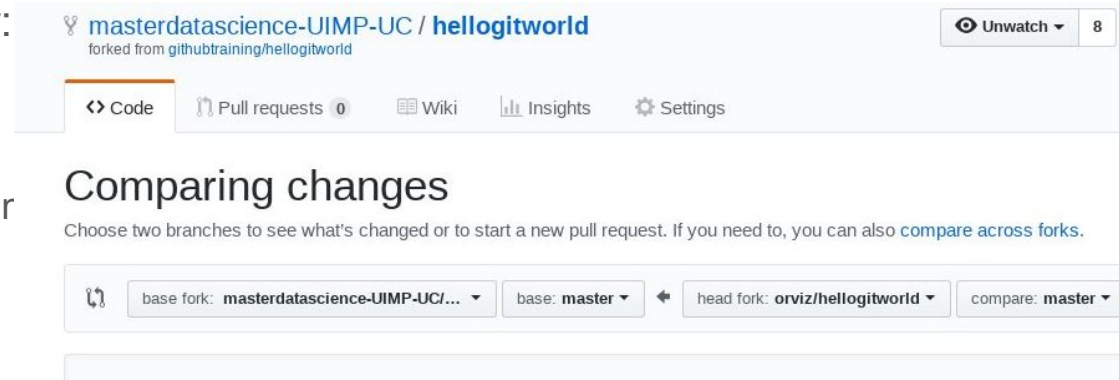
GitHub Pull requests

The importance of review..

- GitHub offers, through PRs, a nice **place for discussion** => code review
- Owner/s, collaborator/s and (if public) any external user/expert can comment and suggest further modifications, e.g.:
  - Goal/scope of the change
  - If source code, any suggestion to optimize the execution (efficiency, efficacy) of the code
  - Commit/s message/s description
- Code review process may result in the change being merged into the production version (`master` branch) => extra careful!
  - Usually is complemented with the execution of a set of automatic tests => GitHub Integrations

Creating a Pull Request from a Fork

Pull requests can only be created if the source & target repositories differ:



1. Navigate to the forked repository page ir
2. For our example:
   a. Be sure to select branch master
3. Click on New pull request button
4. In the *Comparing Changes* page you have:
   a. *Base fork*
      i. Target repository, the one that we want to add the changes
   b. *Head fork*
      i. Our repository, the one we were working in
5. Type a title and description for the PR
6. Click on Create pull request button

Exercise with Pull Requests

Adding your Fork URL to the teacher's repository

We will simulate the delivery of an assignment to a teacher's repository.

1. (GitHub) Create a fork of the following teacher's repository in your personal account: https://github.com/masterdatascience-UIMP-UC/gittutorial-teacher-repo
2. (Terminal) Clone your forked repository
3. (Terminal) Create a new file by copying from assignment.txt
   - Name it assignment-<name-and-last-name>.txt
4. (Terminal) Add your Fork URL to the student-exercises-urls.txt
5. (Terminal) Commit both changes
6. (Terminal) Push the change to your remote fork
7. (GitHub) Create a Pull Request

Maintaining a fork up to date