---

preliminaries

preparation

To install `git` in a linux machine:

```
$ sudo apt-get install git-core
```

With the following commands some info will be added to your `git` configuration file (`~/.gitconfig`):

```
$ git config --global user.name "Despo Panoglou"
$ git config --global user.email "desprh@yahoo.gr"
```

You can override this data for a specific project, running the same commands but without the `--global` option. The changes will be recorded in `.git/config`, where `.git` is a directory located in the project's folder.

With

```
$ man git config
```

you access a manual with all the options.

terminology

- We modify our code, add/delete files, change the subdirectory structure etc. From time to time (usually after we are sure that the changes we made work alright), we want to register those changes. then we make a *commit*. Finally, we *push* those registered changes in the main repository. Every time we *push*, we actually send all the commits we made since we last *push*ed.
- In cases we have two different versions or branches of a project, we often want to merge the two into one. Then we say we *merge*.

  A very simple case we might need to merge is the following: Person A makes some changes in a code, person B makes some other changes. Both of the modification sets have been requested, so now we want everything *merge*d together in a single code.

  In some cases, there might exist some conflicts, e.g. if a specific piece of one file was modified by both people. Then *git* could not possibly know which change to select and put forward. In those cases there will be a warning message, and the person whoc attempts to merge the two versions has to select one of the two for this exact point within the code.

  In some other cases out project will not work well, but we will not be able to know it until we run. Those cases are far more tricky, but also are those cases that programmers very often encounter, when they work alone on one code: You make a change and then everything goes wrong! But *git* cannot possibly foresee this kind of problems, and therefore after every *merge*, you have to attempt a run and check it out.
- *Branches* are some kind of different routes an initial version of a project might take. Whenever you create a branch, it is identical to the *parent branch* (meaning the branch that was being used at the time of creation of the new branch), i.e. whatever exists in the parent branch will be present in the new branch.

  Why this happens is obvious. We want to start from a point where our code works in a stable way, and we want to test or experiment on something, without harming the parent initial version.

general information

To check the commit history of the project, including the commit ID, the name of the person who made each commit, the date and the commit message, you have to type `git log`. This will print a list of the commit history - by pressing "space" you can reach the beginning of the git history. Here we show an example of the last three commits:

```
$ git log
commit 1710b030a285861f4ec4a8b2f487fef5599c2f08
Author: Despo Panoglou <desprh@yahoo.gr>
Date:   Mon Feb 10 18:05:15 2014 +0200

    updated the HDUST manual
```

```
     * added a section of the list of runs, describing the sample master file
       step by step
     * added a section on the SIMULATION section, with information of the
       different modes (STEP's)
     * some additional minor changes here and there

commit 633c60fd49c84c4e069b87eca3e925743e4356b7
Author: Despo Panoglou <desprh@yahoo.gr>
Date:   Mon Feb 10 01:23:32 2014 +0200

    added a first manual for HDUST

commit cb9e1f9b2409e641d4a7c571230e0f33c5b8fa47
Author: Dan Moser <dmfaes@gmail.com>
Date:   Thu Feb 6 17:39:32 2014 -0200

    Primeiro commit de Moser
```

So if you want to see the changes between the last commit by me and the commit of Daniel (see notes on commit ID), you have to type:

```
git diff 1710b030a2..cb9e1f9b
```

When the changes are made on text files, it's much easier to compare, whereas when the changes are made on binary files (such as `pdf`), there's not much to understand (at least for me). That's why `git` is mainly used for programming codes.

Other commands which provide information:

```
$ git log --stat   # shows the commit history (lines changed in each file)
$ git log foo.txt  # shows the commit history of file "foo.txt"
$ git status       # shows whether there are changes to be commited
$ git diff         # see what is there to be commited (differencies from last commit)
```

---

creation of a repository

in bitbucket

To create a new repository in `bitbucket`

1. Log in in bitbucket
2. At the top of the page, click on `Create` (a new repository.)
3. See here for how to replicate and process a project at this new repository.

Please, DO NOT do in `bitbucket` anything else than view. DO NOT upload into or download things from bitbucket. In general DO NOT use the visual interface of bitbucket. Use only the command line. When our repository here is ready, we will certainly need to do everything from the command line. So you should better learn the real commands and NOT `bitbucket`'s interface.

What you expect to do by using the `upload` button of `bitbucket`, you should do with add+commit+push. And whatever you expect to accomplish by using the `upload` button of `bitbucket`, you should do with `git pull`. It will be quicker in the first place (ignoring the fact that you will have to enter your password). *[dP: Add information on how you can save the password, so as to get over this delay.]*

local repository

To create a new repository in your own computer:

```
$ REPOS=/jungle/Backup/repos
$ SRC=/home/despo/Documents/programs/tests
$ SRCGIT=tests.git
$ cd $REPOS
$ git init --bare $SRCGIT
$ cd $SRC
$ git init
$ git add .
$ git commit -m "my first comment: commit" .
$ git remote add origin $REPOS/$SRCGIT
$ git push -u origin master
```

---

commits

commit ID

The full ID of a commit is a 40-character string. As `git` makes sure that the first few characters of the full ID is a unique sequence among the distinct versions of the same repository, you can refer to each commit with its ID's first few characters (usually 6 characters are enough).

selection of files

After you make some changes in any of the files within the project directory, you can give the following commands:

```
$ git add .
$ git commit .
```

The dot (`.`) in the end of both commands means add+commit of all changes encountered in current directory. If you don't want to add+commit the changes in all files, but only the changes in file `foo.txt`, then the above commands have to be given as:

```
$ git add foo.txt
$ git commit foo.txt
```

comments

Let's explore the command:

```
$ git commit -m "first commit" .
```

If you omit the "-m message" option, then by running the commit command, it will prompt you for a message. Instead of the short form allowed in the command line, now you will be allowed to give a short message of 50 characters, then leave an empty line, and finally write a more detailed description of the changes incorporated in this commit (as in the first commit message of the sample that is shown in § general information).

---

branches

branching

Imagine that you have a 1D code and are about to make changes in order to add a 2D option.

```
$ git branch          # list existing branches (* points to current/active branch)
  master
* onedim
$ git branch twodim   # create a new branch initially identical to onedim
$ git checkout twodim # switch to (enter) the branch "twodim"
$ git branch          # list existing branches (* points to current/active branch)
  master
  onedim
* twodim
$ vi Makefile         # modify some file(s)
```

If you now go back to the master branch or the `onedim` branch, with

```
$ git checkout {master|onedim}
```

you will see that none of the changes you made in `Makefile` can be found there. To add to the original repository the branch you created in your local directory:

```
$ git push --set-upstream origin twodim
```

Your current branch is `twodim`. You can merge it to `onedim` by

```
$ git merge onedim
```

If there is any conflict in merging the two branches, it will return a relevant message. You can see the conflicts by:

```
$ git diff
```

Remember that the same command can be given before any commit, in order to see what there is to be commited, i.e. the differencies from last commit (in current branch; see § general information).

If you want to delete a branch:

```
$ git branch -d onedim   # keeping it in the history tree
$ git branch -D onedim   # removing it permanently from the history tree
```

If you get stuck with the conflicts and want to start over:

```
git reset --hard HEAD      # in case you have not yet commited the merge
git reset --hard ORIG_HEAD # in case you have already commited the merge
```

You can check the differencies between two branches with

```
git diff onedim..twodim
```

versions

Let's say we have a big project that has been being developed for years, and there are more than one versions of it, say version 1 and version 2, which are located in directories `Dv1` and `Dv2`, respectively. I will create a short shell script (no need for a long one) on how to create a git repository that will include the two versions (if you copy it, make sure to correct the directory names; it is recommended that you type the commands one by one, though). If you set `Dv1` and `Dv2` as shown in the first two lines, the rest of the lines may be copied as they are. You should check the path names, while you might also want to change the path/name of the repository directory (`TARGET`).

Each directory may be accessed both locally and internally, since we'll use the command `rsync`, which may contact remote systems via a remote shell program (e.g. `ssh`) or through contacting an `rsync` deamon directly via TCP. We will suppose that `Dv1` is in `alphacrucis`, and that `Dv2` is in my `home` directory:

```
Dv1=despo@alphacrucis.iag.usp.br:/sto/home/despo/HDUSTv1
Dv2=$HOME/Projects/HDUSTv2
TARGET=$HOME/repos/HDUSTworking
rsync -avz $Dv1 $TARGET
cd $TARGET
git init
git add .
git commit -m "version 1" .
```

The execution of the last command will output a few lines that describe the differences that were commited.

```
commit 1fd89e9c014ea69982ee2f941d55066c3d58b32f
Author: Despo Panoglou <desprh@yahoo.gr>
Date:   Thu Feb 6 15:07:20 2014 +0200

    version 1
```

It's you first commit, no changes on files, therefore no more information are printed out. You might also see something as simple like this.

```
 [master 1fd89e9] version 1
```

Which form of message you will see depends on the configuration. In any case, the first of these line contains a 40-character string, which is the full ID of the commit, whereas only the first part of it (`1fd89e`) is reported in the short version. But we don't want to have to remember this strange number, even if it is as short as 6 characters long. We can tag this version of the code with the following command:

```
$ git tag -a "HDUSTv1" 1fd89e9c014ea69982ee2f941d55066c3d58b32f
```

so that now we can refer to it just by `HDUSTv1`. Although just by tagging each version we make it possible to turn back to it at any time, for various reasons it is good that we also make a branch for version 1.

```
$ git branch version1
```

As we are still in the beginning, we are in the master branch, and by the last command we created another branch called `version1`.

Now we have to enter version 2 in the repository.

```
$ rsync -avz --delete  --exclude '*.git*' $Dv2 $TARGET
```

Take caution not to add any slashes in the end of the directory names. This command should substitute everything in current directory with the contents of our local copy of version 2, deleting everything from the old version that does not exist in the new version (except from the `*.git*` files, otherwise it would delete all previous information about the repository, i.e. it would forget all about version 1, including the previous logs).

```
$ git add .
$ git commit -m "version 2" .

[master 1cf4231] version 2
 39 files changed, 39 insertions(+), 88 deletions(-)
 mode change 100644 => 100755 README
  [. . .]
```

We tag this commit and branch it as in the case of version 1:

```
$ git tag -a "HDUSTv2" 1cf4231
$ git branch version2
```

Say we don't yet have a version 3 of the code. In case we do have an intermediate version in some other local directory `$Dv3`, we may copy it as before.

```
$ rsync -avz --delete  --exclude '*.git*' $Dv3 $TARGET
```

Now we can add+commit, but there's no need to tag+branch, as we're still working on it. We make some changes, add+commit and so on, until we do have a stable version 3, and that's when we tag+branch.

At any time we can switch to previously created branches, e.g.

```
$ git checkout version1
```

and then we can return to our master branch:

```
$ git checkout master
```

At any time we can see the differences between two tags, branches or commits, e.g.

```
$ git diff 1cf4231..1fd89e9
$ git diff version1..master
$ git diff HDUSTv2..HDUSTv1
```

---

replication

evaluation of methods

There are ways to copy/replicate a `git` repository. The easiest and simplest is "cloning". "Adding" is just referred to for completeness. I am not aware of any real differencies between the two ways. But I say that cloning is simpler for the following (minor) reasons:

- In "adding", you have to create and enter the new directory, while in "cloning" the directory is created at the time of "cloning".
- In "cloning" defaults are automatically assumed, and you do not have to configure anything (except if you want to change the defaults), while in "adding" you have to spend some time configuring (although not so long), after you have just added the repository.

add

1. Go to the directory where you wish to work on the project:

   ```
   $ mkdir test
   $ cd test
   $ git init
   $ git remote add origin https://desprh@bitbucket.org/desprh/test.git
   $ git pull https://desprh@bitbucket.org/desprh/test
   ```

2. Every time you have made some changes (e.g. create/modify/delete a file), you'd better `add` those changes and `commit` them to the current branch (`master`). You can add+commit as often as you want.
3. After one or more `commit`s, you may `push` to the original repository. You can `push` as often as you want.

   ```
   $ git push --set-upstream origin master
   ```

   After the first time you give that command (where you have defined the `upstream`, i.e. where you are psuhing to by default), you can simply `push` with

   ```
   $ git push
   ```

   Note: Instead of setting the `upstream` in the `push` command as above, you can simply set the `upstream` with the command

   ```
   $ git config --global push.default simple
   ```

clone

To clone an existing repository (that was created e.g. in `bitbucket` as explained in § in bitbucket):

```
git clone https://desprh@bitbucket.org/desprh/test.git
```

The name of the directory that will be created will be `test`

```
$ cd test
$ git add .
$ git commit -m "first commit" .
$ git push
```

When you have cloned, by default it pushes directly to the master branch of the repository.

You go home and want to update what is done. Provided that in the past you have cloned the repository, you receive the changes since last commit by:

```
$ git pull
```

Remember that `pull=fetch+merge`, i.e. `git` goes to the repository, fetches the changes, and merges them with your current version. `pull` always merges with your current branch.

example

Let's try to clone a project that is in one of Olivia's directories (`/home/olivia/hdust1`). Provided that the directory `/home/olivia/hdust1` includes a `.git` directory, Tony can clone this directory and work on her own version, in the same machine.

```
tony:$ cd /home/tony/hdust2
tony:$ git clone /home/olivia/hdust1 hdust2
```

Tony modifies the code (i.e. commits and adds), and Olivia wants to receive those changes. So Olivia merges with her own version:

```
olivia:$ cd /home/olivia/hdust1
olivia:$ git pull /home/tony/hdust2
```

This will fetch and merge Olivia's version with Tony's. But usually this is not what we want, especially when changes on the code are performed continuously by more than one persons. Note that `pull=fetch+merge`, and Olivia would rather fetch and keep the clone separately from her current version. So what we do is the following:

```
olivia:$ git remote add tony /home/tony/hdust2
olivia:$ git fetch tony
```

In this way, Tony's version is kept in a separate branch, `tony/master`.
This command will show the changes made by Tony since she branched from Olivia's master branch:

```
olivia:$ git log -p master..tony/master
```

Remember that we are currently in Olivia's master branch, so if logs look OK, Olivia can merge:

```
olivia:$ git merge tony/master
```

Olivia has merged and then modified, added, commited, pushed to the main repository. After that Tony can update with all those changes:

```
olivia:$ git pull
```

Note that Tony does not have to give the path to Olivia's repository, since when he first cloned it was exactly from it, therefore this is the default where `git` pulls from and pushes to. So by default this command has been given automatically:

```
tony:$ git config --get remote.origin.url /home/olivia/hdust1
```

If you want to change the default repository, you have to give that same command with `/home/olivia/hdust1` substituted by the new repository. The complete configuration created during cloning is shown with

```
$ git config -l
```