Git Handout 1*

Houtan Bastani†‡

September 10, 2012

Topics addressed today will be for the general git user. We'll go over what git is, setting up your environment, accessing a repository from a server and the usual workflow that you'll follow. I hope to address the most common git commands. After this class, you should be able to handle 80% of your interactions with git. At the beginning it'll be annoying and you'll feel like git is slowing you down. But, with time, as you become more comfortable with the idea of git and the commands listed below, you'll become more accustomed to the way git works and be able to learn more on your own.[1]

0.1 How To Approach Learning Git

When you begin learning git, things can feel a bit overwhelming. You can feel as though you're swimming in a soup of git adds, commits, fetches, pulls, and on and on. To avoid this "git command soup" (and the resultant frustration) make sure you understand the big picture before you focus on the details (*i.e.,* git commands). Try to understand why you're using git,the general git workflow, and the appropriate action to take in a given situation (which can be project dependent). After you understand what it is you're trying to accomplish and how git works, the commands will become much clearer and easier to pick up. ☺

1 What is Git?

- Git is a distributed version control system, developed by Linus Torvalds (of Linux kernel fame).
  - **Version Control System**: Something that keeps track of the changes made to files (`commit`)
  - **Distributed**: Everyone has a copy (`clone`) of the git repository
  - So, a **Distributed Version Control System** allows you to have a copy of the complete development history of the project on your local workstation (*i.e.*, all changes ever made in the history of the project are in on your computer!!).
- This means that at any moment you can go back to the state of the project as it existed at any other point in time. (`reset`)
- Because of the distributed nature of git (*i.e.*, everyone has a repository clone on their computer), they can develop locally and, when ready, send (`push`) those changes to the global repository (*i.e.*, to the server).

*Handout available at: http://www.dynare.org/houtan/handouts

†Dynare & GPM Teams, CEPREMAP

‡houtan.bastani@ens.fr

[1] Disclaimer: Git is a software tool to help you organize and facilitate your *actual* work. No one sets out to learn git for the fun of it. To that end, I have learned the things that I need to do in order to perform my work and am by no means a git expert. Herein, I hope to share with you the things I found most important that may or may not help you, as it will depend primairly on what you need to do with git and how you work.

2 Before You Do Anything Else

At the command prompt, set up your global git information. This will allow you to be identified by name rather than just by email. Do the following:

```
> git config --global user.name "Your Name"
> git config --global user.email you@something.something
```

You only have to do this once on every computer that you use.

3 Cloning A Repository

When working on a project with others, you'll need to clone the global git repository. This command will copy the current state of the git repository from the server to your workstation. The repository is completely self-contained and can be copied from one directory to another and from one computer to another. As mentioned above, it contains the entire development history of the project. To get this copy of the global git repository onto your computer, do the following:

```
> git clone <<repository.git>>
```

There are several different protocals that you can use to connect to a git repository (*e.g.*, `http, ftp, ssh, rsync`). For example, to clone the Dynare repository, you could do:

```
> git clone http://www.dynare.org/git/dynare.git
> git clone ssh://yourlogin@kirikou.dynare.org/srv/d_kirikou/git/dynare.git
```

where `http` is open to everyone and `ssh` is available to those with access to `kirikou.dynare.org`. Of course, if you're working directly on the sevrer where the git repository lies, you can simply do:

```
> git clone /srv/git/repoName.git
```

where `/srv/git/repoName.git` is the full filename of the git repository.

4 Basic Workflow

Ok, you've done it! Now you have a copy of the git repository on your computer. Now what? Well, if you're a developer, you may want to actually edit files in the repository and then share the changes you've made with the rest of the team. As this will represent the majority of your interactions with git, it seems like a good place to start.

4.1 Updating

4.1.1 Branches[2]

Before we can talk about updating your local repository, we should talk about the concept of branches within git. Remember, your local repository is a copy of the global repository. So, it is useful to know BOTH where the current state of the global repository is and where the current state of your local repository is. When you first clone the repository, you'll have:

- `origin/HEAD`: tells you the branch that's checked out by default when you clone the repository (usually it points to `origin/master`)

[2]This can become more complicated than the cursory/incomplete treatment I'm providing here. If this topic becomes confusing, feel free to ignore it until you are more comfortable with the basic git concepts/commands.

- `origin/master`: indicates the main development branch of the repository
- `master`: is your local pointer that tracks `origin/master`. When you make changes, this pointer will advance past `origin/master`. Once your changes have been `pushed` to the global repository, `origin/master` will come into line with your local `master` branch

The command:

```
> git branch
```

tells you which branch you're currently on. One great benefit of git is that you can have several branches, both locally and on the server. This is useful for software releases (*e.g.*, at Dynare, in addition to the master branch, we have a branch for every major release) as well as just for testing ideas out locally.

4.1.2 Daily Updating Procedures

Before editing files within a repository, you should *always* do the following:

```
> git branch
```

to ensure you're on the `master` branch. Then,

```
> git fetch
> git rebase origin/master
```

where, `origin` is an alias for a `remote`[3] and `master` is the name of a `branch`. So, what do these commands do? Well,

- `git fetch`: brings in any changes from the global git repository, updating pointers to the global branches (*i.e.*, `origin/master` and `origin/HEAD`)
- `git rebase origin/master` takes your local `master` branch pointer and places all local commits (those that have not yet been pushed to the global git repo) above the pointers to the pointers to the global branches (*i.e.*, `origin/master` and `origin/HEAD`). In other words it "changes the base" of the current branch (*i.e.,* `master`) to be on to of the `origin/master` branch.

With this procedure, *every* developer is responsible for ensuring that their commits fit on top of the development history. This creates a *clean* tree (a straight line) without unneccessary merge commits. It also forces the developer to resolve conflicts him/herself and does not rely on the handwaving that can be implicit in `git merge`. Doing this on a regular basis (before you start working for the day, or even as often as every email that notifies you that the repository has been updated) can save you a lot of pain and help you avoid conflicts.

NB: NEVER USE `git pull`; it's a high-level command (combining `fetch` and `merge`) that does some fancy stuff. Though it'll usually work, when it doesn't it becomes unneccessarily confusing / problematic / annoying / frustrating to fix it because it takes time / git know-how to find out why. It also poses a problem when git-novices use it by creating unneccessary merge commits, making the git tree itself confusing, making it more difficult to use `git bisect`, and making gitk and other git GUIs more difficult to use as well. Avoid all updating problems by simply following the procedure outlined above.☺

4.1.3 Helper Commands

To change your current working branch:

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

Use this to switch to the `master` branch if you are not currently on it.

[3]More about this in handout two. For now, don't worry about it

4.2 Committing

After having edited some files, you are ready to create a commit. A commit is a logical change; it can be as short as one character or as long as you want. The point is that if it were ever to be undone (`reverted`) only that logical change should be undone. If there are more than one type of change within a commit, reverting it would undo things that you may want to keep in the repository. More than that, it helps everyone working on the project better understand exactly what was changed without being obliged to look at the code.

```
> git add <file to be committed>
> git commit -m "descriptive message"
```

- `git add` adds an edited file to the staging area, preparing it to be commited. NB: if you modify the file after doing `git add`, you will need to `add` it again in order for those changes to be included in the commit
- `git commit` creates the logical commit on your current branch (`master` for now). `master` moves forward by one node, leaving `origin/master` behind. Your local repository is now ahead of the global repository. NB: A commit is denoted by an SHA.

After a commit, if you realize that you have forgotten to include a file that logically belongs there, don't worry! Just,

```
> git add <file to be committed>
> git commit --amend
```

to include it in the commit.

4.2.1 Helper Commands

To create a commit that undoes a previous commit (denoted by the SHA):

```
> git revert <<SHA>>
```

To delete all changes and move your current branch pointer to the commit denoted by SHA:

```
> git reset --hard <<SHA>>
```

NB: this is DANGEROUS! Use it with care not out of frustration!! Unfortunately, I speak from experience.

4.2.2 .gitignore

This is a file that provides git with a list of files to ignore. Basically, git knows everything that's changed within your local repository. It differentiates between files in different states:

- **Staged**: files that are ready to be committed (placed here using `git add`)
- **Unstaged**: files that are currently tracked by git (*i.e.*, have already been committed), that have been edited but NOT yet placed in the staging area
- **Untracked**: files in the repository that have never been committed and that are not ignored by `.gitignore`

Sample `.gitignore`

```
*~
*.o
...
```

4.2.2.1 Helper Commands To check which of the above three categories the edited/non-.gitignored files are:

```
> git status
```

To put the tracked and edited files in a sandbox (ie WIP commit):

```
> git stash
```

4.3 Pushing

Well, well, well. Now you've updated your repository, committed some changes and are ready to share those changes with everyone else. Before you do so, update your repository again! Do this just to make sure that in the period between when you last updated your repository and when you want to push your changes, someone else hasn't introduced changes to the repository. Now, after doing another round of `git fetch` and `git rebase origin/master`, do:

```
> git push
```

and your changes will be sent to the global repository, updating the `origin/HEAD` and `origin/master` pointers.☺

5 More on Branches

As mentioned above, the use (and idea of) branches can quickly become complicated. Hopefully, here, it will become a bit more understandable. In my work, as stated earlier, I have encountered two primary uses for branches: 1) software versioning, and 2) experimentation / sideline development. Let's look at the use of both of these independently.

5.1 Software Versioning

For every major Dynare release (4.0, 4.1, 4.2, etc.) we fork the master branch. The main development branch (which keeps the name `master`) always contains the unstable/development version of Dynare. On the other hand, the release branch contains a branch wherein new features will no longer be introduced. In a sense, on that branch, we freeze production and the only commits that will be added to it are either bugfixes or build system changes to prepare it for release. This branch, logically, takes a name equivalent to the relase version of the software. So, creating such a release branch would look like:

```
> git branch 4.3
```

This command would create a branch with the tag `4.3`. Once this command is issued, `master, origin/master, origin/HEAD,` and `4.3` point to the same commit. Now, we may want to change the build/packaging system to prepare it for public release. To do this, we would change over to the 4.3 branch, edit the appropriate files, commit them and push the new commits (and the new branch) to the main repository:

```
git checkout 4.3
> git add <<file>>
> git commit -m "message"
> git push origin 4.3
```

Now, `master, origin/master` and `origin/HEAD` point to the same commit they did before. The `4.3` branch has now been pushed to the server and thus, `4.3` and the new `origin/4.3` point to the same commit containing the build system edits.

As with all software, Dynare has bugs. And, after it's released to the public, those bugs are found quickly and we do our best to fix them. When we fix a bug, we fix it on the `master` branch. However, given that from time to time we provide bugfix releases of our stable versions (e.g. 4.3.1, 4.3.2, etc), we want to have those bug fixes on the stable release branch as well. In order to accomplish this, we do:

```
> git checkout master
> git fetch
> git rebase origin/master
> git add <<file>>
> git commit -m "bugfix"
> git checkout 4.3
> git rebase origin/4.3
> git cherry-pick -x master
> git push
```

Here, the cherry-pick accomplishes the desired task by applying the same changes to the 4.3 branch as it did on the master branch. And, now, git push will push both master and 4.3 to the repository (since we have previously pushed 4.3 to the repo).

Finally, when we want to make a bug fix public release of Dynare, we:

```
> git checkout 4.3
> git rebase fetch
> git rebase origin/4.3
> git tag 4.3.1
```

Here, we checkout the 4.3 branch, update it to make sure we have all the bugfixes that everyone may have cherry-picked onto it and then tag it with the name `4.3.1`. In this way, we always know where in the repository we have the different versions of the software that we released.

5.2 Experimentiation / Sideline Development

With software versioning, there is no intention of merging the branch back into the development branch; essentially, development on that branch dies and it remains as a record of what the software was at different points in time. On the other hand, you may need to work on a side project that may or may not eventually be included on the main development branch. In order to do this, you would follow the same process as above for creating a branch and for switching to it:

```
> git branch myProject
> git checkout myProject
```

Now, again, `myProject` becomes the active branch. You can push it to the sever (if you want others to have access to your development work) and commit to it in the same way as above.

Now, when your project is complete and if you decide to include it in the main development branch, you will do the following:

```
> git checkout master
> git fetch
> git rebase origin/master
> git merge myProject
> git push
```

This will merge the changes you made on the `myProject` branch into `origin/master`. Of course, you can encounter conflicts doing this and we can talk about this next time.