Git Well Soon: a crash course in version control

September 15, 2017

> If you're not familiar with using a terminal to navigate, all you need for this session are the following commands: `$cd [folder]` will move you into [folder]; `$cd ..` will move you up to the parent folder; `$ls` will show files in the current folder; and `$mkdir [name]` will create a new directory called [name].

Ciaran has a set of SAR tools he is maintaining using a Github source code repository; [LINK] Here's how you can get that code for your own use, make any changes you feel are needed and share it with anyone else who would require it.

What is Git?

Git is a way of managing and keeping track of a piece of software's history – who made what change when. It is also a way combining code from multiple developers together into a single piece of software, while still keeping track of who did what.

Installing Git

If you're on Windows, then you can install Git for Windows via the University's Program Installer. If you're on an Ubuntu or other Linux machine, then Git is usually pre-installed. If not, then you can install it with

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

on the command line.

> From here out, it's assuming that you are using a standard Bash terminal for navigation. If you are on Windows, a good enough emulation comes with Git for Windows; just type 'Git Bash' into the search bar.

Getting the code

First, navigate to a convenient directory using cd and mkdir, as appropriate. Once there:

```
$git clone [LINK]
```

This is now the **root** of your local repository – as far as Git is concerned, everything is relative to this point.

What is happening here

This will copy all of Ciaran's code from Github to your machine into a **local repository**, complete with the history and branches [see later] of his project right to the beginning – you can view this with the command `$ git log`. Press 'q' to exit the log viewer.

Why do this?

As well as getting the entire project's codebase, when you cloned the repository with `$ git clone`, you also cloned its history and configuration. This means that you can:

- Keep it up to date with any revisions Ciaran might make in the future
- Review any changes he might have made in the past, and understand why he made those changes.
- If you make a huge, unfixable mistake, or if the code was working last Tuesday and isn't now, you can roll the entire project back to a point where everything was working
- Fix any bugs or glitches you might find
- Add any further features that you think the code might need, and share those features with anyone else.

Keeping the code up-to-date

Whenever Ciaran or anyone else makes a change or adds a new feature, you can easily update his code on your own machine with this command:

```
$git pull
```

You should now see a short summary of what has been changed; you can get more information using the command `$ git diff`, or see the later section on using graphical tools.

IF YOU DO NOT WISH TO EDIT ANY CODE, YOU CAN STOP HERE.

Editing code of your own

Before you go any further in this tutorial, you will need to go to www.github.com and register an account.

Let's say you wanted to add a function to Ciaran's codebase. You've written the script – let's call it [newthing] – in a file of its own (`newthing.py`), and now you want to add it to Ciaran's repository so everyone else can use it. This process has a few steps.

1. Put newthing.py in a sensible place in your local repository using a file manager. You can check it's there with `$git status` As you might gather from the status message, Git does not know that you want to add newthing.py to the repository.
2. Use the command

```
$git add FILEPATH
```

where `FILEPATH` is the path from the root of your local repository. If in doubt, use the message that git status gives you.

This adds newthing.py to your staging area. This is where any changes you make are stored until you commit them to your local repository in the next step.

3. Commit the changes to your local repository with the command

```
$git commit --m "some informative message here"
```

The message you write in here is what will appear when anyone runs `$ git log`, so make sure it's informative and clean(ish)

> If you forget the –m option, by default Git will open vim; a powerful but non-intuitive text editor. To exit Vim, press 'esc' then type ':q!' without the quotes. If you want to *learn* Vim, clear an hour in your schedule and run `vimtutor` from the command line.

> When to commit: Early and often is the mantra from the community; the smaller each commit is, the more powerful the 'time-travel' tools become. Find a commit pace that works for you, but if in doubt more often is usually better.

This commits snapshots of your code to your local repository;

4. Continue to edit and `$ git commit` your code, using [add] to add new files as required. Bear in mind none of these changes will go to Ciaran's repository yet.

5. Once you are happy with your new feature and are reasonably sure that it doesn't break anything else, use the command

```
$git push
```

to put your changes into Cieron's repository. You might be prompted for your Github username and password here.

> When you're doing a large commit (such as adding a new feature to a repo), it's good to write a longer commit message using a text editor – the default is Vim, but you can change this using the command git config –global core.editor [editor]. For a quick guide to writing useful commit messages, see `https://chris.beams.io/posts/git-commit/`.

Branching

It is a sad fact that when working on improving a piece of code, you will almost certainly break it or something else instead. To avoid this and the associated recriminations, Git allows you to make local timelines of the project that you can mess around with as you please, without affecting anyone else's work. These local timelines are called 'branches'.

By convention, each Git repository starts with a branch called 'master'. Once the initial code repository is written, this is usually the 'this code works' branch – anyone wanting to use Cieran's tools will use the code stored in 'master'.

It is good practice whenever you are writing a new feature to create a branch to develop it in. This way, others can still use the original version of the program and also contribute to your improved version, and you can test your new features without having to worry about breaking the software for everyone.

To create a new branch, use the command

```
$git branch [branchname]
```

to create the branch, and the command

```
$git checkout [branchname]
```

to start work in the new branch. As long as you have commited your work, you can move between branches with the `$git checkout` command and see the difference between them with `$git diff [branch1] [branch2]`.

This branch is in your **local repository**. If you `$ git push` while you're working in a branch, Git will add that branch to the **remote repository** for others to access.

Merging

Merging is when you take the changes made in one branch and apply them to another branch. Git is very smart about this, and changes are done more or less line-by-line - even if two people are working on different parts of the same file in different branches, Git can collate those changes without user input (but see the next section). There are two situations where you need to merge regularly;

- When you have finished work on a branch: You will want to merge your shiny new feature into the 'master' branch, so everyone can use it. To do this, first make sure you have `$ git commit` -ed your changes to your branch. Then, move to the 'master' branch using

  ```
  $git checkout master
  ```

  Once there, use the command

  ```
  $ git merge [branchname]
  ```

  This will bring the changes out have made in [branchname] into the 'master' branch; you can then push that branch to the remote repository using `$ git push`.

- Someone else has made a change to 'master' while you are working on your branch: you will want to make sure that this new change does not affect your work. Therefore, you will want to merge the changes made in 'master' with your own branch. To do this, first you will need to pull the changes from the remote repository to your local repository. Make sure you are in [branchname] using `git status`, then use:

  ```
  $git fetch origin master
  ```

  to get the change. This actually creates a new branch, 'origin/master' in your local repository that contains said changes. Then use

```
git merge origin/master
```

to bring the changes into [branchname].

Conflicts

It can happen that two people working on the same repository both change the same part of the same file in different branches. When this happens, Git does not know how to combine the two files - this is a **merge conflict**, and it will cause the merge to fail. Git can't solve this on its own; you will have to talk it over and decide which revision to keep, how to reconcile the conflict, ect. Them make the edits and commit as normal.

Rolling back

The final basic trick that Git provides is rolling back. Think of this a time-machine for your project; you can jump back to any point in the project's history that was `$git commit` -ed. As a rule, any piece of work that was committed is recoverable *somehow*; it is quite difficult to permanently lose work with Git. If you have lost some work that you've committed, talk to the repo administrator – they can probably get it back for you.

You might have noticed so far that each commit has a long string of meaningless numbers and letters associated with it; this is that commit's **hash**. Any time you need to refer to a given commit, you can use **the first four characters** of it's hash.

Rolling individual files back to the last checkout

If you've made some changes to a file since your last commit and want to restore it to its original state

```
$git checkout -- [filename]
```

will overwrite the current version of [filename] with the last version you committed. Be careful; **this will lose your changes for good**.

Looking backwards in time

If you wish to see what the project looked like at a given commit point,

```
$git checkout [hash of commit]
```

will move your project back to the state it was in at that point in time. If you want to make changes from that point, you will need to start a new branch. Instead of the above, use the command

```
$git checkout -b [new branch name] [hash of commit]
```

This will start a new branch from that point that you can checkout and merge as required. You can return to the present with

```
$git checkout [branch you were working on]
```

Undoing large changes If you wish to undo a specific commit;

```
$git revert [hash of commit]
```

This will create a new commit that removes the effects of the old commit – this has the same effect as you going over each of the changes from the old commit one-by-one and putting them back the way they were, but with far less effort. NOTE: This does not work back through the history; it only undoes the changes made by **that specific commit**. If you want to know more, Git has a more sophisticated way of defining ranges of commits: `seehttps://git-scm.com/book/en/v2/Git-Tools-Revision-Selection`.

GUI-based Git

If you prefer something you can click on, rin `$ gitk` on the command line. This will open up a history viewer for the repo you are presently in. There is also a full GUI for controlling Git included in the Windows distribution; most of the commands here you can do via that, if you prefer.

Acknowledgements and further reading

Most of the content of this guide came from Pro Git, by Scott Chacon and Ben Straub. The whole book is available for free here:`https://git-scm.com/book/en/v2`. I recommend reading through chapters 2 and 3 (especially 3.5), and sections 7.1, 7.2 and 7.3. Remember, the arrows in the diagrams point backwards in time.

You can also work through an online introduction to Git at `https://try.github.io`.

As mentioned above, I also recommend you read `https://chris.beams.io/posts/git-commit/` for a set of instructions about writing informative and useful commit messages.

Some last words to remember

- Always `$ git commit` before you `$git pull`
- Always `$ git pull` before you `$git push`
- Never use any command with –force unless you're *absolutely sure* you know what you are doing. Even if the Internet says otherwise.

Quick reference table

When you want to...	**Use the commands...**
Add a new code repository	`$git clone [repo URL]`
Get any changes made to that code	`$git pull`
View those changes	`$git log`
Check what files you've changed since your last commit	`$git log`

Add a new file to your local repo	`$git add [files]`
	`$git status`
	`$git commit -m 'Some informative message'`

Branch the repo to start work on a new feature	`$git branch [branchname]`
	
	`$git checkout [branchname]`

Merge changes from another branch	`$git merge [otherbranch]`
Move a file back to it's last commit (LOSING ANY CHANGES YOU'VE MADE SINCE THEN)	`$git checkout -- [filename]`
Rewind the entire project back to a certain revision	`$git checkout [hash]`
Create a new commit that undoes any changes made in commit[hash]	`$git revert [hash]`