Git Well Soon: a crash course in version control

Ciaran has a set of SAR tools he is maintaining using Github; [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.

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 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.

[CALLOUT: 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’.

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, 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 fetch origin

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

What is happening here

TODO more here

[CALLOUT: When you first cloned Ciaran’s repository, your local copy of Git stored a reference to it under the name ‘origin’. The repository is set up such that the default branch is ‘master’ - more on branches in the next section.]

Editing code of your own

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.

1. 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. Once a file is added to your staging area, it will stay there until deleted or explicitly removed; you do not need to keep [add]ing it.

1. 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)

[CALLOUT: If you run ]

[CALLOUT; WHEN TO COMMIT]

This commits snapshots of your code to your local repository;

1. Continue to edit and [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.
2. Once you are happy with your new feature, use the command

$git push origin master

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

[CALLOUT] 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 the ‘working’ 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 commands

$git branch [branchname]

to create the branch, and

$git checkout [branchname]

to start make it your active branch. Now, just continue working, staging and committing as above – any edits you make will go into [branchname] You can check what branch you’re currently working on with $git status.

This branch is in your **local repository.** If you push (as above) while you’re working in a branch, Git will add that branch to the **remote repository** for others to access. [note on

Merging

Merging is when you take the changes made in one branch and apply them to another branch. There are two situations where you need to do this day-to-day.

* 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 committed your changes to your branch. Then, move to the ‘master’ branch using

$git checkout master

Then, to do the merge, use

$git merge [branchname]

This will bring the changes you have made into the ‘master’ branch; you can then push that branch to the remote repository using [$git push origin master].

* 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 using

$git fetch origin

To do this, make sure you are in [branchname] using $git status, then use

$git fetch origin

To get the changes as before. This actually creates a new branch, ‘origin/master’ in your local repository that contains said changes.

You can then use

$git merge origin/master

to bring the changes into [branchname]

Conflicts

It might happen that your new feature overwrites an addition that Cieran made to his own branch; say, you both changed the same line of code in a file. 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

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 [$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 it happens to 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 commits **hash**. Any time you need to refer to a given commit, Git will recognise it using **at least the first four characters**.

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 the project was in at that point.

If you want to make changes from this 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 combine the two above commands to pull;

$git checkout

Undoing large changes

If you wish to undo a specific commit;

$git revert [hash of old 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: see <https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection>.

-Quick reference at end