Introduction to git

Lorne Whiteway
lorne.whiteway@star.ucl.ac.uk

Astrophysics Group
Department of Physics and Astronomy
University College London

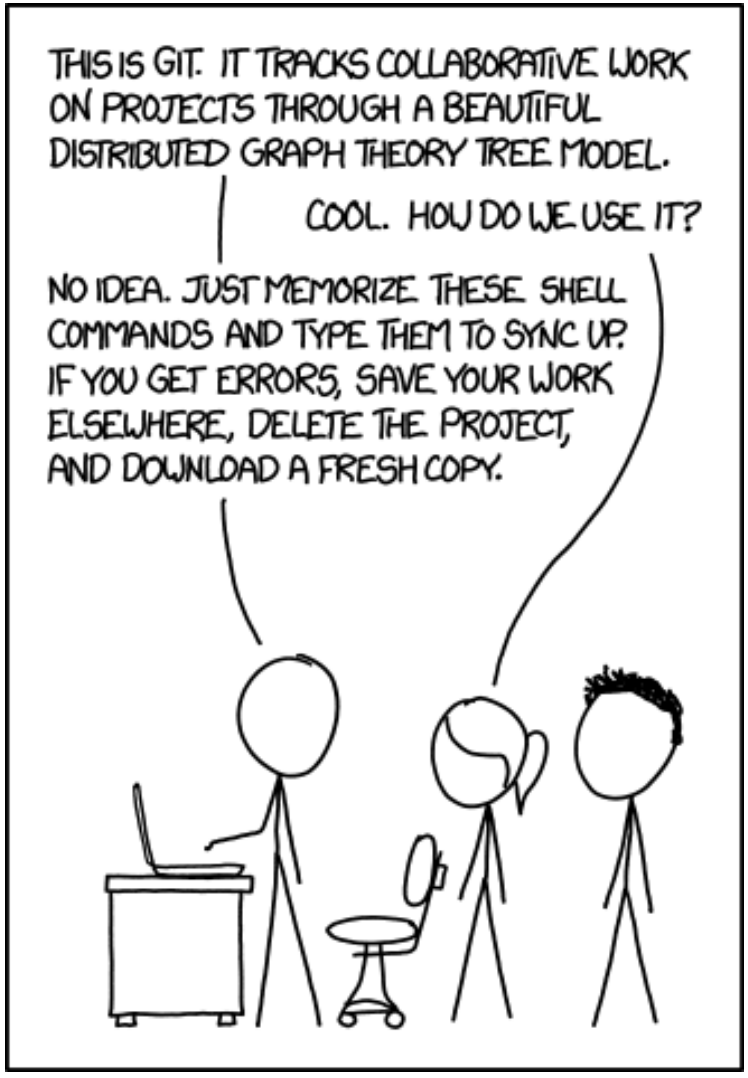
17 November 2017

Where to find this presentation

Find the presentation at https://tinyurl.com/y8pr4mvq.

On this page click on 'Download' to get a copy of the presentation.

https://xkcd.com/1597/ Creative Commons Attribution-NonCommercial 2.5 License.



Some commentary...

https://stevebennett.me/2012/02/24/10-things-i-hate-about-git/

Purpose of presentation

- I don't want to teach you how to use git.
- Rather I want to illustrate (part of) git's 'internal model' and to define certain key git terminology so that you will be better prepared to teach yourself git.
- My examples assume you are calling git from the command-line. Friendlier interfaces to git exist - but you still need to know the underlying model to use them effectively.

Why is git (relatively) hard to come to terms with?

- The internal model is complicated.
- The interface is inconsistent.
- The documentation is suboptimal.
- Several key ideas have been given misleading names.
- It uses a ‘distributed’ model whereas what you usually want is a ‘client/server’ model. So you tend to be ‘fighting against the paradigm’...

Source control

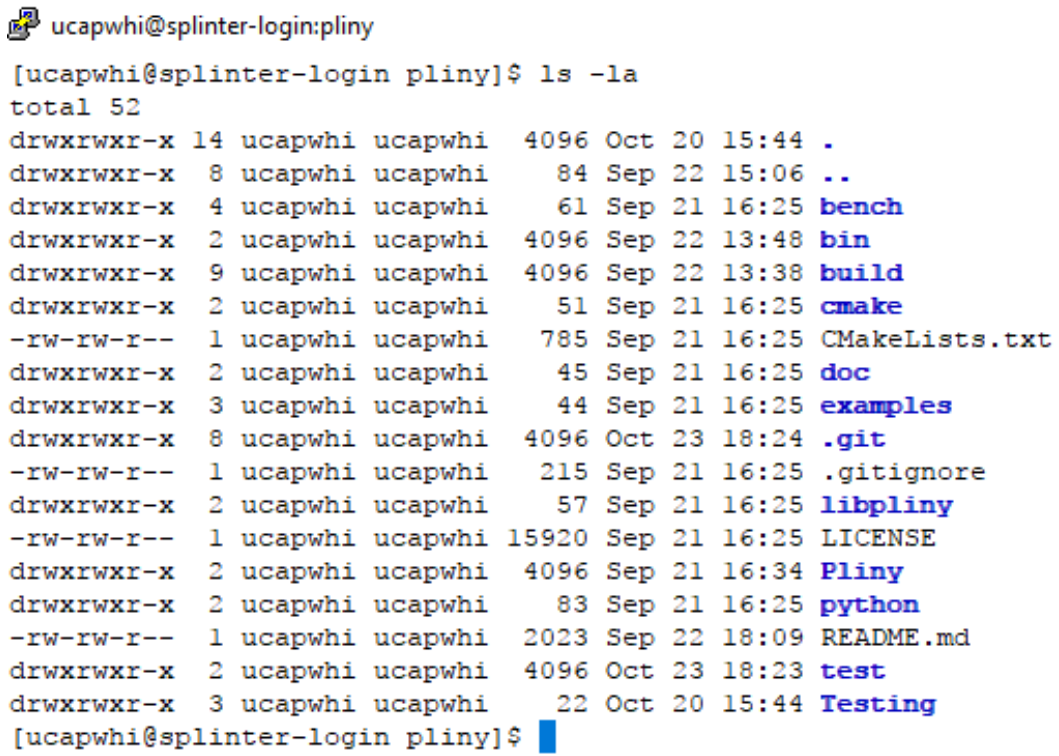
- Source control is software to 'keep track of' (i.e. store) successive versions as we edit a collection of *source* files (computer code, LaTeX documents, etc.)
- Works best if the source is text, not binary. Intermediate files are usually not kept track of. Output files might be - your choice.
- Any serious project should be under source control.

Working directory and repository

- You need a *working directory* and a *repository*.
- The working directory and its subdirectories contain the actual files that you are editing.
- The repository is some sort of database containing all previous versions.
- One model would be to put the repository on the Internet or Intranet where everyone can see it...

Location of git repositiory

- ... But in git the repository is **next to** the working directory, in a hidden subdirectory (called .git) of the top-level working directory.

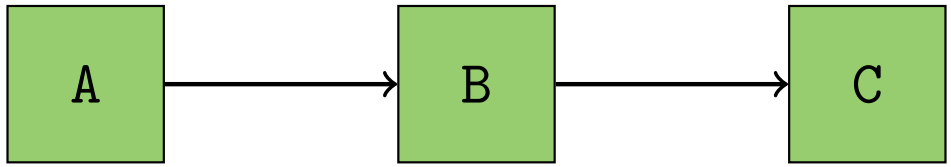


This has consequences...

- You therefore need to have a git repository next to your working directory on your local directory (where you are doing the actual editing).
- But typically you will also need one on the internet (for backup and for collaboration and sharing).
- So you will typically be dealing with **two** git repositories (and dealing with the issues of keeping them in synch.

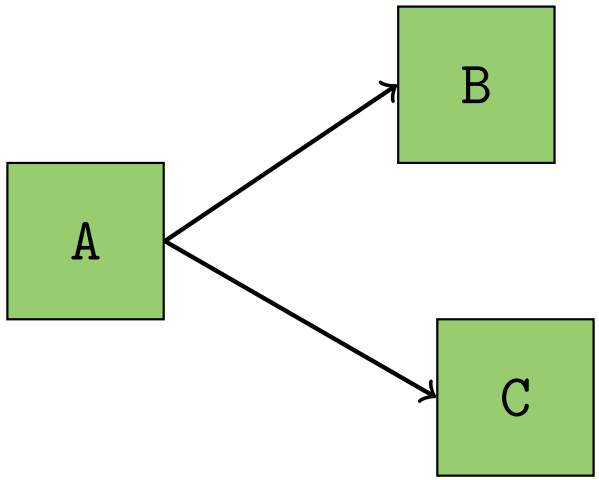
- The upside is that you can still do version control even if you are not connected to the Internet.

Example repository content



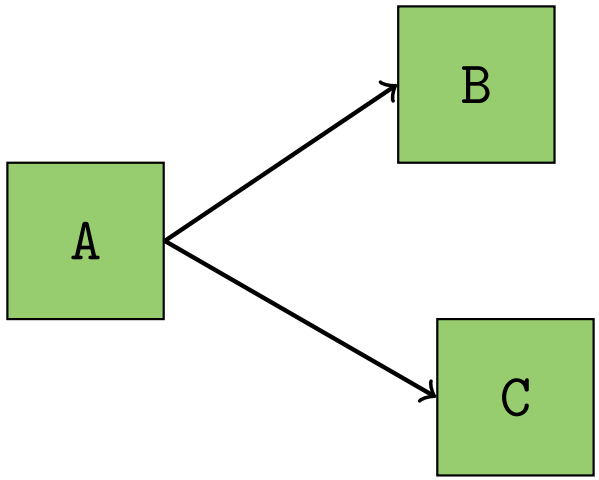
- This repository contains three successive versions of the files in the working directory (and its subdirectories).
- Each version is represented here as a *node* (in green).
- An initial set of files (version A) was comitted to the repository; the files were then edited and the new file set (version B) was comitted; the files were then edited and comitted a third time (version C).

Another example



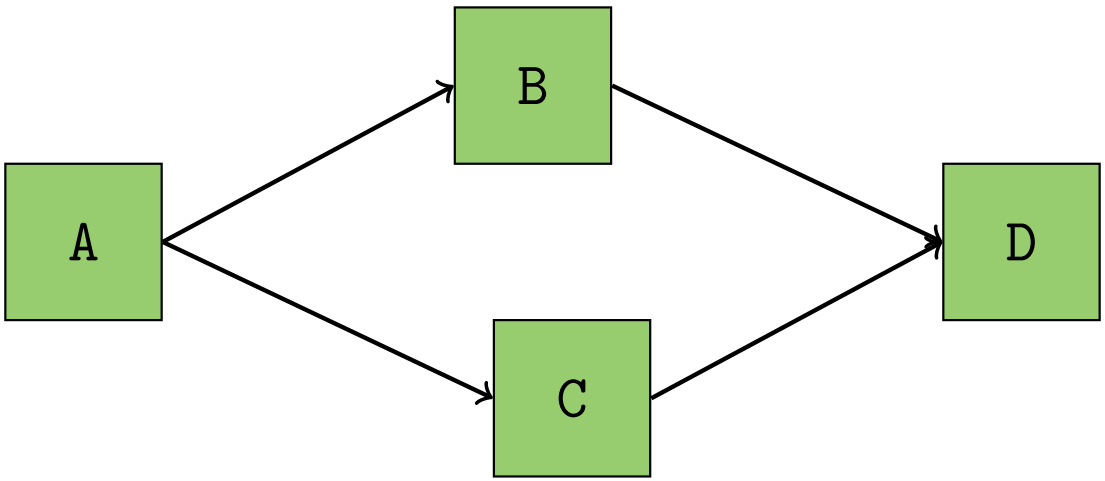
- Here we committed A...
- Then we edited A (to form B) and committed B...
- Then we went back to A, made a perhaps different set of edits (to form C) and committed C.

What do the arrows actually stand for?



- An arrow respects time (pointing from an earlier version to a later version), and indicates that a node was derived from an earlier node by editing.
- Q: There exists a set of edits that would take you from B to C, so why not show that arrow as well? A: It's an *itinerary* (showing the route we took), not a *map* of all possible routes.

Merging



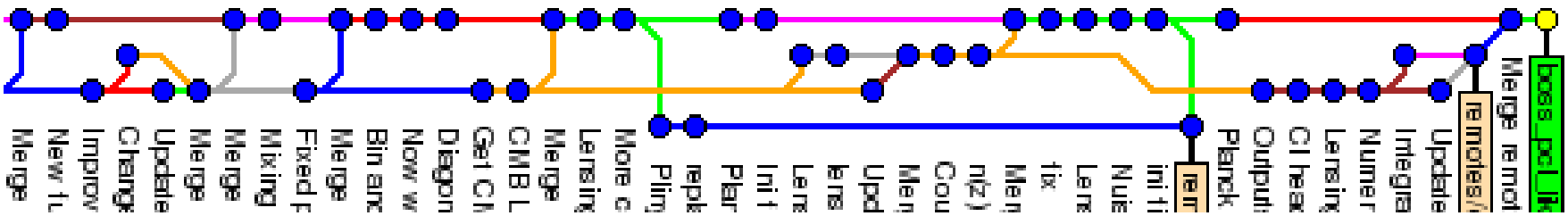
- Here we combined the ‘A to B’ edits and the ‘A to C’ edits (to form D), which was committed.
- More on such *merging* later.

DAG

- Hence we get a graph (in the mathematical sense of nodes plus edges).
- The graph is not a *tree* (because of merging).
- It is *directed* (edges have arrows) and *acyclic* (cycles would break causality), so we have a *DAG* (directed acyclic graph).
- Nodes have *parents* and *children*, and hence *ancestors* and *descendents*.
- One node - the initial *root* node - has no parents. All other nodes have one or two parents. Thus the graph is connected, and any two nodes have ancestors in common.
- *Graph theory* is an interesting part of mathematics - but alas not useful here.

Seeing the DAG

- Run `gitk --all` to see the DAG (lots of other information as well).



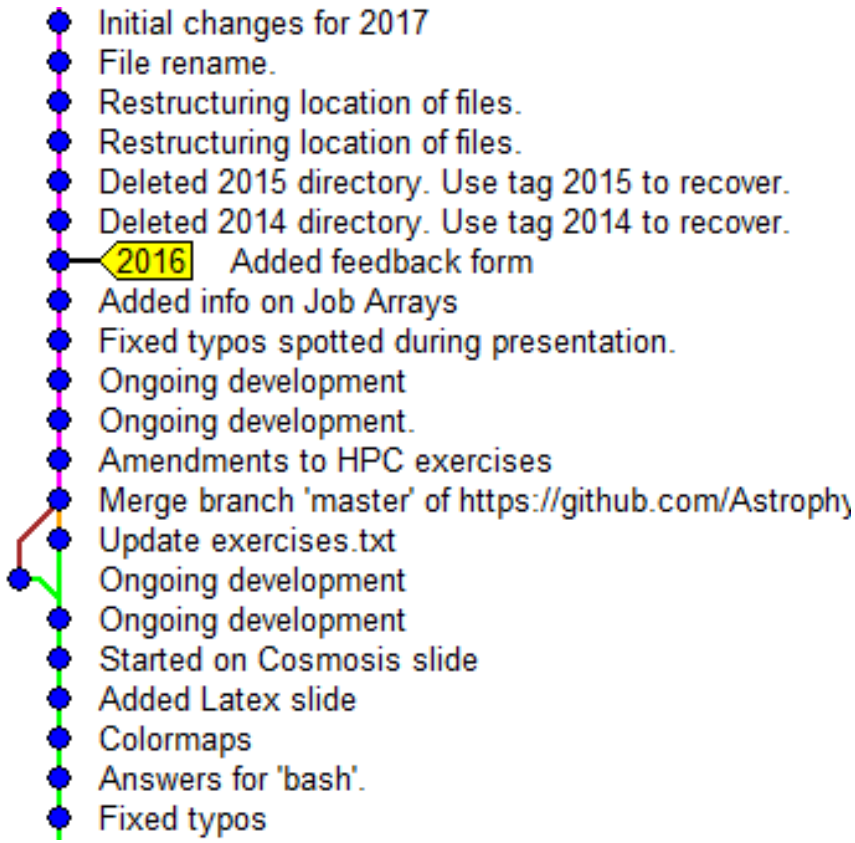
What's in a node?

At least:

- Pointers to enough information to reconstruct the working directory as of that instant.
- Administrative information: who made the commit, a 'commit message', etc.
- A permanent node name (SHA1 format - 40 hexa digits e.g. c2d2ea34cec13a0956488f2b919861fccad8a448). You can abbreviate this to an initial substring (provided that it is long enough to be unique).

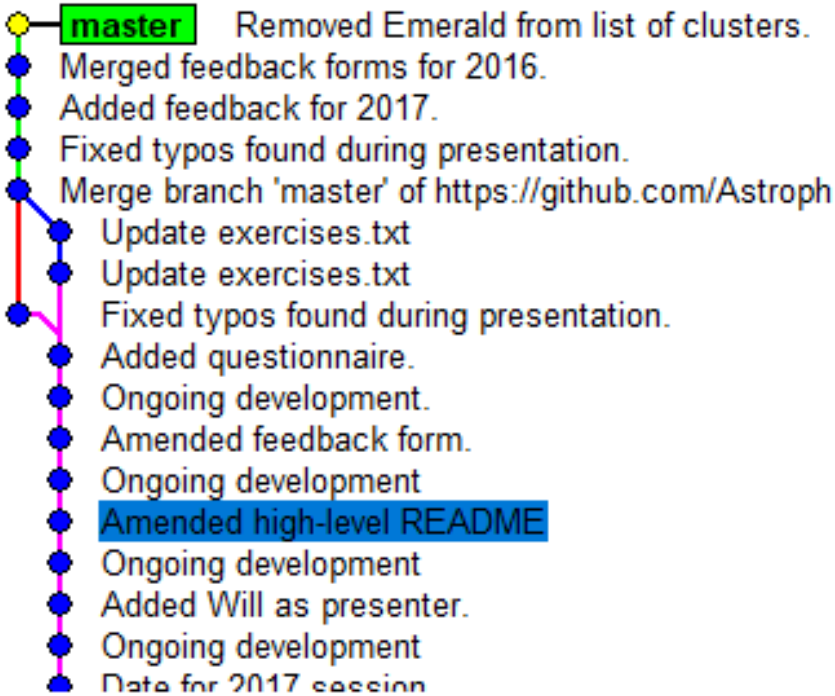
Pointers to nodes - tags

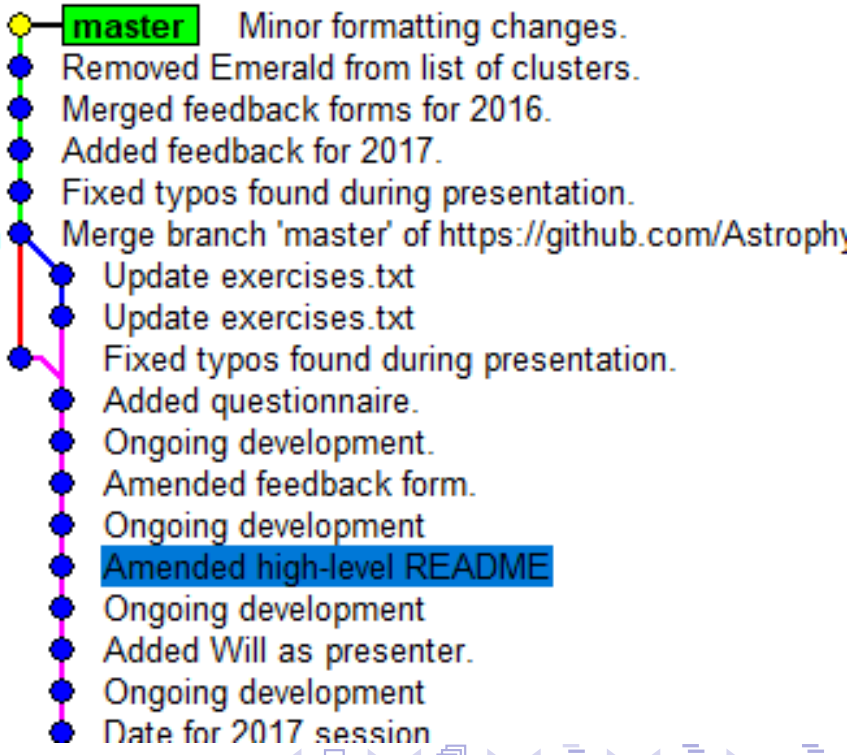
- A *tag* is a label that is attached to a node.

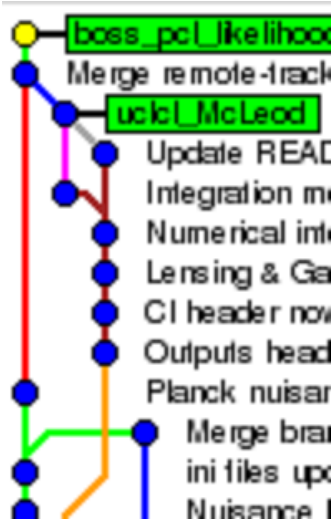


Pointers to nodes - branches (1)

- A *branch* is a moveable label that is attached to a node.
- Here ‘moveable’ means that if you commit an edit to that node then the branch label moves to the child node.





Pointers to nodes - branches (2)

- A branch can point to any node in the DAG - not just to a terminal node (i.e. node with no descendents).
- So 'branch' is not a perfect metaphor; better is 'pointer to a place where growth may occur' (which may be on the trunk.