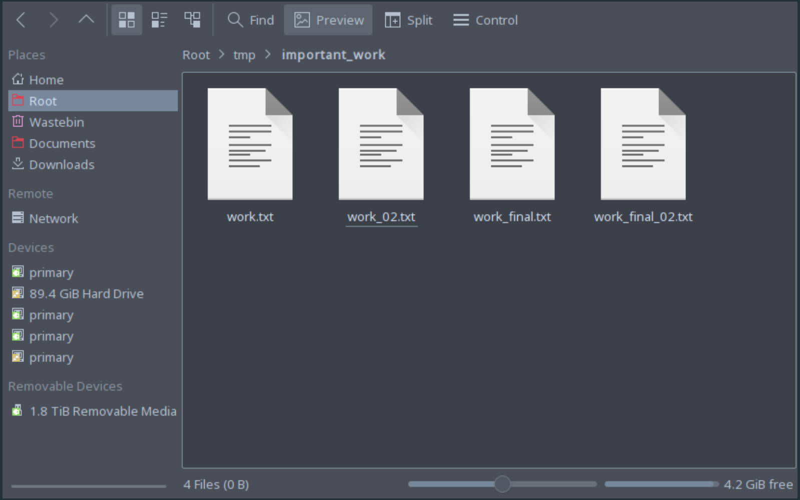
Introduction to Version Control

Peter Hill

Outline

- What and why version control
- Basics of git
- Using a web service

What is version control?



Is_this_version_control_meme.jpg

What is version control?

- Version control systems record changes to a file/set of files over time
    - Not just software! This talk is under version control
    - Allows you revert files back to a previous state, compare changes over time, see who last modified something, etc.
- Instead of keeping multiple copies of the same file, normally just store the *differences* (“diffs”) between versions of the files

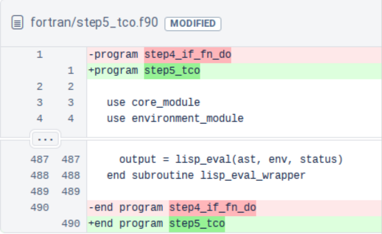
Why is version control important?

- Tracking versions
  - Know instantly which is the latest version
  - Roll back to previous versions
  - See history of project/file/line
  - Find out when bugs were introduced
  - Maintain/compare different versions
- Coordination between developers
  - Easier to keep track of when changes are made
  - Easier to work on separate features
  - Easier to merge distinct changes from separate developers
  - Easier to resolve conflicts on same features
  - Tracking who made what changes

If it's not under version control, it doesn't exist!

The gist of git

The building blocks: diffs



fortran/step5_tco.f90 MODIFIED

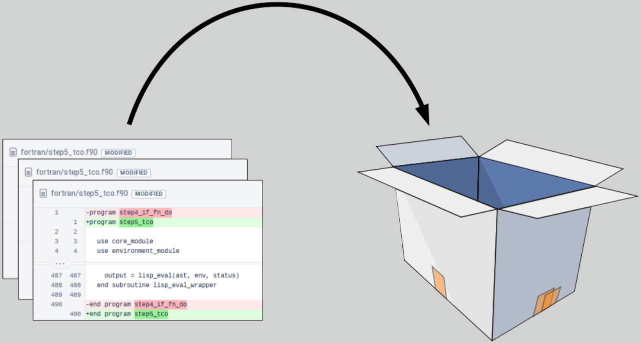
```
-program step4_if_fn_do
+program step5_tco

   use core_module
   use environment_module
...
     output = lisp_eval(ast, env, status)
   end subroutine lisp_eval_wrapper

-end program step4_if_fn_do
+end program step5_tco
```

A simple diff

- Shows differences between individual lines
- Lines beginning with "–" have been removed
- Lines beginning with "+" have been added
- Changed lines are shown as removal plus addition

The gist of git

Add diffs to a staging area



fortran/step5_tco.f90 MODIFIED

fortran/step5_tco.f90 MODIFIED

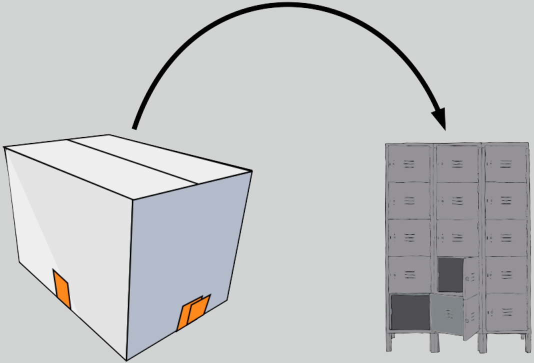
fortran/step5_tco.f90 MODIFIED

```
-program step4_if_fn_do
+program step5_tco

   use core_module
   use environment_module
...
     output = lisp_eval(ast, env, status)
   end subroutine lisp_eval_wrapper

-end program step4_if_fn_do
+end program step5_tco
```

Add diffs to stage

The gist of git

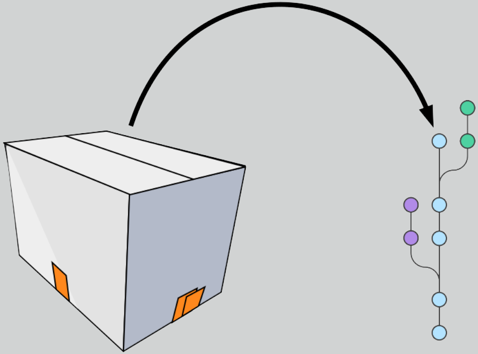
Commit the staging area to the repository



Commit changes to the repository

The gist of git

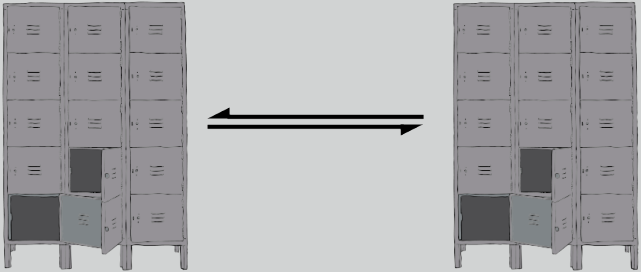
Commit the staging area to the repository



Commit changes to the repository

This gist of git

Sync with other people's repositories

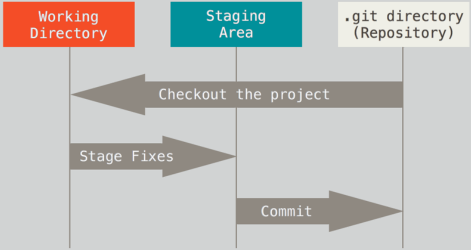


Sync local and remote repositories

Using git

The Three Stages

- Important to understand correctly
- Three main states that files can be in:
  1. Committed: data stored in repo
  2. Modified: file is changed but not committed
  3. Staged: modified file marked to go into next commit



git three stages

Using git

Graphical interfaces

For Mac and Windows:

- Sourcetree: https://www.sourcetreeapp.com/

For Linux:

- git-cola: http://git-cola.github.io/

For Emacs:

- magit: https://magit.vc/

Today’s toolkit

What you’ll need

- Linux
- A terminal
- A text editor
- Optionally: a file browser

First steps

Making a repository

- We need to create a repository first

```
$ mkdir my_git_test
$ git init
```

- Let's see what it looks like
- What do you see after typing each of the following commands?

```
$ ls
$ ls -A
$ git status
```

git commands

Getting help

```
$ git <command> --help
```

git status

- `git status`: Show the working tree status
- **Glossary** “Working tree”: what the repository directory looks like, including any changes
- Going to be our most used command today!
- Use it whenever you’re not sure what’s going on

First steps

Initial commit

- Add some text to a new file and save it in your repository
- What does `git status` show now?
- Follow the instructions to add your new file to the staging area
- Check `git status` again

```
$ echo "Some text" > newfile.txt
$ git status
$ git add newfile.txt
$ git status
```

First steps

Initial commit

- Now we need to actually commit our commit

```
$ git commit
```

- Your default editor should pop up
  - If you hate it, change your `EDITOR` variable
- The traditional first commit message is “Initial commit”
- Now check `git status` again

git commands

git add

- `git add <file>`: Add file to the index
- **Glossary** “Index”: the stored form of the working tree, i.e. the staging area, our “box”

git commit

- `git commit`: Record changes to the repository
- Until you run `git commit`, the changes made to the staging area (index) remain separate from the working tree and repository

Writing commit messages

- Writing good commit messages is a skill!
- Commit messages serve as documentation for your project
- Finish the sentence: “This commit will. . . ”

Good

- Fix bug in boundary conditions
- Add new routine for calculating potential

Bad

- update code
- I fixed some stuff

First steps

Making our first change

- Change the text in your file however you like
- Time for `git status`
- Ok, but how to actually see the changes?

```
$ git diff               # All files
$ git diff <file>...     # Just certain files
```

- Press q to quit the “pager”

Commit the change

- It’s a two-step step: `add` then `commit`

```
$ git add <file>
$ git commit
```

git commands

git diff

- `git diff`: Show changes between commits, commit and working tree, etc
- Without arguments, or with just files, shows differences between the working tree and the staging area
- Use `git diff --staged` to see the difference between staging area and latest commit (i.e. what's going into the *next* commit)

The basics

Updating the staging area

- Make a change and `git add` it
- Now make another change to the same bit of that file
- Check `git status`, what do you see?
- Try `git diff`, then `git diff --staged`, what's the difference?
- `git add` your second change, and try the two `diff` commands again

The basics

Updating the staging area

```
# Edit file
$ git add <file>
# Edit file again
$ git status
$ git diff
$ git diff --staged
$ git add <file>
$ git status
$ git diff
$ git diff --staged
```

The basics

Looking back

- Make some more changes and make two or three more commits
- View the history so far:

  ```
  $ git log
  ```

- View a particular commit in more detail:

  ```
  $ git show <hash>
  ```

- **Glossary** “commit hash”: a 40-digit hexadecimal “hash” that uniquely identifies a commit. Generally only ~7 digits are needed. The largest projects need upto 12

git commands

git log

- `git log`: Show commit logs
- There are lots of options here!
- Lots of ways to format the log, or search for particular commits

git show

- `git show`: Show various types of objects
- Also lots of options for formatting the output, etc.

The basics

Undoing unwanted changes

- Delete your file!
- Check `git status` – can you see how to get it back (i.e. discard the change)?

Undoing changes to the index

- Make a change to a file and now add but **don't** commit
- The usual – `git status` – how do you remove something from the staging area?

The basics

Undoing things

```
# Make a change to a file
$ git checkout -- <file>
# Make another change
$ git add <file>
$ git reset HEAD <file>
```

git commands

git checkout

- `git checkout [options]`: Switch branches or restore working tree files
- **Glossary** “checkout”: modify the working tree *en masse*
- Remove unwanted changes to a file:

  ```
  $ git checkout -- <file>
  ```

- Note: one of the few things in git that can’t be undone!

git commands

git reset

- `git reset`: Reset current HEAD to the specified state
- **Glossary** “HEAD”: the current commit that the working tree is based on
- `reset` can change the working tree and/or staging area, but doesn’t change HEAD, i.e. what commit you’re working from
- Remove a file from the staging area (but leave the changes in the working tree):

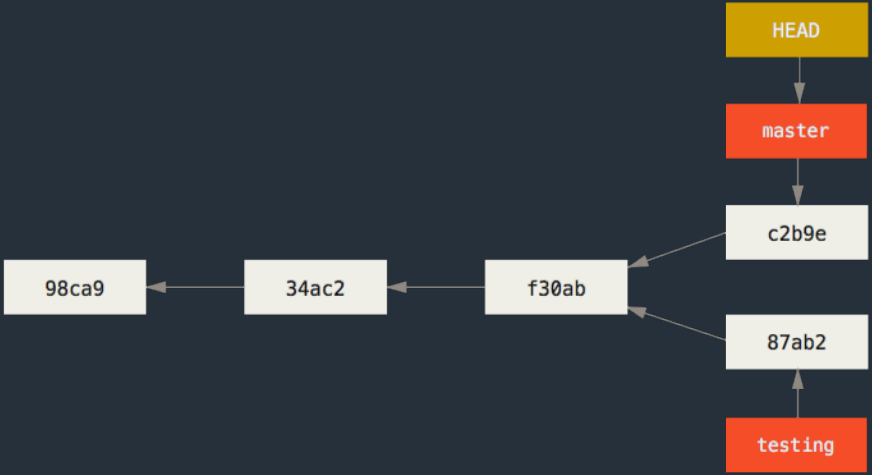
```
$ git reset HEAD <file>
```

Beyond the basics: branches

What is a branch?

- A git repo is like a tree (technically a directed acyclic graph), and like trees has branches
- More practically, a branch is just a label for a particular commit
- Can have lots of different labels on a given commit, i.e. lots of branches all the same
- Default first branch is called “master”
- You’re not forced to share your local branches with others
- Can experiment and explore, then chuck away branches that didn’t work out
- Making a new branch is always safe!

Beyond the basics: branches



Beyond the basics: branches

Making a new branch

- `git branch` command is normally not what we want
- Make a new branch and check the `status` and `log`:

```
$ git checkout -b new_branch
$ git status
$ git log
```

Switching branches

- Switch back to master, check `status` and `log`:

```
$ git checkout master
$ git status
$ git log
```

Beyond the basics: branches

Making changes on a branch

- Switch back to your new branch
- Create a new file, add some text and save it in your repo
- Add and commit this new file
- Let's look at a summary of all our branches:

```
$ git log --oneline --all --graph
```

- Can you see where `master`, `HEAD` and your new branch are?
- Switch back to `master` – what's happened to your new file?

Beyond the basics: branches

Making changes on a branch

```
$ git checkout <your branch>
# Add a new file
$ git add <new file>
$ git commit
$ git log --oneline --all --graph
$ git checkout master
$ ls
```

Moving forward: Merging

Merging

- Branches are only really useful if we can get those changes back into our main branch
- Merges are probably the biggest cause of problems, as it can be a tricky problem!
- Remember: it's **always** safe to make a new branch and try something out there!
- Several different ways to do a merge, with differing results

Merging

Fast-forwarding

- This works when the branch being merged *from* was forked from the tip of the branch being merged *into*
- Often the nicest way to merge if it's possible

```
 ^        C <-- branch2
 |        |
 |        B
 |       /
time    A   <-- branch1
```

Merging

Fast-forwarding

- This works when the branch being merged *from* was forked from the tip of the branch being merged *into*
- Often the nicest way to merge if it's possible

```
 ^        C <-- branch2  >   C <-- branch1, branch2
 |        |              >   |
 |        B              >   B
 |       /               >   |
time    A   <-- branch1  >   A
```

Merging

A simple case

- You should have two branches: `master` and `<your branch>`
- Only difference should be `<your branch>` has an extra file in it
- Very simple to merge this case!
- Checkout master and `merge` your branch
- What does the full log look like now?

```
$ git checkout master
$ git merge <your branch>
$ git log --oneline --all --graph
```

git commands

git merge

- `git merge`: Join two or more development histories together
- From the branch you want to merge something **into**, run

```
$ git merge <other branch>
```

- If you run into trouble, abort the merge, create a new branch from your “main” branch and try things out in there

```
$ git merge --abort
$ git checkout -b test-merge-branch
```

Merging

Non fast-forwarding

- If there are any “conflicts” between the two branches, it’s a little trickier
- You’ll need to resolve the conflicts, and a special “merge commit” will be created
  - This is special as it has two parents
- Some people/projects prefer to always have a merge commit as this may be easier to remove a feature if it has multiple commits

```
 ^      C    <- branch1
 |      | B  <- branch2
 |      |/
time    A
```

Merging

Non fast-forwarding

- If there are any “conflicts” between the two branches, it’s a little trickier
- You’ll need to resolve the conflicts, and a special “merge commit” will be created
  - This is special as it has two parents
- Some people/projects prefer to always have a merge commit as this may be easier to remove a feature if it has multiple commits

```
             >  D   <- branch1 (merge commit)
             >  |\
 ^      C    >  C |
 |      | B  >  | B <- branch2
 |      |/   >  |/
time    A    >  A
```

Merging

Fixing conflicts

- Conflicts happen when both branches touch the same line(s) in a file
- Conflicts are marked with a diff-like syntax
- To resolve the conflict, just go to the conflicting files and edit them appropriately
- There are lots of tools that can help you with this, e.g. ediff, meld, diff3

```
<<<<<<< HEAD
line changed in branch1
=======
line changed in branch2
>>>>>>> merging branch
```

- Just delete all the special markers and the lines(s) you don't want to keep
- Sometimes you want some combination of both regions – just edit the lines to keep what you want
- You can bail out of a merge with `git merge --abort`

Merging

Creating conflicts

- Make a new file call `conflicts.txt` with the contents "some words"
- Commit this file to `master`
- Now create and checkout a new branch called `conflict-branch`
- Change `conflicts.txt` to read "some **more** words"
- Commit this file on `conflict-branch`
- Checkout `master` again and change `conflicts.txt` to read "some **other** words"
- Commit this file on `master`
- Now try to merge `conflict-branch` into `master`
- Fix the conflict, save the file

Merging

Creating conflicts

```
$ echo "some words" > conflicts.txt
$ git add conflicts.txt; git commit
$ git checkout -b conflict-branch
$ echo "some more words" > conflicts.txt
$ git add conflicts.txt; git commit
$ git checkout master
$ git log --all --oneline --graph
$ echo "some other words" > conflicts.txt
$ git add conflicts.txt; git commit
$ git merge conflict-branch
# Remove conflict markers from conflicts.txt
$ git add conflicts.txt; git commit
$ git log --all --oneline --graph
```

Merging

Rebasing

- **DANGER!** This rewrites history!
- This has the biggest potential to cause headaches
- Rebasing a branch onto another one means to change the first branch's "base" to the new branch
- It works by "replaying" the commits on top of the second branch

```
  ^        C   <- branch1
  |        | B <- branch2
  |        |/
time       A
```

Merging

Rebasing

- **DANGER!** This rewrites history!
- This has the biggest potential to cause headaches
- Rebasing a branch onto another one means to change the first branch’s “base” to the new branch
- It works by “replaying” the commits on top of the second branch

```
             >   B' <- branch2
 ^      C    >   |
 |      | B  >   C  <- branch1
 |      |/   >   |
time    A    >   A
```

- B' contains (roughly) the same diffs as B

Beyond the basics: branches

Stashes

- Very often the case that you want to switch branches, but you have made changes that either clash with the other branch, or you just don't want to carry over
- Stashes are like commits on "anonymous" branches
- They save both your staged and unstaged changes, then discard them from your working tree
- Change your file, then `stash` it
- View the stashes with `stash list`, and re-apply the latest stash with `stash apply`

```
$ git stash
$ git stash list
$ git stash apply
```

Getting started with Bitbucket

Signing up

- Sign up with York email address to get academic account
- Two steps: sign up for Bitbucket account, and then get a username
    - I know, it's weird

Getting started with Bitbucket

Make a repository on Bitbucket

- Click the big + on the left
- Click “Repository”
- Give it a name and decide if you want it public or private
- Click “Create repository”

Add the remote repository

- Follow the instructions on Bitbucket:

```
# Add the remote
$ git remote add origin https://bitbucket.org/<username>/into_to_git.g
$ git push origin master
```

git commands

git push

- `git push`: Update remote refs along with associated objects
- **Glossary** “remote”: a version of this repository that is located elsewhere
- **Glossary** “refs”: reference to some git object (normally a branch)
- **Glossary**: “tracking branch”: a local reference to this branch on a remote repo

Getting started with Bitbucket

Getting remote changes

- From the three-dot menu in the top right, click “Add file”
- Name the file “README.md” and some text
- Click “Commit” in the bottom right
- Now we need to get this file in our local version. . .
- The quick way:

```
$ git pull
```

git commands

git pull

- `git pull`: Fetch from and integrate with another repository or a local branch
- If the branch has a tracking branch (i.e. is linked to some branch on a remote), then `git pull` does the Right Thing
- Otherwise, specify remote and branch: `git pull <remote> <branch>`

Random notes

Working with others

- git does not enforce a particular way of working with other people
- Easiest method is “feature branches”
- Everybody works in branches off the main “master” branch
- When it’s ready to share with others, make a “pull request”
- Other people can check your work
- Easy to resolve conflicts

Further reading

- Git book: **https://git-scm.com/book**
- Atlassian tutorial: **https://www.atlassian.com/git/tutorials**
- Codecademy: **https://www.codecademy.com/learn/learn-git**

Image credits

- openclipart:
  - https://openclipart.org/detail/16441/box
  - https://openclipart.org/detail/6902/lockers
  - https://openclipart.org/detail/219891/readme-fixed
- Atlassian