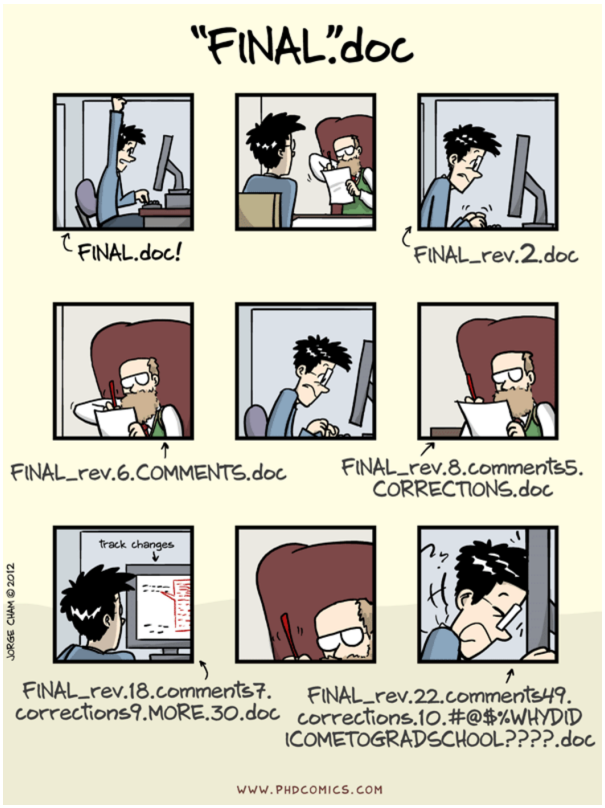
Git for Metadata - Data Scripting Bootcamp

Introduction to Git

What is Git?

Git is a 'free and open source distributed version control system'.



Replace the draft document with metadata. We are working on a dataset and we attempt to maintain multiple versions of this data, all at different stages of completeness, many of which may have comments, suggestions or other forms of input from other people.

Version control addresses this problem by recording the changes made to a file as we add to it, edit it, and improve it. Each 'commit' we make ('commit' is version control terminology for 'save') is recorded. We can go back to earlier versions of that file, or of other files we have committed, and look at the changes have been made.

Distribution of Development

True version control allow us to take this one step further. It can not only record changes to a file one person is working on but lets multiple people work on that same file and record the changes *they* make to it. It is then possible to merge these multiple files ('versions') into one.

This method came about to help developers work collaboratively on coding projects. With version control, groups did not have to wait for someone else to finish working, nor did they have compare changes manually.

When we make 'commits', we record a range of metadata about that change. As librarians, we might already know creating metadata is useful but an example of what information is recorded will illustrate why the information recorded by version control systems in particular is useful. 'Commits' record the time and date a commit was made. Although we can often see information about when we last edited or saved say, a Word document, the document itself reflects only the most recent version - all earlier versions are lost, unless we have saved them under different names.

When we make commits in a version control system, we can record a message explaining, in as much detail as we like, what changes we have made to a document or file. This makes it especially useful for collaborating. Rather than sending an email with a document with track changes and some comments, we can include all that information with the document itself. This makes it easy to get an overview of changes that have been made to a document by looking at a log of all the changes that have been made over time. And all earlier versions of the documents still remain in their original form, should we ever wish to 'roll back' to them.

Git vs. GitHub

We often hear the terms Git and GitHub used interchangeably but they are slightly different things. Git refers to the software and principles used for a particular flavour of version control system, also called VCS in short (there are other systems such as Mercurial and SVN). GitHub is a popular Web site which hosts Git repositories. The majority of the content that GitHub hosts is open source software, though increasingly it is being used for other projects such as open access journals, blogs, and constantly updated text books. GitHub is a great place to learn how to use Git but once you have learned the ideas and processes behind GitHub you can use Git on other storage systems. You can even host repositories on your own server or computer if you want to keep your files private or if you wanted to encrypt your repository. You can get private repositories on GitHub for a fee.

Using Git

One of the main barriers to getting started with Git is the language. Although some of the language used in Git is fairly self-explanatory, other terms are not so clear. The best way to get to learn Git language - which consists of a number of verbs, or commands, e.g. add, commit, preceded by the word Git - is by using it but having an overview of the language and the way Git is used will provide a good starting point.

Exercise for our first git repository

We will try to do this session as a group, but those who prefer to go at a slower pace can follow the instructions on the github page.

Some Git commands/language - what they mean and how to use them

Repository: A repository is the place where projects and associated changes are stored. Repositories can contain one single README file or hundreds of different folders making up the source code for extensive projects. We can create repositories in a number of different ways; we can make our own from scratch, we can fork (copy) an existing repository or we can create a Git repository from an existing folder we have been working on.

init: Create a repository

Whenever we use Git on the command line, we need to preface our command with `git`. This is so the computer knows we are trying to get Git to do something rather than another program.

To try out some of the Git commands, we can make a repository for today's session. We can either delete this directory later or keep it as a place to test out new Git commands.

```
$ mkdir git_test
$ cd git_test
$ git init
```

This initiates `git_test` as a git repository.

If you do an `ls` now, the repository might seem empty. However, an **`ls -a`** will show the hidden files, which includes the new file `.git`.

This signifies that the directory is now a Git repository. Were the `.git` file ever to be deleted after you have begun committing files, all versioning of the data would be lost. We now need to configure Git locally - this need only be done once, i.e. the first time we are setting up Git. However, each of the settings can be changed at any time if we decide we want to use a different email address, say, or switch to a different text editor.

```
$ git config --global user.name "Mary Citizen"
$ git config --global user.email "mary_citizen@gitmail.com"
$ git config --global color.ui "auto"
$ git config --global core.editor "nano -w"
```

Now we have set the directory up as a repository, and configured it locally, we can see how we are going.

Git status

we can use `git status` at any time to let us know what Git is up to.

If we try it now, we should get something like this

```
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

This is telling us that we are on the master branch (more on this later) and that we have nothing to commit (nothing to save changes from). If we use list directory we can see currently we don't have any files to track. Let's change that by adding a txt file. Touch allows us to create an empty file.

```
$ ls
$ touch git_test.txt
```

We now have a txt file. If we try `git status` again, we will get the following

```
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    git_test.txt

nothing added to commit but untracked files present (use "git add" to track)
```

This status is Git telling us that it has noticed a new file in our directory that we are not yet tracking. With colorised output, the filename will appear in red.

To change this, and to tell Git we want to track any changes we make to git_test.txt, we use `git add`

```
$ git add git_test.txt
```

This adds our txt file to the **staging area** (the area where Git checks for file changes). Because we are not alerted that this has happened, we might want to use `git status` again.

```
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   git_test.txt
```

If we opted for colorized output, we can see that the file has changed color (from red to green) and Git also tells us that it has got a new file.

Let's make some changes to this file before we commit it:

```
# for windows
$ notepad git_test.txt
# for mac
$ open git_test.txt
```

Open the file in whatever text editor is not Word - textedit, notepad, nano, vi, emacs, atom, sublime, textwrangler, ...

We should now be able to add some text to our text file. For now let's just write 'hello world', save and exit.

If we try `git status` again. We should get the following message

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   git_test.txt
```

This lets us know that Git has spotted changes to our txt file but that it hasn't yet 'staged' them. This means Git won't currently record the changes we made. We can add the file to the staging area again

```
$ git add git_test.txt
```

We can now **commit** our first changes. Commit is similar to 'saving' a file to Git. However compared to saving, a lot more information about the changes we made is recorded and visible to us later.

```
$ git commit -m 'hello world'

[master (root-commit) 27c3f19] hello world
 1 file changed, 1 insertion(+)
 create mode 100644 git_test.txt
```

We can see that one file changed and we made one insertion which was our 'hello world'. We have now recorded our changes and we can later go back and see when we made changes to our file and decided to add 'hello world'. We do have a problem now though. At the moment our changes are only recorded on our computer. At the moment, if we wanted to work with someone else, they would have no way of seeing what we've done. Let's fix that. Let's jump to the GitHub website where we could decide to host some of our work. Hosting here will allow us to share our work with our friends and colleagues but will also allow other people to use or build on our work.

Sharing your work on GitHub

When we have logged in to GitHub, we will see an option to create a new repository. Let's make one for the GitHub experiments we are going to do today.

- new repository
- give it a name

GitHub will ask you to create README.md, add a license and a `.gitignore` file. **Do not do any of that for now.** Once we have created the new repository, we still need to link the repository we have on our computer with the one we've just set up on GitHub.

```
$ git remote add origin <web address of your repo.git>
```

This will add a repository on GitHub for our changes to be committed to.

Check that it is set up correctly with the command:

```
$ git remote -v
origin        <web address of your repo.git> (fetch)
origin        <web address of your repo.git> (push)
```

Then enter:

```
$ git push -u origin master
```

git push

git push will add the changes made in our local repository to the repository on GitHub. The nickname of our remote is origin and the default local branch name is master. The `-u` flag tells Git to remember the parameters, so that next time we can simply run `git push` and Git will know what to do. Go ahead and push it!

You will be prompted to enter your GitHub username and password to complete the command.

When we do a `git push`, we will see Git 'pushing' changes upstream to GitHub. Because our file is very small, this won't take long but if we had made a lot of changes or were adding a very large repository, we might have to wait a little longer. We can get to see where we're at with `git status`.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

This is letting us know where we are working (the master branch). We can also see that we have no changes to commit and everything is in order.

We can use `git diff` to see changes we have made before making a commit. Let's add another line to our text file.

```
$ open git_test.txt
$ git diff
diff --git a/git_test.txt b/git_test.txt
index 70c379b..1d55e1a 100644
--- a/git_test.txt
+++ b/git_test.txt
@@ -1 +1,2 @@
-Hello world
\ No newline at end of file
+Hello world
+More changes to my first github file
\ No newline at end of file
```

We can see the changes we have made.

1. The first line tells us that Git is producing output similar to the Unix diff command comparing the old and new versions of the file.
2. The second line tells exactly which versions of the file Git is comparing; `df0654a` and `315bf3a` are unique computer-generated identifiers for those versions.
3. The third and fourth lines once again show the name of the file being changed.
4. The remaining lines are the most interesting; they show us the actual differences and the lines on which they occur. In particular, the + markers in the first column show where we have added lines.

We can now commit these changes again:

```
$ git add git_test.txt
$ git commit -m 'second line of changes'
```

Say we are very forgetful and have already forgotten what we changes we have made. `git log` allows us to look at what we have been doing to our Git repository (in reverse chronological order, with the very latest changes first).

```
$ git status
commit 40d3f7af25e19c06fa839a570d51e38fdb374e80
Author: davanstrien <email@gmail.com>
Date:   Sun Oct 18 15:25:19 2015 +0100
    second line of changes
commit 27c3f19c3340d930234d592928b11b63403d0696
Author: davanstrien <email@gmail.com>
Date:   Sun Oct 18 13:27:31 2015 +0100
    hello world
```

This shows us the two commits we have made and shows the messages we wrote. It is important to try to use meaningful commit messages when we make changes. This is especially important when we are working with other people who might not be able to guess as easily what our short cryptic messages might mean.

git pull

We can try to see how this works by making changes on the GitHub website and then 'pulling' them back to our computer.

Let's go to our repository. We can see our txt file and make changes. However, you may have noticed only our first change is there. This is because we didn't push our last commit yet. This might seem like a mistake in design but it is often useful to make a lot of commits for small changes so you are able to make careful revisions later and you don't necessarily want to push all these changes one by one.

Let's go back and push our changes

```
$ git push
```

Now if we go back to our GitHub repo, we can see all our changes. In the GitHub repo website, let's add an extra line of text there, and commit these changes. When we commit changes on GitHub itself, we don't have to push these.

Now let's get the third line on to our computer.

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/davanstrien/git_lesson_example
   40d3f7a..ef95e51  master     -> origin/master
Updating 40d3f7a..ef95e51
Fast-forward
 git_test.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

If we open our text file again

```
$ open git_test.txt
```

we can see our new lines from the GitHub website.

When we begin collaborating on more complex projects, we may have to consider more aspects of Git functionality, but this should be a good start.

Review & Exercises

To try to reinforce how things work we can work in groups to develop diagrams to illustrate Git functions and language. This should make carrying out more complicated aspects of Git clearer in our heads.

(Optional) In groups:

- illustrate the concepts discussed in the first hour
- try to 'draw' what different commands mean
- try to come up with synonyms for what the commands are doing.

(If time) Exercise - visualizing git

In group work, spend some time trying to illustrate some of the commands we've used with Git:

- try to express git commands in a non 'git' way
- try to visualise what commits are doing to your repository

If you want to practice more feel free to keep practicing making changes to your file and committing the changes. If you want to explore more git commands, search for some more online or follow one of the suggested links below.

(If time) Exercise - create a branch of the workshop repository and push your changes

Branches are a helpful way in a git repository to create a repository-local copy of the data in a new tracking space, add updates for a particular issue, then eventually merge that branch back with the master or main working branch.

Alone or in small groups, try pulling the workshop repository to your local computer, creating a new branch, adding some changes, then pushing those changes in that new branch back to the GitHub repository. The steps are below; see if you can start to pick up what they do:

```
# Change into the Home Directory
$ cd
# Change into the Desktop (or wherever you're putting your workshop materials)
$ cd Desktop/
# Create & move into a new directory for our GitHub experiment
$ mkdir github-branch-experiment
$ cd github-branch-experiment
# clone the Workshop GitHub Repository to our local machine
$ git clone https://github.com/cmh2166/CUL-MWG-Workshop.git
  Cloning into 'CUL-MWG-Workshop'...
  remote: Counting objects: 74, done.
  remote: Compressing objects: 100% (54/54), done.
  remote: Total 74 (delta 16), reused 69 (delta 13), pack-reused 0
  Unpacking objects: 100% (74/74), done.
# Move into the Git repository we just cloned locally.
$ cd CUL-MWG-Workshop/
# Check out the current branch (there is only 1)
$ git branch
  * master
# create a new branch - your uni then '_branch'
$ git branch cmh329_branch
$ git checkout cmh329_branch
  Switched to branch 'cmh329_branch'
# touch is a bash command to create a new, empty file.
$ touch my_new_file.txt
# add some text to that new file.
$ echo "Hey new text file, how are you." >> my_new_file.txt
# add that new file to our git repository, then commit it.
$ git add .
$ git commit -m 'I added my new text file for a test'
  [cmh329_branch d55367e] I added my new text file for a test
   1 file changed, 1 insertion(+)
  create mode 100644 my_new_file.txt
# push our changes back to GitHub, but creating our new branch remotely as well
# you'll need to alert Christina to your GitHub username so she can add you to it
# for this to work
$ git push origin cmh329_branch
  Counting objects: 3, done.
  Delta compression using up to 4 threads.
  Compressing objects: 100% (2/2), done.
  Writing objects: 100% (3/3), 333 bytes | 0 bytes/s, done.
  Total 3 (delta 1), reused 0 (delta 0)
  remote: Resolving deltas: 100% (1/1), completed with 1 local objects.
  To https://github.com/cmh2166/CUL-MWG-Workshop.git
   * [new branch]      cmh329_branch -> cmh329_branch
# Go to GitHub and see if you can find your new branch.
```