Merchant Venturers School of Engineering

Outreach Programme

What Is Git?

And how do I use it?

Created by

Ed Nutting

Organised by

Caroline.Higgins@bristol.ac.uk

Published on July 21, 2016

Notes to the reader

- This workshop is intended to last 1 hour.
- This workshop is intended for people with no prior knowledge of Git, GitHub or GitKraken. You don't need to be able to use the command line for this tutorial (or to use Git!).
- This workshop aims to teach the skills needed for working on small Outreach projects (i.e. other workshops). As such, we will ignore many of the feature of Git that you would need in industry. This is a good starting point though.
- The content is intended to be learnt through self-directed individual learning by following the worksheet and practicing actions repeatedly until you remember how to do them.
- The learning platform is Git for Windows or Linux and GitHub. For both platforms, we will use the GitKraken user interface (sorry, but command line people can probably already figure out how to use Git on their own).
- This workshop probably works on Mac but is untested (unless anyone fancies donating a Mac to me? :) #worthatry)
- There are many of versions of Git, all of which should be compatible with this workshop (but might not be - if you follow this guide, you'll end up with the right one installed.)
- You should already be comfortable using your platform of choice including: installing programs, opening programs, managing files and folders, signing up for stuff via email and accessing the internet (which is presumably how you got this worksheet in the first place?)
- This workshop teaches the following skills:
  - Creating an account and repository on Github
  - Cloning a new/empty repository
  - Configuring a new/empty repository
  - Installing a git client
  - Configuring a git client
  - Adding (/creating) files inside your project
  - Pushing and pulling files
  - Viewing the commit history
  - Correcting a mistake in a file
  - Discarding some or all changes to a file
  - Viewing, creating and commenting-on issues on GitHub

1 Introduction

Hi! In this short workshop we're going to try to introduce you to Git - industry's (pretty-much) standard tool for version control and what we use in the MVSE Outreach programme for managing files. Don't worry if you have no idea what "Version Control" is just yet - we're going to start from the ground up.

1.1 What is version control?

Let's suppose you create a file - something simple like a Word document. You write an essay, suppose 10,000 words, as a first draft. Somewhere along the line you write a really good paragraph with a great explanation of a concept. You send your essay off for review. When you get it back, its got lots of comments on it. So you save the comments and duly start editing. After a few days, you realise that as part of your editing you've deleted that really good paragraph because it didn't seem relevant but now you want it again - what do you do? Try to remember it and write it again? What a pain that would be...

A lot of people solve this by saving new copies of files as "My File Version 1/2/3/4/etc." or (possibly better) "My File - YYYY-MM-DD hh:mm". This has several issues, namely: it generates a lot of files, it's very hard to compare versions, it's hard to find which the first/last version was that had a certain bit of content, it's hard to manage and maintain, it only takes on wrong save to destroy a version, and so on and so forth. Ultimately, it's an ineffective way of managing lots of versions of a file. It also becomes next to impossible to share all the versions with other people and takes up a huge amount of disk space with lots of duplication.

So, version control is an efficient, effective way of tracking anything from very few to very many versions of very few to very many files managed by one person or many people.

1.2 Why do we need version control software?

To solve all the aforementioned problems and many other problems. Ultimately, version control is about: maintaining a stable latest version, keeping a history of previous versions, being able to recover stuff from previous versions, and being able to share any and all versions with other people.

Most version control also includes a way of comparing versions to see what changed (sometimes call a comparison or "diff" (difference) tool). "diff'ing" two files or folders means using a diff tool to highlight all the differences between the two files/folders. Usually we apply this to two versions of the same file, but, for example, a plagiarism tool might use a diff tool to find similarities between unrelated files.

1.3 So what is Git?

Git is a version control system. It is a system split across multiple computers - so try not to get confused. There is no "one thing" which is Git - several things piece together to make up the complete system.

What makes up the Git system then?

Several parts:

The Respository	This is the collective bucket of stuff which contains: all the versions of all the files you are tracking, information about all the people who have ever edited (inc. created, modified or deleted) a file in the repository, and some other information like branches (we won't be looking at branches in this tutorial).
The Server	Often called the "Remote Repository" - as the name suggests, it's the copy of the repository ("repo" for short) on the internet that's available to everybody else. The Remote Repo can (essentially) only ever be added to - you cannot delete versions (but you can delete content/files - a new version of something can be a version in which the thing does not exist!).
The Client(s)	Often called the "Local Respoistory" - as the name suggests, it's your copy of the repository; the copy on your computer. Nobody else can actually access what's in your local repo. If your hard drive became corrupted, there's no way to retrieve stuff that's in your local repo that hadn't been copied to the remote repo. We'll discuss later how you transfer your local copy of the repo (i.e. your local copy of all the versions) to the remote copy.

What's with the name?

I'll just quote the official answer from Kernel.org Git FAQ : https://git.wiki.kernel.org/index.php/Git

Quoting Linus: "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'".

('git' is British slang for "pig headed, think they are always correct, argumentative").

Alternatively, in Linus' own words as the inventor of Git: "git" can mean anything, depending on your mood:

- Random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- Stupid. Contemptible and despicable. Simple. Take your pick from the dictionary of slang.
- "Global information tracker": you're in a good mood, and it actually works for you. Angels sing and light suddenly fills the room.
- "Goddamn idiotic truckload of sh*t": when it breaks

1.4 I've heard a lot about the command line...

Fear not - you do not need to use the command line to use Git. The keyboard warrior CS students will, at this point, spitefully stop reading. But honestly, some very good Git User Interfaces exist and they're better than using command line (because nobody I've ever met has used Git command line without making endless mistakes or simply being less efficient than the GUI versions are). Command line is useful if you want to automate stuff or live under the belief that a mouse is the embodiment of Satan.

1.5 What is a repository (a.k.a. repo)?

I mentioned this above: it's the name for the collection of information being tracked by Git. "Repository" is often shortened to just "repo".

Git is very clever - it doesn't have to know about everything, so you can tell it to ignore files (e.g. temporary Word files). So the Repository refers to anything Git knows about. When Git knows about something, it means it's tracking it. If you create a new version of a tracked file (by editing it in any way, including deleting it), Git will notice and allow you to save that version permanently. "Saving a version" is called "committing" a version - more on this later.

The repo also contains information about people; Specifically, about the people who have created versions. It tracks this so you can compare version and search for who did what and when.

1.6 So what is GitHub?

GitHub is a company who offer a free (for public use) version of the Git Server (the Git Remote Repositories). They are a company that has taken the Git Server software, bought a tonne of hardware and storage space and combined them to give you a place to use for hosting your remote repositories. All public repositories are completely free.

If you want free, private reopositories, take a look at BitBucket. Again, BitBucket is just a company who have taken the Git Server software, added some hardware and offered it for you to use for free.

The point is, GitHub and BitBucket are not "Git" - they are just two of many companies offering you a convenient way of setting up the Server part of the Git System (described previously).

1.7 Remote vs. local versions

We've talked about this a bit already. There's a copy of the repo on the computer you are using (the Local Repo) and there's a copy on the server (the Remote Repo). You save versions to your Local Repo. Then, once you're happy with your most recent version(s), you Push them the the Remopte Repo. "Pushing changes" means copying all the new versions you've saved locally up to the Remote Repo - this makes them available for everyone else to see.

1.8 And what about GitKraken?

GitKraken is a new, recently-out-Beta-testing, pretty-good application for using Git. We'll be using it in this tutorial for managing our Local Repo including pushing changes to the Remote repo.

2 Setup GitHub

2.1 Creating an account

Head to GitHub.com - https://github.com/ and sign up (or log in if you already have an account). If you're going to be working with the University of Bristol Outreach team, please sign up using your Bristol Uni email address (you'll also get access to student perks and discounts this way).

2.2 Joining the MVSE Outreach group

Email Caroline Higgins (or speak to someone already on the group) to be granted access. Please remember to provide the email address you signed up with (which should have been your Bristol Uni email) - it needs to be exactly the same, not one of the many possible variants. You find find who's (publicly visible) in the group here: MVSE Outreach People - https://github.com/orgs/MVSE-Outreach/people. By default, membership is private - it's up to you if you want to be publicly visible.

2.3 Creating a repository

Okay, you've signed up and become a member of the MVSE Outreach group. So that we don't clutter up the Outreach group, we're going to work separately from the group for the rest of the tutorial, but in future just head to the group - https://github.com/MVSE-Outreach to create new repositories within the group.

Actions

1. For now, head to Your Profile - https://github.com/YourUserName e.g. Ed Nutting - https://github.com/EdNutting.
2. Click the Repositories tab at the top
3. Click the green New button (top-right)
4. Enter a name (no spaces) e.g. LearningGit
5. Enter a description if you want to.
6. Select "Public"
7. Check "Initialize this repository with a README"
8. Click "Add .gitignore" and type in "C" (for this demo. In future, use whichever language the majority of your code will be written in for that repo)
9. Click "Add license" then select "None" (Outreach stuff is done totally open source and free)
10. Click "Create Repository"

Licenses

Use no license or a permissive license like The MIT License or The Unlicense when working on Outreach stuff. Preferably don't use any of the GNU or BSD licenses. For you own projects/repos, you're free to use whatever license you like.

2.4 Configuring a repository

Your new repo has now been created. You should have been taken to its main page and been presented with the list of files. If not, head to https://github.com/YourUserName/YourRepoName There's not much to do to configure your new repository. We're not going to touch the settings, just edit a few of the files first. We're going to edit them through the web browser for now, which allows you to make new versions in the Remote Repo directly - you wouldn't normally do this! We'll learn later how to edit files on your computer normally then push (i.e. upload) them to the remote (i.e. online) version.

.gitignore

This file uses wildcard file names (look that up online) to exclude files from the repo. Specifically, it tells the complete Git system to ignore files names matching a certain pattern thus those files aren't tracked so you can't save versions of them. This is mostly used for ignoring temporary files and build folders.

Actions

1. Click ".gitignore" in the file list
2. Click the "Edit this file" icon in the top-right corner of the file view (looks like a pen)
3. Copy and paste the text from after this box into the bottom of the file on a new line. This will ignore any temporary files created by using LaTeX - you are expected to write worksheets for your workshop using the (very simple and easy to use) LaTeX template found in https://github.com/MVSE-Outreach/LaTeX-Worksheet-Templates/

```
*.tps

## Core latex/pdflatex auxiliary files:
*.aux
*.lof
*.log
*.lot
*.fls
*.out
*.toc
*.fmt
*.fot
*.cb
*.cb2

## Intermediate documents:
*.dvi
*-converted-to.*
# these rules might exclude image files for figures etc.
# *.ps
# *.eps
# *.pdf

## Generated if empty string is given at "Please type another file name for output:"
.pdf

## Bibliography auxiliary files (bibtex/biblatex/biber):
*.bbl
*.bcf
```

```
*.blg
*-blx.aux
*-blx.bib
*.brf
*.run.xml

## Build tool auxiliary files:
*.fdb_latexmk
*.synctex
*.synctex(busy)
*.synctex.gz
*.synctex.gz(busy)
*.pdfsync

## Auxiliary and intermediate files from other packages:
# algorithms
*.alg
*.loa

# achemso
acs-*.bib

# amsthm
*.thm

# beamer
*.nav
*.snm
*.vrb

# cprotect
*.cpt

# fixme
*.lox

#(r)(e)ledmac/(r)(e)ledpar
*.end
*.?end
*.[1-9]
*.[1-9][0-9]
*.[1-9][0-9][0-9]
*.[1-9]R
*.[1-9][0-9]R
*.[1-9][0-9][0-9]R
*.eledsec[1-9]
*.eledsec[1-9]R
*.eledsec[1-9][0-9]
*.eledsec[1-9][0-9]R
*.eledsec[1-9][0-9][0-9]
*.eledsec[1-9][0-9][0-9]R

# glossaries
*.acn
*.acr
*.glg
*.glo
*.gls
*.glsdefs

# gnuplottex
*-gnuplottex-*

# gregoriotex
*.gaux
*.gtex

# hyperref
*.brf

# knitr
*-concordance.tex
# TODO Comment the next line if you want to keep your tikz graphics files
*.tikz
```

```
*-tikzDictionary

# listings
*.lol

# makeidx
*.idx
*.ilg
*.ind
*.ist

# minitoc
*.maf
*.mlf
*.mlt
*.mtc
*.mtc[0-9]
*.mtc[1-9][0-9]

# minted
_minted*
*.pyg

# morewrites
*.mw

# mylatexformat
*.fmt

# nomencl
*.nlo

# sagetex
*.sagetex.sage
*.sagetex.py
*.sagetex.scmd

# scrwfile
*.wrt

# sympy
*.sout
*.sympy
sympy-plots-for-*.tex/

# pdfcomment
*.upa
*.upb

# pythontex
*.pytxcode
pythontex-files-*/

# thmtools
*.loe

# TikZ & PGF
*.dpth
*.md5
*.auxlock

# todonotes
*.tdo

# easy-todo
*.lod

# xindy
*.xdy

# xypic precompiled matrices
*.xyc

# endfloat
```

```
*.ttt
*.fff

# Latexian
TSWLatexianTemp*

## Editors:
# WinEdt
*.bak
*.sav

# Texpad
.texpadtmp

# Kile
*.backup

# KBibTeX
*~[0-9]*
```

Actions

4. Click the green "Commit changes" button

Read-Mes

The Read Me (README.md) file is a Markdown formatted text file which, by default, is displayed beneath the list of files for your repo online. Edit at some point to include information about your workshop Like putting your name on it so you can use it in your CV ;))

3 Setup GitKraken

3.1 Re-cap: What is GitKraken?

Okay, let's re-cap:

- Git is a system
- The system is made up of the Remote Repo, the Local Repo and the Git Client
- The Remote Repository is the copy of the repo on the server
- The Local Repository is the copy of the repo on your computer
- The Git Client is the program running on your computer which manages the local repo and handles syncing it with the remote repo.

Notes

To satisfy your curiosity, yes, there is a Git Server which does the counter-part to the Git Client - it's software running on the server (e.g. the servers provided by GitHub) that manages the remote repository (including handling syncing from many clients at once).

However, the Git Client is, unfortunately, not just one piece of software. Git Client is often used to refer to both:

- the underlying Git Client Program (which is what does all the hard work and uses a command-line interface),
- and the GUI application used to wrap-up the complexities of using the command line into a (relatively) easy to use and helpful user interface

GitKraken is a Git Client in the sense that it provides both a stable version of the Git Client Program and it provides a User Interface. In other words, GitKraken is packaged with Git and its own GUI software.

3.2 Installing GitKraken

Installing GitKraken is really easy.

Actions

1. Head to GitKraken Downloads - https://www.gitkraken.com/download
2. Download the latest version for your platform.
3. Then just run the installer!

3.3 Configuring GitKraken

When you first start GitKraken it'll take you through a tutorial / configuration procedure.

Actions

1. Try to follow the setup tutorial. Once you're done with the tutorial, follow the next few steps.
2. Open Preferences by going to File → Preferences...

Actions

3. Under General:

Project Directory	Browse to the folder you want to store your projects in. I recommend (for Windows): Documents Outreach Projects
Auto-Fetch Interval	60
Auto-Prune	Checked
Merge Tool	<None> or download and install the free version of "Beyond Compare" - it's very good!
Delete .orig files after merging	Checked
Send usage data	(Your choice)
Send crash data	(Your choice)

Actions

4. Under Git Config:

Name	Your full name please! We (current students) might know who you are now, but this name will be in the repo for years to come, so future years need your full name to know who you are!
Email	Your University email address (must be precisely the same as what you signed up to GitHub with)
AutoCRLF	Checked

Actions

5. Under Authentication:

SSH Defaults	Leave as-is
GitHub.com	Follow the in-app instructions to connect GitKraken to GitHub - this makes everything much easier later on.
Bitbucket.org	(Ignore)

Notes

The rest of this workshop will assume you have linked GitKraken to your GitHub account.

Actions

6. Under Git Flow: Leave everything as-is (should be as below)

Master	master
Develop	develop
Feature	feature/
Release	release/
Hotfix	hotfix/
Version Tag	<Empty>

Actions

7. Under UI Preferences: Choose your theme (I prefer Light)

4 Using GitKraken

Notes
This section is going to go through the basics of using any Git Client but specifically we'll use Git Kraken. All the terminology stays the same between Git Clients though, in case you want to use a different one.

Notes
I do not attempt to cover every aspect of using Git, not even some of the more fundamental (though not basic) topics such as things called "branches". The steps covered here are designed to be enough for you to get up and running with a workshop repo for contributing to the MVSE Outreach programme. There are plenty of good, more thorough tutorials online that are only a Google search away.

4.1 Cloning a repository

This is the first step in starting work on a repo on any computer for the first time. "Cloning" basically means "downloading". So "Cloning a repo" means "Downloading a copy of the remote repo to create a new local copy of the repo". I've added a bit more detail there.

It's important to realise that Cloning a repo creates a NEW local copy of the repo (i.e. a new local repo). You can't use cloning to update an existing local copy of the repo and you can't use cloning to merge an existing folder of code with the repo.

We clone a repo by telling the Git Client:

- Where to clone the repo to i.e. the folder on your computer to copy the remote repo into
- The repo to clone using the GitHub tab of the Clone window - we'll go through this in a moment

Notes

Cloning cannot be used to update an existing local repo to match the remote copy.

Notes

Cloning cannot be used to merge a remote repo with some existing local files/folders.

Notes

Following that last note, this means when you Clone a repo, the folder you choose to clone to (i.e. to copy the remote repo into) must be empty before you start. This is most easily achieved by creating a new folder.

Notes

To really stress the point: You should only ever have to clone a repo once per computer you work on. **NEVER** delete the folder containing the local copy of the repo - doing so will only bring you pain and misery and solves literally nothing. Problems with Git are solved by working within the repo, not by deleting the repo!

Actions

1. Go to File →Clone
2. Select Github.com
3. Pick a root folder (e.g.
   Documents
   Outreach
   Projects
) - a new subfolder for your project will automatically be created.
4. Select teh repo to clone from the list (e.g. LearningGit)
5. Double check the full path (displayed only after selecting both of the above) is as you desire.
6. Click the green "Clone the repo!" button
7. After cloning, you may be asked if you wish to open the repo - click "Open the repo" is asked.

4.2 Steps for making changes

Changing (i.e. creating, editing, deleting, copying or moving) files is fairly easy but has a few steps. Always ensure you follow these steps. Over time, you'll find they become automatic and quicker.

Notes
Git doesn't actually "track" folders - only files. So empty folders are ignored by Git. Git will only notice folders if you put a file or non-empty subfolder in them.

Actions

1. Pull any new changes from the remote repo by clicking the "down arrow".

Notes
Pulling changes means syncing your local copy of the repo with the remote one. If you, or anybody else, has uploaded new versions (i.e. commits) to the remote repo (on a computer other than the one you are using), then you have to download those changes. Downloading new commits (i.e. versions) is called pulling.

Notes
If you make changes in your local repo, then pull, and the remote version also has changes to the same file your local version does, then you will get a Merge Conflict. This is because Git never obliterates work and, since there are two versions of the file that have been edited in parallel, it can't work out which to use when it downloads the remote one. Merge Conflicts are annoying and fiddly, so always make sure you pull frequently and certainly before starting any new work on a given morning/afternoon.

Actions

2. Good, you've pulled any new changes so you're up to date - now to start editing.
3. First, edit the file you want to edit. With Git, it's best to make small changes and commit new (even broken) versions of files often - think "little and often".

   Don't mass-edit a file than save that version in one big go. You should aim to commit about every 15 to 30 mins (assuming you're being productive, not just reading Facebook ;))

A **commit** is a set of changes - a set of new versions of files (remember: deleting a file is a new version in which the file does not exist). When we want to "save new versions of files" we create a new commit. Git is clever in allowing you to save only certain changes. We won't go into lots of detail about this here.

Essentially, a commit consists of a set of changes to a set of files. When you want to create a new commit, you have a choice for each file: whether to include changes to that file in the commit or not.

Notes

Files with changes that are going to be included are called “staged files”.

Files with changes that are NOT going to be included are called “unstaged files”.

Notes

Changes not included in a commit do not end up in the commit history of the local repo. (They, therefore, also don’t end up in the remote repo). This means if you lose “uncomitted” changes, they are lost permanently. This is another argument for little-and-often changes/commits.

So to make a new commit:

Actions

4. Open GitKraken
5. At the top of the history list (in the middle of the main window) you will see a "Working copy" item (it won't show if you haven't made any changes)
6. On the right hand side, you will see a column showing staged and unstaged changes: Stage any changes you want to include in the new commit by hovering over the file in the list and clicking the Stage button.
7. Enter a commit Summary in the headline box at the bottom - this should be the overarching description of the most significant change or the collective purpose of the changes.
8. Enter a detailed description of any and all changes you made in the commit (e.g. "Implemented the USB Host Controller GetMessage function." or "Wrote the content for sections 1 and 2 of the worksheet.") Bullet point lists of changes often work well here. Descriptions should describe in human terms what the changes are, not just be a verbatim copy of what was changed (since that's included in the commit anyway!)
9. Click the green "Commit" button
10. Click the Up Arrow (top-middle of the window) to Push your changes. This does the inverse of pulling. It uploads your changes (i.e. commits) to the remote repo (i.e. the server). This allows everyone else to see them.

Notes

Sometimes, pushing changes will fail. This usually means someone else has committed changes since you last Pulled. Pray and hope they haven't changed any of the same files as you did, then hit the Pull (down-arrow) button as before. Hopefully, you won't get merge conflicts and it'll all go smoothly. You can then attempt the Push (up-arrow) again.

Goals

Done! You should now have discovered how to pull changes, edited a file (e.g. README.md) and committed the changes to that file.

4.3 Viewing the commit history in GitKraken

Over time you'll build up lots of commits. This is called your commit history (and is what is actually uploaded/downloaded between local/remote repos). It's very helpful to be able to view it to see what you changed and (if your commit messages were any good!) why you made certain changes. It's so useful, it forms the central view of almost every Git Client GUI app - including GitKraken.

Actions

1. To view your commit history, just scroll up and down the list. Latest commits are shown at the top of the list.
2. Click a commit in the list to view the changes included in that commit.

   Changes are displayed as the differences between the version before and the version in the commit.

   Red changes are lines which were "deleted". Green changes are lines which were "added".

Notes

Lines which are changed (e.g. just one word changed) are generally displayed as a deletion and addition i.e. the old line was deleted, the new line was added. More sophisticated diff tools (like Beyond Compare) can show you changed lines as one line and highlight the individual words/bits which were added or removed.

4.4 Reverting changes

Actions

1. Find the commit that contains the changes you wish to revert
2. Hover over the block or individual line(s) you wish to revert
3. Click the revert button
4. This applies the inverse change to your current local repo, resulting in new changes to the file. Thus you will need to commit the new (reverted) changes.
5. It's best practice to commit reverted changes on their own, to make it absolutely clear that you're undoing a previous change and, through the commit message, why you are doing so.

4.5 Discarding changes before committing

If you have made changes but haven't committed them yet, and actually you decide you don't want them, you can remove them (permanently and entirely) by discarding them.

Actions

1. Unstage the file containing the changes you wish to discards
2. Click on the file in the unstaged list - this will open the file in GitKraken for viewing.
3. Click the button to discard all changes to the file (top-right at time of writing) or, hover over particular blocks/lines and click their respective buttons to delete the changes.

Notes

There is no way to undo discarding changes - it permanently deletes the changes - so **be careful!**

4.6 Amending commits

Notes
Don't. Never check the "Amend" box.

Never try to amend commits - it almost always ends badly. Just create a new commit and include a note in the commit message that you meant to include it in the previous.

In theory, amending works but it only works for local commits which haven't been pushed to any remote repo. Amending a commit that has been pushed will result in a complete mess and GitKraken (and most other GUIs) usually crash trying to do it.

5 Using GitHub

5.1 Viewing your repository

Actions

1. Go to the URL for your repo e.g.
   https://github.com/YourUserName/YourRepoName
   https://github.com/MVSE-Outreach/LaTeX-Worksheet-Templates

5.2 Viewing the commit history online

Actions

1. On the main repo page online, click the link in the top-left which says "XX commits" (where XX is a number).

5.3 Viewing issues (Part 1)

Actions

1. On the main repo page online, click the link in the top-left which says "Issues".

5.4 Creating an issue

Actions

1. On the main repo page online, click the link in the top-left which says "Issues".
2. Check the list of issues (e.g. by searching) to make sure your issue hasn't already been created
3. If your issues hasn't already been raised, click the green "New Issue" button on the top-right
4. Fill out the relevant information.

5.5 Viewing issues (Part 2)

Actions

1. On the main repo page online, click the link in the top-left which says "Issues".
2. Find an issue you want to read
3. Click on the issue to open/view it

5.6 Commenting on an issue

Actions

1. View/open the issue you wish to comment on
2. At the bottom of the page, fill out the comment box
3. Click the green Comment button

5.7 Closing an issue

Eventually, hopefully, an issue will be resolved. At this point:

Actions

1. Write a comment to describe what the fix was and when it was done (include the commit number from the commit history)
2. Click the "Close and comment" button (instead of the green "Comment" button).

6 Wrap-up

We hope you enjoyed this workshop! If you have any questions or run into problems, just ask one of the other outreach people - I'm sure they'll be happy to help.

7 Extra Resources

Here's a few extra resources to help you along with this worksheet and some stuff to try later.

- Google is your friend : http://www.google.com