Version Control Basics using Git

Alec Clews

# Introduction

## What is Version Control and how do Version Control Systems work?

Version Control (VC) is a common practice used to track all the changes that occur to the files in a project over time. It needs a Version Control System (VCS) tool to work.

Think about how you work on a computer. You create “stuff”, it might be a computer program you are modifying, resume for a job application, a podcast or an essay. The process we all follow is usually the same. You create a basic version and you improve it over time by making lots of different changes. You might test you code, spell check your text, add in new content, re-structure the whole thing and so on. After you finish your project (and maybe release the content to a wider audience) the material you created can be used as the basis for a new project. A good example is writing computer programs which usually consist of several different files that make up the project. Once you create a version you are happy with programs often have to be changed many times to fix bugs or add new features. Programs are often worked on and modified by many different people, many of home want to add features specific to their needs. Things can get confusing very quickly!

Because this module is written for students on the Raspberry Pi the examples we will use from now on will be based on software development projects, but remember that you can apply to the principles to any set of computer files.

The way that a VCS works by recording a history of changes. What does that mean?

Every time a change is completed (for example fixing a bug in a project) the developer decides a logical ‘’save’’ point has been reached and will store all the file changes that make up the fix in the VCS database.

The term often used for a group or changes that belong together like this is a **changeset**. As well as changing lines of code in source files there might be changes to configuration files, documentation, graphic files and so on.

Along with the changes to the files the developer will be prompted by the VCS to provide a description of the change with a **commit message** which is appended to the **commit log**.

The process of storing the changes in the VCS database (usually refereed to as the **repository** or repo for short) is called **making a commit**.

The hard work in making a commit is done by the VCS, all the developer does is issue the commit command and provide the commit message. The VCS software calculates which files have changed since the last commit and what has changed. It then stores these changes, plus the commit message, the date, time, name of the developer (committer) and other information in the repository.

Version Control is also sometimes refereed to as [Revision Control](http://en.wikipedia.org/wiki/Revision_control)

Now let’s add another layer of complexity. Our project might be big enough that we are team working on the project together and we all make changes to the digital files (also called assets). That will introduce a lot of potential problems. We’ll talk about those, and how a VCS can help.

VCS We call this creating a branch, an offshoot on which we do work that is not currently part of the main trunk – more on that later. How do you keep track of these changes? Remove mistakes, bring old material forward into new versions, merge changes from one branch to another. You could think of each job application as a mini project (as well as a resume you will need to record details of phone calls, referees, research notes etc.) and each job application will use the work you completed in your previous projects. *So how does Version Control help keep track of your work on digital files?* We’ll explain that in a minute.

## Why is Version Control is so important?

Imagine a software project. It might have hundreds of files (for example source code, build scripts, graphics, design documents, plans etc.) and dozens of people working on the project making different types of changes. There are several problems that will happen:

1. Two people might be editing the same file at once and changes can be overwritten
2. After the project has been running for some time it’s very hard to understand how the project has evolved and what changes have been made. How can we locate a problem that might have been introduced some time ago. Just fixing the problem may not be enough, we probably also need to to understand the change that introduced it.
3. If two people want to change the same file one will have to wait for the other to finish, this is inefficient
4. If two people people are making (long running) changes to the project it may take some time for the both sets of changes to be compatible with each other. If the same copy if the project is being updated with both sets of changes then the project may not work correctly or even compile

There are three core things a VCS helps do:

1. Answer the following questions: “What changes were made in the past?”, “Why were they made?” and “Who made them?” (via commit history and commit messsage)
2. Individual developers find this information useful as part of their daily workflow and it also helps organisations with their compliance and audit management if needed
3. Undo a half complete or incorrect change made in error and “roll back” to a previous version
4. Recreate a “snapshot” of the project as it was at some point in the past
5. Allow two streams of changes to be made independently of each other and then integrated at a later date (parallel development). This feature depends on the specific features of the VCS tool you are using

You may find the following additional reading useful in introducing important ideas: <http://tom.preston-werner.com/2009/05/19/the-git-parable.html>

## Types of Tools available

Distributed vs. Centralised

Modern VCS work on a distributed model (DVCS).

This means that every member of the project team keeps a complete local copy of all the changes. The previous model, still widely used with tools like Subversion, is centralised. There is only one central database with all the changes and team members only have a copy of the change they are currently working on in their local workspace.

(In version control terminology a local workspace is often called a working copy and it will contain a specific revision of files plus changes)

Open Source and Commercial Tools

There are many commercial and open source tools available in the market.

As well as the core VC operations tools will offer different combinations of features, support and integrations.

In this article we will be using a VCS called Git, a popular open source tool that uses a distributed model with excellent support for parallel development.

# Summary: What do version control tools do?

* Provide comprehensive historical information about the work done on the project
* Help prevent the lost of information (e.g. edits being overwritten)
* Help the project team be more efficient by using parallel development (and often integrating with other tools such as: Ticket Systems; built Systems; project management etc.)
* Helping individual developers be more efficient with tools such as difference reports

# Example VCS operations using Git

The rest of this article will take an hands on approach by demonstrating the use of Git to manage a simple set of changes. You should follow along on your own Raspberry Pi using a new test project as explained below.

[Git](http://git-scm.com/) is very popular DVCS originally developed to maintain the GNU/Linux kernel source code (the operating system that usually runs on the Raspberry Pi). It is now used by many very large open source projects and a lot of commercial development teams. Git is very flexible and has a reputation of being hard to use, but we are only going to concentrate on the ten or so commands you need to be useful day to day.

There are many excellent tutorials for Git on the Internet. See the External References section below.

These examples assume that you are using Raspian (Debian) Linux on a Raspberry Pi and have downloaded the Python Snakes project from <https://www.dropbox.com/s/25lxmg2bkgv4hfr/game.tar.gz> into a directory called snakes.

You can do that by running the following commands in your terminal program. You can start the terminal in the LXDE GUI from the program menu by selecting “Accessories” and then “LXTerminal”. Alternatively you can just *not* run the startx command when you log in.

mkdir snakes  
wget https://www.dropbox.com/s/25lxmg2bkgv4hfr/game.tar.gz  
cd snakes  
tar -xzf ../game.tar.gz

Initially this example assumes that the current directory is your snakes directory.

If you are unfamiliar with using commands from the terminal there is a tutorial on how to us the Linux shell, the program you are using inside the terminal, <http://linuxcommand.org/learning_the_shell.php>.

## Git Setup

1. Make sure you have the correct tools installed by typing the following commands:

* sudo apt-get install git git-gui gitk git-doc

1. Test the installation with the command

* git --version
* you should see something like
* git version 2.0.4

1. Tell Git who you are (this is very important information and is recorded in every change you make)

* git config --global user.name "My Name"  
    
  git config --global user.email "myname@example.com"

You must of course substitute your own name and email address in the correct places. Git records that information in a user configuration file called .gitconfig in your home directory

In case you exchange files with developers working on a Microsoft Windows, (which is highly likely) you should also run the command

git config --global core.autocrlf input

See <https://help.github.com/articles/dealing-with-line-endings#platform-all> for further details.

More information on setting up Git at <http://git-scm.com/book/en/Getting-Started-First-Time-Git-Setup>.

## Starting a new project by creating a repo

The next thing we need to do it create an empty Git database, called a repo (short for repository) inside our snakes directory

~ $*cd snakes*  
~/snakes $*git init*  
  Initialized empty Git repository in /Users/alec/snakes/.git/

Notice that the VC tool has created a hidden directory called .git. In Linux all file and directory (folder) names that start with a “.” are normally hidden, but you can see them with the command ls -A.

Next we issue a status command. Notice that in Git all commands are typed after the word git (e.g. git init or git status). The output from the status command is

~/snakes $*git status*  
  On branch master  
   Initial commit  
   Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
   game/  
  helloworld.py  
  if.py  
  maths.py  
  variables.py  
  while.py  
   nothing added to commit but untracked files present (use "git add" to track)

We can ignore most of the detail for now. What important is that Git:

1. Warns us that some files are not being controlled (untracked) by the VCS
2. Lists the files and directories with their status. We will see this change as we progress further in the example.

### Add the project files to version control

Before changes are added to the repo database we have to decide what will be in the commit. There might be a many changes in the files we are working on, but our changset is actually only a small number of changes.

Git has a novel solution to this called the index. Before a file change can be committed to the repo it is first added to the index. As well as adding files to the index, files can be moved or deleted. Once all the parts of the commit are complete a commit command is issued. The following examples are simple and for the time being you should just expect that before a commit is done changes are added to the index as the following example shows. (Note the trailing . to represent the current directory and its subdirectories)

~/snakes $*git add .*

This command does not produce any output by default so don’t be concerned if you get no messages. If you get a message similar to warning: CRLF will be replaced by LF ... then this is normal as well (some versions of the Snakes project are provided in Windows format text files, you can fix this with the dos2unix utility).

If we run the git status command now we get different output

~/snakes $*git status*  
  On branch master  
   Initial commit  
   Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
   new file: game/game0.py  
  new file: game/game1.py  
  new file: game/game2.py  
  new file: game/game3.py  
  new file: game/game4.py  
  new file: game/snake.py  
  new file: helloworld.py  
  new file: if.py  
  new file: maths.py  
  new file: variables.py  
  new file: while.py

This time each file that will be committed is listed, not just the directory, and the status has changed from untracked to new file.

Now that the file contents have been added to the index we can commit these changes as our first commit with git commit command. Git adds the files and related information to our repo and provides a rather verbose set of messages about what it did

~/snakes $*git commit -m “Initial Commit”*  
  [master (root-commit) 841ae8c] Initial Commit  
  11 files changed, 693 insertions(+)  
  create mode 100755 game/game0.py  
  create mode 100755 game/game1.py  
  create mode 100755 game/game2.py  
  create mode 100755 game/game3.py  
  create mode 100755 game/game4.py  
  create mode 100755 game/snake.py  
  create mode 100755 helloworld.py  
  create mode 100755 if.py  
  create mode 100755 maths.py  
  create mode 100755 variables.py  
  create mode 100755 while.py

Some interesting commands we can now run. For instance the output of git status is now

~/snakes $*git status*  
  On branch master  
  nothing to commit, working directory clean

This means that the contents of our working working copy are identical to the latest versions stored in our repo.

Another command worth running is git log, which is currently very brief as we have only have one commit. Mine looks like this

~/snakes $*git log*  
  commit 841ae8c672abac0ad9d8483fc3d68f060d9dd5d8  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Initial Commit

The meaning of the Author, Date and comment field should be obvious. The commit field will be explained later.

We now have our project under version control.

## Making a change

Now lets make a change. The first step is to create a work area in which to make the change. In Git (and many other VC tools) this dedicated work area is called a **branch**. When you first create a repo the default branch that is created is called **master**, but it’s important to know that there is nothing special about master branch, it can be treated in exactly the same way as any branches you create yourself.

If you look at the output from the status command above you can see that we are currently using the master branch in our working area.

What Change do I want to make? When I play the games of snakes the rocks are represented by “Y” which I want to change to “R”. The line in I need to change is in the file game/snake.py (lines 50 and 52 in my version).

Let’s create a branch to work on.

~/snakes $*git branch make\_rocks\_R*

No message means the command was successful (note that spaces are not allowed in the branch name). Creating a branch means that I have a working area in my project (you can think of as a sandbox for a mini project) that stops my change from breaking (or impacting) any other work that is going on in the snakes project.

You can get a lit of all the branches with the git branch command

~/snakes $*git branch*  
  make\_rocks\_R  
  \* master

The asterisk shows the current branch.

To make the make\_rocks\_R the current branch use the checkout command

~/snakes $*git checkout make\_rocks\_R*  
  Switched to branch 'make\_rocks\_R'

Now git branch displays

~/snakes $*git branch*  
  \* make\_rocks\_R  
  master

In technical terms what has happened is that Git has *checked out* the branch make\_rocks\_R into our *working directory*. The working directory contains that set of files, from specific branch, that we are currently working on. Any changes I now make are isolated in the branch and don’t impact anything else.

At this point you make want to play snakes for a couple of minutes, so that you will be able to see the difference later of course. Use the command python game/snake.py

* Changing the file

Edit the file game/snake.py using your favourite text editor. In the version of snakes I had there are two changes to make; a comment on line 50; and the actual code on line 52. Save the changes and test the game by playing it again. The rocks should now look like “R” instead of “Y”.

* Showing the diff

So let’s see what has changed. Git can provide a nice listing. The simplest way is by using the command git diff, try that know and you should see a report similar to this

~/snakes $*git diff*  
  diff --git a/game/snake.py b/game/snake.py  
  index cef8d07..7e65efe 100755  
  --- a/game/snake.py  
  +++ b/game/snake.py  
  @@ -47,9 +47,9 @@ def add\_block(scr, width, height):  
  empty = False  
    
  if empty:  
  - # if it is, replace it with a "Y" and return  
  + # if it is, replace it with a "R" and return  
    
  - scr.addch(y, x, ord("Y"), curses.color\_pair(2))  
  + scr.addch(y, x, ord("R"), curses.color\_pair(2))  
  return  
    
  def snake(scr):

This report can be a little confusing the first time you see it. However if you look carefully you can see lines marked with + and -, these are the lines that have been changed. If we have made changes to more than one file then each set of file differences would be listed. This type of information are often referred to as a **diff report** or **diff output**

You can get a more user friendly display of these differences by using a graphical compare tool. Refer to the appendixes for information on how to install and use the Kdiff3 graphical tool.

* Committing the change

Now that we have a change, it’s tested and we have verified it using the diff tool it’s time to add the change to our version control history.

This is two stage process, in a similar way to our first commit.  
 \* Add the changes to the index \* Commit the change to the repo, along with a useful comment

The first part is simple as only one file has changed.

~/snakes $*git add game/snake.py*

You should then verify that the add was successful by running a git status command.

This time when we commit we want to add a more complete report (**commit message**) and so first let’s make sure that our editor is set up in Git. As an example we’ll set up leafpad as the editor.

git config --global core.editor "/usr/bin/leafpad"

N.B. leafpad it a GUI editor and you will need to run X-Windows for it to work. If you are not using X-Windows or prefer a different editor then use the appropriate program name e.g. /usr/bin/vim

Now let’s make the commit. This time the command is a little simpler git commit but something a little more spectacular will happen. You editor will pop into life in front of you with information ready for your to write a commit message.

You now have two choices:

1. Exit the editor without saving any changes to the commit message : The commit is aborted and no changes occur in the repo (but the index **still** contains the change)
2. Enter some text, save it and exit the editor  
   : The commit is completed and all changes are recorded in the repo.

* A word about commit messages: The commit messages consists of two parts. Line 1 is the header and should be followed by a blank line. The header is displayed in short log messages. After the blank line comes the message body which contains the details. A detailed set of suggestions can be read at <http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html>

An example of the commit message that might be used for the change we have just made

  Changed Rocks Y -> R  
   1. Changed all references to rocks from the char "Y" to "R"  
  a. In a comment  
  b. In a single line of code  
   2. Tested

so when we run the git commit we get the following output

  [make\_rocks\_R 14ed2f8] Changed Rocks Y -> R  
  1 file changed, 2 insertions(+), 2 deletions(-)

* Showing the history

~/snakes $*git log*  
  commit 14ed2f813323a897f92db92af7a5002f15cfe68b  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Changed Rocks Y -> R  
    
  1. Changed all references to rocks from the char "Y" to "R"  
  a. In a comment  
  b. In a single line of code  
    
  2. Tested  
   commit 841ae8c672abac0ad9d8483fc3d68f060d9dd5d8  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Initial Commit

You might care to look at  
<http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository>

* Branches

We now have two branches master and make\_rocks\_R. Let’s make another change on a new branch and then look at the history.

1. Make sure that we are using the master branch  
   ~/snakes $*git checkout master*  
     Switched to branch 'master'

Now let’s examine the file snake.py again. This time I’ve noticed that when setting up colours (with the method call curses.color\_pair()) the original programmer used a literal constant. It is good practice to use a more meaningful symbolic names (like curses.COLOR\_RED instead of the literal value ‘1’).

So I’m going to make two changes. The text curses.color\_pair(2) will be changed to curses.color\_pair(curses.COLOR\_GREEN) and the text curses.color\_pair(1) will be changed to curses.color\_pair(curses.COLOR\_RED)

(documentation on the Curses library is at <http://docs.python.org/howto/curses.html>)

~/snakes $*git branch use\_curses\_symbols*  
 ~/snakes $*git checkout use\_curses\_symbols*  
  Switched to branch 'use\_curses\_symbols'

If I run the command git diff I can see the following report  
~/snakes $*git diff*  
  diff --git a/game/snake.py b/game/snake.py  
  index cef8d07..a35c1df 100755  
  --- a/game/snake.py  
  +++ b/game/snake.py  
  @@ -49,7 +49,7 @@ def add\_block(scr, width, height):  
  if empty:  
  # if it is, replace it with a "Y" and return  
    
  - scr.addch(y, x, ord("Y"), curses.color\_pair(2))  
  + scr.addch(y, x, ord("Y"), curses.color\_par(curses.COLOR\_GREEN))  
  return  
    
  def snake(scr):  
  @@ -145,7 +145,7 @@ def snake(scr):  
    
  # replace the character with a "O"  
    
  - scr.addch(y, x, ord("O"), curses.color\_pair(1))  
  + scr.addch(y, x, ord("O"), curses.color\_par(curses.COLOR\_RED))  
    
  # update the screen  
  

Now run the program to make sure it still works correctly

Now we can add and commit our changes.

~/snakes $*git add game/snake.py*  
 ~/snakes $*git commit -m “Use curses lib symbolic names in color\_pair() method calls”*  
  [use\_curses\_symbols 6d08577] Use curses lib symbolic names in color\_pair() method calls  
  1 file changed, 2 insertions(+), 2 deletions(-)

Now if we run the git log command we only see two commits  
~/snakes $*git log*  
  commit 6d0857785aa630075f279cb862583fa7e4b73683  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Use curses lib symbolic names in color\_pair() method calls  
   commit 841ae8c672abac0ad9d8483fc3d68f060d9dd5d8  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Initial Commit

What happened to our other commit where we changed the colour of our rocks? The answer is that it’s on another branch – it’s not part of the history of our current workspace.

Add the option --all to see all the commits across all the branches.

~/snakes $*git log –all*  
  commit 14ed2f813323a897f92db92af7a5002f15cfe68b  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Changed Rocks Y -> R  
    
  1. Changed all references to rocks from the char "Y" to "R"  
  a. In a comment  
  b. In a single line of code  
    
  2. Tested  
   commit 841ae8c672abac0ad9d8483fc3d68f060d9dd5d8  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Initial Commit  
   commit 6d0857785aa630075f279cb862583fa7e4b73683  
  Author: Alec Clews <acdsip61-pi@yahoo.com>  
  Date: Thu Aug 14 09:36:43 2014 +1000  
   Use curses lib symbolic names in color\_pair() method calls

As you can see git commands take extra parameters to change the way the work. A useful way to see the above history using quite a complex log

git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s  
 %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative --all

It’s quite hard work to type this in, luckily Git has an alias feature to make life a lot simpler. Use the following command

git config --global alias.lg  
 log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s  
 %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative --all

Both these examples need to be entered on a single line of course.

Now all you need to do is type git lg as lg has become an alias for the much longer version if log I showed above. More information about aliases at <https://git.wiki.kernel.org/index.php/Aliases>

If you have installed the gitk program (as suggested previously) you can also this information in a graphical program by running gitk --all&

All the various reports that git log and gitk refer to our branches. In addition there is a HEAD. This is a reference meaning the current stuff checkout into our working copy. The HEAD always points to the commit that we last checked out.

# Merging

#TODO

# Working with others

#TODO

# Remote repos

#TODO # Patches

# Appendices

## Appendix A: External References

Several videos that introduce the basic ideas of version control can be found at <http://git-scm.com/videos> or on YouTube at:

* [Episode 1](http://www.youtube.com/watch?v=K2wBGt-j0fE)
* [Episode 2](http://www.youtube.com/watch?v=0BIGxolZQHo)
* [Episode 3](http://www.youtube.com/watch?v=ojVzmIp6Xv0)
* [Episode 4](http://www.youtube.com/watch?v=pv25aLwYpEU)
* [Pro Git](http://progit.org/), an online and published book
* [Introduction to Git](http://youtu.be/ZDR433b0HJY), video with Scott Chacon of GitHub

## Appendix B: Notes for Teachers and group facilitators

Git is a useful platform for students to work together. Additional if their programming work needs to be accessed then it provides a useful mechanism for students and teams to submit their work.

Students can collaborate using four different models:

1. Using a third party service such as GitHub, Gitorious, Gitlab or BitBucket. Examples are given below for GitHub
2. Via a private server provided by the school. This is beyond the scope of this document
3. Students can exchange work using a peer to peer model. This requires a local area for the students to exchange work #todo Add notes on remote
4. Students can exchange using Git patches on USB drives (a **sneakernet**). This is most technology simple solution but is more complicated for students and is beyond the scope of this document

### Using a third party web hosting service. GitHub

(GitHub)[GitHub.com] provide specific git hosting service for schools. see <https://github.com/edu> for details.

1. Create a a GitHub organisation for your school or organisation
2. Let GitHub know the details of your account using the form at <https://github.com/edu>. This will allow your students to have private repos. N.B. Normally all free accounts are public and the code they contain in freely available.
3. Your students should create their own account on GitHub (see Appendix B). Students can follow the instructions at <http://help.github.com/>
4. You then add your students to the appropriate GitHub organisation

## Appendix C: Notes for self study students

Students are recommended to use a remote hosting account for their code. It develops good technique and provides a useful back up

There are a variety of free services available (see appendix A for some suggestions). GitHub has easy to follow instructions at <http://help.github.com/>

## Appendix D: Using the Git Graphical Tools git gui and gitk

Al of the examples above use the command line interface. However Git does come with two GUI interfaces – git gui and gitk. gitk is useful looking at the history of changes in a repository and git gui can be used to peform operations such as add, commit, checkout etc.

Let’s replicate our previous examples using the standard git GUI tools. Create a directory called ~/snakes2 and unpack the games file into it.

Now run the command git gui

## Appendix E: Installing and Using Kdiff3

First install the kdiff3 program

sudo apt-get install kdiff3-qt

Now, instead of using git diff to get a text report of the differences in your change you can run git difftoolto scroll through a side by side list. difftool supports serveral different GUI style tools to present the differences, setting them up is left as an exercise.