Version control basics using Git

Alec Clews
edited and adapted by
Luis Kornblueh

March 16, 2015

1 Introduction to Version Control

When producing an article for a magazin like *The MagPi* up to five different people may work on the article, in addition to the author. There is testing, layout, graphics, proof reading plus edits by the Issue Editor. To ensure that all changes are recorded and everyone is working on the latest version, we use a tool called Git. Git is widely used by many organisations so it is a useful skill to have. It is also a great tool for student projects. We asked Alec Clews to explain how it all works.

1.1 What is Version Control?

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 your code, spell check your text, add in new content, restructure 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 whom want to add features specific to their needs. Things can get confusing very quickly!

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

1.2 How does a VCS work?

The way that a VCS works is 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 that a logical *save* point has been reached and will store all the file modifications that make up the change in the VCS database.

The term often used for a group of changes that belong together is **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 referred 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 referred to as Revision Control.

Now let us add another layer. Our project might be big enough that we are a 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 will now talk about those and how a VCS can help.

1.3 Why is Version Control 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 is 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 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 are making (long running) changes to the project it may take some time for both sets of changes to be compatible with each other. If the same copy of the project is being updated with both sets of changes then the project may not work correctly or even compile.

There are three core questions a VCS helps to answer via the commit history and commit messsage - what changes were made in the past, why were they made and who made them?

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.

There are also three core things a VCS helps do:

1. Undo a half complete or incorrect change made in error and **roll back** to a previous version.
2. Recreate a **snapshot** of the project as it was at some point in the past.
3. Allow two streams of changes to be made independently of each other and then integrate them at a later date (parallel development). This feature depends on the specific features of the VCS tool you are using.

You may find the article at

`http://tom.preston-werner.com/2009/05/19/the-git-parable.html`

useful in introducing important ideas.

2 Types of VCS tools available

2.1 Distributed vs Centralised

Modern VCS tools 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. Here 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.

2.2 Open source and commercial tools

There are many commercial and open source tools available in the market. As well as the core version control operations, different 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.

3 Summary on Version Control Systems

Version Control tools:

- Provide comprehensive historical information about the work done on a project.
- Help prevent the loss 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 bug tracking systems, project build systems, project management etc.)
- Help individual developers be more efficient with tools such as difference reports.

4 Example VCS operations using Git

In the following we will take a hands on approach by demonstrating the use of Git to manage a simple set of changes. You should follow along on your own computer account using a new test project as explained below.

Git is a very popular DVCS originally developed to maintain the GNU/Linux kernel source code (the operating system that usually runs on Linux boxes including 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 thus has a reputation of being hard to use, but we are only going to concentrate on the ten or so commands you need day by day.

The following examples assume that you are using a Debian based Linux (like Raspbian and Ubuntu). First we are going to download an example Python project called Snakes, which we will store in a directory called `snakes`.

You can do that by running the following commands from the command line:

```
cd ~
mkdir snakes
wget -O game.tar.gz http://goo.gl/nB4tYe
cd snakes
tar -xzf ../game.tar.gz
```

If you are unfamiliar with using commands from the terminal there is a tutorial on how to use the Linux command line at

http://linuxcommand.org/lc3_learning_the_shell.php

4.1 Git setup

Make sure you have the correct tools installed by typing the following command:

```
git --version
```

You should see something like (or newer):

```
git version 1.17.10.4
```

Tell Git who you are. This is very important information and is recorded in every change you make. You must of course substitute your own name and email address in the correct places:

```
git config --global user.name "My Name"
git config --global user.email "a@b.com"
```

Git records that information in a user configuration file called `.gitconfig` in your home directory. Note that files and directories that are prefixed with a period (`.`) are hidden. If you enter the command `ls` you will not see these files. Instead enter `ls -A` to see everything.

In case you exchange files with developers working on Microsoft Windows, 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 can be found at

http://git-scm.com/book/en/Getting-Started-First-Time-Git-Setup

4.2 Start a new project by creating a repo

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

```
cd snakes
git init
```

You should see something like:

```
Initialized empty Git repository in /home/pi/snakes/.git/
```

Git has now created a hidden directory called `.git`. Remember, use `ls -A` to see it. Next we issue a `git status` command. Notice that in Git all commands are typed after the word git (e.g. `git init` or `git status`). Enter:

```
git status
```

The output from the status command is:

```
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 is important is that Git:

- Warns us that some files are not being controlled (untracked) by the VCS.
- Lists the files and directories with their status.

We will see this change as we progress further in the example.

4.3 Add the project files to Git

Before changes are added to the repo database we have to decide what will be in the commit. There might be many changes in the files we are working on, but our changeset 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 also 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 period (`.`) to represent the current directory and its subdirectories:

```
git add .
```

This command does not produce any output by default so do not 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 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:

```
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 the `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. Enter:

```
git commit -m "Initial Commit"
```

The output from the command should be like:

```
[master (root-commit) 34738dd] 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
```

Now try the `git status` command again. The output is:

```
# On branch master
nothing to commit, working directory clean
```

This means that the contents of our working copy are identical to the latest version stored in our repo. Another command worth running is `git log`, which is currently very brief as we have only one commit. Mine looks like this:

```
commit 34738ddba9b16ee4835369abc5a72d2cb809ee7d
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:30:46 2015 +0100

    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.

4.4 Making changes

Now I will demonstrate the value of running Version Control software by showing what happens when we make changes to our project files. Before we start, make sure you are in the snakes folder. Enter:

```
cd ~/snakes
```

Now let's 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 the **master**. However, it is important to know that there is nothing special about the 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 previous `git status` command you can see that we are currently using the master branch in our working area.

4.5 What change do I want to make?

When I play the game of snakes the rocks are represented by "Y" which I want to change to "R". The lines I need to change are in the file `game/snake.py` (lines 50 and 52 in my version).

Let's create a branch to work on. Enter:

```
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 it 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 list of all the branches with the `git branch` command. Enter:

```
git branch
```

You will see something similar to:

```
  make_rocks_R
* master
```

The asterisk shows the current branch. To make the `make_rocks_R` the current branch use the `git checkout` command. Enter:

```
git checkout make_rocks_R
```

You should see the following result:

```
Switched to branch 'make_rocks_R'
```

Now when you enter the `git branch` command it displays:

```
* 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 the specific branch that we are currently working on. Any changes we now make are isolated in the branch and will not impact anything else.

At this point you may want to play snakes for a couple of minutes, so that you will be able to see the difference later. Use the cursor keys to control the snake, press <Spacebar> to restart and press <Ctrl>+<C> to exit. Enter:

```
python game/snake.py
```

4.6 Changing the file

Edit the file `game/snake.py` using your favourite text editor. In the version of snakes I have there are two changes to make - a comment on line 50 and the actual code on line 52. The changes to do are replacing the "Y" by "R".

Save the changes and test the game by playing it again. The rocks should now look like "R" instead of "Y".

4.7 Showing the diff

So let us see what has changed. Git can provide a nice listing. The simplest way is by using the command `git diff`. Enter:

```
git diff
```

You should see a report similar to this:

```
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 had made changes to more than one file then each set of file differences would be listed. This type of information is 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. Now, instead of using `git diff` to get a text report of the differences in your change you can run `git difftool` to scroll through a side by side list. The difftool command supports several different GUI style tools to present the differences. Setting them up is left as an exercise.

4.8 Committing the change

Now that we have a change and we have tested it and have verified it using the difftool command, it is time to add the change to our version control history.

This is a 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. Enter:

```
git add game/snake.py
```

You should then verify that the addition was successful by running a `git status` command. This time when we commit we want to add a more complete report (called a commit message). But first let us make sure that our editor is set up in Git. As an example we will set up emacs as the editor. Enter:

```
git config --global core.editor "/usr/bin/emacs"
```

Now let's make the commit. This time the command, `git commit`, is a little simpler but something a little more spectacular will happen. Your editor will pop into life in front of you with information ready for you 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 message 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

The following is an example of a 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
```

Enter the `git commit` command and use the above commit message:

```
git commit
```

You should get output similar to the following:

```
[make\_rocks\_R b829990] Changed Rocks Y > R
1 file changed, 2 insertions(+), 2 deletions(-)
```

4.9 Showing the history

Notice how, in the picture on the next page, the arrow points from the child commit to the parent commit. This is an important convention. Enter:

```
git log
```

You will see something like:

```
commit b829990f085f2e117950ea9b18e0d08eeebf29c7
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:39:34 2015 +0100

    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 34738ddba9b16ee4835369abc5a72d2cb809ee7d
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:30:46 2015 +0100

    Initial Commit
```

Figure 1 : A simple picture of the current repo history

For more details you might want to look at

http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository

4.10 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. Make sure that you are using the master branch. Enter:

```
git checkout master
```

You will see the output:

```
Switched to branch 'master'
```

Now let's examine the file game/snake.py again. This time I have 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 more meaningful symbolic names (like `curses.COLOR_RED` instead of the literal value 1).

We are 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 available at

https://docs.python.org/2/howto/curses.html

But before doing the editing we create a new branch. Enter:

```
git branch use_curses_symbols
git checkout use_curses_symbols
```

You will now see the output:

```
Switched to branch 'use_curses_symbols'
```

With the above commands I created a new branch (from `master`, not from `make_rocks_R`) called `use_curses_symbols` and checked it out.

Edit the file `game/snake.py` using your favourite text editor. In the version of snakes I have there are two code changes to make - one on line 52 and the other on line 148. Make the changes as described above. Save the changes and test the game by playing it again.

If I run the command `git diff` I can see the following report:

```
iff --git a/game/snake.py b/game/snake.py
index cef8d07..ec8ee6e 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_pair(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_pair(curses.COLOR_RED)
   )

       # update the screen
```

Now we can add and commit our changes. Enter:

```
git add game/snake.py
git commit -m "Use curses lib symbolic names in color_pair() method
   calls"
```

You should see the following output:

```
[use_curses_symbols 5ad7b0b] 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:

```
[use_curses_symbols 5ad7b0b] Use curses lib symbolic names in
   color_pair() method calls
 1 file changed, 2 insertions(+), 2 deletions(-)
m214089@huanglung% git log
commit 5ad7b0b666ee2e7f87fc8fbb97fc1a293f371514
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:45:05 2015 +0100

    Use curses lib symbolic names in color_pair() method calls

commit 34738ddba9b16ee4835369abc5a72d2cb809ee7d
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:30:46 2015 +0100

    Initial Commit
```

What happened to our other commit where we changed the character for our rocks? The answer is that it is on another branch – it is not part of the history of our current workspace. Add the option `-all` to see all the commits across all the branches. Enter:

```
git log --all
```

You will see something like the following:

```
commit 5ad7b0b666ee2e7f87fc8fbb97fc1a293f371514
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:45:05 2015 +0100

    Use curses lib symbolic names in color_pair() method calls

commit b829990f085f2e117950ea9b18e0d08eeebf29c7
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:39:34 2015 +0100

    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 34738ddba9b16ee4835369abc5a72d2cb809ee7d
Author: Luis Kornblueh <luis.kornblueh@mpimet.mpg.de>
Date:   Sat Mar 14 16:30:46 2015 +0100

    Initial Commit
```

Figure 2: The current repo history with three branches and one commit on each branch.

As you can see git commands take extra parameters to change the way they work. A useful way to see the above history using quite a complex log command is shown below (enter it all as one continuous line):

```
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 is quite hard work to type this in. Fortunately Git has an alias feature which allows us to simplify commands. Enter the following command, again as one continuous line:

```
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"
```

Now all you need to do in future is enter the command `git lg` as lg has become an alias for the much longer version of log shown above. More information about aliases is available at

https://git.wiki.kernel.org/index.php/Aliases

As you have installed the gitk program you can also use this to display this log information in a graphical program. Enter:

```
gitk --all &
```

All the various reports from `git log` and `gitk` refer to our branches by name. In addition, there is a HEAD revision label. This is a reference to the last commit we made on a branch, so every branch has a HEAD. Generally the HEAD refers to the last commit on the current default branch.

4.11 Commit IDs

I promised I would explain the commit field (commit ID), as shown in the output of the `git log` command. The commit ID is an important concept that deserves its own section.

In many VCS tools it is enough to give each new commit a revision number such as 1, 2, 3, and so on. We can also identify branches by using dotted numbers. For example, 3.2.5 would be the 5th revision of the 2nd branch from version 3. However in Git we are not sharing a single repo database and there has to be a way of keeping all the possible commits on a distributed project unique. Git solves this problem by using a SHA-1 string. SHA-1 (Secure Hash Algorithm) is a computer algorithm that, when presented with a string of bits (1 's and 0's), will present a different 40 character result even when two strings are different in any way, even just one bit.

You can see this effect by running the following experiment. Enter:

```
echo 'Hello World' | git hash-object --stdin
```

The result will be:

```
557db03de997c86a4a028e1ebd3a1ceb225be238
```

Now enter:

```
echo 'Hello World!' | git hash-object --stdin
```

This time the result is:

```
980a0d5f19a64b4b30a87d4206aade58726b60e3
```

This is exactly what Git does for each commit, only it uses the contents of the committed files (plus the ID of the commit parents) to calculate the new SHA-1 commit ID. If two commits from two different repos have the same ID they are the same commits and we consider them identical.

5 Using the Git graphical tools

Most of the examples so far have used the command line interface. However, Git does come with two GUI interfaces – gitk and git gui. You have already seen that gitk is useful for looking at the history of changes in a repository. git gui can be used to perform 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 `game.tar.gz` file into it. Enter the following command and explore:

```
git gui
```

6 Merging

Let's look again at the current structure of our commit tree.

Figure 1 : The current repo history with three branches and one commit on each branch.

At some point we need to bring both our changes, which we are now happy with, back into the master branch. This will make them part of the default code and we can make new changes on top of that. This process is called **merging**.

The concept is simple enough, but it is important to remember that we have three branches in this example, `master`, `make_rocks_R` and `use_curses_symbols`. Each branch has only one commit.

6.1 Fast-forward merging

The first step is to merge `make_rocks_R` into `master`. Notice that this operation is not commutative. So `make_rocks_R` merged into `master` is not the same as `master` merged into `make_rocks_R`. Make the current branch `master`. Enter:

```
cd ~/snakes
git checkout master
```

You should see the following output:

```
Switched to branch ' master'
```

Now merge from `make_rocks_R` into the current branch. Enter:

```
git merge make_rocks_R
```

You will see something similar to:

```
Updating 34738dd..b829990
Fast-forward
 game/snake.py | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Notice the phrase Fast-forward. This is because `master` has no changes of its own since `make_rocks_R` was created. In this case, all that happend was that the master pointer was moved up the graph until it pointed to the HEAD of `make_rocks_R`. In a minute we will create a merge that cannot be fast-forwarded.

The repo graph now looks like . . .

Figure 2: The repo history after our first merge.

6.2 Merging with conflicts

Now let's perform a more complex merge using `use_curses_symbols`. First let's check we are on the correct branch, `master`. Enter:

```
git branch
```

You will see the following:

```
  make_rocks_R
* master
  use_curses_symbols
```

Now enter:

```
git merge use_curses_symbols
```

This results in the following output:

```
Auto-merging game/snake.py
CONFLICT (content): Merge conflict in game/snake.py
Automatic merge failed; fix conflicts and then commit the result.
```

Now we are getting a conflict, which means that Git cannot automatically merge the two versions. This is because we have changed the same line in both branches. The git status tells that we have a half complete commit with some instructions on what to do next. Enter:

```
git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   game/snake.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Let's see what our conflict looks like. Enter:

```
git diff
```

```
diff --cc game/snake.py
index 7e65efe,ec8ee6e..0000000
--- a/game/snake.py
+++ b/game/snake.py
@@@ -47,9 -47,9 +47,13 @@@ 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

++<<<<<<< HEAD
 +          scr.addch(y, x, ord("R"), curses.color_pair(2))
++=======
+           scr.addch(y, x, ord("Y"), curses.color_pair(curses.
   COLOR_GREEN))
++>>>>>>> use_curses_symbols
            return

  def snake(scr):
```

Again we can ignore most of the this report. What is interesting is the text between the markers

<<<<<<<

=======

>>>>>>>

The markers are inserted by Git to show the lines that are different in each version.

To fix this we only need to edit the file `snake.py` and edit the text between the two markers (including the markers) to be what we want. Once this is complete try `git diff` again:

```
git diff
```

```
diff --cc game/snake.py
index 7e65efe,ec8ee6e..0000000
--- a/game/snake.py
+++ b/game/snake.py
@@@ -47,9 -47,9 +47,10 @@@ 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("R"), curses.color_pair(2))
 -         scr.addch(y, x, ord("Y"), curses.color_pair(curses.
   COLOR_GREEN))
++         scr.addch(y, x, ord("R"), curses.color_pair(curses.
   COLOR_GREEN))
           return

  def snake(scr):
```

It will probably take a little while to verify that this report shows we have completed the change. Once we are happy, (and we should also probably do a test as well), then we can add and commit the change. Enter:

```
git add .
git commit -m "Merged in Rocks being 'R' "
```

```
[master d4fb570] Merged in Rocks being 'R'
```

So `master` has now got a new commit (compared to the previous merge where it was able to *reuse* the HEAD commit on another branch i.e. the fast-forward). The new commit contains both sets of changes.

Figure 3: The repo history after our second merge.

The example merge we just completed required us to edit the merge halfway through. Life is usually much simpler as Git can perform the edit for us if the changes do not overlap, the commit is then completed in a single merge command.

6.3 Rebase

Git also has a `git rebase` command which allows us to bring branches together in very convenient ways. However, we do not have enough space to discuss that in this article but later I will suggest some online resources for you to use. I recommend getting familiar with all the great things git rebase can do.

6.4 Graphical helpers

Previously I mentioned the `git gui` program that provides a GUI interface to most of the commands we have been using so far (e.g. init, add, commit). Another program that I use a lot is `gitk` which provides a nice list of the all the commits and is easier to browse than the `git log` command. Use the `-all` parameter to see all the branches in the current repo.

6.5 Difftool

As we have already seen, the output from running the `git diff` command is not always obvious. Fortunately Git provides the `git difftool` command to display side by side differences in the GUI . A variety of third party tools are supported.

To see the difference between the `master` and `make_rocks_R` branches, enter:

```
git difftool master make_rocks_R
```

```
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld kompare
   gvimdiff
diffuse ecmerge p4merge araxis bc3 emerge vimdiff

Viewing: 'game/snake.py'
Launch 'opendiff' [Y/n] :
```

Press <Y> and the selected tool should appear.

7 Wrap up

7.1 Working with other people's code

I hope to cover this topic in a lot more detail in the future.

However, before we wrap up it is probably worth introducing the `git clone` command. This is identical to `git init` in that it creates a new repository. But it then copies the contents of another repository so that you can start working on it locally. For instance if you want to get a copy of this article to improve, enter:

```
git clone https://github.com/alecthegeek/version-control-basics.git
```

```
Cloning into 'version-control-basics' ...
```

7.2 Ignoring files

By default, every time the `git status` command is used Git reminds us about all files that are not under version control. However in most projects there are files we do not care about (e.g. editor temporary files, build time object files, etc). Create the file .gitignore in the top project directory and list all the files you want to ignore.

Note: You should still check the .gitignore files into your repo along with the other files.

8 Further reading and help

We have covered the following Git workflows:

1. Creating a new repo
2. Adding code to the repo
3. Making changes and using the index
4. Creating branches to keep changes separate
5. Using merge to bring our changes together

I had to skip over a few things so please make sure you use the following great resources to improve your knowledge.

A great jumping off point for Git is the web site

http://git-scm.com

This contains links to software, videos, documentation and tutorials.

Additionally, "Pro Git"

http://progit.org

is a highly recommended online book.

Also watch the "Introduction to Git" video with Scott Chacon of GitHub

http://youtu.be/ZDR433b0HJY

Thanks to Matthew McCullough from Github for his help with this article.