Curious git

Release

Matthew Brett

Oct 02, 2021

Contents

1 Learn git right for a long and happy life

1.1 Git – love – hate

I've used git now for a long time. I think it is a masterpiece of design, I use it all day every day and I can't imagine what it would be like not to use it. So, no question, I *love* git.

As y'all may know, Linus Torvalds wrote git from scratch. He loves it too. Here is Linus talking about git in a question and answer session:

> Actually I'm proud of git. I want to say this. The fact that I had to write git was accidental, but Linux, the design came from a great mind, and that great mind was not mine. I mean you have to give credit for the design of Linux to Kernighan and Ritchie and Thompson. I mean there's a reason I like Unix and I wanted to redo it. I do want to say that git is a design that is mine and unique, and I'm proud of the fact that I can damn well also do good design from scratch.

But some people hate git. Really *hate* it. They find it confusing and error prone and it makes them angry. Why are there such different views?

I think the reason some people hate git, is because they don't yet understand it. The reason I can say this without being patronizing is because I went through the same thing myself.

When I first started using git, I found it uncomfortable. I could see it was very powerful, but I sometimes got lost and stuck and had to Google for a set of magic commands to get me out of trouble. I once accidentally made a huge mess of our project's main repository by running a command I didn't understand. Git often made me feel stupid. It felt like a prototype race car with a badly designed dashboard that I couldn't control, and that was about to take me off the road, possibly at very high speed.

Then, one day, I read the git parable. The git parable is a little story about a developer trying to work out how to make a version control system. It gradually builds up from copying whole directories of files to something very much like git. I didn't understand it all right away, but as soon as I read that page, the light-bulb went on – I got git. At once I started

to feel comfortable. I knew that I could work out why git worked the way it did. I could see that it must be possible to do complicated and powerful things, and I could work out how to do them.

Reading the git parable took me about 45 minutes, but those 45 minutes changed me from an unhappy git user to someone who uses git often every day, but, happily, knowing that I have the right tool for the job.

So, my experience tells me that to use git – yes *use* git – you need to spend the short amount of time it takes to *understand* git. You don't believe me, or you think that I'm a strange kind of person not like you who probably likes writing their own operating systems. Not so - the insight I'm describing comes up over and over. From the git parable:

> Most people try to teach Git by demonstrating a few dozen commands and then yelling "tadaaaaa." I believe this method is flawed. Such a treatment may leave you with the ability to use Git to perform simple tasks, but the Git commands will still feel like magical incantations. Doing anything out of the ordinary will be terrifying. Until you understand the concepts upon which Git is built, you'll feel like a stranger in a foreign land.

From understanding git conceptually:

> When I first started using Git, I read plenty of tutorials, as well as the user manual. Though I picked up the basic usage patterns and commands, I never felt like I grasped what was going on "under the hood," so to speak. Frequently this resulted in cryptic error messages, caused by my random guessing at the right command to use at a given time. These difficulties worsened as I began to need more advanced (and less well documented) features.

Here's a quote from the pro git book by Scott Chacon. The git book is a standard reference that is hosted on the main git website.

> Chapter 9: Git Internals
>
> You may have skipped to this chapter from a previous chapter, or you may have gotten here after reading the rest of the book – in either case, this is where you'll go over the inner workings and implementation of Git. I found that learning this information was fundamentally important to understanding how useful and powerful Git is, but others have argued to me that it can be confusing and unnecessarily complex for beginners. Thus, I've made this discussion the last chapter in the book so you could read it early or later in your learning process. I leave it up to you to decide.

Of course it would be easier if you didn't need the deep shit – but if you want to spend the least time suffering from git, and the most time enjoying it, then you *do* need the deep shit. Luckily the deep shit isn't that deep. When you have got to the bottom of it, I'm betting that you'll agree that the alchemist has succeeded at last, and the – er – lead has finally turned into gold.

So – please – invest an hour and a half of your life to understand this stuff. Concentrate, go slowly, make sure you get it. In return for 90 minutes you will get many happy years for which git will appear in its true form, both beautiful and useful.

1.2 The one thing about git you really need to understand

git is not really a "Version Control System". It is better described as a "Content Management System", that turns out to be really good for version control.

I'll say that again. Git is a content management system. To quote from the root page of the git manual: "git - the stupid content tracker".

The reason that this is important is that git thinks in a very simple way about files and directories. You will ask git to keep snapshots of files in a directory, and it does just this; it stores snapshots of the files, so you can go back to them later.

Now you know that actual fact, you are ready to read *A curious tale*.

2 A curious tale

Here is a story where we develop a very simple system for storing file snapshots. We soon find it starts to look just like git.

2.1 The end of the story

To understand why git does what it does, we first need to think about what a content manager should do, and why we would want one.

If you've read the git parable (please do), then you'll recognize many of the ideas. Why? Because they are good ideas, worthy of re-use.

As in the git parable, we will try and design our own content manager, and then see what git has to say.

(If you don't mind reading some Python code, and more jokes, then also try my git foundation page).

While we are designing our own content management system, we will do a lot of stuff longhand, to show how things work. When we get to git, we will find it does these tasks for us.

2.2 The story begins

You are writing a breakthrough paper showing that you can explain how the brain works by careful processing of some interesting data. You've got the analysis script, the data file and a figure for the paper. These are all in a directory modestly named `nobel_prize`.

You can get this, the first draft, by downloading and unzipping `nobel_prize`.

Here's the current contents of our `nobel_prize` directory:

```
nobel_prize
├── clever_analysis.py [618B]
├── expensive_data.csv [244K]
└── fancy_figure.png [183K]
```

2.3 The dog ate my results

You've been working on this study for a while.

At first, you were very excited with the results. You ran the script, made the figure, and the figure looked good. That's the figure you currently have in `nobel_prize` directory. You took this figure to your advisor, Josephine. She was excited too. You get ready to publish in Science.

You've done a few changes to the script and figure since then. Today you finished cleaning up for the Science paper, and reran the analysis, and it doesn't look quite the same. You go to see Josephine. She says "It used to look better than that". That's what you think too. But:

- **Did it really look different before?**
- If it did, **what caused the change in the figure?**

2.4 Deja vu all over again

Given you are so clever and you have discovered how the brain works, it is really easy for you to leap in your time machine, and go back two weeks to start again.

What are you going to do differently this time?

2.5 Gitwards 1: make regular snapshots

You decide to make your own content management system. It's the simplest thing that could possibly work, so you call it the "Simple As Possible" system, or SAP for short.

Every time you finish doing some work on your paper, you make a snapshot of all the files for the paper.

The snapshot is a copy of all the files in the working directory.

First you make a directory called `working`, and move your files to that directory:

```
nobel_prize
└── working
    ├── clever_analysis.py [618B]
    ├── expensive_data.csv [244K]
    └── fancy_figure.png [183K]
```

When you've finished work for the day, you make a snapshot of the directory containing the files you are working on. The snapshot is just a copy of your working directory:

```
nobel_prize
├── working
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   └── fancy_figure.png [183K]
└── snapshot_1
    ├── clever_analysis.py [618B]
    ├── expensive_data.csv [244K]
    └── fancy_figure.png [183K]
```

You are going to do this every day you work on the project.

On the second day, you add your first draft of the paper, `nobel_prize.md`. You can download this ground-breaking work at `nobel_prize.md`.

```
nobel_prize
├── working
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   └── nobel_prize.md [730B]
└── snapshot_1
    ├── clever_analysis.py [618B]
    ├── expensive_data.csv [244K]
    └── fancy_figure.png [183K]
```

At the end of the day you make your second snapshot:

```
nobel_prize
├── working
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   └── nobel_prize.md [730B]
├── snapshot_2
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   └── nobel_prize.md [730B]
└── snapshot_1
    ├── clever_analysis.py [618B]
    ├── expensive_data.csv [244K]
    └── fancy_figure.png [183K]
```

On the third day, you did some edits to the analysis script, and refreshed the figure by running the script. You did a third snapshot.

```
nobel_prize
├── working
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
│   └── nobel_prize.md [730B]
├── snapshot_3
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
│   └── nobel_prize.md [730B]
├── snapshot_2
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   └── nobel_prize.md [730B]
└── snapshot_1
    ├── clever_analysis.py [618B]
    ├── expensive_data.csv [244K]
    └── fancy_figure.png [183K]
```

To make the directory listing more compact, I'll sometimes show only the number of files / directories in a subdirectory. For example, here's a listing of the three snapshots, but only showing the contents of the third snapshot:

```
nobel_prize
├── working
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
│   └── nobel_prize.md [730B]
├── snapshot_3
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
```

(continues on next page)

(continued from previous page)

```
    └── nobel_prize.md [730B]
├── snapshot_2
    (4 files)
└── snapshot_1
    (3 files)
```

Finally, on the fourth day, you make some more edits to the script, and you add some references for the paper.

```
nobel_prize
├── working
    ├── clever_analysis.py [752B]
    ├── expensive_data.csv [244K]
    ├── fancy_figure.png [251K]
    ├── nobel_prize.md [730B]
    └── references.bib [244B]
├── snapshot_4
    ├── clever_analysis.py [752B]
    ├── expensive_data.csv [244K]
    ├── fancy_figure.png [251K]
    ├── nobel_prize.md [730B]
    └── references.bib [244B]
├── snapshot_3
    (4 files)
├── snapshot_2
    (4 files)
└── snapshot_1
    (3 files)
```

You are ready for your fateful meeting with Josephine. Again she notices that the figure is different from the first time you showed her. This time you can go and look in `nobel_prize/snapshot_1` to see if the figure really is different. Then you can go through the snapshots to see where the figure changed.

You've already got a useful content management system, but you are going to make it better.

Note: We are already at the stage where we can define some terms that apply to our system and that will later apply to git:

Commit A completed snapshot. For example, `snapshot_1` contains one commit.

Working tree The files you are working on in `nobel_prize/working`.

2.6 Gitwards 2: reminding yourself of what you did

Your experience tracking down the change in the figure makes you think that it would be good to save a message with each snapshot (commit) to record the commit date and some text giving a summary of the changes you made. Next time you need to track down when and why something changed, you can look at the message to give yourself an idea of the changes in the commit. That might save you time when you want to narrow down where to look for problems.

So, for each commit, you write write a file called `message.txt`. The message for the first commit looks like this:

Contents of `snapshot_1/message.txt`

```
Date: April 1 2012, 14.30
Author: I. M. Awesome
Notes: First backup of my amazing idea
```

There is a similar `messsage.txt` file for each commit. For example, here's the message for the third commit:

Contents of `snapshot_3/message.txt`

```
Date: April 3 2012, 11.20
Author: I. M. Awesome
Notes: Add another fudge factor
```

This third message is useful because it gives you a hint that this was where you made the important change to the script and figure.

Note:

Commit message Information about a commit, including the author, date, time, and some information about the changes in the commit, compared to the previous commits.

2.7 Gitwards 3: breaking up work into chunks

Now you are used to your new system, you find that you like to break your changes up into self-contained chunks of work, each with their own commit, and a matching commit message. When you look back at your fourth commit, it looks like you included two separate chunks of work into the same commit. You've even confessed to this in your commit message:

Contents of `snapshot_4/message.txt`

```
Date: April 4 2012, 01.40
Author: I. M. Awesome
Notes: Change analysis and add references
```

You realize that you will often be in the situation where you have made several changes in the working tree, and you want to split those changes up into different commits, with their own commit messages. How can you adapt SAP to deal with that situation?

To help yourself think about this problem, you decide to scrap your last commit, and go back to the situation where your working tree has the changes, but the snapshots (commits) do not. All you have to do to get there, is delete the `snapshot_4` directory:

```
nobel_prize
├── working
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
```

(continues on next page)

(continued from previous page)

```
├── snapshot_3
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
│   ├── nobel_prize.md [730B]
│   └── message.txt [80B]
...
```

You still have your changes in the working tree. You have changed the analysis script and figure, and you have the new `references.bib` file.

You want to break these changes up into two separate commits:

- a commit with the changes to the analysis script and figure, but without the references;
- another commit to add the references.

To do this kind of thing, you are going to use a new directory called `staging`. The `staging` directory starts off with the files from the last commit. When you want to add some changes that will go into your next commit, you copy the changes from the working tree to the `staging` directory. You make the commit by copying the contents of `staging` to a new snapshot directory, and adding a commit message.

To get started, you make the new `staging` directory by copying the contents of the last commit (except the commit message):

```
nobel_prize
├── working
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── staging
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
│   └── nobel_prize.md [730B]
├── snapshot_3
│   ├── clever_analysis.py [716B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [239K]
│   ├── nobel_prize.md [730B]
│   └── message.txt [80B]
...
```

Call the `staging` directory – the **staging area**. Your new sequence for making a commit is:

- copy any changes for the next commit from the working tree to the staging area;
- make the commit by copying the contents of the staging area to a snapshot directory, and adding a commit message.

You are doing this by hand, but later git will make this much more automatic.

Now you are ready to make the first of your two new commits. You copy the changed analysis script and figure from the working tree to the staging area:

```
$ cp working/clever_analysis.py staging
$ cp working/fancy_figure.png staging
```

The staging directory (staging area) now contains the right files for the first of your two commits.

Next you make a commit by copying the staging area to `snapshot_4` and adding a message:

Contents of `snapshot_4/message.txt`

```
Date: April 4 2012, 01.40
Author: I. M. Awesome
Notes: Change parameters of analysis
```

This gives:

```
nobel_prize
├── working
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── staging
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   └── nobel_prize.md [730B]
├── snapshot_4
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   └── message.txt [85B]
...
```

To finish, you make the second of the two commits. Remember the sequence:

- copy any changes for the next commit from the working tree to the staging area;
- cake the commit by taking a snapshot of the staging area.

You copy the rest of the changes to the staging area:

```
$ cp working/references.bib staging
```

Finally, you do the commit by copying the contents of `staging` to a new directory `snapshot_5`, and adding a commit message:

Contents of `snapshot_5/message.txt`

```
Date: April 4 2012, 02.10
Author: I. M. Awesome
Notes: Add references
```

Now you have:

```
nobel_prize
├── working
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── staging
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── snapshot_5
│   ├── clever_analysis.py [752B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [251K]
│   ├── nobel_prize.md [730B]
│   ├── references.bib [244B]
│   └── message.txt [70B]
...
```

We can add a couple of new terms to our vocabulary:

Note:

Staging area Temporary area that contains the contents of the next commit. We copy changes from the working tree to the staging area to **stage** those changes. We make the new **commit** from the contents of the **staging area**.

2.8 Gitwards 4: getting files from previous commits

Remember that you found the figure had changed?

You also found that the problem was in the third commit.

Now you look back over the commits, you realize that your first draft of the analysis script was correct, and you decide to restore that.

To do that, you will **checkout** the script from the first commit (`snapshot_1`). You also want to checkout the generated figure.

Following our new standard staging workflow, that means:

- get the files you want from the old commit into the working directory, and the staging area;
- make a new commit from the staging area.

For our simple SAP system, that looks like this:

```
$ # Copy files from old commit to working tree
$ cp snapshot_1/clever_analysis.py working
$ cp snapshot_1/fancy_figure.png working
```

```
$ # Copy files from working tree to staging area
$ cp working/clever_analysis.py staging
$ cp working/fancy_figure.png staging
```

Then do the commit by copying `staging`, and add a message:

Contents of `snapshot_6/message.txt`

```
Date: April 5 2012, 18.40
Author: I. M. Awesome
Notes: Revert to original script & figure
```

This gives:

```
nobel_prize
├── working
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── staging
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── snapshot_6
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   ├── nobel_prize.md [730B]
│   ├── references.bib [244B]
│   └── message.txt [90B]
├── snapshot_5
│   (6 files)
├── snapshot_4
│   (5 files)
├── snapshot_3
│   (5 files)
├── snapshot_2
│   (5 files)
└── snapshot_1
    ├── clever_analysis.py [618B]
    ├── expensive_data.csv [244K]
    ├── fancy_figure.png [183K]
    └── message.txt [87B]
```

Note:

Checkout (a file) To **checkout** a file is to restore the copy of a file as stored in a particular commit.

2.9 Gitwards 5: two people working at the same time

One reason that git is so powerful is that it works very well when more than one person is working on the files in parallel.

Josephine is impressed with your SAP content management system, and wants to use it to make some edits to the paper. She takes a copy of your `nobel_prize` directory to put on her laptop.

She goes away for a conference.

While she is away, you do some work on the analysis script, and regenerate the figure, to make `shapshot_7`:

```
nobel_prize
├── working
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── staging
│   (5 files)
├── snapshot_7
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [730B]
│   ├── references.bib [244B]
│   └── message.txt [75B]
...
```

Meanwhile, Josephine decides to work on the paper. Following your procedure, she makes a commit herself.

What should Josephine's commit directory be called?

She could call it `snapshot_7`, but then, when she gets back to the lab, and gives you her `nobel_prize` directory, her copy of `nobel_prize` and yours will both have a `snapshot_7` directory, but they will be different. It would be easy to copy Josephine's directory over yours or yours over Josephine's, and lose the work.

For the moment, you decide that Josephine will attach her name to the commit directory, to make it clear this is her snapshot. So, she makes her commit into the directory `snapshot_7_josephine`. When she comes back from the conference, you copy her `snapshot_7_josephine` into your `nobel_prize` directory:

```
nobel_prize
├── working
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [730B]
│   └── references.bib [244B]
├── staging
│   (5 files)
├── snapshot_7
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [730B]
```

(continues on next page)

(continued from previous page)

```
│   ├── references.bib [244B]
│   └── message.txt [75B]
├── snapshot_7_josephine
│   ├── clever_analysis.py [618B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [183K]
│   ├── nobel_prize.md [979B]
│   ├── references.bib [244B]
│   └── message.txt [80B]
...
```

After the copy, you still have your own copy of the working tree, without Josephine's changes to the paper. You want to combine your changes with her changes. To do this you do a **merge** by copying her changes to the paper into the working directory.

```
$ # Get Josephine's changes to the paper
$ cp snapshot_7_josephine/nobel_prize.md working
```

Now you do a commit with these merged changes, by copying them into the staging area, and thence to `snapshot_8`, with a suitable message:

```
nobel_prize
├── working
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [979B]
│   └── references.bib [244B]
├── staging
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [979B]
│   └── references.bib [244B]
├── snapshot_8
│   ├── clever_analysis.py [692B]
│   ├── expensive_data.csv [244K]
│   ├── fancy_figure.png [279K]
│   ├── nobel_prize.md [979B]
│   ├── references.bib [244B]
│   └── message.txt [82B]
...
```

This new commit is the result of a merge, and therefore it is a **merge commit**.

Note:

Merge To make a new **merge commit** by combining changes from two (or more) commits.

2.10 Gitwards 6: how should you name your commit directories?

You like your new system, and so does Josephine, but you don't much like your solution of adding Josephine's name to the commit directory – as in `snapshot_7_josephine`. There might be lots of people working on this paper. With your naming system, you have to give out a unique name to each person working on `nobel_prize`. As you think about this problem, you begin to realize that what you want is a system for giving each commit directory a unique name, that comes from the contents of the commit. This is where you starting thinking about **hashes**.

2.11 A diversion on cryptographic hashes

This section describes "Cryptographic hashes". These will give us an excellent way to name our snapshots. Later we will see that they are central to the way that git works.

See : Wikipedia on hash functions.

A *hash* is the result of running a *hash function* over a block of data. The hash is a fixed length string that is the characteristic *fingerprint* of that exact block of data. One common hash function is called SHA1. Let's run this via the command line:

```
$ # Make a file with a single line of text
$ echo "git is a rude word in UK English" > git_is_rude
$ # Show the SHA1 hash
$ shasum git_is_rude
30ad6c360a692c1fe66335bb00d00e0528346be5  git_is_rude
```

Not too exciting so far. However, the rather magical nature of this string is not yet apparent. This SHA1 hash is a *cryptographic* hash because:

- the hash value is (almost) unique to this exact file contents, and
- it is (almost) impossible to find some other file contents with the same hash.

By "almost impossible" I mean that finding a file with the same hash is roughly the same level of difficulty as trying something like 16^{40} different files (where 16 is the number of different hexadecimal digits, and 40 is the length of the SHA1 string).

In other words, there is no practical way for you to find another file with different contents that will give the same hash.

For example, a tiny change in the string makes the hash completely different. Here I've just added a full stop at the end:

```
$ echo "git is a rude word in UK English." > git_is_rude_stop
$ shasum git_is_rude_stop
23d97b00299c250b43049be67c97cf2e5ffc2428  git_is_rude_stop
```

So, if you give me some data, and I calculate the SHA1 hash value, and it comes out as `30ad6c360a692c1fe66335bb00d00e0528346be5`, then I can be very sure that the data you gave me was exactly the ASCII string "git is a rude word in UK English".

2.12 Gitwards 7: naming commits from hashes

Now you have hashing under your belt, maybe it would be a good way of making a unique name for the commits. You could take the SHA1 hash for the `message.txt` for each commit, and use that SHA1 hash as the name for the commit directory. Because each message has the date and time and author and notes, it's very unlikely that any two `message.txt` files will be the same. Here are the hashes for the current `message.txt` files:

```
$ # Show the SHA1 hash values for each message.txt
$ shasum snapshot*/message.txt
99b52473039acea4427e13e42b96c78776e2baf5  snapshot_1/message.txt
d396475cc691c8ac7ba7a318726f220c924cf60b  snapshot_2/message.txt
6a04028960ead9e8db180dd6ebc108e1fc52891d  snapshot_3/message.txt
1bae5edb2785ae73353ae8a691cdb45d182db0c6  snapshot_4/message.txt
2bef6ebeef1dee1610f6581e42fa1a74045042c2  snapshot_5/message.txt
5709c1f836244dc039f05e5d3b0c31267ee785c5  snapshot_6/message.txt
62f0c997a4ddbfe07d6d2524d23883a3ac6ac051  snapshot_7/message.txt
4063178db6bf986c679bfc2c4dde01092146d197  snapshot_7_josephine/message.txt
df3e787c2a89ac991b1c89b39fb014c1f049a939  snapshot_8/message.txt
```

For example you could rename the `snapshot_1` directory to `99b52473039acea4427e13e42b96c78776e2baf5`, then rename `snapshot_2` to `d396475cc691c8ac7ba7a318726f220c924cf60b` and so on.

```
nobel_prize
├── working
│   (5 files)
├── staging
│   (5 files)
├── df3e787c2a89ac991b1c89b39fb014c1f049a939
│   (6 files)
├── 4063178db6bf986c679bfc2c4dde01092146d197
│   (6 files)
├── 62f0c997a4ddbfe07d6d2524d23883a3ac6ac051
│   (6 files)
├── 5709c1f836244dc039f05e5d3b0c31267ee785c5
│   (6 files)
├── 2bef6ebeef1dee1610f6581e42fa1a74045042c2
│   (6 files)
├── 1bae5edb2785ae73353ae8a691cdb45d182db0c6
│   (5 files)
├── 6a04028960ead9e8db180dd6ebc108e1fc52891d
│   (5 files)
├── d396475cc691c8ac7ba7a318726f220c924cf60b
│   (5 files)
└── 99b52473039acea4427e13e42b96c78776e2baf5
    (4 files)
```

The problem you have now is that the directory names no longer tell you the sequence of the commits, so you can't tell that `snapshot_2` (now `d396475cc691c8ac7ba7a318726f220c924cf60b`) follows `snapshot_1` (now `99b52473039acea4427e13e42b96c78776e2baf5`).

OK – you scratch the renaming for now while you have a rethink.

```
nobel_prize
├── working
```

(continues on next page)

(continued from previous page)

```
│   (5 files)
├── staging
│   (5 files)
├── snapshot_8
│   (6 files)
├── snapshot_7
│   (6 files)
├── snapshot_7_josephine
│   (6 files)
├── snapshot_6
│   (6 files)
├── snapshot_5
│   (6 files)
├── snapshot_4
│   (5 files)
├── snapshot_3
│   (5 files)
├── snapshot_2
│   (5 files)
└── snapshot_1
    (4 files)
```

You still want to rename the commit directories, from the `message.txt` hashes, but you need a way to store the sequence of commits, after you have done this.

After some thought, you come up with a quite brilliant idea. Each `message.txt` will point back to the previous commit in the sequence. You add a new field to `messsage.txt` called `Parents`. `snapshot_1/message.txt` stays the same, because it has no parents:

```
$ cat snapshot_1/message.txt
Date: April 1 2012, 14.30
Author: I. M. Awesome
Notes: First backup of my amazing idea
```

`snapshot_2/message.txt` does change, because it now points back to `snapshot_1`. But, you're going to rename the snapshot directories, so you want `snapshot_2/message.txt` to point back to the hash for `snapshot_1/message.txt`, which you know is `99b52473039acea4427e13e42b96c78776e2baf5`:

```
$ cat snapshot_2/message.txt
Date: April 2 2012, 18.03
Author: I. M. Awesome
Notes: Add first draft of paper
Parents: 99b52473039acea4427e13e42b96c78776e2baf5
```

Now we've changed the contents and therefore the hash for `snapshot_2/message.txt`. The hash was `d396475cc691c8ac7ba7a318726f220c924cf60b`, but now it is:

```
$ shasum snapshot_2/message.txt
9bf4cdd8847feeb49fe94cb96298acf7986a587d  snapshot_2/message.txt
```

You keep doing this procedure, for all the commits, modifying `message.txt` and recalculating the hash, until you come to `snapshot_8`, the merge commit. This commit is the result of merging two commits: `snapshot_7` and `snapshot_7_josephine`. You can record this information by putting *two* parents into the `Parents` field of

`snapshot_8/message.txt`, being the new hashes for `snapshot_7/message.txt` and `snapshot_7_josephine/message.txt`:

```
$ shasum snapshot_7/message.txt
bee09a3b1c11456c6e1bc6784319954f40fef24b  snapshot_7/message.txt
```

```
$ shasum snapshot_7_josephine/message.txt
905376d349af193c367217712bc231186f17fca7  snapshot_7_josephine/message.txt
```

```
$ cat snapshot_8/message.txt
Date: April 7 2012, 15.03
Author: I. M. Awesome
Notes: Merged Josephine's changes
Parents: bee09a3b1c11456c6e1bc6784319954f40fef24b␣
↪905376d349af193c367217712bc231186f17fca7
```

With the new `Parents` field, you have new hashes for all the `message.txt` files, except `snapshot_1` (that has no parent):

```
$ shasum snapshot_*/message.txt
99b52473039acea4427e13e42b96c78776e2baf5  snapshot_1/message.txt
9bf4cdd8847feeb49fe94cb96298acf7986a587d  snapshot_2/message.txt
d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682  snapshot_3/message.txt
00d03e9d1bf4ebaea380da3c62e9226189e39ff4  snapshot_4/message.txt
c203516a49d48389826b041f4d7bd2d39bdf4a57  snapshot_5/message.txt
9920fff7b55370a2c4c7566011de8aa2a263171f  snapshot_6/message.txt
bee09a3b1c11456c6e1bc6784319954f40fef24b  snapshot_7/message.txt
905376d349af193c367217712bc231186f17fca7  snapshot_7_josephine/message.txt
bd9dc5fb9aacf208536fad1fad22696c9a79e277  snapshot_8/message.txt
```

You can now rename your snapshot directories with the hash values, safe in the knowledge that the `message.txt` files have the information about the commit sequence.

```
nobel_prize
├── working
│   (5 files)
├── staging
│   (5 files)
├── bd9dc5fb9aacf208536fad1fad22696c9a79e277
│   (6 files)
├── 905376d349af193c367217712bc231186f17fca7
│   (6 files)
├── bee09a3b1c11456c6e1bc6784319954f40fef24b
│   (6 files)
├── 9920fff7b55370a2c4c7566011de8aa2a263171f
│   (6 files)
├── c203516a49d48389826b041f4d7bd2d39bdf4a57
│   (6 files)
├── 00d03e9d1bf4ebaea380da3c62e9226189e39ff4
│   (5 files)
├── d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682
│   (5 files)
├── 9bf4cdd8847feeb49fe94cb96298acf7986a587d
```

(continues on next page)

(continued from previous page)

```
    (5 files)
└── 99b52473039acea4427e13e42b96c78776e2baf5
    (4 files)
```

Now the commit directories are hash names, it is harder to see which commit is which, so here's the directory listing where the commit directories have a label from the `Notes:` field in `message.txt`:

```
nobel_prize
├── working
    (5 files)
├── staging
    (5 files)
├── bd9dc5fb9aacf208536fad1fad22696c9a79e277    "Merged Josephine's changes"
    (6 files)
├── 905376d349af193c367217712bc231186f17fca7    "Expand the introduction"
    (6 files)
├── bee09a3b1c11456c6e1bc6784319954f40fef24b    "More fun with fudge"
    (6 files)
├── 9920fff7b55370a2c4c7566011de8aa2a263171f    "Revert to original script & figure"
    (6 files)
├── c203516a49d48389826b041f4d7bd2d39bdf4a57    "Add references"
    (6 files)
├── 00d03e9d1bf4ebaea380da3c62e9226189e39ff4    "Change parameters of analysis"
    (5 files)
├── d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682    "Add another fudge factor"
    (5 files)
├── 9bf4cdd8847feeb49fe94cb96298acf7986a587d    "Add first draft of paper"
    (5 files)
└── 99b52473039acea4427e13e42b96c78776e2baf5    "First backup of my amazing idea"
    (4 files)
```

Note:

Commit hash The hash value for the file containing the **commit message**.

2.13 Gitwards 8: the development history is a graph

The commits are linked by the "Parents" field in the `message.txt` file. We can think of the commits in a graph, where the commits are the nodes, and the links between the nodes come from the hashes in the "Parents" field.

2.14 Gitwards 9: saving space with file hashes

While you've been working on your system, you've noticed that your snapshots are not efficient on disk space. For example, every commit / snapshot has an identical copy of the data `expensive_data.csv`. If you had bigger files or a longer development history, this could be a problem.

Likewise, `fancy_figure.png` and `clever_analysis.py` are the same for the first two commits, and then again when you reverted to that copy in `snapshot_6` (that is now commit `9920fff7b55370a2c4c7566011de8aa2a263171f`).

You can show these files are the same by checking their hash strings. If their hash strings are different, the files must be different. All copies of `expensive_data.csv` have the same hash, and are therefore identical:

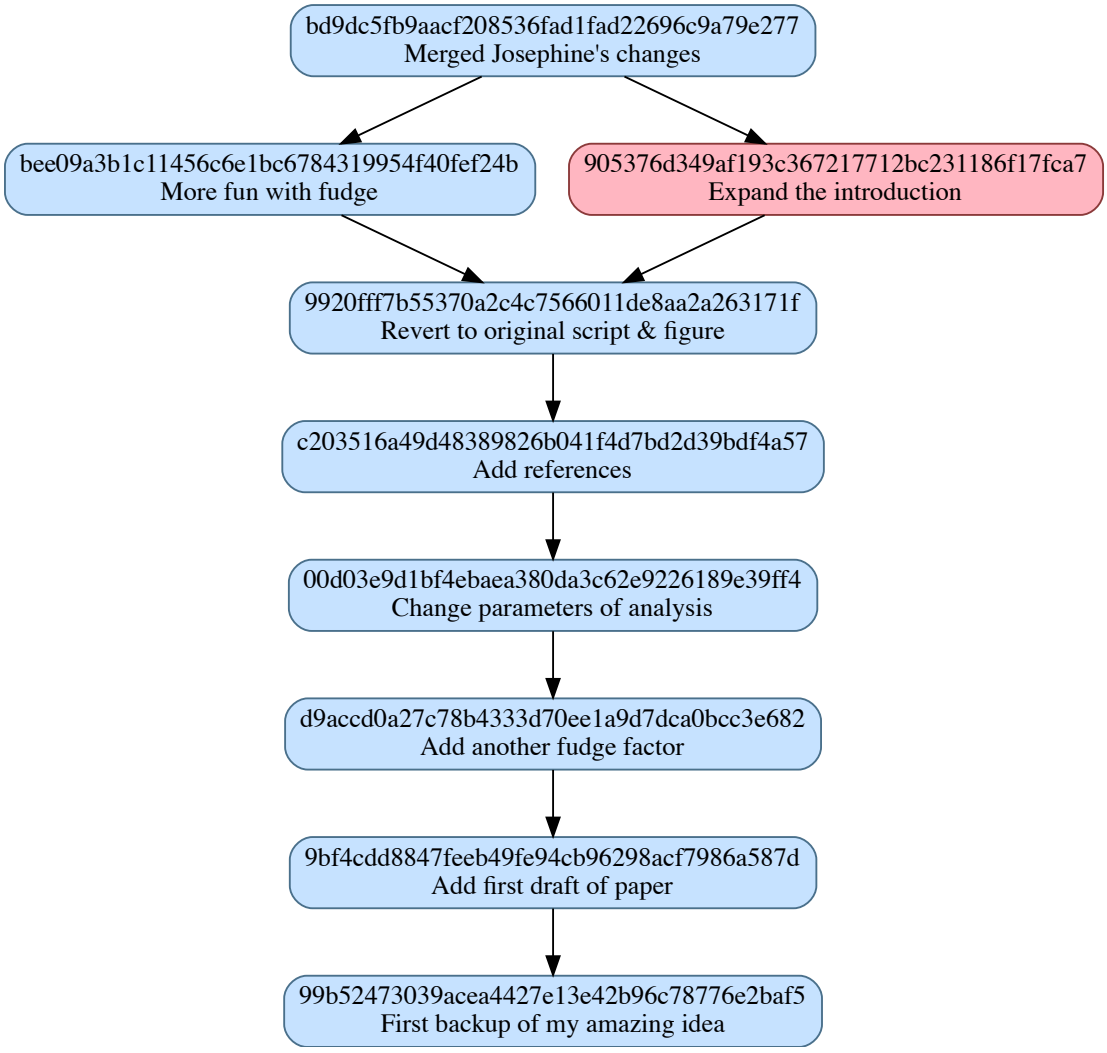

Fig. 1: Graph of development history for your SAP content management system. The most recent commit is at the top, the first commit is at the bottom. Your commits are in blue, Josephine's are in pink. Each commit label has the hash for the commit message, and the note in the `message.txt` file.

```
$ shasum */expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  00d03e9d1bf4ebaea380da3c62e9226189e39ff4/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  905376d349af193c367217712bc231186f17fca7/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  9920fff7b55370a2c4c7566011de8aa2a263171f/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  99b52473039acea4427e13e42b96c78776e2baf5/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  9bf4cdd8847feeb49fe94cb96298acf7986a587d/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  bd9dc5fb9aacf208536fad1fad22696c9a79e277/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  bee09a3b1c11456c6e1bc6784319954f40fef24b/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  c203516a49d48389826b041f4d7bd2d39bdf4a57/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682/
↪expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  staging/expensive_data.csv
bf2c7158afc60b244cba860fb0b2c04bb6481daf  working/expensive_data.csv
```

`fancy_figure.png` is the same for the first two commits, changes for the third commit, and reverts back to the same contents at the 6th commit:

```
$ # First commit
$ shasum 99b52473039acea4427e13e42b96c78776e2baf5/fancy_figure.png
e8fe49b1faf23b428865d0315c4763c6ec459897  99b52473039acea4427e13e42b96c78776e2baf5/fancy_
↪figure.png
```

```
$ # Second commit
$ shasum 9bf4cdd8847feeb49fe94cb96298acf7986a587d/fancy_figure.png
e8fe49b1faf23b428865d0315c4763c6ec459897  9bf4cdd8847feeb49fe94cb96298acf7986a587d/fancy_
↪figure.png
```

```
$ # Third commit
$ shasum d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682/fancy_figure.png
863cf24c03c6ab1eb33c1022d6fc8f7a2876875a  d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682/fancy_
↪figure.png
```

```
$ # Sixth commit
$ shasum 9920fff7b55370a2c4c7566011de8aa2a263171f/fancy_figure.png
e8fe49b1faf23b428865d0315c4763c6ec459897  9920fff7b55370a2c4c7566011de8aa2a263171f/fancy_
↪figure.png
```

You wonder if there is a way to store each unique version of the file just once, and make the commits point to the matching version.

First you make a new directory to store files generated from your commits:

```
$ mkdir repo
```

Next you make a sub-directory to store the unique copies of the files in commits:

```
$ mkdir repo/objects
```

You play with the idea of calling these unique versions something like `repo/objects/fancy_figure.png.v1`, `repo/objects/fancy_figure.png.v2` and so on. You would then need something like a text file called `directory_listing.txt` in the first commit directory to say that the file `fancy_figure.png` for this commit is available at `repo/objects/fancy_figure.png.v1`. This could be something like:

```
# directory_listing.txt in first commit
fancy_figure.png -> repo/objects/fancy_figure.png.v1
```

`directory_listing.txt` for the second commit would point to the same file, but the third commit would have something like:

```
# directory_listing.txt in third commit
fancy_figure.png -> repo/objects/fancy_figure.png.v2
```

You quickly realize this is going to get messy when you are working with other people, because you may store `repo/objects/fancy_figure.png.v3` while Josephine is also working on the figure, and is storing her own `repo/objects/fancy_figure.png.v3`. You need a unique file name for each version of the file.

Now you have your second quite brilliant hashing idea. Why not use the **hash** of the file to make a unique file name?

For example, here are the hash values for the files in the first commit:

```
$ shasum 99b52473039acea4427e13e42b96c78776e2baf5/*
c25a0bd91a7e8eb574b17296a25111233eac94b5  99b52473039acea4427e13e42b96c78776e2baf5/
↪clever_analysis.py
bf2c7158afc60b244cba860fb0b2c04bb6481daf  99b52473039acea4427e13e42b96c78776e2baf5/
↪expensive_data.csv
e8fe49b1faf23b428865d0315c4763c6ec459897  99b52473039acea4427e13e42b96c78776e2baf5/fancy_
↪figure.png
99b52473039acea4427e13e42b96c78776e2baf5  99b52473039acea4427e13e42b96c78776e2baf5/
↪message.txt
```

To store the unique copies, you copy each file in the first commit to `repo/objects` with a unique file name. **The file name is the hash of the file contents**. For example, the hash for `fancy_figure.png` is `e8fe49b1faf23b428865d0315c4763c6ec459897`. So, you do:

```
$ cp 99b52473039acea4427e13e42b96c78776e2baf5/fancy_figure.png repo/objects/
↪e8fe49b1faf23b428865d0315c4763c6ec459897
```

The hash values for `clever_analysis.py` and `expensive_data.csv` are `c25a0bd91a7e8eb574b17296a25111233eac94b5` and `bf2c7158afc60b244cba860fb0b2c04bb6481daf` respectively, so:

```
$ cp 99b52473039acea4427e13e42b96c78776e2baf5/clever_analysis.py repo/objects/
↪c25a0bd91a7e8eb574b17296a25111233eac94b5
$ cp 99b52473039acea4427e13e42b96c78776e2baf5/expensive_data.csv repo/objects/
↪bf2c7158afc60b244cba860fb0b2c04bb6481daf
```

These hash values become the `directory_listing.txt` for the first commit:

```
$ cat 99b52473039acea4427e13e42b96c78776e2baf5/directory_listing.txt
c25a0bd91a7e8eb574b17296a25111233eac94b5  clever_analysis.py
```

(continues on next page)

(continued from previous page)

```
bf2c7158afc60b244cba860fb0b2c04bb6481daf  expensive_data.csv
e8fe49b1faf23b428865d0315c4763c6ec459897  fancy_figure.png
```

Finally, you can delete `fancy_figure.png`, `clever_analysis.py` and `expensive_data.csv` in the first commit directory, because you have them backed up in `repo/objects`.

So far you haven't gained anything much except some odd-looking filenames. The payoff comes when you apply the same procedure to the second commit. Here are the hashes for the files in the second commit:

```
$ shasum 9bf4cdd8847feeb49fe94cb96298acf7986a587d/*
c25a0bd91a7e8eb574b17296a25111233eac94b5  9bf4cdd8847feeb49fe94cb96298acf7986a587d/
↪clever_analysis.py
bf2c7158afc60b244cba860fb0b2c04bb6481daf  9bf4cdd8847feeb49fe94cb96298acf7986a587d/
↪expensive_data.csv
e8fe49b1faf23b428865d0315c4763c6ec459897  9bf4cdd8847feeb49fe94cb96298acf7986a587d/fancy_
↪figure.png
9bf4cdd8847feeb49fe94cb96298acf7986a587d  9bf4cdd8847feeb49fe94cb96298acf7986a587d/
↪message.txt
47eef829586738a0855060dc83f8e0781fbcac2c  9bf4cdd8847feeb49fe94cb96298acf7986a587d/nobel_
↪prize.md
```

Remember that, in the second commit, all you did was add the first draft of the paper as `nobel_prize.md`. So, all the other files in the second commit (apart from `message.txt` that you are not storing) are the same as for the first commit, and therefore have the same hash. You already have these files backed up in `repo/objects` so all you need to do is point `directory_listing.txt` at the original copies in `repo/objects`.

For example, the hash for `fancy_figure.png` in the second commit is `e8fe49b1faf23b428865d0315c4763c6ec459897`. When you are storing the files for the second commit in `repo/objects`, you notice that you already have a file named `e8fe49b1faf23b428865d0315c4763c6ec459897` in `repo/objects`, so you do not copy it a second time. By checking the hashes for each file in the commit, you find that the only file you are missing is the new file `nobel_prize.md`. This has hash `47eef829586738a0855060dc83f8e0781fbcac2c`, so you do a single copy to `repo/objects`:

```
$ # Only one copy needed to store files in second commit
$ cp 9bf4cdd8847feeb49fe94cb96298acf7986a587d/nobel_prize.md repo/objects/
↪47eef829586738a0855060dc83f8e0781fbcac2c
```

As before, you can make `directory_listing.txt` for the second commit by recording the hashes of the files:

```
$ cat 9bf4cdd8847feeb49fe94cb96298acf7986a587d/directory_listing.txt
c25a0bd91a7e8eb574b17296a25111233eac94b5  clever_analysis.py
bf2c7158afc60b244cba860fb0b2c04bb6481daf  expensive_data.csv
e8fe49b1faf23b428865d0315c4763c6ec459897  fancy_figure.png
47eef829586738a0855060dc83f8e0781fbcac2c  nobel_prize.md
```

Before you start this procedure of moving the unique copies into `repo/objects`, your whole `nobel_prize` directory is size:

```
$ # Size of the contents of nobel_prize before moving to repo/objects
$ du -hs .
5.4M        .
```

When you run the procedure above on every commit, moving files to `repo/objects`, you have this:

```
nobel_prize
├── working
│   (5 files)
├── staging
│   (5 files)
├── repo
│   └── objects
│       ├── 18e92be4df9036826fc983c42e2e4fb5c6fed1cb [692B]
│       ├── 27e85e8f524a26a3a77f1c7ea631d6b2ad2b018a [251K]
│       ├── 47eef829586738a0855060dc83f8e0781fbcac2c [730B]
│       ├── 4e3c43a5c0d530fa0ba4305b3c52a9f7a4cb9140 [244B]
│       ├── 5840cded1e61d0a22c0e9f7d62de8c43ff47a402 [716B]
│       ├── 71892a8d170f113b778a9458cd6930abb387328f [238B]
│       ├── 80bf412d6273626674658862caebb131c2df8ddc [979B]
│       ├── 815bbcf64edacb6791cc2ebdc3afeb09797c9347 [181B]
│       ├── 863cf24c03c6ab1eb33c1022d6fc8f7a2876875a [239K]
│       ├── bf2c7158afc60b244cba860fb0b2c04bb6481daf [244K]
│       ├── c25a0bd91a7e8eb574b17296a25111233eac94b5 [618B]
│       ├── e8fe49b1faf23b428865d0315c4763c6ec459897 [183K]
│       ├── efb5308cc0f74a2515ce668adf237b28527fd2bc [279K]
│       └── f2e5e8c8122533fbc1da0684abe7e43471fe1674 [752B]
├── bd9dc5fb9aacf208536fad1fad22696c9a79e277    "Merged Josephine's changes"
│   ├── directory_listing.txt [290B]
│   └── message.txt [173B]
├── 905376d349af193c367217712bc231186f17fca7    "Expand the introduction"
│   ├── directory_listing.txt [290B]
│   └── message.txt [130B]
├── bee09a3b1c11456c6e1bc6784319954f40fef24b    "More fun with fudge"
│   ├── directory_listing.txt [290B]
│   └── message.txt [125B]
├── 9920fff7b55370a2c4c7566011de8aa2a263171f    "Revert to original script & figure"
│   ├── directory_listing.txt [290B]
│   └── message.txt [140B]
├── c203516a49d48389826b041f4d7bd2d39bdf4a57    "Add references"
│   ├── directory_listing.txt [290B]
│   └── message.txt [120B]
├── 00d03e9d1bf4ebaea380da3c62e9226189e39ff4    "Change parameters of analysis"
│   ├── directory_listing.txt [234B]
│   └── message.txt [135B]
├── d9accd0a27c78b4333d70ee1a9d7dca0bcc3e682    "Add another fudge factor"
│   ├── directory_listing.txt [234B]
│   └── message.txt [130B]
├── 9bf4cdd8847feeb49fe94cb96298acf7986a587d    "Add first draft of paper"
│   ├── directory_listing.txt [297B]
│   └── message.txt [130B]
└── 99b52473039acea4427e13e42b96c78776e2baf5    "First backup of my amazing idea"
    ├── directory_listing.txt [241B]
    └── message.txt [87B]
```

The whole `nobel_prize` directory is now smaller because you have no duplicated files:

```
$ # Size of the contents of nobel_prize after moving to repo/objects
$ du -hs .
```

(continues on next page)

(continued from previous page)

```
2.3M        .
```

The advantage in size gets larger as your system grows, and you have more duplicated files.

2.15 Gitwards 10: making the commits unique

Up in *Gitwards 7: naming commits from hashes* you used the hash of `message.txt` as a nearly unique directory name for the commit. Your thinking was that it was very unlikely that any two commits would have the same author, date, time, and note. You have since added the `Parents` field to `message.txt` to make it even more unlikely. But – it could still happen. You might be careless and make another commit very quickly after the previous, and without a note. You could even point back to the same parent.

You would like to be even more confident that the commit message is unique to the commit, including the contents of the files in the commit.

You now have a way of doing this. The `directory_listing.txt` files contain a list of hashes and corresponding file names for this commit (snapshot). For example, here is `directory_listing.txt` for the first commit:

```
$ cat 99b52473039acea4427e13e42b96c78776e2baf5/directory_listing.txt
c25a0bd91a7e8eb574b17296a25111233eac94b5 clever_analysis.py
815bbcf64edacb6791cc2ebdc3afeb09797c9347 directory_listing.txt
bf2c7158afc60b244cba860fb0b2c04bb6481daf expensive_data.csv
e8fe49b1faf23b428865d0315c4763c6ec459897 fancy_figure.png
```

The contents of this file are (very nearly) unique to the contents of the files in the snapshot. If any of the files changed, then the hash of the file would change and the corresponding line in `directory_listing.txt` would change. If you renamed the file, the name of the file would change and the corresponding line in `directory_listing.txt` would change.

Now you know what to do. You take a hash of the `directory_listing.txt` file:

```
$ shasum 99b52473039acea4427e13e42b96c78776e2baf5/directory_listing.txt
85d49d8555aeb254d1d4acd377034772944a35fe  99b52473039acea4427e13e42b96c78776e2baf5/
↪directory_listing.txt
```

You put this has into a new field in `message.txt` called `Directory hash:`:

```
Date: April 1 2012, 14.30
Author: I. M. Awesome
Notes: First backup of my amazing idea
Directory hash: 85d49d8555aeb254d1d4acd377034772944a35fe
```

Now, if any file in the commit changes, `directory_listing.txt` will change, and so its hash will change, and so `message.txt` will change.

Now you've added the `Directory hash` field to `messsage.txt` you have also changed the hash values of the `message.txt` files. Because you've changed the hashes of the `message.txt` files, you go back through your commits updating the parent hashes to the new ones, and renaming the commit directories with the new hashes. You end up with this:

```
nobel_prize
├── working
│   (5 files)
├── staging
```

(continues on next page)

(continued from previous page)

```
│   (5 files)
├── repo
│   (1 directory)
├── f46773cff294d07a331a6aa0ee32b3615cb009b8    "Merged Josephine's changes"
│   (2 files)
├── 505228057403295fdc16888bed49515a7c1d80a1    "Expand the introduction"
│   (2 files)
├── 5bdbf9b8ac9e998a95495e2d7ae370242c267c52    "More fun with fudge"
│   (2 files)
├── 41234bea0657aecaa234cee2b146051f90062078    "Revert to original script & figure"
│   (2 files)
├── c1072adf649f3ec439e6925af01251b9ccd91410    "Add references"
│   (2 files)
├── 917fb4cfe6c5d13bd4efbac1b54b26ae2967c839    "Change parameters of analysis"
│   (2 files)
├── 2eb0b1ea04d70129e4a4e219e8d51e10effdb8fe    "Add another fudge factor"
│   (2 files)
├── 117e7de8e27a54104f7a08d0f75b9ca214eaccb3    "Add first draft of paper"
│   (2 files)
└── 9c0b782981cc2ee322ec4b595dc472799f8808bd    "First backup of my amazing idea"
    (2 files)
```

With your new system, if any two commits have the same `message.txt` then they also have the same date, author, note, parents and file contents. They are therefore exactly the same commit.

Note: The commit message is unique to the contents of the files in the snapshot (because of the directory hash) and unique to its previous history (because of the parent hash(es)).

2.16 Gitwards 11: away with the snapshot directories

You are reflecting on your idea about hashing the directory listing, and your eye falls idly on the current directory tree of `nobel_prize`:

```
nobel_prize
├── working
│   (5 files)
├── staging
│   (5 files)
├── repo
│   └── objects
│       (14 files)
├── f46773cff294d07a331a6aa0ee32b3615cb009b8    "Merged Josephine's changes"
│   ├── directory_listing.txt [290B]
│   └── message.txt [230B]
├── 505228057403295fdc16888bed49515a7c1d80a1    "Expand the introduction"
│   ├── directory_listing.txt [290B]
│   └── message.txt [187B]
├── 5bdbf9b8ac9e998a95495e2d7ae370242c267c52    "More fun with fudge"
│   ├── directory_listing.txt [290B]
│   └── message.txt [182B]
```

(continues on next page)

(continued from previous page)

```
├── 41234bea0657aecaa234cee2b146051f90062078    "Revert to original script & figure"
│   ├── directory_listing.txt [290B]
│   └── message.txt [197B]
├── c1072adf649f3ec439e6925af01251b9ccd91410    "Add references"
│   ├── directory_listing.txt [290B]
│   └── message.txt [177B]
├── 917fb4cfe6c5d13bd4efbac1b54b26ae2967c839    "Change parameters of analysis"
│   ├── directory_listing.txt [234B]
│   └── message.txt [192B]
├── 2eb0b1ea04d70129e4a4e219e8d51e10effdb8fe    "Add another fudge factor"
│   ├── directory_listing.txt [234B]
│   └── message.txt [187B]
├── 117e7de8e27a54104f7a08d0f75b9ca214eaccb3    "Add first draft of paper"
│   ├── directory_listing.txt [297B]
│   └── message.txt [187B]
└── 9c0b782981cc2ee322ec4b595dc472799f8808bd    "First backup of my amazing idea"
    ├── directory_listing.txt [241B]
    └── message.txt [144B]
```

It occurs to you that you can move the `directory_listing.txt` and `message.txt` files into your `repo/objects` directory. When you have done that, you can get rid of the commit directories entirely.

First you take the hash of each `directory_listing.txt` and move it into the `repo/objects` directory as you did for the other files:

```
$ shasum 9c0b782981cc2ee322ec4b595dc472799f8808bd/directory_listing.txt
85d49d8555aeb254d1d4acd377034772944a35fe  9c0b782981cc2ee322ec4b595dc472799f8808bd/
↪directory_listing.txt
```

```
$ cp 9c0b782981cc2ee322ec4b595dc472799f8808bd/directory_listing.txt repo/objects/
↪85d49d8555aeb254d1d4acd377034772944a35fe
```

Then you do the same for the `message.txt` file:

```
$ cp 9c0b782981cc2ee322ec4b595dc472799f8808bd/message.txt repo/objects/
↪9c0b782981cc2ee322ec4b595dc472799f8808bd
```

There are 9 commits, so there are 9 x 2 new files with hash filenames in `repo/objects` (a hashed copy of `directory_listing.txt` and `message.txt` for each commit).

Now you don't need the snapshot directories at all, because the hashed files in `repo/objects` have all the information about the snapshots.

```
nobel_prize
├── working
│   (5 files)
├── staging
│   (5 files)
└── repo
    └── objects
        (32 files)
```

Note: In git as in your SAP content management system, a **repository directory** stores all the data from the snapshots.

In your case that directory is `repo`. For git, it will be a directory called `.git`.

2.17 Gitwards 12: where am I?

You have one last problem to face – where is your latest commit?

When your snapshot directory names had numbers, like `snapshot_8`, you could use the numbers to find the most recent commit. Now all you have is a directory called `repo/objects` with unhelpful file names made from hashes. Which of these files has your latest commit?

You could write down the latest commit hash on a piece of paper, after you make the commit, but this sounds like a job better done by a computer.

So, when you make a new commit, you store the hash for that commit in a file called `repo/my_bookmark`. It is a text file with the hash string as contents. Your last commit was `f46773cff294d07a331a6aa0ee32b3615cb009b8`, so `repo/my_bookmark` has contents:

```
$ cat repo/my_bookmark
f46773cff294d07a331a6aa0ee32b3615cb009b8
```

You can imagine that, when Josephine is working on the same set of files, she might want her own bookmark, maybe in a file called `josephines-bookmark`.

Note: You keep track of the latest commit in a particular sequence by storing the latest **commit hash** in a bookmark file. In git this bookmark is called a **branch**.

2.18 You are on the on-ramp

You now know all the main ideas in git. Follow me then to *Curious git* to see these ideas come to life in your actual git.

3 Curious git

In *A curious tale*, you built your own content management system. Now you have done that, you know how git works – because it works in exactly the same way as your own system. You will recognize hashes for files, directories and commits, commits linked by reference to their parents, the staging area, the `objects` directory, and bookmarks (branches).

Armed with this deep understanding, we retrace our steps to do the same content management tasks in git.

3.1 Basic configuration

We need to tell git our name and email address before we start.

Git will use this information to fill in the author information in each **commit message**, so we don't have to type it out every time.

```
$ git config --global user.name "Matthew Brett"
$ git config --global user.email "matthew.brett@gmail.com"
```

The `--global` flag tells git to store this information in its default configuration file for your user account. On Unix (e.g. OSX and Linux) this file is `.gitconfig` in your home directory. Without the `--global` flag, git only applies the configuration to the particular **repository** you are working in.

Every time we make a commit, we need to type a commit message. Git will open our text editor for us to type the message, but first it needs to know what text editor we prefer. Set your own preferred text editor here:

```
# gedit is a reasonable choice for Linux
# "vi" is the default.
git config --global core.editor gedit
```

Next we set the name of the default *branch*. We will explain branches later on, but, for now, just apply this configuration for compatibility with modern versions of the Git package:

```
$ # Set the default branch name to "main"
$ git config --global init.defaultBranch main
```

We also turn on the use of color, which is very helpful in making the output of git easier to read:

```
$ git config --global color.ui "auto"
```

3.2 Getting help

```
$ git help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
   sparse-checkout   Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
   bisect            Use binary search to find the commit that introduced a bug
   diff              Show changes between commits, commit and working tree, etc
   grep              Print lines matching a pattern
   log               Show commit logs
   show              Show various types of objects
   status            Show the working tree status

grow, mark and tweak your common history
   branch            List, create, or delete branches
```

(continues on next page)

(continued from previous page)

```
   commit            Record changes to the repository
   merge             Join two or more development histories together
   rebase            Reapply commits on top of another base tip
   reset             Reset current HEAD to the specified state
   switch            Switch branches
   tag               Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch             Download objects and refs from another repository
   pull              Fetch from and integrate with another repository or a local branch
   push              Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

Try `git help add` for an example.

Note: The git help pages are famously hard to read if you don't know how git works. One purpose of this tutorial is to explain git in such a way that it will be easier to understand the help pages.

3.3 Initializing the repository directory

We first set this `nobel_prize` directory to be version controlled with git. We start off the working tree with the original files for the paper:

Note: I highly recommend you type along. Why not download `nobel_prize.zip` and unzip the files to make the same `nobel_prize` directory as I have here?

```
nobel_prize
├── clever_analysis.py [618B]
├── expensive_data.csv [244K]
└── fancy_figure.png [183K]
```

To get started with git, create the git **repository directory** with `git init`:

```
$ cd nobel_prize
$ git init
Initialized empty Git repository in /Users/mb312/dev_trees/curious-git/working/nobel_
↪prize/.git/
```

What happened when we did `git init`? Just what we were expecting; we have a new repository directory in `nobel_prize` called `.git`

```
.git
├── refs
│   ├── tags
│   └── heads
```

(continues on next page)

(continued from previous page)

```
├── objects
│   ├── pack
│   └── info
├── info
│   └── exclude [240B]
├── hooks
│   (13 files)
├── HEAD [21B]
├── config [137B]
└── description [73B]
```

The `objects` directory looks familiar. It has exactly the same purpose as it did for your SAP system. At the moment it contains a couple of empty directories, because we have not added any objects yet.

3.4 Updating terms for git

Working directory The directory containing the files you are working on. In our case this is `nobel_prize`. It contains the **repository directory**, named `.git`.

Repository directory Directory containing all previous commits (snapshots) and git private files for working with commits. The directory has name `.git` by default, and almost always in practice.

3.5 `git add` – put stuff into the staging area

In the next few sections, we will do our first commit (snapshot).

First we will put the files for the commit into the staging area.

The command to put files into the staging area is `git add`.

To start, we show ourselves that the **staging area** is empty. We haven't yet discussed the git implementation of the staging area, but this command shows us which files are in the staging area.

```
$ git ls-files --stage
```

As expected, there are no files in the staging area yet.

Note: `git ls-files` is a specialized command that you will not often need in your daily git life. I'm using it here to show you how git works.

Now we do our add:

```
$ git add clever_analysis.py
```

Sure enough:

```
$ git ls-files --stage
100755 6560135a5943c0509608fee6d900b775e3041197 0       clever_analysis.py
```

3.6 The git staging area

It is time to think about what the staging area is, in git. In your SAP system, the staging area was a directory. You also started off by using directories to store commits (snapshots). Later you found you could do without the commit directories, because you could store the files in `repo/objects` and the directory structure in `directory_listing.txt` text files.

In git, the staging area is a single file called `.git/index`. This file contains a directory listing that is the equivalent of the `staging` directory in SAP. When we add a file to the staging area, git backs up the file with its hash to `.git/objects`, and then changes the directory listing inside `.git/index` to point to this backup copy.

If all that is true, then we now expect to see a) a new file `.git/index` containing the directory listing and b) a new file in the `.git/objects` directory corresponding to the hash for the `clever_analysis.py` file. We saw from the output of `git ls-files --stage` above that the hash for `clever_analysis.py` is `6560135a5943c0509608fee6d900b775e3041197`. So – do we see these files?

First – there is now a new file `.git/index` that was not present in our first listing of the `.git` directory above:

```
$ ls .git/index
.git/index
```

Second, there is a new directory and file in `.git/objects`:

```
objects
├── pack
├── info
└── 65
    └── 60135a5943c0509608fee6d900b775e3041197 [335B]
```

The directory and filename in `.git/objects` come from the hash of `clever_analysis.py`. The first two digits of the hash form the directory name and the rest of the digits are the filename[3]. So, the file `.git/objects/65/60135a5943c0509608fee6d900b775e3041197` is the copy of `clever_analysis.py` that we added to the staging area.

For extra points, what do you think would happen if we deleted the `.git/index` file (answer[1])?

3.7 Git objects

Git objects are nearly as simple as the objects you were writing in your SAP. The hash is not the hash of the raw file, but the raw file prepended with a short housekeeping string. See reading_git_objects for details.

We can see the contents of objects with the command `git cat-file -p`. For example, here are the contents of the backup we just made of `clever_analysis.py`:

```
$ git cat-file -p 6560135a5943c0509608fee6d900b775e3041197
# The brain analysis script
import numpy as np

import matplotlib.pyplot as plt
```

(continues on next page)

[3] When git stores a file in the `.git/objects` directory, it makes a hash from the file, takes the first two digits of the hash to make a directory name, and then stores a file in this directory with a filename from the remaining hash digits. For example, when adding a file with hash `d92d079af6a7f276cc8d63dcf2549c03e7deb553` git will create `.git/objects/d9` directory if it doesn't exist, and stores the file contents as `.git/objects/d9/2d079af6a7f276cc8d63dcf2549c03e7deb553`. It does this so that the number of files in any one directory stay in a reasonable range. If git had to store hash filenames for every object in one flat directory, the directory would soon have a very large number of files.

[1] What would happen if we delete the `.git/index` file? Remember, the `.git/index` file contains the directory listing for the staging area. If we delete the file, git will assume that the directory listing is empty, and therefore that there are no files in the staging area.

(continued from previous page)

```
FUDGE = 42

# Load data from the brain
data = np.loadtxt('expensive_data.csv', delimiter=',')

# First column is something from world, second is something from brain
from_world, from_brain = data.T

# Process data
from_brain_processed = np.log(from_brain) * FUDGE * np.e ** np.pi

# Make plot
plt.plot(from_world, from_brain_processed, 'r:')
plt.plot(from_world, from_brain_processed, 'bx')
plt.xlabel('Data from the outside world')
plt.ylabel('Data from inside the brain')
plt.title('Important finding')
plt.savefig('fancy_figure.png')
```

Note: I will use `git cat-file -p` to display the content of nearly raw git objects, to show the simplicity of git's internal model, but `cat-file` is a specialized command that you won't use much in daily work.

Just as we expected, it is the current contents of the `clever_analysis.py`.

The `6560135a5943c0509608fee6d900b775e3041197` object is a hashed, stored raw file. Because the object is a stored file rather than a stored directory listing text file or commit message text file, git calls this type of object a **blob** – for Binary Large Object. You can get the object *type* from the object hash with the `-t` flag to `git cat-file`:

```
$ git cat-file -t 6560135a5943c0509608fee6d900b775e3041197
blob
```

3.8 Hash values can usually be abbreviated to seven characters

We only need to give git enough hash digits for git to identify the object uniquely. 7 digits is nearly always enough, as in:

```
$ git cat-file -p 6560135
# The brain analysis script
import numpy as np

import matplotlib.pyplot as plt

FUDGE = 42

# Load data from the brain
data = np.loadtxt('expensive_data.csv', delimiter=',')

# First column is something from world, second is something from brain
from_world, from_brain = data.T
```

(continues on next page)

(continued from previous page)

```
# Process data
from_brain_processed = np.log(from_brain) * FUDGE * np.e ** np.pi

# Make plot
plt.plot(from_world, from_brain_processed, 'r:')
plt.plot(from_world, from_brain_processed, 'bx')
plt.xlabel('Data from the outside world')
plt.ylabel('Data from inside the brain')
plt.title('Important finding')
plt.savefig('fancy_figure.png')
```

3.9 `git status` – showing the status of files in the working tree

The working tree is the contents of the `nobel_prize` directory, excluding the `.git` repository directory.

`git status` tells us about the relationship of the files in the working tree to the repository and staging area.

We have done a `git add` on `clever_analysis.py`, and that added the file to the staging area. We can see that this happened with `git status`:

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   clever_analysis.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        expensive_data.csv
        fancy_figure.png
```

Sure enough, the output tells us that `new file:  clever_analysis.py` is in the `changes to be committed`. It also tells us that the other two files in the working directory are `untracked`.

An untracked file is a file with a filename that is not listed the staging area directory listing. Until you run `git add` on an untracked file, git will ignore these files and assume you don't want to keep track of them.

3.10 Staging the other files with git add

We do want to keep track of the other files, so we stage them:

```
$ git add fancy_figure.png
$ git add expensive_data.csv
$ git status
On branch main

No commits yet
```

(continues on next page)

(continued from previous page)

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   clever_analysis.py
        new file:   expensive_data.csv
        new file:   fancy_figure.png
```

We have staged all three of our files. We have three objects in `.git/objects`:

```
objects
├── pack
├── info
├── 7b
│   └── 37886351b3df2463fd29c87bc5184b637f0926 [119K]
├── 65
│   └── 60135a5943c0509608fee6d900b775e3041197 [335B]
└── 1e
    └── d447c15c125991b8a292bdb433aaf19998a3e9 [179K]
```

3.11 `git commit` – making the snapshot

```
[desktop]$ git commit -m "First backup of my amazing idea"
[main (root-commit) 75206bc] First backup of my amazing idea
 3 files changed, 5023 insertions(+)
 create mode 100755 clever_analysis.py
 create mode 100644 expensive_data.csv
 create mode 100644 fancy_figure.png
```

Note: In the line above, I used the `-m` flag to specify a message at the command line. If I had not done that, git would open the editor I specified in the `git config` step above and ask me to enter a message. I'm using the `-m` flag so the commit command runs without interaction in this tutorial, but in ordinary use, I virtually never use `-m`, and I suggest you don't either. Using the editor for the commit message allows you to write a more complete commit message, and gives feedback about the `git status` of the commit to remind you what you are about to do.

Following the logic of your SAP system, we expect that the action of making the commit will generate two new files in `.git/objects`, one for the directory listing text file, and another for the commit message:

```
objects
├── pack
├── info
├── ff
│   └── c871b48a6b9df8dc4a13e8e5da99ccf2ce458d [150B]
├── 7b
│   └── 37886351b3df2463fd29c87bc5184b637f0926 [119K]
├── 75
│   └── 206bcb33ff9ad4f15f89b52cdf95bf666d67a8 [148B]
├── 65
│   └── 60135a5943c0509608fee6d900b775e3041197 [335B]
└── 1e
    └── d447c15c125991b8a292bdb433aaf19998a3e9 [179K]
```

Here is the contents of the commit message text file for the new commit. Git calls this a **commit object**:

```
$ git cat-file -p 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
tree ffc871b48a6b9df8dc4a13e8e5da99ccf2ce458d
author Matthew Brett <matthew.brett@gmail.com> 1333287013 +0100
committer Matthew Brett <matthew.brett@gmail.com> 1333287013 +0100

First backup of my amazing idea
```

```
$ # What type of git object is this?
$ git cat-file -t 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
commit
```

As for SAP, the commit message file contains the hash for the directory tree file (`tree`), the hash of the parent (`parent`) (but this commit has no parents), the author, date and time, and the note.

Here's the contents of the directory listing text file for the new commit. Git calls this a **tree** object.

```
$ git cat-file -p ffc871b48a6b9df8dc4a13e8e5da99ccf2ce458d
100755 blob 6560135a5943c0509608fee6d900b775e3041197        clever_analysis.py
100644 blob 7b37886351b3df2463fd29c87bc5184b637f0926        expensive_data.csv
100644 blob 1ed447c15c125991b8a292bdb433aaf19998a3e9        fancy_figure.png
```

```
$ git cat-file -t ffc871b48a6b9df8dc4a13e8e5da99ccf2ce458d
tree
```

Each line in the directory listing gives the file permissions, the type of the entry in the directory (where "tree" means a sub-directory, and "blob" means a file), the file hash, and the file name (see git-object-types).

3.12 `git log` – what are the commits so far?

```
$ git log
commit 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
Author: Matthew Brett <matthew.brett@gmail.com>
Date:   Sun Apr 1 14:30:13 2012 +0100

    First backup of my amazing idea
```

Notice that git log identifies each commit with its hash. The hash is the hash for the contents of the commit message. As we saw above, the hash for our commit was `75206bcb33ff9ad4f15f89b52cdf95bf666d67a8`.

We can also ask to the see the parents of each commit in the log:

```
$ git log --parents
commit 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
Author: Matthew Brett <matthew.brett@gmail.com>
Date:   Sun Apr 1 14:30:13 2012 +0100

    First backup of my amazing idea
```

Why are the output of `git log` and `git log --parents` the same in this case? (answer[2]).

[2] Why are the output of `git log` and `git log --parents` the same in this case? They are the same because this is the first commit, and the first commit has no parents.

3.13 `git branch` - which branch are we on?

Branches are bookmarks. They associate a name (like "my_bookmark" or "main") with a commit (such as `75206bcb33ff9ad4f15f89b52cdf95bf666d67a8`).

The default branch (bookmark) for git is called `main`. Git creates it automatically when we do our first commit.

```
$ git branch
* main
```

Asking for more verbose detail shows us that the branch is pointing to a particular commit (where the commit is given by a hash):

```
$ git branch -v
* main 75206bc First backup of my amazing idea
```

In this case git abbreviated the 40 character hash to the first 7 digits, because these are enough to uniquely identify the commit.

A branch is nothing but a name that points to a commit. In fact, git stores branches as we did in SAP, as tiny text files, where the filename is the name of the branch, and the contents is the hash of the commit that it points to:

```
$ ls .git/refs/heads
main
```

```
$ cat .git/refs/heads/main
75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
```

We will soon see that, if we are working on a branch, and we do a commit, then git will update the branch to point to the new commit.

3.14 A second commit

In our second commit, we will add the first draft of the Nobel prize paper. As before, you can download this from `nobel_prize.md`. If you are typing along, download `nobel_prize.md` to the `nobel_prize` directory.

The staging area does not have an entry for `nobel_prize.md`, so `git status` identifies this file as **untracked**:

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        nobel_prize.md

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

We add the file to the staging area with `git add`:

```
$ git add nobel_prize.md
```

Now `git status` records this file being in the staging area, by listing it under "changes to be committed":

```
$ git status
On branch main
Changes to be committed:
```

(continues on next page)

(continued from previous page)

```
  (use "git restore --staged <file>..." to unstage)
        new file:   nobel_prize.md
```

Finally we move the changes from the staging area into a commit with `git commit`:

```
[desktop]$ git commit -m "Add first draft of paper"
[main 7919d37] Add first draft of paper
 1 file changed, 29 insertions(+)
 create mode 100644 nobel_prize.md
```

Git shows us the first 7 digits of the new commit hash in the output from `git commit` – these are `7919d37`.

Notice that the position of the current `main` branch is now this last commit:

```
$ git branch -v
* main 7919d37 Add first draft of paper
```

```
$ cat .git/refs/heads/main
7919d37dda9044f00cf2dc0677eed18156f75404
```

We use `git log` to look at our short history.

```
$ git log
commit 7919d37dda9044f00cf2dc0677eed18156f75404
Author: Matthew Brett <matthew.brett@gmail.com>
Date:   Mon Apr 2 18:03:00 2012 +0100

    Add first draft of paper

commit 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
Author: Matthew Brett <matthew.brett@gmail.com>
Date:   Sun Apr 1 14:30:13 2012 +0100

    First backup of my amazing idea
```

We add the `--parents` flag to show that the second commit points back to the first via its hash. Git lists the parent hash after the commit hash:

```
$ git log --parents
commit 7919d37dda9044f00cf2dc0677eed18156f75404 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
Author: Matthew Brett <matthew.brett@gmail.com>
Date:   Mon Apr 2 18:03:00 2012 +0100

    Add first draft of paper

commit 75206bcb33ff9ad4f15f89b52cdf95bf666d67a8
Author: Matthew Brett <matthew.brett@gmail.com>
Date:   Sun Apr 1 14:30:13 2012 +0100

    First backup of my amazing idea
```

3.15 `git diff` – what has changed?

Our next commit will have edits to the `clever_analysis.py` script. We will also refresh the figure with the result of running the script.

I open the `clever_analysis.py` file in text editor and adjust the fudge factor, add a new fudge factor, and apply the new factor to the data.

Now I've done these edits, I can ask `git diff` to show me how the files in my working tree differ from the files in the staging area.

Remember, the files the staging area knows about so far are the files as of the last commit.

```
$ git diff
diff --git a/clever_analysis.py b/clever_analysis.py
index 6560135..99cd07b 100755
--- a/clever_analysis.py
+++ b/clever_analysis.py
@@ -3,7 +3,8 @@ import numpy as np

 import matplotlib.pyplot as plt

-FUDGE = 42
+FUDGE = 106
+MORE_FUDGE = 2.0

 # Load data from the brain
 data = np.loadtxt('expensive_data.csv', delimiter=',')
@@ -13,6 +14,8 @@ from_world, from_brain = data.T

 # Process data
 from_brain_processed = np.log(from_brain) * FUDGE * np.e ** np.pi
+# Apply the new factor
+from_brain_processed = from_brain_processed / MORE_FUDGE

 # Make plot
 plt.plot(from_world, from_brain_processed, 'r:')
```

A - at the beginning of the `git diff` output means I have removed this line. A + at the beginning means I have added this line. As you see I have edited one line in this file, and added three more.

Open your text editor and edit `clever_analysis.py`. See if you can replicate my changes by editing the file, and checking with `git diff`.

Now check the status of `clever_analysis.py` with:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   clever_analysis.py

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

3.16 You need to `git add` a file to put it into the staging area

Remember that git only commits stuff that you have added to the staging area.

`git status` tells us that `clever_analysis.py` has been "modified", and that these changes are "not staged for commit".

There is a version of `clever_analysis.py` in the staging area, but it is the version of the file as of the last commit, and so that version is different from the version we have in the working tree.

If we try to do a commit, git will tell us there is nothing to commit, because there is nothing new in the staging area:

```
$ git commit
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   clever_analysis.py

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

To stage this version of `clever_analysis.py` we use `git add`:

```
$ git add clever_analysis.py
```

Git status now shows these changes as "Changes to be committed".

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   clever_analysis.py
```

We can update the figure by running the `analysis_script.py` script. The script analyzes the data and writes the figure to the current directory. If you have Python installed, with the `numpy` and `matplotlib` packages, you can run the analysis yourself with:

```
python clever_analysis.py
```

If not, you can download a `version of the figure I generated earlier`. After you have generated or downloaded the figure:

```
$ git add fancy_figure.png
```

Do a final check with `git status`, then make the commit with:

```
[desktop]$ git commit -m "Add another fudge factor"
[main 003c54a] Add another fudge factor
 2 files changed, 4 insertions(+), 1 deletion(-)
 rewrite fancy_figure.png (96%)
```

The branch bookmark has moved again:

```
$ git branch -v
* main 003c54a Add another fudge factor
```

3.17 An ordinary day in gitworld

We now have the main commands for daily work with git;

- Make some changes in the working tree;
- Check what has changed with `git status`;
- Review the changes with `git diff`;
- Add changes to the staging area with `git add`;
- Make the commit with `git commit`.

3.18 Commit four

For our next commit, we will add some more changes to the analysis script and figure, and add a new file, `references.bib`.

To follow along, first download `references.bib`.

Next, edit `clever_analysis.py` again, to make these changes:

```
$ git diff
diff --git a/clever_analysis.py b/clever_analysis.py
index 99cd07b..e5b2efa 100755
--- a/clever_analysis.py
+++ b/clever_analysis.py
@@ -5,6 +5,7 @@ import matplotlib.pyplot as plt

 FUDGE = 106
 MORE_FUDGE = 2.0
+NUDGE_FUDGE = 1.25

 # Load data from the brain
 data = np.loadtxt('expensive_data.csv', delimiter=',')
@@ -14,8 +15,8 @@ from_world, from_brain = data.T

 # Process data
 from_brain_processed = np.log(from_brain) * FUDGE * np.e ** np.pi
-# Apply the new factor
-from_brain_processed = from_brain_processed / MORE_FUDGE
+# Apply the new factor(s)
+from_brain_processed = from_brain_processed / MORE_FUDGE / NUDGE_FUDGE

 # Make plot
 plt.plot(from_world, from_brain_processed, 'r:')
```

Finally regenerate `fancy_figure.png`, or download the updated copy `from here`.

What will git status show now?

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

(continues on next page)

(continued from previous page)

```
        modified:   clever_analysis.py
        modified:   fancy_figure.png

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        references.bib

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

The staging area does not list a file called `references.bib` so this file is "untracked". The staging area does contain an entry for `clever_analysis.py` and `fancy_figure.png`, so these files are tracked. Git has checked the hashes for these files, and they are different from the hashes in the staging area, so git knows these files have changed, compared to the versions listed in the staging area.

Before we add our changes, we confirm that they are as we expect with:

```
$ git diff
diff --git a/clever_analysis.py b/clever_analysis.py
index 99cd07b..e5b2efa 100755
--- a/clever_analysis.py
+++ b/clever_analysis.py
@@ -5,6 +5,7 @@ import matplotlib.pyplot as plt

 FUDGE = 106
 MORE_FUDGE = 2.0
+NUDGE_FUDGE = 1.25

 # Load data from the brain
 data = np.loadtxt('expensive_data.csv', delimiter=',')
@@ -14,8 +15,8 @@ from_world, from_brain = data.T

 # Process data
 from_brain_processed = np.log(from_brain) * FUDGE * np.e ** np.pi
-# Apply the new factor
-from_brain_processed = from_brain_processed / MORE_FUDGE
+# Apply the new factor(s)
+from_brain_processed = from_brain_processed / MORE_FUDGE / NUDGE_FUDGE

 # Make plot
 plt.plot(from_world, from_brain_processed, 'r:')
diff --git a/fancy_figure.png b/fancy_figure.png
index 81fc437..d1f40df 100644
Binary files a/fancy_figure.png and b/fancy_figure.png differ
```

Notice that git does not try and show the line-by-line differences between the old and new figures, guessing correctly that this is a binary and not a text file.

Now we have reviewed the changes, we add them to the staging area and commit:

```
$ git add references.bib
$ git add clever_analysis.py
$ git add fancy_figure.png
```

```
[desktop]$ git commit -m "Change analysis and add references"
[main 2d9e1df] Change analysis and add references
 3 files changed, 13 insertions(+), 2 deletions(-)
 rewrite fancy_figure.png (89%)
 create mode 100644 references.bib
```

The branch bookmark has moved to point to the new commit:

```
$ git branch -v
* main 2d9e1df Change analysis and add references
```

3.19 Undoing a commit with `git reset`

As you found in the SAP story, this last commit doesn't look quite right, because the commit message refers to two different types of changes. With more git experience, you will likely find that you like to break your changes into commits where the changes have a particular theme or purpose. This makes it easier to see what happened when you look over the history and the commit messages with `git log`.

So, as in the SAP story, you decide to undo the last commit, and replace it with two commits:

- One commit to add the changes to the script and figure;
- Another commit on top of the first, to add the references file.

In the SAP story, you had to delete a snapshot directory manually, and reset the staging area directory to have the contents of the previous commit. In git, all we have to do is reset the current `main` branch bookmark to point to the previous commit. By default, git will also reset the staging area for us. The command to move the branch bookmark is `git reset`.

3.20 Pointing backwards in history

The commit that we want the branch to point to is the previous commit in our commit history. We can use `git log` to see that this commit has hash `003c54a`. So, we could do our reset with `git reset 003c54a`. There is a simpler and more readable way to write this common idea, of one commit back in history, and that is to add `~1` to a reference. For example, to refer to the commit that is one step back in the history from the commit pointed to by the `main` branch, you can write `main~1`. Because `main` points to commit `2d9e1df`, you could also append the `~1` to `2d9e1df`. You can imagine that `main~2` will point two steps back in the commit history, and so on.

So, a readable reset command for our purpose is:

```
$ git reset main~1
Unstaged changes after reset:
M       clever_analysis.py
M       fancy_figure.png
```

Notice that the branch pointer now points to the previous commit:

```
$ git branch -v
* main 003c54a Add another fudge factor
```

Remember in SAP that your procedure for breaking up the snapshot was to 1) delete the old snapshot and 2) reset the staging area to reflect the previous commit. After you did this, the working tree contains your changes, but the staging area does not. You could make your new commits in the usual way, by adding to the staging area, and doing the commits.

Notice that `git reset` has done the same thing. It has reset the staging area to the state as of the older commit, but it has left the working tree alone. That means that `git status` will show us the changes in the working tree compared to the commit we have just reset to:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   clever_analysis.py
        modified:   fancy_figure.png

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        references.bib

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

We have the changes from our original fourth commit in our working tree, but we have not staged them. We are ready to make our new separate commits.

3.21 A new fourth commit

As we planned, we make a commit by adding only the changes from the script and figure:

```
$ git add clever_analysis.py
$ git add fancy_figure.png
```

```
[desktop]$ git commit -m "Change parameters of analysis"
[main d0ef727] Change parameters of analysis
 2 files changed, 3 insertions(+), 2 deletions(-)
 rewrite fancy_figure.png (89%)
```

Notice that git status now tells us that we still have untracked (and therefore not staged) changes in our working tree:

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        references.bib

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

3.22 The fifth commit

To finish our work splitting the fourth commit into two, we add and commit the `references.bib` file:

```
$ git add references.bib
```

```
[desktop]$ git commit -m "Add references"
[main c3f19b2] Add references
 1 file changed, 10 insertions(+)
 create mode 100644 references.bib
```

3.23 Getting a file from a previous commit – `git checkout`

In the SAP story, we found that the first version of the analysis script was correct, and we made a new commit after restoring this version from the first snapshot.

As you can imagine, git allows us to do that too. The command to do this is `git checkout`

If you have a look at `git checkout --help` you will see that git checkout has two roles, described in the help as "Checkout a branch or paths to the working tree". We will see checking out a branch later, but here we are using checkout in its second role, to restore files to the working tree.

We do this by telling git checkout which version we want, and what file we want. We want the version of `clever_analysis.py` as of the first commit. To find the first commit, we can use git log. To make git log a bit less verbose, I've added the `--oneline` flag, to print out one line per commit:

```
$ git log --oneline
c3f19b2 Add references
d0ef727 Change parameters of analysis
003c54a Add another fudge factor
7919d37 Add first draft of paper
75206bc First backup of my amazing idea
```

Now we have the abbreviated commit hash for the first commit, we can checkout that version to the working tree:

```
$ git checkout 75206bc clever_analysis.py
Updated 1 path from ffc871b
```

We also want the previous version of the figure:

```
$ git checkout 75206bc fancy_figure.png
Updated 1 path from ffc871b
```

Notice that the checkout also added the files to the staging area:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   clever_analysis.py
        modified:   fancy_figure.png
```

We are ready for our sixth commit:

```
[desktop]$ git commit -m "Revert to original script & figure"
[main 677c9a6] Revert to original script & figure
 2 files changed, 1 insertion(+), 5 deletions(-)
 rewrite fancy_figure.png (97%)
```

3.24 Using bookmarks – `git branch`

We are at the stage in the SAP story where Josephine goes away to the conference.

Let us pretend that we are Josephine, and that we have taken a copy of the *nobel_prize* directory to the conference. The copy includes the `.git` subdirectory, containing the git repository.

Now we (as Josephine) start doing some work. We don't want to change the previous bookmark, which is `main`:

```
$ git branch -v
* main 677c9a6 Revert to original script & figure
```

We would like to use our own bookmark, so we can make changes without affecting anyone else. To do this we use `git branch` with arguments:

```
$ git branch josephines-branch main
```

The first argument is the name of the branch we want to create. The second is the commit at which the branch should start. Now we have a new branch, that currently points to the same commit as `main`:

```
$ git branch -v
  josephines-branch 677c9a6 Revert to original script & figure
* main              677c9a6 Revert to original script & figure
```

The new branch is nothing but a text file pointing to the commit:

```
$ cat .git/refs/heads/josephines-branch
677c9a6ac3004e5d84d9739751a89867e09238f4
```

Now we have two branches, git needs to know which branch we are working on. The asterisk next to `main` in the output of `git branch` means that we are working on `main` at the moment. If we make another commit, it will update the `main` bookmark.

Git stores the current branch in the file `.git/HEAD`:

```
$ cat .git/HEAD
ref: refs/heads/main
```

Git commands often allow you to write `HEAD` meaning "the branch or commit you are currently working on". For example, `git log HEAD` means "show the log starting at the branch or commit you are currently working on". In fact, this is also the default behavior of `git log`.

We now want to make `josephines-branch` current, so any new commits will update `josephines-branch` instead of `main`.

3.25 Changing the current branch with `git checkout`

We previously saw that `git checkout <commit> <filename>` will get the file `<filename>` as of commit `<commit>`, and restore it to the working tree. This was the second of the two uses of `git checkout`. We now come to the first and most common use of `git checkout`, which is to:

- change the current branch to a given branch or commit;
- restore the working tree and staging area to the file versions from the given commit.

We are about to do `git checkout josephines-branch`. When we do this, we are only going to see the first of these two effects, because `main` and `josephines-branch` point to the same commit, and so have the same file contents:

```
$ git checkout josephines-branch
Switched to branch 'josephines-branch'
```

The asterisk has now moved to `josephines-branch`:

```
$ git branch -v
* josephines-branch 677c9a6 Revert to original script & figure
  main              677c9a6 Revert to original script & figure
```

This is because the file `HEAD` now points to `josephines-branch`:

```
$ cat .git/HEAD
ref: refs/heads/josephines-branch
```

If we do a commit, git will update `josephines-branch`, not `main`.

3.26 Making commits on branches

Josephine did some edits to the paper. If you are typing along, make these changes to `nobel_prize.md`:

```
$ git diff
diff --git a/nobel_prize.md b/nobel_prize.md
index 3ef5df2..19fd4c5 100644
--- a/nobel_prize.md
+++ b/nobel_prize.md
@@ -5,6 +5,12 @@
 The brain thinks in straight lines once you do some poorly-motivated
 corrections on some brain recordings.

+Other people have done brain recordings and claimed that they were
+interesting, but they are not as interesting as our recordings.
+
+In our previous work, we have done some other brain recordings, that were also
+interesting, but in a different way.
+
 == Methods

 We took some recordings of someone's brain while we showed them stuff
```

As usual, we add the file to the staging area, and check the status of the working tree:

```
$ git add nobel_prize.md
$ git status
On branch josephines-branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   nobel_prize.md
```

Finally we make the commit:

```
[desktop]$ git commit -m "Expand the introduction"
[josephines-branch 0a5a1b9] Expand the introduction
 1 file changed, 6 insertions(+)
```

The `main` branch has not changed, but `josephines-branch` changed to point to the new commit:

```
$ git branch -v
* josephines-branch 0a5a1b9 Expand the introduction
  main              677c9a6 Revert to original script & figure
```

Now we go back to being ourselves, working in the lab. We change back to the `main` branch:

```
$ git checkout main
Switched to branch 'main'
```

The asterisk now points at `main`:

```
$ git branch -v
  josephines-branch 0a5a1b9 Expand the introduction
* main              677c9a6 Revert to original script & figure
```

If you look at the contents of `nobel_prize.md` in the working directory, you will see that we are back to the contents before Josephine's changes. This is because `git checkout main` reverted the files to their state as of the last commit on the `main` branch.

Now we make our own changes to the script and figure. Here are the changes to the script:

```
$ git diff
diff --git a/clever_analysis.py b/clever_analysis.py
index 6560135..cf163af 100755
--- a/clever_analysis.py
+++ b/clever_analysis.py
@@ -4,6 +4,7 @@ import numpy as np
 import matplotlib.pyplot as plt

 FUDGE = 42
+HOT_FUDGE = 1.707

 # Load data from the brain
 data = np.loadtxt('expensive_data.csv', delimiter=',')
@@ -13,6 +14,7 @@ from_world, from_brain = data.T

 # Process data
 from_brain_processed = np.log(from_brain) * FUDGE * np.e ** np.pi
+from_brain_processed = from_brain_processed / HOT_FUDGE
```

(continues on next page)

(continued from previous page)

```
 # Make plot
 plt.plot(from_world, from_brain_processed, 'r:')
```

If you are typing along, then you will also want to regenerate the figure with `python clever_analysis.py` or `download the new version`.

This gives:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   clever_analysis.py
        modified:   fancy_figure.png

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

As usual, we add the files and do the commit:

```
$ git add clever_analysis.py
$ git add fancy_figure.png
```

```
[desktop]$ git commit -m "More fun with fudge"
[main 7598e4e] More fun with fudge
 2 files changed, 2 insertions(+)
 rewrite fancy_figure.png (96%)
```

Because `HEAD` currently current points to `main`, git updated the `main` branch with the new commit:

```
$ git branch -v
  josephines-branch 0a5a1b9 Expand the introduction
* main              7598e4e More fun with fudge
```

3.27 Merging lines of development with `git merge`

We next want to get Josephine's changes into the `main` branch.

We do this with `git merge`:

```
$ git merge josephines-branch
Merge made by the 'recursive' strategy.
 nobel_prize.md | 6 ++++++
 1 file changed, 6 insertions(+)
```

This commit has the changes we just made to the script and figure, and the changes the Josephine made to the paper.

The commit has two parents, which are the two commits from which we merged:

```
$ git log --oneline --parents
3ae3b49 7598e4e 0a5a1b9 Merge branch 'josephines-branch'
0a5a1b9 677c9a6 Expand the introduction
```

(continues on next page)

(continued from previous page)

```
7598e4e 677c9a6 More fun with fudge
677c9a6 c3f19b2 Revert to original script & figure
c3f19b2 d0ef727 Add references
d0ef727 003c54a Change parameters of analysis
003c54a 7919d37 Add another fudge factor
7919d37 75206bc Add first draft of paper
75206bc First backup of my amazing idea
```

3.28 The commit parents make the development history into a graph

As you saw in your SAP system, we can think of the commits as nodes in a graph. Each commit stores the identity of its parent commit(s). The pointers from each commit back to its parent(s) link the commits (nodes) to form edges.

It is common to see a git history shown as a graph, and it is often useful to think of this graph when we are working with a git repository.

There are a lot of graphical tools to show the git history as a graph, but `git log` has a useful flag called `--graph` which shows the commits as a graph using text characters:

```
$ git log --oneline --graph
*   3ae3b49 Merge branch 'josephines-branch'
|\
| * 0a5a1b9 Expand the introduction
* | 7598e4e More fun with fudge
|/
* 677c9a6 Revert to original script & figure
* c3f19b2 Add references
* d0ef727 Change parameters of analysis
* 003c54a Add another fudge factor
* 7919d37 Add first draft of paper
* 75206bc First backup of my amazing idea
```

This kind of display is so useful that many of us have a shortcut to this command, that we use instead of the standard git log. You can make customized shortcuts to git commands by setting `alias` entries using `git config`. For example, you may want to set up an alias like this:

```
$ git config --global alias.slog "log --oneline --graph"
```

Now you can use the command `git slog` to mean `git log --oneline --graph`. Because of the `--global` flag, this command sets up the `slog` alias as the default for your user account, so you can use `git slog` whenever you are using git as this user on this computer.

```
$ git slog
*   3ae3b49 Merge branch 'josephines-branch'
|\
| * 0a5a1b9 Expand the introduction
* | 7598e4e More fun with fudge
|/
* 677c9a6 Revert to original script & figure
* c3f19b2 Add references
* d0ef727 Change parameters of analysis
* 003c54a Add another fudge factor
```

(continues on next page)

(continued from previous page)

```
* 7919d37 Add first draft of paper
* 75206bc First backup of my amazing idea
```

3.29 Other commands you need to know

This tutorial gives you the basics on working with files on your own computer, and on your own repository.

You will also need to know about:

- git remotes – curious_remotes;
- tags – making static bookmarks to commits;

You will probably also find use for:

- git reflog – show a list of previous commits that you have made;
- git rebase – rewrite the development history by altering or transplanting commits. See rebase without tears.

3.30 Git: are you ready?

If you followed this tutorial, you now have a good knowledge of how git works. This will make it much easier to understand why git commands do what they do, and what to do when things go wrong. You know all the main terms that the git manual pages use, so git's own help will be more useful to you. You will likely lead a long life of deep personal fulfillment.

3.31 Other git-ish things to read

As you've seen, this tutorial makes the bold assumption that you'll be able to understand how git works by seeing how it is *built*. These documents take a similar approach to varying levels of detail:

- The Git parable by Tom Preston-Werner;
- The visual git tutorial gives a nice visual idea of git at work;
- Understanding Git Conceptually gives another review of how the ideas behind git;
- For more detail, see the start of the excellent Pro Git online book, or similarly the early parts of the Git community book. Pro Git's chapters are very short and well illustrated; the community book tends to have more detail and has nice screencasts at the end of some sections;
- git foundation;

You might also try:

- For windows users, an Illustrated Guide to Git on Windows is useful in that it contains also some information about handling SSH (necessary to interface with git hosted on remote servers when collaborating) as well as screenshots of the Windows interface.
- Git ready A great website of posts on specific git-related topics, organized by difficulty.
- QGit: an excellent Git GUI Git ships by default with gitk and git-gui, a pair of Tk graphical clients to browse a repo and to operate in it. I personally have found qgit to be nicer and easier to use. It is available on modern Linux distros, and since it is based on Qt, it should run on OSX and Windows.
- Git Magic : Another book-size guide that has useful snippets.

- Github Guides have tutorials on a number of topics, some specific to Github hosting but much of it of general value.
- A port of the Hg book's beginning The Mercurial book has a reputation for clarity, so Carl Worth decided to port its introductory chapter to Git. It's a nicely written intro, which is possible in good measure because of how similar the underlying models of Hg and Git ultimately are.
- Intermediate tips: A set of tips that contains some very valuable nuggets, once you're past the basics.