Git Internals and Tricks: how it works, and how to get out of trouble

Author: Eric Silberstein

Claps: 1

Date: May 15

I met Josh Bradt during the lunch part of my interview at Klaviyo. Heâ€™d recently received a doctorate in physics, joined Klaviyo as a software engineer, and was working on scaling our profile segmentation system. Fast forward 5+ years and Josh is now a lead software engineer. (He and his colleagues are once again working on scaling our segmentation engine, except now to a level we couldnâ€™t imagine needing back then, and with an entirely new architecture. Itâ€™s a fascinating technical story that he and the team will be telling in a series of upcoming blog posts.)

Early last year, Josh gave an internal tech talk: *Git Internals and Tricks: How it works, and how to get out of trouble.* He motivated his talk with:

> â€œWhat is Git and how does it work inside? And you might wonder why should I care about the internals of this tool â€” I can just use it. But I think itâ€™s good to understand how it works on the inside because then you can understand what itâ€™s doing and learn how to write the commands you need from scratch rather than being dependent on magical things you find on the internet. It also means you can understand how to fix problems when you do something wrong. Also Git is just an interesting system so I think itâ€™s worth studying in its own right.â€

Josh had me at â€œrather than being dependent on magical thingsâ€¦â€ Like everyone, Iâ€™ve been using Git for years, and like probably a lot of people, I never took the time to understand it. Most of the time thatâ€™s fine. And every so often itâ€™s incredibly frustrating, and becomes only more frustrating and dangerous when the magical command you look up on Stack Overflow doesnâ€™t do what you want.

Joshâ€™s talk:

The examples

I learned a lot watching Joshâ€™s talk, and even more when I reproduced his examples.

Commits, blobs, and trees

Make an empty directory and initialize git:

```
> mkdir scratch
> cd scratch
> git init
Initialized empty Git repository in /Users/eric/tmp/scratch/.git/
```

```
> ls -a
. .. .git
```

Look inside `.git`. No objects yet:

```
> find .git
.git
.git/config
.git/objects
.git/objects/pack
.git/objects/info
.git/HEAD
.git/info
.git/info/exclude
.git/description
.git/hooks
.git/hooks/commit-msg.sample
...
.git/refs
.git/refs/heads
.git/refs/tags
```

Create and commit a single file:

```
> echo "Hello World" > test.txt
> git add test.txt
> git commit -m "Added test file"
[master (root-commit) a56b8ea] Added test file
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

Look inside `.git` again:

```
> find .git
.git
.git/config
.git/objects
.git/objects/a5
.git/objects/a5/6b8ea9312bc208263674f0334c53b854b48e56
.git/objects/pack
.git/objects/info
.git/objects/55
.git/objects/55/7db03de997c86a4a028e1ebd3a1ceb225be238
.git/objects/1e
.git/objects/1e/6dbf97adb05c42dcb537cd717e368812dc23b5
.git/HEAD
.git/info
.git/info/exclude
.git/logs
.git/logs/HEAD
.git/logs/refs
.git/logs/refs/heads
.git/logs/refs/heads/master
.git/description
```

```
.git/hooks
...
.git/refs
.git/refs/heads
.git/refs/heads/master
.git/refs/tags
.git/index
.git/COMMIT_EDITMSG
```

Each object is a **commit** object, a **tree** object, or a **blob** object as Josh explains. Look at them starting with the **commit** object:

```
> git cat-file -p a56b8ea # commit hash from above
tree 1e6dbf97adb05c42dcb537cd717e368812dc23b5
author Eric Silberstein <eric.silberstein@klaviyo.com> 1667418085 -0400
committer Eric Silberstein <eric.silberstein@klaviyo.com> 1667418085 -0400

Added test file
```

Look at the **tree** object referenced in the commit:

```
> git cat-file -p 1e6dbf9
100644 blob 557db03de997c86a4a028e1ebd3a1ceb225be238 test.txt
```

The tree contains a single **blob** object which is the contents of our file `test.txt`:

```
> git cat-file -p 557db03
Hello World
```

As Josh points out, these objects really are just zlib compressed files. Create a python script to decompress and look at the commit object:

```
> cat zlib_cat.py
import sys, zlib

with open(sys.argv[1], "rb") as f:
 sys.stdout.write(zlib.decompress(f.read()).decode())

> python zlib_cat.py .git/objects/a5/6b8ea9312bc208263674f0334c53b854b48e5
commit 210tree 1e6dbf97adb05c42dcb537cd717e368812dc23b5
author Eric Silberstein <eric.silberstein@klaviyo.com> 1667418085 -0400
committer Eric Silberstein <eric.silberstein@klaviyo.com> 1667418085 -0400

Added test file
```

HEAD points to refs/heads/master:

```
> cat .git/HEAD
ref: refs/heads/master
```

refs/heads/master points to our commit:

```
> cat .git/refs/heads/master
a56b8ea9312bc208263674f0334c53b854b48e56
```

All of which matches what `git log` tell us:

```
> git log --graph --oneline --all
* a56b8ea (HEAD -> master) Added test file
```

Now create and commit a new directory and file:

```
> mkdir stuff
> echo "Something else" > stuff/another_file.txt
> git add stuff
> git commit -m "Added more stuff"
[master 41878b4] Added more stuff
```

This new commit object references a newly created tree object and also references the parent commit a56b8ea:

```
> git cat-file -p 41878b4
tree 0a732c07e2fc35d27ca1310c7853f4b73efca273
parent a56b8ea9312bc208263674f0334c53b854b48e56
author Eric Silberstein <eric.silberstein@klaviyo.com> 1667419210 -0400
committer Eric Silberstein <eric.silberstein@klaviyo.com> 1667419210 -0400

Added more stuff
```

And the new tree object lists the old, unchanged blob object for `test.txt` and a subtree for the `stuff` directory:

```
> git cat-file -p 0a732c07
040000 tree ed54e23257692b7102c4c82f5ca73ad8eedfc2b5 stuff
100644 blob 557db03de997c86a4a028e1ebd3a1ceb225be238 test.txt
```

Look at what `log` and `rev-parse` tell usâ€¦all as expected:

```
> git log --graph --oneline --all
* 41878b4 (HEAD -> master) Added more stuff
* a56b8ea Added test file

> git rev-parse master
41878b490d1c0184c8c32580e398a7a204895404

> git rev-parse HEAD
41878b490d1c0184c8c32580e398a7a204895404

> git rev-parse HEAD~1 # the prior commit
a56b8ea9312bc208263674f0334c53b854b48e56
```

Example: Undoing a commit to master

(This example builds on the steps above.)

Add a new file and commit it:

```
> echo "things" > things.txt
> git add things.txt
```

```
> git commit -m "Added things"
[master f5df808] Added things
```

Oops! We were on master:

```
git log --graph --oneline --all
* f5df808 (HEAD -> master) Added things
* 41878b4 Added more stuff
* a56b8ea Added test file
```

Create a new branch at the same commit:

```
> git switch -c work_on_things
Switched to a new branch 'work_on_things'
```

Now we have two branches (pointers) at this commit and HEAD points to the new `work_on_things` branch.

```
> git log --graph --oneline --all
* f5df808 (HEAD -> work_on_things, master) Added things
* 41878b4 Added more stuff
* a56b8ea Added test file
```

Switch back to master and reset it to where it was initially. You can also reset to origin/master if you have a remote branch to refer to.

```
> git switch master
Switched to branch 'master'

> git reset --hard HEAD~1 # HEAD~1 is the commit before HEAD
HEAD is now at 41878b4 Added more stuff
```

Now everything is back to normal!

```
> git log --graph --oneline --all
* f5df808 (work_on_things) Added things
* 41878b4 (HEAD -> master) Added more stuff
* a56b8ea Added test file

> ls
stuff  test.txt zlib_cat.py

> git switch work_on_things
Switched to branch 'work_on_things'

> ls
stuff  test.txt things.txt zlib_cat.py
```

Example: Pulling a branch that someone else rebased

Create a remote. For the sake of this example, it will actually be another local directory. We initialize in bare mode meaning itâ€™s intended as a shared repository and doesnâ€™t have a working directory.

```
> cd ..
> pwd
> /Users/eric/tmp

> git init --bare scratch_remote.git
Initialized empty Git repository in /Users/eric/tmp/scratch_remote.git/
```

Set that as the remote for the repo we initialized above:

```
> cd scratch
> git checkout master
> git remote add origin ~/tmp/scratch_remote.git
> git remote -v
origin /Users/eric/tmp/scratch_remote.git (fetch)
origin /Users/eric/tmp/scratch_remote.git (push)

> git push --set-upstream origin master
...
Branch 'master' set up to track remote branch 'master' from 'origin'.

> git log --graph --oneline --all
* f5df808 (work_on_things) Added things
* 41878b4 (HEAD -> master, origin/master) Added more stuff
* a56b8ea Added test file
```

Get rid of the branch `work_on_things` so thereâ€™s one less thing to distract as we do this example:

```
> git branch --delete --force work_on_things
> git log --graph --oneline --all
* 41878b4 (HEAD -> master, origin/master) Added more stuff
* a56b8ea Added test file
```

Clone into directory `scratch_2` where our *person 2* will work:

```
> cd ..
> pwd
/Users/eric/tmp
> git clone ~/tmp/scratch_remote.git scratch_2
Cloning into 'scratch_2'...
done.
```

Have this *person 2* make a new branch and add file `foo.txt` to it:

```
> cd scratch_2
> git switch -c "working_on_foo"
Switched to a new branch 'working_on_foo'

> echo "Hello" > foo.txt
> git add foo.txt
> git commit -m "added foo.txt"
[working_on_foo 57bf94f] added foo.txt

> git push --set-upstream origin working_on_foo
```

```
...
Branch 'working_on_foo' set up to track remote branch 'working_on_foo' fro

> git log --graph --oneline --all
* 57bf94f (HEAD -> working_on_foo, origin/working_on_foo) added foo.txt
* 41878b4 (origin/master, origin/HEAD, master) Added more stuff
* a56b8ea Added test file
```

Have our original *person 1* add and commit new file `bar.txt` on master:

```
> cd ../scratch
> echo "Hello" > bar.txt
> git add bar.txt
> git commit -m "added bar.txt" bar.txt
[master 9b80604] added bar.txt

> git push
...
   41878b4..9b80604  master -> master
```

Now *person 2* asks *person 1* to look at their branch `working_on_foo`, so pull that:

```
> git fetch
...
> git switch working_on_foo
...
Switched to a new branch 'working_on_foo'

> git log --graph --oneline --all
* 9b80604 (origin/master, master) added bar.txt
| * 57bf94f (HEAD -> working_on_foo, origin/working_on_foo) added foo.txt
|/
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
foo.txt  stuff  test.txt zlib_cat.py
```

Meanwhile, *person 2* pulls master and then rebases their branch `working_on_foo`:

```
> cd ../scratch_2
> git fetch
> git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 1 commit, and can be fast-forward
  (use "git pull" to update your local branch)

> git pull
Updating 41878b4..9b80604
Fast-forward
 bar.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 bar.txt
```

```
> git checkout working_on_foo
Switched to branch 'working_on_foo'
Your branch is up to date with 'origin/working_on_foo'.

> git log --graph --oneline --all
* 9b80604 (origin/master, origin/HEAD, master) added bar.txt
| * 57bf94f (HEAD -> working_on_foo, origin/working_on_foo) added foo.txt
|/
* 41878b4 Added more stuff
* a56b8ea Added test file

> git rebase master
Successfully rebased and updated refs/heads/working_on_foo.

> git push --force
...
 + 57bf94f...1421d17 working_on_foo -> working_on_foo (forced update)

> git log --graph --oneline --all
* 1421d17 (HEAD -> working_on_foo, origin/working_on_foo) added foo.txt
* 9b80604 (origin/master, origin/HEAD, master) added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
bar.txt  foo.txt  stuff  test.txt
```

Now *person 1* wants to get to their local to the latest `working_on_foo`, but fast forward merge wonâ€™t work as Josh explains in his talk:

```
> cd ../scratch
> git fetch
> git log --graph --oneline --all
* 1421d17 (origin/working_on_foo) added foo.txt
* 9b80604 (origin/master, master) added bar.txt
| * 57bf94f (HEAD -> working_on_foo) added foo.txt
|/
* 41878b4 Added more stuff
* a56b8ea Added test file

> git pull --ff-only
fatal: Not possible to fast-forward, aborting.
```

But all *person 1* really wants is to move their local branch so it points to the `origin/working_on_foo` commit:

```
> git reset --hard origin/working_on_foo
HEAD is now at 1421d17 added foo.txt
```

Now everything is right:

```
git log --graph --oneline --all
* 1421d17 (HEAD -> working_on_foo, origin/working_on_foo) added foo.txt
* 9b80604 (origin/master, master) added bar.txt
```

```
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
bar.txt  foo.txt  stuff  test.txt zlib_cat.py
```

Example: Unwinding a bad rebase

Starting state (continued from above):

```
> cd ../scratch
> git log --graph --oneline --all
* 1421d17 (HEAD -> working_on_foo, origin/working_on_foo) added foo.txt
* 9b80604 (origin/master, master) added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file
```

Merge and delete the `working_on_foo` branch so we can focus on this new example:

```
> git checkout master
> git merge working_on_foo
> git push
> git push -d origin working_on_foo
> git branch --delete working_on_foo
Deleted branch working_on_foo (was 1421d17).

> git log --graph --oneline --all
* 1421d17 (HEAD -> master, origin/master) added foo.txt
* 9b80604 added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file
```

As *person 2*, add and commit file `whatever.txt`:

```
> cd ../scratch_2
> git checkout master
> git pull

> echo "Hello" > whatever.txt
> git add whatever.txt
> git commit -m "Added whatever"
> git push
```

As *person 1*, on a new branch, commit `zlib_cat.py`, then modify it and commit it again:

```
> cd ../scratch
> git switch -c "add_zlib_cat"
> git add zlib_cat.py
> git commit -m "Added zlib_cat.py"
> echo "# some docs" >> zlib_cat.py
> git add zlib_cat.py
> git commit -m "Added some docs"
> git fetch
```

```
> git checkout master
> git pull
```

Now hereâ€™s our state. We have two new commits based off the old master commit.

```
> git checkout add_zlib_cat

> git log --graph --oneline --all
* 43b2dc8 (HEAD -> add_zlib_cat) Added some docs
* e65765d Added zlib_cat.py
| * ea23b77 (origin/master, master) Added whatever
|/
* 1421d17 added foo.txt
* 9b80604 added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
bar.txt  foo.txt  stuff  test.txt zlib_cat.py
```

Rebase onto master (**this command is intentionally wrong**):

```
> git rebase --onto master HEAD
Successfully rebased and updated refs/heads/add_zlib_cat.
```

Oops, we lost our changes! (We lost `zlib_cat.py`)

```
> git log --graph --oneline --all
* ea23b77 (HEAD -> add_zlib_cat, origin/master, master) Added whatever
* 1421d17 added foo.txt
* 9b80604 added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
bar.txt  foo.txt  stuff  test.txt
```

Fix:

```
> git reflog add_zlib_cat
ea23b77 (HEAD -> add_zlib_cat, origin/master, master) add_zlib_cat@{0}: re
43b2dc8 add_zlib_cat@{1}: commit: Added some docs
e65765d add_zlib_cat@{2}: commit: Added zlib_cat.py
1421d17 add_zlib_cat@{3}: branch: Created from HEAD

> git reset --hard 43b2dc8
HEAD is now at 43b2dc8 Added some docs
```

Now weâ€™re back!

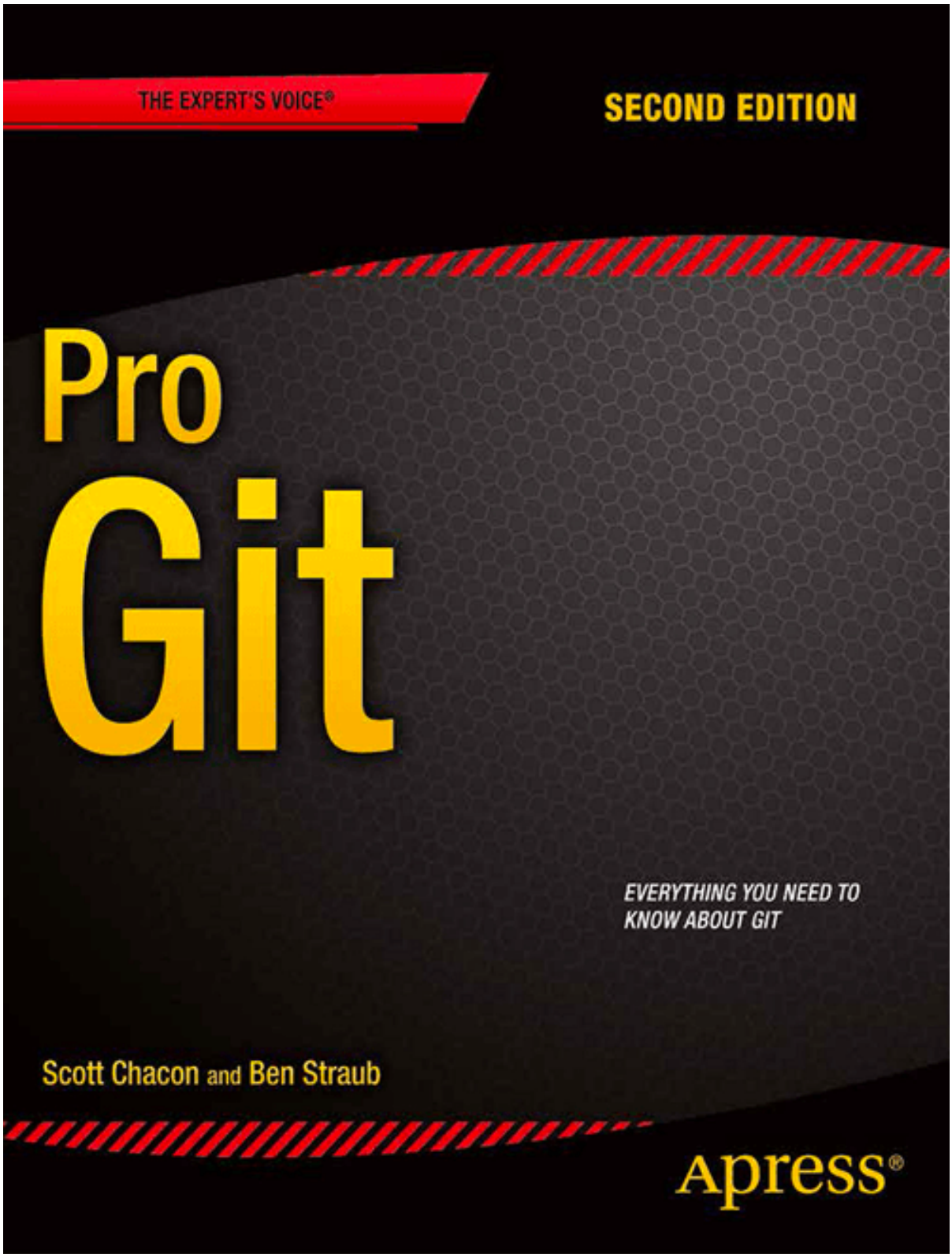
```
> git log --graph --oneline --all
* 43b2dc8 (HEAD -> add_zlib_cat) Added some docs
* e65765d Added zlib_cat.py
| * ea23b77 (origin/master, master) Added whatever
```

```
|/
* 1421d17 added foo.txt
* 9b80604 added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
bar.txt  foo.txt  stuff  test.txt zlib_cat.py
```

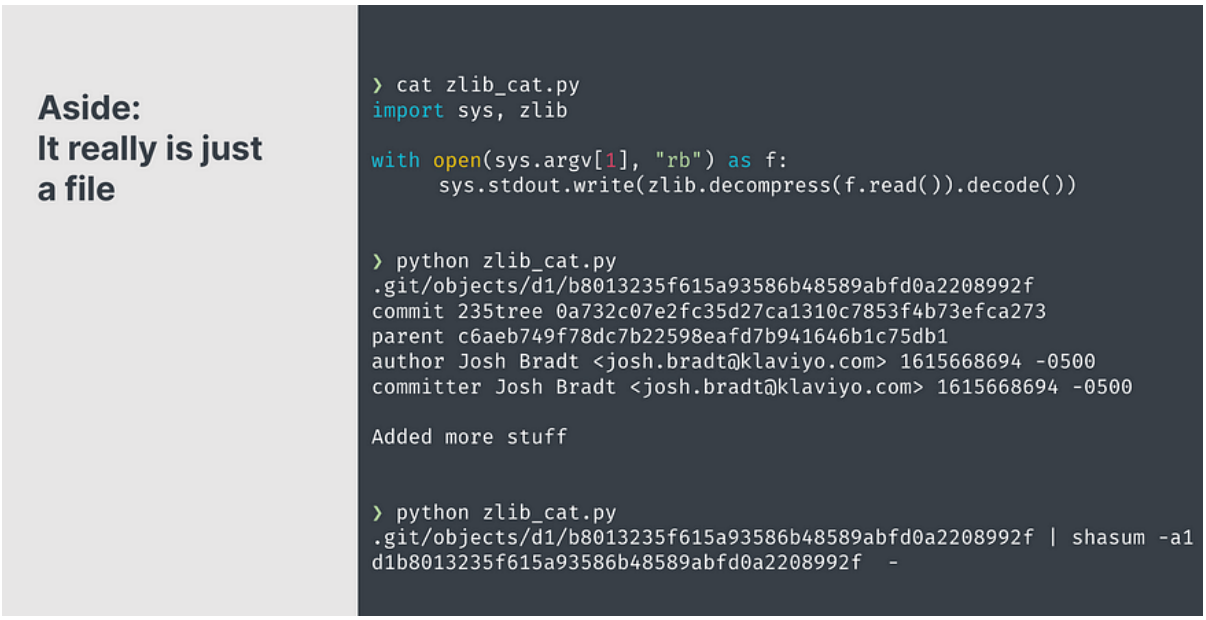
And can we do what we meant to do in the first place!

```
> git rebase master
Successfully rebased and updated refs/heads/add_zlib_cat.

> git log --graph --oneline --all
* 54620a5 (HEAD -> add_zlib_cat) Added some docs
* cd54329 Added zlib_cat.py
* ea23b77 (origin/master, master) Added whatever
* 1421d17 added foo.txt
* 9b80604 added bar.txt
* 41878b4 Added more stuff
* a56b8ea Added test file

> ls
ls
bar.txt  foo.txt  stuff  test.txt whatever.txt zlib_cat.py
```

Pro Git



As Josh says, his talk covers a tiny subset of what you can learn in the amazing free book Pro Git.



Slide from Joshâ€™s talk: Git Internals and Tricks