# Version Control with Git

“The Angry Penguin“, used under creative commons licence
from Swantje Hess and Jannis Pohlmann.

Warwick RSE

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

- Linus Torvalds, Inventor of Linux (and Git)

# Part 1 - Motivations

# Overview

- Version control
  - Record changes that you make to a file or system of files
  - Allows you to keep a log of why/by whom those changes were made
  - Allows you to go back through those changes to get back to old versions
  - Help deal with merging incompatible changes from different sources
- Similar term “Source Code Management”

# Why use version control?

- "I didn't mean to do that!"
    - Can go back to before you made edits that haven't worked
- "What did this code look like when I wrote that?"
    - Can go back as far as you want to look at old versions that you used for papers or talks
- "How can I work on these different things without them interfering?"
    - Branches allow you to work on different bits and then merge them at the end

# Why use version control?

- "I want a secure copy of my code"
    - Most version control systems have a client-server functionality. Can easily store an offsite backup.
    - Many suitable free services, and can easily set up your own
- "How do I work with other people collaboratively?"
    - Most modern version control systems include specific tools for working with other people.
    - There are more powerful tools to make it even easier too

# Why use version control?

- “My funder wants me to”
    - More and more funding bodies want code to be managed and made available online
    - Version control is a good way of doing it

# What version control is not

- Not a backup
  - If you use a remote server are safe against disk failure etc
  - But other people can still wipe out your work
- Not a collaborative editing tool
  - You can merge changes from many people
  - But it is hard work, not intended to handle editing the same files
- Not magic
  - Some language awareness, has to be conservative
  - Wont fix all your problems

# How did we get here?

- Version control is literally as old as computers
- Earliest computers programmed by setting switches and “plugboards”
- People wrote down the settings that they used in lab notebooks
  - Same as they did for setting up experiments
- Starts getting more troublesome as computers get bigger

# How did we get here?

- United States National Archives Records Service punch card storage warehouse in 1959
- ~100MB / Forklift pallet
- Stored both programs and data
- Important programs would be kept in archives and repunched when changed
- Old versions kept for some time

# How did we get here?

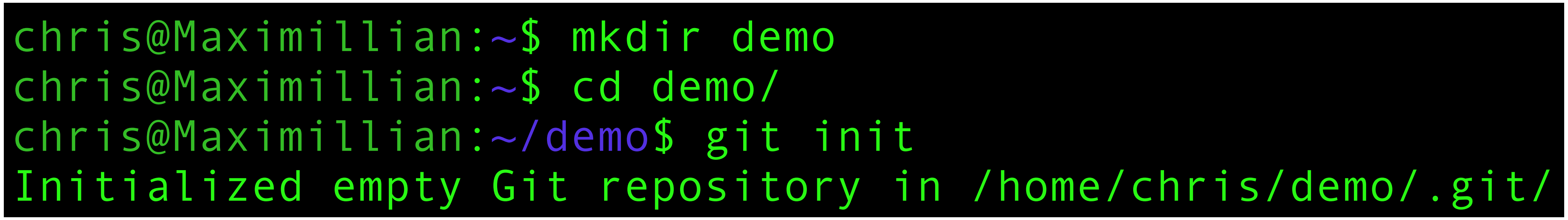
- 1982 - Revision Control System (RCS - various commands)
- 1990 - Concurrent Versions System (CVS)
- 2000 - Subversion (SVN)
- 2000 - Bitkeeper (BK)
  - 2005 - Git (GIT)
- Others (Mercurial, GNU Arch, ArX etc.)

# Part 2 - Basic Git

# Repositories

- The basic idea of git version control is that you create a **repository** that holds files and directories
- Repositories are created in a specific directory and all files and directories within a repository must be in that directory or a subdirectory of it
  - You cannot create a repository within a repository
    - Files can only be in one repository at a time
    - Be careful about creating a git repository where you don't intend to!

# Repositories

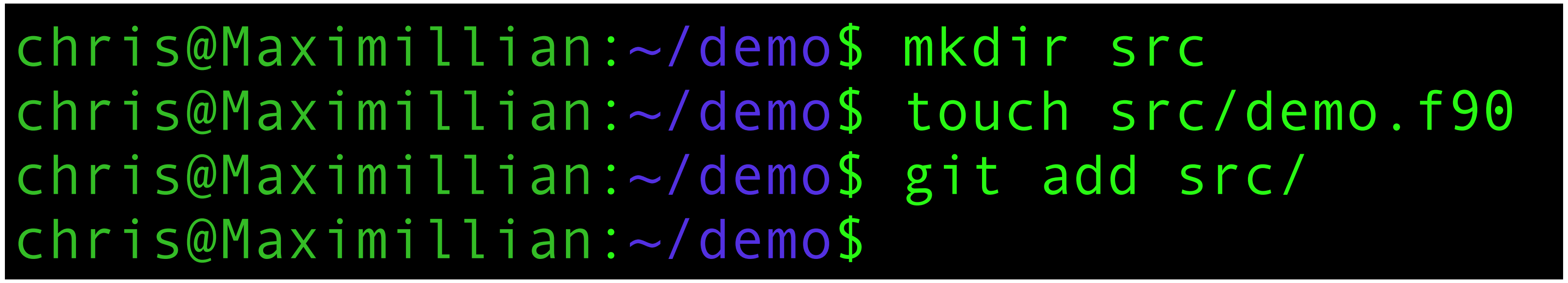


```
chris@Maximillian:~$ mkdir demo
chris@Maximillian:~$ cd demo/
chris@Maximillian:~/demo$ git init
Initialized empty Git repository in /home/chris/demo/.git/
```

# Repositories

- It is very important to note that repositories are **held** in directories, they aren't directories
  - There can be files and directories within the directory that holds a git repository that are not in the repository
- You manually add files and directories that you want in the repository
  - If you add a file in a directory then the directory and just that file is added
  - If you add a directory directly then all files in the directory are added

# Adding



```
chris@Maximillian:~/demo$ mkdir src
chris@Maximillian:~/demo$ touch src/demo.f90
chris@Maximillian:~/demo$ git add src/
chris@Maximillian:~/demo$
```

# Staging and Commits

- Git records changes as a series of **commits**
    - Each **commit** represents a state of the repository that can be recovered in future
- In general you might not want every change that you have made to every file to be part of a commit
    - You flag files when you want to include them in the next **commit** by **add**ing them again
    - Formally this adds them to the **staging area**

# Staging and Commits

- It’s important that there are four states that a file (or directory) can be in
  - Untracked - Not in the repository, can be **add**ed
  - Up to date - file is in repository and is in the same state as the last **commit**
  - Unstaged - file is in repository, is in a different state to the stored version but is not flagged as being part of the next **commit**
  - Staged - file has been added to the **staging area** as being part of the next **commit**
- A single file can have both staged and unstaged changes if you have **add**ed the file and then changed it

# Staging and Commits

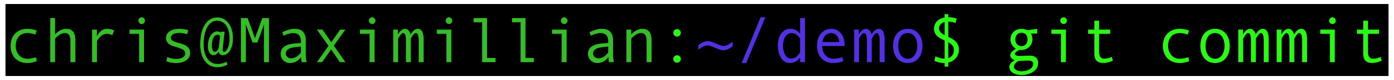
- Note that the state of the file is recorded when you **add** it

  - Even if you make further changes they will not be contained in the next **commit**

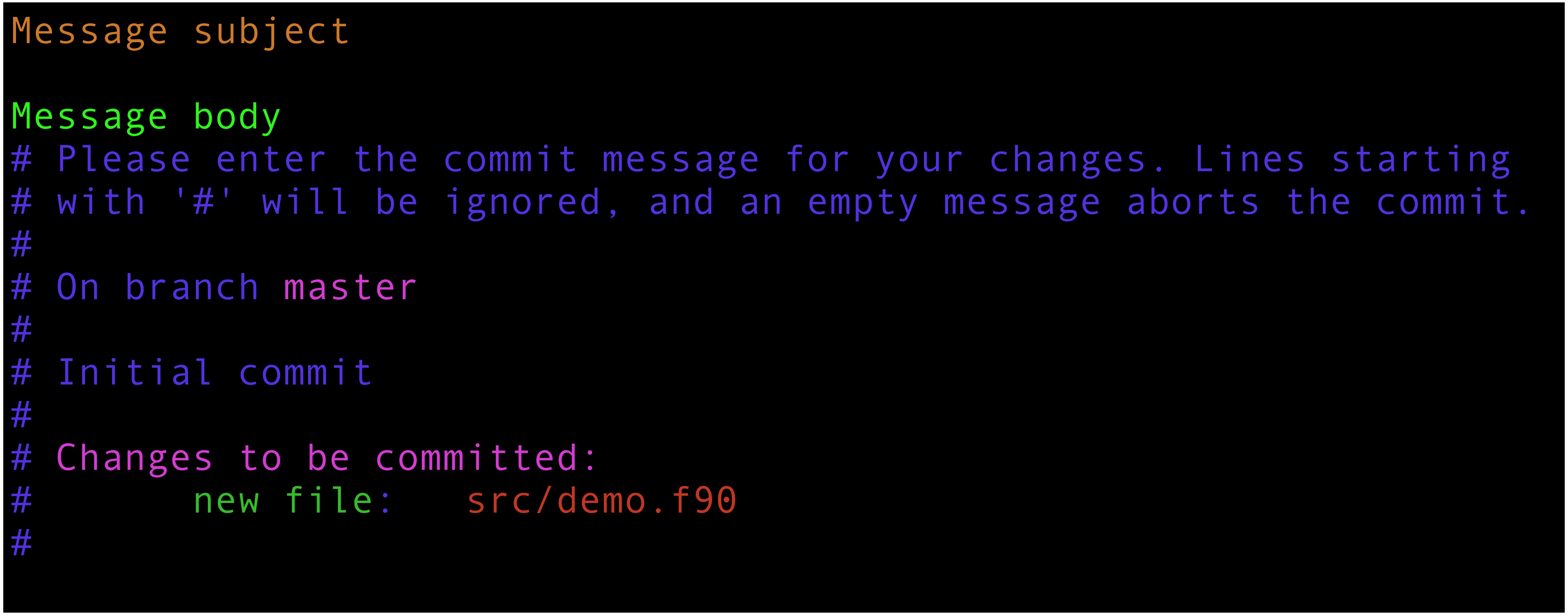
  - You have to **add** the file again to record further changes

- To remove a file from the staging area you **reset** it

# Staging and Commits



```
chris@Maximillian:~/demo$ git commit
```



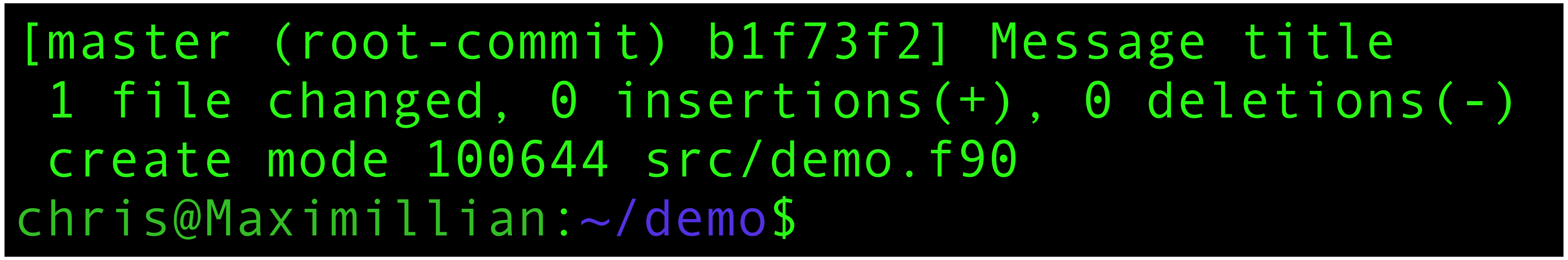
```
Message subject

Message body
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   src/demo.f90
#
```

# Git commit message

```
Message subject

Message body
```

- First line is the subject. Keep it to <= 50 characters
- Second line should be blank
- Subsequent lines are the “body” of the message
- Should limit body lines to <=72 characters
- As many as you want, but be concise

# After writing message



```
[master (root-commit) b1f73f2] Message title
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 src/demo.f90
chris@Maximillian:~/demo$
```

- When you save the file and exit your editor git will give you a summary of what's just happened
  - In this case, it's created the file "demo.f90" as I wanted it to
- If you quit your editor without saving this cancels the commit
- "demo.f90" is now under version control, and I can always get back to this version

# Basic Workflow

1. "**git init**"

2. Create files, make changes etc

3. "**git add {filenames}**" or "**git add .**" to add everything

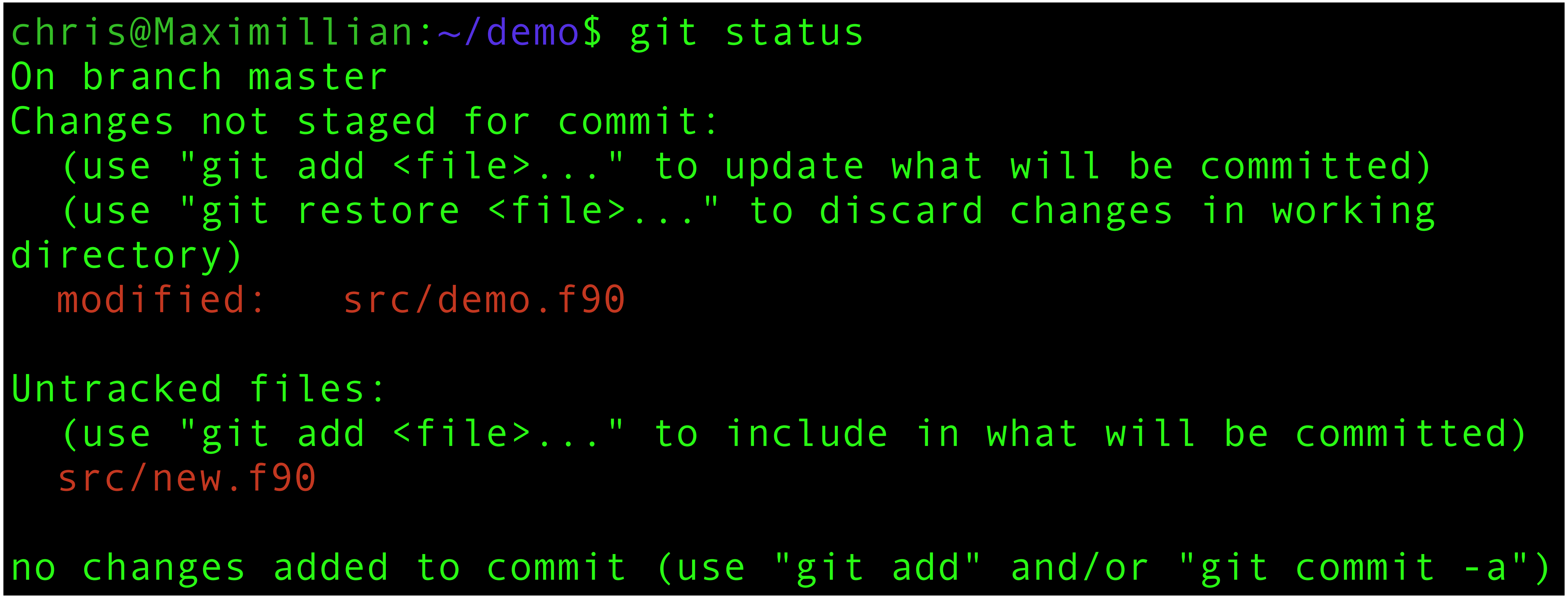
4. "**git commit**"

5. Write a useful commit message

6. Return to step 2

# Part 3 - Additional Commands

# git status

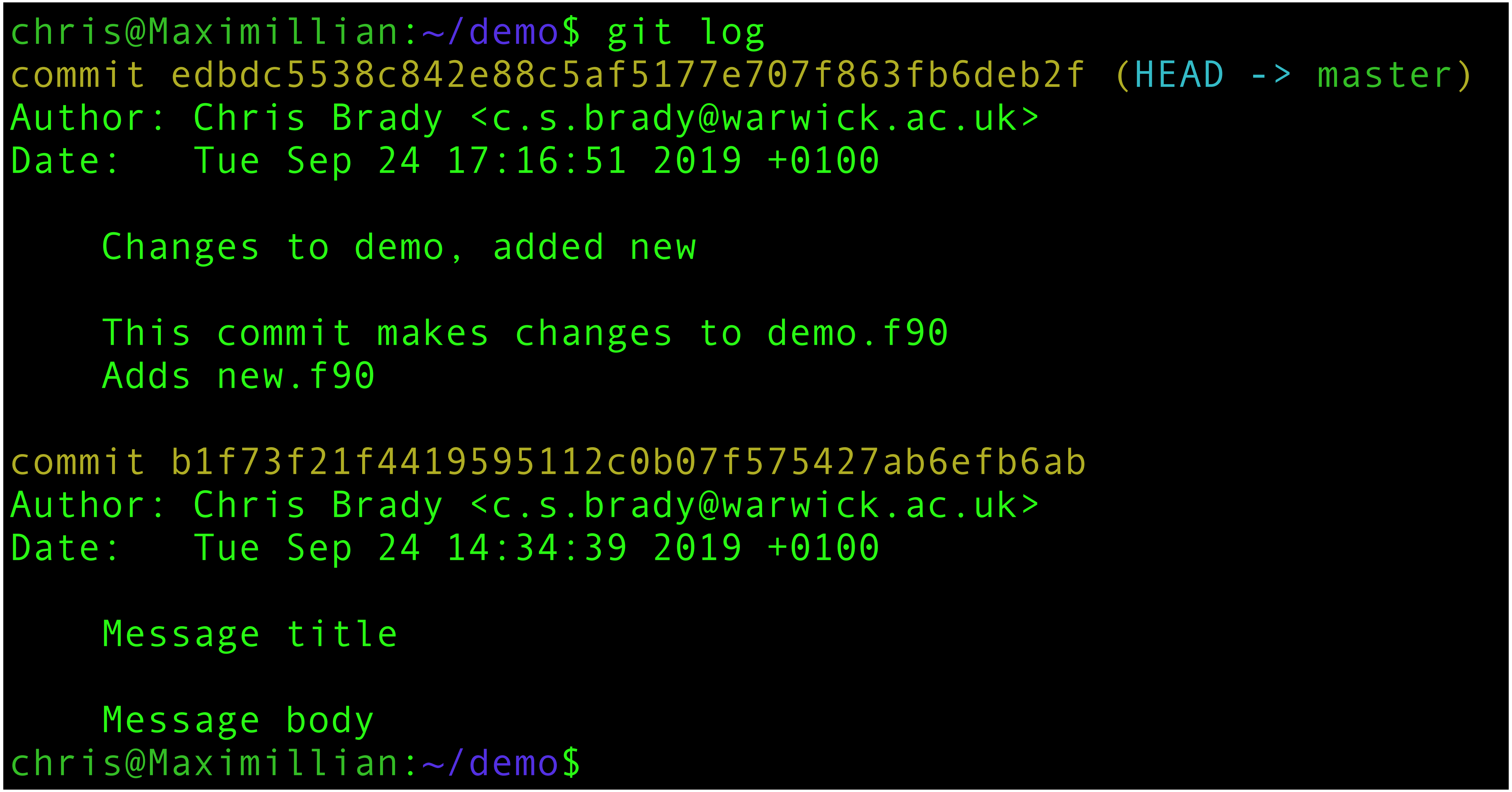


```
chris@Maximillian:~/demo$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
  modified:   src/demo.f90

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  src/new.f90

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

- Gives information about the current state of the repository

- This example shows one file with changes and one file that is in the directory but not in the repository

# git log



```
chris@Maximillian:~/demo$ git log
commit edbdc5538c842e88c5af5177e707f863fb6deb2f (HEAD -> master)
Author: Chris Brady <c.s.brady@warwick.ac.uk>
Date:   Tue Sep 24 17:16:51 2019 +0100

    Changes to demo, added new

    This commit makes changes to demo.f90
    Adds new.f90

commit b1f73f21f4419595112c0b07f575427ab6efb6ab
Author: Chris Brady <c.s.brady@warwick.ac.uk>
Date:   Tue Sep 24 14:34:39 2019 +0100

    Message title

    Message body
chris@Maximillian:~/demo$
```

- Shows the list of commits. Gives unique commit id for each

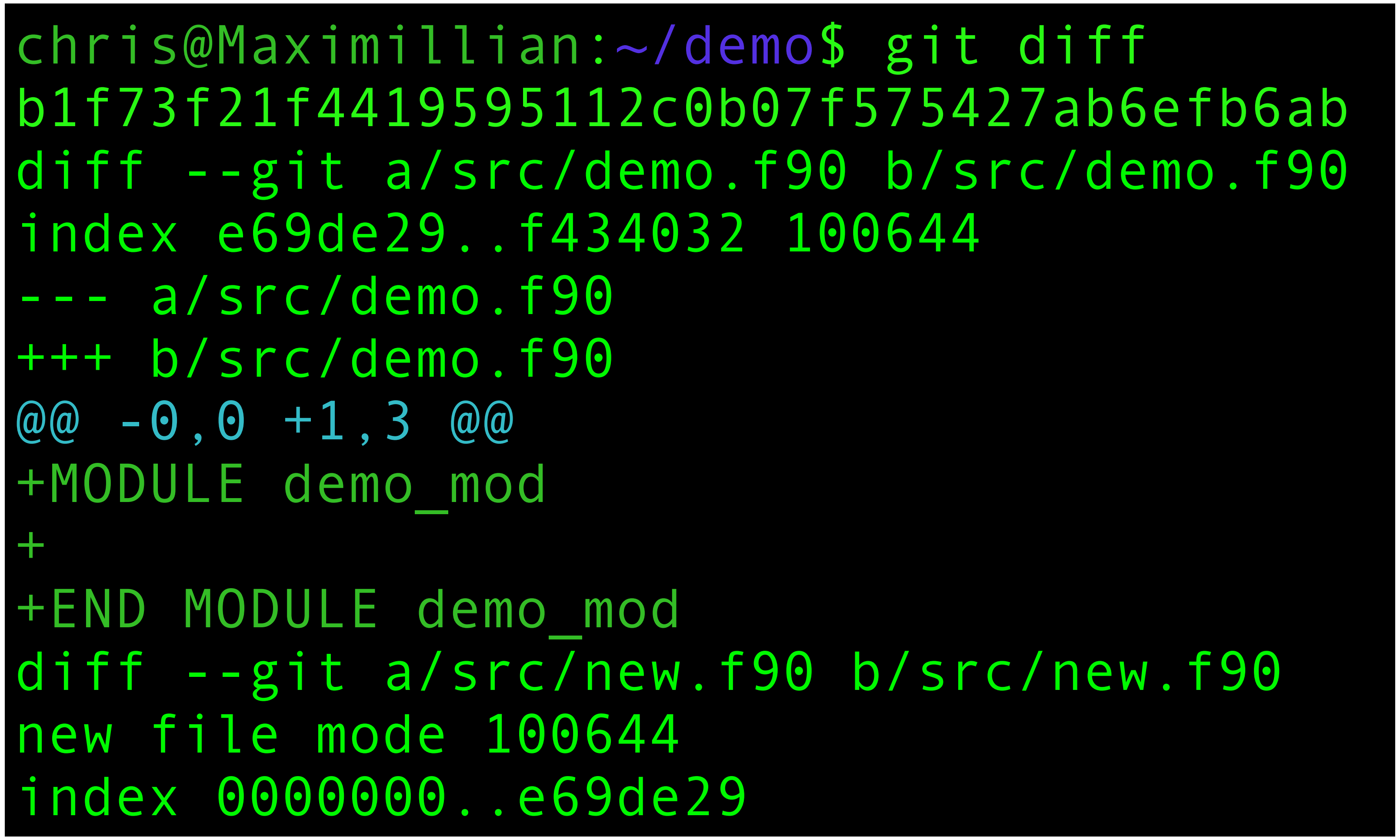
# git diff

- Using the command "**git diff**" followed by a commit ID shows you the changes between the current state of the code and the one referred to in the by the commit ID
  - If you don't specify a commit ID it shows the difference between the current state and the last commit
  - If you specify two commit IDs then it shows the differences between the commits
- Adding a list of filenames at the end allows you to see the differences in only specific files

# git diff

- The result of the command is in “git-diff” format
  - Lines with a + have been added since the specified commit
  - Lines with a - have been removed
  - Lines without a symbol are only there for context and are unchanged

# git diff output



```
chris@Maximillian:~/demo$ git diff
b1f73f21f4419595112c0b07f575427ab6efb6ab
diff --git a/src/demo.f90 b/src/demo.f90
index e69de29..f434032 100644
--- a/src/demo.f90
+++ b/src/demo.f90
@@ -0,0 +1,3 @@
+MODULE demo_mod
+
+END MODULE demo_mod
diff --git a/src/new.f90 b/src/new.f90
new file mode 100644
index 0000000..e69de29
```

- Example git diff output
- Added new lines to demo.f90
- new.f90 is a new file

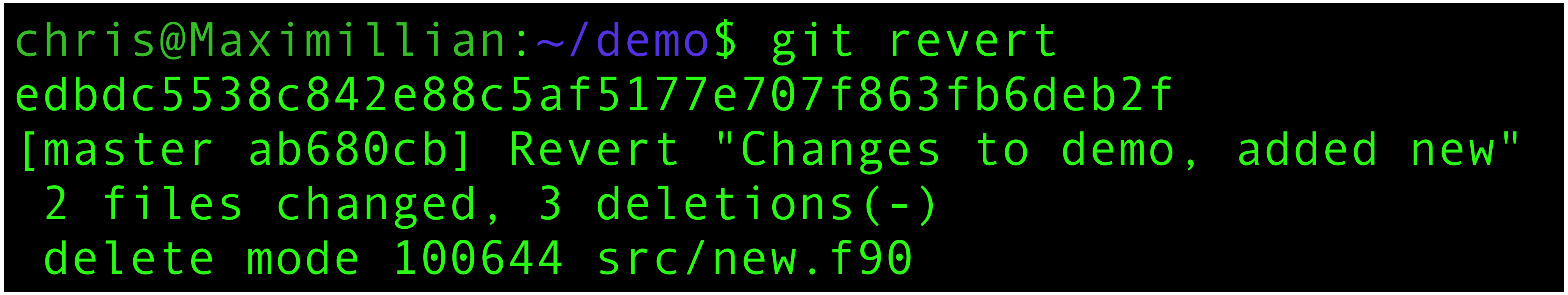
# git apply

- Diff output is a standard format
  - Can share it as file called a “patch”
    - git diff > output.patch
  - Apply patches with “**git apply {filename}**”
  - Can in theory apply to different code state, not always smoothly

# Reverting to undo bad changes

- Undoing changes in git can be a mess
  - Distributed system, so if code has ever been out of your control you can’t just go back
  - Reverts are in general simply changes that put things back to how they used to be
  - Git log will show original commits and reverts
- Command is “**git revert**”

# git revert



```
chris@Maximillian:~/demo$ git revert
edbdc5538c842e88c5af5177e707f863fb6deb2f
[master ab680cb] Revert "Changes to demo, added new"
 2 files changed, 3 deletions(-)
 delete mode 100644 src/new.f90
```

- Lots of flexibility, but mostly you want to do
  - **git revert {lower_bound_commit_id}..{upper_bound_commit_id}**
- Lower bound is exclusive
- Upper bound is inclusive

# git revert

- When git revert operates, it creates a new commit undoing each commit that you want to revert

- You get an editor pop-up for each with a default message that says

  - **Revert "{original commit message}"**

  - No real need to change them

# Incremental Changes

- Git works by recording **changes** rather than entire states
  - You can get back to a known state by either playing states forward from the initial check in or undoing states from the end
  - There isn't just a single state that can be returned to
- That's why it is hard to edit the history of a git repository
  - If you go back in history and remove a part of the history then the future changes may make no sense
  - You can fix this but it is a lot of work, hence the general idea is that reverting works by adding new commits that undo changes rather than changing history

# Part 4 - Branches

# git branch

- If you are working on multiple features then branches are useful

- Branches are code versions that git keeps separate for you

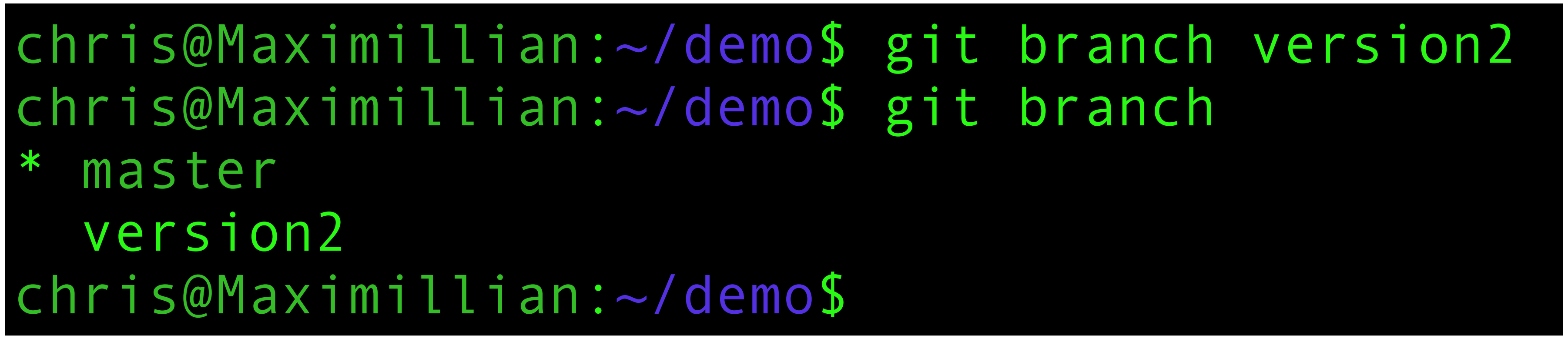
- Changes to one branch do not affect any other

- There is a default branch called “master” created when you create the repository

- A git repository is always working on one branch or another (sometime a temporary branch, but ignore this here)

- Adds and commits are always to the branch that you are working on

# git branch



```
chris@Maximillian:~/demo$ git branch version2
chris@Maximillian:~/demo$ git branch
* master
  version2
chris@Maximillian:~/demo$
```

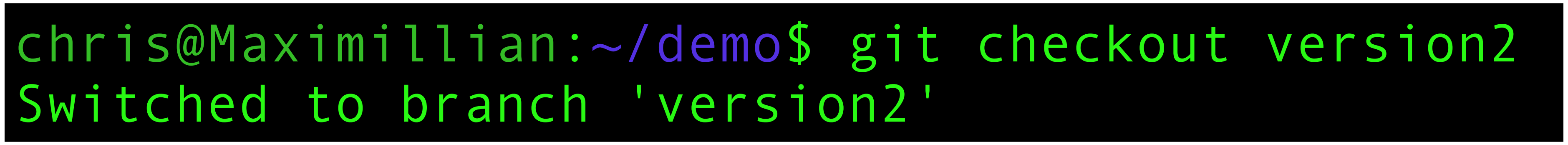
- To create a branch, just type “**git branch {name}**”

- A new branch is created based on the last commit in the branch that you are on

- Simply creating a branch does not move you to it. You are still exactly where you are before

- You can check what branch you are on by typing “**git branch**” with no parameters

# git checkout

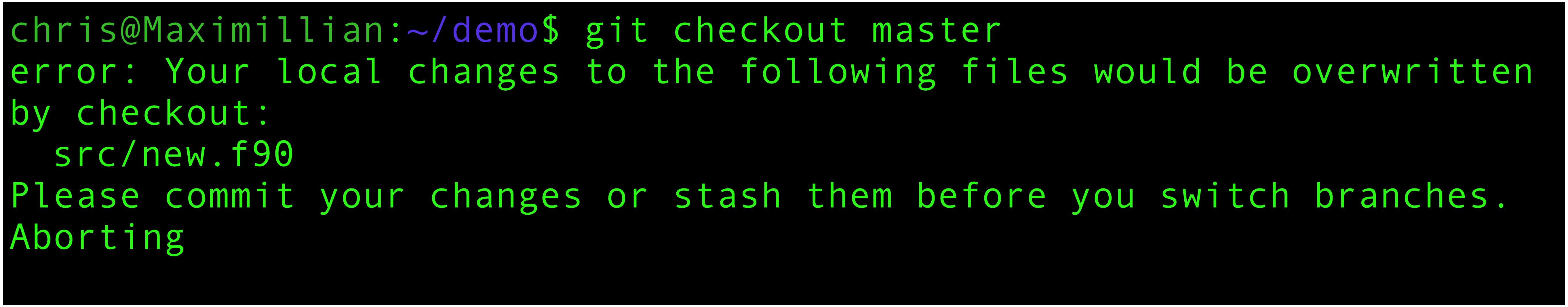


```
chris@Maximillian:~/demo$ git checkout version2
Switched to branch 'version2'
```

- To move between branches, you use “**git checkout {branch_name}**”

- This will tell you that it has switched to the named branch if it has managed to do so

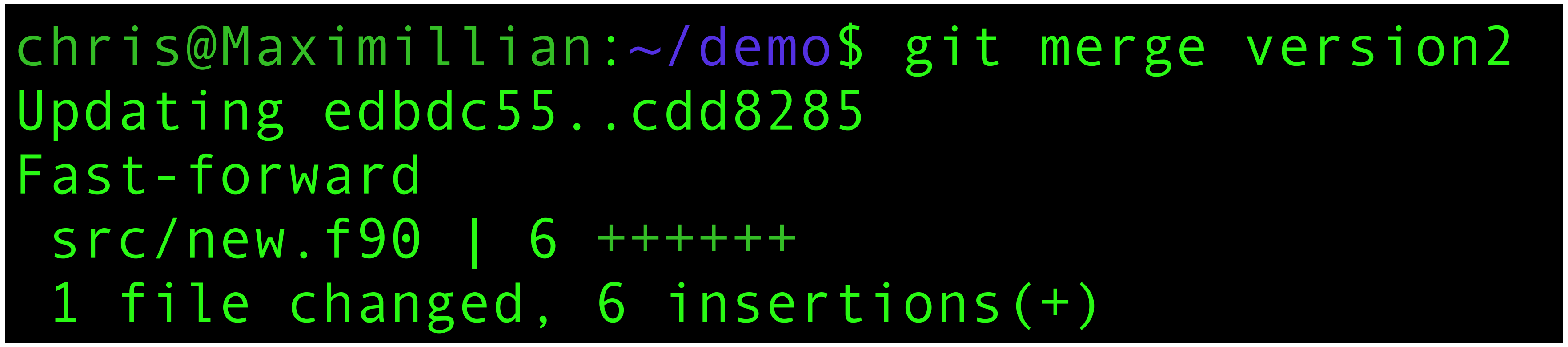
# Changing branches



```
chris@Maximillian:~/demo$ git checkout master
error: Your local changes to the following files would be overwritten
by checkout:
  src/new.f90
Please commit your changes or stash them before you switch branches.
Aborting
```

- Once branches have changed relative to each other you can no longer carry changes between them
- If you make changes in a branch and then try to move to another branch, without committing the changes you will get an error message
- Either
  - commit the changes in the branch that you are on
  - use git-stash (https://git-scm.com/docs/git-stash)

# Bringing branches back



```
chris@Maximillian:~/demo$ git merge version2
Updating edbdc55..cdd8285
Fast-forward
 src/new.f90 | 6 ++++++
 1 file changed, 6 insertions(+)
```

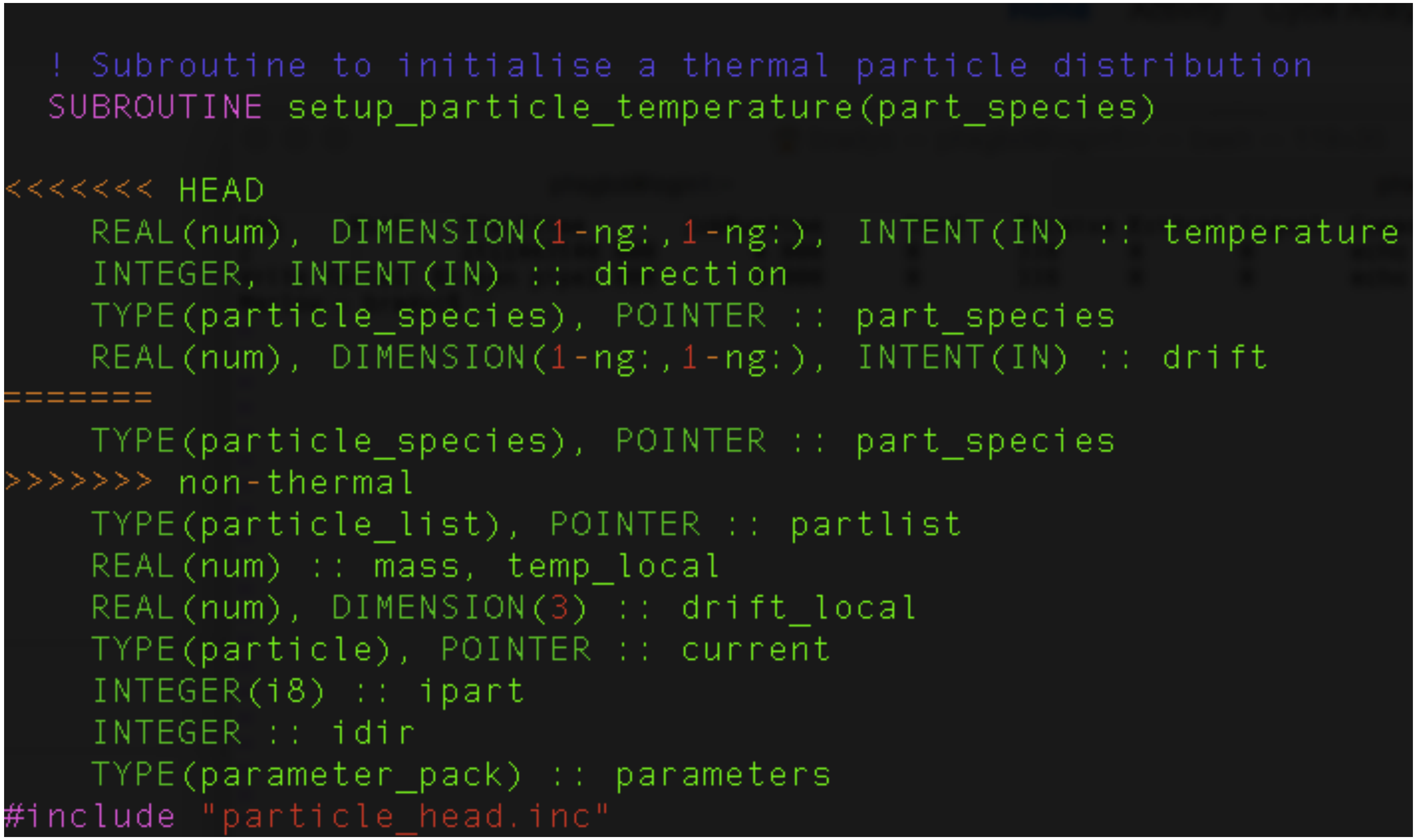
- If you're using branches to develop features (a very common way of working) you'll want to bring them back together to form a single version with all the features

- Termed "merging"

- "**git merge {other_branch_name}**" brings the other branch's content into this branch

- If you're lucky, you'll see what's at the top and the merge is automatic (fast-forward merge)

# Manual Merge



```
  ! Subroutine to initialise a thermal particle distribution
  SUBROUTINE setup_particle_temperature(part_species)

<<<<<<< HEAD
    REAL(num), DIMENSION(1-ng:,1-ng:), INTENT(IN) :: temperature
    INTEGER, INTENT(IN) :: direction
    TYPE(particle_species), POINTER :: part_species
    REAL(num), DIMENSION(1-ng:,1-ng:), INTENT(IN) :: drift
=======
    TYPE(particle_species), POINTER :: part_species
>>>>>>> non-thermal
    TYPE(particle_list), POINTER :: partlist
    REAL(num) :: mass, temp_local
    REAL(num), DIMENSION(3) :: drift_local
    TYPE(particle), POINTER :: current
    INTEGER(i8) :: ipart
    INTEGER :: idir
    TYPE(parameter_pack) :: parameters
#include "particle_head.inc"
```

- If git can't work out how to combine the changes between the versions then it'll put diff markers into the file to say what's changed and where

# Manual Merge

- You have to go through and remove these markers, leaving a single working version of the code

- Commit the finished version using "**git add**" and "**git commit**" as normal (or "**git merge --continue**" in newer versions of git)

- There are tools to help, but it's never fun

# Part 5 - Remotes

# Git remote server

- Git is a distributed, networked version control system.
- Has commands to control this
- Collectively called “**git remote**” commands
- You can clone a remote repository and it remembers that it’s attached to that remote
- A local repository can be told that it’s a local copy of an remote repository
- There may be access controls on a remote server and you will be asked for a username and password. You should know these if you need them

# git remote add

```
chris@Maximillian:~/demo$ git remote add upstream https://github.com/csbrady-warwick/DemoRepo.git
```

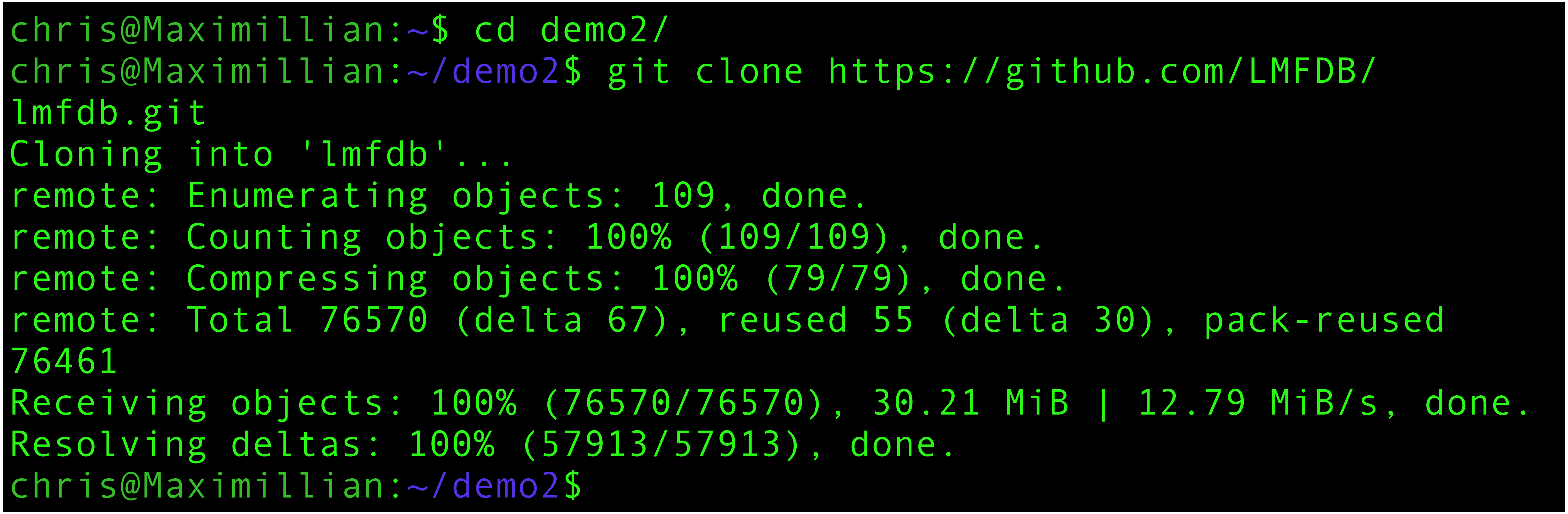
- You can have multiple named remote repositories “connected” to a single local repository

- Each one has a unique name

  - The default remote repository is called “origin”

  - “upstream” is quite common for when you are tracking another repository

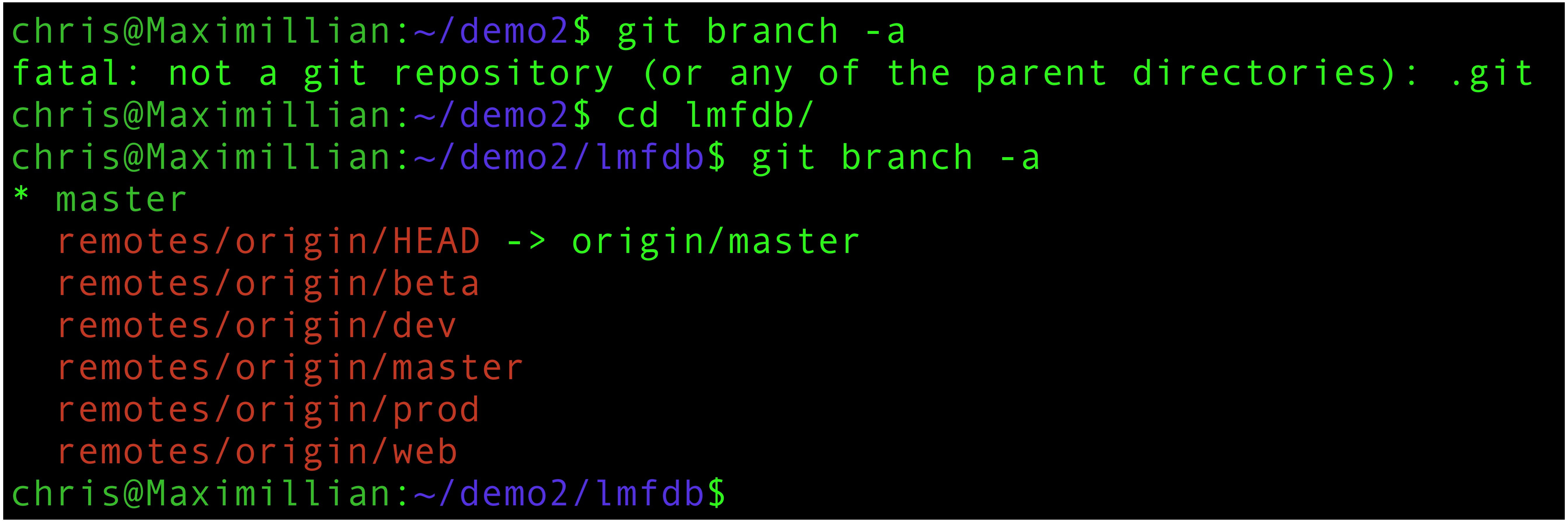
# git clone



```
chris@Maximillian:~$ cd demo2/
chris@Maximillian:~/demo2$ git clone https://github.com/LMFDB/
lmfdb.git
Cloning into 'lmfdb'...
remote: Enumerating objects: 109, done.
remote: Counting objects: 100% (109/109), done.
remote: Compressing objects: 100% (79/79), done.
remote: Total 76570 (delta 67), reused 55 (delta 30), pack-reused
76461
Receiving objects: 100% (76570/76570), 30.21 MiB | 12.79 MiB/s, done.
Resolving deltas: 100% (57913/57913), done.
chris@Maximillian:~/demo2$
```

- To clone a remote repository, you need to have a URL for the remote server
    - This is a github repository, so big green button
- Command is then "**git clone {remote_url}**"
- Creates new functioning local repository in a subdirectory of where you ran the command

# git branch -a



```
chris@Maximillian:~/demo2$ git branch -a
fatal: not a git repository (or any of the parent directories): .git
chris@Maximillian:~/demo2$ cd lmfdb/
chris@Maximillian:~/demo2/lmfdb$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/beta
  remotes/origin/dev
  remotes/origin/master
  remotes/origin/prod
  remotes/origin/web
chris@Maximillian:~/demo2/lmfdb$
```

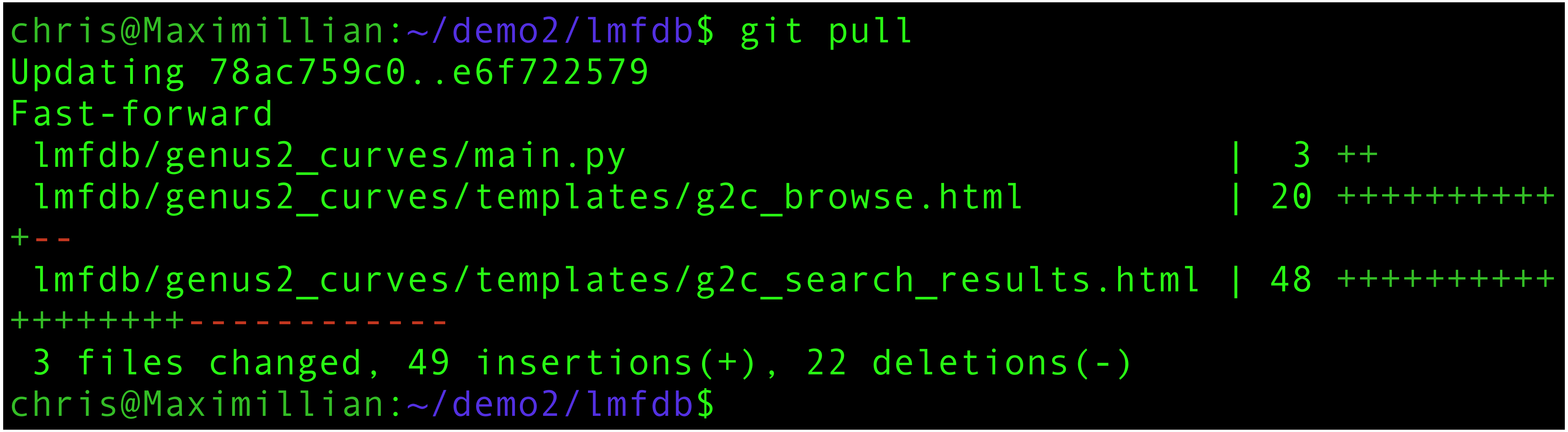
- Running “**git branch -a**” also tells you about remote branches

- Once again, there exists a “master” branch, which is now a local reference to “remotes/origin/master”

- You do not by default have copies of all of those remote branches

- You get them using “**git checkout**”

# git pull



```
chris@Maximillian:~/demo2/lmfdb$ git pull
Updating 78ac759c0..e6f722579
Fast-forward
 lmfdb/genus2_curves/main.py                           |  3 ++
 lmfdb/genus2_curves/templates/g2c_browse.html         | 20 ++++++++++
+--
 lmfdb/genus2_curves/templates/g2c_search_results.html | 48 ++++++++++
++++++++------------
 3 files changed, 49 insertions(+), 22 deletions(-)
chris@Maximillian:~/demo2/lmfdb$
```

- If you have a copy of a repository that is less recent than the version on the remote server you can update it using “**git pull**”

- Pull is a per branch property. You are pulling the specific branch that you are on

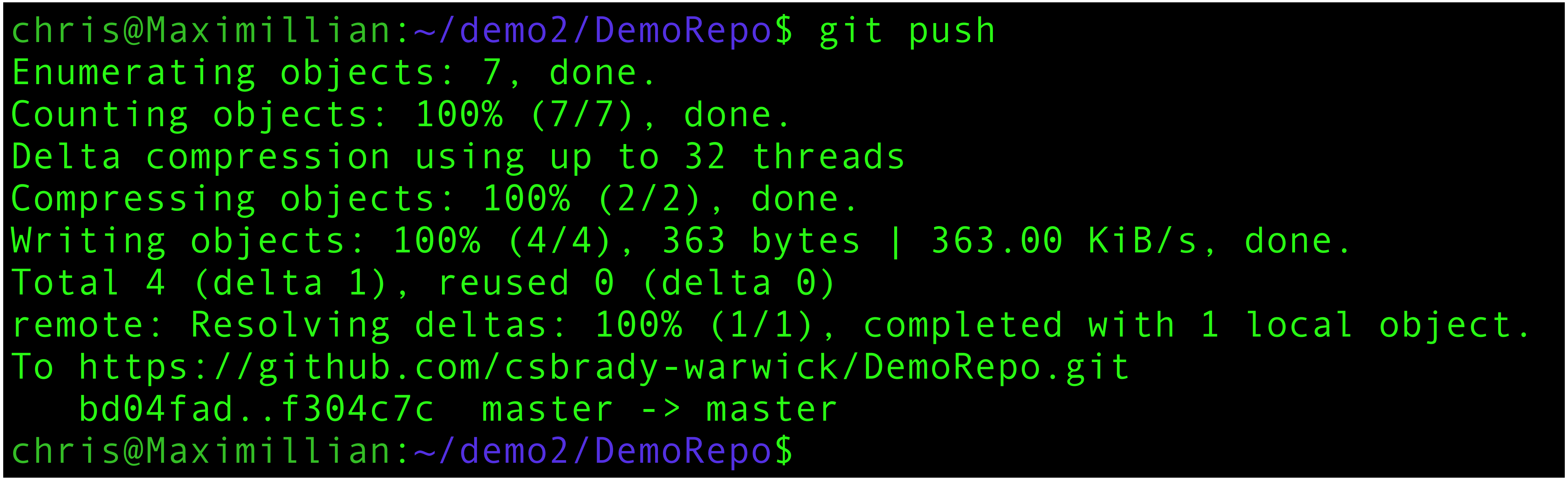
# git fetch / git merge

- Behind the scenes, “**git pull**” is a combination of
    - “**git fetch**” - pull data from remote server
    - “**git merge**” - merge the changes in that data
- All of the problems that can happen in a merge
- Added difficulty that now can be changes due to other developers

# git push

- The opposite of pull
    - Pushes your changes to a code to the remote server
    - Will not generally work unless git can automatically merge those changes with the version on the server
        - “**git pull**” then “**git push**”
- Be careful! If not your repository people might not like you doing it
    - Shouldn’t be able to if you shouldn’t

# git push



```
chris@Maximillian:~/demo2/DemoRepo$ git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 32 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 363 bytes | 363.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/csbrady-warwick/DemoRepo.git
   bd04fad..f304c7c  master -> master
chris@Maximillian:~/demo2/DemoRepo$
```

- If it works, should see something like that

- Push can be a much more complicated command if you want to push different local branches or the name of the local branch and the remote branch are different

- Read the documentation

# Github

- GITHUB IS NOT GIT!
- By far the most popular public remote git server platform at the moment
- Easy to use
    - Gives a lot of help for setting up remote repositories
    - Same basic stuff that we've talked about here
- Provides a lot of nice extra features for developers
    - Support forums
    - Issue trackers

# The End