# A brief introduction to git

*Michael Hannon*

*2016-01-22*

## Contents

## References

https://git-scm.com/documentation

https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

http://shop.oreilly.com/product/0636920022862.do

# What is `git`?

`git` is a version-control system for software. What's that mean?

## About version control

A version-control system is a system that records changes to a file or set of files over time so that you can recall specific versions later.

There is some discussion of the concept at, for instance:

https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

## Features of `git`

There are many version-control systems in the wild. What are the distinguishing features of `git`?

From the O'Reilly reference, here are the desirable features of `git`:

- Facilitate Distributed Development
- Scale to Handle Thousands of Developers
- Perform Quickly and Efficiently
- Maintain Integrity and Trust
- Enforce Accountability (`git blame...`)
- Immutability
- Atomic Transactions
- Support and Encourage Branched Development (and merging)
- Complete Repositories
- A Clean Internal Design
- Be Free, as in Freedom [and beer]

## The `git` repository model

Some version-control systems operate by keeping a master copy of a project, with changes in the project indicated by a set of of differences:

Note that, in general, this requires you to have a connection to the server that stores the master copy.

On the other hand, `git` stores data more like a series of snapshots of *the entire project*:

(The dashed lines indicate that `git` is storing only a pointer to an unchanged file, rather than duplicating the entire file.) This approach means that operations on a git repository can be and usually are entirely local, independent of a connection to a master server.

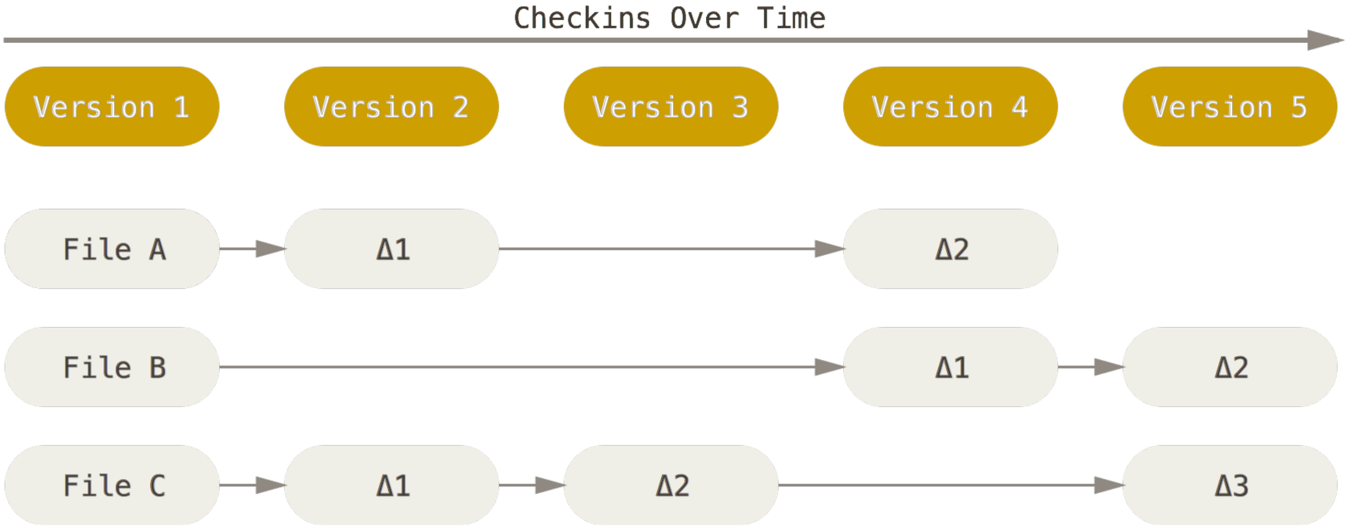


Figure 1: The “non-git” approach to storing changes to a project

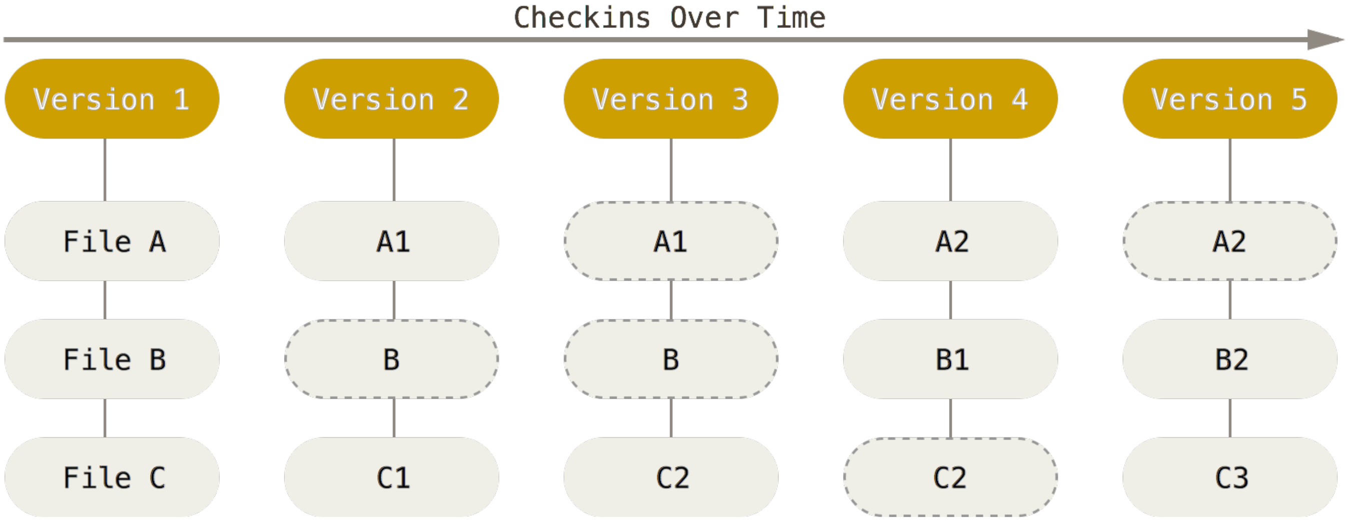


Figure 2: Storing data as snapshots of the project over time

# Installing git

Available via your package manager in linux:

```
sudo apt-get install git  ## Debian-related systems (Ubuntu)
sudo dnf install git      ## Redhat-related systems (Fedora)
```

or in MacOS:

```
brew install git          ## Using `homebrew` package mgr
```

Or (for `Windows` [or other systems]) download directly from the `git` site:
https://git-scm.com/downloads

# Using git

## The command line

### Summary of usage

Type `git --help` to see common subcommands and options:

```
git --help
```

```
## usage: git [--version] [--help] [-C <path>] [-c name=value]
##            [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
##            [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
##            [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
##            <command> [<args>]
##
## The most commonly used git commands are:
##    add        Add file contents to the index
##    bisect     Find by binary search the change that introduced a bug
##    branch     List, create, or delete branches
##    checkout   Checkout a branch or paths to the working tree
##    clone      Clone a repository into a new directory
##    commit     Record changes to the repository
##    diff       Show changes between commits, commit and working tree, etc
##    fetch      Download objects and refs from another repository
##    grep       Print lines matching a pattern
##    init       Create an empty Git repository or reinitialize an existing one
##    log        Show commit logs
##    merge      Join two or more development histories together
##    mv         Move or rename a file, a directory, or a symlink
##    pull       Fetch from and integrate with another repository or a local branch
##    push       Update remote refs along with associated objects
##    rebase     Forward-port local commits to the updated upstream head
##    reset      Reset current HEAD to the specified state
##    rm         Remove files from the working tree and from the index
##    show       Show various types of objects
##    status     Show the working tree status
##    tag        Create, list, delete or verify a tag object signed with GPG
##
## 'git help -a' and 'git help -g' lists available subcommands and some
## concept guides. See 'git help <command>' or 'git help <concept>'
## to read about a specific subcommand or concept.
```

(Actually, typing `git` alone would produce the same output.)

### Help for git subcommands

Each `git` subcommand has its own help section:

```
git commit --help | head
```

```
## GIT-COMMIT(1)                     Git Manual                     GIT-COMMIT(1)
##
##
##
## NAME
##        git-commit - Record changes to the repository
##
## SYNOPSIS
##        git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
##                   [--dry-run] [(-c | -C | --fixup | --squash) <commit>]
```

(Output truncated for brevity)

## Creating a repository

There are two ways to make a `git` repository (collection of files and data):

- Create your own from scratch
- Copy (clone) from another location

We'll look at each of those in turn.

## Creating your own repository

### First, create a directory, `test` (name is arbitrary)

```
if [ -d ~/test ]; then
    \rm -rf ~/test
fi

mkdir ~/test    ## Create a sub-dir `test` in my login directory
```

### Next, make a file in the `test` directory and make it executable

```
cd ~/test        ## Go to the directory just created

cat <<EOF > hw.py
#!/usr/bin/python

print("Hello from Python")
EOF

chmod +x hw.py    ## make the file executable
```

### Now look at the file and run it, just to verify it's there:

```
cd ~/test

cat hw.py  ## display the contents of the file

echo ""

./hw.py    ## run the file
```

```
## #!/usr/bin/python
##
## print("Hello from Python")
##
## Hello from Python
```

### Now make the directory into a git repository

```
cd ~/test
git init
```

```
## Initialized empty Git repository in /home/mike/test/.git/
```

### Take a look at the “hidden” .git directory and its contents

```
cd ~/test

ls -a
echo "########"
ls .git
```

```
## .
## ..
## .git
## hw.py
## ########
## branches
## config
## description
## HEAD
## hooks
## info
## objects
## refs
```

### Starting to exercise `git`

Now where are we in `git` land?

```
cd ~/test
git status
```

```
## On branch master
##
## Initial commit
##
## Untracked files:
##   (use "git add <file>..." to include in what will be committed)
##
```

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

So `git` has "noticed" our `hw.py` file ("untracked"), but we haven't told it what to do with the file. Note the helpful hint:

```
(use "git add" to track)
```

Before we proceed, let's back up and look at how `git` handles files.

Our file, `hw.py` is currently in the *Working Directory*. Now we want to let `git` know that we're seriously interested in this file, which we do by "adding" the file to the *Staging Area*, a.k.a, the *Index*. The point here is that we're required to make a conscious decision as to what files `git` should track. For instance, we probably do *not* want `git` to track the MP3 file that happened to wind up in our directory by mistake.

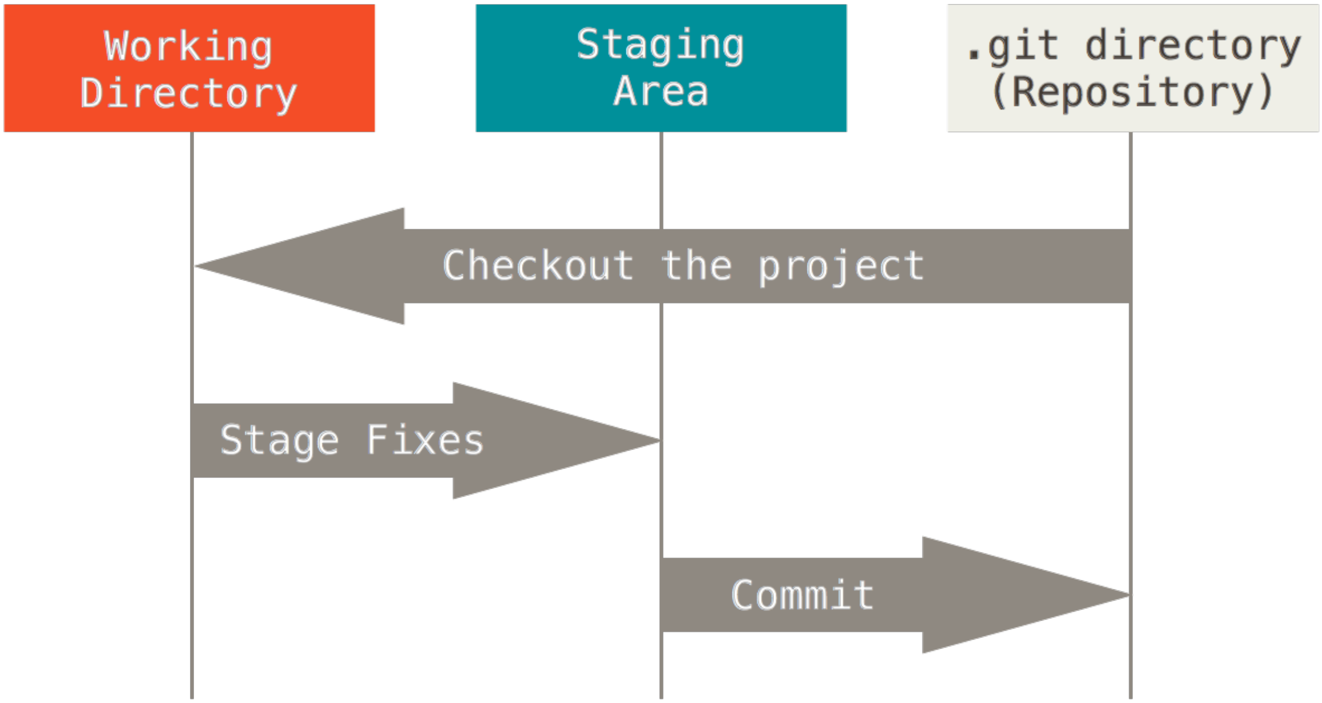


Figure 3: Sections of a `git` project

### Add a file to the repository

Here we take `git` up on the suggestion to "add" the file.

```
cd ~/test
git add hw.py
git status
```

```
## On branch master
##
## Initial commit
##
## Changes to be committed:
##   (use "git rm --cached <file>..." to unstage)
##
##  new file:   hw.py
```

OK, we've got `git` paying real attention to our `hw.py` file. Now let's assume we've tweaked and honed our code until it shines like the top of the Chrysler building, and we won't be making changes to it in the near future. We're now ready to "commit" the file to the repository, which means that `git` will now not only be aware of the file, but it will store the contents in a "secret" place, and will store as well various pieces of meta-information about the file.

As a preliminary to the next step, we do some bookkeeping to keep `git` from asking and/or guessing about this stuff:

```
## First some bookkeeping
git config --global user.name "Michael Hannon"
git config --global user.email "jmhannon.ucdavis@gmail.com"
```

The "global" configuration options reside in a file called `.gitconfig` (note the initial period) in your login directory:

```
cd    ## goes to login directory
cat .gitconfig
```

```
## [user]
##  name = Michael Hannon
##  email = jmhannon.ucdavis@gmail.com
## [push]
##  default = simple
```

Note that it's also possible to have a *per-repository* `.gitconfig` file, in case you want to have some `git` options that apply specifically to the given project. (And the settings in the per-repository `.gitconfig` file override those in the global file.)

All set. Commit the file to the repository:

```
cd ~/test
git commit -m "Initial contents of our hello-world file" \
           --author="Michael Hannon <jmhannon.ucdavis@gmail.com>"
git status
```

```
## [master (root-commit) 4c55e1a] Initial contents of our hello-world file
##  1 file changed, 3 insertions(+)
##  create mode 100755 hw.py
## On branch master
## nothing to commit, working directory clean
```

Note that if you omit the commit message (the `-m` option), git will open an editor for you in which to type the message. You can choose the editor to use by setting the `GIT_EDITOR` environment variable. See the O'Reilly reference for details.

### Further work on our project

Time passes. You think of a modification you might make to the `hw.py` file, and you think of another file that might be useful in this project. First append a line to the `hw.py` file:

```
cd ~/test
echo -e "print('From Davis, in the heart of California.')" >> hw.py
```

Second, make a file to do some calculations:

```
cd ~/test

cat <<EOF > calc.py
#!/usr/bin/python

import sys

prog, arg1, arg2, op = sys.argv
val1 = float(arg1)
val2 = float(arg2)

if op == "add": print (val1 + val2)
else:
    print ("Whut?")
EOF
```

We've created the script. We'll make it executable and run some examples, just to be sure it's working:

```
cd ~/test

chmod +x calc.py    ## make the file executable
./calc.py 2 3 add
./calc.py 42 314 logarithm
```

```
## 5.0
## Whut?
```

OK, we've got a modified file and a new file. The next thing to do is to stage the files and then do a commit:

```
cd ~/test
git add hw.py calc.py
git commit -m "Better hello-world message; new calculation module."
git status
```

```
## [master 561e327] Better hello-world message; new calculation module.
##  2 files changed, 12 insertions(+)
##  create mode 100755 calc.py
## On branch master
## nothing to commit, working directory clean
```

That was easy enough. Let's review our changes. First, look at the log of the commits we've made so far:

```
cd ~/test

git log
```

```
## commit 561e327e8423ee37e99b5b750c1e1670010273a9
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:01 2016 -0800
##
##     Better hello-world message; new calculation module.
##
## commit 4c55e1ae3865a3223ee1044fd6dc8d5c93b5f31a
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:01 2016 -0800
##
##     Initial contents of our hello-world file
```

Some further introspection:

```
cd ~/test

git diff HEAD HEAD^
```

```
## diff --git a/calc.py b/calc.py
## deleted file mode 100755
## index 9df65e7..0000000
## --- a/calc.py
## +++ /dev/null
## @@ -1,11 +0,0 @@
## -#!/usr/bin/python
## -
## -import sys
## -
## -prog, arg1, arg2, op = sys.argv
## -val1 = float(arg1)
## -val2 = float(arg2)
## -
## -if op == "add": print (val1 + val2)
## -else:
## -    print ("Whut?")
## diff --git a/hw.py b/hw.py
## index 07537c0..706b198 100755
## --- a/hw.py
## +++ b/hw.py
## @@ -1,4 +1,3 @@
##  #!/usr/bin/python
## 
##  print("Hello from Python")
## -print('From Davis, in the heart of California.')
```

Here `HEAD` and `HEAD^` are `git` shorthands for the current commit and the previous commit, respectively. More about this later.

### git to the rescue

One important feature of `git` that we haven't mentioned is the ability to move gracefully among different versions of stored files (or sets of stored files).

Here's a simple example. First, a brief look at our current commits:

```
cd ~/test
git log --pretty=oneline --abbrev-commit
```

```
## 561e327 Better hello-world message; new calculation module.
## 4c55e1a Initial contents of our hello-world file
```

Here are the current contents of our `hw.py` file:

```
cd ~/test
cat hw.py
```

```
## #!/usr/bin/python
##
## print("Hello from Python")
## print('From Davis, in the heart of California.')
```

Now suppose that we've managed to get confused and mess up our current `hw.py` to a more-or-less-unrecoverable state. It would be nice if we could just start over. We *can* do that, simply by checking out a previous version of `hw.py`:

```
cd ~/test
git checkout master~1 hw.py
cat hw.py
```

```
## #!/usr/bin/python
##
## print("Hello from Python")
```

The notation `master~1` is a shorthand for the first upstream commit from the current `HEAD` of the `master` branch.

Note that now-checked-out version of `hw.py` does *not* contain the line about "…the heart of California".

This would be followed by whatever edits, adds, and commits were required to rectify the problems we had created.

We can also recover from accidental deletions in our working directory. Here we (deliberately) remove `hw.py` and restore it from the repository. Here's what we start with:

```
cd ~/test
git checkout -q  master
ls
```

```
## calc.py
## hw.py
```

Now we unintentionally remove a file, `hw.py`:

```
cd ~/test
rm hw.py    #### Didn't mean to do this!!!
ls
```

```
## calc.py
```

No problem. The file is still in the repository:

```
cd ~/test

git checkout hw.py
ls
```

```
## calc.py
## hw.py
```

### Miscellaneous

A few notes:

- It's possible to use various `git` shenanigans to amend commits, checkout entire previous snapshots, etc. We won't cover these here.
- Once you have a file in your `git` repository, you should **not** use regular shell commands to, for instance, remove it. Instead you would use: `git rm unneeded.txt`
- There are now *many* GUI front ends for `git`. Have a look at: https://git-scm.com/downloads/guis

### Our current status

For our small project we now have:

- A list of every change we've made
- The time the change was made
- The identity of the person that made the change
- A *universally unique* identifier for the content of each commit
- The ability to recover lost files and/or previous versions of a given file with only a small amount of effort

All the same would be true for a project of arbitrarily-large size with an arbitrarily-large number of contributors to the project.

### Digression: Globally Unique Identifiers

```
An important characteristic of the SHA1 hash computation is that
it always computes the same ID for identical content, regardless
of where that content is. In other words, the same file content in
different directories and even on different machines yields the
exact same SHA1 hash ID. Thus, the SHA1 hash ID of a file is an
effective globally unique identifier.  A powerful corollary is
that files or blobs of arbitrary size can be compared for equality
across the Internet by merely comparing their SHA1 identifiers.
```

(The O’Reilly reference, page 33)

By the way, by my count there are more than **100 options** to the `git log` command. In this and other places we’ve just scratched the surface.

## Branching and merging – conceptual view

Once you get a project running, you probably don’t want to mess with it: break something and you may lose customers, for example.

On the other hand, no project is ever perfect, and your competitors more or less compel you to keep developing.

The traditional response to this dilemma is to copy all your files to a different directory and make modifications, additions, etc., only in that new directory. This does work, of course, but the bookkeeping can be cumbersome, particularly if you need to make changes (bug fixes) to the original at the same time.

The `git` approach makes it relatively straightforward to manage parallel lines of development.

Here is a series of images that illustrate the idea conceptually:

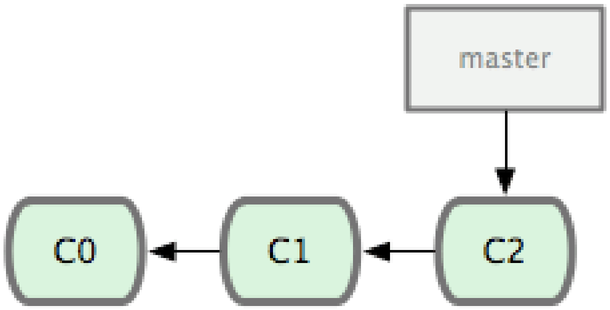


Figure 4: Original repository

Note: there is *always* a branch, and it is called `master` by default. (The name can be changed, but that is seldom done.)

Note also: each commit has a pointer to its “ancestor”.

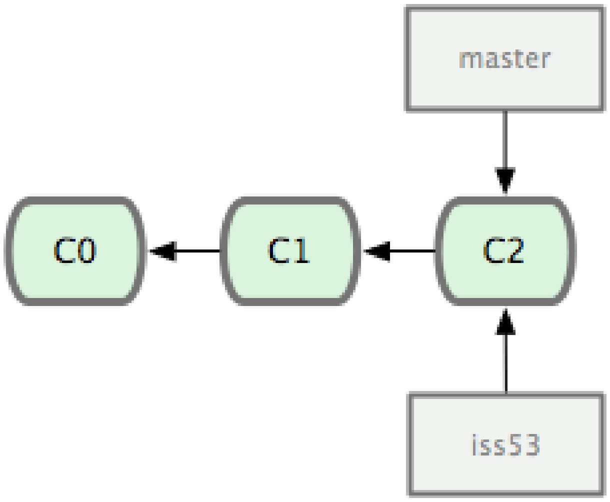


Figure 5: Create new branch to deal with “issue 53”

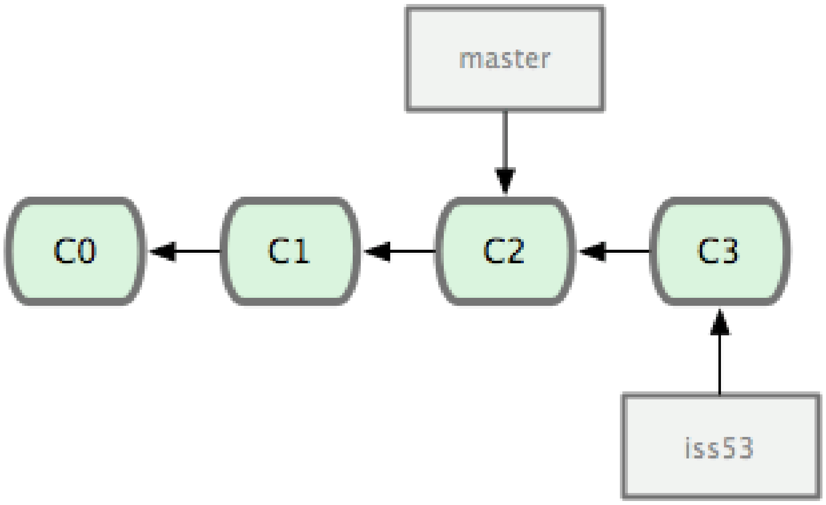


Figure 6: Make a commit to fix “issue 53”

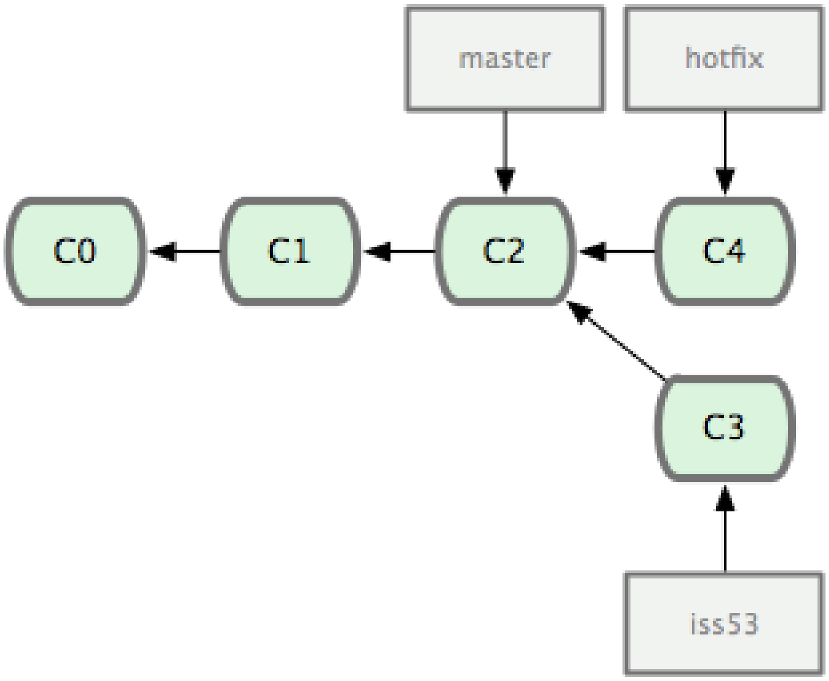


Figure 7: New branch for quick “hotfix”

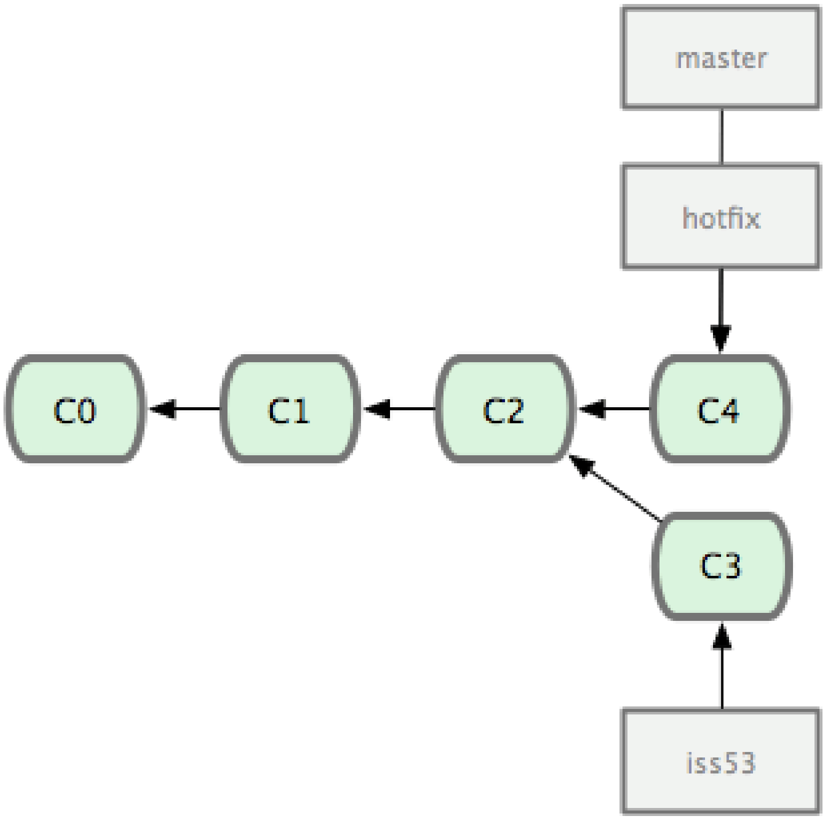


Figure 8: Merge hotfix into master

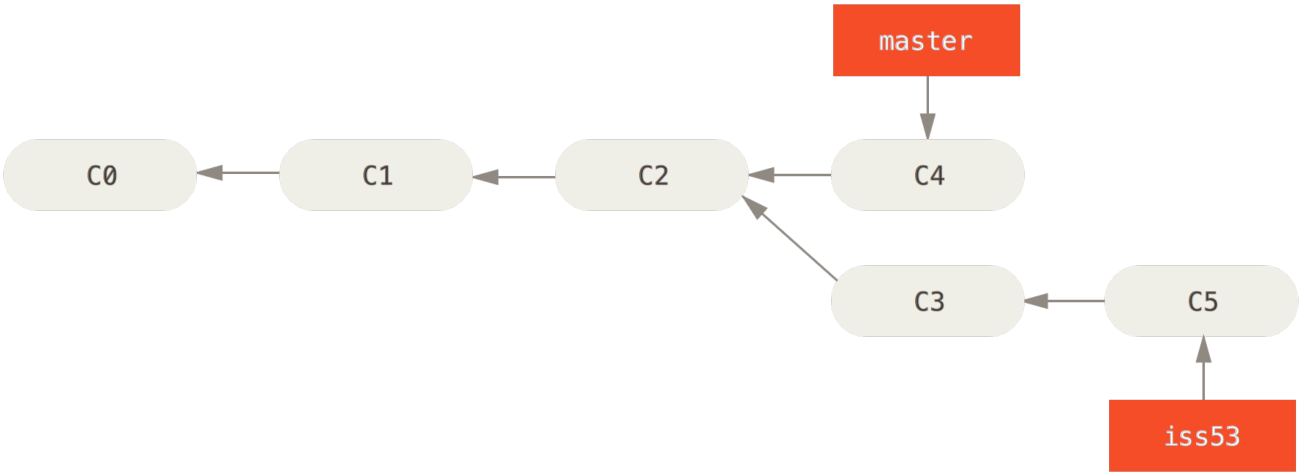


Figure 9: More work on “issue 53”

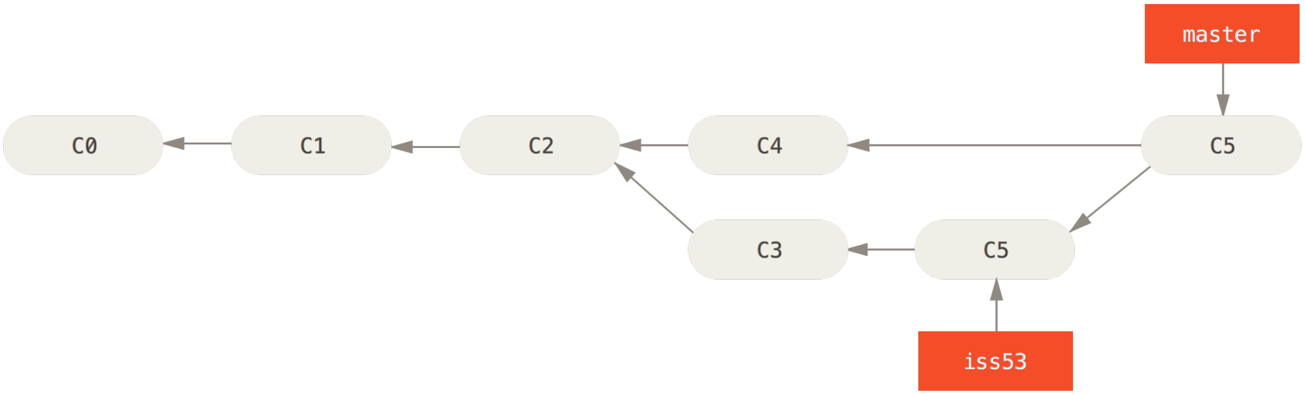


Figure 10: Merge master and iss53

## Branching and merging – a concrete example

### Setting up some repositories

We start by creating a new repository and populating it with two simple files.

First, make the directory and initialize an empty git repository:

```
if [ -d ~/test ]; then
    \rm -rf ~/test
fi

mkdir ~/test
cd ~/test

git init
```

```
## Initialized empty Git repository in /home/mike/test/.git/
```

Next, create the two files, hw.py and calc.py:

```
cd ~/test

########## hw.py ##########

cat <<EOF > hw.py
#!/usr/bin/python

print("Hello from Python")
EOF

########## calc.py ##########

cat <<EOF > calc.py
#!/usr/bin/python

import sys

prog, arg1, arg2, op = sys.argv
val1 = float(arg1)
val2 = float(arg2)

if op == "add":
    print (val1 + val2)
else:
    print ("Whut?")
EOF
```

Now we run the files, just to confirm that everything is working:

```
cd ~/test

chmod +x hw.py calc.py

./hw.py
./calc.py 42 314 add
```

```
## Hello from Python
## 356.0
```

Everthing looks good, so we `commit` the two files to our `git` repository:

```
cd ~/test

git add hw.py calc.py
git commit -m "Added the first two files, hw.py and calc.py"
```

```
## [master (root-commit) fe43925] Added the first two files, hw.py and calc.py
##  2 files changed, 15 insertions(+)
##  create mode 100755 calc.py
##  create mode 100755 hw.py
```

Next we make a new branch, `newmath`, and change to that branch:

```
cd ~/test

git branch newmath
git branch alternate    ########## for later use
git checkout newmath
```

```
## Switched to branch 'newmath'
```

Let's see what we've got on the new branch:

```
cd ~/test
ls -l
```

```
## total 8
## -rwxrwx--- 1 mike mike 168 Jan 22 19:11 calc.py
## -rwxrwx--- 1 mike mike  46 Jan 22 19:11 hw.py
```

What's *in* the files?

```
cd ~/test

cat calc.py  #########

echo -e "\n########\n"

cat hw.py    ########
```

```
## #!/usr/bin/python
##
## import sys
##
## prog, arg1, arg2, op = sys.argv
## val1 = float(arg1)
## val2 = float(arg2)
##
## if op == "add":
##     print (val1 + val2)
## else:
##     print ("Whut?")
##
## ########
##
## #!/usr/bin/python
##
## print("Hello from Python")
```

At this point the `newmath` branch is a no more than a snapshot of the `master` branch.

But we have the epiphany that having even more arithmetic operations in our `calc.py` file would be useful, so we add those, and in the process we happen to use a slightly different syntax:

```
cd ~/test

########## New calc.py ##########

cat <<EOF > calc.py
#!/usr/bin/python

import sys

prog, arg1, arg2, op = sys.argv
val1 = float(arg1)
val2 = float(arg2)

if op == "sum":
    print (val1 + val2)
elif op == "diff":
    print (val1 - val2)
elif op == "prod":
    print (val1 * val2)
elif op == "quot":
    print (val1 / val2)
else:
    print ("Whut?")
EOF
```

Let's check that our new routines are working:

```
cd ~/test
./calc.py 3 14 sum
./calc.py 5 10 diff
./calc.py 6 7 prod
./calc.py 72 9 quot
./calc.py 42 314 bessel
```

```
## 17.0
## -5.0
## 42.0
## 8.0
## Whut?
```

That all looks good, so we add `calc.py` to the staging area, then commit it:

```
cd ~/test

git add calc.py
git commit -m "Added some more functions to calc.py, changed op names"
```

```
## [newmath 7c4d502] Added some more functions to calc.py, changed op names
##  1 file changed, 7 insertions(+), 1 deletion(-)
```

## A first look at merging branches

In the scenario above the motivation for creating a branch was that we wanted to continue tinkering with and developing our software while our "production" branch stayed stable.

But at some point we'll want to fold all the great new developments from our `newmath` branch back into our production (`master`) branch, so that everybody gets the benefit of all our improvements.

The `git merge` command allows us to do that more or less painlessly. Here's an example:

```
cd ~/test
git checkout master
git merge newmath
```

```
## Switched to branch 'master'
## Updating fe43925..7c4d502
## Fast-forward
##  calc.py | 8 +++++++-
##  1 file changed, 7 insertions(+), 1 deletion(-)
```

What has this done for us? Take a look:

```
cd ~/test
git diff HEAD HEAD^
```

```
## diff --git a/calc.py b/calc.py
## index 64d1590..2374fa2 100755
## --- a/calc.py
## +++ b/calc.py
## @@ -6,13 +6,7 @@ prog, arg1, arg2, op = sys.argv
##  val1 = float(arg1)
##  val2 = float(arg2)
##
## -if op == "sum":
## +if op == "add":
##      print (val1 + val2)
## -elif op == "diff":
## -    print (val1 - val2)
## -elif op == "prod":
## -    print (val1 * val2)
## -elif op == "quot":
## -    print (val1 / val2)
##  else:
##      print ("Whut?")
```

The file `calc.py` has been modified, as indicated above. In fact, the file `calc.py` in the `master` branch is now identical to the `calc.py` in the `newmath` branch:

```
cd ~/test
git checkout -q master
echo -e "\nNew version of calc.py in master branch ##########\n"
cat calc.py
```

```
##
## New version of calc.py in master branch ##########
##
## #!/usr/bin/python
##
## import sys
##
## prog, arg1, arg2, op = sys.argv
## val1 = float(arg1)
## val2 = float(arg2)
##
## if op == "sum":
##     print (val1 + val2)
## elif op == "diff":
##     print (val1 - val2)
## elif op == "prod":
##     print (val1 * val2)
## elif op == "quot":
##     print (val1 / val2)
## else:
##     print ("Whut?")
```

Note that the line for addition was *different* in the two versions of `calc.py`:

```
if op == "add":
```

versus:

```
if op == "sum":
```

But `git` overwrote the `add` version without pausing to get our approval. Why is that?

The answer seems to be as follows. We created `newmath` as a snapshot of the original `master` branch, with the intention of improving the stuff on the `master` branch. We made those improvements on `newmath`, **without** doing any further work on `master` and then said, in effect, **give me all those improvements**. And that's what we got, including the change in syntax.

In fact, if you look at the topology of the commits, you don't see anything that looks like a branch: only a linear series of commits. It's more like we bent the oririnal branch, rather than making a new one.

## Merging branches II – conflict arises

Sometimes even `git` is not smart enough to know how to do merging. Let's look at an example of that. First, we make a new branch, `newmsg`:

```
cd ~/test
git checkout -q master
git checkout -b newmsg
```

```
## Switched to a new branch 'newmsg'
```

We now proceed to modify our `hw.py` file in the `newmsg` branch:

```
cd ~/test

cat <<EOF > hw.py
#!/usr/bin/python

print("newmsg sez: Hello from Python")
EOF

cat hw.py
```

```
## #!/usr/bin/python
##
## print("newmsg sez: Hello from Python")
```

This looks OK, so we proceed to add and commit the new file:

```
cd ~/test
git add hw.py
git commit -m "Tagged the hw.py message with 'newmsg'"
```

```
## [newmsg 27700b6] Tagged the hw.py message with 'newmsg'
##  1 file changed, 1 insertion(+), 1 deletion(-)
```

We now get an urgent request to modify our working copy of `hw.py` in the `master` branch. Perhaps against our better judgment, we proceed to make the requested change:

```
cd ~/test
git checkout -q master

cat <<EOF > hw.py
#!/usr/bin/python

print("master sez: Hello from Python")
EOF

cat hw.py
```

```
## #!/usr/bin/python
##
## print("master sez: Hello from Python")
```

This looks OK as well, so we proceed to add and commit the new file:

```
cd ~/test
git add hw.py
git commit -m "Tagged the hw.py message with 'master'"
```

```
## [master a3baa08] Tagged the hw.py message with 'master'
##  1 file changed, 1 insertion(+), 1 deletion(-)
```

We've fixed the urgent problem. Now back to business as usual. We want to merge into `master` all the fine work we did on the `newmsg` branch:

```
cd ~/test
git checkout -q master
git merge newmsg || true
```

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

Oops. What happened here? Abort the commit and check the logs:

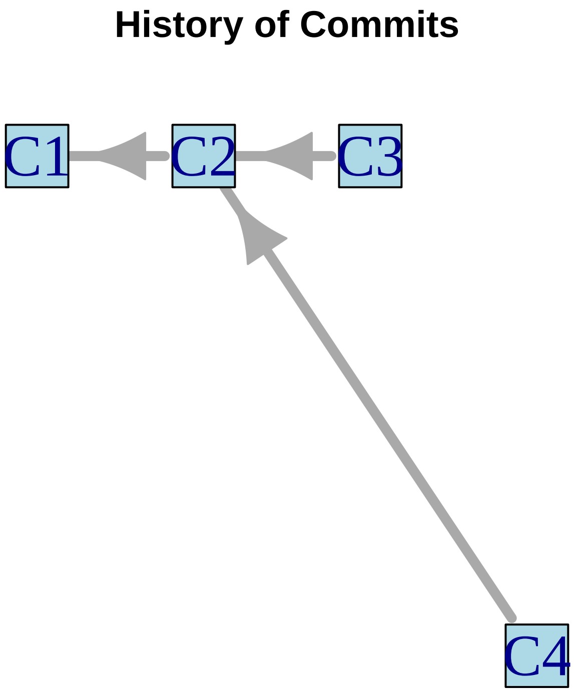
```
cd ~/test
git merge --abort
git log --pretty=oneline --abbrev-commit
```

```
## a3baa08 Tagged the hw.py message with 'master'
## 7c4d502 Added some more functions to calc.py, changed op names
## fe43925 Added the first two files, hw.py and calc.py
```

```
cd ~/test
git checkout newmsg
git log --pretty=oneline --abbrev-commit
```

```
## Switched to branch 'newmsg'
## 27700b6 Tagged the hw.py message with 'newmsg'
## 7c4d502 Added some more functions to calc.py, changed op names
## fe43925 Added the first two files, hw.py and calc.py
```

Here's a picture:



The gist of the above is that the HEAD pointer is pointing to different commits on the respective branches. Here's a digression, an under-the-hood look at that pointer.

### Digression on the mechanics of git

```
cd ~/test
ls .git
```

```
## branches
## COMMIT_EDITMSG
## config
## description
## HEAD
## hooks
## index
## info
## logs
## objects
## ORIG_HEAD
## refs
```

The .git directory is the place where all the double-secret git stuff is stored. One of the files there is HEAD, which term is a synonym for the current commit on a given branch. Here's what it looks like on the master branch:

```
cd ~/test/
git checkout -q master
```

```
cd ~/test/.git
cat HEAD
cat refs/heads/master
```

```
## ref: refs/heads/master
## a3baa08bb6b44183fba4867df5cd2707aae7c390
```

Here's the same look on the `newmsg` branch:

```
cd ~/test/
git checkout -q newmsg
```

```
cd ~/test/.git
cat HEAD
cat refs/heads/newmsg
```

```
## ref: refs/heads/newmsg
## 27700b64a1f8ec3fcf776cc9b0cbd48308b15dc0
```

## Merging branches III – conflict resolved

Returning to the mergeing conflict mentioned above. We took a snapshot of our `master` branch when we created the `newmsg` branch. In our *original* merge scenario, we created a branch (`newmath`), made some improvements on that branch (more math functions supported), and then merged `newmath` back into `master`, with no "complaints" from `git`.

The difference in *this* case, with the `newmsg` branch is that *after* taking the snapshot of our `master` branch to make improvements in the `newmsg` branch, we were forced by circumstances *also* to modify the `master` branch.

In some situations this would not be a problem, and `git` would figure out what to do without help from us.

But *this* time it so happens that we made *different* improvements in *exactly* the same piece of code on the respective branches.

Recall that one of the desirable features of `git` is:

```
Maintain Integrity and Trust
```

We certainly couldn't trust `git` if it were to fail silently and/or make random guesses as to our intentions in the case of conflict.

Hence, `git` doesn't fail silently, but neither does it leave us grasping for clues as to the source of the conflict. It *explicitly* tells us exacty the things it is unable to resolve. Here's an example. Again we attempt to merge our two branches, which is going to fail, but this time we won't abort the commit: we'll stop and poke around.

```
cd ~/test
git checkout -q master
git merge newmsg || true
```

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

As advertised, the commit failed. Let's have a look at the current state of the `hw.py` file, the one mentioned in the merge-failure message.

```
cd ~/test
cat hw.py
```

```
## #!/usr/bin/python
##
## <<<<<<< HEAD
## print("master sez: Hello from Python")
## =======
## print("newmsg sez: Hello from Python")
## >>>>>>> newmsg
```

The file has been modified by `git` to show the conflict. The section of the file starting with:

```
<<<<<<< HEAD
```

indicates the successive line (or lines) is from the version of `hw.py` on the current `HEAD`, i.e., the `master` branch.

The content from the `master` branch (only one line in this simple case) continues down to the separator line:

```
=======
```

The content *below* the separator line is the conflicting content from the other version of `hw.py`, the one on the `newmsg` branch, and this is indicated by the line at the bottom:

```
>>>>>>> newmsg
```

The next thing to do is just to pick the content we want to keep and discard the rest (including discarding the <<<<<<<, =======, and >>>>>>> lines). We could do this with a text editor (particularly easy in this simple case), or we could invoke any one of a number of GUI-based tools to sort out the conflict.

It's hard to show the interactive use of an editor in a static document, so we'll just re-create hw.py with the lines we want to keep:

```
cd ~/test
cat <<EOF > hw.py
#!/usr/bin/python

print("master sez: Hello from Python")
EOF
echo ""
cat hw.py
```

```
##
## #!/usr/bin/python
##
## print("master sez: Hello from Python")
```

Now we've resolved the conflict to our satisfaction, so we add the file and commit it:

```
cd ~/test
git add hw.py
git commit -m "Resovled conflict in hw.py"
```

```
## [master 72e595d] Resovled conflict in hw.py
```

It's also worth mentioning that the command:

```
git diff
```

is also useful in exploring merge conflicts, but it doesn't provide any new information in this simple case.

Here's a sketch of the GUI-based approach:

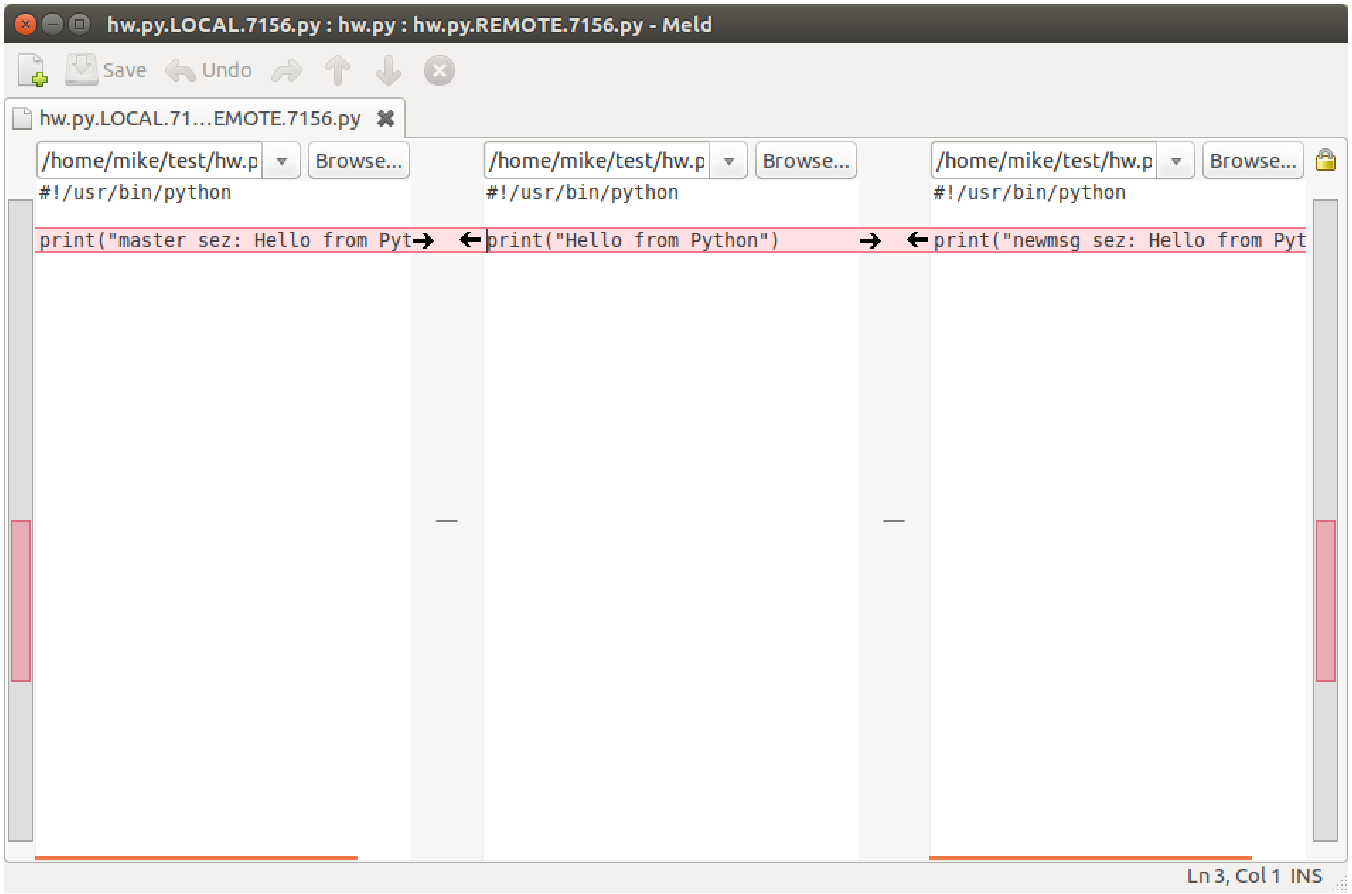
```
git mergetool -t meld
```

This brings up some additional dialog:

```
Merging:
hw.py

Normal merge conflict for 'hw.py':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```

After you hit `return` the following window pops up:



hw.py.LOCAL.7156.py : hw.py : hw.py.REMOTE.7156.py - Meld

```
#!/usr/bin/python

print("master sez: Hello from Pyt
```

```
#!/usr/bin/python

print("Hello from Python")
```

```
#!/usr/bin/python

print("newmsg sez: Hello from Pyt
```

Figure 11: The meld mergetool

The three sub-windows are, from left to right:

- The local (`master`) branch
- The common ancestor
- The remote (`newmsg`) branch

You do the editing just by clicking the black arrows to move code to the place where you want it to go.

There are many mergetools available. To see a list on your system, type:

```
git mergetool --tool-help
```

(has to be done in a directory under control of git).

## Cloning (copying) a repository

We've looked at creating and using our own repository. Another very-common use of `git` is to copy a repository from elsewhere, a process referred to as "cloning" in `git` terminology.

Here's the general idea:

```
git clone /path/to/repo  /path/to/new/repo # for a repo on the same computer

git clone username@host:/path/to/repository \ # remote repo
    [optional path name]
```

Here are some specific examples.

## Cloning a local repository

First we issue the `clone` command:

```
cd

if [ -d ~/test2 ]; then
    \rm -rf ~/test2
fi


git clone test test2
```

```
## Cloning into 'test2'...
## done.
```

Now let's go into the cloned directory and have a look around:

```
cd ~/test2

git log

git status
```

```
## commit 72e595d8d43757f4b3a48236b1d0a0c2622c8673
## Merge: a3baa08 27700b6
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:02 2016 -0800
##
##     Resovled conflict in hw.py
```

```
##
## commit a3baa08bb6b44183fba4867df5cd2707aae7c390
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:01 2016 -0800
##
##     Tagged the hw.py message with 'master'
##
## commit 27700b64a1f8ec3fcf776cc9b0cbd48308b15dc0
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:01 2016 -0800
##
##     Tagged the hw.py message with 'newmsg'
##
## commit 7c4d5023345ba54a35e5ebca605286f3804ad194
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:01 2016 -0800
##
##     Added some more functions to calc.py, changed op names
##
## commit fe439254734083e1855faf523ea69c1ccb0b9fc2
## Author: Michael Hannon <jmhannon.ucdavis@gmail.com>
## Date:   Fri Jan 22 19:11:01 2016 -0800
##
##     Added the first two files, hw.py and calc.py
## On branch master
## Your branch is up-to-date with 'origin/master'.
##
## nothing to commit, working directory clean
```

Note the comment from `git status`:

```
Your branch is up-to-date with 'origin/master'.
```

The `origin/master` bit means the branch `master` in the original repository. The identity of the original repository is located in the file:

```
.git/config
```

This is a text file that includes a bunch of information about the current repository, including, in this case, the lines:

```
[remote "origin"]
    url = /home/mike/test
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Note also that we don't seem to have the `newmath` branch in our `test2` repository. In fact, the new repository **is** aware of the `newmath` branch, but it isn't tracking it.

We can see this *and* change this (if we want to) as follows:

```
cd ~/test2

git branch -a
```

```
## * master
##   remotes/origin/HEAD -> origin/master
##   remotes/origin/alternate
##   remotes/origin/master
##   remotes/origin/newmath
##   remotes/origin/newmsg
```

All the branches are there. We instantiate a `nemath` branch in our cloned directory:

```
cd ~/test2

git checkout -b newmath origin/newmath
```

```
## Switched to a new branch 'newmath'
## Branch newmath set up to track remote branch newmath from origin.
```

And check our status:

```
cd ~/test2
git status
```

```
## On branch newmath
## Your branch is up-to-date with 'origin/newmath'.
##
## nothing to commit, working directory clean
```

The point of all this is that we can now keep things up-to-date in multiple places by using the `git` commands:

```
git pull
git push
```

(with the proviso that we have already set up the appropriate tracking information and have already set up the appropriate file permissions).

Here's an example where we make a new file in the `test2` repository and `push` it back to the `test` repository.

First we make the file:

```
cd ~/test2
git checkout master
cat <<EOF > bye.py
#!/usr/bin/python

print("Goodbye, cruel world!")
EOF
chmod +x bye.py
```

```
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Now we push the new file back to the original repository. Note that we first have to "close" the `master` branch in the original repository to avoid confusing the bookkeeping. (This would not be necessary if the original were a so-called "bare" repository, a common arrangement in "real-world" repositories.)

```
cd ~/test

git checkout newmath
```

```
## Switched to branch 'newmath'
```

Now we go back to our cloned repository:

```
cd ~/test2

git checkout master
```

```
## Already on 'master'
## Your branch is up-to-date with 'origin/master'.
```

Let's have a look at the new file and run it, just to make sure everything is OK:

```
cd ~/test2
cat bye.py

echo -e "\n########"

./bye.py
```

```
## #!/usr/bin/python
##
## print("Goodbye, cruel world!")
##
## ########
## Goodbye, cruel world!
```

Things seem OK, so we add and commit the `bye.py` file, as usual:

```
cd ~/test2

git add bye.py

git commit -m "Added a goodbye msg in bye.py"
```

```
## [master b864e0d] Added a goodbye msg in bye.py
##  1 file changed, 3 insertions(+)
##  create mode 100755 bye.py
```

Before proceeding, we set a `git` option so as to avoid an annoying warning message:

```
cd ~/test2

git config --global push.default simple
```

Now let's "push" our new file back to the `origin`:

```
cd ~/test2

git push
```

```
## To /home/mike/test
##    72e595d..b864e0d  master -> master
```

So, did this actually work? We can check. First we go back to our original respository and check out the `master` branch:

```
cd ~/test

git checkout -q master
```

What files do we have now in `origin/master`?

```
cd ~/test
ls
```

```
## bye.py
## calc.py
## hw.py
```

Aah, there's something there that wasn't there before: `bye.py`. Does it work?

```
cd ~/test

git checkout -q master

./bye.py
```

```
## Goodbye, cruel world!
```

Yep.

## Cloning a remote repository

A lot of software, documentation, etc., is distributed from web-based `git` sites (e.g., `github`) these days. But getting access to this stuff presents something of a chicken-and-egg problem. E.g., you might have to be logged in to `github` to be able to see, hence, clone, some repositories.

Here's a not-very-interesting clone command:

```
git clone git@github.com:DavisDaddy/remoteAdd.git
```

This simply clones the toy repository discussed in this document.

This will clone into a directory called `remoteAdd`. If you want it to go elsewhere, just tack that onto the end, as:

```
    git clone git@github.com:DavisDaddy/remoteAdd.git theTalk
```

There's not much point in cloning the repository we already have, but it does demonstrate how easy it is to get information this way.

There typically is a box some place on the project page where you can copy the clone URL (i.e., `git@github.com:...`).

It's also possible to `fork` a given repository (online or local) so that you can do your own independent development. For our purposes this isn't much different from cloning the repository.

## Pushing content to a remote repository

A common mode of operation is to create a `git` repository on your own computer, do some work on the corresponding project, and then put your "final" version of the project onto a web server that provides `git` services so that others can benefit from your work. The `github` site:

https://github.com/

is probably the most-famous such site.

The gist of the idea is to make a repository (aka "project") on the server with the same name as the name of the directory in which your repository resides on your own computer, then copy the files from your local repository to the remote one. Here's an example. First we make a new directory, go there, and initialize an empty `git` repository:

```
cd                 # go to login directory

if [ -d ~/remoteAdd ]; then
    \rm -rf ~/remoteAdd
fi

mkdir remoteAdd    # make your project directory

cd remoteAdd       # go there

git init           # initialize an empty repository
```

```
## Initialized empty Git repository in /home/mike/remoteAdd/.git/
```

Next, we make a new file, `README.md`, then stage and commit the file:

```
cd ~/remoteAdd

cat <<EOF > README.md       # add the first file
This repository began life on my home PC (linux box)

EOF

git add README.md   # add and commit the file

git commit -m "Initial commit w/README.md"
```

```
## [master (root-commit) 7527453] Initial commit w/README.md
##  1 file changed, 2 insertions(+)
##  create mode 100644 README.md
```

The next step (in building the infrastructure) is to login to the `github` site:

https://github.com/

and make a new repository or "project". You do this as follows:

Click the green `+ New repository` sign on the right-hand side to create, well, a new repository.

For the "Repository name" use:

```
remoteAdd
```

I.e., use the same as the name of your local directory (see the `mkdir` command above). Leave the rest of the items alone and select the green `Create repository` button.

After you make the repository, `github` gives you instructions as to how to proceed. In this case, we're adding content from our local directory to the web server. We add a remote branch and push our content to it:

```
cd ~/remoteAdd

git remote add origin git@github.com:DavisDaddy/remoteAdd.git
git push -u origin master   # push our master branch to the origin
```

```
## To git@github.com:DavisDaddy/remoteAdd.git
##  * [new branch]      master -> master
## Branch master set up to track remote branch master from origin.
```

**Important:** You need to substitute *your own* `github` login name in place of `DavisDaddy` in the above.

The `-u` option sets up "tracking" between the local and remote branches. This simplifies the subsequent use of the commands:

```
git push
git pull
```

We can ask `git` for some details of the set-up:

```
cd ~/remoteAdd

git remote -v
```

```
## origin   git@github.com:DavisDaddy/remoteAdd.git (fetch)
## origin   git@github.com:DavisDaddy/remoteAdd.git (push)
```

And some more details:

```
cd ~/remoteAdd

git remote show origin
```

```
## * remote origin
##   Fetch URL: git@github.com:DavisDaddy/remoteAdd.git
##   Push  URL: git@github.com:DavisDaddy/remoteAdd.git
##   HEAD branch: master
##   Remote branch:
##     master tracked
##   Local branch configured for 'git pull':
##     master merges with remote master
##   Local ref configured for 'git push':
##     master pushes to master (up to date)
```

And now just to round out this example, we make an additional file and push it to `gitlab`:

```
cd ~/remoteAdd

cat <<EOF > millerTale
And so it was that later
As the miller told his tale
That her face, at first just ghostly,
Turned a whiter shade of pale
EOF
```

And now we add and commit as usual:

```
cd ~/remoteAdd

git add millerTale
git commit -m "Procol Harum -- A Whiter Shade of Pale"
```

```
## [master 8bb7201] Procol Harum -- A Whiter Shade of Pale
##  1 file changed, 4 insertions(+)
##  create mode 100644 millerTale
```

Now let's copy the file to the `github` site:

```
cd ~/remoteAdd

git push
```

```
## To git@github.com:DavisDaddy/remoteAdd.git
##    7527453..8bb7201  master -> master
```

## Miscellaneous

### Ignoring files: `.gitignore`

One more thing. Over the course of time a project typically accumulates a lot of “cruft”: log files, temporary files, etc. In addition, there are often files that can easily be regenerated, such as a PDF document generated from some text-based source file. We probably don’t want to clutter our repository with those things.

`git` does not *insist* that you include such things, but it will repeatedly *remind* you that you have unstaged files if you don’t explicitly add them.

You can use a file called `.gitignore` to tell `git` to, well, ignore such files. There are many examples of such files in the wild. There’s a long discussion in the `git` help page:

```
git help gitignore
```

Here’s a simple example:

```
cd ~/test

cat <<EOF > .gitignore

# Ignore ...
#   compiled python code
*.pyc

#   editor auto-save files (ending in tilde)
*~

#   html generated from our source docs
*.html

#   except don't ignore our original html file
!orig.html

#   files resulting from compilation
*.o
*.exe

#   log files
*.log

#   everything in the "tmp" subdirectory
tmp/

EOF
```

Here's a simple test. We add an empty log file to our project directory, then check to see if `git` "complains" about it.

```
cd ~/test

echo -e "Creating an empty log file:\n"
touch junk.log

echo -e "\nThe file shows up in a listing of the directory:\n"
ls

echo -e "\nBut git does not 'see' it:\n"
git status
```

```
## Creating an empty log file:
##
##
## The file shows up in a listing of the directory:
##
## bye.py
## calc.py
## hw.py
## junk.log
##
## But git does not 'see' it:
##
## On branch master
## Untracked files:
##   (use "git add <file>..." to include in what will be committed)
##
##  .gitignore
##
## nothing added to commit but untracked files present (use "git add" to track)
```

Notice that `git` *did* complain about our newly-created `.gitignore` file, but it did *not* complain about the log file.