
# Git and Github

*MiCM Workshop 2019-06-03*

# License



# Setting Up Git

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

- our name and email address,
- what our preferred text editor is
- and that we want to use these settings globally (i.e. for every project).

On a command line, Git commands are written as `git [verb] [options]`, where `verb` is what we actually want to do and `options` is additional optional information which may be needed for the verb. So here is how Dracula sets up his new laptop:

```
$ git config --global user.name "Vlad Dracula"
$ git config --global user.email "vlad@tran.sylvan.ia"
```

This user name and email will be associated with your subsequent Git activity, which means that any changes pushed to GitHub, BitBucket, GitLab or another Git host server in a later lesson will include this information.

For these lessons, we will be interacting with GitHub and so the email address used should be the same as the one used when setting up your GitHub account. If you are concerned about privacy, please review GitHub's instructions for keeping your email address private.

If you elect to use a private email address with GitHub, then use that same email address for the `user.email` value, e.g. `username@users.noreply.github.com` replacing username with your GitHub one.

### Line Endings

As with other keys, when you hit `Return` on your keyboard, your computer encodes this input as a character. Different operating systems use different character(s) to represent the end of a line. (You may also hear these referred to as newlines or line breaks.) Because Git uses these characters to compare files, it may cause unexpected issues when editing a file on different machines. Though it is beyond the scope of this lesson, you can read more about this issue on this GitHub page.

You can change the way Git recognizes and encodes line endings using the core.autocrlf command to git config. The following settings are recommended:

On macOS and Linux:

```
$ git config --global core.autocrlf input
```

And on Windows:

```
$ git config --global core.autocrlf true
```

Dracula also has to set his favorite text editor. In the lesson, I used nano:

```
$ git config --global core.editor “nano -w”
```

The four commands we just ran above only need to be run once: the flag `--global` tells Git to use the settings for every project, in your user account, on this computer.

You can check your settings at any time:

```
$ git config --list
```

You can change your configuration as many times as you want: use the same commands to choose another editor or update your email address.

### Git Help and Manual

Always remember that if you forget a git command, you can access the list of commands by using `-h` and access the Git manual by using `--help`:

```
$ git config -h
$ git config --help
```

Then we tell Git to make `planets` a repository—a place where Git can store versions of our files:

```
$ git init
```

It is important to note that `git init` will create a repository that includes subdirectories and their files—there is no need to create separate repositories nested within the `planets` repository, whether subdirectories are present from the beginning or added later. Also, note that the creation of the planets directory and its initialization as a repository are completely separate processes.

If we use `ls` to show the directory’s contents, it appears that nothing has changed.

But if we add the `-a` flag to show everything, we can see that Git has created a hidden directory within planets called `.git`:

```
$ ls -a
.    ..   .git
```

Git uses this special subdirectory to store all the information about the project, including all files and sub-directories located within the project’s directory. If we ever delete the `.git` subdirectory, we will lose the project’s history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status
On branch master
Initial commit
```

```
nothing to commit (create/copy files and use "git add" to track)
```

If you are using a different version of git, the exact wording of the output might be slightly different.

# Creating a Repository

First, let's create a directory in `Desktop` folder for our work and then move into that directory:

```
$ cd ~/Desktop
$ mkdir planets
$ cd planets
```

# Tracking Changes

First let's make sure we're still in the right directory. You should be in the `planets` directory.

```
$ cd ~/Desktop/planets
```

Let's create a file called `mars.txt` that contains some notes about the Red Planet's suitability as a base. We'll use nano to edit the file; you can use whatever editor you like. In particular, this does not have to be the core.editor you set globally earlier. But remember, the bash command to create or edit a new file will depend on the editor you choose (it might not be nano). For a refresher on text editors, check out "Which Editor?" in The Unix Shell lesson.

```
$ nano mars.txt
```

Type the text below into the `mars.txt` file:

```
Cold and dry, but everything is my favorite color
```

Let's first verify that the file was properly created by running the list command (`ls`):

```
$ ls
mars.txt
```

`mars.txt` contains a single line, which we can see by running:

```
$ cat mars.txt
Cold and dry, but everything is my favorite color
```

If we check the status of our project again, Git tells us that it's noticed the new file:

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

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

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of. We can tell Git to track a file using `git add`:

```
$ git add mars.txt
```

and then check that the right thing happened:

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
 new file:   mars.txt
```

Git now knows that it's supposed to keep track of `mars.txt`, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
$ git commit -m "Start notes on Mars as a base"
[master (root-commit) 5dd3e94] Start notes on Mars as a base
 1 file changed, 2 insertions(+)
 create mode 100644 mars.txt
```

When we run `git commit`, Git takes everything we have told it to save by using git add and stores a copy permanently inside the special `.git` directory. This permanent copy is called a commit (or revision) and its short identifier is `5dd3e94`. Your commit may have another identifier.

We use the `-m` flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run `git commit` without the `-m` option, Git will launch nano (or whatever other editor we configured as core.editor) so that we can write a longer message.

Good commit messages start with a brief (<50 characters) statement about the changes made in the commit. Generally, the message should complete the sentence "If applied, this commit will" . If you want to go into more detail, add a blank line between the summary line and your additional notes. Use this additional space to explain why you made changes and/or what their impact will be.

If we run git status now:

```
$ git status
On branch master
nothing to commit, working directory clean
```

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
$ git log
commit 5dd3e94dac1536bbf1ba260b66e006f4e2fecd09 (HEAD -> master)
```

```
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Fri Jun 2 13:43:54 2019 -0400

    Start notes on Mars as a base
```

`git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier), the commit's author, when it was created, and the log message Git was given when the commit was created.

> **Where Are My Changes?**
>
> If we run `ls` at this point, we will still see just one file called mars.txt. That's because Git saves information about files' history in the special `.git` directory mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version).

Now suppose Dracula adds more information to the file. (Again, we'll edit with `nano` and then `cat` the file to show its contents; you may use a different editor, and don't need to `cat`.)

```
$ nano mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   mars.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let's do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff
diff --git a/mars.txt b/mars.txt
index 9b3a2ee..5178740 100644
--- a/mars.txt
```

```
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

The output is cryptic because it is actually a series of commands for tools like editors and `patch`, telling them how to reconstruct one file given the other. If we break it down into pieces:

- The first line tells us that Git is producing output similar to the Unix diff command comparing the old and new versions of the file.
- The second line tells exactly which versions of the file Git is comparing; `9b3a2ee` and `5178740` are unique computer-generated labels for those versions.
- The third and fourth lines once again show the name of the file being changed.
- The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the + marker in the first column shows where we added a line.

After reviewing our change, it's time to commit it:

```
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
 modified:   mars.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Whoops: Git won't commit because we didn't use `git add` first. Let's fix that:

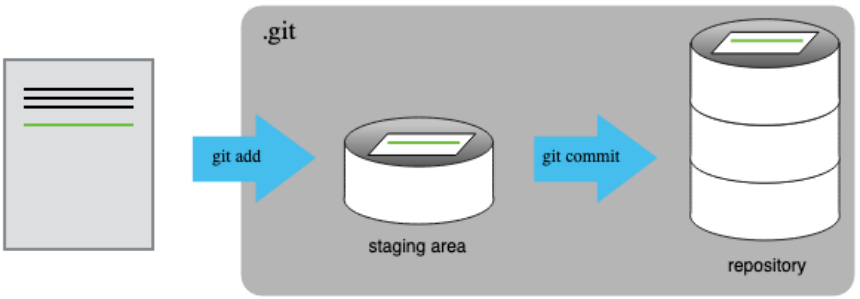
```
$ git add mars.txt
$ git commit -m "Add concerns about effects of Mars' moons on Wolfman"
[master 34961b1] Add concerns about effects of Mars' moons on Wolfman
 1 file changed, 1 insertion(+)
```

Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to relevant research to our thesis. We might want to commit those additions, and the corresponding bibliography entries, but not commit some of our work drafting the conclusion (which we haven't finished yet).

To allow for this, Git has a special staging area where it keeps track of things that have been added to the current changeset[1] but not yet committed.

### Staging Area

If you think of Git as taking snapshots of changes over the life of a project, `git add` specifies what will go in a snapshot (putting things in the staging area), and `git commit` then actually takes the snapshot, and makes a permanent record of it (as a commit). If you don't have anything staged when you type git commit, Git will prompt you to use `git commit -a` or `git commit --all`, which is kind of like gathering everyone to take a group photo! However, it's almost always better to explicitly add things to the staging area, because you might commit changes you forgot you made. (Going back to the group photo simile, you might get an extra with incomplete makeup walking on the stage for the picture because you used `-a`!) Try to stage things manually, or you might find yourself searching for "git undo commit" more than you would like!




Let's watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we'll add another line to the file:

```
$ nano mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity

$ git diff
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

So far, so good: we've added one line to the end of the file (shown with a + in the first column). Now let's put that change in the staging area and see what `git diff` reports:

[1] A group of changes to one or more files that are or will be added to a single commit in a version control repository.

```
$ git add mars.txt
$ git diff
```

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

```
$ git diff --staged
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

```
$ git commit -m "Discuss concerns about Mars' climate for Mummy"
[master 005937f] Discuss concerns about Mars' climate for Mummy
 1 file changed, 1 insertion(+)
```

check our status:

```
$ git status
On branch master
nothing to commit, working directory clean
```

and look at the history of what we've done so far:

```
$ git log
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5 (HEAD -> master)
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2019 -0400

    Discuss concerns about Mars' climate for Mummy

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2019 -0400
```

```
    Add concerns about effects of Mars' moons on Wolfman

commit 5dd3e94dac1536bbf1ba260b66e006f4e2fecd09
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2019 -0400

    Start notes on Mars as a base
```

### A More Helpful Log

To avoid having `git log` cover your entire terminal screen, you can limit the number of commits that Git lists by using `-N`, where N is the number of commits that you want to view. For example, if you only want information from the last commit you can use:

```
$ git log -1
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2019 -0400

   Discuss concerns about Mars' climate for Mummy
```

You can also reduce the quantity of information using the `--oneline` option:

```
$ git log --oneline
005937f Discuss concerns about Mars' climate for Mummy
34961b1 Add concerns about effects of Mars' moons on Wolfman
f22b25e Start notes on Mars as a base
```

You can also combine the `--oneline` option with others. One useful combination adds `--graph` to display the commit history as a text-based graph and to indicate which commits are associated with the current `HEAD`, the current branch master, or other Git references:

```
$ git log --oneline --graph
* 005937f (HEAD -> master) Discuss concerns about Mars' climate for Mummy
* 34961b1 Add concerns about effects of Mars' moons on Wolfman
* f22b25e Start notes on Mars as a base
```

# Exploring History

As we saw in the previous section, we can refer to commits by their identifiers. You can refer to the most recent commit of the working directory by using the identifier `HEAD`.

We've been adding one line at a time to mars.txt, so it's easy to track our progress by looking, so let's do that using our HEADs. Before we start, let's make a change to `mars.txt`, adding yet another line.

```
$ nano mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
An ill-considered change
```

Now, let's see what we get.

```
$ git diff HEAD mars.txt
diff --git a/mars.txt b/mars.txt
index b36abfd..0848c8d 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,3 +1,4 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
 But the Mummy will appreciate the lack of humidity
+An ill-considered change.
```

which is the same as what you would get if you leave out `HEAD` (try it). The real goodness in all this is when you can refer to previous commits. We do that by adding `~1` (where "`~`" is "tilde", pronounced [til-duh]) to refer to the commit one before `HEAD`.

```
$ git diff HEAD~1 mars.txt
```

If we want to see the differences between older commits we can use git diff again, but with the notation `HEAD~1`, `HEAD~2`, and so on, to refer to them:

```
$ git diff HEAD~3 mars.txt
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,4 @@
```

```
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
+An ill-considered change
```

We could also use `git show` which shows us what changes we made at an older commit as well as the commit message, rather than the differences between a commit and our working directory that we see by using `git diff`.

```
$ git show HEAD~3 mars.txt
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2019 -0400

    Start notes on Mars as a base

diff --git a/mars.txt b/mars.txt
new file mode 100644
index 0000000..df0654a
--- /dev/null
+++ b/mars.txt
@@ -0,0 +1 @@
+Cold and dry, but everything is my favorite color
```

In this way, we can build up a chain of commits. The most recent end of the chain is referred to as `HEAD`; we can refer to previous commits using the `~` notation, so `HEAD~1` means "the previous commit", while `HEAD~123` goes back 123 commits from where we are now.

We can also refer to commits using those long strings of digits and letters that git log displays. These are unique IDs for the changes, and "unique" really does mean unique: every change to any set of files on any computer has a unique 40-character identifier. Our first commit was given the ID `f22b25e3233b4645dabd0d81e651fe074bd8e73b`, so let's try this:

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
diff --git a/mars.txt b/mars.txt
index df0654a..93a3e13 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,4 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

```
+But the Mummy will appreciate the lack of humidity
+An ill-considered change
```

That's the right answer, but typing out random 40-character strings is annoying, so Git lets us use just the first few characters (typically four or five for normal size projects):

```
$ git diff f22b25e mars.txt
```

All right! So we can save changes to files and see what we've changed—now how can we restore older versions of things? Let's suppose we change our mind about the last update to mars.txt (the "ill-considered change").

`git status` now tells us that the file has been changed, but those changes haven't been staged:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   mars.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

We can put things back the way they were by using `git checkout`:

```
$ git checkout HEAD mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
$ git checkout f22b25e mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   mars.txt
```

Notice that the changes are currently in the staging area. Again, we can put things back the way they were by using `git checkout`:

```
$ git checkout HEAD mars.txt
```

### Don't Lose Your HEAD

Above we used

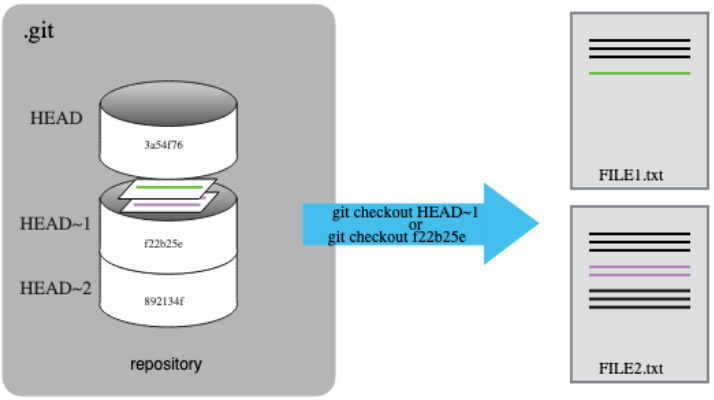
```
$ git checkout f22b25e mars.txt
```

to revert `mars.txt` to its state after the commit `f22b25e`. But be careful! The command checkout has other important functionalities and Git will misunderstand your intentions if you are not accurate with the typing. For example, if you forget `mars.txt` in the previous command.

```
$ git checkout f22b25e
Note: checking out 'f22b25e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

 git checkout -b <new-branch-name>

HEAD is now at f22b25e Start notes on Mars as a base
```

The "detached HEAD" is like "look, but don't touch" here, so you shouldn't make any changes in this state. After investigating your repo's past state, reattach your HEAD with `git checkout master`.

It's important to remember that we must use the commit number that identifies the state of the repository before the change we're trying to undo. A common mistake is to use the number of the commit in which we made the change we're trying to discard. In the example below, we want to retrieve the state from before the most recent commit (`HEAD~1`), which is commit `f22b25e`:




### Simplifying the Common Case

If you read the output of `git status` carefully, you'll see that it includes this hint:

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

As it says, `git checkout` without a version identifier restores files to the state saved in `HEAD`. The double dash `--` is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the commit identifier.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it's hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

# Ignoring Things

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis? Let's create a few dummy files:

```
$ mkdir results
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

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

 a.dat
 b.dat
 c.dat
 results/

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

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`:

```
$ nano .gitignore
$ cat .gitignore
*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the results directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of `git status` is much cleaner:

```
$ git status
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)
```

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

```
$ git add .gitignore
$ git commit -m "Ignore data files and the results folder."
$ git status
On branch master
nothing to commit, working directory clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding to the repository files that we don't want to track:

```
$ git add a.dat
The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. For example, `git add -f a.dat`. We can also always see the status of ignored files if we want:

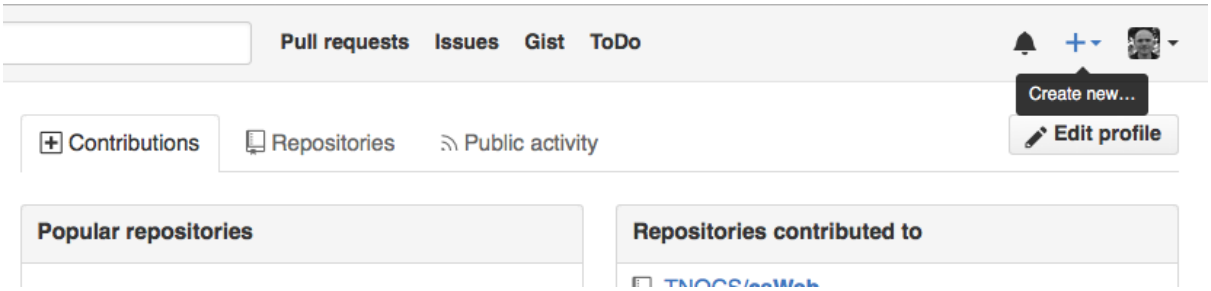
```
$ git status --ignored
On branch master
Ignored files:
 (use "git add -f <file>..." to include in what will be committed)

        a.dat
        b.dat
        c.dat
        results/

nothing to commit, working directory clean
```

# Remotes and GitHub

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

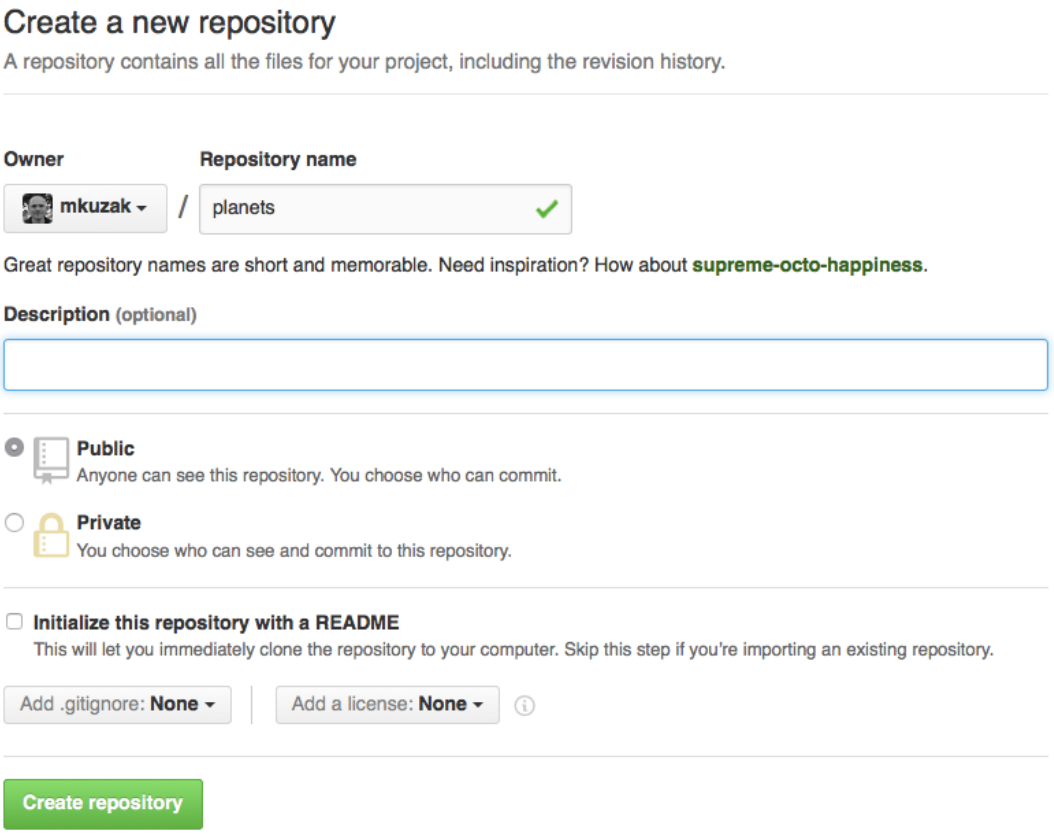
Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub, Bitbucket or GitLab to hold those master copies; we'll explore the pros and cons of this in the final section of this lesson.

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository called `planets`:

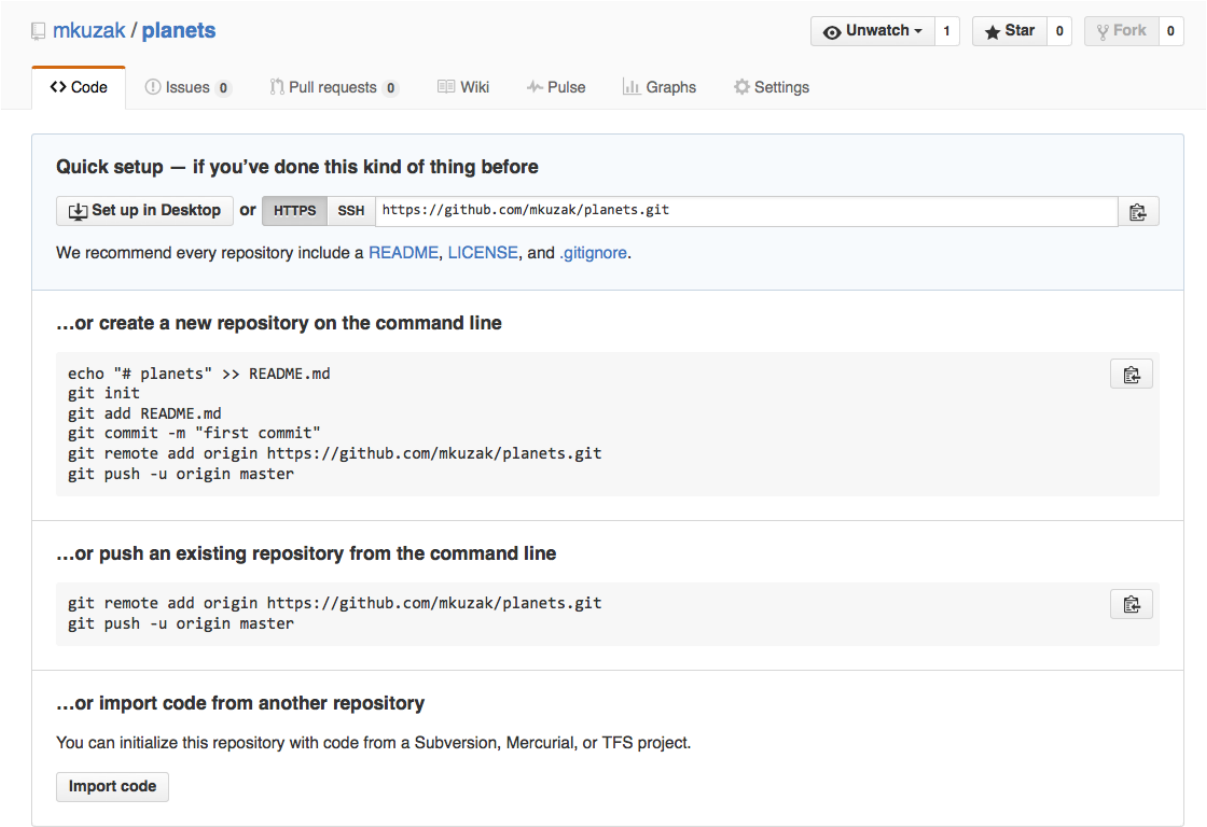



Name your repository "planets" and then click "Create Repository".

Note: Since this repository will be connected to a local repository, it needs to be empty. Leave "Initialize this repository with a README" unchecked, and keep "None" as options for both "Add .gitignore" and "Add a license." See the "GitHub License and README files" exercise below for a full explanation of why the repository needs to be empty.



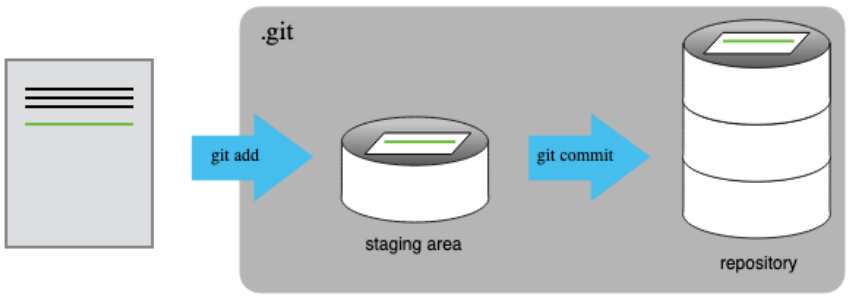

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository:



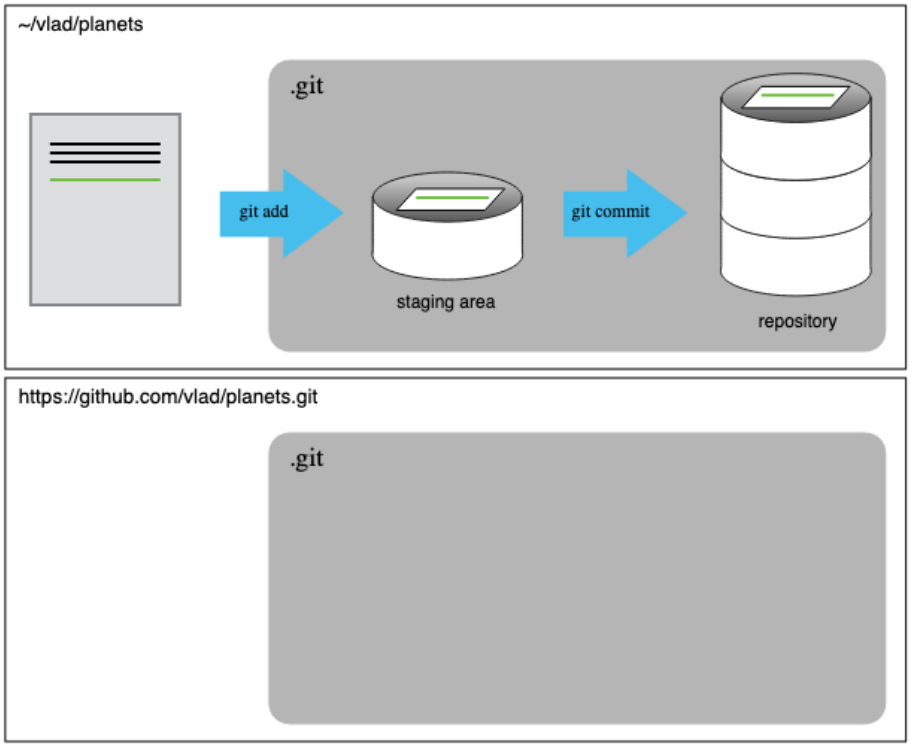

This effectively does the following on GitHub's servers:

```
$ mkdir planets
$ cd planets
$ git init
```

If you remember back to the earlier section where we added and committed our earlier work on `mars.txt`, we had a diagram of the local repository which looked like this:




Now that we have two repositories, we need a diagram like this:




Note that our local repository still contains our earlier work on `mars.txt`, but the remote repository on GitHub appears empty as it doesn't contain any files yet.

### Adding the Remote

The next step is to connect the two repositories. We do this by making the GitHub repository a remote for the local repository. The home page of the repository on GitHub includes the string we need to identify it:


Click on the 'HTTPS' link to change the protocol from SSH to HTTPS.

Copy that URL from the browser, go into the local `planets` repository, and run this command:

**HTTPS vs. SSH**

We use HTTPS here because it does not require additional configuration. After the workshop you may want to set up SSH access, which is a bit more secure, by following one of the great tutorials from GitHub, Atlassian/Bitbucket and GitLab (this one has a screencast).

```
$ git remote add origin https://github.com/vlad/planets.git
```

Make sure to use the URL for your repository rather than Vlad's: the only difference should be your username instead of `vlad`.

`origin` is a local name used to refer to the remote repository. It could be called anything, but `origin` is a convention that is often used by default in git and GitHub, so it's helpful to stick with this unless there's a reason not to.
We can check that the command has worked:

```
$ git remote -v
origin   https://github.com/vlad/planets.git (push)
origin   https://github.com/vlad/planets.git (fetch)
```

### Pushing to Remote

Once the remote is set up, this command will push the changes from our local repository to the repository on GitHub:

```
$ git push origin master
Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (16/16), 1.45 KiB | 372.00 KiB/s, done.
Total 16 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/vlad/planets.git
 * [new branch]      master -> master
```

### Pulling from Remote

We can pull changes from the remote repository to the local one as well:

```
$ git pull origin master
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

# Conflicts

As soon as people can work in parallel, they'll likely step on each other's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy. Version control helps us manage these conflicts[2] by giving us tools to resolve[3] overlapping changes.

To see how we can resolve conflicts, we must first create one. The file `mars.txt` currently looks like this in both partners' copies of our `planets` repository:

```
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

Let's add a line to one partner's copy only:

```
$ nano mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
This line added to Wolfman's copy
```

and then push the change to GitHub:

```
$ git add mars.txt
$ git commit -m "Add a line in our home copy"
[master 5ae9631] Add a line in our home copy
 1 file changed, 1 insertion(+)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 331 bytes | 331.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/vlad/planets.git
   29aba7c..dabb4c8  master -> master
```

[2] A change made by one user of a version control system that is incompatible with changes made by other users. Helping users resolve conflicts is one of version control's major tasks.
[3] To eliminate the conflicts between two or more incompatible changes to a file or set of files being managed by a version control system.

Now let's have the other partner make a different change to their copy without updating from GitHub:

```
$ nano mars.txt
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We added a different line in the other copy
```

We can commit the change locally:

```
$ git add mars.txt
$ git commit -m "Add a line in my copy"
[master 07ebc69] Add a line in my copy
 1 file changed, 1 insertion(+)
```

but Git won't let us push it to GitHub:

```
$ git push origin master
To https://github.com/vlad/planets.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/vlad/planets.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Git rejects the push because it detects that the remote repository has new updates that have not been incorporated into the local branch. What we have to do is pull the changes from GitHub, merge them into the copy we're currently working in, and then push that. Let's start by pulling:

```
$ git pull origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
    29aba7c..dabb4c8  master     -> origin/master
Auto-merging mars.txt
CONFLICT (content): Merge conflict in mars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

The `git pull` command updates the local repository to include those changes already included in the remote repository. After the changes from remote branch have been fetched, Git detects that changes made to the local copy overlap with those made to the remote repository, and therefore refuses to merge the two versions to stop us from trampling on our previous work. The conflict is marked in in the affected file:

```
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
<<<<<<< HEAD
We added a different line in the other copy
=======
This line added to Wolfman's copy
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Our change is preceded by `<<<<<<< HEAD`. Git has then inserted `=======` as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with `>>>>>>>`. (The string of letters and digits "dabb4c…" after that marker identifies the commit we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change made in the local repository, keep the change made in the remote repository, write something new to replace both, or get rid of the change entirely. Let's replace both so that the file looks like this:

```
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

To finish merging, we add mars.txt to the changes being made by the merge and then commit:

```
$ git add mars.txt
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

       modified:   mars.txt
$ git commit -m "Merge changes from GitHub"
[master 2abf2b1] Merge changes from GitHub
$ git push origin master
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 645 bytes | 645.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/vlad/planets.git
   dabb4c8..2abf2b1  master -> master
```

Git keeps track of what we've merged with what, so we don't have to fix things by hand again when the collaborator who made the first change pulls again:

```
$ git pull origin master
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 4), reused 6 (delta 4), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
    dabb4c8..2abf2b1  master     -> origin/master
Updating dabb4c8..2abf2b1
Fast-forward
 mars.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

We get the merged file:

```
$ cat mars.txt
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

```
We removed the conflict on this line
```

We don’t need to merge again because Git knows someone has already done that.

Git’s ability to resolve conflicts is very useful, but conflict resolution costs time and effort, and can introduce errors if conflicts are not resolved correctly. If you find yourself resolving a lot of conflicts in a project, consider these technical approaches to reducing them:

- Pull from upstream more frequently, especially before starting new work
- Use topic branches to segregate work, merging to master when complete
- Make smaller more atomic commits
- Where logically appropriate, break large files into smaller ones so that it is less likely that two authors will alter the same file simultaneously

Conflicts can also be minimized with project management strategies:

- Clarify who is responsible for what areas with your collaborators
- Discuss what order tasks should be carried out in with your collaborators so that tasks expected to change the same lines won’t be worked on simultaneously
- If the conflicts are stylistic churn (e.g. tabs vs. spaces), establish a project convention that is governing and use code style tools (e.g. htmltidy, perltidy, rubocop, etc.) to enforce, if necessary.

# Additional Help

There are lots of great resources to help clarify git concepts and deepen your understanding of version control.

As mentioned earlier, this workshop was based on the Software Carpentry workshop "Version Control with Git", which has some extra material not covered here.

If you want a more visual representation of how your commands affect the git repository (similar to the physical model we built in the workshop), have a look at Learn Git Branching.

If you have any questions, feel free to reach out to me, Rob Syme. I'm "robsyme" most places on the web, but our team at C3G has weekly "Open Door" sessions where anybody is free to drop by and ask for help, particularly questions about data analysis, HPC, genomics, bioinformatics, sequencing, etc. Simply drop us a line here: https://www.computationalgenomics.ca/open-door/

Very best of luck, and I hope that the workshop proves helpful in your research.