Exploring History

As we saw in the previous episode, 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 `HEAD`s. 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**-d*uh*]) 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 <fenago@example.com>
Date:   Thu Aug 22 09:51:46 2013 -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, so let's try this:

```
$ git diff UPDATE_ID_HERE 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 seven for normal size projects):

```
$ git diff UPDATE_ID_HERE 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
```

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 main
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 UPDATE_ID_HERE mars.txt
```

```
$ cat mars.txt
```

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

```
$ git status
```

```
On branch main
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
```

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 main
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 main
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 main
nothing to commit, working tree clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding files to the repository 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 main
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 tree clean
```

> ### *Ignoring Nested Files*
>
> *Given a directory structure that looks like:*
>
> ```
> results/data
> results/plots
> ```
>
> *How would you ignore only `results/plots` and not `results/data` ?*
>
> > ### *Solution*
> >
> > *If you only want to ignore the contents of `results/plots`, you can change your `.gitignore` to ignore only the `/plots/` subfolder by adding the following line to your .gitignore:*
> >
> > ```
> > results/plots/
> > ```
> >
> > *This line will ensure only the contents of `results/plots` is ignored, and not the contents of `results/data`.*

> *As with most programming issues, there are a few alternative ways that one may ensure this ignore rule is followed. The "Ignoring Nested Files: Variation" exercise has a slightly different directory structure that presents an alternative solution. Further, the discussion page has more detail on ignore rules.*

Including Specific Files

How would you ignore all `.dat` files in your root directory except for `final.dat`? Hint: Find out what `!` (the exclamation point operator) does

Solution

You would add the following two lines to your .gitignore:

```
*.dat           # ignore all data files
!final.dat      # except final.data
```

> *The exclamation point operator will include a previously excluded entry.*
>
> *Note also that because you've previously committed `.dat` files in this lesson they will not be ignored with this new rule. Only future additions of `.dat` files added to the root directory will be ignored.*

Ignoring Nested Files: Variation

Given a directory structure that looks similar to the earlier Nested Files exercise, but with a slightly different directory structure:

```
results/data
results/images
results/plots
results/analysis
```

How would you ignore all of the contents in the results folder, but not `results/data`?

Hint: think a bit about how you created an exception with the `!` operator before.

Solution

If you want to ignore the contents of `results/` but not those of `results/data/`, you can change your `.gitignore` to ignore the contents of results folder, but create an exception for the contents of the `results/data` subfolder. Your .gitignore would look like this:

```
results/*               # ignore everything in results folder
!results/data/          # do not ignore results/data/ contents
```

Ignoring all data Files in a Directory

Assuming you have an empty .gitignore file, and given a directory structure that looks like:

```
results/data/position/gps/a.dat
results/data/position/gps/b.dat
results/data/position/gps/c.dat
```

```
results/data/position/gps/info.txt
results/plots
```

> *What's the shortest* `.gitignore` *rule you could write to ignore all* `.dat` *files in* `result/data/position/gps` *? Do not ignore the* `info.txt` *.*
>
> ### *Solution*
>
> *Appending* `results/data/position/gps/*.dat` *will match every file in* `results/data/position/gps` *that ends with* `.dat` *. The file* `results/data/position/gps/info.txt` *will not be ignored.*

Ignoring all data Files in the repository

Let us assume you have many `.dat` *files in different subdirectories of your repository. For example, you might have:*

```
results/a.dat
data/experiment_1/b.dat
data/experiment_2/c.dat
data/experiment_2/variation_1/d.dat
```

> *How do you ignore all the* `.dat` *files, without explicitly listing the names of the corresponding folders?*
>
> ### *Solution*
>
> *In the* `.gitignore` *file, write:*
>
> ```
> **/*.dat
> ```
>
> *This will ignore all the* `.dat` *files, regardless of their position in the directory tree. You can still include some specific exception with the exclamation point operator.*

The Order of Rules

Given a `.gitignore` *file with the following contents:*

```
*.dat
!*.dat
```

> *What will be the result?*
>
> ### *Solution*
>
> *The* `!` *modifier will negate an entry from a previously defined ignore pattern. Because the* `!*.dat` *entry negates all of the previous* `.dat` *files in the* `.gitignore` *, none of them will be ignored, and all* `.dat` *files will be tracked.*

Log Files

You wrote a script that creates many intermediate log-files of the form `log_01`, `log_02`, `log_03`, etc. You want to keep them but you do not want to track them through `git`.

1. *Write **one** `.gitignore` entry that excludes files of the form `log_01`, `log_02`, etc.*
2. *Test your "ignore pattern" by creating some dummy files of the form `log_01`, etc.*
3. *You find that the file `log_01` is very important after all, add it to the tracked files without changing the `.gitignore` again.*
4. *Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and thus would exclude via `.gitignore`.*

Solution

1. *append either `log_*` or `log*` as a new entry in your .gitignore*
2. *track `log_01` using `git add -f log_01`*