**Reflect: Using diff to Find Bugs**

**Make sure you can access the command line**

If you are on Windows, you’ll need to install some software in order to use a unix-like command line (different from the Windows built-in Command Prompt), as we mentioned previously. For more info, [check out this page](https://www.udacity.com/wiki/ud775/command-line-instructions#windows-users).

**Choose a text editor**

As noted in the previous video, you should make sure to use a simple text editor like [Notepad++](http://notepad-plus-plus.org/), [Sublime](http://www.sublimetext.com/2), [Atom](https://atom.io/), emacs, vim, etc. and **not** a rich-text editor like Microsoft Word or OpenOffice, so that you can easily look at your files' content on the command line.

If you don’t have one that you like yet, Sublime is a good option that will work on Windows, Mac, and Linux. We have provided more detailed instructions for setting up Sublime than for other editors, and we use Sublime for all the examples in the course. You can download Sublime [here](http://www.sublimetext.com/2).

**Make sure you can launch your editor from the command line**

It will be helpful to be able to launch your text editor from the command line. See [here](https://www.udacity.com/wiki/ud775/sublime) for instructions on how to do this for Sublime. If you have trouble getting this working, videos at the end of the lesson called "Setting up Your Workspace on Windows" and "Setting Up Your Workspace on Mac" will demonstrate this process, so you can wait until then.

**Set up your course workspace**

Right now, you should create a version-control directory (a more computer-science-y term for “folder”) to hold all your files for this course in an easy to remember location, then create a reflections sub-directory, and within that, create a plain text file called lesson\_1\_reflections.txt for the questions from this lesson.

You can do this by running the following commands in either Git Bash or the terminal (the bits after the # signs are comments, anything after those are not interpreted):

cd ~ # change directories to your home directory

mkdir version-control # make version-control directory

cd version-control # go to version-control directory

mkdir reflections # create reflections directory

cd reflections # go to reflections directory

subl lesson\_1\_reflections.txt # launch sublime with file called lesson\_1\_reflections.txt (you can replace subl with another editor here if you prefer a different one)

If you prefer, rather than creating the file from scratch, you could download the lesson\_1\_reflections\_prompts.txt file from the Downloadables section, place it in the reflections directory and rename it, then add your response after the first prompt.

Once you’ve saved your file, if you want to double-check that everything has gone as planned, try these commands:

pwd # print working directory - shows what directory you are in

ls # list the files in this directory

If you're having trouble getting this working, videos at the end of the lesson called "Setting up Your Workspace on Windows" and "Setting Up Your Workspace on Mac" will demonstrate parts of this process, so you might want to wait until then. In the mean time, you can create the directories and file using your OS’s Graphical Interface for working with files (Finder, Windows Explorer, etc). It's good to get some practice using the command line, though, since we'll be using it a lot in this course, so once you've watched the workspace instructions, make sure you can complete actions like this on the command line.

You are also welcome to use a different naming scheme, but later in the course we will refer to this file structure and it will be up to you to translate to whatever naming scheme you chose instead!

**Use short lines**

Many command line tools, including Git, are less useful if your files contain very long lines. For example, if you use diff to compare two files that have all their content on the same line, diff will only show you that the two files are different. It will not be able to pinpoint the location of the difference for you.

For this reason, it is a good idea to make sure you keep your lines reasonably short when writing your reflections (or other plain-text files). The exact limit is a matter of personal preference. Many developers use a max line length of 80 to 120 characters. Some editors can automatically insert line breaks for you, but for others, like Sublime, you will need to remember to press enter when you want to create a new line.

**Do the first reflection exercise**

Populate lesson\_1\_reflections.txt with the following question and your thoughts on it:

**How did viewing a diff between two versions of a file help you see the bug that was introduced?**

When you've created your document, written down your thoughts, and saved the file, click "Next" to learn about some systems that can help you create these versions of your files.

Like Sarah said, there is a lot of research out there on reflection. If you want a quick intro with some neat graphics, check out [this site](https://sites.google.com/site/reflection4learning/why-reflect).

For a more in-depth look, check out [this seminal 20-page paper](http://184.182.233.150/rid=1LW06D9V6-26428MK-1Z64/Mezirow%27s%20chapter,%20How%20Critical%20Refletion%20Triggers%20TL.pdf) on the topic.

## Reflect: Using Git to View History

Now that you've had some experience using Git yourself, go add the following question and your thoughts on it to your reflections file:

**How can you use the commands git log and git diff to view the history of files?**

When you've updated your document, click "Next" to add more concepts to your map with Sarah. Then you'll learn how to use Git to return to previous versions of your files.

**Identify changes –** git log **Identify changes status –** git log –stat

**Identify differences –** git diff [arq1] [arq2] **Identify changes lines –** git log -stat [arq]

**Commit**

Commits make up the log output, so this is definitely a valid interpretation, as long as you point the arrow from commit to log.

## Git Errors and Warnings Solution

**Should not be doing an octopus**  
Octopus is a strategy Git uses to combine many different versions of code together. This message can appear if you try to use this strategy in an inappropriate situation.

**You are in 'detached HEAD' state**  
HEAD is what Git calls the commit you are currently on. You can “detach” the HEAD by switching to a previous commit, which we’ll see in the next video. Despite what it sounds like, it’s actually not a bad thing to detach the HEAD. Git just warns you so that you’ll realize you’re doing it.

**Panic! (the 'impossible' happened)**  
This is a real error message, but it’s not output by Git. Instead it’s output by GHC, the compiler for a programming language called Haskell. It’s reserved for particularly surprising errors!

**Takeaway** We hope these errors and warnings amused you as much as they amused us! Now that you know what kind of errors Git can throw, you’re ready to start checking out previous versions of files with Caroline.

### QuickEdit Mode

To make copying and pasting in GitBash easier by turning on QuickEdit mode, follow the instructions [here](https://www.udacity.com/wiki/ud775/git-bash-copy-paste).

### Most Recent Commit

The commit ID of the most recent commit is 3884eab839af1e82c44267484cf2945a766081f3. You can use this commit ID to return to the latest commit after checking out an older commit.

### Format of git checkout

The command Caroline types to checkout the "Revert controls" commit is git checkout b0678b161fcf74467ed3a63110557e3d6229cfa6.

### Windows Explorer

When Caroline mentions opening a "file navigation GUI" on Windows, she is referring to the Windows Explorer.

### Entering commit IDs

If it is easier, you may enter the first four or more characters of the commit ID rather than pasting the entire ID.

### Most Recent Commit

The commit ID of the most recent commit is 3884eab839af1e82c44267484cf2945a766081f3. You can use this commit ID to return to the latest commit after checking out an older commit.

## Reflect: Confidence from Version Control

Now that you know how to return to a previous version of your files using Git, go add the following question and your thoughts on it to your reflections file:

**How might using version control make you more confident to make changes that could break something?**

When you've updated your document, click "Next" to start configuring a workspace that will make you more efficient at using Git.

## Configure Windows Workspace

[**https://classroom.udacity.com/courses/ud775/lessons/2980038599/concepts/33417185870923**](https://classroom.udacity.com/courses/ud775/lessons/2980038599/concepts/33417185870923)

video: Setting Up Your Workspace on Windows

### Changing background color

If you prefer the background color of Git Bash to be something other than black, you can change it in the "Defaults" menu under the "Colors" tab. If you like the background color as-is, you don't need to make any changes.

### Downloading necessary files

* Save [this file](https://raw.githubusercontent.com/git/git/master/contrib/completion/git-completion.bash) in your home directory with the name git-completion.bash.
* Save [this file](https://raw.githubusercontent.com/git/git/master/contrib/completion/git-prompt.sh) in your home directory with the name git-prompt.sh.
* Download bash\_profile\_course from the Downloadables section.
* If you already have a file in your home directory named .bash\_profile, copy the content from bash\_profile\_course and paste it at the bottom of .bash\_profile. Otherwise, move bash\_profile\_course to your home directory and rename it to .bash\_profile. (If you're curious to learn more about how bash prompts work, see [this page](http://www.cyberciti.biz/tips/howto-linux-unix-bash-shell-setup-prompt.html).)

### Making Git configurations

Run the following Git configuration commands. The first one will need to be modified if you are using a text editor other than Sublime, or if Sublime is installed in another location for you. See [this page](https://help.github.com/articles/associating-text-editors-with-git/) for the correct command for a couple of other popular text editors. For any other editor, you'll need to enter the command you use to launch that editor from Git Bash.

git config --global core.editor "'C:/Program Files/Sublime Text 2/sublime\_text.exe' -n -w"

git config --global push.default upstream

git config --global merge.conflictstyle diff3

### Make sure you can start your editor from Git Bash

If you use Sublime, you can do this by adding the following line to your .bash\_profile:

alias subl="C:/Program\ Files/Sublime\ Text\ 2/sublime\_text.exe"

### Restart Git Bash

You'll need to close and re-open Git Bash before all your changes take effect.

#### Supporting Materials

[bash\_profile\_course](https://www.udacity.com/api/nodes/3341718587/supplemental_media/bash-profile-course/download" \t "_blank)

## Tracking versions using Git

**You would be able to see the difference between the two versions, but you would no longer be able to directly access the old version.**

This is false. You could still access the old version of the file by checking out the commit associated with that version. Then the recipe would temporarily revert to its state at the time that commit was made.

**Using git diff to compare the two versions would show the same changes as diff -u did in the previous exercise.**

This is true. diff -u and git diff show very similar outputs. Even if the exact format was slightly different, the actual changes indicated would be the same.

**The name of the file would change when you saved a second version in Git.**

This is false. The name of the file would remain the same. Git does not rename files when you save a new commit. Instead, Git uses the commit IDs to refer to different versions of the files, and you can use git checkout to access old versions of your files.

**To save two versions of the file, you would create two commits.**

This is true. Commits are Git's way of saving versions, so to save two different versions, you would create two commits.

## Git command review

**Compare two commits, printing each line that is present in one commit but not the other.**

git diff will do this. It takes two arguments - the two commit ids to compare.

**Make a copy of an entire Git repository, including the history, onto your own computer.**

git clone will do this. It takes one argument - the url of the repository to copy.

**Temporarily reset all files in a directory to their state at the time of a specific commit.**

git checkout will do this. It takes one argument - the commit ID to restore.

**Show the commits made in this repository, starting with the most recent.**

git log will do this. It doesn't take any arguments.

## Behavior of git clone

**If someone else gives you the location of their directory or repository, you can copy or clone it to your own computer.**

This is true for both copying a directory and cloning a repository.

As you saw in the previous lesson, if you have a URL to a repository, you can copy it to your computer using git clone.

For copying a directory, you weren't expected to know this, but it is possible to copy a directory from one computer to another using the command scp, which stands for "secure copy". The name was chosen because the scp command lets you securely copy a directory from one computer to another.

**The history of changes to the directory or repository is copied.**

This is true for cloning a repository, but not for copying a directory. The main reason to use git clone rather than copying the directory is because git clone will also copy the commit history of the repository. However, copying can be done on any directory, whereas git clone only works on a Git repository.

**If you make changes to the copied directory or cloned repository, the original will not change.**

This is true for both copying a directory and cloning a repository. In both cases, you're making a copy that you can alter without changing the original.

**The state of every file in the directory or repository is copied.**

This is true for both copying a directory and cloning a repository. In both cases, all the files are copied.

## Behavior of git log

git log lists the most recent commit first, as you can verify by checking the commit dates. The middle commit probably contains the code for the mute button, since the commit message indicates that the mute button was added in that commit. The top commit also probably contain the mute button, since that commit is more recent and nothing suggests the mute button has been removed. The bottom commit probably does not contain the mute button, since that commit was created before the commit that added the mute button.

To summarize:

* commit 7be5a12f1567866b0d77ccdf2055d1a33831da78 (the top commit listed)

Yes, probably contains the mute button.

* commit 06d72e1f95f046002ec46f41cf71957227111141 (the middle commit listed)

Yes, probably contains the mute button.

* commit 3d4d45b246aad6a1cd0afaf7cfae26966110727e (the bottom commit listed)

No, probably does not contain the mute button.

## git log output

For reference, here is the git log output again:

commit 7be5a12f1567866b0d77ccdf2055d1a33831da78

Author: Ellison Leão <el@gmail.com>

Date: Fri Jul 11 12:56:26 2014 -0300

Add sound for the wing.

commit 06d72e1f95f046002ec46f41cf71957227111141

Author: Ellison Leão <el@gmail.com>

Date: Wed Jul 9 23:42:55 2014 -0300

Add mute button.

commit 3d4d45b246aad6a1cd0afaf7cfae26966110727e

Author: Ellison Leão <el@gmail.com>

Date: Mon Jul 7 17:35:47 2014 -0300

Fix leaderboard button

## Behavior of git diff

The middle commit, 06d72e, is the first commit with the mute button, so comparing that commit and the previous commit, 3d4d45, would show the changes that add the mute button.

In order for the changes adding the mute button to be shown as additions, the commit with the mute button needs to be the second argument to git diff. That is because git diff considers the first argument as the "original", and the second argument as the "new" version, so additions are lines present in the second argument but not the first.

Thus, the last command listed, git diff 3d4d45 06d72e, is correct, and would show the mute button lines as additions. Reversing the arguments and running git diff 06d72e 3d4d45 would instead show the mute button lines as deletions.

## git log output

For reference, here is the git log output again:

commit 7be5a12f1567866b0d77ccdf2055d1a33831da78

Author: Ellison Leão <el@gmail.com>

Date: Fri Jul 11 12:56:26 2014 -0300

Add sound for the wing.

commit 06d72e1f95f046002ec46f41cf71957227111141

Author: Ellison Leão <el@gmail.com>

Date: Wed Jul 9 23:42:55 2014 -0300

Add mute button.

commit 3d4d45b246aad6a1cd0afaf7cfae26966110727e

Author: Ellison Leão <el@gmail.com>

Date: Mon Jul 7 17:35:47 2014 -0300

Fix leaderboard button

**Behavior of git checkout**

**Checking out an earlier commit will change the state of at least one file.**

This is sometimes true. Git doesn't allow you to save a new commit if no files have been updated, so you might think this is always true. However, it's possible to do the following:

* Save a commit (call this commit 1).
* Update some files and save another commit (call this commit 2).
* Change all the files back to their state during commit 1, then save again (call this commit 3).

This sometimes happens if commit 2 contained a bug, and it's important to fix the bug quickly. The easiest thing to do might be to remove all the changes introduced by commit 2 to fix the bug, then figure out how to safely reintroduce the changes later.

At this point, commit 3 is the latest commit, so if you checkout commit 1, none of the files will be changed.

**Checking out an earlier commit will change the state of more than one file.**

**Checking out an earlier commit will change the state of every file in the repository.**

Both of these are sometimes true. Since each commit tracks the state of all files in the repository, it is possible that checking out an earlier commit will change the state of multiple files, or even all the files in the repository. However, it is possible to save a new commit after changing only one file, so it is possible only one file will change.

**After checking out a commit, the state of all the files in the repository will be from the same point in time.**

This is always true. A commit saves a snapshot of all files in the repository at the time the commit was made, so checking out an earlier commit will result in all the files being reverted to their state at the time the commit was made. That is, the files will be in a consistent state.