# Git diff

## Comparing changes with git diff

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more. This document will discuss common invocations of git diff and diffing work flow patterns. The git diff command is often used along with git status and git log to analyze the current state of a Git repo.

## Reading diffs: outputs

### Raw output format

The following examples will be executed in a simple repo. The repo is created with the commands below:

$:> mkdir diff\_test\_repo

$:> cd diff\_test\_repo

$:> touch diff\_test.txt

$:> echo "this is a git diff test example" > diff\_test.txt

$:> git init .

Initialized empty Git repository in /Users/kev/code/test/.git/

$:> git add diff\_test.txt

$:> git commit -am"add diff test file"

[master (root-commit) 6f77fc3] add diff test file

1 file changed, 1 insertion(+)

create mode 100644 diff\_test.txt

If we execute git diff at this point, there will be no output. This is expected behavior as there are no changes in the repo to diff. Once the repo is created and we've added the diff\_test.txt file, we can change the contents of the file to start experimenting with diff output.

$:> echo "this is a diff example" > diff\_test.txt

Executing this command will change the content of the diff\_test.txt file. Once modified, we can view a diff and analyze the output. Now executing git diff will produce the following output:

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Let us now examine a more detailed breakdown of the diff output.

### 1. Comparison input

diff --git a/diff\_test.txt b/diff\_test.txt

This line displays the input sources of the diff. We can see that a/diff\_test.txt and b/diff\_test.txt have been passed to the diff.

### 2. Meta data

index 6b0c6cf..b37e70a 100644

This line displays some internal Git metadata. You will most likely not need this information. The numbers in this output correspond to Git object version hash identifiers.

### 3. Markers for changes

--- a/diff\_test.txt

+++ b/diff\_test.txt

These lines are a legend that assigns symbols to each diff input source. In this case, changes from a/diff\_test.txt are marked with a --- and the changes from b/diff\_test.txt are marked with the +++ symbol.

### 4. Diff chunks

The remaining diff output is a list of diff 'chunks'. A diff only displays the sections of the file that have changes. In our current example, we only have one chunk as we are working with a simple scenario. Chunks have their own granular output semantics.

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

The first line is the chunk header. Each chunk is prepended by a header inclosed within @@ symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes. In a more realistic diff, you would see a header like:

@@ -34,6 +34,8 @@

In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.

The remaining content of the diff chunk displays the recent changes. Each changed line is prepended with a + or - symbol indicating which version of the diff input the changes come from. As we previously discussed, - indicates changes from the a/diff\_test.txt and + indicates changes from b/diff\_test.txt.

## Highlighting changes

### 1. git diff --color-words

git diff also has a special mode for highlighting changes with much better granularity: ‐‐color-words. This mode tokenizes added and removed lines by whitespace and then diffs those.

$:> git diff --color-words

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

this is agit difftest example

Now the output displays only the color-coded words that have changed.

### 2. git diff-highlight

If you clone the git source, you’ll find a sub-directory called contrib. It contains a bunch of git-related tools and other interesting bits and pieces that haven’t yet been promoted to git core. One of these is a Perl script called diff-highlight. Diff-highlight pairs up matching lines of diff output and highlights sub-word fragments that have changed.

$:> git diff | /your/local/path/to/git-core/contrib/diff-highlight/diff-highlight

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Now we’ve pared down our diff to the smallest possible change.

## Diffing binary files

In addition to the text file utilities we have thus far demonstrated, git diff can be run on binary files. Unfortunately, the default output is not very helpful.

$:> git diff

Binary files a/script.pdf and b/script.pdf differ

Git does have a feature that allows you to specify a shell command to transform the content of your binary files into text prior to performing the diff. It does require a little set up though. First, you need to specify a textconv filter describing how to convert a certain type of binary to text. We're using a simple utility called pdftohtml (available via homebrew) to convert my PDFs into human readable HTML. You can set this up for a single repository by editing your .git/config file, or globally by editing ~ /.gitconfig

[diff "pdfconv"]

textconv=pdftohtml -stdout

Then all you need to do is associate one or more file patterns with our pdfconv filter. You can do this by creating a .gitattributes file in the root of your repository.

\*.pdf diff=pdfconv

Once configured, git diff will first run the binary file through the configured converter script and diff the converter output. The same technique can be applied to get useful diffs from all sorts of binary files, for example: zips, jars and other archives: using unzip -l (or similar) in place of pdf2html will show you paths that have been added or removed between commits images: exiv2 can be used to show metadata changes such as image dimensions documents: conversion tools exist for transforming .odf, .doc and other document formats to plain text. In a pinch, strings will often work for binary files where no formal converter exists.

## Comparing files: git diff file

The git diff command can be passed an explicit file path option. When a file path is passed to git diff the diff operation will be scoped to the specified file. The below examples demonstrate this usage.

git diff HEAD ./path/to/file

This example is scoped to ./path/to/file when invoked, it will compare the specific changes in the working directory, against the index, showing the changes that are not staged yet. By default git diff will execute the comparison against HEAD. Omitting HEAD in the example above git diff ./path/to/file has the same effect.

git diff --cached ./path/to/file

When git diff is invoked with the --cached option the diff will compare the staged changes with the local repository. The --cached option is synonymous with --staged.

## Comparing all changes

Invoking git diff without a file path will compare changes across the entire repository. The above, file specific examples, can be invoked without the ./path/to/file argument and have the same output results across all files in the local repo.

## Changes since last commit

By default git diff will show you any uncommitted changes since the last commit.

git diff

## Comparing files between two different commits

git diff can be passed Git refs to commits to diff. Some example refs are, HEAD, tags, and branch names. Every commit in Git has a commit ID which you can get when you execute GIT LOG. You can also pass this commit ID to git diff.

$:> git log --prety=oneline

957fbc92b123030c389bf8b4b874522bdf2db72c add feature

ce489262a1ee34340440e55a0b99ea6918e19e7a rename some classes

6b539f280d8b0ec4874671bae9c6bed80b788006 refactor some code for feature

646e7863348a427e1ed9163a9a96fa759112f102 add some copy to body

$:> git diff 957fbc92b123030c389bf8b4b874522bdf2db72c ce489262a1ee34340440e55a0b99ea6918e19e7a

## Comparing branches

### Comparing two branches

Branches are compared like all other ref inputs to git diff

git diff branch1..other-feature-branch

This example introduces the dot operator. The two dots in this example indicate the diff input is the tips of both branches. The same effect happens if the dots are omitted and a space is used between the branches. Additionally, there is a three dot operator:

git diff branch1...other-feature-branch

The three dot operator initiates the diff by changing the first input parameter branch1. It changes branch1 into a ref of the shared common ancestor commit between the two diff inputs, the shared ancestor of branch1 and other-feature-branch. The last parameter input parameter remains unchanged as the tip of other-feature-branch.

## Comparing files from two branches

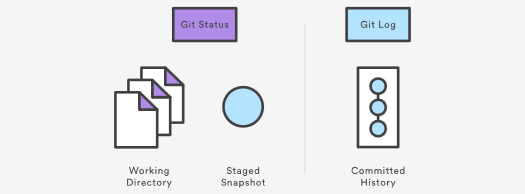
To compare a specific file across branches, pass in the path of the file as the third argument to git diff

git diff master new\_branch ./diff\_test.txt

# Inspecting a repository

## git log

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.



Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.

The git log command is Git's basic tool for exploring a repository’s history. It’s what you use when you need to find a specific version of a project or figure out what changes will be introduced by merging in a feature branch.

commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7

Author: John Smith

Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commit is an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

This ID can be used in commands like git log <since>..<until> to refer to specific commits. For instance, git log 3157e..5ab91 will display everything between the commits with ID's 3157e and 5ab91. Aside from checksums, branch names (discussed in the [Branch Module](https://www.atlassian.com/git/tutorials/using-branches)) and the HEAD keyword are other common methods for referring to individual commits. HEAD always refers to the current commit, be it a branch or a specific commit.

The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.

The idea behind all of these identification methods is to let you perform actions based on specific commits. The git log command is typically the starting point for these interactions, as it lets you find the commits you want to work with.

### Usage

$ git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

$ git log -n <limit>

Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.

$ git log --oneline

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

$ git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

$ git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

$ git log --author="<pattern>"

Search for commits by a particular author. The <pattern> argument can be a plain string or a regular expression.

$ git log --grep="<pattern>"

Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.

$ git log <since>..<until>

Show only commits that occur between <since> and <until>. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).

$ git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

$ git log --graph --decorate --oneline

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

### Discussion

### Example

The Usage section provides many examples of git log, but keep in mind that several options can be combined into a single command:

git log --author="John Smith" -p hello.py

This will display a full diff of all the changes John Smith has made to the file hello.py.

The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in master.

git log --oneline master..some-feature

# Git blame

* + The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

The git blame command is a versatile troubleshooting utility that has extensive usage options. The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to examine specific points of a file's history and get context as to who the last author was that modified the line. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

Git blame is often used with a GUI display. Online Git hosting sites like [Bitbucket](http://bitbucket-marketing.atlassian.com/product) offer blame views which are UI wrappers to git blame. These views are referenced in collaborative discussions around pull requests and commits. Additionally, most IDE's that have Git integration also have dynamic blame views.

## How It Works

In order to demonstrate git blame we need a repository with some history. We will use the open source project [git-blame-example](https://bitbucket.org/kevzettler/git-blame-example). This open source project is a simple repository that contains a README.md file which has a few commits from different authors. The first step of our git blame usage example is to git clone the example repository.

git clone https://kevzettler@bitbucket.org/kevzettler/git-blame-example.git && cd git-blame-example

Now that we have a copy of the example code we can start exploring it with git blame. The state of the example repo can be examined using [git log](https://www.atlassian.com/git/tutorials/git-log). The commit history should look like the following:

$ git log

commit 548dabed82e4e5f3734c219d5a742b1c259926b2

Author: Juni Mukherjee <jmukherjee@atlassian.com>

Date: Thu Mar 1 19:55:15 2018 +0000

Another commit to help git blame track the who, the what, and the when

commit eb06faedb1fdd159d62e4438fc8dbe9c9fe0728b

Author: Juni Mukherjee <jmukherjee@atlassian.com>

Date: Thu Mar 1 19:53:23 2018 +0000

Creating the third commit, along with Kev and Albert, so that Kev can get git blame docs.

commit 990c2b6a84464fee153253dbf02e845a4db372bb

Merge: 82496ea 89feb84

Author: Albert So <aso@atlassian.com>

Date: Thu Mar 1 05:33:01 2018 +0000

Merged in albert-so/git-blame-example/albert-so/readmemd-edited-online-with-bitbucket-1519865641474 (pull request #2)

README.md edited online with Bitbucket

commit 89feb84d885fe33d1182f2112885c2a64a4206ec

Author: Albert So <aso@atlassian.com>

Date: Thu Mar 1 00:54:03 2018 +0000

README.md edited online with Bitbucket

git blame only operates on individual files. A file-path is required for any useful output. The default execution of git blame will simply output the commands help menu. For this example, we will operate on the README.MD file. It is a common open source software practice to include a README file in the root of a git repository as documentation source for the project.

git blame README.MD

Executing the above command will give us our first sample of blame output. The following output is a subset of the full blame output of the README. Additionally, this output is static is reflective of the state of the repo at the time of this writing.

$ git blame README.md

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 1) # Git Blame example

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 2)

89feb84d (Albert So 2018-03-01 00:54:03 +0000 3) This repository is an example of a project with multiple contributors making commits.

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 4)

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 5) The repo use used elsewhere to demonstrate `git blame`

82496ea3 (kevzettler 2018-02-28 13:37:02 -0800 6)

89feb84d (Albert So 2018-03-01 00:54:03 +0000 7) Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod TEMPOR incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum

89feb84d (Albert So 2018-03-01 00:54:03 +0000 8)

eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 9) Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.

eb06faed (Juni Mukherjee 2018-03-01 19:53:23 +0000 10)

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 11) Creating a line to support documentation needs for git blame.

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 12)

548dabed (Juni Mukherjee 2018-03-01 19:55:15 +0000 13

This is a sample of the first 13 lines of the README.md file. To better understand this output lets break down a line. The following table displays the content of line 3 and the columns of the table indicate the column content.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Author | Timestamp | Line Number | Line Content |
| 89feb84d | Albert So | 2018-03-01 00:54:03 +0000 | 3 | This repository is an example of a project with multiple contributors making commits. |

If we review the blame output list, we can make some observations. There are three authors listed. In addition to the project's maintainer Kev Zettler, Albert So, and Juni Mukherjee are also listed. Authors are generally the most valuable part of git blame output. The timestamp column is also primarily helpful. What the change was is indicated by line content column.

## Common Options

git blame -L 1,5 README.md

The -L option will restrict the output to the requested line range. Here we have restricted the output to lines 1 through 5.

git blame -e README.md

The -e option shows the authors email address instead of username.

git blame -w README.md

The -w option ignores whitespace changes. If a previous author has modified the spacing of a file by switching from tabs to spaces or adding new lines this, unfortunately, obscures the output of git blame by showing these changes.

git blame -M README.md

The -M option detects moved or copied lines within in the same file. This will report the original author of the lines instead of the last author that moved or copied the lines.

git blame -C README.md

The -C option detects lines that were moved or copied from other files. This will report the original author of the lines instead of the last author that moved or copied the lines.

## Git Blame vs Git Log

While git blame displays the last author that modified a line, often times you will want to know when a line was originally added. This can be cumbersome to achieve using git blame. It requires a combination of the -w, -C, and -M options. It can be far more convenient to use the [git log](https://www.atlassian.com/git/tutorials/git-log) command.

To list all original commits in-which a specific code piece was added or modified execute git log with the -S option. Append the -S option with the code you are looking for. Let's take one of the lines from the README output above to use as an example. Let us take the text "CSS3D and WebGL renderers" from Line 12 of the README output.

$ git log -S"CSS3D and WebGL renderers." --pretty=format:'%h %an %ad %s'

e339d3c85 Mario Schuettel Tue Oct 13 16:51:06 2015 +0200 reverted README.md to original content

509c2cc35 Daniel Tue Sep 8 13:56:14 2015 +0200 Updated README

cb20237cc Mr.doob Mon Dec 31 00:22:36 2012 +0100 Removed DOMRenderer. Now with the CSS3DRenderer it has become irrelevant.

This output shows us that content from the README was added or modified 3 times by 3 different authors. It was originally added in commit cb20237cc by Mr.doob. In this example, git log has also been prepended with the --pretty-format option. This option converts the default output format of git log into one that matches the format of git log. For more information on usage and configuration options visit the [git log](https://www.atlassian.com/git/tutorials/git-log) page.

## Summary

The git blame command is used to examine the contents of a file line by line and see when each line was last modified and who the author of the modifications was. The output format of git blame can be altered with various command line options. Online Git hosting solutions like Bitbucket offer blame views, which offer a superior user experience to command line git blame usage. git blame and git log can be used in combination to help discover the history of a file's contents. The git log command has some similar blame functionality, to learn more visit the [git log](https://www.atlassian.com/git/tutorials/git-log) overview page.