## **Revision Selection**

Git allows you to refer to a single commit, set of commits, or range of commits in a number of ways.

**Single Revisions**

You can obviously refer to any single commit by its full, 40-character SHA-1 hash, but there are more human-friendly ways to refer to commits as well.

**Short SHA-1**

Git is smart enough to figure out what commit you’re referring to if you provide the first few characters of the SHA-1 hash, as long as that partial hash is at least four characters long and unambiguous.

If you want to see which specific SHA-1 a branch points to

$ git rev-parse topic1

ca82a6dff817ec66f44342007202690a93763949

**RefLog**

“reflog” A log of where your HEAD and branch references have been for the last few months.

**$ git reflog**

734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated

d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.

Every time your branch tip is updated for any reason, Git stores that information for you in this temporary history. You can use your reflog data to refer to older commits as well.

If you want to see the fifth prior value of the HEAD of your repository, you can use the @{5} reference that you see in the reflog output:

$ git show HEAD@{5}

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your master branch was yesterday, you can type.

$ git show master@{yesterday}

# That would show you where tip of your master branch was yesterday. This technique only works for data that’s still in your reflog, so you can’t use it to look for commits older than a few months.

It’s important to note that reflog information is strictly local — it’s a log only of what *you’ve* done in *your* repository. The references won’t be the same on someone else’s copy of the repository; also, right after you initially clone a repository, you’ll have an empty reflog, as no activity has occurred yet in your repository.

**Ancenstry Reference**

If you place a ^ (caret) at the end of a reference, Git resolves it to mean the parent of that commit.

you can see the previous commit by specifying HEAD^, which means “the parent of HEAD”

$ git show HEAD^

HEAD~ and HEAD^ are equivalent.

HEAD~2 means “the first parent of the first parent,” or “the grandparent” — it traverses the first parents the number of times you specify.

**Commit Ranges**

This is particularly useful for managing your branches — if you have a lot of branches, you can use range specifications to answer questions such as, “What work is on this branch that I haven’t yet merged into my main branch?”

**Double Dot**

This basically asks Git to resolve a range of commits that are reachable from one commit but aren’t reachable from another.



# Say you want to see what is in your experiment branch that hasn’t yet been merged into your master branch.

#master..experiment means “all commits reachable from experiment that aren’t reachable from master

$ git log master..experiment

D

C

# experiment..master shows you everything in master not reachable from experiment:

$ git log experiment..master

F

E

$ git log origin/master..HEAD

This command shows you any commits in your current branch that aren’t in the master branch on your origin remote. If you run a git push and your current branch is tracking origin/master, the commits listed by git log origin/master..HEAD are the commits that will be transferred to the server.

Multiple Points

$ git log refA refB ^refC

$ git log refA refB --not refC

you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from refA or refB but not from refC, you can use either of

**Triple Dot**

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them.

If you want to see what is in master or experiment but not any common references, you can run:

$ git log master...experiment

F

E

D

C

## **Interative Staging**

These tools are helpful if you modify a number of files extensively, then decide that you want those changes to be partitioned into several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate changesets and can be reviewed easily by the developers working with you

**Interative Staging**

$ git add -i

staged unstaged path

1: unchanged +0/-1 TODO

2: unchanged +1/-1 index.html

3: unchanged +5/-1 lib/simplegit.rb

\*\*\* Commands \*\*\*

1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd untracked

5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now>

Staging Patches

It’s also possible for Git to stage certain *parts* of files and not the rest. For example, if you make two changes to your simplegit.rb file and want to stage one of them and not the other.

$ git add -i

Type p

Then type s for split y for yes n for no

or $ git add -p <fle name>

## **Stashing and Cleaning**

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

**Stashing your work**

$ git status

$ git stash Staged & unstaged

# The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

**Re-apply your stashed changed**

$ git stash pop By default pop recent stash

**#** Move the recent stash to Workinf directory

$ git stash apply

# Copy recent stash to working directory

**Cleaning up your Stash**

$ git stash drop # deletes one by one

$ git stash clear # Deletes alll

**Creative Stash**

$ git stash –keep-index #It keeps stashed file in working directory

**Stashing Untracked or ignored files**

$ git stash -u # stash untracked files except .ignore file

$ git stash -a # Stash all incuding .ignore files

**Stashing patch**

$ git stash --patch

Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory

**Creating a Branch from a stash**

$ git stash branch <new\_branch>

**Cleaning your working directory**

$ git clean -f -d # To remove all the untracked files in your working directory

Dry run [-n] $ git clean -d -n

## **Signing your work**