gitrevisions

gitrevisions(7) ==========

NAME ---- gitrevisions - Specifying revisions and ranges for Git

DESCRIPTION -----------

Many Git commands take revision parameters as arguments. Depending on the command, they denote a specific commit or, for commands which walk the revision graph (such as linkgit:git-log[1]), all commits which are reachable from that commit. For commands that walk the revision graph one can also specify a range of revisions explicitly.

In addition, some Git commands (such as linkgit:git-show[1] and linkgit:git-push[1]) can also take revision parameters which denote other objects than commits, e.g. blobs ("files") or trees ("directories of files").

SPECIFYING REVISIONS --------------------

A revision parameter ’<rev>’ typically, but not necessarily, names a commit object. It uses what is called an ’extended SHA-1’ syntax. Here are various ways to spell object names. The ones listed near the end of this list name trees and blobs contained in a commit.

> This document shows the "raw" syntax as seen by git. The shell and other UIs might require additional quoting to protect special characters and to avoid word splitting.

’<sha1>’, e.g. ’dae86e1950b1277e545cee180551750029cfe735’, ’dae86e’
The full SHA-1 object name (40-byte hexadecimal string), or a leading substring that is unique within the repository. E.g. dae86e1950b1277e545cee180551750029cfe735 and dae86e both name the same commit object if there is no other object in your repository whose object name starts with dae86e.

’<describeOutput>’, e.g. ’v1.7.4.2-679-g3bee7fb’
Output from `git` `describe`; i.e. a closest tag, optionally followed by a dash and a number of commits, followed by a dash, a ’g’, and an abbreviated object name.

’<refname>’, e.g. ’master’, ’heads/master’, ’refs/heads/master’
A symbolic ref name. E.g. ’master’ typically means the commit object referenced by ’refs/heads/master’. If you happen to have both ’heads/master’ and ’tags/master’, you can explicitly say ’heads/master’ to tell Git which one you mean. When ambiguous, a ’<refname>’ is

disambiguated by taking the first match in the following rules:

1. If ’$GIT_DIR/<refname>’ exists, that is what you mean (this is usually useful only for `HEAD`, `FETCH_HEAD`, `ORIG_HEAD`, `MERGE_HEAD`, `REBASE_HEAD`, `REVERT_HEAD`, `CHERRY_PICK_HEAD`, `BISECT_HEAD` and `AUTO_MERGE`);
2. otherwise, ’refs/<refname>’ if it exists;

1. otherwise, ’refs/tags/<refname>’ if it exists;

1. otherwise, ’refs/heads/<refname>’ if it exists;

1. otherwise, ’refs/remotes/<refname>’ if it exists;

1. otherwise, ’refs/remotes/<refname>/HEAD’ if it exists.

`HEAD`::: names the commit on which you based the changes in the working tree. `FETCH_HEAD`::: records the branch which you fetched from a remote repository with your last `git fetch` invocation. `ORIG_HEAD`::: is created by commands that move your `HEAD` in a drastic way (`git am`, `git merge`, `git rebase`, `git reset`), to record the position of the `HEAD` before their operation, so that you can easily change the tip of the branch back to the state before you ran them. `MERGE_HEAD`::: records the commit(s) which you are merging into your branch when you run `git merge`. `REBASE_HEAD`::: during a rebase, records the commit at which the operation is currently stopped, either because of conflicts or an `edit` command in an interactive rebase. `REVERT_HEAD`::: records the commit which you are reverting when you run `git revert`. `CHERRY_PICK_HEAD`::: records the commit which you are cherry-picking when you run `git cherry-pick`. `BISECT_HEAD`::: records the current commit to be tested when you run `git bisect --no-checkout`. `AUTO_MERGE`::: records a tree object corresponding to the state the ’ort’ merge strategy wrote to the working tree when a merge operation resulted in conflicts.

Note that any of the ’refs/*’ cases above may come either from the `$GIT_DIR/refs` directory or from the `$GIT_DIR/packed-refs` file. While the ref name encoding is unspecified, UTF-8 is preferred as some output processing may assume ref names in UTF-8.

’@’
’@’ alone is a shortcut for `HEAD`.

’[<refname>]@{<date>}’, e.g. ’master@{yesterday}’, ’HEAD@{5 minutes ago}’
A ref followed by the suffix ’@’ with a date specification enclosed in a brace pair (e.g. ’{yesterday}’, ’{1 month 2 weeks 3 days 1 hour 1 second ago}’ or ’{1979-02-26 18:30:00}’) specifies the value of the ref at a prior point in time. This suffix may only be used immediately following a ref name and the ref must have an existing log (’$GIT_DIR/logs/<ref>’). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local ’master’ branch last week. If you want to look at commits made during certain times, see `--since` and `--until`.

’<refname>@{<n>}’, e.g. ’master@{1}’
A ref followed by the suffix ’@’ with an ordinal specification enclosed in a brace pair (e.g. ’{1}’, ’{15}’) specifies the n-th prior value of that ref. For example ’master@{1}’ is the immediate prior value of ’master’ while ’master@{5}’ is the 5th prior value of ’master’. This suffix may only be used immediately following a ref name and the ref must have an existing log (’$GIT_DIR/logs/<refname>’).

’@{<n>}’, e.g. ’@{1}’
You can use the ’@’ construct with an empty ref part to get at a reflog entry of the current branch. For example, if you are on branch ’blabla’ then ’@{1}’ means the same as ’blabla@{1}’.

’@{-<n>}’, e.g. ’@{-1}’
The construct ’@{-<n>}’ means the <n>th branch/commit checked out before the current one.

’[<branchname>]@{upstream}’, e.g. ’master@{upstream}’, ’@{u}’
A branch B may be set up to build on top of a branch X (configured with `branch.<name>.merge`) at a remote R (configured with `branch.<name>.remote`). B@{u} refers to the remote-tracking branch for the branch X taken from remote R, typically found at `refs/remotes/R/X`.

’[<branchname>]@{push}’, e.g. ’master@{push}’, ’@{p}’
The suffix ’@{push}’ reports the branch "where we would push to" if `git` `push` were run while `branchname` was checked out (or the current `HEAD` if no branchname is specified). Like for ’@{upstream}’, we report the remote-tracking branch that corresponds to that branch at the remote. If the push destination is a URL and exactly one configured remote uses that URL for pushing, ’@{push}’ reports that remote’s remote-tracking branch.

Here’s an example to make it more clear:

```
$ git config push.default current
$ git config remote.pushdefault myfork
$ git switch -c mybranch origin/master

$ git rev-parse --symbolic-full-name @{upstream}
refs/remotes/origin/master

$ git rev-parse --symbolic-full-name @{push}
refs/remotes/myfork/mybranch
```

Note in the example that we set up a triangular workflow, where we pull from one location and push to another. In a non-triangular workflow, ’@{push}’ is the same as ’@{upstream}’, and there is no need for it.

This suffix is also accepted when spelled in uppercase, and means the same thing no matter the case.

’<rev>^[<n>]’, e.g. ’HEAD^, v1.5.1^0’
A suffix ’^’ to a revision parameter means the first parent of that commit object. ’^<n>’ means the <n>th parent (i.e. ’<rev>^’ is equivalent to ’<rev>^1’). As a special rule, ’<rev>^0’ means the commit itself and is used when ’<rev>’ is the object name of a tag object that refers to a commit object.

’<rev>~~[<n>]’, e.g. ’HEAD~~, master~~3’~~
~~A suffix ’~~’ to a revision parameter means the first parent of that commit object. A suffix ’~<n>’ to a revision parameter means the commit object that is the <n>th generation ancestor of the named commit object, following only the first parents. I.e. ’<rev>~3’ is equivalent to ’<rev>^^^’ which is equivalent to ’<rev>^1^1^1’. See below for an illustration of the usage of this form.

’<rev>^{<type>}’, e.g. ’v0.99.8^{commit}’
A suffix ’^’ followed by an object type name enclosed in brace pair means dereference the object at ’<rev>’ recursively until an object of type ’<type>’ is found or the object cannot be dereferenced anymore (in which case, barf). For example, if ’<rev>’ is a commit-ish, ’<rev>^{commit}’ describes the corresponding commit object. Similarly, if ’<rev>’ is a tree-ish, ’<rev>^{tree}’ describes the corresponding tree object. ’<rev>^0’ is a short-hand for ’<rev>^{commit}’.

’<rev>^{object}’ can be used to make sure ’<rev>’ names an object that exists, without requiring ’<rev>’ to be a tag, and without dereferencing ’<rev>’; because a tag is already an object, it does not have to be dereferenced even once to get to an object.

’<rev>^{tag}’ can be used to ensure that ’<rev>’ identifies an existing tag object.

’<rev>^{}’, e.g. ’v0.99.8^{}’
A suffix ’^’ followed by an empty brace pair means the object could be a tag, and dereference the tag recursively until a non-tag object is found.

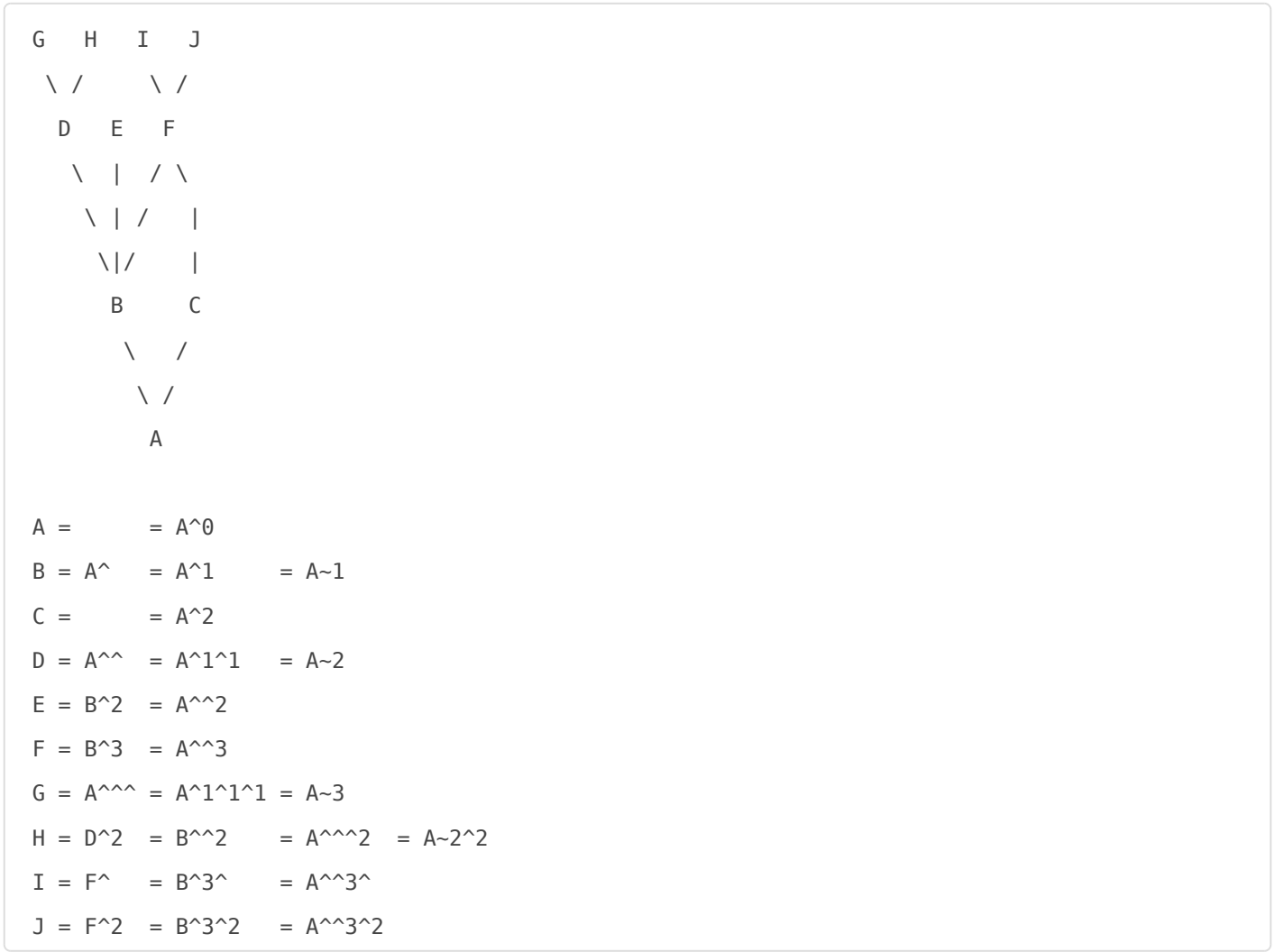
’<rev>^{/<text>}’, e.g. ’HEAD^{/fix nasty bug}’
A suffix ’^’ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the ’:/fix nasty bug’ syntax below except that it returns the youngest matching commit which is reachable from the ’<rev>’ before ’^’.

’:/<text>’, e.g. ’:/fix nasty bug’
A colon, followed by a slash, followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from any ref, including HEAD. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. ’:/^foo’. The special sequence ’:/!’ is reserved for modifiers to what is matched. ’:/!-foo’ performs a negative match, while ’:/!!foo’ matches a literal ’!’ character, followed by ’foo’. Any other sequence beginning with ’:/!’ is reserved for now. Depending on the given text, the shell’s word splitting rules might require additional quoting.

’<rev>:<path>’, e.g. ’HEAD:README’, ’master:./README’
A suffix ’:’ followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. A path starting with ’./’ or ’../’ is relative to the current working directory. The given path will be converted to be relative to the working tree’s root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

’:[<n>:]<path>’, e.g. ’:0:README’, ’:README’
A colon, optionally followed by a stage number (0 to 3) and a colon, followed by a path, names a blob object in the index at the given path. A missing stage number (and the colon that follows it) names a stage 0 entry. During a merge, stage 1 is the common ancestor, stage 2 is the target branch’s version (typically the current branch), and stage 3 is the version from the branch which is being merged.

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.



SPECIFYING RANGES ----------

History traversing commands such as `git` `log` operate on a set of commits, not just a single commit.

For these commands, specifying a single revision, using the notation described in the previous section, means the set of commits `reachable` from the given commit.

Specifying several revisions means the set of commits reachable from any of the given commits.

A commit’s reachable set is the commit itself and the commits in its ancestry chain.

There are several notations to specify a set of connected commits (called a "revision range"), illustrated below.

Commit Exclusions ~~~~~~~~~~~~~~~~~~

’^<rev>’ (caret) Notation
To exclude commits reachable from a commit, a prefix ’^’ notation is used. E.g. ’^r1 r2’ means commits reachable from ’r2’ but exclude the ones reachable from ’r1’ (i.e. ’r1’ and its ancestors).

Dotted Range Notations ~~~~~~~~~~~~~~~~~~~~~~

The ’..’ (two-dot) Range Notation
The ’^r1 r2’ set operation appears so often that there is a shorthand for it. When you have two commits ’r1’ and ’r2’ (named according to the syntax explained in SPECIFYING REVISIONS above), you can ask for commits that are reachable from r2 excluding those that are reachable from r1 by ’^r1 r2’ and it can be written as ’r1..r2’.

The ’...’ (three-dot) Symmetric Difference Notation
A similar notation ’r1...r2’ is called symmetric difference of ’r1’ and ’r2’ and is defined as ’r1 r2 --not $(git merge-base --all r1 r2)’. It is the set of commits that are reachable from either one of ’r1’ (left side) or ’r2’ (right side) but not from both.

In these two shorthand notations, you can omit one end and let it default to HEAD. For example, ’origin..’ is a shorthand for ’origin..HEAD’ and asks "What did I do since I forked from the origin branch?" Similarly, ’..origin’ is a shorthand for ’HEAD..origin’ and asks "What did the origin do since I forked from them?" Note that ’..’ would mean ’HEAD..HEAD’ which is an empty range that is both reachable and unreachable from HEAD.

Commands that are specifically designed to take two distinct ranges (e.g. "git range-diff R1 R2" to compare two ranges) do exist, but they are exceptions. Unless otherwise noted, all "git" commands that operate on a set of commits work on a single revision range. In other words, writing two "two-dot range notation" next to each other, e.g.

```
$ git log A..B C..D
```

does **not** specify two revision ranges for most commands. Instead it will name a single connected set of commits, i.e. those that are reachable from either B or D but are reachable from neither A or C. In a linear history like this:

```
---A---B---o---o---C---D
```

because A and B are reachable from C, the revision range specified by these two dotted ranges is a single commit D.

Other <rev>^ Parent Shorthand Notations ~~~~~~ Three other shorthands exist, particularly useful for merge commits, for naming a set that is formed by a commit and its parent commits.

The ’r1^@’ notation means all parents of ’r1’.

The ’r1^!’ notation includes commit ’r1’ but excludes all of its parents. By itself, this notation denotes the single commit ’r1’.

The ’<rev>^-[<n>]’ notation includes ’<rev>’ but excludes the <n>th parent (i.e. a shorthand for ’<rev>^<n>..<rev>’), with ’<n>’ = 1 if not given. This is typically useful for merge commits where you can just pass ’<commit>^-’ to get all the commits in the branch that was merged in merge commit ’<commit>’ (including ’<commit>’ itself).

While ’<rev>^<n>’ was about specifying a single commit parent, these three notations also consider its parents. For example you can say ’HEAD^2^@’, however you cannot say ’HEAD^@^2’.

Revision Range Summary ----------------------

’<rev>’
Include commits that are reachable from <rev> (i.e. <rev> and its ancestors).

’^<rev>’
Exclude commits that are reachable from <rev> (i.e. <rev> and its ancestors).

’<rev1>..<rev2>’
Include commits that are reachable from <rev2> but exclude those that are reachable from <rev1>. When either <rev1> or <rev2> is omitted, it defaults to `HEAD`.

’<rev1>...<rev2>’
Include commits that are reachable from either <rev1> or <rev2> but exclude those that are reachable from both. When either <rev1> or <rev2> is omitted, it defaults to `HEAD`.

’<rev>^@’, e.g. ’HEAD^@’
A suffix ’^’ followed by an at sign is the same as listing all parents of ’<rev>’ (meaning, include

anything reachable from its parents, but not the commit itself).

’<rev>^!’, e.g. ’HEAD^!’
A suffix ’^’ followed by an exclamation mark is the same as giving commit ’<rev>’ and all its parents prefixed with ’^’ to exclude them (and their ancestors).

’<rev>^-<n>’, e.g. ’HEAD^-, HEAD^-2’
Equivalent to ’<rev>^<n>..<rev>’, with ’<n>’ = 1 if not given.

Here are a handful of examples using the Loeliger illustration above, with each step in the notation’s expansion and selection carefully spelt out:

```
   Args   Expanded arguments    Selected commits
   D                            G H D
   D F                          G H I J D F
   ^G D                         H D
   ^D B                         E I J F B
   ^D B C                       E I J F B C
   C                            I J F C
   B..C   = ^B C                C
   B...C  = B ^F C              G H D E B C
   B^-    = B^..B
□  = ^B^1 B             E I J F B
   C^@    = C^1
□  = F                  I J F
   B^@    = B^1 B^2 B^3
□  = D E F              D G H E F I J
   C^!    = C ^C^@
□  = C ^C^1
□  = C ^F               C
   B^!    = B ^B^@
□  = B ^B^1 ^B^2 ^B^3
□  = B ^D ^E ^F         B
   F^! D  = F ^I ^J D          G H D F
```

SEE ALSO -------- linkgit:git-rev-parse[1]

GIT --- Part of the linkgit:git[1] suite

□□□ #1
□ □□□ □□□ 2026-10-31 20:14:28 UTC
□ □□□ □□□ 2026-10-31 20:14:28 UTC