

**NAME**

gitrevisions - specifying revisions and ranges for Git

**SYNOPSIS**

gitrevisions

**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 [git-log\(1\)](#)), all commits which can be reached from that commit. In the latter case one can also specify a range of revisions explicitly.

In addition, some Git commands (such as [git-show\(1\)](#)) 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.

*<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* and *CHERRY\_PICK\_HEAD*);
2. otherwise, *refs/<refname>* if it exists;
3. otherwise, *refs/tags/<refname>* if it exists;
4. otherwise, *refs/heads/<refname>* if it exists;
5. otherwise, *refs/remotes/<refname>* if it exists;
6. 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, 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`. *CHERRY\_PICK\_HEAD* records the commit which you are cherry-picking when you run `git cherry-pick`.

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 relog 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}`

The suffix `@{upstream}` to a branchname (short form `<branchname>@{u}`) refers to the branch that the branch specified by `branchname` is set to build on top of (configured with `branch.<name>.remote` and `branch.<name>.merge`). A missing `branchname` defaults to the current one.

`<rev>^`, 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. `master~3`

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. If the commit message starts with a `!` you have to repeat that; the special sequence `:/!`, followed by something else than `!`, is reserved for now. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. `:/^foo`.

`<rev>:<path>`, e.g. `HEAD:README`, `: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. `:path` (with an empty part before the colon) is a special case of the syntax described next: content recorded in the index at the given path. 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.

G H I J

/ /

D E F

| /

| / |

| / |

B C

/

/

A

$A = A^0$

$B = A^1 = A^1 = A^1$

$C = A^2 = A^2$

$D = A^{11} = A^{11} = A^2$

$E = B^2 = A^{22}$

$F = B^3 = A^{33}$

$G = A^{111} = A^{111} = A^3$

$H = D^2 = B^{22} = A^{222} = A^{222}$

$I = F^1 = B^{31} = A^{331}$

$J = F^2 = B^{32} = A^{332}$

## SPECIFYING RANGES

History traversing commands such as `git log` operate on a set of commits, not just a single commit. To these commands, specifying a single revision with the notation described in the previous section means the set of commits reachable from that commit, following the commit ancestry chain.

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`.

This 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`.

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* or *r2* but not from both.

In these two shorthands, 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.

Two other shorthands for naming a set that is formed by a commit and its parent commits exist. The `r1^@` notation means all parents of *r1*. `r1^!` includes commit *r1* but excludes all of its parents.

To summarize:

`<rev>`

Include commits that are reachable from (i.e. ancestors of) `<rev>`.

`^<rev>`

Exclude commits that are reachable from (i.e. ancestors of) `<rev>`.

`<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 then all its parents prefixed with `^` to exclude them (and their ancestors).

Here are a handful of examples:

```
D G H D
D F G H I J D F
^G D H D
^D B E I J F B
B..C C
B...C G H D E B C
^D B C E I J F B C
C I J F C
C^@ I J F
C^! C
F^! D G H D F
```

## SEE ALSO

[git-rev-parse\(1\)](#)

**GIT**

Part of the [git\(1\)](#) suite