cheat sheets.

$ cheat git

```
Setup
-----

git clone <repo>
  clone the repository specified by <repo>; this is similar to "checkout" in
  some other version control systems such as Subversion and CVS

Add colors to your ~/.gitconfig file:

  [color]
    ui = auto
  [color "branch"]
    current = yellow reverse
    local = yellow
    remote = green
  [color "diff"]
    meta = yellow bold
    frag = magenta bold
    old = red bold
    new = green bold
  [color "status"]
    added = yellow
    changed = green
    untracked = cyan

Highlight whitespace in diffs

  [color]
    ui = true
  [color "diff"]
    whitespace = red reverse
  [core]
    whitespace=fix,-indent-with-non-tab,trailing-space,cr-at-eol

Add aliases to your ~/.gitconfig file:

  [alias]
    st = status
    ci = commit
    br = branch
    co = checkout
    df = diff
    dc = diff --cached
    lg = log -p
    lol = log --graph --decorate --pretty=oneline --abbrev-commit
    lola = log --graph --decorate --pretty=oneline --abbrev-commit --all
    ls = ls-files

    # Show files ignored by git:
    ign = ls-files -o -i --exclude-standard


Configuration
-------------

git config -e [--global]
  edit the .git/config [or ~/.gitconfig] file in your $EDITOR

git config --global user.name 'John Doe'
git config --global user.email johndoe@example.com
  sets your name and email for commit messages

git config branch.autosetupmerge true
  tells git-branch and git-checkout to setup new branches so that git-pull(1)
  will appropriately merge from that remote branch.  Recommended.  Without this,
  you will have to add --track to your branch command or manually merge remote
  tracking branches with "fetch" and then "merge".

git config core.autocrlf true
  This setting tells git to convert the newlines to the system's standard
  when checking out files, and to LF newlines when committing in

git config --list
  To view all options

git config apply.whitespace nowarn
  To ignore whitespace

You can add "--global" after "git config" to any of these commands to make it
apply to all git repos (writes to ~/.gitconfig).


Info
----
git reflog
  Use this to recover from *major* mess ups! It's basically a log of the
  last few actions and you might have luck and find old commits that
  have been lost by doing a complex merge.

git diff
  show a diff of the changes made since your last commit
  to diff one file: "git diff -- <filename>"
  to show a diff between staging area and HEAD: `git diff --cached`

git status
  show files added to the staging area, files with changes, and untracked files

git log
  show recent commits, most recent on top. Useful options:
  --color       with color
  --graph       with an ASCII-art commit graph on the left
  --decorate    with branch and tag names on appropriate commits
  --stat        with stats (files changed, insertions, and deletions)
  -p            with full diffs
  --author=foo  only by a certain author
  --after="MMM DD YYYY" ex. ("Jun 20 2008") only commits after a certain date
  --before="MMM DD YYYY" only commits that occur before a certain date
  --merge       only the commits involved in the current merge conflicts

git log <ref>..<ref>
  show commits between the specified range. Useful for seeing changes from
  remotes:
  git log HEAD..origin/master # after git remote update

git show <rev>
  show the changeset (diff) of a commit specified by <rev>, which can be any
  SHA1 commit ID, branch name, or tag (shows the last commit (HEAD) by default)

  also to show the contents of a file at a specific revision, use
     git show <rev>:<filename>
  this is similar to cat-file but much simpler syntax.

git show --name-only <rev>
```

```
  show only the names of the files that changed, no diff information.

git blame <file>
  show who authored each line in <file>

git blame <file> <rev>
  show who authored each line in <file> as of <rev> (allows blame to go back in
  time)

git gui blame
  really nice GUI interface to git blame

git whatchanged <file>
  show only the commits which affected <file> listing the most recent first
  E.g. view all changes made to a file on a branch:
    git whatchanged <branch> <file>  | grep commit | \
         colrm 1 7 | xargs -I % git show % <file>
  this could be combined with git remote show <remote> to find all changes on
  all branches to a particular file.

git diff <commit> head path/to/fubar
  show the diff between a file on the current branch and potentially another
  branch

git diff --cached [<file>]
  shows diff for staged (git-add'ed) files (which includes uncommitted git
  cherry-pick'ed files)

git ls-files
  list all files in the index and under version control.

git ls-remote <remote> [HEAD]
  show the current version on the remote repo. This can be used to check whether
  a local is required by comparing the local head revision.

Adding / Deleting
-----------------

git add <file1> <file2> ...
  add <file1>, <file2>, etc... to the project

git add <dir>
  add all files under directory <dir> to the project, including subdirectories

git add .
  add all files under the current directory to the project
  *WARNING*: including untracked files.

git rm <file1> <file2> ...
  remove <file1>, <file2>, etc... from the project

git rm $(git ls-files --deleted)
  remove all deleted files from the project

git rm --cached <file1> <file2> ...
  commits absence of <file1>, <file2>, etc... from the project

Ignoring
---------

Option 1:

Edit $GIT_DIR/info/exclude. See Environment Variables below for explanation on
$GIT_DIR.
```

```
Option 2:

Add a file .gitignore to the root of your project. This file will be checked in.

Either way you need to add patterns to exclude to these files.

Staging
-------

git add <file1> <file2> ...
git stage <file1> <file2> ...
  add changes in <file1>, <file2> ... to the staging area (to be included in
  the next commit

git add -p
git stage --patch
  interactively walk through the current changes (hunks) in the working
  tree, and decide which changes to add to the staging area.

git add -i
git stage --interactive
  interactively add files/changes to the staging area. For a simpler
  mode (no menu), try `git add --patch` (above)

Unstaging
---------

git reset HEAD <file1> <file2> ...
  remove the specified files from the next commit


Committing
----------

git commit <file1> <file2> ... [-m <msg>]
  commit <file1>, <file2>, etc..., optionally using commit message <msg>,
  otherwise opening your editor to let you type a commit message

git commit -a
  commit all files changed since your last commit
  (does not include new (untracked) files)

git commit -v
  commit verbosely, i.e. includes the diff of the contents being committed in
  the commit message screen

git commit --amend
  edit the commit message of the most recent commit

git commit --amend <file1> <file2> ...
  redo previous commit, including changes made to <file1>, <file2>, etc...


Branching
---------

git branch
  list all local branches

git branch -r
  list all remote branches

git branch -a
```

```
          list all local and remote branches

        git branch <branch>
          create a new branch named <branch>, referencing the same point in history as
          the current branch

        git branch <branch> <start-point>
          create a new branch named <branch>, referencing <start-point>, which may be
          specified any way you like, including using a branch name or a tag name

        git push <repo> <start-point>:refs/heads/<branch>
          create a new remote branch named <branch>, referencing <start-point> on the
          remote. Repo is the name of the remote.
          Example: git push origin origin:refs/heads/branch-1
          Example: git push origin origin/branch-1:refs/heads/branch-2
          Example: git push origin branch-1 ## shortcut

        git branch --track <branch> <remote-branch>
          create a tracking branch. Will push/pull changes to/from another repository.
          Example: git branch --track experimental origin/experimental

        git branch --set-upstream <branch> <remote-branch> (As of Git 1.7.0)
          Make an existing branch track a remote branch
          Example: git branch --set-upstream foo origin/foo

        git branch -d <branch>
          delete the branch <branch>; if the branch you are deleting points to a
          commit which is not reachable from the current branch, this command
          will fail with a warning.

        git branch -r -d <remote-branch>
          delete a remote-tracking branch.
          Example: git branch -r -d wycats/master

        git branch -D <branch>
          even if the branch points to a commit not reachable from the current branch,
          you may know that that commit is still reachable from some other branch or
          tag. In that case it is safe to use this command to force git to delete the
          branch.

        git checkout <branch>
          make the current branch <branch>, updating the working directory to reflect
          the version referenced by <branch>

        git checkout -b <new> <start-point>
          create a new branch <new> referencing <start-point>, and check it out.

        git push <repository> :<branch>
          removes a branch from a remote repository.
          Example: git push origin :old_branch_to_be_deleted

        git co <branch> <path to new file>
          Checkout a file from another branch and add it to this branch. File
          will still need to be added to the git branch, but it's present.
          Eg. git co remote_at_origin__tick702_antifraud_blocking
          ..../...nt_elements_for_iframe_blocked_page.rb

        git show <branch> -- <path to file that does not exist>
          Eg. git show remote_tick702 -- path/to/fubar.txt
          show the contents of a file that was created on another branch and that
          does not exist on the current branch.

        git show <rev>:<repo path to file>
          Show the contents of a file at the specific revision. Note: path has to be
```

```
  absolute within the repo.

Merging
-------

git merge <branch>
  merge branch <branch> into the current branch; this command is idempotent
  and can be run as many times as needed to keep the current branch
  up-to-date with changes in <branch>

git merge <branch> --no-commit
  merge branch <branch> into the current branch, but do not autocommit the
  result; allows you to make further tweaks

git merge <branch> -s ours
  merge branch <branch> into the current branch, but drops any changes in
  <branch>, using the current tree as the new tree


Cherry-Picking
--------------

git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] <commit>
  selectively merge a single commit from another local branch
  Example: git cherry-pick 7300a6130d9447e18a931e898b64eefedea19544


Squashing
---------
WARNING: "git rebase" changes history. Be careful. Google it.

git rebase --interactive HEAD~10
  (then change all but the first "pick" to "squash")
  squash the last 10 commits into one big commit


Conflicts
---------

git mergetool
  work through conflicted files by opening them in your mergetool (opendiff,
  kdiff3, etc.) and choosing left/right chunks. The merged result is staged for
  commit.

For binary files or if mergetool won't do, resolve the conflict(s) manually
and then do:

  git add <file1> [<file2> ...]

Once all conflicts are resolved and staged, commit the pending merge with:

  git commit


Sharing
-------

git fetch <remote>
  update the remote-tracking branches for <remote> (defaults to "origin").
  Does not initiate a merge into the current branch (see "git pull" below).

git pull
  fetch changes from the server, and merge them into the current branch.
  Note: .git/config must have a [branch "some_name"] section for the current
  branch, to know which remote-tracking branch to merge into the current
  branch.  Git 1.5.3 and above adds this automatically.

git push
  update the server with your commits across all branches that are *COMMON*
  between your local copy and the server.  Local branches that were never
  pushed to the server in the first place are not shared.

git push origin <branch>
  update the server with your commits made to <branch> since your last push.
  This is always *required* for new branches that you wish to share. After
  the first explicit push, "git push" by itself is sufficient.

git push origin <branch>:refs/heads/<branch>
  E.g. git push origin twitter-experiment:refs/heads/twitter-experiment
  Which, in fact, is the same as git push origin <branch> but a little
  more obvious what is happening.

Reverting
---------

git revert <rev>
  reverse commit specified by <rev> and commit the result.  This does *not* do
  the same thing as similarly named commands in other VCS's such as "svn
  revert" or "bzr revert", see below

git checkout <file>
  re-checkout <file>, overwriting any local changes

git checkout .
  re-checkout all files, overwriting any local changes.  This is most similar
  to "svn revert" if you're used to Subversion commands


Fix mistakes / Undo
-------------------

git reset --hard
  abandon everything since your last commit; this command can be DANGEROUS.
  If merging has resulted in conflicts and you'd like to just forget about
  the merge, this command will do that.

git reset --hard ORIG_HEAD or git reset --hard origin/master
  undo your most recent *successful* merge *and* any changes that occurred
  after.  Useful for forgetting about the merge you just did.  If there are
  conflicts (the merge was not successful), use "git reset --hard" (above)
  instead.

git reset --soft HEAD^
  forgot something in your last commit? That's easy to fix. Undo your last
  commit, but keep the changes in the staging area for editing.

git commit --amend
  redo previous commit, including changes you've staged in the meantime.
  Also used to edit commit message of previous commit.


Plumbing
--------

test <sha1-A> = $(git merge-base <sha1-A> <sha1-B>)
  determine if merging sha1-B into sha1-A is achievable as a fast forward;
  non-zero exit status is false.
```

```
Stashing
--------

git stash
git stash save <optional-name>
  save your local modifications to a new stash (so you can for example
  "git svn rebase" or "git pull")

git stash apply
  restore the changes recorded in the stash on top of the current working tree
  state

git stash pop
  restore the changes from the most recent stash, and remove it from the stack
  of stashed changes

git stash list
  list all current stashes

git stash show <stash-name> -p
  show the contents of a stash - accepts all diff args

git stash drop [<stash-name>]
  delete the stash

git stash clear
  delete all current stashes


Remotes
-------

git remote add <remote> <remote_URL>
  adds a remote repository to your git config.  Can be then fetched locally.
  Example:
    git remote add coreteam git://github.com/wycats/merb-plugins.git
    git fetch coreteam

git push <remote> :refs/heads/<branch>
  delete a branch in a remote repository

git push <remote> <remote>:refs/heads/<remote_branch>
  create a branch on a remote repository
  Example: git push origin origin:refs/heads/new_feature_name

git push <repository> +<remote>:<new_remote>
  replace a <remote> branch with <new_remote>
  think twice before do this
  Example: git push origin +master:my_branch

git remote prune <remote>
  prune deleted remote-tracking branches from "git branch -r" listing

git remote add -t master -m master origin git://example.com/git.git/
  add a remote and track its master

git remote show <remote>
  show information about the remote server.

git checkout -b <local branch> <remote>/<remote branch>
  Eg git checkout -b myfeature origin/myfeature
  Track a remote branch as a local branch.
```

```
git pull <remote> <branch>
git push
  For branches that are remotely tracked (via git push) but
  that complain about non-fast forward commits when doing a
  git push. The pull synchronizes local and remote, and if
  all goes well, the result is pushable.

git fetch <remote>
  Retrieves all branches from the remote repository. After
  this 'git branch --track ...' can be used to track a branch
  from the new remote.

Submodules
----------

git submodule add <remote_repository> <path/to/submodule>
  add the given repository at the given path. The addition will be part of the
  next commit.

git submodule update [--init]
  Update the registered submodules (clone missing submodules, and checkout
  the commit specified by the super-repo). --init is needed the first time.

git submodule foreach <command>
  Executes the given command within each checked out submodule.

Removing submodules

   1. Delete the relevant line from the .gitmodules file.
   2. Delete the relevant section from .git/config.
   3. Run git rm --cached path_to_submodule (no trailing slash).
   4. Commit and delete the now untracked submodule files.

Updating submodules
  To update a submodule to a new commit:
    1. update submodule:
        cd <path to submodule>
        git pull
    2. commit the new version of submodule:
        cd <path to toplevel>
        git commit -m "update submodule version"
    3. check that the submodule has the correct version
        git submodule status
  If the update in the submodule is not committed in the
  main repository, it is lost and doing git submodule
  update will revert to the previous version.

Patches
-------

git format-patch HEAD^
  Generate the last commit as a patch that can be applied on another
  clone (or branch) using 'git am'. Format patch can also generate a
  patch for all commits using 'git format-patch HEAD^ HEAD'
  All page files will be enumerated with a prefix, e.g. 0001 is the
  first patch.

git format-patch <Revision>^..<Revision>
  Generate a patch for a single commit. E.g.
    git format-patch d8efce43099^..d8efce43099
  Revision does not need to be fully specified.

git am <patch file>
  Applies the patch file generated by format-patch.
```

```
git diff --no-prefix > patchfile
  Generates a patch file that can be applied using patch:
    patch -p0 < patchfile
  Useful for sharing changes without generating a git commit.

Tags
----

git tag -l
  Will list all tags defined in the repository.

git co <tag_name>
  Will checkout the code for a particular tag. After this you'll
  probably want to do: 'git co -b <some branch name>' to define
  a branch. Any changes you now make can be committed to that
  branch and later merged.

Archive
-------

git archive master | tar -x -C /somewhere/else
  Will export expanded tree as tar archive at given path

git archive master | bzip2 > source-tree.tar.bz2
  Will export archive as bz2

git archive --format zip --output /full/path master
  Will export as zip

Git Instaweb
------------

git instaweb --httpd=webrick [--start | --stop | --restart]


Environment Variables
---------------------

GIT_AUTHOR_NAME, GIT_COMMITTER_NAME
  Your full name to be recorded in any newly created commits.  Overrides
  user.name in .git/config

GIT_AUTHOR_EMAIL, GIT_COMMITTER_EMAIL
  Your email address to be recorded in any newly created commits.  Overrides
  user.email in .git/config

GIT_DIR
  Location of the repository to use (for out of working directory repositories)

GIT_WORKING_TREE
  Location of the Working Directory - use with GIT_DIR to specifiy the working
  directory root
  or to work without being in the working directory at all.
```