

Crash course in version control (focusing mainly on *git*)

Cheng H. Lee
Lab7 Systems, Inc.

7 March 2014

What is version control?

All source code changes (bug fixes, improvements, etc.)

Need some way of managing changes; one naïve way:

my-script.py

my-script.py.0

my-script.py.1

my-script.py.2

-or-

my-script.py

my-script.py.2013-05-01

my-script.py.2012-12-20

my-script.py.2012-10-31

What is version control?

Many, many problems with the naïve approach:

- Requires needless duplication, clutters up filesystem
- Numbering scheme often delicate, hard to maintain
- Hard to understand history, relationships between versions and files.
- Hard to share and develop with multiple people

Most, if not all, of these problems solved by some sort of **version control system (VCS)**.

What is version control *not*?

**** A VCS is NOT a substitute for actual backups! ****

Can help in recovering code (especially if distributed)...

But, most VCSes deal badly with large and/or binary files.

So, I do *NOT* recommended using a VCS to manage:

- Large collections of binary files (e.g., PDFs)
- Large data files (e.g., genome references)

Basic VCS terminology

Repository: Some place that stores the files, their past versions, and any associated meta data.

Working copy: Version of the repository currently being worked on, where changes to be added back to the repository are first produced.

Diff or **patch:** A description of how a specific file has changed.

Commit: A set of diffs and its associated metadata (e.g., who made the change and when) that describe how the repository has changed from one version to another.

Lots of VCS out there

Centralized: single server storing the repository; all commits must be put onto this server.

E.g.: Subversion, CVS

Distributed: each developer has a copy of the repository; all commits happen "locally" but can be shared.

E.g.: Git, Mercurial

Also: Bzr, ClearCase, SourceSafe, RCS (not really...)

Basic centralized VCS workflow

1. Check out a working copy from VCS server.
2. Make changes in working copy.
3. Test changes to make sure they work.
4. Commit changes back to central server.
5. Repeat steps 2 through 4.

Basic distributed VCS workflow

Very similar...

1. Copy (or clone in git parlance) a repository.
2. Make changes in your local copy.
3. Test changes to make sure they work.
4. Commit changes to your local copy.
5. Repeat steps 2 through 4.

But we have the option of:

6. Sending our changes to someone else's repository, or
7. Pulling in changes from someone else's repository.

Getting started with git

Download and install:

Main page: <http://git-scm.com/downloads>

Windows: [TortoiseGit](#) (integrates with Explorer)

OS X: Use git-scm.com version (X Code version is old)

Debian/Ubuntu: "apt-get install git"

Tell git who you are:

```
$ git config user.name "first last"
```

```
$ git config user.email "me@institute.org"
```

Cloning a git repository

Cloning gets a repository from someone else, including all of its tracked files and their history.

```
# "git clone" will create a new subdirectory  
# underneath your current location  
$ cd $HOME/projects  
$ ls  
project1  project2
```

Cloning a git repository

Cloning gets a repository from someone else, including all of its tracked files and their history.

```
# Usage: "git clone <url>", where <url> is  
# provided by person you're cloning from; e.g.,  
$ git clone git@bitbucket.org:myorg/projectX.git  
Cloning into 'projectX'  
# ... bunch of other status messages ...
```

Cloning a git repository

Cloning gets a repository from someone else, including all of its tracked files and their history.

```
$ ls
```

```
project1 project2 projectX
```

```
$ cd projectX
```

```
$ ls
```

```
# ... contents of the "projectX" repository ...
```

Setting up your own git repository

What if you have a project on your own computer that hasn't been shared with anyone else?

```
$ cd /path/to/my/project
```

```
$ ls -a
```

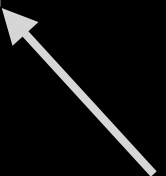
```
file1.txt file2.txt subdir/
```

```
$ git init
```

```
Initialized empty Git repository in /path/to/my/project/.git/
```

```
$ ls -a
```

```
.git/ file1.txt file2.txt subdir/
```



Where the git magic happens;
remove at your own peril

Adding files to version control

Git (and most other VCSes) do *not* automatically put files under version control.

Makes sense: don't want to create a repository and add a whole bunch of useless stuff (temporary files, large files, binary data, etc.) to the repository.

You must explicitly tell git what files you want to track.

Adding files to version control

What's in our project directory?

```
$ ls .  
file1.txt  file2.txt  subdir/  
$ ls subdir/  
file3.txt  ignore-me.txt
```

Adding files to version control

```
# "git status": what's changed in your working directory
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
# file1.txt
# file2.txt
# subdir/
nothing added to commit but untracked files present (use "git
add" to track)
```


Adding files to version control

"git status": what's changed in your working directory

```
$ git status
```

On branch master

```
# Untracked files:
```

(use "git add <file>..." to include in what will be committed)

```
#
```

```
# file1.txt
```

```
# file2.txt
```

```
# subdir/
```

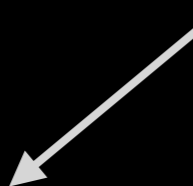
"untracked": files git notices on your filesystem that are not yet under version control

nothing added to commit but untracked files present (use "git add" to track)

Adding files to version control

```
# "git status": what's changed in your working directory
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
# file1.txt
# file2.txt
# subdir/
nothing added to commit but untracked files present (use "git
add" to track)
```

Note that subdirectory contents aren't listed; we'll come back to that in a bit.



Adding files to version control

"git status": what's changed in your working directory

\$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

#


file1.txt

file2.txt

subdir/

nothing added to commit but untracked files present (use "git add" to track)

Git tells you exactly what to do



Adding files to version control

What's in our project directory?

```
$ ls .  
file1.txt  file2.txt  subdir/  
$ ls subdir/  
file3.txt  ignore-me.txt
```

Adding files to version control

What's in our project directory?

```
$ ls .  
file1.txt  file2.txt  subdir/  
$ ls subdir/  
file3.txt  ignore-me.txt
```

Let's say we only want to track file1.txt & file2.txt:

Adding files to version control

What's in our project directory?

```
$ ls .  
file1.txt  file2.txt  subdir/  
$ ls subdir/  
file3.txt  ignore-me.txt
```

Let's say we only want to track file1.txt & file2.txt:

```
$ git add file1.txt  
$ git add file2.txt
```

Adding files to version control

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   file1.txt
#   new file:   file2.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   subdir/
```

Adding files to version control

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   file1.txt
#   new file:   file2.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   subdir/
```

"staged": git has detected changes, but hasn't saved ("committed") them yet.

Adding files to version control

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   file1.txt
# new file:   file2.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/
```

← in this case, two new files

Committing files to version control

```
# "git commit" puts stuff in the repository...  
$ git commit -m "my first commit"  
[master (root-commit) ec4107d] my first commit  
1 files changed, 4 insertions(+), 0 deletions(-)  
create mode 100644 file1.txt  
create mode 100644 file2.txt
```

Committing files to version control

"git commit" puts stuff in the repository...


```
$ git commit -m "my first commit"
```

```
[master (root-commit) ec4107d] my first commit  
1 files changed, 4 insertions(+), 0 deletions(-)  
create mode 100644 file1.txt  
create mode 100644 file2.txt
```

commit message: tells people what you did

Committing files to version control

```
# "git commit" puts stuff in the repository...  
$ git commit -m "my first commit"  
[master (root-commit) ec4107d] my first commit  
1 files changed, 4 insertions(+), 0 deletions(-)  
create mode 100644 file1.txt  
create mode 100644 file2.txt
```

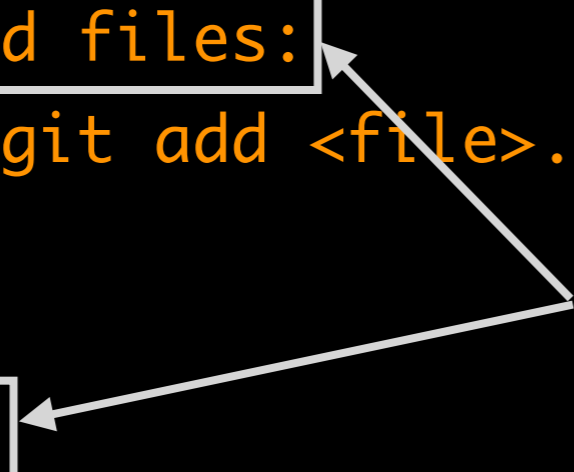


SHA1 checksum: uniquely identifies this commit; the checksum is actually 40-characters long, but we can usually use just the 1st seven characters

What happens after the first commit?

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/
nothing added to commit but untracked files present (use "git
add" to track)
```

Git tells us there's still stuff we aren't tracking.



Dealing with subdirectories

```
$ ls subdir/
```

```
file3.txt  ignore-me.txt
```

```
$ git add subdir ← "git add <subdirectory name>"
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
#   new file:   subdir/file3.txt
```

```
#   new file:   subdir/ignore-me.txt
```

```
#
```

Dealing with subdirectories

```
$ ls subdir/
```

```
file3.txt  ignore-me.txt
```

```
$ git add subdir ← "git add <subdirectory name>"
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# new file:   subdir/file3.txt
```

```
# new file:   subdir/ignore-me.txt
```

```
#
```

↑
adds all the files in the directory;
(might not be the desired behavior)

Dealing with subdirectories

```
$ git add subdir/file3.txt ← "git add <file name>"
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   subdir/file3.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/ignore-me.txt
```

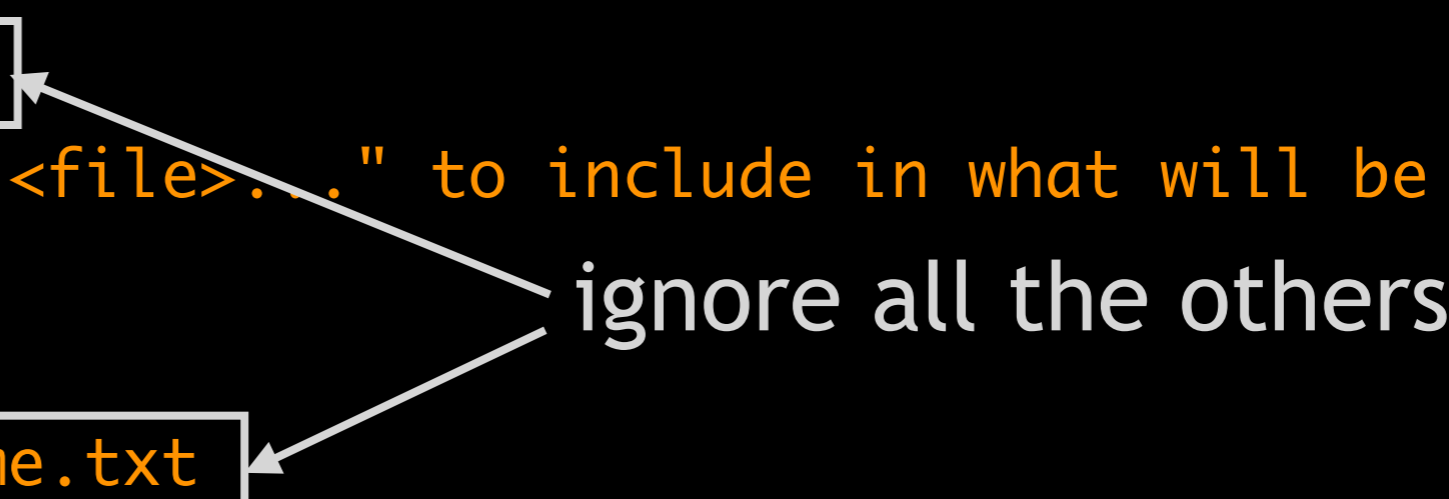

Dealing with subdirectories

```
$ git add subdir/file3.txt ← "git add <file name>"
$ git status
# On branch master
# Changes to be committed: ← add just the file(s) you want
#   (use "git reset HEAD <file>..." to unstage)
#   (don't forget to commit!)
#
# new file:   subdir/file3.txt ←
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#   subdir/ignore-me.txt
```

Dealing with subdirectories

```
$ git add subdir/file3.txt ← "git add <file name>"
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   subdir/file3.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#  subdir/ignore-me.txt
```

ignore all the others



Dealing with subdirectories

As a general rule,

```
$ git <action> <subdirectory>
```

will apply said action to all files in the subdirectory.

When this is *not* what you want, you'll have to apply the action to each file individually:

```
$ git <action> subdir/file_a
```

```
$ git <action> subdir/file_b
```

```
$ .....
```

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the `.gitignore` file to ignore them:

```
$ cd /path/to/my/project
```

```
$ ls -a
```

```
.git/ file1.txt file2.txt subdir/
```



easiest to put it where your repository's ".git" directory is

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the `.gitignore` file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/  file1.txt  file2.txt  subdir/
$ echo "subdir/ignore-me.txt" > .gitignore
$ echo ".*.swp" >> .gitignore
$ echo "*~" >> .gitignore
```

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the `.gitignore` file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/  file1.txt  file2.txt  subdir/
$ echo "subdir/ignore-me.txt" > .gitignore
$ echo "*.swp" >> .gitignore
$ echo "*~" >> .gitignore
```

↑
ignore specific source files

Ignoring certain files

Having files show up as "untracked" all the time can be annoying. Use the `.gitignore` file to ignore them:

```
$ cd /path/to/my/project
$ ls -a
.git/ file1.txt file2.txt subdir/
$ echo "subdir/ignore-me.txt" > .gitignore
$ echo ".*.swp" >> .gitignore
$ echo "*~" >> .gitignore
```

↑
things like editor temp. files

Ignoring certain files

".gitignore" is a regular text file.

You can edit it with any text editor.

```
$ nano .gitignore
# ... add ".gitignore" as a new line to have git
# ignore the ".gitignore" file ...
```

You can add it to version control.

```
# useful for multi-person projects
$ git add .gitignore
$ git commit -m "added a .gitignore file"
... info about the commit ...
```


Adding more files to the repository

```
# Create a new file; hopefully, you're doing  
# something a little more impressive.  
$ echo "hello world" > subdir/file4.txt
```

Adding more files to the repository

```
# Create a new file; hopefully, you're doing  
# something a little more impressive.
```

```
$ echo "hello world" > subdir/file4.txt
```

```
$ git status
```

```
# On branch master
```

```
# Untracked files:
```

```
# (use "git add <file>..." to include in what will be  
committed)
```

```
#
```

```
# subdir/file4.txt
```

```
nothing added to commit but untracked files present (use "git  
add" to track)
```

Adding more files to the repository

Follow the standard approach:

```
$ git add subdir/file4.txt
$ git commit -m "added file4.txt"
[master 1fede62] added file4.txt
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 subdir/file4.txt
```

What's happened to our code?

To get a history of commits to your repository:

```
$ git log
```

```
commit 1fede6267aaa964995f722f8aa5503cd390f946e  
Author: Cheng H. Lee <cheng.lee@lab7.io>  
Date: Thu May 2 19:35:32 2013 -0500
```

```
added file4.txt
```

```
commit 3e36430d2a9d519897e5c6f7e1922a31e3ab4d14  
Author: Cheng H. Lee <cheng.lee@lab7.io>  
Date: Thu May 2 19:21:22 2013 -0500
```

```
added a .gitignore file
```

```
... and so on ...
```

What's happened to our code?

To get a history of commits to your repository:

```
$ git log
```

```
commit 1fede6267aaa964995f722f8aa5503cd390f946e
Author: Cheng H. Lee <chlee@utexas.edu>
Date: Thu May 2 19:35:32 2013 -0500

added file4.txt
```

```
commit 2026120d7c0d51020705c6f701077a3103ab4d14
Author: The most recent commit...
Date: Thu May 2 19:41:22 2013 -0500

added a .gitignore file
```

```
... and so on ...
```

What's happened to our code?

To get a history of commits to your repository:

```
$ git log
```

```
commit 1fede6267aaa964995f722f8aa5503cd390f946e
Author: Cheng H. Lee <chlee@utexas.edu>
Date: Thu May 2 19:35:32 2013 -0500
```

```
added file4.txt
```

```
commit 3e36430d2a9d519897e5c6f7e1922a31e3ab4d14
Author: Cheng H. Lee <chlee@utexas.edu>
Date: Thu May 2 19:21:22 2013 -0500
added a .gitignore file
```

... and so on ...



...and the one before that

What's happened to our code?

"git log" has lots of options:

```
$ git log -5          # only the last 5 commits  
... as before, but we'll only get 5 messages ...
```

```
$ git log --oneline   # abbreviated log  
1fede62 added file4.txt  
3e36430 added a .gitignore file  
3212151 added file3.txt  
ec4107d my first commit
```

```
$ git log -- file1.txt # show commits involving file1.txt
```

```
$ git help log        # bring up help page for more options
```

Committing edits to the repository

Let's say I've just finished editing "file1.txt".

```
$ git status
On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#   modified:   file1.txt
#
no changes added to commit (use "git add" and/or "git commit -
a")
```

← git has detected that the file has changed.

Committing edits to the repository

To figure out what has changed:

```
$ git diff
diff --git a/file1.txt b/file1.txt
index 939f749..3e15a88 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,5 @@
   this is line 1
   this is line 2
+this is a line I added
   this is line 3
-this is line 4
+this is the last line
```

Committing edits to the repository

To figure out what has changed:

```
$ git diff
```

```
diff --git a/file1.txt b/file1.txt
```

```
index 939f749..3e15a88 100644
```

```
--- a/file1.txt ← old version of the file
```

```
+++ b/file1.txt
```

```
@@ -1,4 +1,5 @@
```

```
  this is line 1
```

```
  this is line 2
```

```
+this is a line I added
```

```
  this is line 3
```

```
-this is line 4 ← line that was deleted
```

```
+this is the last line
```

Committing edits to the repository

To figure out what has changed:

```
$ git diff
```

```
diff --git a/file1.txt b/file1.txt
```

```
index 939f749..3e15a88 100644
```

```
--- a/file1.txt
```

```
+++ b/file1.txt
```

← new version of the file

```
@@ -1,4 +1,5 @@
```

```
  this is line 1
```

```
  this is line 2
```

```
+this is a line I added
```

```
  this is line 3
```

```
-this is line 4
```

```
+this is the last line
```

← lines that were added

Committing edits to the repository

VCSes don't record changes until you **commit**.

Unlike other VCSes, git "requires" a two-step commit:

```
$ git add file1.txt      # "stages" file1
$ git commit -m "edits made to file1"
[master 51cb5a3] edits made to file1
1 files changed, 2 insertions(+), 1 deletions(-)
```

If you forget to **stage** a file with "git add", "git commit" won't actually commit its changes into the repository.

Committing edits to the repository

There is a short-cut for the lazy. Suppose:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#       modified:   file2.txt
#       modified:   subdir/file3.txt
#
no changes added to commit (use "git add" and/or "git commit -
a")
```

Committing edits to the repository

The "long" way of committing both files:

```
$ git add file2.txt subdir/file3.txt  
$ git commit -m "Changes to file2 and file3"  
[master 0724984] changes to file2 and file3  
2 files changed, 5 insertions(+), 0 deletions(-)
```

Committing edits to the repository

The "short" way of committing both files:

```
$ git commit -a -m "Changes to file2 and file3"  
[master 0724984] changes to file2 and file3  
2 files changed, 5 insertions(+), 0 deletions(-)
```

"**git commit -a**": "stage all tracked files that have been modified and then commit them".

This mimics the "commit" behavior of other VCSes.

Committing edits to the repository

Caveat: "git commit -a" does *not* automatically add untracked files to the commit. If you create a new file, you must explicitly use "git add" to commit it.

E.g., say you modified "file2.txt" and "file3.txt" and added a new file called "useful-code.py". To commit all three, you *must* run the following:

```
$ git add useful-code.py
$ git commit -a -m "my commit message"
[master 4f9a57f] my commit message
2 files changed, 5 insertions(+), 0 deletions(-)
create mode 100644 useful-code.py
```


Quick summary

`git clone <url>`: Copy a repository from someone else.

`git init`: Set up a new git repository in this directory.

`git add <new_file>`: start tracking <new_file>; also stages it so it's added to the repository in the next commit

`git status`: show changes in the working directory relative to the last commit.

`git diff`: show changes between the current (unstaged) contents of tracked files and the corresponding contents in the last commit.

`git commit -a -m <msg>`: commit all modifications to tracked files (and any new files that were added) to the repository, using "<msg>" as the commit log message.

`git log -<N>`: show the commit log messages for the last <N> commits.

Removing files

Occasionally useful to remove files from your working copy; e.g., old code that conflicts with your new code:

```
$ ls
```

```
file1.txt file2.txt old-script.py subdir/
```

```
$ git rm old-script.py
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#  
# deleted:    old-script.py
```

staged but doesn't take effect until commit.

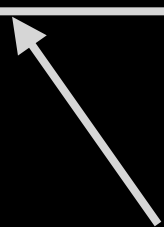
Removing files

Occasionally useful to remove files from your working copy; e.g., old code that conflicts with your new code:

```
$ git commit -m "removed obsolete script"  
[master 9458cbb] removed obsolete script  
1 files changed, 0 insertions(+), 4 deletions(-)  
delete mode 100644 old-script.py
```

```
$ ls
```

```
file1.txt  file2.txt  subdir/
```



"old-script.py" no longer exists in the directory.

Moving or renaming files

Often need to move or rename files:

```
$ git mv file2.txt subdir/new-name.txt
# As with "git rm", this stages but does not commit the file.
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    file2.txt -> subdir/new-name.txt
#
$ git commit -m "renamed file2.txt to subdir/new-name.txt"
$ ls subdir
new-name.txt
```

Dead but not forgotten

Why use a VCS?

Once something is in the repository, it is never lost*.

Among other things, we can:

- Save ourselves from a common type of trouble.
- Compare any two previous (committed) versions.
- Backing out from recent changes.
- Even bring back a file from the dead.

* Well, unless the entire repository itself (i.e., the ".git" directory) is lost.**

** Or with git, you do something really bad like rebase a public branch, then run gc.

Saving yourself from trouble

Commonly, trigger happiness with "rm":

```
$ ... do some work ...
```

```
$ ls
```

```
file1.txt  file2.txt  file_a.txt  file_b.txt  subdir/
```

```
# "file_a.txt" and "file_b.txt" were generated as temporary
```

```
# files while I was doing work; don't need them any more...
```

```
$ rm -f file*
```

```
# OOPS!
```

```
$ ls
```

```
subdir/
```

Saving yourself from trouble

After deleting files:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be
committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#   deleted:    file1.txt
#   deleted:    file2.txt
#
no changes added to commit (use "git add" and/or "git commit -
a")
```

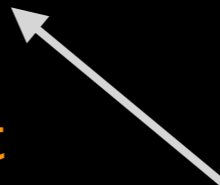
← Important files I cared about; but remember, what was not committed can't be saved.

Saving yourself from trouble

After deleting files:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be
committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#       deleted:    file1.txt
#       deleted:    file2.txt
#
no changes added to commit (use "git add" and/or "git commit -
a")
```

Follow the instructions to "recover"



Saving yourself from trouble

Recovering files from the repository:

```
$ git checkout -- file1.txt file2.txt  
$ ls  
file1.txt  file2.txt  subdir/
```

Important caveat: "git checkout" can only recover the files up to the last commit.

Any changes made after that will be permanently lost; i.e., this is not a foolproof/miraculous way of saving yourself from "rm".

Looking at/comparing to previous commits

Two main tools to look at old versions (commits):

- git log: fetch the previous commit logs and metadata
- git diff: generate a diff between two commits

"git log" general command format

```
$ git log <options> <since commit>..<until commit> -- <files>
```

"git diff" general command format

```
$ git diff <options> <since commit> -- <files>
```

git has multiple ways of referring commits; the "--" is a way of saying everything after this is the name of a file, not the name of a commit

How git refers to commits

Two more common ways:

<SHA1 checksum>: Absolute & unambiguous way

<commit>~<N>: <N>th-generation ancestor of commit

But there are many other ways; see "git help revisions".

"HEAD": Special name referring to the last commit*

"git status": compare current state to HEAD

"HEAD~5": 6 commits ago

* "last commit from where you are now, which might not be the latest commit."

How git refers to commits

```
$ git log --oneline
```

```
# working directory (with possible modifications) is here
```

```
9458cbb removed obsolete script
```

← HEAD

```
71875bd added less than useful python script
```

```
4f9a57f my commit message
```

```
51cb5a3 edits made to file1
```

```
1fede62 added file4.txt
```

```
3e36430 added a .gitignore file
```

```
3212151 added file3.txt
```

```
ec4107d my first commit
```

How git refers to commits

```
$ git log --oneline
```

```
# working directory (with possible modifications) is here
```

```
9458cbb removed obsolete script
```

```
← HEAD
```

```
71875bd added less than useful python script
```

```
← HEAD~1
```

```
4f9a57f my commit message
```

```
51cb5a3 edits made to file1
```

```
1fede62 added file4.txt
```

```
3e36430 added a .gitignore file
```

```
3212151 added file3.txt
```

```
ec4107d my first commit
```

How git refers to commits

```
$ git log --oneline
```

```
# working directory (with possible modifications) is here
```

```
9458cbb removed obsolete script
```

```
← HEAD
```

```
71875bd added less than useful python script
```

```
4f9a57f my commit message
```

```
← HEAD~2
```

```
51cb5a3 edits made to file1
```

```
1fede62 added file4.txt
```

```
3e36430 added a .gitignore file
```

```
3212151 added file3.txt
```

```
ec4107d my first commit
```

How git refers to commits

```
$ git log --oneline
```

```
# working directory (with possible modifications) is here
```

```
9458cbb removed obsolete script
```

```
← HEAD
```

```
71875bd added less than useful python script
```

```
4f9a57f my commit message
```

```
51cb5a3 edits made to file1
```

```
1fede62 added file4.txt
```

```
3e36430 added a .gitignore file
```

```
← HEAD~5
```

```
3212151 added file3.txt
```

```
ec4107d my first commit
```

How git refers to commits

What's changed in the repository since 4 commits ago?

```
# "git log" is not inclusive of the <since> commit.  
# Also, if we leave off a commit reference, git assumes  
# "HEAD"; so, these two are the same command:
```

```
$ git log --oneline HEAD~3..HEAD
```

```
$ git log --oneline HEAD~3..
```

```
9458cbb removed obsolete script
```

← HEAD

```
71875bd added less than useful python script
```

```
4f9a57f my commit message
```

```
51cb5a3 edits made to file1
```

```
1fede62 added file4.txt
```

```
3e36430 added a .gitignore file
```

```
3212151 added file3.txt
```

```
ec4107d my first commit
```

← Not shown

How git refers to commits

Relative references (~<N>) are for commits, not files.

```
$ git log --oneline -- file1.txt
```

```
51cb5a3 edits made to file1 ← HEAD~3
```

```
ec4107d my first commit ← HEAD~7
```

What's changed in file1.txt in the last 2 commits?

```
$ git log --oneline HEAD~2..
```

```
9458cbb removed obsolete script
```

```
71875bd added less than useful python script
```

```
$ git log --oneline HEAD~2.. -- file1.txt
```

```
$ ← No output since  
nothing changed  
in file1.txt
```

Quick word about commit logs

So far, we've been using "`commit -m 'one line message'`" to generate our commit logs.

Better practice for commits is:

```
$ git commit -a
```

... Brings up a text editor for you to enter a log message ...

This allows you to provide more informative messages.

Six months from now, you'll appreciate it.

Quick word about commit logs

De-facto community standard for log message.

First line: short description of what was changed (<50 chars)

--- empty second line ---

Multiple lines providing more details about what was changed (e.g., what algorithm was implemented), and more importantly, why it was changed.

Often wrapped to 72 characters per line.

Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
```

```
commit b09eee938ce52b35026972b76897086c992145a2
```

```
Author: Cheng H. Lee <cheng.lee@lab7.io>
```

```
Date: Mon Apr 29 13:22:32 2013 -0500
```

```
CORE-258 mutation detection for JSONHstore by default
```

```
Made SQLAlchemy mutation detection and notification  
the default behavior for JSONHstore; fixed problems we've  
had with multiple JSON-encoding passes by using the prefix  
tagging trick used with JSONArray (commit 7728c56).
```


Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
```

```
commit b09eee938ce52b35026972b76897086c992145a2
```

```
Author: Cheng H. Lee <cheng.lee@lab7.io>
```

```
Date: Mon Apr 29 13:22:32 2013 -0500
```

```
CORE-258 mutation detection for JSONHstore by default
```



Short description: bug id, what was fixed

What shows up when we do "git log --oneline"

tagging trick used with JSONArray (commit 7728c56).

've
efix

Quick word about commit logs

Example from one of my projects:

```
$ git log -1 b09eee9
```

```
commit b09eee938ce52b35026972b76897086c992145a2
```

```
Author: Cheng H. Lee <cheng.lee@lab7.io>
```

Gory details: why we fixed it, the algorithm/hack I used, and where I got such a terrible idea.

```
CORE-230 mutation detection for JSONStore by default
```



Made SQLAlchemy mutation detection and notification the default behavior for JSONHstore; fixed problems we've had with multiple JSON-encoding passes by using the prefix tagging trick used with JSONArray (commit 7728c56).

Comparing to older versions

What have I changed since the last commit?

```
$ echo "this is the new last line" >>file1.txt
```

```
# git diff compares your edited version with some commit
```

```
# Implicitly, this is HEAD. So, these are equivalent:
```

```
$ git diff -- file1.txt
```

```
$ git diff HEAD -- file1.txt
```

```
diff --git a/file1.txt b/file1.txt
```

```
index 3721789..e77d501 100644
```

```
--- a/file1.txt
```

```
+++ b/file1.txt
```

```
@@ -3,3 +3,5 @@
```

```
... rest of diff output ...
```


Comparing to older versions

Can also get a single diff against any previous version

```
$ git log --oneline -- file1.txt  
51cb5a3 edits made to file1  
ec4107d my first commit
```

Comparing to older versions

Can also get a single diff against any previous version

```
$ git diff 51cb5a3 -- file1.txt
diff --git a/file1.txt b/file1.txt
index 3721789..06b3d59 100644
--- a/file1.txt
+++ b/file1.txt
@@ -3,3 +3,4 @@ this is line 2
    this is a line I added
    this is line 3
    this is the last line
+this is the new last line
```

Comparing to older versions

Can also get a single diff against any previous version

```
$ git diff ec4107d -- file1.txt  
diff --git a/file1.txt b/file1.txt  
index 939f749..06b3d59 100644
```

```
--- a/file1.txt
```

```
+++ b/file1.txt
```

```
@@ -1,4 +1,6 @@
```

```
  this is line 1
```

```
  this is line 2
```

```
+this is a line I added
```

```
  this is line 3
```

```
-this is line 4
```

```
+this is the last line
```

```
+this is the new last line
```

changes from
ec4107d to 51cb5a3

added since 51cb5a3

Bringing back an old version

Suppose you realize the old version of a file was better:

```
$ git log --oneline -- file1.txt
```

```
51cb5a3 edits made to file1
```

```
ec4107d my first commit
```

```
$ git checkout ec4107d -- file1.txt
```

```
$ cat file1.txt
```

```
# ... should see the contents of ec4107d here ...
```

Warning: This will silently and irrevocably destroy any changes you've made to "file1.txt" since its last commit!

Bringing back an old version

Checkout only stages the file:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   file1.txt
#
$ git commit -m "restored original version of file1"
```

Old version won't be fully restored in the repository until the actual commit.

"checkout" to undelete a file

Implicit HEAD is why the "undelete" trick works:

```
# Trick we used to restore a deleted file from the repo...
```

```
$ git checkout -- oops-deleted-file.txt
```

```
# Equivalent to this...
```

```
$ git checkout HEAD -- oops-deleted-file.txt
```

Unlike previous examples, git doesn't stage the file since there've been no changes since the last commit (HEAD).

"checkout" to undelete a file

Can use checkout to restore a file deleted by "git rm":

```
# Use "git log" to find the commit that deleted the file
#   "--diff-filter=D": look for commits that deleted a file
#   "-1": show only the last relevant commit
$ git log --diff-filter=D -1 --oneline -- old-script.py
9458cbb removed obsolete script

# Need to go back one commit (~1) so the file exists...
$ git checkout 9458cbb~1 -- old-script.py

$ git commit -m "restored my old python script"
```

Be careful with checkout!

Make sure you supply "-- <filename>"; without it:

```
$ git checkout ec4107d
```

```
... Warning about 'detached HEAD' state ...
```

Rolls your working directory & all files back to their state in the specified commit (probably not what you want).

To get out of this situation:

```
$ git checkout --force master
```


Getting the contents of an old version

Sometimes, we just want to see the contents of an old version of a file (without restoring in the repository):

```
# Dump the contents to the terminal
```

```
$ git show <commit>:my-old-file.txt
```

```
# Dump the contents to a file named "new-file.txt"
```

```
$ git show <commit>:my-old-file.txt > new-file.txt
```

```
# <commit> can be any valid commit reference; e.g.,
```

```
$ git show HEAD~1:file1.txt # relative to last commit
```

```
$ git show 51cb5a3:file1.txt # absolute commit identifier
```

Another quick summary

`git rm <file>`: "Remove" file from the working directory and the repository; must use "git commit" for this to have a lasting effect.

`git mv <old_name> <new_name>`: Rename and/or move file identified by <old_name> to <new_name>; must use "git commit" for this to have a lasting effect.

`git log <commit_id>.. -- <file>`: Show commit log messages for <file> starting from <commit_id> up to the last commit (HEAD)

`git diff <commit_id> -- <file>`: Show the differences between the version of <file> from <commit_id> and the version currently in the working directory

`git checkout <commit_id> -- <file>`: Replace the contents of <file> with the contents from <commit_id>; must use "git commit" for this to have a lasting effect.

`git show <commit_id>:<file>`: Print the contents of <file> from version <commit_id>; leaves working directory version of <file> untouched.

Things not covered

This should be enough to get you started...

But git (& most VCSes) have a ton of other useful features:

- **Tagging**: labeling certain commits (e.g., "v1.0")
- **Branching & merging**: manage parallel development tracks
- **Remotes**: dealing with other people's repositories
- **Bug finding**: bisect and blame
- **Rebasing**: rewriting history (use with extreme caution)

Also, not covered is working with large open-source projects:

- **Hosting services**: GitHub, BitBucket, Google Code, etc.
- **Forks, pull requests**: how your changes are integrated back

Odds and ends

Getting help:

- `git help <command>`
- Git Book: <http://git-scm.com/book>
- StackOverflow

Visual tools (useful for managing commits and history browsing):

- Windows: TortoiseGit has tools built in
- OSX, Windows: SourceTree (<http://sourcetreeapp.com/>)
- Linux: gitk (pretty ugly tool though...)

