CS 520

Theory and Practice of Software Engineering
Fall 2019

Version Control

September 24, 2019

Working in Teams

TEAMWORK
Large ambitious goals usually require that people work together.

Thursday (September 26)

- First in-class exercise
- On using git (today is a prelude with useful info)
- Form 4-person teams
  - Use moodle to self-select a team; can do it before Thursday or on Thursday
- At least one person per team needs to bring a laptop

BRING A LAPTOP!

Our Goal

- Learn about different kinds of VCS
- Overview the basics of git
- Touch some intermediate git topics
- Clear up common points of confusion
  - Branch vs. Fork?
  - Merge vs. Pull Request?
  - Pull vs. Fetch?
  - Fork vs. Clone?

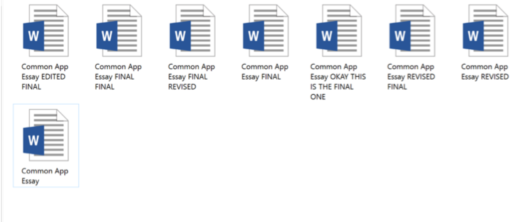
What Is VCS?

What are Version Control Systems

A *Version Control System* (VCS) records changes to a file set over time, making it easy to review or revert to specific versions later

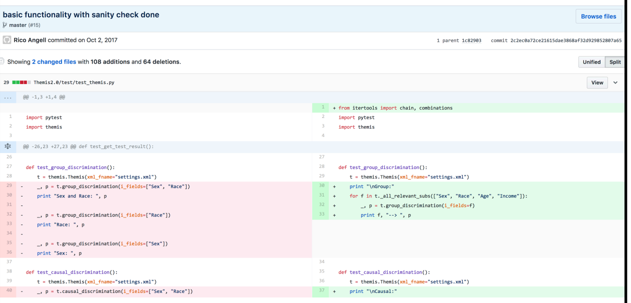
Why Use VCS?

Why Use Version Control?


Common App Essay EDITED FINAL
Common App Essay FINAL FINAL
Common App Essay FINAL REVISED
Common App Essay FINAL
Common App Essay OKAY THIS IS THE FINAL ONE
Common App Essay REVISED FINAL
Common App Essay REVISED
Common App Essay

Why Use Version Control?

- Easy to revert to previous versions
- Work on multiple features in parallel
- Makes collaboration easier
- Narrate the evolution of codebase with messages
- Nice tools such as GitHub (and GitLab (and BitBucket...)) with advanced features such as pipelines, issue tracking, wikis, etc...
- Can store a *backup* remotely and automatically - easy to keep this up to date!
- Helps keep your working space clean



basic functionality with sanity check done

master (#15)

Rico Angell committed on Oct 2, 2017

1 parent 1c82903 commit 2c2ec0a72ce21615dae3868af32d929852807a65

Showing 2 changed files with **108 additions** and **64 deletions.**

29 Themis2.0/test/test_themis.py

```
@@ -1,3 +1,4 @@
 import pytest
 import themis

@@ -26,23 +27,23 @@ def test_get_test_result():

 def test_group_discrimination():
     t = themis.Themis(xml_fname="settings.xml")
-    _, p = t.group_discrimination(i_fields=["Sex", "Race"])
-    print "Sex and Race: ", p
-
-    _, p = t.group_discrimination(i_fields=["Race"])
-    print "Race: ", p
-
-    _, p = t.group_discrimination(i_fields=["Sex"])
-    print "Sex: ", p

 def test_causal_discrimination():
     t = themis.Themis(xml_fname="settings.xml")
-    _, p = t.causal_discrimination(i_fields=["Sex", "Race"])
```

```
+ from itertools import chain, combinations
 import pytest
 import themis


 def test_group_discrimination():
     t = themis.Themis(xml_fname="settings.xml")
+    print "\nGroup:"
+    for f in t._all_relevant_subs(["Sex", "Race", "Age", "Income"]):
+        _, p = t.group_discrimination(i_fields=f)
+        print f, "--> ", p

 def test_causal_discrimination():
     t = themis.Themis(xml_fname="settings.xml")
+    print "\nCausal:"
```

Who Uses VCS?

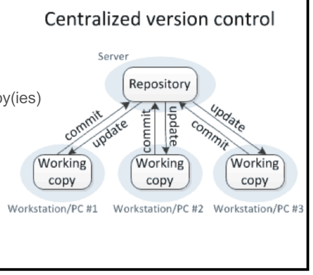
Who Uses Version Control?

- Programmers
- Applications (Microsoft Word, Google Docs, ...)
- Organizations
  - VCS can be used to sync data, not just code

Types of VCS

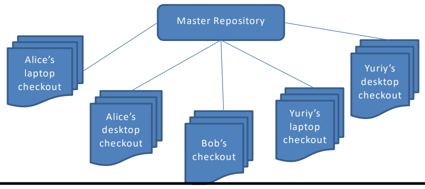
Types of VCS -- Centralized

- There exists a single "central" copy of the project
  - All developers commit to this single copy
- Each developer has local working copy(ies)
  - As soon as they commit, the central repo reflects the changes

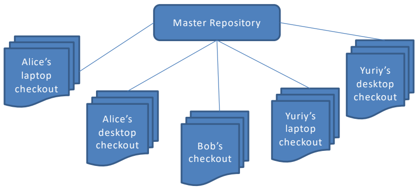


Centralized version control

- (old model)
- Examples: Concurrent Versions System (CVS)
  Subversion (SVN)



Doing work



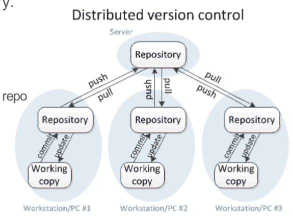
- I update my checkout (working copy)
- I edit
- I update my checkout again
- I merge changes if necessary
- I commit my changes to the Master

Problems with centralized VC

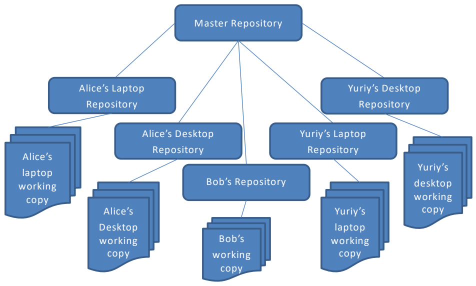
- What if I don't have a network connection?
- What if I am implementing a big change?
- What if I want to explore project history later?

Types of VCS -- Distributed

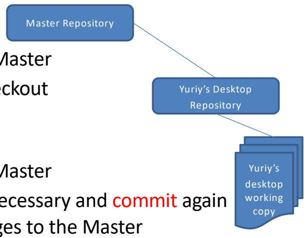
- Each developer has their own repository.
  - Created by the developer, or
  - Cloned from an existing (*remote*) repository
- Developers work on their own repos
  - They can commit, branch, etc.
  - Activity is local unless it is *pushed* to remote repo
  - Remote activity is not seen until dev *fetches* from the remote repo
- Examples: Mercurial (hg), git



Distributed version control model


Master Repository
Alice's Laptop Repository
Yuriy's Desktop Repository
Alice's Desktop Repository
Yuriy's Laptop Repository
Bob's Repository
Alice's laptop working copy
Alice's Desktop working copy
Bob's working copy
Yuriy's laptop working copy
Yuriy's desktop working copy

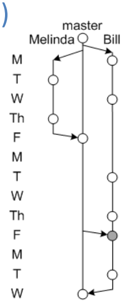
Doing work



- I pull from the Master
- I update my checkout
- I edit
- I commit
- I pull from the Master
- I merge tips if necessary and commit again
- I push my changes to the Master

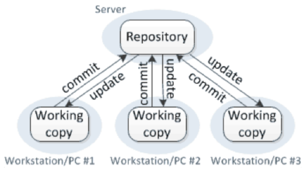
History view (log)

- Bill and Melinda work at the same time
- At the end, all repositories have the same, rich history

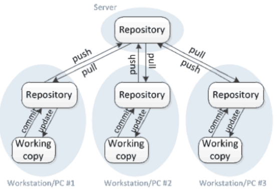


Pros and Cons of Centralized VCS

Centralized version control



Distributed version control



A Motivating Example: What is this git command?

```
NAME
       git-______ - ______ file contents to the index
SYNOPSIS
       git ______ [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
DESCRIPTION
This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically ______s the
current content of existing paths as a whole, but with some options it can also
be used to ______ content with only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.
```

A Motivating Example: What is this git command?

```
NAME
       git-add - Adds file contents to the index
SYNOPSIS
       git add [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
DESCRIPTION
This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically adds the
current content of existing paths as a whole, but with some options it can also
be used to add content with only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.
```

A Motivating Example: What is this git command?

```
NAME
       git-______ - Switch branches or restore working tree files

SYNOPSIS
       git ______ [-q] [-f] [-m] [<branch>]

DESCRIPTION
Updates files in the working tree to match the version in the index or the
specified tree. If no paths are given, git ______ will also update HEAD to set
the specified branch as the current branch.
```

A Motivating Example: What is this git command?

```
NAME
       git-checkout - Switch branches or restore working tree files

SYNOPSIS
       git checkout [-q] [-f] [-m] [<branch>]

DESCRIPTION
Updates files in the working tree to match the version in the index or the
specified tree. If no paths are given, git checkout will also update HEAD to set
the specified branch as the current branch.
```

A Motivating Example: What is this git command?

```
NAME
       git-______ - Forward-port local commits to the updated upstream head
SYNOPSIS
       git ______ [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
               [<upstream> [<branch>]]
       git ______ [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
               --root [<branch>]
       git ______ --continue | --skip | --abort | --edit-todo
DESCRIPTION
If <branch> is specified, git ______ will perform an automatic git checkout
<branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote
and branch.<name>.merge options will be used (see git-config[1] for details) and
the --fork-point option is assumed. If you are currently not on any branch or if
the current branch does not have a configured upstream, the ______ will abort.
```

A Motivating Example: What is this git command?

```
NAME
       git-rebase - Forward-port local commits to the updated upstream head
SYNOPSIS
       git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
               [<upstream> [<branch>]]
       git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
               --root [<branch>]
       git rebase --continue | --skip | --abort | --edit-todo
DESCRIPTION
If <branch> is specified, git rebase will perform an automatic git checkout
<branch> before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote
and branch.<name>.merge options will be used (see git-config[1] for details) and
the --fork-point option is assumed. If you are currently not on any branch or if
the current branch does not have a configured upstream, the rebase will abort.
```

Our goal with git

Be able to understand the git man-pages

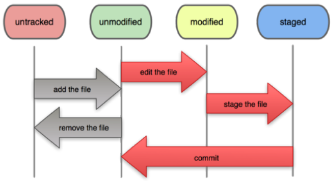
Git Basics

How Git Works

Git Basics -- Tracked vs. Untracked

- **untracked file** - a file not currently under version control
- **tracked file** - a file that *is* under version control

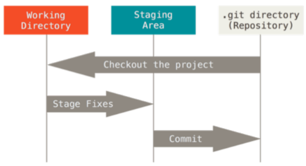
File Status Lifecycle



Git Basics -- Three Main Stages

1. **Committed:** Everything in the file is currently in the database
2. **Modified:** Changed the file but have not committed to the database
3. **Staged:** Marked the file for addition to the database in the next commit

Note that all of the above pertain to *tracked* files.



Git Basics -- Creating Repositories

Initializing a repository

- `git init` - Create an empty git repository or reinitialize an existing one
  - `--bare` - create a bare repository
  - `[directory]` - git init is run inside the provided directory
- `git init` creates a `.git` folder in the directory chosen

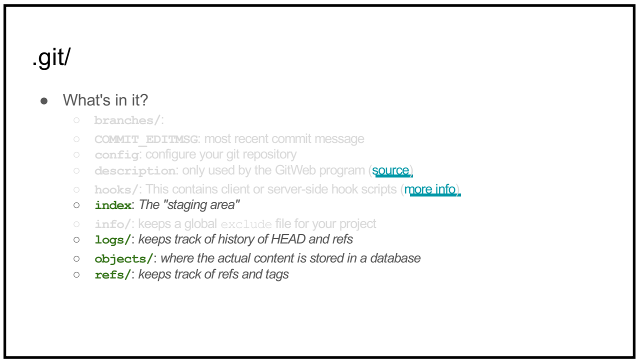
Git Basics -- Creating Repositories

Cloning a Repository

- `git clone` - Clone a repository into a new directory
  - `--depth <depth>` - Create a shallow clone with a history truncated to `<depth>` commits
  - `--branch <name>` - Point local `HEAD` to specific branch (more on `HEAD` in a bit!)
  - `--origin <name>` - Use `<name>` to keep track of remote repo instead of `'origin'`
- Basically, clone just:
  - calls `init`
  - points some meta variables at an existing repository
  - copies the data to the new repo

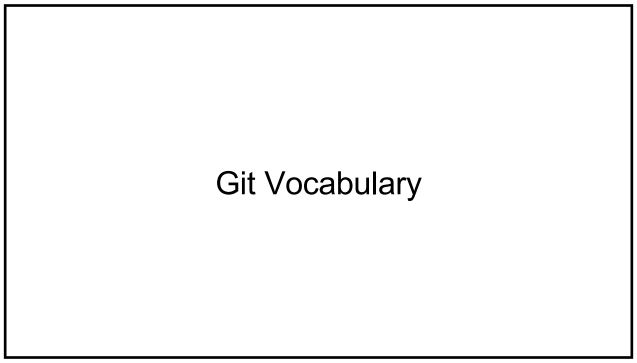
.git/

- What's in it?
  - **`branches/`**:
  - **`COMMIT_EDITMSG`**: most recent commit message
  - **`config`**: configure your git repository
  - **`description`**: only used by the GitWeb program (source)
  - **`hooks/`**: This contains client or server-side hook scripts (more info)
  - **`index`**: The "staging area"
  - **`info/`**: keeps a global `exclude` file for your project
  - **`logs/`**: keeps track of history of HEAD and refs
  - **`objects/`**: where the actual content is stored
  - **`refs/`**: keeps track of refs and tags

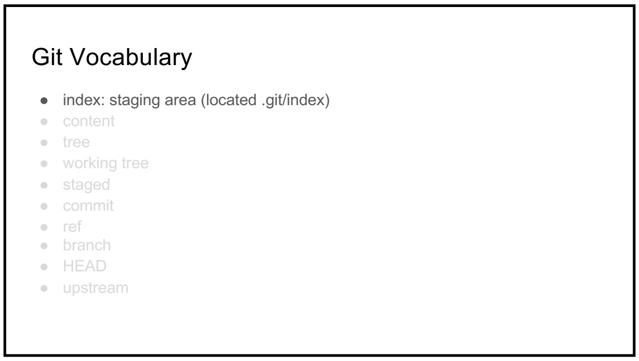


.git/

- What's in it?
  - `branches/`:
  - `COMMIT_EDITMSG`: most recent commit message
  - `config`: configure your git repository
  - `description`: only used by the GitWeb program (source)
  - `hooks/`: This contains client or server-side hook scripts (more info)
  - **`index`**: *The "staging area"*
  - `info/`: keeps a global `exclude` file for your project
  - **`logs/`**: *keeps track of history of HEAD and refs*
  - **`objects/`**: *where the actual content is stored in a database*
  - **`refs/`**: *keeps track of refs and tags*

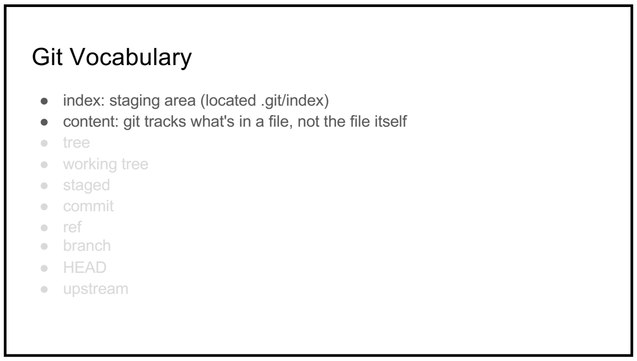


Git Vocabulary



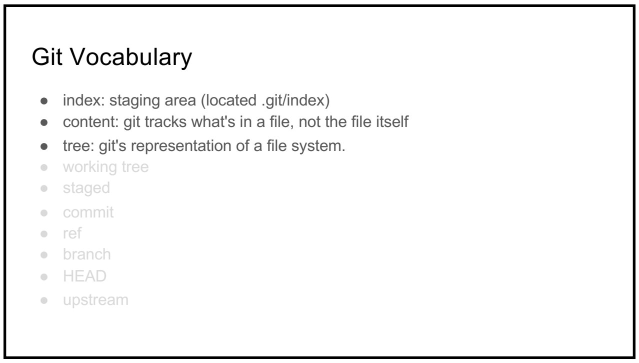
Git Vocabulary

- index: staging area (located .git/index)
- content
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream



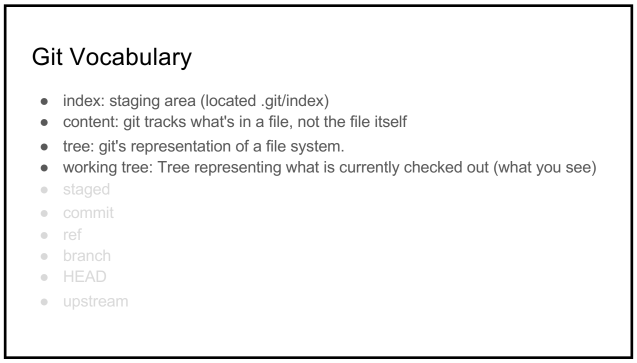
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream



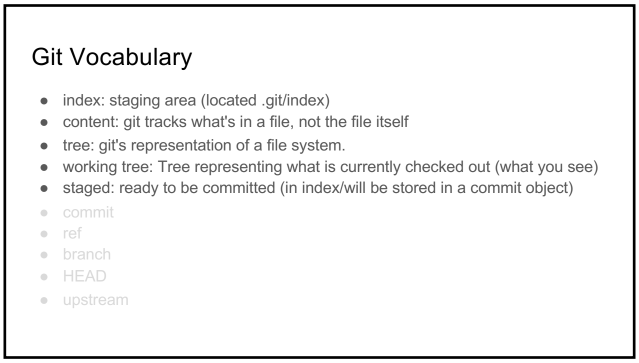
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream



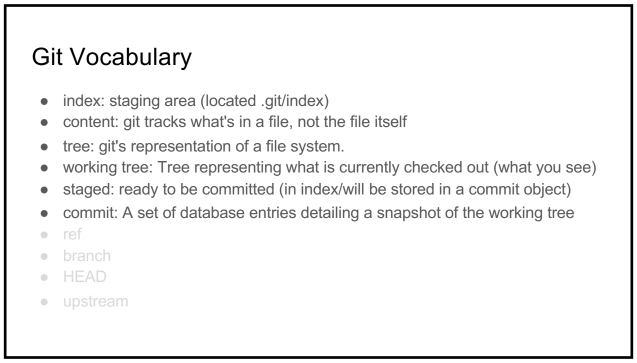
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged
- commit
- ref
- branch
- HEAD
- upstream



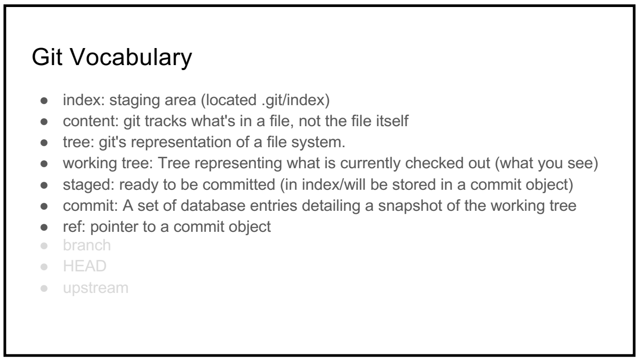
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit
- ref
- branch
- HEAD
- upstream



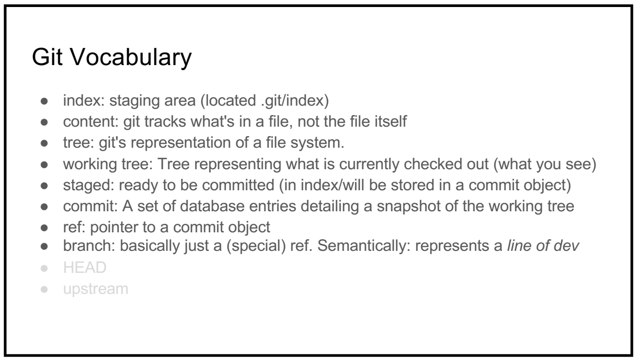
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref
- branch
- HEAD
- upstream



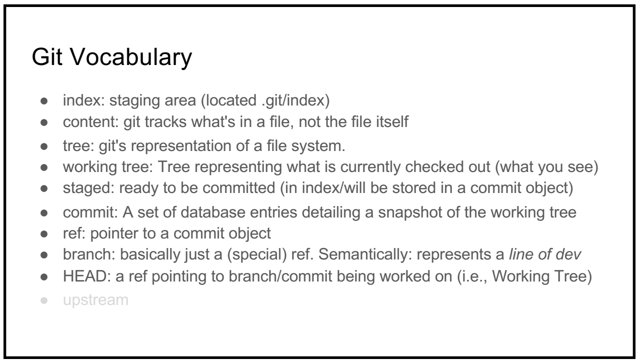
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch
- HEAD
- upstream



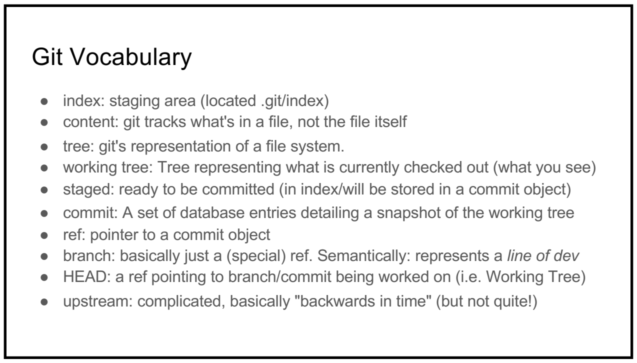
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a *line of dev*
- HEAD
- upstream



Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a *line of dev*
- HEAD: a ref pointing to branch/commit being worked on (i.e., Working Tree)
- upstream



Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a *line of dev*
- HEAD: a ref pointing to branch/commit being worked on (i.e. Working Tree)
- upstream: complicated, basically "backwards in time" (but not quite!)

Git Basics

Working Locally

Git Basics: *Changing Content* -- `git add`

`git add` does *two things*:

1. given an untracked file it will
   a. start tracking it
   b. update /.git/index using the current *content* found in the working tree to prep the content for the next commit (*i.e., the content is **staged***)
2. given a modified unstaged file it will
   a. stage its *contents* for commit

`--patch, -p`: start an interactive staging session that lets you choose portions of a file to add to the next commit.

Git Basics: *Changing Content* -- `git commit`

`git commit` updates the Git database with staged content in `/.git/index`

- Note that staged files can have *unstaged changes*
- By default this will open an editor for you to enter a commit message

`--message=<msg>, -m <msg>`: Add `<msg>` as the commit message. If multiple messages are given, concatenate as separate paragraphs
`--patch, -p`: Use the interactive patch selection interface to choose which changes to commit (similar to `git add -p`)

Git Basics

Making Queries

Git Basics: *Making Queries* -- `git status`

`git status` shows the working tree status. This command displays:

- paths that have differences between the index file and the current HEAD
- paths that have differences between the working tree and the index file
- paths in the working tree that are not tracked by git

`--short, -s`: Give the output in the short-format
`--ignored`: Show ignored files

Git Basics: *Making Queries* -- `git log`

`git log` inspects commit history with multiple display options

- git log is basically a wrapper around `git rev-list` and `git diff-*` (don't worry about these - I sure don't't!)

Some Examples

git log

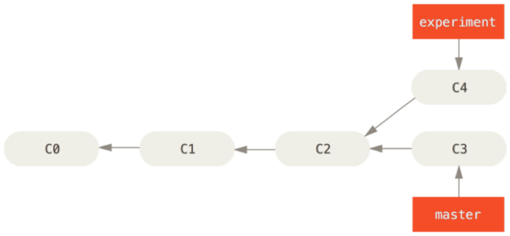
git log --graph
git log --graph --all
git log --graph --all --oneline

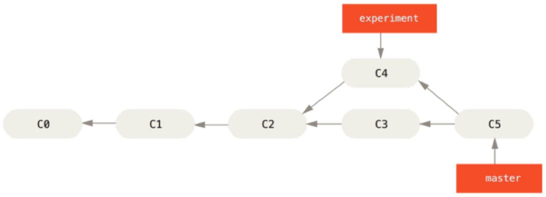
Git Basics: *Making Queries* -- `git log`

...Some Examples

```
git log --graph --abbrev-commit --decorate --
format=format:'%C(bold blue)%h%C(reset) - %C(bold
cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset) %C(bold
cyan)(committed: %cD)%C(reset) %C(auto)%d%C(reset)%n''
%C(white)%s%C(reset)%n''          %C(dim white)- %an <%ae>
%C(reset) %C(dim white)(committer: %cn <%ce>)%C(reset)'
```

Git Merge

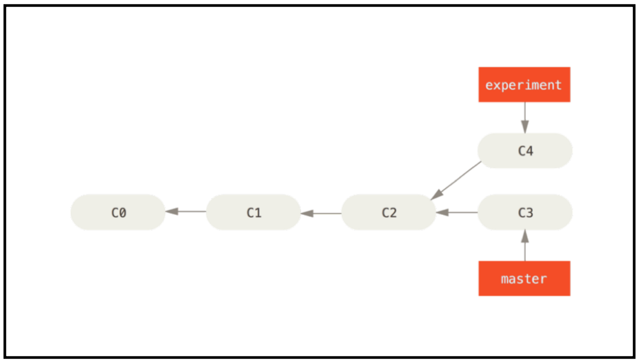


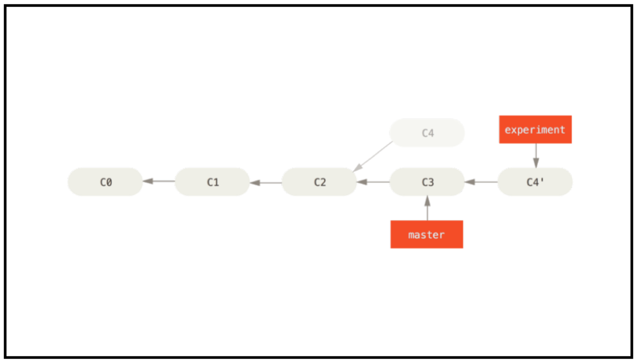


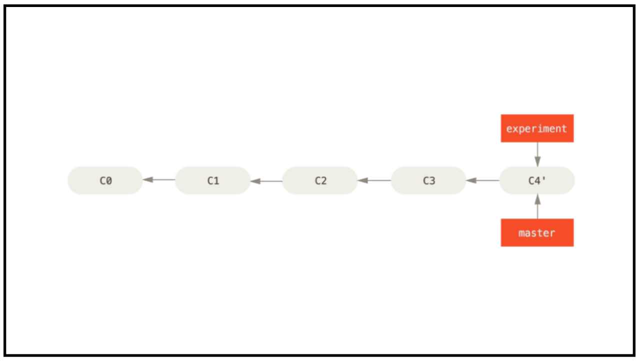
Git Rebase

Changing Commit History with Rebase

- Git rebase lets us change our commit history
- rebase is a powerful tool, but we will only scratch the surface

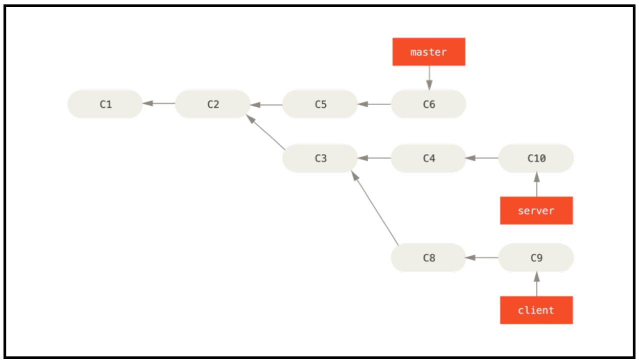


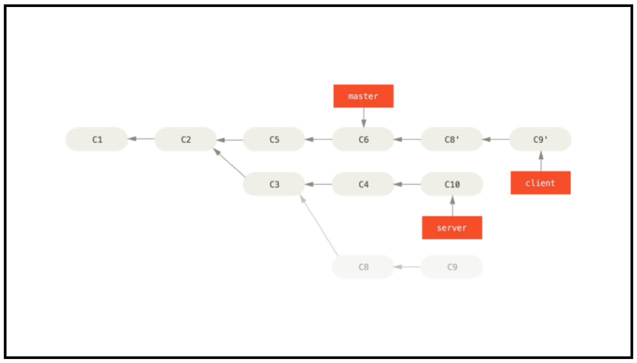


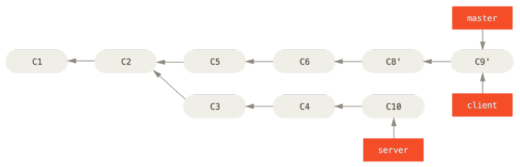


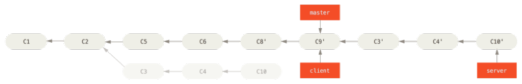
Changing Commit History with Rebase

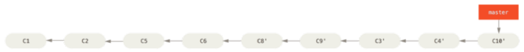
- Git rebase --onto gives us a bit more power











Why use Git Rebase?

Points of Confusion

Fork vs. Clone

Fork vs. Clone

Fork

Fork is NOT A GIT CONCEPT

- it was invented by GitHub
- Fork stores extra information and makes pull requests possible

Clone

Clone IS A GIT CONCEPT

- clone extends init
- exists independent of github

Branch vs. Clone

Branch vs. Clone

Branch

Branch creates a ref

Clone

Clone creates a new repository

Pull vs. Fetch

Pull vs. Fetch

Fetch

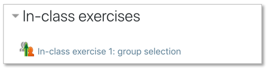
- Take target branch from a remote repository and store it in `.git/refs/remotes/`
- NOT integrated/merged with local branches!!!!!

Pull

- Fetches remote branch and *merges* with local branch or repository

Next time: Thursday (September 26)

- First in-class exercise
- On using git (today was a prelude with useful info)
- Form 4-person teams
  - Use moodle to self-select a team; can do it before Thursday or on Thursday



In-class exercises

In-class exercise 1: group selection

- At least one person per team needs to bring a laptop

BRING A LAPTOP!