Git Workshop @Salk Documentation

Release .02

Chris Hiestand

October 21, 2014

Contents

CHAPTER 1

Version Control Systems

If you write code you should be using a version control system. If you collaborate with other coders, you **really** should be using a version control system.

1.1 Terminology

- **Repository**: A collection of files, including a complete history and metatdata about those files including: all changes to the files, who made those changes, and when the changes were made.
- **Commit**: A group of files and folders that is a snapshot in time of a repository. A commit is triggered by the user. Commits are represented by a unique identifier, often a hash ID.
- **Branch**: Like a branch on a tree, a repository branch splits off from the trunk (or master, or default) and development happens independently.
- **Head** or **Tip**: The most recent commit in a branch. In git, HEAD is the latest commit in the current branch.
- **Tag**: A human readable label for a specific commit. Tags are created when you want to create a substantial reference point: e.g. "version1.0" or "Nature-July-2013"

1.2 Best Practices

1.2.1 What to put under version control

Any text files or small binary files that are a part of your project that you want version controlled. Images or figures that are part of your project are fine to include.

Examples of things often version controlled:

- Python
- Matlab
- HTML
- C
- Java
- LaTeX
- Images used by your project

- Various system configuration files

1.2.2 What not to put under version control

- Very large files (e.g. raw data)
- Large binary files (e.g. movies)
- Dynamic Files or Cache

Large files will quickly bloat your repository and slow down repository updates. Diffs generally cannot be made on binary files, so a small change to a large file will result in the whole file being stored twice. And because binary files cannot be parsed by the version controller, there is less utility in versioning them anyway.

Dynamic files that change automatically or by external processes generally have little value in being version controlled because it is not practical to keep repositories up-to-date and it is harder to discern important changes from typical ones. Or dyanmic files may bear little or no importance to the contents of the repository.

Fortunately if your project has such files, you can use your version control's ignore feature to ignore any project files you don't want version controlled.

That said, any of these may be version controlled if you have a good reason to do so.

1.2.3 Other tips

1. Before you commit, run a diff. Otherwise you may accidentally commit debug messages, unrelated changes, or development code that isn't ready.
2. All changes within a commit should have a logical unity. This makes changes easier to follow and easier to rollback. If you need two independent changes, commit one first, then the other.
3. In Unix or Linux, all version control systems use the EDITOR environment variable to pick which text editor to use for e.g. commit messages. Be sure to set your EDITOR environment variable to something you like.
4. If you want your results to be 100% reproducible, you should open source your code. After developing code in a DVCS, you can push your code to a public DVCSWebsite

1.2.4 How to write a commit message

The first line should be 50 characters or less, a short summary. Followed by a blank line and then a more in depth explanation, wrapped at around 72 characters.

Example Commit Message

Here's a great example taken from http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html .

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the present tense: "Fix bug" and not "Fixed
```

```
bug."  This convention matches up with commit messages generated by
commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

1.3 Traditional Version Control Systems (VCS)

A central repository is typically in one location. You can checkout copies of the centrally-located repository and then commit your changes.
RCS (don't use this, it's ancient)
CVS (don't use this, it's very old and not nearly as good as SVN)
Subversion (or svn for short) (note: SVN has recently adopted some features of a DVCS)

1.4 Distributed/Decentralized Version Control (DVCS)

These differ from the previous group in that there is not necessarily a central repository. Everytime you clone one of these, you are actually downloading the entire repository. You gain a lot of flexibility if you decide to use one of these. Because of their flexible nature, they all support excellent branching and merging features.

These are newer than the traditional type. If you are choosing which VCS to use and you don't have a specific need for SVN, you should use one of these. git is probably the most popular but it is perhaps the most difficult to learn. Mercurial (Hg) is easier to learn but not quite as popular.

For more help choosing one see:
http://en.wikipedia.org/wiki/Comparison_of_revision_control_software.

If you are totally indifferent, this author recommends git because it is roughly equivalent to the best available and it is the most popular. Popularity has a significant impact thanks to social coding.

All three of these are very similar in theory and usage; they just vary in implementation details:

- git
- Mercurial
- Bazaar

1.4.1 DVCS Websites

These are places where users can host their own DVCS repositories, with heavy social-coding features, web-based repository browsers, and other productivity features. If you want a free public repository where your code may be publicly downloaded or modified (with your approval), use one of these. They typically will sell private repositories at a modest price.

- github.com
- bitbucket.org
- launchpad.net
- sourceforge.net
- https://gitorious.org/
- http://code.google.com/

1.5 About git

git is a Version_Control_System designed by the Linux kernel development team with a couple design goals:

- Heavy support for branching and merging
- High performance

git achieves these priorities well; it is substantially faster than subversion. The expense is that sometimes the commands are a bit obtuse and the learning curve can be high. However for basics usage, git is not terribly hard to learn.

git uses sha-1 sums to keep track of versions. Unlike subversion, there is no simple version number.

git repositories can be bare or not. Bare means that the folder only contains the git repository. Not bare (or regular) means that the folder contains both the repository (in the root level .git folder) and the contents of the repository. Bare repositories are used as distribution points, normal repository are used as development points. bare repositories are usually named projectname.git. non-bare repositories are named projectname, with the .git repository folder as a child of that folder. The workflow goes:

1. You start hacking the code in your non-bare development repository
2. After your want to distribute your commits, you push your code to a bare repository. From there, other developers can pull your changes to their non-bare repositories.

The information presented here is not intended to cover all aspects of git. For further information please see the Git#References at the bottom of this page.

1.6 Private Collaboration git Repositories

If you would like to collaborate via the git version control system with others who do not have SNL accounts, we can now setup a private repository for you and your collaborators.

Just email us the name of the repository, the name and email of your collaborators, and a public SSH Key for each collaborator, including yourself.

If all collaborators have SNL accounts, it may be easier to just use the SNL file server(s) and shell.snl.salk.edu instead. Repositories can be migrated to the collaboration server later.

1.7 Best Practices

See Version_Control_Systems#Best_Practices for general version control best practices

1.8 Cheatsheet

```
git config --list                                       #View current options
git fetch                                               #Fetch commits from remote repository
git pull                                                #Fetch commits from remote repository and mer
git add filename                                        #Add filename to be tracked by the repository
git mv was.py nowis.py                                  #Rename file was.py to nowis.py
git rm filename                                         #Delete filename
git diff                                                #List the differences between HEAD and the cu
git diff master mybranch                                #List the differences between branches master
git diff HEAD~1                                         #List the differences between the parent and 
git diff HEAD~2                                         #List the differences between the grandparent
git diff master~1                                       #List the differences between the parent of m
git diff > patch.diff                                   #Create a simple patch
git apply patch.diff                                    #Apply a simple patch
git format-patch origin/master --stdout > big.patch     #Create patch in mailbox format (good for mul
git am < big.patch                                      #Apply patch in mailbox format
git commit -a                                           #Commit all changes
git commit --amend                                      #Reword the previous commit message
git revert HEAD                                         #Undo the commit that has already been pushed
git push                                                #Push commits to remote repository
git branch                                              #List all branches
git branch newbranch                                    #Create a new branch named newbranch
git checkout newbranch                                  #Switch to branch: newbranch
git checkout -b newbranch                               #Create a new branch named newbranch and swit
git merge master                                        #Merge commits from master into the current b
git clone REPO PATH                                     #clone repository from REPO to PATH
git tag                                                 #List all tags
git log --since = "7 days"                              #Show commit logs made in the past 7 days
git log --graph                                         #Show commit logs alongside an ascii-art depi
git log HEAD~4..HEAD                                    #Show commit logs for the previous 4 commits 
git log HEAD~4..                                        #Show commit logs for the previous 4 commits 
gitk                                                    #Starts graphical log viewer
git remote                                              #Show the associated remote repositories
git show                                                #Show changes made during the last commit
git show HEAD:path/to/file | wc -c                      #Show the filesize of the original file befor
git whatchanged filename                                #Show all commit logs that have modified this
git reset --hard ORIG_HEAD                              #Undo the previous merge
git rebase -i SINCE                                     #Rebase the commit tree since revision SINCE 
git fetch; git rebase origin master                     #Make your local branch identical to origin m
git stash list                                          #View stashes
git stash save "reminder msg"                           #Stash uncommited changes for use later.
git stash apply stash@{0}                               #Apply stash 0
git gc                                                  #Run garbage collection on repository (clears
```

1.9 git Commands

Use the **git help** command for a list of common commands.

```
chiestand@freeman:/cnl/data/chiestand/myproject$ git help
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [--html-path]
           [-p|--paginate|--no-pager] [--no-replace-objects]
           [--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE]
           [-c name=value] [--help]
           COMMAND [ARGS]
```

```
The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and merge with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG

See 'git help COMMAND' for more information on a specific command.
```

Use **git help [subcommand]** for further information, e.g.:

```
git help push
```

This brings up the man page.

1.10 Workshop Walk-Through

1.10.1 Configure git

```
#Set your name and email
git config --global user.name "FirstName LastName"
git config --global user.email "username@salk.edu"

#Colors improve readability
git config --global color.ui auto
git config --global color.status auto
git config --global color.branch auto
```

Double-check that it worked:

```
git config --global --get-regexp 'user.*'
```

You should see something like:

```
user.name FirstName LastName
user.email username@salk.edu
```

Be sure your EDITOR variable is set to a text editor you like:

```
echo $EDITOR
```

Output

```
/usr/bin/vim
```

If you just want a super-simple text editor, this should do the trick:

```
export EDITOR=/usr/bin/nano
echo "export EDITOR=/usr/bin/nano" >> ~/.bash_profile
```

If you don't understand how to set a text editor, ask for help.

1.10.2 Create a project and/or repository

Create a bare empty repository

You might do this if you are going to setup a central repository for yourself or other collaborators for a new project. Create this repository in a place where any collaborators can access it.

Don't forget to end your bare repository's folder name with .git, to follow convention.

```
cd
git init --bare myproject.git
#You can now clone this empty repository to development locations
```

Output:

```
Initialized empty Git repository in /home.local/chrish/myproject.git/
```

Create a normal empty repository

Create this first if you do not yet have an immediate need for a central repository for a new project. Instead, your work will start in a private development repository - you can always add a remote repository later.

```
cd
git init myproject
```

Alternatively, you could also create the folder first

```
#Alternatively
mkdir myproject
cd myproject
git init
```

Output:

```
Initialized empty Git repository in Initialized empty Git repository in /home.local/chrish/myproject
```

note: the .git folder is stored in the root directory. The .git folder is exactly the same as a bare repository by itself.

Now you can start version controlled work in the folder you've created.

```
#Clean this up for now, we'll create it again later
cd
rm -rf myproject
```

Create a bare and non-bare repositories by importing a non-versioned project

Do this if you have an existing un-versioned project you want to put under version control. This will create both a private development repository and a shared collaboration repository. If you do not need the shared repository you can skip those steps.

Go to the top level of the project and type:

```
mkdir my_previous_project #Now I'm inside the top level of my project
cd my_previous_project
echo 'file contents 1' > file1.txt
echo 'file contents 2' > file2.txt
git init
```

Output:

```
Initialized empty Git repository in /home.local/chrish/my_previous_project/.git/
```

Then add all files into the repository:

```
git add .
```

Verify there are no files that haven't been tracked (eg . hidden files):

```
git status
```

Output:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   file1.txt
#   new file:   file2.txt
#
```

Commit your files.

```
git commit -a -m "Initial import my_previous_project into a git repository"
```

Output

```
[master (root-commit) ba306e5] Initial import my_previous_project into a git repository
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 1
 create mode 100644 2
```

Lastly, you'll probably want to clone this git repository to a bare repository so you and colleagues can push to it.

```
cd
mkdir git   #I might store all my shared git projects in this subfolder for convenience

#Now I clone the git repository I've just created into a shared location
git clone --bare my_previous_project git/my_previous_project.git

#In actual use, this might be a website like github instead
```

Output:

```
Cloning into bare repository git/my_previous_project.git...
done.
```

Now set the new bare repository as the remote origin for the non-bare repository

```
cd my_previous_project
git remote add origin "$HOME/git/my_previous_project.git"

#Confirm that fetch works
git fetch

Everything up-to-date
```

1.10.3 Clone a repository

You can clone or fetch or push git data via a local file system, ssh, the git protocol, http, and ftp.

Clone locally

```
cd
git clone myproject.git myproject

Cloning into myproject...
done.

#Clean this up for now, we'll create it again later
cd
rm -rf myproject
```

Clone via git protocol

The git protocol is a simple and efficient protocol made just for transmitting git repository data. One place you might use this is cloning from a DVCS website like github.

For example, to clone from github:

```
#This will not work because our test server is not directly connected to the internet
# git clone https://github.com/sharpee/mid.git

#Instead for this class we will clone via HTTPS
git clone https://github.com/sharpee/mid.git
```

Output

```
Cloning into mid...
remote: Counting objects: 76, done.
remote: Compressing objects: 100% (55/55), done.
remote: Total 76 (delta 13), reused 76 (delta 13)
Receiving objects: 100% (76/76), 43.31 KiB, done.
Resolving deltas: 100% (13/13), done.
```

clone/push/fetch over ssh

If given a hostname as part of the source and no other protocol, git will ssh to the remote host to clone the repository. We recommend you have SSH Keys setup if you do this. If you use ssh more than once a day, you should have SSH Keys setup anyway because it prevents you from having to type in your password every time.

```
git clone git-workshop.snl.salk.edu:~/myproject.git myproject
```

Output:

```
Cloning into 'myproject'...
```

git will ssh for you any time you need to do a fetch or push

1.10.4 Manipulating Code in the Checked-Out Repository

All of these changes require commits afterwards to finalize. On the flipside, they are all easily undone until they are committed. Even when committed, undo is very easy until you have pushed. After a push, reversing a commit requires some thought: pushes cannot simply be reversed.

Adding Files or Folders

```
cd myproject
cp -r /workshop/vcs-example/* .  #copy everything over
```

These files are still unknown by git and must be added before they are tracked:

```
git status
```

Output

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README.md
#   brownian-1d.py
#   brownian-2d.py
#   library/
#   rank-nullspace.py
nothing added to commit but untracked files present (use "git add" to track)
```

So add them:

```
git add .
```

Added files still need to be committed or else will not be pushed.

```
git status
```

Output

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#   new file:   brownian-1d.py
#   new file:   brownian-2d.py
#   new file:   library/brownian.py
#   new file:   library/rank_nullspace.py
#   new file:   rank-nullspace.py
#
```

note: git cannot add an empty directory. If you want to add an empty directory put any file such as .gitignore inside the directory and add it.

Finally, make your first commit:

```
git commit -a
```

Leave a informational note in your text editor, then save the file and quit. Congratulations, this is the first commit.

Verify that there is nothing left unknown by git:

```
git status
```

```
# On branch master
nothing to commit (working directory clean)
```

Moving Files or Folders

```
git mv rank-nullspace.py rank.py
git status
```

Output

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   renamed:    rank-nullspace.py -> rank.py
#
```

Now commit your change

```
git commit -a -m "Renamed rank-nullspace.py to rank.py"
```

Output

```
[master 329d922] Renamed rank-nullspace.py to rank.py
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename rank-nullspace.py => rank.py (100%)
```

Copying Files or Folders

Because git stores content, there is no special copy command. git will recognize that content has been copied and will not store an additional copy in the database on a commit.

```
cp -r rank.py rank-test.py
git status
```

Output

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

git add rank-test.py
git commit -m "Added testing version of rank file"
```

Output

```
[master e5ba37c] Added testing version of rank-test.py
 1 files changed, 85 insertions(+), 0 deletions(-)
 create mode 100644 rank-test.py
```

Viewing Diffs

Now make a change to rank-test.py via a text editor, e.g.:

```
vim rank-test.py

git status
```

Output

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   rank-test.py
#
no changes added to commit (use "git add" and/or "git commit -a")

git diff
```

Output

```
diff --git a/rank-test.py b/rank-test.py
index 7a3c1d0..6679143 100644
--- a/rank-test.py
+++ b/rank-test.py
@@ -2,7 +2,7 @@ import numpy as np
 from numpy.linalg import svd


-def rank(A, atol=1e-13, rtol=0):
```

```
+def rank(A, atol=1e-13, rtol=1):
     """Estimate the rank (i.e. the dimension of the nullspace) of a matrix.

     The algorithm used by this function is based on the singular value
```

Now commit your change:

```
git commit -a -m "Changed relative tolerance for test"
```

Output

```
[master f5e7b68] Changed relative tolerance for test
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Ignoring files or folders

In order to make this program successfully run, you'll need an X11 session open. If you don't have one, don't worry, it's not terribly important. Type this command either way:

```
python brownian-1d.py
```

Then run git status and notice a .pyc booger is created

```
git status

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

We could just add this file, but it might be optimized differently on another computer. Further, it's going to change anything your code does - you don't want to have to track two different files when you only need to track one. Ignoring files is an important part of any VCS. Python programming is a good example, because the python compiler will create pre-optimized .pyc files which, as a programmer, you don't really care about. So let's ignore them.

Run this command:

```
echo -n "*.pyc" > .gitignore

#Alternatively
# vim .gitignore #and put *.pyc on the first line, save and quit
```

Now check the status again and notice that there is no mention of library/brownian.pyc

```
git status

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

Git tracks all your ignores in .gitignore files. Add this file so git will always ignore .pyc files within this repository, no matter where it has been cloned to.

```
git add .gitignore
git commit -a -m "Ignore .pyc files"
```

Output

```
[master eb1c4d9] Ignore .pyc files
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
```

Reverting Changes Before a Commit

There are a few ways to go about this. Careful, these will destroy your changes to everything if a specific file is not specified.

First make a change you will revert, e.g.:

```
vim rank-test.py
```

Here is the simplest method to revert it

```
git checkout -- rank-test.py
```

Another method

```
git reset --hard
```

Output

```
HEAD is now at eb1c4d9 Ignore .pyc files
```

Note: git reset is a powerful command. See **git help reset** for more.

Whichever method you use, double check that git is now in a clean state:

```
git status
```

Output:

```
# On branch master
nothing to commit (working directory clean)
```

Deleting Files or Folders

```
git rm rank.py
```

Output

```
rm 'rank.py'
```

Now commit your change:

```
git commit -a -m "Removed rank code"
```

Output

```
[master 3a736a0] Removed test rank code
 1 files changed, 0 insertions(+), 85 deletions(-)
 delete mode 100644 rank-test.py
```

Revewing the changelog

```
git log
```

Output

```
commit 3a736a0da6e24cd38ee33c59811daf7a784674da
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:48:26 2012 -0700

    Removed rank code

commit eb1c4d917381ca7a955c7effe1a295cbc390df55
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:43:40 2012 -0700

    Ignore .pyc files

commit 9f60b55f78fe0a65115f8cbc6c7f76f9703bae06
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:35:19 2012 -0700

    Adding brownian motion scripts

commit f5e7b6843493a6414c5beb53b60c8c691a40014f
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:31:44 2012 -0700

    Changed relative tolerance for test

commit b5d77dc1c584f62564a4161eb3f886c707163ba1
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:29:31 2012 -0700

    Added testing version of rank file

commit 8446a130e8f4730cd6d4821e7fc4759bbfe7628d
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:29:18 2012 -0700

    Renamed rank_nullspace.py to rank.py

commit bfe09051c18b39cfdda94fd62aa7ba58d81e5061
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:28:53 2012 -0700

    Initial commit of scipy example code
```

Who Was Responsible for This?

Find out who made the most recent change to every line in a file. Columns are in order: commit, file, Author and date, line number and line value.

```
git blame rank-test.py
```

Output

```
^bfe0905 rank_nullspace.py (Chris Hiestand 2012-03-30 19:28:53 -0700  1) import numpy as np
^bfe0905 rank_nullspace.py (Chris Hiestand 2012-03-30 19:28:53 -0700  2) from numpy.linalg import svd
^bfe0905 rank_nullspace.py (Chris Hiestand 2012-03-30 19:28:53 -0700  3)
^bfe0905 rank_nullspace.py (Chris Hiestand 2012-03-30 19:28:53 -0700  4)
570bdcf8 rank-test.py      (Chris Hiestand 2012-03-30 19:51:24 -0700  5) def rank(A, atol=1e-13, rtol
^bfe0905 rank_nullspace.py (Chris Hiestand 2012-03-30 19:28:53 -0700  6)     """Estimate the rank (i
^bfe0905 rank_nullspace.py (Chris Hiestand 2012-03-30 19:28:53 -0700  7)
...
```

What is the history of this string?

To track the history of a string in your repository, in any file throughout the history, use the pickaxe feature.

```
git log -S"import numpy as np"

commit 3a736a0da6e24cd38ee33c59811daf7a784674da
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:48:26 2012 -0700

    Removed rank code

commit b5d77dc1c584f62564a4161eb3f886c707163ba1
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:29:31 2012 -0700

    Added testing version of rank file

commit 8446a130e8f4730cd6d4821e7fc4759bbfe7628d
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:29:18 2012 -0700

    Renamed rank_nullspace.py to rank.py

commit bfe09051c18b39cfdda94fd62aa7ba58d81e5061
Author: Chris Hiestand <chiestand@salk.edu>
Date:   Fri Mar 30 19:28:53 2012 -0700

    Initial commit of scipy example code
```

git submodules - using multiple repositories

Adding a submodule to your repository

git submodules allow you to include other git repositories into your repository. These are critically useful to build on other people's projects.

```
#Clone Tatyana Sharpee's MID and put it into the mid directory
git submodule add https://github.com/sharpee/mid.git mid
```

Output

```
Cloning into mid...
remote: Counting objects: 76, done.
remote: Compressing objects: 100% (55/55), done.
remote: Total 76 (delta 13), reused 76 (delta 13)
```

```
Receiving objects: 100% (76/76), 43.31 KiB, done.
Resolving deltas: 100% (13/13), done.
```

Notice how this changes your repository:

```
git status
```

Output

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   .gitmodules
#   new file:   mid
#
```

Now commit

```
git commit -a -m "Adding mid submodule to mid folder"
```

Output

```
[master f28bcf9] Adding mid submodule to mid folder
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 mid
```

Initializing a submodule in a cloned repository

Clone the first repository elsewhere

```
cd ..
git clone myproject myproject2
```

Output

```
Cloning into myproject2...
done.
```

Initialize the submodules

```
cd myproject2/
git submodule init
```

Output

```
Submodule 'mid' (https://github.com/sharpee/mid.git) registered for path 'mid'
```

Update the submodule(s)

```
git submodule update
```

Output

```
Cloning into mid...
remote: Counting objects: 76, done.
remote: Compressing objects: 100% (55/55), done.
remote: Total 76 (delta 13), reused 76 (delta 13)
Receiving objects: 100% (76/76), 43.31 KiB, done.
```

```
Resolving deltas: 100% (13/13), done.
Submodule path 'mid': checked out 'af9a133d3dcd6da020c20b2fe73afbcf9a4bbb27'
```

Where does the repository live?

The primary method is:

```
git remote -v
```

Output

```
origin      /home.local/chrish/myproject (fetch)
origin      /home.local/chrish/myproject (push)
```

Or look in .git/config in the root folder, or type:

```
git config -f .git/config --get remote.origin.url
```

Output

```
/home.local/chrish/myproject
```

Somebody moved the repository, how do I update my checked-out version?

You can use git remote:

```
git remote rm origin
git remote add origin protocol://user@remotehost.tld/path/to/new/repo.git
```

Or edit .git/config in the root folder and update the remote origin url or:

```
git config -f .git/config --replace-all remote.origin.url protocol://user@remotehost.tld/path/to/new/
```

1.10.5 Branching and Merging

This is where a DVCS like git really shines.

```
cd
git init --bare helloworld.git
git clone helloworld.git helloworld
cd helloworld
$EDITOR hello.py
#Put in the following code then save the file
```

```
print "hello world"
```

```
#make sure it runs
python hello.py
```

Output

```
hello world
```

```
git add .
git commit -a -m "initial creation of hello world python app"
git push origin master
```

Output

```
[master (root-commit) 2471a36] initial creation of hello world python app
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py

Counting objects: 3, done.
Writing objects: 100% (3/3), 256 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home.local/chrish/helloworld.git
 * [new branch]      master -> master
```

Creating a Branch

The first branch of every repository is master. Before switching to another branch, there should be nothing uncommitted in your current branch.

Create the branch experimental

```
git status #Visually check that working directory is clean
git checkout -b py3 #Creates branch py3 and switches to it
vim hello.py
```

```
print("hello world")
git commit -a -m "Switched to python3 style print"
```

In parallel, let's make a change to the master branch

```
git checkout master
vim hello.py
```

```python
print "hello world!"
```

```
git commit -a -m "Added some enthusiasm"
```

Now let's try to merge the two branches. What do you think will happen?

```
git status #verify we are in master branch. The branch that is changed is always the one currently ch
git merge py3
```

Output

```
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
```

Conflict Resolution

If two commits in separate branches change the same line in the same file, they cannot be automatically merged so a conflict is thrown. You must manually resolve the conflict for git to continue.

We have to manually resolve the conflict

```
$EDITOR hello.py
```

```python
print("hello world!")
```

Test our change

```
python hello.py
```

Output

```
hello world!
```

Now mark the merge as resolved by using git add

```
git add hello.py
git commit
git push origin master
```

Output

```
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 749 bytes | 0 bytes/s, done.
Total 9 (delta 1), reused 0 (delta 0)
To /home.local/chrish/helloworld.git
   2471a36..3af56e2  master -> master
```

If you want the remote repository to have a copy of the py3 branch, push it there:

```
git push origin py3
```

Output

```
Total 0 (delta 0), reused 0 (delta 0)
To /home.local/chrish/helloworld.git
 * [new branch]      py3 -> py3
```

Listing branches in the repository

The current branch gets an asterisk next to it:

```
git branch
```

Output

```
* master
  py3
```

Tagging

Imagine you have published a paper, and you want to freeze your code at the point in time used to generate data for the paper. Simply make a tag for that code.

```
#Tag the current version of master
git tag j-neurosci-hippo-2014

#Alternatively, make a tag for a previous commit:
git tag j-neurosci-hippo-2014 HEAD~1
#or
git tag j-neurosci-hippo-2014 a6f8d555e142d83f03261fb762b26120d2646ad1

#Check your list of tags:
git tag
```

Output

```
j-neurosci-hippo-2014

#Try checking out a tag:
 git checkout j-neurosci-hippo-2014
```

Output

```
Note: checking out 'j-neurosci-hippo-2014'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 3af56e2... Merge branch 'py3'

#switch back to master
git checkout master
```

1.10.6 Rebasing

Rebasing, contrasted with merging, results in all commits appearing to have been done in series (instead of done in parallel and eventually merged). Rebasing can be done at any point before a push or merge. If your workflow involved a lot of branching and merging, you may choose to rebase instead of merge in order to make the history easier to parse. Rebasing is potentially destructive of a change history, so if you want to be sure to keep your development before a rebase make you're developing in a branch - otherwise your work may be deleted after garbage collection.

Interactive Rebasing

This is a simple form of rebasing that is useful to create feature-complete commits. Imagine you are working on feature X which involves changes to several places in the code. It would be a best practice if you committed your changes every step along the way. But when you push your changes back to the bare repository, for others to use, it would be convenient if you only pushed back a single, feature-complete, commit containing all of of your changes.

You can use interactive rebasing to squash previous commits and combine them into one. This is the voltron of git commands.

```
#Create a new branch and make 3 commits within it
git status  #Visually ensure we are on master
git checkout -b multi-lingual
```

First new commit

```python
import sys
print("hello world!")
```

```
git commit -a -m "first new commit"
```

Second new commit

```
import sys

language = 'english'

if language == 'english':
    print("hello world!")

else:
    print("universal translator needs tuning")
```

```
git commit -a -m "second new commit"
```

Third new commit

```
import sys

if len(sys.argv) > 1:
    language = sys.argv[1]
else:
    language = "english"    #the default

if language == 'english':
    print("hello world!")
else:
    print("universal translator needs tuning")
```

```
git commit -a -m "third new commit"
```

Let's clean up the commit history and combine the last three commits

```
git rebase -i HEAD~3
```

So we'll rebase HEAD against HEAD three generations ago:

The interactive git rebase menu

```
pick 7404803 first new commit
pick cf3752c second new commit
pick 11de03d third new commit

# Rebase 3af56e2..11de03d onto 3af56e2
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Squash 3 of the commits:

```
pick 7404803 first new commit
squash cf3752c second new commit
squash 11de03d third new commit
```

Write your new log message:

```
added logic for a multi-lingual hello world
```

The final output should look like this:

```
[detached HEAD e96223f] added logic for a multi-lingual hello world
 1 file changed, 11 insertions(+), 1 deletion(-)
Successfully rebased and updated refs/heads/feature1.
```

Now if you look at the log or diff history, you'll only see one commit instead of four.

```
git log

#push the new branch up to origin
git push origin multi-lingual
```

Though it is common to use rebasing to squash commits, it's even more common to use rebasing as a substitute for merging, as we will see next.

Typical Ideal Workflow

A really clean workflow is if you rebase (as opposed to merging) changes from master into a branch, and merge (as opposed to rebasing) changes from a branch into master.

```
git fetch                          #check for collaborator changes
git checkout master                #Ensure we are on master
git merge --ff-only origin/master  #ensure most up-to-date branch
```

Let's create a new feature branch

```
git checkout -b monty-python
$EDITOR hello-monty.py
```

```python
import random

things = ['world', 'duck', 'goat cheese', 'moon', 'quantum flapdoodle']
thing = random.choice(things)

print("hello {thing}!".format(thing=thing))
```

```
git add hello-monty.py
git commit -a -m "added python-esque hello world"
```

In parallel, we can make changes to the master branch. For ease of demonstration, we will ensure they are non-conflicting changes by modifying a different file.

```
git checkout master
$EDITOR hello.py
```

```python
from __future__ import print_function
from __future__ import unicode_literals

print("hello world!")
```

```
git commit -a -m "add future unicode/print"
```

Now let's rebase the master changes into our monty-python branch

```
git checkout monty-python
git log
git rebase master

First, rewinding head to replay your work on top of it...
Fast-forwarded monty-python to master.

git log #Note the last commit to master is functionally here
```

Note that the commit to master is listed *below* the commit to monty-python

Now let's make another change to monty-python

```
$EDITOR hello-monty.py

import random

things = ['spam', 'duck', 'goat cheese', 'moon', 'quantum flapdoodle']
thing = random.choice(things)

print("hello {thing}!".format(thing=thing))

git commit -a -m "What kind of python list has no spam?"
```

We intend to merge the monty-python branch back into master. But first, let's ensure that monty-python is up to date, in that it already has the latest commit from master

```
git rebase master

Current branch monty-python is up to date.
```

Now we're ready to merge to master, knowing we can fast forward merge.

```
git checkout master
ls
git log
git merge --ff-only monty-python
ls
git log
```

1.11 References

http://gitref.org/ : The git reference site
http://schacon.github.com/git/gittutorial.html : The official git tutorial
http://git-scm.com/documentation : Official documentation
http://book.git-scm.com/ : The Git community book
http://githowto.com/ : A good git tutorial
http://vimeo.com/14629850 : A great intro video to using git