Version Control Systems

Demitri Muna
OSU

23 June 2014

What You Want When Writing Code

- Backups
- Ability to see a previous version of your code
- Marking code that works / is stable
- Marking code that ran a particular analysis
- Access to your code from anywhere
- Synchronize changes to code across multiple computers.
- Share your code with people.

Most people try to accomplish some of these things by hand, but often forget to do (or just skip!) one more steps because it isn't easy or is time consuming.

(And really, you want most of these things for *all* your files!)

File Versioning Tools

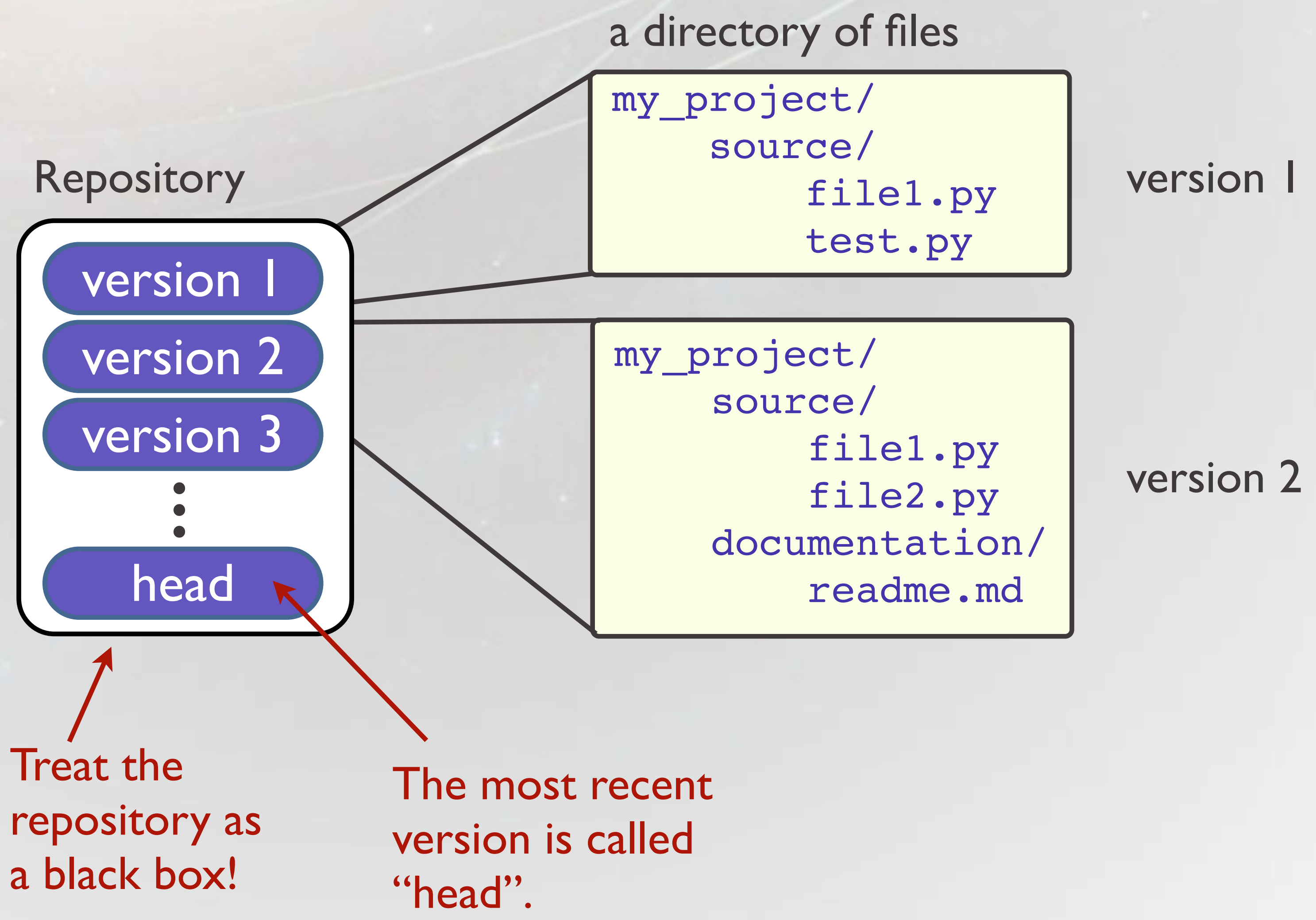
- There are software tools that provide all of the above.
- Easy to integrate with your daily habits with minimal effort.
- Most popular tools you'll come across:
    - Git – the most popular tool now
    - Mercurial – a good, simpler alternative to Git
    - Subversion – still in common use, easy to use
    - CVS – older, hard to use, pretty much obsolete
- All have web interfaces for access anywhere (or convenient public access). This is *not* built into any of the above.
- Some editors integrate these tools (Xcode, Eclipse).

Integrating Version Control Into Your Workflow

- Many ways to organize a repository; I'll show you one, but feel free to adapt to your needs.
- The most important thing is to *use* it!
- Most any file can be saved into a repository (text, images, mp3s, ... nearly anything).

The Repository

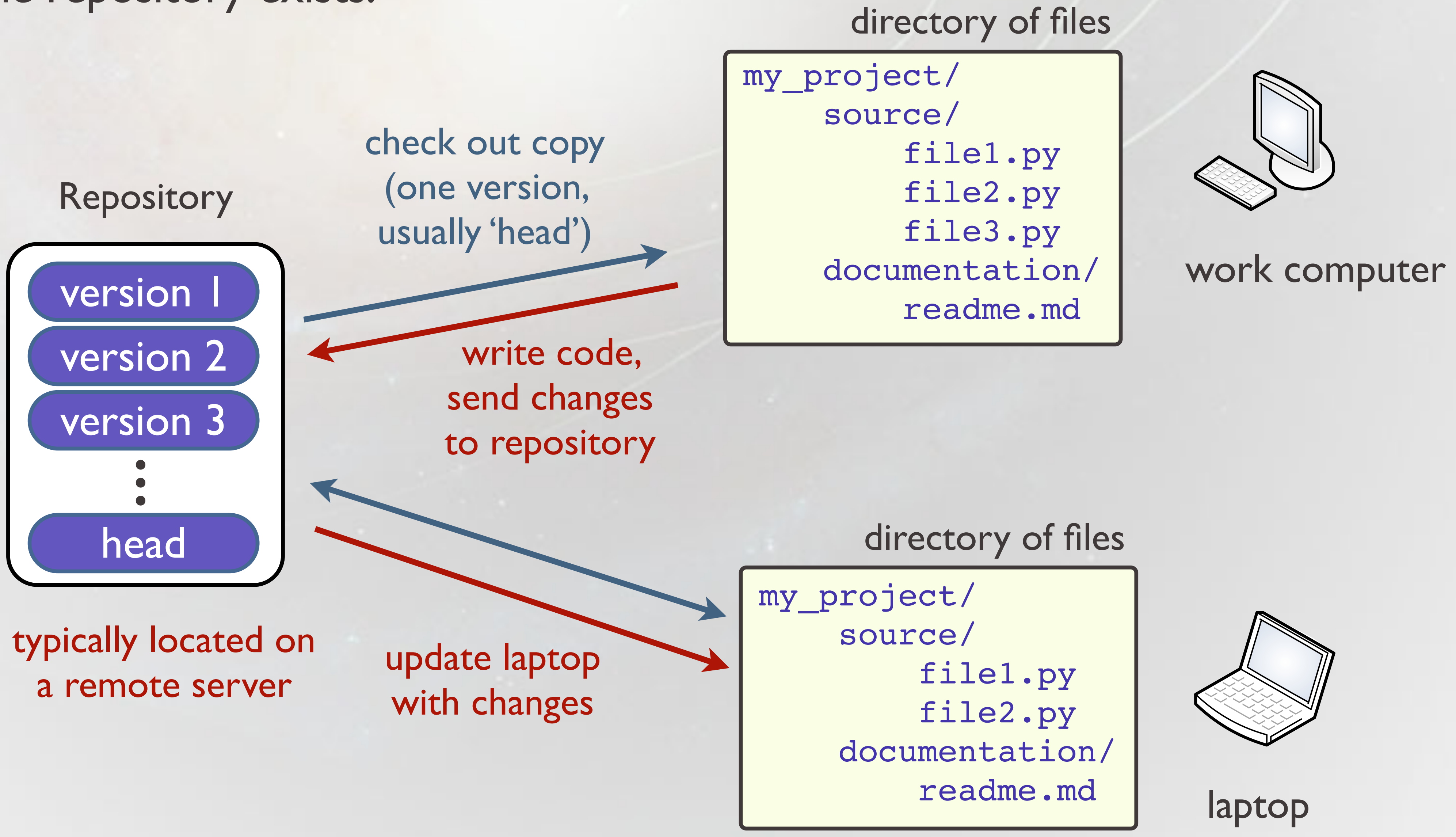
A place where all versions of all files are stored. With this, the current version or any prior version of any file in the repository can be recovered. Can be locally or remotely located. If only a local copy (i.e. on your own hard drive) is created, it doesn't provide a backup in case of computer failure.



Kinds of Repositories

Central Repository

Only one repository exists.

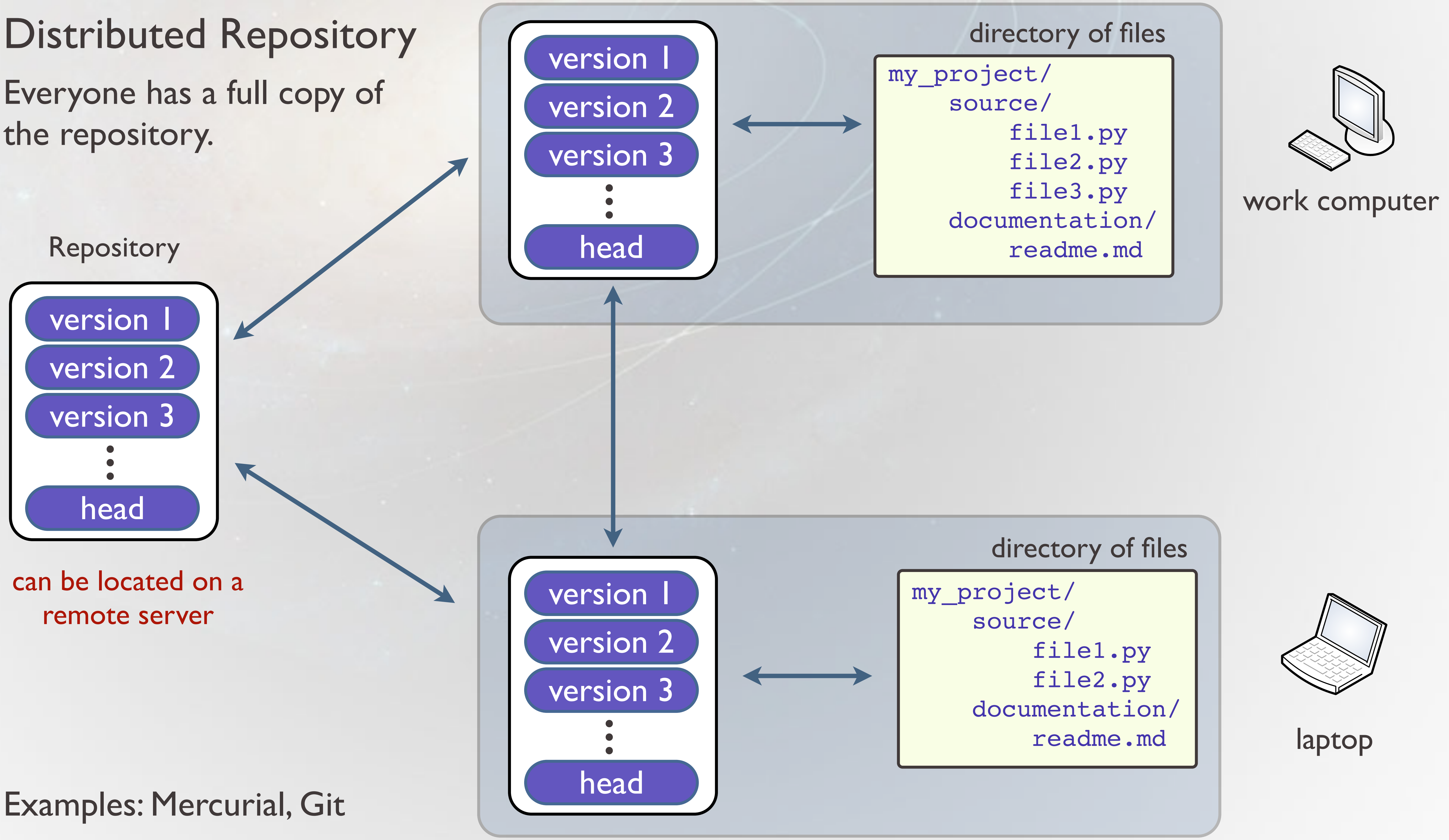


Examples: SVN, CVS

Kinds of Repositories

Distributed Repository

Everyone has a full copy of the repository.



Examples: Mercurial, Git

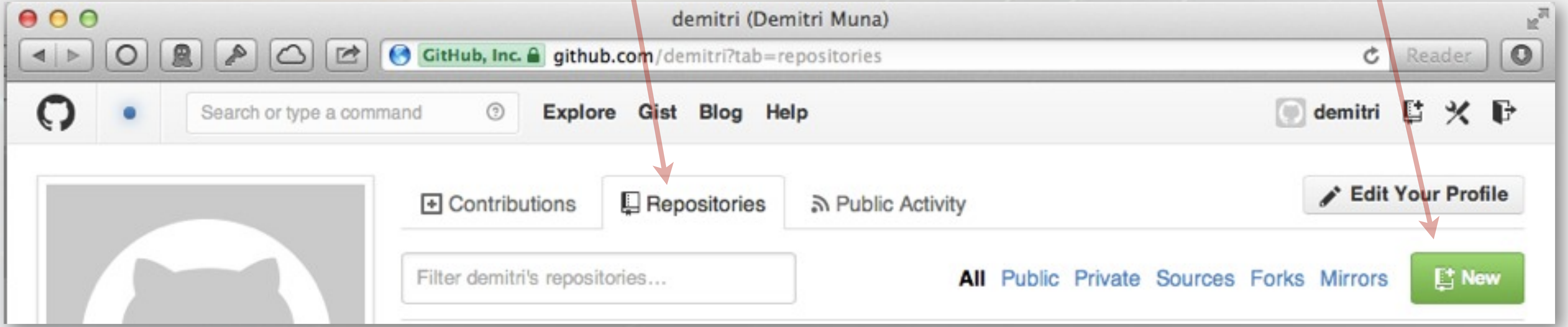
How Many Repositories?

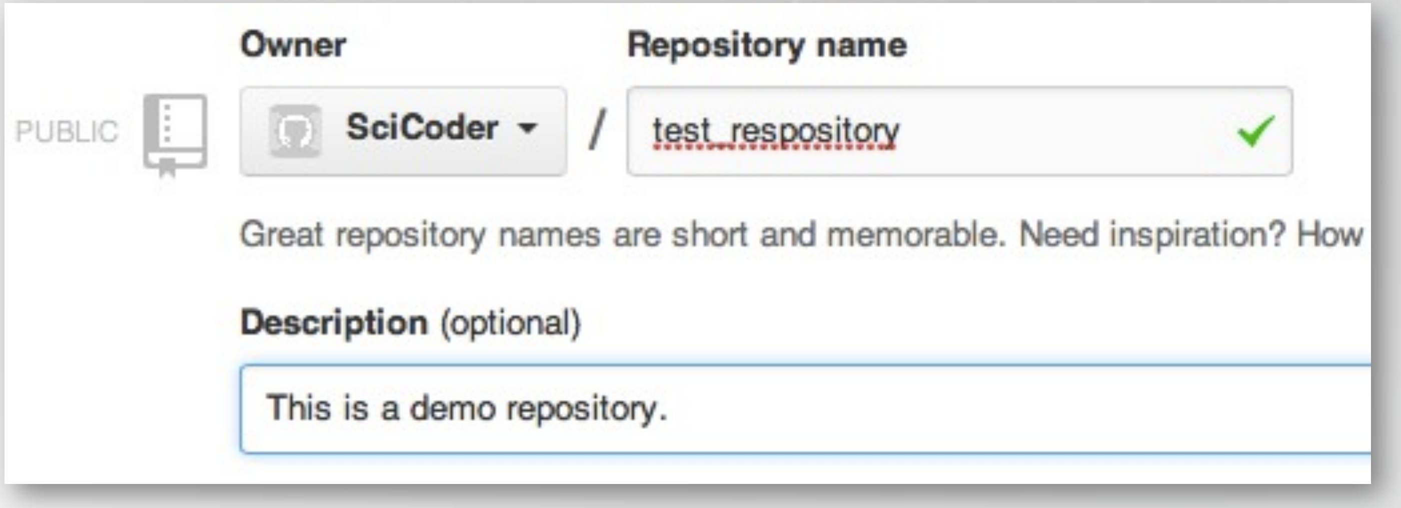
- Typically, you'll create one repository for every project. This allows you to provide access to others on a project-by-project basis.
- Create your own repository to contain all code you write that is not otherwise contained in another repository.
- Avoid keeping very large data files in your repository. Small data files as appropriate are ok (e.g. data that code depends on).

Creating a New Repository on GitHub

Go to your account on GitHub, select "Repositories".

Create new repository.



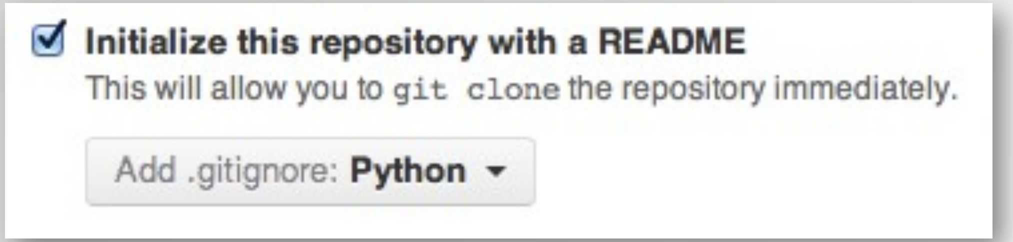


Owner: SciCoder / Repository name: test_respository

Great repository names are short and memorable. Need inspiration? How

Description (optional)

This is a demo repository.



☑ Initialize this repository with a README

This will allow you to `git clone` the repository immediately.

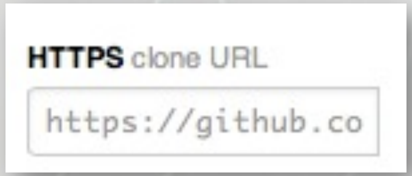
Add .gitignore: Python

Initialize repository, select primary language to use (more on "ignores" laters).

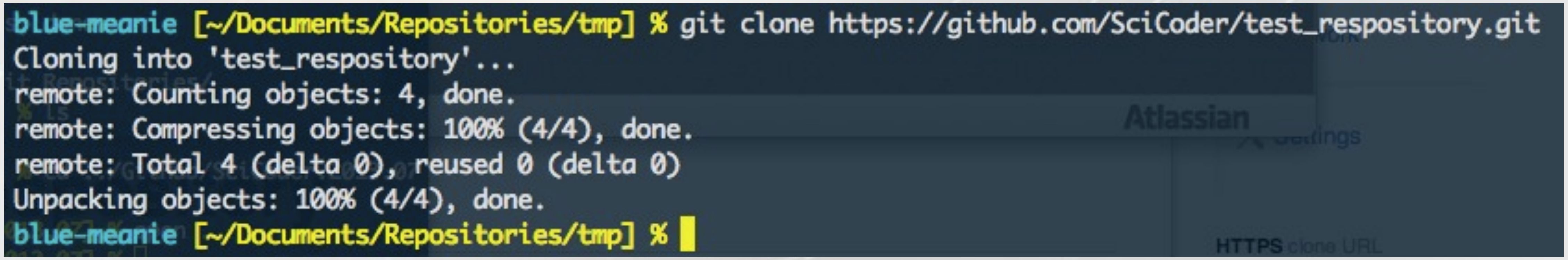
Fill in a name & description.

Cloning the Repository

On the lower right on the next page, copy the "clone URL".

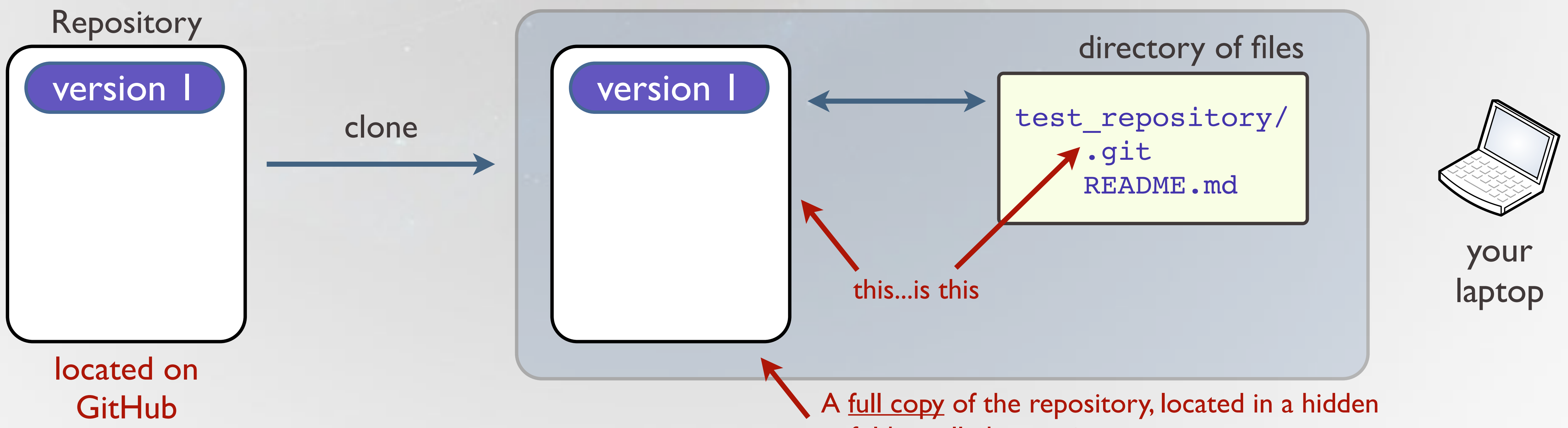


In the terminal, enter: `git clone <URL>`



```
blue-meanie [~/Documents/Repositories/tmp] % git clone https://github.com/SciCoder/test_respository.git
Cloning into 'test_respository'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
blue-meanie [~/Documents/Repositories/tmp] %
```

What does this do? You've made a local copy of the repository. Now there are two copies of the repository and one copy of the files (your "working directory").



A full copy of the repository, located in a hidden folder called `.git` in `test_repository`.

Cloning the Repository

In git, repositories do not have names. However, we do need to refer to other repositories, e.g. the one on a remote server versus the one on our computer.

The command

```
git clone <URL>
```

will automatically copy the repository at the given URL to your computer and then name the remote repository so you can refer to it. The default name it gives is "origin". This command is equivalent to the above:

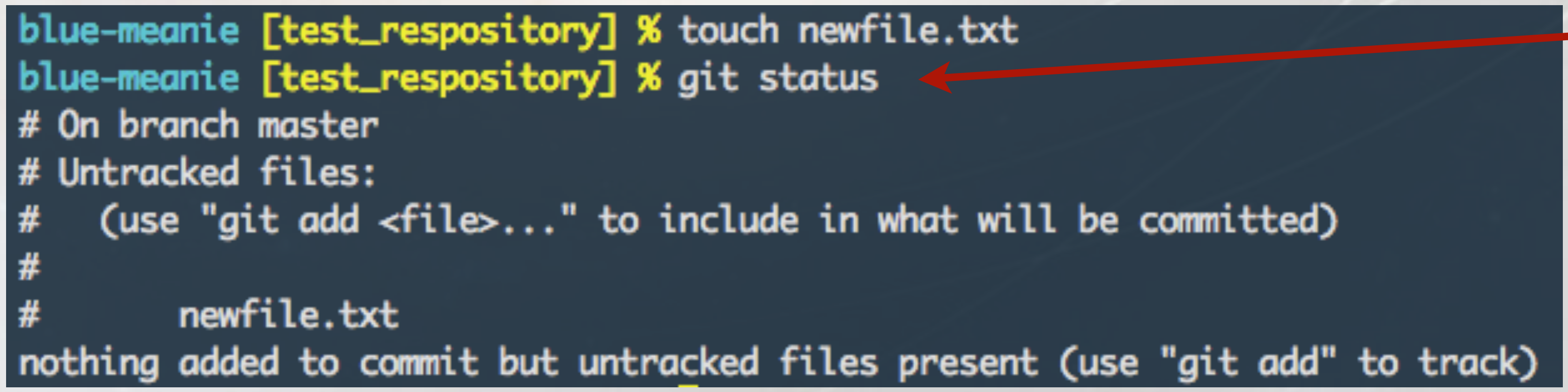
```
git clone --origin origin <URL>
```

the flag the name

You'll see "origin" a lot – this is what it is referring to.

Adding a File to Your Repository

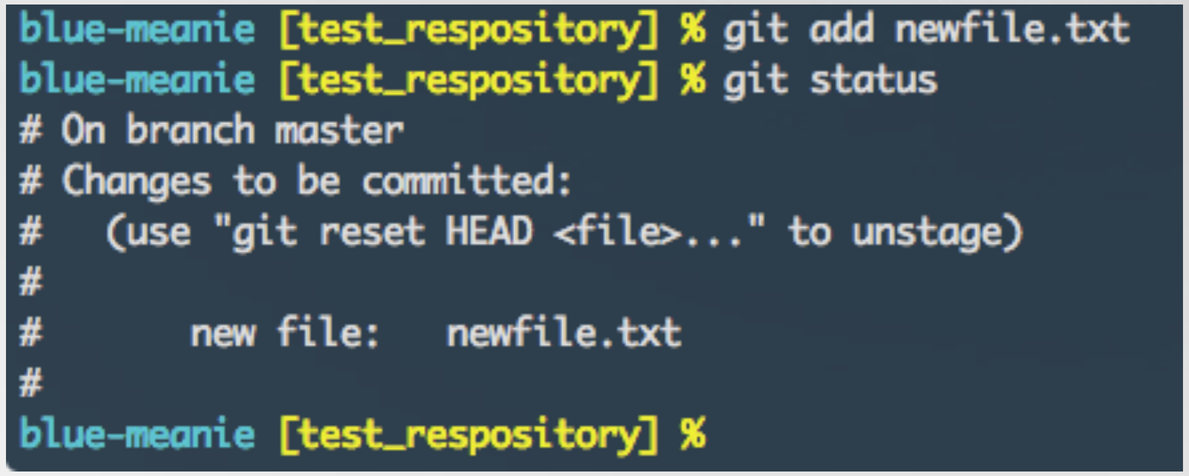
Create a new file an place it into the repository (`touch newfile.txt` is useful here). It's currently untracked. This means that git won't save or do anything with this file. To see this, type "`git status`".



```
blue-meanie [test_respository] % touch newfile.txt
blue-meanie [test_respository] % git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       newfile.txt
nothing added to commit but untracked files present (use "git add" to track)
```

gives you a status of your working directory & local repository - you'll use this command a lot

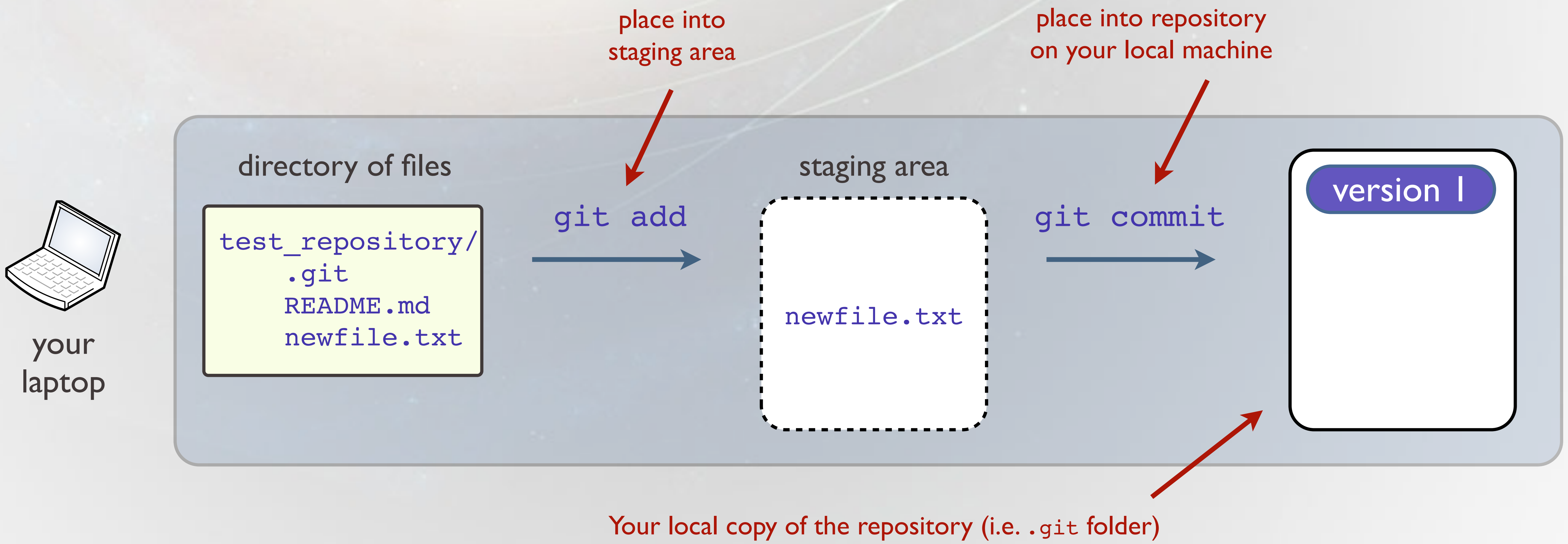
We want to add this file to the repository, so we say `git add newfile.txt` to do so. From now on, this file is "tracked" by git, but it's not yet in the repository.



```
blue-meanie [test_respository] % git add newfile.txt
blue-meanie [test_respository] % git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   newfile.txt
#
blue-meanie [test_respository] %
```

Git Staging

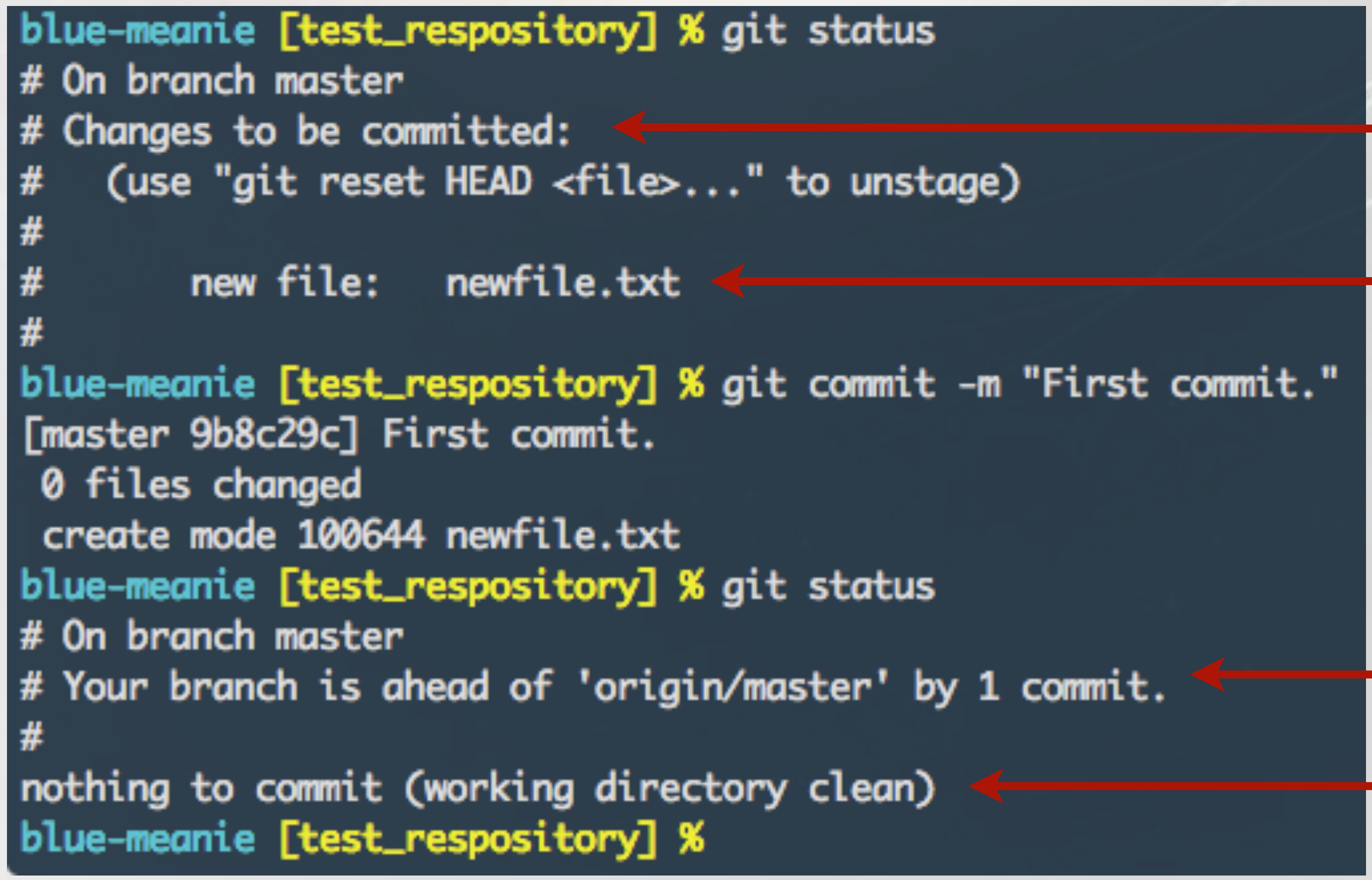
Adding a file puts it into a "staging area", essentially telling git you want to save this version of the file. Later. Not now though. It's like a promise ring. "Get ready git. I might make a commitment. At some point."



The `git commit` command will then actually save the file into the repository.

Committing Files

You need to be roughly aware staging happens. I do not recommend you use this feature. When you area ready to add files, place them into the repository IMMEDIATELY with the `commit` command.



```
blue-meanie [test_respository] % git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   newfile.txt
#
blue-meanie [test_respository] % git commit -m "First commit."
[master 9b8c29c] First commit.
 0 files changed
 create mode 100644 newfile.txt
blue-meanie [test_respository] % git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
blue-meanie [test_respository] %
```

i.e. in staging area limbo

list of files that will be committed

A description of the changes *must* be specified with every commit, easiest with the "-m" flag.

Yeah, that's helpful. We'll get to this.

This means that the latest version in the repository matches the files in the working directory.

NOTE: Committing files only saves them to your local repository.

Committing Files

Can we just pretend the whole staging this doesn't exist?! YES!

Staging is annoying, and I won't use it.
Mnemonic: **a**dd things for me

```
% git commit -a -m "Fixed all the bugs."
```

```
git add "file1.txt"
git add "file2.txt"
git commit -m "bugs fixed"
```

=

```
git commit -a -m "bugs fixed"
```

But *only* for files that are being tracked! You have to "add" them yourself at least once.

IF you use staging (don't):

- The typical workflow will be to edit files, '`git add`' them, then immediately commit with a message.
- If you '`git add`', commit immediately. Don't leave files in the staging area.
- If you '`git add`', then modify the file, you will need to '`git add`' again.
- '`git commit`' by itself will open your designated text editor to prompt (force) you to enter a message. Just always use the '`-m`' flag.

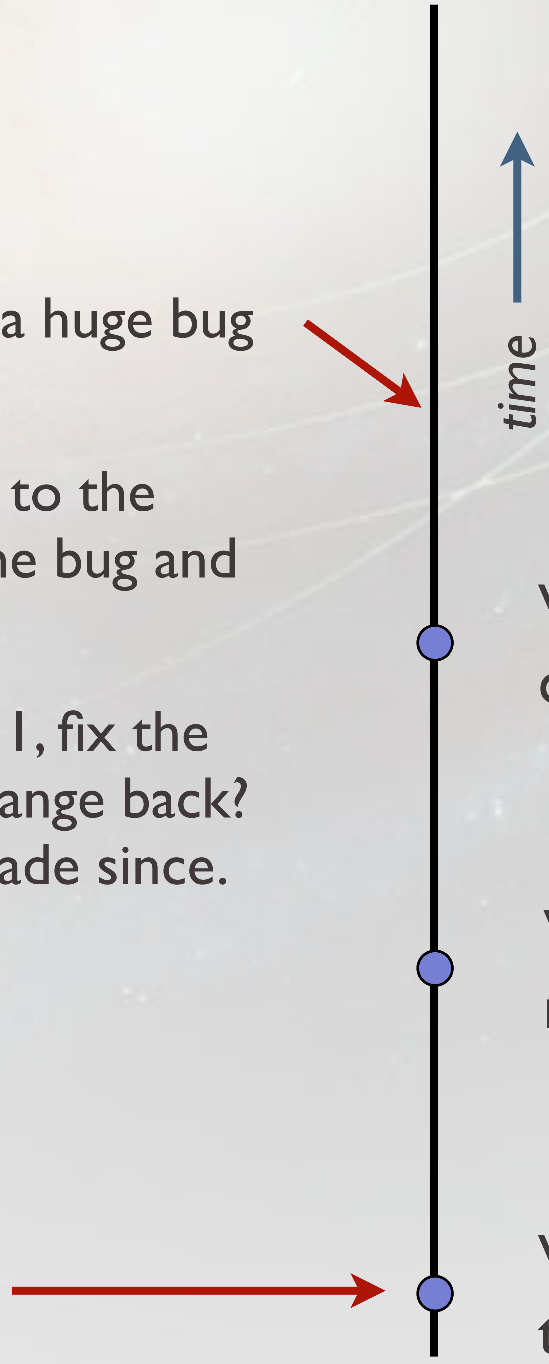
Branching

OK, now it gets complicated.

Someone sends an email about a huge bug in version 1. Oops.

But we've made lots of changes to the code now; we can't simply fix the bug and release a new update.

Well, we can check out version 1, fix the bug, but how do we save the change back? Too many changes have been made since.



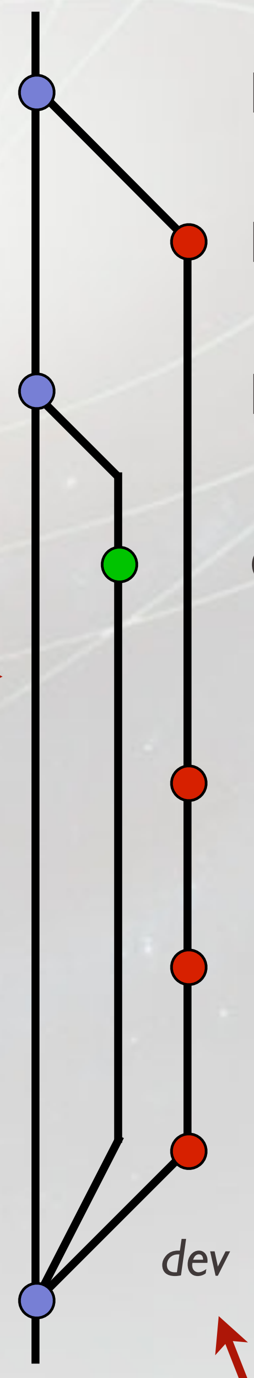
Version 3; active development now, more changes made.

Version 2; we've made lots of changes to add new features.

Version 1 of our repository – we release this code for other people to use.

Branching

Let's try this again.



Merge development back into *master*, release code.

More development.

Merge fix back into *master*, release this code.

Create a new branch "*bugfix*" based off version 1, not the current code. Fix bug.

Someone sends an email about a huge bug.

Active development now, more changes made.

Lots of changes to add new features.

Start a new *branch* – an independent copy of one version of the repository. This is like having two repositories in one.

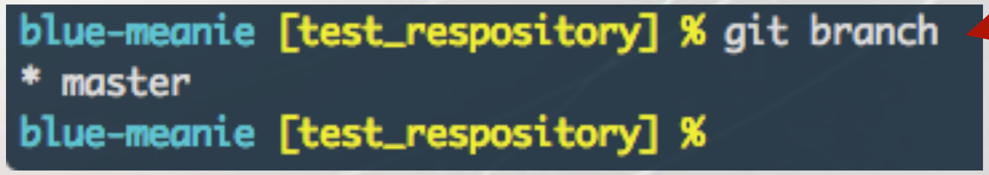
Version 1 of our repository – we release this code for other people to use

each branch has its own name

Branching

Your local working directory always represents one branch. To see what branches are in our repository:

first, default branch is automatically named *master*



```
blue-meanie [test_respository] % git branch
* master
blue-meanie [test_respository] %
```

list all branches

the current branch has a * in front of the name

To create a new branch:

`git branch` *`branchname`*

This creates a new branch, but your working directory does not change. To change your working directory to the new branch:

`git checkout` *`branchname`*

```
git branch branchname
git checkout branchname
```

=

```
git checkout -b branchname
```

Saving To Another Repository

What did this line mean...?

name we call the remote repo

now we know this is the name of the main branch

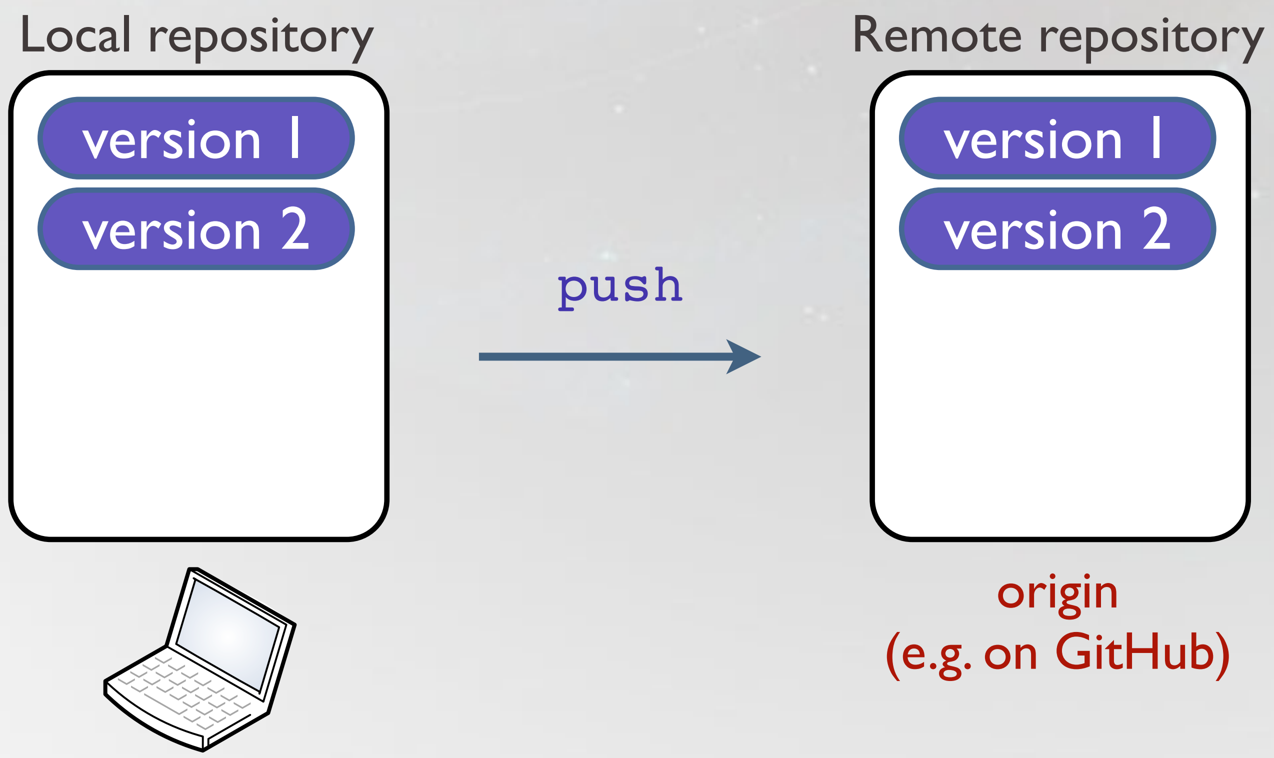
```
# Your branch is ahead of 'origin/master' by 1 commit.
```

The local repository (your computer) has a newer update than what is on the remote server (which we call "`origin`"). The line above means that one commit occurred after the last commit on the master branch on the remote repository `origin`.

We want to send those changes to the remote repository... this is called a *push*:

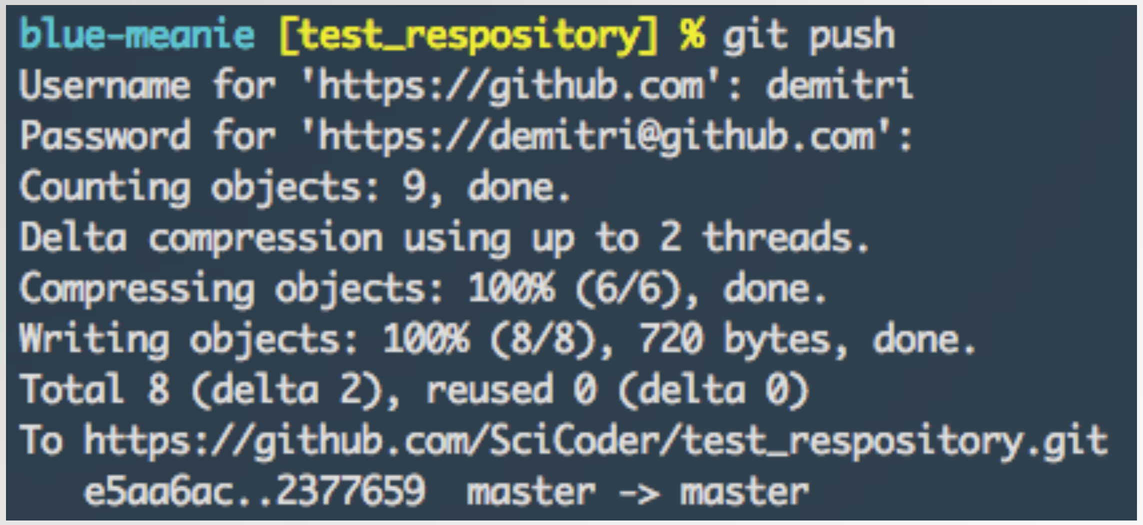
```
git push <remote repo name> <branch name>
```

note default remote name is "origin" and default branch is "master", defined in .git/config



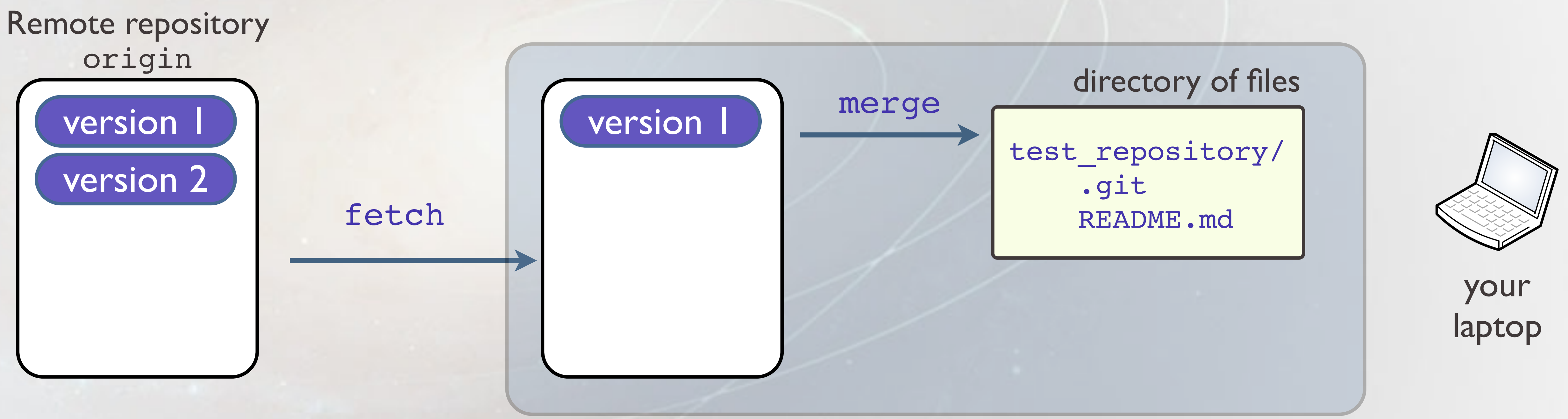
In our example from before, this would be:

```
git push origin master
```



```
blue-meanie [test_respository] % git push
Username for 'https://github.com': demitri
Password for 'https://demitri@github.com':
Counting objects: 9, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 720 bytes, done.
Total 8 (delta 2), reused 0 (delta 0)
To https://github.com/SciCoder/test_respository.git
   e5aa6ac..2377659  master -> master
```

Getting Changes from the Remote Repository



This command will retrieve changes from the remote repository to your local repository:

```
git fetch
```

This does not update the files in your working directory. To do that, follow the `fetch` with:

```
git merge
```

Or, if you actually have better things to do with your time, use this command:

```
git pull
```

$\boxed{\texttt{git pull}} = \boxed{\begin{array}{l}\texttt{git fetch}\\ \texttt{git merge}\end{array}}$

Deleting Files in Git

If a file is in your working directory and you have never added it (with `git add`), you can just delete it.

If your file is being tracked (i.e. you did a `git add` at least once):

```
git rm filename
```

If your file is being tracked and you want git to stop tracking it (remove it from the repository), but to not actually delete it:

```
git rm --cached filename
```

Remember to git commit after any of these commands!

The .gitconfig File

- File located in your home directory that contains settings that git will use.
- These settings apply to all of your repositories. You can override them in the `.git/config` file in each checked out repository.

Example .gitconfig file

```
[user]
    name = Demitri Muna
    email = demitri.muna@gmail.com
[core]
    editor = vim
    excludesfile = /Users/demitri/.gitignore_global
```

The .gitignore File

- There are some files you know beforehand you'll *never* want to commit, e.g. *.pyc files, .DS_Store files on Mac.
- The `.gitignore` file tells git which files to ignore. Can be done on a global basis (all your repositories) or on an individual basis by placing a `.gitignore` file in the top level of your repository (or any subdirectory).
- Better to be conservative with these; files you would ignore in a LaTeX directory might not be the same as in others.

Example `.gitignore_global` file
(location pointed to in `$HOME/.gitconfig` file.

```
# comment (ignored)
.DS_Store
*.pyc
*.so
*.dll
*.exe
*.o
*.swp
*~
Icon?
```

A collection of useful configurations: https://github.com/github/gitignore

Hosting Your Repository

- Free services available to host your repository for free.
- Repositories can be shared between users (read/write access).
- GitHub (http://github.com) very popular (but only available with git).
- GitHub only allows public repositories for free. Accounts are available for free for academics to host private repositories (go to http://github.com/edu to request one).
- Bitbucket (http://bitbucket.org/) supports git, mercurial, unlimited private or public repositories for up to five users. Automatically upgrades your account if you register with an academic address.
- All provide a web interface to view/download/edit your files.

SciCoder Repository

SVN repository for this week (read only):

```
https://github.com/SciCoder/SciCoder2014.git
```

Go ahead and check this out now:

```
% git clone https://github.com/SciCoder/SciCoder2014.git
```

Files, code, data, and updates will be distributed to you through this repository.

Have GitHub remember your password:

https://help.github.com/articles/set-up-git#password-caching

Subversion

Subversion was the most popular version control system a few years ago, but has largely been supplanted by Git, and to a lesser extent, Mercurial. It can still be found (e.g. SDSS), but most people are using Git now.

The following slides shows you how to use Subversion if you need it.

Setting Up Your SVN Repository

Instructions may vary slightly between servers.

1. Log into server where your repository will be held. Create a new repository. You'll probably set up more than one repository, so create a directory to hold them all, e.g. `~/Repositories`. (Your sys admin might have provided a specific directory.) Come up with a name for your repository, e.g. `muna_code`.

```
% cd ~/Repositories
% svnadmin create muna_code
```

2. On your local machine, create the directory that will house your code. You can start with it empty, or else place code that you want to start with there now. Don't agonize about the structure of the files – you can always change it. Go to the directory *above* this one and import it into your repository.

```
% svn import <project_directory_name> ssh+svn://user@host/svn_repositories
```

local directory name, will import all files inside

authentication via ssh

your details

path to the repository on the server

Setting Up Your SVN Repository

3. Before completing, svn will prompt you to write some comments. Write a short description of the repository or else "Initial import."

4. Rename the folder you just imported, e.g.

```
% mv project_folder project_folder.bkup
```

5. Check out the project (i.e. pull a copy from the server). This copy will contain the details of the repository.

same as used in initial import

```
% svn co ssh+svn://user@host/svn_repositories
```

6. cd into the newly checked out directory and verify the files are there. Can also verify server info with:

```
% svn info
```

7. It's ready to use!

Connection URLs

There are three main ways to connect to an SVN server:

1. Local file system : `file:///path/to/repository/repo`
    - The client is on the same computer as the repository.
    - The host name must be "localhost" (assumed if left out).

2. http/https URL : `http[s]://host/path/to/repository/repo`
    - Access is through WebDAV (must be set up in Apache server).

3. Connect to `svnserve` server : `svn://host/path/to/repository/repo`
    - An svn server application runs on the host, connects to that.

4. Tunnel svn over SSH : `svn+ssh://host/path/to/repository/repo`
    - Connect to svn server over ssh.
    - Authentication is handled through the ssh credentials, not svnserve.

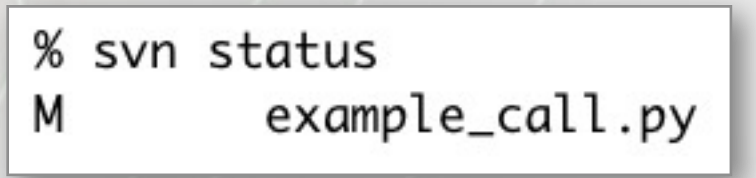
Option 4 is the strongly recommended method.

Checking in Changes

- Create a new file in the directory.
- Edit an existing file.

```
% svn status
```

status flag



```
% svn status
M       example_call.py
```

Shows a list of files that have changed since the last check-in. This only shows *local* changes. This list won't include the new file – you have to make svn aware of it (i.e. adding it to version control).

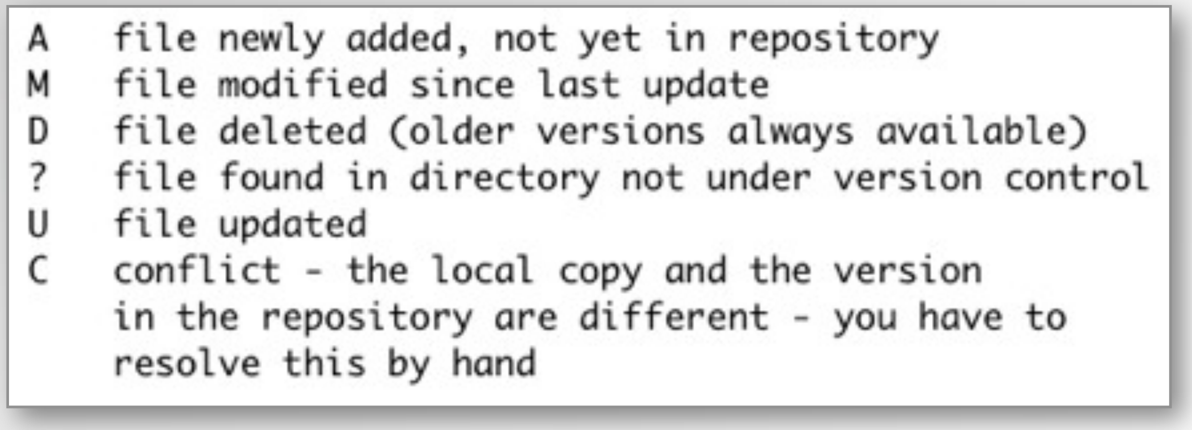
```
% svn add newFile.txt
```

To push the changes to the repository:

svn always requires a comment on commit

```
% svn commit -m "brief explanation of changes."
```

Most common status flags



```
A    file newly added, not yet in repository
M    file modified since last update
D    file deleted (older versions always available)
?    file found in directory not under version control
U    file updated
C    conflict - the local copy and the version
     in the repository are different - you have to
     resolve this by hand
```

Renaming/Moving/Deleting Files

- Don't rename or delete any files under version control – svn loses track of them and freaks out. Don't make svn freak out.
- Don't move files in the directory as you normally would.
- Instead, let svn handle it:

```
% svn mv oldName.txt newName.txt
% svn rm oldName.txt
% svn mv oldDir newDir
```

- When you move/rename files, `svn status` will show that the old file is being deleted (D) and the new version is being added (A).

Comparing Versions

- Each check-in increments the version number. You can compare a file from any version with any other version.
- To compare a locally modified file with the last saved version:

```
% svn diff fileToCompare.txt
```

- Comparing two arbitrary versions requires knowing the version number. There are command line tools, but this is more easily done with a GUI program (e.g. Cornerstone on the Mac).
- To see what differences there are between your local directory and what is on the server:

```
% svn status -u
```

Ignoring Files

There are some files that you want svn to always ignore. To ignore certain files in all of your repositories, edit the file:

```
~/.subversion/config
```

Look for the line starting with "global-ignores". This is what I have set:

emacs backup files

Macs make tons of these

```
global-ignores = *~ .*~ .DS_Store *.pyc *.pyo *.egg-
info *.o *.lo *.la *.so *.so.[0-9]* *.a *(Autosaved)*
```

typical intermediate C/C++ compiler objects - libraries won't work across different architectures or OS's; best to rebuild

To ignore a specific file in a directory (e.g. you want to place data in your code directory but not keep it in the repository):

```
% svn propedit svn:ignore ./specific_directory
```

Your preferred text editor will open, and you just add the files, one per line, that you want svn to ignore.

trunk

When you create a new code repository, the first thing you should do is:

```
% svn mkdir trunk tags branches
```

(Don't do this for repositories where you want to just keep track of random files.)

Place all of your files in `trunk`; this is your working area. For example, an initial import would look like this:

all on one line

```
% svn import code_directory svn+ssh://user@host/path/to/repo/
repository_name/trunk/code_directory
```

Tagging

When you have stable and working code that you do a particular analysis with, it's useful to take a snapshot that you can refer to. We label this snapshot with a "tag." Do this instead of making a copy outside of the repository if you want to keep a certain version.

To create a new tag:

your tag name

```
% svn copy svn+ssh://.../repository_name/trunk svn+ssh://.../
repository_name/tags/v1.0 -m "tagged v1.0"
```

The tag name can be anything you want. You can then check out a specific version by tag name any time, even after you've made dozens of new changes. The latest version of your code will still be under `trunk`.

Tags are cheap to make since they point to files already in the repository. Making 100 tags won't increase your repository 100x.

Branching

Let’s say your code is working nicely, but you have a radical idea that you want to try. It will involve changing lots of files in your code, but you’re afraid that if your idea doesn’t work, you’ll have to spend a lot of time changing things back.

Or, maybe your next version of the code involves many changes, but you need the current version to keep working until you’re done.

The solution is branching – this allows you to make radical changes, while keeping the current working version intact.

This is left as an exercise for the reader (use the SciCoder forum or your local system administrator for questions).

Web Interfaces

- WebSVN - http://www.websvn.info
  - Looks most promising
- SVN::Web - http://freshmeat.net/projects/svnweb/
- Trac - http://trac.edgewall.org/
  - Includes wiki, ticket system
  - Pain to install!
  - Great for a project with many people, a little too heavy for a single user.
- Commercial Services
  - These offer free accounts, varying by interface, number of users, disk space (~100MB-2GB)
  - http://github.com/
  - https://bitbucket.org/
  - http://www.assembla.com/
  - http://beanstalkapp.com/

Create Your Repository

On remote server:

```
% svnadmin create /path/to/newrepos
```

Initial import from your computer:

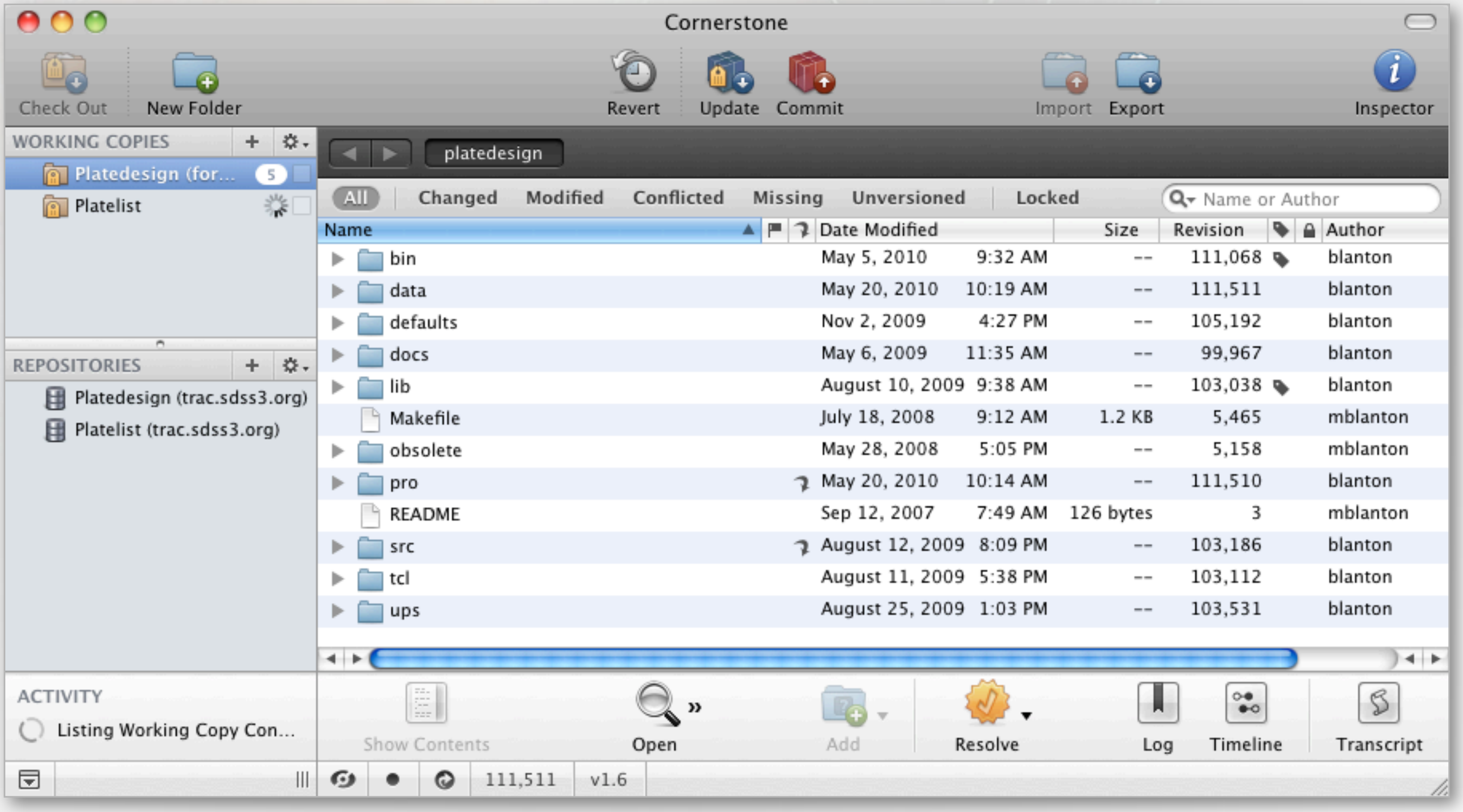
"Project" is your code or project directory

```
% mkdir new_repos
% mkdir new_repos/trunk new_repos/branches new_repos/tags
% svn import new_repos svn+ssh://user@host/newrepos
```

Rename project directory, check out from server:

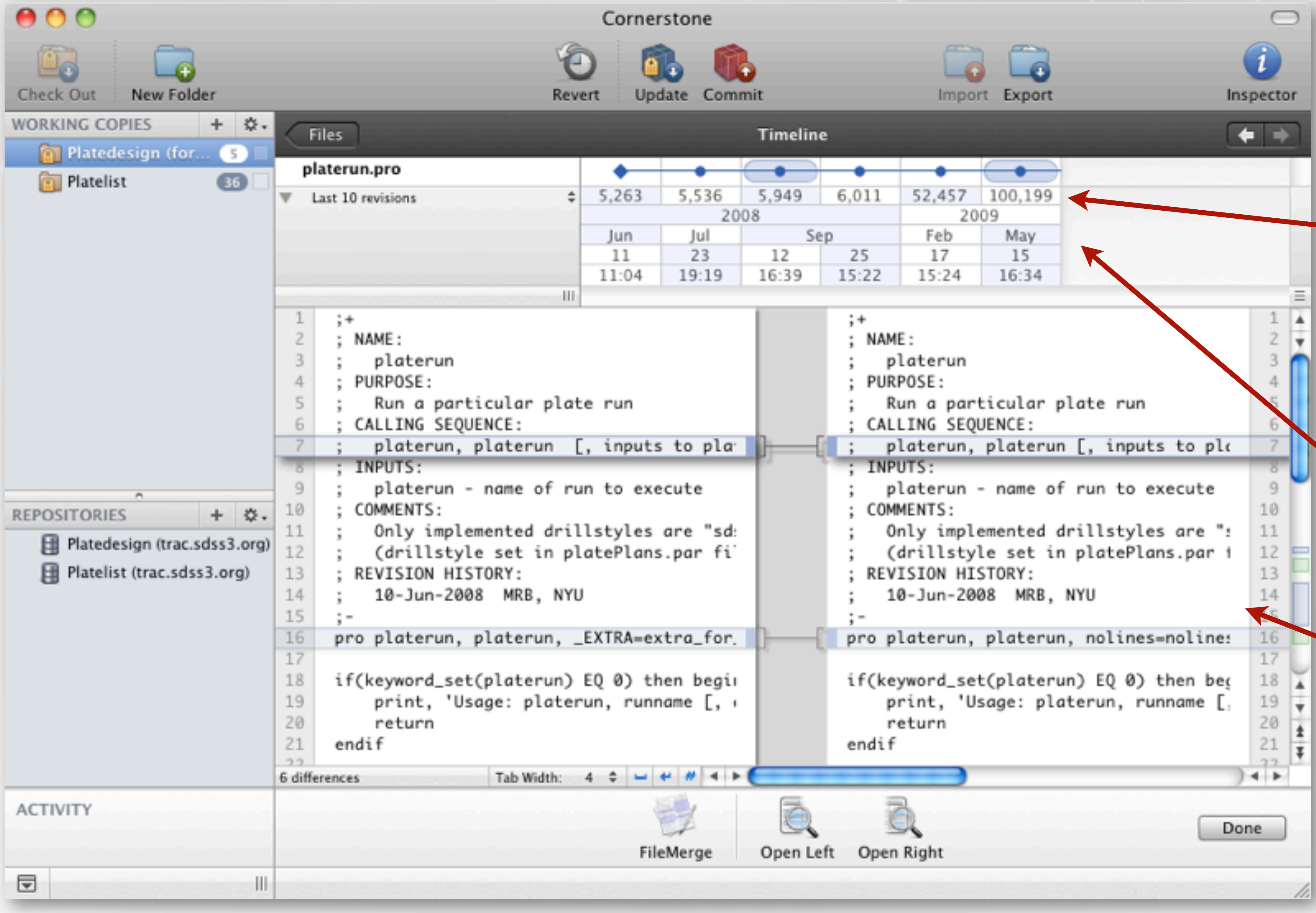
```
% svn co svn+ssh://user@host/newrepos
```

Cornerstone



Name	Date Modified		Size	Revision	Author
bin	May 5, 2010	9:32 AM	--	111,068	blanton
data	May 20, 2010	10:19 AM	--	111,511	blanton
defaults	Nov 2, 2009	4:27 PM	--	105,192	blanton
docs	May 6, 2009	11:35 AM	--	99,967	blanton
lib	August 10, 2009	9:38 AM	--	103,038	blanton
Makefile	July 18, 2008	9:12 AM	1.2 KB	5,465	mblanton
obsolete	May 28, 2008	5:05 PM	--	5,158	mblanton
pro	May 20, 2010	10:14 AM	--	111,510	blanton
README	Sep 12, 2007	7:49 AM	126 bytes	3	mblanton
src	August 12, 2009	8:09 PM	--	103,186	blanton
tcl	August 11, 2009	5:38 PM	--	103,112	blanton
ups	August 25, 2009	1:03 PM	--	103,531	blanton

Cornerstone



5,263	5,536	5,949	6,011	52,457	100,199
2008				2009	
Jun	Jul	Sep		Feb	May
11	23	12	25	17	15
11:04	19:19	16:39	15:22	15:24	16:34

```
;+
; NAME:
;   platerun
; PURPOSE:
;   Run a particular plate run
; CALLING SEQUENCE:
;   platerun, platerun  [, inputs to pla
; INPUTS:
;   platerun - name of run to execute
; COMMENTS:
;   Only implemented drillstyles are "sd:
;   (drillstyle set in platePlans.par fi
; REVISION HISTORY:
;   10-Jun-2008  MRB, NYU
;-
pro platerun, platerun, _EXTRA=extra_for.

if(keyword_set(platerun) EQ 0) then begi
   print, 'Usage: platerun, runname [, '
   return
endif
```

```
;+
; NAME:
;   platerun
; PURPOSE:
;   Run a particular plate run
; CALLING SEQUENCE:
;   platerun, platerun [, inputs to pl
; INPUTS:
;   platerun - name of run to execute
; COMMENTS:
;   Only implemented drillstyles are ":
;   (drillstyle set in platePlans.par i
; REVISION HISTORY:
;   10-Jun-2008  MRB, NYU
;-
pro platerun, platerun, nolines=noline:

if(keyword_set(platerun) EQ 0) then beg
   print, 'Usage: platerun, runname [.
   return
endif
```

>100K revisions, but only six for this file – easily see which ones

visual timeline

visually compare any two versions

Other SVN GUIs

- **Linux**
  - KDESvn http://kdesvn.alwins-world.de/
- **Mac**
  - Cornerstone http://zennaware.com
  - Xcode (part of Apple Developer Tools)
  - Versions http://versionsapp.com/
- **Windows**
  - TortoiseSVN http://tortoisesvn.tigris.org/ - Windows shell extension
- **Multi-platform (Mac/Linux/Windows)**
  - RapidSVN http://rapidsvn.tigris.org/
  - eSVN http://zoneit.free.fr/esvn/
  - Eclipse (has an SVN module called Subclipse) - http://subclipse.tigris.org/

Command Line vs GUI

- Be familiar with the command line – it’s actually simple to use.
- The GUI tools are good for viewing older versions of files – the command line is much more awkward for this.
- GUI tools are useful for keeping track of all the repositories you have access to – you don’t even have to keep a checked out version on your hard drive.

Further Documentation

Version Control with
Subversion

Free book located here (both as html and pdf):

http://svnbook.red-bean.com/

(and of course Google is your friend)

A simple reference help is also available on the command line via:

```
% svn help
```

or for more specific help on a given command, e.g. 'commit':

```
% svn help commit
```