Version Control With Git & GitHub

Demitri Muna
OSU

8 June 2015

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!)
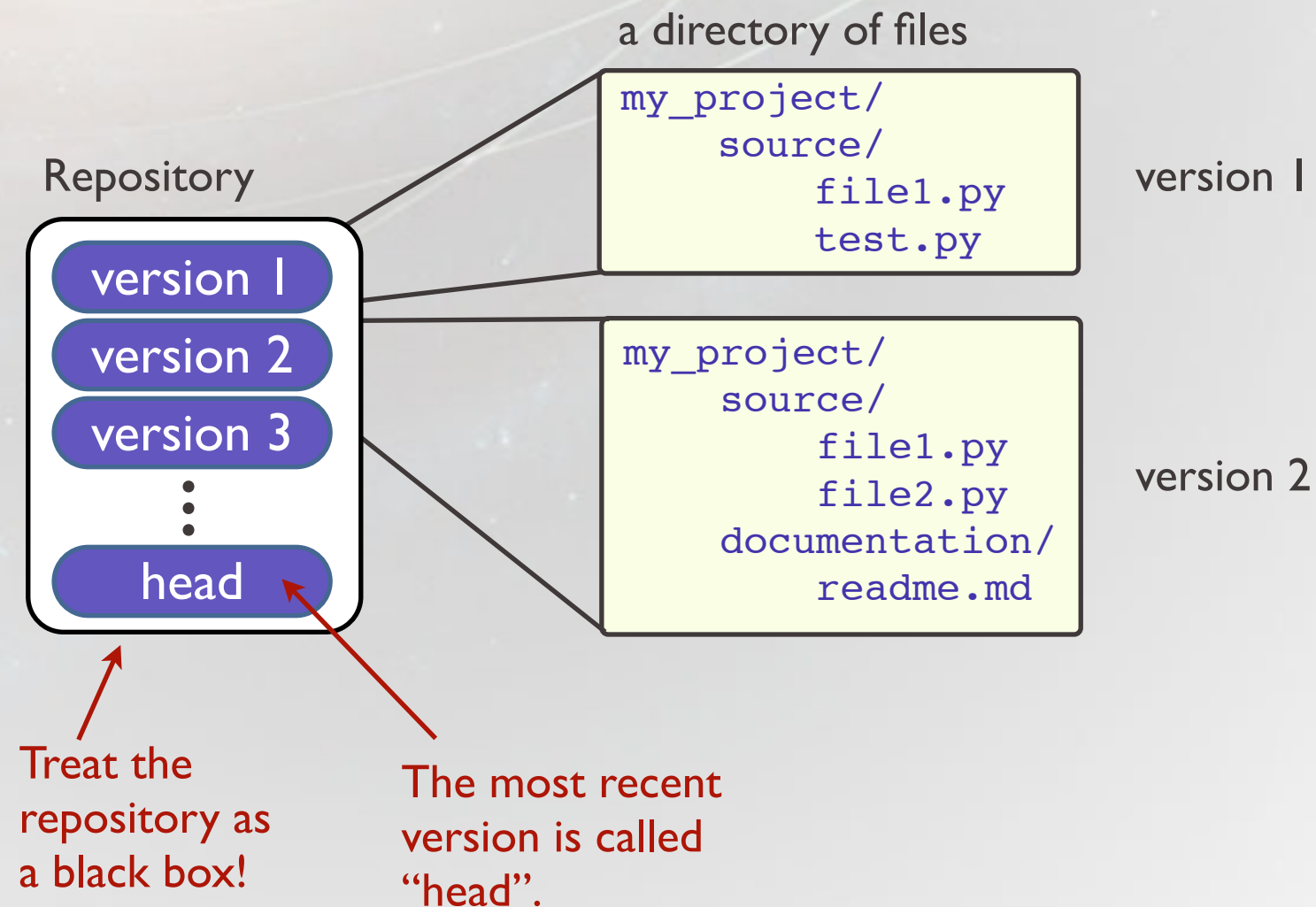
File Versioning Tools

- There are software tools that provide all of the above.

- Easy to integrate with your daily habits with minimal effort.

- Most popular system now is Git.

- Older systems:

  - Mercurial – Python-based alternative to Git

  - Subversion – still in use, being phased out

  - 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).
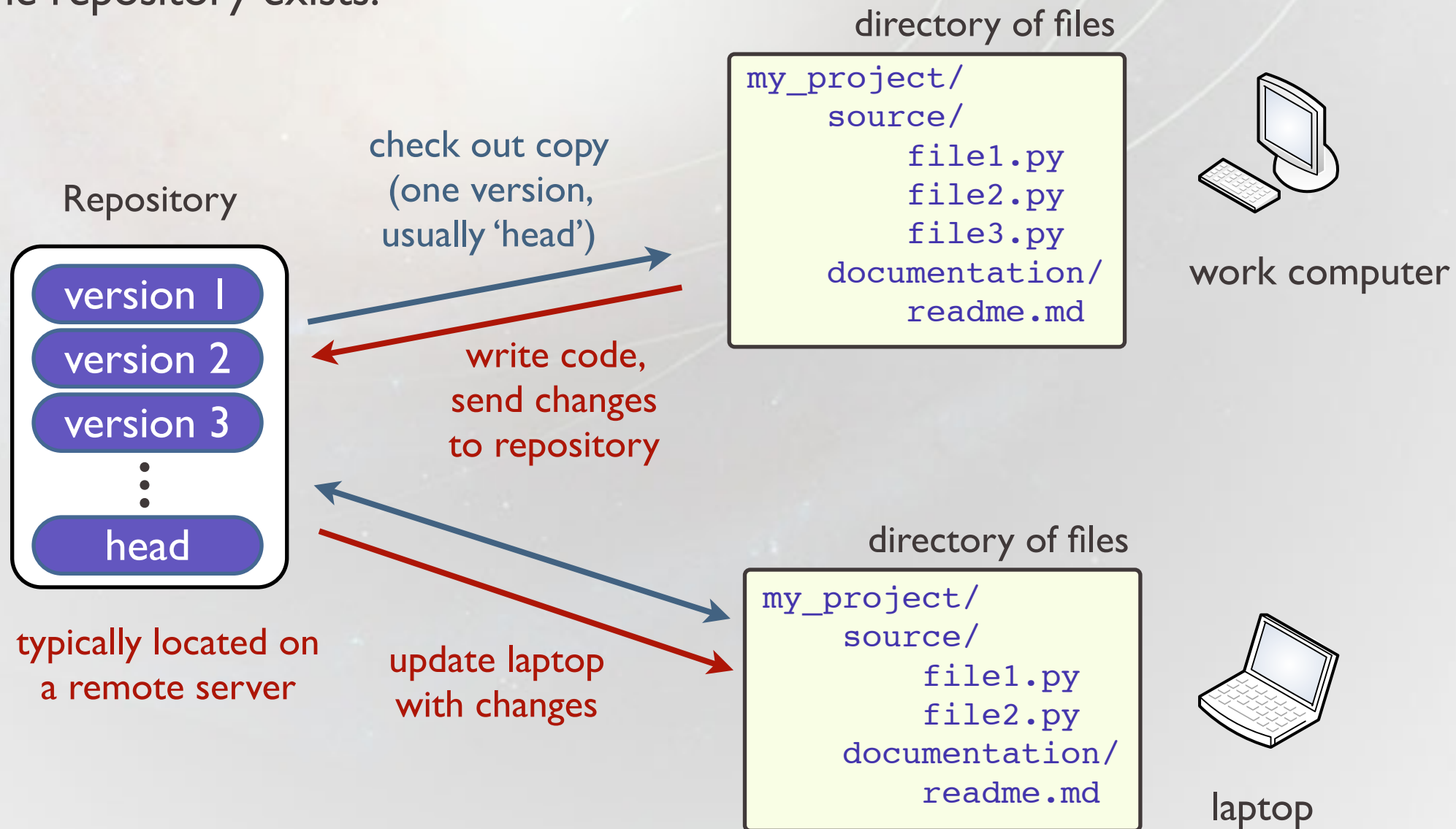
The Repository

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.

a directory of files

```
my_project/
    source/
        file1.py
        test.py
```
version 1

Repository

version 1

version 2

version 3

...

head

```
my_project/
    source/
        file1.py
        file2.py
    documentation/
        readme.md
```
version 2

Treat the repository as a black box!

The most recent version is called "head".

Kinds of Repositories

Central Repository

Only one repository exists.

directory of files

```
my_project/
    source/
        file1.py
        file2.py
        file3.py
    documentation/
        readme.md
```

work computer

Repository

check out copy
(one version,
usually 'head')

version 1

version 2

version 3
⋮
head

write code,
send changes
to repository

typically located on
a remote server

update laptop
with changes

directory of files

```
my_project/
    source/
        file1.py
        file2.py
    documentation/
        readme.md
```

laptop

Examples: SVN, CVS

Kinds of Repositories

Distributed Repository

Everyone has a full copy of the repository.

Repository

version 1
version 2
version 3
⋮
head

can be located on a remote server

this is a "bare" repository in Git: no working directory

Examples: Mercurial, Git

version 1
version 2
version 3
⋮
head

directory of files

```
my_project/
    source/
        file1.py
        file2.py
        file3.py
    documentation/
        readme.md
```

work computer

version 1
version 2
version 3
⋮
head

directory of files

```
my_project/
    source/
        file1.py
        file2.py
    documentation/
        readme.md
```

laptop

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

Creating a new repository:

```
                    new_project — bash — 80×24
Last login: Fri Jun  5 14:23:40 on ttys002
MacBook-Air [~] % mkdir new_project
MacBook-Air [~] % cd new_project/
MacBook-Air [~/new_project] % git init
Initialized empty Git repository in /Users/demitri/new_project/.git/
MacBook-Air [~/new_project] % ls -la
total 0
drwxr-xr-x    3 demitri  staff   102 Jun  5 14:24 ./
drwxr-xr-x@ 179 demitri  staff  6086 Jun  5 14:23 ../
drwxr-xr-x   10 demitri  staff   340 Jun  5 14:24 .git/
MacBook-Air [~/new_project] %
```

Note the new ".git" directory.

Similarly, you can "git init" in any existing directory to convert it to a repository. There are tools to then upload it to GitHub (links below). It's easier to just create a new repository on GitHub, clone it, then copy files into the resulting directory.

```
https://import.github.com/new
https://help.github.com/articles/importing-a-git-repository-using-the-command-line/
```

Creating a New Repository on GitHub

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

Create new repository.
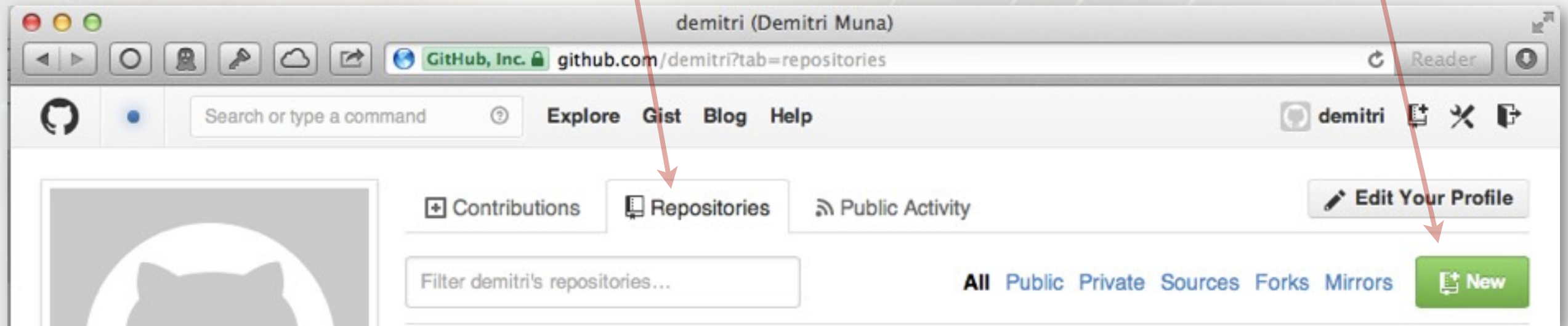
Fill in a name & description.

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

Licenses are a whole other topic…

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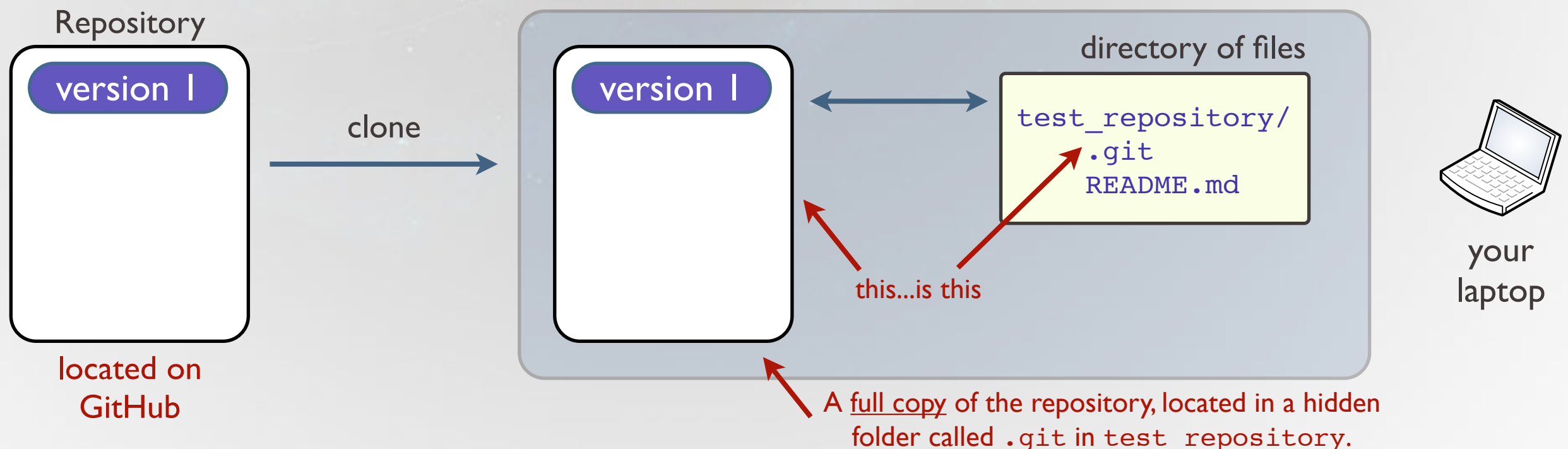
On the lower right on the next page, copy the "clone URL".

HTTPS clone URL
https://github.co

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").

Repository

version 1

clone

version 1

directory of files

test_repository/
   .git
   README.md

this...is this

your laptop

located on GitHub

A <u>full copy</u> 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 name is assigned locally – the remote still doesn't have a name.) 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 and 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."

place into
staging area

place into repository
on your local machine

directory of files

staging area

version 1

test_repository/
    .git
    README.md
    newfile.txt

`git add`

newfile.txt

`git commit`

your laptop

Your local copy of the repository (i.e. `.git` folder)
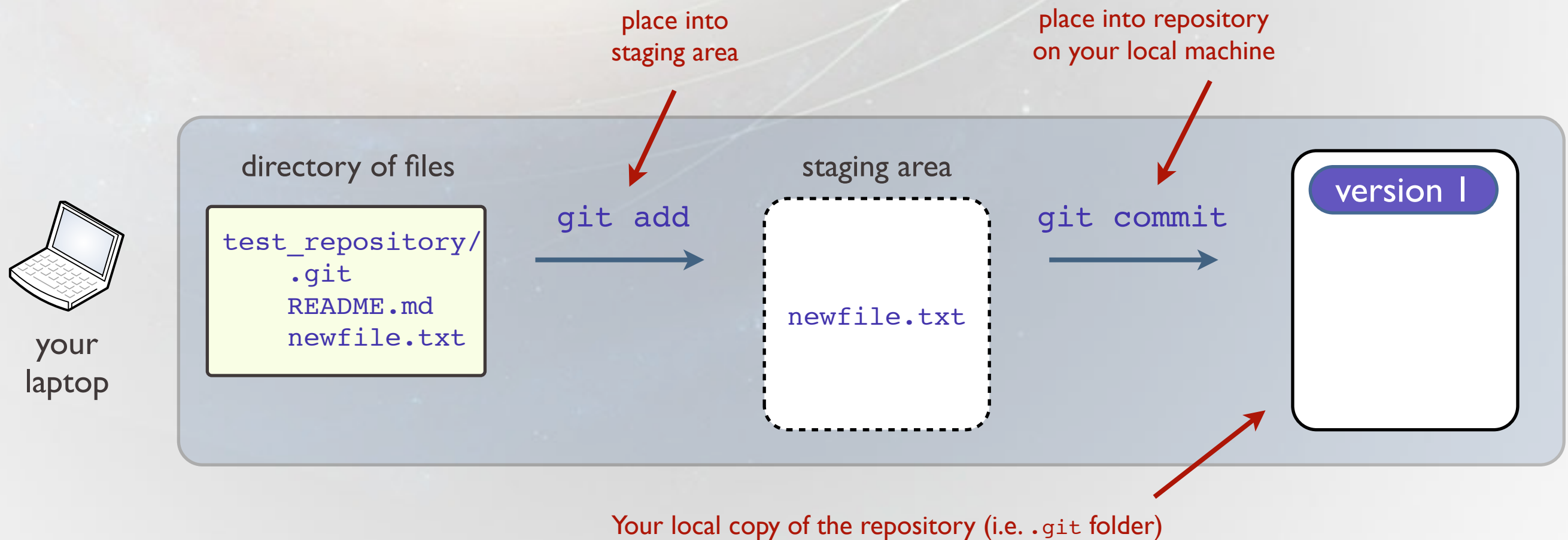
The `git commit` command will then actually save the file into the repository.

Committing Files

You need to be roughly aware staging happens. I do not recommend you use this feature. When you are 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:        ← i.e. in staging area limbo
#    (use "git reset HEAD <file>..." to unstage)
#
#        new file:   newfile.txt    ← list of files that will be committed
#
blue-meanie [test_respository] % git commit -m "First commit."    ← A description of the changes must be specified with every commit, easiest with the "-m" flag.
[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.    ← Yeah, that's helpful. We'll get to this.
#
nothing to commit (working directory clean)    ← This means that the latest version in the repository matches the files in the working directory.
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

If you try to commit without staging, you'll get this error:

```
blue-meanie [~/test_repository] % git commit
On branch master
Changes not staged for commit:
    modified:    newfile.txt

no changes added to commit
blue-meanie [~/test_repository] %
```

Not terribly user-friendly output...

If you commit without the −m (message) flag, the default editor on your shell will open to prompt you to enter a message describing the changes.

Avoid committing something before checking to see if there is a remote update. The solution is painful.

Committing Files

Can we just pretend the whole staging thing 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 -am "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.

time

Version 3; active development now, more changes made.

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

start here !

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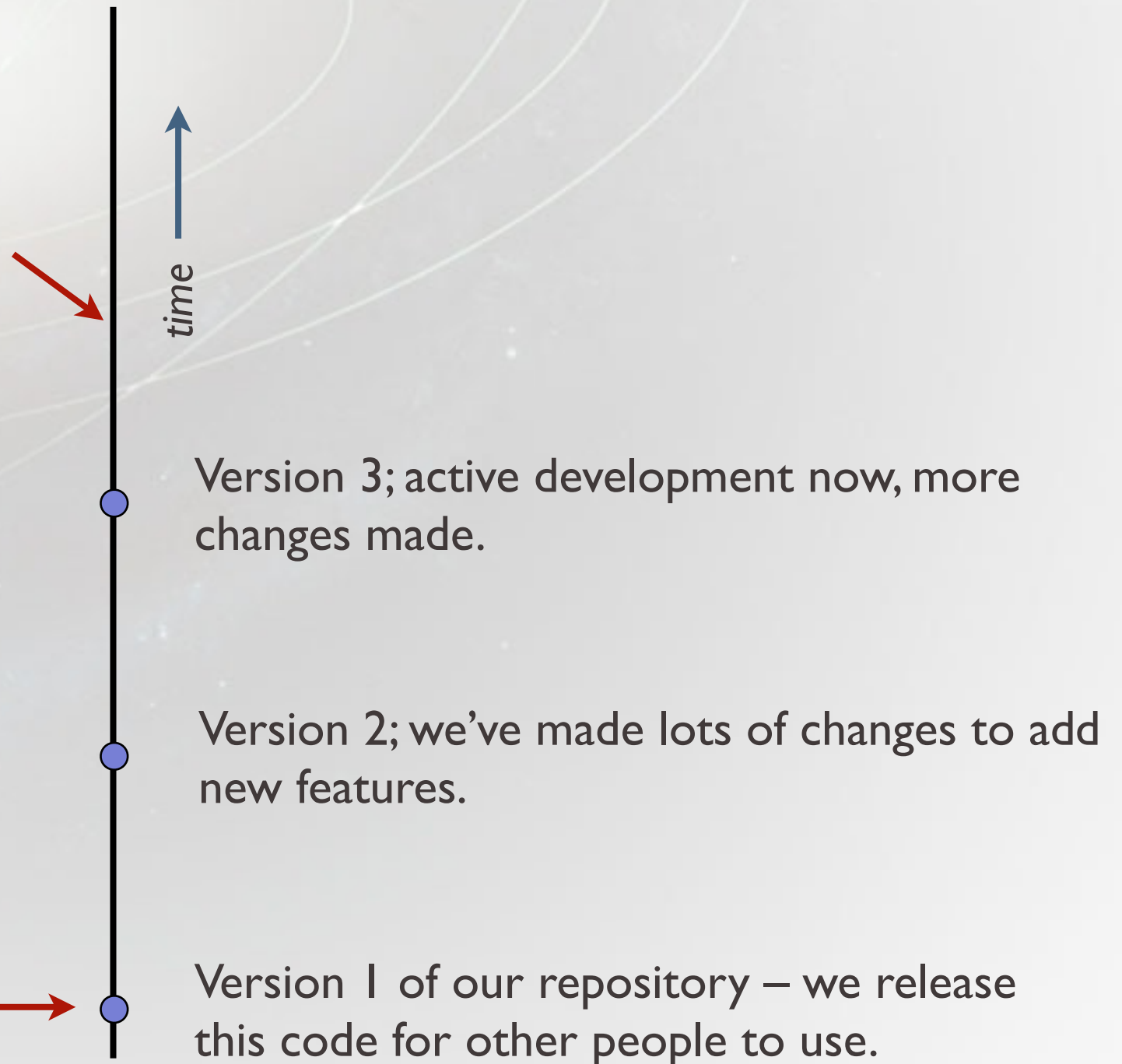
Branching

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.

dev

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

master

each branch has its own name
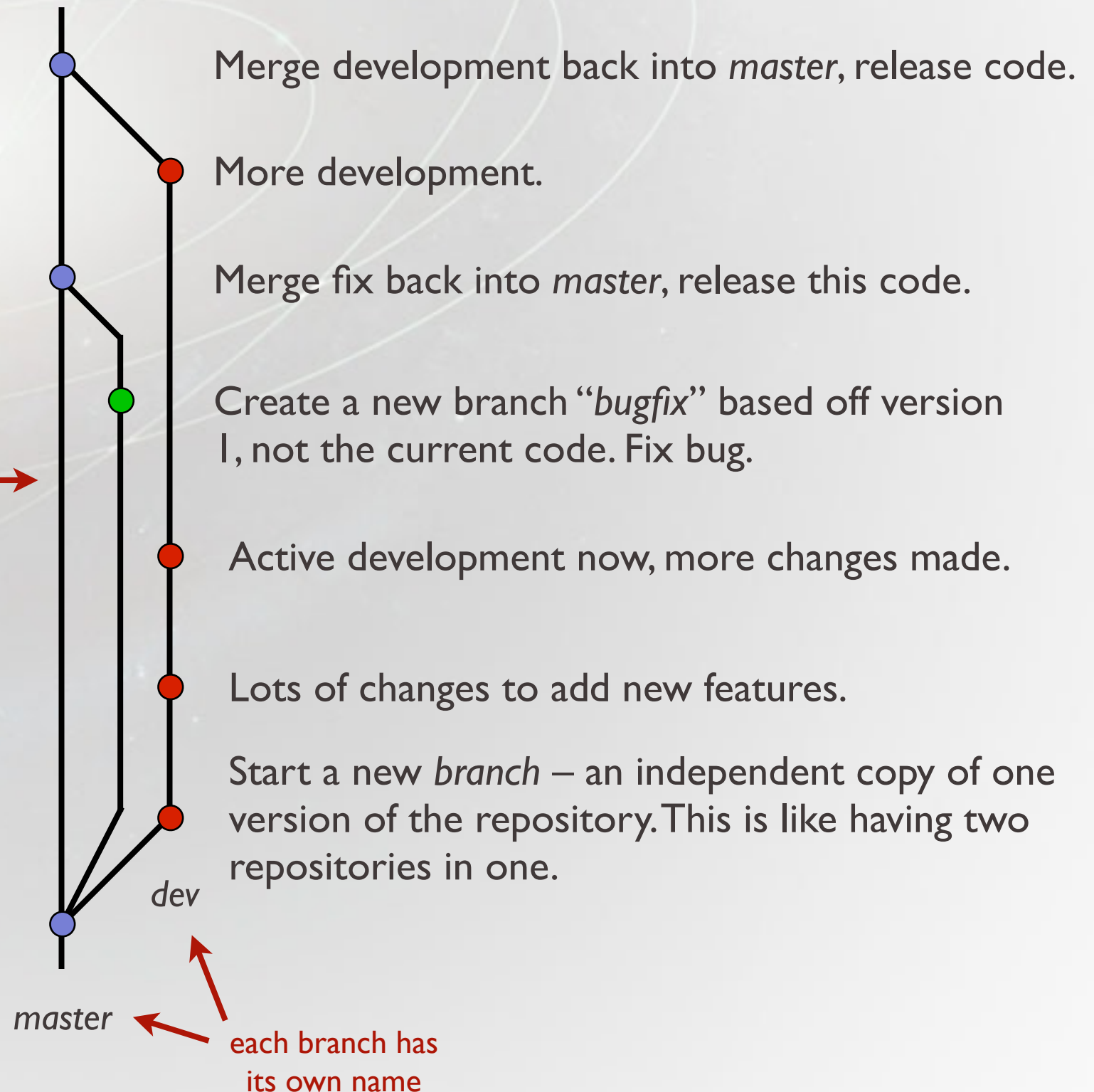
Branching

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

list all branches

first, default branch is automatically named *master*

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

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 →

Local repository

> version 1
> version 2
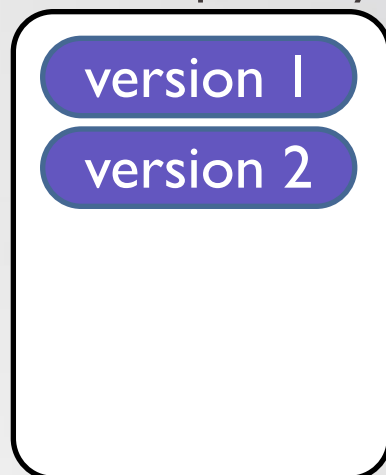
push →

Remote repository

> version 1
> version 2
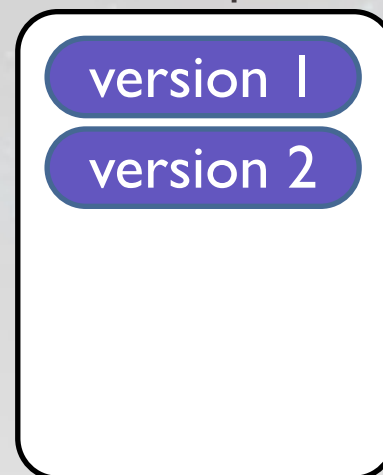
origin
(e.g. on GitHub)

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

Remote repository
`origin`

| version 1 |
| version 2 |

fetch →

| version 1 |

merge →

directory of files

```
test_repository/
    .git
    README.md
```

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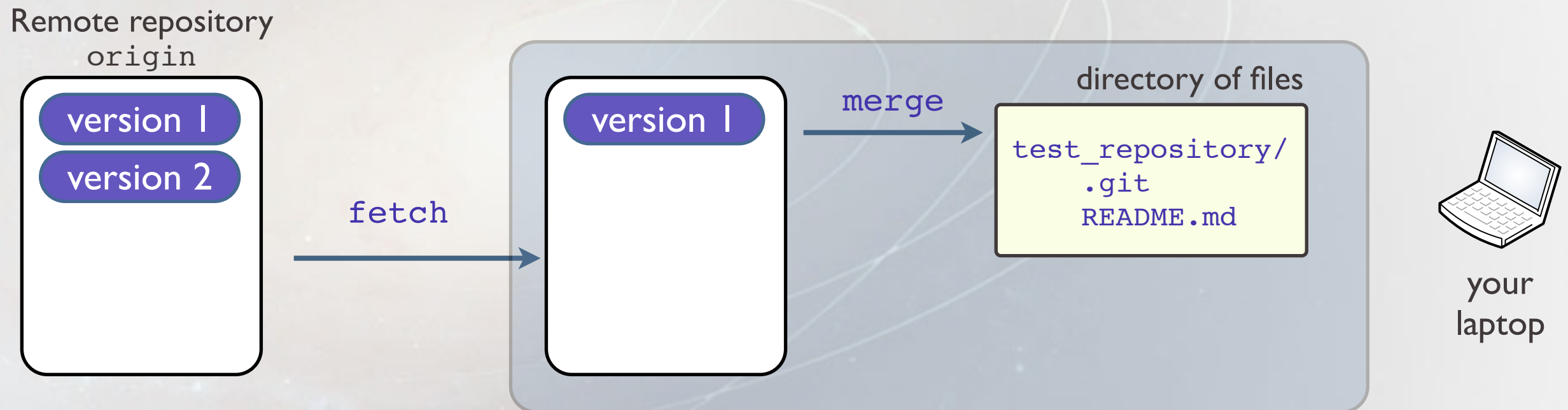
This command will retrieve changes from the remote repository to your local repository:

`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`

| `git pull` | = | `git fetch` `git merge` |

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.

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

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

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

Hosting Your Repository

Most commonly used, particularly for open source projects.

Open source repos are free; private repos require paid account. Go to http://github.com/edu to request free upgrade for academic users. Note the academic upgrade expires after two years; your private repos will be quietly deleted!!

Unlimited private and public repos. Private repos limited to sharing between five users. Upgraded accounts (automatic with academic address) removes this limit.

Can host Git or Mercurial repositories.

Better for hosting a repo that you don't want to be public (private code, writing a paper with a few people).

It's very common to have an account with both services. These sites provide web access, an "issues" system, wiki, etc. for your repositories. None of these features are built into Git or Mercurial.

SciCoder Repository

SciCoder 6 repository:

```
https://demitri@bitbucket.org/demitri/scicoder-2015-
workshop.git
```

Go ahead and check this out now from the command line:

```
% git clone https://demitri@bitbucket.org/demitri/
scicoder-2015-workshop.git
```

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

Have GitHub remember your password:

https://help.github.com/articles/caching-your-github-password-in-git/

Contributing to an Existing Repository

source repo

1.

| version 1 |
| version 2 |
| version 3 |

4.

send request to repo
owner to apply new
version to source repo
(*pull request*)

fork

2. (new, independent copy)

3. commit new version

| version 1 |
| version 2 |
| version 3 |
| version 4 |

your copy
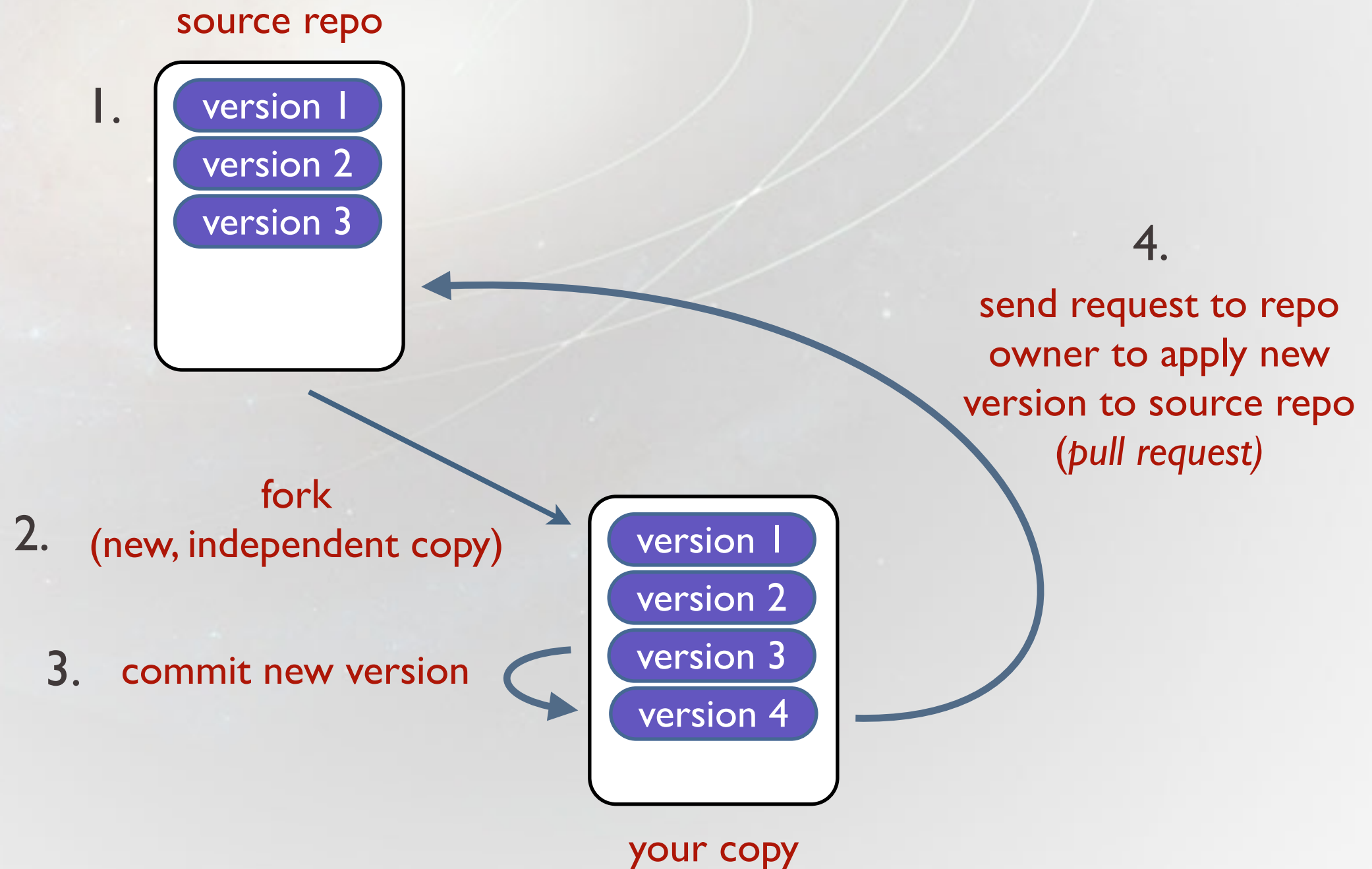
Forking on GitHub

Fork an existing repository. This makes a copy of the repository and places it under your account.

click here to create a new fork!
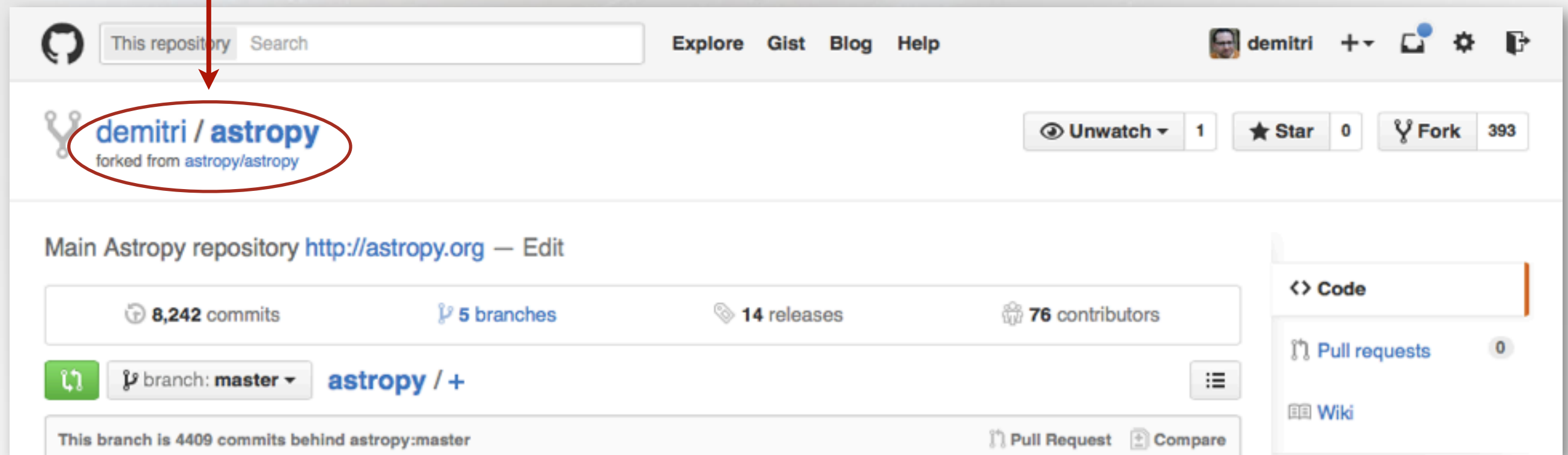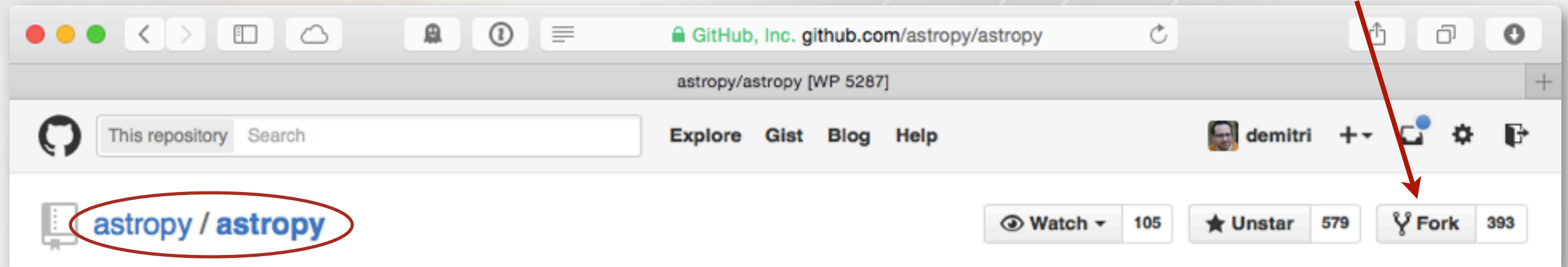
Forking on GitHub

The forked repository is a new, independent repo – your own copy. You can also form a project to use it as a starting point and then develop it in a different direction.

Once forked, clone the copy to your computer as you normally would. I recommend you give it a name that makes it obvious that it's a fork:

`git clone https://github.com/demitri/astropy.git astropy-demitri`

source
repo name

user name

You can now make changes to the files and commit them. The commits stay in your repository (fork) – they will *not* be pushed back to the original repository.

Pull Requests

A *pull request* is a request to take a particular commit that you've made on your fork and merge it back to the source repository. To create a pull request on GitHub:

click here

then here
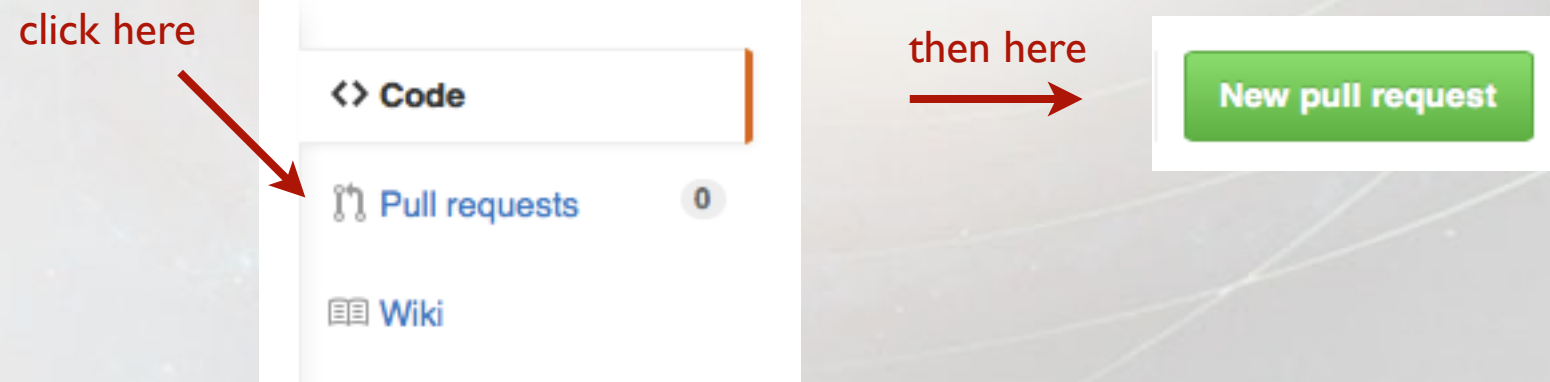
When you make a pull request, you can include comments. GitHub provides a page for each pull request where anyone can discuss the request before it gets accepted. Only the source repo owner can accept the request.

Updating a Fork

Let's say you fork a project and leave it alone for a while (e.g. Astropy). In that time, many commits have been made to the original project. Your fork is now very out of date. Before working on your fork, you want to bring it up to date.

You can do this directly on GitHub, or you can do it on the command line.

The recipe(s) are here:

http://stackoverflow.com/questions/7244321/how-to-update-github-forked-repository

git-info

Subversion (another version control system) had a convenient "svn info" command that would list a repository's source, URL, etc. Git doesn't have this, but this is a script that will produces the same results:

https://github.com/gitbits/git-info/

To use, just put the script on your $PATH.

Graphical Clients

You should (more or less) know how to use Git on the command line, but a graphical tool is much easier to use. Git can get much more complicated beyond what has been presented here, and a GUI can help manage this.

GitHub has their own graphical client. I don't recommend it; it's not very good, and it only works with repositories on GitHub.

SourceTree (Mac/Windows)

add an already cloned repo or pull one straight from a URL

organize the many repositories you use/follow

see what repos have updates at a glance

list of all your Git or Mercurial repositories on your computer

SourceTree (Mac/Windows)

SourceTree (Mac/Windows)

Linux GUI Tools for Git

I don't use Linux, so you'll have to explore this area on your own!

Here are some places to start:

https://apps.ubuntu.com/cat/applications/gitg/

http://unix.stackexchange.com/questions/144100/is-there-a-usable-gui-front-end-to-git-on-linux