

PRACTICAL GIT FOR SCIENTIFIC CODING:

The Bare Essentials for Optimizing your Research Experience

Rich Pang
Computational Neuroscience Center, University of Washington,
[v0.1 2018-08-14]
[v0.2 2018-08-22]

With helpful suggestions by Matthew Farrell, Ali Weber, Stefano Recanatesi, and Eric Shea-Brown.

DISCLAIMER

These notes are for researchers writing code to investigate scientific questions. If your goal is application development they will probably offend you, for I've completely omitted the concept of branching, and you should seek guidance elsewhere. I've made this decision because in my experience branching (1) is usually overkill for small-scale research, (2) is easily misused without extensive practice and tenacity, and (3) often confuses academic Git novitiates so much that they abandon version control altogether.

I've instead focused on a few minimum Git operations that will significantly improve your scientific coding process: saving and reviewing project history, backing up and viewing code on the web, transferring and synchronizing code between computers, handling merge conflicts common to research, and structuring equal-permissions collaborations. While this is still somewhat involved, I've tried hard to present things as clearly and concisely as possible without sacrificing any key concepts or operations.

That said, Git is an elegant piece of software, and I'd recommend learning more about it if you have the time. Check out <https://git-scm.com/> or <https://www.atlassian.com/git/tutorials> to get started.

CONTENTS

[READ THIS FIRST](#)

[MOTIVATION](#)

[OBJECTIVE](#)

[OUTLINE](#)

[PRO TIPS FOR AMATEURS](#)

[USEFUL COMMANDS FOR SCIENTIFIC CODING](#)

[LINEAR GIT](#)

- [HOW TO THINK ABOUT GIT](#)
- [SAVING PROJECT HISTORY](#)
- [HOW TO EXIT VIM](#)
- [PUSHING PROJECT HISTORY TO GITHUB AND REVIEWING ONLINE](#)
- [RUNNING OLD CODE](#)
- [RESTORING PROJECT TO PREVIOUS STATE](#)
- [EXCLUDING FILES AND FILE TYPES FROM GIT](#)

[NONLINEAR GIT](#)

- [CLONING YOUR PROJECT TO A NEW COMPUTER FROM GITHUB](#)
- [BRINGING SERVER \(OR LOCAL\) REPO UP-TO-DATE WITH GITHUB](#)
- [MERGING GITHUB HISTORY INTO LOCAL HISTORY \(AND HANDLING CONFLICTS\)](#)
- [HANDLING MERGES WHEN PUSHING TO GITHUB](#)
- [GENERALIZED WORKFLOW AND BASIC COLLABORATION](#)

[APPENDIX: INSTALLING GIT](#)

[APPENDIX: POPULAR GUIs FOR GIT](#)

[APPENDIX: EXAMPLE REPOSITORY USED IN THIS TUTORIAL](#)

[APPENDIX: EXAMPLE REPOSITORY ORGANIZATION OF PROJECT WITH DATA](#)

READ THIS FIRST

BASIC TERMS

Version control system: software-based system for organizing history of coding project.

Repository: root directory of your coding project that additionally contains all your project history.

Git: popular, free, open-source version control software, created by Linus Torvalds in 2005. Other version control systems exist, such as [Mercurial](#) or [Subversion](#), but we will focus on Git.

GitHub: popular website for hosting Git repositories and for facilitating collaboration. Other popular websites for hosting Git repositories are [BitBucket](#) and [GitLab](#).

ORGANIZATION OF THIS TUTORIAL

This tutorial alternates between high-level ideas and how to actually do things using an example repository. All “how to” sections begin with “**HOW TO**”. All command-line commands are depicted in **violet**.

Further, don’t worry about memorizing anything quite yet. The table of contents has direct links to all operations we’ll cover, so you can easily refer back to them later.

OTHER NOTES

Note 1: Git is entirely local, e.g. only on your laptop. Therefore **if you delete your project repository and don’t have any backups, your code will be gone forever**. The simplest way of making backups is to use an online service like GitHub. After covering the basics of Git on a local machine, we’ll discuss this in [Pushing Project History To Github And Reviewing Online](#).

Note 2: Git is powerful but not easy. Even experts often admit they don’t really know what they’re doing. Ironically, while learning Git it’s remarkably easy to accidentally delete everything. Therefore, until you know exactly what you’re doing, **back up all code before any command sequence**.

Note 3: This tutorial uses the command line, but many excellent GUIs (graphical user interfaces) exist also. I contend, however, that learning Git through the command line is useful because (1) you often only have command line access (e.g. on a remote computing cluster), (2) it helps you understand how Git works at ground level, (3) it helps you understand what the GUIs are doing under the hood, (4) it makes it easier to copy/paste error messages into search engine to get help, and (5) it builds character. Therefore, to follow this, you should be familiar with extremely basic command line usage (specifically, commands **cd** and **ls -a**).

MOTIVATION

Scientific coding is highly iterative. How should we best track project history?

We'll use a toy computational neuroscience project as an example throughout this document.

Bad solution (look familiar?):

```
dim_reduc.py
dim_reduc_new.py
dim_reduc_new_final.py
pop_stats.py
pop_stats_v2.py
pop_stats_v2_final.py
pop_stats_v2_final_updated.py
plot.py
plot_2.py
plot_2_colors_added.py
```

Unsystematic, hard to track, easy to make mistakes or forget to update file references.

Better (but not great) solution:

```
dim_reduc_v1.py
dim_reduc_v2.py
dim_reduc_current.py
pop_stats_v1.py
pop_stats_v2.py
pop_stats_v3.py
pop_stats_current.py
plot_v1.py
plot_v2.py
plot_current.py
```

More systematic, but still can't tell which file versions go together.

Good solution:

Use *version control system*: a software system for managing project history without changing filenames.

Why use version control?

- Simplifies reviewing project history (e.g. to explore past analysis or copy code from prior version).
- Simplifies restoring project to previous state (e.g. if you break something you can't fix).
- Simplifies transferring code to and from remote servers.
- Provides software framework for collaborative coding.

OBJECTIVE

1. Learn to use Git within research context. Specifically:

- Understand basic concepts and terminology in Git-based version control.
- Learn how to perform most useful Git operations in scientific coding.

2. Provide reference for looking up basic commands as they arise throughout project progression.

Note: Git is very powerful, but we will only cover most useful functions for scientific coding.



OUTLINE

[PRO TIPS FOR AMATEURS](#)

[USEFUL GIT COMMANDS FOR SCIENTIFIC CODING](#)

[LINEAR GIT](#)

- [HOW TO THINK ABOUT GIT](#)
- [SAVING PROJECT HISTORY](#)
- [HOW TO EXIT VIM](#)
- [PUSHING PROJECT HISTORY TO GITHUB AND REVIEWING ONLINE](#)
- [RUNNING OLD CODE](#)
- [RESTORING PROJECT TO PREVIOUS STATE](#)
- [EXCLUDING FILES AND FILE TYPES FROM GIT](#)

[NONLINEAR GIT](#)

- [CLONING YOUR PROJECT TO A NEW COMPUTER FROM GITHUB](#)
- [BRINGING SERVER \(OR LOCAL\) REPO UP-TO-DATE WITH GITHUB](#)
- [MERGING GITHUB HISTORY INTO LOCAL HISTORY \(AND HANDLING CONFLICTS\)](#)
- [HANDLING MERGES WHEN PUSHING TO GITHUB](#)
- [GENERALIZED WORKFLOW AND BASIC COLLABORATION](#)

[APPENDIX: INSTALLING GIT](#)

[APPENDIX: POPULAR GUIs FOR GIT](#)

[APPENDIX: EXAMPLE REPOSITORY USED IN THIS TUTORIAL](#)

[APPENDIX: EXAMPLE REPOSITORY ORGANIZATION OF PROJECT WITH DATA](#)

PRO TIPS FOR AMATEURS

(If these don't make sense, read through tutorial first.)

Until you know exactly what you're doing, back up all code before any command sequence.

Don't checkout, push, or pull with an unclean working directory.

- If you get a message telling you to commit or "stash" changes, this is probably why.
- Either undo all changes since last commit using `git reset --hard`
- Or commit all changes using `git add -A`, and `git commit -m "<message>"`

Only check out a previous commit if you plan on re-running its code.

- If just viewing, do so online at GitHub, etc., since this is read-only copy you can't mess up.
- When done, only have to close browser window.

Before working on code on local computer, make sure local repo is up-to-date with GitHub

- Run `git pull origin master` and resolve any conflicts before continuing.
- This limits unexpected divergence of local repo and repo on GitHub.

Only version control your source code and documentation.

- Things like data, PDFs, images, videos, etc. (i.e. large files) should be ignored using `.gitignore` file (see [EXCLUDING FILES AND FILE TYPES FROM GIT](#)).

Practice using Git with example repositories.

- Feel free to clone and play with tutorial's example repository: https://github.com/rkp8000/ei_ntwk

Read terminal output to ensure Git's doing what you want it to, and if not, why not.

If you really mess up:

- First locate any backups of code (e.g. on GitHub) and make sure they're safe.
- Now in the worst case scenario you can delete your Git repository and replace it with the backup.
- However, even if you have a backup, try to solve your issue using Git commands.

Version control is not an alternative to good code organization.

Version control is not an alternative to good communication among collaborators.

USEFUL GIT COMMANDS FOR SCIENTIFIC CODING

(Read rest of tutorial for these to make sense.)

LINEAR

`git status` (check repository status)

`git log --oneline` (show concise commit history)

`git log` (show verbose commit history)

`git init` (turn directory into Git repository)

`git add -A` (stage all changes to working directory)

`git commit -m "commit message"` (create new commit from staged files)

`git remote add origin https://github.com/me/my_repo` (link local repo to GitHub repo)

`git remote -v` (list existing remote links)

`git push origin master` (attempt to upload local repo to GitHub and merge commit histories)

`git reset` (unstage files)

`git reset --hard` (undo all changes since last commit; this cannot be reversed!)

`git checkout -- dim_reduc.py` (undo all changes to specific file [dim_reduc.py] since last commit; this cannot be reversed!)

`git checkout 49a732c` (check out previous commit to re-run/tinker with)

NONLINEAR

`git status` (check working directory/push/pull/merge conflict status)

`git clone https://github.com/me/my_repo` (copy repo from GitHub onto local machine)

`git pull origin master` (copy GitHub repo onto local machine and attempt to merge commit histories)

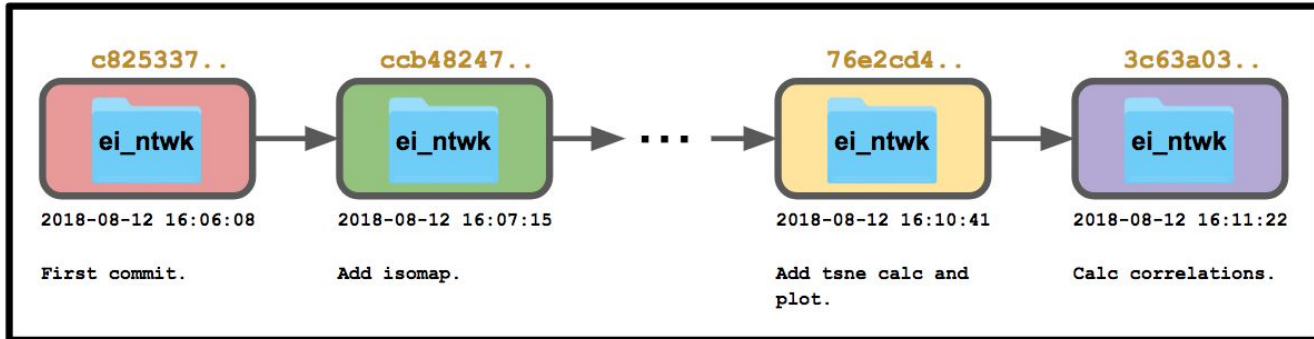
`git rm my_file.py` (stage file deletion)

`git log --oneline --graph` (show log with branching structure in terminal)

LINEAR GIT: HOW TO THINK ABOUT GIT

What is most practical way to organize project history?

Answer according to Git: save snapshots, called **commits**, of entire project (“ei_ntwk” in this case, [“excitatory-inhibitory network”, a common model in computational neuroscience]).



Each commit contains:

- ID (random string)
- complete copy of project
- timestamp copy was made
- commit message (describing changes since last commit)

Benefits of commit-based organization:

- Groups code together meant to be run together.
- Easy to view/explore past project state.
- Easy to restore past project state.
- No filename changes required.

What commit history looks like in terminal (to get a feel for this idea):

```
me@my_computer:~/projs/ei_ntwk$ git log
commit 3c63a03a397f08efe8c837050c4ae043fe2cf7eb
Author: me <me@uw.edu>
Date:   Sun Aug 12 16:11:22 2018 -0700

    Calc correlations.

commit 76e2cd46009e7f2f3c470e3154a28bb6f4b6a719
Author: me <me@uw.edu>
Date:   Sun Aug 12 16:10:41 2018 -0700

    Add tsne calc and plot.

commit bcf23210b078e9b41b3cce7853942b8bed2120e3
Author: me <me@uw.edu>
Date:   Sun Aug 12 16:09:41 2018 -0700
```

Add colored raster.

commit 49a732c2dc30778a2dc5bacaa137a286388e74d0

Author: me <me@uw.edu>

Date: Sun Aug 12 16:08:44 2018 -0700

Calc mean spike rate.

commit 9e72fec5ae4115b08142a16a6e3a36796300df78

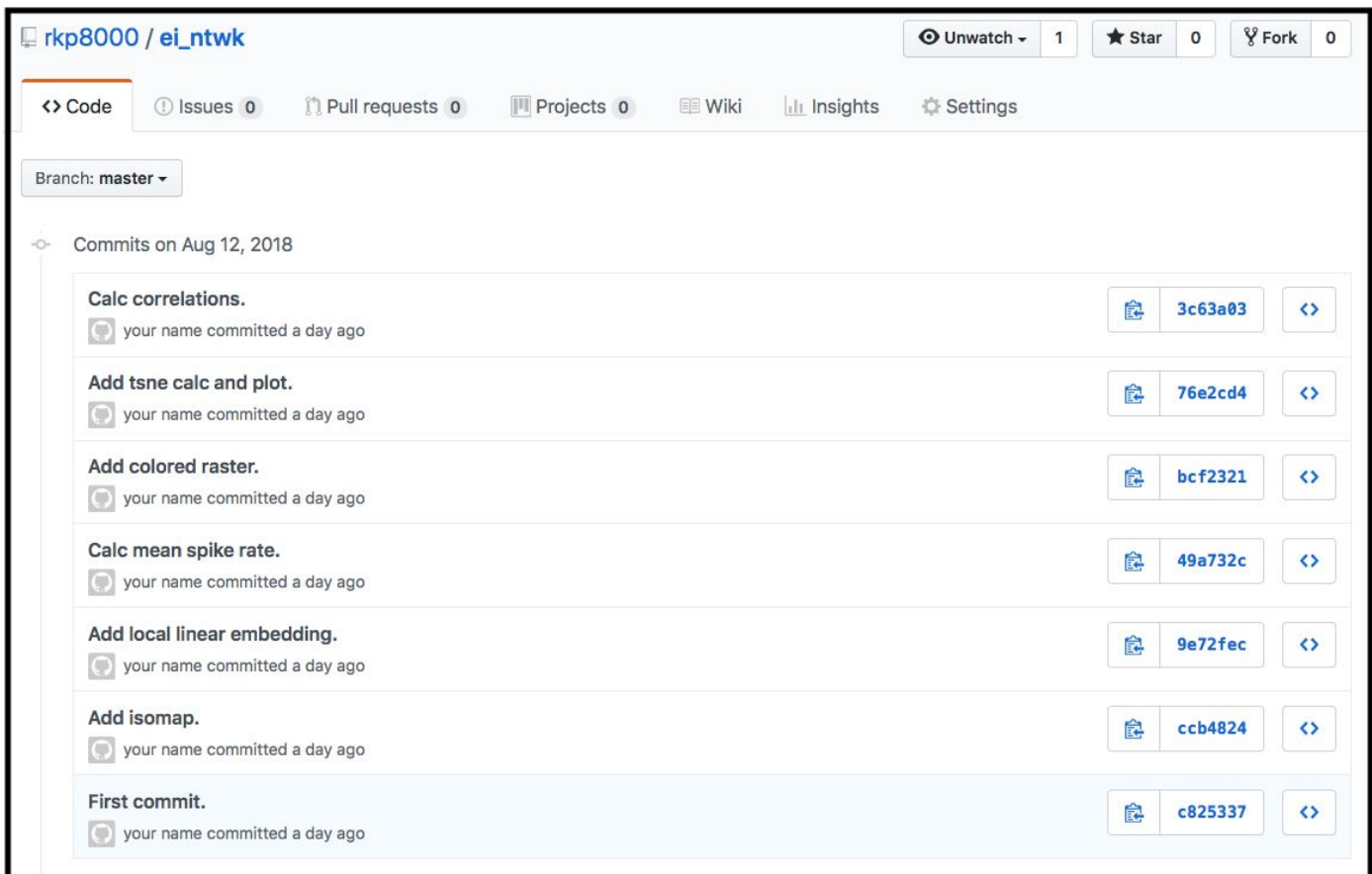
Author: me <me@uw.edu>

Date: Sun Aug 12 16:08:00 2018 -0700

Add local linear embedding.

...

What same commit history looks like on GitHub:



The screenshot shows the GitHub interface for a repository named 'rkp8000 / ei_ntwk'. The top navigation bar includes links for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Insights, and Settings. The repository has 1 Unwatch, 0 Stars, and 0 Forks. The current branch is 'master'. The commit history is displayed as a list of commits on August 12, 2018, ordered from most recent to oldest. Each commit entry includes a description, a commit icon, the commit hash, and a code icon. The commits are: 'Calc correlations.' (3c63a03), 'Add tsne calc and plot.' (76e2cd4), 'Add colored raster.' (bcf2321), 'Calc mean spike rate.' (49a732c), 'Add local linear embedding.' (9e72fec), 'Add isomap.' (ccb4824), and 'First commit.' (c825337). Each commit is attributed to 'your name' and dated 'a day ago'.

Commit Description	Commit Hash
Calc correlations.	3c63a03
Add tsne calc and plot.	76e2cd4
Add colored raster.	bcf2321
Calc mean spike rate.	49a732c
Add local linear embedding.	9e72fec
Add isomap.	ccb4824
First commit.	c825337

LINEAR GIT: SAVING PROJECT HISTORY

Will likely make up most of your interactions (as researcher) with Git.

WHERE HISTORY IS SAVED

Physically, commit history lives inside hidden folder called “.git”:

```
me@laptop:~/projs/ei_ntwk$ ls -a
.  ..  dim_reduc.py  .git  plot.py  pop_stats.py
```

Project directory together with commit history is called **repository**.

Note: feel free to look around inside .git but don't worry too much about what the different files are.

HOW TO: MAKE A PROJECT DIRECTORY A REPOSITORY (IF YOU HAVEN'T ALREADY)

Change directory to top-level project directory.

```
me@laptop:~$ cd ~/projs/ei_ntwk
```

```
me@laptop:~/projs/ei_ntwk$ ls -a
.  ..  dim_reduc.py  plot.py  pop_stats.py
```

Run `git init` to convert to git repository:

```
me@laptop:~/projs/ei_ntwk$ git init
Initialized empty Git repository in ~/projs/ei_ntwk/.git/
```

```
me@laptop:~/projs/ei_ntwk$ ls -a
.  ..  dim_reduc.py  .git  plot.py  pop_stats.py
```

Note existence of new directory .git.

To check if project is already Git repository, run `git status`. If not Git repository, will get error:

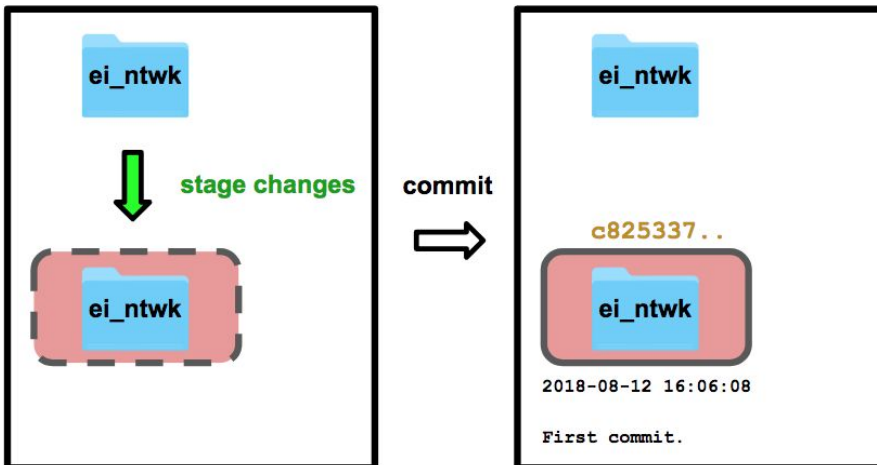
```
me@laptop:~/projs/ei_ntwk$ git status
fatal: Not a git repository (or any of the parent directories): .git
```

HOW COMMITS WORK

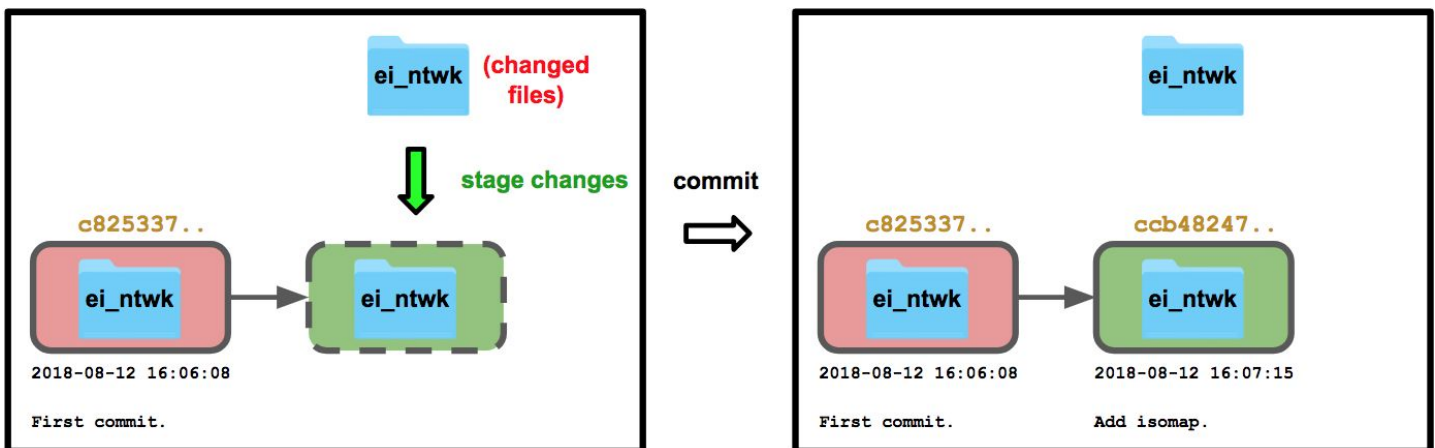
Two-part process to make new commit: stage changes & commit.

Staging tells Git what files to include in snapshot. Committing takes snapshot.

First commit:



Subsequent commits:



HOW TO: MAKE YOUR FIRST COMMIT

Check status:

```
me@laptop:~/projs/ei_ntwk$ git status
```

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

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

```
dim_reduc.py
```

```
plot.py
pop_stats.py
```

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

(For now "On branch master" just means we're groovy.)

Stage files to commit:

```
me@laptop:~/projs/ei_ntwk$ git add -A
me@laptop:~/projs/ei_ntwk$ git status
On branch master
```

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

```
new file:   dim_reduc.py
new file:   plot.py
new file:   pop_stats.py
```

(Staging a file marks it to be included in next commit/snapshot.)

(git add -A stages *all* changes since last commit, since it's also possible to only stage some changes.)

Commit staged files:

```
me@laptop:~/projs/ei_ntwk$ git commit -m "First commit."
[master (root-commit) c825337] First commit.
 3 files changed, 16 insertions(+)
 create mode 100644 dim_reduc.py
 create mode 100644 plot.py
 create mode 100644 pop_stats.py
```

(The -m tag indicates a commit message follows.)

Note 1: to stage individual files use git add <file_1> <file_2>, e.g.:

```
me@laptop:~/projs/ei_ntwk$ git add dim_reduc.py plot.py
```

This can be useful if you only want to include certain modified files in the next commit/snapshot.

Note 2: if truly your first commit, may encounter the following error message when committing:

```
me@laptop:~/projs/ei_ntwk$ git commit -m "First commit."
```

```
*** Please tell me who you are.
```

Run

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

to set your account's default identity.

Omit `--global` to set the identity only in this repository.

```
fatal: unable to auto-detect email address (got 'me@laptop.(none)')
```

To fix this, simply run the commands it tells you to:

```
me@laptop:~/projs/ei_ntwk$ git config --global user.email "me@uw.edu"
me@laptop:~/projs/ei_ntwk$ git config --global user.name "My Name"
```

This allows Git to assign authorship to commits (e.g. when collaborating).

HOW TO: MAKE SUBSEQUENT COMMITS

Edit one or more files.

E.g. we'll add isomap calculation to `dim_reduc.py`:

`dim_reduc.py`:

```
# dimensionality reduction analysis

...

# isomap
results_iso = isomap(data)
print(results_iso)
```

Check that Git has detected change:

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working directory)

```
    modified:   dim_reduc.py
```

no changes added to commit (use "git add" and/or "git commit -a")

Stage modified file:

```
me@laptop:~/projs/ei_ntwk$ git add -A
me@laptop:~/projs/ei_ntwk$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    modified:   dim_reduc.py
```

Commit:

```
me@laptop:~/projs/ei_ntwk$ git commit -m "Add isomap."
[master ccb4824] Add isomap.
 1 file changed, 4 insertions(+)
```

Note: Git treats files equally whether they live in the top-level directory or in any arbitrarily deep subdirectories.

HOW TO: REVIEW COMMIT HISTORY IN GIT

E.g. after a few more commits:

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
3c63a03 (HEAD -> master) Calc correlations.
76e2cd4 Add tsne calc and plot.
bcf2321 Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.
```

Or more verbosely:

```
me@laptop:~/projs/ei_ntwk$ git log
commit 3c63a03a397f08efe8c837050c4ae043fe2cf7eb (HEAD -> master)
Author: your name <you@example.com>
Date:   Sun Aug 12 16:11:22 2018 -0700
```

```
    Calc correlations.
```

```
commit 76e2cd46009e7f2f3c470e3154a28bb6f4b6a719
Author: your name <you@example.com>
Date:   Sun Aug 12 16:10:41 2018 -0700
```

```
    Add tsne calc and plot.
```

```
commit bcf23210b078e9b41b3cce7853942b8bed2120e3
Author: your name <you@example.com>
Date:   Sun Aug 12 16:09:41 2018 -0700
```

Add colored raster.

```
commit 49a732c2dc30778a2dc5bacaa137a286388e74d0
Author: your name <you@example.com>
Date:   Sun Aug 12 16:08:44 2018 -0700
```

Calc mean spike rate.

...

HOW TO: "UNSTAGE" ALL FILES

"Unstage" all files using `git reset`. This is useful if you realize you don't want to commit quite yet.

E.g. we stage our files in the usual way:

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working directory)

```
modified:   dim_reduc.py
modified:   plot.py
modified:   pop_stats.py
```

no changes added to commit (use "git add" and/or "git commit -a")

```
me@laptop:~/projs/ei_ntwk$ git add -A
```

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master

Changes to be committed:

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

```
modified:   dim_reduc.py
modified:   plot.py
modified:   pop_stats.py
```

Upon reflection we decide we want to make some more edits before committing.

We unstage the files first:

```
me@laptop:~/projs/ei_ntwk$ git reset
```

Unstaged changes after reset:

```
M    dim_reduc.py
M    plot.py
M    pop_stats.py
```

```
me@laptop:~/projs/ei_ntwk$ git status
```


On branch master

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working directory)

modified: dim_reduc.py

modified: plot.py

modified: pop_stats.py

no changes added to commit (use "git add" and/or "git commit -a")

HOW TO: PERMANENTLY UNDO COMMITS

Permanently erase all commits after a specific commit using: `git reset --hard <commit ID>`.

This can't be undone, so make sure you won't be losing important code!

BTW this would be an excellent time to make a backup first just in case.

Pick what commit you want to erase history after:

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
```

```
3c63a03 (HEAD -> master) Calc correlations.
```

```
76e2cd4 Add tsne calc and plot.
```

```
bcbf2321 Add colored raster.
```

```
49a732c Calc mean spike rate.
```

```
9e72fec Add local linear embedding.
```

```
ccb4824 Add isomap.
```

```
c825337 First commit.
```

Erase commit history after 49a732c.

```
me@laptop:~/projs/ei_ntwk$ git reset --hard 49a732c
```

```
HEAD is now at 49a732c Calc mean spike rate.
```

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
```

```
49a732c (HEAD -> master) Calc mean spike rate.
```

```
9e72fec Add local linear embedding.
```

```
ccb4824 Add isomap.
```

```
c825337 First commit.
```

LINEAR GIT: HOW TO EXIT VIM

If you forget to add commit message, by default you will be dropped into text editor Vim.

HOW TO: EXIT VIM

Don't be scared. Proceed as follows.

E.g.

```
me@laptop:~/projs/ei_ntwk$ git commit
```

drops you into:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#   modified:   dim_reduc.py
#
~
~
~
"/Dropbox/Repositories/git_tut/init/ei_ntwk/.git/COMMIT_EDITMSG" 8L, 213C
```

This is just a text editor for entering a commit message.

Vim has two modes: INSERT and COMMAND.

By default you will be dropped into COMMAND mode.

Type `i` to enter INSERT mode. Type commit message.

Add isomap.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#   modified:   dim_reduc.py
#
~
~
~
~
-- INSERT --
```

Hit ESC when done to return from INSERT to COMMAND mode.

```
Add isomap.  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
#  
# On branch master  
# Changes to be committed:  
#       modified:   dim_reduc.py  
#  
~  
~  
~  
~
```

Type :wq (make sure you're in COMMAND mode) and hit ENTER to save and quit:

```
Add isomap.  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
#  
# On branch master  
# Changes to be committed:  
#       modified:   dim_reduc.py  
#  
~  
~  
~  
~  
:wq
```

This returns you to the command line, and completes the commit output:

```
[master 1851c16] Add isomap.  
1 file changed, 3 insertions(+)
```

Note: If you mess up in Vim, hit ESC a bunch of times to make sure you're in command mode, then type :wq and hit ENTER to save your commit message and exit Vim.

You can learn more about Vim [here](#).

LINEAR GIT: PUSHING PROJECT HISTORY TO GITHUB AND REVIEWING ONLINE

GitHub: website/service that hosts git-based project history. Popular alternatives to GitHub are [BitBucket](#) and [GitLab](#).

HOW TO: MAKE ONLINE GITHUB REPOSITORY LINKED WITH LOCAL REPOSITORY

Sign up for free account [here](#).

Create repository with same name as project.


Note: Leave “Initialize this repository with a README” unchecked (to prevent merge conflict on first push).

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

Repository name

 rkp8000 ▾


 /

ei_ntwk ✓


Great repository names are short and memorable. Need inspiration? How about **urban-happiness**.

Description (optional)

Example research project for git practice.

☒  **Public**

Anyone can see this repository. You choose who can commit.


☐  **Private**

You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾

Add a license: **None** ▾ 

Create repository

Click “Create Repository” and you’ll land here:

Quick setup — if you've done this kind of thing before

 Set up in Desktop or **HTTPS** **SSH** 

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# ei_ntwk" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/rkp8000/ei_ntwk.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/rkp8000/ei_ntwk.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Now connect local repository to GitHub repository by adding “remote” (link) named “origin” (name “origin” is convention):

```
me@laptop:~/projs/ei_ntwk$ git remote add origin
https://github.com/me_on_github/ei_ntwk.git
```

Note: the above is all one line. Don't hit ENTER until after you've pasted the GitHub link.

Check that Git has added remote named “origin”:

```
me@laptop:~/projs/ei_ntwk$ git remote -v
origin      https://github.com/me_on_github/ei_ntwk.git (fetch)
origin      https://github.com/me_on_github/ei_ntwk.git (push)
```

HOW TO: PUSH LOCAL REPOSITORY TO GITHUB

Pushing means sending your local repository to GitHub so that the GitHub repository becomes identical to your local one.

First make sure commits are up-to-date:

```
me@laptop:~/projs/ei_ntwk$ git status
On branch master
nothing to commit, working tree clean
```

Push repository to GitHub using “origin” link:

```
me@laptop:~/projs/ei_ntwk$ git push origin master
Username for 'https://github.com': me_on_github
Password for 'https://me_on_github@github.com':
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (24/24), done.
Writing objects: 100% (24/24), 2.16 KiB | 2.16 MiB/s, done.
Total 24 (delta 9), reused 0 (delta 0)
remote: Resolving deltas: 100% (9/9), done.
To https://github.com/me_on_github/ei_ntwk.git
 * [new branch]      master -> master
```

Note 1: “master” refers to the master “branch” of your project (which is your only branch in Linear Git). Don’t worry about it.

Note 2: It’s good practice to push to GitHub after every commit.

Note 3: If `git push origin master` throws an error, this means there is a “merge conflict” (i.e. someone else or a different version of you has pushed something else to GitHub that differs from the code you’re trying to push)— see [Nonlinear Git: Handling Merges When Pushing To Github](#)

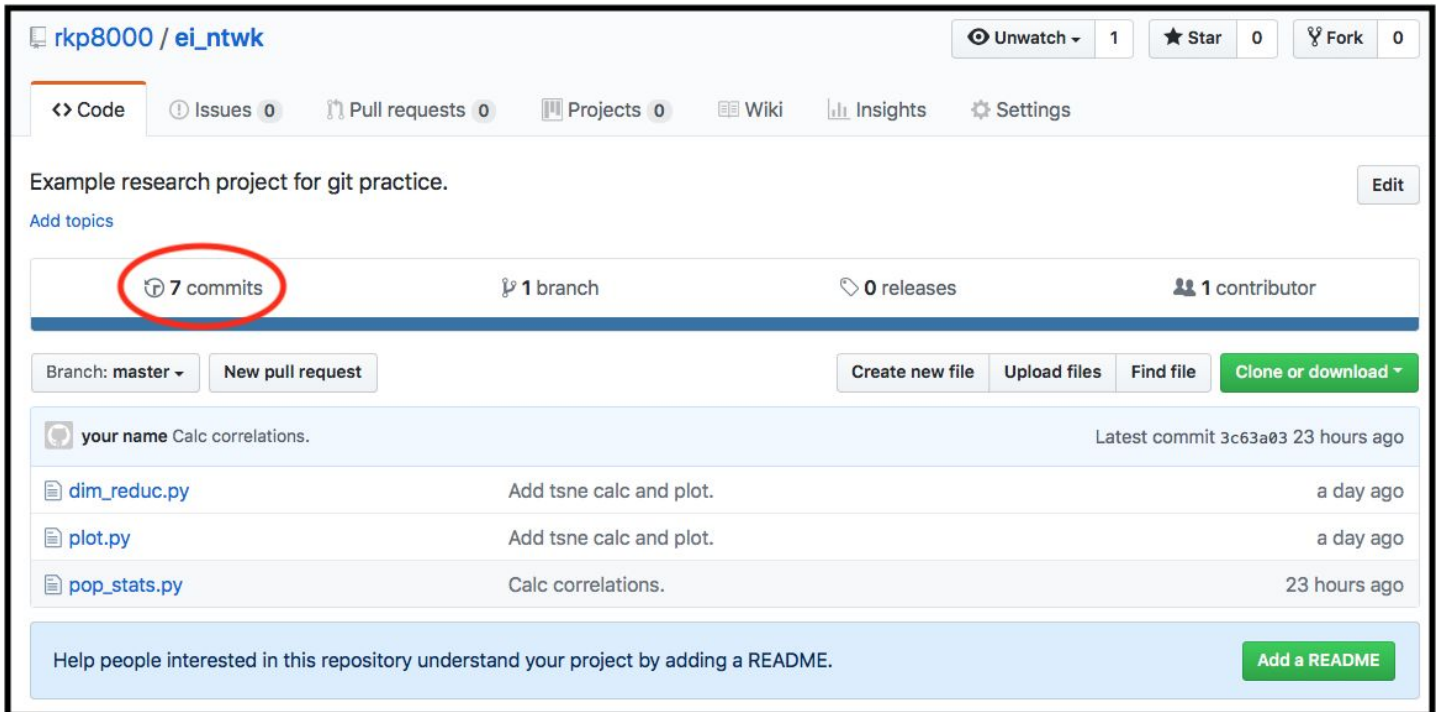
WHAT YOUR PROJECT NOW LOOKS LIKE ON GITHUB

Navigate to https://github.com/me_on_github/ei_ntwk.git in your browser again. You can see your project and look at all the files.

The screenshot shows the GitHub interface for a repository named 'ei_ntwk' by user 'rkp8000'. At the top, there are buttons for 'Unwatch', 'Star' (0), and 'Fork' (0). Below this is a navigation bar with tabs for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Insights', and 'Settings'. The main content area starts with a description: 'Example research project for git practice.' followed by an 'Edit' button and a link to 'Add topics'. Below this is a summary bar showing '7 commits', '1 branch', '0 releases', and '1 contributor'. A secondary bar contains a 'Branch: master' dropdown, a 'New pull request' button, and buttons for 'Create new file', 'Upload files', 'Find file', and a green 'Clone or download' button. The commit history is listed below, showing three commits by 'your name': 'Calc correlations.' (latest commit 3c63a03, 23 hours ago), 'dim_reduc.py' (Add tsne calc and plot, a day ago), 'plot.py' (Add tsne calc and plot, a day ago), and 'pop_stats.py' (Calc correlations, 23 hours ago). At the bottom, there is a light blue banner encouraging the user to 'Add a README' to help people understand the project.

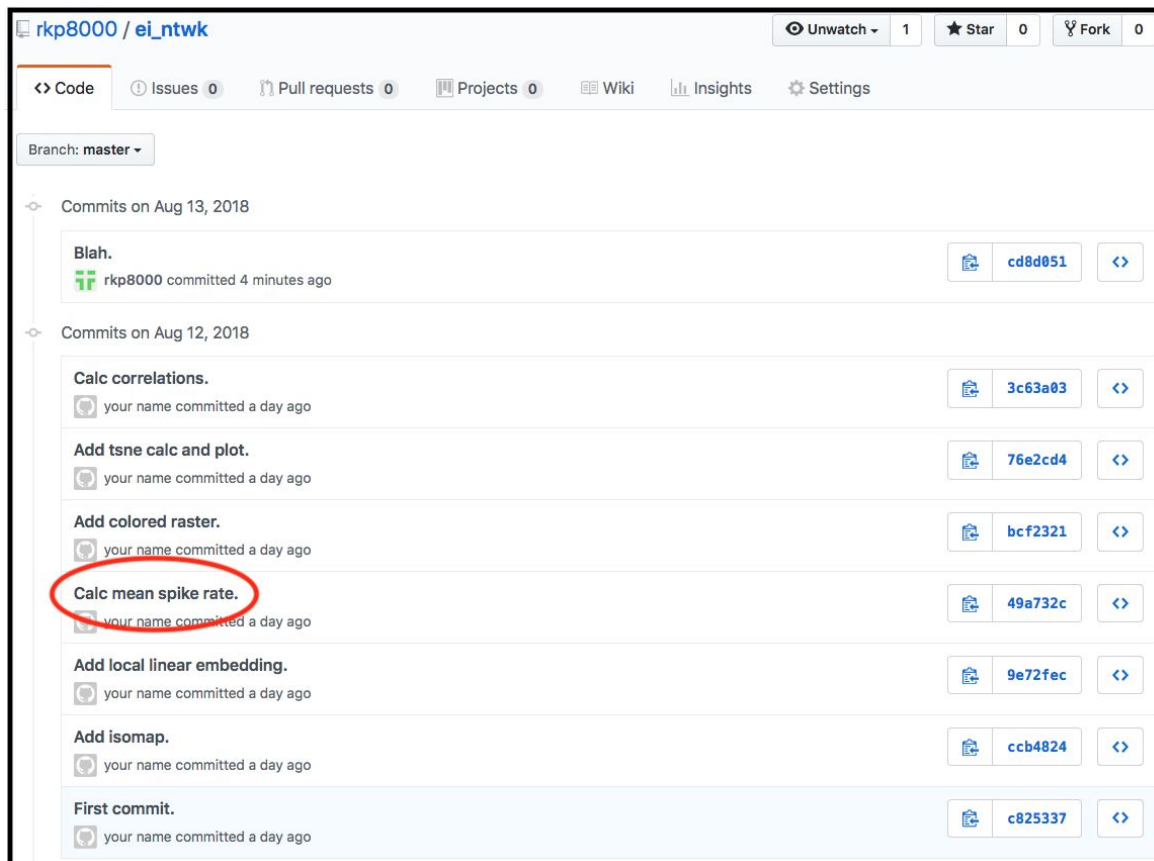
HOW TO: REVIEW PROJECT HISTORY ON GITHUB

Click on "<N> commits":



The screenshot shows the GitHub repository page for 'rkp8000 / ei_ntwk'. The repository description is 'Example research project for git practice.' The repository statistics show 7 commits, 1 branch, 0 releases, and 1 contributor. The '7 commits' link is circled in red. Below the statistics, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. A list of commits is shown, with the latest commit 'Calc correlations.' by 'your name' 23 hours ago. Below the commits, there is a button to 'Add a README'.

Click on commit you want to view:



The screenshot shows the GitHub commit history page for 'rkp8000 / ei_ntwk'. The page displays a list of commits, with the commit 'Calc mean spike rate.' by 'your name' 1 day ago circled in red. The commit history is filtered by date, showing commits from August 13, 2018, and August 12, 2018. The commit 'Calc mean spike rate.' has a commit hash of 49a732c. Below the commit history, there is a button to 'Add a README'.

Click “Browse files” to see snapshot of project at selected commit:

rkp8000 / ei_ntwk

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

Calc mean spike rate. **Browse files**

your name committed a day ago 1 parent 9e72fec commit 49a732c2dc30778a2dc5bacaa137a286388e74d0

Showing 1 changed file with 4 additions and 0 deletions. Unified Split

4 pop_stats.py View

```
... @@ -1,3 +1,7 @@
1 1 # analyze population spiking stats
2 2
3 3 spikes = get_spikes()
4 +
5 + mean_fr = calc_mean_fr(spikes)
6 + print(mean_fr)
7 +
```

0 comments on commit 49a732c Lock conversation

Write Preview AA B i “ < > ☰ ☷ ☹ @ 📎 ↩

Leave a comment

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported Comment on this commit

You can now view your project exactly as it was after the selected commit:

rkp8000 / ei_ntwk

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

Example research project for git practice. Edit

Add topics

4 commits 1 branch 0 releases 1 contributor

Tree: 49a732c2dc New pull request Create new file Upload files Find file Clone or download

your name Calc mean spike rate. Latest commit 49a732c a day ago

dim_reduc.py	Add local linear embedding.	a day ago
plot.py	First commit.	a day ago
pop_stats.py	Calc mean spike rate.	a day ago

Note: URL contains user, repository, and commit ID. E.g.:

https://github.com/me_on_github/ei_ntwk/tree/49a732c2dc30778a2dc5bacaa137a286388e74d0

LINEAR GIT: RUNNING OLD CODE

Note 1: Do the following if you want to run/tinker with old code but don't plan on saving any changes. If you plan on saving changes in new commits, you'll need to use branches, which we don't discuss in detail in this document. If you're considering doing the latter, I would recommend you stop and ask yourself whether this is really the best way to achieve your current scientific goal.

Note 2: If you're still learning Git, this is a good time to make a backup copy of your project repository before doing anything, just in case things go awry.

HOW TO: CHECK OUT PREVIOUS VERSION OF YOUR PROJECT

First make sure working directory is clean. This way you won't lose anything.

```
me@laptop:~/projs/ei_ntwk$ git status
On branch master
nothing to commit, working directory clean
```

Note 1: If working directory is dirty, either:

1. *Make a new commit.*
2. *Reset all changes (note: changes since most recent commit will be permanently lost!):*

```
me@laptop:~/projs/ei_ntwk$ git reset --hard
HEAD is now at 3c63a03 Calc correlations.
```

Note 2: if you don't want to delete your working directory changes or create a new commit, you can use [git stash](#), but this is more complicated.

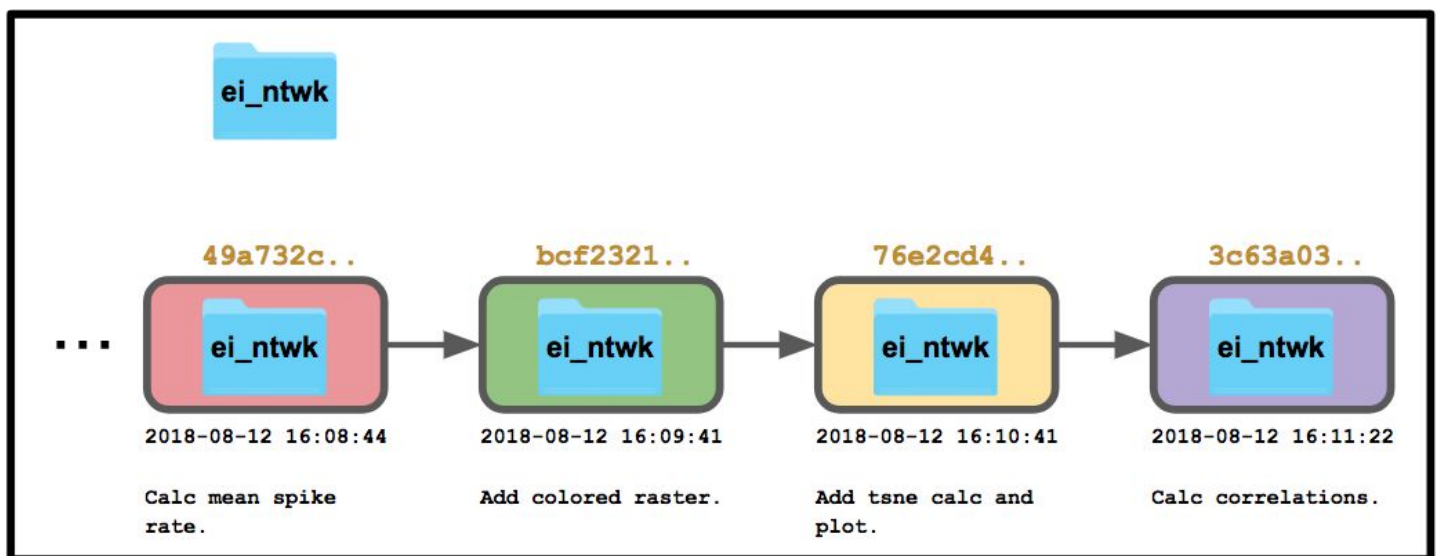
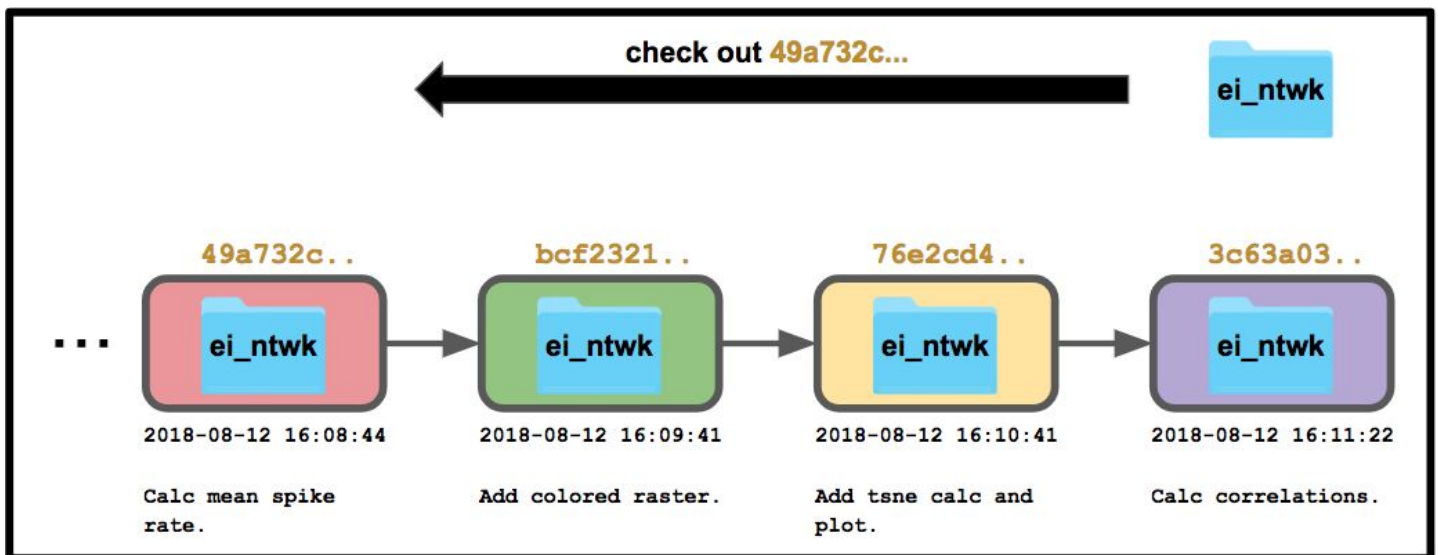
Get ID of commit to check out:

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
3c63a03 (HEAD -> master) Calc correlations.
76e2cd4 Add tsne calc and plot.
bcf2321 Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.
```

We'll choose **49a732c** in this example.

Check out previous commit

Once we check out `49a732c`, working directory will now show project as it was in commit `49a732c`. Diagrammatically, checking out previous commit looks like this, making working directory look like previous project version.



Check out previous commit with `git checkout <commit ID>`.

```
me@laptop:~/projs/ei_ntwk$ git checkout 49a732c
```

```
Note: checking out '49a732c'.
```

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 49a732c... Calc mean spike rate.

(Don't worry about what "detached HEAD" state or branches mean.)

Now working directory shows project as it was after commit 49a732c.

```
me@laptop:~/projs/ei_ntwk$ git status
HEAD detached at 49a732c
```

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
49a732c (HEAD) Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.
```

E.g. if we look at `pop_stats.py` now we see mean firing rate calculation (from 49a732c) but not correlation calculation (since this was added in commit 3c63a03):

`pop_stats.py`:

```
# analyze population spiking stats

spikes = get_spikes()

mean_fr = calc_mean_fr(spikes)
print(mean_fr)
```

Run/tinker with code from checked out commit.

You can now run your code as it was after commit 49a732c, as well as tinker with it and re-run it. This can be useful if you want to probe how an old analysis was done beyond just inspecting the code.

Remember: I'm assuming you don't want to save any changes you make here as new commits, so we're next going to undo them all.

HOW TO: UNDO CHANGES TO CHECKED OUT COMMIT

When done running/tinkering with code, undo changes with `git reset --hard` (*changes will be lost!*).

```
me@laptop:~/projs/ei_ntwk$ git status
HEAD detached at 49a732c
Changes not staged for commit:
```

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

(use "git checkout -- <file>..." to discard changes in working directory)

```
modified:    dim_reduc.py
```

no changes added to commit (use "git add" and/or "git commit -a")

```
me@laptop:~/projs/ei_ntwk$ git reset --hard
```

HEAD is now at 49a732c Calc mean spike rate.

```
me@laptop:~/projs/ei_ntwk$ git status
```

HEAD detached at 49a732c

nothing to commit, working tree clean

Now let's go back to our most recent commit.

HOW TO: RETURN TO MOST RECENT COMMIT

Run command git checkout master:

```
me@laptop:~/projs/ei_ntwk$ git checkout master
```

Previous HEAD position was 49a732c... Calc mean spike rate.

Switched to branch 'master'

Make sure you're back where you started

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master

nothing to commit, working tree clean

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
```

3c63a03 (HEAD -> master) Calc correlations.

76e2cd4 Add tsne calc and plot.

bcf2321 Add colored raster.

49a732c Calc mean spike rate.

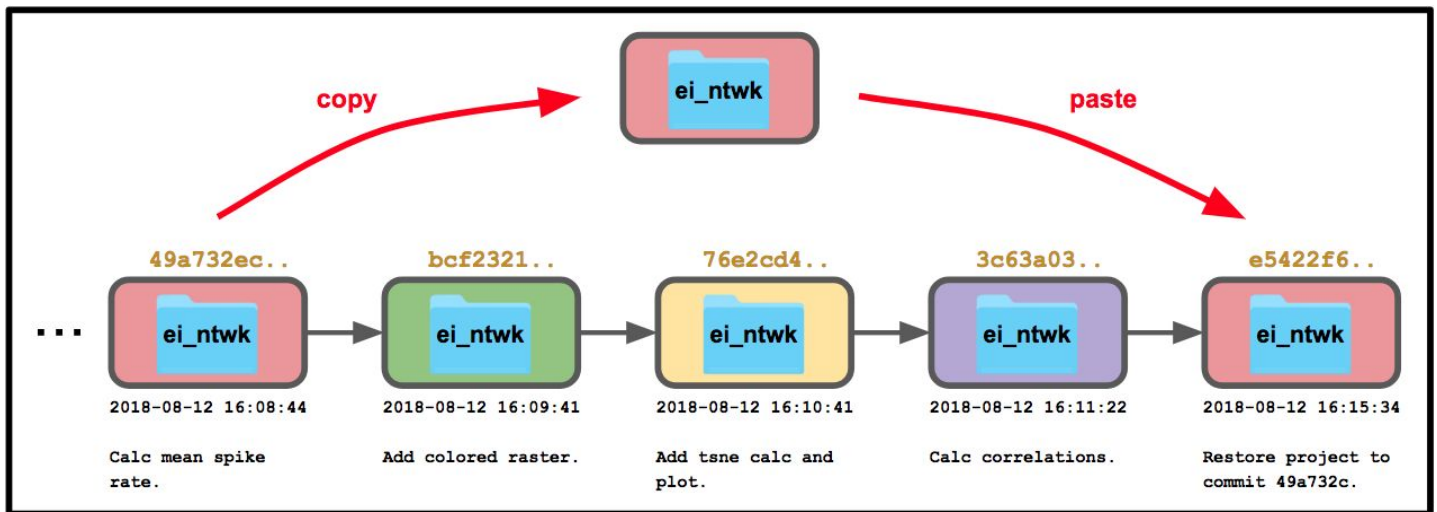
9e72fec Add local linear embedding.

ccb4824 Add isomap.

c825337 First commit.

LINEAR GIT: RESTORING PROJECT TO PREVIOUS STATE

Restore a project's previous state by copying previous commit to top of commit history, as shown below:



Note 1: This diagram doesn't depict exactly how Git does this, but it's okay to think about it like this for now.

Note 2: This doesn't erase any committed history, so you can always "unrestore" your project in the same way you restored it, e.g. in the diagram by performing the same operation on commit 3c63a03.

HOW TO: RESTORE PROJECT TO PREVIOUS STATE

Make sure all changes are committed (otherwise they'll be lost):

```
me@laptop:~/projs/ei_ntwk$ git status
```

```
On branch master
```

```
nothing to commit, working directory clean
```

Identify ID of commit to restore:

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
```

```
3c63a03 (HEAD -> master) Calc correlations.
```

```
76e2cd4 Add tsne calc and plot.
```

```
bcf2321 Add colored raster.
```

```
49a732c Calc mean spike rate.
```

```
9e72fec Add local linear embedding.
```

```
ccb4824 Add isomap.
```

```
c825337 First commit.
```

Here we'll choose to restore 49a732c. This will make working directory and top of project history identical to project as it was at commit 49a732c.

There are two ways to do this:

Method A: One-liner project restore

This creates new commit undoing all changes after 49a732c, but will throw error if any “nonlinearities” exist in project history.

Stage changes taking you back to 49a732c:

```
me@laptop:~/projs/ei_ntwk$ git revert --no-commit 49a732c..HEAD
```

If the preceding command throws an error, then go to **Method B** instead.

Otherwise, check status to see which files required “undo” operations:

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master

You are currently reverting commit 49a732c.

(all conflicts fixed: run "git revert --continue")

(use "git revert --abort" to cancel the revert operation)

Changes to be committed:

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

modified: dim_reduc.py

modified: plot.py

modified: pop_stats.py

Commit staged changes:

```
me@laptop:~/projs/ei_ntwk$ git commit -m "Restore proj to 49a732c."
```

[master 544108a] Restore proj to 49a732c.

3 files changed, 20 deletions(-)

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master

nothing to commit, working tree clean

```
me@laptop:~/projs/ei_ntwk$ git log --oneline
```

544108a (HEAD -> master) Restore proj to 49a732c.

3c63a03 Calc correlations.

76e2cd4 Add tsne calc and plot.

bcf2321 Add colored raster.

49a732c Calc mean spike rate.

9e72fec Add local linear embedding.

ccb4824 Add isomap.

c825337 First commit.

Now commit 544108a is identical to 49a732c.

Method B: Project restore through explicit copying of previous commit

The following will work even if there are “nonlinearities” in commit history.
(Thanks to Stack Overflow user [VonC](#) for the following.)

Note: the copy/paste lingo doesn't exactly mirror what Git's doing under the hood, but if you do the following it will work as if a copy/paste operation was performed.

Make sure you're starting from the right spot:

```
me@laptop:~/projs/ei_ntwk$ git checkout master
Already on 'master'
```

“Copy” commit 49a732c:

```
me@laptop:~/projs/ei_ntwk$ git checkout -b tmp
Switched to a new branch 'tmp'
```

```
me@laptop:~/projs/ei_ntwk$ git reset --hard 49a732c
HEAD is now at 49a732c Calc mean spike rate.
```

“Paste” commit 49a732c on top of project history/working directory:

```
me@laptop:~/projs/ei_ntwk$ git reset --soft master
me@laptop:~/projs/ei_ntwk$ git status
On branch tmp
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    modified:   dim_reduc.py
    modified:   plot.py
    modified:   pop_stats.py
```

Add commit message:

```
me@laptop:~/projs/ei_ntwk$ git commit -m "Restore project to commit 49a732c."
[tmp e5422f6] Restore project to commit 49a732c.
 3 files changed, 16 deletions(-)
```

Finalize “pasting” process:

```
me@laptop:~/projs/ei_ntwk$ git branch -M tmp master
```

Make sure everything worked:

```
me@laptop:~/projs/ei_ntwk$ git status
```

On branch master
nothing to commit, working tree clean

me@laptop:~/projs/ei_ntwk\$ git log --oneline

e5422f6 (HEAD -> master) Restore project to commit 49a732c.
3c63a03 Calc correlations.
76e2cd4 Add tsne calc and plot.
bcf2321 Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.

LINEAR GIT: EXCLUDING FILES AND FILE TYPES FROM GIT

For scientific projects, should generally only version control code and documentation with Git. E.g. don't version control data, pdfs, temp files, machine-specific files, compiled code, etc.

Recall that all included files will be duplicated and stored in history forever if any changes are made. For large files (e.g. images, videos), this can make Git repository very large and make upload/download times slow.

Note: version controlling data is a whole other can of worms.

E.g. what .gitignore might look like after subsequent edits:

```
# files types to ignore
temp_data.csv
*.pyc
```

To denote files for Git to ignore, create or edit hidden file called .gitignore in top-level project directory.

Here, temp_data.csv will be ignored by Git, as well as all files ending with .pyc. I.e. Git won't try to track these files or ask you to include them in commits.

HOW TO: OPEN/CREATE .gitignore FILE

If your .gitignore file is blank, you can edit it with Vim. Run `vi .gitignore` to edit .gitignore with Vim:

```
me@laptop:~/projs/ei_ntwk$ vi .gitignore
```

Edit with Vim, then save and exit (see [How To Exit Vim](#)).

```
me@laptop:~/projs/ei_ntwk$ ls -a
.          .git          dim_reduc.py pop_stats.py
..         .gitignore    plot.py
```

You can also ignore everything except a subset of files. In this case, .gitignore would look like:

```
# ignore all files
*
# except these
!*.py
!*.ipynb
!.gitignore
```

Here all files except .py (Python source code), .ipynb (Jupyter notebook files), and the .gitignore file itself would be ignored by Git.

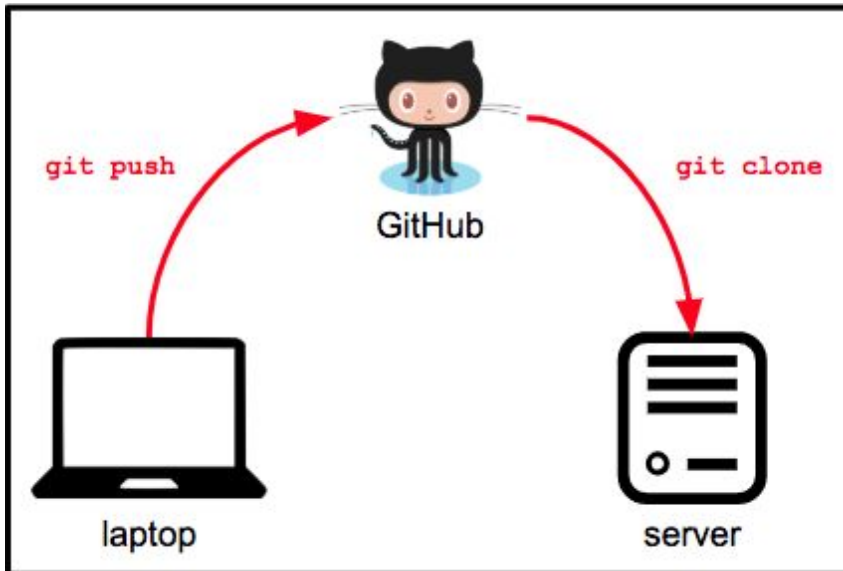
Note: Remember to stage .gitignore after editing it and commit.

NONLINEAR GIT: CLONING YOUR PROJECT TO A NEW COMPUTER FROM GITHUB

Sometimes you need to copy your code onto a new computer (e.g. a remote server).

For this tutorial we'll call local computer "laptop" and your secondary computer "server".

This diagram shows the basic idea:



HOW TO: CLONE YOUR REPOSITORY ONTO A NEW COMPUTER (E.G. A REMOTE SERVER)

On the server, change directory to where you want to copy code:

```
me@server:~$ cd ~/projs
```

Copy link to GitHub repo (e.g. https://github.com/me_on_github/ei_ntwk.git) and paste after `git clone`:

```
me@server:~/projs$ git clone https://github.com/me_on_github/ei_ntwk.git
Cloning into 'ei_ntwk'...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 24 (delta 9), reused 24 (delta 9), pack-reused 0
Unpacking objects: 100% (24/24), done.
```

This copies your GitHub repo into a local repo with same name:

```
me@server:~/projs$ ls
ei_ntwk
```

Move into the repo and it will be identical to the most recent commit on GitHub and have all commit history.

```
me@server:~/projs$ cd ei_ntwk/
me@server:~/projs/ei_ntwk$ ls
dim_reduc.py    plot.py        pop_stats.py

me@server:~/projs/ei_ntwk$ git log --oneline
3c63a03 (HEAD -> master, origin/master, origin/HEAD) Calc correlations.
76e2cd4 Add tsne calc and plot.
bcf2321 Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.

me@server:~/projs/ei_ntwk$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Note 1: A remote named origin pointing to GitHub repo is already included in cloned repo.

```
me@server:~/projs/ei_ntwk$ git remote -v
origin      https://github.com/me_on_github/ei_ntwk.git (fetch)
origin      https://github.com/me_on_github/ei_ntwk.git (push)
```

Note 2: Only have to clone once. If you push again from laptop to GitHub and want to bring server up-to-date, use `git pull` (described below). However, if this gets scary, can always delete server repo and git clone from GitHub again (just be sure you won't be deleting anything important on the server!).

NONLINEAR GIT: BRINGING SERVER (OR LOCAL) REPO UP-TO-DATE WITH GITHUB

If project already on server, no need to clone again. Simply bring server repo up-to-date with GitHub repo:

Ensure server repository has clean working directory:

```
me@server:~/projs/ei_ntwk$ git status
```

On branch master

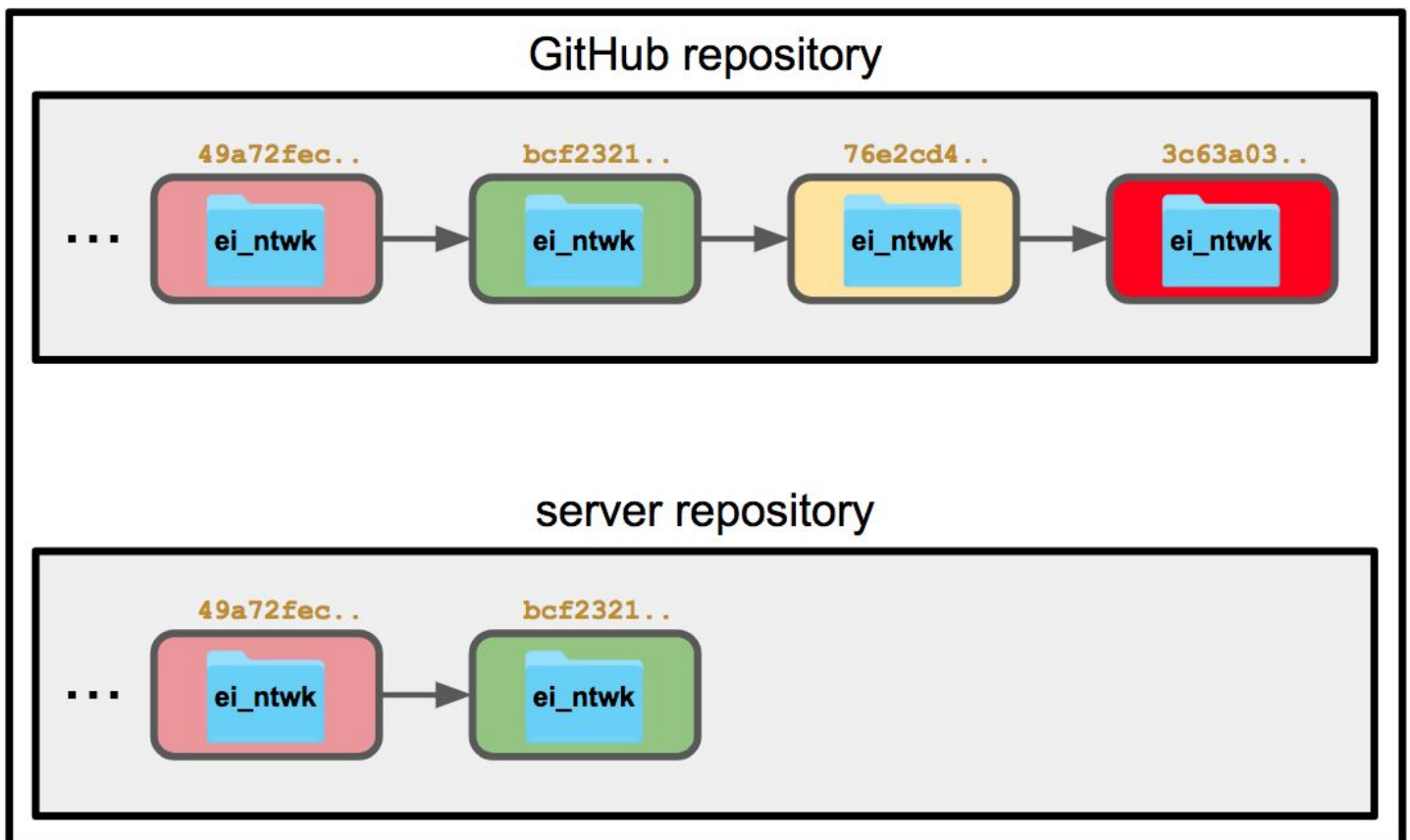
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.

(use "git pull" to update your local branch)

Don't worry about stuff after On branch master. (This just indicates GitHub repo is strict extension of server repo by 2 commits, as in Case A below.)

Next, two possible cases:

Case A: GitHub commit history is strict extension of server commit history.



First, can verify that local repo lags GitHub repo by comparing local commit history to GitHub commit history.

Local commit history:

```
me@server:~/projs/ei_ntwk$ git log --oneline
```

bcf2321 (HEAD -> master) Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.

Commit history on GitHub:

Commits on Aug 12, 2018		
Calc correlations. your name committed a day ago	3c63a03	<>
Add tsne calc and plot. your name committed a day ago	76e2cd4	<>
Add colored raster. your name committed a day ago	bcf2321	<>
Calc mean spike rate. your name committed a day ago	49a732c	<>
Add local linear embedding. your name committed a day ago	9e72fec	<>
Add isomap. your name committed a day ago	ccb4824	<>
First commit. your name committed a day ago	c825337	<>

HOW TO: BRING SERVER REPOSITORY UP-TO-DATE WITH GITHUB REPO

Update server repo with `git pull origin master`:

```
me@server:~/projs/ei_ntwk$ git pull origin master
```

```
From https://github.com/me_on_github/ei_ntwk
* branch          master      -> FETCH_HEAD
Updating bcf2321..3c63a03
Fast-forward
 dim_reduc.py | 4 ++++
 plot.py      | 4 ++++
 pop_stats.py | 3 +++
 3 files changed, 11 insertions(+)
```

Again, this only works if working directory is clean and GitHub commit is strict extension of local commit history.

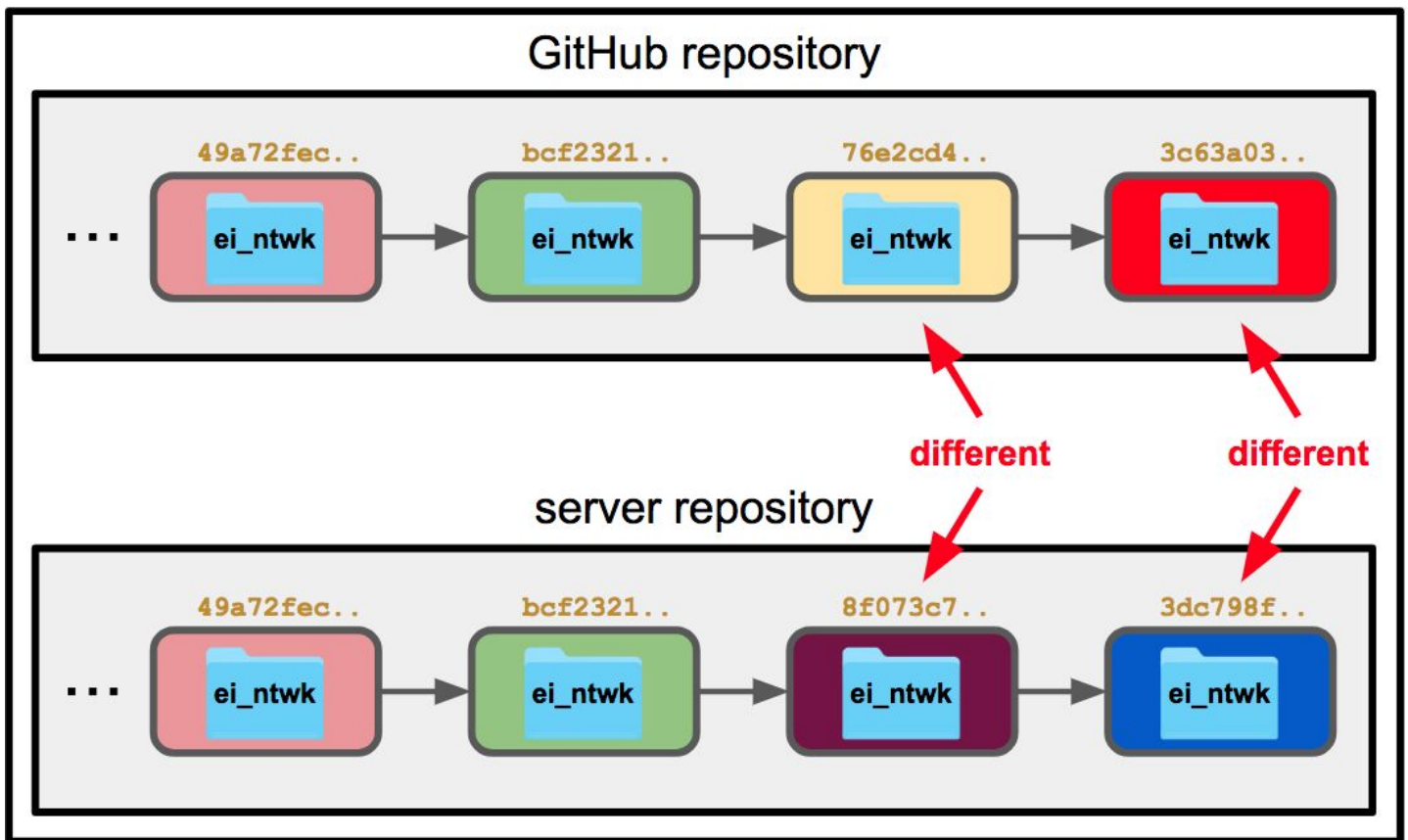
Case B: server commit history is NOT strict extension of GitHub commit history

If you get an error using `git pull origin master` this is almost certainly why.

To resolve such error, need to understand fundamental cause of it (see next section).

NONLINEAR GIT: MERGING GITHUB HISTORY INTO LOCAL HISTORY (AND HANDLING CONFLICTS)

Consider **Case B** from previous section—server commit history is NOT strict extension of GitHub history:



Here commits after `bcf2321` differ between GitHub and server repositories.

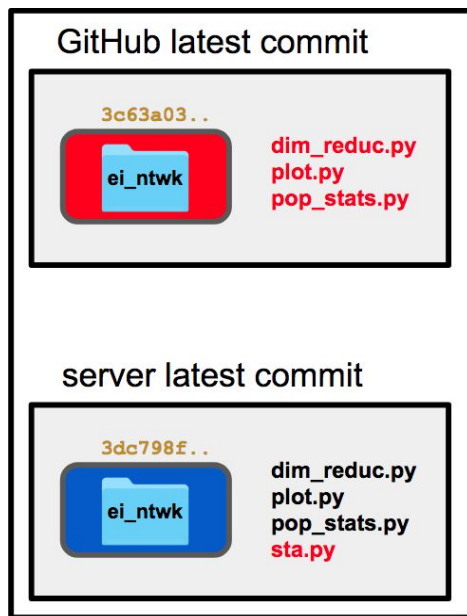
HOW TO: HANDLE MERGES WHEN SERVER/LOCAL AND GITHUB COMMIT HISTORY DIVERGE

Note: executing `git pull origin master` may modify your code, even if merge conflict arises and it looks like it threw an error, so if code worked beforehand it might not run after this command (until you fix things). This would therefore be another good time to make a backup of your repository before proceeding.

If you run `git pull origin master` Git will either:

1. Merge `3c63a03` into `3dc798f` without conflicts

This occurs if latest commit on GitHub and latest commit on server have created/modified/deleted *strictly non-overlapping sets* of files since commit `bcf2321`. Here **red text** indicates files modified since commit `bcf2321`:



In this example, the latest GitHub commit has modified `dim_reduc.py`, `plot.py`, and `pop_stats.py` since the commit where the server and GitHub diverged (`bcf2321`). The server's latest commit, however, has only created and modified the file `sta.py`:

```
me@server:~/projs/ei_ntwk$ git log --oneline
3dc798f (HEAD -> master) Normalize STA calc.
8f073c7 Add STA calc.
bcf2321 Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.
```

Running `git pull origin master` now drops you into Vim with default commit message for you to edit/approve.

```
me@server:~/projs/ei_ntwk$ git pull origin master
```


drops you into Vim (if you don't remember what Vim is, see [Linear Git: How To Exit Vim](#)):

```
Merge branch 'master' of https://github.com/me_on_github/ei_ntwk

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
~
"/projs/ei_ntwk/.git/MERGE_MSG" 7L, 287C
```

The top line shows the default merge message.

Edit this message using VIM (see [LINEAR GIT: HOW TO EXIT VIM](#)) **or accept it and quit by typing**
:wq **and hitting** ENTER

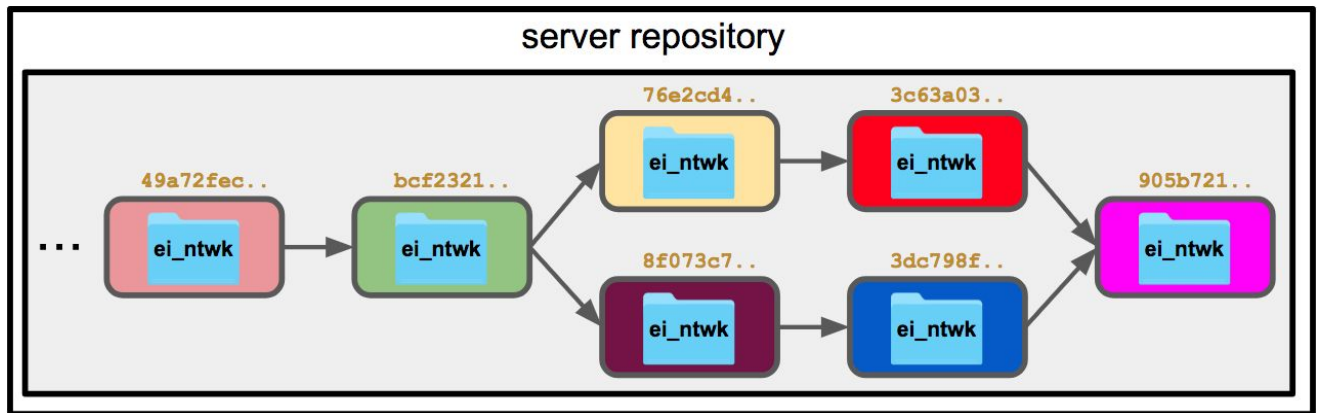
```
Merge branch 'master' of https://github.com/me_on_github/ei_ntwk

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
~
:wq
```

This returns you to the command prompt:

```
me@server:~/projs/ei_ntwk$ git pull origin master
From https://github.com/me_on_github/ei_ntwk
 * branch                master          -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 dim_reduc.py | 4 ++++
 plot.py      | 4 ++++
 pop_stats.py | 3 +++
 3 files changed, 11 insertions(+)
me@server:~/projs/ei_ntwk$
```

The output from Git tells you the two “branches” (the GitHub and server branches) have merged. Your local commit history now looks like this:



You can even see the branch structure in the command line:

```

me@server:~/projs/ei_ntwk$ git log --oneline --graph
* 905b721 (HEAD -> master) Merge branch 'master' of https://githu...
| \
| * 3c63a03 (origin/master, origin/HEAD) Calc correlations.
| * 76e2cd4 Add tsne calc and plot.
* | 3dc798f Normalize STA calc.
* | 8f073c7 Add STA calc.
| /
* bcf2321 Add colored raster.
* 49a732c Calc mean spike rate.
* 9e72fec Add local linear embedding.
* ccb4824 Add isomap.
* c825337 First commit.
  
```

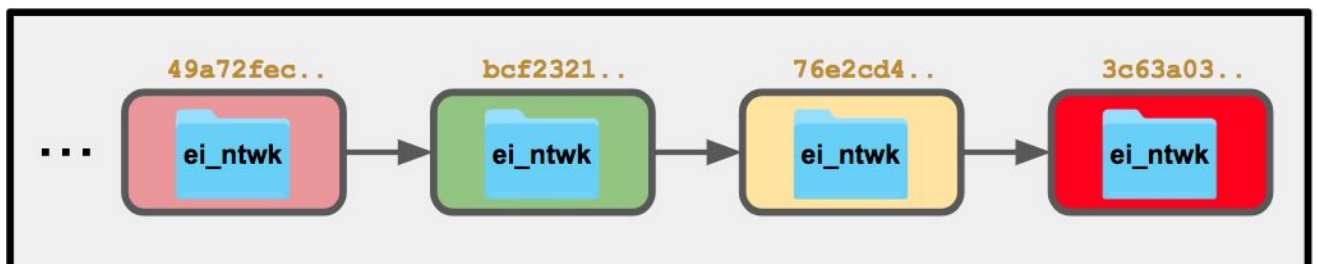
Run `git status` to see that everything is fine and dandy.

```

me@server:~/projs/ei_ntwk$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

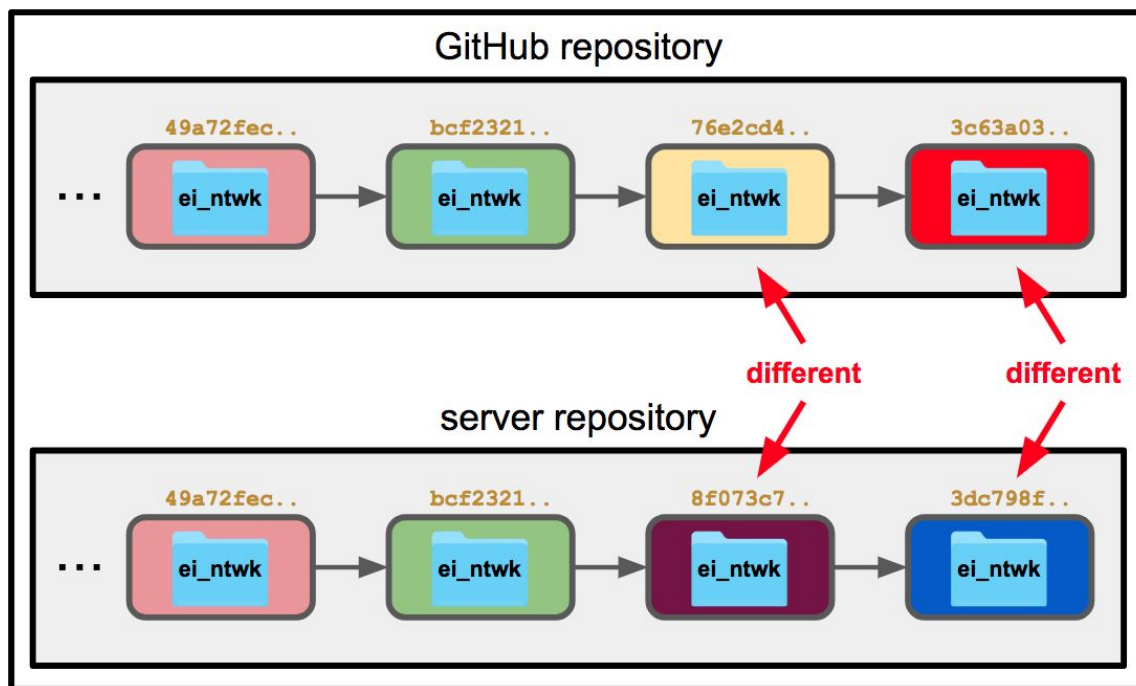
nothing to commit, working tree clean
  
```

Note: the server commit history is now ahead of the GitHub repo, which still looks like:

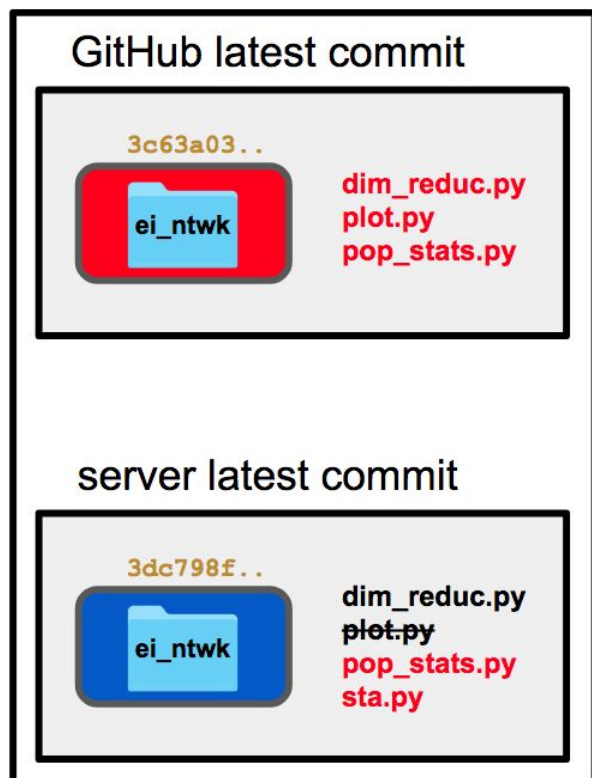


2. Alert you that there is a merge conflict to resolve

This occurs if latest commit on GitHub vs. on server have modified/deleted overlapping sets of files. Recall how the GitHub and server commit histories diverged:



Now consider case that latest commit on GitHub and latest commit on server have modified some of same files.



Here both server and GitHub latest commits have modified `pop_stats.py`, and server latest commit has deleted `plot.py` whereas GitHub latest commit has modified it. In particular, server commit history might look like:

```
me@server:~/projs/ei_ntwk$ git log --oneline
3dc798f (HEAD -> master) Add cov calc.
8f073c7 Add STA and delete plots.
bcf2321 Add colored raster.
49a732c Calc mean spike rate.
9e72fec Add local linear embedding.
ccb4824 Add isomap.
c825337 First commit.
```

If we `git pull origin master` now Git won't know what to do, since not clear if GitHub version or the server version of each conflicted file should be kept, or if you want to keep pieces of both.

Specifically, Git will tell us there is merge conflict:

```
me@server:~/projs/ei_ntwk$ git pull origin master
From https://github.com/me_on_github/ei_ntwk
 * branch                master          -> FETCH_HEAD
Auto-merging pop_stats.py
CONFLICT (content): Merge conflict in pop_stats.py
CONFLICT (modify/delete): plot.py deleted in HEAD and modified in
3c63a03a397f08efe8c837050c4ae043fe2cf7eb. Version
3c63a03a397f08efe8c837050c4ae043fe2cf7eb of plot.py left in tree.
Automatic merge failed; fix conflicts and then commit the result.
```

Basically, Git tried to merge but failed.

Look at `CONFLICT` **lines**. These tell us:

1. Both GitHub and server latest commit have modified `pop_stats.py`.
2. Server latest commit (HEAD) has deleted `plot.py` and GitHub latest commit (`3c63a03..`) has modified it.

Get more info with `git status`:

```
me@server:~/projs/ei_ntwk$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 2 different commits each, respectively.
    (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
    (fix conflicts and run "git commit")
    (use "git merge --abort" to abort the merge)
```

Changes to be committed:

```
modified:   dim_reduc.py
```

Unmerged paths:

(use "git add/rm <file>..." as appropriate to mark resolution)

```
deleted by us:    plot.py
both modified:    pop_stats.py
```

This tells us `dim_reduc.py` (which was only modified in GitHub latest commit since `bcf2321` and not in server latest commit since `bcf2321`) is staged.

Lines in red, however, tell us where merge conflicts lie.

We must now resolve merge conflicts.

First, note which files are present in working directory:

```
me@server:~/projs/ei_ntwk$ ls
dim_reduc.py  plot.py      pop_stats.py  sta.py
```

Conflicted files are `plot.py` and `pop_stats.py`.

If we look inside `pop_stats.py` we'll see:

```
# analyze population spiking stats

spikes = get_spikes()

mean_fr = calc_mean_fr(spikes)
print(mean_fr)

<<<<<<< HEAD
covs = calc_covs(spikes)
=====
corrs = calc_corrs(spikes)
print(corrs)

>>>>>>> 3c63a03a397f08efe8c837050c4ae043fe2cf7eb
```

Git has identified conflicting changes and said who has made what changes. In example, server latest commit (HEAD) has added `calc_covs` whereas GitHub latest commit (`3c63a03` . .) has added `calc_corrs`. We can keep one, both, or neither. Let's keep both, so `pop_stats.py` looks like:

```
# analyze population spiking stats

spikes = get_spikes()
```

```
mean_fr = calc_mean_fr(spikes)
print(mean_fr)
```

```
cov = calc_cov(spikes)
```

```
corrs = calc_corrs(spikes)
print(corrs)
```

We can now stage pop_stats.py:

```
me@server:~/projs/ei_ntwk$ git add pop_stats.py
me@server:~/projs/ei_ntwk$ git status
```

On branch master

Your branch and 'origin/master' have diverged,
and have 2 and 2 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Changes to be committed:

```
    modified:   dim_reduc.py
    modified:   pop_stats.py
```

Unmerged paths:

(use "git add/rm <file>..." as appropriate to mark resolution)

```
    deleted by us:   plot.py
```

Finally, let's handle plot.py conflict.

If we look at plot.py we'll see:

```
# plotting functions
```

```
def raster(spikes):
    fig = make_fig()
    make_raster(fig, spikes)
```

```
def colored_raster(spikes, colors):
    fig = make_fig()
    make_raster(fig, spikes, colors)
```

```
def tsne(embedding):
    fig = make_fig()
```

```
scatter(embedding)
```

Here we have no interference from Git inside file. This is because conflict is this version (GitHub's) of plot.py vs no plot.py at all (server's version).

We can either stage GitHub's version of plot.py with `git add plot.py` or we can remove it using `git rm plot.py`. Let's remove it to fix the conflict:

```
me@server:~/projs/ei_ntwk$ git rm plot.py
```

```
plot.py: needs merge
rm 'plot.py'
```

```
me@server:~/projs/ei_ntwk$ git status
```

```
On branch master
Your branch and 'origin/master' have diverged,
and have 2 and 2 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)
```

```
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)
```

Changes to be committed:

```
modified:   dim_reduc.py
modified:   pop_stats.py
```

Finally, run `git commit -m "<commit message>"` to create the merge commit:

```
me@server:~/projs/ei_ntwk$ git commit -m "Merge GitHub and server repos."
```

```
[master 3e8a1af] Merge GitHub and server repos.
```

```
me@server:~/projs/ei_ntwk$ git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
(use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

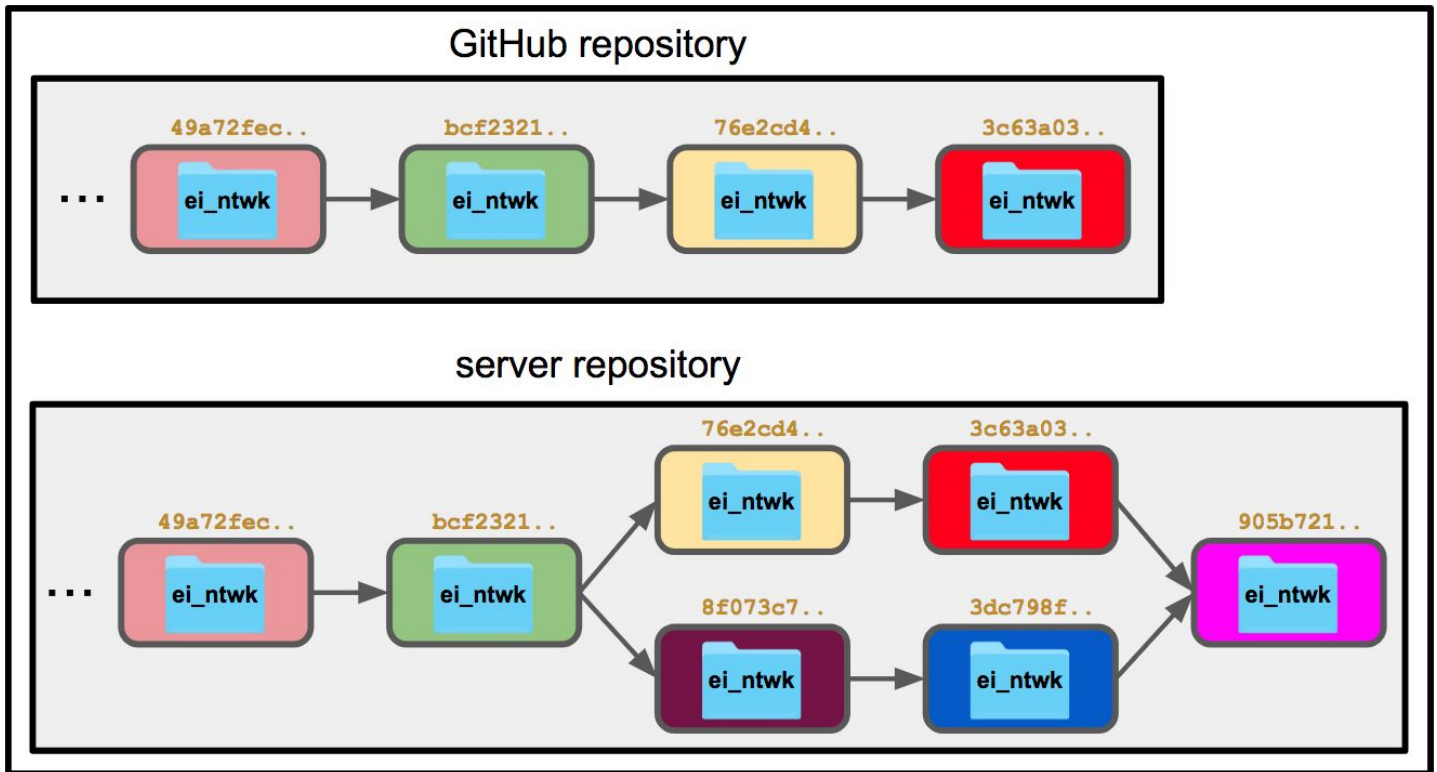
```
me@server:~/projs/ei_ntwk$ $ git log --oneline --graph
```

```
*    3e8a1af (HEAD -> master) Merge GitHub repo into server repo.
| \
| * 3c63a03 (origin/master, origin/HEAD) Calc correlations.
| * 76e2cd4 Add tsne calc and plot.
* | b06f1bc Add cov calc.
* | f6f7889 Add STA and delete plots.
|/
* bcf2321 Add colored raster.
* 49a732c Calc mean spike rate.
```

- * 9e72fec Add local linear embedding.
- * ccb4824 Add isomap.
- * c825337 First commit.

Now everything is dandy.

*Note: after merging GitHub commit history into server commit history in both **Case 1** and **Case 2**, server commit history will now be ahead of GitHub commit history:*

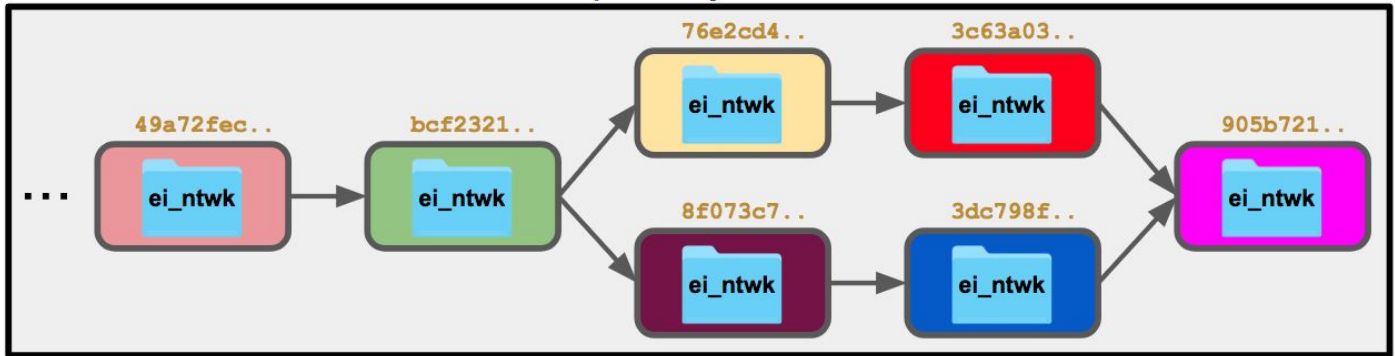


To bring the GitHub repo up-to-date with the server repo, you can run `git push origin master`. As long as GitHub commit history hasn't changed since you performed merge on server, everything will work. This is because GitHub commit history is strict subgraph of server commit history.

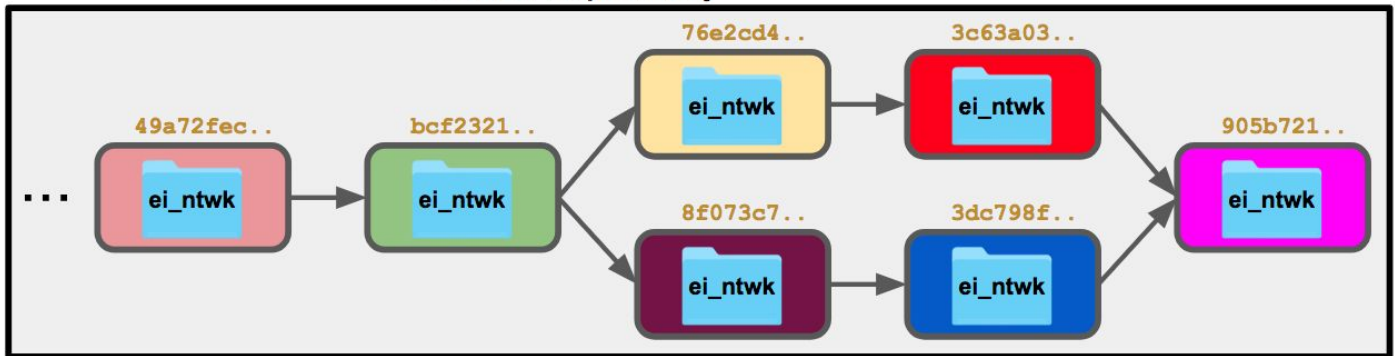
```
me@server:~/projs/ei_ntwk$ git push origin master
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 771 bytes | 385.00 KiB/s, done.
Total 8 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 2 local objects.
To https://github.com/me_on_github/ei_ntwk.git
3c63a03..905b721 master -> master
```

Result is GitHub and server repositories are now identical again.

GitHub repository



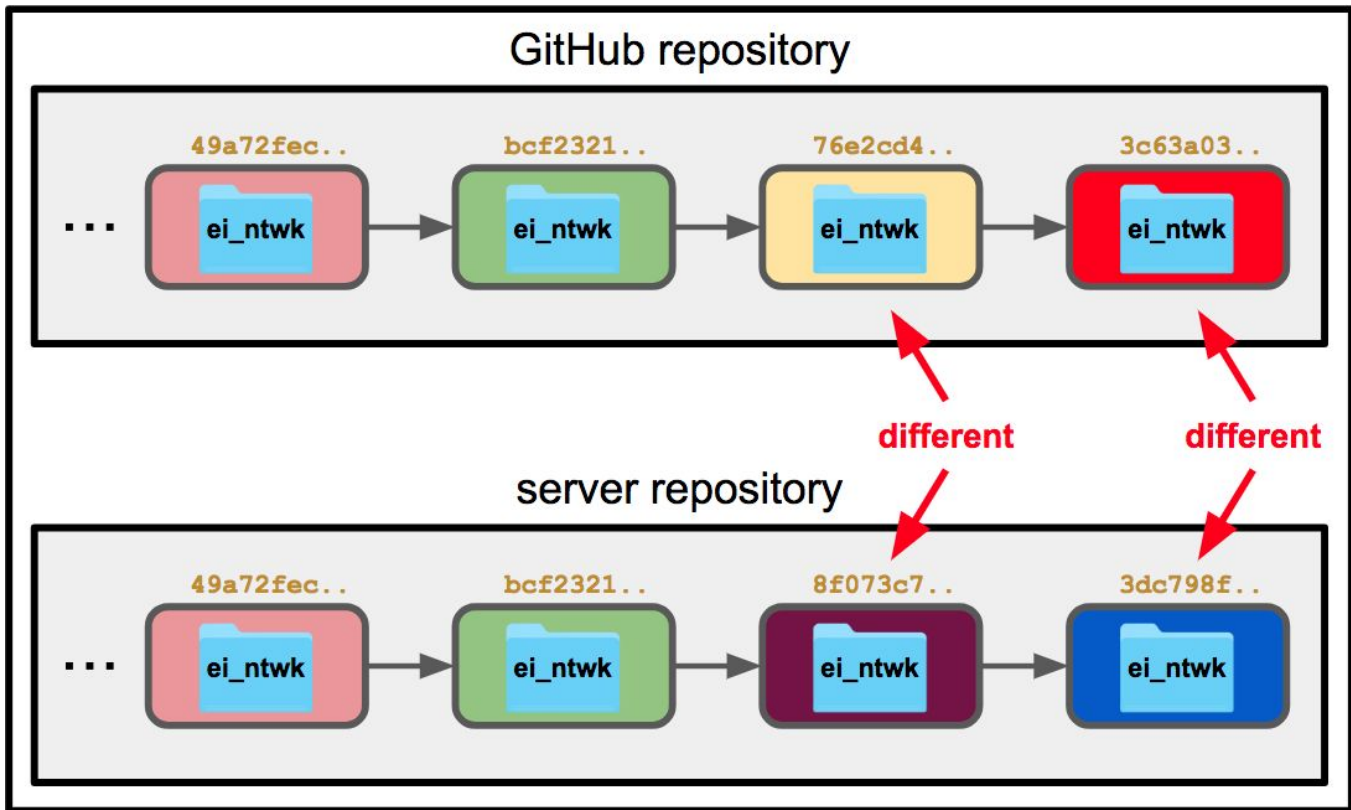
server repository



NONLINEAR GIT: HANDLING MERGES WHEN PUSHING TO GITHUB

Other common situation: GitHub and server (or laptop) commit histories differ after branch point, and you try to push from server (or laptop) to GitHub.

E.g. starting with



and running `git push origin master` yields the following error:

HOW TO: FIX MERGE CONFLICTS WHEN PUSH TO GITHUB IS REJECTED

For example:

```
me@server:~/projs/ei_ntwk$ git push origin master
To https://github.com/me_on_github/ei_ntwk.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to
'https://github.com/me_on_github/ei_ntwk.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This means there is conflict, i.e. GitHub commit history is not strict subgraph of server commit history, so Git doesn't know what to do.

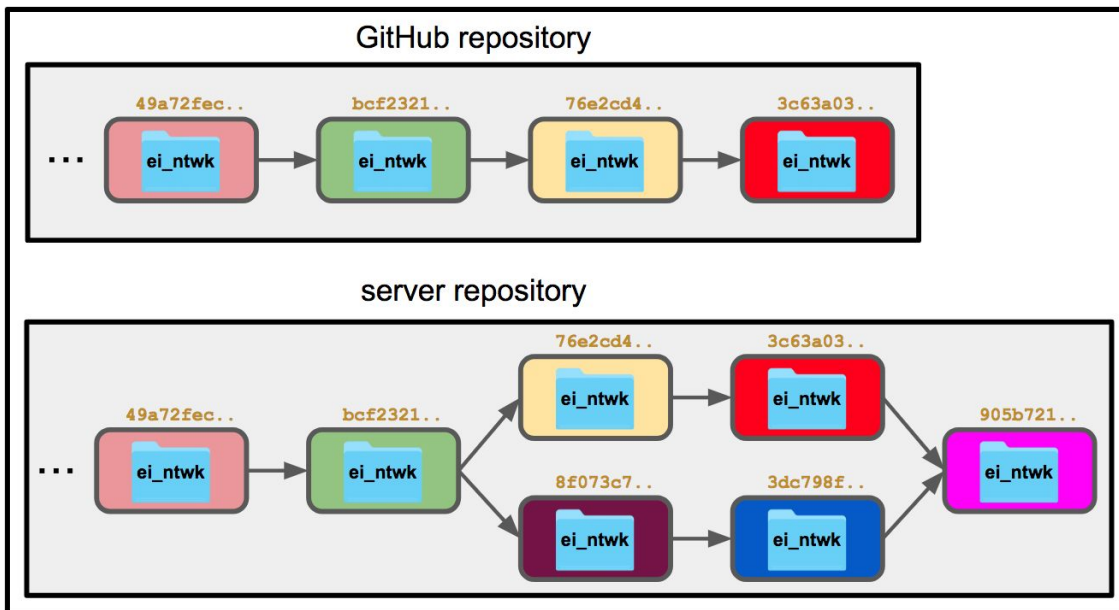
To fix this: pull GitHub commit history onto server (as described in previous section), fix conflicts, then push back to GitHub.

Pull from GitHub onto server:

```
me@server:~/projs/ei_ntwk$ git pull origin master
```

Next enter merge message in Vim [[Case 1 above](#)] or resolve conflicts and commit [[Case 2 above](#)].

This will yield a server repository with this commit history structure:



Then push back to GitHub (this works because GitHub repo is now subgraph of server repo, so there are no more conflicts).

```
me@server:~/projs/ei_ntwk$ git push origin master
```

```
Counting objects: 8, done.
```

```
Delta compression using up to 4 threads.
```

```
Compressing objects: 100% (6/6), done.
```

```
Writing objects: 100% (8/8), 771 bytes | 385.00 KiB/s, done.
```

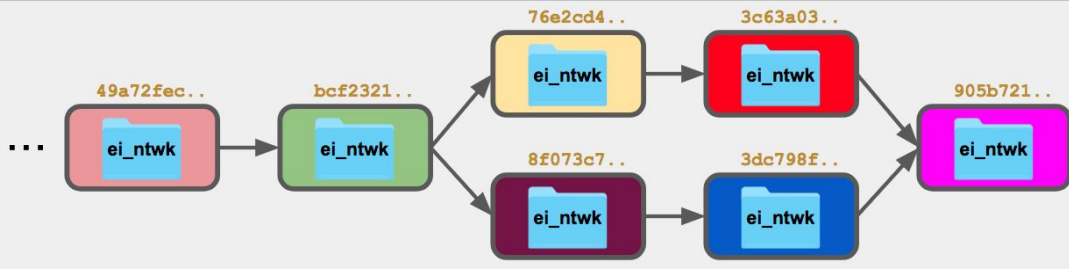
```
Total 8 (delta 3), reused 0 (delta 0)
```

```
remote: Resolving deltas: 100% (3/3), completed with 2 local objects.
```

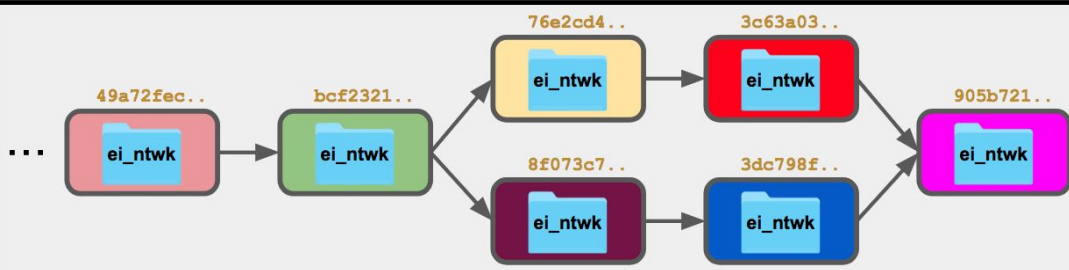
```
To https://github.com/me_on_github/ei_ntwk.git
```

```
3c63a03..905b721 master -> master
```

GitHub repository

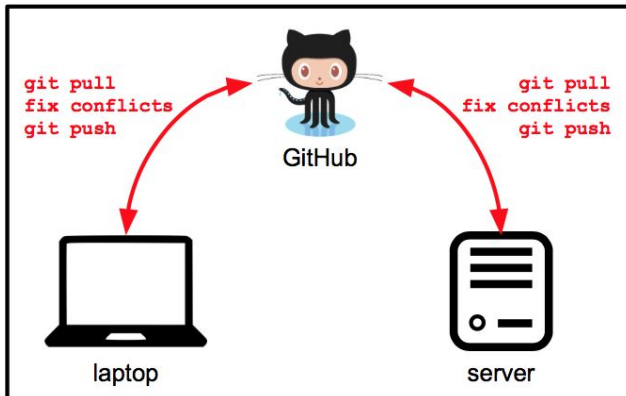


server repository

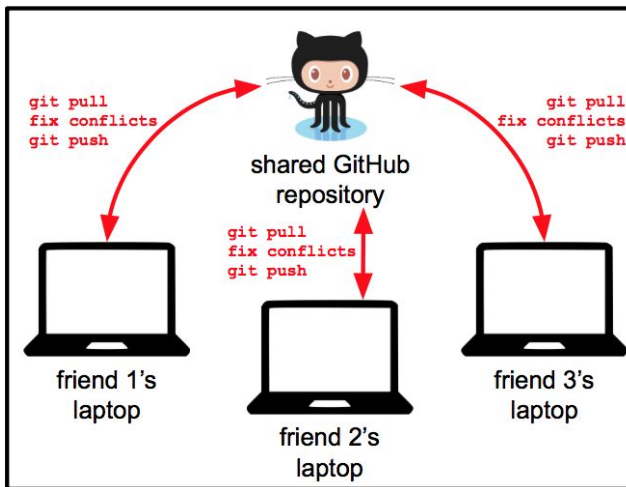


NONLINEAR GIT: GENERALIZED WORKFLOW AND BASIC COLLABORATION

Merging (with or without conflicts) is generally necessary whenever GitHub commit history diverges from commit history on computer your working on (either laptop or server) and you want to push or pull:

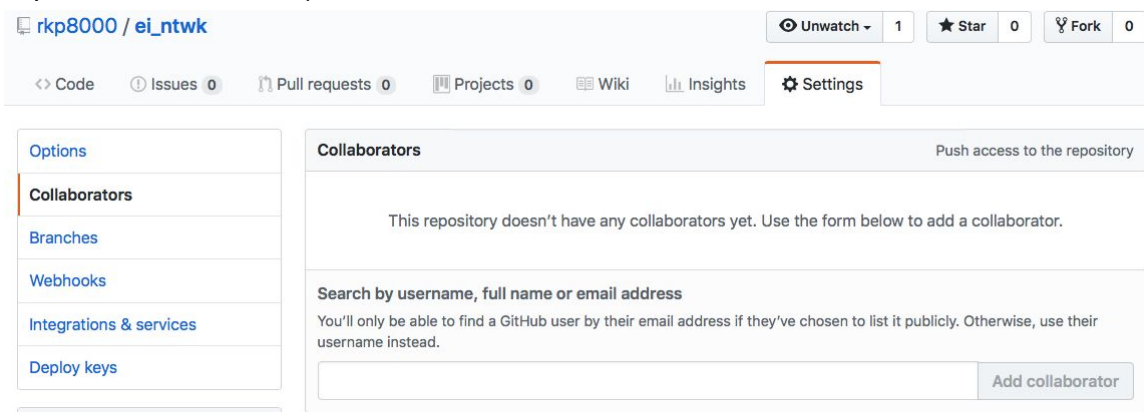


This also frames basic collaboration workflow on GitHub (multiple collaborators using same GitHub repo):



Essentially, to work on shared code, pull code from GitHub onto local computer, resolve conflicts, make edits, then push back to GitHub. However, if someone else updates GitHub repo while you're working on your code, you'll have to pull again from GitHub and resolve conflicts again before pushing back to GitHub.

Note: collaborators can be added to GitHub repository through Settings → Collaborators. (All collaborators require GitHub account.)



APPENDIX: INSTALLING GIT

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

APPENDIX: POPULAR GUIs FOR GIT

<https://www.slant.co/topics/4985/~visual-git-guis>

APPENDIX: EXAMPLE REPOSITORY IN THIS TUTORIAL

https://github.com/rkp8000/ei_ntwk

APPENDIX: EXAMPLE REPO ORGANIZATION OF PROJECT WITH DATA

Many scientific projects involve datasets, but data should not be version controlled with Git since it will weigh down your repository. However, it's wise to keep data within repository so project stays self-contained. One useful way of doing this is as follows.

Directory organization:

```
|-- ei_ntwk
    |-- .gitignore
    |-- .git
    |-- ...
    ...
    |-- dim_reduc.py
    |-- plot.py
    |-- pop_stats.py
    |-- data
    |-- ...
    ...
```

Note: If you must put your data folder outside of your repo (e.g. in a shared folder somewhere else on your computer), create a [symbolic link](#) to it inside your repo so that it can be accessed by your code as if it did live inside your repo.

What to put in .gitignore:

```
# ignore Python bytecode
*.pyc

# ignore data
data
```

Here Git will ignore all Python bytecode (created by Python to save time on next code execution), as well as everything inside the data folder.

Alternative .gitignore organization:

```
# ignore all files
*

# except these
!.gitignore
!*.py
!*.ipynb
```

Here, everything except .py and .ipynb (IPython/Jupyter notebooks) files will be ignored, including everything contained in the data folder.

