Basic Git and GitHub

STSCI 4780/5780 Lab
Tom Loredo

Agenda

- Background technology
  - Markdown
  - Hashes/checksums
  - Diffs and patches
  - File systems
- Git
- GitHub
- Exercises & Assignment01

Markdown — A mark*up* language

Plain text

README.md 1287 Words

```
# STSCI 4780 - Bayesian data analysis: principles and practice

**Lectures:**  Tuesdays & Thursdays, 1:25pm - 2:40pm, in Upson 109
**Labs:**  Fridays, 2:55pm - 4:10pm, in Phillips 213

**Instructor:**
Tom Loredo
Center for Radiophysics & Space Research, and Field of Statistics
620 Space Sciences Building
loredo@astro.cornell.edu
Office hours:  Wednesdays, 3pm - 4pm, and by appointment

**Teaching Assistant:**
Kerstin Frailey
Dept. of Statistical Science
Malott Hall
kef73@cornell.edu
Office hours TBD

# Course goals

Provide students:

- A basic understanding of the principles and foundations underlying the Bayesian approach
- Practical experience using basic/intermediate Bayesian methods
- Experience with widely-used tools and software development practices for producing and sharing collaborative, reproducible statistical research
- Exposure to the Bayesian academic research literature
- An understanding of key differences between Bayesian and frequentist approaches

# Grading

Assignments (lecture + lab):  40%
In-class quick quizzes:  30%
Final project:  30%

*Everyone has a rough week now and then.* The lowest-graded assignment and the two lowest-graded in-class quizzes will be dropped in everyone's final grade calculation. If you wish, you may skip an assignment and two quizzes without prejudice, but please do so cautiously.

*Class participation is important.* As statisticians, clear communication of understanding and uncertainty is something that will be expected of you, and you need to be able to do this verbally as well as in documents. I will keep
```

Two spaces at end of line here!

Rendered text

STSCI 4780 - Bayesian data analysis: principles and practice

Lectures: Tuesdays & Thursdays, 1:25pm - 2:40pm, in Upson 109
Labs: Fridays, 2:55pm - 4:10pm, in Phillips 213

Instructor:
Tom Loredo
Center for Radiophysics & Space Research, and Field of Statistics
620 Space Sciences Building
loredo@astro.cornell.edu
Office hours: Wednesdays, 3pm - 4pm, and by appointment

Teaching Assistant:
Kerstin Frailey
Dept. of Statistical Science
Malott Hall
kef73@cornell.edu
Office hours TBD

Course goals

Provide students:

- A basic understanding of the principles and foundations underlying the Bayesian approach
- Practical experience using basic/intermediate Bayesian methods
- Experience with widely-used tools and software development practices for producing and sharing collaborative, reproducible statistical research
- Exposure to the Bayesian academic research literature
- An understanding of key differences between Bayesian and frequentist approaches

Grading

Assignments (lecture + lab): 40%
In-class quick quizzes: 30%
Final project: 30%

Everyone has a rough week now and then. The lowest-graded assignment and the two lowest-graded in-class quizzes will be dropped in everyone's final grade calculation. If you wish, you may skip an assignment and two quizzes without prejudice, but please do so cautiously.

Class participation is important. As statisticians, clear communication of understanding and uncertainty is something that will be expected of you, and you need to be able to do this

See software setup web page for info on Markdown editors

Links and tables

README.md 1287 Words

```
in places (reflecting its history). Persi Diaconis, an influential
mathematician and Bayesian statistician (and former Cornellian, and
magician!), wrote a wonderful, frank, and very positive review that is worth
reading:

["A Frequentist Does This, A Bayesian That" (Diaconis's review of Jaynes's
PTLOS)](http://www.siam.org/news/news.php?id=81)

I've put several other useful books on reserve; I'll add comments about some
of them here as we get to corresponding material in class.

# Lecture plan

 # | Date    | Topic
---|---------|------
1|  Jan 22 | Course intro; Motivation: Models, measurements, arguments
2|  Jan 27 | Assessing deductive arguments: Propositional logic, Boolean
algebra
3|  Jan 29 | Assessing inductive arguments:  Probability theory
4|  Feb 3 | Key theorems
5|  Feb 5 | Discrete data:  Bernoulli, binomial, beta
6|Feb   10| More counting: Multinomial/Dirichlet, Poisson/gamma; nuisance
parameters
7|Feb   12| Continuous data: Normal distribution, Student's t

After the Feb break, we'll synthesize what we've learned to a general
prescription for inference with parametric models, and then continue with
more sophisticated models---Bayesian counterparts to multi-parameter
conventional regression models.

Next we'll focus on Bayesian computation, culminating with Markov chain Monte
Carlo (MCMC).

With flexible computational tools in hand, we'll explore richer model
structures---*hierarchical Bayesian models* (also known as multilevel models,
or probabilistic graphical models).

At this point you will be defining your final projects.  I have a large menu
of further topics; we'll choose from them based in part on relevance to
student projects.

# Lab plan

For the first few weeks, the labs will operate somewhat separate from the
lectures, aiming to build familiarity with the tools we'll use to implement
nontrivial Bayesian computations later in the course.

 # | Date    | Topic
---|---------|------
1|  Jan 23 | Markdown, Git, GitHub
```

computational implementation of Bayesian methods, but of course a deep understanding of foundations and fundamentals is a great help for practical use. The book is quite polemical in places (reflecting its history). Persi Diaconis, an influential mathematician and Bayesian statistician (and former Cornellian, and magician!), wrote a wonderful, frank, and very positive review that is worth reading:

"A Frequentist Does This, A Bayesian That" (Diaconis's review of Jaynes's PTLOS)

I've put several other useful books on reserve; I'll add comments about some of them here as we get to corresponding material in class.

Lecture plan

#	Date	Topic
1	Jan 22	Course intro; Motivation: Models, measurements, arguments
2	Jan 27	Assessing deductive arguments: Propositional logic, Boolean algebra
3	Jan 29	Assessing inductive arguments: Probability theory
4	Feb 3	Key theorems
5	Feb 5	Discrete data: Bernoulli, binomial, beta
6	Feb 10	More counting: Multinomial/Dirichlet, Poisson/gamma; nuisance parameters
7	Feb 12	Continuous data: Normal distribution, Student's t

After the Feb break, we'll synthesize what we've learned to a general prescription for inference with parametric models, and then continue with more sophisticated models---Bayesian counterparts to multi-parameter conventional regression models.

Next we'll focus on Bayesian computation, culminating with Markov chain Monte Carlo (MCMC).

With flexible computational tools in hand, we'll explore richer model structures---*hierarchical Bayesian models* (also known as multilevel models, or probabilistic graphical models).

At this point you will be defining your final projects. I have a large menu of further topics; we'll choose from them based in part on relevance to student projects.

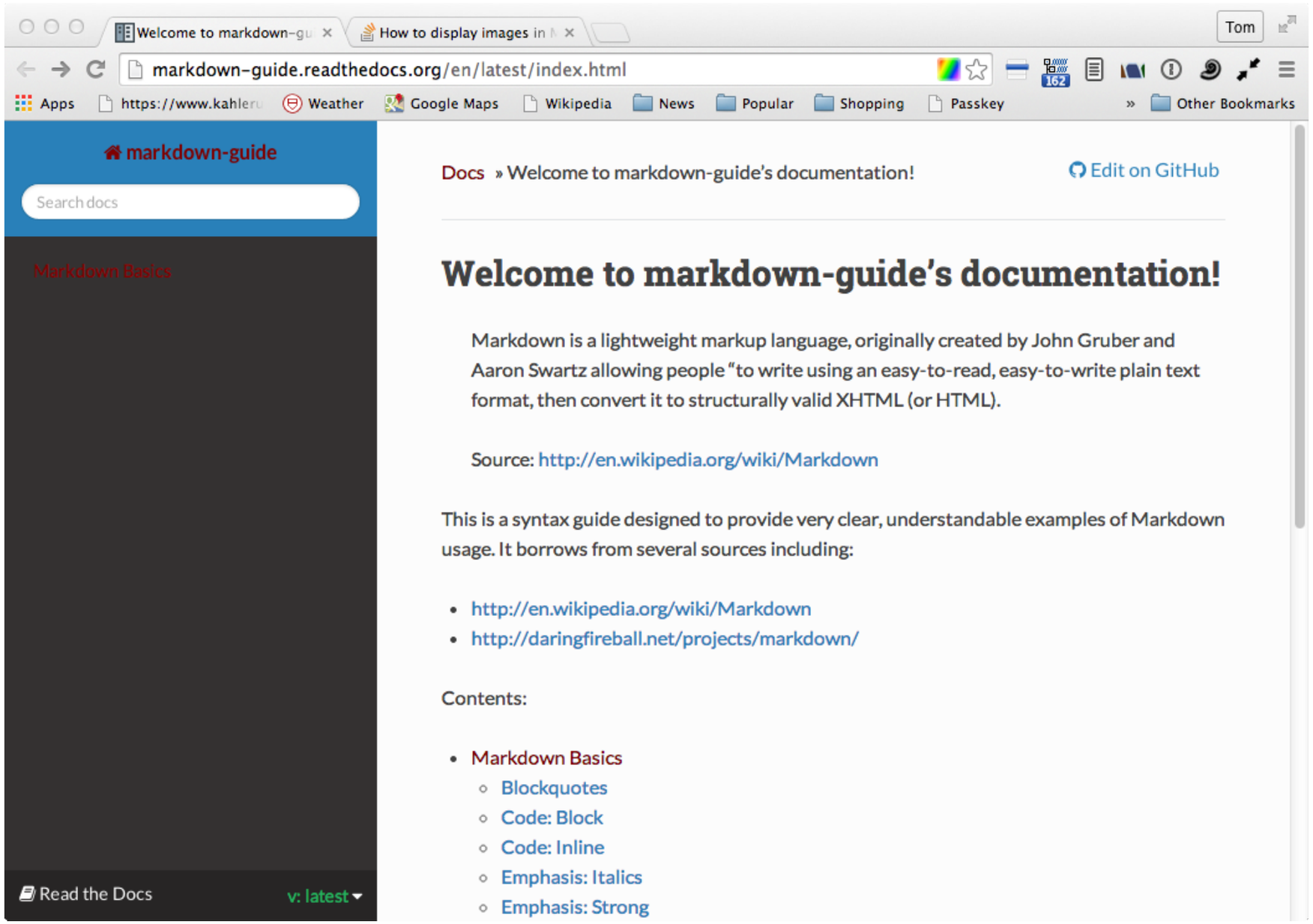
Lab plan

For the first few weeks, the labs will operate somewhat separate from the lectures, aiming to build familiarity with the tools we'll use to implement nontrivial Bayesian computations later in the course.

#	Date	Topic

Markdown documentation

Markdown Guide at ReadTheDocs.org



Many, many other guides and cheatsheets online...

Markdown is not standardized—there are many Markdown "flavors"

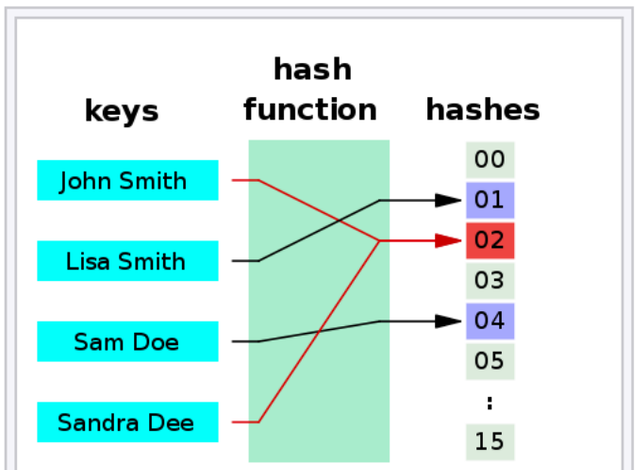
Hashes/checksums

Hash = A "fingerprint" for a sequence of bits/bytes, such as the contents of a file

Hash function

From Wikipedia, the free encyclopedia

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called *hash values*, *hash codes*, *digests*, or simply *hashes*. The values are usually used to index a fixed-size table called a *hash table*. Use of a hash function to index a hash table is called *hashing* or *scatter storage addressing*.



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

Hash functions and their associated hash tables are used in data storage and retrieval applications to access data in a small and nearly constant time per retrieval. They require an amount of storage space only fractionally greater than the total space required for the data or records themselves. Hashing is a computationally and storage space-efficient form of data access that avoids the non-linear access time of ordered and unordered lists and structured trees, and the often exponential storage requirements of direct access of state spaces of large or variable-length keys.

Use of hash functions relies on statistical properties of key and function interaction: worst-case behaviour is intolerably bad with a vanishingly small probability, and average-case behaviour can be nearly optimal (minimal collision).[1]

SHA-1 hash

SHA-1

From Wikipedia, the free encyclopedia

In cryptography, **SHA-1** (**Secure Hash Algorithm 1**) is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value known as a message digest - typically rendered as a hexadecimal number, 40 digits long. It was designed by the United States National Security Agency, and is a U.S. Federal Information Processing Standard.[3]

Linux/macOS:

```
CourseInfo:106$ shasum README.md
4c05239ff718ea4dfc22d1794be17db3cb33d614  README.md
```

Windows:

```
$ certutil.exe -hashfile console.xml SHA1
SHA1 has of file console.xml:
26 e2 16 34 [...]
```

Diffs and patches

diff = A line-based summary of the differences between two files

patch = A diff in a format that can be used to transform one file to another

```
$ cat > HelloWorld1.txt
Hello, world!
$ cat > HelloWorld2.txt
Hello world!

$ diff HelloWorld1.txt HelloWorld2.txt
1c1
< Hello, world!
---
> Hello world!
```

Longer example on Wikipedia

Make a "unified diff" comparing 1 to 2:

```
$ diff -u HelloWorld1.txt HelloWorld2.txt > patchfile.txt
$ cat patchfile.txt
--- HelloWorld1.txt2018-01-26 12:44:34.000000000 -0500
+++ HelloWorld2.txt2018-01-26 12:44:48.000000000 -0500
@@ -1 +1 @@
-Hello, world!
+Hello world!
```

Patch file 1 to make it match file 2:

```
$ patch < patchfile.txt
patching file HelloWorld1.txt

$ cat HelloWorld1.txt
Hello world!
```

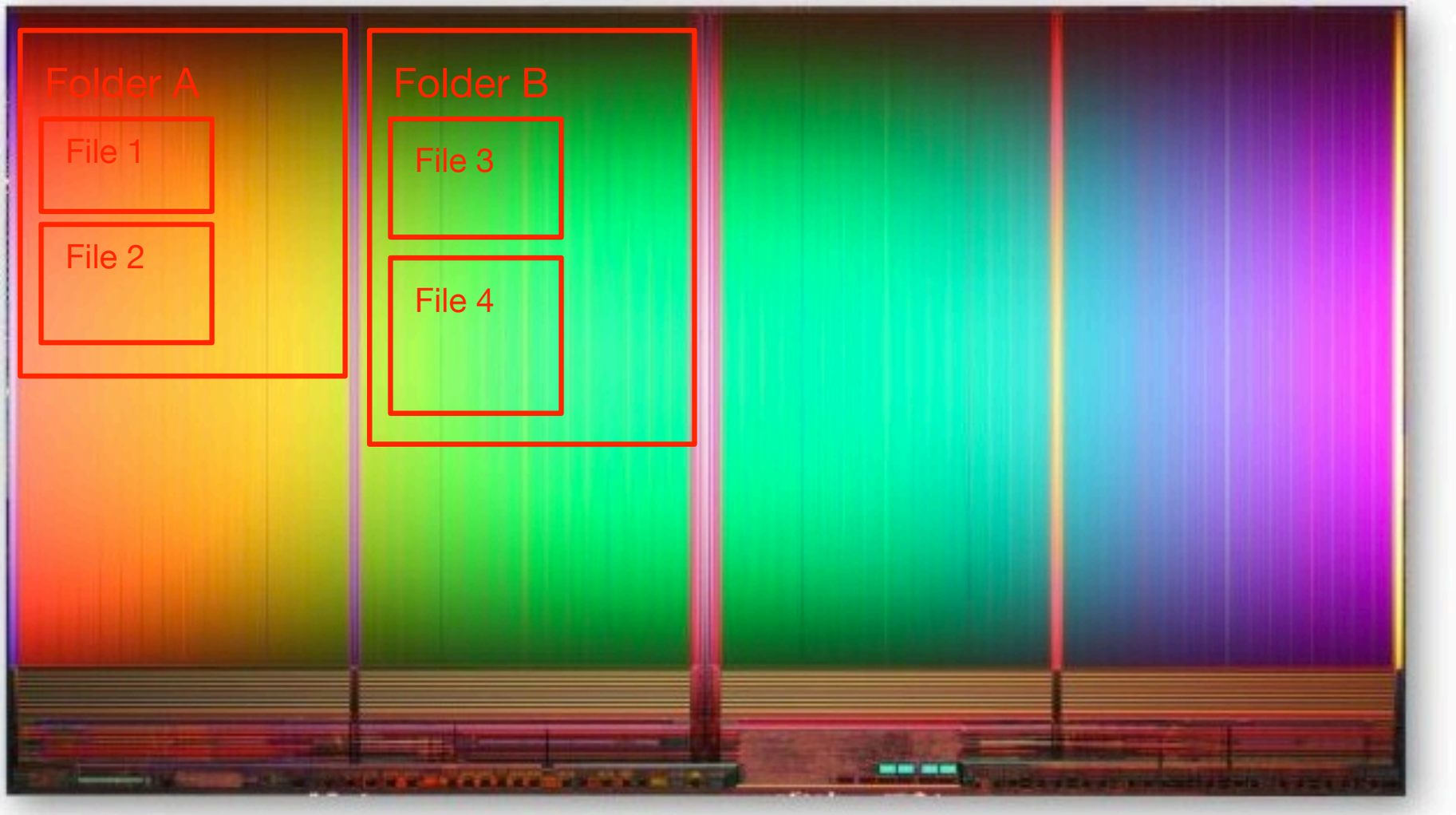
File systems

Your operating system maintains a *file system* for each volume or partition of persistent storage (hard drive, SSD, USB drive...), managing storage and retrieval of data.

The desktop GUI provides an interface to the file system using the metaphor of a file cabinet, containing (nested) folders and documents.

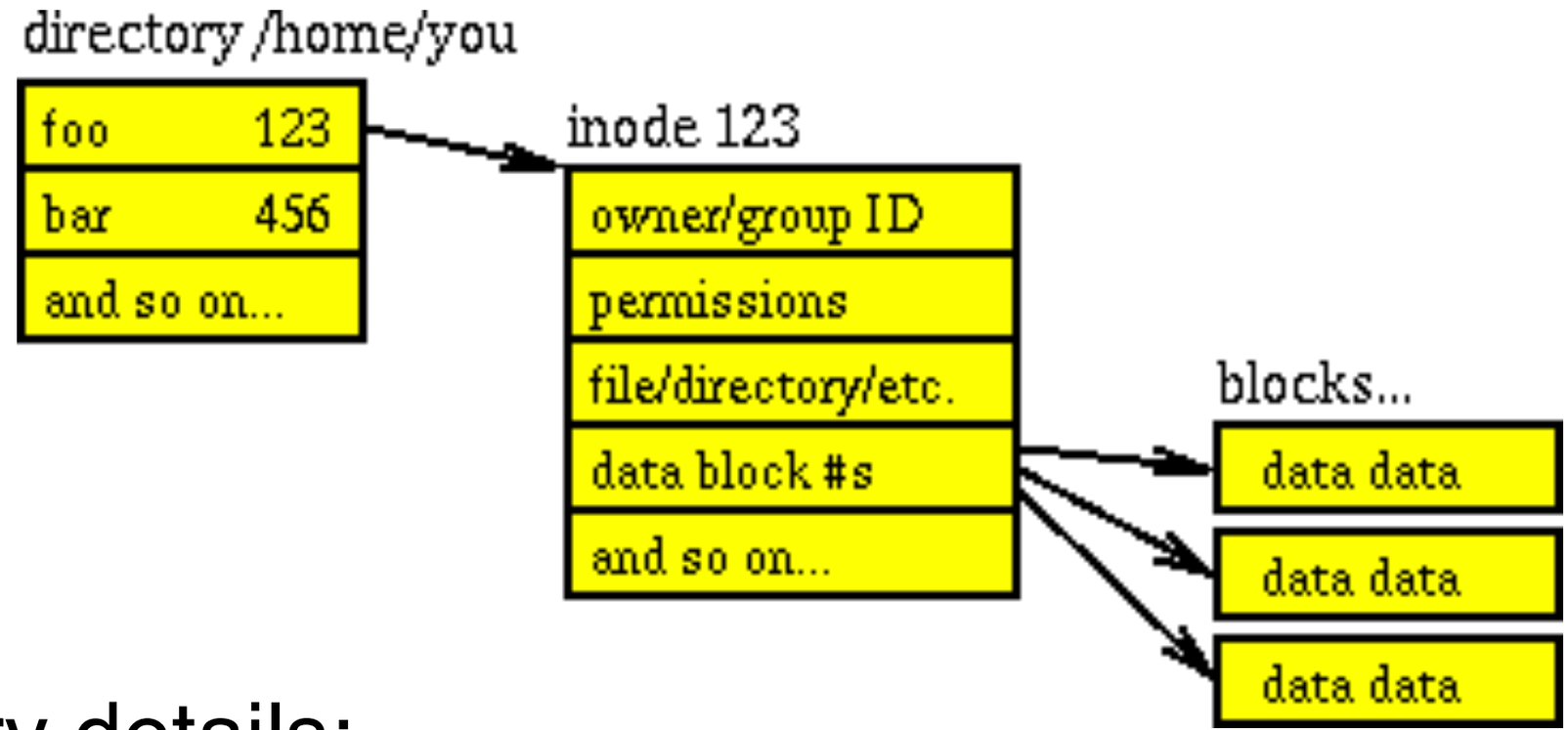
A file system is really a sophisticated and complex database, with little resemblance to a file cabinet!

What a file system is *not*



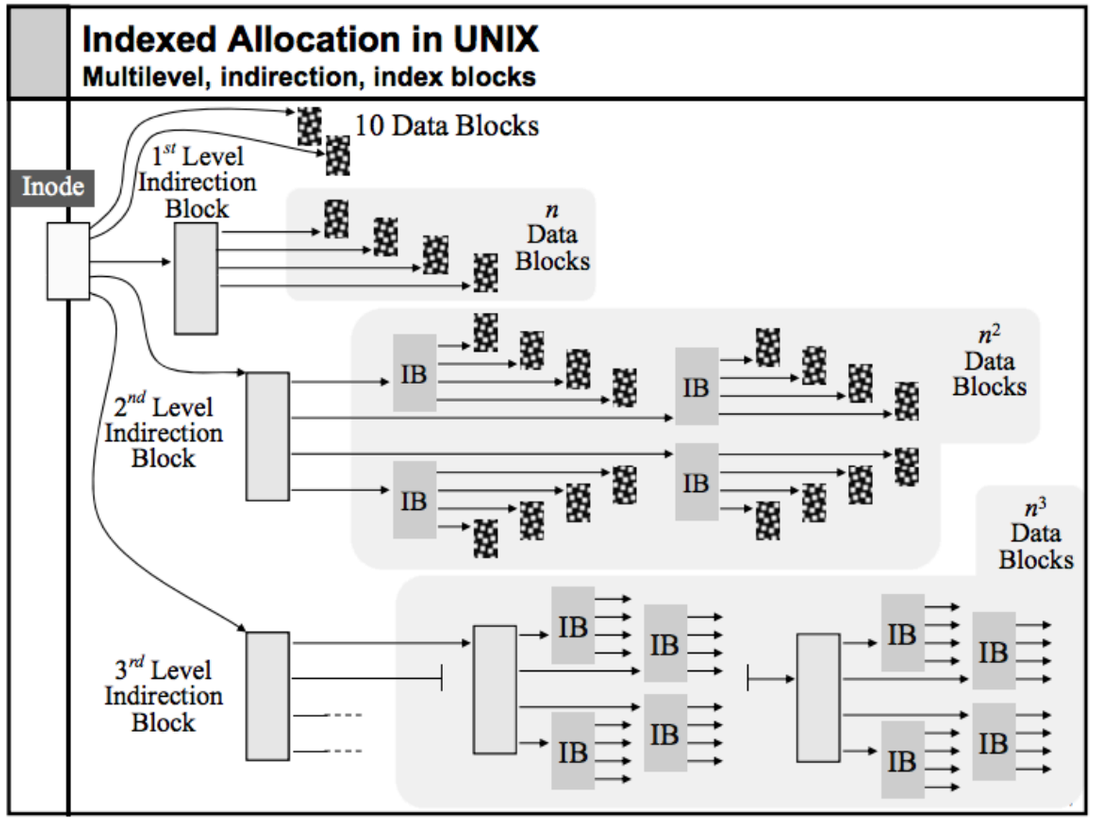
Intel 32 GB NAND flash chip

Outline of a Linux file system:



SuperUser.com

inode gory details:



UTexas OS class notes

Takeaways

- Files are not monolithic pieces of content with names attached.

- The name is separate from the content (e.g., *hard links* assign separate names to the same content). A directory is a special type of file that associates names with content.

- The content is stored in many chunks (blocks) that may be strewn across the media.

- A file system is a complex database, not an electronic file cabinet.

What is Git?

Git is a ***distributed version control system (DVCS)***

- **Version control:** Git lets you have a single, visible working copy of a directory (folder), but maintain a history of snapshots that you can move between.

- **Distributed:** Git enables multiple users to work on a collection of files independently, sharing changes in a collaborative fashion, including a record of changes.

- **System:** Git is a collection of dozens of command-line programs, most of them accessed via the master program, "git".

What's different about Git?

- **Distributed:** Most earlier VCSs required complicated locked checking in and out of files to support collaboration. Git lets users work independently on the same files, providing tools for handling conflicts when merging collaborators' work.

- **Blobs vs. diffs:** Other VCSs maintain a *history of diffs*, changing the repo from one state to another. Git maintains a *history of "blobs,"* chunks of file content associated with a file name. It is essentially a ***virtual file system***.

Two ways to use Git

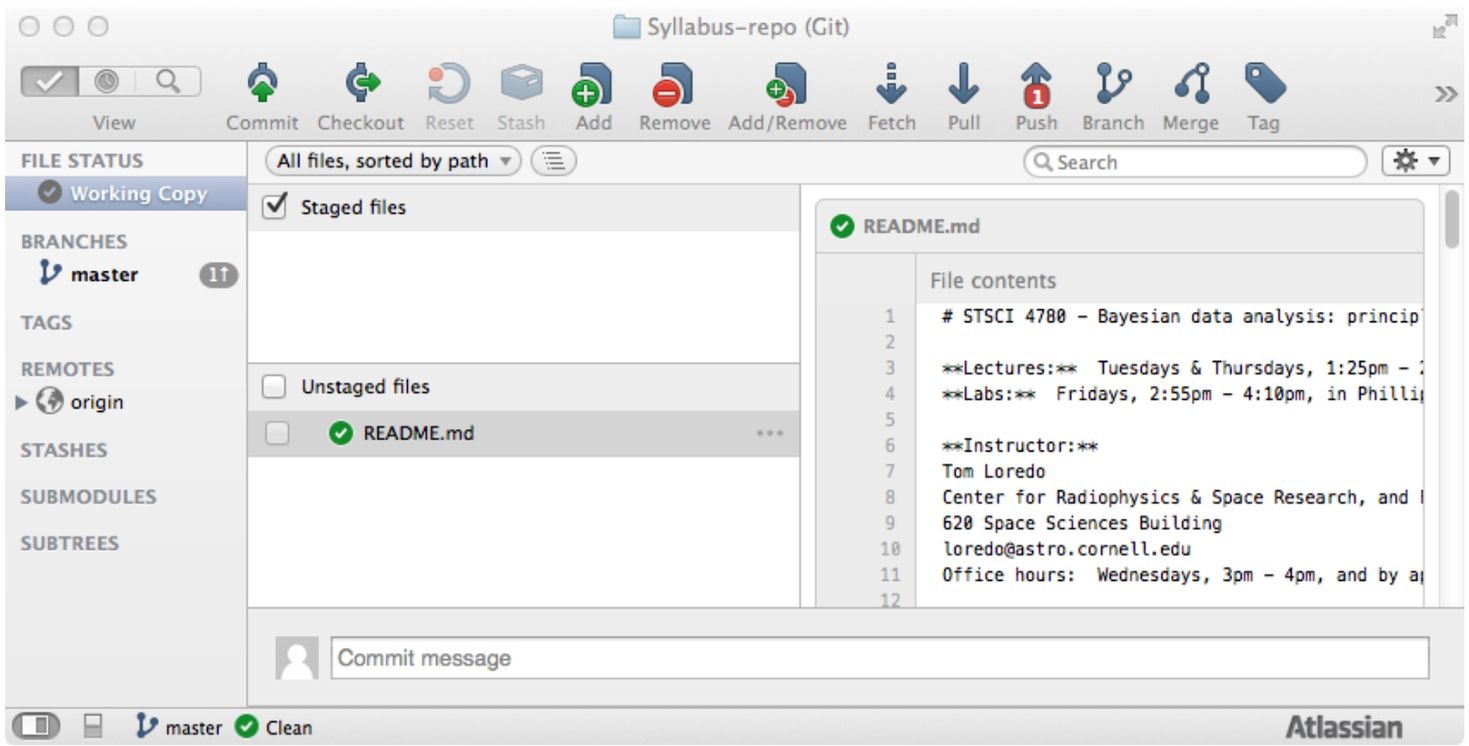
Command line

```
git add README.md
git status
git help commit
git commit -m "First draft of README file"
```

prints useful info, including instructions; use this often!

prints documentation for the command; use it!

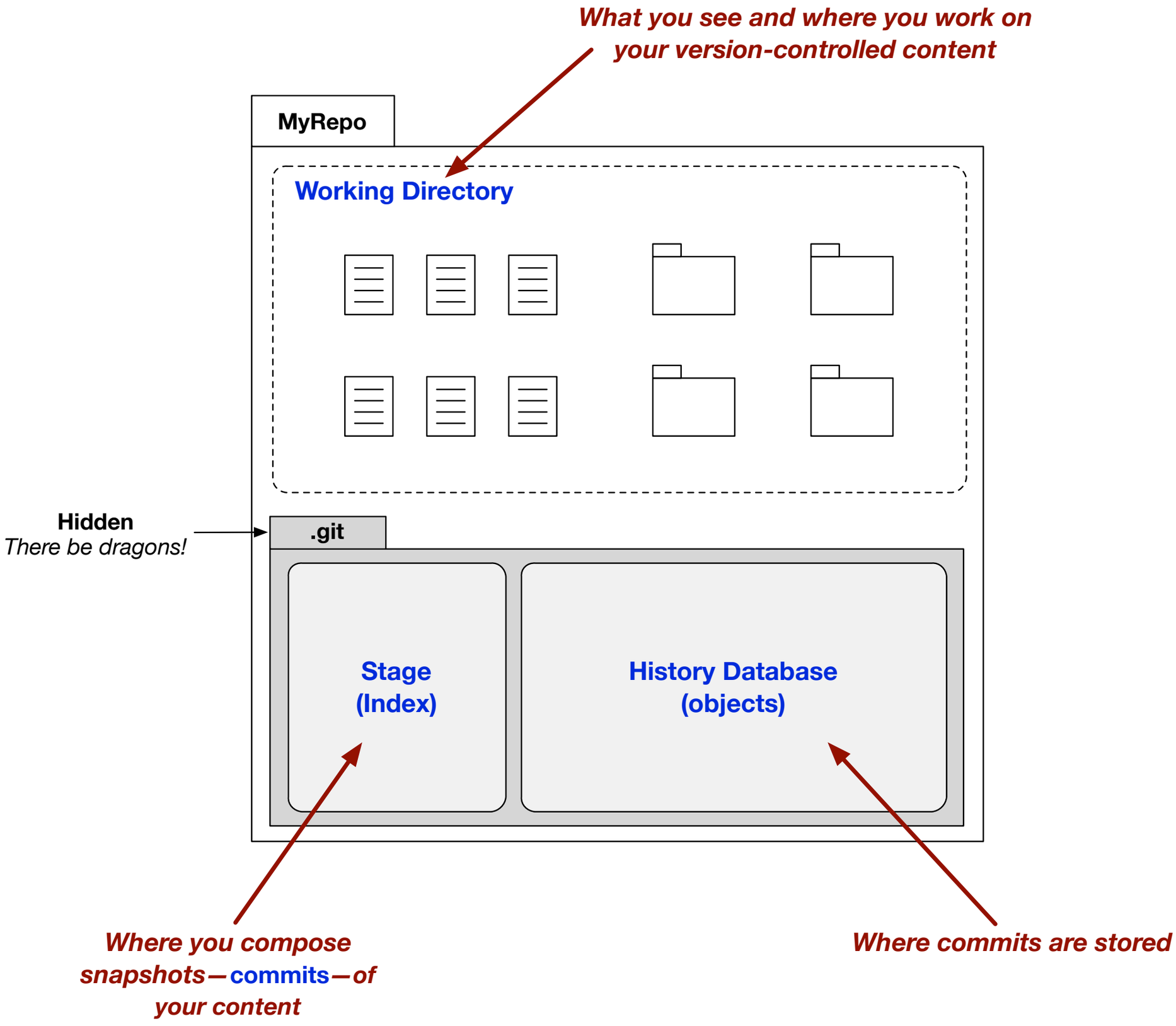
Graphical user interface (GUI)



SourceTree for Mac/Win; many other options:
http://git-scm.com/downloads/guis

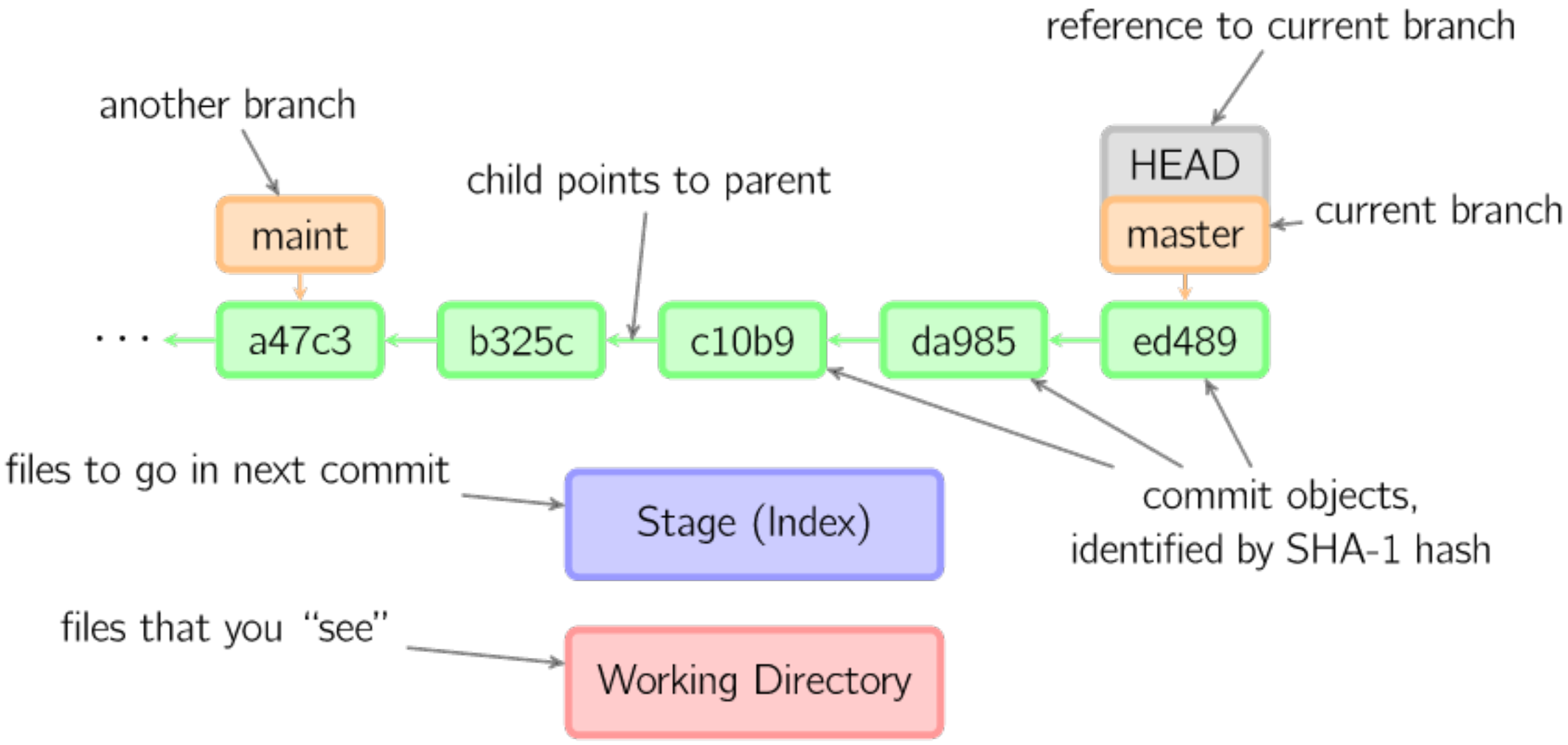
No GUI does everything the Git command line can do; *you need to be familiar with command-line Git*

A Git repository



Commits (snapshots)

- A commit stores a tree data structure associating directory and file names with blobs of content—a snapshot of the WD file system
- Each is assigned a **unique ID** via SHA-1 hash → 40 hex digits
  Looks like: `09fac8dbfd27bd9b4d23a00eb648aa751789536d`
- Each is associated with a **log message** the user writes that should *briefly* describe the content or reason for the commit
- History: Except for 1st commit, each commit keeps a pointer to the previous commit → a **commit graph** (diagram with nodes and links)
- Access commits in multiple ways:
  - *By ID — Usually 1st few digits suffice*
  - *By **branch** — A named pointer to a point in the history*
  - *By **tag** — Special name you assign to bookmark important points in the history*
  - ***"HEAD"** — Fixed name for the current commit on the current branch*



From "A Visual Git Reference"
http://marklodato.github.io/visual-git-guide/index-en.html

Why the Stage?

- Staging helps you split up one large change into multiple commits

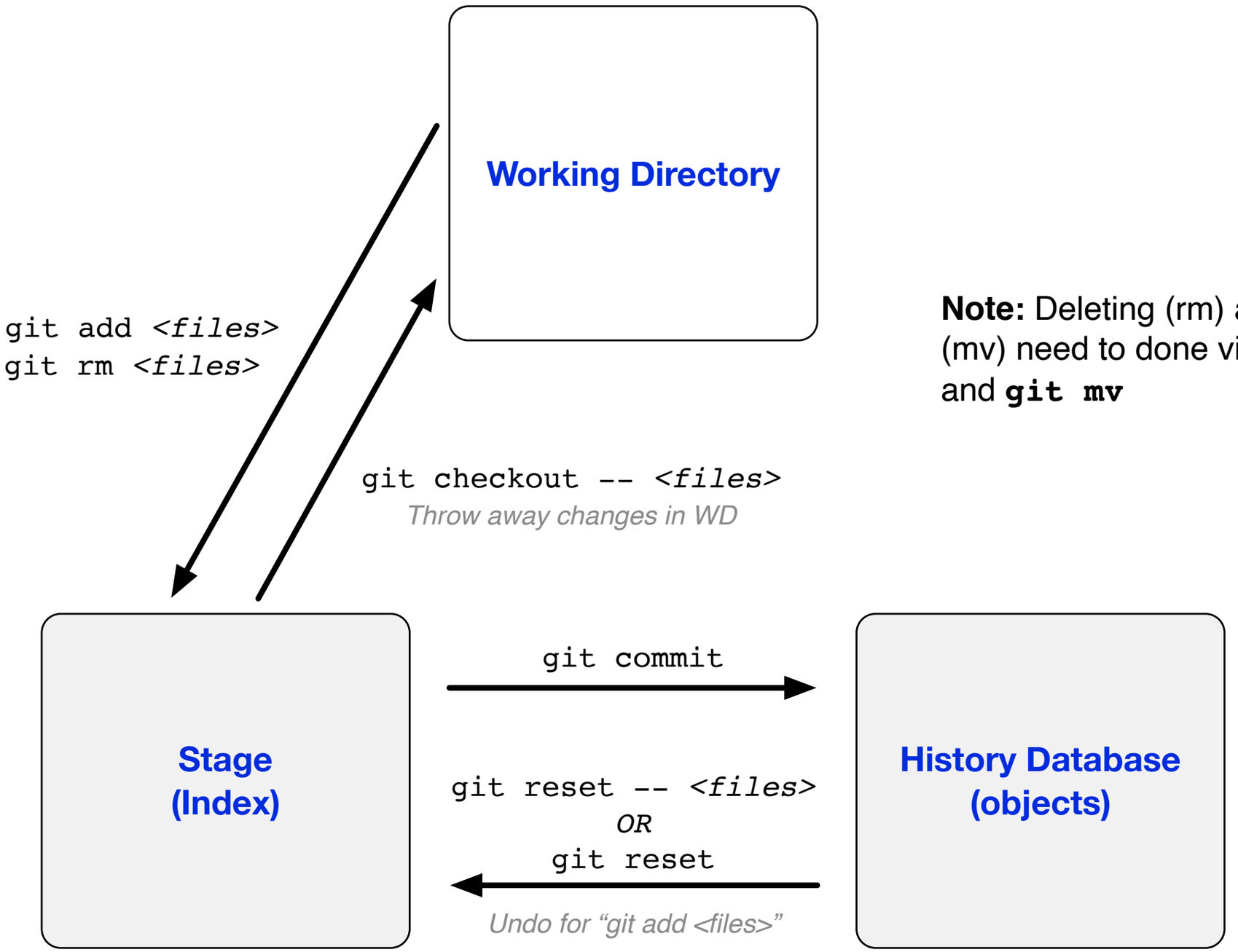
  Example: You are working on several changes (in your working directory), but a colleague needs one of them *right now*. Add/commit the needed files, while you continue working on the others.

- Staging helps in reviewing changes

- Staging helps when a merge between branches has conflicts in just some of the changed files

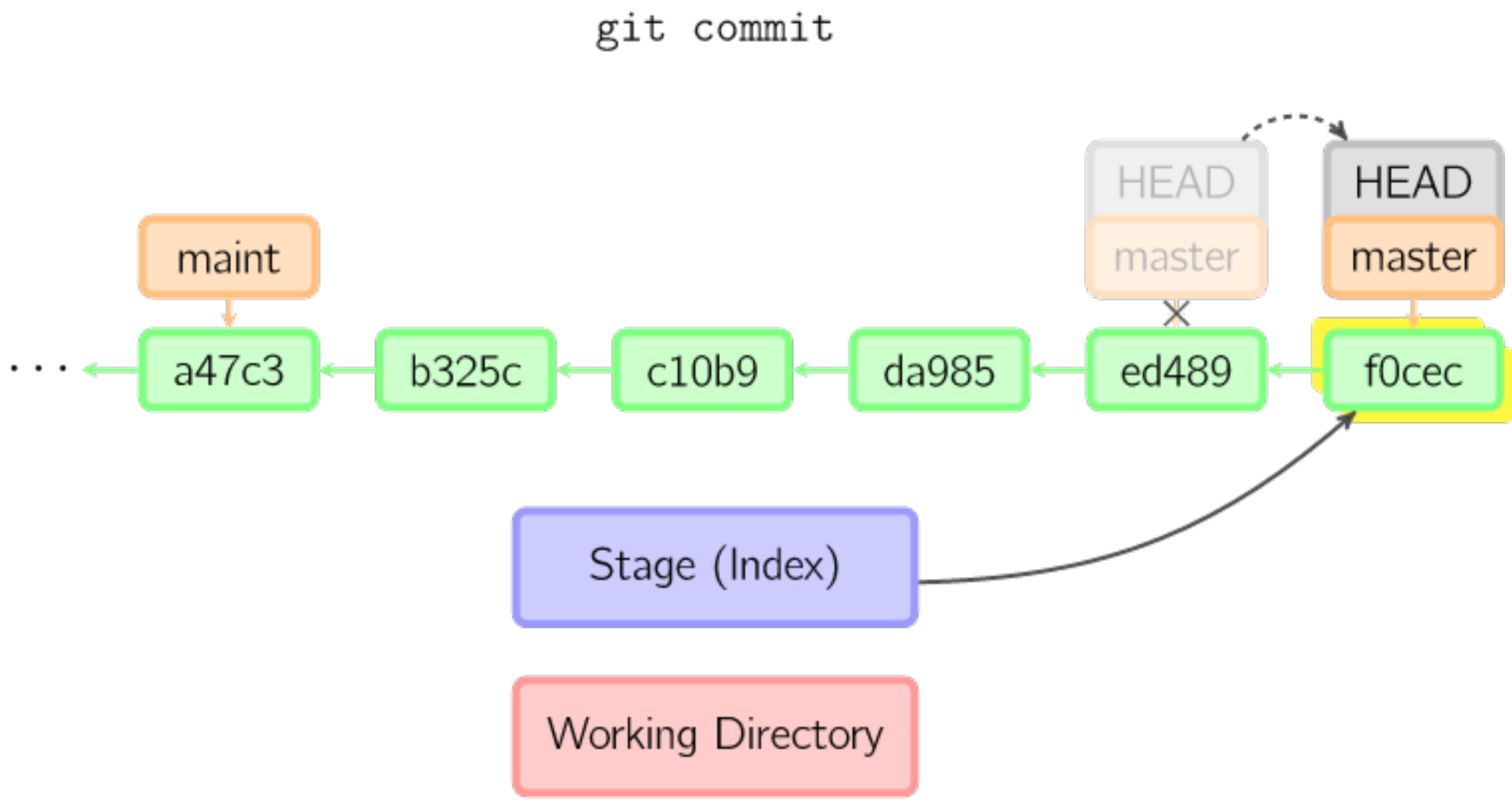
See:
http://gitolite.com/uses-of-index.html
http://programmers.stackexchange.com/questions/69178/what-is-the-benefit-of-gits-two-stage-commit-process-staging

Basic usage



Note: Deleting (rm) and renaming (mv) need to done via **`git rm`** and **`git mv`**

Effect of `commit` on the history



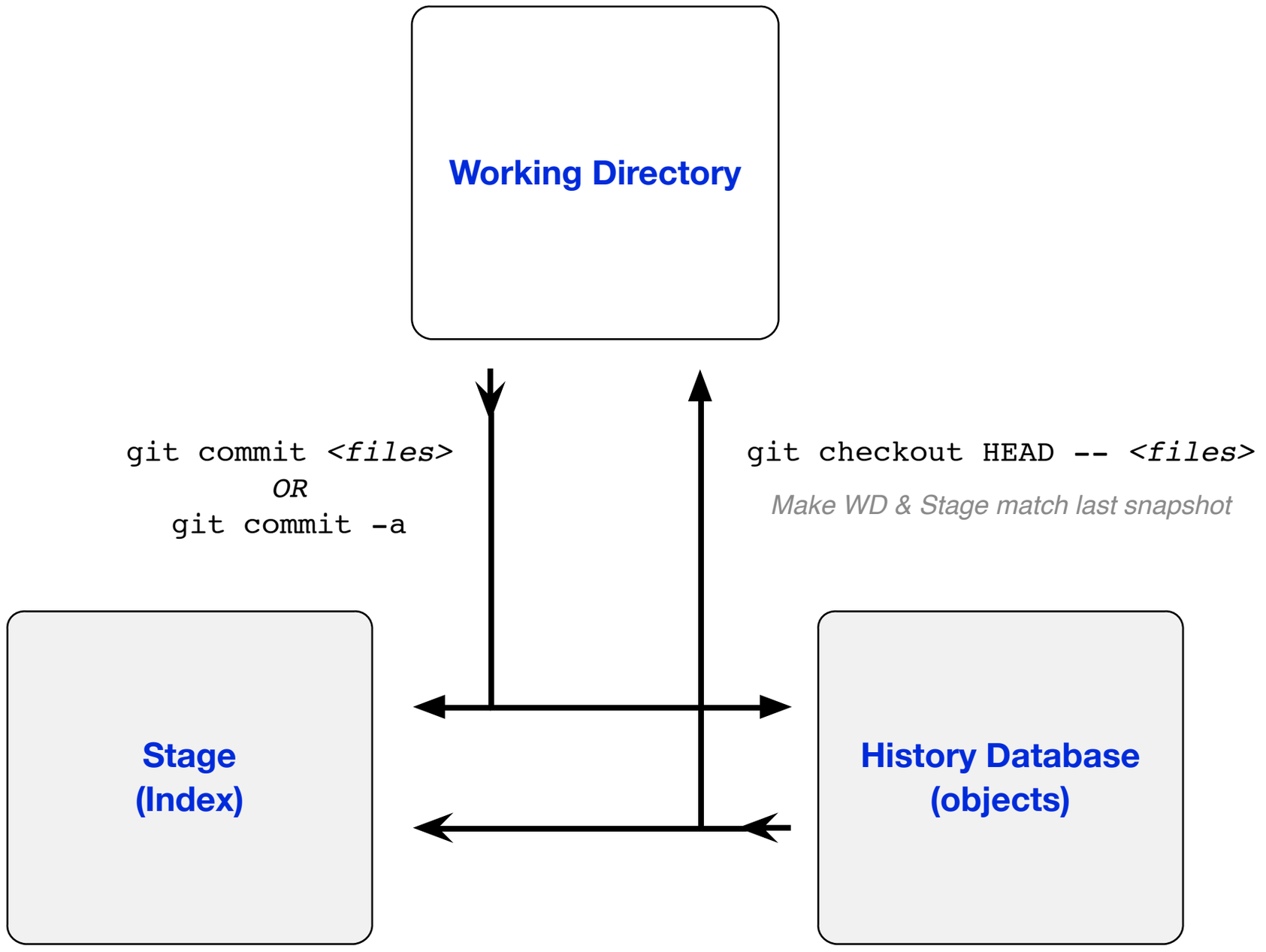
The commit message

- **`git commit`** will start a text editor in your console/terminal window, allowing you to enter a commit message. By default it may use an unfamiliar editor; the choice is customizable. When you save the message and quit the editor, Git completes the commit.

- Most commonly, a short, single-line message suffices, and you can ***use a shortcut*** to avoid invoking an editor:
  `git commit -m "commit message string in quotes"`

  Tip: On Linux/macOS, the default terminal-based editor is `vi`, and it's not obvious how to quit it if you launched `vi` by mistake. To quit a `vi` session, type "`:q`" (the colon switches to command mode, and the `q` command quits).

Shortcutting the stage

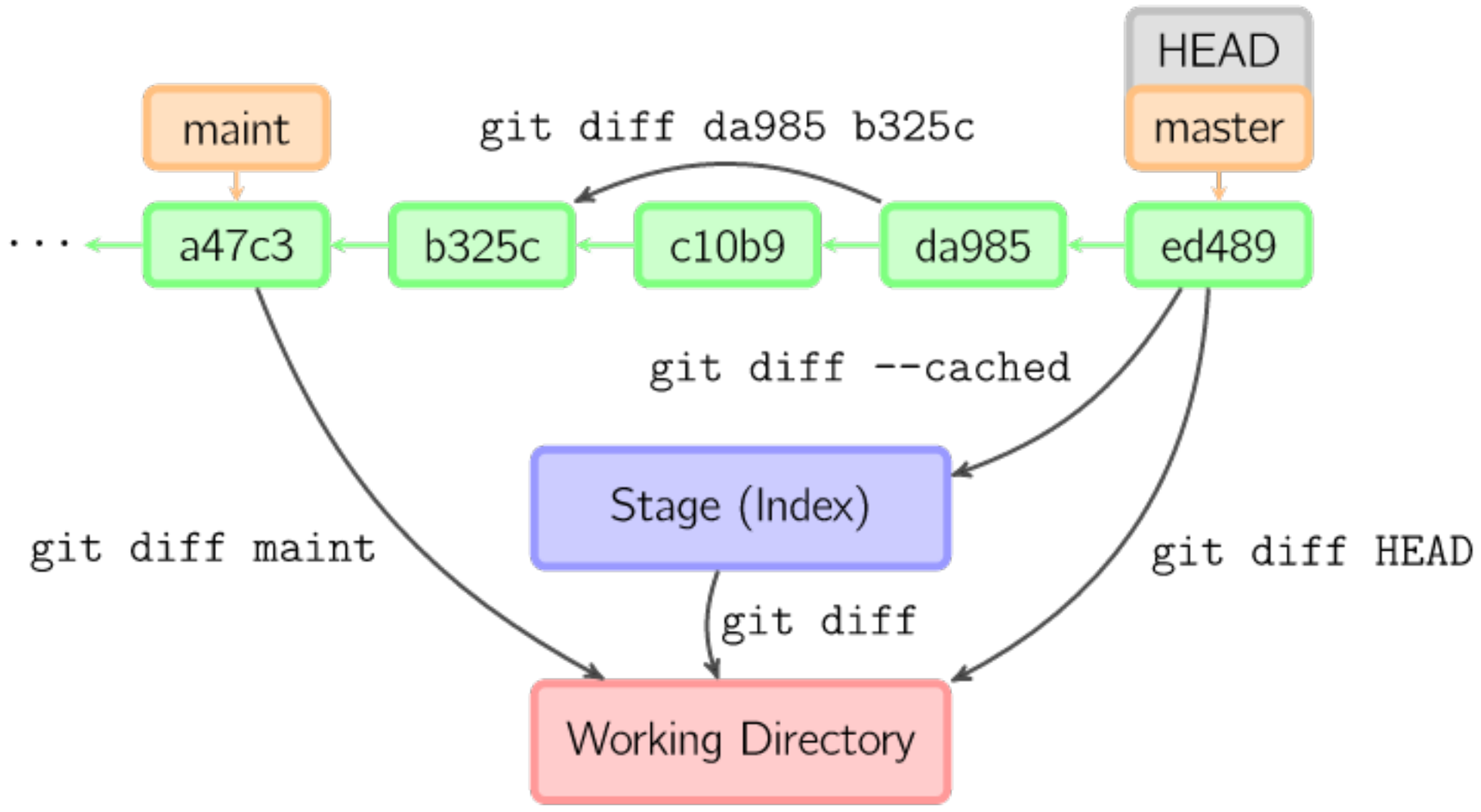


Checking state

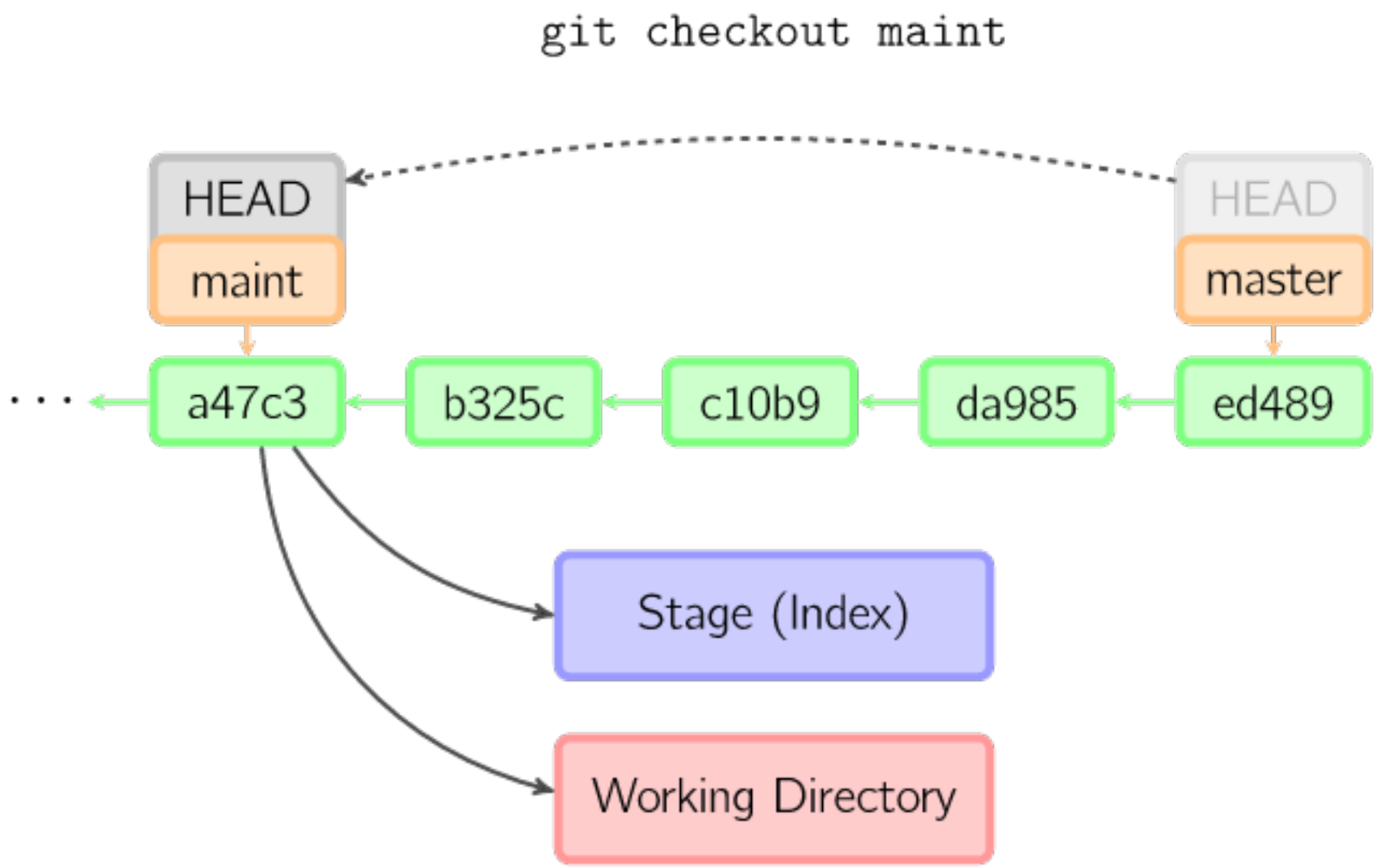
`git status`

Generates a status report — ***use often!***

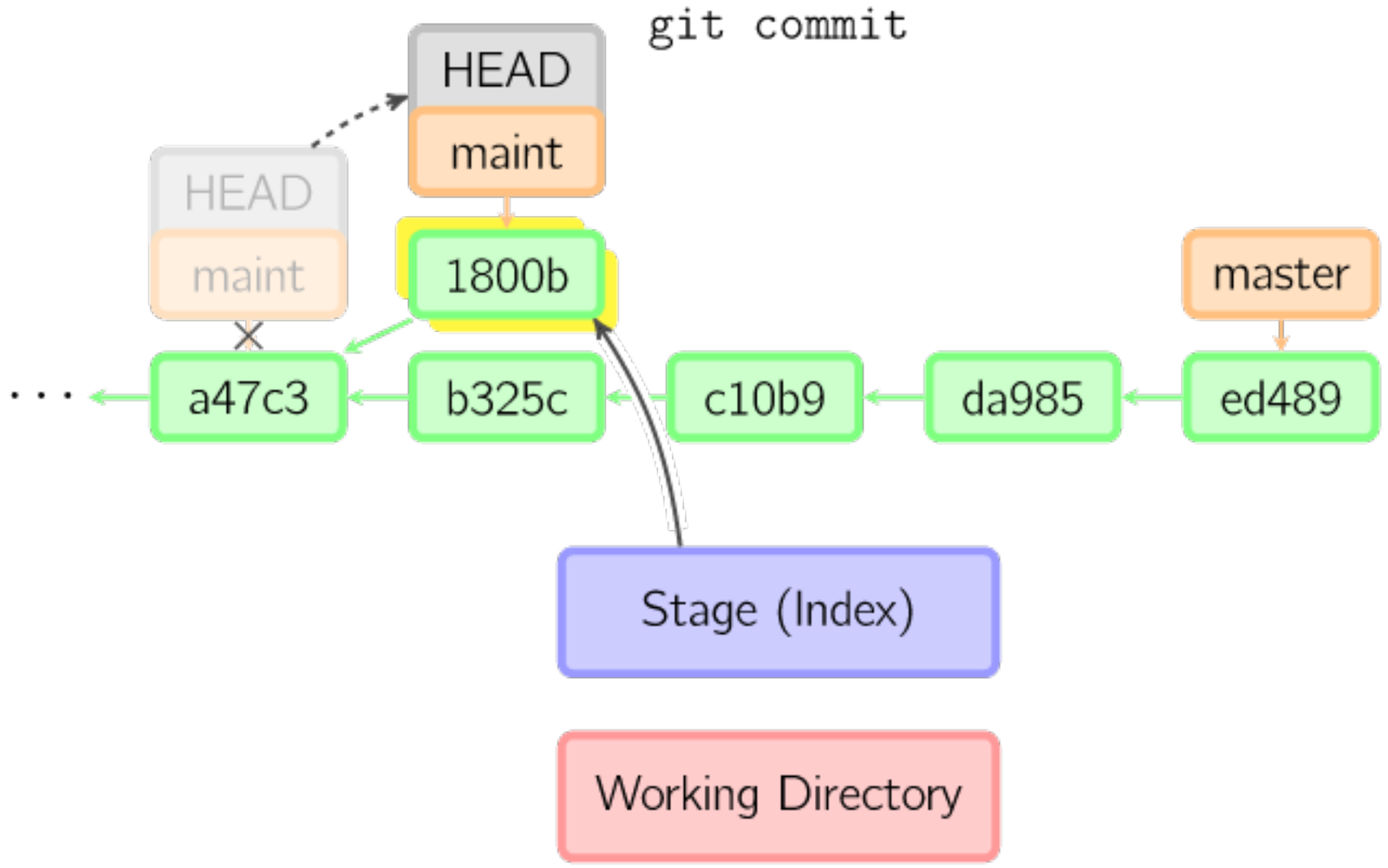
`git diff ...`



Switching to another branch moves HEAD and updates Stage & WD



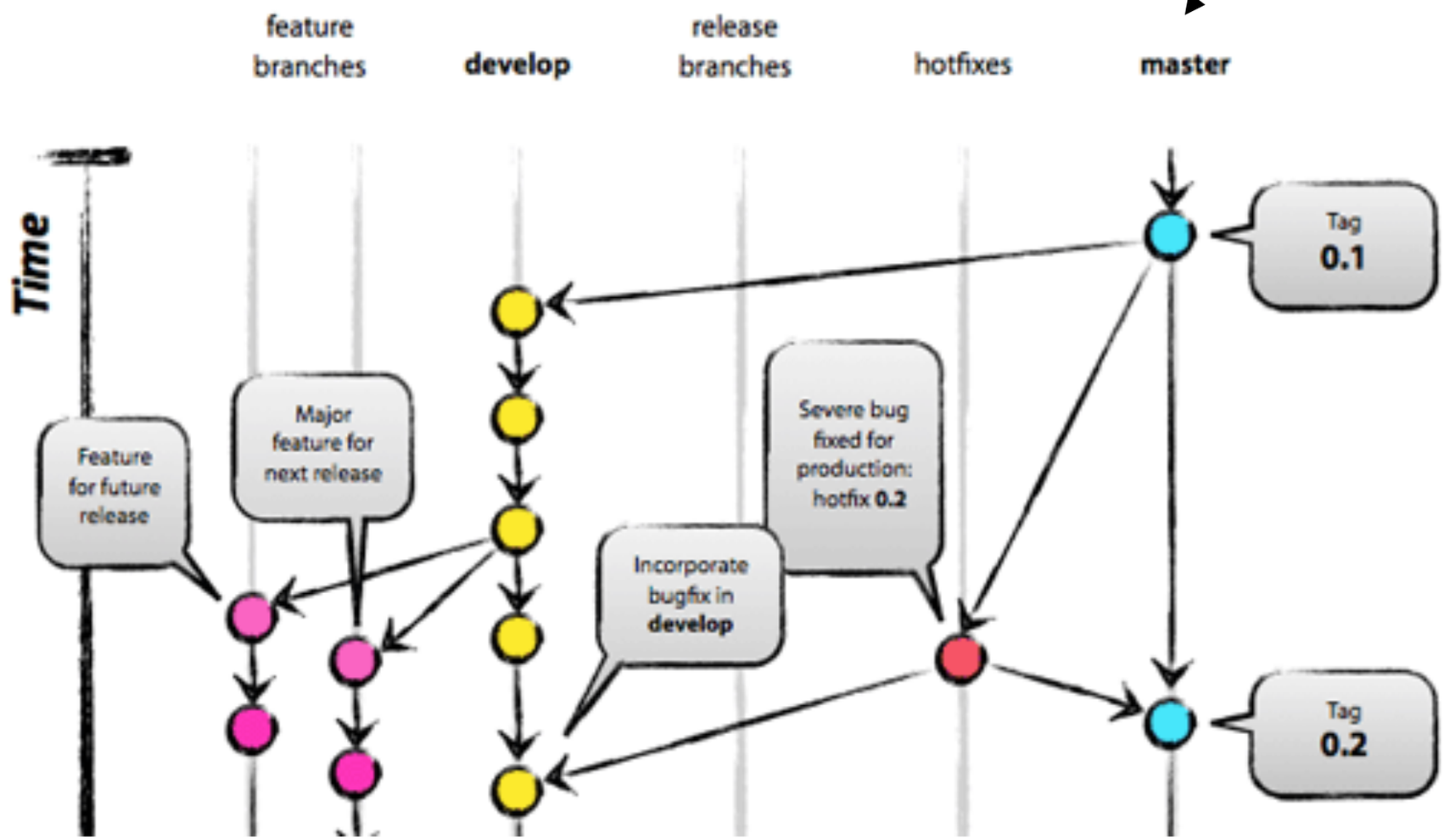
Committing on a branch creates a parallel history



Easily switch between branches (with **`git checkout`**);
merge verified new content with **`git merge`**

Branching strategy

main or **master** is the (default) deployable branch—what you want non-developers/non-collaborators to see



From http://betterexplained.com/articles/aha-moments-when-learning-git/

.gitignore

The (optional) `.gitignore` file tells Git what files to ignore when it checks the repo status to advise you on what to add & commit. E.g.:

- Large data files that won't be edited/changed, or that you don't want copied when your repo is cloned or pushed to another location
- Binary executables (large but easy to re-create)
- Interpreter/compiler output (.o, .pyc...)
- IDE or editor metadata files (.Rproj.user/...)
- Images?

See https://github.com/github/gitignore/ for a collection of example `.gitignore` files.

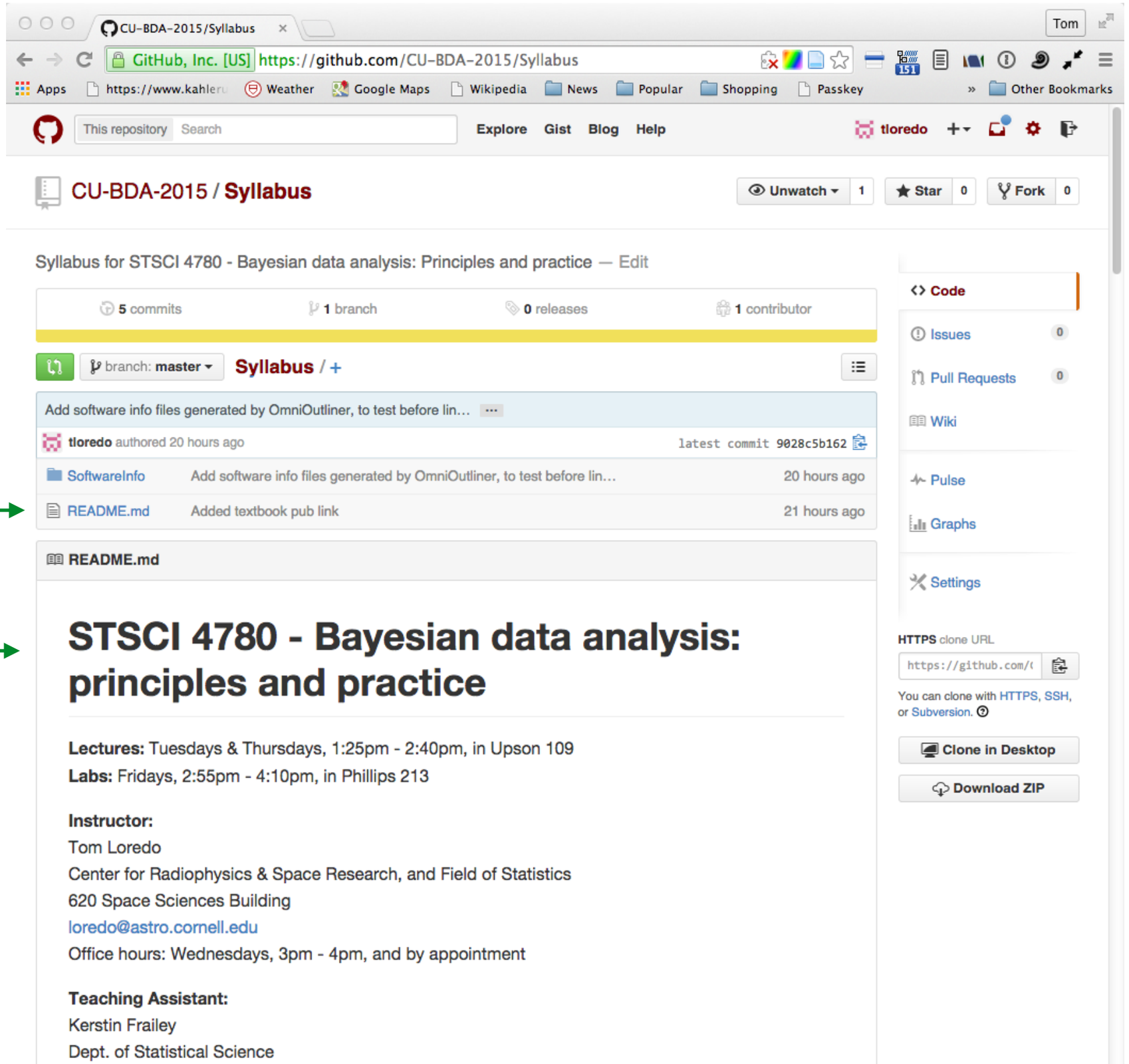
We will provide a `.gitignore` file for your assignments repo; use it!

GitHub

- A web-based Git repository hosting service
  - *Usually just host commits there*
  - *Collaborate with other developers*
  - *Share work with the public*
- Sync GitHub repo with local repo via **`push`** and **`pull`**
- Other features: GitHub Pages, Wikis, issue tracking...
- GitHub organizations: A collection of separate repos associated with teams; provides membership/access control

README handled specially →

Rendered README Markdown →

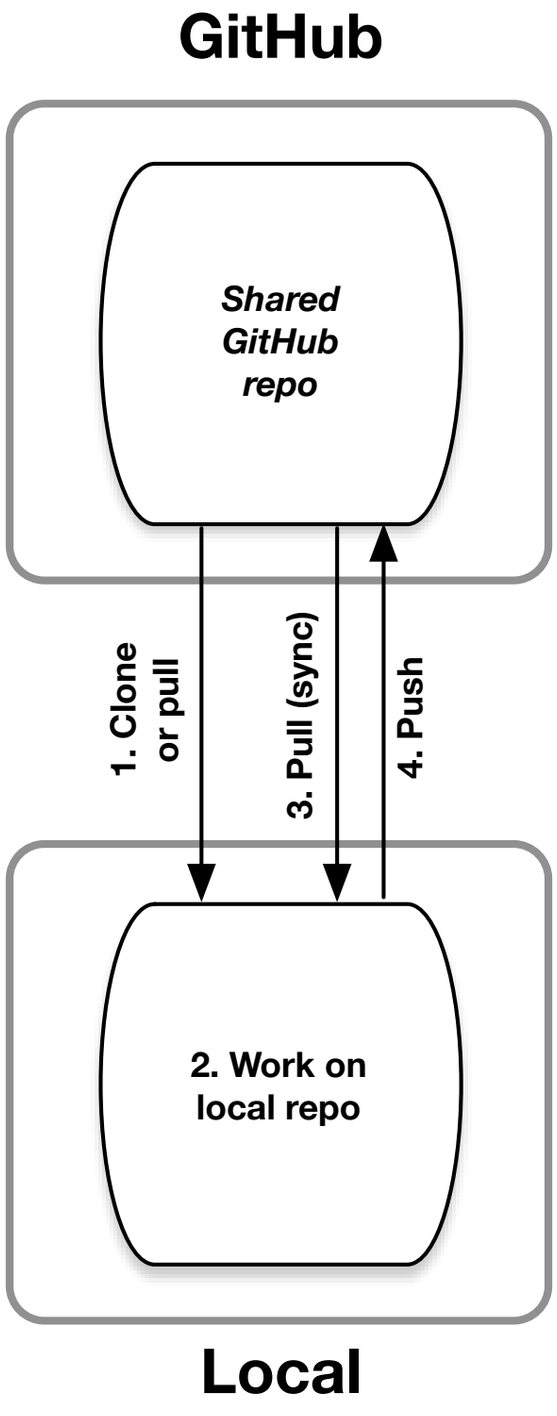


GitHub workflows

Centralized ⟵

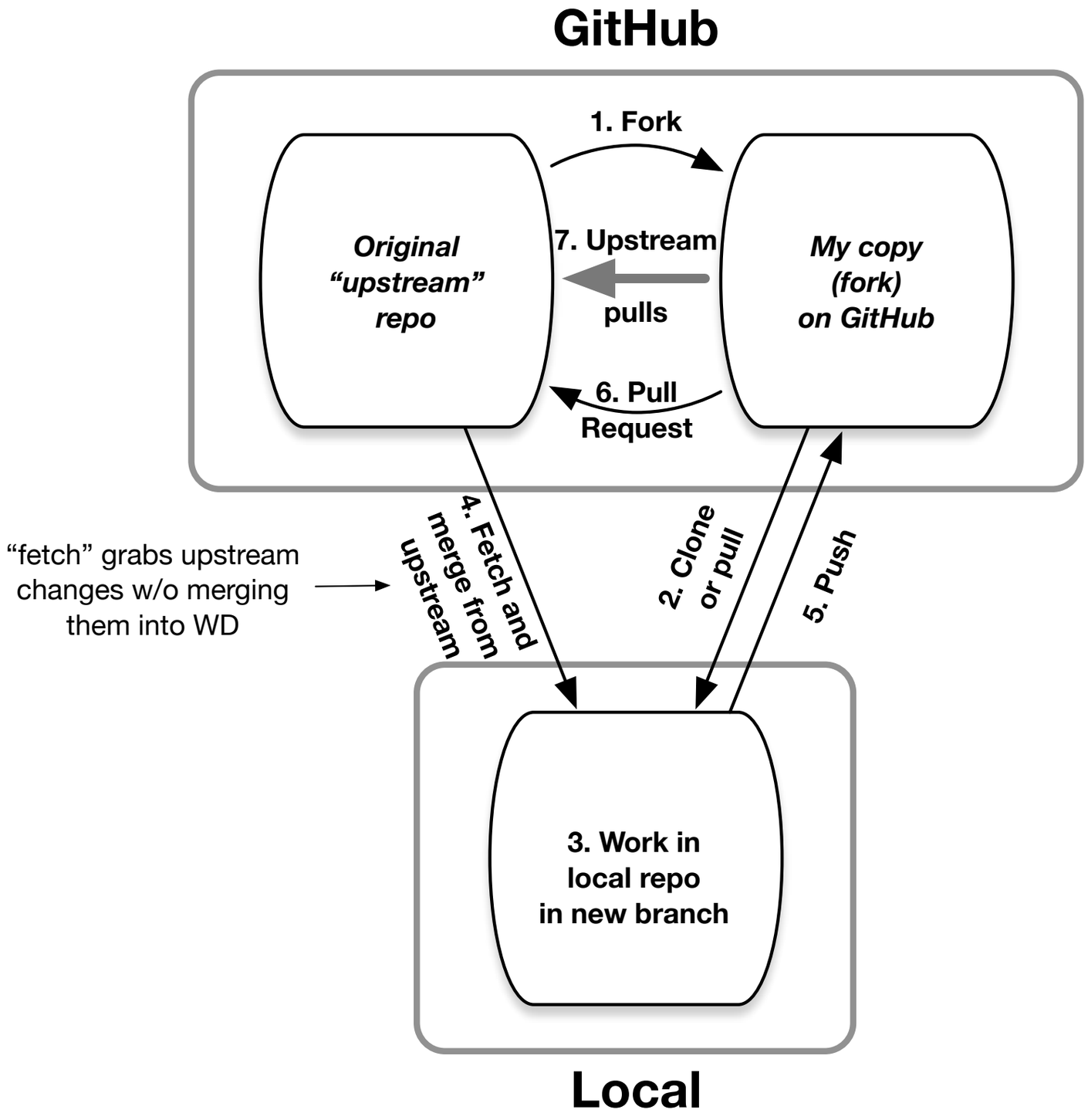
The workflow we'll use for assignments

Always do (1) before you start working on anything that others may have contributed to. *If you have work you haven't pushed back yet, commit first.*



GitHub workflows

Fork-based ← The workflow used for open-source projects

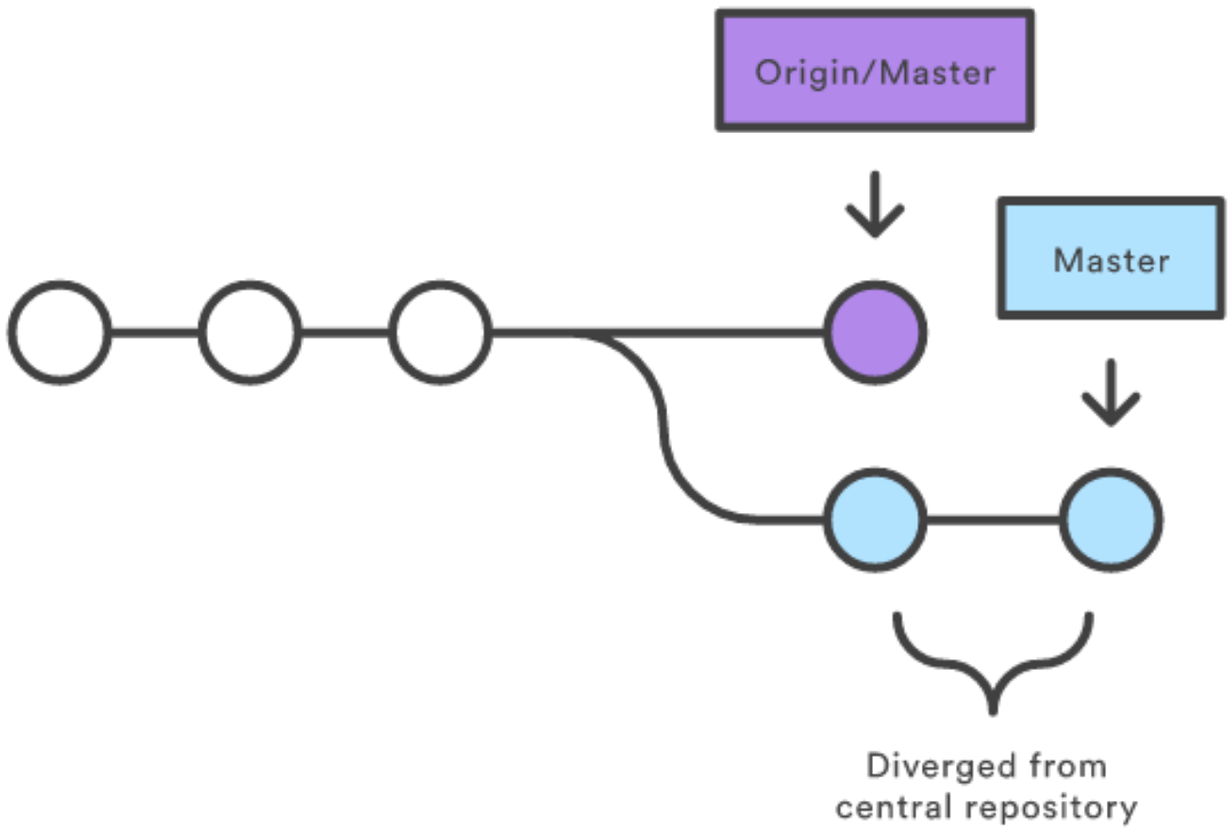


For details see:

- Git/GitHub Walkthrough (CU CS5152)
- Forking workflow tutorial (Atlassian)
- GitHub forking workflow

Pulling: merge or rebase?

If others have pushed changes upstream in the time since your previous pull or clone, then your commits effectively have created a branch (your branch is the local *master* branch, the new content on GitHub is on an *origin/master* branch from your perspective):



Two ways to handle this—*merge policy* vs. *rebase policy*; see:

https://www.atlassian.com/git/articles/git-team-workflows-merge-or-rebase/

https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow

Merge policy

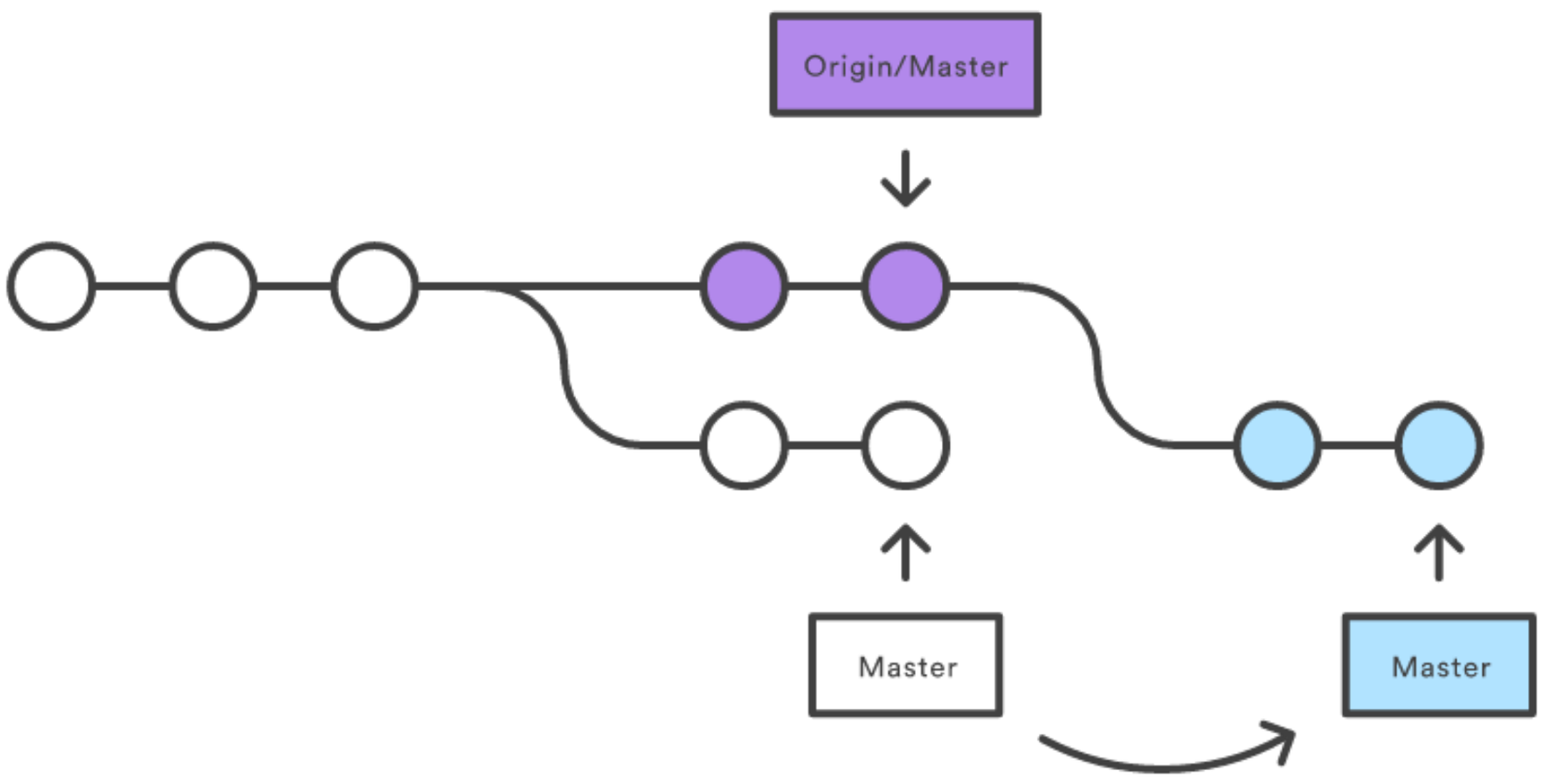
Treat the upstream commits like a genuine branch and **merge** them into the local *master*; this is what `git pull` offers to do by default.* This is the simpler option. This records merges in the repo history, making the history look complicated even if there are no conflicts between changes. But this can be useful for tracing changes. ***This is safest for non-experts.***

* As long as there are no conflicting edits in the pulled content, git pull will merge the pulled content into your history, and launch an editor asking you for a message explaining the merge. *You can safely ignore the request and quit the editor*. Some Git GUIs do this by default.

Rebase policy

Rebase your master branch (alter the history of the master branch), pulling the changes from *origin/master*, and shifting your new commits **after** the pulled commits in the local *master* history. This makes it harder to trace the commit history, but keeps the history linear. *Rebasing will cause problems if origin/master has some of your post-branch changes!* Only do it if none of your changes have been shared with anyone.

```
git pull --rebase origin master
```



See https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow

Getting started with Git

Identify yourself

```
git config --global user.name "YOUR NAME HERE"
git config --global user.email "me@mydomain"
```

Suppress an annoying warning (may not be necessary)

```
git config --global push.default simple
```

To create a repo—two options

- New (empty) repo in the current directory:
  `git init`
  E.g., use this to make a repo for experimenting with Git commands.
- Clone an existing repo in the current directory (e.g., CourseInfo):
  `git clone https://github.com/CU-BDA-2022/CourseInfo.git`

URL grabbed from GitHub page

Git/GitHub Exercise

- **Command-line practice** (in a folder where you'll do your project coding)
  - On GitHub, create a new repo in your own GitHub account (not the BDA org), selecting to use the default README.md
  - Clone the GitHub repo onto your computer, in your practice folder, using: `git clone` *`URL-to-repo`*
  - Inside your practice folder, use a text editor to edit the `README.md` plain text file, changing its Markdown content, and save it
  - Using git, add and commit the new content to your local repo
  - Push your changes back to GitHub (remember to *pull first* to sync with any changes made in the meantime); check on GitHub to verify the push worked
- **GUI practice** (e.g., with SourceTree; you may have to give it your GitHub ID)
  - Connect your GUI with your local copy of your project's GitHub repo
  - Pull any pending changes
  - Make some changes to your `README.md` file, and save it
  - Use the GUI to add and commit your changes
  - Push the changes back to GitHub (pull first to sync!)

Assignment

Due: Thursday, 3 Feb, 11:59pm (i.e., end of Thursday)

- Create a new, empty repo **in our GitHub org** (not in your personal GitHub account); name it **`me-BDAOrg`** where "me" is your *CU NetID* (it would be "tjl9-BDAOrg" for me). Accept the default "Private" setting when you create the repo. *This will be your repo for all of your BDA course assignments,* with each future assignment in its own directory.
- Create a directory on your computer where you'll maintain a local copy of your repo (and perhaps other course repos); start a command-line session and make this your working directory (e.g., "`cd`" into it).
- Clone the GitHub repo onto your computer by grabbing its URL from the GitHub page and using it like this:
  `git clone` ***`PASTE-URL-HERE`***
  You may get a warning that the repo is empty; it's a warning, not an error, so you can ignore it.
  Note that cloning *creates a new directory* in your current directory for the repo; you will work inside that directory. Feel free to move it after cloning; that doesn't affect its connection to GitHub.
- Create a `README.md` file in the clone's working directory using the text or Markdown editor of your choice, or edit the default README.md if you asked GitHub to create one.
- Write a ***short*** (about 1/2 to 1 page if printed) description of yourself and your motivation for taking this course, including:
  - Your name, major, and student status (year)
  - A brief description of a data analysis problem or class of problems that interests you. This could be a problem from a lab course, a thesis project, academic literature in your field, or from the public media.
  - Include one image that shows an example of data pertaining to the identified problem. Copy the image itself into your repo, and use it in your Markdown text. See the GitHub Markdown Guide for quick instructions, or the GitHub Markdown Specification for gory details (the "`PythonForBDA.md`" file in the LabResources repo provides an example).
  - As you work, commit your work into your local repo multiple times, when you reach natural stopping points (see note on next slide).

- **Note:** Once you've made your first commit in the initially empty local repo, git automatically creates the default ***main*** branch. It does not yet exist back on GitHub. If you run `git status` (as you should, often!), you'll get a message that "the upstream is gone." Ignore it; it's just cryptically stating the obvious that what you just created locally is not yet back on GitHub. Once you push your revised repo back to GitHub, `main` will exist there and the message won't reappear.

- If you are using a plain text editor, you won't be able to see the rendered `README.md` file until you commit it and send it back to GitHub. To do so, you need to **`push`** your repo back to GitHub (you'll be prompted for you GitHub login):
  `git push`
  If you are using a Markdown editor, you won't have to push until you're finished.

- When you are finished, after your final commit, "bookmark" that commit with an ***annotated tag***:
  `git tag -a Submitted -m "Submitted assignment"`

- Push your final tagged state back up to GitHub. By default, git does not push tags, so you have to tell it to do so; this is to make sure you really want to share the tag with *everyone* (vs. just using it for your own bookkeeping):
  `git push --follow-tags`

- Verify the project on GitHub:
  - Go to your repo and make sure the `README.md` file appears as it should.
  - Make sure the tag was pushed: Near the top of the GitHub repo page there will be a line reporting the number of commits and branches. It should also say there is 1 "***release***"—GitHub treats tags specially, as a version of your project you want to release to users. If you click on "release" you'll find GitHub has created an easy-to-download compressed archive of your project's tagged working directory.

Grading: This assignment is worth 3 points
3=All requirements met; 2=One or two notable shortcomings;
1=Appears to be half-hearted attempt; 0=Not handed in