

Git, GitHub, Software Licences, and the World of Open-Source Software

CS50 for JDs, Winter 2023
With Inno Munai

Agenda



- **Version Control Systems**
- **Git**
 - Repositories
 - Commits
 - Branches and Merges
- **GitHub**
 - Forking and cloning
 - Pushing and pulling changes
 - Pull requests
- **Software Licences and Open-Source Software**

Scenarios

Scenario 1

Working on a CS50 lab assignment — say `cases.py`.

You're able to get some portions of the lab working perfectly (you can open the file and read/print the lines/rows).

However, you are confused on how to print the details of the case(s) that the user requests.

While attempting to make more progress, you accidentally mess up the working code that you already had!

In hindsight, what could you have done to prevent this?

Scenario 2

Two software engineers want to team up to build a website.

They are trying to decide how to split up tasks.

One really wants to design the home page, including the fonts, styles, colors, text, etc.,

While the other really wants to implement the sign up and login features, also on the home page.

How can we deal with this?

Version Control

Version Control Systems (VCS)

Version Control Systems (VCS)

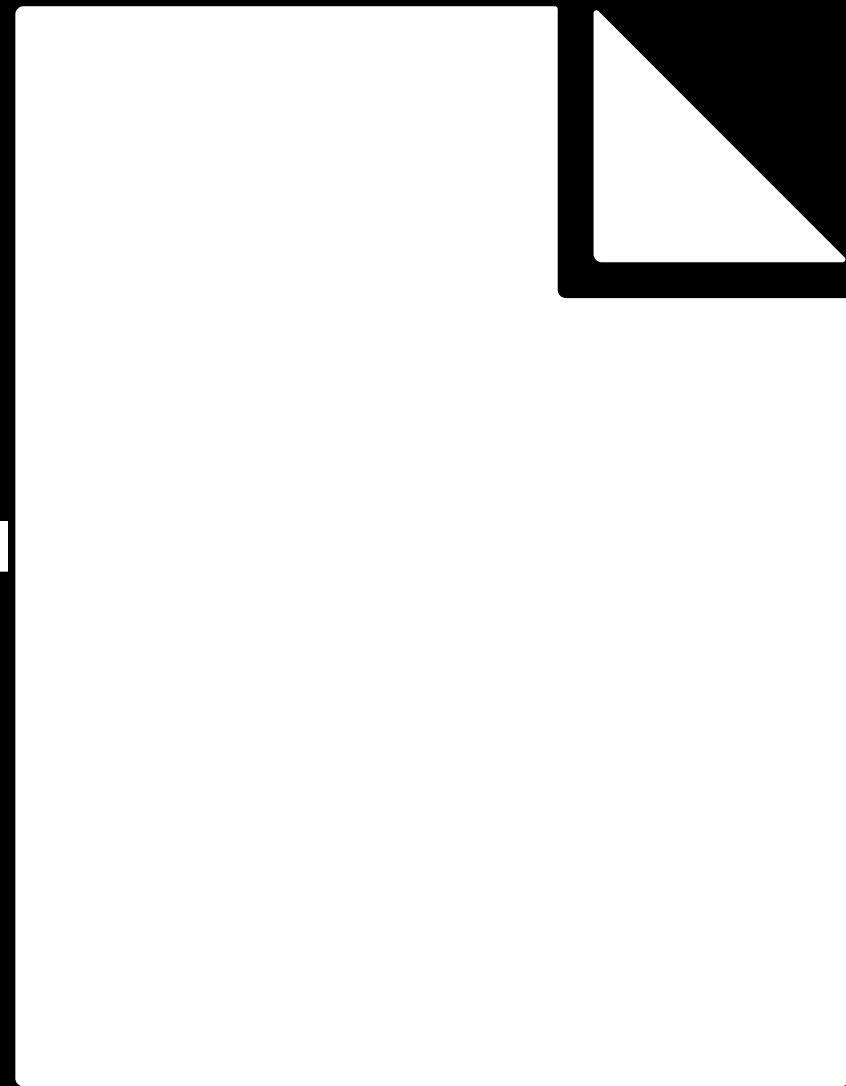
- Keep track of changes to code
- Testing changes to code without losing the original
- Synchronize changes between people
- And more!

Keeping track of changes to code.

Keeping track of changes to code.

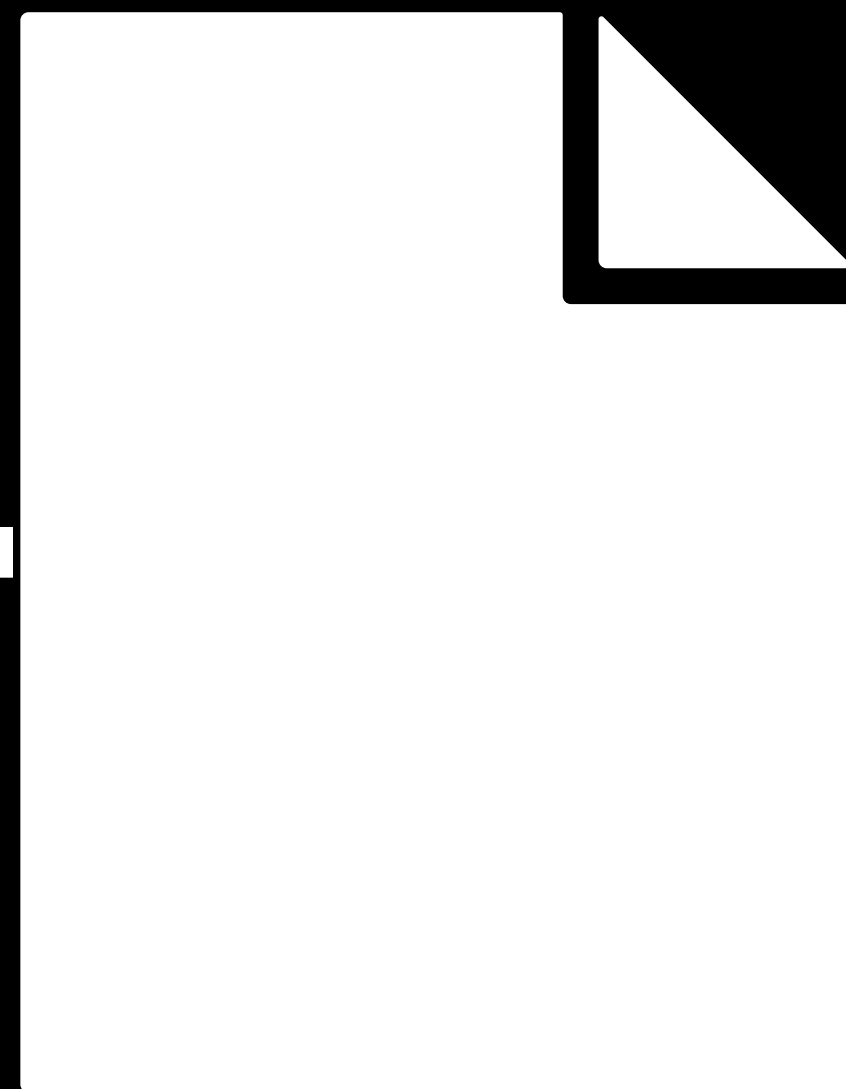


Keeping track of changes to code.

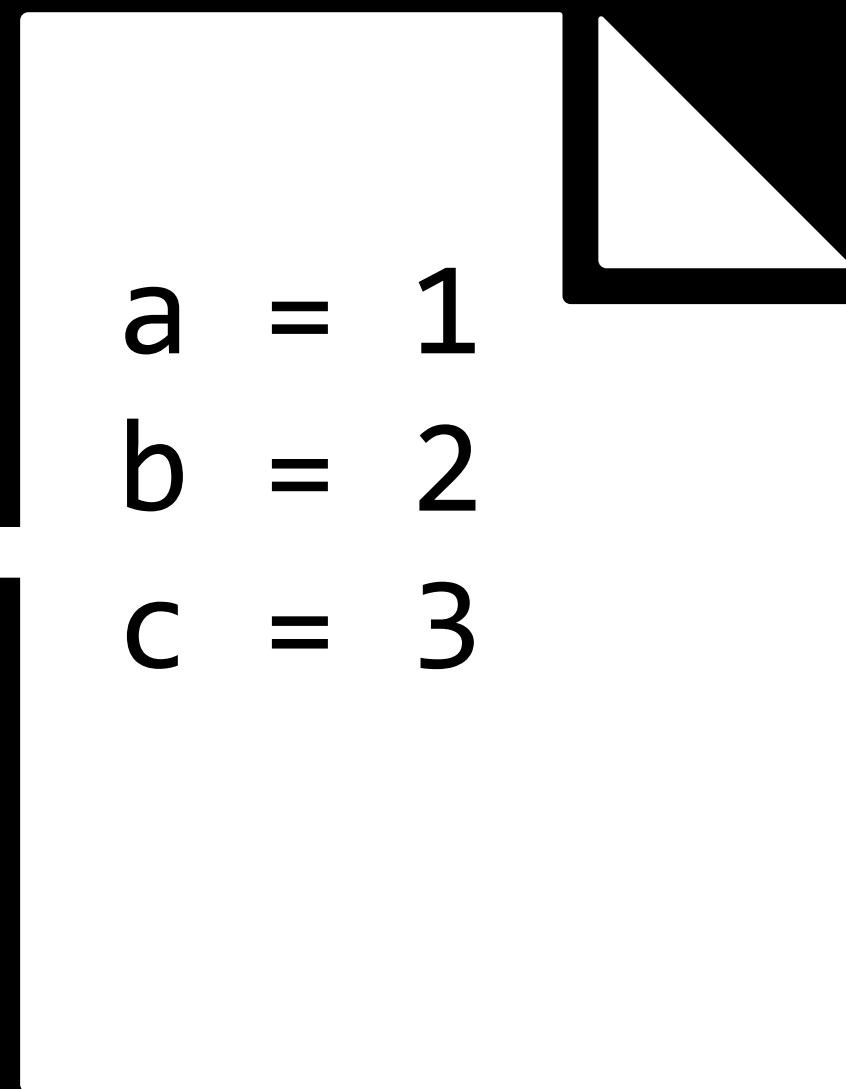


“Create a file”

Keeping track of changes to code.

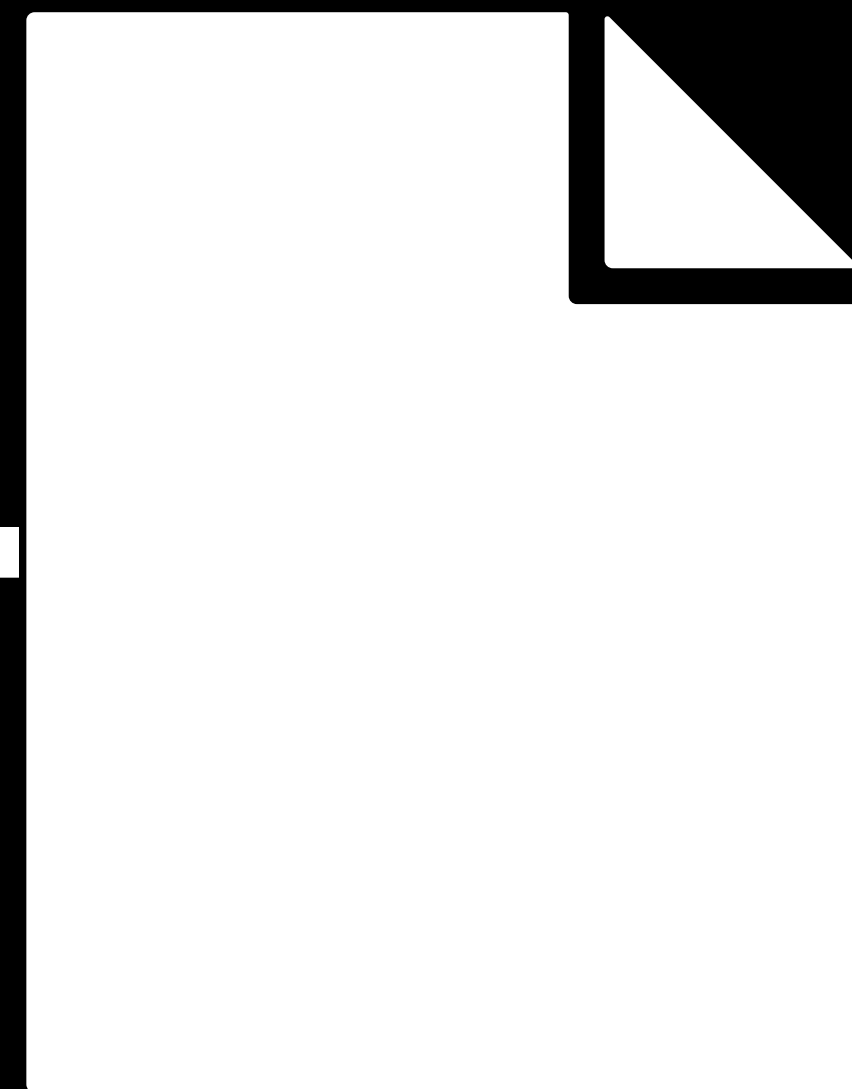


“Create a file”

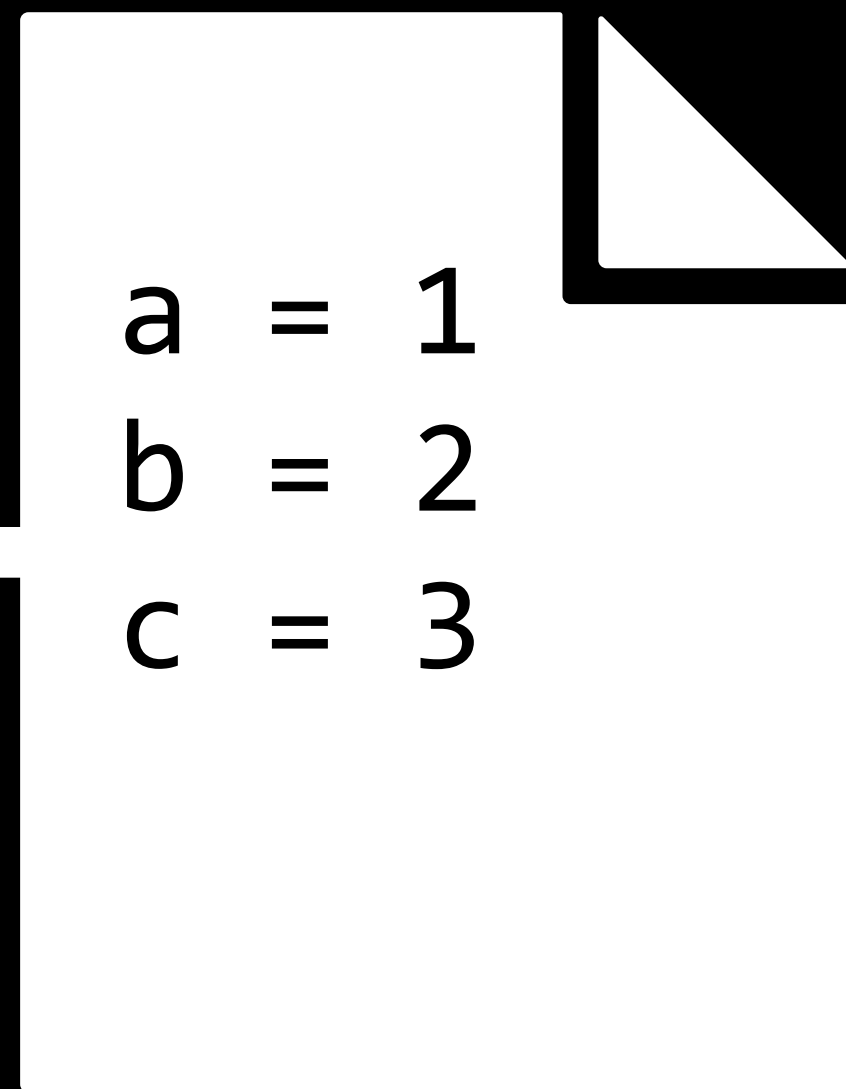


“add variables a,b,c”

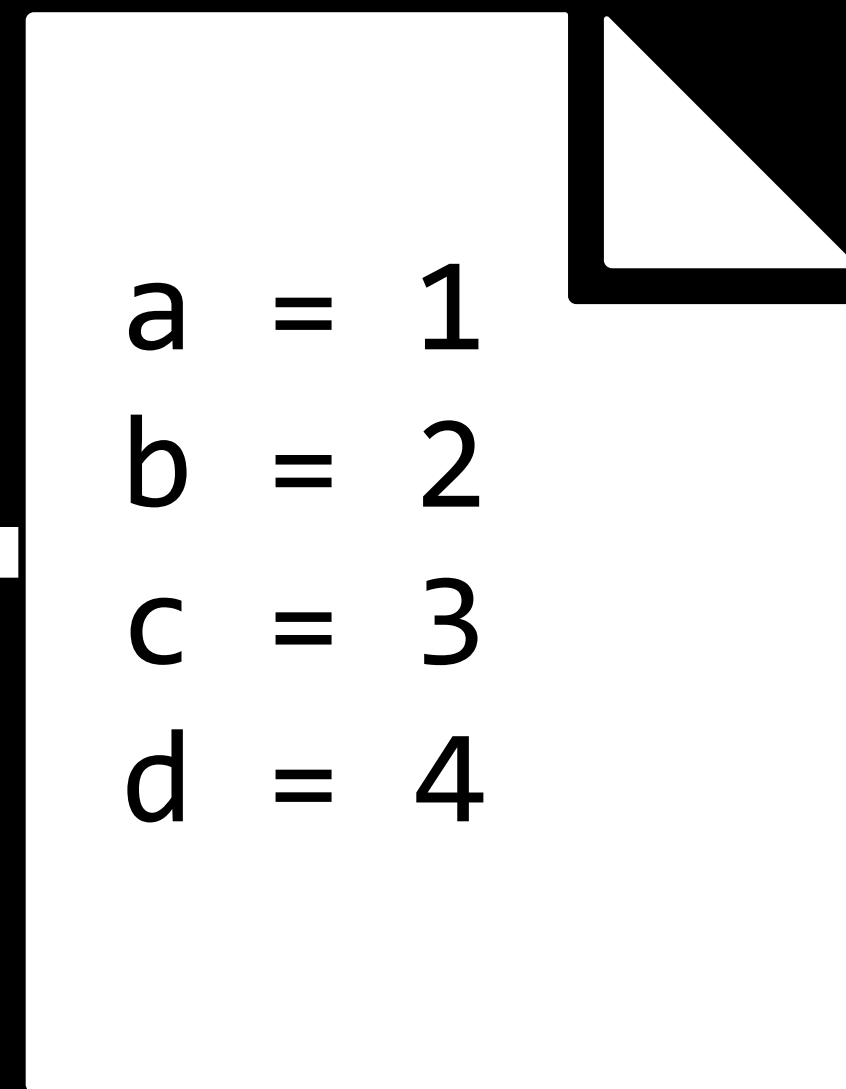
Keeping track of changes to code.



“Create a file”

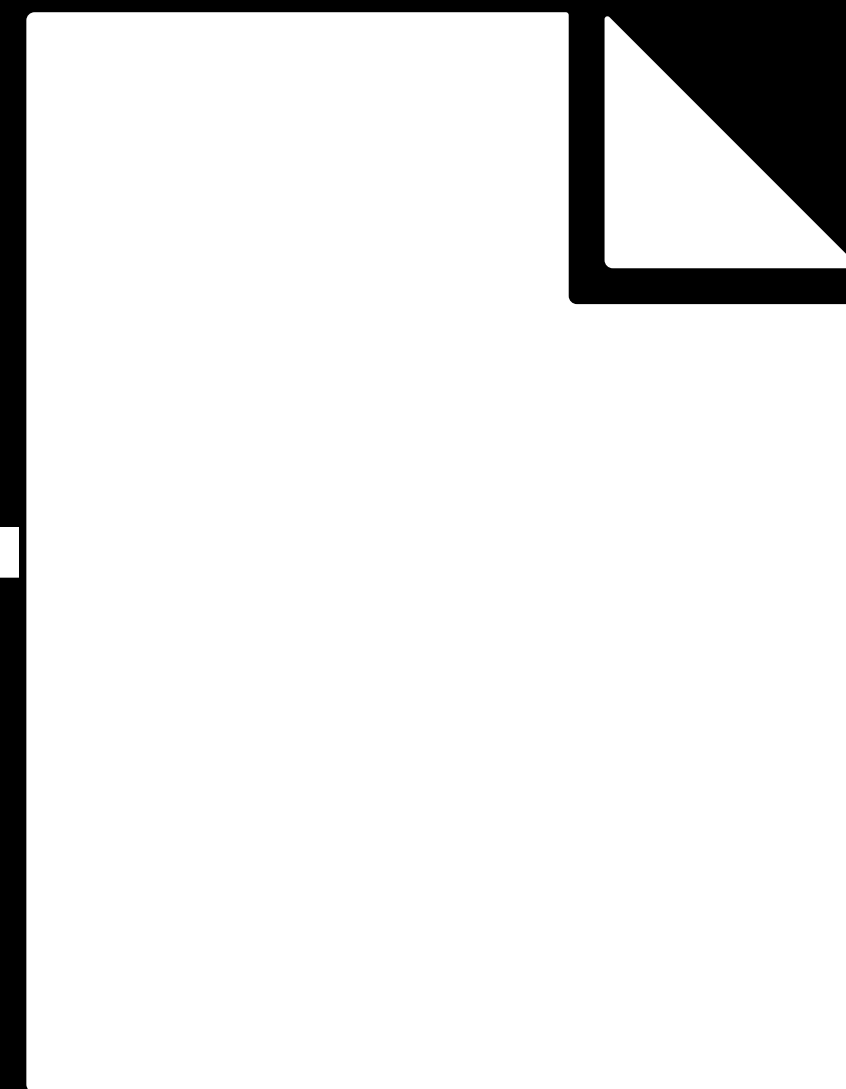


“add variables a,b,c”

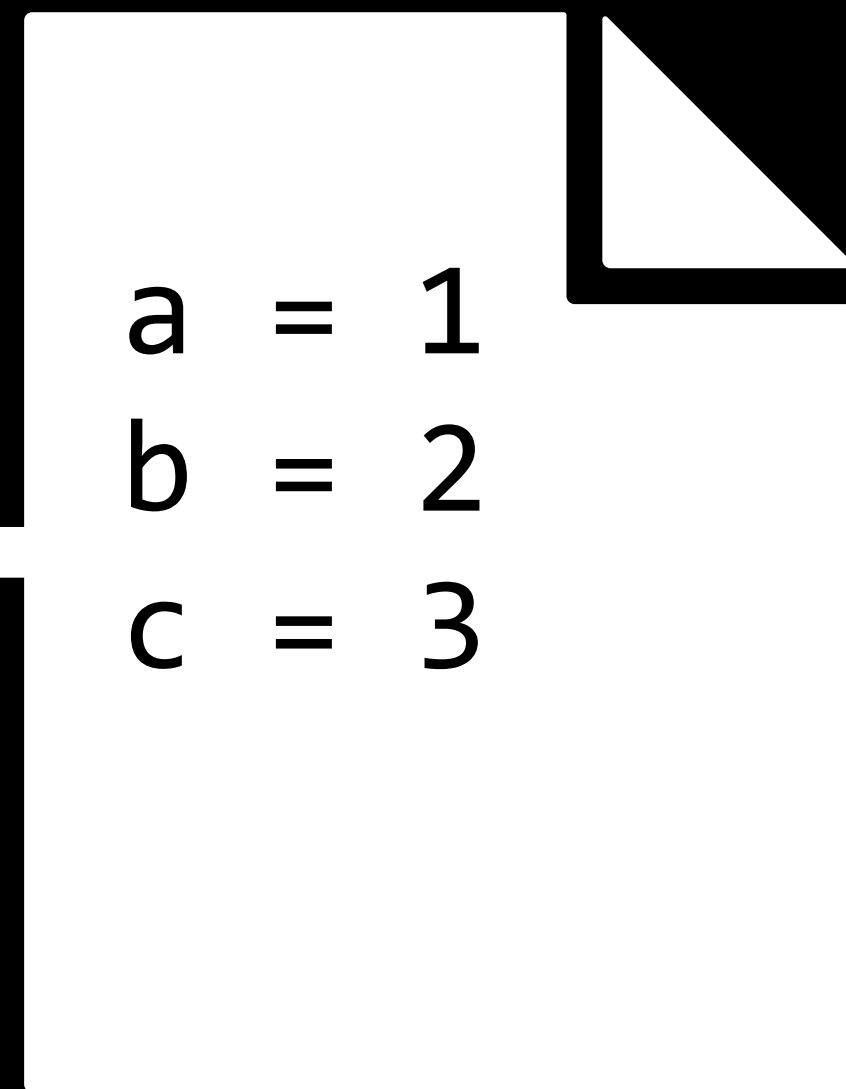


“set d to be = 4”

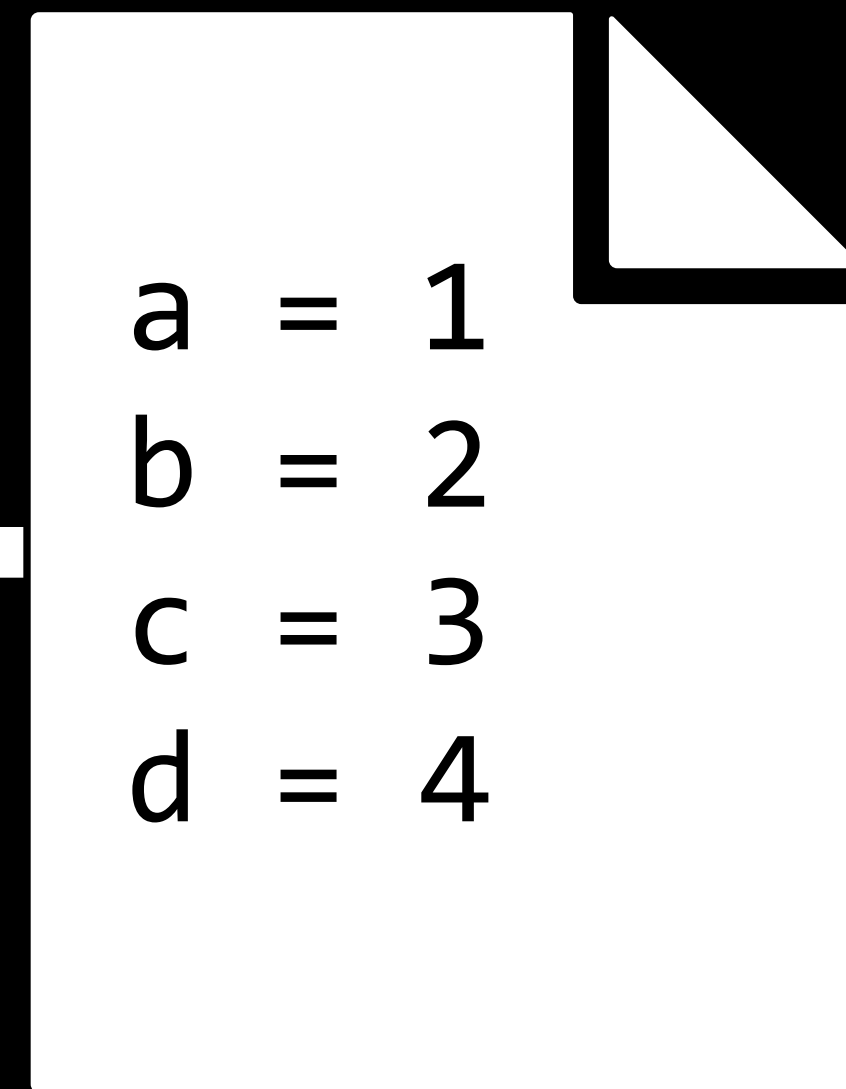
Keeping track of changes to code.



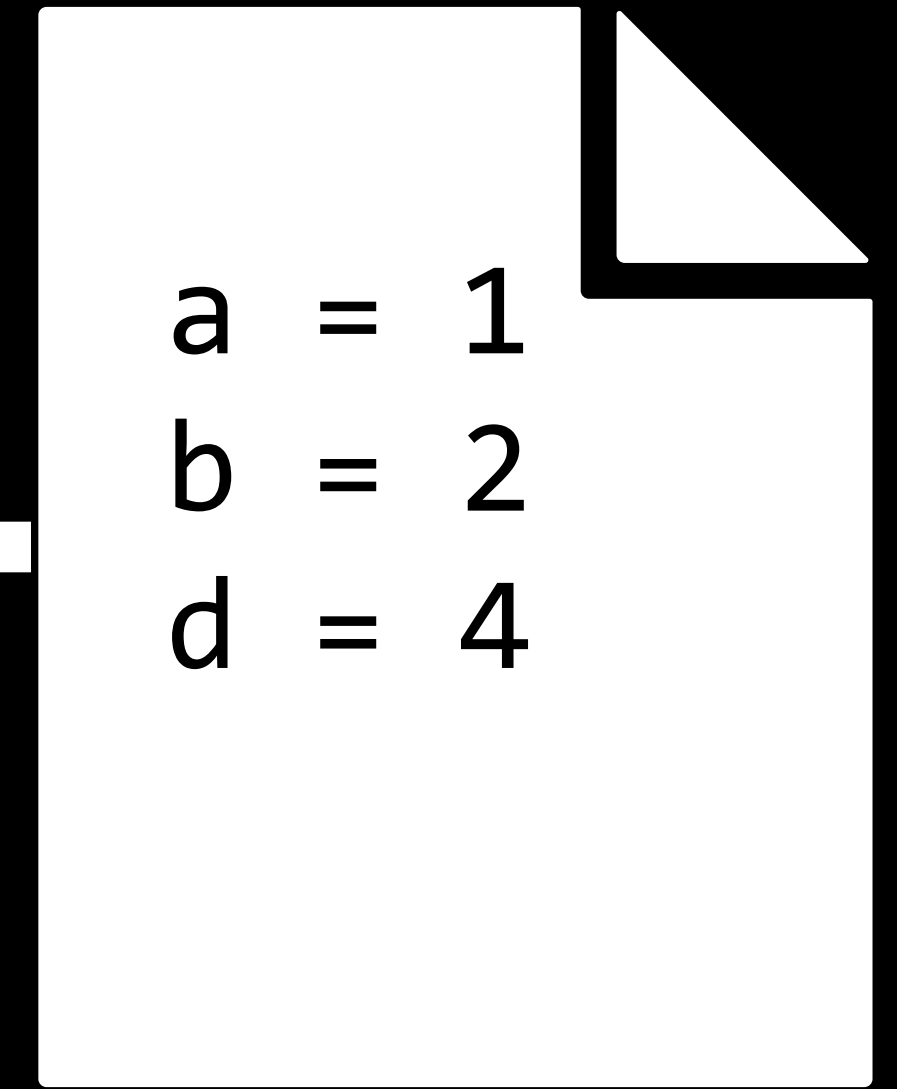
“Create a file”



“add variables a,b,c”

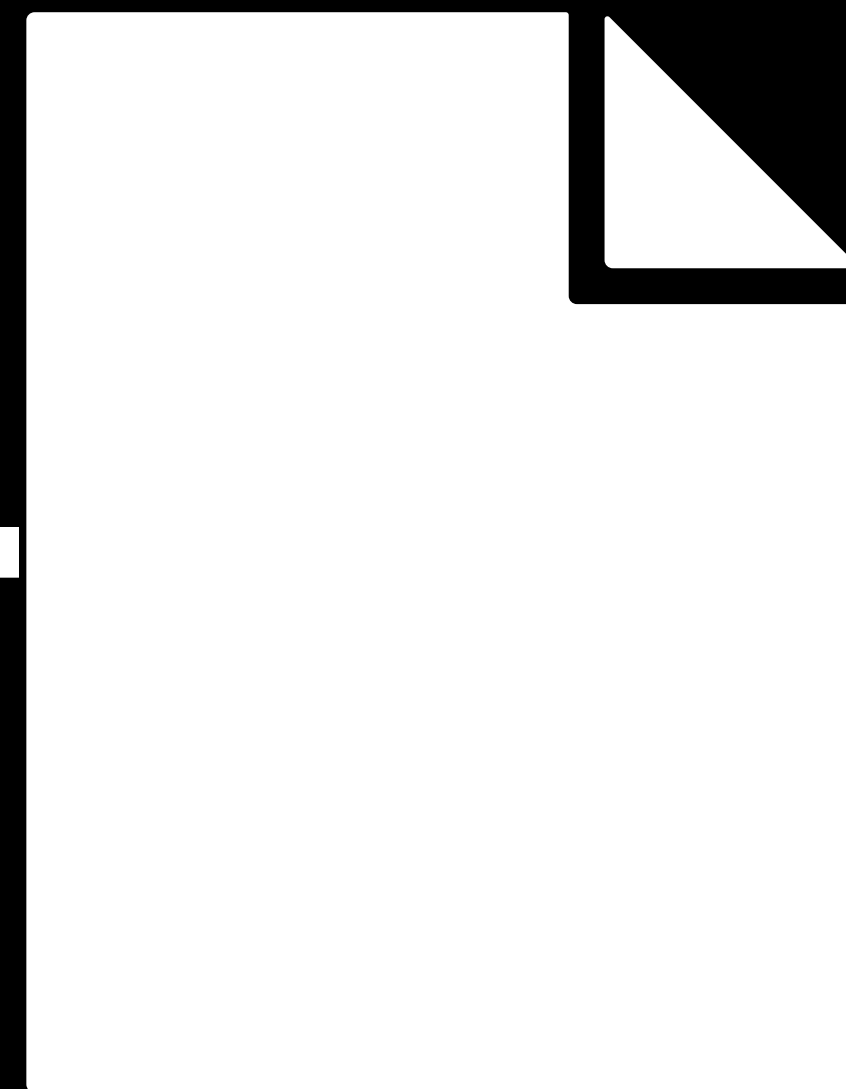


“set d to be = 4”

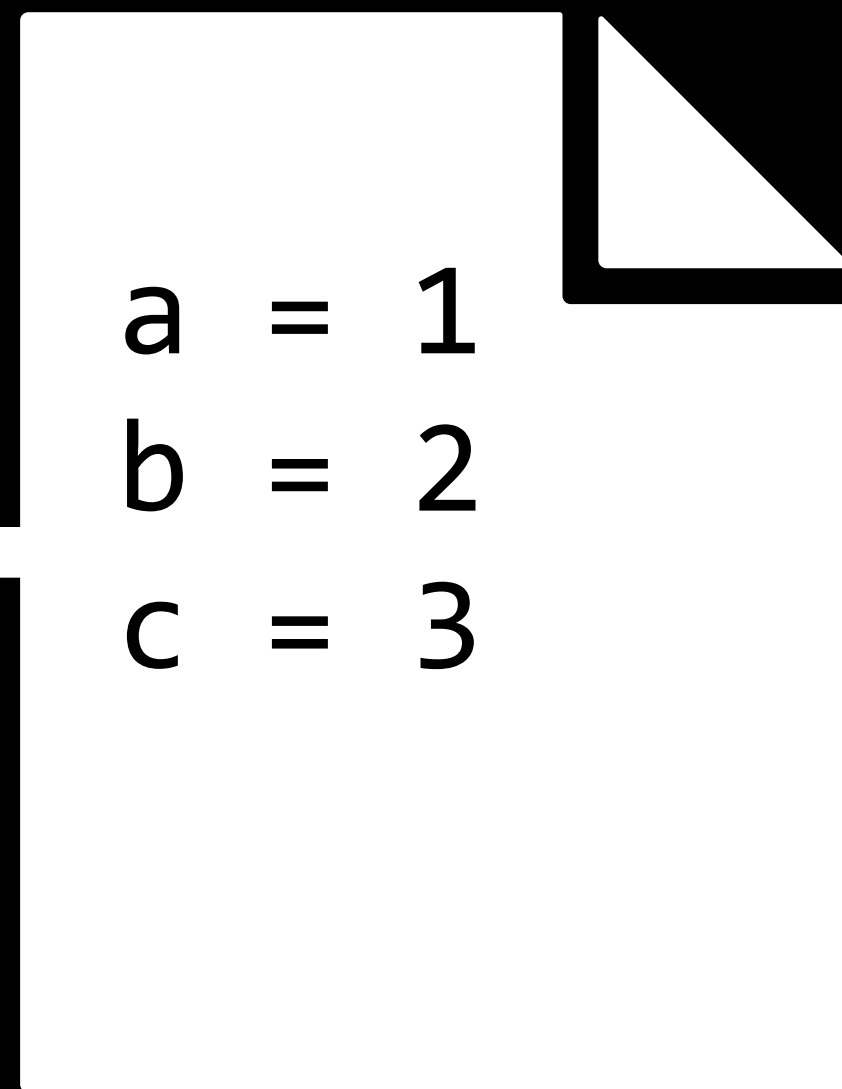


“Remove var c”

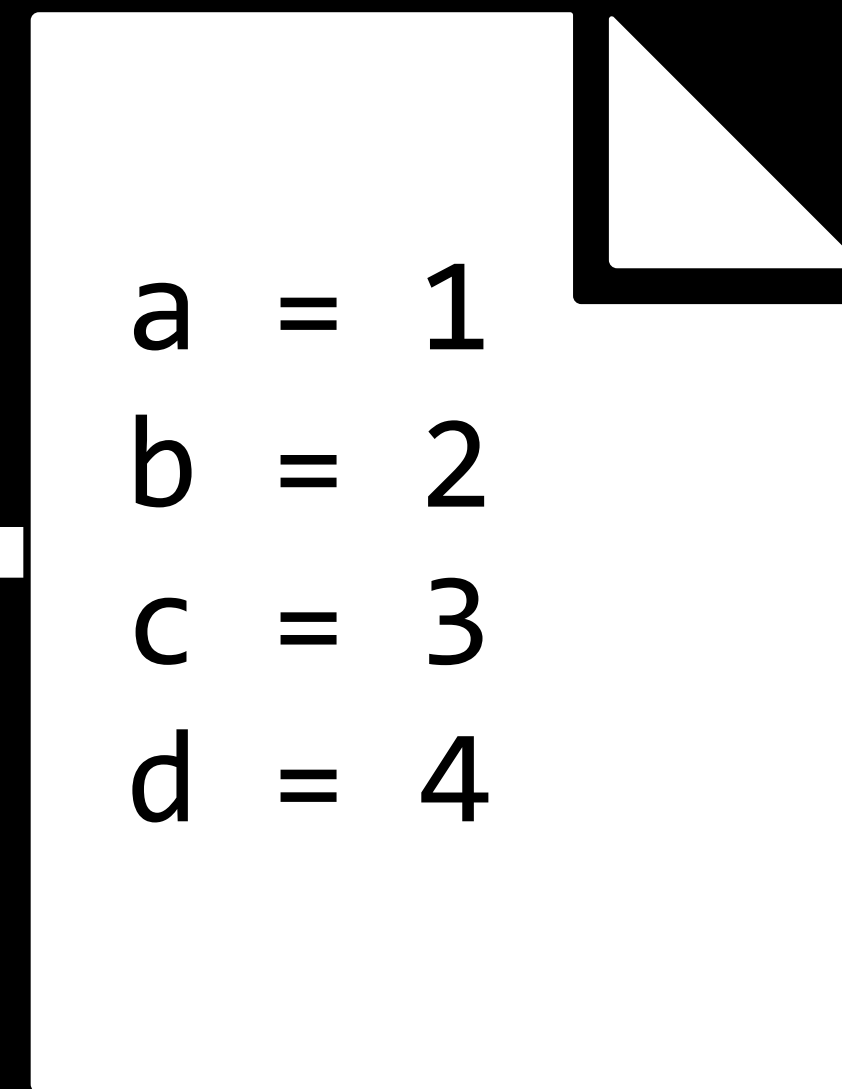
Revert back to old versions of code.



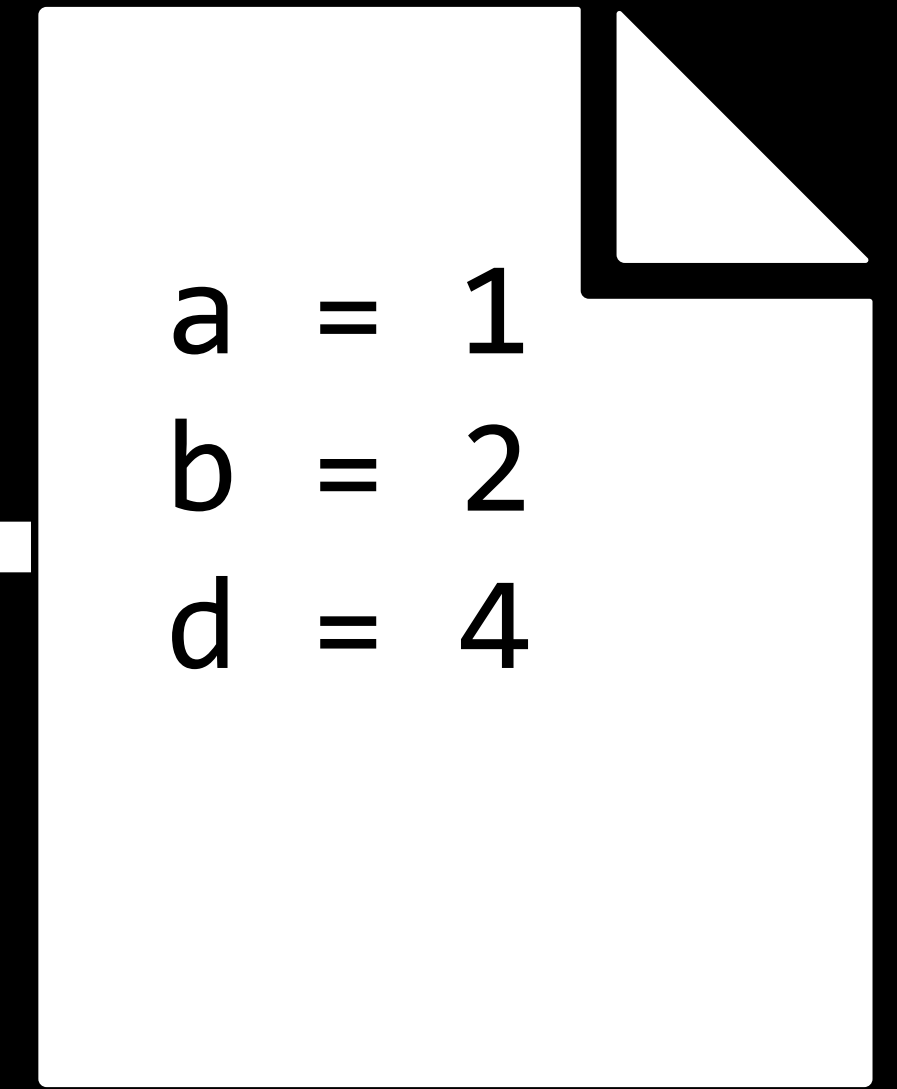
“Create a file”



“add variables a,b,c”

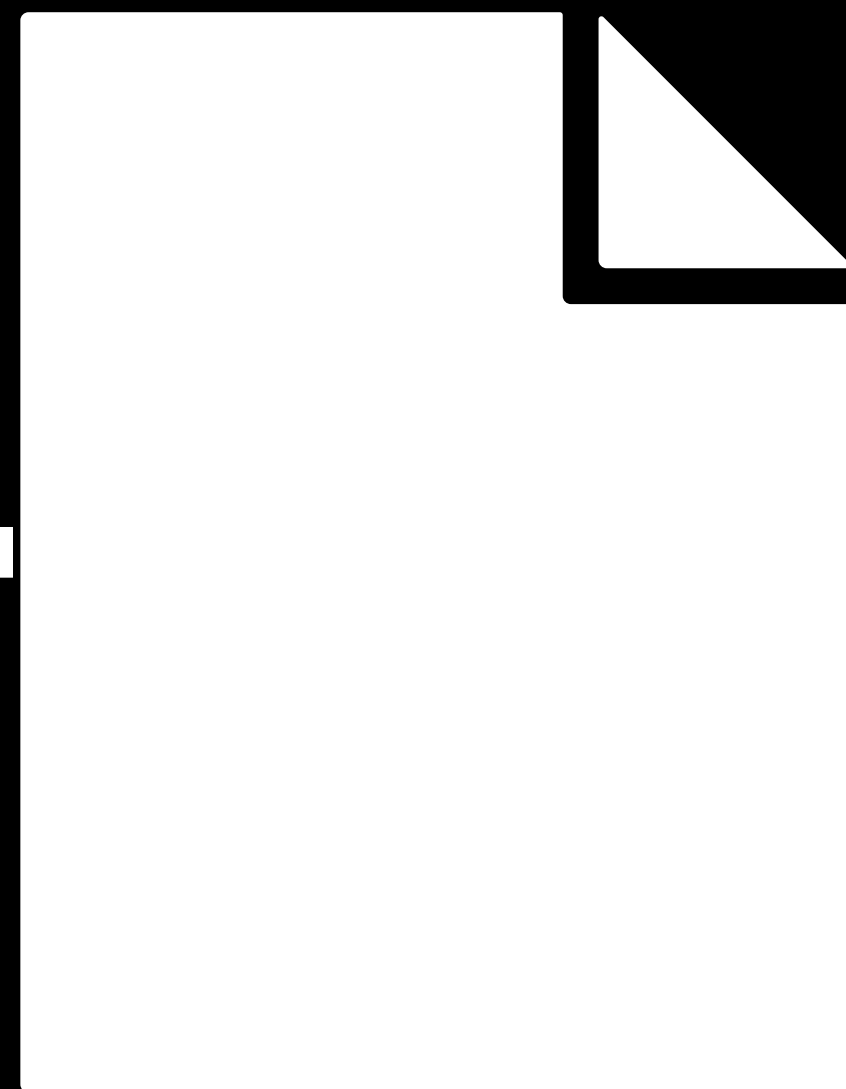


“set d to be = 4”

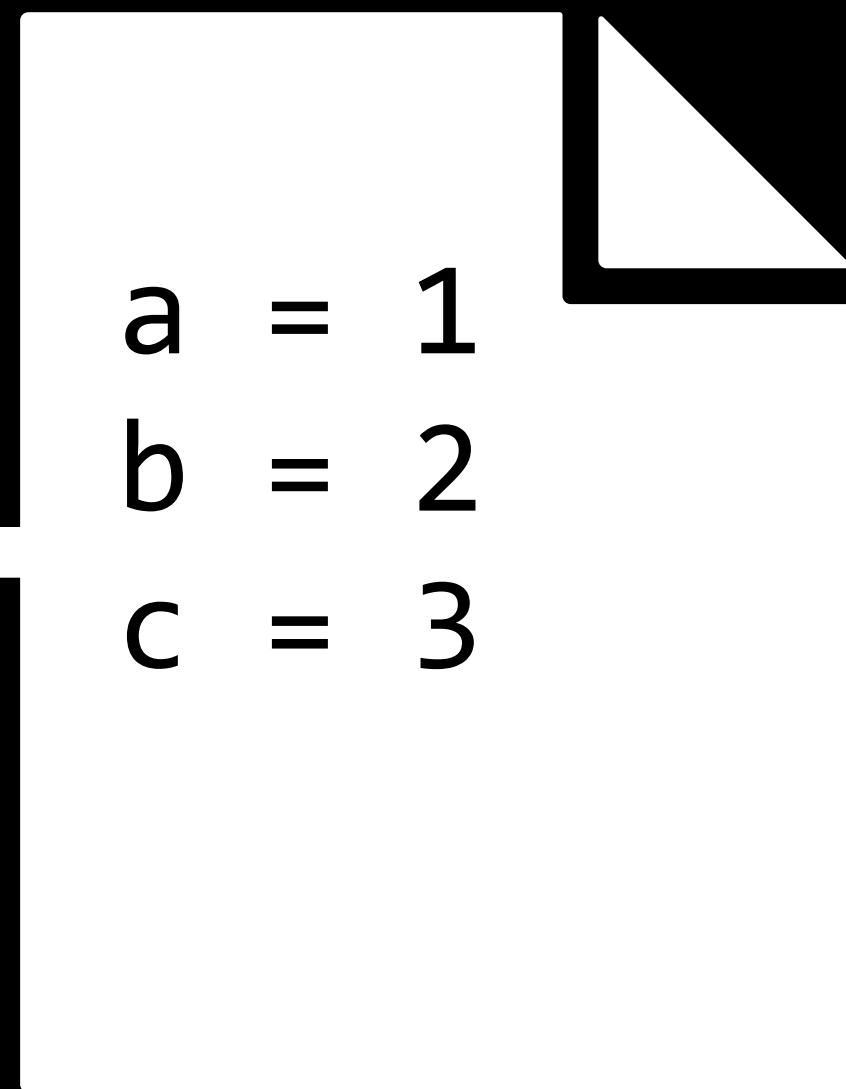


“Remove var c”

Revert back to old versions of code.



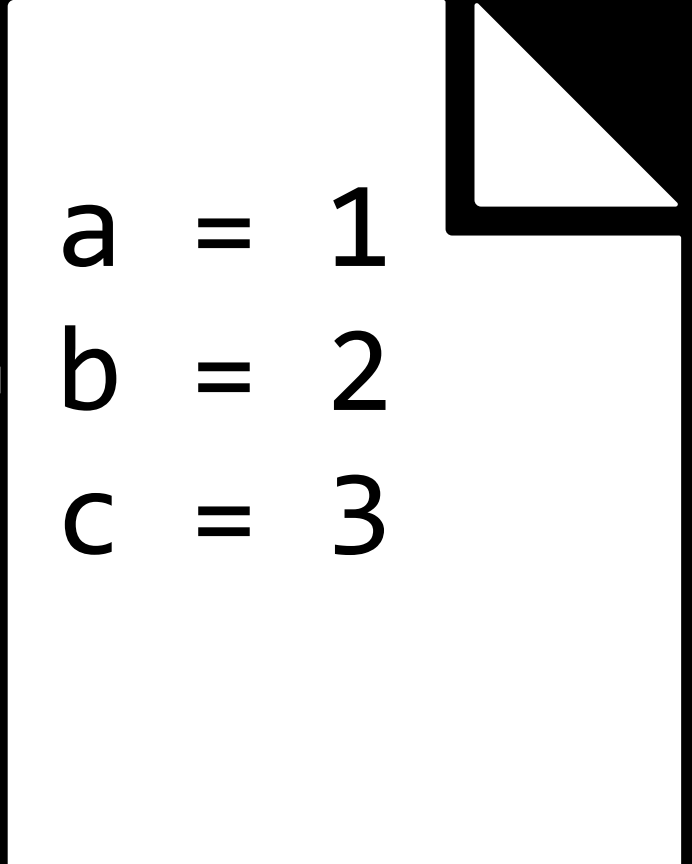
“Create a file”



“add variables a,b,c”

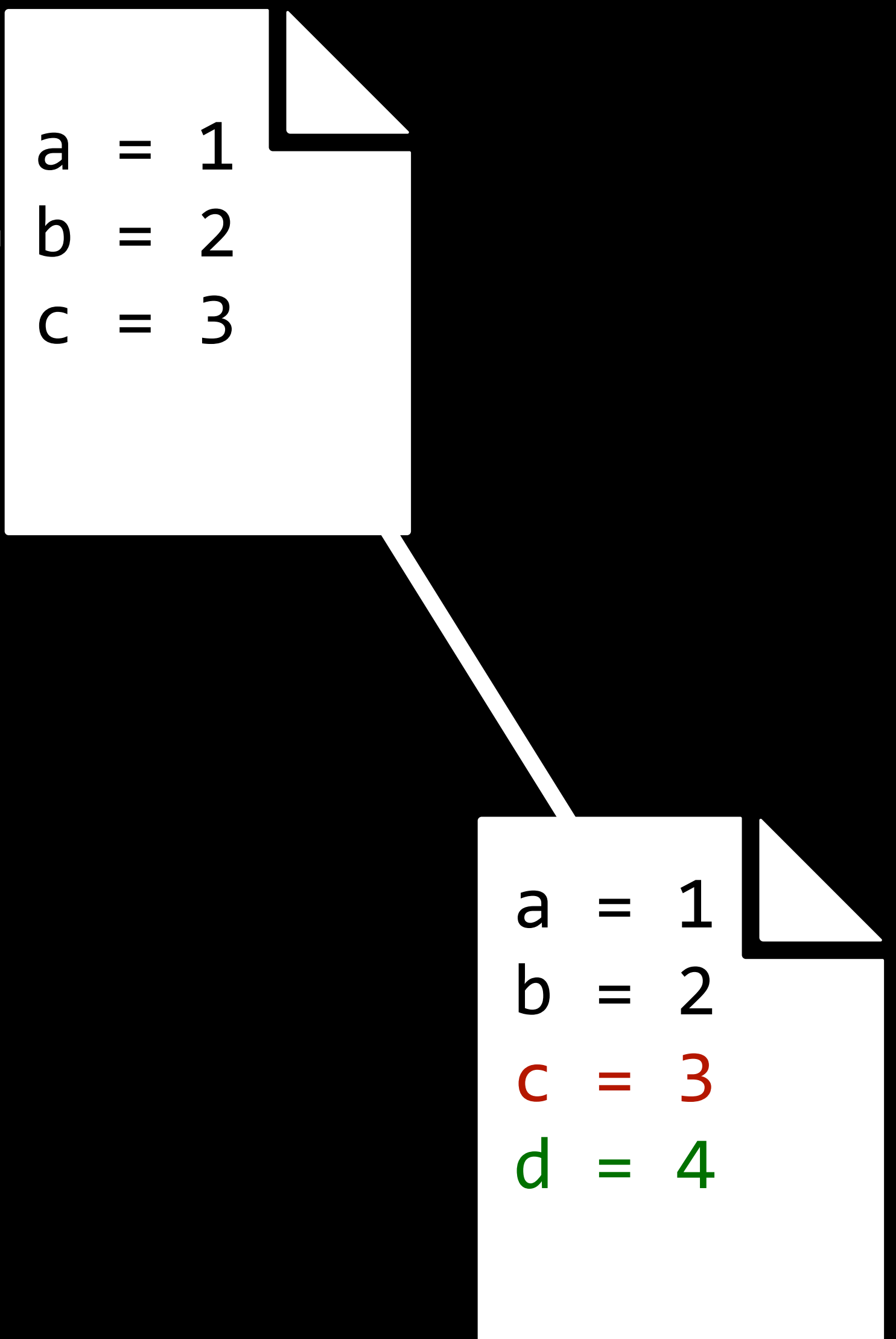
Test changes to code without losing the original.

Test changes to code without losing the original.



```
a = 1  
b = 2  
c = 3
```

Test changes to code without losing the original.



The diagram illustrates a workflow for testing code changes. It features two white rectangular boxes with a folded top-right corner, representing code files. The top box contains the original code: `a = 1`, `b = 2`, and `c = 3`. A horizontal line extends from the left side of this box. A diagonal line connects the bottom-right corner of the top box to the top-left corner of the bottom box. The bottom box contains the modified code: `a = 1`, `b = 2`, `c = 3` (with `c` and `3` in red), and `d = 4` (with `d` and `4` in green).

```
a = 1  
b = 2  
c = 3
```

```
a = 1  
b = 2  
c = 3  
d = 4
```

Test changes to code without losing the original.

```
a = 1  
b = 2  
c = 3
```

The diagram illustrates a workflow for testing code changes. It starts with an original code block containing three lines: 'a = 1', 'b = 2', and 'c = 3'. A line from this block points to a second code block, which represents a test version. This second block contains the same three lines, but with 'c = 3' highlighted in red and a new line 'd = 4' added in green. A line from this second block points to a third code block, which represents the original code after the test. This third block contains 'a = 1', 'b = 2', and 'd = 4', with 'c = 3' removed. The original 'c = 3' line is preserved in the second block, ensuring it is not lost during the testing process.

```
a = 1  
b = 2  
c = 3  
d = 4
```

```
a = 1  
b = 2  
d = 4
```

Test changes to code without losing the original.

```
a = 1  
b = 2  
c = 3
```

```
a = 1  
b = 2  
d = 4
```

```
a = 1  
b = 2  
c = 3  
d = 4
```

```
a = 1  
b = 2  
d = 4
```

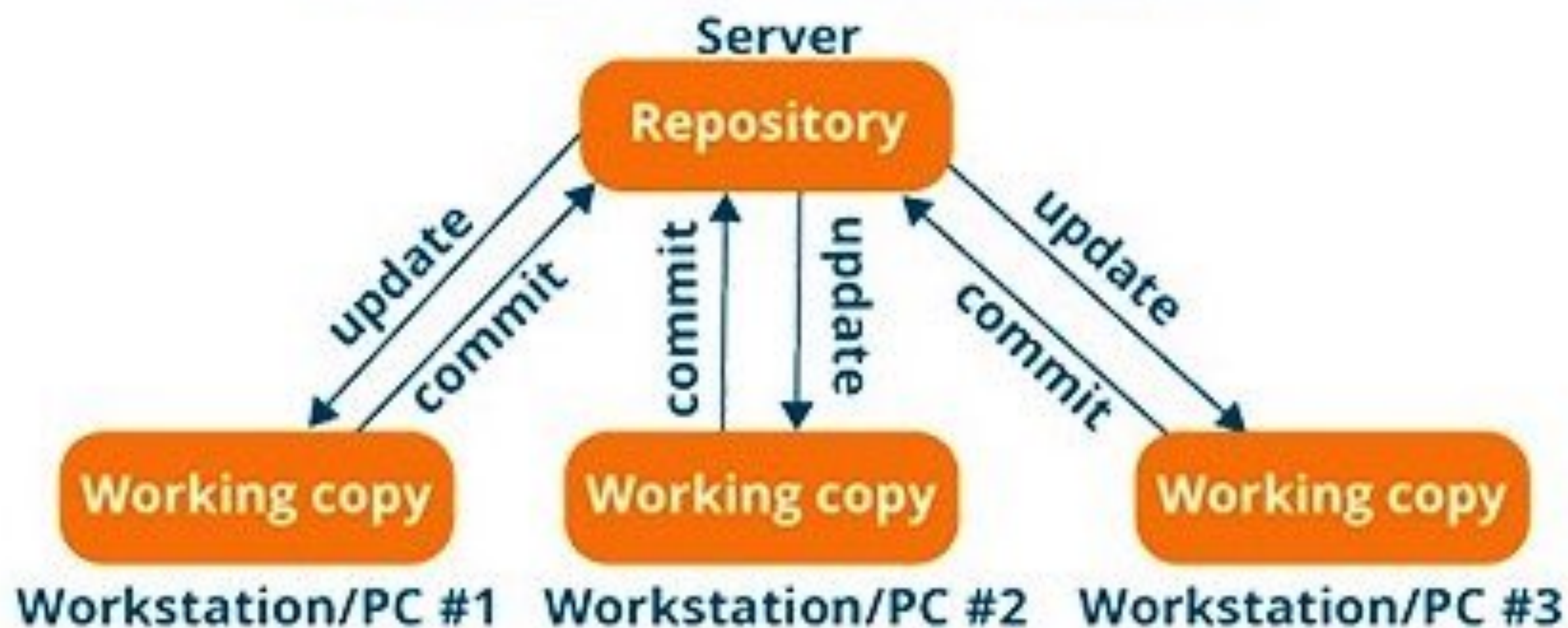
Version Control Systems (VCS)

- VCS can be:
 - Centralized
 - Distributed

Centralized VCS

- Single central copy of your project somewhere (probably on a server)
- Programmers will “commit” their changes to this central copy
- Other programmers will update their projects to get the new changes
- Examples: Subversion (SVN), CVS, and Perforce

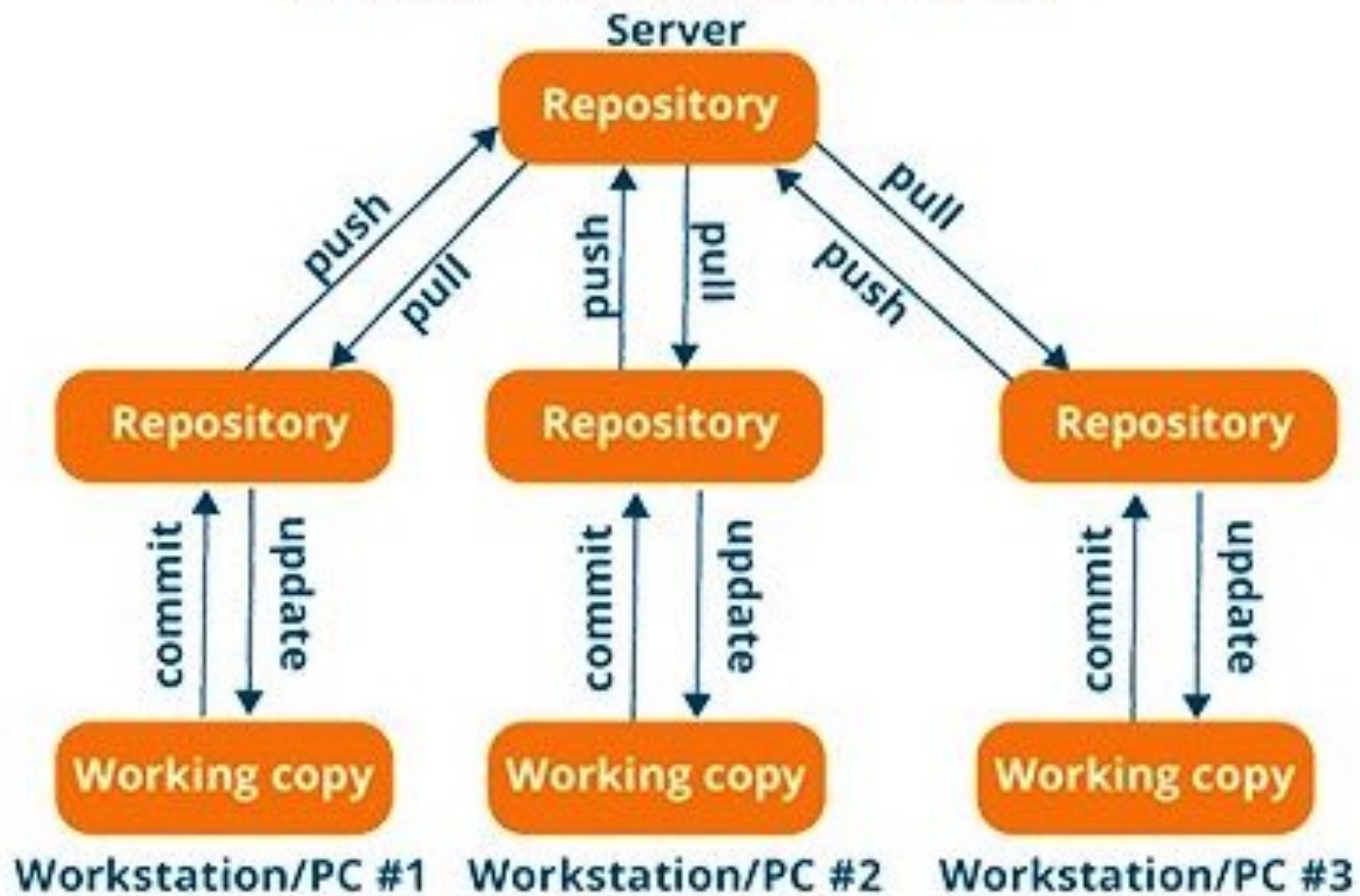
Centralized version control system



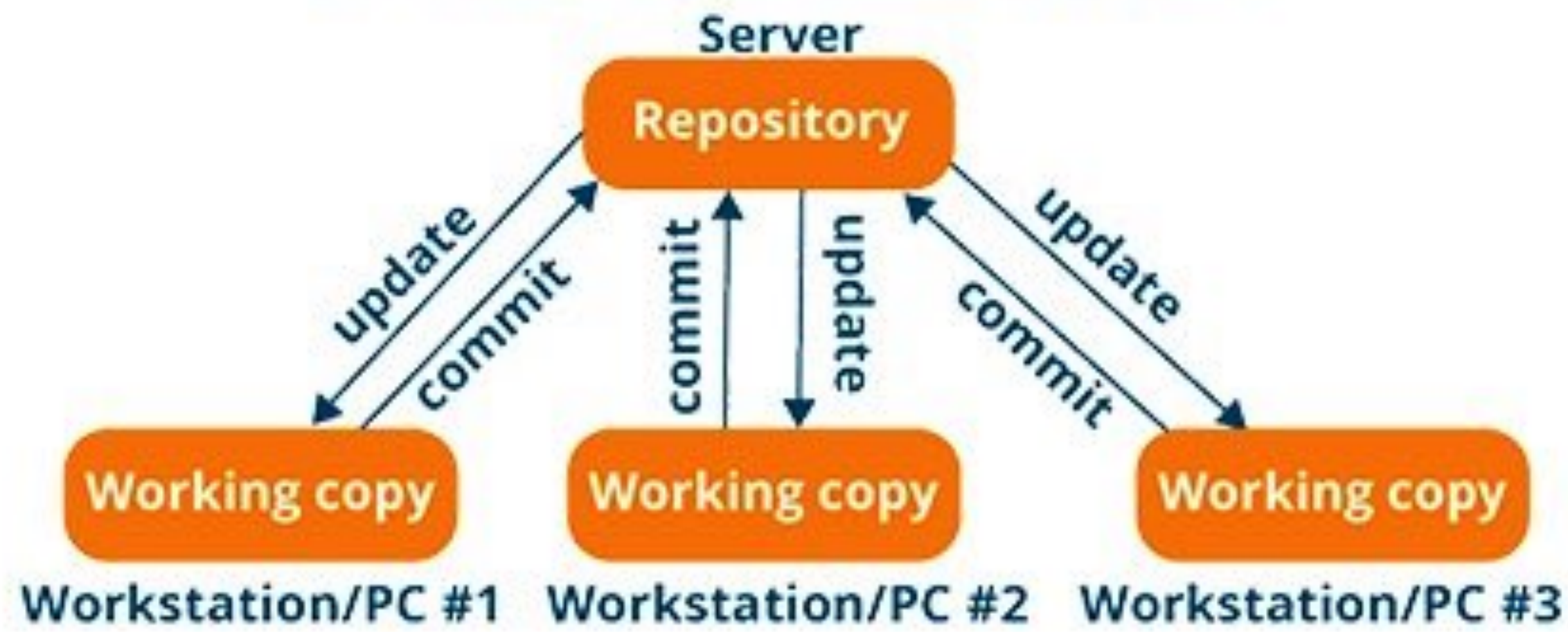
Distributed VCS

- Do not rely on a central server to store all the versions of the project's files.
- Instead, every developer “clones” a copy of the entire repository and has the full history of the project on their local machine.
- The copy (or “clone”) has all of the metadata of the original.
- Examples: **Git**, Mercurial, and Bazaar.

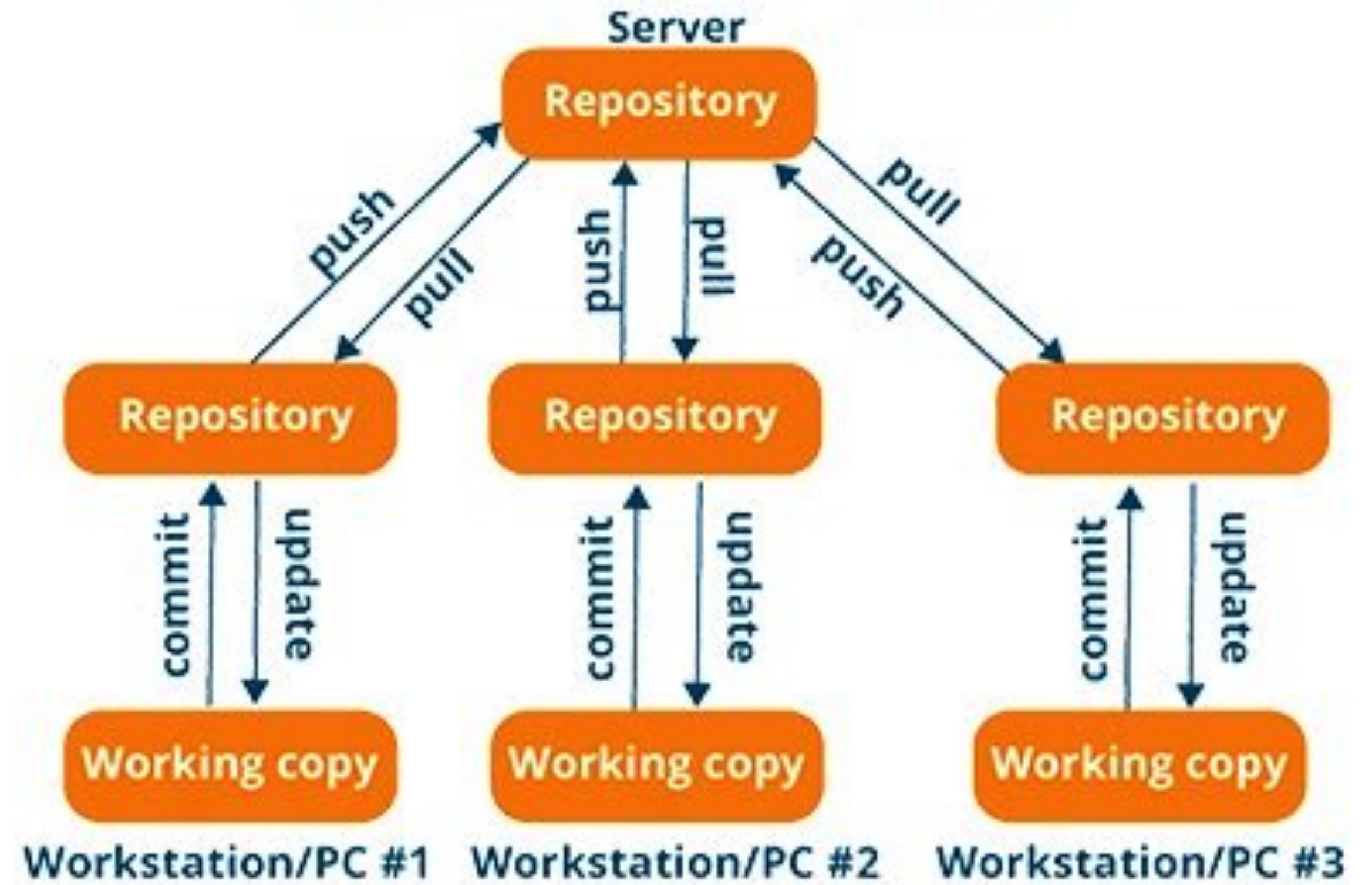
Distributed version control system



Centralized version control system



Distributed version control system



Git

What is Git?

- A Distributed Version Control System
- Created in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel.
- By far, the most used VCS
- Free and open source under the GNU GPLv2 (General Public License version 2.0)
- It is a program written in C, for speed and portability

Installing Git

- <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>
 - ⚠ NOTE: If working on CS50's Codespace, no need to install anything
- To check the current version of Git installed, run `git --version`
 - ⚠ NOTE: If working on CS50's Codespace, first run the command `cd /workspaces`

Terminologies

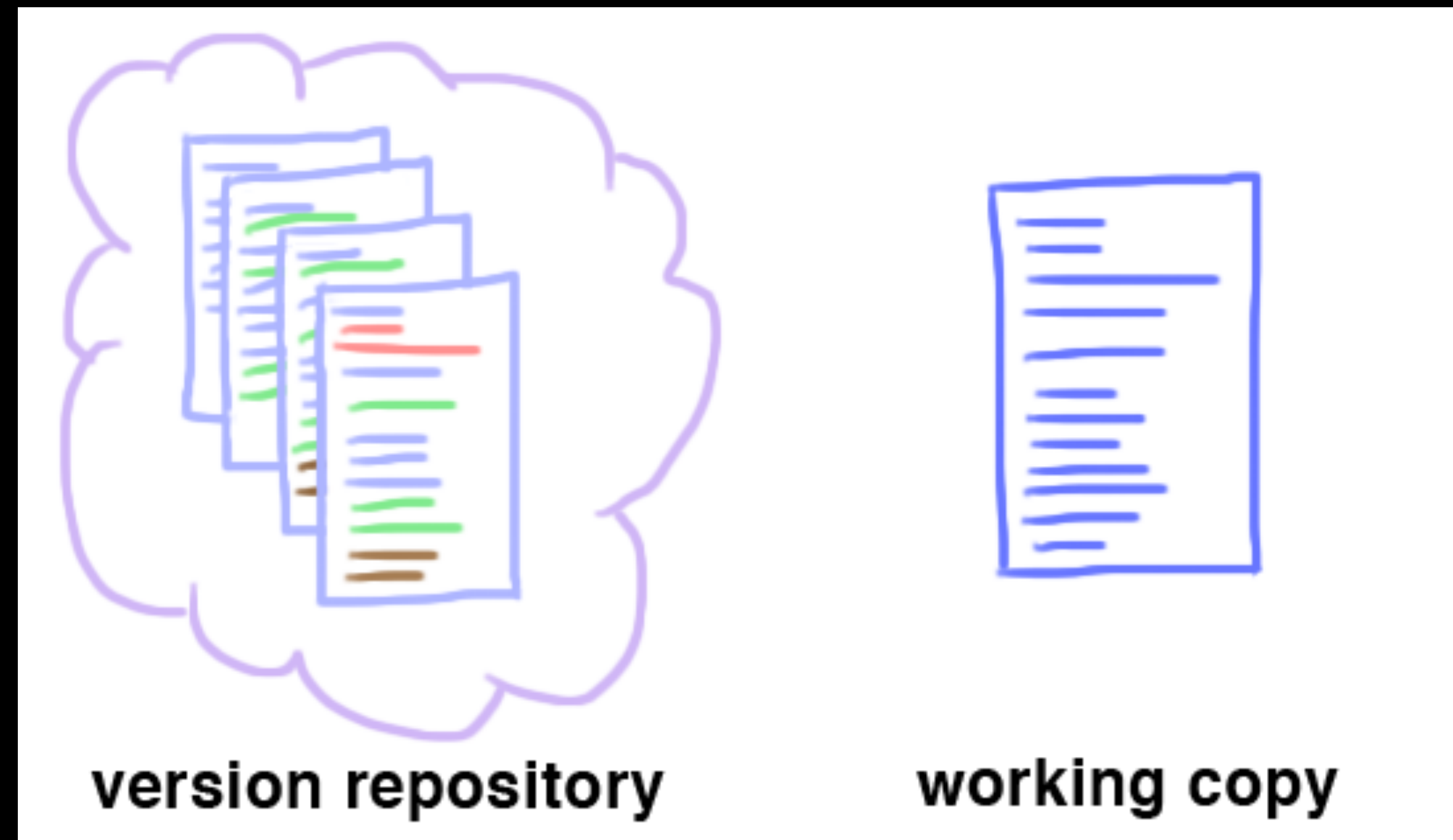
- **Repository**
- **Working copy**
- **Staging area**

Repository

- A repository is a data structure that stores metadata for a set of directories/folders and files
- You can think of a repository as a bunch of version-controlled folders and files.
- With Git, a repository is specifically the `.git` file that is hidden by default — you don't need look at it neither touch it
- Repositories can be:
 - Local - exists in your local machine
 - Remote - existing at some server or remote computer

Working copy

- Aka Working Tree, Workspace
- The folders and files in the directory that you can see and edit at the moment



Staging area

- Aka the Git Index
- The “intermediate” area between the Working Copy and the Git Repository
- Used to set up and combine all changes together before you commit them to your local repository

Steps to keep track of changes

1. Turn a directory into a git repository (if it isn't already).
2. Make changes and add them to the staging area.
3. Commit or “take a snapshot” of the changes and save to the git repository.
4. Undo changes or revert to a previous version, if needed.

git init

- Initializes a Git repository in the current directory
- Creates a `.git` hidden file
- You can view the file by running `ls -a`

git add <filename>


- Adds changes made in the working copy to the <filename> file to the staging area
- Alternatively, you can add the changes made to all the files in the working copy by running the command: `git add .`

git commit -m “message”

- Captures a snapshot of the project's currently staged changes
- The changes are now safely saved in the git repository
- Each commit has
 - A unique ID which is hashed value,
 - The author
 - Date and time
 - And more...
- Alternatively, `git commit -am “message”` → add all + commit

Make *small* **commits**

git reset <commit ID>

- Undoes all commits after <commit ID> commit, but preserves the changes locally
-  More dangerously, you can change history by:
 - git reset --hard <commit ID>
 - Discards all history and changes back to the specified commit

.gitignore

- A .gitignore file specifies intentionally untracked files that Git should ignore.
- Files already tracked by Git are not affected
- You may specify a specific file name or a pattern of a name (i.e hello.txt or /hello.*)

Other useful commands

- `git diff` : Shows the changes between the working copy and the previous commit
- `git status` : Shows displays the state of the working directory and the staging area
- `git log` : Lists version history for the current branch
- `git log --follow <file>` : Lists version history for a specific file

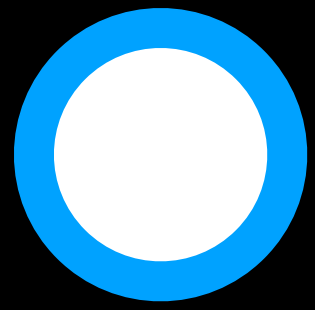
Demo 1

Keeping track of changes

Branches and Merges

Branches

- So far, we have only been working on the main branch, but we can create more!
- A branch represents an independent line of development
- With it's own working directory, staging area, and project history line
- We can then later “merge” two or more branches together to combine the changes
- We can delete unwanted branches

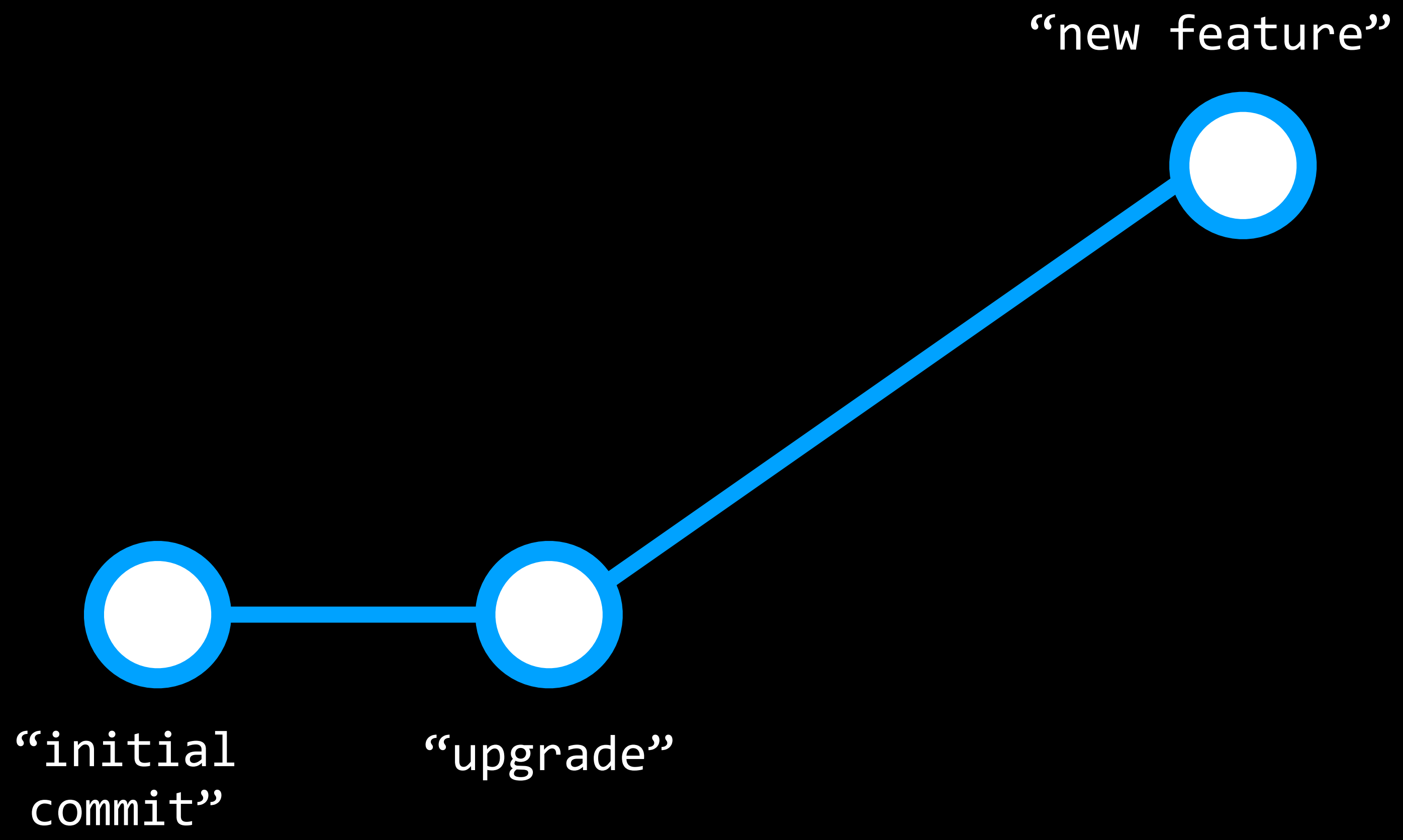


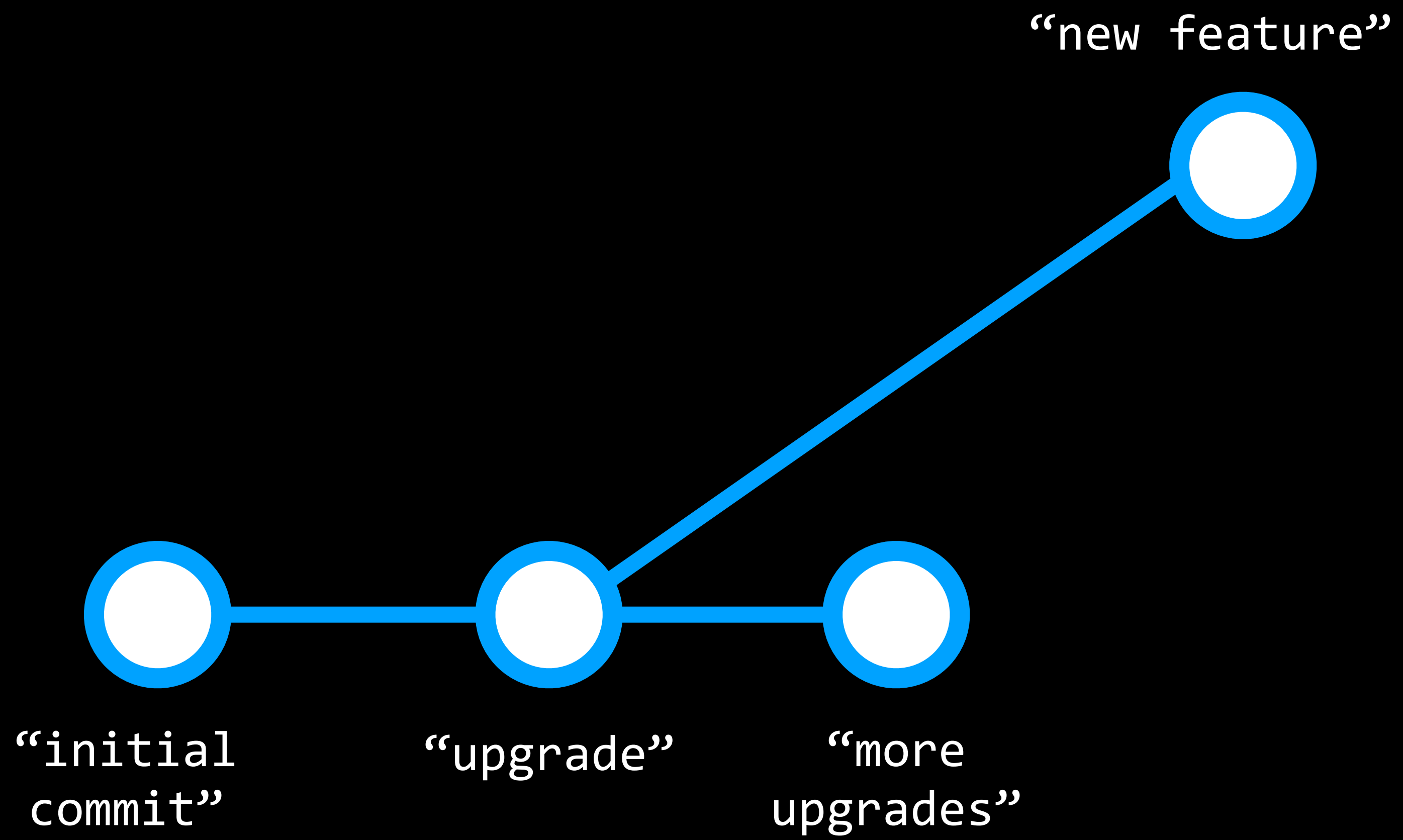
“initial
commit”

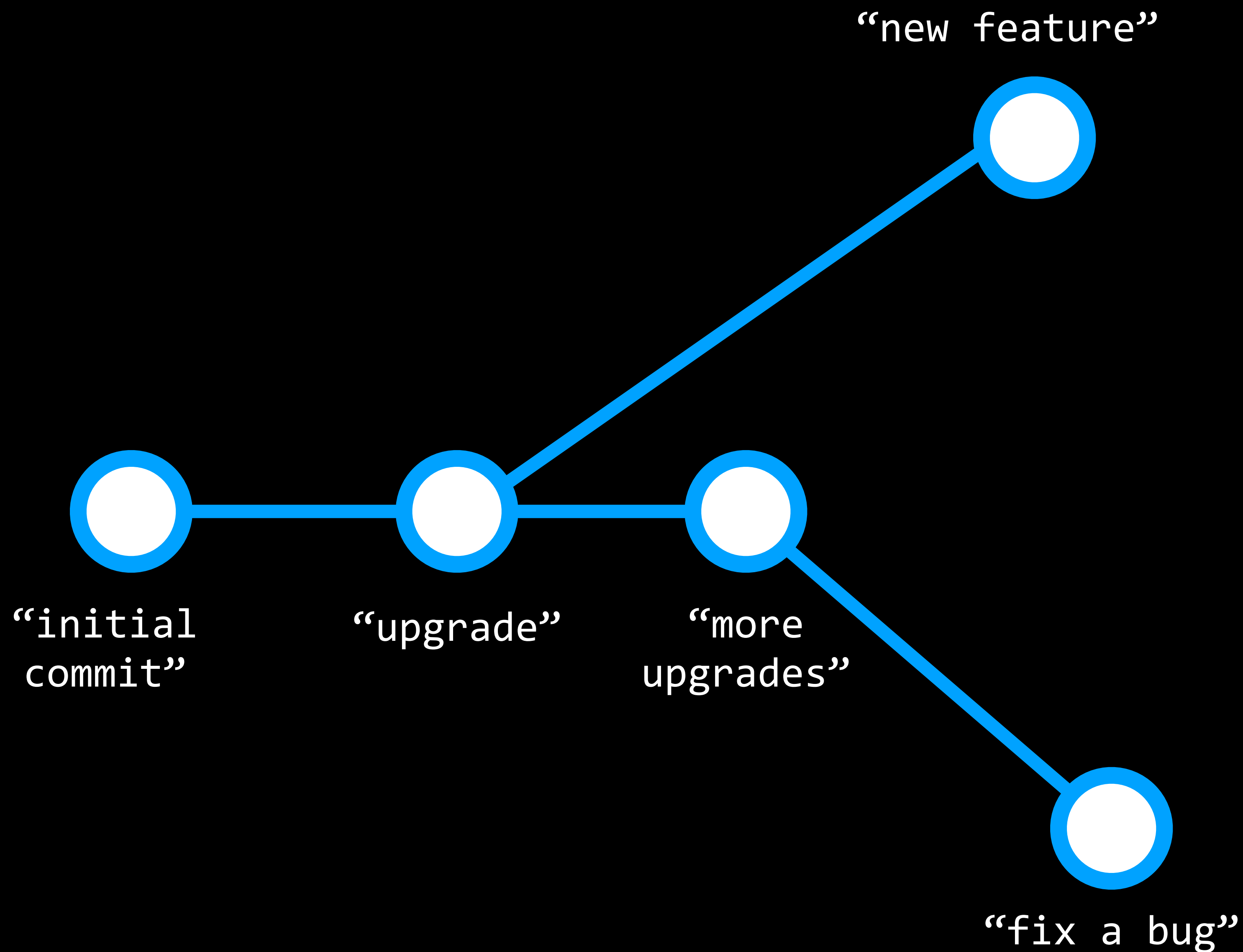


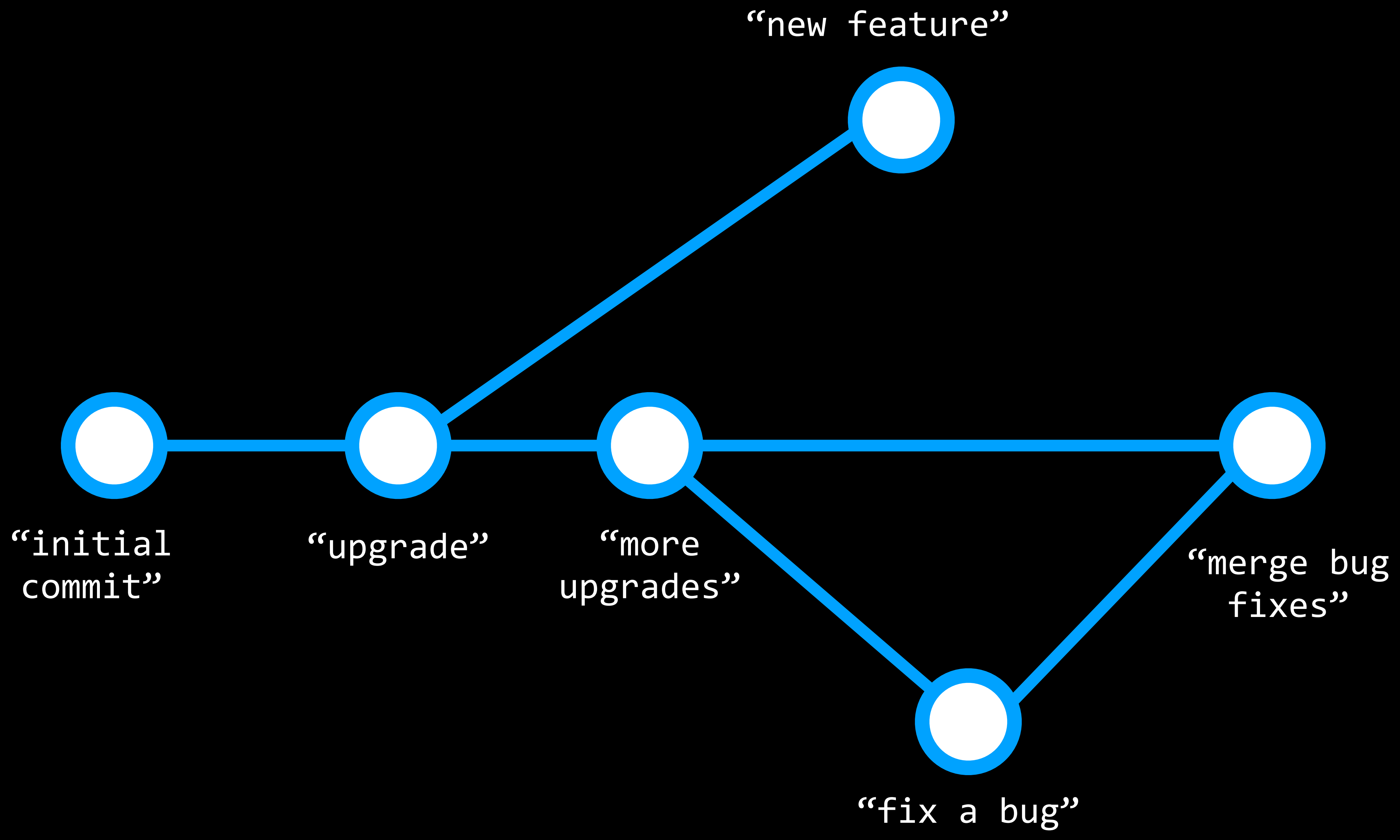
“initial
commit”

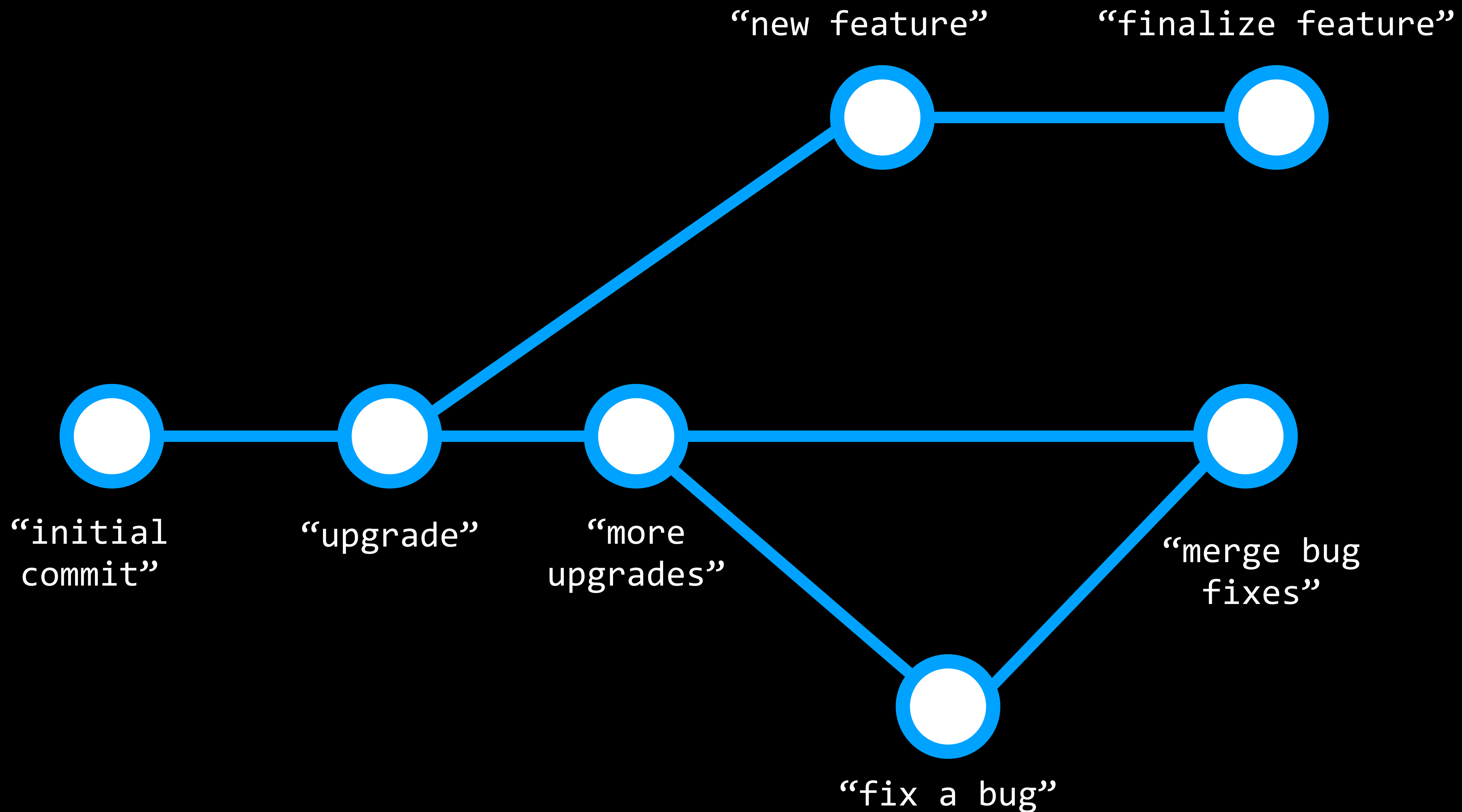
“upgrade”

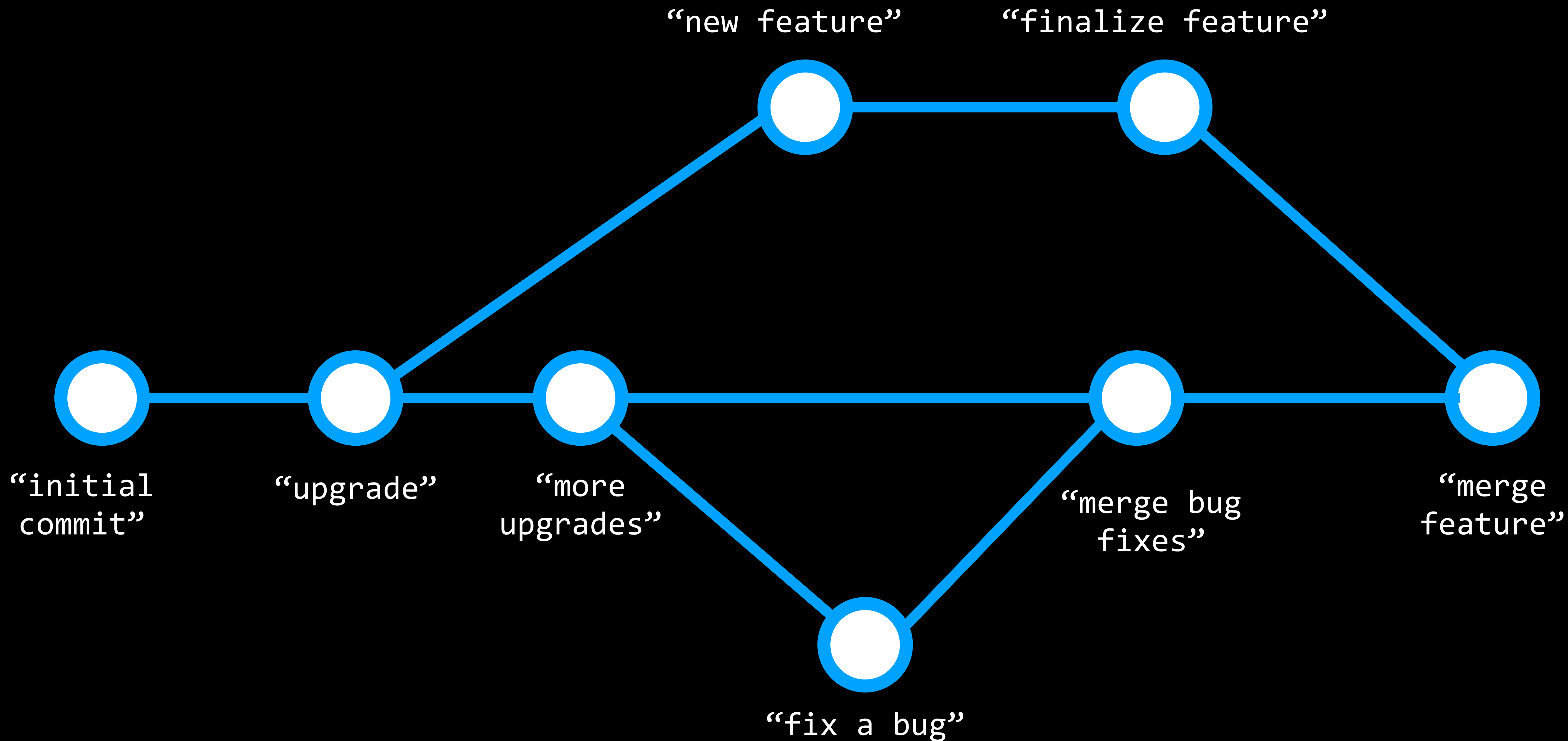












git branch

- Lists all branches of the local repository
- Highlights the branch that we are currently sitting on, with an asterisk (*) and some color

`git branch <branch-name>`

- Creates a new branch called `<branch-name>`
- The new branch is now only on the `.git` repository

git checkout <branch-name>

- Switches to the <branch-name> branch
- Now the working directory is sitting on the new branch
- Alternatively, you may create + checkout to a new branch all at once by running: `git checkout -b <branch-name>`

git merge <source-branch>

- Automatically merges into the current branch and updates it to reflect the merge
- The <source-branch> will be completely unaffected
- What can go wrong in this case?

**What if the two branches changed the
same part of the same file?**

What if the two branches changed the
same part of the same file?

Merge Conflict!

Resolving conflicts

- Accept current changes
- Accept incoming changes
- Accept both changes
- Compare changes and make new edits
- Abort merging:
 - `git merge --abort`

Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes

8 <<<<<<< HEAD (Current Change)

9 z = 4


10 =====

11 z = 6

12 >>>>>>> branch-b (Incoming Change)

13

git branch --delete <branch-name>

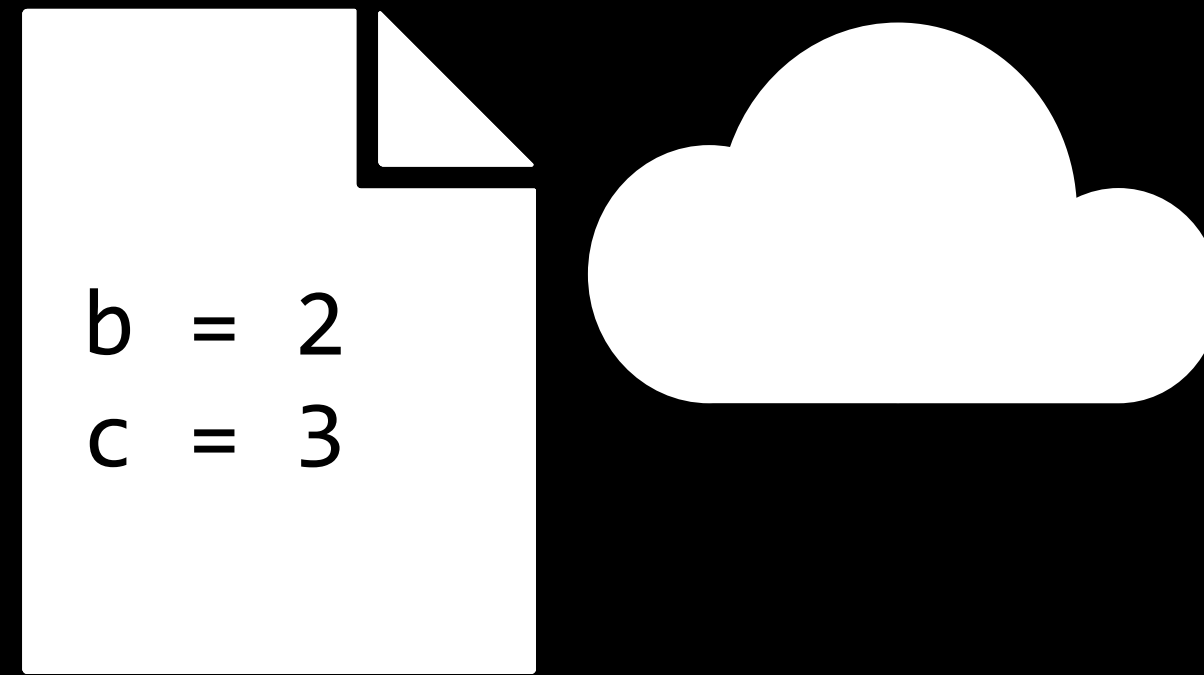
- Deletes <branch-name> branch
- Fails if the branch has unmerged or uncommitted changes, for safety
-  You may forcefully delete by running:
 - git branch --delete --force <branch-name>

Demo 2

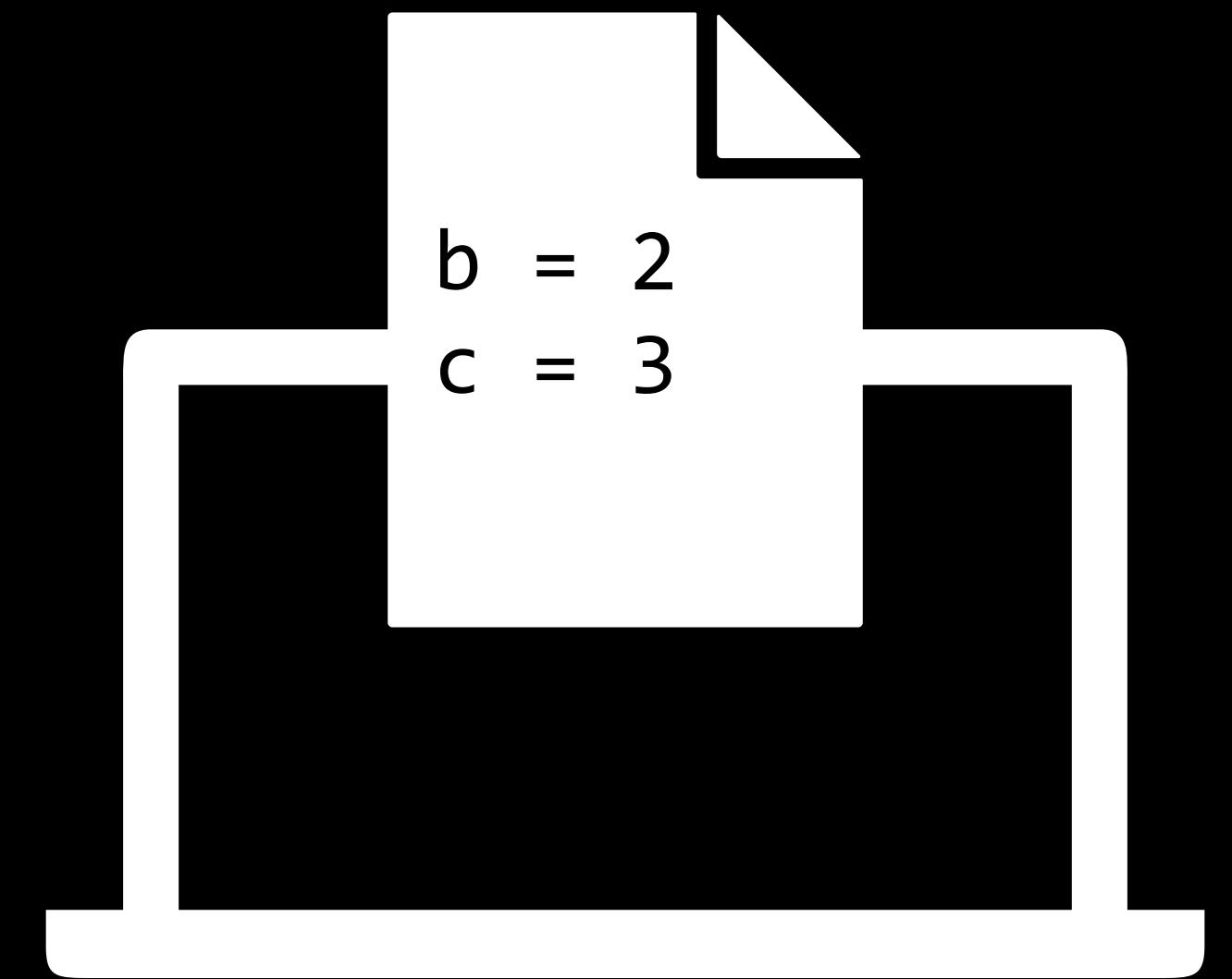
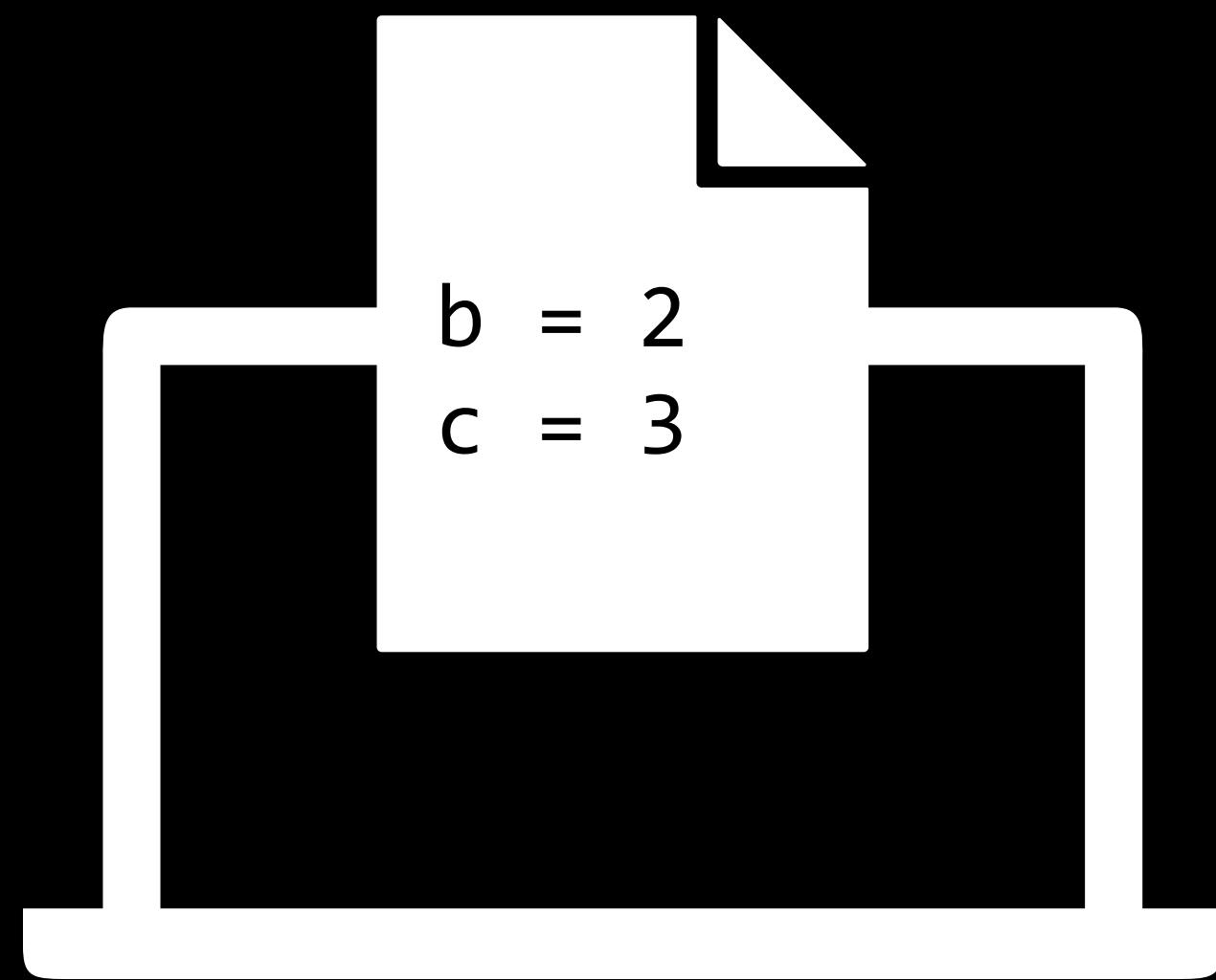
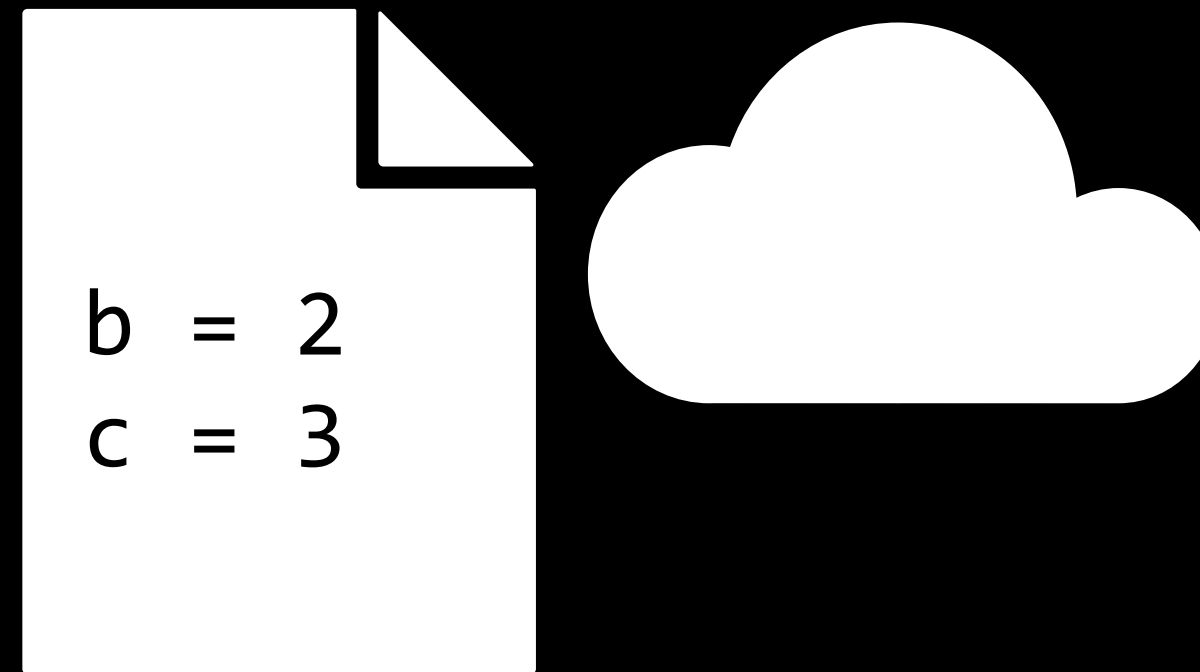
Branches and Merges

Synchronize changes between different people

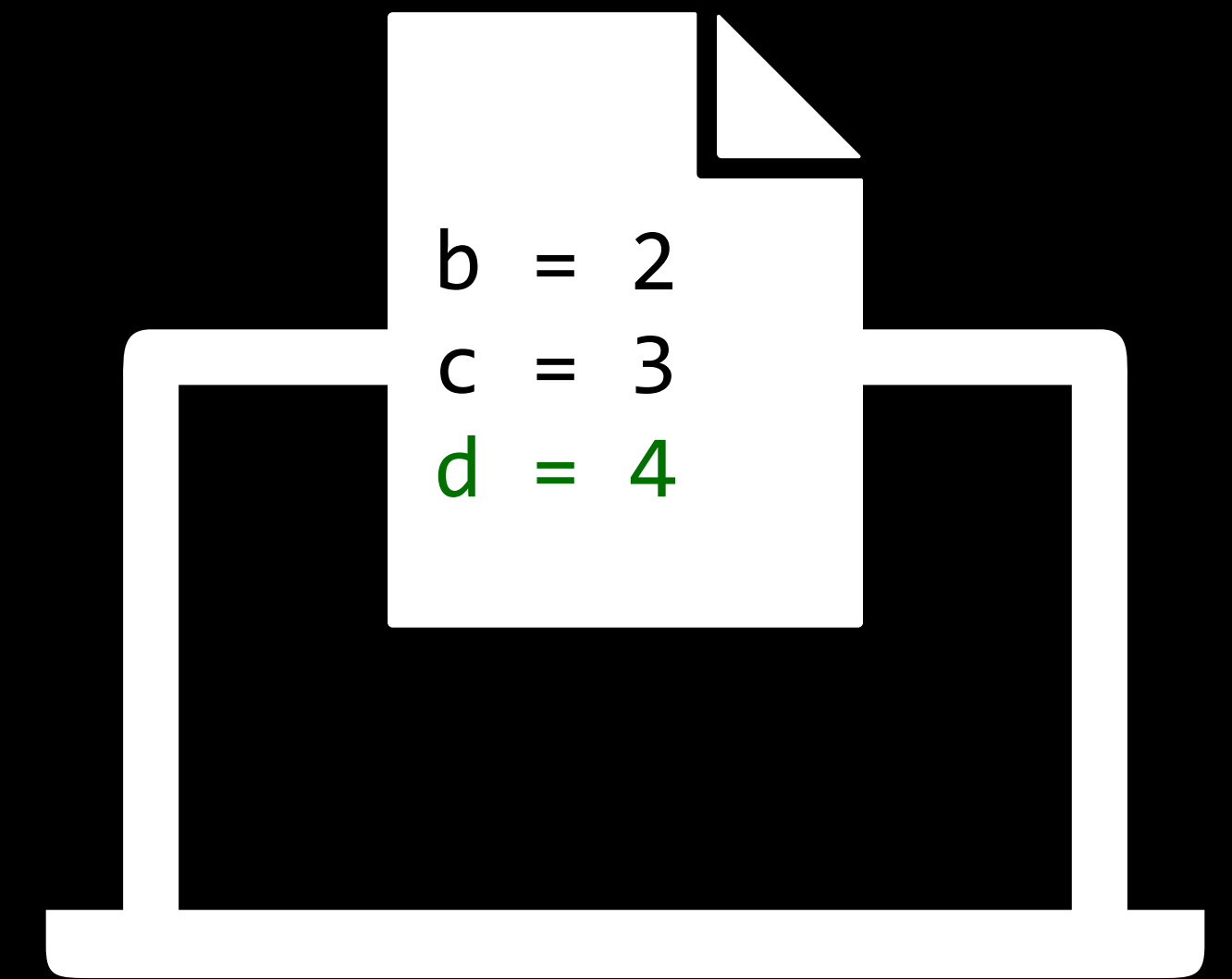
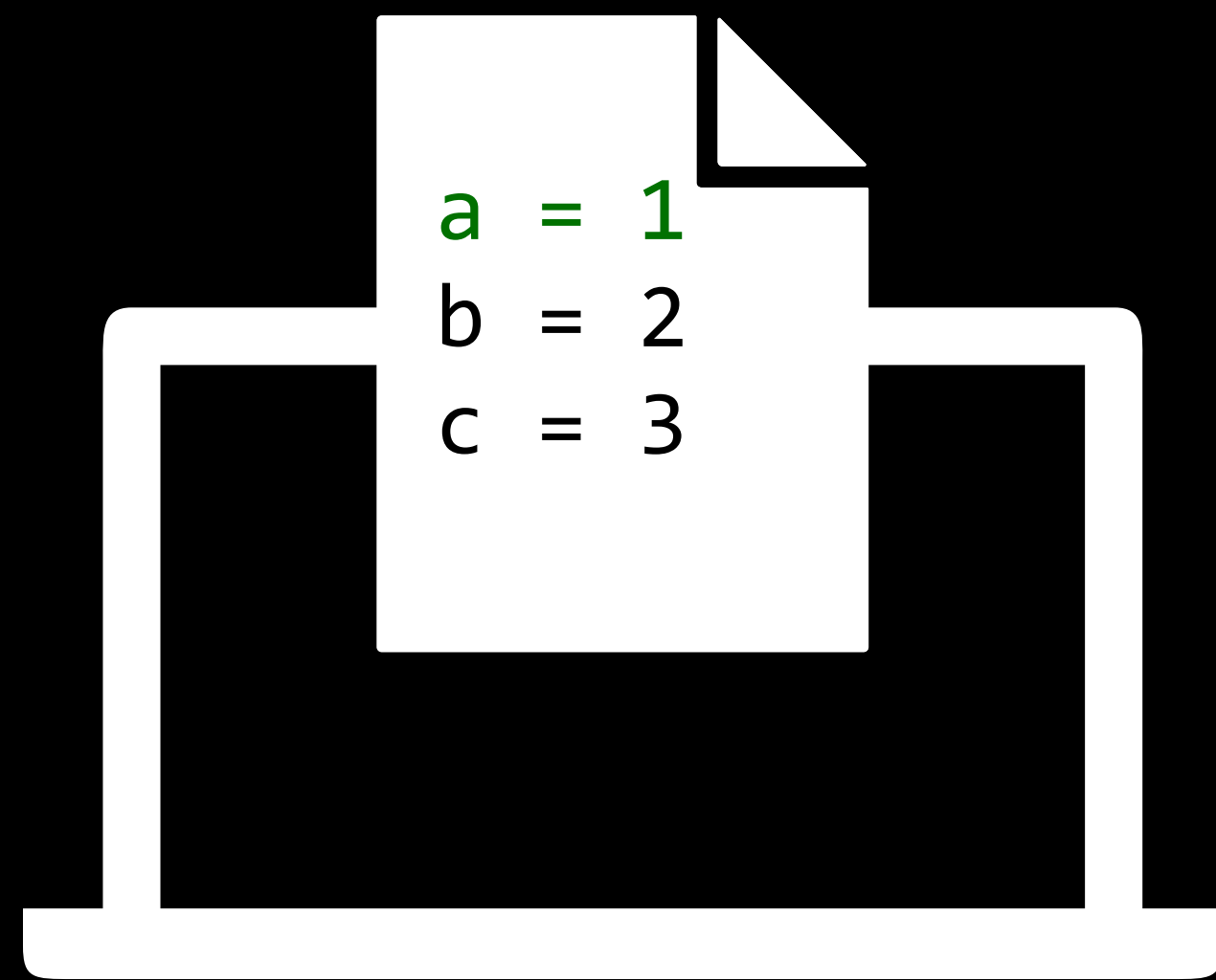
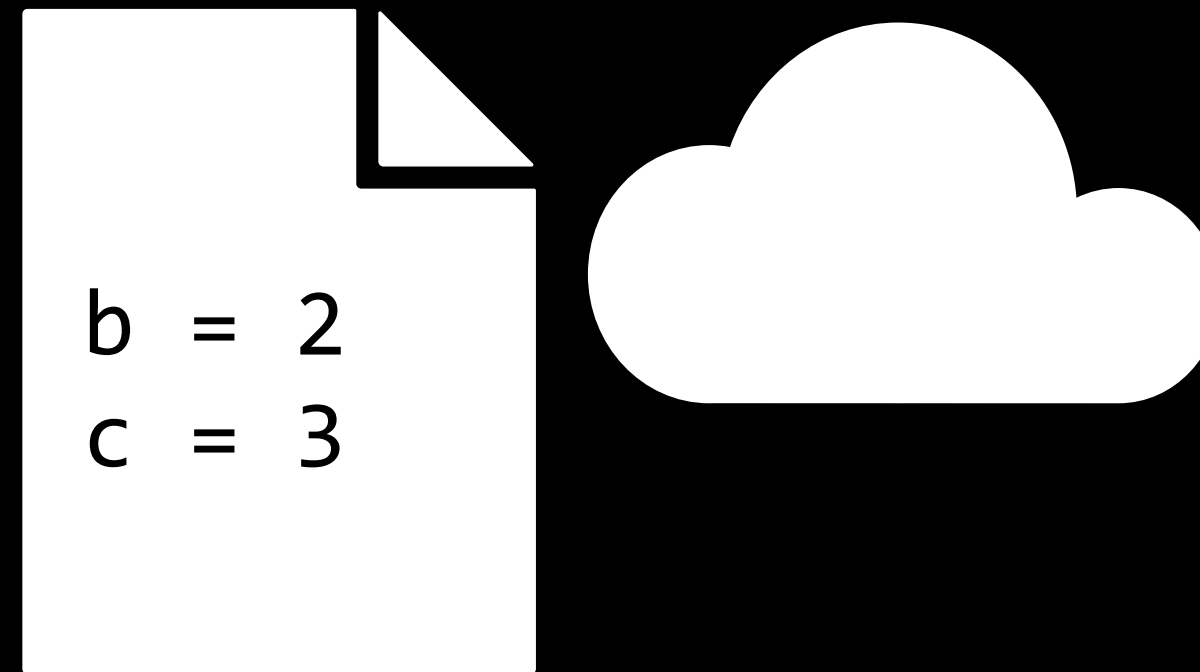
Synchronize changes between different people



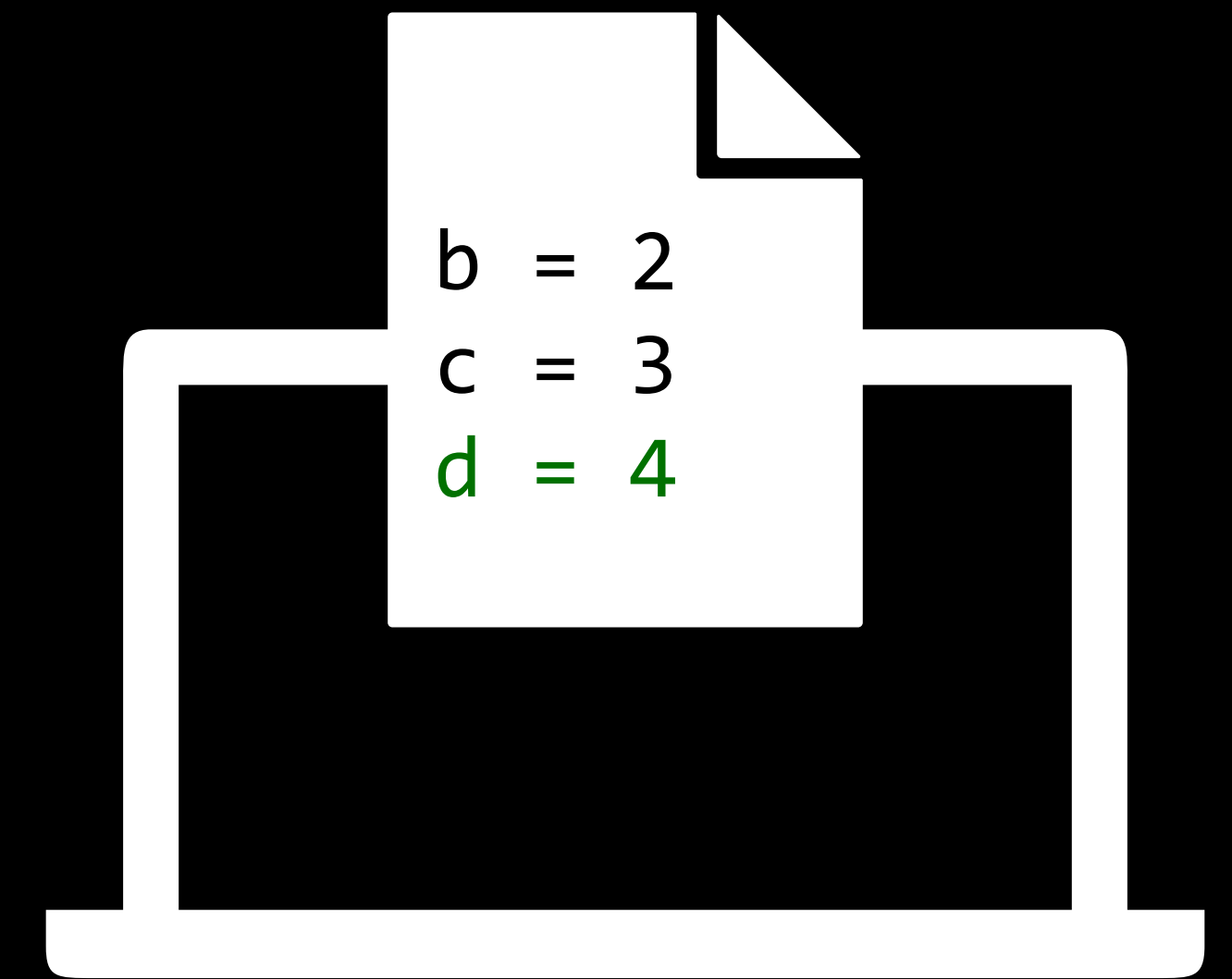
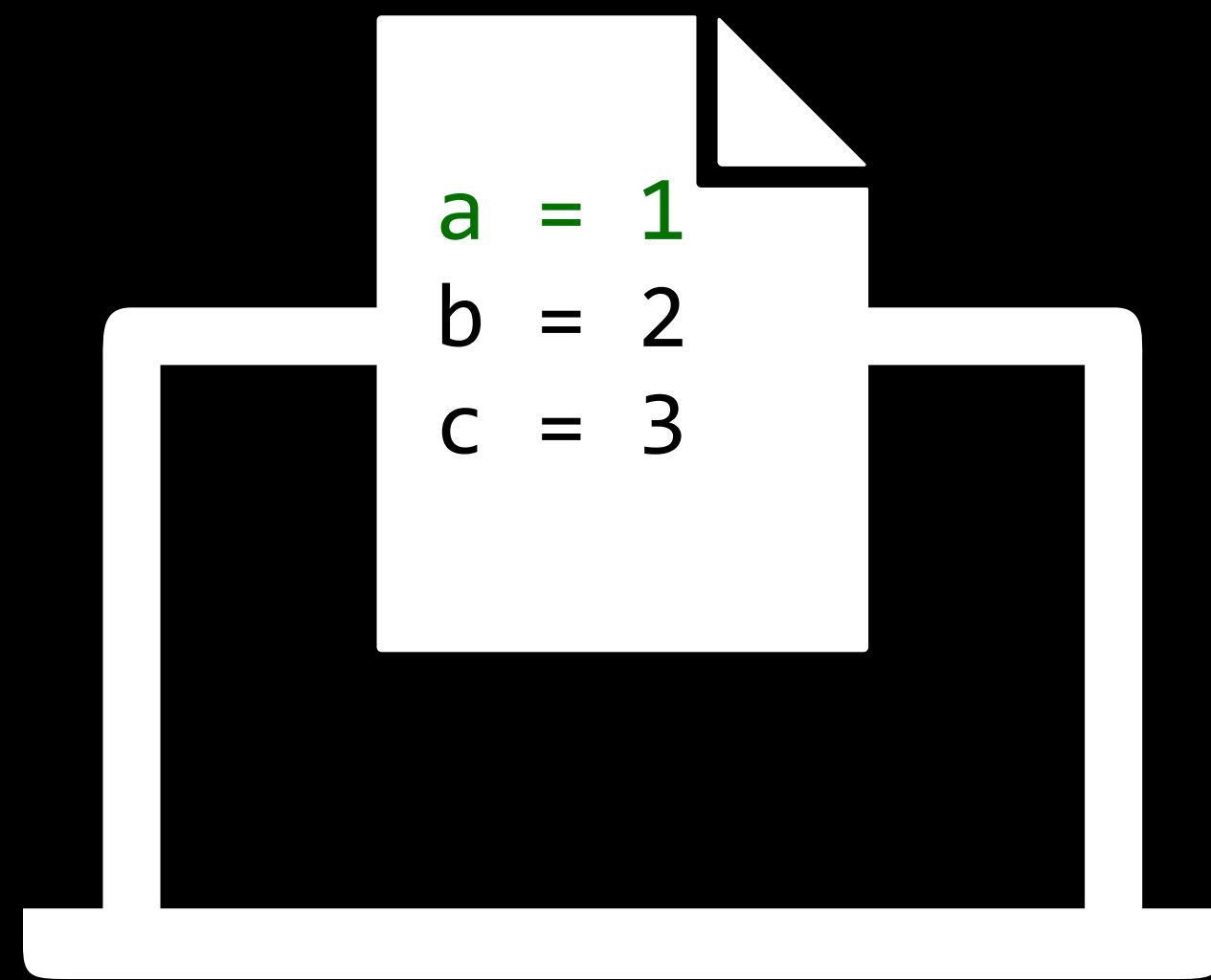
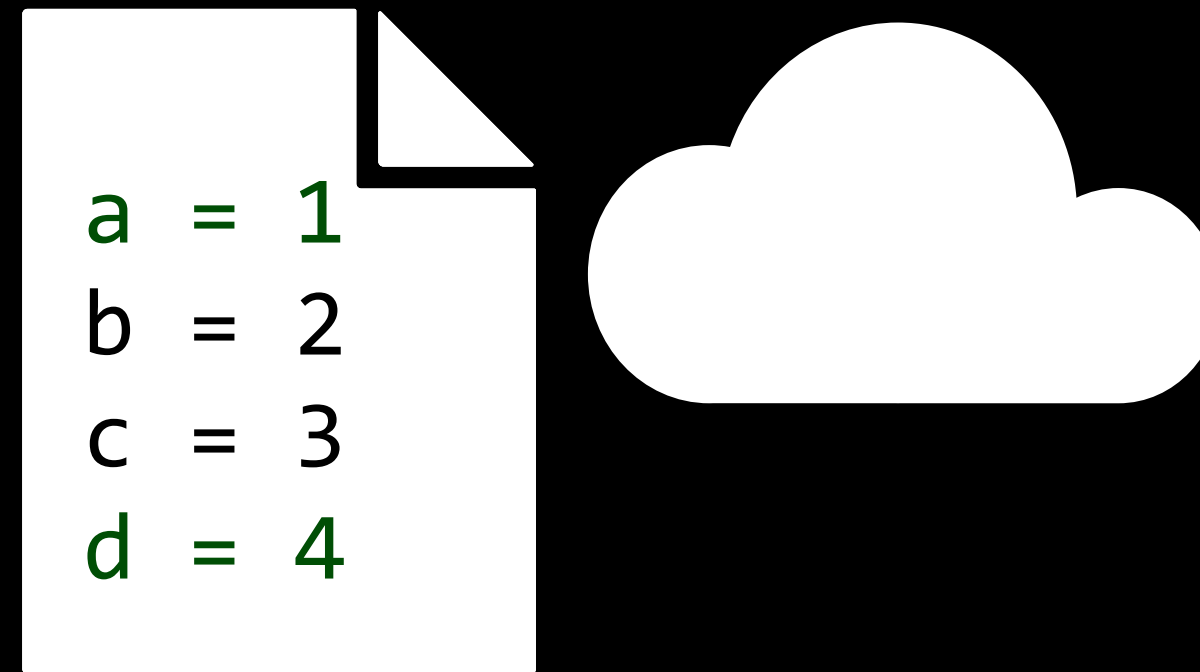
Synchronize changes between different people



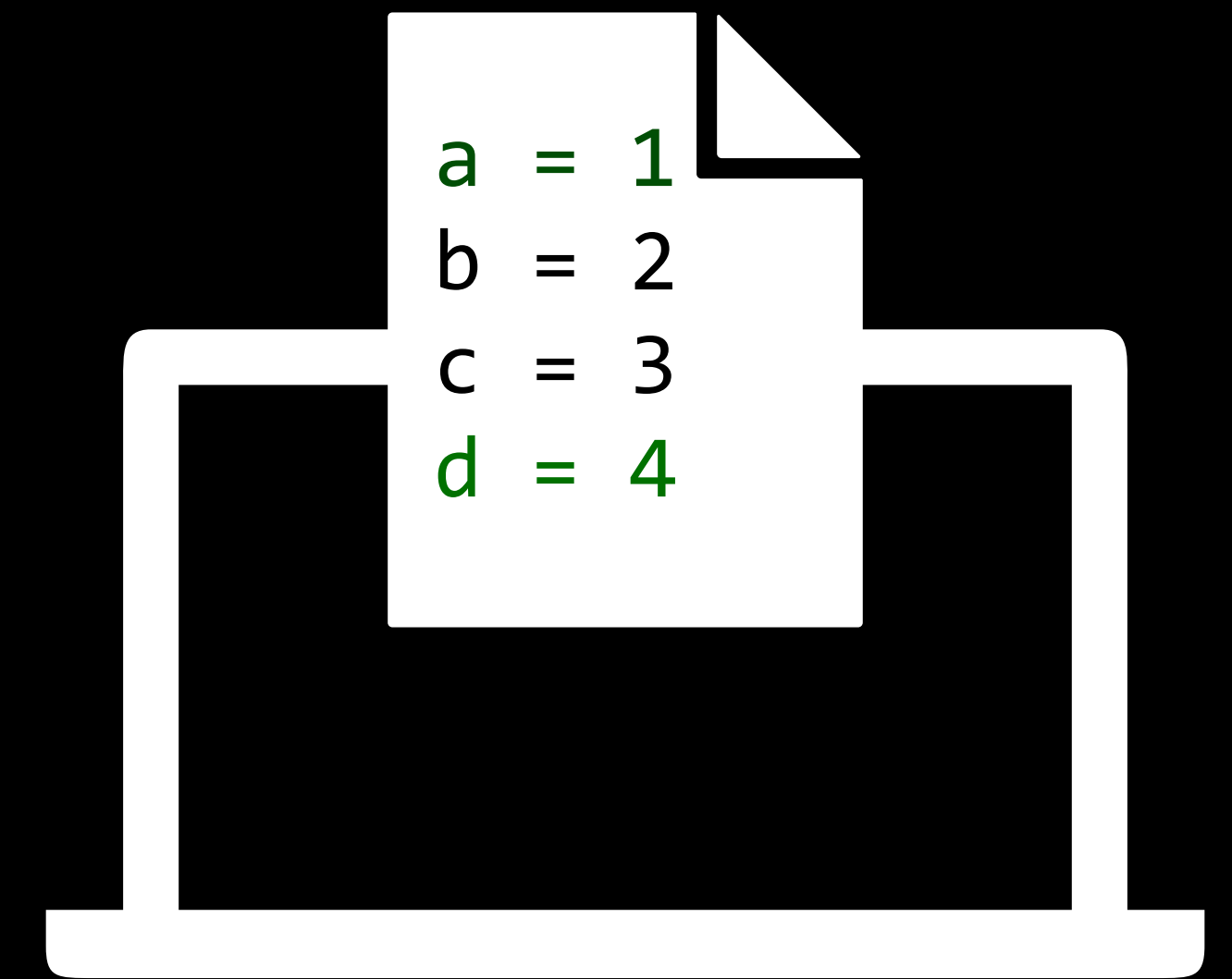
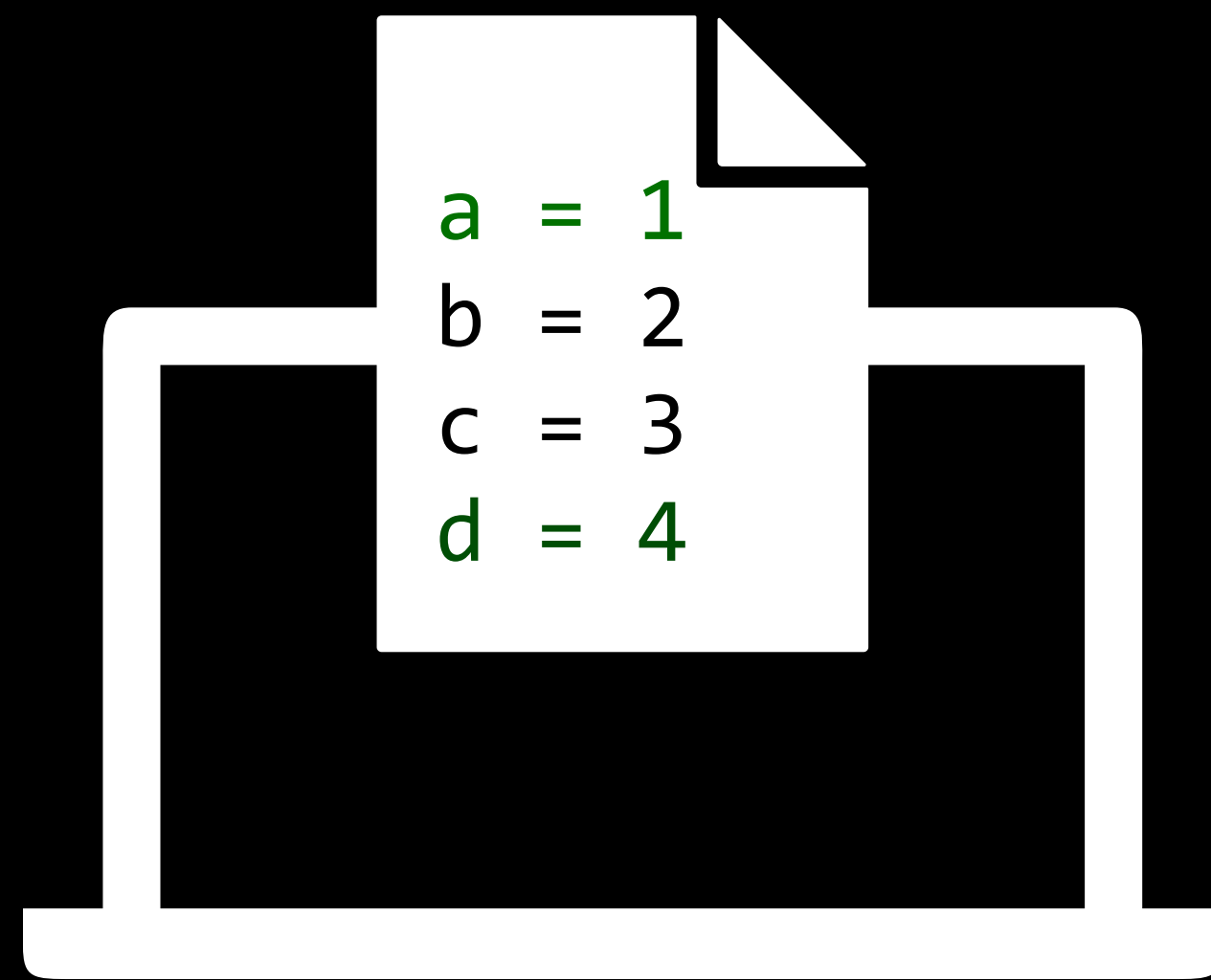
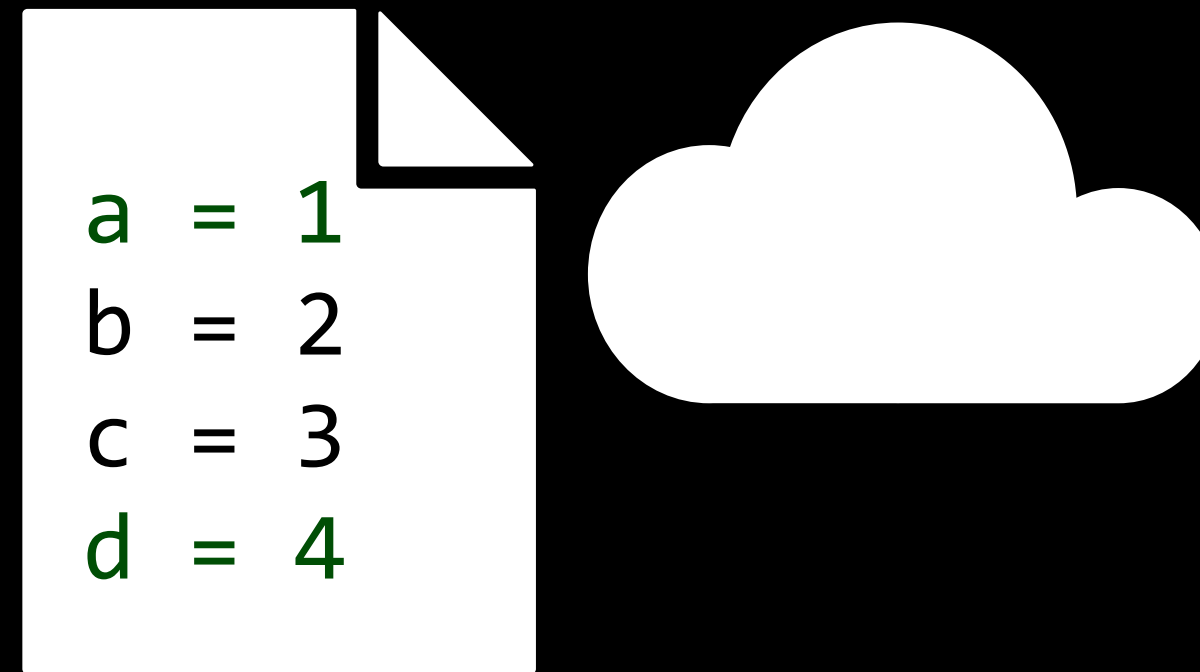
Synchronize changes between different people



Synchronize changes between different people



Synchronize changes between different people



Remotes

Remotes

- Distributed VCS, like Git, require hosting platforms such as:
 1. **GitHub**
 2. Bitbucket
 3. GitLab

GitHub

What is GitHub?

- Platform that offers a cloud-based Git repository hosting + collaboration services
- Provides a "remote" location for storing your git repositories
- A subsidiary of Microsoft since 2018
- Commonly used to host open source software development projects

Authenticating local machine to GitHub

- When you connect to a GitHub repository from Git, you will need to authenticate with GitHub using either HTTPS or SSH.
- If using CS50's Codespace, odds are that you will find SSH to be more convenient
- Visit the link below and follow the steps 1-12 under "SSH":

<https://cs50.readthedocs.io/github/#ssh>

Creating a new remote repository

- Visit <https://github.com> and log in
- Open the menu, and navigate to “Your repositories. Click the green button written “New”
- Alternatively, go to: <https://github.com/new>
- Input a name that is similar to your local repository



New

Connecting local repo. to remote remote

- Set the new remote URL, specifying it as the “origin”:
 - `git remote add origin <remote_URL>`
- Make sure the branch name is main:
 - `git branch -M main`
- Push the changes in your local repository to GitHub.com:
 - `git push -u origin main`

Demo 3

Connecting local repository to remote

Collaboration using GitHub

Collaboration using GitHub

- Fork
- Clone
- Pull changes (== fetch + merge)
- Push changes
- Make Pull Requests (aka PRs)
- Issues and discussions

Forking a repository

- A fork is a new repository that shares code and visibility settings with the original “upstream” repository.
- To fork a repo on GitHub, visit the repo’s home page and click “fork”
- You will then have a fork of the repository on your account, with write access to it

`git clone <remote-URL>`

- Creates a clone of the forked repository from GitHub into your local computer
- Will automatically set the “upstream” relationship, which can be verified via:
 - `git remote -v`
- You can make changes to it with the usual work flow

git pull [origin main]

- Equivalent to running both `git fetch` followed by `git merge`
 - `git fetch`: Fetches changes from the origin by default
 - `git merge`: Merges fetched changes into the current branch
- Always pull before pushing

git push

- Push local changes on the current branch to the corresponding branch on the remote repository

Sync a fork

- When the fork is ahead/behind the upstream by several commits
- You can either discard the changes, or **update the branch** to merge the latest changes from the upstream

Pull Requests (PRs)

- Creating a PR is how you “ask” the maintainers of the project to review your changes and merge them into the upstream repositories
- GitHub automatically checks for merge-ability, provides a way to comment and discuss issues, and more...

Demo 4

Collaborating using GitHub

<https://github.com/innocentmunai/git-seminar-winter-2023>

Software Licences and Open-Source Software

Software License

- A legal instrument (usually by way of contract law) governing the use or redistribution of software
- Two common categories for software under copyright law:
 - Proprietary Software
 - Free and Open-Source Software (FOSS)

Proprietary Software vs FOSS

- Overall difference that lies between the two is the granting of rights to **modify** and **re-use** a software product obtained by a customer/licensee
- FOSS software licenses both rights to the customer and so bundles the modifiable source code with the software (“open-source”)
- Proprietary Software typically does not license these rights and therefore keeps the source code hidden (“closed source”).

Non-free Software

- Proprietary License
 - Traditional use of copyright, no rights need to be granted
- Noncommercial license
 - Grants right for noncommercial use only; could be combined with copyleft
- Trade secret
 - Private internal software; No information made public

FOSS

- ★ Software must have source code provided
- FOSS can be:
 - Public Domain
 - Permissive License
 - Copyleft (Protective license)

Public Domain

- Grants all rights
- No exclusive intellectual property rights apply
- Public-domain-equivalent equivalent licence include:
 - Unlicensed
 - CC0 by Creative Commons

Permissive License

- Grants use rights, including right to relicense
- Allows prioritization
- No exclusive intellectual property rights apply
- Permissive licences include:
 - MIT License — One of the most popular
 - Apache license — By the Apache Software Foundation

Copyleft (Protective license)

- Grants use right, but forbids proprietization
- May be used, modified and distributed freely on condition that the derivative is bound by same conditions
- Copyleft licences include:
 - GNU GPL (General Public License)
- Similar in spirit to CS50's, which is CC BY-NC-SA 4.0 international license.

How do FOSS thrive? 💰

- GitHub has a feature called GitHub Sponsors where people/teams can sponsor FOSS projects
- What other ways?

Resources

- GitHub's Git cheatsheet: <https://training.github.com/downloads/github-git-cheat-sheet.pdf>
- Git Documentation: <http://git-scm.com/docs>
- Git Book: <http://git-scm.com/book/en/v2>
- Open Source Legal Guide: <https://opensource.guide/legal/>

This was CS50 for JDs