Version control with

Marie-Hélène Burle

training@westgrid.ca

September 11, 2020

- Why version control?
- Git
- Configuration
- Documentation
- Troubleshooting & getting help
- Recording history

Break

- Working with branches

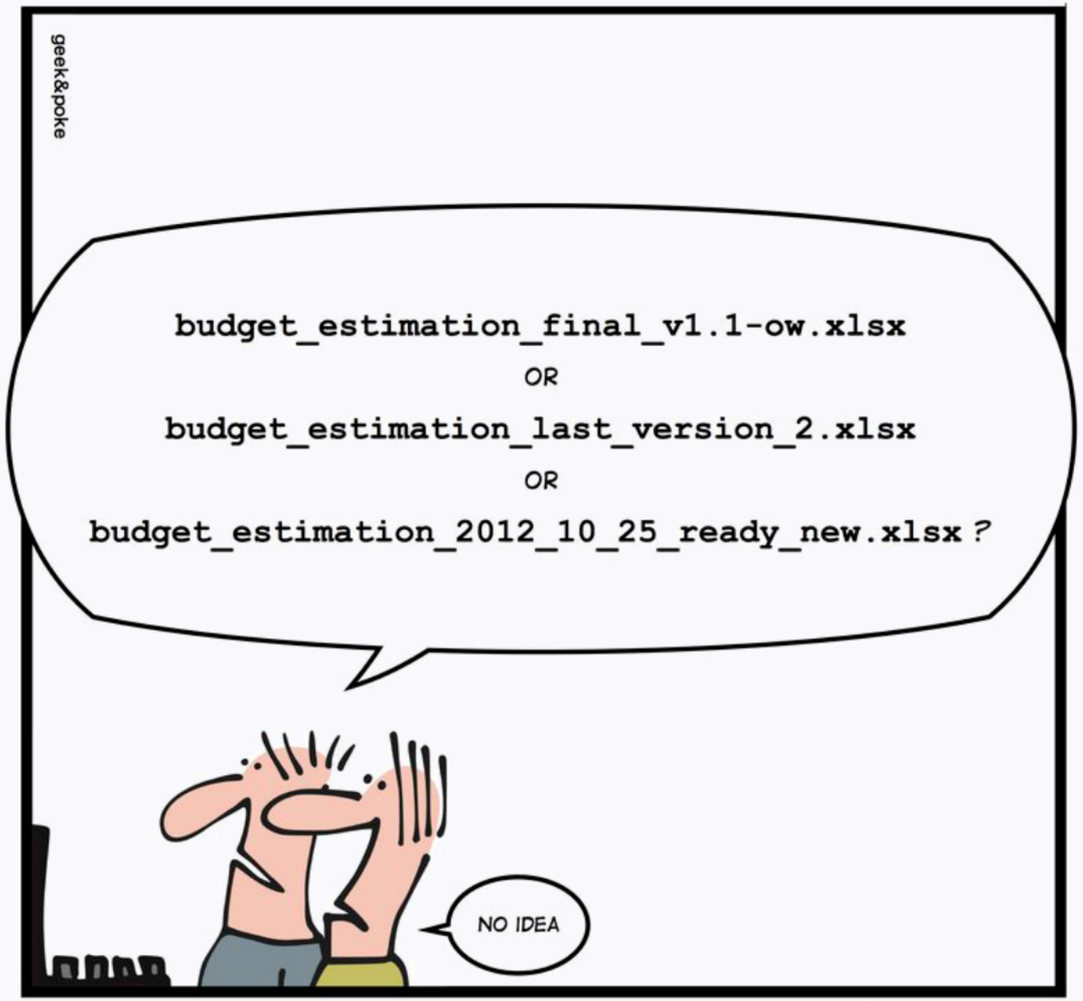
Lunch Break

- Exploring the past
- Undoing

Break

- Remotes
- Collaborating

Why version control?

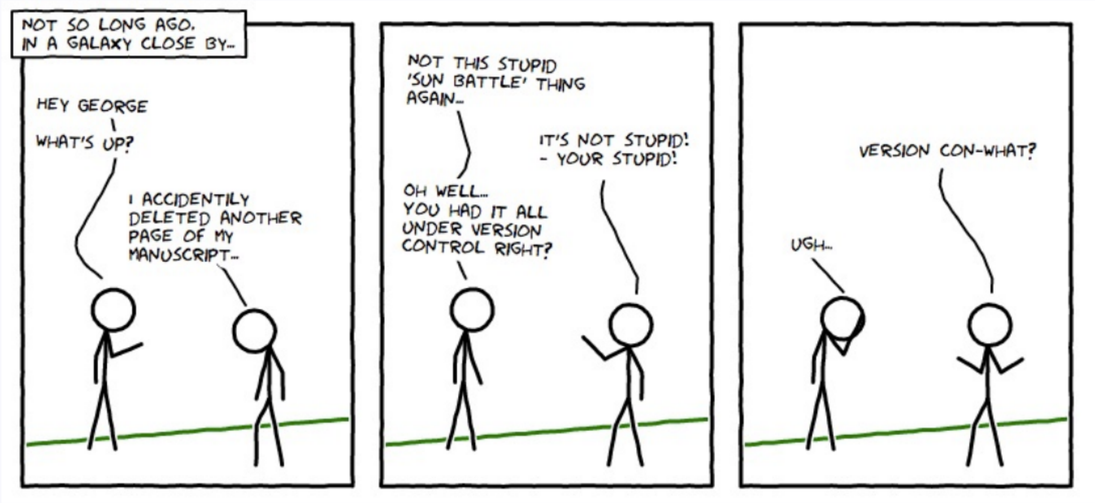


VERSION CONTROL

from Geek&Poke

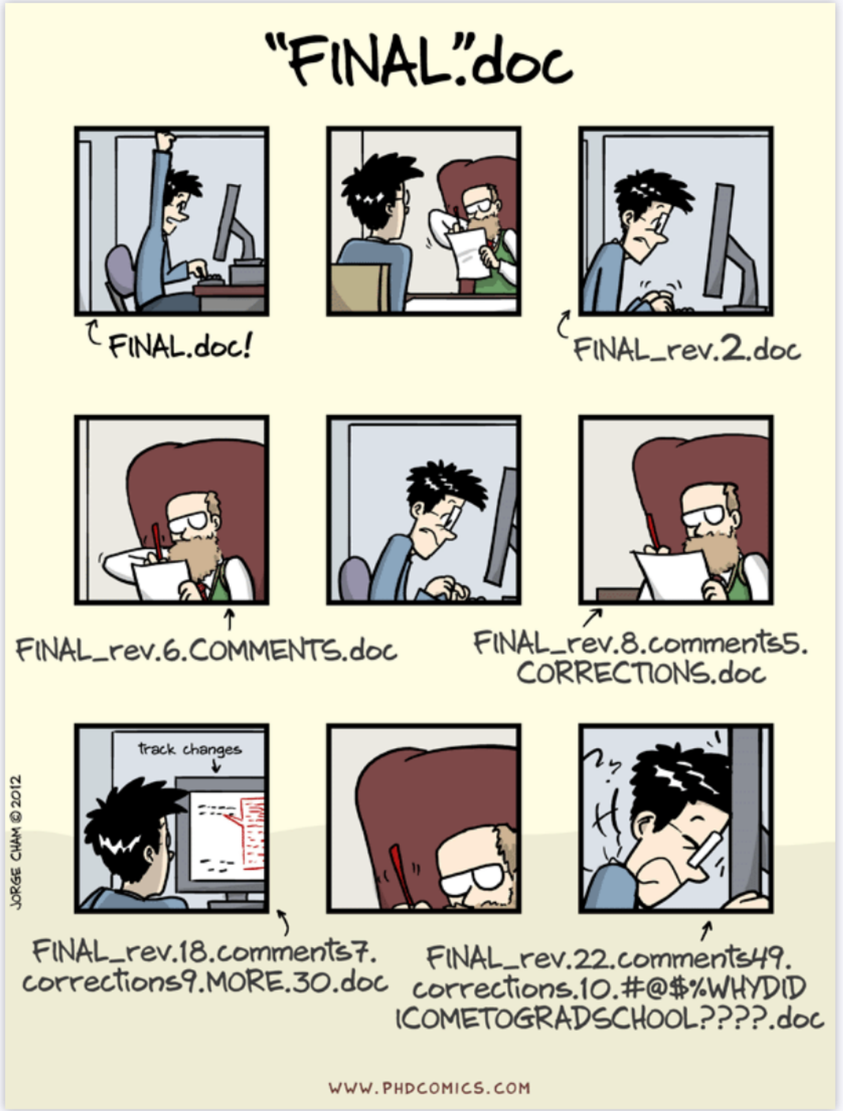
A sophisticated form of backup

> *There are two kinds of people: those who do their backups well and those who will.*

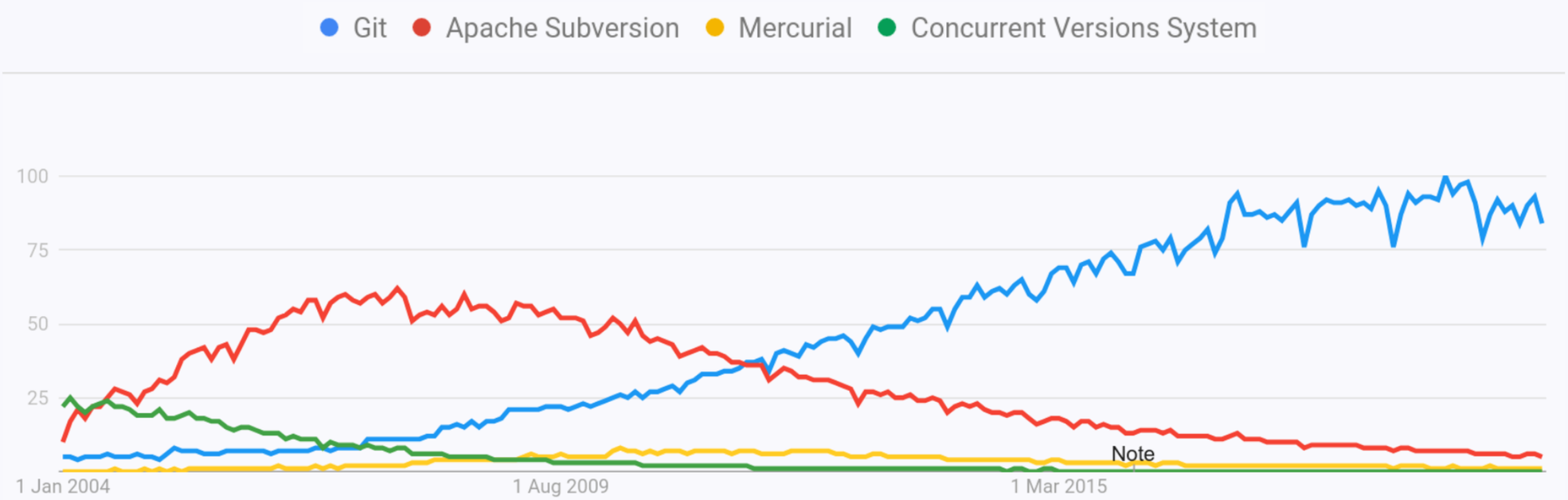


from smutch

But much more



Git



from Google Trends

Git is an open source distributed version control system (DVCS) created in 2005 by Linus Torvalds for the versioning of the Linux kernel during its development.

In distributed version control systems, the full history of projects lives on everybody's machine—as opposed to being only stored on a central server as was the case with centralized version control systems CVCS. This allows offline work, huge speedups, easy branching, and multiple backups. DVCS have taken over CVCS.

Git is extremely powerful and has strong branching capabilities. Since the early 2010s, it has become the most popular DVCS, increasingly rendering other systems quite marginal.

All commands start with `git`.

A typical command is of the form:

```
git <command> [flags] [arguments]
```

Example:

We already saw the following:

```
git config --global "Your Name"
```

Configuration

Global configuration

From anywhere, with the `--global` flag.

There are a number of configurations necessary to set before starting to use Git.

Global configuration

Set the name and email address that will appear as signature of your commits:

```
git config --global user.name "Your Name"
git config --global user.email "your@email"
```

Global configuration

Set the text editor you want to use with Git:

```
git config --global core.editor "editor"  # e.g. "nano", "vim", "emacs"
```

Global configuration

Format line endings properly:

```
git config --global core.autocrlf input   # if you are on macOS or Linux
git config --global core.autocrlf true    # if you are on Windows
```

Global configuration

To see your current configuration:

```
git config --list
```

Project-specific configuration

You can set configurations specific to a single repository (e.g. maybe you want to use a different email address for a certain project).

In that case, **make sure that you are in the repository you want to customize** and run the command without the `--global` flag.

Example:

```
cd /path/to/project
git config user.email "your_other@email"
```

Documentation

Man pages

You can access the **man page** for a git command with either of:

```
git <command> --help
git help <command>
man git-<command>
```

Note:
Throughout this workshop, I will be using `<` *and* `>` *to indicate that an expression needs to be replaced by the appropriate expression* (*without those signs*).

Man pages

Example:

In [ ]:
```
man git-commit
```

Command options

To get a list of the **options** for a command, run:

```
git <command> -h
```

Command options

Example:

In [ ]:
```
git commit -h
```

Resources

Official Git manual

Git Software Carpentry lesson

Git tutorial by Atlassian

WestGrid Summer School 2020 Git course

WestGrid workshop "Collaborating through GitHub"

WestGrid workshop "Contributing to GitHub projects"

Troubleshooting
Getting help

from xkcd.com

"Listen" to Git!

Git is extremely verbose: by default, it will return lots of information. Read it!

These messages may feel overwhelming at first, but:

- they will make more and more sense as you gain expertise
- they often give you clues as to what the problem is
- even if you don't understand them, you can use them as Google search terms

(Re-read) the doc

As I have no memory, I need to check the man pages all the time. That's ok! It is quick and easy.

For more detailed information and examples, I really like the Official Git manual.

Search online

- Google
- WestGrid workshop: "Collaborating through GitHub"
- Stack Overflow [git] tag

Don't panic
Be analytical

It is easy to panic and feel lost if something doesn't work as expected.

Take a breath and start with the basis:

- make sure you are in the repo (`pwd`) and the files are where you think they are (`ls -a`)
- inspect the repository (`git status`, `git diff`, `git log`). Make sure not to overlook what Git is "telling" you there

Commit and push often to be safe.

from jscript

Recording history

Create the project root

1. Navigate to the location where you want to create your project.
2. Create a new directory with the name of your project.

   Never use spaces in names and paths.

In [ ]:
```
pwd
```

In [ ]:
```
cd ~/parvus/ptmp
```

In [ ]:
```
pwd
```

In [ ]:
```
ls
```

In [ ]:
```
mkdir ocean_temp
```

In [ ]:
```
ls
```

Put the project under version control

Make sure to enter your new directory before initializing version control.

A classic mistake leading to lots of confusion is to run `git init` outside the root of the project.

In [ ]:
```
pwd
```

In [ ]:
```
cd ocean_temp
```

In [ ]:
```
pwd
```

In [ ]:
```
ls -a
```

In [ ]:
```
git init
```

In [ ]:
```
ls -a
```

Put the project under version control

Make sure to enter your new directory before initializing version control.

A classic mistake leading to lots of confusion is to run `git init` outside the root of the project.

```
In [ ]: pwd
```

```
In [ ]: cd ocean_temp
```

```
In [ ]: pwd
```

```
In [ ]: ls -a
```

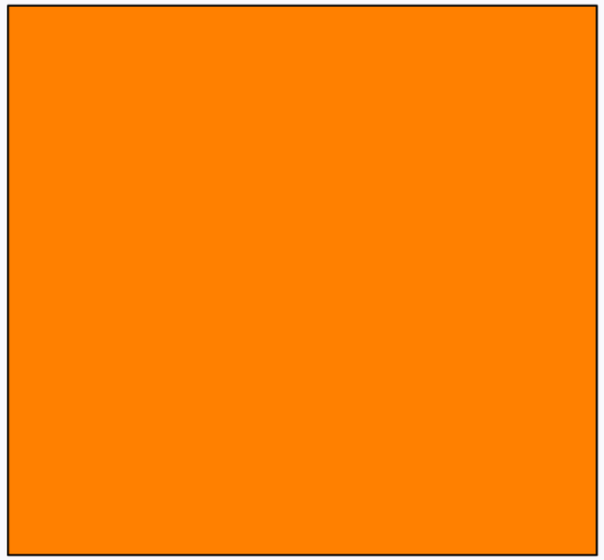
```
In [ ]: git init
```

```
In [ ]: ls -a
```

```
In [ ]: ls -a .git
```

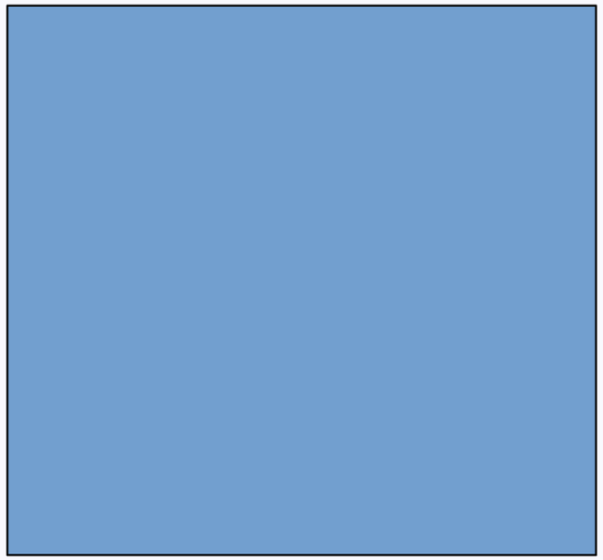
```
In [ ]: git status
```

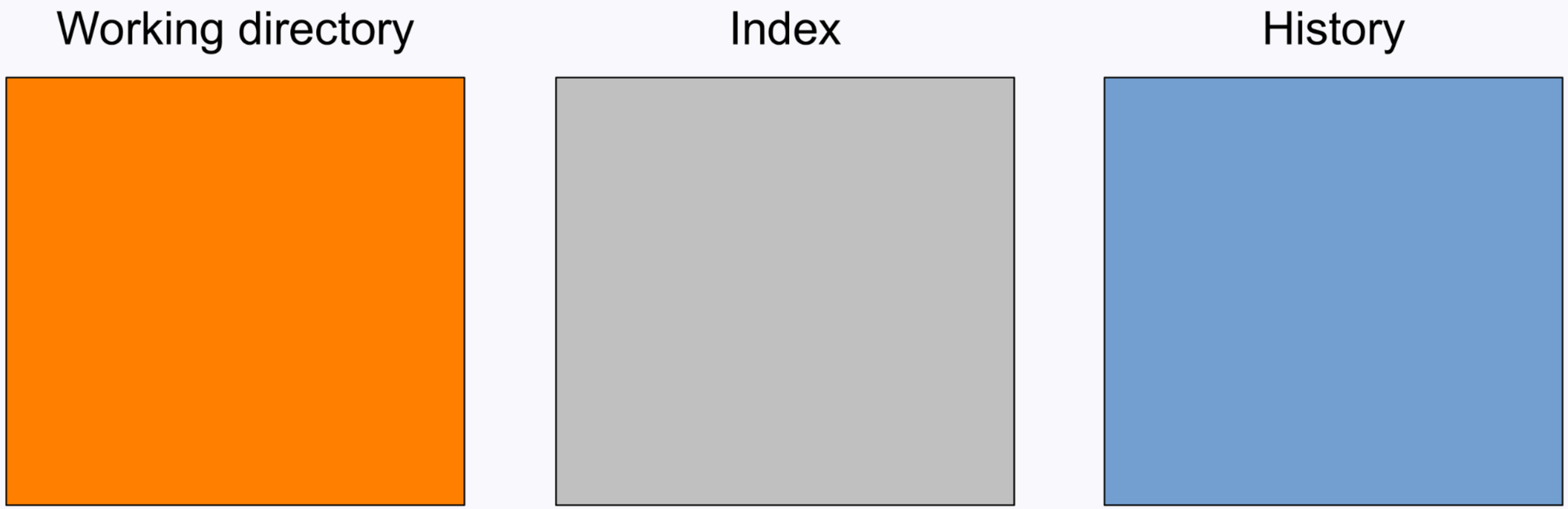
Working directory



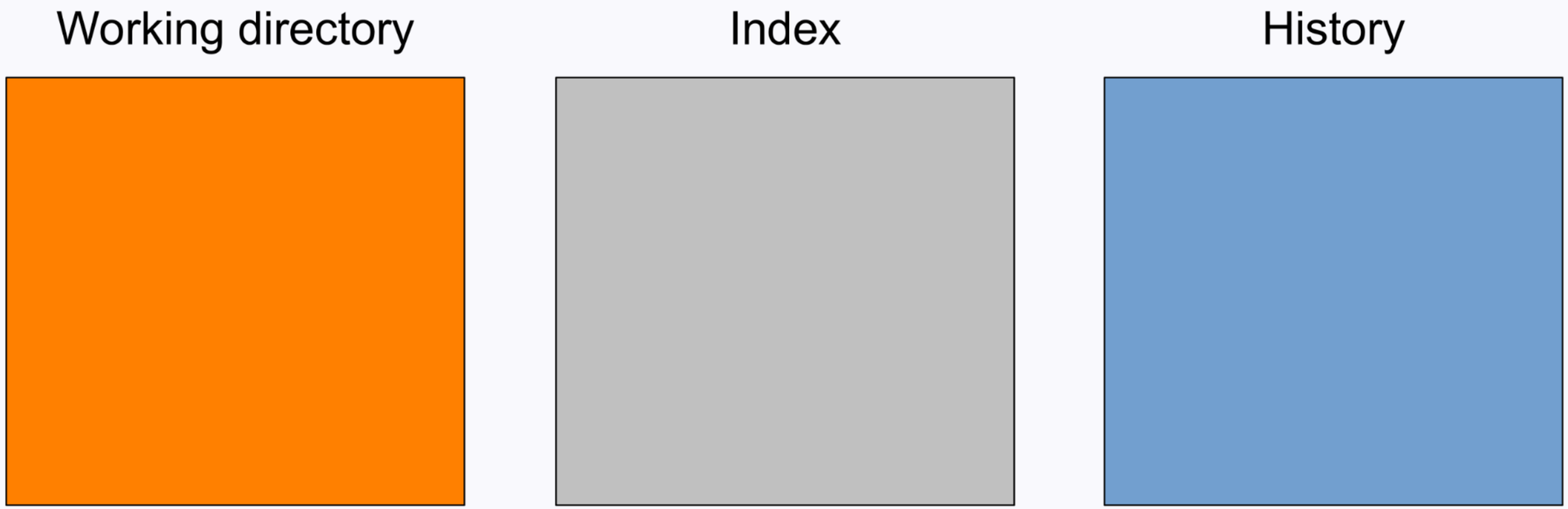
Staging area

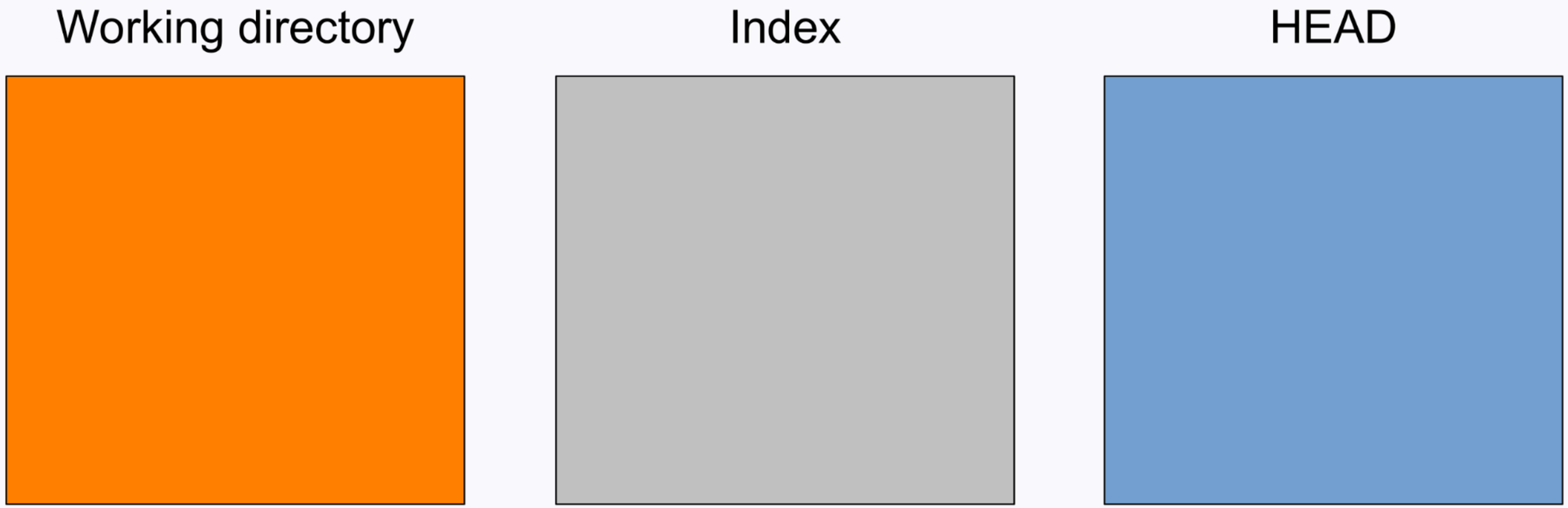
History

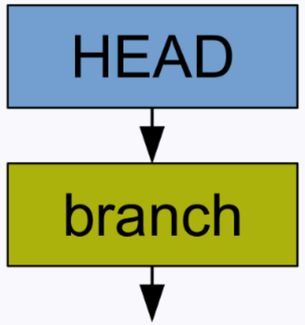


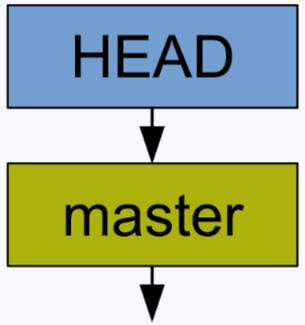

Working directory
Index
History

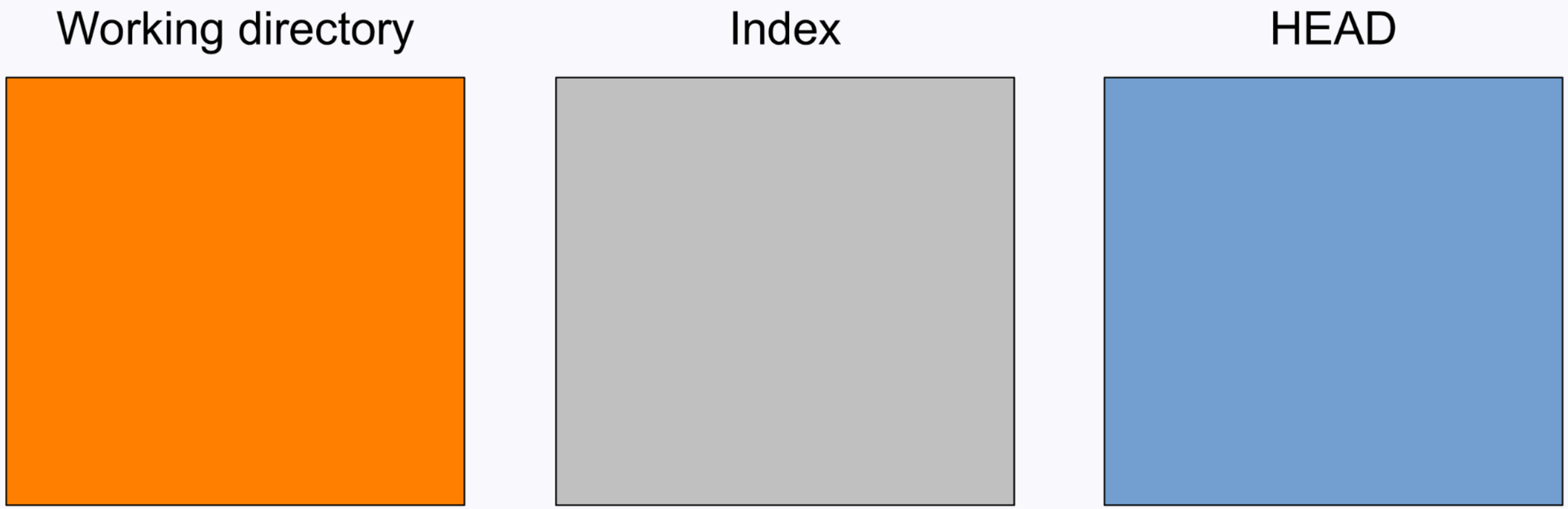

Outside .git/
.git/index
.git/objects

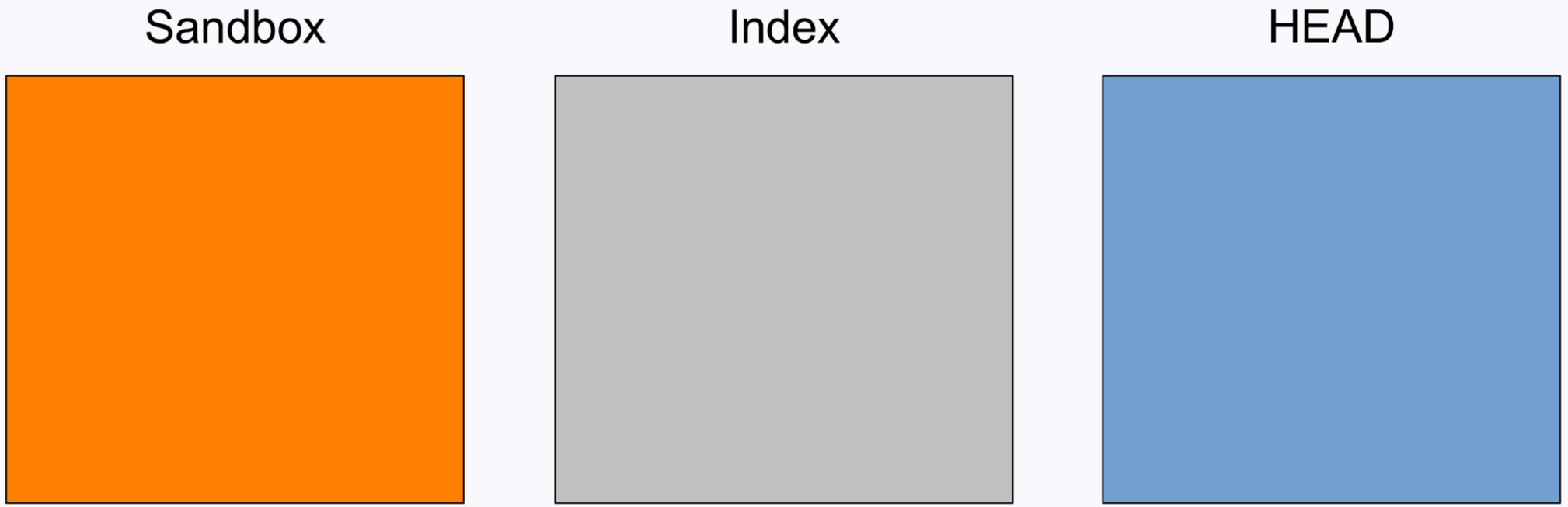

Working directory
Index
History


Working directory
Index
HEAD


HEAD
branch


HEAD
master


Working directory
Index
HEAD


Sandbox
Index
HEAD

Add first file

Note: Git—which is such a powerful tool—works on any text files.
If you write your manuscript as a text file (e.g. `.org`, `.md`, `.Rmd`, `.txt`, `.ipynb`) rather than a MS Word or LibreOffice Writer file, you can put it under version control.
This has countless advantages, from easy versioning to easy collaboration.

In [ ]:
```
echo "import numpy as np

years = list(range(2001, 2020))" > src/enso_model.py
```

In [ ]:
```
tree
```

In [ ]:
```
git status
```

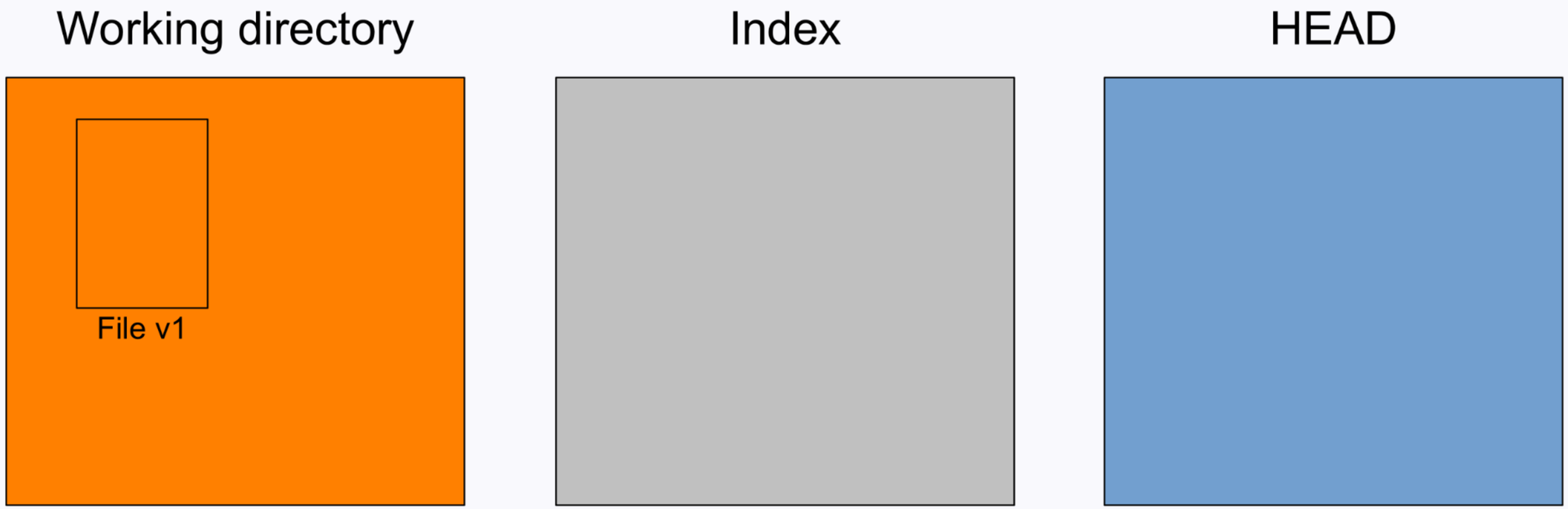
Create a sensible project structure

In [ ]:
```
mkdir src result ms data
```

In [ ]:
```
ls -a
```

In [ ]:
```
tree
```

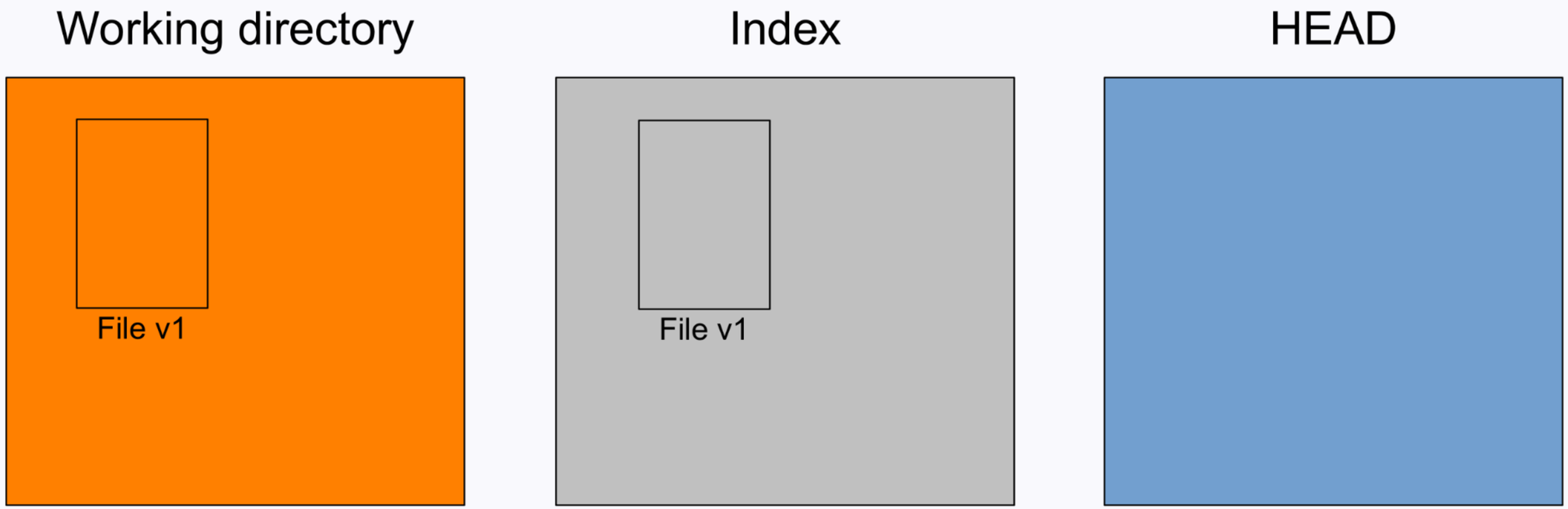
In [ ]:
```
git status
```


Working directory
Index
HEAD
File v1

Stage our file

In [ ]:
```
git add .
```

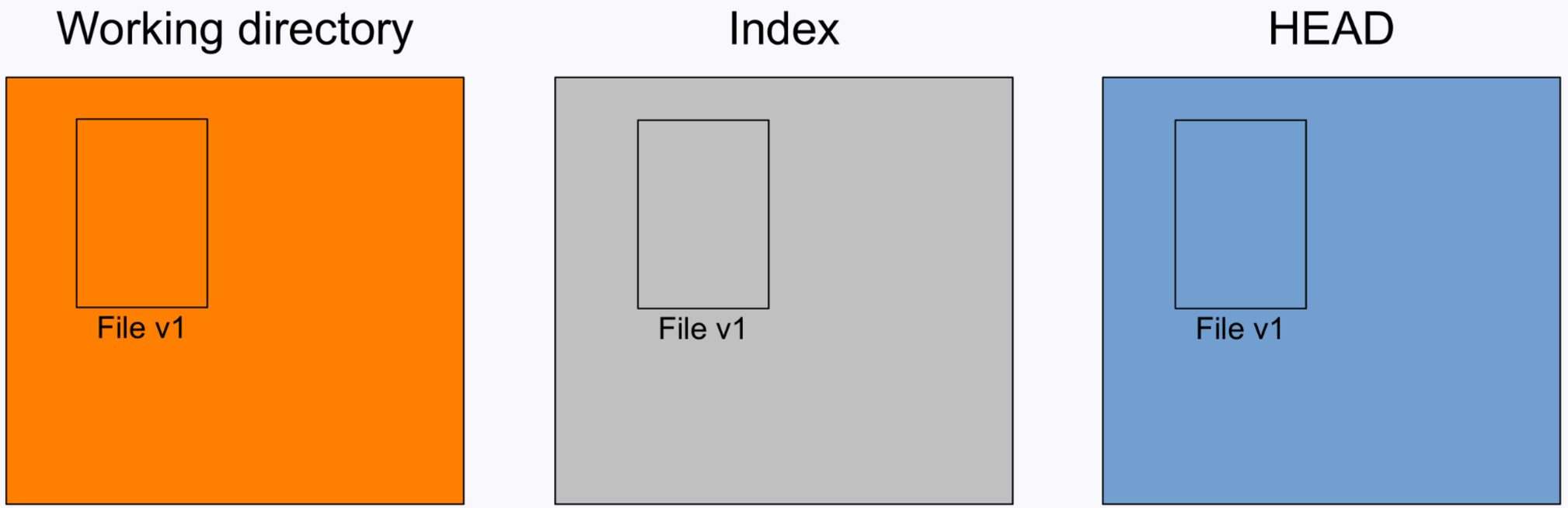
In [ ]:
```
git status
```

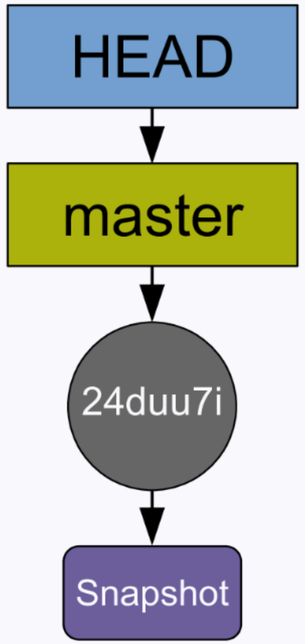

Working directory
Index
HEAD
File v1
File v1

Create a first snapshot (the initial commit)

In [ ]:
```
git commit -m "Initial commit"
```

In [ ]:
```
git status
```


Working directory
File v1
Index
File v1
HEAD
File v1


HEAD
master
24duu7i
Snapshot

SHA-1 checksum

Each commit is identified by a unique 40-character SHA-1 checksum. People usually refer to it as a “hash”.

The short form of a hash only contains the first 7 characters, which is generally sufficient to identify a commit.

After you committed, Git gave you the short form of the hash of your first commit.

On writing good commit messages

from xkcd.com

- Use the present tense
- The first line is a summary of the commit and is less than 50 characters long
- Leave a blank line below
- Then add the body of your commit message with more details

- Use the present tense
- The first line is a summary of the commit and is less than 50 characters long
- Leave a blank line below
- Then add the body of your commit message with more details

Example of a good commit message:

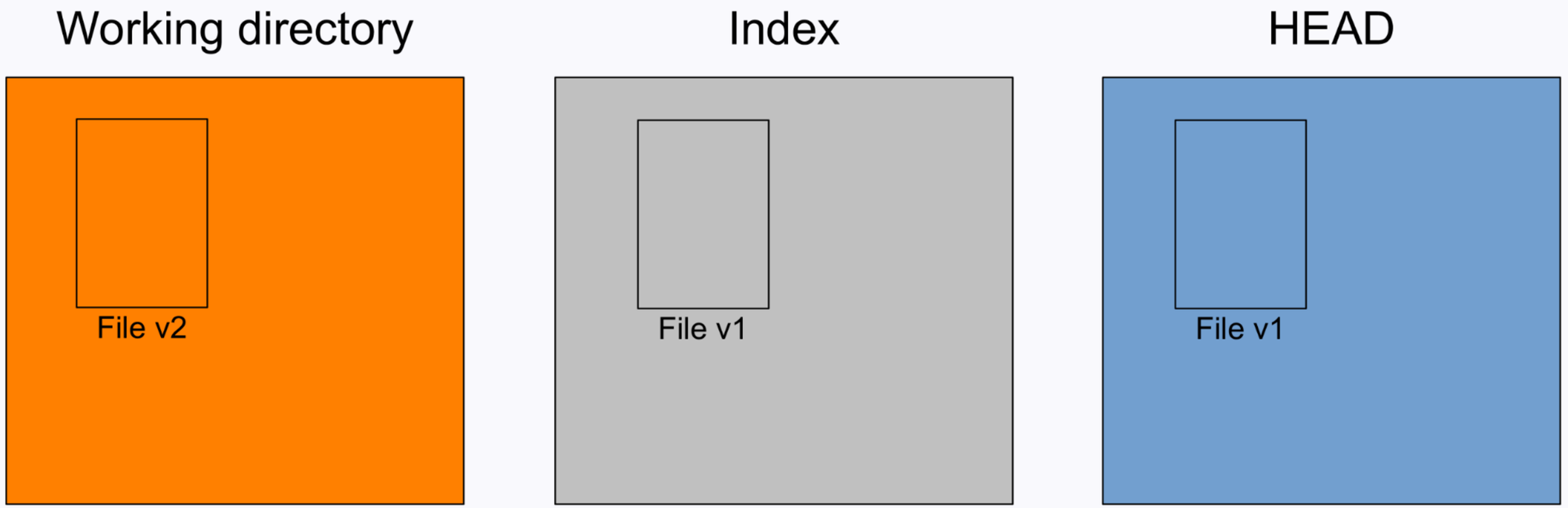
```
git commit -m "Reduce boundary conditions by a
factor of 0.3

Update boundaries
Rerun model and update table
Rephrase method section in ms"
```

Let's make changes to our file

In [ ]:
```
emacsclient -c src/enso_model.py  # Replace 'emacsclient -c' by your text editor of choice
```

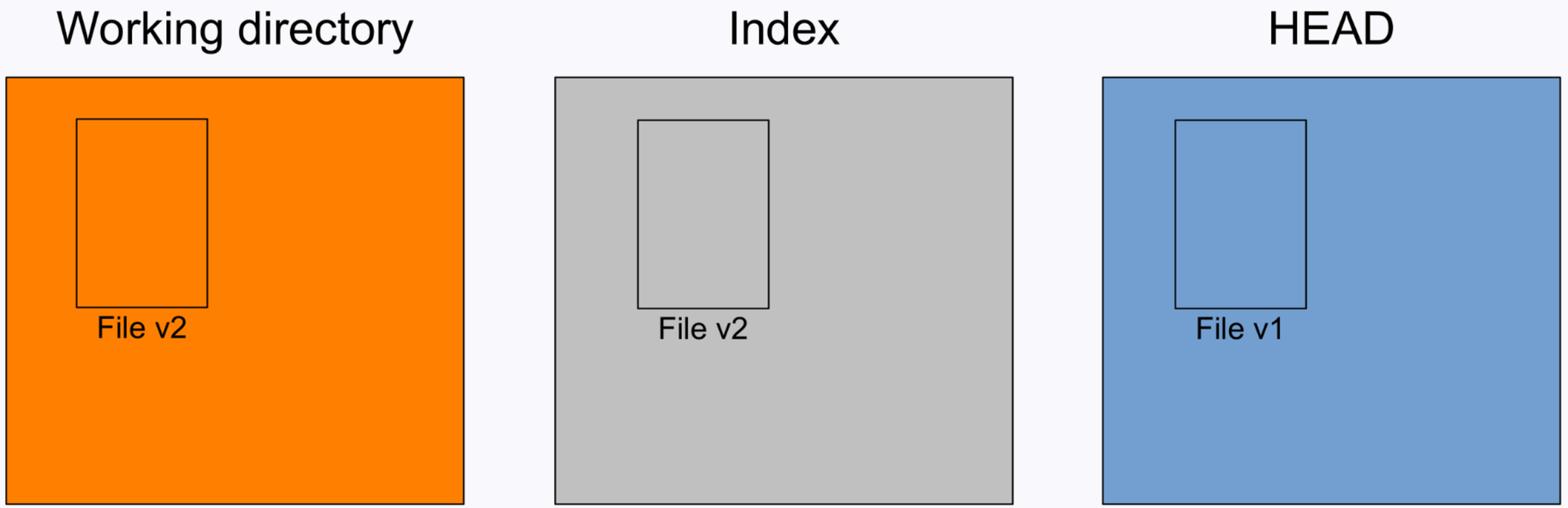
In [ ]:
```
git status
```


Working directory
File v2
Index
File v1
HEAD
File v1

Stage our file again

In [ ]:
```
git add .
```

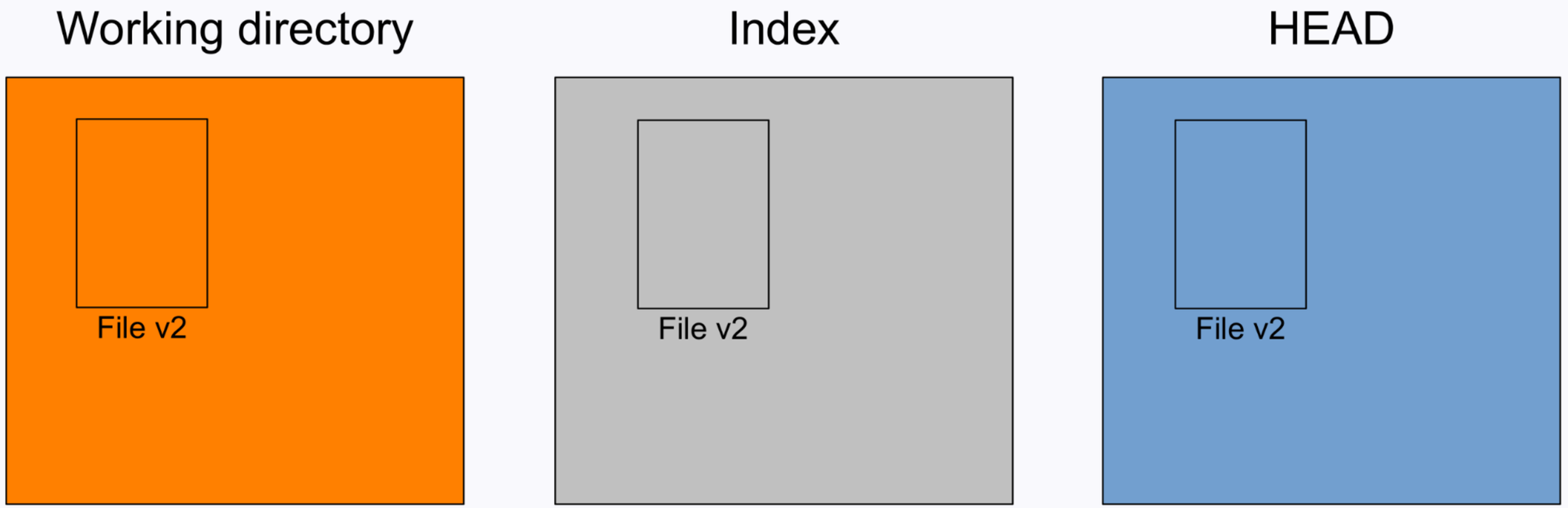
In [ ]:
```
git status
```

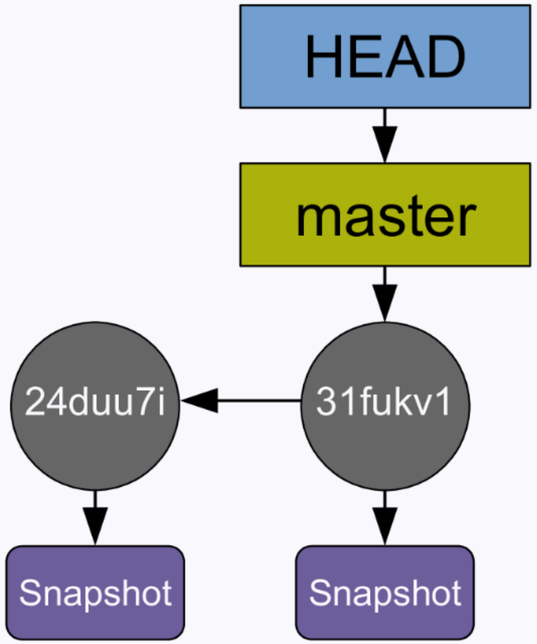

Working directory
File v2
Index
File v2
HEAD
File v1

Create a second commit

In [ ]:
```
git commit -m "Modify enso script"
```

In [ ]:
```
git status
```


Working directory
File v2
Index
File v2
HEAD
File v2


HEAD
master
24duu7i
31fukv1
Snapshot
Snapshot

Excluding from version control

There are files you really **should** put under version control, but there are files you shouldn't.

Put under vc

- Scripts
- Manuscripts and notes
- Makefile and the like

Do *not* put under vc

- Non-text files (e.g. images, office documents)
- Outputs that can be recreated by running code

Excluding from version control

You want to have a clean working directory, so you need to tell Git to ignore those files.

You do this by adding them to a file that you create in the root of the project called `.gitignore`.

In [ ]:
```
touch result/graph.png
```

In [ ]:
```
tree
```

In [ ]:
```
git status
```

In [ ]:
```
echo /result/ > .gitignore
```

In [ ]:
```
cat .gitignore
```

In [ ]:
```
git status
```

.gitignore rules

Each line in a `.gitignore` file specifies a pattern.

Blank lines are ignored and can serve as separators for readability.

Lines starting with `#` are comments.

To add patterns starting with a special character (e.g. `#`, `!`), that character needs escaping with `\`.

Trailing spaces are ignored unless they are escaped with `\`.

`!` negates patterns (matching files excluded by previous patterns become included again). **However** it is not possible to re-include a file if one of its parent directories is excluded (Git doesn't list excluded directories for performance reasons). One way to go around that is to force the inclusion of a file which is in an ignored directory with the option `-f`.

Example: `git add -f <file>`

Example: `git add -f <file>`

Patterns ending with `/` match directories. Otherwise patterns match both files and directories.

`/` at the beginning or within a search pattern indicates that the pattern is relative to the directory level of the `.gitignore` file. Otherwise the pattern matches anywhere below the `.gitignore` level.

> *Examples*:
>
> - `foo/bar/` *matches the directory* `foo/bar`*, but not the directory* `a/foo/bar`
> - `bar/` *matches both the directories* `foo/bar` *and* `a/foo/bar`

`*` matches anything except `/`.

`?` matches any one character except `/`.

The range notation (e.g. `[a-zA-Z]`) can be used to match one of the characters in a range.

A leading `**/` matches all directories.

> *Example*: `**/foo` *matches file or directory* `foo` *anywhere. This is the same as* `foo`

A trailing `/**` matches everything inside what it precedes.

Examples:

- `foo/bar/` *matches the directory* `foo/bar`, *but not the directory* `a/foo/bar`
- `bar/` *matches both the directories* `foo/bar` *and* `a/foo/bar`

`*` matches anything except `/`.

`?` matches any one character except `/`.

The range notation (e.g. `[a-zA-Z]`) can be used to match one of the characters in a range.

A leading `**/` matches all directories.

Example: `**/foo` *matches file or directory* `foo` *anywhere. This is the same as* `foo`

A trailing `/**` matches everything inside what it precedes.

Example: `abc/**` *matches all files (recursively) inside directory* `abc`

`/**/` matches zero or more directories.

Example: `a/**/b` *matches* `a/b`, `a/x/b`, *and* `a/x/y/b`

Tagging

Annotated tag

In [ ]:
```
git tag
```

In [ ]:
```
git tag -a J_Climate_2009 -m "State of project at the publication of paper"
```

In [ ]:
```
git show J_Climate_2009
```

In [ ]:
```
git tag
```

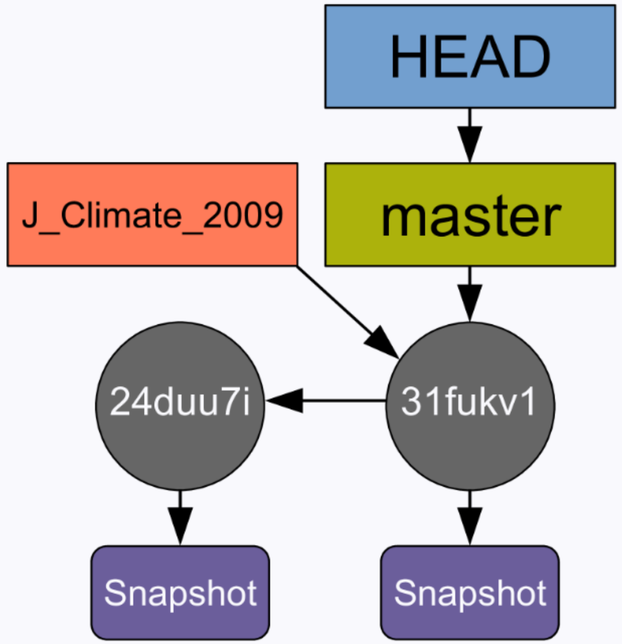
Tagging

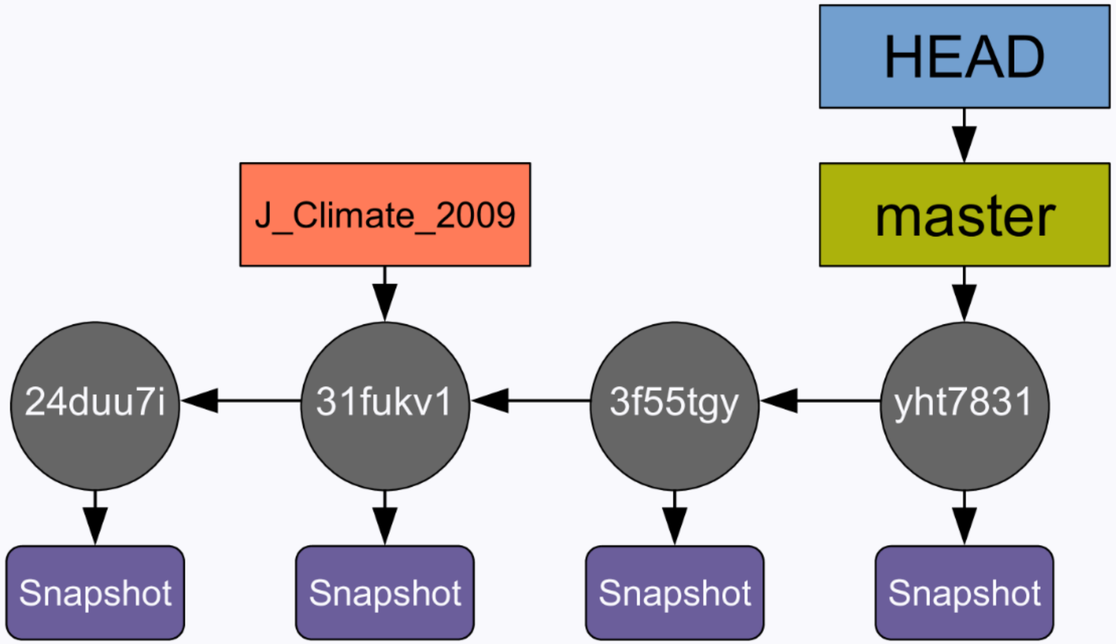
Leightweight tag

In [ ]:
```
git tag J_Climate_2009_light
```

In [ ]:
```
git show J_Climate_2009_light
```

In [ ]:
```
git tag
```


HEAD
J_Climate_2009
master
24duu7i
31fukv1
Snapshot
Snapshot


HEAD
master
J_Climate_2009
24duu7i
31fukv1
3f55tgy
yht7831
Snapshot
Snapshot
Snapshot
Snapshot

Tagging

Deleting tags

In [ ]:
```
git tag -d J_Climate_2009_light
```

In [ ]:
```
git tag
```

Let's create more selective snapshots

We made our first commit with:

```
git add .
git commit -m "Initial commit"
```

`git add .` stages all new changes in the repo.

It is even possible to commit all changes to the tracked files, staged or not, with `git commit -a -m "Some message"`. With this command, you can thus skip the staging area entirely.

While these commands are convenient, you seldom want to do that: chances are, you'd be committing a mixed bag of changes that aren't grouped sensibly.

This creates a messy history that will be hard to navigate in the future (and will be hell for your collaborators).

Let's create more selective snapshots

What you want to do is to create commits that are meaningful.

This is why Git has this 2-step process to make snapshots:

- first you stage
- then you commit

The staging area allows you to pick and choose changes that you want to commit together.

Let's create more selective snapshots

`git add <file>` allows you to only add the changes you made in `<file>` to the staging area (leaving changes to other files unstaged).

Even better, `git add -p <file>` allows you to stage only some of the changes made in `<file>`.

This gives you entire control over your recording of history.

Let's create more selective snapshots

`git add -p <file>` starts an interactive staging session.

For each modified section (called "hunk"), Git will ask you:

```
y    yes (stage this hunk)
n    no (don't stage this hunk)
a    all (stage this hunk and all subsequent ones in this file)
d    do not stage this hunk nor any of the remaining ones
s    split this hunk (if possible)
e    edit
?    print help
```

In [ ]:
```
git status
```

In [ ]:
```
echo "# Effect of Enso on SST in the North Pacific between the years 2001 and 2020

## Introduction

## Methods

## Results

## Conclusion" > ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git add ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -m "Add first draft enso effect ms"
```

In [ ]:
```
git status
```

In [ ]:
```
echo "Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe" >> ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
echo "Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch" >> src/enso_model.py
```

In [ ]:
```
git status
```

In [ ]:
```
git add ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -m "Add Jabberwock 1st paragraph to the enso effect ms"
```

In [ ]:
```
git status
```

In [ ]:
```
emacsclient -c ms/enso_effect.md
```

In [ ]:
```
git status
```

(run from cli): `git add -p ms/enso_effect.md`

In [ ]:
```
git status
```

In [ ]:
```
git add src/enso_model.py
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -m "Edits intro and conclusion ms

First draft intro Jabberwock
Format conclusion and rephrase last paragraph"
```

In [ ]:
```
git status
```

In [ ]:
```
git add .gitignore
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -m "Add .gitignore with result dir"
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Add methods and result ms"
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Add methods and result ms"
```

In [ ]:
```
git status
```

In [ ]:
```
echo "Add content to the ms" >> ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Minor edits enso model ms"
```

In [ ]:
```
echo "Add code to the script" >> src/enso_model.py
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Minor edits script"
```

In [ ]:
```
git status
```

In [ ]:
```
git status
```

Inspecting changes

We saw that `git status` is the key command to get information on the current state of the repo.

While this gives us the list of new files and files with changes, it doesn't allow us to see what those changes are. For this, we need `git diff`.

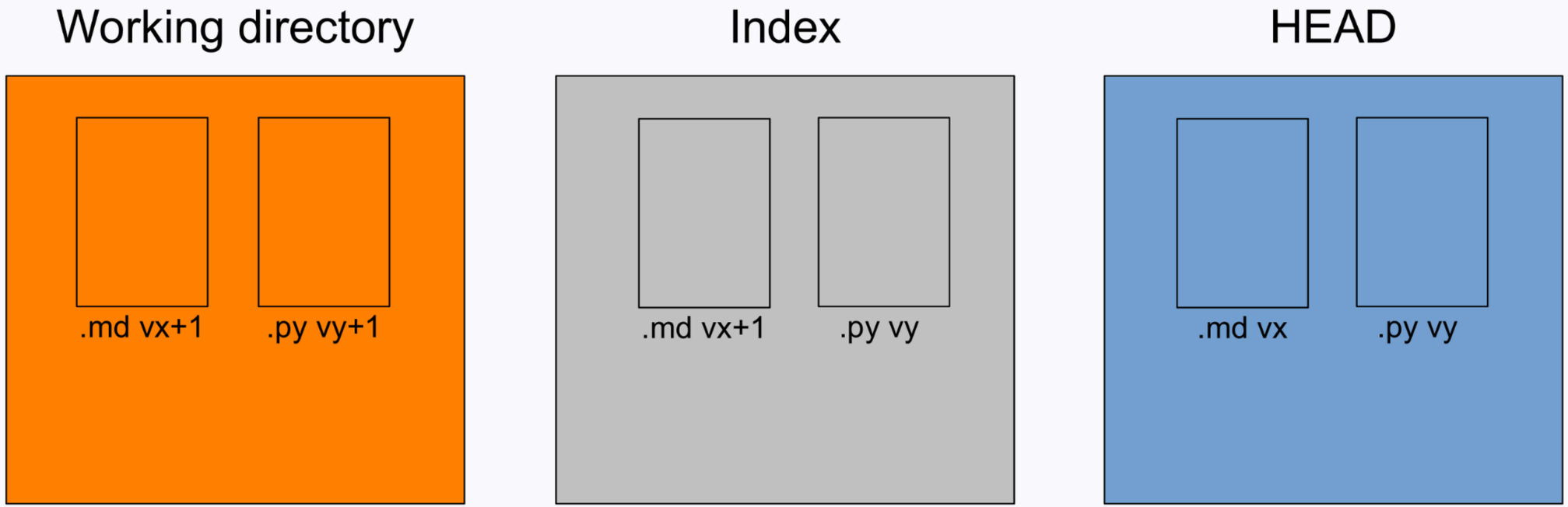
`git diff` shows changes between any two elements (e.g. between commits, between a commit and your working tree, between branches, etc.).

In [ ]:
```
git status
```

In [ ]:
```
echo "Adding some ending to the ms" >> ms/enso_effect.md
```

In [ ]:
```
echo "Adding more code to the script" >> src/enso_model.py
```

In [ ]:
```
git add ms/enso_effect.md
```

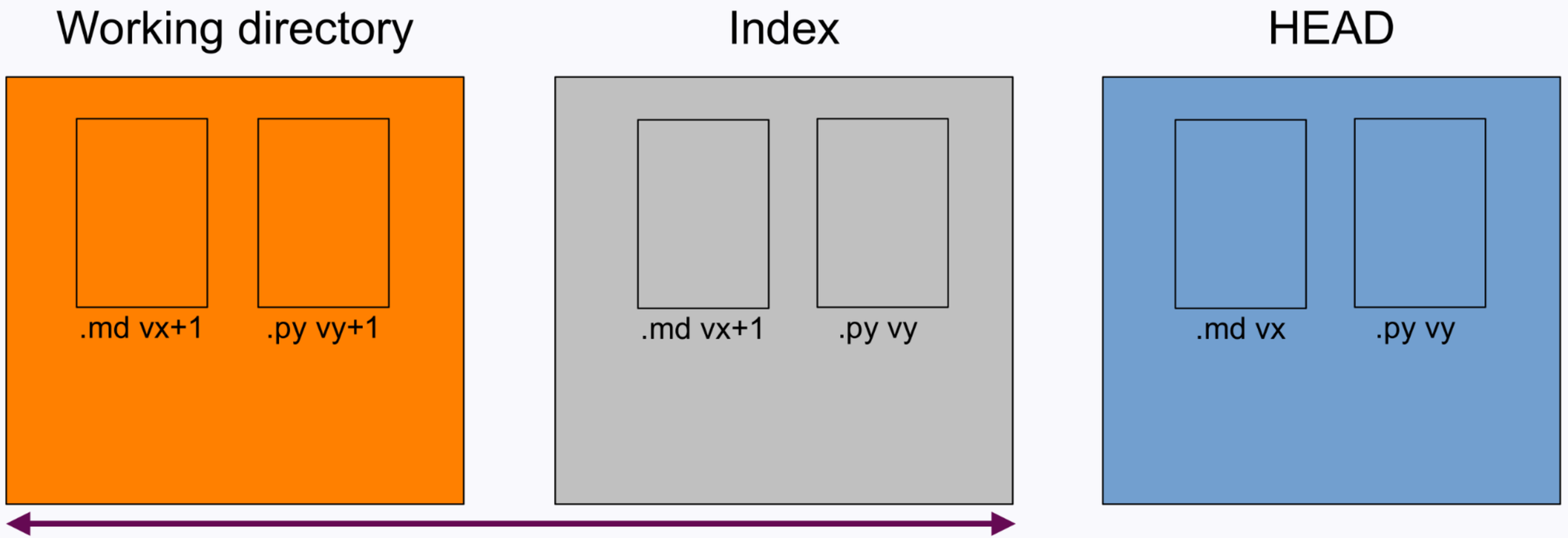

Working directory
.md vx+1
.py vy+1
Index
.md vx+1
.py vy
HEAD
.md vx
.py vy

Inspecting changes

Difference between the working directory and the index

That's all your unstaged changes *on tracked files*.

Git can see new files you haven't staged: it lists them in the output of `git status`. Until you put them under version control by staging them for the first time however, Git has no information about their content: at this point, they are untracked and they are not part of the working tree yet. So their content never appears in the output of `git diff`.


Working directory
Index
HEAD
.md vx+1
.py vy+1
.md vx+1
.py vy
.md vx
.py vy

Recording history

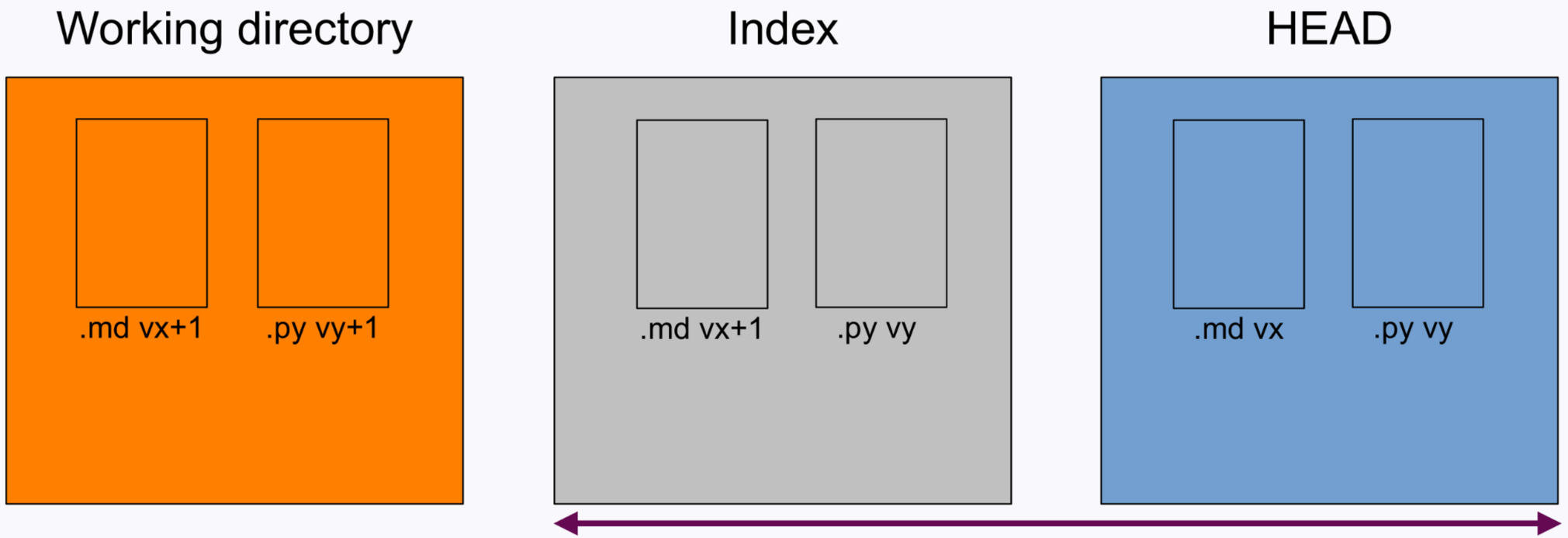
In [ ]:
```
git diff
```

Inspecting changes

Difference between the index and your last commit

That's your staged changes ready to be committed.

That is, that's what would be committed with `git commit -m "Some message"`.


Working directory
Index
HEAD
.md vx+1
.py vy+1
.md vx+1
.py vy
.md vx
.py vy

Recording history

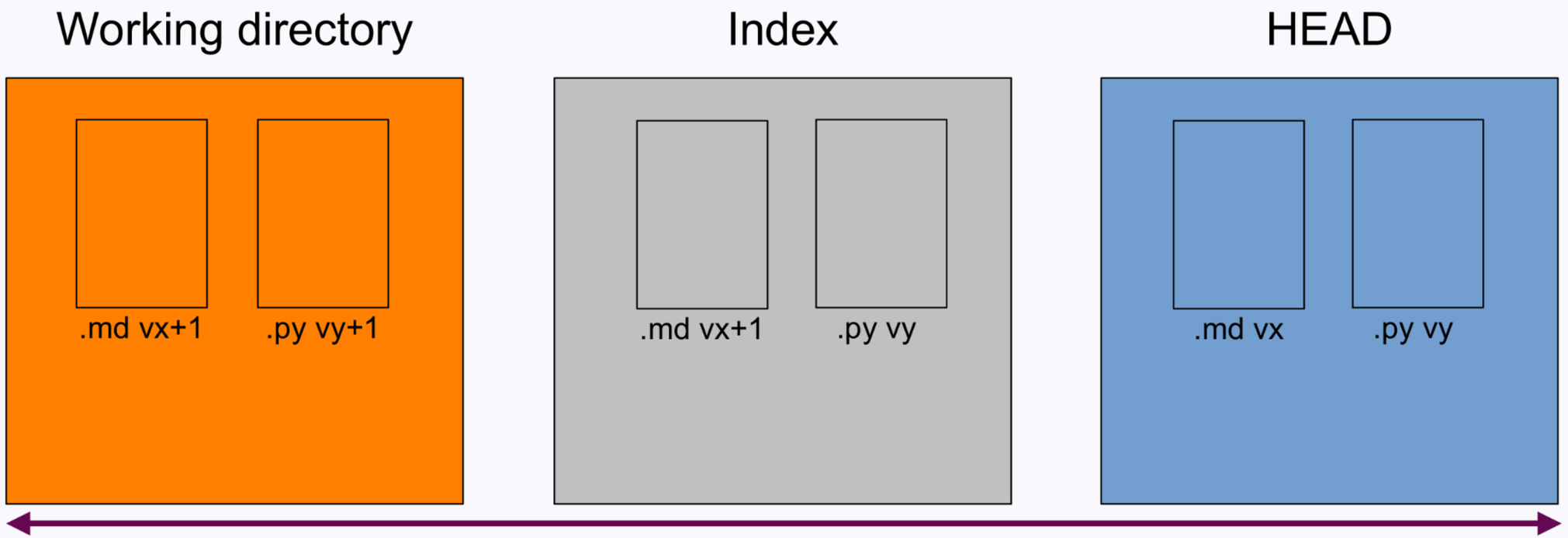
In [ ]:
```
git diff --cached
```

Inspecting changes

Difference between the working directory and your last commit

So both of the above combined.

That's all your staged and unstaged changes (again, only on tracked files).


Working directory
Index
HEAD
.md vx+1
.py vy+1
.md vx+1
.py vy
.md vx
.py vy

Recording history

In [ ]:
```
git diff HEAD
```

In [ ]:
```
echo "Manuscript on long-term acidity change in the Pacific" > ms/acidity.md
```

In [ ]:
```
git status
```

In [ ]:
```
git diff HEAD
```

Inspecting changes

Difference between commits

In [ ]:
```
git diff HEAD~ HEAD
```

In [ ]:
```
git diff HEAD HEAD~
```

In [ ]:
```
git rev-parse HEAD
```

In [ ]:
```
git rev-parse --short HEAD
```

In [ ]:
```
git rev-parse --short HEAD~
```

In [ ]:
```
git diff 265338c 62bfbea
```

Inspecting changes

`git diff` uses a pager (by default, less).

To navigate in the pager:

```
SPACE   scroll one screen down
b       scroll one screen up
q       exit
```

Type `man less` and look at the "COMMANDS" section for more info.

You can circumvent the pager.

In [ ]:
```
git --no-pager diff HEAD
```

Inspecting changes

`git show` shows one object. Applied to a commit, shows the log and changes made at that commit.

In [ ]:
```
git show
```

In [ ]:
```
git show HEAD
```

In [ ]:
```
git show HEAD~
```

In [ ]:
```
git show HEAD~2
```

In [ ]:
```
git show HEAD~2 --oneline
```

Working with branches

Putting aside for a while (stashing)

Before moving HEAD around (amongst branches or in the past), make sure to have a clean working directory.

If you aren't ready to create a commit (messy, unfinished changes, etc.), you can stash those changes and retrieve them later.

In [ ]:
```
git status
```

In [ ]:
```
git stash -u  # -u to include untracked files
```

In [ ]:
```
git status
```

In [ ]:
```
git stash list
```

In [ ]:
```
git stash apply --index  # --index to restage the files that were staged before
```

If you aren't ready to create a commit (messy, unfinished changes, etc.), you can stash those changes and retrieve them later.

In [ ]:
```
git status
```

In [ ]:
```
git stash -u  # -u to include untracked files
```

In [ ]:
```
git status
```

In [ ]:
```
git stash list
```

In [ ]:
```
git stash apply --index  # --index to restage the files that were staged before
```

In [ ]:
```
git stash drop  # delete the stash
```

In [ ]:
```
git stash list
```

In [ ]:
```
git stash -u
```

In [ ]:
```
git status
```

Putting aside for a while (stashing)

A few notes:

- You can apply a stash on a dirty directory
- You can apply a stash on another branch
- You can apply and drop a stash in one command with `git stash pop`, but the `--index` option is not available
- You can have several stashes. In that case, Git always assumes you want to perform an action on the last one. If that is not the case, you have to provide the name of the stash you want to use (e.g. `git stash apply stash@{1}`. You can find that name with `git stash list`).

Creating branches

"master"

When you initialized your repository with `git init`, a branch got created. It is called master (you could rename it to something else if you wanted—that initial branch, despite its name, has nothing special).

So as soon as you start working on your project, there is one branch (master) and you are on it.

Creating branches

Additional branches

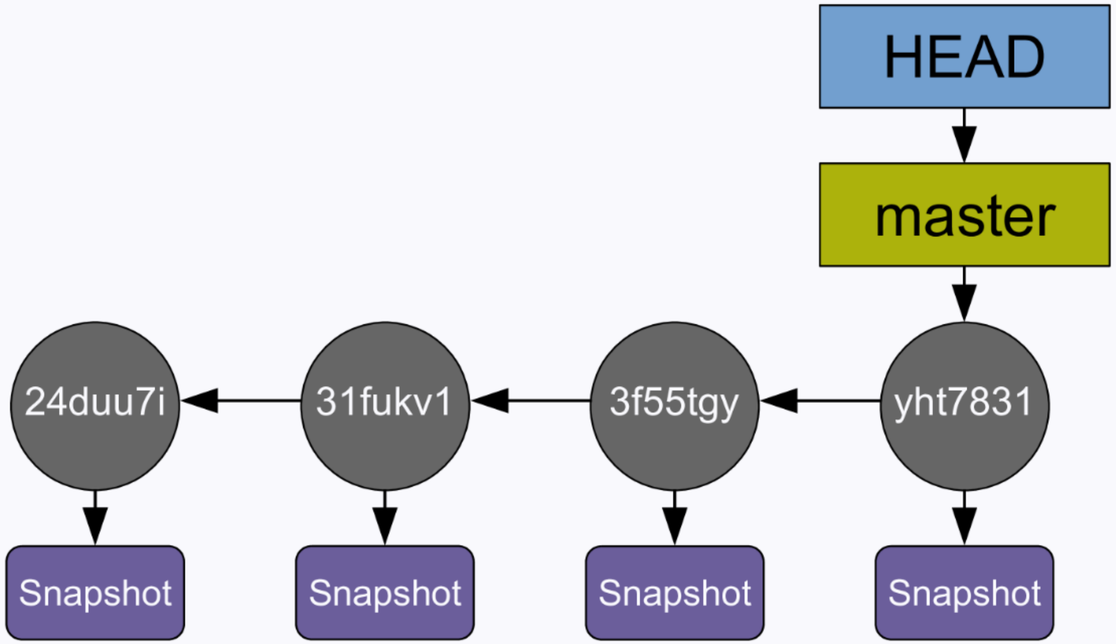
You can create additional branches with `git branch <branch-name>`.

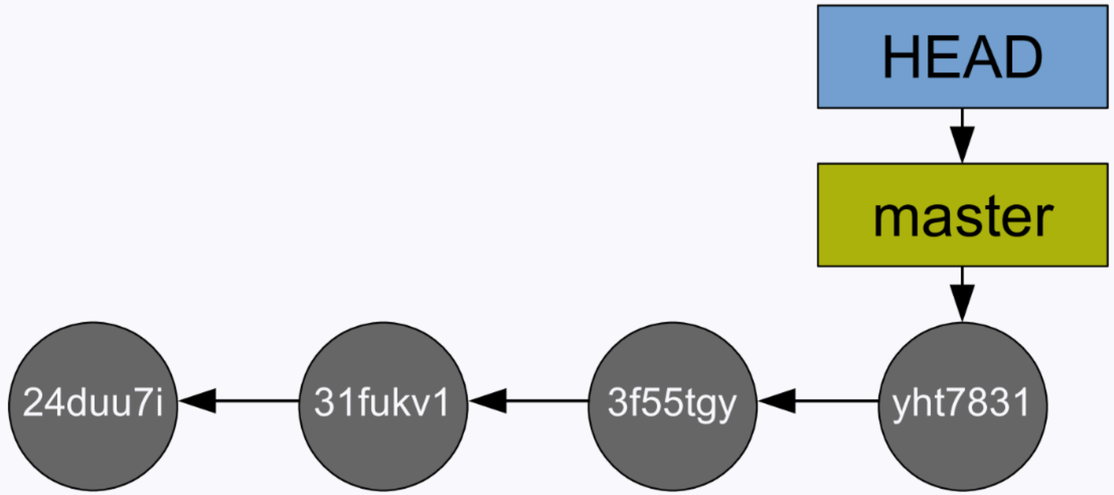
In [ ]:
```
git branch
```

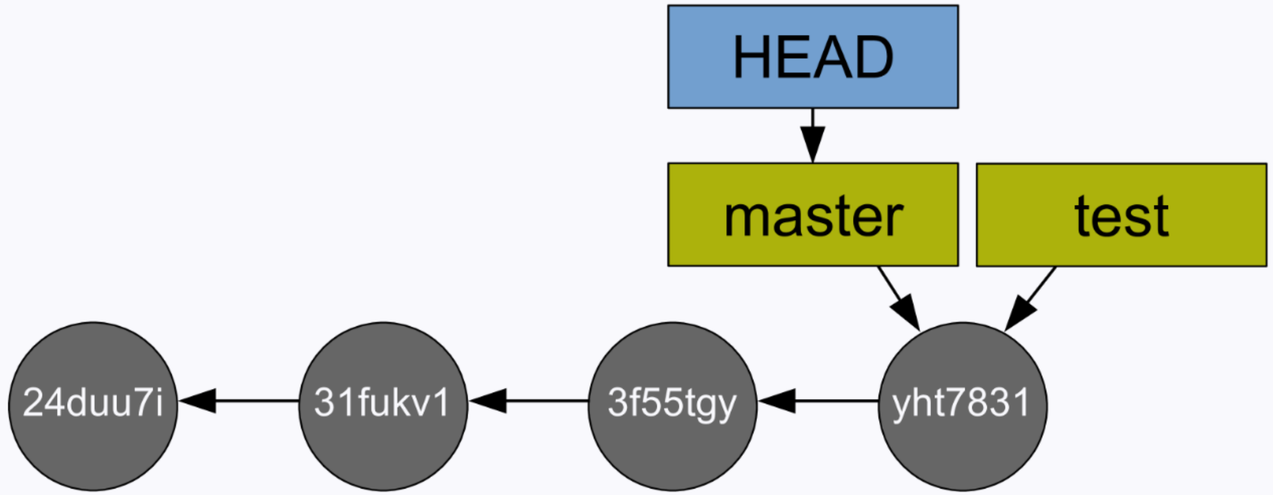
In [ ]:
```
git branch test
```

In [ ]:
```
git status
```

In [ ]:
```
git branch
```


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
Snapshot
Snapshot
Snapshot
Snapshot


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831

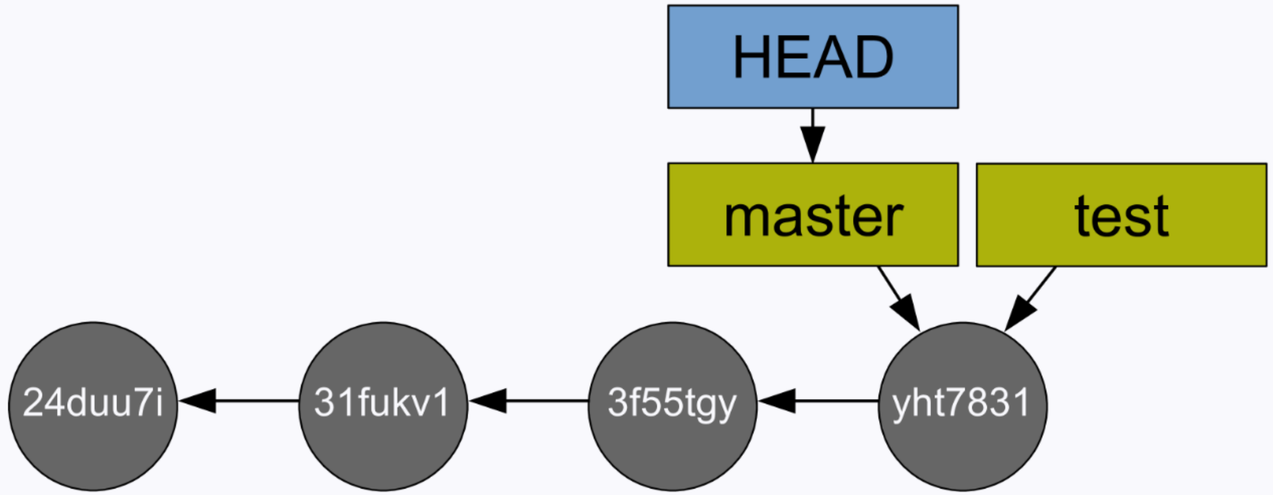
Switching branch

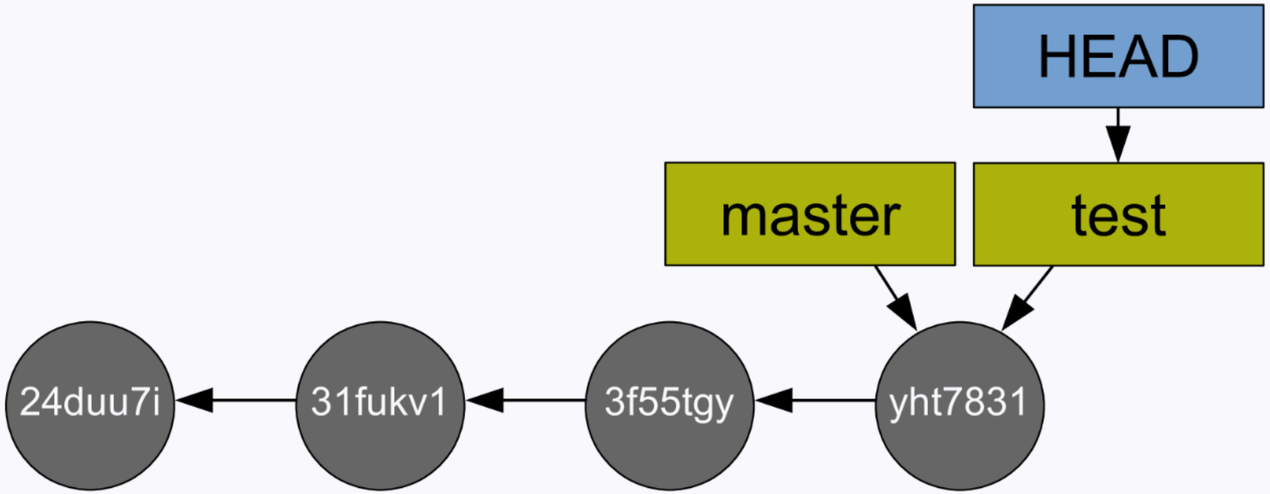
To switch branch, you use `git checkout <branch-name>`.

In [ ]:
```
git checkout test
```

In [ ]:
```
git status
```

In [ ]:
```
git branch
```


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831

Creating a branch & switching to it immediately

When you create a branch, most of the time you want to switch to it. So there is a command which allows to create a branch and switch to it immediately without having to do this in two steps: `git checkout -b <branch-name>`.

This command is convenient: when you create a branch with `git branch <branch-name>`, it is very easy to forget to switch to the new branch before making commits!

In [ ]:
```
git checkout -b dev
```

In [ ]:
```
git status
```

In [ ]:
```
git branch
```

Creating commits on the new branch

In [ ]:
```
git checkout test
```

In [ ]:
```
touch src/acidity.py
```

In [ ]:
```
git status
```

In [ ]:
```
git add src/acidity.py
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -m "Add new acidity script"
```

In [ ]:
```
git status
```

In [ ]:
```
echo "Some content" >> src/acidity.py
```

In [ ]:
```
git status
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Add some content to acidity script"
```

In [ ]:
```
git status
```

In [ ]:
```
ls src/
```

In [ ]:
```
tree
```

In [ ]:
```
git checkout master
```

In [ ]:
```
git status
```

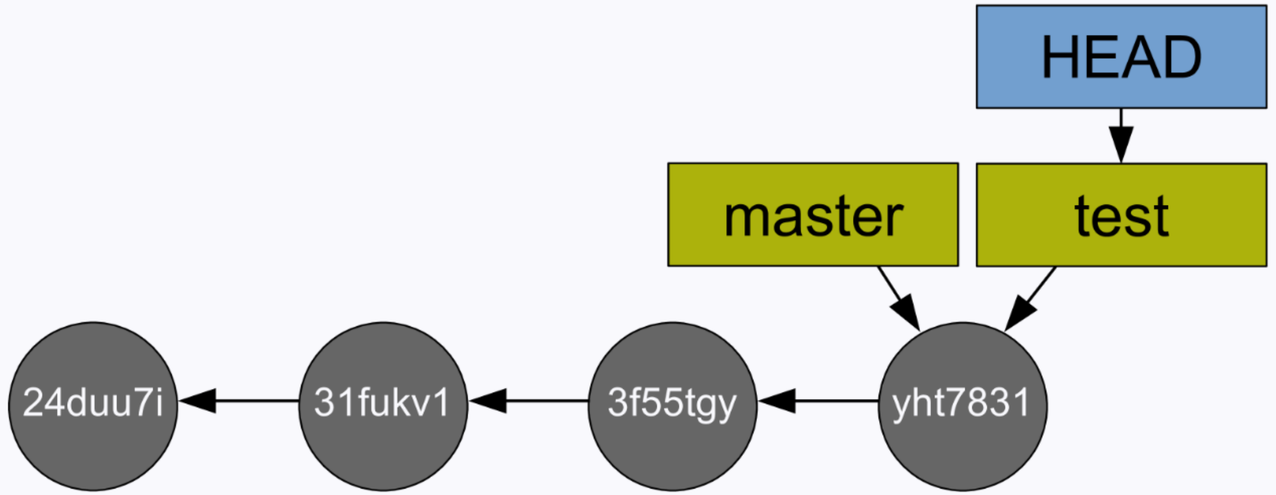
In [ ]:
```
ls src/
```

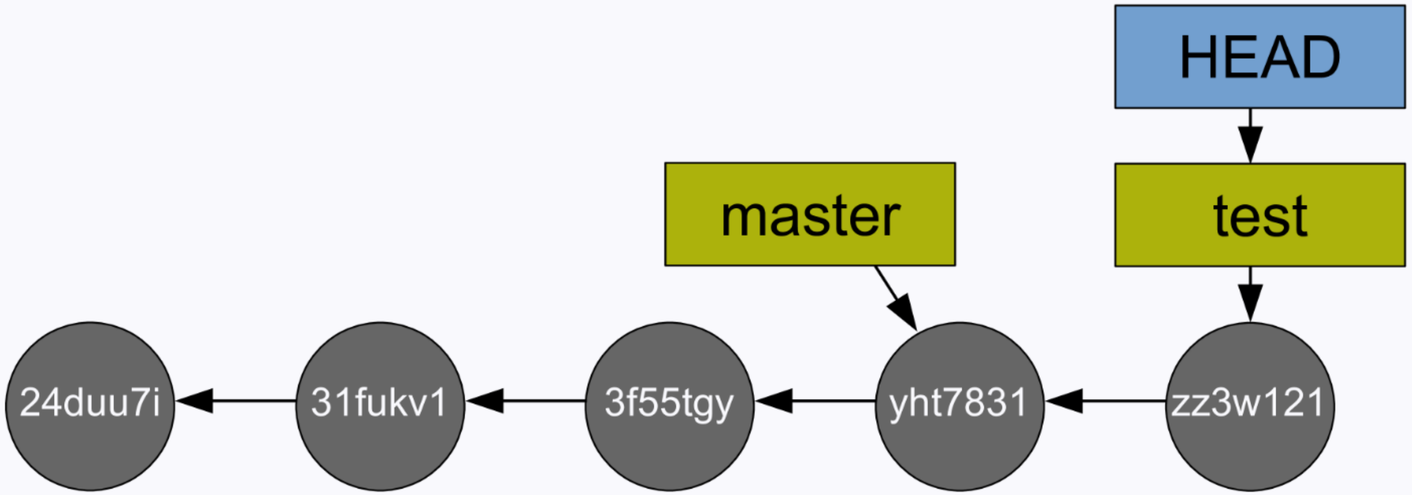
In [ ]:
```
tree
```

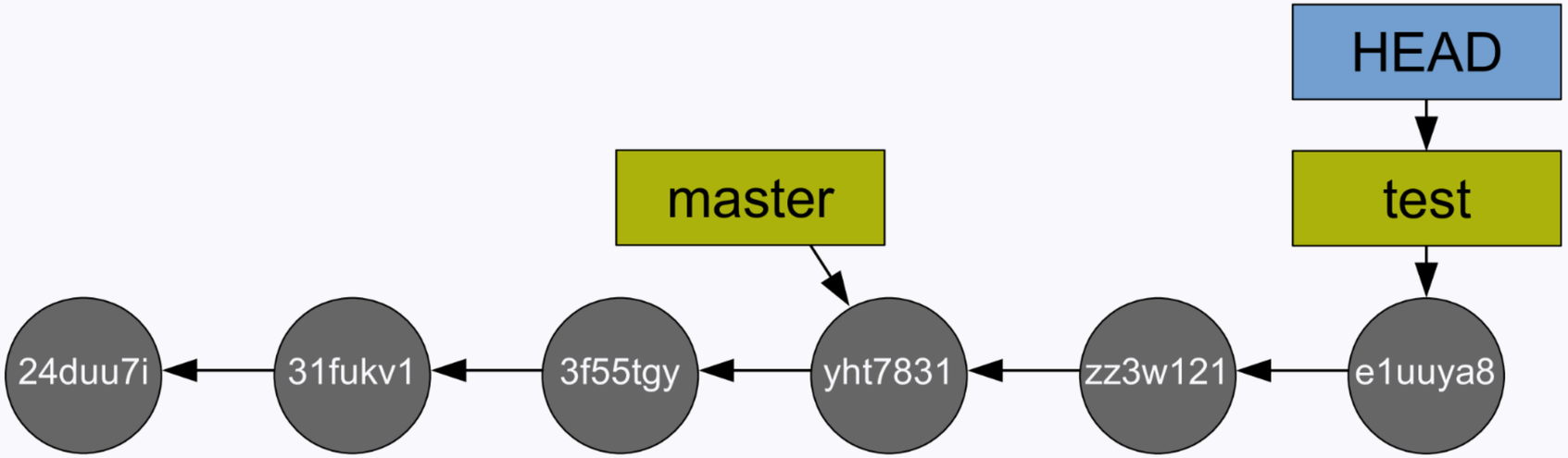
In [ ]:
```
git checkout test
```

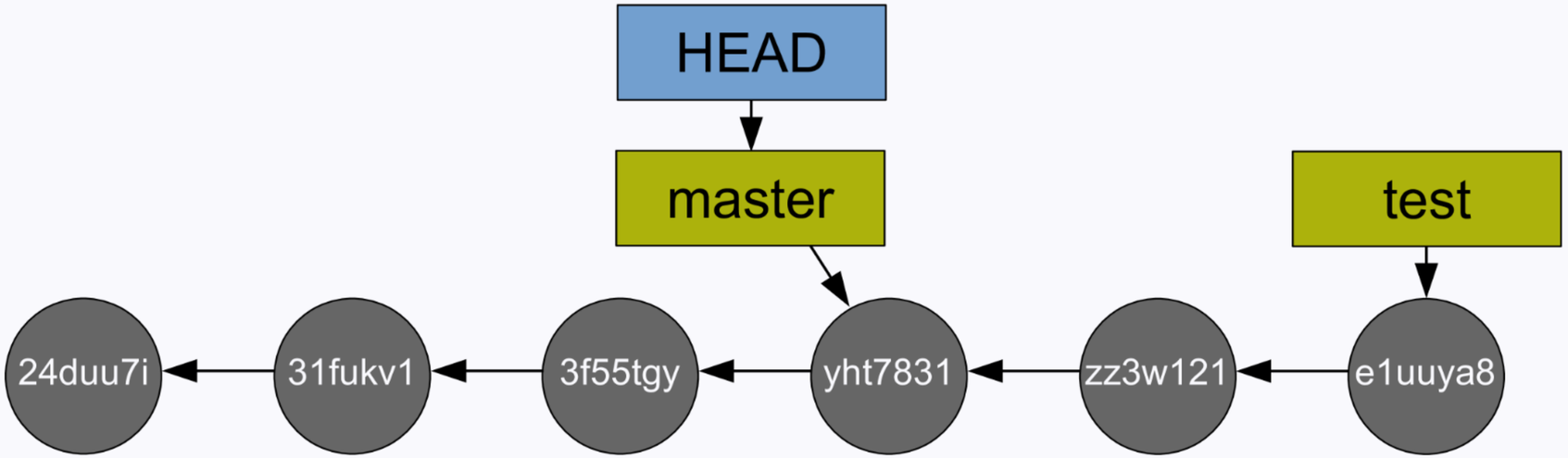
In [ ]:
```
ls src/
```

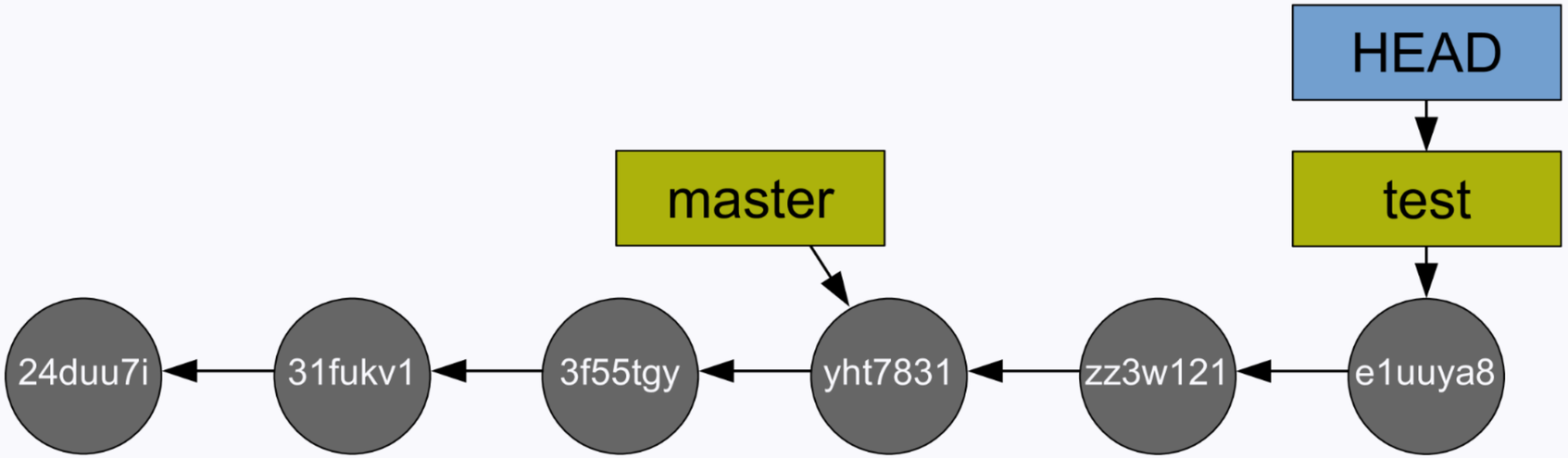
In [ ]:
```
tree
```


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8

Comparing branches

In [ ]:
```
git diff test master
```

In [ ]:
```
git diff master test
```

In [ ]:
```
git diff dev master
```

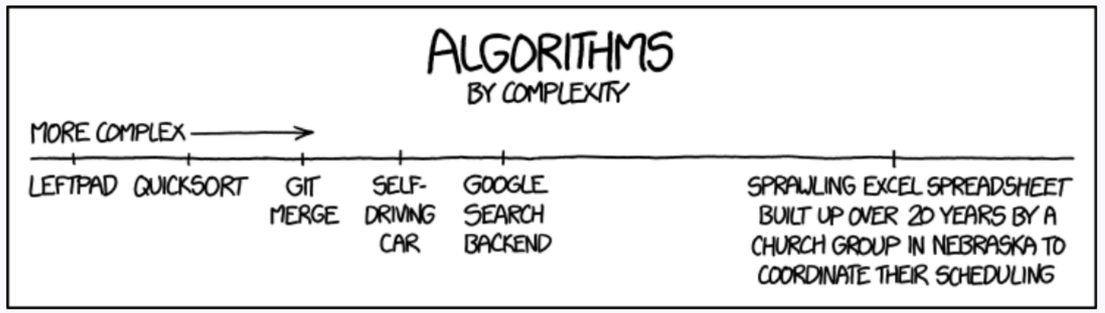
Merging branches

One thing that makes Git branches powerful is—as we just saw—how easy it is to create new branches and to switch from one branch to another. Another thing is how easy it is to merge branches together.

If you created an experimental branch and are happy with the result, you'll want to merge it into your main branch.

First, switch to the main development branch, then merge your experimental branch into the main branch:

```
git merge <branch-to-merge-into-current-branch>
```



from xkcd.com

Merging branches

Fast-forward merge

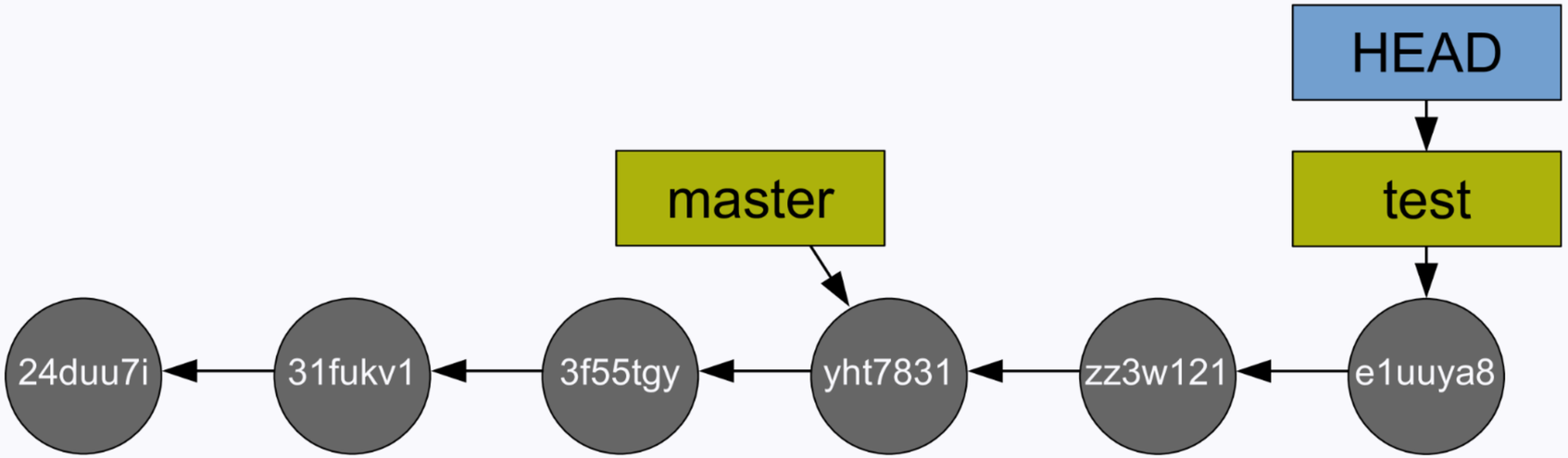
In [ ]:
```
git branch
```

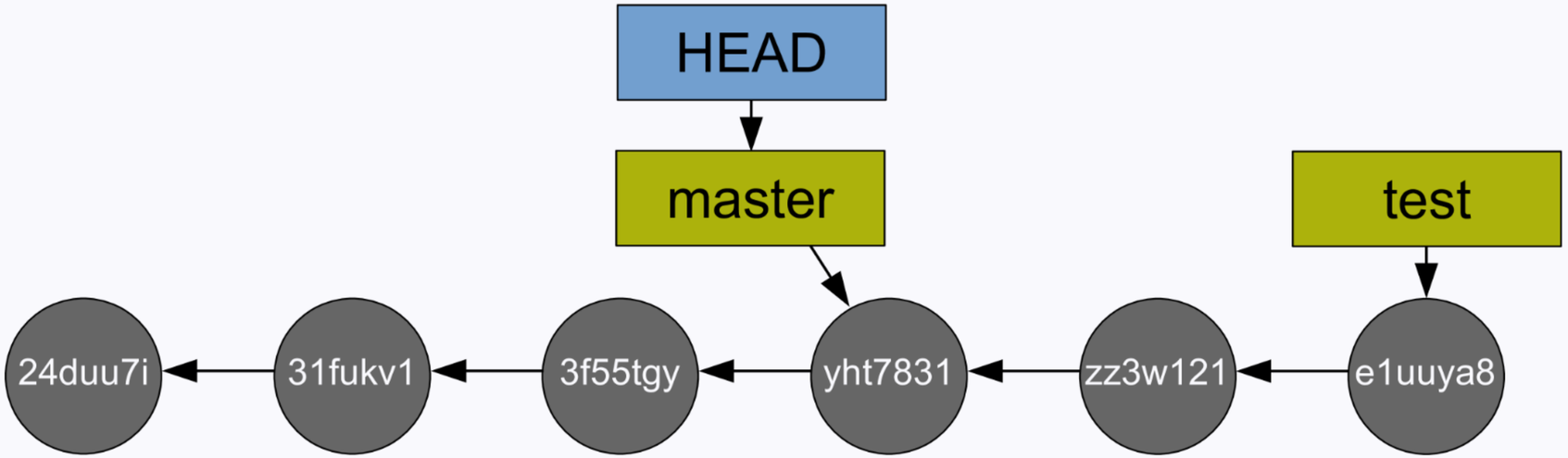
In [ ]:
```
git checkout master
```

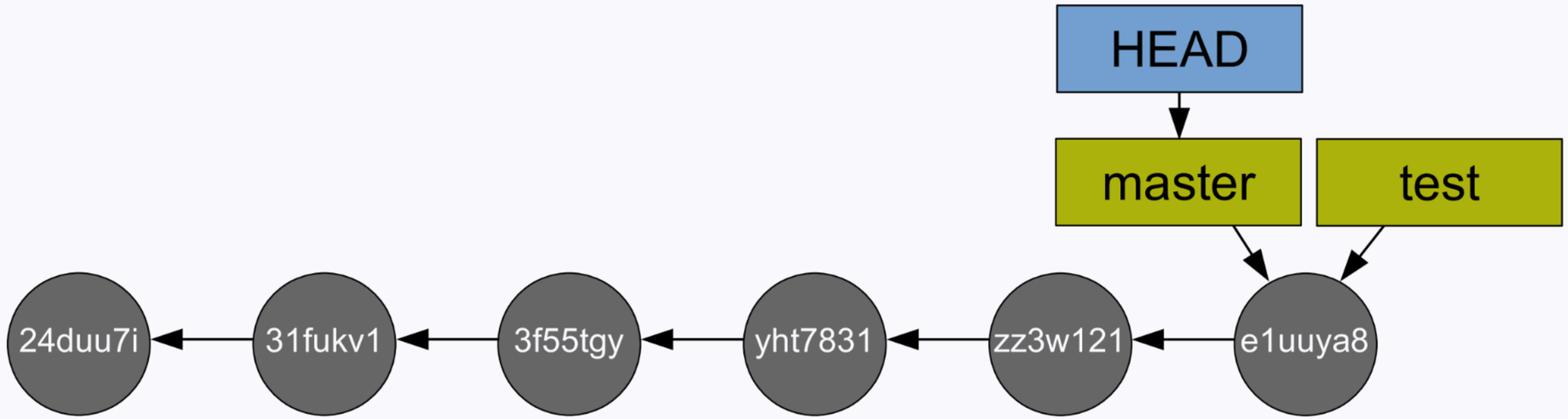
In [ ]:
```
git status
```

In [ ]:
```
git merge test
```

In [ ]:
```
git status
```


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8

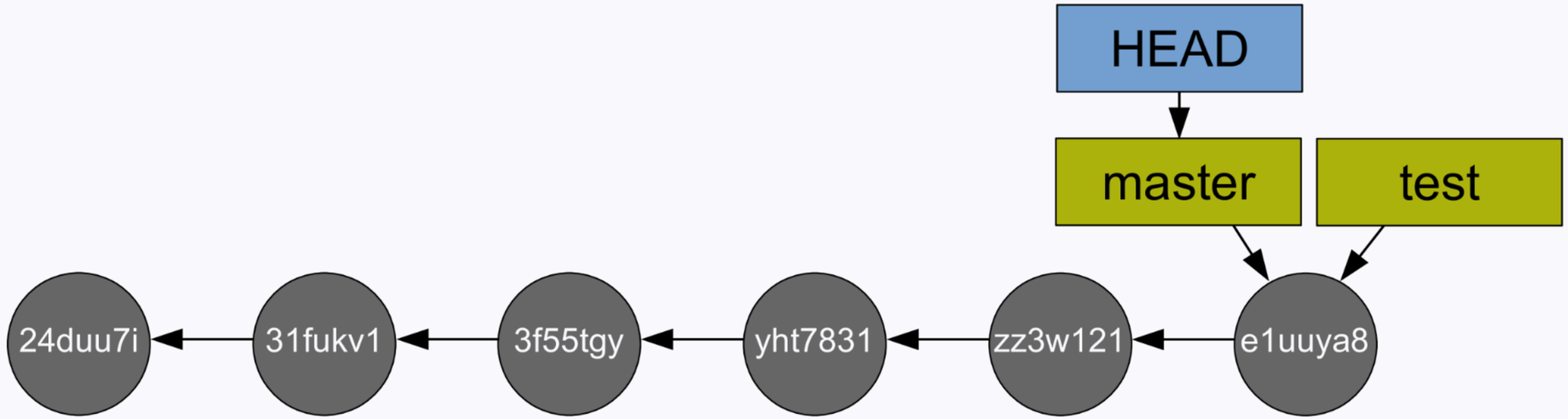

HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8

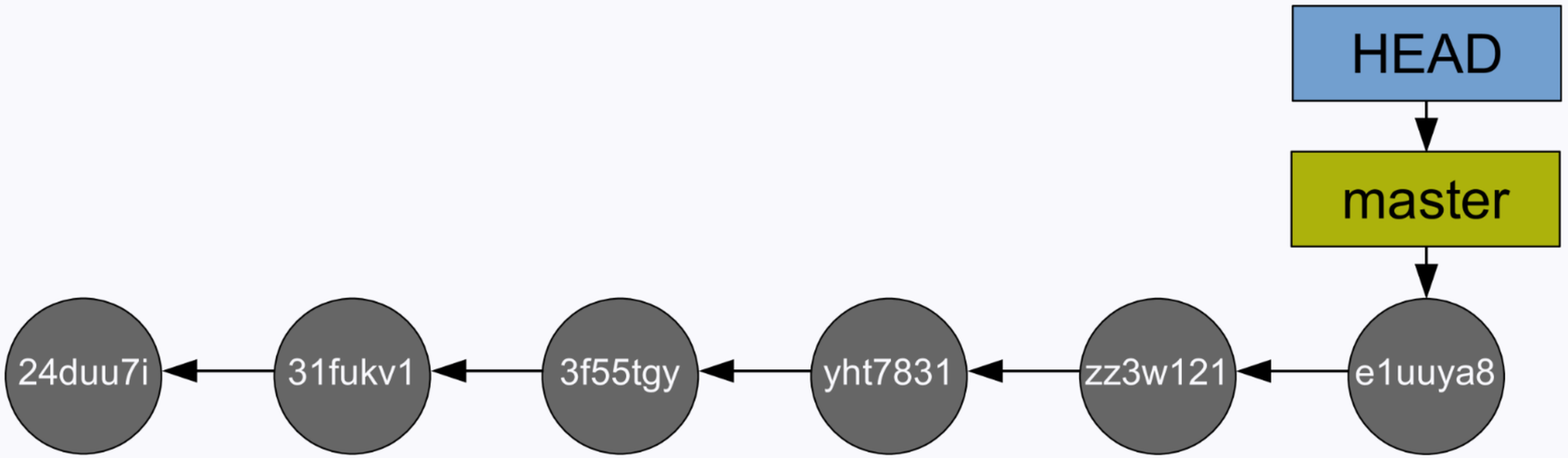

HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8

Deleting branches

Once you have merged a branch into another or if you decide that the experiments on a branch are not worth keeping, you can delete that branch.

To do so, we could run `git branch -d test`, but we will keep it for now as it will be useful later on.


HEAD
master
test
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
zz3w121
e1uuya8

Merging branches

Merge commit

In [ ]:
```
git branch test2
```

In [ ]:
```
git checkout test2
```

In [ ]:
```
echo "Some edits to the enso ms" >> ms/enso_effect.md
```

In [ ]:
```
git commit -a -m "Edit enso ms"
```

In [ ]:
```
git checkout master
```

In [ ]:
```
echo "Add some code to the script" >> src/enso_model.py
```

In [ ]:
```
git commit -a -m "Add code enso script"
```

Merge commit

In [ ]:
```
git branch test2
```

In [ ]:
```
git checkout test2
```

In [ ]:
```
echo "Some edits to the enso ms" >> ms/enso_effect.md
```

In [ ]:
```
git commit -a -m "Edit enso ms"
```

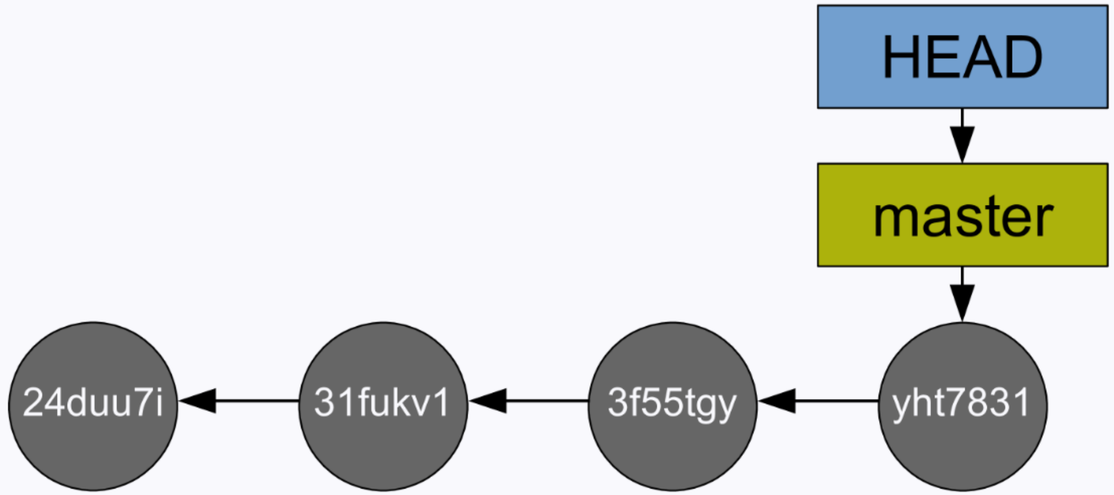
In [ ]:
```
git checkout master
```

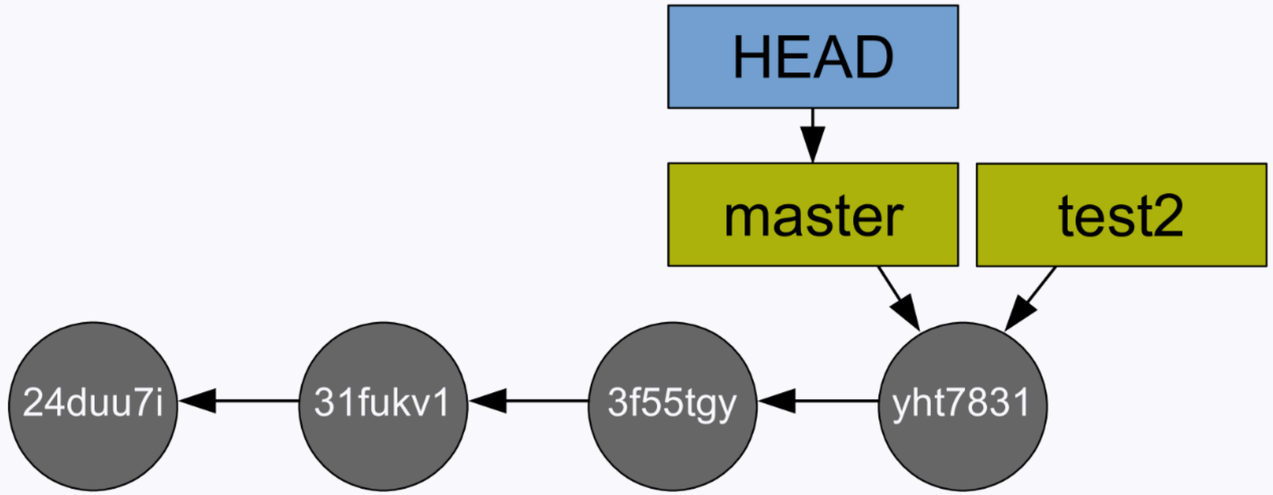
In [ ]:
```
echo "Add some code to the script" >> src/enso_model.py
```

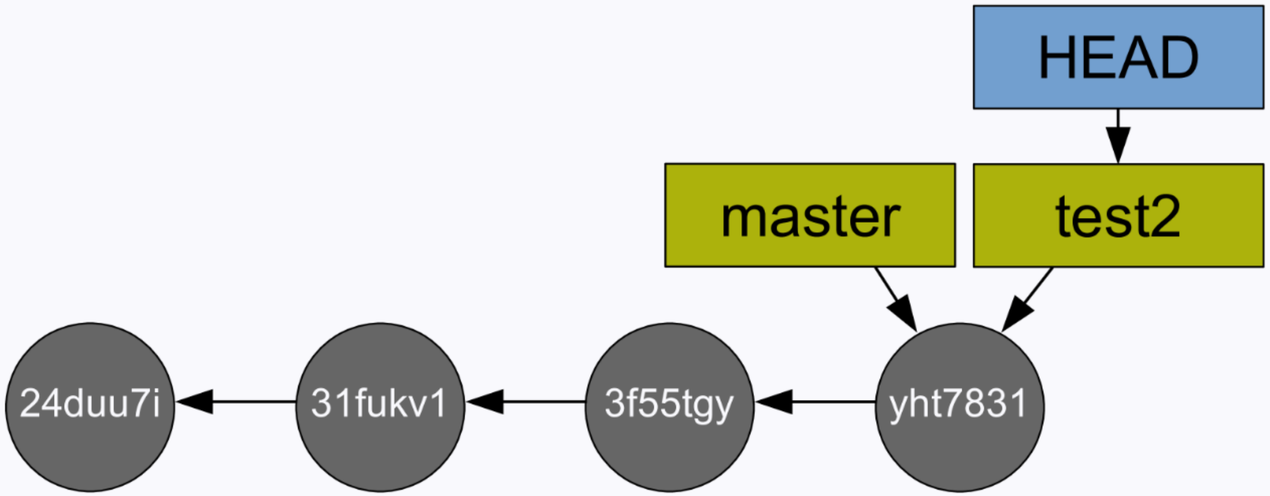
In [ ]:
```
git commit -a -m "Add code enso script"
```

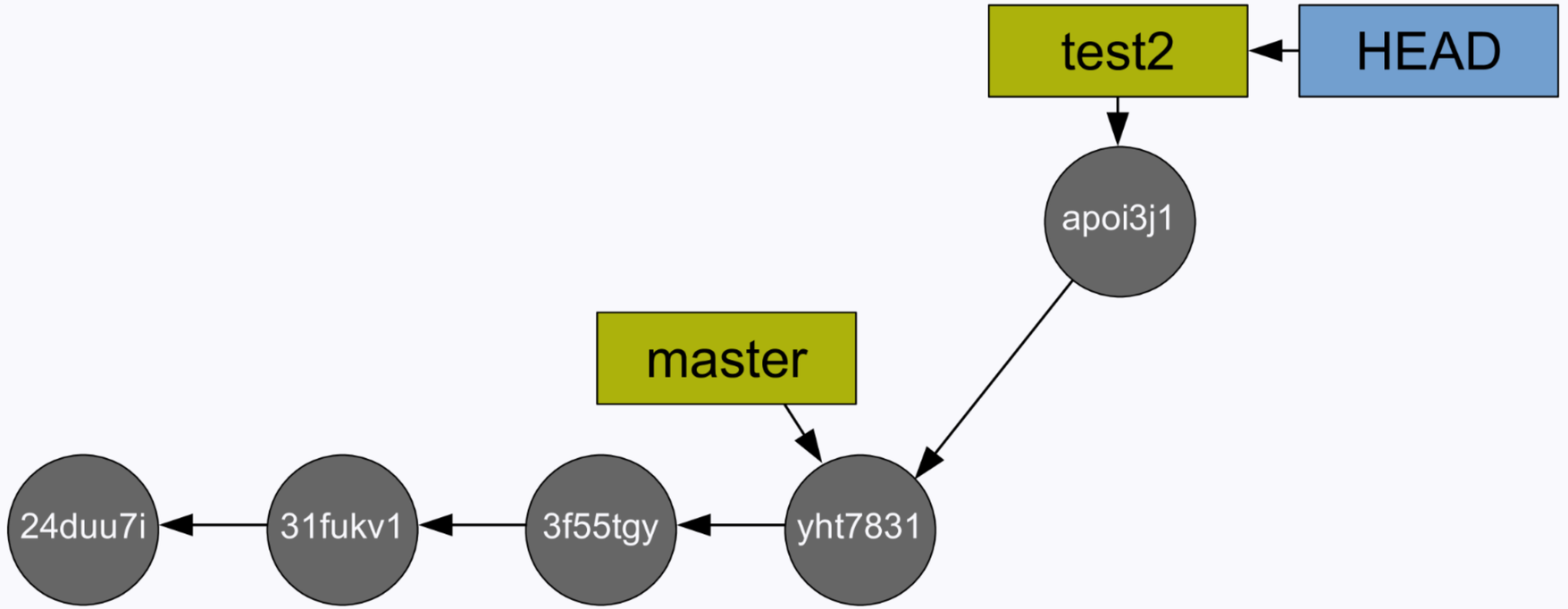
In [ ]:
```
git merge test2
```

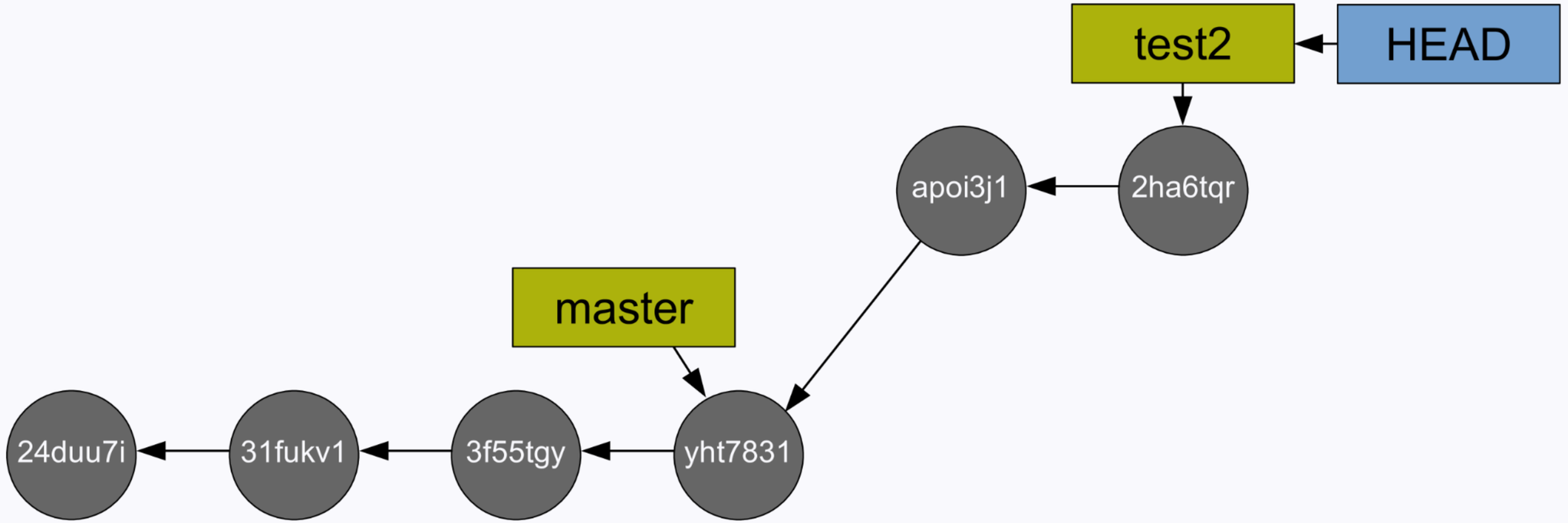
In [ ]:
```
# git branch -d test2 (not run because I will use it later)
```

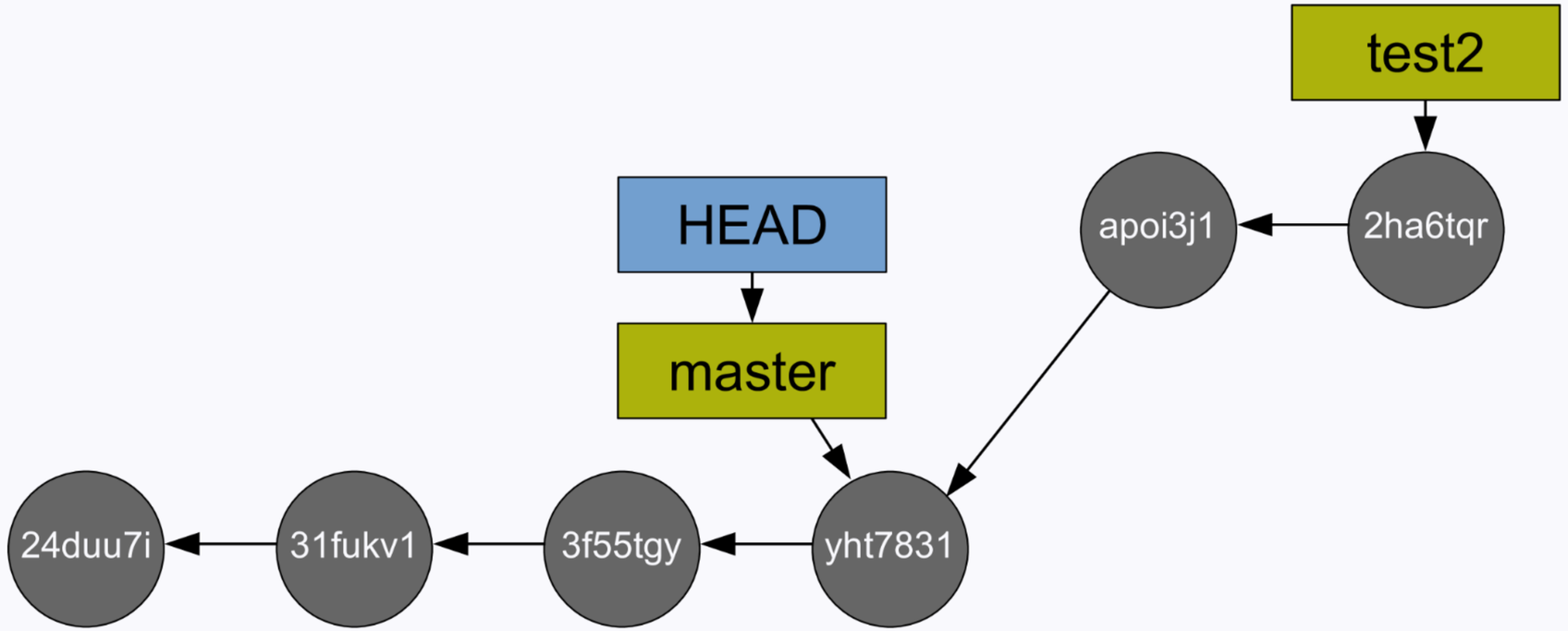

HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831

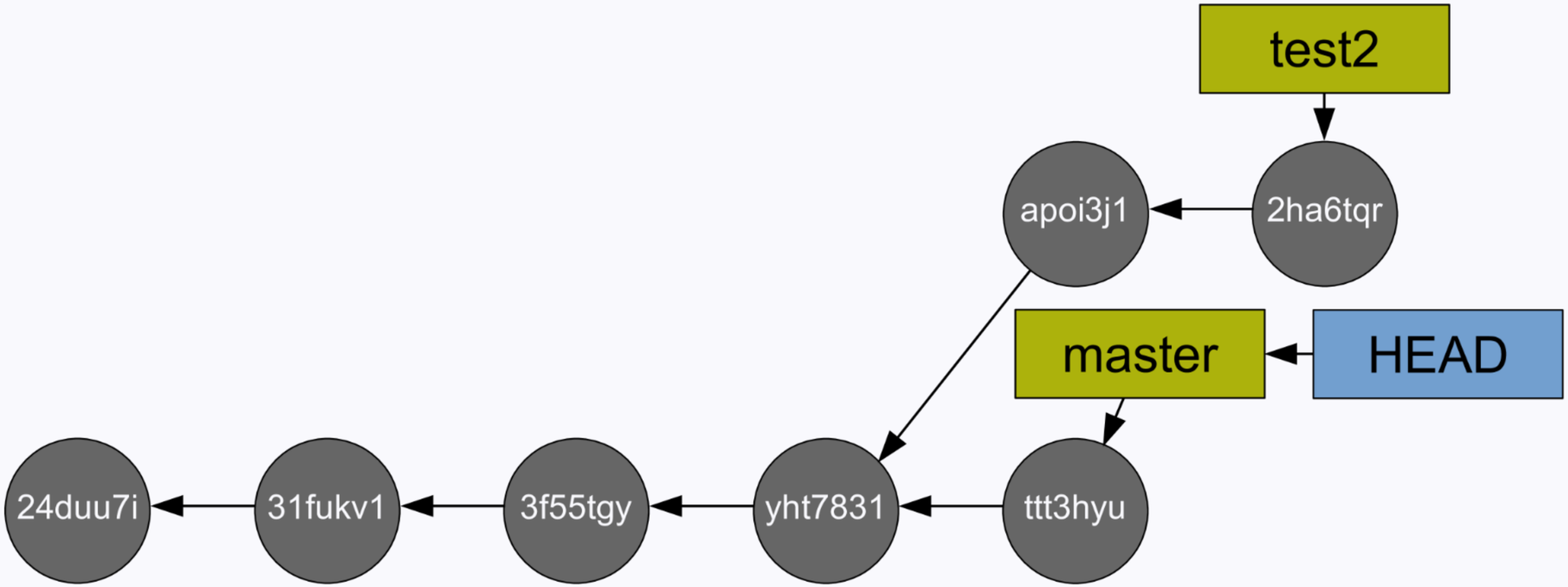

HEAD
master
test2
24duu7i
31fukv1
3f55tgy
yht7831

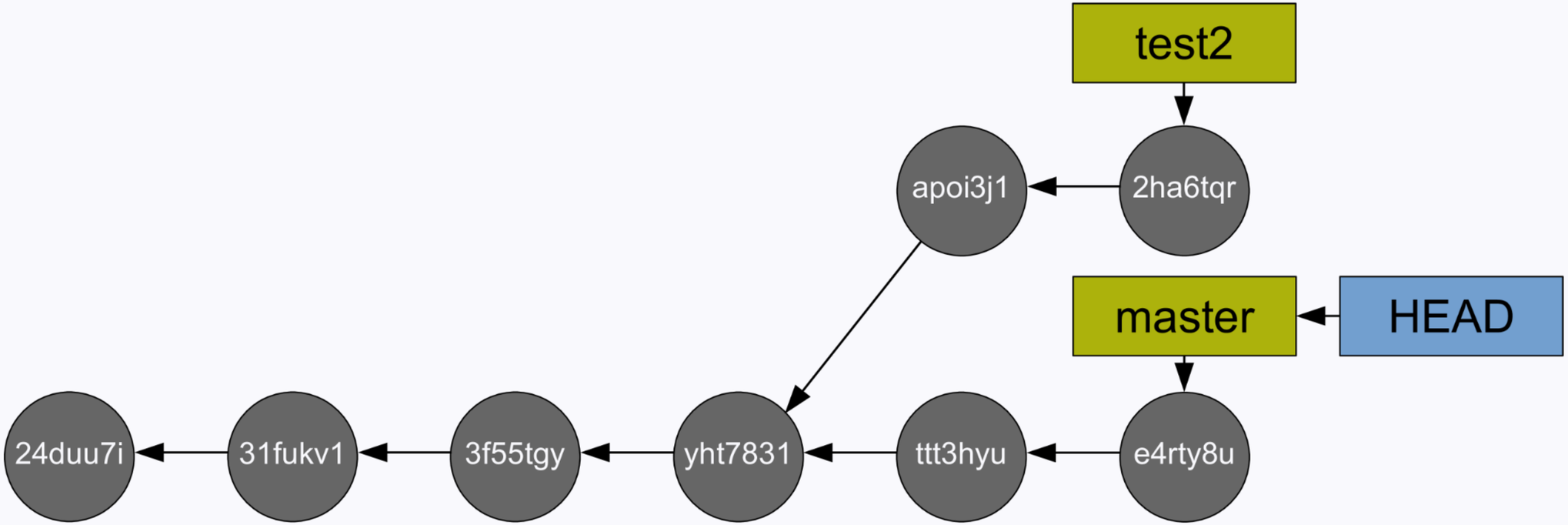

HEAD
master
test2
24duu7i
31fukv1
3f55tgy
yht7831

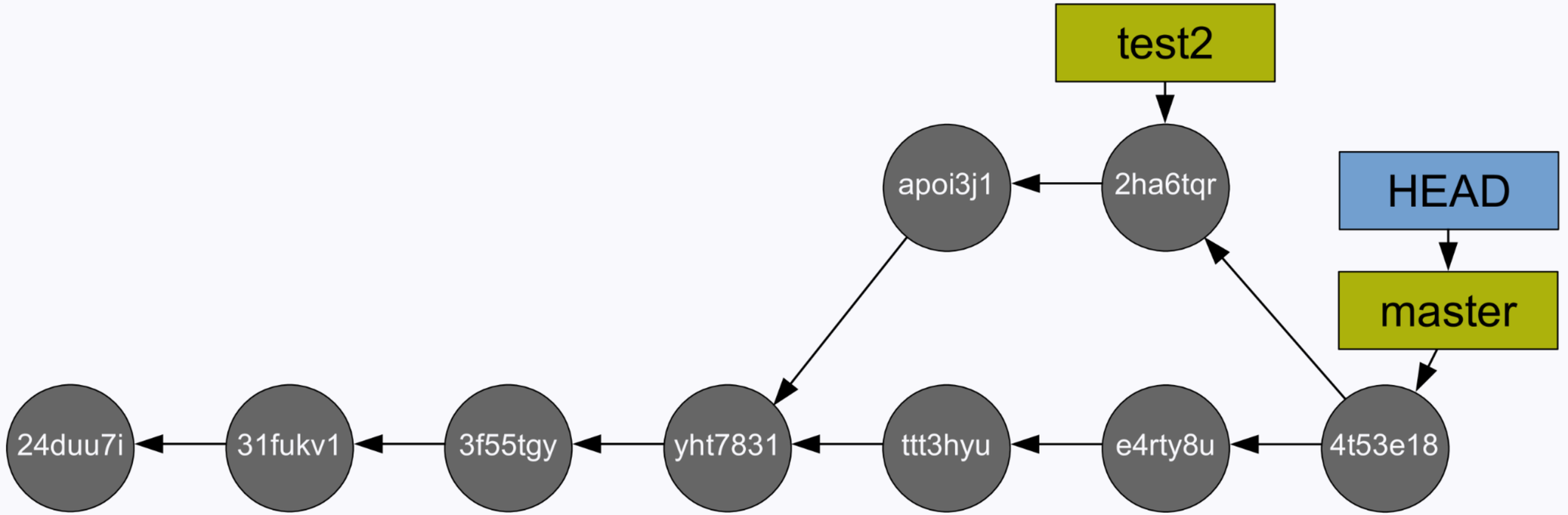

test2
HEAD
apoi3j1
master
24duu7i
31fukv1
3f55tgy
yht7831

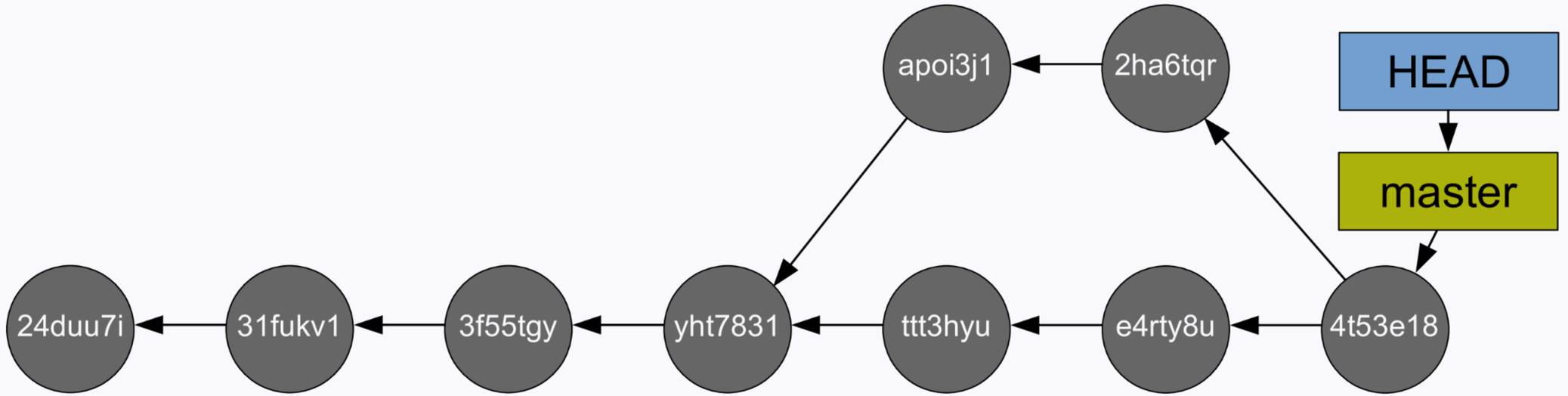

test2
HEAD
apoi3j1
2ha6tqr
master
24duu7i
31fukv1
3f55tgy
yht7831


test2
HEAD
apoi3j1
2ha6tqr
master
24duu7i
31fukv1
3f55tgy
yht7831


test2
apoi3j1
2ha6tqr
master
HEAD
24duu7i
31fukv1
3f55tgy
yht7831
ttt3hyu


test2
apoi3j1
2ha6tqr
master
HEAD
24duu7i
31fukv1
3f55tgy
yht7831
ttt3hyu
e4rty8u


test2
apoi3j1
2ha6tqr
HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
ttt3hyu
e4rty8u
4t53e18


apoi3j1
2ha6tqr
HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
ttt3hyu
e4rty8u
4t53e18

Merge conflicts

As you were developing your experimental branch, maybe you were also developing your main branch. As long as the differences between the branches do not overlap (you have been working on different parts of the project in each branch, which can include different parts of the same file), there is no problem.

If the two branches contain different versions of the same part of a file however, Git cannot know which of the versions you want to keep. The merge will then be interrupted and Git will ask you to resolve the conflict(s) before the merge can be completed.

Conflicts will look like this:

```
<<<<<<< HEAD
Version of this section of the file on your checkedout branch
=======
Alternative version of the same section of the file
>>>>>>> alternative version
```

Merge conflicts

In [ ]:
```
git checkout -b test3
```

In [ ]:
```
emacsclient -c ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git add ms/enso_effect.md
```

In [ ]:
```
git commit -m "Make some edits enso ms"
```

In [ ]:
```
git status
```

In [ ]:
```
git checkout master
```

In [ ]:
```
emacsclient -c ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -m "Make some edits enso ms"
```

In [ ]:
```
git status
```

In [ ]:
```
git checkout master
```

In [ ]:
```
emacsclient -c ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git add ms/enso_effect.md
```

In [ ]:
```
git commit -m "Make conflicting edits enso ms"
```

In [ ]:
```
git status
```

In [ ]:
```
git checkout master
```

In [ ]:
```
git merge test3
```

In [ ]:
```
git status
```

Resolving conflicts

Merge tools allow you to jump from conflict to conflict within a file and ask you to decide which version you want to choose for each of them (you can also write a combination of the two).

In [ ]:
```
git mergetool
```

In [ ]:
```
git mergetool --tool-help
```

Resolving conflicts

If you don't use any merge tool, you can edit those sections manually in any text editor.

You can also in one swoop keep *our* version (i.e. the version of the branch you are currently on or HEAD) or all of *their* version (the alternative version of the file you are merging into your branch) for all of the sections.

```
git checkout --ours <file>
git checkout --theirs <file>
```

In [ ]:
```
emacsclient -c ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git add ms/enso_effect.md
```

In [ ]:
```
git commit
```

Exploring the past

Overview of commit history

`git log` shows the log of commits.

In its simplest form, it gives a list of past commits in a pager.

In [ ]:
```
git log
```

Overview of commit history

This log can be customized greatly by playing with the various flags.

In [ ]:
```
git log --oneline
```

In [ ]:
```
man git-log
```

Overview of commit history

You can make it really clean and fancy:

```
git log \
    --graph \
    --date-order \
    --date=short \
    --pretty=format:'%C(cyan)%h %C(blue)%ar %C(auto)%d'`
                   `'%C(yellow)%s%+b %C(magenta)%ae'
```

Overview of commit history

Or you can make it as a graph.

In [ ]:
```
git log --graph
```

In [ ]:
```
git log --graph --all
```

Revisiting old commits

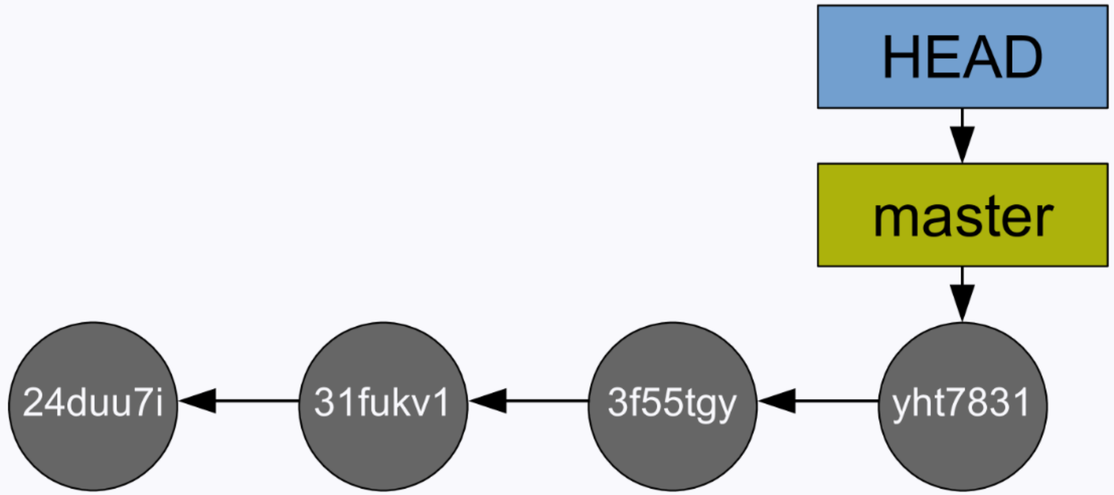
```
git checkout <commit-hash>
```

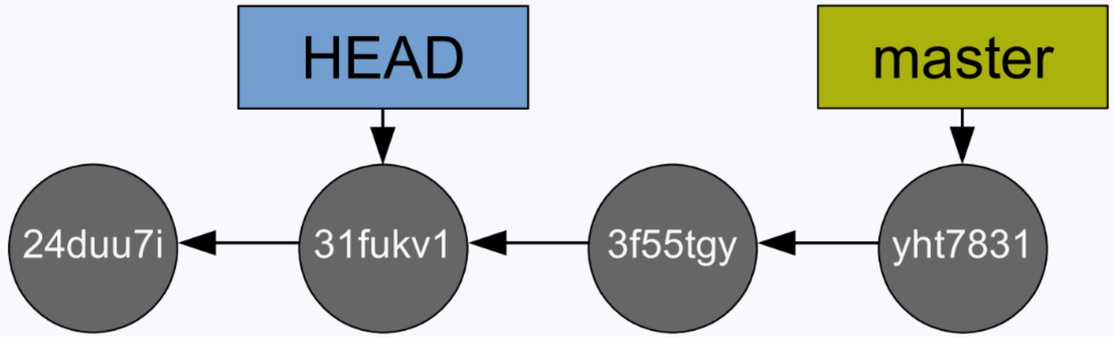
You can also use tags:

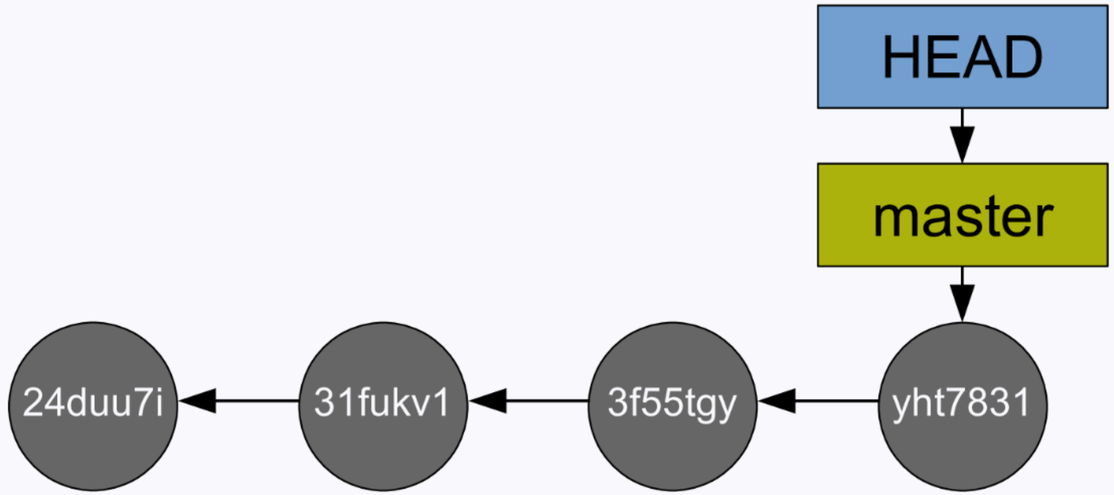
```
git checkout <tag-name>
```

In [ ]:
```
git checkout xxxx
```

In [ ]:
```
git checkout master
```


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831

Detached HEAD

Dangerous workflow

In [ ]:
```
git checkout xxxx
```

In [ ]:
```
echo "lala" >> ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Exploration from commit xxx"
```

In [ ]:
```
git status
```

In [ ]:
```
echo "tutut" >> ms/enso_effect.md
```

In [ ]:
```
git commit -a -m "Another commit on that branch"
```

Dangerous workflow

In [ ]:
```
git checkout xxxx
```

In [ ]:
```
echo "lala" >> ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Exploration from commit xxx"
```

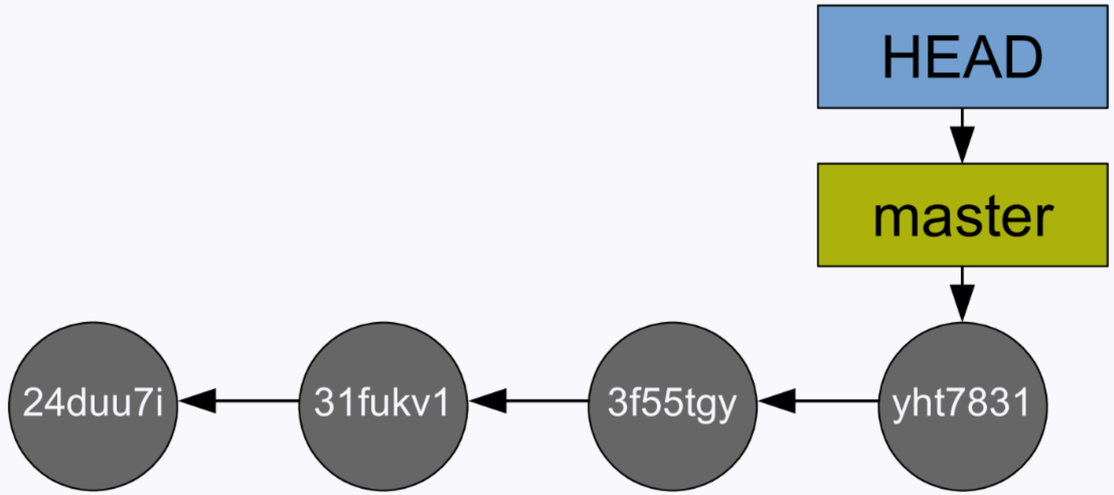
In [ ]:
```
git status
```

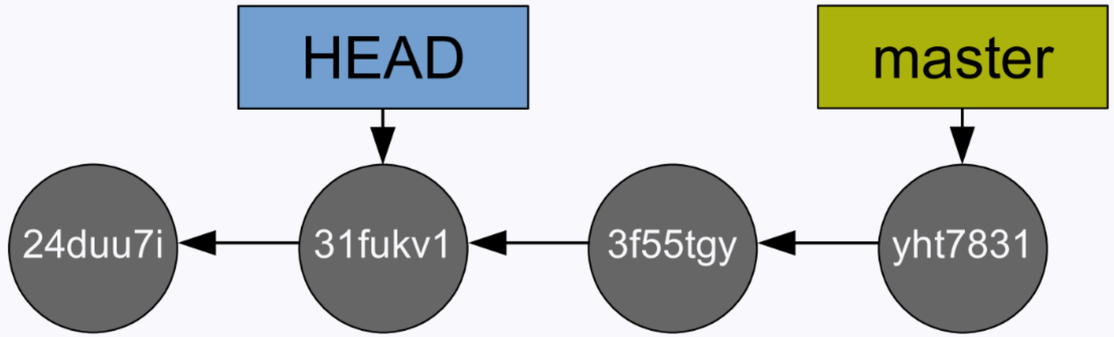
In [ ]:
```
echo "tutut" >> ms/enso_effect.md
```

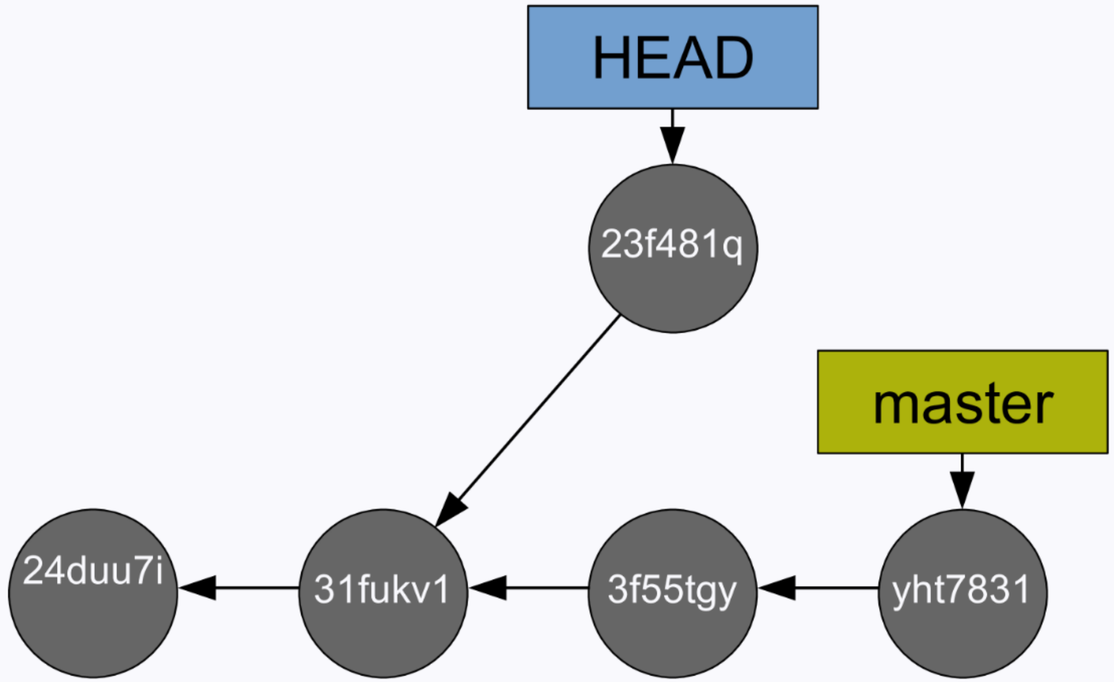
In [ ]:
```
git commit -a -m "Another commit on that branch"
```

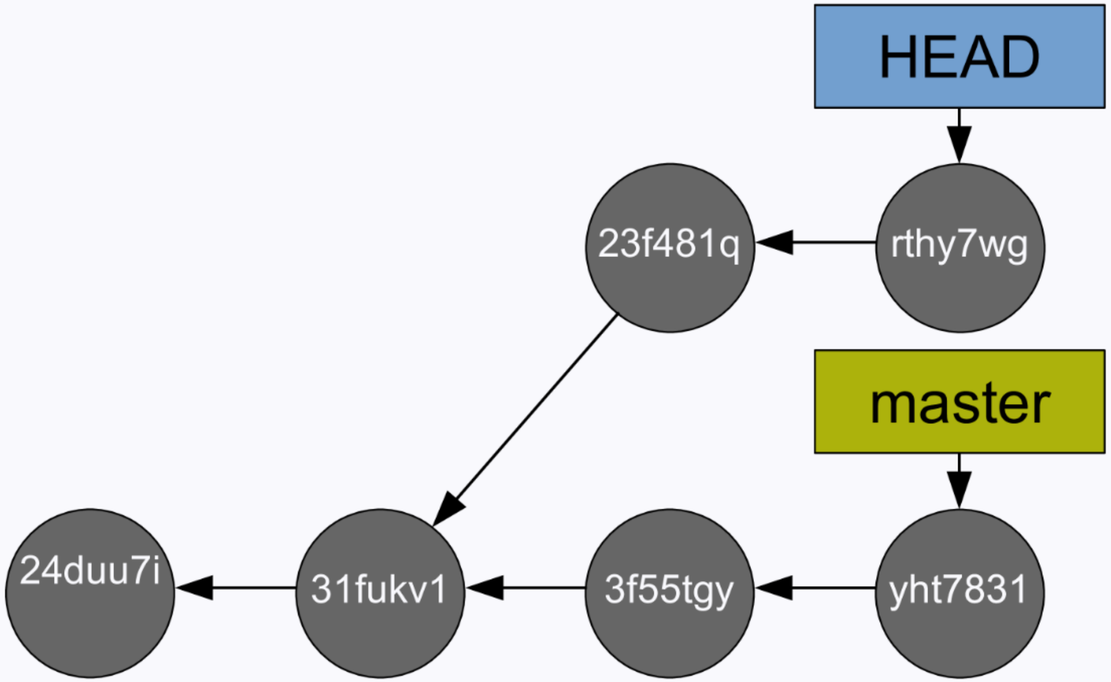
In [ ]:
```
git status
```

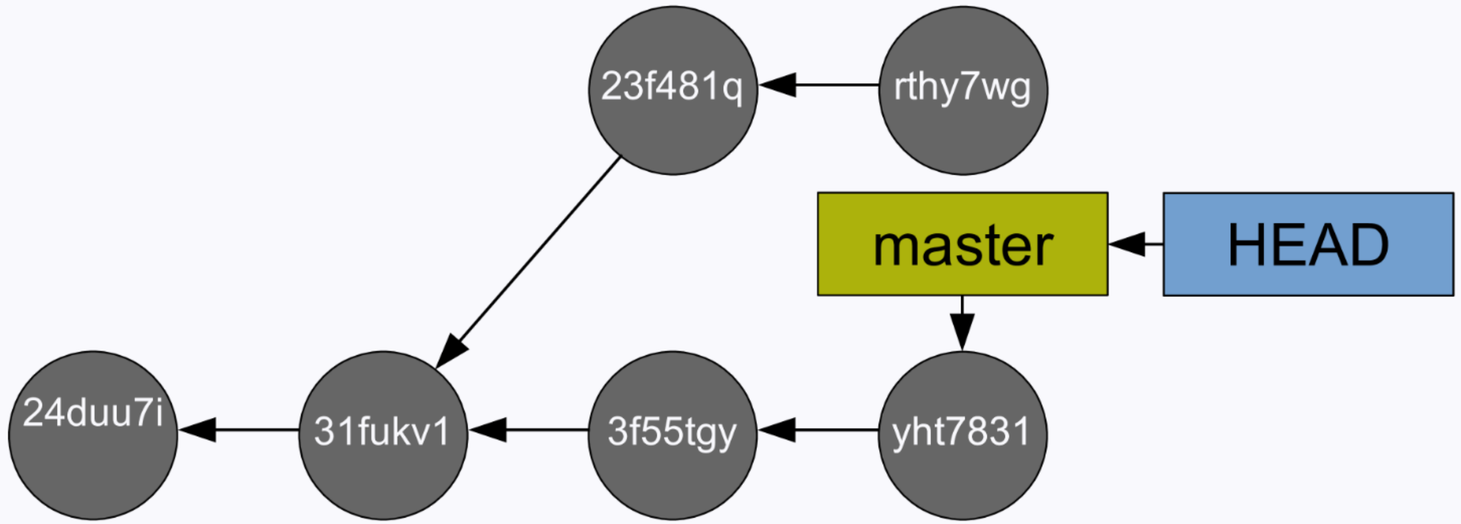
In [ ]:
```
git checkout master
```


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
23f481q
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
23f481q
rthy7wg
master
24duu7i
31fukv1
3f55tgy
yht7831


23f481q
rthy7wg
master
HEAD
24duu7i
31fukv1
3f55tgy
yht7831

Detached HEAD

Safe approach

In [ ]:
```
git checkout xxxx
```

In [ ]:
```
echo "lala" >> ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Exploration from commit xxx"
```

In [ ]:
```
echo "tutut" >> ms/enso_effect.md
```

In [ ]:
```
git commit -a -m "Another commit on that branch"
```

In [ ]:
```
git status
```

In [ ]:
```
git checkout xxxx
```

In [ ]:
```
echo "lala" >> ms/enso_effect.md
```

In [ ]:
```
git status
```

In [ ]:
```
git commit -a -m "Exploration from commit xxx"
```

In [ ]:
```
echo "tutut" >> ms/enso_effect.md
```

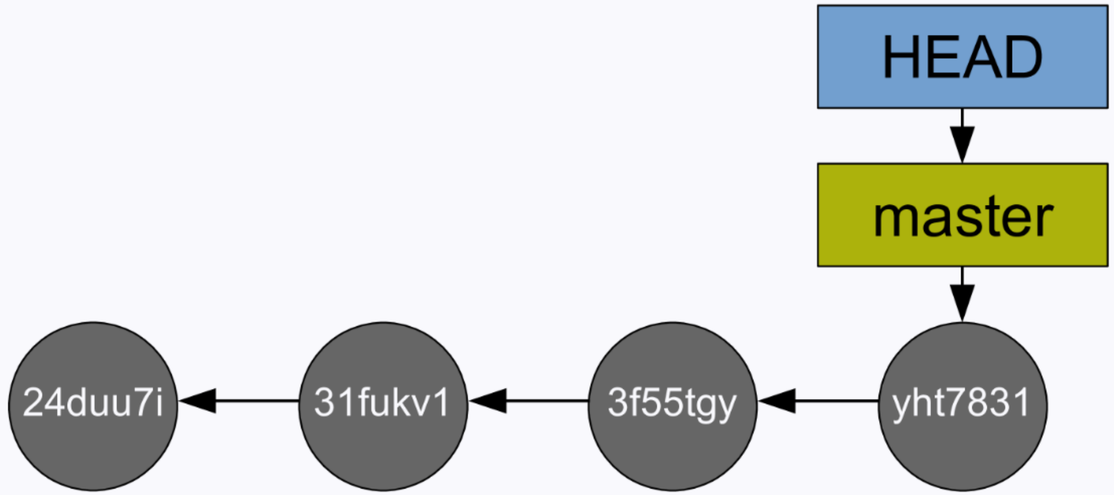
In [ ]:
```
git commit -a -m "Another commit on that branch"
```

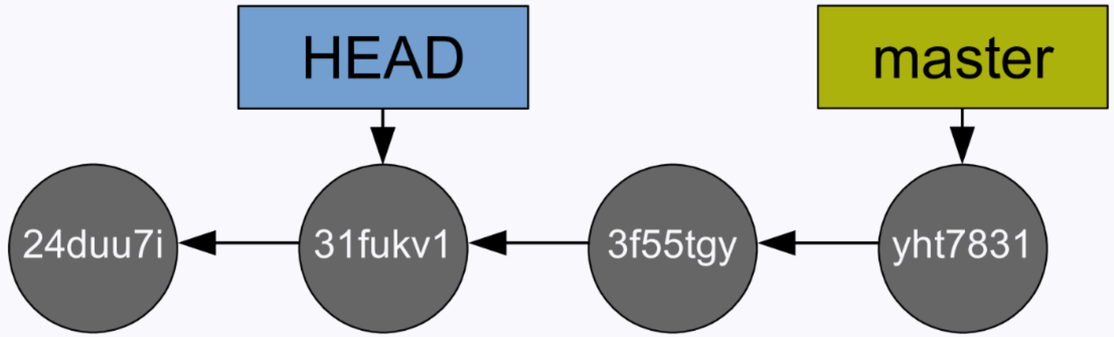
In [ ]:
```
git status
```

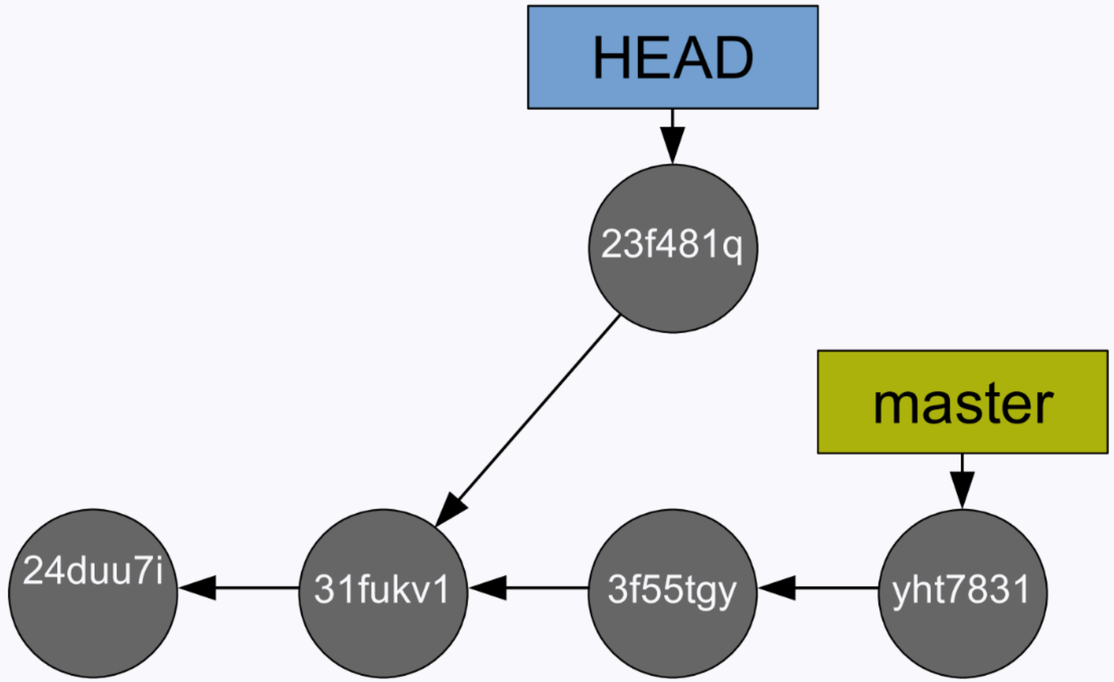
In [ ]:
```
git checkout -b alternative
```

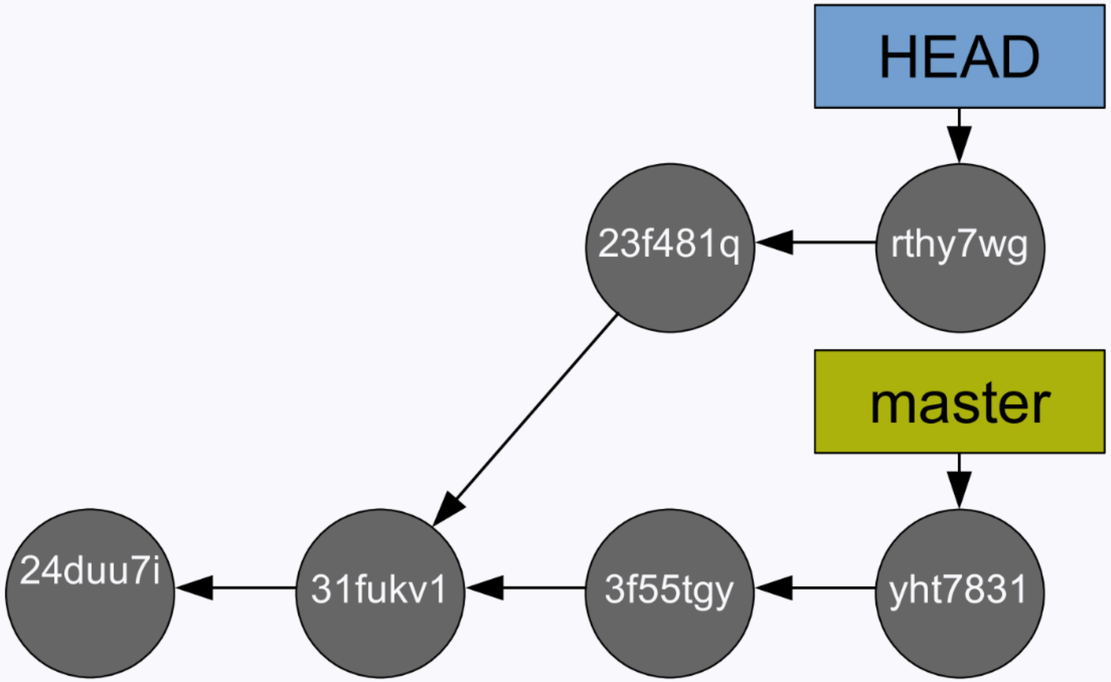
In [ ]:
```
git status
```

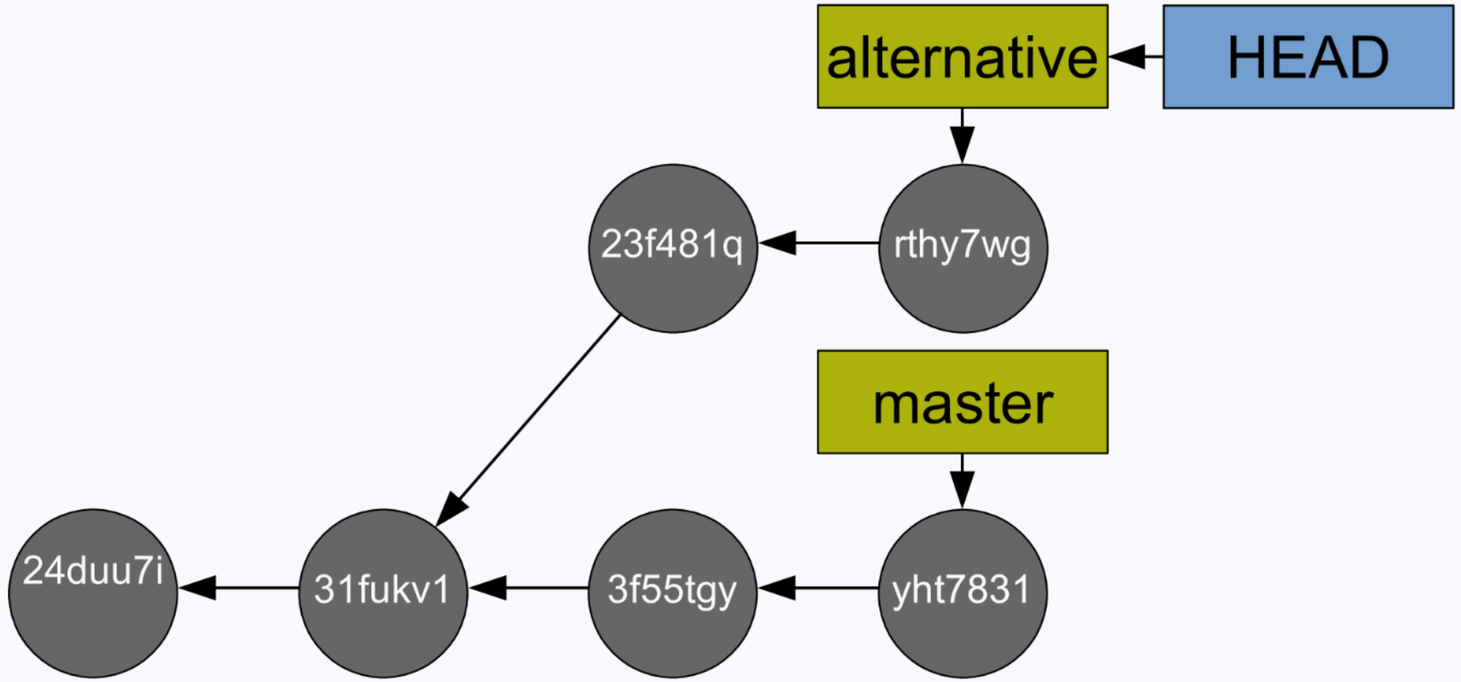
In [ ]:
```
git checkout master
```

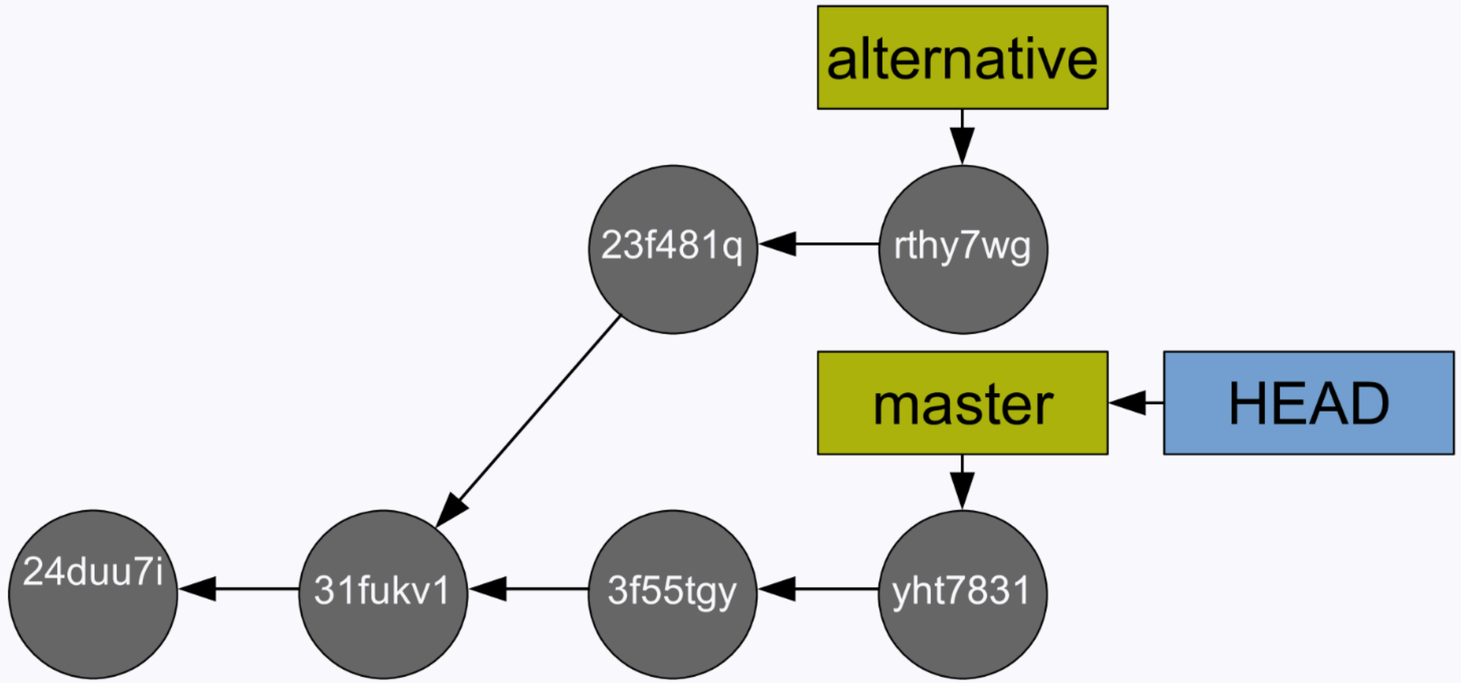

HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
23f481q
master
24duu7i
31fukv1
3f55tgy
yht7831


HEAD
23f481q
rthy7wg
master
24duu7i
31fukv1
3f55tgy
yht7831


alternative
HEAD
23f481q
rthy7wg
master
24duu7i
31fukv1
3f55tgy
yht7831


alternative
23f481q
rthy7wg
master
HEAD
24duu7i
31fukv1
3f55tgy
yht7831

Detached HEAD

How to recover if you loose commits?

Do this as soon as you can as the commits which are not on a branch will be deleted at the next garbage collection.

In [ ]:
```
git reflog
```

In [ ]:
```
git checkout xxxx
```

In [ ]:
```
git checkout -b new_branch
```

Codes

Safe

You can do this safely at any time as you can always go back to where you were before doing it.

! Data loss

Warning: this involves the loss of some information. Make sure that you do not want that information before doing this.

! Collaboration

Warning: this should **not** be done on something you already pushed to a remote when you are collaborating with others.

Codes

Safe

Workflows with branches and `git revert` are safe. They can make for tortuous and messy histories however.

! Data loss

Information can be lost when you:

- discard uncommitted work,
- let the garbage collection eliminate commits that are not on a branch,
- discard stashes that haven't been reapplied.

In any of these situations, make sure you really don't want to keep that data in your history.

! Collaboration

Whenever you touch at commits, there is a potential for messing up the workflow of collaborators.

Best to keep these for local work.

For local work (before pushing to a remote) however, they allow to fix horrible histories.

Reverting Safe

The working directory must be clean.

Create a new commit which reverses the effect of past commit(s).

In [ ]:
```
git log --graph --oneline
```

In [ ]:
```
echo "Add line before reverting" >> ms/enso_effect.md
```

In [ ]:
```
git commit -a -m "Add test line"
```

In [ ]:
```
git log --graph --oneline
```

In [ ]:
```
cat ms/enso_effect.md
```

In [ ]:
```
git revert HEAD~
```

Create a new commit which reverses the effect of past commit(s).

In [ ]:
```
git log --graph --oneline
```

In [ ]:
```
echo "Add line before reverting" >> ms/enso_effect.md
```

In [ ]:
```
git commit -a -m "Add test line"
```

In [ ]:
```
git log --graph --oneline
```

In [ ]:
```
cat ms/enso_effect.md
```

In [ ]:
```
git revert HEAD~
```

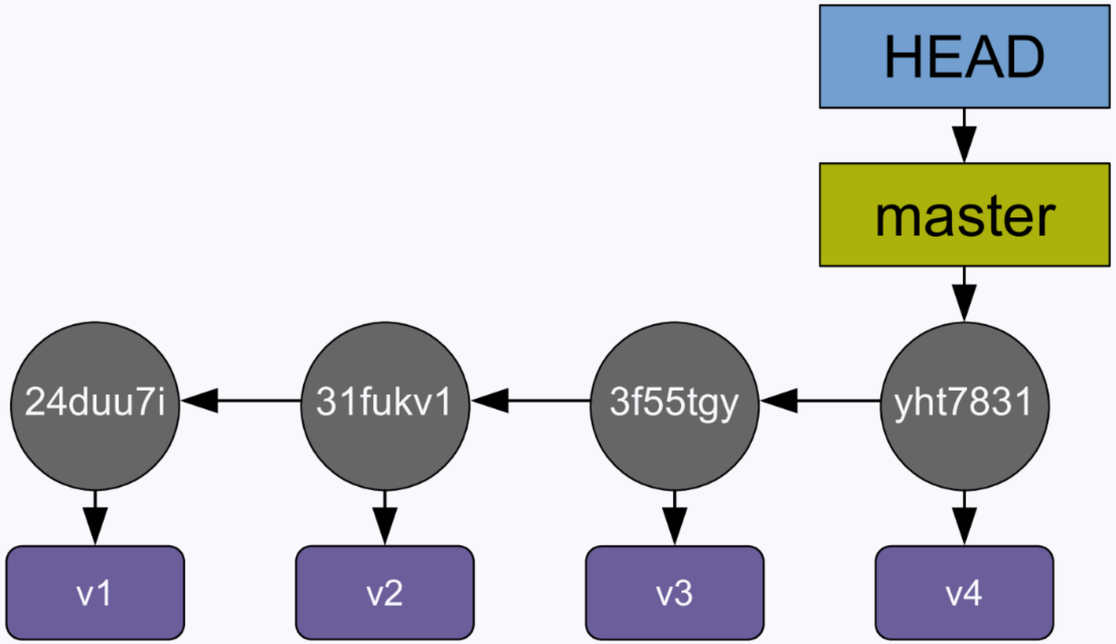
In [ ]:
```
git log --graph --oneline
```

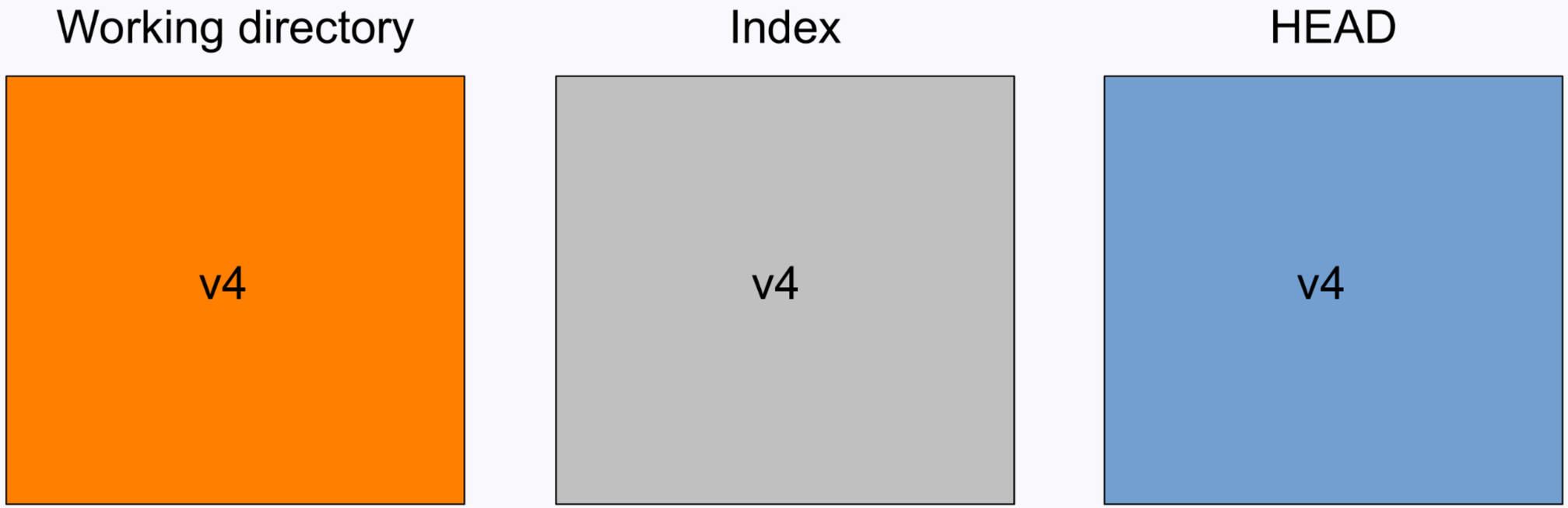
In [ ]:
```
cat ms/enso_effect.md
```

In [ ]:
```
git checkout HEAD~
```

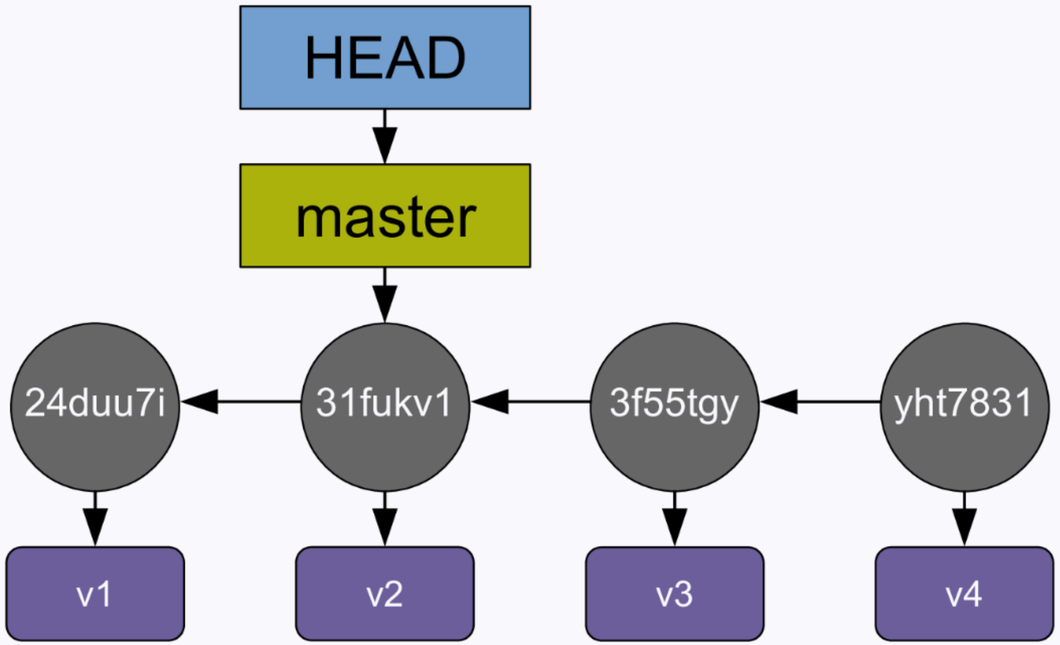
In [ ]:
```
git checkout -b new_start
```

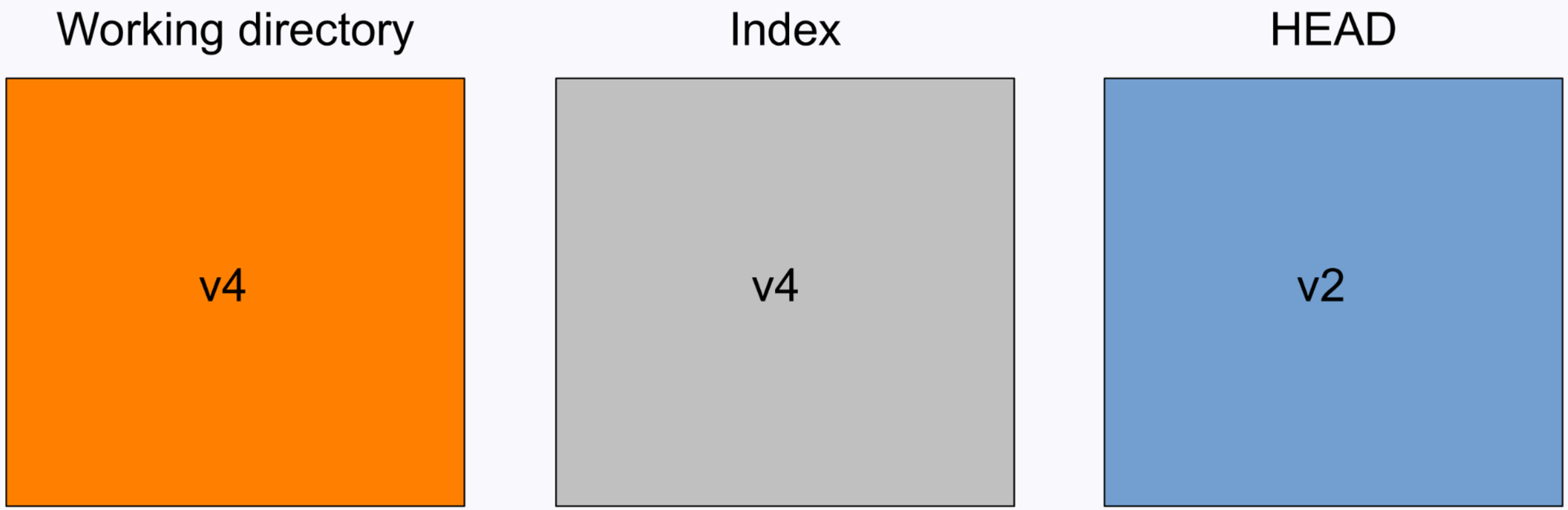
Git reset

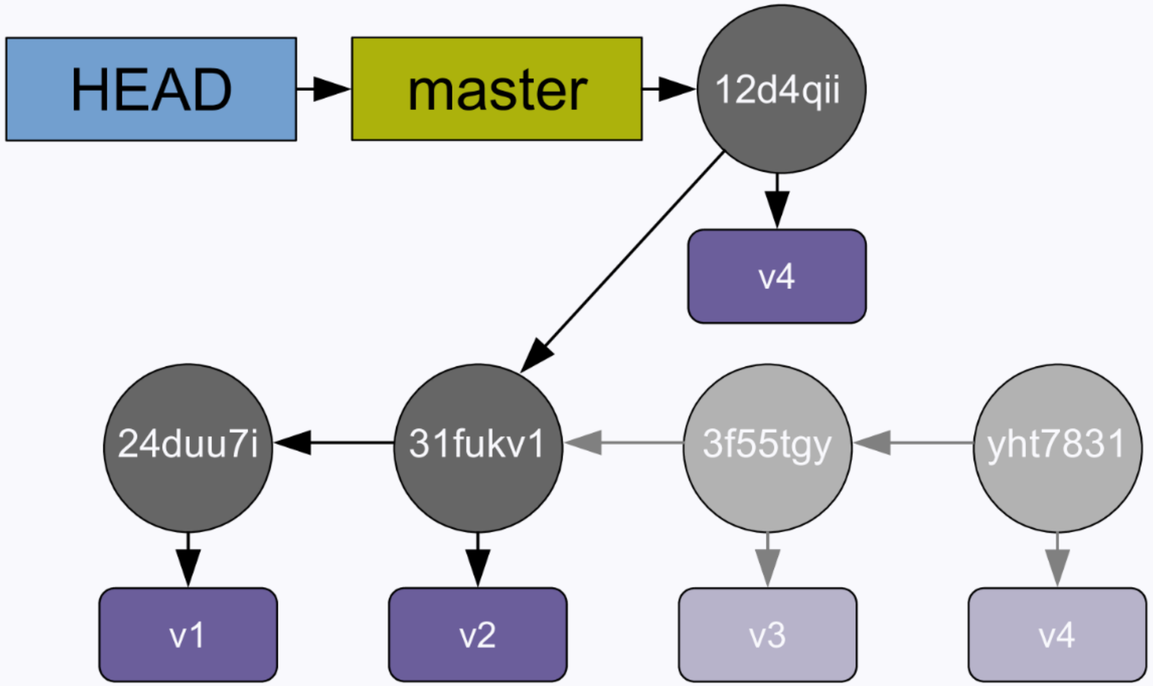

HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4

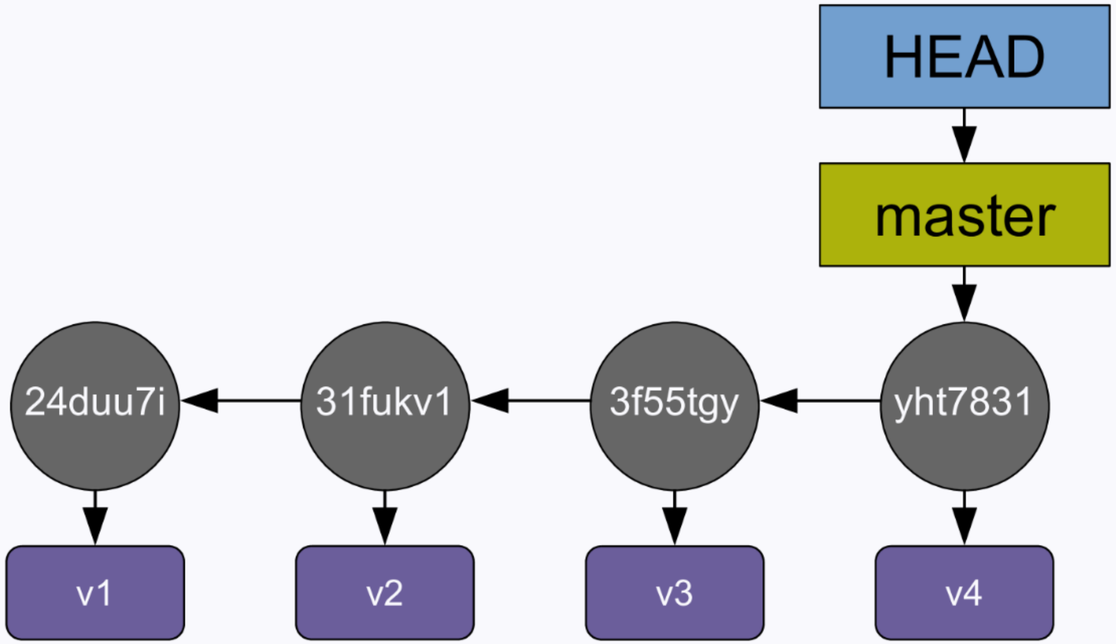

Working directory
v4
Index
v4
HEAD
v4

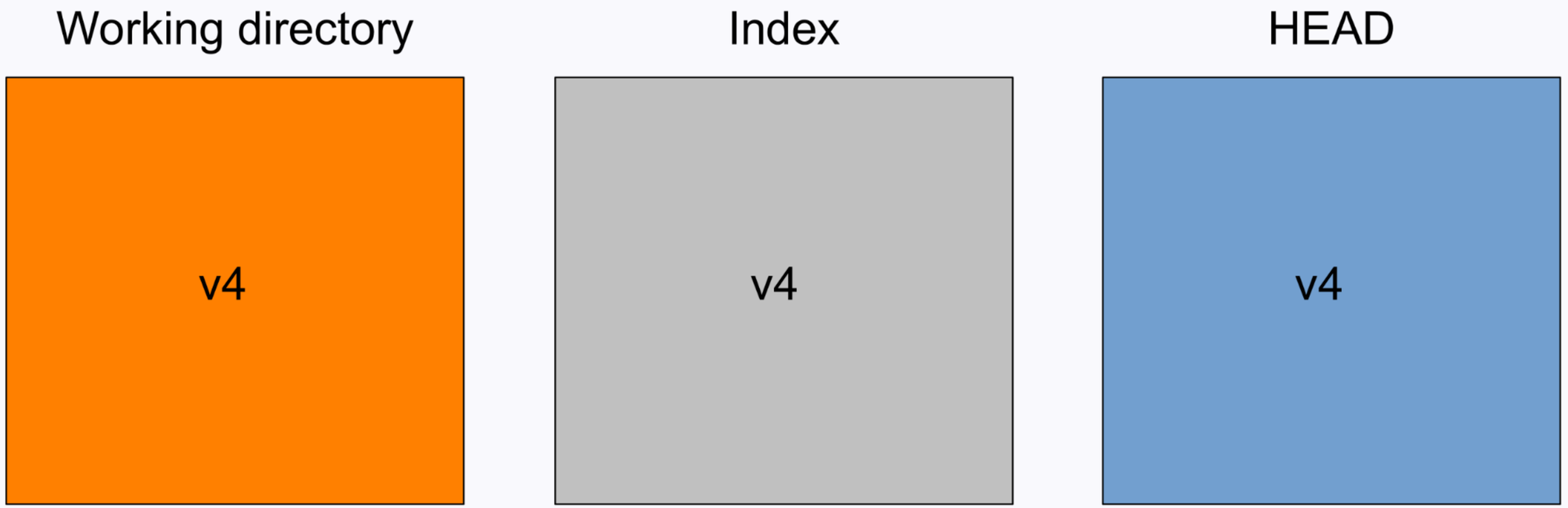
git reset --soft HEAD~2


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4

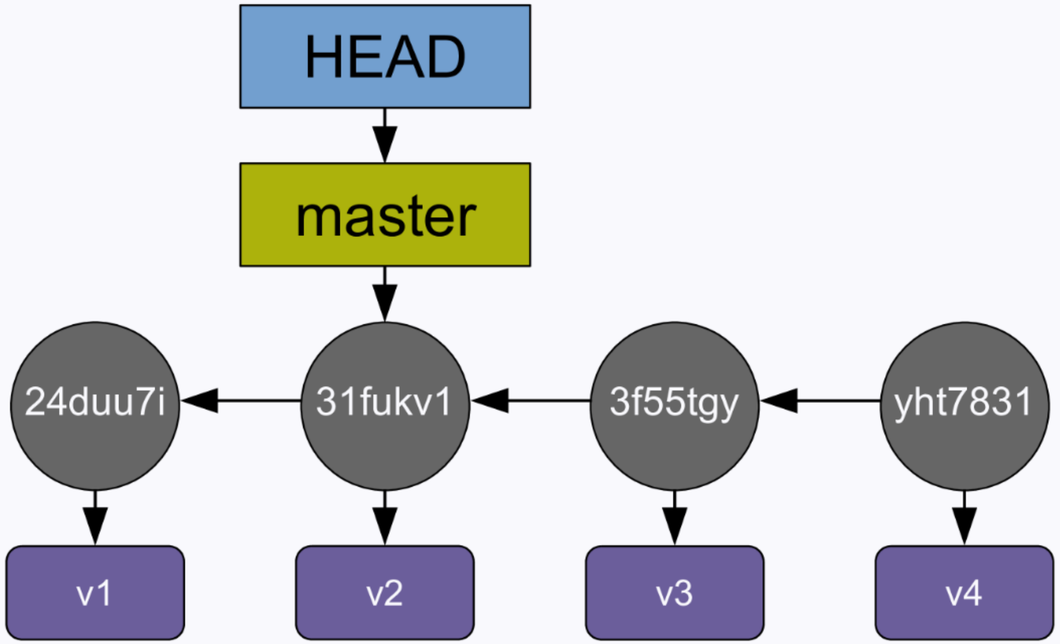

Working directory
Index
HEAD
v4
v4
v2

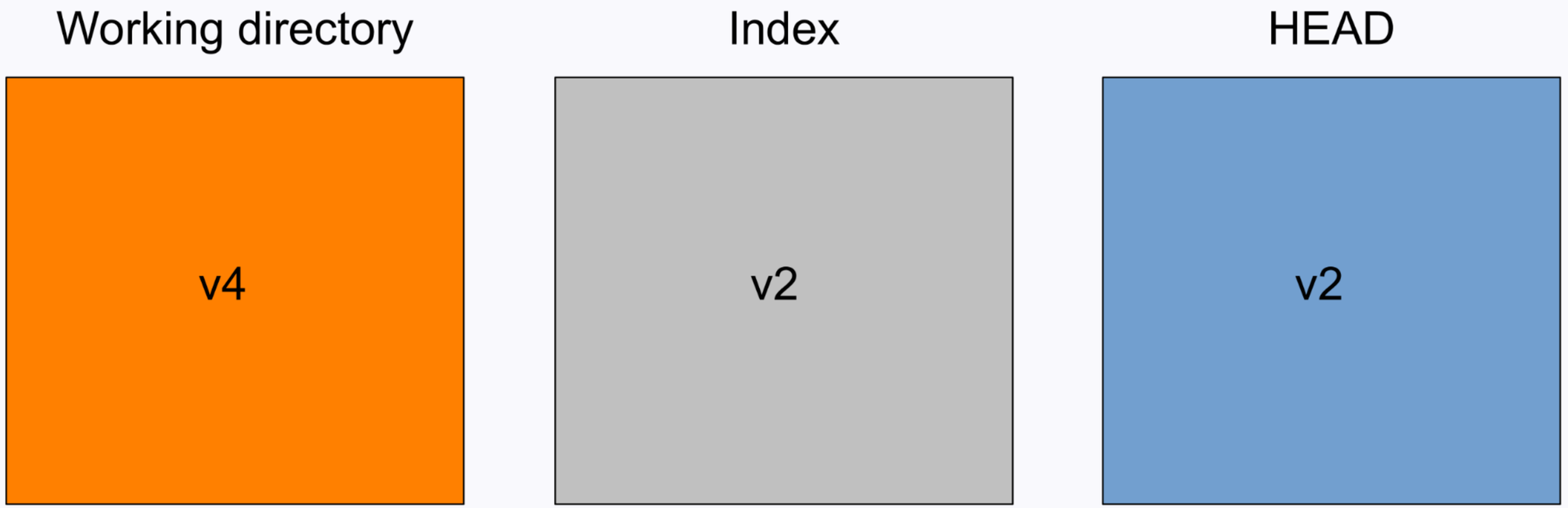

HEAD
master
12d4qii
v4
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4


HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4

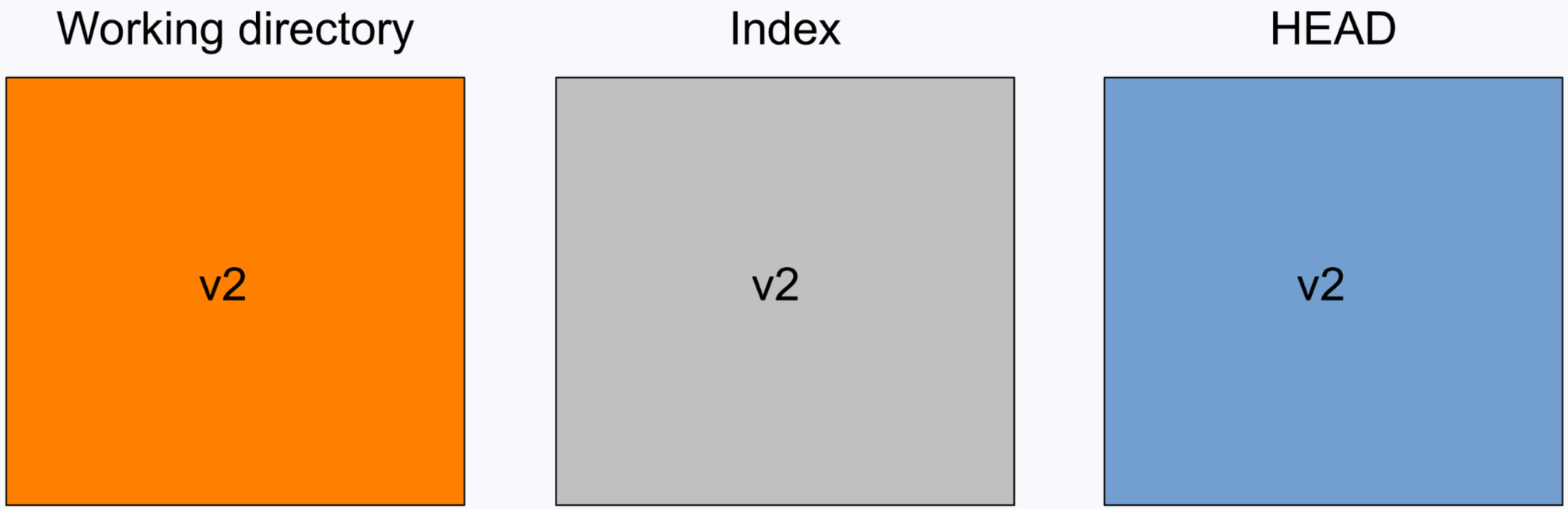

Working directory
Index
HEAD
v4
v4
v4

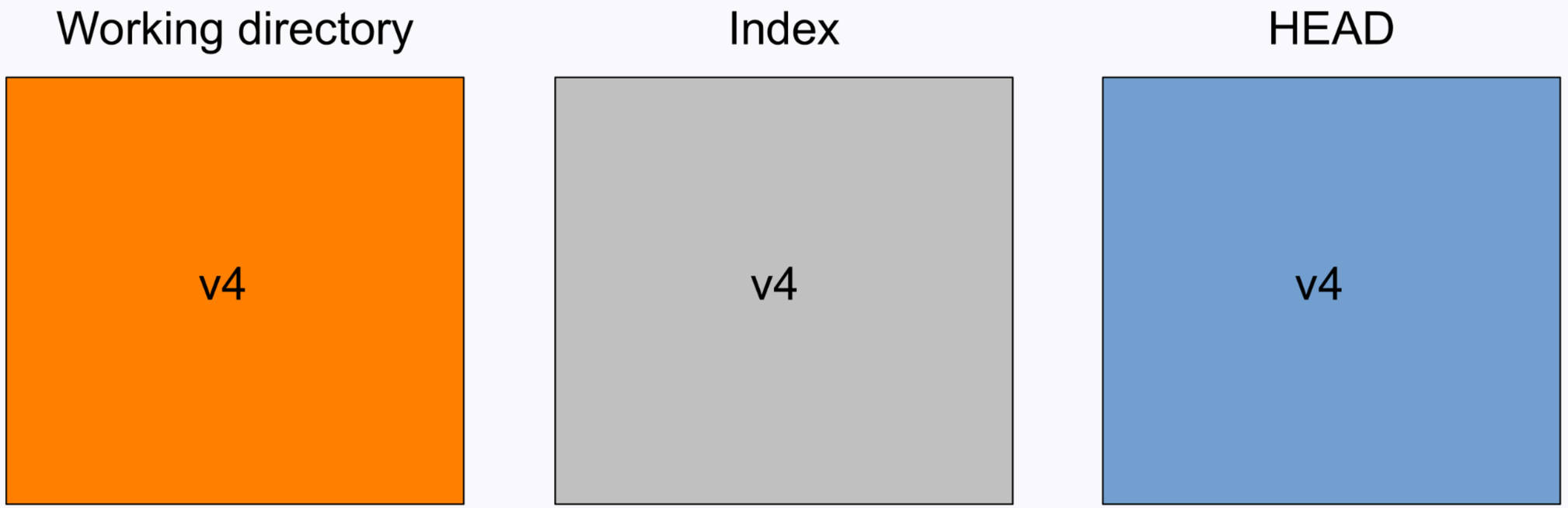
git reset HEAD~2

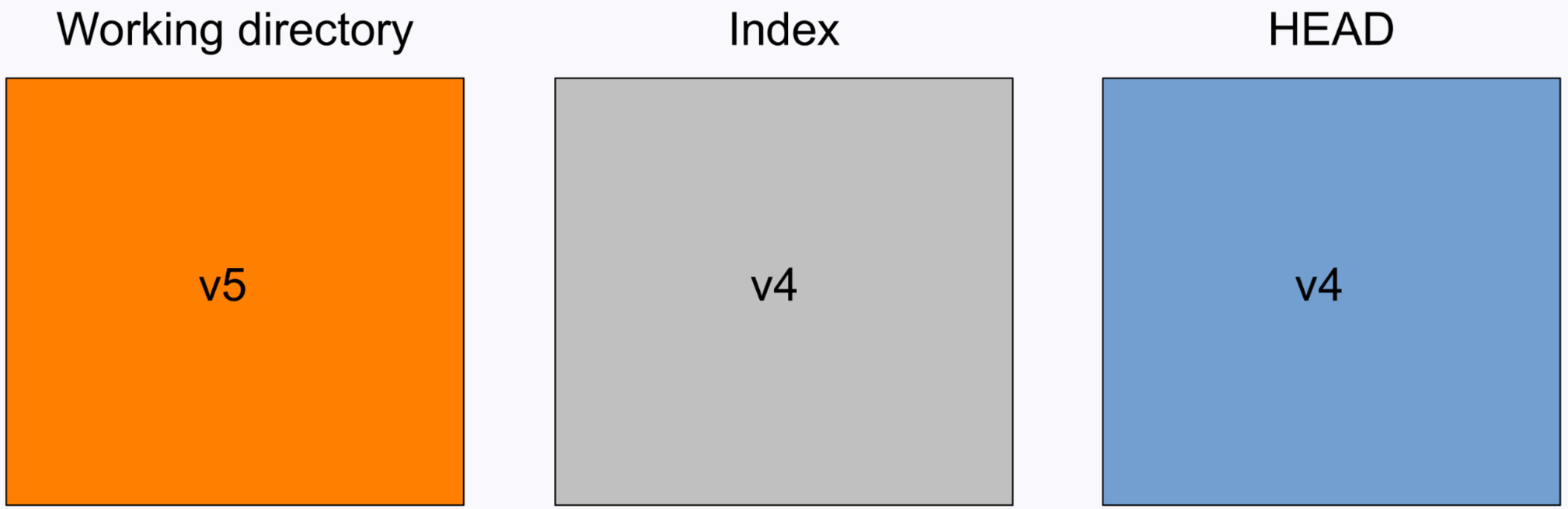

HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4

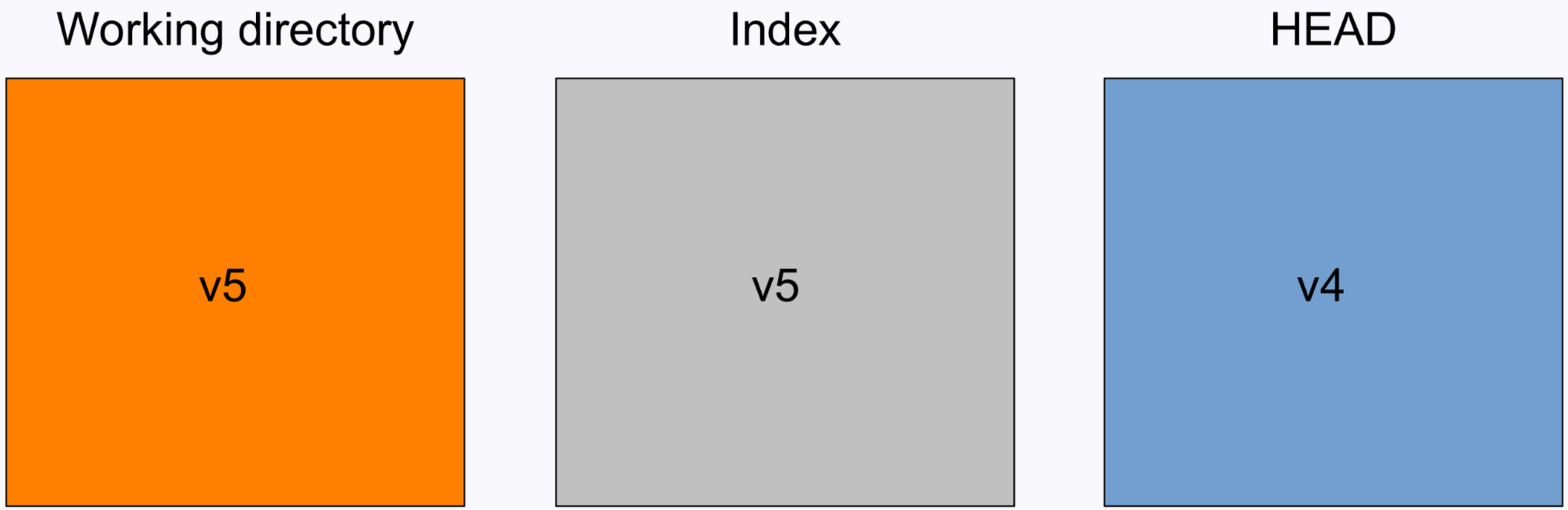

Working directory
Index
HEAD
v4
v2
v2

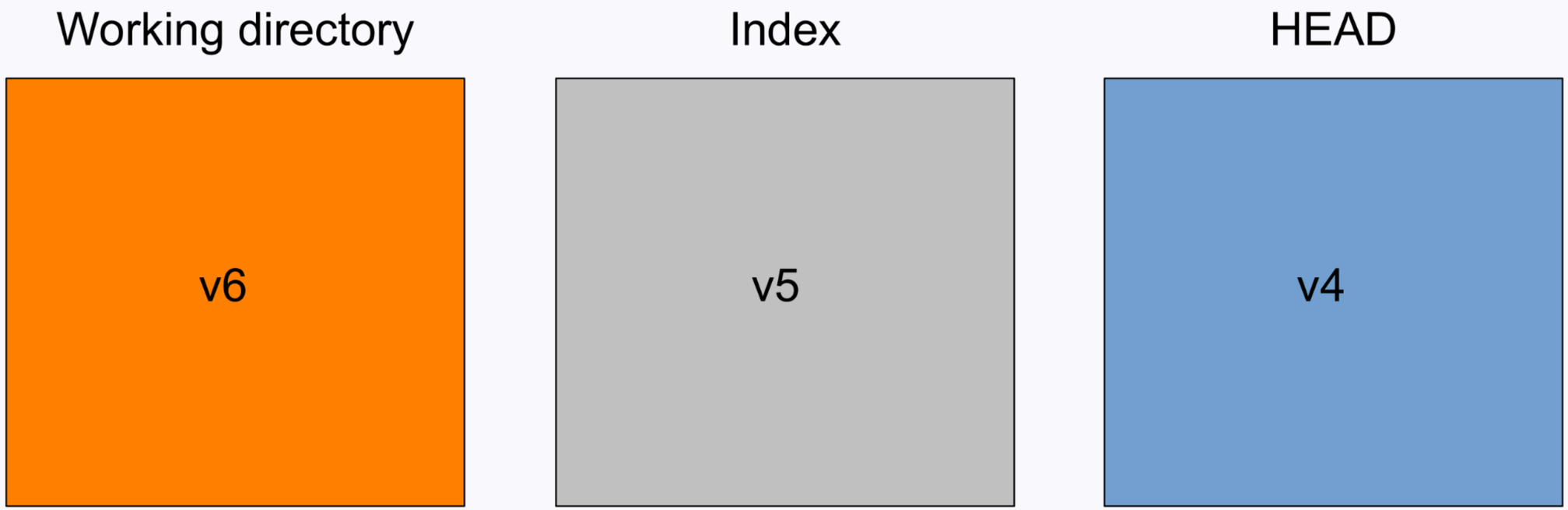
git reset --hard HEAD~2


Working directory
Index
HEAD
v2
v2
v2

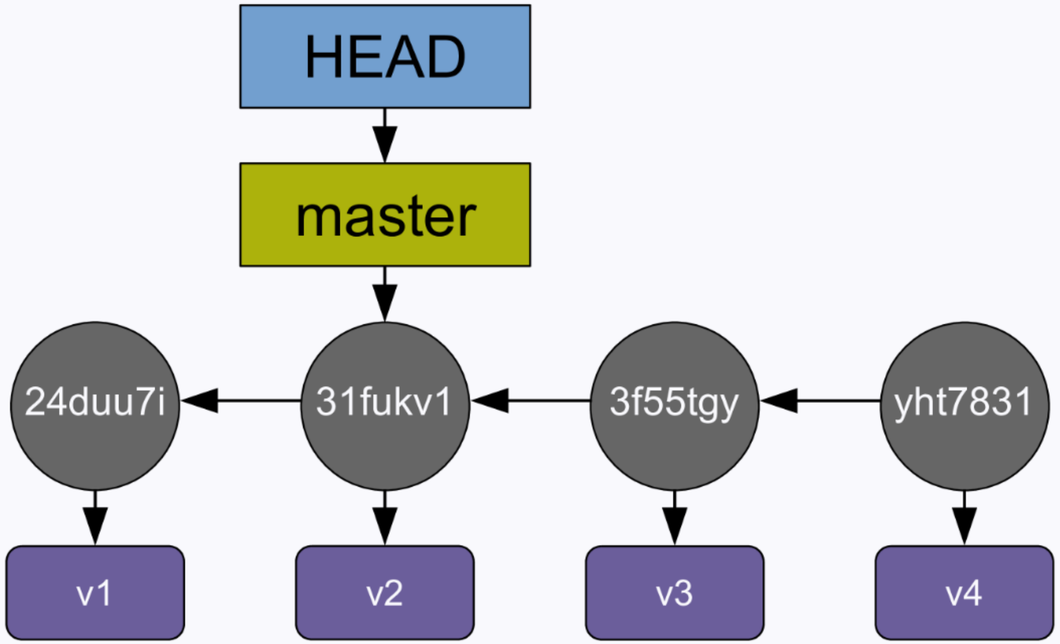

Working directory
Index
HEAD
v4
v4
v4

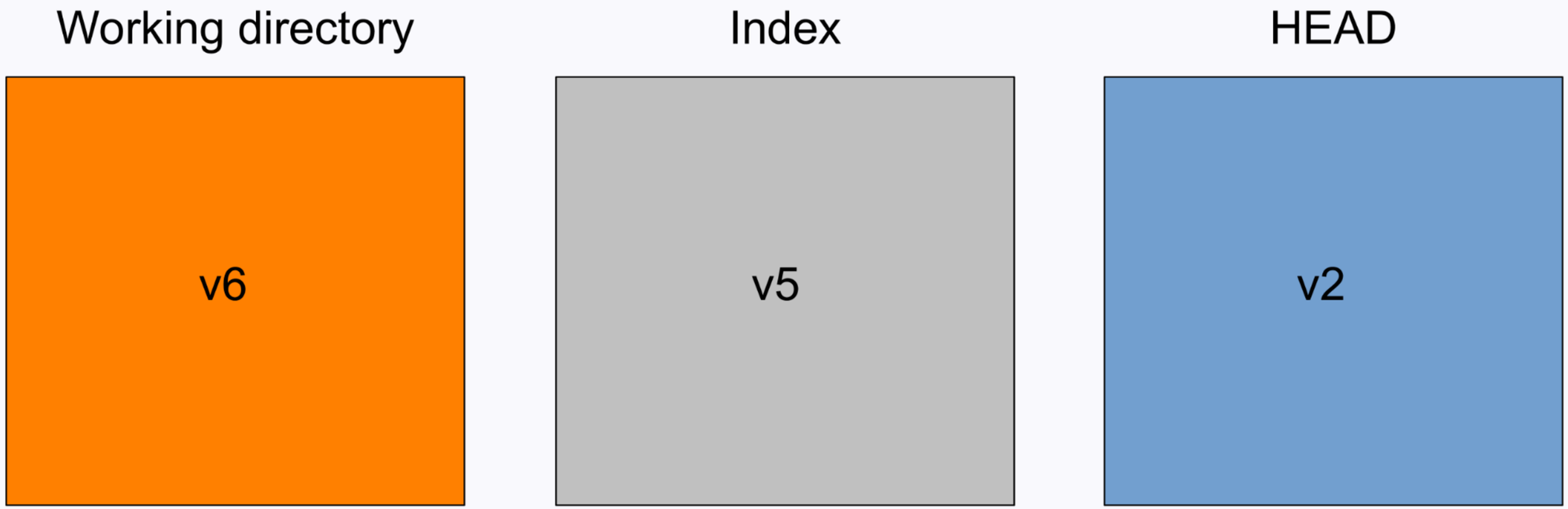

Working directory
Index
HEAD
v5
v4
v4


Working directory
v5
Index
v5
HEAD
v4

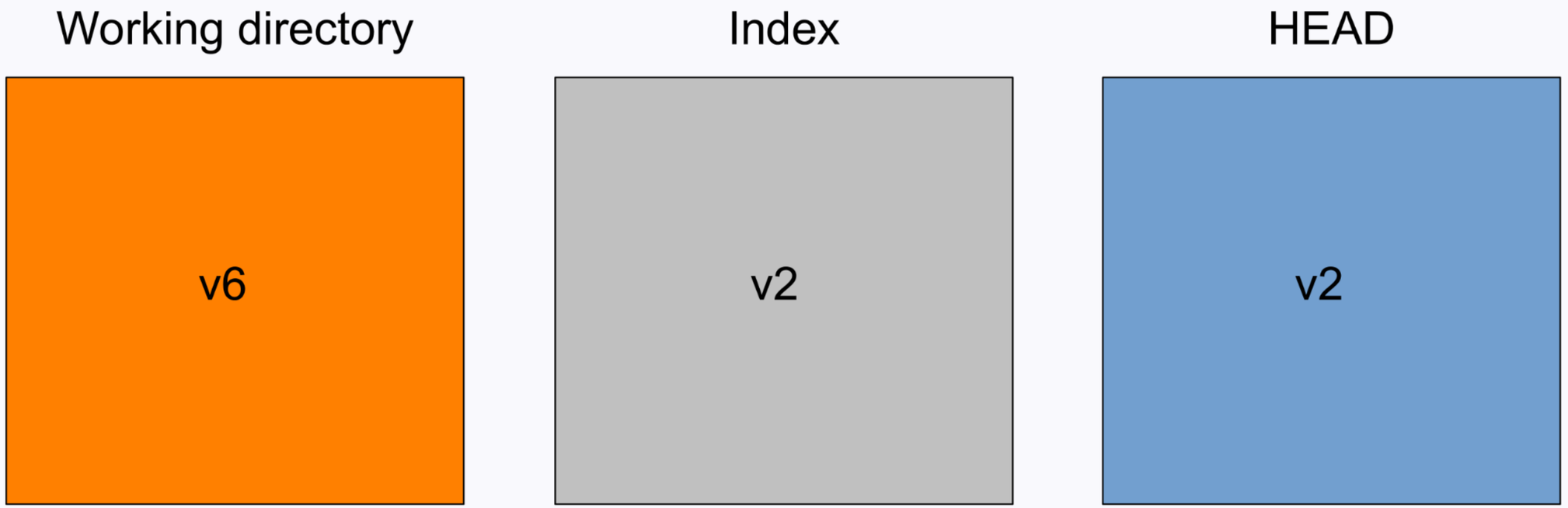

Working directory
Index
HEAD
v6
v5
v4

git reset --soft HEAD~2

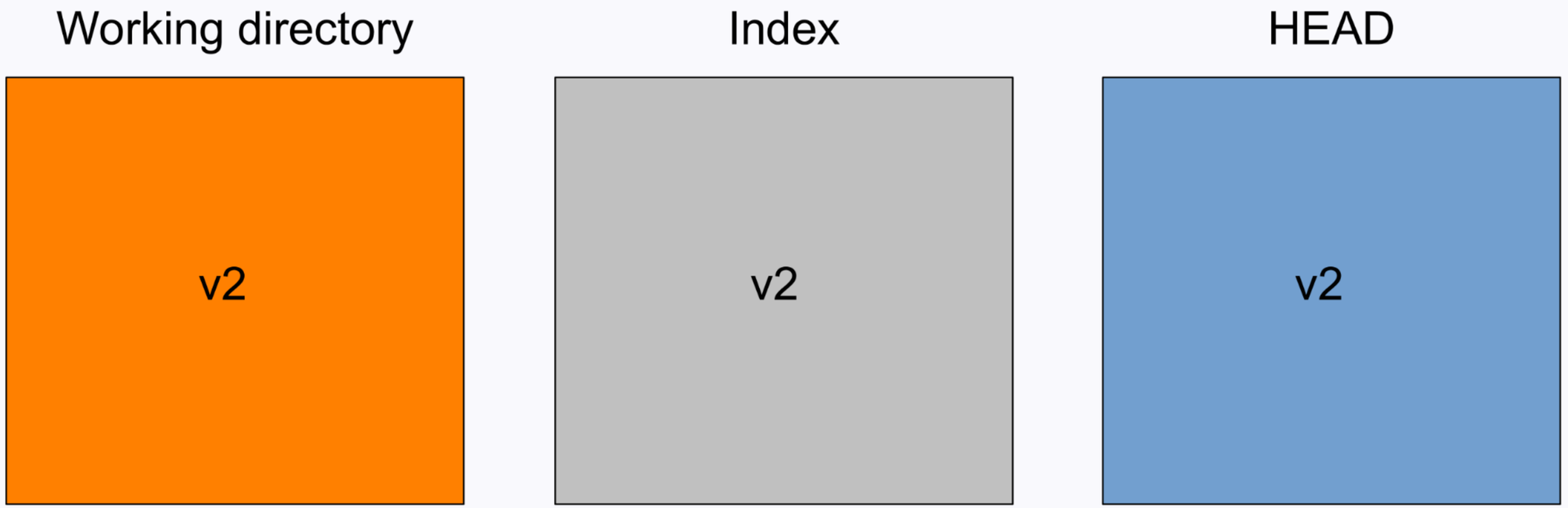

HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4

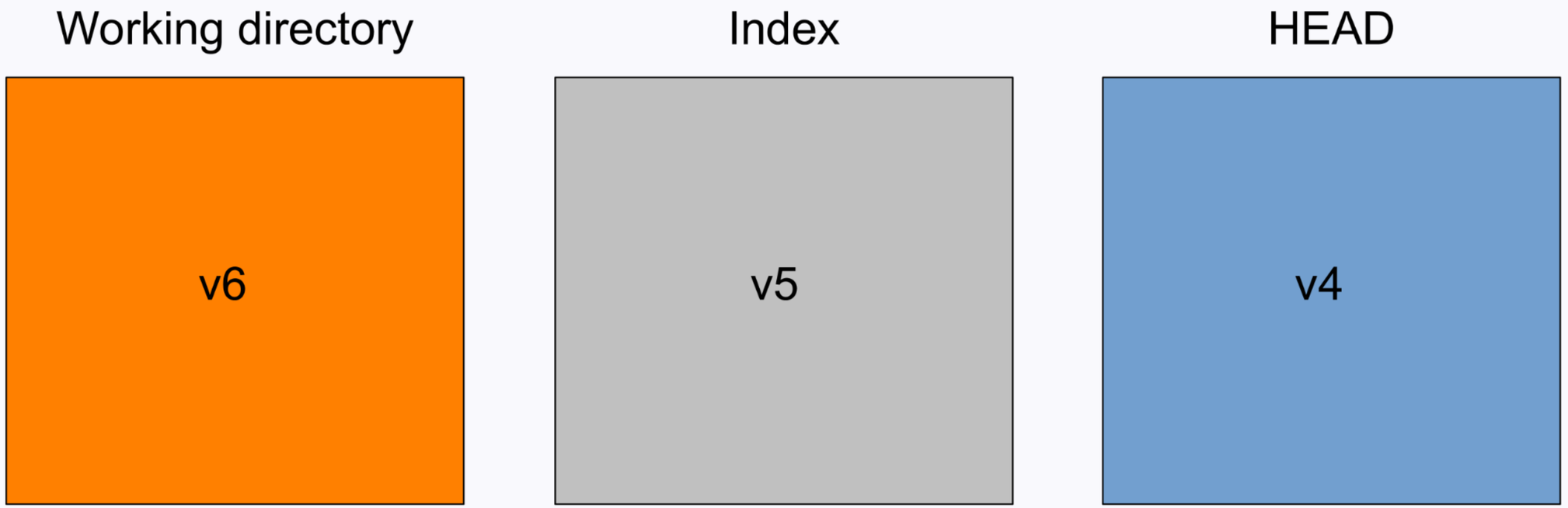

Working directory
Index
HEAD
v6
v5
v2

git reset HEAD~2

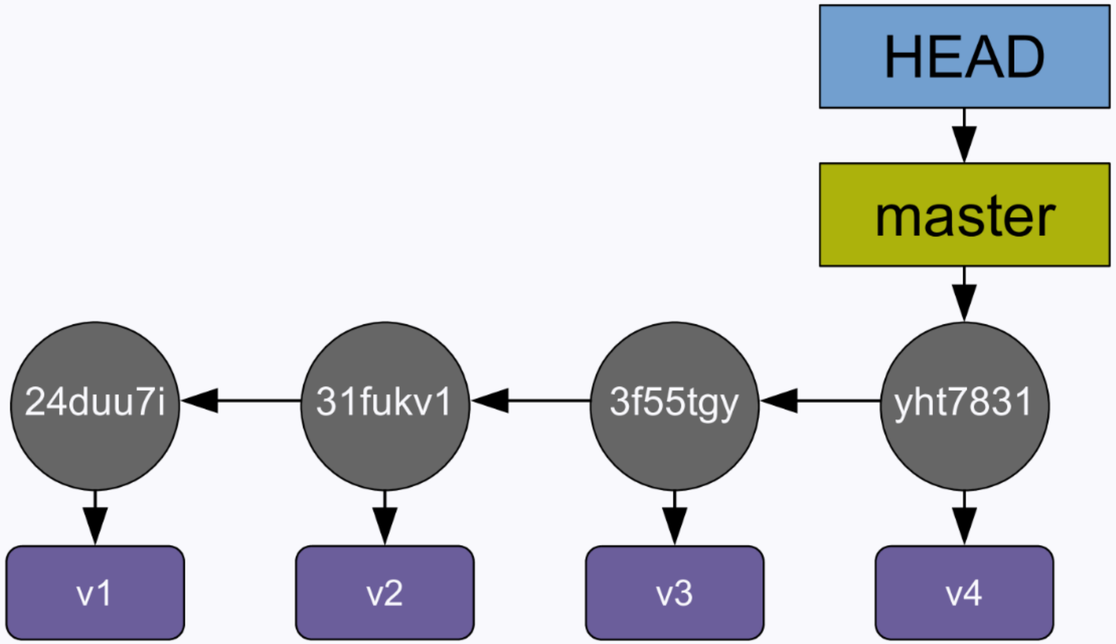

Working directory
Index
HEAD
v6
v2
v2

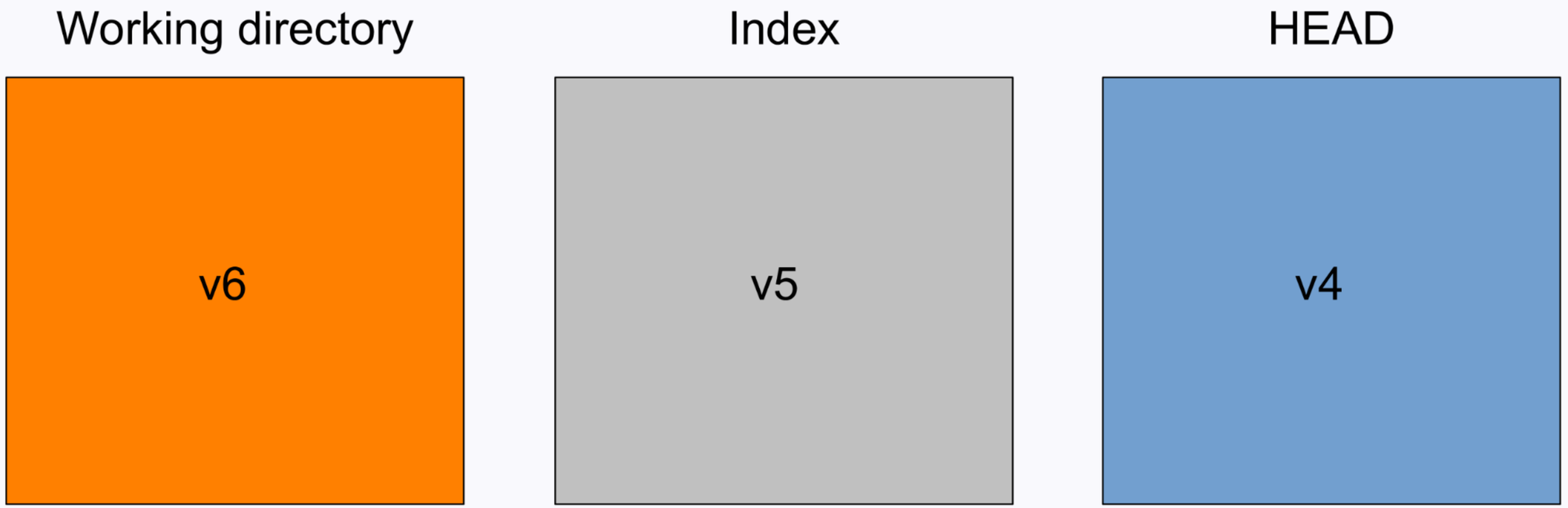
git reset --hard HEAD~2


Working directory
Index
HEAD
v2
v2
v2

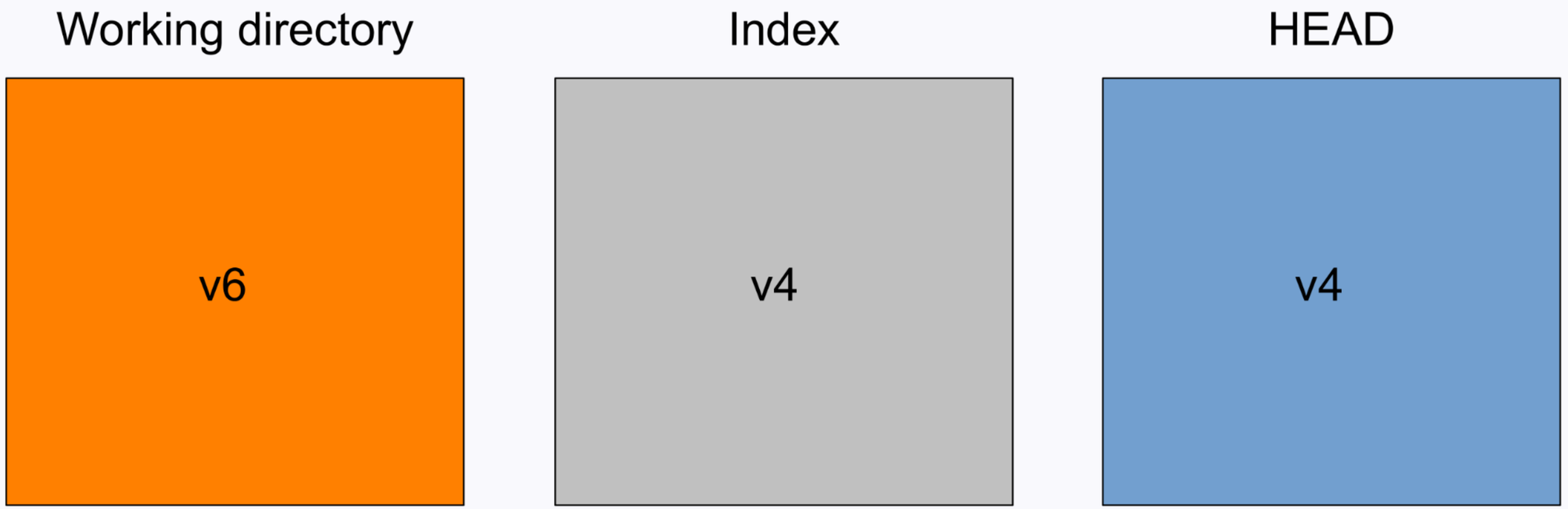

Working directory
Index
HEAD
v6
v5
v4

git reset --soft HEAD

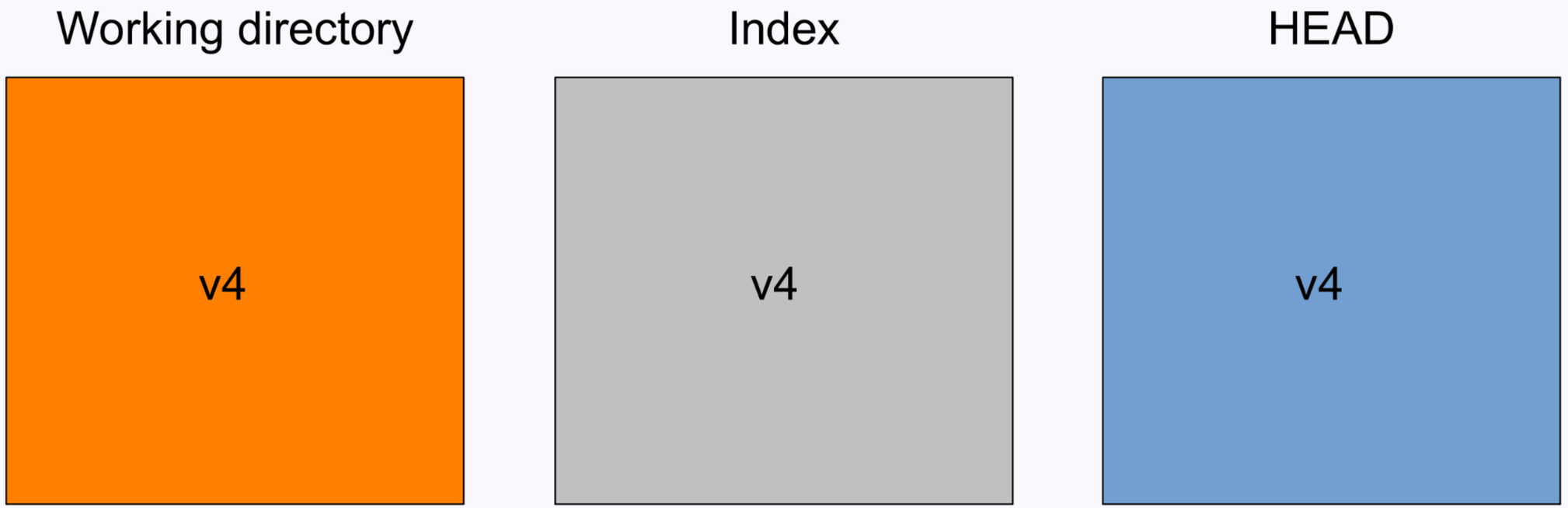

HEAD
master
24duu7i
31fukv1
3f55tgy
yht7831
v1
v2
v3
v4


Working directory
Index
HEAD
v6
v5
v4

git reset HEAD


Working directory
Index
HEAD
v6
v4
v4

git reset --hard HEAD


Working directory
Index
HEAD
v4
v4
v4

Undoing the last commit ! Collaboration

```
git reset --soft HEAD~
```

In [ ]:

Undoing several commits (while keeping the changes staged)

! Collaboration

```
git reset --soft HEAD~x
```

```
In [ ]:
```

Restoring the index (unstaging) Safe

Single file:

```
git reset HEAD <file>
```

All files:

```
git reset HEAD
```

Note: for versions newer than 2.23, Git suggests using a new command: `git restore --staged <file>`
See my answer on Stack Overflow for more details.

Undoing the last commit and unstaging

! Collaboration

```
git reset HEAD~
```

```
In [ ]:
```

Undoing several commits and unstaging

! Collaboration

```
git reset HEAD~x
```

```
In [ ]:
```

Throwing away changes since last commit

! Data loss

```
git reset --hard HEAD
```

```
In [ ]:
```

Undoing the last commit & throwing away changes

! Collaboration ! Data loss

```
git reset --hard HEAD~
```

In [ ]:

Undoing several commits & throwing away changes

! Collaboration ! Data loss

```
git reset --hard HEAD~x
```

In [ ]:

Throwing away unstaged changes (unmodifying)

! Data loss

Single file:

```
git checkout -- <file>
```

All files:

```
git checkout -- .
```

Note: for versions newer than 2.23, Git suggests using a new command: `git restore --staged <file>`
See my answer on Stack Overflow for more details.

Modifying the last commit message

! Collaboration

In [ ]:
```
git commit --amend -o
```

In [ ]:
```
git commit --amend -o -m "Much better commit message"
```

Modifying the last commit

! Collaboration

In [ ]:
```
git commit --amend
```

In [ ]:
```
git commit --amend --no-edit
```

In [ ]:
```
git commit --amend -m "New commit message for the replacement commit"
```

Modifying older commits

! Collaboration

In [ ]:
```
git rebase -i HEAD~3
```

Remotes

What is a remote?

Any version of the project that is somewhere else.
"Somewhere else" can be anywhere, including on the same machine.

Usually, remotes are on the internet (on hosting services such as GitHub, GitLab, or Bitbucket) or on servers.

This allows easy collaboration.

Adding remotes

First, create the remote

Adding remotes

Then, add it to your project

```
git remote add <remote-name> <remote-address>
```

The `<remote-address>` can be, amongst others, in the form of:

- `<user>@<server>:<project>.git` for a server with SSH protocol
- `git@<hosting-site>:<user>/<project>.git` for a web hosting service accessed with SSH address
- `https://<hosting-site>/<user>/<project>.git` for a web hosting service accessed with HTTPS address

In [ ]:
```
git remote add origin git@gitlab.com:prosoitos/ocean_temp.git
```

In [ ]:
```
git remote add origin https://gitlab.com/prosoitos/ocean_temp.git
```

Listing remotes

In [ ]:
```
git remote
```

In [ ]:
```
git remote -v
```

Renaming remotes

```
git remote rename <old-name> <new-name>
```

In [ ]:

Removing remotes

```
git remote remove <remote-name>
```

In [ ]:

Fetching

```
git fetch <remote-name>
```

In [ ]:

Pulling (fetching + merging)

```
git pull
```

In [ ]:

Pushing

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

To associate a branch with a remote, you can run:

```
git push -u <remote-name> <branch-name>
```

After which, you will only have to run:

```
git push
```

(Unless you want to push a new branch. Then you have to associate that new branch to the remote with `-u` as well).

In [ ]:
```
git push origin master
```

In [ ]:
```
git push
```

To associate a branch with a remote, you can run:

```
git push -u <remote-name> <branch-name>
```

After which, you will only have to run:

```
git push
```

(Unless you want to push a new branch. Then you have to associate that new branch to the remote with `-u` as well).

In [ ]:
```
git push origin master
```

In [ ]:
```
git push
```

In [ ]:
```
git push -u origin master
```

In [ ]:
```
git push
```

Pushing tags

Pushing will not push tags to the remote unless you add the `--tags` tag.

In [ ]:
```
git push origin --tags
```

In [ ]:
```
git push origin --delete <tagname>
```

Collaborating

from crystallize comics

Cloning a repo

```
git clone git@<hosting-site>:<user>/<project>.git
git clone https://<hosting-site>/<user>/<project>.git
```

When cloning, the remote is automatically named `origin` and the main branch is automatically associated with the remote.

Let's practice with this project.

In [ ]:
```
git clone git@gitlab.com:prosoitos/collab.git
```