**Git commands**

Git is a software that allows you to keep track of changes made to a project over time. Git works by recording the changes you make to a project, storing those changes, then allowing you to reference them as needed.

**git init** the Word init means set up all the tools to begin tracking changes made to the project.

1. **Working Directory 🡪** where you’ll be doing all the work: creating, editing, deleting, and organizing files.
2. **Staging Area 🡪** where you’ll list changes you make to the working directory
3. **Repository 🡪** where Git permanently stores those changes as different *versions* of the project

The Git workflow consist of editing files in the working directory, adding files to the staging area, and saving changes to a Git repository. In Git, we save changes with a *commit*.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **WORKING DIRECTORY** | **STAGING AREA** | **REPOSITOR** |
| Make changes to files:  + additions  - deletions modifications | Bring changes into the staging area | Save changes to the repository as a ‘commit’ |

As you write in the bash, you will be changing the contents of the working directory. You can check the status of those changes with: **git status**

**$ git status**

Initial commit

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

**init\_test.rb  
 scene-1.txt**

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

In the output, notice the file in red under untracked files. Untracked means that Git sees the file but has not started tracking changes yet.

In order for Git to start tracking **scene-1.txt**, the files need to be added to the staging area. We can add a file to staging area with: **git add filename**

# Adding files to the Staging Area

**$ git add scene-1.txt**

**$ git status**

On branch master

Initial commit

Changes to be committed:

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

**new file: scene-1.txt**

Untracked files:

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

**add\_test.rb  
 init\_test.rb**

In the output, notice that Git indicates the changes to be committed with “new file: scene-1.txt” in green text. Here Git tells us the file was added to the staging area.

# git diff

Imagine that we type another line in **scene-1.txt**. Since the file is tracked, we can check the differences between the working directory and the staging area with: **git diff filename**

Here, filename is the actual name of the file.

**$ git diff scene-1.txt**

diff --git a/scene-1.txt b/scene-1.txt

index ef88cdb..c812f88 100644

--- a/scene-1.txt

+++ b/scene-1.txt

**@@ -1,2 +1,5 @@**

“prior lines that had been added to the scene-1.txt file”

**+ line added to the file  
+ Another addition**

\ No newline at end of file

The prior lines that had been added to the scene-1.txt file are the staging area as indicated in white. Changes in the file are marked with a + and are indicated in green. **IMPORTANT**: press **q** on your keyboard to exit diff mode.

**$ git add scene-1.txt**

**$ git status**

On branch master

Initial commit

Changes to be committed:

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

**new file: scene-1.txt**

Untracked files:

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

**add\_test.rb  
 add\_tracked\_test.rb  
 init\_test.rb  
 wd\_change\_test.rb**

# git commit

A commit is the last step in our Git workflow. A commit permanently stores changes from the staging area inside the repository.

**git commit** is the command we’ll do next. However, one more bit of code is needed for a commit: the option **–m** followed by a message. Standard Convention for Commit Messages:

* Must be in quotation marks
* Written in the present tense
* Should be brief (50 characters or less) when using **–m**

**$ git commit -m "All lines of the file are completed"**

[master (root-commit) 1601a01] All lines of the file are completed

1 file changed, 5 insertions(+)

create mode 100644 scene-1.txt

# git log

Often with Git, you’ll need to refer back to an earlier version of a project. Commits are stored chronologically in the repository and can be viewed with: **git log**

**$ git log**

**commit 1601a01bf70fd085a59222d65d790cb7dff46ce3**Author:   
Date: Sat Feb 10 14:02:18 2018 +0000  
 All required lines of the file are completed

In the output, notice:

* A 40-character code, called a SHA, that uniquely identifies the commit. This appears in orange text.
* The commit author (you!), The date and time of the commit and the commit message

# Generalizations

Git is the industry-standard version control system for web developers. Use Git commands to help keep track of changes made to a project:

* **git init** creates a new Git repository
* **git status** inspects the contents of the working directory and staging area
* **git add** adds files from the working directory to the staging area
* **git diff** shows the difference between the working directory and the staging area
* **git commit** permanently stores file changes from the staging area in the repository
* **git log** shows a list of all previous commits

# Backtracking

When working on a Git project, sometimes we make changes that we want to get rid of. Git offers a few eraser-like features that allow us to undo mistakes during project creation.

In Git, the commit you are currently is known as the HEAD commit. In many cases, the most recently made commit is the HEAD commit. To see the HEAD commit enter: **git show HEAD**

The output of this command will display everthing the git log command displays for the HEAD commit, plus all the file changes that were committed.

**$ git show HEAD**

**commit ee061b38715941df7bcc76ca2580ffcd2f639549**

Author:   
Date:

“Commit message”

diff --git a/filename.extension b/filename.extension

index b12dd97..5dd5d4e 100644

--- a/ filename.extension  
+++ b/ filename.extension  
**@@ -12,3 +12,7 @@** Prior to last commit lines

**+New changes   
+More line changes commited  
+Last commited line.**  
\ No newline at end of file

# Git checkout

What if you decide to change the commit in the working directory, but then decide you wanted to discard that change?

**git checkout HEAD filename** will restore the file in your working directory to look exactly as it did when you last made a commit. Filename again is the actual name of the file.

Use **git diff** to see the difference between the file as it appears in the working directory vs how it appears in your last commit.

**$ git diff**

diff --git a/scene-5.txt b/scene-5.txt

index 5dd5d4e..c6b7caa 100644

--- a/scene-5.txt  
+++ b/scene-5.txt  
**@@ -14,5 +14,5 @@** Last line before the commit.

Lines that remain without changes from the last commit  
**- Lines that appeared in the last commit and now don’t  
+ New changes**\ No newline at end of file

It is needed to press **q** on the keyboard to restore the terminal.

Use the new Git command **$ git checkout HEAD filename** to restore the file in your working directory to look as it did when you last made a commit. Notice that the changes you made to the ghost’s line have been discarded.

# More git add

In Git, it’s common to change many files, add those files to the staging area with **git add filename\_1 filename\_2**, and commit them to a repository in a single commit. The files added to the staging area belong in the same commit. What if, before you commit, you accidentally delete an important line from another file?

Unthinkingly, you add another thirdFile to the staging area. The file change is unrelated with the changes of the other two files and you don’t want to include it in the commit. You can *unstage* that file from the staging area using: **git reset HEAD filename**

The command resets the file in the staging area to be the same as the HEAD commit. It does not discard file changes from the working directory, it just removes them from the staging area.

**$ git add thirdFile**

**$ git status**

On branch master

Changes to be committed:

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

**Modified files in the staging area**

In the output, notice the files under “Changes to be commited”

**$ git reset HEAD thirdFile**

Unstaged changes after reset:

M thirdFile

Notice in the output, "Unstaged changes after reset" M file(s)

M is short of “modification”

Now that changes made to the thirdFile have been booted out of the staging area, you’re ready to commit. From the terminal make a commit to save the prior changes.

**$ git status**

On branch master

Changes to be committed:

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

**modified: filename\_1  
 modified: filename\_2**

Changes not staged for commit:

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

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

**modified: thirdFile.txt**

**$ git commit -m "changes in filename\_1 & filename\_2"**

[master b5c2f40] changes in filename\_1 & filename\_2  
 2 files changed, 6 insertions(+), 6 deletions(-)

Creating a project is like hiking in a forest. Sometimes you rake a wrong turn and find yourself lost. Just like retracing your steps on that hike, Git enables you to rewind to the part before you made the wrong turn. You can do this with: **git reset commit\_SHA**

This command works by using the first 7 characters of the SHA of a previous commit. For example, use **git reset 5d69206** if the SHA of the previous commit is **5d692065cf51a2f50ea8e7b19b5a7ae512f633ba**

HEAD is now set to that previous commit.

**$ git log**

**commit b5c2f40e106bd854752c146f0667ad46e78ef39d**

Author:   
Date: Last commit

changes in filename\_1 & filename\_2

**commit d640dc3b22b3ee1d0a6e87477a35fd5e75c0712e**

Author:  
Date: Previous commit

Commit name

...Prior commits

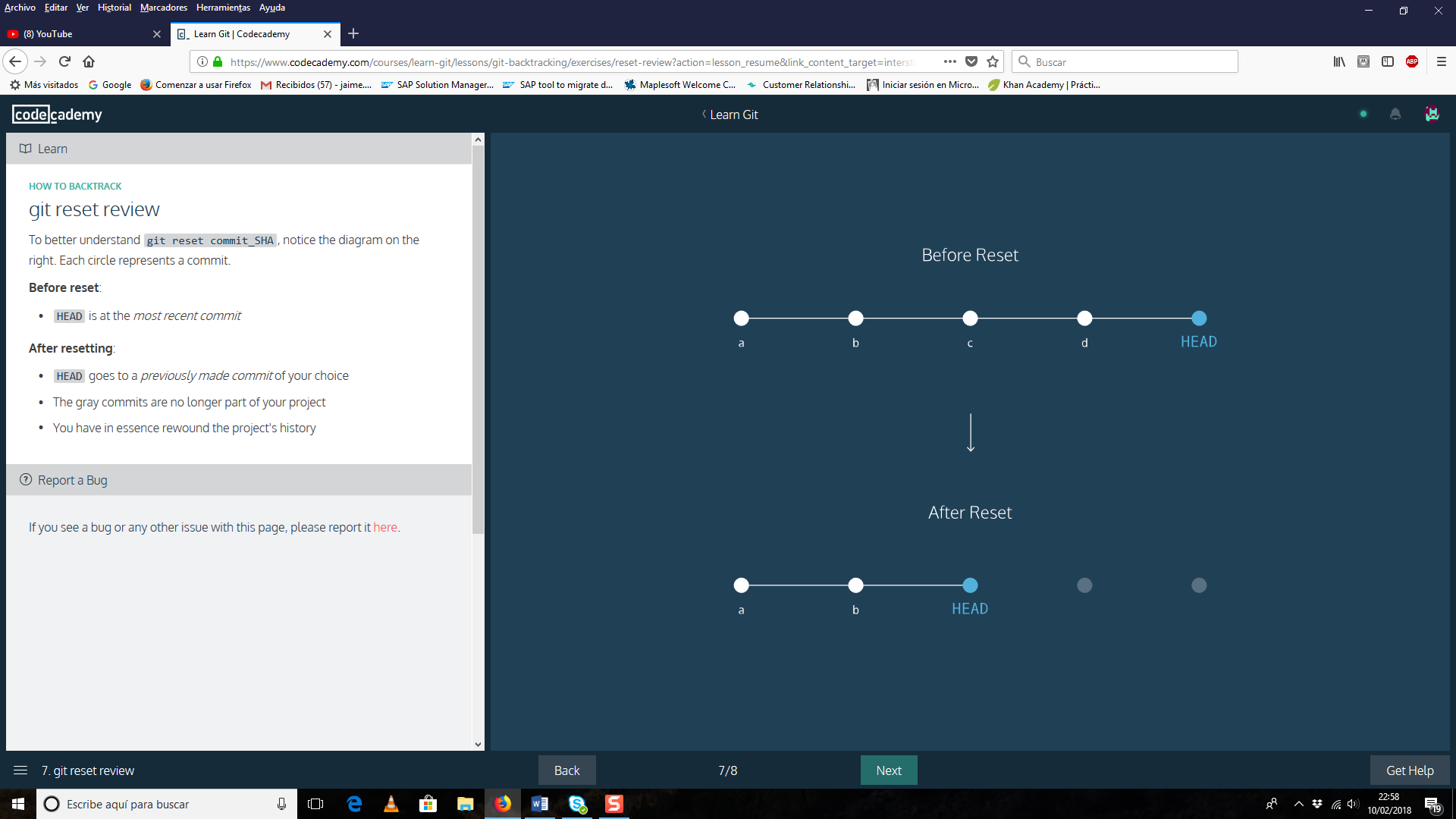
**Note**: If your cursor gets stuck in "git log" mode in the terminal, press "q" on your keyboard to escape

From the terminal, enter the command to reset to a previous commit, using the first 7 characters of one of the past commit SHAs in your Git log.

**$ git reset b5c2f40**

Unstaged changes after reset:

M scene-2.txt



To undo changes made to your Git project review the new commands:

**$ git checkout HEAD filename 🡪** Discards changes in the working directory.

**$ git reset HEAD filename 🡪** Unstages file changes in the staging area.

**$ git reset commit\_SHA** 🡪 Resets to a previous commit in your commit history.

Git *branching* allows users to experiment with different versions of a project by checking out separate *branches* to work on. The following commands are useful in the Git branch workflow:

* **git branch** Lists all a Git project’s branches
* **git branch branch\_name** Creates a new branch
* **git checkout branch\_name** Used to switch from one branch to another
* **git merge branch\_name** Used to join file changes from one branch to another
* **git branch –d branch\_name** Deletes the branch specified

# git branch

Up to this point, you’ve worked in a single Git branch called master. Git allows us to create *branches* to experiment with versions of a project. Imagine you want to create a version of a story with happy ending. You can create a new branch and make the happy ending changes to that branch only. It will have no effect on the master branch until you’re ready to merge the happy ending to the master branch. Branch names can’t contain whitespaces.

You can use the command below to answer the question: “which branch am I on? **git branch**

In the output, the \* (asterisk) is showing you what branch you’re on.

The circles are commits, and together form the Git project. It contains commits from Master but also has commits that Master does not have. All commits made in the master branch will be inherited in the new branches.

# git merge

What if you want to include all the changes made to a branch on the master branch? We can easily accomplish this by *merging* the branch into master with: **git merge branch\_name**

Master is the receiver branch, since it accepts the changes provided by the giver branch.

**$ git log**

**commit b5c2f40e106bd854752c146f0667ad46e78ef39d**

**...**

**$ git merge branch\_name**

Updating 79a1cc5..new first 7 characters

Fast-forward

resume.txt | 2 +-  
 1 file changed, 1 insertion(+), 1 deletion(-)

What would happen if you made a commit on master before you merged the two branches? Furthermore, what if the commit you made on master altered the same exact text you worked on in the other branch? When you switch back to master and ask Git to merge the two branches, Git doesn’t know which changes you want to keep. This is called a *merge conflict*.

If changes had been added to the staging area from the master and also from another branch, it will be a conflict when merging:

**Important:** Before merging, be sure that you are in the master branch!

**$ git checkout master**

Switched to branch 'master'

**$ git merge someBranch**

Auto-merging resume.txt

CONFLICT (content): Merge conflict in resume.txt  
Automatic merge failed; fix conflicts and then commit the result.

Git uses markings to indicate the HEAD (master) version of the file and the otherBranch version of the file, like this:

<<<<<<< HEAD

**master version of line**

=======

**anotherBranch version of line**

>>>>>>> anotherBranch

Delete **all of Git’s special markings** including the words HEAD and anotherBranch. If any of Git’s markings remain, for example, >>>>>> and =======, the conflict remains.

After solving the conflicts add the files to the staging area from the master branch, and then, make a commit with the commit message: “Resolve merge conflict” to indicate the purpose of the commit.

# Delete branch

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the master branch. After the branch has been integrated into master, it has served its purpose and can be deleted.

The command **git branch –d branch\_name** will delete the specified branch from your Git project.

Git offers a suite of collaboration tools to make working with others on a project easier:

* A complete replica of the project on each worker’s computer
* A way to keep track of and review each other’s work
* Access to a definitive project version

You can accomplish all of this by using *remotes*. A remote is a shared Git repository that allows multiple collaborators to work on the same Git project from different locations. Collaborators work on the project independently. And merge changes together when they are ready to do so.

# git clone

Sally has created the remote repository, **science-quizzes** in the directory **curriculum**, which teachers on the school’s shared network have access to. In order to get your own replica of **science-quizzes**, you’ll need to *clone* it.

**git clone remote\_location clone\_name**

**remote\_location 🡪** tells Git where to go to find the remote. This could be a web address, or a filepath, such as: /Users/teachers/Documents/some-remote

**clone\_name 🡪** is the name you give to the directory in which Git will clone the repository.

**$ git clone science-quizzes my-quizzes**

Cloning into 'my-quizzes'...

done.

Git informs us that it’s copying everthing from **science-quizzes** into the **m-quizzes** repository. **my-quizzes** is my *local* copy of **science-quizzes** Git project. If you commit changes to the project here, Sally will not know about them. One thing that Git does behind the scenes when you clone **science-quizzes** is give the remote address the name *origin*, so that ou can refer to it more conveniently. In this case, Sally’s remote is *origin*. You can see a list of a Git project’s remotes with the command: **git remote –v**

**$ git remote –v**

origin /home/ccuser/workspace/curriculum/science-quizzes (fetch)

origin /home/ccuser/workspace/curriculum/science-quizzes (push)

Git lists the name of the remote, ***origin***, as well as its location. The remote is listed twice: once for ***(fetch)*** and once for ***(push)***.

# git fetch

An easy way to see if changes have been made to the remote and bring the changes down to your local copy is with: **git fetch** This command will not *merge* changes from the remote into your local repository. It brings those changes onto what’s called a *remote branch*.

**$ cd my-quizzes**

**$ git fetch**

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (5/5), done.

remote: Total 5 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (5/5), done.

From /home/ccuser/workspace/curriculum-a/science-quizzes

\* [new branch] master -> origin/master

# git merge

Even though Sally’s new commits have been fetched to your local copy of the Git project, those commits are on the origin/master branch. Your *local* master branch has not been updated yet, so you can’t view or make changes to any of the work she has added. Now we’ll use the **git merge** command to integrate origin/master into your *local* master branch. The command **git merge origin/master** will accomplish this for us.

**$ git log**

**commit 2fd7d9b248e0b4a3b531b9af3bb61916d42ad45f**

Author:   
Date:

Add first question to physics quiz

...

**$ cd my-quizzes**

**$ git merge origin/master**

Updating 2fd7d9b..3a29454

Fast-forward

biology.txt | 4 ++++

1 file changed, 4 insertions(+)

create mode 100644 biology.txt

Git has performed a “fast-forward” merge, bringing your local master branch up to speed with Sally’s most recent commit on the remote.

**$ git log**

**commit 3a294546f4a55f02bf37233ef8988d8b9dd7ce59**

Author:   
Date:

Add heading and comment to biology quiz

**commit 6aa7704a31d05541141fbb529abf946bd2fd416b**

Author:   
Date:

Add biology quiz

**commit 2fd7d9b248e0b4a3b531b9af3bb61916d42ad45f**

Author:   
Date:

Add first question to physics quiz

After printing the commit history, the HEAD commit has changed.

# Git workflow

The workflow for Git collaborations typically follows this order:

1. Fetch and merge changes from the remote
2. Create a branch to work on a new project feature
3. Develop the feature on your branch and commit your work
4. Fetch and merge from the remote again (in case new commits were made while you were working)
5. *Push* your branch up to the remote for review

Steps 1 and 4 are a safeguard against *merge conflicts*, which occur when two branches contain file changes that cannot be merged with the **git merge** command. Step 5 involves **git push**

**$ cd my-quizzes**

**$ git branch bio-questions**

**$ git checkout bio-questions**

Switched to branch 'bio-questions'

* On your branch, open **biology.txt** file in the code editor and make the changes

**$ git add biology.txt**

**$ git commit -m "changes in biology.txt"**

[bio-questions eeed488] changes in biology.txt

1 file changed, 6 insertions(+)

# git push

The command **git push origin your\_branch\_name** will push your branch up to the remote, origin. From there, Sally can review your branch and merge your work into the master branch, making it part of the definitive project version.

**$ cd my-quizzes**

**$ git push origin bio-questions**

Counting objects: 3, done.

Delta compression using up to 16 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 394 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

To /home/ccuser/workspace/curriculum-a/science-quizzes

\* [new branch] bio-questions -> bio-questions

# Generalizations

A *remote* is a Git repository that lives *outside* your Git project folder. Remotes can live on the web, on a shared network or even in a separate folder on your local computer.

The *Git Collaborative Workflow* are steps that enable smooth project development when multiple collaborators are working on the same Git project.

* **git clone** Creates a local copy of a remote
* **git remove -v** Lists a Git project’s remotes
* **git fetch** Fetches work from the remote into the local copy
* **git merge origin/master <branch\_name>** Pushes a local branch to the origin remote
* **git push origin <branch\_name>** Pushes a local branch to the origin remote

Git projects are usuall managed on Github, a website that hosts Git projects for millions of users. With Github you can access your projects from anywhere in the world by using the basic workflow you learned here.