Installing Git

Download the installer for Windows from the [Git official site](https://git-scm.com/).

Execute the downloaded file. In the page Select Components you can leave the options at their defaults.

The Windows Explorer integration > Context menu entries option allows opening the Git command prompt (Git Bash) from any folder by clicking with the right mouse button on the folder and selecting Git Bash Here. The last option is also interesting in that it installs a better font for all console windows.

Note: Git for Windows comes with its own command prompt (Git Bash) that, besides git commands, has some useful Unix commands (and it looks better than the Windows default prompt).

On the next screen (Adjusting your path environment), I choose the most conservative option: Use Git Bash only. This will make git commands available only in Git Bash and will not alter your PATH variable. Click Next.

Another important setting: line endings. As you may know, Windows and Unix systems (Linux, Mac) have different formats of line breaks on text files. If you write a file with Windows line breaks, another person may have problems opening that file in Linux and vice-versa. The line endings setting allows you to normalize this.

I prefer choosing the second option (Checkout as-is, commit Unix-style line endings), that won't change the line breaks when the file arrives but will convert them to Unix-style when you commit. That way, you don't risk committing Windows-style line breaks and everything is kept in Unix-style. Don't worry, even though you are in Windows, most of the text editors can read Unix line breaks just fine.

After that one more Next, Finish, and Git is installed!

Creating a Local Repository

Let's test it. Create a folder, right-click, and choose Git Bash Here.

Before anything else, let's inform Git who you are so that your commits can be identified. Enter the commands, replace the quoted data with your real name and e-mail: (press Enter after each one).

1

2

git config --global user.name "Firstname Lastname"

git config --global user.email "your\_email@example.com"

bash

Now let's initialize a Git repository on this folder:

1

git init

shell

See that (master) on the command line? It tells you the current branch you are in a Git repository. The master branch is the main branch on every Git repo.

Now let's add a new file and commit it. Look at the command sequence (press Enter after each one):

code test.txt

git add .

git commit -m "First commit"

shell

First, we create an empty text file (you can create the file any way you like, not necessarily with the touch command). Then we add all new and modified files to the Git index (we tell Git which files we want to commit on the next commit). And finally we commit the changes with a message.

Sharing Your Code on GitHub

Cool! You have a Git repo in your machine but how about sharing your code on GitHub?

Initial Setup

If you don't have a GitHub account yet, go to [http://github.com](http://github.com/) and create one. It's free.

After you signup and login, let's add an SSH key so GitHub can link your account with this computer. That way it won't have to ask for your password on every commit.

On Git Bash enter the command:

1

ssh-keygen -t rsa -C "your\_email@example.com"

bash

Use the same email you registered at GitHub.

On the next question, press Enter to choose the default value.

Now it will ask for a password. Enter a password (this is NOT your GitHub password). When it asks for a confirmation, enter the password again. Now enter the command:

1

notepad ~/.ssh/id\_rsa.pub

bash

To open on Notepad the file that was created.

On GitHub, go to Settings and then SSH and GPG Keys. Click New SSH key. Enter a title to identify this computer and in the field Key paste all the contents of the file id\_rsa.pub.

Be careful to copy and paste all the contents of the file, beginning at "ssh-ras ..." up to your email (including it). Click Add SSH key.

Let's check if everything is ok. On Git Bash enter:

1

ssh -T git@github.com

bash

It will ask if you want to connect to a remote machine. Type yes and press Enter. Next, it will ask for a password. Enter the password you used on the ssh-keygen command.

If you see a message like:

1

Hi user! You've successfully authenticated, but GitHub does not provide shell access.

Then everything is correct!

Creating Your First Remote Repository

On GitHub, let's create a new repository (button New repository on your dashboard). Enter a name; it should not have spaces or special characters, as it will be part of the URL of your new repo. You can leave the rest of the options at their defaults.

You will be taken to the main page of your repository, that doesn't have any files yet.

On Git Bash (on the folder of your local repository) enter:

1

git remote add origin git@github.com:user/repo\_name.git

bash

Note that user/repo\_name must be entered the same way they appear in your repository URL, like:

<https://github.com/user/repo_name>

Now, to send your files to GitHub, enter:

1

git push origin master

bash

Inform the password of the SSH key if it asks.

Reload the page of your repo on GitHub and you should see your committed files.

Conclusion

Even though Git originated on Linux (did you know that it was created by Linus Torvalds, the same guy who created Linux?), developers on all system can benefit from it. Git is an excelleSnt SCM (source code management) system, widely adopted, and the open-source community on GitHub is vibrant! You can find code for pretty much anything you want, contribute with other developers and share your own solutions.

**Create a Git Repository and Branching Code**

This guide is not intended as a complete reference on Git, but as an overview of creating and working with repositories. As such, it assumes that the utility is installed on your system and that you are using a Unix-based operating system. If you are on Windows, Git commands are identical, and you will only need to modify some of the examples.

## Introducing Git

Up until the early 2000s, software developers used to share their work from person to person. As more individuals were involved in the same projects, this approach became time-consuming, error-prone, and far from effective. Version control systems, a set of software tools that help teams share files and track changes over time, were born to address those needs.

In 2002, the Linux kernel community was among the first to adopt a version control system known as BitKeeper. Git was later devised by Linus Torvalds and launched in May 2005 after an open and public conflict on licensing terms between these parties. Two months later, a Japanese software engineer named Junio Hamano was appointed as the maintainer, a role that he still fills to this day.

Before we dive any further, it is worthy and well to define some common terms we will encounter later:

* Web-based solutions, such as [GitHub](https://github.com/), [Bitbucket](https://bitbucket.org/product), or [GitLab](https://about.gitlab.com/), must not be confused with Git itself. These tools only provide space to store code in the cloud and a friendly interface to perform several operations. In this guide, we will use GitHub, but the process is very similar if you choose another solution.
* A repository is a folder that contains the files and subdirectories of a project. It can be either public or private, depending on who should have access (anyone or only members of a team, respectively).
* A branch is a separate development path (in the same repository) that is often used to work on new features without interfering with the main project. Once the code is reviewed and tested, a repository administrator can merge the changes into the master branch.
* A commit is a snapshot of a repository at a point in time. It allows users to include comments and ask for feedback from other people. Using its hash, one can easily return to a previous state of the project if needed. Before file and directories can be committed, we need to instruct Git to track them. We usually refer to this step simply as adding or staging files.
* A pull request is a method to inform other developers about and discuss recent changes before incorporating them into the main development path.
* A fork is an independent project which is based off a given repository. Unlike branches, it is not local to the latter, but it can also be merged into it through a proper pull request.
* A .gitignore file can be used to indicate which local content should not be committed to the repository. This is particularly useful to avoid pushing temporary files to or exposing sensitive information (such as passwords, SSH or API keys, and credit card numbers) in a web-based solution.

With that in mind, let us learn how to use Git and leverage GitHub for a software development project. Albeit simple, the following example will help us illustrate the basics of these utilities.

## Creating a Repository

To begin, we will need to create a separate directory and then initialize Git on it:

1

2

3

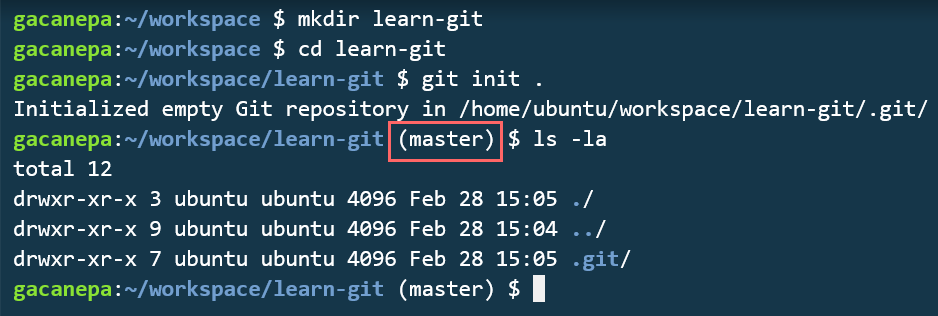
mkdir learn-git

cd learn-git

git init .

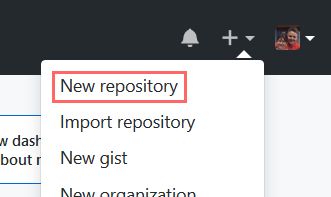
bash

As shown in Fig. 1, the last command automatically placed us in the master branch and generated a hidden subdirectory called .git. This folder contains all the files and directories that allow Git to manage the repository.

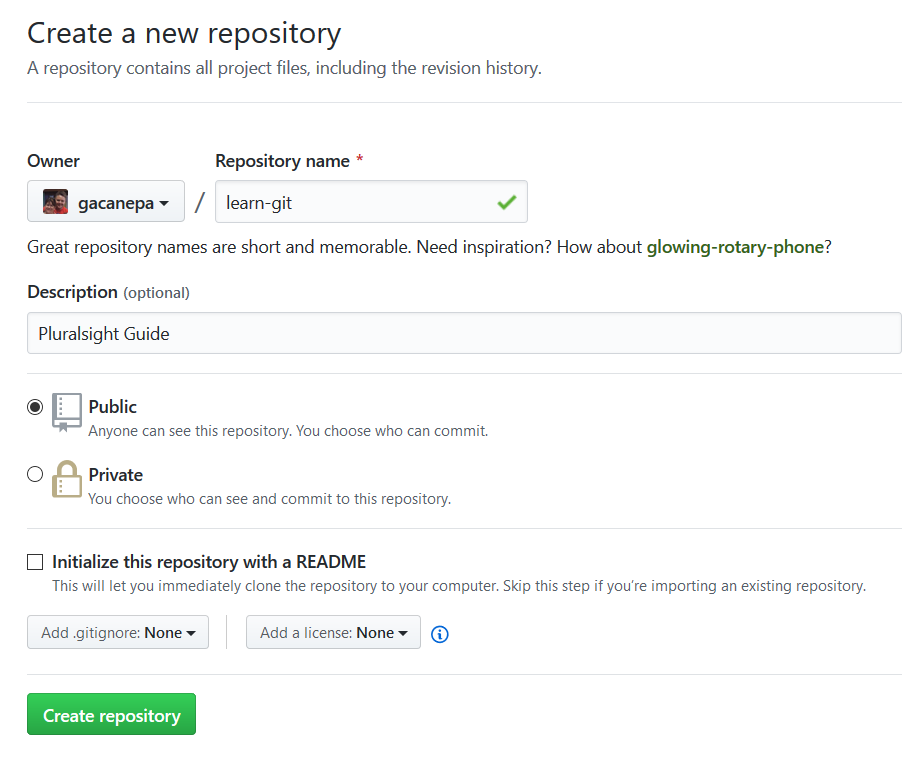


At this point, we have a local working repository but have not added or committed any files yet. Before we do that, we will follow these steps to create an empty remote repository in GitHub where we will push our code.

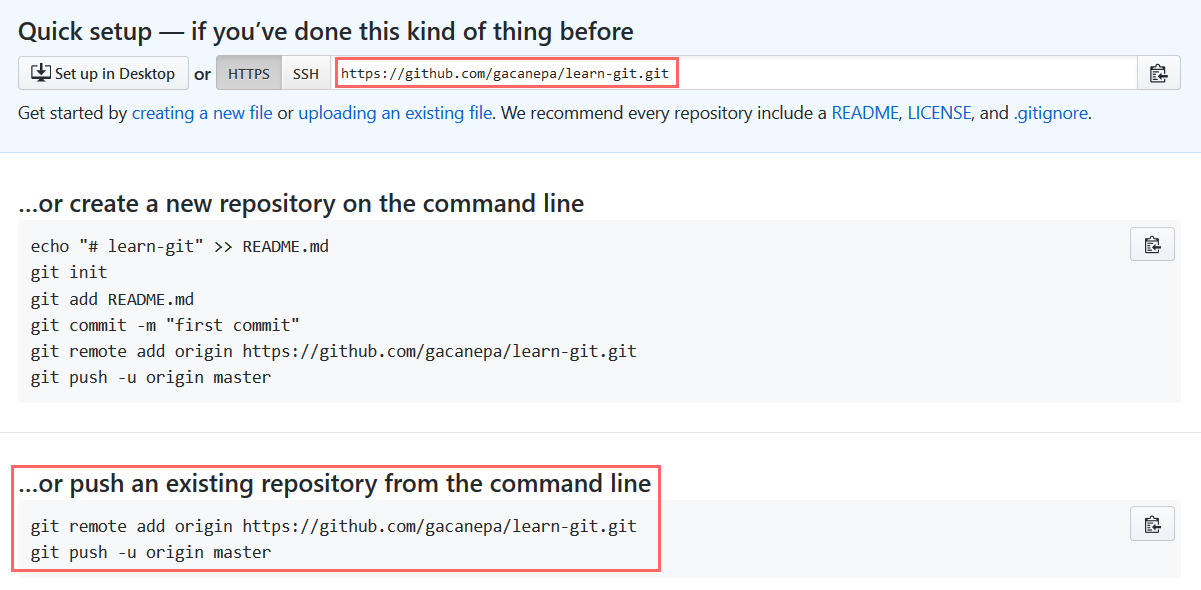
1. Click on the plus sign in the top right corner and choose New repository as seen in Fig. 2:



1. Enter a repository name, an optional description, and choose Public as illustrated in Fig. 3. Do not initialize the repository with a README or add a .gitignore file at this point, as we will do that later - along with the license. Finally, click on Create repository.



1. Copy the remote repository URL and take note of the suggested commands to push the local one over to GitHub through the command line but do not run them yet. Fig. 4 shows this step in our case.



We are now prepared to start adding files to our staging area and committing them.

## Our First Commit

For this test, we will create a short Python file named disk\_info.py with the following content:

1 import shutil

def disk\_percent\_usage(directory):

'''

Return percent usage of disk where directory resides

'''

disk\_info = shutil.disk\_usage(directory)

return round(disk\_info.used \* 100 / disk\_info.total, 2)

python

When this module is used, it is compiled to byte code in a file with the .pyc extension, usually in the same directory or in a subdirectory called \_\_pycache\_\_. In any event, we certainly want to include disk\_info.py in our repository but not the associated .pyc file - and that is where .gitignore enters the picture.

To avoid keeping track of byte code files, we can add a single line to .gitignore as follows:

1

echo "\*.pyc\*" >> .gitignore

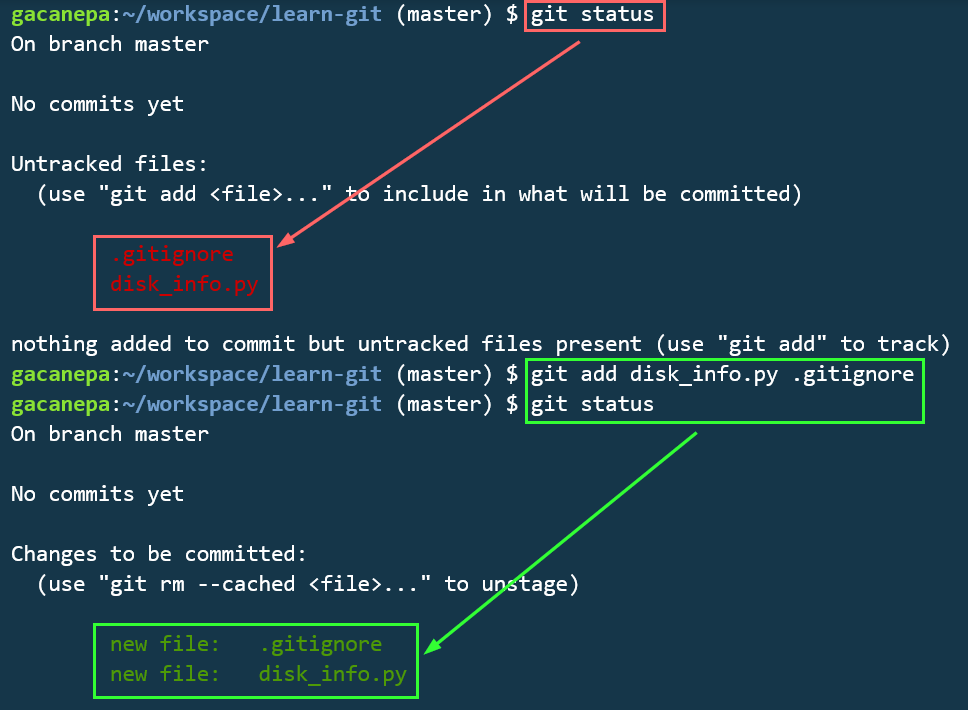
bash

With that, let us add both files (disk\_info.py and .gitignore) to the staging area. Fig. 5 shows the difference in the output of git status before and after this step.

1

git add disk\_info.py .gitignore

bash



The next step consists of committing the files to the repository. The -m option allows to include a message to describe the operation.

1

git commit -m "Initial commit for Pluralsight guide"

bash

*Until files are committed as explained above, they are not part of the repository although they reside in the same directory where it is initialized.*

Finally, push the local repository over to GitHub. Note that you will be prompted for your credentials as seen in Fig. 6:

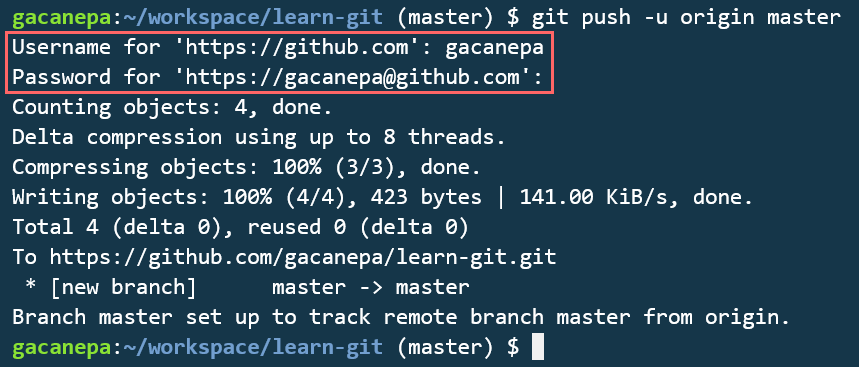
1

2

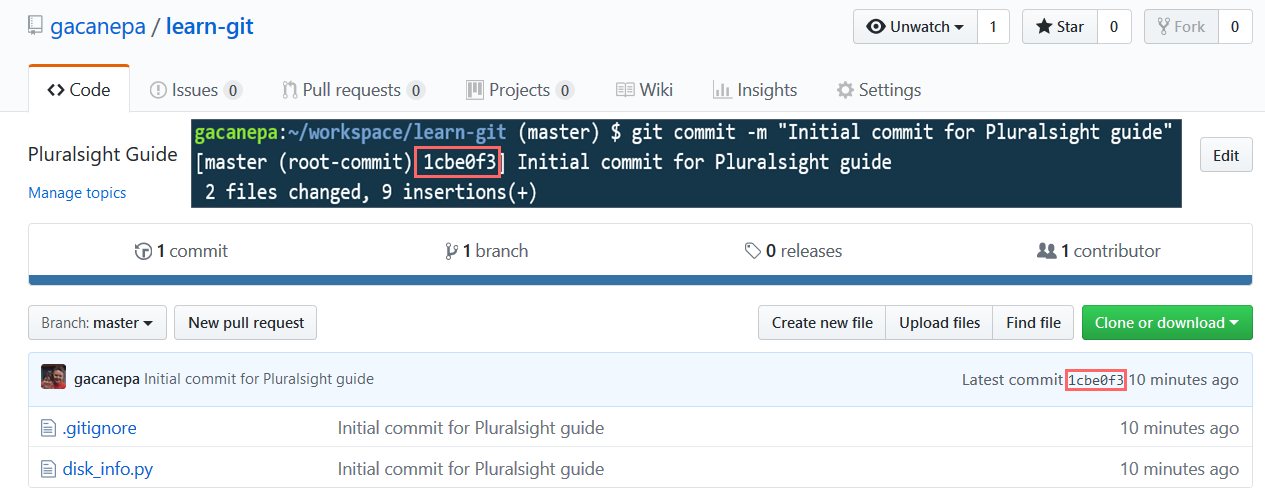
git remote add origin https://github.com/gacanepa/learn-git.git

git push -u origin master

bash



If we now browse to our GitHub repository, we should see the files as shown in Fig. 7. The commit hash is enclosed in a red rectangle and should be identical to the one that was returned by git commit earlier.



You can always view the changes that were introduced in a given commit via the URL for that commit. In the example above, it is <https://github.com/gacanepa/learn-git/commit/1cbe0f3>. Additionally, the full commit list (including messages, dates, hashes, and the user account who was responsible for each change) is available through git log.

## Branching Code

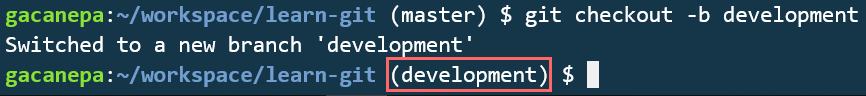
So far, we have only used the master branch, which is where our production-ready program resides. If we want to add a new feature, we should consider creating a separate branch called development, although you can choose another name if you wish, and make changes there to start.

1

git checkout -b development

bash

The above command not only created the branch but switched us to it, as you can see in Fig. 8:



We can now open our text editor and create a new file named system\_info.py with the following lines:

16 import platform

def kernel\_info():

'''

Return Linux kernel information

'''

kernel\_info = platform.uname()

return {'release': kernel\_info.release, 'version': kernel\_info.version}

python

Next, we will proceed to stage the file, commit the change, and push it to the remote repository.

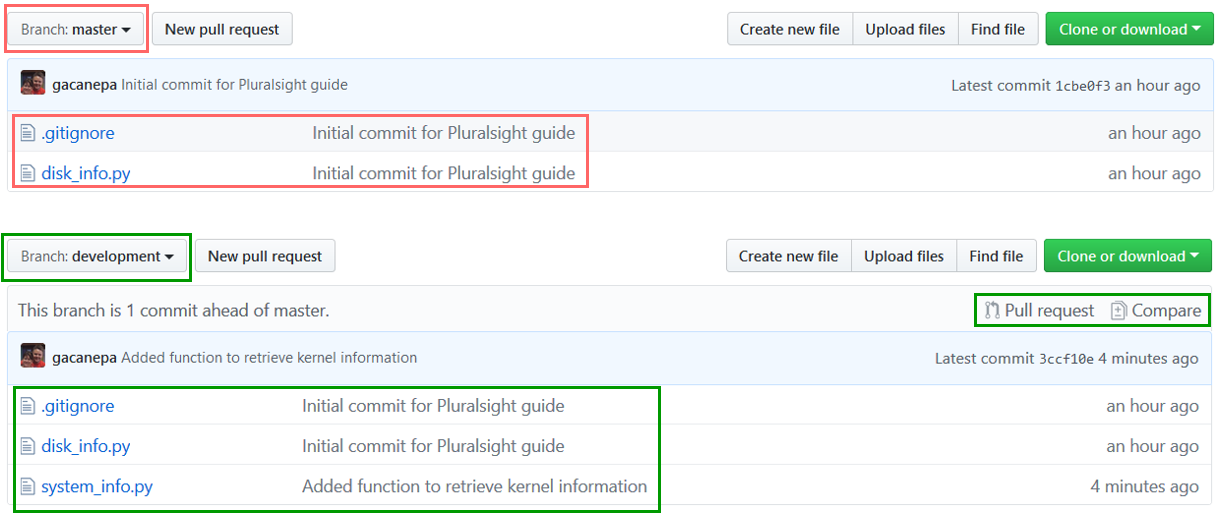
1 git add system\_info.py

git commit -m "Added function to retrieve kernel information"

git push -u origin development

bash

To switch between branches in GitHub, we can use the dropdown list as illustrated in Fig. 9. If we select development, we should see the file we just added. Additionally, we can compare branches and create a pull request to merge development into master, but we will leave that for later.



Once our code is in GitHub, other developers can clone our repository to start working on it on their local machines.

## Cloning a Repository

Before another developer can contribute to our repository, we need to add him or her as a collaborator in GitHub. To do so, we should go to Settings, click on Collaborators, and type the username or email address to send an invitation. Once our colleague accepts it, he or she will be able to:

1. Clone the repository:

1

git clone https://github.com/gacanepa/learn-git.git

bash

1. Switch to the development branch:

1

git checkout development

bash

1. Make a change. In this case, we will add a README file using Markdown format:

1

2

echo "# learn-git" > README.md

echo "Check out my guide at Pluralsight" >> README.md

bash

1. Stage the file:

1

git add README.md

bash

*If you need to add several files at once, you may want to stage the entire current directory with git add . instead.*

1. Make the commit:

1

git commit -m "Added README.md"

bash

1. Push the file:

1

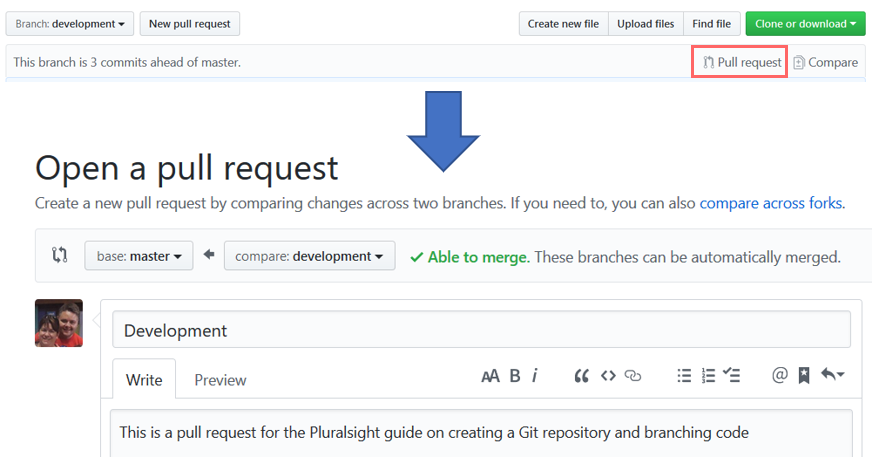
git push

bash

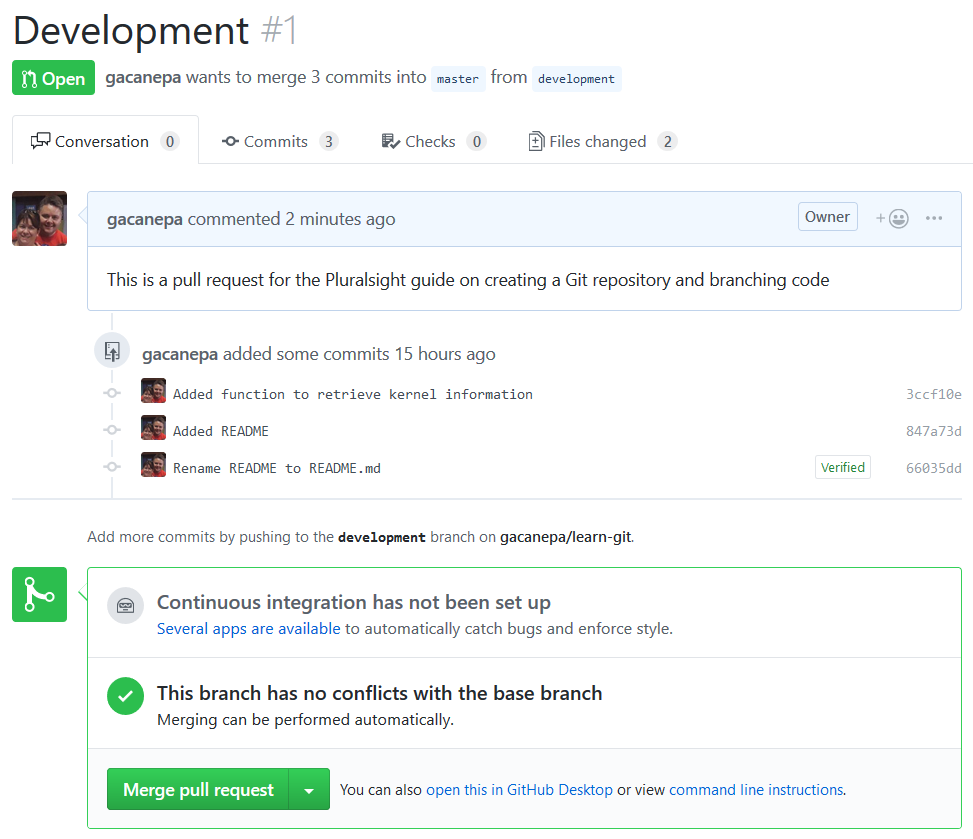
After making all these changes in the development branch, it is time to create a pull request for merging them into master.

## Creating a Pull Request

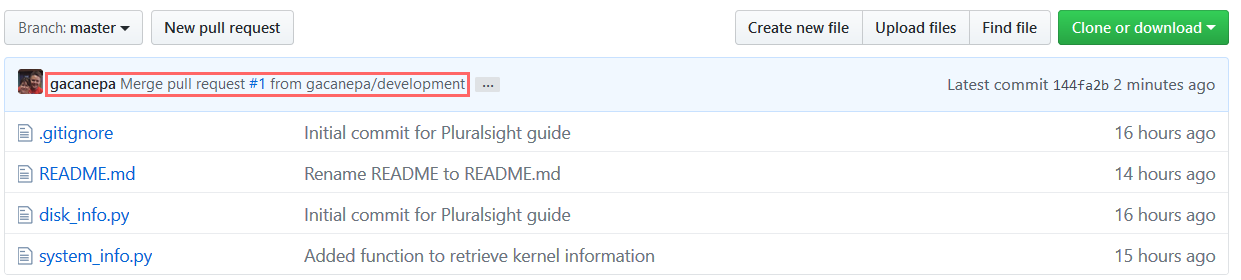
To incorporate the latest changes to the master branch, we will raise a pull request. Although this can also be done through the command line, it is much easier in GitHub. When you click on Pull request, you will be taken to a page where you can view the list of the commits in development to be included and add comments as you can see in Fig. 10. Next, click on Create pull request to continue:



Finally, go to Pull requests and click on Merge pull request as shown in Fig. 11. Note that you will still be asked to Confirm merge afterward:



At this point, the master branch should be up to date when compared to development as you can confirm in Fig. 12:



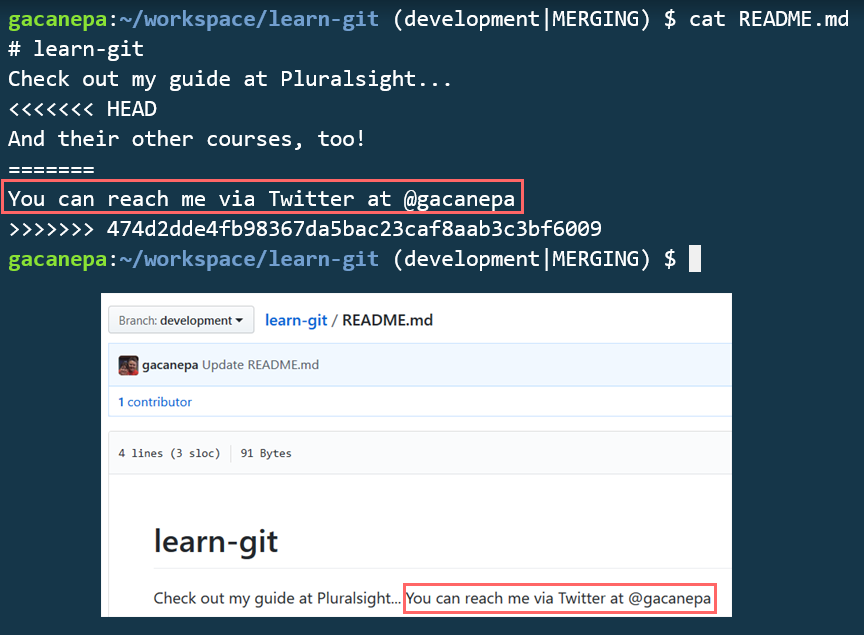
Although it was not the case in this example, Git also provides a method to resolve conflicts between commits in the same branch if needed, as we will see next.

## Resolving Merge Conflicts and Returning to Previous Commits

If a developer attempts to do a git pull after committing a file and there are discrepancies between his local repository and the remote one, a merge conflict occurs.

*Generally speaking, a merge conflict arises whenever more than one person edits one or more lines in a file at the same time.*

When this happens, an indication is added to the offending file(s) as you can see in Fig. 13. To resolve the conflict, you need to edit the file leaving the line(s) you want to keep, commit again, and push.



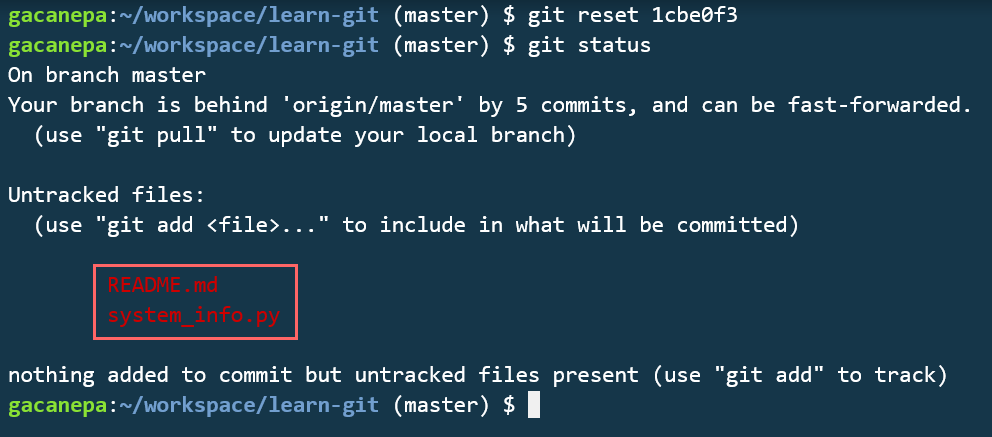
As we mentioned previously, Git also allows us to return to a previous commit if we wish. To do so, we can do git log to view the history and choose a specific snapshot using its hash. For example, to reset the status of master to what it was at commit 1cbe0f3, do

1

git reset 1cbe0f3

bash

This results in the changes that can be seen in Fig. 14:



Thus, README.md and system\_info.py are still present in the current directory but were removed from the staging area after we ran the previous command.

## Summary

In this guide, we have learned how to create a Git repository, work with branches, merge changes, and resolve conflicts. With these new skills, you will be able to work with other developers more effectively from here on out.

**Tips and Cool Tricks to Use in Git**

Introduction

This tutorial is designed not to teach Git fundamentals but to demonstrate some important strategies for using Git.

I will cover aliases, autocomplete, checkout/reset, stashes, amending, and differences between Git commits.

Alias (co > checkout)

You can create [alias](https://git-scm.com/book/tr/v2/Git-Basics-Git-Aliases) for any Git command. For example, the following creates an alias st for the Git command status:

1

$ git config --global alias.st status

shell

Just replace the alias with the name that you want, followed by the original command. With a shell alias of g and git alias of co, the following command will achieve the same effect as git checkout:

1

$ g co

shell

To create a shell alias you can add a one off alias to your current shell with alias g='git '

To create a "permanent"/persistent alias for your user you can simply add the above command to your user's shell rc file. This will be ~/.bashrc or ~/.zshrc for most systems.

With your favorite editor open the file for your current shell, you can find this shell by running echo $SHELL. Add the following snippet near the bottom of the file, the trailing space is so you can pass additional arguments to the alias and have them passed correctly to the program you can calling, in this case git.

1

alias g='git '

shell

It is a little different, but over time you earn enough in productivity.

Autocomplete

When using Git via command line, it is always useful to have autocomplete enabled. If you use the git shell by ruWindows, autocomplete should already be set.

If autocomplete is not enabled on your terminal, you can download the autocomplete script for Git from [Github](https://github.com/git/git/blob/master/contrib/completion/git-completion.bash).

After downloading, copy the file to the home directory, and add the line below to your *.bashrc* file:

1

source ~/git-completion.bash

shell

Now when you enter a git command and press tab, it should display all Git commands which start with *co*:

1

$ git co<tab>

shell

1

commit config

shell

Having this autocomplete ability at your fingertips speeds up implementation.

Checkout and reset on files

You may have used checkout and reset to work with branches and commits. However, you can also use the checkout and reset commands on files.

When we use git reset on a file in git, the reset command will update the staging area, causing a particular file to revert to its previous state.

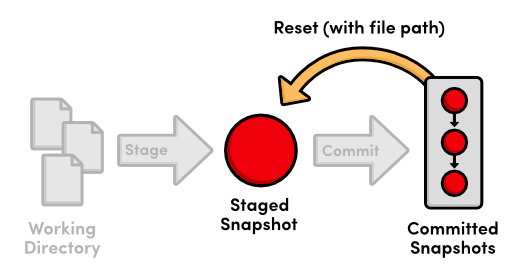
As a result, users can remove the file *"jedi.js"* from the staging area and make it equal to its version in *HEAD* with the command:

1

$ git reset HEAD jedi.js

shell

As seen in the diagram above, the file is no longer in staging and has the same content as the latest commit.



Pretty cool! Now let's take a look at checkout.

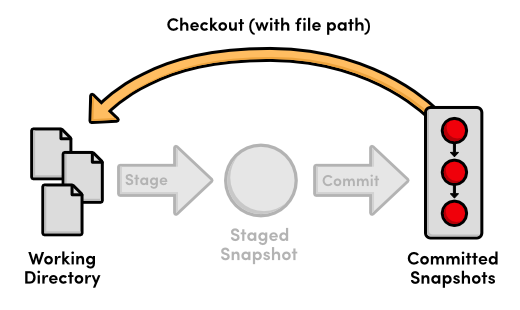
Good old git checkout is typically used to switch between branches. When the command is used with a file path, the specified file is reset to the same state as the branch (HEAD in this case).

1

$ git checkout HEAD jedi.js

shell

The above command restores the file "jedi.html" to the state of the file pointed to by *HEAD*, and it is removed from staging.



*Save it somewhere, really. This file will definitely come in handy.*

Thus, you can only restore the file to a version pointed to by another branch (or the same). You can make changes to the file then add this modified file to staging with git add ..

Stash saves time

The command git stash, picks all the changes in staging and save it in a separate place. Thus, this useful and aptly named command clears your staging area.

That way you can save a set of changes you have made but do not yet want to commit.

1

$ git stash

shell

Or, if you would need to stash untracked changes as well, use -u argument:

1

$ git stash -u

shell

In particular, I use more [stash](https://git-scm.com/book/pt-br/v1/Ferramentas-do-Git-Fazendo-Stash) when I need to git pull and want to avoid conflicts between local changes and changes upstream.

To restore your local changes back to staging, you need to *apply* your stash. The following command recreates the latest changes that were stashed away:

1

$ git stash apply

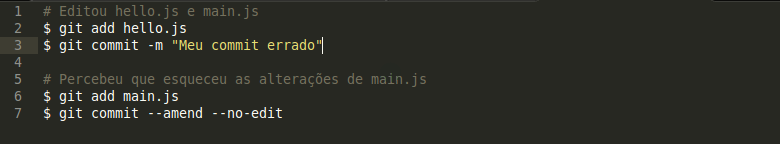
shell

You can have more than one stash. By default, your stashes will be applied in FILO (First-in, last-out) order.

Amend also saves time

If, after committing some changes, you remember that you forgot to add a modification to a file, you can use the [*--amend*](https://git-scm.com/book/pt-br/v1/Git-Essencial-Desfazendo-Coisas)option.

The following demonstrates a use case of amend, another important pro tool:



Here, the parameter *- no-edit*fixes the file without changing the commit message.

Differences between commits

To see the changes of the last commit, you can use:

1

$ git log --stat

shell

This command will show the files and the number of lines added and removed by file in each commit.

To see what exactly was changed in a commit, use [git diff](https://git-scm.com/docs/git-diff).

To see the difference between two commits using [sha](https://git-scm.com/book/en/v2/Git-Internals-Git-Objects)s of commits in hand (0da94be and 59ff30c), use:

1

$ git diff 0da94be 59ff30c

bash

If you use GitHub, you may better see the differences there. [+ see how](https://help.github.com/articles/comparing-commits-across-time/)