A basic GIT tutorial

Amir Omidvarnia

This tutorial demonstrates how to perform basic Git operations such as cloning a repository, checking the status, creating and switching branches, and pushing changes to a remote repository. It uses a dummy epository including a basic project structure and organization to illustrate these operations. The tutorial also covers more advanced topics such as merging branches, resolving merge conflicts, reverting commits, and using `.gitignore` and `.gitkeep` files.

Repository Structure

```
.
  README.md
  data
      sample_data.dat
  docs
      example_docs.docx
  empty-directory
  examples
      example1.ipynb
  scripts
      script1.py
  tests
       test_script1.py
```

Directories

docs

Contains project documentation.

- `example_docs.docx`: Example documentation files.

examples

Includes example code or usage scenarios.

- `example1.ipynb`: An exemplary jupyter notebook file.

scripts

Stores utility scripts and main project code.

- `script1.py`: An example Python script.

data

Houses data files used in the project.

- `sample_data.dat`: Example data for testing or demonstration.

tests

Contains test files and test suites.

- `test_script1.py`: Script for running tests.

Getting Started

Create and Navigate to a Directory on your local computer

```
$ mkdir Git_Tutorial
$ cd Git_Tutorial
```

Clone the repository

```
$ git clone https://github.com/omidvarnia/git_tutorial
Cloning into 'git_tutorial'...
remote: Enumerating objects: 33, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 33 (delta 6), reused 14 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (33/33), 42.19 KiB | 1.24 MiB/s, done.
Resolving deltas: 100% (6/6), done.
```

View Repository Contents

```
$ cd git_tutorial
$ tree
.
  README.md
  docs
      example_docs
  examples
      example1
  scripts
      script1.py
  data
      sample_data
  tests
       test_script

5 directories, 6 files
```

Check Repository Status

```
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main

$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Add and Commit a File

```
$ touch new_file.txt
$ git add .
$ git commit -m "Added a new file for demonstration purposes."
[main 6446be7] Added a new file for demonstration purposes.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 new_file.txt
```

Push Changes to Remote

```
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 312 bytes | 312.00 KiB/s, done.
To https://github.com/omidvarnia/git_tutorial
   952e06a..6446be7  main -> main
```

Delete and Commit File Changes

```
$ rm -f new_file.txt
$ git add .
$ git commit -m "Removed the new file."
[main 6c600dc] Removed the new file.
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 new_file.txt
```

Create and Switch Branches

```
$ git branch new_branch
$ git checkout new_branch
Switched to branch 'new_branch'
```

```
$ touch additional_file.txt
$ git add .
$ git commit -m "Created a new branch and added a new file."
[new_branch a30118d] Created a new branch and added a new file.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 additional_file.txt

$ git push origin new_branch
To https://github.com/omidvarnia/git_tutorial
 * [new branch]      new_branch -> new_branch
```

Return to Main Branch

```
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

Merge Branches

Switch to the branch you want to merge into (e.g., main).

```
$ git checkout main
```

Merge another branch into it (e.g., new_branch).

```
$ git merge new_branch
```

Resolve Merge Conflicts

Now, we make a simple scenario to demonstrate how to resolve merge conflicts.
Make a sample text file on both branches with some conflicting changes.

```
$ git checkout main
$ echo "This is a conflicting line." > conflicting_file.txt
$ git add .
$ git commit -m "Added conflicting file to main branch."
On branch main
Your branch is up to date with 'origin/main'.

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

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

```
$ git checkout new_branch
$ echo "This is a conflicting line." > conflicting_file.txt
```

```
$ git add .
$ git commit -m "Added conflicting file to new branch."
```

Now, try to merge `new_branch` into `main`.

```
$ git checkout main
$ git merge new_branch
Auto-merging conflicting_file.txt
CONFLICT (content): Merge conflict in conflicting_file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Then, you will be shown a text editor where you are asked to explain the conflict and commit the change.

```
Merge branch 'new_branch'
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
"~/GIT_repositories/git_tutorial/.git/MERGE_MSG" 6L, 252B
```

You can close this interface after adding your explanations and comments there by pressing `:q`. We have the same situation on the new branch as well.

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
```

To solve the problem, open conflicting files and resolve conflicts manually. After resolving, add and commit changes.

```
$ git add .
$ git commit -m "Resolved merge conflicts."
```

Finally, we need to **rebase**the changes, i.e., moving the sequence of commits to a new base commit.

```
$ git rebase --continue
```

Revert a Commit

```
$ git revert HEAD
```

Reset to a Previous Commit

Soft reset (preserves changes).

```
$ git reset --soft commit_hash
```

Hard reset (discards changes)—be careful!

```
$ git reset --hard commit_hash
```

Using `.gitignore`

The `.gitignore` file tells Git to ignore specified files and directories when making commits. This is useful for excluding specific file formats or directories from version control. Here's how to create and use it:

- Create a `.gitignore` file in the root directory of your project:

```
$ touch .gitignore
```

- Edit the file to specify which files or directories to ignore. For example:

```
# Ignore the data folder from being sznchronized
data

# Ignore all .txt files
*.txt

# Ignore .env files containing sensitive information
.env
```

Using `.gitkeep`

The `.gitkeep` file is an empty file placed in an otherwise empty directory to force Git to track that directory. This is useful for maintaining directory structures in your repository, even if the directories are initially empty. Here's how to use it:

- Create an empty directory you want Git to track:

```
$ mkdir empty-directory
```

- Add a `.gitkeep` file to the empty directory:

```
$ touch empty-directory/.gitkeep
```

- Add and commit the changes:

```
$ git add empty-directory/.gitkeep
$ git commit -m "Add empty directory with .gitkeep"
```