A Very Quick `git` Primer

Letting `git` know who you are:

```
git config --global user.name “Your Name”
git config --global user.email “your email address”
```

(Note: these work in or outside any repository and change your user info for every repository on that computer. In a specific repository, you can change your user info for just that repository by running the same commands without the `--global`.)

Cloning a repository:

```
git clone <clone URL>
```

(Note: If the repository you're cloning is hosted on GitHub, you can copy-paste the clone URL from the webpage associated with the repository – generally github.com/<username>/<name of repo>. If you have SSH keys set up, you can use the SSH clone URL; otherwise, use the HTTPS URL and type in your GitHub username and password when prompted.)

Committing:

```
git status
git add <filename>
git commit -m “Informative commit message here”
```

(Note: before you commit, it's always a good idea to see what files you've changed, whether they're already being tracked by git or not, and whether you'd previously added anything to the staging index. If you've only changed files that are already being tracked, you can skip the `git add` step and instead use `git commit -am “Commit message”`.)

Pulling and pushing:

```
git pull origin <branch name>
git push origin <branch name>
```

(Note: If you've pulled and pushed from this branch of this repository before, `git` most likely already knows where to pull from and push to, so you can just use `git push` and `git pull`. If you're familiar with rebasing, you can also use `git pull --rebase`. And remember – **always pull before you push!**)

Dealing with branches:

```
git checkout -b <new branch name>
git checkout <existing branch name>
git branch
```

(Note: `git branch` shows you all your existing local branches and what branch you're currently on. To switch which branch you're on, use `git checkout`.)

Merging:

```
git merge <child branch name>
```

(Note: You should be on the parent branch when you do this. Generally you're done with the child branch after you've merged it into the parent branch – since all of its changes are now in the parent branch – so you can delete the child branch with `git branch -d <child branch name>`, and work in a new branch for the next feature.)

Other useful features:

`git log`

(Shows a list of previous commits with authors, dates, and commit messages.)

`git revert <commit hash>`

(Undoes a specific commit by creating a new commit that exactly reverses the changes made in that commit. You can get the commit hash of the commit you want to revert from `git log`.)

`git checkout <commit hash>`

(Puts you back at what the code looked like immediately after a specific commit, e.g., to determine when a particular bug was introduced. If you want to make a new branch starting from this commit, you can do `git checkout -b <new branch name>` just as you usually would. To return to where you were before checking out the commit, just use `git checkout <previous branch name>`.)

`git cherry-pick <commit hash>`

(Applies the changes from a specific commit – usually from another branch – to the current branch.)

Advanced features:

`git rebase <parent branch name>`

(Note: You should be on the **child** branch when you do this. Incorporates all of parent branch's changes into child branch as if they happened before new changes in child branch. **This modifies your commit history and thus should be used with extreme caution!** Don't rebase in a branch that's been made public, as this might change history that's already in someone else's local copy. Note that `git pull --rebase` is generally safe even when working on a branch that's already public, because you're pulling by definition from the public history that everyone else has.)

`git rebase --interactive HEAD~<number>`

(Lets you modify the past <number> commits by hand – this can include changing commit messages, combining commits, removing commits, and more. **This modifies your commit history and thus should be used with extreme caution!** Most common use case: you commit, but **before pushing**, realize that you made some dumb error or left out a small change that doesn't really warrant its own commit. You can make the change, commit it, and then do `git rebase --interactive HEAD~2`, where you can choose to "squash" the latter commit into the former. You can then push as if those changes were always in the same commit. However, if you already pushed the first commit, you're out of luck and have to just push the small changes as a separate commit – **never change public history**.)

Note that all of these commands have additional options and other uses, plus there are many, many more features to explore. You can find more information about the above features plus many more details at git-scm.com/docs/.