GitSummary

I.Z.

September 29, 2018

Installation

Before starting to use Git you should first tell Git who you are so that you can be identified when you make contributions to projects. This can be done by specifying your name and email address.

In the shell, use

```
git config --global user.name "Your Name Comes Here"
git config --global user.email your_email@yourdomain.example.com
```

These two commands only need to be executed once. The information you provide is included in a `.gitconfig` file in your home directory and is used to label your actions in the Git log.

If you have spaces in the name you provide then be sure to enclose the name in <<<<<<< HEAD quotation marks. (You can check if you're set up correctly by running `git config --global --list`.) ======= quotation marks. >>>>>>> fedeb5447469deab4beecc73411763cf43937e02

Once you are in the project directory, you need to initialize your Git repository with the command git init so that Git knows that it needs to track changes here.

```
luke@nokomis ~/myproject% git init
Initialized empty Git repository in /home/luke/myproject/.git/
```

If you type `ls -a` you will see a directory named `.git` has been added. This directory stores all of the history information and other configuration data. Don't touch this directory.

Record changes

1. Visualize changes:

In the shell, use

```
git status
git diff
```

to see an overview of changes and to show detailed differences. In RStudio click on Diff. The background colours tells you whether the text has been added (green) or removed (red). (If you're colourblind you can use the line numbers in the two columns at the far left as a guide: a number in the first column identifies the old version, a number in second column identifies the new version.) The grey lines of code above and below the changes give you additional context.

2. Commits:

The fundamental unit of work in Git is a commit. A commit takes a snapshot of your code at a specified point in time. Using a Git commit is like using anchors and other protection when climbing. If you're crossing a dangerous rock face you want to make sure you've used protection to catch you if you fall. Commits play a similar role: if you make a mistake, you can't fall past the previous commit. Coding without commits is like

free-climbing: you can travel much faster in the short-term, but in the long-term the chances of catastrophic failure are high! Like rock climbing protection, you want to be judicious in your use of commits. Committing too frequently will slow your progress; use more commits when you're in uncertain or dangerous territory. Commits are also helpful to others, because they show your journey, not just the destination.

2.A. Creating commits:

- There are five key components to every commit:
  - A *unique identifier*, called a SHA (short for secure hash algorithm).
  - A *changeset* that describes which files were added, modified and deleted.
  - A human-readable *commit message*.
  - A *parent*, the commit that came before this one. (There are two exceptions to this rule: the initial commit doesn't have a parent, and merges, which you'll learn about later, have two parents.)
  - An *author*.

You create a commit in two stages:

1. You **stage** files, telling Git which changes should be included in the next commit.
2. You **commit** the staged files, describing the changes with a message.

In RStudio, staging and committing are done in the same place, the commit window, which you can open by pressing Ctrl + Alt + m.

The commit window is made up of three panes:

1. Top-left pane: current status as the Git pane in the main RStudio window
2. Bottom pane: shows the diff of the currently selected file (exactly the same window you see when clicking Diff)
3. Top-right pane: commit message

To create a new commit:

1. Save your changes.
2. Open the commit window by ressing Ctrl + Alt + m.
3. Select files: to stage (select) a single file for inclusion, tick its check box, to stage all files press Ctrl/Cmd + A, then click stage. As you stage each file, you'll notice that its status changes: the icon will change columns (within the "Status column") from right (unstaged) to left (staged), and you might see one of two new icons:
   - Added: after staging an untracked file, Git now knows that you want to add it to the repo.
   - Renamed: If you rename a file, Git initially sees it as a deletion and addition. Once you stage both changes, Git recognises that it's a rename.
   - Sometimes you'll see a status in both columns. This means that you have both staged and unstaged changes in the same file. This happens when you've made some changes, staged them, and then made some more. Clicking the staged checkbox will stage your new changes, clicking it again will unstage both sets of changes.
4. Stage files, as above.
5. Write a commit message (top-right panel) which describes the changes that you've made. The first line of a commit message is called the subject line and should be brief (50 characters or less). For complicated commits, you can follow it with a blank line and then a paragraph or bulleted list providing more detail. Write messages in imperative, like you're telling someone what to do: "fix this bug", not "fixed this bug" or "this bug was fixed".
6. Click Commit.

Staging files is a little more complicated in the shell. You use `git add` to stage new and modified files, and `git rm` to stage deleted files. To create the commit, use `git commit -m <message>`.

2.B. Commit best practices:

Ideally, each commit should be minimal but complete:

- **Minimal**: A commit should only contain changes related to a single problem. This will make it easier to understand the commit at a glance, and to describe it with a simple message. If you should discover a new problem, you should do a separate commit.
- **Complete**: A commit should solve the problem that it claims to solve. If you think you've fixed a bug, the commit should contain a unit test that confirms you're right.

Each commit message should:

- **Be concise, yet evocative**: At a glance, you should be able to see what a commit does. Yet there should be enough detail so you can remember (and understand) what was done.
- **Describe the why, not the what**: Since you can always retrieve the diff associated with a commit, the message doesn't need to say exactly what changed. Instead it should provide a high-level summary that focuses on the reasons for the change.

If you do this:

- It'll be easier to work with others. For example, if two people have changed the same file in the same place, it'll be easier to resolve conflicts if the commits are small and it's clear why each change was made.
- Project newcomers can more easily understand the history by reading the commit logs.
- You can load and run your package at any point along its development history. This can be tremendously useful with tools like bisectr, which allow you to use binary search to quickly find the commit that introduced a bug.
- If you can figure out exactly when a bug was introduced, you can easily understand what you were doing (and why!).

You might think that because no one else will ever look at your repo, that writing good commit messages is not worth the effort. But keep in mind that you have one very important collaborator: future-you! If you spend a little time now polishing your commit messages, future-you will thank you if and when they need to do a post-mortem on a bug.

Remember that these directives are aspirational. You shouldn't let them get in your way. If you look at the commit history of my repositories, you'll notice a lot of them aren't that good, especially when I start to get frustrated that I still haven't managed to fix a bug. Strive to follow these guidelines, and remember it's better to have multiple bad commits than to have one perfect commit.

2.C. Ignoring files

Often, there are files that you don't want to include in the repository. They might be transient (like LaTeX or C build artefacts), very large, or generated on demand. Rather than carefully not staging them each time, you should instead add them to `.gitignore`. This will prevent them from accidentally being added. The easiest way to do this is to right-click on the file in the Git pane and select Ignore. If you want to ignore multiple files, you can use a wildcard "glob" like *.png.

Some developers never commit derived files, files that can be generated automatically. For an R package this would mean ignoring the files in the NAMESPACE and man/ directories because they're generated from comments. From a practical pespective, it's better to commit these files: R packages have no way to generate .Rd files on installation so ignoring derived files means that users who install your package from GitHub will have no documentation.

Details about how git works

1. Pull

You use push to send your changes to GitHub. If you're working with others, they also push their changes to GitHub. But, to see their changes locally you'll need to pull their changes from GitHub. In fact, to make sure everyone is in sync, Git will only let you push to a repo if you've retrieved the most recent version with a pull.

When you pull, Git first downloads (fetches) all of the changes and then merges them with the changes that you've made. In particular, `git pull` does:

1. `git fetch origin master` to update the local `origin/master` branch with the latest commits from GitHub.
2. `git merge origin/master` to combine the remote changes with your changes

Working with others

1. Branching

Sometimes you want to make big changes to your code without having to disturb your main stream of development. Maybe you want to break it up into multiple simple commits so you can easily track what you're doing. Maybe you#re not sure what you've done is the best approach and you want someone else to review your code. Or, maybe you want to try something experimental (you can merge it back only if the experiment succeeds). Branches and pull requests provide powerful tools to handle these situations.

Although you haven't realised it, you're already using branches. The default branch is called **master**; it's where you've been saving your commits. If you synchronise your code to GitHub you'll also have a branch called **origin/master**: it's a local copy of all the commits on GitHub, which gets synchronised when you pull.

It's useful to create your own branches when you want to (temporarily) break away from the main stream of development. You can create a new branch with

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

Names should be in lower case letters and numbers, with - to separate words.

Switch between branches with `git checkout`, e.g. to return to the main line of development use

```
git checkout master.
```

You can also use the branch switcher at the top right of the Git pane.

If you've forgotten the name of your branch in the shell, you can use

```
git branch
* master
  test
```

to list all existing branches. The asterisk indicates you are on the master branch, so you can do the merge.

If you try to synchronise this branch to GitHub from inside RStudio, you'll notice that push and pull are disabled. To enable them, you'll need to first tell Git that your local branch has a remote equivalent:

```
git push --set-upstream origin <branch-name>
```

After you've done that once, you can use the pull and push buttons as usual.

If you've been working on a branch for a while, other work might have been going on in the master branch. To integrate that work into your branch, run

```
git merge master
```

You will need to resolve any merge conflicts (see below). It's best to do this fairly frequently - the less your branch diverges from the master, the easier it will be to merge. Once you're done working on a branch, merge it back into the master, then delete the branch:

```
git checkout master
git merge <branch-name>
git branch -d <branch-name>
```

(Git won't let you delete a branch unless you've merged it back into the master branch. If you do want to abandon a branch without merging it, you'll need to force delete with -D instead of -d. If you accidentally delete a branch, don't panic. It's usually possible to get it back. See the advice about undoing mistakes).

After a while, suppose you feel the work done on some `test` branch is in fact useful and you want to merge that into your master branch. Again, you can call `git merge` to merge two branches together. You need to do this from a checkout of the `master` branch:

```
luke@nokomis ~/myproject% git merge test
Merge made by recursive.
 doc.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 doc.txt
```

Running `git log` gives us the history of the two branches merged together with a *merge commit*. Because we merged the test branch into the master branch we no longer need it. A branch can be deleted with the `-d` switch to `git branch`. Be careful when deleting branches; if you have not merged that branch into the "master"" then deleting a branch will lose all of the history associated with that branch.

A useful tool for viewing the branch structure of a Git archive is `gitk`. Running `gitk` with the `--all` switch indicates that gitk should show all branches.

2. Merging

When you pull, Git first downloads (fetches) all of the changes and then merges them with the changes that you've made. A merge is a commit with two parents. It takes two different lines of development and combines them into a single result. In many cases, Git can do this automatically: for example, when changes are made to different files, or to different parts of the same file. However, if changes are made to the same place in a file, you'll need to resolve the merge conflict yourself.