Using Github

Asher Spector

February 10, 2018

Dev note: I recognize this is inefficiently written. When I finish writing out all the content, I will ruthlessly cut it down, don’t worry :)

Note to self: add git status in

This guide is a basic introduction to Git and Github for near-absolute beginners. It will start by discussing the importance of git and github, how to install git, and how to use git/github at a basic level. Lastly, it will provide links to further reading regarding advanced functionality for git.

1. Why use git/github

Git and github are actually two different things, and deserve their own sections. Before diving into specifics, however, it’s important to recognize that git and github are just tools that developers use to more effectively organize their code. Git and github are designed to solve three main organizational problems.

First, imagine that two programmers (we’ll refer to them as Carlos and Grace) are working together to try to build a website that automatically displays polling results from US elections, and suppose Grace modifies a script to slightly change the method for aggregating polling data. It would be tedious for Carlos and Grace to email the script back and forth each time they updated it. Instead, they need a repository where they can easily update and sync their code to make sure they’re on the same page. Github allows programmers to create such repositories.

Second, imagine that farther along in the project, Grace decides that she needs to totally rewrite the code which stores and organizes polling data. Unfortunately, while Grace modifies the system which stores the data, it will be offline, which is unfortunate because Carlos needs to use the data for his work on the project. To avoid this dilemma, Git and Github allow programmers to create different versions, called branches, of their code. Using branching, Grace can modify the storage system, and in the meantime, Carlos can continue to use the old storage system for his work.

Third, suppose that once Grace and Carlos have finished their website, they want to publish all the statistical functions and tools they built so that other researchers can use those tools to reproduce Grace and Carlos’s research. It’s often tedious to write documentation about huge quantities of code, but thankfully, Git and Github offer tools that allow programmers to more easily document their code.

What is git?

Git is a local version control system. In other words, Git automatically organizes and saves versions of code on your computer, but does not connect to the internet: it’s purely accessible through the command prompt on a computer (there are also other ways to use git, but it’s generally best to use the command prompt). Git organizes code in two ways.

First, Git sends discrete versions of code to a local database or repository on the computer on which it is installed. To understand this, imagine that Carlos is writing an R script which determines the proper weights for aggregating polling data, and at some point he finishes the first version of the script. At this point, Carlos can send his code to the local repository, in an action called *commiting* his script. We’ll discuss how to do this later, but note that scripts are not automatically committed to the repository. If Carlos deleted all the code in the script and saved it, he could still access the script by going into the repository. Moreover, the repository also stores all of the previous versions of code, and can backtrack through the various commits, so if Carlos decides the version of the script he built last year worked better, he can use git to access it.

Second, Git can create branches, or separate versions of code, on the computer it's installed to. Remember how Grace wanted to revamp the storage system for polling data without preventing Carlos from accessing the old storage system? Using Git, Grace should create a new *branch* in the repository. Creating a branch creates (for all practical purposes) an entirely separate version of the entire polling project in the Git repository, which Grace can modify and experiment with without compromising the functionality of the *master branch,* the original scripts that Grace had written. While she's working, Grace can use Git to move back and forth between the branches: doing so will make the contents of her R scripts (which show up in the same folder on her computer) seem to magically change.

__ insert graphic demonstrating this property of branching __

Then, when Grace finishes designing and testing the new storage system, she can *merge* the two branches together, and Carlos's scripts will automatically start using the new storage system.

(Insert graphic about organization of git, including branches,)

What is github?

Where Git creates *local* repositories on an individual computer, Github allows programmers to create *remote* repositories, stored online, for their code. This is pretty simple concept, but it's extroardinarily useful. It means that when Grace finishes updating a series of scripts, she can simply *push* (push means send) them to Github, and then Carlos can in one command *pull* (download) the modified versions onto his computer to work with them. Github provides other important tools, but we will not discuss those until later.

__ insert graphic about github, git, etc__

2. Setting up Git/Github

__ general problem - what about other OSs? __

Making a Github account

Signing up for a Github is fairly simple - just go to github.com, click "sign up" in the upper right hand corner, and follow the instructions. We recommend, at least to begin with, getting a free version of Github. (Note: ask Simo - we recommend this right?)

How to Install and Set Up Git

Because Git is a form of *local* source control, it does require installation. To install Git onto your computer, go to one of the following sites:

- For Linux: Head to https://git-scm.com/download/linux (https://git-scm.com/download/linux) and follow the instructions.
- For Windows: Go to https://git-scm.com/download/win (https://git-scm.com/download/win) and the download should automatically start.
- For Mac: Click on https://git-scm.com/download/mac (https://git-scm.com/download/mac) and the download should automatically start.

If you are running Windows/Mac and the websites above download an installer, simply follow the instructions from the installer - it's fine to just use default settings for now.

Before you can start using git, you do have to do a one time set up. Git is built to track and organize different versions of code, so it will need you to set your username and email before it will let you start modifying programs. To set your username and email after installing, open the command line on your computer and type the following commands into it:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@email.com"
```

__ insert screenshot of git bash/cmd line __

Now you're ready to start using Git/Github!

3. Starting to use Git/Github

Creating Projects and Repositories

The first step to using git is creating a project, or repository (repo for short). There are generally two ways to create a repo in Git.

Method 1: Start your project on Github

The first way to start a repo is by creating it on github. To do so, log into your Github account, click "Repositories" in the top tab, and then click the green "New" button on the right.

__ do we want to insert pictures? __

Github will prompt you to choose a repository name and description of the repo, which you should enter, and will also ask you whether you'd like to initialize the repository with a README. You should generally add README and .gitignore files, although we'll discuss their functions and importance later. For now, just add the README.

__ discussion of gitignores?? __

Once you've created the repository on Github, the next step is to link it with a local repository on git. To do this, open the *command prompt* on your computer. You'll need to enter a couple of commands to get the repo set up on your computer.

First, you'll need to set the *working directory* in the command prompt. The working directory is the location where git and command prompt will do all of their editing - if you add a file, it will add the file into your working directory. Thus, before you download your repository, you'll want to navigate to the folder in which you want the repository to be located. For example, Carlos might want to put a repo in his "R" folder inside the "Documents" folder on my computer. To do this, you can use the 'cd' command, which stands for "change directory," and then copy and paste the path of the folder you'd like to place the repository in. An example is given below:

```
$ cd C:/Users/Carlos/Documents/R
```

Next, you will need to tell git to download, or *clone*, the repo. You can do this using the "git clone" command, which allows you to download and sync a repository *for the first time.* This will download the repo, with all its contents, onto your computer and automatically set up a local version of the repo. See an example of the command below:

```
$ git clone https://github.com/Carlos_Account/repository_name/
```

Method 2: Git init

You can also initialize repos through Git and then upload them to github. To do this, first create a folder where you'd like the repo to be located. Then, open the command line, and like before, navigate to the folder using the 'cd' command. Once you've navigated there, you can initialize the repo locally by typing "git init""

```
$ cd C:/Users/Carlos/Documents/R/repository_name
$ git init
```

This sets up a local repository! We'll go over how to link a local repository set up this way to github in the next section.

Modifying Projects

Say you have created some code in your project directory and want to add it to the repository. To do this, you need to use three commands: *git add*, *git commit*, and *git push.* Note that sending scripts to the local repository is (perhaps counterintuitively) known as *commiting* them, not adding them.

First, you need to *add* scripts to Git (adding scripts doesn't add them to the repository - rather, it just lets Git know that you will eventually *commit* or send them to the repository). To do this, navigate to the correct working directory and type 'git add [your script's filepath]' to the directory. You can add multiple files in a row with no problem.

Second, you need to *commit* scripts, which means actually sending them to the local repository. To do this, simply type *git commit* to the command line after you've added all the scripts you want to send to the repository. Git will open a text editor and prompt you to add a message to explain what the commit is for - type in your desired message, then hit "escape" to leave the text editor, and type ":wq" to save and exit the message interface. Once you've committed files, all of the files you've previously added will be saved on your local repository.

```
$ cd C:/Users/Carlos/Documents/R/repository_name
$ git add 'scriptname.R'
$ git add 'scriptname2.R'
$ git commit
```

Lastly, you need to *push* the files, which means sending them to Github online. Before you do this, if you created the repository locally and have not already done so, you'll need to specify to Git the url to send the commits to. To do this, type 'git remote add origin [your url]' into your command line. Then, you can *push* the file by writing 'git push -u origin master,' which will send the file to the Github repo.

```
$ git remote add origin https://github.com/user_name/repository_name
$ git push -u origin master
```

__ git commit -m or git commit? Depends on the command line/bash distinction __

Accessing older versions of code

You should never be scared of git/github, because git/github only add information - they almost never delete it. In practice, this means that if you accidentally (or intentionally) commit horrible changes to your code, you can easily revert to a previous version. For example, let's say Grace wrote some new functions for a script, deleted them (and committed the deltes) later on, but nowwants the functions back. She can easily access the older (nonempty) script using git. First, she should use the "git log" command to see what commits have been made recently.

```
$ git log
```

This command might give an output like this:

```
commit 06c347sa (HEAD -> master, origin/master, origin/HEAD)
Author: Grace Smith <gracesmith@gmail.com>
Date:   Sun Feb 18 23:13:32 2018 -0500

    Remove storage functions

commit b69a14b
Author: Grace Smith <gracesmith@gmail.com>
Date:   Sat Feb 17 23:11:28 2018 -0500

    Add storage functions

commit 8a237d1
Author: Grace Smith <gracesmith@gmail.com>
Date:   Fri Feb 16 21:28:46 2018 -0500

    Initial commit
```

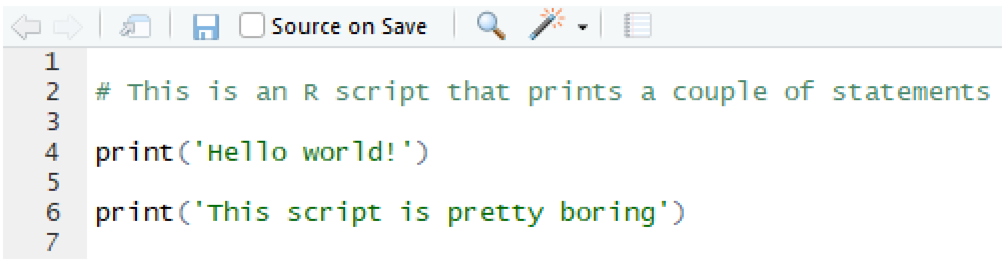
Clearly, Grace should restore the second commit listed, named 'b69a14b,' in order to access the functions she deleted. To do so, Grace can use the "git reset" command.

```
$ git reset --soft b69a14b
```

This will update the current version of the script to the b69a14b version, which Grace can then inspect and test until she's ready to commit it to git/github. Note that there is alos a "git reset –hard" option, which works the same way, but is a pretty serious action to take because it will effectively delete all the commits prior to the version being restored. The soft option is usually a better way to go.

Branching

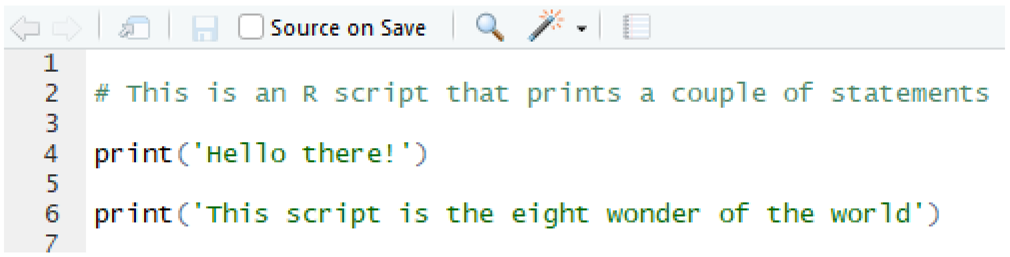
As we discussed earlier, branches allow you to modify scripts while simaltaneous keeping the old versions easily accessible. In practice, the way this works is that the branch you select in git changes the way that scripts you open using explorer/finder appear. For example, if Carlos has the following script as the master (default) branch in Git, the script might look something like this when he opens it on his computer:



```

# This is an R script that prints a couple of statements

print('Hello world!')

print('This script is pretty boring')

```

Version 1 of the Hello World Script

However, if he uses Git to switch to a test branch, he might open the exact same file using his file explorer and see the following:



```
# This is an R script that prints a couple of statements

print('Hello there!')

print('This script is the eight wonder of the world')
```

Version 2 of the Hello World Script

Let's discuss how to actually use git to branch.

Creating and Switching Branches

To create a branch, use the "git checkout" command followed with a "-b" and the name of the branch:

```
$ git checkout -b branch_name
```

This will create a new branch which is identical to the initial (master) branch. You can modify it as you like, and you will still be able to easily access the initial (master) branch. To do this, simply use the "git checkout" command without the "-b" and type the name of the branch you want to switch to:

```
$ git checkout switch_to_this_branch
```

Remember, switching to a branch will change the way the file shows up once you open it from your computer.

Merging Branches

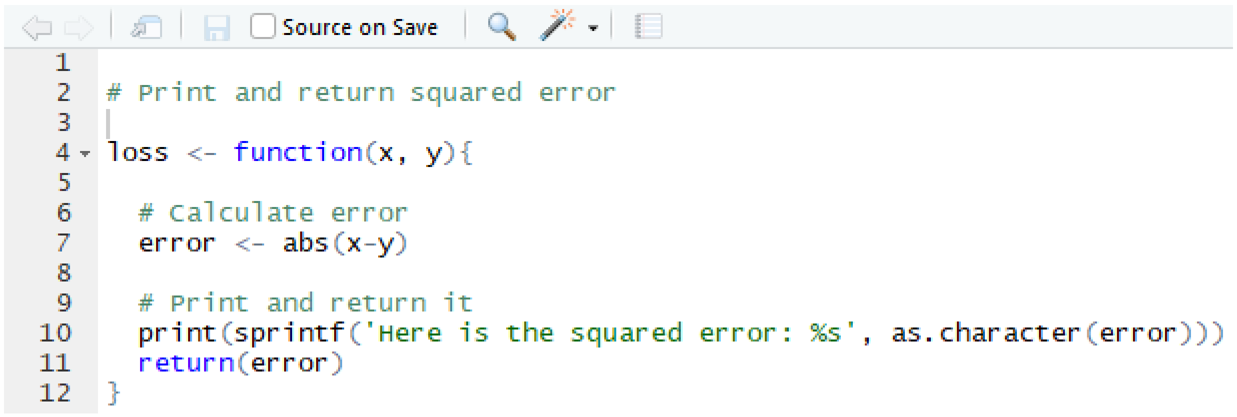
Suppose you have created a test branch, tested it to make sure it works, and now want to combine it with the original master branch. To do this, use the "git checkout" command in combination with the "git merge" command as follows:

```
$ git checkout master
$ git merge test
```

These two commands will merge the test branch into the master branch, meaning that the master branch at the end of the merging will look like the test branch.

Conflicts

Sometimes, however, Git will be unable to push or pull branches because it is getting conflicting information from two users. For example, suppose Grace has written the following function, which calculates the squared error between two values:



```
# Print and return squared error

loss <- function(x, y){

  # Calculate error
  error <- abs(x-y)

  # Print and return it
  print(sprintf('Here is the squared error: %s', as.character(error)))
  return(error)
}
```

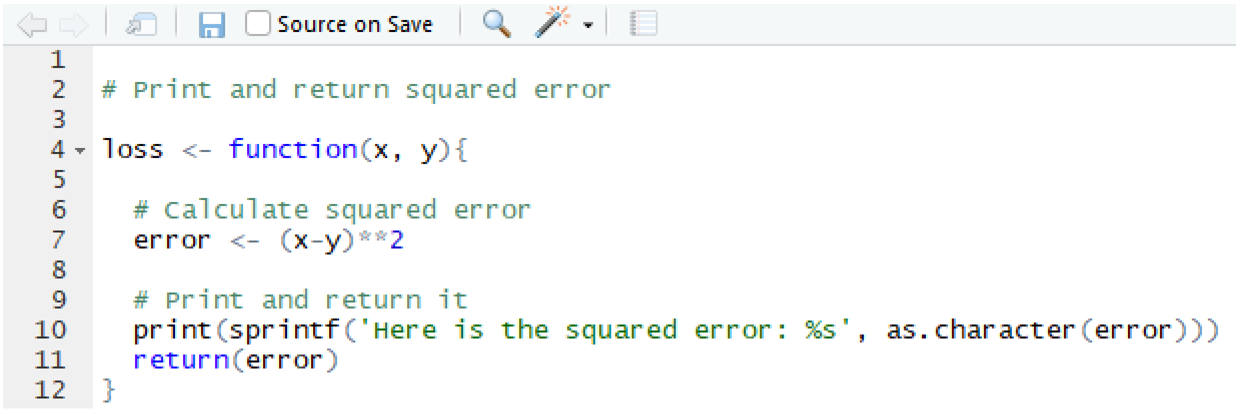
Original Version of Loss Script

This function clearly has a bug in it, because it claims to return the squared error, but instead, it returns the absolute value of the error. Carlos notices this, and modifies the function so that it accurately reports that it calculates the absolute value, and pushes his modified version to Github:

```
 1
 2  # Print and return abs value error
 3
 4  loss <- function(x, y){
 5
 6    # Calculate absolute value of error
 7    error <- abs(x-y)
 8
 9    # Print and return it
10    print(sprintf('Here is the abs value error: %s', as.character(error)))
11    return(error)
12  }
```

Carlos's Pushed Version of Loss Script

But before Grace realizes Carlos has modified the script, she also fixes the bug, but instead by changing the absolute value to a squaring function:



```
 1
 2  # Print and return squared error
 3
 4  loss <- function(x, y){
 5
 6    # Calculate squared error
 7    error <- (x-y)**2
 8
 9    # Print and return it
10    print(sprintf('Here is the squared error: %s', as.character(error)))
11    return(error)
12  }
```

Grace's Modified Version of Loss Script

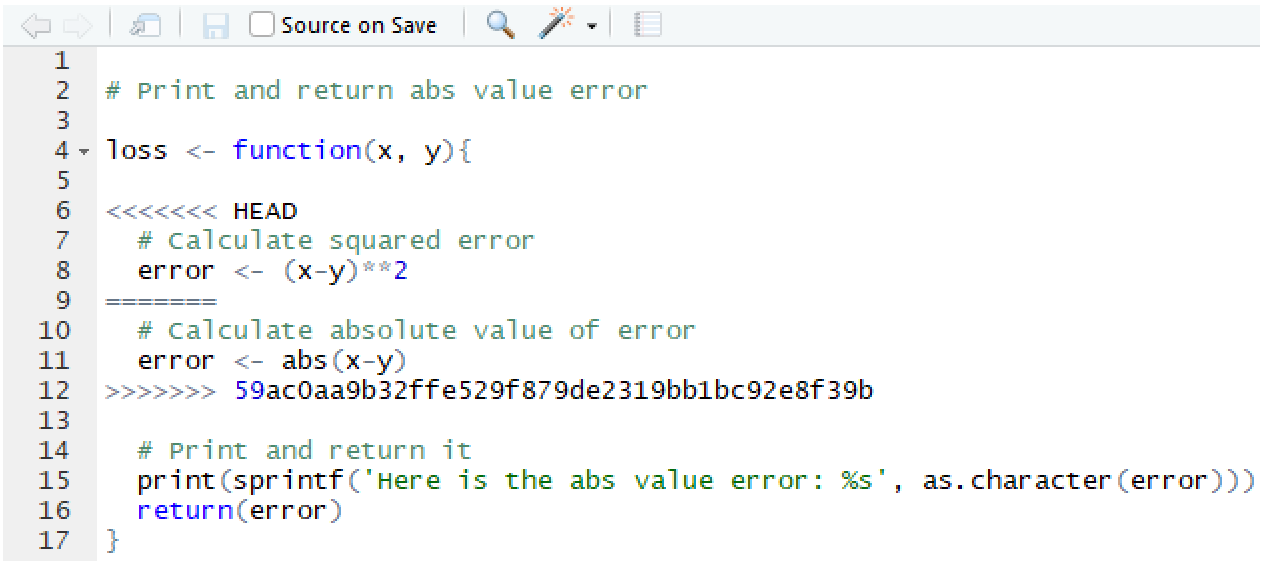
Git is smart, so When Grace tries to push her version to github, git will recognize that the two versions of the script conflict. As a result, it will throw the following error:

```
C:\Users\gracesmith\Documents\R\repo_name>git push
To https://github.com/gracesmith/repo_name
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/gracesmith/repo_name'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Effectively, Git recognizes that Carlos has made modifications to the script which were pushed to Github that Grace doesn't have. This usually happens *when two programmers modify exactly the same line in the single script*, such as line 6 in the above examples. To solve this problem, Grace should follow Git's advice, and try the "git pull" command. This will lead to the following message from git:

```
C:\Users\gracesmith\Documents\R\repo_name>git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/gracesmith/repo_name
   e719863..59fc0ba  master     -> origin/master
Auto-merging scriptname.R
CONFLICT (content): Merge conflict in scriptname.R
Automatic merge failed; fix conflicts and then commit the result.
```

Again, Git is letting Grace know that it sees there's a conflict and that Git needs human help to fix it. Once Grace has pulled Carlos's changes, she can go open the script in her computer and should see something like the following:



```
# Print and return abs value error

loss <- function(x, y){

<<<<<<< HEAD
  # Calculate squared error
  error <- (x-y)**2
=======
  # Calculate absolute value of error
  error <- abs(x-y)
>>>>>>> 59ac0aa9b32ffe529f879de2319bb1bc92e8f39b

  # Print and return it
  print(sprintf('Here is the abs value error: %s', as.character(error)))
  return(error)
}
```

Conflicted Version of Loss Script

The "<<<<" and ">>>" lines are in the script to signal the beginning and end of a merge conflict, whereas the "===" line separates the two different versions. On the top is Grace's version and on the bottom is Carlos's. To resolve the merge conflict, Grace should manually select the two lines she prefers and then delete all of the "<, =, >" symbols to let Git know that the conflict has been resolved. Then, she can commit and push the file to Git, and she'll be good to go!

Overall: merge conflicts are rather annoying and can be confusing, so the best way to avoid them is to clearly deliniate which programmers will be working on which sections of which scripts. This will prevent Git (and you) from getting confused.

Tips and Tricks

Git can be confusing, but there are three general rules of thumb you can follow to figure out any bugs that come up:

1. Git is super smart. If there's a bug and Git is recommending a particular way of fixing it, there's a very high chance that Git is right.
2. Use the git status command. After entering git status, git will give you a rundown of the status of the entire repository.

3. If that doesn't work (or you don't understand what is going on), you can always copy and paste the error from the command line into Google and reference stackexchange. There is a wealth of online help!

Try not to worry too much! Git almost exclusively *adds information*, meaning that you can always just go back and restore a previous version - you'll never lose your work.

Cheat sheats

Lastly, there are a couple of cheat sheets that you can use to make your life easier.

- Karl Broman's tutorial (http://kbroman.org/github_tutorial/pages/init.html) runs you through initializing a repository
- This official github cheat sheet (https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf) lists all of the general commands you'll need

4. Further reading

If you're interested in learning more about Git and Github, you might want to take a look at the following resources:

- The Software Carpentry Foundation has a great intro-mid level git use tutorial here (https://swcarpentry.github.io/git-novice/)
- Atlassian has some wonderful advanced git use tutorials here (https://www.atlassian.com/git/tutorials/advanced-overview)

5. Sources cited

http://kbroman.org/github_tutorial/pages/init.html (http://kbroman.org/github_tutorial/pages/init.html) https://git-scm.com/book/en/v2/Getting-Started-Git-Basics (https://git-scm.com/book/en/v2/Getting-Started-Git-Basics) https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup (https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup) https://www.git-tower.com/learn/git/faq/restore-repo-to-previous-revision (https://www.git-tower.com/learn/git/faq/restore-repo-to-previous-revision)