Software Tools for Business Analytics

Lecture 2: Version Control, Git, and Github

Bradley Sturt • January, 2017 • Massachusetts Institute of Technology

Last Revision: January 22, 2017

Table of Contents

1 Introduction

1.1 What is Version Control, Git, and Github?

Version control systems are software tools for saving snapshots of a set of files and directories on our computers. For example, suppose we and a friend are consultants, and working on a report for our client. The report includes a Word document (`report.txt`), high-resolution figures to go along with the report (`fig1.png`, `fig2.png`), and some accompanying Julia code (`my_code.jl`). We organize all of these files by placing them into a directory, which we name `my_report/`.

As we and our friend work on the report on your separate computers, a variety of situations arise.

- We updated the introduction in `report.txt` late last night, and made some other minor changes, and we want our friend to build on the updated copy.
- Our friend added some features to `my_code.jl`, but they make the code crash, so we want to revert the file to an older version.
- We want to keep track of the different versions of the `fig1.png` and `fig2.png` over time, since we're not sure which versions we will want to send to the client.

Version control systems are designed to solve these problems. They allow us to build a history of "snapshots" of a project over time, revert the state of our project to earlier snapshots, and collaborate with others on a single project.

A variety of version control systems exist, and one of the most popular is **Git**. Git is a version control system written by Linus Torvalds (the creator of Linux, an alternative operating system to Windows and Macs). Linux is available for free, and it is is developed for free by a highly-decentralized community of thousands of volunteers. Linus designed Git to help this large and scattered community, working on their own computers and in different time zones, collaborate on Linux and share code easily.

When collaborating with others on a project, it is useful to "host" our project on some website. By hosting our project on some website, we can share the project with others more easily. Also, if every collaborator's computer crashed, there is a backup of the project and its snapshots on that website. **Github** is a popular hosting server for Git projects. It also functions as a place to search for other people's projects. When people write programs, software tools, and personal websites, they often upload their projects publicly on Github, where anyone can search for and download their projects.

In this lecture, we will focus on a simple, practical workflow for using Git and Github, which will have immediate and long-term value. In the short run, this understanding will help us access the lecture materials for the course. In the long run, through learning Git and Github, we will build a practical understanding of how to use version control to help us work on projects more productively, collaborate more effectively, and have the tools to accessing code that others have put on Github.

Before moving on, we want to mention that there is an abundance of fantastic and readable websites for learning Git, from which these notes are based. These lecture does not aspire to be a replacement for those tutorials and resources available online; rather, our goal here is to briefly introduce the components of Git and Github that will be used in this course.

1.2 Why learn these tools?

Git is by no means the only version control system available, neither is it the simplest. However, we believe it is worthwhile at this point to learn Git, for a variety of reasons, including the following:

- Its popularity is likely to continue.
  - Git has become a standard version control systems, both for personal use and in industry.
  - When you collaborate with developers or researchers in the future, it is likely that Github will be their preferred choice.
- Github is ubiquitous for sharing code
  - Many great computer tools and programming resources are available only on Github
  - Therefore, we are at a disadvantage if we don't know how to use Github
- Git is very powerful
  - Git has an extensive list of useful features, to help us collaborate on projects easily
  - Once we pass the initial learning curve, we have a wide variety of new techniques at our disposal
- Github let's us back up our projects
  - By hosting our projects on Github, we have a back up of our projects, in case our computer crashes.

2 Git Basics

2.1 Basic Concepts

We begin by discussing some basic terminology of Git, which we will use throughout the lecture.

In our example of the report for the client, we will use Git and Github to aid collaboration and to keep track of snapshots of the project, which include the directory `my_report/`, and the files contained in it. In Git, each snapshot is called a **commit**; in other words, each commit is a copy of the entire state of the files and directories in the project, exactly as they were when the commit was made.

All previous commits (by you and the collaborators) to your project are stored as a **repository**, often abbreviated as the **repo**. There is a copy of the repository stored on your computer, on Github, and on your collaborators' computers.

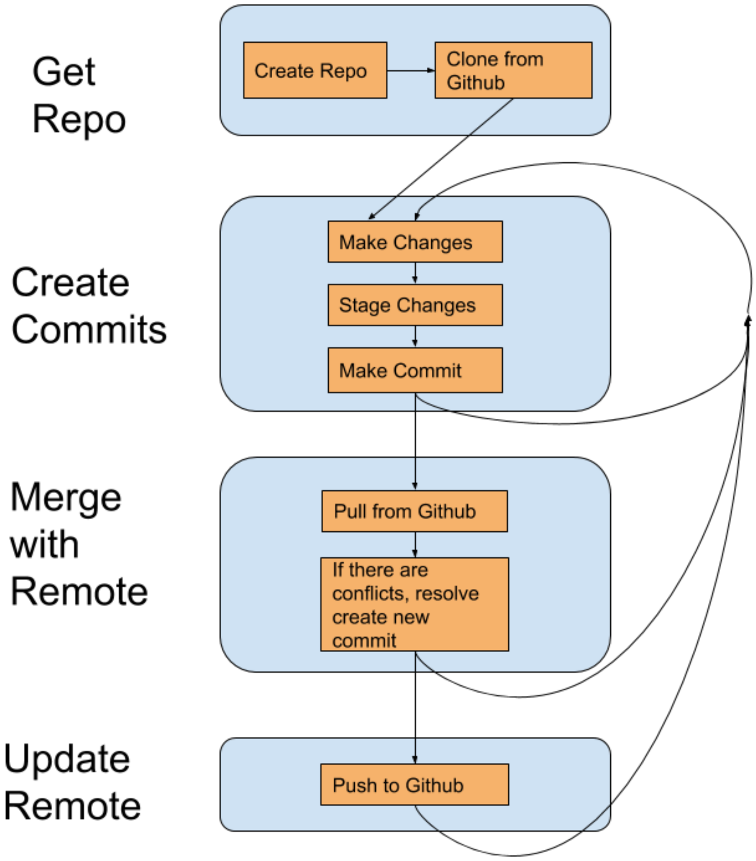
Section 2.1 – Key Ideas

- A commit is a snapshot of the project at some point in time
- The history of commits is stored in a repository
- There is a copy of the repository on each person's computer, and on Github

3 Git Workflow

3.1 Overview

In this course, we will work with Git and Github using the work flow represented in the diagram below. The workflow is split into four parts: getting a repo, creating commits, merging with remote commits, and updating the remote repository. We will explain these parts in more detail in the following sections. For now, we just glance at the diagram, take note of the four different parts, and move right along onto the next sections.



3.2 Getting a Repository

3.2.1 Repository Overview

Our first step is to learn how to get a copy of someone else's repository on our own computer. As we discussed before, a repository is the history of commits for a project. In other words, a repository is the collection of the snapshot of the project.

Furthermore, there is a copy of the repository that is hosted on Github. That copy of the repository in Github is called the **shared repository**, or the **remote repository**. When we work with Git and Github, we work on a copy of the shared repositories on our local computer. So, we want to make a **local copy** of the shared repository on our own computer. The process of making a local copy of the shared repository is called **cloning** their repository.

There are two ways to get a repository. We can create a new repository, or we can get a copy of (i.e. "clone") an existing repository from Github. We first discuss getting a copy of an existing repository from Github, since creating a new repository is essentially the same process, with just one extra step.

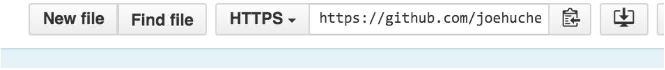
Section 3.2 – Key Ideas

- The first part of the workflow is to get a local copy of the shared / remote repository
- The shared repository / remote repository is on Github, and cloning creates a local copy
- To create a new repository, we create a new repository on Github

3.2.2 Cloning Existing Repositories

Let's now dive into the details for cloning an existing repository, using this class's course materials as an example. In this IAP course, all of the course materials on Github in a single repository, named `OR-software-tools-2016`. It has many files in it. In particular, some of the files contain code that we will want to run and modify in later lectures. So, we want to make a copy of the repository on our own computers.

To clone the repository off Github, we need the link for the repository. We find it near the top of the repository's Github webpage.



https://github.com/joehuche

We then open up our our terminal, and move into the directory where we want to place the local copy of the repository. For convenience, let's make a directory which will store all of the git repositories in this class.

```
$ mkdir iap_class
$ cd iap_class
$ pwd
/my/path/to/iap_class/
```

Now, we use `git clone` to copy the materials into this directory:

```
$ git clone https://github.com/joehuchette/OR-software-tools-2016.git
Cloning into 'OR-software-tools-2016'...
remote: Counting objects: 88, done.
remote: Compressing objects: 100\% (4/4), done.
```

```
remote: Total 88 (delta 0), reused 0 (delta 0), pack-reused 84
Unpacking objects: 100\% (88/88), done.
Checking connectivity... done.
```

Let's see what happened:

```
$ pwd
/my/path/to/iap_class/
$ ls
OR-software-tools-2016/
$ cd OR-software-tools-2016/
$ ls
1-intro-terminal-and-R/          2-data-wrangling          ...
```

To recap: we ran `git clone`, with the link to the repository on github. This cloned (i.e. copied) the repository (i.e. the most up-to-date commit of the project, along with the history of all previous commits) onto our computer, into a new directory called `OR-software-tools-2016/`.

We now understand how to clone a repository from Github. This is the process we will use, any time we want to work with a repository from Github.

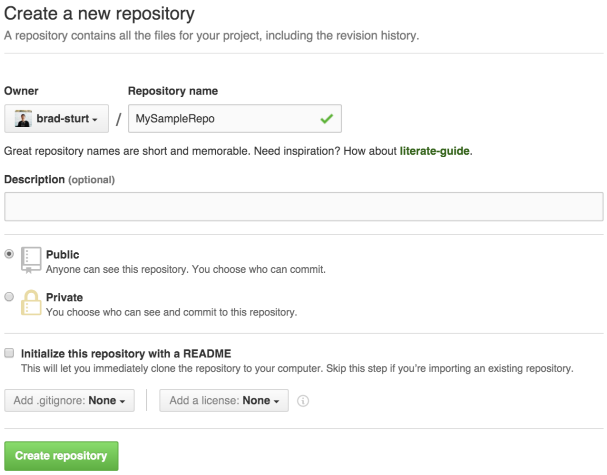
Section 3.2.2 – Key Ideas

- `git clone`, followed by the link to the shared repository, creates the local copy

3.2.3 Creating New Repositories

In the previous section, we learned how to clone a repository from Github. Here, we will see that the process for creating a new repository is quite similar.

To create a new repository, we open up the Github webpage. Near the top of the page, we find the button for "Create New", and then select "Create New Repository". We add a name for our new repository. We have an option of setting the repo to "public" or "private". We choose "private" only when we don't want the project to be accessible to random people. We leave the rest of the settings as default, and then create the repo.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: brad-sturt / Repository name: MySampleRepo

Great repository names are short and memorable. Need inspiration? How about literate-guide.

Description (optional)

● Public
Anyone can see this repository. You choose who can commit.

○ Private
You choose who can see and commit to this repository.

□ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

We now have a new repository! To get a copy on our local computer, we clone it using the same process as above.

Section 3.2.3 – Key Ideas

- We create new repository through the Github website
- Once it is created, we clone the repository to our local computer

3.3 Creating Commits

3.3.1 Commit Overview

In the previous section, we showed how to clone and create repositories, which corresponded to the first part of the workflow. Now that we have a local copy of a repository, we get started working on the project. We can make changes to the files, updating existing files, add new files, deleting files, moving files, etc. This constitutes the "Make Changes" step of our workflow.

At some point, after we have been making changes to the files, we want to create a new snapshot (commit) to record and possibly share the state of the project. Commits can be made at any time, but it is considered good practice to make commits frequently, each time a small task in the project is completed.

Before a new commit can be made, though, Git needs to be explicitly told which files in the project have been changed since the previous commit. The process of telling Git what files have been changed is called **staging** the changes, and corresponds to the "Stage Changes" step.

During the staging process, we don't need to stage every file that is being **tracked** by Git (i.e. every file that has been included in previous commits); we just need to stage the files that have been changed since the previous commit.

Section 3.3.1 – Key Ideas

- After we make changes to the local copy, we save the project as a commit
- Before we make a commit, we must stage the changes to the project
- It is good practice to create commits frequently, after completing small changes to the project.

3.3.2 Commit Details

Let’s go through the specific commands used to stage changes and create a commit.

The command `git status` tells us which files have been changed, and thus need to be staged. To see `git status` in action, let’s return to our original example of the consulting report. Just as before, we are working on a consulting project with a friend, and our repository originally consists of a directory `my_report/`, which contains files `fig1.png`, `fig2.png`, `my_code.jl`, and `report.txt`.

We cloned the repository a while back, and we have since made some changes. We added some edits to the introduction of `report.txt`, and we added a new figure to the project, called `fig3.png`. We decide we are ready to make a new commit, so we run `git status` to see what files need staging:

```
$ git status
On branch master
Changes not staged for commit:
  (use ‘git add <file>...’ to update what will be committed)
  (use ‘git checkout -- <file>...’ to discard changes in working directory)

        modified:   report.txt

Untracked files:
  (use ‘git add <file>...’ to include in what will be committed)

        fig3.png
```

First, we notice that the second line says `Changes not staged for commit:`, followed below by the line `modified: report.txt`. Thus, `report.txt` is being tracked (i.e. it was included in the previous commit), has been changed, but has not yet been staged. Second, we find the line `Untracked files:` followed by the line `fig3.png`. Therefore, `fig3.png` is a file that now appears in the directory that was not tracked in a previous commit. We conclude that, in order to save the project as a commit, `report.txt` and `fig3.png` need to be staged.

To stage files that have been added or changed, we use `git add`, followed by the name of the file. Let’s see this in action:

```
$ git add report.txt
$ git add fig3.png
$ git status
On branch master
```

```
Changes to be committed:
  (use ‘git reset HEAD <file>...’ to unstage)

        new file:   fig3.png
        modified:   report.txt
```

By running `git status`, we see that all of the changes have been staged, and they are now ready to be made into a commit. To make a commit, we run the command `git commit -m "commit summary"`, where the text `commit summary` is replaced by a short summary of the changes made to the project since the last commit. For example:

```
$ git commit -m "Added␣new␣figure,␣updated␣report"
[master bb6f55b] Added new figure, updated report
 2 files changed, 1 insertion(+)
 create mode 100644 fig3.png
$ git status
On branch master
nothing to commit, working directory clean
```

After creating the commit, we checked that the commit was successful `git status`. We see that the working directory is clean, meaning that there have been no changes to the project since the previous commit, meaning that everything worked!

When we staged the changes, there are a lot of changes, like deleting a file, that are not staged with `git add`. For example, if a file is deleted, the change is staged with `git rm`, followed by the name of the file that was deleted. To see a complete list of possible stages, we search on Google.

Section 3.3 – Key Ideas

- `git status` tells us what files have been changed, and thus need to be staged
- `git add`, `git rm`, etc stage the changes
- `git commit -m`, followed by a message, creates a commit with the staged files

3.4 Merging with Remote

3.4.1 Merging Overview

In the previous two sections, which corresponded to the first two parts of our workflow, we saw how to get a repository (i.e. the history of commits to a project) on our local computer, and how to create new commits after making changes to our local repository.

While we have been working on our local copy of the repository, our collaborators may have also been working on their local copies of the repository. If so, it is very possible that, in the time since we cloned the repository, our collaborators have made some commits on their own computers, and updated the shared repository with those commits.

To exemplify this more clearly, let's consider again the consulting project example. Suppose our friend has changed the file `my_report.txt` on her own computer by adding a new section to the report. She then created a commit on her own computer, and then "pushed" the commit to the Github repository (more on this later). Therefore, the shared repository is no longer the same as it was when we cloned our local copy.

We want to update our local copy of the repository with the changes that our friend made, so we can be in sync with our friend and have our own repository be up to date. In other words, we want to **merge** the most recent commit on the remote repository with our most recent commit, which is the third part of the workflow.

A couple of notes. First, Git does not allow us to begin the merge process unless we have a **clean working directory**; that is, the state of the project is identical to that of the previous commit. So, if we have made changes to the files since the previous commit, we must make a new commit before Git allows us to merge. Second, it is good practice to keep our local repository up to date with the shared repository as often as possible; that way, our copy of the repository won't diverge too far away from our collaborators' local copies. Thus, whenever we finish making a commit, we do a `git pull` to merge that commit with the shared repository.

Section 3.4.1 – Key Ideas

- In the time since we originally cloned the repository, the shared repository may have been updated by our collaborators
- Merging is the process of integrating the new commits in the shared repository into our local repository
- Before we can begin merging, our working directory must be clean. That is, the files in the project must not have changed since our most recent commit.
- It is good practice to merge often with the remote repository

3.4.2 Merging Details

Merging is done between the most recent commit in the shared repository and the most recent commit on our computer. In other words, merging will look at the differences between the most recent commit that has been "pushed" by a collaborator to the Github repository, and our most recent commit, and try to combine the changes together.

The main command in merging is `git pull`. When we execute the command `git pull`, Git takes two actions. First, it downloads a copy of the remote repository from Github onto our local computer. Second, it tries to automatically integrate the most recent commit from the shared repository with the most recent commit on our computer.

Most of the time, Git figures out a way to combine the changes from the two commits. If that occurs, the merging process is done! However, in some cases, such as where the two commits have made changes to a file that contradict each other, Git requires us to specify how the merging should take place. These are called **conflicts**, and we need to tell Git how to **resolve**. After we resolve the conflicts, we will need to stage the files where the conflicts occurred, and make a new commit.

Let's see this in action. Suppose we have changed `report.txt` since we cloned the repository, changing it to only the line `Local Change!`. We can view this with `cat`:

```
$ cat report.txt
Local change!
```

We created a commit on our local computer with this change. Meanwhile, our friend has changed the same file on her own computer to contain only the line `Remote change!`. She made a commit, and "pushed" the commit to the shared repository. Let's see what happens when we try to do a `git pull`.

```
$ git pull
remote: Counting objects: 6, done.
remote: Compressing objects: 100\% (2/2), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100\% (6/6), done.
From https://github.com/.../my_report
   494dbfd..0036b05  master     -> origin/master
Auto-merging report.txt
CONFLICT (content): Merge conflict in report.txt
Automatic merge failed; fix conflicts and then commit the result.
```

When we see the line `CONFLICT (content): Merge conflict in report.txt`, we know that there was a conflict in the file `report.txt`. To see what conflict occurred, let's look in `report.txt`.

```
$ cat report.txt
<<<<<<< HEAD
Local change!
=======
Remote change!
>>>>>>> 0036b05ecd828fddf405c83efdef2bbfcc5bbc63
```

We now see that the conflict occurred because Git didn't know if the first line of `report.txt` should be `Local change!` or `Remote change!`. To resolve the conflict, we edit `report.txt` and delete the lines that we don't want to keep. We then make a new commit with the changes.

Section 3.4.2 – Key Ideas

- Merging is done between the most recent commit on our computer, and the most recent commit in the shared repository.
- `git pull` downloads a copy of the remote repository, and tries to automatically merge the two commits into one
- Sometimes, `git pull` cannot figure out how to reconcile differences between the commits. These result in conflicts
- If there are conflicts, we must resolve each conflict by editing files, and create a new commit

3.5 Updating Remote

In the previous sections, we have seen how to make commits on our local copy of a repository. Also, we saw that collaborators can update the shared repository with their own commits, and we learned how to merge the new commits on the shared repository into our local copy. Here, in the final part of our workflow, we learn how to update the remote repository ourselves.

After we have made changes to the project, and saved those changes as commits, we want to update the shared repository with those commits. That way, the commits will be saved on Github (in case our computer crashes), and our collaborators can build on an updated version of the project.

The process of updating the remote repository with our commits is called **pushing** our changes to the remote repository. Before we can push, we must have a clean working directory, and must be up to date with the remote repository; that is, we must have done a `git pull` since the remote repository was last updated.

So, when we are ready to push our changes to Github, we first use `git status` to check that we have a clean working directory. If there are any unstaged changes, we must stage them and make a commit. Second, we run `git pull` to check that we are up to date with the remote repository, and merge any new changes.

Once we have a clean working directory and are up to date with the remote repository, we are ready to push our changes. This just requires the command `git push`. Git will ask us for our Github username and password, to make sure that we have the correct permissions to update the repository. We type those in and press enter. Once that is done, we open up the Github website, and we see that the repository has been updated with our commits.

Section 3.5 – Key Ideas

- `git push` updates the remote repository with our commits
- Before we push our updates, we run `git status` and `git pull`, to ensure that we have a clean working directory and are up to date with the remote repo