

Version Control and GitHub

```
$ echo "Data Sciences Institute"
```

Prerequisites:

- GitHub account

Key Texts:

- Chacon and Straub, 2014, Pro Git, 2nd Edition.
- Timbers, Campbell, Lee, 2021, Data Science: A First Introduction, <https://ubc-dsci.github.io/introduction-to-datascience/>

References

- Chacon and Straub: Chapter 1
- Timbers: Chapter 12.3 - 12.4, 13.3.1

Version Control

What is Version Control?

Version control is a system that records changes to a file or a set of files over time so that we can recall a specific version later. We may already do this by copying files to another directory to save past versions. While it is simple, it lacks flexibility and complexity.

Version Control Systems (VCS) can do a number of things and can be applied on nearly any type of file on our computers:

- revert files to a previous state
- revert entire project to a previous state
- compare changes over time
- see who modified something last
- who introduced an issue and when
- recover lost files

Local Version Control Systems

Local VCSs were developed to keep track of changes to our files by putting them in a version database.

Centralized Version Control Systems

Centralized VCSs (CVCS) were developed to enable collaboration with developers on other systems. CVCSs have a single server that contains all the versioned files.

CVCSSs allow some level of transparency to others' work and give Administrators a level of control over what developers can and can't do.

Unfortunately, a single server means that if it ever goes down, all collaboration halts for however long that lasts for. Additionally, if backups haven't been kept, work could easily be lost.

Distributed Version Control Systems

To handle the limitations of LVCSs and CVCSs, Distributed VCSs were created. This includes Git, Mercurial and Bazaar.

Collaborators mirror the entire repository, therefore if a server dies, any one of the collaborators' repositories can be copied back to the server to restore it.



center

Questions?

Git

Git Basics

Git thinks of data in a very different way than other VCSSs. Instead of storing a set of files and the changes over time, Git thinks of its data more like a set of snapshots of a mini file system.

If files have not changed, Git does not store the file again, it links to the previous identical file already stored.



center

Local Operations

Most operations on Git only need local files and resources to operate. Git also keeps the entire history of our projects on our local disks meaning we can see changes made months ago without a remote server.

We also don't need to be connected to the server to get work done, rather we only need to be connected when we want to upload our work.

Benefits

Git uses a check-summing mechanism called *SHA-1 hash* which is calculated based on the contents of a file or directory structure in Git. It looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00393
```

This checksum means it's impossible to change the contents of any file or directory without Git knowing about it.

Git generally only adds data, making it fairly difficult to lose data once we've committed, which we'll learn about later.

The Three States

There are three main states that our files can reside in:

- Committed:
 - data is safely stored on local database
- Modified:
 - file has been changed but not yet committed
- Staged:
 - modified file has been marked to go into the next commit

The Three Main Sections

There are three main sections to a Git project:

- The Git directory
- The working directory
- The staging area

The Git Directory

The Git directory is where Git stores the metadata and object database for our projects. It is what is copied when we clone a repository from another computer.

The Working Directory

The working directory is a single checkout of one version of our projects. These files are pulled out of the compressed database in the Git directory and placed on the disk for us to modify.

The Staging Area

The staging area is a simple file that stores information about what will go into our next commit.

Workflow

A basic workflow will look something like this:

1. Modify files in our working directory
2. Stage the files in the staging area
3. Commit the changes which takes the files from the staging area and stores them on the Git directory.

Questions?

Installing Git

Typically, Git is already installed on our system but we can check for that using the `git` command:

```
$ git --version
```

Does anyone not see a version?

Installing on Linux

If you're on Ubuntu:

```
$ sudo apt install git
```

If you're on Fedora, RHEL or CentOS:

```
$ sudo dnf install git
```

```
$ sudo yum install git
```

Installing on Mac

You can install Git via Homebrew, if you have Homebrew installed (<https://brew.sh/>).

```
$ brew install git
```

Finally, you can install Git from source at this link:
<https://sourceforge.net/projects/git-osx-installer/>

Installing on Windows

The download will start automatically through this link:

<https://git-scm.com/download/win>

Questions?

Git Setup

The first thing to do now that we have Git installed on our system is to customize it. These changes will remain despite any upgrades to Git that we install.

Using the command `git config`, we can set configuration variables that control all aspects of how Git looks and operates.

Checking Configurations

Before we change any of our global configurations, we can check what they are:

```
$ git config --list
```

If we haven't configured Git, we can do that now!

Identity

First, we'll set our username and email address. Git uses this information everytime we commit.

```
$ git config --global user.name "Rachael Lam"  
$ git config --global user.email "rachael.a.lam@gmail.com"
```

The option `--global` means that we only have to pass this through once.

Editor

Next, we'll configure our the default text editor for when Git needs to type in a message. Git uses our system's default editor (usually Vi or Vim) but we can change it if we prefer. If we want to change the editor to emacs, we would do so below:

```
$ git config --global core.editor emacs
```

Diff Tool

We can also set the default diff tool which is used to resolve merge conflicts:

```
$ git config --global merge.tool vimdiff
```

Checking the Setting

We can use the `git config --list` command to see all Git settings. See the values of a specific setting:

```
$ git config user.name
```

Help

If we ever need help, even offline, we can access the manual page three ways:

1. \$ git help <verb>
2. \$ git <verb> --help
3. \$ man git-<verb>

For example, we can get help for the `config` command:

```
$ git help config
```

Questions?

Git Basics

References

- Chacon and Straub: Chapter 2

```
$ git init / $git clone
```

Respositories in an Existing Directory

We're quickly getting into how to start our first Git repository, or commonly known as repo. First we'll learn how to import an existing repo into Git:

```
$ git init
```

```
$ git init -b main
```

Here we're creating a new subdirectory named `.git` that will contain all our necessary repo files. The option `-b` will create a new branch called `main`.

Cloning an Existing Repository

If we want to collaborate on an existing repo, we need to clone the repo from GitHub. If we don't have a project set up yet, we'll need to do that first.

1. Create a new project

 center

2. Add name and optional description

 center

3. Choose public or private and add initialize



center

There are a number of automatically generated files such as log files that we might not want Git to add or show as untracked.

We can create a file called `.gitignore` to ignore the automatically generated files.

The `.gitignore` is dependent on the type of coding language you are using but can also be modified to fit specific purposes.

If we created a repo on GitHub, we can choose a `.gitignore` template. We can select a template specific to the coding language we are using.

 center

Once we have our repo, we can clone it:

```
$ git clone https://github.com/rachaellam/git-module.git
```

Using this code, we've created a repo called `git-module` (by taking the last part of the link) and initialized a `.git` directory and pulled all data for that repository while checking for the latest copy.

The url used in the previous code block is copied directly from GitHub by clicking code and copying the HTTPS:

 center

If we want to change the name of the repo, we can specify that as the next command line option:

```
$ git clone https://github.com/rachaellam/git-module.git mymodule
```

Questions?

Git Commands

References

- Chacon and Straub: Chapter 2
- Timbers: Chapter 12.5

```
$ git status
```

Tracked and Untracked Files

Files in our working directory can either be tracked or untracked. Tracked files are files that were in the last snapshot and can be unmodified, modified or staged. Untracked files are files that aren't in our last snapshot or staging area.

When we modify a file, Git keeps track of the modifications even before we've decided to commit. We can then stage the modifications and then commit.



center

File Status

To better understand what state our files are in, we can check the status:

```
$ git status
```

If we've just created our repo, we should see (or something similar):

```
# On branch main
# Your branch is up to date with 'origin/main'.
#
# nothing to commit, working tree clean
```

Let's now add a README.md file, because every good repo has a good README.

```
$ touch README.md
```

And see the status:

```
$ git status
```

On branch main

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)
README.md

Here we can see that we still haven't committed anything and that we have an untracked README.md file. Git also gives us a bit of information including how to add a file to track.

```
$ git add
```

Tracking New Files

To track new files, or stage new files, we can use `git add` along with the file that we want to track:

```
$ git add README.md
```

We can run `git status` again to see the results of `git add .`

```
On branch main
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)  
  new file: README.md
```

Now we can see that our README.md file is staged to be committed.

Let's say we add some more info to our README.md file, which has now been tracked. If we run `git status`, we can know:

```
On branch main
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)  
  new file: README.md
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
  modified: README.md
```

We can stage our additional changes and check the status:

```
$ git add README.md  
$ git status
```

```
On branch main
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)  
  new file:   README.md
```

Let's try adding another file into our directory. It can be something that you've been working on independently, or we can add our project from the previous Unix module.

If we modify many things at once, we can add the option `-A` to add all files, rather than adding one by one

```
$ git add -A
```

A little note about this: it's best to upload your work in small chunks for readability and for collaboration. So if you have a bunch of files, it's recommended to split them into smaller chunks.

Questions?

```
$ git diff
```

If we want to see more details of the changes that we've made, we can use the command `git diff`.

`git diff` compares what is in our working directory to what is in our staging area. If we've made changes to our files without running `git add`, we'll see the comparison. If there are no differences, nothing will be shown.

```
diff --git a/README.md b/README.md
index e69de29..4711fce 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+# git-r
\ No newline at end of file
```

```
diff --git a/README.md b/README.md
```

This is telling us what we're comparing. In this case, it's the difference between a previous version of the README file and the current one

```
index e69de29..4711fce 100644
```

Here is some meta data, or hash identifier that we likely won't need.

```
---- a/README.md  
+++ b/README.md
```

This is acting as a legend. Changes from a/README.md are marked by ---- and changes from b/README.md are marked by +++

```
@@ -0,0 +1 @@
+# git-r
```

Here we're being told the lines that have changed and what on those lines changed. Because there was nothing removed, this is a bit of a simplistic representation.

We might see something more like...

```
@@ -21,5 +77, 12
```

This is telling us 5 lines were removed starting on line 21 and 12 lines were added starting on line 77.

--staged

If we want to see the details of what will go into the next commit, we can use `git diff` with the option `--staged`

```
$ git commit
```

Once we've staged your selected files, it's time to commit the changes. Anything that wasn't staged (any modifications since `git add`) will not be included in the commit.

`git commit` is most easily run with the option `-m`. This adds a message to your commit

```
$ git commit -m "adding a message here"
```

-m

Messages should be clear. They can also be extremely detailed if needed. By not including the option `-m`, Git will provide the latest output of `git status`. If you want even more information, you can use the option `-v`.

Messages are extremely important for our own records and also when collaborating with others. They can act as a reminder for what our commit includes, and also tell our collaborators what we did last.

It's important to commit often as well so that merges are easier to locate and fix.

It's also helpful if you want to go back to an earlier version. You have more options to choose from.

Practices around messages can vary but if we want to add a longer message we can remove the `-m` option.

```
$ git commit
```

Then hit `i` to add a message. You'll see `-- INSERT --` at the bottom and you can begin typing your message.

When finished, press `esc` then `:wq` or `:x`.

`w` means write and `q` means quit. `x` is shorthand for `wq`

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body, the blank line separating the summary from the body is critical (unless you omit the body entirely).

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space with blank lines in between, but conventions vary here

-a

If we want to commit all the files we've worked on without putting them in the staging area, we can use the option **-a**. This will avoid using **git add** and condense our workflow.

```
$ git commit -a -m "skip staging add message"
```

Here we've used two options, **-a** and **-m** to skip the staging and add a message.

Questions?

```
$ git rm
```

If we delete a file from our working directory after staging it using `rm` without `git`, the file will show up in our untracked files. We can then use `git rm` to stage the file's removal.

Let's follow the code below to understand this better:

```
$ touch test.sh  
$ git status  
$ rm test.sh  
$ git status
```

Because we haven't tracked the `test.sh` file so we can remove it and we don't need to tell git to also remove it.

What happens if we add a file to our staging area but then want to delete it? Let's try the two codes below:

```
$ touch test.sh  
$ git add test.sh  
$ git rm test.sh
```

```
$ touch test.sh  
$ git add test.sh  
$ rm test.sh  
$ git rm test.sh
```

-f

If we've modified and staged a file, we have to force the removal with the option `-f`. This is a safety feature so that we don't accidentally delete something.

```
$ touch testfile  
$ git add testfile  
$ git rm -f testfile
```

--cached

The option `--cached` allows us to remove a file from our staging area without permanently deleting it from our local drive.

```
$ git rm --cached testfile
```

We can use wildcards to remove files from our staging area in bulk, although we have to add a backslash in front of `*` because Git does its own filename expansion.

```
$ git rm -f \*.txt
```

We can also delete files in a folder of our working directory:

```
$ git rm -f dir1/*.sh
```

```
$ git mv
```

Using `git mv`, we can rename files conveniently and succinctly:

```
$ git mv test.txt test.sh
```

Questions?

```
$ git log
```

Sometimes we might want to see a history of our commits or we want to see previous commits after cloning an existing repository. We can do this using the `git log` command.

```
$ git log
```

There are a number of options that help us see even more, or sometimes less, information about each commit.

If we attempt to run a log before any commits have been made, we will get an error:

```
fatal: your current branch 'main' does not have any commits yet
```

-p

Adding the option `-p` will show the `diff` introduced in each commit. We can also pass a number option that will limit the number of entries shown:

```
$ git log -p -2
```

Entries can be any number of entries (`-<n>`) but is limited to one page of log out puts

--stat

The `--stat` option shows abbreviated stats for each commit:

```
$ git log --stat
```

```
commit 6c91df668d1899317a643153bd169d37fe05d9f1 (HEAD -> main)
Author: Rachael Lam <rachael.a.lam@gmail.com>
Date:   Fri Feb 18 14:56:27 2022 -0500
```

first commit

```
.gitignore |  4 +++
README.md   |  1 +
test.Rproj  | 13 ++++++
testfile.r  |  0
4 files changed, 18 insertions(+)
```

+ or - (if there were any) show the number of insertions or deletions. We can also see the date of the commit, who committed and the message.

--pretty

The `--pretty=` option is an interesting feature that enables us to specify the log output when we combine it with `format:` , creating an extremely useful data extraction feature:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

Formatting Options

Option	Description
%H	Commit hash
%h	Abbreviated commit hash
%t	Abbreviated tree hash
%p	Abbreviated parent hashes

Option	Description
%an	Author name
%ae	Author email
%ad	Author date (ex. Thu Dec 2 14:14:55 2021 -0500)
%ar	Author date relative (ex. 26 hours ago)
%cn	Committer name
%s	Subject (-m)

--since / --until

The options `--since=` and `--until=` are more usually more useful than `-(n)`. They produce the logs of any time before (`--until`) or after (`--since`) a certain date. You can specify an exact date or relative date:

```
$ git log --since=2.weeks
```

```
$ git log --since="2 days 3 minutes ago"
```

```
$ git log --until="2021-11-20"
```

We can also combine log options to generate specific outputs:

```
$ git log --pretty=format:"%h: %s" --author=Rachael
```

```
$ git log --after="2020-11-01" --since="2020-11-30"
```

Finally, and a favourite for quick glances:

```
$ git log --oneline
```

Questions?

undo undo undo

Changing Commit

If we already committed a few files but forgot to add one or made modifications since our commit that we want to add, we can use the option `--amend`

```
$ git commit -m "initial commit"  
$ git add missed_file  
$ git commit --amend -m "initial commit with missed_file"
```

We can still add the `-m` option to add a new comment.

Unstaging

When we want to remove a file from our staging area because we accidentally added one too many files, we can use the code below:

```
$ git reset HEAD README.md
```

If we ever forget how to do this, running `git status` will remind us.

Unmodify

We can also revert our files back to the version from our previous commit using `git checkout --`. It's important to realize that this command essentially rewrites the file so any changes that were made will not be able to be recovered.

As well, any commit can usually be recovered but anything that was never committed will most likely be lost forever.

```
$ git checkout -- README.md
```

Select Previous Commit

To select a previous commit to revert to, we need the hash of the commit:

```
$ git log  
$ git checkout <HASH> file1
```

This can be used forwards or backwards, ie. you can also "revert" to a commit that later than your current version.

You can also revert several files at the same time

```
$ git checkout <HASH> file1 file2
```

Questions?

Remote Repositories

References

- Chacon and Straub: Chapter 2
- Timbers: Chapter 12.5-12.6

```
$ git remote
```

Remote repos are versions of our projects that are hosted on the internet or some network. This allows us to collaborate with others outside of our local repo.

We can see the remote servers we've configured using `git remote`. If we add the option `-v`, we can see the URL:

```
$ git remote -v
```

Cloned repos will be displayed as origin by default.

Remote Setup

Before we connect our local repo to a remote repo, we need to setup our permissions. This is so we can send and retrieve work to and from our remote repositories. There are two ways to do this:

1. Access Tokens
2. SSH

Access Tokens

 left

 right



center

SSH

```
$ ls -al ~/.ssh
```

If SSH has not been set up on your computer, you should see something like:

```
ls: cannot access '/c/Users/rachaellam/.ssh': No such file or directory
```

Otherwise you'll see filenames `id_ed25519` and `id_ed25519.pub` OR `id_rsa` and `id_rsa.pub` which represent your public and private keys.

```
$ ssh-keygen -t ed25519 -C "rachael.lam@mail.utoronto.ca"
```

Use the code above but with your email. This will output:

```
Generating public/private ed25519 key pair.  
Enter file in which to save the key (/c/Users/rachaellam/.ssh/  
id_ed25519):
```

Press **enter** to use the default file.

You will then be prompted to add a passphrase. You cannot reset this passphrase, so be sure to remember it or write it down somewhere safe:

```
Created directory '/c/Users/Vlad Dracula/.ssh'.
Enter passphrase (empty for no passphrase):
```

It will then ask you to reenter the passphrase:

```
Enter same passphrase again:
```

You will then get a confirmation with a random piece of art at the end. It will show the private key (*identification*) which you should never share, the *public key* and the *key fingerprint* which is a shorter version of the public key.

```
Your identification has been saved in /c/Users/rachaellam/.ssh/  
id_ed25519
```

```
Your public key has been saved in /c/Users/rachaellam/.ssh/  
id_ed25519.pub
```

The key fingerprint is:

```
SHA256:SMSPIStNyA00KPxuYu94KpZgRAYjgt9g4BA4kFy3g1o  
rachael.lam@mail.utoronto.ca
```

Now we can check that we have the public and private key files:

```
$ ls -al ~/.ssh
```

It's time to give GitHub our public key so let's read the public key file and copy it:

```
$ cat ~/.ssh/id_ed25519.pub
```

Output:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NPN7AAAAIDmRA3d51X0uu9wXek559gfn6UFNF  
69yZjChyBIU2qKI rachael.lam@mail.utoronto.ca
```

Copy the long public key to add to GitHub.

Settings --> SSH and GPG keys --> New SSH key

Add a title like `rachael's key` and paste the public key then click *Add SSH key*.

Finally, we can check that it's been authenticated:

```
$ ssh -T git@github.com
```

remote add

To add a remote repo, we can use `git remote add` followed by the name and URL. Now we can connect our local repo to a remote repo:

```
$ git remote add origin https://github.com/rachaellam/git-r.git  
$ git remote -v
```

After checking we'll see:

```
origin https://github.com/rachaellam/git-r.git (fetch)  
origin https://github.com/rachaellam/git-r.git (push)
```

If we want to see more information about a remote repo, we can use the command:

```
$ git remote show origin
```

Here we can see the URL that we're fetching and pulling from, our remote branches, and configurations for git push (to the main branch or another).

To send and retrieve work between our local and remote repositories, we have to authenticate a personal access token:

 left right



center

Questions?

\$ git fetch / \$ git push

When collaborating with others, changes might be made that are important to copy to your local directory. `git fetch` will get any new changes but it won't merge it to our work or modify our work.

```
$ git fetch origin
```

`git pull` will automatically fetch and merge a remote branch to our current branch (more on branching later). It's a good practice to pull before every work session, especially when working with others. Otherwise, a collaborator might have made changes, and you won't be able to push your changes to GitHub.

```
$ git pull
```

If we've create our remote repository using `init` and `remote add`, we need to specify the remote that we want to pull to and the branch we want to pull from.

```
$ git pull origin main
```

`origin` being the name of the remote repo we created earlier and `main` being the main branch on our GitHub repo.

Questions?

```
$ git push
```

When we're ready to share our modifications, we have to push our project and files upstream using `git push`

```
$ git push origin main
```

Here we're pushing to our origin server on your main branch. The main branch is sometimes called the master branch.

This command only works if we have write access and if no collaborator is pushing upstream at the same time as we are. We'd have to instead pull and merge their work before pushing our own.

Questions?

Git Branching

References

- Chacon and Straub: Chapter 3
- Timbers: Chapter 12.8

Branching allows us to diverge from the main line to do work without accidentally messing with the main line. This helps with testing without making any accidental changes to the working branch.

To understand how branching works, let's go back and understand how Git saves files.

- blob
- tree
- pointer



center

A branch is a way to move different pointers to a specific commit. In Git, the default branch is named *master* or *main*. When we first start making commits, we start at the master branch that automatically points to the last commit made.

 center

```
$ git branch
```

We can make a new branch which creates a new pointer for us to move around. We can do this by using the command `git branch` :

```
$ git branch testing
```

Here, we've created a branch called testing, which means we've created a new pointer that could point to our current commit.

 center

```
$ git checkout
```

Git tracks what branch we're on using a pointer called `HEAD`. If we move the `HEAD` to the branch *main*, we'll see:

```
Already on 'main'
```

To move `HEAD` to point to the testing branch that we just created, we use `git checkout`:

```
$ git checkout testing
```

and we should see..

```
Switched to branch 'testing'
```

 center

If we make some changes to our testing branch and commit, our head will move with the new commit.

 center

If we want to go back to an older version of our project and make changes, we can use `git checkout` again to redirect the head back to our master branch:

```
$ git checkout main
```

Using this command will move the `HEAD` pointer back to our master branch and revert our files in our working directory back to the snapshot that the master branch points to.

Questions?

Branching and Merging

Let's take a look at a workflow that you might encounter:

```
$ git commit -m "commits to master branch"
```

 center

```
$ git checkout -b iss53
```

 center

```
$ git commit -a -m "commits to iss53"
```

 center

```
$ git checkout master  
$ git checkout -b 'hotfix'  
$ git commit -m "commits to hotfix"
```

 center

```
$ git checkout master  
$ git merge hotfix
```



center

```
$ git merge
```

In the last step we saw a command called `git merge`. Once we've committed changes and are ready to deploy, we can use `git merge` to merge our working branch back into our master branch.

```
$ git merge testing
```

 center

We can then delete the branch that we've created, as the master branch points to the same place.

Adding the option `-d` will delete the branch that had been merged with the main, as we no longer need it.

```
$ git branch -d testing
```

Remember that changes to our master branch have not been added to our *iss53* branch. We either need to `pull` them in or wait to integrate them when we `pull iss53` into the master branch



If we're merging a branch with the main that has been changed since we diverged, merging isn't as simple for Git.

Git will create a new snapshot of the merge and automatically create a new commit that points to it, called a `merge commit`.



center

We saw `git branch` earlier with the option `-d` to delete a branch, but to get a list of our current branches, we can run `git branch` without any arguments.

```
$ git branch
```

The `*` indicates the branch we are currently on or have checked out (`git checkout`)

If we run `git branch` with the option `-v`, we can see the last commit on each branch. This is another reason why comments are so important to add to our commits: they can be extremely useful when looking back at our work and seeing what we've done.

We can also add the options `--merged` or `--no-merged` to `git branch`. `--merged` allows us to see what branches been merged to the branch we're currently on. Branches without the `*` are generally safe to delete because we've already merged our work with our main branch.

```
$ git branch --merged
```

On the other hand, `--no-merged` allows us to see all the branches that haven't been merged.

```
$ git branch --no-merged
```

If we try to delete one of these branches, we will receive an error. We can force delete using the option `-D`.

Merge Conflicts

Often times, merging our work with other topic branches or the main branch creates errors.

For example, if we've changed the same part of the same file differently in the two branches we're merging, we will encounter a conflict.

Luckily, Git helps us see where the error is to correct it.

Git shows us the beginning of the merge conflict with
`<<<<< HEAD` and the end with `>>>>>`.

===== separates the differences.

To fix the merge, you can choose one set of changes, the difference you prefer or re-write it entirely. You have to remove all identifiers of the merge conflict as well.

Questions?

Branching Workflow

Long-Running Branches

Multiple long running branches are helpful when tackling large and complex projects.

Typically, developers will keep the master branch as the stable branch or code that has been or will be released. They will then have parallel branches that are used for development and testing.

Branches can also have various levels of stability, and will graduate/merge branches once they're fully tested.



center

Topic Branches

Topic branches are short-lived branches that are created for a particular feature or related work. They allow us to quickly switch between to.../pics and keep changes there for as long or as little as needed, regardless of the created or modified order, before merging.

 left  right

Questions?

Remote Branches

Remote branches are pointers to the state of branches on our remote repositories. Our remote repositories can have multiple remote branches, just as we can have multiple branches on our local repositories.

The format is `(remote)/(branch)` or `(remote) (branch)`

If branches already exist on your GitHub repo, you will have access to these branches. If we're working with a branch that does not exist yet, we can push it to our remote repo.

Pushing

When we're ready to share our work, we'll use `git push`. If the remote branch already exists, we can push directly to that branch:

```
$ git checkout testing  
$ git add -A  
$ git commit -m "testing branch commit"  
$ git push origin testing
```

This will push our changes to the existing testing branch on GitHub.

If we were working with a branch that only exists locally, we can push it to GitHub with a slight tweak:

```
$ git checkout new-branch  
$ git add -A  
$ git commit -m "new branch commit"  
$ git push origin main:new-branch
```

This will create a new branch on GitHub called `new-branch`. From here, if we want to continue updating this branch, we can just run `git push origin new-branch`.

Fetching

When we `fetch` or `pull` files from our remote repos, we don't automatically have access to local, editable copies of files of the remote branches.

We can do this in several steps. First we're going to fetch the remote branches:

```
$ git fetch
```

We can then see what branches exist remotely:

```
$ git branch -v -a
```

And we'll see something like this:

* main	3d850f2 a commit
remotes/origin/HEAD	-> origin/main
remotes/origin/main	3d850f2 another commit
remotes/origin/testing	3d850f2 another committ

Then we'll create a branch that exists on our local drive:

```
git checkout -b testing origin/testing
```

Here we're pointing the `HEAD` to the new branch (`-b`) called `testing` from `origin/testing`

Tracking Branches

Tracking branches are branches that have a direct relationship with a remote branch. We can `push` and `pull` to and from these branches, as Git automatically knows which server and branch we're working with.

For this to work, the name of your local branch must be the same as the remote branch

If the branches are named differently, we must run a different command for the push to be successful:

```
$ git push origin HEAD:remote-branch
```

Deleting Branches

If we've merged all our changes into our main branch, we can delete the remote branch with the following code:

```
$ git push origin :testing
```

Questions?

Collaborating

References

- Chacon and Straub: Chapter 3 + 5
- Timbers: Chapter 12.8

Much of the work that we do will involve working with others. It's important that we learn how best do this so we can successfully collaborate and avoid conflicts where possible. If conflicts arise, good collaboration practices help us resolve them with ease.

So far we've learned several practices and commands that help us collaborate with others, including remote repositories and branches, `git pull` `git push` and `git merge` but we'll learn more practices that make collaboration straightforward.

There are many different factors that influence what workflow you might follow and how you might contribute to a project including:

1. Active contributor size

Teams can vary from a few collaborators to thousands, varying the number of commits per day.

2. Chosen workflow

Each project could have a different process to check patches including an integration manager or peer reviews.

3. Commit access

Policies regarding how to contribute work can differ between projects, even by how much work or how often.

Let's take a look at a couple possible workflows:





center

GitHub

Adding Collaborators

To collaborate with others on our GitHub repo, we can add collaborators so they have direct access to the repo:

 center



center



center

Access does not have to be permanent. We can remove collaborators at any time and add additional ones when needed.

Granting access to your repo this way, enables collaborators to make changes and push them to the repo without our constant permission. If we do not add push access, collaborators have to fork the repo and create pull request.

Forking Projects

Forking allows us to collaborate on projects without push access. We can fork a public project on GitHub and then clone it into our local server to begin making changes.

 center

Once a project has been forked, we can find the repo in our GitHub repositories. We can then clone the repo (`git clone`), make changes and push our changes without altering the original repo.

Alternatively, we can clone the original repo, make our changes, fork the original repo and then merge our branch to the master branch of the forked repo.

If we're collaborating with someone and we want our changes to be merged to the original repo, we can create a pull request.

Pull Request

After making a few changes, we now want to create a pull request to merge our changes with the original repo. We can do this directly in GitHub:

 center

To the pull request, we can see what branches and repos we're attempting to merge:

 center

We can also see the changes that were made:

 center

GitHub will also check to make sure that there are no conflicts with the base branch:



Pull requests with no merge conflicts are easy to merge into the branches but it gets more complicated if there are merge conflicts:

 center

You can still create a pull request with merge conflicts:

 center



center

To resolve conflicts, it's very similar to merging conflicts through terminal:



Because resolving conflicts is done on GitHub, it's a good practice to resolve conflicts before creating a pull request.

Questions?

Conflicts

References

- Chacon and Straub: Chapter 3 + 6
- Timbers: Chapter 12.5

Conflicts are going to arise at some point, especially when working with others. It's important that we learn how to handle these conflicts for easier and more successful collaboration.

GitHub Issues

GitHub issues are an extremely useful tool for communicating decisions, ideas and problems that are project specific.

They are an alternative to email or Slack that keep communication isolated to a particular project.

Issues can be *opened* on GitHub and even when they're *closed*, they remain available. They're also accessible to all collaborators for transparency.

To open an issue, navigate to the project page and click */issues*:



Then open a new issue:

 center

From here, we can add a title and description of the issue, and add any specific collaborators, labels, etc.

 center

Information

Title: should be descriptive and quickly convey what the issue is about

Description: explain the purpose of the issue and how to potentially resolve it. If it's a bug fix, include a reprex, what you wanted to happen and what actually happen. You can also include steps already taken to solve the issue.

Reprex

- A reprex is a REPRoducible EXample.
- It contains just enough of the code to reproduce the error, ie. it is self-contained
- We might have to create a smaller version of the code in order to create the reprex. Don't include anything that isn't related to the problem.
- Sometimes, this process will help us solve our issue.

Inclusions

A minimal dataset to demonstrate the problem. This could be a regularly used one such as *iris*

```
install.packages("dplyr")
library(dplyr)
head(mtcars)
```

or one easily built yourself.

```
df <- data.frame (col1 = c(1, 2),
                  col2 = c(3, 4))
df
```

- Make sure to include classes that are necessary to your reprex (ex. dates, factors, etc.)
- If you're using randomly sampled data, set the seed to so the same data is produced each time.

```
set.seed(853)
```

Include all packages that you need.

- Make sure they are placed at the top of the script so it's quick and easy to see what is necessary for the reprex.

Other Inclusions

- Details about the issues you are facing.
- Comments that will add clarification to your error.
- Add what fixes have been attempted. This could include pages to StackOverflow articles that you've viewed.
- Communicate clearly what you're desired outcome is.

Task Lists

If an issue is quite large, it's possible to add tasks lists to break the issue into smaller pieces.

- Use square brackets – []
- To mark it complete, use – [x]
- Issues can be linked to previous issues using
 - the number – [x] #11
 - a URL – [x] <https://github.com/rachaellam/git-issues/11>

Once an issue has been opened, we can respond and comment.

When we decide it has been resolved, we can close the issue.

The history of the issues can still be seen, even if it has been closed.

Questions?

Debugging

File Annotation

File annotation can help us resolve issues in our code if we know where the issue is. We can see when the code was introduced and by whom, line by line, using the aptly named `git blame`.

```
$ git blame -L 1,3 script.sh
^8e9b89da (Rachael Lam 2021-12-02 15:01:02 -0500 1) #line 1
8e9b89da (Rachael Lam 2021-12-02 15:01:02 -0500 2) #line 2
8e9b89da (Rachael Lam 2021-12-02 15:01:02 -0500 3) #line 3
```

`git blame` is combined with the filename we want to inspect.

We can also use the option `-L` followed by two numbers to limit the number of lines shown.

We can then see the partial SHA-1 of the commit that last modified the line, the author name and date of the commit, and the content of the file by line.

When the SHA-1 is preceded by a `^`, it indicates that those commits were when the file was first added to the project and have not changed since.

Binary Search

If we don't know where the issue is, we can use `git bisect` to get identify the commit that introduced an issue.

```
$ git bisect start  
$ git bisect bad  
$ git bisect good [good_commit]
```

First, we've started the bisect program. We then told the system that the current commit is broken using `bisect bad` followed by the last good commit using `bisect good [good_commit]`. We can see the different commit if we run `git log` that we learned earlier.

Git produced the number of commits that were between the good and the bad commit and then checked out the middle one. From here, we can run our test to see if the issue still exists. If it does, it means the issue was introduced in a commit before this middle commit and we can run `git bisect bad` to tell the system that there is still an issue.

If it does not, then the issue was introduced after and we can run `git bisect good`.

We can keep running this loop until we find the commit that introduced an issue and make our corrections.

When we're finished, we can run `git bisect reset` to reset our `HEAD` to where we were before we started.

Best Practices

- Topic branches should be used to try out new code before integrating. They enable us to play around or leave for the time being it if it's not working.
- Commit often rather than submitting a massive commit. This makes it easier to review and merge changes, or revert if necessary.

- Create quality commit messages so that your collaborators can easily understand what has been done. For example:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body, the blank line separating the summary from the body is critical (unless you omit the body entirely).

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space with blank lines in between, but conventions vary here

Questions?

Reproducibility

- Reproducibility is the ability for independent researchers to obtain the same or similar results when repeating an experiment or test.
- This concept has been widely used in natural sciences, but is not yet as popular in data science.
- Remember, data science is a science. We question, hypothesize, test, and therefore, we should also have the same rigour of confirmation.

- Skepticism should always be able to be independently verified. We should be able to defend our results and decisions.
- Who would believe your results otherwise? More importantly, you should not believe results if they cannot be verified.

Why is reproducibility important?

1. New Insights
2. Reduce Error Risks
3. Validate Results
4. Transparency

How can we make our work reproducible?

There are a number of practices that can help make our work reproducible including:

- Reproducible Examples
- Commenting Code
- Technical Documentation
- Folder Structure

Reproducible Examples

Reprex

- A reprex is a REPRoducible EXample.
- It contains just enough of the code to reproduce the error, ie. it is self-contained
- We might have to create a smaller version of the code in order to create the reprex. Don't include anything that isn't related to the problem.
- Sometimes, this process will help us solve our issue.

Inclusions

A minimal dataset to demonstrate the problem. This could be a regularly used one such as *iris*

```
install.packages("dplyr")
library(dplyr)
head(mtcars)
```

or one easily built yourself.

```
df <- data.frame (col1 = c(1, 2),
                  col2 = c(3, 4))
df
```

- Make sure to include classes that are necessary to your reprex (ex. dates, factors, etc.)
- If you're using randomly sampled data, set the seed to so the same data is produced each time.

```
set.seed(853)
```

Include all packages that you need.

- Make sure they are placed at the top of the script so it's quick and easy to see what is necessary for the reprex.

Other Inclusions

- Details about the issues you are facing.
- Comments that will add clarification to your error.
- Add what fixes have been attempted. This could include pages to StackOverflow articles that you've viewed.
- Communicate clearly what you're desired outcome is.

Commenting Code

How does commenting code help in reproducibility?

Commenting code is an important practice that benefits both ourselves and collaborators.

Not only can we understand what we did to fix our own errors or improve our work, but others can better understand our code to reproduce it.

[Ellen Spertus](#) outlines 9 rules to follow:

1. Comments should not duplicate the code
2. Good comments do not excuse unclear code
3. If you can't write a clear comment, there may be a problem with the code
4. Comments should dispel confusion, not cause it

5. Explain unidiomatic code in comments
6. Provide links to the original source of copied code
7. Include links to external references where they will be most helpful
8. Add comments when fixing bugs
9. Use comments to mark incomplete implementations

1. Comments should not duplicate the code

- Comments should add value to whoever is reading your code.
- Duplicating code adds unnecessary bulk and can actually make it more difficult to understand the code.

Can you think of a bad example?

Here is an example of what you should not do:

```
x=5

if [ $x = 5 ]; then
    echo "x equals 5." # if x = 5 then ouput x equals 5

else
    echo "x does not equal 5." # otherwise output x does not equal 5

fi
```

2. Good comments do not excuse unclear code

- Our aim should always be having clear code, rather than relying on our comments to add clarity.
- Remember, we should not be adding more bulk to the code that makes it more difficult to understand.

3. If you can't write a clear comment, there may be a problem with the code

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

- Kernighan's Law

4. Comments should dispel confusion, not cause it

- If our comments are adding further confusion, we should either rewrite the comment or remove it entirely.
- A could comment should always be written with the intent to help better understand what is being done.

5. Explain unidiomatic code in comments

- If we've purposefully written code that others may find unnecessary, we need to comment our reasoning.
- Others may try to simplify our code if we don't explain our reasoning.

Can you think of an example?

6. Provide links to the original source of copied code

- Often times, we'll use code that others have written. It's important to give credit to the original source, but as well as give us a reminder as to where we got the code to reference it later if we need.
- Referencing the source can also provide other information such as what the problem was, why the solution was recommended and how it can be improved. It also means, we don't have to comment all of these details again in our own code.

An example:

```
# I got these 9 rules from Ellen Spertus' blog post on  
# StackOverflow: https://stackoverflow.blog/2021/12/23/  
# best-practices-for-writing-code-comments/
```

- It's best to include the URL so other's don't have to search for the exact location.
- Remember: never copy code that you don't personally understand.
- Code from StackOverflow falls under Create Commons licenses so a reference comment is needed.

7. Include links to external references where they will be most helpful

- References don't just have to be used for copied code. They can also provide information on decisions made or changes in practices

8. Add comments when fixing bugs

- Comments can help others understand what we modified, if the modification is still needed, and how to test our modifications
- Although `git blame` can be used to find the commit that modified the code, a good comment can help locate the change and are quite brief.

9. Use comments to mark incomplete implementations

- Sometimes we have limitations in our knowledge or time. Adding code documenting these limitations can allow us to better address and fix the issues.

Some other good practices:

- Comments should be clear and efficient. Don't add more information than necessary, but don't be too vague
- Remember to update your comments if you update your code. Old comments can add more confusion.
- Inline comments can add noise as they're mixed with our code. Spacing can be helpful here:

```
colors = [[213/255, 94/255, 0],           # vermillion
          [86/255, 180/255, 233/255],      # sky blue
          [230/255, 159/255, 0],           # orange
          [204/255, 121/255, 167/255]]    # reddish purple
```



Code tells you how, comments tell you why.

- Jeff Atwood, Co-founder of StackOverflow

Technical Documentation

Writing

What is technical documentation writing?

Why is it important to write a good technical documentation?

Technical documents are necessary for reproducibility as they relay important information about your project to others. Writing technical documents is not easy but should not be overlooked. A well done technical document will communicate the goals of a project and in doing so, can generate interest in the project.

GitHub outlines several pieces of information to include:

1. What the project does
2. Why the project is useful
3. How users can get started with the project
4. Where users can get help with the project
5. Who maintains and contributes to the project

This is just part of the story and we'll add more to this in the coming slides.

README

- Technical documentation writing is typically found in a `README.md` file.
- If the `README.md` file is placed in our repo's root, `doc` folder, or hidden in the `.github` directory, GitHub will place the contents of the `README.md` on the main repo page.
- The `README.md` file will be the first thing visitors see when they come to the project page so it's important to make it as appealing as possible.

Examples

Let's walk through some good examples of `README.md` files:

- [Create Go App CLI](#)
- [Human Activity Recognition](#)
- [Markdownify](#)
- [More!](#)

What did you like about these README files?

What similarities can you see?

What should be included?

1. Name of the project
2. What the project does
3. The project's usages
4. How to get started
5. Where to find help
6. Who contributes

1. Name of the Project

- The name of your project should be unambiguous.

2. What the project does

- This should be a description of the project.
- Provide context to the project and any reference links.
- Include features or background information
- *Can be titled "Description"*

3. The project's usages

- This should include how the project can be used.
- Provide examples using the code along with the expected output of said code.
- It should be a smaller example. Longer examples can be linked to.
- *Can be titled "Usages"*

4. How to get started

- This is the installation guide.
- Think of your particular audience and how much detail you might need to include.
- Add a requirements section if there are specific dependencies or needs to run in a particular programming language.
- *Can be titled "Installation"*

5. Where to find help

- Direct people on where to find help if they need.
- This could be the issues page on GitHub, a forum, or an email address.
- *Can be titled "Support"*

6. Who contributes

- This should outline how others can contribute to your project and what your requirements are for accepting contributions.
- *Can be titled "Contributing"*

Additional Additions

- Visuals: Visuals can grab people's attention, but they can also be helpful for showcasing what the code does. Include screenshots or GIFs that demonstrate your project.
- Badges: Badges provide metadata such as issue tracking, test results and downloads. [Shields.io](#) provides this service and you can also look at their [GitHub](#) for more information.
- Acknowledgements: Include the authors or anyone that helped with the project.

Markdown

- As noted by the extension, `README.md` files are usually written in markdown, thus using markdown syntax for styling.
- [GitHub](#) provides a good reference on how to write your README in markdown.

Headings

```
# Largest Heading  
## Second Largest Heading  
### Third Largest Heading
```

 center

Text Styling

```
bold  
*italic*  
~~strikethrough~~  
this is a *nested* example  
*bold and italic*
```

 center

Quoting

> Block quote some text

 center

Unordered Lists

- this is an unordered list
- second item
 - nested
 - second nest

 center

Ordered Lists

1. This is an ordered list
2. This is the second item
 - with some additional information
3. This is the third

 W:1000 center

Codeblock

Wrap your code in ` `` to create a codeblock.

 W:1000 center

Links

[Rachael's GitHub] (<https://github.com/rachaellam>)

 W:1000 center

Images

```
! [w:1000 center](../pics/picture.png)
```



As we see, images can also be GIFs. We can also play around with the size and alignment.

Folder Structure

What is folder structure and why is important?

A good folder structure is important for reproducibility because it easily allows for others to navigate and implement our projects. If someone references a file that is self contained, they know they won't have to change the file path to gain access.

For example, what is the difference between these two paths:

1. "/Users/rachaellam/Documents/all-projects/this-project/data/"
2. "this-project/data/"

Folder structure can vary based on the project but a basic one to follow is...

- /inputs
 - Everything that will not be edited including raw data and references
- /outputs
 - Everything that was created during the project and your results
- /scripts
 - All code that was written for the project

Wilson et. al also outline a file structure that is similar...

- /doc
 - All text documents including documentation or references
- /data
 - All raw data and metadata
- /results
 - Files generated during the analysis including generated data or cleaned data
 - Results can be further divided into subdirectories that contain intermediate files and finished files
- /src
 - All code that was written for the project

References

Reproducibility:

- Reproducibility and Research Integrity
- Reproducibility, Replicability, and Reliability

Commenting:

- Elena Kosourova
- Ellen Spertus

Technical Documentation Writing:

- GitHub README
- GitHub Markdown
- KyuWoo Choi
- Make a README
- Matias Singers

Folder Structure:

- Rohan Alexander
- Wilson et. al