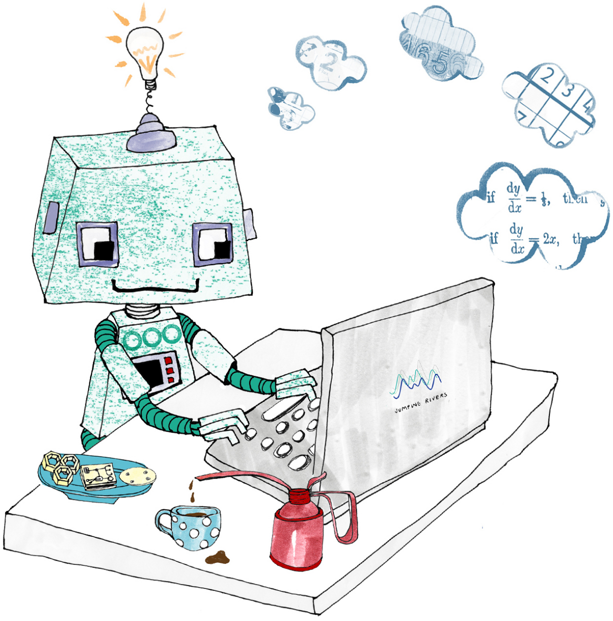
GIT FOR ME

jumping rivers

jumpingrivers.com

CREATING CLARITY WITH YOUR DATA.

Saving organisations like yours thousands of work hours.



Jumping Rivers is an analytics company whose passion is data and machine learning. We help our clients move from data storage to data insights. Our trainers and consultants come with over 40 years combined experience in R, Python, Stan, Scala and other programming languages.

Consultancy

- RStudio certified - one of only seven full-service partners.
- We have plans for managing all RStudio products. From on-demand support to full care packages.
- As an RStudio reseller, we can provide free consultancy.
- Experts in both data science and data engineering.
- Creation of bespoke dashboards and Shiny apps.
- Optimisation of your algorithms.

R Studio®

Training

- R/Python/Stan/Scala
- The Tidyverse
- Machine Learning and Tensorflow
- Dashboards with Shiny
- Statistical Modelling
- And many more!

jumping rivers
jumpingrivers.com
info@jumpingrivers.com
@jumping_uk

1 Introduction

1.1 *Why do we need Git?*

Do any of these file names look familiar to you?

```
code_v1.R
code_v1b_add_plot.R
code_v1b_add_plot_FINAL.R
code_v1b_add_plot_FINAL_FINALv2_sept2020.R
code_v1b_add_plot_FINAL_FINALv2_sept2020_Maria_edits.R
```

If so, you need a version control system such as Git! What are the benefits?

- Backups of your files are stored remotely using an online host such as GitHub or GitLab.
- If you are working in a team, backups are stored on multiple locations.
- Git tracks the changes you make to files, so you have a record of what has been done, and you can revert to specific versions should you ever need to. This is called version control.
- Git allows changes by multiple people to be merged into a single source, thus improving collaborative working.

1.2 *What is Git?*

Figure 1.1: `https://git-scm.com/`

Git is a version control system[1]. It's designed to help teams work together on projects simultaneously. Git tracks the files in your project - called a repository - in a structured way. It was designed to deal with anything from small projects to big reports[2] - with great emphasis being on speed and simplicity of editing. To put it simply, we use Git to track the changes we or others make to files. This becomes particularly advantageous when more than one person is working on the same project and we need to edit someone else's work. Git also ensures that we are not at risk of losing our work if our local machine breaks or crashes. It does so by ensuring that we have a copy of all files stored in a remote, virtual machine - sort of like e-mailing yourself an attachment so you can access it on another device.

1.3 *What are GitHub and GitLab?*

A simple analogy is that to when we use *e-mail*, we sign up to Gmail or B.T. We can, in theory, host our own e-mail server, but typically that's too

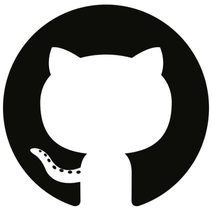


Figure 1.2: The GitHub logo.

[1] There were version control systems before Git. For example, svn and cvs.

[2] Git is used on massive software projects that may have hundreds of developers.

much hassle and overkill. Git is like *e-mail*, and GitHub/GitLab are the Gmail in this example. We don't *have* to use GitHub/GitLab, but they offer convenient platforms that remove the hassle of maintaining[3] our own Git server.

GitHub and GitLab are some of the many companies[4] that offer the service of hosting your repositories. Your repository is stored online, like a backup and you can download a fresh copy, any time you like. This allows other people to see your files, sync up with you, and perhaps even make changes. Both services offer a variety of public and private repositories.

For today we're going to focus on GitHub/GitLab. You'll need an account set up in order to do the practicals.

Figure 1.3: The GitLab logo.

1.4 *Using Git inside RStudio*

The RStudio IDE has a Git graphical user interface (GUI) panel in the top right corner. Don't worry if you can't see it yet, it only appears when you are working on a project which has Git associated with it. When we click on a button in the Git GUI, it runs a Git command in the *terminal*. Using the RStudio Git GUI is a great way to learn Git, as it allows us to get used to the workflow without having to remember a long list of commands.

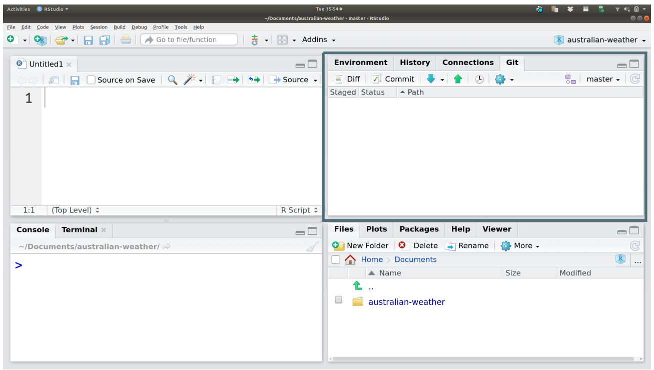


Figure 1.4: The Git tab in RStudio.

1.5 *Installing Git*

Installing Git, setting it up and linking Git with RStudio can be a little painful at times. It's something you only have to do once. Hopefully before attending this course you *should* have everything installed and ready to go, however, if you still need support, or you want help setting up in future, this tutorial is the best we've found:

`https://happygitwithr.com/install-git.html`

1.6 *Introduce yourself to Git*

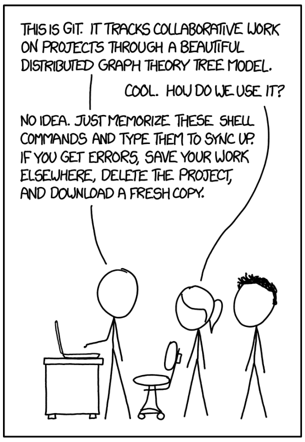
So, we've got a GitHub account, and we've installed Git on our computer. First we need tell Git who we are. This is so that Git can associate a name with any commits that we make. We need to tell Git our name and our e-mail address. The easiest way to do this is with the `use_git_config()` function in the **usethis** package.

[3] Many institutions run their own server for security reasons. For example, banks and certain government departments would typically host their own Git server.

[4] Other popular companies include Bitbucket, SourceForge, Launchpad and Apache Allura just to name a few.

```
library("usethis")
use_git_config(user.name = "Maria Garcia",
               user.email = "maria@example.com")
```

It's a good idea to use the same e-mail that you set up GitHub with. You will only need to do this once on your computer. Now we're set up and ready to start.


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

1.7 *Set up a personal access token*

GitHub now requires users to set up a personal access token (PAT) in order to establish an HTTPS connection between a local machine and a remote repository. A PAT key is essentially an alternative to using passwords for authentication. It is more secure for a number of reasons:

- It is an auto-generated *strong* passphrase (not 'monkey123'!)
- The user can set an automatic expiry date
- The user can set different permissions for each key

A really helpful guide to creating a PAT key can be found here:

`https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token`

For the exercises in this course, we will need to be able to access repositories from the command line. To enable this, you will need to check the box next to 'repo' when selecting the scopes for your token.

When you have generated your token, you should copy it and store it somewhere safe.

2

Getting Started

Working with Git isn't that different from your usual working habits. Let's imagine you're working on a data science project. In your repository you've got some data, some R scripts and maybe an R Markdown file reporting your findings. Usually, as you are working, you might stop and save the files, to update your progress. When working with Git, instead of just saving your files, you save your files ***and*** make a *commit*, which takes a snapshot of all the files in the entire project. Regularly, you *push* commits to your remote repository, such as GitHub or GitLab. This is like sharing a document with colleagues on DropBox or sending it out as an email attachment.

2.1 *What is a commit?*

> Using a Git commit is like using anchors and other protection when climbing. If you're crossing a dangerous rock face you want to make sure you've used protection to catch you if you fall. Commits play a similar role: if you make a mistake, you can't fall past the previous commit. Coding without commits is like free-climbing: you can travel much faster in the short-term, but in the long-term the chances of catastrophic failure are high! Like rock climbing protection, you want to be judicious in your use of commits. Committing too frequently will slow your progress; use more commits when you're in uncertain or dangerous territory. Commits are also helpful to others, because they show your journey, not just the destination.
>
> — R Packages, Hadley Wickham.

A *commit* is like a snapshot of all the files in the repository, at a specific moment in time. The commit will contain information about exactly what changed, who changed it, and why. Every time you make a commit you must also write a short commit message. Ideally, this conveys the motivation for the change.

Note that you have to explicitly tell Git which changes you want to include in a commit. This means that a file won't be automatically included in the next commit just because it was changed. Instead, you need to *add* the files to the *staging area* before committing. We'll see an example of this in Section 2.5.

2.2 *Git terminology*

Here is an overview table of the main Git terms we're going to cover in this chapter. Keep referring back to this if you forget the definition of certain terms!

Term	Simple Description
`add`	Tell Git which files you wish to commit
`clone`	Download a copy of your repository
`commit`	Create a snapshot of the changes
`diff`	Show the difference between two commits
`fork`	Copy someone else's repository
`push`	Send local commits to the remote repository
`pull`	Merge changes in the remote repo to the local directory
`status`	Check the state of Git
`remote repo`	Repository accessed by all team members
`local repo`	Repository located on your machine

Table 2.1: This is a simplified guide. It attempts to be as accurate as possible while avoiding some of the messier details.

2.3 *Creating a Git repository*

There are a couple of different ways that we can start using a Git repository:

- Clone an existing repository[1]
- Create a new repository from scratch

As a repository only needs to be created by one member of the workforce, the most common action out of those two is cloning a Git repository.

Cloning a Git repository

For most work, you'll be cloning an existing repository[2]. When you *clone* a repository, you are downloading a copy of that repository to your laptop. This will download the repository from the GitHub/GitLab to your laptop and extract the latest snapshot of the repository (all the files) to your working directory[3]. By default it will be saved in a folder with the same name as the repository. We'll teach you *how* to clone a repository from GitHub/GitLab in the practical.

Forking a Git repository

When you *fork* a repository, GitHub creates a copy of that repository in your GitHub account. We haven't downloaded anything onto our laptops, we've just simply created a copy of someone else's repository on our personal GitHub account. You would then clone your forked repository. This allows you to make edits to your forked repository, without affecting the original project. Later down the line, it is possible to merge your forked repo back into the original repo[4].

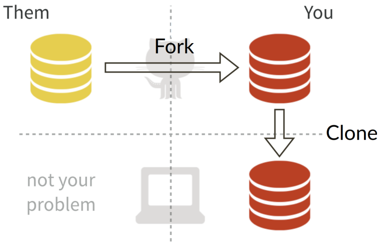


Figure 2.1: Forking and cloning remote repositories.

Forking a project is as easy as clicking the Fork button on GitHub/GitLab. In the practical, you'll be forking a repository that Jumping Rivers created and cloning it on to your laptop. The reason we want you to fork it first, is so that each of you has your own personal copy of our repository. If you all just cloned the repository without forking first, we'd have 15 people all working in the same repository at once which might (definitely) get a bit messy!

[1] Cloning a repository is just like copying.

[2] If we were using the terminal, instead of the RStudio Git GUI, we would type the command `git clone`.

[3] It will also download the previous fifty commits to the repository.

[4] We'll talk more about merging in Chapter 5.

Creating a new repository

There are a couple of ways to create a new repository. Generally the easiest way is to use the user interface on GitHub/GitLab. There is a button to create a new repository on GitHub/GitLab. You'll be asked to give the repository a name and a description. Once it's been created, you can simply clone the new repository onto your laptop, in the same way that we discussed above. Once it's on your laptop you can add some files and start working.

Practical 2 - part 1.

2.4 *Example: Australian weather*

Today we're going to be working on the `jumpingrivers/australian-weather` repository. This is a simple example of a data science project, where we're investigating historical weather data from Australia. You can find this at:

GitHub: `https://github.com/jumpingrivers/australian-weather`
GitLab: `https://gitlab.com/jumpingrivers-public/australian-weather`

Let's have a look at our example project. It's a very simple data science project with just three folders:

- `data/`
- `graphics/`
- `R/`

Inside the `data/`, folder we have the `.csv` file which contains the weather data for Australia. Inside the `R/` folder, we have one R script, `create-plot.R`, which reads in the data, and creates a simple scatter plot for Brisbane. The scatter plot is then saved in the folder `graphics/`.

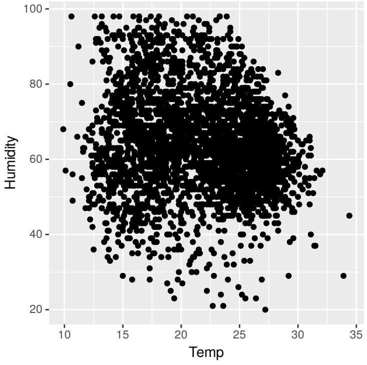
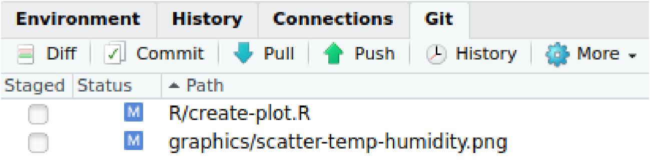


Figure 2.2: Temperature and Humidity for Brisbane at 9am daily.

Here is the code in `create-plot.R`

```
library("tidyverse")
df = read_csv("data/weatherAUS.csv")

# Create a scatter plot of Humidity9am and Temp9am
# for a specific location
city = "Brisbane"

scatter_plot =
df %>%
  filter(Location == city) %>%
  ggplot(aes(x = Temp9am, y = Humidity9am)) +
  geom_point()

# Save the plot in graphics/
ggsave(filename = "graphics/scatter-temp-humidity.png",
       plot = scatter_plot)
```

Let's say that instead of creating the scatter plot for Brisbane, we want to create a scatter plot for Sydney. To do that, we would need to replace `city = "Brisbane"` with `city = "Sydney")`, and re-run the script to save the



Staged	Status	Path
☐	M	R/create-plot.R
☐	M	graphics/scatter-temp-humidity.png

Figure 2.3: To get an overview of which files have changed, we can look at the status of each file in the Git tab.

new scatter plot.

Example: Spot the difference

We've changed some files, specifically

- `R/create-plot.R`
- `graphics/scatter-temp-humidity.png`.

Git examines all the files in our repository to see if we have made any edits to the project[5] that have not yet been committed.

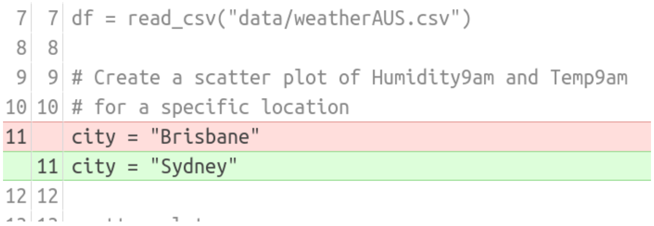
Any time we make changes to a file, Git notices and keeps a track of the changes we've made. Once a file has been edited and saved, it will appear in the Git tab, with a status symbol next to it. There are a number of different status symbols, these are described the table below.

Symbol	Description
A	file has been added
C	file has been copied
D	file has been deleted
M	file has been modified
R	file has been renamed
?	file not tracked by Git
U	two files have been unmerged

This M symbol only tells us that the file has been modified. To see exactly **how** the file has been modified we can use the `diff` button. The `diff` button[6] allows us to inspect the difference between the most recent version of the file tracked by Git, and the current version you are working on.

- Green lines indicate lines which are new.
- Red lines indicate lines which have been removed.

For binary files, such as image files or Word documents, `diff` is not appropriate.



```
df = read_csv("data/weatherAUS.csv")

# Create a scatter plot of Humidity9am and Temp9am
# for a specific location
city = "Brisbane"
city = "Sydney"

```

Figure 2.4: The diff for our R script.

2.5 *Staging and committing*

We have modified the R script, and the plot, and they currently have a the `Modified` symbol (M) beside them. If we have made all the changes we wish to make then using Git lingo, we say that these files are "ready to be staged".

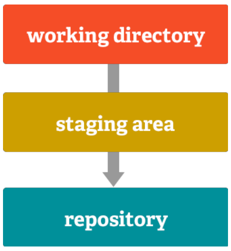
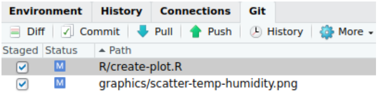


Figure 2.5: The Git staging area: `https://git-scm.com`.

Before we can commit our changes, we first have to tell Git *which* files we want to commit. We do this by adding our files to the *staging area*. We can think of the staging area as a box. Any files that you put in the box are being collected together ready to be committed. To add[7] a file to the staging area in RStudio, all we have to do is tick the checkbox corresponding to a particular file, under the "staged" header, as in Figure 2.6.



Staged	Status	Path
☑	M	R/create-plot.R
☑	M	graphics/scatter-temp-humidity.png

Figure 2.6: Two files have been staged, ready to be commited.

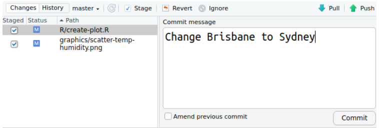
[5] Anything from changing a file name to adding a completely new file qualifies as an edit.

[6] Terminal: `git diff`.

[7] Terminal: `git add`.

Once we have made changes that have been staged for a commit, we can actual commit them by pressing on the `Commit` button. An RStudio pop-up window will appear, prompting you to enter a commit message. An example of this is shown in Figure 2.7. In Figure 2.7 we can see the files that we have staged to commit. We now go ahead and commit the files, writing a brief message to document what you have changed.

Terminal: `git commit -m "Change Brisbane to Sydney"`.



Change Brisbane to Sydney

Figure 2.7: The pop up window used to complete a commit.

2.6 *Stay up-to-date with pull*

Before we start push our changes, we need to make sure that we have the most recent version of the project on our computer. This is important as our team mates might have made changes since you cloned the repository. The way to ensure you acquire the most up-to-date project version is to click the **pull** button[8] as seen in Figure 2.8. If there have been changes in the remote repository, Git will update your local repository in two steps.

1. Fetch/download any new changes from the remote repository
2. Merge the new changes into your local repository

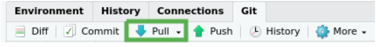


Figure 2.8: The Pull button in the Git tab.

Most of the time Git is clever enough to automatically merge the remote changes with our local changes and everything works seamlessly. However, sometimes a *merge conflict* can arise. This occurs when the same line of a file has changed in both the remote and the local, and Git doesn't know which commit should take precedence. When a merge conflict occurs, the user has to manually edit the file, to determine the correct change. We'll cover this in greater detail in the Chapter 4.

2.7 *Share your changes with push*

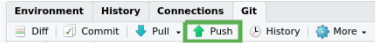
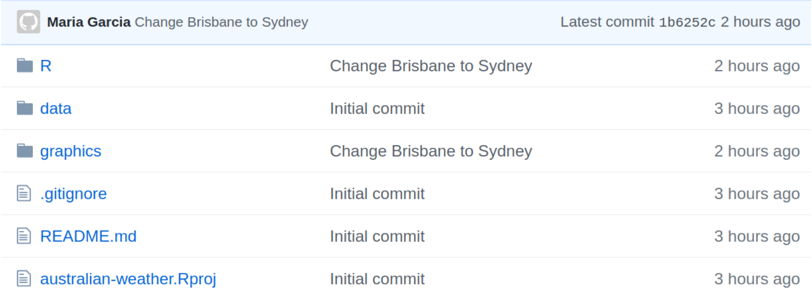


Figure 2.9: The push button in the Git tab.

Once you've made changes, staged and committed them with a useful message, and pulled remote changes, all that's left to do is to push your changes back up to the remote repository. If you look at the Git tab in RStudio you'll probably see a message like the one below:

```
Your branch is ahead of 'origin/main' by 1 commit.
```



Maria Garcia Change Brisbane to Sydney		Latest commit 1b6252c 2 hours ago
R	Change Brisbane to Sydney	2 hours ago
data	Initial commit	3 hours ago
graphics	Change Brisbane to Sydney	2 hours ago
.gitignore	Initial commit	3 hours ago
README.md	Initial commit	3 hours ago
australian-weather.Rproj	Initial commit	3 hours ago

Git is alerting you to the fact that your local, cloned repository has more recent edits than the remote version of your work hosted on GitHub. To push your local changes and commits to GitHub, simply click the "Push" button[9]. You'll be asked to enter your GitHub username and password. You can then view your repository on GitHub, and check that the changes have been pushed.

[8] Terminal: `git pull`.

[9] Terminal: `git push`.

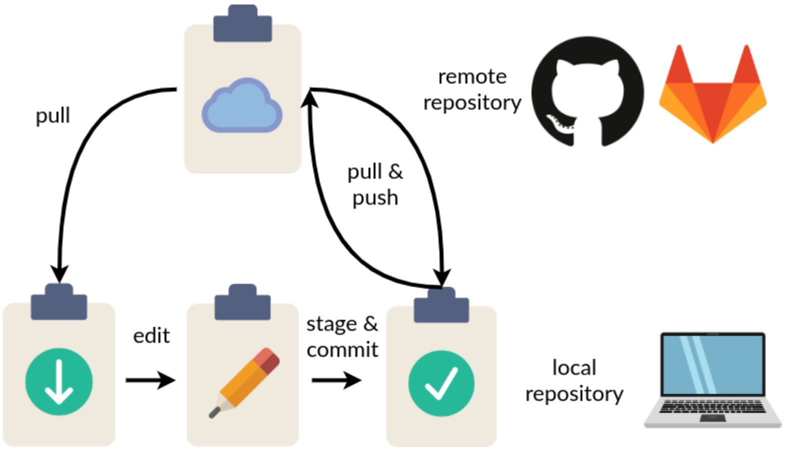


Figure 2.10: The full Git workflow.

2.8 *Recap of the workflow*

We've covered a lot of commands and processes this chapter, so let's just recap the workflow when using Git to track changes. A typical Git workflow involves:

1. Initialise: clone a remote repository to your local machine.
2. Update repository: pull from a remote repository to make sure your code is up to date.
3. Edit:
   (a) Add, delete, and edit files.
   (b) Move changes to the staging area.
   (c) Commit changes with a sensible message.
4. Update repository: pull from a remote repository to make sure your code is up to date.
5. Push your local changes to the remote repository.
6. Return to step 2.

⚒ **Practical** 2 part 2.

3

Using the Terminal

So far we've been using Git via the RStudio GUI tab. The great thing about the GUI is that you can always see the state of your files as well as everything being visually orientated. While the visual aid of a GUI makes it intuitive to learn, at some point you will need to know how to use Git in the terminal. This depends on which features you are wanting to use git for. In the margin you can see a table of which Git features can be completed by the using GUI and the terminal.

Feature	GUI	Terminal
`add`	✓	✓
`clone`	✓	✓
`commit`	✓	✓
`diff`	✓	✓
`push`	✓	✓
`pull`	✓	✓
`status`	✓	✓
`merge` branches	✗	✓

Table 3.1: Comparing GUI vs terminal.

3.1 *What is a terminal*

The terminal[1] is an interface in which you can type and execute text based commands. You can do almost anything in the terminal, from moving files around and creating new documents, to installing software. Once you get the hang of using the terminal, it's a lot quicker than using a GUI.

3.2 *Why use it?*

Anything the RStudio GUI can do, the terminal can do (and more)! The terminal gives us access to many more commands and features of Git. It can also be much faster than the RStudio GUI for some tasks. When choosing between using the terminal and RStudio GUI you must consider what you want to use Git for. Will Git just be used for staging, committing and pushing? Or will you want to work collaboratively using branches? Of course, you can use the terminal and GUI interchangeably, so you could stage, commit and push with the GUI whilst branching and merging with the terminal.

3.3 *How do I use it?*

The Unix shell has different names depending on what operating system you are using.

- Terminal on macOS
- Command Prompt on Windows
- Linux Terminal on Linux

We'll refer to it as the terminal throughout the notes but all commands are interchangeable between operating systems. Conveniently, RStudio has a tab to access the terminal[2]. You'll find it in the same panel as the Console

[1] It's also a wonderful film starring Tom Hanks.

[2] The short cut for opening the terminal in RStudio is `Shift+Alt+T`. To return to the console use `Ctrl+2`.

(usually the bottom left corner). Today, we are going to using the RStudio tab to access your terminal.

Be careful

A small warning. The terminal is very powerful and so that can make it a dangerous place. Commands such as `rm` will remove files permanently – i.e you won't be able to save them from the recycling bin. If you don't understand what a command does, think twice before running it.

3.4 *Exploring your files - `cd`*

There are two main terminal commands that allow us to root around our directories in our local machine. The first of these is `cd` which stands for "change directory". This allows you to switch between directories effortlessly. We can move into or out of directories.

```
~/Documents/awesome-project/R$           cd ../../
~/Documents$                             cd cat-pictures
~/Documents/cat-pictures$
```

The command `cd ../` will move us up (back) one directory. If you want to move up two directories, use `cd ../../` etc. The tab button is your friend here, as it will auto complete what you are typing. The command `cd cat-pictures` will move us into the directory named `cat-pictures`

3.5 *Listing your files - `ls`*

Another commonly used command is `ls`. This lists the sub-directories and files we have stored in the current directory.

```
~/Documents/cat-pictures$ ls
kittens
felix.png
mollie.png
sooty.jpg
```

Here, we have one directory called `kittens` and three files; `felix.png`, `mollie.png` and `sooty.jpg`.

3.6 *Using Git in the terminal*

If you want to use Git in the terminal, there are two rules you must follow:

1. You must be within the directory that is your git repository.
2. Every command where you are instructing Git to do something has to start with the word `git`. So for example, if we wanted to pull recent changes to our local machine, instead of clicking the `Pull` button in the RStudio GUI tab, we could use the terminal tab and type `git pull`.

Here is an example of what Git workflow would look like in the terminal.

```
cd ~/my_git_repo
git pull
git status
git diff
git add R/create_plot.R
git commit -m "Add title to plot"
git push
```

Commands such as push, pull, diff, commit you'll recognise from the R GUI buttons. A new command we haven't seen yet is `git status`. This will list any files that have changed but not been staged for commit as well as any staged files which have yet to be committed. It's similar to glancing at your Git GUI tab and looking at the modified M symbol.

```
~/Documents/cat-pictures$ git status
Changes to be committed:
    modified:  felix.png

Changes not staged for commit:
    modified:   sooty.jpg
```

If you've committed everything, and have no changed files, you'll get this lovely message[3].

```
~/Documents/cat-pictures$ git status
nothing to commit, working tree clean
```

AAAAARRGH!!
MY ARM! MY ARM! I BROKE ARM!
HELP! HELP!! WHY WON'T ANYONE HELP ME? CAN NO ONE HEAR ME?

Figure 3.1: Avoid broken trees.

3.7 *Options*

Notice the `-m` after `git commit` in our example above. The `-m` is short hand for the flag `--message`. Flags are often used in the terminal to set options. The `-m` option allows us to type the commit message directly in the terminal when we type the `git commit` command. If we just ran `git commit` without `-m "My message"`, a pop up would appear, prompting the user to type in a message. We tend to use `-m` as it's quicker and we're lazy - see table 3.2 for other helpful flags.

When using any of these short cuts, be sure to run `git status` first, so that you understand exactly which files you'll be adding.

3.8 *Example: Australian weather*

We'll return to the Australian weather example from the previous chapter, but this time, we'll do it on the command line for practice. This example is intended to mirror what we did in Chapter 2.

First, let's clone the repository into the directory `terminal_aus` using the `git clone` command:

```
git clone https://github.com/jumpingrivers/australian-weather.git terminal_aus
#> Cloning into 'terminal_aus'...
```

or

```
git clone https://gitlab.com/jumpingrivers-public/australian-weather.git terminal_aus
```

If we omit the `terminal_aus` option, then it would create a directory called

[3] We love a clean working tree.

Table 3.2: Useful flags for git commands.

Command	Description
`git add .`	Add all changed files to be staged. (New files not affected)
`git add *.R`	Adds all files with the .R extension to the staging area
`git commit -a`	`git add .` and `git commit` in a single command
`git commit -am "My commit"`	Short for `git commit -a -m "My commit"`

`australian-weather`. We can use `cd` to change directory and then look at the current status of the repository

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

As in Chapter 2, we replace "Brisbane" and "Sydney". This time when we look at the status, we see that two files have been modified

```
git status
#> On branch main
#> Your branch is up to date with 'origin/main'.
#>
#> 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:   R/create-plot.R
#>  modified:   graphics/scatter-temp-humidity.png
#>
#> no changes added to commit (use "git add" and/or "git
#> commit -a")
```

If we are curious or forgetful, then we examine *what* has changed using the `diff` command

```
git diff R/create-plot.R
#> diff --git a/R/create-plot.R b/R/create-plot.R
#> index d272805..a7eac09 100644
#> --- a/R/create-plot.R
#> +++ b/R/create-plot.R
#> @@ -9,7 +9,7 @@ df = read_csv("data/weatherAUS.csv")
#>  # Create a scatter plot of Humidity9am and Temp9am
#>  # a location
#>
```

```
#> -city = "Brisbane"
#> +city = "Sydney"
#>
#>  scatter_plot =
#>  df %>%
```

The change looks correct, so now we just have to add and commit the files

```
git add R/create-plot.R graphics/scatter-temp-humidity.png
git commit -m  "Change Brisbane to Sydney"
git status
#>
#> *** Please tell me who you are.
#>
#> Run
#>
#>   git config --global user.email "you@example.com"
#>   git config --global user.name "Your Name"
#>
#> to set your account's default identity.
#> Omit --global to set the identity only in this
#> repository.
#>
#> fatal: unable to auto-detect email address (got
#> 'root@runner-nthfetyx-project-17428478-concurrent-0.
#> (none)')
#> On branch main
#> Your branch is up to date with 'origin/main'.
#>
#> Changes to be committed:
#>   (use "git restore --staged <file>..." to unstage)
#>  modified:   R/create-plot.R
#>  modified:   graphics/scatter-temp-humidity.png
```

Then round off the process with a push

```
git push
```

4

Branching

4.1 *Why should we branch?*

Git repositories are split into branches (yes, like a tree!). The default branch is the *main* branch. So far this is the only branch we have been using. Think of main as the official working version of your project. It's the code that people will see when they visit `github.com/yourname/projectname`. [1]

It's important to ensure that your main branch has working code on it at all times, as it is the central repository of your code. If you mess something up on main it will affect everyone who is working on your code and anyone currently using your code. If you want to experiment with your code, or add a new feature safely, instead of working directory on *main* we can create a new *branch*.

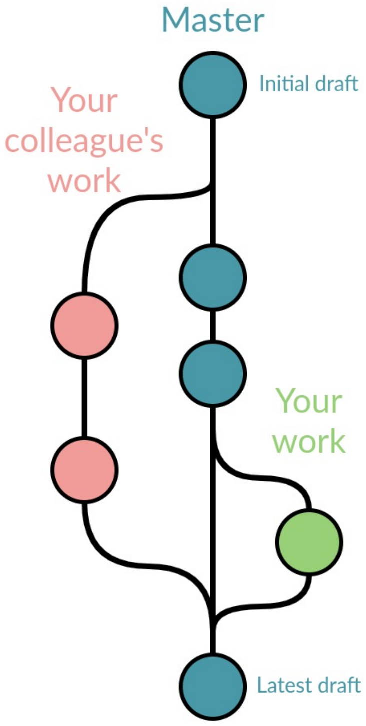


Figure 4.1: An example of a workflow in git.

4.2 *What is a branch*

Branches are a way of splitting your work up into strands. The main strand of your work, the *main* branch, stays they same, and you start a new strand or *branch* to develop the new feature you want to implement.

When you're happy and you've finished developing the feature on your branch, you'll probably want to incorporate your new code back into the main code. To do this you *merge* your branch with the *main branch*. You can add as many branches as you want, although generally it's good practice to not have too many branches.

4.3 *Creating a branch*

RStudio provides us with a nice easy-to-use branching interface within the Git tab. To create a new branch[2], we click "New Branch" in the top right corner of the pane, as seen in Figure 4.2. This produces a pop up window where we can give the new branch a name. Just like a commit message, you want to give a sensible name which conveys the reason for branching. For example, if you were going to be playing around with plot styling you might call your branch `plot_style`.

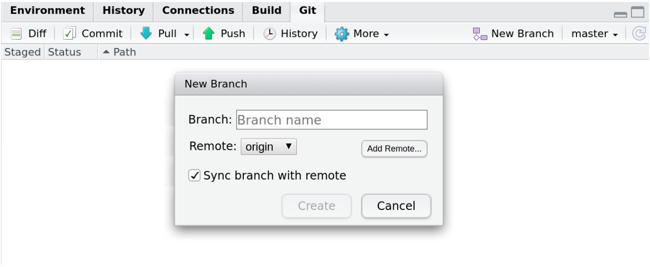
3

When a new branch is created, Git creates an identical copy of your current branch. In our case, this is currently `main`. This means `plot_style` would be an identical copy of `main`. This then allows us to make changes on our files, outside of the current working `main` branch.

[1] For many years, services referred to the default branch as the **master** branch. You may still come across this term in some repositories and services. However, many services now choose to use the name **main**, in order to remove unnecessary references to slavery and replace them with more inclusive terms.

[2] Terminal:
`git checkout -b "branch_name"`.

[3] To delete a branch without merging use `git branch -d "branch_name"`.



Branch: Branch name

Remote: origin

☑ Sync branch with remote

Figure 4.2: Branches in RStudio.

4.4 *Switching branches*

We can switch between branches easily[4] by clicking on the branch dropdown button in the top right corner of the Git tab, as seen in. The current branch you are working on is always shown next to the "New Branch" button. In Figure 4.2, we can see that we are currently on the main branch, as we hadn't created the new branch at the time.

Once we've create a new branch and have switched to work on it, we can edit and develop our code as normal. We make edits, stage changes, commit and push regularly. However, instead of committing to *main* we'll now be committing to our new branch. Now we will always have a working version of our repository, whilst able to add new features and test.

4.5 *Merging main changes into your new branch*

At some point, you'll be happy that the new features you've added to your code on your branch are working. We now need to check if there have been changes made to the main branch[5]. To do this we switch to main and run `git pull`. In the event that the remote main was ahead of our local main branch (i.e. Another developer has pushed a new commit object to the remote repository), we want to *merge* these changes into our new branch we are working on. Unfortunately, you can't do this bit in RStudio, and we need to work in the terminal. First you have to make sure you're on your new branch. You can do this bit in the RStudio GUI by clicking the dropdown menu and selecting the name you have given your new branch. Once you're on this new branch, the command to merge main **into** your new branch is

```
git merge main
```

Git will take any new commits that have been pushed to `main` and try to combine them into the new `plot_style` branch automatically.

4.6 *Pushing your branch to the remote*

We now need to push our new branch to the remote repository. However, only the main branch currently exists on the remote (i.e. GitHub/GitLab). There is not a `plot_style` branch on the remote that we can push to. Currently, if we were to try and run `git push` , we'd get an error

```
git push
# fatal: The current branch plot_style has no upstream branch.
```

[4] To view branches in the terminal use `git branch`, and to switch to main use `git checkout main`.

[5] In working practice, we would be checking to see if another project developer has pushed any new commits to the remote main branch.

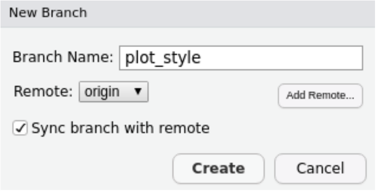
```
# To push the current branch and set the remote as upstream, use

# git push --set-upstream origin plot_style
```

This is because we need explicitly tell Git, to create a remote branch and link it to your local `plot_style`. This is technically called *"setting the upstream branch"*. In order to resolve this, we just need to run the code that the error has given us i.e.

```
git push --set-upstream origin plot_style
```

You only need to run this the first time you push a new branch. Each successive time, you can just run `git push`.

However, if you created the branch in the RStudio GUI, the hard work has been done for you. When you create a branch in the GUI, a checkbox labelled "Sync branch with remote" is ticked by default. This automatically creates the remote branch and links it with the local.



Branch Name: plot_style

Remote: origin

☑ Sync branch with remote

Figure 4.3: Syncing branches in the GUI.

4.7 *Merge conflicts*

Most of the time, the merge will go smoothly. However, if both the branches you are merging have changed in the same part of the same file you will get a *merge conflict*.

```
git merge main
# Auto-merging create_plot.R
# CONFLICT (content): Merge conflict in create_plot.R
# Automatic merge failed; fix conflicts and then
#     commit the result.
```

First, we need to work out which files are causing the problem. We can read in the message above that `Merge conflict in create_plot.R`. Remember, you can also always check the state of all files with `git status`:

```
git status
# On branch plot_style
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#     both modified:      create_plot.R
#
# no changes added to commit
#   (use "git add" and/or "git commit -a")
```

To actually resolve the merge conflict, all we need to do is open the affected file, and view the contents to find out which lines are in conflict. In this case we can see the line affected is the `geom_point()` line of `create_plot.R`.

```
<<<<<<< plot_style:create_plot.R
geom_point(colour = "red")
```

```
=======
geom_point(colour = "blue")
>>>>>>> HEAD:create_plot.R
```

In this conflict, the lines between

`<<<<<< HEAD:create_plot.R`

and

`======`

are the content from the branch you are currently on — plot_style. The lines between

`======`

and

`>>>>>>> main:create_plot.R`

are from the branch we are merging. In this case, we have set `colour = "red"` in `geom_point()` on the `plot_style` branch. Whilst we were working on our `plot_style` branch, someone else has changed main branch so that `colour = "blue"`. Git doesn't know which line is the correct version, and so it asks us to manually resolve the merge conflict.

To resolve the conflict, simply edit this section until it reflects the state you want in the merged result[6]

```
geom_point(colour = "red")
```

Now run `git add create_plot.R` and `git commit` to finalise the merge. Simple. The important thing when handling merge conflicts is not to panic, and to pause and think before you start editing. Most of the time they are relatively small and straightforward to resolve.

4.8 *Pull/merge requests*

Depending on what host you are using, the type of request we're about to cover has a slightly different name.

- GitHub - pull request
- GitLab - merge request

To make things slightly easier, we'll just refer to it as a pull request from now on.

Now that we have pushed our local changes to the remote branch, we now need to pull them into the remote main branch. To do this, we need to create a "pull request". We can achieve this by heading to the "branches" page and click the relevant "new pull request" button. On GitLab this is just "merge request". This is useful for a couple of things:

1. We can let other members of the company to review our changes
2. It allows people to comment and give feedback.

Once the whole team is happy with the changes and Git is able to cleanly merge without a conflict, we can then click "merge pull request". In GitLab this is just "merge".

[6] Remember to remove the markers <<<<<<, ====== and >>>>>>>.

5 Good Practices

5.1 The .gitignore file

A .gitignore file is a hidden file[1] which lists any files in your repository that you don't want Git to track.

Why should I use a .gitignore?

There are a number of reasons that you might not want a file to be tracked in Git. For example, if you are using an API, you might have a script which contains the API keys and secrets. Your code needs these to work, but obviously you don't want to share them with the world. By naming this file in .gitignore, the file won't be tracked in Git, and therefore not pushed to GitHub. However, it will stay in your local repository, which means your code will still work locally.

Generally it's bad practice to commit files which are created by other files[2]. For example, committing .html files which are generated by an .Rmd file. In this case you might want to add *.html to your .gitignore[3]. This will tell Git to ignore any files with the html extension.

R projects generate lots of hidden files which don't need committing, such as your .Rhistory and your .Rproj.user/. Not only are these file unnecessary, but if you commit these files, you might accidental reveal information that you didn't intend to. For example, if you ran some code involving a secret for an API, that line of code would appear in your .Rhistory file. Even if you didn't explicitly commit the code that generates your secret, committing the .Rhistory file could expose it for you.

Creating a .gitignore file

When you create a repository on GitHub/GitLab, you will get the option to add a .gitignore. You should tick the box! Alternatively, you can create it in the usual way you would create a file. Using R [4]

```r
file.create(".gitignore")
file.edit(".gitignore")
```

There's even a button in the RStudio Git GUI to edit your .gitignore file. You can find it under the More dropdown menu with the cog icon. A good, example .gitignore file can be found at

https://github.com/github/gitignore/blob/master/R.gitignore

[1] Hidden files start with a . and aren't usually visible in your file explorer. You can view them by ticking "Show hidden files" under "More" in the "Files" tab.

[2] Question: In the previous chapter, we tracked a generated file - the graphic. Is this a good idea?

[3] Note that .gitignore files don't support general regular expressions.

[4] The **usethis** package has a function git_vaccinate() which adds .DS_Store, .Rproj.user, and .Rhistory to your global .gitignore. This is good practice as it ensures that you will never accidentally leak credentials to GitHub.

5.2 *Writing a good commit message*

Just as there are conventions for styling your code and writing good file names, there are also conventions for writing a good Git commit message. Which of the following Git commits would you rather read?

- `Add branded colours to scatter plot`
- `Fix typo in markdown report`
- `more fixes for broken stuff.`
- `fixing the plotting function that was breaking cos I forgot to use aes in ggplot2.`

COMMENT
DATE
CREATED MAIN LOOP & TIMING CONTROL 14 HOURS AGO
ENABLED CONFIG FILE PARSING 9 HOURS AGO
MISC BUGFIXES 5 HOURS AGO
CODE ADDITIONS/EDITS 4 HOURS AGO
MORE CODE 4 HOURS AGO
HERE HAVE CODE 4 HOURS AGO
AAAAAAAA 3 HOURS AGO
ADKFJSLKDFJSDKLFJ 3 HOURS AGO
MY HANDS ARE TYPING WORDS 2 HOURS AGO
HAAAAAAAAANDS 2 HOURS AGO
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Figure 5.1: `https://xkcd.com/1296/`

Generally, the top two commits, are much more readable and understandable than the bottom two. Here are some general rules we think are sensible for writing Git commits[5].

Limit the commit message to 50 characters

Keeping commit messages short ensures that they are readable, and forces you to stop and think about the most concise way to explain what's going on. Fifty characters is not a hard limit, just a rule of thumb.

If you're finding it difficult summarising your commit into a short statement, you might be committing too many changes at once.

Use a fixed style

Keeping the style consistent makes reading multiple commits easier. We think that using sentence case[6] looks best. Also, because your Git commits should only be a short statement, there is no need to put punctuation at the end of the commit. For example:

```
Fix bug in plotting function
```

KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

Figure 5.2: `https://xkcd.com/1513/`

Use the imperative mood

The imperative mood is just a fancy way of saying the words used to give a command or instruction. For example "Do the washing up" or "Eat your vegetables". At first, writing commits in this way can sound a little rude. However, its a great way to explain clearly and concisely what a commit does. Also, when Git creates commits on your behalf, such as pulling new changes, it will write commits in this way. For example

```
Merge branch 'main' of github.com:username/repo
```

You should always write your commands in this tense because version control is all about flexibility through time. You're not writing about what a commit did, you're writing about what a commit does. So write

```
Add title to scatter plot
```

rather than

```
Added title to scatter plot
```

5.3 *Tidy up*

Good practice with Git generally means keeping things as tidy as possible.

[5] We are opinionated! We are also two-faced and don't always do this!

[6] Capitalising the first word, and any proper nouns.

Let's say you've created a new branch called `linear_regression` to add a new linear regression model to your code. When you've finished working on that branch and merged it into main successfully, it's a good idea to delete the branch. You're not actively working on it, and the main has been updated so there is no reason to still have it hanging around. If you want to do some more work on your linear regression model at a later date, you can simply create a new branch.

To remove a branch on both your laptop and GitHub, open the Terminal and type `git push origin --delete linear_regression`. Alternatively, you could navigate to the branch on GitHub/GitLab and delete it from there. This is what I tend to do.

Figure 5.3: `https://xkcd.com/1077/`

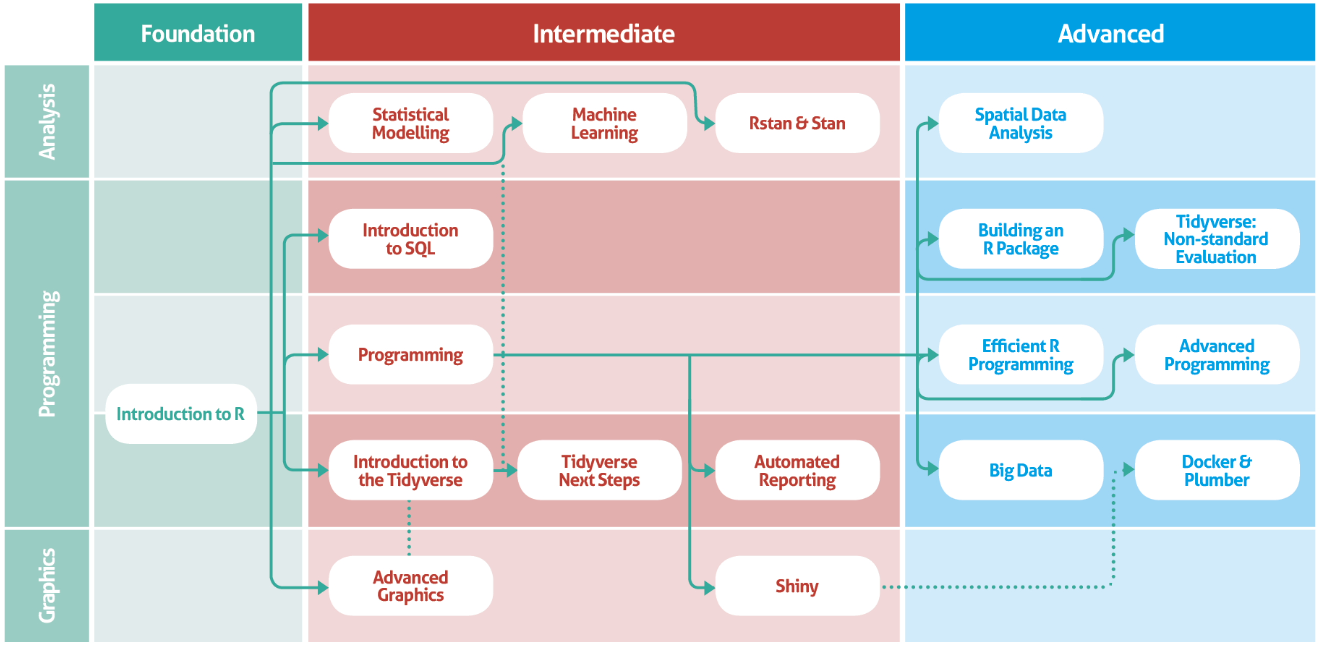
5.4 *Help, I messed up!*

Sometimes, you get yourself into a Git pickle. This happens to all of us from time to time. The best way to solve this, is to not get yourself into a Git pickle in the first place! You can do this by having clear communication with your colleagues on who is working on what, and make sure that you pull from the remote often.

There are ways to revert the status of your repository to a past commit using Git, but that's not a matter for this one-day introductory course.

If everything does go pear-shaped on your laptop, remember that you always have that remote repository as a back up. In a worst-case scenario, you can always delete the folder on your laptop, and re-clone the repository from GitHub/GitLab. Although not ideal, it's a handy strategy to have to get you to your back up. Obviously this strategy only works if you've been pushing your work often to the remote. So remember: Commit early, commit often and don't panic. Good luck!

Training Course Dependencies



Foundation

Intermediate

Advanced

Analysis

Programming

Graphics

Introduction to R

Statistical Modelling

Machine Learning

Rstan & Stan

Introduction to SQL

Programming

Introduction to the Tidyverse

Tidyverse Next Steps

Automated Reporting

Advanced Graphics

Shiny

Spatial Data Analysis

Building an R Package

Tidyverse: Non-standard Evaluation

Efficient R Programming

Advanced Programming

Big Data

Docker & Plumber

OUR PARTNERS

Newcastle University

ROYAL STATISTICAL SOCIETY
DATA | EVIDENCE | DECISIONS

RStudio

Microsoft

CODECLAN