**Introduction to Git/GitHub**

**Basic setup**

This document provides accompanying training material used in the Introduction to Git/GitHub Training session, conducted by the Data & Analysis R training group. Prior to joining the session, you should ensure you are set up on the Analytical Platform - see <https://user-guidance.services.alpha.mojanalytics.xyz/get-started.html>.

We expect participants to have completed the Introduction to using R on the Analytical Platform (or equivalent standard) and Introduction to R (or equivalent standard). Previous recordings and accompanying materials of sessions can be found here - <https://moj-analytical-services.github.io/ap-tools-training/>.

You should then work through the following:

1. Deploy (if necessary) and open RStudio: <https://user-guidance.services.alpha.mojanalytics.xyz/tools/control-panel.html#control-panel>
2. Connect RStudio to GitHub: <https://user-guidance.services.alpha.mojanalytics.xyz/github.html#setup-github-keys-to-access-it-from-r-studio-and-jupyter>
3. Clone the GitHub repository for this course (<https://github.com/moj-analytical-services/intro_to_github_training>) by following step 1 here: <https://user-guidance.services.alpha.mojanalytics.xyz/github.html#r-studio>
4. In the Console window in RStudio, enter this command to make sure you have the required packages installed: renv::restore()

**Contents**

This session will provide you with an understanding of what Git/GitHub is and explain the key benefits of using it alongside the Analytical Platform and RStudio. Step-by-step instructions will be provided about how to use GitHub in RStudio and how to collaborate effectively with others. Even if you are coding a one-off product, we highly recommend you use GitHub.

**Sessions aims**

1. What Git and GitHub are?
2. Understanding how to use GitHub
3. Ways to approach Git to enable effective use and collaboration
4. Security considerations when using GitHub
5. Further information and useful links

As this session is focused on those who are new to Git/Github, it will not cover more advanced commands and topics. We welcome comments and volunteers to develop a session in the future.

* 1. **What are Git and GitHub?**

First, Git and GitHub are two separate things.

Git is a version control system. Version control refers to the process of saving different files or ‘versions’ throughout the various stages of a project. It’s a bit like “Microsoft Office’s track changes on steroids”. This enables users to keep track of changes to code. All code written on the Analytical Platform should be stored in a Git repository on GitHub, including Python scripts and Jupyter notebooks. In this session, we will show how to use Git in RStudio.

GitHub makes it easier to collaborate using git. It’s a platform and website that provides a home for your coding projects allowing multiple people to work on a single project and see edits to the code in real-time. The code is held in a repository on the GitHub website – a repository is a file location where your project is stored. This means your code can be made publicly available in GitHub repositories on the GitHub website so others can interact with and contribute to the code. GitHub is therefore, an open-source platform which is free to use.

To gain access to the Analytical Platform, you would have needed to use GitHub to create a GitHub account. The Analytical Platform team use this authentication service of GitHub to enable users to have one account for the Analytical Platform, RStudio and GitHub. If you have not set up an Analytical Platform or GitHub account, then please refer to the guidance here [Get Started - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/get-started.html) and watch the Introduction to using R on the Analytical Platform training session found here <https://moj-analytical-services.github.io/ap-tools-training/>.

* 1. **Advantages of using Git and GitHub**

As your projects develop and you introduce new features or make improvements to the code, it is helpful to keep track of what changes are being made and be able to go back to previous versions if all else fails. Git and GitHub keep track of evolving projects in a sensible, precise way, with the ability to view, compare in detail and restore previous versions. It is also collaborative, allowing multiple people to work on a project at one time with a managed way of combining several people’s work. This makes it easier to resolve errors and fix other mistakes that might occur through development of code with the ability to restore previous versions.

**1.2.1 Advantages of Git**

As Git is different to GitHub, it is important to understand their differences and benefits. Git has many great advantages, from easily creating new code that can be stored and saved, to managing and creating new versions of your project and quality checking your code. Some of the advantages are as follows:

* **Security:** Git has the ability to keep your version or different versions secure, saved and easily accessible
* **Flexibility:** Git’s flexibility allows the user to have multiple different versions of your code where changes made in different versions do not affect each other.
* **Version Control:** Having different versions of your code will allow for greater quality checks and each different version can be checked individually.

**1.2.2 Advantages of GitHub**

GitHub is your user-friendly interface of Git that brings everything to the forefront, in an easy to understand and comprehendible way. This allows someone to look at the project and understand why the changes were made earlier on. You can note changes in each version to help team members stay up to date on what’s completed and what needs to be accomplished. More benefits include:

* **Accessibility:** You can access the code files from another computer and so can other developers.
* **Documentation:** ReadME.md file one of the first things other users see within your repository/project. The README.md file is used for documentation purposes – providing other users with an overview of the project so they run the code in RStudio.
* **Collaboration:** GitHub provides you with tools which allow you or others to review each other's work, combining your code with others or making changes/updates to your own. It can do all this while keeping a track of the version history which can be changed back to old versions if required.
* **Project management:** Coordinate and track your project in one place, such as project boards and tracking of issues found in the code. but allows you to include issues you have found in the code.
* **GitHub issues:** This is a central place to maintain a ‘to do’ list for a project, and to discuss them with your team. ‘Issues’ can be bug fixes (such as ‘fix divide by zero errors in output tables’), or features (e.g. ‘add a percentage change column to output table’), or anything else you want.
* **Version control management:** By using issues and projects you can keep track of who is working on what. If you use issues, you automatically preserve a record of why changes were made to code. So, you can see when a line of code was last changed, and which issue it related to, and who wrote it.

Git and GitHub have a multitude of benefits that help individuals manage their projects, and if used correctly and efficiently GitHub will give you additional security in your projects allowing you to track, manage and maintain the standard of your work. With this additional quality assurance measure you can be certain that your code and methods are exactly what you expect!

**1.3 Ways to use Git in RStudio**

To take advantage of GitHub, we need to know how to use Git to interact between RStudio on the Analytical Platform and GitHub.

There are two ways to use Git in RStudio:

* **The terminal** – this is the command line interface. This allows users to write commands that associate directly with the terminal window. Users must be aware of the commands to perform any task they want and understand the responses from the terminal.
* **The graphical user interface (GUI)** – this lets users use graphics for interacting with Git such as buttons, windows etc. It’s easier to use but some people find it hard to know what action Git is performing.

Some people prefer the terminal, some prefer the GUI and some people use a mixture of the two. The more you continue to use Git and GitHub, the more comfortable you’ll get using the commands and/or interface. The key is to start small and maintain your momentum. It will eventually get easier as you build small projects and host them on GitHub using Git.

Later in this session, we will provide a step-by-step guide on how to use the terminal and the graphical user interface (GUI).

To note, the terminal in RStudio is very similar to the terminal in Jupyter notebook for Python users.

**1.4 Introduction to Git concepts**

Once you have set up a project to use Git, you will see a .git folder in your working directory. Git is now monitoring all the changes you make to the directory. You can create some files, and whenever you want, you can tell Git to take a snapshot of their current state. A snapshot is called a commit. We can represent a commit as a blob.

1

You can then continue working - modifying files. At any point, you can take another snapshot. The project now looks like version 2, but version 1 is still saved, and you can go back and look at it if you want. (Note that a real commit in Git don’t have an integer as an identifier, but rather a hexadecimal string called a hash.)

1

2

Git calls the series of commits a branch. By default, Git labels your initial branch "main".

main

1

2

A branch is just a pointer to a commit. Each time you add a commit, the pointer automatically moves along to the new commit.

main

1

2

3

If you want, you can go off on a tangent by creating a new branch, for example to work on a new feature. The new branch starts as just a new pointer to the current commit.

main

new\_feature

If you add more commits, they will now be added to the new\_feature branch.

main

new\_feature

|  |  |
| --- | --- |
| main  new\_feature | main  new\_feature |

main

new\_feature

While you are working on your feature branch, you (or a colleague) might make changes to the main branch, so the branches will diverge - each branch contains changes that the other does not have.

main

new\_feature

Eventually, once your feature is complete, you can tell Git to merge it back into the main branch. Git handles this merge process so that the main branch then contains all the changes from both branches. If there are conflicting changes, Git will require you to resolve them manually.

main

new\_feature

Branches are a powerful feature of Git. An important benefit is they allow multiple people to work on separate features at the same time, each working on a separate branch. Git then handles the process of merging all the changes together (although some manual work might be needed if there are conflicts, for example where two branches both try to modify the same lines in a file).

* 1. **Understanding how to use Git and GitHub**

**Walk-through of the basic Git workflow**

We will walk through the basic steps on how to use Git and GitHub for you to take away and practice with your own projects. We will do this in four main sections:

1. Starting a repository and creating a branch
2. Editing files in your repository and committing your changes
3. Pushing your changes to Github
4. Pulling your changes to RStudio

Each section will first explain theoretically the action we are performing and then demonstrate how to use this in the GUI and the terminal – providing you with both options. We will then complete an exercise for you to practice either the GUI or the terminal.

**2.1 Starting a repository and creating a branch**

* + 1. **Create/clone a repository**

A repository is a place to store your code and record changes to your project.

On the homepage of the MoJ Analytical Services organisation on GitHub (<https://github.com/moj-analytical-services>) you can see all the repositories related to work in MoJ Data and Analysis that you have access to. A repository (known as a ‘repo’) is an area on GitHub where you can store code and documentation for a particular project. Each project in RStudio should have a separate repo associated with it.

Start by creating a new repository for your project on GitHub (see the [Analytical Platform guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#create-a-new-project-in-github)) or find the repository you want to work on if it exists already. You then need to create a local copy of the repository that you can work on and change (see the [Analytical Platform guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/rstudio-git.html#work-with-git-in-rstudio) and [Intro to us R on the AP](https://moj-analytical-services.github.io/ap-tools-training/)). Creating a local copy of a repository is called cloning the repository.

In this session, we will use the repository created for this training - <https://github.com/moj-analytical-services/intro_to_github_training>. You should have cloned this repository into RStudio before this session and have it ready in your projects. If not, then please follow the steps <https://user-guidance.services.alpha.mojanalytics.xyz/github.html#r-studio>. It is not essential to follow along during this course, but it is useful to practice the steps whilst watching the presenters.

* + 1. **Create a branch**

Once you have created a repo or have cloned a repo into your RStudio, then you will want to add or make changes to your R project. You can make changes to your ‘main’ branch on the repository, or you can create a new branch to make changes to the project separately. Branches allow multiple features to be developed at the same time. We will explain ways of working with branches in section 3. When you create a new branch you will need to give it a name. Here are some tips for naming your branch to make it easy to understand at a glance..

1. Use separators​ (avoid using spaces in names use \_ or – or /)
2. Start naming with category word​ (for example: if adding a graph, start the branch name with Plotting)
3. Include authors name​ (so others know who created the branch)
4. Avoid using only numbers​ (Text is more descriptive)
5. Avoid long branch names​ (One or few words, not whole sentences)
6. Be consistent
   1. **Editing files in your repository and committing your changes**
      1. **Edit some files, for example write some code in an R script**

Next, you would make some changes to a file in your repository. For example, you could create an R script and write some code in it. You can push any change to GitHub – comments, code, documents etc.

* + 1. **Stage and commit a group of changes**

Once you are happy with the changes to your files, you can save a snapshot containing the state of all your files by creating a “commit”. A commit can contain changes to one file or multiple files. The ability to create commits, each containing a different state of your repository, and to switch between the commits, is what makes Git a version control system.

Before you can commit changes, you must add them to the “staging area”, which tells Git to include the changes in your next commit. The purpose of the staging area is to allow you to select just certain files (or sections within files) to commit.

Ideally, a commit should be a self-contained, working unit of change to your code.

The steps for committing changes are:

* + - 1. Save the changes to your files
      2. Add the modified files you wish to commit to the staging area.
      3. Commit the staged files.

Every commit needs an accompanying commit message in which you should briefly describe the changes made. The commit message will appear alongside the commit in the repository history, helping you and others to see what changes were made. The message should be clear on the changes made.

Once your commit has been created, you will receive an output message showing the number of files you have changed, the number of lines inserted (+) and the number of lines deleted (-). Note that a modified line may be treated as an insertion and a deletion.

* 1. **Pushing your changes to GitHub**
     1. **Push changes**

Once you have committed your changes locally, you need to upload your commits to the remote GitHub repo. This is called pushing your commits. You push commits from a local branch to a remote copy of the branch.

**2.3.2 Merging and pull requests**

When you’ve finished making all the changes you want to include in your branch, you will probably want to merge your branch into the main branch of the repository, which will incorporate your commits into that branch. There are two ways to merge a branch, either locally in the terminal or on GitHub via a “pull request”. In this training session, we will concentrate on merging using GitHub.

A pull request is a GitHub feature that handles the merge process on the remote repo. It allows collaborators to discuss and review the proposed changes before they are merged. This is particularly useful for quality assurance. To ensure that every pull request into a branch such as the main branch is reviewed, a branch can be protected so that pull requests can only be merged once reviewed and approved by a colleague. Another benefit of the pull request feature is for project management – the discussion thread can be used to explain the proposed changes and link to a project planning platform or issues log.

Note that once you have opened a pull request, you can continue to commit to the branch. When you push additional commits, the pull request page will update to show the latest changes.

To merge a branch on GitHub via a pull request:

If you recently pushed a change to GitHub and navigate to the online repo, GitHub notices the changes to the branch and asks whether you want to open a pull request. Otherwise, to open a pull request on GitHub:

* + 1. Go to the "Pull request" tab.
    2. Click "New pull request".
    3. Use the drop-down selectors to select the base branch and comparison branch.
    4. Click "Create pull request"
    5. **Confirm the pull request**

To merge a pull request, select "Merge pull request" at the bottom of the conversation thread. If there are conflicting changes in the branches you are trying to merge, GitHub will tell you that there are conflicts that must be resolved before you can merge. Merge conflicts will be discussed in section 3.

Once you have merged the pull request, GitHub will give you the option to delete the branch. This is safe to do because all the commits from the pull request have now been incorporated into the main branch. The pull request page, with the discussion thread and review comments, will remain on GitHub – it is just the branch label that will be deleted. Deleting merged branches makes it easier to find the branches that are still being worked on.

* 1. **Pull changes into a local branch**

If there have been changes made to a branch on the remote repo, you will need to update your local branch to incorporate the changes. For example, if a pull request has been merged into the main branch on GitHub, you will probably want to update your local version of the main branch.

* 1. **View the commit history**

Once a commit has been made, it is retained in the project’s history, which means it can be viewed and returned to at any later point. It is often useful to see the history of the most recent commits on a branch, which you can do using RStudio or the git log command.

**2.6 Continuing with a new local branch**

Once your local copy of the main branch has been updated by pulling the changes from GitHub, your main branch contains all the commits from the merged branch. You can create a new branch, branching off the main branch, to start work on your next contribution.

* 1. **Exercise:**

Now we will practice the four steps explained in Section 2 – this will allow you to familiarise yourself with using Git and GitHub in R studio. Please only use the terminal commands OR the GUI for this exercise. Slides have been uploaded to the GitHub repo showing you the steps, which you can refer to during the exercise.

1. **Starting a repository and creating a branch**
   * 1. If not done so already, please clone the repo required for this training session into your R studio. See steps explained in section 2.1.1. <https://github.com/moj-analytical-services/intro_to_github_training>
     2. If you cloned the repo a few days ago, we would recommend pulling the changes from the main to ensure your RStudio is up to date.
     3. Create a branch. Call your branch that is relevant to the changes you are making to the code/project.
2. **Editing files in your repository and committing your changes**
   * 1. Create a new folder within ‘practice area’ and then create new R script within the folder. Write something in the R script. Press ‘save’ to save the script.
     2. Once you’re happy with the changes to the files, you need to add your changes to the ‘staging area’ and then commit them. Add a commit message that briefly describes the changes made to the files.
3. **Pushing your changes to Github**
   * 1. Once you have committed your changes locally, you need to upload your commits to the remote Github repo by pushing your commits.
     2. Then merge your branch via a pull request in Github.
4. **Pulling your changes to RStudio**
   * 1. Now that your branch is merged to the main branch on Github, you need to update your local R studio to incorporate the changes.
     2. Make sure you’re on the main branch in R studio and then pull the changes from GitHub.
     3. You can now check your commit history to see that your changes have been made.
   1. **Ways to approach Git to enable effective use and collaboration**

There is no limit to the number of people who can potentially be working on a repo at the same time. However, there are technical limits to Git’s ability to integrate people’s work effectively, and established processes and practices for dealing with these limits.

**3.1 'Undoing' commits**

There are a number of commands that can be run in the terminal, which can undo work, e.g. git restore, git revert and git reset. These all do slightly different things and it’s a good idea to understand what they do before using them. See section 5 for more details on these.

**3.2 Merge conflicts**

One of the main reasons for using git to manage collaborative work is that it can automatically merge together the work of multiple contributors to a single project – even to the same file.

Git can easily merge branches that differ due to:

* file additions
* changes to different files
* changes in non-adjacent lines in the same file

However, on occasion you will find that git cannot perform a merge without input from you. When you go to open a pull request to merge branches that git cannot automatically merge, you will be warned of a **merge conflict**.

The merge conflict simply means that git has conflicting information from the two merging branches about what the merged branch should look like.

A merge conflict will usually arise because on the two merging branches, there are changes to the same or adjacent lines within the same file. It will also happen if a file has been deleted on one branch but not another.

When this happens, you can still open the pull request, but you will need to tell git how you would like the conflict to be resolved. This can be done on GitHub itself or in RStudio.

**3.2.1 Resolving a merge conflict in GitHub**

Resolving a merge conflict is something that any new user to git should know how to do.

I will demonstrate how to resolve a merge conflict on GitHub. You don’t need to try to do anything yourself. You can just watch along.

When you go to open the pull request you will be warned of the merge conflict.

Note at this point the number of commits to the merging branch to compare with what we see after *the conflict has been resolved.*

Create the pull request.

Note the box containing:

- the warning ‘**This branch has conflicts that must be resolved**’

- The ‘**Conflicting files'** list

The ‘**Resolve conflicts’** button

After clicking on ‘**Resolve conflicts**’ you will be taken to a screen that has two windows. The first window has a box for each file with conflicts. Clicking on each of these boxes will show you where the conflict exists within each file.

Alternative versions of the same content are represented between arrows (e.g. <<<<<<<), with equality signs (=======) separating them. There can be any number of these within a file, depending on the degree of conflict. Resolve by:

* For every instance of a conflict, remove one of the alternative sections, and removing the rows marked with arrows and equality signs. When you do that, you are signalling to git how you want the conflict to be resolved.
* Repeat this do this for each instance of conflict within a file.
* The button ‘**Mark as resolved**’ will remain greyed out for each file for as long as any marked rows exist. When all marked rows are removed, click that button to signal that the file has been resolved.
* Repeat for all files where there is conflict.
* When all files have been resolved, the green ‘**Commit merge**’ will appear, and you may click it.

When you do this and go back to the main pull request page, you might notice the number of commits has increased by one. This is because when you merge you are adding an extra commit to the merging branch. You can see this by clicking on ‘Commits’. The commit description describes the commit as a merge of the base branch into the merging branch.

At this point it would be a good idea to pull from GitHub to your local repo in RStudio and confirm that your code runs. After all, git is only checking for inconsistencies in character strings. It is not checking anything about your code’s ability to function as you would like

You can also merge your branch locally using the terminal. We recommend you merge on GitHub as it’s easier to avoid merge conflicts, however, it may become useful when you’re more familiar with GitHub and need to update your feature branches to the main. Please see section 5 for more details.

**3.3 Effective collaboration**

Getting used to working together in this way, adding to and drawing from a shared synchronised resource (the ‘origin’ on GitHub), can be difficult. There is a number of different practices that your project can adopt in order to make the best use of git and Github. If you follow these you will minimise merge conflicts and get the best out of git and GitHub generally as collaborative tools.

**Project management tools:** Teams can manage work allocation and PR review assignment using project management tools such Github’s [project board,](https://github.com/moj-analytical-services/intro_to_github_training/projects?query=is%3Aopen) or even just using Github’s [Issues](https://github.com/moj-analytical-services/intro_to_github_training/issues) features [(see this video)](https://web.microsoftstream.com/video/c8821c1c-7779-443a-8084-8b1e01897ba2)**.** Agree on your methods of communication with one another (e.g. using Teams/slack).

**Follow the guidance on the AP to setup your project repo:** [The guidance starts here](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html). You should be starting from the MOJ Analytical Services organisational page, not starting the repo from your own personal user page. Follow the AP guidance on naming conventions, e.g. using dashes rather than dots or hyphens to separate words.

**Document your repo:** [When you create your repo](https://github.com/organizations/moj-analytical-services/repositories/new), you will be given the option of creating a markdown [README file](https://github.com/moj-analytical-services/intro_to_github_training#readme). You should always take this option and use this as a resource to provide basic information about your repo to anyone in the organisation who sees it. It should cover how the project is structured via a README, any associated manual steps, and the datasets and code used. Format the content appropriately, e.g., using code blocks to indicate code. Guides to markdown formatting are widely available (do not confuse with R Markdown). The file will be placed in the root directory of your repo and can be edited on Github itself or in RStudio. Git or GitHub will show this in rich markdown at the front of the repo page. There is also a wiki feature.

**Manage project access:** [Access to repos is via ‘teams’](https://user-guidance.analytical-platform.service.justice.gov.uk/github/manage-access.html) (not MS Teams), with each team having different levels of access, e.g. read access vs. Write access vs. Admin access. On larger projects it is wise to restrict membership of your admin team. You should create these from the main MoJ Analytical Services page when you create your repo. Give each team a specific name related to their access level and the project/repo. If you do not, and just give people direct access, a Data Enginering bot will put everyone in a team for that repo.

**Gitflow:** Workflows are branching patterns that teams can adopt when using git. All projects need to follow one workflow. [Analysts within MoJ are required to use one called Gitflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow) *(not to be confused with the similarly named Github flow).* The key feature of Gitflow is the use of an additional ‘development’ branch, conventionally called ‘dev’, as an insulator between main and feature development branches. Section 5 has more details on Gitflow. Gitflow should be used unless there is a good reason to do otherwise, such as when you are only making one development to the main branch.

**Protect your branches:** Project administrators can tweak the settings in branches branch protection rules to prevent sudden changes happening to the repo. In line with Gitflow, in Settings, ‘dev’ can be the default base branch. In Settings > Branches > Branch protection rules, you can require that a PR is submitted before a merge and control who can review, e.g. preventing merges from dev to main.

**Work on separate project features:** As a general rule, collaboration on a project works best if people try to work not only on separate files, but on separate project features, so as to minimise prospect of changes to overlapping code. Combining this approach with Gitflow is a pretty solid formula.

**Give files specific names:** For example, ‘code.R’ is ambiguous and not helpful to your collaborators. In addition, non-descriptive file names like this increase the risk of two people trying to create file with the same name.

**Small branches and regular merges:** This will reduce the risk of merge conflicts. To make this work effectively, communicate regularly with your team to ensure that PRs can be swiftly reviewed *(note that this is not itself a tactic associated with Gitflow, which is designed to be robust to long-lived branches)*

**Unit tests:** Formal comprehensive systematic testing of code across the repo before committing (and reviewing) will reduce the chances that code changes will break part of the project, e.g. using [the R package ‘testthat’](https://testthat.r-lib.org/).

**Small and regular commits:** Ideally each commit should describe a discrete change or group of associated changes, involving few files. *It should be possible with RStudio integration to stage individual lines as well as chunks, although some users on the AP have reported problems with this.*

**Delete merged branches:** It’s good to keep things tidy so that people know what work is still ongoing. Your commit history will remain if it is part of the tree of merged work. Note that closed PRs for abandoned branches [will remain and retain commit history](https://github.com/moj-analytical-services/intro_to_github_training/pulls?q=is%3Apr+is%3Aclosed).

* 1. **Security considerations when using GitHub**

GitHub is a website with public areas accessible to all internet users. Within GitHub we have a private internal area called moj-analytical-services. Despite this, it is important to think about the sensitivity of the information you push to Github. You should not add data to GitHub that isn’t already publicly available - this would be a breach. See here for more guidance on security and what to do if you accidentally pushed sensitive data <https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#security-in-github>.

Amazon S3 buckets should be used to store data. It is a web-based cloud stage platform and all data and final analytical outputs in Data and Analysis is stored in an Amazon S3 bucket. To learn more about Amazon s3 buckets, please refer to section 5 of the Introduction Using R on the AP training session. Previous recordings and accompanying materials of sessions can be found here - <https://moj-analytical-services.github.io/ap-tools-training/>.

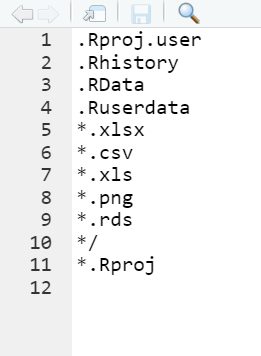
**4.1 Git ignore files**

You may also not want to push some files/information/images that are required for your project onto GitHub due to sensitive information. This could also include files such as the renv folder, which is used for package management (other than the renv.lock file and renv/activate.R, in RStudio, and requirements.txt, for python module metadata, the other files in the renv/ folder contain copies of the packages you are using which can be quite large and not necessary to be tracked).

To help ensure we don’t push something by accident to GitHub, we use Git ignore files. You should use the Git ignore file in your repository to tell Git which files and directories to ignore when you make a commit – allowing you to control which file types are shared on the GitHub website. Once the Git ignore file is on Github, then others can pull this from the project into RStudio to ensure they also follow the rules of pushing information to GitHub.

Git ignore files have a list of recommended files to ignore – you can edit and save this to ensure it is relevant to your project. For example, if I saved some data for a report in a .xlsx file in the same directory as my R project, GitHub would attempt to upload this to the GitHub website with any code amendments. This could potentially lead to a breach as sensitive data has been shared online.

To avoid this, we can use a .gitignore file – as seen below. We can add file types with a .gitignore file using simple syntax.



This .gitignore file stops .Rproj.user, .Rhistory, .RData, .Ruserdata to be uploaded to GitHub when the project is pushed (the dot in front is because they are technically hidden files in R studio). It also doesn’t allow .xlsx, .csv, .cls, .png, .rds, .Rproj . An \* in front of a file type means ‘any characters followed by’. You can use an exclamation mark to include files that you want to be an exception to these rules.

Users should also avoid putting outputs of R markdown reports on GitHub as this could contain official sensitive information and results of analysis. Adding \*.docx to the .gitignore file would stop any documents being added to GitHub.

We would also recommend adding the core file to the .gitignore file. This is generated when using RStudio crashes when trying to work with large datasets. These should not be committed to gitignore and should be deleted when they appear in the files section of RStudio. More information can be found here [Security in GitHub - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#protecting-information-in-github).

We would advise having a look at the Developing R packages & RAP ways of working course <https://github.com/moj-analytical-services/rpackage_training#10-excluding-sensitive-data)>, which includes a [gitignore template](https://github.com/ukgovdatascience/dotfiles/blob/master/.gitignore) developed by ukdatascience.

**5. Further information on Git and GitHub**

**5.1 Advanced topics**

We have added some information and links below on some other aspects of GitHub that users could explore that haven’t been covered in this session. These are more advanced features of GitHub and are not required to be able to use Git and GitHub as a beginner.

* Git releases/versions - useful for package development and those working on production/publication code - [About releases - GitHub Docs](https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases).
* Git stash – when you want to store changes to code when they are not ready to be committed and revert to a previous commit [git stash - Saving Changes | Atlassian Git Tutorial](https://www.atlassian.com/git/tutorials/saving-changes/git-stash)
* Git detached head - [Git Detached Head: What Is It & How to Recover (cloudbees.com)](https://www.cloudbees.com/blog/git-detached-head)
* Packages on Github - Github allows you to host your software packages internally, privately or publicly and use them within your projects.
  + To learn more about creating packages and about reproducible ways of working, then listen to the session on Developing R packages and RAP ways of working - <https://moj-analytical-services.github.io/ap-tools-training/ITG.html#ITG>.
  + There also some packages created by analysts with MoJ that are hosted within moj-analytical-services on GitHub – for example, mojrap which includes functions that have been created in various Reproducible Analytical Pipelines <https://github.com/moj-analytical-services/mojRpackages>
  + [Install packages from GitHub - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/github/install-packages.html#install-packages-from-github)

**5.1.1 Additional information on Gitflow**

Gitflow is one practice that we recommend you give serious consideration to adopting to any project that involves multiple contributors. There was a bitesize session on Gitflow [here](https://web.microsoftstream.com/video/fcf60e66-e7fe-4bf7-9ab4-e2c692637018?channelId=5a6012a2-efd6-4b86-902d-98c864427caa).

**The dev branch**

One feature that we recommend is using an additional branch as an insulator between the main branch and feature development branches, usually called dev.

We saw how git can handle changes to the same file without there being merge conflicts. But even if colleagues work on different files and avoid conflicts, it doesn’t mean that all merged code will function as they intended.

Following a fast-forward merge (a merge where only one branch has changed since divergence), the code in a merged branch (main) will work if the code in the merging branch (Anna) worked. But that isn’t necessarily the case if Bruce’s work then merges in. Anna and Bruce’s code changes may not work when combined. If so, their project has a problem that Git doesn’t know about. It would only emerge when they try to run the code in the new merged main branch.

Using a dev branch provides an easy way to mitigate. It sits between feature branches and the main branch. Like main, we don’t make changes directly in dev. Anna and Bruce branch from and merge back into dev, instead of main.

With that in place, it's easy to manage things. They test the code in dev. If dev works, then after they’ve done their work it can be fast-forward merged to main. But if the combination of Anna and Bruce’s changes has broken the code, they can make another branch to fix it, and then do a fast-forward merge to dev before the fast-forward merge of dev into main.

The dev branch is not a perfect solution to be relied upon its own – *e.g. what if there’s a need for another branch off while the fix is happening? That person may have to work with broken code or may reintroduce the same problem back into dev later*

*(For future reference, other possibilities for dealing with this specific problem include automated integration tests with github actions, and an alternative to merging called rebasing, but these are technically challenging and/or complex and beyond the scope of this class).*

**5.1.2 The ‘undo’ commands**

There is a number of different Git commands that can be used to effectively undo your work, after you’ve committed. Let’s start with changing saved objects in the working directory.

**git restore**

**git restore** is used to restore a specific file to a previous state.

You must specify a filepath, e.g.

**git restore path/to/file**

By default it will restore the file to the state that it was in at the last commit. In RStudio with git integration you can also do this with the revert button (not to be confused with git revert). But you also have the option of specifying a different branch, if you so wish, e.g.

**git restore --source=branchname path/to/file**

**git revert**

git revert is a useful command that works at the commit level. It creates a new commit that reverts a previous one.

So if you run

**git revert HEAD**

I will create a new commit that reverts the last one. You will get prompted to enter a method. There is a default so you can just press caret -X

You can actually revert to a state that undoes any previous commit by providing the commit hash. You can find these a number of ways. You can find them with **git log or git reflog**, or you can find them on github. You simply append it to your git revert command, e.g.

**git revert 7b668c1**

This creates a commit that undoes the diffs in that commit.

The great thing about **git revert** is that the commit history is preserved, which is generally considered good for collaboration.

If you’re wanting to undo some work someone else has done that you’ve pulled, you should be using

**git revert**

It is also arguably conducive to transparency to use git revert within your own code development to keep a record of what you’ve done

**git reset**

Finally, we make you aware of another option: **git reset.** This allows you to move back to a previous commit (e.g. git reset HEAD~1 will take you back one). Although it can make your commit history cleaner to look through, it is considered more risky as commits that have been reset over will ultimately be ‘orphaned’, which ultimately causes discarded work to become completely irretrievable. It’s therefore less conducive to transparency and accountability and recovering work that you might have changed your mind about discarding. In particular, it should not generally be used to undo others’ work. However, it can be a useful tool in some contexts, such as when working locally.

**5.2.2. Merging and resolving conflicts on the command line via RStudio**

In addition to merging and resolving conflicts on GitHub, you can also merge your branch locally using the terminal.

To merge a branch locally using the terminal (*With base branch* ***main*** *and merging branch* ***feature)***

1. Switch to the destination branch with git checkout, for example if you want to merge a branch into the main branch, run **git checkout main**
2. Ensure the destination branch is up to date with the remote repo with: **git pull**
3. Use git merge to merge in the branch containing changes: **git merge feature**

If there is a conflict, you will be informed of this in the terminal, and the files will appear with the markers indicating alternative content as described earlier, in which case, you will need to follow these additional steps:

1. Resolve the conflict in all files as described earlier, ensuring that all marker rows are removed.
2. Test code.
3. Stage and commit files with an informative message.
4. Push branch to GitHub.
   1. **Useful links**

Analytical Platform Guidance and slack channels

* Use this link to access the AP control panel: <https://controlpanel.services.analytical-platform.service.justice.gov.uk/tools/>.
* Lots more helpful guidance can be found in the MoJ Analytical Platform Guidance: <https://user-guidance.services.alpha.mojanalytics.xyz/>
* Specific Git section on AP guidance and how to set up slack channels: [Git and GitHub - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/github/index.html)
* Instructions for getting your GitHub account connected to your AP account: <https://user-guidance.services.alpha.mojanalytics.xyz/github/set-up-github.html#set-up-github>
* Gitflow in the MoJ RAP (Reproducible Analytical Pipeline) manual: <https://moj-analytical-services.github.io/rap-manual/git-flow.html>

Training resources

* Analytical platform and related tools training: <https://moj-analytical-services.github.io/ap-tools-training/index.html>
* Previous recordings and accompanying materials of sessions can be found here [3 Internal Training Group materials | Analytical Platform and related tools training (moj-analytical-services.github.io)](https://moj-analytical-services.github.io/ap-tools-training/ITG.html#ITG)
* The repo for this course <https://github.com/moj-analytical-services/intro_to_github_training>
* The GitHub repo for Introduction using R on the AP: <https://github.com/moj-analytical-services/intro_using_r_on_ap>
* The GitHub repo for the Introduction to R course: <https://github.com/moj-analytical-services/IntroRTraining>
* Collaborating as a team using GitHub projects <https://web.microsoftstream.com/video/c8821c1c-7779-443a-8084-8b1e01897ba2>
* Github flow bitesize session [Watch a video | Microsoft Stream (Classic)](https://web.microsoftstream.com/video/fcf60e66-e7fe-4bf7-9ab4-e2c692637018?channelId=5a6012a2-efd6-4b86-902d-98c864427caa)
* Feedback form on training in D&A <https://airtable.com/shr9u2OJB2pW8Y0Af>

Cheat sheets/further links

* <https://github.com/moj-analytical-services/intro_to_github_training/blob/main/github_cheatsheet_RStudio.pdf>
* [git-cheat-sheet-education (github.com)](https://education.github.com/git-cheat-sheet-education.pdf)
* [Let’s Git started | Happy Git and GitHub for the useR (happygitwithr.com)](https://happygitwithr.com/index.html)
* [Hello World - GitHub Docs](https://docs.github.com/en/get-started/quickstart/hello-world)
* [Git from the inside out (maryrosecook.com)](https://maryrosecook.com/blog/post/git-from-the-inside-out)
* [Git from the inside out - YouTube](https://www.youtube.com/watch?v=fCtZWGhQBvo)