**Introduction to Git and GitHub**

**Set-up steps**

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 and GitHub are and explain the key benefits of using them alongside the Analytical Platform and RStudio. Step-by-step instructions will be provided about how to use Git in RStudio and how to collaborate with others using GitHub.

**Sessions aims**

1. Describe what Git and GitHub are
2. Explain the basic concepts underpinning them
3. Demonstrate the basic Git and GitHub workflows
4. Give awareness of more advanced features that you are likely to come across
5. Ways to approach Git to enable effective use and collaboration
6. Describe some good practice for effective collaboration
7. Mention security considerations when using GitHub

The session is aimed at people who are new to Git and GitHub. It will explain the basic workflows involved. We will mention few more advanced features that it is helpful to be aware of but we will not go into detail on how to implement them.

There is an exercise giving the opportunity to practice using the basic Git and GitHub workflow.

We will have a 10-minute break around the middle of the session.

* 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 but much more powerful and flexible. Git keeps track of the history of all changes made to a project’s files and handles the process of combining changes made by different contributors.

GitHub is an online platform that hosts coding projects. It uses Git software and provides features that help multiple people work on a project at the same time. The online storage area for a project on GitHub is a repository, or “repo” for short.

GitHub is a public website, but Data and Analysis has a private space called the “MoJ Analytical Services” organisation. Access to repositories within the organisation can be restricted to certain individuals, granted to all members of the organisation, or made publicly available. The Analytical Platform user guidance contains guidelines on which access setting to use. All code written on the Analytical Platform should be stored in a GitHub repository, including Python scripts and Jupyter notebooks, although In this session we will focus on using Git in RStudio.

To gain access to the Analytical Platform, you will have needed a GitHub account because the Analytical Platform uses GitHub authentication to provide access to all Analytical Platform services.

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

**Benefits of Git**

Some of the benefits of Git are as follows:

* **Version Control:** Having access to previous versions of your code ensures your work is secure and reproducible.
* **Collaboration:** Multiple people can work on a project simultaneously, with Git keeping track of all the separate changes and managing the process of merging them together.

**Benefits of GitHub**

Benefits of GitHub include:

* **Online access:** The central copy of your project is held online and can be viewed and accessed by collaborators, according to the repository visibility settings.
* **Collaboration:** Everyone with access to the repository can view all contributions. GitHub provides tools to discuss and review each other's work before it is merged into the main branch.
* **Audit trail:** The review process for integrating changes into the repo can take place within GitHub, preserving a record of changes and the QA carried out.
* **Project documentation:** README files are shown alongside the project folders they sit it and provide a visible way to document the purpose of the repo and any other documentation you choose.
* **GitHub issues:** The GitHub Issues feature allows you to log and keep track of suggested changes to the repository, such as enhancements or bug fixes.

**1.3 Git in RStudio**

We will discuss how to use Git and GitHub via RStudio on the Analytical Platform.

There are two ways to use Git in RStudio:

* **The graphical user interface (GUI)** – RStudio contains a Git pane with information and buttons for interacting with Git and GitHub. The most commonly needed actions are available.
* **The terminal** – this is the command line interface. This is the general way to interact with Git and isn’t specific to RStudio. (For example, the terminal works the same way using JupyterLab on the Analytical Platform). The full range of Git commands are available.

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

1. **Conceptual overview of Git and GitHub**

**2.1 Commits and branches**

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 look at it or switch back to 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

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.

Git calls the series of commits a branch. By default, Git labels your initial branch "main". Note that if you are using an older repo, the “main” branch may be called “master”.

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

Typically, you don’t want to develop by adding commits to the main branch. If you want to develop a new feature, or change something in the project, you should create a new branch. Then you can develop the new feature by making commits to the new branch. Once the feature is working and has been quality assured, you will merge it back into the main branch. Following this workflow ensures that the main branch is always a fully functioning and stable version of the project.

A 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 |

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

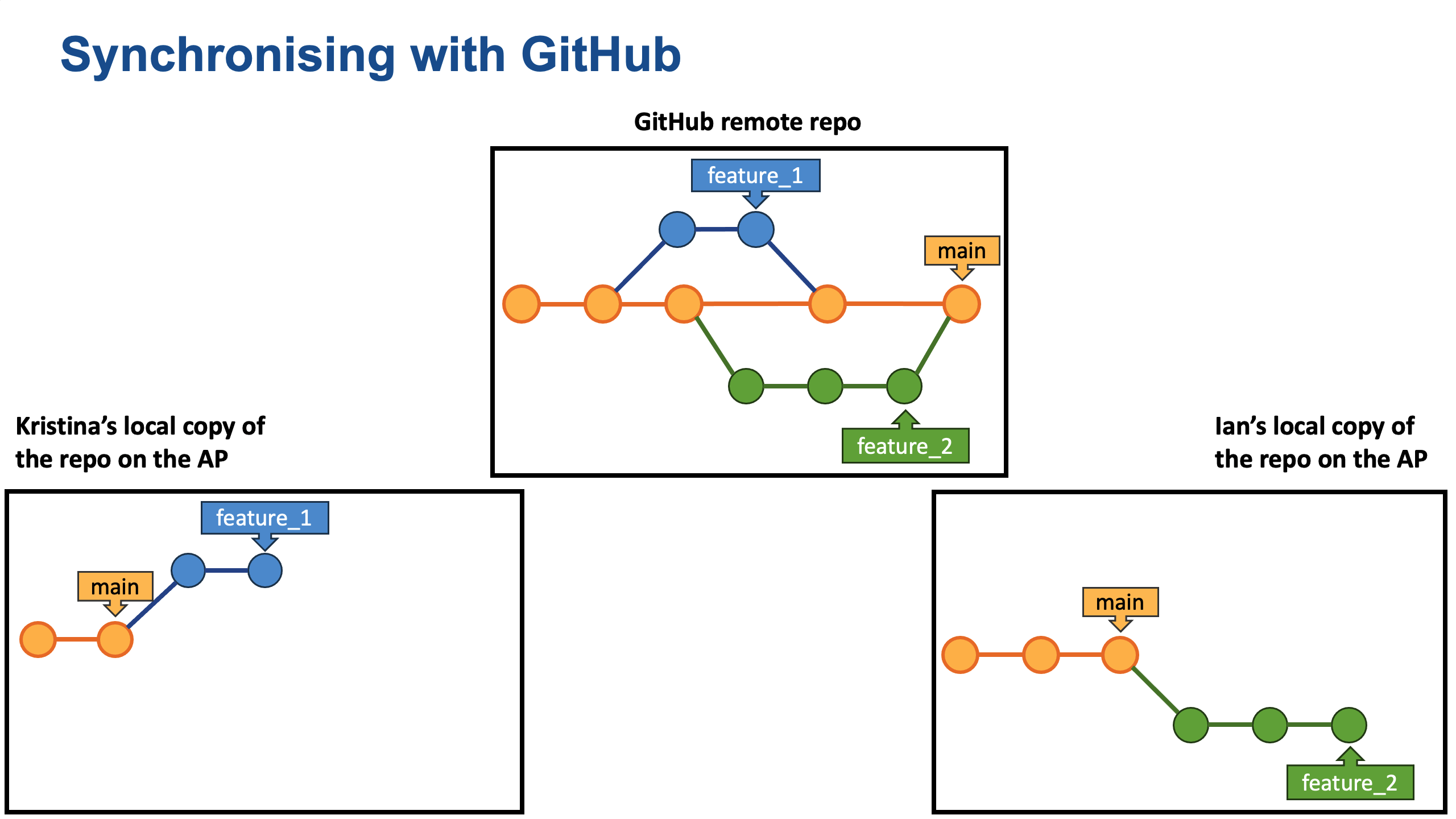
Eventually, once your feature is complete, you can tell Git to merge it back into the main branch. Git handles this merge process so that the main branch then contains all the changes from both branches. If there are conflicting changes, Git will require you to resolve them manually.

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.

* 1. **Synchronising with GitHub**

****

* 1. **File states**

**A black background with orange rectangles and white text

Description automatically generated**

1. **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 use in 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 into 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.

* 1. **Starting a repository and creating a branch**

**3.1.1 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 [Introduction to using R on the Analytical Platform](https://github.com/moj-analytical-services/intro_using_r_on_ap/tree/main)). 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. If not, 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 presentation.

* + 1. **Create a branch**

Once you have cloned a repo into your local directory on the Analytical Platform, you are ready to contribute to the project. Normally you should create a new branch to contribute changes (as explained in section 2.1). When you create a new branch you will need to give it a name. Here are some tips for naming branches.

1. Use a descriptive name that describes the branch’s purpose
2. Be concise
3. Use – or \_ as separators between words
   1. **Editing files in your repository and committing your changes**
      1. **Edit some files**

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 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.

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 lines 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 can 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.

**3.3.2 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. There are two ways to merge a branch, either locally in the terminal or on GitHub via a “pull request”. If you are merging into the main branch, you will usually want to do that as a pull request on GitHub so that colleagues can see and review the changes before they are merged.

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. Pull requests also help with project management – the discussion thread can be used to explain the proposed changes and link to a project planning platform or issues log. The pull request page also provides an audit trail that preserves details of the review process and any discussion that took place during review.

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 4.

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.

**3.6 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.

**3.7 Exercise:**

Now we will practice the four steps explained in Section 3 – 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 3.1.1. <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\_folder’ and then create a 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.

**4. Advanced topics**

**4.1 Resetting changes**

Git has options for making changes to work you have already staged or committed. It is straightforward to unstage changes that you have not yet committed. It is also straightforward to make changes to your most recent commit – provided you haven’t pushed it to GitHub. It is easiest to not edit an earlier commit than that, or a commit you have already pushed to GitHub.

To unstage a file that you have staged but not yet committed, untick the checkbox by the filename in the RStudio Git pane.

If you want to edit your most recent commit, you can amend it using the RStudio interface. Make the changes you want, stage them, and click on “commit”, but before typing a commit message, tick the “Amend previous commit” checkbox.

The terminal commands for resetting changes include:

* git reset, which can be used to reset the staging area (but can also be used to undo commits)
* git commit --amend allows you to add your staged changes to the most recent commit.
* git revert, to make a commit that reverses a previous commit

The details of using those commands are beyond the scope of this course, but further details can be found at the links in section 6.

**4.2 Stashing changes**

git stash is a command that temporarily stores your work without making a commit, allowing you to re-apply it later. An example where this can be useful is if you need to temporarily switch branches but aren’t ready to commit yet. Committing is generally easier and safer, but git sometimes prints error messages telling you that you need to either commit or stash your changes, so it is worth at least being aware of the functionality.

**4.3 Merge conflicts**

Git can normally handle the process of merging changes made in different branches automatically. However, if two branches have made changes to the same lines of code, Git does not know which change to accept, and will inform the user that there is a merge conflict that must be resolved manually before the merge can complete.

Git will add the changes from both branches to the file containing the conflict, surrounded by conflict markers as shown below.

A screenshot of a computer code

Description automatically generated

The code between <<<<<<< and ======= is the code in the current branch. (“HEAD” is the label Git uses to refer to the current commit.) The code between ======= and >>>>>>> is the code in the branch that is being merged in. To continue with the merge, edit the code to contain the final version you want. That could be the version from either of the two branches or some new combination. Then delete the conflict markers. To complete the merge, stage the file and commit.

See the additional resources section below for some links with more details of resolving conflicts using RStudio and the Git command line.

4**.4 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 are 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.

**Small and self-contained commits:** Commit regularly. Ideally each commit should describe a discrete change or group of associated changes.

**Clear commit messages:** Commit messages should summarise the changes made so that it is easy to read through the commit history and identify the commit that introduced a particular change.

**Small branches and regular merges:** Make sure each branch has a clearly defined purpose and make sure the scope doesn’t expand to include anything extra as you work on it. This will make it easier to review pull requests and reduce the risk of merge conflicts. If a branch is getting too big, consider whether you can split it up.

**Pull requestion descriptions:** Use the pull request description box to describe the purpose of the pull request and summarise all the changes introduced. You can also use this space to suggest to colleagues how to review and test the changes.

**Branch protection rules:** Project administrators can add branch protection rules to control how changes can be made to a branch. For a team project, it is a good idea to protect your main branch by requiring pull requests before merging and by requiring approvals. That will ensure no changes are made to the main branch that haven’t been reviewed and approved by a colleague.

**Delete merged branches:** Once a branch has been merged, it is safe to delete it since all the commits will be part of the target branch. The closed pull request page remains, so there is a record of what was merged, and there is an option there to restore the branch if needed. Deleting merged branches makes the active branches easy to see since they aren’t obscured by a long list of old branches.

**Document your repository:** When you create a new repository, you will be given the option of initialising it with a README file. The [Analytical Platform User Guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#readme) recommends selecting this option. (Alternatively, you can add a README.md file to the top level of the repository at a later time.) The README file can be used to tell readers the purpose of the repository and how to get started using it.

**Managing repository access:** Access to repositories within the moj-analytical-services organisation is controlled using GitHub teams. See the Analytical Platform User Guidance on [managing access in GitHub](https://user-guidance.analytical-platform.service.justice.gov.uk/github/manage-access.html#manage-access-in-github).

**5. Security considerations when using GitHub**

**5.1 Repository visibility**

GitHub is a public website visible to anyone on the internet. When creating a repository, you have the option to make it public, internal (to MoJ Analytical Platform users) or private. For guidance on which to use, see the Analytical Platform User Guidance on [repository visibility](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#repository-visibility). For most projects that contain files that should not be in the public domain, the guidance recommends the “Internal” setting.

**5.2 Storing data**

The Analytical Platform Guidance page [Security in GitHub](https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#security-in-github) says that GtHub should primarily be used to store code. Data, especially sensitive data should use warehouse data sources, specifically the Amazon S3 service. The Analytical Platform is set up so that by default, data files will not be tracked by Git or uploaded to GitHub. If you accidentally publish sensitive data to GitHub, see the Analytical Platform guidance on [accidentally publishing data to GitHub](https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#accidentally-publishing-data-to-github).

For information on storing data on Amazon S3, refer to the Analytical Platform User Guidance page [Amazon S3](https://user-guidance.analytical-platform.service.justice.gov.uk/data/amazon-s3/#amazon-s3).

**5.3 Git ignore files**

A .gitignore file can be used to list files that git should ignore. This can include configuration files and data files. There is an option to select a .gitignore template for R when you create a repository, which is recommended by the [Analytical Platform User Guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#gitignore).

**6. Additional resources**

MoJ resources:

* Git and GitHub help in the MoJ Analytical Platform User Guidance:

<https://user-guidance.analytical-platform.service.justice.gov.uk/github/index.html#git-and-github>

* Training material and recordings of courses, including this course, from the MoJ R (and SQL) Training Group:

<https://moj-analytical-services.github.io/ap-tools-training/ITG.html#ITG>

Cheat sheets and other resources:

* Cheetsheet “Using Git and GitHub with RStudio”:

<https://rstudio.github.io/cheatsheets/git-github.pdf>

* Git cheatsheet from DataCamp:

<https://www.datacamp.com/cheat-sheet/git-cheat-sheet>

* Online book on using Git and GitHub with R, "Happy Git and GitHub for the useR”, by Jenny Bryan:

<https://happygitwithr.com/>

Undoing changes:

* Undoing the most recent commit, from Jenny Bryan’s book:

<https://happygitwithr.com/reset>

* Detailed tutorial on undoing commits and changes, from Atlassian:

<https://www.atlassian.com/git/tutorials/undoing-changes>

Merge conflicts:

* Dealing with conflicts in RStudio, from Jenny Bryan’s book:

<https://happygitwithr.com/git-branches#dealing-with-conflicts>

* Atlassian tutorial on understanding and resolving merge conflicts:

<https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts>