A Quick Guide to Git & GitHub

Contents

Preface

This is a quick guide for git, including GitHub, and the integrations into RStudio. There are references for further details and training on each of these topics. A PDF copy of this guide can be downloaded using the PDF button in the toolbar - please note that you'll need to download this again after any changes or updates to ensure you have the latest copy for guidance and recommended workflows. It will always be possible to bookmark this page and return to whenever required.

In this guide, there are six sections which identify setup, steps to follow, "cheat-sheets", and further reference materials:

1. Introduction
2. Project Workflow - using RStudio
3. GitHub
4. Git
5. Cheat sheet
6. References

The book is created using bookdown (an R package) and hosted using GitHub. This should allow convenience for development and use, hassle-free updates, and contribution from users (from typos to suggestions).

Introduction & Setup

As you may already know, or will find out, Git is a command line (Shell/Bash) tool used for version control. Throughout this guide, the aim has been to provide a clear workflow and guidance that will allow you to work with these tools with minimal command line interaction. However, if you'd like to know more about this topic, Software Carpentry offer a fantastic Introduction to Shell lesson.

The target audience for this guide is analysts within NHS National Services Scotland, particularly those who use R for data analysis. It may be that others find use in this guide, if so, fantastic! If you feel that something is missing or could be better, please raise a GitHub issue.

Throughout this guide where there are code examples, <> are used to show areas that define the users input. The arrow symbols shouldn't be entered along with the rest of the command.

Version Control, Git, and Remotes

Version Control

A version control system (VCS) tracks the history of changes as people and teams collaborate on projects together. As the project evolves, teams can run tests, fix bugs, and contribute new code with the confidence that any version can be recovered at any time. Developers can review project history to find out:

- Which changes were made?
- What was changed?
- Who made the changes?
- When were the changes made?
- Why were changes needed?

Software teams that do not use any form of version control often run into problems. Developers who have never used version control may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new "final" version. Perhaps you've dealt with commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

Git

Git is a version control system (VCS). We recommend that Git is used along with a hosting repository (a remote), such as GitHub (or Gitea, GitLab, etc.) and that the GitHub Workflow is followed. However, Git can be used locally while working on a project. The steps outlined in Quick Start Git are for this purpose and should only be followed for a personal project that is **local or on your personal network folder**. For collaborative work, see Quick Start GitHub.

According to the latest Stack Overflow developer survey, more than 70 percent of developers use Git, making it the most-used VCS in the world. Git is commonly used for both open source and commercial software development, with significant benefits for individuals, teams and businesses:

- Entire timeline of changes, decisions, and progression of any project in one place.
- With a VCS like Git, collaboration can happen any time while maintaining source code integrity. Using branches, developers can safely propose changes to production code.
- Businesses using Git can break down communication barriers between teams and keep them focused on doing their best work. Plus, Git makes it possible to align experts across a business to collaborate on major projects.

A **repository** is Git's encompassing of the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as snapshots in time called commits (versions), these commits can be organized into multiple lines of development called branches (seperate areas of development on the same project). Using the command line or other ease-of-use interfaces, a Git repository also allows for: interaction with the history, cloning (copying), creating branches, committing (saving a version), merging (bringing two versions together), comparing changes across versions of code, and more. This is a lot of information but as you start using Git it'll all embed and the terminology will be second nature.

Branches

Creating branches allows development and work to continue without affecting the main stream, it also allows many people to do work in parallel without overwriting each other's work. Branching works by taking a copy of the main code and then later merging back when the code is ready. Before the marge takes place, reviews can take place to gain feedback and ensure the implemented changes will have the desired effect.

Remotes

Version control has great potential to facilitate collaborative working with colleagues and third parties. This can be utilised with the use of "remote" repositories - copies of a project hosted away from the workspace e.g. on GitHub or Gitea. There's more about this later when we talk about GitHub, but for just now, this is a little introduction so you're familiar with the term.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web/private network rather than on someone's laptop. This is where remotes come in. Most programmers use hosting services like GitHub, BitBucket or GitLab to hold those master copies. We currently use **GitHub.**

GitHub is a Git hosting repository that provides users with tools to ship better code through command line features, issues (threaded discussions), pull requests, and code review. GitHub builds collaboration directly into the development process. Work is organised into repositories, where users can outline requirements or direction and set expectations for team members. Then, using the GitHub Workflow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and discuss changes, and merge pull requests once everyone is on the same page. See GitHub Workflow for more details.

Workflows

GitHub Workflow

This is our recommended workflow. The workflow is branch-based and involves keeping the master branch clean and 'production ready'/fully-functional at all times.

1. **Create a branch** - topic/per-person branches created from the master allow teams to contribute to seperate issues at the same time.
2. **Add commits** - these are snapshots of development within a branch and create safe, revertible points in the project's history.
3. **Open a pull request** - pull requests are the mechanism for which modifed code within a branch is merged back into the master branch, i.e. bringing the master branch up to date with changes made within a branch.
4. **Discuss and review code** - teams participate in code reviews by commenting, testing, and reviewing open pull requests. Pull requests should be approved by at least one collaborator - we recommend you assign a specific reviewer(s) to check the work. In order to ensure the master branch remains 'production ready' we also recommend that the master branch is protected and any changes are tested by the reviewer before approving.
5. **Merge** - upon clicking merge, GitHub automatically performs the equivalent of a local `git merge` operation. Although you'll likely never have to manually enter this command, it's good to understand that the buttons available for interaction with git are just performing these commands for us. GitHub also keeps the entire branch development history on the merged pull request.
6. **Deploy** (*optional*) - teams can choose the best release cycles or incorporate continuous integration tools and operate with the assurance that code on the deployment branch has gone through a robust workflow.

Setup

Git

These intructions should only need to be followed once or when setting up a new device.

1. **Install Git** - Git is a free, open-source, software available from https://Git-scm.com/. If you work within NHS National Services Scotland, request the software from IT through the Service Portal. Once authorised, this will allow you to download it from the Software Center on your machine.

This will install three applications on your computer, go to Start > Git (folder):

- **Git Bash** - this is the command line (Shell) interface for Git. This will allow you to enter Git commands, i.e. `git init`, and can be launched by right clicking in a directory/folder and selecting "Git Bash Here".
- **Git CMD** - this is a deprecated version, similar to the Bash interface, and shouldn't be used here.
- **Git GUI** - this is Git's version of a Graphical User Interface. This will perform the same functions as "Git Bash", but employs a point-and-click interface instead. This guide does not cover Git GUI and instead focuses on the integrated Git GUI in RStudio.

2. **Configure your details** - Git needs to know who you are, use the following commands to configure your username and email using Git Bash (*if you're a GitHub user, use the same email and username you're registered with on GitHub.* This is a necessary step for GitHub to recognise the account where the commits come from, even with proper authentication GitHub will use these details to assign the commits.) Remember not to type the arrow symbols `<>` when entering the command:

- `Git config --global user.email <NHS email address>`
- `Git config --global user.name <your name>`

At any time you can check what the current user details are using:

- `Git config --global user.email`

GitHub

These instructions are for first-time users of GitHub and highlight the steps in setting up a GitHub account.

1. **Sign up** - In order to use GitHub, you'll need to have a personal GitHub account, you can set that up on the GitHub website. If using your NHS email address or LDAP username ensure you **don't use the same password as your NHS email account or LDAP login**.
2. **Edit your details** - you will be able to set these items up as part of the sign up process. Otherwise, follow the links and edit them now.

- Go to your Profile Settings and add your details with a picture of yourself if you feel comfortable to do so.
- Go to your Account Settings and make sure your username is set to be your full name. This helps to identify users as generated GitHub handles tend to pretty obscure.

3. **Get added to Organisations** - this step is optional.

- If you're doing work within the Transforming Publishing team. Email one of the admins here to request being added to the organisation.
- If you work within Public Health & Intelligence (PHI), the Health & Social Care Scotland GitHub may be of interest. Repositories can be requested and set up for team working.

RStudio

First R and RStudio must be installed on your machine. Just as with Git, if you work within NHS National Services Scotland these can be requested through the Service Portal. Once authorised, this will allow you to download it from the Software Center on your machine. You may also be able to use the server versions of RStudio and the project workflow will be the same.

1. **Once in RStudio, go to Tools in the menu bar > Global Options > Git/SVN**:

2. **If "Git executable:" shows "*(none)*", click "Browse" and select the Git executable installed on your system.** - On Windows, the filepath should point to folder git was installed to and finish `.../bin/git.exe`. If you're on the server, it should be `/usr/bin/git`. That's it for attaching RStudio and Git. This will allow you to utilise the RStudio Git GUI tools inside R Projects.

3. **Create an SSH Key** - this step is recommended for users who will be using a remote such as GitHub with their project. The other option is HTTPS (which doesn't require setup), however SSH is a faster and more convienent method of connecting to GitHub.

    1. If you see anything within the 'SSH RSA Key:' box, you can skip this step, else click on 'Create RSA Key…', click 'Create' and close the dialog box that appears confirming the key creation. If this doesn't work, there may be access/priviledge issues which will need to be resolved through IT and logging a ticket through the Service Portal.
    2. Now, click on 'View public key' and copy the text string that appears in a dialog box.
    3. Go to your GitHub SSH Settings and click 'New SSH key'.
    4. Give the key a meaningful title that you will recongnise and paste the key from RStudio. Complete the process by clicking 'Add SSH key'.

Project Workflow with RStudio

RStudio (the most commonly used R IDE) comes with some useful git integreation in the form of buttons. While some advanced git features still require the command line, RStudio has a nice interface for many common git operations. As such, this guide and the supporting workshops focus on utilising this interface for learning and utilising git.

RStudio allows us to create a project associated with a given folder/directory to bundle everything together. It is through a project that RStudio gets its git integration, so projects can be used to version control any file within the associated directory. RStudio is able to connect to remote repositories on platforms such as GitHub, so the command line is not necessary to "push" or "pull" code from there.

Quick Start RStudio

1. **Set up R Project**

- **New Project** - any time you do any work that requires code, you should create an R project. This keeps your work together, seperates settings for different projects, and allows the use of version control. To do that, go to File in the menu bar and select "New Project". In the window that appears, name your project, choose a folder/directory for where to store it, and ensure "Create a git repository" is selected.

- **Existing Project** - if you have already started working on an R Project without initialising git, you can start using it at any point (the earlier the better). On an existing R Project go to Tools > Project Options > git/SVN > and select "git" in the version control drop down. This will ask if you want to initialise a repo, click "yes". To use the terminal/command line, go to: Tools > Terminal > New Terminal, in that panel type `git init`.
- **Exisiting Remote Repo** - if you're going to be working on a project that already exisits on a remote repository (e.g. a GitHub project), you can link to this/clone it through RStudio too. Go to File > New Project > Version Control > Git and enter the details in the screen that follows (this will create a folder with a copy of any contents that were in the GitHub repository).
  - **Repository URL**: Using SSH, this guide's link is: "git@github.com:NHS-NSS-transforming-publications/git-guide.git".
  - **Project directory name**: this is up to you and will be used to name the folder/directory that contains a copy of the repo. Give it any meaningful name, for example this guide is named "git-guide", the same name as on the remote repo.
  - **Create project as subdirectory of**: this is the location of where the folder/directory that contains the copy of the repo will be stored.

2. **Change, stage...** - RStudio will keep track of all changes made to the files within your R project (not just R files). Any changes made will show up in the "Git" tab, which can be found in the same pane as the R environment. This is typically the top-right pane, but their positions can be changed so it may not be in all cases. When you're ready to commit, save the files you're working with and stage them by ticking the box to the left of the files in the Git pane, and then click on the "Commit" button.

3. **Commit** - After clicking on "Commit", a window will appear with the changes made for review and a prompt for a "Commit message". Your commit message should describe the changes made and be consise, meaningful and written in imperative mood. Then, just click on "Commit"! *This has saved all the changes to the project. Going forward, it will be possible to look back and see the full history of changes to the project, who made those changes, and revert to previous versions where necessary.*

4. **Branching** - As part of the recommended workflow, you should use branches to develop individual features, this means that you can edit files but keep an original to revert back to if required. Additionally, when team working, it allows for people to work on different things in parallel. This can help improve efficiency as people don't have to wait for others to finish what they're doing before starting something else.

- **New Branch** - To create a new branch in RStudio, just click the "New Branch" button in the Git pane and give it a name. To switch between branches, click the dropdown to the right of the "New Branch" button and click the name of the branch you want to switch to. *(When you select a branch, the state of the files in that branch will appear in both the Files pane in RStudio, and in your computer's file explorer.)*

- **Merge Branches** - When the working branch should be merged back to the master branch, this must be done on the command line as there is no way to do this using the GUI. This can be accessed from the "More" menu in the Git pane and selecting "Shell" or in the RStudio Terminal window. Once you're in either of these, enter the command `git merge <branch>`.

```
username@NSS000000  ~/Documents/demo-project  (master)
$ git merge <branch>
```

5. **Link to GitHub** - If this is a new project, it won't be linked to GitHub yet. *See Quick Start GitHub for instructions on how to set up a repo on GitHub.* RStudio currently doesn't have a GUI option for this and must be done on the command line. As above, this can be accessed from the "More" menu in the Git pane and selecting "Shell" or in the RStudio Terminal window. Once you're in either of these, enter the command `git remote add <name> <url>` (*`<name>` is any name to refer to the GitHub connection but `origin` is the most common/preferred name to use*), pulling in any changes that have been made on GitHub in the meantime using `git pull <name> <branch>` and then finally sending the changes to GitHub using `git push <name> <branch>`.

6. **Pull and push** - After linking the project to a GitHub repository, RStudio has a GUI interface for pulling and pushing changes. It's good practice to pull from GitHub to check for changes that have occured since the last time and to help avoid merge conflicts. After that you can click push to send all your commits to GitHub, now everything is in-sync again.

GitHub

Systems like git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub, BitBucket or GitLab to hold those master copies. Here, we'll explore GitHub.

GitHub is a git hosting repository that provides users with tools to ship better code through command line features, issues (threaded discussions), pull requests, and code review. GitHub builds collaboration directly into the development process. Work is organised into repositories, where users can outline requirements or direction and set expectations for team members. Then, using the GitHub Workflow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and dicsuss changes, and merge pull requests once everyone is on the same page.

Quick Start GitHub

New Project Repository

Just like in Git, a repository is used to organise a project. Repositories contain files, folders, images, data-sets (with caution), and anything else the project may need.

If you're setting up a new project on GitHub, follow these instructions. It is recommended to include a README or some other file with information about the project, this can be done at the same time the new repository is created.

1. **Go to https://github.com and sign in**
2. **Create the repo** - in the upper right corner, next to your avatar, click the `+` button and select "*New repository*".

3. **Fill in the details** - select the owner, give it a name (short, specfic, memorable, and preferably lowercase-with-hyphens), write a short description, and consider if initialising with a README is appropriate. *A README file is used to provide up front information about what the project does, why it's useful, how users can interact, where to get help, and who maintains and contributes to the project. This is really useful for anyone landing on the GitHub page to find out more. Initialising a repository on GitHub with a README will simply add a markdown file to which you can add the details to later.*
4. **Click "Create repository"**

5. **Link local project to GitHub** - if you're not using a GUI (Graphical User Interface) such as RStudio which has Git and GitHub integrations (see Quick Start RStudio), follow along using command line tools, introduced above in Quick Start Git. In order to commit to Git and then link to GitHub, some file needs to exist. In this example, a README.md file is created. Then, we add the link to the GitHub repo (you only need to do this once per project) with `git remote add <name> <url>` (*`<name>` is any name to refer to the GitHub connection but `origin` is the most common/preferred name to use*) and then sending the first set of changes to GitHub using `git push <name> <branch>`. *Also, this example uses the HTTPS connection (a URL) to GitHub. Your URL will be unique to your project. Check the GitHub code page for the instructions.*

```
username@NSS000000 MINGW64 ~/Documents/demo-project
$ echo "# demo-project" >> README.md
$ git init
Initialized empty Git repository in C:/Users/username/Documents/demo-project/.git/

username@NSS000000 MINGW64 ~/Documents/demo-project
$ git add README.md
$ git commit -m "first commit"
[master (root-commit) 1ab2cde] first commit
  1 file changed, 1 insertion(+)
  create mode 100644 README.md

username@NSS000000 MINGW64 ~/Documents/demo-project
$ git remote add origin <url>
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 220 bytes | 220.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To <url>
* [new branch]    master -> master
Branch master set up to track remote branch master from origin.

username@NSS000000 MINGW64 ~/Documents/demo-project
$
```

Existing Project Repository

1. **Download the repository from GitHub** - get the URL for the project you are looking to contribute to and using `git clone <url>` make a local copy on your machine. This will create a local folder with the same name as the repo and a copy of all the files inside. In order to continue version control on that folder/repo, move into the folder on the command line using `cd <folder-name>`.

```
username@NSS000000 MINGW64 ~/Documents
$ git clone <url>
Cloning into 'existing-project'...
remote: Enumerating objects: 7, done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
Unpacking objects: 100% (7/7), done.

username@NSS000000 MINGW64 ~/Documents
$ cd exising-project

username@NSS000000 MINGW64 ~/Documents/existing-project
$
```

2. **Set up a branch, make some changes, stage them, commit them, and push them back to GitHub**

```
username@NSS000000 MINGW64 ~/Documents/existing-project
$ git checkout -b 'feature'
Switched to branch 'feature'

username@NSS000000 MINGW64 ~/Documents/existing-project
```

```
$ git add new-file.md

username@NSS000000 MINGW64 ~/Documents/existing-project
$ git commit -m "add new-file"

username@NSS000000 MINGW64 ~/Documents/existing-project
$ git push -u origin feature
Counting objects: 1, done.
Writing objects: 100% (1/1), 10 bytes | 10.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To <url>
* [new branch]    feature -> feature
Branch feature set up to track remote branch feature from origin.

username@NSS000000 MINGW64 ~/Documents/existing-project
$
```

3. **Open a pull request** - when you've made changes that are ready to be included into production, open a pull request to propose and collaborate. These changes are proposed in that branch, which ensures that the master branch only contains finished and approved work.

    1. On GitHub, go to the main page of the repository.
    2. In the "Branch:" menu, choose the branch that contains *your* commits.
    3. To the right of the "Branch" menu, click "New pull request".
    4. Use the "base:" branch dropdown menu to select the branch you'd like to merge your changes into.
    5. Use the "compare:" branch dropdown menu to choose the topic branch your changes are in.
    6. Type a title as a description for your pull request.
    7. If your pull request is ready for review, click "Create Pull Request". *To create a draft pull request, use the dropdown menu and select "Create Draft Pull Request", then click "Draft Pull Request".*

Git

Git

Git is a version control system (VCS). We recommend that Git is used along with a hosting repository, such as GitHub (or Gitea, GitLab, etc.) and that the GitHub Workflow is followed. However, Git can be used locally while working on a project. The steps outlined in Quick Start Git are for this purpose and should only be followed for a personal project that is **local or on your personal network folder**. For collaborative work, see Quick Start GitHub.

Quick Start Git

1. **Navigate to the directory** - navigate to the project folder on the shell command line. From here, you can use Git and shell/bash commands. *The line prompt for the user to enter commands is* ***$****, other lines are generated by the Shell.*

```
username@NSS000000 MINGW64 ~
$ cd Documents/project

username@NSS000000 MINGW64 ~Documents/project
$
```

You can also navigate the project folder using Windows Explorer, right click, and select "Git Bash Here", this will open a command-line interface for that folder.

2. **Initialise Git** - in the command line enter `git init`. Git is now initialised inside that project folder/directory and can track any files or sub-folders.

```
username@NSS000000 MINGW64 ~/Documents/project
$ git init
Initialized empty Git repository in C:/Users/username/Documents/project/.git/

username@NSS000000 MINGW64 ~/Documents/project (master)
$
```

You'll notice that you're on the "*master*" branch. For set-up purposes, we'll continue to work on the *master* branch but note that it's best practice to do any work in a separate branch which is later merged into the *master* when ready.

3. **Check for changes** - Git will recongise any files that have been added or changed. Enter `git status` to see an overview.

```
username@NSS000000 MINGW64 ~/Documents/project (master)
$ git status
On branch master

No commits yet
```

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

      .gitignore
      project.Rproj

nothing added to commit but untracked files present (use "git add" to track)

username@NSS000000 MINGW64 ~/Documents/project (master)
$
```

In this example, an R Project has been started with a .gitignore file. *A .gitignore file is used to tell Git which files and folders to ignore, this is particularly imporant when working with GitHub and ensures that no sensitive information or data files are uploaded to GitHub. An example .gitignore file can be found here. Be aware that other file formats may also need to be considered to be added in this file.*

4. **Track files** - in this example Git has recognised two untracked files. In order to track them we must first *stage* them using `git add <file name>` (if the file name contains spaces you will need to put single quotation marks around it, e.g. `'file name'`) You can stage each file individually or use `git add .` to stage all files which have been detected by Git. **It is usually safer to stage each file separately.**

```
username@NSS000000 MINGW64 ~/Documents/project (master)
$ git add .gitignore

username@NSS000000 MINGW64 ~/Documents/project (master)
$ git add project.Rproj

username@NSS000000 MINGW64 ~/Documents/project (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   .gitignore
      new file:   project.Rproj


username@NSS000000 MINGW64 ~/Documents/project (master)
$
```

5. **Commit changes** - now, a set of staged changes can be committed using `git commit -m <commit message>`. When entering you commit message, ensure that it's consise, meaningful and written in imperative mode.

```
username@NSS000000 MINGW64 ~/Documents/project (master)
$ git commit -m "Create R project"
[master (root-commit) 1ab2cde] Create R project
  2 files changed, 17 insertions(+)
```

```
  create mode 100644 .gitignore
  create mode 100644 project.Rproj

username@NSS000000 MINGW64 ~/Documents/project (master)
$
```

Git has now stored a snapshot of the project folder and its content at that point in time. Going forward, it will now be possible to check back through the old version of the folder via the commits and, if necessary, revert to a previous version. To see a history of ocmmits on a branch use `git log`.

6. **Create a branch** - when first created, a branch is an exact copy of the original folder and contents (the master branch). As you work on the project, the working branch will change but leave the master branch untouched. This means that you always retain a master copy of the project and you only merge changes when you're satisfied that they're ready. To create a branch use `git branch <name of branch>`. To switch to working on the new working branch, use `git checkout <name of branch>`. These two steps can be done via one command, `git checkout -b <name of branch>`.

```
username@NSS000000 MINGW64 ~/Documents/project (master)
$ git checkout -b feature
Switched to branch 'feature'

username@NSS000000 MINGW64 ~/Documents/project (feature)
$
```

7. **Check for changes to branch** - like in step 3, you'll do work in your project and have changes to commit, use `git status` as before to see an overview. Then, keep saving and committing your work using `git add <file>` and `git commit -m <commit message>`.

```
username@NSS000000 MINGW64 ~/Documents/project (feature)
$ git status
On branch feature
Changes not staged for commit:
  (use "git add <file>..." to include in what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   script.R

no changes added to commit (use "git add" and/or "git commit -a")

username@NSS000000 MINGW64 ~/Documents/project (feature)
$ git add script.R

username@NSS000000 MINGW64 ~/Documents/project (feature)
$ git commit -m "Set up new script"
[feature a1bc23d] Set up new script
  1 file changed, 1 insertion(+)
```

8. **Check for branch differences** - when you want to merge the changes into the master branch, you'll want to compare them first. You can see the changes made by using `git diff <master branch> <working branch>`. On the command line, insertions have plus (+) signs at the start of the line while any deletions will have a negative (-) sign.

```
username@NSS000000 MINGW64 ~/Documents/project (feature)
$ git checkout master
Switched to branch 'master'

username@NSS000000 MINGW64 ~/Documents/project (master)
$ git diff master feature
diff --git a/script.R b/script.R
new file modile 100644
index 0000000..fe32d10
--- /dev/null
+++ b/script.R
@@ -0,0 +1,1 @@
+ #This is a demo R script with no content
\ No newline at end of file

username@NSS000000 MINGW64 ~/Documents/project (master)
$
```

9. **Merge changes into master branch** - when you're ready to make the merge into the master branch, ensure you're on the master branch and use `git merge <working branch>`.

```
username@NSS000000 MINGW64 ~/Documents/project (master)
$ git merge feature
Updating 111abc0..f999ed0
Fast-forward
  script.R | 1 +
  1 file changed, 1 insertion (+)
  create mode 100644 script.R

username@NSS000000 MINGW64 ~/Documents/project (master)
$
```

10. **Delete working branch** - when you're finished with your working branch delete it using `git branch -d <branch name>`, this will help to avoid merge conflicts. There is no need to have long-living branches; when you want to make further changes to your project, simply create a new working branch.

Git Cheat Sheet

Git Basics

Command	Description
‘git config –global user.name ”[name]”‘	Sets author name to be used for all your commits.
‘git config –global user.email ”[email]”‘	Sets author email to be used for all your commits.
‘git init [project-name]‘	Create empty git repo in specified directory.
‘git clone [repo]‘	Clone repo with version history located at ‘[repo]‘.
‘git status‘	List all new or modified files to be committed.
‘git diff‘	Show file changes not yet staged.
‘git add [file]‘	Stage all changes in ‘[file]‘ for the next commit.
‘git diff –staged‘	Shows files differences between staging and the last file version.
‘git reset [file]‘	Unstages the file, but preserves its contents.
‘git commit -m “[message]”‘	Commit the staged snapshot to permanent version history.
‘git log‘	Display the entire commit history for the current branch.

Undoing changes

Command	Description
‘git revert [commit]‘	Create new commit that undoes all of the changes made in ‘[commit]‘, then apply it to the current branch.
‘git reset [file]‘	Remove ‘[file]‘ from the staging area, but leave the working directory unchanged. This un-stages a file without overwriting any changes.
‘git clean -n‘	Shows which files would be removed from working directory. Use the ‘-f‘ flag in place of the ‘-n‘ flag to execute the clean.

Rewriting Git History

Command	Description
‘git commit —amend‘	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit’s message.
‘git rebase [base]‘	Rebase the current branch onto ‘[base]‘. ‘[base]‘ can be a commit ID, a branch name, a tag, or a relative reference to ‘HEAD‘.
‘git reflog‘	Show a log of changes to the local repository’s ‘HEAD‘. Add ‘–relative-date‘ flag to show date info or ‘–all‘ to show all refs.

Git Branches

Command	Description
‘git branch‘	List all of the branches in your repo. Add a ‘branch‘ argument to create a new branch with the name ‘branch‘.
‘git checkout -b [branch]‘	Create and check out a new branch named ‘[branch]‘. Drop the ‘-b‘ flag to checkout an existing branch.
‘git merge [branch]‘	Merge ‘[branch]‘ into the current branch.

Remote Repositories

Command	Description
‘git remote add [name] [url]‘	Create a new connection to a remote repo.
‘git fetch [remote]‘	Downloads all history from the remote repo.
‘git merge [remote]/[branch]‘	Combines the remote branch into the current local branch
‘git pull [remote]‘	Downloads and merges remote’s copy of current branch.
‘git push [remote] [branch]‘	Uploads the branch and history to ‘[remote]‘.

Git Log

Command	Description
‘git log -[limit]‘	Limit number of commits by ‘[limit]‘. E.g. ‘git log -5‘ will limit to 5 commits.
‘git log –oneline‘	Condense each commit to a single line.
‘git log -p‘	Display the full diff of each commit.
‘git log —-stat‘	Include which files were altered and the relative number of lines that were added or deleted from each of them.
‘git log —-author= ”[pattern]”‘	Search for commits by a particular author.
‘git log —grep=”[pattern]”‘	Search for commits with a commit message that matches [pattern].
‘git log [since]..[until]‘	Shows commits that occur between ‘[since]‘ and ‘[until]‘. Args can be a commit ID, branch name, ‘HEAD‘, or any other kind of revision reference.
‘git log – [file]‘	Only display commits that have the specified file.
‘git log –graph –decorate‘	‘–graph‘ flag draws a text based graph of commits on left side of commit msgs. ‘–decorate‘ adds names of branches or tags of commits shown.

Git Diff

Command	Description
‘git diff HEAD‘	Show difference between working directory and last commit.
‘git diff –cached‘	Show difference between staged changes and last commit.

Git Reset

Command	Description
‘git reset‘	Reset staging area to match most recent commit, but leave the working directory unchanged.
‘git reset –hard‘	Reset staging area and working directory to match most recent commit and *overwrites all changes* in the working directory.
‘git reset [commit]‘	Move the current branch tip backward to ‘[commit]‘, reset the staging area to match, but leave the working directory alone.
‘git reset –hard [commit]‘	Same as previous, but resets both the staging area & working directory to match. Deletes uncommitted changes, and all commits after ‘[commit]‘.

Reference

Top Tips

These are some tips/references to keep in mind when working with git:

- Although it's good to use version control throughout a project, git can be initialised on an exisitng project folder.
- Commit often. Commits are the project history so commit after a section of code is completed, before lunch, at the end of the day, etc.
- Write good commit messages. The messages should be succinct, meaningful, and written in imperative form (i.e. "Add x, y, z" not "Added x, y, z").
- Delete branches after merging. This reduces risks of merge conflicts and keeps your work set-up tidy.
- It's possible to have multiple branches for working on or fixing multiple features of your project, but be aware that this increases the chance of a merge conflict.

Git

Pro Git Book - A complete reference text for Git.

Atlassian - Git - Tutorials, tips, and the latest news about git.

Try Git - Learn by doing. This will allow you to work with git commands on a secure tutorial system.

Software Carpentry - Git Lesson - *"Version control is the lab notebook of the digital world: it's what professionals use to keep track of what they've done and to collaborate with other people. Every large software development project relies on it, and most programmers use it for their small jobs as well. And it isn't just for software: books, papers, small data sets, and anything that changes over time or needs to be shared can and should be stored in a version control system."*

Department for Education - VSTS for Analysis - *"This book aims act as a resource for analysts on how and why they should use version control."*

GitHub

GitHub Guides - A variety of guides available directly from GitHub.

Udacity - How to Use Git and GitHub - *"Effective use of version control is an important and useful skill for any developer working on long-lived (or even medium-lived) projects, especially if more than one developer is involved. This course, built with input from GitHub, will introduce the basics of using version control by focusing on a particular version control system called Git and a collaboration platform called GitHub."*

Happy Git and GitHub for the useR - *"Integrate Git and GitHub into your daily work with R and R Markdown."*

RStudio

RStudio - Using Projects - A support article for using RStudio projects. There is a need to use projects for version control so this article provides an introduction to this topic.

RStudio - Version Control with Git and SVN - A support article for using version control in RStudio.

R Package Pull Requests: A Newbie's Guide - A blog post that discusses the experience of submitting pull requests for existing R packages.