Introduction to Git for Bioinformaticians

Jakob Willforss

May 30, 2016
v1.1

Contents

Chapter 1

Introduction to version control

1.1 Acknowledgements

This material has been written with support from PlantLink, a research network for plant science in southern Sweden.

Many thanks to Ellen Sunström for providing extensive proof-reading.

1.2 Welcome

Welcome to this introduction to version control and the version control software "Git" in particular. This material is geared towards bioinformaticians and puts an emphasis on parts of Git that are particularly useful when doing bioinformatics, but it also aims to give a general understanding of version control, how to use Git and a bit of insight into how Git works internally.

1.3 The structure of this document

Each chapter is divided into three parts:

- The reading part - Where the concepts are introduced together with demonstrations of the commands.
- The exercise part - Where the concepts are applied.
- The recap part - Where you can check and reinforce your understanding of the introduced concepts and commands.

In the text, you will encounter the following types of colored boxes:

Command line examples are written within gray boxes.

```
$ Lines starting with "$" are commands run in the terminal
# Lines starting with "#" are comments
Other lines are output received when running the commands
```

Note

Important notes and comments about potential pitfalls are shown in red boxes.

`git add [-A] <path>`

Git commands are introduced in blue boxes. Optional arguments are shown within square brackets. Input arguments (strings, paths) are shown in angular brackets.

In the example above, the `[-A]` indicates that the command can take an `-A` flag which doesn't require an input argument. If the flag would have taken an argument, it would have been shown as: `[-A <argument>]`. The command requires an input path.

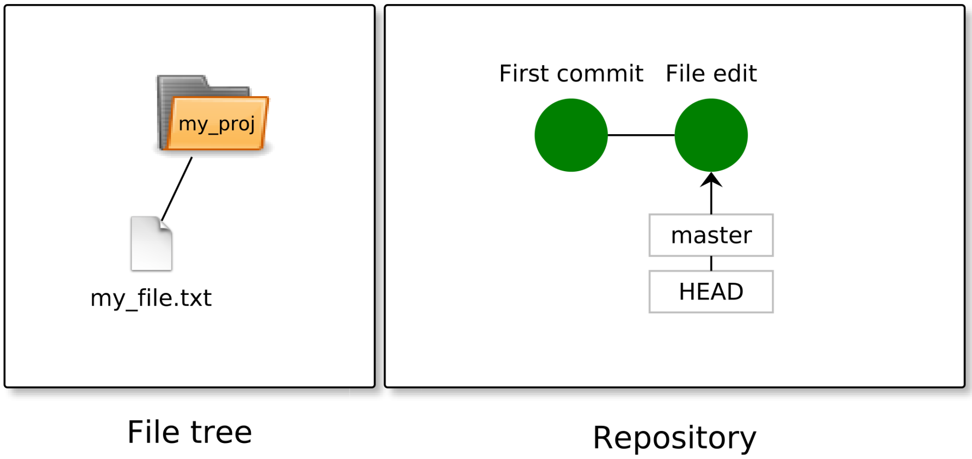


Figure 1.1: Example visualization of the file system and the repository

1.3.1 Visualizing Git

This material makes frequent use of the kind of visualization shown in Figure 1.1. In this case it shows both the file tree (left), and the state of the `repository` which stores the history of the files (right). This will be explained in more detail in the next chapter.

We will also sometimes visualize a third component of Git - the `stage` (also called the "index"). The stage is used as a gathering-place for file edits which are to be stored in the history. An example of this is shown below in Figure 1.2. This will also be further explained in the following chapter.

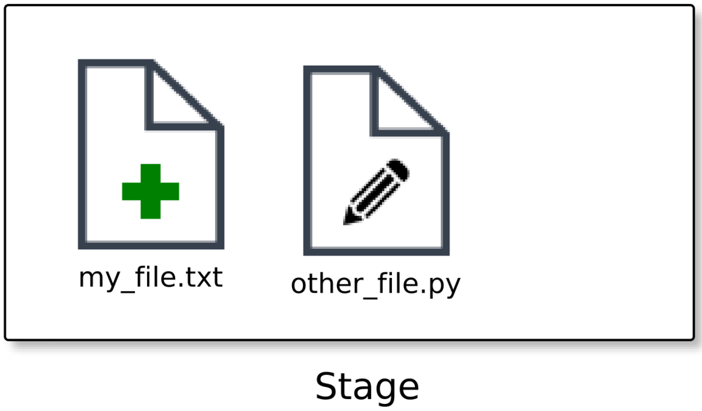


Figure 1.2: Example visualization of the stage

1.4 What is version control?

```
"Version control is a system that records changes to a file or set of files
 over time so that you can recall specific versions later"
- Pro Git (book)
```

A version control system keeps track of all the changes that you make within a designated directory, called the `repository`. It allows you to recall previous versions of your files at a later time-point.

Repositories hosted in a version control system often have a central repository hosted on a server computer, called the `remote`. This repository can be accessed from any computer with the right access rights. GitHub is a popular service providing remote hosting. More on this in Chapter 5.

The version control system provides a structured way of saving states of the file tree in the form of `commits`. A commit contains a snapshot of the repository in a particular state.

There are many different version control systems. Here, we will be using Git. Git was initially conceived by Linus Torvald, the creator of the Linux kernel, and was developed by the Linux kernel developers. Git is currently (May, 2016) one of the most widely used version control systems. Other popular version control systems include Subversion (SVN) and Mercurial.

1.5 Why use version control?

A version control system can serve many different purposes. In this material, it will be presented from the perspective of how it can be useful to a bioinformatician.

Reproducible research It is not uncommon for bioinformaticians to have many copies of a particular script laying around in different locations, often in different versions. In order to be able to reproduce the analysis at a later stage, it is crucial that you later are able to re-run the exact same version of the code. Git helps you with this as you:

1. Can refer to the exact state of the code used. Git provides the ability to tag commits to easily keep track of important versions.
2. Have a central place for your code - You know exactly where to look for the different versions.

Easy back-tracking of errors When you introduce bugs in your code, or accidentally remove an important paragraph, you can always back-track the changes made since the last working version.

Clear history of tasks By looking at the history of commit messages, you get a clear overview of of the changes that have been made in the project at which times. This also allows for the pinpointing of changes made at particular points in time.

Essential for code teamwork If multiple people are working on the same source code, a version control system is a must. It provides a structured way of implementing changes from different users, even when the changes are done on the same set of files.

Code clarity As all your previously saved code is stored within the history of the version control system, there is no longer any need to keep unused code which 'might come to use later'.

1.5.1 Downsides to Git

Git is useful in many ways, but as with all tools, the benefits come at a cost. Use your judgement to decide on when the benefits outweight the costs.

Learning curve It is a new system to learn and understand. The basic commands are easy to pick up, but it takes some effort to be able to use Git fluently outside the basic commands. Hopefully this material can help you with that!

Another layer of complexity It introduces another step into your workflow. This usually means running a few more commands per working-session, but can be more complex when running into problems.

Geared for raw text Version control systems like Git are designed to work with raw text. Git is able to keep track of binary files, but unable to track exact changes - it stores the full versions of the binary. It is mostly not a good idea to track binary files in Git.

1.6 Why understand Git?

One goal with this material is to give you some insight into how Git structures things internally and what happens when you run the different commands. Often, one can simply run the commands without understanding the inner workings of Git, but some knowledge of the inner workings makes it much smoother to debug errors and perform more complex tasks.

But, there is a lot going on inside the Git system which is outside the scope of a one-day course. If you have the time and the interest of digging deeper, you can take a look at the book *Git from the bottom up* by John Wiegley. A web version of the book can be found at the following link:

`jwiegley.github.io/git-from-the-bottom-up`

1.7 Exercises

The exercises in this material gives you the opportunity to get hands-on practice with the introduced concepts. They closely follow the material. If you get stuck, you can usually find the answer in the text of the chapter.

Some of the exercises are marked with asterisks (*). Those are more challenging and/or time consuming exercises. If you are short on time, skip those on the first run-through. If you have the time - Do them. They will help you dig deeper into the introduced concepts.

1.7.1 How to get help

1. Make sure that you have a working version of Git running on your computer. Running the command `git` by itself should result in a number of commonly used commands.

   ```
   $ git
   ```

   We will go through a number of those commands. If you instead got an error message telling you that the command wasn't found, Git is likely not properly installed on your computer.

2. To find out more about a Git command, type out the command (example: `git add`) followed by the `-h` flag. If you want to get information about the `git add` command, you could type the following:

   ```
   $ git add -h
   ```

   To open the manual pages for a command, use the `--help` flag. This opens the manual page in `less`. (You exit `less` by pressing `q`).

   ```
   $ git add --help
   ```

1.7.2 Get used to finding help (*)

1. Take a look into the help-pages for the following commands:

   - `git init`
   - `git status`
   - `git add`
   - `git commit`

   We will work with these commands in the coming chapter.

1.8 Recap

Every chapter ends with a recap of the introduced commands and concepts. Make sure that you have a good grasp of both the meaning of the commands and the different concepts.

1.8.1 Concepts

- How can Git be useful for you as a bioinformatician?
- What are the potential troubles and limitations of Git?

1.8.2 Commands

- `git <command> -h`

  Print brief information about a Git command to the terminal.

- `git <command> --help`

  Open the manual page for the command in `less`.

Chapter 2

Understanding and using a repository

2.1 Overview

In this chapter, we will:

- Do a quick demonstration of how to set up a repository and add files to its history
- Introduce and explain important core concepts in Git
- Go through the demonstrated commands at a slower pace

2.2 Crash and burn

As a first taste of Git, you will be shown a commonly used sequence of commands that:

- Creates a new Git repository
- Creates some files
- Prepares edited files to be included in the next commit by adding them to the stage
- Creates a new commit including the staged changes

After running through the commands, we will back up and go through the process step by step, explaining what is going on.

2.2.1 Running the commands

To create a repository within a directory, we use the command `git init`. This initializes a Git repo (repository) within the current working directory.

```
$ mkdir MyRepo
$ cd MyRepo
$ git init
Initialized empty Git repository in /home/user/MyRepo/.git/
```

Note

It is recommended that you run the commands alongside the material and look for yourself how each command effects your Git repository.

You will not always be able trace the state of the repository (and the output) exactly as the text sometimes makes jumps back time to make it possible to illustrate the repository, but you should be able to run the commands on your own repository and create your own history.

Next, we add two files to the file tree. Let's create a README-file and a ”Hello world” script.

```
$ echo 'README - Course repository' > README.md
$ echo '#!/bin/bash' > hello_world.sh
$ echo 'echo "Hello world"' >> hello_world.sh
```

Next, we decide on which changes to include in the next commit - the next version of the file tree which will be frozen in the history of the repository. In this case, we include both the README and the Hello world-script in the commit.

```
$ git add README.md
$ git add hello_world.sh
```

Finally, we are ready to create a commit including the newly created files.

```
$ git commit -m "Creates README and Hello World script"
[master (root-commit) 086acdf] Creates README and Hello World
script
 2 files changed, 3 insertions(+)
 create mode 100644 README.md
 create mode 100644 hello_world.sh
```

Done! Now we have started a repository in our working directory, created a file, and added that file to our repository. The current state of the files is saved in the repository.

The next step here is usually to sync the changes with a remote repository. This will be introduced in Chapter 5.

2.3 Backing up - Terms and concepts

Now, let's back up a bit. What is actually going on in the repository? Here, we will present some core terms and concepts which makes Git considerably easier to understand. It is worth putting in some extra reading-effort here, as the presented concepts helps understanding the commands we will be using throughout this material.

We will go through:

- What a *commit* represents, and which information it contains
- How files transition between the *file tree*, the *stage* and the *repository*, and the relationship between them
- What *heads* are, and how the `HEAD` head is special

2.3.1 Vocabulary

To understand a concept you need to know its vocabulary. You will frequently encounter the words presented here in the rest of this material. The concepts are explained in more depth below.

file tree A set of files and directories originating in a particular directory. The subdirectories and files of a directory is its file tree. This is present independently of the version control system.

commit A snapshot of a particular state of the *file tree*. A commit contains an ID, the state of the file tree and a reference to its parent commit among with some other information.

repository A repository is attached to a *file tree*. Commits made for its *file tree* are saved in the *repository*.

staging area / index Location used to choose changes that were made to the *file tree* and that should be included in the next *commit*.

branch The *commits* of a *repository* can be branched, with multiple features being developed in parallel. This is called *branching* and is a powerful part of Git which will be revisited in Chapter 4.

heads References to particular *commits*. `HEAD` is a *head* referring to the commit currently represented in the *file tree*. `master` is a *head* commonly referring to the latest commit on the main development branch.

2.3.2 A commit

A commit takes a snapshot of a particular state of the file tree (which files are present and what their current content is). The commit is actually storing the changes made compared to its parent commit. This can be used together with all parent commits to retrieve the file structure that is represented in this particular snapshot.

A commit contains the following information (this is visualized in Figure 2.2):

- A unique ID, which can be used to refer to the commit
- Information about the commit, which includes the snapshot of the file tree and the descriptive message provided together with the commit

- A reference to the parent commit

The ID is an SHA-1 hash based on the content of the commit. It is represented as a 40 character long string. An example is shown in Figure 2.1. Commits are often referred to using only the first 6-7 letters/digits of their IDs.

```
74a3f1511429d8b688aad299485546b4cc86cb75
```

Figure 2.1: An SHA-1 hash commit ID

The parent commit represents the state of the repository before making the changes in the current commit.

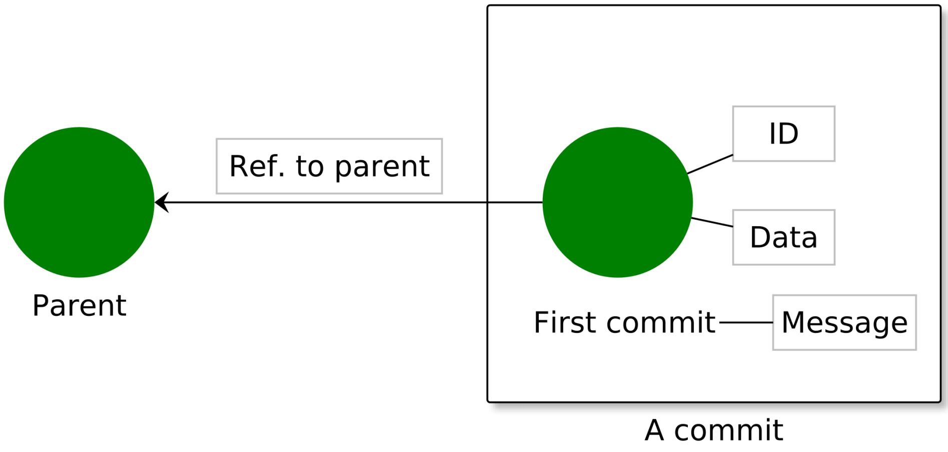


Figure 2.2: A commit consists of three parts. Its ID, a reference to its parent, and its data.

2.3.3 The repository, the file tree and the stage / index

When working with Git, you constantly interact with the file tree, the stage and the repository.

You are used to the *file tree*. It is simply the structure of files and directories which you use everyday when not using Git.

The *stage* is a place to which you add the changes you want to include in your next commit. You don't need to include all the changes made to the file tree in a particular commit. It is often easier to navigate the history if you include changes related to a particular feature in one commit.

The repository is the system keeping track of all the commits. Every time you make a new commit (based on the staged file-changes), it is added to the repository.

2.3.4 The concept of *heads*

Heads are used in Git to refer to particular commits. They are used to keep track of the leading commits of branches, and for keeping track of which commit that is currently

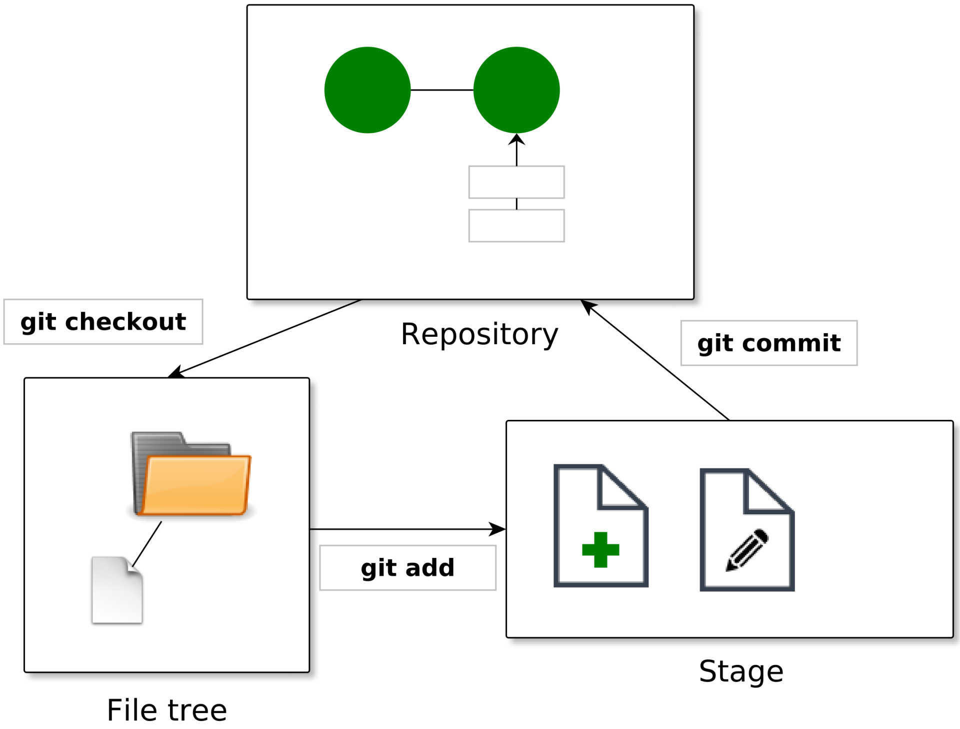


Figure 2.3: The three major entities in the Git system

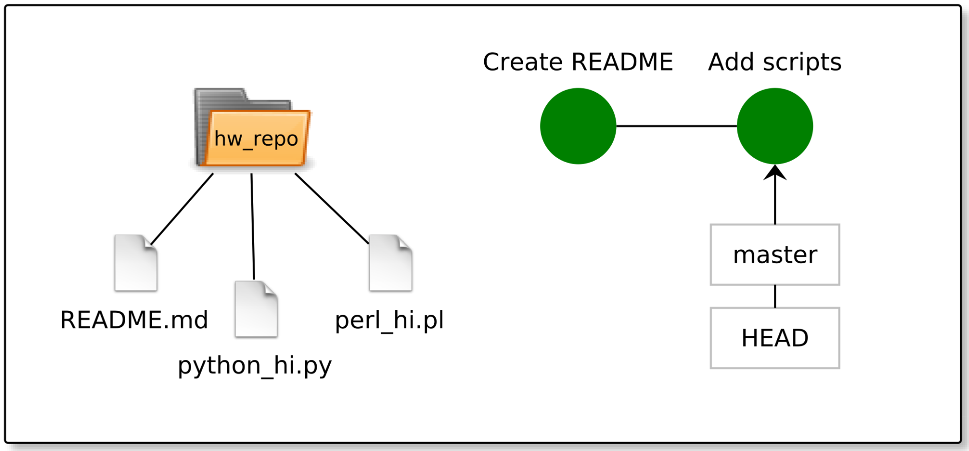


Figure 2.4: A file tree and repository with the heads "HEAD" and "master"

represented in the file tree (the `HEAD`). In Figure 2.4 the heads `HEAD` and `master` are both pointing to the second commit.

When creating an empty repository, you always get two heads: `master` and `HEAD`. When we later on make branches for our repository, we will create one new head per branch.

master Generally the central development branch, where we insert changes that we want to include in the official versions of the code / text.

HEAD Special head pointing to the currently active commit. We can investigate the file tree of different commits by moving the `HEAD` to the corresponding commit. This is further investigated in Chapter 3.

We will later on move `HEAD` to different commits, which changes the file tree to the one represented by that commit.

2.3.5 Concept-recap

Before leaving this section, make sure that you have an idea of the following concepts:

- What a commit represents, and which information it contains.
- The file tree, the stage and the repository, and how they are related.
- What heads are, and how the `HEAD` head is special.

2.4 Creating a new repository, slowly...

2.4.1 Creating the repository - First steps

As a first step, let's create a directory called `hw_repo` (for ”Hello World repository”) which we will use for our project, and change our working directory to that directory (use a more descriptive name when not constrained by the page width of your teaching material). At this point, our folder contains an empty file tree, and no repository or stage.

> **Note**
>
> The initial steps here are the same as the ones presented at the start of this chapter. If you ran them before - Choose yourself if you want to repeat them or not.

```
$ mkdir hw_repo
$ cd hw_repo
```

Next, we initialize the repository within the folder. This will create a (hidden) folder named `.git` containing the repository.

```
$ git init
Initialized empty Git repository in /home/user/hw_repo/.git/
```

At this point the directory only contains the hidden `.git`-folder containing the repository files. This is shown in Figure 2.6.

Command: `git init`

Initializes a new repository within the current file tree. The repository is related to the current working directory and its subdirectories.

Figure 2.5: git init - Initialize a new repository

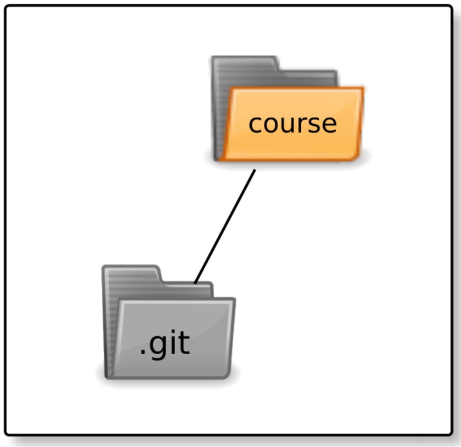


Figure 2.6: The file system of a newly initialized repository. The gray folder is hidden, and will not be shown in the rest of the figures.

2.5 Saving changes in a commit

Now, let’s create a README file, add it to the stage and commit it to the repository. To do this we’ll need to introduce two new commands: `git add` and `git commit`.

First, we create a new file within the file tree. We use the `echo` command to create the readme `README.md` containing the line `’A Hello World collection’`. The `README.md` file is used on services like GitHub to display a README for the repository on the site.

```
$ echo ’A Hello World collection’ > README.md
```

We can at any time see the status of the repository using the command `git status`. This gives us (among other things) information about changes made to the file tree and which files that are staged and ready to be committed.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will
  be committed)
    README.md
nothing added to commit but untracked files present
(use "git add" to track)
```

This tells us that:

1. The current branch is `master`
2. It is the first commit of the repository (”Initial commit”)
3. We have one new file in the file tree which isn’t tracked by the repository - `README.md`
4. There are currently no staged files

Let’s add `README.md` to the stage using `git add`, and check the status again.

```
$ git add README.md
$ git status
On branch master

Initial commit

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

    new file:   README.md
```

Now, the status looks slightly different. We don’t have any untracked files in the file tree. We have one file (`README.md`) ready to be committed. The staged changes ready to be committed are visualized in Figure 2.7.

When doing a commit, we must always supply a commit message describing what the purpose of the changes made in the commit is. We can supply the commit description using the `-m` flag. When running `git commit` without the `-m` flag, a default text editor will automatically be opened.

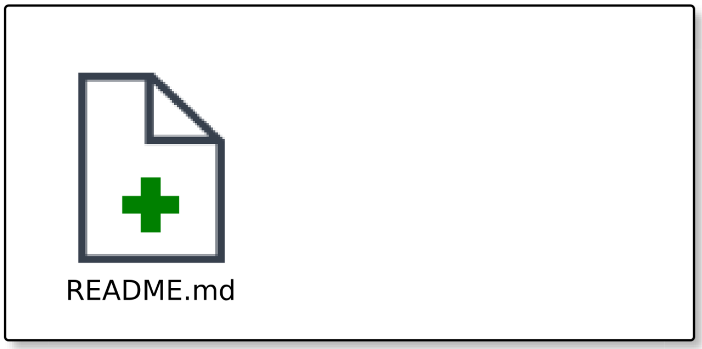


Figure 2.7: Stage after adding the newly created README.md

```
$ git commit -m "Create README"
[master (root-commit) b58c6c3] Create README
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ git status
On branch master
nothing to commit, working directory clean
```

We have our first commit! Also, there are no untracked changes in the file system and no staged changes. The current state of this file tree / repository is visualized in Figure 2.8. At this point, we have initialized two heads: `master` and `HEAD`. They both point to our latest and only commit.

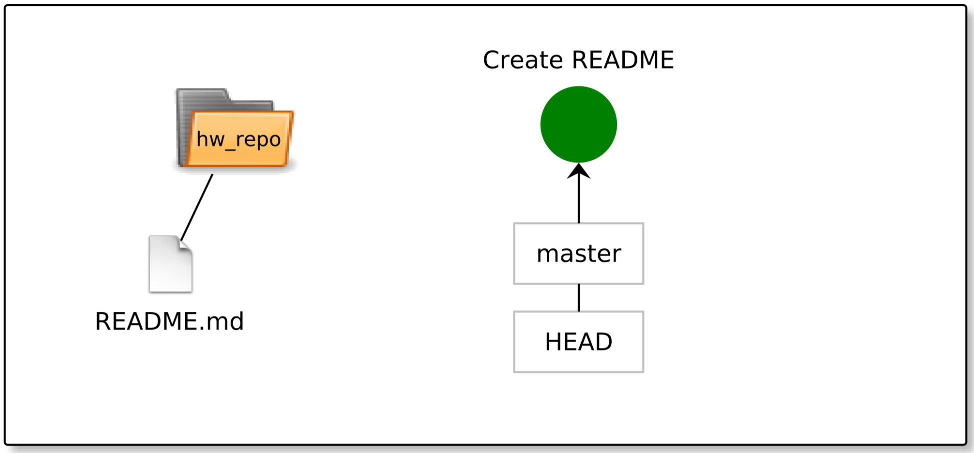


Figure 2.8: The state of the file system (left) and the repository (right) after the first commit

Note

It is considered good practice to write commit messages in *present tense*, ”Create README” rather than ”Created README” or ”Creates README”.
Although, the most important practice when it comes to commit messages is to write clear and descriptive messages. The commit messages used in this material are in many cases *not* good examples of descriptive messages - They are kept shorter than usual to fit into code boxes and figures.

Command: `git add [-A] <path>`

Stages changes made to the file tree. The `-A` flag can be used to stage files removed from the repository. If used *with* the `-A`, moved files and removed files can also be staged. It can be used to stage specific files, or directories. When staging directories, all sub-files are included.

Figure 2.9: git add - Stage changes made to the file tree

Command: `git commit [-m <message>]`

Commits a set of changes to the history, taking a snapshot of the current state of the repository. It is required to supply a commit message together with the commit. This commit message should describe the meaning of the commit. It can either be supplied using the `-m` flag or in the text editor which automatically opens when the command is run without the flag.

Figure 2.10: git commit - Create a snapshot implementing changes added to the stage

Command: `git status`

Prints the current status of the repository. Gives information about changed files and which files that are staged and ready to be committed. When working with remotes, also provides information about if the local repository is synced with the remote.

Figure 2.11: git commit - Create a snapshot implementing changes added to the stage

2.6 Committing changes and checking status

Up until now we have only added files to our file-tree. We can also include file edits in commits. This section will demonstrate how the stage can be useful to pick a set of changes to include into a particular commit.

We make three edits to the file tree:

- We add another line to the README
- We create a Python Hello world script
- We create a Perl Hello world script

```
$ echo ’Hello World in different languages’ >> README.md

# Let’s create hello world in Python
$ echo ’#!/usr/bin/python3’ > python_hi.py
$ echo ’print("Hello world")’ >> python_hi.py

# Let’s create hello world in Perl
$ echo ’#!/usr/bin/perl’ > perl_hi.pl
$ echo ’print("Hello world");’ >> perl_hi.pl
```

Next, we check the status for the repository.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)

modified:   README.md

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

perl_hi.pl
python_hi.py

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

At this point we have a number of unstaged changes. The repository sees that `README.md` has been modified. It also recognizes `perl_hi.pl` and `python_hi.py` as two new untracked files.

To keep the history of the commits nice and tidy, let’s commit the ’Hello world’ scripts separately from the edit made to `README.md`. Then, we can decide on what to do with the changes made to `README.md`. To do this, we stage only the Hello World-scripts.

```
$ git add perl_hi.pl
$ git add python_hi.py
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   perl_hi.pl
    new file:   python_hi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)

    modified:   README.md
```

Git tells us that the two Hello world scripts are ready to be committed. The staged changes are visualized in Figure 2.12. We are now ready to commit the addition of the two "Hello world"-scripts. The edit of the `README.md` files is not staged, and will not be included in this commit.

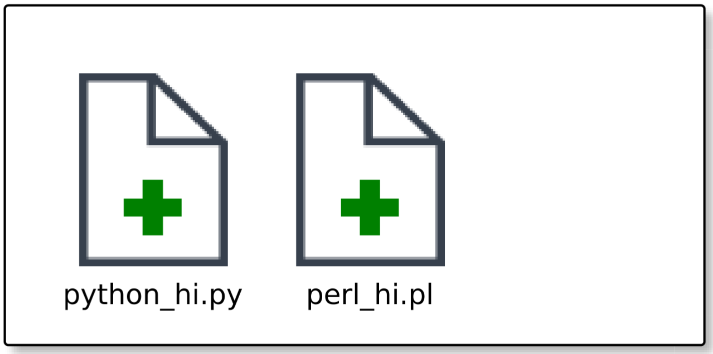


Figure 2.12: Stage after adding two newly created hello-world scripts

```
$ git commit -m "Add scripts"
[master 7100d36] Add scripts
 2 files changed, 4 insertions(+)
 create mode 100644 perl_hi.pl
 create mode 100644 python_hi.py
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)

    modified:   README.md

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

The file tree and repository after doing this second commit is shown in Figure 2.13. We are committing the the `master` head, which now points to our latest commit. The `HEAD` head has also switched to the latest commit.

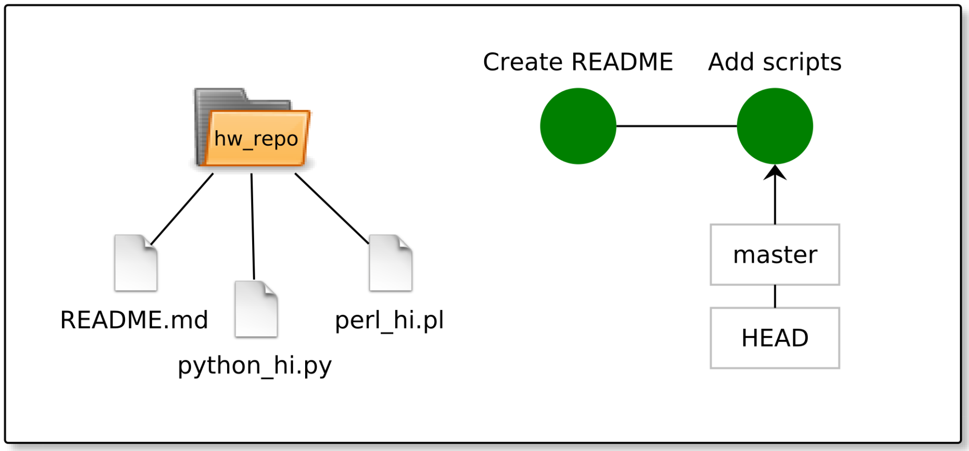


Figure 2.13: The state of the file system and repository after the second commit

What do we want to do with the edits to `README.md`? We get some hints by the `git status` output. If we want to reset it to the state of the last commit, we can run `git checkout` for that particular file (we will come back to `git checkout`). In this case, we want to keep the changes. So, we stage them and commit them.

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

At this point the changes previously made to `README.md` are staged, shown in Figure 2.14.

Figure 2.14: Staged modifications to README.md

```
$ git commit -m "Update README"
[master 6ab7792] Update README
 1 file changed, 1 insertion(+)
```

Figure 2.15 shows the state after staging and committing the changes made to `README.md`.

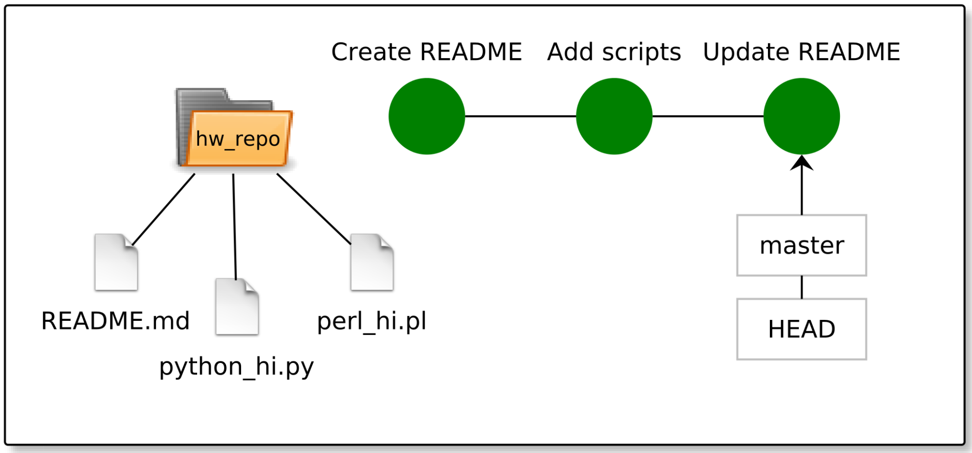


Figure 2.15: The state of the file system and repository after the third commit

2.7 Staging all the changes

Changes can always be staged one at a time. Often, it can be useful to stage many files at once. This can be done by supplying a directory to the `git add` command rather than a specific file. Then all the files contained in that directory are staged.

To stage files starting in your current directory, use the `.` as argument. (The dot refers to the present working directory in UNIX). If you want to stage all files in a Git repository, you can use the `:/` as argument. This is the ”Git-root” - starting from the top directory in your Git repository.

Note!

The `git add` command does by default not stage removal of files. To stage the removal of files, either use the `git rm` command, or use the `-A` flag (`git add -A`).

2.8 Exercises

Note: You can unstage staged files using the `git reset` command.

Now, you will independently create a new repository. All the commands needed have been presented in this chapter. If you get stuck, you can take a look at the previous pages.

2.8.1 Getting to know the commands

While running the commands below, trace the current state of the repository using the command `git status`.

1. Create an empty directory and initialize a new repository within it.
2. Create a new file within the directory (`git init`).
3. Prepare the file for being committing by adding it to the stage (`git add`).
4. Create your first commit, storing the changes you have made in the repository (`git commit`).
5. Add multiple files, and try including only some of them in a commit.
6. Remove one of your files, and include the removal in a commit.
7. Continue making changes (adding, editing and removing files) to your file tree and commit them to the repository until you feel comfortable using those commands.

2.8.2 Understanding the system

1. Visualize the file tree, the repository, the stage and the heads in the steps done in the previous exercise. Either do this on paper, or if you are in a hurry/lazy, in your head. How does each of the commands effect the file tree, stage and repository?

2.9 Recap

2.9.1 Concepts

- What does a commit represent, and what information does it contain?
- What roles do the file tree, the repository and the stage/index have?
- What are *heads*, and what are their purpose?
- How is the `HEAD` head special?

2.9.2 Commands

Make sure that you are comfortable with the following commands.

```
git init

git add [-A] <path>

git commit [-m <message>]

git status
```

Chapter 3

Investigating your history

3.1 Overview

In this chapter, we will:

- Investigate the history using the commands `git log` and `git checkout`
- Discuss how to compare states of the repository using `git diff`
- Discuss regretting changes and wanting to change history

3.2 Looking at the log

To get the history of a commit, we use the command `git log`. This shows all the previous commits leading to the current HEAD. It shows the following information about each commit:

- ID
- Author and the author's email
- Commit date
- Commit message

```
$ git log
commit 6ab77927c5dabcf7ea81bc9145ca88ebde002222
Author: Jakob Willforss <my_mail@mail.com>
Date:   Fri May 27 08:46:10 2016 +0200

    Update README

commit 7100d369d954e958632ba29dfcedf94e00f816fb
Author: Jakob Willforss <my_mail@mail.com>
Date:   Wed May 25 18:22:37 2016 +0200

    Add scripts

commit b58c6c339f8393879368db6006d0ccf738e13efa
Author: Jakob Willforss <my_mail@mail.com>
Date:   Wed May 25 18:05:36 2016 +0200

    Create README
```

Command: `git log [--oneline]`

Print the commits preceeding the current `HEAD`.

Figure 3.1: git checkout - Check out commit or head

Those commits correspond to the commits we have been creating in this chapter. They are the three commits visualized in figure 3.2.

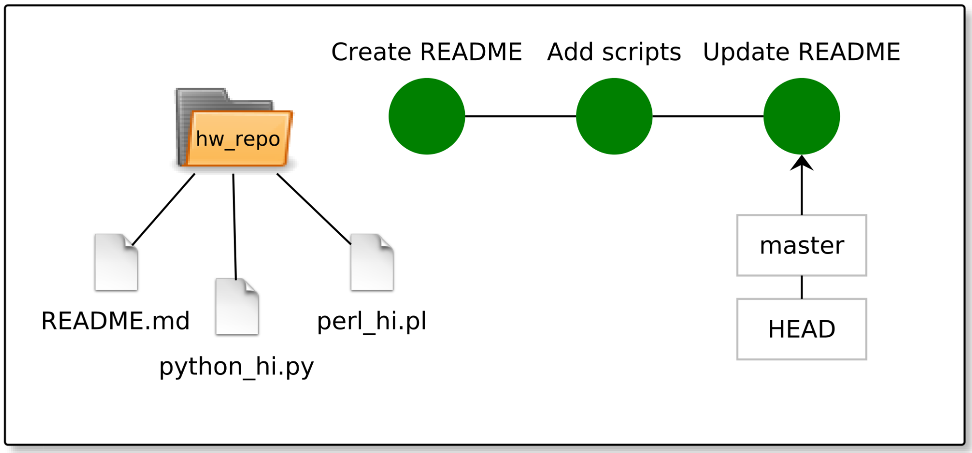


Figure 3.2: The state of the file system and repository

Sometimes, it is useful to look at the logs in a more compact form. Then, we can add the `--oneline` flag.

```
$ git log --oneline
6ab7792 Update README
7100d36 Add scripts
b58c6c3 Create README
```

Note

When running the command `git log` you don’t necessarily see all of the commits made in your repository. You get the history leading to the commit to which the `HEAD` reference currently points.

Note

On some configurations, Git opens the `git log` output in `less`. If you prefer to get it printed directly to the terminal, you can edit this setting by changing the ”core-pager”, the default program for visualizing longer text:

```
git config --global core.pager cat
```

If you later want to reset the core pager to `less`, you can do so by running:

```
git config --global core.pager less
```

3.3 Investigating differences between commits

To compare differences between commits, we can use the command `git diff`. You can retrieve the ID for the commits using the `git log` command, and then specify which commits you want to compare. If you only specify one commit, it will be compared to the currently active `HEAD` commit.

```
$ git diff b58c6c3
... comparing HEAD to specified commit
$ git diff b58c6c3 7100d36
... comparing specified commits
```

Command: `git diff <commit2> [<commit2>]`

Either compare current commit to target commit (one argument), or specify two commits to compare. Prints information about what lines that have been changed between the commits.

Figure 3.3: git diff - Print differences between commits

3.4 Going back in time

To check out the previous commits, we can use the `git checkout` command. There are some different ways we can specify a commit:

- Using the commit's ID - It is enough to specify the initial 6-8 letters
- Using a head pointing to the target commit
- A `tag` which can be used to mark important commits (more on tags in Chapter 6)

When you run the `git checkout` command, your `HEAD` is moved to the target commit and your file tree is changed to match the state represented by the commit. Make sure that you have committed your changes before running `git checkout` - otherwise, Git will protest.

Note, if you are tracing the commits your commits will have different IDs. Use them instead of the ones shown below.

```
$ git checkout b58c6c3  # Use your own commit ID
Note: checking out 'b58c6c3'.
... (more text)
```

The file tree is now changed to its state before adding the Hello World scripts and before adding a second line to the `README.md` We can freely investigate the current state of the file tree.

```
$ ls
README.md
$ cat README.md
A Hello World collection
```

The state of the repository after checking out the first commit is shown in Figure 3.4. How do you think the history looks now? Let's find out.

```
$ git log --oneline
b58c6c3 Create README
```

Note

After checking out a commit, you will get a warning that you now are in a 'detached HEAD' state, meaning that your HEAD currently isn't referring to the head of a branch. You can make commits here, but to save them you will need to create a new branch for them - otherwise they will be removed on the next `git checkout`.

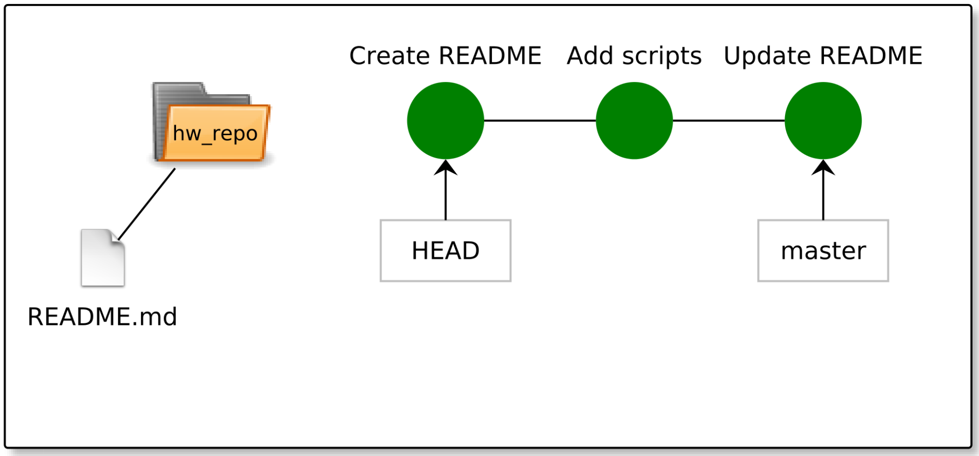


Figure 3.4: The state of the file system and repo after checking out the first commit

Now we only have one commit in the history. The reason for this is that the starting point for the `git log` command is the first commit (b58c6c3). That commit has no way of seeing downstream commits as the commits only know about their parents. When going back to the master, we will see the three commits again.

We can at any time return to the master branch, and its latest commit.

```
$ git checkout master
Previous HEAD position was b58c6c3... Create README
Switched to branch 'master'
```

Now, the file tree is back to how we left it after the last commit. The state of the repository is shown in figure 3.5, identical to how it was before checking out the history.

```
$ ls
perl_hi.pl  python_hi.py  README.md
$ cat README.md
A Hello World collection
Hello World in different languages
```

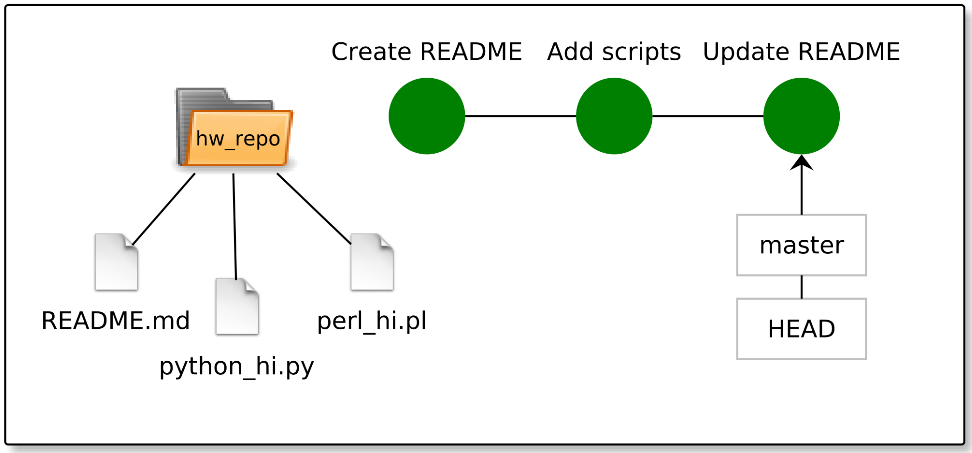


Figure 3.5: The state of the repository after returning to the master

Command: `git checkout <commit/head/tag>`

Moves the HEAD (and changes the file tree) to target commit, branch and/or tag.

Figure 3.6: git checkout - Check out commit or head

3.5 Unstaging changes

If we have staged the file, but realize that we don't want to include it in the next commit, we can unstage it using the command `git reset`.

```
$ echo '#!/usr/bin/beef' > hi.bf
$ echo '++++++++++[>+++++++>++++++++++>+++>+<<<<-]\
>++.>+.+++++++..+++.>++.<<+++++++++++++++.\
>.+++.------.--------.>+.>.' >> hi.bf
$ git add hi.bf
```

At this point, the Hello World script `hi.bf` has been created and staged, ready to be committed. If we at this point changes our mind, we can unstage and remove the file from the repository.

```
$ git reset hi.bf   # Unstage hi.bf
$ rm hi.bf          # No traces left of hi.bf
```

Plus points if you can figure out which programming language this version of "Hello world!" was written in.

Command: `git reset <filepath>`

Unstages a staged file.

Figure 3.7: git reset - Unstages a staged file

3.6 Reverting to older version

If you want to reset the file to the state of an older commit, you can use the `git checkout` command instead. To demonstrate, let’s do an unwanted edit to one of our scripts.

```
$ echo ’print("Unwanted text")’ >> python_hi.py
$ git add python_hi.py
$ git commit -m "Add unwanted line"
[unwanted_edit 80d3304] Add unwanted line
 1 file changed, 1 insertion(+)
```

If we want to revert the file to a previous stage, we find the commit we want to revert it to, and check out that particular file. Then we commit the older version of the file.

```
$ git log --oneline
80d3304 Add unwanted line
6ab7792 Update README
7100d36 Add scripts
b58c6c3 Create README
$ cat python_hi.py
#!/usr/bin/python3
print("Hello world")
print("Unwanted text")
$ git checkout 6ab7792 python_hi.py
$ cat python_hi.py
#!/usr/bin/python3
print("Hello world")
$ git commit -m "Restore script"
[unwanted_edit 4d1a062] Restore script
 1 file changed, 1 deletion(-)
```

Note what happens if we run `git log` now. We have reverted the file *by adding another commit reverting the file.*

```
$ git log --oneline
4d1a062 Restore script
80d3304 Add unwanted line
6ab7792 Update README
7100d36 Add scripts
b58c6c3 Create README
```

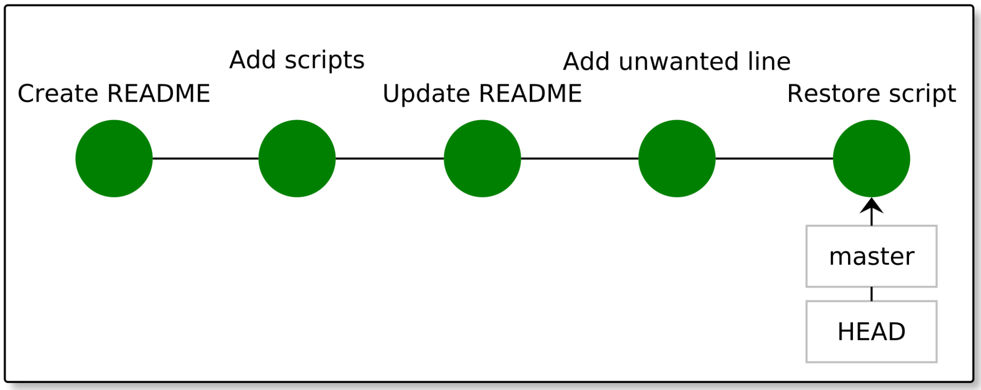


Figure 3.8: Repository after correcting unwanted edits in fourth commit

This means that the version of the file where it had an extra line still exists within the Git history. This is good - History is history. We can write new history which changes a file back to an earlier state, but should almost, almost never fiddle with the existing history. The repository for the final state is shown in figure 3.8. Note that the state of the *file tree* would be exactly the same if the third commit were checked out.

3.7 Thoughts on regretting commits

It is generally considered bad practice to go back and edit existing commits, *especially* if you have pushed them to a remote - sent them to a central repository which you share with others. If you start editing the history on a collaborative project, you not only change the history for yourself but for your collaborators too.

Under exceptional circumstances, it might be reasonable to fiddle with the history. For example if you accidentally commit 20 GB of sequence data to your repository, it might be better to correct the commit than sending it to your collaborators. Otherwise, it is generally better to create another commit which undoes your mistakes.

3.8 Exercises

3.8.1 Trying out the commands

Let's continue working with the repository that you created in the exercises for Chapter 2.

1. Print the log of your repository. Try both full format and one-line format. Can you easily trace the history of your files based on your commit messages?
2. Take a look at your current file tree (simply by using `ls` and `cat`). Run `git checkout` for one of your commits. Check the file system again. Can you see the changes?
3. Before going back to the master, compare the `git log` output to what you got before. Do you see a difference? Why?
4. Go back to the `master`.
5. Revert one of your files to an older version and commit the revert.
6. Navigate around in your repository until you feel comfortable using the commands `git log` and `git checkout`.

3.8.2 Understanding "git checkout" and HEAD

1. For the previous exercises, visualize how each step effects the file tree, the repository and in particular the HEAD.

3.8.3 Exploring .git (*)

1. Take a look into the .git-directory located at the top level in your Git repository. This directory contains everything related to the repository.
2. See if you can figure out how files in this directory are related to the output you get from for example running `git log`.

3.9 Recap

3.9.1 Concepts

- How are the file tree and the HEAD effected when running `git checkout`? Is the repository effected?
- How are the output from `git log` and the `HEAD` related?
- How do you best handle resetting a particular file to a previous version?

3.9.2 Commands

- `git log [--oneline]`
- `git checkout <commit ID/head/tag>`

Chapter 4

Branches

4.1 Overview

In this chapter, you will learn:

- What branches are in Git and how they are useful
- How to create branches
- How to merge branches back into your main branch
- How to handle conflicting file changes when merging branches

4.2 What are branches?

A `branch` in Git allows you to work in parallel on different features before integrating them into your main branch (commonly the `master` branch).

Often, the the `master` branch is used for stable versions of the program. Branches can be used to develop features independently without disrupting the stable version. The branches are then integrated back into the master when the feature is completed.

4.3 Creating a branch

In this section we will demonstrate the process of creating a new branch and use it to develop a feature which finally is merged back into the `master`. Figure 4.1 shows the state of the repository before creating the branch.

Note

Our example repository have backed up to the third commit of our repository (the state after Chapter 2) in order to make it easier to visualize. If you are following along, you can simply continue developing on the last commit.

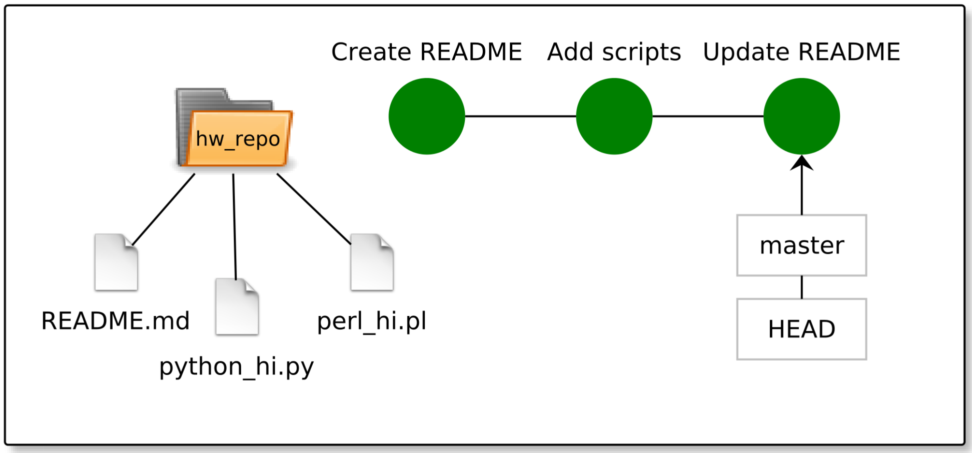


Figure 4.1: State of repository before branching

We create the new branch by initiating a new `head` at the current commit. To initiate the head, we use the command `git branch <branch name>`:

```
$ git log --oneline
6ab7792 Update README
7100d36 Add scripts
b58c6c3 Create README
$ git branch py_upgrade
```

We initiated a new branch named `py_upgrade`. This resulted in the repository creating a new head. At this point, our repository looks like figure 4.2.

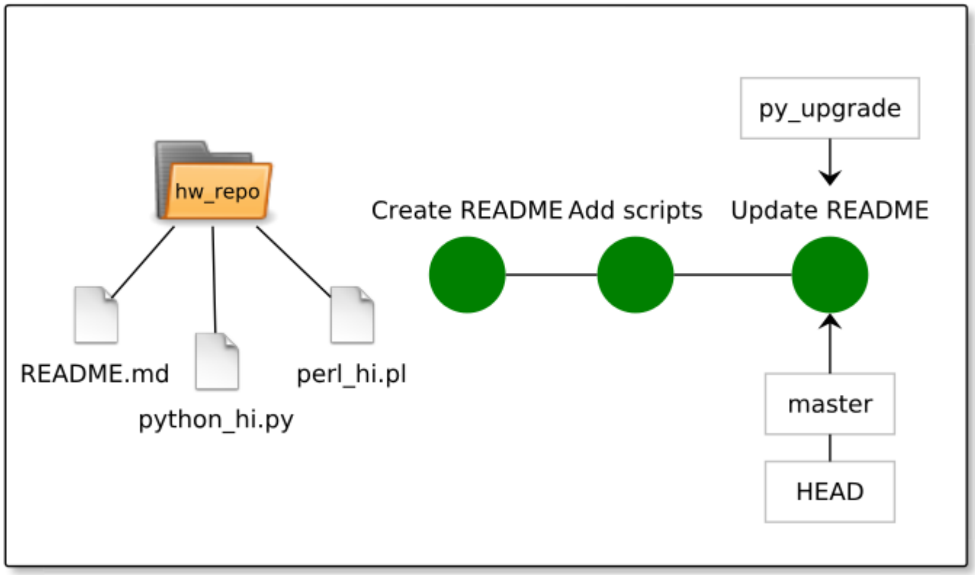


Figure 4.2: State of with new branch initiated

We have only created a new head at this point - there are no commits added to the branch and our HEAD still refers to the master branch. If we want to start working with the new branch, we need to switch HEAD to my_feature.

We can see all available branches using the git branch without any input. The asterisk shows where the HEAD is currently pointing. Let's switch to our new branch using the git checkout command.

```
$ git branch
* master
  py_upgrade
$ git checkout py_upgrade
Switched to branch 'py_upgrade'
$ git branch
  master
* py_upgrade
```

Let's make a new feature for the Python-script. In this case, the script was opened in the nano text editor. The cat command is used to show the file after the edits were completed.

```
$ nano python_hi.py
$ cat python_hi.py
#!/usr/bin/python3
import random
print("Hello world")
if random.random() > 0.5:
    print("Again: Hello world!!")
```

The Python script now randomly adds an extra "Hello world!" half of the times it is run. Next, we stage and commit the changes.

```
$ git add python_hi.py
$ git commit -m "Upgrade py"
[py_upgrade 159c0db] Upgrade py
 1 file changed, 3 insertions(+)
$ git log --oneline
159c0db Upgrade py
6ab7792 Update README
7100d36 Add scripts
b58c6c3 Create README
```

We added a new commit containing our new feature. Figure 4.3 shows the repository at this point.

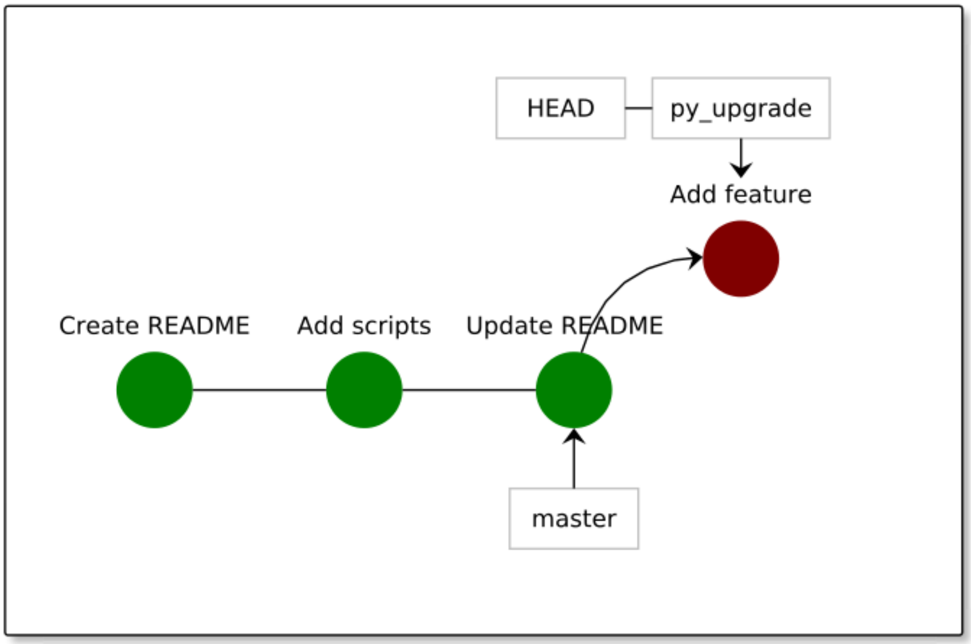


Figure 4.3: Branch with added commit

Note

Note that the current file tree represents the state of the repository where the HEAD is pointing. If we go back to the `master` head, we will return to the state of the files before we started developing our new feature.

Command: `git branch [<head-name>] [<starting-point>]`

Creates a new head with the given name and point that to the given commit object. If no starting-point argument is supplied, the branch will created from the current commit (`git branch branch_name`).
If used without supplied argument, it will list available branches (`git branch`).

Figure 4.4: git branch - Create a new branch in your repository

4.3.1 Merging branches

It is time to merge our changes back into the master branch. To do this, we use the `git merge` command. `git merge` incorporates the changes made by the other branch into the current branch. It will create a new commit, and both the heads will end up pointing to the same commit.

Let's merge our changes back into the `master`. There are no conflicts here which will make the re-merge smooth and pain-free.

Command: `git merge <branch_to_merge>`

Merges the given branch into the currently active branch.

Figure 4.5: git merge - Merge a branch into the current branch

```
$ git checkout master
Switched to branch 'master'
$ git merge py_upgrade
Updating 6ab7792..159c0db
Fast-forward
 python_hi.py | 3 +++
 1 file changed, 3 insertions(+)
```

No problems here - A new commit was created implementing the changes to the `new_feature` branch into the `master`. Figure 4.6 shows the repository at this stage.

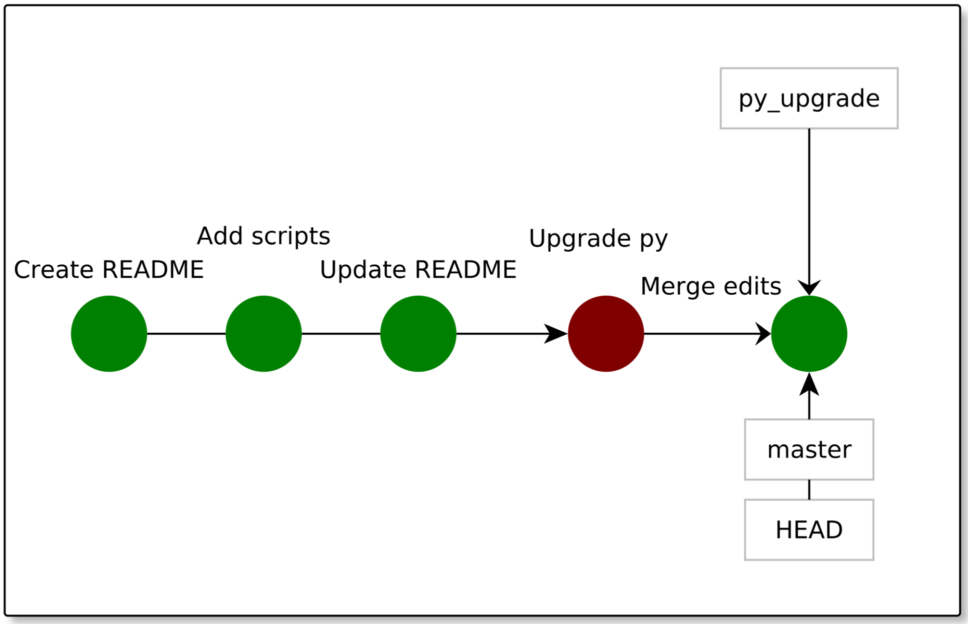


Figure 4.6: Smoothly merged repository

4.3.2 Dealing with merge conflicts

The previous merge went smoothly as we didn't have any conflicting changes (changes made to the same part of the same file) made between the branches. Now, let's see what happens if we try to merge our branches back together after doing some conflicting changes. We now back up to the state of the repository shown in figure 4.1.

Note

For ease-of-presentation, the repository has been reverted to its state before performing the merge in the example below. This means that you aren't able to exactly trace the state used in the example. It is recommended that you add a new commit to your branch, and then create a parallel edit on your master in order to try out solving a merge conflict.

Let's first go back in time to before we merged the branch into the master. Now, we implement clashing edits on the `master` branch.

```
$ git branch
* master
  py_upgrade
$ nano python_hi.py
$ cat python_hi.py
#!/usr/bin/python3
print("Hello world")
print("Clashing edits!")
$ git add python_hi.py
$ git commit -m "Create clash"
[master 4241ef3] Create clash
 1 file changed, 1 insertion(+)
```

Now, we have parallel changes on the `new_feature`-branch and on the `master`-branch. The state of our repository is shown in figure 4.7.

If the changes are made in different files, or even in different parts of the same file, no merge conflicts will occur. The default text editor will be opened, and you will be able to enter a commit message for the merge commit.

If you on the other hand have merge conflicts by editing the same file in the same place (as we do here), you need to manually pick the edits you want to use.

```
$ git merge py_upgrade
Auto-merging python_hi.py
CONFLICT (content): Merge conflict in python_hi.py
Automatic merge failed; fix conflicts and then commit the
result.
```

You will have to resolve this manually, either by editing the file directly or using a merge-tool. Here, we will edit the file directly. If you encounter merge-problems frequently, take a look into merge-tools like *meld* (`http://meldmerge.org/`).

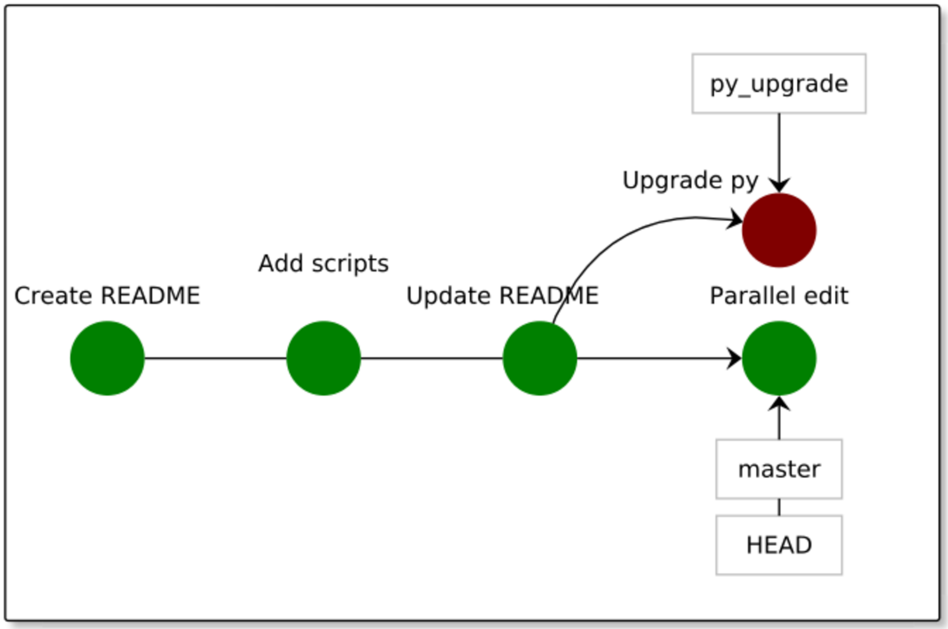


Figure 4.7: Parallel edits on master and branch

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      python_hi.py

no changes added to commit (use "git add" and/or "git commit
-a")
$ cat python_hi.py
#!/usr/bin/python3
import random
print("Hello world")
<<<<<<< HEAD
print("Clashing edits!")
=======
if random.random() > 0.5:
    print("Again: Hello world!!")
>>>>>>> py_upgrade
```

The lines between `<<<<<< HEAD` and `=======` are present in the current HEAD (in this case - the master branch). The lines between ======= and >>>>>>> py_upgrade are present in the python_hi_upgrade-branch.

In this case, we would like to retain the changes made to the py_upgrade branch. To do this we simply remove the dividers inserted by Git (<<<, === and >>>), alongside with the lines that we don't want to keep.

```
$ nano python_hi.py  # Remove merge-lines and "Clashing edits"
$ cat python_hi.py
#!/usr/bin/python3
import random
print("Python say: Hello world!")
if random.random() > 0.5:
    print("Again: Hello world!!")
$ git add python_hi.py
$ git commit -m "Merging clashing changes"
[master 791e895] Merge edits
$ git log --oneline
791e895 Merge edits
4241ef3 Create clash
159c0db Upgrade py
6ab7792 Update README
7100d36 Add scripts
b58c6c3 Create README
```

Now, we have all the history in place, and our changes has been merged into the master branch. At this point, our repository looks like the following:

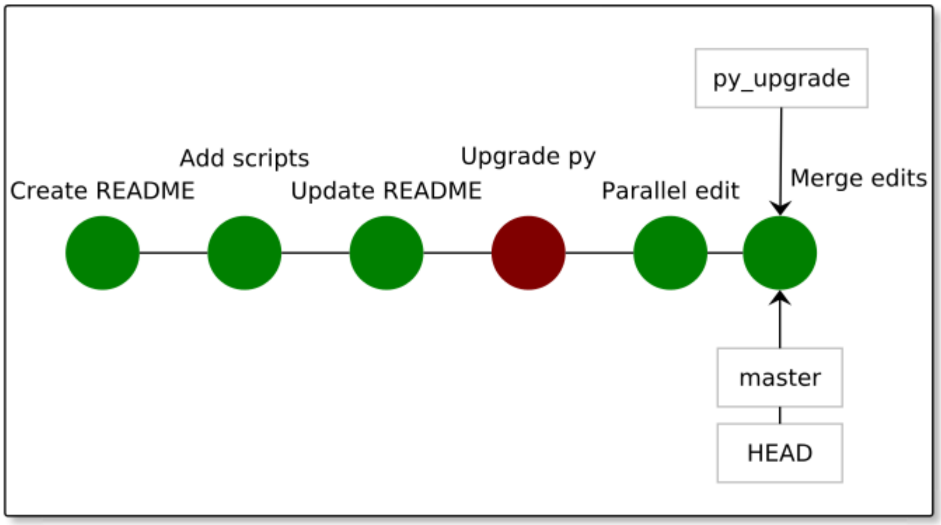


Figure 4.8: Successful merge after merge conflict

Note that the py_upgrade branch is still there, even if it has been merged back into the master. If we don't want to use it any more we could simply remove it from the repository.

```
$ git branch
* master
  py_upgrade
$ git branch -d py_upgrade
Deleted branch py_upgrade (was 159c0db).
$ git branch
* master
```

4.4 Exercises

The exercises presented here will work fine for any repository. You could either continue using the repository you built in the previous chapters, or create a new one.

The concepts of this chapter are important, especially understanding merge-conflicts which occurs both when working with branches and remote repositories. With that said, if you are short on time, make sure that you make it through Chapter 5. Then you could return here at a later point.

4.4.1 Using branching (*)

Here, we will go through the steps for branching that were explained in the previous text. If you get stuck, or aren't sure on the meaning of the steps, take a look into the previous text.

1. Run the `git branch` command and check what branches are present in your repository before initiating new branches.
2. Create a new branch for a new feature you want to implement. Check the output of `git branch`.
3. Switch to your new branch using the command `git checkout`. Check the output of `git branch`. Remember, Git will protest if you have any uncommitted changes.
4. Implement a new feature using one or more commits, preferably including changes to at least one of your existing files.
5. Return to the master by running `git checkout`.
6. Now, add some commits to your master. Make sure to include some edits to the same files which you worked with on your branch, with at least one edit directly clashing with the edits made on the branch.
7. Let's merge the feature branch back into the master using `git merge`. Before merging, stop for a moment, and visualize the current state of the repository and the file tree. How will they look before and after performing the merge?
8. Perform the merge. You will likely need to make a merge commit. You can check the current status using `git status`. Do you have any files for which the merge failed? (Written as "both modified" in the `git status` output). Resolve the conflicts by inspecting the files and removing the lines that you don't want to keep.

Merge conflicts can be a particularly confusing part when getting starting with Git. When you encounter a merge conflict, make sure that you understand what is going on and what you want to accomplish. You can always refer to this material if you are unsure on what is going on.

4.4.2 Further branching (*)

- The asterisk in the `git branch` output shows you the currently active branch. What output do you get if you check out a commit not related to a branch? What does it mean?

- Visualize the steps of the previous exercise. Make sure that you have a good understanding of what happens with the file tree, repository and heads when running each of the commands.

4.5 Recap

4.5.1 Concepts

- Can you name one or more potential purposes of branches in Git?
- What is a merge conflict, and when does it occur?

4.5.2 Commands

- `git branch [<branch_name>] [<target_commit>]`
- `git branch -d <branch_name>`
- `git merge <branch_name>`
- `git checkout <branch_name>`

Chapter 5

Remote repositories

5.1 Overview

In this chapter, we will go through:

- Remote repositories
- Cloning the repository of a bioinformatic software (STAR)
- Merge conflicts with remote
- Setting up a new GitHub repository

Up until now we have been working with a single, local, instance of our repository. This is useful by itself, but Git really starts to shine when you start using a remote repository. By using a remote repository, you can access your files and history from any computer and allow others to access and contribute to the code base.

Furthermore, if you are using a service such as GitHub, you are able to access your code through the web browser at any time.

5.1.1 File tree / stage / repository / remote

In this chapter, we are introducing the final layer used in Git. Previously, we have been working with the file tree, the stage and the repository. Hopefully, you both have a firm grasp of their different purposes now, and are feeling comfortable with the commands used for working with a local repository.

Now, we will introduce the remote repository, which simply is another copy of our repository which mostly resides on a server-computer (for example on GitHub's servers). We can interact with the remote to retrieve changes sent to it from other users, and we can send our own changes to the remote. The whole process is visualized in figure 5.1.

Files commonly travel the following path in Git:

1. They get created or added to the file tree.
2. The changes are added to the stage, prepared for being committed to history.

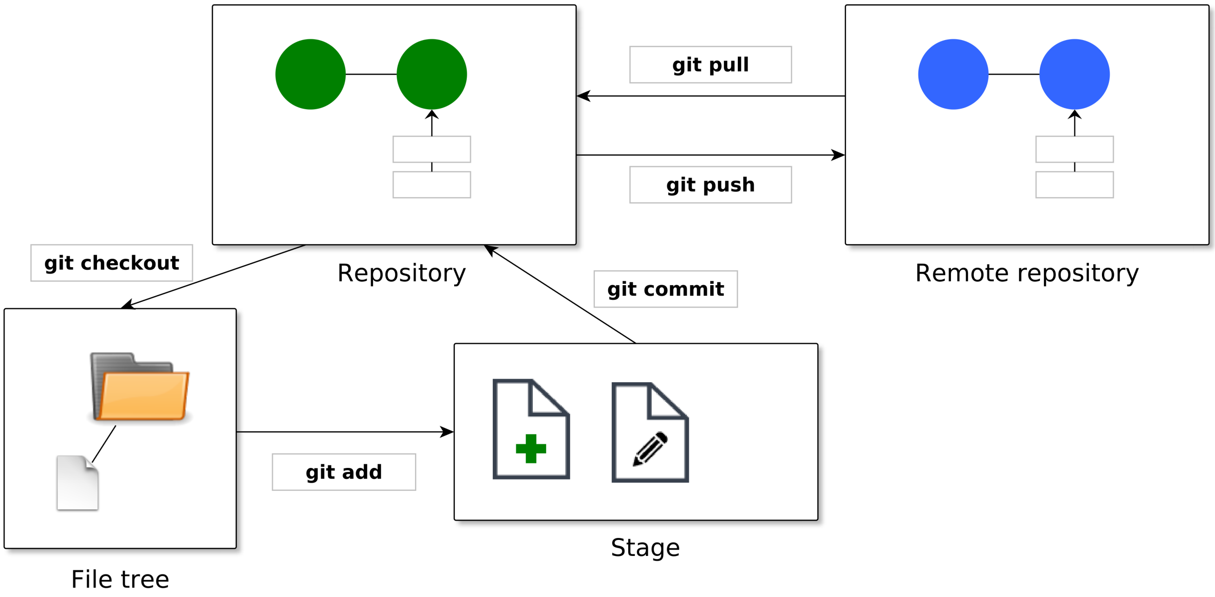


Figure 5.1: Visualization of the different stages in Git

3. The staged set of changes gets captured into a commit, which is stored in the repository.
4. Changes made to the repository are synced with the remote repository. This means that changes made to the remote is implemented in our local repository, and changes made to our local repository is implemented into the remote.

The remove repository is simply a regular repository which we have specified as the `remote`. Commonly, many local repositories have the same remote specified and uses it as a central point to send and retrieve changes to/from. This is visualized in Figure 5.2.

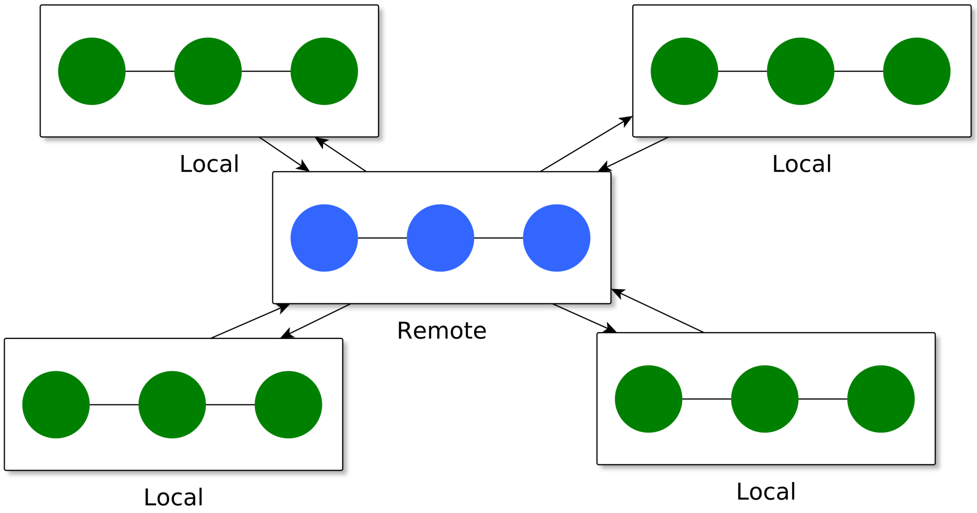


Figure 5.2: Many local repositories are often interacting with a single remote

5.2 Cloning a repository

When you copy an existing Git repository, it is said that you `clone` that repository. When cloning in Git, you get a full local version of the repository which is identical to the one you cloned. You then have access to the entire history of the repository.

Often anyone can to retrieve the entire repository, and later edits made to it, but in general, only selected users are able to send their changes directly. In open source development something called `pull requests` is used where user can propose edits to the remote which then can be approved or rejected by an admin.

5.2.1 Cloning the STAR repository

Much bioinformatic software is available in open repositories, often hosted on GitHub. In this chapter, we will work with the STAR repository. STAR (Spliced Transcripts Alignment to a Reference) is a popular spliced-read-aligner, similar to TopHat but with much faster processing speed (and higher RAM requirements). It is hosted on GitHub at the following address:

https://github.com/alexdobin/STAR

If you navigate to that address you will see a visual display of the source files, a README and some graphical tools which can be used to interact with and find out more about the repository. This is a Git repository similar to the one we created in the previous chapters which is hosted on GitHub's servers.

To clone the repository, we need to use its address. It is the same as the URL for the GitHub page, with an added `.git` at the end.

`https://github.com/alexdobin/STAR.git`

To clone the repository, we run the `git clone` command, and supply the URL to the Git repository.

```
$ git clone https://github.com/alexdobin/STAR.git
```

Command: `git clone <path>`

Create a full local copy of the target Git-repository.

Figure 5.3: git clone - Create a local copy of a repository

This will create a folder in your current working directory, which will contain the repository. We can now navigate into the repository and take a look. We will investigate it further in the exercises.

5.3 Clone and work with an existing repository

In the previous section, we created a copy of the STAR repository. The version hosted on GitHub is the `remote` repository for STAR. After cloning we have an exact local copy of it. The only thing distinguishing the remote is in this case that the developers have agreed on using the GitHub-version as their remote. Otherwise, there is nothing special with it.

If we have the access rights (for example if we are the original author of the repository) we can make changes to the remote by first making local changes, and then pushing them to the remote location.

5.3.1 Cloning the remote

Here, we will do a brief demonstration of how it could look if you had access rights to a particular repository and wanted to make changes after cloning it. Note - You do not have access to the particular repository used in this example, which means that you cannot push changes to it.

```
$ git clone https://github.com/Jakob37/GitCourse.git
Cloning into 'GitCourse'...
remote: Counting objects: 15, done.
remote: Total 15 (delta 0), reused 0 (delta 0), pack-reused 15
Unpacking objects: 100% (15/15), done.
Checking connectivity... done.
$ cd GitCourse
$ git log --oneline
ff79a43 Add files
c43f754 First commit
```

At this point, we have cloned the remote repository which contains two commits. Now, we have a local copy of it. Figure 5.4 shows how the repository looks before making any changes. Our repository contains two commits, and is synced with the remote.

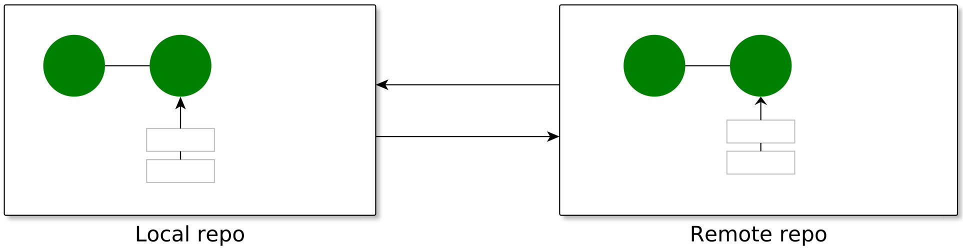


Figure 5.4: State of the local and remote repositories - No local changes

5.3.2 Making local changes

```
$ nano ruby_hi.rb
$ cat ruby_hi.rb
#!/usr/bin/ruby
print("HI!\n")
$ git add ruby_hi.rb
$ git commit -m "Add Ruby Hi script"
```

Now, we added a small Ruby script printing "HI!". We also staged the file for being committed, and finally created a commit.

The current state of the repository is shown in figure 5.5. The purple commit is the newly created local commit. As we haven't sent the commit to the remote, our remote only contains the two first commits.

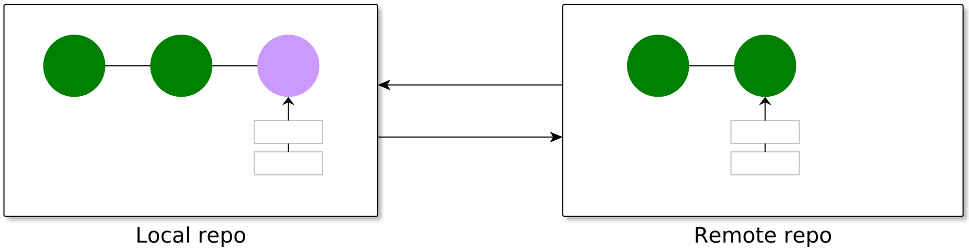


Figure 5.5: Stage of the repo, local and remote - Local changes pushed to remote

The changes are only present locally. If others should be able to see the changes, we need to send them to the remote too. The repository pushes per default the repository to a head called `origin`. To see the current remote paths for pulling and pushing, we can use the command `git remote -v`.

```
$ git remote -v
origin  https://github.com/Jakob37/GitCourse.git (fetch)
origin  https://github.com/Jakob37/GitCourse.git (push)
```

It looks fine here - our remote was set up properly when cloning the repository. When creating our own remote we will need to set this up for ourselves.

Command: `git remote [-v]`

Get the current remote for the repository. If used with the `-v` flag, the full URL is printed.

Figure 5.6: git clone - Create a local copy of a repository

Command: `git fetch`

Update information about state of the remote.

Figure 5.7: git fetch - Sync with the remote

Command: `git pull`

Retrieve changes made to the remote

Figure 5.8: git pull - Retrieve changes made to the remote

5.3.3 Pushing to the remote

Before pushing new commits, we want to retrieve existing changes made to the remote.

We have two commands for this. `git fetch` fetches information about changes done on the remote - but doesn't retrieve them. `git pull` retrieves the changes from the remote, inserting the new commits into our local repository.

```
$ git fetch
# No output - no new changes
$ git pull
Already up-to-date.
```

In this case, there weren't any new changes. We are free to push our new commits to the repository, so that all collaborators can access them. Note: This is only possible to do if you have access to the repository.

```
$ git push
# You are prompted for username and password
Username for ’https://github.com’: Jakob37
Password for ’https://Jakob37@github.com’:
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 614 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
To https://github.com/Jakob37/GitCourse.git
   984f0ac..3ab3714  master -> master
```

When pushing changes, you are prompted for your username and password. Figure 5.9 shows the state of the local and remote repositories after pushing the changes.

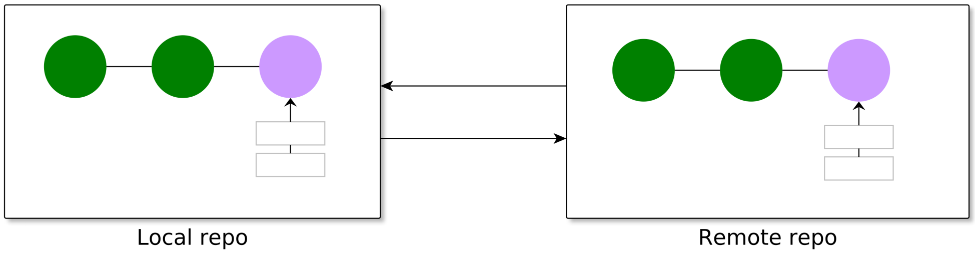


Figure 5.9: Stage of the repo - Local changes pushed to the remote

Command: `git push`

Send local commits to the remote

Figure 5.10: git push - Send local commits to the remote

5.4 Parallel changes to the local and to the remote

Let’s back up to before pushing our changes to the remote and investigate how it would look if there would have been changes made to the remote. If there already were changes made to the remote, we would get the following output from the `git fetch` command:

```
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/Jakob37/GitCourse
   a75107d..984f0ac  master     -> origin/master
```

Looks like we have changes to to take into account. We can check the current status of our local master-branch compared to the `origin` by running `git status`

```
$ git status
On branch master
Your branch and ’origin/master’ have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working directory clean
```

Figure 5.11 shows the current state of the repositories.

If we try to push at this point we will get prompted for username and password, and then get the following message:

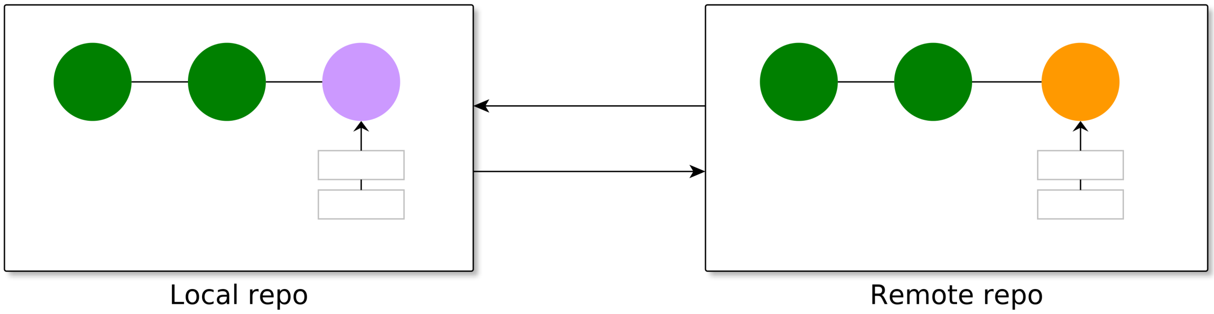


Figure 5.11: Diverging commits on master and remote

```
$ git push
Username for ’https://github.com’: Jakob37
Password for ’https://Jakob37@github.com’:
To https://github.com/Jakob37/GitCourse.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to
’https://github.com/Jakob37/GitCourse.git’
hint: Updates were rejected because the tip of your current
hint: branch is behind its remote counterpart. Integrate the
hint: remote changes (e.g. ’git pull ...’) before pushing
hint: again. See the ’Note about fast-forwards’ in
hint: ’git push --help’ for details.
```

We need to retrieve and integrate the remote edits in our repository before pushing our changes to the remote.

To do this, we will create a merge commit (similarly to when we were working with branches). If we are lucky, there are no conflicts with the remote changes.

```
$ git pull
# Here, the default text editor is opened
# Enter commit message, and then save/quit
 perl_hi.pl | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 perl_hi.pl
```

It seems like there were no conflicts. If there are, those are managed similarly to what was shown for branches in the previous chapter. The state of the local and remote repositories is shown in figure 5.12. We have both the local and the remote edits, as well as a merge commit.

We are ready to push our edits.

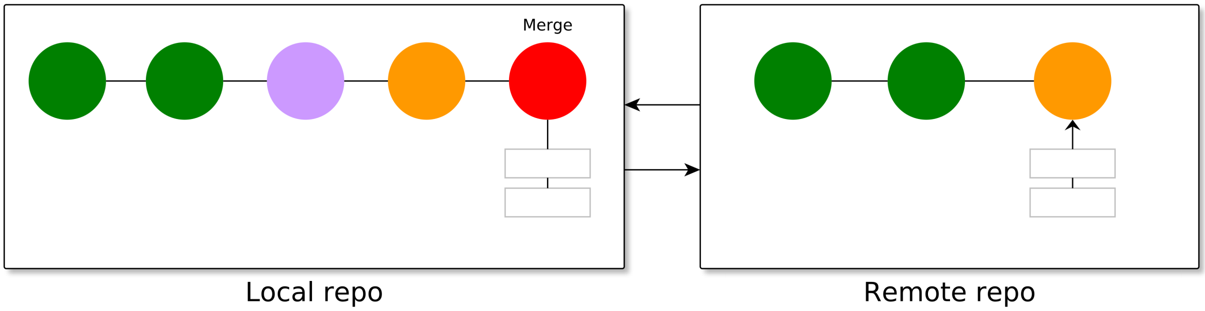


Figure 5.12: Integrating remote changes

```
$ git push
Username for 'https://github.com': Jakob37
Password for 'https://Jakob37@github.com':
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 614 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
To https://github.com/Jakob37/GitCourse.git
   984f0ac..3ab3714  master -> master
```

Now, the remote has finally received our changes. The final state of the repositories is shown in figure 5.13:

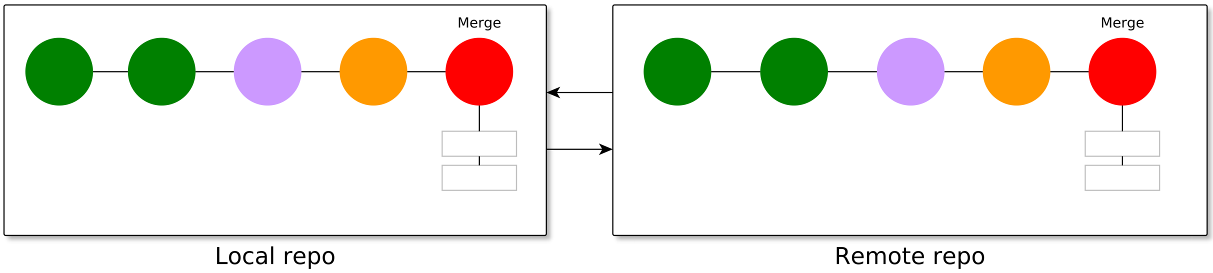


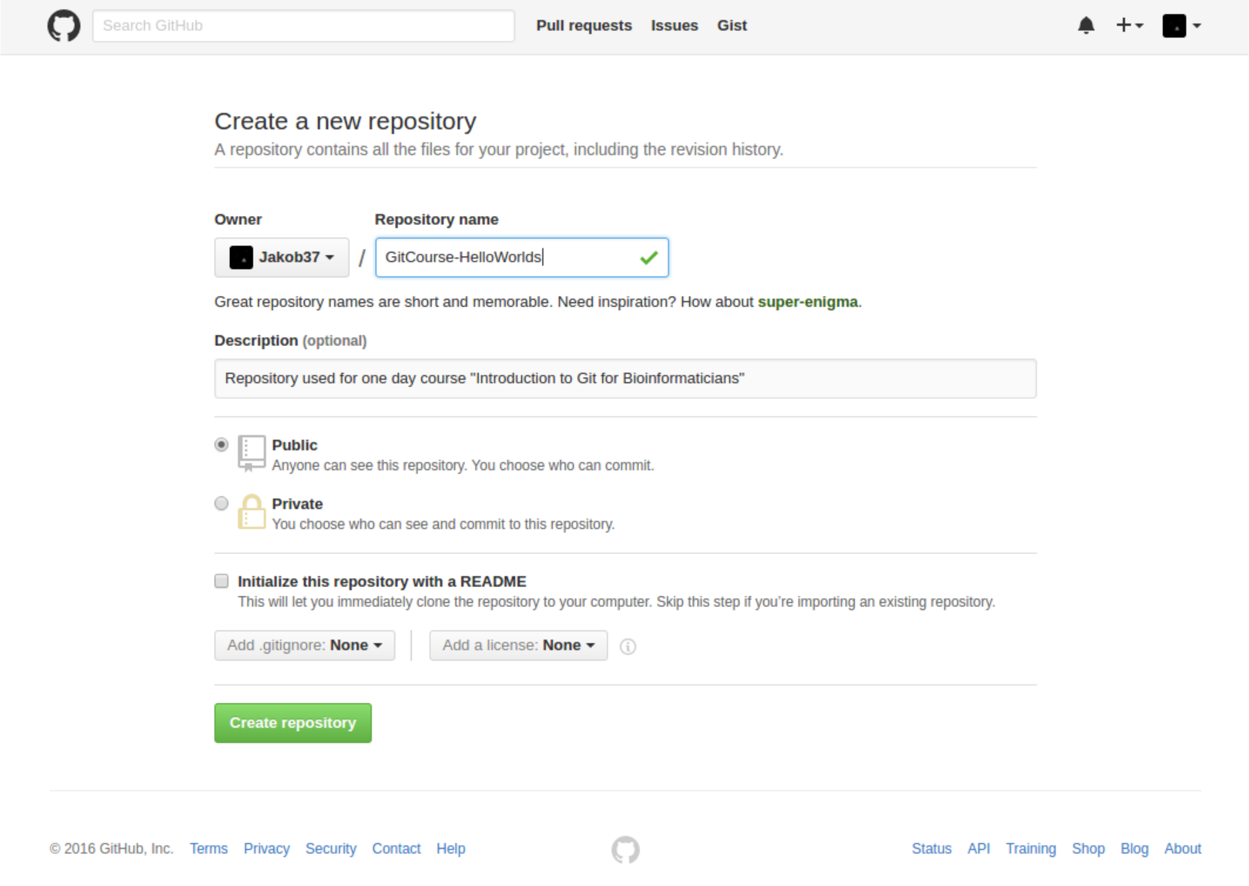
Figure 5.13: Final state after pushing merged changes

5.5 Setting up a remote on GitHub

Now, let's see how we can set up a remote repository for a local repository which we already have been working with. In this case, we will go though how to set it up on GitHub. If using other services, the way of creating the remote will be slightly different.

To use GitHub, we first need to create a new account. After going through the usual steps, we are ready to start creating a new repository by clicking "Create repository".

Here, we can fill in some information about the remote. The absolute minimum is a name for our newly created GitHub repo. This is shown in figure 5.14



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: Jakob37 ▾ / Repository name: GitCourse-HelloWorlds ✓

Great repository names are short and memorable. Need inspiration? How about **super-enigma**.

Description (optional)

Repository used for one day course "Introduction to Git for Bioinformaticians"

● **Public**
Anyone can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.

□ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾ | Add a license: **None** ▾

Figure 5.14: Details for the new repository

Now, let's initiate a local repository for which we want to assign our remote.

```
$ git init
$ touch a_file.txt
$ git add a_file.txt
$ git commit -m "First commit"
```

We are now ready to assign this as the remote for our local repository. To do this, we need to first use the `git remote add` command to set up the repository as our new remote. Then, initiate using this remote by using the `--set-upstream` flag with the `git push` command.

Finally, check that you have assigned the correct URLs using the `git remote -v` command.

```
$ git remote add origin https://github.com/Jakob37/GitCourse.git
$ git push --set-upstream origin master
Username for 'https://github.com': Jakob37
Password for 'https://Jakob37@github.com':
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 221 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/Jakob37/testtesttest.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
$ git remote -v
origin https://github.com/Jakob37/GitCourse.git (fetch)
origin https://github.com/Jakob37/GitCourse.git (push)
```

We are now ready to push and pull similarly to how we did before. If you get error messages, double-check that you have the correct paths.

Command: `git remote add origin <path>`
Command: `git push --set-upstream origin master`
Command: `git remote -v`

Collection of commands used to set up the link to the remote properly, and to check that the URLs are correct.

Figure 5.15: Commands for setting up the remote

5.6 GitHub, Bitbucket, and other places to host your repo

You can set up a remote on your own server (if you have access to one), but there are also a lot of great services available for hosting. Two of the most popular services are GitHub and Bitbucket, which both provides a graphical interface with a lot of functionality for working with the remote.

5.6.1 GitHub

https://github.com

GitHub is perhaps the most widely used hosting service for Git repositories, and is frequently used for hosting of open source bioinformatics software.

GitHub allows hosting of an unlimited number of public repositories for free. If you want to keep your code private (i.e. not openly accessible on GitHub), you need to pay $7 per month. Since May 2016 paying users can have an unlimited number of repositories - earlier you were limited to a maximum of five.

5.6.2 Bitbucket

https://bitbucket.org

Bitbucket is another popular hosting service provided by the software group Atlassin. Its main edge again GitHub is the ability for single users and small teams to have an unlimited number of private repositories for free.

They use a slightly different payment model. Instead of paying for having private repositories, you pay for the number of collaborators (people with edit rights to the repositories). If you want to have more than five contributors to a repository, you will need to enroll for a payment plan.

5.7 Exercises

5.7.1 Working with remote GitHub repositories

1. Go to github.com and create a new account.

2. Create a new repository on GitHub by clicking ”New repository” (Green button). Give it a name, and make sure to pick the ”Public” option.

3. Set up this as a remote for the repository you have been building throughout this course.

4. Push your repository, and check on GitHub that you can see all of your files and commits in the interface.

5.7.2 Collaborating with classmates on GitHub repositories (*)

Do this exercise if you have the time. If you are in a hurry, come back to this exercise after reading through the final chapter.

1. Give a friend with a GitHub account access to your repository (and make sure that you get access in return).

2. Clone your friends repository. Investigate the files, and the log. Have your friend been using clear commit messages?

3. Contribute to your friends repository by creating a new file, or making (reasonable) edits to one of his or her files.

4. Push the changes to the remote.

5. Retrieve the changes which your friend have made to your repository. You can investigate the changes made in the commit by clicking ”commits” and the commit(s) created by your friend.

6. As a challenge: Create commits editing the same files as the ones your friend have pushed to the remote. Solve the merge conflict that arises when you attempt to sync your changes with the repository.

5.8 Recap

5.8.1 Concepts

- What is a remote repository, and how is it useful?
- What are the benefits of using a hosting-service like GitHub and Bitbucket? (And why do you think GitHub is so popular for hosting bioinformatic software)
- When do you encounter merge-conflicts when working with remotes?

5.8.2 Commands

- `git clone <path>`
- `git fetch`
- `git pull`
- `git push [--set-upstream origin master]`
- `git remote add origin <path>`
- `git remote -v`

Chapter 6

Version control for bioinformaticians

6.1 Overview

Here, we will discuss Git-concepts which are especially useful in the context of bioinformatics.

- Excluding files residing in the file tree from the repository
- Version-tracking by tagging commits

6.2 Keeping the right files in version control

You sometimes encounter a situation where you want to keep files within the file-tree of a repository without having the repository track them. For example:

1. Either your editor or your code generates temporary files when in use.
2. Configuration files which are linked to one specific computer.
3. You want to keep data (for example - test datasets) in the directory of a script.

If the files are computer generated and automatically updating, or linked to some setup of your particular computer, you should probably exclude them from the repository. For example: This document is written in LaTeX and is version controlled in Git, but temporary files generated by the text editor (for example when generating the PDFs) are excluded from the repository.

It is generally not a good idea to track bioinformatic data in a repository. It is definitely not a good idea to track or try to push 10GB of sequence data to GitHub. Both GitHub and Bitbucket will both reject pushes containing too large files. It can sometimes be useful to keep very small test-datasets in the repository, but you should think carefully before committing non-source-code files to the repository.

6.2.1 .gitignore

To exclude particular folders or file-patterns from the Git repository, special files named ’.gitignore’ are used. To use those, simply create a file named ’.gitignore’ in the directory you want to control, and it will effect all its sub-directories.

To exclude a specific file, simply put its path into a `.gitignore` file. Relative paths will start from the directory of the `.gitignore` file.

```
$ echo "Subfolder/exact_file.log" >> .gitignore
```

To exclude all files in a directory, add the path to the directory. For example, if we want to exclude all files in the ”Subfolder” directory, then we would use the following:

```
$ echo "Subfolder/" >> .gitignore
```

There are also some patterns we can use to match particular suffixes or directory names, excluding files matching a particular pattern. An asterisk (*) can be used to match part of a file name. Two asterisks can be used to match any number of directories (**).

If we for example want to exclude all files with the suffix .log, we would do the following:

```
$ echo "*.log" >> .gitignore
```

If we want to exclude all directories named ”output” with all their sub-files, we would do the following:

```
$ echo "**output/" >> .gitignore
```

6.2.2 Removing a file from the repository, while keeping it in the file tree

The changes of the `.gitignore`-file is seen in the `git status` output as soon as the file is being tracked by the repository. This will not effect already committed files - It only ignores files which aren’t being tracked. Files that already have been committed need to be removed from the repository before being ignored.

To remove files from the repository without removing them from the file tree, we can use the `git rm --cached` command (add the `-r` flag to traverse down into directories).

If we for example want to remove all log-files that currently are being tracked, we would do the following:

```
$ git rm -r --cached *.log
rm 'file1.log'
rm 'file2.log'
```

If we want to remove the `subdir/` directory from the repository, we would do the following:

```
git rm -r subdir/
rm 'subdir/README.txt'
rm 'subdir/hello_world.cpp'
```

Note that this doesn't remove it from neither the file-tree, nor Git's `history`. It simply produces a commit where the files are removed from being tracked. They will then show up when running `git status` until you have entered them into your `.gitignore` file.

Further examples of usages for the `.gitignore` file is found at:

https://git-scm.com/docs/gitignore

6.3 How to keep track of versions

When doing bioinformatics, it is imperative to at all times be able to reproduce all the steps performed in your analysis. If it isn't possible to replicate your analysis exactly, it is in principle worthless. Science requires the ability for you and others to replicate your findings. This means that you need to be able to trace the exact commands that you have been using in your analysis. Furthermore, it should be possible to use the *exact same versions* of all the software you have been using.

Git is great help for helping keeping track of software versions. If you keep your script in version control at all times, you will know that the version of the code that you used is there, and you could easily keep track of which exact version that you have been using.

6.3.1 Tagging commits

It is often useful to tag particular commits linked to the versions. This makes it easy to at a later point revisit important stages of the script, and you are able to refer to the versions used in the documentation of your analysis.

You can use the `git tag` command to tag your current commit. Providing the tag name using the `--annotate` (short: `-a`) flag and the description using the `--message` (short: `-m`) flag.

```
$ git tag -a v1.0 -m "This version was a major revision of.."
```

You can list all available tags with the `git tag` command.

```
$ git tag
v1.0
```

To get further information about the tag, you can run the `git show` command:

```
$ git show v1.0
tag v1.0
Tagger: Jakob Willforss <jakob.willforss@hotmail.com>
Date:   Thu May 12 13:38:55 2016 +0200

My description!

commit 3ab37147b530336b09913a0afdd1fa910c041149
Merge: cf8a9d5 984f0ac
Author: Jakob <jakob.willforss@hotmail.com>
Date:   Wed May 11 14:11:07 2016 +0200

Merge branch 'master' of https://github.com/Jakob37/GitCourse
```

Command: `git tag -a -m`
Command: `git tag`

Create a new tag for the current commit with target annotation and message. Use the command without any argument to list all current tags.

Figure 6.1: git tag - Tag a commit for easy access

Command: `git show <tag>`

Show more information about target object (for example a tag).

Figure 6.2: git show - Show more information about objects like tags

6.3.2 Checking out tags

There are two ways to check out the tagged version of the file tree:

- Use the `git show` command to get the commit ID, and run `git checkout` for that
- Create and check out a new branch/head based on the tag

Example of checking out tag by creating a new branch.

```
$ git checkout -b version1.0 v1.0
```

6.3.3 Pushing tags to remote

The tags you add are per default only present on the local branch. If you want them to be pushed to the remote and made available for your collaborators (or yourself on GitHub), you need to specify that you want to push them too.

You can push them one and one by using their names. To push many tags at once, the flag `--follow-tags` is recommended.

```
$ git push origin v1.0
Username for 'https://github.com': Jakob37
Password for 'https://Jakob37@github.com':
Counting objects: 1, done.
Writing objects: 100% (1/1), 173 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/Jakob37/GitCourse.git
 * [new tag]         v1.0 -> v1.0
# The following command would push all tags at once
$ git push --follow-tags
... output
```

Tagging of commits are described in further detail at the following page:

https://git-scm.com/book/en/v2/Git-Basics-Tagging

Reasons on why pushing multiple tags using the `--follow-tags` is preferred to simply pushing all tags with the `--tags` flag is discussed on:

http://stackoverflow.com/questions/5195859/push-a-tag-to-a-remote-repository-using-git

6.4 Exercises

6.4.1 Ignoring files with .gitignore

1. Setup a .gitignore file within your repository. Try adding a file to the file tree, and then specifically excluding it in the .gitignore file.

2. Ignore all files with a particular file ending. Then, use the `git rm --cached` command to remove all already tracked files (that are included in the repository) with that particular file ending.

3. Create a directory named ”Data” within your file tree, and create one or more files within it. Setup your repository so that you freely can generate data files within the directory without they being tracked by the repository.

6.4.2 Tagging commits

1. Create a tag for your current commit, and push it to the remote.

2. Continue developing your repository by adding at least another commit.

3. List all your current tags, and investigate your recently created tag using the `git show` command.

4. Check out the state of your files at the tag, before going back to your master.

6.5 Further reading

You have now reached the end of this material. Hopefully, you now have an understanding of the core concepts of Git, and are comfortable enough with Git to be able to set up and run your own repositories.

A good way to gain fluency in Git is to use Git. Start using source control for your software (or text documents), and host it on a service like GitHub or Bitbucket. When you encounter problems, you will be able to find the solution on the internet (or in this material). Hopefully you now have some understanding of what is going on, and will be well prepared to investigate and solve future issues.

If you have some extra time, take a look into the further reading of this chapter (see below). The first article on 'Project structure' is recommended in particular.

Good luck with your future versioning!

6.5.1 Project structure

If you are interested in how you can approach organizing your bioinformatic projects, you are warmly recommended to read the article: *A Quick Guide To Organizing Computational Biology Projects* by William Stafford Noble. Here, different approaches to making the project structure more manageable are discussed. In particular, note the section *The Value of Version Control*.

`http://journals.plos.org/ploscompbiol/article/asset?id=10.1371%2Fjournal.pcbi.1000424.PDF`

6.5.2 Merge-tools

If working in complex projects and you are merging an extensive new feature into your master, you can end up with complex merge conflicts. Merge-tools are software designed to help you manage those and set up the code in the way you want.

One popular merge-tools is `Meld` which provides a nice graphical interface for the merging:

`http://meldmerge.org/`

If you want to stay purely terminal-based (and is comfortable using the `vim`/`vi` editors), then `vimdiff` is an alternative.

`http://www.rosipov.com/blog/use-vimdiff-as-git-mergetool/`