

# Git

## An Introduction to the Version Control System

Oliver Henrich

Email: [oliver.henrich@strath.ac.uk](mailto:oliver.henrich@strath.ac.uk)

Email: [rse-help@lists.strath.ac.uk](mailto:rse-help@lists.strath.ac.uk)

**Strathclyde Research Software Engineers**

**The University of Strathclyde:  
the place of useful learning**

---

## Table of Contents

### 1. Automated Version Control

### 2. Basic Tasks

#### 2.1 Setting Up Git

#### 2.2 Creating A Git Repository

#### 2.3 Tracking Changes

#### 2.4 Exploring History

### 3. Collaborative Software Development

#### 3.1 Creating Remote Repositories on GitHub

#### 3.2 Collaborating On GitHub

#### 3.3 Conflicts

### 4. Further Information

# 1. Automated Version Control

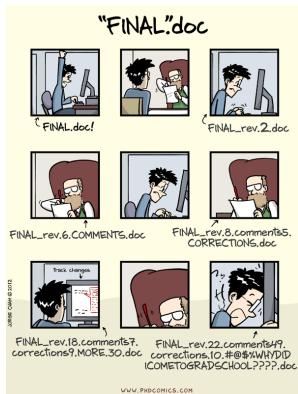
*Why version control?*

## Problems

- ▶ **Multiple, nearly identical versions** of the same document
- ▶ **Tracking changes is not an option** for source code
- ▶ **No protection** against accidental deletion

## Version control systems

- ▶ **start with a base version** of the document
- ▶ **record changes** you make each step of the way
- ▶ **can revert** to any previous versions if necessary
- ▶ **never lose** a previous state of your document
- ▶ allow many people to **work in parallel**



## 1. Automated Version Control

### Sequential changes



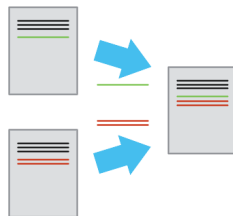
Start at the base document, apply each change, arrive at the more recent version

### Diverging versions



Two users can make independent sets of changes on the same document

### Merge versions



Incorporate two sets of changes into the same base document

## 1. Automated Version Control

---

### Reasons for using Git

- ▶ **Collaboration** as everyone uses Git
- ▶ **Sequence of clean, logical patches**, not uncorrelated random changes
- ▶ Git for **reproducing results**, not only source code
  - ▶ configuration changes
  - ▶ data sets
  - ▶ anything in ASCII
  - ▶ L<sup>A</sup>T<sub>E</sub>X source code
- ▶ Git as starting point for **automated unit and regression tests**, e.g. via Gitlab, GitHub

## 1. Automated Version Control

---

Example: Find bug via bisection

### Manually

1. Define (latest) buggy version **B**
2. Find some working version **W**
3. Check out intermediate version **I** half-way between **B** and **W**, build, run
  - ▶ Working?  $\Rightarrow \mathbf{W} = \mathbf{I}$
  - ▶ Not working?  $\Rightarrow \mathbf{B} = \mathbf{I}$
4. Goto 3
5. Identified buggy version?
  - ▶ Identify change that causes the bug:  
Do `diff` on **all** source code files
  - ▶ **Can take hours, days or weeks** depending on code size and code differences between versions

## 1. Automated Version Control

---

Example: Find bug via bisection

### Automated

1. Start bisect wizzard with `git bisect start`
2. Define buggy version with `git bisect bad someCommitID`
3. Define working version with `git bisect good anotherCommitID`
4. Git checks out intermediate version, build, run
  - ▶ Working?  $\Rightarrow$  `git bisect good`
  - ▶ Not working?  $\Rightarrow$  `git bisect bad`
5. Git goes to 4
6. Identified buggy version?
  - ▶ **Read associated change**
  - ▶ **Takes seconds!**

## 2. Basic Tasks

---

In this section

- ▶ Setting Up Git
- ▶ Creating A Git Repository
- ▶ Tracking Changes
- ▶ Exploring History



## 2.1. Setting Up Git

---

Git comes in many different forms. We use **Git on the command line**.

- ▶ It's the only place you can run **all** Git commands.
- ▶ If you know the command line version, you can figure out how to run the GUI version.
- ▶ Everyone has the same command line tools.

### On Linux

```
$ sudo dnf install git-all (Fedora)
```

```
$ sudo apt install git-all (Debian)
```

### On Mac

```
$ brew install git (Homebrew)
```

```
$ port install git (Macports)
```

part of XCode IDE

### On Windows

Git for Windows: <https://git-scm.com/download/win>

GitHub Desktop: <https://desktop.github.com>

Git Chocolatey: <https://chocolatey.org/packages/git>

## 2.1. Setting Up Git

---

When we use Git for the first time, we need to configure a few things.

Here are a few examples of configurations we will set as we get started with Git:

- ▶ your name and email address
- ▶ and that we want to use these settings globally (i.e. for every project)

On a command line, Git commands are written as **git verb options**, where **verb** is what we actually want to do and **options** is additional optional information which may be needed for the verb.

So here is how Dracula sets up his new laptop:

```
$ git config --global user.name "Vlad Dracula"  
$ git config --global user.email  
    "vlad@tran.sylvan.ia"
```

## 2.1. Setting Up Git

---

Quite helpful and important are the commands to look up the manual

For a general overview of a range of Git commands

```
$ git --help
```

For a overview of specific Git command (here command = verb)

```
$ git verb -h
```

For an in-depth manual of specific Git command

```
$ git verb --help
```

For more information please check the **Git online documentation** at

<https://git-scm.com/docs>

## 2.2. Creating A Git Repository

---

Create a directory in your current working directory

```
$ mkdir git_exercise
```

Change to the new directory

```
$ cd git_exercise
```

Create the Git repository

```
$ git init
```

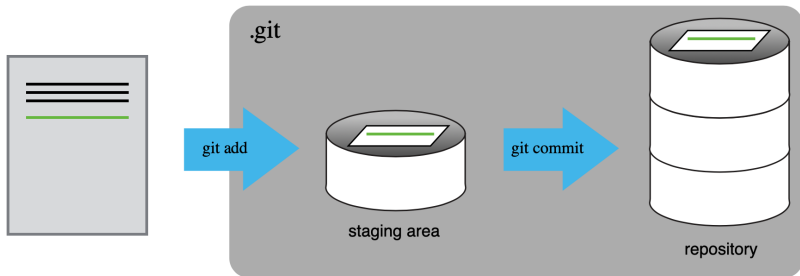
### Note

An invisible file `.git` is created that stores all the history and dependencies of the repository.

## 2.3. Tracking Changes

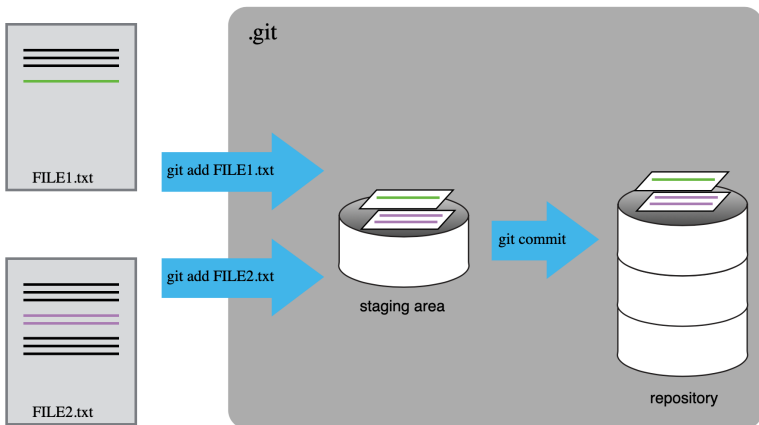
Think of Git as taking snapshots of your project in a two-step process:

- ▶ `git add` specifies **what** will go into a snapshot
- ▶ `git commit` **takes the actual snapshot** and records it permanently



## 2.3. Tracking Changes

It is of course possible to add multiple files (or changes thereof) before committing these, i.e. taking the snapshot.



## 2.3. Tracking Changes

Check the status

```
$ git status
```

Output if working directory / repository is brand new

```
On branch master
No commits yet
nothing to commit (create/copy files and use "git
    add" to track)
```

Now create a new file `hello_world.py` in your working directory that prints 'Hello World!'

## 2.3. Tracking Changes

When you check the status again, you will now see there is an untracked file.

```
$ git status
```

Output if working directory differs, but new file is not yet being tracked

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will  
be committed)  
hello_world.py
```

```
nothing added to commit but untracked files present
```

```
(use "git add" to track)
```



## 2.3. Tracking Changes

Add your new file and start the tracking.

```
$ git add hello_world.py
```

Check the status again

```
$ git status
```

Output if working directory differs and file is being tracked

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file: hello_world.py
```

## 2.3. Tracking Changes

Now commit the change and record it permanently.

```
$ git commit -m 'Added new file'
```

Output of `git commit`

```
[master (root-commit) b3557d4] Added new file
1 file changed, 8 insertions(+)
create mode 100644 hello_world.py
```

## 2.3. Tracking Changes

Check the status with `git status` and you get the following message:

```
On branch master
nothing to commit, working tree clean
```

You can check the log to see your commits. The most recent appears first.

```
$ git log
```

```
commit 249156049502e47d839735c34e31830885bc5092
(HEAD -> master)
Author: Oliver Henrich
<ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:56:07 2020 +0100
    Added new file
```

## 2.4. Exploring History

When working with repos, you often want to **review changes before committing them** or **revert to a previous version** of the file.

Add an additional line to the previous `hello_world.py` file. Check the **differences between your local and remote repository** with

```
$ git diff
```

Additional line `'print('Hello Scotland!')'` in local file (+)

```
diff --git a/hello_world.py b/hello_world.py
index 73fb7c3..e6f9107 100644
--- a/hello_world.py
+++ b/hello_world.py
@@ -1,2 @@
 print('Hello World!')
+print('Hello Scotland!')
```

## 2.4. Exploring History

Commit the change

```
$ git add hello_world.py  
$ git commit -m 'Added additional line'
```

and check the differences again.

```
$ git diff
```

There are no differences anymore as you committed your change.

Now check the status again.

```
$ git status
```

Output of `git status`

```
On branch master  
nothing to commit, working tree clean
```

## 2.4. Exploring History

Add another line, commit the change again and check your commit log.

### Output of `git log`

```
commit 908944eb711c90f5bd46297639b34d8fc70993f0
(HEAD -> master)
Author: Oliver Henrich
<ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:59:00 2020 +0100
    Added another additional line

commit 28f46c36b5729ab26ca719cc1468b1a6e734d597
Author: Oliver Henrich
<ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:58:15 2020 +0100
    Added additional line

commit 249156049502e47d839735c34e31830885bc5092
Author: Oliver Henrich
<ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:56:07 2020 +0100
    Added new file
```

**All commits have a unique ID**, but Git knows a simple way to address them:

The **last commit** appears at the top and is marked with **HEAD**.

The **two previous commits** are not marked, but can be conveniently addressed with **HEAD~1** and **HEAD~2**.

## 2.4. Exploring History

If we want to see what the differences are between the current version (**HEAD**) and version two commits ago, we can issue for instance

```
$ git diff HEAD~2
```

Additional two lines marked as different in local file (+)

```
diff --git a/hello_world.py b/hello_world.py
index 73fb7c3..547a19b 100644
--- a/hello_world.py
+++ b/hello_world.py
@@ -1,3 @@
 print('Hello World!')
+print('Hello Scotland!')
+print('Hello Glasgow!')
```

## 2.4. Exploring History

Assume you want to obtain the previous version without the additional line.  
First you need to **check the log for the ID of the previous commit**.

### Output of `git log`

```
commit 28f46c36b5729ab26ca719cc1468b1a6e734d597
Author: Oliver Henrich
       <ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:58:15 2020 +0100
    Added additional line
```

The commit ID is the long bit  
starting 28f46c36b5...

It is **usually sufficient to  
specify only 7 digits**.

Use the `git checkout` command to retrieve a previous version.

```
$ git checkout 28f46c36b5 hello_world.py
```

**Note: Do not forget the filename at the end as this will 'detach the HEAD'!**

To retrieve the latest version again use

```
$ git checkout master hello_world.py
```



## 2.4. Exploring History

The previous command will not revert the commit (check e.g. `git log`).

To **revert an erroneous commit**, first look for its **ID** and use `git revert`.

### Output of `git log`

```
commit 908944eb711c90f5bd46297639b34d8fc70993f0
(HEAD -> master)
Author: Oliver Henrich
<ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:59:00 2020 +0100
    Added another additional line

commit 28f46c36b5729ab26ca719cc1468b1a6e734d597
Author: Oliver Henrich
<ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:58:15 2020 +0100
    Added additional line
```

We want to revert the commit starting 908944eb71...

We want the current version to be this one.

```
$ git revert 908944eb71
```

This creates a new commit with the previous version of the file.

## 2.4. Exploring History

Revert "Added another additional line"

This reverts commit 908944eb711c90f5bd46297639b34d8fc70993f0.

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#   modified:   hello_world.py
#
```

A dialogue window opens that asks you for a message providing a template.

```
commit 15f36c3bd31f594504756326df6b3baeb2d0982c (HEAD -> master)
Author: Oliver Henrich <ohenrich@users.noreply.github.com>
Date: Thu Sep 3 17:29:03 2020 +0100
    Revert "Added another additional line"
    This reverts commit 908944eb711c90f5bd46297639b34d8fc70993f0.
```

```
commit 908944eb711c90f5bd46297639b34d8fc70993f0
Author: Oliver Henrich <ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:59:00 2020 +0100
    Added another additional line
```

```
commit 28f46c36b5729ab26ca719cc1468b1a6e734d597
Author: Oliver Henrich <ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:58:15 2020 +0100
    Added additional line
```

```
commit 249156049502e47d839735c34e31830885bc5092
Author: Oliver Henrich <ohenrich@users.noreply.github.com>
Date: Wed Sep 2 16:56:07 2020 +0100
    Added new file
```

Your commit log has now an extra entry.

### 3. Collaborative Software Development

---

In this section

- ▶ Creating Remote Repositories on GitHub
- ▶ Collaborating On GitHub
- ▶ Conflicts

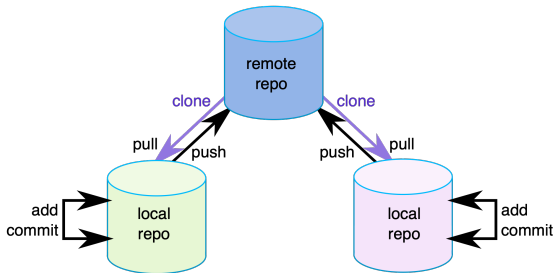
## 3.1. Creating Remote Repositories on GitHub

One of the main reasons for using repositories is also to **collaborate with other people** and **work on the same code**. This is done through a **remote repository**.

***Pulling*** retrieves from the remote repo.

***Pushing*** deposits into the remote repo.

***Cloning*** checks out a **private copy** of the remote repo.

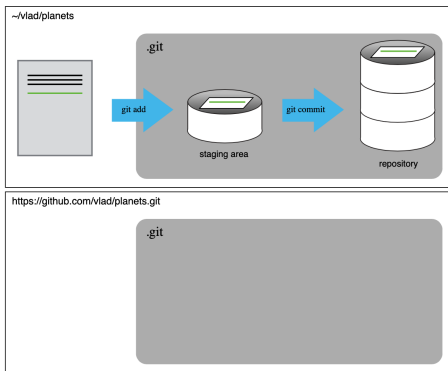


~you/cs31/labs/lab01/

~partner/cs31/labs/lab01/

## 3.1. Creating Remote Repositories on GitHub

Your current situation looks like this



We will now **export your existing local** repo to the **newly created remote** repo on GitHub.

## 3.1. Creating Remote Repositories on GitHub

---

The full documentation is available on <https://docs.github.com/en>, under **GitHub.com** → **Importing your projects** → **Adding an existing project to GitHub using the command line**.

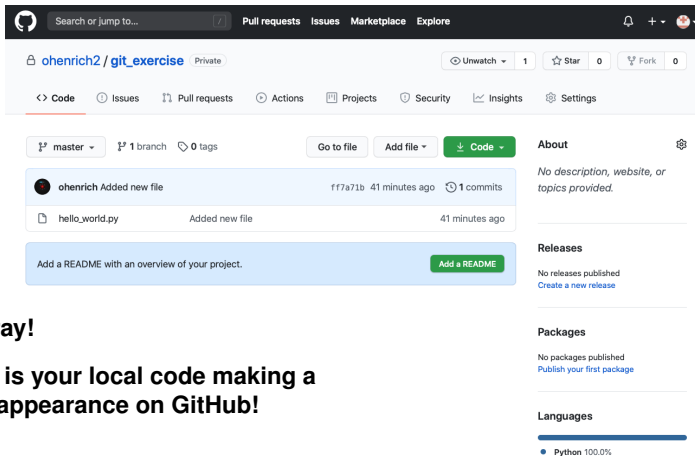
**On GitHub**, look up the URL of your remote repo.

**On the command line** change to the directory of your local repo and issue the following sequence of commands replacing username, etc accordingly.

```
$ cd git_exercise
$ git remote add origin
  https://github.com/username/repo_name.git
$ git branch -M master
$ git push -u origin master
```

## 3.1. Creating Remote Repositories on GitHub

On GitHub check what's in your remote repository.



The screenshot shows the GitHub interface for a repository named 'git\_exercise' by user 'ohenrich2'. The repository is private and has 1 commit, 0 stars, and 0 forks. The 'Code' tab is selected, showing a file named 'hello\_world.py' added 41 minutes ago. A green button 'Add a README' is visible. The right sidebar shows sections for 'About', 'Releases', 'Packages', and 'Languages'. The 'Languages' section shows 'Python' at 100.0%.

Search or jump to... Pull requests Issues Marketplace Explore

ohenrich2 / git\_exercise Private Unwatch 1 Star 0 Fork 0

<> Code Issues Pull requests Actions Projects Security Insights Settings

master 1 branch 0 tags Go to file Add file Code

ohenrich Added new file ff7a71b 41 minutes ago 1 commits

hello\_world.py Added new file 41 minutes ago

Add a README with an overview of your project. Add a README

About No description, website, or topics provided.

Releases No releases published Create a new release

Packages No packages published Publish your first package

Languages Python 100.0%

**Hooray!**

**Here is your local code making a first appearance on GitHub!**

## 3.2. Collaborating On GitHub

---

In [Section 3.1](#) we **exported** an existing *local repository* to your GitHub profile to **create a remote repository**.

The inverse process of **importing** an existing *remote repository* from GitHub is called **cloning**.

**Cloning** produces a **local copy of the remote repository** on your machine. It requires

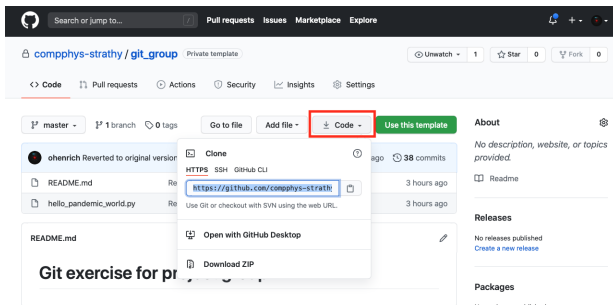
- ▶ the **URL of the remote repository**
- ▶ the `git clone` command

You can modify the local copy and **push the changes to the remote repository** on GitHub to share them with your collaborators.



## 3.2. Collaborating On GitHub

First navigate to the repository that you want to clone and click on the "Code" button



Copy the URL, e.g. by pressing the button next to it.

## 3.2. Collaborating On GitHub

On your command line in your working directory issue the following command replacing **URL** with the actual URL:

```
$ git clone URL
```

### Output of `git clone URL`

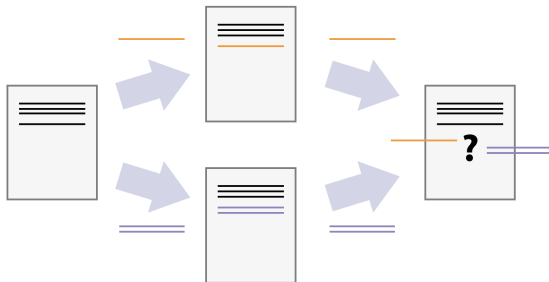
```
Cloning into 'git_group'...
remote: Enumerating objects: 109, done.
remote: Counting objects: 100% (109/109), done.
remote: Compressing objects: 100% (79/79), done.
remote: Total 109 (delta 33), reused 96 (delta 27), pack-reused 0
Receiving objects: 100% (109/109), 14.32 KiB | 4.77 MiB/s, done.
Resolving deltas: 100% (33/33), done.
```

You can clone the repository it into a different name than the default name (the name on GitHub), e.g. **blablabla** by adding this after the URL.

```
$ git clone URL blablabla
```

### 3.3. Conflicts

**Conflicts** emerge when **several** people work on the **same** code and **update** the **remote repo**.



The **orange line** and the **purple line** are approximately at the same position in the file.

### 3.3. Conflicts

Let's look at our Hello World! example.

Your collaborator has checked in the following version:

```
print('Hello World!')
print('Hello Scotland!')
print('Hello City of
      Glasgow!')
```

Your version differs in the last line:

```
print('Hello World!')
print('Hello Scotland!')
print('Hello Greater
      Glasgow!')
```

#### Output of `git push origin master`

```
To https://github.com/compphys-strathy/git_exercise.git
! [rejected] master -> master (fetch first)
error: failed to push some refs to 'https://github.com/compphys-strathy/git_exercise.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

### 3.3. Conflicts

What we have to do is **pull the changes from the remote repo** on GitHub into your local repo.

Git **tries to merge them automatically** into your local copy, and **if successful this can be pushed** to the remote repo on GitHub.

```
$ git pull origin master
```

Output of `git pull origin master`

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 682 bytes | 682.00 KiB/s, done.
From https://github.com/compphys-strathclyde/git_exercise
 * branch      master    -> FETCH_HEAD
   dbd3016..5b9c121 master -> origin/master
Auto-merging hello_world.py
CONFLICT (content): Merge conflict in hello_world.py
Automatic merge failed; fix conflicts and then commit the result.
```

### 3.3. Conflicts

Check what Git has done to your local file.

```
print('Hello World!')
print('Hello Scotland!')
<<<<<<< HEAD
print('Hello Greater Glasgow!')
=====
print('Hello City of Glasgow!')
>>>>>>> 5b9c121bac
```

Our change is preceded by  
**<<<<<<< HEAD.**

Git inserted **=====** as  
separator between the  
conflicting changes.

The end of the content  
downloaded from GitHub is  
marked with **>>>>>>>.**

We need to **remove** these markers, **reconcile** the changes and **check in a new version.**

### 3.3. Conflicts

```
print('Hello World!')  
print('Hello Scotland!')  
print('Hello Greater Glasgow!')  
print('Hello City of Glasgow!')
```

We remove the markers and keep both lines.

Lets' first check the status.

#### Output of `git status` after editing

```
git status  
On branch master  
Your branch and 'origin/master' have diverged,  
and have 2 and 1 different commits each, respectively.  
(use "git pull" to merge the remote branch into yours)  
  
You have unmerged paths.  
(fix conflicts and run "git commit")  
(use "git merge --abort" to abort the merge)  
  
Unmerged paths:  
(use "git add <file>..." to mark resolution)  
  both modified: hello_world.py
```

We are using the last option.

### 3.3. Conflicts

```
$ git add hello_world.py  
$ git commit -m 'Resolved conflict'
```

Output of `git commit`

```
[master c7c8fb8] Resolved conflict
```

```
$ git push origin master
```

**Minimise the number of conflicts by using this workflow:**

1. Update local `git pull origin master`
2. Make changes
3. Stage changes `git add your_edited_file.py`
4. Commit changes `git commit -m "Your commit message"`
5. Update remote `git push origin master`



## 4. Further Information

---

- ▶ Excellent **Git tutorial materials** are available on the Software Carpentry website:

<http://swcarpentry.github.io/git-novice>



- ▶ **Git Tutorial** by Robert Atkey (CIS):  
<https://gitlab.cis.strath.ac.uk/jjb15109/git-tutorial>
- ▶ **OhMy Git! Game** – learning the playful way: <https://ohmygit.org>
- ▶ Join us at our **monthly Hacky Hour every first Wednesday 2-3PM**:  
Zoom meeting room: 957-329-701  
Password: HackyHour
- ▶ Subscribe to **rse-announce mailing list** for regular information:  
<http://lists.strath.ac.uk/mailman/listinfo/rse-announce>