

@penCHORD_UoE
@peninsula_ARC



Module 2 : Open Source Collaborative Development

Session 2B : Principles of FOSS and Version Control

Dr Daniel Chalk

“The Git Crowd”



#hsma5isalive

With Thanks

Thanks to my colleague Mike Allen for providing much of the content and ideas for the “Why FOSS?” section

What is FOSS?

Free and Open Source Software (FOSS) refers to software that :

- can be used and changed by anyone for whatever purpose they need (Free)*
- can be interrogated by anyone to see how it works (Open Source)

FOSS Software differs from *Proprietary* software, where the software is distributed under copyright licensing, and the source code is typically hidden.

*It is possible to have Open Source software that is not Free.

History of FOSS

There is sometimes a misconception that FOSS is a relatively modern idea. But this is a misconception, brought about (arguably) because of the dominance of proprietary computer software over the first few decades of the home computer boom.

Particularly from one company. Mentioning no names.

<<cough>> Microsoft.. <<cough>>



The first non-trivial computer program developed in 1843 by Ada Lovelace for Charles Babbage's Analytical Engine.
And it was open and shared :)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data										Working Variables.										Result Variables.											
						V ₁	V ₂	V ₃	v ₄	v ₅	v ₆	v ₇	v ₈	v ₉	v ₁₀	v ₁₁	V ₁₂	V ₁₃	V ₁₄	V ₁₅	V ₁₆	V ₁₇	V ₁₈	V ₁₉	V ₂₀	B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇					
1	$\times V_2 \times V_3$	V_4, V_5, V_6	$V_2 = V_2$	$= 2n$		1	2	n	2n	2n	2n																										
2	$- V_4 - V_1$	V_4	$V_4 = V_4$	$= 2n - 1$		1																															
3	$+ V_5 + V_1$	V_5	$V_5 = V_5$	$= 2n + 1$		1																															
4	$+ V_6 + V_4$	V_4	$V_6 = V_6$	$= 2n + 1$		1																															
5	$+ V_{11} - V_2$	V_{11}	$V_{11} = V_{11}$	$= 1 - 2n - 1$		2																															
6	$- V_{13} - V_1$	V_{13}	$V_{13} = V_{13}$	$= -\frac{1}{2} \cdot 2n - 1 = A_0$		1																															
7	$- V_3 - V_1$	V_3	$V_3 = V_3$	$= n - 1 (= 3)$		1		n																													
8	$+ V_2 + v_7$	V_7	$V_2 = V_2$	$= 2 + 0 = 2$		2																															
9	$+ V_6 + V_{11}$	V_{11}	$V_6 = V_6$	$= 2n$																																	
10	$\times V_{12} \times V_{13}$	V_{12}, V_{13}	$V_{12} = V_{12}$	$= B_1 \cdot \frac{2}{3} = B_1 A_1$																																	
11	$+ V_{12} + V_{13}$	V_{12}, V_{13}	$V_{12} = V_{12}$	$= -\frac{1}{2} \cdot 2n - 1 + B_1 \cdot \frac{2n}{3}$																																	
12	$- V_{10} - V_1$	V_{10}	$V_{10} = V_{10}$	$= n - 2 (= 2)$		1																															
13	$- V_6 - V_1$	V_6	$V_6 = V_6$	$= 2n - 1$		1																															
14	$+ V_1 + V_2$	V_2	$V_1 = V_1$	$= 2 + 1 = 3$		1																															
15	$- 2V_6 + V_7$	V_7	$V_7 = V_7$	$= \frac{n-1}{3}$																																	
16	$\times V_8 \times V_{11}$	V_8, V_{11}	$V_8 = V_8$	$= 2n - 1$																																	
17	$- V_6 - V_1$	V_6	$V_6 = V_6$	$= 2n - 2$		1																															
18	$+ V_1 + V_2$	V_2	$V_1 = V_1$	$= 3 + 1 = 4$		1																															
19	$- 2V_6 + V_5$	V_5	$V_5 = V_5$	$= 2 - 2$																																	
20	$\times V_9 \times V_{11}$	V_9, V_{11}	$V_9 = V_9$	$= 2n - 2 - 1$																																	
21	$\times V_{22} \times V_{12}$	V_{22}, V_{12}	$V_{22} = V_{22}$	$= B_2 \cdot \frac{2}{3} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = B_2 A_2$																																	
22	$+ V_{12} + V_{13}$	V_{12}, V_{13}	$V_{12} = V_{12}$	$= A_0 + B_1 A_1 + B_2 A_2$																																	
23	$- V_{10} - V_1$	V_{10}	$V_1 = V_1$	$= n - 3 (= 1)$		1																															
24	$+ V_{13} + V_{20}$	V_{24}	$V_2 = V_2$	$= B_7$																																	
25	$+ V_1 + V_3$	V_3	$V_3 = V_3$	$= n + 1 = 4 + 1 = 5$		1		n + 1				0	0																								

Here follows a repetition of Operations thirteen to twenty-three.

Making Money from Software

In the early days of computing, software was often bundled with hardware. So, people (or, in reality in the early days, companies and later hobbyists) were buying computers (that made the money) that came with the software they needed (for free).

In the late 60s / early 70s, a new “software market” started to emerge. Software was increasingly seen as standalone and separate. But if you need to make money, and you’re not selling hardware, you need to start charging...

Bill Gates writes to fellow nerds in 1977

AN OPEN LETTER TO HOBBYISTS

To me, the most critical thing in the hobby market right now is the lack of good software courses, books and software itself. Without good software and an owner who understands programming, a hobby computer is wasted. Will quality software be written for the hobby market?

Almost a year ago, Paul Allen and myself, expecting the hobby market to expand, hired Monte Davidoff and developed Altair BASIC. Though the initial work took only two months, the three of us have spent most of the last year documenting, improving, and adding features to BASIC. Now we have 4K, 8K, EXTENDED, ROM and DISK BASIC. The value of the computer time we have used exceeds \$40,000.

The feedback we have gotten from the hundreds of people who say they are using BASIC has all been positive. Two surprising things are apparent, however. 1) Most of these "users" never bought BASIC (less than 10% of all Altair owners have bought BASIC), and 2) The amount of royalties we have received from sales to hobbyists makes the time spent on Altair BASIC worth less than \$2 an hour.

Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid?

Is this fair? One thing you don't do by stealing software is get back at MITS for some problem you may have had. MITS doesn't make money selling software. The royalty paid to us, the manual, the tape and the overhead make it a break-even operation. One thing you do do is prevent good software from being written. Who can afford to do professional work for nothing? What hobbyist can put 3 man-years into programming, finding all bugs, documenting his product and distribute for free? The fact is, no one besides us has invested a lot of money in hobby software. We have written 6800 BASIC, and are writing 8080 APL and 6800 APL, but there is very little incentive to make this software available to hobbyists. Most directly, the thing you do is theft.

What about the guys who re-sell Altair BASIC, aren't they making money on hobby software? Yes, but those who have been reported to us may lose in the end. They are the ones who give hobbyists a bad name, and should be kicked out of any club meeting they show up at.

I would appreciate letters from anyone who wants to pay up, or has a suggestion or comment. Just write me at 1180 Alvarado SE, #114, Albuquerque, New Mexico, 87108. Nothing would please me more than being able to hire ten programmers and deluge the hobby market with good software.

Bill Gates
General Partner,
Micro-Soft

Most of

these "users" never bought BASIC
(less than 10% of all Altair owners
have bought BASIC)

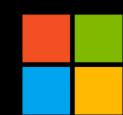
Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid?

I would appreciate letters from anyone who wants to pay up

Locking Down

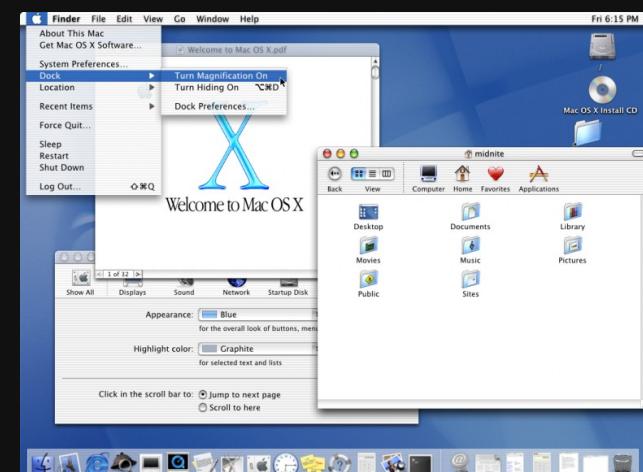
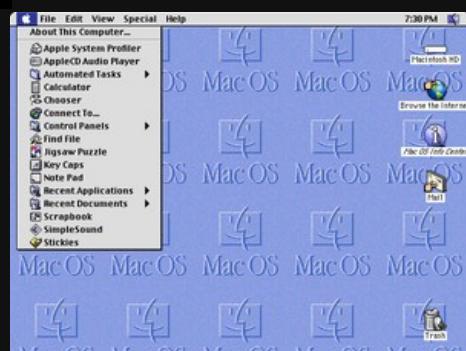
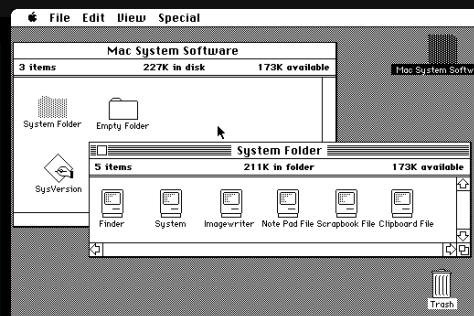
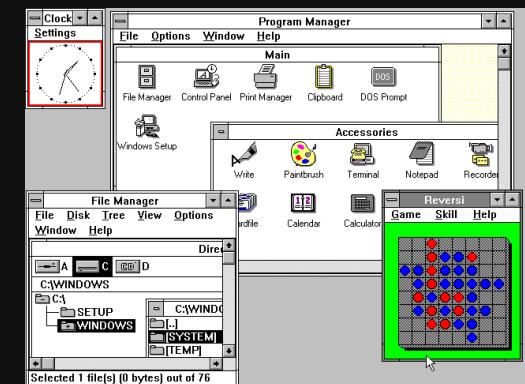
In the 70s and 80s, software companies began “hiding” the code for their software – “Closed Source Software”.

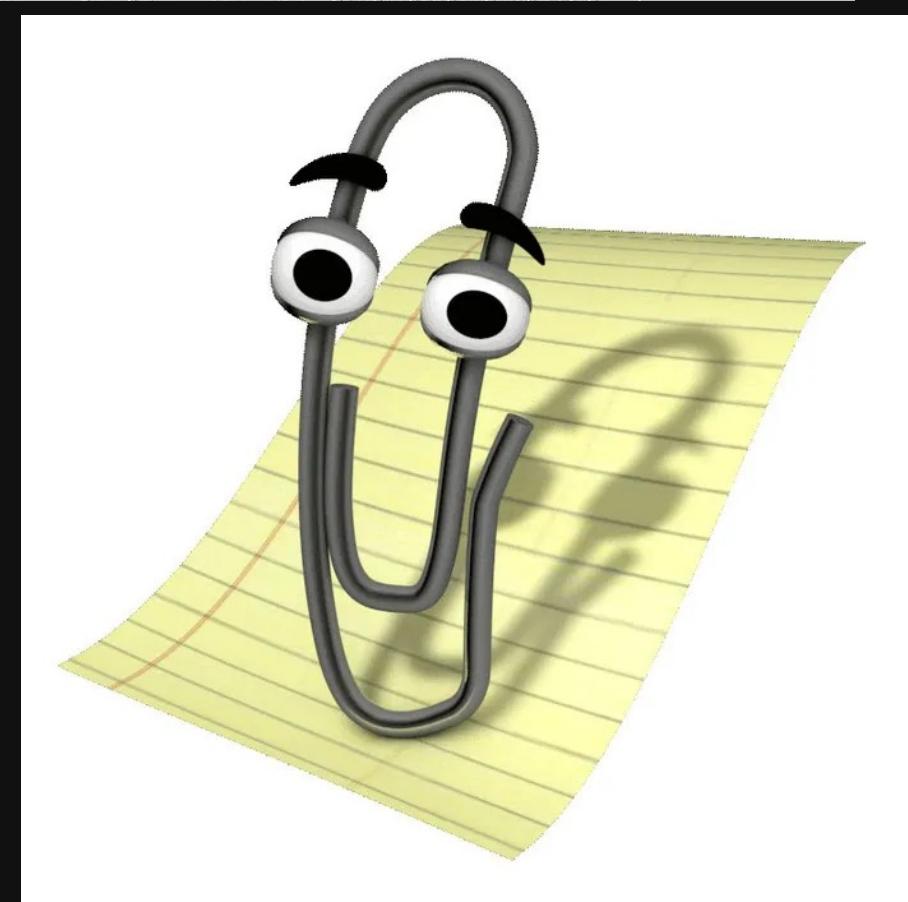
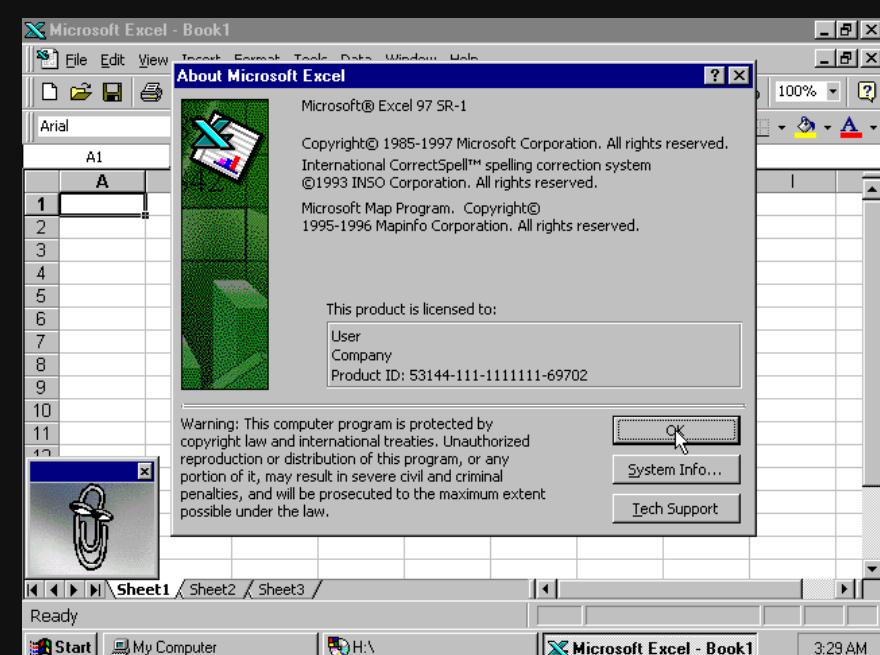
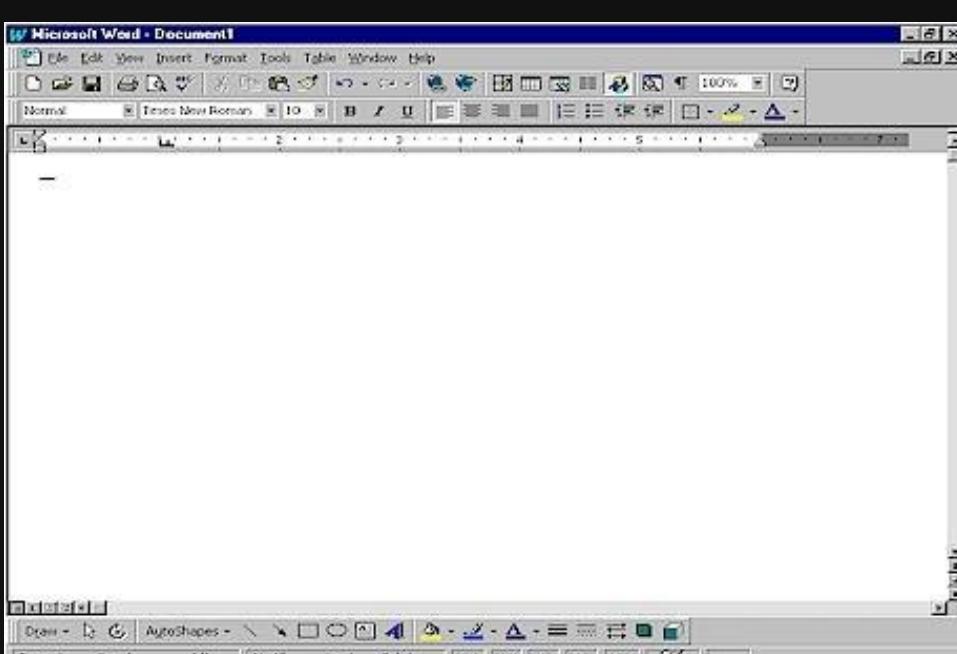
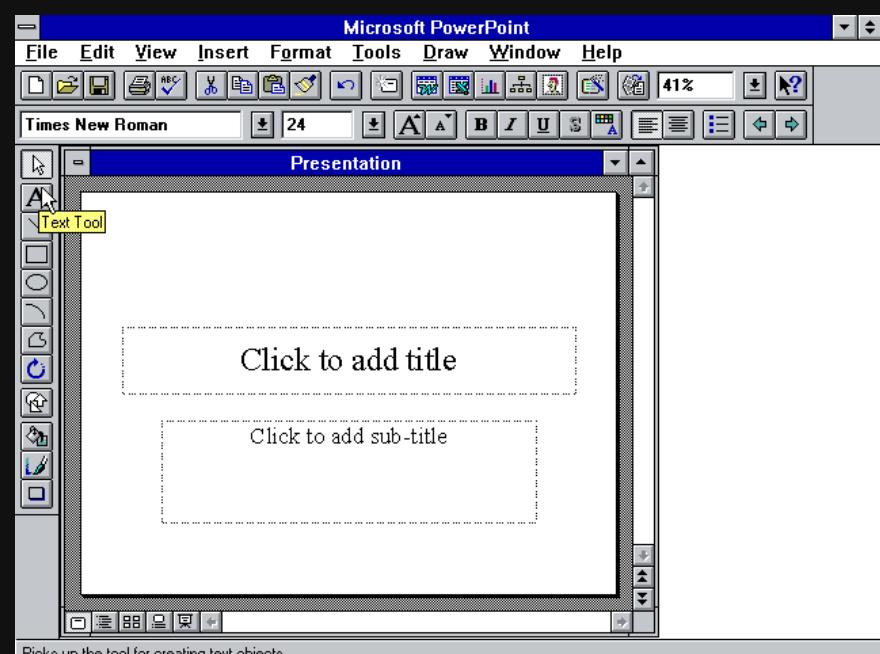
And in 1980, Copyright Law was extended in the US to cover software (which had previously been considered to be not copyrightable because they were ideas / methods / processes).



Starting MS-DOS . . .

C:\>_





Meanwhile, whilst Microsoft (and, to a lesser extent initially, Apple) were taking over the world, the idea that software should be open hadn't gone away...

A close-up portrait of Richard Stallman, a man with long, curly brown hair and a full, grey beard. He is wearing glasses and a red, yellow, and blue patterned shirt. He is holding a black microphone in his right hand, which has a tattoo on the wrist. He is looking slightly to the right of the camera with a slight smile.

1983

Richard Stallman
(member of the
hacking community
at MIT) announces
the GNU Project

(GNU is Not Unix)

The goal of the GNU Project was to give computer users freedom and control in how they used their computers, allowing them to run, copy, distribute, study and modify software code.

Stallman's Four Freedoms

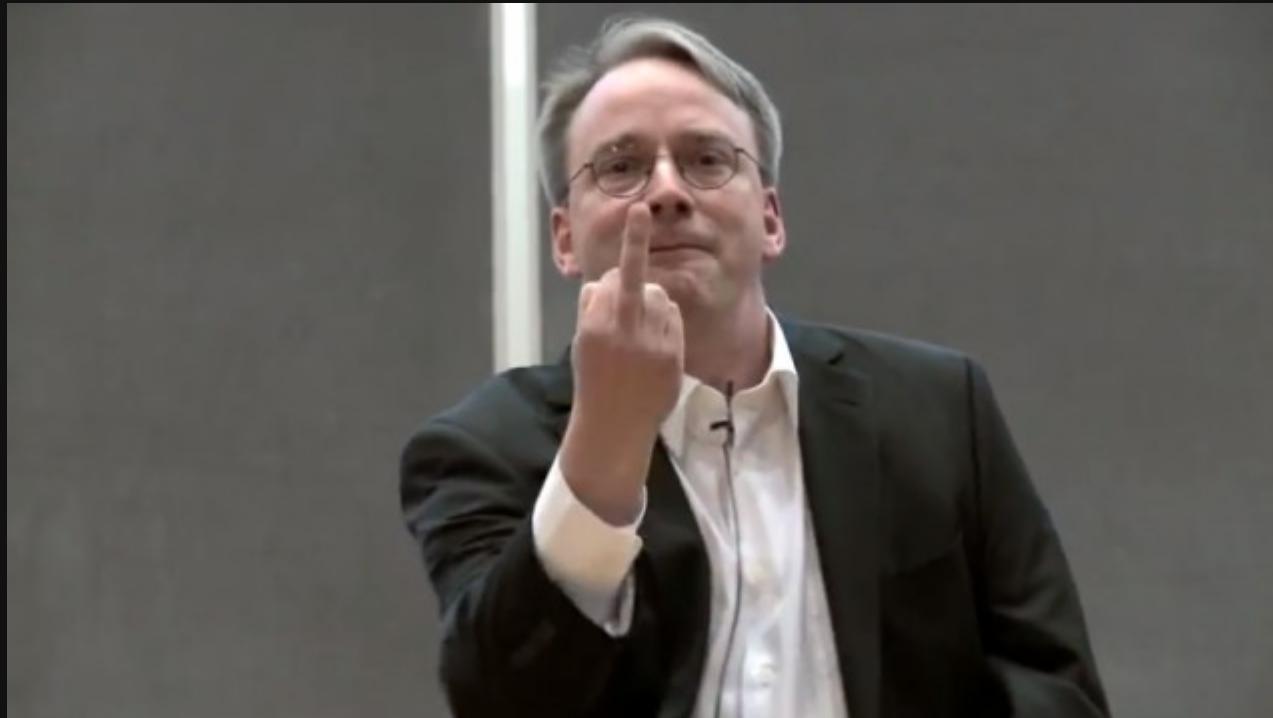
- Freedom to see source code and use in any way
- Freedom to adapt for own use
- Freedom to share original software
- Freedom to share adapted software

The GNU Project needed an Operating System (OS) that was free and open in the same way. They created an OS called GNU that was based on Unix (a Proprietary OS) but with alterations (remember, **GNU** is **Not Unix**)

But GNU lacked one vital element,
preventing it from becoming a fully
functioning Operating System...

...the ‘kernel’ (the core of an OS that
controls all the hardware and
software interactions)

In steps Finnish-American software engineer (and not particularly sociable*) Linus Torvalds



*as an example, he once responded to developers who had created security software that required an admin password to make changes, calling the idea “mentally diseased”, “moronic” and suggested to the developers “just kill yourself now. The world will be a better place”

But ignoring his character “quirks”, in 1991 he did something rather wonderful.

He created the *Linux* kernel, with freely modifiable source code.

In 1992, he licensed the project under the GNU General Public License.

And the first truly Free and Open Source OS was born.

(He also created Git, which you'll learn about later)



And slowly developers start to write
Free and Open Source software for
GNU/Linux

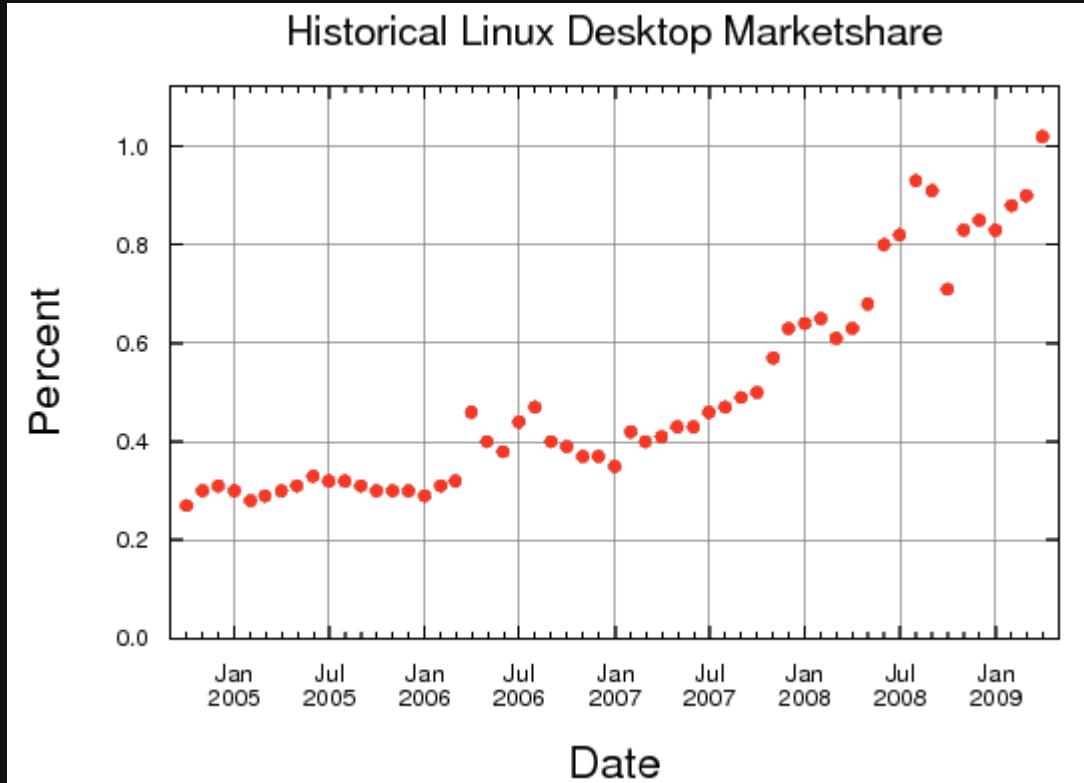
Not everyone at Microsoft was fan

‘They (Sun, Oracle) of course are civilised
competitors - but the Linux crowd are
communists’

(Steve Balmer, CEO Microsoft, 2000)

‘Linux is a cancer’

(Steve Balmer, not feeling any kinder, 2001)



Linux Market Share Passes 1%
in 2009!

(currently at 2.45% at the time
of writing)

There's a long way to go. But there are signs of
change...

Satya Nadella, current CEO Microsoft



Windows 10 Linux Subsystem Setup



Windows Subsystem for Linux – run Linux programs natively in Windows!

2016: GitHub (main FOSS repository)
announced that Microsoft were the single
largest contributor to FOSS for the year

2017: Microsoft announce that their cloud services ('Azure') will now also run on Linux, and will adopt FOSS SQL database systems

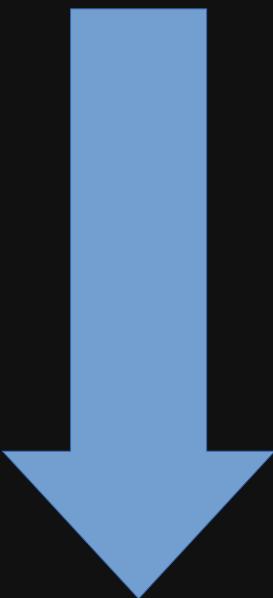
2018: Microsoft switch from Windows to
Linux for new ‘Internet of Things’ (IoT)
operating system

2018: Microsoft breaks up Windows division

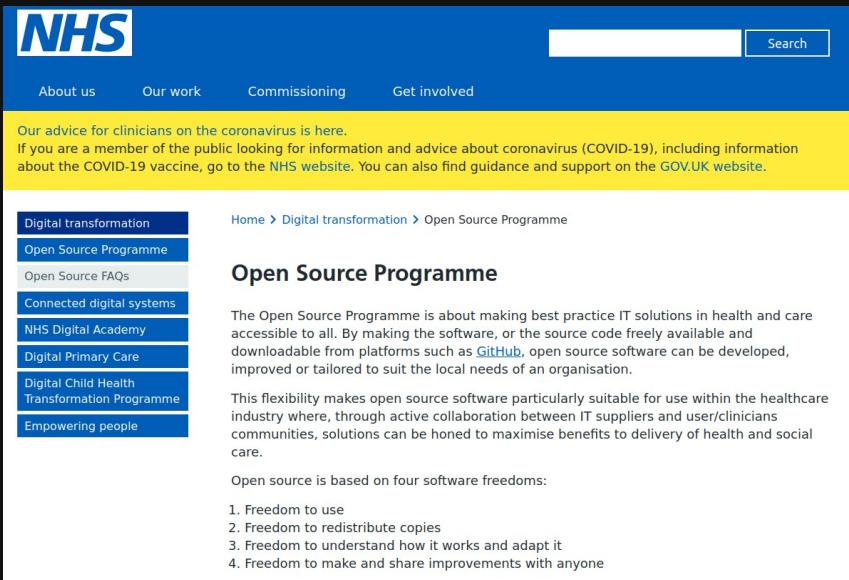


2019 : Dr Katie Bouman and team's algorithm CHIRP reconstructs first ever image of black hole. All using FOSS (including MatPlotLib for the image, which we covered earlier in the course :))

PenCHORD FOSS History



And things are changing in the NHS too...



The screenshot shows the NHS England Open Source Programme website. The header includes the NHS logo, a search bar, and navigation links for About us, Our work, Commissioning, and Get involved. A yellow banner at the top provides advice for clinicians on the coronavirus. The main content area shows the 'Open Source Programme' page, which discusses the benefits of open source software in healthcare and lists four software freedoms. A sidebar on the left contains links to various digital transformation initiatives.

Open Source Programme

The Open Source Programme is about making best practice IT solutions in health and care accessible to all. By making the software, or the source code freely available and downloadable from platforms such as [GitHub](#), open source software can be developed, improved or tailored to suit the local needs of an organisation.

This flexibility makes open source software particularly suitable for use within the healthcare industry where, through active collaboration between IT suppliers and user/clinicians communities, solutions can be honed to maximise benefits to delivery of health and social care.

Open source is based on four software freedoms:

1. Freedom to use
2. Freedom to redistribute copies
3. Freedom to understand how it works and adapt it
4. Freedom to make and share improvements with anyone

What does it mean for NHSX to be an 'open source' organisation?

Terence Eden, 23 April 2019 - [Collaboration](#), [Digital services](#), [Skills and culture](#)



NHSX is an open and transparent organisation. We believe that the people who employ us – the public – have a right to see the code we create. They have the right to understand the algorithms we use. They should be able to examine the code for flaws, and be able to suggest improvements.

NHSX uses open-source technology. The NHS benefits from free access to technology and the ability to swap between multiple suppliers.

What is 'open source'?

NHS England Open Source Programme

<https://bit.ly/3fLNeTl>

<https://bit.ly/3g8I59B>

“The public – have a right to see the code we create. They have a right to understand the algorithms we use. They should be able to examine the code for flaws, and be able to suggest improvements.”

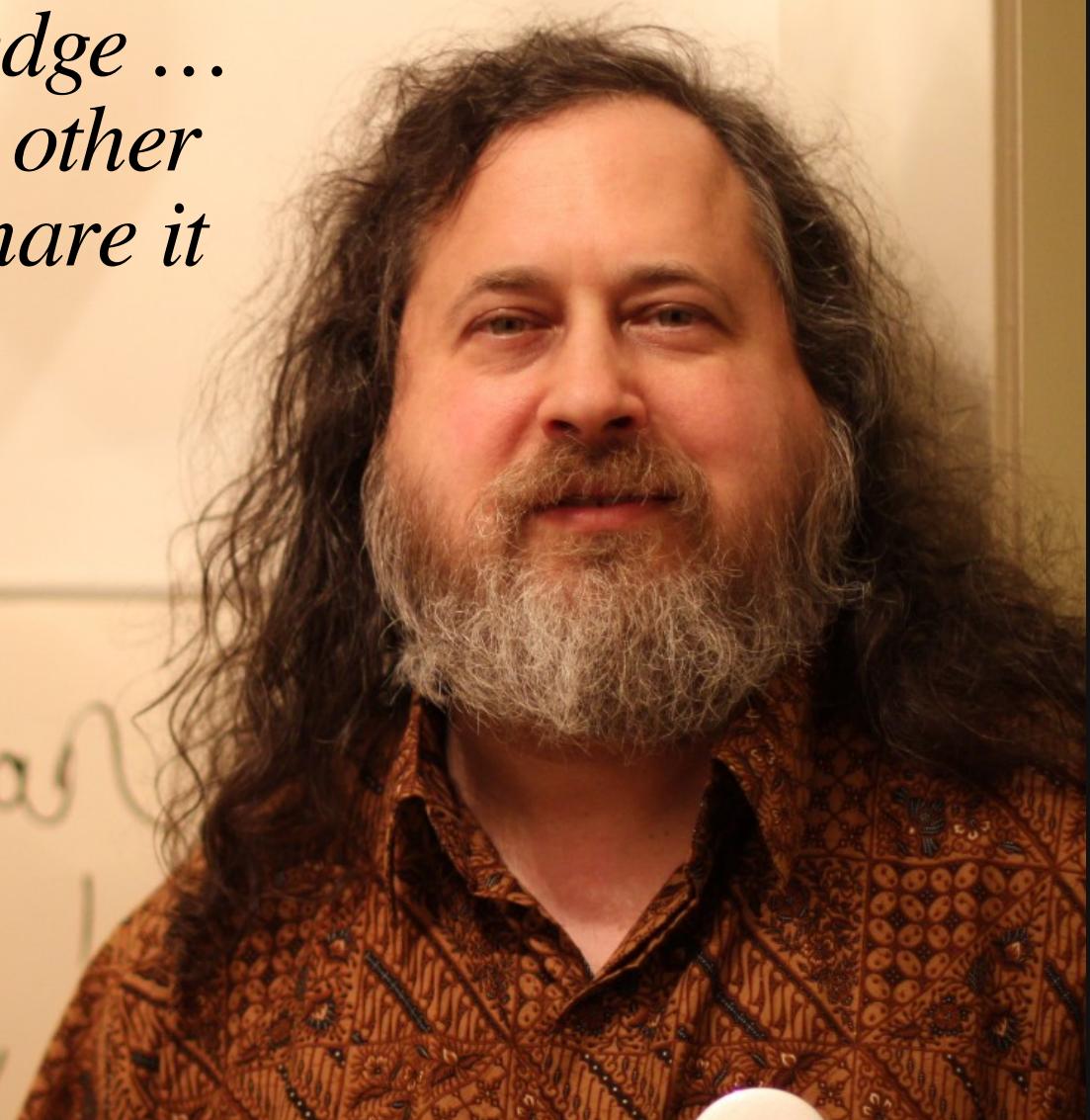
As HSMAs, you will be building models and algorithms that impact the lives of people, possibly significantly.

They have a right to be able to interrogate why your models / algorithms came to the conclusions they did, and question the assumptions and simplifications you made.

And, to spread good practice, you should be sharing your work to allow others to benefit.

*Be generous with knowledge ...
share it freely, and give other
people the freedom to share it
freely*

(Richard Stallman)





The Free Software Song

Richard Stallman

Join us now and share the software;
You'll be free, hackers, you'll be free.
Join us now and share the software;
You'll be free, hackers, you'll be free.

Hoarders can get piles of money,
That is true, hackers, that is true.
But they cannot help their neighbors;
That's not good, hackers, that's not good.

When we have enough free software
At our call, hackers, at our call,
We'll kick out those dirty licenses
Ever more, hackers, ever more.

Join us now and share the software;
You'll be free, hackers, you'll be free.
Join us now and share the software;
You'll be free, hackers, you'll be free.



Exercise 1

In your groups, I now want you to discuss the following :

- Can you think of current or historic examples where the use of proprietary software / closed approaches has limited collaboration and sharing in your organisation and with other organisations?
- What do you think are some key areas where a movement towards a FOSS ethos in your organisation could lead to real benefit?
- How would you like to see your organisation embrace FOSS in the future? What are some of the challenges you envisage, and how might they be overcome?

You have 30 minutes (+ 10 minute comfort break)

Git and GitHub

GitHub is the world's largest online repository hosting open source code projects (a subsidiary of Microsoft since 2018, fact fans!)

GitHub allows users to collaborate on projects and share modifications.

GitHub has, at its core, Git functionality, which allows for version control.

So let's talk about version control and Git. With thanks to my colleague Tom Monks for his excellent Git training materials upon which this is based.

Version Control

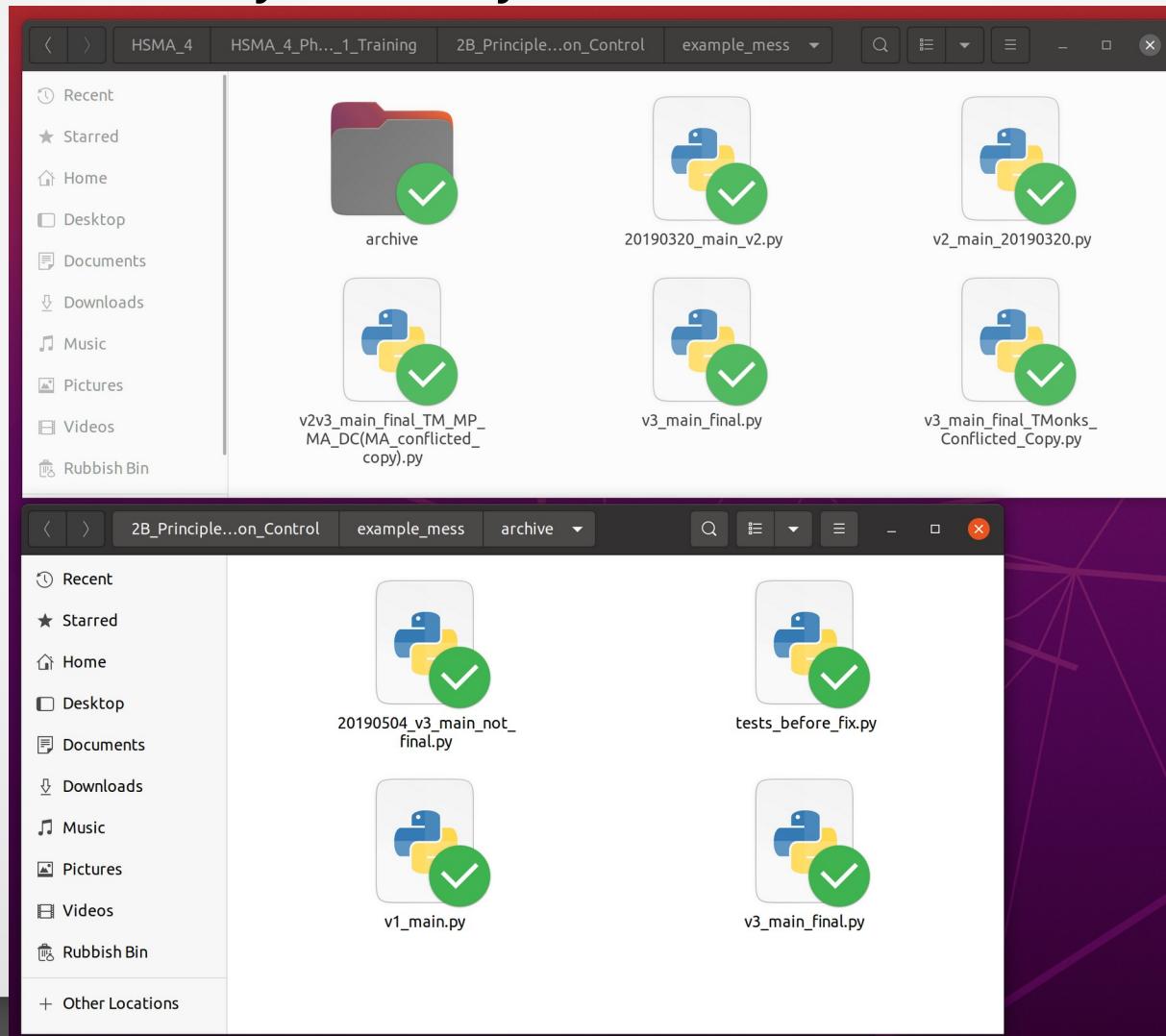
Version Control refers to the tracking and managing of changes to code.

Why is this important? Let's watch a short educational video, in which we imagine the code we've just written is a new car.



Version Control

You've been asked by your organisation to take over a project undertaken by one of last year's HSMAAs. Here are the files you've been given. Which file do you think you need to use?



Version Control

You have been given some code by an analyst who long since left the organisation. You can see from the comments in the code that the code has been modified by multiple people over many years, but not how many times, by who, what they did, or for what reason.

It's your job to fix it.

Where do you start? The reality is, you'd likely need to go over the *whole code* to work out what's happening, and why it's not working as expected.

Version Control

The previous examples show us that we need something to :

- *track* what changes have been made, when, and by whom
- *manage* the versions we have so we always know which is the latest version, and what the previous or alternative versions represented in terms of their contributed changes, and we can switch between version as needed.

Step in Git...

Git

Git is a word that quite often describes me.

Git is also a *Distributed Version Control* system that is widely used in the development community.

Distributed means that the complete *codebase* (all the code and its history) is mirrored on the computer of every developer on the project (rather than just being centrally held). This means that we *merge* when things come back together.

Commit to Git

commit

stage

example.py

Repository

A complete history of all the files in the project, and the changes that were made

Finally, we *commit* the changes to the repository. This creates a snapshot version of the code, along with the log of changes made (which was created as part of the *stage* process).

We then *stage* the file. This basically means that we tell Git we want our file (or rather the changes to the file) to be included in the next *commit* to the repository.

We make the changes to our file and save it.

Change Logs and diff

A commit represents a *snapshot* of your file (or group of files) at a point in time.

Git tracks the changes in each file between commits, in terms of the things we added and the things we removed.

This allows us to easily see what changes were made, when and by whom. It also means that if something went wrong after a change was made, we can see the point to which we need to revert back.

We can view the changes between specific commits – this is known as the *difference* or **diff**.

diff Example

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 12 09:51:57 2021

@author: dan
"""

my_list = [1,2,3,4,5,6,7,8,9,10]

for num in my_list:
    if num % 2 == 0:
        print (f"{num} is even")
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 12 09:51:57 2021

@author: dan
"""

import random

my_list = [1,2,3,4,5,6,7,8,9,10]

for num in my_list:
    if num % 2 == 0:
        sampled_num = random.randint(1,10)
        result = num * sampled_num

        print ("An even number detected!")
        print (f"{num} * {sampled_num} = {result}")
```

```
diff --git a/num_fun.py b/num_fun.py
index 76089a4..eeb5373 100755
--- a/num_fun.py
+++ b/num_fun.py
@@ -6,9 +6,15 @@ Created on Thu Aug 12 09:51:57 2021
@author: dan
"""

+import random
+
my_list = [1,2,3,4,5,6,7,8,9,10]

for num in my_list:
    if num % 2 == 0:
-        print (f"{num} is even")
+        sampled_num = random.randint(1,10)
+        result = num * sampled_num
+
+        print ("An even number detected!")
+        print (f"{num} * {sampled_num} = {result}")
```

Setting up Git and the Git Repository

Let's now walk through a simple example. If you're using Windows, you access the Git command line by opening *Git Bash*. If you're using MacOS or Linux, you simply open a new Terminal (assuming Git is installed). We interact with Git by issuing commands in the shell / terminal.

If this is the first time you've used Git, the first thing you will have to do is specify the username and email address you want to use. It is recommended you use the same username and email that you will use (or have used) for GitHub.

```
git config --global user.name "dan_is_awesome"  
git config --global user.email "dan@danmail.com"
```

It's git, init?

Now we need to set up our Git repository. We do this by navigating to the directory that stores (or is going to store) our code for our project, and then we call the command :

`git init`

This will create a new Git repository based on the files in that directory. If it's a new (empty) directory, it'll create a new empty repository. If there are files already there, it'll note they're there, but won't begin tracking them for future changes until we tell it to do so.

Note – to switch to the directory you want (or create a new one) you can use the following terminal commands :

`ls`
`dir`

Either of these list the contents of the current directory

`cd test`

Changes into the directory called “test” (if it's in the current directory)

`cd ..`

Changes into the parent directory of the one you're in (ie go one level up)

`mkdir test`

Creates a new directory called “test” in the current directory

Add some code

Let's add a .py file to our directory. I'll use Spyder here and create a very simple program. I'll save it as my_calc.py in my new directory git_example.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""

Created on Thu Aug 12 11:46:21 2021

@author: dan
"""

def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result
```

Git status

We can see the status of our repository (in terms of the files that have been modified, ones that are ready for commit etc) by using the command :

git status

```
on branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    my_calc.py
nothing added to commit but untracked files present (use "git add" to track)
```

The name of the branch we're on
(we'll come back to that later)

This tells us that we haven't
made any commits to the
repository yet

The list of files that are in the folder that aren't yet being tracked, because they haven't been staged. Git won't track any changes until they've been staged at least once – then it'll track any changes from that point forward

Staging the file

Let's now tell Git we want this file to be included in the next commit (and also to start tracking changes to this file). We do this using the command *git add* and the name of the file we want to stage :

```
git add my_calc.py
```

We can also use :

```
git add .
```

 to add all files in the directory

```
git add *.py
```

 to add all files of a given type in the directory (.py files in this example)

Let's call *git status* to look at the status again now (it is good practice to keep checking status before and after stages and commits) :

```
On branch master
```

```
No commits yet
```

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

We can see that my_calc is now staged, and ready to be committed

Commit Message

We're now ready to *commit* our file to the repository. To do this, we need to supply a *commit message*. This is a short description of what the commit represents (ie what you've added / changed etc).

Commit messages should be *concise*, but also *specific*, so that someone else (or you, later down the road) can understand what this “snapshot” represents.

Tip (thanks Tom) : you might consider splitting your commit message into two parts – 1) the type of commit you’re making (eg an initial upload of a file, a bug fix, upload of some documentation etc) and 2) the description of changes.

e.g “INIT: add my_calc.py” clearly tells us that this is an initial upload of a file, and the file that was uploaded was my_calc.py. Feel free to use your own style, but make sure it is *consistent* and that someone else could understand it!

Warning!

If you forget to add a commit message, Git will take you into a text editor to write such a message.

If you're using Windows, and you ignored the warning about the default text editor when Git was installed, you may find yourself in a rather unfriendly (and seriously hardcore) text editor called Vim...

Vim is notoriously difficult to use. You will find this out very quickly if you use Vim, as you will have no idea how to even exit the program (indeed, entire books have been written about that very thing).

Unless you're familiar with Vim, I'd advise using one of the alternatives (such as Notepad++).

If you didn't change this on install, find instructions for changing from Vim to Notepad++ here : <https://bit.ly/3ACGVtC>

Commit

Let's make our commit (which at this stage is just adding this new `my_calc.py` file to our repository) (the `-m` option below tells us we want to add a commit message. You *must* add a commit message – if you don't, Git will take you into the text editor to write one) :

```
git commit -m "INIT: add my_calc.py"
```

If successful, you'll get a message like this :

```
[master (root-commit) a5c8a37] INIT: add my_calc.py
 1 file changed, 15 insertions(+)
 create mode 100755 my_calc.py
```

Let's look at the status of our repository again using `git status` :

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

(“clean” just means that there is nothing staged (waiting to be committed) and no files that have been modified that haven't already been committed. Once you modify something, your tree becomes “dirty” again)

Git Log

We can view a complete (or partial) history of all the commits to our repository by viewing the Log :

git log

The commit *identifier*. Every commit has an identifier so we can refer to it if we need to revert back to a previous commit or look at differences etc

HEAD means that this is the latest commit (although we only have one so far). master is the name of the branch (again, we'll come back to that)

```
commit a5c8a37b13552ae02604a3c9abd7277fae04a958 (HEAD -> master)
Author: hsma_master (email redacted)
Date:   Thu Aug 12 13:42:24 2021 +0100

INIT: add my_calc.py
```

Here we can see who made the commit and when

The commit message associated with this commit

Git Log Options

It is good practice to avoid commits that have large numbers of changes, as this can make it difficult to follow (and unpick, if something goes wrong). Instead, you should commit often with smaller changes in each.

This means we may have large commit histories, and we may not want to view the whole history every time we view the log. So, we can use :

`git log -2` to view the n most recent commits (2 here)

`git log --oneline` to view each commit on a single line with condensed information (this is very useful, as it also gives us a 7 digit version of the commit identifier, which is typically all we need to specify a commit)

`git log --oneline -2` to combine the two, and view a single line version of each commit for the n most recent commits

Note – if you’re viewing a multi-page log, when you get to the end you may appear to get stuck with the word “END”. To quit, simply type q (and possibly push Enter too if you’re on Windows)

Making changes

Let's modify our code and stage and commit the changes to see what happens. I'm going to add a subtract function to my_calc.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 12 11:46:21 2021

@author: dan
"""

def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result

def subtract_numbers(num_1, num_2):
    result = num_1 - num_2

    return result
```

Making changes

After we've saved the changes in our `my_calc.py` file, let's first have a look at the status of the repository using `git status`:

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   my_calc.py

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

We can see that, because we staged and committed the `my_calc.py` file before, Git is now tracking the file, and has picked up that the file has been modified.

Making changes

Let's stage the file using *git add*, and have a look at the status again :

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   my_calc.py
```

And then commit :

```
git commit -m "MY_CALC: +subtraction function"
```

Now let's look at our log again :

```
commit a98883120b6ac23674b0d79578648dc2438b2cec (HEAD -> master)
Author: hsma_master [REDACTED] (email redacted)
Date:   Thu Aug 12 15:26:13 2021 +0100

  MY_CALC: +subtraction function

commit a5c8a37b13552ae02604a3c9abd7277fae04a958
Author: hsma_master [REDACTED] (email redacted)
Date:   Thu Aug 12 13:42:24 2021 +0100

  INIT: add my_calc.py
```

Roll Back

Let's modify our code again. I'm going to add a multiply function. But this time, I'm going to introduce a bug...

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 12 11:46:21 2021

@author: dan
"""

def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result

def subtract_numbers(num_1, num_2):
    result = num_1 - num_2

    return result

def multiply_numbers(num_1, num_2):
    while True:
        result = num_1 * num_2

    return result
```

Roll Back

Let's imagine I haven't noticed this bug, and I haven't properly tested my code yet. So I go and stage the file and commit the changes. My log will now look like this :

```
commit 5922e7945c8237b5d1912b0d2317a2e79ef75d3c (HEAD -> master)
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 15:45:50 2021 +0100
```

 MY_CALC: +multiply function

```
commit a98883120b6ac23674b0d79578648dc2438b2cec
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 15:26:13 2021 +0100
```

 MY_CALC: +subtraction function

```
commit a5c8a37b13552ae02604a3c9abd7277fae04a958
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 13:42:24 2021 +0100
```

 INIT: add my_calc.py

Roll Back

The next day I go to run my calculator programme, and imagine my horror when I notice that my new multiply function traps me in an infinite loop! I haven't felt like that since I accidentally loaded up Vim.

But being a good Git (user), I don't panic. The first thing I do is check to see what changes were made between the previous commit and the latest one (where the error's appeared).

Remember – I can look at the *diff* to do this.

Spot the diff

To look at the difference (*diff*) between commits, I use the *git diff* command. But I need to specify the two commits I want to compare by providing their identifiers. Now, I could give the really long versions of the numbers. Or, to make it a bit simpler, I could just use the first 7 digits from each identifier. And remember – if I use the `--oneline` option of *git log*, it'll give me the 7-digit versions of the identifiers.

```
git log --oneline
```

```
5922e79 (HEAD -> master) MY_CALC: +multiply function
a988831 MY_CALC: +subtraction function
a5c8a37 INIT: add my_calc.py
```

So, I want to look at the *diff* between a988831 (the previous commit) and 5922e79 (the latest commit – the HEAD).

Spot the diff

Let's use *git diff* to look at the difference between these two latest commits (note the order – this will look at the changes going from a988831 to 5922e79) :

```
git diff a988831 5922e79
```

```
diff --git a/my_calc.py b/my_calc.py
index 0bbb6ba..2800258 100755
--- a/my_calc.py
+++ b/my_calc.py
@@ -17,3 +17,9 @@ def subtract_numbers(num_1, num_2):

    return result

+def multiply_numbers(num_1, num_2):
+    while True:
+        result = num_1 * num_2
+        [REDACTED]
+        return result
+
```

We can see from this the added function becomes highlighted in the report. And it's now easy to see where things have gone wrong. (Yes – I know in this example it was easy to see before this. But imagine you've got some real code that's much more complex).

Git Revert

There are many different ways to undo changes in Git, which can make things a bit confusing.

Here, we're going to use a *Git Revert*. A revert command will revert a specified commit back to the previous version. It is considered a *safe* command, because you don't lose any history – the old commit, with the bug you introduced, remains in the history, and we can go back to it if needed.

To call a *git revert*, we need to specify the commit that we want to revert back to the previous version. So, in this case, we need to specify the latest commit – the one that introduced the error – so that Git gives us back the version that came before.

Git Revert

Let's use the log to grab the identifier we need (the latest one – the HEAD – in this case) (I'll use the `--oneline` option so that we get the shortened 7 digit identifier) :

```
5922e79 (HEAD -> master) MY_CALC: +multiply function
a988831 MY_CALC: +subtraction function
a5c8a37 INIT: add my_calc.py
```

5922e79 introduced the bug, and so we want to revert to the one before that (a988831). So we specify 5922e79 in the revert command :

```
git revert 5922e79
```

As the revert is creating a new commit, it'll take you into a text editor to ask you for a commit message. But as it's a revert, it'll have a default message that explains it's a revert. Usually, you'd just keep that, unless you want to add more info, so we just exit the editor.

```
Revert "MY_CALC: +multiply function"

This reverts commit 5922e7945c8237b5d1912b0d2317a2e79ef75d3c.

# 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:   my_calc.py
#
```

Git Revert

Now the revert has happened, let's have a look at the log again (we'll use the full description version this time) :

```
commit 6eea86119d42fcc378bdace354fa769d4f41a4c7 (HEAD -> master)
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 16:52:11 2021 +0100

    Revert "MY_CALC: +multiply function"

    This reverts commit 5922e7945c8237b5d1912b0d2317a2e79ef75d3c.

commit 5922e7945c8237b5d1912b0d2317a2e79ef75d3c
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 15:45:50 2021 +0100

    MY_CALC: +multiply function

commit a98883120b6ac23674b0d79578648dc2438b2cec
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 15:26:13 2021 +0100

    MY_CALC: +subtraction function

commit a5c8a37b13552ae02604a3c9abd7277fae04a958
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Aug 12 13:42:24 2021 +0100

    INIT: add my_calc.py
```

The reverted version (pre inclusion of the multiply function) is now the HEAD (latest version)

But we still have access to the history where we added the buggy code

Git Revert

If we flick back to our code in Spyder...

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 12 11:46:21 2021

@author: dan
"""

def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result

def subtract_numbers(num_1, num_2):
    result = num_1 - num_2

    return result
```

...we see that it's magically changed back to the pre-bug version! And herein lies one of the beautiful things of Git (and other version control systems) – you only have a single file that is your “working version” that you’re working on at any one time.

Exercise 2

Now we've covered a few of the basics, it's time for you to have a practice before we move on to some of the other features of Git.

In your groups, you will now work through the instructions in `2b_exercise_2.pdf`. You can work through things individually or as a group, but either way make sure you're all checking in with each other to make sure everyone is following.

You have 40 minutes. When we resume after lunch, we will look at branches.

Branches

Before we move on, it's time we talked about *branches*.

In Git, a *branch* refers to a line of development, including its changes and history.

So far, we've just used the central branch – referred to by default as the *Master* branch (note – it is now recommended you change this branch name; instructions in a moment). That means all changes we make are made to the main codebase.

But, that's not a good way to work, even if we're working alone. For example, we may end up making a number of commits and then realise that we're not happy with the way in which the features we're developing are working. We'd have changed the main code, and would have to revert everything back.

It would be better to create a new branch where we'll do our development work. Then, once we're certain we're happy with that phase of development, we'll *merge* the changes back into the master branch.

Renaming Master

Sometimes the tech industry can be a little way behind in issues of inclusiveness. Language like “master” can not only be racially charged, but cause feelings of isolation and the whole point of FOSS is openness and inclusivity.

For this reason, in 2020 GitHub (under new owners Microsoft) committed to changing the default branch name from “master” to “main”.

However, Git (the version control software) still uses “master” as the default.

I'll now show you how to change this – it's very quick and easy for new repositories where you haven't uploaded anything remotely.

GitHub abandons 'master' term to avoid slavery row

© 15 June 2020



DEVELOPER

GitHub will replace terms associated with slavery like ‘master’ and ‘whitelist’

Tech > Computing

Microsoft’s GitHub is removing coding terms like ‘master’ and ‘slave’

"It's a great idea and we are already working on this," said GitHub's CEO.

Renaming Master

To change the name of a branch, we simply use the following command :

```
git branch -m master main
```

Move or rename a branch

Current name of branch

New name for branch

If we then issue the command *git status*, we can see the change has worked :

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

In line with the changes, I have also changed my username from hsma-master (named after the old “master” branch) to hsma-chief-elf. There may still be some references to my old username in the following slides, but you will also see that this has changed later.

Renaming Default Branch Globally

If we want to change the default branch name in Git for all future created repositories, we can do this easily using the following command (the below example changes all future default branch names to *main* but you can choose something else if you prefer) :

```
git config --global init.defaultBranch main
```

Creating a Branch, and Switching

To create a branch in Git, we use the *git branch* command, and we supply the name of the branch. So, to create a branch called *dev* (which will be an identical copy of the *main* branch initially), we'd use the command :

```
git branch dev
```

If we use the command *git branch* without supplying a name, we get a list of the current branches in the repository :

```
git branch      dev
* main
```

We see from the list above that we have two branches – *main*, and the new branch *dev* that we've just created. The * indicates that it is the main branch that is currently active, so any development we do will still be on the *main* branch. So, to start using the *dev* branch, we need to *switch* to it :

```
git switch dev
```

```
Switched to branch 'dev'
* dev
  main
```

Git Restore

Now we've created and switched to a new branch, let's get our buggy code fixed. Remember, at the moment the *dev* branch is an identical copy of the main branch. So let's access our history :

```
git log --oneline
```

```
4f02176 (HEAD -> dev, main) Revert "MY_CALC: +multiply function"
bd995a1 MY_CALC: +multiply function
22499fb MY_CALC: +subtraction function
b5f4d24 INIT: add my_calc.py
```

So, we want to revert back to the version with the new function we added that's faulty (the multiply function), so we can fix it. Remember, if we revert back in this branch, it won't affect the *main* branch, so it's safe. (Note – the commit numbers are different here to earlier, as these examples are from a newer repo. They will be the same numbers as you had earlier in your repo for you).

We can see above we need to revert back to commit bd995a1. This time, we'll use a *restore* command to do this, which allows us to specify a commit and it'll restore the content from that commit as the currently active code. **This is different to a revert, which goes back to a previous version and creates a new commit from it – reinstating it. Here, we're just pulling in the code that was in a previous version. There's no commit until we tell it to commit.**

Fixing our code

Let's use the restore command to restore the code from the buggy version and make it our current working code :

```
git restore --source bd995a1 my_calc.py
```

--source indicates that you want to restore from a different commit (rather than the current commit; don't worry about that at this stage). We also have to provide the filename (or filepath) of the file (or files) we want to restore (the file my_calc.py in this case).

If we flip back to Spyder after the restore, we can see our buggy code is back as the current version (but remember – only in this *dev* branch, which is our currently active branch).

```
def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result

def subtract_numbers(num_1, num_2):
    result = num_1 - num_2

    return result

def multiply_numbers(num_1, num_2):
    while True:
        result = num_1 * num_2

    return result
```

Fixing our code

Now let's fix our code, so we've got a version with a working multiply function :

```
def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result

def subtract_numbers(num_1, num_2):
    result = num_1 - num_2

    return result

def multiply_numbers(num_1, num_2):
    result = num_1 * num_2

    return result
```

Once the changes have been saved, we stage and commit the changes :

```
git add my_calc.py
git commit -m "FIX: multiply_numbers() patched"
```

Fixing our code

Let's have a look at the last two entries of the commit history now :

```
git log -2
```

```
commit 424b2c6451a120530834e17fbb821e959a29909c (HEAD -> dev)
Author: hsma_master < [REDACTED] (email redacted)
Date:  Tue Jul 5 13:18:51 2022 +0100
```

```
FIX: multiply_numbers() patched
```

```
commit 4f0217621743f8aabfe1839f0cf4c2e01dc87650 (main)
Author: hsma_master < [REDACTED] (email redacted)
Date:  Thu Jun 30 10:55:40 2022 +0100
```

```
Revert "MY_CALC: +multiply function"
```

```
This reverts commit bd995a1ceb44ce8e34e6262f70dcabd3daeb73b4.
```

If we switch back to the *main* branch, and look at the history, we'll see that the latest commit (HEAD) for that branch is the one where we reverted back to the bug free version (before we added multiply) :

```
commit 4f0217621743f8aabfe1839f0cf4c2e01dc87650 (HEAD -> main)
Author: hsma_master < [REDACTED] (email redacted)
Date:  Thu Jun 30 10:55:40 2022 +0100
```

```
Revert "MY_CALC: +multiply function"
```

```
This reverts commit bd995a1ceb44ce8e34e6262f70dcabd3daeb73b4.
```

This is the commit we just made, where we grabbed the code from the buggy version, and fixed it.

Here's the commit where we reverted back to a previous (bug-free) version. This is the latest commit on the *main* branch.

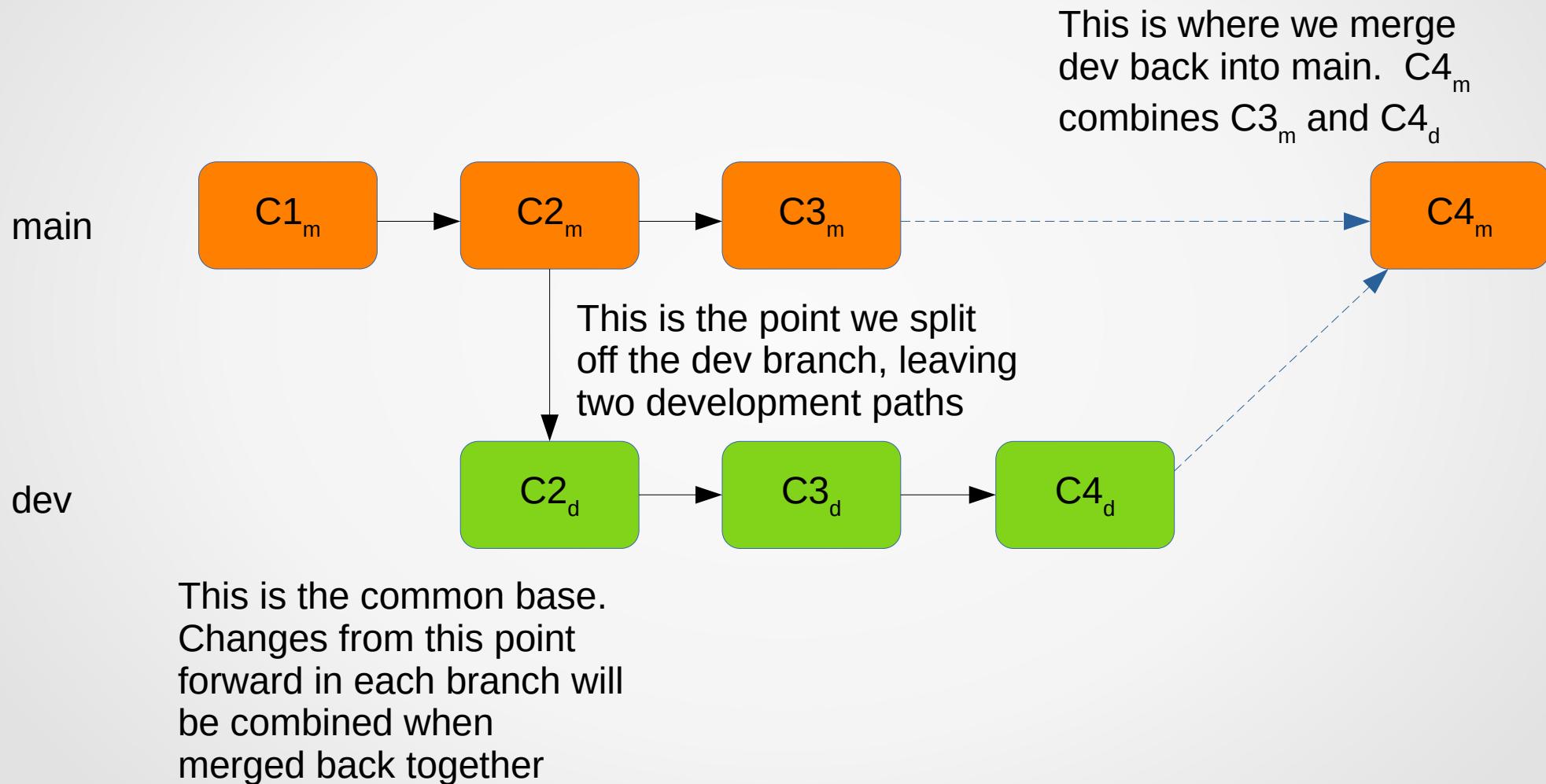
Merge

Once we've tested the code we fixed on the *dev* branch, and we're happy that everything's ok, we can add the changes back into the main code base – the *main* branch. We do this using a *merge*, which *combines* the changes from multiple branches.

We can merge two branches together (most common for smaller or lone developer projects) or multiple branches (more common for large multi-developer projects).

For a merge where we're combining the changes from two branches, Git will look for a common base commit (in our example, that'll be the HEAD commit at the point when we created the *dev* branch – as both branches were identical at this point). Once it's found that common base commit, it will create a *new* commit in the branch into which the changes are being merged that incorporates the changes from *both* branches' commit histories.

Merge



Merge

To perform the merge in our example, we're going to merge the dev branch changes back into the main branch.

We first switch back to the branch into which we want to merge the changes (the *main* branch in this case) :

```
git switch main
```

Then we use the *merge* command, and tell Git the name of the branch we want to merge into this one (*dev* in this case) :

```
git merge dev
```

```
Updating 4f02176..424b2c6
Fast-forward
  my_calc.py | 5 +////
  1 file changed, 5 insertions(+)
```

Merge

Now let's look at the history again using the *log* command on the *main* branch to see what's happened :

```
git log -3
```

```
commit 424b2c6451a120530834e17fbb821e959a29909c (HEAD -> main, dev)
Author: hsma_master [REDACTED] (email redacted)
Date: Tue Jul 5 13:18:51 2022 +0100

    FIX: multiply_numbers() patched

commit 4f0217621743f8aabfe1839f0cf4c2e01dc87650
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Jun 30 10:55:40 2022 +0100

    Revert "MY_CALC: +multiply function"
    This reverts commit bd995a1ceb44ce8e34e6262f70dcabd3daeb73b4.

commit bd995a1ceb44ce8e34e6262f70dcabd3daeb73b4
Author: hsma_master [REDACTED] (email redacted)
Date: Thu Jun 30 10:53:00 2022 +0100

    MY_CALC: +multiply function
```

This is where we merged the development work we did on the *dev* branch (where we fixed the *multiply* function) back into the *main* branch (note both branches named in the HEAD info)

This is where we reverted back to the version before the buggy *multiply* function was added (this was the common base in this example)

This is where we added the buggy *multiply* function

Merge

And if we go back to Spyder in the *main* branch, we see we've now got our working code here :

```
def add_numbers(num_1, num_2):
    result = num_1 + num_2

    return result

def subtract_numbers(num_1, num_2):
    result = num_1 - num_2

    return result

def multiply_numbers(num_1, num_2):
    result = num_1 * num_2

    return result
```

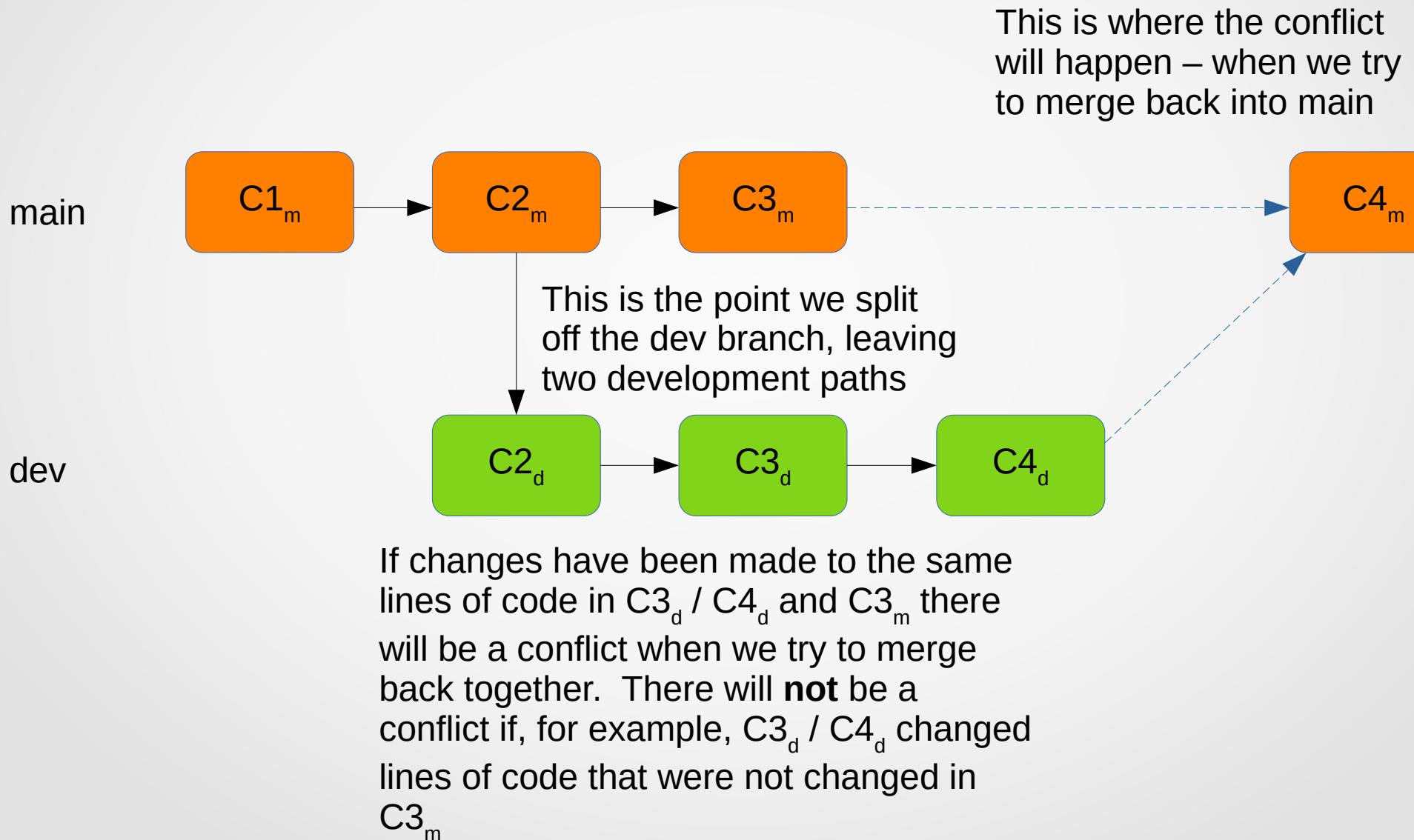
Merge Conflicts

You may have already considered – what happens in a merge if there is conflict between two change histories? This is known as a *merge conflict*. There are actually two ways in which there can be merge conflicts :

- the branches cannot be merged, because there are pending changes in the branch into which the changes are trying to be merged that could be overwritten by the changes we're trying to merge in. In this situation, Git will refuse the merge until the “local state” can be “stabilised” (e.g. pending changes committed etc)
- there's a conflict between the same line(s) of code when trying to merge (e.g. two developers have changed the same line of code in different ways, or maybe one developer has removed a file entirely!). A conflict will occur here if these changes have been made to the same code in different branches after the common base commit.

In both cases, the developer needs to resolve the conflict.

Merge Conflicts



Merge Conflicts

Let's simulate a merge conflict to see what happens. I'll create a file called `my_conflict.py` and save it in the repository folder.

```
def my_function(a, b):
    result = a + b

    return result
```

Now I'll call `git status`, and note that Git has identified the file is there but isn't currently being tracked.

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

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

Next I'll stage and commit the file, and look at the latest commit.

```
commit a4dabd1e07c3aa7115c20a05e9aaf4e8dd15e265 (HEAD -> main)
Author: hsma-chief-elf <d.chalk@exeter.ac.uk>
Date:   Wed Jul 6 09:21:49 2022 +0100

  INIT: add my_conflict.py
```

Merge Conflicts

Now, I want to switch out to my *dev* branch again to do some development. But remember, *dev* is still stuck in the past – it hasn't been updated with the new file I've added, as I did that in the *main* branch. So I'll switch to *dev*, but then I'll merge *main* into *dev*, so I can work from this latest version of my project in *dev*.

```
git switch dev  
git merge main
```

```
Updating 424b2c6..a4dabd1  
Fast-forward  
 my_conflict.py | 14 ++++++  
 1 file changed, 14 insertions(+)  
 create mode 100755 my_conflict.py
```

Now, I'll make a change to one of the lines of code, and stage & commit

```
def my_function(a, b):  
    result = a - b  
  
    return result  
  
git add my_conflict.py  
git commit -m "MY_CONFLICT: changed add to subtract operation"
```

Merge Conflicts

Now I'm going to switch back to the *main* branch, and change that same line of code, but change the add to a multiply operation, instead of a subtract operation. I'll save, stage and commit the changes.

```
def my_function(a, b):
    result = a * b

    return result
```

```
git add my_conflict.py
git commit -m "MY_CONFLICT: changed add to multiply operation"
```

Now we've got the recipe for a conflict in our project. In both branches, we've changed the first line of code of our `my_function(a, b)` function, but we've changed it to a subtract function in *dev* and a multiply function in *main*.

Merge Conflicts

Now we'll try merging the work in the *dev* branch into the *main* branch.

```
git merge dev
```

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

We get a message saying that the merge has failed because of a merge conflict in *my_conflict.py*. We need a bit more information to try to fix this. Let's start by calling a *git status* command.

```
On branch main
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:  my_conflict.py

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

Merge Conflicts

We know the problem is that both branches have modified `my_conflict.py` in ways that cannot be automatically resolved. But we need a bit more information. Let's flip back to `my_conflict.py` in Spyder...

```
def my_function(a, b):
<<<<< HEAD
    result = a * b
=====
    result = a - b
>>>>> dev

    return result
```

We see we've got some weird new notation in our file! Git has marked the conflict for our attention. The ===== indicates the centre of the conflict. Everything between that and the <<<<< HEAD is the content that's in the HEAD of the branch we're currently in (the *main* branch here). Everything between the centre and the >>>>> dev is the content that's in the branch we're trying to merge into this one (*dev* into *master* in this case)

Merge Conflicts

Now we know where the conflict lies, we need to decide how to fix it. In this case, we need to decide whether to keep the code added in the *main* branch (the multiply operation) or the code added in the *dev* branch (the subtraction operation). Or combine them both in some way.

Here, I'll fix the conflict by keeping what was added to the *main* branch and removing the subtraction operation added by the *dev* branch. I'll also need to remove the markers that Git added, otherwise we won't have valid code.

```
def my_function(a, b):
    result = a * b

    return result
```

IMPORTANT : as the merge failed, we're in *Merge Mode* in Git. This means that we can't do things like switch branches, and Git is waiting for us to fix the problem. So when we make the changes above, we're declaring "this is how the merged code should look".

Merge Conflicts

Now I've saved the changes, I need to first stage them. I'll then check the status of the repository.

```
git add my_conflict.py
git status
```

On branch main
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Git is happy that the conflicts have been fixed (because we've said how we want the merged code to look). But we're still in merge mode. We need to commit the changes to finalise things.

```
git commit -m "FIX: resolved merge conflict in my_function() of my_conflict.py"
git log -1
```

commit 84a5872186897a91d67645bc4fc9bd706fd924b3 (HEAD -> main)
Merge: 99ec513 9d84d84
Author: hsma-chief-elf (email redacted)
Date: Wed Jul 6 09:37:54 2022 +0100

FIX: resolved merge conflict in my_function() of my_conflict.py

We see from the log above that the fixed merge is now the HEAD of our *main* branch. But you'll notice that *dev* isn't listed in the HEAD information. That's because we made a fix to make the merge, but the HEAD of the *dev* branch will still have the subtract code. We should bear that in mind if we're doing any more development in the *dev* branch – this will be resolved if we merge *main* back into *dev* when we next branch off to use *dev* again.

.gitignore

When we're working on a real coding project, it's likely we'll have lots of code files, and we may want to stage and commit batches of files at the same time. But there may be files that we don't ever want to include in our repository. For example, we may have results files generated by our model. These files will change each time we run our code, and we don't want to track all these changes! (because a) we don't need to, and b) it'll lead to a very bloated repository!). Or we may have our raw data files that we're using as input that we don't want to share (because they contain sensitive information)

Instead, if there are files that we always want Git to ignore, we can create a `.gitignore` file with the details of the file types and / or directories that we want Git to ignore.

To do this, we simply create a file called `.gitignore` in our directory and list the files / file types / directories we want Git to ignore, and then stage and commit the file (remember to do this for any changes to the file too!). (Note – a `.gitignore` file is preceded by a full stop, which means that your OS will treat this as a *hidden file*. If you want to see / access the file, you'll need to select the option to view hidden files / directories in your OS).

.gitignore

```
.gitignore
1 # analysis log files
2 *.log
3
4 # the results file for our model
5 results.csv
6
7 # results directory
8 results/
9
10
```

The .gitignore file above will tell Git to ignore :

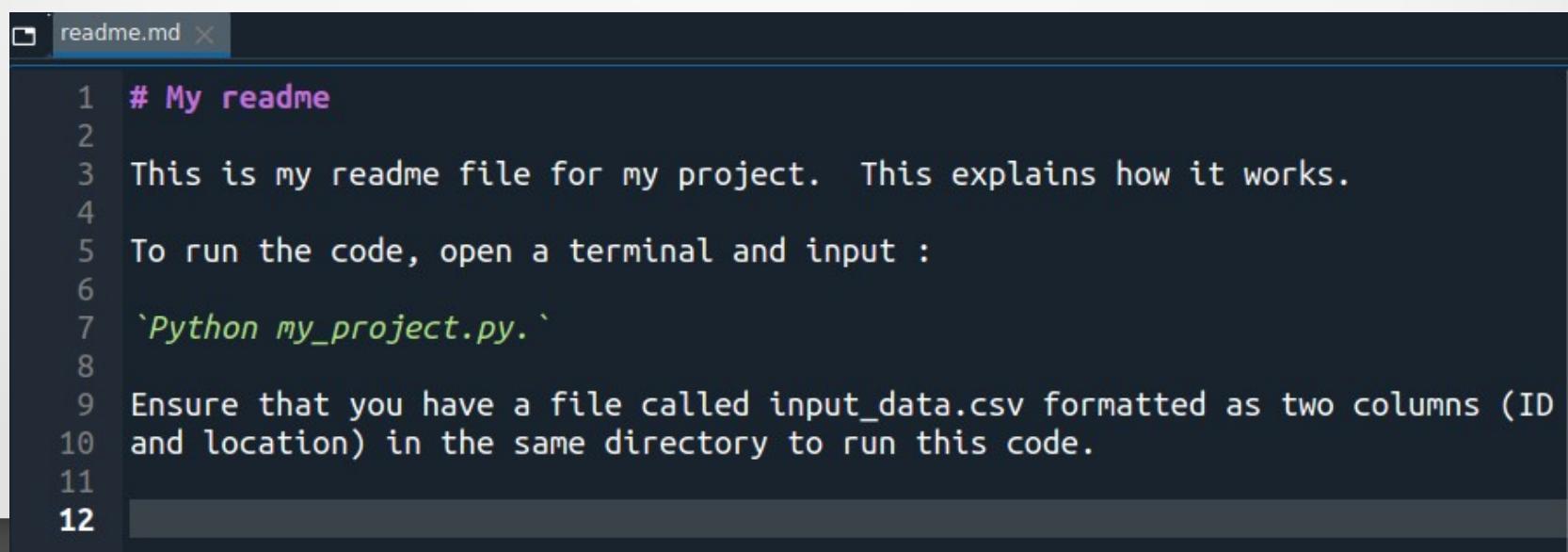
- any files with a file extension of *.log*
- the file *results.csv*
- anything in the directory *results*

There's lots of information here <https://git-scm.com/docs/gitignore> about the “patterns” that you can use in a .gitignore file.

readme.md

It is also good practice to include documentation in your repository that explains what the project is / does / how it works / how to run etc. This is clearly important when you're sharing the project with others, but also even if you're the only one coding and using it – believe me, you *will* forget things about your own code.

We can easily add documentation to a Git repository by creating a file called *readme.md*, and then staging and committing the file. If we're using a repository hosting service such as GitHub, a *readme.md* file will automatically be designated as the documentation for your project, allowing people to easily access it.

A screenshot of a code editor window titled "readme.md". The file contains the following text:

```
1 # My readme
2
3 This is my readme file for my project. This explains how it works.
4
5 To run the code, open a terminal and input :
6
7 `Python my_project.py.`
8
9 Ensure that you have a file called input_data.csv formatted as two columns (ID
10 and location) in the same directory to run this code.
11
12
```

The code editor has a dark theme with syntax highlighting for Python code.

Exercise 3

Now you'll have a chance to practice working with branches and merging.

In your groups, you will now work through the instructions in 2b_exercise_3.pdf. You can work through things individually or as a group, but either way make sure you're all checking in with each other to make sure everyone is following.

You have 40 minutes + 10 minute break.

GitHub

GitHub is a service that hosts Git repositories. It is the largest such service on the internet, and is now owned by Microsoft.

GitHub allows developers from all across the world to not only share their work, but also collaborate with each other.

We'll cover a few of the basics of GitHub today, but will also point you to further information on both Git and GitHub. We strongly recommend that you use GitHub (or an equivalent repository hosting service) as well as Git for your HSMA project work.

GitHub Personal Access Tokens

Since August 2021, GitHub now requires any Git operations on GitHub to be authorised using a *Personal Access Token*, rather than simply a password. This includes pushing and pulling to and from remote repositories using the Git command line.

Instructions for creating a Personal Access Token can be found here : <https://tinyurl.com/mry8s5tm>. And further information about the change can be found here : <https://tinyurl.com/2h3ap3j7>.

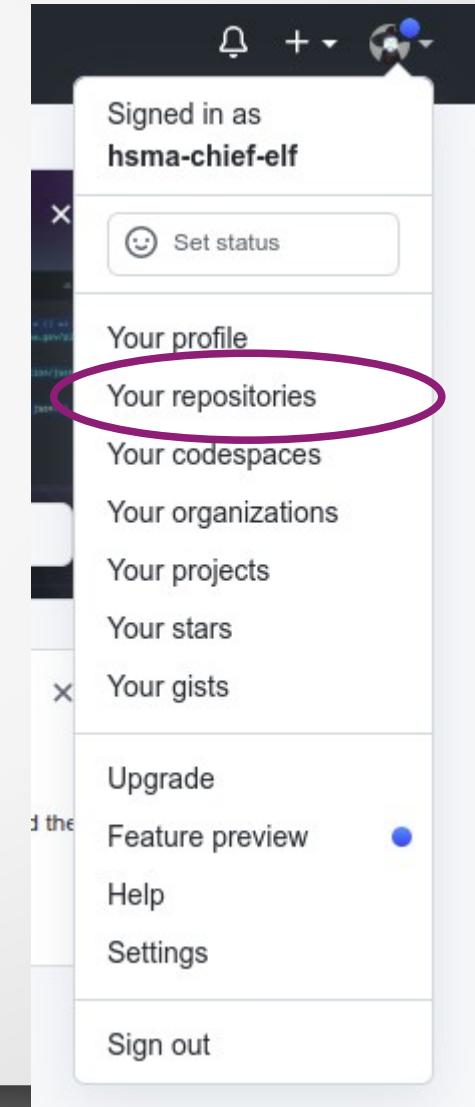
In short, you create a Personal Access Token for a specific purpose (with a specified expiration date). You then use that Personal Access token when asked for your password by the Git command line interface.

Creating a new Repository on GitHub

To create a new repository on GitHub, visit <https://github.com/> and sign into your account (or create one if you haven't got one).

Then click on your profile picture at the top right and click “Your Repositories”

This will take you to a page which lists your repositories (if you have any). Click the green “New” button on this page.



Creating a new Repository on GitHub

The next page asks you to input some information about your new repository. What you need to do here will be different depending on whether you plan on importing a repository (such as the one we've created locally today), or you're creating a brand new one.

Note – any new repositories created on GitHub now will use *main* as the default branch name.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner *  hsma-chief-elf

Repository name * /

Great repository names are short and memorable. Need inspiration? How about [bookish-octo-giggle](#)?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

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

Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore
Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template:

Choose a license
A license tells others what they can and can't do with your code. [Learn more.](#)

License:

 You are creating a public repository in your personal account.

Importing an existing repository

If we want to import an existing repository from Git, we simply follow the instructions given on the landing page for the new repository under “..or push an existing repository from the command line”. We input these commands whilst working in the repository we want to upload.

...or push an existing repository from the command line

```
git remote add origin https://github.com/hsma-chief-elf/test_2.git  
git branch -M main  
git push -u origin main
```

Note the default branch for a new repo in GitHub is already *main* (not *master*)

This url will be different for you – it'll be whatever the url is for your new GitHub repo

Importing an existing repository

Once you have issued the commands provided, you will be asked to input your username and password.

The username should be your GitHub username (which should be the same as your Git username if you followed the earlier advice).

When asked for your password, you should copy and paste your Personal Access Token (as per earlier instructions).

Assuming everything has worked, you should get a message similar to the below :

```
Enumerating objects: 3, done.  
Counting objects: 100% (3/3), done.  
Delta compression using up to 8 threads  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 344 bytes | 344.00 KiB/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To https://github.com/hsma-chief-elf/test_2.git  
 * [new branch]      main -> main  
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

origin

You'll have seen in the previous slides references to something called *origin*.

origin is an alias used to refer to the remote repository, without having to write out the url (beyond the initial command where you specified the url to be given the *origin* alias).

Git Push and Pull

Before we go any further, we need to talk about two key concepts in Git when working with remote repositories : *push* and *pull*.

Pushing means sending commits to a remote git repository. We just effectively did a push when we imported our local repository to our new GitHub repo – we pushed our commit history from our local repository up to the remote repository.

Pulling means retrieving commits from a remote git repository. In addition, if someone has taken a copy of your repo (typically by *cloning* or *forking* your repo), they can request you *pull* their changes into your repository. Pulls need *pull requests* which the owner of the repo can choose whether or not to approve.

We'll talk about *forking*, *cloning* and *pulling* in a bit.

Push

Let's start with a *push* example. Let's imagine we have a local repository that we've uploaded to GitHub that contains a single file – `gh_test.py`. The file contains the following code :

```
print ("hello world")
```

Let's have a look at the status of the repository using *git status* :

```
On branch main
Your branch is up-to-date with 'origin/main'.

nothing to commit, working tree clean
```

This tells us that we are “in sync” with the main branch of the remote repository (*origin*) – we haven’t yet made any changes. If we look at *git log*, we can also see that the main branch of the remote repository (*origin*) is listed in the head.

```
commit 060130492354432ebeca1e3c0fdaf452b1d0071a (HEAD -> main, origin/main)
Author: hsma-chief-elf [REDACTED] (email redacted)
Date:   Wed Jul 6 16:21:18 2022 +0100

INIT: add gh_test.py
```

Push

Now we'll make a change to the code. I'll add a second line :

```
print ("hello world")
print ("bonjour")
```

I'll now save the file, and stage and commit the changes. Let's check the status again :

```
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

This tells us that we are now ahead of *origin* (the remote repository) by 1 commit on the *main* branch, because we've made and committed a change that hasn't yet been updated on the remote repository.

Push

Let's have another look at the log :

This is the latest local commit on the *main* branch. Note that *origin/main* is no longer listed in the head.

```
commit 40ccffa5d096edd433edd4bf6c05084323b2d37c (HEAD -> main)
Author: hsma-chief-elf [REDACTED] (email redacted)
Date:   Wed Jul 6 17:16:35 2022 +0100

    GH_TEST: +second print message

commit 060130492354432ebeca1e3c0fdaf452b1d0071a (origin/main)
Author: hsma-chief-elf [REDACTED] (email redacted)
Date:   Wed Jul 6 16:21:18 2022 +0100

    INIT: add gh_test.py
```

This is the latest commit on the *main* branch of the *origin* (remote) repository.

Push

Let's now *push* our changes to the remote repository (*origin*) :

```
git push origin main
```

The command *git push* initiates a push

The alias of the remote repository (this is nearly always *origin*)

The name of the branch whose commits we want to upload to the remote repository (push)

```
Enumerating objects: 5, done.  
Counting objects: 100% (5/5), done.  
Delta compression using up to 8 threads  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 316 bytes | 316.00 KiB/s, done.  
Total 3 (delta 1), reused 0 (delta 0)  
remote: Resolving deltas: 100% (1/1), completed with 1 local object.  
To https://github.com/hsma-chief-elf/test_2.git  
 0601304..40ccffa main -> main
```

Push

Now we've pushed to origin, a *git status* shows us that we are now in sync with the remote repository :

```
On branch main
Your branch is up-to-date with 'origin/main'.

nothing to commit, working tree clean
```

And the *git log* lists the *main* branch of *origin* in the head once again :

```
commit 40ccffa5d096edd433edd4bf6c05084323b2d37c (HEAD -> main, origin/main)
Author: hsma-chief-elf <d.chalk@exeter.ac.uk>
Date:   Wed Jul 6 17:16:35 2022 +0100

    GH_TEST: +second print message
```

And if we look at our repo on GitHub, we can see the latest commit message :



hsma-chief-elf GH_TEST: +second print message

40ccffa 18 hours ago 2 commits

gh_test.py

GH_TEST: +second print message

18 hours ago

Push

In the GitHub repo, if we click on the filename, and then click “History” over to the right, it’ll show us the commit history for this file (note the times / dates refer to when it was committed to the *local* repository, not when it was *pushed*) :

History for [test_2 / gh_test.py](#)

- o Commits on Jul 6, 2022
 - GH_TEST: +second print message
hsma-chief-elf committed 18 hours ago
 - INIT: add gh_test.py
hsma-chief-elf committed 19 hours ago
- o End of commit history for this file

Or if we click on the commit message, we can see the *diff* :

Showing 1 changed file with 2 additions and 0 deletions.

v ↻ 2 ████ ████ gh_test.py ↻

...	00 -8,3 +8,5 00
8	8
9	9 print ("hello world")
10	10
11	+ print ("bonjour")
12	+

Pulling – Forking and Cloning

Let's say we've seen a repository on GitHub to which we're interested in contributing, or we're working on a collaborative project with others.

There are two things that we could do to grab a copy of the repository to work on : we could *fork* it, or we could *clone* it.

If we *fork* the repo, we create a copy of the original repo that we can then work on entirely separately, but this copy is made *remotely* (ie we create a copy of it on GitHub, not locally). This is useful if, for example, you've seen a project you like on GitHub and want to use that as a starting point to go off and do your own thing.

If we *clone* the repo, we still create a copy but this is downloaded locally and that *local* copy remains in contact with the *remote* repository. This means we can easily keep up to date with changes to the original repository, and can more easily contribute back to it. This is useful if, for example, you are working on a collaborative project with others. Cloning is a *pull*, because we are retrieving commits from a remote repository and storing them locally.

If you are not part of the project team for a repo, you should *fork* a repo first before you *clone* it. This means you have an entirely separate codebase that you can work with. You can still request that any changes are made back into the original repo (we'll see an example of that in a moment).

Forking

Let's look at an example of *forking* a repo (I'm using the random repo <https://github.com/balikasg/topicModelling> as an example). To fork a repo, we first click the "Fork" button at the top right of the page in the repository we want to fork.



We're now presented with the following, which allows us to rename the forked repo if we want (the default is to keep the name the same), and to change the description if we want.

Create a new fork

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks](#).

Owner *



hsma-chief-elf ▾

Repository name *

topicModelling



By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

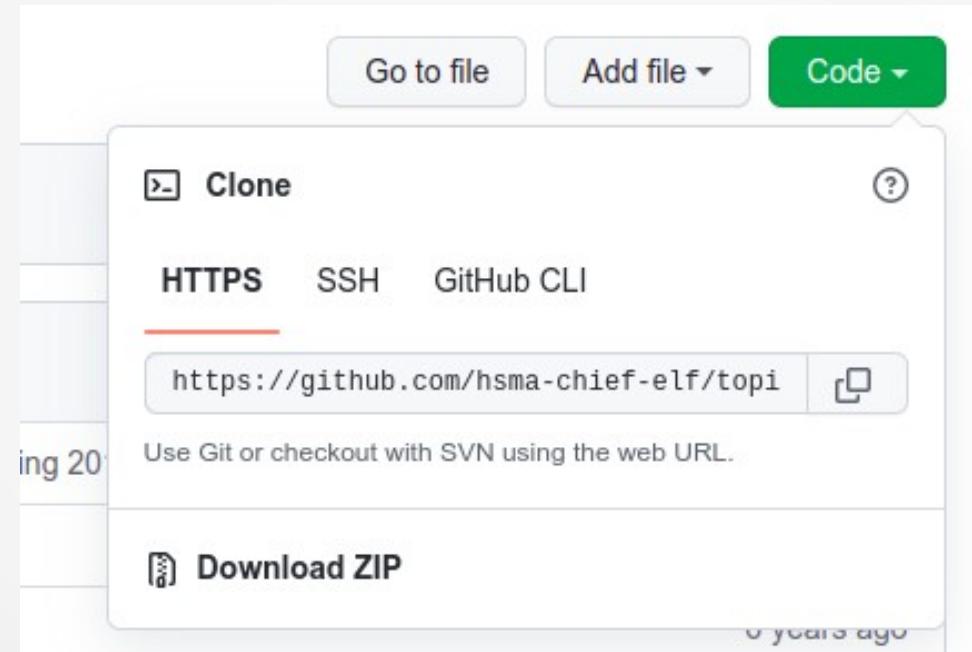
Example of forking for HSMA using <https://github.com/balikasg/topicModelling/fork> as an example repo

ⓘ You are creating a fork in your personal account.

Create fork

Cloning

Once the repository has been forked, we can clone it to download it locally (“pull” it). To do this, we first need to get the https URL of the repository we want to clone (ie the one we’ve just created from a fork). We can find this by clicking the green “Code” button and then copying the link under HTTPS :



Note – SSH is more secure than HTTPS (it uses a “key” based system), but you may find that you cannot use SSH on your machine because of required permissions. We will default to using HTTPS here, which is generally considered fine for most things (especially now Personal Access Tokens are mandatory). But it does mean you have to input your password (Personal Access Token) every time you interact with the remote repository via Git. If you want to use SSH, you may need to speak with your IT department.

Cloning

Now, in our command line interface, we navigate to a directory where we want to store the repo locally and use a *git clone* command, followed by the https address we just copied :

```
(base) dan@lucy:~/Dropbox/HSMA_Onslaught/HSMA_5/Phase_1/h5_P1_Training/2B_Principles_of_ples_of_Fpl  
Version_Control/cloned_repo_example$ git clone https://github.com/hsma-chief-elf/topicModelling.git  
Cloning into 'topicModelling'...  
remote: Enumerating objects: 72, done.  
remote: Total 72 (delta 0), reused 0 (delta 0), pack-reused 72  
Unpacking objects: 100% (72/72), 3.27 MiB | 1.46 MiB/s, done.
```

This will have created a new directory with the repo name in our current location (in this case, a directory called topicModelling). We now need to change into this directory using the command *cd* :

```
cd topicModelling
```

If we now call *git status*, we can see that we're now in a Git repository :

```
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
nothing to commit, working tree clean
```

Cloning

This also means that we can use *git log* to see a commit history :

```
commit ba5795eccbf23223d236c999c9cbbcd3d0c11e95 (HEAD -> master, origin/master, origin/HEAD)
Author: balikasg <geompalik@hotmail.com>
Date:   Thu Dec 22 17:02:09 2016 +0100

    Update README.md

commit bc64ec406bafdf14ced7907fcb2bc20c074dafd4a
Author: balikasg <geompalik@hotmail.com>
Date:   Mon Oct 10 14:53:07 2016 +0200

    Update README.md

commit 4530fd51a2848a2342cd526d7bd7fb530143cb22
Author: balikasg <geompalik@hotmail.com>
Date:   Mon Oct 10 14:48:34 2016 +0200

    Update README.md

commit 81fec36b1b6c60257f4b885a02b1bd87a7ff85f8
Author: balikasg <geompalik@hotmail.com>
```

Making changes

Now let's create a new branch and make some changes to the code.

```
git branch dev
xample/topicModelling$ git branch
  dev
* master

xample/topicModelling$ git switch dev
Switched to branch 'dev'
import nltk, re, string
from nltk.corpus import conll2000
from nltk.tokenize import sent_tokenize
import numpy as np
recover_list = {"wa": "was", "ha": "has"}

print ("Dan is the best")

class VocabularySentenceLayer:
    def __init__(self, stopwords, wl, excludes_stopwords=False):
        self.vocas = []          # id to word
        self.vocas_id_dict()     # word to id
```

I've made a change to the file vocabulary.py here. I'll let you guess what the change was...

Making changes

git status shows us we've modified the file, so let's stage and commit it.

On branch dev

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: copulaLDA/vocabulary.py

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

`git add copulaLDA/vocabulary.py`

`git commit -m "VOCABULARY.PY: added important print message"`

Push and Pull the commit

Now we've committed the change to the local repository, we next need to *push* the commit to the remote repository (this is our forked repo, forked from the original one. Remember – that's the one we cloned, not the original repo).

```
git push origin dev
```

If we go to our remote repo on GitHub, we'll see the following :

The screenshot shows a GitHub repository page for 'hsma-chief-elf / topicModelling'. The 'Code' tab is selected. A yellow banner at the top right says 'dev had recent pushes 3 minutes ago'. A green button labeled 'Compare & pull request' is circled in red. To the right, there's an 'About' section with a note about forking for HSMA. Below the banner, it shows 'master' branch, '2 branches', and '0 tags'. It also says the branch is up to date with 'balikasg/topicModelling:master'. There are two commits listed: 'balikasg Update README.md' and 'copulaLDA add copulaLDA from the coling 2016 paper'. On the far right, there are stats: 0 stars, 0 watching, and 58 forks.

We now click the “Compare & pull request” button

Push and Pull Request the commit

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

IMPORTANT – if we create a Pull Request, we will send a request to the owner of the *original repo* (the one from which we forked). This will basically request that the changes we've made in our forked repo are incorporated into their repo (which they have to approve). If you want to hear a funny story about what happened when Dan was writing this training...

We see here we are comparing the *master* branch of the original repo (owned by balikasg) with the *dev* branch of my forked repo (owned by me – hsma-chief-elf). GitHub tells us these branches can be merged (because there are no conflicts – remember, all I've done here is added a new print message, and the same lines of code haven't been changed in *master* in the original repo (this is an old repo, so still uses *master* as the default branch name)). BUT the owner has to approve it...

Push and Pull Request the commit

If we scroll down on the same screen, we can also see the *diff*, so we can see precisely what was changed in this branch. We can also add comments in the box at the top of the page, to communicate further to the repo owner. Once we're happy, we click the “Create Pull Request” button.

-o 1 commit ± 1 file changed ✊ 1 contributor

Commits on Jul 12, 2022

VOCABULARY.PY: added important print message
hsma-chief-elf committed 39 minutes ago

Showing 1 changed file with 1 addition and 0 deletions.

copulaLDA/vocabulary.py

```
@@ -4,6 +4,7 @@
 4   4   import numpy as np
 5   5   recover_list = {"wa": "was", "ha": "has"}
 6   6
 7 + 7   + print ("Dan is the best")
 8   8
 9   9   class VocabularySentenceLayer:
10  10     def __init__(self, stopwords, wl, excludes_stopwords=False):
...+...
```

Pull Request and Pull

Now we've submitted a Pull Request, the GitHub repo owner can view the details of any Pull Requests. Changes aren't automatically updated – they need to be approved by the owner.

You can view Pull Requests in the Pull Requests tab of your repo. Here's a screenshot I took shortly after I realised I'd accidentally submitted the pull request to the owner of this repo and had to swiftly close it...

The screenshot shows a GitHub repository interface. The top navigation bar includes links for Code, Issues, Pull requests (which is underlined in red), Actions, Projects, Wiki, Security, and Insights. The main title of the pull request is "VOCABULARY.PY: added important print message #3". A red button indicates the pull request is "Closed". Below the title, it says "hsma-chief-elf wants to merge 1 commit into `balikasg:master` from `hsma-chief-elf:dev`". The pull request has 1 commit, 0 checks, and 1 file changed. The commit message is "VOCABULARY.PY: added important print message". The author of the commit is "hsma-chief-elf" and it was made 20 minutes ago. In the comments section, "hsma-chief-elf" comments 12 minutes ago: "Apologies - this was submitted in error; I'm writing training on GitHub and accidentally submitted the example Pull Request!". Finally, "hsma-chief-elf" closes the pull request 12 minutes ago. At the bottom, "hsma-chief-elf" deletes the "dev" branch 11 minutes ago, with a "Restore branch" link available.

Cloning without Forking

For completeness, let's just briefly walk through the scenario where we create a brand new repo on GitHub and then want to download a copy (*pull* it by *cloning* it) locally, so we can work on it. This is the largely the same as before, except we don't need to fork the repo first, and we don't need to import an existing local repo (by *pushing* it up to GitHub). But it's worth stepping through as this may well be the most likely workflow for your projects.

1. Create new repository on GitHub.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner * Repository name *

 hsma-chief-elf ▾ / my_shiny_new_repo ✓

Great repository names are short and memorable. Need inspiration? How about [psychic-spoon](#)?

Description (optional)

Example repo for training

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

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

Cloning without Forking

2. Copy the URL from the blue box at the top of your new repo, then navigate to the local folder where you want to download (*pull*) the repo using your command line and use a *git clone* command.

Quick setup — if you've done this kind of thing before

or

HTTPS

SSH

https://github.com/hsma-chief-elf/my_shiny_new_repo.git

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

```
git clone https://github.com/hsma-chief-elf/my_shiny_new_repo.git
Cloning into 'my_shiny_new_repo'...
warning: You appear to have cloned an empty repository.
```

Note the warning above if you are cloning a brand spanking new repo with no files in it whatsoever.

Cloning without Forking

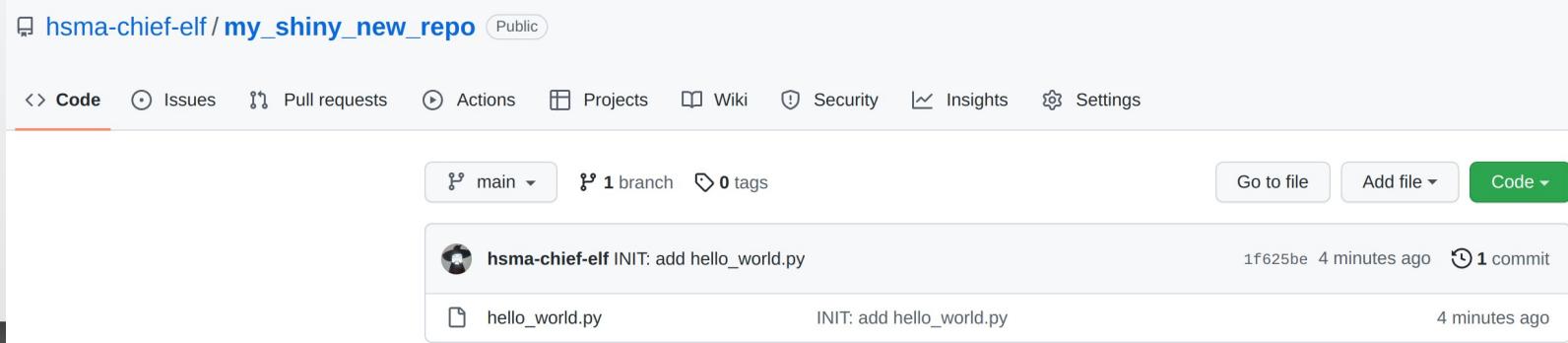
3. *cd* into the directory that's just been downloaded (that has the name of the repo you've just cloned).

```
cd my_shiny_new_repo/
```

You can now work on your repo locally, and *push* changes up to the remote repository (*origin*). Let's just add a simple file and do that.

```
hello_world.py X
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Tue Jul 12 13:12:16 2022
5
6 @author: dan
7 """
8
9 print ("hello world")
```

```
git add hello_world.py
git commit -m "INIT: add hello_world.py"
git push origin main
```



The screenshot shows a GitHub repository page for 'hsma-chief-elf / my_shiny_new_repo'. The 'Code' tab is selected. At the top, it shows 'main' branch, '1 branch', and '0 tags'. Below that, a commit history is displayed:

- hsma-chief-elf INIT: add hello_world.py** (1f625be, 4 minutes ago, 1 commit)
- hello_world.py** (INIT: add hello_world.py, 4 minutes ago)

Cloning without Forking

And if we wanted to push up commits to origin from the non-default branch :

```
git branch dev
```

```
git switch dev
```

```
print ("hello world")
print ("new stuff")
```

```
git add hello_world.py
```

```
git commit -m "HELLO_WORLD.PY: added second print message"
```

```
git push origin dev
```

☞ **dev** had recent pushes less than a minute ago

Compare & pull request

Cloning without Forking

[Open a pull request](#)

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

As I'm the sole owner of the repo, I can create a pull request and then decide whether I want to approve it. This is good practice for tracking your development on sole ownership projects.

Cloning without Forking

HELLO_WORLD.PY: added second print message #1

Open hsma-chief-elf wants to merge 1 commit into `main` from `dev`

Conversation 0 Commits 1 Checks 0 Files changed 1

hsma-chief-elf commented 2 minutes ago

I felt it needs a second print message for maximum awesomeness.

HELLO_WORLD.PY: added second print message dc6fd45

Add more commits by pushing to the `dev` branch on [hsma-chief-elf/my_shiny_new_repo](#).

Continuous integration has not been set up GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch Merging can be performed automatically.

Merge pull request or view command line instructions.

Add more commits by pushing to the `dev` branch on [hsma-chief-elf/my_shiny_new_repo](#).

Merge pull request #1 from hsma-chief-elf/dev

HELLO_WORLD.PY: added second print message

Confirm merge Cancel

Cloning without Forking



Pull request successfully merged and closed

You're all set—the `dev` branch can be safely deleted.

Delete branch

We can now see that the `dev` branch I pushed has been merged with `main` on `origin` via an approved *pull request*.

main ▾

2 branches

0 tags

Go to file

Add file ▾

Code ▾



hsma-chief-elf Merge pull request #1 from hsma-chief-elf/dev ...

bed0148 2 minutes ago ⏲ 3 commits

hello_world.py

HELLO_WORLD.PY: added second print message

3 hours ago

Help people interested in this repository understand your project by adding a README.

Add a README

Cloning without Forking

However, if we switch back to *main* on our local repository :

```
print ("hello world")
```

We're still on the old version. That's because we merged our *dev* branch with *main* on *origin*, but not in the local repository. We're out of sync with *origin*. This is a good opportunity to show you how to use a *git pull* to resync from *origin* :

```
git pull origin main
```

The above command will *pull* the updates from the remote repository (*origin*) to the local repository. If we switch back to Spyder, we can see we're back to the up-to-date version of the code :

```
print ("hello world")
print ("new stuff")
```

Remember – if you're working on a collaborative project, you should make sure you're *pulling* in up-to-date code before you start working on your bits. You don't have to pull in the new code every time changes are made though - a good project manager will make sure people are working on different elements to reduce the potential for *merge conflicts*.

Licenses

If you're uploading code to GitHub, you should consider providing a license that specifies how people can use your code. If you supply no license, standard copyright law will apply – in other words, it belongs to you, and nobody else can do anything with it. But that's not very open source!

There are *lots* of different open source licenses. However, some popular ones are Creative Commons licenses, MIT licenses, and GNU licenses.

You may have noticed that the HSMA Programme training materials are protected by a Creative Commons License – specifically, the CC BY-NC-SA 4.0 license (Creative Commons Attribution-NonCommercial-Sharealike 4.0 International License). This license means that anyone can share, adapt and use the materials as long as they :

- give appropriate credit to the original author, provide a link to the license and indicate if changes were made (attribution)
- do not use either the original content or the adapted content for commercial purposes (noncommercial)
- use the same license for any work adapted from the original work (sharealike)

However, Creative Commons licenses are not generally recommended for software projects (read why here :

<https://creativecommons.org/faq/#can-i-apply-a-creative-commons-license-to-software>).

Licenses

Here's a good starting point for exploring different licenses :

<https://opensource.org/licenses>

You can use this website to help you choose an appropriate license :

<https://choosealicense.com/>

But a good first start is the MIT license family :

<https://opensource.org/licenses/MIT>

Whichever license you choose, all you need to do to apply the license is either :

- upload a .txt file containing the license text to the root directory of your repository
- when you first create a GitHub repository, you can choose a license as part of the set up (under “Choose a License”)
- if you’ve already set up a GitHub repository, you can still use GitHub to choose a license. Just click “Add File” → “Create New File” then type “LICENSE” in the name field box that appears and click the “Choose a license template” button that appears on the right.

Further Reading

Before we move onto the final exercise, I just wanted to flag up that today we've just covered the basics of version control, Git and GitHub to get you started using version control in your coding projects. But there's *much* more that you can learn about these topics. Here are some good starting resources :

- <https://bit.ly/3kirdNI> (Tom's excellent online materials on the basics of version control and Git, upon which this training is based)
- <https://bit.ly/3B88sDj> (an excellently written tutorial by Software Carpentry that takes you through the basics + goes into details about collaborative version control, licensing options for your code and using Git with Rstudio)
- https://ohsh*tgit.com/ (replace the * with a vowel that might help to express severe frustration...) (a good little resource for showing you how to get out of things when you muck up in Git)
- <https://bit.ly/3ygzsyy> (a nice little tutorial that takes you through from the basics of Git to then using GitHub with your repository)

But there's much more out there, and lots of help in the community, so go forth and explore!

Exercise 4 – To Me, To You

For the remainder of the session today, you will once again work in your groups, and work through the instructions in [2b_exercise_4.pdf](#).

The tasks in the exercise will get you to try out some of the basics of GitHub that we've gone through this afternoon.

Note – it is possible that some of you may have had restrictions placed on your computers that limit or prevent your usage of GitHub. If that's the case, you should speak to your IT departments to have this removed, but in the meantime, identify someone in your group who has no such restrictions who can take the lead for this group exercise.

