Module 2 : Open Source Collaborative Development

Session 2B : Principles of FOSS and Version Control

Dr Daniel Chalk

"The Git Crowd"

NIHR | Applied Research Collaboration South West Peninsula

@peninsula_ARC #HSMA4

HSMA 4

HSMA 4 Everyone

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 :)

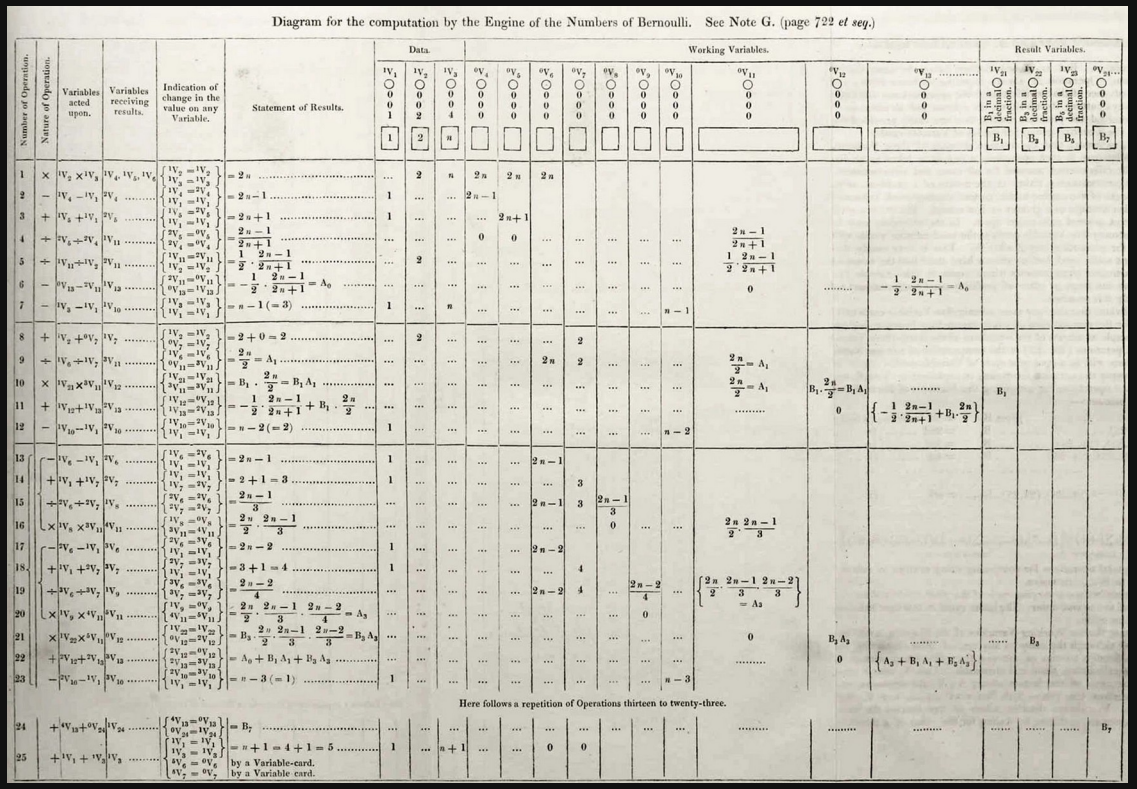


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

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)

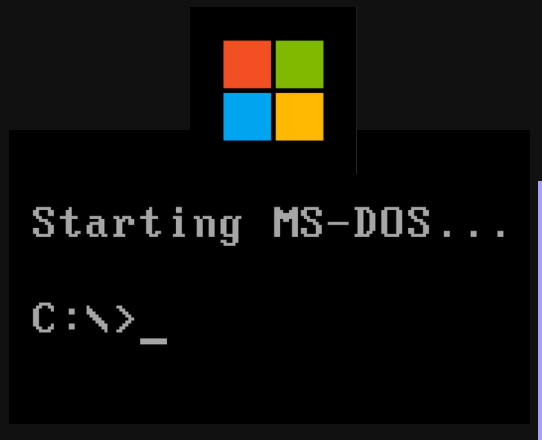
Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid?

I would appreciate letters from anyone who wants to pay up

Locking Down

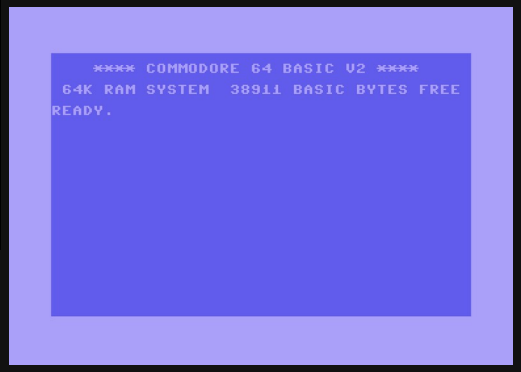
In the 70s and 80s, software companies began “hiding” the code for their software – “Closed Source Software”.

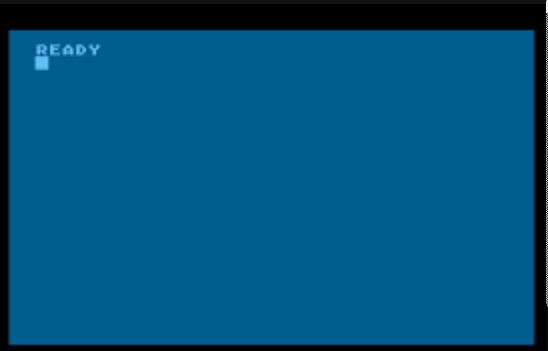
And in 1980, Copyright Law was extended in the US to cover software (which had previously been considered to be not copyrightable because they were ideas / methods / processes).

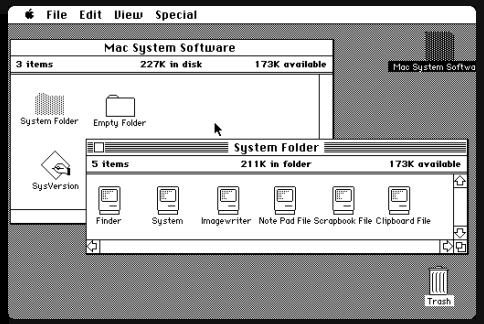

Starting MS-DOS...
C:\>_

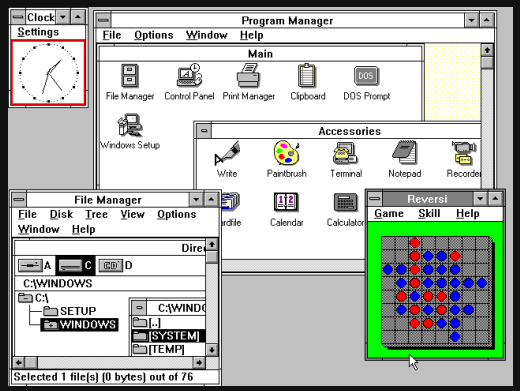

Commodore

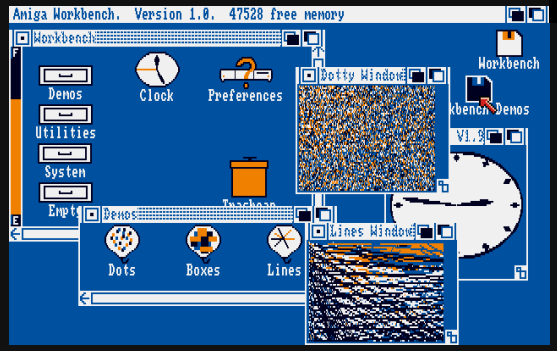

ATARI


**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.

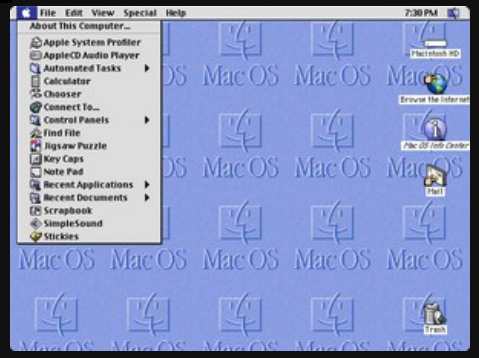

READY


File Edit View Special
Mac System Software
3 items 227K in disk 173K available
System Folder
Empty Folder
SysVersion
System Folder
5 items 211K in folder 173K available
Finder
System
Imagewriter
Note Pad File
Scrapbook File
Clipboard File
Trash


Clock
Settings
Program Manager
File Options Window Help
Main
File Manager
Control Panel
Print Manager
Clipboard
DOS Prompt
Windows Setup
Accessories
Write
Paintbrush
Terminal
Notepad
Recorder
Calendar
Calculator
File Manager
File Disk Tree View Options
Window Help
C:\WINDOWS
SETUP
WINDOWS
SYSTEM
TEMP
Reversi
Game Skill Help
Selected 1 file(s) (0 bytes) out of 76


Amiga Workbench, Version 1.0, 47528 free memory
Workbench
Demos
Utilities
System
Clock
Preferences
Dotty Window
Workbench
Demos
Dots
Boxes
Lines
Lines Window

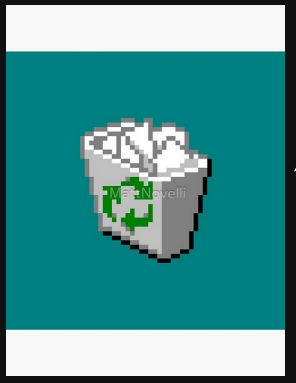

Desk File View Options
FLOPPY DISK
FLOPPY DISK
TRASH
CARTRIDGE


File Edit View Special Help
About This Computer...
Apple System Profiler
AppleCD Audio Player
Automated Tasks
Calculator
Chooser
Connect To...
Control Panels
Find File
Jigsaw Puzzle
Key Caps
Note Pad
Recent Applications
Recent Documents
Scrapbook
SimpleSound
Stickies
MacOS

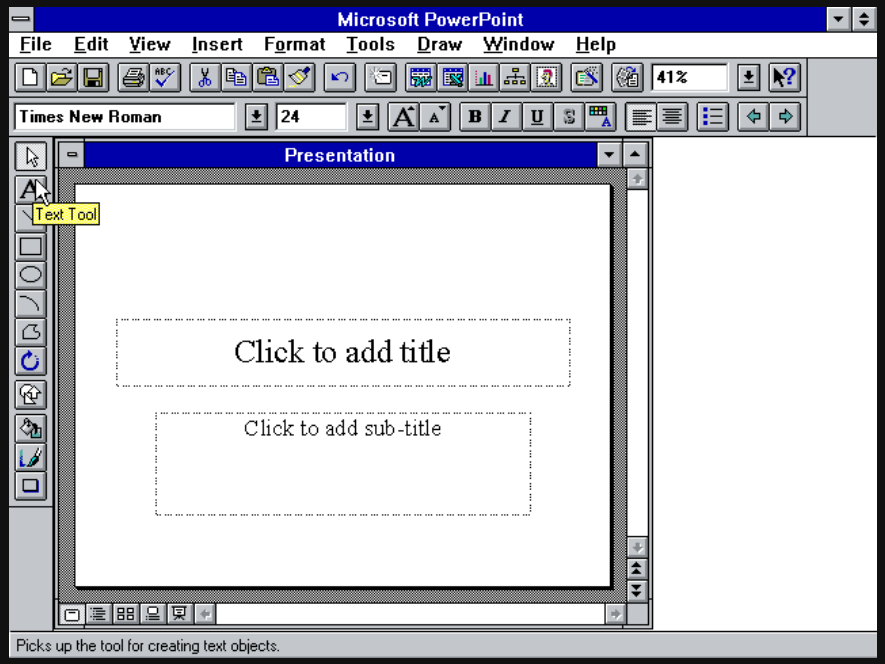

sinclair

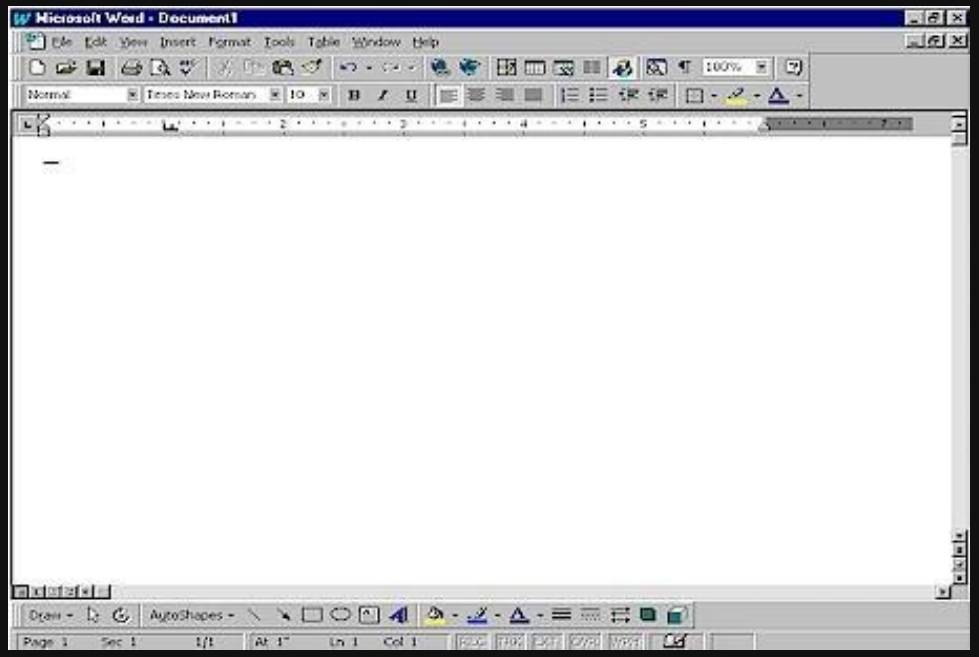

AMSTRAD

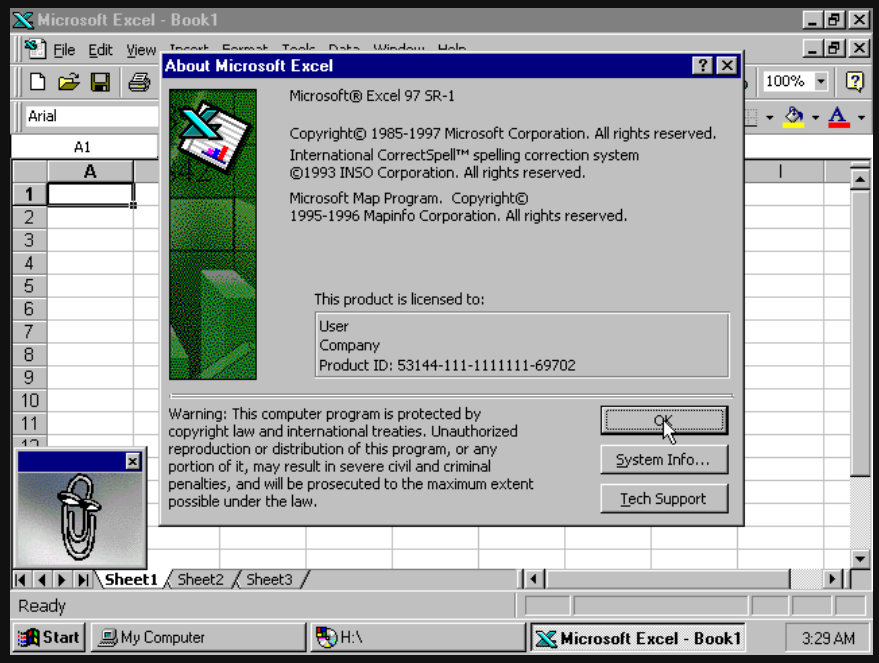

My Computer
Network Neighborhood
Inbox
Recycle Bin
The Microsoft Network
My Briefcase
Welcome
Welcome to Windows 95
Did you know...
To open a program, you just click the Start button, and then click the program's icon.
What's New
Online Registration
Close
Start
Welcome
6:18 PM

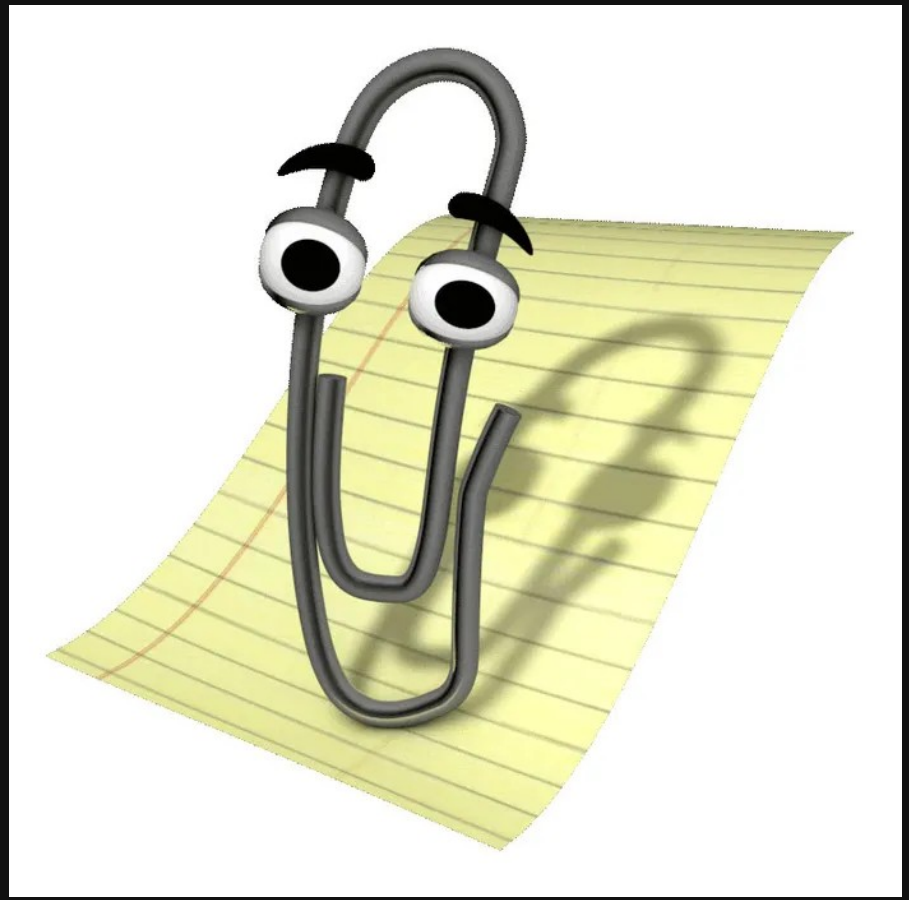



Finder File Edit View Go Window Help
About This Mac
Get Mac OS X Software...
System Preferences...
Dock
Location
Recent Items
Force Quit...
Sleep
Restart
Shut Down
Log Out...
Turn Magnification On
Turn Hiding On
Dock Preferences...
Welcome to Mac OS X
midnite
Desktop
Documents
Library
Movies
Music
Pictures
Public
Sites
Mac OS X Install CD


Microsoft PowerPoint
File Edit View Insert Format Tools Draw Window Help
Times New Roman
24
Presentation
Text Tool
Click to add title
Click to add sub-title
Picks up the tool for creating text objects.


Microsoft Word - Document1


Microsoft Excel - Book1
About Microsoft Excel
Microsoft® Excel 97 SR-1
Copyright© 1985-1997 Microsoft Corporation. All rights reserved.
International CorrectSpell™ spelling correction system
©1993 INSO Corporation. All rights reserved.
Microsoft Map Program. Copyright©
1995-1996 Mapinfo Corporation. All rights reserved.
This product is licensed to:
User
Company
Product ID: 53144-111-1111111-69702
Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.
OK
System Info...
Tech Support
Sheet1 Sheet2 Sheet3
Ready
Start
My Computer
H:\
Microsoft Excel - Book1
3:29 AM



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...

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

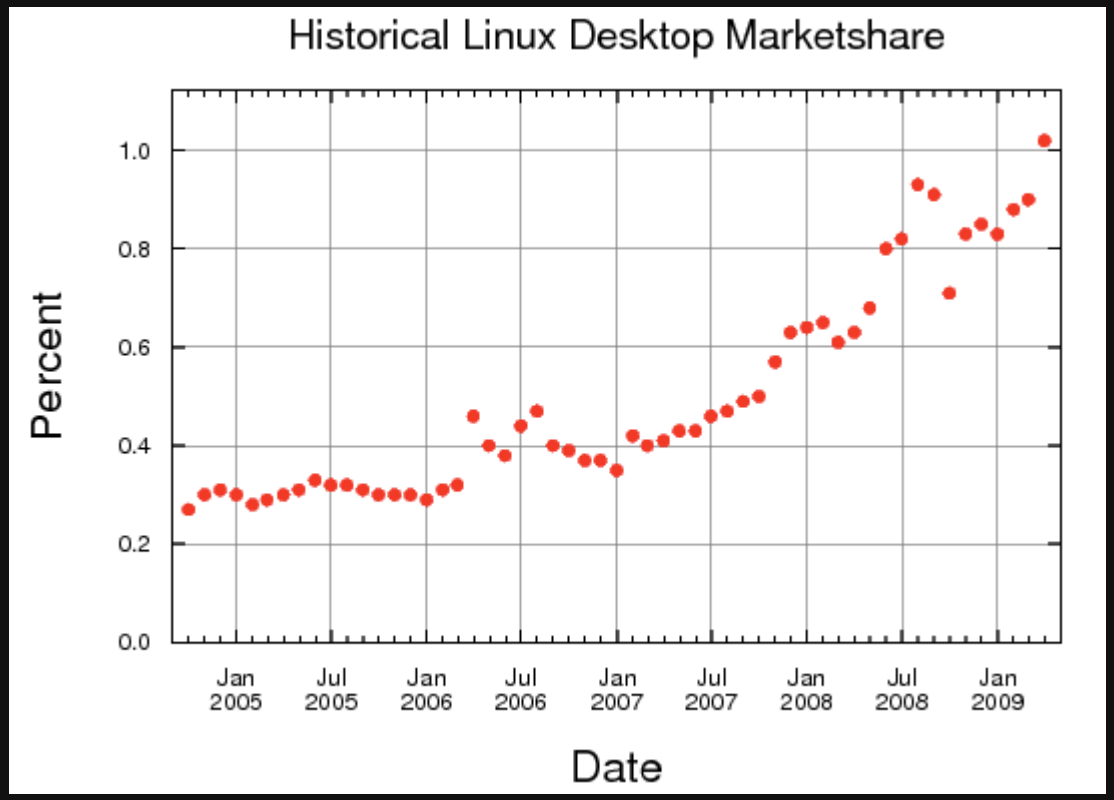
Not everyone at Microsoft was fan

‘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 2.38% at the time of writing)

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

Satya Nadella, current CEO Microsoft

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

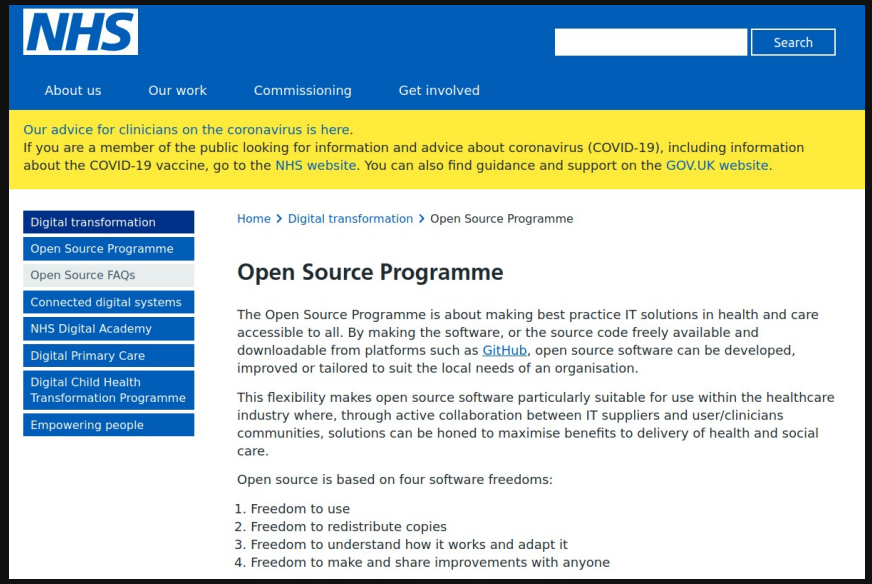
2018: Microsoft switch from Windows to Linux for new ‘Internet of Things’ (IoT) operating system

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 :))

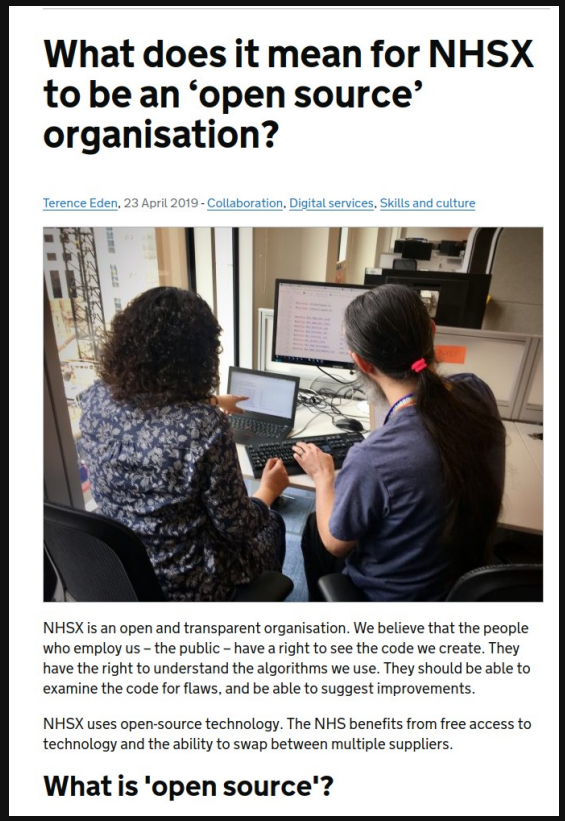
PenCHORD FOSS History

And things are changing in the NHS too…



NHS England Open Source Programme

https://bit.ly/3fLNeTI



NHSX as an Open Source Organisation

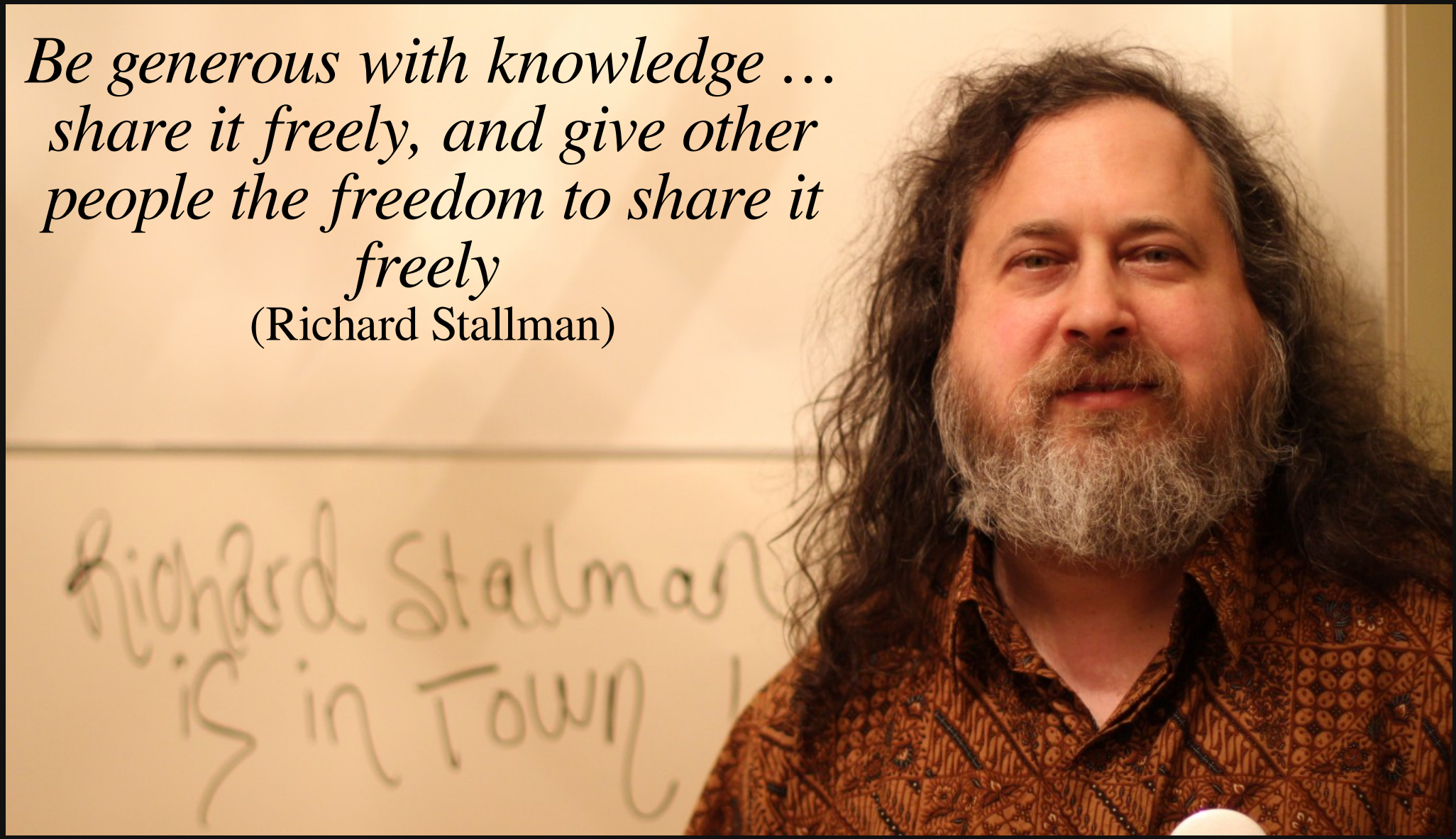
https://bit.ly/3g8I59B

"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)
Richard Stallman is in Town!

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.

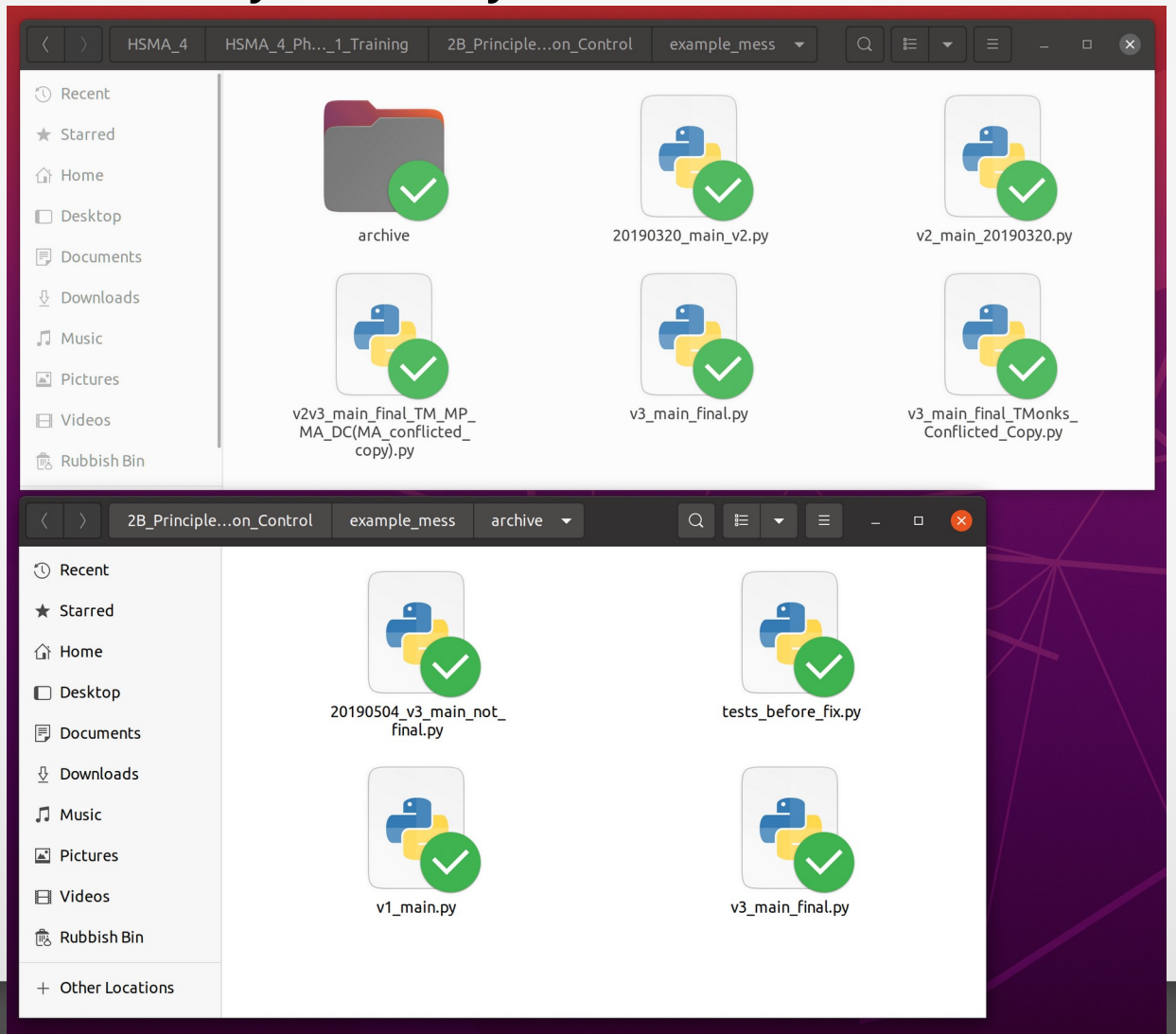
Version Control

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

Why is this important? Let’s consider some examples.

Version Control

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



Version Control

You have been give 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...

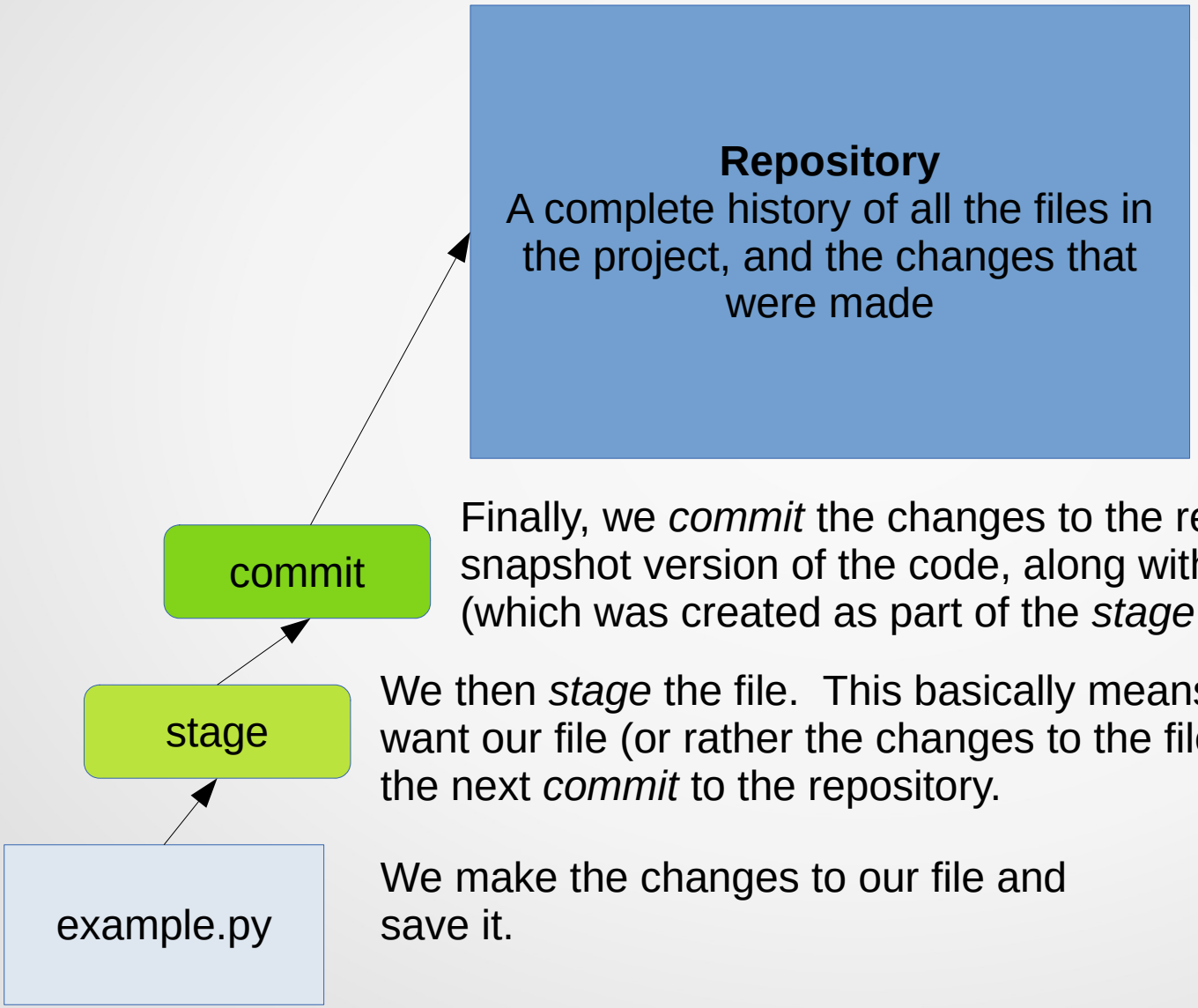
Git

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



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

```python
#!/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")
```

```python
#!/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
```

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

```
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)
```

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

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 (email redacted)
Date:   Thu Aug 12 15:26:13 2021 +0100

    MY_CALC: +subtraction function

commit a5c8a37b13552ae02604a3c9abd7277fae04a958
Author: hsma_master (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 (email redacted)
Date:   Thu Aug 12 15:45:50 2021 +0100

    MY_CALC: +multiply function

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

    MY_CALC: +subtraction function

commit a5c8a37b13552ae02604a3c9abd7277fae04a958
Author: hsma_master (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 :

```
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
+
+    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 (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 (email redacted)
Date:   Thu Aug 12 15:45:50 2021 +0100

    MY_CALC: +multiply function

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

    MY_CALC: +subtraction function

commit a5c8a37b13552ae02604a3c9abd7277fae04a958
Author: hsma_master (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.

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 as the *Master* branch (note – this name is soon going to change to the *Main* branch). 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.

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 *master* 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
* master
```

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

```
git switch dev
```

```
Switched to branch 'dev'
* dev
  master
```

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 master branch. So let's access our history :

```
git log --oneline
```

```
6eea861 (HEAD -> dev, master) Revert "MY_CALC: +multiply function"
5922e79 MY_CALC: +multiply function
a988831 MY_CALC: +subtraction function
a5c8a37 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 *master* branch, so it's safe.

We can see above we need to revert back to commit 5922e79. 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 5922e79 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 8c12420134fbde68ece2ba6418ef6970fe8bfa36 (HEAD -> dev)
Author: hsma_master (email redacted)
Date:   Tue Aug 17 16:52:30 2021 +0100

    FIX: multiply_numbers() patched

commit 6eea86119d42fcc378bdace354fa769d4f41a4c7 (master)
Author: hsma_master (email redacted)
Date:   Thu Aug 12 16:52:11 2021 +0100

    Revert "MY_CALC: +multiply function"

    This reverts commit 5922e7945c8237b5d1912b0d2317a2e79ef75d3c.
```

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 *master* branch.

If we *switch* back to the *master* 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 6eea86119d42fcc378bdace354fa769d4f41a4c7 (HEAD -> master)
Author: hsma_master (email redacted)
Date:   Thu Aug 12 16:52:11 2021 +0100

    Revert "MY_CALC: +multiply function"

    This reverts commit 5922e7945c8237b5d1912b0d2317a2e79ef75d3c.
```

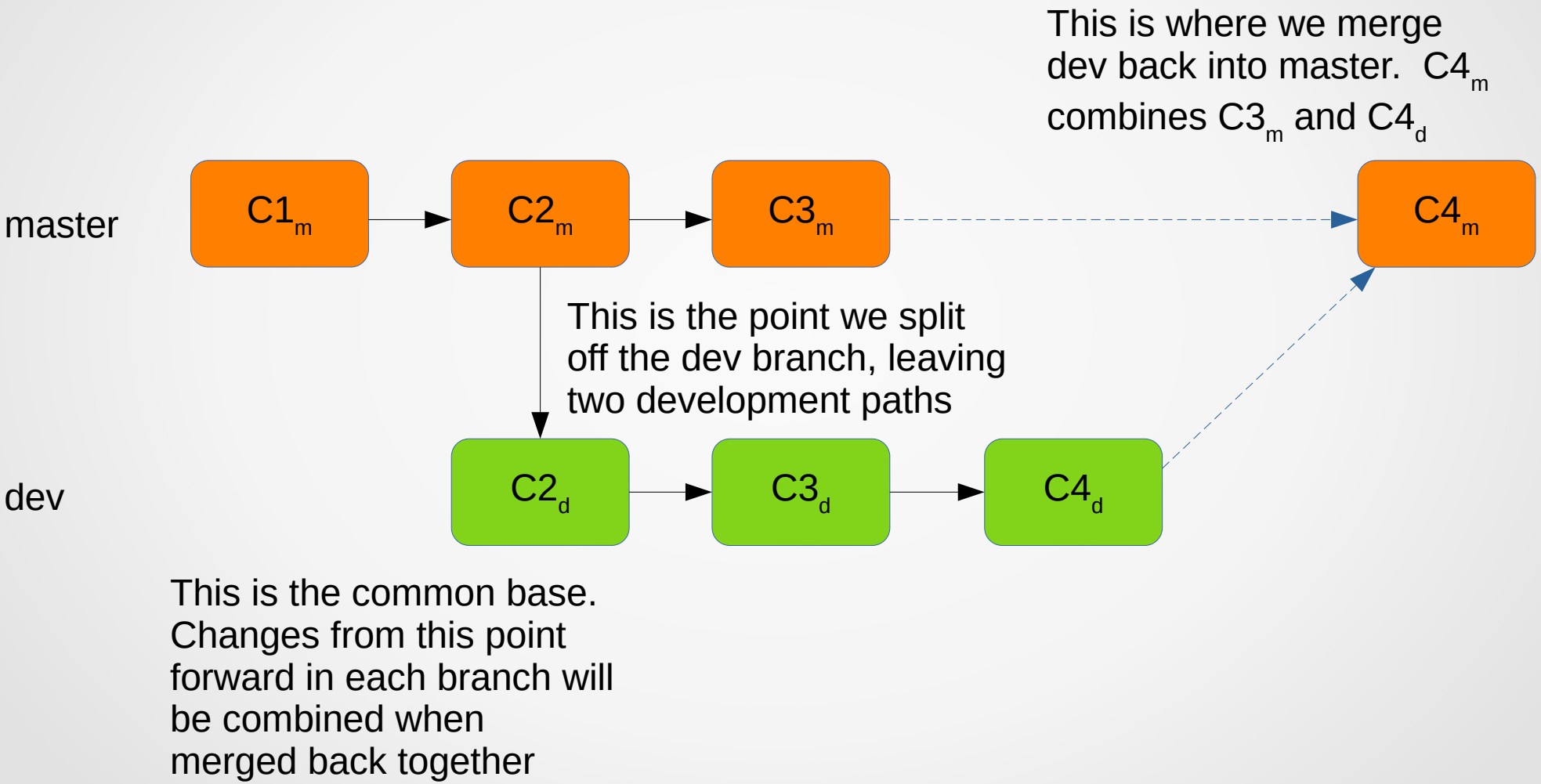
Merge

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 *master* 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


This is where we merge dev back into master. C4m combines C3m and C4d
master
C1m
C2m
C3m
C4m
This is the point we split off the dev branch, leaving two development paths
dev
C2d
C3d
C4d
This is the common base. Changes from this point forward in each branch will be combined when merged back together

Merge

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

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

```
git switch master
```

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 6eea861..8c12420
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 *master* branch to see what's happened :

```
git log -3
```

```
commit 8c12420134fbde68ece2ba6418ef6970fe8bfa36 (HEAD -> master, dev)
Author: hsma_master (email redacted)
Date:   Tue Aug 17 16:52:30 2021 +0100

    FIX: multiply_numbers() patched

commit 6eea86119d42fcc378bdace354fa769d4f41a4c7
Author: hsma_master (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 (email redacted)
Date:   Thu Aug 12 15:45:50 2021 +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 *master* 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 is where we added the buggy multiply function

Merge

And if we go back to Spyder in the *master* 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 :

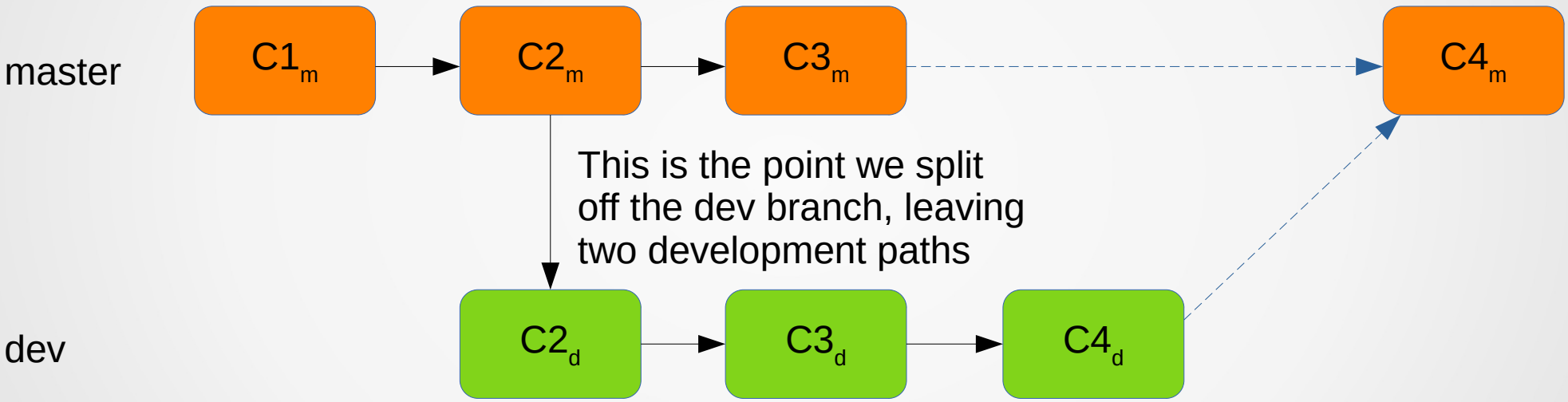
- the branches cannot be merged, because there are pending changes in the branch into which the changes are trying to be merged that could be overwritten by the changes we're trying to merge in. In this situation, Git will refuse the merge until the "local state" can be "stabilised" (e.g. pending changes 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

This is where the conflict will happen – when we try to merge back into master



If changes have been made to the same lines of code in $C3_d$ / $C4_d$ and $C3_m$ there will be a conflict when we try to merge back together. There will **not** be a conflict if, for example, $C3_d$ / $C4_d$ changed lines of code that were not changed in $C3_m$

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 master
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 c78082340dd03e9d3c419417161b6dc2a8da44f7 (HEAD -> master)
Author: hsma_master <d.chalk@exeter.ac.uk>
Date:   Wed Aug 18 15:44:09 2021 +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 *master* branch. So I'll switch to *dev*, but then I'll merge *master* into *dev*, so I can work from this latest version of my project in dev.

```
git switch dev
```

```
git merge master
```

```
Updating 8c12420..c780823
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 to add to subtract operation"
```

Merge Conflicts

Now I'm going to switch back to the *master* 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 the 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 *master*.

Merge Conflicts

Now we'll try merging the work in the *dev* branch into the *master* 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 master
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 *master* 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 *master* 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 taking what was added to the *master* 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 master
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 3f8f64d409bfeba4960b84c160f7ba3382b4546e (HEAD -> master)
Merge: bc1935a 31ee9c1
Author: hsma_master <d.chalk@exeter.ac.uk>
Date:   Wed Aug 18 16:58:48 2021 +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 *master* 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 *master* 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 add 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.

readme.md

```
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
```

Further Reading

Today, we're just covering the basics of version control and Git 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 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)

Exercise 2

You will now have a chance to practice the basics of Git we've just covered. Follow the instructions in the file 2b_exercise_2.pdf. You'll be put back into your breakout rooms, and you can choose to work individually or as group, but either way, make sure you use your group if you get stuck.

You have until the end of the session.