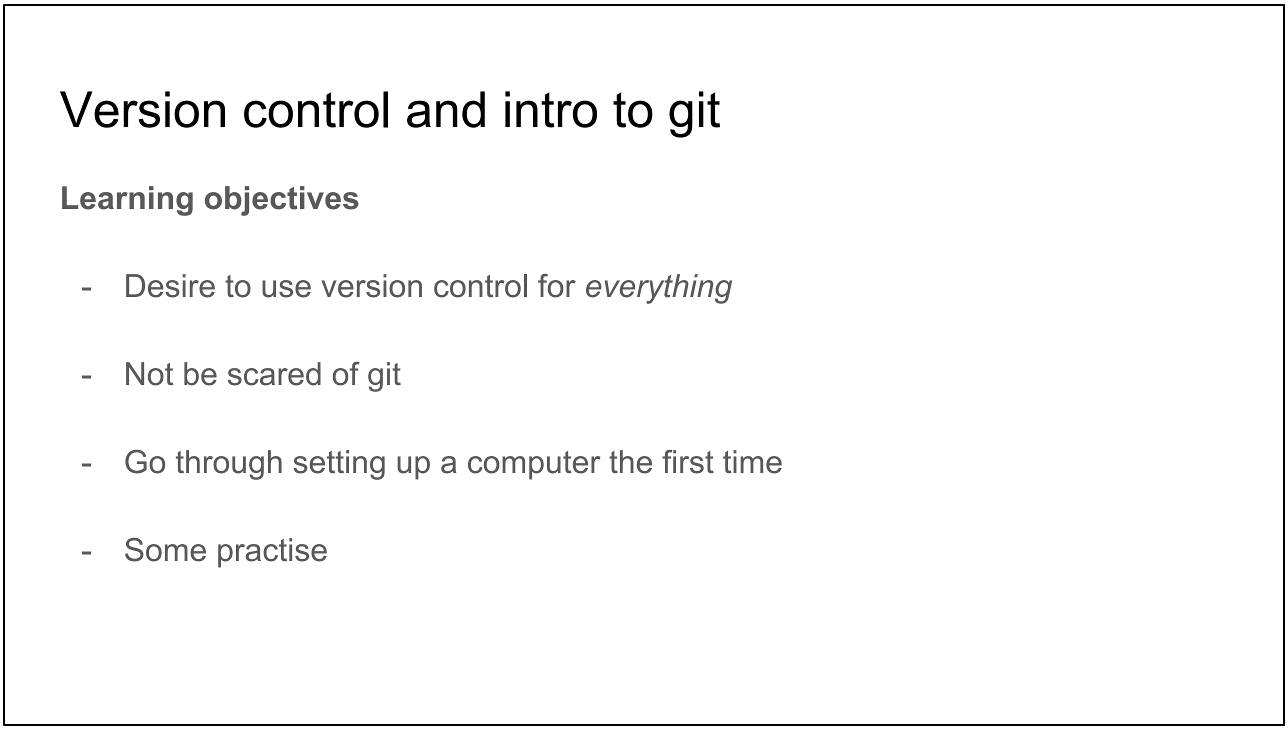
Version control

why you should want it



Version control and intro to git

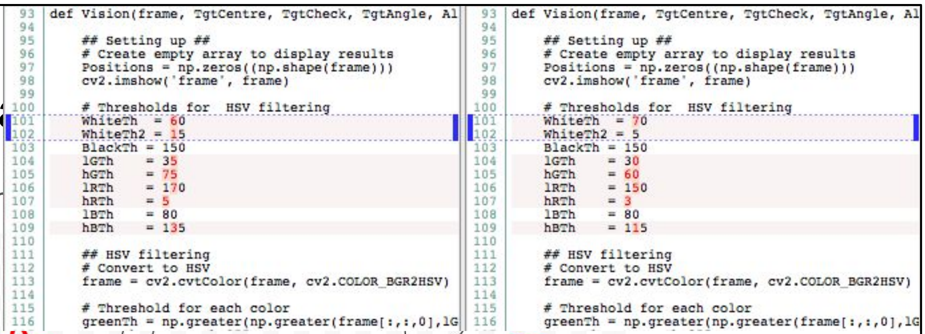
Learning objectives

- Desire to use version control for *everything*
- Not be scared of git
- Go through setting up a computer the first time
- Some practise

I’m not going to teach you the commands you need to use git. (I may mention some in passing). There are tutorials for that. Use them.
I’m hopefully going to inspire you to want to use git *for your own work*.
Most people think version control is necessary for collaboration. The person you will collaborate with the most is confused FUTURE YOU and tired, stressed PAST YOU.
90% of the benefits of version control will apply to lone coding as much as collaborative coding.
CAVEAT: git is powerful. Super powerful. Don’t be put off. It’s very hard to go wrong in an unresolvable way.
I’m not going to mention github until the very end. Everything can (and should) be done on your own computer.

An inspiration

(based in a parallel ur



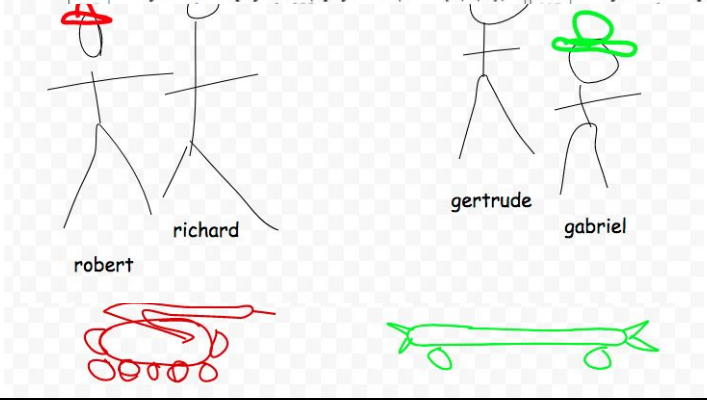
```
def Vision(frame, TgtCentre, TgtCheck, TgtAngle, Al

    ## Setting up ##
    # Create empty array to display results
    Positions = np.zeros((np.shape(frame)))
    cv2.imshow('frame', frame)

    # Thresholds for  HSV filtering
    WhiteTh  = 60
    WhiteTh2 = 15
    BlackTh = 150
    lGTh     = 35
    hGTh     = 75
    lRTh     = 170
    hRTh     = 5
    lBTh     = 80
    hBTh     = 135

    ## HSV filtering
    # Convert to HSV
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Threshold for each color
    greenTh = np.greater(np.greater(frame[:,:,0],lG
```

```
def Vision(frame, TgtCentre, TgtCheck, TgtAngle, Al

    ## Setting up ##
    # Create empty array to display results
    Positions = np.zeros((np.shape(frame)))
    cv2.imshow('frame', frame)

    # Thresholds for  HSV filtering
    WhiteTh  = 70
    WhiteTh2 = 5
    BlackTh = 150
    lGTh     = 30
    hGTh     = 60
    lRTh     = 150
    hRTh     = 3
    lBTh     = 80
    hBTh     = 115

    ## HSV filtering
    # Convert to HSV
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Threshold for each color
    greenTh = np.greater(np.greater(frame[:,:,0],lG
```

MONDAY

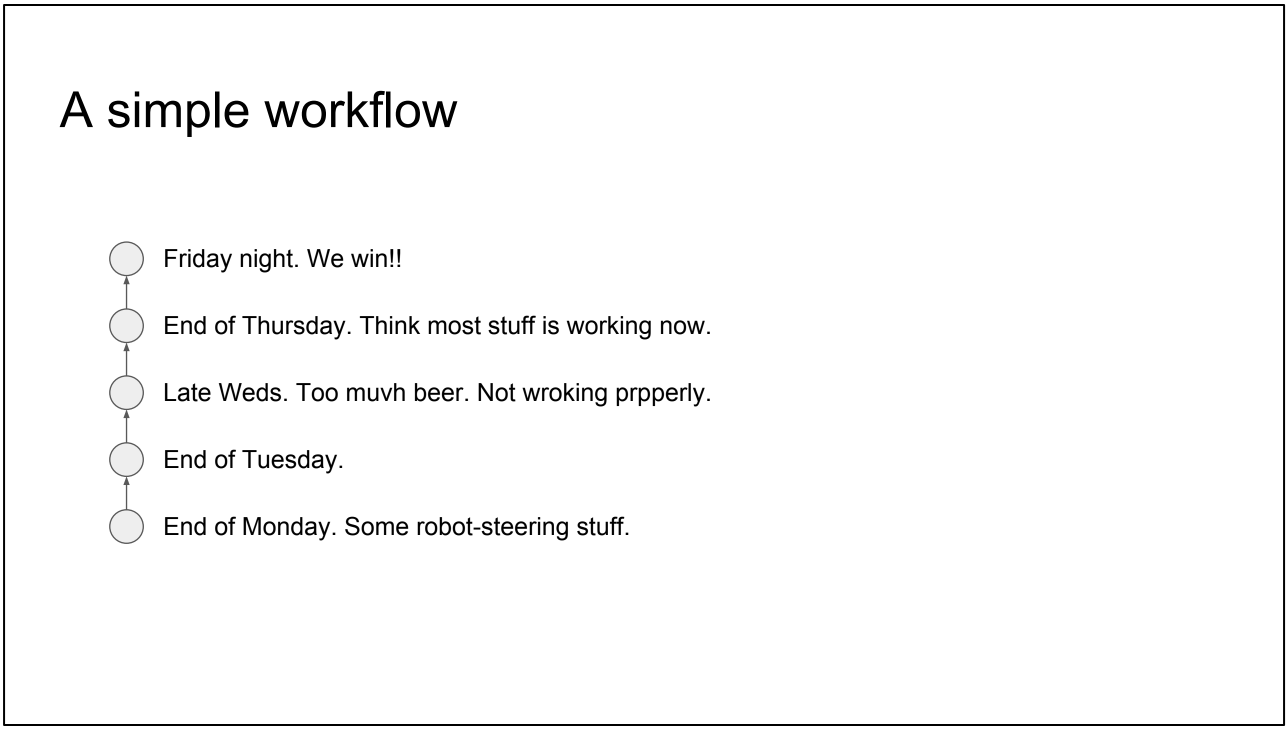
TUESDAY

WEDNESDAY



THURSDAY

FRIDAY
Competition day!



A simple workflow

Friday night. We win!!

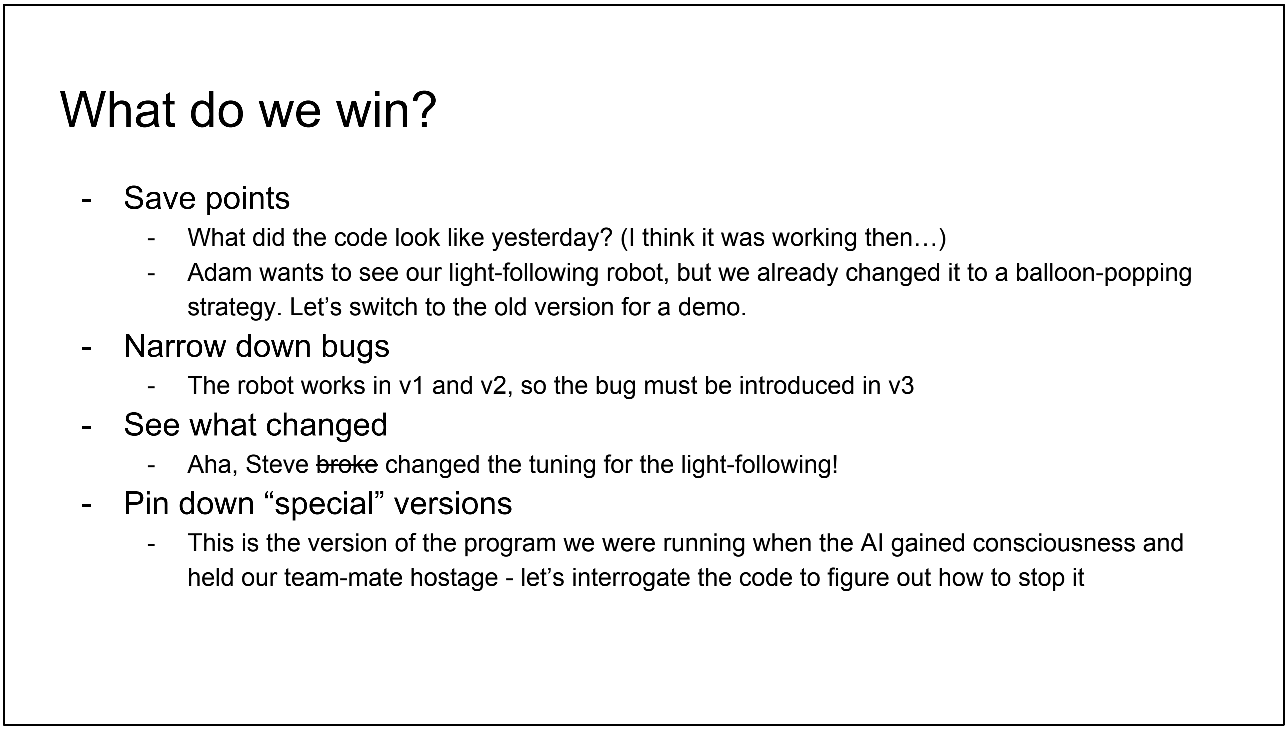
End of Thursday. Think most stuff is working now.

Late Weds. Too muvh beer. Not wroking prpperly.

End of Tuesday.

End of Monday. Some robot-steering stuff.

- This is a snapshot stored in a ZIP file. Later we’ll consider it a git COMMIT.
- This allows us to answer some of the motivating questions but not all. If every version *works* then you can narrow a bug down to a day’s work. If they don’t, it may be several days.

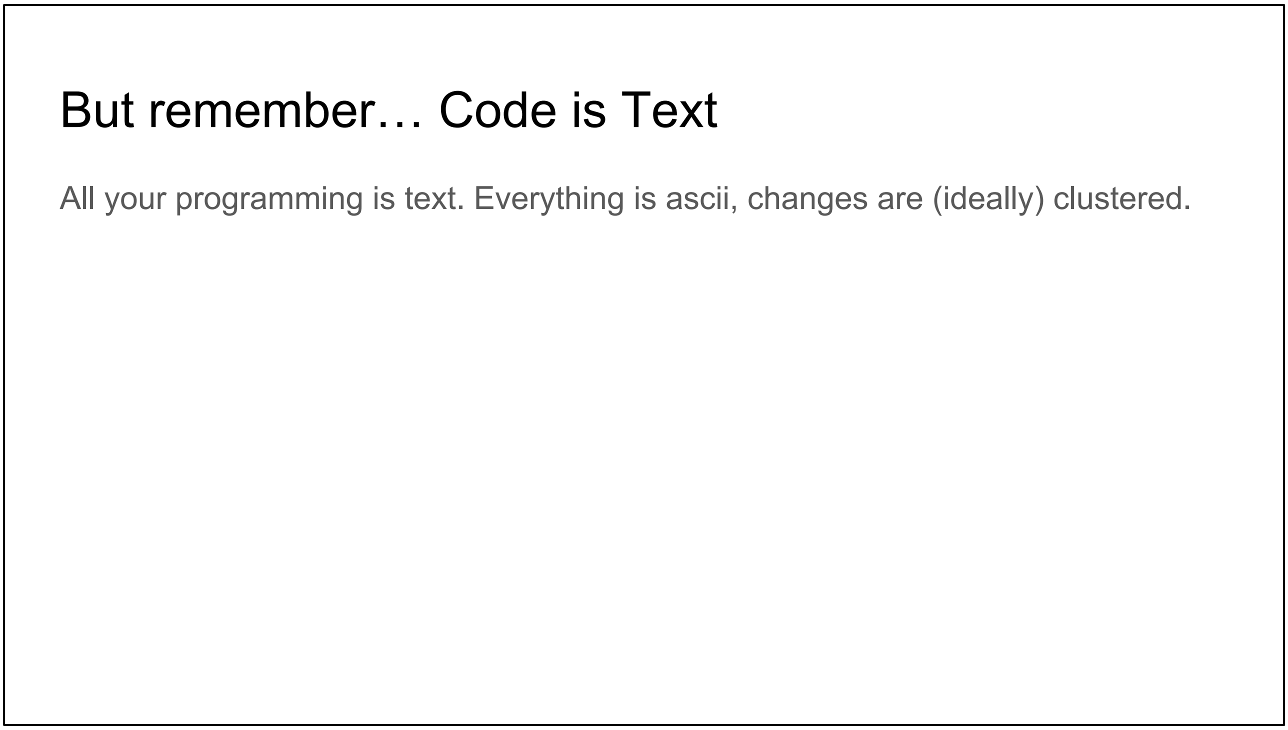


What do we win?

- Save points
    - What did the code look like yesterday? (I think it was working then…)
    - Adam wants to see our light-following robot, but we already changed it to a balloon-popping strategy. Let’s switch to the old version for a demo.
- Narrow down bugs
    - The robot works in v1 and v2, so the bug must be introduced in v3
- See what changed
    - Aha, Steve ~~broke~~ changed the tuning for the light-following!
- Pin down “special” versions
    - This is the version of the program we were running when the AI gained consciousness and held our team-mate hostage - let’s interrogate the code to figure out how to stop it

These are all good things… but, I’m hoping to persuade you that if you adopt a different way of thinking about the code you write, you can harness version control for much, much more.

Note here that we haven’t even introduced collaboration and we’re already winning. All arguments for collaboration are applicable to lone coding. The person you collaborate most with in your life is FUTURE YOU



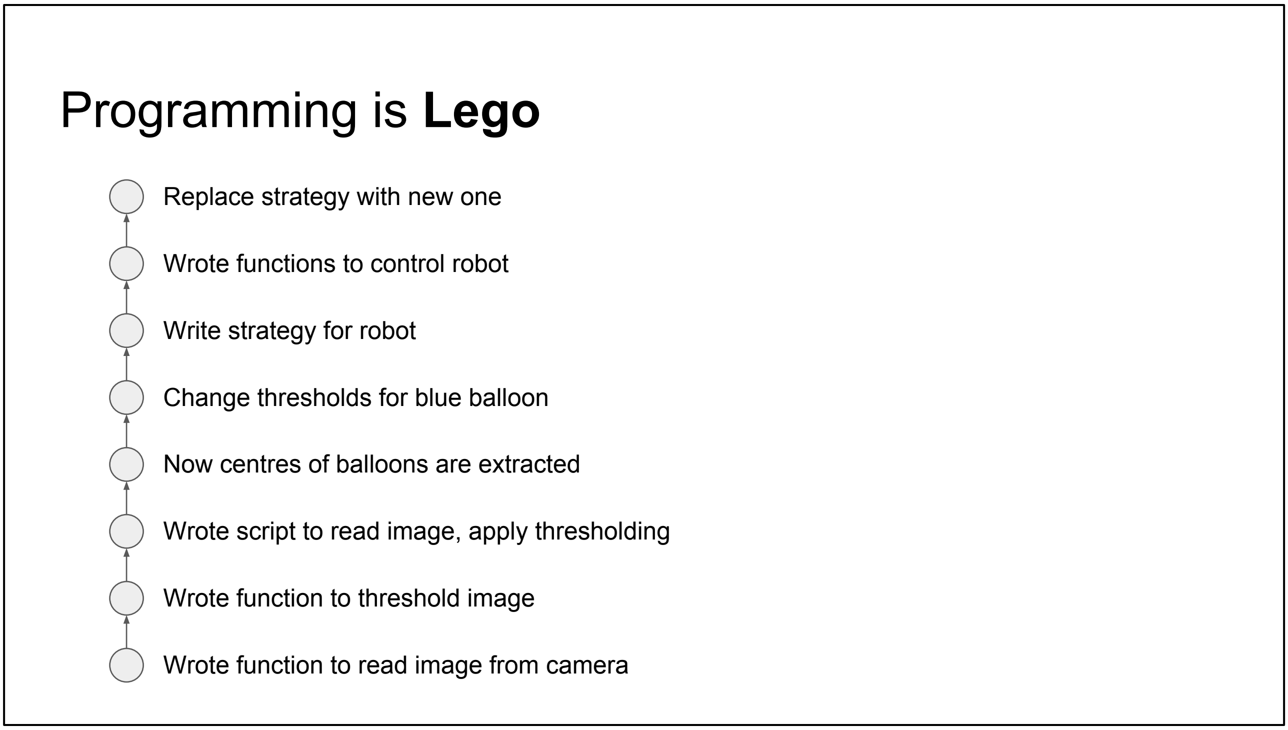
But remember… Code is Text

All your programming is text. Everything is ascii, changes are (ideally) clustered.

(Some file formats are annoying like jupyter notebooks, but they can be worked around.)

If everything is text, it’s very easy to say what has changed at any point. This lets us think more carefully about what we’re actually doing when we’re programming.

Generally you write new words, near other words you’ve recently changed (your changes are clustered)



Programming is **Lego**

Replace strategy with new one

Wrote functions to control robot

Write strategy for robot

Change thresholds for blue balloon

Now centres of balloons are extracted

Wrote script to read image, apply thresholding

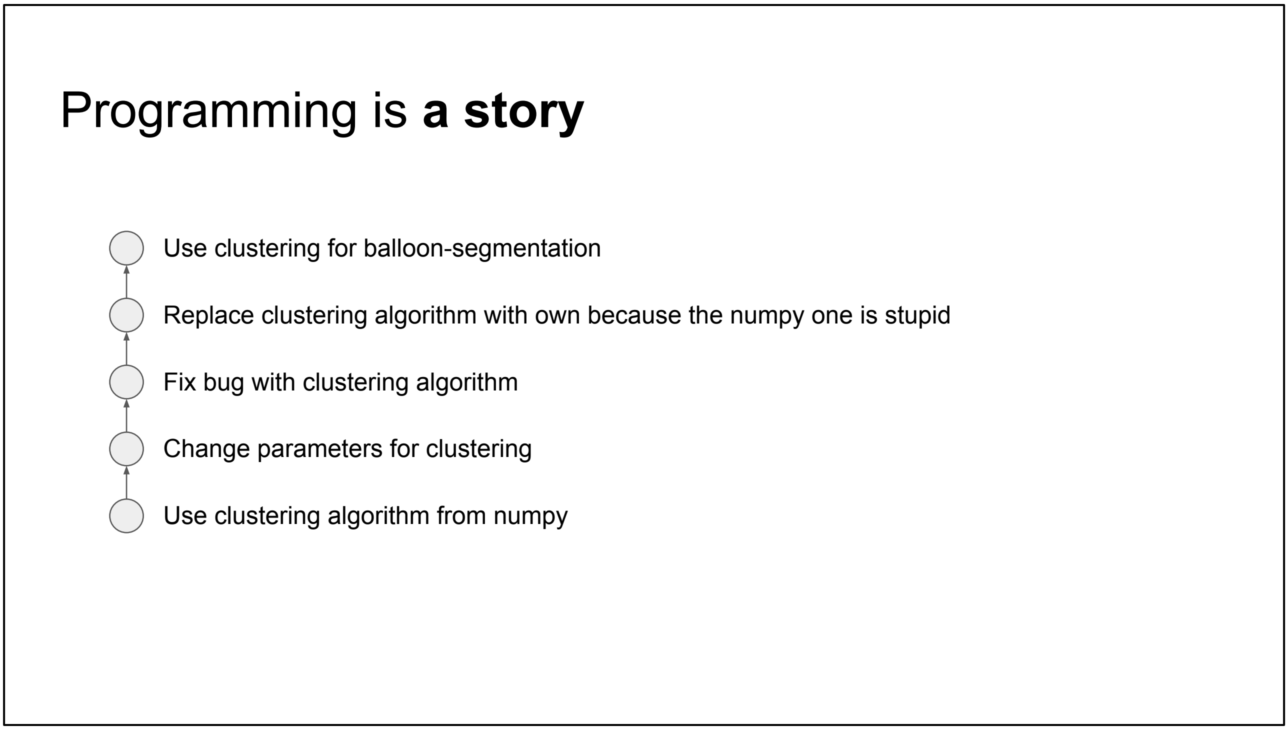
Wrote function to threshold image

Wrote function to read image from camera

… or a narrative
Everything comes down to:

- Add some text
- Remove some text
- Change some text

What if you saved a zip of your entire project every time you did anything? Crazy, right! Git let’s us do this, and it gives us lots of power.



Programming is **a story**

Use clustering for balloon-segmentation

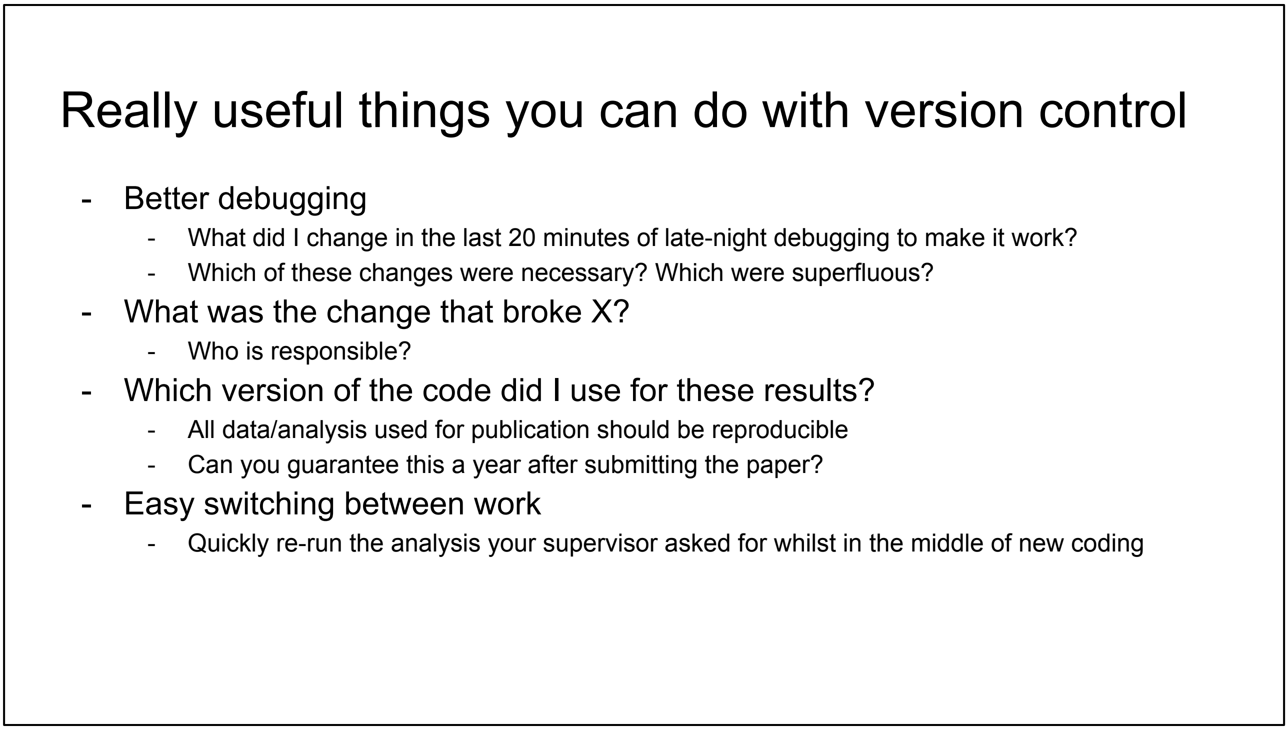
Replace clustering algorithm with own because the numpy one is stupid

Fix bug with clustering algorithm

Change parameters for clustering

Use clustering algorithm from numpy

- Sometimes programming isn’t entirely linear
- This might look embarrassing but it helps future you (or any collaborators)
- The adventure happened, and it’s helpful to document your decisions along the way
- 6 months later - why didn’t I use the built in thresholding algorithm??
- (most arguments for collaborative coding practises also help your most common collaborator: future you)

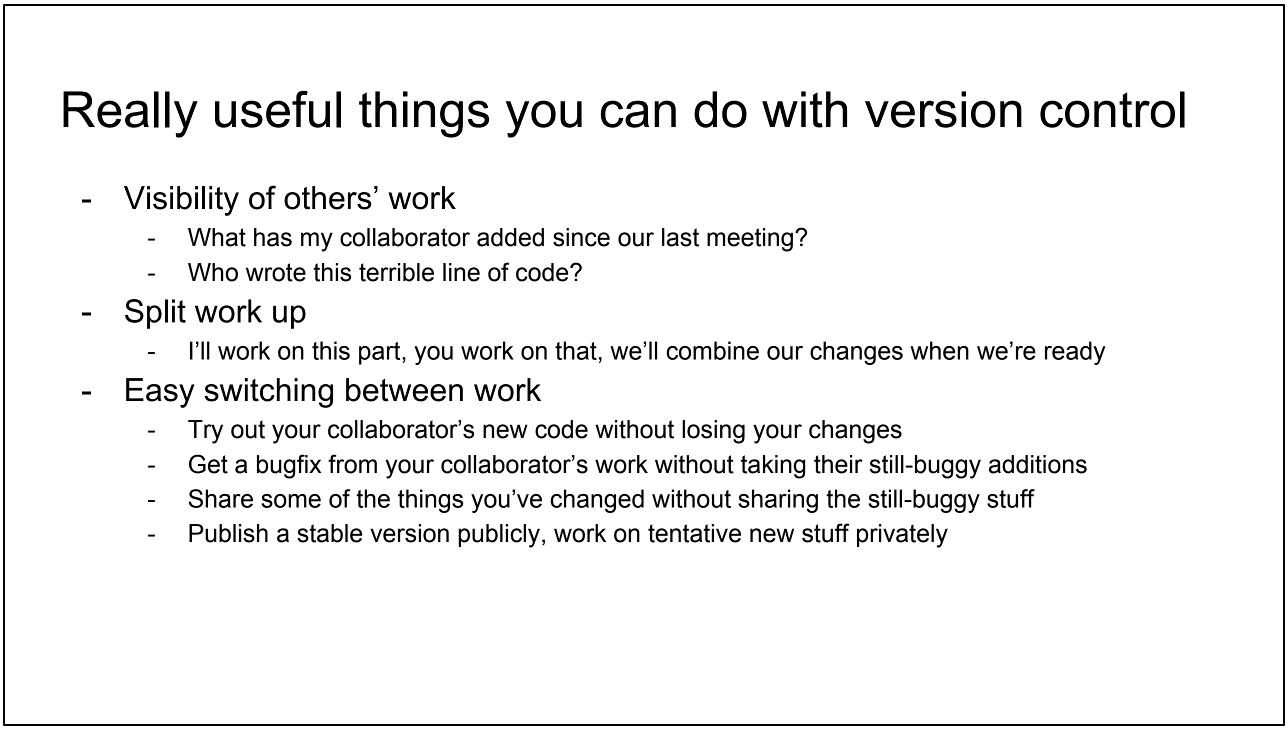


Really useful things you can do with version control

- Better debugging
  - What did I change in the last 20 minutes of late-night debugging to make it work?
  - Which of these changes were necessary? Which were superfluous?
- What was the change that broke X?
  - Who is responsible?
- Which version of the code did I use for these results?
  - All data/analysis used for publication should be reproducible
  - Can you guarantee this a year after submitting the paper?
- Easy switching between work
  - Quickly re-run the analysis your supervisor asked for whilst in the middle of new coding

What if you’re half way through working on the next bit of analysis and your supervisor asks you to rerun some previous analysis with some different parameters? Can you do this?

Label all processed results with a SHA.
Tag all “used” analysis scripts as a release.



Really useful things you can do with version control

- Visibility of others’ work
  - What has my collaborator added since our last meeting?
  - Who wrote this terrible line of code?
- Split work up
  - I’ll work on this part, you work on that, we’ll combine our changes when we’re ready
- Easy switching between work
  - Try out your collaborator’s new code without losing your changes
  - Get a bugfix from your collaborator’s work without taking their still-buggy additions
  - Share some of the things you’ve changed without sharing the still-buggy stuff
  - Publish a stable version publicly, work on tentative new stuff privately

What if you’re half way through working on the next bit of analysis and your supervisor asks you to rerun some previous analysis with some different parameters? Can you do this?

Label all processed results with a SHA.
Tag all “used” analysis scripts as a release.

Let’s set up git

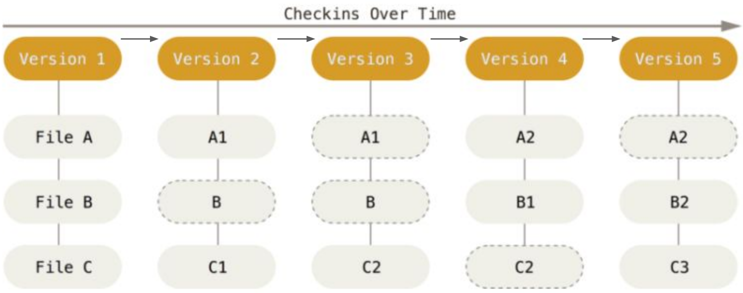
Boring setup, follow through on your laptop and we’ll explain it all later.

https://wiki.ucl.ac.uk/display/SWCM/Git+version+control

- Follow “Git setup for OSX/Windows/Ubuntu”
- Fix the instructions if they aren’t working for you!

How git works

- Git provides a database handling version control for us
  - Stores a zip file with each version of each file
  - .git directory holds all the magic



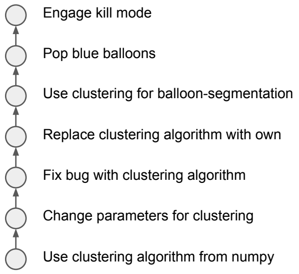
Git is beautifully designed internally, and horrendously designed from a user's perspective.

1) It prioritises doing powerful things over keeping most stuff simple.
2) It is utterly inconsistent in its conventions for command line syntax, and naming of actions.

You just gotta accept that sometimes you need to look things up on stackoverflow or ask a friend.

The most important thing is that you have a picture in your head of what's going on; once that's sorted we can find the magic spell to do what we need to do.

How to use git



Engage kill mode

Pop blue balloons

Use clustering for balloon-segmentation

Replace clustering algorithm with own

Fix bug with clustering algorithm

Change parameters for clustering

Use clustering algorithm from numpy

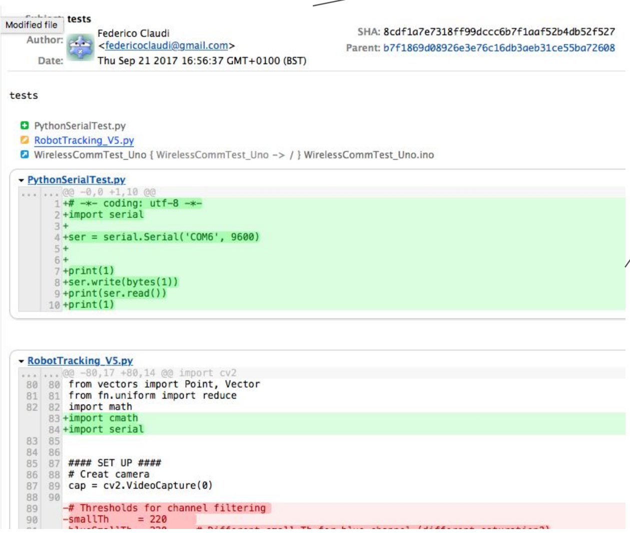
- We need to tell git what to put in each zip file
  - Which files should it “track” ?
  - What changes are relevant to this “commit”
  - Choose what to “stage” (add to a “commit”)
    - Which new files?
    - Which changes to tracked files? (per line)
    - Be aware of .gitignore for files you never want to track
    - (switch to gitx demo)
- Once it’s in git, it is safe (you can only *add* to a git repo)
  - You can “check out” any of your historical versions, and your local files will be updated to reflect that version.
- Think of commits as representing the *changes* to the code

Each commit stores a record of a version of code
But it’s often helpful to view each commit as “what is different to the previous version|. Your GUI should show you this. This also allows you to think of things like “I want to revert this change” or “I only want this change but not the others”. This is where git beats zip files hand down.

Think about:

- What do you want to keep changes of? Generally any code/text
- What is automatically produced and therefore irrelevant? Compiled stuff, put in .gitignore or .gitexcludes
- Text, not binary. Generally avoid large datasets. Keep them elsewhere. Your repo contains every version of every file - if it’s not ASCII, this will quickly get enormous.
  - Maybe consider a script that pulls down the “right” data from elsewhere. There are some wrappers for this (git fat, github has something)

Commits as diffs



Modified file

Subject: tests

Author: Federico Claudi <federicoclaudi@gmail.com>

Date: Thu Sep 21 2017 16:56:37 GMT+0100 (BST)

SHA: 8cdf1a7e7318ff99dccc6b7f1aaf52b4db52f527

Parent: b7f1869d08926e3e76c16db3aeb31ce55ba72608

tests

- PythonSerialTest.py
- RobotTracking_V5.py
- WirelessCommTest_Uno { WirelessCommTest_Uno -> / } WirelessCommTest_Uno.ino

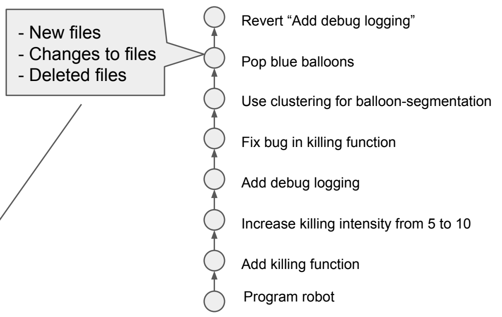
PythonSerialTest.py

```
@@ -0,0 +1,10 @@
+# -*- coding: utf-8 -*-
+import serial
+
+ser = serial.Serial('COM6', 9600)
+
+
+print(1)
+ser.write(bytes(1))
+print(ser.read())
+print(1)
```

RobotTracking_V5.py

```
@@ -80,17 +80,14 @@ import cv2
 from vectors import Point, Vector
 from fn.uniform import reduce
 import math
+import cmath
+import serial


 #### SET UP ####
 # Creat camera
 cap = cv2.VideoCapture(0)

-# Thresholds for channel filtering
-smallTh     = 220
```

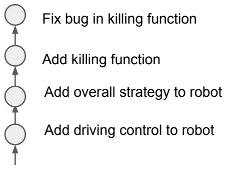
```
git revert 5da72be
```



- New files
- Changes to files
- Deleted files

Revert "Add debug logging"
Pop blue balloons
Use clustering for balloon-segmentation
Fix bug in killing function
Add debug logging
Increase killing intensity from 5 to 10
Add killing function
Program robot

```
git cherry-pick e3ab89ee
```

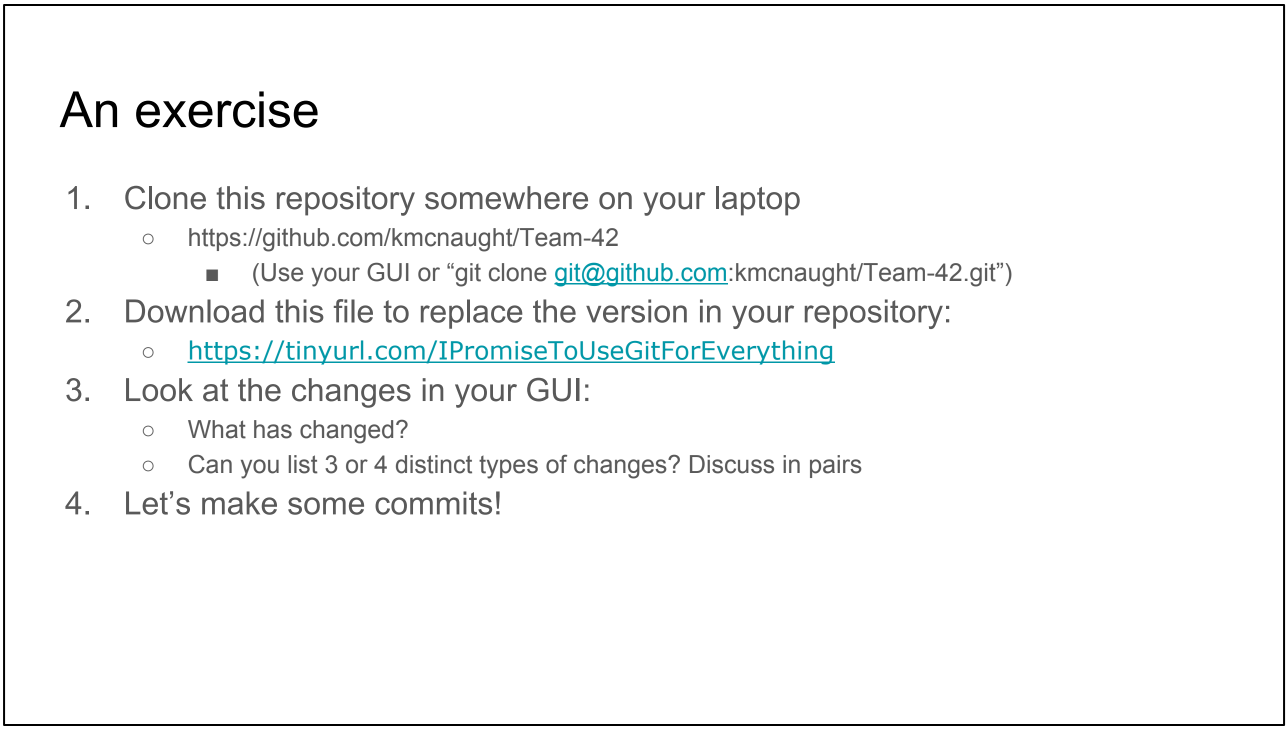


Fix bug in killing function
Add killing function
Add overall strategy to robot
Add driving control to robot

Each commit stores a record of a version of code
But it's often helpful to view each commit as "what is different to the previous version|. Your GUI should show you this. This also allows you to think of things like "I want to revert this change" or "I only want this change but not the others". This is where git beats zip files hand down.

Think about:

- What do you want to keep changes of? Generally any code/text
- What is automatically produced and therefore irrelevant? Compiled stuff, put in .gitignore or .gitexcludes
- Text, not binary. Generally avoid large datasets. Keep them elsewhere. Your repo contains every version of every file - if it's not ASCII, this will quickly get enormous.
  - Maybe consider a script that pulls down the "right" data from elsewhere. There are some wrappers for this (git fat, github has something)



An exercise

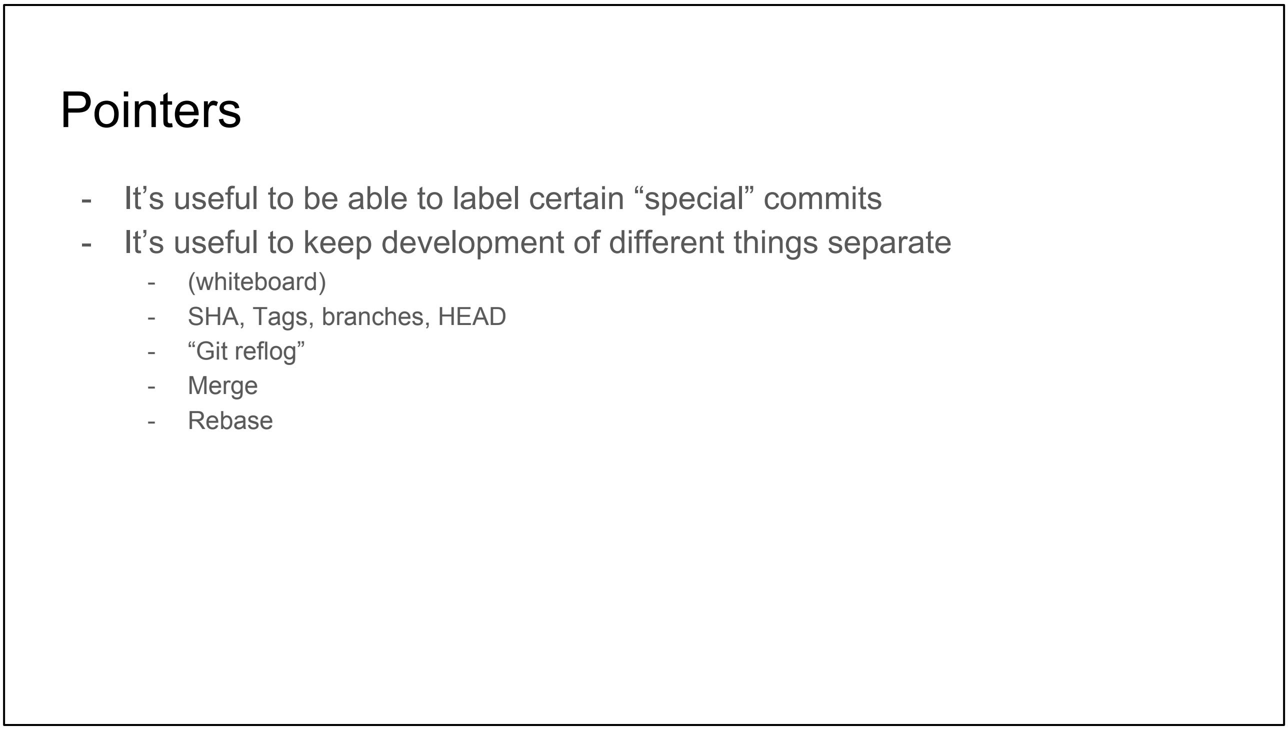
1. Clone this repository somewhere on your laptop
   - https://github.com/kmcnaught/Team-42
     - (Use your GUI or “git clone git@github.com:kmcnaught/Team-42.git”)
2. Download this file to replace the version in your repository:
   - https://tinyurl.com/IPromiseToUseGitForEverything
3. Look at the changes in your GUI:
   - What has changed?
   - Can you list 3 or 4 distinct types of changes? Discuss in pairs
4. Let’s make some commits!

Now let’s discuss what separate changes should be
Why should we keep these separate?

- Maybe logging is temporary? Might want to change back later
- Changes that don’t effect code should be ignored when debugging

Make your 3 or 4 commits using your GUI.

- Now revert the logging one.



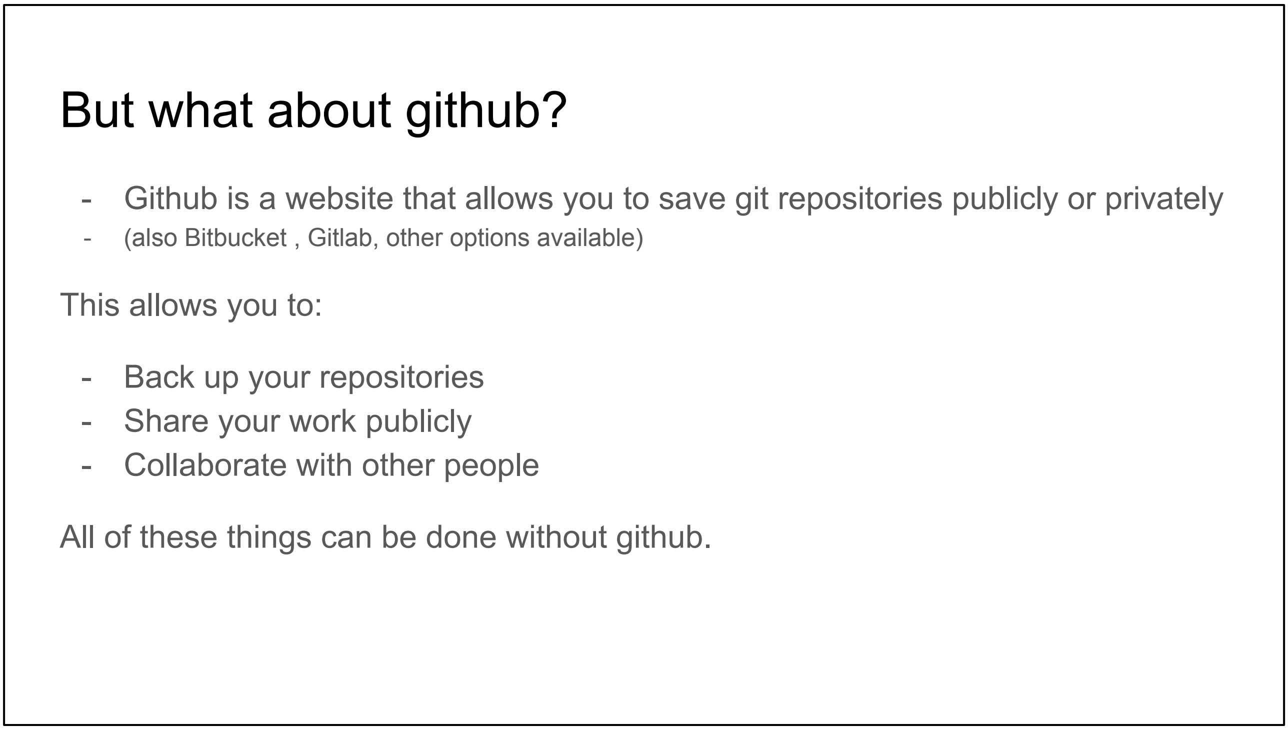
Pointers

- It’s useful to be able to label certain “special” commits
- It’s useful to keep development of different things separate
    - (whiteboard)
    - SHA, Tags, branches, HEAD
    - “Git reflog”
    - Merge
    - Rebase

Up until this point

Reminder

- Programming is Text
- Programming is Lego
- Git stores your changes in a graph.
  - You can manipulate it however you like.
- Branches, tags, are just pointers
- You should never change a line of code without version control
  - It’s really easy to create a local repository and start using it.
  - You will *always* know what you’ve changed



But what about github?

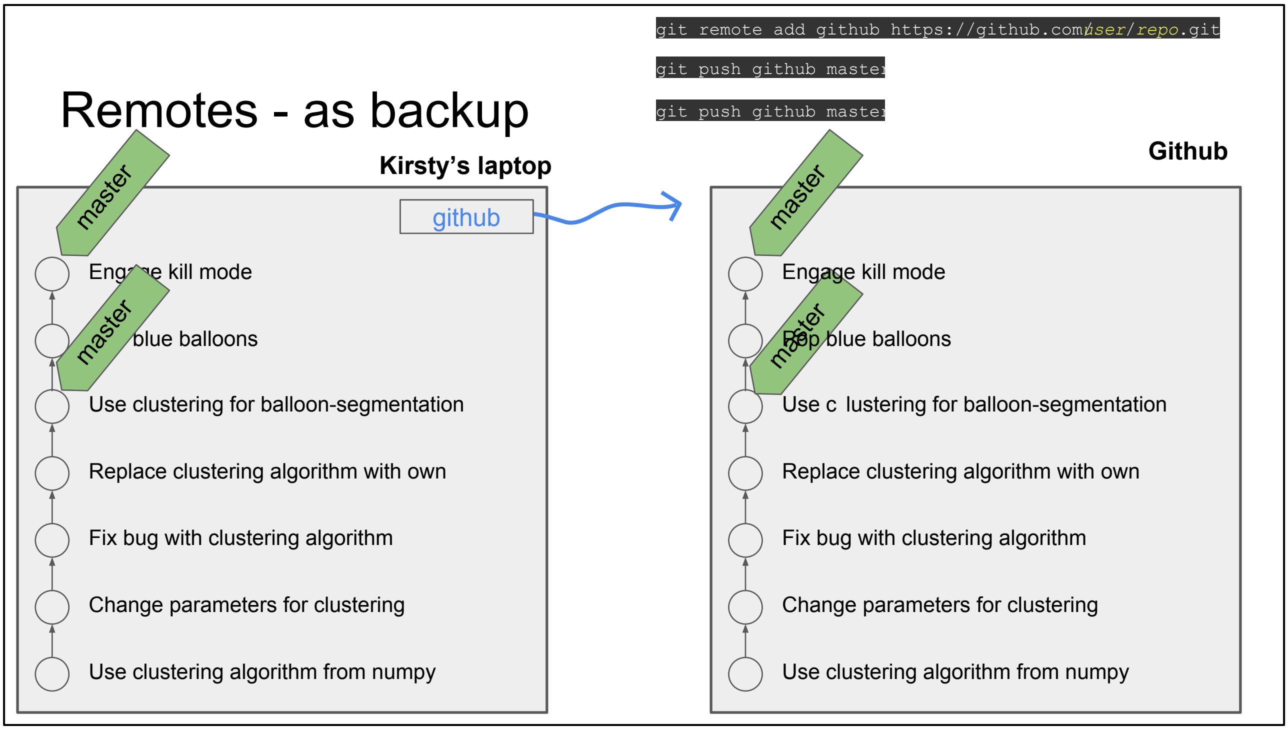
- Github is a website that allows you to save git repositories publicly or privately
- (also Bitbucket , Gitlab, other options available)

This allows you to:

- Back up your repositories
- Share your work publicly
- Collaborate with other people

All of these things can be done without github.

Github (et al) mainly provide a place to store a copy of your .git directory. Sometimes they also give you nice extra ‘project management’ features like comments, code reviews, “pull requests”.



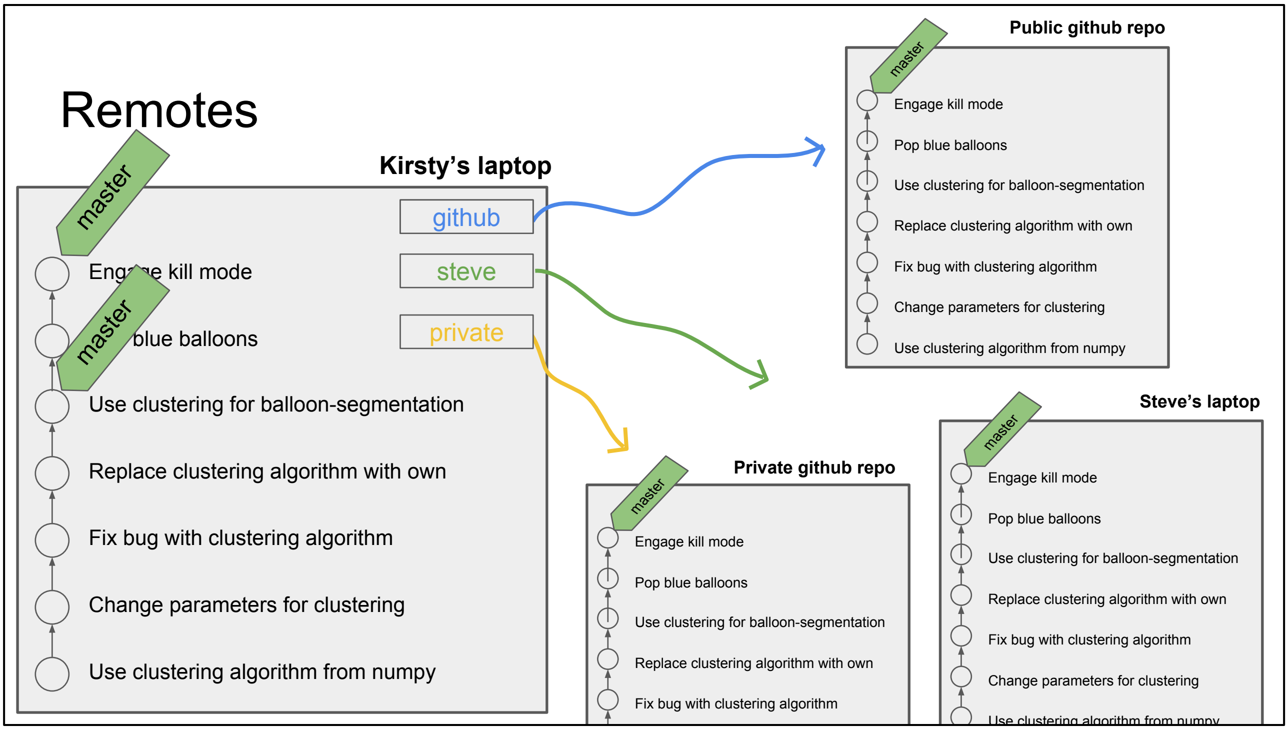
Remotes - as backup

```
git remote add github https://github.com/user/repo.git
git push github master
git push github master
```

Kirsty's laptop
github
master
master
Engage kill mode
Pop blue balloons
Use clustering for balloon-segmentation
Replace clustering algorithm with own
Fix bug with clustering algorithm
Change parameters for clustering
Use clustering algorithm from numpy
Github
master
master
Engage kill mode
Pop blue balloons
Use clustering for balloon-segmentation
Replace clustering algorithm with own
Fix bug with clustering algorithm
Change parameters for clustering
Use clustering algorithm from numpy

Previously we considered this git history.
Let's say we set up the repository on our own laptop
If my laptop gets stolen, I lose everything.
Let's make a copy of it on github…
Create a repo (it's currently empty)
Add a remote pointing to github (

Remotes



Kirsty's laptop

Engage kill mode

Pop blue balloons

Use clustering for balloon-segmentation

Replace clustering algorithm with own

Fix bug with clustering algorithm

Change parameters for clustering

Use clustering algorithm from numpy

Public github repo

Engage kill mode

Pop blue balloons

Use clustering for balloon-segmentation

Replace clustering algorithm with own

Fix bug with clustering algorithm

Change parameters for clustering

Use clustering algorithm from numpy

Private github repo

Engage kill mode

Pop blue balloons

Use clustering for balloon-segmentation

Replace clustering algorithm with own

Fix bug with clustering algorithm

Steve's laptop

Engage kill mode

Pop blue balloons

Use clustering for balloon-segmentation

Replace clustering algorithm with own

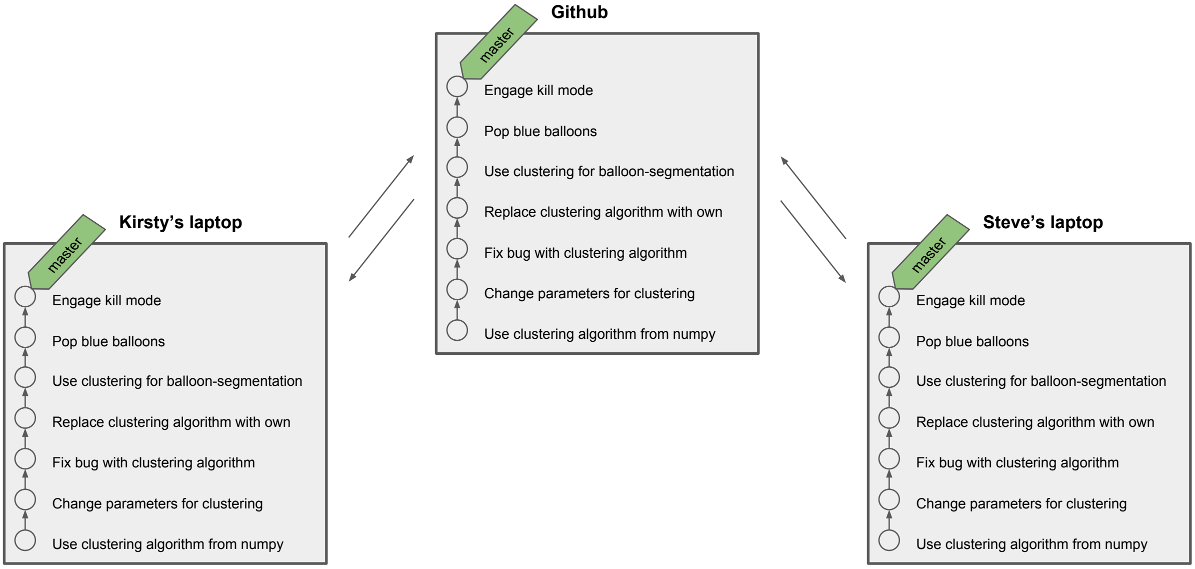
Fix bug with clustering algorithm

Change parameters for clustering

Use clustering algorithm from numpy

There's nothing saying you can't have more than one remote.
Maybe you want to keep cutting edge stuff private, but shared the master branch when it's stable.
There's nothing special about github. A remote just points at a .git repo on another computer. Maybe it's steve's laptop.

Remotes - for collaboration



Github

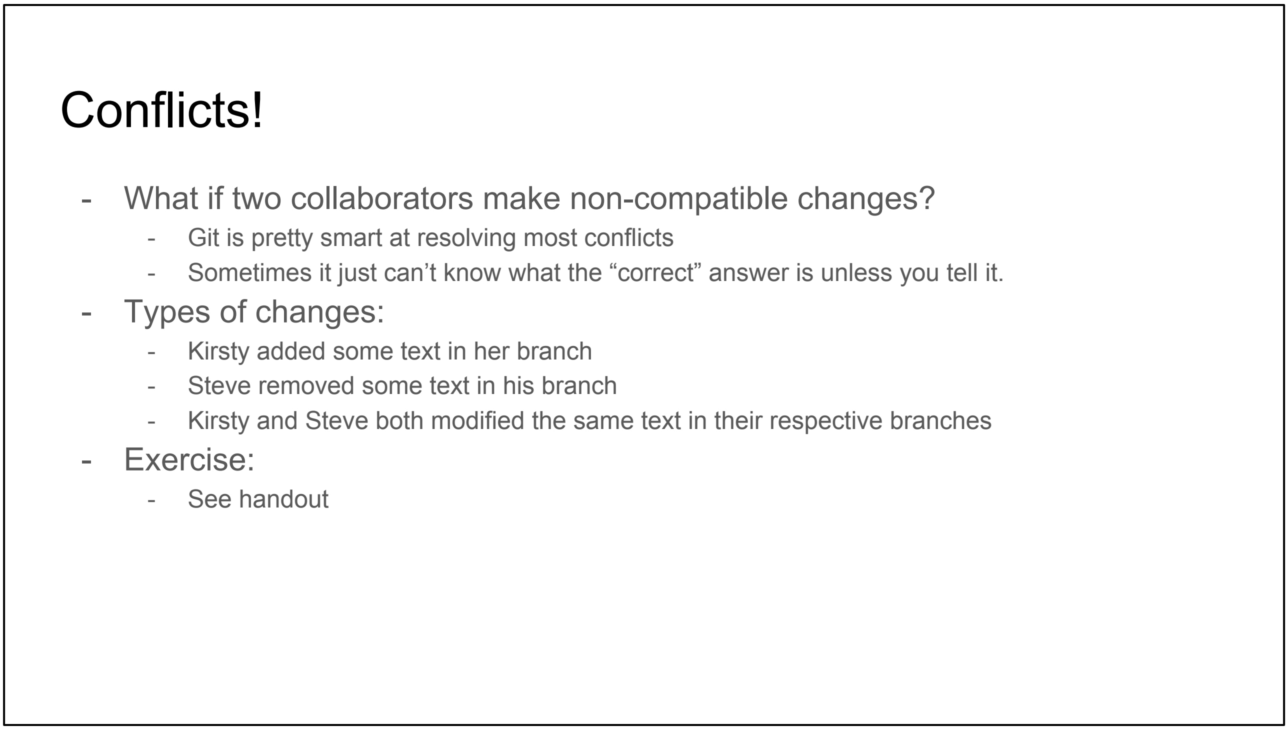
- Engage kill mode
- Pop blue balloons
- Use clustering for balloon-segmentation
- Replace clustering algorithm with own
- Fix bug with clustering algorithm
- Change parameters for clustering
- Use clustering algorithm from numpy

Kirsty’s laptop

- Engage kill mode
- Pop blue balloons
- Use clustering for balloon-segmentation
- Replace clustering algorithm with own
- Fix bug with clustering algorithm
- Change parameters for clustering
- Use clustering algorithm from numpy

Steve’s laptop

- Engage kill mode
- Pop blue balloons
- Use clustering for balloon-segmentation
- Replace clustering algorithm with own
- Fix bug with clustering algorithm
- Change parameters for clustering
- Use clustering algorithm from numpy



Conflicts!

- What if two collaborators make non-compatible changes?
  - Git is pretty smart at resolving most conflicts
  - Sometimes it just can’t know what the “correct” answer is unless you tell it.
- Types of changes:
  - Kirsty added some text in her branch
  - Steve removed some text in his branch
  - Kirsty and Steve both modified the same text in their respective branches
- Exercise:
  - See handout

One of the best bits about git is that it’s pretty smart at handling conflict resolution.
One of the scariest bits about starting to use git is that sometimes you have to figure out conflicts yourself.
Use a good diff tool from the beginning.
If it goes wrong you can try again.

Good commit messages and “units” really help things here - you need to figure out what the other person was trying to achieve.