HDAT9800 Visualisation and Communication of Health Data

Chapter 1 - git and GitHub

Drs James Farrow & Tim Churches

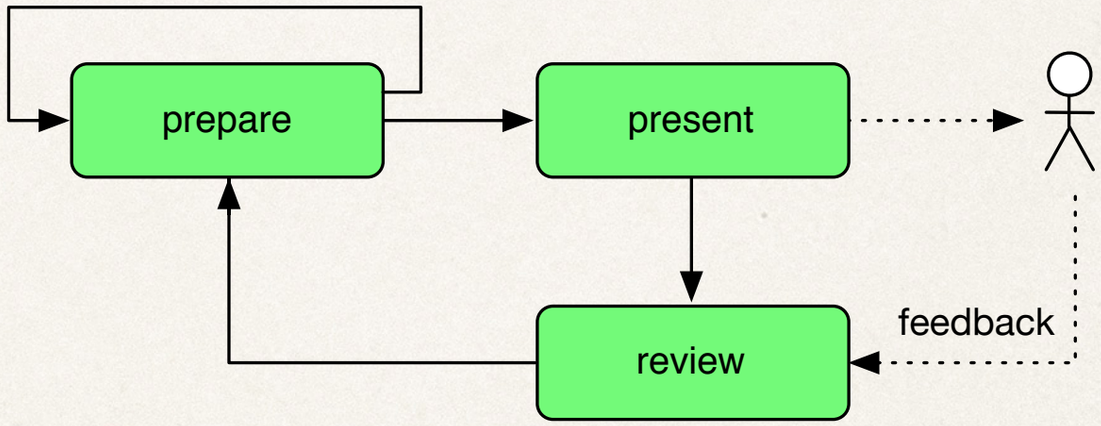
Communication is a process

Communication begins before the presentation

Communication continues after the presentation

An iterative process

- prepare (iterative)
- present
- feedback
- review (iterate back to beginning)



Revision management

We need to manage the artefacts of our communication

Iteration means revision

Revision means multiple versions

The content of any project will change over time

A typical project

Ad hoc management of files

Commented out chunks of old code in scripts

Copies of older versions ‘just in case’

Typically using suffixes on files and directories

- .new, .old, .orig, .new.old, .new.new, .new2
- oldproject, project.jun, project.0410

A real mess

Shift in thinking

Instead of taking copies ‘beside’ one another…

…we’re going to keep a history of changes

This will let us delete ‘old’ documents, data and code

Everything is recoverable

Our projects stay ‘clean’

We ‘save’ every once in a while ‘just in case’, this is no more intrusive than that once you’re used to it

Version control systems

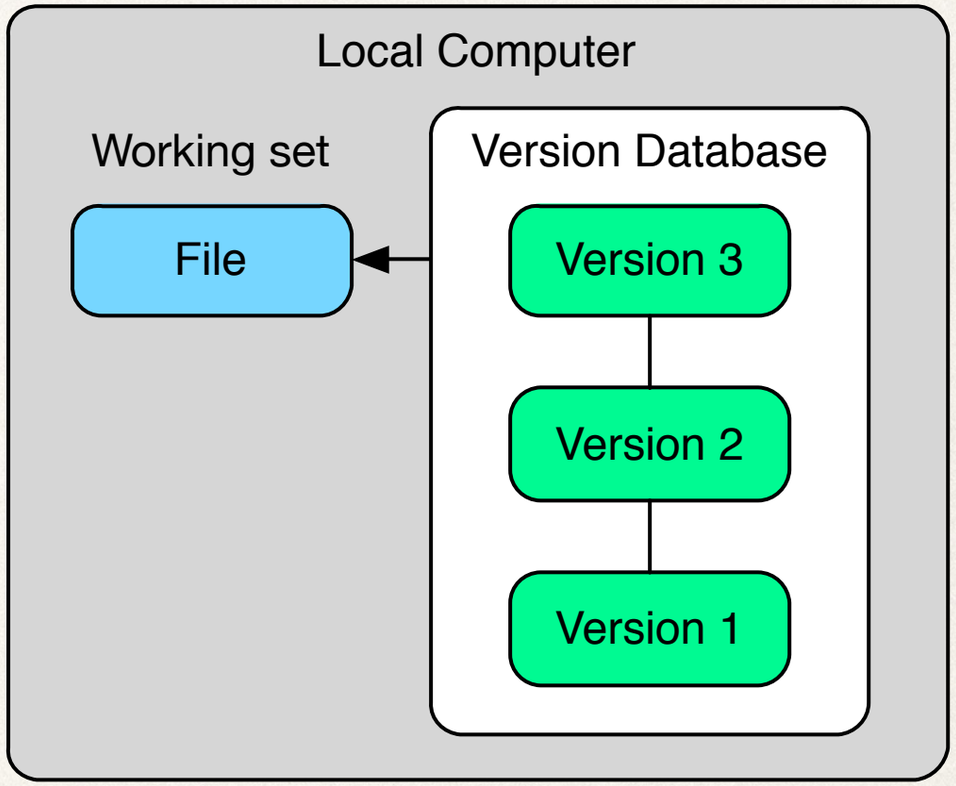
Track changes to directories and files over time

Keeps state apart from working files

Allow previous versions to be recovered

No need for old versions to clutter things up

If mistakes get made ‘roll back’ to good version



http://git-scm.com/book/en/Getting-Started-About-Version-Control

Distributed collaboration

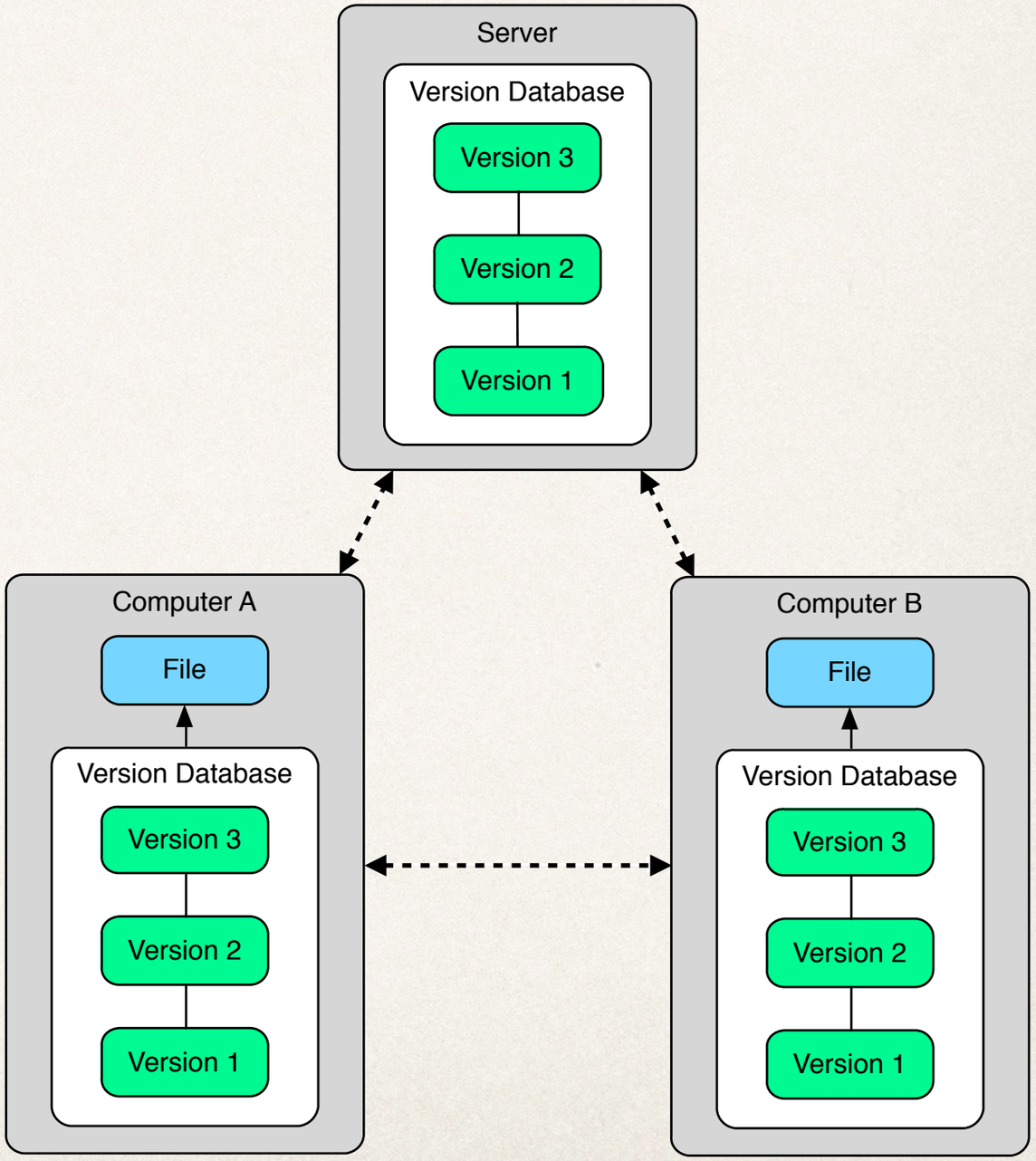
Multiple people in multiple locations with multiple copies

Independent code edits

Share changes

Possibly a central repository

Easy to see who did what when (and roll back)



http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows

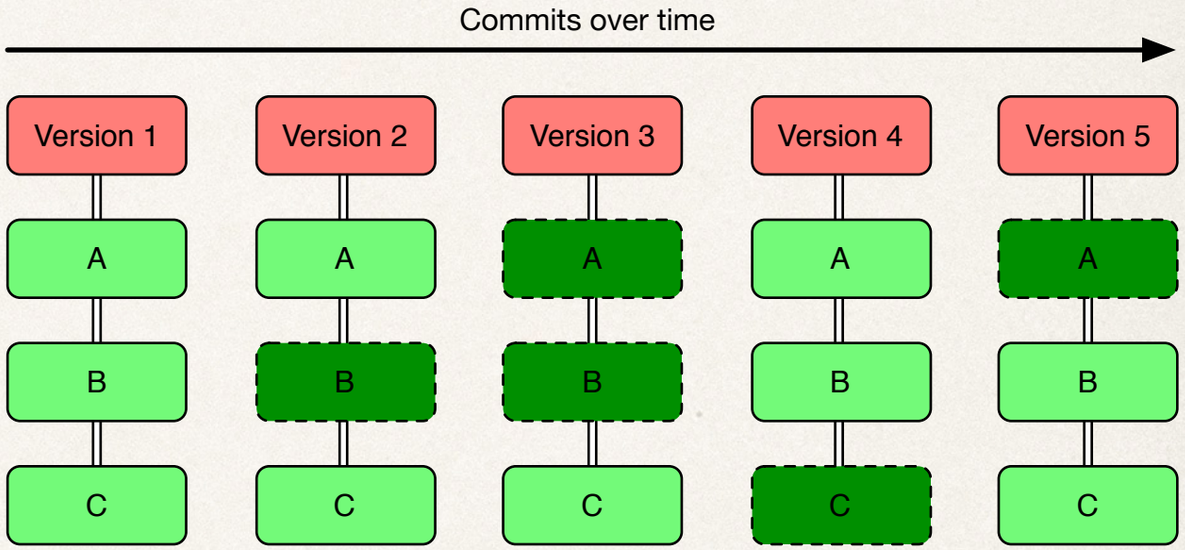
Snapshots

Think of the repository as a series of snapshots (saves)

The repository is like a mini file system

Changes to the repository are scheduled using commands like *add, remove, move* and so on

Snapshots are made using *commit*



http://git-scm.com/book/en/Getting-Started-Git-Basics

File workflow

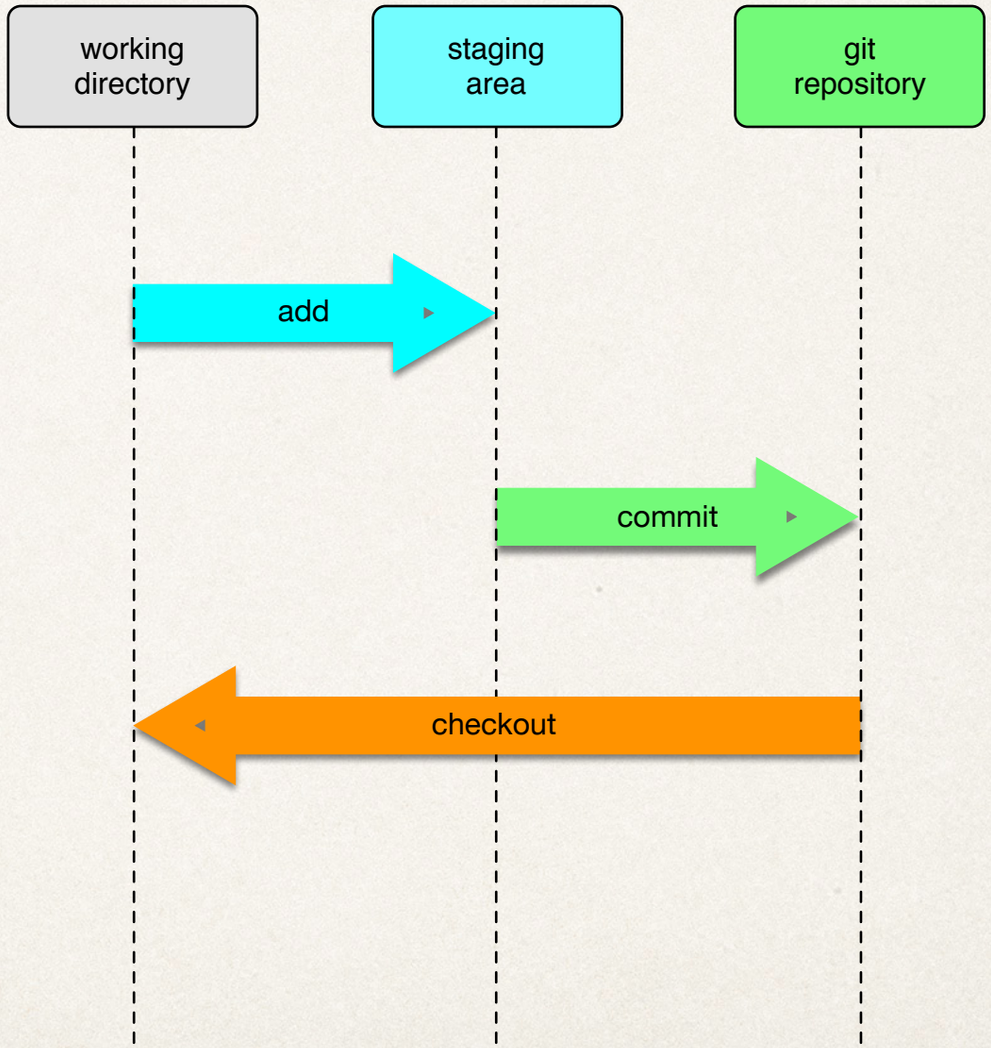
Modify files locally

Stage files for addition, removal, &c.

Perform a *commit* which makes a snapshot out of the staging area

Files can be

- modified (yet to be staged)
- staged (yet to be committed)
- committed



http://git-scm.com/book/en/Getting-Started-Git-Basics

Removing files

We add files for tracking using `git add`, remove them using `git rm`

An IDE with integrated revision control (like RStudio) does this for us

Very important to do it this way rather than just remove the file

Source control is all about keeping track of changes

If you just remove a file, *git* doesn't know you've removed it

The nice thing about having a history though is nothing is gone forever

Tracking changes means telling *git* about the changes

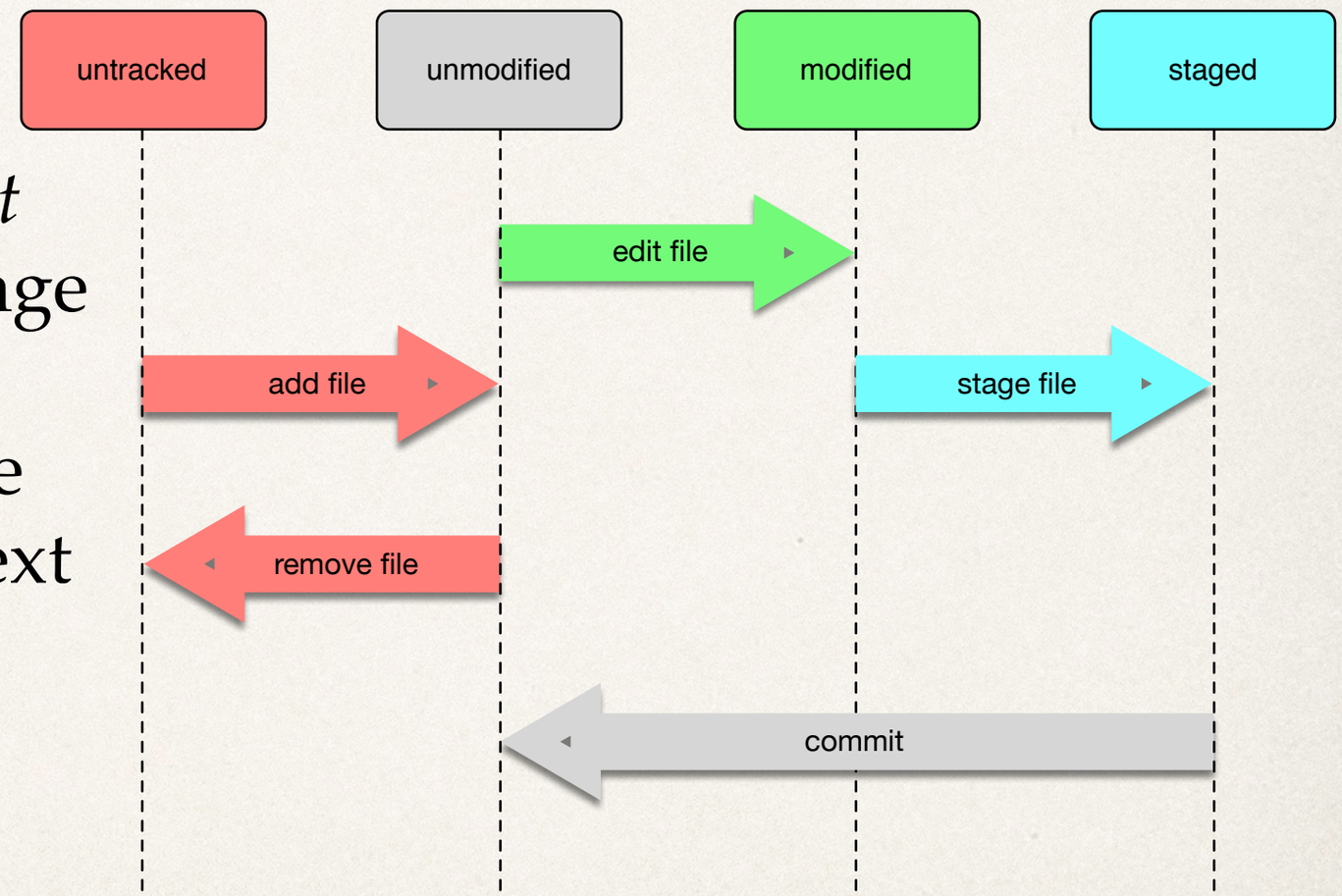
http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository#Removing-Files

File status

`git status` shows the status of files

As we *add*, *edit*, *stage* and *commit* files the status of a file will change

During a *commit*, staged files are put into the repository as the next snapshot

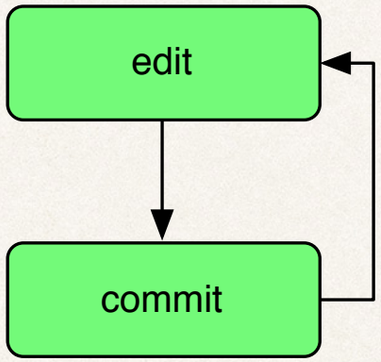


http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository

Routine workflow

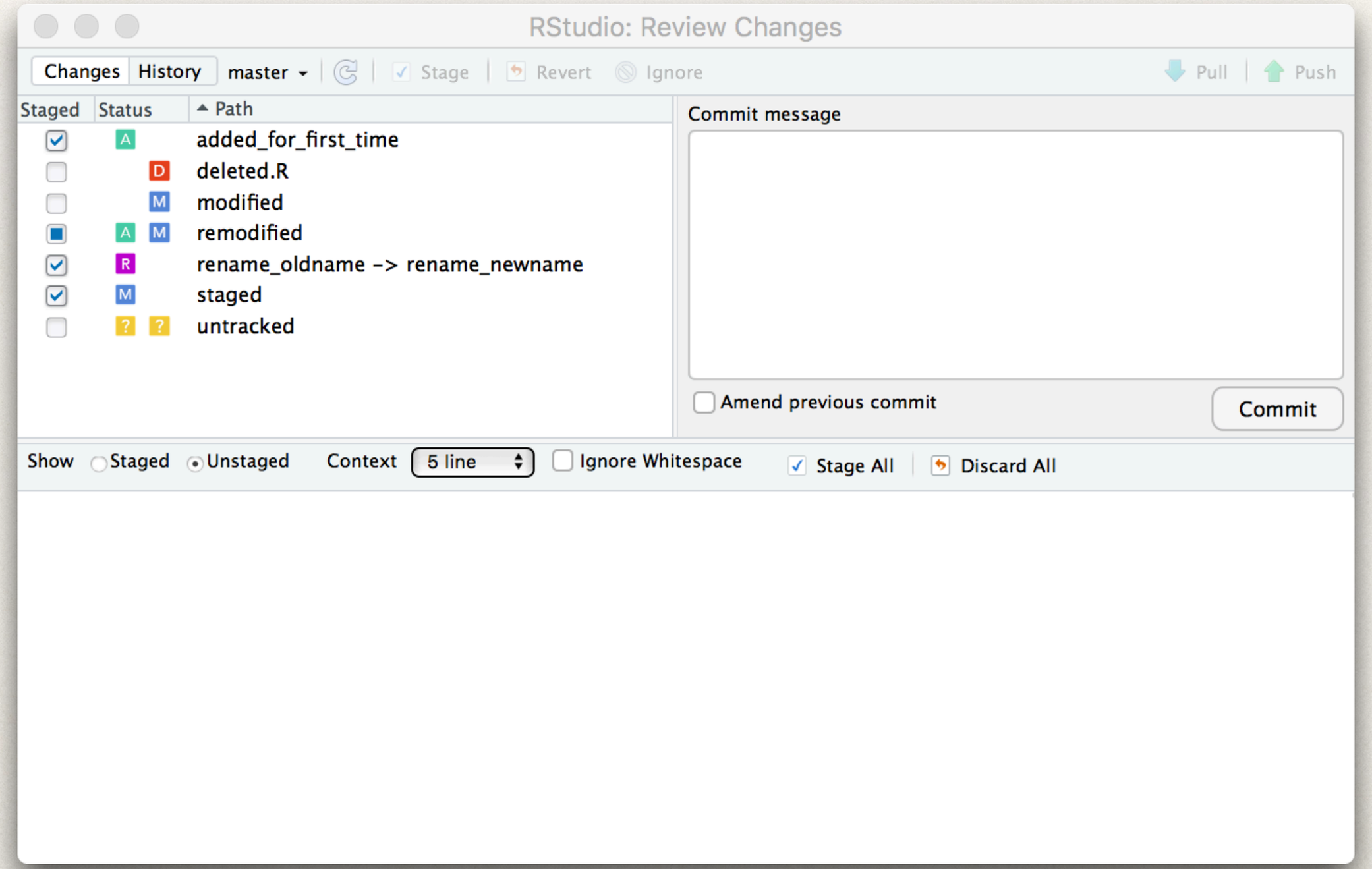
The routine file workflow

Edit → add → commit → edit → add → commit → …



http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository

Committing in RStudio



Commit messages

When we commit we have to supply a message saying what is in the commit, *i.e.* what changes have been made

Bad commit messages

- Fixed bug
- Committed changes
- asdifhasodifuhaspdifu

Good commit message

- Bump sorting column index by one to fix a problem where the addition of the source study column meant tables were no longer sorted by descending time.

The commit log: *git log*

The `git log` command show us the history of commits

```
$ git log
commit b026324c6904b2a9cb4b88d6d61c81d16d7fce9f (HEAD -> master)
Author: James Farrow <james@fn.com.au>
Date:   Fri Apr 13 18:15:22 2017 +1000

    [TYPES] Refactor extract.py.

commit f3cb6641123c49a8b036b408a548baab47267f03
Author: James Farrow <james@fn.com.au>
Date:   Fri Apr 13 17:59:30 2017 +1000

    [TYPES] Add extract type facility to extract.py

commit 1dcca23355272056f04fe8bf20edfce09b6252c8
Author: James Farrow <james@fn.com.au>
Date:   Fri Apr 13 16:57:46 2017 +1000

    Better handling of python executable and default_address.py.
```

Each commit has a name

Consider it a long id

```
$ git log -n 1 b026324c6904b2a9cb4b88d6d61c81d16d7fce9f
commit b026324c6904b2a9cb4b88d6d61c81d16d7fce9f
Author: James Farrow <james@fn.com.au>
Date:   Fri Apr 13 18:15:22 2017 +1000

    [TYPES] Refactor extract.py.
```

Only a unique prefix is needed so usually you'll only need shorter references to commits like `b026324`

```
$ git log -n 1 b026324
commit b026324c6904b2a9cb4b88d6d61c81d16d7fce9f
Author: James Farrow <james@fn.com.au>
Date:   Fri Apr 13 18:15:22 2017 +1000

    [TYPES] Refactor extract.py.
```

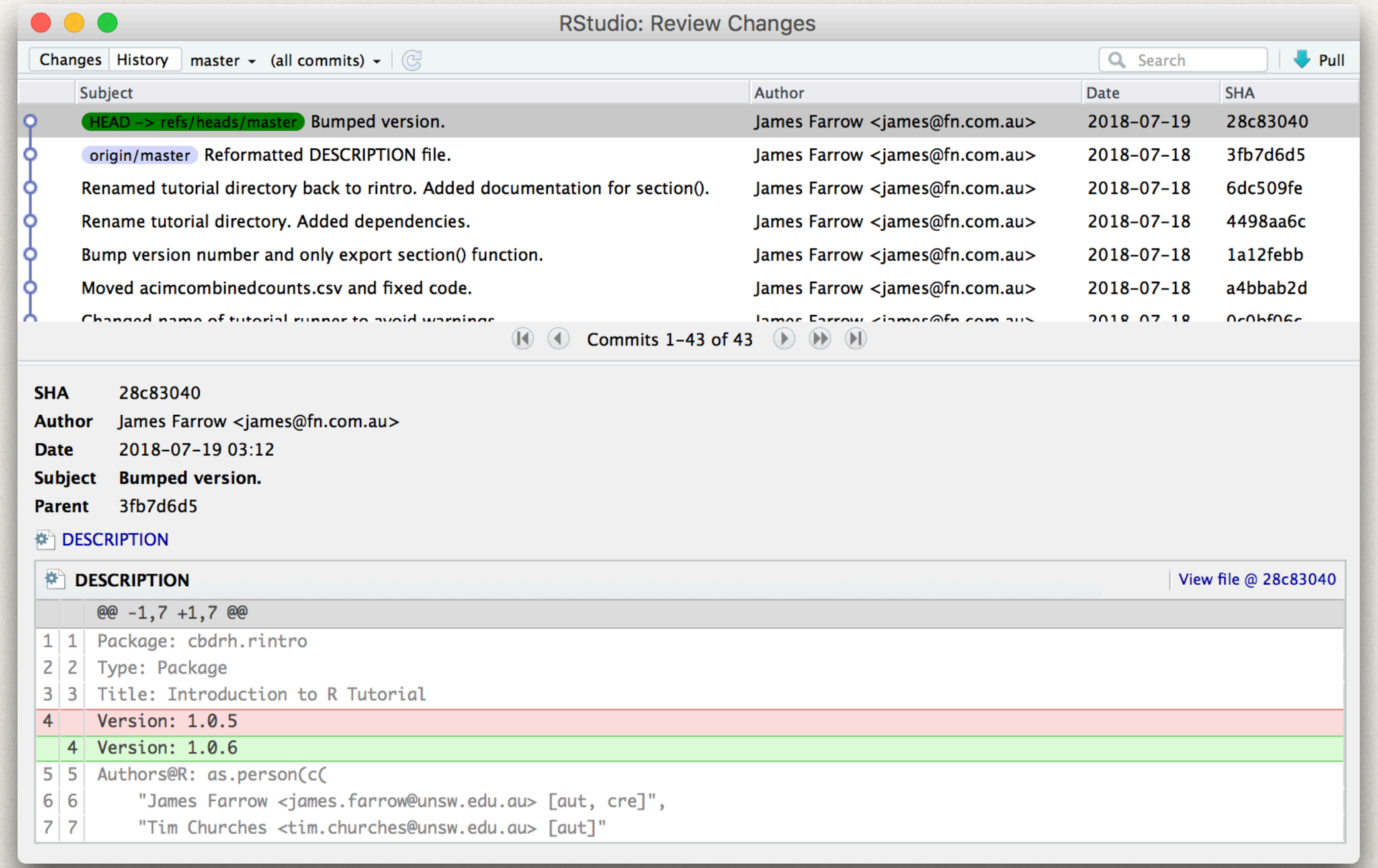
Commit hashes are not integers

If you've used version control systems before you may be used to a version number of a steadily increasing integer

Because *git* is distributed and disconnected and many people can work on a project at the same time it is not possible to assign such an identifier

They're just used to identify a particular commit

Commit history in RStudio



Talking about revisions

We can of course refer to revisions by those specific long ids

Git gives us other, more convenient, ways to talk about revisions

HEAD is the current revision of the branch that you're working on

The name of a branch, *e.g.* master can be used to refer to the latest revision on that branch

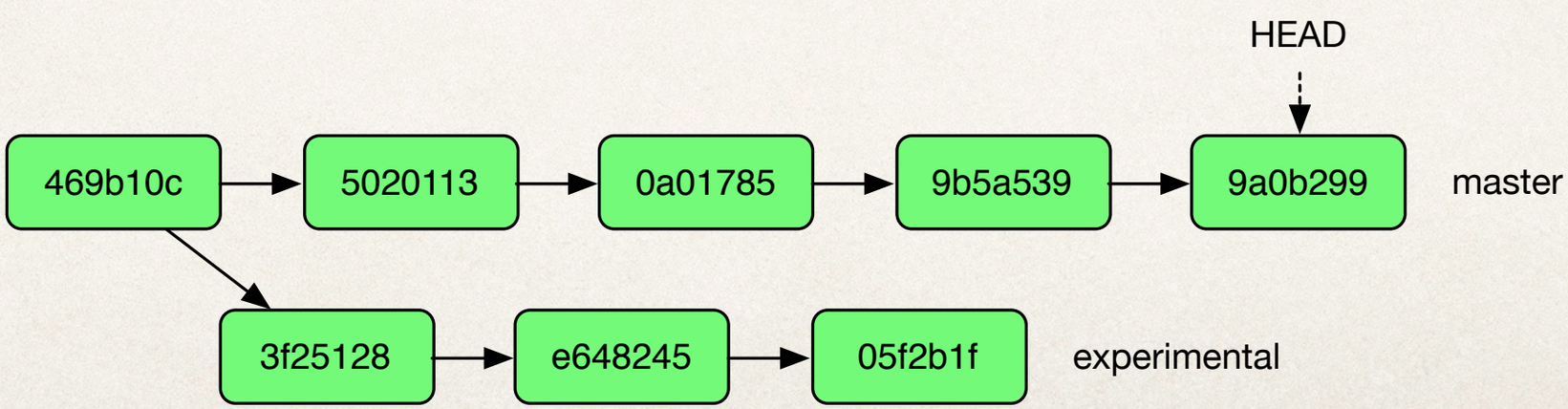
HEAD is not always the same as the end of the master branch, *i.e.* the most recent main revision but in our case it usually will be for now

All of these references are *local* to a particular repository

Branches, revisions and HEAD

In the scenario below

- *master* would refer to commit `9a0b299`
- *experimental* would refer to commit `05f2b1f`
- this repository seems to currently be editing the master branch so *HEAD* would be the same as *master*: `9a0b299`



Good news!

You don’t really need to remember these details right now

Just rest assured that *git* is more than capable of managing quite complex changes and tracking everything that has gone on

Because *git* has a picture of everything that has happened and how a document went from version A to version B it gives us great control over our documents

Some useful notations

Adding `^` to the end of a reference means ‘one commit earlier’

So `HEAD^` means ‘the revision previous to this one’

While `master^` means ‘the revision just before the latest on the *master* branch’

Adding `@{...}` after a reference allows time-based expressions

- `master@{yesterday} HEAD@{one month ago}`

We can use these to help compare files and see what has changed

More complex workflows

Now we have a history of each file in a project and a way to talk about them we can do more

- compare different versions of files
- return to previous working versions
- safely try out an approach to see if it works and rollback if not
- have multiple people work on a project concurrently without overwriting each other's files

Finding differences: *git diff*

The *git diff* command shows us what has changed in a file

There are more ways to use *git diff* but here are some important ones

```
git diff [file …]

git diff <commit> [--] [file …]

git diff <commit> <commit> [--] [file …]
```

So for example

```
git diff HEAD@{yesterday}
```

git diff in RStudio

RStudio: Review Changes

Changes | History | master | (all commits) | Search | Pull

Merge branch 'master' of github.com:CBDRH/Intro-to-R	James Farrow <james@fn.com.au>	2018-07-16	64d09a19
Section 6 and moved 'Special values' from section 6 to section 1.	James Farrow <james@fn.com.au>	2018-07-16	69ec4d80
Fixes to section 1	Tim Churches <tim.churches@gmail.con	2018-07-16	9486a00f
Added section 5.	James Farrow <james@fn.com.au>	2018-07-13	21e089db
Incorporated Sabita's feedback on sections 1-3.	James Farrow <james@fn.com.au>	2018-07-13	d290a8e5
Added section 4.	James Farrow <james@fn.com.au>	2018-07-13	6db6910e

Commits 1-43 of 43

SHA d290a8e5
Author James Farrow <james@fn.com.au>
Date 2018-07-13 05:02
Subject **Incorporated Sabita's feedback on sections 1-3.**
Parent 6db6910e

Intro to R section 1.Rmd
Intro to R section 2.Rmd
Intro to R section 3.Rmd

Intro to R section 1.Rmd

```
@@ -27,7 +27,7 @@ This section deals with
27 27  * Variable types
28 28  * Basic arithmetic operators: `+`, `-`, `*`, `/`, `^`
29 29  * Basic comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
30     - * Basic logical operations: `&, `|`, `^`, `!`
   30  + * Basic logical operations: `&`, `|`, `^`, `!`
31 31
32 32  ## What is R?
33 33
@@ -38,7 +38,7 @@ R is a high-level, open-source programming language.
```

Reverting: *git checkout*

The *git checkout* command lets us pull specific files out of the repository

```
git checkout <commit> file …
```

- this pulls a specific version of a file out of the repository and *replaces* the current version of the file

- **any uncommitted work will be lost: if in doubt commit first**

If, for example, we found that we had completely damaged a file and wanted to go back to the way it was yesterday and throw away all our changes we could run

```
git checkout "HEAD@{yesterday}" helpers.R
```

http://git-scm.com/book/en/Git-Basics-Undoing-Things

Reverting: *git checkout*

The *git checkout* command lets us pull specific files out of the repository

```
git checkout <commit> file …
```

- this pulls a specific version of a file out of the repository and *replaces* the current version of the file

- **any uncommitted work will be lost: if in doubt commit first**

If, for example, we found that we had completely damaged a file and wanted to go back to the way it was yesterday and throw away all our changes we could run

```
git checkout "HEAD@{yesterday}" helpers.R
```

http://git-scm.com/book/en/Git-Basics-Undoing-Things

Reverting and commit size

Being able to revert changes makes revision control very powerful

Of course it needs to be coupled with sensible commit sizes

Too big and too many changes are lumped together

Too small and it's just inconvenient and changes are spread over multiple commits

http://git-scm.com/book/en/Git-Basics-Undoing-Things

Branches

Repositories can contain ‘experimental’ versions of a project

Sometimes we want to ‘try something out’

Rather than disrupt everyone we use a *branch*

Work usually occurs on the *master* branch

http://git-scm.com/book/en/Git-Branching

Branching

Branches split off from the default *master* copy

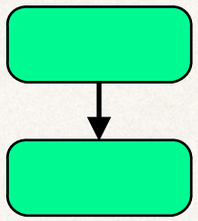
Weird and wonderful experimentation take place

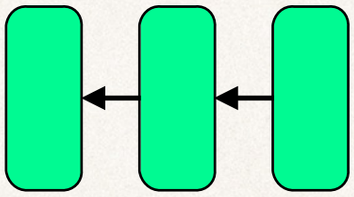
Work can be folded back into *master*, or…

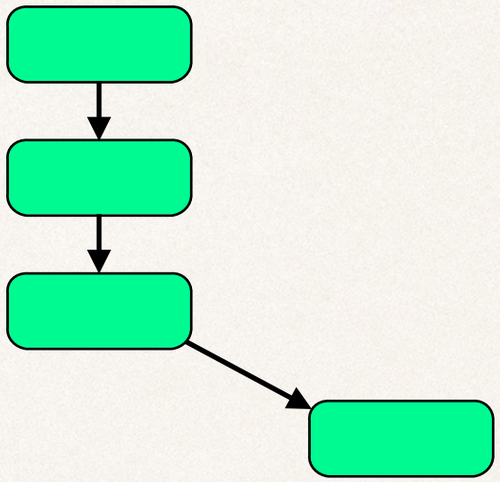
…work can be ignored and abandoned (but not forgotten)

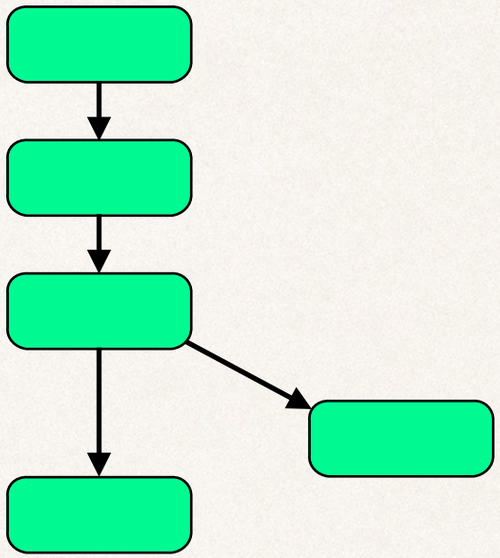
The *master* is the branch everyone uses initially

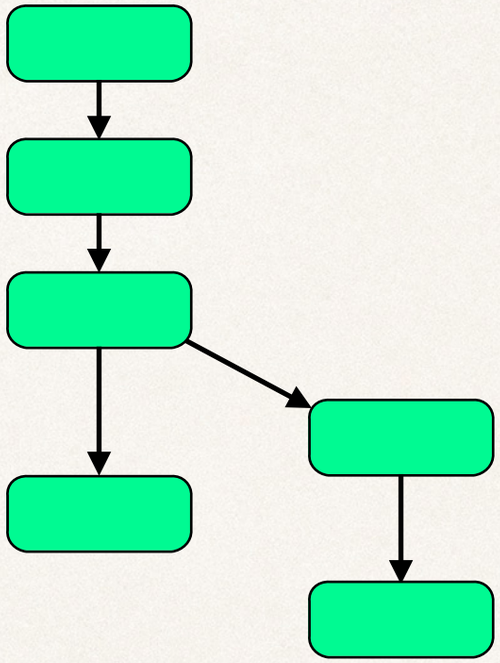
http://git-scm.com/book/en/Git-Branching

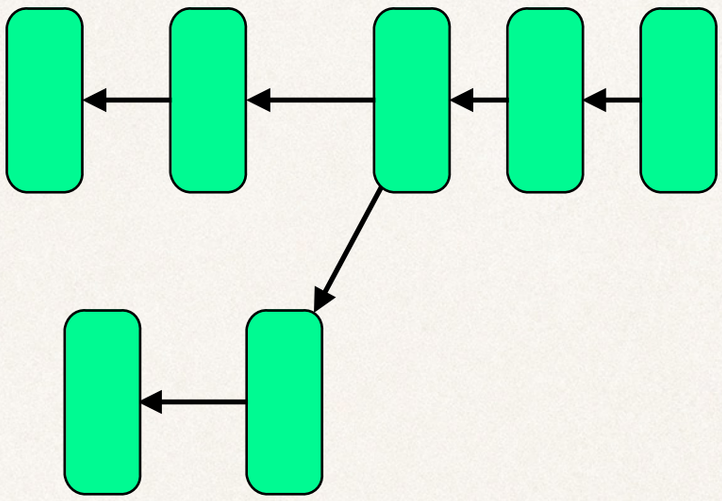


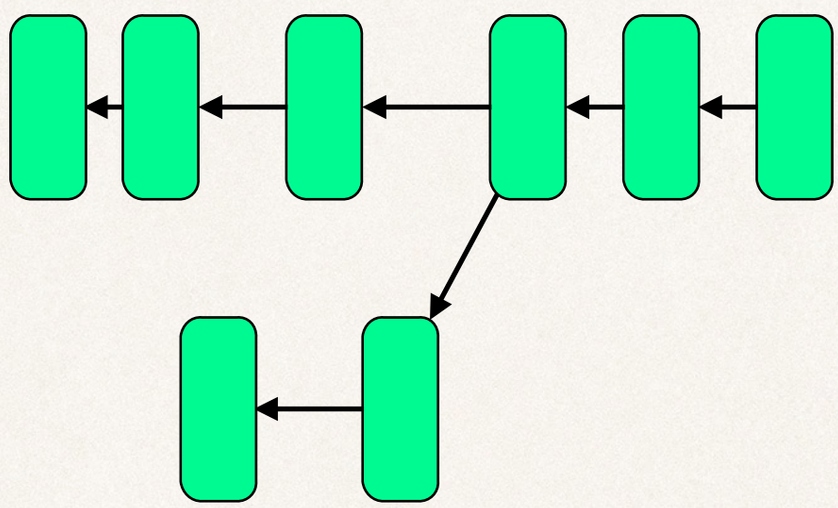


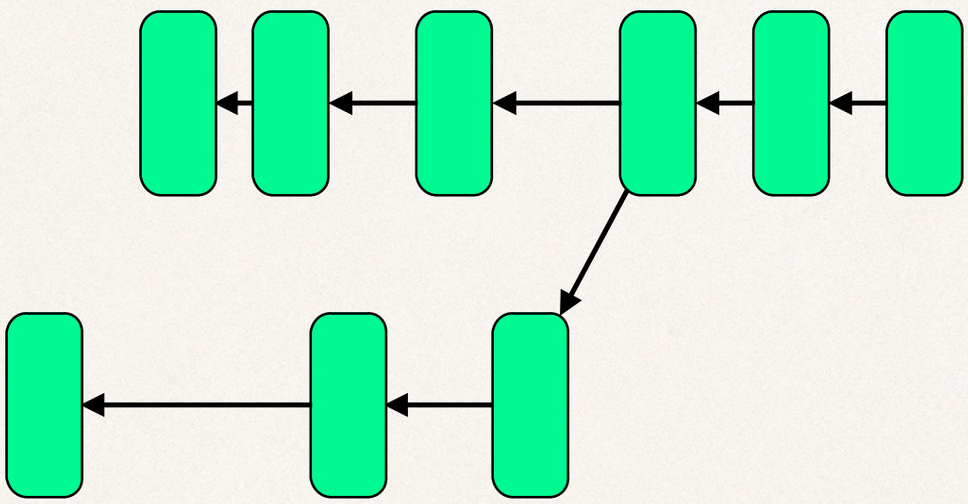


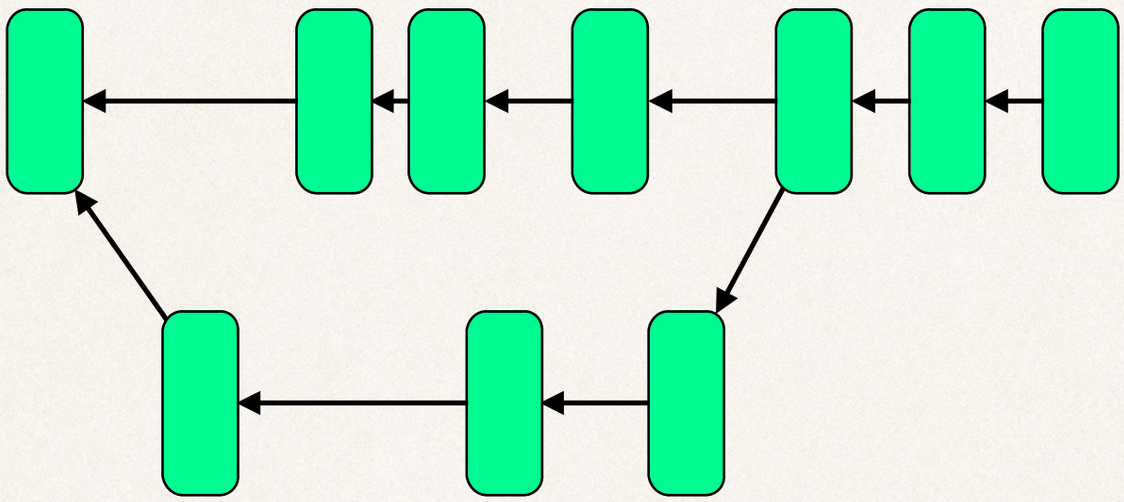


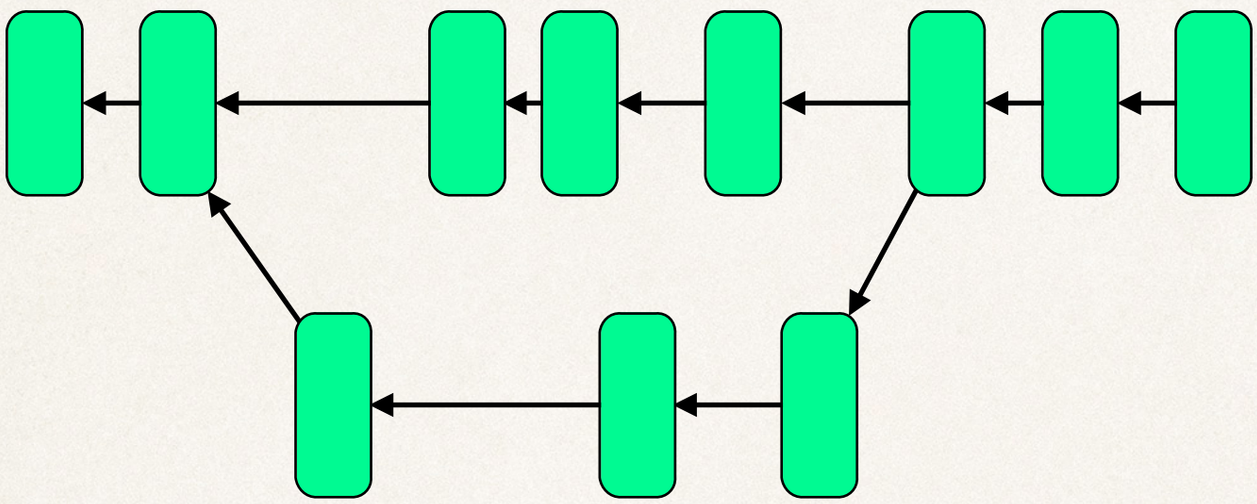


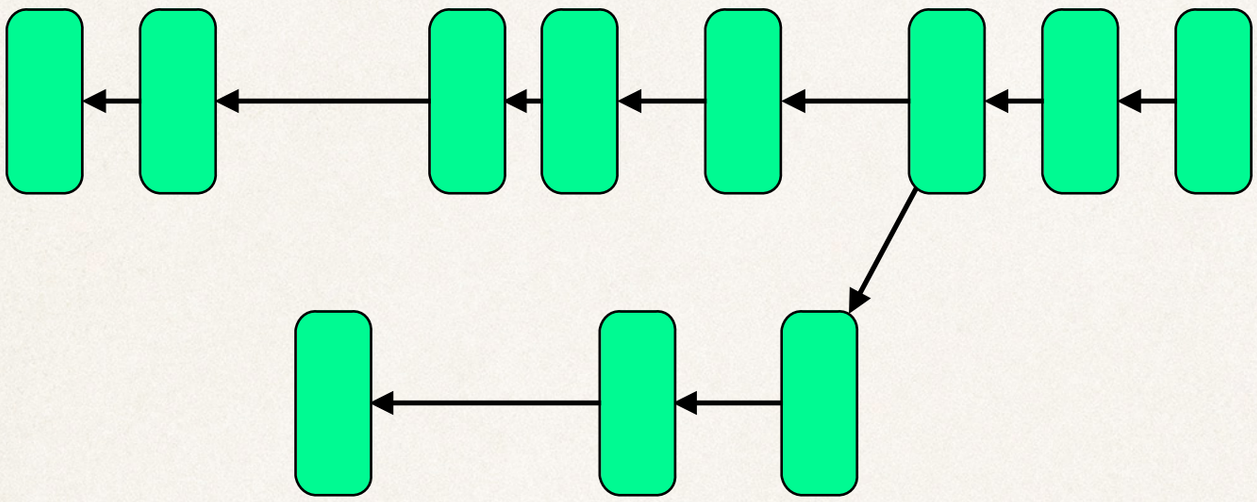












Branches

Branches have the same structure as the trunk from which they are spawned

In fact, at the point of branching they are identical

This is not taking a copy of a project and calling it *my_project_v2*

This is having two versions which you can swap back and forth between at will

Keeping the same structure means it's easier to compare the trunk and the branch and see exactly what has changed

http://git-scm.com/book/en/Git-Branching

Branching

We're not going to look at branching today but be aware it exists

It's a powerful mechanism to make major changes to a project while isolating those changes so that they don't affect everyone else

It's also a good mechanism to make major changes and keep them separate from your own work on the mainline project development

Instead of working for days and not saving 'because you don't want to overwrite the original' or saving a copy as *my_prog.experimental.R* 'to keep it separate' we can continue to work on *my_prog.R* and continue to exploit the benefits of regular commits

http://git-scm.com/book/en/Git-Branching

Remote repositories

So far we have just talked about working within a local repository

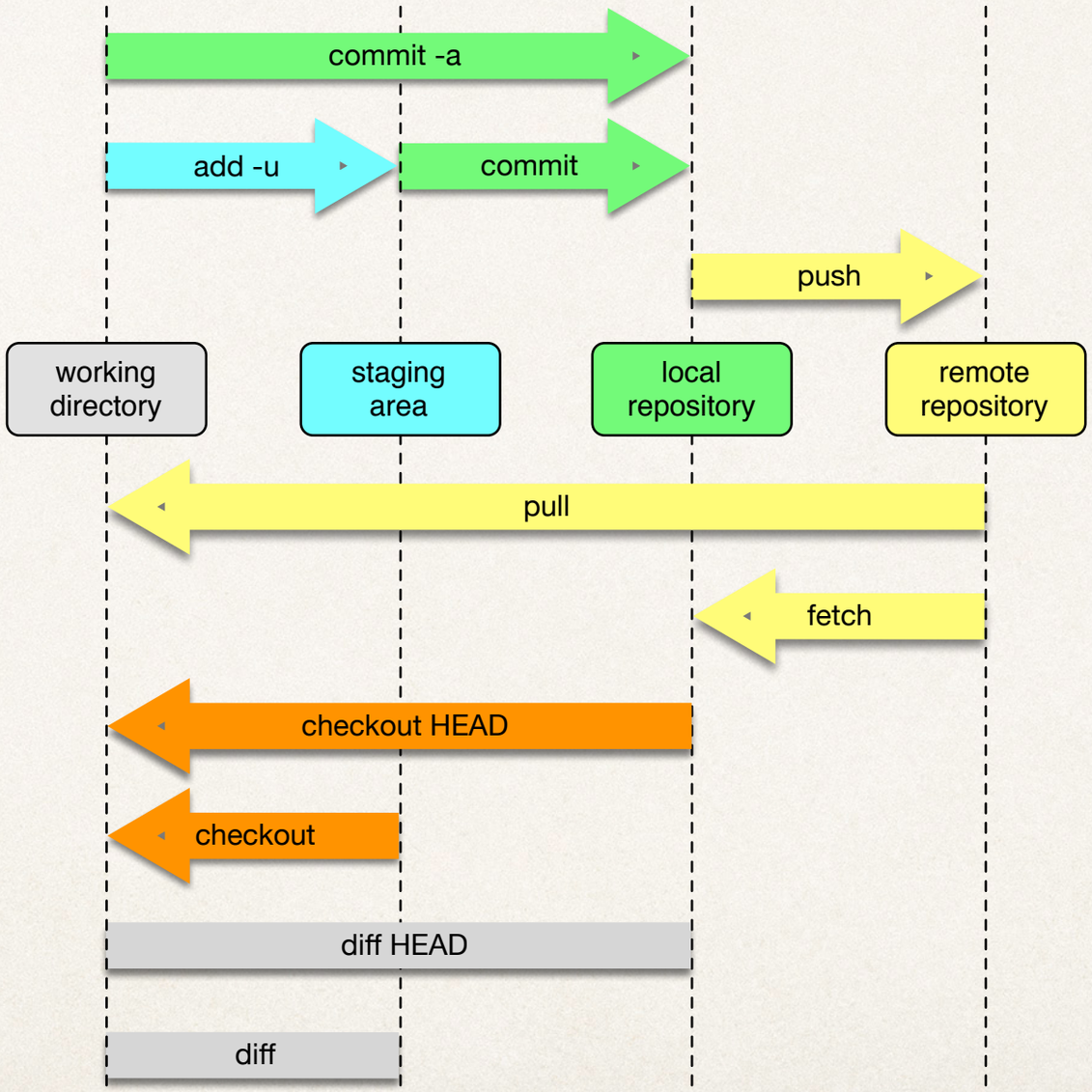
Repositories can also ‘push’ and ‘pull’ changes between repositories

We call repositories that aren’t the local repository *remote* repositories

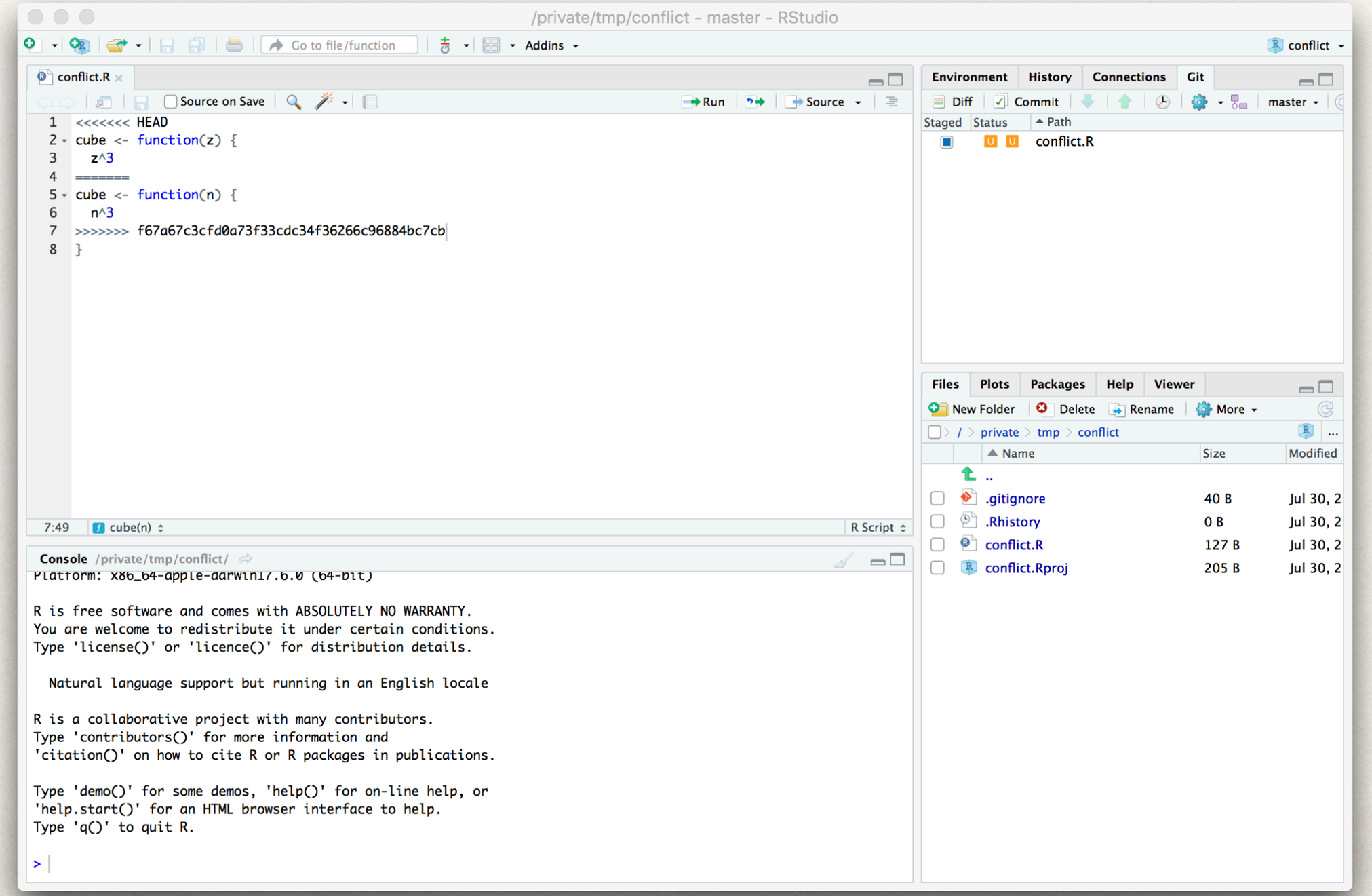
Remote repositories are important for collaboration

http://git-scm.com/book/en/Git-Basics-Working-with-Remotes

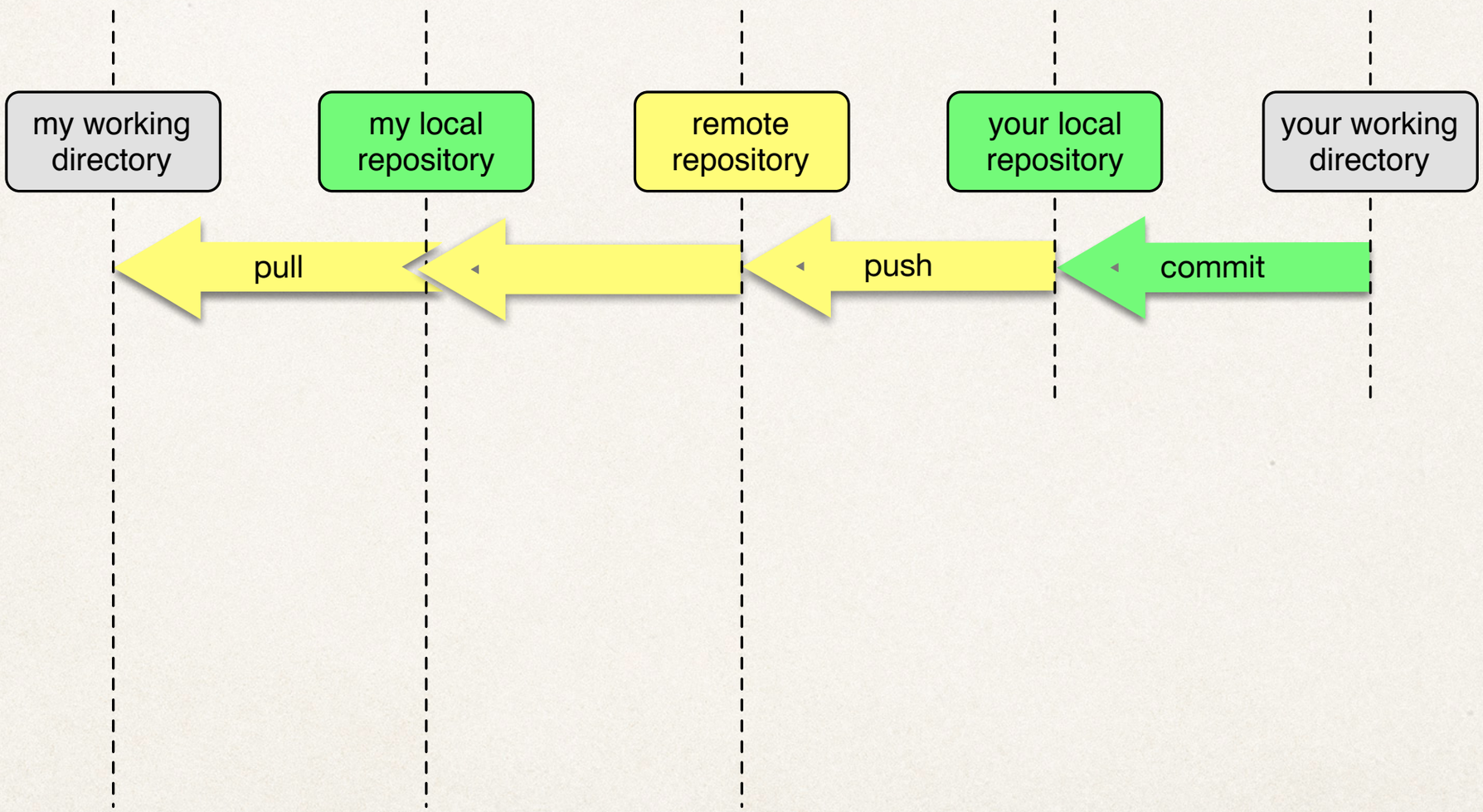
File workflow revisited



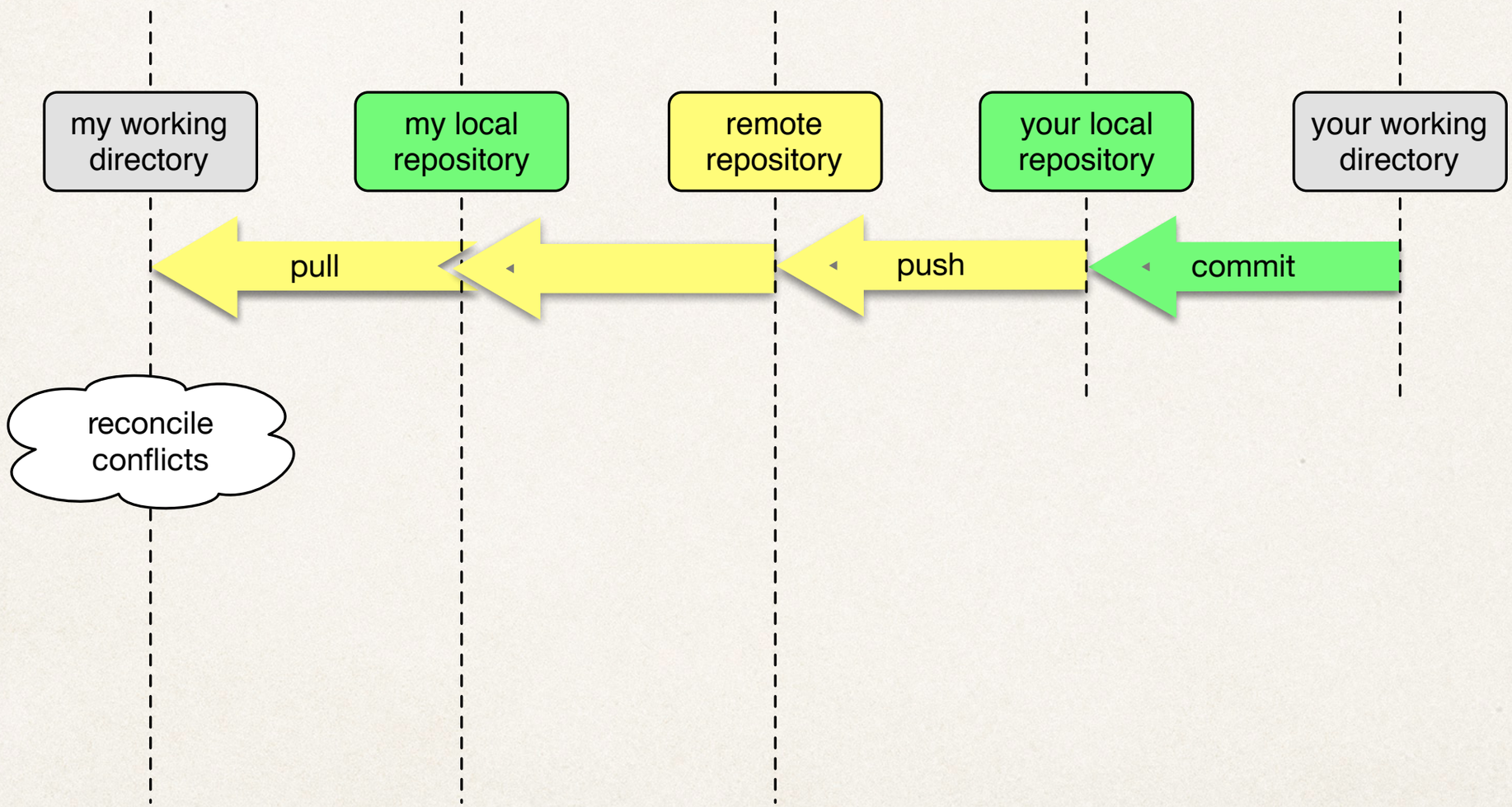
Conflicts in RStudio



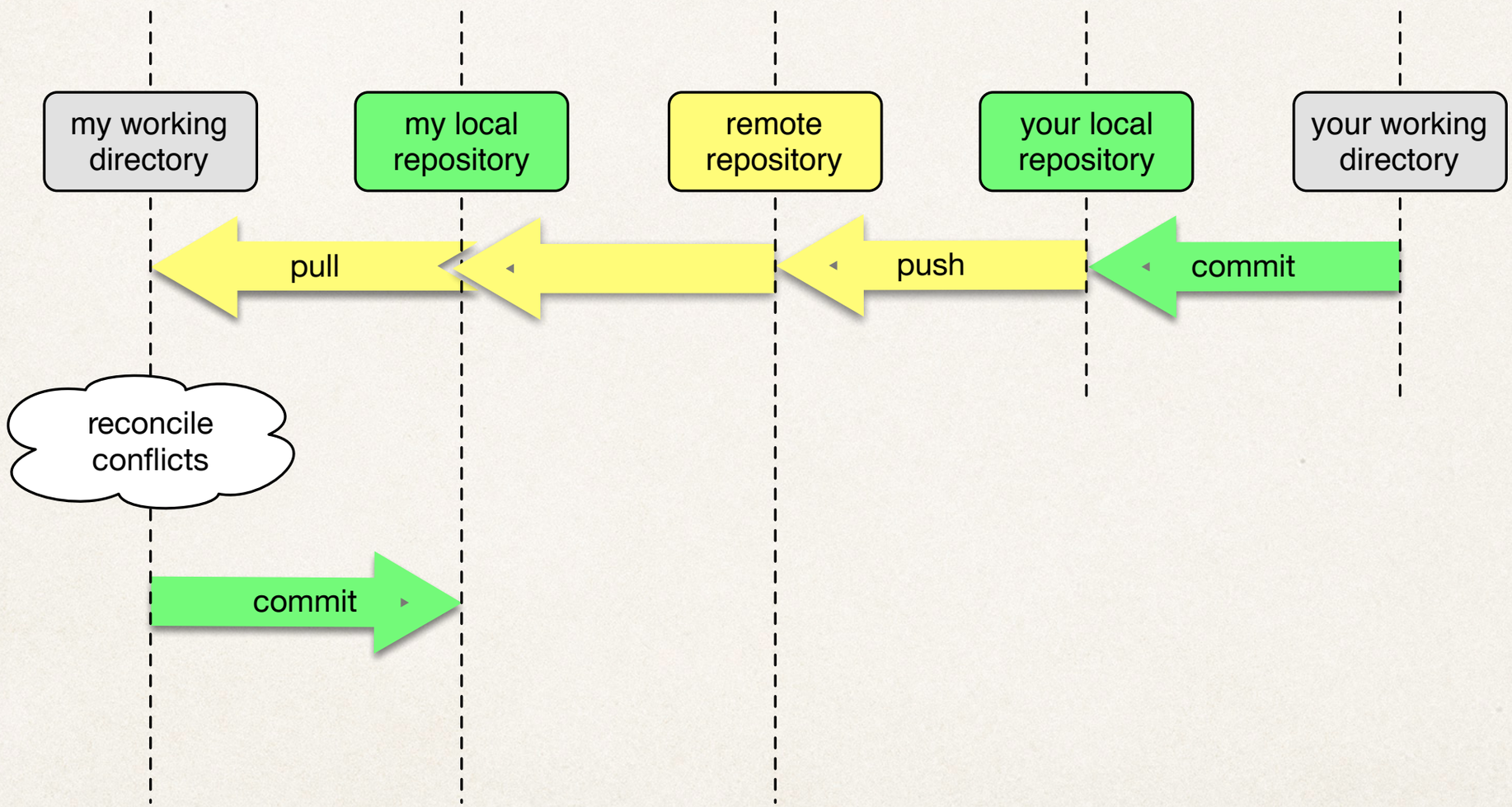
Collaborative workflow



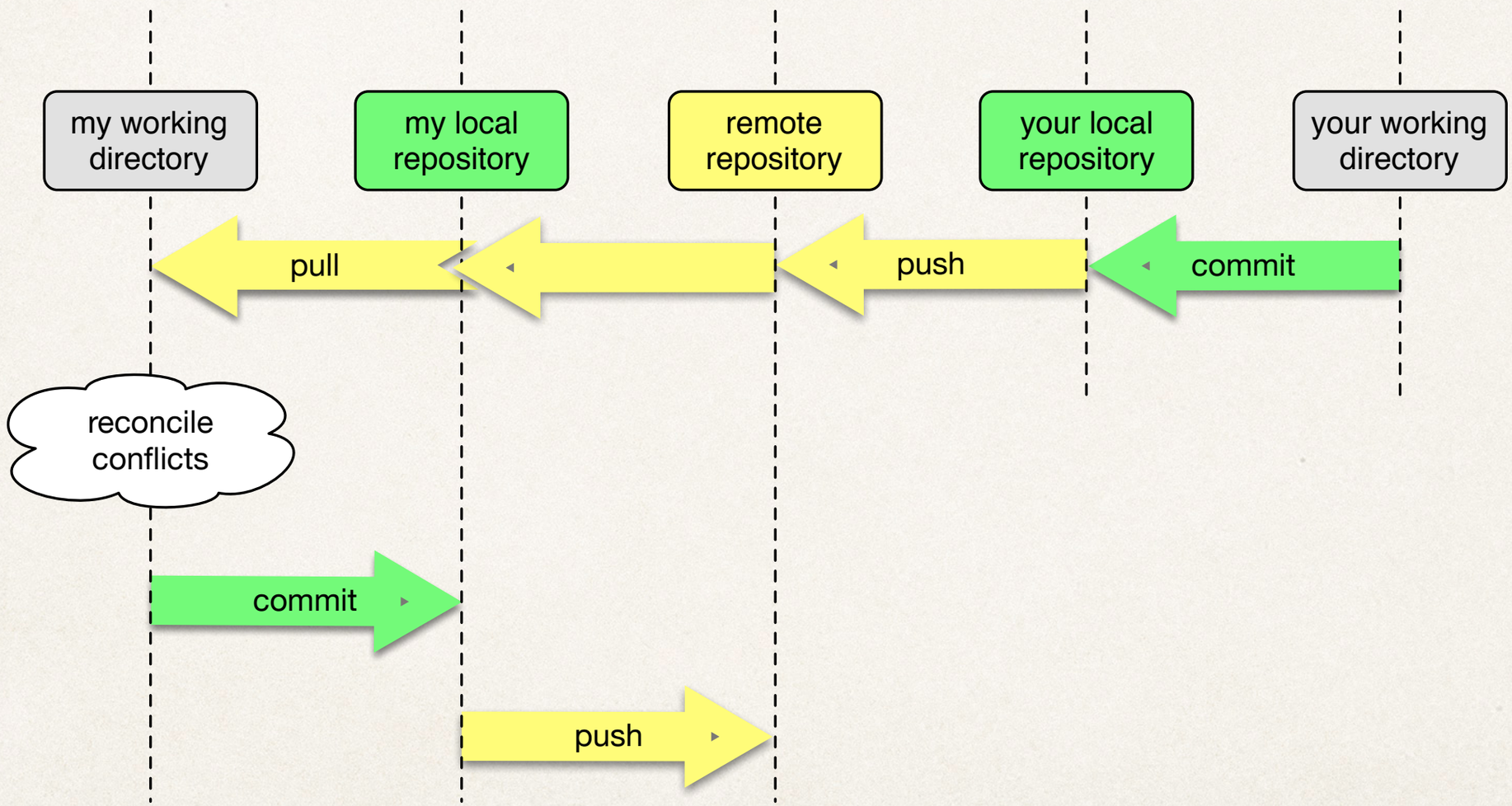
Collaborative workflow



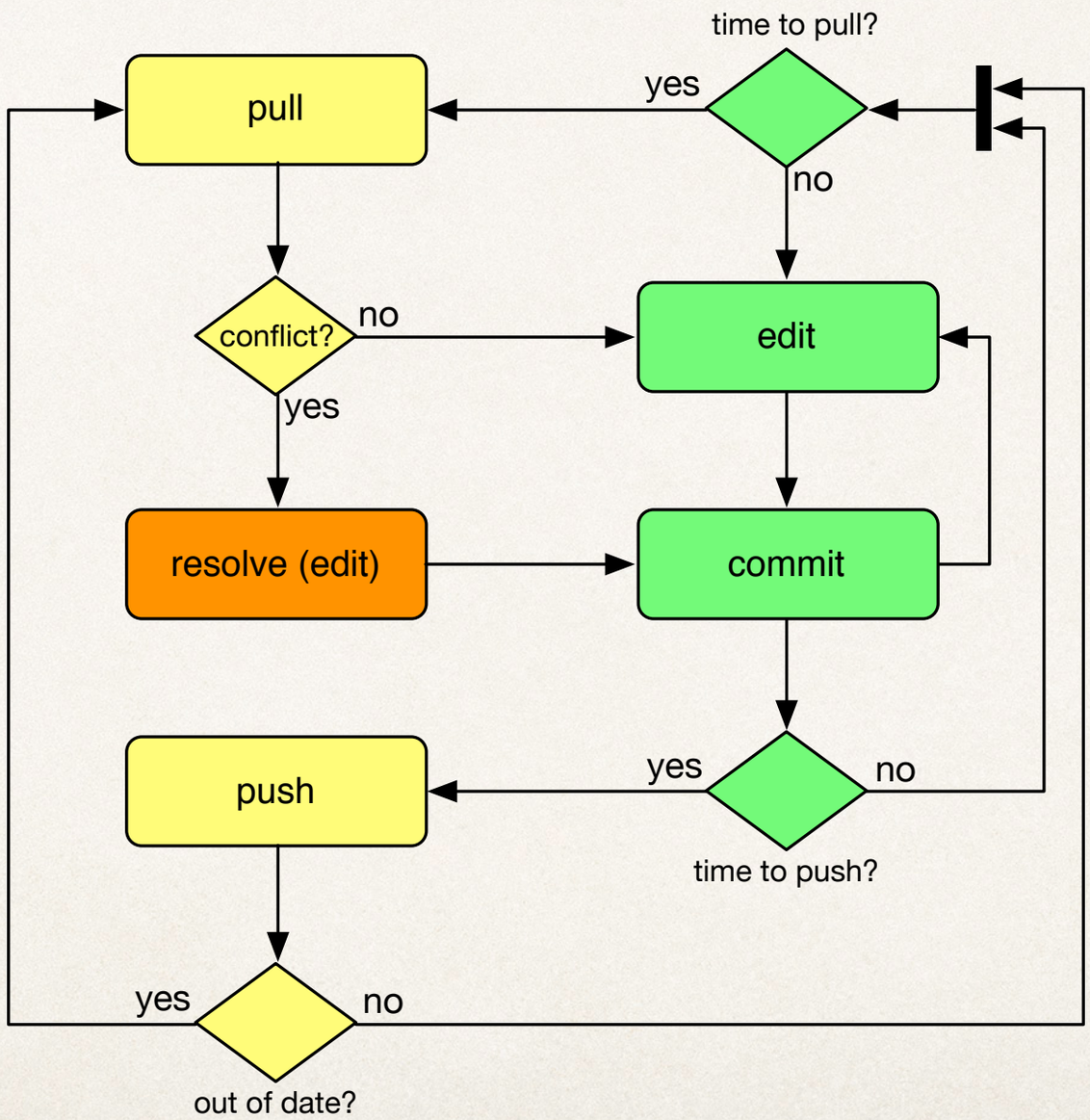
Collaborative workflow



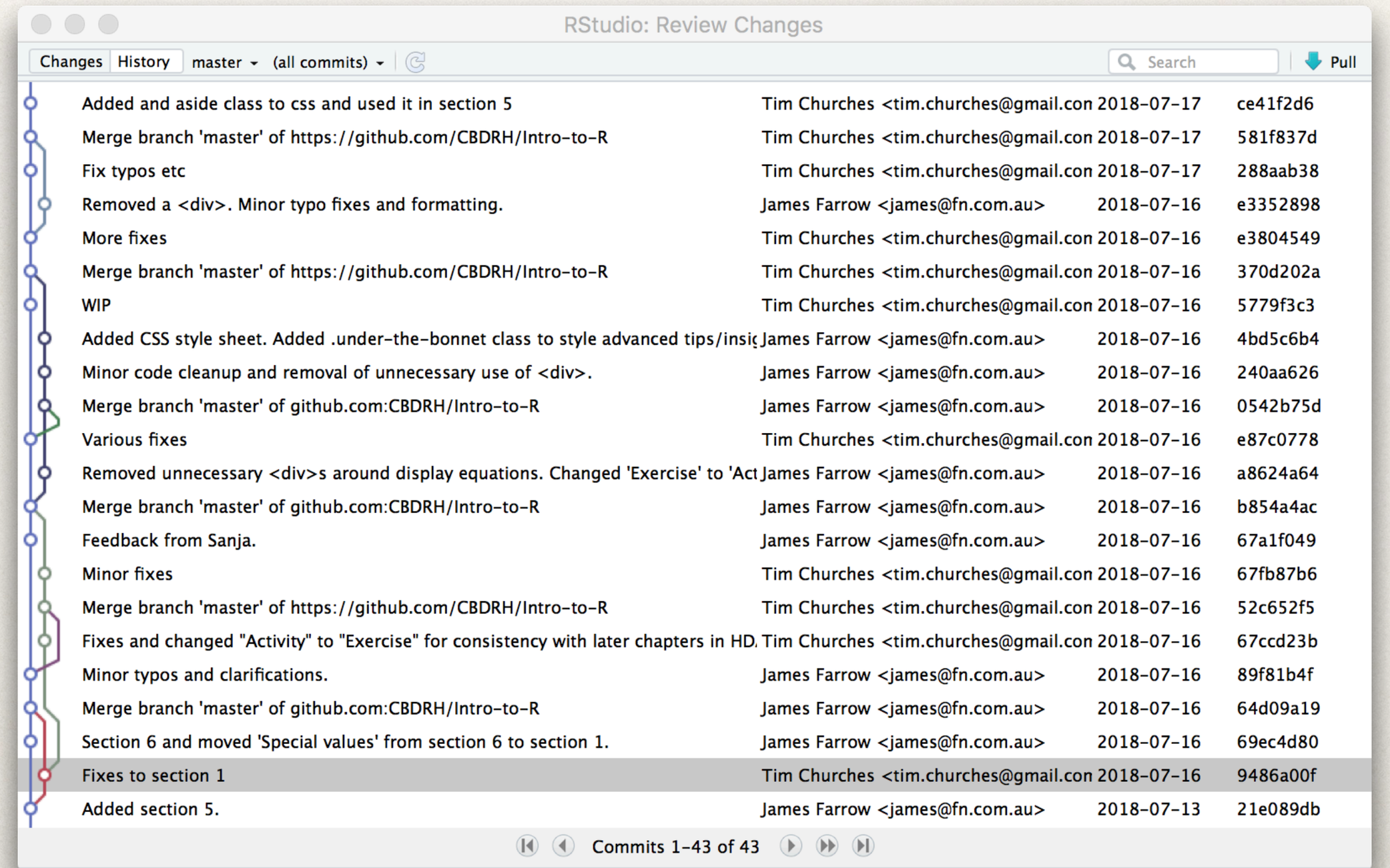
Collaborative workflow



Collaborative workflow



Commit history in RStudio



Setting up *git*

Install *git*

Windows

- download from *https://git-scm.com/download/win*

MacOS

- install Xcode tools: `xcode-select --install`
- install via Homebrew: *https://brew.sh*

Linux users should use their package manager

Installing *git* on Windows

Choose an editor that you're happy using: it only needs to edit text files

For Windows 'Use git from the the Window Command Prompt' is the best option unless you need something different

Use OpenSSL unless you have corporate requirements

For line endings, choose the first (`core.autocrlf = true`) option

Use *minTTY* for terminal emulation

Basically, leave things at their *default values*

Configuring *git*

You need to let *git* know who you are

This can be done *globally* (for all local repositories) and overridden in each local repository as necessary

This enables *git* to record *authorship* information: who did what

You want your contributions to be recognised, yes?

- especially when it comes to assignments 😁

Setting up GitHub

GitHub is not *git*

GitHub is a site which manages a git repository

We will be using GitHub for assignments and in workshops

Go to *https://github.com* and create an account if you don't have one

You don't need to use your uni/work email address

GitHub can be configured to recognise many addresses as 'you'

Configuring *git*

Open a terminal window

- The *Terminal* app under macOS can be found using *Spotlight* or opened from the *Utilities* folder opened from the *Go* menu in the *Finder*
- Open *Git Bash* under Windows using the *Start* menu

Type the following (replacing the details between the "…"

```
git config --global user.name "James Farrow"

git config --global user.email "james.farrow@unsw.edu.au"
```

Configuration

You may also be using *git* and *GitHub Classroom* in other courses

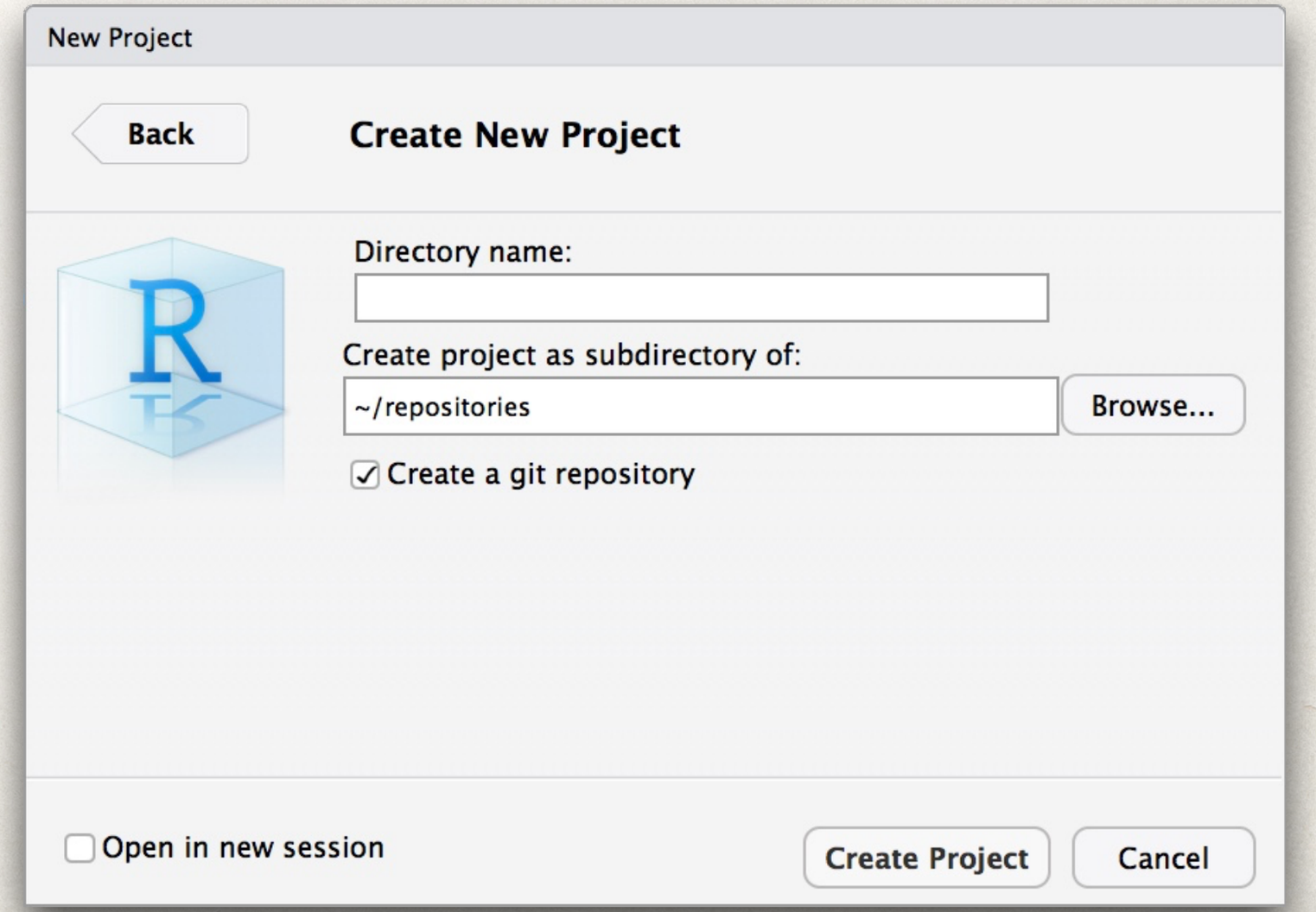
Set up your GitHub account and configure *git* with the same identity

Setting up RStudio to use *git*

When *git* is installed RStudio can be configured to use it for a project

Always do this

There is a checkbox on the 'New project' dialog to enable *git*



Adding *git* to an existing project

Run `git init` in the top level directory

This will create a new *.git* directory with an empty local repository

- Never touch anything in this folder

Use `git add` and `git commit` to add in the existing file

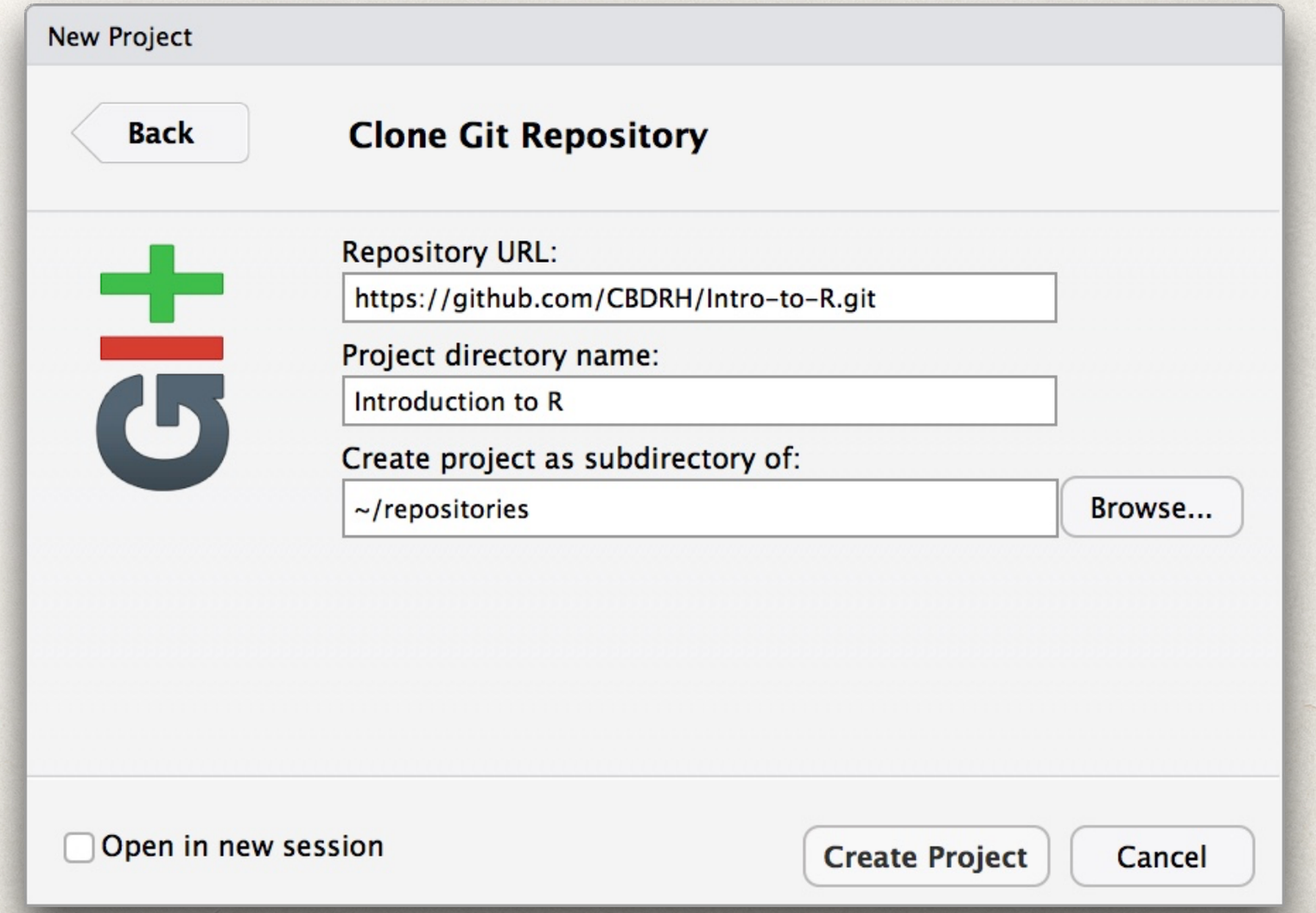
From then on it's business as usual

Cloning a project from *GitHub*

A project repository may already exist on GitHub

To clone this start a new project in RStudio but choose *Version Control* rather than *New Directory* in the initial dialogue box

Then choose *Git* to clone an existing project



Using GitHub Classroom

We will be doing weekly assessments (and the group assignment) using *GitHub Classroom*

Each week you will receive a link for that week's assessment

You *must* have a GitHub account to participate in the assessments

Following the link will create a new repository for you for each assessment with the start files for that assessment already set up

You need to *clone* each repository, *commit* your work and *push* the results

Whatever is in the repository at each deadline is what will be marked

Assessment repositories

Each assessment will have a name like *Chapter 1 Assessment*

When you join each assessment exercise *GitHub Classroom* will create a new repository for you with your *GitHub* username appended

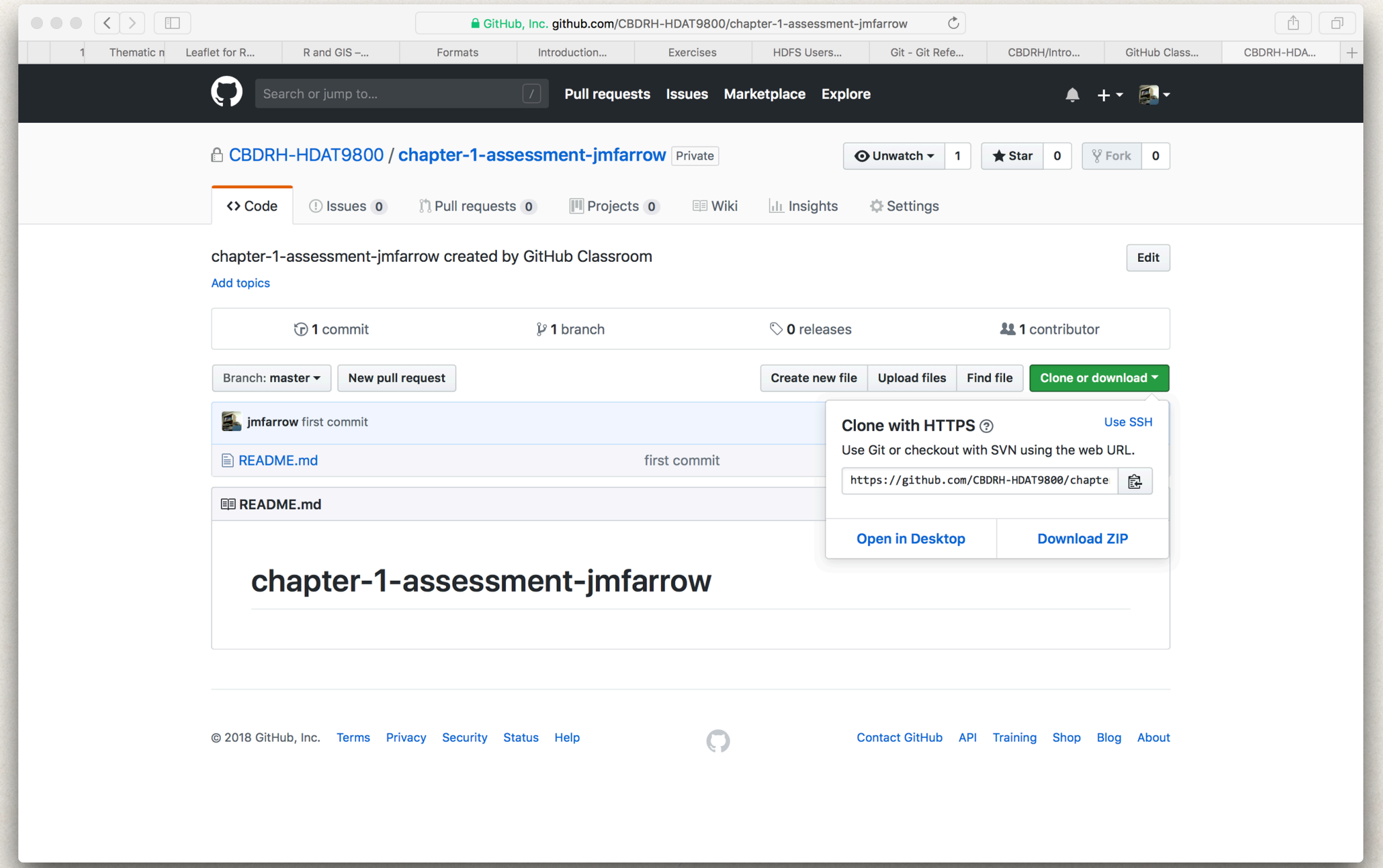
CBDRH-HDAT9800 is the organisation

Your repository for Chapter 1 Assessment will be something like

- *https://github.com/CBDRH-HDAT9800/chapter-1-assessment-jmfarrow*

GitHub Classroom will send you links to all of this, you don't need to remember it

Cloning the assessment repository



Cloning the assessment repository

It is suggested you make a folder for this course

You can keep all the materials for this course organised within that folder however you like however, *each assessment will need its own folder*

Do not create git managed projects inside other git projects

- That means don't create one week's folder inside another's

Use the HTTPS URL from GitHub to *clone* the project using RStudio

From then on use the RStudio IDE to *commit* and *push* to *GitHub*

Chapter 1 Assessment

Get the classroom link from the Chapter 1 Assessment page.

Use this link to create a Chapter 1 Assessment repository

Clone the repository

Edit the *questions.txt* file in the repository to add your answers

Commit your changes and *push*

Chapter 1 assessment

(To be submitted using GitHub Classroom. This slide is for reference only.)

1. What is your previous R and programming experience? [1 mark for answering; we'd like your input to know where you're at]

2. A colleague comes to you and claims 'I always use a bar chart because it's always the best presentation format.' Is your colleague speaking sense? If so why is the statement right? If not how do you counter the argument? [2 marks]

3. When communicating it is very important to 'know your …'? Finish this sentence. [1 mark]

4. Why is an automated approach to data manipulation better than a manual approach? Why should we write code to manipulate data rather than do it by hand? [3 marks]

5. What does CRAN stand for and what is it? [2 marks]

6. State one advantage to the literate programming approach? [1 mark]