Effective Data Science

Zak Varty

2023-07-27

Table of contents

About this Course

Model building and evaluation are are necessary but not sufficient skills for the effective practice of data science. In this module you will develop the technical and personal skills that are required to work successfully as a data scientist within an organisation.

During this module you will critically explore how to:

- effectively scope and manage a data science project;
- work openly and reproducibly;
- efficiently acquire, manipulate, and present data;
- interpret and explain your work for a variety of stakeholders;
- ensure that your work can be put into production;
- assess the ethical implications of your work as a data scientist.

This interdisciplinary course will draw from fields including statistics, computing, management science and data ethics. Each topic will be investigated through a selection of lecture videos, conference presentations and academic papers, hands-on lab exercises, along with readings on industry best-practices from recognised professional bodies.

Schedule

These notes are intended for students on the course **MATH70076: Data Science** in the academic year 2023/24.

As the course is scheduled to take place over five weeks, the suggested schedule is:

- 1st week: effective data science workflows;
- 2nd week: acquiring and sharing data;
- 3rd week: exploratory data analysis and visualisation;
- 4th week: preparing for production;
- 5th week: ethics and context of data science.

A pdf version of these notes may be downloaded here. Please be aware that these are very rough and will be updated less frequently than the course webpage.

Learning outcomes

On successful completion of this module students should be able to:

1. Independently scope and manage a data science project;
2. Source data from the internet through web scraping and APIs;
3. Clean, explore and visualise data, justifying and documenting the decisions made;
4. Evaluate the need for (and implement) approaches that are explainable, reproducible and scalable;
5. Appraise the ethical implications of a data science projects, particularly the risks of compromising privacy or fairness and the potential to cause harm.

Allocation of Study Hours

Lectures: 10 Hours (2 hours per week)

Group Teaching: 5 Hours (1 hour per week)

Lab / Practical: 10 hours (2 hours per week)

Independent Study: 100 hours (15 hours per week + 30 hours coursework)

Drop-In Sessions: Each week there will be a 1-hour optional drop-in session to address any questions about the course or material. This is where you can get support from the course lecturer or GTA on the topics covered each week, individually or in small groups.

Office Hours: Additionally, there will be an office hour each week. This is a weekly opportunity for 1-1 discussion with the course lecturer to address any individual questions, concerns or problems that you might have. These meetings can be in person or on Teams and can be academic (relating to course content or progress) or pastoral (relating to student well-being) in nature. To book a 1-1 meeting please use the link on the course blackboard page.

Assessment Structure

The course will be assessed entirely by coursework, reflecting the practical and pragmatic nature of the course material.

Coursework 1 (30%): To be completed during the fourth week of the course.

Coursework 2 (70%): To be released in the last week of the course and submitted following the examination period in Summer term.

Acknowledgements

These notes were created by Dr Zak Varty. They were inspired by a previous lecture series by Dr Purvasha Chakravarti at Imperial College London and draw from many resource that were made available by the R community.

Part I

Effective Workflows

> **Note**
>
> Effective Data Science is still a work-in-progress. This chapter should be readable but is currently undergoing final polishing.
> If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

As a data scientist you will never work alone.

Within a single project a data scientist is likely that you will interact with a range of other people, including but not limited to: one or more project managers, stakeholders and subject matter experts. These experts might come from a single specialism or form a multidisciplinary team, depending on the type of work that you are doing.

To get your project put into use and working at scale you will likely have to collaborate with data engineers. You will also work closely with other data scientists, to review one another's work or to collaborate on larger projects.

Familiarity with the skills, processes and practices that make for collaboration is instrumental to being a successful as a data scientist. The aim for this part of the course is to provide you with a structure on how you organise and perform your work, so that you can be a good collaborator to current colleges and your future self.

This is going to require a bit more effort upfront, but the benefits will compound over time. You will get more done by wasting less time staring quizzically at messy folders of indecipherable code. You will also gain a reputation of someone who is good to work with. This promotes better professional relationships and greater levels of trust, which can in turn lead to working on more exciting and impactful projects.

1 Organising your work

```
source("_common.R")
status("polishing")
```

> **Note**
>
> Effective Data Science is still a work-in-progress. This chapter should be readable but is currently undergoing final polishing.
> If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

Welcome to this course on effective data science. This week we'll be considering effective data science workflows. These workflows are ways of progressing a project that will help you to produce high quality work and help to make you a good collaborator.

In this Chapter, we'll kick things off by looking at how you can structure data science projects and organize your work. Familiarity with these skills, processes and practices for collaborative working are going to be instrumental as you become a successful data scientist.

1.1 What are we trying to do?

First, let's consider why we want to provide our data science projects with some sense of structure and organization.

As a data scientist you'll never work alone. Within a single project you'll interact with a whole range of other people. This might be a project manager, one or more business stakeholders or a variety of subject matter experts. These experts might be trained as sociologists, chemists, or civil servants depending on the exact type of data science work that you're doing.

To then get your project put into use and working at scale you'll have to collaborate with data engineers. You'll also likely work closely with other data scientists. For smaller projects this might be to act as reviewers for one another's work. For larger projects working collaboratively will allow you to tackle larger challenges. These are the sorts of project that wouldn't be feasible alone, because of the inherent limitations on the time and skill of any one individual person.

Even if you work in a small organization, where you're the only data scientist, then adopting a way of working that's focused on collaborating will pay dividends over time. This is because when you inevitably return to the project that you're working on in several weeks or months or years into the future you'll have forgotten almost everything of what you did the first time around. You'll also have forgotten why you made the decisions that you did and what other potential options there were that you didn't take.

This is exactly like working with a current colleague who has shoddy or poor working practices. Nobody wants to be that colleague to somebody else, let alone to their future self. Even when working alone, treating your future self as a current collaborator (and one that you want to get along well with) makes you a kind colleague and a pleasure to work with.

The aim of this week is to provide you with a guiding structure on how you organize and perform your work. None of this is going to be particularly difficult or onerous. However it will require a bit more effort up front and daily discipline. Like with flossing, the daily effort required is not large but the benefits will compound over time.

You'll get more done by wasting less time staring quizzically at a mess of folders and indecipherable code. You'll also get a reputation as someone who's well organized and good to work with. This promotes better professional relationships and greater levels of trust within your team. These can then, in turn, tead to you working on more exciting and more impactful projects in the future.

1.2 An R Focused Approach

The structures and workflows that are recommend here and throughout the rest of this module are focused strongly on a workflow that predominantly uses R, markdown and LaTeX.

Similar techniques, code and software can achieve the same results that I show you here when coding in Python or C, or when writing up projects in Quarto or some other markup language. Similarly, different organizations have their own variations on these best practices that we'll go through together. Often organisations will have extensive guidance on these topics.

The important thing is that once you understand what good habits are and have build them in one programming language or business, then transferring these skills to a new setting is largely a matter of learning some new vocabulary or slightly different syntax.

With that said, let's get going!

1.3 One Project = One Directory

If there's one thing you should take away from this chapter, it's this one Golden Rule:

> Every individual project you work on as a data scientist should be in a single, self-contained directory or folder.

This is worth repeating. Every single project that you work on should be self-contained and live in a single directory. An analogy here might be having a separate ring-binder folder for each of your modules on a degree program.

1 project = 1 directory

This one golden rule is deceptively simple.

The first issue here is that it requires a predetermined scope of what is and what isn't going to be covered by this particular project. This seems straightforward but at the outset of the project you often do not know exactly where your project will go, or how it will link to other pieces of work within your organization.

The second issue is the second law of Thermodynamics, which applies equally well to project management as it does to the heatdeath of the universe. It takes continual external effort to prevent the contents of this one folder from becoming chaotic and disordered over time.

That being said, having a single directory has several benefits which more than justify this additional work.

1.4 Properties of a Well-Orgainsed Project

What are the properties that we would like this single, well-organized project to have? Ideally, we'd like to organize our projects so that I have the following properties:

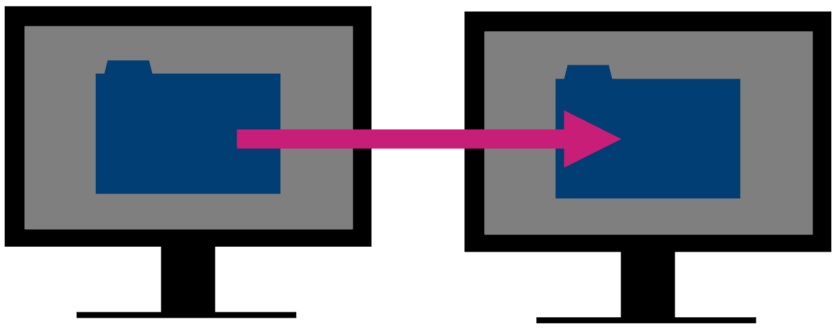
- Portable
- Version Control Friendly

- Reproducible
- IDE friendly.

Don't worry if you haven't heard of some of these terms already. We're going to look at each of them in a little bit of detail.

1.4.1 Portability

A project is said to be portable if it can be easily moved without breaking.



This might be a small move, like relocating the directory to a different location on your own computer. It might also mean a moderate move, say to another machine if yours dies just before a big deadline. Alternatively, it might be a large shift - to be uses by another person who is using a different operating system.

From this thought experiment you can see that there's a full spectrum of how portable a project may or may not need to be.

1.4.2 Version Control Friendly

A project under Version Control has all changes tracked either manually or automatically. This means that snapshots of the project are taken regularly as it gradually develops and evolves over time. Having these snapshots as many, incremental changes are made to the project allow it to be rolled back to a specific previous state if something goes wrong.

A version controlled pattern of working helps to avoid the horrendous state that we have all found ourselves in - renaming `final_version.doc` to `final_final_version.doc` and so on.

By organising your workflow around incremental changes helps you to acknowledge that no work is ever finally complete. There will always be small changes that need to be done in the future.

1.4.3 Reproducibility

> A study is reproducible if you can take the original data and the computer code used to analyze the data and recreate all of the numerical findings from the study.
>
> Broman et al (2017). "Recommendations to Funding Agencies for Supporting Reproducible Research"

In their paper, Broman et al define reproducibility as a project where you can take the original data and code used to perform the analysis and using these we create all of the numerical findings of the study.

This definition leads naturally to several follow-up questions.

Who exactly is *you* in this definition? Does it specifically mean yourself in the future or should someone else with access to all that data and code be able to recreate your findings too? Also, should this reproducibility be limited to just the numerical results? Or should they also be able to create the associated figures, reports and press releases?

Another important question is *when* this project needs to be reproduced. Will it be in a few weeks time or in 10 years time? Do you need to protect your project from changes in dependencies, like new versions of packages or modules? How about different versions of R or Python? Taking this time scale out even further, what about different operating systems and hardware?

It's unlikely you'd consider someone handing you a floppy disk of code that only runs on Windows XP to be acceptably reproducible. Sure, you could probably find a way to get it to work, but that would be an awful lot of effort on your end.

That's perhaps a bit of an extreme example, but it emphasizes the importance of clearly defining the level of reproducibility that you're aiming for within every project you work on. This example also highlights the amount of work that can be required to reproduce an analysis, especially after quite some time. It is important to explicitly think about how we dividing that effort between ourselves as the original developer and the person trying to reproduce the analysis in the future.

1.4.4 IDE Friendly

Our final desirable property is that we'd like our projects to play nicely with integrated development environments.

When you're coding document and writing your data science projects it'd be possible for you to work entirely in either a plain text editor or typing code directly at the command line. While these approaches to a data science workflow have the benefit of simplicity, they also expect a great deal from you as a data scientist.

These workflows expect that you should type everything perfectly accurately every time, that you recall the names and argument orders of every function you use, and that you are constantly aware of the current state of all objects within your working environment.

Integrated Development Environments (IDEs) are applications that help to reduce this burden, helping make you a more effective programmer and data scientist. IDEs offer tools like code completion and highlighting to make your code easier to read and to write. They offer tools for debugging, to fix where things are going wrong, and they also offer environment panes so that you don't have to hold everything in your head all at once. Many IDEs also often have templating facilities. These let you save and reuse snippets of code so that you can avoid typing out repetitive, boilerplate code and introducing errors in the process.

Even if you haven't heard of IDEs before, you've likely already used one. Some common examples might be RStudio for R-users, PyCharm for python users, or Visual Studio as a more language agnostic coding environment.

Whichever of these we use, we'd like our project to play nicely with them. This lets us reap their benefits while keeping our project portable, version controlled, and reproducible for someone working with a different set-up.

1.5 Project Structure

I've given a pretty exhaustive argument for why having a single directory for each project is a good idea. Let's now take a look *inside* that directory and define a common starting layout for the content of all of your projects.

Having this sort of project directory template will mean that you'll always know where to find what you're looking for and other members of your team will too. Again, before we start I'll reiterate that we're taking an opinionated approach here and providing a sensible starting point for organizing many projects.

Every project is going to be slightly different and some might require slight alterations to what I suggest here. Indeed, even if you start as I suggest then you might have to adapt your project structure as it develops and grows. I think it's helpful to consider yourself as a tailor when making these changes. I'm providing you with a one size fits all design, that's great for lots

of projects but perfect for none of them. It's your job to alter and refine this design for each individual case.

One final caveat before we get started: companies and businesses will many times have a house style how to write and organize your code or projects. If that's the case, then follow the style guide that your business or company uses. The most important thing here is to be consistent at both an individual level and across the entire data science team. It's this consistency that reaps the benefits.

Okay, so imagine now that you've been assigned a shiny new project and have created a single directory in which to house that project. Here we've, quite imaginatively, called that directory `exciting-new-project`. What do we populate this folder with?

In the rest of this video, I'll define the house-style for organizing the root directory of your data science projects in this module.

Within the project directory there will be some subdirectories, which you can tell a folders in this file structure because they have a forward slash following their names. There will also be some files directly in the root directory. One of these is called `readme.md` and the another called either `makefile` or `make.r`. We're going to explore each of these files and directories in turn.

1.5.1 README.md

Let's begin with the readme file. This gives a brief introduction to your project and gives information on what the project aims to do. The readme file should describe how to get started using the project and how to contribute to its development.

exciting-new-project

README.md

data/

src/

tests/

analyses/

outputs/

reports/

makefile

exciting-new-project

README.md

data/

src/

tests/

analyses/

outputs/

reports/

makefile

Exciting-new-project/README.md

```
# My Exciting New Project

Here is a short description of what
the project is about.

- Aim 1
- Aim 2
- Aim 3

The code and analysis are structured
as follows:

## data/

This directory contains all raw and
derived data sets. Further information
can be found in
[metadata.txt](./data/metadata.txt)…
```

The readme is written either in a plain text format so `readme.txt` or in markdown format `readme.md`. The benefit of using markdown is that it allows some light formatting such as sections headers and lists using plain text characters. Here you can see me doing that by using hashes to mark out first and second level headers and using bullet points for a unnumbered list. Whichever format you use, the readme file for your project is always stored in the root directory and is typically named in all uppercase letters.

The readme file should be the first thing that someone who's new to your project reads. By placing the readme in the root directory and capitalising the file name you are increase the visibility of this file and increase the chances of this actually happening.

An additional benefit to keeping the readme in the root directory of your project is that code hosting services like GitHub, GitLab or BitBucket will display the contents of that readme file next to the contents of your project. Those services will also nicely format any markdown that you use for you in your readme file.

When writing the readme, it can be useful to imaginge that you are writing this for a new, junior team member. The readme file should let them get started with the project and make some simple contributions after reading only that file. It might also link out to more detailed project documentation that will help the new team member toward a more advanced understanding or complex contribution.

1.5.2 Inside the README

let's take a quick aside to see in more detail what should be covered within a readme file.

A readme we should include the name of the project, which should be self-explanatory (so nothing like my generic choice of `exciting-new-project`). The readme should also give the project status, which is just a couple of sentences to say whether your project is still under development, the version oft the current release or, on the other end of the project life-cycle, if the project is being deprecated or closed.

Following this, we should also include a description of your project. This will state the purpose of your work and to provide, or link to, any additional context or references that visitors aren't assumed to be familiar with.

If your project involves code or depends on other packages then you should give some instruction on how to install those dependencies and run your code. This might just be text but it could also include things like screenshots, code snippets, gifs or a video of the whole process.

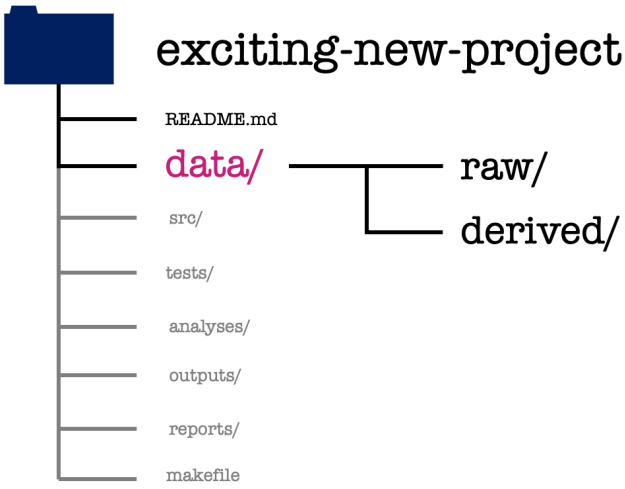
It's also a good practice to include some simple examples of how to use the code within your project an the expected results, so that new users can confirm that everything is working on their local instance. Keep the examples as simple and minimal as you can so that new users

For longer or more complicated examples that aren't necessary in this short introductory document you can add links to those in the readme and explain them in detail elsewhere.

There should ideally be a short description of how people can report issues with the project and also how people can get started in resolving those issues or extend the project in some way.

That leads me on to one point that I've forgotten to list here. There there should be a section listing the authors of the work and the license in which under which it's distributed. This is to give credit to all the people who've contributed to your project and the license file then says how other people may use your work. The license declares how other may use your project and whether they have to give direct attribution to your work in any modifications that they use.

1.5.3 data



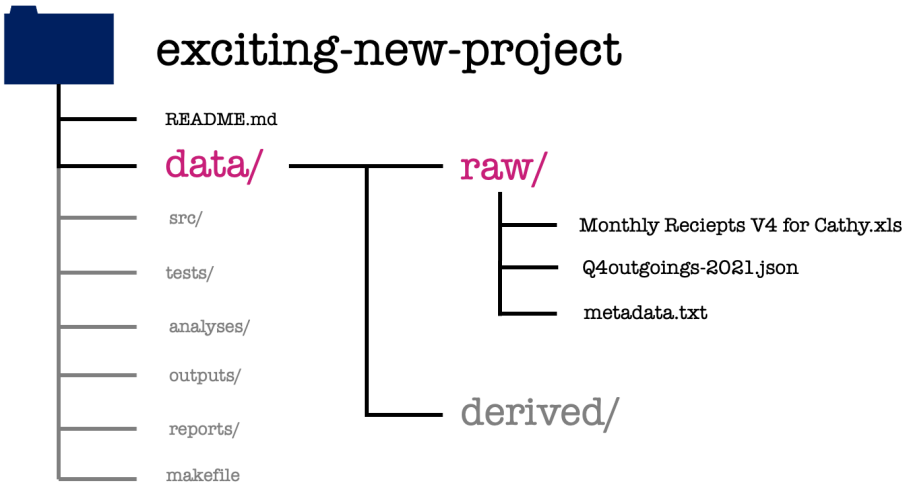
Moving back to our project structure, next we have the data directory.

The data directory will have two subdirectories one called `raw` and one called `derived`. All data that is not generate as part of your project is stored in the `raw` subdirectory. To ensure that a project is reproducible, data in the Raw folder should never be edited or modified.

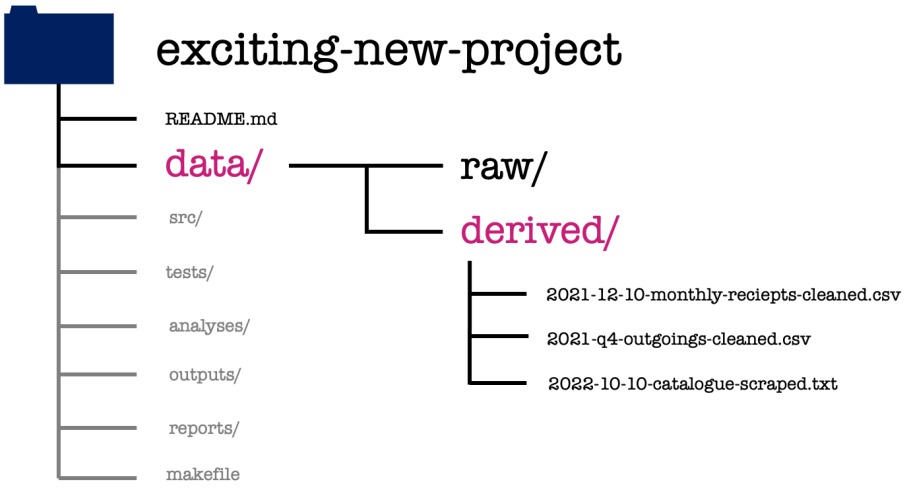
In this example we've got two different data types: an Excel spreadsheet the XLS file and a JSON file. These files are exacty as we received them from our project stakeholder.

The text file `metadata.txt` is a plain text file explaining the contents and interpretation of each of the raw data sets. This metadata should include descriptions of all the measured variables, the units that are recorded in, the date the file was created or acquired, and the source from which it was obtained.

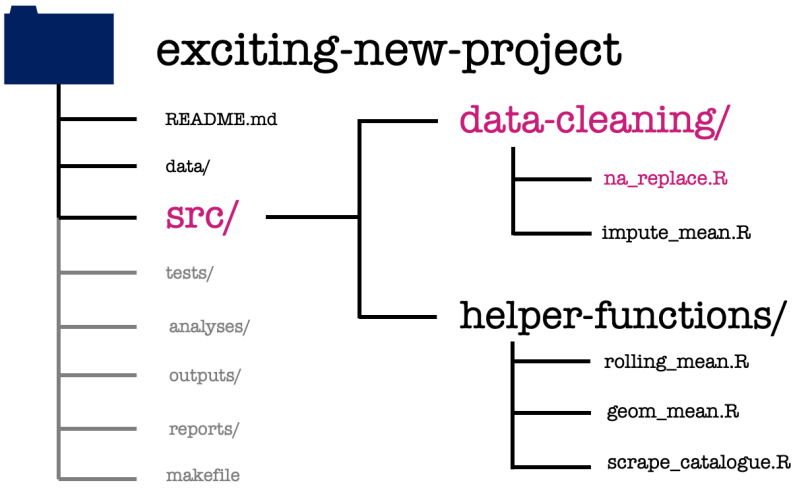
The raw data likely isn't going to be in a form that's amenable to analyzing straight away. To get the data into a more pleasant form to work, it will require some data manipulation and



cleaning. Any manipulation or cleaning that is applied should be well documented and the resulting cleaned files saved within the `derived` data directory.



In our exciting new project, we can see the clean versions of the previous data sets which are ready for modelling. There's also a third file in this folder. This is data that we've acquired for ourselves through web scraping, using a script within the project.



1.5.4 src

The `src` or source directory contains all the source code for your project. This will typically be the functions that you've written to make the analysis or modelling code more accessible.

Here we've saved each function in its own R script and, in this project, we've used subdirectories to organise these by their use case. We've got two functions used in data cleaning: the first replaces `NA` values with a given value, the second replaces these by the mean of all non-missing values.

We also have three helper functions: the first two calculate rolling mean and the geometric mean of a given vector, the third is a function that scrapes the web data we saw in the derived data subdirectory.

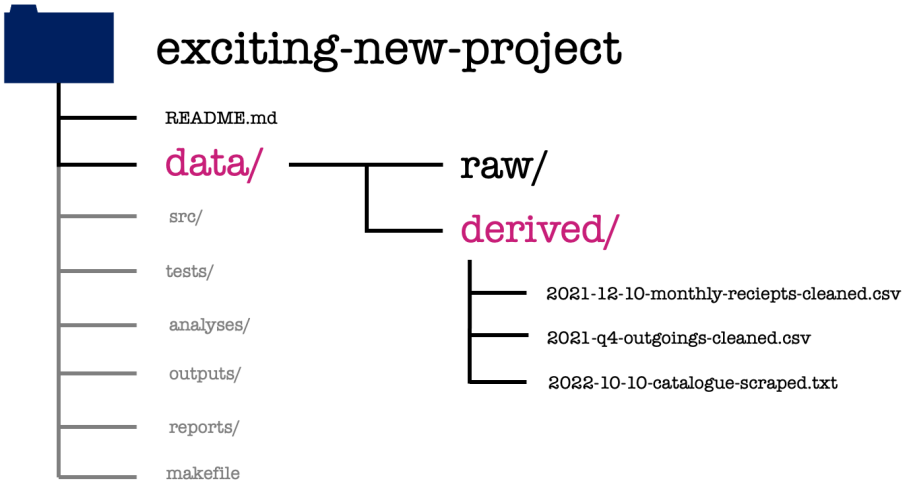
1.5.5 tests

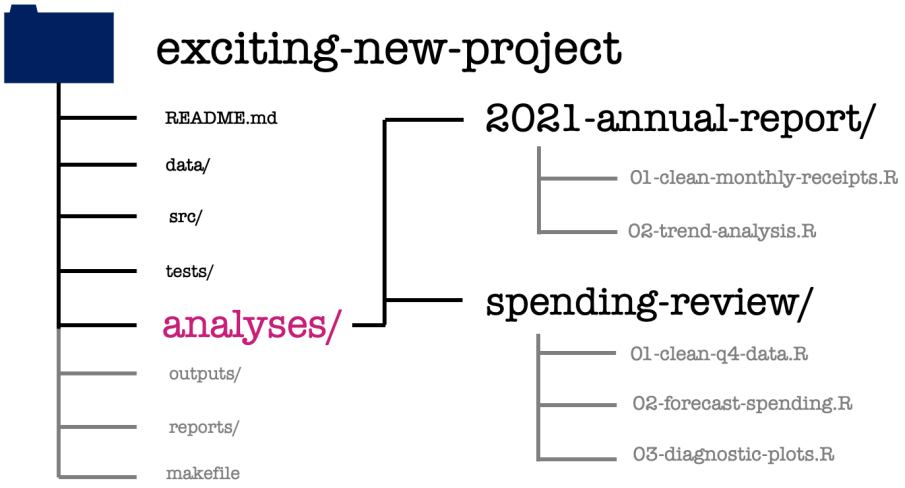
Moving on then to the tests directory. The structure of this directory mirrors that of the source directory. Each function file has its own counterpart file of tests.

These test files provide example sets of inputs and the expected outputs for each function. The test files are used to check edge cases of a function or to assure yourself that you haven't broken anything while fixing some small bug or adding new capabilities to that function.

1.5.6 analyses

The analyses directory contains what you probably think of as the bulk of your data science work. It's going to have one subdirectory for each major analysis that's performed within your


exciting-new-project
README.md
data/
raw/
derived/
2021-12-10-monthly-reciepts-cleaned.csv
2021-q4-outgoings-cleaned.csv
2022-10-10-catalogue-scraped.txt
src/
tests/
analyses/
outputs/
reports/
makefile

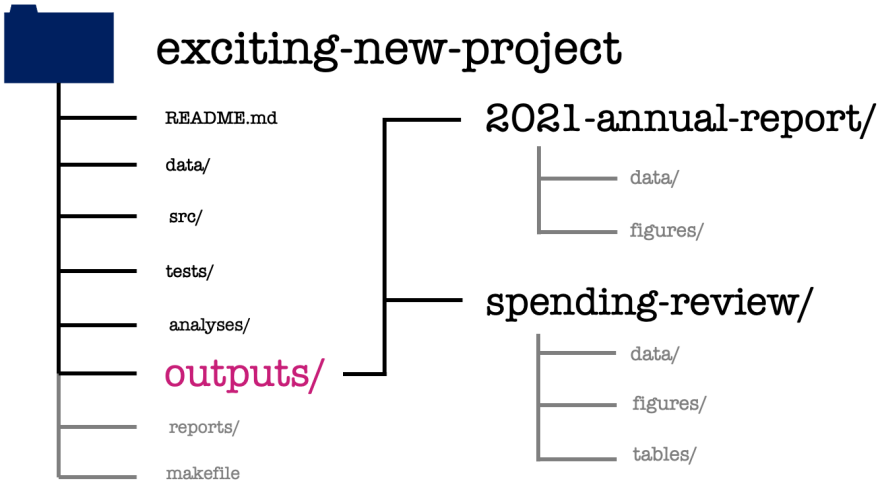

exciting-new-project
README.md
data/
src/
tests/
analyses/
outputs/
reports/
makefile
2021-annual-report/
01-clean-monthly-receipts.R
02-trend-analysis.R
spending-review/
01-clean-q4-data.R
02-forecast-spending.R
03-diagnostic-plots.R

project and within each of these there might be a series of steps that we collect into separate scripts.

The activity performed at each step is made clear by the name of each script, as is the order in which we're going to perform these steps. Here we can see the scripts used for the 2021 annual report. First is a script used to take the raw monthly receipts and produce the *cleaned* version of the same data set that we saw earlier. This is followed by a trend analysis of this cleaned data set.

Similarly for the spending review we have a data cleaning step, followed by some forecast modelling and finally the production of some diagnostic plots to compare these forecasts.

1.5.7 outputs

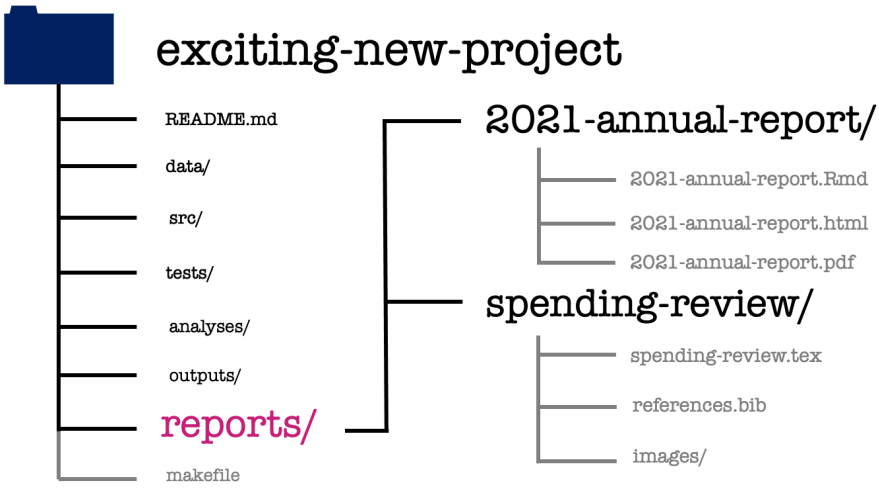


The outputs directory has again one subdirectory for each meta-analysis within the project. These are then further organized by the output type whether that be some data, a figure, or a table.

Depending on the nature of your project, you might want to use a modified subdirectory structure here. For example, if you're doing several numerical experiments then you might want to arrange your outputs by experiment, rather than by output type.

1.5.8 reports

The reports directory is then where everything comes together. This is where the written documents that form the final deliverables of your project are created. If these final documents



are written in LaTeX or markdown, both the source and the compiled documents can be found within this directory.

When including content in this report, for example figures, I'd recommend against making copies of those figure files within the reports directory. If you do that, then you'll have to manually update the files every time you modify them. Instead you can use relative file paths to include these figures. Relative file paths specify how to get to the image, starting from your TeX document and moving up and down through the levels of your project directory.

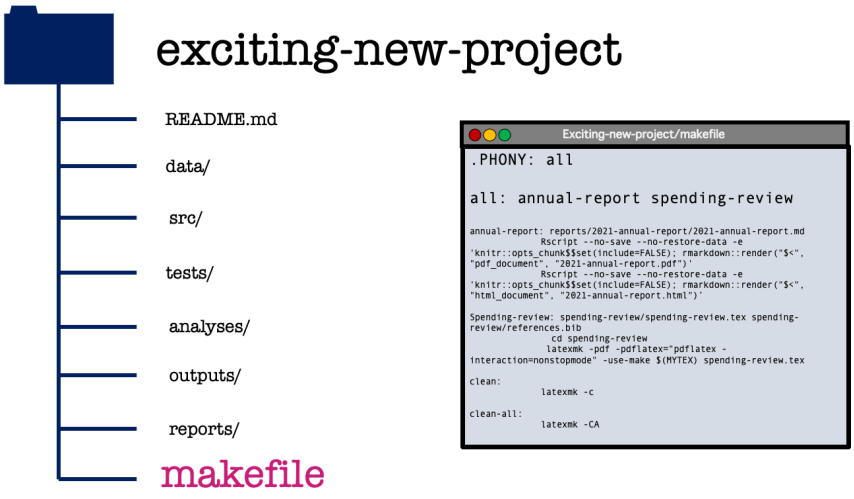
If you're not using markdown or LaTeX to write your reports, but instead use an online platform like overleaf as a latex editor or Google docs to write collaboratively then links to them in the reports directory using additional readme files. Make sure you set the read and write permissions for those links appropriately, too.

When using these online writing systems, you'll have to manually upload and update your plots whenever you modify any of your earlier analysis. That's one of the drawbacks of these online tools that has to be traded off against their ease of use.

In our exciting new project, here we can see that the annual report is written in a markdown format, which is compiled to both HTML and PDF. The spendiing review is written in LaTeX and we only have the source for it, we don't have the compiled pdf version of the document.

1.5.9 make file

The final element of our template project structure is a make file. We aren't going to cover how to read or write make files in this course. Instead, I'll give you a brief description of what they are and what it is supposed to do.



At a high level, the make file is just a text file. What makes it special is what it contains. Similar to a shell or a bash script, make file contains code that could be run at the command line. This code will create or update each element of your project.

The make file defines shorthand commands for the full lines of code that create each element of your project. The make file also records the order in which these operations have to happen, and which of these steps are dependent on one another. This means that if one step part of your project is updated then any changes will be propagated through your entire project. This is done in quite a clever way so the only part of your projects that are re-run are those that need to be updated.

We're omitting make files from this course not because they're fiendishly difficult to write or read, but rather because they require a reasonable foundation in working at the command line to be understood. What I suggest you do instead throughout this course is to create your own R or markdown file called make. This file will define the intended running order and dependencies of your project and if it is an R file, it might also automate some parts of your analysis.

1.6 Wrapping up

Wrapping up then, that's everything for this chapter.

I've introduced a project structure that will serve you well as a baseline for the vast majority of projects in data science.

In your own work, remember that the key here is standardisation. Working consistently across projects, a company or a group is more important than sticking rigidly to the particular structure that I have defined here.

There are two notable exceptions where you probably don't want to use this project structure. That's when you're building an app or you're building a package. These require specific organisation of the files within your project directory. We'll explore the project structure used for package development during the live session this week.

2 Naming Files

> **Note**
>
> Effective Data Science is still a work-in-progress. This chapter should be readable but is currently undergoing final polishing.
> If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

2.1 Introduction

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> **Phil Karlton, Netscape Developer**

When working on a data science project we can in principle name directories, files, functions and other objects whatever we like. In reality though, using an ad-hoc system of naming is likely to cause confusion, headaches and mistakes. We obviously want to avoid all of those things, in the spirit of being kind to our current colleges and also to our future selves.

Coming up with good names is an art form. Like most art, naming things is an activity that you get better at with practice. Another similarity is that the best naming systems don't come from giving data scientists free reign over their naming system. Like all art, the best approaches to naming things give you strong guidelines and boundaries within which to express your creativity and skill.

In this lecture we'll explore what these boundaries and what we want them to achieve for us. The content of this lecture is based largely around a talk of the same name given by Jennifer Bryan and the tidyverse style guide, which forms the basis of Google's style guide for R programming.

2.2 Naming Files

We'll be begin by focusing in on what we call our files. That is, we'll first focus on the part of the file name that comes before the dot. In the second part of this video, we'll then cycle back around to discuss file extensions.

2.2.1 What do we want from our file names?

Before we dive into naming files, we should first consider what we want from the file names that we choose. There are three key properties that that we would like to satisfy.

1. Machine Readable
2. Human Readable
3. Order Friendly

Thee first desirable property is for file names to be easily readable by computers, the second is for the file names to be easily readable by humans and finally the file names should take advantage of the default ordering imposed on our files.

This set of current file names is sorely lacking across all of these properties:

```
abstract.docx
Effective Data Science's module guide 2022.docx
fig 12.png
Rplot7.png
1711.05189.pdf
HR Protocols 2015 FINAL (Nov 2015).pdf
```

We want to provide naming conventions to move us toward the better file names listed below.

```
2015-10-22_human-resources-protocols.pdf
2022_effective-data-science-module-guide.docx
2022_RSS-conference-abstract.docx
fig12_earthquake-timeseries.png
fig07_earthquake-location-map.png
ogata_1984_spacetime-clustering.pdf
```

Let's take a few minutes to examine what exactly we mean by each of these properties.

2.2.2 Machine Readable

What do we mean by machine readable file names?

- Easy to compute on by *deliberate use of delimiters*:
  - `underscores_separate_metadata`, `hyphens-separate-words`.
- Play nicely with *regular expressions* and *globbing*:
  - avoid spaces, punctuation, accents, cases;
  - `rm Rplot*.png`

Machine readable names are useful when:

- *managing files*: ordering, finding, moving, deleting:
- *extracting information* directly from file names,
- *working programmatically* with file names and regex.

When we are operating on a large number of files it is useful to be able to work with them programmatically.

One example of where this might be useful is when downloading assessments for marking. This might require me to unzip a large number of zip files, copying the pdf report from each unzipped folder into a single directory and all of the R scripts from each unzipped folder into another directory. The marked scripts and code then need to be paired back up in folders named by student, and re-zipped ready to be returned.

This is *monotonously* dull and might work for ~50 students but not for ~5000. Working programmatically with files is the way to get this job done efficiently. This requires the file names to play nicely with the way that computers interpret file names, which they regard as a string of characters.

It is often helpful to have some meta-data included in the file name, for example the student's id number and the assessment title. We will use an underscore to separate elements of meta-data within the file name and a hyphen to separate sub-elements of meta-data, for example words within the assessment title.

Regular expressions and globbing are two ideas from string manipulation that you may not have met, but which will inform our naming conventions. Regular expressions allow you to search for strings (in our case file names) that match a particular pattern. Regular expressions can do really complicated searches but become gnarly when you have to worry about special characters like spaces, punctuation, accents and cases, so these should be avoided in file names.

A special type of regular expression is called globbing where a star is used to replace any number of subsequent characters in a file name, so that here we can delete all png images that

begin with Rplot using a single line of code. Globbing becomes particular powerful when you use a consistent structure to create your file names.

As in the assessment marking example, having machine readable file names is particularly useful when managing files, such as ordering, finding, moving or deleting them. Another example of this is when your analysis requires you to load a large number of individual data files.

Machine readable file names are also useful for extracting meta-information from files without having to open them in memory. This is particularly useful when the files might be too large to load into memory, or you only want to load data from a certain year.

The final benefit we list here is the scalability, reduction in drudgery and lowered risk for human error when operating on a very large number of files.

2.2.3 Order Friendly

The next property we will focus on also links to how computers operate. We'd like our file names to exploit the default orderings used by computers. This means starting file names with character strings or metadata that allow us order our files in some meaningful way.

2.2.3.1 Running Order

One example of this is where there's some logical order in which your code should be executed, as in the example analysis below.

```
diagnositc-plots.R
download.R
runtime-comparison.R
...
model-evaluation.R
wrangle.R
```

Adding numbers to the start of these file names can make the intended ordering immediately obvious.

```
00_download.R
01_wrangle.R
02_model.R
...
09_model-evaluation.R
10_model-comparison-plots.R
```

Starting single digit numbers with a leading 0 is a very good idea here to prevent script 1 being sorted in with the tens, script 2 in with the twenties and so on. If you might have over 100 files, for example when saving the output from many simulations, use two or more zeros to maintain this nice ordering.

2.2.3.2 Date Order

A second example of orderable file names is when the file has a date associated with it. This might be a version of a report or the date on which some data were recorded, cleaned or updated.

```
2015-10-22_human-resources-protocols.pdf
...
2022-effective-data-science-module-guide.docx
```

When using dates, in file names or elsewhere, you should conform to the ISO standard date format.

> ISO 8601 sets an international standard format for dates: `YYYY-MM-DD`.

This format uses four numbers for the year, followed by two numbers for the month and two numbers of the day of the month. This structure mirrors a nested file structure moving from least to most specific. It also avoids confusion over the ordering of the date elements. Without using the ISO standard a date like `04-05-22` might be interpreted as the fourth of May 2022, the fifth of April 2022, or the twenty-second of May 2004.

2.2.4 Human Readable

The final property we would like our file names to have is human readability. This requires the names of our files to be meaningful, informative and easily read by real people.

The first two of these are handled by including appropriate metadata in the file name. The ease with which these are read by real people is determined by the length of the file name and by how that name is formatted.

There are lots of formatting options with fun names like `camelCase`, `PascalCase`, and `snake_case`.

```
easilyReadByRealPeople (camelCase)
EasilyReadByRealPeople (PascalCase)
easily_read_by_real_people (snake_case)
easily-read-by-real-people (skewer-case)
```

There is weak evidence to suggest that snake case and skewer case are most the readable. We'll use a mixture of these two, using snake case *between* metadata items and skewer case *within* them. This has a slight cost to legibility, in a trade-off against making computing on these file names easier.

The final aspect that you have control over is the length of the name. Having short, evocative and useful file names is not easy and is a skill in itself. For some hints and tips you might want to look into tips for writing URL slugs. These are last part of a web address that are intended to improve accessibility by being immediately and intuitively meaningful to any user.

2.2.5 Naming Files - Summary

1. File names should be meaningful, informative and scripts end in `.r`
2. Stick to letters, numbers underscores (`_`) and hyphens (`-`).
3. Pay attention to capitalisation `file.r` $\neq$ `File.r` on all operating systems.
4. Show order with left-padded numbers or ISO dates.

2.3 File Extensions and Where You Work

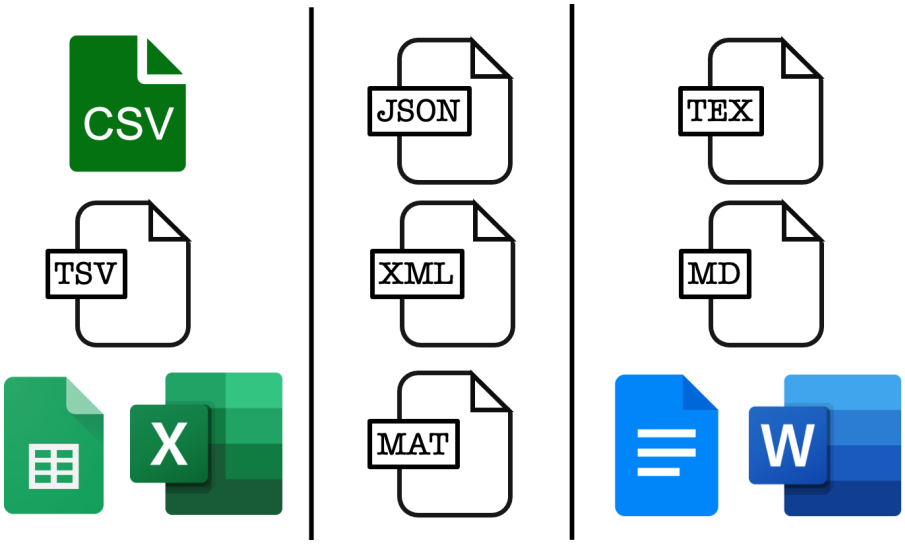
So far we have focused entirely on what comes before the dot, that is the file name.Equally, if not more, important is what comes after the dot, the file extension.

```
example-script.r
example-script.py

project-writeup.doc
project-writeup.tex
```

The file extension describes how information is stored in that file and determines the software that can use, view or run that file.

You likely already use file extensions to distinguish between code scripts, written documents, images, and notebook files. We'll now explore the benefits and drawbacks of various file types with respect to several important features.



2.3.1 Open Source vs Proprietary File Types

The first feature we'll consider is whether the file type is open source, and can be used by anyone without charge, or if specialist software must be paid for in order to interact with those files.

In the figure above, each column represents a different class of file, moving left to right we have example file types for tabular data, list-like data and text documents. File types closer to the top are open source while those lower down rely on proprietary software, which may or may not require payment.

To make sure that our work is accessible to as many people as possible we should favour the open source options like csv files over Google sheets or excel, JSON files over Matlab data files, and tex or markdown over a word or Google doc.

This usually has a benefit in terms of project longevity and scalability. The open source file types are often somewhat simpler in structure, making them more robust to changes over time less memory intensive.

To see this, let's take a look inside some data files.

2.3.2 Inside Data Files

2.3.2.1 Inside a CSV file

CSV or comma separated value files are used to store tabular data.

In tabular data, each row of the data represents one record and each column represents a data value.

A csv encodes this by having each record on a separate line and using commas to separate values with that record. You can see this by opening a csv file in a text editor such as notepad.

The raw data stores line breaks using `\n` and indicates new rows by `\r`. These backslashed indicae that these are *escape characters* with special meanings, and should not be literally interpreted as the letters n and r.

```
#> [1] "Name,Number\r\nA,1\r\nB,2\r\nC,3"
```

When viewed in a text editor, the example file would look something like this.

```
Name,Number
A,1
B,2
C,3
```

2.3.2.2 Inside a TSV file

TSV or tab separated value files are also used to store tabular data.

Like in a csv each record is given on a new line but in a tsv tabs rather than commas are used to separate values with each record. This can also be seen by opening a tsv file in a text editor such as notepad.

```
#> [1] "Name\tNumber\r\nA\t1\r\nB\t2\r\nC\t3"
```

```
Name    Number
A   1
B   2
C   3
```

One thing to note is that tabs are a separate character and are not just multiple spaces. In plain text these can be impossible to tell apart, so most text editors have an option to display tabs differently from repeated spaces, though this is usually not enabled by default.

2.3.2.3 Inside an Excel file

When you open an excel file in a text editor, you will immediately see that this is not a human interpretable file format.

```
504b 0304 1400 0600 0800 0000 2100 62ee
9d68 5e01 0000 9004 0000 1300 0802 5b43
6f6e 7465 6e74 5f54 7970 6573 5d2e 786d
6c20 a204 0228 a000 0200 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
.... .... .... .... .... .... .... ....
0000 0000 0000 0000 ac92 4d4f c330 0c86
ef48 fc87 c8f7 d5dd 9010 424b 7741 48bb
2154 7e80 49dc 0fb5 8da3 241b ddbf 271c
1054 1a83 0347 7fbd 7efc cadb dd3c 8dea
.... .... .... .... .... .... .... ....
```

Each entry here is a four digit hexadecimal number and there are a lot more of them than we have entries in our small table.

This is because excel files can carry a lot of additional information that a csv or tsv are not able to, for example cell formatting or having multiple tables (called sheets by excel) stored within a single file.

This means that excel files take up much more memory because they are carrying a lot more information than is strictly contained within the data itself.

2.3.2.4 Indise a JSON file

JSON, or Java Script Object Notation, files are an open source format for list-like data. Each record is represented by a collection of `key:value` pairs. In our example table each entry has two fields, one corresponding to the `Name` key and one corresponding to the `Number` key.

```
[{
    "Name": "A",
    "Number": "1"
}, {
    "Name": "B",
    "Number": "2"
}, {
    "Name": "C",
    "Number": "3"
```

```
}]
```

This list-like structure allows non-tabular data to be stored by using a property called nesting: the value taken by a key can be a single value, a vector of values or another list-like object.

This ability to create nested data structures has lead to this data format being used widely in a range of applications that require data transfer.

2.3.2.5 Inside an XML file

XML files are another open source format for list-like data, where each record is represented by a collection of `key:value` pairs.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <row>
    <Name>A</Name>
    <Number>1</Number>
  </row>
  <row>
    <Name>B</Name>
    <Number>2</Number>
  </row>
  <row>
    <Name>C</Name>
    <Number>3</Number>
  </row>
</root>
```

The difference from a JSON file is mainly in how those records are formatted within the file. In a JSON file this is designed to look like objects in the Java Script programming language and in XML the formatting is done to look like html, the markup language used to write websites.

2.3.3 A Note on Notebooks

- There are two and a half notebook formats that you are likely to use: `.rmd`, `.ipynb` or alternatively `.qmd`.
- R markdown documents `.rmd` are plain text files, so are very human friendly.
- ***JuPy***te***R*** notebooks have multi-language support but are not so human friendly (JSON in disguise).

- Quarto documents offer the best of both worlds and more extensive language support. Not yet as established as a format.

In addition to the files you read and write, the files that you code in will largely determine your workflow.

There are three main options for the way that you code: first is typing it directly at the command line, second is using a text editor or IDE to write scripts and third is writing a notebook that mixes code, text and output together in a single document.

We'll compare these methods of working soon, but first let's do a quick review of what notebooks are available to you and why you might want to use them.

As a data scientist, there are two and a half notebook formats that you're likely to have met before. The first two are Rmarkdown files for those working predominantly in R and interactive Python or jupyter notebooks for those working predominantly in Python. The final half format are quarto markdown documents, which are relatively new and extend the functionality of Rmarkdown files to provide multi-language support.

The main benefit of R markdown documents is that they're plain text files, so they're very human friendly and work very well with version control software like git. ***JuPy***te***R*** notebooks have the benefit of supporting code written in Julia, Python or R, but are not so human friendly - under the hood these documents are JSON files that should not be edited directly (because a misplaced bracket will break them!).

Quarto documents offer the best of both worlds, with plain text formatting and even more extensive language support than jupyter notebooks. Quarto is a recent extension of Rmarkdown, which is rapidly becoming popular in the data science community. Quarto also allows you to create a wider range of documents, including websites, these course notes and the associated slides.

Each format has its benefits and drawbacks depending on the context in which they are used and all have some shared benefits and limitations by nature of them all being notebook documents.

2.3.4 File Extensions and Where You Code

Property	Notebook	Script	Command Line
reproducible	~		X
readable	~		~
self-documenting		X	X
in production	X		~
ordering / automation	~		~

The main benefit of notebook documents is that they are self-documenting, in that they can mix the documentation, code and report all into a single document. Notebooks also provide a level of interactivity when coding that is not possible when working directly at the command line or using a text editor to write scripts. This limitation is easily overcome by using an integrated development environment when scripting, rather than a plain text editor.

Writing code in `.r` files is not self-documenting but this separation of code, documentation and outputs has many other benefits. Firstly, the resulting scripts provide a reproducible and automatable workflow, unlike one-off lines of code being run at the command line. Secondly, using an IDE to write these provides you with syntax highlighting and code linting features to help you write readable and accurate code. Finally, the separation of code from documentation and output allows your work to be more easily or even directly put into production.

In this course we will advocate for a scripting-first approach to data science, though notebooks and command line work definitely have their place.

Notebooks are great as teaching and rapid development tools but have strong limitations with being put into production. Conversely, coding directly at the command line is perfect for simple one-time tasks but it leaves no trace of your workflow and leads to an analysis that cannot be easily replicated in the future.

2.3.5 Summary

Finally, let's wrap things up by summarising what we have learned about naming files.

Before the dot we want to pick file names that machine readable, human friendly and play nicely with the default orderings provided to us.

Name files so that they are:

- Machine Readable,
- Human Readable,
- Order Friendly.

After the dot, we want to pick file types that are widely accessible, easily read by humans and allow for our entire analysis to be reproduced.

Use document types that are:

- Widely accessible,
- Easy to read and reproduce,
- Appropriate for the task at hand.

Above all we want to name our files and pick our file types to best match with the team we are working in and the task that is at hand.

3 Code

> **Note**
>
> Effective Data Science is still a work-in-progress. This chapter should be readable but is currently undergoing final polishing.
> If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

3.1 Introduction

We have already described how we might organise an effective data science project at the directory and file level. In this chapter we will delve one step deeper and consider how we can structure our work within those files. In particular, we'll focus on code files here.

We'll start by comparing the two main approaches to structuring our code, namely functional programming and object oriented programming. We'll then see how we should order code within our scripts and conventions on how to name the functions and objects that we work with in our code.

Rounding up this chapter, we'll summarise the main points from the R style guide that we will be following in this course and highlight some useful packages for writing effective code.

3.2 Functional Programming

A functional programming style has two major properties:

- Object immutability,
- Complex programs written using function composition.

This first point here states that the original data or objects should never be modified or altered by the code we write. We have met this idea before when making new, cleaner versions of our raw data but taking care to leave the original messy data intact. Object immutability is the exact same idea but in a code context rather than data context.

Secondly, in functional programming, complex problems are solved by decomposing them into a series of smaller problems. A separate, self-contained function is then written to solve each sub-problem. Each individual function is, in itself, simple and easy to understand. This makes these small functions easy to test and easy to reuse in many places. Code complexity is then built up by composing these functions in various ways.

It can be difficult to get into this way of thinking, but people with mathematical training often find it quite natural. This is because mathematicians have many years of experience in working with function compositions in the abstract, mathematical sense.

$$y = g(x) = f_3 \circ f_2 \circ f_1(x).$$

3.2.1 The Pipe Operator

One issue with functional programming is that lots of nested functions means that there are also lots of nested brackets. These start to get tricky to keep track of when you have upwards of 3 functions being composed. This reading difficulty is only exacerbated if your functions have additional arguments on top of the original inputs.

```
log(exp(cos(sin(pi))))
#> [1] 1
```

The pipe operator `%>%` from the `{magrittr}` package helps with this issue. It works exactly like function composition: it takes the whatever is on the left (whether that is an existing object or the output of a function) and passes it to the following function call as the first argument of that function.

```
library(magrittr)
pi %>%
  sin() %>%
  cos() %>%
  exp() %>%
  log()
#> [1] 1
```

```
iris %>%
  head(n = 3)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
```

The pipe operator is often referred to as "syntactic sugar". This is because it doesn't add anything to your code in itself, but rather it makes your code *so* much more palatable to read.

In R versions 4.1 and greater, there's a built-in version of this pipe operator, `|>`. This is written using the vertical bar symbol followed by a greater than sign. To type the vertical bar, you can usually find it found above backslash on the keyboard. (Just to cause confusion, the vertical bar symbol is also called the pipe symbol and performs a similar operation in general programming contexts.)

```
library(magrittr)
pi |>
  sin() |>
  cos() |>
  exp() |>
  log()
#> [1] 1
```

```
iris |>
  head(n = 3)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
```

The base R pipe usually behaves in the same way as the pipe from magrittr, but there are a few cases where they differ. For reasons of back-compatibility and consistency we'll stick to the {magrittr} pipe in this course.

3.2.2 When not to pipe

Pipes are designed to put focus on the the actions you are performing rather than the object that you are preforming those operations on. This means that there are two cases where you should almost certainly not use a pipe.

The first of these is when you need to manipulate more than one object at a time. Using secondary objects as reference points (but leaving them unchanged) is of course perfectly fine, but pipes should be used when applying a sequence of steps to create a new, modified version of one primary object.

Secondly, just because you *can* chain together many actions into a single pipeline, that doesn't mean that you necessarily should. Very long sequences of piped operations are easier to read than nested functions, however they still burden the reader with the same cognitive load on

their short term memory. Be kind and create meaningful, intermediate objects with informative names. This will help the reader to more easily understand the logic within your code.

3.3 Object Oriented Programming

The main alternative to functional programming is object oriented programming.

- Solve problems by using lots of simple objects
- R has 3 OOP systems: S3, S4 and R6.
- Objects belong to a class, have methods and fields.
- Example: agent based simulation of beehive.

3.3.1 OOP Philosophy

In functional programming, we solve complicated problems by using lots of simple functions. In object oriented programming we solve complicated problems using lots of simple objects. Which of these programming approaches is best will depend on the particular type of problem that you are trying to solve.

Functional programming is excellent for most types of data science work. Object oriented comes into its own when your problem has many small components interacting with one another. This makes it great for things like designing agent-based simulations, which I'll come back to in a moment.

In R there are three different systems for doing object oriented programming (called S3, S4, and R6), so things can get a bit complicated. We won't go into detail about them here, but I'll give you an overview of the main ideas.

This approach to programming might be useful for you in the future, for example if you want to extend base R functions to work with new types of input, and to have user-friendly displays. In that case (Advanced R)[https://adv-r.hadley.nz/] by Hadley Wickham is an excellent reference text.

In OOP, each object belongs to a class and has a set of methods associated with it. The class defines what an object *is* and methods describe what that object can *do*. On top of that, each object has class-specific attributes or data fields. These fields are shared by all objects in a class but the values that they take give information about that specific object.

3.3.2 OOP Example

This is is all sounding very abstract. Let's consider writing some object oriented code to simulate a beehive. Each object will be a bee, and each bee is an instance of one of three bee classes: it might be a queen, a worker or a drone for example. Different bee classes have different methods associated with them, which describe what the bee can do, for example all bees would have 6 methods that let them move up, down, left, right, forward and backward within the hive. An additional "reproduce" method might only be defined for queen bees and a pollinate method might only be defined for workers. Each instance of a bee has its own fields, which give data about that specific bee. All bees have x, y and z coordinate fields giving their location within the hive. The queen class might have an additional field for their number of offspring and the workers might have an additional field for how much pollen they are carrying.

As the simulation progresses, methods are applied to each object altering their fields and potentially creating or destroying objects. This is very different from the preservation mindset of functional programming, but hopefully you can see that it is a very natural approach to many types of problem.

3.4 Structuring R Script Headers

TL;DR

- Start script with a comment of 1-2 sentences explaining what it > does.
- `setwd()` and `rm(ls())` are the devil's work.
- "Session" > "Restart R" or Keyboard shortcut: crtl/cmd + shift > + 0
- Polite to gather all `library()` and `source()` calls.
- Rude to mess with other people's set up using > `install.packages()`.
- Portable scripts use paths relative to the root directory of the project.

First things first, let's discuss what should be at the top of your R scripts.

It is almost always a good idea to start your file with a few commented out sentences describing the purpose of the script and, if you work in a large team, perhaps who contact with any questions about this script. (There is more on comments coming up soon, don't worry!)

It is also good practise to move all `library()` and `source()` calls to the top of your script. These indicate the packages and helper function that are dependencies of your script; it's useful to know what you need to have installed before trying to run any code.

That segues nicely to the next point, which is never to hard code package installations. It is extremely bad practise and very rude to do so because then your script might alter another person's R installation. If you don't know already, this is precisely the difference between an `install.packages()` and `library()` call: `install.packages()` will download the code

for that package to the users computer, while `library()` takes that downloaded code and makes it available in the current R session. To avoid messing with anyone's R installation, you should always type `install.package()` commands directly in the console and then place the corresponding `library()` calls within your scripts.

Next, it is likely that you, or someone close to you, will commit the felony of starting every script by setting the working directory and clearing R's global environment. This is *very* bad practice, it's indicative of a workflow that's not project based and it's problematic for at least two reasons. Firstly, the path you set will likely not work on anyone else's computer. Secondly, clearing the environment like this may *look* like it gets you back to fresh, new R session but all of your previously loaded packages will still be loaded and lurking in the background.

Instead, to achieve your original aim of starting a new R session, go to the menu and select the "Session" drop down then select "Restart R". Alternatively, you can use keyboard shortcuts to do the same. This is "crtl + shift + 0" on Windows and "cmd + shift + 0" on a mac. The fact that a keyboard shortcut exists for this should quite strongly hint that, in a reproducible and project oriented workflow, you should be restarting R quite often in an average working day. This is the scripting equivalent of "clear all output and rerun all" in a notebook.

Finally, let's circle back to the point I made earlier about setting the working directory. The reason that this will not work is because you are likely giving file paths that are specific to your computer, your operating system and your file organisation system. The chances of someone else having all of these the same are practically zero.

3.5 Portable File paths with `{here}`

```
# Bad - breaks if project moved
source("zaks-mbp/Desktop/exciting-new-project/src/helper_functions/rolling_mean.R")

# Better - breaks if Windows
source("../../src/helper_functions/rolling_mean.R")

# Best - but use here:here() to check root directory correctly identified
source(here::here("src","helper_functions","rolling_mean.R"))

# For more info on the here package:
vignette("here")
```

To fix the problem of person- and computer-specific file paths you can have two options.

The first is to use *relative* file paths. In this you assume that each R script is being run in its current location and my moving up and down through the levels of your project directory you point to the file that you need.

This is good in that it solves the problem of paths breaking because you move the project to a different location on your own laptop. However, it does not fully solve the portability problem because you might move your file to a different location *within* the same project. It also does not solve the problem that windows uses MacOS and linux use forward slashes in file paths with widows uses backslashes.

To resolve these final two issues I recommend using the `here()` function from the {here} package. This package looks for a `.Rproj` or `.git` file to identify the root directory of your project and creates file paths relative to the root of your project, that are suitable for the operating system the code is being run on.

It really is quite marvellous. For more information on how to use the here package, explore its chapter in R - What They Forgot, R for Data Science or this project oriented workflow blog post.

3.6 Code Body

Moving on now, we will go from the head to the body of the code. Having well named and organised code will facilitate both reading and understanding. Comments and sectioning do the rest of this work.

This section is designed as an introduction to the tidyverse style guide and not as a replacement to it. ### Comments

```
# This is an example script showing good use of comments and sectioning

library(here)
source(here("src","helper_functions","rolling_mean.R"))

#=============================================================================  <- 80 ch
# Major Section on Comments ----
#=============================================================================

#-----------------------------------------------------------------------------
##  Minor Section on inline comments ----
#-----------------------------------------------------------------------------
x <- 1:10 # this is an inline comment

#-----------------------------------------------------------------------------
##  Minor Section on full line comments ----
#-----------------------------------------------------------------------------
rolling_mean(x)
```

```
# This is an full line comment
```

Comments may be either short in-line comments at the end of a line or full lines dedicated to comments. To create either type of comment in R, simply type hash followed by one space. The rest of that line will not be evaluated and will function as a comment. If multi-line comments are needed simply start multiple lines with a hash and a space.

The purpose of these comments is to explain the *why* of what you are doing, not the what. If you are explaining *what* you are doing in most of your comments then you perhaps need to consider writing more informative function names, something we will return to in the general advice section.

Comments can also be used to add structure to your code, buy using commented lines of hyphens and equal signs to chunk your files into minor and major sections.

Markdown-like section titles can be added to these section and subsection headers. Many IDEs, such as RStudio, will interpret these as a table of contents for you, so that you can more easily navigate your code.

3.6.1 Objects are Nouns

- Object names should use only lowercase letters, numbers, and `_`.
- Use underscores (`_`) to separate words within a name. (`snake_case`)
- Use nouns, preferring singular over plural names.

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

When creating and naming objects a strong guideline in that objects should be named using short but meaningful nouns. Names should not include any special characters and should use underscores to separate words within the object name.

This is similar to our file naming guide, but note that hyphens can't be used in object names because this conflicts with the subtraction operator.

When naming objects, as far as possible use singular nouns. The main reason for this is that the plurisation rules in English are complex and will eventually trip up either you or a user of your code.

3.6.2 Functions are Verbs

- Function names should use only lower-case letters, numbers, and `_`.
- Use underscores (`_`) to separate words within a name. (`snake_case`)
- Suggest imperative mood, as in a recipe.
- Break long functions over multiple lines. 4 vs 2 spaces.

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()

long_function_name <- function(
    a = "a long argument",
    b = "another argument",
    c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

The guidelines for naming functions are broadly similar, with the advice that functions should be verbs rather than nouns.

Functions should be named in the imperative mood, like in a recipe. This is again for consistency; having function names in a range of moods and tenses leads to coding nightmares.

As with object names you should aim to give your functions and their arguments short, evocative names. For functions with many arguments or a long name, you might not be able to fit the function definition on a single line. In this case you can should place each argument on its own double indented line and the function body on a single indented line.

3.6.3 Casing Consistantly

As we have mentioned already, we have many options for separating words within names:

- `CamelCase`
- `pascalCase`
- `snakecase`
- `underscore_separated`
- `hyphen-separated`
- `point.separated`

For people used to working in Python it is tempting to use point separation in function names, in the spirit of methods from object oriented programming. Indeed, some base R functions even use this convention.

However, the reason that we advise against it is because it is already used for methods in some of R's inbuilt OOP functionality. We will use underscore separation in our work.

3.6.4 Style Guide Summary

1. Use comments to structure your code
2. Objects = Nouns
3. Functions = Verbs
4. Use snake case and consistant grammar

3.7 Further Tips for Friendly Coding

In addition to naming conventions the style guide gives lots of other guidance on writing code in a way that is kind to future readers of that code.

I'm not going to go repeat all of that guidance here, but the motivation for all of these can be boiled down into the following points.

- Write your code to be easily understood by humans.
- Use informative names, typing is cheap.

```
# Bad
for (i in dmt) {
  print(i)
}

# Good
```

```
for (temperature in daily_max_temperature) {
  print(temperature)
}
```

- Divide your work into logical stages, human memory is expensive.

When writing your code, keep that future reader in mind. This means using names that are informative and reasonably short, it also means adding white space, comments and formatting to aid comprehension. Adding this sort of structure to your code also helps to reduce the cognitive burden that you are placing on the human reading your code.

Informative names are more important than short names. This is particularly true when using flow controls, which are things like for loops and while loops. Which of these for loops would you like to encounter when approaching a deadline or urgently fixing a bug? Almost surely the second one, where context is immediately clear.

A computer doesn't care if you call a variable by only a single letter, by a random key smash (like `aksnbioawb`) or by an informative name. A computer also doesn't care if you include no white space your code - the script will still run. However, doing these things are friendly practices that can help yourself when debugging and your co-workers when collaborating.

3.8 Reduce, Reuse, Recycle

In this final section, we'll look at how you can make your workflow more efficient by reducing the amount of code you write, as well as reusing and recycling code that you've already written.

3.8.1 DRY Coding

This idea of making your workflow more efficient by reducing, reusing and recycling your code is summarised by the DRY acronym: don't repeat yourself.

This can be boiled down to three main points:

- if you do something twice in a single script, then write a function to do that thing;
- if you want to use your function elsewhere *within* your project, then save it in a separate script;
- If you want to use your function *across* projects, then add it to a package.

Of course, like with scoping projects in the first place, this requires some level of clairvoyance: you have to be able to look into the future and see whether you'll use a function in another script or project. This is difficult, bordering on impossible. So in practice, this is done retrospectively - you find a second script or project that needs a function then pull it out its own separate file or include it in a package.

As a rule of thumb, if you are having to consider whether or not to make the function more widely available then you should do it. It takes much less effort to do this work now, while it's fresh in your mind, than to have to re-familiarise yourself with the code in several years time.

Let's now look at how to implement those sub-bullet points: "when you write a function, document it" and "when you write a function, test it".

3.8.2 Rememer how to use your own code

When you come to use a function written by somebody else, you likely have to refer to their documentation to teach or to remind yourself of things like what the expected inputs are and how exactly the method is implemented.

When writing your own functions you should create documentation that fills the same need. Even if the function is just for personal use, over time you'll forget exactly how it works.

> *When you write a function, document it.*

But what should that documentation contain?

- Inputs
- Outputs
- Example use cases
- Author (if not obvious or working in a team)

Your documentation should describe the inputs and outputs of your function, some simple example uses. If you are working in a large team, the documentation should also indicate who wrote the function and who's responsible for maintaining it over time.

3.8.3 `{roxygen2}` for documentation

In the same way that we used the `{here}` package to simplify our file path problems, we'll use the `{roxygen2}` package to simplify our testing workflow.

The {roxygen2} package gives us an easily insert-able temple for documenting our functions. This means we don't have to waste our time and energy typing out and remembering boilerplate code. It also puts our documentation in a format that allows us to get hints and

auto-completion for our own functions, just like the functions we use from packages that are written by other people.

To use Roxygen, you only need to install it once - it doesn't need to be loaded with a library call at the top of your script. After you've done this, and with your cursor inside a function definition, you can then insert skeleton code to document that function in one of two ways: you can either use the Rstudio menu or the keyboard short cut for your operating system.

1. `install.packages("roxygen2")`
2. With cursor inside function: Code > Insert Roxygen Skeleton
3. Keyboard shortcut: *cmd + option + shift + r* or *crtl + option + shift + r*
4. Fill out relevant fields

3.8.4 An `{roxygen2}` example

Below, we've got an example of an Roxygen skeleton to document a function that calculates the geometric mean of a vector. Here, the hash followed by an apostrophe is a special type of comment. It indicates that this is function documentation rather than just a regular comment.

```
#' Title
#'
#' @param x
#' @param remove_NA
#'
#' @return
#' @export
#'
#' @examples
geometric_mean <- function(x, remove_NA = FALSE){
  # Function body goes here
}
```

We'll fill in all of the fields in this skeleton apart from export, which we'll remove. If we put this function in a R package, then the export field makes it available to users of that package. Since this is just a standalone function we won't need the export field, though keeping it wouldn't actually cause us any problems either.

```
#' Calculate the geometric mean of a numeric vector
#'
#' @param x numeric vector
#' @param remove_NA logical scalar, indicating whether NA values should be stripped before
```

```
#'
#' @return the geometric mean of the values in `x`, a numeric scalar value.
#'
#' @examples
#' geometric_mean(x = 1:10)
#' geometric_mean(x = c(1:10, NA), remove_NA = TRUE)
#'
geometric_mean <- function(x, remove_NA = FALSE){
  # Function body goes here
}
```

Once we have filled in the skeleton documentation it might look something like this. We have described what the function does, what the expected inputs are and what the user can expect as an output. We've also given an few simple examples of how the function can be used.

For more on Roxygen, see the package documentation or the chapter of R packages on function documentation.

3.8.5 Checking Your Code

> *If you write a function, test it.*

Testing code has two main purposes:

- To warn or prevent user misuse (e.g. strange inputs),
- To catch edge cases.

On top of explaining how our functions *should* work, we really ought to check that they *do* work. This is the job of unit testing.

Whenever you write a function you should test that it works as you intended it to. Additionally, you should test that your function is robust to being misused by the user. Depending on the context, this might be accidental or malicious misuse. Finally, you should check that the function behaves properly for strange, but still valid, inputs. These are known as edge cases.

Testing can be a bit of a brutal process, you've just created a beautiful function and now you're job is to do your best to break it!

3.8.6 An Informal Testing Workflow

1. Write a function
2. Experiment with the function in the console, try to break it
3. Fix the break and repeat.

Problems: Time consuming and not reproducible.

An informal approach to testing your code might be to first write a function and then play around with it in the console to check that it behaves well when you give it obvious inputs, edge cases and deliberately wrong inputs. Each time you manage to break the function, you edit it to fix the problem and then start the process all over again.

This *is* testing the code, but only informally. There's no record of how you have tried to break your code already. The problem with this approach is that when you return to this code to add a new feature, you'll probably have forgotten at least one of the informal tests you ran the first time around. This goes against our efforts towards reproducibility and automation. It also makes it very easy to break code that used to work just fine.

3.8.7 A Formal Testing Workflow

We can formalise this testing workflow by writing our tests in their own R script and saving them for future reference. Remember from the first lecture that these should be saved in the `tests/` directory, the structure of which should mirror that of the `src/` directory for your project. All of the tests for one function should live in a single file, which is named after that function.

One way of writing these tests is to use lots of if statements. The `{testthat}` can do some of that syntactic heavy lifting for us. It has lots of helpful functions to test that the output of your function is what you expect.

```
testthat::expect_equal(
  object = geometric_mean(x = c(1, NA), remove_NA = FALSE),
  expected = NA)

# Error: geometric_mean(x = c(1, NA), remove_NA = FALSE) not equal to NA.
# Types not compatible: double is not logical
```

In this example, we have an error because our function returns a logical `NA` rather than a double `NA`. Yes, R really does have different types of `NA` for different types of missing data, it usually just handles these nicely in the background for you.

This subtle difference is probably not something that you would have spotted on your own, until it caused you trouble much further down the line. This rigorous approach is one of the benefits of of using the `{testthat}` functions.

To fix this test we change out expected output to `NA_real_`.

We'll revisit the `{testthat}` package in the live session this week, when we will learn how to use it to test functions within our own packages.

3.9 Summary

- Functional and Object Oriented Programming
- Structuring your scripts
- Styling your code
- Reduce, reuse, recycle
- Documenting and testing

Let's wrap up by summarising what we have learned in this chapter.

We started out with a discussion on the differences between functional and object oriented programming. While R is capable of both, data science work tends to have more of a functional flavour to it.

We've then described how to structure your scripts and style your code to make it as human-friendly and easy to debug as possible.

Finally, we discussed how to write DRY code that is well documented and tested.

Workflows Checklist

> **Note**
>
> Effective Data Science is still a work-in-progress. This chapter is largely complete and just needs final proof reading.
> If you would like to contribute to the development of EDS, you may do so at https://github.com/zakvarty/data_science_notes.

Videos / Chapters

- ☐ Organising your work (30 min) [slides]
- ☐ Naming Files (20 min) [slides]
- ☐ Organising your code (27 min) [slides]

Reading

Use the workflows section of the reading list to support and guide your exploration of this week's materials. Note that these texts are divided into core reading, reference materials and materials of interest.

Tasks

Core:

- Find 3 data science projects on Github and explore how they organise their work. Write a post on the EdStem forum that links to all three, and in a couple of paragraphs describe the content and structure of one project.
- Create your own project directory (or directories) for this course and its assignments.
- Write two of your own R functions. The first should calculate the geometric mean of a numeric vector. The second should calculate the rolling arithmetic mean of a numeric vector.

Bonus:

- Re-factor an old project to match the project organisation and coding guides for this course. This might be a small research project, class notes or a collection of homework assignments. Use an R-based project if possible. If you only have python projects, then either translate these to R or apply the PEP8 style guide. Take care to select a suitably sized project so that this is a meaningful exercise but does not take more than a few hours.
- If you are able to do so, host your re-factored project publicly and share it with the rest of the class on the EdStem Discussion forum.

Live Session

In the live session we will begin with a discussion of this week's tasks. We will then create a minimal R package to organise and test the functions you have written.

Please come to the live session prepared to discuss the following points:

- Did you make the assignment projects as subdirectories or as their stand alone projects? Why?
- What were some terms that you had not met before during the readings? How did you find their meanings?
- What did you have to consider when writing your rolling mean function?

Part II

Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.