DevOps Carpentry

A Researcher's Guide to the Open Source Ecosystem

Matthias Bannert

7/12/2022

Table of contents

Preface

Note: This book will be published by Chapman & Hall/CRC. The online version of this book is free to read here (thanks to Chapman & Hall/CRC), and licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you have any feedback, please feel free to file an issue on GitHub. Thank you!

> "It is much easier to put existing resources to better use, than to develop resources where they do not exist." – George Soros

The vast majority of data has been created within the last decade. In turn many fields of research are confronted with an unprecented wealth of data. The sheer amount of information but also the complexity of modern datasets continues to point a kind researcher to programming approaches who had not considered programming to process data so far. *DevOps Carpentry* aims at two things: First, to give a big picture overview and starting point to reach what the open source software community calls a 'software carpentry' level. Second, the book gives an understanding of the opportunities of automation and reproducibility as well as the effort to maintain the required environment. This book argues a solid software carpentry skill level and self-operation is totally in reach for most researchers. And most importantly, investing is worth the effort: being able to code leverages field specific expertise and fosters interdisciplinary collaboration as source code continues to become an important communication channel.

Figure 1: Meet Dr. Egghead who started his quest to figure out how his assistant got a week's work done in two hours. "Hacker's wear hoodies, you know," he mumbles as he pulls up his coat's hood and starts to think…

1 Introduction

1.1 Why Work Like a Software Engineer?

First of all, because everybody and their grandmothers seem to do it. Statistical computing continues to be on the rise in many branches of research.

Source code can be a tremendously sharp, unambiguous and international communication channel.

Second because it's reproducible. Code has become a tremendous communication channel. Your web scraper does not work? Instead of reaching out in a clumsy but wordy cry for help, posting what you tried so far described by source code will often get you good answers within hours on platforms like Stackoverflow or Crossvalidated. Or think of feature requests: After a little code ping pong with the package author your wish eventually becomes clearer. Let alone chats with colleagues and co-authors. Sharing code just works. Academic journals have found that out, too in the meantime. Many outlets require you to make the data and source code behind your work available. Social Science Data Editors is a bleeding edge project at the time of writing this, but is already referred to by top notch journals like American Economic Review (AER).

Third, because it scales and automates. Automation is not only convenient. Like when you want to download data, process and create the same visualization and put it on your website any given Sunday. Automation is inevitable. Like when you have to gather daily updates from different outlets or work through thousands of .pdfs.

Last but not least because of things you couldn't do w/o being an absolute guru (if at all) if wasn't for programming. Take visualization. Go, check these D3 Examples. Now, try to do that in Excel. If you do these things in Excel it'd make you an absolute spreadsheet visualization Jedi, probably missing out on other time consuming skills to master. Moral of the story is, with decent, carpentry level programming skills – that'd be the upfront investment – you can already do so many spectular things while not really specializing and staying very flexible.

1.2 How to Read this Book?

The focal goal of this book is to map out the open source ecosystem, identify neuralgic components and give you an idea of how to improve not only in programming but also in navigating the wonderful but vast open source world. Chapter 2 is the roadmap for this book: it describes and classifies the different parts of the open source stack and explains how these pieces relate to each other. Subsequent chapters highlight core elements of the open source toolbox one at a time and walk through applied examples mostly written in the R language.

This book is a companion. The companion I wish I had when I started an empirical, data intensive PhD in economics. Yet, the book is written years after said PhD was completed and with the hindsight of 10+ years in academia. *Programming with Data* is written based on the experience of helping students and seasoned researchers of different fields with their data management, processing and communication of results.

If you are confident in your ambition to amp up your programming to at least solid software carpentry level within the next few months, I suggest to get an idea of your starting point relative to this book.The upcoming *backlog* section is essentially a list of suggested to-dos on your way to solid software carpentry. Obviously, you may have cleared a few tasks of this backlog item before reading this book which is fine. On the other hand you

might not know about some of the things that are listed in the *requirement* section which is fine, too. The backlog and requirements section just mean to give you some orientation.

If you do not feel committed, revisit the previous section, discuss the need for programming with peers from your domain and possibly talk to a seasoned R or Python programmer. Motivation and commitment are key to the endurance needed to develop programming into a skill that truly leverages your domain specific expertise.

1.3 Why Work Like an Operations Engineer ?

Why even think about Operations? In recent years the devOps approach has taken software engineering by storm. The combination of *dev*elopment and *op*eration*s* promises greater agility and better quality compared to a more traditional, strict distinction between IT system administration and product minded development.

Automation is key to the devOps approach and the main reason why devOps thinking is also very well suited for academic researchers and business analysts who use programming for their analyses. So called *continuous integration* is suitable to enforce a battery of quality checks such as unit tests or installation checks. Let's say a push to certain branch of a git repository leads to the checks described above. In a typical workflow, succesfull completion of quality checks triggers continuous deployment to a blog, rendering into a paper or interactive data visualization.

By embracing a devOps approach researchers do not only gain extra efficiency, but more importantly improve reproducibility and therefore accountability and quality of their work. Similar to the well established term *software carpentry* that advocates a solid, application minded understanding of programming with data, I suggest a *devOps carpentry* level understanding of development and operations is desirable for the programming data analyst.

The effect of a devOps approach on quality control is not limited to reproducible research in a publication sense, but also enforces rules during collaboration: no matter who contributes, contribution gets gut checked and only deployed if checks passed. Simply put, the devOps carpentry is the art of making our work deployable to other machines.

1.4 Backlog

Other than the classification and overview that *Programming with Data* provides, its introduction to **git version control** and the **collaboration workflow associated with git** are likely the most impactful items on your backlog. If you are not familiar with commits, pulls, pushes, branches, forks and pull requests, *Programming with Data* will open up a new world for you. by introducing you to industry standard collaboration. A solid *git* foundation makes you fit into plethora of (software) teams around the globe – in academia in beyond.

Your backlog en route to a researcher who is comfortable programming with data obviously contains a **strategy to improve your programming** itself and is complemented by a solid understanding of the **challenges of data management** from persistent storage to access restrictions. Plus, modern data driven science often has to handles datasets so large, **infrastructure** other than local desktops come into play. Yet, high performance computing (HPC) is by far not the only reason why its handy for a researcher to have a basic understanding of **infrastructure**. **Communication** of results including data dissemination or interactive online reports require the content to be served from a server with permanent online access. Basic workflow **automation** of regular procedures, e.g., for a repeated extract-transform-load (ETL) process to update data, is a low hanging (and very useful) fruit for a programming researcher.

The case studies at the end of the book are not exactly a backlog item like the above but still a recommended read. The case studies in this

book are hands-on programming examples – mostly written in R – to showcase tasks from API usage to geospatial visualization in reproducible fashion. Reading and actually running other developer's code does not only improve one's own code, but helps to see what makes code inclusive and what hampers comprehensibility.

1.5 Requirements

Essentially, the requirements in this section just extend the list of backlog items suggested above with a few things not covered in this book. This does not mean that every single item is strictly required as a prerequisite to make sense of this *Programming with Data*. The requirments are more to say that

- initial steps with a scripting language such as R or Python
- console / terminal basics (moving around the filesystem, ssh)

are explained more comprehensively elsewhere: (Big Book of R, R for Data Science, The Carpentries)

like backlog, but rather before you should go on this journey.

What You Need to Know to Make the Most of This Book

Though the book offers many entry points and strives to make advanced considerations accessible, *Programming with Data* provides the most value for readers whose prior knowledge is beyond certain threshold. The book is **not** an introduction to R nor Python. It's **not** an introduction to data science or machine learning either. Neither knowledge is strictly required to follow the book, but a basic understanding of statistical methods and the challenges applied statisticians face certainly helps to motivate getting invested into programming. Likewise, prior knowledge of a *scripting language* like *R, Python* or *Javascript*, *git version control*, as well as familiarity with *console / terminal basics* will help the reader sail smoothly.

Still though, to self-reflect on one's starting point remains the most important requirement to leverage the book.

Programming with Data willingly accepts to be overwhelming at times. Given the variety of topics touched in an effort to show the big picture, I encourage the reader to remain relaxed about a few blanks even when it comes to fundamentals. The open source community offers plenty of great resources to selectively upgrade skills and this books intends to show how to evaluate the need for help and how to find the right sources.

Programming with Data was written with the idea in mind that well maintained open source projects are the best to document themselves. Therefore I rather reference to existing documentation as opposed to rewriting it. Because even if my wording was better than the original, it is only a matter of time before my documentation is outdated. The bigger the stack the harder to keep track. So whenever, you stumble upon something you do not understand, make sure to look it up. (I tried hard to provide good a glossary and reference wherever possible, but if not - still push yourself to look things up).

That being said, to have an idea about following technologies and ideas will certainly help to get the most out of spending time with *Programming with Data*:

- Basic R or Python
- basic understanding of git version control
- basic understanding of GitHub based workflows, e.g., issue tracker, kanban boards…
- Terminal / Console and Shell and a shell, e.g., bash, fish

In other words: if none of the above means something to you, I recommend to make yourself comfortable with the basics of 2-3 of the above fields before you start to read this book or just cherry pick single topics.

1.6 Backlog

What you need to know to become comfortable programming with data...

2 Stack - A Developer's Toolkit

The goal of this chapter (and probably the most important goal of this entire book) is to help you with the big picture of which tool does what. The following sections will group common programming-with-data components by purpose. All of the resulting groups represent aspects of programming with data and I will move on to discuss these groups in a dedicated chapter each.

Just like natural craftsmen, digital carpenters depend on their toolbox and their mastery of it. A project's *stack* is what developers call the choice of tools used in a project. Even though different flavors come down to personal preferences, there is a lot of common ground in *programming with data* stacks. Below are some of the components I use most often. Of course this is a personal choice. Obviously, I do not use *all* of these components in every single small project. *Git, R* and *R Studio* would be a good minimal version.

Component	Choice
Interpreter / Language	R, Python, Javascript
IDE / Editor	R Studio,VS Code, Sublime
Version Control	Git
Project Management	GitHub, GitLab
Database	PostgreSQL
'Virtual' Environments	Docker
Communication (Visualization, Web)	Node, Quasar (vue.js)
Website Hosting	Netlify, GitHub Pages
Workflow Automation	Apache Airflow

Component	Choice
Continous Integration	GitLab CI

Throughout this book, often a choice for one piece of software needs to be made in order to illustrate things. To get the most out of the book, keep in mind that these choices are examples and try to focus on the role of an item in the big picture.

2.1 Programming Language

In Statistical Computing the interface between the researcher and the computation node is almost always an interpreted progamming language as opposed to a compiled one. Compiled languages like C++ require the developer to write source code and compile, i.e., translate source code into what a machine can work with *before* runtime. The result of the compilation process is a binary which is specific to an operating system. Hence you will need one version for Windows, one for OSX and one for Linux if you intend to reach a truly broad audience with your program. The main advantage of a compiled language is speed in terms of computing performance because the translation into machine language does not happen during runtime. A reduction of development speed and increase in required developer skills are the downside of using compiled languages.

> Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it. – Dan Ariely, Professor of Psychology and Behavioral Economics, on twitter

The above quote became famous in the hacking data community, not only because of the provocative, fun part of it, but also because of the implicit advice behind it. Given the enormous gain in computing power in recent

Figure 2.1: Aaaaaaah! Don't panic, Dr. Egghead! All these components are here to help you and you won't need all of them from the start.

decades, but also methodological advances, interpreted languages are often fast enough for many social science problems. And even if it turns out, your data grow out of your setup, a well written proof of concept written in an interpreted language can be a formidable blueprint. **Source code is turning into an important scientific communication channel.** Put your money on it, your interdisciplinary collaborator from the High Performance Computing (HPC) group, will prefer some Python code as a briefing for their C++ or FORTRAN program over a wordy description out of your field's ivory tower.

Interpreted languages are a bit like pocket calculators, you can look at intermediate results, line by line. R and Python are the most popular OSS choices in hacking with data, Julia is an up and coming, perfomance focused language with a much slimmer ecosystem. A bit of Javascript can't hurt for advanced customization of graphics and online communication of your results.

2.2 Interaction Environment

While the fact that a software developer needs to choose a programming language to work with is rather obvious, the need to compose and customize an environment to interact with the computer is much less obvious to many. Understandably so, because outside of programming, software such as word processors, video or image editors presents itself to the user as a single program. That single program
takes the user's input, processes the input (in memory) and stores a results on disk for persistence – often in a proprietary, program specific format.

Yet, despite all the understanding for nonchalantly saying we keep documents **in** Word or data **in** R Studio, it's beyond nerdy nitpicking when I insist that data are kept in files (or databases) – NOT in a program. And that R is NOT R Studio: This sloppy terminology is contributes to making us implicitly accept that our office documents live in one single

program. (And that there is only one way to find and edit theses files: through said program).

It is important to understand that source code of essentially any programming languages is a just a plain text file and therefore can be edited in any editor. Editors come in all shades of gray: from lean, minimal code highlighting support to full fledged integrated development environments (IDE) with everything from version control integration to sophisticated debugging.

Which way of interaction the developer likes better is partly a matter of personal taste, but it also depends on the programming language, the team's form of collaboration and size of the project. In addition a customized editor, most developers use some form of a console to communicate with their operating system. The ability to send commands instead of clicking is not only reproducible and shareable it also outpaces mouse movement by lot when commands come in batches. Admittedly configuration, practice and getting up to speed takes its time but once you have properly customized the way you interact with your computer when writing code you will never look back.

In any case, make sure to customize your development environment: choose the themes you like, make sure the cockpit you spent your day in is configured properly and feels comfy.

2.3 Version Control

To buy into the importance of managing one's code professionally may be the single most important take away from this book. Being able to work with version control will help you fit into a lot of different teams that have contact points with data science and programming, let alone if you become part of a programming or data science team.

While version control has a long history dating back to CVS and SVN, the good news for the learner is, that there is a single dominant approach when it comes to version control in the data analysis world. Despite the fact that its predecessors and alternatives such as mercurial are stil around, git is the one you have to learn. To learn more about the history of version control and approaches other than git, Eric Sink's Version Control by Example is for you.

So what does git do for us as researchers? How is it different from dropbox?

```
git does not work like dropbox. git does not work like dropbox.
git does not work like dropbox. git does not work like dropbox.
git does not work like dropbox. git does not work like dropbox.
git does not work like dropbox. git does not work like dropbox.
git does not work like dropbox. git does not work like dropbox.
git does not work like dropbox. git does not work like dropbox.
```

The idea of thinking of a sync, is what interferes with comprehension of the benefit of version control (which why I hate that git GUIs call it 'sync' anyway to avoid irritation of user's initial believes.). Git is a decentralized version control system that keeps track of a history of semantic commits. Those commits may consist of changes to multiple files. A commit message summarizes the gist of a contribution. *Diffs* allow to compare different versions.

The *diff* output shows an edit during the writing of this book. The line preceeded by '-' was replaced with the line preceeded by '+'.

Git is well suited for any kind of text file. May it be source code from Python or C++, or some text written in markdown or LaTeX. Binaries like .pdfs or Word documents are possible, too, but certainly not the type of file for which git is really useful. This book contains a detailed, applied introduction tailed to researchers as part of the Programmers' Practices

and Workflows chapter, so let's dwell with the above contextualization for a bit.

2.4 Data Management

One implication of bigger datasets and/or bigger teams is the need to manage data. Particularly when projects establish themselves or update their data on a regularly basis, well defined processes, consistent storage and possibly access management become relevant. But even if you worked alone, it can be very helpful to have an idea about data storage in files systems, memory consumption and archiving data for good.

Data come in various forms and dimensions, potentially deeply nested. Yet researchers and analysts are mostly trained to work with one-row-one-observation formats in which columns typically represent different variables. In other words two dimensional representation of data remains the most common and intuitive form of data to many. Hence office software offers different, text based and proprietary spreadsheet file formats. On disk, comma separated files (.csv)[1] are probably the purest representation of two dimensional data that can processed comfortably by any programming language and be read by many programs. Other formats such as .xml or .json allow to store even deeply nested data.

In-memory, that is when working interactively with a programming languages such as R or Python, data.frames are the most common represenation of two dimensional data. Data.frames, their progressive relatives like tibbles or data.tables and the ability to manipulate them in reproducible, shareable and discussible fashion is is the first superpower upgrade over pushing spreadsheets. Plus, other representation such as arrays, dictionaries or lists represent nested data in memory. Though in memory data

[1] Some dialects use different separators like ';' or tabs, partly because of regional differences like the use of commas as decimal delimiters.

manipulation is very appealing, memory is limited and needs to be managed. Making the most of the memory available is one of the driving forces behind extensions of the original data.frame concept.

The other obvious limitation of data in memory is the lack of persistent storage. Therefore in-memory data processing needs to be combined with with file based storage are a database. The good news is, languages like R and Python are well equipped to interface with a plethora of file based approaches as well as databases. So well, that I often recommend these languages to researchers who work with other less equipped tools, solely as an intermediate layer.

To evaluate which relational (essentially SQL) or non-relational database to pick up just seems like the next daunting task of stack choice. Luckily, in research first encounters with a database are usually passive, in the sense that you want to query data from a source. In other words the choice has been made for you. So unless you want to start your own data collection from scratch, simply sit back, relax and let the internet battle out another conceptual war. For now, let's just be aware of the fact that a data managemnt game plan should be part of any analysis project.

2.5 Infrastructure

For researchers and business analysts who want to program with data the starting infrastructure is very likely their own desktop or notebook computer. Nowadays this already means access to remarkable computing power suited for many tasks.

Yet, it is good to be aware of the many reasons that can turn the infrastructure choice from a no-brainer into a question with many options and consequences. Computing power, repeated or scheduled tasks, hard-to-configure or non-compatible runtime environment, online publication or high availability may be some of the needs that make you think about infrastructure beyond your own notebook.

Today, thanks to software-as-a-service (SaaS) offers and cloud computing, none of the above needs imply running a server let alone a computing center on your own. Computing has not only become more powerful over the last decade but also more accessible. Entry hurdles are lower than ever. Many needs are covered by services that do not require serious upfront investment. It has become convenient to try and let the infrastructure grow with the project.

From a researchers' and analysts' perspective one of the most noteworthy infrastructure developments of recent may be the arrival of *containers* in data analysis. Containers are isolated, single purpose environments that run either on a single container host or with an orchestrator in a cluster. Albeit technically different from virtual machines we can regard containers as virtual computers running on a computer for now. With containers data analysts can create isolated, single purpose environments that allow to reproduce analysis even after one's system was upgraded and with it all the libraries used to run the analysis. Or think of some exotic LaTeX configuration, Python environment or database drivers that just can't manage to run on your local machine. Bet there is container blueprint (aka image) around online for exactly this purpose.

> "Premature optimization is the root of all evil." – Donald Knuth.

On the flipside, product offerings and therefore our options have become a fast growing, evolving digital jungle. So rather than trying to keep track (and eventually loosing it) of the very latest gadgets, this book intends to show a big picture overview and pass on a strategy to evaluate the need to add to your infrastructure. Computing power, availability, specific services and collaboration are the main drivers for researcher to look beyond their own hardware. Plus, public exposure, i.e., running a website as discussed in the publishing section asks for a web host beyond our local notebooks.

2.6 Automation

"Make repetitive work fun again!" could be the claim of this section. Yet, it's not just the typical intern's or student assistant's job that we would love to automate. Regular ingestion of data from an API or a web scraping process are one form of reocurring tasks, often called cron jobs after the Linux cron command which is often used to schedule execution of any Linux command. Regular computation of an index, triggered by incoming data would be a less time focused, but more event triggered example of an automated task.

The one trending form of automation though, is *continuous integration / continuous development (CI/CD)*. *CI/CD* processes are typically closely linked to a git version control repository and are triggered by a particular action done to the repository. For example, in case some pre-defined part (branch) of a git repository gets updated, an event is triggered and some process starts to run on some machine (usually some container). Builds of software packages are very common use cases of such a *CI/CD* process. Imagine your developing an R package and want to run all the tests you've written, create the documentation and test whether the package can be installed. Once you've made your changes and push to your remote git repository. Your push triggers the tests, the check, the documentation rendering and the installation. Potentially a push to the main branch of your repository could even deploy a package that cleared all of the above to your production server.

Rendering documentation, e.g., from markdown to HTML into a website or presentation or a book like this one is a very similar example of CI/CD process. Major git providers like Gitlab (Gitlab CI/CD) or GiTHub (GitHub Actions) offer CI/CD tool integration. In addition, standalone services like circle-ci can be used as well open source, self-hosted software like drone.

2.7 Communication Tools

Its community is one of the reasons for the rise of open source software over the last decades. Particularly newcomers would miss a great chance for a kickstart into programming if they did not connect with the community. Admittedly, not all of the community's communication habits are for everyone, yet it is important to make your choices and pick up a few channels.

Chat clients like Slack, Discord or the Internet Relay Chat (IRC) (for the pre-millenials among readers) are the most direct form of asynchronous communication. Though many of the modern alternative are not open-source themselves, they offer free options and remain popular in the community despite self-hosted approaches such as matrix along with element. Many international groups around data science and statistical computing such as RLadies or the Research Software Enginnering society have Slack space that are busy 24/7 around the world.

Social media is less directed and more easily consumed in passive fashion than chat space. Twitter is a very active resource to find and exchange good reads or online tutorials. LinkedIn is another good option to connect and find input, particularly in parts of the world where Twitter is less popular. Due to its activity social media is also a great way to stay up-to-date with the latest development.

Mailing lists are another more old fashioned form to discuss things. They do not require a special client but just an email address to subscribe to their regular updates. If you intend to avoid social media as well as signing up at knowledge sharing platforms such as stackoverflow.com or reddit mailing lists are a good way to get help from experienced developers.

Issue trackers are one form of communication that is often underestimated by newcomers. Remotely hosted git repositories, e.g., repositories hosted at GitHub or Gitlab, typically come with an integrated issue trackers to report bugs. The community discusses a lot more than problems on issue

trackers: feature requests are common use case and even the future direction of an open source project may be affected substantially by discussions in its issue tracker.

2.8 Publishing and Reporting

Data analysis hopefully yields results that the analyst intends to report internally within their organisation or share with the broader public. This has lead to a plethora of options on the reporting end. Actually data visualization and reproducible, automated reporting is one of the main drivers researchers and analysts turn to programming for their data analysis.

In general we can distinguish between two forms of output: pdf-like, print oriented output and HTML-centered web content. Recent toolchains have enabled analysts without strong backgrounds in web frontend development (HTML/CSS/Javascript) to create nifty reports and impressive, interactive visualizations. Frameworks like R Shiny or Python Dash even allow to create complex interactive websites with data science environment.

Notebooks that came out of the Python world established themselves in many other languages implementing the idea of combining text with code chunks that get executed when the text is rendered to HTML or PDF. This allows researchers to describe, annotate and discuss results while sharing the code that produced the results described. In academia, progressive scholars and journals embrace this form of creating manuscripts as *reproducible research* that improves trust in the presented findings.

Besides academic journals that start to require researchers to hand in their results in reproducible fashion, reporting based on so called static website generators[2] has taken data blogs and reporting outlet including

[2] As opposed to content management systems (CMS) that keep content in a database and put content and layout template together when users visit a website, static website generators render a website once triggered by an event. If users want to update a static website, they simply re-run the render process and push the HTML

presentations by storm. Platforms like GitHub render markdown files to HTML automatically, displaying formatted plain text as a decent website. Services such as netlify allow to use a broad variety of build tools to render input that contains text and code.

Centered around web browsers to display the output, HTML reporting allows to create simple reports, blogs, entire books (like this one) or presentation slides for talks. But thanks to documented converters like pandoc and a typesetting juggernaut called LaTeX, rendering sophisticated .pdf is possible, too. Some environments even allow to render to propietary word processesor formats.

output of said process online.

3 Programming 101

Obviously, the craft of *programming* is essential to handling data with a programming language. Though programming languages can differ quite a bit from each other, there is common ground and an approach to programming that is crucial to understand and important to learn by – particularly for programmers without a formal computer science background. This chapter wants to point researchers, analysts and data scientists to the low hanging, high impact fruits of software engineering.

Programming is a form of communication. Communication with others but also with your future self. It actually may be the best way to define complex contexts in reproducible fashion. Therefore source code needs to be written inclusive fashion. Programming needs to be seen as a chance to make complexity accessible to those who are experts in a particular domain. The fact that programs actually run makes them more accessible to many than formal mathematical definitions.

The programming examples in this book mostly stick to the [R] language which is easy to install and run on one's own local computer. All of the concepts shown can easily transfer to other languages. Though potentially a bit more tricky to install and run across operating systems, Python would have a been an equally good choice, but at there a had to be a decision for one language for this book…

3.1 The Choice that Doesn't Matter

The very first (and intimidating) choice a novice hacker faces is which is programming language to learn. Unfortunately the medium popularily summed up as the internet offers a lot of really really good advice on the matter. The problem is, however, that this advice does not necessarily agree which language is the best for research. In the realm of data science – get accustomed to that label if you are a scientist who works with data – the debate basically comes down to two languages: The R Language for Statistical Computing and Python.

At least to me, there is only one valid advice: **It simply does NOT matter**. If you stick around in data science long enough you will eventually get in touch with both languages and in turn learn both. There is a huge overlap of what you can do either of those languages. R came out of the rather specific domain of statiscs 25+ years ago and made its way to a more general programming language thanks to 15K+ extension packages (and counting). Built by a mathmatician, Python continues to be as general purpose as it's ever been. But it got more scientific, thanks to extension packages of its own such as pandas, SciPy or numPy. As a result there is a huge overlap of what both languages can do and both will extend your horizon in unprecendented fashion if you did not use a full fledged programming language for your analysis before.

But why is there such a heartfelt debate online, if it doesn't matter? Let's pick up a random argument from this debate: R is easier to set up and Python is better for machine learning. If you worked with Java or another environment that's rather tricky to get going, you are hardened and might not cherish easy onboarding. If you got frustrated before you really started, you might feel otherwise. You may just have been unlucky making guesses about a not so well documented paragraph, trying to reproduce a nifty machine learning blog post. Just because you installed the wrong version of Python or didn't manage to make sense of virtualenv right from the beginning.

Figure 3.1: R: "Dplyr smokes pandas." Python: "But Keras is better for ML!" Language wars can be entertaining, sometimes spectacular, but most times they are just useless…

The point is, rest assured, if you just start doing analytics using a programming languages both languages are guaranteed to carry you a long way. There is no way to tell for sure which one will be the more dominant language in 10 years from now or whether both still be around holding their ground the way they do now. But once you reached a decent software carpentry level in either language, it will help you a lot learning the other. If your peers work with R, start with R, if your close community works with Python, start with Python. If you are in for the longer run either language will help you understand the concepts and ideas of programming with data. Trust me, there will be a natural opportunity to get to know the other.

If you associate programming more often than not with hours of fiddling, tweaking and fighting to galvanize approaches found online, this chapter is for you. Don't expect lots of syntax. If you came for examples of useful little programs from data visualization to parallel computing, check out the case studies.

The following sections share a blueprint to go from explorative script to production ready package. Organise your code and accompany the evolution of your project: start out with experiments, define your interface, narrow down to a proof of concept and scale up. Hopefully the tips, tricks and the guidance in this chapter will help you to experience the rewarding feeling of a software project coming together like a plan originated by Hannibal Smith.

3.2 Plan Your Program

How much planning ahead is optimal for your project ultimately depends on your experience, number of collaborators and size of your project. But still, a rough standard checklist helps any project.

3.2.1 Think library!

The approach that I find practical for applied, empirical research projects involving code is: think library. Think package. Think *reusable* code. Don't think you can't do it. Let me de-mystify packages for you: **Packages are nothing else than source code organized in folders following some convention**. Thanks to modern IDEs, it has never been easier to stay inline with conventions. Editors like R Studio ship with built-in support to create package skeletons with a few clicks. Thousands of open source extension packages allow you to learn from their structure. Tutorials like Packing Python Projects or Hadley Wickham's free online book R Packages explain how to create packages good enough to make the official PyPi or CRAN package repository.

In other words, it is unlikely that someone with moderate experience comes up the best folder structure ever invented. Sure, every project is different and not every aspect (folder) is needed in every project. Nevertheless, there are well established blueprints, guides and conventions that suit almost any project. Unlike Office type of projects which center around one single file, understand a research project will live in a folder with many subfolders and files. Not in one single file.

Trust me on this one: The package approach will pay off early. Long before you ever thought about publishing your package. Write your own function definition, rather than just calling functions line by line. Write code as if you need to make sure it runs on another computer. Write code as if you need to maintain it.

Go from scripts like this

```
# This is just data from the sake of
# reproducible example
set.seed(123)
d1 <- rnorm(1000)
```

Figure 3.2: “Over the days are when creating packages for gurus was only.”

```
d2 <- rnorm(1000)

# Here's where my starts
# let's create some custom descriptive
# stats summary for the data generated above
d1_mean <- mean(d1)
d1_sd <- sd(d1)
d1_q <- quantile(d1)
desc_stats_d1 <- list(d1_mean = d1_mean,
                      d1_sd = d1_sd,
                      d1_q = d1_q)

d2_mean <- mean(d2)
d2_sd <- sd(d2)
d2_q <- quantile(d2)
desc_stats_d2 <- list(d2_mean = d2_mean,
                      d2_sd = d2_sd,
                      d2_q = d2_q)
```

To function defintions and calls like that

```
# Imagine you had thousand of datasets.
# Imagine you wanted to add some other stats
# Imagine all the error prone c&p with
# the above solution.
# Think of how much easier this is to document.
# This is automation. Not cooking.
create_basic_desc <- function(distr){
  out <- list(
    mean = mean(distr),
    sd = sd(distr),
    quantiles = quantile(distr)
```

```
  )
  out
}

create_basic_desc(d1)
```

```
$mean
[1] 0.01612787

$sd
[1] 0.991695

$quantiles
          0%          25%          50%          75%         100%
-2.809774679 -0.628324243  0.009209639  0.664601867  3.241039935
```

```
create_basic_desc(d2)
```

```
$mean
[1] 0.04246525

$sd
[1] 1.009674

$quantiles
         0%         25%         50%         75%        100%
-3.04786089 -0.65322296  0.05485238  0.75345037  3.39037082
```

Start to document functions and their parameters using Roxygen syntax and you're already very close to creating your first package. **Pro-tipp**:

Hit Cmd+Alt+Shift+R[1] while inside a function definition with you cursor. When working with R Studio, it will create a nifty roxygen skeleton with all your function's parameters.

```
#' Create Basic Descriptive Statistics
#'
#' Creates means, standard deviations and default quantiles from an numeric input vector.
#'
#' @param distr numeric vector drawn from an arbitraty distribution.
#' @export
create_basic_desc <- function(distr){
  out <- list(
    mean = mean(distr),
    sd = sd(distr),
    quantiles = quantile(distr)
  )
  out
}
```

Writing *reusable code* will improve your ability to remember syntax and apply concepts to other problems. The more you do it, the easier and more natural becomes. Just like a toddler figuring out how to speak in a natural language. At first progress seems small, but once kids understand the bits and pieces of a language they start building at a remarkable speed, learn and never forget again.

3.2.2 Documentation

First things first. Write the first bit of documentation before your first line of code. Documentation **written** with hindsight will always be written

[1] On Windows / Linux use Ctrl instead of Cmd.

with an all-knowing, smartest-person-in-the-room mindset and the motivation of someone who already gave her best programming. Understand, I am not talking about the finetuning here, but about a written outline. Describe **how** parts of the code are going to do stuff. Also, examples can't hurt to illustrate what you meant. Research projects often take breaks and getting back to work after months should be as easy as possible.

Pseudo Code is a good way of writing up such an outline documentation. Take a simple API wrapper for example. Assume there is an API that returns numeric ids of hit entries when queried for keywords. These ids can be passed on to yet another endpoint, to obtain a profile. A rough game plan for an API Wrapper could like this:

```
# function: keyword_search(keyword, url = "https://some.default.url.com")
# returns numeric ids according to some api documentation


# function: query_profile(vec_in_ids)
# a json object that should be immediately turned into list by the functio
# returns list of properties
```

Documentation should use your ecosystem's favorite documentation framework. Yet, your comments within the code are the raw, initial form of documentation. Comments help to understand key parts of a program as well as caveats. Comments help tremendously during development time, when debugging or coming back to a project. Let alone when joining a project started by others.

While pseudo code where comments mimmick code itself is the exception to that rule, good comments should always follow the **not-what-but-why** principle. Usually, most high level programming languages are fairly easy to read and remind of rudimentary English. Therefore a *what* comment like this is considered rather useless:

```
# compute the cumulative sum of a vector
cumsum(c(T,F,F,F,F,T,F,F,T,F,F,F,T))
```

Whereas this *why* comment may actually be helpful:

```
# use the fact that TRUE is actually stored as 1
# to create a sequence until the next true
# this is useful for splitting the data later on.
cumsum(c(T,F,F,F,F,T,F,F,T,F,F,F,T))
```

Comment on why you do things, especially with which plan for future use in mind. Doing so will certainly foster exchange with others who enter or re-visit the code at a later stage (including yourself).

3.2.3 Design Your Interface

In other languages it is fairly common to define the data type of both: the input and the output[2]. Though doing so is not necessary in R, it is good practice to define the types of all parameters and results in your comments / documentation.

Once you know a bit more about your direction of travel, it's time to think about how to modularize your program. How do different parts of the program play together. users interact with your program: Will your code just act as a storage pit of tools, a loose collection of commands for adhoc use? Are others using the program, too? Will there be machine-to-machine interaction? Do you need graphical user interface (GUI) like shiny?

These questions will determine whether you use a strictly functional approach, a rudimentary form of object orientation like S3, a stricter implementation like R6 or something completely exotic. There a plenty of

[2]See statically typed language vs. dynamically typed language.

great resources out there, so I will not elaborate on this for the time being. The main message of this section is: Think about the main use case. Is it interactive? Is it a program that runs in batch typically? Do your users code? Would they prefer a GUI?

3.2.4 Dependencies

One important word of advice for novice package developers is to think about your dependencies. Do not take depedencies lightly. Of course it is intriguiging to stand on the shoulders of giants. Isn't R great because of its 15K+ extension packages? Isn't exactly this was made R such as popular language?

Yes, extension packages are cool. Yes, the ease with with CRAN packages are distributed is cool. But, just because packages are easy to install and free of license costs it does not mean leaning on a lot of packages comes at no costs: One needs to stay informed about updates, issues, breaking changes or undesired interdependencies between packages.

The problem is mitigated a bit when a) a package is required in an interactive script and b) one is working with a very popular package. Well managed packages with a lot of reverse dependencies tend to deprecate old functionality more smoothly as authors are aware of the issues breaking changes cause to a package's ecosystem.

In R, the tidyverse bundle of packages seems ubiquitous and easy to use. But it leads to quite a few dependencies. The data.table ecosystem might be less popular but provides its functionality with a single R package dependency (the {methods} package).

Often it does not take much to get rid of dependency:

```
library(stringr)
cow <- "A cow sounds off: mooooo"
```

```
str_extract(cow,"moo+")
```

```
[1] "mooooo"
```

Sure, the above code is more intuitive, but shrewd use of good ol' *gsub* and back referencing allows you to do the very same thing in base R.

```
gsub("(.+)(mooo+)","\\2",cow)
```

```
[1] "mooooo"
```

Again, {stringr} is certainly a well crafted package and it is definitely not the worst of all packages. But when you just loaded a package because it adds convenience to one single line or worse just because you found your solution online, think again before adding more dependencies to a production environment.

3.2.5 Folder Structure

In R, packages may have the following folders. Note that this does not mean a package has to contain all of these folders. FWIW, an R package needs to have NAMESPACE and DESCRIPTION files, but that is not the point here. Also there are more comprehensive, better books on the matter than this little section. The point of this section though is to discuss the role of folders and how they help you structure your work, even if you don't want to create an R package in first place.

This chapter describes the role of different folders in a package and what these folders are good for. More likely than not, this will cover a lot of the aspects of your project, too.

- R
- data
- docs
- vignettes
- src
- inst
- man

The below description explains the role of all of these folders.

R

A folder to store function definitions as opposed to function calls. Typically every function goes into a separate file. Sometimes it makes sense to group multiple functions into a single file when functions are closely related. Another reason for putting more than one function into a single file is when you have a collection of relatively simple, short helper functions. The R folder MUST NOT contain calls[3].

```
my_func_def <- function(param1, param2){
  # here goes the function body, i.e., what the function does
  a <- (param1 + param2) * param3
  # Note that in R, return statements are not necessary and even
  # relatively uncommon, R will return the last unassigned statement
  return(a)
}
```

man

This folder contains the context manual of your package. What you'll find here is the so called *function reference*, basically a function and dataset specific documentation. It's what you see when you run `?function_name`.

[3] Essentially, examples are calls, too. Note, I do recommend to add examples. Hadley Wickham's guide to document functions within packages shows how to add examples correctly.

The content of the `man/` folder is usually created automatically from the roxygen style documentation (note the #' styled comments) during a '`devtools::document()` run. Back in the days when people wore pijamas and lived life slow, the man folder was filled up manually with some LaTeX reminiscant .rd files, but ever since R Studio took over in 2012, most developers use Roxygen and render the function reference part of the documentation from their comments.

```
#' Sum of Parameters Multiplied by First Input
#'
#' This functions only exists as a show case.
#' It's useless but nevertheless exported to the NAMESPACE of this
#' package so users can see it and call the function by it's name.
#'
#' @param param1 numeric input
#' @param param2 numeric input
#' @export
my_func_def <- function(param1, param2){
  # here goes the function body, i.e., what the function does
  a <- (param1 + param2) * param1
  # Note that in R, return statements are not necessary and even
  # relatively uncommon, R will return the last unassigned statement
  return(a)
}
```

docs

This folder is typically not filled with content manually. When pushed to github a docs folder can easily be published as website using Github Pages. With Github pages you can host a decently styled modern website for free. Software projects often use GitHub Pages to market a product or project or simply for documentation purposes. All you need to do is check a couple of options inside the Github Web GUI and make sure the docs/ folder contains .md or .html files as well as stylesheets (.css). The latter

may sound a bit like Latin to people without a basic web development background, but there is plenty of help. The R ecosystem offers different flavors of the same idea: use a bit of markdown + R to generate website code. There is blogdown for your personal website or blog. There is pkgdown for your packages documentation. And there is even bookdown to write an online book like this. Write the markdown file, render it as HTML into the docs folder, push the docs folder to GitHub. Done. Your website will be online at username.github.io/reponame.

Here's a an example of a package down website: https://mbannert.github.io/timeseriesdb/

data

If you have file based data like .csv, .RData, .json or even .xlsx put them in here. Keeping data in a separate folder inside the project directory helps to keep reference to the data relative. There is nothing more greenhorn than `r read.csv("C:\mbannert\My Documents\some_data.csv")`. Even if you like this book, I doubt you have a folder named 'mbannert' on your computer. Ah, an in case you wondered, extensive use of `setwd()` is even worse. Keep your reference to data (and functions alike) relative. If you are sourcing data from a remote NAS drive as it is common at many universities, you can simply mount this drive to you folder (LTMGTFY: How to mount a network drive Windows / OSX).

vignettes

Admittedly not the most intuitive names for a folder that is supposed to contain articles. Vignettes are part of the documentation of a good package. It's kind of a description as if you were to write a paper about your package, including some examples of how to use it. For modern packages, vignettes are often part of their package down based online documentation. Feel free, to name this folder differently, though sticking to the convention will make it easier to turn your project into a project at a later stage. This folder typically contains Markdown or RMarkdown files.

src

The source folder is just here for the sake of completeness and is not needed in projects that only involve R source code. It's reserved for those parts of a package that need compilation, e.g., C++ or FORTRAN source code.

inst

When you install an R package using `install.packages()` it will be installed in some deep dungeon on your computer where R lives within your OS. The inst/ folder allows you to ship non R files with your installation. The files of the inst folder will just be copied into the package root folder inside your installation of that package.

inst is also a great place to store experimental function calls or playground files once the package ambitions become more concrete and those type of files do not live conveniently in the project root anymore. Also. I sometimes put shiny apps for local use into the inst/ folder if I want to make them part of a package.

3.3 Naming Conventions: Snake, Camel or Kebap

Before we start with files and folders, let me drop a quick, general note on naming. As in how to name files, folders and functions. It may look like a mere detail, but concise formatting and styling of your code will be appreciated by your peers and by those you ask for help. Plus, following an established convention will not make you look like a complete greenhorn.

- Do NOT use spaces in folder or file names! Never. If you need lengthy descriptions, use underscores '_', dashes'-' or camelCase.
- avoid umlauts and special characters. Encoding and internationalization is worth a book of its own. It's not like modern programming environments can't handle it, but encoding will introduce further complications. These are exactly the type of complications that may lead to an unplanned, frustrating waste of hours. You may be lucky

enough to find a quick fix, but you may as well not. Avoid encoding issues if do not plan to build a deeper understanding of encoding on the fly. This is especially true for cross platform collaboroations (Windows vs. Unix / OSX).

- either go for camelCase, snake_case or kebap-case. Otherwise prefer lower case characters. Also make sure to not switch styles within a project. There a plenty of style guides around, go with whatever your lab or community goes.

3.4 Testing

One of the things that helps scientists and business analysts reach the next level in programming is to develop an understanding of *testing* the way software engineers use the term. The colloquial understanding of testing basically comes down to do a couple of dry runs before using code in production. Looking at tests as a systematic and standardized procedure manifested in code substantially improves the quality and reliability of one's code.

When writing software for statistical analysis, testing mostly refers to unit tests. Unit tests are expectations expressed in code often using a testing framework to help define expectations. In R the *testthat* or *tinytest* R packages are examples for such frameworks.

```
# the function
cow <- function(){

  sound_of_a_cow <- "moo"
  sound_of_a_cow

}
```

```
# A test that uses the
# testthat package to formulate
# and check expectations
library(testthat)
test_that("The cow still mooos...", {
  expect_gte(nchar(cow()),3)
  expect_true(grepl("^m",cow()))
})
```

```
Test passed
```

The above dummy function simply returns a character. The accompanying test checks whether the "moo" of the cow is loud enough (=has at least 3 characters) and whether it starts with "m" (so it's not a "woof"). Note how tests typically come in bunches to thoroughly test functionality.

So why don't we write code correctly in the first place instead of writing code to check everything that could eventually go wrong? When developing *new* features we might be confident that newly introduced features do not break existing functionality. At least until we test it :) . Experience proves that seemingly unrelated edits do cause side effects. That is why well maintained packages have so called unit tests that are run when the package is re-build. If one of the checks fails, the developer can take a look before the a change that broke the code is actually deployed. To foster the development of these type of tests there are unit testing frameworks for many languages and frameworks. In R the most prominent testing packages are {tinytest} and {testthat}.

Hadley's book R packages has a more thorough introduction to testing in R with the {testthat} packages. Though the book is focused on R its introduction to formal testing is very illustrative for anyone interested to add testing to their development routine. Despite the fact that a comprehensive introduction to testing is out of the scope of this book, I would like to briefly introduce the concept of code cco

3.5 Debugging

In programming there is no way around debugging. From copy&paste artists to the grandmasters of hacking: writing code implies the need to debug. The main difference between amateur and professional is the degree to which the hunt for errors a.k.a. bugs is done systematically. This section gives a few tips to help organize a debugging strategy and assess how much honing of one's debugging approach is reasonable.

3.5.1 Read Code from the Inside Out

Many scripting languages allow some form of nesting code. In order to understand what's going on reading and running code from innermost element first helps. Even if you are not an R user, applying the inside-out idea helps to understand what's going on. Consider the following piece of R code:

```
identical(sum(cumsum(1:10)),sum(cumsum(rev(1:10))))
```

```
[1] FALSE
```

The re-formatted version below helps to identify the innermost part of the code at the first glimpse:

```
identical(
  sum(
      cumsum(1:10)
    ),
  sum(
    cumsum(
      rev(1:10)
      )
```

```
    )
  )
```

To understand the above demo code let's take closer look at the innermost element(s). Also consider looking at the single functiosn documentation, e.g., *?rev*:

```
1:10
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
rev(1:10)
```

```
 [1] 10  9  8  7  6  5  4  3  2  1
```

The calls to cumsum() and sum() are the next layers. Finally, identical() is the outermost function.

3.5.2 Debugger, Breakpoints, Traceback

Typically modern source code editors (see chapter IDEs) provide support to make your debugging approach. more systematic - use debug - set breakpoints (IDE feature)

3.6 A Word on Peer Programming

Peer programming, also called pair programming just means two developers sit in front of the same screen to collaborate on a piece of code. So why is there such a buzz about it? Why is there even a term for it? And why is there a section in an applied book on it?

That is because novice programmers (and their scientific supervisors) often doubt the efficiency of two paid persons working at the same work station. But programming is not about digging a hole with two shovels. Particularly not when it comes to building the software basis or frame of a project.

Working together using one single keyboard and screen or the virtual equivalent thereof can be highly efficient. The virtual equivalent, i.e., in fact using two computers but sharing the screen while in call, helps tremendously with a) your concept, b) your documentation. Plus, it is a code review at the same time. But most importantly both developers learn from each other. Having to explain and being challenged, deepens the understanding of experienced developers and ruthlessly identifies holes in one's knowledge. One important advice when peer programming is to switch the driver's seat from time to time. Make sure the lesser programmer holds the keys from time to time and maneuvers through articles, repositories, code and data. Doing so prevents the co-pilot from taking a back seat and letting the veteran do the entertainment. Visual Studio Code Live Share is a great feature for next level virtual peer programming as it allows for two drivers using two cursors.

Of course there are downsides of the pair programming approach, too. Also, timing within the lifecycle of a project is an important factor and not every project is the same fit for this agile method. But given there are so many approaches, I will leave the back and forth to others. The goal of this section is to point the reader to a practical approach that tends to work well in programming with data setups in social sciences. Googlers Jeff Dean and Sanjay Ghemawat had its fair of success, too, according to the

New Yorker's https://www.newyorker.com/magazine/2018/12/10/the-friendship-that-made-google-huge.

4 Interaction Environment

As kid of the 90s, reading about a new editor or fancy shell coming out triggers playing Inspector Gadget's tune in my head. To invest a few fun hours into exploring a new tool that might save me a split second or two in processes that I run a hundred times on a daily basis may not always pay off by the hour even in a lifetime of programming, but is totally worth it in most case - at least to me. Repetitive work is tiring and error prone, adds to cognitive workload and shifts focus away from hard-to-tackle puzzles.

That being said, I am nowhere near the vim aficionados whose jaw-dropping editing speed probably takes 1000s of hours to master and configure, and would not advise to overengineer setting up your environment. Like often, finding a personal middle ground between feeling totally uncomfortable outside of graphical user interfaces and editing code at a 100 words per minute pace is key.

Probably, investing in an initial setup of a solid editor plus some basic terminal workflows is a good starting point that can be revisted every once in a while, e.g., when we start a larger new project. What editor you will center your environment around will depend a lot on the programming language you chose. Yet, it also depends on your personal preference of how much investment, support and supportive features you want. The following features / criteria are likely the most influential when composing a programming environment for statisitcal computing:

- *code highlighting*, i.e., use of colors to highlight the structure of your code

- *linting* scans the code for potential errors and displays hints in the editor
- integrated *debuggers* allow to run parts of the code line by line and to inspect the inner workings of functions
- *multi language support* is important when work multiple programming or markup languages.
- *terminal integration* helps to run stuff using system commands
- git integration helps interact with git version control and do basic add, commit, push operations through the editor's GUI
- build tools for programs that need rendering and compilation
- customizable through add-ins / macros

4.1 Integrated Development Environments (IDE)

While some prefer lightning fast editors such as Sublime Text that are easy to customize but rather basic out-of-the-box, IDEs are the right choice for most people. In other words, it's certainly possible to move a five person family into a new home by public transport, but it is not convenient. The same holds for (plain) text editors in programming. You can use them, but most people would prefer an Integrated Development Environment (IDE) just like they prefer to use a truck when they move. IDEs are tailored to the needs and idiosyncrasies of a language, some working with plugins and covering multiple languages. Others have a specific focus on a single language or a group of languages.

4.1.1 RStudio

RStudio's IDE *RStudio* has become the default environment for most R people and people who mostly use R but use C or Python and Markdown in addition. The open source RStudio IDEs version ships for free as RStudio Desktop and RStudio Server Open Source. In addition the creators of RStudio offer commercially supported versions of both the desktop and

server version (RStudio Workbench). If you want your environment to essentially look like the environment of your peers, R Studio is a good choice. To have the same visual image in mind can be very helpful in workshops, coaching or teaching.

RStudio has 4 control panes which the user can fill with a different functionality like script editor, R console, terminal, file explorer, environment explorer, test suite, build suite and many others. I advise newcomers to change the default to have the script editor and the console next to each other (instead of below each other). That is because (at least in the Western world) we read from left to right and send source code from left to right in order to execute it in the console. Combine this practice with the *run selection shortcut* (cmd+enter or ctrl+enter on a PC) and you have gained substantial efficiency compared to leaving your keyboard, reaching four mouse and finding the right button. In addition, this workflow should allow you to see larger chunks[1] of your code as well as your output.

Explore Extensions

- explore addins
- explore the API

Favorite Shortcuts

- use cmd+enter (ctrl+enter on PCs) to send selected code from the script window (on the left) to the console (on the right)
- cmd+shift+option+R (roxygen while on function)
- use ctrl 1,2 to

[1] Many coding conventions recommend to have no more than 80 characters in one line of code. Sticking to this convention should prevent cutting off your code horizontally.

Pitfalls

- save defaults
- git integration
- below vs beyond

4.1.2 Visual Studio Code

Unlike RStudio, Microsoft's Visual Studio Code started out as modular, general purpose IDE not focused on a single language. In the meantime there is not only great Python support, but also auto-completion, code highlighting for R or Julia and many other languages.

4.1.3 Others

Here are some of the features you are looking for in an IDE for programming with data:

For R, the Open Source Edition of R Studio Desktop is the right choice for most people. (If you are working in a team, R Studio's server version is great. It allows to have a centrally managed server which clients can use through their a web browser without even installing R and R Studio locally.) R Studio has solid support for a few other languages often used together with R, plus it's customizable. The French premier thinkR Colin_Fay gave a nice tutorial on Hacking R Studio at the useR! 2019 conference.

Screenshot of the coverpage of the R Studio website in fall 2020. The site advertises R Studio as an IDE for both languages R and Python. Remember The Choice that Doesn't Matter?

While R Studio managed to hold its ground among R aficionados as of fall 2020, Microsoft's free *Visual Studio Code* has blown the competition out of the water otherwise. Microsoft's IDE is blazing fast, extendable

and polyglot. VS Code Live Share is just one rather random example of its remarkably well implemented features. Live share allows developers to edit a document simultaneously using multiple cursors in similar fashion to Google Docs, but with all the IDE magic. And in a Desktop client.

Another approach is to go for a highly customizable editor such as Sublime or Atom. The idea is to send source code from the editor to interchangeable REPLs which can be adapted according to the language that needs to be interpreted. That way a good linter / code highlighter for your favorite language is all you need to have a lightweight environment to run things. An example of such a customization approach is Christoph Sax' small project Sublime Studio.

Other examples for popular IDEs are Eclipse (mostly Java but tons of plugins for other languages), or JetBrains' IntelliJ (Java) and PyCharm (Python).

A word on vim and emacs (just so you know what people are talking about)

4.2 The Console / Terminal

In addition to the editor you will spent most of your time with, it is also worthwile to put some effort into configuring keyboard driven interaction with your operating systems. And again, you do not need to be a terminal virtuoso to easily outperform mouse pushers, a terminal carpentry level is enough. Terminals come in all shades of gray, integrated into IDEs, built into our operating system or as pimped third party applications. A program called a *shell* runs inside the terminal application to run commands and display output. *bash* is probably the most known shell program, but there tons of different flavors. FWIW, I love fish shell (combined with iterm2) for its smooth auto-completion, highlighting and its extras.

In the Windows world, the use of terminal applications has been much less common than for OSX/Linux – at least for non-systemadmins. Git Bash which ships with git version control installations on Windows mitigates this shortcoming as it provides a basic UNIX style console on windows. For a full fledged UNIX terminal experience, I suggest to use a full terminal emulator like CYGWIN. More recent, native approaches like powershell brought the power of keyboard interaction at OS level to a broader Windows audience – albeit with different Windows specific syntax. The ideas and examples in this book are limited to UNIX flavored shells.

Table 4.1: Basic Unix Terminal commands

Command	What it does
ls	list files (and directories in a directory)
cd	change directory
mv	move file (also works to rename files)
cp	copy files
mkdir	make directory
rmdir	remove directory
rm -rf	(!) delete everything recursively. DANGER: shell does not ask for confirmation, just wipes out everything.

4.2.1 Remote Connections SSH, SCP

One of the most important use cases of the console for data analysts is the ability to log into other machines, namely servers that most often run on LINUX. Typically we use the SSH protocol to connect to a (remote) machine that allows to connect through port 22.

> Note that sometimes firewalls limit access to ports other than those needed for surfing the web (8080 for http:// and 443 for https://) so you can only access port 22 inside an organization's VPN network.

To connect to a server using a username and password simply use your console's ssh client like this:

```
ssh mbannert@someserver.org
```

You will often encounter another login procedure though. RSA key pair authentication is more secure and therefore preferred by many organizations. You need to make sure the public part of your key pair is located on the remote server and hand the ssh command the private file:

```
ssh -i ~/.ssh/id_rsa mbannert@someserver.org
```

For more details on RSA key pair authentication, check this example from the case study chapter Section 11.1.

While ssh is designed to log in to a remote server and from then on issue commands like the server was a local linux machine, `scp` copies files from one machine to another.

```
scp -i ~/.ssh/id_rsa ~/Desktop/a_file_on_my_desktop
 mbannert@someserver.org:/some/remote/location/
```

The above command copies a file dwelling on the users desktop into a /some/remote/location on the server. Just like ssh, secure copy (scp) can use RSA key authentication, too.

4.2.2 Git through the Console

Another common use case of the terminal is managing git version control. Admittedly, there is git integration for many IDEs that allow you to point and click your way commits, pushes and pulls as well as dedicated git clients like GitHub Desktop or Source Tree. But there is nothing like the console in terms of speed and understanding what you really do. The Chapter 5 sketches an idea of how to operate git through the console from the very basics to a feature branch based workflow.

4.2.3 A Word On Cron Jobs

Named after the UNIX job scheduler *cron*, a cron job is a task that runs periodically at fixed times. Pre-installed in most LINUX server setups, data analysts often use cronjobs to regularly run batch jobs on remote servers. Cron jobs use a funny syntax to determine when jobs run.

```
#min hour day month weekday
15 5,11 * * 1 Rscript run_me.R
```

The first position denotes the minute mark at which the jobs runs - in our case 15 minutes after the new hour started. The second mark denotes hours during the day – in our case the 5th and 11th hour. The asterisk * is a wildcars expression running the job on every day of the month and in every month throughout the year. The last position denotes the weekday, here we run our job solely on Mondays.

More advanced expressions can also handle running a job at much shorter intervals, e.g., every 5 minutes.

```
*/5 * * * * Rscript check_status.R
```

To learn and play with more expressions check crontab guru. If you have more sophisticated use cases, like overseeing a larger amount of jobs or execution on different nodes consider using Apache Airflow as workflow scheduler.

5 Git Version Control

As stated before, version control may be the single most important thing to take away from *Hacking for Social Sciences*. In this chapter about the way developers work, I will stick to version control with *git*. The stack discussion of the previous chapter features a few more version control systems, but given git's dominant position, we will stick solely to git in this introduction to version control.

5.1 What is Git Version Control?

Git is a decentralized version control system. It manages different versions of your source code (and other text files) in a simple but efficient manner that has become the industry standard: The git programm itself is a small console programm that creates and manages a hidden folder inside the folder you put under version control (you know those folders with a leading dot in their foldername, like .myfolder). This folder keeps track of all differences between the current version and other versions before the current one.

Meaningful commit messages help to make sense of a project's history.

The key to appreciate the value of git is to appreciate the value of semantic versions. Git is *not* Dropbox nor Google Drive. It does *not* sync automagically (even if some Git GUI Tools suggest so). Because these GUIs tools[1]

[1] GitHub Desktop, Atlassian's Source Tree and Tortoise are some of the most popular choices if you are not a console person.

may be convenient but do not really help to improve your understanding of the workflow, we will use the git console throughout this book. As opposed to the sync approaches mentioned above, a version control system allows to summarize a contribution across files and folders based on what this contribution is about. Assume you got a cool pointer from an econometrics professor at a conference and you incorporated her advice in your work. That advice is likely to affect different parts of your work: your text and your code. As opposed to syncing each of these files based on the time you saved them, version control creates a version when you decide to bundle things together and to commit the change. That version could be identified easily by its commit message 'incorporated advice from Anna (discussion at XYZ Conf 2020)'.

5.2 Why Use Version Control in Research?

> A version control based workflow is a path to your goals that rather consists of semantically relevant steps instead of semantically meaningless chunks based on the time you saved them.

In other more blatant, applied words: naming files like `final_version_your_name.R` or `final_final_correction_collaboratorX_20200114.R` is like naming your WiFi `dont_park_the_car_in_the_frontyard` or `be_quiet_at_night` to communicate with your neighbors. Information is supposed to be sent in a message, not a file name. With version control it is immediately clear what the most current version is - no matter the file name. No room for interpretation. No need to start guessing about the delta between the current version and another version.

Also, you can easily try out different scenarios on different branches and merge them back together if you need to. Version control is a well established industry standard in software development. And it is relatively easy to adopt. With datasets growing in size and complexity, it is only natural to improve management of the code that processes these data.

Academia has probably been the only place that would allow you to dive into hacking at somewhat complex problems for several years w/o ever taking notice of version control. As a social scientist who rather collaborates in small groups and writes moderate amount of code, have you ever thought about how to collaborate with 100+ persons in a big software project? Or to manage ten thousands of lines of code and beyond? Version control is an important reason why these things work. And it's been around for decades. But enough about the rant…

5.3 How Does Git Work ?

> This introduction tries narrow things down to the commands that you'll need if want to use git in similar fashion to what you learn from this book. If you are looking for more comprehensive, general guides, three major git platforms, namely Atlassian's Bitbucket, GitHub and Gitlab offer comprehensive introductions as well as advanced articles or videos to learn git online.

The first important implication of decentralized version control is that all versions are stored on the local machines of every collaborator, not just on a remote server (this is also a nice, natural backup of your work). So let's consider a single local machine first.

Locally, a git repository consists of a *checkout* which is also called current *working copy* soon. This is the status of the file that your file explorer or your editor will see when you use them to open a file. To checkout a different version, one needs to call a commit by its unique commit hash and checkout that particular version.

If you want to add new files to version control or bundle changes to some existing files into a new commit, add these files to the staging area, so

they get committed next time a commit process is triggered. Finally committing all these staged changes under another commit id a new version is created.

5.4 Moving Around

So let's actually do it. Here's a three stage walk through of git commands that should have you covered in most use cases a researcher will face. Note that git has some pretty good error message that guess what could have gone wrong. Make sure to read them carefully. Even if you can't make sense of them, your online search will be a lot more efficient when you include these messages.

Stage 1: Working Locally

Command	Effect
git init	put current directory and all its subdirs under version co
git status	shows status
git add file_name.py	adds file to tracked files
git commit -m 'meaningful msg'	creates a new version/commit out of all staged files
git log	show log of all commit messages on a branch
git checkout some-commit-id	go to commit, but in detached HEAD state
git checkout main-branch-name	leave temporary state, go back to last commit

Stage 2: Working with a Remote Repository

Though git can be tremendously useful even without collaborators, the real fun starts when working together. The first step en route to get others involved is to add a remote repository.

Command	Effect
git clone	creates a new repo based on a remote one
git pull	get all changes from a linked remote repo
git push	deploy all commit changes to the remote repo
git fetch	fetch branches that were created on remote
git remote -v	show remote repo URL
git remote set-url origin https://some-url.com	set URL to remote repo

Stage 3: Branches

Branches are derivatives from the main branch that allow to work on different feature at the same time without stepping on someone elses feet. Through branch repositories can actively maintain different states.

Command	Effect
git checkout -b branchname	create new branch named branchname
git branch	show locally available branches
git checkout branchname	switch to branch named branchname
git merge branchname	merge branch named branchname into current branch

Fixing Merge Conflicts

In most cases *git* is quite clever and can figure out which is the desired state of a file when putting two versions of it together. When git's *recursive strategy* is possible, git it wil merge versions automatically. When the same lines were affected in different versions, git cannot tell which line should be kept. Sometimes you would even want to keep both changes.

But even in such scenario fixing the conflict is easy. Git will tell you that your last command caused a merge conflict and which files are conflicted. Open these files and see all parts of the file that are in question.

Ouch! We created a conflict by editing the same line in the same file on different branches and trying to but these branches back together.

Luckily git marks the exact spot where the conflict happens. Good text editors / IDEs ship with cool colors to highlight all our options.

go for the current status or take what's coming in from the a2 branch?

Some of the fancier editors even have git conflict resolve plugins that let you walk through all conflict points.

In VS Code you can even click the option.

At the and of the day, all do the same, i.e., remove the unwanted part including all the marker gibberish. After you have done so, save, commit and push (if you are working with a remote repo) . Don't forget to make sure you kinked out ALL conflicts.

5.5 Collaboration Workflow

The broad acceptance of git as a framework for collaboration has certainly played an important role in git's establishment as an industry standard.

5.5.1 Feature Branches

This section discusses real world collaboration workflows of modern open source software developers. Hence the prerequisites are a bit different in order to benefit the most from this section. Make sure you are past being able to describe and explain git basics, be able to create and handle your own repositories.

If you had only a handful of close collaborators so far, you may be fine with staying on the main branch and trying not to step on each others feet. This is reasonable because it is not useful to work asynchronously on exact the same lines of code anyway. Nevertheless, there is a reason

why *feature-branch-based* workflows became very popular among developers: Imagine you collaborate less synchronously, maybe with someone in another timezone. Or with a colleague who works on your project, but in a totally different month during the year. Or, most obviously, we someone you have never met. Forks and *feature-branch-based* workflows is the way a lot of modern open source projects are organized.

Forks are just a way to contribute via feature branches even in case you do not have write access to a repository. But let's just have look at the basic case in which you are part of the team first. Assume there is already some work done, some version of the project is already up on GitHub. You join as a collaborator and are allowed to push changes now. It's certainly not a good idea to simply add things without review to a project's production. Like if you got access to modify the institute's website and you made your first changes and all of a sudden the website looks like this:

It used to be subtle and light gray. I swear!

Bet everybody on the team took notice of the new team member by then. In a feature branch workflow you would start from the latest production version. Remember, git is decentralized and you have all versions of your team's project. Right at your fingertips on your local machine. Create a new branch named indicative of the feature you are looking to work on.

```
git checkout -b colorways
```

You are automatically being switched to the freshly created branch. Do your thing now. It could be just a single commit, or several commits by different persons. Once you are done, i.e., commited all changes, add your branch to the remote repository by pushing.

```
git push -u origin colorways
```

This will add a your branch called *colorways* to the remote repository. If you are on any major git platform with your project, it will come with a decent web GUI. Such a GUI is the most straight forward way to do the next step: get your Pull Request (PR) going.

Github pull request dialog: Select the *pull request*, choose which branch you merge into which target branch.

As you can see, git will check whether it is possible to merge automatically w/o interaction. Even if that is not possible, you can still issue the pull request. When you create the request you can also assign reviewers, but you could also do so at a later stage.

Note, even after a PR was issued you can continue to add commits to the branch about to be merged. As long as you do not merge the branch through the Pull Request, commits are added to the branch. In other words your existing PR gets updated. This is a very natural way to account for reviewer comments.

Pro-Tipp: Use commit messages like 'added join to SQL query, closes #3'. The key word 'closes' or 'fixes', will automatically close issues referred to when merged into the main branch.

Once the merge is done, all your changes are in the main branch and you and everyone else can pull the main branch that now contains your new feature. Yay!

5.5.2 PRs and Forks

6 Data Management

Admittedly, I have said the same thing about Chapter 5 on version control, yet *data management* may be the single most impactful chapter of this book. This may be the case in particular in case you came from an environment that mostly organized data in spreadsheets shared via e-mail or network drives. To contextualize data and think about (long term) data management is a step into a bigger world.

After decades of information engineering and computer science, some can't help but wonder why we have not found one perfect, one-size-fits-all form of data. In fact, a substantial part of programming with data deals with transforming data from one form into another. This chapter intends to give an idea of the aspects of data management most relevant for data analytics. Hopefully this chapters helps the reader to assess how far they wanted to dig into data management.

6.1 Forms of Data

In research and analytics, data appear in a plethora of different forms. Yet, most researchers and business analysts are mostly trained to handle different flavors of two dimensional data as in **one-observation-one-row**. Adhoc studies conducted once result in *cross sectional data*: one line per observation, columns represent variables. Sensor data, server logs or forecasts of official statistics are examples of single variable data observed over time. These single variable, longitudinal data are also known as *time series*. Multivariate time series, i.e., multi-variable, longitudinal

data are often referred to as *panel data*. In our heads, all of these forms of data are typically represented as rectangular, two-dimensional one-line-per-observation, spreadsheet like tables. Here are a few easy-to-reproduce examples using popular R demo datasets.

```
h <- head(mtcars)
h
```

```
                  mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
dim(h)
```

```
[1]  6 11
```

The above output shows an excerpt of the *mtcars* **cross-sectional** dataset with 6 lines and 11 variables. *Airpassenger* is a **time series** dataset represented in an R *ts* object which is essentially a vector with time based index attribute.

```
AirPassengers
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949 112 118 132 129 121 135 148 148 136 119 104 118
1950 115 126 141 135 125 149 170 170 158 133 114 140
```

```
1951 145 150 178 163 172 178 199 199 184 162 146 166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
1955 242 233 267 269 270 315 364 347 312 274 237 278
1956 284 277 317 313 318 374 413 405 355 306 271 306
1957 315 301 356 348 355 422 465 467 404 347 305 336
1958 340 318 362 348 363 435 491 505 404 359 310 337
1959 360 342 406 396 420 472 548 559 463 407 362 405
1960 417 391 419 461 472 535 622 606 508 461 390 432
```

Let's create **multivariate time series (panel)** dataset, i.e., multiple variables observed over time:

```
d <- data.frame(Var1 = rnorm(10, 0),
           Var2 = rnorm(10, 10),
           Var3 = rnorm(10, 30))
multi_ts <- ts(d, start = c(2000,1), frequency = 4)
multi_ts
```

```
              Var1      Var2     Var3
2000 Q1 -0.26675672  9.657051 29.05627
2000 Q2 -1.77292465 10.364089 30.98669
2000 Q3 -0.07073601  8.163066 29.11507
2000 Q4  0.30744169 10.083991 31.19471
2001 Q1  0.52014406  9.913426 29.38247
2001 Q2  1.03881994  9.389232 29.24246
2001 Q3 -0.08779039 10.269901 29.10766
2001 Q4  0.21828083  9.649982 30.63123
2002 Q1 -0.92812849 11.056149 29.20288
2002 Q2  1.01645582  9.270222 28.58044
```

A Note on Long Format vs. Wide Format

The above multi-variable time series is shown in what the data science community calls *wide* format. In this most intuitive format, every column represents one variable, time is on the Y-axis. The counterpart is the so called *long* format shown below. The long format is a machine friendly, flexible way to represent multi-variable data without altering the number of columns with more variables.

```
library(tsbox)
ts_dt(multi_ts)[1:15,]
```

```
      id       time       value
 1: Var1 2000-01-01 -0.26675672
 2: Var1 2000-04-01 -1.77292465
 3: Var1 2000-07-01 -0.07073601
 4: Var1 2000-10-01  0.30744169
 5: Var1 2001-01-01  0.52014406
 6: Var1 2001-04-01  1.03881994
 7: Var1 2001-07-01 -0.08779039
 8: Var1 2001-10-01  0.21828083
 9: Var1 2002-01-01 -0.92812849
10: Var1 2002-04-01  1.01645582
11: Var2 2000-01-01  9.65705094
12: Var2 2000-04-01 10.36408913
13: Var2 2000-07-01  8.16306584
14: Var2 2000-10-01 10.08399057
15: Var2 2001-01-01  9.91342586
```

The ability to transform data from one format into the other and to manipulate both formats is an essential skill for any data scientist or data engineer. It is important to point out that the ability to do the above transformations effortlessly is an absolute go-to skill for people who want to use programming to

run analysis. (Different analyses or visualizations may require one form or the other and ask for quick transformation).

Hence popular data science programming languages offer great toolsets to get the job done. Mastering these toolboxes is not the focus of this book. R for Data Science and the Carpentries are good starting points if you feel the need to catch up or solidify your know-how.

Yet, not all information suits a two dimensional form. Handling nested or unstructured information is one of the fields where the strength of a programming approach to data analysis and visualization comes into play. Maps are a common form of information that is often represented in nested fashion. For an example of nested data, let's take a look at the map file and code example case study in Section 11.6. In memory, i.e., in our R session the data is represented in a list that contains multiple list may contain more lists nested inside.

```
library(jsonlite)

json_ch <- jsonlite::read_json(
  "https://raw.githubusercontent.com/mbannert/maps/master/ch_bfs_regions.geojson"
)
ls.str(json_ch)
```

```
crs : List of 2
 $ type      : chr "name"
 $ properties:List of 1
features : List of 7
 $ :List of 3
 $ :List of 3
 $ :List of 3
 $ :List of 3
 $ :List of 3
 $ :List of 3
```

```
 $ :List of 3
type :  chr "FeatureCollection"
```

Another example of nested but structured data are HTML or XML trees obtained from scraping websites. Typically, web scraping approaches like rvest or beautiful soup parse the hierarchical Document Object Model (DOM) and turn it into an in-memory representation of a website's DOM. For DOM parsing example see CASE STUDY XYZ.

6.2 Representing Data in Files

To create the above examples of different forms of data, it was mostly sufficient to represent data in memory, in this case within an R session. As an interpreted language, an R interpreter has to run at all times when using R. The very same is true for Python. Users of these language can work interactively, very much like with a pocket calculator on heavy steroids. All functions, all data, are in loaded into a machine's RAM (memory) represented as objects of various classes. This is convenient, but has an obvious limitation: once the sessions ends, the information is gone. Hence we need to have a way to store at least the results of our computation in persistent fashion.

Just like in office or image editing software, the intuitive way to store data persistently from a programming language is to store data into files. The choice of the file format is much less straight forward in our case though. The different forms of data discussed above, potential collaborators and interfaces are factors among others than weigh into our choice of a file format.

6.2.1 Spreadsheets

Based on our two dimension focused intuition and training spreadsheets are the on-disk analog of data.frames, data.tables and tibbles. Formats like **.csv** or **.xlsx** are the most common way to represent two dimensional data on disk.
On the programming side the ubiquity of spreadsheets leads to a wide variety of libraries to parse and write different spreadsheet formats.

```
import csv
import pandas as pd

d = {'column1': [1,2], 'column2': [3,4]}
df = pd.DataFrame(data=d)
df.to_csv("an_example.csv", sep=";", encoding='utf-8')
```

Comma separated values (.csv)[1] are good and simple option. Their text based nature makes .csv files language agnostic and human readable through a text editor.

```
;column1;column2
0;1;3
1;2;4
```

Though Excel spreadsheets are a convenient interface to office environments that offer extras such organization into workbooks, the simpler .csv format has advatanges in machine-to-machine communication and as an interface between different programming languages and tools. For example, web visualization libraries such as highcharts or echarts are most commonly written in javascript and can conveniently consume data from .csv

[1] Note that commas are not always necessarily the seperator in .csv files. Because of the use of commas as decimal delimiters in some regions, columns are also often separated by semicolons to avoid conflicts.

files. The above example .csv file was written by Python and is now easily read by R.

```
library(readr)
csv <- readr::read_csv2("an_example.csv")
```

```
i Using "','" as decimal and "'.'" as grouping mark. Use `read_delim()` for m
```

```
New names:
* `` -> ...1
```

```
Rows: 2 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ";"
dbl (3): ...1, column1, column2

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this mess
```

```
csv
```

```
# A tibble: 2 x 3
   ...1 column1 column2
  <dbl>   <dbl>   <dbl>
1     0       1       3
2     1       2       4
```

6.2.2 File Formats for Nested Information

For many data engineers and developers. Javascript Object Notation (JSON) has become the go-to file format for nested data. Just like with .csv basically every programming language used in data science and analytics has libraries to serialize and de-serialize JSON (read and write). Though harder to read for humans than .csv, pretty-fied JSON with a decent highlighting color scheme is easy to read and gives the human reader a good understanding of the hierarchy at hand. The added complexity comes mostly from the nested nature of the data, not so much from the file format.

```
library(jsonlite)

li <- list(
  element_1 = head(mtcars, 2),
  element_2 = head(iris, 2)
)

toJSON(li, pretty = TRUE)
```

```
{
  "element_1": [
    {
      "mpg": 21,
      "cyl": 6,
      "disp": 160,
      "hp": 110,
      "drat": 3.9,
      "wt": 2.62,
      "qsec": 16.46,
      "vs": 0,
      "am": 1,
```

```
      "gear": 4,
      "carb": 4,
      "_row": "Mazda RX4"
    },
    {
      "mpg": 21,
      "cyl": 6,
      "disp": 160,
      "hp": 110,
      "drat": 3.9,
      "wt": 2.875,
      "qsec": 17.02,
      "vs": 0,
      "am": 1,
      "gear": 4,
      "carb": 4,
      "_row": "Mazda RX4 Wag"
    }
  ],
  "element_2": [
    {
      "Sepal.Length": 5.1,
      "Sepal.Width": 3.5,
      "Petal.Length": 1.4,
      "Petal.Width": 0.2,
      "Species": "setosa"
    },
    {
      "Sepal.Length": 4.9,
      "Sepal.Width": 3,
      "Petal.Length": 1.4,
      "Petal.Width": 0.2,
      "Species": "setosa"
    }
```

```
  ]
}
```

The above example shows the first two lines of two different, unrelated rectangular datasets. Thanks to the hierarchical nature of JSON both datasets can be stored in the same file albeit totally different columns. Again, just like .csv, JSON works well as an interface, but is more flexible than the former.

Besides JSON, **XML** is the most common format to represent nested data in files. Though there are a lot of overlapping use cases, there is a bit of a different groove around both of these file formats. JSON is perceived as more light weight and close to 'the web' while XML is the traditional, very explicit no-nonsense format. XML has a **Document Type Defintion (DTD)** that defines the structure of the document and which elements and attributes are legal. Higher level formats use this more formal approach for as XML based defintion. SDMX[2], a world-wide effort to provide a format for exchange statistical data and meta data, is an example of such a higher level format build on XML.

```
<CompactData xmlns:c="http://www.SDMX.org/resources/SDMXML/schemas/v2_0/compact" xmlns:sd
<Header>
<ID>kofbarometer</ID>
<Test>false</Test>
<Prepared>2022-07-27 23:19:15</Prepared>
<Sender id="KOF">
```

[2]SDMX.org about SDMX: SDMX, which stands for Statistical Data and Metadata eXchange is an international initiative that aims at standardising and modernising ("industrialising") the mechanisms and processes for the exchange of statistical data and metadata among international organisations and their member countries. SDMX is sponsored by seven international organisations including the Bank for International Settlements (BIS), the European Central Bank (ECB), Eurostat (Statistical Office of the European Union), the International Monetary Fund (IMF), the Organisation for Economic Cooperation and Development (OECD), the United Nations Statistical Division (UNSD), and the World Bank

```
<Name xml:lang="en">KOF Swiss Economic Institute</Name>
<Contact>
<Name>KOF Swiss Economic Institute</Name>
<URI>http://www.kof.ethz.ch/prognosen-indikatoren/indikatoren/kof-konjunkt
</Contact>
</Sender>
</Header>
<sdds:DataSet>
<sdds:Series DATA_DOMAIN="FLI" REF_AREA="CH" INDICATOR="AL" COUNTERPART_AR
<sdds:Obs TIME_PERIOD="1991-01" OBS_VALUE="79.7534465342489" OBS_STATUS="A
<sdds:Obs TIME_PERIOD="1991-02" OBS_VALUE="71.8659035408967" OBS_STATUS="A

....

<sdds:Obs TIME_PERIOD="2022-05" OBS_VALUE="96.4419659191275" OBS_STATUS="A
<sdds:Obs TIME_PERIOD="2022-06" OBS_VALUE="95.1748194208808" OBS_STATUS="A
<sdds:Obs TIME_PERIOD="2022-07" OBS_VALUE="90.08821804515" OBS_STATUS="A"/
</sdds:Series>
</sdds:DataSet>
</CompactData>
<!--  i: 1403 : 1660204311  -->
```

The above example shows and excerpt of the main economic forward looking indicator (FLI) for Switzerland, the KOF Economic Barometer, represented in an SDMX XML file. Besides the value and the date index, several attributes provide the consumer with an elaborate data description. In addition, other nodes and their children provide information like *Contact* or *ID* in the very same file. Note that modern browser often provide code folding for nodes and highlighting to improve readability.

6.2.3 A Word on Binaries

Unlike all file formats discussed above, binaries cannot be read by humans using a simple text editor. In other words, you will need the software that wrote the binary to read it again. If that software was expensive and/or exotic, your work is much less accessible, more difficult to share and harder to reproduce. Though this disadvantage of binaries is mitigated when you use freely available open source software, storing data in binaries can still be a hurdle.

But of course binaries do have advantages, too: binaries can compress their content and save space. Binaries can take on all sorts of in-memory objects including functions, not just datasets. In other words, binaries can bundle stuff. Consider the following load/save operation in R:

```
bogus <- function(a,b){
  a + b
}

data(Airpassengers)
data(mtcars)

s <- summary(mtcars)

save("bogus", "Airpassengers","s", file="bundle.RData")
```

In memory, *bogus* is a *function*, Airpassengers is an R *time series* object and *s* is a *list* based summary object. All of these objects can be stored in a single binary RData file using *save()*. A fresh R session can now *load()* everything stored in that file.

```
load("bundle.RData")
```

> Notice that unlike reading a .csv or .json file, the call does not make any assignments into a target object. This is because all objects are loaded into an R environment (*.globalEnv* by default) with their original names.

6.2.4 Interoperable File Formats

Interoperable file formats cover some middle ground between the options described above. The *cross-language in-memory development platform* Apache Arrow is a well established project that also implements file formats that work across many popular (data science) environments. Though the major contribution of the Apache Arrow project is to allow to share in-memory data store across environments, I will just show it as an example for interoperable file formats here. Nevertheless if you're interested in a modern, yet established cross environment data science project, diggin deeper into Apache Arrow is certainly a fun experience.

From the Apache Arrow documentation:

```
library(dplyr)
library(arrow)
data("starwars")
file_path_sw <- "starwars.parquet"
write_parquet(starwars, file_path_sw)
```

The above R code writes the *starwars* demo dataset from the *dplyr* R package to a temporary .parquet file. The arrow R packages comes with the necessary toolset to write the open source columnar[3] .parquet format. Though they are not text files, .parquet files can be read and written from different environments and consume the file written with R. The below code uses the arrow library for Python to read the file we have just written with R.

[3] see also

```
import pyarrow.parquet as pa
pa.read_table("starwars.parquet")
```

```
pyarrow.Table
name: string
height: int32
mass: double
hair_color: string
skin_color: string
eye_color: string
birth_year: double
sex: string
gender: string
homeworld: string
species: string
films: list<item: string>
  child 0, item: string
vehicles: list<item: string>
  child 0, item: string
starships: list<item: string>
  child 0, item: string
----
name: [["Luke Skywalker","C-3PO","R2-D2","Darth Vader","Leia Organa",...,"Rey","Poe Dameron"
height: [[172,167,96,202,150,...,null,null,null,null,165]]
mass: [[77,75,32,136,49,...,null,null,null,null,45]]
hair_color: [["blond",null,null,"none","brown",...,"brown","brown","none","unknown","brown"]
skin_color: [["fair","gold","white, blue","white","light",...,"light","light","none","unknow
eye_color: [["blue","yellow","red","yellow","brown",...,"hazel","brown","black","unknown","b
birth_year: [[19,112,33,41.9,19,...,null,null,null,null,46]]
sex: [["male","none","none","male","female",...,"female","male","none",null,"female"]]
gender: [["masculine","masculine","masculine","masculine","feminine",...,"feminine","masculi
homeworld: [["Tatooine","Tatooine","Naboo","Tatooine","Alderaan",...,null,null,null,null,"Na
```

...

Here's Julia reading our Parquet file:

```
using Parquet
sw = Parquet.File("starwars.parquet")
```

```
Parquet file: starwars.parquet
    version: 2
    nrows: 87
    created by: parquet-cpp-arrow version 9.0.0
    cached: 0 column chunks
```

```
sw.schema.schema
```

```
21-element Vector{Parquet.PAR2.SchemaElement}:
 required schema {

 optional BYTE_ARRAY name # (from UTF8)

 optional INT32 height

 optional DOUBLE mass

 optional BYTE_ARRAY hair_color # (from UTF8)

 optional BYTE_ARRAY skin_color # (from UTF8)

 optional BYTE_ARRAY eye_color # (from UTF8)

 optional DOUBLE birth_year
```

```
optional BYTE_ARRAY sex # (from UTF8)

optional BYTE_ARRAY gender # (from UTF8)


optional films # (from LIST)  {

repeated list {

optional BYTE_ARRAY item # (from UTF8)

optional vehicles # (from LIST)  {

repeated list {

optional BYTE_ARRAY item # (from UTF8)

optional starships # (from LIST)  {

repeated list {

optional BYTE_ARRAY item # (from UTF8)
```

> When I composed this example reading and writing parquet files in different environments I ran into several compatibility issues. This shows that the level of interoperability is not at the same level as the interoperability of text files.

6.2.4.1 A Note on Overhead {.unnumbered}

The *parquet* format is designed to read and write files swiftly and to consume less disk space than text files. Both features can become particularly

relevant in the cloud. Note though that parquet comes with some overhead which may eat up gains if datasets are small. Consider our *starwars* dataset. At 87 rows and 14 columns the dataset is rather small.

```
library(readr)
write_csv(starwars, file = "starwars.csv")
dim(starwars)
```

```
[1] 87 14
```

```
round(file.size("starwars.parquet") / file.size("starwars.csv"), digits =
```

```
[1] 1.47
```

Hence the overhead of a schema implementation and other meta information outweighs parquet's compression for such a small dataset leading to a parquet file that is almost 1.5 times larger than the corresponding csv file. Yet, parquet already turns the tables for the *diamonds* demo dataset from the *ggplot2* R package which is by no means a large dataset.

```
library(ggplot2)
data(diamonds)
write_csv(diamonds, file = "diamonds.csv")
write_parquet(diamonds, "diamonds.parquet" )
round(file.size("diamonds.parquet") / file.size("diamonds.csv"), digits =
```

```
[1] 0.21
```

The parquet file for the diamonds dataset has roughly one fifth of the size of the corresponding text file. This is a great example of why there is not one single, perfect, one-size-fits all form of data that emerged from decades of information engineering. So when you choose how you are going to represent data in our project, think about your goals, your most common use or query and a smooth data transformation strategy for when the use cases or goals change.

6.3 Databases

Given the options that file based approaches provide, what is a) the difference and b) the added value of going for database to manage data? The front-and-center difference is the client interface, but there are many more differences and benefits.

The above figure shows a terminal client which users use to send queries written in a query language. The client sends these queries to a database host and either performs an operation on the database quietly or returns a result. The most common example of such a query language is the Structured Query Language (SQL). This leads to a standard way of interaction with the data no matter how the dataset looks like in terms of dimensions, size etc. SQL databases have been around much longer than data science itself and continue to be inevitable as application backends and data archives for many use cases.

```
SELECT * FROM myschema.demotable
```

The above query would return all rows and all columns from a table called *demotable* in a schema called *myschema*. Such a query can easier be send from a standalone database client, a database specific IDE with a built in client such as DataGrid or a programming language. Given the ubiquity of database most basically any programming language has native interfaces to the most common database. And if that is not the case there is the

database management system agnostic ODBC standard that is supported by all major SQL database. The below code shows how to connect from R to PostgreSQL, send queries from within R and receive results as R objects.

```
library(RPostgres)
con <- dbConnect(
  host = "localhost",
  user = "bugsbunny",
  passwd = .rs.AskForPassword("Enter Pw"), # only works within RStudio,
  dbname = "some_db_name"
)

# the result is an R data.frame
res <- dbSendQuery(con, "SELECT * FROM myschema.demotable")

# and we can do R things with it
# such as show the first 6 lines.
head(res)
dbDisconnect(con)
```

Obviously the above example barely shows the tip of the iceberg as it is just meant to illustrate the way we interact with databases as opposed to a file system. To dig a little deeper into databases, I recommend to get a solid understanding of the basic CREATE, SELECT, INSERT, UPDATE, DELETE, TRUNCATE, DROP processes as well as basic JOINs and WHERE clauses. Also, it is helpful to understand the concept of normalization up to the third normal form.

6.3.1 Relational Database Management Systems (RDBMS)

When you need to pick a concrete database technology for your project, the first major choice is whether to go for a relational system or not. Unless

you have a very compelling reason not to, you are almost always better off with a relational database: Relational databases are well established and accessible from any programming language used in programming with data that one could think of. In addition, modern RDBMS implmentations offer many non-relational features such as JSON field types and operations.

I would classify the most popular relational database implementations as follows. First, there is SQLite. As the name suggestions, *SQLite* is a light-weight, stripped down, easy-to-use and install implementation.

> SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. – SQLite.org

SQLite data lives in a single file that the user queries through the *SQLite* engine. Here is an example using that engine from R.

```
library(RSQLite)
db_path <- "devopscarpentry.sqlite3"
con <- dbConnect(RSQLite::SQLite(), db_path)
dbWriteTable(con, dbQuoteIdentifier(con,"mtcars"), mtcars, overwrite = T)
dbWriteTable(con, dbQuoteIdentifier(con,"flowers"), iris, overwrite = T)
```

The above code initiates a *SQLite* database and continues to write the built-in R demo datasets into separate tables in that newly created database. Now we can use SQL to query the data. Return the first three rows of *flowers*:

```
dbGetQuery(con, "SELECT * FROM flowers LIMIT 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
```

Return cars that are more fuel efficient than 30 miles per gallon:

```
dbGetQuery(con, "SELECT * FROM mtcars WHERE mpg > 30")
```

```
   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
2 30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
3 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
4 30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

MySQL can do a little more and also immensely popular, particularly as a database backend for web content management systems and other web based applications. The so-called LAMP stack (Linux, Apache, MySQL and PHP) contributed to its rise decades ago when it fueled many smaller and medium level web projects around the world. In its early days MySQL used to be an independent open source project, but was later on acquired by database juggernaught Oracle as a lite version to go with its flagship product.

While certainly doing its job in millions of installation, MySQL it is not in at same level as Microsoft SQL Server (MSSQL), PostgreSQL and Oracle Database and I suggest one of these three enterprise level database as data store for a research projects that go beyond hosting a blog. Especially when it comes to long term conservation of data and enforcing consistency MSSQL, PostgreSQL and Oracle are hard to beat. Among the three, personally I would always lean towards the license cost free open source PostgreSQL, but fitting into existing ecosystems is a good reason to go

with either of MSSQL or Oracle if you can afford the licenses. For many use cases, there is hardly any difference for the analyst or scientific enduser. PostgreSQL may have the coolest spatial support, MSSQL T-SQL dialect may have some extra convenient queries if your developers mastered the dialect and Oracle may have the edge in performance and Java interaction here and there, but none of these systems is a bad choice.

Another database management that gets a lot of attention recently (and rightfully so) is DuckDB. Because it is mentioned so overwhelmingly positive and often, it is important to understand what it is and when to use it. DuckDB is not yet another competitor that tries to gain some ground from the big three of MSSQL, PostgreSQL and Oracle. DuckDB does offers an SQL interface but is very different in its aims from the traditional SQL databases. DuckDB is serverless and allows to access parquet files via a very fast SQL interface. This makes DuckDB a great tool for interactive analysis and transfer of large result sets but not so suitable for enterprise data warehousing.

6.3.2 A Word on Non-Relational Databases

Among other things, relational databases are ACID (Atomicity, Consistency, Isolation, and Durability) compliant and ask for very little in return to provide us with a framework to keep our data quality high for decades. So unless, you have a very specific use case that translates to a compelling reason to use a non-relational database – stick to SQL. Document oriented storage or very unstructured information could be such a reason to use a non-relational database, yet their JSON support allow to also handle JSON in database cells. About a decade ago mongoDB gained traction, partly piggy-backing the success of Javascript and server-side javascript in particular. In web development the MEAN (mongoDB, expressjs, angular and node) stack become popular and with the bundle the idea of non-relational databases as fast track to a backend spread.

Columnar stores which are also considered non-relational are conceptional very similar to relational databases though denormalized and designed to structure sparse data. Database systems like Apache Cassandra are designed to scale horizontally and be highly available managing massive amounts of data. Cloud applications that distribute data across multiple nodes for high availability benefit from such an approach. Other options include Redis or Couchbase. If your are not happy with the 'beyond-the-scope-of-this-book' argument, blogging experts like Lukas Eder maybe biased but much better educated (and fun) to educate you here.

6.4 Non Technical Aspects of Managing Data

The fact that we can do more data work single handedly than ever before does not only equate to more options. It also means we need to be aware of new issues and responsibilities.

6.4.1 Etiquette

For example, the ability to scrape a website daily and doing so with good intent for the sake of science does not mean a website's AUP (Accetable Use Policy) allows to systematically archive its content.

> Be responsible when scraping data from websites by following polite principles: introduce yourself, ask for permission, take slowly and never ask twice. –CRAN description of the polite R package.

In other words, the new type of researcher discussed in this book needs awareness for potential legal consequences and online community etiquette, and ultimately the ability to choose an approach. The {polite} R package quoted above is an example for an alternative approach that favors etiqutte over hiding IP addresses to avoid access denial.

6.4.2 Privacy

- aggreagtion level, assess the sensitivity of data, anonymize data

6.4.3 Security

- do not store passwords
- tokens, RSA Keys
- access control easier via databases.

6.4.4 Open Data

- publicly funded data

7 Infrastructure

Yes, there is a third area besides your research and carpentry level programming that I suppose you should get an idea of. Again, you do not have to master hosting servers or even clusters, but a decent overview and an idea of when to use what will help you tremendously to plan ahead.

7.1 Why Go Beyond a Local Notebook?

Admittedly, unless your favorite superhero is Inspector Gadget or you just always had a knack for Arduinos, Raspberry Pis or the latest beta version of the software you use, infrastructure may be the one area you perceive as overhead. So why leave the peaceful, well-known shire of our local environment for the uncharted, rocky terroritory of unwelcoming technical documentations and time consuming rabbit holes?

Performance, particularly in terms of throughput is one good reason to look beyond desktop computers. Data proctection regulations that prohibit data downloads may simply force us to not work locally. Or we just do not want a crashing office application bring down a computation that ran for hours or even days. Or we need a server that is online 24/7 to publish a report, website or data visualization.

7.2 Where to Go?

So, where should we go with our project when it outgrows the local environment of our notebooks? This has actually become a tough question because of all the reasonable options out there. Technical advances, almost unparalleled scalability and large profits for the biggest players made modern infrastructure providers offer an incredible variety of products to choose from. Obviously a description of product offerings in a vastly evolving field are not well suited for discussions in a book. Hence *DevOps Carpentry* intends to give an overview to classify the general options and common business models.

7.2.1 Software-as-a-Service (SaaS)

The simplest solution and fastest time-to-market is almost always to go for a SaaS product – particularly if you are not experienced in hosting and just want to get started without thinking about which Linux to use and how to maintain your server. SaaS products abstract all of that away from the user and focus to do one single thing well. The shinyapps.io platform is great example of such a service: users can sign up and deploy their web applications within minutes. The shinyapps.io platform is a particularly interesting example of a SaaS product, because R developers who come from field specific backgrounds other than programming are often not familiar with web development and hosting websites. Some of these developers for whom R might be their first programming language, are suddenly empowered to develop and run online content thanks to the R shiny web application framework that uses R to generate HTML, CSS and Javascript based applications. Still, those applications need to be hosted somewhere. This is exactly what shinyapps.io does. The service solely hosts web applications that were created with the shiny web application framework. This ease of use is also the biggest limitations of SaaS products. A website generated with another tool cannot be deployed easily.

In addition the mere bang-for-buck price is rather high compared to self-hostimg as users pay for a highly, standardized, managed hosting product. Nevertheless, because of the low volume of most projects, SaaS is feasible for many many projects, especially at a proof of concept stage.

> In case you are interested in getting started with shiny, take a look at the shiny case study in this book. The study explains basic concepts of shiny to the user and walks readers through the creation and deployment of a simple app.

SaaS, of course, is neither an R nor a data science idea. Modern providers offer databases, storage calendars, face recognition, location services among other things.

7.2.2 Self Hosted

The alternative approach to buying multiple managed services, is to host your applications by yourself. Since – this applies to most users at least – you do not take your own hardware, connect to your home WiFi and aim to start your own hosting provider, we need to look at different degrees of self-hosting. Larger organizations, e.g., universities, often like to host application on their own hardware, within their own network to have full control of their data. Yet, self-hosting exposes you to issues, such as attacks, that you would not need to worry about as much in a software as a service setting (as long as you trust the service).

Self-hosting allows you to host all the applications you want on a dedicated machine. Self-hosters can configure their server depending on their access rights. Offerings range from root access that allows users to do anything to different shades of managed hosting with more moderated access. Backed by virtual infrastructure, modern cloud provider offer a very dynamic form of self-hosting: their clients can use a web GUIs and/or APIs to add, remove, reboot and shutdown nodes. Users can spin up anything from pre-configured nodes optimized for different use cases to

containerized environments and entire Kubernetes (K8s) clusters in the cloud. Flexible pricing models allow to pay based on usage in very dynamic fashion.

7.3 Building Blocks

Exactly because of this dynamic described above and the ubiquity of the cloud, it is good to know about the building blocks of modern IT infrastructure.

7.3.1 Virtual Machines

Virtual machines (VMs) remain the go-to building blocks for many set ups. Hence university IT, private sector IT, independent hosting providers and online giants all offer VMs. Virtual machines allow to run a virtual computers that have own operating system on some host machine. Running applications on such a virtual computer feels like running an application on a standalone computer dedicated to run this amplication.

> Oracle's Virtual Box is great tool to use and try virtual machines locally. Virtual Box allows to run a Virtual Windows or Linux inside Mac OS and vice versa. Running a virtual box locally may not be the most performant solution but it allows to have several test environments without altering one's main environment.

7.3.2 Containers & Images

At the first glimpse containers look very much like Virtual Machines to the practitioner. The difference is that every Virtual Machine has its own operating system while containers use the the host OS to run a container

engine on to top of the OS. By doing so containers can be very light weight and may take only a few seconds to spin up while spinning up Virtual Machines can take up to a few minutes - just like booting physical computers. Hence docker containers are often used as single purpose environments: Fire up a container, run a task in that environment, store the results outside of the container and shut down the container again.

Docker (Why docker?) is the most popular containerized solution and quickly became synonym to container environments configured in a file. So called docker images are build layer by layer based on other less specific docker images. A DOCKERFILE is the recipe for a new image. Images are blueprints for containers, an image's running instance. A docker runtime environment can build images from DOCKERFILEs and distribute these images to an image registry. The platform dockerhub hosts a plethora of pre-built docker images from ready-to-go database to Python ML environments or minimal Linux containers to run a simple shell script in a lab type environment.

Containers run in a docker runtime environment and can either be used interactively or in batches which execute a single task in an environment specifically built for this task. One of the reasons why docker is attractive to researchers is its open character: Dockerfiles are a good way to share a configuration in a simple, reproducible file, making it easy to reproduce setups. Less experienced researchers can benefit from Dockerhub which shares images for a plethora of purposes from mixed data science setups to database configuration. Side effect free working environments for all sorts of tasks can especially be appealing in exotic and/or dependency heavy cases.

Beside simplification of system administration, *docker* is known for its ability to work in the cloud. All major cloud hosters offer docker environments and the ability to deploy docker containers that were previously developed and tested locally. You can also use docker to tackle throughput problems using container orchestration tools like Docker Swarm or

K8s (say: Kubernetes) to run hundreds of containers (depending on your virtual resources).

7.3.3 Kubernetets (K8s)

Though hosting Kubernetes is clearly beyond the scope of carpentry level devOps, the ubiquity of the term and technology as well as the touching points and similarities with the previously introduced concept of containers justify a brief positioning of Kubernetes. We cannot really see Kubernetes as a building block like the technologies introduced above. K8s is a complete cluster with plenty of features to manage the system and its applications. Kubernetes is designed to run on multiple virtual nodes and distribute processes running in so-called pods across its nodes.

Because Kubernetes is quite some overhead to manage, the big three and their alternatives all offer services such as a container registry to support their clients in running Kubernetes. Without any support and/or preconfigured parts it is…

- manage nodes in line with resource
- HA mode
- management features
- minikube

8 Automation

Make repetitive work fun again!

In recent years, declarative approaches helped to make task automation more inclusive and appealing to a wider audience. Not only a workflow itself but also the environment a workflow should live and run in is often defined in declarative fashion. This development does not only make maintenance easier, it also helps to make a setting reproducible and shareable.

8.1 Infrastructure as Code

Infrastructure as code approaches do not only describe infrastructure in declarative and reproducible fashion as stated above, infrastructure as code can also be seen as a way to automate setting up the environment you work with.

DOCKERFILE definitions as described in the previous chapter are a recipe to build docker images using some build environment, e.g., a local docker runtime environment such as Docker Desktop or a build tool like drone. Just like many automation tools, docker has a console based client interface (CLI) to communicate with the user efficiently. A simple docker command issued in the directory that contains the DOCKERFILE triggers the build of an image.

```
docker build .
```

Usually the next step is to push the resulting image into a public or private registry. This step as well as the sequence of commands, often referred to as pipeline oder build chain, can be automated, too. It is also possible to build an application that uses multiple single purpose containers, e.g., an online survey that stores its results in database. In such a case one container could serve a web frontend using node while another PostgreSQL container provides a database backend. Without going immediately going for a cluster, docker compose can bundle multiple containers and deploy them jointly so they can smoothly interact with each other. Shell scripts can help to automate simpler processes and put several steps after one another, but if your can invest a bit more time, I highly recommend to use the more powerful, Red Hat backed ansible automation tool. Ansible also comes with a client interface, defines playbooks and role using .yaml files in declarative, human-read-friendly fashion.

```yaml
some:
    yaml: "example"
```

Automation is more complex for cluster setups, because among other things, applications need to be robust against pods getting shut down on one node and spawned on another node allowing to host applications in *high availability* mode. On Kubernetes clusters, helm, Kubernetes' package manager, is part of the solution to tackle this added complexity. Helm is "the best way to find share and use software built for Kubernetes". Terraform can manage Kubernetes clusters on clouds like Amazon's AWS, Google's GPC, Microsoft's Azure and many other cloud systems using declarative configuration files. Again, a console based client interface is used to apply a declarative confiugration plan to a particular cloud.

8.2 CI/CD

The first type of automation described here refers to automation of your development workflow. That it is, you standardize your path from draft to program to deployment to production. Modern version control software accompanies this process with a toolchain that is often fuzzily called CI/CD. While CI stands for *continuous integration* and simply refers to a workflow in which the team tries to release new features to production as continuously as possible, CD stands for either *continuous delivery* or *continuous deployment*.

However, in practice the entire toolchain referred to as *CI/CD* has become broadly available in well documented fashion when git hosting powerhouses GitHub and GitLab introduced their flavors of it: [GitHub Actions])(https://docs.github.com/en/actions) and GitLab CI. In addition services like Travis CI or Circle CI offer this toolchain independently of hosting git repositories.

Users of these platforms can upload a simple textfile that follows a name convention and structure to trigger a step based toolchain based on an event. An example of an event may be the push to a repository's main branch. A common example would be to run tests and/or build a package and upon success deploy the newly created package to some server – all triggered by simple push to master. One particularly cool thing is, that there multiple services who allow to run the testing on their servers using container technologies. This leads to great variety of setups for testing. That way software can easily be tested on different operating systems / environments. Also the mentioned website rendering approach mentioning in the previous section as a potential CI/CD application.

Here is a simple example of a *.gitlab-ci.yml* configuration that builds and tests a package and deploys it. It's triggered on push to master:

```
stages:
```

```
- buildncheck
- deploy_pack

test:
image:
name: some.docker.registry.com/some-image:0.2.0
entrypoint:
- ""
stage: buildncheck
artifacts:
untracked: true
script:
- rm .gitlab-ci.yml # we don't need it and it causes a hidden file NOTE
- install2.r --repos custom.mini.cran.ch .
- R CMD build . --no-build-vignettes --no-manual
- R CMD check --no-manual *.tar.gz

deploy_pack:
only:
- master
stage: deploy_pack
image:
name: byrnedo/alpine-curl
entrypoint: [""]
dependencies:
- 'test'
script:
- do some more steps to login and deploy to server ...
```

For more in depth examples of the above, Jim Hester's talk on GitHub Actions for R is a very good starting point.

The other automation tool I would like to mention is Apache Airflow because of its ability to help researchers keep an overview of regularly

running processes. Examples of such processes could be daily or monthly data sourcing or timely publication of a regularly published indicator. I often referred to it as cronjobs on steroids. Airflow ships with a dashboard to keep track of many timed processes, plus a ton of other log and reporting features worth a lot when maintaining reocurring processes.

8.3 Workflow Scheduling: Apache Airflow

- cronjobs
- MWAA

8.4 Make, target & co.

task automation, caching.. monitoring etc.

9 Community

Besides the fact that it is free of license costs most of the time, open source communities may be the most popular argument in favor of open source software. In academia, free quality online resources and support can be extremely helpful, particularly when one's curriculum did not contain an applied form of programming with data.

Still, developer communities do have their idiosyncrasies. This section intends to help overcome entry barriers and encourage the reader to connect to the developer communities – even if one is not a regular visitor of community offers.

9.1 Stay up to Date and a Vastly Evolving Field – Social Media

I hate to admit it, because I am not a big fan of social media and don't use social media apart from professional use, I am convinced it is a great way to get my regular dose of what's new in tech. My platform of choice is twitter. I use it as a bookmark tool and get some feedback from publicly sharing resources. Plus I follow some folks to get updates from different the fields. I try to not get caught in any politics, memes or cat pictures, even if that's a hard thing to do.

CHEWBACCA cat.

Start with a few accounts to follow, adapt them regularly and use lists in case you need to organize your input a bit more. tweetdeck is a good standard option of a more advanced use of twitter.

If you plan to build some following of your own, learn how to schedule tweets and put some thought into the timing of your tweets. However,

Globally different preferences.

CITE useR! nature correspondence

9.2 Get an Account at Stackoverflow.com

9.3 Attend Conferences - Online Can Be a Good Option !

9.4 Join Slack Spaces (or other Chats)

Slack or Discord

9.5 Look for Local Community Group

job market!

10 Publishing & Reporting

Business analysts are used to report to management and/or clients on a regular basis. Academia has a lively debate about science communication. So what does a researcher need to know about publishing and reporting in order to make right choices? The key term about modern reporting in both cases is *regular*. Data might get revised or otherwise updated, journals, reviewers and readers will ask for reproducible research. Dissemination of datasets and results in visually appealing as well as machine readable communication are equally important in modern publishing and reporting.

Because of frequent updates and reproducibility, manually composed outlets become less sustainable. Luckily, markdown based reporting offers great solutions for presentations, websites, reports, blogs, office documents or even printed output. Document converters can turn markdown into HTML/CSS/Javascript output for flexible use in a standard web browser. Likewise, markdown renders to PDF or even Microsoft Word.

Approaches such as Quarto, RMarkdown or Jupyter Notebooks mix plain text with dynamically rendered code chunks that create tables or figures and embedd these into the document in the right size and spot. Such a setup is the basis to fully automate the way from analysis to report no matter whether the outlet is printed, a Word document, a presentation, blog or website.

10.1 Static Website Generators

Let's limit our look at content renderers to outlets for the web for now. We do so to see how the result of a rendering process goes online in fully automated fashion. Unlike in the case of printed output, processing of web output is very standardized: web browser can display content from the HTML/CSS/Javascript combination.

The distribution follows in line after chain and automated as well.

10.2 Hosting Static Website

netlify, ghpages, anywhere, a note on self-contained

10.3 Visualization Libraries

- apache echarts
  - highcharts
  - dygraphs (very specific)

10.4 Data Publications

- visualization libraries
- web applications
- think about the channel
- visualization libs
- data publications (opendata, link to data management chapter)

- conduct surveys (DGPs)
- static websites GitHub Pages & Netlify (maybe case study)
- quarto (importance of a decent preview)

11 Case Studies

While the rest of the book provided more of a big picture type of insight, this section is all about application minded examples that most of the time feature code to reproduce.

11.1 RSA Key Pair Authentication

This section could be headed 'log in like a developer'. RSA Key Pairs are a convenient, relatively secure way to log into an account. SSH based connections, including secure copy (SCP), often make use of RSA Key Pairs instead of using a combination of username and password. Also, most git platforms use this form of authentication. The basic idea of key pairs is to have a public key and a private key. While the private key is never shared with anyone, the public key is shared with any server you want to log in to. It's like getting a custom door for any house that you are allowed to enter: share your public key with the server admin / or web portal and you'll be allowed in when you show your private key. In case you loose your private key or suspect it has been stolen, simply inform the admin, so she can remove the door (the public key). This is were a little detail comes into play: you can password protect the authentication process. Doing so buys you time to remove the key from the server before your password gets bruteforced. The downside of this additional password is its need for interaction. So when you are setting up a batch that talks to a remote server that is when you do *not* want a key /w password.

Step one en route to log in like a grown up, is the create an RSA key pair. Github has a 1-2-3 type of manual to get it done. Nevertheless, I would like the R Studio (Server) specific way here.

1. Login to https://teaching.kof.ethz.ch/ (or use your local R Studio Desktop)
2. Go to *Tools -> Global Options -> Git/SVN*
3. Hit Create RSA KEY (When you some crazy ASCII art reminiscant of a rabbit, it's just ok.)
4. Click 'View Public Key'
5. Copy this key to the your clipboard.

The R Studio GUI is an easy way to create an RSA Key Pair.

6. You can paste the key you obtained to your Github settings or put it into your server's authorized keys file.

Congrats you may log in now!

11.2 Consuming APIs

An Application Programming Interface (API) is nothing else but an interface to facilitate machine to machine communication. An interface can be anything, any protocol or pre-defined process. But of course there are standard and not-so-standard ways to communicate. Plus some matter-of-taste type of decisions. But security and standard compliance are none of the latter. There are standards such as the popular, URL based REST that make developers' lives a lot easier – regardless of the language they prefer.

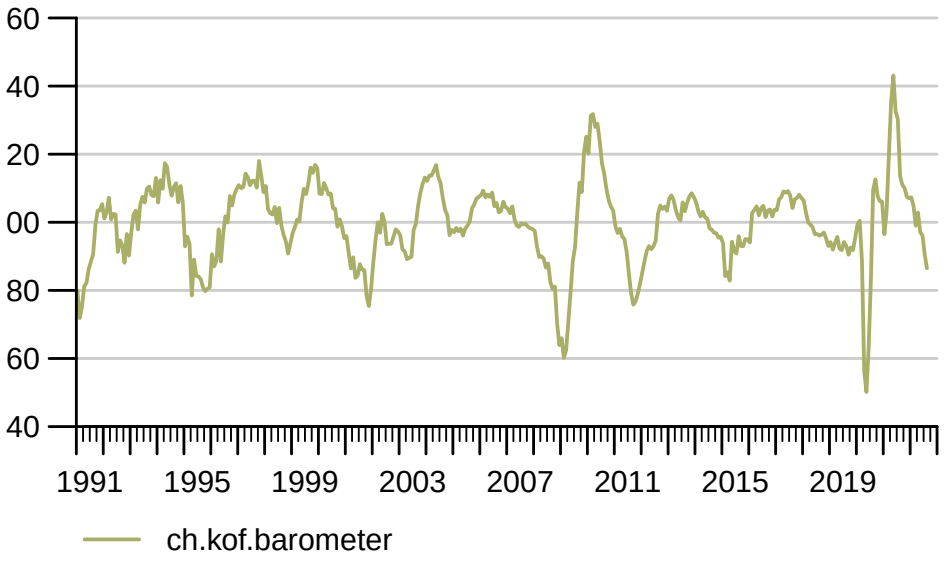
Many services such as Google Cloud, AWS, your university library, your favorite social media platform or your local metro operator provide an API. Often either the platform itself or the community provide what's called an API wrapper: A simple program wraps the process of using the

interface though dynamic URLs into a parameterized function. Because the hard work is done serverside by the API backend, building API wrappers is fairly easy and if you're lucky wrappers for your favorite languages exit already. If that is the case end users can simply use functions like `get_dataset(dataset_id)` to download data programatically.

11.2.1 Example 1: The {kofdata} R package

The KOF Swiss Economic Institute at ETH Zurich provides such a wrapper in an R package. The underlying API allows to access the KOF time series archive database and obtain data and meta information alike. The below code snippet gets data from the API and uses another KOF built library ({tstools}) to visualize the returned time series.

```
library(kofdata)
# just for viz
library(tstools)
tsl <- get_time_series("ch.kof.barometer")
tsplot(tsl)
```



11.2.2 Example 2: The {OECD} R package

Also large organizations like the Organization for Economic Co-operation and development (OECD) provide API wrappers to facilitate data consumption.

11.2.3 Build Your Own API Wrapper

Here's an example of a very simple API wrapper that makes use of the Metropolitan Museum of Modern Art's API to obtain identifiers of pictures based on a simple search.

```
# Visit this example query
# https://collectionapi.metmuseum.org/public/collection/v1/search?q=umbrel
# returns a json containing quite a few ids of pictures that were tagged '
```

```
#' Search MET
#'
#' This function searches the MET's archive for keywords and
#' returns object ids of search hits. It is a simple wrapper
#' around the MET's Application Programming interface (API).
#' The function is designed to work with other API wrappers
#' and use object ids as an input.
#' @param character search term
#' @return list containing the totoal number of objects found
#' and a vector of object ids.
#'
# Note these declaration are not relevant when code is not
# part of a package, hence you need to call library(jsonlite)
# in order to make this function work if you are not building
# a package.
#' @examples
#' search_met("umbrella")
#' @importFrom jsonlite formJSON
#' @export
search_met <- function(keyword){
  # note how URLencode improves this function
  # because spaces are common in searches
  # but are not allowed in URLs
  url <- sprintf("https://collectionapi.metmuseum.org/public/collection/v1/search?q=%s",
  fromJSON(url)
}
```

You can use these ids with another endpoint in order to receive the pictures themselves.

```
download_met_images_by_id <- function(ids,
                                      download = "primaryImage") {
```

```
  # Obtain meta description objects from MET API
  obj_list <- lapply(ids, function(x) {
    req <- download.file(sprintf("https://collectionapi.metmuseum.org/publ
                  x),destfile = "temp.json")
    fromJSON("temp.json")
  })

  # Extract the list elements that contains
  # img URLs in order to pass it to the download function
  img_urls <- lapply(obj_list, "[[", download)
  # Note the implicit return, no return statement needed
  # last un-assigned statement is returned from the function
  lapply(seq_along(img_urls), function(x) {
    download.file(img_urls[[x]],
                  destfile = sprintf("data/image_%d.jpg", x)
    )
  })
}


# Step 4: Use the Wrapper
umbrella_ids <- search_met("umbrella")
umbrella_ids
download_met_images_by_id(umbrella_ids$objectIDs[2:4])
```

11.3 Create Your Own API

Being able to expose data is a go-to skill in order to make research reproducible and credible. Especially when data get complex and require thorough description in order to remain reproducible for others, a programmtic, machine readable approach is the way to go.

11.3.1 GitHub to Serve Static Files

Exposing your data through an API is not something you would need a software engineer or an own server infrastructure for. Simply hosting a bunch of .csv spreadsheet alongside a good description (in separate files!!) on, e.g., GitHub for free can be an easy highly available solution to serve static files.

The KOF High Frequency Economic Monitoring dashboard simply shares standardized .csv (data) and .json (description) files based on a Github.

INSERT SCREENSHOT OF GITHUB HERE

To make it look at little niftier, the dashboard uses a quasar frontend to guide the human user, but it would not be necessary to have such a framework.

INSERT SCREENSHOT OF KOFDATA HERE

11.3.2 Simple Dynamic APIs

Even going past serving static files, does not require much software development expertise. Thanks to frameworks such as express.js or the {plumbr} it easy to create an API that turns a URL into a server side action and returns a result.

Assume you've installed node.js already, you can set up a simple API on your local computer just like this.

```
# run initialization in a dedicated folder
mkdir api
cd api
npm init
```

just sleep walk through the interactive dialog accepting all defaults. Once done, add install express using the npm package manager.

```
npm install express --save
```

11.4 A Minimal Webscraper: Extracting Publication Dates

Even though KOF Swiss Economic Institute offers a REST API to consume publicly available data, publication dates are unfortunately not available through in API just yet. Hence, in order to automate data consumption based on varying publication dates, we need to extract upcoming publication dates of the Barometer from KOF's media release calendar. Fortunately all future releases are presented online an easy-to-scrape table. So here's the plan:

1. Use Google Chrome's *inspect element* developer feature to find the X-Path (location in the Document Object Model) of the table.
2. Read the web page into R using `rvest`.
3. Copy the X-Path string to R to turn the table into a data.frame
4. use a regular expression to filter the description for what we need.

Let's take a look at our starting point, the media releases sub page, first.

The website looks fairly simple and the jackpot is not hard, presented in a table right in front of us. Can't you smell the data.frame already?

Right click the table to see a Chrome context window pop up. Select *inspect*.

Hover over the blue line in the source code at the bottom. Make sure the selected line marks the table. Right click again, select copy -> copy X-Path.

On to R!

```
library(rvest)
# URL of the media release subsite
url <- "https://kof.ethz.ch/news-und-veranstaltungen/medien/medienagenda.html"
# Extract the DOM object from the path we've previously detected using
# Chrome's inspect feature
table_list <- url %>%
  read_html() %>%
  html_nodes(xpath = '//html/body/div[6]/section/div/section/div[2]/div/div[3]/div/div/di
  # turn the HTML table into an R data.frame
  html_table()
# because the above result may potentially contain multiple tables, we just use
# the first table. We know from visual inspection of the site that this is the
# right table.
agenda_table <- table_list[[1]]

//*[@id="t-6b2da0b4-cec0-47a4-bf9d-bbfa5338fec8-tbody-row-2-cell-2"]

# extract KOF barometer lines
pub_date <- agenda_table[grep("barometer",agenda_table$X3),]
pub_date
```

Yay! We got everything we wanted. Ready to process.

11.5 Automate Script Execution: A GitHub Actions Example

11.6 Choropleth Map: Link Data to a geojson Map File

Data visualization is a big reason for researchers and data analysts to look into programming languages. Programming languages do not only provide unparalleled flexibility, they also make data visualization reproducible and allow to place charts in different contexts, e.g., websites, printed outlets or social media.

One of the more popular type of plots that can be created smoothly using a programing language is the so called *choropleth*. A *choropleth* maps values of a variable that is available by region to a given continuous color palette on a map. Let's break down the ingredients of the below map of Switzerland.

First, we need a *definition of a country's shape*. Those definitions come in various format from traditional *shape files* to web friendly *geoJSON*. Edzer Pebesma's useR! 2021 keynote has a more thorough insight.

Second, we need some data.frame that simply connects values to regions. In this case we use regions defined by the Swiss Federal Statistical Office (FSO). Because our charting library makes use of the geoJSON convention to call the region label 'name' we need to call the column that holds the region names 'name' as well. That way we can safely use defaults when plotting. Ah, and note that the values are absolutely bogus that came to my mind while writing this (so please do not mull over how these values were picked).

```
d <- data.frame(
  name = c("Zürich",
           "Ticino",
```

```
            "Zentralschweiz",
            "Nordwestschweiz",
            "Espace Mittelland",
            "Région lémanique",
            "Ostschweiz"),
  values = c(50,10,100,50,23,100,120)
)
```

Last but not least where calling our charting function from the *echarts4r* package. *echarts4r* is an R wrapper for the feature rich Apache Echarts Javascript plotting library. The example uses the base R pipes (available from 4+ on, former versions needed to use pipes via extension packages.). Pipes take the result of one previous line and feed it as input into the next line. So the data.frame *d* is linked to a charts instance and the *name* column is used as the link. Then a map is registered as *CH* and previously read json content is used to describe the shape.

```
d |>
  e_charts(name) |>
  e_map_register("CH", json_ch) |>
  e_map(serie = values, map = "CH") |>
  e_visual_map(values,
               inRange = list(color = viridis(3)))
```

Also note the use of the viridis functions which returns 3 values from the famous, colorblind friendly viridis color palette.

```
viridis(3)
```

```
[1] "#440154FF" "#21908CFF" "#FDE725FF"
```

Here's the full example:

```
library(echarts4r)
library(viridisLite)
library(jsonlite)

json_ch <- jsonlite::read_json(
  "https://raw.githubusercontent.com/mbannert/maps/master/ch_bfs_regions.geojson"
)



d <- data.frame(
  name = c("Zürich",
           "Ticino",
           "Zentralschweiz",
           "Nordwestschweiz",
           "Espace Mittelland",
           "Région lémanique",
           "Ostschweiz"),
  values = c(50,10,100,50,23,100,120)
)

d |>
  e_charts(name) |>
  e_map_register("CH", json_ch) |>
  e_map(serie = values, map = "CH") |>
  e_visual_map(values,
               inRange = list(color = viridis(3)))
```

11.7 Web Applications /w R Shiny

For starters, let me de-mistify {shiny}. There are basically two reasons why so many inside data science and analytics have shiny on their bucket list of things to learn. First, it gives researchers and analysts home court advantage on a webserver. Second, it gives our online appearances a kick-start in the dressing room.

Don't be surprised though if your web development professional friend outside data science and analytics never heard of it. Compared to web frontend framework juggernauts such as *react*, *angular* or *vue.js* the shiny web application framework for R is rather a niche ecosystem.

Inside the data science and analytics communities, fancy dashboards and the promise of an easy, low hurdle way to create nifty interactive visualizations have made {shiny} app development a sought after skill. Thanks to pioneers, developers and teachers like Dean Attali, John Coene, David Granjon, Colin Fay and Hadley Wickham, the sky seems the limit for R shiny applications nowadays.

This case study does not intend to rewrite {shiny}'s great documentation or blogs and books around it. I'd rather intend to help you get your first app running asap and explain a few basics along the way.

11.7.1 The Web Frontend

Stats & figures put together by academic researchers or business analysts are not used to spend a lot of time in front of the mirror. (Often for the same reason as their creators: perceived opportunity costs.)

Shiny bundles years worth of lime light experience and online catwalk professionalism into an R package. Doing so allows us to use all this design expertise through an R interface abstracting away the need to dig deep into web programming and frontend design (you know the HTML/CSS/Javascript).

Let's consider the following web frontend put together with a few lines of R code. Consider the following, simple web fronted that lives in a dedicated user interface R file, called *ui.R*:

```
library(shiny)
library(shinydashboard)

dashboardPage(
  dashboardHeader(title = "Empty App"),
  dashboardSidebar(),
  dashboardBody(
    fluidPage(
      fluidRow(
        box(title = "Configuration",
            sliderInput("nobs",
                        "Number of Observations",
                        min = 100,
                        max = 10000,
                        value = 500),
            sliderInput("mean_in","Mean",
                        min = 0,
                        max = 10,
                        value = 0),
            sliderInput("sd_in","SD",
                        min = 1,
                        max = 5,
                        value = 1),
            width = 4),
        box(title = "Distribution",
            plotOutput("dist"),
            width = 8)
      ),
      fluidRow(
```

```
        box("Tabelle",
            dataTableOutput("tab"),
            width=12
            )
      )
    )
  )
)
```

Besides the shiny package itself, the ecosystem around shiny brings popular frontend frameworks from the world outside data science to R. In the above case, a boilerplate library for dashboards is made available through the add-on package {shinydashboard}.

Take a moment to consider what we get readily available at our finger tips: Pleasant user experience (UX) comes from many things. Fonts, readability, the ability to adapt to different screens and devices (responsiveness), a clean, intuitive design and many other aspects. {shinydashboard} adds components like *fluidPages* or *fluidRow* to implement a responsive (google me), grid based design using R. Note also how similar the hierarchical, nested structure of the above code is to HTML tagging. (Here's some unrelated minimal HTML)

```
<!-- < > denotes an opening, </ > denotes an end tag. -->
<html>
  <body>
  <!-- anything in between tags is affected by the tags formatting.
       In this case bold -->
    <b> some bold title </b>
    <p>some text</p>
  </body>
</html>
```

{shiny} ships with many widgets[1] such as input sliders or table outputs that can simply be placed somewhere on your site. Again, add-on packages provide more widgets and components beyond those that ship with shiny.

11.7.2 Backend

While the frontend is mostly busy looking good, the backend has to do all the hard work, the computing, the querying – whatever is processed in the background based on user input.

Under-the-hood-work that is traditionally implemented in languages like Java, Python or PHP[2] can now be done in R. This is not only convenient for the R developer who does not need to learn Java, it's also incredibly comfy if you got data work to do. Or put differently: who would like to implement logistic regression, random forests or principal component analysis in PHP?

Consider the following minimal backend *server.R* file which corresponds to the above *ui.R* frontend file. The anonymous (nameless) function which is passed on to the *ShinyServer* function takes two named lists, *input* and *output*, as arguments. The named elements of the input list correspond to the *widgetId* parameter of the UI element. In the below example, our well known base R function *rnorm* takes *nobs* from the input as its *n* sample size argument. Mean and standard deviation are set in the same fashion using the user interface (UI) inputs.

[1] online widget galleries like R Studio's shiny widget gallery that help to 'shop' for the right widgets.

[2] Yup, there is node and javascript on web servers, too, but let's keep things simple and label javascript clientside here. And yes, shiny server used to require node itself, too.

```
library(shiny)

shinyServer(function(input,output){

  output$dist <- renderPlot({
    hist(
      rnorm(input$nobs,
            mean = input$mean_in,
            sd = input$sd_in),
      main = "",
      xlab = ""
      )
  })

  output$tab <- renderDataTable({
    mtcars
  })



})
```

The vector that is returned from *rnorm* is passed on to the base R *hist* which returns a histogram plot. This plot is then rendered and stored into an element of the *output* list. The *dist* name is arbitrary but again matched to the UI. The *plotOutput* function of *ui.R* puts the rendered plot onto the canvas so it's on display in people's browsers. *renderDataTable* does so in analog fashion to render and display the data table.

11.7.3 Put Things Together and Run Your App

The basic app shown above consists of a *ui.R* and a *server.R* file living in the same folder. The most straight forward way to run such an app is to call the *runApp()* function and provide the location of the folder that contains both of the aforementioned files.

```
library(shiny)
runApp("folder/that/holds/ui_n_server")
```

This will use your machine's built-in web server and run shiny locally on your notebook or desktop computer. Even if you never put your app on a web server and run a website with it this is already a legitimate way to distribute your app. If it was part of an R package, everyone who download your package could use it locally, maybe as a visual inspection tool or way to derive inputs interactively and feed them into your R calculation.

11.7.4 Serve your App

Truth be told, the full hype and excitement of a shiny app only comes into play when you publish your app and make it available to anyone with a browser, not just the R people. Though hosting is a challenge in itself let me provide you a quick shiny specific discussion here. The most popular options to host a shiny app are

- **software-as-a-service (SaaS)**. No maintenance, hassle free, but least bang-for-buck. The fastest way to hit the ground running is R Studio's service *shinyapps.io*.
- **on premise aka in house**. Either download the open source version of shiny server, the alternative shiny proxy or the R Studio Connect premium solution and install them on your own Virtual machine.

- use a **shiny server docker image** and run a container in your preferred environment.

11.7.5 Shiny Resources

One of the cool things of learning shiny is how shiny and its ecosystem allow you to learn quickly. Here are some of my favorite resources to hone your shiny skills.

- R Studio Shiny's Widget Gallery
- shinydashboard
- Mastering Shiny by Hadley Wickham
- Engineering Production Grade Shiny Apps
- RInterface by David Granjon, John Coene, Victor Perrier and Isabelle Rudolf

11.8 Project Management Basics

The art of stress free productivity as I once called it in '10 blog post, has put a number of gurus on the map and whole strand of literature to our bookshelves. So rather than adding to that, I would like to extract a healthy, best-of-breed type of dose here. The following few paragraphs do not intend to be comprehensive – not even for the scope of software projects, but inspirational.

In tje software development startup community, the *waterfall* approach became synonym to conservative, traditional and ancient: Overspecification in advance of the project, premature optimization and a lawsuit over expectations that weren't met. Though waterfall project may be better

than their reputation and specifications should not be too detailed and rigid.

Many software projects are rather organized in *agile* fashion with SCRUM and KANBAN being the most popular derivatives. Because empirical academic projects have a lot in common with software projects inasmuch that there is a certain expectation and quality control, but the outcome is not known in advance. Essentially in agile project management you roughly define an outcome along the lines of a minimum viable product (MVP). That way you do not end up with nothing after a year of back and forth. During the implementation you'd meet regularly, let's say every 10 days, to discuss development since the last meet and what short term plans for the next steps ahead. The team picks splits work into task items on the issue tracker and assigns them. Solution to problems will only be sketched out and discussed bilaterally or in small groups. By defining the work package for only a short timespan, the team stays flexible. In professional setups agile development is often strictly implemented and makes use of sophisticated systems of roles that developers and project managers can get certified for.

Major git platforms ship with a decent, carpentry level project management project management GUI. The issue tracker is at the core of this. If you use it the minimal way, it's simply a colorful to-do list. Yet, with a bit of inspiration and use of tags, comments and projects, an issue tracker can be a lot more

The Github issue tracker (example from one of the course's repositories) can be a lot more than a todo list.

Swimlanes (reminiscant of a bird's eye view of an Olympic pool) can be thought of columns that you have to walk through from left to right: To Do, Doing, Under Review, Done. (you can also customize the number and label of lanes and event associate actions with them, but let's stick to those basic lanes in this section.) The idea is to use to keep track of the process and making the process transparent.

GitHub's web platform offers swimlanes to keep a better overview of issues being worked on.

"'{, type='note'} **Tipp**: No lane except 'Done' should contain more than 5-6 issues. Doing so prevents clogging the lanes at particular stage which could potentially lead to negligent behavior, e.g., careless reviews.

```
<!--
## Articles, Presentations, Reports and Websites /w Markdown and RMarkdown

Working with data asks for reproducible reports, presentations and articles t
This case study explains how to create documents like this book, a presentati

*Markdown* is simple language based approach that can handle all of these tas
*Markdown* helps users to format plain text using reserved characters. For ex

Various rendering engines can render *Markdown* into HTML, PDF or even MS Wor

- [*Markdown* source of this particular site that you are reading here]().
- [*Markdown* source of a presentation slides]()




## Spatial Visualization with Leaflet and Open Streetmap



## Basic Parallel Programming

Here's an R example using the {microbenchmark} package to check the effect pa
```

```
monthly time series about airline passengers from 1949-1960.



::: {.cell}

```{.r .cell-code}
data("AirPassengers")
tsl <- list()
for(i in 1:1000){
  tsl[[i]] <- AirPassengers
}
```

:::

Now, let's load the {seasonal} package and perform a basic seasonal adjustment of each of these time series. The first statement performs 1000 adjustments sequentially, the second statement uses parallel computing to spread computations across my machines multiple processors. The key take away from this exercise is not the parallel computation itself, but the ability to set up a benchmark.

```
  library(seasonal)
  library(microbenchmark)
```

Obviously the absolute computation time depends on the hardware used, but also the operating system can be an important factor depending on the task at hand. Even though the combination of algorithm, hardware, operating system and software used for the computation can make assessment a daunting, complex task, easily venturing into expert realm, a basic understanding and a ballpark relation between multiple approaches can carry you a long way.

–>
```

Appendix

Glossary

Term	Description
API	**A**pplication **P**rogramming **I**nterface
CamelCase	Convention to spell file, variable or function names reminiscant of a camel, e.g., doS
CMS	**C**ontent **M**anagement **S**ystem
Console	Also known as terminal, the console is an interface which takes written user comma
Deployment	The art of delivering a piece software to production
Endpoint	Part of an API, a generic URL that follows a systematic that can be exploited to au
Fork	a clone of a repository that you (usually) do not own.
GUI	**G**raphical **U**ser **I**nterface
IDE	**I**ntegrated **D**evelopment **E**nvironment
Kebap Case	Spelling convention less known than snake case and camel case, kebap case looks lik
Lexical Scoping	Look up of variables in parent environments when they can't be found in the curren
Merge Request	See Pull Request.
OS	**O**perating **S**ysten
OSS	**O**pen **S**ource **S**oftware
Pull Request (PR)	Request to join a feature branch into another branch, e.g., main branch. Sometimes
Regular Expression	Pattern to extract specific parts from a text, find stuff in a text.
REPL	[read-eval-print-loop](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%
Reproducible Example	A self-contained code example, including the data it needs to run.
Snake_case	Convention to spell file, variable or function names reminiscant of a snake, e.g., do_
Stack	selection of software used in a project
SQL	**S**tructured **Q**uery **L**anguage

Term	Description
Swimlanes	(Online) Board of columns (lanes). Lanes progress from from left
Throughput Problem	A bottleneck which can be mitigated by parallelization, e.g., mult
Transactional Database	Database optimised for production systems. Such a database is g
Virtual Machine (VM)	A virtual computer hosted on your computer. Often used to run

References