Introduction to Data Science

Session 10: Debugging, automation, and packaging

Simon Munzert
Hertie School | GRAD-C11/E1339

Table of contents

Strategies for debugging

What's debugging?

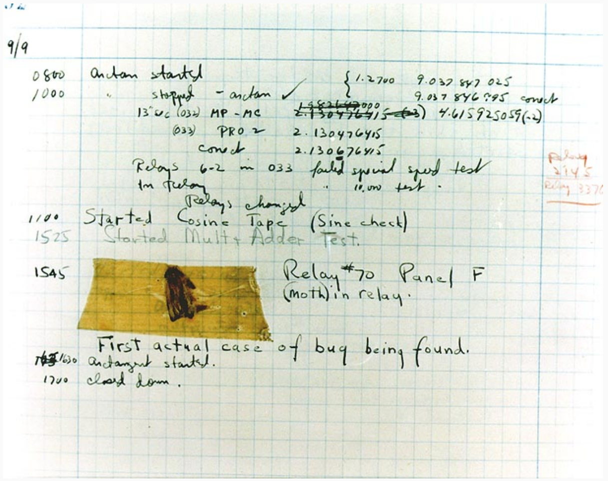
Straight from the Wikipedia

"Debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems."

A famous (yet not the first) bug:

The term "bug" was used in an account by computer pioneer Grace Hopper (see on the right). While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. This bug was carefully removed and taped to the log book (see on the right).

Above: Grace Hopper, *Below:* The bug



Why debugging matters

The Wikipedia list of software bugs with significant consequences is growing and you don't want to be on it.

NASA software engineers are famous for producing bug-free code. This was learned the hard and costly way though. Some highlights from space:

- 1962: A booster went off course during launch, resulting in the destruction of NASA Mariner 1 . This was the result of the failure of a transcriber to notice an overbar in a handwritten specification for the guidance program, resulting in an incorrect formula the FORTRAN code.
- 1999: NASA's Mars Climate Orbiter was destroyed, due to software on the ground generating commands based on parameters in pound-force (lbf) rather than newtons (N)
- 2004: NASA's Spirit rover became unresponsive on January 21, 2004, a few weeks after landing on Mars. Engineers found that too many files had accumulated in the rover's flash memory (the problem could be fixed though by deleting unnecessary files, and the Rover lived happily ever after. Until it froze to death in 2011).

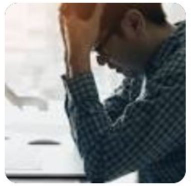
Why debugging matters (cont.)

ZDNet

Microsoft Excel blunder: Developers blamed for loss of thousands of COVID-19 test results

The error has hampered the UK's contact-tracing program at a time when the country is undergoing a second wave of coronavirus infections. By using the XLS ...

1 month ago



ABC News

Excel glitch leads to nearly 16,000 confirmed coronavirus cases going unreported in United Kingdom

For the test and trace program to work well, contacts should be notified as soon as possible. Health Secretary Matt Hancock told MPs that the problem related to ...

1 month ago

Metro

How were 16,000 Test and Trace coronavirus cases lost on Excel?

The cases were lost due to a technical error on a Microsoft Excel spreadsheet. ... Trace 'immediately' after the issue was resolved and thanked contact tracers for ...

1 month ago

Does Contact Tracing Work? Quasi-Experimental Evidence from an Excel Error in England*

Thiemo Fetzer† Thomas Graeber‡

November 24, 2020

Abstract

Contact tracing has been a central pillar of the public health response to the COVID-19 pandemic. Yet, contact tracing measures face substantive challenges in practice and well-identified evidence about their effectiveness remains scarce. This paper exploits quasi-random variation in COVID-19 contact tracing. Between September 25 and October 2, 2020, a total of 15,841 COVID-19 cases in England (around 15 to 20% of all cases) were not immediately referred to the contact tracing system due to a data processing error. Case information was truncated from an Excel spreadsheet after the row limit had been reached, which was discovered on October 3. There is substantial variation in the degree to which different parts of England areas were exposed – by chance – to delayed referrals of COVID-19 cases to to the contact tracing system. We show that more affected areas subsequently experienced a drastic rise in new COVID-19 infections and deaths alongside an increase in the positivity rate and the number of test performed, as well as a decline in the performance of the contact tracing system. Conservative estimates suggest that the failure of timely contact tracing due to the data glitch is associated with more than 125,000 additional infections and over 1,500 additional COVID-19-related deaths. Our findings provide strong quasi-experimental evidence for the effectiveness of contact tracing.

Keywords: Health, Coronavirus
JEL Classification: I31, Z18

Why debugging matters (cont.)

Technology

Facebook made big mistake in data it provided to researchers, undermining academic work

Company accidentally left out half of all of its U.S. users in providing data to a research consortium

The error resulted from Facebook accidentally excluding data from U.S. users who had no detectable political leanings — a group that amounted to roughly half of all of Facebook's users in the United States. Data from users in other countries was not affected.

"It's data. Of course, there are errors," said Gary King, a Harvard professor who co-chairs Social Science One. "This, of course, was a big error."

King, director of the university's Institute for Quantitative Social Science, said dozens of papers from researchers affiliated with Social Science One had relied on the data since Facebook shared the flawed set in February 2020, but he said the impact could be determined only after Facebook provided corrected data that could be reanalyzed. He said some of the errors may cause little or no problems, but others could be serious.

Social Science One shared the flawed data with at least 110 researchers, King said.

An Italian researcher, Fabio Giglietto, discovered data anomalies last month and brought them to Facebook's attention. The company contacted researchers in recent days with news that they had failed to include roughly half of its U.S. users — a group that likely is less politically polarized than Facebook's overall user base. The New York Times first reported Facebook's error.

Source Washington Post

Sol Messing @SolomonMg · Sep 11

What happened that generated the error: TBD. I'd bet that U.S. user-political affinity was joined to the rest of the data using a LEFT JOIN instead of a LEFT OUTER JOIN. Again FB folks are likely working to fix this ASAP.

3 10 35

Sol Messing @SolomonMg · Sep 11

What was the likely consequence: people in the U.S. with no interest in political information were excluded. Substantively this would make FB look more hyper-partisan, as per @deaneckles' tweet here:

> **Dean Eckles** @deaneckles · Sep 11
>
> That is, contra some reactions that somehow this error "helped" Facebook, I would expect this made FB look more filled with misinfo & polarizing content than it was.
>
> Obviously, this error will have lasting consequences… twitter.com/daveyalba/stat…
>
> Show this thread

1 8 31

Sol Messing @SolomonMg · Sep 11

What are the broader systematic issues in play here: researchers didn't have access to the raw data or pipeline code. That's a huge deal and makes it nearly impossible to do the usual, focused deep dive data forensics that research often entails.

Source Solomon Messing / Twitter

A general strategy for debugging

1. Google
2. Reset
3. Debug
4. Deter

Google

According to this analysis, the most common error types in R are:[1]

1. `Could not find function` errors, usually caused by typos or not loading a required package.
2. `Error in if` errors, caused by non-logical data or missing values passed to R's `if` conditional statement.
3. `Error in eval` errors, caused by references to objects that don't exist.
4. `Cannot open` errors, caused by attempts to read a file that doesn't exist or can't be accessed.
5. `no applicable method` errors, caused by using an object-oriented function on a data type it doesn't support.
6. `subscript out of bounds` errors, caused by trying to access an element or dimension that doesn't exist
7. Package errors caused by being unable to install, compile or load a package.

[1]Do you get an error message you don't understand? That's good news actually, because the really nasty bugs come without errors.

Google

According to this analysis, the most common error types in R are:[1]

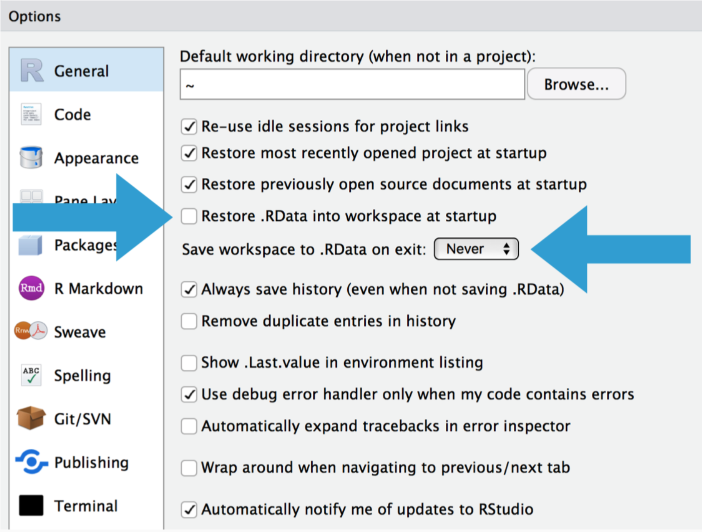
1. `Could not find function` errors, usually caused by typos or not loading a required package.
2. `Error in if` errors, caused by non-logical data or missing values passed to R's `if` conditional statement.
3. `Error in eval` errors, caused by references to objects that don't exist.
4. `Cannot open` errors, caused by attempts to read a file that doesn't exist or can't be accessed.
5. `no applicable method` errors, caused by using an object-oriented function on a data type it doesn't support.
6. `subscript out of bounds` errors, caused by trying to access an element or dimension that doesn't exist
7. Package errors caused by being unable to install, compile or load a package.

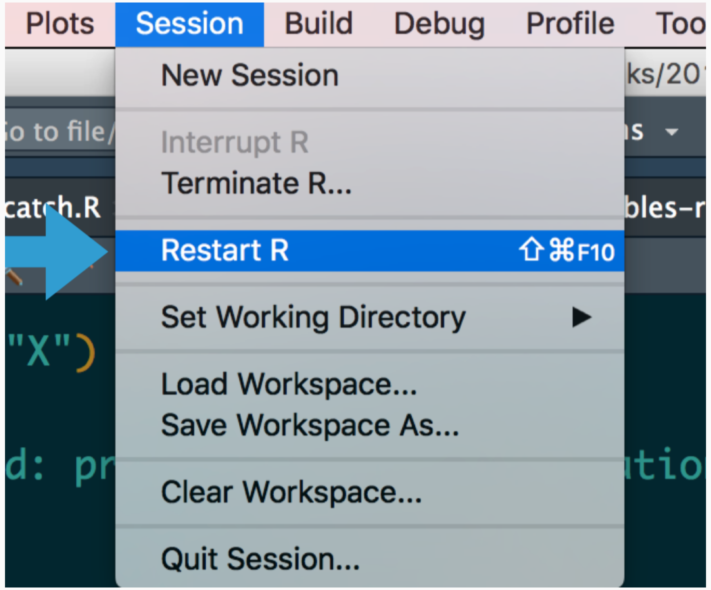
Whenever you see an error message, start by googling it. Improve your chances of a good match by removing any variable names or values that are specific to your problem. Also, look for Stack Overflow posts and list of answers.

[1]Do you get an error message you don't understand? That's good news actually, because the really nasty bugs come without errors.

Reset

- If at first you don't succeed, try exactly the same thing again.
- Have you tried turning it off and on again?
- Do you use `rm(list = ls())`? Don't. Packages remain loaded, options and environment variables set, … all possible sources of error!
- A fresh start clears the workspace, resets options, environment variables, and the path.





Debug

Make the error repeatable.

- Execute the code many times as you consider and reject hypotheses. To make that iteration as quick possible, it's worth some upfront investment to make the problem both easy and fast to reproduce.
- Work with reproducible and minimal examples by removing innocuous code and simplifying data.
- Consider automated testing. Add some nearby tests to ensure that existing good behaviour is preserved.

Track the error down.

- Execute code step by step and inspect intermediate outputs.
- Adopt the scientific method: Generate hypotheses, design experiments to test them, and record your results.

Once found, fix the error and test it.

- Ensure you haven't introduced any new bugs in the process.
- Make sure to carefully record the correct output, and check against the inputs that previously failed.
- Reset and run again to make sure everything still works.

Deter

Defensive programming

- **Pay attention.** Do results make sense? Do they look different from previous results? Why?
- **Know what you're doing**, and what you're expecting.
  - Avoid functions that return different types of output depending on their input, e.g., `[]` and `sapply()`.
  - Be strict about what you accept (e.g., only scalars).
  - Avoid functions that use non-standard evaluation (e.g., `with()`)
- **Fail fast**.
  - As soon as something wrong is discovered, signal an error.
  - Add tests (e.g., with the `testthat` package).
  - Practice good condition/exception handling, e.g., with `try()` and `tryCatch()`.
  - Write error messages for humans.

Transparency

- Collaborate! Pair programming is an established software development technique that increases code robustness. It also works from remote.
- Be transparent! Let others access your code and comment on it.

Debugging R

What you get

```
Error : .onLoad failed in loadNamespace() for 'rJava', details:
call: dyn.load(file, DLLpath = DLLpath, ...)
error: unable to load shared object '/Users/janedoe/Library/R/3.6/library/rJava/libs/rJava.so':
libjvm.so: cannot open shared object file: No such file or directory
Error: loading failed
Execution halted
ERROR: loading failed
* removing '/Users/janedoe/Library/R/3.6/library/rJava/'
Warning in install.packages :
installation of package 'rJava' had non-zero exit status
```

`Credit` Jenny Bryan

What you see

```
Error : blah failed blah blah() blah 'blah', blah:
call: blah.blah(blah, blah = blah, ...)
error: unable to blah blah blah '/blah/blah/blah/blah/blah/blah/blah/blah/blah.so':
blah.so: cannot open blah blah blah: No blah blah blah blah
Error: blah failed
blah blah
ERROR: blah failed
* removing '/blah/blah/blah/blah/blah/blah/blah/'
Warning in blah.blah :
blah of blah 'blah' blah blah-blah blah blah
```

`Credit` Jenny Bryan

Strategies to debug your R code

Sometimes the mistake in your code is hard to diagnose, and googling doesn't help. Here are a couple of strategies to debug your code:

- Use `traceback()` to determine where a given error is occurring.
- Output diagnostic information in code with `print()`, `cat()` or `message()` statements.
- Use `browser()` to open an interactive debugger before the error
- Use `debug()` to automatically open a debugger at the start of a function call.
- Use `trace()` to make temporary code modifications inside a function that you don't have easy access to.

Locating errors with traceback()

Motivation and usage

- When an error occurs with an unidentifiable error message or an error message that you are in principle familiar with but cannot locate its sources, the `traceback()` function comes in handy.
- The `traceback()` function prints the sequence of calls that led to an uncaught error error.
- The `traceback()` output reads from bottom to top.
- Note that errors caught via `try()` or `tryCatch()` do not generate a traceback!
- If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`.

Locating errors with traceback()

Motivation and usage

- When an error occurs with an unidentifiable error message or an error message that you are in principle familiar with but cannot locate its sources, the `traceback()` function comes in handy.
- The `traceback()` function prints the sequence of calls that led to an uncaught error error.
- The `traceback()` output reads from bottom to top.
- Note that errors caught via `try()` or `tryCatch()` do not generate a traceback!
- If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`.

Example

In the call sequence below the execution of `g()` triggers an error:

```
R> f ← function(x) x + 1
R> g ← function(x) f(x)
R> g("a")
```

```
#> Error in x + 1 : non-numeric argument to binary o
```

Doing the traceback reveals that the function call f(x) is what lead to the error:

```
R> traceback()
```

```
#> 2: f(x) at #1
#> 1: g("a")
```

Interactive debugging with browser()

Motivation and usage

- Sometimes, you need more information than the precise location of an error in a function to fix it.
- The interactive debugger lets you pause the run of a function and interactively explore its state.
- Two options to enter the interactive debugger:
    1. Through RStudio's "Rerun with Debug" tool, shown to the right of an error message.
    2. You can insert a call to `browser()` into the function at the stage where you want to pause, and re-run the function.
- In either case, you'll end up in an interactive environment inside the function where you can run arbitrary R code to explore the current state. You'll know when you're in the interactive debugger because you get a special prompt, `Browse[1]>`.

Interactive debugging with browser()

Motivation and usage

- Sometimes, you need more information than the precise location of an error in a function to fix it.
- The interactive debugger lets you pause the run of a function and interactively explore its state.
- Two options to enter the interactive debugger:
  1. Through RStudio's "Rerun with Debug" tool, shown to the right of an error message.
  2. You can insert a call to `browser()` into the function at the stage where you want to pause, and re-run the function.
- In either case, you'll end up in an interactive environment inside the function where you can run arbitrary R code to explore the current state. You'll know when you're in the interactive debugger because you get a special prompt, `Browse[1]>`.

Example

```
R> h ← function(x) x + 3
R> g ← function(b) {
+   browser()
+   h(b)
+ }
R> g(10)
```

Some useful things to do are:

1. Use `ls()` to determine what objects are available in the current environment.
2. Use `str()`, `print()` etc. to examine the objects.
3. Use `n` to evaluate the next statement.
4. Use `s`: like `n` but also step into function calls.
5. Use `where` to print a stack trace (→ traceback).
6. Use `c` to exit debugger and continue execution.
7. Use `Q` to exit debugger and return to the R prompt.

Debugging other peoples' code

Motivation

- Sometimes the error is outside your code in a package you're using, you might still want to be able to debug.
- Two options:
  1. Download the package code locally and debug it is if it were your own.
  2. Use functions which which allow you to start a browser in existing functions, including `recover()` and `debug()`.

Debugging other peoples' code (cont.)

Motivation

- `recover()` serves as an alternative error handler which you activate by calling `options(error = recover)`.
- You can then select from a list of current calls to browse.
- `options(error = NULL)` turns off this debugging mode again.
- A simpler alternative is `options(error = browser)`, but this only allows you to browse the call where the error occurred.

Debugging other peoples' code (cont.)

Motivation

- `recover()` serves as an alternative error handler which you activate by calling `options(error = recover)`.
- You can then select from a list of current calls to browse.
- `options(error = NULL)` turns off this debugging mode again.
- A simpler alternative is `options(error = browser)`, but this only allows you to browse the call where the error occurred.

Example

- Activate debugging mode; then execute (flawed) function:

```
R> options(error = recover)
R> lm(mpg ~ wt, data = "mtcars")
```

```
Error in model.frame.default(formula = mpg ~ wt, data = "mtcars", drop
 'data' must be a data.frame, environment, or list

Enter a frame number, or 0 to exit

1: lm(mpg ~ wt, data = "mtcars")
2: eval(mf, parent.frame())
3: eval(mf, parent.frame())

Selection:
```

- Deactivate debugging mode:

```
R> options(error = NULL)
```

Debugging other peoples' code (cont.)

Motivation

- `debug()` activates the debugger on any function, including those in packages (see on the right). `undebug()` deactivates the debugger again.
- Some functions in another package are easier to find than others. There are
  - *exported* functions which are available outside of a package and
  - *internal* functions which are only available within a package.
- To find (and debug) exported functions, use the `::` syntax, as in `ggplot2::ggplot`.
- To find un-exported functions, use the `:::` syntax, as in `ggplot2:::check_required_aesthetics`.

Debugging other peoples' code (cont.)

Motivation

- `debug()` activates the debugger on any function, including those in packages (see on the right). `undebug()` deactivates the debugger again.
- Some functions in another package are easier to find than others. There are
  - *exported* functions which are available outside of a package and
  - *internal* functions which are only available within a package.
- To find (and debug) exported functions, use the `::` syntax, as in `ggplot2::ggplot`.
- To find un-exported functions, use the `:::` syntax, as in `ggplot2:::check_required_aesthetics`.

Example

- Activate debugging mode for `lm()` function; then execute function:

```
R> debug(stats::lm)
R> lm(mpg ~ weight, data = "mtcars")
```

- Interactive debugging mode for `lm()` is entered; use the common `browser()` functionality to navigate:

```
debugging in: lm(mpg ~ weight, data = mtcars)
debug: {
    ret.x <- x
    ...
Browse[2]>
```

- Deactivate debugging mode:

```
R> undebug(stats::lm)
```

Debugging in RStudio

Debug Mode

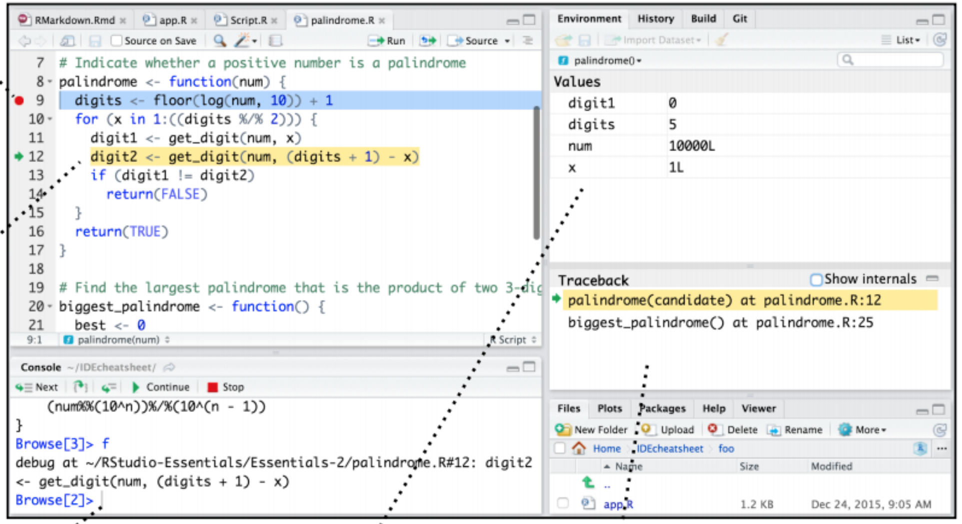
Open with **debug(), browser(),** or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

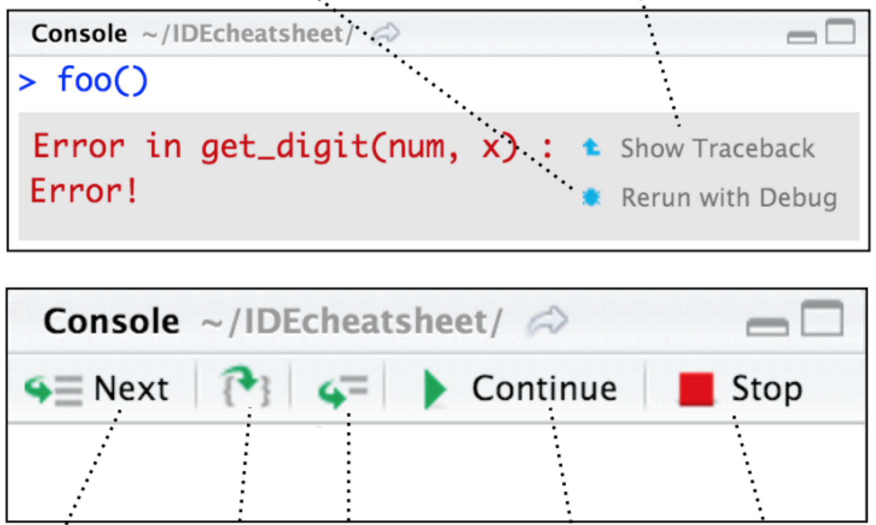
Launch debugger mode from origin of error

Open traceback to examine the functions that R called before the error occurred

Click next to line number to add/remove a breakpoint.

Highlighted line shows where execution has paused





Run commands in environment where execution has paused

Examine variables in executing environment

Select function in traceback to debug

Step through code one line at a time

Step into and out of functions to run

Resume execution

Quit debug mode

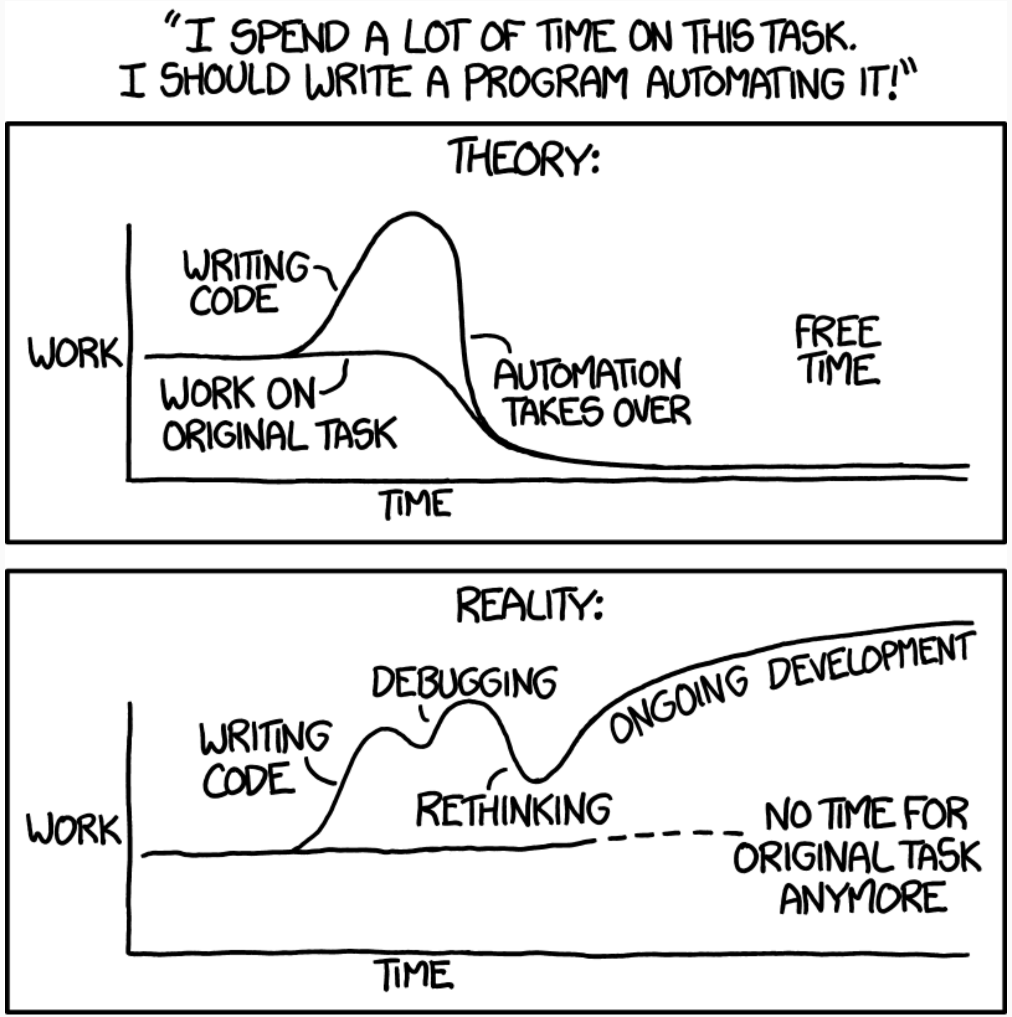
More on debugging R

Further reading

- 12-minute video on debugging in R
- Jenny Bryan's talk on debugging at rstudio::conf 2020
- Jenny Bryan and Jim Hester's "What They Forgot to Teach You About R", Chapter 11: Debugging R code
- Jonathan McPherson's Debugging with RStudio

Automation and scripting

Automation



Credit Randall Munroe/xkcd 1319

Automation

Motivation

- We spend too much time on repetitive tasks.
- We're already automating using scripts that bundle multiple commands! Next step: The pipeline as a series of scripts and commands.
- Good pipelines are modular. But you don't want to trigger 10 scripts sequentially by hand.
- Some tasks are to be repeated on a regular basis (schedule).

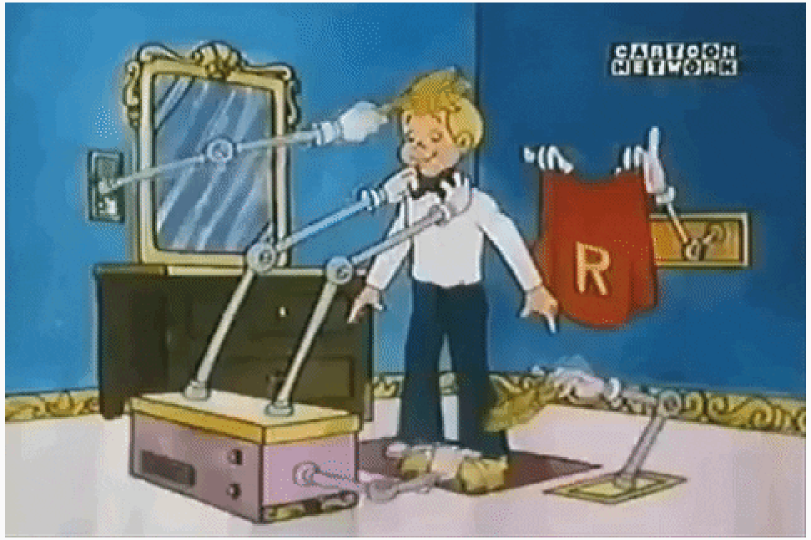
When automation makes sense

- The input is variable but the process of turning input into output is highly standardized.
- You use a diverse set of software to produce the output.
- Others (humans, machines) are supposed to run the analyses.
- Time saved by automation >> Time needed to automate.

Different ways of doing it

We will consider automation

- using **R**,
- using the **Shell** and **RScript**,
- using **make**, and
- using dedicated **scheduling tools**.



Thinking in pipelines

Key characteristics

- Pipelines make complex projects easier to handle because they break up a monolithic script into **discrete, manageable chunks**.
- If properly done, each stage of the pipeline defines its input and its outputs.
- Pipeline modules **do not modify their inputs** (*idempotence*). Rerunning one module produces the same results as the previous run.

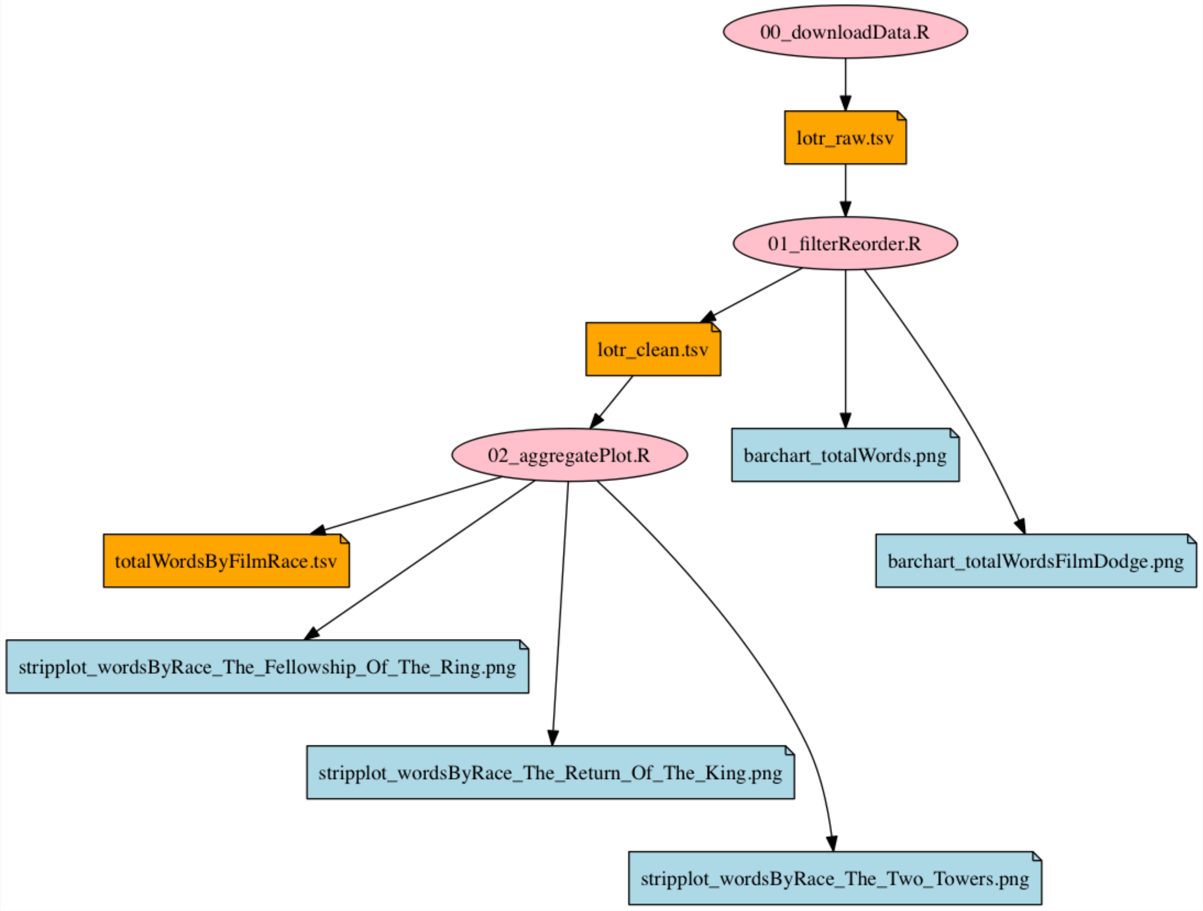
Key advantages

- When you modify one stage of the pipeline, you only have to rerun the downstream, dependent stages.
- Division of labor is straightforward.
- Modules tend to be a lot easier to debug.

A data science pipeline is a graph

Wait what

- Scripts and data files are vertices of the graph.
- Dependencies between stages are edges of the graph.
- Pipelines are not necessarily DAGS. Recursive routines are imaginable (but to be avoided?).
- Also, scripts are not necessarily hierarchical (e.g., multiple different modeling approaches of the same data in different scripts).
- An automation script gives *one* order in which you can successfully run the pipeline.



An example pipeline

In the following, we will work with this toy pipeline:

[1]Courtesy of Jenny Bryan.

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,

`00-packages.R`:

```
R> # install packages from CRAN
R> p_needed ← c("tidyverse" # tidyverse packages
+ )
R> packages ← rownames(installed.packages())
R> p_to_install ← p_needed[!(p_needed %in% packages)]
R> if (length(p_to_install) > 0) {
+   install.packages(p_to_install)
+ }
R> lapply(p_needed, require, character.only = TRUE)
```

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,
- `01-download-data.R` downloads a spreadsheet, which is stored as `lotr_raw.tsv`,

`01-download-data.R`:

```
R> ## download raw data
R> download.file(url = "http://bit.ly/lotr_raw-tsv",
+                      destfile = "lotr_raw.tsv")
```

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,
- `01-download-data.R` downloads a spreadsheet, which is stored as `lotr_raw.tsv`,
- `02-process-data.R` imports and processes the data and exports a clean spreadsheet as `lotr_clean.tsv`, and

`02-process-data.R`:

```
R> ## import raw data
R> lotr_dat ← read_tsv("lotr_raw.tsv")
R>
R> ## reorder Film factor levels based on story
R> old_levels ← levels(as.factor(lotr_dat$Film))
R> j_order ← sapply(c("Fellowship", "Towers", "Return"),
+                      function(x) grep(x, old_levels))
R> new_levels ← old_levels[j_order]
R>
R> ## process data set
R> lotr_dat ← lotr_dat %>%
+   # apply new factor levels to Film
+     mutate(Film = factor(as.character(Film), new_levels),
+     # revalue Race
+     Race = recode(Race, `Ainur` = "Wizard", `Men` = "Man")) %>%
+ ## <skipping some steps here to avoid slide overflow>
+
+ ## write data to file
+ write_tsv(lotr_dat, "lotr_clean.tsv")
```

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,
- `01-download-data.R` downloads a spreadsheet, which is stored as `lotr_raw.tsv`,
- `02-process-data.R` imports and processes the data and exports a clean spreadsheet as `lotr_clean.tsv`, and
- `03-plot.R` imports the clean dataset, produces a figure and exports it as `barchart-words-by-race.png`.

`03-plot.R`:

```
R> ## import clean data
R> lotr_dat ← read_tsv("lotr_clean.tsv") %>%
+ # reorder Race based on words spoken
+ mutate(Race = reorder(Race, Words, sum))
R>
R> ## make a plot
R> p ← ggplot(lotr_dat, aes(x = Race, weight = Words)) + geom_bar()
R> ggsave("barchart-words-by-race.png", p)
```

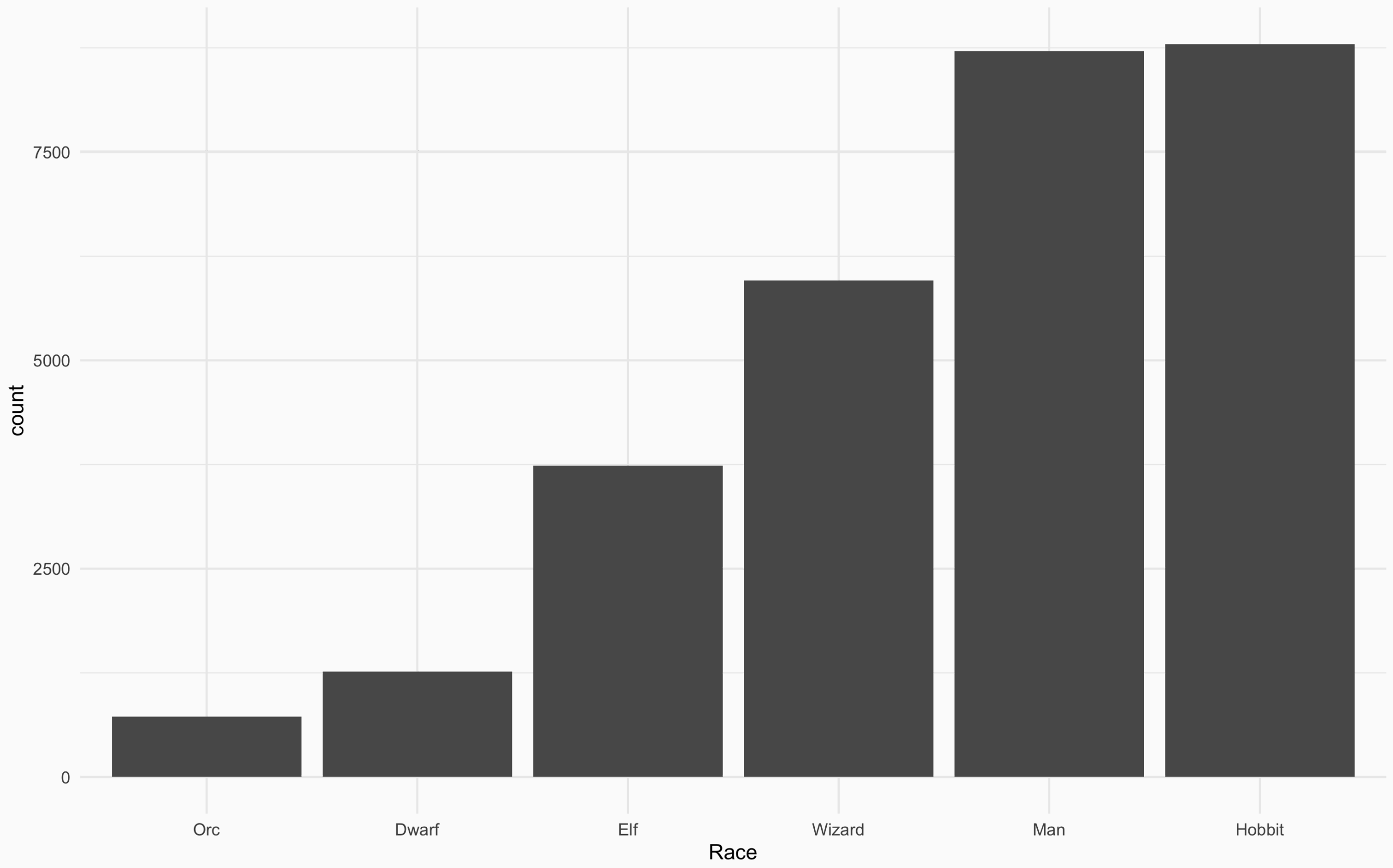
An example pipeline

```
R> slice_sample(lotr_dat, n = 10)
```

```
# A tibble: 10 × 5
   Film                        Chapter                   Character     Race   Words
   <chr>                       <chr>                     <chr>         <chr> <dbl>
 1 The Return Of The King      64: The Mouth Of Sauron   Aragorn       Man      23
 2 The Fellowship Of The Ring 36: The Bridge Of Khazad… Frodo         Hobb…     4
 3 The Two Towers              36: Isengard Unleashed    Saruman       Wiza…    50
 4 The Fellowship Of The Ring 42: The Great River       Sam           Hobb…    37
 5 The Return Of The King      42: Breaking The Gate Of… Gandalf       Wiza…    21
 6 The Two Towers              45: The Glittering Caves  Legolas       Elf      36
 7 The Two Towers              35: Helm's Deep           Rohan Warri… Man      22
 8 The Fellowship Of The Ring 33: Moria                 Aragorn       Man      31
 9 The Fellowship Of The Ring 43: Parth Galen           Aragorn       Man      79
10 The Return Of The King      24: Courage Is The Best … Gothmog       Orc       4
```

An example pipeline

```
R> p ← ggplot(lotr_dat, aes(x = Race, weight = Words)) +
+   geom_bar() + theme_minimal()
```



Automation using pipelines in R

Motivation and usage

- The `source()` function reads and parses R code from a file or connection.
- We can build a pipeline by sourcing scripts sequentially.
- This pipeline is usually stored in a "master" script.
- The removal of previous work is optional and maybe redundant. Often the data is overwritten by default.
- It is recommended that the individual scripts are (partial) standalones, i.e. that they import all data they need by default (loading the packages could be considered an exception).
- Note that as long as the environment is not reset, it remains intact across scripts, which is a potential source of error and confusion.

Automation using pipelines in R

Motivation and usage

- The `source()` function reads and parses R code from a file or connection.
- We can build a pipeline by sourcing scripts sequentially.
- This pipeline is usually stored in a "master" script.
- The removal of previous work is optional and maybe redundant. Often the data is overwritten by default.
- It is recommended that the individual scripts are (partial) standalones, i.e. that they import all data they need by default (loading the packages could be considered an exception).
- Note that as long as the environment is not reset, it remains intact across scripts, which is a potential source of error and confusion.

Example

The master script `master.R`:

```
R> ## clean out any previous work
R> outputs <- c("lotr_raw.tsv",
+              "lotr_clean.tsv",
+              list.files(pattern = "*.png$"))
R> file.remove(outputs)
R>
R> ## run scripts
R> source("00-packages.R")
R> source("01-download-data.R")
R> source("02-process-data.R")
R> source("03-plot.R")
```

Automation using the Shell and Rscript

Motivation and usage

- Alternatively to using an R master script, we can also run the pipeline from the command line.
- Note that here, the environments don't carry over across `Rscript` calls. The scripts definitely have to run in a standalone fashion (i.e., load packages, import all necessary data, etc.).
- The working directory should be set either in the script(s) or in the shell with `cd`.

Automation using the Shell and Rscript

Motivation and usage

- Alternatively to using an R master script, we can also run the pipeline from the command line.
- Note that here, the environments don't carry over across `Rscript` calls. The scripts definitely have to run in a standalone fashion (i.e., load packages, import all necessary data, etc.).
- The working directory should be set either in the script(s) or in the shell with `cd`.

Example

The master script `master.sh`:

```
#!/bin/sh
cd /Users/simonmunzert/github/examples/02-automation
set -eux
Rscript 01-download-data.R
Rscript 02-process-data.R
Rscript 03-plot.R
```

The `set` command allows to adjust some base shell parameters:

- `-e`: Stop at first error
- `-u`: Undefined variables are an error
- `-x`: Print each command as it is run

For more information on `set`, see here.

Automation using the Shell and Rscript

Motivation and usage

- Alternatively to using an R master script, we can also run the pipeline from the command line.
- Note that here, the environments don't carry over across `Rscript` calls. The scripts definitely have to run in a standalone fashion (i.e., load packages, import all necessary data, etc.).
- The working directory should be set either in the script(s) or in the shell with `cd`.
- One advantage of this approach is that it can be easily coupled with other command line tools, building a **polyglot pipeline**.

Example

The master script `master.sh`:

```
#!/bin/sh
cd /Users/simonmunzert/github/examples/02-automation
set -eux
curl -L http://bit.ly/lotr_raw-tsv > lotr_raw.tsv
Rscript 02-process-data.R
Rscript 03-plot.R
```

The `set` command allows to adjust some base shell parameters:

- `-e`: Stop at first error
- `-u`: Undefined variables are an error
- `-x`: Print each command as it is run

For more information on `set`, see here.

Automation using Make

Motivation and usage

- Make is an automation tool that allows us to specify and manage build processes.
- It is commonly run via the shell.
- At the heart of a make operation is the `makefile` (or `Makefile`, `GNUmakefile`), a script which serves as a recipe for the building process.
- A `makefile` is written following a particular syntax and in a declarative fashion.
- Conceptually, the recipe describes which files are built how and using what input.

Advantages of Make

- It looks at which files you have and automatically figures out how to create the files that you have. For complex pipelines this "automation of the automation process" can be very helpful.
- While shell scripts give *one* order in which you can successfully run the pipeline, Make will figure out the parts of the pipeline (and their order) that are needed to build a desired target.

Automation using Make (cont.)

Basic syntax

Each batch of lines indicates

- a file to be created (the target),
- the files it depends on (the prerequisites), and
- set of commands needed to construct the target from the dependent files.

Dependencies propagate.

- To create any of the `png` figures, we need `lotr_clean.tsv`.
- If this file changes, the `png`s change as well when they're built.

Example `makefile`

```
all: lotr_clean.tsv barchart-words-by-race.png words-histogram.png

lotr_raw.tsv:
    curl -L http://bit.ly/lotr_raw-tsv > lotr_raw.tsv

lotr_clean.tsv: lotr_raw.tsv 02-process-data.R
    Rscript 02-process-data.R

barchart-words-by-race.png: lotr_clean.tsv 03-plot.R
    Rscript 03-plot.R

words-histogram.png: lotr_clean.tsv
    Rscript -e 'library(ggplot2);
    qplot(Words, data = read.delim("$<"), geom = "histogram");
    ggsave("$@")'
    rm Rplots.pdf

clean:
    rm -f lotr_raw.tsv lotr_clean.tsv *.png
```

Automation using Make (cont.)

Getting Make to run

- Using the command line, go into the directory for your project.
- Create the `Makefile` file.[1]
- The most basic Make commands are `make all` and `make clean` which builds (or deletes) all output as specified in the script.

Example `makefile`

```
all: lotr_clean.tsv barchart-words-by-race.png words-histogram.png

lotr_raw.tsv:
    curl -L http://bit.ly/lotr_raw-tsv > lotr_raw.tsv

lotr_clean.tsv: lotr_raw.tsv 02-process-data.R
    Rscript 02-process-data.R

barchart-words-by-race.png: lotr_clean.tsv 03-plot.R
    Rscript 03-plot.R

words-histogram.png: lotr_clean.tsv
    Rscript -e 'library(ggplot2);
    qplot(Words, data = read.delim("$<"), geom = "histogram");
    ggsave("$@")'
    rm Rplots.pdf

clean:
    rm -f lotr_raw.tsv lotr_clean.tsv *.png
```

[1] While the basic syntax is simple (see right), the devil's in the detail. Check out resources listed on the next slide if you want to learn more.

Automation using Make - FAQ

Does it work on Windows?

To install an run `make` on Windows, check out these instructions.

Where can I learn more?

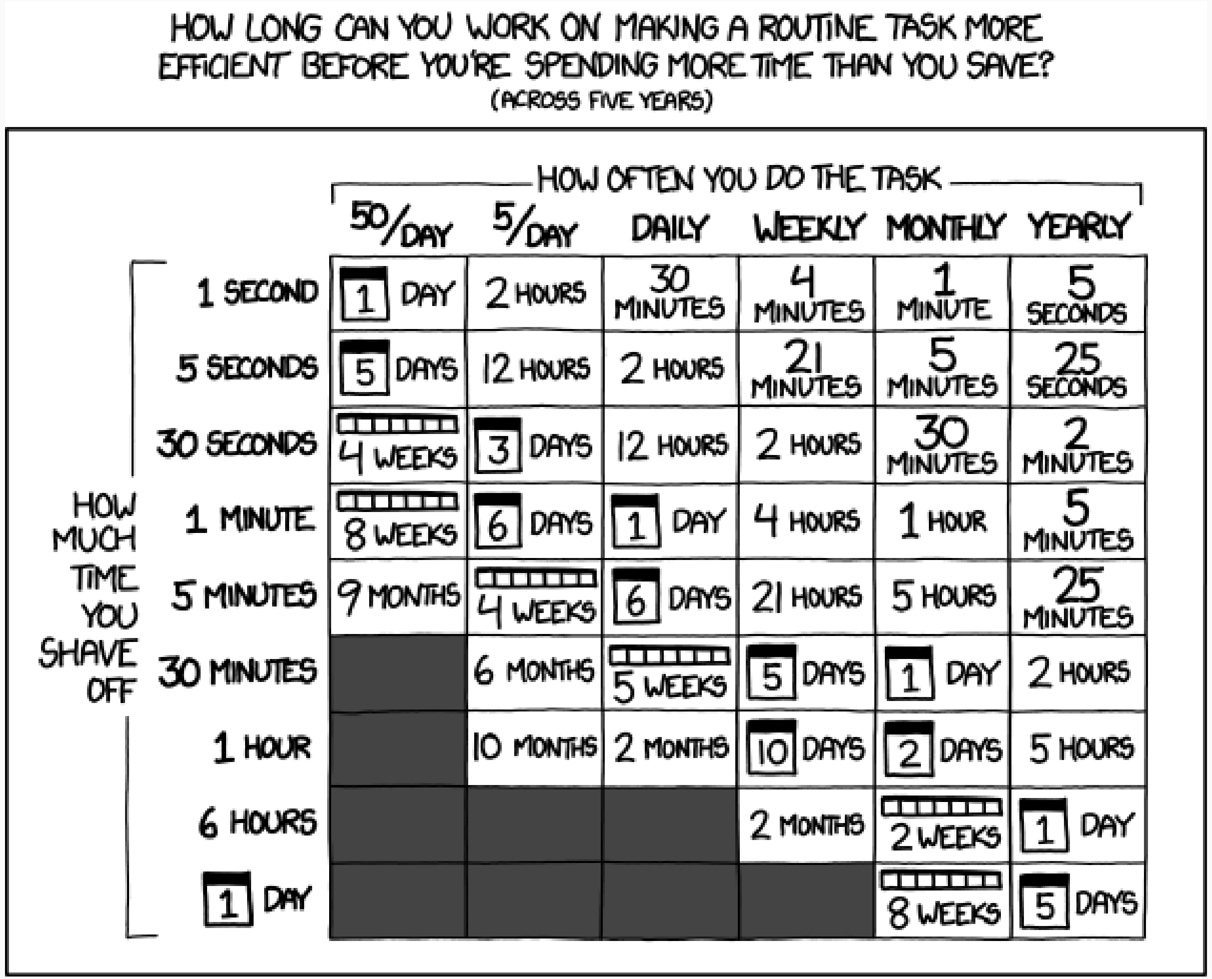
If you consider working with Make, check out the official manual, this helpful tutorial, Karl Broman's excellent minimal make introduction, or this Stat545 piece.

This is dusty technology. Are there alternatives?

In the context of data science with R, the `targets` package is an interesting option. It provides R functionality to define a Make-stype pipeline. Check out the overview and manual.

Scheduling

Scheduling



Credit Randall Munroe/xkcd 1205

Scheduling scripts and processes

Motivation

- So far, we have automated data science pipelines.
- But the execution of these pipelines still needs to be triggered.
- In some cases, it is desirable to also **automate the initialization** of R scripts (or any processes for that matter) **on a regular basis**, e.g. weekly, daily, on logon, etc.
- This makes particular sense when you have moving parts in your pipeline (most likely: data).

`Credit` Simone Giertz

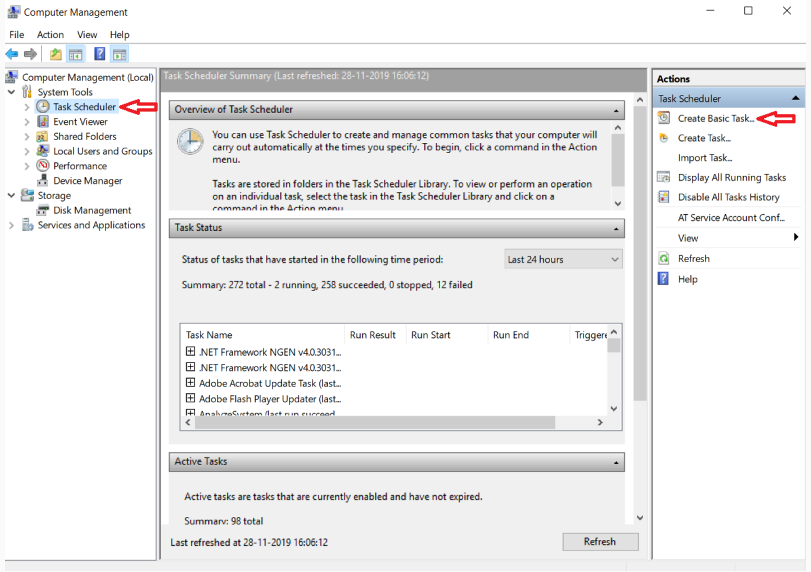
Common scenarios for scheduling

1. You fetch data from the web on a regular basis (e.g., via scraping scripts or APIs).
2. You generate daily/weekly/monthly reports/tweets based on changing data.
3. You build an alert control system informing you about anomalies in a database.

Scheduling scripts and processes on Windows

Scheduling options

- Processes on Windows can be scheduled with the Windows Task Scheduler.
- Manage them via a GUI (→ Control Panel) or the command line using `schtasks.exe`.
- The R package `taskscheduleR` provides a programmable R interface to the WTS.



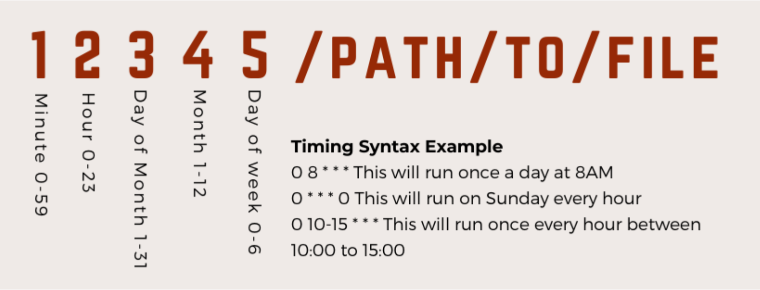
`taskscheduleR` example

```
R> library(taskscheduleR)
R> myscript <- "examples/scrape-wiki.R"
R> ## Run every 5 minutes, starting from 10:40
R> taskscheduler_create(
+   taskname = "WikiScraperR_5min", rscript = myscript,
+   schedule = "MINUTE", starttime = "10:40", modifier = 5)
R>
R> ## Run every week on Saturday and Sunday at 09:10
R> taskscheduler_create(
+   taskname = "WikiScraperR_SatSun", rscript = myscript,
+   schedule = "WEEKLY", starttime = "09:10",
+   days = c('SAT', 'SUN'))
R>
R> ## Delete task
R> taskscheduler_delete("WikiScraperR_SatSun")
R>
R> ## Get a data.frame of all tasks
R> tasks <- taskscheduler_ls()
R> str(tasks)
```

Scheduling scripts and processes on a Mac

Scheduling options

- On macOS you can schedule background jobs using `cron` and `launchd`.
- `launchd`[1] was created by Apple as a replacement for the popular Linux utility `cron` (deprecated but still usable).
- The R package `cronR` provides a programmable R interface.
- `cron` syntax for more complex scheduling:



`cronR` example

```
R> library(cronR)
R> myscript ← "examples/scrape-wiki.R"
R> # Create bash code for crontab to execute R script
R> cmd ← cron_rscript(myscript)
R>
R> ## Run every minute
R> cron_add(command = cmd, frequency = 'minutely',
+          id = 'ScraperR_1min', description = 'Every 1min')
R>
R> ## Run every 15 minutes (using cron syntax)
R> cron_add(cmd, frequency = '*/15 * * * *',
+          id = 'ScraperR_15min', description = 'Every 15 mins')
R>
R> ## Check number of running cronR jobs
R> cron_njobs()
R>
R> ## Delete task
R> cron_rm("WikiScraperR_1min", ask = TRUE)
```

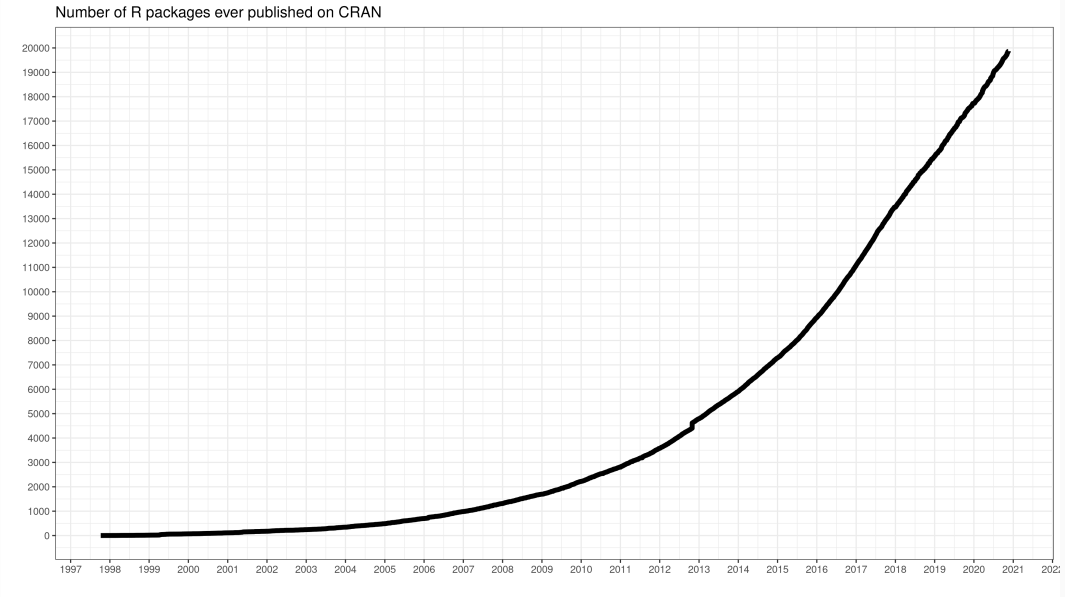
[1] For more resources on scheduling with `launchd`, check out this and this and this.

R packages

Writing an R package

The state of the R package ecosystem

- As of November 2021, the CRAN package repository features more than 18,000 packages.
- Many, many more are available on GitHub and other code sharing platforms.
- R has a vivid community that continuous to create and build extensions and maintain the existing environment. Many of them have much more training and time to invest in software development.
- So, why should we (and with that I mean YOU) write yet another R package?



`Credit` daroczig

Why create another R package?

1. **Thinking in functions.** R is a functional programming language, and packages bundle functions. Thinking of projects as packages is consistent with a functional mindset.

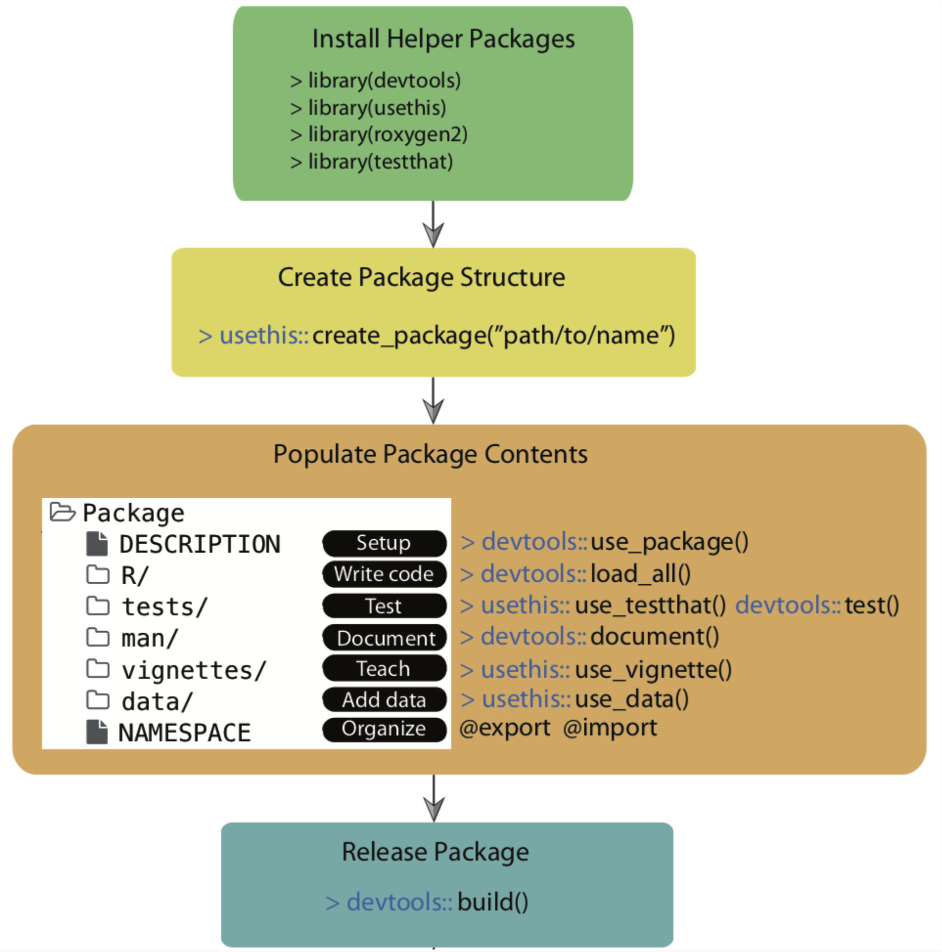
2. **Automation and transportability.** By turning tasks into functions you save repetitive typing, keep frequently-used code together, and let code travel across projects.

3. **Collaboration and transparency.** Packages are ideal to make functionality available to others, but also to let others contribute. As a side effect, it nudges you to document your functions properly and gives you the opportunity to let others review and improve your code easily.

4. **Visibility and productization.** Publishing code in packages is potentially giving you project a big boost in visibility. Also, it is more likely to be perceived as a product than an insular project.

Creating a package from start to finish

1. Choose a package name
2. Set up your package with RStudio (and GitHub)
3. Fill your package with life
   - Add functions
   - Write help files
   - Write a `DESCRIPTION`
   - Add internal data
4. Check your package
   - Write tests
   - Check on various operating systems
   - Check for good coding practice
5. Submit to CRAN (or GitHub early in the process)
6. Promotion
   - Write a vignette
   - Build a package website



`Credit` Simo Goshev, Steve Worthington

Tools to get you started

devtools

- The workhorse of package development in R
- Provides functions that simplify common tasks, such as package setup, simulating installs, compiling from source

usethis

- Provides workflow utilities for project development (loaded by `devtools`)
- Many `use_*()` functions to help create package tests, data, description, etc.

testthat

- Provides functions that make it easy to describe what you expect a function to do, including catching errors, warnings, and messages.

roxygen2

- Provides functions to streamline/automate the documentation of your packages and functions

An example walkthrough

In the following we will briefly study the process of creating a package.

The example is taken from Methods Bites, the Blog of the MZES Social Science Data Lab, and developed by Cosima Meyer and Dennis Hammerschmidt.

The idea is to create a package `overviewR` that helps you to get an overview – hence, the name – of your data with particular emphasis on the extent that your distinct units of observation are covered for the entire time frame of your data set.

The package is real and lives on both CRAN and GitHub. Check out the vignette.

Step 1: Idea and name

Idea

- I'll leave you alone with that one.
- … but you might want to check out the over 18k existing ones that live on CRAN.

Name

- Package names can only be letters and numbers and must start with a letter.
- The package `available` helps you — both with getting inspiration for a name and with checking whether your name is available.

Example

```
R> library(available)
R> # Check for potential names
R> available::suggest("Easily extract information about sample")
```

```
  easilyr
```

```
R> # Check whether it's available
R> available::available("overviewR", browse = FALSE)
```

```
  ── overviewR ──────────────────────────────────────
  Name valid: ✔
  Available on CRAN: ✖
  Available on Bioconductor: ✔
  Available on GitHub:  ✖
  Abbreviations: http://www.abbreviations.com/overview
  Wikipedia: https://en.wikipedia.org/wiki/overview
  Wiktionary: https://en.wiktionary.org/wiki/overview
  Urban Dictionary:
    a general [summary] of a subject "the [treasurer] gave [a brief]
    http://overview.urbanup.com/3904264
```

Step 2: Set up your package

Option 1: via RStudio and GitHub

- Use RStudio's Project Wizard and click on `File` > `New Project...` > `New Directory` > `R Package`.
- Check the box `Create a git` to set up a local git.

Option 2: `usethis`

- Use `usethis::create_package()`, which will set up a template package directory in the specified folder.
- You have to take care of version control yourself (recommendation: initiate project on GitHub first).

Example

```
R> create_package("overviewR", open = FALSE)
```

```
✓ Creating 'overviewR/'
✓ Setting active project to '/Users/simonmunzert/github/intro-to-
✓ Creating 'R/'
✓ Writing 'DESCRIPTION'
Package: overviewR
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.2
✓ Writing 'NAMESPACE'
```

Step 2: Set up your package (cont.)

Basic components

1. The `DESCRIPTION` file
    - stores metadata about the package
    - lists dependencies if any
    - is pre-generated by `roxygen2`

Example

```
Package: overviewR
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person(given = "First",
           family = "Last",
           role = c("aut", "cre"),
           email = "first.last@example.com",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.2
```

Step 2: Set up your package (cont.)

Basic components

1. The `DESCRIPTION` file
    - stores metadata about the package
    - lists dependencies if any
    - is pre-generated by `roxygen2`
    - it will later look like this

Example

```
Type: Package
Package: overviewR
Title: Easily Extracting Information About Your Data
Version: 0.0.2
Authors@R: c(
    person("Cosima", "Meyer", email = "XX@XX.com", role = c("cre"
    person("Dennis", "Hammerschmidt", email = "XX@XX.com", role =
Description: Makes it easy to display descriptive information on
    a data set.  Getting an easy overview of a data set by displa
    visualizing sample information in different tables (e.g., tim
    scope conditions).  The package also provides publishable TeX
    present the sample information.
License: GPL-3
URL: https://github.com/cosimameyer/overviewR
BugReports: https://github.com/cosimameyer/overviewR/issues
Depends:
    R (≥ 3.5.0)
Imports:
    dplyr (≥ 1.0.0)
Suggests:
```

Step 2: Set up your package (cont.)

Basic components

1. The `DESCRIPTION` file
   - stores metadata about the package
   - lists dependencies if any
   - is pre-generated by `roxygen2`
   - it will later look like this
   - and displayed online like this

Example

overviewR: Easily Extracting Information About Your Data

Makes it easy to display descriptive information on a data set. Getting an easy overview of a data set by displaying and visualizing sample information in different tables (e.g., time and scope conditions). The package also provides publishable 'LaTeX' code to present the sample information.

Version:	0.0.7
Depends:	R (≥ 3.5.0)
Imports:	dplyr (≥ 1.0.0), ggplot2 (≥ 3.3.2), tibble (≥ 3.0.1)
Suggests:	covr, devtools, knitr, pkgdown, rmarkdown, spelling, testthat
Published:	2020-11-23
Author:	Cosima Meyer [cre, aut], Dennis Hammerschmidt [aut]
Maintainer:	Cosima Meyer <cosima.meyer at gmail.com>
BugReports:	https://github.com/cosimameyer/overviewR/issues
License:	GPL-3
URL:	https://github.com/cosimameyer/overviewR
NeedsCompilation:	no
Language:	en-US
Materials:	README NEWS
CRAN checks:	overviewR results

Step 2: Set up your package (cont.)

Basic components

1. The `DESCRIPTION` file
   - stores metadata about the package
   - lists dependencies if any
   - is pre-generated by `roxygen2`
   - it will later look like this
   - and displayed online like this
2. The `NAMESPACE` file
   - will later contain information on exported and imported functions.
   - helps you manage (and avoid) function clashes
   - will be populated automatically using `devtools::document()`

Example

```
# Generated by roxygen2: do not edit by hand

export(overview_crossplot)
export(overview_crosstab)
export(overview_heat)
export(overview_na)
export(overview_overlap)
export(overview_plot)
export(overview_print)
export(overview_tab)
importFrom(dplyr,"%>%")
importFrom(ggplot2,ggplot)
importFrom(ggrepel,geom_text_repel)
importFrom(ggvenn,ggvenn)
importFrom(stats,reorder)
importFrom(tibble,"rownames_to_column")
```

Step 2: Set up your package (cont.)

Basic components

1. The `DESCRIPTION` file
    - stores metadata about the package
    - lists dependencies if any
    - is pre-generated by `roxygen2`
    - it will later look like this
    - and displayed online like this
2. The `NAMESPACE` file
    - will later contain information on exported and imported functions.
    - helps you manage (and avoid) function clashes
    - will be populated automatically using `devtools::document()`
3. The **R** folder
    - this is where all the functions you will create go

Step 3: Fill your package with life

Adding functions

The folder **R** contains all your functions and each function is saved in a new R file where the function name and the file name are the same.

In the preamble of this file, we can add information on the function. This information will be used to render the help files.

Example

```
#' @title overview_tab
#'
#' @description Provides an overview table for the time and scope
#'      a data set
#'
#' @param dat A data set object
#' @param id Scope (e.g., country codes or individual IDs)
#' @param time Time (e.g., time periods are given by years, month
#'
#' @return A data frame object that contains a summary of a sampl
#'      can later be converted to a TeX output using \code{overvie
#' @examples
#' data(toydata)
#' output_table ← overview_tab(dat = toydata, id = ccode, time =
#' @export
#' @importFrom dplyr "%>%"
```

Step 3: Fill your package with life (cont.)

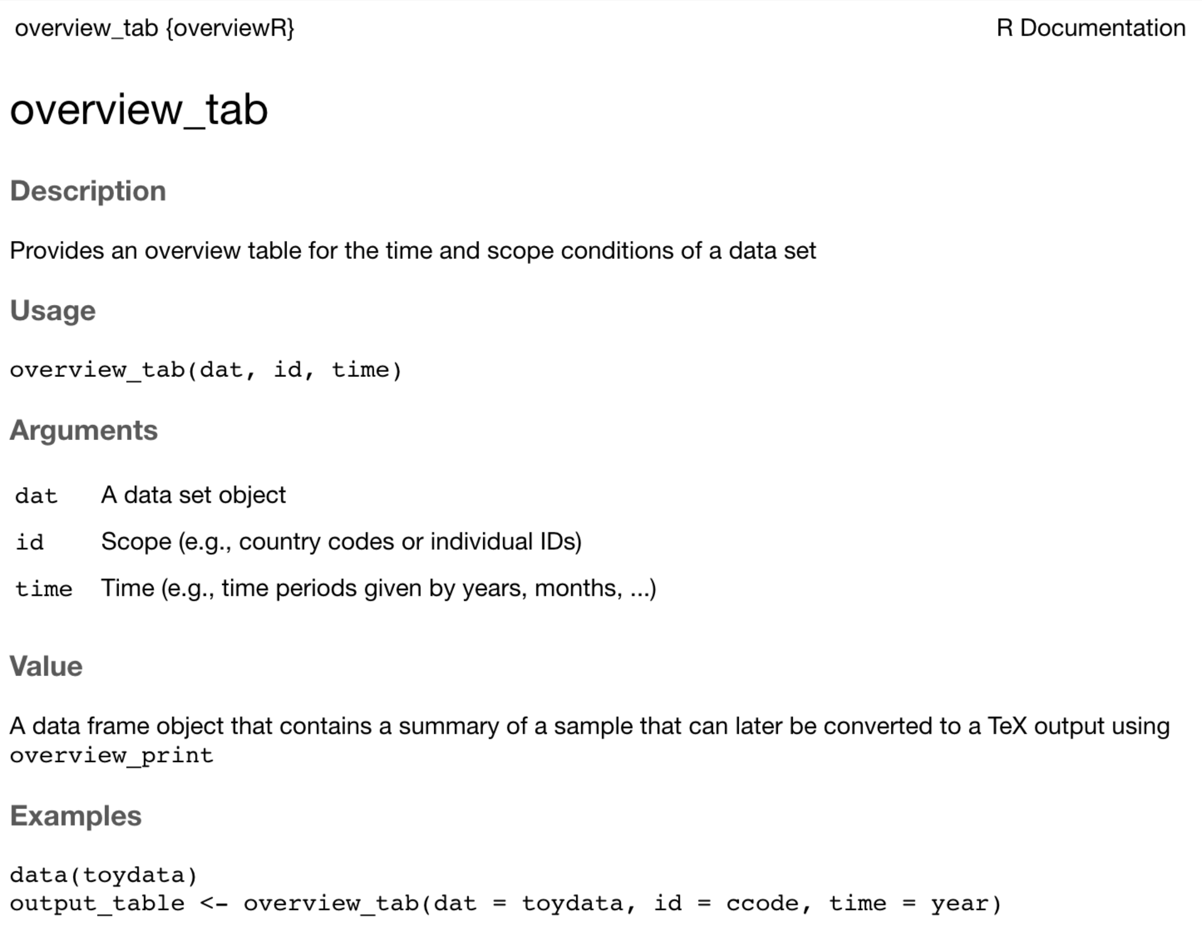
Adding functions

The folder **R** contains all your functions and each function is saved in a new R file where the function name and the file name are the same.

In the preamble of this file, we can add information on the function. This information will be used to render the help files.

When you execute `devtools::document()`, R automatically generates the respective help file in man as well as the new `NAMESPACE` file.

Example



overview_tab {overviewR} R Documentation

overview_tab

Description

Provides an overview table for the time and scope conditions of a data set

Usage

```
overview_tab(dat, id, time)
```

Arguments

`dat`	A data set object
`id`	Scope (e.g., country codes or individual IDs)
`time`	Time (e.g., time periods given by years, months, ...)

Value

A data frame object that contains a summary of a sample that can later be converted to a TeX output using `overview_print`

Examples

```
data(toydata)
output_table <- overview_tab(dat = toydata, id = ccode, time = year)
```

Step 6: Install your package!

Installing a local package

We are now ready to load a developmental version of the package. This works with `devtools::install()`, which will also try to install dependencies of the package from CRAN, if they're not already installed.

You need to run this from the parent working directory that contains the package folder.

We're now ready to call functions from the package.

Example

```
R> install("overviewR")
```

```
✓  checking for file ‘/Users/simonmunzert/github/intro-to-data-sc
–  preparing ‘overviewR’:
✓  checking DESCRIPTION meta-information ...
–  checking for LF line-endings in source and make files and shel
–  checking for empty or unneeded directories
   Omitted ‘LazyData’ from DESCRIPTION
–  building ‘overviewR_0.0.0.9000.tar.gz’

Running /Library/Frameworks/R.framework/Resources/bin/R CMD INSTA
  /var/folders/38/fqbc3hzd0rl23h350bh27_540000gp/T//RtmpAuLJL4/ov
  --install-tests
 installing to library ‘/Library/Frameworks/R.framework/Versions/
 installing *source* package ‘overviewR’ ...
 testing if installed package can be loaded from temporary locati
 testing if installed package can be loaded from final location
 testing if installed package keeps a record of temporary install
 DONE (overviewR)
```

Steps 3-6

We skipped a couple of important (and some optional) steps now, including:

- Build and check a package, clean up → `devtools::check()`
- Iterative loading and testing → `devtools::load_all()`
- Adding unit tests → `usethis::use_testthat()`
- Import functions from other packages (CRAN package dependency) → `usethis::use_package()`
- Git version control and collaboration → `usethis::use_github()`
- Add a proper public description → `usethis::use_readme_rmd()`
- Build PDF manual → `devtools::build_manual()`
- Add vignettes → `usethis::use_vignette()`
- Add a licence → `usethis::use_gpl_license()`, `usethis::use_mit_license()`, ...
- Convert into a single bundled file (binary or zipped) → `devtools::build()`
- Submit to CRAN → `devtools::release()`
- Build website for your package → `pkgdown::build_site()`

Be sure to check out the motivating example and more resources (next slide).

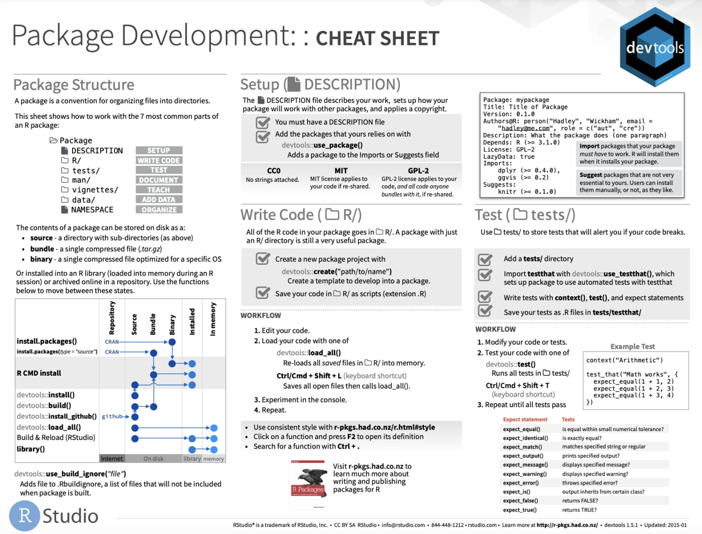
Writing R packages - FAQ

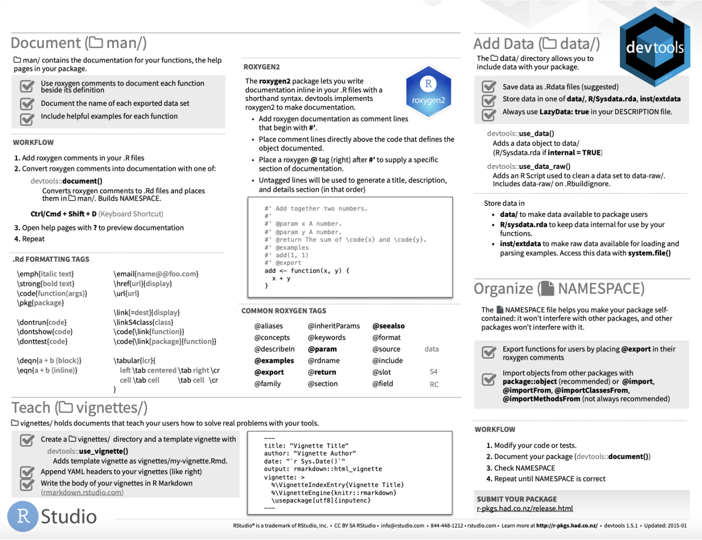
Is learning this worth the time?

Yes.

Where can I learn more?

Glad that you're asking! There's tons of materials out there. Apart from the used tutorial and the R packages book, have a look at the devtools cheatsheet and another overview over at RStudio. Knowing how to turn a package into a website within minutes is fascinating, too.





When do we need a package, and when is a GitHub repo simply enough?

Do you think of your work as a project or a product? If it's the latter, maybe a package is right for you. (But… a research paper is also a product, right? 🤯)

Next steps

Assignment

No further assignment! Be sure to hand in assignment 5 until the updated deadlines.

Next lecture

We turn to the next (and sometimes final) step in the data science workflow, **Monitoring and communication.**