Introduction to Data Science

Session 11: Automation, scheduling, and packages

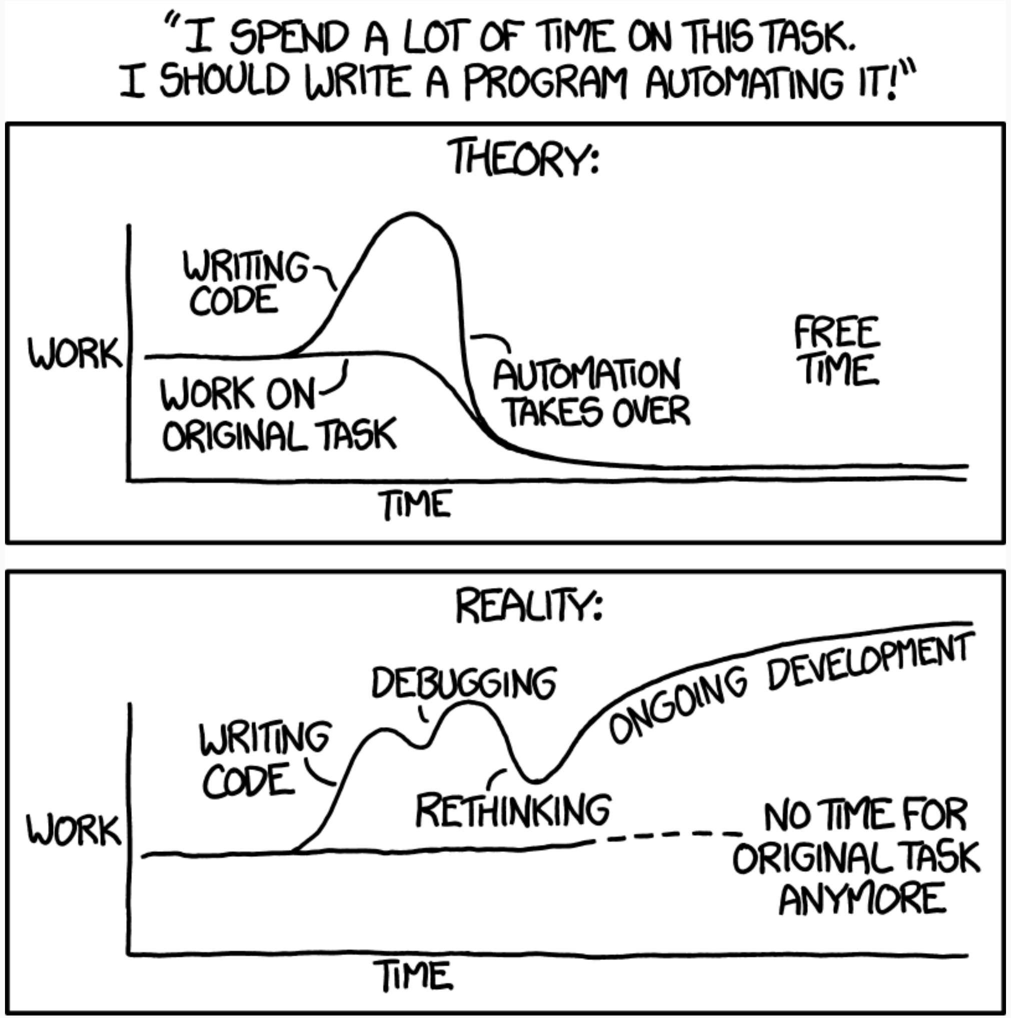
Simon Munzert
Hertie School | GRAD-C11/E1339

Table of contents

1. Automation and scripting
2. Scheduling
3. R packages

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).

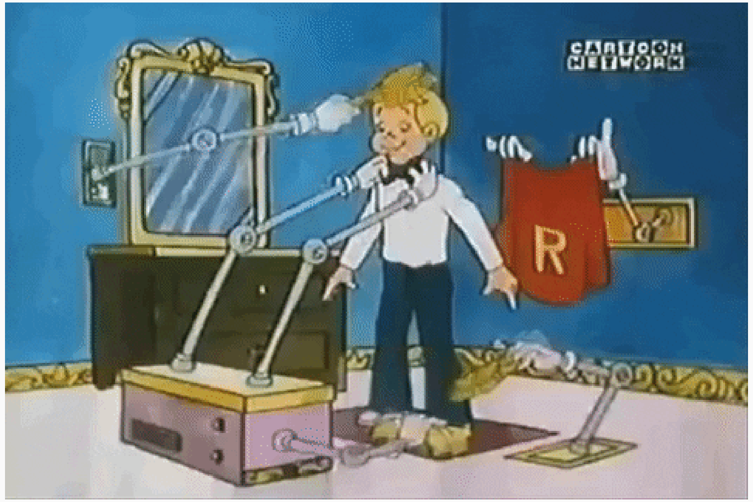
When automation makes sense

- 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.

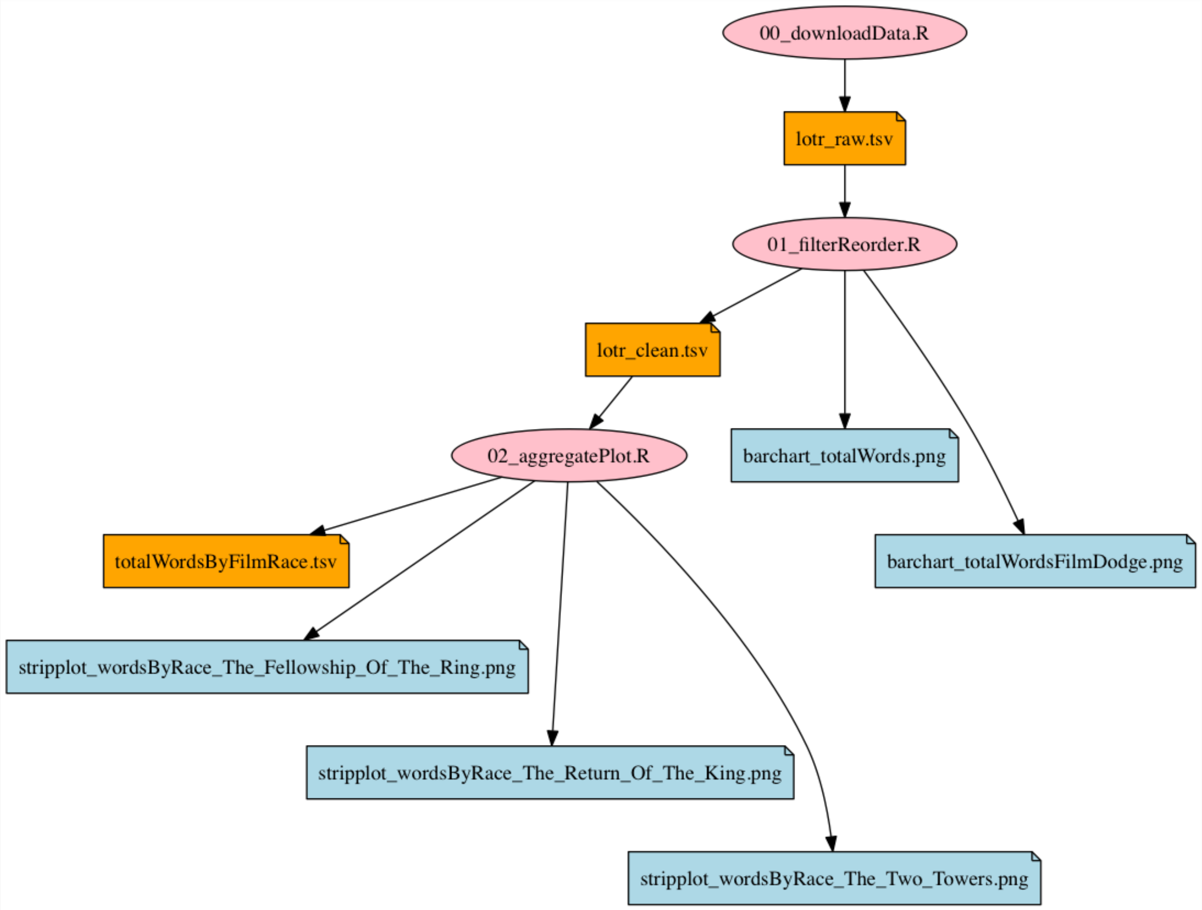
Key advantages

- 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.


00_downloadData.R
lotr_raw.tsv
01_filterReorder.R
lotr_clean.tsv
02_aggregatePlot.R
barchart_totalWords.png
barchart_totalWordsFilmDodge.png
totalWordsByFilmRace.tsv
stripplot_wordsByRace_The_Fellowship_Of_The_Ring.png
stripplot_wordsByRace_The_Return_Of_The_King.png
stripplot_wordsByRace_The_Two_Towers.png

An example pipeline

In the following, we will work with this toy pipeline:[1]

[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)
```

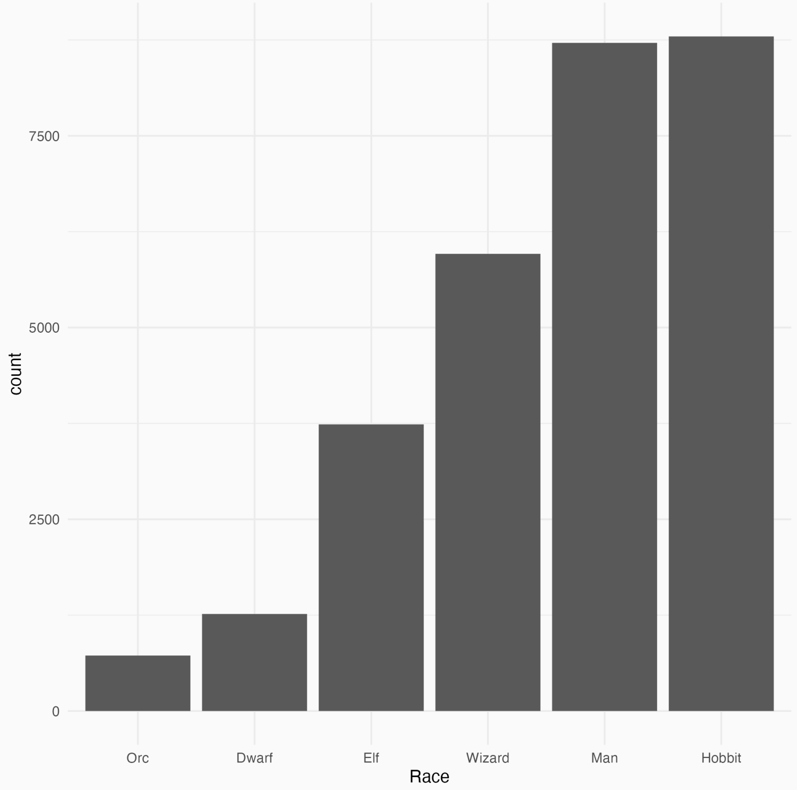
An example pipeline

```
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-dûm 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 Go… Gandalf   Wiza…    21
 6 The Two Towers             45: The Glittering Caves     Legolas   Elf      36
 7 The Two Towers             35: Helm's Deep              Rohan Wa… 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 Def… 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`:

```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`:

```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?

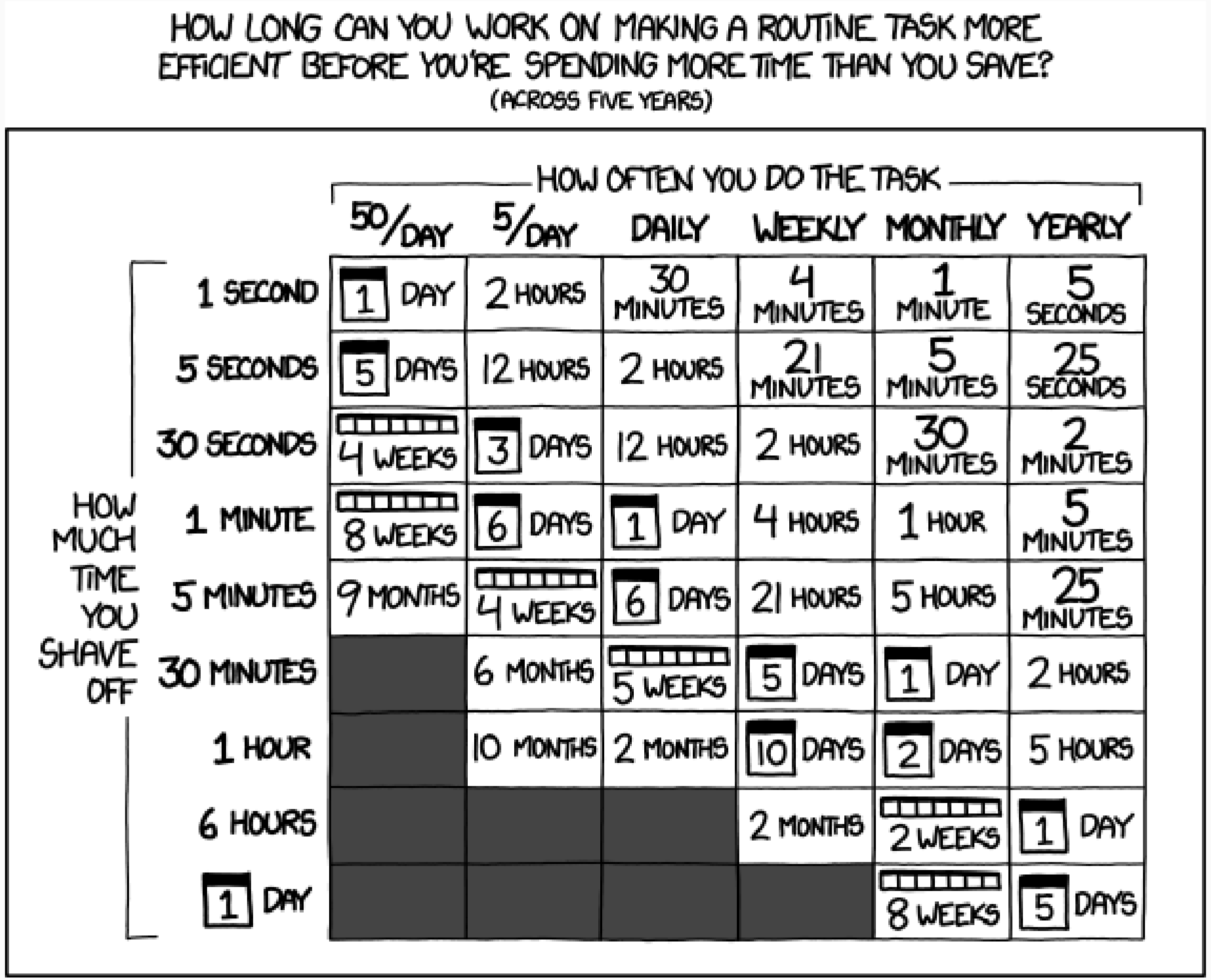
If you consider working with Make, check out the official manual, this helpful tutorial, Karl Broman's excellent minimal make introduction, or this Stat545 piece.

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



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES	■	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR	■	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS	■	■	■	2 MONTHS	2 WEEKS	1 DAY
	1 DAY	■	■	■	■	8 WEEKS	5 DAYS

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

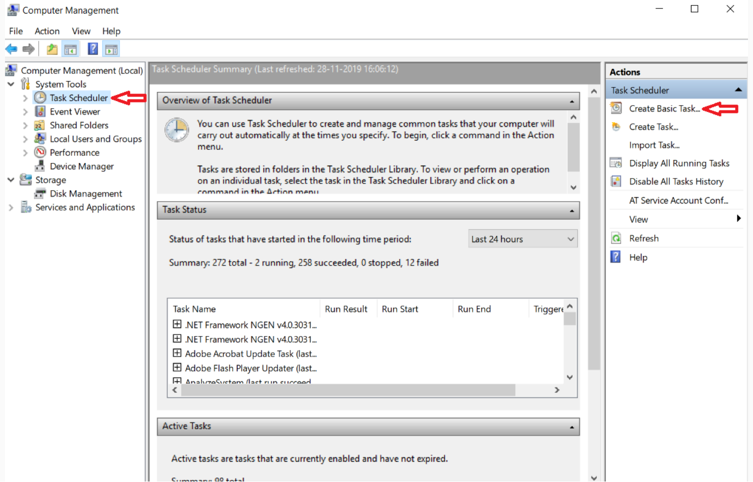
Common scenarios for scheduling

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

- Schedule tasks on Windows with 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.



Task Scheduler Summary (Last refreshed: 28-11-2019 16:06:12)

Overview of Task Scheduler

You can use Task Scheduler to create and manage common tasks that your computer will carry out automatically at the times you specify. To begin, click a command in the Action menu.

Tasks are stored in folders in the Task Scheduler Library. To view or perform an operation on an individual task, select the task in the Task Scheduler Library and click on a command in the Action menu.

Task Status

Status of tasks that have started in the following time period: Last 24 hours

Summary: 272 total - 2 running, 258 succeeded, 0 stopped, 12 failed

Task Name	Run Result	Run Start	Run End	Triggere
⊞ .NET Framework NGEN v4.0.3031...				
⊞ .NET Framework NGEN v4.0.3031...				
⊞ Adobe Acrobat Update Task (last...				
⊞ Adobe Flash Player Updater (last...				
⊞ AnalyzeSystem (last run succeed...				

Active Tasks

Active tasks are tasks that are currently enabled and have not expired.

Summary: 98 total

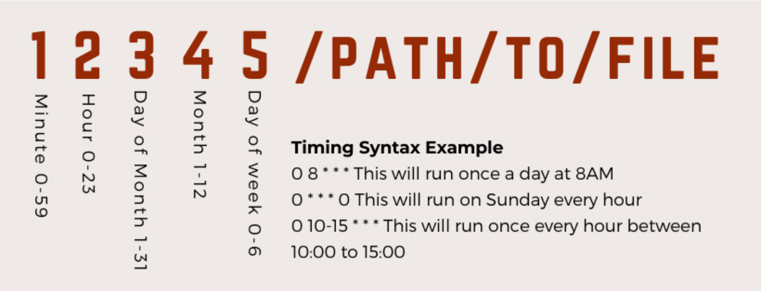
`taskscheduleR` example

```
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:



Timing Syntax Example
0 8 * * * This will run once a day at 8AM
0 * * * 0 This will run on Sunday every hour
0 10-15 * * * This will run once every hour between 10:00 to 15:00

`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)
```

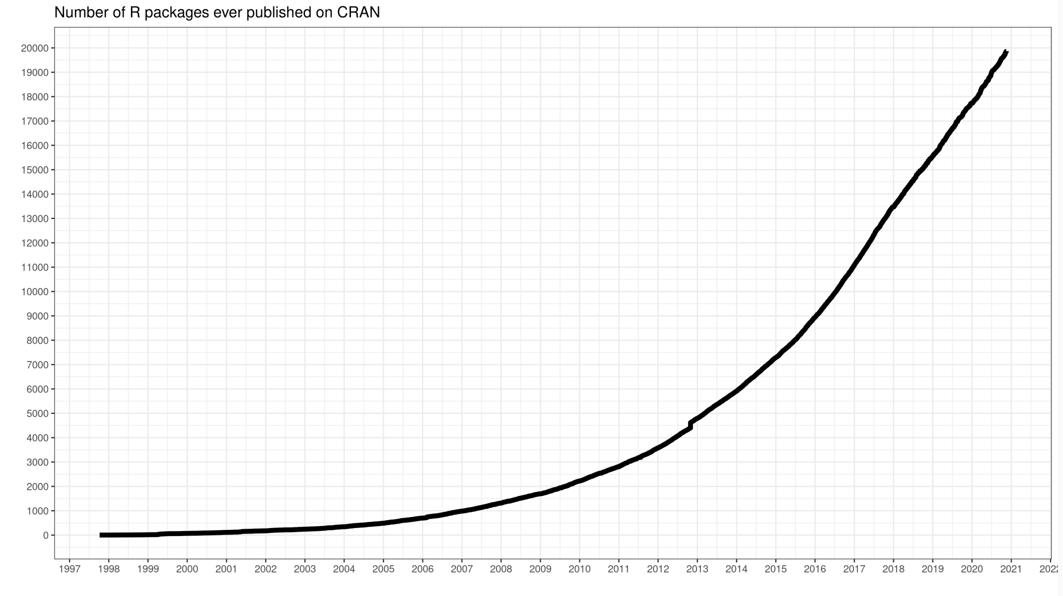
[1] For more resources on scheduling with `launchd`, check out this and this.

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.

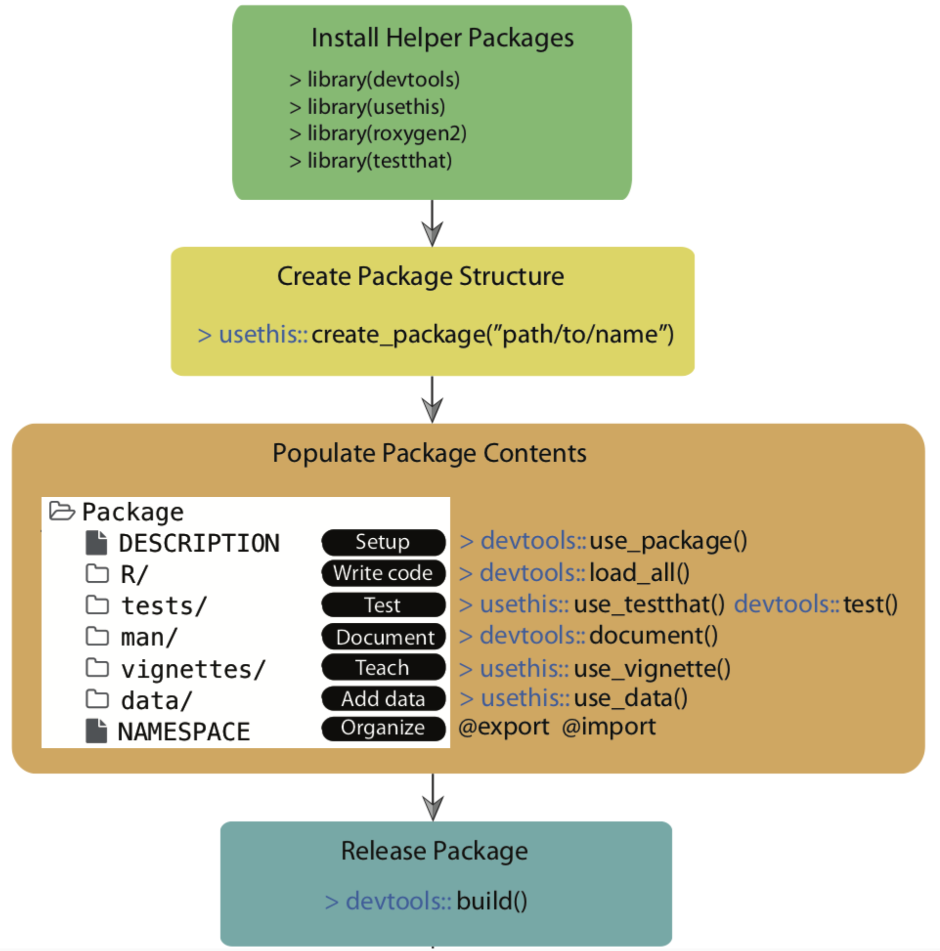
2. **Automation and transportability.** By turning tasks into functions, you save repetitive typing, keep frequently-used code together, and let code travel across projects.

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 your 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



Install Helper Packages

```
> library(devtools)
> library(usethis)
> library(roxygen2)
> library(testthat)
```

Create Package Structure

```
> usethis::create_package("path/to/name")
```

Populate Package Contents

Package		
DESCRIPTION	Setup	> devtools::use_package()
R/	Write code	> devtools::load_all()
tests/	Test	> usethis::use_testthat() devtools::test()
man/	Document	> devtools::document()
vignettes/	Teach	> usethis::use_vignette()
data/	Add data	> usethis::use_data()
NAMESPACE	Organize	@export @import

Release Package

```
> devtools::build()
```

`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
  Sentiment:???
```

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'
✓ Writing 'overviewR.Rproj'
```

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

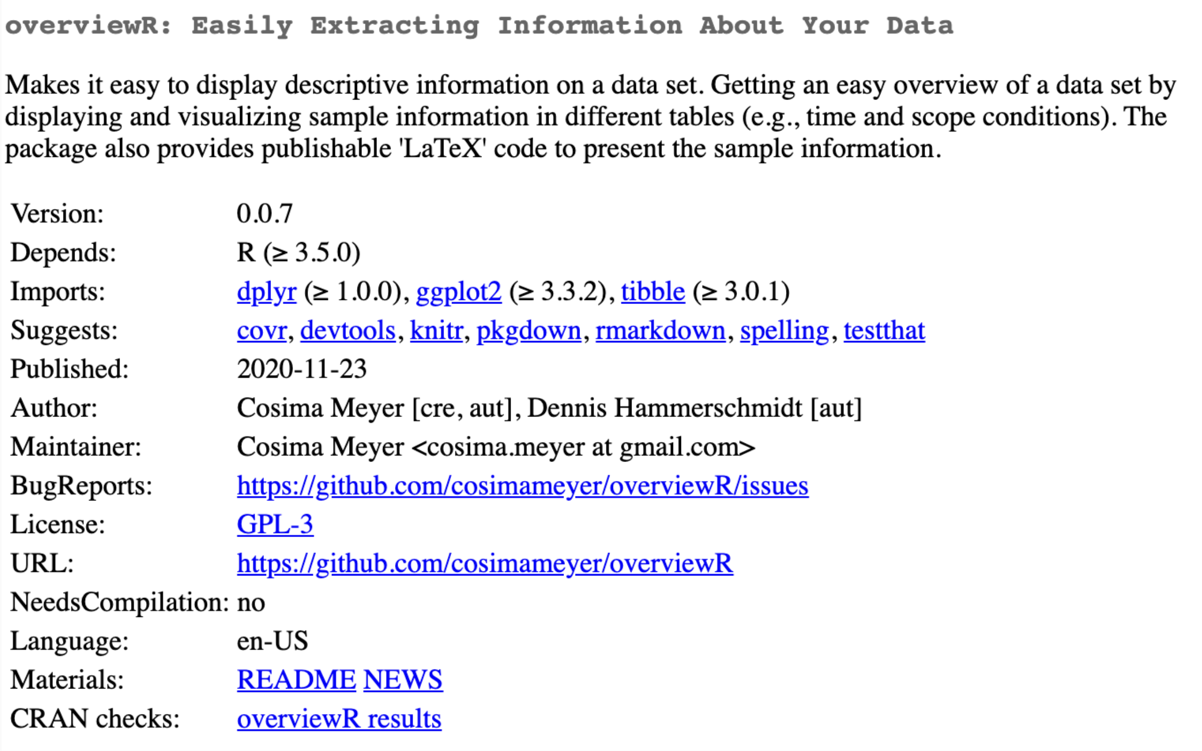
```
Type: Package
Package: overviewR
Title: Easily Extracting Information About Your Data
Version: 0.0.2
Authors@R: c(
    person("Cosima", "Meyer", email = "XX@XX.com", role = c("cre
    person("Dennis", "Hammerschmidt", email = "XX@XX.com", role =
Description: Makes it easy to display descriptive information on
    a data set.  Getting an easy overview of a data set by displa
    visualizing sample information in different tables (e.g., tin
    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.)

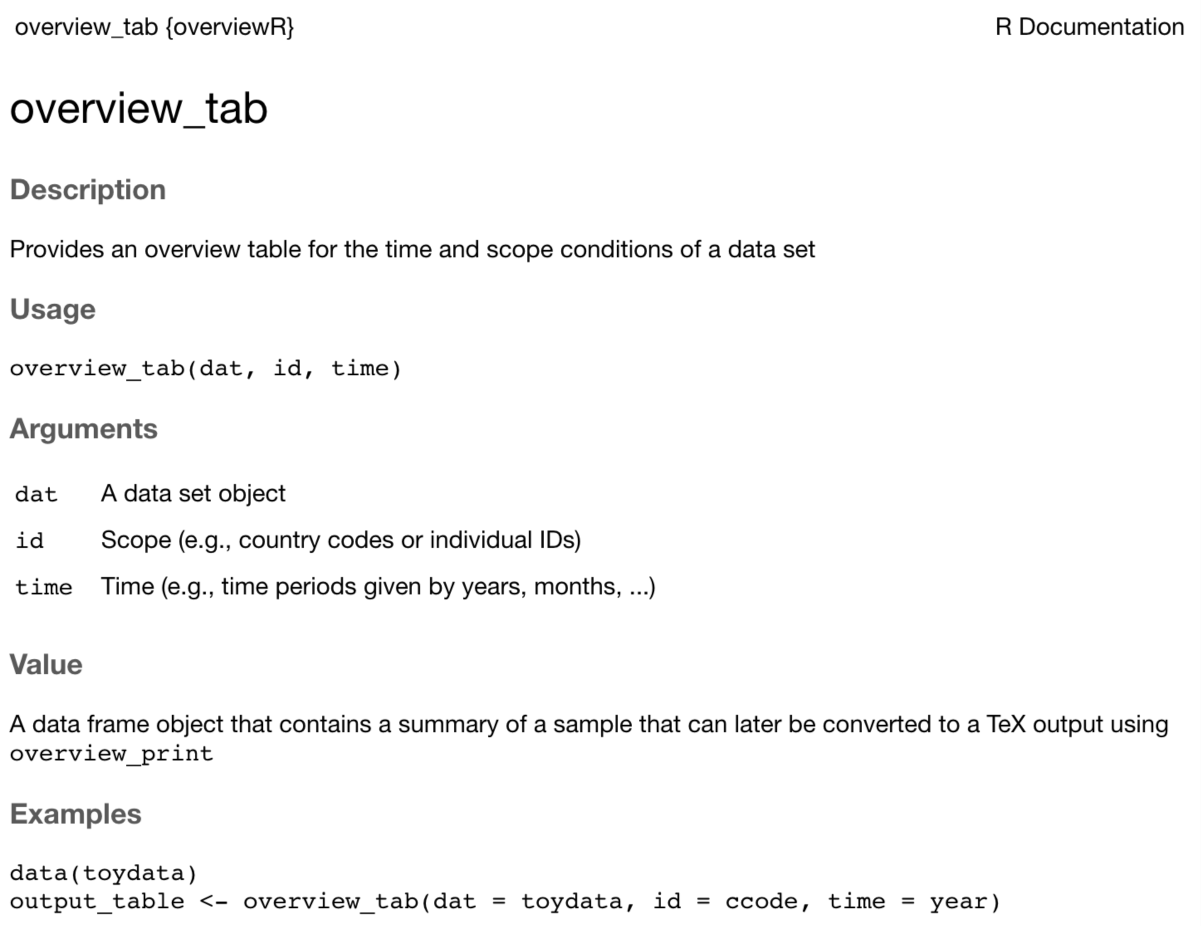
Adding functions

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?

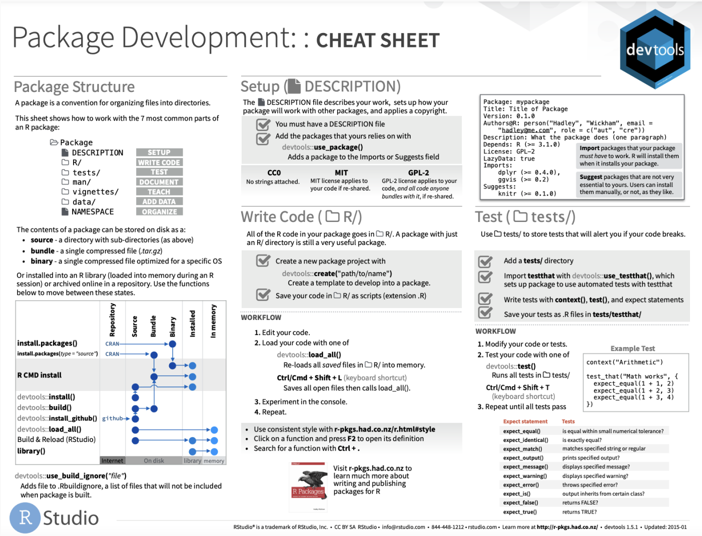
Yes.

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? 🤯)



Package Development: : CHEAT SHEET

Package Structure

A package is a convention for organizing files into directories.

This sheet shows how to work with the 7 most common parts of an R package:

```
Package
  DESCRIPTION    SETUP
  R/             WRITE CODE
  tests/         TEST
  man/           DOCUMENT
  vignettes/     TEACH
  data/          ADD DATA
  NAMESPACE      ORGANIZE
```

The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as above)
- **bundle** - a single compressed file (*.tar.gz*)
- **binary** - a single compressed file optimized for a specific OS

Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.

	Repository	Source	Bundle	Binary	Installed	In memory
install.packages()	CRAN					
install.packages(*type = "source"***)**	CRAN					
R CMD install						
devtools::**install()**						
devtools::**build()**						
devtools::**install_github()**	github					
devtools::**load_all()**						
Build & Reload (RStudio)						
library()						
	Internet	On disk			library	memory

devtools::**use_build_ignore(***"file"***)**
Adds file to .Rbuildignore, a list of files that will not be included when package is built.

Setup (DESCRIPTION)

The DESCRIPTION file describes your work, sets up how your package will work with other packages, and applies a copyright.

☑ You must have a DESCRIPTION file

☑ Add the packages that yours relies on with
devtools::**use_package()**
Adds a package to the Imports or Suggests field

CC0	**MIT**	**GPL-2**
No strings attached.	MIT license applies to your code if re-shared.	GPL-2 license applies to your code, *and all code anyone bundles with it*, if re-shared.

```
Package: mypackage
Title: Title of Package
Version: 0.1.0
Authors@R: person("Hadley", "Wickham", email =
    "hadley@me.com", role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: GPL-2
LazyData: true
Imports:
    dplyr (>= 0.4.0),
    ggvis (>= 0.2)
Suggests:
    knitr (>= 0.1.0)
```

Import packages that your package *must have* to work. R will install them when it installs your package.

Suggest packages that are not very essential to yours. Users can install them manually, or not, as they like.

Write Code (R/)

All of the R code in your package goes in R/. A package with just an R/ directory is still a very useful package.

☑ Create a new package project with
devtools::**create(**"path/to/name"**)**
Create a template to develop into a package.

☑ Save your code in R/ as scripts (extension .R)

WORKFLOW

1. Edit your code.
2. Load your code with one of
   devtools::**load_all()**
   Re-loads all *saved* files in R/ into memory.
   Ctrl/Cmd + Shift + L (keyboard shortcut)
   Saves all open files then calls load_all().
3. Experiment in the console.
4. Repeat.

- Use consistent style with **r-pkgs.had.co.nz/r.html#style**
- Click on a function and press **F2** to open its definition
- Search for a function with **Ctrl + .**

Visit **r-pkgs.had.co.nz** to learn much more about writing and publishing packages for R

Test (tests/)

Use tests/ to store tests that will alert you if your code breaks.

☑ Add a **tests/** directory

☑ Import **testthat** with devtools::**use_testthat()**, which sets up package to use automated tests with testthat

☑ Write tests with **context()**, **test()**, and expect statements

☑ Save your tests as .R files in **tests/testthat/**

WORKFLOW

1. Modify your code or tests.
2. Test your code with one of
   devtools::**test()**
   Runs all tests in tests/
   Ctrl/Cmd + Shift + T
   (keyboard shortcut)
3. Repeat until all tests pass

Example Test

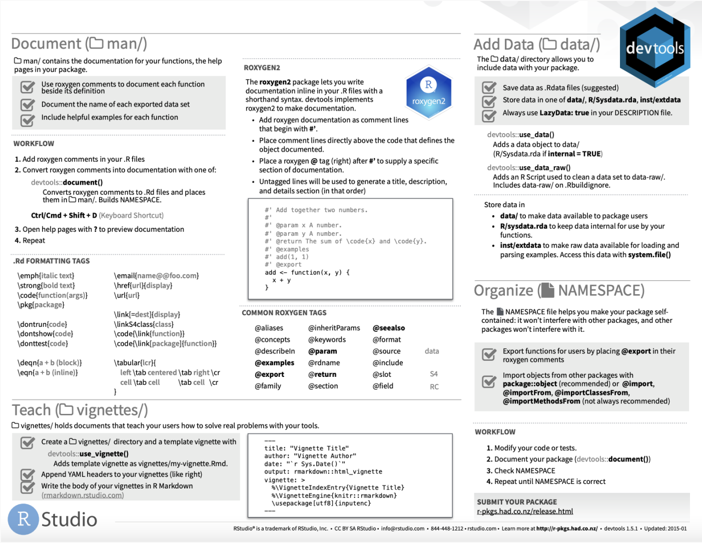
```
context("Arithmetic")

test_that("Math works", {
  expect_equal(1 + 1, 2)
  expect_equal(1 + 2, 3)
  expect_equal(1 + 3, 4)
})
```

Expect statement	Tests
expect_equal()	is equal within small numerical tolerance?
expect_identical()	is exactly equal?
expect_match()	matches specified string or regular
expect_output()	prints specified output?
expect_message()	displays specified message?
expect_warning()	displays specified warning?
expect_error()	throws specified error?
expect_is()	output inherits from certain class?
expect_false()	returns FALSE?
expect_true()	returns TRUE?

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **http://r-pkgs.had.co.nz/** • devtools 1.5.1 • Updated: 2015-01



Document (man/)

man/ contains the documentation for your functions, the help pages in your package.

☑ Use roxygen comments to document each function beside its definition

☑ Document the name of each exported data set

☑ Include helpful examples for each function

WORKFLOW

1. Add roxygen comments in your .R files
2. Convert roxygen comments into documentation with one of:
   devtools::**document()**
   Converts roxygen comments to .Rd files and places them in man/. Builds NAMESPACE.
   Ctrl/Cmd + Shift + D (Keyboard Shortcut)
3. Open help pages with **?** to preview documentation
4. Repeat

.Rd FORMATTING TAGS

```
\emph{italic text}              \email{name@@foo.com}
\strong{bold text}              \href{url}{display}
\code{function(args)}           \url{url}
\pkg{package}
                                \link[=dest]{display}
\dontrun{code}                  \linkS4class{class}
\dontshow{code}                 \code{\link{function}}
\donttest{code}                 \code{\link[package]{function}}

\deqn{a + b (block)}            \tabular{lcr}{
\eqn{a + b (inline)}              left \tab centered \tab right \cr
                                  cell \tab cell      \tab cell  \cr
                                }
```

ROXYGEN2

The **roxygen2** package lets you write documentation inline in your .R files with a shorthand syntax. devtools implements roxygen2 to make documentation.

- Add roxygen documentation as comment lines that begin with **#'**.
- Place comment lines directly above the code that defines the object documented.
- Place a roxygen **@** tag (right) after **#'** to supply a specific section of documentation.
- Untagged lines will be used to generate a title, description, and details section (in that order)

```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' @export
add <- function(x, y) {
  x + y
}
```

COMMON ROXYGEN TAGS

@aliases	@inheritParams	**@seealso**	
@concepts	@keywords	@format	
@describeIn	**@param**	@source	data
@examples	@rdname	@include	
@export	**@return**	@slot	S4
@family	@section	@field	RC

Teach (vignettes/)

vignettes/ holds documents that teach your users how to solve real problems with your tools.

☑ Create a vignettes/ directory and a template vignette with
devtools::**use_vignette()**
Adds template vignette as vignettes/my-vignette.Rmd.

☑ Append YAML headers to your vignettes (like right)

☑ Write the body of your vignettes in R Markdown (rmarkdown.rstudio.com)

```
---
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

Add Data (data/)

The data/ directory allows you to include data with your package.

☑ Save data as .Rdata files (suggested)

☑ Store data in one of **data/**, **R/Sysdata.rda**, **inst/extdata**

☑ Always use **LazyData: true** in your DESCRIPTION file.

devtools::**use_data()**
Adds a data object to data/
(R/Sysdata.rda if **internal = TRUE**)

devtools::**use_data_raw()**
Adds an R Script used to clean a data set to data-raw/. Includes data-raw/ on .Rbuildignore.

Store data in

- **data/** to make data available to package users
- **R/sysdata.rda** to keep data internal for use by your functions.
- **inst/extdata** to make raw data available for loading and parsing examples. Access this data with **system.file()**

Organize (NAMESPACE)

The NAMESPACE file helps you make your package self-contained: it won't interfere with other packages, and other packages won't interfere with it.

☑ Export functions for users by placing **@export** in their roxygen comments

☑ Import objects from other packages with **package::object** (recommended) or **@import**, **@importFrom**, **@importClassesFrom**, **@importMethodsFrom** (not always recommended)

WORKFLOW

1. Modify your code or tests.
2. Document your package (devtools::**document()**)
3. Check NAMESPACE
4. Repeat until NAMESPACE is correct

SUBMIT YOUR PACKAGE
r-pkgs.had.co.nz/release.html

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **http://r-pkgs.had.co.nz/** • devtools 1.5.1 • Updated: 2015-01

Next steps

Quiz

One more quiz to go!

Next lecture

One more session to go. We're going to talk about communication and monitoring.