

Staying clean and DRY

best practices for code reuse and collaboration

Peter Foley

23 November, 2019

Functions

```
plot_prices <- function(data) {  
  ggplot(data, aes(x=sqft, y=price)) +  
  geom_point() + geom_smooth()  
}
```

Projects



Glossary

Glossary

Don't repeat yourself

Side Effects

Clean Code

Glossary - Don't repeat yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Maybe a bit strict for our purposes.

Don't have big copy/paste blocks.

Don't have 4 different "connect_to_database" functions floating around a project.

Glossary - Side Effects

Changing state in a way that could change how other code runs.

```
pot <- fresh_pot(); mug <- new_mug(capacity_ounces = 12)
```

Glossary - Side Effects

```
temp <-  
  temperature(pot)  
  
pour(from=pot,  
     into=mug,  
     ounces=12)
```

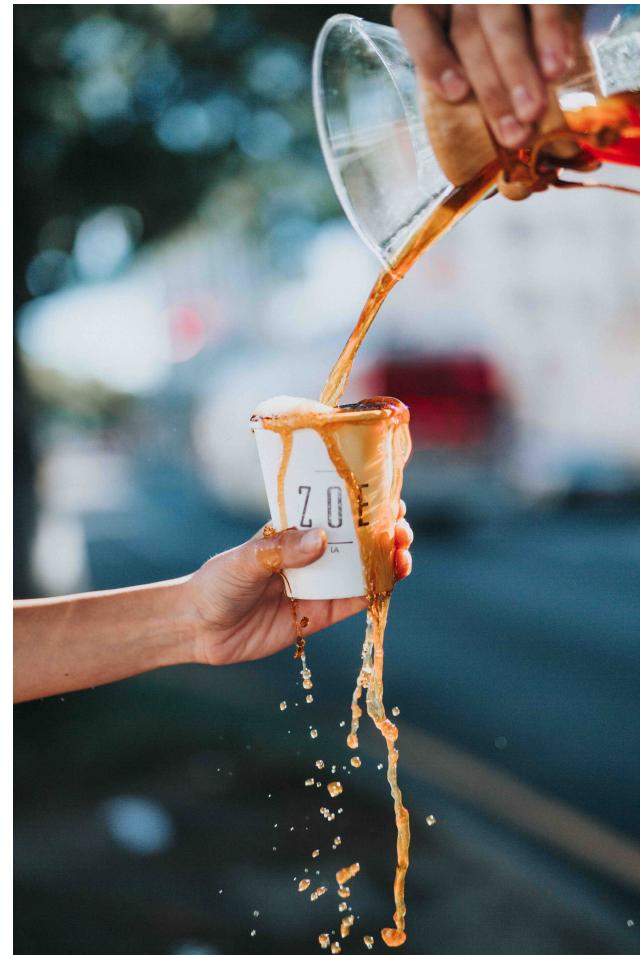


Glossary - Side Effects

```
# again!
pour(from=pot,
     into=mug,
     ounces=12)
```

Defining `pour` is safe, calling `pour()` can be dangerous.

```
plot()
write_csv()
Sys.setenv()
library()
#> stats vs. dplyr filter
```



Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship



Glossary - Clean Code

"Clean code can be read, and enhanced by a developer other than its original author."

"You know you are working on clean code when each routine you read turns out to be pretty much what you expected."

Data projects have a lot of verbose code to move, clean, reshape, or enhance the data. It has a tendency to pile up and bury the big picture.

That boilerplate is rarely tested and often copy/pasted.

Reading code is harder than writing it.

Time to get organized!

Options for organizing code

- standalone scripts
- utility scripts (`source("utils.R")`)
- utility package (`devtools::load_all("utils_pkg")`)
- standalone package (`library(package)`)

The Initial Project

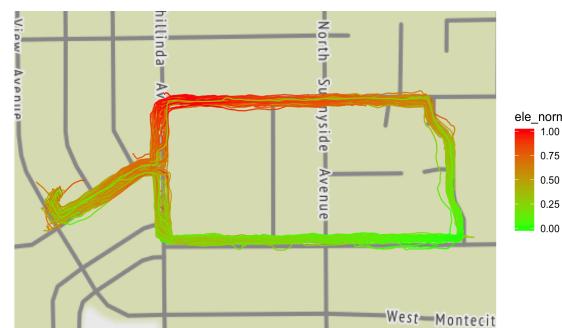
Plot all my dog walks, colored by elevation.



X



=



setup_keys.R

```
# set up garmin password in keyring
keyring::key_set_with_value(
  "sso.garmin.com",
  username=rstudioapi::askForPassword("Garmin Connect username/email"),
  password=rstudioapi::askForPassword("Garmin Connect password"))
```

download_data.R

```
## Get all the activity data
library(here)
library(reticulate)
library(keyring)
library(rcurl)
library(dplyr)
library(readr)

# get garmin creds
gc_user <- "pwfoley@gmail.com"
gc_pass <- keyring::key_get("sso.garmin.com",username=gc_user)

# run the downloader tool with reticulate
output_dir <- here("private_data/gcexport")
# most recent N, or "all" gets everything
max_activities <- "all"
gcexport_dir <- here("external_tools/garmin-connect-export")
gcexport <- import_from_path("gcexport", path = gcexport_dir)
export_args <- c(
  "--username", gc_user,
  "--password", gc_pass,
  "--activities",
  "--format", "original",
  "--directory", output_dir,
  "--subdir", "fit",
  "--unzip"
)
res <- gcexport$main(c("gcexport.py", export_args))

# deduplicate the csv file that it created
# read_lines instead of read_csv to preserve weird quoting
# and to preserve CRLF
read_lines(file.path(output_dir,"activities.csv")) %>%
  unique %>%
  write_lines(file.path(output_dir,"activities.csv"),
             sep = "\r\n")
```

get_downloader.R

```
# clone or update the downloader tool
library(here)
library(fgets)

exporter_dir <- here("external_tools/garmin-connect-export")
repo.create(exporter_dir, showWarnings = FALSE, recursive = TRUE)
repo_already_cloned <- file.exists(path(exporter_dir, ".git/"))
if(repo_already_cloned) {
  pull_retval <- system2("git",c("-C", exporter_dir, "pull"), stdout = FALSE, stderr=FALSE)
  if(!identical(pull_retval,0L)) {
    warning("update of downloader tool failed")
  }
} else {
  clone_retval <- system2("git",c("clone","https://github.com/pe-st/garmin-connect-export",exporter_dir), stdout = FALSE, stderr=FALSE)
  if(!identical(clone_retval,0L)) {
    stop("clone of downloader tool failed")
  }
}
```

map_dog_walks.R

```
# Map walks
library(dplyr)
library(km_per_mile)
library(furrr)
library(readr)
library(here)
library(fs)
library(xml2)

future::plan(future::multiprocess())

here <- here::here
gcexport_dir <- here("private_data/gcexport")

activities <- read_csv(file.path(gcexport_dir,"activities.csv")) %>%
  distinct

fit_from_id <- function(id) {
  fit_path <- path(gcexport_dir,"fit",paste0("activity_",id,".fit"))
  fit_data <- fit::read.fit(fit_path)
  fit_data$record
}

km_per_mile <- 1.60934
dog_walks <- activities %>%
  filter(Activity_ID == "Pasadena Walking",
        Distance (km) %% between(1.25 * km_per_mile,
        1.4 * km_per_mile)) %>%
  mutate(data = future_map(`Activity ID`, fit_from_id, .progress = TRUE))

all_flat <- dog_walks %>%
  filter(`Activity ID` != 2858485459,
        `Activity ID` != 3062996451) %>%
  tidyverse::nest(col="data") %>%
  mutate(lat = altitude,
        lon = position_lat,
        lon = position_long)

latlon_bb <- function(df) {
  # left/bottom/right/top
  clean <- function(x) {
    x <- na.omit(x)
    x <- x[linear(abs(x), 100, tol=.001)]
    x
  }
  lat <- clean(df$lat)
  lon <- clean(df$lon)
  c(min(lon, na.rm=T),
    min(lat, na.rm=T),
    ... (34 more lines)...)
```


Utility scripts

.R files that are `source()`'ed to define functions or variables.

```
library(dplyr)
library(ggplot2)

source("utils_data.R")
source("utils_mapping.R")

get_the_data(date="2019-11-23") %>%
  filter(driver=="Penelope") %>%
  map_all_drives()
```

Standalone → Utility scripts

Why put things in utility scripts?

- less code
- write once for all collaborators
- separates "define how to do X" from "do X"
 - "Teach your code to fish"
 - more readable "Action" scripts: `catch_a_fish()`

Standalone → Utility scripts

Do put in utility scripts:

- Shared configuration (file locations, etc.)
- Shared needs
 - DB authentication
 - loading particular data
 - data filter/cleanup functions
 - standardized definitions
- Big ugly function definitions

Standalone → Utility scripts

Do NOT put in:

- code that actually takes actions
 - reading data
 - writing data
 - saving plots
 - user interaction
- `source()` calls to other utility scripts
- `library()`

New utility scripts

utils/config.R

```
strict_config(value, ...)
```

utils/mapping.R

```
get_all_activities()
get_activity_data(id)
flatten_activities(nested)
latlon_bb(df, lat=df$lat,
          lon=df$lon)
pad_bb(bb, lat=1.1,
        lon=lat)
make_basemap(flat_data, zoom=16)
```

New mapping script

map_runs.R

```
# Map walks
library(dplyr)
library(purrr)
library(furrr)

source("utils/mapping.R")
source("utils/config.R")

future::plan(future::multiprocess())
here <- here::here

km_per_mile <- 1.60934
all_flat <- get_all_activities() %>%
  filter(`Activity Name` == "Pasadena Running",
    `Distance (km)` %>%
      between(4.5 * km_per_mile,
        6 * km_per_mile)) %>%
  mutate(data = future_map(`Activity ID`, get_activity_data, .progress = TRUE)) %>%
  flatten_activities()

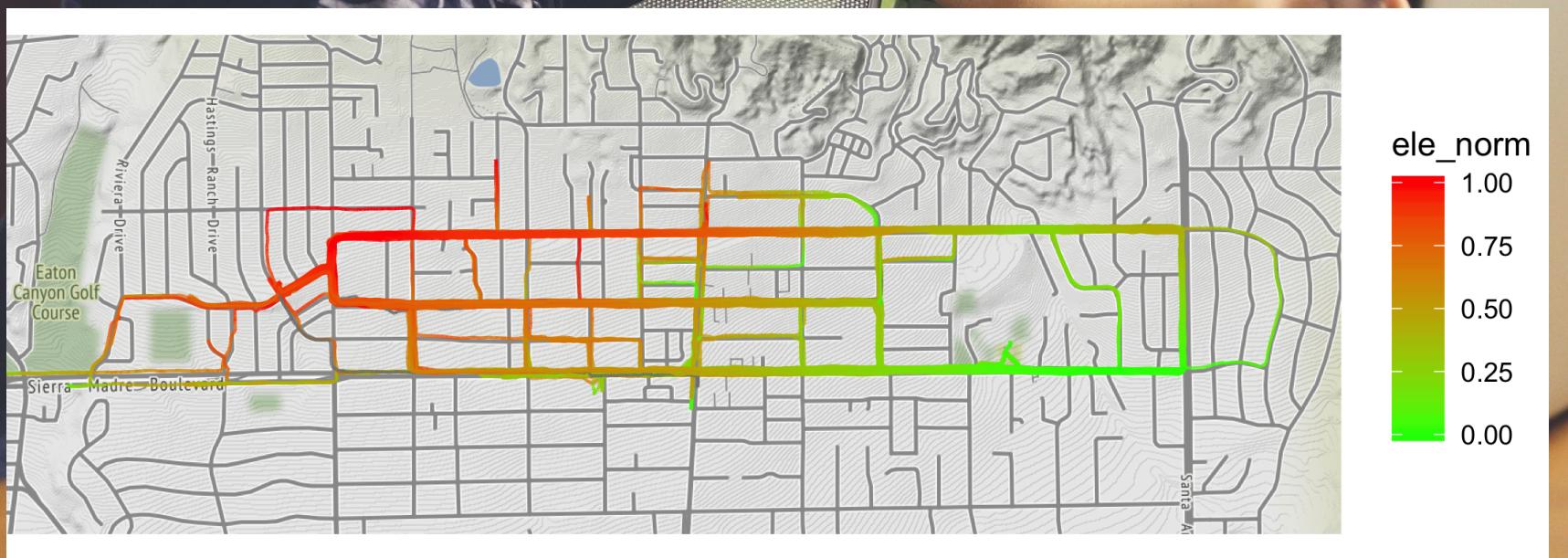
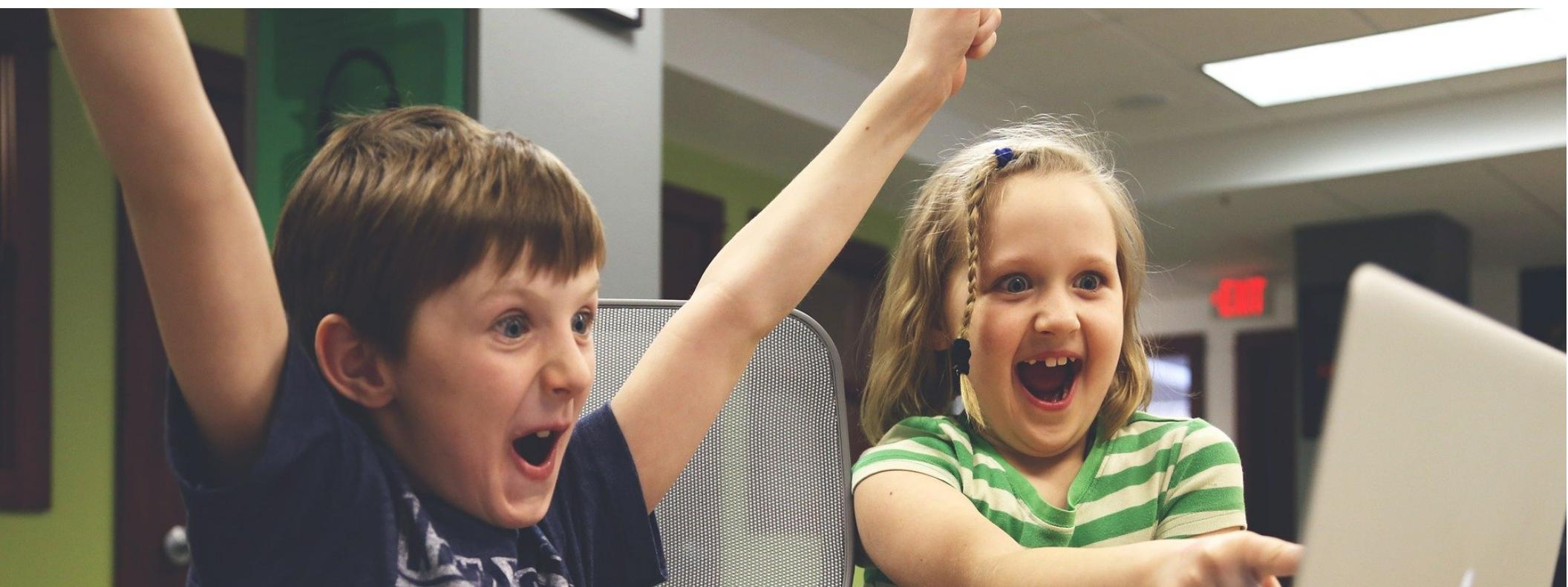
smaller_area <- all_flat %>%
  group_by(`Activity ID`) %>%
  filter(min(lon) > -118.0867)

library(ggmap)
basemap <- make_basemap(smaller_area, zoom=15)

pl_dat <- all_flat %>%
  group_by(`Activity ID`) %>%
  mutate(ele_norm = (ele - min(ele))/(max(ele)-min(ele)))

run_plot <- basemap +
  geom_path(data=pl_dat, aes(x=lon,y=lat, group=`Activity ID`, color=ele_norm)) +
  scale_colour_gradient(low="green", high="red")

gsave(run_plot,
  file="runs.png",
  height=2.5, width=7)
```

What did we win?

All configuration in one place

Very easy to add a new mapping script.

RStudio will pop up argument names when you're typing the function.

```
> get_activity_data(id)  
>  
> get_activity_data()
```

What's missing?

Function help/documentation

```
> ?get_activity_data  
No documentation for 'get_activity_data' in specified packages and libraries:  
you could try '??get_activity_data'
```

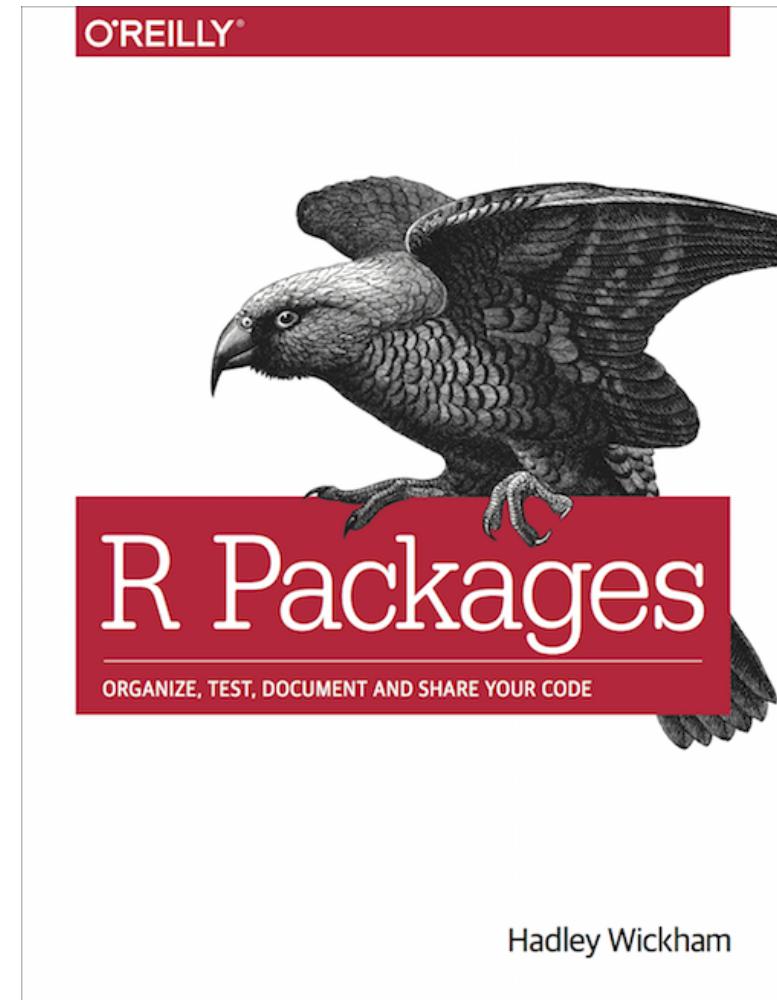
Gets complicated when utility functions need each other

Mapping scripts only need `utils/config.R` to let `utils/mapping.R` functions find data.

Hard to know what packages the utilities require:

`utils/config.R` needs `config` `utils/mapping.R` needs `ggmap`, `fs`, ...

That sounds familiar...



A Package!

Docs/help integrate nicely with RStudio

Namespace lets everything work together

Dependencies are well defined

BUT

Packages are complicated

and I just have a little code

just for this one project

and I don't want to have to install it

and please don't make me pick a name

The solution

```
devtools::load_all()
```

?devtools::load_all()

Load Complete Package.

load_all : loads a package. It roughly simulates what happens when a package is installed and loaded with `library()`.

```
load_all(path = ".", reset = TRUE, recompile = FALSE,  
         export_all = TRUE, helpers = TRUE, quiet = FALSE, ...)
```

export_all : If TRUE (the default), export all objects. If FALSE, export only the objects that are listed as exports in the NAMESPACE file.

`devtools::load_all()` lets you

House utility functions alongside "Action" code.

Keep a tidy namespace.

Write documentation -- **or not.**

Write tests -- **or not.**

Define dependencies -- **or not.**

Utility Scripts → Utility Package

1. Make a package skeleton

```
usethis::create_package(  
  path = "myutils",  
  rstudio=FALSE, open=FALSE  
)  
)
```

2. Move all your utility scripts in the R/ folder.

```
system("mv utils/*.R myutils/R/")
```

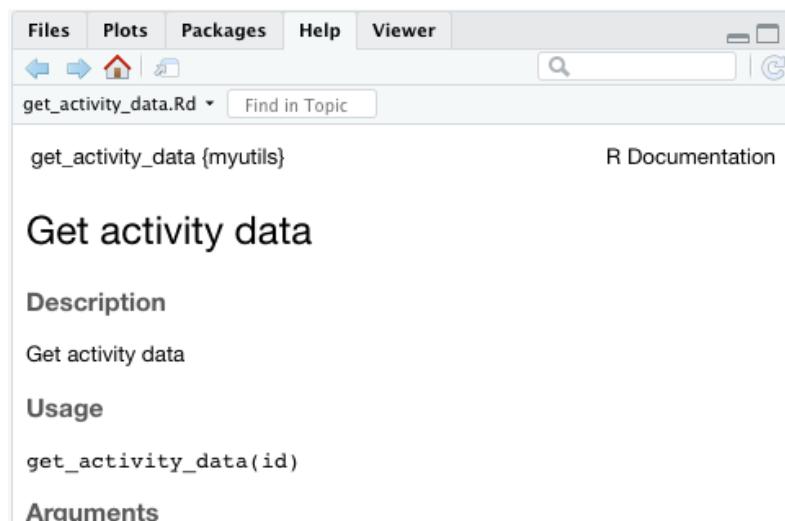
3. Be done if you feel like it.

```
devtools::load_all("myutils")
```

Got roxygen2 docs?

```
#' Get activity data
#'
#' @param id Garmin Activity ID
#'
#' @return dataframe with one row
#' @export
get_activity_data <- function(id)
  fit_path <- fs::path(strict_conf)
  fit_data <- fit::read.fit(fit_pa
  fit_data$record
}
```

```
> devtools::document("myutils")
Updating myutils documentation
Writing NAMESPACE
Loading myutils
Writing NAMESPACE
> devtools::load_all("myutils")
Loading myutils
> ?get_activity_data
Rendering development documentation
```



The screenshot shows the RStudio Help Viewer interface. The title bar includes tabs for 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the tabs, there are navigation icons for back, forward, search, and help. A dropdown menu shows 'get_activity_data.Rd' and a 'Find in Topic' search bar. The main content area displays the documentation for 'get_activity_data' from the 'myutils' package. The title is 'Get activity data'. The 'Description' section states 'Get activity data'. The 'Usage' section shows the function signature 'get_activity_data(id)'. The 'Arguments' section is currently empty.

Like tests?

1. Add the boilerplate

```
setwd("myutils")
usethis::use_testthat()
```

2. Write your tests

```
usethis::use_test("onething")
# opens tests/testthat/test-onething.R for editing
```

3. Test!

```
devtools::test(".")
```

Have dependencies?

Use this will put them in DESCRIPTION for you. Same for github packages!

```
usethis::use_package("ggmap")
devtools::install_github("kuperov/fit")
usethis::use_dev_package("fit")
```

- add @import roxygen tags if you want to use fun() instead of ggmap::fun() within myutils
- use_dev_package requires you install the package first so it can find the metadata.

Once DESCRIPTION is set up, installing all the dependencies is trivial.

```
devtools::install_deps("myutils")
```

Using your utility package

```
source("utils/config.R")
source("utils/mapping.R")
```

```
devtools::load_all("myutils")
```

| Add to Depends in DESCRIPTION if you want to skip the library() calls

What's different from utility scripts?

Good

- Functions can work together better
- Documentation
- Tests
- Dependency handling

Bad

- Bit more work
- Function definitions are harder to find

Same

Project utilities maintained within project repo

How is this not a regular package?

It is a regular package.

```
devtools::install("myutils")
library(myutils)
```

But it's only meant to be used within the project

Though it's possible to do this:

```
devtools::install_github(
  "peterfoley/clean_and_dry",
  subdir="2_load_all/myutils")
```

Standalone package

The code is identical to your utility package.

Difference is in how a "standalone" package is **managed**.

- Users ≠ Developers
- Likely higher stakes

Utility → Standalone package

1. Define the maintainers
2. Move package to its own git repo
3. `usethis::use_travis()`
4. `usethis::use_coverage()`
5. Expand testing
6. Make docs user-friendly

When the options make sense

- | | |
|--------------------|---|
| Bag-of-scripts | <ul style="list-style-type: none">• Early-stage exploratory work. One-offs.• While all the scripts are short• When the scripts aren't duplicative |
| Utility scripts | <ul style="list-style-type: none">• Dividing/specializing labor• Multiple similar tasks in scripts• Utility scripts are few and short |
| Utility package | <ul style="list-style-type: none">• More specialization in labor• More utilities that call each other• Higher stakes (= need tests) |
| Standalone package | <ul style="list-style-type: none">• Useful outside the project |

A close-up photograph of a large, light brown dog with a darker brown patch on its back. The dog is sitting on a vibrant red carpet, looking slightly to the left of the camera. A person's legs and feet are visible in the background, wearing dark trousers and socks. The dog is wearing a black collar with a small purple tag.

Thank you!