Collaborative Data Science Practices

Will Beasley

2019-11-04

Contents

1 Prerequisites **9**

2 Architecture Principles **11**

2.1 Encapsulation 11

2.2 Leverage team member's strenghts & avoid weaknesses 11

2.3 Scales 11

2.4 Consistency 11

2.5 Defensive Style 11

3 Prototypical File **13**

3.1 Clear Memory 13

3.2 Load Sources 14

3.3 Load Packages 14

3.4 Declare Globals 16

3.5 Load Data 16

3.6 Tweak Data 16

3.7 (Unique Content) 16

3.8 Verify Values 16

3.9 Specify Output Columns 16

3.10 Save to Disk or Database 16

4 Prototypical Repository **17**

4.1 Root . . . 17

4.2 Analysis . . . 18

4.3 Data Public . . . 18

4.4 Data Unshared . . . 18

4.5 Documentation . . . 18

4.6 Manipulation . . . 18

4.7 Stitched Output . . . 18

4.8 Utility . . . 18

5 Data at Rest **19**

5.1 Data States . . . 19

5.2 Data Containers . . . 19

6 Patterns **21**

6.1 Ellis . . . 21

6.2 Arch . . . 24

6.3 Ferry . . . 24

6.4 Scribe . . . 24

6.5 Analysis . . . 24

6.6 Presentation -Static . . . 24

6.7 Presentation -Interactive . . . 24

6.8 Metadata . . . 24

7 Security & Private Data **25**

7.1 File-level permissions . . . 25

7.2 Database permissions . . . 25

7.3 Public & Private Repositories . . . 25

8 Automation **27**

8.1 Flow File in R 27

8.2 Makefile 27

8.3 SSIS 27

8.4 cron Jobs & Task Scheduler 27

8.5 Sink Log Files 27

9 Scaling Up **29**

9.1 Data Storage 29

9.2 Data Processing 29

10 Parallel Collaboration **31**

10.1 Social Contract 31

10.2 Code Reviews 31

10.3 Remote 31

10.4 Loose Notes 31

11 Documentation **33**

11.1 Team-wide 33

11.2 Project-specific 33

11.3 Dataset Origin & Structure 33

11.4 Issues & Tasks 33

11.5 Flow Diagrams 33

11.6 Setting up new machine 33

12 Style Guide **35**

12.1 ggplot2 35

13 Publishing Results **37**

13.1 To Other Analysts 37

13.2 To Researchers & Content Experts 37

13.3 To Technical-Phobic Audiences 37

14 Testing, Validation, & Defensive Programming **39**

14.1 Testing Functions 39

14.2 Defensive Programming 39

14.3 Validator 39

15 Troubleshooting and Debugging **41**

15.1 Finding Help 41

15.2 Debugging 41

16 Establishing Workstation **43**

16.1 Required Installation 43

16.2 Recommended Installation 43

16.3 Optional Installation 43

16.4 Asset Locations 43

17 Considerations when Selecting Tools **45**

17.1 General 45

17.2 Languages 46

17.3 R Packages 46

17.4 Database 46

18 Growing a Team **47**

18.1 Recruiting 47

18.2 Training to Data Science 47

18.3 Bridges Outside the Team 47

A Git & GitHub **49**

B Presentations **51**

B.1 CDW 51

B.2 REDCap 51

B.3 Reproducible Research & Visualization 51

B.4 Data Management 52

B.5 GitHub 52

B.6 Software 52
B.7 Architectures 52
B.8 Components 53

C Scratch Pad of Loose Ideas **55**
C.1 Chapters & Sections to Form 55

D Introduction **57**

Chapter 1

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 2

Architecture Principles

2.1 Encapsulation

2.2 Leverage team member's strenghts & avoid weaknesses

1. Focused code files
2. Metadata for content experts

2.3 Scales

1. Single source & single analysis
2. Multiple sources & multiple analyses

2.4 Consistency

1. Across Files {#consistency-files}
2. Across Languages
3. Across Projects

2.5 Defensive Style

Some of these may not be not be an 'architecture', but it guides many of our architectural principles.

1. **Qualify functions**.{#qualify-functions} Try to prepend each function with its package. Write `dplyr::filter()` instead of `filter()`. When two packages contain public functions with the same name, the package that was most recently called with `library()` takes precedent. When multiple R files are executed, the packages' precedents may not be predictable. Specifying the package eliminates the ambiguity, while also making the code easier to follow. For this reason, we recommend that almost all R files contain a 'load-packages' chunk.

Chapter 3

Prototypical File

As stated before, in Consistency across Files, using a consistent file structure can (a) improve the quality of the code because the structure has been proven over time to facilitate good practices and (b) allow your intentions to be more clear to teammates because they are familiar with the order and intentions of the chunks.

We use the term "chunk" for a section of code because it corresponds with knitr terminology (Xie, 2015), and in many analysis files (as opposed to manipulation files), the chunk of our R file connects to a knitr Rmd file.

3.1 Clear Memory

Before the initial chunk many of our files clear the memory of variables from previous run. This is important when developig and debugging because it prevents previous runs from contaminating subsequent runs. However it has little effect during production; we'll look at manipulation files separately from analysis files.

Manipulation R files are `source`d with the argument `local=new.env()`. The file is executed in a fresh environment, so there are no variables to clear. Analysis R files are typically called from an Rmd file's `knitr::read_chunk()`, and code positioned above the first chunk is not called by knitr [1].

However typically do not clear the memory in R files that are `source`d in the same environment as the caller, as it will interfere with the caller's variables.

```
rm(list = ls(all.names = TRUE))
```

[1] Read more about knitr's code externalization

3.2 Load Sources

In the first true chunk, `source` any R files containing global variables and functions that the current file requires. For instance, when a team of statisticians is producing a large report containing many analysis files, we define many of the graphical elements in a single file. This sourced file defines common color palettes and graphical functions so the cosmetics are more uniform across analyses.

We prefer not to have `source`d files perform any real action, such as importing data or manipulating a file. One reason is because it is difficult to be consistent about the environmental variables when the sourced file's functions are run. A second reason is that it more cognitively difficult to understand how the files are connected.

When the sourced file contains only function definitions, these operations can be called at any time in the current file with much tighter control of which variables are modfied. A bonus of the discipline of defining functions (instead of executing functions) is that the operations are typically more robust and generalizable.

Keep the chunk even if no files are sourced. An empty chunk is instructive to readers trying to determine if any files are sourced. This applies recommendation applies to all the chunks discussed in this chapter. As always, your team should agree on its own set of standards.

```
# ---- load-sources ------------------------------------------------------------
base::source(file="./analysis/common/display-1.R")      # Load common graphing functio
```

3.3 Load Packages

The 'load-packages' chunk declares required packages near the file's beginning for three reasons. First, a reader scanning the file can quickly determine its dependencies when located in a single chunk. Second, if your machine is lacking a required package, it is best to know early[2]. Third, this style mimics a requirement of other languages, such as declaring headers at the top of a C++ file.

As discussed in the previous qualify all functions section, we recommend that functions are qualified with their package (*e.g.*, `foo::bar()` instead of merely `bar()`). Consequently, the 'load-packages' chunk calls `requireNamespace()` more frequently than `library()`. `requireNamespace()` verifies the package is

[2]The error message "Error in library(foo) : there is no package called 'foo' " is easier to understand than "Error in bar() : could not find function 'bar' " thrown somewhere in the middle of the file; this check can also illuminate conflicts arising when two packages have a `bar()` function. See McConnell 2004 Section qqq for more about the 'fail early' principle.

available on the local machine, but does not load it into memory; `library()` verifies the package is available, and then loads it.

`requireNamespace()` is not used in several scenarios.

1. Core packages (*e.g.*, 'base' and 'stats') are loaded by R in most default installations. We avoid unnecessary calls like `library(stats)` because they distract from more important features.
2. Obvious dependencies are not called by `reqiureNamespace()` or `library()` for similar reasons, especially if they are not called directly. For example 'tidyselect' is not listed when 'tidyr' is listed.
3. The "pipe" function (declared in the `magrittr' package , *i.e.*,`%>%`) `is attached with`import::from(magrittr, "%>%")'. This frequently-used function called be called throughout the execution without qualification.
4. Compared to manipulation files, our analysis files tend to use many functions in a few concentrated packages so conflicting function names are less common. Typical packages used in analysis are 'ggplot2' and 'lme4'.

The `sourced` files above may load their own packages (by calling `library()`). It is important that the `library()` calls in this file follow the 'load-sources' chunk so that identically-named functions (in different packages) are called with the correct precedent. Otherwise identically-named functions will conflict in the namespace with hard-to-predict results.

Read R Packages for more about `library()`, `requireNamespace()`, and their siblings, as well as the larger concepts such as attaching functions into the search path.

Here are packages found in most of our manipulation files. Notice the lesser-known packages have a quick explanation; this helps maintainers decide if the declaration is still necessary. Also notice the packages distributed outside of CRAN (*e.g.*, GitHub) have a quick commented line to help the user install or update the package.

```
# ---- load-packages -----------------------------------------------------------
import::from(magrittr, "%>%" )

requireNamespace("readr"     )
requireNamespace("tidyr"     )
requireNamespace("dplyr"     )
requireNamespace("config"     )
requireNamespace("checkmate" ) # Asserts expected conditions
requireNamespace("OuhscMunge") # remotes::install_github(repo="OuhscBbmc/OuhscMunge")
```

3.4 Declare Globals

When values are repeatedly used within a file, consider dedicating a variable so it's defined and set only once. This is also a good place for variables that are used only once, but whose value are central to the file's mission. Typical variables in our 'declare-globals' chunk include data file paths, data file variables, color palettes, and values in the *config* file.

The config file can coordinate a static variable across multiple files. Centrally

```
# ---- declare-globals ---------------------------------------------------------
# Constant values that won't change.
config                          <- config::get()
path_db                         <- config$path_database

# Execute to specify the column types.  It might require some manual adjustment (eg do
#   OuhscMunge::readr_spec_aligned(config$path_subject_1_raw)
col_types <- readr::cols_only(
  subject_id          = readr::col_integer(),
  county_id           = readr::col_integer(),
  gender_id           = readr::col_double(),
  race                = readr::col_character(),
  ethnicity           = readr::col_character()
)
```

3.5 Load Data

3.6 Tweak Data

3.7 (Unique Content)

3.8 Verify Values

3.9 Specify Output Columns

3.10 Save to Disk or Database

Chapter 4

Prototypical Repository

https://github.com/wibeasley/RAnalysisSkeleton

4.1 Root

1. `config.R` is simply a plain-text yaml file read by the config package. It is great when a value has to be coordinated across multiple file

```
default:
# To be processed by Ellis lanes
path_subject_1_raw:  "data-public/raw/subject-1.csv"
path_mlm_1_raw:      "data-public/raw/mlm-1.csv"

# Central Database (produced by Ellis lanes).
path_database:       "data-public/derived/db.sqlite3"

# Analysis-ready datasets (produced by scribes & consumed by analyses).
path_mlm_1_derived:  "data-public/derived/mlm-1.rds"

# Metadata
path_annotation:     "data-public/metadata/cqi-annotation.csv"

# Logging errors and messages from automated execution.
path_log_flow:       !expr strftime(Sys.time(), "data-unshared/log/flow-%Y-%m-%d--%H-%M-%S.l

# time_zone_local       :  "America/Chicago" # Force local time, in case remotely run.

# ---- Validation Ranges & Patterns ----
range_record_id         : !expr c(1L, 999999L)
```

```
range_dob               : !expr c(as.Date("2010-01-01"), Sys.Date() + lubridate::d
range_datetime_entry    : !expr c(as.POSIXct("2019-01-01", tz="America/Chicago"),
max_age                 : 25
pattern_mrn_centricity  : "^\\d{16}$"        # The 64-bit int is more easily valid
```

2. `flow.R`
3. `README.md`
4. `*.Rproj`

4.2 Analysis

4.3 Data Public

1. Raw
2. Derived
3. Metadata
4. Database
5. Original

4.4 Data Unshared

4.5 Documentation

4.6 Manipulation

4.7 Stitched Output

4.8 Utility

Chapter 5

Data at Rest

5.1 Data States

1. Raw
2. Derived
    1. Project-wide File on Repo
    2. Project-wide File on Protected File Server
    3. User-specific File on Protected File Server
    4. Project-wide Database
3. Original

5.2 Data Containers

1. csv
2. rds
3. SQLite
4. Central Enterprise database
5. Central REDCap database
6. Containers to avoid for raw/input
    1. Proprietary like xlsx, sas7bdat

Chapter 6

Patterns

6.1 Ellis

6.1.1 Purpose

To incorporate outside data source into your system safely.

6.1.2 Philosophy

- Without data immigration, all warehouses are useless. Embrace the power of fresh information in a way that is:
  - repeatable when the datasource is updated (and you have to refresh your warehouse)
  - similar to other Ellis lanes (that are designed for other data sources) so you don't have to learn/remember an entirely new pattern. (Like Rubiks cube instructions.)

6.1.3 Guidelines

- Take small bites.
  - Like all software development, don't tackle all the complexity the first time. Start by processing only the important columns before incorporating move.
  - Use only the variables you need in the short-term, especially for new projects. As everyone knows, the variables from the upstream source can change. Don't spend effort writing code for variables you won't need for a few months/years; they'll likely change before you need them.

  - After a row passes through the `verify-values` chunk, you're accountable for any failures it causes in your warehouse. All analysts know that external data is messy, so don't be surprised. Sometimes I'll spend an hour writing an Ellis for 6 columns.

- Narrowly define each Ellis lane. One code file should strive to (a) consume only one CSV and (b) produce only one table. Exceptions include:

  1. if multiple input files are related, and really belong together (*e.g.*, one CSV per month, or one CSV per clinic). This scenario is pretty common.
  2. if the CSV should legitimately produce two different tables after munging. This happens infrequently, such as one warehouse table needs to be wide, and another long.

6.1.4 Examples

- https://github.com/wibeasley/RAnalysisSkeleton/blob/master/manipulation/te-ellis.R
- https://github.com/wibeasley/RAnalysisSkeleton/blob/master/manipulation/
- https://github.com/OuhscBbmc/usnavy-billets/blob/master/manipulation/survey-ellis.R

6.1.5 Elements

1. **Clear memory** In scripting languages like R (unlike compiled languages like Java), it's easy for old variables to hang around. Explicitly clear them before you run the file again.

```
rm(list=ls(all=TRUE)) #Clear the memory of variables from previous run. This is no
```

2. **Load Sources** In R, a `source()`d file is run to execute its code. We prefer that a sourced file only load variables (like function definitions), instead of do real operations like read a dataset or perform a calculation. There are many times that you want a function to be available to multiple files in a repo; there are two approaches we like. The first is collecting those common functions into a single file (and then sourcing it in the callers). The second is to make the repo a legitimate R package.

   The first approach is better suited for quick & easy development. The second allows you to add documention and unit tests.

```
# ---- load-sources ------------------------------------------------------------
source("./manipulation/osdh/ellis/common-ellis.R")
```

3. **Load Packages** This is another precaution necessary in a scripting language. Determine if the necessary packages are available on the machine. Avoiding attaching packages (with the `library()` function) when possible. Their functions don't need to be qualified (*e.g.*, `dplyr::intersect()`) and could cause naming conflicts. Even if you can guarantee they don't conflict with packages now, packages could add new functions in the future that do conflict.

```
# ---- load-packages -----------------------------------------------------------
# Attach these package(s) so their functions don't need to be qualified: http://r-pkgs.had.c
library(magrittr            , quietly=TRUE)
library(DBI                 , quietly=TRUE)

# Verify these packages are available on the machine, but their functions need to be qualifi
requireNamespace("readr"        )
requireNamespace("tidyr"        )
requireNamespace("dplyr"        ) # Avoid attaching dplyr, b/c its function names conflict u
requireNamespace("testit")
requireNamespace("checkmate")
requireNamespace("OuhscMunge") #devtools::install_github(repo="OuhscBbmc/OuhscMunge")
```

4. **Declare Global Variables and Functions**. This includes defining the expected column names and types of the data sources; use `readr::cols_only()` (as opposed to `readr::cols()`) to ignore any new columns that may be been added since the dataset's last refresh.

```
# ---- declare-globals ---------------------------------------------------------
```

5. **Load Data Source(s)** Read all data (*e.g.*, database table, networked CSV, local lookup table). After this chunk, no new data should be introduced. This is for the sake of reducing human cognition load. Everything below this chunk is derived from these first four chunks.

```
# ---- load-data ---------------------------------------------------------------
```

6. **Tweak Data**

```
# ---- tweak-data --------------------------------------------------------------
```

7. **Body of the Ellis**

8. **Verify**

```
# ---- verify-values -----------------------------------------------------------
county_month_combo   <- paste(ds$county_id, ds$month)
checkmate::assert_character(county_month_combo, pattern  ="^\\d{1,2} \\d{4}-\\d{2}-\\d{2}$",
```

9. **Specify Columns** Define the exact columns and order to upload to the database. Once you import a column into a warehouse that multiple people are using, it's tough to remove it.

```
# ---- specify-columns-to-upload ----------------------------------------------
```

10. **Welcome** into your warehouse. Until this chunk, nothing should be persisted.

```
# ---- upload-to-db -----------------------------------------------------------
# ---- save-to-disk -----------------------------------------------------------
```

6.2 Arch

6.3 Ferry

6.4 Scribe

6.5 Analysis

6.6 Presentation -Static

6.7 Presentation -Interactive

6.8 Metadata

Chapter 7

Security & Private Data

7.1 File-level permissions

7.2 Database permissions

7.3 Public & Private Repositories

7.3.1 Scrubbing GitHub history

Occassionaly files may be committed to your git repository that need to be removed completely. Not just from the current collections of files (*i.e.*, the branch's head), but from the entire history of the repo.

Scrubbing is require typically when (a) a sensitive file has been accidentally commited and pushed to GitHub, or (b) a huge file has bloated your repository and disrupted productivity.

The two suitable scrubbing approaches both require the command line. The first is the `git-filter-branch` command within git, and the second is the BFG repo-cleaner. We use the second approach, which is [recommended by GitHub]; it requires 15 minutes to install and configure from scratch, but then is much easier to develop against, and executes much faster.

The bash-centric steps below remove any file*s* from the repo history called 'monster-data.csv' from the 'bloated' repository.

1. If the file contains passwords, change them immediately.
2. Delete 'monster-data.csv' from your branch and push the commit to GitHub.

3. Ask your collaborators to push any outstanding commits to GitHub and delete their local copy of the repo. Once scrubbing is complete, they will re-clone it.
4. Download and install the most recent Java JRE from the Oracle site.
5. Download the most recent jar file from the BFG site to the home directory.
6. Clone a fresh copy of the repository in the user's home directory. The `--mirror` argument avoids downloading every file, and downloads only the bookkeeping details required for scrubbing.

```
cd ~
git clone --mirror https://github.com/your-org/bloated.git
```

7. Remove all files (in any directory) called 'monster-data.csv'.

```
java -jar bfg-*.jar --delete-files monster-data.csv bloated.git
```

8. Reflog and garbage collect the repo.

```
cd bloated.git
git reflog expire --expire=now --all && git gc --prune=now --aggressive
```

9. Push your local changes to the GitHub server.

```
git push
```

10. Delete the bfg jar from the home directory.

```
cd ~
rm bfg-*.jar
```

11. Ask your collaborators to reclone the repo to their local machine. It is important they restart with a fresh copy, so the once-scrubbed file is not reintroduced into the repo's history.
12. If the file contains sensitive information, like passwords or PHI, ask GitHub to refresh the cache so the file's history isn't accessible through their website, even if the repo is private.

7.3.1.0.1 Resources

- BFG Repo-Cleaner site
- Additional BFG instructions
- GitHub Sensitive Data Removal Policy

Chapter 8

Automation

8.1 Flow File in R

8.2 Makefile

8.3 SSIS

8.4 cron Jobs & Task Scheduler

8.5 Sink Log Files

Chapter 9

Scaling Up

9.1 Data Storage

1. Local File vs Conventional Database vs Redshift
2. Usage Cases

9.2 Data Processing

1. R vs SQL
2. R vs Spark

Chapter 10

Parallel Collaboration

10.1 Social Contract

1. Issues
2. Organized Commits & Coherent Diffs
3. Branch & Merge Strategy

10.2 Code Reviews

1. Daily Reviews of PRs
2. Periodic Reviews of Files

10.3 Remote

1. Headset & sharing screens

10.4 Loose Notes

10.4.1 GitHub

1. Review your diffs before committing. Check for things like accidental deletions and debugging code that should be deleted (or at least commented out).
2. Keep chatter to a minimum, especially on projects with 3+ people being notified of every issue post.

3. When encountering a problem,

    - Take as much ownership as reasonable. Don't merely report there's an error.

    - If you can't figure it out, ask the question and describe it well.
        - what low-level file & line of code threw the error.
        - how you have tried to solve it.
    - If there's a questionable line/chunk of code, trace its origin. Not for the sake of pointing the finger at someone, but for the sake of understanding its origin and history.

10.4.2 Common Code

This involves code/files that multiple people use, like the REDCap arches.

1. Run the file before committing it. Run common downstream files too (*e.g.*, if you make a change to the arch, also run the funnel).
2. If an upstream variable name must change, alert people. Post a GitHub issue to announce it. Tell everyone, and search the repo (ctrl+shift+f in RStudio) to alert specific people who might be affected.

Chapter 11

Documentation

11.1 Team-wide

11.2 Project-specific

11.3 Dataset Origin & Structure

11.4 Issues & Tasks

11.5 Flow Diagrams

11.6 Setting up new machine

(example)

Chapter 12

Style Guide

Using a consistent style across your projects can increase the overhead as your data science team discusses options, decides on a good choice, and develops in compliant code. But like in most themes in this document, the cost is worth the effort. Unforced code errors are reduced when code is consistent, because mistake-prone styles are more apparent.

For the most part, our team follows the tidyverse style. Here are some additional conventions we attempt to follow.

12.1 ggplot2

The expressiveness of ggplot2 allows someone to quickly develop precise scientific graphics. One graph can be specified in many equivalent styles, which increases the opportunity for confusion. We formalized much of this style while writing a textbook for introductory statistics; the 200+ graphs and their code is publically available.

12.1.1 Order of commands

ggplot2 is essentially a collection of functions combined with the `+` operator. Publication graphs common require at least 20 functions, which means the functions can sometimes be redundant or step on each other toes. The family of functoins should follow a consistent order ideally starting with the more important structural functions and ending with the cosmetic functions. Our preference is:

1. `ggplot2()` is the primary function to specify the default dataset and aesthetic mappings. Many arguments can be passed to `aes()`, and we prefer to follow an order consistent with the `scale_*()` order below.
2. `geom_*()` creates the *geome*tric elements that reperesent the data. Unlike most categories in this list, the order matters. Geoms specified first are drawn first, and therefore can be obscured by subsequent geoms.
3. `scale_*()` describes how a dimension of data (specified in `aes()`) is translated into a visual element. We specify the dimensions in descending order of (typical) importance: `x`, `y`, `group`, `color`, `fill`, `size`, `radius`, `alpha`, `shape`, `linetype`.
4. `coord_*()`
5. `facet_*()` and `label_*()`
6. `theme()`
7. `labs()`

12.1.2 Gotchas

Here are some common mistakes we see not-so-infrequently (even sometimes in our own code).

1. Call `coord_*()` to restrict the plotted x/y values, not `scale_*()` or `lims()`/`xlim()`/`ylim()`. `coord_*()` zooms in on the axes, so extreme values essentially fall off the page; in contrast, the latter three functions essentially remove the values from the dataset. The distinction does not matter for a simple bivariate scatterplot, but likely will mislead you and the viewer in two common scenarios. First, a call to `geom_smooth()` (*e.g.*, that overlays a loess regression curve) ignore the extreme values entirely; consequently the summary location will be misplaced and its standard errors too tight. Second, when a line graph or spaghetti plots contains an extreme value, it is sometimes desirable to zoom in on the the primary area of activity; when calling `coord_*()`, the trend line will leave and return to the plotting panel (which implies points exist which do not fit the page), yet when calling the others, the trend line will appear interrupted, as if the extreme point is a missing value.

Chapter 13

Publishing Results

13.1 To Other Analysts

13.2 To Researchers & Content Experts

13.3 To Technical-Phobic Audiences

Chapter 14

Testing, Validation, & Defensive Programming

14.1 Testing Functions

14.2 Defensive Programming

1. Throwing errors

14.3 Validator

1. Benefits for Analysts
2. Benefits for Data Collectors

Chapter 15

Troubleshooting and Debugging

15.1 Finding Help

1. Within your group (eg, Thomas and REDCap questions)
2. Within your university (eg, SCUG)
3. Outside (eg, Stack Overflow; GitHub issues)

15.2 Debugging

1. `traceback()`, `browser()`, etc

Chapter 16

Establishing Workstation

https://github.com/OuhscBbmc/RedcapExamplesAndPatterns/blob/master/DocumentationGlobal/ResourcesInstallation.md

16.1 Required Installation

16.2 Recommended Installation

16.3 Optional Installation

16.4 Asset Locations

Chapter 17

Considerations when Selecting Tools

17.1 General

17.1.1 The Component's Goal

While disussing the advantages and disadvanages of tools, a colleague once said, "Tidyverse packages don't do anything that I can't already do in Base R, and sometimes it even requires more lines of code". Regardless if I agree, I feel these two points are irrelevant. Sometimes the advantage of a tool isn't to expand existing capabilities, but rather to facilitate development and maintaince for the same capability.

Likewise, I care less about the line count, and more about the readability. I'd prefer to maintain a 20-line chunk that is familar and readable than a 10-line chunk with dense phrases and unfamiliar functions. The bottleneck for most of our projects is human time, not execution time.

17.1.2 Current Skillset of Team

17.1.3 Desired Future Skillset of Team

17.1.4 Skillset of Audience

17.2 Languages

17.3 R Packages

17.4 Database

Chapter 18

Growing a Team

18.1 Recruiting

18.2 Training to Data Science

1. Starting with a Researcher
2. Starting with a Statistician
3. Starting with a DBA
4. Starting with a Software Developer

18.3 Bridges Outside the Team

1. Monthly User Groups
2. Annual Conferences

Appendix A

Git & GitHub

Jenny Bryan and Jim Hester have published a thorough description of using Git from a data scientist's perspective (Happy Git and GitHub for the useR), and we recommend following their guidance. It is consistent with our approach, with a few exceptions noted below. A complementary resource is *Team Geek*, which has insightful advice for the human and collaborative aspects of version control.

Appendix B

Presentations

Here is a collection of presentations by the BBMC and friends that may help demonstrate concepts discussed in the previous chapters.

B.1 CDW

1. **prairie-outpost-public**: Documentation and starter files for OUHSC's Clinical Data Warehouse.
2. **OUHSC CDW**

B.2 REDCap

1. **REDCap Systems Integration**. REDCap Con 2015, Portland, Oregon.
2. **Literate Programming Patterns and Practices with REDCap** REDCap Con 2014, Park City, Utah.
3. **Interacting with the REDCap API using the REDCapR Package** REDCap Con 2014, Park City, Utah.
4. **Optimizing Study Management using REDCap, R, and other software tools**. SCUG 2013.

B.3 Reproducible Research & Visualization

1. **Building pipelines and dashboards for practitioners**: Mobilizing knowledge with reproducible reporting. Displaying Health Data Colloquium 2018, University of Victoria.
2. **Interactive reports and webpages with R & Shiny**. SCUG 2015.

3. **Big data, big analysis: a collaborative framework for multistudy replication**. Conventional of Canadian Psychological Association, Victoria BC, 2016.
4. **WATS: wrap-around time series**: Code to accompany WATS Plot article, 2014.

B.4 Data Management

1. **BBMC Validator**: catch and communicate data errors. SCUG 2016.
2. **Text manipulation with Regular Expressions, Part 1 and Part 2**. SCUG 2016.
3. **Time and Effort Data Synthesis**. SCUG 2015.

B.5 GitHub

1. **Scientific Collaboration with GitHub**. OU Bioinformatics Breakfast Club 2015.

B.6 Software

1. **REDCapR**: Interaction Between R and REDCap.
2. **OuhscMunge**: Data manipulation operations commonly used by the Biomedical and Behavioral Methodology Core within the Department of Pediatrics of the University of Oklahoma Health Sciences Center.
3. **codified**: Produce standard/formalized demographics tables.
4. **usnavy billets**: Optimally assigning naval officers to billets.

B.7 Architectures

1. Linear Pipeline of the R Analysis Skeleton

   .

2. Many-to-many Pipeline of the R Analysis Skeleton

   .

3. Immunization transfer

   .

4. IALSA: A Collaborative Modeling Framework for Multi-study Replication

   .

5. POPS: Automated daily screening eligibility for rare and understudied prescriptions.

   .

B.8 Components

1. **Customizing display tables: using css with DT and kableExtra**. SCUG 2018.
2. **yaml** and **expandable trees** that selectively show subsets of hierarchy, 2017.

Appendix C

Scratch Pad of Loose Ideas

C.1 Chapters & Sections to Form

1. Tools to Consider
    1. tidyverse
    2. odbc
2. ggplot2
    1. use factors for explanatory variables when you want to keep the order consistent across graphs. (genevamarshall)
3. styles
    1. variable names: within a variable name order from big to small terms (lexigraphical scoping) (thomasnwilson)
4. public reports (and dashboards)
    1. when developing a report for a external audience (ie, people outside your immediate research team), choose one or two pals who are unfamilar with your aims/methods as an impromptu focus group. Ask them what things need to be redesigned/reframed/reformated/further-explained. (genevamarshall)
        1. plots
        2. plot labels/axes
        3. variable names
        4. units of measurement (eg, proportion vs percentage on the y axis)

Appendix D

Introduction

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter D. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 2.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure D.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table D.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2019) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

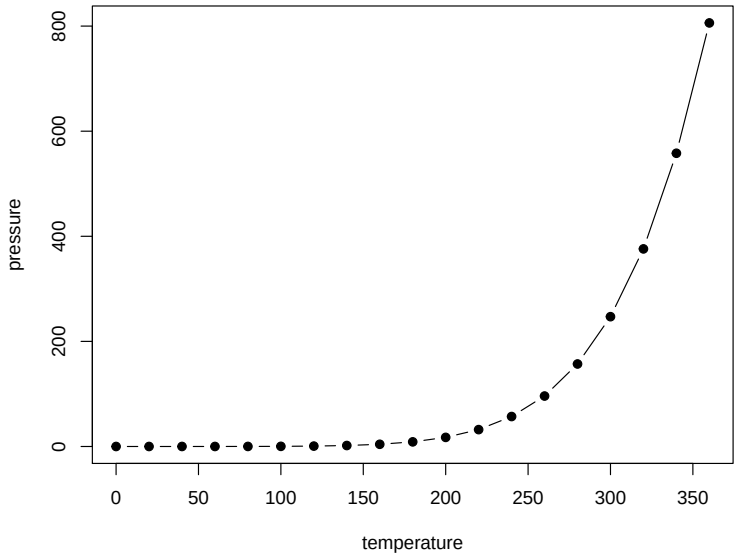

Figure D.1: Here is a nice figure!

Table D.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.14.