Collaborative Data Science Practices

Will Beasley

2020-01-24

Contents

Chapter 1

Introduction

This collection of documents describe practices used by the OUHSC BBMC in our analytics projects.

Chapter 2

Coding Principles

2.1 Simplify

2.1.1 Data Types

Use the simplest data type reasonable. A simpler data type is less likely contain unintended values. As we have seen, a string variable called `gender` can simultaneously contain the values "m", "f", "F", "Female", "MALE", "0", "1", "2", "Latino", "", and `NA`. On the other hand, a boolean variable `gender_male` can be only `FALSE`, `TRUE`, and `NA`.[1]

SQLite does not have a dedicated datatype, so you must resort to storing it as `0`, `1` and `NULL` values. Because a caller can't assume that an obstensible boolean SQLite variable contains only those three values, the variable should be checked.]

Once you have cleaned a variable in your initial ETL files (like an Ellis), lock it down so you do not have to spend time in the downstream files verifying that no bad values have been introduced. As a small bonus, simpler data types are typically faster, consume less memory, and translate more cleanly across platforms.

Within R, the preference for numeric-ish variables is

1. `logical`/boolean/bit,
2. `integer`,
3. `bit64::integer64`, and
4. `numeric`/double-precision floats.

[1]The equivalent of R's `logical` data type is called a `bit` in SQL Server, and a `boolean` in Postgres and MySQL.

The preference for categorical variables is

1. `logical`/boolean/bit,
2. `factor`, and
3. `character`.

2.1.2 Categorical Levels

When a boolean variable would be too restrictive and a factor or character is required, choose the simplest representation. Where possible:

1. Use only lower case (*e.g.*, 'male' instead of 'Male' for the `gender` variable).
2. avoid repeating the variable in the level (*e.g.*, 'control' instead of 'control condition' for the `condition` variable).

2.1.3 Recoding

Almost every project recodes many variables. Choose the simplest function possible. The functions at the top are much easier to read and harder to mess up.

1. **Leverage exising booleans:** Suppose you have the logical variable `gender_male` (which can be only `TRUE`, `FALSE`, or `NA`). Writing `gender_male == TRUE` or `gender_male == FALSE` will evaluate to a boolean –that's unnecessary because `gender_male` is already a boolean.

   1. *Testing for `TRUE`*: use the variable by itself (*i.e.*, `gender_male` instead of `gender_male == TRUE`).
   2. *Testing for `FALSE`*: use `!`. Write `!gender_male` instead of `gender_male == FALSE` or `gender_male != TRUE`.

2. **`dplyr::coalesce()`**: The function evaluates a single variable and replaces `NA` with values from another variable.

   A coalesce like

   ```
   visit_completed = dplyr::coalesce(visit_completed, FALSE)
   ```

   is much easier to read and not mess up than

   ```
   visit_completed = dplyr::if_else(!is.na(visit_completed), visit_completed, FALSE)
   ```

3. **`dplyr::na_if()`** transforms a nonmissing value into an NA.

   Recoding missing values like

```
birth_apgar = dplyr::na_if(birth_apgar, 99)
```

is easier to read and not mess up than

```
birth_apgar = dplyr::if_else(birth_apgar == 99, NA_real_, birth_apgar)
```

4. `<=` (or a similar comparison operator): Compare two quantities to output a boolean variable.
5. **`dplyr::if_else()`**: The function evaluates a single boolean variable. The output branches to only three possibilities: condition is (a) true, (b) false, or (c) (optionally) `NA`. An advantage over `<=` is that `NA` values can be specified directly.

```
date_start <- as.Date("2017-01-01")

# If a missing month element needs to be handled explicitly.
stage       = dplyr::if_else(date_start <= month, "pre", "post", missing = "missing-month")

# Otherwise a simple boolean output is sufficient.
stage_post  = (date_start <= month)
```

6. **`base::cut()`*: The function evaluations only a single numeric variable. It's range is cut into different segments/categories on the one-dimensional number line. The output branches to single discrete value (either a factor-level or an integer).
7. *`dplyr::recode()`**: The function evaluates a single integer or character variable. The output branches to a single discrete value.
8. **lookup table**: It feasible recode 6 levels of race directly in R. It's less feasible to recode 200 provider names. Specify the mapping in a csv, `readr` the csv to a data.frame, and left-join it.
9. **`dplyr::case_when()`**: The function is the most complicated because it can evaluate multiple variables. Also, multiple cases can be true, but only the first output is returned. This 'water fall' execution helps in complicated scenarios, but is overkill for most.

2.2 Defensive Style

2.2.1 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.

See the Google Style Guide for more about qualifying functions

Chapter 3

Architecture Principles

3.1 Encapsulation

3.2 Leverage team member's strenghts & avoid weaknesses

3.2.1 Focused code files

3.2.2 Metadata for content experts

3.3 Scales

3.3.1 Single source & single analysis

3.3.2 Multiple sources & multiple analyses

3.4 Consistency

3.4.1 Across Files

3.4.2 Across Languages

3.4.3 Across Projects

Chapter 4

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.

4.1 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

4.2 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.

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

4.3 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) and follows the tidyverse style guide.

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 `source`d 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.

```r
# ---- 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")
```

4.4 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

```r
# ---- 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()
)
```

4.5 Load Data

All data ingested by this file occurs in this chunk. We like to think of each file as a linear pipe with a single point of input and single point of output. Although it is possible for a file to read data files on any line, we recommend avoiding this sprawl because it is more difficult for humans to understand. If the software developer is a deist watchmaker, the file's fate has been sealed by the end of this chunk. This makes is easier for a human to reason to isolate problems as either existing with (a) the incoming data or (b) the calculations on that data.

Ideally this chunk consumes data from either a plain-text csv or a database.

Many capable R functions and packages ingest data. We prefer the tidyverse readr for reading conventional files; its younger cousin, vroom has some nice advantages when working with larger files and some forms of jagged rectangles[3]. Depending on the file format, good packages to consider are data.table, haven, readxl, openxlsx, arrow, jsonlite, fst, yaml, and rio.

[3]Say a csv has 20 columns, but a row has missing values for the last five columns. Instead of five successive commas to indicate five empty cells exist, a jagged rectangle simply ends after the last nonmissing value. vroom infers the missing values correctly, while some other packages do not.

When used in an Ellis, this chunk likely consumees a flat file like a csv with data or metadata. When used in a Ferry, Arch, or Scribe, this chunk likely consumes a database table. When used in an Analysis file, this chunk likely cosumes a database table or rds (*i.e.*, a compressed R data file).

In some large-scale scenarios, there may be a series of datasets that cannot be held in RAM simultaneously. Our first choice is to split the R file so each new file has only a subset of the datasets –in other words, the R file probably was given too much responsibility. Occassionaly the multiple datasets need to be considered at once, so splitting the R file is not a option. In these scenarios, we prefer to upload all the datasets to a database, which is better manipulating datasets too large for RAM.

An R solution may be to losen the restriction that dataset enter the R file only during the 'load-data' chunk. Once a dataset is processed and no longer needed, `rm()` *rem*oves it from RAM. Now another dataset can be read from a file and manipulated.

loose scrap: the chunk reads 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.

4.6 Tweak Data

loose scrap: It's best to rename the dataset (a) in a single place and (b) early in the pipeline, so the bad variable are never referenced.

```
# OuhscMunge::column_rename_headstart(ds) # Help write `dplyr::select()` call.
ds <-
  ds %>%
  dplyr::select(    # `dplyr::select()` drops columns not included.
    subject_id,
    county_id,
    gender_id,
    race,
    ethnicity
  ) %>%
  dplyr::mutate(

  )  %>%
  dplyr::arrange(subject_id) # %>%
  # tibble::rowid_to_column("subject_id") # Add a unique index if necessary
```

4.7 (Unique Content)

This section represents all the chunks between tweak-data and verify-values. These chunks contain most of of the file's creativity and contribution. In a sense, the structure of the first and last chunks allow these middle chunks to focus on concepts instead of plumbing.

For simple files like the ellis of a metadata file, may not even need anything here. But complex analysis files may have 200+ lines distributed across a dozen chunks. We recommend that you create dedicate a chunk to each conceptual stage. If one starts to contain more than ~20 lines, consider if a more granular organization would clarify the code's intent.

4.8 Verify Values

Running `OuhscMunge::verify_value_headstart(ds)` will

```
# ---- verify-values -----------------------------------------------------------
# Sniff out problems
# OuhscMunge::verify_value_headstart(ds)
checkmate::assert_integer(  ds$county_month_id    , any.missing=F , lower=1, upper=308
checkmate::assert_integer(  ds$county_id          , any.missing=F , lower=1, upper=77
checkmate::assert_date(     ds$month              , any.missing=F , lower=as.Date("201
checkmate::assert_character(ds$county_name        , any.missing=F , pattern="^.{3,12}$
checkmate::assert_integer(  ds$region_id          , any.missing=F , lower=1, upper=20
checkmate::assert_numeric(  ds$fte                , any.missing=F , lower=0, upper=40
checkmate::assert_logical(  ds$fte_approximated   , any.missing=F
checkmate::assert_numeric(  ds$fte_rolling_median , any.missing=T , lower=0, upper=40

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

4.9 Specify Output Columns

This chunk:

1. verifies these variables exist before uploading,
2. documents (to troubleshooting developers) these variables are a product of the file, and
3. reorders the variables to match the expected structure.

Variable order is especially important for the database engines/drivers that ignore the variable name, and use only the variable position.

We use the term 'slim' because typically this output has fewer variables than the full dataset processed by the file.

If you doubt the variable will be needed downstream, leave it in the `dplyr::select()`, but commented out. If someone needs it in the future, they'll easily determine where it might come from, and then uncomment the line (and possibly modify the database table). Once you import a column into a warehouse that multiple people are using, it can be tough to remove without breaking their code.

This chunk follows verify-values because sometimesyou want to check the validity of variables that are not consumed downstream. These variables are not important themselves, but an illegal value may reveal a larger problem with the dataset.

```
# Print colnames that `dplyr::select()`  should contain below:
#   cat(paste0("    ", colnames(ds), collapse=",\n"))

# Define the subset of columns that will be needed in the analyses.
#   The fewer columns that are exported, the fewer things that can break downstream.

ds_slim <-
  ds %>%
  # dplyr::slice(1:100) %>%
  dplyr::select(
    subject_id,
    county_id,
    gender_id,
    race,
    ethnicity
  )

ds_slim
```

4.10 Save to Disk or Database

4.11 Additional Resources

- (Colin Gillespie, 2017), particularly the "Efficient input/output" chapter.

Chapter 5

Prototypical Repository

https://github.com/wibeasley/RAnalysisSkeleton

5.1 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::c
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 vali
```

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

5.2 Analysis

5.3 Data Public

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

5.4 Data Unshared

5.5 Documentation

5.6 Manipulation

5.7 Stitched Output

5.8 Utility

Chapter 6

Data at Rest

6.1 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

6.2 Data Containers

6.2.1 csv

When exchanging data between two different systems, the preferred format is frequently plain text. As opposed to proprietary formats like xlsx or sas7bdat, the file is easily opened and parsable by most statistical software, and even conventional text editors.

Incoming files should be plain-text csv –not Excel or other proprietary or binary format.

6.2.2 rds

6.2.3 SQLite

6.2.4 Central Enterprise database

6.2.5 Central REDCap database

6.2.6 Containers to avoid for raw/input

1. Proprietary like xlsx, sas7bdat

6.3 Storage Conventions

6.3.1 All Sources

Across all file formats, these conventions usually work best.

1. **date format**: use `YYYY-MM-DD` (ISO-8601)
2. **time format**: use `HH:MM` or `HH:MM:SS`. Use a leading zero from midnight to 9:59am, with a colon separating hours, minutes, and seconds (*i.e.*, **0**9:59am)
3. **patient names**: separate the `name_last`, `name_first`, and `name_middle` when possible.
4. **consistency across versions**: Use a script to produce the data sent to us, and inform us when changes occur. Most of our processes are automated, and changes that are trivial to humans (*e.g.*, `yyyy-mm-dd` to `mm/dd-yy`) cause problems for the automation.
5. **currency**: represent money as an integer or floating-point variable. This representation is more easily parsable by software, and enables mathematical operations (like `max()` or `mean()`) to be performed directly. Avoid commas and symbols like "$". If there is a possibility of ambiguity, indicate the denomination in the variable name (*e.g.*, `payment_dollars` or `payment_euros`).

6.3.2 Text

These conventions usually work best within plain-text formats.

1. **csv**: comma separated values are the most common plain-text format, so they have better support than similar formats where cells are separated by tabs or semi-colons. However, if you are receiving a well-behaved file separated by these characters, just go with the flow.

2. **cells enclosed in quotes**: a 'cell' should be enclosed in double quotes, especially if it's a string/character variable.

6.3.3 Excel Exceptions

If Excel is necessary for some reason,

1. **avoid multiple tabs/worksheets**: they are more complicated to read with automation, and the produces the opportunities for inconsistent variables across tabs/worksheets.

6.3.4 Meditech

1. **patient identifier**: `mrn_meditech` instead of `mrn`, `MRN Rec#`, or `Med Rec#`.

2. **account/admission identifier**: `account_number` instead of `mrn`, `Acct#`, or `Account#`.

3. **patient's full name**: `name_full` instead of `Patient Name` or `Name`.

4. **long/tall format**: one row per dx per patient (up to 50 dxs) instead of 50 *columns* of `dx` per patient. Applies to

    1. diagnosis code & description
    2. order date & number
    3. procedure name & number

Meditech Idiosyncracies:

1. **blood pressure**: in most systems the `bp_diastolic` and `bp_systolic` values are stored in separate integer variables. In Meditech, they are stored in a single character variable, separated by a foward slash.

6.3.5 Databases

When exchanging data between two different systems, …

Chapter 7

Patterns

7.1 Ellis

7.1.1 Purpose

To incorporate outside data source into your system safely.

7.1.2 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.)

7.1.3 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.

7.1.4 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

7.1.5 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 w
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)** See load-data chunk described in the prototypical file.

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

6. **Tweak Data**

   See tweak-data chunk described in the prototypical file.

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

7. **Body of the Ellis**

8. **Verify**

9. **Specify Columns**

   See specify-columns-to-upload chunk described in the prototypical file.

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

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

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

7.2 Arch

7.3 Ferry

7.4 Scribe

7.5 Analysis

7.6 Presentation -Static

7.7 Presentation -Interactive

7.8 Metadata

Survey items can change across time (for justified and unjustified reasons). We prefer to dedicate a metadata csv to a single variable

https://github.com/LiveOak/vasquez-mexican-census-1/issues/17#issuecomment-567254695

relationship_id	code_2011	code_2016	relationship	display_order	description_2011	description_2016
1	1	1	Jefe(a)	1	Jefe(a)	Jefe(a)
2	2	2	Esposo(a) o compañero(a)	2	Esposo(a) o compañero(a)	Esposo(a) o compañero(a)
3	3	3	Hijo(a)	3	Hijo(a)	Hijo(a)
4	4	4	Nieto(a)	4	Nieto(a)	Nieto(a)
5	5	5	Yerno/nuera	5	Yerno/nuera	Yerno/nuera
6	6	6	Hermano(a)	6	Hermano(a)	Hermano(a)
7	7	NA	Sobrino(a)	7	Sobrino(a)	NA
8	8	NA	Padre o madre	8	Padre o madre	NA

relationship_id	code_2011	code_2016	relationship	display_order	description_2011	description_2016
9	9	NA	Suegro(a)	9	Suegro(a)	NA
10	10	NA	Cuñado(a)	10	Cuñado(a)	Cuñado(a)
11	11	7	Otros parientes	11	Otros parientes	Otros parientes
12	12	8	No parientes	12	No parientes	No parientes
13	13	9	Empleado(a) doméstico(a)	13	Empleado(a) doméstico(a)	Empleado(a) doméstico(a)
99	99	NA	No especificado	99	No especificado	NA

7.8.1 Primary Rules for Mapping

A few important rules are necessary to map concepts in this multidimensional space.

1. each **variable** gets its own **csv**, such as `relationship.csv` (show above), `education.csv`, `living-status.csv`, or `race.csv`. It's easiest if this file name matches the variable.

2. each **variable** also needs a unique *integer* that identifies the underlying level in the database, such as `education_id`, `living_status_id`, and `relationship_id`.

3. each **survey wave** gets its own **column** within the csv, such as `code_2011` and `code_2016`.

4. each **level** within a variable-wave gets its own **row**, like `Jefe`, `Esposo`, and `Hijo`.

7.8.2 Secondary Rules for Mapping

In this scenarios, the first three columns are critical (*i.e.*, `relationship_id`, `code_2011`, `code_2016`). Yet these additional guidelines will help the plumbing and manipulation of lookup variables.

1. each **variable** also needs a unique *name* that identifies the underlying level for human, such as `education`, `living_status`, and `relationship`. This is the human label corresponding to `relationship_id`. It's easiest if this column name matches the variable.

2. each **survey wave** gets its own **column** within the csv, such as `description_2011` and `description_2016`. These are the human labels corresponding to variables like `code_2011` and `code_2016`.

3. each **variable** benefits from a unique *display order* value, that will be used later in analyses. Categorical variables typically have some desired sequence in graph legends and tables; specify that order here. This helps define the `factor` levels in R or the `pandas.Categorical` levels in Python.

4. Mappings are usually informed by outside documentation. For transparency and maintainability, clearly describe where the documentation can be found. One option is to include it in `data-public/metadata/README.md`. Another option is to include it at the bottonm of the csv, preceded by a `#`, or some 'comment' character that can keep the csv-parser from treating the notes like data it needs to squeeze into cells. Notes for this example are:

   ```
   # Notes,,,,,,
   # 2016 codes come from `documentation/2106/fd_endireh2016_dbf.pdf`, pages 14-15,,,
   # 2011 codes come from `documentation/2011/fd_endireh11.xls`, ‘TSDem’ tab,,,,,
   ```

5. sometimes a `notes` column helps humans keep things straight, especially researchers new to the field/project. In the example above, the `notes` value in the first row might be "*jefe* means 'head', not 'boss' ".

Chapter 8

Security & Private Data

8.1 File-level permissions

8.2 Database permissions

8.3 Public & Private Repositories

8.3.1 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.

8.3.1.0.1 Resources

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

Chapter 9

Automation

9.1 Flow File in R

9.2 Makefile

9.3 SSIS

9.4 cron Jobs & Task Scheduler

9.5 Sink Log Files

Chapter 10

Scaling Up

10.1 Data Storage

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

10.2 Data Processing

1. R vs SQL
2. R vs Spark

Chapter 11

Parallel Collaboration

11.1 Social Contract

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

11.2 Code Reviews

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

11.3 Remote

1. Headset & sharing screens

11.4 Additional Resources

- (Colin Gillespie, 2017), particularly the "Efficient collaboration" chapter.
- (Brian Fitzpatrick, 2012)

11.5 Loose Notes

11.5.1 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.

11.5.2 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 12

Documentation

12.1 Team-wide

12.2 Project-specific

12.3 Dataset Origin & Structure

12.4 Issues & Tasks

12.5 Flow Diagrams

12.6 Setting up new machine

(example)

Chapter 13

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. Many of these were inspired by (Francesco Balena, 2005).

13.1 Readability

13.1.1 Number

The word "number" is ambiguous, especially in data science. Try for these more specific terms

- **count**: the number of discrete objects or events, such as `visit_count`, `pt_count`, `dx_count`.
- **id**: a value that uniquely identifies an entity that doesn't change over time, such as `pt_id`, `clinic_id`, `client_id`,
- **index**: a 1-based sequence that's typically temporary, but unique within the dataset. For instance, `pt_index` 195 in Tuesday's dataset is likey a different person than `pt_index` 195 on Wednesday. On any given day, there is only one value of 195.
- **tag**: it is persistent across time like "id", but typically created by the analysts and send to the research team. See the snippet in the appendix for an example.

- **tally**: a running count
- **duration**: a length of time. Specify the units if it not self-evident.
- physical and statistical quantities like "depth", "length", "mass", "mean", and "sum".

13.1.2 Abbreviations

Try to avoid abbreviations. Different people tend to shorten words differently; this variability increases the chance that people reference the wrong variable. At very least, it wastes time trying to remember if `subject_number`, `subject_num`, or `subject_no` was used. The Consistency section describes how this can reduce errors and increase efficiency.

However, some terms are too long to reasonably use without shortening. We make some exceptions, such as the following scenarios:

1. humans commonly use the term orally. For instance, people tend to say "OR" instead of "operating room".

2. your team has agreed on set list of abbreviations. The list for our CDW team includes: appt (not "apt"), cdw, cpt, drg (stands for diagnosis-related group), dx, hx, icd pt, and vr (vital records).

When your team choose terms (*e.g.*, 'apt' vs 'appt'), try to use a standard vocabulary, such as MedTerms Medical Dictionary.

13.2 Datasets

13.2.1 Filtering Rows

Removing datasets rows is an important operation that is a frequent source of sneaky errors. These practices have hopefully reduced our mistakes and improved maintainability.

13.2.1.1 Dropping rows with missing values

`tidyr::drop_na()` drops rows with a missing value in a specific column.

```r
# Good
ds %>%
  tidyr::drop_na(dob)
```

is cleaner to read and write than these two styles. In particular, it's easy to forget/overlook a `!`.

```r
# Worse
ds %>%
  dplyr::filter(!is.na(dob))

# Worst
ds[!is.na(ds$dob), ]
```

13.2.1.2 Mimic number line

When ordering quantities, go smallest-to-largest as you type left-to-right.

13.2.1.3 Searchable verbs

You've probably asked in frustration, "Where did all the rows go? I had 1,000 in the middle of the file, but now have only 782." Try to keep a consistent tools for filtering, so you can 'ctrl+f' only a handful of terms, such as **`filter`**, **`drop_na`**, and **`summarize/summarise`**.

It's more difficult to highlight the When using the base R's filtering style, (*e.g.*, `ds <- ds[4 <= ds$count, ]`).

13.2.2 Don't attach

As the Google Stylesheet says, "The possibilities for creating errors when using `attach()` are numerous."

13.3 Categorical Variables

There are lots of names for a categorical variable across the different disciplines (*e.g.*, factor, categorical, …).

13.3.1 Explicit Missing Values

Define a level like `"unknown"` so the data manipulation doesn't have to test for both `is.na(x)` and `x=="unknown"`. The explicit labels also helps when included in a statistical procedure and coefficient table.

13.3.2 Granularity

Sometimes it helps to represent the values differently, say a granular and a coarse way. We say `cut7` or `cut3` to denotes the number of levels; this is related to `base::cut()`. 'unknown' and 'other' are frequently levels, and they count toward the quantity.

```r
# Inside a dplyr::mutate() clause
education_cut7      = dplyr::recode(
  education_cut7,
  "No Highschool Degree / GED"  = "no diploma",
  "High School Degree / GED"    = "diploma",
  "Some College"                = "some college",
  "Associate's Degree"          = "associate",
  "Bachelor's Degree"           = "bachelor",
  "Post-graduate degree"        = "post-grad",
  "Unknown"                     = "unknown",
  .missing                      = "unknown",
),
education_cut3      = dplyr::recode(
  education_cut7,
  "no diploma"    = "no bachelor",
  "diploma"       = "no bachelor",
  "some college"  = "no bachelor",
  "associate"     = "no bachelor",
  "bachelor"      = "bachelor",
  "post-grad"     = "bachelor",
  "unknown"       = "unknown",
),
education_cut7 = factor(education_cut7, levels=c(
  "no diploma",
  "diploma",
  "some college",
  "associate",
  "bachelor",
  "post-grad",
  "unknown"
)),
education_cut3 = factor(education_cut3, levels=c(
  "no bachelor",
  "bachelor",
  "unknown"
))
```

13.4 Dates

- yob is an integer, but mob and wob are dates. Typically months are collapsed to the 15th day and weeks are collapsed to Monday, which are the defaults of `OuhscMunge::clump_month_date()` and `OuhscMunge::clump_week_date()`. These help obfuscate the real value, if PHI is involved. Months are centered because the midpoint is usually a better representation of the month's performance than the month's initial day.

- `birth_month_index` can be values 1 through 12, while `birth_month` (or commonly `mob`) contains the year (*e.g.*, 2014-07-15).

13.5 Naming

13.5.1 Datasets

`data.frame`s are used in almost every analysis file, so we put extra effort formulating conventions that are informative and consistent. In the R world, "dataset" is typically a synonym of `data.frame` –a rectangular structure of rows and columns. The database equivalent of a conventional table. Note that "dataset" means a collections of tables in the the .NET world, and a collection of (not-necessarily-rectangular) files in Dataverse.[1]

13.5.1.1 Prefix with `ds_` and `d_`

Datasets are handled so differently than other variables that we find it's easier to identify its type and scope. The prefix `ds_` indicates the dataset is available to the entire file, while `d_` indicates the scope is localized to a function.

```
count_elements <- function (d) {
  nrow(d) * ncol(d)
}

ds <- mtcars
count_elements(d = ds)
```

[1]To complete the survey of "dataset" definitions: TThe Java world is like R, in that "dataset" typically describes rectangular tables (*e.g.*, Java, Spark, [scala]). In Julia and Python, qqq

13.5.1.2 Express the grain

The grain of a dataset describes what each row represents, which is a similar idea to the statistician's concept of "unit of analysis". Essentially it the the most granular entity described. Many miscommunications and silly mistakes are avoided when your team is disciplined enough to define a tidy dataset with a clear grain.

```
ds_student          # One row per student
ds_teacher          # One row per teacher
ds_course           # One row per course
ds_course_student   # One row per student-course combination
```

```
ds_pt         # One row per patient
ds_pt_visit   # One row per patient-visit combination
ds_visit      # Same as above, since it's clear a visit is connected w/ a pt
```

For more insight into grains, Ralph Kimball writes

> In debugging literally thousands of dimensional designs from my students over the years, I have found that the most frequent design error by far is not declaring the grain of the fact table at the beginning of the design process. If the grain isn't clearly defined, the whole design rests on quicksand. Discussions about candidate dimensions go around in circles, and rogue facts that introduce application errors sneak into the design. … I hope you've noticed some powerful effects from declaring the grain. First, you can visualize the dimensionality of the doctor bill line item very precisely, and you can therefore confidently examine your data sources, deciding whether or not a dimension can be attached to this data. For example, you probably would exclude "treatment outcome" from this example because most medical billing data doesn't tie to any notion of outcome.

13.5.1.3 Singular table names

If you adopt the style that the table's name reflects the grain, this is a corollary. If the grain is singular like "one row per client" or "one row per building", the name should be `ds_client` and `ds_building` (not `ds_clients` and `ds_buildings`). If these datasets are saved to a database, the tables are called `client` and `building`.

Table names are plural when the grain is plural. If a record has field like `client_id`, `date_birth`, `date_graduation` and `date_death`, I suggest called the table `client_milestones` (because a single row contains three milestones).

This Stack Overflow post presents a variety of opinions and justifications when adopting a singular or plural naming scheme.

I think it's acceptable if the R vectors follow a different style than R `data.frame`s. For instance, a vector can have a plural name even though each element is singular (*e.g.*, `client_ids <- c(10, 24, 25)`).

13.5.1.4 Use `ds` when definition is clear

Many times an ellis file handles with only one incoming csv and outgoing dataset, and the grain is obvious –typically because the ellis filename clearly states the grain.

13.5.1.5 Use an adjective after the grain, if necessary

If the same R file is manipulating two datasets with the same grain, qualify their differences after the grain, such as `ds_client_all` and `ds_client_michigan`. Adjectives commonly indicate that one dataset is a subset of another.

An occasional limitation with our naming scheme is that the difficult to distinguish the grain from the adjective. For instance, is the grain of `ds_student_enroll` either (a) every instance of a student enrollment (*i.e.*, `student` and `enroll` both describe the grain) or (b) the subset of students who enrolled (*i.e.*, `student` is the grain and `enroll` is the adjective)? It's not clear without examine the code, comments, or documentation.

If someone has a proposed solution, we would love to hear it. So far, we've been reluctant to decorate the variable name more, such as `ds_grain_client_adj_enroll`.

13.5.1.6 Define the dataset when in doubt

If it's potentially unclear to a new reader, use a comment immediately before the dataset's initial use.

```r
# `ds_client_enroll`:
#    grain: one row per client
#    subset: only clients who have successfully enrolled are included
#    source: the `client` database table, where `enroll_count` is 1+.
ds_client_enroll <- ...
```

13.5.2 Semantic sorting

Put the "biggest" term on the left side of the variable.

13.6 Whitespace

Although execution is rarely affected by whitespace in R and SQL files, be consistent and minimalistic. One benefit is that Git diffs won't show unnecessary churn. When a line of code lights up in a diff, it's nice when reflect a real change, and not something trivial like tabs were converted to spaces, or trailing spaces were added or deleted.

Some of these guidelines are handled automatically by modern IDEs, if you configure the correct settings.

1. Tabs should be replaced by spaces. Most modern IDEs have an option to do this for you automatically. (RStudio calls this "Insert spaces for tabs".)
2. Indentions should be replaced by a consistent number of spaces, depending on the file type.
    1. R: 2 spaces
    2. SQL: 2 spaces
    3. Python: 4 spaces
3. Each file should end with a blank line. (RStudio calls this "Ensure that source files end with newline.")
4. Remove spaces and tabs at the end of lines. (RStudio calls this "Strip trailing horizontal whitespace when saving".)

13.7 Database

GitLab's data team has a good style guide for databases and sql that's fairly consistent with our style. Some important similarities and differences are

1. Favor CTEs

2. The name of the primary key should typically contain the table. In the `employee` table, the key should be `empoyee_id`, not `id`.

13.8 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 (Lise DeShea (2015)); the 200+ graphs and their code is publicly available.

There are a few additional ggplot2 tips in the tidyverse style guide.

13.8.1 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 functions should follow a consistent order ideally starting with the more important structural functions and ending with the cosmetic functions. Our preference is:

1. `ggplot()` 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_*()` and `annotate()` creates the *geom*etric elements that represent 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. `guides()`
7. `theme()` (call the 'big' themes like `theme_minimal()` before overriding the details like `theme(panel.grid = element_line(color = "gray"))`)
8. `labs()`

13.8.2 Gotchas

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

13.8.2.1 Zooming

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.

13.8.2.2 Seed

When jittering, set the seed in the 'declare-globals' chunk so that rerunning the report won't create a (slightly) different png. The insignificantly different pngs will consume extra space in the Git repository. Also, the GitHub diff will show the difference between png versions, which requires extra cognitive load to determine if the difference is due solely to jittering, or if something really changed in the analysis.

Chapter 14

Publishing Results

14.1 To Other Analysts

14.2 To Researchers & Content Experts

14.3 To Technical-Phobic Audiences

Chapter 15

Testing, Validation, & Defensive Programming

15.1 Testing Functions

15.2 Defensive Programming

1. Throwing errors

15.3 Validator

1. Benefits for Analysts
2. Benefits for Data Collectors

Chapter 16

Troubleshooting and Debugging

16.1 Finding Help

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

16.2 Debugging

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

Chapter 17

Workstation

We believe it is important to keep software updated and consistent across workstations in your project. This material was originally posted at https://github.com/OuhscBbmc/RedcapExamplesAndPatterns/blob/master/DocumentationGlobal/ResourcesInstallation.md. It should help establish our tools on a new development computer.

17.1 Required Installation

The order matters.

17.1.1 R

R is the centerpiece of the analysis. Every few months, you'll need to download the most recent version. {added Sept 2012}

17.1.2 RStudio

RStudio Desktop is the IDE (integrated design interface) that you'll use to interact with R, GitHub, Markdown, and LaTeX. Updates can be checked easily through the menus `Help` -> `Check for updates`.

17.1.3 Installing R Packages

Dozens of R Packages will need to be installed. Choose between one of the two related scripts. It will install from our list of packages that our data analysts typically need. The script installs a package only if it's not already installed; also an existing package is updated if a newer version is available. Create a new 'personal library' if it prompts you. It takes at least fifteen minutes, so start it before you go to lunch. The list of packages will evolve over time, so please help keep the list updated.

To install our frequently-used packages, run the following snippet. The first lines installs an important package. The second line calls the online Gist, which defines the `package_janitor_remote()` function. The final line calls the function (and passes a specific CSV of packages)[1].

```
if( !base::requireNamespace("devtools") ) utils::install.packages("devtools")
devtools::source_gist("2c5e7459b88ec28b9e8fa0c695b15ee3", filename="package-janitor-bbm

package_janitor_remote(
  "https://raw.githubusercontent.com/OuhscBbmc/RedcapExamplesAndPatterns/master/utility
)
```

Some of our projects require specialized packages that are not typically used. In these cases, we will develop the git repo as an R package that includes a proper DESCRIPTION file. See RAnalysisSkeleton for an example.

When the project is opened in RStudio, `update_packages_addin()` in OuhscMunge will find the DESCRIPTION file and install the package dependencies.

```
if( !base::requireNamespace("remotes"   ) ) utils::install.packages("remotes")
if( !base::requireNamespace("OuhscMunge") ) remotes::install_github("OuhscMunge")
OuhscMunge::update_packages_addin()
```

17.1.4 Updating R Packages

Several R packages will need to be updated every weeks. Unless you have been told not to (because it would break something -this is rare), periodically update the packages by executing the following code `update.packages(checkBuilt=TRUE)`.

17.1.5 GitHub

GitHub registration is necessary to push modified files to the repository. First, register a free user account, then tell the repository owner your exact username, and they will add you as a collaborator (*e.g.*, to https://github.com/OuhscBbmc/RedcapExamplesAndPatterns).

[1]As an alternative to the Gist, run the local R script `install-packages.R` (located in the `utility/` directory) that lives in this repository. The workhorse of this function is `OuhscMunge::package_janitor()`.

17.1.6 GitHub for Windows Client

GitHub for Windows Client does the basic tasks a little easier than the git features built into RStudio. Occasionally, someone might need to use git form the command line to fix problems.

17.2 Recommended Installation

The order does not matter.

- **ODBC Driver for SQL Server** is for connecting to the token server, if your institution is using one. As of this writing, version 17 is the most recent driver version. See if a new one exists. {updated Apr 2018}
- **R Tools for Windows** is necessary to build some packages in development hosted on GitHub. {added Feb 2017}
- **Notepad++** is a text editor that allows you look at the raw text files, such as code and CSVs. For CSVs and other data files, it is helpful when troubleshooting (instead of looking at the file through Excel, which masks & causes some issues). {added Sept 2012}
- **Atom** is a text editor, similar to Notepad++. Notepad++ appears more efficient opening large CSVs. Atom is better suited when editing a lot of files in a repository. For finding and replacing across a lot of files, it is superior to Notepad++ and RStudio; it permits regexes and has a great GUI preview of the potential replacements.

  Productivity is enhanced with the following Atom packages:

  1. Sublime Style Column Selection: Enable Sublime style 'Column Selection'. Just hold 'alt' while you select, or select using your middle mouse button.
  2. atom-language-r allows Atom to recognize files as R. This prevents spell checking indicators and enable syntax highlighting. When you need to browse through a lot of scattered R files quickly, Atom's tree panel (on the left) works well. An older alternative is language-r.
  3. language-csv: Adds syntax highlighting to CSV files. The highlighting is nice, and it automatically disables spell checking lines.
  4. atom-beautify: Beautify HTML, CSS, JavaScript, PHP, Python, Ruby, Java, C, C++, C#, Objective-C, CoffeeScript, TypeScript, Coldfusion, SQL, and more in Atom.
  5. atom-wrap-in-tag: wraps tag around selection; just select a word or phrase and hit Alt + Shift + w.

6. minimap: A preview of the full source code (in the right margin).
7. script: Run scripts based on file name, a selection of code, or by line number.
8. git-plus: Do git things without the terminal (I don't think this is necessary anymore).

The packages can be installed through Atom, or through the `apm` utility in the command line:

```
apm install sublime-style-column-selection atom-language-r language-csv atom-beaut
```

And the following settings keep files consistent among developers.

1. File | Settings | Editor | Tab Length: 2 (As opposed to 3 or 4, used in other conventions)
2. File | Settings | Editor | Tab Type: soft (This inserts 2 spaces instead of a tab when 'Tab' is pressed)

- **Azure Data Studio (ADS)** is now recommended by Microsoft and others for analysts (and some other roles) –ahead of SQL Server Managment Studio.

  Note: here are some non-default changes that facilitate our workflow.

  1. Settings | Text Editor | **Tab Size: 2** {`"editor.tabSize": 2`}
  2. Settings | Text Editor | **Insert Final Newlines: check** {`"files.insertFinalNewline": true`}
  3. Settings | Text Editor | **Trim Final Newlines: check** {`"files.trimFinalNewlines": true`}
  4. Settings | Text Editor | **Trim Trailing Whitespace: check** {`"files.trimTrailingWhitespace": true`}
  5. Data | Sql | **Copy Includes Headers: check** {`"sql.copyIncludeHeaders": true`}

- **Pulse Secure** is VPN client for OUHSC researchers. It's not required for the REDCap API, but it's usually necessary to communicate with other campus data sources.

17.3 Optional Installation

The order does not matter.

- **Git** command-line utility enables some advanced operations that the GitHub client doesn't support. Use the default installation options, except these preferences of ours:

1. Nano is the default text editor.

- **GitLab SSL Certificate** isn't software, but still needs to be configured.

  1. Talk to Will for the server URL and the `*.cer` file.
  2. Save the file in something like `~/keys/ca-bundle-gitlab.cer`
  3. Associate the file with `git config --global http.sslCAInfo ...path.../ca-bundle-gitlab.cer` (but replace `...path...`).

- **MiKTeX** is necessary only if you're using knitr or Sweave to produce *LaTeX* files (and not just *markdown* files). It's a huge, slow installation that can take an hour or two. {added Sept 2012}

- **LibreOffice Calc** is an alternative to Excel. Unlike it Excel, it doesn't guess much with formatting (which usually mess up things, especially dates).

- **Visual Studio Code** is an extensible text editor that runs on Windows and Linux, similar to Atom (described above). It's much lighter than the full Visual Studio. Like Atom, it supports browsing through the directory structure, replacing across files, interaction with git, and previewing markdown. Currently, it supports searching CSVs better than Atom. Productivity is enhanced with the following extensions: {added Dec 2018}

  – Excel Viewer isn't a good name, but I've liked the capability. It displays CSVs and other files in a grid. {added Dec 2018}

  – Rainbow CSV color codes the columns, but still allows you to see and edit the raw plain-text file. {added Dec 2018}

  – SQL Server allows you to execute against a database, and view/copy/save the grid results. It doesn't replicate all SSMS features, but is nice as your scanning through files. {added Dec 2018}

  – Code Spell Checker produces green squiggly lines under words not in its dictionary. You can add words to your user dictionary, or a project dictionary.

  These extensions can be installed by command line.

```
code --list-extensions
code --install-extension GrapeCity.gc-excelviewer
code --install-extension mechatroner.rainbow-csv
code --install-extension ms-mssql.mssql
code --install-extension streetsidesoftware.code-spell-checker
```

- **pandoc** converts files from one markup format into another. {added Sept 2012}

17.4 Ubuntu Installation

Ubuntu desktop 19.04 follows these instructions for the R and RStudio and required these debian packages to be installed before the R packages. The `--yes` option avoids manual confirmation for each line, so you can copy & paste this into the terminal.

Add the following to the sources with `sudo nano /etc/apt/sources.list`. The 'eoan' version may be updated; The 'metrocast' part could be modified too from this list. I found it worked better for a new Ubuntu release than 'cloud.r-project.org'.

```
deb https://cloud.r-project/bin/linux/ubuntu/ eoan-cran35/
deb-src https://cloud.r-project/bin/linux/ubuntu/ eoan-cran35/
deb http://mirror.metrocast.net/ubuntu/ eoan-backports main restricted universe
```

This next block can be copied and pasted (ctrl-shift-v) into the console entirely. Or lines can be pasted individual (without the `( function install-packages {` line, or the last three lines).

```
( function install-packages {
  ### Add the key, update the list, then install base R.
  sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E298A3A825C0D65DFD57CB
  sudo apt-get update
  sudo apt-get install r-base r-base-dev

  ### Git
  sudo apt-get install git-core
  git config --global user.email "wibeasley@hotmail.com"
  git config --global user.name "Will Beasley"
  git config --global credential.helper 'cache --timeout=3600000'

  ### Ubuntu & Bioconductor packages that are indirectly needed for packages and BBMC

  # Supports the `locate` command in bash
  sudo apt-get install mlocate

  # The genefilter package is needed for 'modeest' on CRAN.
  # No longer a modeest dependency: Rscript -e 'BiocManager::install("genefilter")'

  ### CRAN packages that are also on the Ubuntu repositories

  # The 'xml2' package; https://CRAN.R-project.org/package=xml2
  sudo apt-get --yes install libxml2-dev r-cran-xml
```

```
# The 'curl' package, and others; https://CRAN.R-project.org/package=curl
sudo apt-get --yes install libssl-dev libcurl4-openssl-dev

# The 'udunits2' package: https://cran.r-project.org/web/packages/udunits2/index.html
sudo apt-get --yes install libudunits2-dev

# The 'odbc' package: https://github.com/r-dbi/odbc#linux---debian--ubuntu
sudo apt-get --yes install unixodbc-dev tdsodbc odbc-postgresql libsqliteodbc

# The 'rgl' package; https://stackoverflow.com/a/39952771/1082435
sudo apt-get --yes install libcgal-dev libglu1-mesa-dev

# The 'magick' package; https://docs.ropensci.org/magick/articles/intro.html#build-from-source
sudo apt-get --yes install 'libmagick++-dev'

# To compress vignettes when building a package; https://kalimu.github.io/post/checklist-for-r-
sudo apt-get --yes install qpdf

# The 'pdftools' and 'Rpoppler' packages, which involve PDFs
sudo apt-get --yes install libpoppler-cpp-dev libpoppler-glib-dev

# The 'sys' package
sudo apt-get --yes install libapparmor-dev

# The 'sf' and other spatial packages: https://github.com/r-spatial/sf#ubuntu; https://github.c
sudo apt-get --yes install libudunits2-dev libgdal-dev libgeos-dev libproj-dev libgeos++-dev

# For Cairo package, a dependency of Shiny & plotly; https://gykovacsblog.wordpress.com/2017/05
sudo apt-get --yes install libcairo2-dev

# 'rJava' and others; https://www.r-bloggers.com/installing-rjava-on-ubuntu/
sudo apt-get --yes install default-jre default-jdk
sudo R CMD javareconf
sudo apt-get --yes install r-cran-rjava

# For reprex and sometimes ssh keys; https://github.com/tidyverse/reprex#installation
sudo apt-get --yes install xclip

# gifski -apparently the rust compiler is necessary
sudo apt-get --yes install cargo

# For databases
sudo apt-get --yes install sqlite sqliteman
sudo apt-get --yes install postgresql postgresql-contrib pgadmin3
```

```
  # pandoc
  sudo apt-get --yes install pandoc

  # For checking packages. Avoid `/usr/bin/texi2dvi: not found` warning.
  sudo apt-get install texinfo
}
install-packages
)
```

The version of pandoc from the Ubuntu repository may be delayed. To install the latest version, download the .deb file then install from the same directory. Finally, verify the version.

```
sudo dpkg -i pandoc-*
pandoc -v
```

The Postman native app for Ubuntu is installed through snap, which is updated daily automatically.

```
snap install postman
```

17.5 Asset Locations

- **GitHub repository** https://github.com/OuhscBbmc/RedcapExamplesAndPatterns {added Sept 2012}
- **File server directory** Ask your PI. For Peds, it's typically on the "S" drive.
- **SQL Server Database** Ask Thomas, Will or David
- **REDCap database** Ask Thomas, Will or David. It is a http url, and we're trying not to publicize its value.
- **ODBC UserDsn** The name depends on your specific repository, and SQL Server database. Ask Thomas, Will or David for how to set it up.

17.6 Administrator Installation

- **MySQL Workbench** is useful occassionly for REDCap admins.
- **Postman Native App** is useful for developing with the API and has replaced the Chrome app. If that's not possible, a web client is available as well. With either program, do not access any PHI.

- **SQL Server Management Studio (SSMS)** has been replaced by Azure Data Studio for some roles, but is still recommended for database administrators. It is an easy way to access the database and write queries (and transfer the SQL to an R file). It's not required for the REDCap API, but it's usually necessary when integrating REDCap with other databases.

  Note: here are some non-default changes that facilitate our workflow. The first two help when we save the database *structure* (not data) on GitHub, so we can easily track/monitor the structural changes over time. The *tabs* options keeps things consistent between editors. In the SSMS 'Tools | Options' dialog box:

  1. SQL Server Object Explorer | Scripting | Include descriptive headers: False
  2. SQL Server Object Explorer | Scripting | Script extended properties: False
  3. Text Editor | All Languages | Tabs | Tab size: 2
  4. Text Editor | All Languages | Tabs | Indent size: 2
  5. Text Editor | All Languages | Tabs | Insert Spaces: true

  These dont affect the saved files, but make life easier. The first makes the result font bigger.

  1. Environment | Fonts and Colors | Show settings for: Grid Results | Size: 10
  2. Query Results | SQL Server | Results to Grid | Include column headers when copying or saving the results: false'
  3. Designers | Table and Database Designers | Prevent saving changes that require table-recreation: false
  4. Text Editor | Editor Tab and Status Bar | Tab Text | Include Server Name: false
  5. Text Editor | Editor Tab and Status Bar | Tab Text | Include Database Name: false
  6. Text Editor | Editor Tab and Status Bar | Tab Text | Include Login Name: false
  7. Text Editor | All Languages | General | Line Numbers: true

  For more details, see setting-up-dev-machine.md (in a private repo that's restricted to BBMC members).

17.7 Installation Troubleshooting

- **Git**: Will Beasley resorted to this workaround Sept 2012: http://stackoverflow.com/questions/3431361/git-for-windows-the-programcant-start-because-libiconv2-dll-is-missing. And then he copied the following four files from `D:/Program Files/msysgit/mingw/bin/` to `D:/Program Files/msysgit/bin/`: (1) `libiconv2.dll`, (2) `libcurl-4.dll`, (3) `libcrypto.dll`, and (4) `libssl.dll`. (If you install to the default location, you'll move instead from `C:/msysgit/mingw/bin/` to `C:/msysgit/bin/`) {added Sept 2012}

- **Git**: On a different computer, Will Beasley couldn't get RStudio to recognize msysGit, so installed the `Full installer for official Git for Windows 1.7.11` from (http://code.google.com/p/msysgit/downloads/list) and switched the Git Path in the RStudio Options. {added Sept 2012}

- **RStudio** If something goes wrong with RStudio, re-installing might not fix the issue, because your personal preferences aren't erased. To be safe, you can be thorough and delete the equivalent of `C:\Users\wibeasley\AppData\Local\RStudio-Desktop\`. The options settings are stored (and can be manipulated) in this extentionless text file: `C:\Users\wibeasley\AppData\Local\RStudio-Desktop\monitored\user-settings\user-sett` {added Sept 2012}

17.8 Retired Tools

We previously installed this software in this list. Most have been replaced by software above that's either newer or more natural to use.

- **msysGit** allows RStudio to track changes and commit & sync them to the GitHub server. Connect RStudio to GitHub repository. I moved this to optional (Oct 14, 2012) because the GitHub client (see above) does almost everything that the RStudio plugin does; and it does it a little better and a little more robust; and its installation hasn't given me problems. {added Oct 2012}

  - Starting in the top right of RStudio, click: Project -> New Project -> Create Project from Version Control -> Git {added Sept 2012}
  - An example of a repository URL is https://github.com/OuhscBbmc/RedcapExamplesAndPatterns. Specify a location to save (a copy of) the project on your local computer. {added Sept 2012}

- **CSVed** is a lightweight program for viewing data files. It fits somewhere between a text editor and Excel.

- **SourceTree** is a rich client that has many more features than the GitHub client. I don't recommend it for beginners, since it has more ways to mess up things. But for developers, it nicely fills a spot in between the GitHub client and command-line operations. The branching visualization is really nice too. Unfortunately and ironically, it doesn't currently support Linux. {added Sept 2014}.

- **git-cola** is probably the best GUI for Git supported on Linux. It's available through the official Ubuntu repositories with `apt-get` (also see this). The branch visualization features are in a different, but related program, 'git dag'. {added Sept 2014}

- **GitHub for Eclipse** is something I discourage for a beginner, and I strongly recommend you start with RStudio (and GitHub Client or the git capabilities within RStudio) for a few months before you even consider Eclipse. It's included in this list for the sake of completeness. When installing EGit plug-in, ignore eclipse site and check out this youtube video:http://www.youtube.com/watch?v=I7fbCE5nWPU.

- **Color Oracle** simulates the three most common types of color blindness. If you have produce a color graph in a report you develop, check it with Color Oracle (or ask someone else too). If it's already installed, it takes less than 10 second to check it against all three types of color blindness. If it's not installed, extra work may be necessary if Java isn't already installed. When you download the zip, extract the `ColorOracle.exe` program where you like. {added Sept 2012}

Chapter 18

Considerations when Selecting Tools

18.1 General

18.1.1 The Component's Goal

While discussing the advantages and disadvantages 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 maintenance 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 familiar 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.

18.1.2 Current Skillset of Team

18.1.3 Desired Future Skillset of Team

18.1.4 Skillset of Audience

18.2 Languages

18.3 R Packages

1. When developing a codebase used by many people, choose packages both on their functionality, as well as their ease of installation and maintainability. For example, the rJava package is a powerful package that allows R package developers to leverage the widespread Java framework and many popular Java packages. However, installing Java and setting the appropriate path or registry settings can be error-prone, especially for non-developers.

   Therefore when considering between two functions with comparable capabilities (*e.g.*, `xlsx::read.xlsx()` and `readxl::read_excel()`), avoid the package that requires a proper installation and configuration of Java and rJava.

   If the more intensive choice is required (say, you need to a capability in xslx missing from readxl), take:

   1. 20 minutes to **start a markdown file** that enumerates the package's direct and indirect dependencies that require manual configuration (*e.g.*, rJava and Java), where to download them, and the typical installation steps.
   2. 5 minutes to **create a GitHub Issue** that (a) announces the new requirement, (b) describes who/what needs to install the requirement, (c) points to the markdown documentation, and (d) encourages teammates to post their problems, recommendations, and solutions in this issue. We've found that a dedicated Issue helps communicate that the package dependency necessitates some intention and encourages people to assist other people's troubleshooting. When something potentially useful is posted in the Issue, move it to the markdown document. Make sure the document and the issue hyperlink to each other.
   3. 15 minutes every year to re-evaluate the landscape. Confirm that the package is still actively maintained, and that no newer (and easily-maintained) package offers the desired capability.[1] If better fit now

[1] In this case, the openxlsx package is worth consideration because it writes to an Excel file, but uses a C++ library, not a Java library.

> exists, evaluate if the effort to transition to the new package is worth the benefit. Be more willing to transition is the project is relatively green, and more development is upcoming. Be more willing to transition if the transition is relatively in-place, and will not require much modification of code or training of people.

Finally, consider how much traffic passes through the dependency A brittle dependency will not be too disruptive if isolated in a downstream analysis file run by only one statistician. On the other hand, be very protective in the middle of the pipeline where typically most of your team runs.

18.4 Database

1. Ease of installation & maintenance
2. Support from IT –which database engine are they most comfortable supporting.
3. Integration with LDAP, Active Directory, or Shibboleth.
4. Warehouse vs transactional performance

18.5 Additional Resources

- (Colin Gillespie, 2017), particularly the "Package selection" section.

Chapter 19

Growing a Team

19.1 Recruiting

19.2 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

19.3 Bridges Outside the Team

1. Monthly User Groups
2. Annual Conferences

Appendix A

Git & GitHub

A.1 for Code Development

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.

Other Resources 1. Setting up a CI/CD Process on GitHub with Travis CI. Travis-CI blob from August 2019.

A.2 for Collaboration

1. Somewhat separate from it's version control capabilities, GitHub provides built-in tools for coordinating projects across people and time. This tools revolves around GitHub Issues, which allow teammates to

2. track issues assigned to them and others

3. search if other teammates have encountered similar problems that their facing now (*e.g.*, the new computer can't install the rJava package).

There's nothing magical about GitHub issues, but if you don't use them, consider using a similar or more capable tools like those offered by Atlassian, Asana, Basecamp, and many others.

Here are some tips from our experiences with projects involving between 2 and 10 statisticians are working with an upcoming deadline.

1. If you create an error that describes a problem blocking your progress, include both the raw text (*e.g.*, `error: JAVA_HOME cannot be determined from the Registry`) and possibly a screenshot. The text allows the problem to be more easily searched by people later; the screenshot usually provides extra context that allows other to understand the situation and help more quickly.

2. Include enough broad context and enough specific details that teammates can quickly understand the problem. Ideally they can even run your code and debug it. Good recommendations can be found in the Stack Overflow posts, 'How to make a great R reproducible example' and 'How do I ask a good question?'. The issues don't need to be as thorough, because your teammates start with more context than a Stack Overflow reader.

   We typically include

   1. a description of the problem or fishy behavior.
   2. the exact error message (or a good description of the fishy behavior).
   3. a snippet of the 1-10 lines of code suspected of causing the problem.
   4. a link to the code's file (and ideally the line number, such as `https://github.com/OuhscBbmc/REDCapR/blob/master/R/redcap-version.R#L40`) so the reader can hop over to the entire file.
   5. references to similar GitHub Issues or Stack Overflow questions that could aid troubleshooting.

A.3 for Stability

1. Review Git commits closely
   1. No unintended functional difference (*e.g.*, `!match` accidentally changed to `match`).
   2. No PHI snuck in (*e.g.*, a patient ID used while isolating and debugging).
   3. The metadata format didn't change (*e.g.*, Excel sometimes changes the string '010' to the number '10').

Appendix B

Snippets

B.1 Reading External Data

B.1.1 Reading from Excel

Background: Avoid Excel for so many reasons. But if there isn't another good option, be protective. `readxl::read_excel()` allows you to specify column types, but not column order. The names of `col_types` is ignored by `readxl::read_excel()`. To defend against roamining columns (*e.g.*, the files changed over time), `tesit::assert()` that the order is what you expect.

Last Modified: 2019-12-12 by Will

```
# ---- declare-globals ---------------------------------------------------------
config                              <- config::get()

# cat(sprintf('  `%s`              = "text",\n', colnames(ds)), sep="") # 'text' by default --the
col_types <- c(
  `Med Rec Num`         = "text",
  `Admit Date`          = "date",
  `Tot Cash Pymt`       = "numeric"
)

# ---- load-data ---------------------------------------------------------------
ds <- readxl::read_excel(
  path       = config$path_admission_charge,
  col_types  = col_types
  # sheet    = "dont-use-sheets-if-possible"
)
```

```r
testit::assert(
  "The order of column names must match the expected list.",
  names(col_types) == colnames(ds)
)
```

B.1.2 Removing Trailing Comma from Header

Background: Ocassionally a Meditech Extract will have an extra comma at the end of the 1st line. For each subsequent line, `readr:read_csv()` appropriately throws a new warning that it is missing a column. This warning flood can mask real problems.

Explanation: This snippet (a) reads the csv as plain text, (b) removes the final comma, and (c) passes the plain text to `readr::read_csv()` to convert it into a data.frame.

Instruction: Modify `Dx50 Name` to the name of the final (real) column.

Real Example: truong-pharmacist-transition-1 (Accessible to only CDW members.)

Last Modified: 2019-12-12 by Will

```r
# The next two lines remove the trailing comma at the end of the 1st line.
raw_text  <- readr::read_file(path_in)
raw_text  <- sub("^(.+Dx50 Name),", "\\1", raw_text)

ds        <- readr::read_csv(raw_text, col_types=col_types)
```

B.2 Grooming

B.2.1 Correct for misinterpreted two-digit year

Background: Sometimes the Meditech dates are specified like `1/6/54` instead of `1/6/1954`. `readr::read_csv()` has to choose if the year is supposed to be '1954' or '2054'. A human can use context to guess a birth date is in the past (so it guesses 1954), but readr can't (so it guesses 2054). For avoid this and other problems, request dates in an ISO-8601 format.

Explanation: Correct for this in a `dplyr::mutate()` clause; compare the date value against today. If the date is today or before, use it; if the day is in the future, subtract 100 years.

Instruction: For future dates such as loan payments, the direction will flip.

Last Modified: 2019-12-12 by Will

```
ds %>%
dplyr::mutate(
    dob   = dplyr::if_else(dob <= Sys.Date(), dob, dob - lubridate::years(100))
  )
```

B.3 Identification

B.3.1 Generating "tags"

Background: When you need to generate unique identification values for future people/clients/patients, as described in the style guide.

Explanation: This snippet will create a 5-row csv with random 7-character "tags" to send to the research team collecting patients. The

Instruction: Set `pt_count`, `tag_length`, `path_out`, and execute. Add and rename the columns to be more appropriate for your domain (*e.g.*, change "patient tag" to "store tag").

Last Modified: 2019-12-30 by Will

```
pt_count    <- 5L   # The number of rows in the dataset.
tag_length  <- 7L   # The number of characters in each tag.
path_out    <- "data-private/derived/pt-pool.csv"

draw_tag <- function (tag_length = 4L, urn = c(0:9, letters)) {
  paste(sample(urn, size = tag_length, replace = T), collapse = "")
}

ds_pt_pool <-
  tibble::tibble(
    pt_index    = seq_len(pt_count),
    pt_tag      = vapply(rep(tag_length, pt_count), draw_tag, character(1)),
    assigned    = FALSE,
    name_last   = "--",
    name_first  = "--"
  )

readr::write_csv(ds_pt_pool, path_out)
```

The resulting dataset will look like this, but with different randomly-generated tags.

```
# A tibble: 5 x 5
```

```
  pt_index pt_tag  assigned name_last name_first
     <int> <chr>   <lgl>    <chr>     <chr>
1        1 seikyfr FALSE    --        --
2        2 voiix4l FALSE    --        --
3        3 wosn4w2 FALSE    --        --
4        4 jl0dg84 FALSE    --        --
5        5 r5ei5ph FALSE    --        --
```

Appendix C

Presentations

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

C.1 CDW

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

C.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.

C.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.

C.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.

C.5 GitHub

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

C.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.

C.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.

   .

C.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 D

Scratch Pad of Loose Ideas

D.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)
5. documentation - bookdown

Bookdown has worked well for us so far. It's basically independent markdown documents stored on a dedicated git repo. Then you click "build" in RStudio and it converts all the markdown files to static html files. Because GitHub is essentially serving as the backend, everyone can make changes to sections and we don't have to be too worried about

Here's a version that's hosted publicly, but I tested that it can be hosted on our shared file server. (It's possible because the html files are so static.) If this is what you guys want for OU's collective CDW, please tell me:

- who you want to be able to edit the documents without review. I'll add them to the GitHub repo.
- who you want to be able to view the documents. I'll add them to a dedicate file server space.

https://ouhscbbmc.github.io/data-science-practices-1/workstation.html#installation-required

I was thinking that each individual database gets it own chapter. The BBMC has ~4 databases in this sense: a Centricity staging database, a GECB staging database, the central warehouse, and the (fledgling) downstream OMOP database. Then there are ~3 sections within each chapter: (a) a black-and-white description of the tables, columns, & indexes (written mostly for consumers), (b) recommendations how to use each table (written mostly for consumers), and (c) a description of the ETL process (written mostly for developers & admins).

My proposal uses GitHub and Markdown because they're so universal (no knowledge of R is required –really you could write it with any text editor & commit, and let someone else click "build" in RStudio on their machine). But I'm very flexible on all this. I'll support & contribute to any system that you guys feel will work well across the teams.

1. developing packages

- *R packages* by Hadley Wickham

- http://mangothecat.github.io/goodpractice/

D.2 Practices

D.2.1 Date Arithmetic

Don't use `-` to subtract dates, use `difftime(stop, start, units="days")`. It's longer but protects from the scneario that `start` or `stop` are changed upstream to a date time. In that case, `stop - start` equals the number of *seconds* between the two points, not the number of *days*.

Appendix E

Example

This intro was copied from the 1st chapter of the example bookdown repo. I'm keeping it temporarily for reference.

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

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

```r
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 E.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table E.1.

```r
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, 2020) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

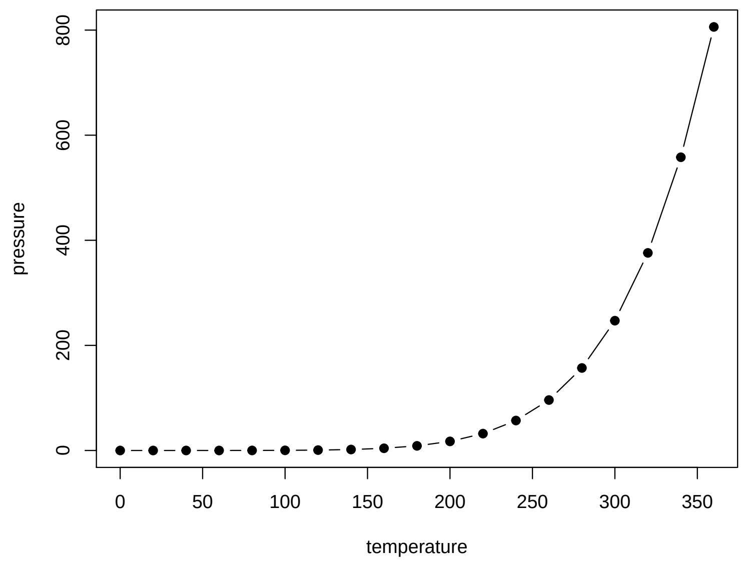

Figure E.1: Here is a nice figure!

Table E.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

Appendix F

Acknowledgements

The authors thank all our colleauges for the discussions and experiences about data science that lead to this book. At OUHSC, this includes @adrose, @aggie-dbc, @ARPeters, @Ashley-Jorgensen, @athumann, @atreat1, @caston60, @chanukyalakamsani, @CWilliamsOUHSC, @DavidBard, @evoss1, @geneva-marshall, @Maleeha, @man9472, @rmatkins, @sbohora, @vimleshbavadiya, @waleboro, @YuiYamaoka, @yutiantang.

Outside the OUHSC, this includes

@andkov, @ben519, @cscherrer, @cmodzelewski, @jimquallen, @mhunter1, @probinso, @russelljonas, and @spopovych.

Bibliography

Brian Fitzpatrick, B. C.-S. (2012). *Team Geek: A Software Developer's Guide to Working Well with Others.* O'Reilly, Cambridge MA, 1st edition. ISBN 978-1449302443.

Colin Gillespie, R. L. (2017). *Efficient R Programming.* O'Reilly, Cambridge MA, 1st edition. ISBN 978-1491950784.

Francesco Balena, G. D. (2005). *Practical Guidelines and Best Practices for Microsoft Visual Basic and Visual C# Developers.* Microsoft Press, Redmond, Washington, 1st edition. ISBN 978-0735621725.

Lise DeShea, L. E. T. (2015). *Introductory Statistics for the Health Sciences.* Chapman & Hall/CRC, Boca Raton, FL, 1st edition. ISBN 978-1466565333.

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

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown.* R package version 0.17.