David J. Stanley

PSYC6060 Course Notes

To my students,

from whom I've learn so much about teaching.

Contents

List of Tables

List of Figures

Preface

R's emerging role in psychology will be described here..

Structure of the book

I'll put more about the structure of the book here in the future.

Software information and conventions

I used the **knitr** package (Xie, 2015) and the **bookdown** package (Xie, 2020) to compile my book. My R session information is shown below:

```
xfun::session_info()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Catalina 10.15.5, RStudio 1.3.959
##
## Locale: en_CA.UTF-8 / en_CA.UTF-8 / en_CA.UTF-8 / C / en_CA.UTF-8 / en_CA.UTF-8
##
## Package version:
##   abind_1.4.5             acepack_1.4.1
##   arm_1.11.1              askpass_1.1
##   assertthat_0.2.1        backports_1.1.8
##   base64enc_0.1-3         BH_1.72.0.3
##   bookdown_0.19.1         boot_1.3.25
##   broom_0.7.0.9000        callr_3.4.3
##   cellranger_1.1.0        checkmate_2.0.0
##   cli_2.0.2               clipr_0.7.0
##    cluster_2.1.0           coda_0.19.3
##    colorspace_1.4-1        compiler_4.0.2
##    crayon_1.3.4            curl_4.3
##    data.table_1.12.8       DBI_1.1.0
##    dbplyr_1.4.3            desc_1.2.0
##    digest_0.6.25           dplyr_1.0.0
##    ellipsis_0.3.1          evaluate_0.14
##    fansi_0.4.1             farver_2.0.3
##    forcats_0.5.0           foreign_0.8-80
##    Formula_1.2-3           fs_1.4.1
##    generics_0.0.2          ggplot2_3.3.2
##    glue_1.4.1              graphics_4.0.2
##    grDevices_4.0.2         grid_4.0.2
##    gridExtra_2.3           gsl_2.1.6
##    gtable_0.3.0            haven_2.3.1
##    highr_0.8               Hmisc_4.4-0
##    hms_0.5.3               htmlTable_2.0.0
##    htmltools_0.4.0         htmlwidgets_1.5.1
##    httr_1.4.1              isoband_0.2.2
##    janitor_2.0.1           jpeg_0.1-8.1
##    jsonlite_1.6.1          knitr_1.28
##    labeling_0.3            lattice_0.20-41
##    latticeExtra_0.6-29     lavaan_0.6.6
##    learnSampling_0.2       lifecycle_0.2.0
##    lme4_1.1.23             lubridate_1.7.8
##    magrittr_1.5            markdown_1.1
##    MASS_7.3.51.6           Matrix_1.2-18
##    matrixcalc_1.0.3        MBESS_4.7.0
##    methods_4.0.2           mgcv_1.8.31
##    mi_1.0                  mime_0.9
##    minqa_1.2.4             mnormt_2.0.0
##    modelr_0.1.8            munsell_0.5.0
##    mvtnorm_1.1.1           nlme_3.1-148
##    nloptr_1.2.2.1          nnet_7.3-14
##    numDeriv_2016.8.1.1     OpenMx_2.17.4
##    openssl_1.4.1           packrat_0.5.0
##    parallel_4.0.2          pbapply_1.4.2
##    pbivnorm_0.6.0          pillar_1.4.4
##    pkgbuild_1.0.8          pkgconfig_2.0.3
##    pkgload_1.1.0           png_0.1-7
##    praise_1.0.0            predictionInterval_1.0.0
##    prettyunits_1.1.1       processx_3.4.2
##    progress_1.2.2          ps_1.3.3
##    psych_1.9.12.31         purrr_0.3.4
##    R6_2.4.1                RColorBrewer_1.1-2
##   Rcpp_1.0.4.6            RcppEigen_0.3.3.7.0
##   RcppParallel_5.0.2      readr_1.3.1
##   readxl_1.3.1            rematch_1.0.1
##   repr_1.1.0              reprex_0.3.0
##   rlang_0.4.6             rmarkdown_2.2
##   rpart_4.1-15            rpf_1.0.4
##   rprojroot_1.3.2         rsconnect_0.8.16
##   rstudioapi_0.11         rvest_0.3.5
##   scales_1.1.1            selectr_0.4.2
##   sem_3.1.11              semTools_0.5.3
##   skimr_2.1.1             snakecase_0.11.0
##   splines_4.0.2           StanHeaders_2.21.0.5
##   statmod_1.4.34          stats_4.0.2
##   stats4_4.0.2            stringi_1.4.6
##   stringr_1.4.0           survival_3.1-12
##   sys_3.3                 testthat_2.3.2
##   tibble_3.0.1            tidyr_1.1.0
##   tidyselect_1.1.0        tidyverse_1.3.0
##   tinytex_0.23            tmvnsim_1.0-2
##   tools_4.0.2             utf8_1.1.4
##   utils_4.0.2             vctrs_0.3.1
##   viridis_0.5.1           viridisLite_0.3.0
##   whisker_0.4             withr_2.2.0
##   xfun_0.14               xml2_1.3.2
##   yaml_2.2.1
```

Package names are in bold text (e.g., **rmarkdown**), and inline code and filenames are formatted in a typewriter font (e.g., `knitr::knit('foo.Rmd')`). Function names are followed by parentheses (e.g., `bookdown::render_book()`).

About the Author

David J. Stanley is an Associate Professor of Industrial and Organizational Psychology at the University of Guelph in Canada. He obtained his PhD from Western University in London, Ontario. David has published articles in Advances in Methods and Practices in Psychological Science, Organizational Research Methods, Journal of Applied Psychology, Perspectives in Psychological Science, Journal of Business and Psychology, Journal of Vocational Behaviour, Journal of Personality and Social Psychology, Behavior Research Methods, Industrial and Organizational Psychology, and Emotion among other journals. David also created the apaTables R package.

1

Introduction

Welcome! In this guide, we will teach you about statistics using the statistical software R with the interface provided by R Studio. The purpose of this chapter to is provide you with a set of activities that get you up-and-running in R quickly so get a sense of how it works. In later chatpers we will revisit these same topics in more detail.

1.1 A focus on workflow

An important part of this guide is training you in a workflow that will avoid many problems than can occur when using R.

1.2 R works with plug-ins

R is a statistical language with many plug-ins called **packages** that you will use for analyses. You can think of R as being like your smartphone. To do things with your phone you need **an App** (R equivalent: a *package*) from the App Store (R equivalent: *CRAN*). Apps need to be **downloaded** (R equivalent: *install.packages*) before you can use them. To use the app you need **Open** it (R equivalent: *library command*). These similarities are illustrated in Table 1.1 below.

TABLE 1.1: R packages are similar to smart phone apps (Kim, 2018)

Smart Phone Terminology	R Terminology
App	package
App Store	CRAN
Download App from App Store	install.packages(”apaTables”, dependencies = TRUE)
Open App	library(”apaTables”)

1.3 Create an account at R Studio Cloud

R Studio Cloud[1] accounts are free and required for this guide. Please go to the website and set up a new account.

1.4 Join the class workspace

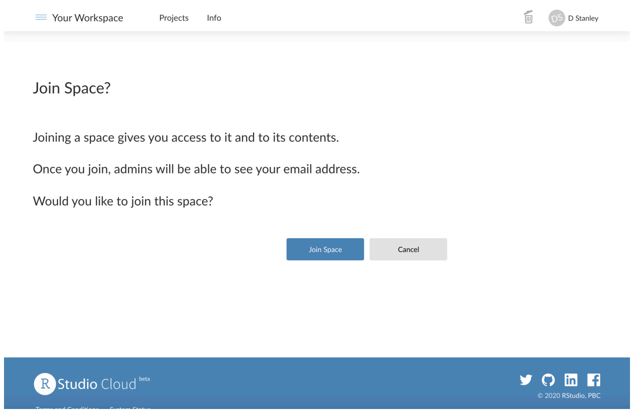
To do the assignment required for this class you need to join the class workspace on R Studio Cloud. To do so:

1. Log into R Studio Cloud (if you haven't already done so).
2. Go to your university email account and find the message with the subject "R Studio Workspace Invitation". In this message there is a link to the class R Studio Cloud workspace.
3. Click on the workspace link in the email or paste it into your web browser. You should see a screen like the one below in Figure 1.1. Click on the Join button.

4. Then you should see the welcome message illustrated in Figure 1.2. Above this message is the Projects menu option. Click on the word Project.

5. You should now see the First Project displayed as in 1.3. Click the Start button. You will then move to a view of R Studio.

[1] http://www.rstudio.cloud



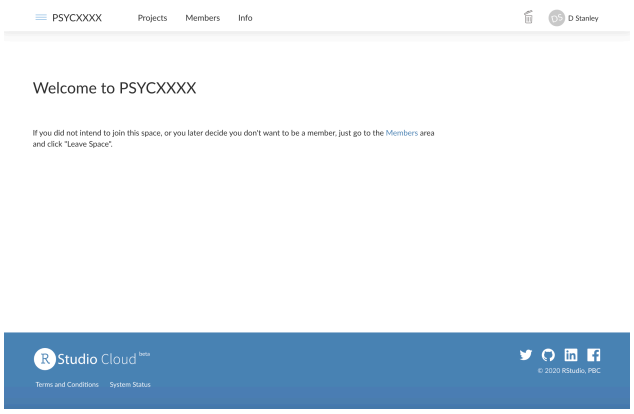
Join Space?

Joining a space gives you access to it and to its contents.

Once you join, admins will be able to see your email address.

Would you like to join this space?

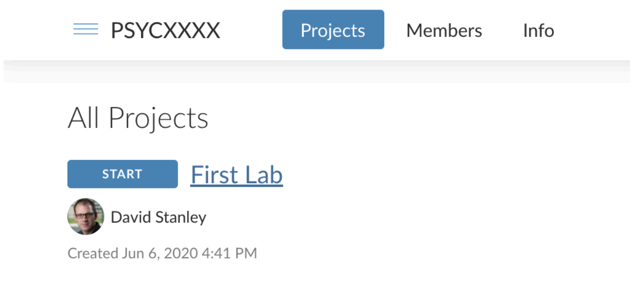
FIGURE 1.1: Screen shot of workspace join message.



Welcome to PSYCXXXX

If you did not intend to join this space, or you later decide you don't want to be a member, just go to the Members area and click "Leave Space".

FIGURE 1.2: Screen shot of welcome messag



All Projects

First Lab

David Stanley

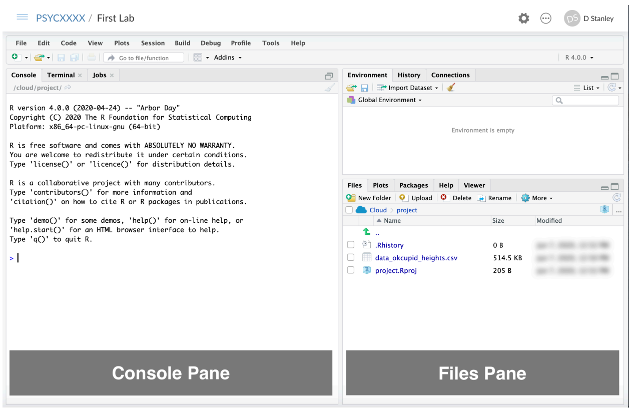
Created Jun 6, 2020 4:41 PM

FIGURE 1.3: Screen shot of starting first assignment

5. In R Studio it is essential you use projects to keep your files organized and in the same spot. For this course, when your start an assignment on R Studio Cloud and the project will already have been made for you. Later you will learn to make your own R Studio Projects.

1.5 Exploring the R Studio Interface

Once you have opened (or created) a Project folder, you are presented with the R Studio interface. There are a few key elements to the user interface that are illustrated in Figure 1.4 In the lower right of the screen you can see the a panel with several tabs (i.e., Files, Plots, Packages, etc) that I will refer to as the Files pane. You look in this pane to see all the files associated with your project. On the left side of the screen is the Console which is an interactive pane where you type and obtain results in real time. I've placed two large grey blocks on the screen with text to more clearly identify the Console and Files panes. Not shown in this figure is the Script panel where we can store our commands for later reuse.



```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Name	Size	Modified
..		
.Rhistory	0 B	
data_okcupid_heights.csv	514.5 KB	
project.Rproj	205 B	

FIGURE 1.4: R Studio interface

1.5.1 Console panel

When you first start R, the Console panel is on the left side of the screen. Sometimes there are two panels on the left side (one above the other); if so, the Console panel is the lower one (and labeled accordingly). We can use R a bit like a calculator. Try typing the following into the Console window: 8 + 6 + 7 + 5. You can see that R immediately produced the result on a line preceded by two hashtags (##).

```
8 + 6 + 7 + 5
```

```
## [1] 26
```

We can also put the result into a variable to store it. Later we can use the print command to see that result. In the example below we add the numbers 3, 0, and 9 and store the result in the variable my_sum. The text "<-" indicate you are putting what is on the right side of the arrow into the variable on the left side of the arrow. You can think of a variable as cup into which you can put different things. In this case, imagine a real-world cup with my_sum written on the outside and inside the cup we have stored the sum of 3, 0, and 9 (i.e., 12).

```
my_sum <- 3 + 0 + 9
```

We can inspect the contents of the my_sum variable (i.e., my_sum cup) with the print command:

```
print(my_sum)
```

```
## [1] 12
```

Variable are very useful in R. We will use them to store a single number, an entire data set, the results of an analysis, or anything else.

1.5.2 Script Panel

Although you can use R with just with the Console panel, it's a better idea to use scripts via the Script panel - not visible yet. Scripts are just text files with the commands you use stored in them. You can run a script (as you will see below) using the Run or Source buttons located in the top right of the Script panel.

Scripts are valuable because if you need to run an analysis a second time you don't have to type the command in a second time. You can run the script again and again without retyping your commands. More importantly though, the script provides a record of your analyses.

A common problem in science is that after an article is published, the authors can't reproduce the numbers in the paper. You can read more about the important problem in a surprising article in the journal Molecular Brain[2]. In this article an editor reports how a request for the data underlying articles resulted in the wrong data for 40 out of 41 papers. Long story short – keep track of the data and scripts you use for your paper. In a later chapter, it's generally poor practice to manipulate or modify or analyze your data using any menu driven software because this approach does not provide a record of what you have done.

1.6 Writing your first script

1.6.1 Create the script file

Create a script in your R Studio project by using the menu File > New File > R Script.

Save the file with an appropriate name using the File menu. The file will be saved in your Project folder. A common, and good, convention for naming is to start all script names with the word "script" and separate words with an underscore. You might save this first script file with the name "script_my_first_one.R". The advantage of beginning all script files with the word script is that when you look at your list of files alphabetically, all the script files will cluster together. Likewise, it's a good idea to save all data files such that they begin with "data_". This way all the data files will cluster together in your directory view as well. You can see there is already a data file with this convention called "data_okcupid.csv".

You can see as discussed previously, we are trying to instill an effective workflow as you learn R. Using a good naming convention (that is consistent with what others use) is part of the workflow. When you write your scripts it's a good idea to follow the tidyverse style guide[3] for script names, variable name, file names, and more.

1.6.2 Add a comment to your script

In the previous section you created your first script. We begin by adding a comment to the script. A comment is something that will be read by humans rather than the computer/R. You make comments for other people that will read your code and need to understand what you have done. However, realize

[2]https://molecularbrain.biomedcentral.com/articles/10.1186/s13041-020-0552-2
[3]https://style.tidyverse.org

that you are also making comments for your future self as illustrated in an XKCD cartoon[4].

A good way to start every script is with a comment that includes the date of your script (or even better when you installed your packages, more on this later). Like smartphone apps, packages are updated regularly. Sometimes after a package is updated it will no longer work with an older script. Fortunately, the checkpoint package[5] lets users role back the clock and use older versions of packages. Adding a comment with the date of your script will help future users (including you) to use your script with the same version of the package used when you wrote the script. Dating your script is an important part of an effective and reproducible workflow.

```
# Code written on: YYYY/MM/DD
# By: John Smith
```

Note that in the above comment I used the internationally accepted date format order Year/Month/Day. Some people use the mnemonic *Your My Dream* to remember this order. Wikipedia provides more information about the Internationally Date Format ISO 8601[6].

Moving forward, I suggest you use comments to make your own personal notes in your own code as your write it.

1.6.3 Background about the tidyverse

There are generally two broad ways of using R, the older way and the newer way. Using R the older way is refered to as using base R. A more modern approach to using R is the tidyverse. The tidyverse represents a collection of packages the work together to give R a modern workflow. These packages do many things to help the data analyst (loading data, rearranging data, graphing, etc.). We will use the tidyverse approach to R in this guide.

A noted the tidyverse is a collection of packges. Each package adds new commands to R. The number of packges and correspondingly the number of new commands added to R by the tidyverse is large. Below is a list of the tidyverse packages:

```
##  [1] "broom"      "cli"        "crayon"     "dbplyr"
##  [5] "dplyr"      "forcats"    "ggplot2"    "haven"
##  [9] "hms"        "httr"       "jsonlite"   "lubridate"
## [13] "magrittr"   "modelr"     "pillar"     "purrr"
```

[4]https://xkcd.com/1421/
[5]https://cran.r-project.org/web/packages/checkpoint/index.html
[6]https://en.wikipedia.org/wiki/ISO_8601

```
## [17] "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"
## [25] "tidyr"      "xml2"       "tidyverse"
```

Before you can use a package it needs to be installed – this is the same as downloading an app from the App Store. Normally, you can install a **single** packages with the install.packages command. Previously, you needed run an install.package command for every package in the tidyverse as illustrated below (though we no longer use this approach).

```
# The old way of installing the tidyverse packages
# Like downloading apps from the app store

install.packages("broom", dep = TRUE)
install.packages("cli", dep = TRUE)
install.packages("ggplot", dep = TRUE)
# etc
```

Fortunately, the tidyverse packages can now by installed with a single install.packages command. Specifically, the install.packages command below will install all of the packages listed above.

Class note: For the "First Lab", I've done the install.packages for you. So there is no need to use the install.packages command below in this first lab.

```
install.packages("tidyverse", dep = TRUE)
```

1.6.4 Add library(tidyverse) to your script

The tidyverse is now installed, so we need to activate it. We do that with the library command. Put the library line below at the top of your script file (below your comment):

```
# Code written on: YYYY/MM/DD
# By: John Smith
library(tidyverse)
```

1.6.5 Activate tidyverse auto-complete for your script

Select the library(tidyverse) text with your mouse/track-pad so that it is highlighted. Then click the Run button in the upper right of the Script panel.

Doing this "runs" the selected text. After you click the Run button you should see text like the following the Console panel:

When you use library(tidyverse) to activate the tidyverse you activate the most commonly used subset of the tidyverse packages. In the output you see checkmarks beside names of the tidyverse packages you have activated.

By activating these packages you have added new commands to R that you will use. Sometimes these packages replace older versions of commands in R. The "Conflicts" section in the output shows you where the packages you activated replaced older R commands with newer R commands. You can activate the other tidyverse package by running a library command for each package – if needed. No need to do so now.

Most importantly, running the library(tidyverse) prior to entering the rest of your script allows R Studio to present auto-complete options when typing your text. Remember to start each script with the library(tidyverse) command and then Run it so you get the autocomplete options for the rest of the commands your enter.

1.7 Loading your data

1.7.1 Use read_csv (not read.csv) to open files.

If you inspect the Files pane on the right of the screen you see the **data_okcupid.csv** data file in our project directory. We will load this data with the commands below. If you followed the steps above, you should have auto-complete for the tidyverse commands you type for now in – in the current R session. Enter the command below into your script. As your start to type read_csv you will likely be presented with an auto-complete option. You can use the arrow keys to move up and down the list of options to select the one you want - then press tab to select it.

Once your command looks like the one below select the text and click on the "Run" button.

Note: If you are not in the class, the data file is available from: https://github.com/dstanley4/psyc6060bookdown

```
okcupid_profiles <- read_csv(file = "data_okcupid.csv")
```

```
## Parsed with column specification:
## cols(
##   age = col_double(),
##   diet = col_character(),
##   height = col_double(),
##   pets = col_character(),
##   sex = col_character(),
##   status = col_character()
## )
```

The output indicates that you have loaded a data file and the type of data in each column. The sex column is of type col_character which indicates it contains text/letters. Most of the columns are of the type character. The age and height columns contain numbers are correspondingly indicated to be the type col_double. The label col_double indicates that a column of numbers represented in R with high precision[7]. There are other ways of representing numbers in R but this is the type we will see/use most often.

1.8 Checking out your data

There many ways of viewing the actual data you loaded. A few of these are illustrated now.

1.8.1 view(): See a spreadsheet view of your data

You can inspect your data in a spreadsheet view by using the view command. Do NOT add this command to your script file – EVER. Adding it to the script can cause substantial problems. Type this command in the Console.

```
view(okcupid_profiles)
```

1.8.2 print(): See you data in the Console

You can inspect the first few rows of your data with the print() command. It is OK to add a print command to your script. Try the print() command below in the Console:

[7] https://en.wikipedia.org/wiki/Double-precision_floating-point_format

```r
print(okcupid_profiles)
```

```
## # A tibble: 59,946 x 6
##      age diet         height pets              sex   status
##    <dbl> <chr>         <dbl> <chr>             <chr> <chr>
##  1    22 strictly an~     75 likes dogs and l~ m     single
##  2    35 mostly other     70 likes dogs and l~ m     single
##  3    38 anything         68 has cats          m     availa~
##  4    23 vegetarian       71 likes cats        m     single
##  5    29 <NA>             66 likes dogs and l~ m     single
##  6    29 mostly anyt~     67 likes cats        m     single
##  7    32 strictly an~     65 likes dogs and l~ f     single
##  8    31 mostly anyt~     65 likes dogs and l~ f     single
##  9    24 strictly an~     67 likes dogs and l~ f     single
## 10    37 mostly anyt~     65 likes dogs and l~ m     single
## # ... with 59,936 more rows
```

1.8.3 head(): Check out the first few rows of data

You can inspect the first few rows of your data with the head() command. Try the command below in the Console:

```r
head(okcupid_profiles)
```

```
## # A tibble: 6 x 6
##     age diet          height pets              sex   status
##   <dbl> <chr>          <dbl> <chr>             <chr> <chr>
## 1    22 strictly any~     75 likes dogs and l~ m     single
## 2    35 mostly other      70 likes dogs and l~ m     single
## 3    38 anything          68 has cats          m     availa~
## 4    23 vegetarian        71 likes cats        m     single
## 5    29 <NA>              66 likes dogs and l~ m     single
## 6    29 mostly anyth~     67 likes cats        m     single
```

You can be even more specific and indicate you only want the first three row of your data with the head() command. Try the command below in the Console:

```r
head(okcupid_profiles, 3)
```

```
## # A tibble: 3 x 6
##     age diet          height pets              sex   status
##   <dbl> <chr>          <dbl> <chr>             <chr> <chr>
## 1    22 strictly any~     75 likes dogs and l~ m     single
## 2    35 mostly other      70 likes dogs and l~ m     single
## 3    38 anything          68 has cats          m     availa~
```

1.8.4 tail(): Check out the last few rows of data

You can inspect the last few rows of your data with the tail() command. Try the command below in the Console:

```
tail(okcupid_profiles)
```

```
## # A tibble: 6 x 6
##     age diet         height pets                sex   status
##   <dbl> <chr>         <dbl> <chr>               <chr> <chr>
## 1    31 <NA>             62 likes dogs          f     single
## 2    59 <NA>             62 has dogs            f     single
## 3    24 mostly anyt~     72 likes dogs and lik~ m     single
## 4    42 mostly anyt~     71 <NA>                m     single
## 5    27 mostly anyt~     73 likes dogs and lik~ m     single
## 6    39 <NA>             68 likes dogs and lik~ m     single
```

You can be even more specific and indicate you only want the last three row of your data with the tail() command. Try the command below in the Console:

```
tail(okcupid_profiles, 3)
```

```
## # A tibble: 3 x 6
##     age diet         height pets                sex   status
##   <dbl> <chr>         <dbl> <chr>               <chr> <chr>
## 1    42 mostly anyt~     71 <NA>                m     single
## 2    27 mostly anyt~     73 likes dogs and lik~ m     single
## 3    39 <NA>             68 likes dogs and lik~ m     single
```

1.8.5 summary(): Quick summaries

You can a short summary of your data with the summary() command. Note that we will use the summary() command in many places in the guide. The output of the summary() command changes depending on what you give it - that is put inside the brackets. You can give the summary() command many things such as data, the results of a regression analysis, etc.

Try the command below in the Console. You will see that summary() give the mean and median for each of the numeric variables (age and height).

```r
summary(okcupid_profiles)
```

```
##       age            diet               height     
##  Min.   : 18.0   Length:59946       Min.   : 1.0  
##  1st Qu.: 26.0   Class :character   1st Qu.:66.0  
##  Median : 30.0   Mode  :character   Median :68.0  
##  Mean   : 32.3                      Mean   :68.3  
##  3rd Qu.: 37.0                      3rd Qu.:71.0  
##  Max.   :110.0                      Max.   :95.0  
##                                     NA's   :3     
##      pets               sex               status         
##  Length:59946       Length:59946       Length:59946      
##  Class :character   Class :character   Class :character  
##  Mode  :character   Mode  :character   Mode  :character  
## 
## 
## 
## 
```

1.9 Run *vs.* Source with Echo *vs.* Source

There are different ways of running commands in R. So far you have used two of these. You can enter them into the Console as we have done already. Or you can put them in your script select the text and clickk the Run button. There are four ways of running commands in your script.

You can:

1. Console: Enter commands directly
2. Script: Select the command(s) and press the Run button.
3. Script: Source (Without Echo)
4. Script: Source With Echo

Two of these approaches involve using the Source button, see Figure 1.5. You bring up the options for the Source button, illustrated in this figure, by clicking on the small arrow to the right of the word Source.

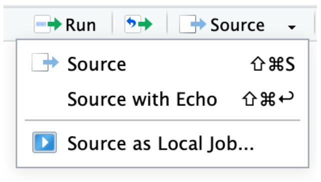


FIGURE 1.5: Source button options

1.9.1 Run select text

The Run button will run the text you highlight and present the relevant output. You have used this command a fair amount already.

I strongly suggest you ONLY use the Run button when testing a command to make sure it works or to debug a script. Or to run library(tidyverse) as you start working on your script so that you get the autocomplete options.

In general, you should always try to execute your R Scripts using the Source with Echo command (preceded by a Restart, see below). This ensures your script will work beginning to end for you in the future and for others that attempt to use it. Using the Run button in an ad lib basis can create output that is not reproducible.

1.9.2 Source (without Echo)

Source (without Echo) is not designed for the typical analysis workflow. It is mostly helpful when you run simulations. When you run Source (without Echo) much of the output you would wish to read is suppressed. In general, avoid this option. If you use it, you often won't see what you want to see in the output.

1.9.3 Source with Echo

The Source with Echo command runs all of the contents of a script and presents the output in the R console. This is the approach you should use to running your scripts in most cases.

Prior to running Source with Echo (or just Source), it's always a good idea to restart R. This makes sure you clear the computer memory of any errors from any previous runs.

So you should do the following EVERY time you run your script.

1. Use the menu item: **Session > Restart R**

2. Click the down arrow beside the Source button, and click on Source With Echo

This will clear potentially problematic previous stats, run the script commands, and display the output in the Console. Moving forward we will use this approach for running scripts. Once you have used Source wiht Echo once, you can just click the Source button and it will use Source with Echo automatically (without the need to use the pull down option for selecting Source with Echo).

- Using Restart R before you run a script, or R code in general, is a critical workflow tip.

1.10 Trying Source with Echo

Put the head(), tail(), and summary() command we used previously into your script. Then save your script using using the File > Save menu. You script should appear as below.

```
# Code written on: YYYY/MM/DD
# By: John Smith
library(tidyverse)

okcupid_profiles <- read_csv(file = "data_okcupid.csv")

head(okcupid_profiles)

tail(okcupid_profiles)

summary(okcupid_profiles)
```

Now do the following:

1. Use the menu item: **Session > Restart R**
2. Click the down arrow beside the Source button, and click on Source With Echo

You should see the output below:

```
# Code written on: YYYY/MM/DD
# By: John Smith
library(tidyverse)

okcupid_profiles <- read_csv(file = "data_okcupid.csv")
```

```
## Parsed with column specification:
## cols(
##   age = col_double(),
##   diet = col_character(),
##   height = col_double(),
##   pets = col_character(),
##   sex = col_character(),
##   status = col_character()
## )
```

```
head(okcupid_profiles)
```

```
## # A tibble: 6 x 6
##     age diet          height pets              sex   status
##   <dbl> <chr>          <dbl> <chr>             <chr> <chr>
## 1    22 strictly any~     75 likes dogs and l~ m     single
## 2    35 mostly other      70 likes dogs and l~ m     single
## 3    38 anything          68 has cats          m     availa~
## 4    23 vegetarian        71 likes cats        m     single
## 5    29 <NA>              66 likes dogs and l~ m     single
## 6    29 mostly anyth~     67 likes cats        m     single
```

```
tail(okcupid_profiles)
```

```
## # A tibble: 6 x 6
##     age diet         height pets                sex   status
##   <dbl> <chr>         <dbl> <chr>               <chr> <chr>
## 1    31 <NA>             62 likes dogs          f     single
## 2    59 <NA>             62 has dogs            f     single
## 3    24 mostly anyt~     72 likes dogs and lik~ m     single
## 4    42 mostly anyt~     71 <NA>                m     single
## 5    27 mostly anyt~     73 likes dogs and lik~ m     single
## 6    39 <NA>             68 likes dogs and lik~ m     single
```

```
summary(okcupid_profiles)
```

```
##       age            diet               height
##  Min.   : 18.0   Length:59946       Min.   : 1.0
##  1st Qu.: 26.0   Class :character   1st Qu.:66.0
##  Median : 30.0   Mode  :character   Median :68.0
##  Mean   : 32.3                      Mean   :68.3
##  3rd Qu.: 37.0                      3rd Qu.:71.0
##  Max.   :110.0                      Max.   :95.0
##                                     NA's   :3
##      pets               sex              status
##  Length:59946       Length:59946       Length:59946
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##
```

Congratulations you just ran your first script!

1.11 A few key points about

Sometimes you will need to send a command additional information. Moreover, that information often needs to be grouped together into a vector or a list before you can send it to the command. We'll learn more about doing so in the future but here is a quick over view of vectors and lists to provide a foundation for future chapters.

1.11.0.1 Vector of numbers

We can create a vector of only numbers using the "c" function - which you can think of as being short for "combine" (or concatenate). In the commands below we create a vector of a few even numbers called "even_numbers".

```
even_numbers <- c(2, 4, 6, 8, 10)
```

```
print(even_numbers)
```

```
## [1]  2  4  6  8 10
```

We can obtain the second number in the vector using the following notation:

```
print(even_numbers[2])
```

```
## [1] 4
```

1.11.0.2 Vector of characters

We can also create vectors using only characters. Note that I use **SHIFT RETURN** after each comma to move to the next line.

```
favourite_things <- c("copper kettles",
                      "woolen mittens",
                      "brown paper packages")
```

```
print(favourite_things)
```

```
## [1] "copper kettles"       "woolen mittens"
## [3] "brown paper packages"
```

As before, can obtain the second item in the vector using the following notation:

```
print(favourite_things[2])
```

```
## [1] "woolen mittens"
```

1.11.1 Lists

Lists are similar to vectors in that you can create them and access items by their numeric position. Vectors must be all characters or all numbers. Lists can be a mix of characters or numbers. Most importantly items in lists can be accessed by their label. Note that I use **SHIFT RETURN** after each comma to move to the next line in the code below.

```
my_list <- list(last_name = "Smith",
                first_name = "John",
                office_number = 1913)

print(my_list)
```

```
## $last_name
## [1] "Smith"
##
## $first_name
## [1] "John"
##
## $office_number
## [1] 1913
```

You can access an item in a list using double brackets:

```
print(my_list[2])
```

```
## $first_name
## [1] "John"
```

You can access an item in a list by its label/name using the dollar sign:

```
print(my_list$last_name)
```

```
## [1] "Smith"
```

```
print(my_list$office_number)
```

```
## [1] 1913
```

1.12 That's it!

Congratulations! You've reached the end of the introduction to R. Take a break, have a cookie, and read some more about R tomorrow!

2

Handling Data with the Tidyverse

2.1 Required

The following packages must be installed before starting this chapter.

Required Packages
tidyverse

The following data files are used in this chapter:

Required Data
data_okcupid.csv
data_experiment.csv

The files are available at: https://github.com/dstanley4/psyc6060bookdown

2.2 Objective

The objective of this chapter is to familiarize you with some key commands in the tidyverse. These commands are used in isolation of each other for the most part. In the next chapter we will use these commands in a more coordinated way as we load a data set and move it from raw data to data that is ready for analysis (i.e., analytic data). You can start this project by Starting the class assignment on R Studio Cloud that corresponds to the chapter name.

2.3 Using the Console

All of the commands in this chapter should be typed into the Console within R. If you see a command split over multiple lines, use SHIFT-RETURN (macOS) or SHIFT-ENTER (Windows) to move the next line that is part of the same command.

2.4 Basic tidyverse commands

If you inspect the Files tab on the lower-right panel in R Studio you will see the file data_okcupid.csv. The code below loads that file.

```
library(tidyverse)
okcupid_profiles <- read_csv("data_okcupid.csv")
```

You can see the first few rows of the data using the print() command. Each row presents a person whereas each column represents a variable. If you have a large number of columns you will only see the first several columns with this approach to viewing your data.

```
print(okcupid_profiles)
```

```
## # A tibble: 59,946 x 6
##      age diet         height pets              sex   status
##    <dbl> <chr>         <dbl> <chr>             <chr> <chr>
##  1    22 strictly an~     75 likes dogs and l~ m     single
##  2    35 mostly other     70 likes dogs and l~ m     single
##  3    38 anything         68 has cats          m     availa~
##  4    23 vegetarian       71 likes cats        m     single
##  5    29 <NA>             66 likes dogs and l~ m     single
##  6    29 mostly anyt~     67 likes cats        m     single
##  7    32 strictly an~     65 likes dogs and l~ f     single
##  8    31 mostly anyt~     65 likes dogs and l~ f     single
##  9    24 strictly an~     67 likes dogs and l~ f     single
## 10    37 mostly anyt~     65 likes dogs and l~ m     single
## # ... with 59,936 more rows
```

But it's also helpful just to see a list of the columns in the data with the glimpse() command:

```
glimpse(okcupid_profiles)
```

```
## Rows: 59,946
## Columns: 6
## $ age    <dbl> 22, 35, 38, 23, 29, 29, 32, 31, 24, 37, ...
## $ diet   <chr> "strictly anything", "mostly other", "an...
## $ height <dbl> 75, 70, 68, 71, 66, 67, 65, 65, 67, 65, ...
## $ pets   <chr> "likes dogs and likes cats", "likes dogs...
## $ sex    <chr> "m", "m", "m", "m", "m", "m", "f", "f", ...
## $ status <chr> "single", "single", "available", "single...
```

The glimpse() command is useful but is quickly allows you to see all of the columns. Moreover, it allows you to see the type for each column. Types were briefly discussed in the last chapter. Notice in the output beside each column name some columns are labeled which is short for double – a type of numeric column. Other columns columns are labeled which is short for character - meaning the columns contain characters. These designations will become important in the next chapter as we prepare data for analysis.

2.4.1 select()

The select() command allows you to obtain a subset of the columns in your data. The commands below can be used to obtain the age and height columns. You can read the command as: take the okcupid_profiles data and then select the age and height columns. The "%>%" symbol can be read as "and then". You can see that this code prints out the data with just the age and height columns. Remember, use SHIFT-ENTER or SHIFT-RETURN to move he next line in the block of code.

```
okcupid_profiles %>%
  select(age, height)
```

```
## # A tibble: 59,946 x 2
##      age height
##    <dbl>  <dbl>
##  1    22     75
##  2    35     70
##  3    38     68
##  4    23     71
##  5    29     66
##  6    29     67
##  7    32     65
##  8    31     65
##  9    24     67
## 10    37     65
## # ... with 59,936 more rows
```

Of course, it's usually of little help to just print the subset of the data. It's better to store it in a new data. In the command below we store the resulting data in a new data set called new_data.

```
new_data <- okcupid_profiles %>%
  select(age, height)
```

The glimpse() command shows us that only the age and height columns are in new_data.

```
glimpse(new_data)
```

```
## Rows: 59,946
## Columns: 2
## $ age    <dbl> 22, 35, 38, 23, 29, 29, 32, 31, 24, 37, ...
## $ height <dbl> 75, 70, 68, 71, 66, 67, 65, 65, 67, 65, ...
```

In the above example we indicated the columns we wanted to retain from the okcupid_profiles data using the select() command. However, we can also indicate the columns we want to drop from okcupid_profiles using a minus sign (-) in front of the columns we specify in the select() command.

```
new_data <- okcupid_profiles %>% select(-age, -height)
```

The glimpse() command shows us that we kept all the columns except the age and height columns when we created new_data.

```
glimpse(new_data)
```

```
## Rows: 59,946
## Columns: 4
## $ diet   <chr> "strictly anything", "mostly other", "an...
## $ pets   <chr> "likes dogs and likes cats", "likes dogs...
## $ sex    <chr> "m", "m", "m", "m", "m", "m", "f", "f", ...
## $ status <chr> "single", "single", "available", "single...
```

2.4.2 summarise()

The summarise() command can be used to generate descriptive statistics for a specified column. You can easily calculate column descriptive statistics using the corresponding commands for mean(), sd(), min(), max(), among others. In the example below we calculate the mean for the age column.

In code below mean(age, na.rm = TRUE) indicates to R that it should calculate the mean of the age column. The na.rm indicates how missing values should be handled. The na stands for not available; in R missing values are classified as Not Available or NA. The rm stands for remove. Consequently, na.rm is asking: "Should we remove missing values when calculating the mean?" The TRUE indicates that yes, missing values should be removed when calculating the mean. The result of this calculation is placed into a variable labelled age_mean, though we could have used any label we wanted instead of age_mean. We see that the mean of the age column is, with rounding, 32.3.

```
okcupid_profiles %>%
  summarise(age_mean = mean(age, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   age_mean
##      <dbl>
## 1     32.3
```

More than one calculation can occur in the same summarise() command. You can easily add the calculation for the standard deviation with the sd() command.

```
okcupid_profiles %>%
  summarise(age_mean = mean(age, na.rm = TRUE),
            age_sd = sd(age, na.rm = TRUE))
```

```
## # A tibble: 1 x 2
##   age_mean age_sd
##      <dbl>  <dbl>
## 1     32.3   9.45
```

Often this process does too much rounding. We can get more exact results by adding an as.data.frame() to the end of the commands.

```
okcupid_profiles %>%
  summarise(age_mean = mean(age, na.rm = TRUE),
            age_sd = sd(age, na.rm = TRUE)) %>%
  as.data.frame()
```

```
##   age_mean age_sd
## 1    32.34  9.453
```

2.4.3 filter()

The filter() command allows you to obtain a subset of the rows in your data. In the example below we create a new data set with just the males from the original data.

Notice the structure of the original data below in the glimpse() output. There is a column called sex that uses m and f to indicate male and female, respectively. Also notice that there are 59946 rows in the okcupid_profiles data.

```
glimpse(okcupid_profiles)
```

```
## Rows: 59,946
## Columns: 6
## $ age    <dbl> 22, 35, 38, 23, 29, 29, 32, 31, 24, 37, ...
## $ diet   <chr> "strictly anything", "mostly other", "an...
## $ height <dbl> 75, 70, 68, 71, 66, 67, 65, 65, 67, 65, ...
## $ pets   <chr> "likes dogs and likes cats", "likes dogs...
## $ sex    <chr> "m", "m", "m", "m", "m", "m", "f", "f", ...
## $ status <chr> "single", "single", "available", "single...
```

We use the the filter command to select a subset of the rows based on the contents of any column. In this case the sex column. Notice the double equals sign is used to indicate "equal to". The reason a double equals sign is used here (instead of a single equals sign) is to distinguish it from the use of the single equals sign in the summarise command above. In the summarise command above, the single equal sign was used to indicate "assign to". That is assign to age_mean the mean of the column age after it is calculated calculated. A single equals sign indicates "assign to" whereas a double equals sign indicates "is equal to".

```
okcupid_males <- okcupid_profiles %>%
  filter(sex == "m")
```

We use glimpse() to inspect these all male data. Notice that only the letter m is in the sex column - indicating only males are in the data set. Also notice that there are 35829 rows in the okcupid_males data; fewer people because males are a subset of the total number of rows.

```
glimpse(okcupid_males)
```

```
## Rows: 35,829
## Columns: 6
## $ age    <dbl> 22, 35, 38, 23, 29, 29, 37, 35, 28, 24, ...
## $ diet   <chr> "strictly anything", "mostly other", "an...
## $ height <dbl> 75, 70, 68, 71, 66, 67, 65, 70, 72, 72, ...
## $ pets   <chr> "likes dogs and likes cats", "likes dogs...
## $ sex    <chr> "m", "m", "m", "m", "m", "m", "m", "m", ...
## $ status <chr> "single", "single", "available", "single...
```

The filter command can be commbined with the summarise command to get the descriptive statistics for males without the hassle of creating a new data. This is again done using the %>% "and then" operator.

```
okcupid_profiles %>%
  filter(sex == "m") %>%
  summarise(age_mean = mean(age, na.rm = TRUE),
            age_sd = sd(age, na.rm = TRUE))
```

```
## # A tibble: 1 x 2
##   age_mean age_sd
##      <dbl>  <dbl>
## 1     32.0   9.03
```

We see that for the 35829 females the mean age is 32.0 and the standard deviation is 9.0.

Likewise, we can obtain the descriptive statistics for females with only a slight modification, changing m to f in the filter command:

```
okcupid_profiles %>%
  filter(sex == "f") %>%
  summarise(age_mean = mean(age, na.rm = TRUE),
            age_sd = sd(age, na.rm = TRUE))
```

```
## # A tibble: 1 x 2
##   age_mean age_sd
##      <dbl>  <dbl>
## 1     32.8   10.0
```

We see that for the 24117 females the mean age is 32.8 and the standard deviation is 10.0.

2.4.4 group_by()

The process we used with the filter command would quickly become onerous if we had many subgroups for a column. Consequently, it's often better to use the group() command to calculate descriptive statistics for the levels (e.g., male/female) of a variable. By telling the computer to group_by() sex the summarise command is run separately for every level of sex (i.e., m and f).

```
okcupid_profiles %>%
  group_by(sex) %>%
  summarise(age_mean = mean(age, na.rm = TRUE),
            age_sd = sd(age, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 3
##   sex   age_mean age_sd
##   <chr>    <dbl>  <dbl>
## 1 f         32.8  10.0
## 2 m         32.0   9.03
```

Fortunately, it's possible to use more than one grouping varible with the group_by() command. In the code below we group by sex and status (i.e., dating status).

```
okcupid_profiles %>%
  group_by(sex, status) %>%
  summarise(age_mean = mean(age, na.rm = TRUE),
            age_sd = sd(age, na.rm = TRUE))
```

```
## `summarise()` regrouping output by 'sex' (override with `.groups` argument)
```

```
## # A tibble: 10 x 4
## # Groups:   sex [2]
##    sex   status         age_mean age_sd
##    <chr> <chr>             <dbl>  <dbl>
##  1 f     available          32.2   8.54
##  2 f     married            33.7   8.13
##  3 f     seeing someone     28.1   6.44
##  4 f     single             33.0  10.2
##  5 f     unknown            27.8   5.91
##  6 m     available          34.8   9.40
##  7 m     married            38.7  10.1
##  8 m     seeing someone     30.8   7.06
##  9 m     single             31.9   9.04
## 10 m     unknown            40.7   8.87
```

The resulting output provide for age the mean and standard deviation for every combination of sex and dating status. The first five rows provide output for females at every level of dating status whereas the subsequent five rows provide output for males at every level of dating status.

2.4.5 mutate()

The mutate() command can be used to calculate a new column in a data. In the example below we calculate a new column called age_centered which is the new version of the age_column where the mean of the column has been removed from every value. This is merely an example of the many different types of calculations can can perform to create a new column using mutate().

```r
okcupid_profiles <- okcupid_profiles %>%
  mutate(age_centered = age - mean(age, na.rm = TRUE))
```

Notice that the glimpse() command reveals that after we use the mutate() command there is a new column called age_centered.

```r
glimpse(okcupid_profiles)
```

```
## Rows: 59,946
## Columns: 7
## $ age          <dbl> 22, 35, 38, 23, 29, 29, 32, 31, 24...
## $ diet         <chr> "strictly anything", "mostly other...
## $ height       <dbl> 75, 70, 68, 71, 66, 67, 65, 65, 67...
## $ pets         <chr> "likes dogs and likes cats", "like...
## $ sex          <chr> "m", "m", "m", "m", "m", "m", "f",...
## $ status       <chr> "single", "single", "available", "...
## $ age_centered <dbl> -10.3403, 2.6597, 5.6597, -9.3403,...
```

2.5 Advanced tidyverse commands

In this advanced selection we revisit the commands from the basic tidyverse section but use more complicated code to either select or apply an action to more than one column at a time. We will indicate the columns that we want to select or apply an action to using: starts_with(), ends_with(), contains(), matches(), or where(). The first four of these are used to indicate columns based on column names. In contrast, the last command, where(), is used to indicate the columns based on the column type (numeric, character, factor, etc.).

We will review all five commands for indicating the columns we want in the select() selection below. Following that we will, for brevity, typically use only one of the five commands when illustrating how they work with summarise() and mutate().

We begin by loading a new data.

```
library(tidyverse)
data_exp <- read_csv("data_experiment.csv")
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   sex = col_character(),
##   t1_vomit = col_double(),
##   t1_aggression = col_double(),
##   t2_vomit = col_double(),
##   t2_aggression = col_double()
## )
```

The glimpse() command reveals that this is a small data where every row represents one rat. The sex of the rat is recorded as well as, for each of two time points, a rating of vomiting and aggression.

```
glimpse(data_exp)
```

```
## Rows: 6
## Columns: 6
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ sex           <chr> "male", "female", "male", "female...
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> 6, 7, 6, 7, 5, 8
```

2.5.1 select()

2.5.1.1 select() using column name

2.5.1.1.1 starts_with()

starts_with() allows us to select columns based on how the column name begins. Here we put the columns that begin with “t1” into a new data called data_time1.

```
data_time1 <- data_exp %>%
  select(starts_with("t1"))
```

The glimpse command shows us the new data only contains the columns that begin with “t1”

```
glimpse(data_time1)
```

```
## Rows: 6
## Columns: 2
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
```

2.5.1.1.2 ends_with()

ends_with() allows us to select columns based on how the column name ends. Here we put the columns that end with “aggression” into a new data set called data_aggression.

```
data_aggression <- data_exp %>%
  select(ends_with("aggression"))
```

```
glimpse(data_aggression)
```

```
## Rows: 6
## Columns: 2
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
## $ t2_aggression <dbl> 6, 7, 6, 7, 5, 8
```

2.5.1.1.3 contains()

contains() allows us to select columns based on the contents of the column name. Here we put the columns that have ”_” in the name into a new data set called new_data.

```
new_data <- data_exp %>%
  select(contains("_"))
```

```
glimpse(new_data)
```

```
## Rows: 6
## Columns: 4
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> 6, 7, 6, 7, 5, 8
```

2.5.1.1.4 matches()

It’s also possible to use *regex* (regular expressiona) to select columns. Regex is a powerful way to specify search/matching requirements for text - in this case the text of column names. An explanation of regex is beyond the scope of this chapter. But the example below selects any column with and underscore in the column name followed by any character. The result is the same as the above for the contains() command. However, the matches() command is more flexible than the contains() command and can take into account substantially more complicated situations.

```
data_matched<- data_exp %>%
  select(matches("(_.)"))
```

You can see the columns selected using regex:

```
glimpse(data_matched)
```

```
## Rows: 6
## Columns: 4
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> 6, 7, 6, 7, 5, 8
```

You can learn about regex at RegexOne[1] and test your regex specification at Regex101[2]. Ideally though, as we discuss in the next chapter, you can use naming conventions that are sufficiently considered that you don't need regex, or only rarely. The reason for this is that regex can be challenging to use. As Twitter user @ThatJenPerson noted "Regex is like tequila: use it to try to solve a problem and now you have two problems." Nonetheless, at one or two points in the future we will use regex to solve a problem (but not tequila).

2.5.1.2 select() using column type

If many cases we will want to select or perform on action on a column based on whether the column is a numeric, character, or factor column (indicated in glimpse output as dbl, chr, and fct, respectively). We will learn more about factors later in this chapter. Each of these column types can be selected by using is.numeric, is.character, or is.factor, respectively, in combination with the where() command.

We can select numeric columns using where() and is.numeric:

```
data_numeric_columns <- data_exp %>%
  select(where(is.numeric))
```

You can see the new data contains only the numeric columns:

```
glimpse(data_numeric_columns)
```

```
## Rows: 6
## Columns: 5
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> 6, 7, 6, 7, 5, 8
```

We can select numeric columns using where() and is.character:

```
data_character_columns <- data_exp %>%
  select(where(is.character))
```

You can see the new data contains only the character columns:

[1] https://regexone.com
[2] https://regex101.com

```
glimpse(data_character_columns)
```

```
## Rows: 6
## Columns: 1
## $ sex <chr> "male", "female", "male", "female", "male",...
```

If a future chapter you will see how we can select factors using where(is.factor).

2.5.2 summarise()

The summarise() command can summarise multiple columns when combined with starts_with(), ends_with(), contains(), matches(), and where(). However, to use these powerful tools for indicating columns with the summarise command we need to help of the across command (i.e., across multiple columns).

If we want to obtain the mean of all the columns that start with "t1" we use the commands below. The across command requires that we indicate the columns we want via the .cols argument and the command/function we want to run on those columns via the .fns argument. In the example below, we also add na.rm = TRUE at the end; this is something we send to the mean command to let it know how we want to handle missing data. We add as.data.frame() to get a larger number of decimals.

```
data_exp %>%
  summarise(across(.cols = starts_with("t1"),
                   .fns = mean,
                   na.rm = TRUE)) %>%
  as.data.frame()
```

```
##   t1_vomit t1_aggression
## 1    1.833           5.5
```

If you want to get fancy, you can also add this .names argument below which tells R to call label each output mean by the column name followed by "_mean".

```
data_exp %>%
  summarise(across(.cols = starts_with("t1"),
                   .fns = mean,
                   na.rm = TRUE,
                   .names = "{col}_mean"))  %>%
  as.data.frame()
```

```
##   t1_vomit_mean t1_aggression_mean
## 1         1.833                5.5
```

Often you want to calculate more than one statistic for each column. For example, you might want the mean, standard deviation, min, and max. These statistics can be calculated via the mean, sd, min, and max commands, respectively. However, you need to create a list with the statistics you desire.

Below we create a list of the descriptive statistics we desire called desired_statistics, but you can use any name you want. This list only needs to be specified once, but we will repeat it in the examples below for clarity.

```
desired_descriptives <- list(
  mean = ~mean(.x, na.rm = TRUE),
  sd = ~sd(.x, na.rm = TRUE)
)
```

Once you have created the list of descriptive statistics you want you can run the command below to obtain those statistics. However, as you will see the output is too wide to be helpful.

```
data_exp %>%
  summarise(across(.cols = starts_with("t1"),
                   .fns = desired_descriptives)) %>%
  as.data.frame()
```

```
##   t1_vomit_mean t1_vomit_sd t1_aggression_mean
## 1         1.833       1.169                5.5
##   t1_aggression_sd
## 1            1.871
```

Consequently, we add the t() command (i.e., transpose command) to the end of the summarise request to get a more readable list of statistics:

```
desired_descriptives <- list(
  mean = ~mean(.x, na.rm = TRUE),
  sd = ~sd(.x, na.rm = TRUE)
)

data_exp %>%
  summarise(across(.cols = starts_with("t1"),
                   .fns = desired_descriptives)) %>%
  as.data.frame() %>%
  t()
```

```
##                       [,1]
## t1_vomit_mean      1.833
## t1_vomit_sd        1.169
## t1_aggression_mean 5.500
## t1_aggression_sd   1.871
```

Note that in the across command above we could also have used: ends_with(), contains(), matches(), or where().

2.5.3 mutate()

The mutate command can also be applied to multiple columns using the across() command. However, sometimes we need to embed our calculation in a custom function. Below is a custom function called make_centered. This custom function takes the values in a column an subtracts the column mean from each value in the column. This is the same task we did the mutate() command in the basic tidyverse section above.

```
make_centered <- function(values) {
  values_out <- values - mean(values, na.rm = TRUE)
  return(values_out)
}
```

The glimpse() command shows us all the column names. Also notice the values in the agresssion columns are integers.

```
glimpse(data_exp)
```

```
## Rows: 6
## Columns: 6
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ sex           <chr> "male", "female", "male", "female...
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> 5, 6, 4, 7, 3, 8
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> 6, 7, 6, 7, 5, 8
```

We combine the mutate() command with the across() command and our custom make_centered() command below. The command “centers” or subtracts the mean from any column that ends with “aggression”.

```
data_exp <- data_exp %>%
  mutate(across(.cols = ends_with("aggression"),
                .fns = make_centered))
```

You can see via the glimpse() output that the contents of all the columns that end with "aggression" have changed. Every value in one these columns has had the column mean subtracted from it.

```
glimpse(data_exp)
```

```
## Rows: 6
## Columns: 6
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ sex           <chr> "male", "female", "male", "female...
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> -0.5, 0.5, -1.5, 1.5, -2.5, 2.5
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> -0.5, 0.5, -0.5, 0.5, -1.5, 1.5
```

Note that in the across command above we could also have used: starts_with(), contains(), matches(), or where().

2.5.3.1 mutate() across rows

Researchers often want to average within rows and across columns to create a new column. That is for each participant (i.e., rat in the current data) we might want to calculate a vomit score that is the average of the two time points (that we will call vomit_avg).

To average within rows (and across columns) we use the rowwise() command to inform R of our intent. After we do the necessary calculations though we have to shut off the rowwise() calculation state by using the ungroup() command. As well, when we are averaging within rows we have to use c_across() instead of across(). The commands below creates a new column called vomit_avg which the average of the vomit ratings across both times. As before, we also include na.rm = TRUE so the computer drops missing values (if present) when calculating the mean.

```
data_exp <- data_exp %>%
  rowwise() %>%
  mutate(vomit_avg = mean( c_across(cols = ends_with("vomit")),
                           na.rm = TRUE)) %>%
  ungroup()
```

You can see the new column we created with the glimpse() command:

```
glimpse(data_exp)
```

```
## Rows: 6
## Columns: 7
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ sex           <chr> "male", "female", "male", "female...
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> -0.5, 0.5, -1.5, 1.5, -2.5, 2.5
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> -0.5, 0.5, -0.5, 0.5, -1.5, 1.5
## $ vomit_avg     <dbl> 2.5, 1.5, 0.5, 2.5, 1.5, 1.5
```

The print() command could make it easier to see the new column is the average of the other two, but if we used the command below it wouldn't work. Why? There are too many columns in the data set so only the first few columns are show.

```
print(data_exp)
```

To see how the new column, vomit_avg, is the average of the other vomit columns we use the select command before print(). This prints only the relvant columns. When this is done, it's easy to see how the values in the vomit_avg column are the mean of the other two columns.

```
data_exp %>%
  select(contains("vomit")) %>%
  print()
```

```
## # A tibble: 6 x 3
##   t1_vomit t2_vomit vomit_avg
##      <dbl>    <dbl>     <dbl>
## 1        3        2       2.5
## 2        2        1       1.5
## 3        0        1       0.5
## 4        3        2       2.5
## 5        2        1       1.5
## 6        1        2       1.5
```

2.5.3.2 mutate() for factors

In R it it is critical that you identify categorical variable as categorical variables. In R categorical variables are referred to as factors. For humans, a factor like sex has three possible levels: female, male, intersex.

An inspection of the glimpse() command output reveals that the sex column has the type character - as indicated by . Also notice as you inspect this output that we use words (e.g., female) to indicate the sex in the column rather than a number represent a female participant (e.g., 2). This is the preferred but less common approach to entering data.

```
glimpse(data_exp)
```

```
## Rows: 6
## Columns: 7
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ sex           <chr> "male", "female", "male", "female...
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> -0.5, 0.5, -1.5, 1.5, -2.5, 2.5
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> -0.5, 0.5, -0.5, 0.5, -1.5, 1.5
## $ vomit_avg     <dbl> 2.5, 1.5, 0.5, 2.5, 1.5, 1.5
```

We need to convert the sex column to a factor in order for R to handle it appropriately in analyses. Failure to indicate column is a factor could result in R conducting all the analyses and presenting the incorrect results. Consequently, it is critical that we covert the column to a factor. Fortunately, that is easily done using the as_factor() command (there is also an as.factor command if as_factor won't work for some reason).

We convert the sex column to a factor with this code:

```
data_exp <- data_exp %>%
  mutate(sex = as_factor(sex))
```

You can confirm this worked with the glimpse() command:

```
glimpse(data_exp)
```

```
## Rows: 6
## Columns: 7
## $ id            <dbl> 1, 2, 3, 4, 5, 6
## $ sex           <fct> male, female, male, female, male,...
## $ t1_vomit      <dbl> 3, 2, 0, 3, 2, 1
## $ t1_aggression <dbl> -0.5, 0.5, -1.5, 1.5, -2.5, 2.5
## $ t2_vomit      <dbl> 2, 1, 1, 2, 1, 2
## $ t2_aggression <dbl> -0.5, 0.5, -0.5, 0.5, -1.5, 1.5
## $ vomit_avg     <dbl> 2.5, 1.5, 0.5, 2.5, 1.5, 1.5
```

If you entered your data using words for each level of sex (e.g., male, female) you're done at this point. However, if you used numbers to represent each level of sex in your data there is one more step. Imagine your data was entered in a poorly advised manner, such that 1 was used to indicate male, 2 was used to indicate female, and 3 was used to indicate intersex. If this was the case, you need to indicate to R what each of those value represents. We do that with the code below.

```
data_exp <- data_exp %>%
  mutate(sex = fct_recode(sex,
                          male = "1",
                          female = "2",
                          intersex = "3"))
```

2.6 Using help

In order to become an efficient at analyzing data using R, you will need to become adapt at reading and understanding the help files associated with each command. After you have activated a package using the library command (e.g., library(tidyverse)) you can access the help page for every command in that package. To access the help page simply type a question mark followed by the command you want to know how to use (no space between them). The code below bring up the help page for the select() command. Notice I put the library() command first - just an reminder that this needs to be done prior to using help for that package. Try the commands below in the Console:

```
library(tidyverse)
?select
```

Examine the page that appears on the Help tab in the panel in the lower right of your screen. Read through the help file comparing what you read there to what we have learned about the select command. Notice how the help file tells you about the argument you send into the select() command and also what the select command returns when it receives those commands. Pay particular attention to the examples near the bottom of the help page.

At the very bottom of the help page you will see [Package dplyr version 1.0.0 Index]. This tell you the select() command is from the dplyr package (part of the tidyverse). Notice that the word Index is underlined. Click on the word Index. You will be presented with list of other commands in the dplyr package.

As you become more experienced with R help pages how you will learn to use new commands. Examine the help pages for the commands below by typing a question mark into the Console followed by the command name. Note that for filter and starts_with you will be presented with a menu instead of help page. This typically occurs because the command is in more than one package. If this does occur, read through the options you are presented with to try and figure out which one you wanted Typically, you want the first option. If you're not sure, try one. It's it's not what you want use the back arrow in the Help panel to go back and pick another one.

- mutate
- filter
- starts_with

2.7 Base R vs tidyverse

All of the commands used to this point in the chapter have been the tidyverse approach to using R. That is the approach we will normally use. However, it's important to note that there is another way of using R, called base R.

Sometimes students have problems with their code when they mix and match these approaches using a bit of both. We will be using the tidyverse approach to using R but on the internet you will often see sample code that uses the older base R approach. A bit of background knowledge is helpful for understanding why we do things one way (e.g., read_csv with the tidyverse) instead of another (e.g., read.csv with base R).

2.7.1 Tibbles vs. data frames

When you load data into R it is typically represented in one of two formats inside the computer - depending on the command you used. The original format for representing a data set in R is the data frame. You will see this term used frequently when you read about R. When you load data using read.csv your data is loaded into a data frame in the computer. That is your data is represented in the memory of the computer in particular format and structure called a data frame. This contrasts with the newer tidyverse approach to representing data in the computer called a tibble - which is just an newer more advanced version of the data frame.

2.7.2 read.csv and data frames

When you read data into R using the command read.csv (with a period) you load the data into a data frame (base R).

```
my_dataframe <- read.csv(file = "data_okcupid.csv")
```

Notice that when you print a data frame it does not show you the number of rows or columns above the data like our example did with the okcupid_profiles data. Likewise, it does not show you the type of data in each column (e..g, dbl, fct, chr. But, it also list all of your data rather than just the first few rows (as the tibble does). As a result in the output below I show only the first 10 rows of the output - because all the rows are printed in your Console with a data frame (too much to show here).

```
print(my_dataframe)
```

```
##    age              diet height                      pets
## 1   22 strictly anything     75 likes dogs and likes cats
## 2   35      mostly other     70 likes dogs and likes cats
## 3   38          anything     68                  has cats
## 4   23        vegetarian     71                likes cats
## 5   29              <NA>     66 likes dogs and likes cats
## 6   29   mostly anything     67                likes cats
## 7   32 strictly anything     65 likes dogs and likes cats
## 8   31   mostly anything     65 likes dogs and likes cats
## 9   24 strictly anything     67 likes dogs and likes cats
## 10  37   mostly anything     65 likes dogs and likes cats
##    sex    status
## 1    m    single
## 2    m    single
## 3    m available
## 4    m    single
## 5    m    single
## 6    m    single
## 7    f    single
## 8    f    single
## 9    f    single
## 10   m    single
```

2.7.3 read_csv and tibbles

When you read data into R using the command read_csv (with an underscore) you load the data into a tibble (tidyverse).

```
my_tibble <- read_csv(file = "data_okcupid.csv")
```

```
## Parsed with column specification:
## cols(
##   age = col_double(),
##   diet = col_character(),
##   height = col_double(),
##   pets = col_character(),
##   sex = col_character(),
##   status = col_character()
## )
```

The tibble is modern version of the data frame. Notice that when you print a tibble it DOES show you the number of rows and columns. As well, it shows you the type of data in each column. Importantly, the tibble only provides the first few rows of output so it doesn't fill your screen.

```
print(my_tibble)
```

```
## # A tibble: 59,946 x 6
##      age diet         height pets              sex   status
##    <dbl> <chr>         <dbl> <chr>             <chr> <chr>
##  1    22 strictly an~     75 likes dogs and l~ m     single
##  2    35 mostly other     70 likes dogs and l~ m     single
##  3    38 anything         68 has cats          m     availa~
##  4    23 vegetarian       71 likes cats        m     single
##  5    29 <NA>             66 likes dogs and l~ m     single
##  6    29 mostly anyt~     67 likes cats        m     single
##  7    32 strictly an~     65 likes dogs and l~ f     single
##  8    31 mostly anyt~     65 likes dogs and l~ f     single
##  9    24 strictly an~     67 likes dogs and l~ f     single
## 10    37 mostly anyt~     65 likes dogs and l~ m     single
## # ... with 59,936 more rows
```

In short you should always use tibbles (i.e., use read_csv not read.csv). The differences between data frames and tibbles run deeper than the superficial output provided here. On some rare occasions an old package or command may not work with a tibble so you need to make it a data frame. You can do so with the commands below. We will flag these rare occurances to you in the next.

```r
# Create a data frame from a tibble
new_dataframe <- as.data.frame(my_tibble)
```

3

Making your data ready for analysis

3.1 Required Packages

The following packages must be installed before starting this chapter.

Required Packages
apaTables
HMisc
janitor
psych
skimr
tidyverse

Important Note: that you should NOT use library(psych) at any point. There are major conflicts between the psych package and the tidyverse. We will access the psych package commands by preceding each command with psych:: instead of using library(psych).

The following data files are used in this chapter:

Required Data
data_ex_between.csv
data_ex_within.csv
data_food.csv
data_item_scoring.csv
data_item_time.csv

The files are available at: `https://github.com/dstanley4/psyc6060bookdown`

3.2 Objective

In this chapter we strongly advocate for you using a naming convention for file, variable, and column names. This convention will save you hours of hassles and permit easy application of certain tidyverse command. I must stress though that although the naming convention I advocate is based on the tidyverse style guide, it is not “right” - there are other naming conventions you can use. And any naming convention is better than no naming convention. The naming convention we advocate for here will solve many problems for you. I encourage to use this system for for weeks or months over many projects - until you see the benefits of this system, and correspondingly it’s shortcomings. After you are well versed in the strengths/weaknesses of the naming conventions used here you may choose to create your own naming convention system.

3.3 Context

Due to a number of high profile failure to replicate study results (Nosek, 2015) it’s become increasingly clear that there is a general crisis of confidence in many areas of science (Baker, 2016). Statistical (and other) explanations have been offered (Simmons et al., 2011) for why it’s hard to replicate results across different sets of data. However, scientists are also finding it challenging to recreate the numbers in their own papers using their own data. Indeed, the editor of Molecular Brain asked authors to submit the data used to create the numbers in published papers and found that the wrong data was submitted for 40 out of 41 papers (Miyakawa, 2020).

Consequently, some researchers have suggested that it is critical to distinguish between replication and reproducibility (Patil P., 2019). Replication refers to trying to obtain the same result from a different data sets. Reproducibility refers to trying to obtain the same results from the same data set. Unfortunately, some authors use these two terms interchangeably and fail to make any distinction between them. I encourage you to make the distinction and the use the terms consist with use suggested by (Patil P., 2019).

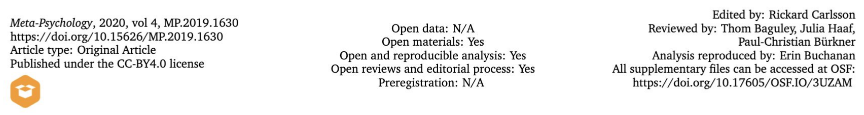
It may seem that reproducibility should be given - but it’s not. Correspondingly, there a trend is for journals and authors to adopt Transparency and Openness Promotion (TOP) guidelines[1]. These guidelines involve such things as making your materials, data, code, and analysis scripts available on public

[1]https://www.cos.io/our-services/top-guidelines

repositories so anyone can check your data. A new open science journal rating system has even emerged called the TOP Factor[2].

The idea is not that open science articles are more trustworthy that other types of articles – the idea is that trust doesn't play a role. Anyone can inspect the data using the scripts and data provided by authors. It's really just the same as making your science available for auditing the way financial records can be audited. But just like in the world of business some people don't like the idea of make it possible for others to audit their work. The problems reported at Molecular Brain (doubtless is common to many journals) are likely avoided with open science - because the data and scripts needed to reproduce the numbers in the articles are uploaded prior to publication.

The TOP open science guidelines have made an impact and some newer journals, such as Meta Psychology, have fully embraced open science. Figure 3.1 shows the header from an article[3] in Meta Psychology that clearly delineates the open science attributes of the article that used computer simulations (instead of participant data). Take note that the header even specifies who checked that the analyses in the article were reproducible.


Meta-Psychology, 2020, vol 4, MP.2019.1630
https://doi.org/10.15626/MP.2019.1630
Article type: Original Article
Published under the CC-BY4.0 license

Open data: N/A
Open materials: Yes
Open and reproducible analysis: Yes
Open reviews and editorial process: Yes
Preregistration: N/A

Edited by: Rickard Carlsson
Reviewed by: Thom Baguley, Julia Haaf, Paul-Christian Bürkner
Analysis reproduced by: Erin Buchanan
All supplementary files can be accessed at OSF: https://doi.org/10.17605/OSF.IO/3UZAM

FIGURE 3.1: Open science in an article header

In Canada, the majority of university research is funded by the Federal Government's Tri-Agency (i.e., NSERC, SSHRC, CIHR). The agency has a new draft Data Management Policy[4] in which they state that "*The agencies believe that research data collected with the use of public funds belong, to the fullest extent possible, in the public domain and available for reuse by others.*" The perspective of the funding agency on data ownership differs substantially from that of some researchers who incorrectly believe "they own their data". In Canada at least, the government makes it clear that when tax payers fund research (through the Tri-Agency) the research data is public property. Additionally the Tri-Agency Data Management policy clearly indicates the responsibilities of funded researchers:

"Responsibilities of researchers include:

- incorporating data management best practices into their research;

[2] https://topfactor.org
[3] https://open.lnu.se/index.php/metapsychology/article/view/1630/2266
[4] https://www.ic.gc.ca/eic/site/063.nsf/eng/h_83F7624E.html

- developing data management plans to guide the responsible collection, formatting, preservation and sharing of their data throughout the entire life cycle of a research project and beyond;
- following the requirements of applicable institutional and/or funding agency policies and professional or disciplinary standards;
- acknowledging and citing data sets that contribute to their research; and
- staying abreast of standards and expectations of their disciplinary community."

As a result of this perspective on data, it's important that you think about structuring your data for reuse by yourself and others before you collect it. Toward this end, properly documenting your data file and analysis scripts is critical.

3.4 Begin with the end in mind

In this chapter we will walk you though the steps from data collection, data entry, loading raw data, and the creation of data you will analyze (analytic data) via pre-processing scripts. These steps are outlined in Figure 3.2. This figure makes a clear distinction between raw data and analytic data. Raw data refers to the data as you entered it into a spreadsheet or received it from survey software. Analytic data is the data that has been structured and processed so that it is ready for analysis. This pre-processing could include such things as identifying categorical variables to the computer, averaging multiple items into a scale scale scores, and other tasks.

It's critical that you don't think of the analysis of your data as being completely removed from the data collection and data entry choices you make. Poor choices at the data collection and data entry stage can make your life substantially more complicated when it comes time to write the pre-processing script that will convert your raw data to analytic data. The mantra of this chapter is *begin with the end in mind*.

It's difficult to being with the end in mind when you haven't read later chapters. So here we will be provide you with some general thoughts around different approaches to structuring data files and the naming conventions you use when creating those data files.

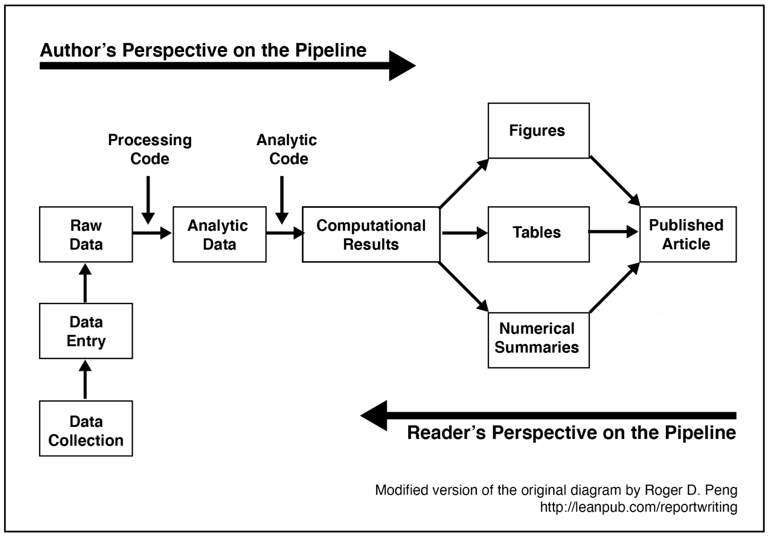

Author's Perspective on the Pipeline
Processing Code
Analytic Code
Raw Data
Analytic Data
Computational Results
Figures
Tables
Numerical Summaries
Published Article
Data Entry
Data Collection
Reader's Perspective on the Pipeline
Modified version of the original diagram by Roger D. Peng
http://leanpub.com/reportwriting

FIGURE 3.2: Data science pipeline by Roger Peng.

3.4.1 Structuring data: Obtaining tidy data

When conducting analyses in R it is typically necessary to have data in a format called tidy data (Wickham, 2014). tidy data[5] as defined by Hadley involves (among other requirements) that:

1. Each variable forms a column.
2. Each observation forms a row.

The tidy data format can be initially challenging for some researchers to understand because it is based on thinking about, and structuring data, in terms of observations/measurements instead of participants. In this section we will describe common approaches to entering animal and human participant data and how they can be done keeping the tidy data requirement in mind. It's not essential that data be entered in a tidy data format but it is essential that you enter data in manner that makes it easy to later convert data to a tidy data format. When dealing with animal or human participant data it's common to enter data into a spreadsheet. Each row of the spreadsheet is typically used to represent a single participant and each column of the spreadsheet is used to represent a variable.

Between participant data. Consider Table 3.3 which illustrates between participant data for six human participants running 5 kilometers. The first

[5] `https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html`

TABLE 3.3: Between participant data entered one row per participant

id	sex	elapsed_time
1	male	40
2	female	35
3	male	38
4	female	33
5	male	42
6	female	36

TABLE 3.4: Within participant data entered one row per participant

id	sex	march	may	july
1	male	40	37	35
2	female	35	32	30
3	male	38	35	33
4	female	33	30	28
5	male	42	39	37
6	female	36	33	31

column is id, which indicates there are six unique participants and provides and identification number for each of them. The second column is sex, which is a variable, and there is one observation per for row, so sex also conforms to the tidy data specification. Finally, there is a last column five_km_time which is a variable with one observation per row – also conforming to tidy data specification. Thus, single occasion between subject data like this conforms to the tidy data specification. There is usually nothing you need to do to convert between participant data (or cross-sectional data) to be in a tidy data format.

Within participant data. Consider Table 3.4 which illustrates within participant data for six human participants running 5 kilometers - but on three different occasions. The first column is id, which indicates there are six unique participants and provides and identification number for each of them. The second column is sex, which is a variable, and there is one observation per for row, so sex also conforms to the tidy data specification. Next, there are three different columns (march, may, july) representing each of which contains and elapsed_time for the runner in a different month. Elapsed run times are spread out over three columns so elapse_time is not in a tidy data format. Moreover, it's not clear from the data file that march, may, and july are levels of a variable called occasion. Nor is it clear that elapsed_times are recorded in each of those columns (i.e., the dependent variable is unknown/not labeled). Although this format is fine as a data entry format it clearly has problems associated with it when it comes time to analyze your data.

TABLE 3.5: A tidy data version of the within participant data

id	sex	occasion	elapsed_time
1	male	march	40
1	male	may	37
1	male	july	35
2	female	march	35
2	female	may	32
2	female	july	30
3	male	march	38
3	male	may	35
3	male	july	33
4	female	march	33
4	female	may	30
4	female	july	28
5	male	march	42
5	male	may	39
5	male	july	37
6	female	march	36
6	female	may	33
6	female	july	31

Thus, a major problem with entering repeated measures data in the one row per person format is that there are hidden variables in the data and you need insider knowledge to know what the columns represent. That said, this is not necessarily a terrible way to enter your data as long as you have all of this missing information documented in a data code book.

Disadvantages one row per participant	Advantages one row per participant
1) Predictor variable (*occasion*) is hidden and spread over multiple columns 2) Unclear that each month is a level of the predictor variable *occasion* 3) Dependent variable (*elapsed_time*) is not indicated 4) Unclear that *elapsed_time* is the measurement in each month column	1) Easy to enter this way

Fortunately, the problems with Table 3.4 can be largely resolved by converting the data to the a tidy data format. This can be done with the pivot_long() command that we will learn about later in this chapter. Thus, we can enter the data in the format of Table 3.4 and later convert it to a tidy data format. After this conversion the data will be appear as in Table 3.5. For elapsed_time variable this data is now in the tidy data format. Each row corresponds to a single elapsed_time observed. Each column corresponds to a single variable. Somewhat problematically, however, sex is repeated three times for each person (i.e., over the three rows) - and this can be confusing. However, if the focus in on analyzing elapsed time this tidy data format makes sense. Importantly, there is an id column for each participant so R knows that this information

3.5 Data collection considerations

Data can be collected in a wide variety of way. Regardless of the methods of location researchers typically come to data in one of two way: 1) a research assistant enters the data into a spreadsheet type interface, or 2) the data is obtained as the output from computer software (e.g., Qualtrics, SurveyMonkey, Noldus, etc.).

Regardless of the approach it is critical to name your variables appropriately. For those using software, such as Qualtrics, this means setting up the software to use appropriate variable names PRIOR to data collection - so the exported file has desirable column names. For spreadsheet users, this means setting up the spreadsheet the data will be recorded in with column names that are amenable to the future analyses you want to conduct.

Although failure to take this thoughtful approach at the data collection stage can be overcome - it is only overcome with substantial manual effort. Therefore, as noted previously, we strongly encourage you to following the naming conventions we espouse here where you set up your data recording regime. Additionally, we encourage you to give careful thought in advance to the codes you will use to record missing data.

3.5.1 Naming conventions

To make your life easier down the road, it is critical you set up your spreadsheet or online survey such that is uses a naming convention prior to data collection. The naming conventions suggested here are adapted from the tidyverse style guide[6].

- Lowercase letters only
- If two word column names are necessary, only use the underscore ("_") character to separate words in the name.
- Avoid short decontextualized variable names like q1, q2, q3, etc.
- Do use moderate length column names. Aim to achieve a unique prefix for related columns so that those columns can be selected using the starts_with() command discussed in the previous chapter. Be sure to avoid short two or three letter prefixes for item names. Use moderate length unique item prefixes so that it will easy to select with those columns using start_with() such that you don't accidentally get additionally columns you don't want - that

[6]`https://style.tidyverse.org`

have a similar prefix. See the Likert-type item section below for additional details.

- If you have a column name that represents the levels of two repeated measures variables only use the underscore character to separate the levels of the different variables. See within-participant ANOVA section below for details.

3.5.2 Likert-type items

A Likert-type item is typically composed of a statement that participants are asked to agree or disagree with. For example, participants could be asked to indicate the extent to which they agree a number of statements such as "I like my job". They would then be presented with response scale such as: 1 - Strongly Disagree, 2 - Moderately Disagree, 3 - Neutral, 4, Moderately Agree, 5 - Strongly Agree. A common question is how should I enter the data?

- **Enter numeric responses not the labels**. You should enter the numeric value for each item response (e.g., 5) into your data - not the label (e.g., Strongly Agree). The labels associated with each value can be applied later in a script, if needed.
- **High numbers should be associated with more of the construct being measured.** When designing your survey or data collection tools, it is important that you set of the response options appropriately. If your scale measures job satisfaction, it is important that you collect data in a manner that ensures high numbers on the job satisfaction scale indicate high levels of job satisfaction. Therefore, assigning numbers makes sense using the 5-point scale: 1 - Strongly Disagree, 2 - Moderately Disagree, 3 - Neutral, 4, Moderately Agree, 5 - Strongly Agree. With this approach high response numbers indicate more job satisfaction. However, using the opposite scale would not make sense: 1 - Strongly Agree, 2 - Moderately Agree, 3 - Neutral, 4, Moderately Disagree, 5 - Strongly Disagree. With this opposite scale high numbers on a job satisfaction scale would indicate lower levels of job satisfaction - a very confusing situation. Avoid this situation, assign numbers so that higher numbers are associated with more of the construct being measured.
- **Use appropriate item names.** As described in the naming convention section, use moderate length names with different labels for each subscale.
- **Use moderate length column names unique to each subscale**. Imagine you have a survey with an 18-item commitment scale (Meyer et al., 1993) composed of three 6-item subscales: affective, normative, and continuance commitment. It would be a poor choice to prefix the labels of all 18 columns in your data with "commit" such that the names would be commit1, commit2, commit3, etc. The problem with this approach is that it fails to

distinguishing between the three subscales in naming convention; making it impossible to select the items for a single subscale using starts_with(). A better, but still poor choice for a naming convention would be use use a two letter prefix for the three scale such ac, nc, and cc. This would result in names for the columns like ac1, ac2, ac3, etc. This is an improvement because you could apparently (but likely not) select the columns using starts_with("ac"). The problem with these short names is that there could be many columns in data set that start with "ac" beside the affective commitment items. You might want to select the affective commitment items using starts_with("ac"); but you would get all the affective commitment item columns but also all the columns measuring other variables that also start with "ac". Therefore, it's a good idea to use a moderate length unique prefix for column names. For example, you might use prefixes like affectcom, normcom, and contincom for the three subscales. This would create column names like affectcom1, affectcom2, affectcom3, etc. These column prefixes are unlikely to be duplicated in other places in your column name conventions making it easy to select those columns using a command like starts_with("affectcom").

- **Indicate in the item name if the item is reversed keyed** Sometimes with Likert-type items, an item is reverse-keyed. For example, on a job satisfaction scale participants will typically respond to items that reflect job satisfaction using the scale: 1 - Strongly Disagree, 2 - Moderately Disagree, 3 - Neutral, 4, Moderately Agree, 5 - Strongly Agree. Higher numbers indicate more job satisfaction. Sometimes however, some items will use the same 1 to 5 response scale but be worded "I hate my job". Responding with a 5 to this item would indicate high job dissatisfaction not high job satisfaction - to the response will need to be flipped in your analysis script after data collection (i.e., 1 need to become a 5 and vice versa). To make it easier to do so, you should indicate if an item is reverse-keyed in the item name. The procedure for doing so is outlined in the next point.
- **Indicate in the item name the range for reverse key items.** If an item is reverse-keyed, the process for the flipping the scores depends upon the range of a scale. Although 5-point scale are common, any number of points are possible. The process for correcting a reverse key item depends upon: 1) the number of points on the scale, and 2) the range of the points on the scale. The reverse-key item correction process is different for an item that uses a 5-point scale ranging from 1 to 5 and from 0 to 4. Both are 5-point scale but your correction process will be different. Therefore, for reverse key items add as suffix at the end of each item name that indicates an item is reverse-keyed and the range of the item. For example, if the third job satisfaction item was reversed keyed on scale using a 1 to 5 response format you might name the item: jobsat3_rev15. The suffix "_rev15" indicates the item is reverse-keyed and the range of responses used on the item is 1 to 5.

Be sure to set up your survey with this naming convention when you collect your data.

- If you collect items over multiple time points use a prefix with a short code to indicate the time followed by and underscore. For example, if you had a multi-item self-esteem scale you might call the column for the firs time "t1_esteem1_rev15". This indicate that you have for time 1 (t1), the first self-esteem item (esteem1) and that item is reverse-keyed on a 1 to 5 scale.

3.5.3 ANOVA between

Avoid numerical representation of categorical variables. Don't use 1 or 2 to represent a variable like sex. Use male and female in your spreadsheet - likewise in your survey program. Similarly, for between participant variables like drug_condition don't use 1 or 2 use "drug" and "placebo" but the actually drug name would be even better than the word "drug."

3.5.4 ANOVA within

If you have a study that involves within-participant predictors naming conventions can become examples. When you have a single repeated measures predictor like occasion in the previous running example, it is often necessary to spread the level of that variable over multiple columns (e.g., march, may, july).

When you have multiple repeated measures predictor the situation is even more complicated. In this case, each column name needs to represent the levels of multiple repeated measures predictors at the time of data entry. For example, imagine you are a food researcher interested in taste ratings (the dependent variable) for various foods and contexts. You have food type (i.e., food_type) as a predictor with three levels (pizza, steak, burger). You have a second predictor temperature with two levels (hot, cold). All participants taste all foods at all temperatures. Thus, six columns are required to record taste rating for each participant: pizza_hot, pizza_cold, steak_hot, steak_cold, burger_hot, and burger_cold. Notice how each name contains one level of each predictor variable. The levels by the two predictor variables are separated by a single underscore. This should be the only underscore in the variable name because that underscore will be used by the computer when changing the data to the tidy format. If you had two underscores an name like "italian_pizza_hot" you would confuse the pivot_longer() command when it attempts to create a tidy version of the data. The computer would think there were three repeated levels variables instead of two. Thus, when dealing with repeated measures predictors, only use underscores to separate levels of predictor variables in column names.

3.6 Following the examples

Below we present example scripts transforming raw data to analytic data for various study designs (experimental and survey). These examples illustrate the value of using the naming conventions outlined previously. Don't just read the example - follow along with the projects by creating a separate script for each example. Resist the urge to cut and paste from this document - type the script yourself.

When first learning iPhone/Mac software development, I did so by taking a course at Big Nerd Ranch[7] - yes, that's a real place. They advised in their material (and now book) the following: "We have learned that "going through the motions" is much more important than it sounds. Many times we will ask you to start typing in code before you understand it. We realize that you may feel like a trained monkey typing in a bunch of code that you do not fully grasp. But the best way to learn coding is to find and fix your typos. Far from being a drag, this basic debugging is where you really learn the ins and outs of the code. That is why we encourage you to type in the code yourself. You could just download it, but copying and pasting is not programming. We want better for you and your skills.", p. xiv, (Keur and Hillegass, 2020). This is excellent advice for a beginning statistician or data scientist as well. And as an aside: if you want to learn iPhone programming you can't go wrong the Big Nerd Ranch guide!

As you work through this chapter, create your own new script for each example. In light of the above advice, avoid copying and pasting code - type it out; you will be the better for it.

Getting started:

The Class: R Studio in the Cloud Assignment

1. The data should be in the assignment project automatically. Just start the assignment.

For everyone in the class, that's it.

For those of you reading this work not in the class, see the two options below:

R Studio Cloud, custom project

1. Create a new Project using the web interface
2. Upload all the example data files in to the project. The data files

[7] `https://www.bignerdranch.com`

need are listed at the beginning of this chapter. The upload button can be found on the Files tab.

R Studio Computer, custom project

1. Create a folder on your computer for the example
2. Place all the example data files in that folder. The data files need are listed at the beginning of this chapter.
3. Use the menu item File > New Project… to start the project
4. On the window that appears select "Existing Directory"
5. On the next screen, press the "Browse" button and find/select the folder with your data
6. Press the Create Project Button

Regardless of whether your are working from the cloud or locally you should now have an R Studio project with your data files in it.

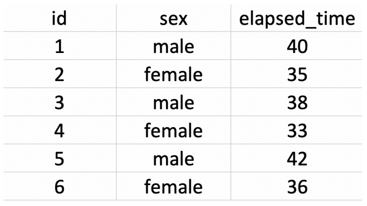
We anticipate many people will doubtless want to refer back to an encapsulated set of instructions for each design. There for the example for each design is written in a way that it stands alone. A consequence of this approach is that there is some redundancy in the code across example. We see this a strength - because readers will see the commonalities across differ types of designs.

As you make a script for each example:

- Recall the instruction from Chapter 1 about putting the date and your name in the script via comments.
- Recall the instruction from Chapter 1 about running library(tidyverse) before you type the rest of each script - this provides you with tidyverse autocomplete for the script.
- After you type each new block of code in an example, save your script.
- After you type each new block of code in an example do two additional things: 1) Session Restart R, 2) Run your script using Source with Echo.

3.7 Entering data into spreadsheets

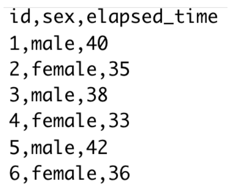
The first example uses a data file data_ex_between.csv that corresponds to a fictitious example where we recorded the run times for a number of male and female participants. How did we create this data file? We used a spreadsheet to enter the data, as illustrated in Figure 3.3. Programs like Microsoft Excel and Google Sheets are good options for entering data.



id	sex	elapsed_time
1	male	40
2	female	35
3	male	38
4	female	33
5	male	42
6	female	36

FIGURE 3.3: Spreadsheet entry of running data

The key to using these types of programs is to the save the data as a .csv file when you are done. CSV is short for Comma Separated Values. After entering the data in Figure 3.3 we saved it as data_ex_between.csv. There is no need to do so, but if you were to open this file in a text editor (such as TextEdit on a Mac or Notepad on Windows) you would see the information displayed in Figure 3.4. You can see there is one row per person and the columns are created by separating each values by a comma; hence, comma separated values.



```
id,sex,elapsed_time
1,male,40
2,female,35
3,male,38
4,female,33
5,male,42
6,female,36
```

FIGURE 3.4: Text view of CSV data

There are many ways to save data, but the CSV data is one of the better ones because it is a non-proprietary format. Some software, such as SPSS, uses a proprietary format (e.g., .sav for SPSS) this makes it challenging to access that data if you don't have that (often expensive) software. One of our goals as scientists is to make it easy for others to audit our work - that allows science to be self-correcting. Therefore, choose an open format for your data like .csv.

3.8 Experiment: Between

This section outlines a workflow appropriate for when you plan to conduct independent groups t-tests or a between-participants ANOVA.

To Begin:

- Use the Files tab to confirm you have the data: data_ex_between.csv
- Start a new script for this example. Don't forget to start the script name with "script_".

As noted previously, these data correspond to a design where the researcher is interested in comparing run times (elapsed_time) based on sex (male/female).

```
# Date: YYYY-MM-DD
# Name: your name here
# Example: Between-participant experiment

# Load data
library(tidyverse)

my_missing_value_codes <- c("-999", "", "NA")

raw_data_beween <- read_csv(file = "data_ex_between.csv",
                     na = my_missing_value_codes)
```

We load the initial data into a raw_data_between but immediately make a copy we will work with called analytic_data_between. It's good to keep a copy of the raw data for reference if you encounter problems.

```
analytic_data_between <- raw_data_beween
```

After loading the data we do initial cleaning to remove empty row/columns and ensure proper naming for columns:

```
library(janitor)

# Initial cleaning
analytic_data_between <- analytic_data_between %>%
  remove_empty("rows") %>%
  remove_empty("cols") %>%
  clean_names()
```

You can confirm the column names follow our naming convention with the glimpse() command.

```
glimpse(analytic_data_between)
```

```
## Rows: 6
## Columns: 3
## $ id           <dbl> 1, 2, 3, 4, 5, 6
## $ sex          <chr> "male", "female", "male", "female"...
## $ elapsed_time <dbl> 40, 35, 38, 33, 42, 36
```

3.8.1 Creating factors

Following initial cleaning, we identify categorical variables as factors. If you plan to conduct an ANOVA - it's critical that all predictor variables are converted to factors. Inspect the glimpse() output - if you followed our data entry naming conventions, categorical variables should be of the type character. We have one variable, sex, that is a categorical variable of type character (i.e., chr). The participant id column is categorical as well, but of type double (i.e., dbl) which is a numeric column.

```r
glimpse(analytic_data_between)
```

```
## Rows: 6
## Columns: 3
## $ id           <dbl> 1, 2, 3, 4, 5, 6
## $ sex          <chr> "male", "female", "male", "female"...
## $ elapsed_time <dbl> 40, 35, 38, 33, 42, 36
```

You can quickly convert all character columns to factors using the code below. In this case, this just converts the sex column to a factor. Because there is only one column (sex) being converted to a factor, we could have treated it the same was as the id column below. However, we use this code because of its broad applicability to many scripts.

```r
analytic_data_between <- analytic_data_between %>%
  mutate(across(.cols = where(is.character),
                .fns = as_factor))
```

The participant identification number in the id column is a numeric column, so it was not converted by the above code. The id column is converted to a factor with the code below.

```r
analytic_data_between <- analytic_data_between %>%
  mutate(id = as_factor(id))
```

You can ensure both the sex and id columns are now factors using the glimpse() command.

```
glimpse(analytic_data_between)
```

```
## Rows: 6
## Columns: 3
## $ id           <fct> 1, 2, 3, 4, 5, 6
## $ sex          <fct> male, female, male, female, male, ...
## $ elapsed_time <dbl> 40, 35, 38, 33, 42, 36
```

This example is so small it's clear you didn't miss converting any columns to factors. In general, however, at this point you should inspect the output of the glimpse() command and make sure you have converted all categorical variables to factors - especially those you will use as predictors.

3.8.2 Factor screening

Inspect the levels of each factor carefully. Make sure that there not any levels for a factor that are incorrect. For example, you wouldn't want to have the following levels for sex: male, mmale, female. Obviously, mmale is an incorrectly typed version of male. Scan all the factors in your data for erroneous factor levels. The code below displays the factor levels:

```
analytic_data_between %>%
  select(where(is.factor)) %>%
  summary()
```

```
##  id        sex
##  1:1   male  :3
##  2:1   female:3
##  3:1
##  4:1
##  5:1
##  6:1
```

The order of the levels influences how graphs are generated. In these data, the sex column has two levels: male and female in that order. The code below adjusts the order of the sex variable because we want the x-axis of a future graph to display columns in the left to right order: female, male.

```
analytic_data_between <- analytic_data_between %>%
  mutate(sex = fct_relevel(sex,
                           "female",
                           "male"))
```

You can see the new order of the factor levels with summary():

```
analytic_data_between %>%
  select(where(is.factor)) %>%
  summary()
```

```
##  id        sex
##  1:1   female:3
##  2:1   male  :3
##  3:1
##  4:1
##  5:1
##  6:1
```

3.8.3 Numeric screening

For numeric variables, you should search for impossible values. For example, in the context of this example you want to ensure non of the elapsed_times are impossible or so large they appear to be data entry errors.

One option for doing so is the summary command again. This time, however, we use "is.numeric" in the where() command.

```
analytic_data_between %>%
  select(where(is.numeric)) %>%
  summary()
```

```
##   elapsed_time
##  Min.   :33.0
##  1st Qu.:35.2
##  Median :37.0
##  Mean   :37.3
##  3rd Qu.:39.5
##  Max.   :42.0
```

Scan the min and max values to ensure there are not any impossible values. If necessary, got back to the original data source and fix these impossible value. Alternatively, you might need to change them to missing values (i.e., NA values).

In this example all the values are reasonable values. However, if we discovered an out of range value (or values) for elapsed time we could convert those values to missing values with the code below. This code changes (i.e., mutates) a value in the elapsed_time column to become NA (not available or missing) if that value is less than zero. If the value is greater than or equal to zero, it stays the same. Note that when using this command we have to be very specific in terms of specifying our missing value. It usually needs to be one of NA_real_ or NA_character_. For numeric columns use NA_real_ and for character columns use NA_character_.

```
analytic_data_between <- analytic_data_between  %>%
    mutate(elapsed_time = case_when(
      elapsed_time < 0 ~ NA_real_,
      elapsed_time >= 0 ~ elapsed_time))
```

Once you are done numeric screening, the data is ready for analysis.

3.9 Experiment: Within one-way

This section outlines a workflow appropriate for when you have a repeated measures design with a single repeated measures predictor. The data corresponds to a design where the researcher is interested in comparing run times (elapsed_time) across three different occasions (march/may/june).

To Begin:

- Use the Files tab to confirm you have the data: data_ex_within.csv
- Start a new script for this example. Don't forget to start the script name with "script_".

```
# Date: YYYY-MM-DD
# Name: your name here
# Example: Within-participant experiment

# Load data
library(tidyverse)

my_missing_value_codes <- c("-999", "", "NA")

raw_data_within <- read_csv(file = "data_ex_within.csv",
                     na = my_missing_value_codes)
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   sex = col_character(),
##   march = col_double(),
##   may = col_double(),
##   july = col_double()
## )
```

We load the initial data into a raw_data_within but immediately make a copy we will work with called analytic_data_within. It's good to keep a copy of the raw data for reference if you encounter problems.

```
analytic_data_within <- raw_data_within
```

After loading the data we do initial cleaning to remove empty row/columns and ensure proper naming for columns:

```
library(janitor)

# Initial cleaning
analytic_data_within <- analytic_data_within %>%
  remove_empty("rows") %>%
  remove_empty("cols") %>%
  clean_names()
```

You can confirm the column names following our naming convention with the glimpse command - and see the data type for each column.

```
glimpse(analytic_data_within)
```

```
## Rows: 6
## Columns: 5
## $ id    <dbl> 1, 2, 3, 4, 5, 6
## $ sex   <chr> "male", "female", "male", "female", "male...
## $ march <dbl> 40, 35, 38, 33, 42, 36
## $ may   <dbl> 37, 32, 35, 30, 39, 33
## $ july  <dbl> 35, 30, 33, 28, 37, 31
```

3.9.1 Creating factors

Following initial cleaning, we identify categorical variables as factors. If you plan to conduct an ANOVA - it's critical that all predictor variables are converted to factors. Inspect the glimpse() output - if you followed our data entry naming conventions, categorical variables should be of the type character.

```
glimpse(analytic_data_within)
```

```
## Rows: 6
## Columns: 5
## $ id    <dbl> 1, 2, 3, 4, 5, 6
## $ sex   <chr> "male", "female", "male", "female", "male...
## $ march <dbl> 40, 35, 38, 33, 42, 36
## $ may   <dbl> 37, 32, 35, 30, 39, 33
## $ july  <dbl> 35, 30, 33, 28, 37, 31
```

We have one variable, sex, that is a categorical variable of type character (i.e., chr). The participant id column is categorical as well, but of type double (i.e., dbl) which is a numeric column.

You can quickly convert all character columns to factors using the code below. In this case, this just converts the sex column to a factor. Because there is only one column (sex) being converted to a factor, we could have treated it the same was as the id column below. However, we use this code because of its broad applicability to many scripts.

```
analytic_data_within <- analytic_data_within %>%
  mutate(across(.cols = where(is.character),
                .fns = as_factor))
```

The participant identification number in the id column is a numeric column, so it was not converted by the above code. The id column is converted to a factor with the code below.

```
analytic_data_within <- analytic_data_within %>%
  mutate(id = as_factor(id))
```

You can ensure both the sex and id columns are now factors using the glimpse() command.

```
glimpse(analytic_data_within)
```

```
## Rows: 6
## Columns: 5
## $ id    <fct> 1, 2, 3, 4, 5, 6
## $ sex   <fct> male, female, male, female, male, female
## $ march <dbl> 40, 35, 38, 33, 42, 36
## $ may   <dbl> 37, 32, 35, 30, 39, 33
## $ july  <dbl> 35, 30, 33, 28, 37, 31
```

This example is so small it's clear you didn't miss converting any columns to factors. In general, however, at this point you should inspect the output of the glimpse() command and make sure you have converted all categorical variables to factors - especially those you will use as predictors.

3.9.2 Factor screening

Inspect the levels of each factor carefully. Make sure that there not any levels for a factor that are incorrect. For example, you wouldn't want to have the following levels for sex: male, mmale, female. Obviously, mmale is an incorrectly typed version of male. Scan all the factors in your data for erroneous factor levels. The code below displays the factor levels:

```
analytic_data_within %>%
  select(where(is.factor)) %>%
  summary()
```

```
##  id        sex
##  1:1   male  :3
##  2:1   female:3
##  3:1
##  4:1
##  5:1
##  6:1
```

The order of the levels influences how graphs are generated. In these data, the sex column has two levels: male and female in that order. The code below adjusts the order of the sex variable because we want the x-axis of a future graph to display columns in the left to right order: female, male.

```
analytic_data_within <- analytic_data_within %>%
  mutate(sex = fct_relevel(sex,
                           "female",
                           "male"))
```

You can see the new order of the factor levels with summary():

```
analytic_data_within %>%
  select(where(is.factor)) %>%
  summary()
```

```
##  id        sex
##  1:1   female:3
##  2:1   male  :3
##  3:1
##  4:1
##  5:1
##  6:1
```

3.9.3 Numeric screening

For numeric variables, it's important find and remove impossible values. For example, in the context of this example you want to ensure none of the elapsed_times are impossible (i.e., negative) or clearly data entry errors.

Because we have several numeric columns that we are screening, we use the skim() command from the skimr package. The skim() command quickly provides basic descriptive statistics. In the output for this command there are also several columns that begin with p: p0, p25, p50, p75, and p100 (p25 and p75 omitted in output due to space). These columns correspond to the 0th, 25th, 50th, 75th, and 100th percentiles, respectively. The minimum and maximum values for the data column are indicated under the p0 and p100 labels. The median is the 50th percentile (p50). The interquartile range is the range between p25 and p75.

```
library(skimr)

analytic_data_within %>%
  select(where(is.numeric)) %>%
  skim()
```

```
##   skim_variable n_missing  mean   sd p0 p50 p100
## 1         march         0 37.33 3.33 33  37   42
## 2           may         0 34.33 3.33 30  34   39
## 3          july         0 32.33 3.33 28  32   37
```

Scan the minimum and maximum values (p0 and p100) to ensure there are not any impossible values. If necessary, got back to the original data source and fix these impossible value. Alternatively, you might need to change them to missing values (i.e., NA values).

In this example all the values are reasonable values. However, if we discovered an out of range value (or values) for elapsed time we could convert those values to missing values with the code below. This code changes (i.e., mutates) a value in the march column to become NA (not available or missing) if that value is less than zero. If the value is greater than or equal to zero, it stays the same. Note that when using this command we have to be very specific in terms of specifying our missing value. It usually needs to be one of NA_real_ or NA_character_. For numeric columns use NA_real_ and for character columns use NA_character_.

```
analytic_data_within <- analytic_data_within  %>%
    mutate(march = case_when(
      march < 0 ~ NA_real_,
      march >= 0 ~ march))
```

3.9.4 Pivot to tidy data

The analytic data in it's current for does not conform to the the tidy data specification. Inspect the data with the print() command. Notice the there is not column for occasion (with levels march/may/july). Instead, there are three columns each of which represents a level of occasion. The levels of occasion are spread across three columns called march, may, and july. Each of these columns contains elapsed time for participants in that month.

```
print(analytic_data_within)
```

```
## # A tibble: 6 x 5
##   id    sex    march   may  july
##   <fct> <fct>  <dbl> <dbl> <dbl>
## 1 1     male      40    37    35
## 2 2     female    35    32    30
## 3 3     male      38    35    33
## 4 4     female    33    30    28
## 5 5     male      42    39    37
## 6 6     female    36    33    31
```

The pivot_longer() command below coverts our data to the tidy data format. In this command we specify the the columns march, may, and june are all levels of a single variable called occasion. We specify the columns involved with the cols argument. The code march:july after the cols argument selects the march column, the july column, and all the columns in between. Each column contains elapsed times at level of the variable occasion. The names_to argument is used to indicate that a new column called occasion should be created to hold the different months. The value_to argument is used to indicate that a new column called elapsed_time should created to hold all the values from the march, may, and july columns.

```
analytic_data_within_tidy <- analytic_data_within %>%
  pivot_longer(cols = march:july,
               names_to = "occasion",
               values_to = "elapsed_time"
  )
```

You can see the data in the new format below.

```
print(analytic_data_within_tidy)
```

```
## # A tibble: 18 x 4
##    id    sex    occasion elapsed_time
##    <fct> <fct>  <chr>           <dbl>
##  1 1     male   march              40
##  2 1     male   may                37
##  3 1     male   july               35
##  4 2     female march              35
##  5 2     female may                32
##  6 2     female july               30
##  7 3     male   march              38
##  8 3     male   may                35
##  9 3     male   july               33
## 10 4     female march              33
## 11 4     female may                30
## 12 4     female july               28
## 13 5     male   march              42
## 14 5     male   may                39
## 15 5     male   july               37
## 16 6     female march              36
## 17 6     female may                33
## 18 6     female july               31
```

But notice that the new column occasion is of the type character. We need it to be a factor. So use the code below to do so:

```
analytic_data_within_tidy <- analytic_data_within_tidy %>%
  mutate(occasion = as_factor(occasion))
```

You can confirm that occasion is now a factor with the glimpse() command. Once this is complete, you are done preparing your one-way within participant analytic data.

```
glimpse(analytic_data_within_tidy)
```

```
## Rows: 18
## Columns: 4
## $ id           <fct> 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4...
## $ sex          <fct> male, male, male, female, female, ...
## $ occasion     <fct> march, may, july, march, may, july...
## $ elapsed_time <dbl> 40, 37, 35, 35, 32, 30, 38, 35, 33...
```

You now have two data sets analytic_data_within and analytic_data_within_tidy. You can calculate descriptive statistics, correlations and general cross-sectional analyses using the analytic_data_within data set. If you want to conduct a repeated measures ANOVA you use the analytic_data_within_tidy data set. Both data sets are now ready for analysis.

3.10 Experiment: Within N-way

This section outlines a workflow appropriate for when you have a repeated measures design with multiple repeated measures predictors. The data corresponds to a design where the researcher is interested in assessing the taste of food as a function of food type (pizza/steak/burger) and temperature (hot/cold).

To Begin:

- Use the Files tab to confirm you have the data: data_food.csv
- Start a new script for this example. Don't forget to start the script name with "script_".

```
# Date: YYYY-MM-DD
# Name: your name here
# Example: 2-way within-participant experiment

# Load data
library(tidyverse)

my_missing_value_codes <- c("-999", "", "NA")

raw_data_within_nway <- read_csv(file = "data_food.csv",
                      na = my_missing_value_codes)
```

We load the initial data into a raw_data_within_nway but immediately make a copy we will work with called analytic_data_within_nway. It's good to keep a copy of the raw data for reference if you encounter problems.

```
analytic_data_within_nway <- raw_data_within_nway
```

After loading the data we do initial cleaning to remove empty row/columns and ensure proper naming for columns:

```
library(janitor)

# Initial cleaning
analytic_data_within_nway <- analytic_data_within_nway %>%
  remove_empty("rows") %>%
  remove_empty("cols") %>%
  clean_names()
```

You can confirm the column names following our naming convention with the glimpse command - and see the data type for each column.

```
glimpse(analytic_data_within_nway)
```

```
## Rows: 6
## Columns: 8
## $ id          <dbl> 1, 2, 3, 4, 5, 6
## $ sex         <dbl> 1, 2, 1, 2, 1, 2
## $ pizza_hot   <dbl> 7, 8, 7, 8, 7, 9
## $ pizza_cold  <dbl> 6, 7, 5, 7, 6, 7
## $ steak_hot   <dbl> 6, 6, 7, 7, 8, 7
## $ steak_cold  <dbl> 3, 3, 4, 5, 7, 8
## $ burger_hot  <dbl> 7, 8, 7, 8, 7, 8
## $ burger_cold <dbl> 4, 3, 3, 3, 2, 5
```

3.10.1 Creating factors

Following initial cleaning, we identify categorical variables as factors. If you plan to conduct an ANOVA - it's critical that all predictor variables are converted to factors. In this example, there are two categorical variables id and sex, but both are represented numerically. As revealed by the glimpse() output.

```
glimpse(analytic_data_within_nway)
```

```
## Rows: 6
## Columns: 8
## $ id          <dbl> 1, 2, 3, 4, 5, 6
## $ sex         <dbl> 1, 2, 1, 2, 1, 2
## $ pizza_hot   <dbl> 7, 8, 7, 8, 7, 9
## $ pizza_cold  <dbl> 6, 7, 5, 7, 6, 7
## $ steak_hot   <dbl> 6, 6, 7, 7, 8, 7
## $ steak_cold  <dbl> 3, 3, 4, 5, 7, 8
## $ burger_hot  <dbl> 7, 8, 7, 8, 7, 8
## $ burger_cold <dbl> 4, 3, 3, 3, 2, 5
```

We convert both the sex and id columns to factors with the mutate() command below:

```
analytic_data_within_nway <- analytic_data_within_nway %>%
  mutate(id = as_factor(id),
         sex = as_factor(sex))
```

The sex column is a factor but we have to tell the computer that 1 indicates male and 2 indicates female.

```
analytic_data_within_nway <- analytic_data_within_nway %>%
  mutate(sex = fct_recode(sex,
                          male = "1",
                          female = "2"))
```

You can ensure all of these columns are now factors using the glimpse() command.

```
glimpse(analytic_data_within_nway)
```

```
## Rows: 6
## Columns: 8
## $ id          <fct> 1, 2, 3, 4, 5, 6
## $ sex         <fct> male, female, male, female, male, f...
## $ pizza_hot   <dbl> 7, 8, 7, 8, 7, 9
## $ pizza_cold  <dbl> 6, 7, 5, 7, 6, 7
## $ steak_hot   <dbl> 6, 6, 7, 7, 8, 7
## $ steak_cold  <dbl> 3, 3, 4, 5, 7, 8
## $ burger_hot  <dbl> 7, 8, 7, 8, 7, 8
## $ burger_cold <dbl> 4, 3, 3, 3, 2, 5
```

Inspect the output of the glimpse() command and make sure you have converted all categorical variables to factors - especially those you will use as predictors. As noted in the previous examples, its common to have additional columns that are categorical predictors but appear in the glimpse() output as being of the type character. That is not the case in these data, but if it were the command below would turn them into factors:

```
analytic_data_within_nway <- analytic_data_within_nway %>%
  mutate(across(.cols = where(is.character),
                .fns = as_factor))
```

3.10.2 Factor screening

Inspect the levels of each factor carefully. Make sure that there not any levels for a factor that are incorrect. For example, you wouldn't want to have the following levels for sex: male, mmale, female. Obviously, mmale is an incorrectly typed version of male. Scan all the factors in your data for erroneous factor levels. The code below displays the factor levels:

```
analytic_data_within_nway %>%
  select(where(is.factor)) %>%
  summary()
```

```
##  id        sex
##  1:1   male  :3
##  2:1   female:3
##  3:1
##  4:1
##  5:1
##  6:1
```

The order of the levels influences how graphs are generated. In these data, the sex column has two levels: male and female in that order. The code below adjusts the order of the sex variable because we want the x-axis of a future graph to display columns in the left to right order: female, male.

```
analytic_data_within_nway <- analytic_data_within_nway %>%
  mutate(sex = fct_relevel(sex,
                           "female",
                           "male"))
```

You can see the new order of the factor levels with summary():

```
analytic_data_within_nway %>%
  select(where(is.factor)) %>%
  summary()
```

```
##  id        sex
##  1:1   female:3
##  2:1   male  :3
##  3:1
##  4:1
##  5:1
##  6:1
```

3.10.3 Numeric screening

For numeric variables, it's important find and remove impossible values. For example, in the context of this example you want to ensure none of the taste ratings the six columns (pizza_hot, pizza_cold, steak_hot, steak_cold, burger_hot, and burger_cold) are outside the range of the 1 to 10 rating scale.

Because we have several numeric columns that we are screening, we use the skim() command from the skimr package. The skim() command quickly provides basic descriptive statistics. In the output for this command there are also several columns that begin with p: p0, p25, p50, p75, and p100 (p25 and p75 omitted in output due to space). These columns correspond to the 0th, 25th, 50th, 75th, and 100th percentiles, respectively. The minimum and maximum values for the data column are indicated under the p0 and p100 labels. The median is the 50th percentile (p50). The interquartile range is the range between p25 and p75.

```
library(skimr)

analytic_data_within_nway %>%
  select(where(is.numeric)) %>%
  skim()
```

```
##   skim_variable n_missing mean   sd p0 p50 p100
## 1     pizza_hot         0 7.67 0.82  7 7.5    9
## 2    pizza_cold         0 6.33 0.82  5 6.5    7
## 3     steak_hot         0 6.83 0.75  6 7.0    8
## 4    steak_cold         0 5.00 2.10  3 4.5    8
## 5    burger_hot         0 7.50 0.55  7 7.5    8
## 6   burger_cold         0 3.33 1.03  2 3.0    5
```

Scan the minimum and maximum values (p0 and p100) to ensure there are not any impossible values. That is values outside the 1 to 10 range in this example. If necessary, got back to the original data source and fix these impossible value. Alternatively, you might need to change them to missing values (i.e., NA values).

In this example all the values are reasonable values. However, if we discovered an out of range value (or values) for elapsed time we could convert those values to missing values with the code below. This code changes (i.e., mutates) a value in the pizza_hot column to become NA (not available or missing) if that value is outside the 1 to 10 range of the rating scale. Note that when using this command we have to be very specific in terms of specifying our missing value. It usually needs to be one of NA_real_ or NA_character_. For numeric columns use NA_real_ and for character columns use NA_character_.

```
# Values lower than 1 are converted to missing values
analytic_data_within_nway <- analytic_data_within_nway  %>%
    mutate(pizza_hot = case_when(
      pizza_hot < 1  ~ NA_real_,
      pizza_hot >= 1 ~ pizza_hot))

# Values greater than 10 are converted to missing values
analytic_data_within_nway <- analytic_data_within_nway  %>%
    mutate(pizza_hot = case_when(
      pizza_hot > 10  ~ NA_real_,
      pizza_hot <= 10 ~ pizza_hot))
```

3.10.4 Pivot to tidy data

The analytic data in it's current for does not conform to the the tidy data specification. Inspect the data with the print() command. Notice there are not columns for temperature (with levels hot/cold) or food type (with levels pizza/steak/burger). Instead, there are six columns that are combinations of the levels of these variables (i.e., pizza_hot, pizza_cold, steak_hot, steak_cold, burger_hot, and burger_cold). Each of these columns contains taste ratings on a 1 to 10 point scale.

```
print(analytic_data_within)
```

```
## # A tibble: 6 x 5
##   id    sex    march   may  july
##   <fct> <fct>  <dbl> <dbl> <dbl>
## 1 1     male      40    37    35
## 2 2     female    35    32    30
## 3 3     male      38    35    33
## 4 4     female    33    30    28
## 5 5     male      42    39    37
## 6 6     female    36    33    31
```

We need to restructure the data into the tidy data format so that we have a food_type column and a temperature column to properly represent these predictors. As well, we need a column that contains all of the taste ratings that is clearly label taste. Doing all of these things will ensure we have one variable per column and one observation per so - consistent with the requirements of tidy data. The pivot_longer() command below coverts our data to the tidy data format.

In this command we specify the the columns march, may, and june are all levels of a single variable called occasion. We specify the columns involved with the cols argument. The code pizza_hot:burger_cold after the cols argument selects the pizza_hot column, the burger_cold column, and all the columns in between. Each column contains taste ratings at combination of the levels of the variables food_type and temperature. The names_to argument is used to indicate that two new columns should be created to represent food and temperature. Notice the order food then temperature. This is consistent with our naming convention; in the column name pizza_hot the food_type is specified before temperature. The value_to argument is used to indicate that a new column called taste should created to hold all the values from the pizza_hot, pizza_cold, steak_hot, steak_cold, burger_hot, and burger_cold columns.

```
analytic_data_nway_tidy <- analytic_data_within_nway %>%
  pivot_longer(cols = pizza_hot:burger_cold,
               names_to = c("food_type", "temperature"),
               names_sep = "_",
               values_to = "taste"
  )
```

You can see the data in the new format below.

```
print(analytic_data_nway_tidy)
```

```
## # A tibble: 36 x 5
##    id    sex    food_type temperature taste
##    <fct> <fct>  <chr>     <chr>       <dbl>
##  1 1     male   pizza     hot             7
##  2 1     male   pizza     cold            6
##  3 1     male   steak     hot             6
##  4 1     male   steak     cold            3
##  5 1     male   burger    hot             7
##  6 1     male   burger    cold            4
##  7 2     female pizza     hot             8
##  8 2     female pizza     cold            7
##  9 2     female steak     hot             6
## 10 2     female steak     cold            3
## # ... with 26 more rows
```

But notice that the new column occasion is of the type character. We need it to be a factor. So use the code below to do so:

```
analytic_data_nway_tidy <- analytic_data_nway_tidy %>%
  mutate(food = as_factor(food_type),
         temperature = as_factor(temperature))
```

You can confirm that occasion is now a factor with the glimpse() command. Once this is complete, you are done preparing your one-way within participant analytic data.

```
glimpse(analytic_data_nway_tidy)
```

```
## Rows: 36
## Columns: 6
## $ id          <fct> 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2,...
## $ sex         <fct> male, male, male, male, male, male,...
## $ food_type   <chr> "pizza", "pizza", "steak", "steak",...
## $ temperature <fct> hot, cold, hot, cold, hot, cold, ho...
## $ taste       <dbl> 7, 6, 6, 3, 7, 4, 8, 7, 6, 3, 8, 3,...
## $ food        <fct> pizza, pizza, steak, steak, burger,...
```

You now have two data sets analytic_data_within_nway and analytic_data_nway_tidy. You can calculate descriptive statistics, correlations and general cross-sectional analyses using the analytic_data_within_nway data set. If you want to conduct a repeated measures ANOVA you use the analytic_data_nway_tidy data set. Both data sets are now ready for analysis.

3.11 Surveys: Single Occassion

This section outlines a workflow appropriate for when you have cross-sectional single occasion survey data. The data corresponds to a design where the researcher has measured, age, sex, eye color, self-esteem, and job satisfaction. Two these, self-esteem and job satisfaction, were measured with multi-item scales with reverse-keyed items.

To Begin:

- Use the Files tab to confirm you have the data: data_item_scoring.csv

- Start a new script for this example. Don't forget to start the script name with "script_".

```
# Date: YYYY-MM-DD
# Name: your name here
# Example: Single occasion survey

# Load data
library(tidyverse)

my_missing_value_codes <- c("-999", "", "NA")

raw_data_survey <- read_csv(file = "data_item_scoring.csv",
                     na = my_missing_value_codes)
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   age = col_double(),
##   sex = col_character(),
##   eye_color = col_character(),
##   esteem1 = col_double(),
##   esteem2 = col_double(),
##   esteem3 = col_double(),
##   esteem4 = col_double(),
##   esteem5_rev15 = col_double(),
##   jobsat1 = col_double(),
##   jobsat2_rev15 = col_double(),
##   jobsat3 = col_double(),
##   jobsat4 = col_double(),
##   jobsat5 = col_double()
## )
```

We load the initial data into a raw_data_survey but immediately make a copy we will work with called analytic_data_survey. It's good to keep a copy of the raw data for reference if you encounter problems.

```
analytic_data_survey <- raw_data_survey
```

Remove empty row and columns from your data using the remove_empty_cols() and remove_empty_rows(), respectively. As well, clean the names of your columns to ensure they conform to tidyverse naming conventions.

```
library(janitor)

# Initial cleaning
analytic_data_survey <- analytic_data_survey %>%
  remove_empty("rows") %>%
  remove_empty("cols") %>%
  clean_names()
```

You can confirm the column names following our naming convention with the glimpse command - and see the data type for each column.

```
glimpse(analytic_data_survey)
```

```
## Rows: 300
## Columns: 14
## $ id           <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...
## $ age          <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 1...
## $ sex          <chr> "male", "female", "male", "female...
## $ eye_color    <chr> "blue", "brown", "hazel", "blue",...
## $ esteem1      <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, ...
## $ esteem2      <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, ...
## $ esteem3      <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4,...
## $ esteem4      <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA,...
## $ esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2, 3...
## $ jobsat1      <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, 3, ...
## $ jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2, 3,...
## $ jobsat3      <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ jobsat4      <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4, N...
## $ jobsat5      <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5, 4,...
```

3.11.1 Creating factors

Following initial cleaning, we identify categorical variables as factors. If you plan to conduct an ANOVA - it's critical that all predictor variables are converted to factors. Inspect the glimpse() output - if you followed our data entry naming conventions, categorical variables should be of the type character.

```
glimpse(analytic_data_survey)
```

```
## Rows: 300
## Columns: 14
## $ id           <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...
## $ age          <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 1...
## $ sex          <chr> "male", "female", "male", "female...
## $ eye_color    <chr> "blue", "brown", "hazel", "blue",...
## $ esteem1      <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, ...
## $ esteem2      <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, ...
## $ esteem3      <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4,...
## $ esteem4      <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA,...
## $ esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2, 3...
## $ jobsat1      <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, 3, ...
## $ jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2, 3,...
## $ jobsat3      <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ jobsat4      <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4, N...
## $ jobsat5      <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5, 4,...
```

We have two variables, sex and eye_color, that are categorical variable of type character (i.e., chr). The participant id column is categorical as well, but of type double (i.e., dbl) which is a numeric column. You can quickly convert all character columns to factors using the code below:

```
analytic_data_survey <- analytic_data_survey %>%
  mutate(across(.cols = where(is.character),
                .fns = as_factor))
```

The participant identification number in the id column is a numeric column, so we have handle that column on it's own.

```
analytic_data_survey <- analytic_data_survey %>%
  mutate(id = as_factor(id))
```

You can ensure all of these columns are now factors using the glimpse() command.

```
glimpse(analytic_data_survey)
```

```
## Rows: 300
## Columns: 14
## $ id           <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...
## $ age          <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 1...
## $ sex          <fct> male, female, male, female, male,...
## $ eye_color    <fct> blue, brown, hazel, blue, NA, haz...
## $ esteem1      <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, ...
## $ esteem2      <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, ...
## $ esteem3      <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4,...
## $ esteem4      <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA,...
## $ esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2, 3...
## $ jobsat1       <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, 3, ...
## $ jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2, 3,...
## $ jobsat3       <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ jobsat4       <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4, N...
## $ jobsat5       <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5, 4,...
```

Inspect the output of the glimpse() command and make sure you have converted all categorical variables to factors - especially those you will use as predictors.

Note: f you have factors like sex that have numeric data in the column (e.g, 1 and 2) instead of male/female you need to handle the situation differently. The preceding section, Experiment: Within N-way, illustrates how to handle this scenario.

3.11.2 Factor screening

Inspect the levels of each factor carefully. Make sure that there not any levels present that are incorrect. For example, you wouldn't want to have the following levels for sex: male, mmale, female. Obviously, mmale is an incorrectly typed version of male. Scan all the factor in your data for erroneous factor levels. The code below displays the factor levels:

```
analytic_data_survey %>%
  select(where(is.factor)) %>%
  summary()
```

```
##        id            sex      eye_color
##  1      :  1   male    :147   blue : 99
##  2      :  1   female  :149   brown: 98
##  3      :  1   intersex:  2   hazel:100
##  4      :  1   NA's    :  2   NA's :  3
##  5      :  1
##  6      :  1
##  (Other):294
```

Also inspect the output of the above summary() command paying attention to the order of the levels in the factors. The order influences how text output and graphs are generated. In these data, the sex column has two levels: male and female in that order. Below we adjust the order of the sex variable because we want the x-axis of a future graph to display columns in the left to right order: female, male.

```
analytic_data_survey <- analytic_data_survey %>%
  mutate(sex = fct_relevel(sex,
                           "intersex",
                           "female",
                           "male"))
```

For eye color, we want to future graph to be have the most common eye colors on the left so we reorder the factor levels:

```
analytic_data_survey <- analytic_data_survey %>%
  mutate(eye_color = fct_infreq(eye_color))
```

You can see the new order of the factor levels with summary():

```
analytic_data_survey %>%
  select(where(is.factor)) %>%
  summary()
```

```
##        id            sex      eye_color
##  1      :  1   intersex:  2   hazel:100
##  2      :  1   female  :149   blue : 99
##  3      :  1   male    :147   brown: 98
##  4      :  1   NA's    :  2   NA's :  3
##  5      :  1
##  6      :  1
##  (Other):294
```

3.11.3 Numeric screening

For numeric variables, it's important find and remove impossible values. For example, in the context of this example you want to ensure none of the elapsed_times are impossible (e.g., outside the 1 to 5 point rating scale for Likert items) or clearly data entry errors.

Because we have several numeric columns that we are screening, we use the skim() command from the skimr package. The skim() command quickly provides basic descriptive statistics. In the output for this command there are also several columns that begin with p: p0, p25, p50, p75, and p100 (p25 and p75 omitted in output due to space). These columns correspond to the 0th, 25th, 50th, 75th, and 100th percentiles, respectively. The minimum and maximum values for the data column are indicated under the p0 and p100 labels.

The median is the 50th percentile (p50). The interquartile range is the range between p25 and p75.

Start by examining the range of non-scale items. In this case it's only age. Examine the output to see if any of the age values are unreasonable. As noted, in the output p0 and p100 indicate the 0th percentile and the 100th percentile; that is the minimum and maximum values for the variable. Check to make sure none of the age values are unreasonably low or high. If they are, you may need to check the original data source or replace them with missing values.

```
library(skimr)
analytic_data_survey %>%
  select(age) %>%
  skim()
```

```
##   skim_variable n_missing  mean   sd p0 p50 p100
## 1           age         3 20.52 2.05 17  20   24
```

With respect to the multi-item scales, it makes sense to look at sets of items rather than all of the items at once. This is because sometime items from different scales use response ranges. For example, for one measure might use a response scale with a range from 1 to 5 whereas for another measure might use a response scale with a range from 1 to 7. This is undesirable from a psychometric point of view, as discussed previously, but it if happens it your data - look at the scale items separately to make it easy to see out of range values.

We begin by looking at the items in the first scale, self-esteem. Possible items responses for this scale range from 1 to 5, make sure all responses are in this range. If any values fall outside this range, you may need to check the original data source or replace them with missing values - as described previously.

```
analytic_data_survey %>%
  select(starts_with("esteem")) %>%
  skim()
```

```
##   skim_variable n_missing mean   sd p0 p50 p100
## 1       esteem1        24 3.39 0.54  3   3    5
## 2       esteem2        28 2.35 0.48  2   2    3
## 3       esteem3        31 3.96 0.37  3   4    5
## 4       esteem4        15 3.54 0.50  3   4    4
## 5 esteem5_rev15        35 2.22 0.47  1   2    3
```

Follow the same process for the job satisfaction items. Write that code on your own now.

Possible items responses for the job satisfaction scale range from 1 to 5, make sure all responses are in this range. If any values fall outside this range, you may need to check the original data source or replace them with missing values - as described previously.

```
analytic_data_survey %>%
  select(starts_with("jobsat")) %>%
  skim()
```

```
##   skim_variable n_missing mean   sd p0 p50 p100
## 1       jobsat1        25 3.34 0.51  3   3    5
## 2 jobsat2_rev15        27 1.51 0.61  1   1    3
## 3       jobsat3        28 2.84 0.37  2   3    3
## 4       jobsat4        35 4.29 0.70  3   4    5
## 5       jobsat5        24 4.57 0.61  3   5    5
```

3.11.4 Scale scores

Scale scores involve averaging, for each person, their score over several items to create an overall scale score. The first step in the creation of scales if correcting the values of any reverse-keyed items.

3.11.4.1 Reverse key items

The way you deal with reverse-keyed items depends on how you scored them. Imagine you had a 5-point scale. You could have scored the scale with the values 1, 2, 3, 4, and 5. Alternatively, you could have scored the scale with the values 0, 1, 2, 3, and 4. The mathematical approach you use to correcting reverse-keyed items depends upon whether the scale starts with 1 or 0.

In this example, we scored the data using the 1 to 5 system. So that is the approach illustrated here. See the extra information box for details on how to fixed reverse-keyed items where the scale begins with zero.

In this data file all the reverse-keyed were identified with the suffix "_rev15" in the column names. This suffix indicates the item was reverse-keyed and that the original scale used the response points 1 to 5. We can see those items with the glimpse() command below. Notice that there are two reverse-keyed items - each on difference scales.

```
analytic_data_survey %>%
  select(ends_with("_rev15")) %>%
  glimpse()
```

```
## Rows: 300
## Columns: 2
## $ esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2, 3...
## $ jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2, 3,...
```

To correct a reverse-keyed items where the lowest possible rating value is a 1 (i.e, 1 on a 1 to 5 scale), we simply subtract all the scores from a value one more than the highest possible rating. For example, if a 1 to 5 response scale was used we subtract each response from 6 to obtain the recoded value.

Original value	Math	Recoded value
1	6 - 1	5
2	6 - 2	4
3	6 - 3	3
4	6 - 4	2
5	6 - 5	1

The code below:

- selects each column that ends with ”_rev15” (i.e., both esteem and jobsat scales)
- subtracts each value in that column from 6
- renames the column by removing ”_rev15” from the name because the reverse coding is complete.

```
analytic_data_survey <- analytic_data_survey %>%
  mutate(6 - across(.cols = ends_with("_rev15")) ) %>%
  rename_with(.fn = str_remove,
              .cols = ends_with("_rev15"),
              pattern = "_rev15")
```

You can use the glimpse() command to see the result of your work. If you compare this to values from the previous glimpse() command you can see they have changed. Also notice the column names no longer indicate the items are reverse-keyed.

```
glimpse(analytic_data_survey)
```

```
## Rows: 300
## Columns: 14
## $ id       <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12...
## $ age      <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 17, N...
## $ sex      <fct> male, female, male, female, male, fem...
## $ eye_color <fct> blue, brown, hazel, blue, NA, hazel, ...
## $ esteem1   <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, 4, N...
## $ esteem2   <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, 2, N...
## $ esteem3   <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4, 4, ...
## $ esteem4   <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA, 4, ...
## $ esteem5   <dbl> 4, 4, 4, 4, 4, NA, NA, 4, 4, 4, 3, 4,...
## $ jobsat1   <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, 3, 4, 4...
## $ jobsat2   <dbl> 5, 5, 5, NA, 5, 5, 4, 5, 4, 4, 3, 5, ...
## $ jobsat3   <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
## $ jobsat4   <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4, NA, 5...
## $ jobsat5   <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5, 4, NA,...
```

- If your scale had used response options numbered 0 to 4 the math is different. For each item you would use subtract values from the highest possible point (i.e, 4) instead of one larger than the highest possible point.

Original value	Math	Recoded value
0	4 - 0	4
1	4 - 1	3
2	4 - 2	2
3	4 - 3	1
4	4 - 4	0

Thus the mutate command would instead be:

mutate(4 - across(.cols = ends_with("_rev15")))

3.11.4.2 Creating scores

The process we use for creating scale scores deletes item level data from analytic_data_survey. This is a desirable aspect of the process because it removes information we are no longer interested in from our analytic data. That said, before we create scale score, we create a backup on the item level data called analytic_data_survey_items. We will need to use this backup later to compute the reliability of the scales we are creating.

```
analytic_data_survey_items <- analytic_data_survey
```

We want to make a self_esteem scale and plan to select items using starts_with("esteem"). But prior to doing this we make sure the start_with() command only gives us the items we want - and not additional unwanted items. The output below confirms there are not problems associated with using starts_with("esteem").

```
analytic_data_survey %>%
  select(starts_with("esteem")) %>%
  glimpse()
```

```
## Rows: 300
## Columns: 5
## $ esteem1 <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, 4, NA,...
## $ esteem2 <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, 2, NA,...
## $ esteem3 <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4, 4, 4,...
## $ esteem4 <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA, 4, 3,...
## $ esteem5 <dbl> 4, 4, 4, 4, 4, NA, NA, 4, 4, 4, 3, 4, 4...
```

Likewise, we want to make a job_sat scale and plan to select items using starts_with("jobsat"). The code and output below using starts_with("jobsat") only returns the items we are interested in.

```
analytic_data_survey %>%
  select(starts_with("jobsat")) %>%
  glimpse()
```

```
## Rows: 300
## Columns: 5
## $ jobsat1 <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, 3, 4, 4, ...
## $ jobsat2 <dbl> 5, 5, 5, NA, 5, 5, 4, 5, 4, 4, 3, 5, 3,...
## $ jobsat3 <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ jobsat4 <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4, NA, 5, ...
## $ jobsat5 <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5, 4, NA, 4...
```

We calculate the scale scores using the rowwise() command. The mean() command provides the mean of columns by default - not people. We use the rowwise() command in the code below to make the mean() command work across columns (within participants) rather than within columns. The mutate command calculates the scale score for each person. The c_across() command combined with the starts_with() command ensures the items we want averaged together are the items that are averaged together. Notice there is a separate mutate line for each scale. The ungroup() command turns off the rowwise() command. We end the code block by removing the item-level data from the data set.

Important: Take note of how we names the scale variables (e.g., self_esteem, job_sat) use a slightly different convention than our items. That is, these scale labels were picked so that they would *not* be selected by a starts_with("esteem") or starts_with("jobsat"). Why? Because we later use those commands to remove the item-level data. We would want the command designed to remove the item-level data to also remove the scale we just calculated! This example illustrates how carefully you need to think about your naming conventions.

```
analytic_data_survey <- analytic_data_survey %>%
  rowwise() %>%
  mutate(self_esteem = mean(c_across(starts_with("esteem")),
                            na.rm = TRUE)) %>%
  mutate(job_sat = mean(c_across(starts_with("jobsat")),
                            na.rm = TRUE)) %>%
  ungroup() %>%
  select(-starts_with("esteem")) %>%
  select(-starts_with("jobsat"))
```

We can see our data now has the self_esteem column, a job_sat column, and that all of the item level data has been removed.

```
glimpse(analytic_data_survey)
```

```
## Rows: 300
## Columns: 6
## $ id          <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
## $ age         <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 17,...
## $ sex         <fct> male, female, male, female, male, f...
## $ eye_color   <fct> blue, brown, hazel, blue, NA, hazel...
## $ self_esteem <dbl> 3.200, 3.800, 3.800, 3.000, 3.400, ...
## $ job_sat     <dbl> 4.00, 5.00, 4.40, 3.50, 4.00, 3.80,...
```

You now have two data sets analytic_data_survey and analytic_data_survey_items. You can calculate descriptive statistics, correlations and most analyses using the analytic_data_survey. To obtain the reliability of the scales you just created though you will need to use the analytic_data_survey_items. Both sets of data are ready for analysis.

3.12 Surveys: Multiple Occasions

This section outlines a workflow appropriate for when you have multiple occasion survey data. The data corresponds to a design where the researcher has measured, age, sex, eye color, self-esteem, and job satisfaction at each of two times points. Self-esteem and job satisfaction were measured with multi-item scales with reverse-keyed items.

To Begin:

- Use the Files tab to confirm you have the data: data_item_time.csv
- Start a new script for this example. Don't forget to start the script name with "script_".

```
# Date: YYYY-MM-DD
# Name: your name here
# Example: Multiple occasion survey

# Load data
library(tidyverse)

my_missing_value_codes <- c("-999", "", "NA")

raw_data_occasions <- read_csv(file = "data_item_time.csv",
                               na = my_missing_value_codes)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   sex = col_character(),
##   eye_color = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

We load the initial data into a raw_data_occasions but immediately make a copy we will work with called analytic_data_occasions. It's good to keep a copy of the raw data for reference if you encounter problems.

```
analytic_data_occasions <- raw_data_occasions
```

Remove empty row and columns from your data using the remove_empty_cols() and remove_empty_rows(), respectively. As well, clean the names of your columns to ensure they conform to tidyverse naming conventions.

```
library(janitor)

# Initial cleaning
analytic_data_occasions <- analytic_data_occasions %>%
  remove_empty("rows") %>%
  remove_empty("cols") %>%
  clean_names()
```

You can confirm the column names following our naming convention with the glimpse command - and see the data type for each column.

```
glimpse(analytic_data_occasions)
```

```
## Rows: 300
## Columns: 24
## $ id              <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,...
## $ age             <dbl> 23, 22, 18, 23, 22, 17, 23, 22...
## $ sex             <chr> "male", "female", "male", "fem...
## $ eye_color       <chr> "blue", "brown", "hazel", "blu...
## $ t1_esteem1      <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, ...
## $ t1_esteem2      <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, ...
## $ t1_esteem3      <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3,...
## $ t1_esteem4      <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, ...
## $ t1_esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2...
## $ t1_jobsat1      <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, ...
## $ t1_jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2,...
## $ t1_jobsat3      <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ t1_jobsat4      <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4...
## $ t1_jobsat5      <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5,...
## $ t2_esteem1      <dbl> 4, 5, 5, 4, NA, 4, 4, 5, 5, 5,...
## $ t2_esteem2      <dbl> 3, 4, 4, 3, 3, 4, 3, 4, 4, 4, ...
## $ t2_esteem3      <dbl> 5, 5, 5, 4, 5, 5, 3, 5, 5, 4, ...
## $ t2_esteem4      <dbl> 4, 5, 5, 4, 5, 5, 5, 5, 4, 5, ...
## $ t2_esteem5_rev15 <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
## $ t2_jobsat1      <dbl> 4, 6, 5, 4, 4, 4, 4, 6, 4, NA,...
## $ t2_jobsat2_rev15 <dbl> 2, 2, 2, 3, 2, 2, 3, 2, 3, 3, ...
## $ t2_jobsat3      <dbl> 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, ...
## $ t2_jobsat4      <dbl> 3, 6, 6, 5, 5, 5, 5, 6, 3, 5, ...
## $ t2_jobsat5      <dbl> 6, 3, 6, 5, NA, 5, 5, 6, 6, 6,...
```

3.12.1 Creating factors

Following initial cleaning, we identify categorical variables as factors. If you plan to conduct an ANOVA - it's critical that all predictor variables are converted to factors. Inspect the glimpse() output - if you followed our data entry naming conventions, categorical variables should be of the type character.

```
glimpse(analytic_data_occasions)
```

```
## Rows: 300
## Columns: 24
## $ id              <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,...
## $ age             <dbl> 23, 22, 18, 23, 22, 17, 23, 22...
## $ sex             <chr> "male", "female", "male", "fem...
## $ eye_color       <chr> "blue", "brown", "hazel", "blu...
## $ t1_esteem1      <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, ...
## $ t1_esteem2      <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, ...
## $ t1_esteem3      <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3,...
## $ t1_esteem4      <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, ...
## $ t1_esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2...
## $ t1_jobsat1      <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, ...
## $ t1_jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2,...
## $ t1_jobsat3      <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ t1_jobsat4      <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4...
## $ t1_jobsat5      <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5,...
## $ t2_esteem1      <dbl> 4, 5, 5, 4, NA, 4, 4, 5, 5, 5,...
## $ t2_esteem2      <dbl> 3, 4, 4, 3, 3, 4, 3, 4, 4, 4, ...
## $ t2_esteem3      <dbl> 5, 5, 5, 4, 5, 5, 3, 5, 5, 4, ...
## $ t2_esteem4      <dbl> 4, 5, 5, 4, 5, 5, 5, 5, 4, 5, ...
## $ t2_esteem5_rev15 <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
## $ t2_jobsat1      <dbl> 4, 6, 5, 4, 4, 4, 4, 6, 4, NA,...
## $ t2_jobsat2_rev15 <dbl> 2, 2, 2, 3, 2, 2, 3, 2, 3, 3, ...
## $ t2_jobsat3      <dbl> 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, ...
## $ t2_jobsat4      <dbl> 3, 6, 6, 5, 5, 5, 5, 6, 3, 5, ...
## $ t2_jobsat5      <dbl> 6, 3, 6, 5, NA, 5, 5, 6, 6, 6,...
```

We have two variables, sex and eye_color, that are categorical variable of type character (i.e., chr). The participant id column is categorical as well, but of type double (i.e., dbl) which is a numeric column. You can quickly convert all character columns to factors using the code below:

```
analytic_data_occasions <- analytic_data_occasions %>%
  mutate(across(.cols = where(is.character),
              .fns = as_factor))
```

The participant identification number in the id column is a numeric column, so we have handle that column on it's own.

```
analytic_data_occasions <- analytic_data_occasions %>%
  mutate(id = as_factor(id))
```

You can ensure all of these columns are now factors using the glimpse() command.

```
glimpse(analytic_data_occasions)
```

```
## Rows: 300
## Columns: 24
## $ id                <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,...
## $ age               <dbl> 23, 22, 18, 23, 22, 17, 23, 22...
## $ sex               <fct> male, female, male, female, ma...
## $ eye_color         <fct> blue, brown, hazel, blue, NA, ...
## $ t1_esteem1        <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, ...
## $ t1_esteem2        <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, ...
## $ t1_esteem3        <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3,...
## $ t1_esteem4        <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, ...
## $ t1_esteem5_rev15 <dbl> 2, 2, 2, 2, 2, NA, NA, 2, 2, 2...
## $ t1_jobsat1        <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, ...
## $ t1_jobsat2_rev15 <dbl> 1, 1, 1, NA, 1, 1, 2, 1, 2, 2,...
## $ t1_jobsat3        <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ t1_jobsat4        <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4...
## $ t1_jobsat5        <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5,...
## $ t2_esteem1        <dbl> 4, 5, 5, 4, NA, 4, 4, 5, 5, 5,...
## $ t2_esteem2        <dbl> 3, 4, 4, 3, 3, 4, 3, 4, 4, 4, ...
## $ t2_esteem3        <dbl> 5, 5, 5, 4, 5, 5, 3, 5, 5, 4, ...
## $ t2_esteem4        <dbl> 4, 5, 5, 4, 5, 5, 5, 5, 4, 5, ...
## $ t2_esteem5_rev15 <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
## $ t2_jobsat1        <dbl> 4, 6, 5, 4, 4, 4, 4, 6, 4, NA,...
## $ t2_jobsat2_rev15 <dbl> 2, 2, 2, 3, 2, 2, 3, 2, 3, 3, ...
## $ t2_jobsat3        <dbl> 4, 3, 4, 4, 4, 4, 4, 4, 4, 4, ...
## $ t2_jobsat4        <dbl> 3, 6, 6, 5, 5, 5, 5, 6, 3, 5, ...
## $ t2_jobsat5        <dbl> 6, 3, 6, 5, NA, 5, 5, 6, 6, 6,...
```

Inspect the output of the glimpse() command and make sure you have converted all categorical variables to factors - especially those you will use as predictors.

Note: f you have factors like sex that have numeric data in the column (e.g, 1 and 2) instead of male/female you need to handle the situation differently.

The preceding section, Experiment: Within N-way, illustrates how to handle this scenario.

3.12.2 Factor screening

Inspect the levels of each factor carefully. Make sure that there not any levels present that are incorrect. For example, you wouldn't want to have the following levels for sex: male, mmale, female. Obviously, mmale is an incorrectly typed version of male. Scan all the factor in your data for erroneous factor levels. The code below displays the factor levels:

```
analytic_data_occasions %>%
  select(where(is.factor)) %>%
  summary()
```

```
##        id             sex       eye_color
##  1      :  1   male    :147   blue : 99
##  2      :  1   female  :149   brown: 98
##  3      :  1   intersex:  2   hazel:100
##  4      :  1   NA's    :  2   NA's :  3
##  5      :  1
##  6      :  1
##  (Other):294
```

Also inspect the output of the above summary() command paying attention to the order of the levels in the factors. The order influences how text output and graphs are generated. In these data, the sex column has two levels: male and female in that order. Below we adjust the order of the sex variable because we want the x-axis of a future graph to display columns in the left to right order: female, male.

```
analytic_data_occasions <- analytic_data_occasions %>%
  mutate(sex = fct_relevel(sex,
                           "intersex",
                           "female",
                           "male"))
```

For eye color, we want to future graph to be have the most common eye colors on the left so we reorder the factor levels:

```
analytic_data_occasions <- analytic_data_occasions %>%
  mutate(eye_color = fct_infreq(eye_color))
```

You can see the new order of the factor levels with summary():

```
analytic_data_occasions %>%
  select(where(is.factor)) %>%
  summary()
```

```
##        id             sex        eye_color
##  1      :  1   intersex:  2   hazel:100
##  2      :  1   female  :149   blue : 99
##  3      :  1   male    :147   brown: 98
##  4      :  1   NA's    :  2   NA's :  3
##  5      :  1
##  6      :  1
##  (Other):294
```

3.12.3 Numeric screening

For numeric variables, it's important find and remove impossible values. For example, in the context of this example you want to ensure none of the elapsed_times are impossible (e.g., outside the 1 to 5 point rating scale for Likert items) or clearly data entry errors.

Because we have several numeric columns that we are screening, we use the skim() command from the skimr package. The skim() command quickly provides basic descriptive statistics. In the output for this command there are also several columns that begin with p: p0, p25, p50, p75, and p100 (p25 and p75 omitted in output due to space). These columns correspond to the 0th, 25th, 50th, 75th, and 100th percentiles, respectively. The minimum and maximum values for the data column are indicated under the p0 and p100 labels. The median is the 50th percentile (p50). The interquartile range is the range between p25 and p75.

Start by examining the range of non-scale items. In this case it's only age. Examine the output to see if any of the age values are unreasonable. As noted, in the output p0 and p100 indicate the 0th percentile and the 100th percentile; that is the minimum and maximum values for the variable. Check to make sure none of the age values are unreasonably low or high. If they are, you may need to check the original data source or replace them with missing values.

```
library(skimr)
analytic_data_occasions %>%
  select(age) %>%
  skim()
```

```
##   skim_variable n_missing  mean   sd p0 p50 p100
## 1           age         3 20.52 2.05 17  20   24
```

With respect to the multi-item scales, it makes sense to look at sets of items rather than all of the items at once. This is because sometime items from different scales use response ranges. For example, for one measure might use a response scale with a range from 1 to 5 whereas for another measure might use a response scale with a range from 1 to 7. This is undesirable from a psychometric point of view, as discussed previously, but it if happens it your data - look at the scale items separately to make it easy to see out of range values.

We begin by looking at the items in the first scale, self-esteem. Possible items responses for this scale range from 1 to 5, make sure all responses are in this range. If any values fall outside this range, you may need to check the original data source or replace them with missing values - as described previously.

Because we want to select the self-esteem items from both time 1 and time 2 we cannot use the starts_with() command. Instead we use the contains() command in the code below.

```
analytic_data_occasions %>%
  select(contains("esteem")) %>%
  skim()
```

```
##       skim_variable n_missing mean   sd p0 p50 p100
## 1        t1_esteem1        24 3.39 0.54  3   3    5
## 2        t1_esteem2        28 2.35 0.48  2   2    3
## 3        t1_esteem3        31 3.96 0.37  3   4    5
## 4        t1_esteem4        15 3.54 0.50  3   4    4
## 5  t1_esteem5_rev15        35 2.22 0.47  1   2    3
## 6        t2_esteem1         5 4.27 0.64  3   4    6
## 7        t2_esteem2         5 3.33 0.47  3   3    4
## 8        t2_esteem3         6 4.77 0.69  3   5    6
## 9        t2_esteem4         3 4.46 0.59  3   5    5
## 10 t2_esteem5_rev15         4 3.19 0.45  2   3    4
```

Follow the same process for the job satisfaction items. Possible items responses for the job satisfaction scale range from 1 to 5, make sure all responses are in this range. If any values fall outside this range, you may need to check the original data source or replace them with missing values - as described previously.

```
analytic_data_occasions %>%
  select(contains("jobsat")) %>%
  skim()
```

```
##       skim_variable n_missing mean   sd p0 p50 p100
## 1        t1_jobsat1        25 3.34 0.51  3   3    5
## 2  t1_jobsat2_rev15        27 1.51 0.61  1   1    3
## 3        t1_jobsat3        28 2.84 0.37  2   3    3
## 4        t1_jobsat4        35 4.29 0.70  3   4    5
## 5        t1_jobsat5        24 4.57 0.61  3   5    5
## 6        t2_jobsat1         2 4.23 0.62  3   4    6
## 7  t2_jobsat2_rev15         3 2.54 0.59  2   2    4
## 8        t2_jobsat3         5 3.76 0.43  3   4    4
## 9        t2_jobsat4         3 5.03 0.99  3   5    6
## 10       t2_jobsat5         3 5.36 0.92  3   6    6
```

3.12.4 Scale scores

Scale scores involve averaging, for each person, their score over several items to create an overall scale score. The first step in the creation of scales if correcting the values of any reverse-keyed items.

3.12.4.1 Reverse key items

The way you deal with reverse-keyed items depends on how you scored them. Imagine you had a 5-point scale. You could have scored the scale with the values 1, 2, 3, 4, and 5. Alternatively, you could have scored the scale with the values 0, 1, 2, 3, and 4. The mathematical approach you use to correcting reverse-keyed items depends upon whether the scale starts with 1 or 0.

In this example, we scored the data using the 1 to 5 system. So use the correct approach for scales that being with 1. The preceding section, Surveys: Single occasion, describes how the math differs when the response scale starts with 0. I encourage you to read that section before going further if you have not done so already.

In this data file all the reverse-keyed were identified with the suffix "_rev15" in the column names. This suffix indicates the item was reverse-keyed and that the original scale used the response points 1 to 5. We can see those items with the glimpse() command below. Notice that there are two reverse-keyed items - each on difference scales.

```
analytic_data_survey %>%
  select(ends_with("_rev15")) %>%
  glimpse()
```

```
## Rows: 300
## Columns: 0
```

To correct a reverse-keyed items where the lowest possible rating value is a 1 (i.e, 1 on a 1 to 5 scale), we simply subtract all the scores from a value one more than the highest possible rating (i.e., 6).

The code below:

- selects each column that ends with "_rev15" (i.e., both esteem and jobsat scales)
- subtracts each value in that column from 6
- renames the column by removing "_rev15" from the name because the reverse coding is complete.

```
analytic_data_occasions <- analytic_data_occasions %>%
  mutate(6 - across(.cols = ends_with("_rev15")) ) %>%
  rename_with(.fn = str_remove,
              .cols = ends_with("_rev15"),
              pattern = "_rev15")
```

You can use the glimpse() command to see the result of your work. If you compare this to values from the previous glimpse() command you can see they have changed. Also notice the column names no longer indicate the items are reverse-keyed.

3.12.4.2 Creating scores

The process we use for creating scale scores deletes item level data from analytic_data_survey. This is a desirable aspect of the process because it removes information we are no longer interested in from our analytic data. That said, before we create scale score, we create a backup on the item level data called analytic_data_survey_items. We will need to use this backup later to compute the reliability of the scales we are creating.

```
analytic_data_occasions_items <- analytic_data_occasions
```

We want to make a self_esteem scale and plan to select items using starts_with("t1_esteem"). But prior to doing this we make sure the start_with() command only gives us the items we want - and not additional unwanted items. The output below confirms there are not problems associated with using starts_with("t1_esteem").

```
analytic_data_occasions %>%
  select(starts_with("t1_esteem")) %>%
  glimpse()
```

```
## Rows: 300
## Columns: 5
## $ t1_esteem1 <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, 4, ...
## $ t1_esteem2 <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, 2, ...
## $ t1_esteem3 <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4, 4,...
## $ t1_esteem4 <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA, 4,...
## $ t1_esteem5 <dbl> 4, 4, 4, 4, 4, NA, NA, 4, 4, 4, 3, 4...
```

Repeat this set of commands using t2_esteem, t1_jobsat, and t2_jobsat. Make sure that those start_with() terms select only the relevant items and not others.

We calculate the scale scores using the rowwise() command. The mean() command provides the mean of columns by default - not people. We use the rowwise() command in the code below to make the mean() command work across columns (within participants) rather than within columns. The mutate command calculates the scale score for each person. The c_across() command combined with the starts_with() command ensures the items we want averaged together are the items that are averaged together. Notice there is a separate mutate line for each scale. The ungroup() command turns off the rowwise() command. We end the code block by removing the item-level data from the data set.

Important: Take note of how we names the scale variables (e.g., esteem_t1, jobsat_t1) use a slightly different convention than our items. That is, these scale labels were picked so that they would *not* be selected by a starts_with("t1_esteem") or starts_with("t1_jobsat"). Why? Because we later use those commands to remove the item-level data. We would want the command designed to remove the item-level data to also remove the scale we just calculated! This example illustrates how carefully you need to think about your naming conventions.

```r
analytic_data_occasions <- analytic_data_occasions %>%
  rowwise() %>%
  mutate(esteem_t1 = mean(c_across(starts_with("t1_esteem")),
                             na.rm = TRUE)) %>%
  mutate(esteem_t2 = mean(c_across(starts_with("t2_esteem")),
                             na.rm = TRUE)) %>%
  mutate(jobsat_t1 = mean(c_across(starts_with("t1_jobsat")),
                             na.rm = TRUE)) %>%
  mutate(jobsat_t2 = mean(c_across(starts_with("t2_jobsat")),
                             na.rm = TRUE)) %>%
  ungroup() %>%
  select(-starts_with("t1_esteem")) %>%
  select(-starts_with("t2_esteem")) %>%
  select(-starts_with("t1_jobsat")) %>%
  select(-starts_with("t2_jobsat"))
```

We can see our data now has the columns t1_esteem, t2_esteem, t1_jobsat, and t2_jobsat. As well, we can see that all of the item level data has been removed from the data set.

```
glimpse(analytic_data_occasions)
```

```
## Rows: 300
## Columns: 8
## $ id        <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12...
## $ age       <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 17, N...
## $ sex       <fct> male, female, male, female, male, fem...
## $ eye_color <fct> blue, brown, hazel, blue, NA, hazel, ...
## $ esteem_t1 <dbl> 3.200, 3.800, 3.800, 3.000, 3.400, 3....
## $ esteem_t2 <dbl> 3.8, 4.4, 4.4, 3.6, 4.0, 4.2, 3.6, 4....
## $ jobsat_t1 <dbl> 4.00, 5.00, 4.40, 3.50, 4.00, 3.80, 3...
## $ jobsat_t2 <dbl> 4.20, 4.40, 5.00, 4.20, 4.25, 4.40, 4...
```

3.12.5 Pivot to tidy data

The analytic data in its current for does not conform to the the tidy data specification. Inspect the data with the print() command. Notice that a single column is not used to represent esteem, jobsat, or time. Rather, there are four columns that are a mix of these variables. The consequence of this is that there are two esteem ratings/observations on each row and two jobsat ratings/observations on each row. Tidy data is structured so that each variable is represented in a single column and each observation has its own row.

```
print(analytic_data_occasions)
```

```
## # A tibble: 300 x 8
##    id      age sex   eye_color esteem_t1 esteem_t2 jobsat_t1
##    <fct> <dbl> <fct> <fct>         <dbl>     <dbl>     <dbl>
##  1 1        23 male  blue            3.2       3.8      4
##  2 2        22 fema~ brown           3.8       4.4      5
##  3 3        18 male  hazel           3.8       4.4      4.4
##  4 4        23 fema~ blue            3         3.6      3.5
##  5 5        22 male  <NA>            3.4       4        4
##  6 6        17 fema~ hazel           3.5       4.2      3.8
##  7 7        23 male  blue            3         3.6      3.6
##  8 8        22 fema~ brown           3.8       4.4      4.6
##  9 9        17 male  hazel           3.6       4.2      3.75
## 10 10       NA fema~ blue            3.6       4.2      3.8
## # ... with 290 more rows, and 1 more variable:
## #   jobsat_t2 <dbl>
```

The pivot_longer() command below coverts our data to the tidy data format. In this command we specify the the columns with data by using esteem_t1:jobsat_t2, this selects these two columns and all of the columns between them. Each of these columns represents a dependent variable at a particular time in the format "variable_time" (e.g., esteem_t1). The code names_to = c(".value", "time") explains this format to R. It indicates first part of the column name (e.g., esteem) contains the name of the variable (expressed in the code as ".value"). It also indicates that the second part of the column name represents time. The line names_sep = "_" tells the R that the underscore character is used to separate the first part of the name from the second part of the name. When this code is executed it creates a tidy version of data set stored in analytic_survey_tidy.

```
analytic_occasion_tidy <- analytic_data_occasions %>%
  pivot_longer(esteem_t1:jobsat_t2,
               names_to = c(".value", "time"),
               names_sep = "_")
```

You can see the new data with the print() command. Notice that each participant has multiple rows associated with them.

```
print(analytic_occasion_tidy)
```

```
## # A tibble: 600 x 7
##    id      age sex    eye_color time  esteem jobsat
##    <fct> <dbl> <fct>  <fct>     <chr>  <dbl>  <dbl>
##  1 1        23 male   blue      t1       3.2   4
##  2 1        23 male   blue      t2       3.8   4.2
##  3 2        22 female brown     t1       3.8   5
##  4 2        22 female brown     t2       4.4   4.4
##  5 3        18 male   hazel     t1       3.8   4.4
##  6 3        18 male   hazel     t2       4.4   5
##  7 4        23 female blue      t1       3     3.5
##  8 4        23 female blue      t2       3.6   4.2
##  9 5        22 male   <NA>      t1       3.4   4
## 10 5        22 male   <NA>      t2       4     4.25
## # ... with 590 more rows
```

You now have three data sets The data analytic_occasion_tidy is appropriate for conducting a repeated measures ANOVA or more complicated analyses. The data analytic_data_occasions is appropriate for calculating descriptive statistics and correlations. The data analytic_occasions_items is appropriate for calculating the reliability of the scales you constructed. These data are ready for analysis.

3.13 Basic descriptive statistics

Regardless of the design of the study, most researchers want to see descriptive statistics for the variables in their study. We offer three approaches for obtaining descriptive statistics below. For convenience we use the recent data set analytic_data_occasions. But recognize the commands below can be used with all the analytic data sets we created for the various designs.

3.13.1 skim()

One approach is the skim() command from the skimr package. The skim() command quickly provides the basic descriptive statistics. In the output for this command there are also several columns that begin with p: p0, p25, p50, p75, and p100 (p25 and p75 omitted in output due to space). These columns correspond to the 0th, 25th, 50th, 75th, and 100th percentiles, respectively. The minimum and maximum values for the data column are indicated under the p0 and p100 labels. The median is the 50th percentile (p50). The interquartile range is the range between p25 and p75. Notice that we run this command on the “wide” version of the data (analytic_data_occasions) rather than tidy version of the data (analytic_occasion_tidy).

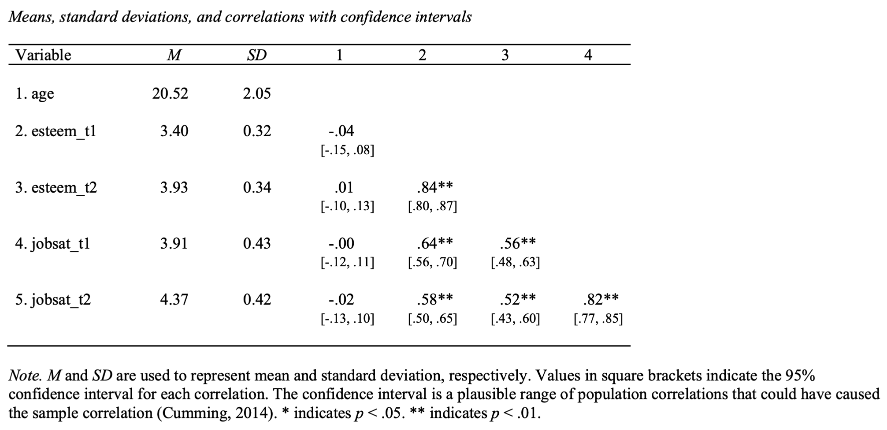
```
library(skimr)
skim(analytic_data_occasions)
```

```
##   skim_variable n_missing  mean   sd   p0  p50  p100
## 1           age         3 20.52 2.05 17.0 20.0 24.00
## 2     esteem_t1         0  3.40 0.32  2.5  3.4  4.25
## 3     esteem_t2         0  3.93 0.34  3.2  4.0  4.80
## 4     jobsat_t1         0  3.91 0.43  2.0  4.0  5.00
## 5     jobsat_t2         0  4.37 0.42  3.0  4.4  5.25
```

3.13.2 apa.cor.table()

Another approach is the apa.cor.table() command from the apaTables package. This quickly provides the basic descriptive statistics as well as correlations among variable. As well, it will even create a Word document with this information, see Figure 3.5. Notice that we run this command on the “wide” version of the data (analytic_data_occasions) rather than tidy version of the data (analytic_occasion_tidy).

```r
library(apaTables)
analytic_data_survey %>%
  select(where(is.numeric)) %>%
  apa.cor.table(filename = "apa_descriptives.doc")
```



Means, standard deviations, and correlations with confidence intervals

Variable	*M*	*SD*	1	2	3	4
1. age	20.52	2.05				
2. esteem_t1	3.40	0.32	-.04 [-.15, .08]			
3. esteem_t2	3.93	0.34	.01 [-.10, .13]	.84** [.80, .87]		
4. jobsat_t1	3.91	0.43	-.00 [-.12, .11]	.64** [.56, .70]	.56** [.48, .63]	
5. jobsat_t2	4.37	0.42	-.02 [-.13, .10]	.58** [.50, .65]	.52** [.43, .60]	.82** [.77, .85]

Note. M and *SD* are used to represent mean and standard deviation, respectively. Values in square brackets indicate the 95% confidence interval for each correlation. The confidence interval is a plausible range of population correlations that could have caused the sample correlation (Cumming, 2014). * indicates *p* < .05. ** indicates *p* < .01.

FIGURE 3.5: Word document created by apa.cor.table

3.13.3 tidyverse

A final approach uses tidyverse commands. This approach is oddly long - and we won’t describe how it works in detail. But, based on the information in the previous chapter you should be able to work out how this code works. Even though this code is long - it provide the ultimate in flexibility. If a new statistics is development that you want to use, you can simply include the command for it in the desired_descriptives list and it will be included in your table. Notice that we run this command on the “wide” version of the data (analytic_data_occasions) rather than tidy version of the data (analytic_occasion_tidy).

```
library(tidyverse)
# HMisc package must be installed.
# Library command not needed for HMisc package.

desired_descriptives <- list(
  mean = ~mean(.x, na.rm = TRUE),
  CI95_LL = ~Hmisc::smean.cl.normal(.x)[2],
  CI95_UL = ~Hmisc::smean.cl.normal(.x)[3],
  sd = ~sd(.x, na.rm = TRUE),
  min = ~min(.x, na.rm = TRUE),
  max = ~max(.x, na.rm = TRUE),
  n = ~sum(!is.na(.x))
)

row_sum <- analytic_data_occasions %>%
  summarise(across(.cols = where(is.numeric),
                   .fns =  desired_descriptives,
                   .names = "{col}___{fn}"))

long_summary <- row_sum %>%
  pivot_longer(cols = everything(),
               names_to = c("var", "stat"),
               names_sep = c("___"),
               values_to = "value")

summary_table <- long_summary %>%
  pivot_wider(names_from = stat,
              values_from = value)

summary_table_rounded <- summary_table %>%
  mutate(across(.cols = where(is.numeric),
                .fns= round,
                digits = 3)) %>%
  as.data.frame()

print(summary_table_rounded)
```

```
##         var   mean CI95_LL CI95_UL    sd  min   max   n
## 1       age 20.522  20.288  20.756 2.048 17.0 24.00 297
## 2 esteem_t1  3.403   3.366   3.440 0.324  2.5  4.25 300
## 3 esteem_t2  3.927   3.889   3.966 0.337  3.2  4.80 300
## 4 jobsat_t1  3.905   3.856   3.955 0.435  2.0  5.00 300
## 5 jobsat_t2  4.368   4.320   4.416 0.425  3.0  5.25 300
```

3.13.4 Cronbach's alpha

If you want Cronbach's alpha to estimate the reliability of the scale, you can use the alpha command from the psych package with the code below. Note we have to use the item level data we created a copy of called analytic_data_survey_items. The glimpse() command illustrates this data set has all the original items (after reverse-coding has been fixed).

```
analytic_data_survey_items %>%
  glimpse()
```

```
## Rows: 300
## Columns: 14
## $ id        <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12...
## $ age       <dbl> 23, 22, 18, 23, 22, 17, 23, 22, 17, N...
## $ sex       <fct> male, female, male, female, male, fem...
## $ eye_color <fct> blue, brown, hazel, blue, NA, hazel, ...
## $ esteem1   <dbl> 3, 4, 4, 3, 3, 3, 3, 4, 4, 4, 3, 4, N...
## $ esteem2   <dbl> 2, 3, 3, 2, 2, 3, 2, 3, 3, 3, 2, 2, N...
## $ esteem3   <dbl> 4, 4, 4, 3, 4, 4, NA, 4, 4, 3, 4, 4, ...
## $ esteem4   <dbl> 3, 4, 4, 3, 4, 4, 4, 4, 3, 4, NA, 4, ...
## $ esteem5   <dbl> 4, 4, 4, 4, 4, NA, NA, 4, 4, 4, 3, 4,...
## $ jobsat1   <dbl> 3, 5, 4, 3, 3, 3, 3, 5, 3, 3, 3, 4, 4...
## $ jobsat2   <dbl> 5, 5, 5, NA, 5, 5, 4, 5, 4, 4, 3, 5, ...
## $ jobsat3   <dbl> 3, NA, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...
## $ jobsat4   <dbl> NA, 5, 5, 4, 4, 4, 4, 5, NA, 4, NA, 5...
## $ jobsat5   <dbl> 5, NA, 5, 4, 5, 4, 4, 5, 5, 5, 4, NA,...
```

We calculated reliability using psych::alpha() command. Cronbach's alpha is labeled "raw alpha" in the output. Cronbach's alpha is an estimate of the proportion of variability in observed scores that is due to actual differences among participants (rather than measurement error). Remember, never use library(psych), it will break the tidyverse packages. Instead, precede all psych package commands with psych:: as we do below with psych::alpha().

```
rxx_alpha <- analytic_data_occasions_items %>%
  select(starts_with("t1_esteem")) %>%
  psych::alpha()

print(rxx_alpha$total)
```

```
##  raw_alpha std.alpha G6(smc) average_r  S/N     ase  mean
##     0.6622    0.6634  0.6173    0.2827 1.97 0.03035 3.403
##      sd median_r
##  0.3239   0.2927
```

4

Populations and samples

4.1 Required Packages

The following packages must be installed before starting this chapter.

Required Packages
tidyverse

The following data files are used in this chapter:

Required Data
data_ex_between.csv

The files are available at: https://github.com/dstanley4/psyc6060bookdown

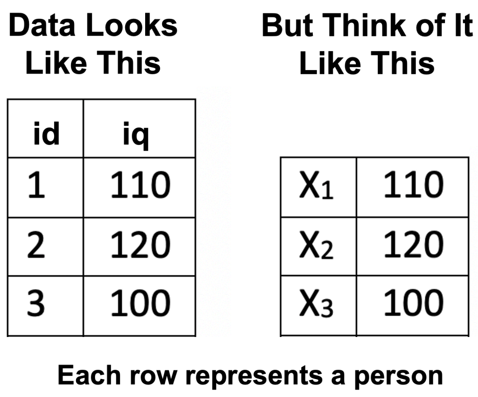
4.2 Objective

4.3 Notation

In this chapter we will use summation notation in the formulas describing population. If you are not familiar with summation notation, we present a brief overview here.

Consider a scenario where we have the IQ data for three participants We use the N symbol to represent sample size. Because we have three participants N = 3. The data for these participants is illustrated in Figure 4.1.

Notice how each person in the data set can be represent by the variable X. The first person by X_1, the second by X2, and the third by X_3. Often we refer to individuals in a data set by using the variable X accompanied by a subscript (e.g., 1, 2, 3, etc.).


Data Looks Like This
But Think of It Like This
Each row represents a person

id	iq
1	110
2	120
3	100

X_1	110
X_2	120
X_3	100

FIGURE 4.1: Data for understanding summation notation

Referring to participants using the variable X and subscript is valuable because it can be used in conjunction with the sigma (i.e., Σ) symbol for summation. Consider the example below in which we use the summation notation to indicate that we want to add all the X values (representing IQ) for the participants. We use a lower case i to represent all possible subscript values. The notation below the Σ, i = 1, indicates that we should start with participant 1. The notation below the Σ, N, indicates that we should iterate i up to the value indicated by N; in this case 3, because there are three participants.

$$\begin{aligned}\sum_{i=1}^{N} X_i &= X_1 + X_2 + X_3 \\ &= 110 + 120 + 100 \\ &= 330\end{aligned}$$

Sometime to simplify the notation, the numbers above and below the *Sigma* symbol are omitted. Likewise, the i subscript is omitted. There is a general understanding that we these components of the notation are omitted that the version of the notation above is implied.

$$\begin{aligned}\sum X &= X_1 + X_2 + X_3 \\ &= 110 + 120 + 100 \\ &= 330\end{aligned}$$

The full version of the notation can be used to indicate how a mean is calculated.

$$\begin{aligned}\bar{X} &= \frac{\sum_{i=1}^{N} X_i}{N} \\ &= \frac{X_1 + X_2 + X_3}{3} \\ &= \frac{110 + 120 + 100}{3} \\ &= \frac{330}{3} \\ &= 110\end{aligned}$$

Likewise, the concise version of the notation can be used to indicate how a mean is calculated.

$$\begin{aligned}\bar{X} &= \frac{\sum X}{N} \\ &= \frac{X_1 + X_2 + X_3}{3} \\ &= \frac{110 + 120 + 100}{3} \\ &= \frac{330}{3} \\ &= 110\end{aligned}$$

A common task in statistics is to calculate 1) the squared distance between each person and the mean, and 2) add up those squared differences. This calculation is easily expressed with the full version of the notation.

$$\begin{aligned}\sum_{i=1}^{N} (X_i - \bar{X})^2 &= (X_1 - \bar{X})^2 + (X_2 - \bar{X})^2 + (X_3 - \bar{X})^2 \\ &= (110 - 110)^2 + (120 - 110)^2 + (100 - 110)^2 \\ &= (0)^2 + (10)^2(-10)^2 \\ &= 0 + 100 + 100 \\ &= 200\end{aligned}$$

Likewise, the sum of the squared deviations from the mean can be expressed using the concise version of the notation.

$$\begin{aligned}\sum (X-\bar{X})^2 &= (X_1-\bar{X})^2+(X_2-\bar{X})^2+(X_3-\bar{X})^2\\ &= (110-110)^2+(120-110)^2+(100-110)^2\\ &= (0)^2+(10)^2(-10)^2\\ &= 0+100+100\\ &= 200\end{aligned}$$

4.4 Population vs samples

As we move closer to conducting our own research it is critical to make a distinction between populations and samples. A populations is the complete set of people/animals about which we want to make conclusions. Samples are a subset of the total number of people/animals in the population. In most scenarios it is impractical to work with an entire population and, for practical reasons, we study a subset of the population called a sample.

Researchers, and consumers of research, typically have little interest in making conclusions at the sample level. In general, we care about conclusions that generalize to the population not conlusions that only apply to specific individuals in the sample. Consider the case of COVID-19. Imagine a team of researchers creates a vaccine that they hope creates immunity to COVID-19. We care very little if the immunity only works for the specific indiviudals in the study. However, we care a great deal with if the immunity works, or is likely to work, for all Canadians or all humans. Thus, we study samples but typically wish to make conclusions that apply to the population.

Statistic tests are means of helping researchers use sample data to make conclusions at the population level. When we calculate numbers based on all of the people/animals in the population we refer to those numbers as **parameters**. We calculate numbers based on the people/animals in the sample we refer to those numbers as **statistics**. In general, statistics are used to estimate parameters.

4.5 Describing populations

```
library(tidyverse)
library(learnSampling)

pop_data <- get_height_population()
```

```
pop_data %>%
  summarise(pop_mean = mean(height)) %>%
  as.data.frame()
```

```
##   pop_mean
## 1    172.5
```

```
# set.seed ensures you get the same random samples
set.seed(1)

many_samples <- get_M_samples(pop.data = pop_data,
                              data.column.name = height,
                              n = 10,
                              number.of.samples = 1000)
```

The many_sample data set contains the results for all of the random samples we specified. Use head() to see the data. Each row corresponds to a random sample. In that row the sample number, sample size, population mean, and sample mean are provided (along with a few other numbers.

```
head(many_samples)
```

```
##   sample.number  n pop.M sample.M pop.var sample.var.n
## 1             1 10 172.5    175.5   157.5       189.85
## 2             2 10 172.5    169.6   157.5       196.64
## 3             3 10 172.5    161.7   157.5        92.61
## 4             4 10 172.5    178.4   157.5       102.04
## 5             5 10 172.5    174.7   157.5       350.81
## 6             6 10 172.5    166.0   157.5       236.00
##   sample.var.n_1
## 1          210.9
## 2          218.5
## 3          102.9
## 4          113.4
## 5          389.8
## 6          262.2
```

We can glimpse() the many_sample data to confirm the total number of rows is that same as we requested.

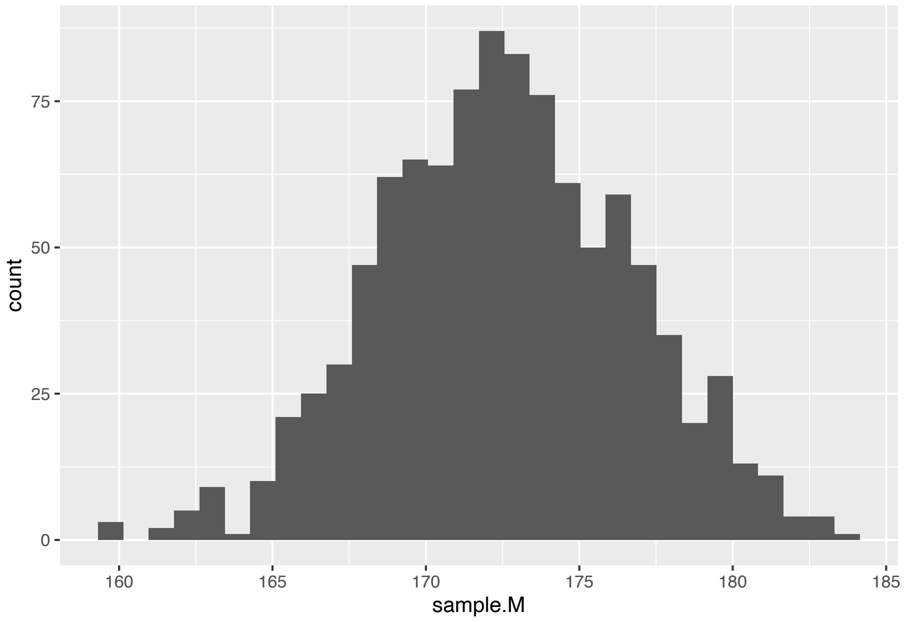
```r
glimpse(many_samples)
```

```
## Rows: 1,000
## Columns: 7
## $ sample.number  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1...
## $ n              <dbl> 10, 10, 10, 10, 10, 10, 10, 10, ...
## $ pop.M          <dbl> 172.5, 172.5, 172.5, 172.5, 172....
## $ sample.M       <dbl> 175.5, 169.6, 161.7, 178.4, 174....
## $ pop.var        <dbl> 157.5, 157.5, 157.5, 157.5, 157....
## $ sample.var.n   <dbl> 189.85, 196.64, 92.61, 102.04, 3...
## $ sample.var.n_1 <dbl> 210.94, 218.49, 102.90, 113.38, ...
```

Sampling variability

```r
ggplot(data = many_samples,
       mapping = aes(x = sample.M)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with
## `binwidth`.
```



Although any

```
many_samples %>%
  summarise(mean_of_means = mean(sample.M))
```

```
##   mean_of_means
## 1         172.6
```

A

More to Say

Yeah! I have finished my book, but I have more to say about some topics. Let me explain them in this appendix.

To know more about **bookdown**, see `https://bookdown.org`.

Bibliography

Baker, M. (2016). 1500 scientists lift the lid on reproducibility. *Nature*, 533.

Keur, C. and Hillegass, A. (2020). *iOS programming: The Big Nerd Ranch guide*. Pearson Technology Group.

Meyer, J. P., Allen, N. J., and Smith, C. A. (1993). Commitment to organizations and occupations: Extension and test of a three-component conceptualization. *Journal of applied psychology*, 78(4):538.

Miyakawa, T. (2020). No raw data, no science: another possible source of the reproducibility crisis. *Mol Brain*, 13(24).

Nosek (2015). Estimating the reproducibility of psychological science. *Science*, 349.

Patil P., Peng R.D., . L. J. (2019). A visual tool for defining reproducibility and replicability. *Nat Hum Behav*, 3:650–652.

Simmons, J. P., Nelson, L. D., and Simonsohn, U. (2011). False-positive psychology: Undisclosed flexibility in data collection and analysis allows presenting anything as significant. *Psychological science*, 22(11):1359–1366.

Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and 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 0.19.1.

Index