Engaging with MIDFIELD Data

Susan Lord, Matthew Ohland, Marisa Orr, Richard Layton, and Russell Long

2021-07-26

Contents

1 Introduction **5**
- 1.1 Objectives . . . 5
- 1.2 Description . . . 5
- 1.3 Pre-workshop homework . . . 6
- 1.4 Agenda . . . 6
- 1.5 Facilitators . . . 6
- 1.6 Licenses . . . 7
- 1.7 Acknowledgement . . . 7

2 Install everything **9**
- 2.1 Already installed? . . . 9
- 2.2 Install R and RStudio . . . 10
- 2.3 Create a project . . . 11
- 2.4 Add some folders . . . 11
- 2.5 Install CRAN packages . . . 12
- 2.6 Install MIDFIELD packages . . . 13
- 2.7 Exiting R and RStudio . . . 14

3 Working with R **15**
- 3.1 Prerequisites . . . 15
- 3.2 New to R? . . . 16
- 3.3 Familiar with R? . . . 16
- 3.4 After the workshop . . . 16

4 R basics **19**
- 4.1 Style guide . . . 19
- 4.2 Open an R script . . . 20
- 4.3 Everything in R has a name . . . 22
- 4.4 Everything in R is an object . . . 22
- 4.5 R functions do things . . . 26
- 4.6 R functions come in packages . . . 29
- 4.7 R objects have class . . . 30
- 4.8 R objects have structure . . . 32

4.9 R only does what you tell it . . . 33
4.10 To get help . . . 35
4.11 Keyboard shortcuts . . . 37
4.12 Next steps . . . 38

5 Graph basics **39**
5.1 Introduction . . . 39
5.2 Expected data structure . . . 40
5.3 Anatomy of a graph . . . 41
5.4 Layer: points . . . 46
5.5 Layer: smooth fit . . . 47
5.6 Layer: scale . . . 50
5.7 Mapping columns to aesthetics . . . 52
5.8 Layer: facets . . . 53
5.9 Ordering panels and rows . . . 57

6 Data basics **61**
6.1 Introduction . . . 61

7 Designing effective displays **63**
7.1 Stuff . . . 63
7.2 More stuff . . . 63

Chapter 1

Introduction

1.1 Objectives

MIDFIELD contains individual Student Unit Record (SUR) data for 1.7M students at 33 US institutions (as of June 2021). MIDFIELD is large enough to permit grouping and summarizing by multiple characteristics, enabling researchers to examine student characteristics (race/ethnicity, sex, prior achievement) and curricular pathways (including coursework and major) by institution and over time.

The goal of this workshop is to engage ASEE members with MIDFIELD data, in part using two R packages: **midfieldr** to provide tools for working with SURs and **midfielddata** to provide practice data.

By the end of the workshop, participants should be able to:

- Describe key variables in MIDFIELD data tables
- Explore and tell a story from MIDFIELD data
- Begin using R, midfieldr, and midfielddata

- Explain key features of effective data displays

1.2 Description

The robustness of the MIDFIELD data allows us to emphasize an intersectional approach to the study of student records, permitting multiple categories of inequity such as race/ethnicity and sex to be considered simultaneously.

To introduce beginners to R, participants work through a self-paced tutorial covering basic elements of the R computing language and environment. To introduce midfieldr and using it to work with student record data, participants

work through software tutorials that explore the SUR data and develop a persistence metric case study.

For more experienced R users or anyone working at a faster pace, we offer a series of self-paced tutorials that introduce key features of midfieldr and how they are applied to compute persistence metrics and graph results.

We also discuss the merits of the multiway graph design that is recommended for displaying results of this type. The agenda includes an interactive session to demonstrate contemporary principles of effective data display.

1.3 Pre-workshop homework

To get the most out of the workshop, you should have the essential software installed and running several days before the workshop to give you time to contact us with questions if anything goes amiss.

Install everything describes the homework.

1.4 Agenda

Our three hours are organized approximately as shown.

Min	Topic
15	Introduction
35	Exploring the data structure
35	Working with R
15	Break
20	Designing effective displays
50	Working with R (continued)
10	Next steps & assessing the workshop

1.5 Facilitators

Susan Lord Director of the MIDFIELD Institute and Professor and Chair of Integrated Engineering at the University of San Diego. She is a Fellow of the IEEE and the ASEE. Dr. Lord has considerable experience facilitating workshops including the National Effective Teaching Institute (NETI) and special sessions at FIE. (slord@sandiego.edu)

Matthew Ohland MIDFIELD Director and Principal Investigator. He is Professor and Associate Head of Engineering Education at Purdue University and a Fellow of IEEE, ASEE, and AAAS. Dr. Ohland has considerable experience facilitating workshops including the NETI and CATME training. (ohland@purdue.edu)

Marisa Orr MIDFIELD Associate Director and Associate Professor in Engineering and Science Education with a joint appointment in Mechanical Engineering at Clemson University. She received the 2009 Helen Plants Award for the best nontraditional session at FIE, "Enhancing Student Learning Using SCALE-UP Format." (marisak@clemson.edu)

Richard Layton MIDFIELD Data Visualization Specialist and Professor Emeritus of Mechanical Engineering at Rose-Hulman Institute of Technology. He is the lead developer of the R packages used in this workshop. Dr. Layton has considerable experience facilitating workshops, including FIE workshops on data visualization (2014) and midfieldr (2018). (graphdoctor@gmail.com)

Russell Long MIDFIELD Managing Director and Data Steward. He developed the stratified data sample for the R packages used in this workshop. Mr. Long is a SAS expert with over twenty years of experience in institutional research and assessment. (ralong@purdue.edu)

1.6 Licenses

1.7 Acknowledgement

Funding provided by the National Science Foundation Grant 1545667 "Expanding Access to and Participation in the Multiple-Institution Database for Investigating Engineering Longitudinal Development."

top of page

Chapter 2

Install everything

If you are trying R for the first time, it is vital that you attempt to set up your computer with the necessary software in advance or it will be difficult to keep up.

This chapter describes all of the pre-workshop homework:

- Already installed?
- Install R and RStudio
- Create a project

- Add some folders
- Install CRAN packages
- Install MIDFIELD packages

2.1 Already installed?

If you do not have R and RStudio installed, please skip this section and start with the next section. If you are already an R and RStudio user, this is a great time to check for updates.

Updating RStudio

- RStudio menu *Help > Check for Updates* will tell you if you are current or not.
- To update, close RStudio on your machine, download the new version from the RStudio website, and run the `RStudio-some-version-number.exe`. (Windows users *might* have to run the executable as an administrator.)

Update your packages

How to upgrade all out-of-date packages in *What They Forgot to Teach You About R* by Jennifer Bryan and Jim Hester.

Updating R

The easiest way to update R is to simply download the newest version. RStudio will automatically use the latest you've installed.

Alternatively, Windows users can use the installr package:

- Install the installr package
- If open, close R and RStudio
- Navigate to your most recent `Rgui.exe` file located in your Programs directory, e.g., `C:\Program Files\R\R-4.0.0\bin\x64\Rgui.exe`
- Right-click on `Rgui.exe` and run as administrator
- In the R GUI window that appears, run the commands

```
# Windows users only
library("installr")
updateR()
```

Updating your R library

How to transfer your library when updating R also by Bryan and Hester. Requires the fs package.

Once your updates are complete

Skip the next section and continue the homework with

- Create a project
- Add some folders
- Install CRAN packages
- Install MIDFIELD packages

2.2 Install R and RStudio

The first steps are to install R and RStudio. Windows users may have to login as an Administrator before installing the software.

- Install R for your operating system
- Install RStudio, a user interface for R

Once the installation is complete, you can take a 2-minute tour of the RStudio interface.

- Let's start (00:57–02:32) by R Ladies Sydney [Richmond, 2018]

The same video includes a longer (7 minute) tour of the four quadrants (panes) in RStudio if you are interested.

- The RStudio quadrants (07:21–14:40) by R Ladies Sydney [Richmond, 2018]

2.3 Create a project

To begin any project, we create an RStudio *Project* file and directory. You can recognize an R project file by its *.Rproj* suffix.

If you prefer your instructions with commentary,

- Start with a Project (02:34–04:50) by R Ladies Sydney [Richmond, 2018]

If you prefer basic written instructions,

- RStudio, *File > New Project… > New Directory > New Project*
- Or, click the *New Project* button in the Console ribbon,

In the dialog box that appears,

- Type the workshop name as the directory name, for example, `workshop`, or if you like more detail, `midfield-workshop-asee-2021`
- Use the browse button to select a location on your computer to create the project folder
- Click the *Create Project* button

Whenever you work with the workshop materials, launch the `workshop.Rproj` file (using the name you actually used) to start the session.

2.4 Add some folders

While file organization is a matter of personal preference, we ask that you use the directory structure shown here for your work in the workshop. Assuming we called our project `workshop`, the minimal directory structure has three folders in it plus the `.Rproj` file at the top level.

```
\workshop
    \data
    \results
    \scripts
    workshop.Rproj
```

We use the folders as follows:

- `data` data files
- `results` finished graphs and tabulated data formatted for display
- `scripts` R scripts that operate on data to produce results

To create folders:

- use your usual method of creating new folders on your machine
- or you can use the *New Folder* button in the Files pane

For a video guide,

- Make some folders (04:50–06:08) by R Ladies Sydney [Richmond, 2018]

2.5 Install CRAN packages

The fundamental unit of shareable code in R is the *package.* For the R novice, an R package is like an "app" for R—a collection of functions, data, and documentation for doing work in R that is easily shared with others [Wickham, 2014].

Most packages are obtained from the CRAN website [cra, 2018-04-22]. To install a CRAN package using RStudio:

- Launch RStudio

The RStudio interface has several panes. We want the *Files/Plots/Packages* pane.

- Select the *Packages* tab

Next,

- Click *Install* on the ribbon
- In the dialog box, type the name of the package. For our first package, type `data.table` to install the data.table package [Dowle and Srinivasan, 2021]
- Check the *Install dependencies* box
- Click the *Install* button

  During the installation, Windows users might get a warning message about Rtools, something like:

  ```
  WARNING: Rtools is required to build R packages but is
  not currently installed. Please download and install the
  appropriate version....
  ```

Rtools is needed for packages with C/C++/FORTRAN code from source—which does not apply to us. You may ignore the warning and carry on.

In the RStudio Console, you should see a message like this one,

```
    package 'data.table' successfully unpacked and MD5 sums checked
```

If successful, the package will appear in the Packages pane, e.g.,

Repeat the process for the following packages

```
wrapr
Rdpack
checkmate
ggplot2
```

Alternatively, you can install them all at once by typing in the Console:

```
packages_we_use <- c("data.table", "wrapr", "Rdpack", "checkmate", "ggplot2")
install.packages(packages_we_use)
```

2.6 Install MIDFIELD packages

2.6.1 midfieldr

midfieldr is not yet available from CRAN. To install the development version of midfieldr from its `drat` repository, type in the Console:

```
# type in the RStudio Console
install.packages("midfieldr",
                 repos = "https://MIDFIELDR.github.io/drat/",
                 type = "source")
```

You can confirm a successful installation by running the following lines to bring up the package help page in the Help window.

```
# type in the RStudio Console
library("midfieldr")
? `midfieldr-package`

# or, equivalently
help("midfieldr-package")
```

If the installation is successful, the code chunk above should produce a view of the help page as shown here.

2.6.2 midfielddata

Because of its size, the data package is stored in a `drat` repository instead of CRAN. Installation takes time; please be patient and wait for the Console prompt ">" to reappear.

Type (or copy and paste) the following lines in the RStudio Console.

```
# type in the RStudio Console
install.packages("midfielddata",
                 repos = "https://MIDFIELDR.github.io/drat/",
                 type = "source")
# be patient
```

Once the Console prompt ">" reappears, you can confirm a successful installation by viewing the package help page. In the Console, run:

```
# type in the RStudio Console
library("midfielddata")
help("midfielddata-package")
```

If the installation is successful, the code chunk above should produce a view of the help page as shown here.

2.7 Exiting R and RStudio

When you exit R/RStudio, you probably get a prompt about saving your workspace image.

The answer is No.

You can turn this prompt off by using the pulldown menu,

- *Tools > Global Options…*
- In the dialog box, *Save workspace to .RData on exit:* Select "Never"

You finished your homework!

top of page

Chapter 3

Working with R

R is an open source language and environment for statistical computing and graphics [R Core Team, 2021], ranked by IEEE in 2020 as the 6th most popular programming language (Python, Java, and C are the top three) [Cass, 2020]. If you are new to R, some of its best features, paraphrasing Wickham [2014], are:

- R is free, open source, and available on every major platform.
- R packages provide effective tools for data analysis and visualization.
- More than 17,750 open-source R packages are available (Jul 2021). Many are cutting-edge tools.

RStudio, an integrated development environment (IDE) for R, includes a console, editor, and tools for plotting, history, debugging, and workspace management as well as access to GitHub for collaboration and version control [RStudio Team, 2016].

3.1 Prerequisites

Before proceeding, you should have completed

- Install everything

- Launched your workshop project—`workshop.Rproj` or other name that you selected—to start the R session

With these prerequisites accomplished, and working in your RStudio `workshop.Rproj` project, you can create a save a script by:

- Use the pulldown menu, *File > New File > R Script*
- File menu > Save As…
- In the dialog box, navigate to your `scripts` directory, type a file name, for example, `01-R-basics.R` (file names in R can start with numerals),

and *Save*.

We suggest you start a new R script for each tutorial and save it to the `scripts` directory. For example, at the end of the workshop, your scripts directory might contain the following files:

```
\scripts
    \01-R-basics.R
    \02-getting-started.R
    \03-case-study-programs.R
    \04-case-study-students.R
    etc.
```

3.2 New to R?

Prerequisites should be completed before proceeding. By the end of the workshop, our R beginners will have made progress on two or possibly three tutorials:

- R basics An introduction to R.
- Getting started: Examine the MIDFIELD practice data
- Case study programs Gather CIP codes and program names

If there is still time remaining, continue to any tutorial listed in the After the workshop section.

3.3 Familiar with R?

Prerequisites should be completed before proceeding. By the end of the workshop, our more experienced R users will have made substantive progress on two or possibly three tutorials:

- Getting started: Examine the MIDFIELD practice data
- Case study programs Gather CIP codes and program names
- Case study students Gather students who pass the data sufficiency criterion.

If there is still time remaining, continue to any tutorial listed in the After the workshop section.

3.4 After the workshop

At his point, the learning is self-directed. Choose the skills you want to continue working on. We have a set of tutorials for

- Developing R skills

- Continuing the case study
- Exploring midfieldr functions

3.4.1 Developing R skills

The basic skills tutorials take about 50 minutes each.

- R basics
- Graph basics

- Data basics

3.4.2 Continuing the case study

The case study is a quick tour of a typical workflow using student unit record data. This is a "big picture" development—functions are used without detailed explanations or development so that we can focus on the logic of the analysis. For exploring midfieldr functions and methods in greater detail, see the vignettes listed in the next section.

- Case study programs
- Case study students
- Case study stickiness
- Case study graduation rate

3.4.3 Exploring midfieldr functions

Deep dive into the midfieldr functionality. The work flow follows the same general pattern as the quicker case study, but pauses to explore each function in more detail, exploring the arguments and strategies for use. In general, each tutorial is self-contained so you may enter at almost any point.

- Program codes and names Practice strategies of searching `cip` for programs we want to study.
- Subsetting MIDFIELD data Use programs codes to subset the MIDFIELD data tables.
- Data sufficiency What it is and how it is applied to student unit-record (SUR) data.
- Timely completion What it is and how it is applied to SUR data.
- FYE programs What they are and how they are accommodated with SUR data.
- Multiway graphs How to graph and interpret a common data structure encountered when working with SUR data.
- Tabulating data How to tabulate multiway data for publication.

top of page

Chapter 4

R basics

This tutorial is an introduction to R adapted from [Healy, 2019b] with extra material from [Matloff, 2019]. If you already have R experience, you might still want to browse this section in case you find something new.

Prerequisites should be completed before proceeding. After that, the tutorial should take no longer than 50 minutes.

4.1 Style guide

A style guide is about making your script readable. We ask you to observe a small set of guidelines that will help us help you when you have questions about your script.

Comments in R are denoted by a hash tag `#`.

- Everything to the right of the hash tag is ignored by R.
- Comments that describe "why" are generally more useful than comments that explain "how."

Spaces around operators. Usewhitespacetoenhancereadability. Place spaces around operators (`=`, `+`, `-`, `<-`, etc.). Always put a space after a comma, but never before (just like in regular English).

```
# poor
height<-feet*12+inches
mean(x,na.rm=10)

# better
height <- (feet * 12) + inches
mean(x, na.rm = 10)
```

Use vertical white space. Lack of vertical white space makes your script harder to read (like a story with no paragraphs).

```
# Poor, no paragraph breaks
library("data.table"); library("GDAdata")
speed_ski <- copy(SpeedSki)
setDT(speed_ski)
speed_ski <- speed_ski[, .(Event, Sex, Speed)]
setnames(speed_ski, old = c("Event", "Sex", "Speed"), new = c("event", "sex", "speed"))
```

Instead, group chunks of code into paragraphs separated by blank lines to reveal the structure of the program. Comments at the start of a code chunk can explain your intent (like a topic sentence). Here we illustrate commenting on "why" rather than "how".

```
# Better example with code in paragraphs

library("data.table")
library("GDAdata")

# Leave the original data unaltered before data.table conversion
speed_ski <- copy(SpeedSki)
setDT(speed_ski)

# Only three variables are required
speed_ski <- speed_ski[, .(Event, Sex, Speed)]

# Lowercase column names are our preferred style
setnames(speed_ski,
         old = c("Event", "Sex", "Speed"),
         new = c("event", "sex", "speed"))

# RDS format preserves factors
saveRDS(speed_ski, "data/speed_ski.rds")
```

For more information of R scripting style generally see McConnell [2004] and Wickham [2019].

4.2 Open an R script

Use *File > New File > R Script* to create a new R script

- Name the script `01-R-basics.R`. By using a number at the start of the file name, the files stay in order in your directory.

- Save it in the `scripts` directory.
- Code chunks like the one below can be copied and pasted to your R script.

Add a minimal header at the top of the script. Use `library()` to load the packages we will be using.

```
# R basics
# name
# date

# packages
library("midfieldr")
library("data.table")
```

After adding a code chunk to your script, run the script. Options for running a script:

- To run an entire script, select all lines with `ctrl A` then run the lines using `ctrl Enter` (for the Mac OS: `cmd A` and `cmd Return`).
- To run select lines, use the cursor to select the lines you want to run, then `ctrl Enter` (for the Mac OS: `cmd Return`).
- To run from the beginning to a line, place your cursor at the line, then `ctrl alt B` (`cmd option B` Mac OS)

If you get an error similar to:

```
Error in library("data.table") : there is no package called 'data.table'
```

then the package needs to be installed. If you need a refresher on installing packages, see Install CRAN packages. Once the missing package is installed, you can rerun the script.

The following code chunk is optional for controlling the number of rows of a data frame that are printed to the Console screen.

```
# Optional code to control data.table printing
options(
  datatable.print.nrows = 10,
  datatable.print.topn = 5,
  datatable.print.class = TRUE
)
```

Guidelines

- As you work through the tutorial, type a line or chunk of code then *File > Save* and run the script.
- Confirm that your result matches the tutorial result.
- **Your turn** exercises give you chance to devise your own examples and check them out. You learn by doing (but you knew that already)!

4.3 Everything in R has a name

In R, every object has a name.

- named entities, like `x` or `y`
- data you have loaded, like `my_data`
- functions you use, like `sin()`

Some names are forbidden

- reserved words, like `TRUE` or `FALSE`
- programming words, like `Inf`, `for`, `else`, and `function`
- special entities, like `NA` and `NaN`

Some names should not be used because they name commonly used functions

- `q()` quit
- `c()` combine or concatenate
- `mean()`
- `range()`
- `var()` variance

Names in R are case-sensitive

- `my_data` and `My_Data` are different objects
- We use the style of naming things in lower case with words separated by underscores (no spaces), e.g., `speed_ski`. The camel-case is also popular, e.g., `SpeedSki` or `speedSki`. The choice is yours.

If you want to know if a name has already been used in a package you have loaded, go to the RStudio console, type a question mark followed by the name, e.g.,

```
# Type in the Console
`? c()`
`? mean()`
```

If the name is in use, a help page appears in the RStudio Help pane.

4.4 Everything in R is an object

Origins of R objects

- Some objects are built in to R
- Some objects are loaded with packages
- Some objects are created by you

Type this line of code in your script, *Save*. `c()` is the function to combine or concatenate its elements to create a vector.

```
# Type in the R script
c(1, 2, 3, 1, 3, 25)
```

Run the script and your Console should show `[1]  1  2  3  1  3 25`.

In these notes, when we show results printed in your Console, we preface the printout with `#>` (which does not appear on your screen) to distinguish the results from the script. For example, we show the line from above and its output like this:

```
c(1, 2, 3, 1, 3, 25)          # <- typed in the script
#> [1]  1  2  3  1  3 25      # <- appears in the Console
```

The `[1]` that leads the output line is a label identifying the index of the element that starts that line. More on that in a little while.

You create objects my assigning them names.

- `<-` is the assignment operator, keyboard shortcut: `alt`– , i.e., the ALT key plus the hyphen key. (Mac OS `option`–)

```
# Practice assigning an object to a name
x <- c(1, 2, 3, 1, 3, 25)
y <- c(5, 31, 71, 1, 3, 21, 6)
```

To see the result in the Console, type the object name in the script, *Save*, and run. (Remember, type the line of code but not the line prefaced by `#>`—that's the output line so you can check your results.)

```
# Type in the R script or in the Console
x
#> [1]  1  2  3  1  3 25

y
#> [1]  5 31 71  1  3 21  6
```

Objects exist in your R project workspace, listed in the RStudio Environment pane

Data are also named objects. For example, midfieldr has several toy data sets included for use in illustrative examples like this one. Type its name in the script,

```
# Examine a data frame included with midfieldr
toy_student
#>             mcid   institution              transfer hours_transfer
#>           <char>        <char>                <char>          <num>
#>   1: MID25783939 Institution M First-Time in College             NA
```

```
#>   2: MID25784402 Institution M First-Time in College             NA
#>   3: MID25805538 Institution M First-Time in College             NA
#>   4: MID25808099 Institution M First-Time in College             NA
#>   5: MID25816437 Institution M First-Time in College             NA
#>  ---
#>  96: MID26656134 Institution L First-Time in College             NA
#>  97: MID26656367 Institution L First-Time in College             NA
#>  98: MID26663803 Institution L First-Time in College             NA
#>  99: MID26678321 Institution L First-Time in College             NA
#> 100: MID26692008 Institution L First-Time in College             NA
#>                 race    sex
#>               <char> <char>
#>   1:           White Female
#>   2:           White   Male
#>   3:           White Female
#>   4:           White Female
#>   5:           White   Male
#>  ---
#>  96: Native American   Male
#>  97: Hispanic/Latinx   Male
#>  98:   International   Male
#>  99:           White Female
#> 100:           White   Male
```

To view the help page for the data, type in the Console

```
# type in the Console
? toy_student
```

If we wanted the first five rows of the toy data, we use the `[]` operator.

```
# Practice using the `[` operator
toy_student[1:5]
#>           mcid   institution              transfer hours_transfer   race    sex
#>         <char>        <char>                <char>          <num> <char> <char>
#> 1: MID25783939 Institution M First-Time in College             NA  White Female
#> 2: MID25784402 Institution M First-Time in College             NA  White   Male
#> 3: MID25805538 Institution M First-Time in College             NA  White Female
#> 4: MID25808099 Institution M First-Time in College             NA  White Female
#> 5: MID25816437 Institution M First-Time in College             NA  White   Male
```

To view the help page for the `[` operator, surround the symbol with "back-ticks" (on your keyboard with the tilde ~ symbol). For example,

```
# view the help page on the R extract operator
? `[`
```

To extract a single column, e.g. the ID column, but preserve the data frame

structure,

```
# Subset a column as a data table
toy_student[, .(mcid)]
#>              mcid
#>            <char>
#>   1: MID25783939
#>   2: MID25784402
#>   3: MID25805538
#>   4: MID25808099
#>   5: MID25816437
#>  ---
#>  96: MID26656134
#>  97: MID26656367
#>  98: MID26663803
#>  99: MID26678321
#> 100: MID26692008
```

We can also extract the column as a vector using slightly different syntax,

```
# Subset a column as a vector
toy_student[, mcid]
#>   [1] "MID25783939" "MID25784402" "MID25805538" "MID25808099" "MID25816437"
#>   [6] "MID25826223" "MID25828870" "MID25831839" "MID25839453" "MID25840802"
#>  [11] "MID25841465" "MID25845841" "MID25846316" "MID25847220" "MID25848589"
#>  [16] "MID25852023" "MID25853332" "MID25853799" "MID25877946" "MID25880643"
#>  [21] "MID25887008" "MID25899243" "MID25911361" "MID25913454" "MID25931457"
#>  [26] "MID25947836" "MID25982250" "MID25995980" "MID25997636" "MID26000057"
#>  [31] "MID26004638" "MID26013461" "MID26020535" "MID26046521" "MID26048632"
#>  [36] "MID26060301" "MID26062203" "MID26062778" "MID26086310" "MID26088450"
#>  [41] "MID26102824" "MID26136319" "MID26138017" "MID26152744" "MID26161677"
#>  [46] "MID26170598" "MID26173721" "MID26181209" "MID26187436" "MID26204281"
#>  [51] "MID26211998" "MID26235812" "MID26244053" "MID26247839" "MID26305709"
#>  [56] "MID26305863" "MID26309255" "MID26319252" "MID26332563" "MID26356320"
#>  [61] "MID26358462" "MID26370377" "MID26383411" "MID26384771" "MID26391215"
#>  [66] "MID26400804" "MID26413466" "MID26417039" "MID26418247" "MID26421588"
#>  [71] "MID26421846" "MID26422829" "MID26429192" "MID26433811" "MID26435945"
#>  [76] "MID26439623" "MID26441609" "MID26453554" "MID26461158" "MID26481120"
#>  [81] "MID26526195" "MID26528318" "MID26546600" "MID26560837" "MID26561940"
#>  [86] "MID26575282" "MID26577489" "MID26578111" "MID26588553" "MID26592425"
#>  [91] "MID26592668" "MID26596818" "MID26605008" "MID26607528" "MID26655230"
#>  [96] "MID26656134" "MID26656367" "MID26663803" "MID26678321" "MID26692008"
```

Here you can see how the row labels in the printed output work. There are 5 IDs per row, so the second row starts with the 6th ID, indicated by `[6]`. The last row starts with the 96th value `[96]` and ends with the 100th value.

The "toy" data sets in midfieldr (`toy_student`, `toy_course`, `toy_term`, and

`toy_degree`) include student unit records for only 100 students—not a statistically representative sample—used for package examples like those above.

4.5 R functions do things

Functions do something useful

- Functions are objects the perform actions for you
- Functions produce output based on the input it receives
- Functions are recognized by the parentheses at the end of their names

The parentheses are where we include the inputs (arguments) to the function

- `c()` concatenates the comma-separated numbers in the parentheses to create a vector
- `mean()` computes the mean of a vector of numbers
- `sd()` computes the standard deviation of a vector of numbers
- `summary()` returns a summary of the object

If we try `mean()` with no inputs, we get an error statement

```
mean()
#> Error in mean.default() : argument "x" is missing, with no default
```

Let's determine some summary statistics on our student transfer hours. Add these lines to your script, *Save*, and run.

```
# Extract a column as a vector
transfer_hours <- toy_student[, hours_transfer]

# Examine the vector
transfer_hours
#>   [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA 30 55 NA 24 NA NA NA
#>  [26] NA NA NA NA  4 NA  2 NA NA NA  1  7  1  3  1  5 NA NA NA NA NA NA NA NA
#>  [51] NA NA NA NA NA NA NA NA NA NA 80 NA NA NA NA NA NA NA NA NA NA NA NA NA
#>  [76] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA

# Operate on the vector
mean(transfer_hours)
#> [1] NA
```

We have to set the optional argument `na.rm` ("remove NA") to take a mean

```
# Operate and ignore NA values
mean(transfer_hours, na.rm = TRUE)
#> [1] 17.75

# Another operation
sd(transfer_hours, na.rm = TRUE)
```

```
#> [1] 25.63068

# The summary also shows the count of NAs
summary(transfer_hours)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#>    1.00    1.75    4.50   17.75   25.50   80.00      88
```

Functions to examine a data frame.

- `names()` Data frame column names
- `head()` and `tail()` First few and last few rows of a data frame

```
# Practice finding column names
names(toy_student)
#> [1] "mcid"           "institution"    "transfer"       "hours_transfer"
#> [5] "race"           "sex"

# Practice examining the first few rows of a data frame
head(toy_student)
#>           mcid   institution               transfer hours_transfer   race    sex
#>         <char>        <char>                 <char>          <num> <char> <char>
#> 1: MID25783939 Institution M First-Time in College             NA  White Female
#> 2: MID25784402 Institution M First-Time in College             NA  White   Male
#> 3: MID25805538 Institution M First-Time in College             NA  White Female
#> 4: MID25808099 Institution M First-Time in College             NA  White Female
#> 5: MID25816437 Institution M First-Time in College             NA  White   Male
#> 6: MID25826223 Institution M   First-Time Transfer             NA  White Female

# Practice examining the last few rows of a data frame
tail(toy_student)
#>           mcid   institution               transfer hours_transfer
#>         <char>        <char>                 <char>          <num>
#> 1: MID26655230 Institution L First-Time in College             NA
#> 2: MID26656134 Institution L First-Time in College             NA
#> 3: MID26656367 Institution L First-Time in College             NA
#> 4: MID26663803 Institution L First-Time in College             NA
#> 5: MID26678321 Institution L First-Time in College             NA
#> 6: MID26692008 Institution L First-Time in College             NA
#>               race    sex
#>             <char> <char>
#> 1:           White Female
#> 2: Native American   Male
#> 3: Hispanic/Latinx   Male
#> 4:   International   Male
#> 5:           White Female
#> 6:           White   Male
```

Functions to examine columns (variables) in a data frame.

- `sort()` and `unique()` often used together
- `is.na()` to return TRUE for every NA element in an object, otherwise FALSE
- `sum()` applied to `is.na()` converts logical TRUE to 1 and FALSE to 0 and adds the elements. The resulting integer is the number of NA values in the vector.

```
# Determine the unique values in a column
sort(unique(toy_student[, sex]))
#> [1] "Female" "Male"

# Find the rows with NA values.
is.na(toy_student[, sex])
#>   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [97] FALSE FALSE FALSE FALSE

# How many values are NA?
sum(is.na(toy_student[, sex]))
#> [1] 0
```

Repeat for other columns.

```
# Practice examining another column
sort(unique(toy_student[, institution]))
#>  [1] "Institution A" "Institution B" "Institution C" "Institution D"
#>  [5] "Institution E" "Institution F" "Institution G" "Institution H"
#>  [9] "Institution J" "Institution K" "Institution L" "Institution M"
sum(is.na(toy_student[, institution]))
#> [1] 0

# Practice examining another column
sort(unique(toy_student[, race]))
#> [1] "Asian"           "Black"           "Hispanic/Latinx" "International"
#> [5] "Native American" "Other/Unknown"   "White"
sum(is.na(toy_student[, race]))
#> [1] 0

# Practice examining another column
```

```
sort(unique(toy_student[, hours_transfer]))
#>  [1]  1  2  3  4  5  7 24 30 55 80
sum(is.na(toy_student[, hours_transfer]))
#> [1] 88
```

The help pages for functions are accessed via the Console. By viewing the help page you can find descriptions of arguments and their default settings if any. Try a few:

- `? mean()`
- `? sd()`
- `? summary()`
- `? names()`
- `? head()`
- `? sort()`
- `? unique()`
- `? is.na()`
- `? sum()`

4.6 R functions come in packages

Functions are bundled in packages

- Families of useful functions are bundled into packages that you can install, load, and use
- Packages allow you to build on the work of others
- You can write your own functions and packages too
- A lot of the work in data science consists of choosing the right functions and giving them the right arguments to get our data into the form we need for analysis or visualization

For example, to see the list of functions in the midfieldr package,

```
# Showing all functions in the midfieldr package
sort(getNamespaceExports("midfieldr"))
#> [1] "add_completion_timely" "add_data_sufficiency"  "add_institution"
#> [4] "add_race_sex"          "add_timely_term"       "condition_fye"
#> [7] "condition_multiway"    "filter_match"          "filter_search"
```

To view a help page, type, e.g.,

```
# Type in the Console to view a help page
? add_race_sex()
```

In contrast, do the same for the data.table package,

```
# A package with many functions.
sort(getNamespaceExports("data.table"))
```

```
#>   [1] "%between%"           "%chin%"              "%flike%"
#>   [4] "%ilike%"             "%inrange%"           "%like%"
#>   [7] ".__C__data.table"    ".__C__IDate"         ".__C__ITime"
#>  [10] ".__T__$:base"        ".__T__$<-:base"      ".__T__[:base"
#>  [13] ".__T__[[<-:base"     ".__T__[<-:base"      ".BY"
#>  [16] ".EACHI"              ".GRP"                ".I"
#>  [19] ".Last.updated"       ".N"                  ".NGRP"
#>  [22] ".rbind.data.table"   ".SD"                 ":="
#>  [25] "address"             "alloc.col"           "as.data.table"
#>  [28] "as.IDate"            "as.ITime"            "as.xts.data.table"
#>  [31] "between"             "chgroup"             "chmatch"
#>  [34] "chorder"             "CJ"                  "copy"
#>  [37] "cube"                "data.table"          "dcast"
#>  [40] "dcast.data.table"    "fcase"               "fcoalesce"
#>  [43] "fifelse"             "fintersect"          "first"
#>  ... etc. truncated
```

Don't panic! We will use only a small number of these functions from data.table. For example, the `%ilike%` function, view its help page by running

```
# Type in the Console to view the help page
? `%ilike%`
```

4.7 R objects have class

Everything is an object and every object has a class.

```
class(x)
#> [1] "numeric"

class(summary)
#> [1] "function"
```

Certain actions will change the class of an object. Suppose we try create a vector from the `x` object and a text string,

```
new_vector <- c(x, "Apple")

new_vector
#> [1] "1"     "2"     "3"     "1"     "3"     "25"    "Apple"

class(new_vector)
#> [1] "character"
```

By adding the word "Apple" to the vector, R changed the class from "numeric" to "character". All the numbers are enclosed in quotes: they are now character strings and cannot be used in calculations.

The most common class of data object we will use is the data frame. The data in midfieldr are stored as data frames, e.g.,

```
# examine another midfieldr data set
study_stickiness
#>        program            race    sex  ever  grad stick
#>         <char>          <char> <char> <int> <int> <num>
#>  1:      Civil           Asian Female    17    12  70.6
#>  2:      Civil           Black Female    54    28  51.9
#>  3:      Civil           White Female   329   232  70.5
#>  4:      Civil           Asian   Male    37    24  64.9
#>  5:      Civil           Black   Male    98    43  43.9
#> ---
#> 34: Mechanical Hispanic/Latinx   Male    76    47  61.8
#> 35: Mechanical   International   Male    37    19  51.4
#> 36: Mechanical Native American   Male    14     8  57.1
#> 37: Mechanical   Other/Unknown   Male    48    28  58.3
#> 38: Mechanical           White   Male  1940  1265  65.2

class(study_stickiness)
#> [1] "data.table" "data.frame"
```

- Six columns: program, race, sex, ever, grad, stick.
- Three columns are labeled `<char>` for character, categorical variables

- Two columns are labeled `<int>` for integer
- One column is labeled `<num>` for double precision

The additional class shown `data.table` is an augmented version of the base R `data.frame` class. When working with these objects you can use base R `data.frame` syntax or data.table syntax.

If you have a data.frame object that is not a data.table, e.g. the `airquality` data frame that comes with R

```
class(airquality)
#> [1] "data.frame"

head(airquality)
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6
```

You can convert it to data.table object with `as.data.table()` and assign it to

a slightly different name.

```
air_quality <- as.data.table(airquality)

class(air_quality)
#> [1] "data.table" "data.frame"

air_quality
#>      Ozone Solar.R  Wind  Temp Month   Day
#>      <int>   <int> <num> <int> <int> <int>
#>   1:    41     190   7.4    67     5     1
#>   2:    36     118   8.0    72     5     2
#>   3:    12     149  12.6    74     5     3
#>   4:    18     313  11.5    62     5     4
#>   5:    NA      NA  14.3    56     5     5
#>  ---
#> 149:    30     193   6.9    70     9    26
#> 150:    NA     145  13.2    77     9    27
#> 151:    14     191  14.3    75     9    28
#> 152:    18     131   8.0    76     9    29
#> 153:    20     223  11.5    68     9    30
```

The data frame as a whole has a class; so do the individual columns.

```
class(air_quality[, Ozone])
#> [1] "integer"

class(air_quality)
#> [1] "data.table" "data.frame"
```

4.8 R objects have structure

To see inside an object ask for its structure using the `str()` function.

```
str(x)
#>  num [1:6] 1 2 3 1 3 25

str(toy_term)
#> Classes 'data.table' and 'data.frame':   169 obs. of  6 variables:
#>  $ mcid       : chr  "MID25899243" "MID26319252" "MID25841465" "MID26560837" ...
#>  $ institution: chr  "Institution B" "Institution E" "Institution M" "Institution J
#>  $ term       : chr  "19943" "20021" "20023" "19981" ...
#>  $ cip6       : chr  "240102" "140801" "260101" "999999" ...
#>  $ level      : chr  "03 Junior" "04 Senior" "02 Sophomore" "02 Sophomore" ...
#>  $ hours_term : num  5 5 16 12 15 21 9 12 17 17 ...
#>  - attr(*, ".internal.selfref")=<externalptr>
```

```
str(airquality)
#> 'data.frame':    153 obs. of  6 variables:
#>  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
#>  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
#>  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
#>  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
#>  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
#>  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...

str(air_quality)
#> Classes 'data.table' and 'data.frame':   153 obs. of  6 variables:
#>  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
#>  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
#>  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
#>  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
#>  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
#>  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
#>  - attr(*, ".internal.selfref")=<externalptr>
```

4.9 R only does what you tell it

Healy [2019a] offers this advice for specific things to watch out for:

- Make sure parentheses are balanced—that every opening `(` has a corresponding closing `)`.

- Expect to make errors and don't worry when that happens. You won't break anything.
- Make sure you complete your expressions. If you see a `+` in the Console instead of the usual prompt `>`, R thinks your expression is incomplete. For example, if you type the following and try to run it,

```
str(air_quality
```

the output in your Console reports:

```
#> str(air_quality
+
```

The plus sign indicates that the expression is incomplete…in this case a missing closing parenthesis. To recover, hit `Esc` or `ctrl C`. Then correct the code.

As you work through the MIDFIELD tutorials, we'll introduce syntax by example from both base R and the data.table package. There can be subtle differences that we will attempt to avoid by consistent usage.

For example, to subset a column from a data frame but keep it as a column (not

a vector), the base R syntax is

```
# base R subset one column
airquality[, "Ozone", drop = FALSE]
#>     Ozone
#> 1      41
#> 2      36
#> 3      12
#> 4      18
#> 5      NA
#> 6      28
#> 7      23
#> 8      19
#> 9       8
#> 10     NA
#> ..., etc., truncated for brevity
```

In data.table syntax, the same operation is as follows. Note we are using the `air_quality` data.table object we created earlier.

```
# data.table extract one column
air_quality[, .(Ozone)]
#>      Ozone
#>      <int>
#>   1:    41
#>   2:    36
#>   3:    12
#>   4:    18
#>   5:    NA
#>  ---
#> 149:    30
#> 150:    NA
#> 151:    14
#> 152:    18
#> 153:    20
```

If we want the same information extracted as a vector, we would:

```
# base R subset one column as a vector
airquality[, "Ozone"]
#>   [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6
#>  [19]  30  11   1  11   4  32  NA  NA  NA  23  45 115  37  NA  NA  NA  NA  NA
#>  [37]  NA  29  NA  71  39  NA  NA  23  NA  NA  21  37  20  12  13  NA  NA  NA
#>  [55]  NA  NA  NA  NA  NA  NA  NA 135  49  32  NA  64  40  77  97  97  85  NA
#>  [73]  10  27  NA   7  48  35  61  79  63  16  NA  NA  80 108  20  52  82  50
#>  [91]  64  59  39   9  16  78  35  66 122  89 110  NA  NA  44  28  65  NA  22
#> [109]  59  23  31  44  21   9  NA  45 168  73  NA  76 118  84  85  96  78  73
#> [127]  91  47  32  20  23  21  24  44  21  28   9  13  46  18  13  24  16  13
```

```
#> [145]  23  36   7  14  30  NA  14  18  20

# data.table subset one column as a vector
air_quality[, Ozone]
#>   [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6
#>  [19]  30  11   1  11   4  32  NA  NA  NA  23  45 115  37  NA  NA  NA  NA  NA
#>  [37]  NA  29  NA  71  39  NA  NA  23  NA  NA  21  37  20  12  13  NA  NA  NA
#>  [55]  NA  NA  NA  NA  NA  NA  NA 135  49  32  NA  64  40  77  97  97  85  NA
#>  [73]  10  27  NA   7  48  35  61  79  63  16  NA  NA  80 108  20  52  82  50
#>  [91]  64  59  39   9  16  78  35  66 122  89 110  NA  NA  44  28  65  NA  22
#> [109]  59  23  31  44  21   9  NA  45 168  73  NA  76 118  84  85  96  78  73
#> [127]  91  47  32  20  23  21  24  44  21  28   9  13  46  18  13  24  16  13
#> [145]  23  36   7  14  30  NA  14  18  20
```

In each case, you will note that the square bracket operators include a comma `[i, j]`. In general, the comma separates row operations `i` from column operations `j`. When there is no row operation (as in the examples above), the `i` position is empty.

In data table syntax, we go one step further and add a grouping index, that is

```
DT[i, j, by]
```

This can be read as "Take DT, subset/reorder rows using `i`, then calculate `j`, grouped by `by`." You don't have to go into it any more deeply than this for now—we will illustrate it by example in the midfieldr tutorials.

4.10 To get help

Online resources

Tutorial

- Getting Started in R: Tinyverse Edition. Highly recommended. An 8-page introduction to R using data.table and ggplot2.

Online Q & A

- Stack Overfow R section. A question-and-answer site.

Cheat sheets. Compact (information dense) summaries of features.

- RStudio
- data.table
- ggplot2

Package main help

For a package's main help page, `help(package = "name_of_package")`, to obtain a list of all the functions (and possibly some data objects) in the package.

```
# type in the Console
help(package = "midfieldr")
```

In the Help pane, click through any of the links for details on the function.

Package vignettes

Some packages come with vignettes, articles explaining how to use the package for specific tasks. For example,

```
# type in the Console
browseVignettes(package = "data.table")
browseVignettes(package = "ggplot2")

# to see a listing of all vignettes in all your installed packages
browseVignettes()
```

Function help

A number of times so far we've shown how to get the help page for a function when we know its name, e.g.,

```
# type in the Console
? str()
```

If you recall only a part of the function name, use `apropos()` to search packages that are currently loaded. "Loading" a package is what we do when we use the `library()` function at the top of a script.

For example, midfieldr has functions with "timely" in their name because they involve timely completion. `apropos()` reports two functions:

```
# type in the Console
apropos("timely")
#> [1] "add_completion_timely" "add_timely_term"
```

Examples

There are often examples at the end of an R function help page. You can run them using the `example()` function. For example,

```
# base R function examples
example("mean")
#>
#> mean> x <- c(0:10, 50)
#>
#> mean> xm <- mean(x)
#>
#> mean> c(xm, mean(x, trim = 0.10))
#> [1] 8.75 5.50
```

try one of the midfieldr function examples,

```
# midfieldr function examples
example("add_institution")
#>
#> add_ns> # Extract a column of IDs from student
#> add_ns> id <- toy_student[, .(mcid)]
#>
#> add_ns> # Add institutions from term
#> add_ns> DT1 <- add_institution(id, midfield_term = toy_term)
#>
#> add_ns> head(DT1)
#>           mcid   institution
#>         <char>        <char>
#> 1: MID25783939 Institution M
#> 2: MID25784402 Institution M
#> 3: MID25805538 Institution M
#> 4: MID25808099 Institution M
#> 5: MID25816437 Institution M
#> 6: MID25826223 Institution M
#>
#> add_ns> nrow(DT1)
#> [1] 100
#>
#> add_ns> # Will overwrite institution column if present
#> add_ns> DT2 <- add_institution(DT1, midfield_term = toy_term)
#>
#> add_ns> head(DT2)
#>           mcid   institution
#>         <char>        <char>
#> 1: MID25783939 Institution M
#> 2: MID25784402 Institution M
#> 3: MID25805538 Institution M
#> 4: MID25808099 Institution M
#> 5: MID25816437 Institution M
#> 6: MID25826223 Institution M
#>
#> add_ns> nrow(DT2)
#> [1] 100
```

4.11 Keyboard shortcuts

If you are working in RStudio, you can see the menu of keyboard shortcuts using the menu *Tools > Keyboard Shortcuts Help*. The shortcuts we use regularly include

Windows / Linux	Action	Mac OS
`ctrl L`	Clear the RStudio Console	`ctrl L`
`ctrl shift C`	Comment/uncomment line(s)	`cmd shift C`
`ctrl X, C, V`	Cut, copy, paste	`cmd X, C, V`
`ctrl F`	Find in text	`cmd F`
`ctrl I`	Indent or re-indent lines	`cmd I`
`alt −`	Insert the assignment operator `<-`	`option −`
`ctrl alt B`	Run from beginning to line	`cmd option B`
`ctrl alt E`	Run from line to end	`cmd option E`
`ctrl Enter`	Run selected line(s)	`cmd Return`
`ctrl S`	Save	`cmd S`
`ctrl A`	Select all text	`cmd A`
`ctrl Z`	Undo	`cmd Z`

Lastly, any time you want a fresh start in your working environment,

- Use the pulldown menu *Session > Restart R*

4.12 Next steps

That concludes our brief introduction to R. To continue, return to your starting point (links below) and from there, select the next link in your progression.

- New to R?
- Familiar with R?
- After the workshop

top of page

Chapter 5

Graph basics

Decline by Randall Munroe (xkcd.com) is licensed under CC BY-NC 2.5

5.1 Introduction

This tutorial is an introduction to ggplot2 adapted from Chapter 3 from [Healy, 2019a]. If you already have R experience, you might still want to browse this section in case you find something new.

Prerequisites should be completed before proceeding. After that, the tutorial should take about an hour.

Create a new script for this tutorial.

- See Open an R script if you need a refresher on creating, saving, and running an R script.
- At the top of the script add a minimal header and install and load the packages indicated.

```
# Graph basics
# Name
# Date

# Packages used in this tutorial
library("midfieldr")
library("midfielddata")
library("data.table")
library("ggplot2")
library("gapminder")

# Optional code to control data.table printing
options(
```

```
  datatable.print.nrows = 10,
  datatable.print.topn = 5,
  datatable.print.class = TRUE
)

# Load midfielddata data sets to use later
data(student)
data(term)
```

If you get an error like this one after running the script,

```
Error in library("gapminder") : there is no package called 'gapminder'
```

then the package needs to be installed. If you need a refresher on installing packages, see Install CRAN packages. Once the missing package is installed, you can rerun the script.

Guidelines

- As you work through the tutorial, type a line or chunk of code then *File > Save* and run the script.
- Confirm that your result matches the tutorial result.
- The **exercises** give you chance to practice your new skills to learn by doing (but you knew that already)!

5.2 Expected data structure

Data for analysis and graphing are often laid out in "block record" or "long" form with every key variable and response variable in their own columns [Mount and Zumel, 2019]. Database designers call this a "denormalized" form; many R users would recognize it as the so-called "tidy" form [Wickham and Grolemund, 2017].

We use this form regularly for preparing data for graphing using the ggplot2 package. The gapminder data we're using in this tutorial is in block-record form. To view its help page, run

```
library("gapminder")
? gapminder
```

```
# Convert the data frame to a data.table structure
gapminder <- data.table(gapminder)
```

And we can just type its name to see a few rows. Note at the top of each column under the column name, the class of the variable is shown: factor `<fctr>`, integer `<int>`, and double-precision `<num>`.

```
gapminder
#>           country continent  year lifeExp      pop gdpPercap
#>            <fctr>    <fctr> <int>   <num>    <int>     <num>
#>    1: Afghanistan      Asia  1952  28.801  8425333  779.4453
#>    2: Afghanistan      Asia  1957  30.332  9240934  820.8530
#>    3: Afghanistan      Asia  1962  31.997 10267083  853.1007
#>    4: Afghanistan      Asia  1967  34.020 11537966  836.1971
#>    5: Afghanistan      Asia  1972  36.088 13079460  739.9811
#>   ---
#> 1700:    Zimbabwe    Africa  1987  62.351  9216418  706.1573
#> 1701:    Zimbabwe    Africa  1992  60.377 10704340  693.4208
#> 1702:    Zimbabwe    Africa  1997  46.809 11404948  792.4500
#> 1703:    Zimbabwe    Africa  2002  39.989 11926563  672.0386
#> 1704:    Zimbabwe    Africa  2007  43.487 12311143  469.7093
```

5.2.1 Exercise

- Examine the `student` data from midfielddata. (Type its name in the console.)
- How many variables? How many observations?
- How many of the variables are numeric? How many are character type?
- Is the data set in block-record form?
- Check your work by comparing your result to the `student` help page (link below).

Help pages for more information:

- `student`
- `gapminder`

5.3 Anatomy of a graph

`ggplot()` is a our basic plotting function. The `data = ...` argument assigns the data frame. The plot is empty because we haven't mapped the data to coordinates yet.

```
ggplot(data = gapminder)
```

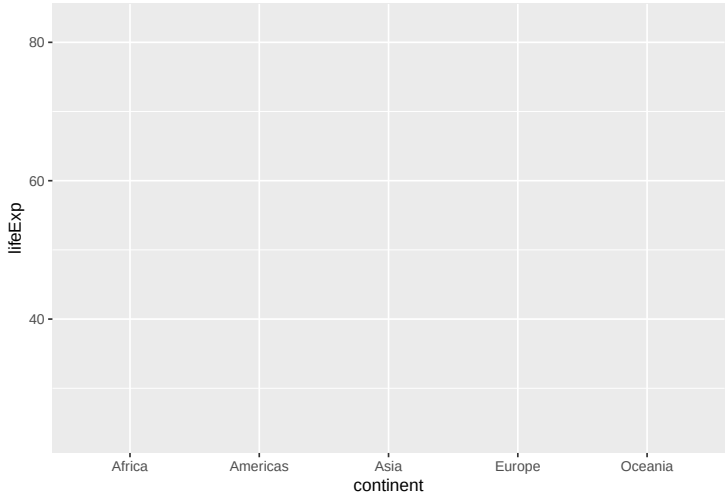
Next we use the mapping argument `mapping = aes(...)` to assign variables (column names) from the data frame to specific aesthetic properties of the graph such as the x-coordinate, the y-coordinate color, fill, etc.

Here we map continent (a categorical variable) to `x` and life expectancy (a quantitative variable) to `y`. To reduce the number of times we repeat lines of code, we can assign a name (`life_gdp`) to the empty graph to which we can add layers later.

```
# Demonstrate aesthetic mapping
life_exp <- ggplot(data = gapminder, mapping = aes(x = continent, y = lifeExp))
```

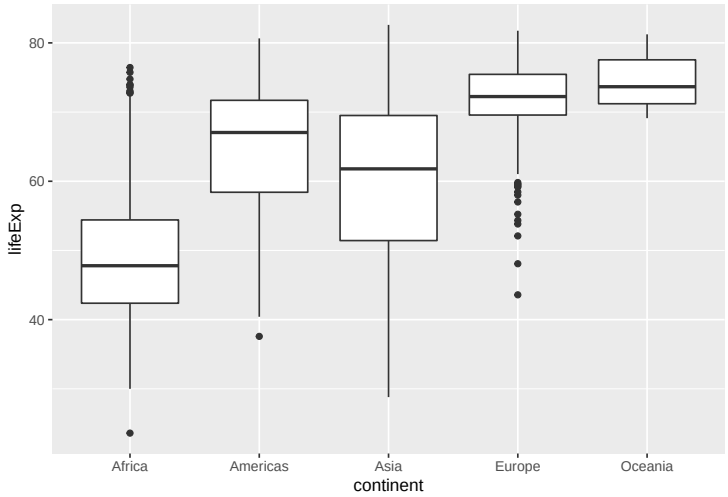
If we print the graph by typing the name of the graph object (everything in R is an object), we get a graph with a range on each axis (from the mapping) but no data shown. We haven't specified the type of visual encoding we want.

```
# Examine the result
life_exp
```



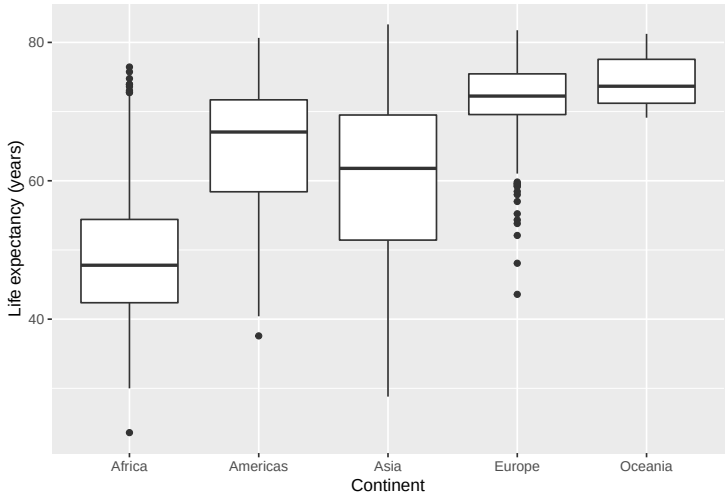
A box-and-whisker plot (or box plot) is designed for displaying the distribution of a single quantitative variable. The visual encoding is specified using the `geom_boxplot()` layer, where a "geom" is a geometric object. The `geom_boxplot()` function requires the quantitative variable assigned to `y` and the categorical variable (if any) to `x`.

```
# Demonstrate adding a geometric object
life_exp <- life_exp +
  geom_boxplot()

# Examine the result
life_exp
```



Notice that the default axis labels are the variables names from the data frame. We can edit those with another layer

```
# Demonstrate editing axis labels
life_exp <- life_exp +
  labs(x = "Continent", y = "Life expectancy (years)")

# Examine the result
life_exp
```



You can always write all the layers in one code chunk and obtain the same result.

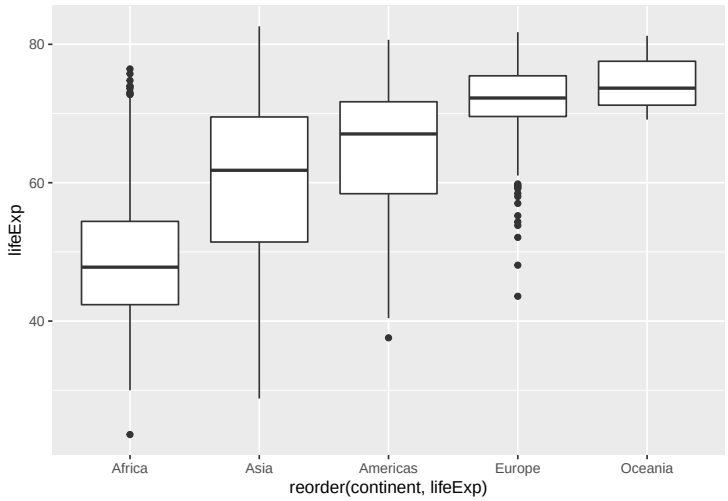
```
# Demonstrate all layers in one code chunk
p <-  ggplot(data = gapminder, mapping = aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  labs(x = "Continent", y = "Life expectancy (years)")
```

To reduce typing, the first two arguments `data` and `mapping` are often used without naming them explicitly, e.g.,

```
# Demonstrate implicit data and mapping arguments
life_exp <-  ggplot(gapminder, aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  labs(x = "Continent", y = "Life expectancy (years)")
```

Next, we often want the categorical variable ordered by the quantitative variable instead of alphabetically. Because `continent` is a factor, we can use the `reorder()` function inside the `aes()` argument.

```
# Demonstrate reordering a categorical variable
life_exp +
  aes(x = reorder(continent, lifeExp), y = lifeExp)
```

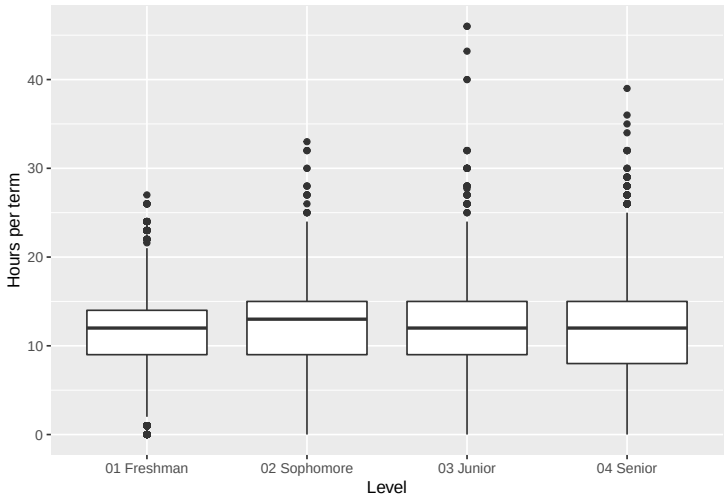


Summary. The basics steps for building up the layers of any graph consist of,

- assign the data frame
- map variables (columns names) to aesthetic properties
- choose geoms
- adjust scales, labels, ordering, etc.

5.3.1 Exercise

- Examine the `term` data set from midfielddata.
- Create a boxplot of the hours per term quantity conditioned by the student level.
- What is the rational for leaving the categorical variable in its native order?
- Check your work by comparing your result to the graph below.



Help pages for more information:

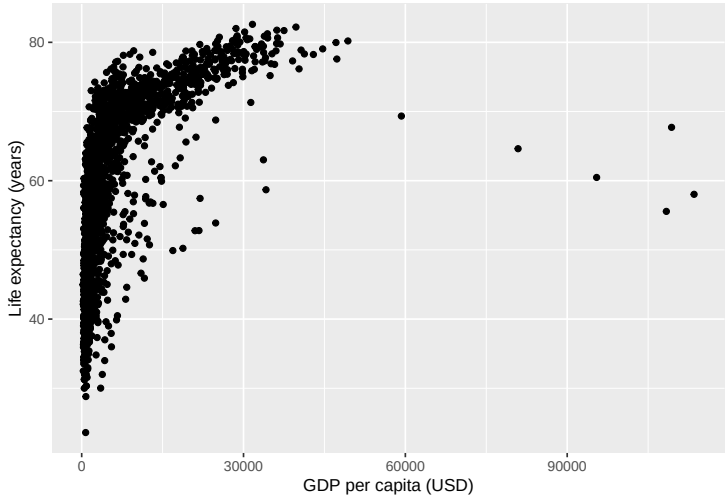
- `term`
- `aes()`

- `geom_boxplot()`
- `geom_labs()`

5.4 Layer: points

A two-dimensional scatterplot reveals the strength of the relationship between two quantitative variables. The ggplot geom for scatterplots is `geom_point()`. To illustrate a scatterplot, we graph life expectancy as a function of GDP.

```
life_gdp <- ggplot(gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  labs(x = "GDP per capita (USD)", y = "Life expectancy (years)")

life_gdp
```



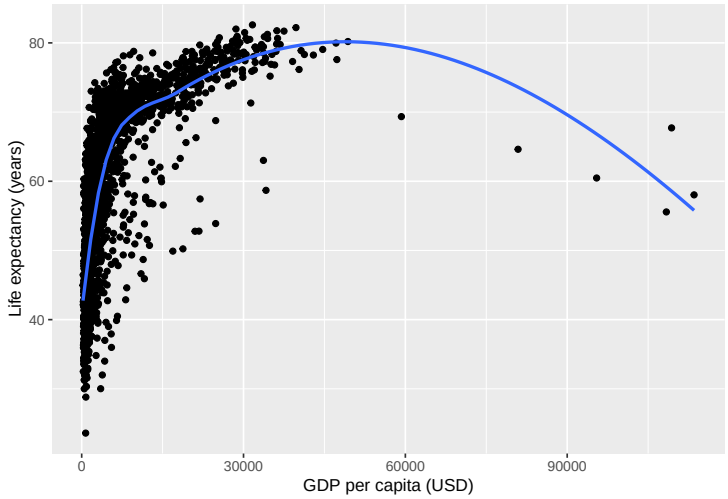
Help pages for more information:

- `geom_point()`

5.5 Layer: smooth fit

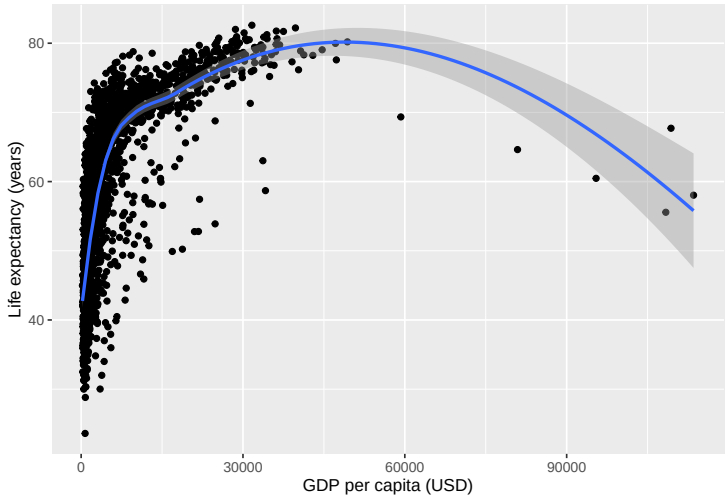
Suppose you wanted a smooth fit curve, not necessarily linear. Add a `geom_smooth()` layer. The name *loess* (pronounced like the proper name Lois) is a nonparametric curve-fitting method based on *local regression*.

```
life_gdp +
  geom_smooth(method = "loess", se = FALSE)
```



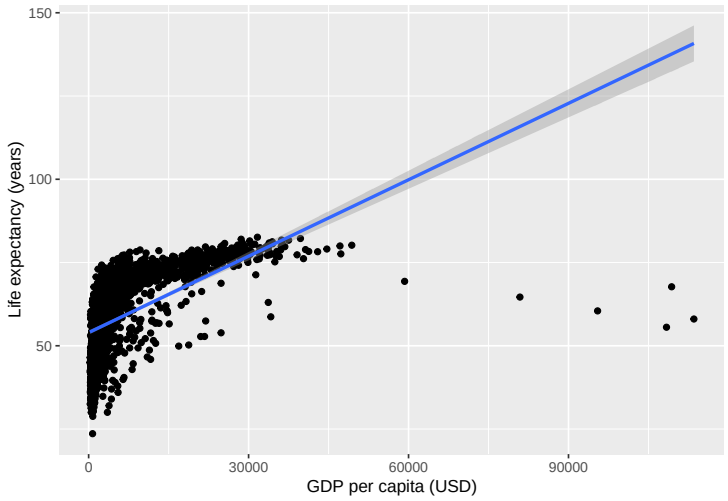
The `se` argument controls whether or not the confidence interval is displayed. Setting `se = TRUE` yields,

```
life_gdp +
  geom_smooth(method = "loess", se = TRUE)
```



For a linear-fit layer, we add a layer with `method` set to `lm` (short for linear model). The linear fit is not particularly good in this case, but now you know how to do one.

```
life_gdp +
  geom_smooth(method = "lm", se = TRUE)
```



Help pages for more information:

- geom_smooth()

5.5.1 Exercise

In this exercise, we do some data manipulation without explaining the steps. We cover such details in the Data basics tutorial. Here we are focusing on constructing graphs.

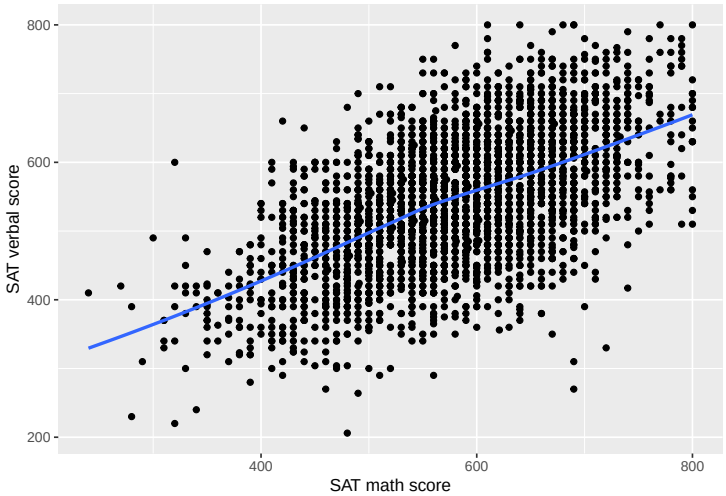
- Use the following code chunk to create a random sample of the `student` data set with columns for SAT math and verbal scores.

```
# We only need two columns for the plot
cols_we_want <- c("sat_math", "sat_verbal")
sat <- student[, ..cols_we_want]

# Before sampling, we remove NA values
sat <- na.omit(sat)

# We sample to reduce computational time
sat <- sat[sample(nrow(sat), 3000)]
```

- Use the resulting `dtf` data frame and create a scatterplot of verbal scores `sat_verbal` as a function of math scores `sat_math`.

- Add a loess fit.

- Check your work by comparing your result to the graph below.



5.6 Layer: scale

We have orders of magnitude differences in the GDP per capita variable. To confirm, we can create a `summary()` of the `gdpPercap` variable. The output shows that the minimum is 241, the median 3532, and the maximum 113,523.

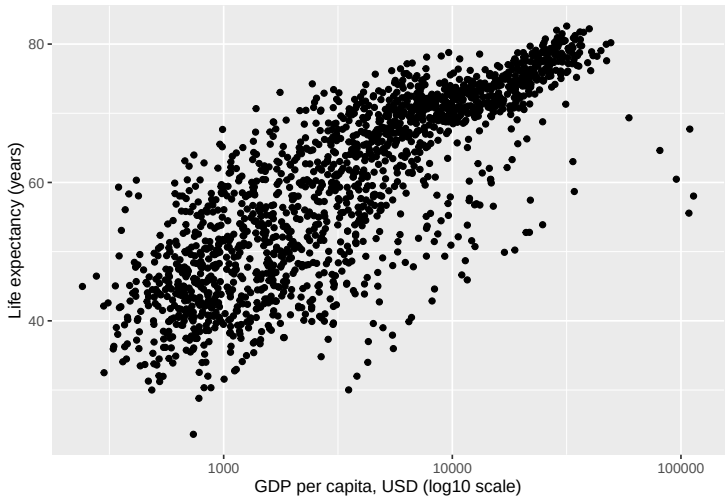
```
# statistical summary of one variable
summary(gapminder[, gdpPercap])
#>     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
#>    241.2   1202.1   3531.8   7215.3   9325.5 113523.1
```

In exploring a graph like this, it might be useful to add a layer that changes the horizontal scale to a log-base-10 scale.

```
life_gdp <- life_gdp +
  scale_x_continuous(trans = "log10")
```

Update the axis labels,

```
life_gdp <- life_gdp +
  labs(x = "GDP per capita, USD (log10 scale)",
       y = "Life expectancy (years)")

life_gdp # display the graph
```



In summary, all the layers could have been be coded at once, for example,

```
ggplot(gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_continuous(trans = "log10") +
  labs(x = "GDP per capita, USD (log10 scale)",
       y = "Life expectancy (years)")
```

With all the layers in one place, we can see that we've coded all the basic steps, that is,

- assign the data frame
- map variables (columns names) to aesthetic properties
- choose geoms
- adjust scales, labels, ordering, etc.

5.6.1 Exercise

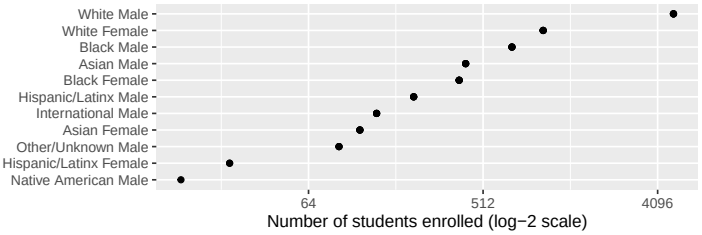
Again we do some data manipulation without explaining the steps. We cover such details in the Data basics tutorial. Here we are focusing on constructing graphs.

We often use graphs to show numbers of students with a log-scale to compare totals that differ by orders of magnitude.

- Use the following code chunk to count the number students ever enrolled in the sample data by race/ethnicity/sex.

```
# Access a data set included with midfieldr
dtf <- copy(study_stickiness)

# Create a combined categorical column
dtf[, race_sex := paste(race, sex)]

# Sum the number of students by this category
dtf[, ever := sum(ever), by = "race_sex"]

# Factor class enables reordering
dtf[, race_sex := factor(race_sex)]
```

- Use the resulting `dtf` data frame to graph `ever` on the x-axis with `race_sex` on the y-axis.
- Reorder `race_sex` by `ever`.
- Use a log-base-2 scale.
- You can omit the y-axis label by setting `y = ""` in the `labs()` argument.
- Check your work by comparing your result to the graph below.



Help pages for more information:

- scale_x_continuous()

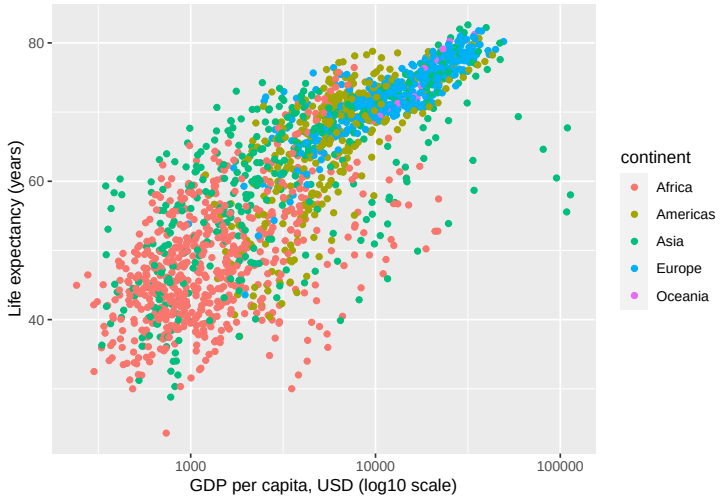
5.7 Mapping columns to aesthetics

Mappings in the `aes()` function of `ggplot()` can involve the names of variables (column s) only. So far, the only mappings we've used are from column names to an x or y aesthetic.

Another useful mapping is from a column name to the `color` argument, which then separates the data by the values of the categorical variable selected and automatically creates the appropriate legend.

Here we map the `continent` column to the `color` aesthetic, adding a third data variable to the display.

```
ggplot(gapminder, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_continuous(trans = "log10") +
  labs(x = "GDP per capita, USD (log10 scale)",
       y = "Life expectancy (years)")
```

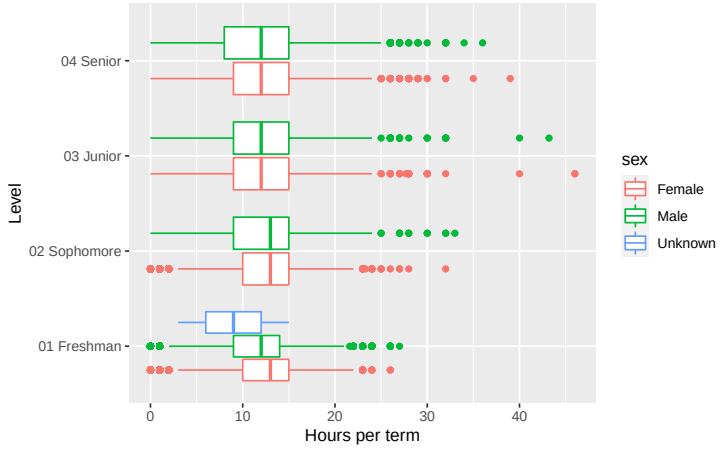


5.7.1 Exercise

Use the following code chunk to construct a data frame with hours per term and level from `term` merged with `sex` data from `student`. Again, we leave explanations of the data manipulation to the Data basics tutorial.

```
dtf <- add_race_sex(term,
                    midfield_student = student)
cols_we_want <- c("mcid", "term", "hours_term", "level", "sex")
dtf <- dtf[, ..cols_we_want]
dtf <- unique(dtf)
```

- Use the resulting `dtf` data frame to a boxplot of hours per term as a function of level like we did earlier.
- Add a third column name to `aes()` to add `sex` by color to the graph.
- Add the `coord_flip()` layer to your graph to swap the position of the x, y coordinates to obtain a horizontal boxplot.
- Check your work by comparing your result to the graph below.



Help pages for more information:

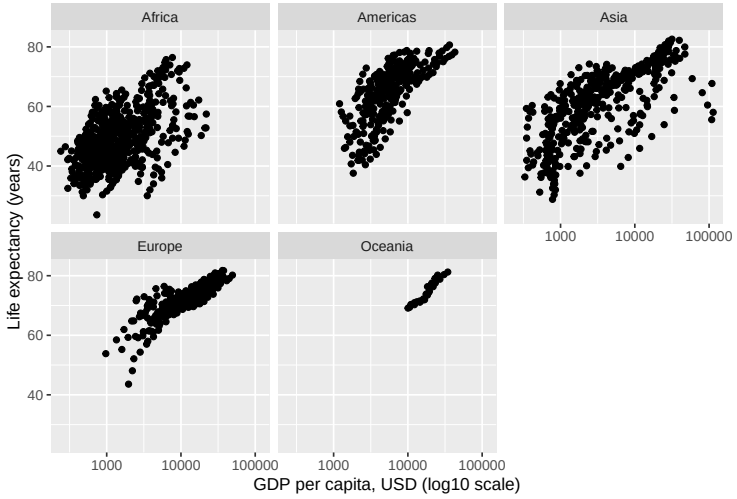
- `add_race_sex()`
- `coord_flip()`

5.8 Layer: facets

In the earlier graph where we mapped continent to color, there was a lot of overprinting, making it difficult to compare the continents. Instead of using color to distinguish the continents, we can plot in different panels by continent.

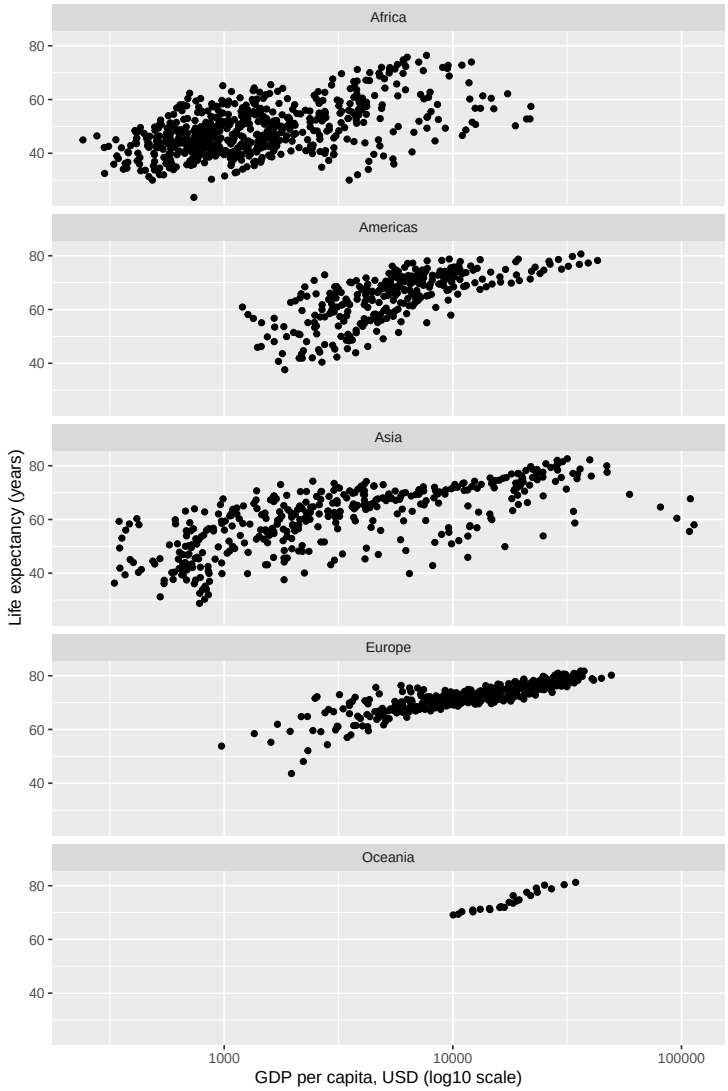
The `facet_wrap()` layer separates the data into different panels (or facets). Like the `aes()` mapping, `facet_wrap()` is applied to a variable (column name) in the data frame.

```
life_gdp <- life_gdp +
  facet_wrap(facets = vars(continent))

life_gdp # print the graph
```



Comparisons are facilitated by having the facets appear in one column, by using the `ncol` argument of `facet_wrap()`.

```
life_gdp <- life_gdp + facet_wrap(facets = vars(continent), ncol = 1)

life_gdp # print the graph
```



In a faceted display, all panels have identical scales (the default) to facilitate comparison. Again, all the layers could have been be coded at once, for example,

```
ggplot(gapminder, aes(x = gdpPercap, y = lifeExp)) +
  facet_wrap(facets = vars(continent), ncol = 1) +
  geom_point() +
  scale_x_continuous(trans = "log10") +
  labs(x = "GDP per capita, USD (log10 scale)",
       y = "Life expectancy (years)")
```

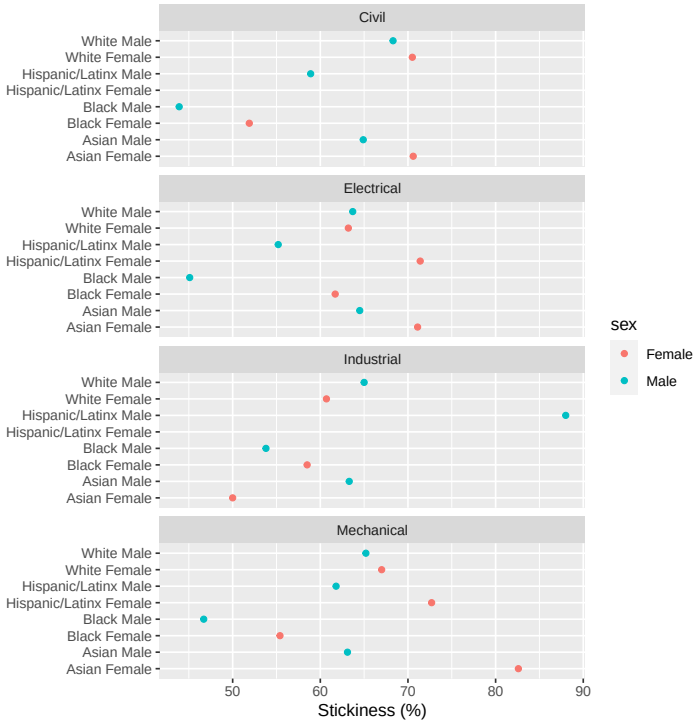
5.8.1 Exercise

Use the following code chunk to construct a data frame of program stickiness by program, race/ethnicity, and sex.

```
# Access a data set included with midfieldr
stickiness <- copy(study_stickiness)

# Create a combined categorical column
stickiness[, race_sex := paste(race, sex)]

# Factor class enables reordering
stickiness[, race_sex := factor(race_sex)]
stickiness[, program := factor(program)]

# Omit ambiguous race/ethnicities and small populations
stickiness <- stickiness[!race_sex %ilike% c("Native|International|Other")]
```

- Use the resulting `dtf` data frame plot stickiness (x-axis) as a function of race/ethnicity/sex (y-axis) and faceted by program.

- When that graph seems OK, add a third column name to `aes()` to add `sex` by color to the graph.
- Check your work by comparing your result to the graph below.



Help pages for more information:

- facet_wrap()

5.9 Ordering panels and rows

Panels (facets) by default are nearly always ordered alphabetically. In most cases, ordering the panels by the data improves the display. Earlier, we used `reorder()` to order a set of boxplots. This function operates correctly on "factor" variables only.

A factor is special data structure in R for categorical variables. In a factor, the levels of the category—typically character strings—are known and fixed. However, factors are stored internally as integers—a critical design tool for meaningfully ordering the rows and panels of a display involving categorical variables.

In the gapminder data, the `continent` column is already a factors.

```
gapminder
#>           country continent  year lifeExp      pop gdpPercap
#>            <fctr>    <fctr> <int>   <num>    <int>     <num>
#>    1: Afghanistan      Asia  1952  28.801  8425333  779.4453
```

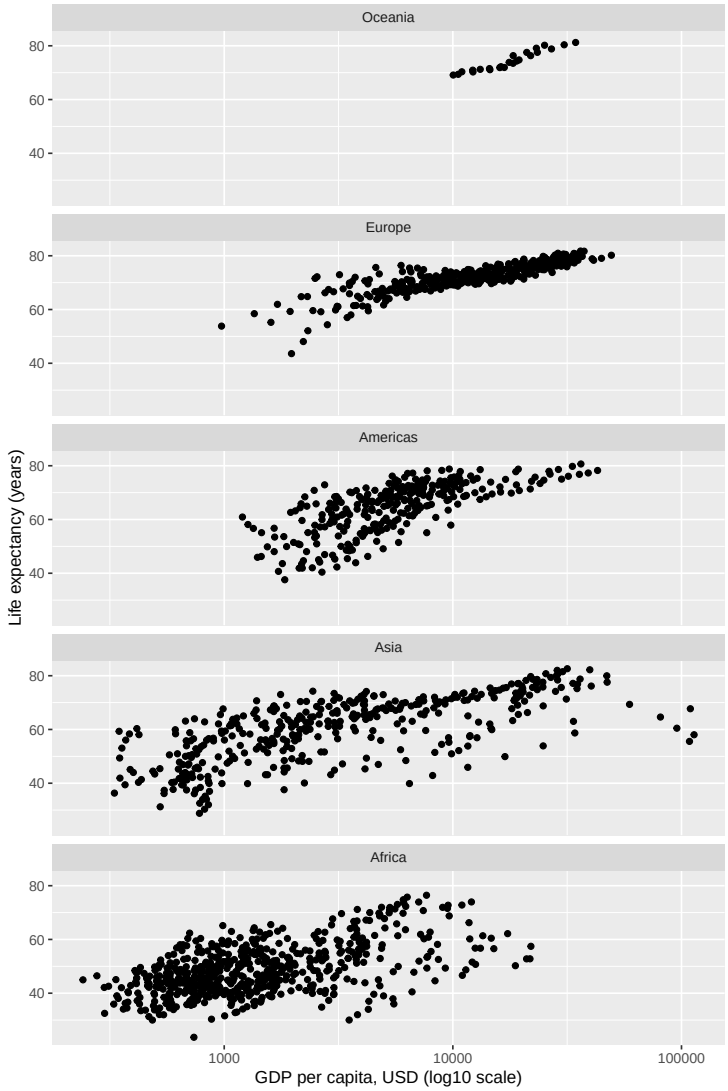
```
#>    2: Afghanistan      Asia  1957  30.332  9240934  820.8530
#>    3: Afghanistan      Asia  1962  31.997 10267083  853.1007
#>    4: Afghanistan      Asia  1967  34.020 11537966  836.1971
#>    5: Afghanistan      Asia  1972  36.088 13079460  739.9811
#>   ---
#> 1700:    Zimbabwe    Africa  1987  62.351  9216418  706.1573
#> 1701:    Zimbabwe    Africa  1992  60.377 10704340  693.4208
#> 1702:    Zimbabwe    Africa  1997  46.809 11404948  792.4500
#> 1703:    Zimbabwe    Africa  2002  39.989 11926563  672.0386
#> 1704:    Zimbabwe    Africa  2007  43.487 12311143  469.7093
```

We can reorder the levels of the factor before graphing as follows,

```
gapminder[, continent := reorder(continent, lifeExp)]
```

Then graph using much the same code chunk as before with one addition. We add the `as.table = FALSE` argument to the `facet_wrap()` function. "Table-order" of panels is increasing from top to bottom; "graph-order" increases (like a graph scale) from bottom to top.

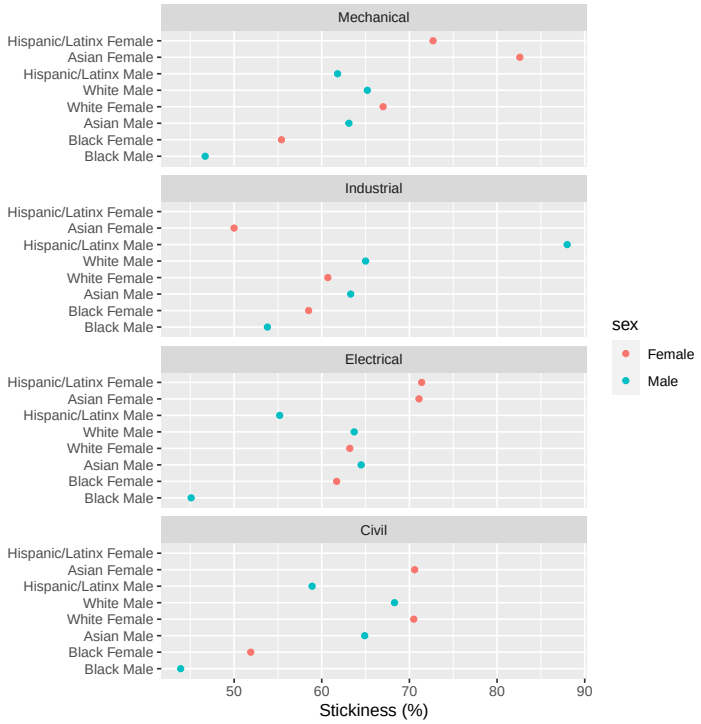
```
ggplot(gapminder, aes(x = gdpPercap, y = lifeExp)) +
  facet_wrap(facets = vars(continent), ncol = 1, as.table = FALSE) +
  geom_point() +
  scale_x_continuous(trans = "log10") +
  labs(x = "GDP per capita, USD (log10 scale)",
       y = "Life expectancy (years)")
```



5.9.1 Exercise

- Continue using the `stickiness` data frame from the previous section.

- Order the panels by the stickiness variable.
- Order race/ethnicity/sex by the stickiness variable.
- Check your work by comparing your result to the graph below.



Help pages for more information:

- reorder()

top of page

Chapter 6

Data basics

6.1 Introduction

This tutorial is an introduction to ...

If you already have R experience, you might still want to browse this section in case you find something new.

Prerequisites should be completed before proceeding. After that, the tutorial should take about an hour.

Create a new script for this tutorial.

- See Open an R script if you need a refresher on creating, saving, and running an R script.
- At the top of the script add a minimal header and install and load the packages indicated.

```
# Data basics
# name
# date

library("midfieldr")
library("data.table")
library("ggplot2")
library("gapminder")

# Optional code to control data.table printing
options(
  datatable.print.nrows = 10,
  datatable.print.topn = 5,
  datatable.print.class = TRUE
```

```
)
```

If you get an error like this one after running the script,

```
Error in library("gapminder") : there is no package called 'gapminder'
```

then the package needs to be installed. If you need a refresher on installing packages, see Install CRAN packages. Once the missing package is installed, you can rerun the script.

Guidelines

- As you work through the tutorial, type a line or chunk of code then *File > Save* and run the script.
- Confirm that your result matches the tutorial result.
- **Your turn** exercises give you chance to devise your own examples and check them out. You learn by doing (but you knew that already)!

top of page

Chapter 7

Designing effective displays

7.1 Stuff

7.2 More stuff

top of page

Bibliography

The Comprehensive R Archive Network, 2018-04-22. URL https://cran.r-project.org/.

Stephen Cass. Top programming languages. *IEEE Spectrum*, July 22 2020. URL https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020.

Matt Dowle and Arun Srinivasan. *data.table: Extension of 'data.frame'*, 2021. URL https://CRAN.R-project.org/package=data.table. R package version 1.14.0.

Kiernan Healy. *Data Visualization: A Practical Introduction*. Princeton University Press, Princeton, NJ, 2019a. URL https://kieranhealy.org/publications/dataviz/.

Kiernan Healy. *Get Started*, pages 32–53. Princeton University Press, Princeton, NJ, 2019b. URL https://kieranhealy.org/publications/dataviz/.

Norm Matloff. fasteR: Fast Lane to Learning R!, 2019. URL https://github.com/matloff/fasteR.

Steve McConnell. *Code Complete*. Microsoft Press, Redmond, WA, 2 edition, 2004.

John Mount and Nina Zumel. Coordinatized data: A fluid data specification, 2019. URL http://winvector.github.io/FluidData/RowsAndColumns.html.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021. URL https://www.R-project.org/.

Jenny Richmond. Basic Basics Lesson 1: An opinionated tour of RStudio, 2018. URL https://youtu.be/kfcX5DEMAp4.

RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA, 2016. URL http://www.rstudio.com/.

Hadley Wickham. *Advanced R*. Chapman & Hall/CRC The R Series. Taylor & Francis, 2014. ISBN 9781466586963.

Hadley Wickham. The tidyverse style guide, 2019. URL https://style.tidyverse.org/.

Hadley Wickham and Garrett Grolemund. *R for Data Science*. O'Reilly Media, Inc., Sebastopol, CA, 2017. URL https://r4ds.had.co.nz/.