Week-1: Introduction

NM2207: Computational Media Literacy

Narayani Vedam, Ph.D.

Department of Communications and New Media

NUS National University of Singapore | Faculty of Arts & Social Sciences

This week

This week

I. Introduction to people (click here)

II. Introduction to the course (click here)

III. Preliminaries (click here)

IV. Computational thinking (click here)

V. R Studio (click here)

VI. R Markdown (click here)

I. Meet the people

About me

> I am Narayani Vedam, you can address me by my initials, NV, for brevity's sake 😂 !
>
> You can learn more about me and my work here

Academic Background

1. Postdoctoral Fellow, Department of Communications and New Media
2. Ph.D., Department of Aerospace Engineering, The Indian Institute of Science, Bengaluru
3. M.S., Department of Electrical and Computer Science, Texas A&M University, College Station

Research Interests

1. Human behaviour on online social platforms
2. Multi-agent systems and controls

Know your Tutors

Aaron Liu (Wednesdays)

Hello!

- I am a Year 4 major in **Applied Mathematics**.
- My interests lie in quantitative finance, especially financial risk.
- I just finished my summer internship in **Hong Kong at ASTRI** doing Fintech research in big data and artificial intelligence.
- I also did a winter exchange at **Yonsei University in South Korea** and am seen wearing the jacket below!

Know your Tutors

Bebin Joseph (Mondays)

Hello, I took NM2207 the previous semester and I am super excited to meet y'all!

- I am in Year 3 majoring in Economics.
- I used to major in **Political Science**, and I switched over to **Economics** last year, so a lot of my academic interests lies in the intersection of economics and politics.
- Lastly, some of you might feel stressed about coding. As your fellow CHS/FASS student, I understand the concern and I want to assure you that you can do this!

Know your Tutors

Long Ngyuen Tan (Mondays)

Hello!

- I major in **Computer Science** & **Business**
- My research interests are Artificial Intelligence, Optimization
- I have experience in the Healthcare & Finance industry
- You could learn more about me here

Know your Tutors

Naman Agrawal (Wednesdays)

Hello!

- I am a third-year undergraduate student at the National University of Singapore.
- I major in **Data Science and Analytics** and **Economics** with a minor in Computer Science
- I enjoy experimenting with audio signal processing, 3D computer vision, and econometric modelling to develop novel insights.
- I'm currently working on developing complex-valued convolutional neural networks for applications in musical engineering.

II. All about the course

About the course: background

> Course is for **non-programmers**
>
> Literacy through **hands-on activity**
>
> Learning to **code** in R

Why learn coding?

Many real-world jobs rely on programming;

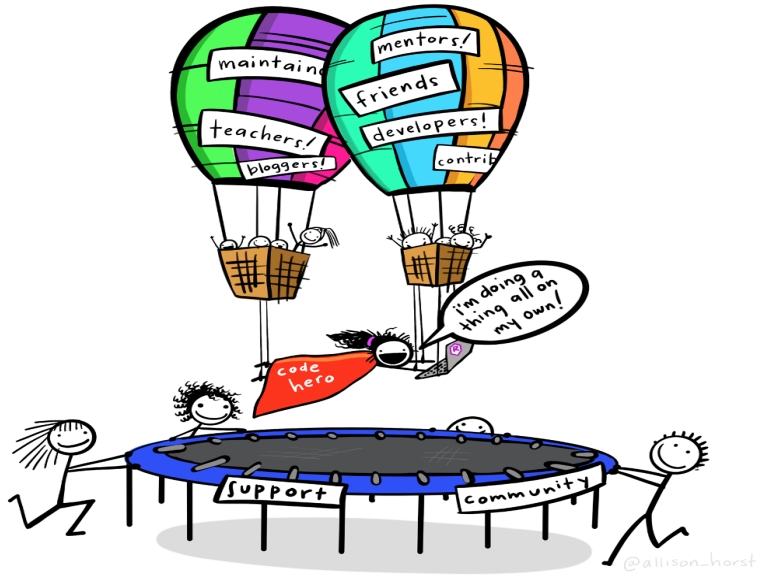
- to engage with large-scale data
- to complete statistical analyses

About the course: background

> Course is for **non-programmers**
>
> Literacy through **hands-on activity**
>
> Learning to **code** in R

Why R?

- Great data manipulation and visualization suite
- Comprehensive statistical packages
- Has a vibrant online community
- **Easiest** language for beginners
- Open source



Source: https://allisonhorst.com/everything-else

About the course: contents

Part-1: Data

- **Week-1:** Introduction to NM2207
- **Week-2:** Introduction to data & visualization
- **Week-3:** Variables & its types
- **Week-4:** Manipulating data
- **Week-5:** Functions
  - **Alert:** Introduction to github (not graded)
- **Week-6:** Iterations
  - **Alert:** Creating a web portfolio (not graded)

Part-2: Visualization

- **Week-7:** Visualization with `ggplot2`
  - **Alert:** Work on project proposal!!
- **Week-8:** Visualization with `Shiny`
- **Week-9:** Exploratory data analyses
  - **Submission alert:** Project proposal!!
- **Week-10 onward:** Practice project(s)

About the course: components

> 4-MC, Workload: approx. 10hrs weekly

1. Video "lectures"

- Explanatory videos of various concepts
- Code along with videos (templates will be provided)
- Videos to be watched before attending classes
- **Average time:** 4hrs

About the course: components

> 4-MC, Workload: approx. 10hrs weekly

2. Classroom "challenges"

- Recommend to work in **GROUPS**
  - Group sizes can be 2/3
- Questions can be,
  - Coding, narrations, MCQs
- Tutors will be present to guide
- Solution Templates in R Markdown for submissions will be provided
- **Average time:** 3hrs

1. Video "lectures"

About the course: components

> 4-MC, Workload: approx. 10hrs weekly

3. Final Project

A data story implementing concepts learned in class + new concepts you self-learn

a. It could be a webpage;

- data visualization conveying a story
- Example: https://hedonometer.org/

b. It could be a game;

- an app that let's the users interact, score points, chat, etc.
- Example: https://shiny.posit.co/r/gallery/miscellaneous/hex-memory/

1. Video "lectures"

2. Classroom "challenges"

An instance of a data story

1. Ask a question

2. Use data to answer

RESEARCH ARTICLE | PSYCHOLOGICAL AND COGNITIVE SCIENCES |

The emotional and mental health impact of the murder of George Floyd on the US population

Johannes C. Eichstaedt, Garrick T. Sherman, Salvatore Giorgi, +3, and Sharath Chandra Guntuku

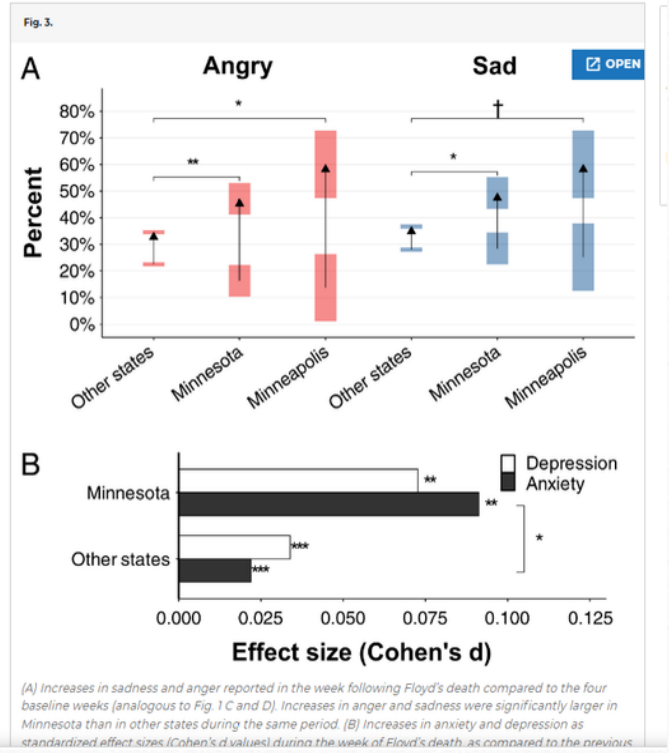
Authors Info & Affiliations

Edited by Douglas S. Massey, Princeton University, Princeton, NJ, and approved July 21, 2021 (received for review June 3, 2021)

September 20, 2021 | 118 (39) e2109139118 | https://doi.org/10.1073/pnas.2109139118

THIS ARTICLE HAS BEEN UPDATED

https://www.pnas.org/doi/10.1073/pnas.2109139118



(A) Increases in sadness and anger reported in the week following Floyd's death compared to the four baseline weeks (analogous to Fig. 1 C and D). Increases in anger and sadness were significantly larger in Minnesota than in other states during the same period. (B) Increases in anxiety and depression as standardized effect sizes (Cohen's d values) during the week of Floyd's death, as compared to the previous

About the course: submission policy

1. Challenges

- Complete Week N's challenge by the end of Week N's tutorial;
    - Inform the TA and submit by,
        - the end of class for 100% grades
        - class-day midnight for 50% grades

About the course: submission policy

2. Weekly assignments (if any)

- On time submissions, 100% grades
- Late submissions upon notifying TA, 50% grades

1. Challenges (Attendance/Participation)

About the course: submission policy

3. Quiz

- Only one quiz during **Week 6**

1. Challenges (Attendance/Participation)

2. Weekly assignments (if any)

About the course: submission policy

4. Final Project

- **Week 9:** Intermediate submission
  - Proposal with data sources and plan for data preparation
- **Week 10 onward:** Progress report
  - Write-up on the progress of the project
  - Highlight *show-stoppers*
- **Week 13:** Final project
  - A working website conveying the story
  - Documentation of the idea and execution

All project submissions are expected on friday of the respective week by 17:00hrs

1. Challenges (Attendance/Participation)

2. Weekly assignments (if any)

3. Quiz

About the course: evaluation

1. Tutorial/Participation: 30%

- attendance,
- preparation,
- and other evidence

About the course: evaluation

2. Quiz: 30%

- only one quiz
- will be in week-6, includes syllabus of weeks 1-6

1. Tutorial/Participation: 30%

About the course: evaluation

3. Project: 40%

- Final write-up: 25%
  - Background/Approach/Final outcome
- Demonstrated understanding: 20%
- Complexity: 15%
- Demonstrated learning: 15%
- Diary entries (weeks 9, 10, 11, 12): 10%
- Creativity: 10%
- Works well: 5%

1. Tutorial/Participation: 30%

2. Quiz: 30%

How to ace?

> Like with everything else in life, practice helps!

- Practice regularly
- Use Google and the online resources extensively
- Work with a friend(s)
- Do not hesitate to **ASK** questions
  - If you are shy to do so in person, use Canvas
  - Every week a new discussion thread will be initiated
  - Feel free to post questions, however insignificant you may think!
  - Please check for repetitions before posting

Learning outcomes

a. Mine insights from data-sets of interest

- Involves the ability to curate, manipulate, and analyze data using R

b. Visualize useful insights through plots and dashboards

c. Communicate the insights more comprehensively through an interactive website using R Markdown and Shiny

- Without the need to learn core HTML5 technologies

III. Preliminaries

Files

> They are objects on the computer

- They store data or commands for a computer program
  - They could be an application/data/system file
  - Extensions are used to recognise the file type
  - Examples: .docx, .xlsx, .txt, .jpeg, .png, .csv, *etc*.
- They are created using a computer software program
  - Examples:
    - To create a text file, we use a text editor
    - To create an image file, we use an image editor
    - To create a document, we use a document processor
- They are stored in the memory of the computer

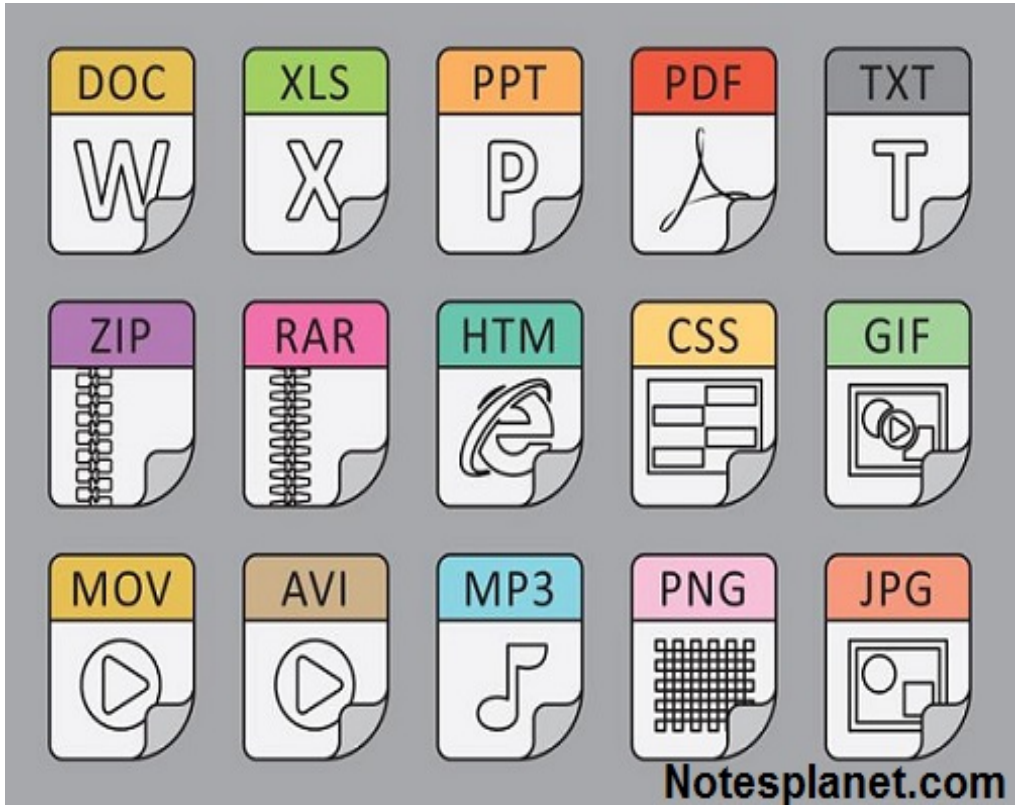


Figure: Different types of files and their extensions

Folders

- Folders can contain file(s) or other folders
- They can be thought of as storage boxes/organisers

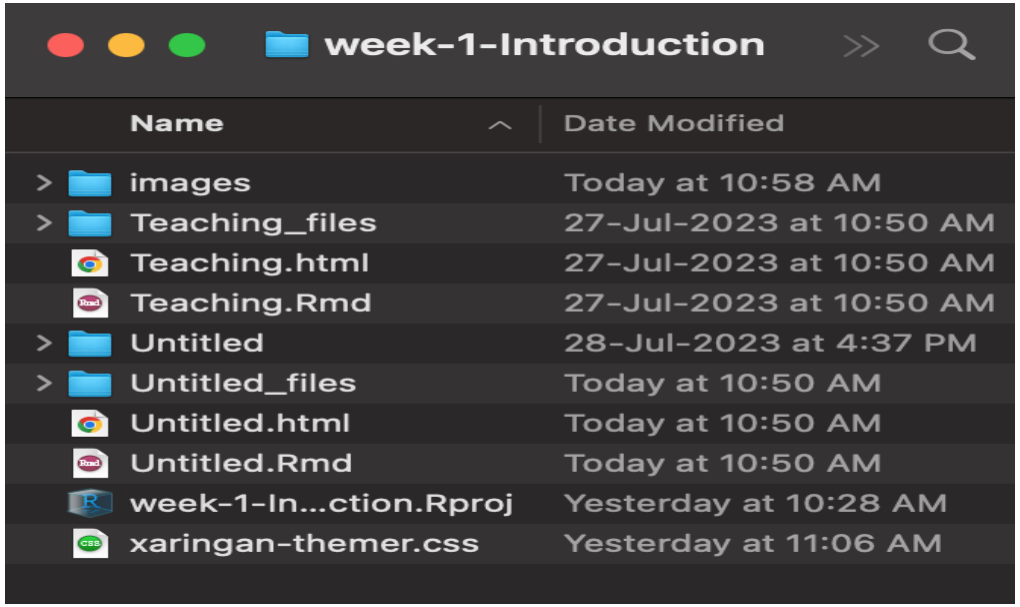


Figure: A folder in macOS

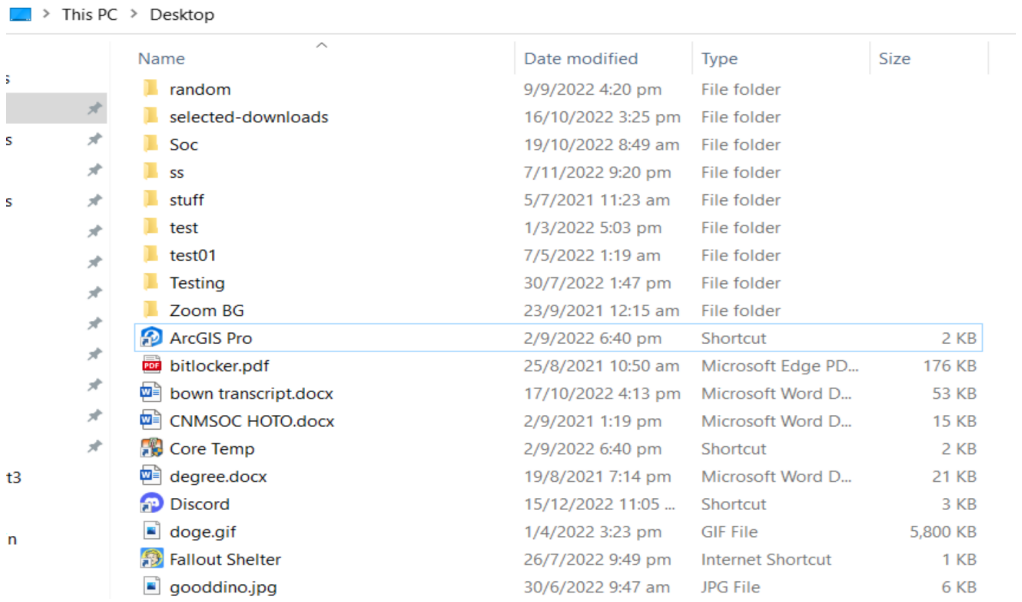


Figure: A folder in Windows OS

Paths

> They are the addresses to different files or folders

- Examples
  - `C:/Users/nvedam` path to a folder named `nvedam` on a Windows machine
  - `/Users/nvedam` path to a folder named `nvedam` on a macbook
  - `/Users/nvedam/example.txt` path to a file named `example.txt` on a macbook
  - `https://www.w3schools.com/images/picture.jpg` it could also be a path to files on the internet

IV. Computational thinking

What does it entail?

Programmers use the computer to solve problems. This includes,

a. Breaking down the problem into solvable steps

b. Identifying the inputs and outputs

c. Identifying the intermediate operations

d. Writing down the steps identified (**Algorithm!**)

- flowcharts
- pseudocode

Flowchart

Some guidelines for developing flowcharts,

- It can have only one start and stop symbol
- General flow is either,
  - Top to bottom, or
  - Left to right
- Arrows should not cross each other

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Figure: Flowchart symbols and their meanings

Flowchart: Example 1

An algorithm to add two numbers provided by the user and print the result;

- **Inputs:** `num1`, `num2`
- **Outputs:** `sum`
- **Operations:** addition, print

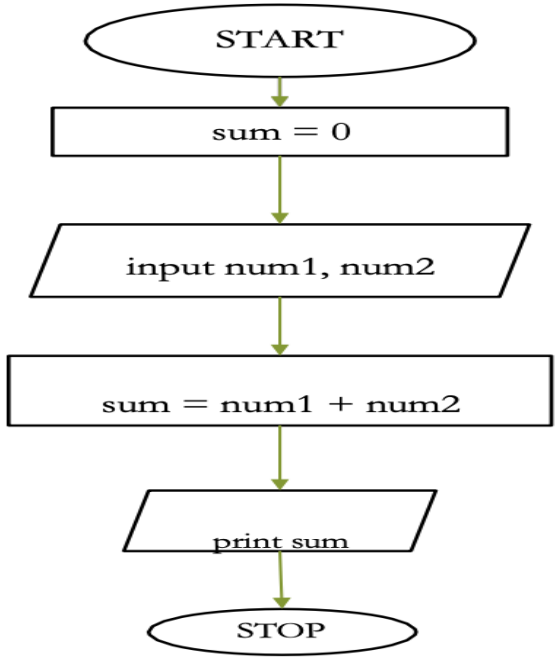


Figure: Flowchart for example 1

Flowchart: Example 2

An algorithm to add ten numbers provided by the user and print the result;

- **Inputs:** `num1`, `num2`, ..., `num10`
- **Intermediary:** `counter`
- **Outputs:** `sum`
- **Operations:** addition, print

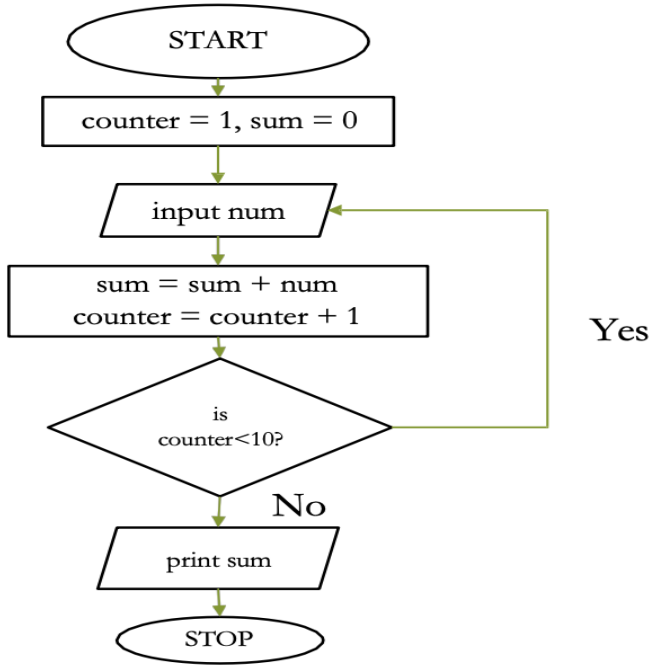


Figure: Flowchart for example 2

Pseudocode

- The steps of a program
- The steps include the following components,
  - Variables (example: `num1`, `num2`, `sum`)
  - Assignment (example: `num1=0`, `sum=0`, `sum=a+5`)
  - Decision (example: `sum>0`)
  - Repetition (example: repeat some steps for more than 10 times)

Pseudocode: Example 1

An algorithm to add two numbers provided by the user and print the result;

Step-1: Start

Step-2: Declare variables

```
sum=0
num1=0
num2=0
```

Step-3: Input values

```
print("enter values of num1 and num2 ")
```

Step-4: Perform the sum

```
sum = num1+num2
```

Step-5: Print the sum

```
print(sum)
```

Step-6: Stop

Pseudocode: Example 2

Write pseudo code that tells a user that the number they entered is not a 5 or a 6

Step-1: Start

Step-2: Input `num`

Step-3:

```
if(num!=5 && num!=6)
{
print("your number is not 5 or 6")
}
```

Note: && and != are called logical operators

Step-3: Stop

Pseudocode: Example 3

Write pseudo code that tells a user that the number they entered is not a 5 or a 6

Step-1: Start

Step-2: Input `num`

Step-3:

```
if( num ==5 || num ==6)
```

```
{go to **Step-5**}
```

```
else
```

```
{go to **Step-4**}
```

Note: || and == are called logical operators

Step-4: `print("your number is not a 5 or a 6")`

Step-5: Stop

V.  and R Studio

What is R Studio

- It is an Integrated Development Environment, also called the IDE
- It adds a layer of aesthetics and functionality over R
- While installing them, always make sure that you,
  - Install R
  - Install R Studio

Overview

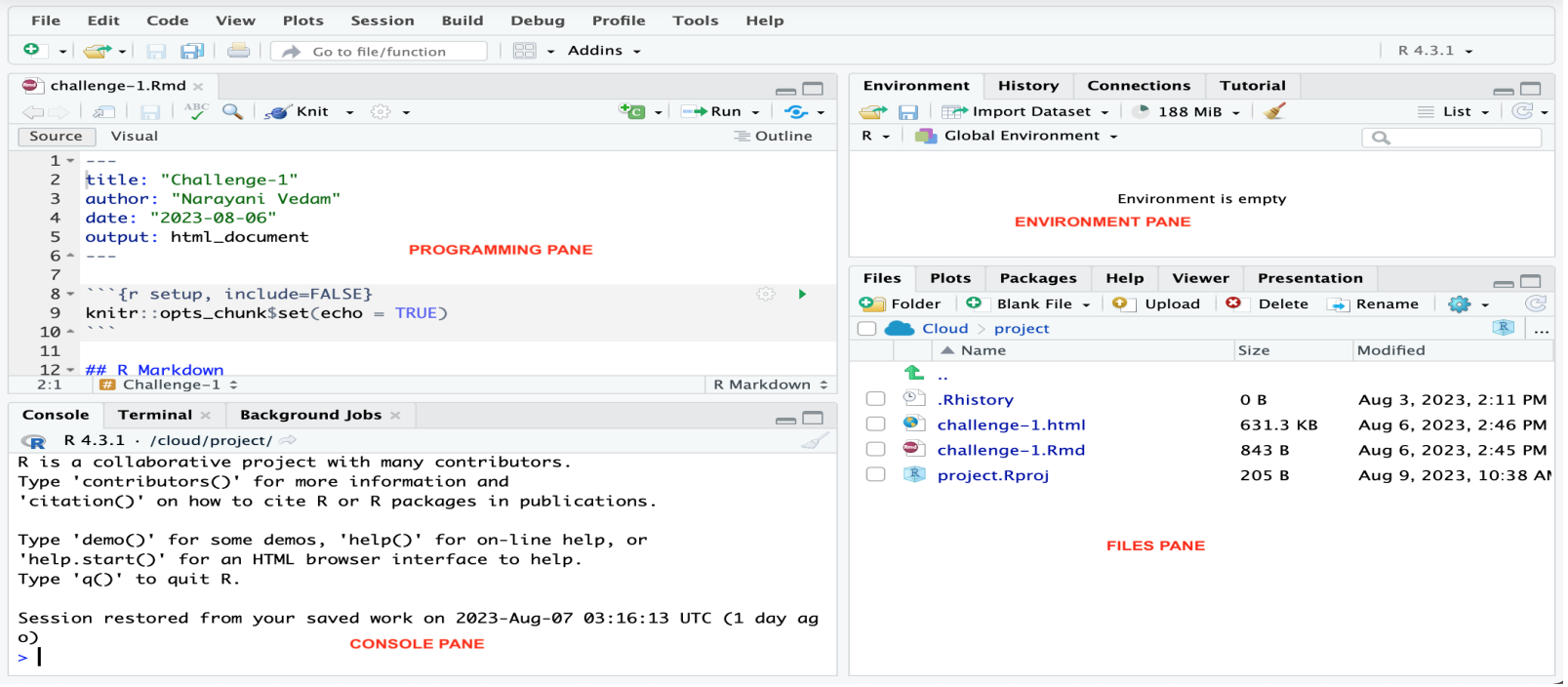


Figure: A typical layout of R Studio

Overview

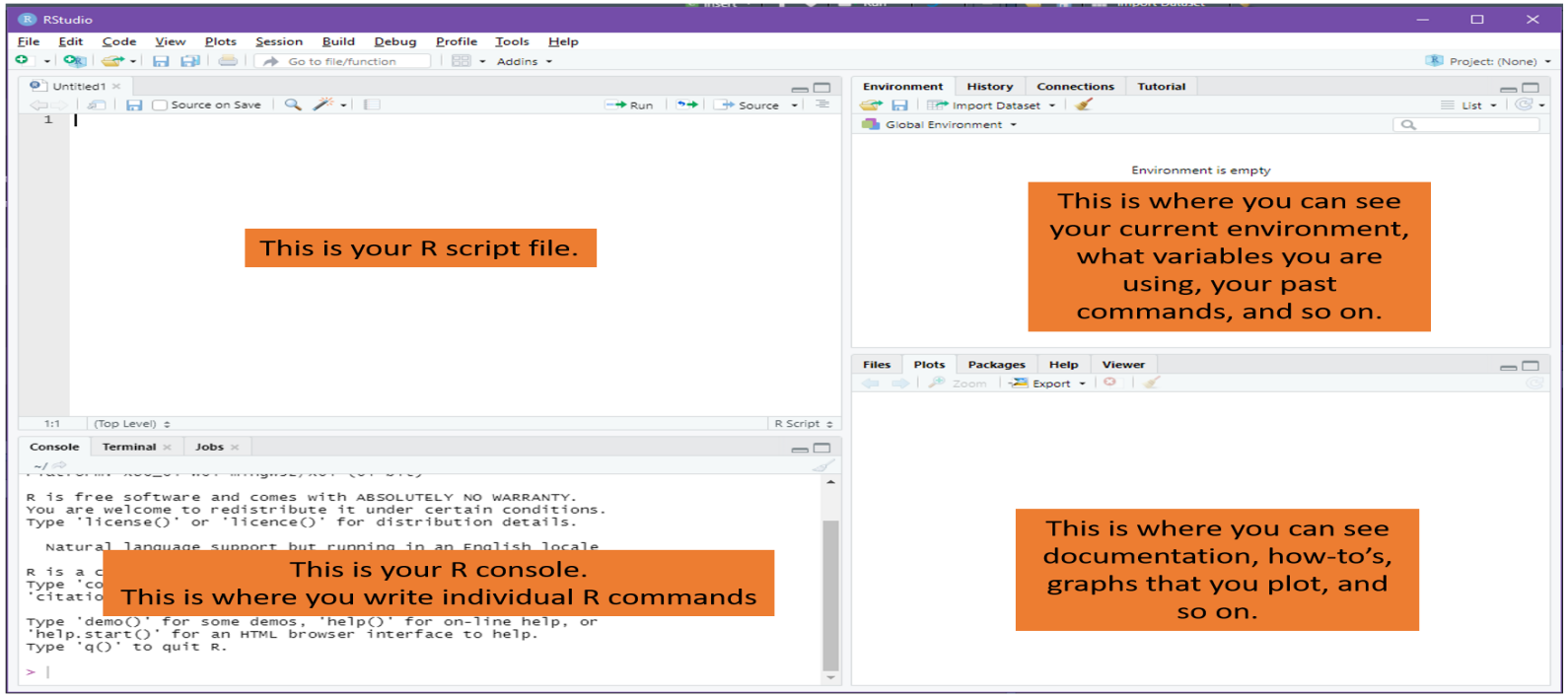


Figure: Different panes in the layout

VI. R Markdown

Overview

> It provides an authoring framework for data science.

- You can use a single R Markdown file to do the following
  - blend text, visualization and code
  - generate high quality reports that can be shared with the audience
- It supports a variety of output formats like PDF, HTML, among others
- Fully reproducible reports – each time you knit the analysis is run from the beginning
- Simple Markdown syntax for text

Note: Think of it as your "virtual notebook" that helps you code and document, all in one place!

What does it entail?

- An R Markdown file is a simple text file with a `.rmd` extension
- Notice in the figure beside that it has 3 types of content,
    1. An *optional* YAML header enclosed within `---`
    2. code chunks within ```` ``` ````
    3. Text with simple text formatting

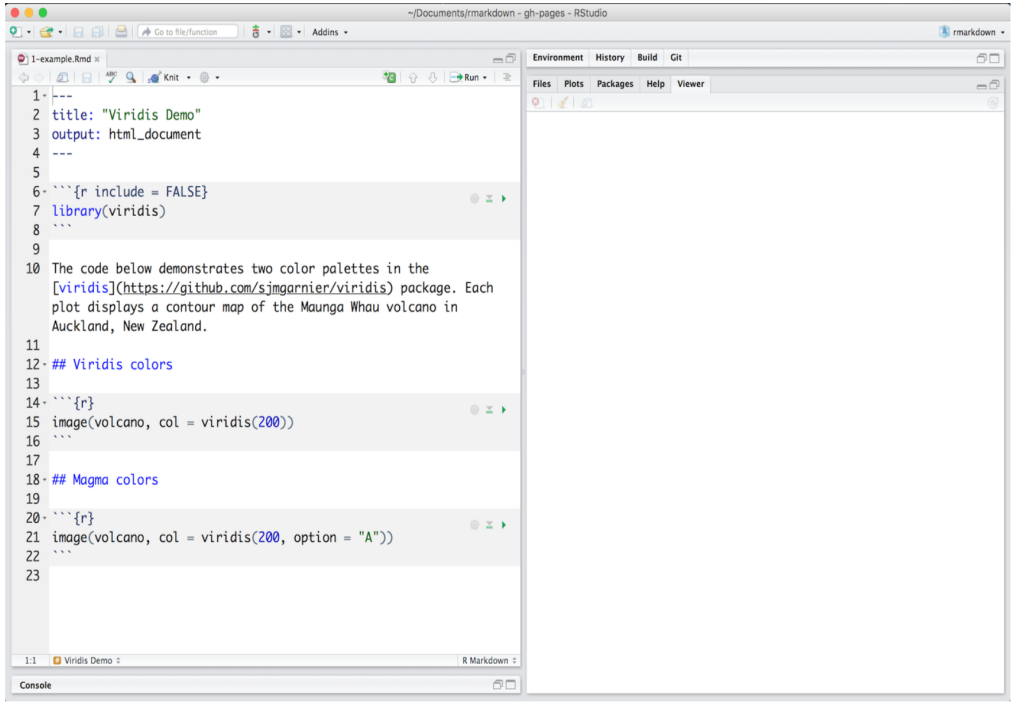


Figure: Typical R Markdown file

What does it entail?

An *optional* YAML header enclosed within `---`

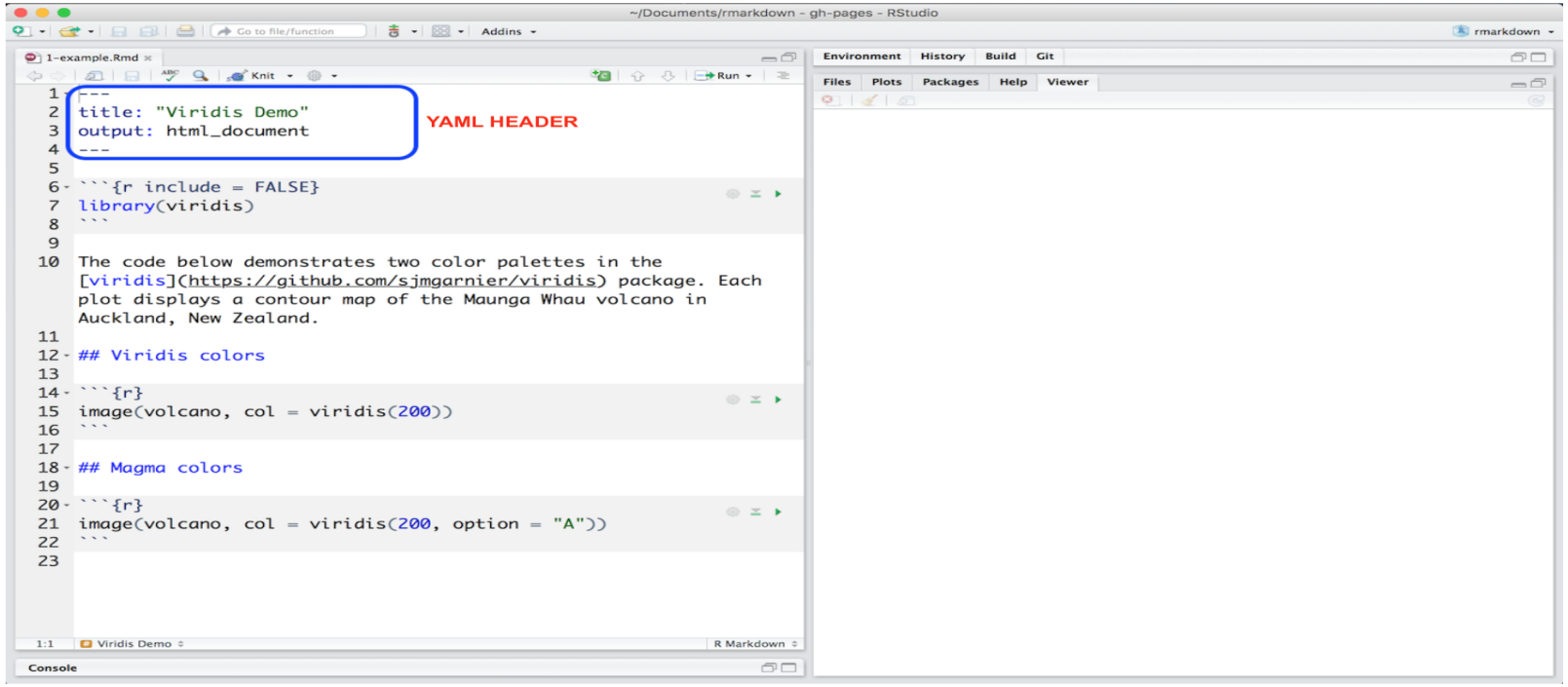


Figure: YAML header in a R Markdown file

What does it entail?

code chunks within ```

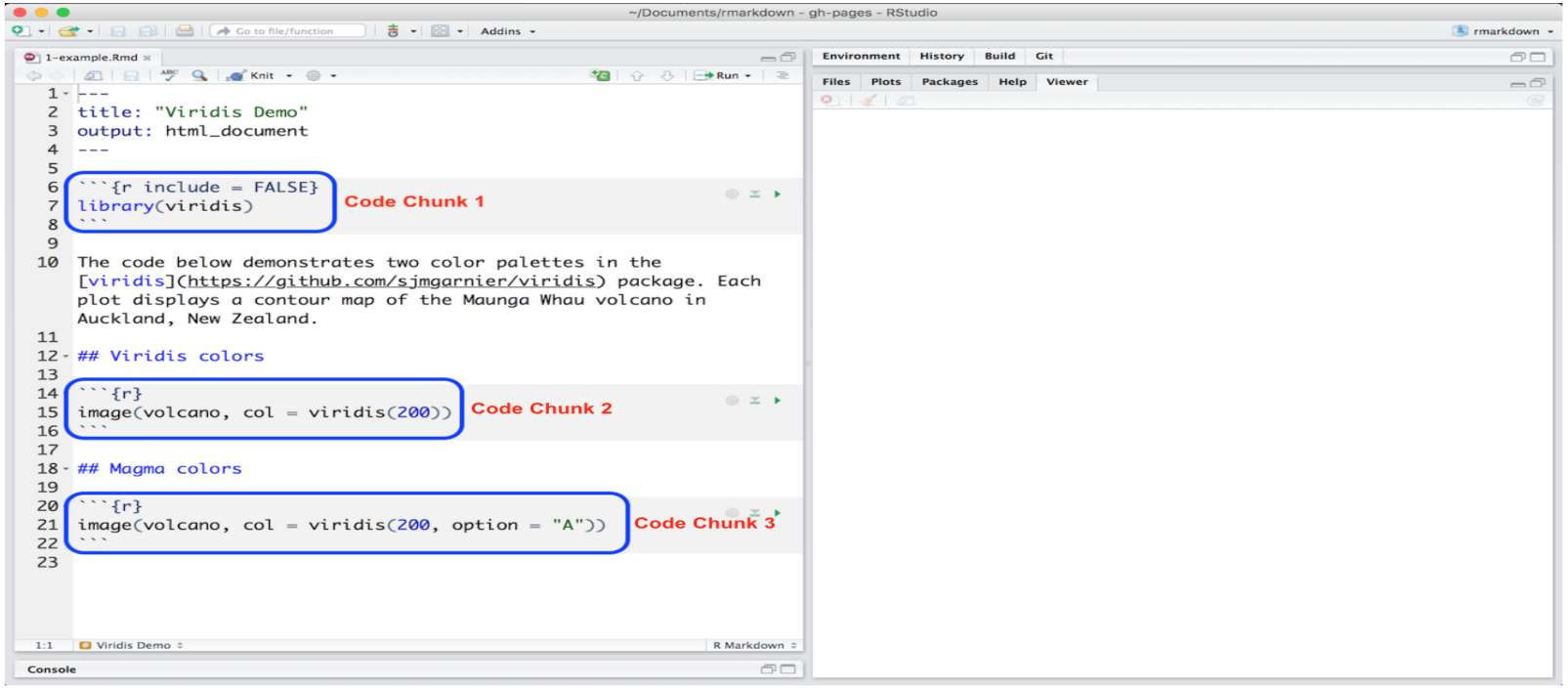


Figure: Code chunks in a R Markdown file

What does it entail?

Text with simple text formatting

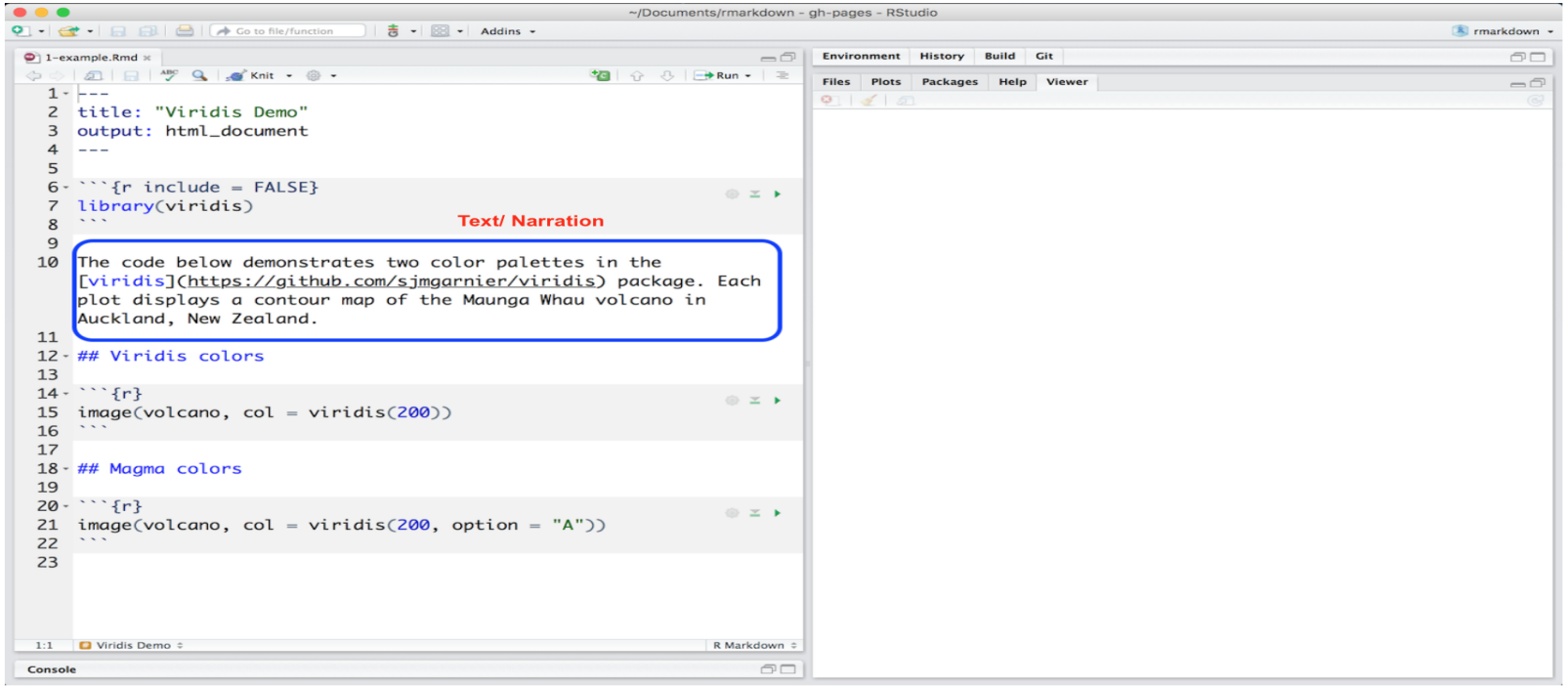


Figure: Text in a R Markdown file

Generating the report

To generate the output in the desired format, use the **Knit** button in posit Cloud IDE

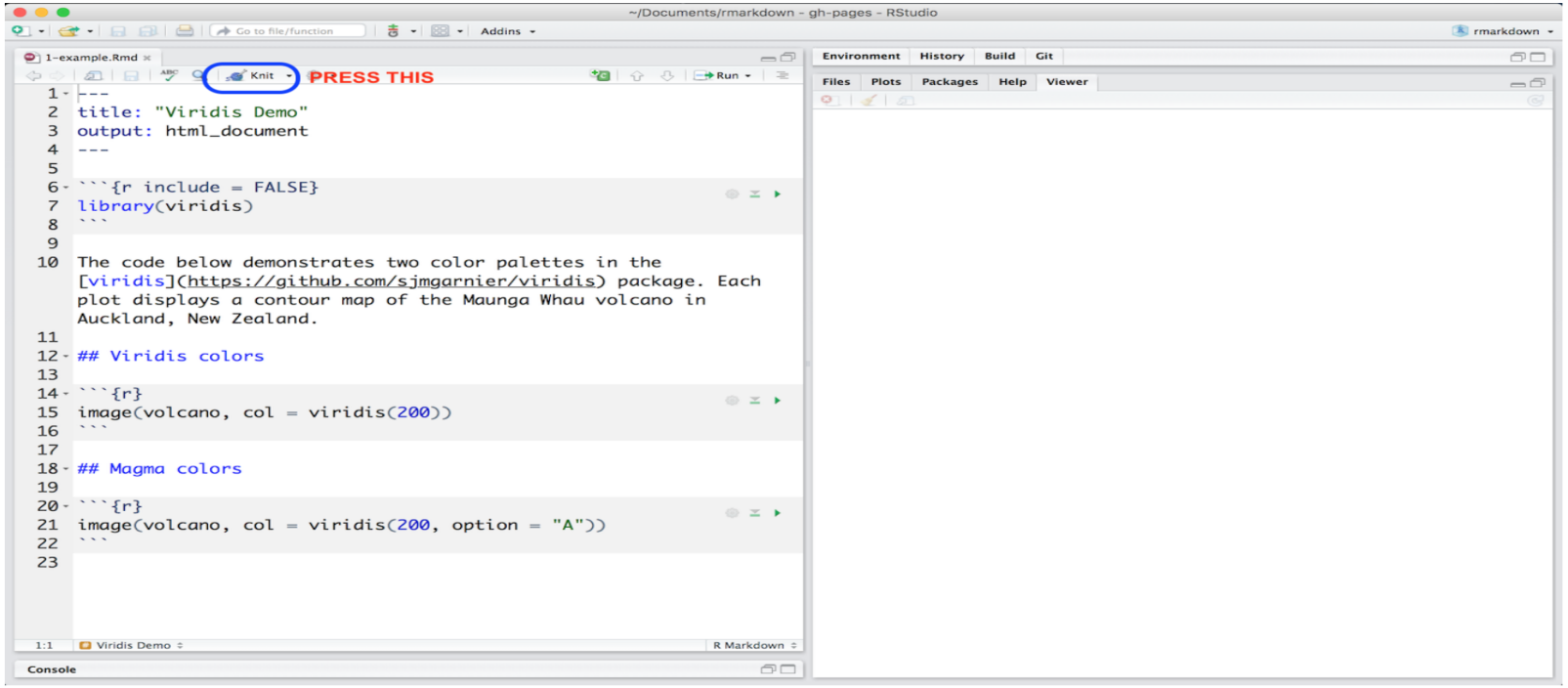


Figure: Text in a R Markdown file

How will we be using R Markdown?

- All the presentations in this course were prepared using R Markdown!!
- You will have to turn in all your submissions compiled using R Markdown 🥳
- Worry not, you will be provided with a R Markdown document template to work with 😀
- The amount of scaffolding in the template will decrease over the course of the semester

Thanks!

Slides created via the R packages:

xaringan
gadenbuie/xaringanthemer.

Faculty of Arts
& Social Sciences

Week-2: Introduction to Data & Visualization

NM2207: Computational Media Literacy

Narayani Vedam, Ph.D.

Department of Communications and New Media

NUS National University of Singapore | Faculty of Arts & Social Sciences

This week

Table of contents

I. Introduction to data

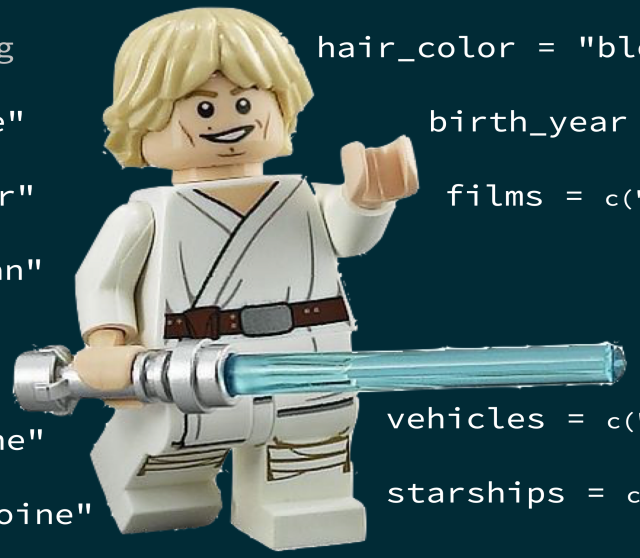
What constitutes a data-set?

```r
# Load R packages for data science
library(tidyverse)
# Data in starwars data-set
starwars
```

```
## # A tibble: 87 × 14
##    name      height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>      <int> <dbl> <chr>      <chr>      <chr>           <dbl> <chr> <chr>
##  1 Luke Sk…    172    77 blond      fair       blue             19   male  mascu…
##  2 C-3PO       167    75 <NA>       gold       yellow          112   none  mascu…
##  3 R2-D2        96    32 <NA>       white, bl… red              33   none  mascu…
##  4 Darth V…    202   136 none       white      yellow           41.9 male  mascu…
##  5 Leia Or…    150    49 brown      light      brown            19   fema… femin…
##  6 Owen La…    178   120 brown, gr… light      blue             52   male  mascu…
##  7 Beru Wh…    165    75 brown      light      blue             47   fema… femin…
##  8 R5-D4        97    32 <NA>       white, red red              NA   none  mascu…
##  9 Biggs D…    183    84 black      light      brown            24   male  mascu…
## 10 Obi-Wan…    182    77 auburn, w… fair       blue-gray        57   male  mascu…
## # ℹ 77 more rows
## # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Luke Skywalker

```
height = 172 cm
weight = 77 kg
eye_color = "blue"
skin_color = "fair"
species = "Human"
sex = "male"
gender = "masculine"
homeworld = "Tatooine"
name = "Luke Skywalker"
hair_color = "blond"
birth_year = 19 BBY
films = c("The Empire Strikes Back",
          "Revenge of the Sith",
          "Return of the Jedi",
          "A New Hope",
          "The Force Awakens")
vehicles = c("Snowspeeder",
             "Imperial Speeder Bike")
starships = c("X-wing",
              "Imperial shuttle")
```



Source: https://datasciencebox.org/

Overview of the data-set

```
# Salient features of the data-set
?starwars
```

starwars {dplyr} R Documentation

Starwars characters

Description

The original data, from SWAPI, the Star Wars API, https://swapi.dev/, has been revised to reflect additional research into gender and sex determinations of characters.

Usage

```
starwars
```

Format

A tibble with 87 rows and 14 variables:

name

Name of the character

height

Height (cm)

mass

Weight (kg)

hair_color,skin_color,eye_color

Hair, skin, and eye colors

birth_year

Year born (BBY = Before Battle of Yavin)

sex

The biological sex of the character, namely male, female, hermaphroditic, or none (as in the case for Droids).

gender

The gender role or gender identity of the character as determined by their personality or the way they were programmed (as in the case for Droids).

Dissecting the data-set: a glimpse

```
# Catch a glimpse starwars data-set
glimpse(starwars)
```

```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or…
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2…
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.…
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N…
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "…
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",…
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, …
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",…
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini…
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T…
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma…
## $ films      <list> <"The Empire Strikes Back", "Revenge of the Sith", "Return…
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp…
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",…
```

Dissecting the data-set: columns

> Each **column** is a variable

Notice the *type* of each variable in the previous slide. They are,

1. `chr` (**character** for text-based entries)

2. `int` (**integer** for whole numbers)

3. `dbl` (**double** for decimal numbers)

4. `list` (**list** for entries of more than one type `chr`, `int`, `dbl`)

Dissecting the data-set: column of type integer

Let us look at the entries of one of the columns of type `int`

```
# Access column "height"
starwars$height
```

```
##  [1] 172 167  96 202 150 178 165  97 183 182 188 180 228 180 173 175 170 180  66
## [20] 170 183 200 190 177 175 180 150  NA  88 160 193 191 170 196 224 206 183 137
## [39] 112 183 163 175 180 178  94 122 163 188 198 196 171 184 188 264 188 196 185
## [58] 157 183 183 170 166 165 193 191 183 168 198 229 213 167  79  96 193 191 178
## [77] 216 234 188 178 206  NA  NA  NA  NA  NA 165
```

Dissecting the data-set: column of type double

Let us look at the entries of one of the columns of type `dbl`

```
# Access column "mass"
starwars$mass
```

```
##  [1]   77.0   75.0   32.0  136.0   49.0  120.0   75.0   32.0   84.0   77.0
## [11]   84.0     NA  112.0   80.0   74.0 1358.0   77.0  110.0   17.0   75.0
## [21]   78.2  140.0  113.0   79.0   79.0   83.0     NA     NA   20.0   68.0
## [31]   89.0   90.0     NA   66.0   82.0     NA     NA     NA   40.0     NA
## [41]     NA   80.0     NA   55.0   45.0     NA   65.0   84.0   82.0   87.0
## [51]     NA   50.0     NA     NA   80.0     NA   85.0     NA     NA   80.0
## [61]   56.2   50.0     NA   80.0     NA   79.0   55.0  102.0   88.0     NA
## [71]     NA   15.0     NA   48.0     NA   57.0  159.0  136.0   79.0   48.0
## [81]   80.0     NA     NA     NA     NA     NA   45.0
```

Dissecting the data-set: column of type character

Let us look at entries of one of the columns of type `chr`

```
# Access column "gender"
starwars$gender
```

```
##  [1] "masculine" "masculine" "masculine" "masculine" "feminine"  "masculine"
##  [7] "feminine"  "masculine" "masculine" "masculine" "masculine" "masculine"
## [13] "masculine" "masculine" "masculine" "masculine" "masculine" "masculine"
## [19] "masculine" "masculine" "masculine" "masculine" "masculine" "masculine"
## [25] "masculine" "masculine" "feminine"  "masculine" "masculine" "masculine"
## [31] "masculine" "masculine" "masculine" "masculine" "masculine" "masculine"
## [37] NA          "masculine" "masculine" NA          "feminine"  "masculine"
## [43] "masculine" "feminine"  "masculine" "masculine" "masculine" "masculine"
## [49] "masculine" "masculine" "masculine" "feminine"  "masculine" "masculine"
## [55] "masculine" "masculine" "masculine" "feminine"  "masculine" "masculine"
## [61] "feminine"  "feminine"  "feminine"  "masculine" "masculine" "masculine"
## [67] "feminine"  "masculine" "masculine" "feminine"  "feminine"  "masculine"
## [73] "feminine"  "masculine" "masculine" "feminine"  "masculine" "masculine"
## [79] "masculine" NA          "masculine" "masculine" "feminine"  "masculine"
## [85] NA          NA          "feminine"
```

Dissecting the data-set: column of type list

Let us look at entries of one of the columns of type `list`

```
# Access column "films"
starwars$films[1:3]
```

```
## [[1]]
## [1] "The Empire Strikes Back" "Revenge of the Sith"
## [3] "Return of the Jedi"      "A New Hope"
## [5] "The Force Awakens"
##
## [[2]]
## [1] "The Empire Strikes Back" "Attack of the Clones"
## [3] "The Phantom Menace"      "Revenge of the Sith"
## [5] "Return of the Jedi"      "A New Hope"
##
## [[3]]
## [1] "The Empire Strikes Back" "Attack of the Clones"
## [3] "The Phantom Menace"      "Revenge of the Sith"
## [5] "Return of the Jedi"      "A New Hope"
## [7] "The Force Awakens"
```

Dissecting the data-set: rows

> Each **row** is an observation

```
# Rows of interest
filter_rows <- c("Luke Skywalker","R2-D2","Darth Vader")
# Extract row corresponding to Luke Skywalker
starwars %>% filter(name%in%filter_rows)
```

```
## # A tibble: 3 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>           <dbl> <chr> <chr>
## 1 Luke Sky…    172    77 blond      fair       blue             19   male  mascu…
## 2 R2-D2         96    32 <NA>       white, bl… red              33   none  mascu…
## 3 Darth Va…    202   136 none       white      yellow           41.9 male  mascu…
## # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Dissecting the data-set: rows with selected variables

Filter some rows and **select** some of their attributes

```
# Rows of interest
filter_rows <- c("Luke Skywalker","R2-D2")
# Extract row corresponding to Luke Skywalker
starwars %>% filter(name%in%filter_rows) %>% select(name,height,mass,homeworld,films)
```

```
## # A tibble: 2 × 5
##   name           height  mass homeworld films
##   <chr>           <int> <dbl> <chr>     <list>
## 1 Luke Skywalker    172    77 Tatooine  <chr [5]>
## 2 R2-D2              96    32 Naboo     <chr [7]>
```

Notice how the entries of the column, films, do not show up!

Dissecting the data-set: rows with selected variables (continued)

Select attributes of type `list`

```
# Rows of interest
filter_rows <- c("Luke Skywalker","R2-D2")
# Extract rows in 'rows'
starwars %>% filter(name%in%filter_rows) %>% pull(films)
```

```
## [[1]]
## [1] "The Empire Strikes Back" "Revenge of the Sith"
## [3] "Return of the Jedi"      "A New Hope"
## [5] "The Force Awakens"
##
## [[2]]
## [1] "The Empire Strikes Back" "Attack of the Clones"
## [3] "The Phantom Menace"      "Revenge of the Sith"
## [5] "Return of the Jedi"      "A New Hope"
## [7] "The Force Awakens"
```

Dissecting the data-set: basic properties

```
# Number of rows in the data-set
nrow(starwars)
```

```
## [1] 87
```

```
# Number of columns in the data-set
ncol(starwars)
```

```
## [1] 14
```

```
# Number of rows and columns in the data-set
dim(starwars)
```

```
## [1] 87 14
```

II. Exploratory Data Analyses (EDA)

What is EDA?

- It is a systematic and iterative approach to explore data, and includes,
  - A hypothesis
  - Visualization, transformation or modelling data for answers
  - Substantiation or refinement of the hypothesis

- We will focus on **exploring** -- wrangling, manipulating, transforming --- and **visualizing** data

Need for EDA

1. Are the attributes of the starwars characters related?
    - For instance, mass and height of the characters

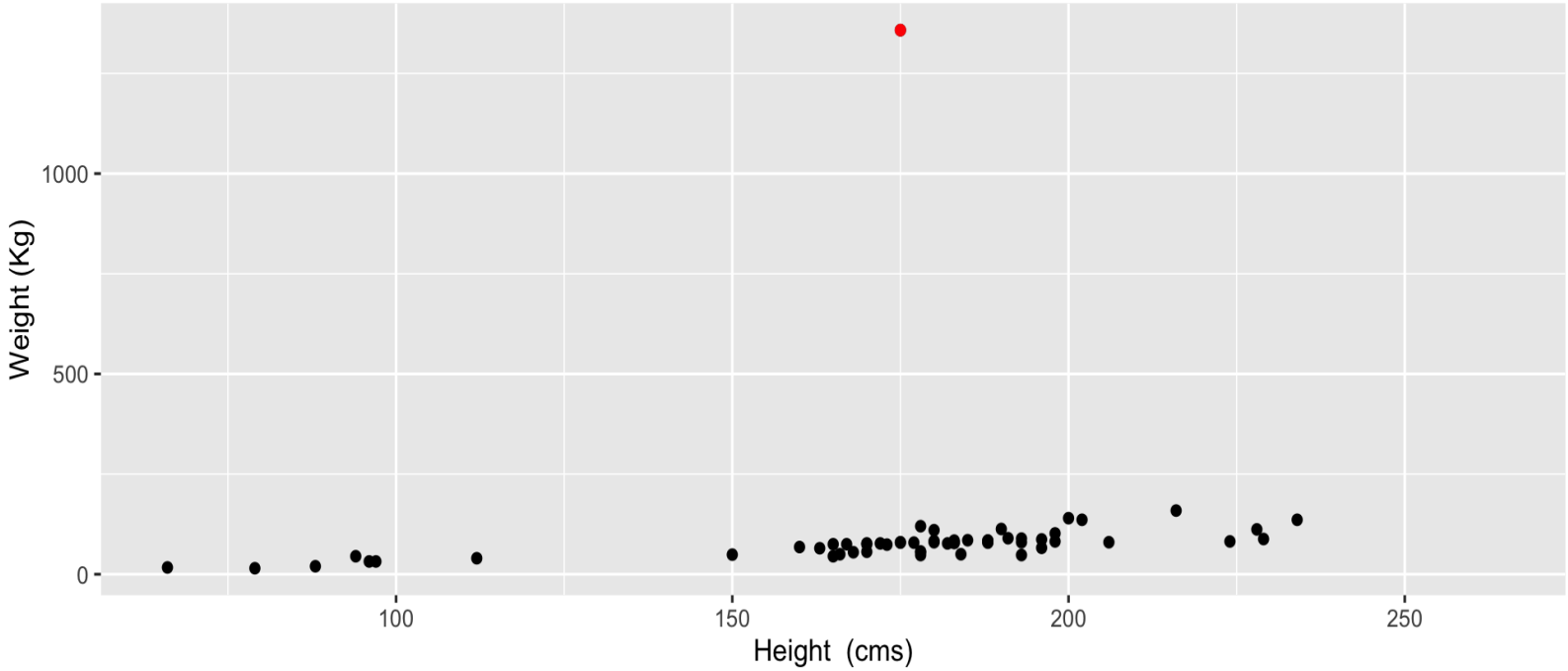
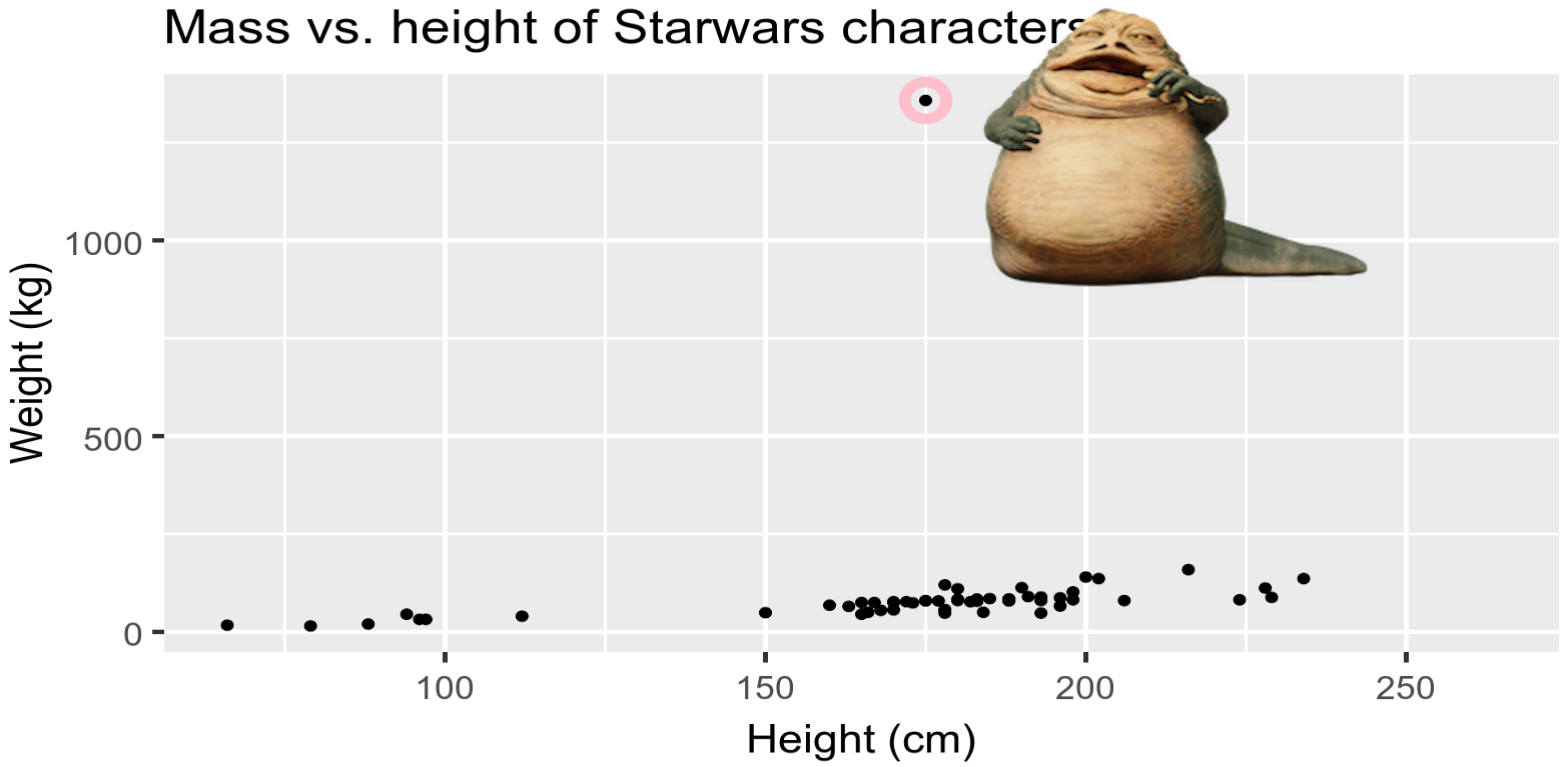


Figure: Relationship between weight and height

Jabba!



Source: https://datasciencebox.org/

Data visualization

> "A picture is worth a thousand words" - Frederick R. Bernard

1. Is just a visual representation of data
2. Entails creation and study of such representations
3. Among the many packages that are available for visualization, we will use `ggplot2` and `Shiny`

Need for visualization: Anscombe's quartet

1. Consider a data-set, the **Anscombe's quartet**
2. It has four sets of data indicated by *I*, *II*, *III* and *IV*
3. Each set of data has 11 points, `(x,y)`

```
# Invoke the library
library(Tmisc)
# Filter data-set I in quartet
quartet %>% filter(set=="I")
```

```
##    set  x     y
## 1    I 10  8.04
## 2    I  8  6.95
## 3    I 13  7.58
## 4    I  9  8.81
## 5    I 11  8.33
## 6    I 14  9.96
## 7    I  6  7.24
## 8    I  4  4.26
## 9    I 12 10.84
```

```
# Invoke the library
library(Tmisc)
# Filter data-set II in quartet
quartet %>% filter(set=="II")
```

```
##    set  x    y
## 1   II 10 9.14
## 2   II  8 8.14
## 3   II 13 8.74
## 4   II  9 8.77
## 5   II 11 9.26
## 6   II 14 8.10
## 7   II  6 6.13
## 8   II  4 3.10
## 9   II 12 9.13
```

Need for visualization: Anscombe's quartet

1. Consider a data-set, the **Anscombe's quartet**
2. It has four sets of data indicated by *I*, *II*, *III* and *IV*
3. Each set of data has 11 points, `(x,y)`

```
# Invoke the library
library(Tmisc)
# Filter data-set III in quartet
quartet %>% filter(set=="III")
```

```
##    set  x     y
## 1  III 10  7.46
## 2  III  8  6.77
## 3  III 13 12.74
## 4  III  9  7.11
## 5  III 11  7.81
## 6  III 14  8.84
## 7  III  6  6.08
## 8  III  4  5.39
## 9  III 12  8.15
```

```
# Invoke the library
library(Tmisc)
# Filter data-set IV in quartet
quartet %>% filter(set=="IV")
```

```
##    set  x     y
## 1   IV  8  6.58
## 2   IV  8  5.76
## 3   IV  8  7.71
## 4   IV  8  8.84
## 5   IV  8  8.47
## 6   IV  8  7.04
## 7   IV  8  5.25
## 8   IV 19 12.50
## 9   IV  8  5.56
```

Anscombe's quartet: features (continued)

- And look at some statistics,

  a. Mean of `x` in each set, `mean_x`

  b. Mean of `y` in each set, `mean_y`

  c. Standard deviation of `x` in each set, `sd_x`

  d. Standard deviation of `y` in each set, `sd_y`

  e. Correlation between `x` and `y`, `r`

```
# Obtain the needed statistics
grouped_quartet %>%
  summarise(
    mean_x = mean(x),
    mean_y = mean(y),
    sd_x = sd(x),
    sd_y = sd(y),
    r = cor(x, y)
  )
```

```
## # A tibble: 4 × 6
##   set   mean_x mean_y  sd_x  sd_y     r
##   <fct>  <dbl>  <dbl> <dbl> <dbl> <dbl>
## 1 I          9   7.50  3.32  2.03 0.816
## 2 II         9   7.50  3.32  2.03 0.816
## 3 III        9   7.5   3.32  2.03 0.816
## 4 IV         9   7.50  3.32  2.03 0.817
```

Anscombe's quartet: features (continued)

Will the **distributions** of the four sets of data points be identical too? Let us find out!

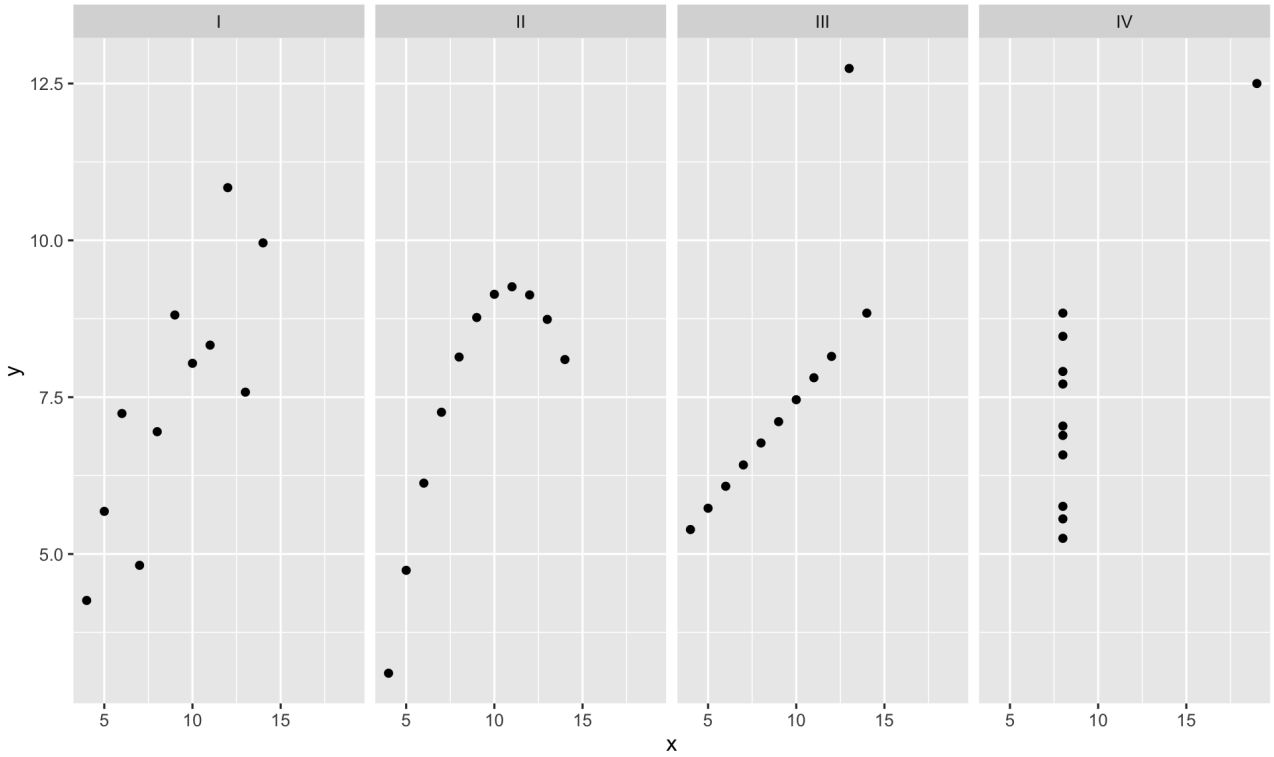


Figure: Data in each of the four data sets

III. `ggplot2`

Data visualization with `ggplot2`

- It is the `tidyverse` library's visualization package
- `gg` in `ggplot` stands for **G**rammar of **G**raphics
- Enables us to describe different components of a graphic

Figure: Different layers of ggplot

Let us construct our first plot: Mass versus Height

- The main function is `ggplot()`
- We will work with `starwars` data-set

```
# Plot the data
ggplot(data=starwars) # <-- Look here!
```


Figure: Constructing Mass versus Height plot

Mass versus Height: independent variable (IV)

- The **independent variable** (along x-axis) is the variable/column -- `height`

```
# Plot height along x-axis
ggplot(data=starwars) +
  aes(x=height) # <-- Look here!
```

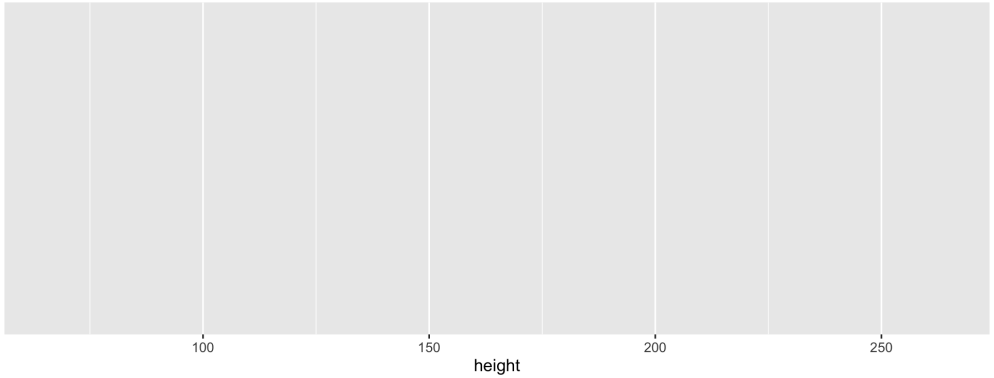


Figure: Adding independent variable

Mass versus Height: dependent variable (DV)

- Similarly, the **dependent variable** (along y-axis) is the variable/column -- `mass`

```
# Plot mass along y-axis
ggplot(data=starwars) +
  aes(x=height,y=mass) # <-- Look here!
```

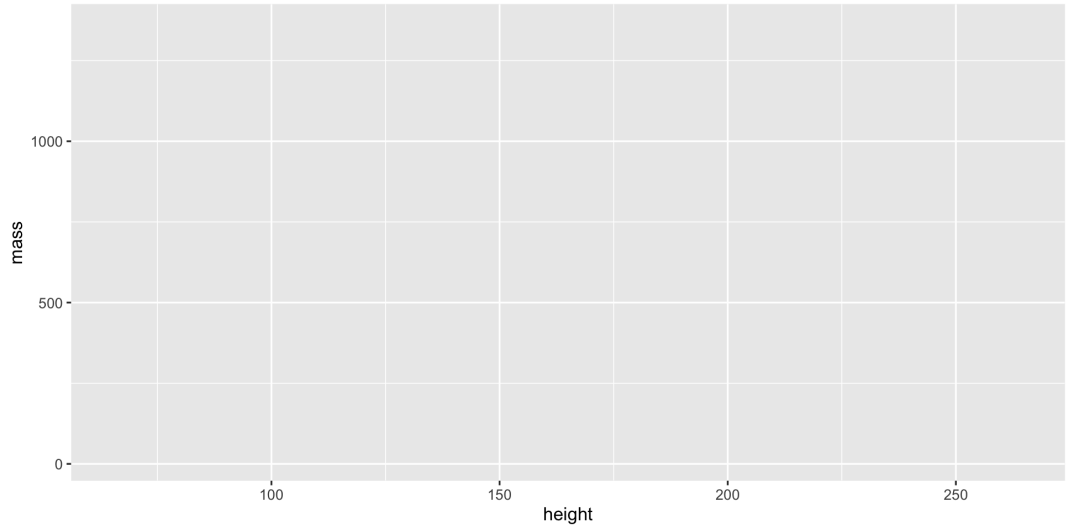


Figure: Adding dependent variable

Mass versus Height: visualize data as dots

- Let us choose to visualize the data as **points** (dots)

```
ggplot(data=starwars) + aes(x=height,y=mass) +
  geom_point() # <-- Look here!
```

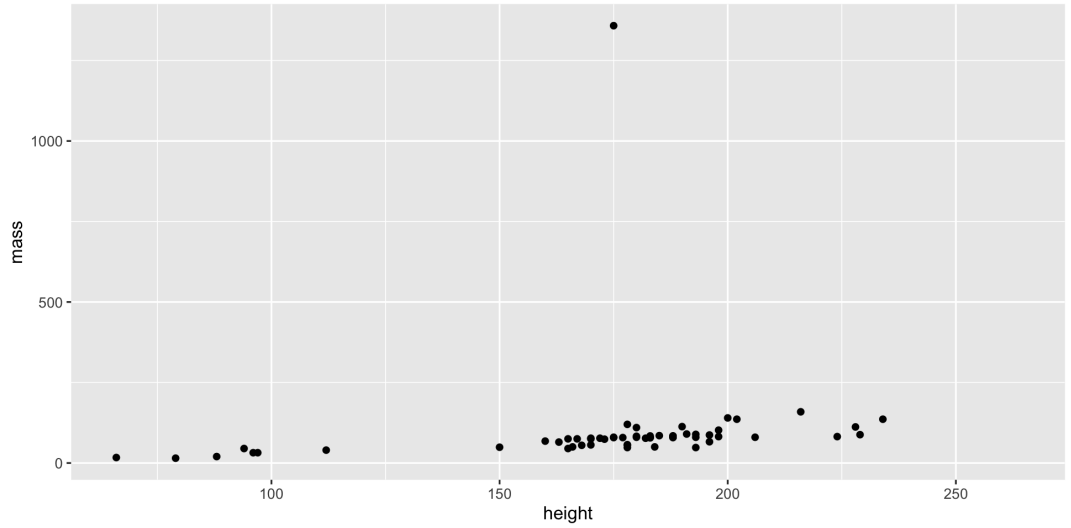


Figure: Visualizing data as points

Mass versus Height: re-write axes labels

- Let us rewrite **labels** for the **axes** -- `x` and `y`

```
ggplot(data=starwars) + aes(x=height,y=mass) + geom_point() +
  labs(x="Height (cm)",y="Weight (Kg)") # <-- Look here!
```

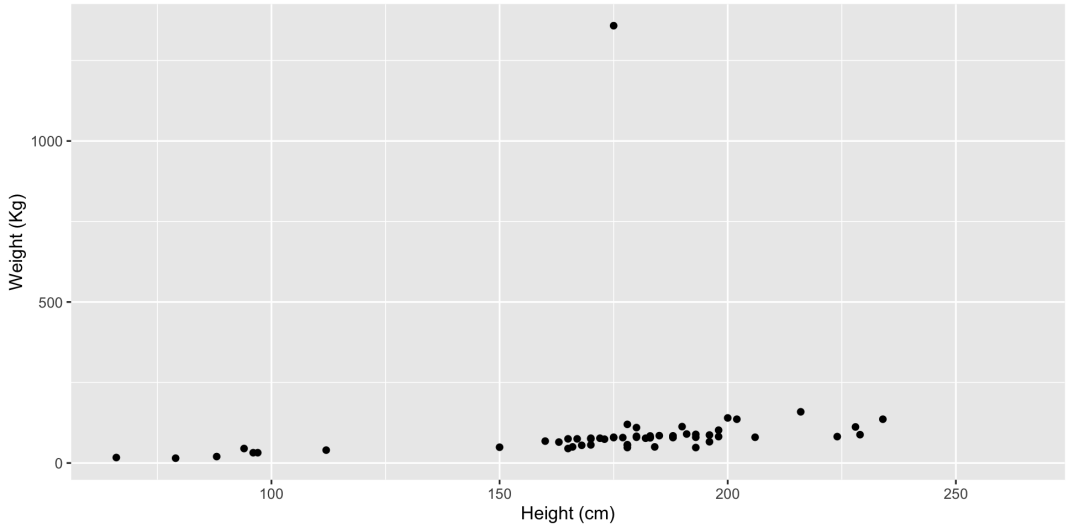


Figure: Inserting labels

Mass versus Height: add title

- Let us insert a **title** for the plot

```
ggplot(data=starwars,mapping=aes(x=height,y=mass)) +
  geom_point() +
  labs(x="Height (cm)",y="Weight (Kg)",
       title="Mass versus Height")# <-- Look here!
```

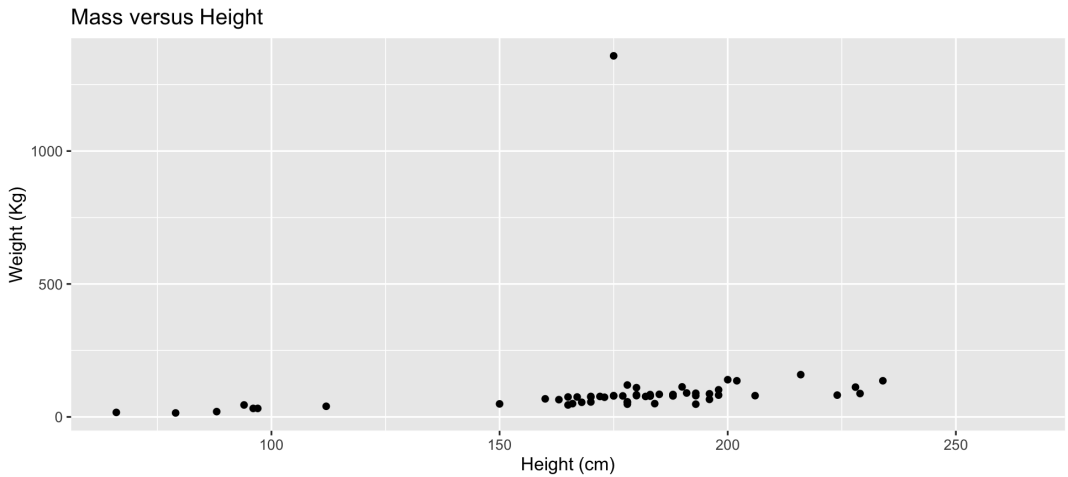


Figure: Inserting title

Mass versus Height: add caption

- Let us insert a **caption** for the plot

```
ggplot(data=starwars,mapping=aes(x=height,y=mass)) +
  geom_point() +
  labs(x="Height (cm)",y="Weight (Kg)",
       title="Mass versus Height",
       caption="Source: tidyverse/ starwars dataset") # <-- Look here!
```

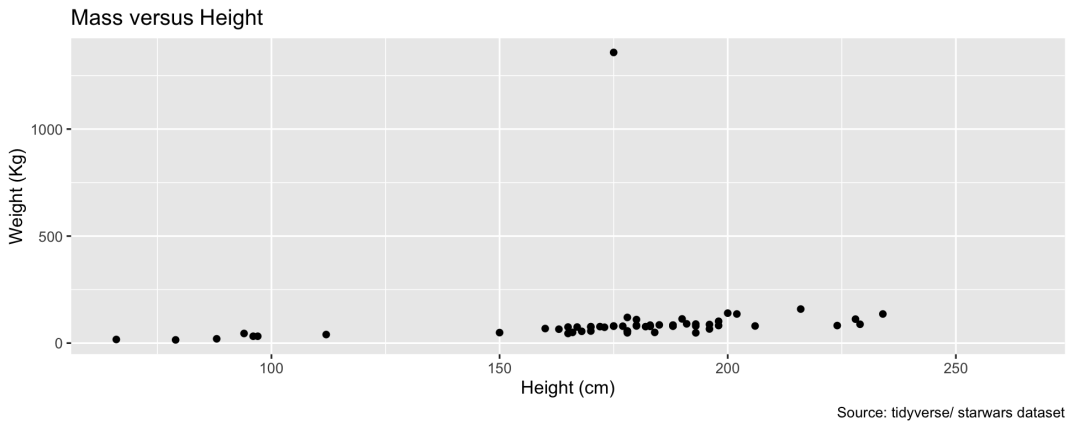


Figure: Inserting caption

Mass versus Height: complete code

```
ggplot(starwars) +
  aes(x=height,y=mass) +
    geom_point() +
      labs(x="Height (cm)",
             y="Weight (Kg)",
               title="Mass versus Height",
                 caption="Source: tidyverse/ starwars dataset")
```

Mass versus Height: tadaa!

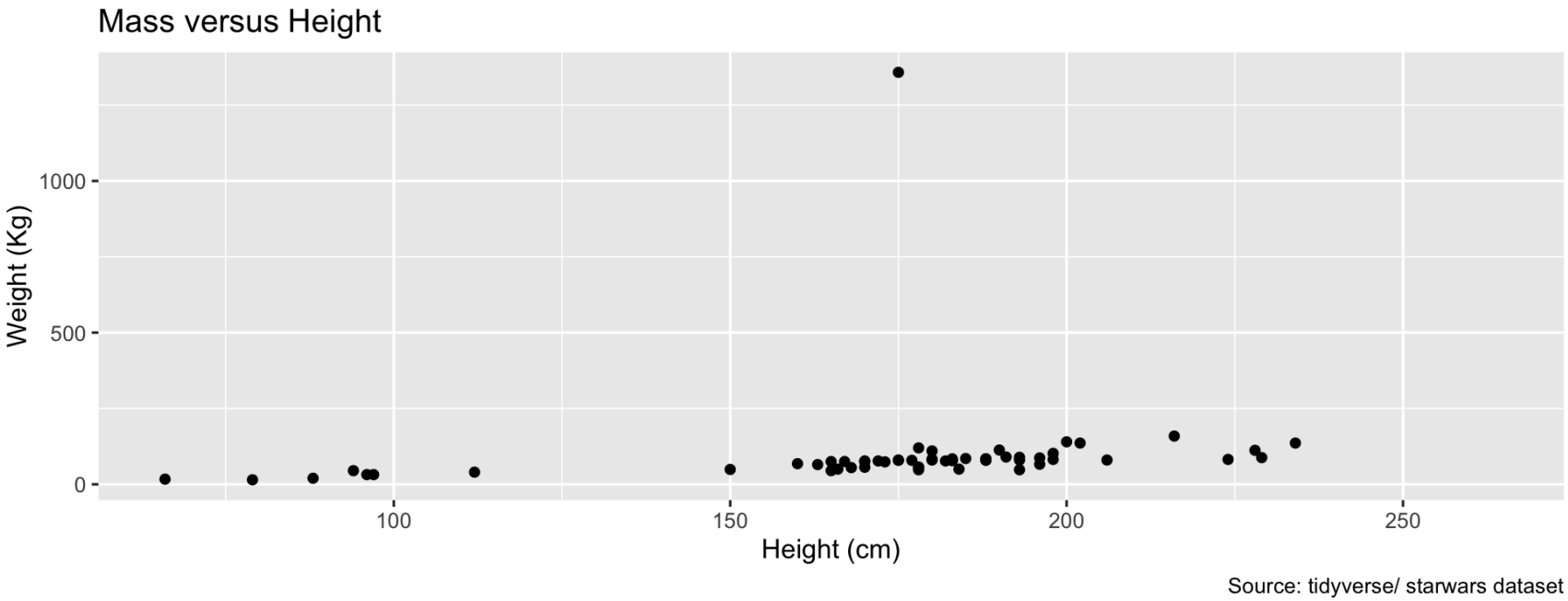


Figure: Final Mass versus Height plot

IV. Shiny

`Shiny`: introduction

1. `Shiny` is an R package that facilitates interactive web apps more here. Using which, apps
   a. can be hosted on a webpage
   b. embedded in a dashboard, or
   c. embedded in an R Markdown document
2. We shall explore some examples in the subsequent slides

`Shiny`: example

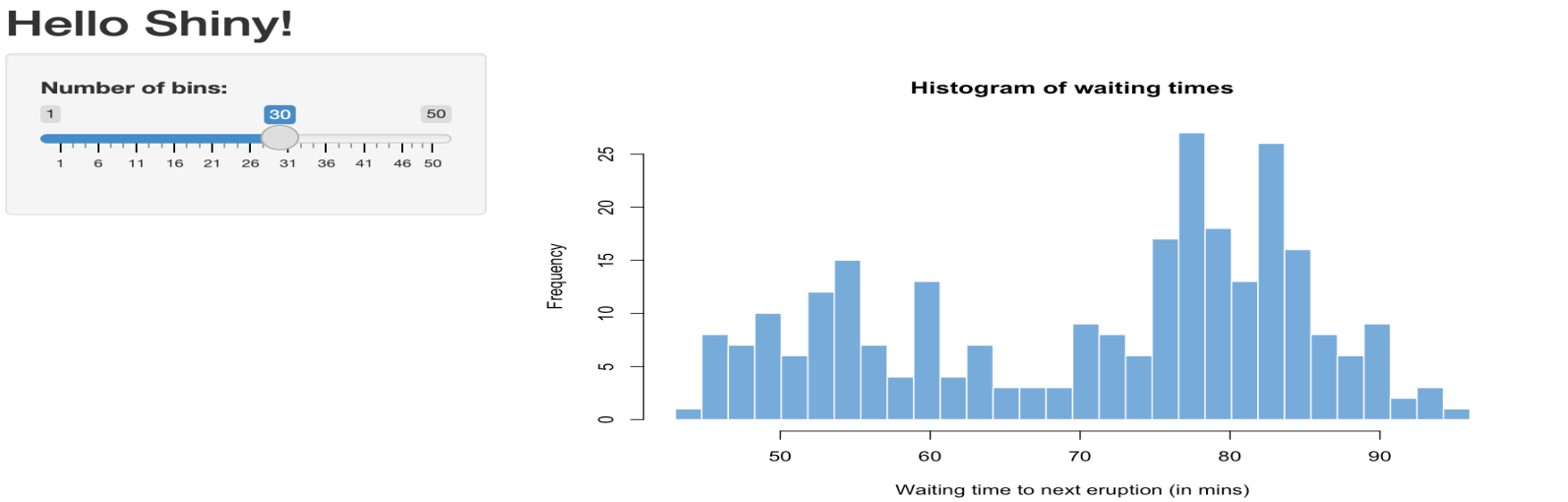
- The package has eleven built-in examples to demonstrate how `Shiny` works
- To explore the first example, run the code below

```
# Install package
install.packages("shiny")

# Invoke the package
library(shiny)

# Run an example from the library
runExample("01_hello")
```

- You could try to list all the examples using `runExample()`
- If you haven't been using `?` operator before every function you use, start doing so
- It reveals a lot of information about the function
- If you tried the code, move on to the next slide for the output

Example

`Shiny` dashboard with example - "01_hello"



This small Shiny application demonstrates Shiny's automatic UI updates.

Move the *Number of bins* slider and notice how the `renderPlot` expression is automatically re-evaluated when its dependant, `input$bins`, changes, causing a histogram with a new number of bins to be rendered.

app.R

show with app

```
library(shiny)

# Define UI for app that draws a histogram ----
ui <- fluidPage(

  # App title ----
  titlePanel("Hello Shiny!")
```

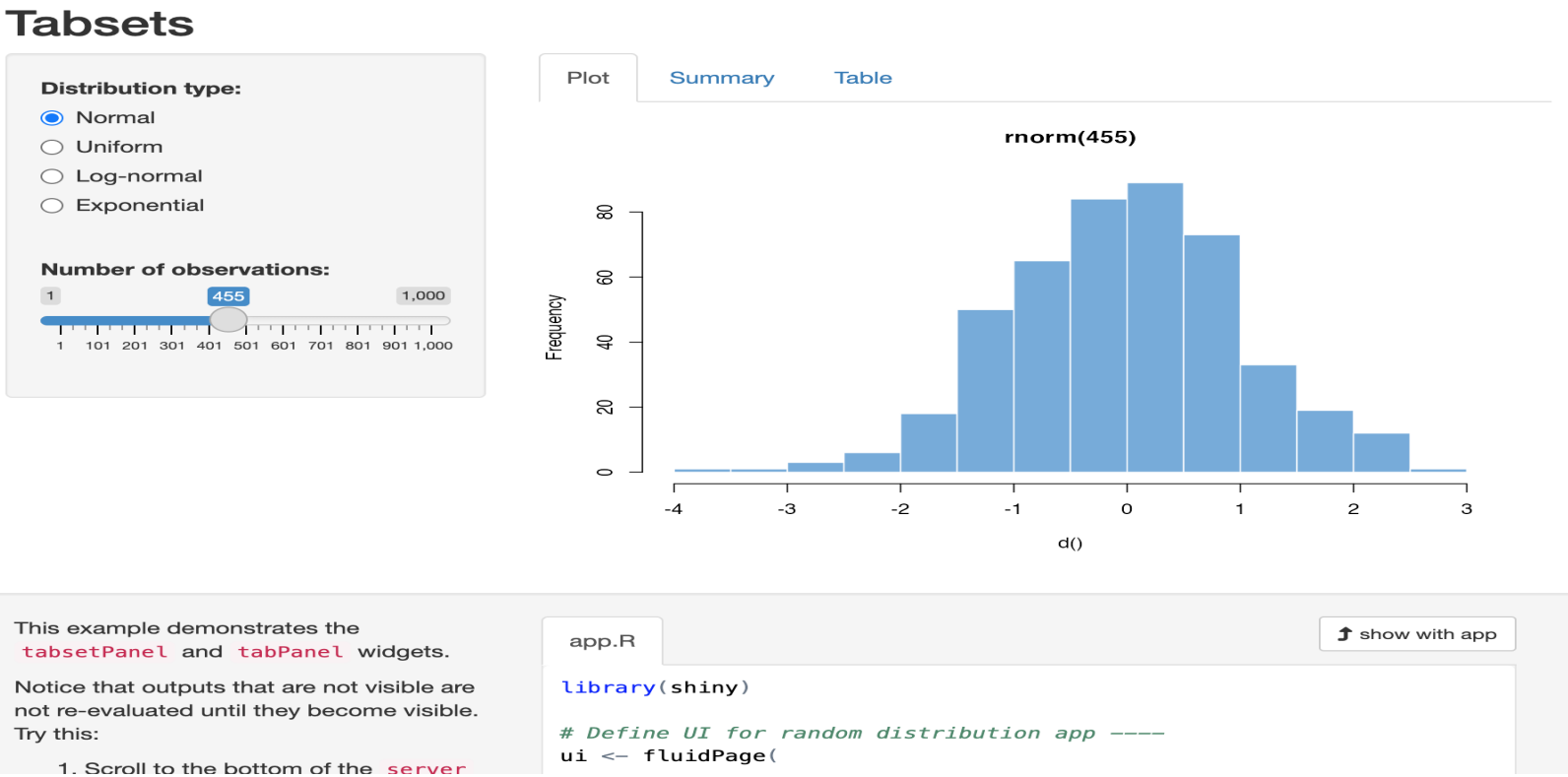
`Shiny`: comprehensive example

```
# Install package
install.packages("shiny")

# Invoke the package
library(shiny)

# Run an example from the library
runExample("06_tabsets")
```

Head to the next slide for the output!

Comprehensive example

`Shiny` dashboard with example - "06_tabsets"



More examples

To try more examples, just replace the example name in the code snippet from the list

1. 01_hello
2. 02_text
3. 03_reactivity
4. 04_mpg
5. 05_sliders
6. 06_tabsets
7. 07_widgets
8. 08_html
9. 09_upload
10. 10_download
11. 11_timer

```
# Install package
install.packages("shiny")

# Invoke the package
library(shiny)

# Run an example from the library
runExample("insert_example_name_here")
```

Commands and operators revisited

1. `?` - operator to learn more about an R command
2. `$` - operator to extract a specific part of data
3. `%>%` - pipe operator that passes the output of one function as an input to another
4. `%in%` - operator to check if an element belongs to a data-set
5. `library(package_name)` - to load an add-on package
6. `glimpe(dataset_name)` - get a glimpse of your data
7. `filter(row_attributes)` - subset rows using column values
8. `select(col_names)` - subset columns using their names or types
9. `pull(column_name)` - extracts a single column
10. `nrow/ncol/dim(dataset_name)` - number of rows/columns/dimension of a data-set

Thanks!

Slides created via the R packages:

xaringan
gadenbuie/xaringanthemer.

Faculty of Arts
& Social Sciences

Week-3: Variables & their types

NM2207: Computational Media Literacy

Narayani Vedam, Ph.D.

Department of Communications and New Media

NUS National University of Singapore | Faculty of Arts & Social Sciences

This week

Table of contents

I. Variables

What are variables?

- In programming, a **variable** is a name given to locations in the memory of the computer
- They allow the user to *store* and *manipulate* data
- R has six primitive types of variables,

    1. character
    2. double (real numbers)
    3. integer
    4. complex
    5. logical
    6. raw

Note: Think of them to be wallets that hold data instead of currency!

Variables (continued)

- Each variable has four attributes

> 1. an identifier (or name)
> 2. location
> 3. type
> 4. value

- The *identifier* and *value* are assigned by the user
  - Remember to always give names that are meaningful and related to the data
  - Avoid using space between characters/words in the name
- The *value* assigned by the user determines it *type*
- The *location* and *type* are automatically assigned, unless the user desires to override them

Variables: Classification

Variables can be broadly classified into two kinds,

1. **Numeric variables**
   a. Continuous
   b. Discrete
2. **Non-numeric variables/Categoric**
   a. Ordinal
   b. Nominal
   c. Binary

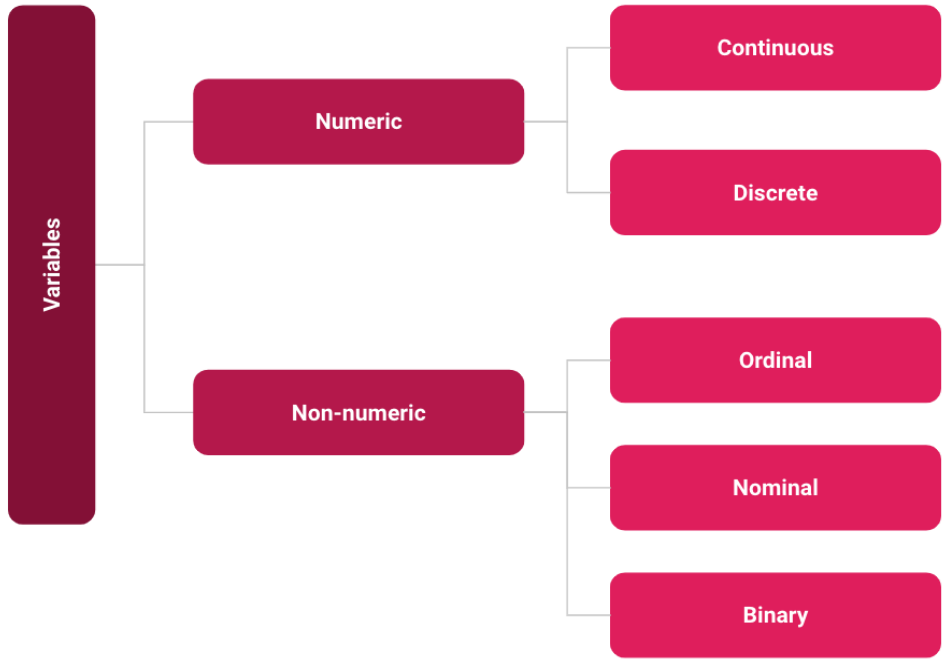


Figure: Classification of variables

Variables: Numeric

Numeric variables could be one of the following,

1. **integer**: are whole numbers, *without* decimal values

   Examples

   1000, 100, 200, 0, 1, 121353346970980,-1,-10

2. **double**: are numbers *with* decimal values

   Examples

   1.1, 0.89, 200.0, 0.001, 1000.5090809090068,-0.98,-200.0

3. **complex**: are *imaginary* numbers

   Examples

   1i, 2+10i, -200, 0.001, 1000.5090809090068-896.43124319i

Numeric variables: classification

> They can be **continuous**

- Of *type* double
- Can have infinite <u>decimal</u> values

<u>Examples</u>

Weight, Temperature, Height

> They can also be **discrete**

- Of *type* integer or double
- They are finite

<u>Examples</u>

Population, Number of occurrences

<!--

Non-numeric variables: Categoric

Non-numeric variables are also called categoric variables. Additionally, they can be,

1. **character**: single alphabet or numeral

   Examples

   'A', 'v', 'X', 'P', 'd', '1', '2'

2. **character**: string of alphabets or numerals

   Examples

   "hello", "world", "hi there", "20"

3. **logical**: either `TRUE` or `FALSE`

   Note: Character variables can be enclosed either within single or double quotes. However, the practice is to use single quotes for a character and double quotes for a string of characters.

Categoric variables: Ordinal

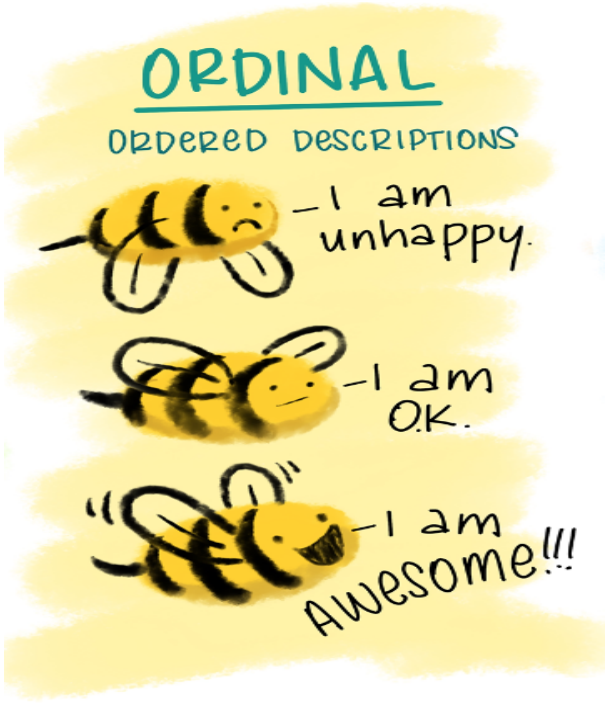
> Categoric variables can be ordinal

- They can be of type - **character**
- They have *natural ordering*

Examples

Survey responses ("Strongly agree", "Agree", "Neutral", "Disagree", "Strongly disagree")

Survey responses ("Low", "Medium", "High")



Source: https://allisonhorst.com/everything-else

Categoric variables: Nominal

> Categoric variables can be nominal

- They can be of either types - **character** or **logical**
- They *do not* have a natural ordering

<u>Examples</u>

Gender ("Male", "Female")

Color ("Red", "Blue", "Green")

Source: https://allisonhorst.com/everything-else

Categoric variables: Binary

> Categoric variables can also be binary

- They can be of either types - **character**, **logical** or **numeric**
- They *do not* have a natural ordering
- They take only two mutually exclusive values

<u>Examples</u>

"Yes", "No"

"TRUE", "FALSE"

1, 0

Source: https://allisonhorst.com/everything-else

Working with variables in R

a. character

```
x <- 'A'
```

b. character (string)

```
x <- "Apple"
```

c. logical

```
x <- FALSE
```

d. integer

```
x <- 5L
```

e. numeric (double)

```
x <- 5
```

f. complex

```
x <- 1i
```

Notice how numeric assignments in d and e differ

Working with variables in R

```
x <- 'A'
typeof(x)
```

```
## [1] "character"
```

```
x <- "Apple"
typeof(x)
```

```
## [1] "character"
```

```
x <- FALSE
typeof(x)
```

```
## [1] "logical"
```

```
x <- 5L
typeof(x)
```

```
## [1] "integer"
```

```
x <- 5
typeof(x)
```

```
## [1] "double"
```

```
x <- 1i
typeof(x)
```

```
## [1] "complex"
```

Need for data types

Example: Cat lovers

A survey asked respondents their name and number of cats. The instructions said to enter the number of cats as a numerical value.

```
cat_lovers <- read_csv("cat-lovers.csv")
```

```
## # A tibble: 60 × 3
##    name           number_of_cats handedness
##    <chr>          <chr>          <chr>
##  1 Bernice Warren 0              left
##  2 Woodrow Stone  0              left
##  3 Willie Bass    1              left
##  4 Tyrone Estrada 3              left
##  5 Alex Daniels   3              left
##  6 Jane Bates     2              left
##  7 Latoya Simpson 1              left
##  8 Darin Woods    1              left
##  9 Agnes Cobb     0              left
## 10 Tabitha Grant  0              left
## # ℹ 50 more rows
```

Example: Cat lovers

```
mean(cat_lovers$number_of_cats)
```

```
## Warning in mean.default(cat_lovers$number_of_cats): argument is not numeric or
## logical: returning NA
```

```
## [1] NA
```

Example: Cat lovers (Continued)

```
?mean
```

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An `R` object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical evaluating to `TRUE` or `FALSE` indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Example: Cat lovers - Coercion

```
mean(as.integer(cat_lovers$number_of_cats))
```

```
## Warning in mean(as.integer(cat_lovers$number_of_cats)): NAs introduced by
## coercion
```

```
## [1] NA
```

Example: Cat lovers - Investigation

```
cat_lovers$number_of_cats
```

```
##  [1] "0"
##  [2] "0"
##  [3] "1"
##  [4] "3"
##  [5] "3"
##  [6] "2"
##  [7] "1"
##  [8] "1"
##  [9] "0"
## [10] "0"
## [11] "0"
## [12] "0"
## [13] "1"
## [14] "3"
## [15] "3"
## [16] "2"
## [17] "1"
## [18] "1"
## [19] "0"
## [20] "0"
## [21] "1"
```

Example: Cat lovers - Investigation (Continued)

```
as.integer(cat_lovers$number_of_cats)
```

```
## Warning: NAs introduced by coercion
```

```
##  [1]  0  0  1  3  3  2  1  1  0  0  0  0  1  3  3  2  1  1  0  0  1  1  0  0  4
## [26]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  3  3  2  1 NA  0  0
## [51]  1  0  1 NA  1  1  1  0  0  2
```

II. Data Structures

What are data structures

- They are a collection of values, of one or more types
- Data structures in R include the below types,
  - (atomic) vectors
  - lists
  - matrix
  - factors
- Among them, lists and vectors are the most common and basic data structures in R
- They are pretty much the workhorses of R

Note: Imagine a wallet that can store currencies of one or more nationalities

Vectors & their types

- A vector is a collection of elements of the same type
- They could be of type, character, logical, integer or double
- Let us create an empty vector `x`

```
# An empty vector
x <- vector()
```

- By default, the *type* of an empty vector is logical

```
# Type of the empty vector
typeof(x)
```

```
## [1] "logical"
```

Vectors & their types

One can be more explicit and create vectors of the needed *type*

- Different ways to create vectors of *type*, **logical**

```
# Different ways to create vectors: method
x<-vector("logical",length=5)

# Different ways to create vectors: method
x<-logical(5)

# Different ways to create vectors: method
x<-c(TRUE,FALSE,TRUE,FALSE,TRUE)

typeof(x)
```

```
## [1] "logical"
```

- Different ways to create vectors of *type*, **character**

```
# Different ways to create vectors: method
x<-vector("character",length=5)

# Different ways to create vectors: method
x<-character(5)

# Different ways to create vectors: method
x<-c('A','b','r','q')

typeof(x)
```

```
## [1] "character"
```

Vectors & their types

One can be more explicit and create vectors of the needed *type*

- Different ways to create vectors of *type*, **integer**

```
# Different ways to create vectors: method
x<-vector("integer",length=5)

# Different ways to create vectors: method
x<-integer(5)

# Different ways to create vectors: method
x<-c(1,2,3,4,5)

# Different ways to create vectors: method
x<-seq(from=1,to=5,by=0.1)

# Different ways to create vectors: method
x<-1:5

typeof(x)
```

```
## [1] "integer"
```

- Different ways to create vectors of *type*, **double**

```
# Different ways to create vectors: method
x<-vector("double",length=5)

# Different ways to create vectors: method
x<-double(5)

# Different ways to create vectors: method
x<-c(1.787,0.63573,2.3890)

typeof(x)
```

```
## [1] "double"
```

Vectors: mixed types

If elements of the vector differ in their types,

- R will convert the vector to accommodate all the element types
- This automatic conversion by R from one type to another is called **coercion**
- When R converts the type based on its contents it is called, **implicit** coercion
- Forcing conversion manually is called, **explicit** coercion

Implicit coercion

Example 1: Automatic conversion of *types* by R from double to character

```
# Create a vector
x <- c(1.8)
# Check the type of x
typeof(x)
```

```
## [1] "double"
```

```
# Add a character to the vector
x <- c(x,'a')
# Check the type of x
typeof(x)
```

```
## [1] "character"
```

Note: how `c()` is used to insert elements to a vector

Implicit coercion

Example 2: Automatic conversion of *types* by R from logical to double

```
# Create a vector
x <- c(TRUE)
# Check the type of x
typeof(x)
```

```
## [1] "logical"
```

```
# Add a number to the vector
x <- c(x,2)
# Check the type of x
typeof(x)
```

```
## [1] "double"
```

Note: how `c()` is used to insert elements to a vector

Implicit coercion

Example 3: Automatic conversion of *types* by R from character to character

```
# Create a vector
x <- c('a')
# Check the type of x
typeof(x)
```

```
## [1] "character"
```

```
# Add a logical value to the vector
x <- c(x,TRUE)
# Check the type of x
typeof(x)
```

```
## [1] "character"
```

Note: how `c()` is used to insert elements to a vector

Implicit coercion

Example 4: Automatic conversion of *types* by R from integer to double

```
# Create a vector
x <- c(1L)
# Check the type of x
typeof(x)
```

```
## [1] "integer"
```

```
# Add a number to the vector
x <- c(x,2)
# Check the type of x
typeof(x)
```

```
## [1] "double"
```

Note: how `c()` is used to insert elements to a vector

Explicit coercion

Example 1: Explicit coercion from integer to character

```
# Create a vector
x <- c(1L)
# Check the type of x
typeof(x)
```

```
## [1] "integer"
```

```
# Convert the vector to type character
x <- as.character(x)
# Check the type of x
typeof(x)
```

```
## [1] "character"
```

Note: how `as.character()` is used for conversion

Explicit coercion

Example 2: Explicit coercion from character to double

```
# Create a vector
x <- c('A')
# Check the type of x
typeof(x)
```

```
## [1] "character"
```

```
# Convert the vector to type double
x <- as.numeric(x)
# Check the type of x
typeof(x)
```

```
## [1] "double"
```

Note: how `as.numeric()` is used for conversion

Accessing elements of a vector

Consider the following vector,

```
# Create a vector
x <- c(1,10,9,8,1,3,5)
```

a. Accessing the elements by index: one element

```
# One index
x[3]
```

```
## [1] 9
```

b. Accessing the elements by index: many elements

```
# Consecutive indices
x[2:4]
```

```
## [1] 10  9  8
```

```
# Non-consecutive indices
x[c(1,3,5)]
```

```
## [1] 1 9 1
```

Accessing elements of a vector

Consider the following vector,

```
# Create a vector
x <- c(1,10,9,8,1,3,5)
```

c. Accessing elements using a logical vector

```
# Use of logical vector
x[c(TRUE,FALSE,FALSE,TRUE,FALSE,FALSE,TRUE)]
```

```
## [1] 1 8 5
```

d. Accessing elements using conditional operator

```
# Use of conditional operator
x[x<10]
```

```
## [1] 1 9 8 1 3 5
```

Examining vectors

a. Number of elements in a vector

```
# length of the vector
length(x)
```

```
## [1] 7
```

b. *type* of the object

```
# class of the vector
typeof(x)
```

```
## [1] "double"
```

c. Display the structure of the object

```
# structure of the vector
str(x)
```

```
##  num [1:7] 1 10 9 8 1 3 5
```

- It is of class num, **numeric**
- It has 7 elements, [1:7]

Lists

> A list is a special vector whose elements can be of varied types

a. Creating a list

```
# Initialise a list
x<-list(1,"a",0.289,TRUE)
```

b. Conversion to a list: vector to list

```
# Initialise a vector
x<-c(1,2,3,4,10)
# Convert to a list
x<-as.list(x)
```

Lists

- Elements of a list can be named
- A list is printed on the console along with the name
- Each element of the list starts on a new line

Example

```
# Initialise a named list
my_pie = list(type="key lime", diameter=7, is
my_pie
```

```
## $type
## [1] "key lime"
##
## $diameter
## [1] 7
##
## $is.vegetarian
## [1] TRUE
```

Lists

a. Print names of the list

```
# Elicit names of the list
names(my_pie)
```

```
## [1] "type"           "diameter"      "is.vege
```

b. Access elements of the list by their name

```
# Retrieve the element named type
my_pie$type
```

```
## [1] "key lime"
```

c. Access elements of the list using `[]`: returns a list

```
# Retrieve a truncated list
my_pie["type"]
```

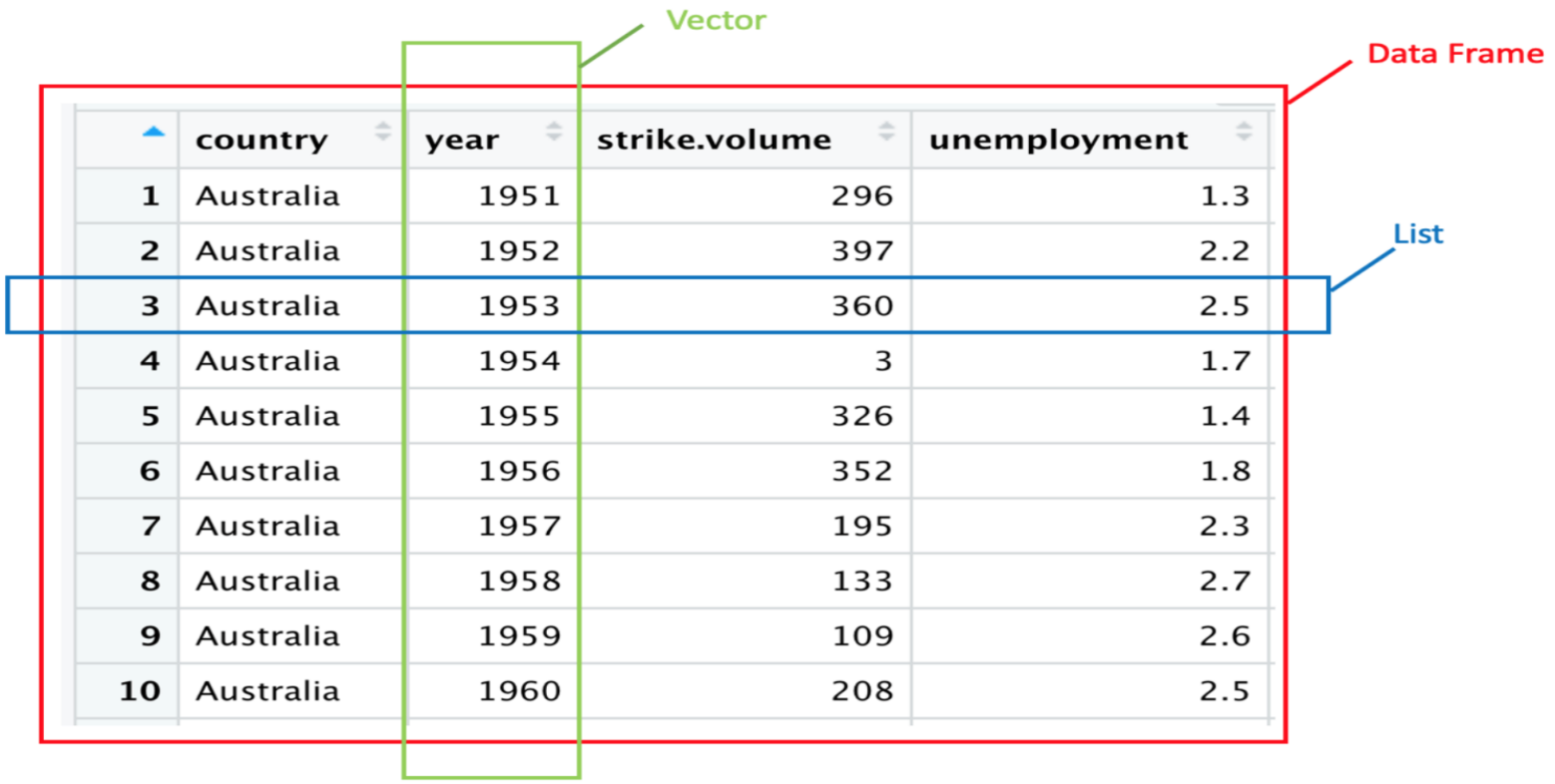
```
## $type
## [1] "key lime"
```

d. Access elements of the list using `[[]]`: returns the element

```
# Retrieve the element named type
my_pie[["type"]]
```

```
## [1] "key lime"
```

Data-set overview



	country	year	strike.volume	unemployment
1	Australia	1951	296	1.3
2	Australia	1952	397	2.2
3	Australia	1953	360	2.5
4	Australia	1954	3	1.7
5	Australia	1955	326	1.4
6	Australia	1956	352	1.8
7	Australia	1957	195	2.3
8	Australia	1958	133	2.7
9	Australia	1959	109	2.6
10	Australia	1960	208	2.5

III. Exploring a data-set

Let us explore a data-set

Example data-set

- Let us consider a data-set, **Lending club**
- The data-set is from a platform that enables loans between individuals
- Not all requests are treated equally;
  - Approval of loan is dependent on the borrower's ability to repay
- The data-set has a compilation of all sanctioned loans
  - There are no loan applications

Lending club: packages

- Install the package containing the data-set

```
# Install package
install.packages("openintro")
```

- Load the installed package

```
# Load package
library(openintro)
```

```
## Loading required package: airports
```

```
## Loading required package: cherryblossom
```

```
## Loading required package: usdata
```

Lending club: packages

- Load `tidyverse` package to manipulate the data-set

```
# Load package
library(tidyverse)
```

Lending club: overview

An overview of the contents of the data-set

```
# Catch a glimpse of the data-set
glimpse(loans_full_schema)
```

```
## Rows: 10,000
## Columns: 55
## $ emp_title                    <chr> "global config engineer ", "warehouse…
## $ emp_length                   <dbl> 3, 10, 3, 1, 10, NA, 10, 10, 10, 3, 1…
## $ state                        <fct> NJ, HI, WI, PA, CA, KY, MI, AZ, NV, I…
## $ homeownership                <fct> MORTGAGE, RENT, RENT, RENT, RENT, OWN…
## $ annual_income                <dbl> 90000, 40000, 40000, 30000, 35000, 34…
## $ verified_income              <fct> Verified, Not Verified, Source Verifi…
## $ debt_to_income               <dbl> 18.01, 5.04, 21.15, 10.16, 57.96, 6.4…
## $ annual_income_joint          <dbl> NA, NA, NA, NA, 57000, NA, 155000, NA…
## $ verification_income_joint    <fct> , , , , Verified, , Not Verified, , ,…
## $ debt_to_income_joint         <dbl> NA, NA, NA, NA, 37.66, NA, 13.12, NA,…
## $ delinq_2y                    <int> 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0…
## $ months_since_last_delinq     <int> 38, NA, 28, NA, NA, 3, NA, 19, 18, NA…
## $ earliest_credit_line         <dbl> 2001, 1996, 2006, 2007, 2008, 1990, 2…
## $ inquiries_last_12m           <int> 6, 1, 4, 0, 7, 6, 1, 1, 3, 0, 4, 4, 8…
## $ total_credit_lines           <int> 28, 30, 31, 4, 22, 32, 12, 30, 35, 9,…
## $ open_credit_lines            <int> 10, 14, 10, 4, 16, 12, 10, 15, 21, 6,…
```

Lending club: Numeric variables

Let us select some variables of *type* Numeric,

a. pass the data-set through a pipe operator (`%>%`)

b. `select()` variables of interest

- `paid_total`
- `term`
- `interest_rate`
- `annual_income`
- `paid_late_fees`
- `debt_to_income`

```
loans <- loans_full_schema %>% # <-- pipe ope
  select(paid_total, term, interest_rate,
         annual_income,paid_late_fees,debt_to_
glimpse(loans)
```

```
## Rows: 10,000
## Columns: 6
## $ paid_total     <dbl> 1999.330, 499.120, 28
## $ term           <dbl> 60, 36, 36, 36, 36, 3
## $ interest_rate  <dbl> 14.07, 12.61, 17.09,
## $ annual_income  <dbl> 90000, 40000, 40000,
## $ paid_late_fees <dbl> 0, 0, 0, 0, 0, 0, 0,
## $ debt_to_income <dbl> 18.01, 5.04, 21.15, 1
```

Lending club: Numeric variables

Variable	Description
paid_total	Repaid loan amount, in US dollars
term	The length of the loan, which is always set as a whole number of months
interest_rate	Interest rate on the loan, in an annual percentage
annual_income	Borrower's annual income, including any second income, in US dollars
paid_late_fees	Penalty paid for defaulting on payment of interest, in US dollars
debt_to_income	Debt-to-income ratio

Lending club: classification of Numeric variables

Variable	Type
paid_total	Continuous
term	Discrete
interest_rate	Continuous
annual_income	Continuous
paid_late_fees	Continuous
debt_to_income	Continuous

Lending club: overview revisited

```
# overview of the data-set
glimpse(loans_full_schema)
```

```
## Rows: 10,000
## Columns: 55
## $ emp_title                    <chr> "global config engineer ", "warehouse…
## $ emp_length                   <dbl> 3, 10, 3, 1, 10, NA, 10, 10, 10, 3, 1…
## $ state                        <fct> NJ, HI, WI, PA, CA, KY, MI, AZ, NV, I…
## $ homeownership                <fct> MORTGAGE, RENT, RENT, RENT, RENT, OWN…
## $ annual_income                <dbl> 90000, 40000, 40000, 30000, 35000, 34…
## $ verified_income              <fct> Verified, Not Verified, Source Verifi…
## $ debt_to_income               <dbl> 18.01, 5.04, 21.15, 10.16, 57.96, 6.4…
## $ annual_income_joint          <dbl> NA, NA, NA, NA, 57000, NA, 155000, NA…
## $ verification_income_joint    <fct> , , , , Verified, , Not Verified, , ,…
## $ debt_to_income_joint         <dbl> NA, NA, NA, NA, 37.66, NA, 13.12, NA,…
## $ delinq_2y                    <int> 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0…
## $ months_since_last_delinq     <int> 38, NA, 28, NA, NA, 3, NA, 19, 18, NA…
## $ earliest_credit_line         <dbl> 2001, 1996, 2006, 2007, 2008, 1990, 2…
## $ inquiries_last_12m           <int> 6, 1, 4, 0, 7, 6, 1, 1, 3, 0, 4, 4, 8…
## $ total_credit_lines           <int> 28, 30, 31, 4, 22, 32, 12, 30, 35, 9,…
## $ open_credit_lines            <int> 10, 14, 10, 4, 16, 12, 10, 15, 21, 6,…
## $ total_credit_limit           <int> 70795, 28800, 24193, 25400, 69839, 42…
## $ total_credit_utilized        <int> 38767, 4321, 16000, 4997, 52722, 3898…
```

Lending club: Categoric variables

Let us select some variables of *type* Categoric,

a. pass the data-set through a pipe operator (`%>%`)

b. `select()` variables of interest

- `grade`
- `state`
- `homeownership`
- `disbursement_method`

```
loans <- loans_full_schema %>% # <-- pipe ope
  select(grade,state,homeownership,disburseme
glimpse(loans)
```

```
## Rows: 10,000
## Columns: 4
## $ grade                <fct> C, C, D, A, C,
## $ state                <fct> NJ, HI, WI, PA,
## $ homeownership        <fct> MORTGAGE, RENT,
## $ disbursement_method <fct> Cash, Cash, Cas
```

Lending club: Categoric variables

Variable	Description
grade	Loan grade, which takes values A through G and represents the quality of the loan and its likelihood of being repaid
state	US state where the borrower resides
homeownership	Indicates whether the person owns, owns but has a mortgage, or rents
disbursement	Mode of loan disbursement

Lending club: classification of Categoric variables

Variable	Type
grade	Ordinal
state	Nominal
homeownership	Nominal
disbursement	Nominal

Thanks!

Slides created via the R packages:

xaringan
gadenbuie/xaringanthemer.

Week-4: Manipulating data

NM2207: Computational Media Literacy

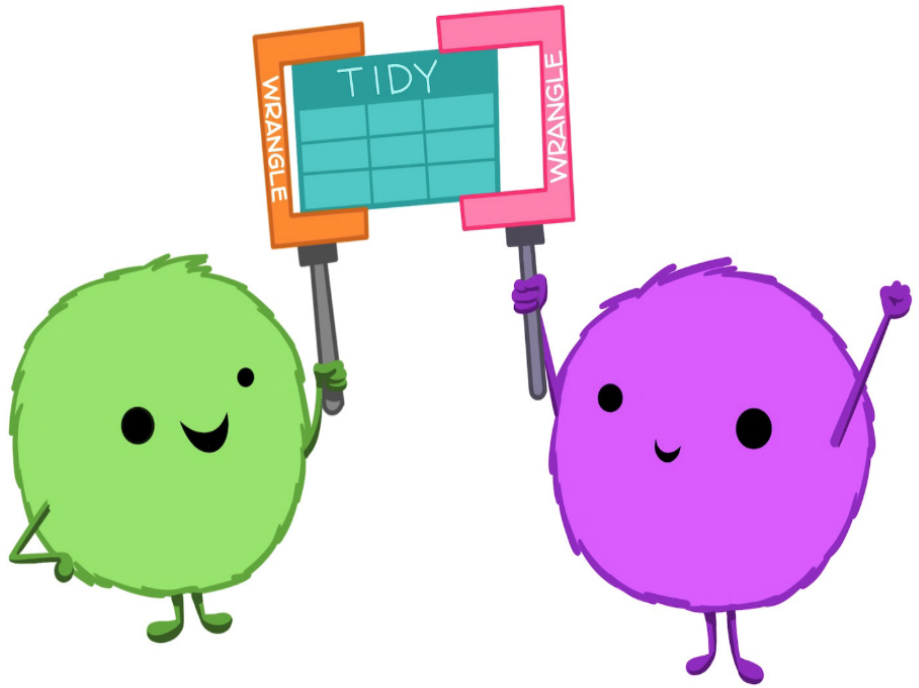
Narayani Vedam, Ph.D.

Department of Communications and New Media

NUS National University of Singapore | Faculty of Arts & Social Sciences

This week

Table of contents

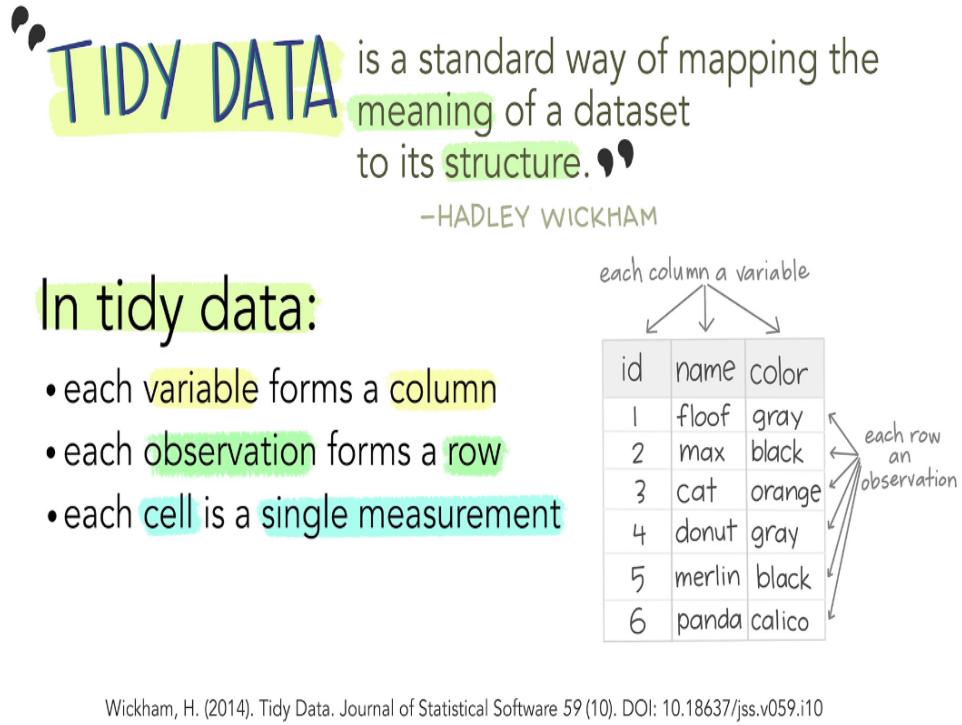


Source: Allison Horst (https://allisonhorst.com/other-r-fun)

I. Tidy data

Tidy data: characteristics

- Every column is a variable
- Every row is an observation
- Each type of observational unit forms a table



Wickham, H. (2014). Tidy Data. Journal of Statistical Software 59 (10). DOI: 10.18637/jss.v059.i10

Source: Allison Horst (https://allisonhorst.com/other-r-fun)

Tidy data: example

country <chr>	year <int>	cases <int>	population <int>
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

6 rows

Observation

Variable

Table

Table 1: Illustration of Tidy Data

Tidy data: what makes data untidy?

Look closely! the table has the same variables as before, but in a different *format*.

country <chr>	year <int>	type <chr>	count <int>
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272

1-10 of 12 rows Previous 1 2 Next

Table 2: Illustration of data with non-tidy structure

Tidy data: what makes data untidy?

Multiple variables (year, months) in columns!

Airplanes on Hand in the AAF, By Major Type: Jul 1939 to Aug 1945

End of Month	Total	Very Heavy Bombers	Heavy Bombers	Medium Bombers	Light Bombers	Fighters	Recon-naissance	Transports	Trainers	Communi-cations
1939										
Jul	2,402	-	16	400	276	494	356	118	735	7
Aug	2,440	-	18	414	276	492	359	129	745	7
[Germany invades Poland, 1 Sep 1939]										
Sep	2,473	-	22	428	278	489	359	136	754	7
Oct	2,507	-	27	446	277	490	365	137	758	7
Nov	2,536	-	32	458	275	498	375	136	755	7
Dec	2,546	-	39	464	274	492	378	131	761	7
1940										
Jan	2,588	-	45	466	271	464	409	128	798	7
Feb	2,658	-	49	470	271	458	415	128	860	7
Mar	2,709	-	54	468	267	453	415	125	920	7
Apr	2,806	-	54	468	263	451	416	125	1,022	7
May	2,906	-	54	470	259	459	410	124	1,123	7
Jun	2,966	-	54	478	166	477	414	127	1,243	7
[France surrenders to Germany, 25 Jun 1940] [Battle of Britain begins, 10 July 1940]										
Jul	3,102	-	56	483	161	500	410	128	1,357	7
Aug	3,295	-	65	485	158	539	407	128	1,506	7

Table 3: Example of non-tidy dataframe (source: Army Air Forces Statistical Digest, WW II)

Tidy data: what makes data untidy?

Observations are in columns!

	A	AA	AB	AC	AD	AE	AF	AG	AH
1	Estimated HIV Prevalence% - (Ages 15-49)	2004	2005	2006	2007	2008	2009	2010	2011
2	Abkhazia								
3	Afghanistan						0.06	0.06	0.06
4	Akrotiri and Dhekelia								
5	Albania								
6	Algeria	0.1	0.1	0.1	0.1	0.1			
7	American Samoa								
8	Andorra								
9	Angola	1.9	1.9	1.9	1.9	2	2.1	2.1	2.1
10	Anguilla								
11	Antigua and Barbuda								
12	Argentina	0.4	0.4	0.4	0.4	0.5	0.4	0.4	0.4
13	Armenia	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2
14	Aruba								
15	Australia	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2
16	Austria	0.2	0.2	0.2	0.3	0.3	0.3	0.4	0.4
17	Azerbaijan	0.06	0.06	0.06	0.1	0.1	0.1	0.1	0.1
18	Bahamas	3	3	3	3.1	3.1	2.9	2.8	2.8

Table 4: Example of non-tidy dataframe (source: Gapminder, Estimated HIV prevalence among 15-49 year olds)

Tidy data: what makes data untidy?

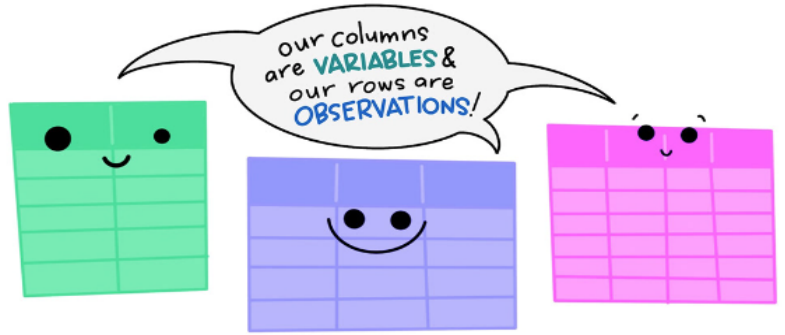
Multiple variables (year, events) in columns!

Subject	United States			
	Estimate	Margin of Error	Percent	Percent Margin of Error
EMPLOYMENT STATUS				
Population 16 years and over	255,797,692	+/-17,051	255,797,692	(X)
In labor force	162,184,325	+/-135,158	63.4%	+/-0.1
Civilian labor force	161,159,470	+/-127,501	63.0%	+/-0.1
Employed	150,599,165	+/-138,066	58.9%	+/-0.1
Unemployed	10,560,305	+/-27,385	4.1%	+/-0.1
Armed Forces	1,024,855	+/-10,363	0.4%	+/-0.1
Not in labor force	93,613,367	+/-126,007	36.6%	+/-0.1
Civilian labor force	161,159,470	+/-127,501	161,159,470	(X)
Unemployment Rate	(X)	(X)	6.6%	+/-0.1
Females 16 years and over	131,092,196	+/-11,187	131,092,196	(X)
In labor force	76,493,327	+/-75,824	58.4%	+/-0.1
Civilian labor force	76,350,498	+/-75,238	58.2%	+/-0.1
Employed	71,451,559	+/-79,007	54.5%	+/-0.1
Own children of the householder under 6 years	22,939,897	+/-14,240	22,939,897	(X)
All parents in family in labor force	14,957,537	+/-36,506	65.2%	+/-0.1
Own children of the householder 6 to 17 years	47,007,147	+/-19,644	47,007,147	(X)
All parents in family in labor force	33,238,793	+/-49,036	70.7%	+/-0.1

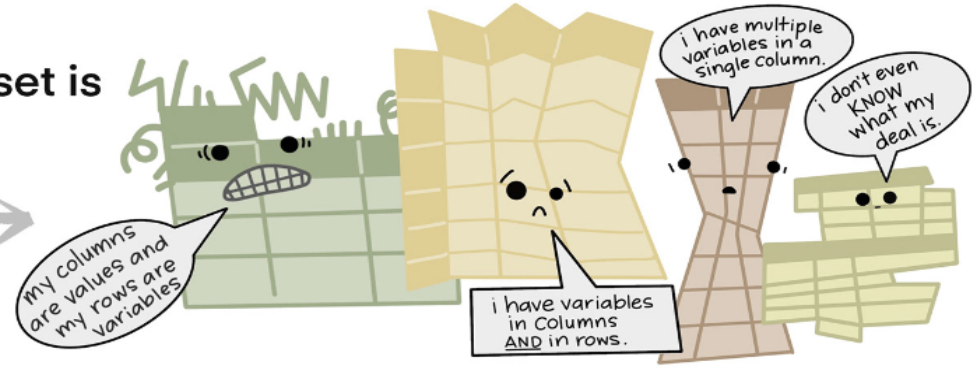
Figure: Example of non-tidy dataframe (Source: GUS Census Fact Finder, General Economic Characteristics, ACS 2017)

Tidy data: what makes data untidy?

The standard structure of tidy data means that "tidy datasets are all alike..."



"...but every messy dataset is messy in its own way."
–HADLEY WICKHAM



Source: Allison Horst (https://allisonhorst.com/other-r-fun)

Ease of manipulation: Tidy vs Non-tidy data

Let us compute the mean number of cases;

- For data in **Tidy** format the code would be,

```
# Tidy data in table1
mean(table1$cases)
```

- For data in **Non-Tidy** format the code would be,

```
# Non-tidy data in table2
mean(table2$count[seq(from=1,to=nrows(table2)
```

- *Which line of code is easier to write?*
- *Which line could you write if you've only looked at the first row of the data?*

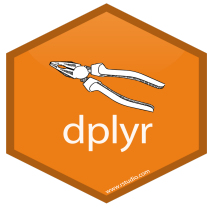
II. Data wrangling

Basics of data wrangling

Once you load data in R, you would want to explore it

- To do so, `dplyr` package is used. It is contained in `tidyverse`
- Some basic functions are,
  - `select()` to pick columns by name
  - `arrange()` to order data
  - `mutate()` to create new columns
  - `filter()` to pick rows matching the criteria

- The first argument (*what goes inside the brackets*) to the functions is always a data-set
- The subsequent arguments indicate operations on the data-set
- The output of the functions is also a data-set
- To store the output, it has to be assigned to a variable



Loading data from files

- **Download** the file with data

- **Save** the file in the same directory as your `.R` file or `.Rmd` file

- We will be using `read_csv()` function to **read** a `.csv` file
  - `.csv` files are text files that can be opened in a text editor
  - They are structured like dataframes or excel sheets with commas separating the values (hence .csv)

Data: hotel booking

- Data of two hotels - a city hotel and a resort
- Each row corresponds to a booking
- It was collected with the aim of developing prediction models to classify a hotel booking's likelihood to be cancelled
    - See: Antonio et al. 2019

```
# Load the package
library(tidyverse)
# Read data from the csv file
hotels <- read_csv("hotels.csv")
```

Data: hotel booking

The raw file looks like this (using Notepad on a Windows PC or TextEdit on Mac)

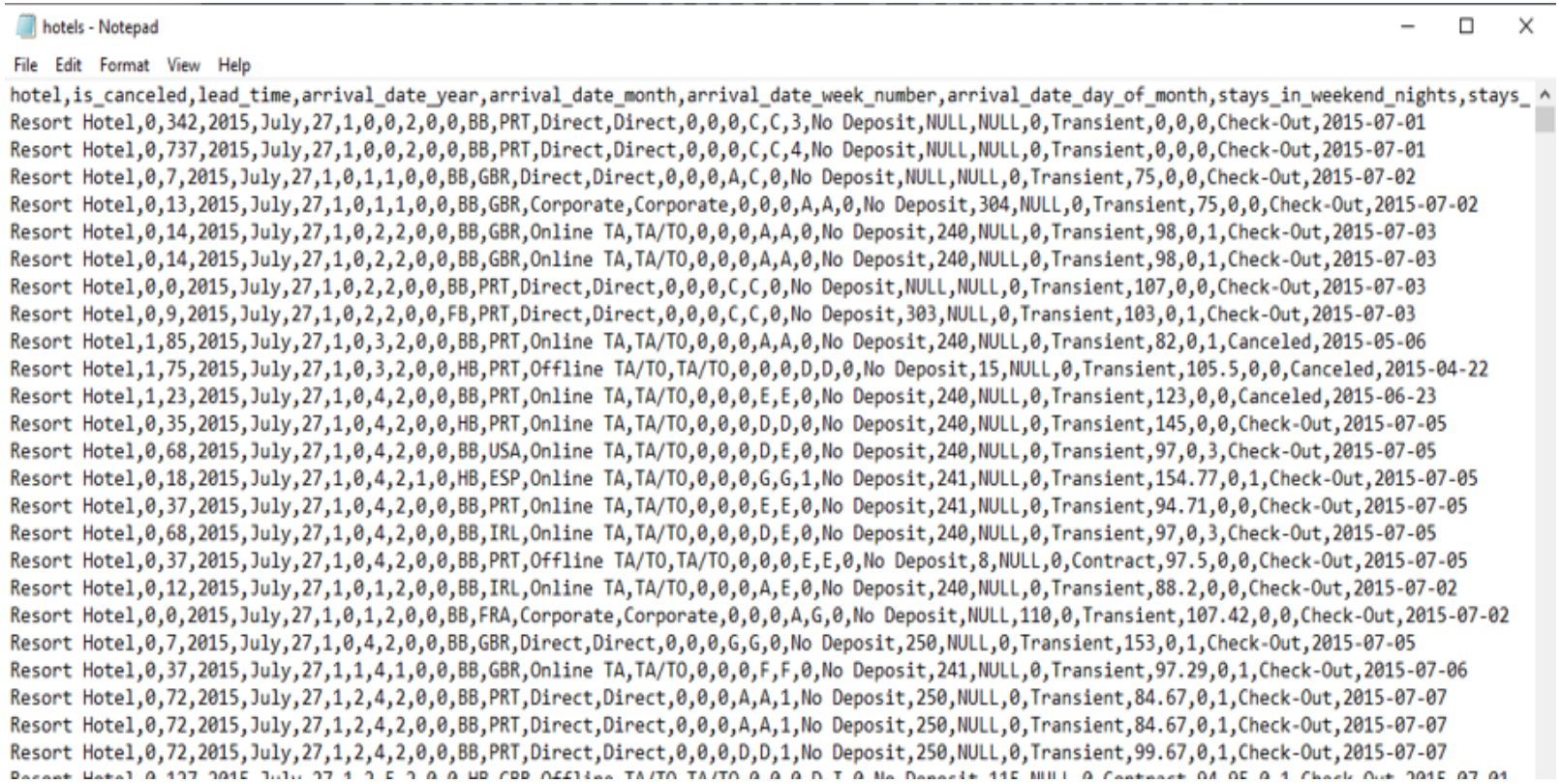
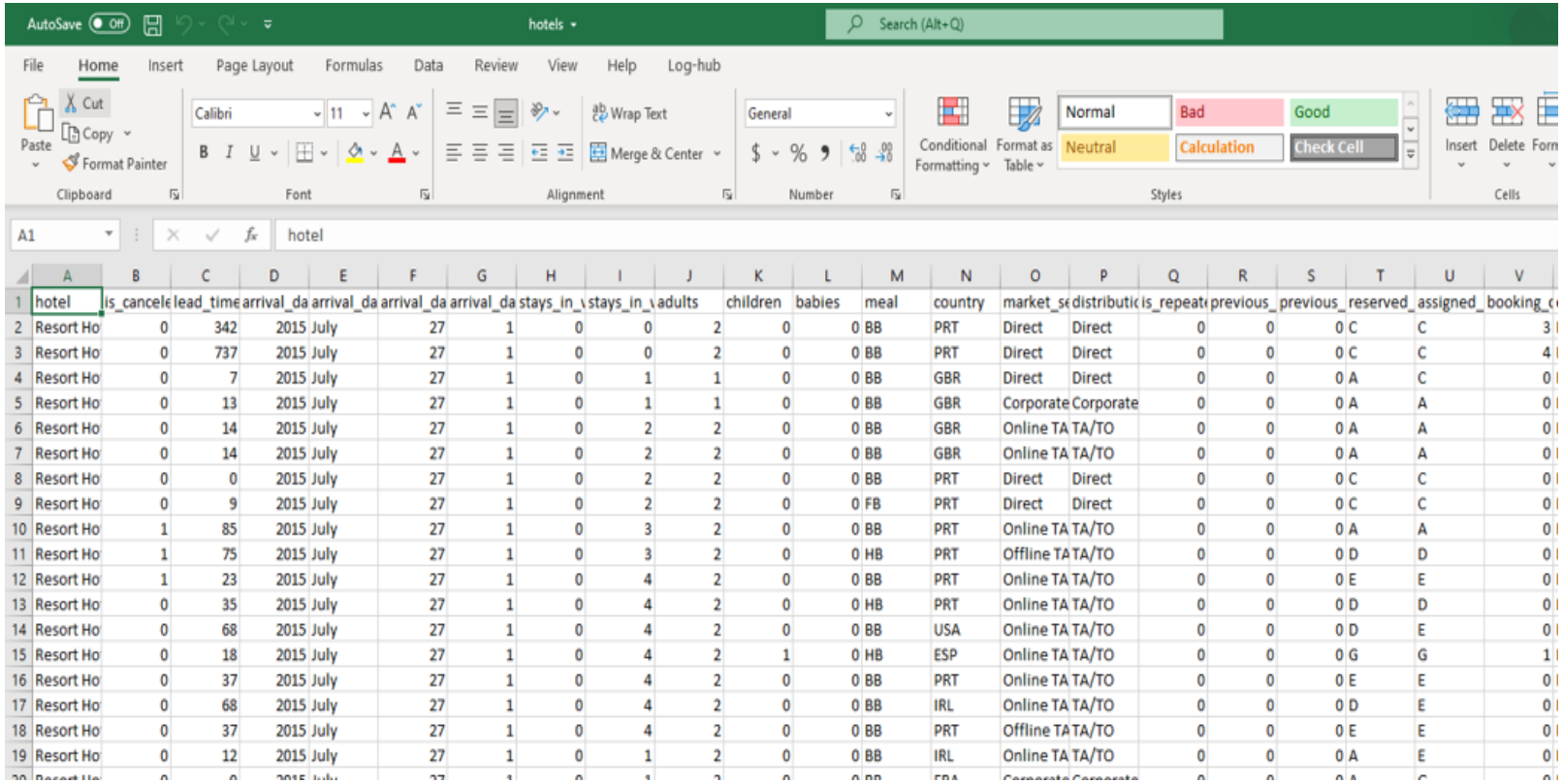


Figure: Hotel booking data

Data: hotel booking

You can open the file in excel too



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	hotel	is_cancele	lead_time	arrival_da	arrival_da	arrival_da	arrival_da	stays_in_v	stays_in_v	adults	children	babies	meal	country	market_se	distributi	is_repeate	previous_	previous_	reserved_	assigned_	booking_c
2	Resort Ho	0	342	2015	July	27	1	0	0	2	0	0	BB	PRT	Direct	Direct	0	0	0	C	C	3
3	Resort Ho	0	737	2015	July	27	1	0	0	2	0	0	BB	PRT	Direct	Direct	0	0	0	C	C	4
4	Resort Ho	0	7	2015	July	27	1	0	1	1	0	0	BB	GBR	Direct	Direct	0	0	0	A	C	0
5	Resort Ho	0	13	2015	July	27	1	0	1	1	0	0	BB	GBR	Corporate	Corporate	0	0	0	A	A	0
6	Resort Ho	0	14	2015	July	27	1	0	2	2	0	0	BB	GBR	Online TA	TA/TO	0	0	0	A	A	0
7	Resort Ho	0	14	2015	July	27	1	0	2	2	0	0	BB	GBR	Online TA	TA/TO	0	0	0	A	A	0
8	Resort Ho	0	0	2015	July	27	1	0	2	2	0	0	BB	PRT	Direct	Direct	0	0	0	C	C	0
9	Resort Ho	0	9	2015	July	27	1	0	2	2	0	0	FB	PRT	Direct	Direct	0	0	0	C	C	0
10	Resort Ho	1	85	2015	July	27	1	0	3	2	0	0	BB	PRT	Online TA	TA/TO	0	0	0	A	A	0
11	Resort Ho	1	75	2015	July	27	1	0	3	2	0	0	HB	PRT	Offline TA	TA/TO	0	0	0	D	D	0
12	Resort Ho	1	23	2015	July	27	1	0	4	2	0	0	BB	PRT	Online TA	TA/TO	0	0	0	E	E	0
13	Resort Ho	0	35	2015	July	27	1	0	4	2	0	0	HB	PRT	Online TA	TA/TO	0	0	0	D	D	0
14	Resort Ho	0	68	2015	July	27	1	0	4	2	0	0	BB	USA	Online TA	TA/TO	0	0	0	D	E	0
15	Resort Ho	0	18	2015	July	27	1	0	4	2	1	0	HB	ESP	Online TA	TA/TO	0	0	0	G	G	1
16	Resort Ho	0	37	2015	July	27	1	0	4	2	0	0	BB	PRT	Online TA	TA/TO	0	0	0	E	E	0
17	Resort Ho	0	68	2015	July	27	1	0	4	2	0	0	BB	IRL	Online TA	TA/TO	0	0	0	D	E	0
18	Resort Ho	0	37	2015	July	27	1	0	4	2	0	0	BB	PRT	Offline TA	TA/TO	0	0	0	E	E	0
19	Resort Ho	0	12	2015	July	27	1	0	1	2	0	0	BB	IRL	Online TA	TA/TO	0	0	0	A	E	0

Figure: Hotel booking data

Hotel booking: look at the variables

- `names()` is a function
- `hotels` is the argument

```
names(hotels)
```

```
##  [1] "hotel"                          "is_canceled"
##  [3] "lead_time"                      "arrival_date_year"
##  [5] "arrival_date_month"             "arrival_date_week_number"
##  [7] "arrival_date_day_of_month"      "stays_in_weekend_nights"
##  [9] "stays_in_week_nights"           "adults"
## [11] "children"                       "babies"
## [13] "meal"                           "country"
## [15] "market_segment"                 "distribution_channel"
## [17] "is_repeated_guest"              "previous_cancellations"
## [19] "previous_bookings_not_canceled" "reserved_room_type"
## [21] "assigned_room_type"             "booking_changes"
## [23] "deposit_type"                   "agent"
## [25] "company"                        "days_in_waiting_list"
## [27] "customer_type"                  "adr"
## [29] "required_car_parking_spaces"    "total_of_special_requests"
## [31] "reservation_status"             "reservation_status_date"
```

Hotel booking: data overview

```
glimpse(hotels)
```

```
## Rows: 119,390
## Columns: 32
## $ hotel                         <chr> "Resort Hotel", "Resort Hotel", "Resort…
## $ is_canceled                   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, …
## $ lead_time                     <dbl> 342, 737, 7, 13, 14, 14, 0, 9, 85, 75, …
## $ arrival_date_year             <dbl> 2015, 2015, 2015, 2015, 2015, 2015, 201…
## $ arrival_date_month            <chr> "July", "July", "July", "July", "July",…
## $ arrival_date_week_number      <dbl> 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,…
## $ arrival_date_day_of_month     <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
## $ stays_in_weekend_nights       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ stays_in_week_nights          <dbl> 0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 4, 4, 4, …
## $ adults                        <dbl> 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, …
## $ children                      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ babies                        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ meal                          <chr> "BB", "BB", "BB", "BB", "BB", "BB", "BB…
## $ country                       <chr> "PRT", "PRT", "GBR", "GBR", "GBR", "GBR…
## $ market_segment                <chr> "Direct", "Direct", "Direct", "Corporat…
## $ distribution_channel          <chr> "Direct", "Direct", "Direct", "Corporat…
## $ is_repeated_guest             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ previous_cancellations        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ previous_bookings_not_canceled <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
```

III. Choosing rows or columns

Selecting columns

- `select()` let's us select specific columns from a data-set

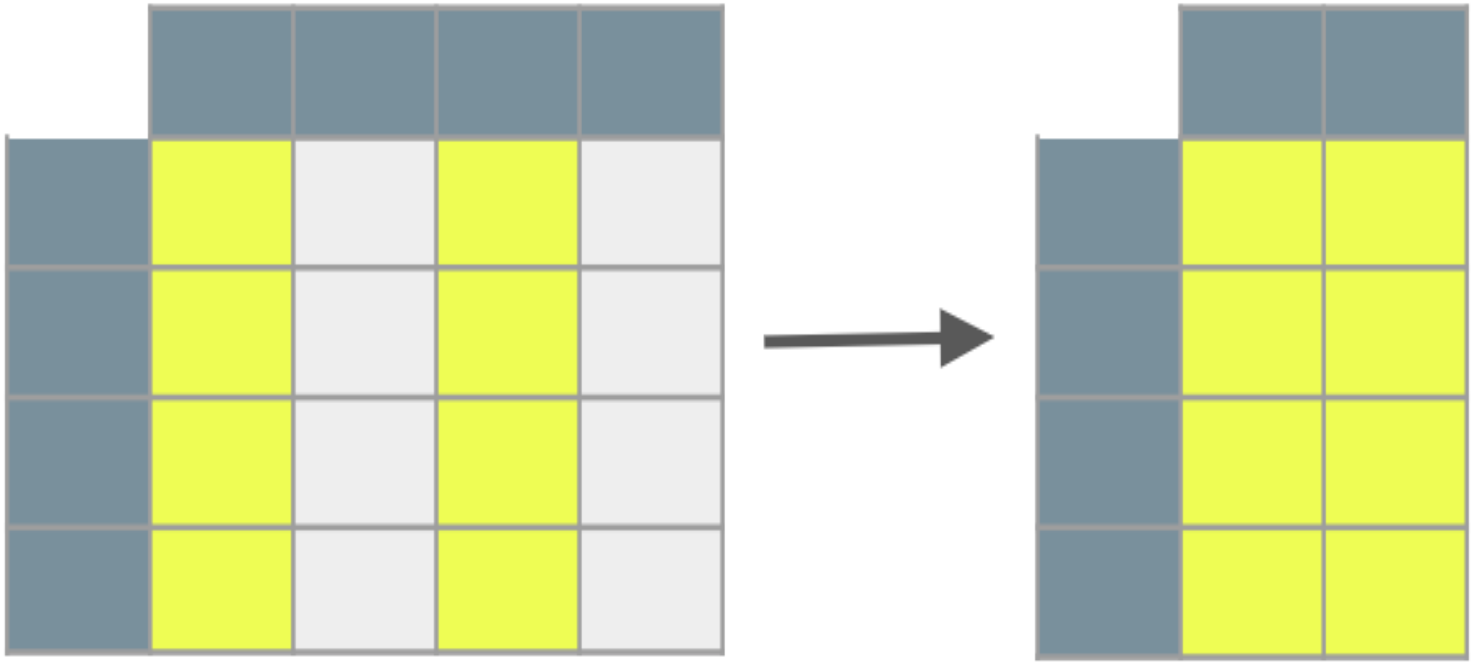


Figure: Selecting columns

Hotel booking: select a single column

- Start with the verb

```
select(              # <-- verb
        hotels,
         lead_time
            )
```

- First argument is the data-set

```
select(
        hotels,     # <-- data-set
         lead_time
            )
```

- Second argument is the column we want to select

```
select(
       hotels,
          lead_time    # <-- chosen column/variable
            )
```

Hotel booking: select a single column

- View only `lead_time` (number of days between booking and arrival)

```
select(hotels, lead_time)
```

```
## # A tibble: 119,390 × 1
##    lead_time
##        <dbl>
##  1       342
##  2       737
##  3         7
##  4        13
##  5        14
##  6        14
##  7         0
##  8         9
##  9        85
## 10        75
## # ℹ 119,380 more rows
```

- **Remember:** `dplyr` functions always expect a dataframe and the output is also a dataframe

Hotel booking: selecting multiple columns

- View `lead_time`, `agent` and `market_segment`

```
select(hotels, lead_time,agent,market_segment)
```

```
## # A tibble: 119,390 × 3
##    lead_time agent market_segment
##        <dbl> <chr> <chr>
##  1       342 NULL  Direct
##  2       737 NULL  Direct
##  3         7 NULL  Direct
##  4        13 304   Corporate
##  5        14 240   Online TA
##  6        14 240   Online TA
##  7         0 NULL  Direct
##  8         9 303   Direct
##  9        85 240   Online TA
## 10        75 15    Offline TA/TO
## # ℹ 119,380 more rows
```

- **Note:** you can use `-c("column_names")` to remove specific columns

Arranging columns

`arrange()` let's us reorder data

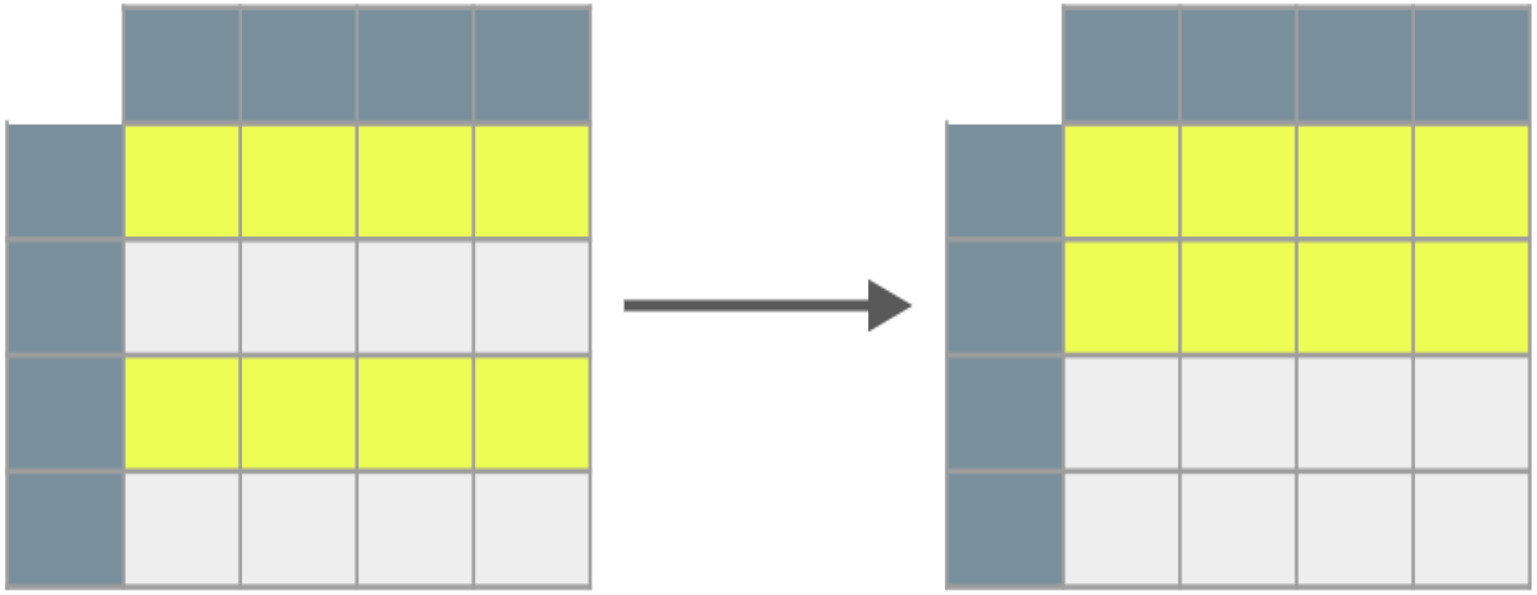


Figure: Arranging columns

Hotel booking: arrange

Let us try reordering entries of the column `lead_time`

- Start with the verb

```
arrange(             # <-- verb
          hotels,
           lead_time
              )
```

- First argument is the data-set

```
arrange(
         hotels,     # <-- data-set
           lead_time
              )
```

- Second argument is the column whose entries we want to reorder

```
arrange(
        hotels,
           lead_time    # <-- chosen column/variable
              )
```

Hotel booking: arrange

- View the rearranged entries of `lead_time`

```
arrange(hotels, lead_time)
```

```
## # A tibble: 119,390 × 32
##    hotel        is_canceled lead_time arrival_date_year arrival_date_month
##    <chr>              <dbl>     <dbl>             <dbl> <chr>
##  1 Resort Hotel           0         0              2015 July
##  2 Resort Hotel           0         0              2015 July
##  3 Resort Hotel           0         0              2015 July
##  4 Resort Hotel           0         0              2015 July
##  5 Resort Hotel           0         0              2015 July
##  6 Resort Hotel           0         0              2015 July
##  7 Resort Hotel           0         0              2015 July
##  8 Resort Hotel           0         0              2015 July
##  9 Resort Hotel           0         0              2015 July
## 10 Resort Hotel           0         0              2015 July
## # ℹ 119,380 more rows
## # ℹ 27 more variables: arrival_date_week_number <dbl>,
## #   arrival_date_day_of_month <dbl>, stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>, children <dbl>, babies <dbl>,
## #   meal <chr>, country <chr>, market_segment <chr>,
```

Hotel booking: arrange

- Original order of the entries of `lead_time`

```
hotels
```

```
## # A tibble: 119,390 × 32
##    hotel        is_canceled lead_time arrival_date_year arrival_date_month
##    <chr>              <dbl>     <dbl>             <dbl> <chr>
##  1 Resort Hotel           0       342              2015 July
##  2 Resort Hotel           0       737              2015 July
##  3 Resort Hotel           0         7              2015 July
##  4 Resort Hotel           0        13              2015 July
##  5 Resort Hotel           0        14              2015 July
##  6 Resort Hotel           0        14              2015 July
##  7 Resort Hotel           0         0              2015 July
##  8 Resort Hotel           0         9              2015 July
##  9 Resort Hotel           1        85              2015 July
## 10 Resort Hotel           1        75              2015 July
## # ℹ 119,380 more rows
## # ℹ 27 more variables: arrival_date_week_number <dbl>,
## #   arrival_date_day_of_month <dbl>, stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>, children <dbl>, babies <dbl>,
## #   meal <chr>, country <chr>, market_segment <chr>,
```

Hotel booking: arrange

- By default, `arrange()` arranges the entries of the chosen column in the **increasing** order
- Re-arranging applies to the entire row and not just the entries of the column
- In other words, the observation (rows) get rearranged so that the entries of `lead_time` are in the ascending order
- To rearrange entries in the descending order, use `desc()`

```
arrange(hotels, desc(lead_time))
```

Hotel booking: select + arrange

- After picking the column/variable `lead_time`, let us say we want to rearrange its entries in the decreasing order

- Step-1 is to pick the desired column, `select(hotels, lead_time)`

```
  select(hotels, lead_time)
```

- Step-2 is to rearrange the output of `select(hotels, lead_time)`

```
  arrange(       # <-- start with the verb
  select(hotels, lead_time), # <-- first argument is the dataframe *
   desc(lead_time) # <--- second argument is the how you want arrange
)                 # i.e. decreasing order of lead_time
```

* *in this case, it is the output of the* `select` *operation.*

Hotel booking: select + arrange

```
select(hotels, lead_time)
```

```
## # A tibble: 119,390 × 1
##    lead_time
##        <dbl>
##  1       342
##  2       737
##  3         7
##  4        13
##  5        14
##  6        14
##  7         0
##  8         9
##  9        85
## 10        75
## # ℹ 119,380 more rows
```

```
arrange(
  select(hotels, lead_time),
desc(lead_time)
)
```

```
## # A tibble: 119,390 × 1
##    lead_time
##        <dbl>
##  1       737
##  2       709
##  3       629
##  4       629
##  5       629
##  6       629
##  7       629
##  8       629
##  9       629
## 10       629
## # ℹ 119,380 more rows
```

Hotel booking: select + arrange

- Did you notice how many functions we used to arrive at a rearranged column?
- A function(s) within another function(s) is called a ***nested function***
- Now, imagine having to do more than ten operations on a data-frame
- How confusing could it get to have functions inside another??!
- You think not? Read on.

```
arrange(                        # <-- Function 1
  select(hotels, lead_time), # <-- Function 2
desc(lead_time) # <-- Function 3
)
```

IV. Combining two or more operations

Nesting operations

- Think of the activities that you might be doing every morning
    - find keys, unlock car, start car, drive to work, park

- As R functions, this would hypothetically look like:

```
park(drive(start_car(find("keys")), to = "work"))
```

- This is very confusing to read, given that the first thing you write, is the last thing you do,
    - `park` then `drive` then `start_car` then `find(keys)`, and then the `to` argument again refers to `drive`, not what is right next to it (i.e. `find`)

- Is there an easier way to tackle this?? **Pipes**

Pipes

- We have already used this operator earlier in our course
- They are a part of `magrittr` package that is automatically loaded when you load `tidyverse`
- `%>%` operator takes the left-hand side and passes it as an input to the right-hand side
- It makes the code easier to read and write
- Let us apply it to the hypothetical scenario from earlier

```
"keys" %>%
  find() %>%
    start_car() %>%
      drive(to = "work") %>%
        park()
```

- *Isn't it easier to read and follow?*

Hotel booking: select + arrange using pipes

```
arrange(
  select(hotels, lead_time),
  desc(lead_time)
)
```

```
## # A tibble: 119,390 × 1
##    lead_time
##        <dbl>
##  1       737
##  2       709
##  3       629
##  4       629
##  5       629
##  6       629
##  7       629
##  8       629
##  9       629
## 10       629
## # ℹ 119,380 more rows
```

```
hotels %>%
  select(lead_time) %>%
  arrange(desc(lead_time))
```

```
## # A tibble: 119,390 × 1
##    lead_time
##        <dbl>
##  1       737
##  2       709
##  3       629
##  4       629
##  5       629
##  6       629
##  7       629
##  8       629
##  9       629
## 10       629
## # ℹ 119,380 more rows
```

Piping versus Layering

- Let us recall visualization using `ggplot2`
  - we created the plot in layers, separated by `+`
- Layering works differently from pipelines of `dplyr`
  - As noted in earlier slides, the output of the previous line of code serves as the input to the next

Piping in `dplyr`

This is incorrect

```
hotels +
  select(hotel,lead_time)
```

```
## Error in select(hotel, lead_time): object 'l
```

This is correct

```
hotels %>%
select(hotel,lead_time)
```

```
## # A tibble: 119,390 × 2
##    hotel        lead_time
##    <chr>            <dbl>
##  1 Resort Hotel       342
##  2 Resort Hotel       737
##  3 Resort Hotel         7
##  4 Resort Hotel        13
##  5 Resort Hotel        14
##  6 Resort Hotel        14
##  7 Resort Hotel         0
##  8 Resort Hotel         9
##  9 Resort Hotel        85
## 10 Resort Hotel        75
## # ℹ 119,380 more rows
```

Layering in `ggplot2`

- This is incorrect

```
ggplot(hotels, aes(x = hotel, fill = deposit_type)) %>%
  geom_bar()
```

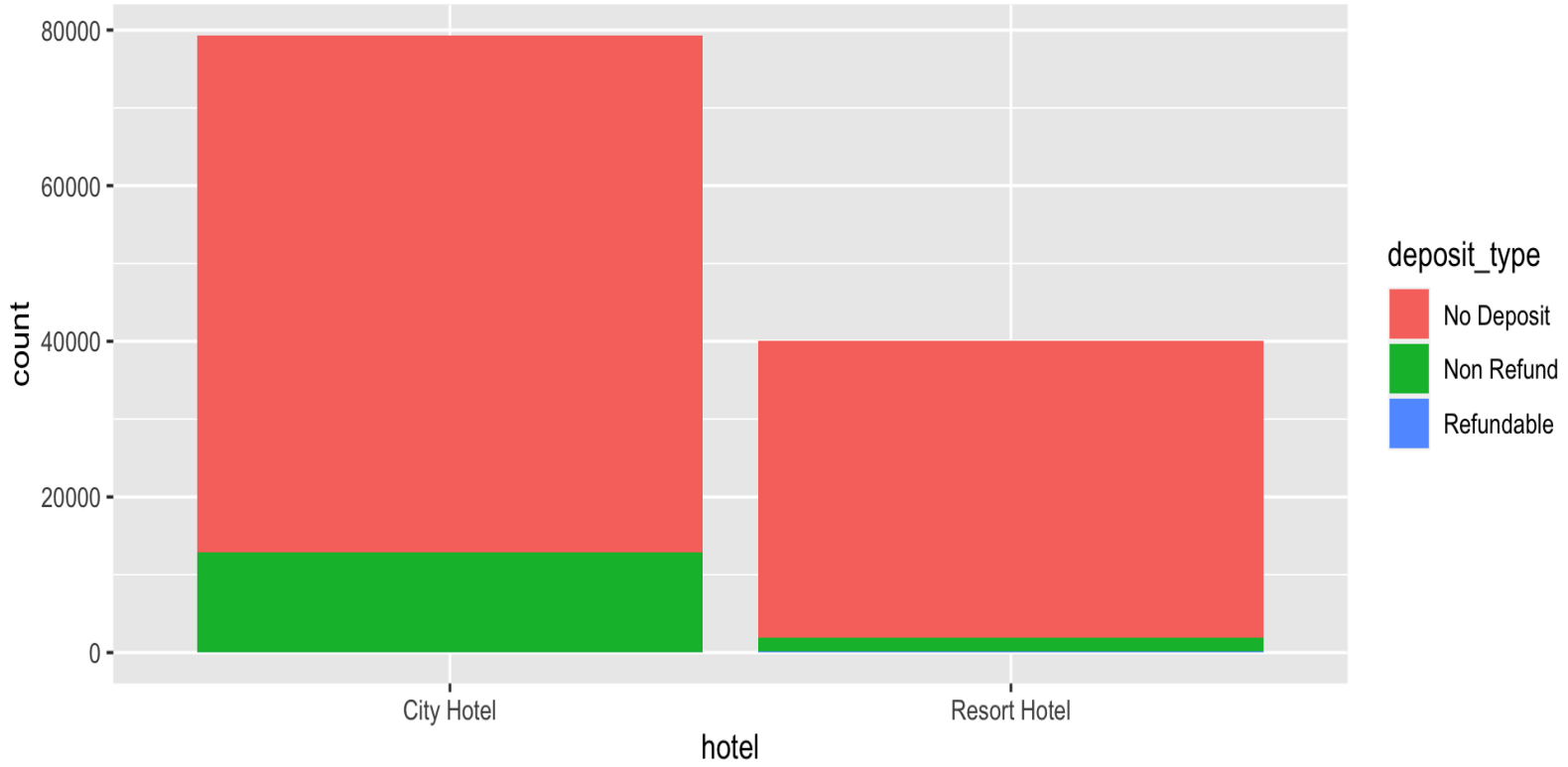
```
## Error in `geom_bar()`:
## ! `mapping` must be created by `aes()`
## ℹ Did you use `%>%` or `|>` instead of `+`?
```

- This is correct

```
ggplot(hotels, aes(x = hotel, fill = deposit_type)) +
  geom_bar()
```

Layering in `ggplot2`

```
ggplot(hotels, aes(x = hotel, fill = deposit_type)) +
  geom_bar()
```



Pick rows that match a condition, `filter()`

- Use `filter()` to pick rows matching a condition

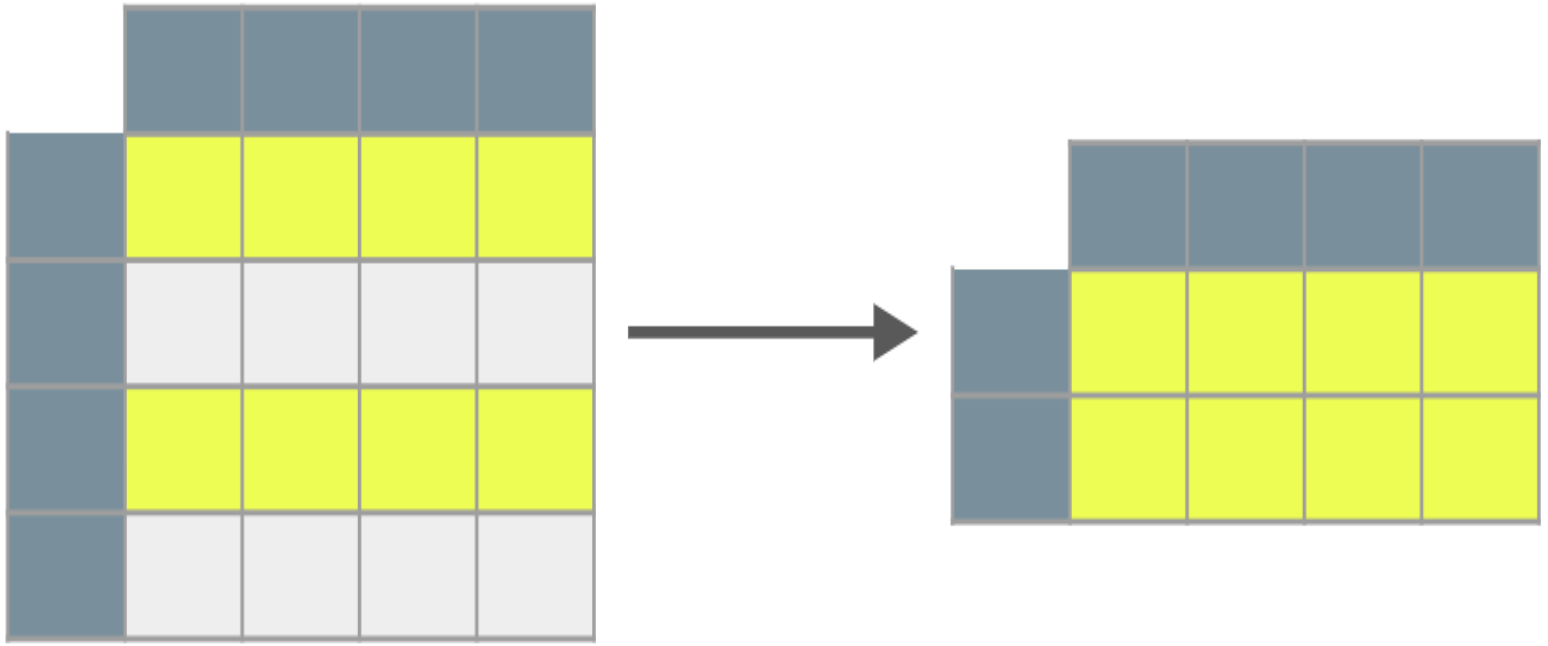


Figure: Selecting rows

Hotel Booking: Pick rows matching a condition using `filter()`

- Start with the verb

```
hotels %>%
   filter(                          # <-- verb 1
    children >= 1
      ) %>%
         select(hotel, children)  # <-- verb 2
```

- Pass the condition, the column/variable on the left of the conditional operator

```
hotels %>%
  filter(
     children >= 1                # <-- condition
      ) %>%
       select(hotel, children)
```

Hotel Booking: Pick rows matching a condition using `filter()`

```
hotels %>%
  filter(children >= 1) %>%
       select(hotel, children)
```

```
## # A tibble: 8,590 × 2
##    hotel        children
##    <chr>           <dbl>
##  1 Resort Hotel        1
##  2 Resort Hotel        2
##  3 Resort Hotel        2
##  4 Resort Hotel        2
##  5 Resort Hotel        1
##  6 Resort Hotel        1
##  7 Resort Hotel        2
##  8 Resort Hotel        2
##  9 Resort Hotel        1
## 10 Resort Hotel        2
## # ℹ 8,580 more rows
```

Hotel Booking: Pick rows matching more than one condition using `filter()`

- Start with the verb

```
hotels %>%
   filter(                          # <-- verb 1
    children >= 1
      ) %>%
         select(hotel, children)   # <-- verb 2
```

- Pass the conditions, separated by `,`

```
hotels %>%
  filter(
    children >= 1,                 # <-- condition 1
    hotel == "City Hotel"          # <-- condition 2
      ) %>%
       select(hotel, children)
```

Hotel Booking: Pick rows matching a condition using `filter()`

```
hotels %>%
  filter(children >= 1,hotel == "City Hotel") %>%
       select(hotel, children)
```

```
## # A tibble: 5,106 × 2
##    hotel      children
##    <chr>         <dbl>
##  1 City Hotel        1
##  2 City Hotel        2
##  3 City Hotel        1
##  4 City Hotel        1
##  5 City Hotel        1
##  6 City Hotel        1
##  7 City Hotel        1
##  8 City Hotel        1
##  9 City Hotel        1
## 10 City Hotel        1
## # ℹ 5,096 more rows
```

Pick rows using `slice()`: No conditional selection

> Choose certain rows/observations without conditions using `slice()`

- Quick recap
  - Choose certain variables/columns using `select()`
  - Conditionally pick rows or observations using `filter()`

Pick rows using `slice()`: No conditional selection

- Start with the verb

```
hotels %>%
  slice(          #<-- verb
        1:5
       )
```

- Pass the indices or the range of indices of the rows to be picked

```
hotels %>%
  slice(
        1:5    #<-- indices
       )
```

Pick rows using `slice()`: No conditional selection

- Using a sequence of indices

```
hotels %>% slice(1:5)
```

```
## # A tibble: 5 × 32
##   hotel        is_canceled lead_time arrival_date_year arrival_date_month
##   <chr>              <dbl>     <dbl>             <dbl> <chr>
## 1 Resort Hotel           0       342              2015 July
## 2 Resort Hotel           0       737              2015 July
## 3 Resort Hotel           0         7              2015 July
## 4 Resort Hotel           0        13              2015 July
## 5 Resort Hotel           0        14              2015 July
## # ℹ 27 more variables: arrival_date_week_number <dbl>,
## #   arrival_date_day_of_month <dbl>, stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>, children <dbl>, babies <dbl>,
## #   meal <chr>, country <chr>, market_segment <chr>,
## #   distribution_channel <chr>, is_repeated_guest <dbl>,
## #   previous_cancellations <dbl>, previous_bookings_not_canceled <dbl>,
## #   reserved_room_type <chr>, assigned_room_type <chr>, …
```

Pick rows using `slice()`: No conditional selection

- Using specific indices

```
hotels %>%
  slice(1,3,5)
```

```
## # A tibble: 3 × 32
##   hotel        is_canceled lead_time arrival_date_year arrival_date_month
##   <chr>              <dbl>     <dbl>             <dbl> <chr>
## 1 Resort Hotel           0       342              2015 July
## 2 Resort Hotel           0         7              2015 July
## 3 Resort Hotel           0        14              2015 July
## # ℹ 27 more variables: arrival_date_week_number <dbl>,
## #   arrival_date_day_of_month <dbl>, stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>, children <dbl>, babies <dbl>,
## #   meal <chr>, country <chr>, market_segment <chr>,
## #   distribution_channel <chr>, is_repeated_guest <dbl>,
## #   previous_cancellations <dbl>, previous_bookings_not_canceled <dbl>,
## #   reserved_room_type <chr>, assigned_room_type <chr>, …
```

Pick unique rows using `distinct()`

- Start with the verb

```
hotels %>%
   distinct(          #<-- verb,
            hotel)
```

- Pass the dataframe

```
 hotels %>%              #<-- dataset
    distinct(
           hotel)
```

- Pass the column or variable whose unique entries are to to be picked

```
hotels %>%
  distinct(
            hotel) #<-- column/variable
```

Pick unique rows using `distinct()`: output

- The output is a dataframe with only those rows with the unique entries of `hotel`

```
hotels %>% distinct(hotel)
```

```
## # A tibble: 2 × 1
##   hotel
##   <chr>
## 1 Resort Hotel
## 2 City Hotel
```

V. Creating new columns

Creating columns with `mutate()`

- Use `mutate()` to create new columns using existing columns

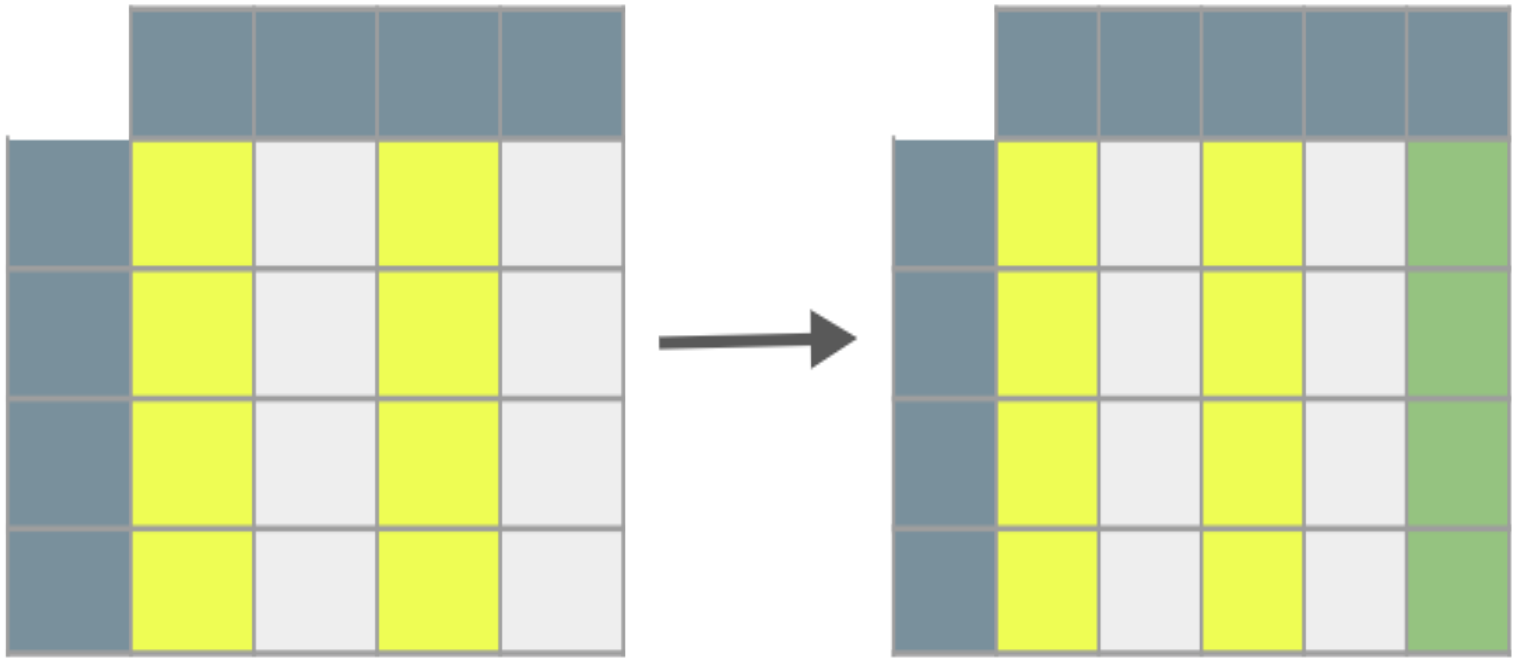


Figure: Create new columns

Creating a single column with `mutate()`

- To create a new column, on the left side of `=` is the name of the new column
- On the right of `=` are the operations on existing columns

```
hotels %>%
   mutate(little_ones = children + babies) %>% #<-- Look here!
    select(hotel, little_ones,children,babies)
```

Creating a single column with `mutate()`

```
hotels %>%
  mutate(little_ones = children + babies) %>%
    select(hotel, little_ones,children,babies)
```

```
## # A tibble: 119,390 × 4
##    hotel        little_ones children babies
##    <chr>              <dbl>    <dbl>  <dbl>
##  1 Resort Hotel           0        0      0
##  2 Resort Hotel           0        0      0
##  3 Resort Hotel           0        0      0
##  4 Resort Hotel           0        0      0
##  5 Resort Hotel           0        0      0
##  6 Resort Hotel           0        0      0
##  7 Resort Hotel           0        0      0
##  8 Resort Hotel           0        0      0
##  9 Resort Hotel           0        0      0
## 10 Resort Hotel           0        0      0
## # ℹ 119,380 more rows
```

Creating multiple columns with `mutate()`

- To create a new column, on the left side of `=` is the name of the new column
- On the right of `=` are the operations on existing columns
- The columns are separated by `,`
- The newer columns can even use a newly created column(s)

```
hotels %>%
  mutate(little_ones = children + babies,                  #<-- New column 1
         average_little_ones = mean(little_ones)) %>%      #<-- New column 2
    select(hotel, little_ones,children,babies)
```

Creating multiple columns with `mutate()`

```
hotels %>%
  mutate(little_ones = children + babies,
         average_little_ones = mean(little_ones)) %>%
    select(hotel, little_ones,children,babies, average_little_ones)
```

```
## # A tibble: 119,390 × 5
##    hotel        little_ones children babies average_little_ones
##    <chr>              <dbl>    <dbl>  <dbl>                <dbl>
##  1 Resort Hotel           0        0      0                   NA
##  2 Resort Hotel           0        0      0                   NA
##  3 Resort Hotel           0        0      0                   NA
##  4 Resort Hotel           0        0      0                   NA
##  5 Resort Hotel           0        0      0                   NA
##  6 Resort Hotel           0        0      0                   NA
##  7 Resort Hotel           0        0      0                   NA
##  8 Resort Hotel           0        0      0                   NA
##  9 Resort Hotel           0        0      0                   NA
## 10 Resort Hotel           0        0      0                   NA
## # ℹ 119,380 more rows
```

VI. More operations with examples

`count()` to get frequencies (counts)

```
hotels %>%
  count(market_segment)
```

```
## # A tibble: 8 × 2
##   market_segment     n
##   <chr>          <int>
## 1 Aviation         237
## 2 Complementary    743
## 3 Corporate       5295
## 4 Direct         12606
## 5 Groups         19811
## 6 Offline TA/TO  24219
## 7 Online TA      56477
## 8 Undefined          2
```

- **Note** that count automatically arranges in alphabetical order

`count()` to get frequencies (counts)

```
hotels %>%
  count(market_segment, sort = TRUE) # <-- decreasing order of counts
```

```
## # A tibble: 8 × 2
##   market_segment     n
##   <chr>          <int>
## 1 Online TA      56477
## 2 Offline TA/TO  24219
## 3 Groups         19811
## 4 Direct         12606
## 5 Corporate       5295
## 6 Complementary    743
## 7 Aviation         237
## 8 Undefined          2
```

`count()` multiple variables

```
hotels %>%
  count(hotel, market_segment)
```

```
## # A tibble: 14 × 3
##    hotel        market_segment     n
##    <chr>        <chr>          <int>
##  1 City Hotel   Aviation         237
##  2 City Hotel   Complementary    542
##  3 City Hotel   Corporate       2986
##  4 City Hotel   Direct          6093
##  5 City Hotel   Groups         13975
##  6 City Hotel   Offline TA/TO  16747
##  7 City Hotel   Online TA      38748
##  8 City Hotel   Undefined          2
##  9 Resort Hotel Complementary    201
## 10 Resort Hotel Corporate       2309
## 11 Resort Hotel Direct          6513
## 12 Resort Hotel Groups          5836
## 13 Resort Hotel Offline TA/TO   7472
## 14 Resort Hotel Online TA      17729
```

`summarise()` for summary statistics

> `summarise()` collapses the dataframe down to a single summary statistic

```
# mean average daily rate for all bookings
hotels %>%
  summarise(mean_adr = mean(adr))
```

```
## # A tibble: 1 × 1
##   mean_adr
##      <dbl>
## 1     102.
```

`summarise()` by variable using `group_by()`

```
# mean average daily rate for all booking at city and resort hotels
hotels %>%
  group_by(hotel) %>%
  summarise(mean_adr = mean(adr))
```

```
## # A tibble: 2 × 2
##   hotel        mean_adr
##   <chr>           <dbl>
## 1 City Hotel      105.
## 2 Resort Hotel     95.0
```

`summarise()` by variable using `group_by()`

```
# Count of bookings at city and resort hotels
hotels %>%
  group_by(hotel) %>%
  summarise(count = n())
```

```
## # A tibble: 2 × 2
##   hotel        count
##   <chr>        <int>
## 1 City Hotel   79330
## 2 Resort Hotel 40060
```

Calculating frequencies

- These two give the same results

```
hotels %>%
  group_by(hotel) %>%
  summarise(count = n())
```

```
## # A tibble: 2 × 2
##   hotel        count
##   <chr>        <int>
## 1 City Hotel   79330
## 2 Resort Hotel 40060
```

```
hotels %>%
  count(hotel)
```

```
## # A tibble: 2 × 2
##   hotel             n
##   <chr>         <int>
## 1 City Hotel    79330
## 2 Resort Hotel 40060
```

`summarise()` for multiple summary statistics

```
hotels %>%
  summarise(
    min_adr = min(adr),
    mean_adr = mean(adr),
    median_adr = median(adr),
    max_adr = max(adr)
  )
```

```
## # A tibble: 1 × 4
##   min_adr mean_adr median_adr max_adr
##     <dbl>    <dbl>      <dbl>   <dbl>
## 1   -6.38     102.       94.6    5400
```

`select()`, `slice()`, and `arrange()`

```
hotels %>%
  select(hotel, lead_time) %>%
  slice(1:5) %>%
  arrange(lead_time)
```

```
## # A tibble: 5 × 2
##   hotel        lead_time
##   <chr>            <dbl>
## 1 Resort Hotel         7
## 2 Resort Hotel        13
## 3 Resort Hotel        14
## 4 Resort Hotel       342
## 5 Resort Hotel       737
```

`select()`, `arrange()`, and `slice()`

- How is that different from?

```
hotels %>%
  select(hotel, lead_time) %>%
  arrange(lead_time) %>%
  slice(1:5)
```

`select()`, `arrange()`, and `slice()`

- How is that different from?

```
hotels %>%
  select(hotel, lead_time) %>%
  arrange(lead_time) %>%
  slice(1:5)
```

```
## # A tibble: 5 × 2
##   hotel        lead_time
##   <chr>            <dbl>
## 1 Resort Hotel         0
## 2 Resort Hotel         0
## 3 Resort Hotel         0
## 4 Resort Hotel         0
## 5 Resort Hotel         0
```

- the **order of the piping** matters!

as ignored code or comments

```
hotels %>%
  # slice the first five rows  # this line is a comment
  #select(hotel) %>%          # this one doesn't run
  slice(1:5)                  # this line runs
```

```
## # A tibble: 5 × 32
##   hotel        is_canceled lead_time arrival_date_year arrival_date_month
##   <chr>              <dbl>     <dbl>             <dbl> <chr>
## 1 Resort Hotel           0       342              2015 July
## 2 Resort Hotel           0       737              2015 July
## 3 Resort Hotel           0         7              2015 July
## 4 Resort Hotel           0        13              2015 July
## 5 Resort Hotel           0        14              2015 July
## # ℹ 27 more variables: arrival_date_week_number <dbl>,
## #   arrival_date_day_of_month <dbl>, stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>, children <dbl>, babies <dbl>,
## #   meal <chr>, country <chr>, market_segment <chr>,
## #   distribution_channel <chr>, is_repeated_guest <dbl>,
## #   previous_cancellations <dbl>, previous_bookings_not_canceled <dbl>,
## #   reserved_room_type <chr>, assigned_room_type <chr>, …
```

`filter()` to select rows based on conditions

- get all rows where `hotel` is "City Hotel" (note the use of `==` as opposed to `=`)

```
# bookings in City Hotels
hotels %>%
  filter(hotel == "City Hotel")
```

```
## # A tibble: 79,330 × 32
##    hotel      is_canceled lead_time arrival_date_year arrival_date_month
##    <chr>            <dbl>     <dbl>             <dbl> <chr>
##  1 City Hotel           0         6              2015 July
##  2 City Hotel           1        88              2015 July
##  3 City Hotel           1        65              2015 July
##  4 City Hotel           1        92              2015 July
##  5 City Hotel           1       100              2015 July
##  6 City Hotel           1        79              2015 July
##  7 City Hotel           0         3              2015 July
##  8 City Hotel           1        63              2015 July
##  9 City Hotel           1        62              2015 July
## 10 City Hotel           1        62              2015 July
## # ℹ 79,320 more rows
## # ℹ 27 more variables: arrival_date_week_number <dbl>,
## #   arrival_date_day_of_month <dbl>, stays_in_weekend_nights <dbl>,
```

`filter()` to select rows based on conditions

- get all rows where adults is 0 and children is greater than or equal to 1

```
hotels %>%
  filter(
    adults == 0,
    children >= 1
    ) %>%
  select(adults, babies, children)
```

```
## # A tibble: 223 × 3
##    adults babies children
##     <dbl>  <dbl>    <dbl>
##  1      0      0        3
##  2      0      0        2
##  3      0      0        2
##  4      0      0        2
##  5      0      0        2
##  6      0      0        3
##  7      0      1        2
##  8      0      0        2
##  9      0      0        2
## 10      0      0        2
## # ℹ 213 more rows
```

`filter()` to select rows based on (complex) conditions

- get all rows where adults is 1 and children is greater than or equal to 1 or babies is greater than or equal to 1

```
hotels %>%
  filter( adults == 1,
    children >= 1 | babies >=1) %>%    # | means OR
  select(adults, babies, children)
```

```
## # A tibble: 450 × 3
##    adults babies children
##     <dbl>  <dbl>    <dbl>
##  1      1      0        2
##  2      1      0        2
##  3      1      0        1
##  4      1      1        0
##  5      1      0        1
##  6      1      0        1
##  7      1      0        2
##  8      1      0        2
##  9      1      0        1
## 10      1      0        1
```

Logical operators in R

Operator	Definition
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
x & y	x AND y
x | y	x OR y
is.na(x)	test if x is NA
!is.na(x)	test if x is not NA
x %in% y	test if x is in y
!(x %in% y)	test if x is not in y
!x	not x

`count()` and `arrange()` instead

```
hotels %>%
  count(market_segment) %>%
  arrange(desc(n)) # <-- decreasing order of counts
```

```
## # A tibble: 8 × 2
##   market_segment     n
##   <chr>          <int>
## 1 Online TA      56477
## 2 Offline TA/TO  24219
## 3 Groups         19811
## 4 Direct         12606
## 5 Corporate       5295
## 6 Complementary    743
## 7 Aviation         237
## 8 Undefined          2
```

`mutate()` to add new variables

```
hotels %>%
  mutate(little_ones = children + babies) %>% # <---
  select(children, babies, little_ones) %>%
  arrange(desc(little_ones))
```

```
## # A tibble: 119,390 × 3
##    children babies little_ones
##       <dbl>  <dbl>       <dbl>
##  1       10      0          10
##  2        0     10          10
##  3        0      9           9
##  4        2      1           3
##  5        2      1           3
##  6        2      1           3
##  7        3      0           3
##  8        2      1           3
##  9        2      1           3
## 10        3      0           3
## # ℹ 119,380 more rows
```

`mutate()` and `filter()` using new columns

```
hotels %>%
  mutate(little_ones = children + babies) %>%
  filter(
    little_ones >= 1,
    hotel == "Resort Hotel"
    ) %>%
  select(hotel, little_ones)
```

```
hotels %>%
  mutate(little_ones = children + babies) %>%
  filter(
    little_ones >= 1,
    hotel == "City Hotel"
    )  %>%
  select(hotel, little_ones)
```

`mutate()` and `filter()` using new columns

```
hotels %>%
  mutate(little_ones = children + babies) %>%
  filter(
    little_ones >= 1,
    hotel == "Resort Hotel"
    ) %>%
  select(hotel, little_ones)
```

```
## # A tibble: 3,929 × 2
##    hotel        little_ones
##    <chr>              <dbl>
##  1 Resort Hotel           1
##  2 Resort Hotel           2
##  3 Resort Hotel           2
##  4 Resort Hotel           2
##  5 Resort Hotel           1
##  6 Resort Hotel           1
##  7 Resort Hotel           2
##  8 Resort Hotel           2
##  9 Resort Hotel           1
## 10 Resort Hotel           1
## # ℹ 3,919 more rows
```

```
hotels %>%
  mutate(little_ones = children + babies) %>%
  filter(
    little_ones >= 1,
    hotel == "City Hotel"
    )  %>%
  select(hotel, little_ones)
```

```
## # A tibble: 5,403 × 2
##    hotel      little_ones
##    <chr>            <dbl>
##  1 City Hotel           1
##  2 City Hotel           1
##  3 City Hotel           2
##  4 City Hotel           1
##  5 City Hotel           1
##  6 City Hotel           1
##  7 City Hotel           1
##  8 City Hotel           1
##  9 City Hotel           1
## 10 City Hotel           1
## # ℹ 5,393 more rows
```

Thanks!

Slides created via the R packages:

xaringan
gadenbuie/xaringanthemer.

Faculty of Arts
& Social Sciences

Week-5: Functions

NM2207: Computational Media Literacy

Narayani Vedam, Ph.D.

Department of Communications and New Media

NUS National University of Singapore | Faculty of Arts & Social Sciences

This week

Table of contents


```
# Example of a function

circle_area <- function(r){

  pi*r^2

}
```

I. Functions and their use

What are functions?

- In mathematics, a function is a rule that maps the input to an output
- In computing, a function is a sequence of instructions within a larger computer program
- We have used so many functions so far, some of them being,
  - `read_csv()`
  - `mean()`
  - `summarize()`
  - `ggplot()`
  - `count()`
- These are functions that are either provided by R or by packages
- But there are times when such an inbuilt function or a package function won't be available

Code repetitions

"You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)" - Hadley Wickham, R for Data Science

Instead of repeating code

```
data %>%
mutate(a_new = (a_old - min(a_old)) / (max(a_old) - min(a_old)), # <-- Notice how the only change
       b_new = (b_old - min(b_old)) / (max(b_old) - min(b_old)), # line is the change in variables
       c_new = (c_old - min(c_old)) / (max(c_old) - min(c_old)), # a_old to b_old
       d_new = (d_old - min(d_old)) / (max(d_old) - min(d_old))) # b_old to c_old, c_old to d_old
```

Write a function!

Code repetitions

Write a function

```
# Generic function
rescale_01 <- function(x) {

  (x - min(x)) / (max(x) - min(x)) #<-- Notice how a,b,c,d are replaced by x

}

data %>%
  mutate(a_new = rescale_01(a_old),

         b_new = rescale_01(b_old),

         c_new = rescale_01(c_old),

         d_new = rescale_01(d_old))
```

Function anatomy

A function consists of,

- Function arguments
  - They are parameters used by instructions in the body of the function
- Function body
  - They contain statements that are executed when the function is called
- Return value
  - The output inside `return()`
  - Could be a vector, list, data frame, another function, or even nothing
  - If unspecified, will be the last thing calculated

```
# Structure of a function
function_name <- function(arguments) {

  body_of_the_function

  return(output)

}
```

Note: We can assign the function to a name like any other object in R.

Function anatomy: example

- arguments: `x`
- body with instructions: `(x - min(x)) / (max(x) - min(x))`
- assign output to a variable: `rescale_01`
- Note that we don't need to explicitly call `return()`
- the last line of the code will be the value returned by the function.

```
rescale_01 <- function(x) {
  (x - min(x)) / (max(x) - min(x))
  }
```

II. Writing functions

Writing a function: printing output

You start writing code to say "Hello" to all of your friends.

```
print("Hello Kashif!")
```

```
## [1] "Hello Kashif!"
```

```
print("Hello Zach!")
```

```
## [1] "Hello Zach!"
```

```
print("Hello Deniz!")
```

```
## [1] "Hello Deniz!"
```

Writing a function: parameterize the code

- **Start** with the body of the function
- **Ask**: What part of the code is changing?
  - Make this an argument
- **Rewrite** the code to accommodate the parameterization
- Check several **potential inputs** to avoid future headaches
- Insert the body of the code within `{}`
- Name the function, `function_name`
- `?function_name` tells you
  - what arguments the function expects
  - what value it produces

```
# print("Hello Kashif!") becomes ...

name <- "Kashif"

print(paste0("Hello ", name, "!"))
```

```
## [1] "Hello Kashif!"
```

Writing a function: add the structure

```
# name <- "Kashiif"

# print(paste0("Hello ", name, "!"))

function(name) {

print(paste0("Hello ", name, "!"))

}
```

```
## function(name) {
##
## print(paste0("Hello ", name, "!"))
##
## }
```

Writing a function: assign to a name

Try to use names that actively tell the user what the code does

- We recommend `verb_thing()`
- good: `calc_size()` or `compare_prices()`
- bad: `prices()`, `calc()`, or `fun1()`

```
# name <- "Kashif"
# print(paste0("Hello ", name, "!"))
say_hello_to <- function(name) {
  print(paste0("Hello ", name, "!"))
}
```

Simple example: printing output

Test out different inputs!

```
say_hello_to('Kashif')
```

```
## [1] "Hello Kashif!"
```

```
say_hello_to('Zach')
```

```
## [1] "Hello Zach!"
```

```
say_hello_to('Deniz')
```

```
## [1] "Hello Deniz!"
```

Technical aside: typeof(your_function)

Like all R objects, functions have types;

- Primitive functions are of type “built-in”

```
typeof(`+`)
```

```
## [1] "builtin"
```

```
typeof(sum)
```

```
## [1] "builtin"
```

Technical aside: typeof(your_function)

Like all R objects, functions have types;

- The following functions are of type "closure"
    - user-defined functions
    - functions loaded with packages
    - many base R functions

```
typeof(say_hello_to)
```

```
## [1] "closure"
```

```
typeof(mean)
```

```
## [1] "closure"
```

Second example: mean of a sample

- For the sake of simplicity, assume that a sample represents a small collection - sample of height of a population, sample of grades of students, etc.
- This collection, numerically, could either be **random** or **normal** (most common kinds!)

```
mean(rnorm(100))
```

```
## [1] 0.1392151
```

```
mean(rnorm(3000))
```

```
## [1] 0.00477852
```

Second example: calculating the mean of a sample

The number is changing, so it becomes the argument.

- The number is the sample size, so I call it `sample_size`.
- `n` would also be appropriate.
- The body code is otherwise identical to the code in the previous slide.

```
calc_sample_mean <- function(sample_size) {
  mean(rnorm(sample_size))
  }
```

Second example: calculating the mean of a sample

For added clarity you can unnest your code, and assign the intermediate results to meaningful names.

- `return()` explicitly tells R what the function will return
  - The last line of code run is returned by default.

```
calc_sample_mean <- function(sample_size) {
  random_sample <- rnorm(sample_size)
  sample_mean <- mean(random_sample)
  return(sample_mean)
}
```

Second example: calculating the mean of a sample

If the function can be fit in one line, then you can write it without the curly brackets like so:

```
calc_sample_mean <- function(n) mean(rnorm(n))
```

Some settings call for ***anonymous*** functions, where the function has no name.

```
function(n) mean(rnorm(n))
```

```
## function(n) mean(rnorm(n))
```

Always test your code

Try to foresee the kind of input you expect to use.

```
calc_sample_mean(1000)
```

```
## [1] -0.06605835
```

We see below that this function is not vectorized. We might hope to get 3 sample means out but only get 1.

```
# read ?rnorm to understand how rnorm
# inteprets vector input.
calc_sample_mean(c(100, 300, 3000))
```

```
## [1] 0.5082238
```

How to deal with unvectorized functions

If we don't want to change our function, but we want to use it to deal with vectors, then we have a couple options:

- Here we are going to use the function `group_by` from `tidyverse` package

```
library(tidyverse)

#creating a vector to test our function
sample_tibble <- tibble(sample_sizes =
                          c(100, 300, 3000))

#using rowwise groups the data by row,
# allowing calc_sample_mean
sample_tibble %>%
group_by(sample_sizes) %>%
mutate(sample_means =
      calc_sample_mean(sample_sizes))
```

```
## # A tibble: 3 × 2
## # Groups:   sample_sizes [3]
##   sample_sizes sample_means
##          <dbl>        <dbl>
## 1          100      0.0502
## 2          300     -0.00606
## 3         3000      0.00573
```

Adding additional arguments

If we want to be able to adjust the details of how our function runs we can add arguments

- typically, we put “data” arguments first
  - and then “detail” arguments after
  - Mind the order of arguments/parameters, because it can’t change when you call the function in the future

```
calc_sample_mean <- function(sample_size,
                             our_mean, our_sd)
{

sample <- rnorm(sample_size,
                mean = our_mean,
                mean(sample),
                sd = our_sd)

}
```

Setting defaults

We usually set default values for “detail” arguments.

```
calc_sample_mean <- function(sample_size,
                             our_mean=0,
                             our_sd=1) {

  sample <- rnorm(sample_size,
                  mean = our_mean,
                  sd = our_sd)

   mean(sample)
}
```

```
# uses the defaults
calc_sample_mean(sample_size = 10)
```

```
## [1] -0.2621734
```

Setting defaults

```
# we can change one or two defaults.
# You can refer by name, or use position
calc_sample_mean(10, our_sd = 2)
```

```
## [1] 0.3151067
```

```
calc_sample_mean(10, our_mean = 6)
```

```
## [1] 5.978619
```

```
calc_sample_mean(10, 6, 2)
```

```
## [1] 5.903527
```

Setting defaults

This won't work though:

```
calc_sample_mean(our_mean = 5)
```

```
## Error in rnorm(sample_size, mean = our_mean, sd = our_sd): argument "sample_size" is missing, wi
```

Some more examples

```r
# Add 2 to the input x
add_two <- function(x) {
  x+2
}
```

```r
add_two(4)
```

```
## [1] 6
```

```r
add_two(-34)
```

```
## [1] -32
```

```r
add_two(5.784)
```

III. Common mistakes to avoid

What could go wrong?

Mismatch in the argument in the definition of the function (`x`) and the variable name used inside the function (`y`)

```
# Erroneous code
add_two <- function(x) {

  y+2

}

# Function call
add_two(4)
```

```
## Error in add_two(4): object 'y' not found
```

```
# Correct code
add_two <- function(x) {

  x+2

}

# Function call

add_two(4)
```

```
## [1] 6
```

Another example: adding two numbers

Mismatch in the names of the function in the definition and in the function call

```
# function definition
add_numbers <- function(x,y) {
  x+y
}

# function call
add_numers(45, 12)
```

```
## Error in add_numers(45, 12): could not find function "add_numers"
```

Another example: adding two numbers

Mismatch in the number of arguments in the function definition and in the function call

```
# function definition
add_numbers <- function(x,y) {
  x+y
}

# function call
add_numbers(45, 12, 72)
```

```
## Error in add_numbers(45, 12, 72): unused argument (72)
```

Another example: adding two numbers

You cannot return more than one value using the `return()` command

```
# function definition
add_numbers <- function(x,y) {
  z = 20
  x+y
  return(x+y,z)
}
# Function call
add_numbers(1,2)
```

```
## Error in return(x + y, z): multi-argument returns are not permitted
```

Another example: adding two numbers

Variables declared inside a function, cannot be accessed outside of it

```
# function definition
add_numbers <- function(x,y) {
  z = 20
  x+y
}
print(z)
```

```
## Error in print(z): object 'z' not found
```

IV. Scope of variables

Scoping

```r
# Initialize z
z <- 1
sprintf("The value assigned to z outside the function is %d",z)
```

```
## [1] "The value assigned to z outside the function is 1"
```

```r
# declare a function, notice how we pass a value of 2 for z
foo <- function(z = 2) {
  # reassigning z
  z <- 3
  return(z+3)
}
foo()
```

```
## [1] 6
```

Scoping

```r
# Initialize z
z <- 1
# declare a function, notice how we pass a value of 2 for z
foo <- function(z = 2) {
  # reassigning z
  z <- 3
  return(z+3)
}
# another reassignment of z
foo(z = 4)
```

```
## [1] 6
```

```r
# Accessing z outside the function
sprintf("The final value of z after reassigning it to a different value inside the function is %d"
```

```
## [1] "The final value of z after reassigning it to a different value inside the function is 1"
```

Scoping

> The scope of a variable is decided by two factors, (a) location of initialization and (b) where we can access it when required

There are mainly two types of variable scopes,

Global

- They are declared outside functions
- They can be accessed from anywhere in the program
- Hence, **global**
- In the code, `z<-1` is the global variable
- Type `z` in the console and check the output

Local

- They are declared inside functions
- They cannot be accessed outside the functions
- Hence, **local**
- In the code, `z<-3` is the local variable

Thanks!

Slides created via the R packages:

xaringan
gadenbuie/xaringanthemer.

Faculty of Arts
& Social Sciences