# Research Toolkit

Nikita Tkachenko

2023-03-05T00:00:00-08:00

# Table of contents

# Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

```
1 + 1
```

```
[1] 2
```

# 1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

# 2 Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# Part I

# Writing and Lit Review

# Part II

# Collecting the Data

# 3 Qualtrics API

Instead of downloading your data through Qualtrics every time we will load data using Qualtrics API. For a complete guide refer to this vignette, which this guide is based on.

University of San Francisco provides access to Qualtrics; however you need to separately request API access. Good news, I have talked to ITS and you should already have it! If not let me know!

First, we need to install a package `qualtRics`.

```
library("tidyverse")
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.2 --
v ggplot2 3.4.1     v purrr   1.0.1
v tibble  3.1.8     v dplyr   1.1.0
v tidyr   1.3.0     v stringr 1.5.0
v readr   2.1.3     v forcats 0.5.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

```
library("qualtRics")
library("scales")
```

```
Attaching package: 'scales'

The following object is masked from 'package:purrr':

    discard

The following object is masked from 'package:readr':

    col_factor
```

```
knitr::opts_chunk$set(echo = T, warning = F, message = F, error = F)
```

## 3.1 Core Functions

Currently, the package contains three core functions:

- `all_surveys()` shows surveys you can access.
- `fetch_survey()` downloads the survey.
- `read_survey()` reads CSV files you downloaded manually from Qualtrics.

It also contains a number of helper functions, including:

- `qualtrics_api_credentials()` stores your API key and base url in environment variables.
- `survey_questions()` retrieves a data frame containing questions and question IDs for a survey;
- `extract_colmap()` retrieves a similar data frame with more detailed mapping from columns to labels.
- `metadata()` retrieves metadata about your survey, such as questions, survey flow, number of responses etc.

Note that you can only export surveys that you own, or to which you have been given administration rights.

## 3.2 Connecting to the API

If you have received API access, now you can connect to the API. To get `api_key` and `base_url` go to Qualtrics Home Page > Account Settings > Qualtrics IDs or click this link. Under "API" click "Generate Topic" and you will be issued a Token. Copy this string a put in "YOUR_API_KEY". Then look at "User" module, copy "Datacenter ID" and ".qualtrics.com" after it. Your Base Url should look something like this: "lad2.qualtrics.com".

```
qualtrics_api_credentials(api_key = "YOUR_API_KEY",
                          base_url = "YOUR_BASE_URL",
                          install = TRUE,
                        # overwrite = TRUE # If you need to update your credentials
                          )
```

```
qualtrics_api_credentials(api_key = "CEOj8Iwh6BUTNZ9J7PXDJLZpStQaXkmlwLPYzCgu",
                          base_url = "lad2.qualtrics.com",
                          install = TRUE)
```

After `qualtrics_api_credintials` stored your credentials, you can use `all_surveys()` to fetch information on your surveys.

```
(surveys <- all_surveys())
```

```
# A tibble: 8 x 6
  id                 name                          ownerId lastM~1 creat~2 isAct~3
  <chr>              <chr>                         <chr>   <chr>   <chr>   <lgl>
1 SV_0rearXjH2Ri6umq Student Satisfaction          UR_2tz~ 2023-0~ 2023-0~ FALSE
2 SV_2gWVWLn2vCCDJGu Luvuyo                        UR_2tz~ 2022-0~ 2022-0~ FALSE
3 SV_3h0HPHQbJStqb8a Pilot (USA) - Giver Motive~ UR_2tz~ 2022-0~ 2022-0~ FALSE
4 SV_3Q011O6gNqwn1Nc Pen Experiment               UR_2tz~ 2022-1~ 2022-1~ FALSE
5 SV_4YDoTFLN61nKecS Pen_Showcase                  UR_2tz~ 2023-0~ 2023-0~ TRUE
6 SV_4Yhvzriag8syz7U Prosociality Experiment       UR_2tz~ 2023-0~ 2022-0~ FALSE
7 SV_bJIs8lwz4CfAAgS test                          UR_2tz~ 2023-0~ 2023-0~ TRUE
8 SV_dnBQ3YB4rLC03J4 test2                         UR_2tz~ 2023-0~ 2023-0~ FALSE
# ... with abbreviated variable names 1: lastModified, 2: creationDate,
#   3: isActive
```

Once you select the questionnaire you want you can refer to it using `id`. If you want redownload the data set `force_request = TRUE`, otherwise it will load prior saved download.

```
survey_data <- fetch_survey(surveyID = "SV_bJIs8lwz4CfAAgS",
             verbose = TRUE,
             force_request = TRUE)
```

```
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
survey_data %>% glimpse()
```

```
Rows: 22
Columns: 89
$ StartDate                 <dttm> 2023-02-07 16:00:52, 2023-02-07 16:01:17, 2~
$ EndDate                   <dttm> 2023-02-07 16:04:08, 2023-02-07 16:04:08, 2~
$ Status                    <chr> "IP Address", "IP Address", "IP Address", "I~
$ IPAddress                 <chr> "138.202.129.171", "138.202.129.164", "138.2~
$ Progress                  <dbl> 100, 100, 100, 100, 100, 100, 100, 100, 100,~
$ `Duration (in seconds)`   <dbl> 196, 171, 104, 187, 189, 198, 210, 209, 180,~
$ Finished                  <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TR~
$ RecordedDate              <dttm> 2023-02-07 16:04:09, 2023-02-07 16:04:09, 2~
$ ResponseId                <chr> "R_3r385toH6wjBmyr", "R_2usVBjQ7yXrmY72", "R~
$ RecipientLastName         <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ RecipientFirstName        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ RecipientEmail            <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ ExternalReference         <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ LocationLatitude          <dbl> 37.78, 37.78, 37.78, 37.78, 37.78, 37.78, 37~
$ LocationLongitude         <dbl> -122.465, -122.465, -122.465, -122.465, -122~
$ DistributionChannel       <chr> "anonymous", "anonymous", "anonymous", "anon~
$ UserLanguage              <chr> "EN", "EN", "EN", "EN", "EN", "EN", "EN", "E~
$ Consent                   <ord> Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes,~
$ Gender                    <ord> Female, Female, Male, Male, Male, Male, Male~
$ Name                      <chr> "Tasha", "Khushboo Patel", "Nikita", "Lawren~
$ Competitive               <ord> Competitive, Competitive, Competitive, Compe~
$ Pizzas_1                  <chr> "Margherita", NA, NA, "Margherita", "Margher~
$ Pizzas_2                  <chr> "Pepperoni", NA, NA, "Pepperoni", "Pepperoni~
$ Pizzas_3                  <chr> NA, "BBQ Chicken", NA, NA, "BBQ Chicken", "B~
$ Pizzas_4                  <chr> "Hawaiian", NA, NA, NA, "Hawaiian", NA, "Haw~
$ Pizzas_5                  <chr> "Veggie", "Veggie", "Veggie", "Veggie", "Veg~
$ Pizzas_6                  <chr> NA, NA, NA, NA, NA, "I also love that one:",~
$ Pizzas_7                  <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ Pizzas_6_TEXT             <chr> NA, NA, NA, NA, NA, "Garlic White Chinese be~
$ Pizzas_DO_1               <dbl> 5, 1, 2, 4, 5, 3, 1, 1, 5, 5, 3, 1, 5, 1, 4,~
$ Pizzas_DO_2               <dbl> 3, 3, 5, 1, 3, 2, 3, 3, 1, 4, 2, 5, 3, 3, 5,~
$ Pizzas_DO_3               <dbl> 2, 2, 3, 5, 4, 1, 4, 5, 2, 2, 1, 3, 1, 2, 2,~
$ Pizzas_DO_4               <dbl> 1, 4, 1, 3, 2, 5, 2, 2, 3, 1, 5, 4, 2, 5, 1,~
$ Pizzas_DO_5               <dbl> 4, 5, 4, 2, 1, 4, 5, 4, 4, 3, 4, 2, 4, 4, 3,~
$ Pizzas_DO_6               <dbl> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,~
$ Pizzas_DO_7               <dbl> 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,~
$ `1_Taste`                 <dbl> 5, NA, NA, 5, 3, NA, 3, 4, 4, 5, 4, NA, NA, ~
$ `1_Healthiness`           <dbl> 5, NA, NA, 4, 4, NA, 2, 3, 3, 5, 2, NA, NA, ~
$ `1_Ease_Of_Preparation`   <dbl> 5, NA, NA, 4, 4, NA, 4, 3, 4, 3, 3, NA, NA, ~
$ `2_Taste`                 <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ `2_Healthiness`           <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
```

```
$ `2_Ease_Of_Preparation`   <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ `3_Taste`                 <dbl> NA, NA, NA, NA, NA, 4, 5, NA, 4, NA, NA, NA,~
$ `3_Healthiness`           <dbl> NA, NA, NA, NA, NA, 3, 3, NA, 2, NA, NA, NA,~
$ `3_Ease_Of_Preparation`   <dbl> NA, NA, NA, NA, NA, 3, 4, NA, 3, NA, NA, NA,~
$ `9_Taste`                 <dbl> 5, NA, NA, 5, 5, NA, 5, NA, NA, NA, 5, NA, 4~
$ `9_Healthiness`           <dbl> 3, NA, NA, 3, 2, NA, 2, NA, NA, NA, 2, NA, 4~
$ `9_Ease_Of_Preparation`   <dbl> 5, NA, NA, 5, 4, NA, 4, NA, NA, NA, 4, NA, N~
$ `10_Taste`                <dbl> NA, 5, NA, NA, 5, 5, 3, 4, 4, NA, 4, NA, 4, ~
$ `10_Healthiness`          <dbl> NA, NA, NA, NA, 2, 3, 2, 3, 3, NA, 2, NA, 4,~
$ `10_Ease_Of_Preparation`  <dbl> NA, NA, NA, NA, 3, 3, 2, 3, 2, NA, 3, NA, NA~
$ `11_Taste`                <dbl> 5, NA, NA, NA, 4, NA, 5, 4, NA, NA, 4, 5, NA~
$ `11_Healthiness`          <dbl> 4, NA, NA, NA, 2, NA, 2, 3, NA, NA, 3, 4, NA~
$ `11_Ease_Of_Preparation`  <dbl> 5, NA, NA, NA, 3, NA, 3, 2, NA, NA, 2, 3, NA~
$ `12_Taste`                <dbl> 5, 4, 4, 4, 4, NA, 3, NA, NA, NA, 4, NA, NA,~
$ `12_Healthiness`          <dbl> 5, 2, 5, 5, 4, NA, 3, NA, NA, NA, 4, NA, NA,~
$ `12_Ease_Of_Preparation`  <dbl> 4, 3, 3, 3, 2, NA, 2, NA, NA, NA, 3, NA, NA,~
$ `match timer_First Click` <dbl> 0.000, 0.000, 0.000, 0.000, 20.317, 0.000, 0~
$ `match timer_Last Click`  <dbl> 0.000, 0.000, 0.000, 0.000, 20.317, 0.000, 0~
$ `match timer_Page Submit` <dbl> 8.296, 12.572, 7.955, 22.206, 21.188, 7.224,~
$ `match timer_Click Count` <dbl> 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,~
$ transfer                  <dbl> 50, NA, 35, NA, 49, NA, NA, 35, NA, 10, 50, ~
$ `get timer_First Click`   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ `get timer_Last Click`    <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ `get timer_Page Submit`   <dbl> 4.999, 15.967, 32.000, 2.787, 3.006, 10.538,~
$ `get timer_Click Count`   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ decision...67             <chr> NA, "I accept A's offer.\n(You get ${e://Fie~
$ Q23                       <ord> Yes, Absolutely, Absolutely, Yes, Yes, Yes, ~
$ researcherID              <chr> "hTtZNw8TA0", "hTtZNw8TA0", "hTtZNw8TA0", "h~
$ studyID                   <chr> "test", "test", "test", "test", "test", "tes~
$ groupID                   <dbl> 6, 6, 5, 5, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11~
$ participantID             <chr> "R_3r385toH6wjBmyr", "R_2usVBjQ7yXrmY72", "R~
$ groupSize                 <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,~
$ numStages                 <dbl> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,~
$ roles                     <chr> "A,B", "A,B", "A,B", "A,B", "A,B", "A,B", "A~
$ participantRole           <chr> "A", "B", "A", "B", "A", "B", "B", "A", "B",~
$ timeOutLog                <chr> "OK -- no issues", "OK -- no issues", "OK --~
$ botMatch                  <chr> "no", "no", "no", "no", "no", "no", "no", "n~
$ total                     <dbl> 100, 100, 100, 100, 100, 100, 100, 100, 100,~
$ offer                     <dbl> 50, 50, 35, 35, 49, 49, 35, 35, 10, 10, 50, ~
$ decision...81             <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
$ payoff                    <dbl> 50, 50, 65, 35, 51, 49, 35, 65, 10, 90, 50, ~
$ sendStage                 <dbl> 1, 2, 1, 2, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2,~
$ sendData                  <chr> "offer", "decision", "offer", "decision", "o~
```

```
$ getStage                  <dbl> 2, 1, 2, 1, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1, 1,~
$ getData                   <chr> "B", "A", "B", "A", "B", "A", "A", "B", "A",~
$ defaultData               <dbl> 2, 20, 2, 5, 2, 71, 61, 1, 83, 2, 2, 81, 1, ~
$ saveData                  <chr> "decision", "offer", "decision", "offer", "d~
$ randomPercent             <dbl> NA, 20, NA, 5, NA, 71, 61, NA, 83, NA, NA, 8~
```

In case you want to see text of the questions use `survey_questions()`.

```
survey_questions <- survey_questions(surveyID = "SV_bJIs8lwz4CfAAgS")
head(survey_questions, n = 5)
```

```
# A tibble: 5 x 4
  qid   qname        question                                                 force~1
  <chr> <chr>        <chr>                                                    <lgl>
1 QID25 Introduction "Welcome to the <strong>University of San Francisc~ FALSE
2 QID26 Consent      "Do you agree to participate in the survey?"         TRUE
3 QID21 Gender       "What is your gender"                                TRUE
4 QID23 Name         "What is your Name?"                                 FALSE
5 QID22 Competitive  "Would you consider yourself competitive?"           FALSE
# ... with abbreviated variable name 1: force_resp
```

## 3.3 Example

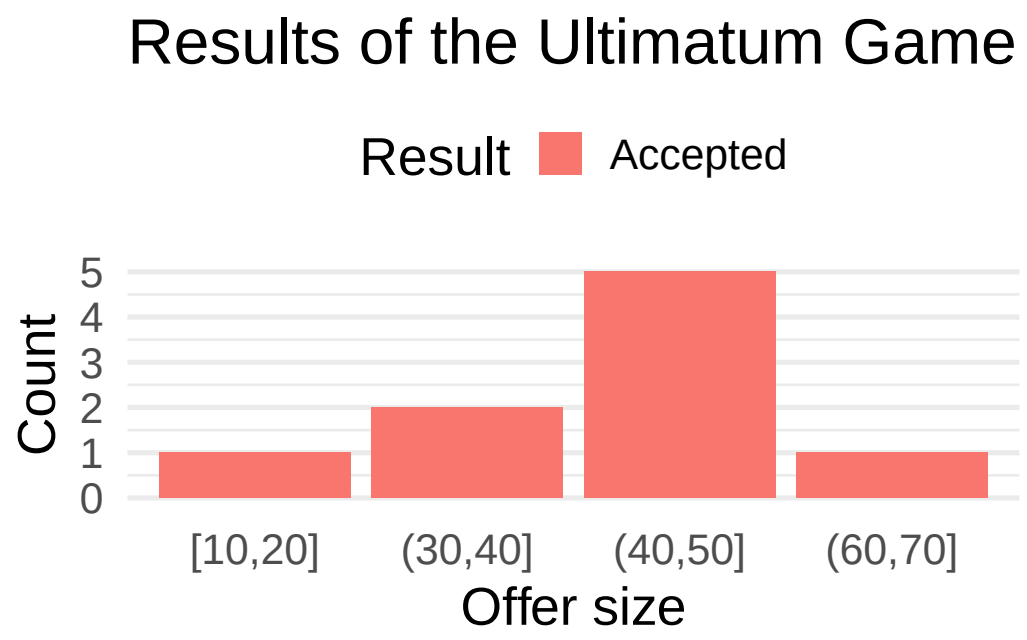
```
survey_data <- fetch_survey(surveyID = "SV_bJIs8lwz4CfAAgS",
                            verbose = TRUE,force_request = TRUE)
```

```
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
survey_data <- survey_data %>% janitor::clean_names()

graph_data <- survey_data %>%select(gender, offer,  decision_81, participant_id, participa
  mutate(participant_role = recode(participant_role,"A" = "dictator", "B" = "recipient" ),
         decision = recode(decision_81, "1" = "Accepted", "2" = "Declined")) %>%
```

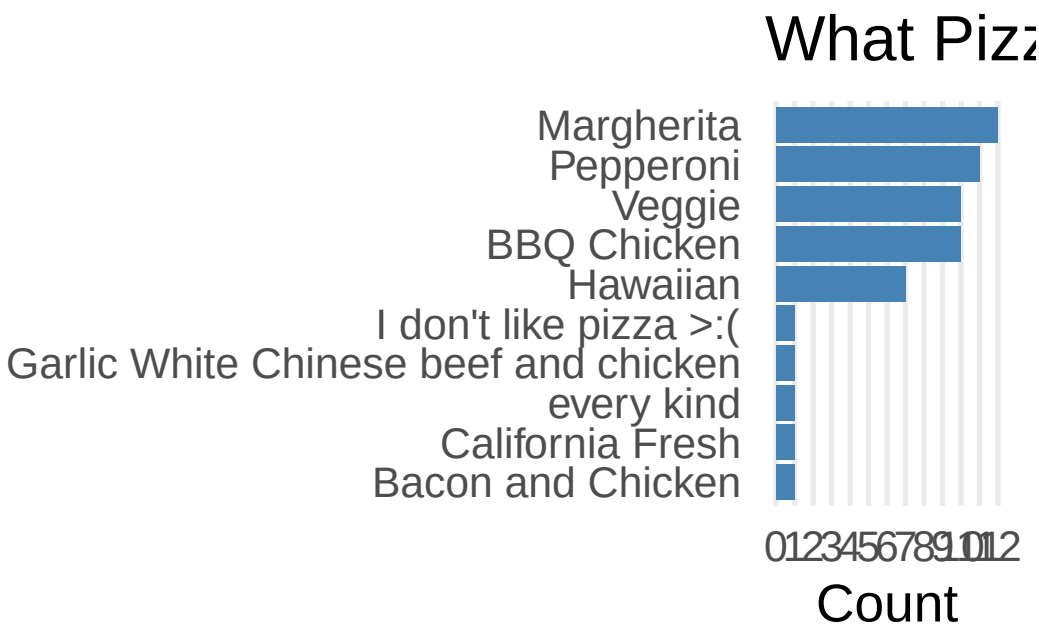
```
  mutate(interval = cut_width(offer, width = 10, center = 45)) %>%
  filter(participant_role == "recipient") %>% count(decision, interval)
```

```
graph_data %>%
  ggplot(aes(x = interval, y = n, fill = as.factor(decision))) +
  geom_col(position = position_stack()) +
  theme_minimal(base_size = 20) +
  scale_y_continuous(breaks = scales::breaks_extended(n = max(graph_data$n))) +
  theme(panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank(),
        legend.position = "top") +
  labs(x = "Offer size", y = "Count", title = "Results of the Ultimatum Game", fill = "Res
```




```
pizza_table <- survey_data[c(22:29,72)] %>% select(-pizzas_6) %>% pivot_longer(-participan

 pizza_table %>% ggplot(aes(x = fct_reorder(value, n), y = n)) +
   geom_col(fill = "steelblue") +
   theme_minimal(base_size = 20) +
   scale_y_continuous(breaks = scales::breaks_extended(n = max(pizza_table$n))) +
   theme(panel.grid.major.y = element_blank(),
         panel.grid.minor.y = element_blank(),
         panel.grid.minor.x = element_blank(),
```

```
      legend.position = "top") +
labs(x = NULL, y = "Count", title = "What Pizzas do you like?") +
coord_flip()
```



# Part III

# Working with Data

# 4 Data Manipulation

Data visualization and manipulation are essential tools for understanding and communicating complex information in data analysis. R, a powerful programming language for data analysis, offers a variety of packages for creating visually appealing plots and manipulating data. One of the most popular and user-friendly collections of packages for data visualization and manipulation in R is Tidyverse created by Hadley Wickham, the Chief Scientist at RStudio. The Tidyverse is a collection of packages that covers all common tasks, it can be installed using `install.packages("tidyverse")` and activated with `library("tidyverse")`. In this introduction, we will explore the basics of using Tidyverse, we will use `readr` to read data, `tidyr` to tidy, `dplyr` to manipulate, and `ggplot2` to visualize. To learn more about the Tidyverse check out their website https://www.tidyverse.org/.

Let us start with some basic concepts! We can use R as a basic calculator

```
# This is a comment use "#" to comment something!
2+2
```

```
[1] 4
```

```
2*4
```

```
[1] 8
```

```
2^8
```

```
[1] 256
```

```
(1+3)/(3+5)
```

```
[1] 0.5
```

```
log(10) # This takes a natural log of 10!
```

```
[1] 2.302585
```

We can define variables and perform operations on them. R uses `=` or `<-` to assign values to a variable name. It is stylistically preferred to use `<-` to avoid confusion and some errors.

```
x <- 2 # same as x = 2
x * 4
```

```
[1] 8
```

`x <- 2` stored `2` in `x`. Later when we wrote `x * 4` R substituted `x` for `2` evaluating `2 * 4` to get `8`. We can update value of `x` as much as we want using `=` or `<-`. Keep in mind R is case sensitive so `X` and `x` are different.

```
x
```

```
[1] 2
```

```
x <- x * 5
```

### 4.0.1 Data Types

R has a number of different data types and classes such as data.frames, which are similar to excel spreadsheets with columns and rows. We will first look at vectors. Vectors can hold multiple values of the same types. Most basic ones are numeric, character and logical.

```
x
```

```
[1] 10
```

```
class(x)
```

```
[1] "numeric"
```

```r
(name <- "Parsa Rahimi") # wrapping with (...) will print the variable
```

```
[1] "Parsa Rahimi"
```

```r
class(name)
```

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

```r
(true_or_false <- TRUE)
```

```
[1] TRUE
```

```r
class(true_or_false)
```

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

Note that `name` is stored as a single character string. What if we want store name and surname separately in the same object? We can use concatenate `c()` to combine objects of similar class into a vector!

```r
(name_surname <- c("Parsa","Rahimi"))
```

```
[1] "Parsa"  "Rahimi"
```

```r
length(name)
```

```
[1] 1
```

```r
length(name_surname)
```

```
[1] 2
```

Notice how length of the `name` is 1 and length of the `name_surname` is 2! Let's make a numeric vector and do some operations on it!

```
(i <- c(1, 2, 3, 4))
```

```
[1] 1 2 3 4
```

```
i + 10 # add 10 to each elements
```

```
[1] 11 12 13 14
```

```
i * 10 # multiply each element by 10
```

```
[1] 10 20 30 40
```

```
i + c(2, 4, 6, 8) # add elements together in matching positions
```

```
[1]  3  6  9 12
```

We haven't modified `i` with any of those operations. The results are just printed and not stored. If we want to preserve the results we have to store them in a variable.

```
name
```

```
[1] "Parsa Rahimi"
```

```
name <- i + c(5,4,2,1)
name
```

```
[1] 6 6 5 5
```

Notice that `name` is no longer "Parsa Rahimi". It has been overwritten by assigning a numeric vector instead of a character string. But be careful we can perform numeric operations only on numeric objects otherwise we will receive an error. You can use `str()` to get structure of the object such as type, length and other.

```
name_surname + 2
```

```
Error in name_surname + 2: non-numeric argument to binary operator
```

```
str(name_surname)
```

```
 chr [1:2] "Parsa" "Rahimi"
```

## 4.1 Downloading Data

If you are familiar with Base R (functions that come with the R on installation) you know about read.csv(). `readr` provides a number of function that solve common issues of base R read function. `read_csv` loads data 10 time faster and produces a tibble instead of a data frame while avoiding inconsistencies of the `read.csv`. 'Wait what is tibble?' you might ask. Tibbles is special type of data frame. There are superior to regular data frames as they load faster, maintain input types, permit columns as lists, allow non-standard variable names, and never create row names. Okay you have your data lets load it! First, you need to know the path to your data. You can go find you file and check its location and then copy paste it. If you are windows user your path might have "\", which is an escape character. To fix that replace "\" with "/". By copying the path you are getting absolute path "/Users/User/Documents/your_project/data/file.csv" alternatively you can use local a local path from the folder of the project "/data/file.csv". Let's read the data! `readr::` specifies which packages to use. Replace the text between "..." to your path.

### 4.1.1 Example Data

I will be using a sample of experiment's results from Climate and Cooperation Experiment from Mexico. During the experiments, subjects were asked to complete three series of ravens matrices, four dictator games, and a single lottery game. Sessions below 30 Celsius are labeled as control, and sessions above 30 Celsius as treatment. We will be primarily using results from Raven's matrices games: 3 sets of 12 matrices. First set, `pr_`, is piece-rate round where participants received points for each correctly solved matrix. Second set, `tr_`, is tournament round where participants competed againts a random opponent and the winner received double point and the loser received nothing. Third set, `ch_`, is choice round. Participants were asked to decide whether they want to play piece-rate or tournament againts a different opponent's score from tournament round.

You can find data in the `data` in GitHub repository. We will need `tidyverse`.

```r
library(tidyverse)
```

```r
data <- readr::read_csv("https://raw.githubusercontent.com/nikitoshina/ECON-623-Lab-2023/m
```

We can use `glimpse()` to get a glimpse at the data. It will give us a sample and type of the column. Another very common way is to use `head()` to get a slice of the top rows or `tail()` to get a slice of bottom rows. You can also view the entire data set with `View()`

```r
data %>% glimpse()
```

```
Rows: 114
Columns: 70
$ id                  <chr> "001018001", "001018002", "001018005", "001018009",~
$ country_city        <chr> "mexico_chapingo", "mexico_chapingo", "mexico_chapi~
$ start_time          <chr> "12H 16M 0S", "12H 16M 0S", "12H 16M 0S", "12H 16M ~
$ end_time            <chr> "13H 14M 0S", "13H 14M 0S", "13H 14M 0S", "13H 14M ~
$ date                <date> 2022-06-20, 2022-06-20, 2022-06-20, 2022-06-20, 20~
$ mean_temp_celsius   <dbl> 28.58772, 28.58772, 28.58772, 28.58772, 28.58772, 2~
$ incentive_local     <dbl> 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 1~
$ exchange_usd_local <dbl> 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,~
$ incentive_usd       <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ~
$ gender              <chr> "Female", "Male", "Male", "Male", "Male", "Female",~
$ point_value         <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,~
$ site_id             <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ session_n           <dbl> 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19,~
$ subject_n           <dbl> 1, 2, 5, 9, 10, 13, 14, 15, 1, 2, 3, 4, 5, 6, 7, 8,~
$ tr_opponet_n        <dbl> 2, 1, 9, 5, 13, 10, 15, 14, 2, 1, 4, 3, 5, 4, 8, 7,~
$ ch_opponent_n       <dbl> 9, NA, NA, 1, 9, NA, NA, NA, NA, 3, 2, NA, NA, 8, N~
$ version             <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "~
$ treatment           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ dc_cl_ps            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ dc_cl_envy          <dbl> 2, 2, 2, 2, 1, 1, 2, 1, 2, 2, 2, 1, 1, 1, 1, 2, 2, ~
$ dc_c_ps             <dbl> 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 2, 2, 2, ~
$ dc_c_envy           <dbl> 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, ~
$ dc_cl_ps_points     <dbl> 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,~
$ dc_cl_envy_points   <dbl> 20, 20, 20, 20, 16, 16, 16, 20, 20, 20, 16, 20, 16,~
$ dc_c_ps_points      <dbl> 16, 16, 16, 16, 12, 20, 16, 16, 12, 20, 12, 20, 16,~
$ dc_c_envy_points    <dbl> 23, 23, 23, 23, 21, 18, 23, 23, 23, 23, 18, 21, 23,~
$ pr_correct          <dbl> 7, 5, 6, 1, 7, 2, 6, 7, 5, 6, 8, 2, 2, 5, 6, 5, 2, ~
$ pr_wrong            <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 1, 1, 0, 1, 0, ~
```

```
$ pr_not_attempted   <dbl> 5, 6, 6, 11, 5, 10, 6, 5, 7, 4, 4, 9, 9, 6, 6, 6, 1~
$ tr_correct         <dbl> 3, 6, 7, 5, 9, 7, 6, 7, 4, 7, 6, 3, 6, 7, 8, 6, 6, ~
$ tr_wrong           <dbl> 1, 1, 1, 4, 0, 0, 2, 1, 2, 2, 2, 0, 3, 1, 1, 4, 1, ~
$ tr_not_attempted   <dbl> 8, 5, 4, 3, 3, 5, 4, 4, 6, 3, 4, 9, 3, 4, 3, 2, 5, ~
$ tr_die             <dbl> 2, 5, 2, 6, 6, 2, 2, 6, 2, 4, 5, 3, 6, 6, 3, 6, 1, ~
$ tr_total           <dbl> 5, 11, 9, 11, 15, 9, 8, 13, 6, 11, 11, 6, 12, 13, 1~
$ tr_guess_correct   <dbl> 5, 7, 8, 6, 11, 5, 8, 5, 5, 6, 8, 5, 5, 8, 8, 7, 3,~
$ tr_guess_die       <dbl> 3, 4, 1, 4, 2, 2, 5, 5, 4, 3, 4, 2, 4, 2, 3, 1, 2, ~
$ tr_guess_total     <dbl> 8, 11, 9, 10, 13, 7, 13, 10, 9, 9, 12, 7, 9, 10, 11~
$ tr_other_correct   <dbl> 6, 3, 5, 7, 7, 9, 7, 6, 7, 4, 3, 6, 6, 3, 6, 8, 8, ~
$ tr_other_die       <dbl> 5, 2, 6, 2, 2, 6, 6, 2, 4, 2, 3, 5, 6, 3, 6, 3, 3, ~
$ tr_other_total     <dbl> 11, 5, 11, 9, 9, 15, 13, 8, 11, 6, 6, 11, 12, 6, 12~
$ tr_result          <dbl> 0, 2, 0, 2, 2, 0, 0, 2, 0, 2, 2, 0, 1, 2, 0, 2, 0, ~
$ tr_points          <dbl> 0, 22, 0, 22, 30, 0, 0, 26, 0, 22, 22, 0, 12, 26, 0~
$ f_happy            <dbl> 5, 9, 8, 4, 8, 8, 3, 7, 6, 10, 8, 9, 4, 5, 9, 8, 6,~
$ f_energetic        <dbl> 7, 8, 7, 3, 10, 9, 3, 5, 7, 10, 9, 8, 5, 6, 9, 6, 8~
$ f_frustrated       <dbl> 8, 3, 3, 0, 4, 8, 8, 2, 7, 0, 3, 0, 5, 0, 1, 2, 0, ~
$ f_last_meal_min    <dbl> 180, 240, 240, 90, 30, 120, 90, 120, 30, 120, 30, 3~
$ ch_correct         <dbl> 9, 4, 7, 7, 10, 5, 7, 6, 4, 4, 8, 6, 5, 6, 6, 6, 5,~
$ ch_wrong           <dbl> 0, 0, 0, 5, 0, 2, 1, 2, 4, 2, 3, 0, 1, 2, 2, 2, 0, ~
$ ch_not_attempted   <dbl> 3, 8, 5, 0, 2, 5, 4, 4, 4, 6, 1, 6, 6, 4, 4, 4, 7, ~
$ ch_die             <dbl> 3, 3, 5, 6, 2, 4, 1, 3, 6, 1, 5, 4, 3, 2, 4, 2, 6, ~
$ ch_total           <dbl> 12, 7, 12, 13, 12, 9, 8, 9, 10, 5, 13, 10, 8, 8, 10~
$ ch_tournament      <dbl> 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, ~
$ ch_other_correct   <dbl> 9, 4, 4, 7, 9, 4, 4, 4, 4, 4, 8, 4, 4, 6, 4, 6, 5, ~
$ ch_other_die       <dbl> 3, 3, 3, 6, 3, 3, 3, 3, 6, 1, 5, 6, 6, 2, 6, 2, 6, ~
$ ch_other_total     <dbl> 12, 7, 7, 13, 12, 7, 7, 7, 10, 5, 13, 10, 10, 8, 10~
$ ch_result          <dbl> 1, NA, NA, 1, 1, NA, NA, NA, NA, 1, 1, NA, NA, 1, N~
$ ch_points          <dbl> 12, 7, 12, 13, 12, 9, 8, 9, 10, 5, 13, 10, 8, 8, 10~
$ ct_selected        <dbl> 2, 2, 3, 6, 6, 2, 1, 1, 1, 6, 2, 1, 1, 1, 6, 6, 6, ~
$ ct_tails           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ ct_points          <dbl> 9.5, 9.5, 8.0, 1.0, 1.0, 9.5, 11.0, 11.0, 11.0, 1.0~
$ task_paid          <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 6, 6, 6, ~
$ complete           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ total_points       <dbl> 23, 23, 23, 23, 21, 18, 23, 23, 0, 22, 22, 0, 12, 2~
$ total_local_paid   <dbl> 330, 330, 330, 330, 310, 280, 330, 330, 100, 320, 3~
$ total_usd_paid     <dbl> 16.5, 16.5, 16.5, 16.5, 15.5, 14.0, 16.5, 16.5, 5.0~
$ final_version      <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ comment            <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
$ f_last_meal_time   <chr> "3 0", "4 0", "4 0", "1 30", "0 30", "2 0", "1 30",~
$ start_time_h       <dbl> 12, 12, 12, 12, 12, 12, 12, 12, 14, 14, 14, 14, 14,~
$ end_time_h         <dbl> 13, 13, 13, 13, 13, 13, 13, 13, 15, 15, 15, 15, 15,~
```

## 4.2 Basic Data Management

dplyr uses a collection of verbs to manipulate data that are piped (chained) into each other with a piping operator `%>%` from `magrittr` package. The way you use functions in base R is you wrap new function over the previous one, such as `k(g(f(x)))` this will become impossible to read very quickly as you stack up functions and their arguments. To solve this we will use pipes `x %>% f() %>% g() %>% k()`! Now you can clearly see that we take x and apply f(), then g(), then k(). Note: base R now also has its own pipe `|>`, but we will stick to `%>%` for compatibility across packages.

### 4.2.1 `select()`

`select()` selects only the columns that you want, removing all other columns. You can use column position (with numbers) or name. The columns will be displayed in the order you list them. We will select subject_id, temperature, gender and results of raven's matrices games.

- `id` is a unique subject identification number, where site_id.session_n.subject_n (001.001.001).
- `mean_temp_celsius` is mean temperature through the session
- `gender` is gender of the subject.
- `pr_correct` is number of correct answers in *piece-rate* round.
- `tr_correct` is number of correct answers in *tournament* round.
- `ch_correct` is number of correct answers in *choice* round.
- `ch_tournament` is 1 if participant decided to play tournament and 0 if choice.

```
data_raven <- data %>% select(id, mean_temp_celsius,gender, pr_correct, tr_correct, ch_tou
head(data_raven)
```

```
# A tibble: 6 x 7
  id        mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_co~1
  <chr>                 <dbl> <chr>       <dbl>      <dbl>         <dbl>   <dbl>
1 001018001              28.6 Female          7          3             1       9
2 001018002              28.6 Male            5          6             0       4
3 001018005              28.6 Male            6          7             0       7
4 001018009              28.6 Male            1          5             1       7
5 001018010              28.6 Male            7          9             1      10
6 001018013              28.6 Female          2          7             0       5
# ... with abbreviated variable name 1: ch_correct
```

You can also exclude columns or select everything else with select using -

```
data_raven %>% select(-gender) %>% head()
```

```
# A tibble: 6 x 6
  id        mean_temp_celsius pr_correct tr_correct ch_tournament ch_correct
  <chr>                 <dbl>      <dbl>      <dbl>         <dbl>      <dbl>
1 001018001              28.6          7          3             1          9
2 001018002              28.6          5          6             0          4
3 001018005              28.6          6          7             0          7
4 001018009              28.6          1          5             1          7
5 001018010              28.6          7          9             1         10
6 001018013              28.6          2          7             0          5
```

### 4.2.2 `filter()`

`filter()` keeps only rows that meet the specified criteria. Let's filter and make 2 data sets one for Males and, one for Females.

```
data_male <- data_raven %>% filter(gender == "Male") # we use == for comparison
data_female <- data_raven %>% filter(gender == "Female")
head(data_male)
```

```
# A tibble: 6 x 7
  id        mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_co~1
  <chr>                 <dbl> <chr>       <dbl>      <dbl>         <dbl>   <dbl>
1 001018002              28.6 Male            5          6             0       4
2 001018005              28.6 Male            6          7             0       7
3 001018009              28.6 Male            1          5             1       7
4 001018010              28.6 Male            7          9             1      10
5 001019002              30.7 Male            6          7             1       4
6 001019008              30.7 Male            5          6             1       6
# ... with abbreviated variable name 1: ch_correct
```

```
head(data_female)
```

```
# A tibble: 6 x 7
  id        mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_co~1
  <chr>                 <dbl> <chr>       <dbl>      <dbl>         <dbl>   <dbl>
1 001018001              28.6 Female          7          3             1       9
```

```
2 001018013              28.6 Female          2          7             0       5
3 001018014              28.6 Female          6          6             0       7
4 001018015              28.6 Female          7          7             0       6
5 001019001              30.7 Female          5          4             0       4
6 001019003              30.7 Female          8          6             1       8
# ... with abbreviated variable name 1: ch_correct
```

We can chain multiple requirements. Here we will look at Males, temperature over 30 celsius or Females, temperature below 30. Notice that we use `&` as "and", `|` as "or, and wrap the two conditions into"()" to avoid confusion. It could read it as `mean_temp_celsius > 30 | gender == "Female"`.

```
data_raven %>%
  filter(
    (gender == "Male" & mean_temp_celsius > 30) | (gender == "Female" & mean_temp_celsius
        )
```

```
# A tibble: 61 x 7
   id        mean_temp_celsius gender pr_correct tr_correct ch_tournam~1 ch_co~2
   <chr>                 <dbl> <chr>       <dbl>      <dbl>        <dbl>   <dbl>
 1 001018001              28.6 Female          7          3            1       9
 2 001018013              28.6 Female          2          7            0       5
 3 001018014              28.6 Female          6          6            0       7
 4 001018015              28.6 Female          7          7            0       6
 5 001019002              30.7 Male            6          7            1       4
 6 001019008              30.7 Male            5          6            1       6
 7 001019009              30.7 Male            2          6            1       5
 8 001019010              30.7 Male            6          8            0       8
 9 001019011              30.7 Male            7          7            1       7
10 001019012              30.7 Male            4          5            1       6
# ... with 51 more rows, and abbreviated variable names 1: ch_tournament,
#   2: ch_correct
```

### 4.2.3 `arrange()`

`arrange()` allows you to order the table using a variable. Let's see which subject scored the worst in `pr_correct`.

```
data_raven %>% arrange(pr_correct) %>% head()
```

```
# A tibble: 6 x 7
  id        mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_co~1
  <chr>                 <dbl> <chr>       <dbl>      <dbl>         <dbl>   <dbl>
1 001018009              28.6 Male            1          5             1       7
2 001018013              28.6 Female          2          7             0       5
3 001019004              30.7 Female          2          3             0       6
4 001019005              30.7 Female          2          6             0       5
5 001019009              30.7 Male            2          6             1       5
6 001021013              30.4 Male            2          3             1       2
# ... with abbreviated variable name 1: ch_correct
```

We can also sort in descending order using `desc()` modifier. Let's look at who scored the most!

```
data_raven %>% arrange(desc(pr_correct)) %>% head()
```

```
# A tibble: 6 x 7
  id        mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_co~1
  <chr>                 <dbl> <chr>       <dbl>      <dbl>         <dbl>   <dbl>
1 001019003              30.7 Female          8          6             1       8
2 001020002              31.6 Male            8          6             1       5
3 001020013              31.6 Male            8          7             0       7
4 001021006              30.4 Male            8          4             0       7
5 001025009              26.8 Male            8          9             1       9
6 001027015              32.2 Female          8          7             0       8
# ... with abbreviated variable name 1: ch_correct
```

### 4.2.4 `mutate()`

`mutate()` adds new columns and modifies current variables in the data set. Let's create a dataset with 10 rows and make three new variable columns as an example

```
tibble(rows = 1:10) %>% mutate(
  One = 1,
  Comment = "Something",
  Approved = TRUE
)
```

```
# A tibble: 10 x 4
    rows   One Comment   Approved
```

```
   <int> <dbl> <chr>     <lgl>
 1     1     1 Something TRUE
 2     2     1 Something TRUE
 3     3     1 Something TRUE
 4     4     1 Something TRUE
 5     5     1 Something TRUE
 6     6     1 Something TRUE
 7     7     1 Something TRUE
 8     8     1 Something TRUE
 9     9     1 Something TRUE
10    10     1 Something TRUE
```

`mutate()` can use existing variables from the data set to create new ones! Lets convert Celsius to Fahrenheit, see how many point more people scored in tournament over piece-rate round, and check how far from the mean they scored in piece-rate round!

```
data_raven %>% mutate(mean_temp_fahrenheit = (mean_temp_celsius * 9/5) + 32,
                      impovement = tr_correct - pr_correct,
                      pr_deviation = pr_correct - mean(pr_correct))
```

```
# A tibble: 114 x 10
   id     mean_~1 gender pr_co~2 tr_co~3 ch_to~4 ch_co~5 mean_~6 impov~7 pr_de~8
   <chr>    <dbl> <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
 1 00101~    28.6 Female       7       3       1       9    83.5      -4   1.66
 2 00101~    28.6 Male         5       6       0       4    83.5       1  -0.342
 3 00101~    28.6 Male         6       7       0       7    83.5       1   0.658
 4 00101~    28.6 Male         1       5       1       7    83.5       4  -4.34
 5 00101~    28.6 Male         7       9       1      10    83.5       2   1.66
 6 00101~    28.6 Female       2       7       0       5    83.5       5  -3.34
 7 00101~    28.6 Female       6       6       0       7    83.5       0   0.658
 8 00101~    28.6 Female       7       7       0       6    83.5       0   1.66
 9 00101~    30.7 Female       5       4       0       4    87.2      -1  -0.342
10 00101~    30.7 Male         6       7       1       4    87.2       1   0.658
# ... with 104 more rows, and abbreviated variable names 1: mean_temp_celsius,
#   2: pr_correct, 3: tr_correct, 4: ch_tournament, 5: ch_correct,
#   6: mean_temp_fahrenheit, 7: impovement, 8: pr_deviation
```

Notice that we can nest functions within `mutate()`: first we took `mean()` of the entire column and then subtracted it from `pr_correct`.

### 4.2.5 recode()

recode() modifies the values within a variable. Here is a template:

data %>% mutate(Variable = recode(Variable, "old value" = "new value"))

Let's use recode() to change "Male" to "M" and "Female" to "F".

```
data_raven <- data_raven %>% mutate(gender = recode(gender, "Male" = "M", "Female" = "F"))
```

## 4.3 summarize()

summarize() collapses all rows and returns a one-row summary. We will use summary to calculation what percentage of participant were male, median score in piece-rate round, max score in tournament, percentage of people choosing tournament in choice and mean score in choice round.

```
data_raven %>%
  summarize(perc_male = sum(gender == "Male", na.rm = T) / n(),
            pr_median = median(pr_correct),
            tr_max = max(tr_correct),
            ch_ratio = sum(ch_tournament) / n(),
            ch_mean = mean(ch_correct))
```

```
# A tibble: 1 x 5
  perc_male pr_median tr_max ch_ratio ch_mean
      <dbl>     <dbl>  <dbl>    <dbl>   <dbl>
1         0         5      9    0.456    6.01
```

## 4.4 group_by() and ungroup()

### 4.4.1 group_by()

group_by() groups data by specific variables for future operations. We can use group_by() and summarize() to calculate different summary statistics for genders!

```
data_raven %>%
  drop_na(gender) %>%  # removes NA gender
  group_by(gender) %>%
```

```
    summarize(pr_mean = mean(pr_correct),
              tr_mean = mean(tr_correct),
              ch_mean = mean(ch_correct),
              pr_sd = sd(pr_correct),
              n = n()) %>%
    ungroup()
```

```
# A tibble: 2 x 6
  gender pr_mean tr_mean ch_mean pr_sd     n
  <chr>    <dbl>   <dbl>   <dbl> <dbl> <int>
1 F         5.22    6.17    5.93  1.58    58
2 M         5.47    6.4     6.09  1.59    55
```

Let's group by gender and choice in choice round and look at points in choice round!

```
  data_raven %>%
    drop_na(gender) %>%  # removes NA gender
    group_by(gender, ch_tournament) %>%
    summarize(ch_mean = mean(ch_correct),
              pr_sd = sd(ch_correct),
              n = n()) %>%
    ungroup()
```

```
# A tibble: 4 x 5
  gender ch_tournament ch_mean pr_sd     n
  <chr>          <dbl>   <dbl> <dbl> <int>
1 F                  0    5.73  1.70    33
2 F                  1    6.2   1.73    25
3 M                  0    6.07  1.69    29
4 M                  1    6.12  2.25    26
```

### 4.4.2 `ungroup()`

`ungroup()` does exactly what you think – removes the grouping! Always ungroup your data after you are done with operation that required grouping, else it will get messy. Look at this example

```
  data_raven %>%
    drop_na(gender) %>%  # removes NA gender
```

```
    group_by(gender) %>%
    mutate(n = n()) %>%
    mutate(mean_male = mean(gender == "Male")) %>%
    ungroup() %>%
    select(id, gender, n, mean_male) %>% head(n = 5)
```

```
# A tibble: 5 x 4
  id        gender     n mean_male
  <chr>     <chr>  <int>     <dbl>
1 001018001 F         58         0
2 001018002 M         55         0
3 001018005 M         55         0
4 001018009 M         55         0
5 001018010 M         55         0
```

Notice how mean_male (ratio of male to total) is 0 for Female and 1 for Male. It is because the data was grouped and we performed operation on Males and Females separately.

```
  data_raven %>%
    drop_na(gender) %>%  # removes NA gender
    group_by(gender) %>%
    mutate(n = n()) %>%
    ungroup() %>%
    mutate(mean_male = mean(gender == "Male")) %>%
    select(id, gender, n, mean_male) %>% head(n = 5)
```

```
# A tibble: 5 x 4
  id        gender     n mean_male
  <chr>     <chr>  <int>     <dbl>
1 001018001 F         58         0
2 001018002 M         55         0
3 001018005 M         55         0
4 001018009 M         55         0
5 001018010 M         55         0
```

This time on the other hand we ungrouped the data, correctly calculating the ratio!

### 4.4.3 `rowwise()`

`rowwise()` produces a row-wise grouping. Later you might want to run a calculation row-wise instead of column-wise, but your column will be filled with an aggregate result. This is where `rowwise()` comes in to save the day! To demonstrate, we will make a dataframe with a column of lists and try to find the length of each list.

```
df <- tibble(
  x = list(1, 2:3, 4:6,7:11)
)

df %>% mutate(length = length(x))
```

```
# A tibble: 4 x 2
  x         length
  <list>     <int>
1 <dbl [1]>      4
2 <int [2]>      4
3 <int [3]>      4
4 <int [5]>      4
```

Hmmm… Instead of lists' lengths we got the total number of rows in the data set (length of column `x`). Now let's use `rowwise()`

```
df %>% rowwise() %>%
  mutate(length = length(x))
```

```
# A tibble: 4 x 2
# Rowwise:
  x         length
  <list>     <int>
1 <dbl [1]>      1
2 <int [2]>      2
3 <int [3]>      3
4 <int [5]>      5
```

Yey! Here R runs `length()` on each list separately, giving us correct lengths! Alternatively you can use `lengths()`, which loops `length()` over each list.

## 4.5 count()

`count()` is similar to `group_by()` and `summarize()` combo – it collapses the rows and counts the number of observations per group of values.

```
data_raven %>% count(gender)
```

```
# A tibble: 3 x 2
  gender     n
  <chr>  <int>
1 F         58
2 M         55
3 <NA>       1
```

Similar to `group_by()` you can select multiple columns to group.

```
data_raven %>% count(gender, ch_tournament)
```

```
# A tibble: 5 x 3
  gender ch_tournament     n
  <chr>          <dbl> <int>
1 F                  0    33
2 F                  1    25
3 M                  0    29
4 M                  1    26
5 <NA>               1     1
```

## 4.6 rename()

`rename()` renames a column. Notice that the "New Name" is on the left and "Old Name" is on the right. Let's rename `id` to `subject_id` and `gender` to `sex`.

```
data_raven %>% rename("subject_id" = "id", "sex" = "gender") %>% head(n = 5)
```

```
# A tibble: 5 x 7
  subject_id mean_temp_celsius sex   pr_correct tr_correct ch_tournament ch_co~1
  <chr>                  <dbl> <chr>      <dbl>      <dbl>         <dbl>   <dbl>
```

```
1 001018001               28.6 F              7          3              1       9
2 001018002               28.6 M              5          6              0       4
3 001018005               28.6 M              6          7              0       7
4 001018009               28.6 M              1          5              1       7
5 001018010               28.6 M              7          9              1      10
# ... with abbreviated variable name 1: ch_correct
```

## 4.7 `row_number()`

`row_number()` fills a column with consecutive numbers. It is especially useful if you need to create an id column. Let's remove `id` column and make a new one with `row_number()`.

```
data_raven %>%
  select(-id) %>%
  mutate(id = row_number()) %>%
  relocate(id) %>% # used to move id at the beginning
  head(n = 5)
```

```
# A tibble: 5 x 7
     id mean_temp_celsius gender pr_correct tr_correct ch_tournament ch_correct
  <int>             <dbl> <chr>       <dbl>      <dbl>         <dbl>      <dbl>
1     1              28.6 F               7          3             1          9
2     2              28.6 M               5          6             0          4
3     3              28.6 M               6          7             0          7
4     4              28.6 M               1          5             1          7
5     5              28.6 M               7          9             1         10
```

# 5 Tidy Data

Wide data and long data are two common forms of data organization.

Wide data, also known as unstacked data, is organized so that each row represents an individual unit of observation and each column represents a variable. This format is often used when the number of variables is relatively small and they are not related to one another in a hierarchical manner.

For example, a wide data table containing information about a group of students might have one row for each student and columns for their name, age, gender, and test scores.

On the other hand, long data, also known as stacked data, is organized so that each row represents a single observation of a variable, and each column represents the variable and an identifier of the unit of observation. This format is often used when the number of variables is large or they are related to one another in a hierarchical manner.

For example, a long data table containing information about the test scores of a group of students might have one row for each student-test combination, with columns for the student's name, the test name and the score.

Choosing between wide and long format depends on the specific analysis and visualization needs. Wide format is more suited for simple exploratory analysis, while long format is more suited for complex analysis and visualization. We will take advantage of long and wide data in visualization section.

A common problem is a dataset where column names are not names of variables, but *value* of a variable. This is true for `pr_correct`, `tr_correct`, `ch_correct` as the column names represent name of the `game` variable, the values in the columns represent number of correct answers, and each row represents two observations, not one.

To tidy a dataset, we need to pivot the offending columns into a new pair of variables. To carry out the operation we need to supply:

1. The columns whose names are values, not variables – columns we want to pivot. Here `pr_correct`, `tr_correct`, `ch_correct`.
2. The name of the variable to move the column names to. Here `game`. The default `name`.
3. The name of the variable to move the column values to. Here `n_correct`. The default `value`.

```
data_raven %>%
  pivot_longer(c(pr_correct, tr_correct, ch_correct), names_to = "game", values_to = "n_co
  select(id,game,n_correct) %>%
  head(n = 5)
```

```
# A tibble: 5 x 3
  id        game       n_correct
  <chr>     <chr>          <dbl>
1 001018001 pr_correct         7
2 001018001 tr_correct         3
3 001018001 ch_correct         9
4 001018002 pr_correct         5
5 001018002 tr_correct         6
```

## 5.1 pivot_wider()

`pivot_wider()` is the opposite of `pivot_longer()`. You use it when an observation is scattered across multiple rows. For example, let's have a look at `data_raven_accident`, where gnomes stacked `mean_temp_celsius` and `ch_tournament`. Here an observation is spread across two rows.

```
data_raven_accident %>% head(n=5)
```

```
# A tibble: 5 x 3
  id        name              value
  <chr>     <chr>             <dbl>
1 001018001 mean_temp_celsius  28.6
2 001018001 ch_tournament       1
3 001018002 mean_temp_celsius  28.6
4 001018002 ch_tournament       0
5 001018005 mean_temp_celsius  28.6
```

To tidy this up, we need two parameters:

1. The column to take variable names from. Here, it's `name`.
2. The column to take values from. Here it's `value`.

```
data_raven_accident %>% pivot_wider(names_from = name, values_from = value) %>% head(n = 5
```

```
# A tibble: 5 x 3
  id        mean_temp_celsius ch_tournament
  <chr>                 <dbl>         <dbl>
1 001018001              28.6             1
2 001018002              28.6             0
3 001018005              28.6             0
4 001018009              28.6             1
5 001018010              28.6             1
```

It is evident from their names that pivot_wider() and pivot_longer() are inverse functions. pivot_longer() converts wide tables to a longer and narrower format; pivot_wider() converts long tables to a shorter and wider format.

## 5.2 `tibble()` and `tribble()`

A tibble is a special kind of data frame in R. Tibbles are a modern re-imagining of the data frame, designed to be more friendly and consistent than traditional data frames.

- Tibbles are similar to data frames in that they can hold tabular data, but they have some key differences:
- Tibbles display only the first 10 rows by default when printed, making them easier to work with large datasets.
- Tibbles use a consistent printing format, making it easier to work with multiple tibbles in the same session.
- Tibbles have a consistent subsetting behavior, making it easier to select columns by name. When printed, the data type of each column is specified.
- Subsetting a tibble will always return a tibble. You don't need to use drop = FALSE compared to traditional data.frames.
- And most importantly, you can have column consisting of lists.

In summary, Tibbles are a more modern and consistent version of data frames, they are less prone to errors and more readable, making them a great choice for data manipulation and exploration tasks.

To make a tibble we can use `tibble()` similar to data.frame()

```
tibble(x = c(1,2,3),
       y = c('one', 'two', 'three'))
```

```
# A tibble: 3 x 2
      x y
  <dbl> <chr>
1     1 one
2     2 two
3     3 three
```

You can also you `tribble()` for creating a row-wise, readable tibble in R. This is useful for creating small tables of data. **Syntax: tribble (~column1, ~column2)** where, Row column — represents the data in row by row layout.

```
tribble(~x, ~y,
        1, "one",
        2, "two",
        3, "three")
```

```
# A tibble: 3 x 2
      x y
  <dbl> <chr>
1     1 one
2     2 two
3     3 three
```

## 5.3 `janitor` Clean Your Data

`janitor` is designed to make the process of cleaning and tidying data as simple and efficient as possible. To learn more about the function check out this vignette! If you are interested in making summary tables with `janitior` check this vignette!

### 5.3.1 `clean_names()`

`clean_names()` is used to clean variable names, particularly those that are read in from Excel files using `readxl::read_excel()` and `readr::read_csv()`. It parses letter cases, separators, and special characters to a consistent format, converts certain characters like "%" to "percent" and "#" to "number" to retain meaning, resolves duplicate names and empty names. It is recommended to call this function every time data is read.

```
# Create a data.frame with dirty names
test_df <- as.data.frame(matrix(ncol = 6))
```

```
  names(test_df) <- c("camelCase", "ábc@!*", "% of respondents (2009)",
                      "Duplicate", "Duplicate", "")
  test_df %>% colnames()
```

```
[1] "camelCase"               "ábc@!*"
[3] "% of respondents (2009)" "Duplicate"
[5] "Duplicate"               ""
```

```
  library(janitor)
  test_df %>% clean_names() %>% colnames()
```

```
[1] "camel_case"                  "abc"
[3] "percent_of_respondents_2009" "duplicate"
[5] "duplicate_2"                 "x"
```

### 5.3.2 `remove_empty()`

`remove_empty()` removes empty row and columns, especially useful after reading Excel files.

```
  test_df2 <- data.frame(numbers = c(1, NA, 3),
                         food = c(NA, NA, NA),
                         letters = c("a", NA, "c"))
  test_df2
```

```
  numbers food letters
1       1   NA       a
2      NA   NA    <NA>
3       3   NA       c
```

```
  test_df2 %>%
    remove_empty(c("rows", "cols"))
```

```
  numbers letters
1       1       a
3       3       c
```

### 5.3.3 `remove_constant()`

`remove_constant()` drops columns from a data.frame that contain only a single constant value (with an na.rm option to control whether NAs should be considered as different values from the constant).

```
test_df3 <- data.frame(cool_numbers = 1:3, boring = "the same")
test_df3
```

```
  cool_numbers   boring
1            1 the same
2            2 the same
3            3 the same
```

```
test_df3 %>% remove_constant()
```

```
  cool_numbers
1            1
2            2
3            3
```

###`convert_to_date()` and `convert_to_datetime()` Remember loading data from Excel and seeing `36922.75` instead of dates? Well, `convert_to_date()` and `convert_to_datetime()` will convert this format and other date-time formats to actual dates! If you need more customization check `excel_numeric_to_date()`.

```
convert_to_date(36922.75)
```

```
[1] "2001-01-31"
```

```
convert_to_datetime(36922.75)
```

```
[1] "2001-01-31 18:00:00 UTC"
```

###`row_to_names()` `row_to_names()` is a function that takes the names of variables stored in a row of a data frame and makes them the column names of the data frame. It can also remove the row that contained the names, and any rows above it, if desired.

```r
test_df4 <- data.frame(X_1 = c(NA, "Names", 1:3),
                       X_2 = c(NA, "Value", 4:6))
test_df4
```

```
    X_1   X_2
1  <NA>  <NA>
2 Names Value
3     1     4
4     2     5
5     3     6
```

```r
row_to_names(test_df4, 2)
```

```
  Names Value
3     1     4
4     2     5
5     3     6
```

# 6 Relational Databases

# Part IV

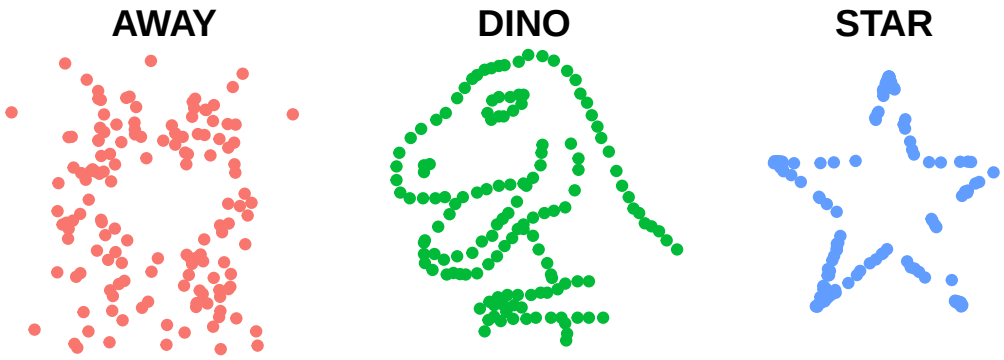
# Presenting the Data

# 7 Data Visualization

Data visualization is an essential tool for understanding and communicating complex information. There are two main types of visualization: 1. Exploratory It is common to look at summary statistics such as mean and standard deviation. But these numbers obscure the datapoints hiding the form of our datasets. Matejka and Fitzmaurice generated datasets with Identical Statistics that look distinctly different. You can access all 12 patterns with `datasauRus` package. It is important to *see* the structure to move your analyses forward.

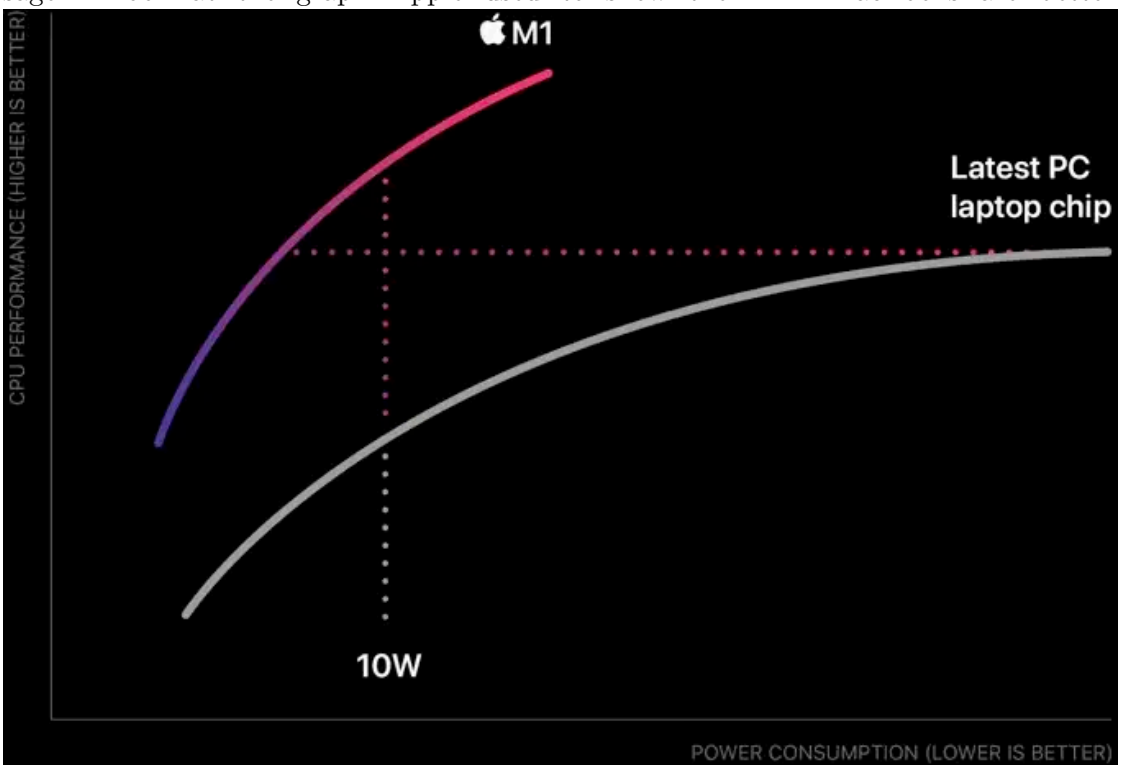
| mean_x | mean_y | std_dev_x | std_dev_y | corr_x_y |
|---|---|---|---|---|
| 54.26 | 47.83 | 16.77 | 26.94 | -0.06 |

```
datasauRus::datasaurus_dozen %>% filter(dataset %in% c("away","dino", "star")) %>%
mutate(dataset = str_to_upper(dataset)) %>%
ggplot(aes(x = x, y = y, colour = dataset)) +
  geom_point() +
  theme_void(base_size = 18) +
  theme(legend.position = "none",
        strip.text = element_text(face = "bold")) +
  facet_wrap(~dataset, ncol = 3) +
  coord_fixed(ratio = 0.8)
```




2. Explanatory So, you got your results together and now you need to not only present them, but also convince non-techical audience. They don't care whether your model user cross-validation or how you optimized your gradient boosted forest, all they want is a convincing simple message. That is why you won't

see fency overloaded graphs in forward facing presentation it all about the message. Look at the graph Apple used to show their M1 MacBooks are better.




R offers a variety of packages for creating visually appealing and informative plots. One of the most popular and versatile packages for data visualization in R is `ggplot2`. We will explore the basics of using `ggplot2` to create different types of plots and customize them to suit your needs. We can load it separately `libary(ggplot2)` or with `libary(tidyverse)`.

## 7.1 Grammar of Graphics

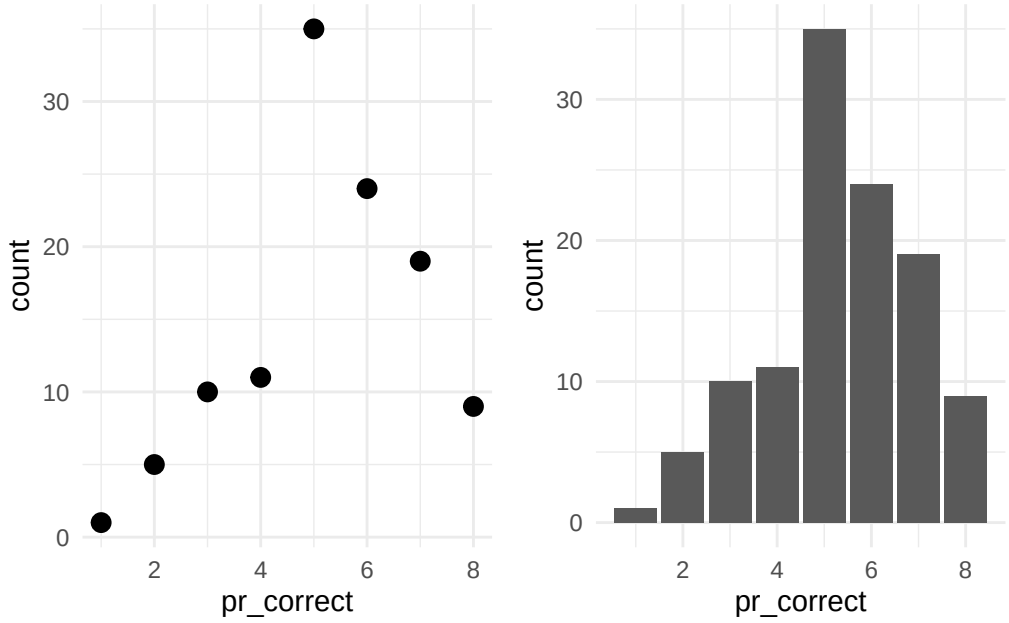
The Grammar of Graphics is a concept in data visualization that was developed by Leland Wilkinson in his book "The Grammar of Graphics" in 1999. The Grammar of Graphics is essentially a system of rules that describes how to represent data visually using a set of graphical elements and mappings between data variables and visual properties.

"Excel Enjoyers" are familiar with the Excel plotting workflow: you select a plot you want and it just produces one for you.

Under this framework scatter plot and bar plots appear completely different:


Figure 7.1: “Excel GUI”

```
point_plot <- data_raven %>% count(pr_correct, name = "count") %>% ggplot(aes(x = pr_corre
col_plot <- data_raven %>% count(pr_correct, name = "count") %>% ggplot(aes(x = pr_correct
point_plot + col_plot
```




However, under the Grammar of Graphics we see how similar these graphics are! They are exactly the same in terms everything, but geometries! The first one use “points” while the second uses “columns” to display the data.

The Grammar of Graphics provides a framework for creating complex visualizations by breaking down the visualization process into a set of components.

1. Data: The information that is being visualized. To explore how grammar of graphics works in `ggplot2` we will use `iris` dataset, which is a built-in dataset of measurements of different parts of iris flowers.

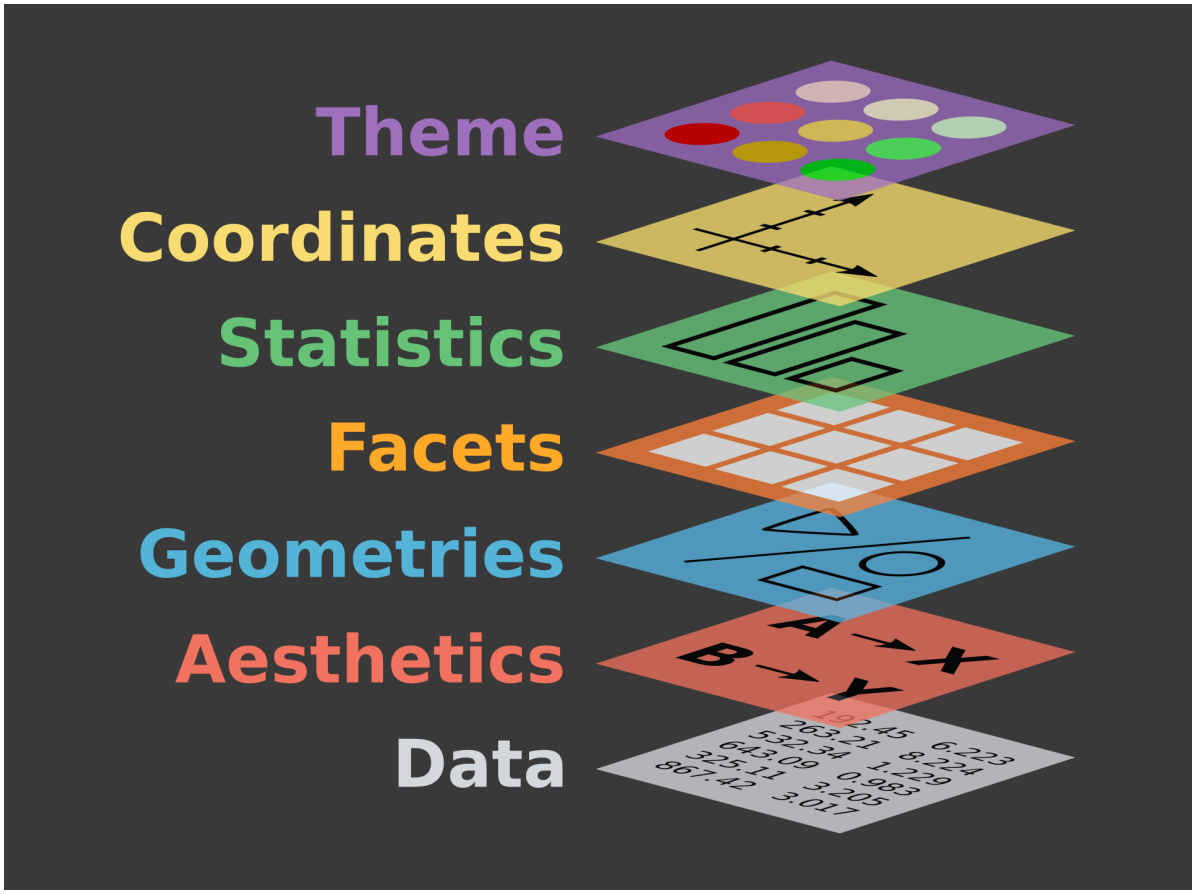



Figure 7.2: “Grammar of Graphics Visual, from QCBS R Workshop 3”

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

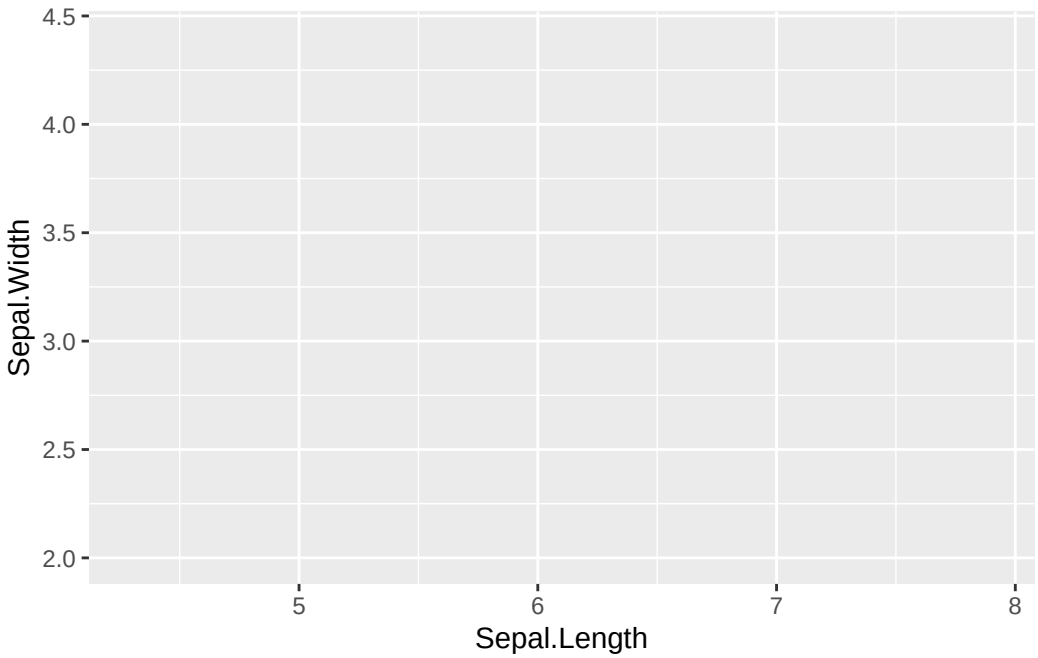
The Data layer of the graph is just a blank canvas. Becase we have not specified any graphing elements yet.

```
data_layer <- ggplot(data = iris)
data_layer
```



2. Aesthetics: The visual properties used to represent the data, such as x, y, color or size. Once we add aethetics we see out plotting area being set up and if we check mapping we see that `Sepal.Length` was assigned to `x` and `Sepal.Width` was assigned to `y`.

```
aes_layer <- ggplot(data = iris,
                    aes(x = Sepal.Length, y = Sepal.Width))
aes_layer
```

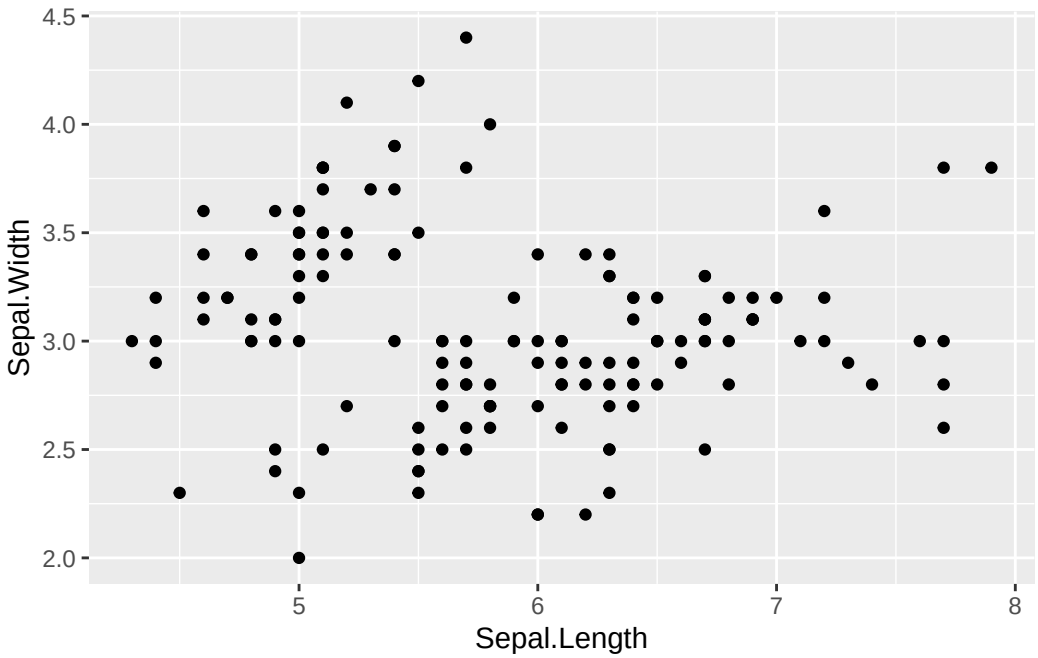



```
aes_layer$mapping
```

```
Aesthetic mapping:
* `x` -> `Sepal.Length`
* `y` -> `Sepal.Width`
```

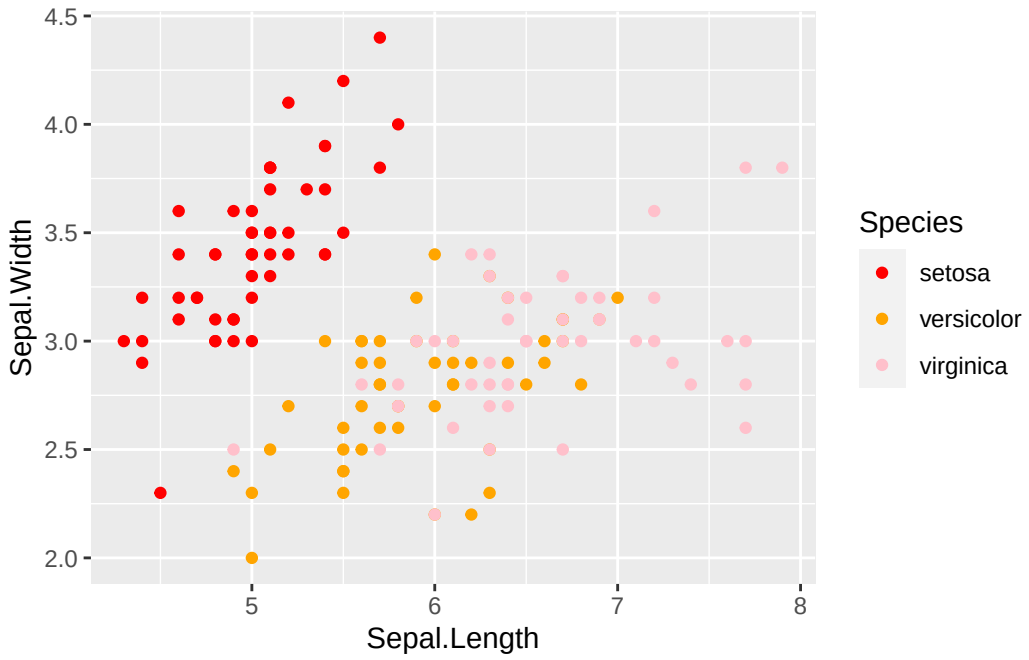
3. Geometries: The visual elements used to represent the data, such as points or bars. Once we add geometry we start seeing our data!

```
geometry_layer <- aes_layer + geom_point()
geometry_layer
```




4. Scales: The mapping between the data and the aesthetics, such as how numeric values are mapped to positions on a graph. There are different scales for color, fill, size, log(x), etc. Here we added scale `color`. Checking the mapping we see `Species` is mapped to `colour`.

```
scales_layer <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species
  geom_point() +
  # We case scale function to edit the scale for example we can set our own color manually
  scale_color_manual(values = c("red","orange","pink"))
scales_layer
```

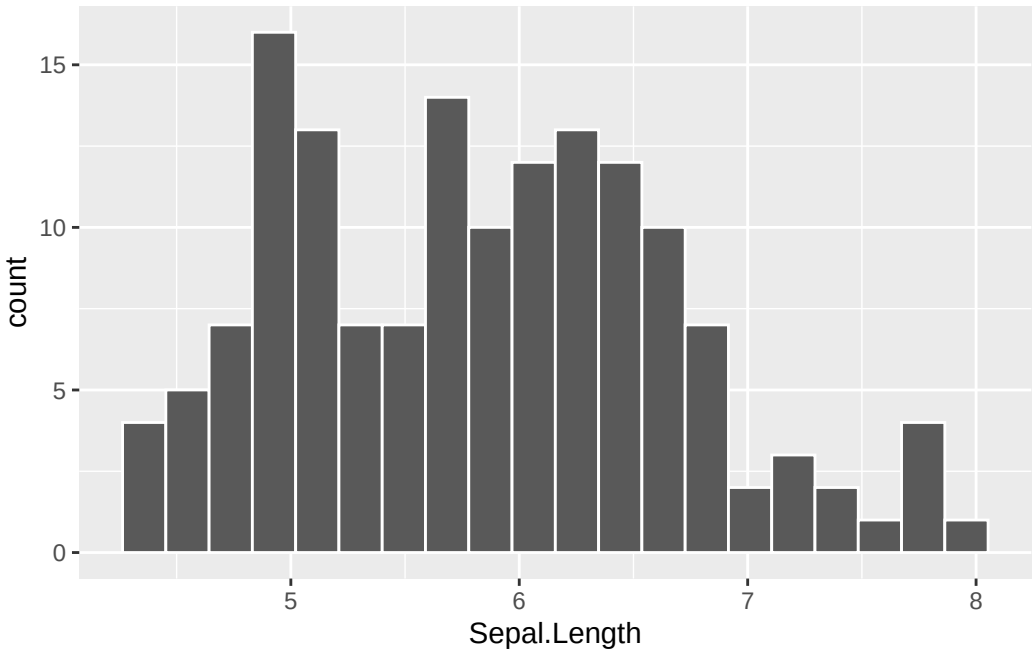



```
scales_layer$mapping
```

```
Aesthetic mapping:
* `x`      -> `Sepal.Length`
* `y`      -> `Sepal.Width`
* `colour` -> `Species`
```

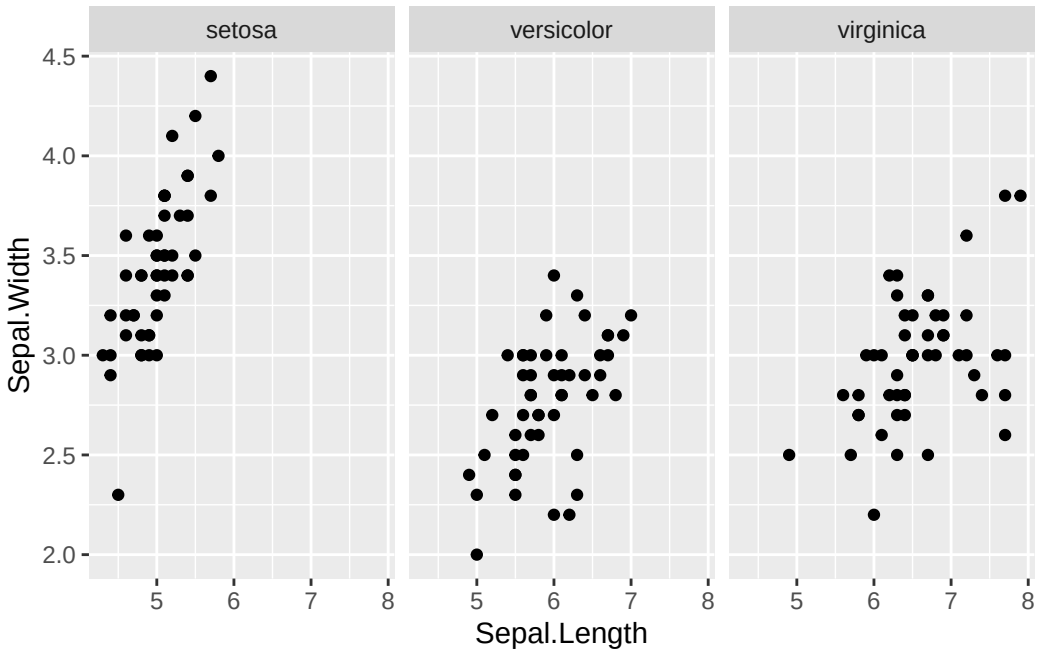
5. Statistics: Mathematical transformations applied to the data before visualization, such as summary statistics or new variables. Histogram for example *splits* data into bins and *counts* observations.

```
stat_layer <- ggplot(data = iris, aes(x = Sepal.Length)) +
  geom_histogram(bins = 20, color = "white")
stat_layer
```



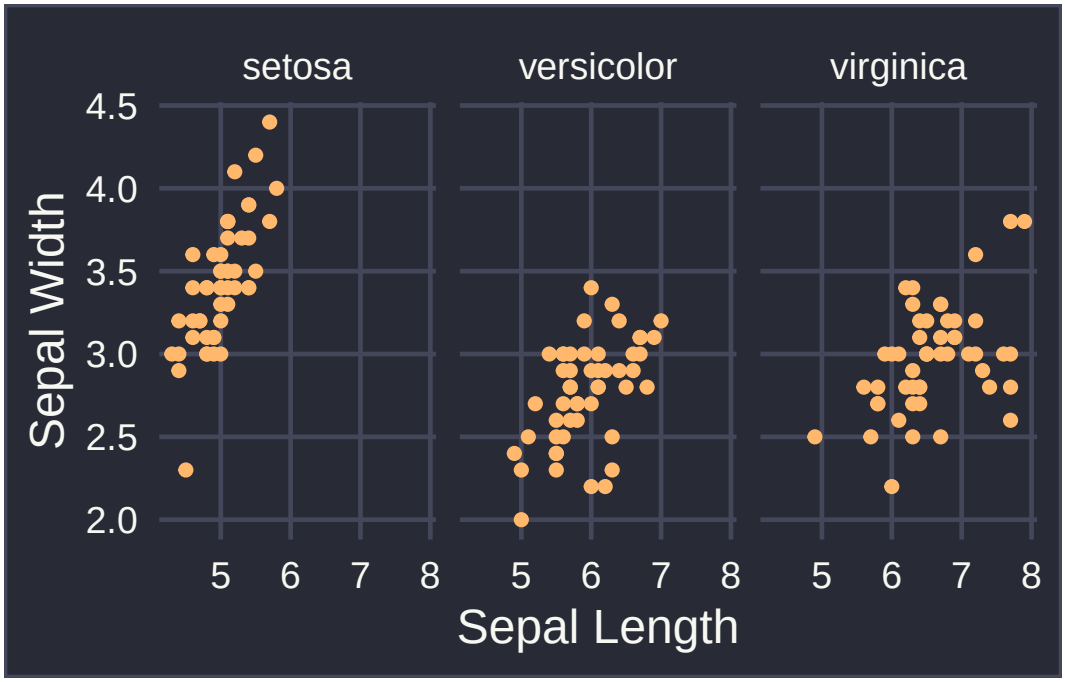

6. Facets: Ways of dividing the data into subgroups and creating separate visualizations for each subgroup.

```
facets_layer <- geometry_layer + facet_wrap(vars(Species), ncol = 3)
facets_layer
```




7. Theme: Adding Polishing touches to your visual and making it look exactly the way you want.

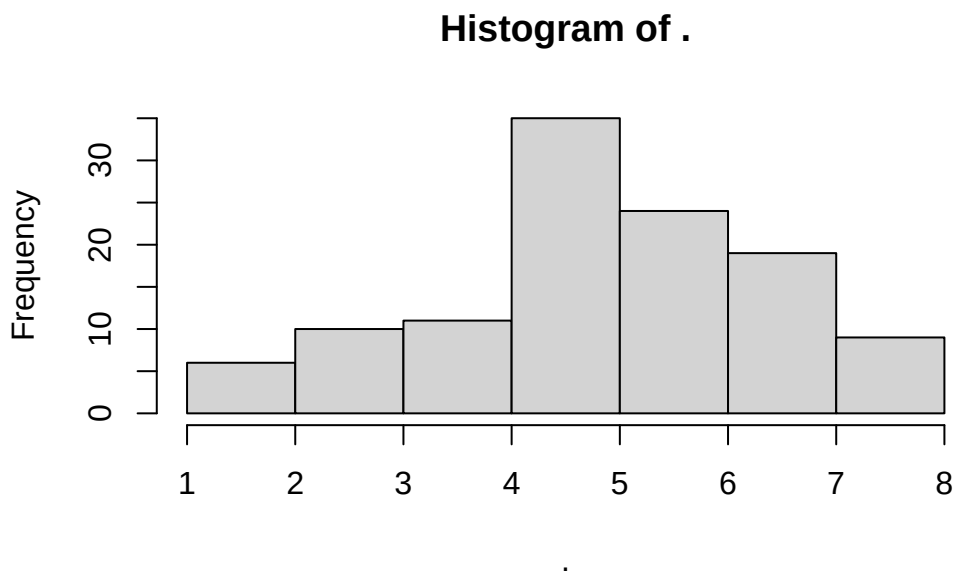
```
theme_layer <- facets_layer + theme_minimal(base_size = 18) +
  geom_point(size = 2, color = "#ffb86c") +
  theme(plot.background = element_rect(fill = "#282a36", color = "#44475A"),
        axis.text = element_text(color = "#f8f8f2"),
        axis.title = element_text(color = "#f8f8f2"),
        strip.text = element_text(color = "#f8f8f2"),
        panel.grid.minor = element_blank(),
        panel.grid.major = element_line(colour = "#44475a")
        ) +
  labs(x = "Sepal Length", y = "Sepal Width")
theme_layer
```



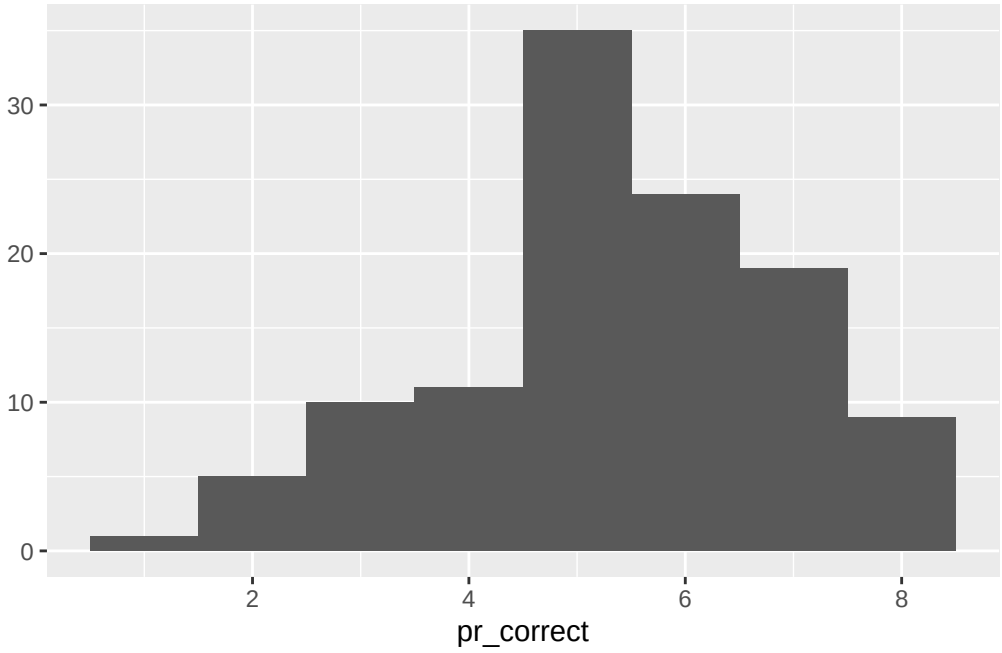

## 7.2 `ggplot()`

If you used R before then you are familiar with the default graphing function `plot`,`hist`, etc. `ggplot2` has it own version of quickly making a graph `qplot()`. To learn about `qplot()` check out this vignette.

```
data_raven %>% pull(pr_correct) %>% hist()
```



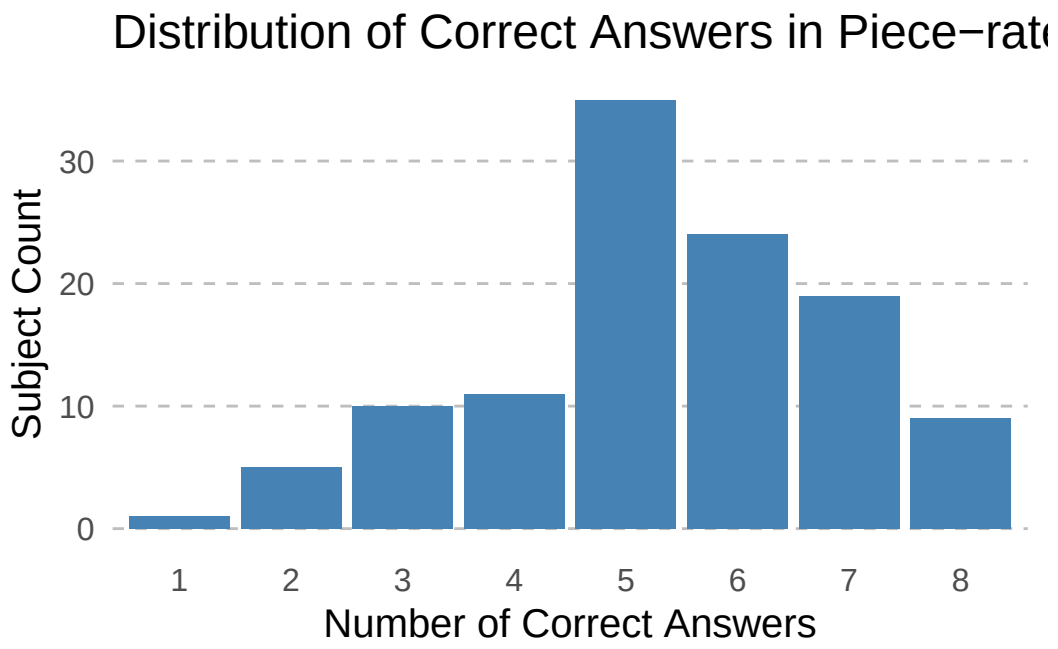

```
data_raven %>% qplot(pr_correct, data = ., geom = 'histogram', bins = length(unique(data_r
```




The `ggplot()` function sets up the basic structure of a plot, and additional layers, such as points, lines, and facets, can be added using `+` operator (like `%>%`, but for `+`). This makes it easy to understand, modify the code, and build complex plots by adding layers. This allows

for easy creation of plots that reveal patterns in the data. In contrast, the basic R plotting functions and `qplot()` have a simpler and less expressive syntax, making it harder to create complex and multi-layered plots. Mastering `ggplot()` is well worth your time and effort as it will teach you how to think about graphs and what goes into building them. For example, let's improve the histogram from earlier!

```
data_raven %>%
  count(pr_correct) %>% # I prefer calculating statistics myself
  ggplot(aes(x = as.factor(pr_correct), y = n)) + # We use aes to set x and y
  geom_col(fill = "steelblue") +
  theme_minimal(base_size = 15) +
  theme(panel.grid = element_blank(),
        panel.grid.major.y = element_line(linewidth = 0.5, linetype = 2, color = "grey"))
  labs(x = "Number of Correct Answers",
       y = "Subject Count",
       title = "Distribution of Correct Answers in Piece-rate Game")
```




Ah much better! We added labels, removed unnecessary grid lines, and added some color. If you want to learn more about ggplot check out ggplot2: Elegant Graphics for Data Analysis) and the cheatsheet.

We can use an amazing package `esquisse` to build our plots with drag-and-drop!

```
# install.packages('esquisse')
library(esquisse)
```

You can access `esquisse` by going to "Addins" in the top panel or with `esquisser(your_data)`. Now go learn more about this package here.

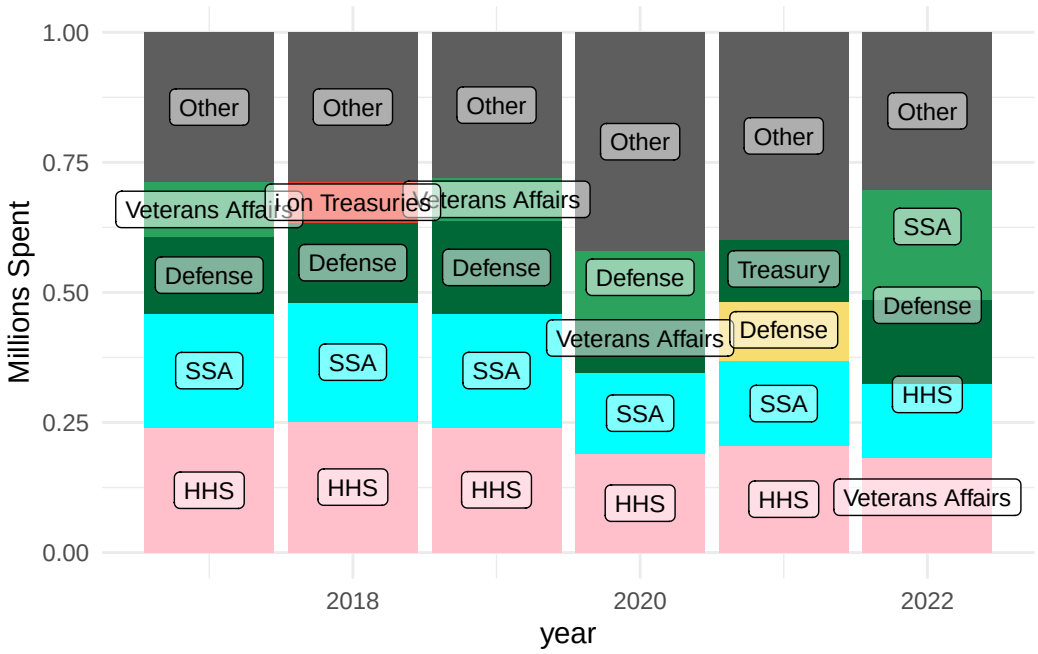
## 7.3 Tips

### 7.3.1 `group`

Usually ggplot groups your data by one the aesthetics you provided such as `color` and `fill`; however, sometimes it fails to do so. When that happens it is worth specifying group argument on your own.

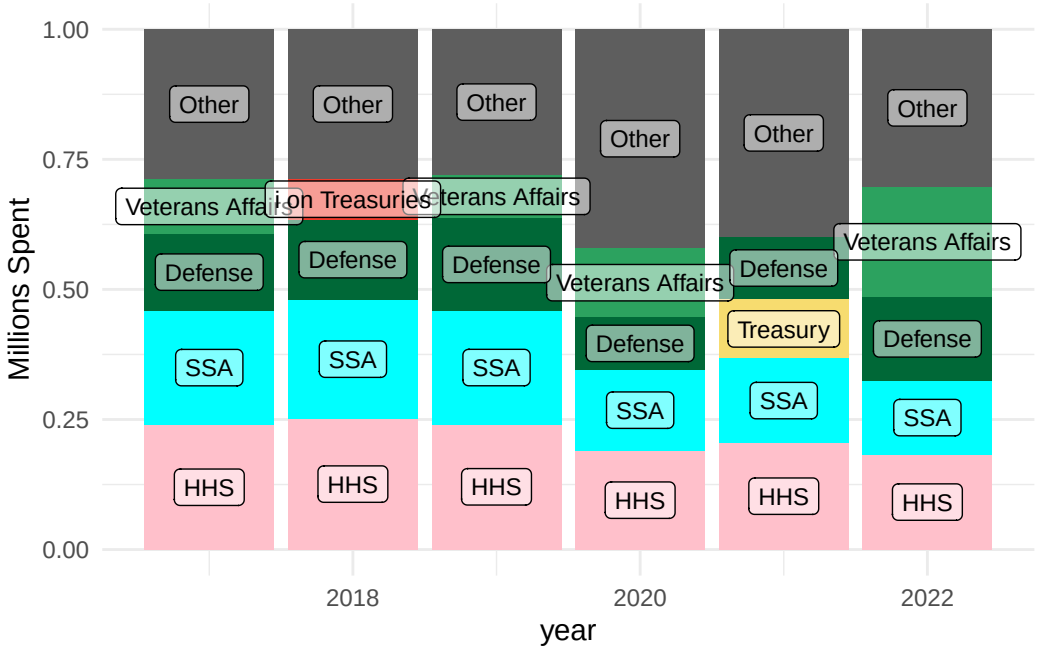
Notice how labels for years 2020, 2021, 2022 are all over the place.

```
spending_plot_data %>%
  ggplot(aes(x = year, y = n, fill = agency, label = agency)) +
  geom_col(position="fill", show.legend = T) +
  scale_fill_manual(
    values = c("#5E5E5E", "#EF3B2C", "#2CA25F", "#006837", "#F7DC6F", "#00FFFF", "#FFC0CB"
  theme_minimal() +
  theme(legend.position = "none") +
  labs(y = "Millions Spent", fill = "Department") +
  geom_label( size = 3, position = position_fill(vjust = 0.5), fill = "white", alpha = 0.5
```




If we specify group aesthetic everything goes back to its place!

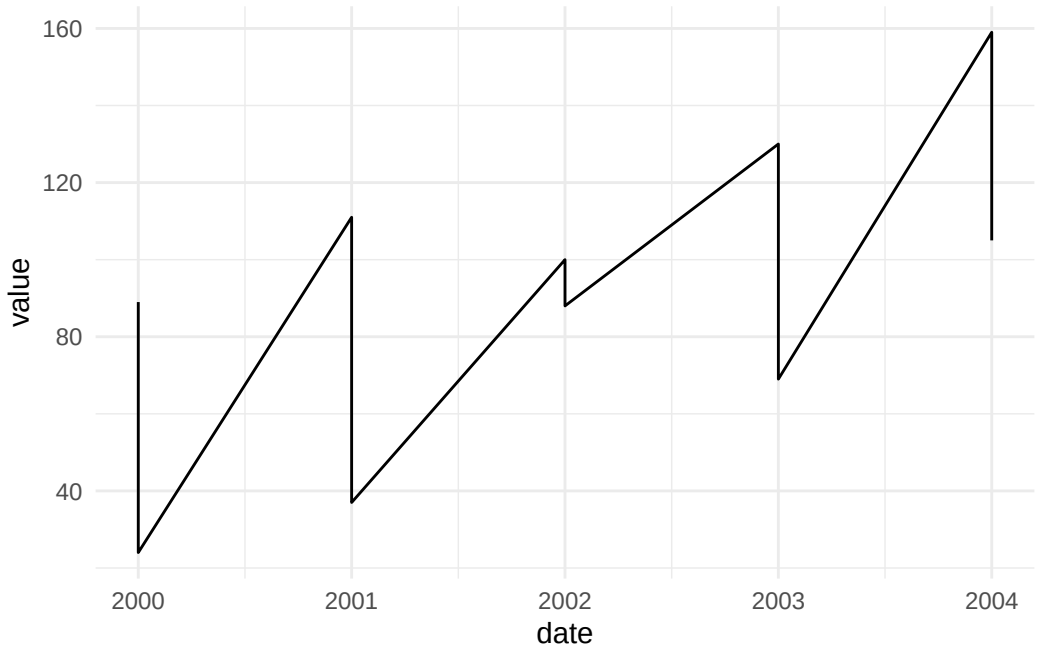
```
spending_plot_data %>%
  ggplot(aes(x = year, y = n, fill = agency, label = agency)) +
  geom_col(position="fill", show.legend = T) +
  scale_fill_manual(
    values = c("#5E5E5E", "#EF3B2C", "#2CA25F", "#006837", "#F7DC6F", "#00FFFF", "#FFC0CB"
  theme_minimal() +
  theme(legend.position = "none") +
  labs(y = "Millions Spent", fill = "Department") +
  geom_label( aes(group = agency),size = 3, position = position_fill(vjust = 0.5), fill =
```



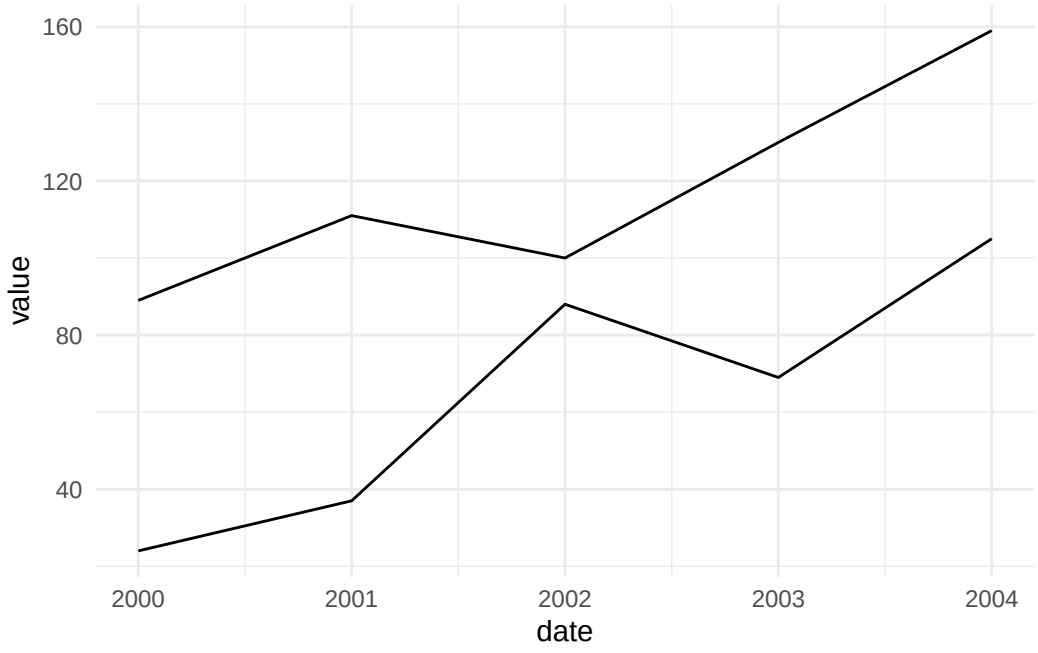

This error is very common in line charts too.

```
group_line_data <- tibble(
 measure = c(rep("hot",5),rep("cool",5)),
 date = rep(seq(2000,2004, by = 1),2),
 value = c(89,111,100,130,159,24,37,88,69,105)
)
```

```
group_line_data %>% ggplot (aes(x=date, y= value)) + geom_line() + theme (legend.position
```




```
group_line_data %>% ggplot (aes(x=date, y= value, group = measure)) + geom_line () + theme
```



# 8 Color

Color is a crucial component of data visualization. It has the power to evoke emotions, highlight patterns, and communicate information that might be difficult to convey through other means. Effective use of color in data visualization can enhance the viewer's understanding and engagement with the data, while poor use of color can obscure important information and create confusion. We will learn how to easily create different color schemes and palettes.

Color is a powerful tool for data visualization because it can help convey information quickly and effectively. By using different colors to represent different data points or categories, we can create visual patterns that are easy to interpret and remember. Color can also be used to highlight specific data points or draw attention to important trends or outliers in the data. Additionally, color can make data visualizations more engaging and appealing, which can help hold viewers' attention and make them more likely to understand and remember the information being presented.

Color evokes emotional responses that can influence people's emotions and perception. Colors being out different emotions for example in the US red is associated with danger or passion, blue with calmness or sadness, and green with nature or health. Cultures have attach different meaning to colors, so tailor color to your audience to elicit desired responses. Color is believed to be so powerful that locker room at Iowa's Kinnick Stadium has been painted pink since 1979 with an idea to lower opponent's testosterone levels.

### 8.0.1 Highlight Important Point

```
gapminder1997eu <- gapminder %>% filter((year == 1997) & (continent == "Europe"))
continents <- gapminder %>% select(-country) %>% mutate(gdp = gdpPercap * pop) %>% group_b
malay_miracle <- gapminder %>% filter(country %in% c("Malaysia", "Vietnam", "Indonesia", "
asian_tigers <- gapminder %>% filter(country %in% c("Hong Kong, China", "Taiwan", "Singapo
  mutate(country = recode(country,  "Hong Kong, China" = "Hong Kong",  "Korea, Rep." = "So
```

We use color to emphasize certain data and give context. Below are graphs comparing GDP of European Countries in 1997. First Graph gives each country its own color, creating a "explosion at a candy factory". Second graph is an improvement as it removes the distraction, emphasizes the country of the interest by highlighting "Greece" and greying out the rest. The "Red" color signals that Greece might be doing not so well.

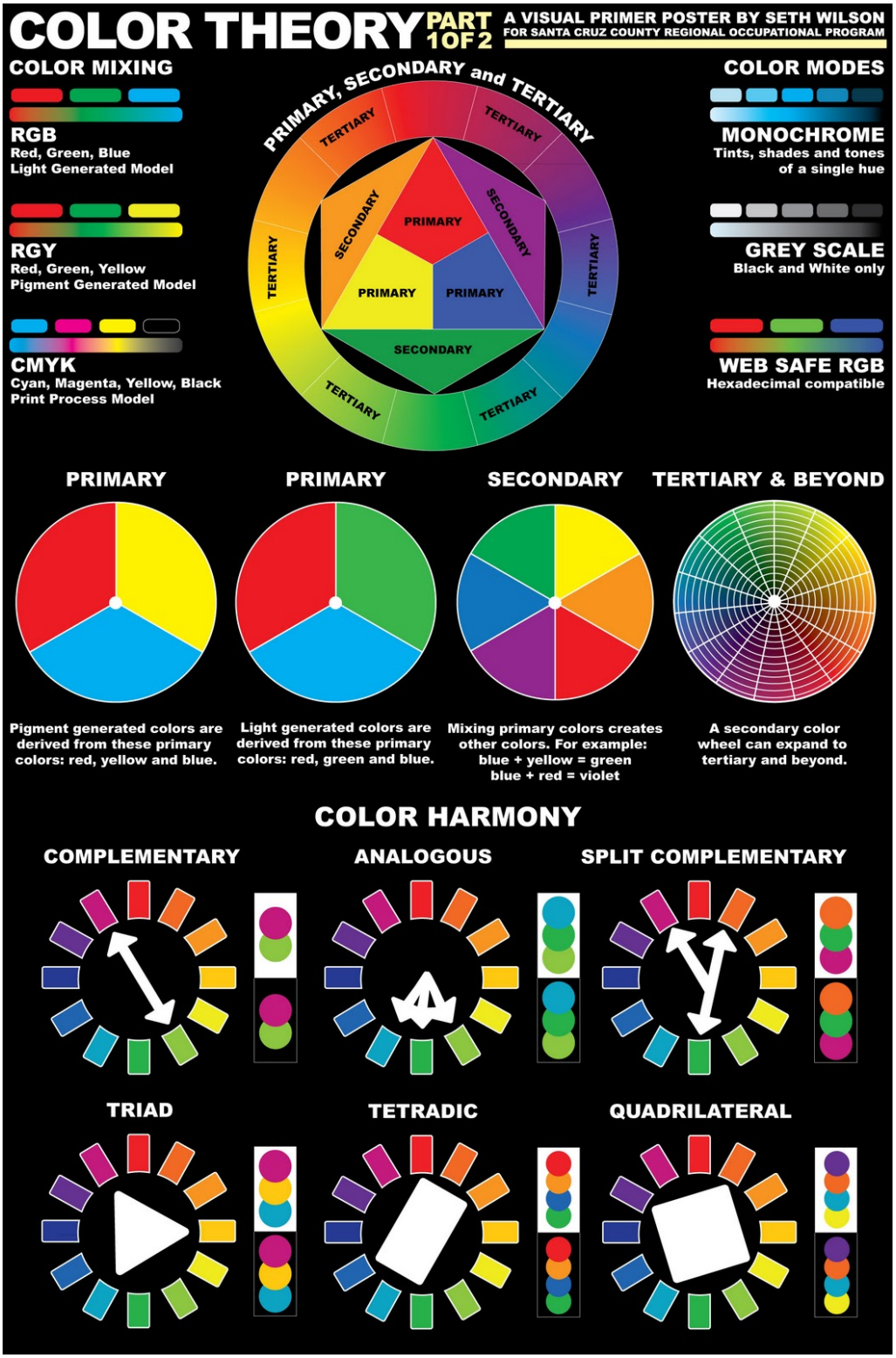
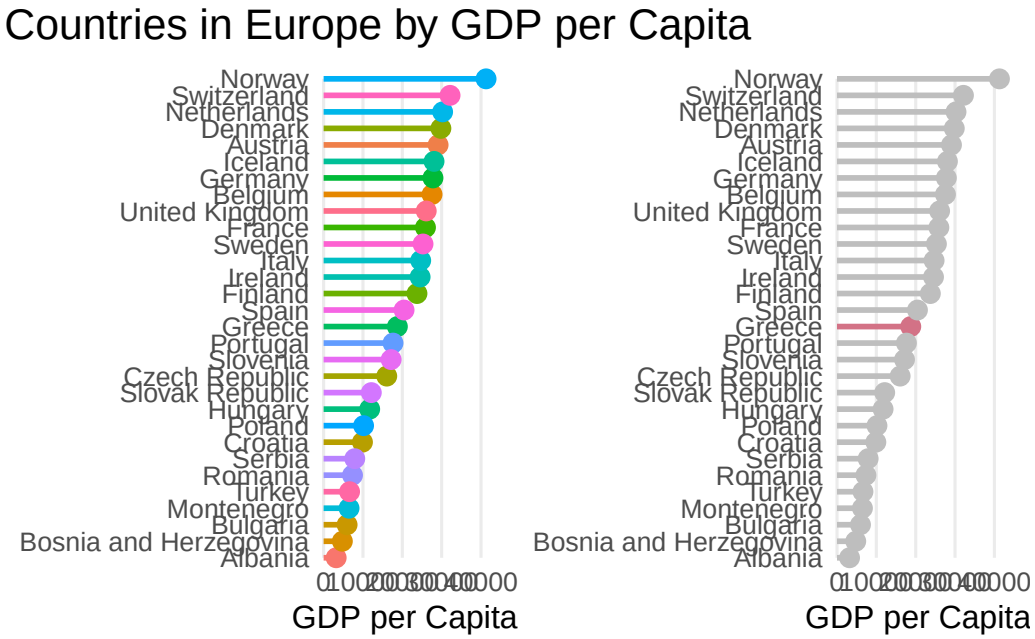



Figure 8.1: src: http://inkfumes.blogspot.com/2011/10/poster-designs-color-design-typography.html

```r
p1 <- gapminder1997eu %>%
  mutate(is.Greece = country == "Greece") %>%
  ggplot(aes(y = fct_reorder(country, gdpPercap), x = gdpPercap)) +
  geom_segment(aes(yend = country, xend=0, color = country), size = 1, show.legend = FALSE
  geom_point(aes(color = country), show.legend = FALSE, size = 3) +
  theme_minimal(base_size = 12) +
  theme(panel.grid.major.y = element_blank(),
        panel.grid.minor = element_blank()) +
 # ggplot2::scale_color_manual(values = c("#7286D3","#D37286")) +
  labs(x = "GDP per Capita", y = element_blank()) +
  coord_cartesian(expand = FALSE, clip = 'off')

p2 <- gapminder1997eu %>%
  mutate(is.Greece = country == "Greece") %>%
  ggplot(aes(y = fct_reorder(country, gdpPercap), x = gdpPercap)) +
  geom_segment(aes(yend = country, xend=0, color = is.Greece), size = 1, show.legend = FAL
  geom_point(aes(color = is.Greece), show.legend = FALSE, size = 3) +
  theme_minimal(base_size = 12) +
  theme(panel.grid.major.y = element_blank(),
        panel.grid.minor = element_blank()) +
  ggplot2::scale_color_manual(values = c("grey","#D37286")) +
  labs(x = "GDP per Capita", y = element_blank()) +
  coord_cartesian(expand = FALSE, clip = 'off')

p1 + p2 + plot_annotation(title = "Countries in Europe by GDP per Capita",theme = theme(pl
```



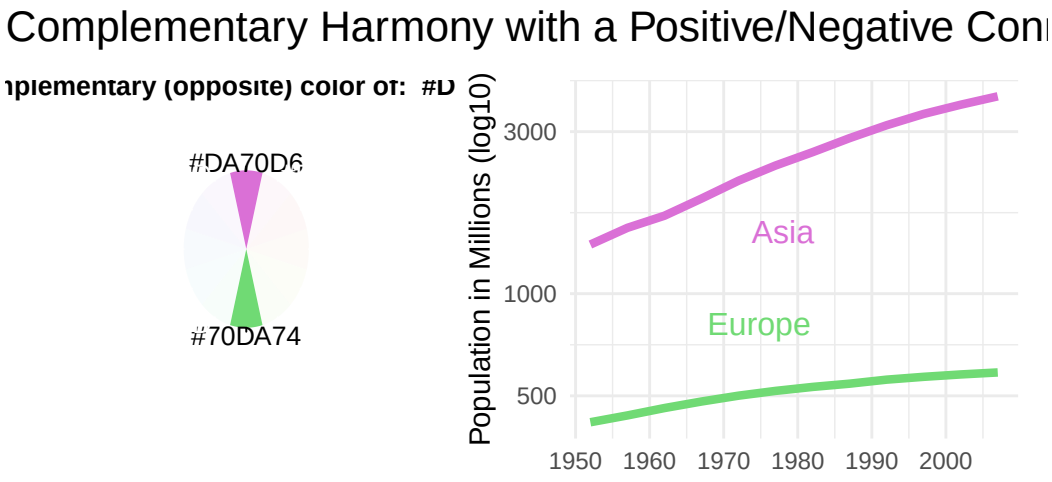

### 8.0.2 Comparing Two Things

#### 8.0.2.1 Complementary Harmony with a Positive/Negative Connotation

Complementary Harmony refers to the use of colors that are opposite each other on the color wheel to create a strong contrast. This creates a positive/negative connotation that is good for showcasing differences. Colors near each other on the wheel can also work well together, but opposite colors provide the strongest support for a key color. The example below shows the comparison between population of Asia and Europe. The use of bright purple emphasizes the outstanding population growth of Asia, while the green color highlights the slower population growth in Europe.

```
p3 <- continents %>%
  filter(continent %in% c("Asia","Europe")) %>%
  ggplot(aes(x=year, y = total_pop_mil, color = continent)) +
  geom_line(linewidth = 1.5) +
  theme_minimal() +
  theme(legend.position = "none") +
  scale_y_log10() +
  scale_color_manual(values = c("#DA70D6", "#70DA74")) +
  geom_dl(aes(label = continent), method = "smart.grid") +
  labs(x = element_blank(), y = "Population in Millions (log10)")
```

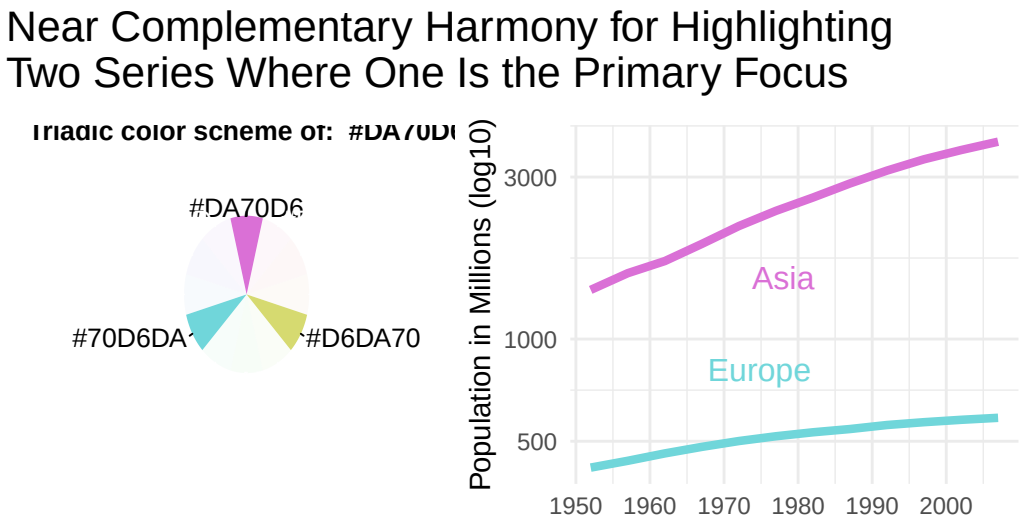
```
complementary + p3 + plot_annotation(title = "Complementary Harmony with a Positive/Negati
```




#### 8.0.2.2 Near Complementary Harmony for Highlighting Two Series Where One Is the Primary Focus

Near Complementary Harmony is a color scheme that creates good contrast without using polar opposite colors. It involves selecting a color that is 33% around the color wheel from the key color instead of the full 50%. This works well for highlighting two series where one is the primary focus. It is best to use warm colors for the key color and cool colors for the complementary colors. If necessary, the complementary colors can be muted by decreasing their saturation or altering their lightness to reduce the contrast with the background. The example below highlights the importance of population growth in Asia, with Europe being presented neutrally as a point of comparison rather than as a slow-growing region.

```
p4 <- continents %>%
  filter(continent %in% c("Asia","Europe")) %>%
  ggplot(aes(x=year, y = total_pop_mil, color = continent)) +
  geom_line(size = 1.5) +
  theme_minimal() +
  theme(legend.position = "none") +
  scale_y_log10() +
  scale_color_manual(values = c("#DA70D6", "#70D6DA"))  +
  geom_dl(aes(label = continent), method = "smart.grid") +
  labs(x = element_blank(), y = "Population in Millions (log10)")

triadic + p4 + plot_annotation(title = "Near Complementary Harmony for Highlighting \nTwo
```

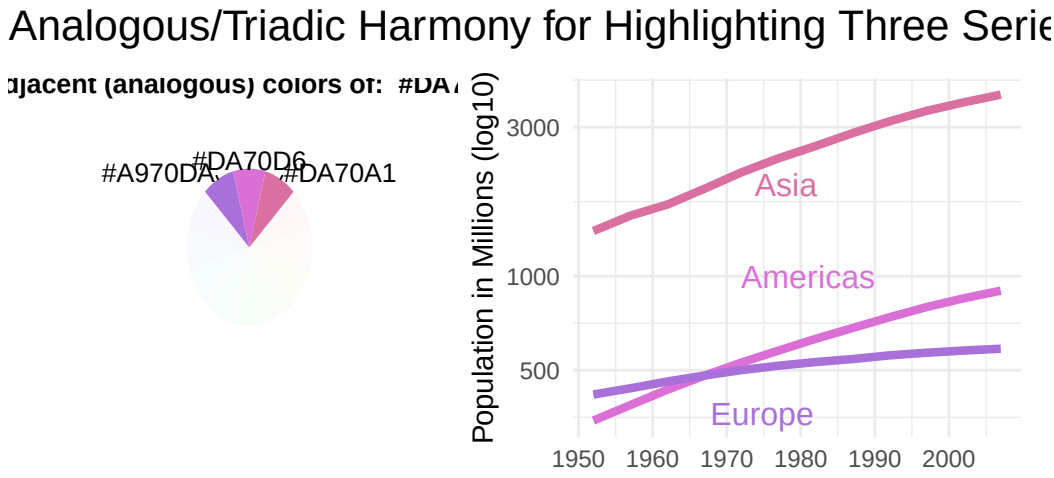



### 8.0.3 Color Palettes for Comparing Three Things

#### 8.0.3.1 Analogous/Triadic Harmony for Highlighting Three Series

Analogous harmony involves using neighboring colors to the key color for simple distinctions among categories. In contrast, triadic harmony uses the key color and two complementary colors evenly spaced around the color wheel for greater contrast, but may lose the emphasis on the key color. The example below displays the population of three countries without any specific emphasis.

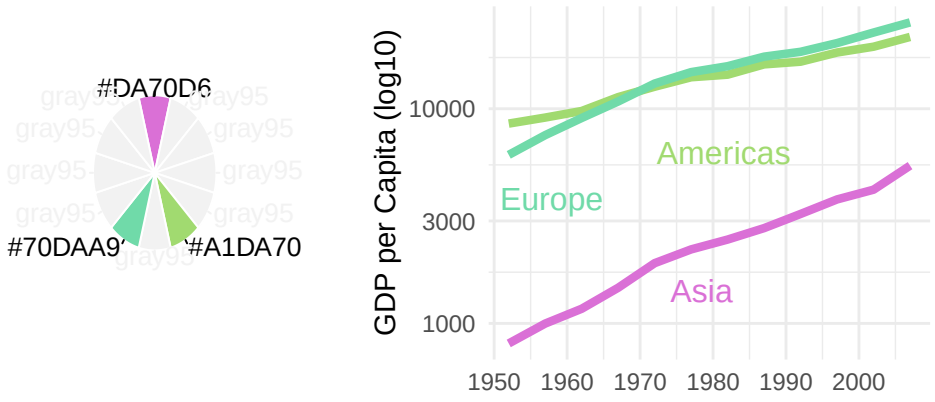
```
p5 <- continents %>%
  filter(continent %in% c("Asia","Europe","Americas")) %>%
  ggplot(aes(x=year, y = total_pop_mil, color = continent)) +
  geom_line(size = 1.5) +
  theme_minimal() +
  theme(legend.position = "none") +
  scale_y_log10() +
  scale_color_manual(values = c("#DA70D6", "#DA70A1", "#A970DA"))  +
  geom_dl(aes(label = continent), method = "smart.grid") +
  labs(x = element_blank(), y = "Population in Millions (log10)")

analogous + p5 + plot_annotation(title = "Analogous/Triadic Harmony for Highlighting Three
```




#### 8.0.3.2 Highlighting One Series Against Two Related Series

Near Complementary Harmony is a color scheme that creates good contrast without using polar opposite colors. It involves selecting a color that is 33% around the color wheel from the key color instead of the full 50%. This works well for highlighting two series where one is the primary focus. It is best to use warm colors for the key color and cool colors for the complementary colors. If necessary, the complementary colors can be muted by decreasing their saturation or altering their lightness to reduce the contrast with the background. The example below highlights the GDPs of 3 countries with emphasis on Asia through the use of the purple color. Europe and the Americas are depicted with similar shades of green, indicating their lesser significance for the narrative.

```
p6 <- continents %>%
  filter(continent %in% c("Asia","Europe","Americas")) %>%
  ggplot(aes(x=year, y = total_gdppc, color = continent)) +
  geom_line(size = 1.5) +
  theme_minimal() +
  theme(legend.position = "none") +
  scale_y_log10() +
  scale_color_manual(values = c( "#A1DA70", "#DA70D6", "#70DAA9")) +
  geom_dl(aes(label = continent), method = "smart.grid") +
  labs(x = element_blank(), y = "GDP per Capita (log10)")

complementary_3 + p6 + plot_annotation(title = "Highlighting One Series Against Two Relate
```

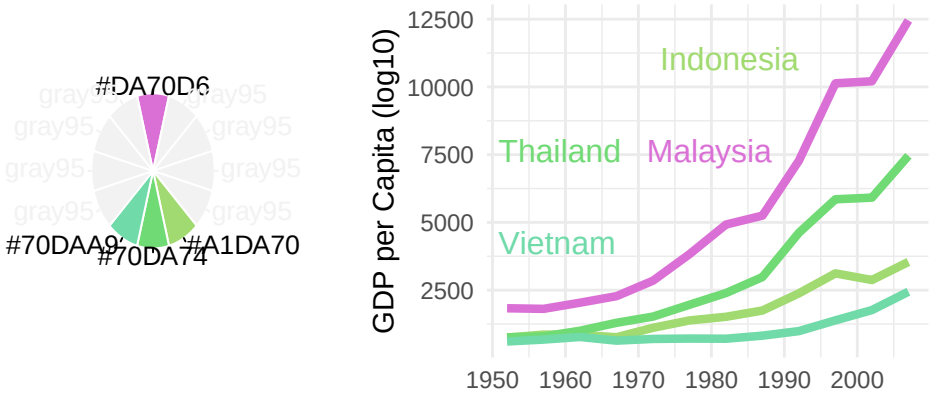



### 8.0.4 Color Palettes for Comparing Four Things

#### 8.0.4.1 Analogous Complementary for One Main Series and Its Three Secondary

Analogous Complementary is a color scheme that uses four colors, where the key color and its complementary color are combined with two colors that are one step away from the complementary color. This scheme still allows for analogous harmony while creating a quartet of colors that can be used for one main series and its three components. The similarities between the three complementary colors make the key color stand out. The example below shows Malaysian Economic Miracle in comparison to Malaysia's three neighbors.

```
p7 <- malay_miracle %>% ggplot(aes(x = year, y = gdpPercap, color = country)) +
  geom_line(linewidth = 1.5) +
  theme_minimal() +
  #scale_y_log10() +
  theme(legend.position = "none") +
  scale_color_manual(values = c("#A1DA70", "#DA70D6", "#70DA74", "#70DAA9")) +
  geom_dl(aes(label = country), method = "smart.grid") +
  labs(x = element_blank(), y = "GDP per Capita (log10)")

complementary_4 + p7 + plot_annotation(title = "Analogous Complementary for One Main Serie
```

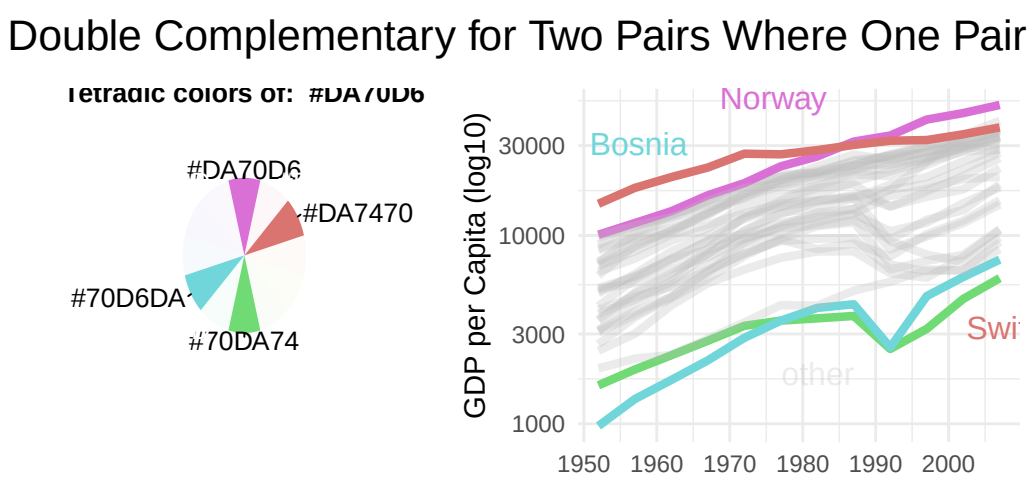



#### 8.0.4.2 Double Complementary for Two Pairs Where One Pair Is Dominant

Double Complementary Harmony is a color scheme suitable for four different data series that are divided into two groups of two series. The scheme involves selecting the key color and one of its two adjacent colors on the color wheel. Then, choose the complements of both the key color and its adjacent color to serve as their respective partners. It is recommended that the key color and its adjacent color be warmer colors, while the complementary colors should be cooler colors. This scheme works well for highlighting two pairs where one pair is dominant. Below is an example that compares the GDP of the two top and two bottom countries in 1952. Switzerland and Norway are assigned purple-ish colors, putting them in one group, while Bosnia and Albania are assigned green-blue colors to differentiate them.

```
p8 <- gapminder %>% filter((continent == "Europe")) %>% mutate(bot_top = case_when(
  country %in% c("Albania") ~ "Albania",
  country %in% c("Bosnia and Herzegovina") ~ "Bosnia",
  country %in% c("Switzerland") ~ "Switzerland",
  country %in% c("Norway") ~ "Norway",
  T ~ "other"
)) %>% ggplot(aes(x = year, y = gdpPercap, color = bot_top, group = country, alpha = bot_t
  geom_line(linewidth = 1.5) +
  theme_minimal() +
  theme(legend.position = "none", legend.title = element_blank()) +
  scale_y_log10() +
  geom_dl(aes(label = bot_top), method = "smart.grid") +
  scale_color_manual(values = c("#70DA74", "#70D6DA", "#DA70D6", "grey", "#DA7470")) +
  scale_alpha_manual(values = c(1,1,1,0.3,1)) +
```

```
  labs(#title = "Growth of the bottom 2 and \ntop 2 countries by GDP in 1952",
       x = element_blank(), y = "GDP per Capita (log10)")


tetradic + p8 + plot_annotation(title = "Double Complementary for Two Pairs Where One Pair
```

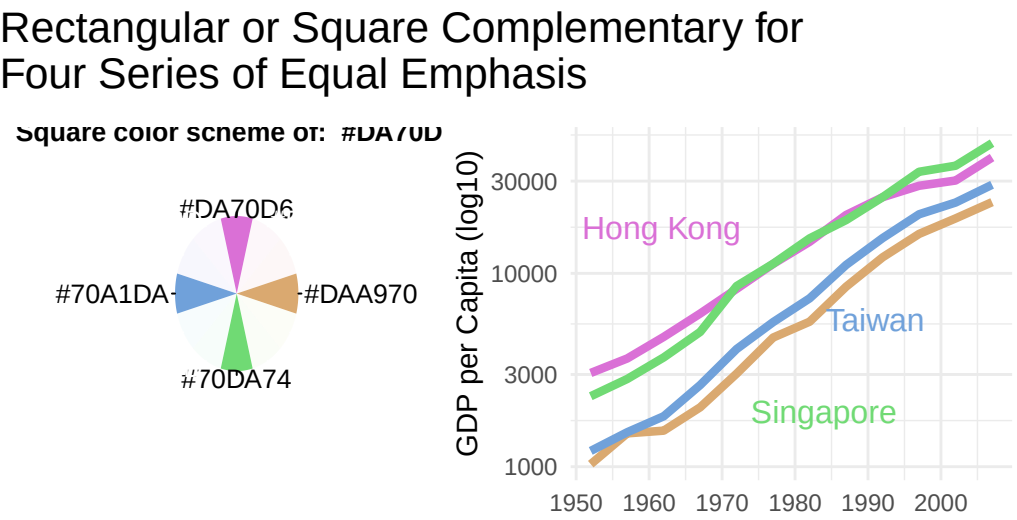



#### 8.0.4.3 Rectangular or Square Complementary for Four Series of Equal Emphasis

Rectangular or Square Complementary is a color scheme suitable for four series of equal emphasis where the objective is to use colors to make categorical distinctions. This scheme involves selecting the key color and its complementary color, but unlike the double complementary scheme, two additional colors are added to create a rectangle or square on the color wheel. The resulting colors create a clear distinction between the four series. This scheme is similar to double complementary but works better when all four series are of equal importance. The example below displays the Four Asian Tigers, a group of four fast-developing economies in East Asia that experienced high growth rates and rapid industrialization from the 1960s to 1990s. Hong Kong, Singapore, South Korea, and Taiwan are all equally important with the emphasis on the dynamic of their economies.

```
p9 <- asian_tigers %>% ggplot(aes(x = year, y = gdpPercap, color = country)) +
  geom_line(linewidth = 1.5) +
  theme_minimal() +
  theme(legend.position = "none") +
  scale_y_log10() +
  scale_color_manual(values = c("#DA70D6", "#DAA970", "#70DA74", "#70A1DA")) +
  geom_dl(aes(label = country), method = "smart.grid") +
  labs(x = element_blank(), y = "GDP per Capita (log10)")
```

```
square + p9 + plot_annotation(title = "Rectangular or Square Complementary for \nFour Seri
```




### 8.0.5 Sequential and Divergent

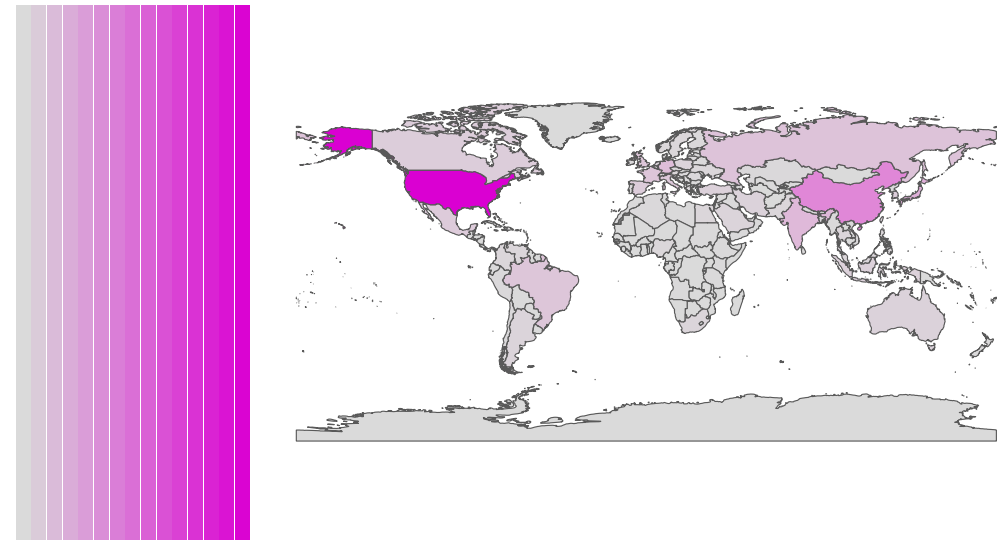
Sequential colors are a gradient of colors from light to dark, assigned to numeric values, based on hue or lightness. The colors depend on the background, with lower values assigned lighter colors, and higher values assigned darker colors. A single hue or a sequence of hues can be used.

Let's use our beloved purple to GDP of different countries.

#### 8.0.5.1 Sequential

```
library("sf")
library("rnaturalearth")
library("rnaturalearthdata")
world <- ne_countries(scale = "medium", returnclass = "sf") %>% mutate(gdppc = gdp_md_est)
```

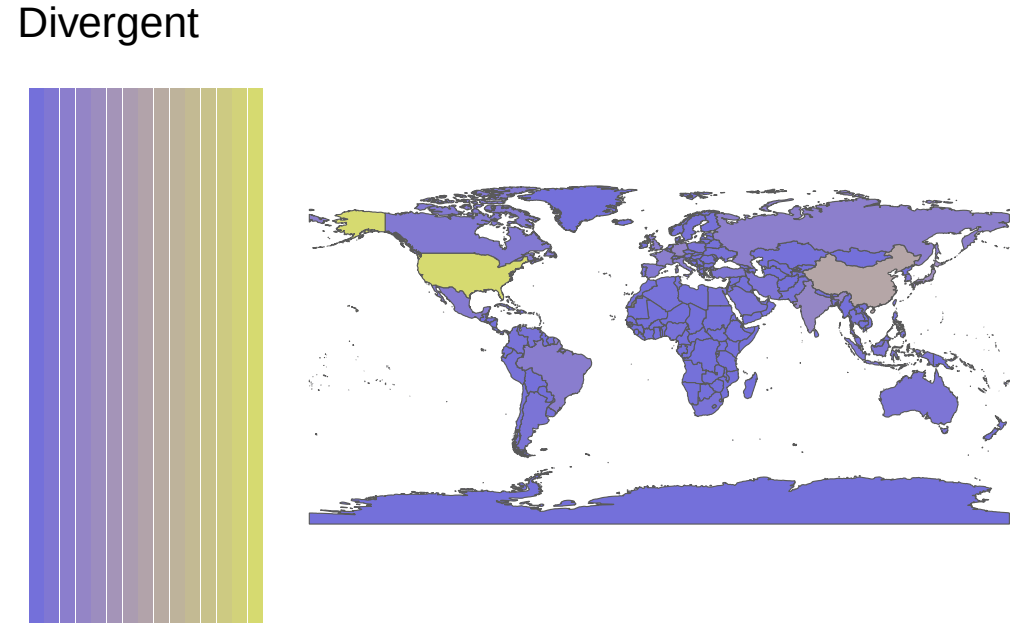
```
(sequential | p10) +
  patchwork::plot_layout(widths = c(1,3)) +
  plot_annotation(title = "Sequential",
                  theme = theme(plot.title = element_text(size = 16)))
```




#### 8.0.5.2 Divergent

Diverging color schemes are used when the numeric variables have a meaningful central value such as zero. They combine two sequential palettes with a shared endpoint that rests on the central value, with positive values assigned colors on one side and negative values on the other side. The central value should have a light color so that darker colors can indicate more distance from the center. It is important to keep the color scheme simple to avoid diluting the meaning and confusing the audience. Proper use of colors can reduce the cognitive load and help people understand complex information more easily.

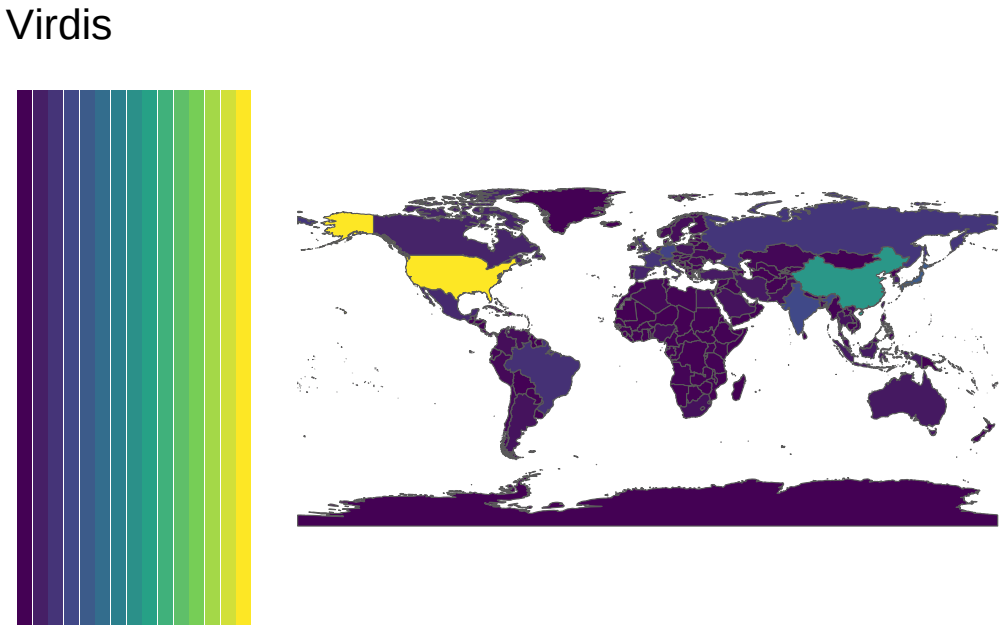
```
(divergent | p11) +
  patchwork::plot_layout(widths = c(1,3)) +
  plot_annotation(title = "Divergent",
                  theme = theme(plot.title = element_text(size = 16)))
```




### 8.0.6 Prebuilt

Prebuilt color scales like "Viridis" are designed with perceptual uniformity in mind, which makes them visually appealing and easy to interpret. They provide a consistent and standardized color scheme, eliminating the need for custom design and testing. In addition, prebuilt color scales can help people with color blindness to better interpret data visualizations, as they use colors with consistent visual contrast. Using prebuilt color scales can help ensure that data visualizations are accessible to the widest possible audience.

```
(viridis | p12) +
  patchwork::plot_layout(widths = c(1,3)) +
  plot_annotation(title = "Virdis",
                  theme = theme(plot.title = element_text(size = 16)))
```




### 8.0.7 Color Systems

There are several popular color systems that are commonly used in digital design and data visualization. The sRGB color system is the standard color space used for displaying images and graphics on digital displays. It is a device-dependent color space that is designed to provide consistent color reproduction across a wide range of devices. The HCV color system is based on hue, chroma, and value, and is used to create visually distinct color palettes for use in data visualization. The HSL color system is based on hue, saturation, and lightness, and is often used to create color palettes for web design and user interfaces. The LAB color system is a device-independent color space that is designed to accurately represent colors across different devices and environments. It is often used in professional printing and color management applications. Each of these color systems has its own strengths and weaknesses, and the choice of which one to use depends on the specific needs of the project.

To see how these spaces look, check out this amazing video!

#### 8.0.7.1 HSL

The HSL color system describes colors using three parameters: hue, saturation, and lightness. Hue is represented by a value from 0 to 360 degrees on the color wheel, and determines the basic color of the pixel. Saturation represents the purity of the hue, or how much gray is mixed into the color. Saturation ranges from 0% (gray) to 100% (pure hue). Lightness, on the other hand, represents the amount of white or black mixed with the color, with 0% being black,

50% being the original color, and 100% being white. HSL is often used in graphic design and web development, as it allows for the easy selection of colors based on hue, saturation, and lightness. However, it has some limitations, such as not being perceptually uniform, meaning that changes in the numeric values of the parameters may not correspond to equal changes in the perceived color.

#### 8.0.7.2 HSV

The HSV color system describes colors using three parameters: hue, saturation, and value. Hue and saturation are the same as in HSL. Value represents the brightness of the pixel, with 0% being black and 100% being the brightest possible color. The HSV color system is often used in graphics and image editing software, as it allows for easy selection of colors. However, it is also not perceptually uniform.

#### 8.0.7.3 LAB

The LAB color system is a device-independent color space that is designed to accurately represent colors across different devices and environments. It consists of three parameters: L (lightness), a (the position between red/magenta and green), and b (the position between yellow and blue). The L parameter represents the brightness of the color, ranging from 0 (black) to 100 (white). The a and b parameters represent the color channels, with positive values representing colors in the red, and yellow directions, respectively, and negative values representing colors in green and blue direction. The LAB color space is used in professional printing and color management applications, as it allows for accurate color matching across different devices and environments. Additionally, more recent LAB color spaces (ex. OKLAB) are perceptually uniform, meaning that equal distances in LAB color space correspond to equal steps in perceived color difference.

#### 8.0.7.4 OKLAB

OKLAB is a color space designed to be more perceptually uniform than other color spaces like sRGB or LAB. It uses an opponent color model, where the color information is encoded L – perceived lightness a – how green/red the color is b – how blue/yellow the color is. This means that the OKLAB color space can accurately represent colors while maintaining perceptual uniformity. OKLAB was developed to address the limitations of other color spaces. The use of OKLAB is becoming increasingly popular in digital design and data visualization, as it can provide more accurate and consistent color representation. You can learn about OKLAB from the video folder.

### 8.0.8 Perceptual Uniformity

Humans are not machines, we wee see things with our eyes and process them with our brain. What might appear like a similar color to a machine for humans will not. As an example, below are two color wheels one is RGB (perceptually non-uniform) and the other is HCL (uniform). When the color spectra are viewed in Gray scale the uneven nature of RGB becomes apparent. A technical definition is that a perceptual uniform color space ensures that the difference between two colors (as perceived by the human eye) is proportional to the Euclidian distance within the given color space.

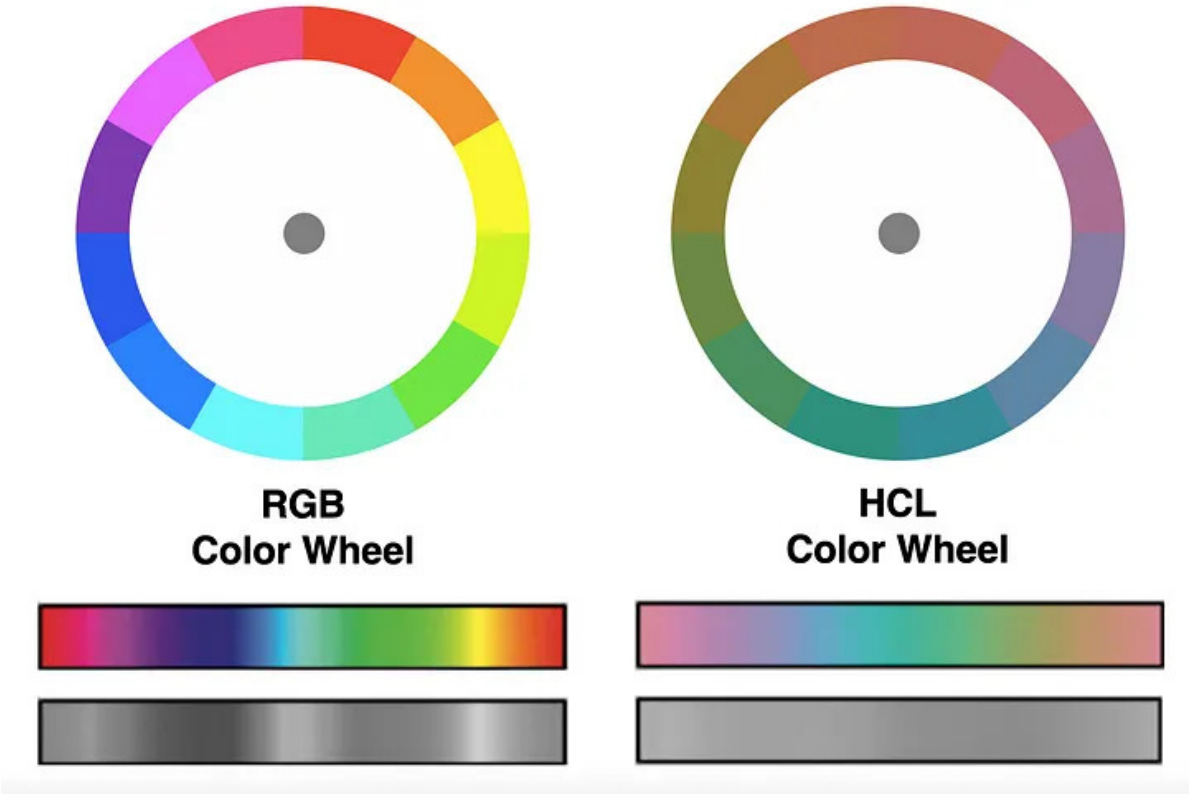


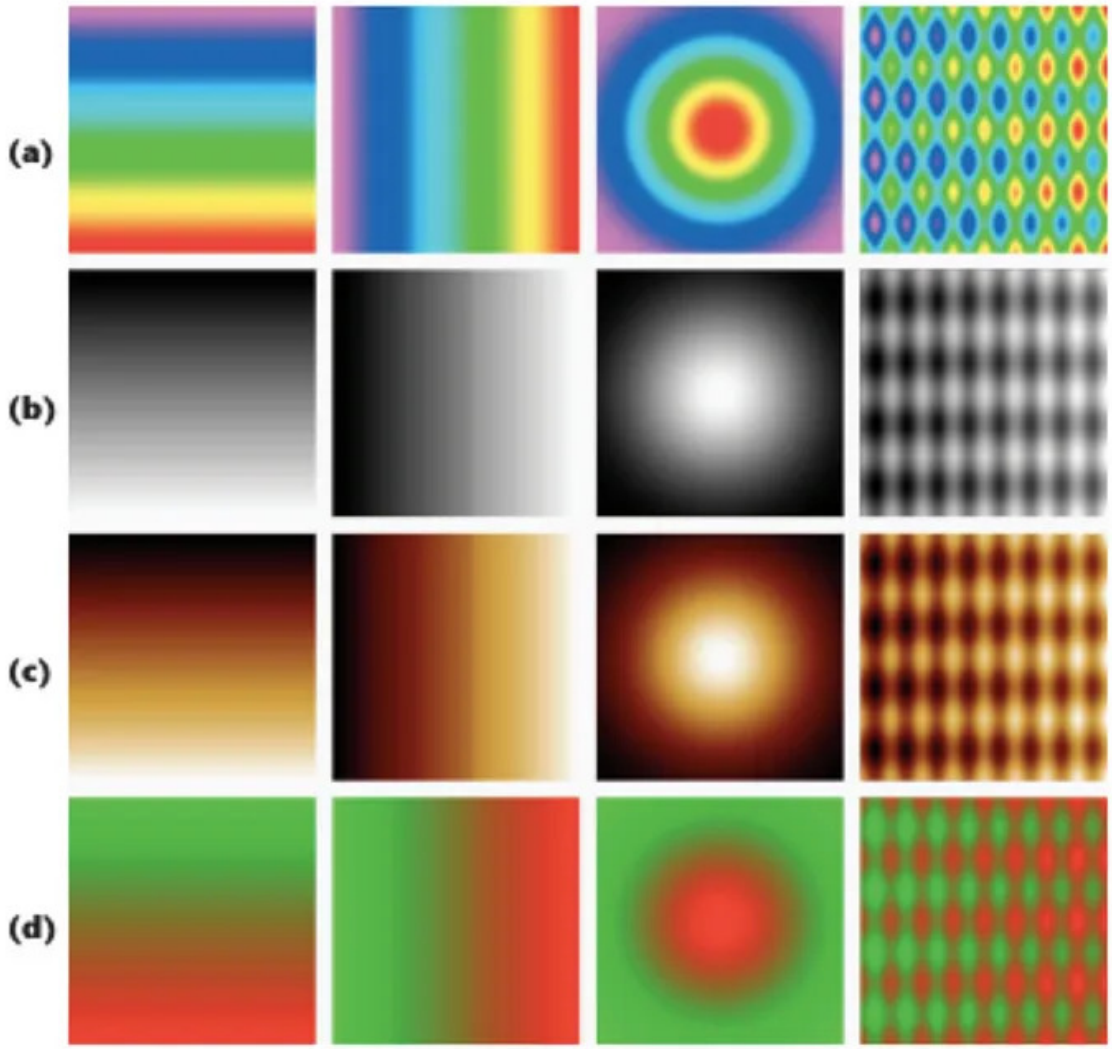

Figure 8.2: uniform_perception

#### 8.0.8.1 Warning Colormaps might Increase Risk of Death!

In 1990s data visualization specialists adopted Rainbow Color Map with the most famous variation being Jet default palette. Many researchers expressed concerns as the non-uniform nature of the palette introduced transitions that could be perceived incorrectly.

Rogowitz and Treinish raised concerns about the Rainbow Color Map in 1998 “Data Visualization: The End of the Rainbow”, and Borland and Taylor highlighted additional concerns

in a 2007 paper "Rainbow Color Map (Still) Considered Harmful". In 2011, Borkin and her team conducted user studies on the application of various color maps, including the Rainbow Color map, to medical visualization problems. Their findings in "Evaluation of Artery Visualizations of Heart Disease Diagnosis" showed that a perceptually uniform color map resulted in fewer diagnostic errors than the Rainbow Color map. So, diagnostic errors could be reduced by simply switching to a proper color palette. The problem of misuse of color still persists as outlined by Crameri, Shephard and Heron in "The misuse of colour in science communication" (2020) – a paper that I believe must be read by any scientist.




All of these issues are amplified once we consider that roughly 8% of all men and 0.5% of all women are colorblind. There are three main forms of red(protan), gren(deutan), and blue

(tritan) disorders, corresponding to color sensitive cones in our eyes. To check whether your visualization is colorblind friendly use Coblis.

To improve the readability of your colors vary their value and hue, but try not to include both red and green into your graphics as red-green colorblindness is the most common.


#### 8.0.8.2 So what should you use?

A simple and correct answer would be to use a scientific color map that you find pretty and use it as your default. If you need some help use this graph from "The misuse of colour in science communication".

If you want to pick colors yourself use HSL, because it is the most intuitive one and the easiest to make color palettes in. I would also recommend tinkering with OKHSL, which is a child of OKLAB and HSL producing a perceptually uniform HSL space. Try out both of them and see the difference here.

### 8.0.9 Where do I find color waves?

Adobe Color - Adobe Color let's you create color palettes using different color harmony rules

and color modes. You can also pick colors from your image, make gradients from images, and test for accessibility.

Paletton - Amazing tool for creating color palettes

Color Brewer - Color Brewer is a web-based tool that provides color schemes perceptual uniform for maps and data visualizations.

Color Thief - Color Thief let's you extract colors from your image to create nature-inspired palettes.

Viz Palette - Viz Palette can be used to check your color palettes before creating visualizations. It allows you to view color sets in example plots, simulate color deficiencies, and modify the colors of your palette.

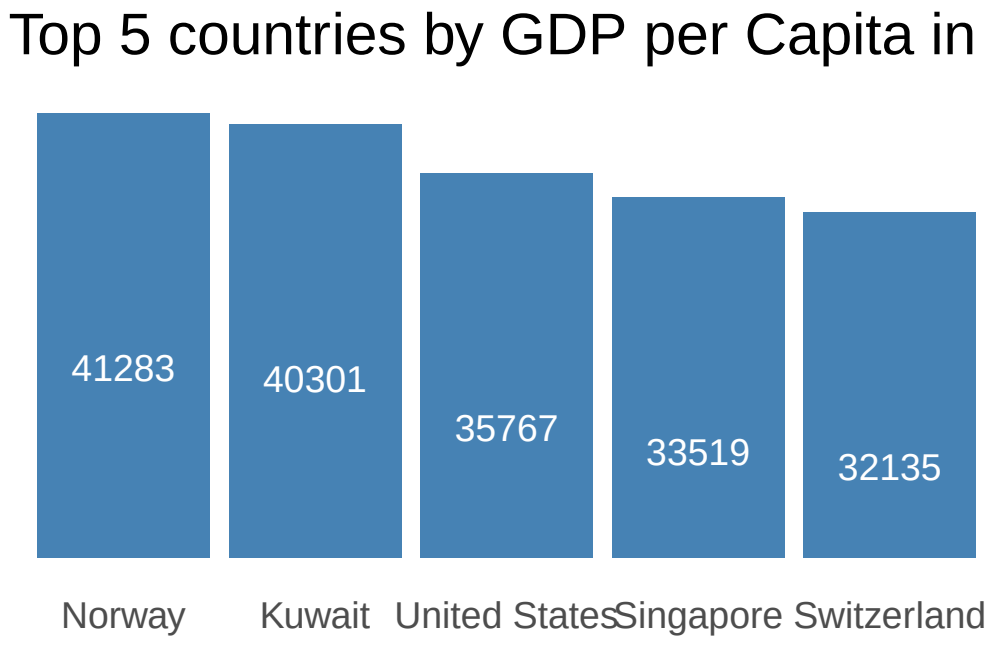
Scientific colour maps - A collection of uniform and readable color maps for scientific use.

# 9 A Graph for The Job

When working on graphs don't think "what chart should I use?", but "what am I trying to show?" In this section we will look at different types of chart, what they show and when to use them. We will not cover all the graphs, but you will definitely expand your kit!

Graphs for category comparison are a type of data visualization that are used to compare and contrast different categories or groups. The most common of them is bar chart! The one below shows the top 5 countries by GDP per Capita in 1997. You can easily see that Norway is first and Switzerland is 5th!

```
gapminder %>% filter(year == 1997) %>% slice_max(gdpPercap, n = 5) %>% ggplot(aes(x = fct_
geom_text(aes(label = round(gdpPercap,0)), vjust = 10, color = "white", size = 5) +
theme(panel.grid = element_blank(), axis.text.y = element_blank())
```

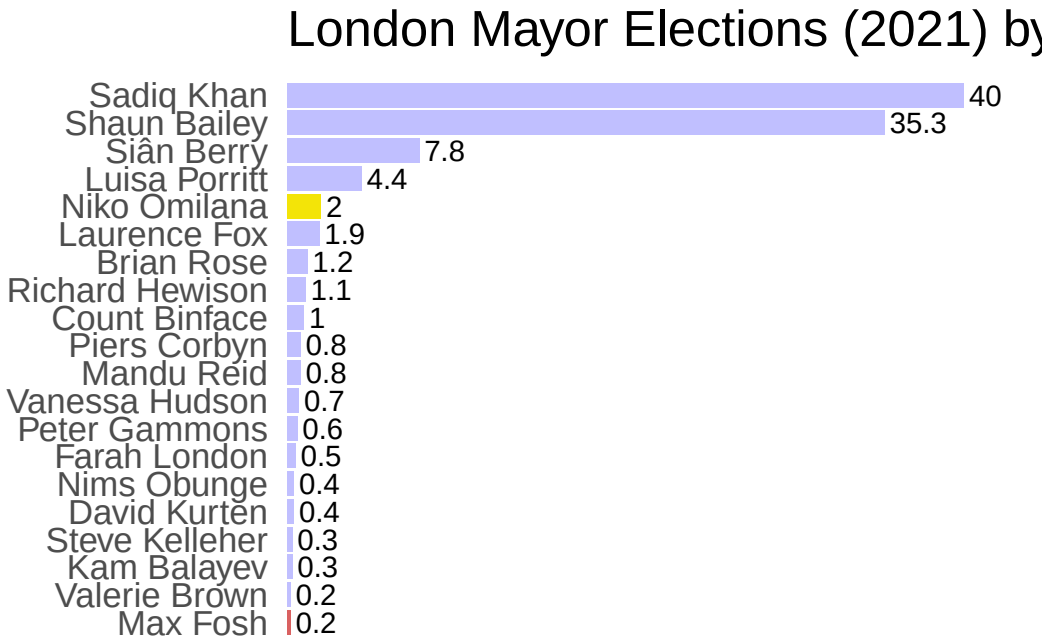



Vertical bar charts are great to provide a quick comparison for a small number of categories (less than 7). But if need to show ranking of more things, flip the axis of the bar chart! Additional bonus, horizontal bar charts are great if you have long names to display. Below are

the results of the 2021 London election. British YouTuber Niko Omilana finished 5th *for the memes*! Max Fosh, another YouTuber, also passed the cut off!

```
barh_data <- tibble(Candidate = c("Sadiq Khan", "Shaun Bailey", "Siân Berry",
"Luisa Porritt", "Niko Omilana", "Laurence Fox", "Brian Rose",
"Richard Hewison", "Count Binface", "Mandu Reid", "Piers Corbyn",
"Vanessa Hudson", "Peter Gammons", "Farah London", "David Kurten",
"Nims Obunge", "Steve Kelleher", "Kam Balayev", "Max Fosh", "Valerie Brown"
),
Percentage = c(40.0, 35.3, 7.8, 4.4, 2.0, 1.9,
1.2, 1.1, 1.0, 0.8, 0.8, 0.7, 0.6, 0.5, 0.4,
0.4, 0.3, 0.3, 0.2, 0.2)) %>% mutate(is.youtuber = case_when(
  Candidate == "Niko Omilana" ~ 1,
  Candidate == "Max Fosh" ~ 2,
  T ~ 0))
```

```
barh_data %>%
ggplot(aes(x = fct_reorder(Candidate, Percentage), y = Percentage, fill = as.factor(is.you
geom_col() +
theme_minimal(base_size = 16) +
coord_flip() +
labs(x = NULL, y = NULL, title = "London Mayor Elections (2021) by % of Votes") +
theme(panel.grid = element_blank(), legend.position = "none", plot.caption.position = "plo
scale_y_discrete(expand = c(0,0,0,3)) +
geom_text(aes(label = Percentage), nudge_y = 0.3, hjust = "left")  +
scale_fill_manual(values = c("#c0bfff","#f3e408","#D96161"))
```

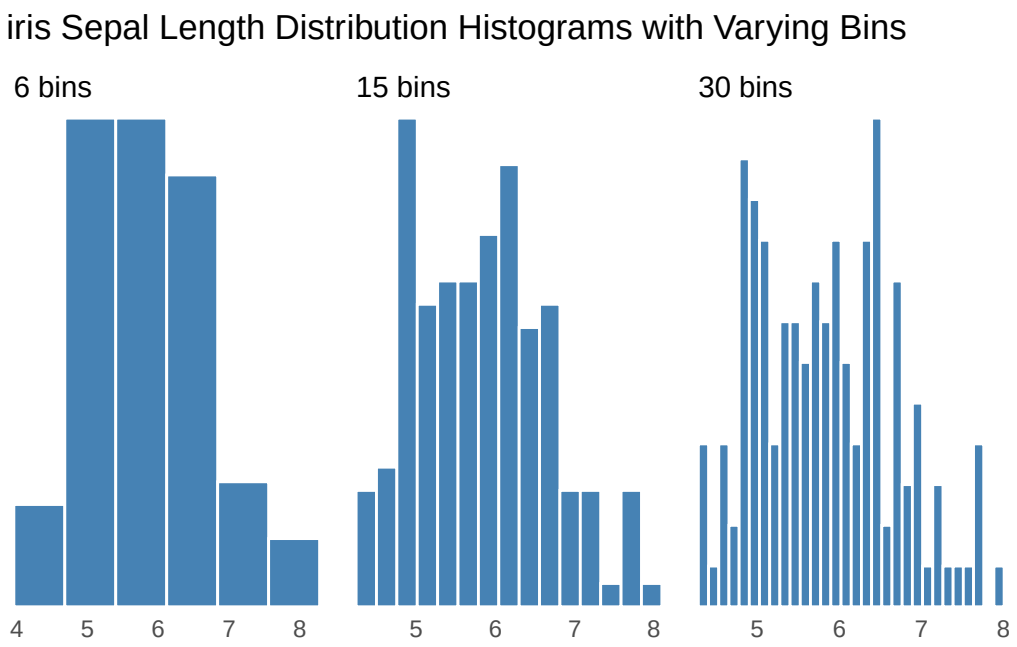



## 9.1 Distribution

### 9.1.1 Histogram

What if you want to show the distribution of the data? We can use a variation of a bar chart – histogram! Histograms show the distribution of continuous data by grouping it into bins and displaying the frequency or proportion of observations that fall into each bin. They are great if you want to show the shape of the distribution, but they are very sensitive to the bins you choose. Notice how the shape of the distribution changes for each number of bins. It is important to strike a balance between too few and too many. 6 bins makes our distribution look pretty normal while 30 bins make it all over the place. 15 bins seems about right it preserves the bimodal feature of the distribution, while keeping the picture legible.

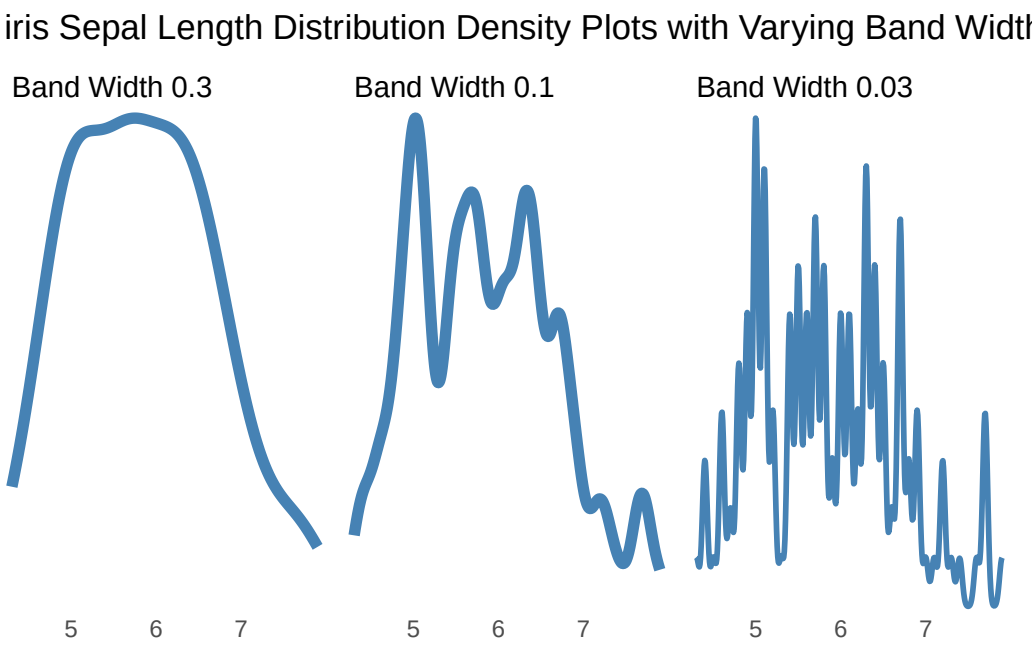
```
base <- iris %>% ggplot(aes(x = Sepal.Length))  + theme_minimal() + theme(panel.grid = ele
  coord_cartesian(expand = FALSE, clip = "off") + labs(y = NULL, x = NULL)
hist_1 <- base + geom_histogram(fill = "steelblue", color = "white", bins = 6) + labs(subt
hist_2 <- base + geom_histogram(fill = "steelblue", color = "white", bins = 15) + labs(sub
hist_3 <- base + geom_histogram(fill = "steelblue", color = "white", bins = 30) + labs(sub

(hist_1 + hist_2 + hist_3) + plot_annotation(title = "iris Sepal Length Distribution Histo
```




### 9.1.2 Density Plot

Unlike histograms, density plots use a continuous line to represent the data instead of bars. This smooth curve provides a more detailed and nuanced representation of the distribution of the data, allowing for easier detection of patterns and trends. The density plot constructs this line by placing many small normal distributions at each point in the data, which are then used to weigh all points within their respective range and draw a curve connecting them. The width of these curves is controlled by the bandwidth of the density plot, which determines how wide the curves span. A larger bandwidth will consider more points, resulting in a smoother curve, while a smaller bandwidth will lead to a jagged line.

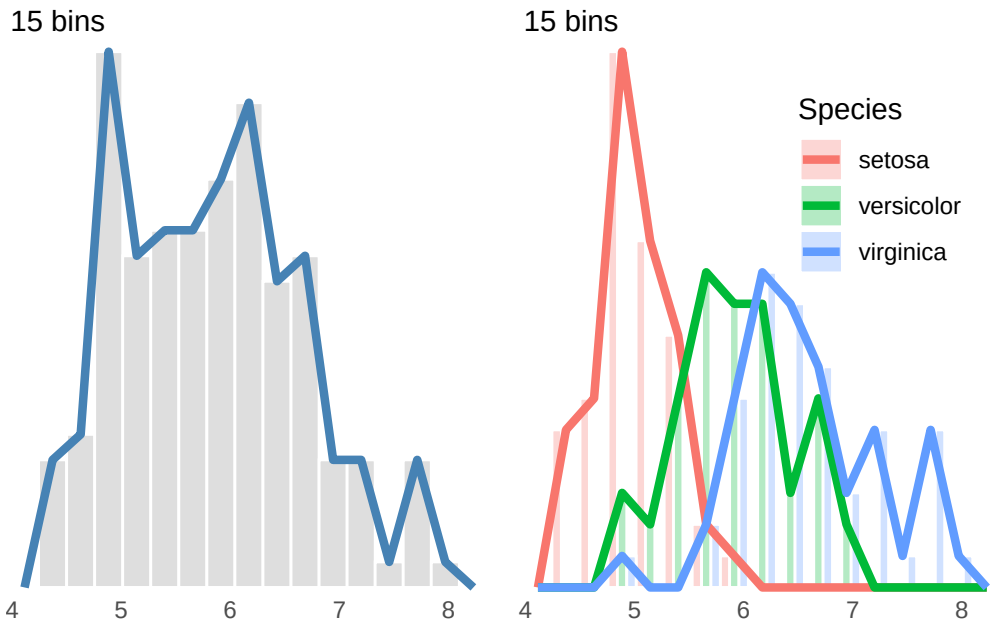
```
base <- iris %>% ggplot(aes(x = Sepal.Length))  + theme_minimal() + theme(panel.grid = ele
  coord_cartesian(expand = FALSE, clip = "off") + labs(y = NULL, x = NULL)
dens_1 <- base + geom_density(color = "steelblue", linewidth = 2, bw = 0.3) + labs(subtitl
dens_2 <- base + geom_density(color = "steelblue", linewidth = 2, bw = 0.1) + labs(subtitl
dens_3 <- base + geom_density(color = "steelblue", linewidth = 1, bw = 0.03) + labs(subtit

(dens_1 + dens_2 + dens_3) + plot_annotation(title = "iris Sepal Length Distribution Densi
```




### 9.1.3 Frequency Polygon

It is similar to a histogram, but instead of bars, it uses a continuous line to connect the points representing the frequencies. Frequency polygons are particularly useful when comparing two or more data sets on the same plot. Just like histogram it relies on the selection of bins.

```
freq_1 <- base + geom_histogram(fill = "grey", color = "white", bins = 15, alpha = 0.5) +

freq_2 <- iris %>% ggplot(aes(x = Sepal.Length))   +
  geom_histogram(aes(fill = Species), position = "dodge", color = "white", bins = 15, alph
  geom_freqpoly(aes(color = Species), bins = 15, linewidth = 1.5) + labs(subtitle = "15 bi
  theme_minimal() +
  theme(panel.grid = element_blank(), axis.text.y = element_blank(),
        legend.position = c(.95, .95),
    legend.justification = c("right", "top"),
    legend.box.just = "right",
    legend.margin = margin(6, 6, 6, 6)) +
  coord_cartesian(expand = FALSE, clip = "off") +
  labs(y = NULL, x = NULL)

freq_1 + freq_2
```




### 9.1.4 Box Plot

This section was inspired by https://www.cedricscherer.com/2021/06/06/visualizing-distributions-with-raincloud-plots-and-how-to-create-them-with-ggplot2/

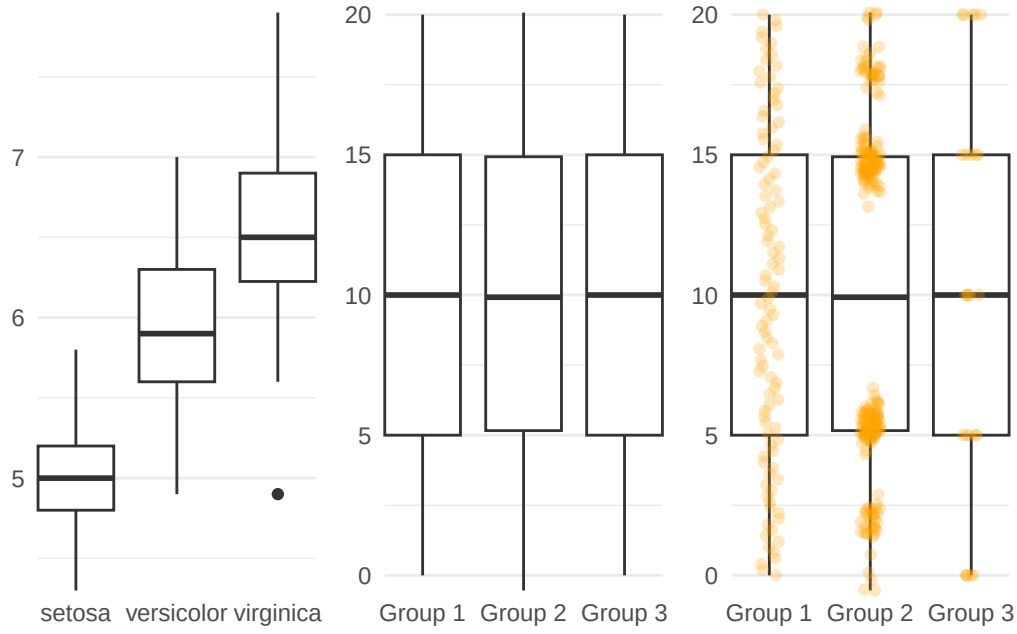
Boxplots provide a summary of the distribution of a dataset, show the median, the lower and upper quartiles, and the minimum and maximum values of a dataset. The box in the middle represents the interquartile range (IQR), which is the range of the middle 50% of the data. The line in the box represents the median, which is the midpoint of the data. The whiskers on the top and bottom extend to the minimum and maximum values, excluding outliers. It is incredible how much information boxplots contain! With just one plot, you can quickly identify outliers and gain a visual understanding of the distribution of the data.

In the context of the iris dataset, the boxplot of Sepal Length across different species provides a clear picture of the distribution of this variable. However, like real boxes, boxplots can also hide important information. To illustrate this point, we can use a dataset with the same summary statistics but different distributions. In the second graph, three identical boxplots are displayed. However, once we add data points to the plot, it becomes evident that the distributions are quite different.

```
set.seed(1337)

data_dist <- tibble(
  group = factor(c(rep("Group 1", 100), rep("Group 2", 250), rep("Group 3", 25))),
```

```
  value = c(seq(0, 20, length.out = 100),
            c(rep(0, 5), rnorm(30, 2, .1), rnorm(90, 5.4, .1), rnorm(90, 14.6, .1), rnorm(
            rep(seq(0, 20, length.out = 5), 5))
  ) %>%
  rowwise() %>%
  mutate(value = if_else(group == "Group 2", value + rnorm(1, 0, .4), value))
```

```
base2 <- iris %>% ggplot(aes(y = Sepal.Length, x = Species))  + theme_minimal() + theme(pa
  coord_cartesian(expand = FALSE, clip = "off") + labs(y = NULL, x = NULL)
box_1 <- base2 + geom_boxplot()

base_dist <- data_dist %>% ggplot(aes(y = value, x = group))  + theme_minimal() + theme(pa
  coord_cartesian(expand = FALSE, clip = "off") + labs(y = NULL, x = NULL)
box_2 <- base_dist + geom_boxplot()
box_3 <- base_dist + geom_boxplot() + geom_point(color = "orange", size = 1.5, alpha = 0.2

box_1 + box_2 + box_3
```

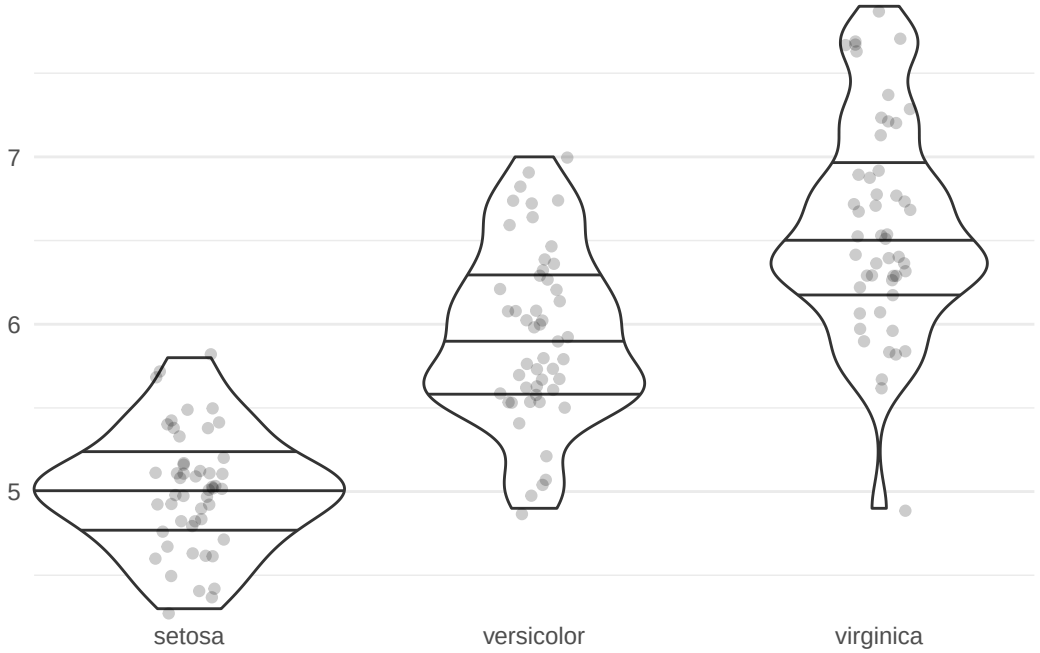



### 9.1.5 Violin Plot

One solution is to use violin plots. In its essence it is a vertical density plot. Look how much more we know about out data distribution of iris species! We can see the density distribution,

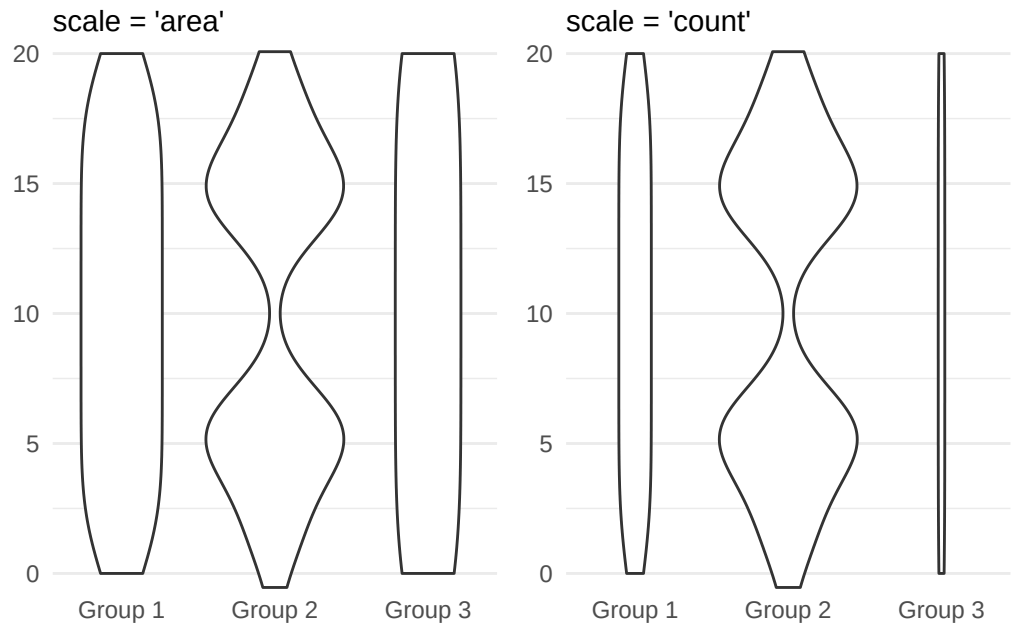
points and quantiles!

```
base2 + geom_violin(draw_quantiles = c(0.25, 0.5, 0.75), bw = 0.15) + geom_jitter(alpha =
```



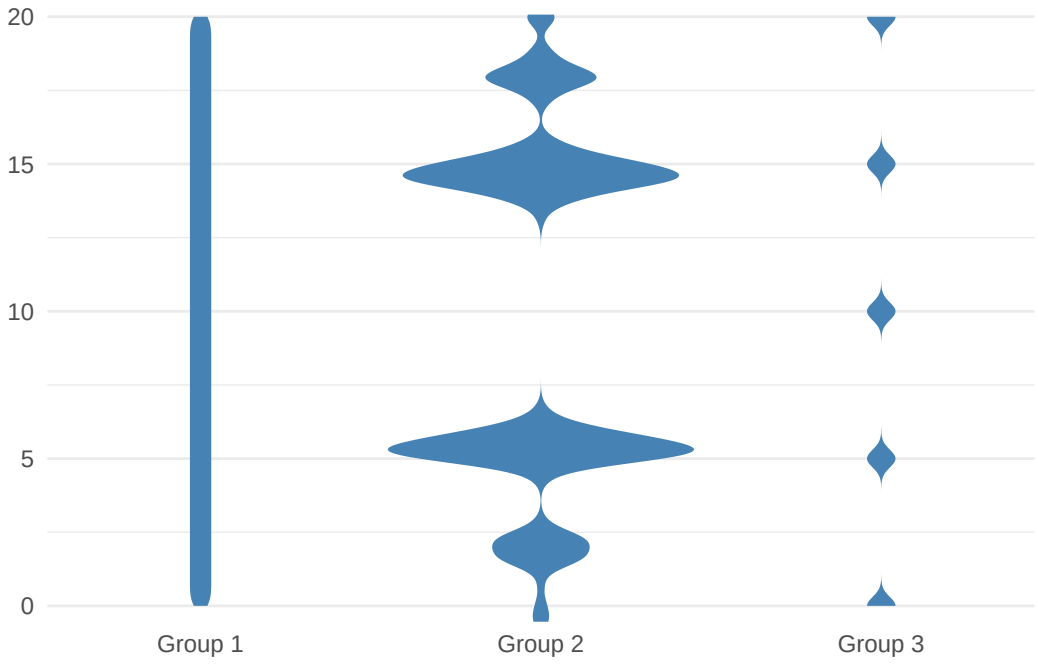

Going back to our new data set, notice how different the datasets look, clearly there are some patterns. Group 1, which has four times more observations, appears to be nearly identical to Group 3. This is because the default setting "scale ="area" " is a little misleading. We can fix that by changing it to "scale ="count" "

```
dist_1 <- base_dist + geom_violin(scale = "area") + labs(subtitle = "scale = 'area'")
dist_2 <- base_dist + geom_violin(scale = "count") + labs(subtitle = "scale = 'count'")
dist_1 + dist_2
```




Do you remember how band width is extremely important when making density plots? Setting an apprpriate band width reveals the true distribution!

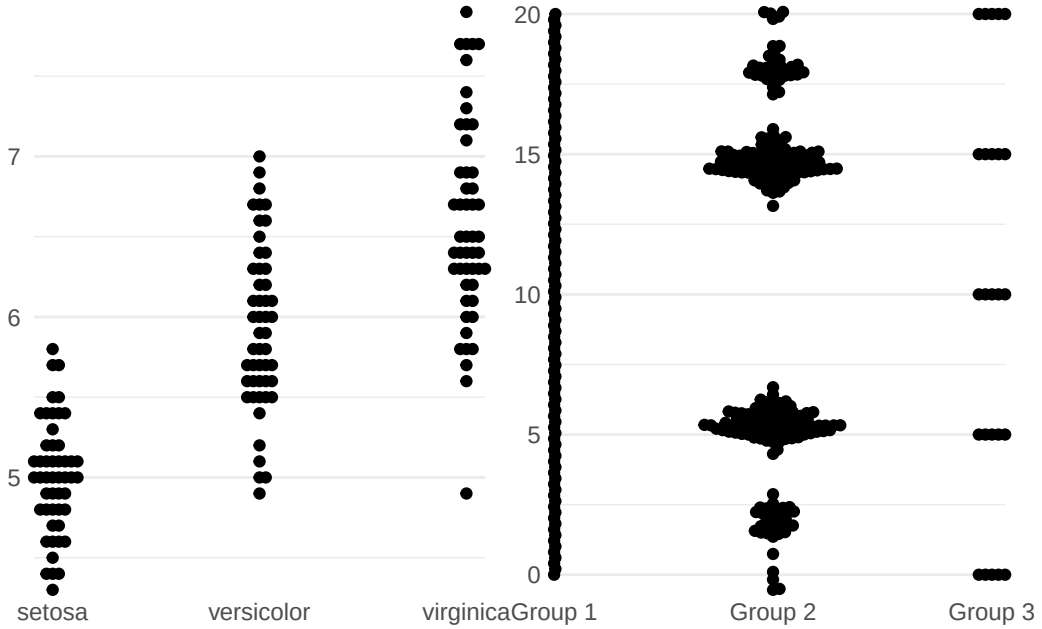
```
base_dist + geom_violin(scale = "count", bw = .3, color = NA, fill = "steelblue")
```



### 9.1.6 Bee Hive Plot

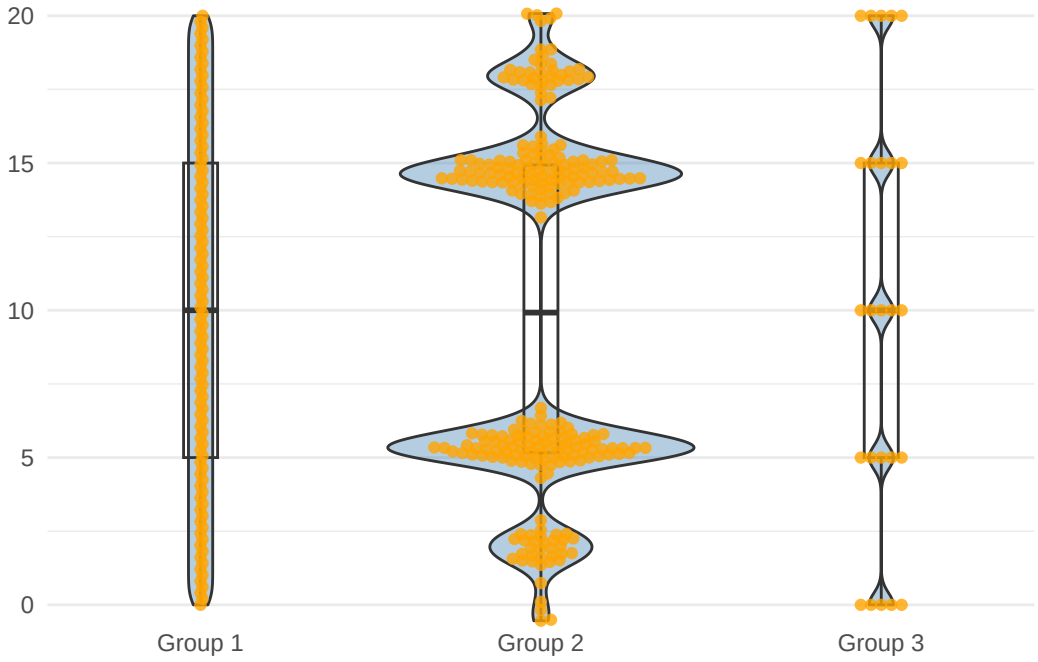
The bee hive plot is a scatter plot that arranges data points as dots to minimize overlap. It's ideal for visualizing small datasets because it creates patterns like a density plot without hiding individual data points.

```
bee_1 <- base2 + geom_beeswarm()
bee_2 <- base_dist + geom_beeswarm()
bee_1 + bee_2
```




All of the plots we have covered so far have their advantages: 1. Box plot shows important statistics 2. Density plot provides high-level view of data shape 3. Bee hive plot "shows" the actual datapoints While combining these plots might make for a crowded visual, with some modifications, it's possible to create a hybrid plot that captures the strengths of each.

```
base_dist + geom_violin(fill = "steelblue", alpha = .4, scale = "count", bw = .4) + geom_b
```



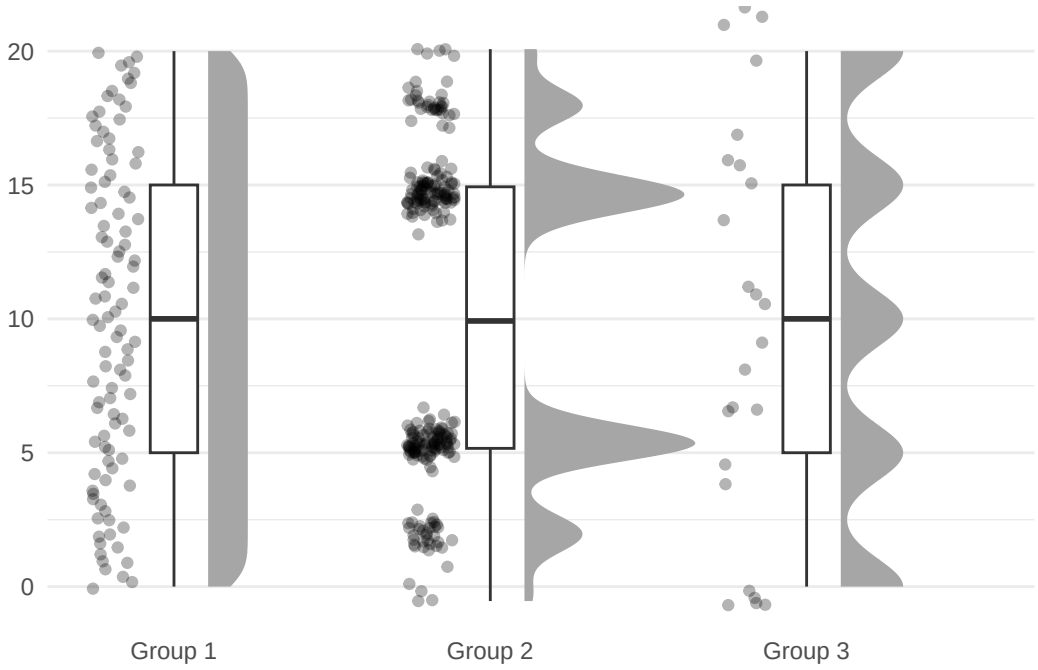

### 9.1.7 Rain Cloud Plot

Rain Cloud Plot combines elements of box plots, violin plots, and density plots. It uses a density plot to show the distribution of the data, a box plot to display the statistics, and individual data points are represented as rain drops. The result is a visually appealing and informative way to visualize a large number of distributions side-by-side, allowing for easy comparisons and identification of patterns.

Isn't this beautiful? We have a box plot, density plot, and jittered points all in the same graph without looking cluttered.

```
base_dist +
  ggdist::stat_halfeye(
    adjust = .3, #bw
    width = .6,
    .width = 0,
    justification = -.2,
    point_colour = NA
  ) +
  geom_boxplot(
    width = .15,
    outlier.shape = NA
  ) +
```

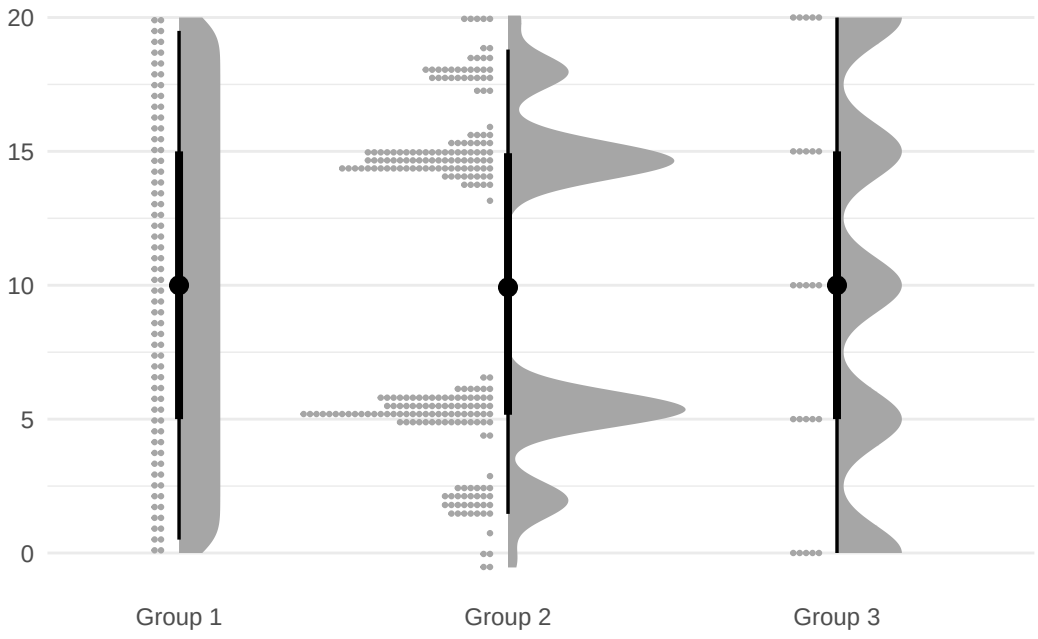
```
  ## add justified jitter from the {gghalves} package
  gghalves::geom_half_point(
    ## draw jitter on the left
    side = "l",
    ## control range of jitter
    range_scale = .4,
    ## add some transparency
    alpha = .3
  ) +
  coord_cartesian(xlim = c(1.2, NA), clip = "off")
```




One alternative option to the boxplot is to stack the data points and use a minimal boxplot representation. While this alternative can be visually appealing, it is important to ensure that your audience understands the visualization and the meaning behind the stacked data points. It may be necessary to provide additional context or include a note explaining the meaning of the stacked slabs to avoid confusion.

```
base_dist +
  ggdist::stat_halfeye(
    adjust = .3,
    width = .6,
    ## set slab interval to show IQR and 95% data range
```

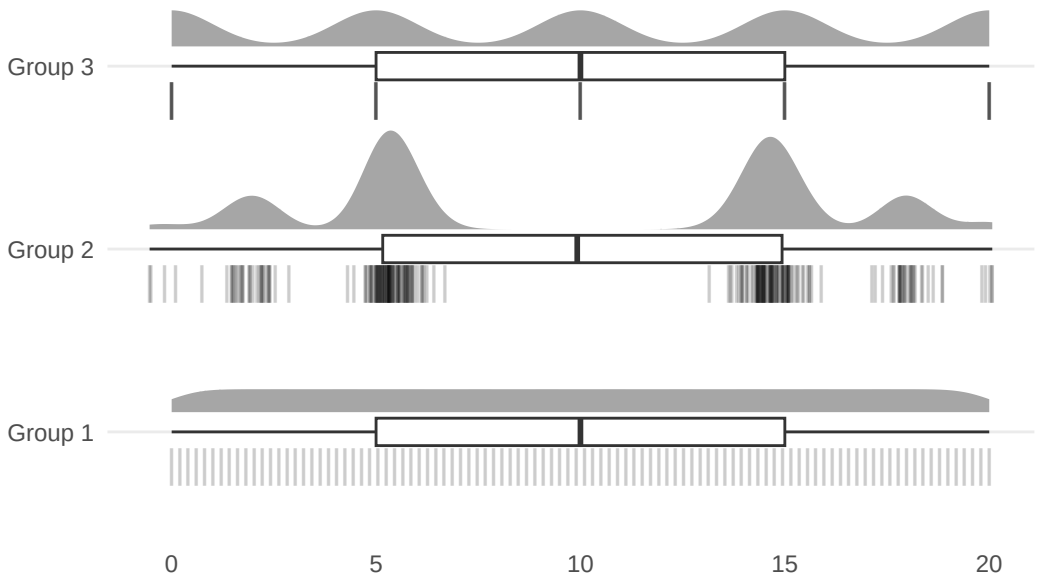
```
    .width = c(.5, .95)
  ) +
  ggdist::stat_dots(
    side = "left",
    dotsize = .8,
    justification = 1.05,
    binwidth = .3
  ) +
  coord_cartesian(xlim = c(1.2, NA))
```




My personal favorite is the rain cloud plot, which combines vertical lines and a bar plot that is rotated horizontally to resemble actual rain clouds.

```
base_dist +
  ggdist::stat_halfeye(
    adjust = .3,
    width = .6,
    .width = 0,
    justification = -.2,
    point_colour = NA
  ) +
  geom_boxplot(
```

```
  width = .15,
  outlier.shape = NA
) +
geom_half_point(
  ## draw horizontal lines instead of points
  shape = "|",
  side = "l",
  size = 5,
  alpha = .2,
  transformation = position_identity()
) +
coord_cartesian(xlim = c(1.2, NA), clip = "off")+ coord_flip()
```




## 9.2 Proportions

Another big collection of graphs is concerned with communicating proportions and composition.

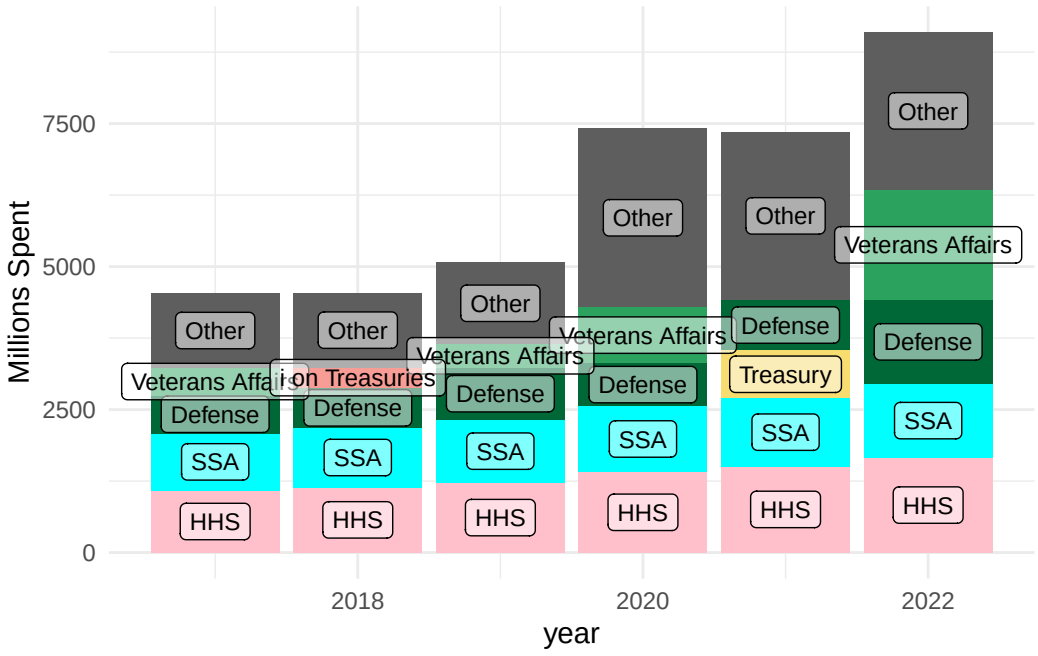
### 9.2.1 Stacked Bar Charts

```
us_spending <- read_csv("data/USFR_StmtNetCost_2017_2022.csv") %>% janitor::clean_names()

spending_plot_data <- us_spending %>% group_by(year) %>% mutate(rank = rank(-1*net_cost_in
  mutate(other = agency == "Other") %>%
  group_by(other) %>%
  arrange(desc(n),.by_group = T) %>%
  ungroup() %>%
  mutate(order = -1*row_number()) %>%
  mutate(agency = recode( agency,
"Department of Veterans Affairs" = "Veterans Affairs",
"Department of Health and Human Services" = "HHS",
"Department of Defense" = "Defense",
"Social Security Administration" = "SSA",
"Department of the Treasury" = "Treasury",
"Interest on Treasury Securities Held by the Public" = "i on Treasuries"
)) %>% mutate(agency = factor(agency,c("Other", "i on Treasuries", "Veterans Affairs", "De
```

A stacked bar chart is a type of graph used to visualize the distribution of a categorical variable. It is similar to a regular bar chart, but in a stacked bar chart, each bar is divided into sections, with each section representing a different category within the variable. The height of each section corresponds to the proportion or frequency of the category within that bar. Stacked bar charts are particularly useful when comparing the distribution of a variable across different subgroups or time periods, as they allow for easy visualization of both the overall distribution as well as the relative proportions of each subgroup or category within the variable.

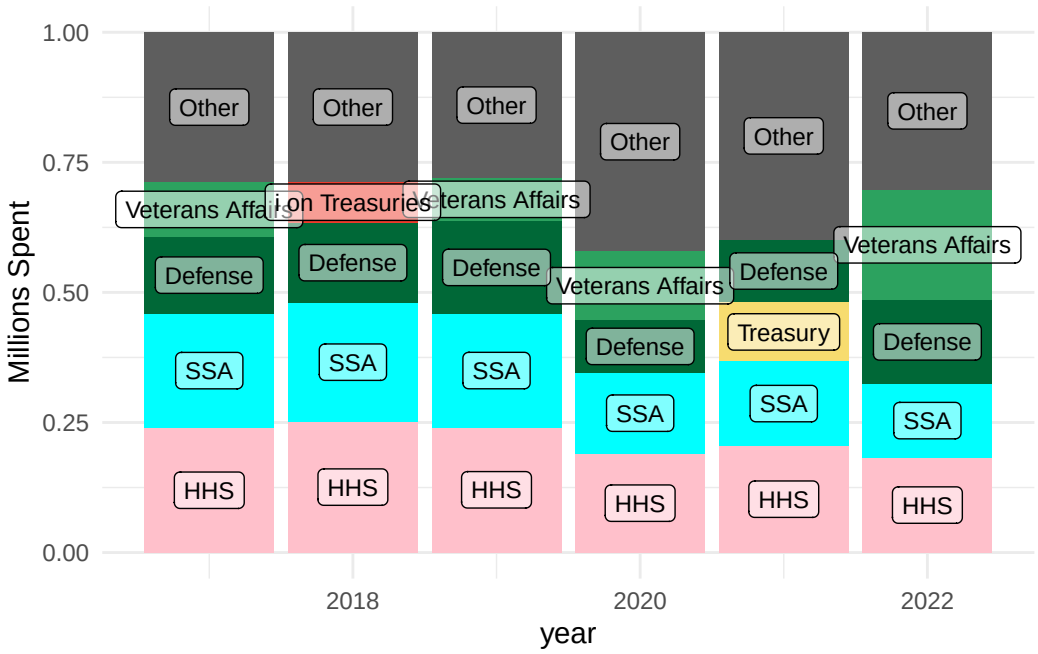
As an example we will use US Experndditures across departments. Only top four departments are shown, the rest are collected into "other". The graph below shows absolute values and its components across years.

```
spending_plot_data %>%
  ggplot(aes(x = year, y = n, fill = agency, label = agency)) +
  geom_col(position="stack", show.legend = F) +
  scale_fill_manual(values = c("#5E5E5E", "#EF3B2C", "#2CA25F", "#006837", "#F7DC6F", "#00
  geom_label(size = 3, aes(group = agency), position = position_stack(vjust = 0.5), fill =
```



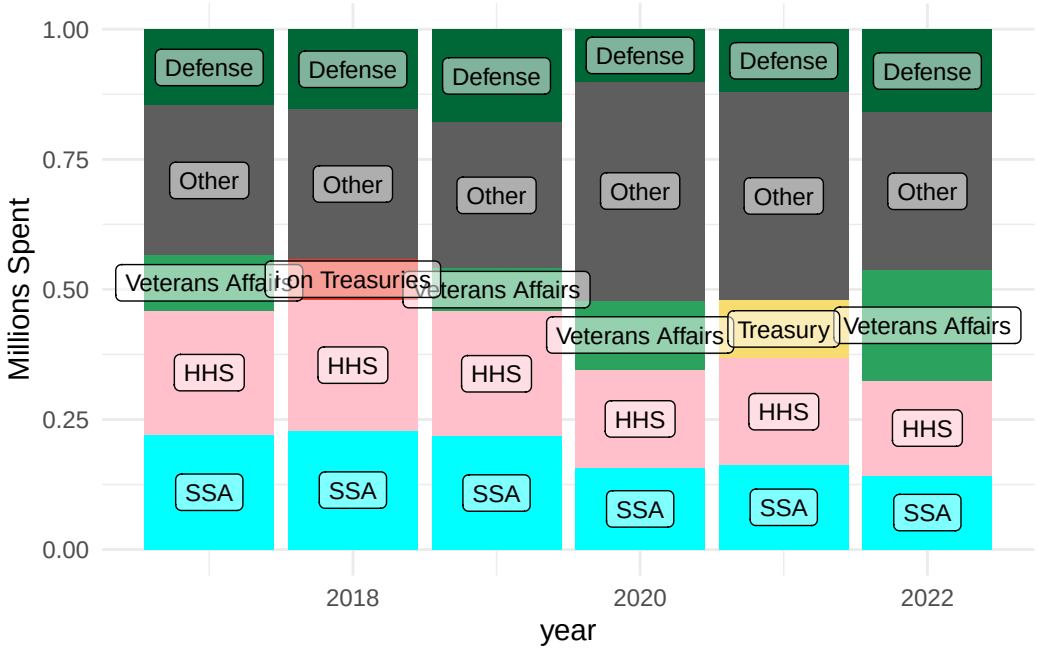

What if we are not concerned with absolute values, but relative proportions? We can use percentage stacked chart.

```
spending_plot_data %>%
  ggplot(aes(x = year, y = n, fill = agency)) +
  geom_col(position="fill", show.legend = T) +
  scale_fill_manual(
    values = c("#5E5E5E", "#EF3B2C", "#2CA25F", "#006837", "#F7DC6F", "#00FFFF", "#FFC0CB"
  theme_minimal() +
  theme(legend.position = "none") +
  labs(y = "Millions Spent", fill = "Department") +
  geom_label(aes(x = year, y = n, label = agency, group = agency), size = 3, position = po
```




Stacked charts are useful for visualizing the distribution of categorical variables, but they can be challenging to compare categories in the middle. Typically, the easiest categories to compare are the ones at the top and bottom of the stack. For example, suppose we want to compare the trend of the Department of Defense and the Social Security Administration (SSA) over time. In this case, we can move these categories to the top and bottom positions of the stacked chart to make it easier to compare their relative sizes and trends.

```
spending_plot_data %>% mutate(agency = factor(agency,c("Defense", "Other", "i on Treasurie
  ggplot(aes(x = year, y = n, fill = agency)) +
  geom_col(position="fill", show.legend = T) +
  scale_fill_manual(
    values = c("#006837", "#5E5E5E", "#EF3B2C", "#2CA25F",  "#F7DC6F", "#FFC0CB", "#00FFFF
  theme_minimal() +
  theme(legend.position = "none") +
  labs(y = "Millions Spent", fill = "Department") +
  geom_label(aes(x = year, y = n, label = agency, group = agency), size = 3, position = po
```




### 9.2.2 Pie Chart

data used from: The Growth Lab at Harvard University. The Atlas of Economic Complexity. http://www.atlas.cid.harvard.edu.

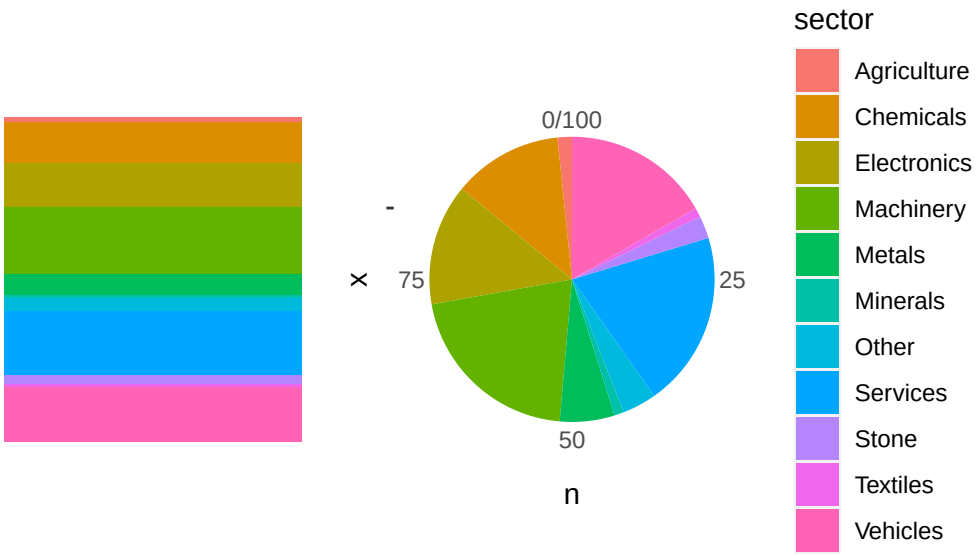
As an example dataset we will be used Japan's export basket from 2020.

```
japan_export <- read_csv("data/japan_export_2020.csv") %>% rename("export" = `Gross Export

japan_sectors <- japan_export %>% count(sector, wt = share)
```

Pie charts are a variation of bar charts where each category is represented as a slice of a circle. While pie charts can effectively communicate when one category is significantly larger or smaller than the others, they become difficult to read and compare accurately when there are many categories or the differences between them are small. Comparing angles and areas of the slices can be confusing, leading to misinterpretation of the data.

```
pie_1 <- japan_sectors %>%
  ggplot(aes(x="", y=n, fill = sector)) +
  geom_bar(stat="identity", width=1) +
    theme_void()  +
  theme(legend.position = "none")
```

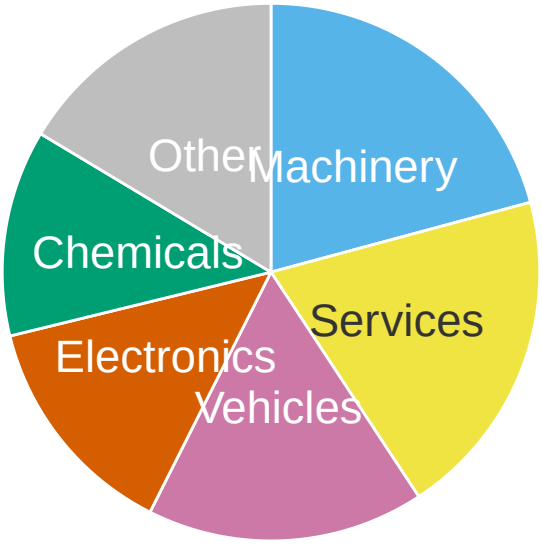
```
pie_2 <- japan_sectors %>%
  ggplot(aes(x="", y=n, fill = sector)) +
  geom_bar(stat="identity", width=1) +
  coord_polar("y", start=0) +
  theme(panel.background = element_rect(fill = "white"))

pie_1 + pie_2
```




But if absolutely must use a pie chart here are some rules to keep in mind: 1. Limit the number of categories to 5-7 at most. 2. Consider grouping small categories into an "Other" category to avoid clutter. 3. Arrange the slices in decreasing order of size, starting at 12 o'clock to aid in comparing them. 4. Include the category labels directly on the chart instead of relying solely on a legend. 5. Add separators between slices to help with distinguishing between them. However, keep in mind that this can also add visual clutter, so use with discretion.

```
japan_sectors %>%
  mutate(sector = ifelse(n<11,"Other",sector)) %>%
  count(sector, wt = n) %>%
  mutate(other = sector == "Other") %>%
  group_by(other) %>%
  arrange(desc(n),.by_group = T) %>%
  ungroup() %>%
  mutate(prop = n / sum(japan_sectors$n) * 100) %>%
  mutate(ypos = cumsum(prop)- 0.5*prop ) %>%
```

```
  mutate(order = -1*row_number()) %>%
ggplot(aes(x="", y=n, fill= fct_reorder(sector, order))) +
  geom_bar(stat="identity", width=1, color =
             "white") +
  coord_polar("y", start=0) +
    theme_void() +
  theme(legend.position="none") +
  geom_text(aes(y = ypos, label = sector), color = c("white","#333333", rep("white",4)), s
  scale_fill_manual(values = c("grey", "#009E73", "#D55E00", "#CC79A7", "#F0E442", "#56B4E
```

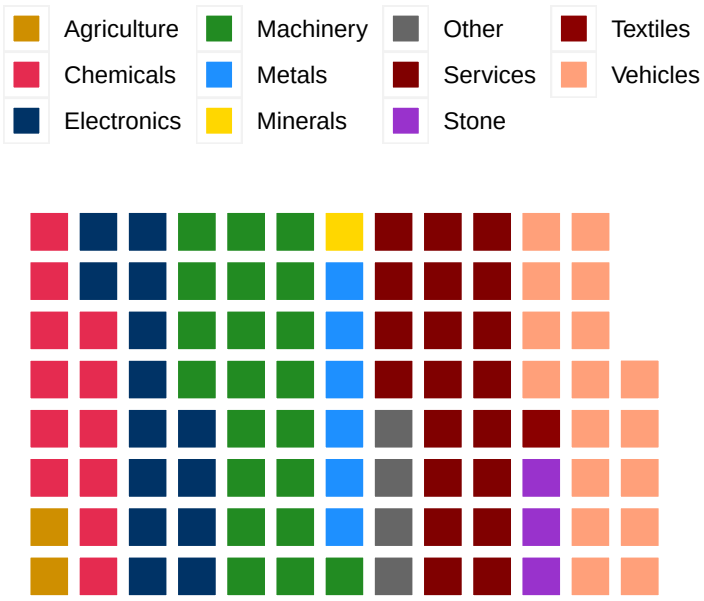



### 9.2.3 Waffle Chart

One alternative to a pie chart could be a waffle chart (these food names make me hungry). It is a grid-like visualization that resembles a waffle or a checkerboard. Each square in the grid represents a proportion of the total data, making it a useful way to visualize proportions or percentages in a visually appealing way. However, they are also vulnerable to large numbers of categories. But what they are truly great at is giving the sense of proportions and sizes. Waffle chart will significantly benefit from interactivity.

```
# waffle_data <- waffle_iron(iris, aes_d(group = Species))
#
# ggplot(waffle_data, aes(x, y, fill = group)) +
#   geom_waffle() +
#   coord_equal() +
```

```
#   scale_fill_viridis_d() +
#   theme_waffle() +
#   theme(legend.position = "top", legend.title = element_blank()) +
#   labs(x = element_blank(), y = element_blank())

japan_sectors %>%
    # mutate(sector = ifelse(n<11,"Other",sector)) %>%
    # count(sector, wt = n) %>%
    # mutate(other = sector == "Other") %>%
    # group_by(other) %>%
    # arrange(desc(n),.by_group = T) %>%
    # ungroup() %>%
    mutate(other = sector == "Other") %>%
    uncount(weights = round(n,0)) %>%
    group_by(other) %>%
    arrange(desc(n),.by_group = T) %>%
    ungroup() %>%
    waffle_iron(aes_d(group = sector)) %>%
    ggplot( aes(x, y, fill = group))+ geom_waffle() + coord_equal() +
    # scale_fill_viridis_d() +
    theme_waffle() +
    theme(legend.position = "top", legend.title = element_blank()) +
    labs(x = element_blank(), y = element_blank()) +
    scale_fill_manual(values = c("#CF8F00", "#E52B50", "#003366", "#228B22", "#1E90FF", "#
```

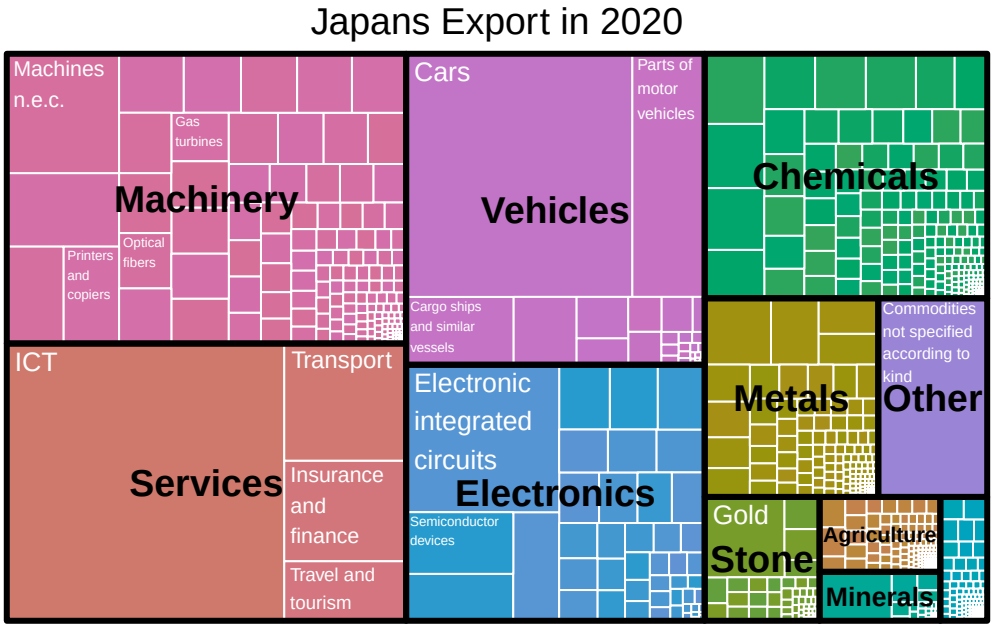


### 9.2.4 Tree Maps

What if we have a lot of data hierarchical data? Treemaps!

Treemaps are a type of visualization that allows you to display hierarchical data in a way that is easy to understand. Each node in the hierarchy is represented by a rectangle, and the size of the rectangle corresponds to the proportion of the total data. The nodes are organized in a way that preserves the hierarchy, with parent nodes containing smaller child nodes. This allows you to quickly identify which nodes are the largest and which are the smallest, as well as the relationships between them. Tree maps are especially useful for displaying large amounts of data in a compact and intuitive way. Tree maps can become very cluttered and interactivity is almost always necessary for such detailed plots. Check out the same plot from the source website.

```
library(treemap)
treemap(japan_export,
        index = c("sector","name"),
        vSize = "export",
        type = "index",
        fontsize.labels = c(14,10),
        fontcolor.labels = c("black","white"),
        fontface.labels = c(2,1),
        bg.labels = 0,
        align.labels = list(
          c("center","center"),
          c("left","top")
        ),
        border.col = c("black","white"),
        border.lwds = c(3,1),
        title = "Japans Export in 2020",
        fontsize.title = 14)
```




## 9.3 Correlation

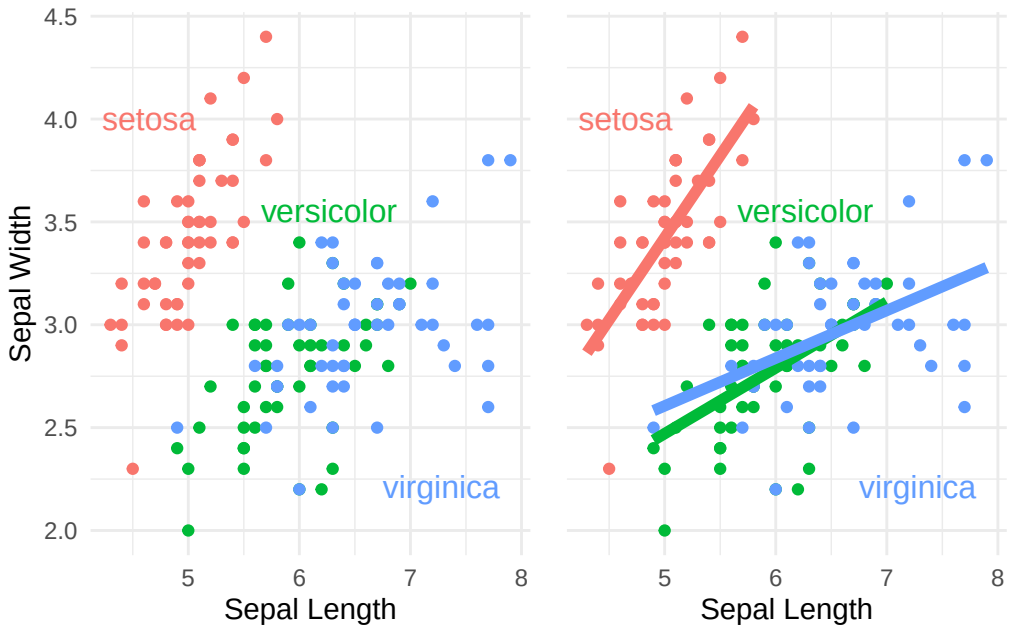
In addition to understanding the distribution of individual variables, it is important to examine the relationship between pairs of variables. Correlation plots are a useful tool for visualizing many aspects of data: relationships between variables (or lack there of), clustering, outliers, etc.

### 9.3.1 Scatter Plot

The most common visualization is scatter plot! It is not a secret for anyone that scatter plots are amazing and perhaps the most persuasive types of plot. We can add a fitted lines to the plot to better show the relationships between the variables.

```
scatter_1 <- iris %>% drop_na() %>% ggplot(aes(x = Sepal.Length, y = Sepal.Width, color =
  geom_point(show.legend = F) +
  geom_dl(aes(label = Species), method = "smart.grid") +
  theme_minimal() +
  labs(x = "Sepal Length", y = "Sepal Width")

scatter_2 <- scatter_1 + geom_smooth(se = F, fullrange = F, show.legend = F, method = "lm"
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank())
```

```
scatter_1 + scatter_2
```




## 9.4 Change over Time

We have already seen plots that incorporate time change. Time series plots typically have time on the x-axis and the variable being measured on the y-axis. They can show trends, patterns, and seasonal fluctuations in the data.

### 9.4.1 Line Chart

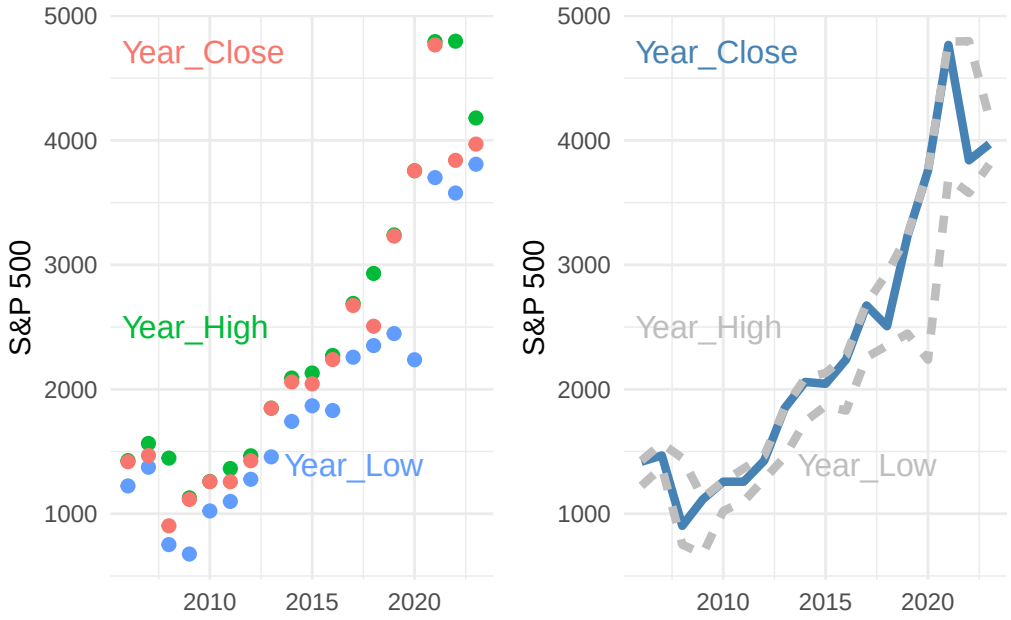
Most common

S&P 500 stock market index since 1927. Historical data is inflation-adjusted using the headline CPI and each data point represents the month-end closing value.

```
sp500 <- tribble(
  ~Year, ~Average_Closing_Price, ~Year_Open, ~Year_High, ~Year_Low, ~Year_Close, ~Annual_P
  2023, 4020.94, 3824.14, 4179.76, 3808.10, 3970.04, 3.40,
  2022, 4097.49, 4796.56, 4796.56, 3577.03, 3839.50, -19.44,
  2021, 4273.41, 3700.65, 4793.06, 3700.65, 4766.18, 26.89,
  2020, 3217.86, 3257.85, 3756.07, 2237.40, 3756.07, 16.26,
```

```
  2019, 2913.36, 2510.03, 3240.02, 2447.89, 3230.78, 28.88,
  2018, 2746.21, 2695.81, 2930.75, 2351.10, 2506.85, -6.24,
  2017, 2449.08, 2257.83, 2690.16, 2257.83, 2673.61, 19.42,
  2016, 2094.65, 2012.66, 2271.72, 1829.08, 2238.83, 9.54,
  2015, 2061.07, 2058.20, 2130.82, 1867.61, 2043.94, -0.73,
  2014, 1931.38, 1831.98, 2090.57, 1741.89, 2058.90, 11.39,
  2013, 1643.80, 1462.42, 1848.36, 1457.15, 1848.36, 29.60,
  2012, 1379.61, 1277.06, 1465.77, 1277.06, 1426.19, 13.41,
  2011, 1267.64, 1271.87, 1363.61, 1099.23, 1257.60, 0.00,
  2010, 1139.97, 1132.99, 1259.78, 1022.58, 1257.64, 12.78,
  2009, 948.05, 931.80, 1127.78, 676.53, 1115.10, 23.45,
  2008, 1220.04, 1447.16, 1447.16, 752.44, 903.25, -38.49,
  2007, 1477.18, 1416.60, 1565.15, 1374.12, 1468.36, 3.53,
  2006, 1310.46, 1268.80, 1427.09, 1223.69, 1418.30, 13.62)
```

```
sp500_scatter <- sp500 %>%
select(-c(Average_Closing_Price,Year_Open,Annual_Percent_Change)) %>% pivot_longer(-Year)
ggplot(aes(x = Year, y = value, color = name)) +
  geom_point(size = 2) +
  geom_dl(aes(label = name), method = "smart.grid") +
  theme_minimal() +
  theme(legend.position = "none") +
  labs(x = element_blank(), y = "S&P 500")

sp500_line <- sp500 %>%
select(-c(Average_Closing_Price,Year_Open,Annual_Percent_Change)) %>% pivot_longer(-Year)
ggplot(aes(x = Year, y = value, color = name)) +
  geom_line(aes(linetype = year_close),linewidth = 1.5) +
  scale_color_manual(values = c("steelblue", "grey", "grey")) +
  geom_dl(aes(label = name), method = "smart.grid") +
  theme_minimal() +
  theme(legend.position = "none") +
  labs(x = element_blank(), y = "S&P 500")

sp500_scatter + sp500_line
```

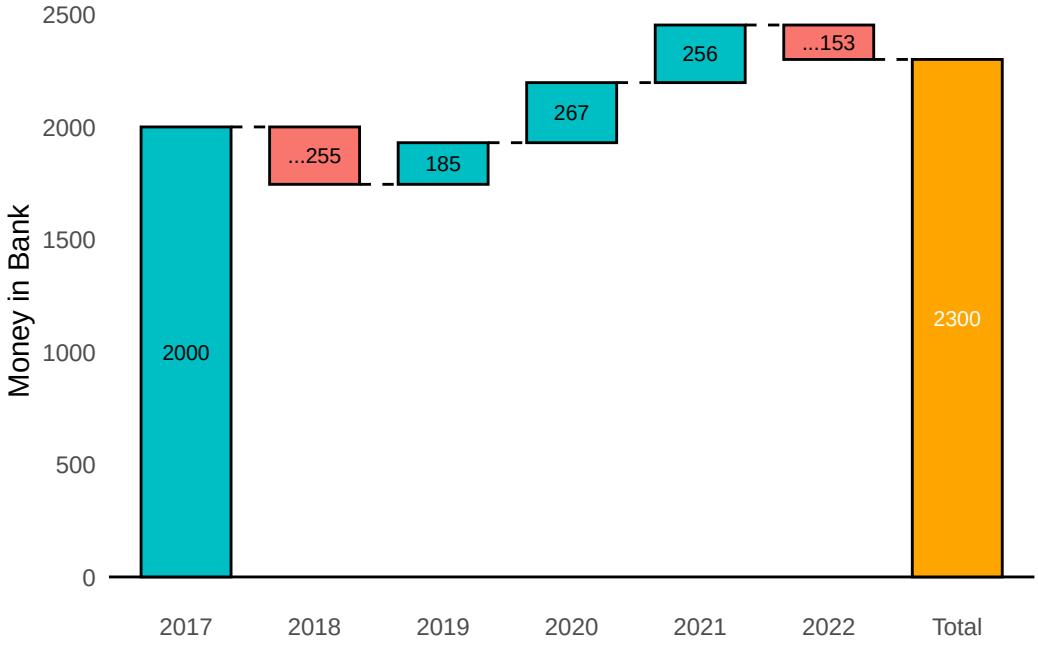



## 9.5 Waterfall Graph

Waterfall charts, also known as bridge charts, are a type of bar chart used to visualize the cumulative effect of sequentially introduced positive or negative values. The graph is named "waterfall" because it resembles a series of falling water droplets. Each bar in the chart represents a value and is color-coded to indicate whether it contributes to an increase or decrease in the cumulative total. They are useful for visualizing the relative contributions of positive and negative factors that affect the net change in the value being analyzed.

```
waterfall_data <- tribble(
  ~year, ~bank, ~change,
  2017, 2000, 2000,
  2018, 1745, -255,
  2019, 1930, 185,
  2020, 2197, 267,
  2021, 2453, 256,
  2022, 2300, -153,
) %>% transmute(as.character(year), change)
library(waterfalls)


waterfall(waterfall_data,  calc_total = TRUE,
```

```
        total_rect_color = "orange",
        total_rect_text_color = "white") +
theme_minimal() +
theme(panel.grid = element_blank()) +
labs(y = "Money in Bank", x = NULL)
```



# Part V

# Power Analysis and Simulations

# References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.