R Workshop

Dilan Caro

2024-05-30

Contents

Welcome

Are you ready to embark on an exciting journey into the world of data analysis and statistical exploration? Imagine having the power to unlock the hidden insights within vast datasets, create stunning visualizations, and make data-driven decisions that can shape the future. Welcome to the world of R programming! R is not just a language; it's your key to uncovering the stories data has to tell. Whether you're a budding data scientist, a curious researcher, or someone who simply loves solving puzzles, R offers you a thrilling adventure where you'll learn to command data with precision and creativity. Get ready to be amazed by the endless possibilities of R and join a global community of data enthusiasts who are reshaping the world, one analysis at a time.

In this website, you will have resources, examples and practice to master R.

Objectives

Session 1: R Basics (1 hour 45 min)

Part 1: Introduction to R

1. What is R and why use it?
2. Installing R and RStudio
3. Basic RStudio layout and functionality

Part 2: R Fundamentals

1. R as a calculator
2. Variables and data types (numeric, character)

3. Basic arithmetic operations

Part 3: Data Structures

1. Vectors: Creating, indexing, and operations
2. Data frames: Creating and exploring data frames
3. Importing and exporting data (CSV files)

Part 4: Data Manipulation

1. Subsetting and filtering data
2. Adding, removing, and renaming columns
3. Basic data summary and exploration

Part 5: Basic Data Visualization

1. Creating simple plots using plot(), hist(), pie(),barplot(),boxplot()

Session 2: Intermediate R I (1 hour 45 min)

Part 1: Functions and Control Structures

1. Writing and using functions
2. If statements and loops (for and while)

Part 2: Data Wrangling: Cleaning and Transformation

1. Reshaping data using dplyr functions (filter, arrange, mutate, summarize)
2. Handling missing data

Part 3: Advanced Data Visualization

1. Creating customized plots with ggplot2
2. Adding titles, labels, and themes to plots

Part 4: Statistical Analysis

1. Introduction to hypothesis testing and statistical tests
2. Performing t-tests and chi-squared tests

(Optional) Part 5: Working with Dates and Times

1. Handling date and time data in R
2. Common date and time functions

(optional) Session 3: Advanced R

Part 1: Advanced Data Manipulation with dplyr

1. Grouping and summarizing data
2. Joining and merging datasets

Part 2: Text Data Processing

1. Manipulating and analyzing text data using regular expressions
2. Text mining basics

Part 3: Building Predictive Models

1. Introduction to machine learning in R
2. Creating predictive models with caret

Part 4: Bookdown: Using R markdown to create books

1. Introduction to Bookdown Package
2. Creating a simple bookdown book

Part 5: Version Control and Collaboration

1. Using Git and GitHub for version control and collaboration in R projects

R Basics

In our first session, we will lay the foundation for your R journey. We'll start with an introduction to R, delving into why it's a popular choice among data enthusiasts. You'll learn how to install R and RStudio, a user-friendly integrated development environment (IDE). We'll explore the basic layout and functionality of RStudio, setting the stage for your coding adventures.

We'll dive into R's fundamentals, treating it as your trusty calculator. You'll discover the various data types, from numeric to character, and grasp essential arithmetic operations. Moving forward, we'll explore R's data structures, including vectors for data storage and data frames for structured datasets. You'll also become adept at importing and exporting data using common formats like CSV files.

We won't stop there. The next part of our session is all about data manipulation. You'll learn how to subset and filter data, add, remove, and rename columns, and gain skills in basic data summarization and exploration. Finally, we'll wrap up with basic data visualization, where you'll create simple plots to visually represent your data.

Part I: Introduction to R

What is `R` ?

R is a programming language and open-source software environment that is widely used for statistical computing, data analysis, and graphics. It was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, in the early 1990s and is now maintained by the R Development Core Team. R is particularly popular among statisticians, data scientists, and researchers for its extensive statistical and graphical capabilities.

In an brief sentence: `R` is a dialect of S.

What is `S` ?

The S language, developed by John Chambers and others at Bell Telephone Laboratories, began in 1976 as an internal statistical analysis tool, originally based on Fortran libraries. It evolved significantly over time: in 1988, it was rewritten in C, leading to a system more akin to its present form (Version 3).

The statistical analysis capabilities of S were detailed in the 1988 book "Statistical Models in S" (the white book) by Chambers and Hastie. The most current version, Version 4, released in 1998 and documented in Chambers' "Programming with Data" (the green book), remains in use. The ownership of S has changed hands several times: Bell Labs licensed it to StatSci (later Insightful Corp.) in 1993, Insightful then bought it from Lucent in 2004, and after a series of acquisitions, TIBCO Software Inc. has owned and exclusively developed S since 2008. Insightful had added features, including GUIs, and marketed it as S-PLUS. Despite these changes, the core S language has remained largely unchanged since 1998, the year it earned the prestigious ACM Software System Award.

Key features and characteristics of R :

1. **Data Analysis and Statistics:** `R` provides a wide range of statistical techniques and libraries for data analysis, hypothesis testing, regression analysis, clustering, and more. It's known for its flexibility in handling data and conducting statistical experiments.

2. **Data Visualization:** `R` offers powerful tools for creating a variety of high-quality data visualizations, including scatterplots, bar charts, histograms, and heatmaps. The ggplot2 package, in particular, is a popular choice for creating customized graphics.

3. **Open Source:** `R` is open-source software, which means that it is freely available for anyone to use, modify, and distribute. This has led to a vibrant community of users and developers who contribute packages and extensions to enhance its functionality.

4. **Package System:** `R` has a rich ecosystem of packages (libraries) that extend its core functionality. These packages cover a wide range of domains, from machine learning and time series analysis to bioinformatics and geospatial data analysis. Users can easily install and use these packages to tailor R to their specific needs.

5. **Cross-Platform:** `R` runs on various operating systems, including Windows, macOS, and Linux, making it accessible to a wide range of users.

6. **Command-Line Interface:** `R` primarily uses a command-line interface, although there are graphical user interfaces (GUIs) available, such as RStudio, which provide a more user-friendly environment for coding and data analysis.

7. **Community Support:** `R` has a large and active community of users and developers who provide support, share code and tutorials, and contribute to the ongoing development of the language.

`R` is a versatile tool used in various fields, including academia, industry, finance, healthcare, and more, for tasks such as statistical analysis, data visualization, and predictive modeling. Its popularity continues to grow as data-driven decision-making becomes increasingly important in many domains.

Installing R and Rstudio

STAR 2024 – R Workshop

Find here the instructions to install R and RStudio.
I have created a step-by-step guide on the installation on MacOS.

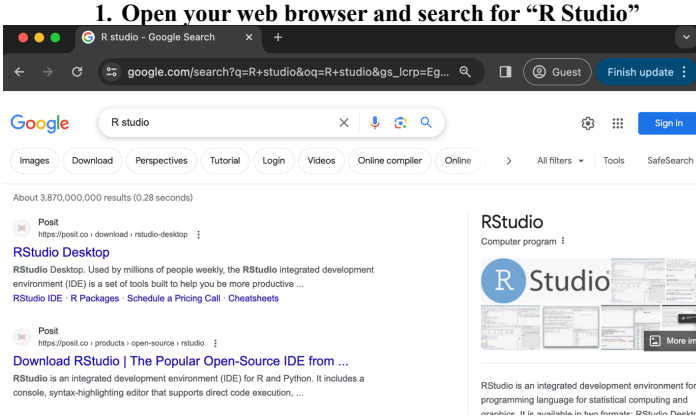
Alternatively, you can watch the following great videos depending on your sy

1. Installation of R on windows by Roger Peng
2. Installation of R on Mac by Roger Peng
3. How to install Rstudio on Mac
4. How to install Rstudio on Windows

If you have any trouble during the installation , please reach out to your mento
will be able to provide some troubleshooting and guidance.

R studio and R installation:

1. Open your web browser and search for "R Studio"



2. Click on https://posit.co/download/rstudio-desktop/

Find the installation guide in the below pdf file.

Alternatively , you can look at these videos .

1. Installation of R on windows by Roger Peng
2. Installation of R on Mac by Roger Peng
3. How to install Rstudio on Mac
4. How to install Rstudio on Windows

Basic RStudio layout and functionality

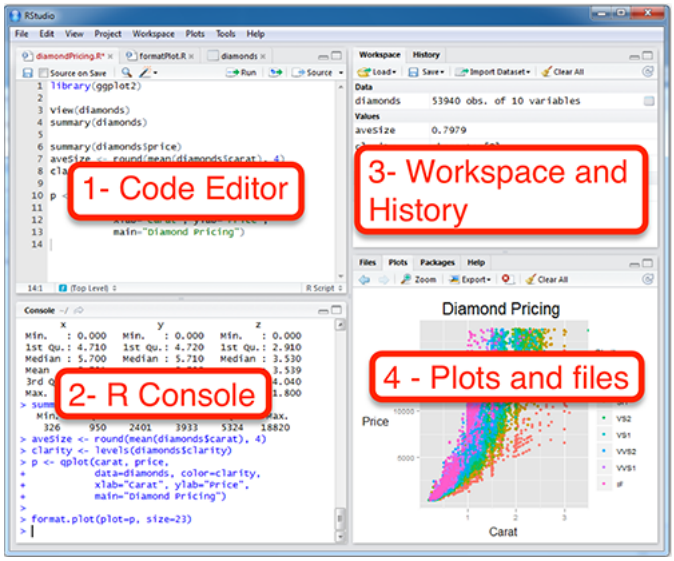


Figure 1: From: http://www.sthda.com/english/wiki/r-basics-quick-and-easy

1. **Code Editor/ R script:** Here you can write either R code , or Rmarkdown. That is we can include instructions for our computer to execute.
2. **R console :** Here you see the output of the code you run , if you write code here, it will automatically be run after `enter` and cannot be traced back, that is why the R Script is useful for reusable code.
3. **Workspace and history:** This space display the variables created , in use, the history , building , git and more. To check if your data has been loaded correctly you can check here to see it loaded.
4. **Plots and files:** It will display the graphs and plots created, you can switch back and forth , export, save and more. Also, you can select packages, get help on `R` functions and more.

What are packages?

The power of R lies in the packages. Since `R` is open source, many people create packages i.e, R scripts that contain functions for specific problems , may it be standard deviation , statistics, machine learning and more.

- To install a package, simply type

```
install.packages("package name")
```

- Install a package from Bioconductor: `biocLite()`

- Install a package from GitHub: `devtools::install_github()`
- View the list of installed packages: `installed.packages()`
- Folder containing installed packages: `.libPaths()`
- To load a package

```
library(package name)
```

- View loaded packages

```
search()
```

- to Unload an R package:

```
detach(package name, unload = TRUE)
```

- Remove installed packages:

```
remove.packages()
```

- Update installed packages:

```
update.packages()
```

Part II: R Fundamentals

Before starting, here are some Official manuals and books we will be learning from:

1. https://cran.r-project.org/doc/manuals/r-release/R-intro.html [R Core Team, 2024c]
2. https://cran.r-project.org/doc/manuals/r-release/R-data.html [R Core Team, 2024a]
3. https://cran.r-project.org/doc/manuals/r-release/R-exts.html [R Core Team, 2024b]
4. https://cran.r-project.org/doc/manuals/r-release/R-lang.html [R Core Team, 2024d]
5. R programming for Data Science by Roger D. Peng. [Peng, 2024]
6. R for Data Science [Wickham and Grolemund, 2017]
7. Intro R book [Antonio, 2020]
8. R workshop [Uni, 2024]
9. Intro to R [Int, 2024]

R as a calculator

The best way to get used to R is to use it as a calculator.

You can start by using the R console and doing simple operations.

Basic arithmetic operations

- Addition

```
3+5
#> [1] 8
```

- Subtraction

```
143-12
#> [1] 131
```

- Multiplication

```
4*5
#> [1] 20
```

- Division

```
180/23
#> [1] 7.826087
```

- Exponientiation

```
4^2
#> [1] 16
```

Arithmetic Functions

this functions may be useful to replace your calculator

- Absolute value

```
abs(-23)
#> [1] 23
```

- Square root

```
sqrt(16)
#> [1] 4
```

- Remainder/modulo

```
7 %% 3
#> [1] 1
```

- Logarithms and exponentials i.e,

$$\log_a b = c, \quad ln_e b = a, \quad e^a = b$$

```
log2(4)
#> [1] 2
log10(1000)
#> [1] 3
log(4)
#> [1] 1.386294
exp(8)
#> [1] 2980.958
2.71828^8
#> [1] 2980.942
```

Variables and data types (numeric,character)

Assigment Operators

in R, to create a variable , you can use the assigment symbol <-, or = however, the later is not commonly used in R.

To assign the value of 7 to x , we will do

```
x <- 7
print(x)
#> [1] 7
```

We can now perform operations on these variables

```
3*x+3 # 3*7+3 = 21+3 =24
#> [1] 24
```

Remark: R is case sensitive so, x is different from X

calling `print(X)` will output error

```
print(X)
#> Error in eval(expr, envir, enclos): object 'X' not found
```

Data types

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

Exercise 1

Run the following code, then use typeof(), class() functions to find out the data type and/or class object.

```
my_numeric <- 42.5
John_jay <- "university"
my_logical <- TRUE
my_date <- as.Date("05/29/2018", "%m/%d/%Y")
```

Use `typeof()` or `class()`function to find out what the data type of a variable is .

What is the difference between typeof() and class()?

Here we can see that `typeof(my_date)` is a double, and `class(my_date)` is `Date`. This is because `typeof` output the lowest level data type of the object.

While `class` outputs the class of the object.

If you are writing code that involves checking whether an element is of an specific data type , then you need to be careful on how you check that. Depending on the function , it may give you a true value when in reality you want a false value to be returned.

For example, Imagine you are asked to check if all dates in a dateframe have the correct data type.

In some cases it might be `"05/29/2018"` but in a rare case (maybe due to a data entry error), there is a `"42.5"`

```
typeof(my_date) == typeof(my_numeric)
#> [1] TRUE
```

In the previous comparison , it returns true, meaning that the two data types are the same, maybe youu thought you are comparing if both are date type, but in reality , you are comparing the lowest level data types which are indeed equal (double)

Instead, you should do.

```
class(my_date) == class(my_numeric)
#> [1] FALSE
```

Character Data type A character stores character values or strings

```
char <- "This is a character data type"
char
#> [1] "This is a character data type"
typeof(char)
#> [1] "character"
```

Numeric Data type

This is for numerical values .

```
num <- 3
print(num)
#> [1] 3
num_2 <- -2.35
num_2
#> [1] -2.35
typeof(num_2)
#> [1] "double"
class(num_2)
#> [1] "numeric"
```

Integer Data type

For integers, we must specify it as so, and to do it , we must convert the data

type. Remark: if it is a decimal, it will remove all decimal, acting as a floor function .

```
int <- as.integer(3.6332)
int
#> [1] 3
typeof(int)
#> [1] "integer"
int2 <- as.integer(7)
int2
#> [1] 7
typeof(int2)
#> [1] "integer"
class(int2)
#> [1] "integer"
```

We can also create a integer by adding an L after it

```
int3 <- 8L
int3
#> [1] 8
```

Remark: It does not work with decimals

```
int4 <- 3.4546L
int4
#> [1] 3.4546
```

Complex Data type Complex data types are stored as `x+yi` , i.e, with the imaginary component

```
compl <- 13+7i
compl
#> [1] 13+7i
typeof(compl)
#> [1] "complex"
complex(real = 23, imaginary = 7)
#> [1] 23+7i
```

Boolean Data type This stores boolean values of `TRUE` and `FALSE`

```
my_bool <- TRUE
my_bool
#> [1] TRUE

typeof(my_bool)
#> [1] "logical"

my_boolean <- F
```

```
my_boolean
#> [1] FALSE
typeof(my_boolean)
#> [1] "logical"
```

Converting Data types

Convert into Numeric

We can convert values as numeric. Using `as.numeric()` to change the type but keeping the values as they are. When converting

- complex: removes the `imaginary` part
- logical: `TRUE` becomes `1` , `FALSE` becomes `0`
- character: to its numerical values, but if it has letters then to `NA`

We can use `is.numeric()` to check if a variable is numeric

```
# Complex
is.numeric(compl)
#> [1] FALSE
number <- as.numeric(compl)
#> Warning: imaginary parts discarded in coercion
number
#> [1] 13
is.numeric(number)
#> [1] TRUE

#Logical
is.numeric(my_bool)
#> [1] FALSE
number2 <- as.numeric(my_bool)
number2
#> [1] 1
is.numeric(number2)
#> [1] TRUE

# Character
char
#> [1] "This is a character data type"
is.numeric(char)
#> [1] FALSE
number3 <- as.numeric(char)
#> Warning: NAs introduced by coercion
number3
#> [1] NA
is.numeric(number3)
```

```
#> [1] TRUE

my_char <- "2023"
is.numeric(my_char)
#> [1] FALSE
number4 <- as.numeric(my_char)
number4
#> [1] 2023
is.numeric(number4)
#> [1] TRUE
```

Convert into integer

```
inte1<-as.integer("234")
inte1
#> [1] 234
typeof(inte1)
#> [1] "integer"

inte2<-as.integer(23+6i)
#> Warning: imaginary parts discarded in coercion
inte2
#> [1] 23
typeof(inte2)
#> [1] "integer"

inte3<-as.integer(F)
inte3
#> [1] 0
typeof(inte3)
#> [1] "integer"
```

Converting into Logical

Return `FALSE` for `0` , `TRUE` otherwise

```
print(as.logical(0))
#> [1] FALSE
typeof(as.logical(0))
#> [1] "logical"

print(as.logical(-324))
#> [1] TRUE
typeof(as.logical(-324))
#> [1] "logical"
```

Exercise 2

Create 1 datatype of each: Character, numeric, integer, complex, Boolean

Getting help

You can use the Plots and files pane (bottom left pane) to click on Help and then search for whichever function you need help with.

You can also use the `?` before each function.

```
?mean
```

This will open the information about the function on the plots and files pane.

Part III: Data Structures

Vectors: Creating, indexing, and operations

```
# Creating a vector
v <- c(1, 2, 3, 4, 5)
print(v)
#> [1] 1 2 3 4 5

# Indexing a vector
print(v[2])  # Access the second element
#> [1] 2

# Vector operations
v2 <- v * 2  # Multiply each element by 2
print(v2)
#> [1]  2  4  6  8 10
```

You can give names to the columns of the vector

```
my_vector <- c("Dilan Caro", "Instructor")
names(my_vector) <- c("Name", "Profession")
my_vector
#>         Name   Profession
#> "Dilan Caro" "Instructor"
```

Exercise 1:

1. Create a vector of your favorite numbers.
2. Access the third element in your vector.
3. Create a new vector that is the square of each element in the original vector.

```
my_vector <- c("Dilan Caro", "Instructor")
names(my_vector) <- c("Name", "Profession")
my_vector
```

```
#>         Name   Profession
#> "Dilan Caro" "Instructor"
```

4. Inspect my_vector using: the attributes(), the length() and the str() function

Matrices

Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

Another example

```
my_matrix <- matrix(1:12, 3, 4, byrow = TRUE)
my_matrix
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    4
#> [2,]    5    6    7    8
#> [3,]    9   10   11   12
```

Matrices can be created by column-binding or row-binding with the `cbind()` and `rbind()` functions.

```
x <- 1:3
y <- 10:12
cbind(x, y)
#>      x  y
#> [1,] 1 10
#> [2,] 2 11
#> [3,] 3 12
rbind(x, y)
#>   [,1] [,2] [,3]
#> x    1    2    3
#> y   10   11   12
```

Seq and rep functions

In R, seq and rep are two functions used to generate sequences and to replicate values, respectively.

0.0.1 `seq` Function:

The seq function is used to create a sequence of numbers.

Usage:

- seq(from, to): Generates a sequence from the 'from' value to the 'to' value with a default increment of 1.
- seq(from, to, by): Generates a sequence from the 'from' value to the 'to' value, with the increment specified by 'by'.
- seq(from, to, length.out): Generates a sequence from the 'from' value to the 'to' value with a specified number of equally spaced points.

Example

```
seq(1, 5)
#> [1] 1 2 3 4 5
seq(1, 10, by = 2)
#> [1] 1 3 5 7 9
seq(1, 10, length.out = 4)
#> [1]  1  4  7 10
```

0.0.2 `rep` Function:

The rep function is used to replicate the values in a vector.

Usage:

- rep(x, times): Replicates each element in 'x' a specified number of 'times'.
- rep(x, each): Replicates each element in 'x' 'each' times before moving to the next element.
- rep(x, length.out): Replicates the values in 'x' up to the 'length.out' number of times in total.

```
rep(1:3, times = 2)
#> [1] 1 2 3 1 2 3
rep(1:3, each = 2)
#> [1] 1 1 2 2 3 3
rep(1:3, length.out = 7)
#> [1] 1 2 3 1 2 3 1
```

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various "apply" functions discussed later, make for a powerful combination.

Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

```
x <- list(1, "a", TRUE, 1 + 4i)
x
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]
#> [1] 1+4i
```

Example

```
my_list <- list(one = 1, two = c(1, 2), five = seq(1, 4, length=5),
          six = c("Dilan", "April"))
names(my_list)
#> [1] "one"  "two"  "five" "six"
```

```
str(my_list)
#> List of 4
#>  $ one : num 1
#>  $ two : num [1:2] 1 2
#>  $ five: num [1:5] 1 1.75 2.5 3.25 4
#>  $ six : chr [1:2] "Dilan" "April"
```

Factors

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a label. Factors are important in statistical modeling and are treated specially by modelling functions like `lm()` and `glm()`.

Using factors with labels is better than using integers because factors are self-

describing. Having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

Factor objects can be created with the factor() function.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
#> [1] yes yes no  yes no
#> Levels: no yes
```

Level are put in alphabetical order, but you can also define the levels.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),levels = c("yes", "no"))
x
#> [1] yes yes no  yes no
#> Levels: yes no
```

Data frames: Creating and exploring data frames

Data frames are used to store tabular data in R.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

```
# Creating a data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Salary = c(50000, 60000, 70000)
)
print(df)
#>      Name Age Salary
#> 1   Alice  25  50000
#> 2     Bob  30  60000
#> 3 Charlie  35  70000

# Exploring data frames
print(dim(df))  # Dimensions of the data frame
#> [1] 3 3
print(colnames(df))  # Column names
#> [1] "Name"   "Age"    "Salary"
print(summary(df))  # Summary statistics
#>      Name                Age           Salary
#>  Length:3           Min.   :25.0   Min.   :50000
#>  Class :character   1st Qu.:27.5   1st Qu.:55000
#>  Mode  :character   Median :30.0   Median :60000
```

```
#>                    Mean   :30.0   Mean   :60000
#>                    3rd Qu.:32.5   3rd Qu.:65000
#>                    Max.   :35.0   Max.   :70000
```

Exercise 2

1. Create a data frame with at least three columns and four rows.
2. Print the number of rows and columns of your data frame.
3. Display summary statistics of your data frame.

Part 2

Exercise 3

1. Inspect a built-in data frame, inspect `mtcars` using `str()`, `head()`
2. Get summary from a variable in a dataframe, use `$` to extract a variable from the dataframe.
3. Now inspect a tibble, inspect `diamonds` from the `ggplot2` library. Use `str()`, `head()`, `summary()`

```
mtcars
str(mtcars)
head(mtcars)
```

```
summary(mtcars$cyl) # use $ to extract variable from a data frame
```

```
library(ggplot2)
head(diamonds)
```

Can you list some differences?

Importing and exporting data (CSV files)

Exporting data to CSV

```
write.csv(df, "my_data.csv", row.names = FALSE)
```

Importing data from CSV

```
df_imported <- read.csv("my_data.csv")
print(df_imported)
#>      Name Age Salary
#> 1   Alice  25  50000
#> 2     Bob  30  60000
#> 3 Charlie  35  70000
```

Exercise 4

1. Create a vector `fav_music` with the names of your favorite artists.
2. Create a vector `num_records` with the number of records you have in your collection of each of those artists.
3. Create a vector `num_concerts` with the number of times you attended a concert of these artists.
4. Put everything together in a data frame, assign the name `my_music` to this data frame and change the labels of the information stored in the columns to `artist`, `records` and `concerts.`
5. Extract the variable `num_records` from the data frame `my_music.`
6. Calculate the total number of records in your collection (for the defined set of artists).
7. Check the structure of the data frame, ask for a `summary.`

Previously, we exported the data and then imported it . Some of you may think, then what is the purpose if we already had the dataframe. The prior was just an example, in reality , you would not have the dataframe loaded in R . You would only have a csv or a data file that a coworker has shared with you or the data engineer has procured for you.

First, we need to obtain the data that we need. For that, please head over to

https://tinyurl.com/JJAY-R-workshop

alternatively,

https://drive.google.com/drive/folders/18W5f2AvKT7IVKnJ73McCzQOOqMdP0CwM?usp=sharing

UPLOAD PICTURE HOLDER

Download the data, for that, click on the arrow of the folder, and choose download. Find where it is located in your computer, obtain the `Path`

UPLOAD PICTURE HOLDER

Some useful instructions regarding path names: get your working directory

- Get working directory

```
getwd()
#> [1] "/Users/dilancaro/Library/Mobile Documents/com~apple~CloudDocs/University/Year 
```

- specify a path name, with forward slash or double back slash

```
path <- file.path("/Users/dilancaro/Library/Mobile Documents/com~apple~CloudDocs/Unive
```

- use a relative path

```
path <- file.path("./Data/")
```

Importing a .txt file

`read.table()` is one great way to import data.

```
path.hotdogs <- file.path(path, "hotdogs.txt")
path.hotdogs    # inspect path name
#> [1] "./Data//hotdogs.txt"
hotdogs <- read.table(path.hotdogs, header = FALSE,
                      col.names = c("type", "calories", "sodium"))
str(hotdogs)    # inspect data imported
#> 'data.frame':    54 obs. of  3 variables:
#>  $ type    : chr  "Beef" "Beef" "Beef" "Beef" ...
#>  $ calories: int  186 181 176 149 184 190 158 139 175 148 ...
#>  $ sodium  : int  495 477 425 322 482 587 370 322 479 375 ...
```

Or like this

```
hotdogs2 <- read.table(path.hotdogs, header = FALSE,
                       col.names = c("type", "calories", "sodium"),
                       colClasses = c("factor", "NULL", "numeric"))
str(hotdogs2)
#> 'data.frame':    54 obs. of  2 variables:
#>  $ type  : Factor w/ 3 levels "Beef","Meat",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
#>  $ sodium: num  495 477 425 322 482 587 370 322 479 375 ...
```

What happened?

Import .csv file

`read.csv()` is another importing function.

Here is an example: - load a data set on swimming pools in Brisbane - column names in the first row; a comma to separate values within rows

```
path.pools <- file.path(path, "swimming_pools.csv")
pools <- read.csv(path.pools)
str(pools)
#> 'data.frame':    20 obs. of  4 variables:
#>  $ Name     : chr  "Acacia Ridge Leisure Centre" "Bellbowrie Pool" "Carole Park" "Centenary Po
#>  $ Address  : chr  "1391 Beaudesert Road, Acacia Ridge" "Sugarwood Street, Bellbowrie" "Cnr Bo
#>  $ Latitude : num  -27.6 -27.6 -27.6 -27.5 -27.4 ...
#>  $ Longitude: num  153 153 153 153 153 ...
```

Import .xlsx file

The package to read excel data into R is `readxl`:

- No external dependencies, easy to download
- Desgined to work with tabular data

```
library(readxl)
path.urbanpop <- file.path(path, "urbanpop.xlsx")
excel_sheets(path.urbanpop) # list sheet names with excel_sheets()
#> [1] "1960-1966" "1967-1974" "1975-2011"
```

Specify a worksheet by name or number, e.g.

```
pop_1 <- read_excel(path.urbanpop, sheet = 1)
pop_2 <- read_excel(path.urbanpop, sheet = 2)
```

inspect and re-combine

```
str(pop_1)
#> tibble [209 x 8] (S3: tbl_df/tbl/data.frame)
#>  $ country: chr [1:209] "Afghanistan" "Albania" "Algeria" "American Samoa" ...
#>  $ 1960   : num [1:209] 769308 494443 3293999 NA NA ...
#>  $ 1961   : num [1:209] 814923 511803 3515148 13660 8724 ...
#>  $ 1962   : num [1:209] 858522 529439 3739963 14166 9700 ...
#>  $ 1963   : num [1:209] 903914 547377 3973289 14759 10748 ...
#>  $ 1964   : num [1:209] 951226 565572 4220987 15396 11866 ...
#>  $ 1965   : num [1:209] 1000582 583983 4488176 16045 13053 ...
```

```
#>  $ 1966   : num [1:209] 1058743 602512 4649105 16693 14217 ...
pop_list <- list(pop_1, pop_2)
```

Import other data formats

The `haven` package enables R to read and write various data formats used by other statistical packages.

It supports:

- SAS: `read_sas()` reads .sas7bdat and .sas7bcat files and `read_xpt()` reads SAS transport files. write_sas() writes .sas7bdat files.
- SPSS: `read_sav()` reads .sav files and `read_por()` reads the older .por files. write_sav() writes .sav files.
- Stata: `read_dta()` reads .dta files. `write_dta()` writes .dta files.

0.1 Create and format dates

To create a Date object from a simple character string in R, you can use the as.Date() function. The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 13 January, 1982):

`%Y`: 4-digit year (1982) `%y`: 2-digit year (82) `%m`: 2-digit month (01) `%d`: 2-digit day of the month (13) `%A`: weekday (Wednesday) `%a`: abbreviated weekday (Wed) `%B`: month (January) `%b`: abbreviated month (Jan)

Exercise 5

Load the following data sets, available in the course material: - the Danish fire insurance losses, stored in `danish.txt` - the severity data set, stored in `severity.sas7bdat`.

Part IV: Data Manipulation

Subsetting and filtering data

Subsetting and filtering data involve selecting specific elements, rows, or columns from a dataset based on certain conditions or criteria. In R, subsetting can be achieved using square brackets `[]`, the `subset()` function, or dplyr package functions like `filter()` for rows and `select()` for columns. Filtering refers more specifically to choosing rows that meet certain conditions, such as values within a range or matching specific characteristics.

```
# Creating a sample data frame
data <- data.frame(
  id = 1:5,
  name = c("Alice", "Bob", "Charlie", "David", "Eva"),
  age = c(25, 30, 22, 28, 24)
)
# Subsetting by a specific column
ages <- data$age
print(ages)
#> [1] 25 30 22 28 24

# Filtering data based on a condition
young_adults <- subset(data, age < 30)
print(young_adults)
#>   id    name age
#> 1  1   Alice  25
#> 3  3 Charlie  22
#> 4  4   David  28
#> 5  5     Eva  24
```

Using the dplyr package

The `%>%` symbol in R is known as the pipe operator, and it's used to pass the result of one expression as the first argument to the next expression

1. Filtering data using dplyr for individuals younger than 30

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
young_adults <- data %>% filter(age < 30)
print(young_adults)
#>   id    name age
#> 1  1   Alice  25
#> 2  3 Charlie  22
#> 3  4   David  28
#> 4  5     Eva  24
```

2. Subsetting columns using dplyr

```
ages <- data %>% select(age)
print(ages)
#>   age
#> 1  25
#> 2  30
#> 3  22
#> 4  28
#> 5  24
```

Adding, removing, and renaming columns

1. Adding a new column 'salary'

```
data$salary <- c(55000, 50000, 60000, 52000, 58000)
print(data)
#>   id    name age salary
#> 1  1   Alice  25  55000
#> 2  2     Bob  30  50000
#> 3  3 Charlie  22  60000
#> 4  4   David  28  52000
#> 5  5     Eva  24  58000
```

2. Removing the 'salary' column

```
data$salary <- NULL
print(data)
```

```
#>   id    name age
#> 1  1   Alice  25
#> 2  2     Bob  30
#> 3  3 Charlie  22
#> 4  4   David  28
#> 5  5     Eva  24
```

3. Renaming the 'name' column to 'first_name'

```
names(data)[names(data) == "name"] <- "first_name"
print(data)
#>   id first_name age
#> 1  1      Alice  25
#> 2  2        Bob  30
#> 3  3    Charlie  22
#> 4  4      David  28
#> 5  5        Eva  24
```

Using the dplyr package

1. Adding a new column 'salary' using mutate

```
data <- data %>%
  mutate(salary = c(55000, 50000, 60000, 52000, 58000))
print(data)
#>   id    name age salary
#> 1  1   Alice  25  55000
#> 2  2     Bob  30  50000
#> 3  3 Charlie  22  60000
#> 4  4   David  28  52000
#> 5  5     Eva  24  58000
```

2. Removing the 'salary' column using select

```
data <- data %>%
  select(-salary)
print(data)
#>   id    name age
#> 1  1   Alice  25
#> 2  2     Bob  30
#> 3  3 Charlie  22
#> 4  4   David  28
#> 5  5     Eva  24
```

3. Renaming the 'name' column to 'first_name' using rename

```r
data <- data %>%
  rename(first_name = name)
print(data)
#>   id first_name age
#> 1  1      Alice  25
#> 2  2        Bob  30
#> 3  3    Charlie  22
#> 4  4      David  28
#> 5  5        Eva  24
```

Why use dyplr

One might think that using pipe operator (`%>%`) from the `magrittr` package, prominently used in dplyr and the wider tidyverse is unnecessarily more complex. While it may seem more complex at first, especially to those accustomed to base R functions and syntax, it offers several benefits that can greatly enhance the readability, efficiency, and overall workflow of data analysis. Some reasons to use it are:

1. Improved Readability and Clarity
2. Easier Debugging and Modification
3. Enhanced Workflow
4. Consistency and Community Adoption
5. Efficiency in Writing Code

An example of the benefits is seen in more complex operations making them more readable

Let's just add salary column again.

```r
data$salary <- c(55000, 50000, 60000, 52000, 58000)
subsetting_data <- within(data[data$age < 30, -which(names(data) == "salary")], names(
subsetting_data
#>   id    name age
#> 1  1   Alice  25
#> 3  3 Charlie  22
#> 4  4   David  28
#> 5  5     Eva  24
```

Now, doing it using dplyr

```r
data <- data %>%
  mutate(salary = c(55000, 50000, 60000, 52000, 58000))

data <- data %>%
  filter(age < 30) %>%
```

```
  select(-salary) %>%
  rename(first_name = name)
data
#>   id first_name age
#> 1  1      Alice  25
#> 2  3    Charlie  22
#> 3  4      David  28
#> 4  5        Eva  24
```

Without using the pipe operator , it looks not so clear.

```
data <- data.frame(
  id = 1:5,
  name = c("Alice", "Bob", "Charlie", "David", "Eva"),
  age = c(25, 30, 22, 28, 24)
)
data <- data %>%
  mutate(salary = c(55000, 50000, 60000, 52000, 58000))


subsetting_data <- rename(select(
                          filter(data, age < 30), -salary),
                          first_name = name)
subsetting_data
#>   id first_name age
#> 1  1      Alice  25
#> 2  3    Charlie  22
#> 3  4      David  28
#> 4  5        Eva  24
```

Basic data summary and exploration

A very brief summary and data exploration is given below.

1. Summary statistics of the data frame

```
summary(data)
#>        id        name                age
#>  Min.   :1   Length:5           Min.   :22.0
#>  1st Qu.:2   Class :character   1st Qu.:24.0
#>  Median :3   Mode  :character   Median :25.0
#>  Mean   :3                      Mean   :25.8
#>  3rd Qu.:4                      3rd Qu.:28.0
#>  Max.   :5                      Max.   :30.0
#>      salary
#>  Min.   :50000
```

```
#>  1st Qu.:52000
#>  Median :55000
#>  Mean   :55000
#>  3rd Qu.:58000
#>  Max.   :60000
```

2. Structure of the data frame

```
str(data)
#> 'data.frame':    5 obs. of  4 variables:
#>  $ id    : int  1 2 3 4 5
#>  $ name  : chr  "Alice" "Bob" "Charlie" "David" ...
#>  $ age   : num  25 30 22 28 24
#>  $ salary: num  55000 50000 60000 52000 58000
```

3. Average age of the individuals in the data frame

```
average_age <- mean(data$age)
print(average_age)
#> [1] 25.8
```

4. Count of unique names in the data frame

```
unique_names_count <- length(unique(data$first_name))
print(unique_names_count)
#> [1] 0
```

Exploratory data analysis

Exploratory Data Analysis (EDA) is a critical initial step in the data analysis process, where the main characteristics of a dataset are examined to understand its structure, uncover patterns, identify anomalies, and test hypotheses. The goal is to use statistical summaries and visualizations to get a sense of the data, which guides further analysis and modeling decisions. EDA is not about making formal predictions or testing hypotheses but rather about asking questions and seeking insights in a more open-ended, exploratory manner.

Key Components of EDA include:

- **Understanding the Distribution** of various variables in the dataset. This involves looking at measures like mean, median, mode, range, variance, and standard deviation, and using visual tools like histograms, box plots, and density plots to understand how the data is spread out.
- **Identifying Patterns** and Relationships between variables using scatter plots, pair plots, and correlation matrices. This helps in understanding how variables are related to each other and can guide more complex analyses like regression or classification.

- **Detecting Anomalies** such as outliers or unexpected values which might indicate errors in data collection or provide insights into unusual occurrences in the data.
- **Cleaning Data** by addressing missing values, duplicate data, and making decisions about how to correct inconsistencies based on the insights gained.
- **Transforming Variables** when necessary to make the data more suitable for analysis. This could involve normalizing the data, creating categorical variables from continuous ones, or engineering new variables from existing ones.

Tools and Techniques

Statistical Summary Functions in R (`summary()`, `mean()`, `sd()`, etc.) provide quick insights into the basic properties of the data.

Visualization Libraries like `ggplot2` in R for creating a wide range of plots and charts that reveal the underlying patterns and structures in the data.

Importance of EDA

- **Data Understanding:** It ensures that the analyst has a thorough understanding of the dataset's features, values, and relationships between variables.
- **Guiding Hypotheses:** Insights gained during EDA can help form hypotheses for statistical testing and predictive modeling.
- **Modeling Strategy:** Identifying the key variables and their relationships helps in choosing appropriate models and techniques for further analysis.

In summary, EDA is an essential practice in data science for making sense of data, discovering patterns, identifying potential problems, and informing subsequent steps in the analytical process. It blends statistical techniques with visual explorations to create a foundation for any data-driven task.

Now we will explore a dataset from the library `AER` , and a numeric variable first

Numeric Variable

```
#install.packages("AER")
library(AER)
#> Loading required package: car
#> Loading required package: carData
#>
#> Attaching package: 'car'
#> The following object is masked from 'package:dplyr':
#>
#>     recode
```

```
#> Loading required package: lmtest
#> Loading required package: zoo
#>
#> Attaching package: 'zoo'
#> The following objects are masked from 'package:base':
#>
#>     as.Date, as.Date.numeric
#> Loading required package: sandwich
#> Loading required package: survival
data("CPS1985")
str(CPS1985)
#> 'data.frame':    534 obs. of  11 variables:
#>  $ wage      : num  5.1 4.95 6.67 4 7.5 ...
#>  $ education : num  8 9 12 12 12 13 10 12 16 12 ...
#>  $ experience: num  21 42 1 4 17 9 27 9 11 9 ...
#>  $ age       : num  35 57 19 22 35 28 43 27 33 27 ...
#>  $ ethnicity : Factor w/ 3 levels "cauc","hispanic",..: 2 1 1 1 1 1 1 1 1 1 ...
#>  $ region    : Factor w/ 2 levels "south","other": 2 2 2 2 2 2 1 2 2 2 ...
#>  $ gender    : Factor w/ 2 levels "male","female": 2 2 1 1 1 1 1 1 1 1 ...
#>  $ occupation: Factor w/ 6 levels "worker","technical",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ sector    : Factor w/ 3 levels "manufacturing",..: 1 1 1 3 3 3 3 3 1 3 ...
#>  $ union     : Factor w/ 2 levels "no","yes": 1 1 1 1 1 2 1 1 1 1 ...
#>  $ married   : Factor w/ 2 levels "no","yes": 2 2 1 1 2 1 1 1 2 1 ...
```

```
head(CPS1985)
#>       wage education experience age ethnicity region gender
#> 1     5.10         8         21  35  hispanic  other female
#> 1100  4.95         9         42  57      cauc  other female
#> 2     6.67        12          1  19      cauc  other   male
#> 3     4.00        12          4  22      cauc  other   male
#> 4     7.50        12         17  35      cauc  other   male
#> 5    13.07        13          9  28      cauc  other   male
#>      occupation        sector union married
#> 1        worker manufacturing    no     yes
#> 1100     worker manufacturing    no     yes
#> 2        worker manufacturing    no      no
#> 3        worker         other    no      no
#> 4        worker         other    no     yes
#> 5        worker         other   yes      no
```

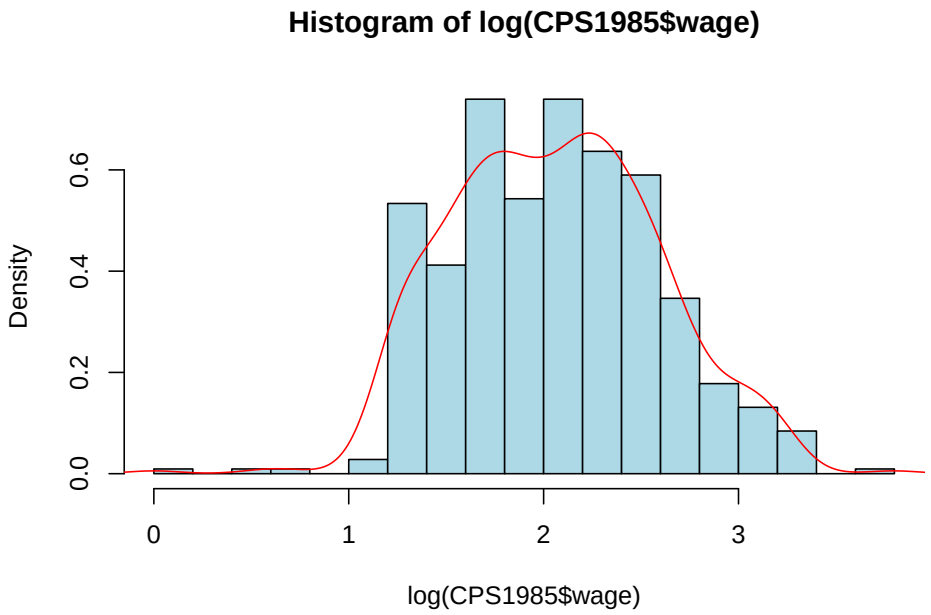
Obtain the summary statistics of the data frame, check whether it is numeric, get the mean , and variance.

```
summary(CPS1985$wage)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   1.000   5.250   7.780   9.024  11.250  44.500
```

```
is.numeric(CPS1985$wage)
#> [1] TRUE
mean(CPS1985$wage)
#> [1] 9.024064
var(CPS1985$wage)
#> [1] 26.41032
```

Now, visualize the wage distribution

```
hist(log(CPS1985$wage), freq = FALSE, nclass = 20, col = "light blue")
lines(density(log(CPS1985$wage)), col = "red")
```



A factor variable

You now explore the occupation variable

```
summary(CPS1985$occupation)
#>     worker  technical   services     office      sales
#>        156        105         83         97         38
#> management
#>         55
```

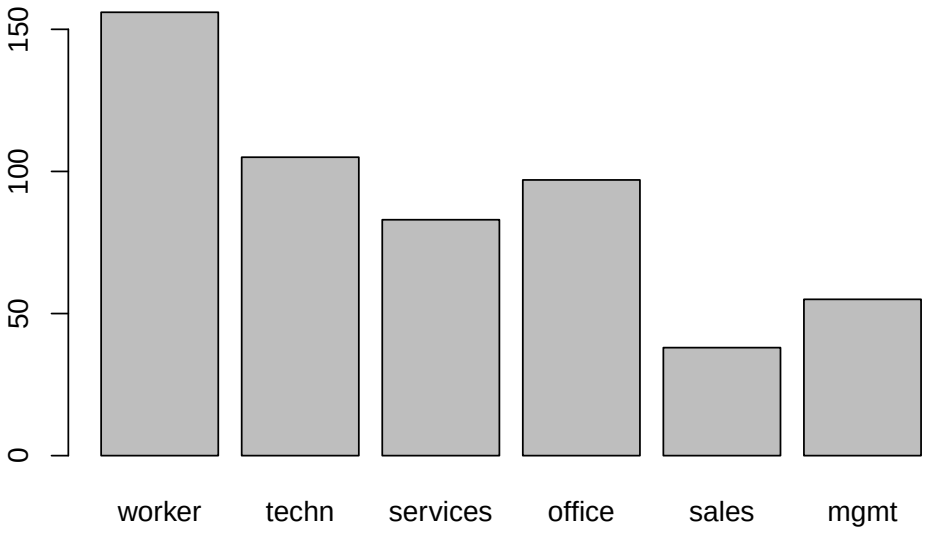
change the names of some of the levels

```
levels(CPS1985$occupation)[c(2, 6)] <- c("techn", "mgmt")
summary(CPS1985$occupation)
#>   worker    techn services   office    sales     mgmt
```

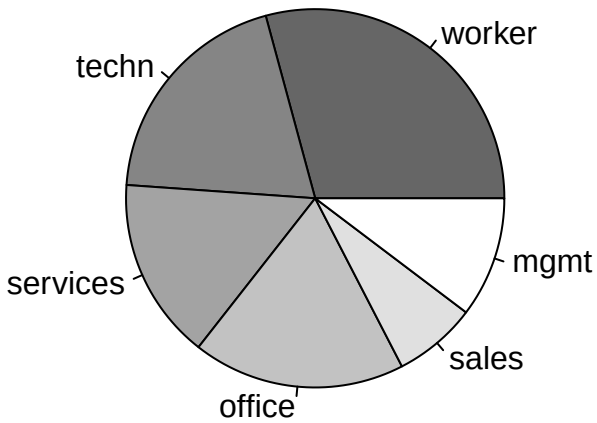
```
#>      156      105       83       97       38       55
```

visualize the distribution

```
tab <- table(CPS1985$occupation)
prop.table(tab)
#>
#>     worker      techn   services     office      sales
#> 0.29213483 0.19662921 0.15543071 0.18164794 0.07116105
#>       mgmt
#> 0.10299625
barplot(tab)
```




```
pie(tab, col = gray(seq(0.4, 1.0, length = 6)))
```



Two factor variables

You now explore the factor variables `gender` and `occupation`.

Use `prop.table()`

The `prop.table()` function in R is used to compute the proportion of table elements over the margin specified (if any). When applied to a contingency table created by the table() function, it transforms the table's counts into proportions, making it easier to analyze the relative distribution of frequencies across different categories.

```
attach(CPS1985) # attach the data set to avoid use the operator $
table(gender, occupation) # no name_df$name_var necessary
#>          occupation
#> gender    worker techn services office sales mgmt
#>   male       126    53       34     21    21   34
#>   female      30    52       49     76    17   21
```

```
prop.table(table(gender, occupation))
#>          occupation
#> gender         worker      techn   services     office
#>   male   0.23595506 0.09925094 0.06367041 0.03932584
#>   female 0.05617978 0.09737828 0.09176030 0.14232210
#>          occupation
#> gender         sales       mgmt
#>   male   0.03932584 0.06367041
#>   female 0.03183521 0.03932584
```

Now try `prop.table(table(gender, occupation), 2)`

```
prop.table(table(gender, occupation), 2) # 1 for row , 2 for columns
#>          occupation
#> gender       worker     techn  services    office     sales
#>   male   0.8076923 0.5047619 0.4096386 0.2164948 0.5526316
#>   female 0.1923077 0.4952381 0.5903614 0.7835052 0.4473684
#>          occupation
#> gender         mgmt
#>   male   0.6181818
#>   female 0.3818182
```

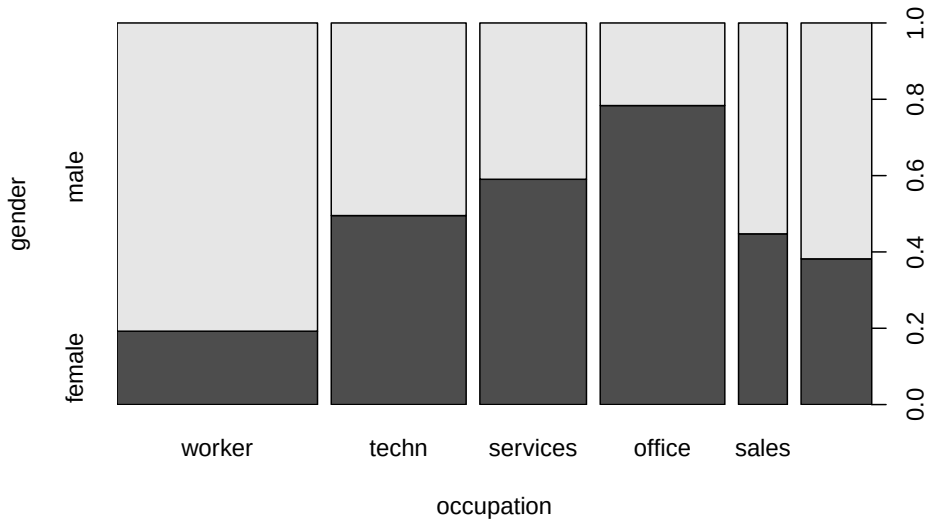
You now explore the factor variables gender and occupation.

Do a mosaic plot

A mosaic plot, also known as a Marimekko diagram or a mosaic chart, is a graphical representation of data that allows for the visualization of the proportions or frequencies of categorical variables in a dataset. It's a type of plot that provides a visual summary of the contingency table associated with two or

more categorical variables. Each rectangle (tile) in the mosaic plot represents a combination of category levels from the variables, with the area of the rectangle proportional to the frequency or proportion of observations in that category combination.

```
plot(gender ~ occupation, data = CPS1985)
```



Now explore the factor gender and the numeric variable wage.

The `tapply()` function in R is used to apply a function to subsets of a vector, where subsets are defined by some other vector, usually a factor. The basic syntax of tapply() is:

- X is the object to be split and operated on.
- INDEX is the factor or list of factors according to which X is split.
- FUN is the function to be applied to each subset of X.
- … are optional arguments to FUN.
- simplify, when TRUE, tries to simplify the result to a vector, matrix, or higher-dimensional array; when FALSE, the result is a list.

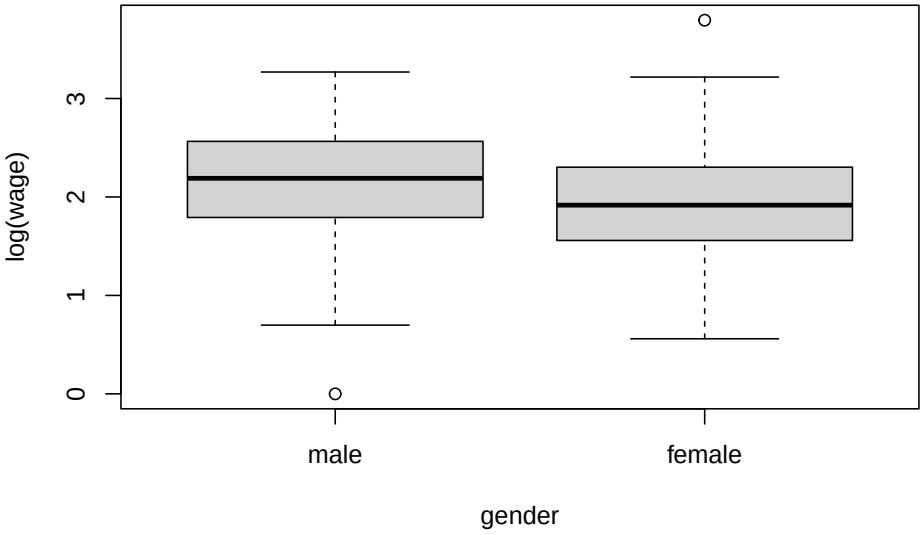
```
tapply(wage, gender, mean)
#>     male   female
#> 9.994913 7.878857
```

```
tapply(log(wage), list(gender, occupation), mean)
#>          worker    techn services   office    sales
#> male   2.100418 2.446640 1.829568 1.955284 2.141071
#> female 1.667887 2.307509 1.701674 1.931128 1.579409
#>            mgmt
#> male   2.447476
#> female 2.229256
```

A factor and a numeric variable

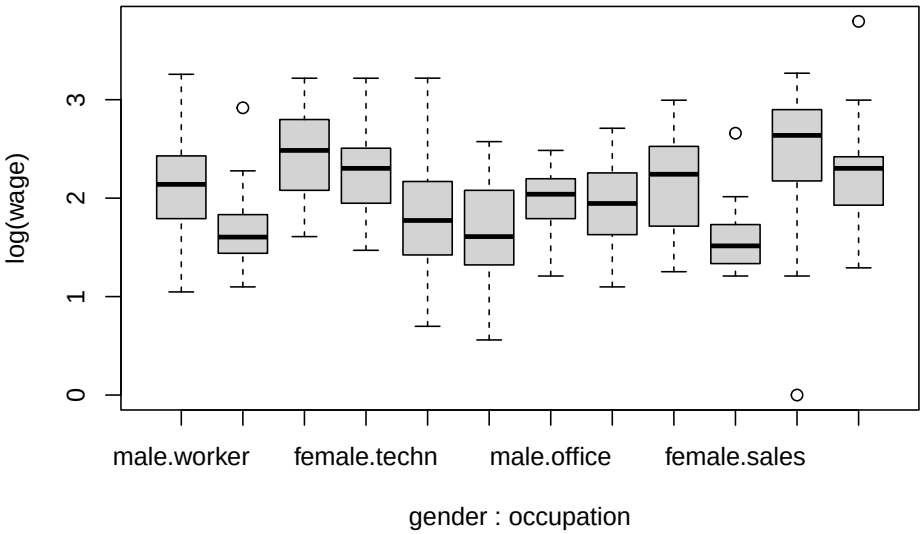
Explore a factor variable and a numeric variable. Visualize the distribution of `wage` per `gender`

```
boxplot(log(wage) ~ gender, data = CPS1985)
```



Now try with

```
boxplot(log(wage) ~ gender + occupation, data = CPS1985)
```




```
detach(CPS1985) # now detach when work is done
```

Exercises

1. Subsetting Data Frames

Create a data frame named student_info with the following columns and data: - student_id (1 to 5) - student_name ('Alice', 'Bob', 'Charlie', 'David', 'Eva') - student_age (25, 30, 22, 28, 24) - student_grade ('A', 'B', 'A', 'C', 'B')

Write a command to subset this data frame to include only students older than 24.

2. Using Conditional Filters

- Use the subset() function to find all students with a grade of 'A'.
- Display the names and ages of these students.

3. Manipulating Data with dplyr

- Load the dplyr package and convert student_info to a tibble.
- Use filter() and select() to show the name and age of students who have a grade better than 'B'.

4. Adding and Removing Columns

- Add a new column student_major with values ('Math', 'Science', 'Arts', 'Math', 'Science') to student_info.
- Then, remove the student_grade column using dplyr.

5. Renaming Columns

- Rename the student_name column to name using base R functions and then using dplyr.

6. Complex dplyr Operations

- Create a new tibble from student_info that includes all students except those studying 'Arts', rename the student_id column to id, and arrange the students by age in descending order.

7. Exploratory Data Analysis with dplyr

- Calculate the average age of students grouped by their major using group_by() and summarize() in dplyr.

Part V: Basic Data Visualization

Creating simple plots using plot(), hist(), pie(),barplot(),boxplot()

In R, the `plot()` function is a generic function used for making a variety of graphs. At its simplest, it is used to create scatter plots but can be customized to create line plots, add model lines, and much more.

Basic Arguments:

- `x`: The coordinates of points in the plot. For a simple scatter plot, this is typically a numeric vector.
- `y`: The coordinates of points in the plot on the y-axis. Should be the same length as x.
- `type`: What type of plot should be drawn. Possible types include "p" for points (the default), "l" for lines, "b" for both, and several others.
- `main`: The main title of the plot.
- `xlab`: The label for the x-axis.
- `ylab`: The label for the y-axis.
- `xlim`: Limits for the x-axis.
- `ylim`: Limits for the y-axis.
- `pch`: Plotting character, or symbol to use in the plot. Different numbers correspond to different symbols.
- `col`: Color for the points. Can also be a vector to color points differently based on a factor.

Additional Customizations:

- `cex`: A numerical value giving the amount by which plotting text and symbols should be magnified relative to the default.
- `lwd`: Line width for the plot, useful when the plot type includes lines.

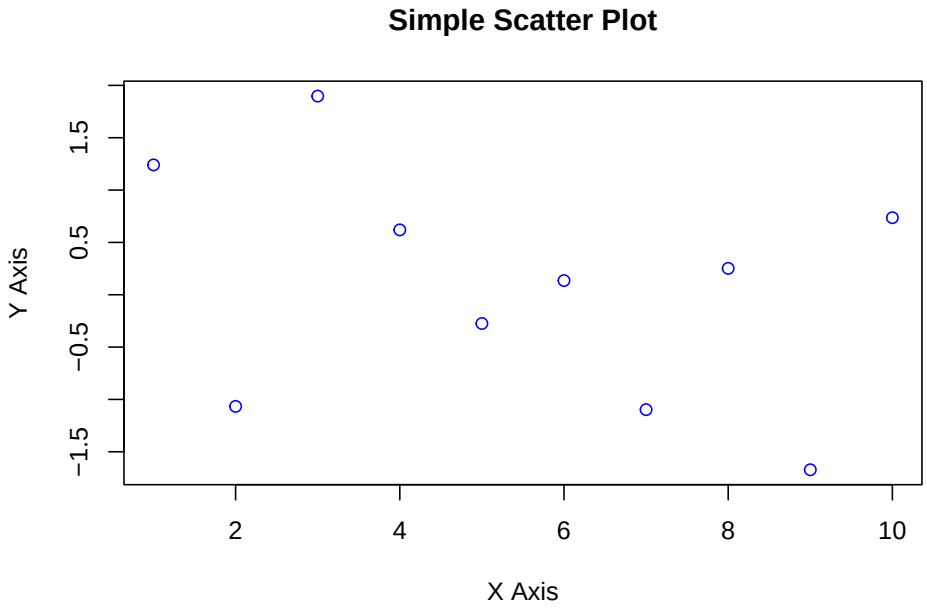
- `bg`: Background color for the open plot symbols specified by pch. Advanced Features:
- `abline`: A function to add straight lines to a plot, either vertical, horizontal, or regression lines.
- `lines`: A function to add lines to a plot, in the context of the existing plot; it doesn't start a new plot. points: Add points to a plot.

Adding a Legend:

To add a legend, you use the `legend()` function. It provides a number of arguments to customize its appearance:

`legend`: A vector of text values or an expression describing the text to appear in the legend. `x, y` or `position`: The location of the legend. `x` and `y` can be numeric positions, or you can use keyword positions like `"topright"`, `"bottomleft"`, `"bottomright"` , `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"right"`, `"center"`. `pch`: The plotting symbols for points appearing in the legend, matching those in the plot. `col`: The colors for points or lines appearing in the legend, matching those in the plot. `lwd`: The line widths for lines appearing in the legend, matching those in the plot. `cex`: Character expansion size for the legend, determining how large the text in the legend should be.

```
x <- 1:10
y <- rnorm(10)
plot(x, y, main = "Simple Scatter Plot", xlab = "X Axis", ylab = "Y Axis", col = "blue
```



Exercise 1

Making a Scatter plot:

- load the journals.txt data set and save as Journals data frame
- Work through the following instructions

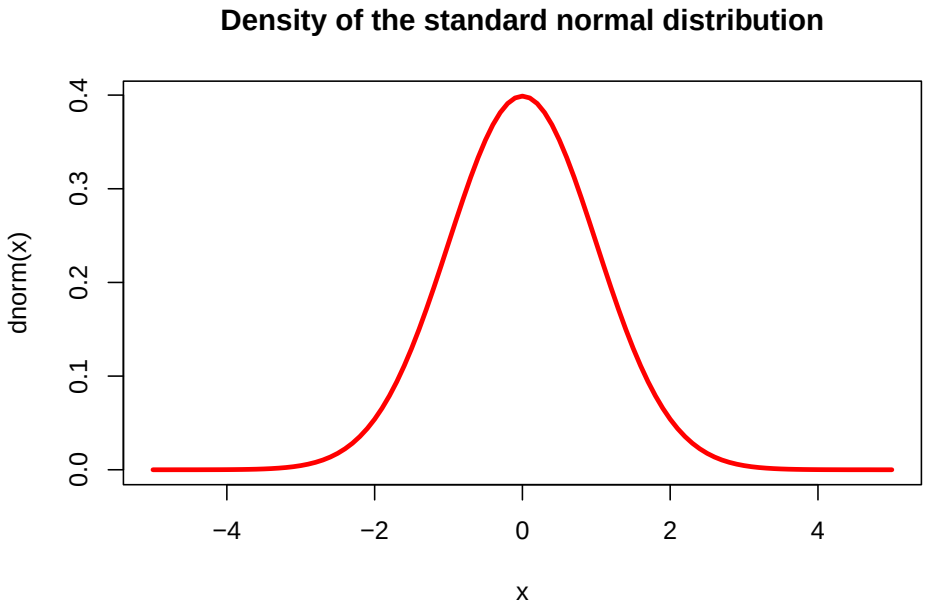
```
plot(log(Journals$subs), log(Journals$price))
rug(log(Journals$subs))
rug(log(Journals$price), side = 2)
```

Now adjust the plotting instructions

```
plot(log(Journals$price) ~ log(Journals$subs), pch = 19,
     col = "blue", xlim = c(0, 7), ylim = c(3, 8),
     main = "Library subscriptions")
rug(log(Journals$subs))
rug(log(Journals$price), side=2)
```

The `curve()` function draws a curve corresponding to a function over the interval [from, to].

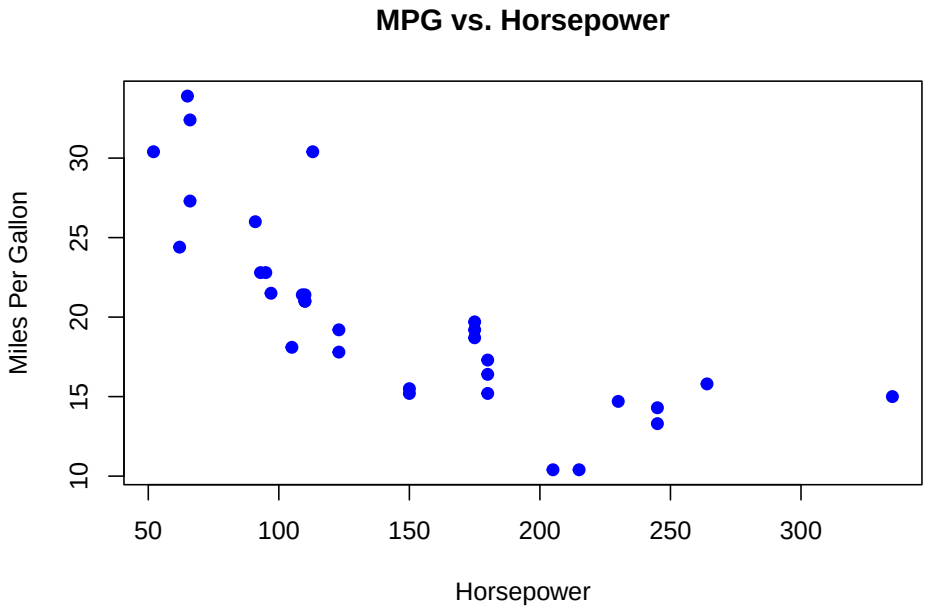
```
curve(dnorm, from = -5, to = 5, col = "red", lwd = 3,
      main = "Density of the standard normal distribution")
```



Example 2

To create a basic scatter plot, you can use the mtcars dataset, which comes built into R. This dataset contains various characteristics of 32 automobiles.

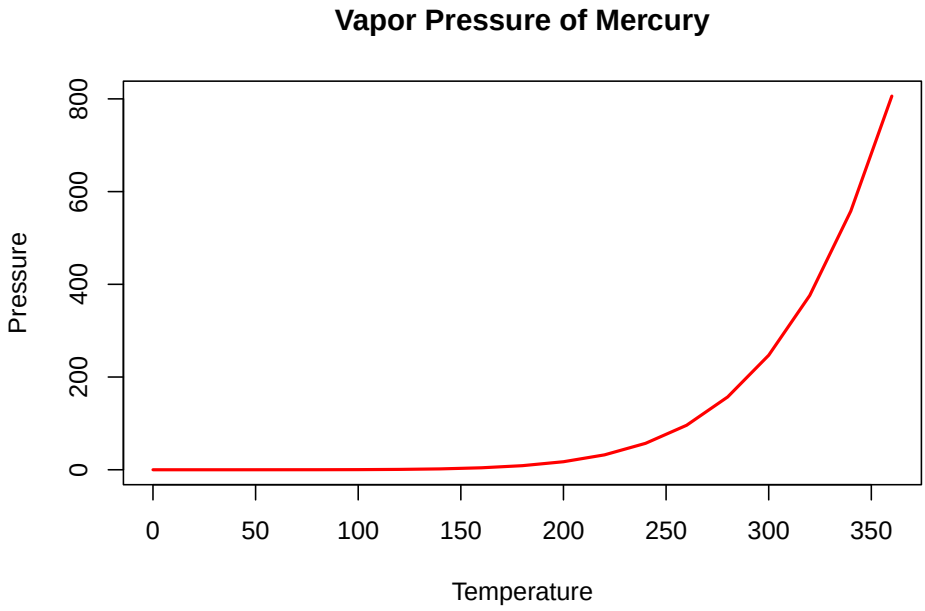
```
data(mtcars)


plot(mtcars$hp, mtcars$mpg, main="MPG vs. Horsepower",
     xlab="Horsepower", ylab="Miles Per Gallon",
     pch=19, col="blue")
```



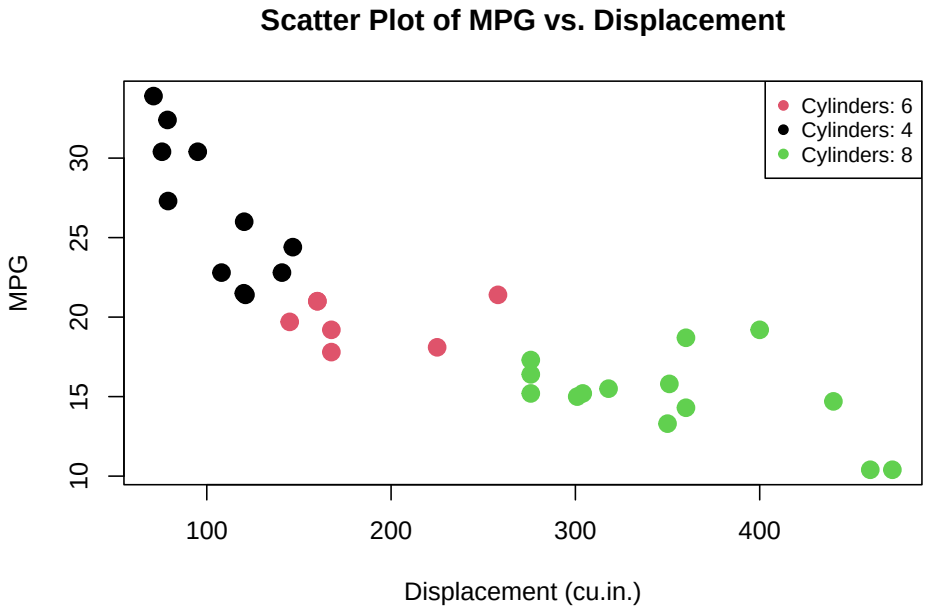
Example 3

Using the pressure dataset, which is also built into R, you can create a simple line plot. The pressure dataset shows the temperature and resulting vapor pressure of mercury.

```
data(pressure)

# Create a line plot
plot(pressure$temperature, pressure$pressure, type="l",
     main="Vapor Pressure of Mercury",
     xlab="Temperature", ylab="Pressure",
     col="red", lwd=2)
```




```r
# Load the mtcars dataset
data(mtcars)

# Plot MPG vs. Displacement, colored by Cylinders
plot(mtcars$disp, mtcars$mpg, col=as.factor(mtcars$cyl),
     main="Scatter Plot of MPG vs. Displacement",
     xlab="Displacement (cu.in.)", ylab="MPG",
     pch=19, cex=1.5)

# Add a legend to the plot
legend("topright",
       legend=paste("Cylinders:", unique(mtcars$cyl)),
       col=unique(as.numeric(as.factor(mtcars$cyl))),
       pch=19, cex=0.8)
```



- `plot()` is the generic function to create a scatter plot. mtcars$dispand$mtcars$mpg are the x and y coordinates for the plot, representing the engine displacement in cubic inches and miles per gallon, respectively.
- `col=as.factor(mtcars$cyl)` specifies the colors for the points on the plot. The cyl variable, which represents the number of cylinders in the car's engine, is converted into a factor. The levels of this factor (unique values of cyl) are automatically given different colors.
- `main` is the main title for the plot.
- `xlab` and `ylab` are labels for the x-axis and y-axis, respectively.
- `pch=19` specifies the plotting symbol (in this case, a solid circle).
- `cex=1.5` sets the size of the plot symbols; cex stands for character expansion factor, where 1.5 means 150% of the default size.

For the legend,

- `legend()` adds a legend to the plot.
- `"topright"` specifies the position of the legend (in this case, at the top right corner of the plotting area).
- `legend=` creates the text for the legend by pasting the word "Cylinders:" in front of each unique value of the cyl column. This indicates what each color in the scatter plot corresponds to.
- `col=` sets the colors used in the legend, which match the colors used for the points in the plot.
- `pch=19` again specifies the plotting symbols used in the legend.
- `cex=0.8` sets the size of the symbols in the legend.

Example 5

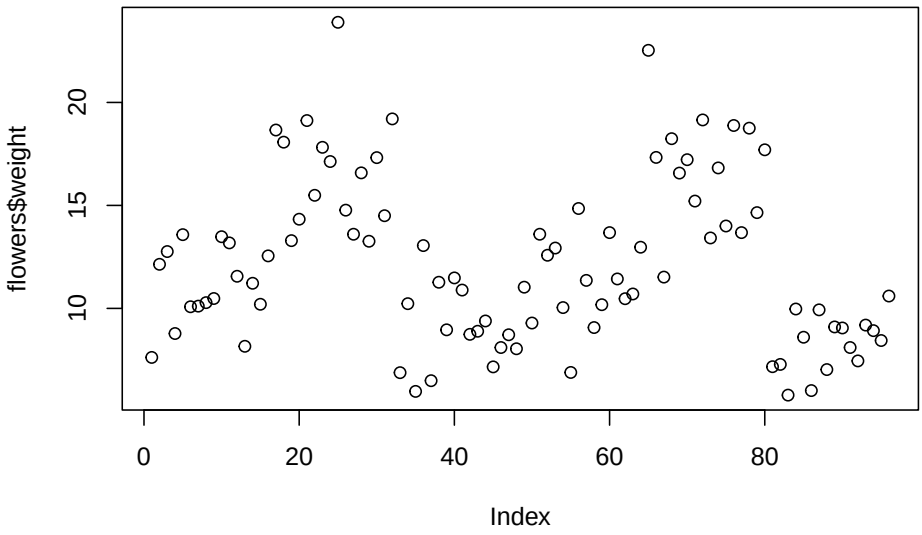
Scatterplot()

The most common high level function used to produce plots in `R` is (rather unsurprisingly) the `plot()` function. For example, let's plot the weight of petunia plants from our flowers data frame from `flower.xls`

Use the library `readxl`, and `read_excel()` function.

Alternatively , you can also use `flower.txt` and use read_table

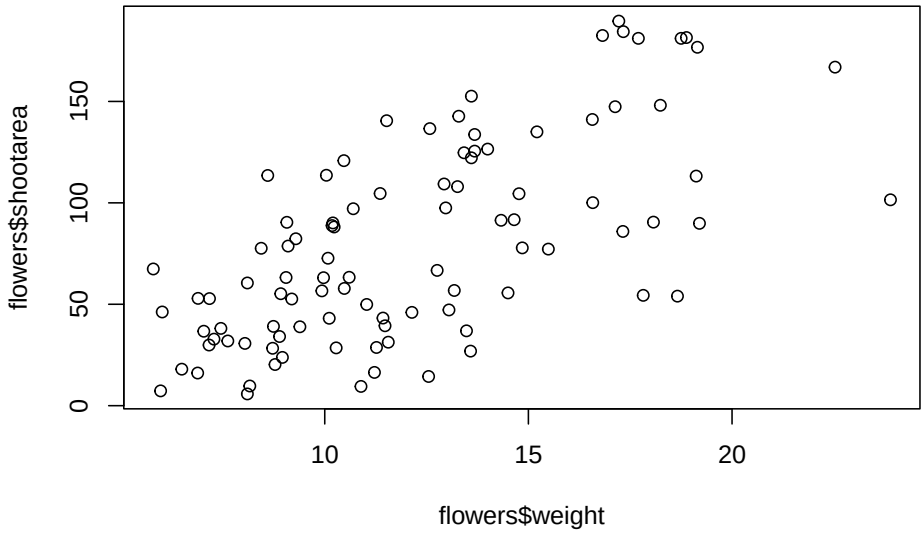
```
library(readxl)
flowers <- read_excel('./Data/flower.xls')

plot(flowers$weight)
```




```
flowers <- read.table(file = './Data/flower.txt',
                      header = TRUE, sep = "\t",
                      stringsAsFactors = TRUE)
```

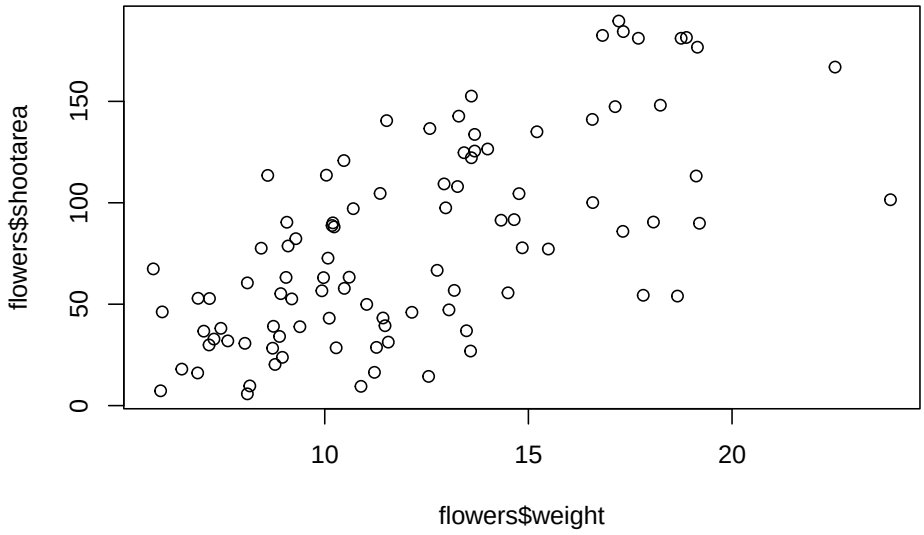
To plot a scatterplot of one numeric variable against another numeric variable we just need to include both variables as arguments when using the `plot()` function. For example to plot `shootarea` on the y axis and `weight` of the x axis.

```
plot(x = flowers$weight, y = flowers$shootarea)
```

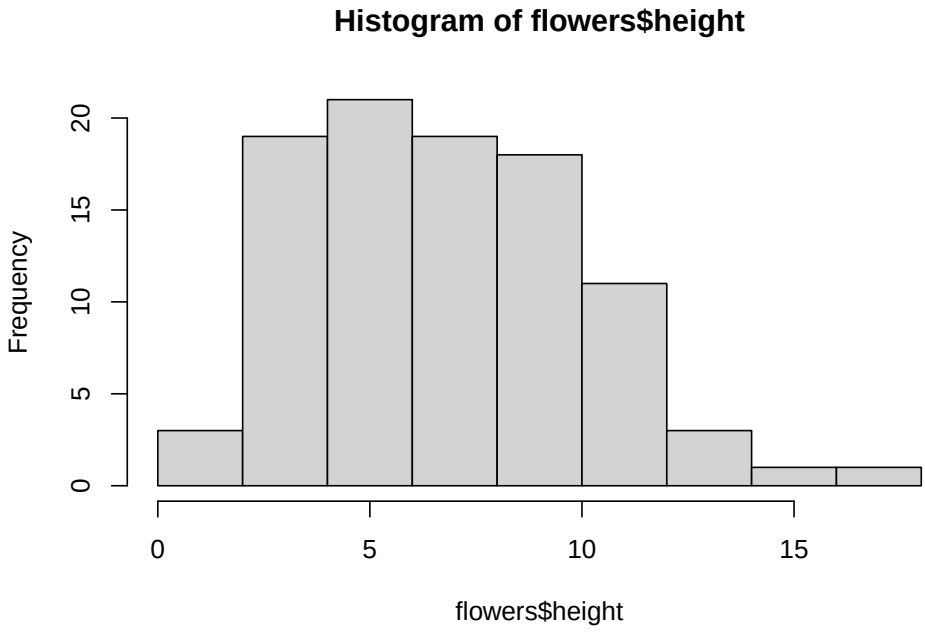


There is an equivalent approach for these types of plots which often causes some confusion at first. You can also use the formula notation when using the `plot()` function. However, in contrast to the previous method the formula method requires you to specify the y axis variable first, then a ~and then our x axis variable.

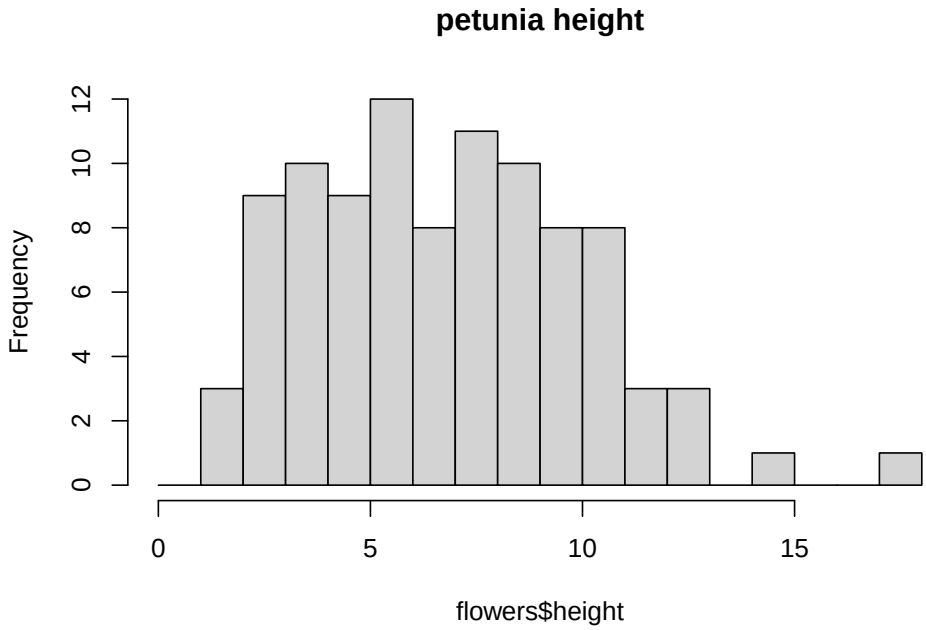
```
plot(flowers$shootarea ~ flowers$weight)
```



Histogram

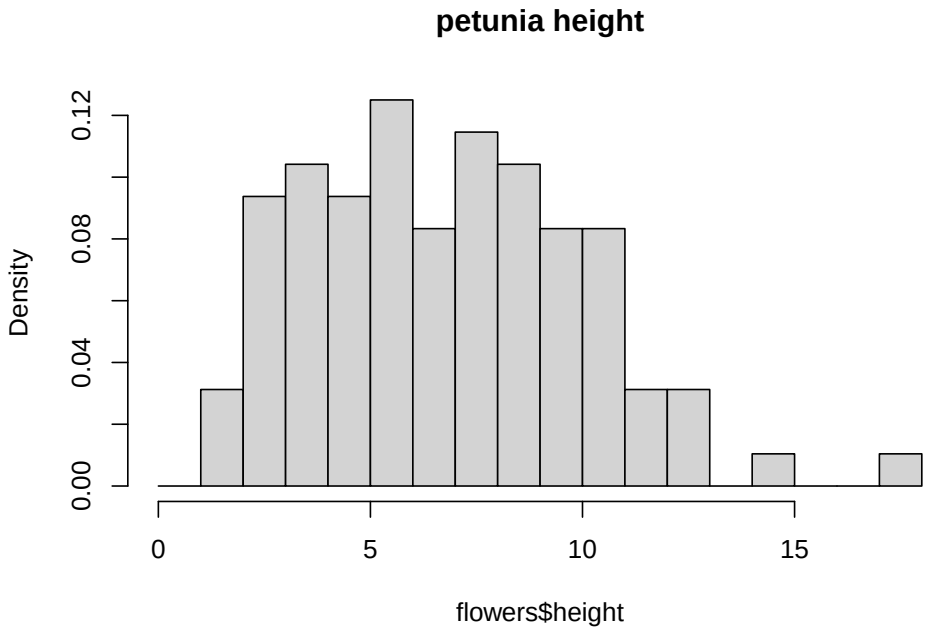



```
brk <- seq(from = 0, to = 18, by = 1)
hist(flowers$height, breaks = brk, main = "petunia height")
```



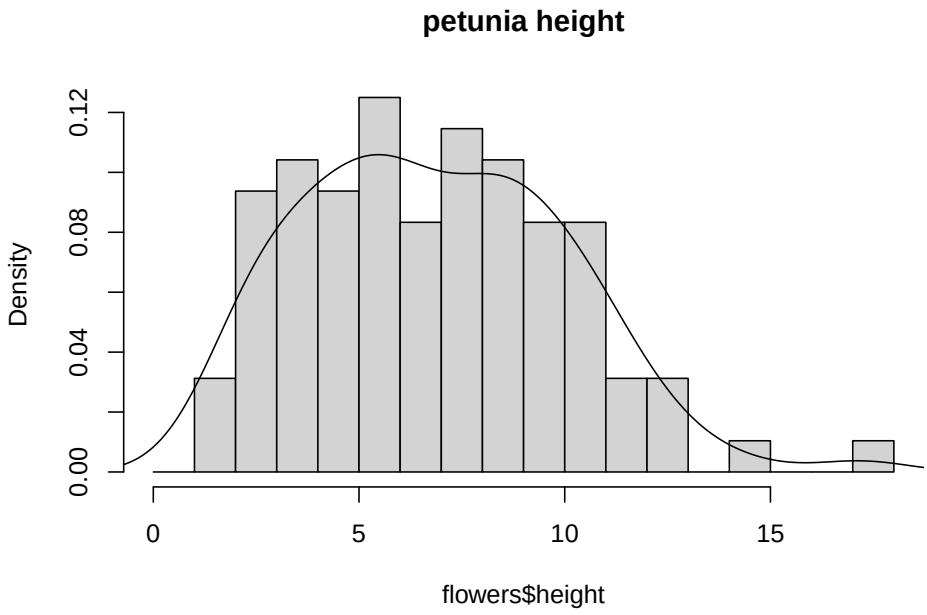
You can also display the histogram as a proportion rather than a frequency by using the freq = FALSE argument.

```
brk <- seq(from = 0, to = 18, by = 1)
hist(flowers$height, breaks = brk, main = "petunia height",
      freq = FALSE)
```



An alternative to plotting just a straight up histogram is to add a kernel density curve to the plot. You can superimpose a density curve onto the histogram by first using the density() function to compute the kernel density estimates and then use the low level function lines() to add these estimates onto the plot as a line.

```
dens <- density(flowers$height)
hist(flowers$height, breaks = brk, main = "petunia height",
      freq = FALSE)
lines(dens)
```



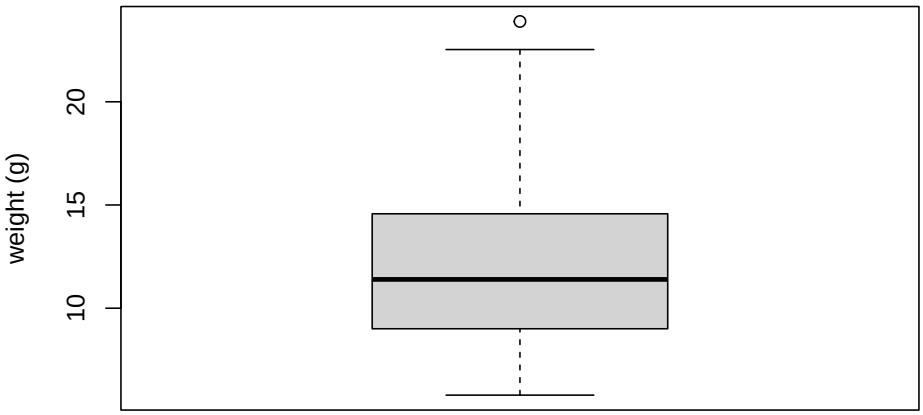
Boxplot

OK, we'll just come and out and say it, we love boxplots and their close relation the violin plot. Boxplots (or box-and-whisker plots to give them their full name) are very useful when you want to graphically summarise the distribution of a variable, identify potential unusual values and compare distributions between different groups. The reason we love them is their ease of interpretation, transparency and relatively high data-to-ink ratio (i.e. they convey lots of information efficiently). We suggest that you try to use boxplots as much as possible when exploring your data and avoid the temptation to use the more ubiquitous bar plot (even with standard error or 95% confidence intervals bars). The problem with bar plots (aka dynamite plots) is that they hide important information from the reader such as the distribution of the data and assume that the error bars (or confidence intervals) are symmetric around the mean. Of course, it's up to you what you do but if you're tempted to use bar plots just

Google 'dynamite plots are evil' see here or here

To create a boxplot in R we use the boxplot() function. For example, let's create a boxplot of the variable weight from our flowers data frame. We can also include a y axis label using the ylab = argument.

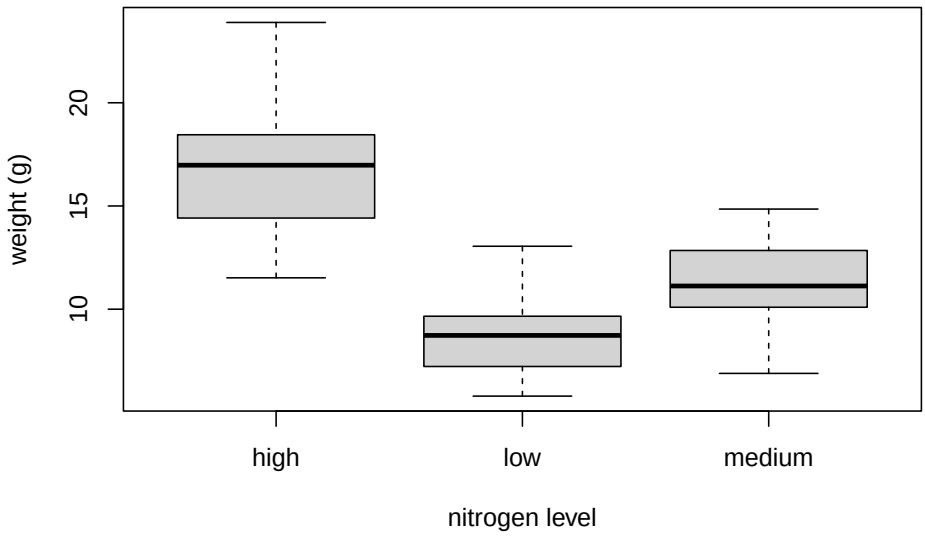
```
boxplot(flowers$weight, ylab = "weight (g)")
```



The thick horizontal line in the middle of the box is the median value of weight (around 11 g). The upper line of the box is the upper quartile (75th percentile) and the lower line is the lower quartile (25th percentile). The distance between the upper and lower quartiles is known as the inter quartile range and represents the values of weight for 50% of the data. The dotted vertical lines are called the whiskers and their length is determined as 1.5 x the inter quartile range. Data points that are plotted outside the the whiskers represent potential unusual observations. This doesn't mean they are unusual, just that they warrant a closer look. We recommend using boxplots in combination with Cleveland dotplots to identify potential unusual observations (see the next section of this Chapter for more details). The neat thing about boxplots is that they not only provide a measure of central tendency (the median value) they also give you an idea about the distribution of the data. If the median line is more or less in the middle of the box (between the upper and lower quartiles) and the whiskers are more or less the same length then you can be reasonably sure the distribution of your data is symmetrical.

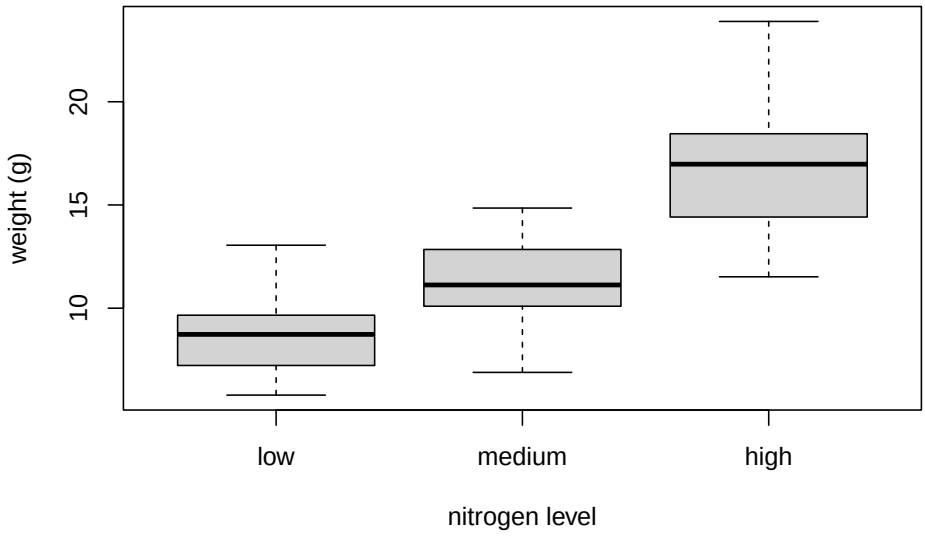
If we want examine how the distribution of a variable changes between different levels of a factor we need to use the formula notation with the boxplot() function. For example, let's plot our weight variable again, but this time see how this changes with each level of nitrogen. When we use the formula notation with `boxplot()` we can use the data = argument to save some typing. We'll also introduce an x axis label using the xlab = argument.

```
boxplot(weight ~ nitrogen, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level")
```



The factor levels are plotted in the same order defined by our factor variable nitrogen (often alphabetically). To change the order we need to change the order of our levels of the nitrogen factor in our data frame using the factor() function and then re-plot the graph. Let's plot our boxplot with our factor levels going from low to high.

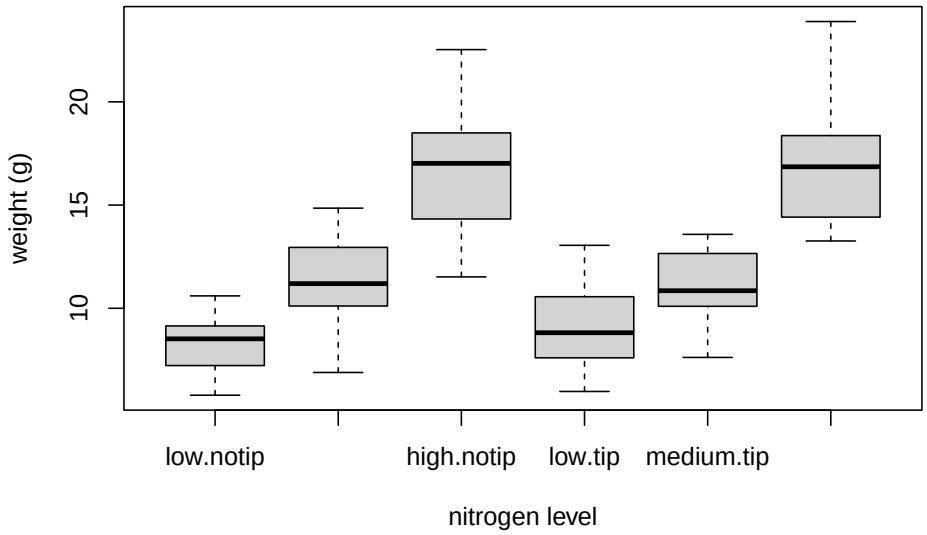
```
flowers$nitrogen <- factor(flowers$nitrogen,
                           levels = c("low", "medium", "high"))
boxplot(weight ~ nitrogen, data = flowers,
          ylab = "weight (g)", xlab = "nitrogen level")
```



We can also group our variables by two factors in the same plot. Let's plot

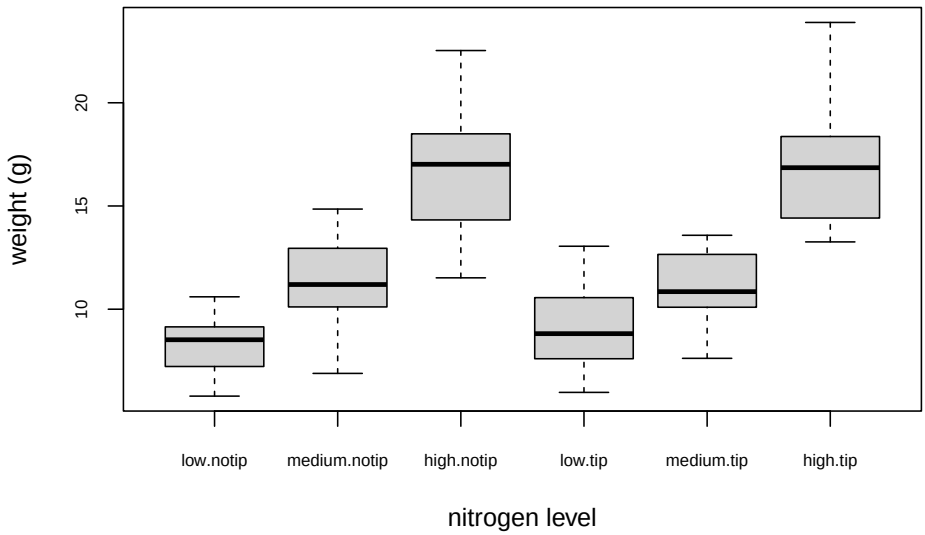
our weight variable but this time plot a separate box for each nitrogen and treatment (treat) combination.

```
boxplot(weight ~ nitrogen * treat, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level")
```



This plot looks OK, but some of the group labels are hidden as they're too long to fit on the plot. There are a couple of ways to deal with this. Perhaps the easiest is to reduce the font size of the tick mark labels in the plot so they all fit using the `cex.axis =` argument. Let's set the font size to be 30% smaller than the default with `cex.axis = 0.7`

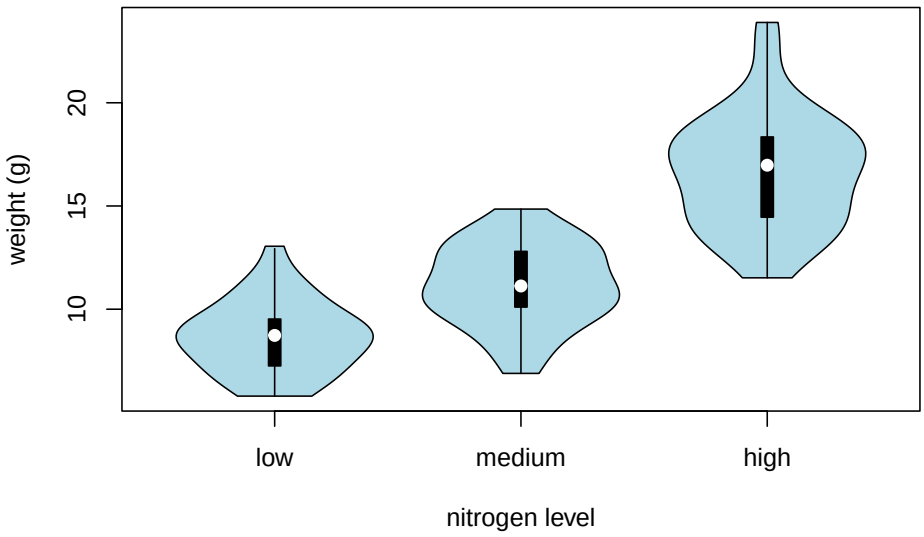
```
boxplot(weight ~ nitrogen * treat, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level",
        cex.axis = 0.7)
```



Violin Plots {-}

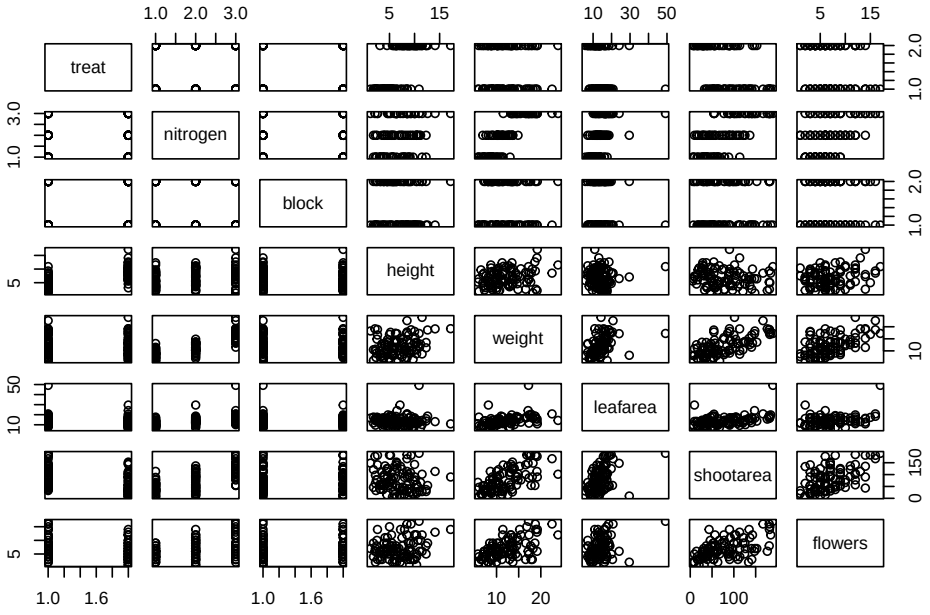
Violin plots are like a combination of a boxplot and a kernel density plot (you saw an example of a kernel density plot in the histogram section above) all rolled into one figure. We can create a violin plot in R using the vioplot() function from the vioplot package. You'll need to first install this package using install.packages('vioplot') function as usual. The nice thing about the vioplot() function is that you use it in pretty much the same way you would use the boxplot() function. We'll also use the argument col = "lightblue" to change the fill colour to light blue.

```
#install.packages("vioplot")
library(vioplot)
#> Loading required package: sm
#> Package 'sm', version 2.2-6.0: type help(sm) for summary information
#> Loading required package: zoo
#>
#> Attaching package: 'zoo'
#> The following objects are masked from 'package:base':
#>
#>     as.Date, as.Date.numeric
vioplot(weight ~ nitrogen, data = flowers,
         ylab = "weight (g)", xlab = "nitrogen level",
         col = "lightblue")
```

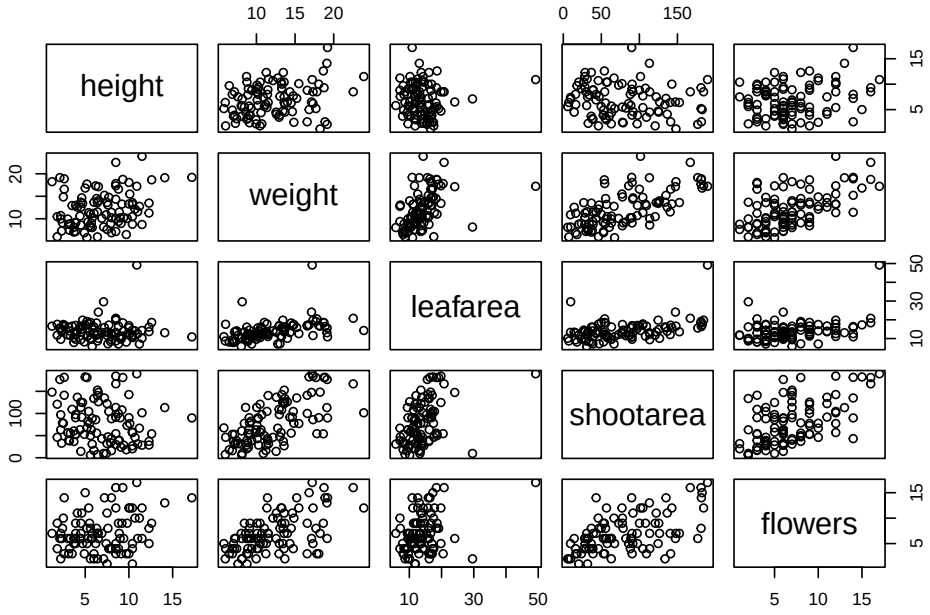


Pairs plot

```
plot(flowers)
```




```
pairs(flowers[, c("height", "weight", "leafarea",
                  "shootarea", "flowers")])
```



Exercise 2

Now, try creating your own visualization using the iris dataset. Here's what you can do:

1. Create a scatter plot using Petal.Length and Petal.Width from the iris dataset.
2. Color the points based on the Species column to differentiate between the species.
3. Add a title, x-axis label, and y-axis label to your plot. Include a legend that indicates which color corresponds to which iris species.

Intermediate R

In this session , we will learn about function and control structures such as writing functions , if statements, and loops.

We will also explore Data Cleaning and Transformations such as handling missing data , reshaping data using the dplyr functions.

First , we need to load this package

```
#install.packages("tidyverse")
library(tidyverse)
#> -- Attaching core tidyverse packages ---- tidyverse 2.0.0 --
#> v dplyr     1.1.4     v readr     2.1.4
#> v forcats   1.0.0     v stringr   1.5.1
#> v ggplot2   3.5.0     v tibble    3.2.1
#> v lubridate 1.9.3     v tidyr     1.3.0
#> v purrr     1.0.2
#> -- Conflicts ---------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
#> i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

It will load

1. `ggplot2`: Used for data visualization using the grammar of graphics.
2. `dplyr`: Provides a set of tools for efficiently manipulating datasets.
3. `tidyr`: Used for tidying data, that is, transforming it to a format that is easy to work with.
4. `readr`: Used to read rectangular data like CSVs and text files into R.
5. `purrr`: Enhances R's functional programming (FP) toolkit, making it easier to work with lists and functions.
6. `tibble`: A modern reimagining of data frames, providing a cleaner and more user-friendly data structure.
7. `stringr`: Simplifies the process of working with strings (text data).
8. `forcats`: Designed to handle categorical variables (factors) with ease.

In this session , we will use some of them

Part I: Functions and Control Structures

Writing and using functions

Example: A simple function to calculate the square of a number

$$f(x) = x^2$$

```
square_function <- function(x) {
  return(x^2)
}

# Using the function
result <- square_function(4)
print(result)
#> [1] 16
```

Exercise 1

Task: Write and Use a Function **Objective:** Create a function that calculates the cube of a number and use this function to calculate the cube of 3.

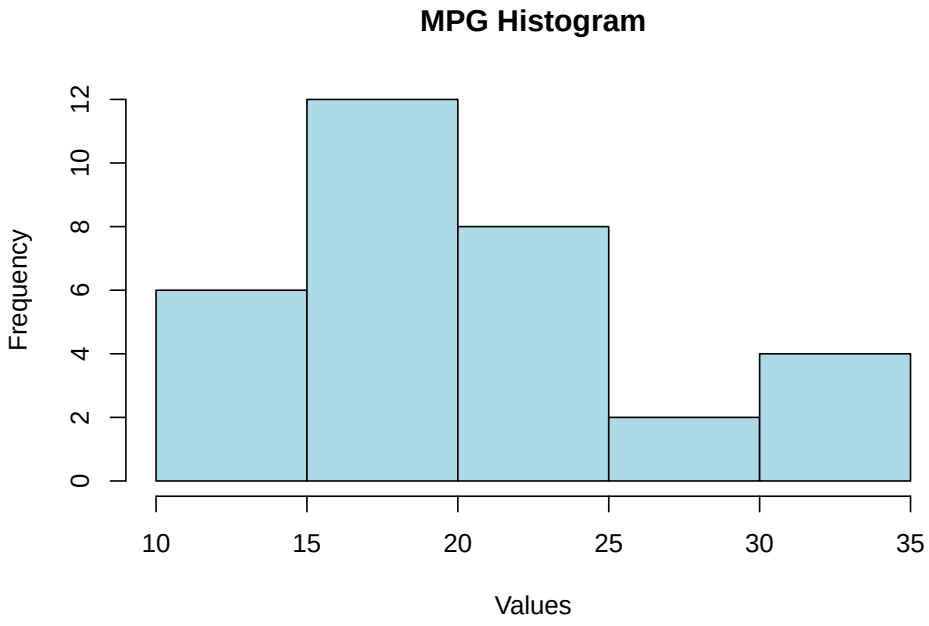
Hint: Use the structure of the square_function as a template.

The reason why this is useful, is because functions are used for anything we want, R functions are just similar to the one we have just created, optimized for the specific tasks they were designed for .

We now create a more complex function , one that takes a vector , finds the mean , standard deviation and the histogram

```
analyze_vector <- function(x, plot_title = "Histogram") {
  # Check if the input is numeric
  if (!is.numeric(x)) {
```

```r
    stop("Input must be a numeric vector")
  }

  # Calculate mean and standard deviation
  mean_value <- mean(x)
  std_value <- sd(x)

  # Output the mean and std
  cat("Mean:", mean_value, "\n")
  cat("Standard Deviation:", std_value, "\n")

  # Create a histogram
  hist(x, main = plot_title, xlab = "Values", col = "lightblue", border = "black")

  # Return a list containing the mean and std
  return(list(mean = mean_value, std = std_value))
}

# Example usage with the mtcars$mpg vector
result <- analyze_vector(mtcars$mpg, "MPG Histogram")
#> Mean: 20.09062
#> Standard Deviation: 6.026948
```



Exercise 2

Task: Analyze a Numeric Vector **Objective:** Write a function named summarize_vector that takes a numeric vector as input and calculates the median, variance, and creates a boxplot. The function should print the median and variance, and return them as a list. Use the airquality$Ozone data for analysis.

Hint: Similar to analyze_vector, check if the input is numeric and use median, var, and `boxplot` functions.

If statements and loops (for and while)

```
# Example: Using if statement
number <- 5
if (number > 0) {
  print("Positive number")
} else {
  print("Non-positive number")
}
#> [1] "Positive number"
```

Exercise 3

Task: Using if Statements **Objective:** Create an R script that checks if a number is negative, zero, or positive and prints an appropriate message. Test your script with the number -4.

Hint: Use an if statement followed by else if and else.

Example:

For loop to calculate the factorial of a number

```
factorial_function <- function(n) {
  result <- 1
  for (i in 1:n) {
    result <- result * i
  }
  return(result)
}

factorial_of_5 <- factorial_function(5)
print(factorial_of_5)
#> [1] 120
```

Exercise 4

Task: For Loop **Objective:** Write a function using a for loop that calculates the sum of squares of numbers from 1 to n. Use this function to calculate the sum of squares for n=10.

Hint: Iterate from 1 to n, and keep adding the square of each number to a result variable.

Example:

While loop to find the first square number greater than 100

```
number <- 1
while (number^2 <= 100) {
  number <- number + 1
}
print(paste("First square number greater than 100 is:", number^2))
#> [1] "First square number greater than 100 is: 121"
```

Exercise 5

Task: While Loop **Objective:** Write a script using a while loop that finds the smallest number whose cube is greater than 100. Print the number and its cube.

Hint: Increment a number starting from 1, and check if its cube is greater than 100 in the while loop condition.

Part II: Data Wrangling

Data wrangling, also known as data munging, is the process of transforming and mapping data from one "raw" form into another format with the intent of making it more appropriate and valuable for a variety of downstream purposes, such as analytics.

In R, data wrangling is often performed using functions from the base R language, as well as a collection of packages known as the tidyverse. The tidyverse is a coherent system of packages for data manipulation, exploration, and visualization that share a common design philosophy.

The tidyverse approach to data wrangling typically involves:

1. Tidying Data: Transforming datasets into a consistent form that makes it easier to work with. This usually means converting data to a tidy format where each variable forms a column, each observation forms a row, and each type of observational unit forms a table.

2. Transforming Data: Once the data is tidy, a series of functions are used for data manipulation tasks such as selecting specific columns (select()), filtering for certain rows (filter()), creating new columns or modifying existing ones (mutate() or transmute()), summarizing data (summarise()), and reshaping data (pivot_longer() and pivot_wider()).

3. Working with Data Types and Structures: Functions from tidyverse allow for the easy manipulation of data types (like converting character vectors to factors with forcats) and data structures (like tibbles with the tibble package, which are a modern take on data frames).

4. Joining Data: Combining different datasets in a variety of ways (like left_join(), right_join(), inner_join(), full_join(), and anti_join()) based on common keys or identifiers.

5. Handling Strings and Dates: The tidyverse includes packages like stringr for string operations and lubridate for dealing with date-time objects, which are essential in many data wrangling tasks.

6. Functional Programming: The package purrr introduces powerful func-

tional programming tools to iterate over data structures and perform operations repeatedly.

The primary goal of data wrangling is to ensure that the data is in the best possible format for analysis. The tidyverse provides tools that make these tasks straightforward, efficient, and often more intuitive than the base R equivalents. The philosophy of the tidyverse is to write readable and transparent code that can be understood even if you come back to it months or years later.

Reshaping data using dplyr functions (filter, arrange, mutate, summarize)

The `dplyr` package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package. The `dplyr` package does not provide any "new" functionality to R per se, in the sense that everything dplyr does could already be done with base R, but it greatly simplifies existing functionality in R.

One important contribution of the `dplyr` package is that it provides a "grammar" (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar). This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the `dplyr` functions are very fast, as many key operations are coded in C++.

The `dplyr` grammar

Some of the key "verbs" provided by the dplyr package are

- select: return a subset of the columns of a data frame, using a flexible notation
- filter: extract a subset of rows from a data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in a data frame
- mutate: add new variables/columns or transform existing variables
- summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata
- `%>%`: the "pipe" operator is used to connect multiple verb actions together into a pipeline.

These all combine naturally with group_by() which allows you to perform any operation "by group".

More on the pipe operator

- It takes the output of one statement and makes it the input of the next statement.
- When describing it, you can think of it as a "THEN". A first example:
  - take the diamonds data (from the ggplot2 package)
  - then subset

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
library(ggplot2)
diamonds %>% filter(cut == "Ideal")
#> # A tibble: 21,551 x 10
#>    carat cut   color clarity depth table price     x     y
#>    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl>
#>  1  0.23 Ideal E     SI2      61.5    55   326  3.95  3.98
#>  2  0.23 Ideal J     VS1      62.8    56   340  3.93  3.9
#>  3  0.31 Ideal J     SI2      62.2    54   344  4.35  4.37
#>  4  0.3  Ideal I     SI2      62      54   348  4.31  4.34
#>  5  0.33 Ideal I     SI2      61.8    55   403  4.49  4.51
#>  6  0.33 Ideal I     SI2      61.2    56   403  4.49  4.5
#>  7  0.33 Ideal J     SI1      61.1    56   403  4.49  4.55
#>  8  0.23 Ideal G     VS1      61.9    54   404  3.93  3.95
#>  9  0.32 Ideal I     SI1      60.9    55   404  4.45  4.48
#> 10  0.3  Ideal I     SI2      61      59   405  4.3   4.33
#> # i 21,541 more rows
#> # i 1 more variable: z <dbl>
```

Filter()

Extract rows that meet logical criteria. Here you go: - inspect the diamonds data set - filter observations with cut equal to Ideal

```
filter(diamonds, cut == "Ideal")
#> # A tibble: 21,551 x 10
#>    carat cut   color clarity depth table price     x     y
#>    <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl>
#>  1  0.23 Ideal E     SI2      61.5    55   326  3.95  3.98
#>  2  0.23 Ideal J     VS1      62.8    56   340  3.93  3.9
#>  3  0.31 Ideal J     SI2      62.2    54   344  4.35  4.37
```

```
#>  4  0.3  Ideal I     SI2      62      54   348  4.31  4.34
#>  5  0.33 Ideal I     SI2      61.8    55   403  4.49  4.51
#>  6  0.33 Ideal I     SI2      61.2    56   403  4.49  4.5
#>  7  0.33 Ideal J     SI1      61.1    56   403  4.49  4.55
#>  8  0.23 Ideal G     VS1      61.9    54   404  3.93  3.95
#>  9  0.32 Ideal I     SI1      60.9    55   404  4.45  4.48
#> 10  0.3  Ideal I     SI2      61      59   405  4.3   4.33
#> # i 21,541 more rows
#> # i 1 more variable: z <dbl>
```

Overview of logical tests

x < y	Less than
x > y	Greater than
x == y	Equal to
x <= y	Less than or equal to
x >= y	Greater than or equal to
x != y	Not equal to
x %in% y	Group membership
is.na(x)	Is NA
!is.na(x)	Is not NA

Mutate()

Create new columns. Here you go: - inspect the diamonds data set - create a new variable price_per_carat

```
mutate(diamonds, price_per_carat = price/carat)
#> # A tibble: 53,940 x 11
#>    carat cut       color clarity depth table price     x     y
#>    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl>
#>  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98
#>  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84
#>  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07
#>  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23
#>  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35
#>  6  0.24 Very G~   J     VVS2     62.8    57   336  3.94  3.96
#>  7  0.24 Very G~   I     VVS1     62.3    57   336  3.95  3.98
#>  8  0.26 Very G~   H     SI1      61.9    55   337  4.07  4.11
#>  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78
#> 10  0.23 Very G~   H     VS1      59.4    61   338  4     4.05
#> # i 53,930 more rows
#> # i 2 more variables: z <dbl>, price_per_carat <dbl>
```

Multistep operations

Use the %>% for multistep operations. Passes result on left into first argument of function on right. Here you go:

```
diamonds %>%
  mutate(price_per_carat = price/carat)  %>%
  filter(price_per_carat > 1500)
#> # A tibble: 52,821 x 11
#>    carat cut       color clarity depth table price     x     y
#>    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl>
#>  1  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84
#>  2  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78
#>  3  0.22 Premium   F     SI1      60.4    61   342  3.88  3.84
#>  4  0.2  Premium   E     SI2      60.2    62   345  3.79  3.75
#>  5  0.23 Very G~   E     VS2      63.8    55   352  3.85  3.92
#>  6  0.23 Very G~   H     VS1      61      57   353  3.94  3.96
#>  7  0.23 Very G~   G     VVS2     60.4    58   354  3.97  4.01
#>  8  0.23 Very G~   D     VS2      60.5    61   357  3.96  3.97
#>  9  0.23 Very G~   F     VS1      60.9    57   357  3.96  3.99
#> 10  0.23 Very G~   F     VS1      60      57   402  4     4.03
#> # i 52,811 more rows
#> # i 2 more variables: z <dbl>, price_per_carat <dbl>
```

Summarize()

Compute table of summaries. Here you go:

- inspect the diamonds data set
- calculate mean and standard deviation of price

```
diamonds %>% summarize(mean = mean(price), std_dev = sd(price))
#> # A tibble: 1 x 2
#>    mean std_dev
#>   <dbl>   <dbl>
#> 1 3933.   3989.
```

Group_by()

Groups cases by common values of one or more columns. Here you go: inspect the diamonds data set calculate mean and standard deviation of price by level of cut

```
diamonds %>%
        group_by(cut) %>%
        summarize(price = mean(price), carat = mean(carat))
```

```
#> # A tibble: 5 x 3
#>   cut       price carat
#>   <ord>     <dbl> <dbl>
#> 1 Fair      4359. 1.05
#> 2 Good      3929. 0.849
#> 3 Very Good 3982. 0.806
#> 4 Premium   4584. 0.892
#> 5 Ideal     3458. 0.703
```

Exercise 1

1. Load the data Parade2005.txt.
2. Determine the mean earnings in California.
3. Determine the number of individuals residing in Idaho.
4. Determine the mean and the median earnings of celebrities.

Transforming a dataframe into tibbles

Transform the mtcars into a tibble and inspect.

```
str(mtcars)
#> 'data.frame':    32 obs. of  11 variables:
#>  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#>  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
#>  $ disp: num  160 160 108 258 360 ...
#>  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
#>  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
#>  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
#>  $ qsec: num  16.5 17 18.6 19.4 17 ...
#>  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
#>  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
#>  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
#>  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
#library(tidyverse)
library(tibble)
as_tibble(mtcars)
#> # A tibble: 32 x 11
#>      mpg   cyl  disp    hp  drat    wt  qsec    vs    am
#>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1  21       6  160    110  3.9   2.62  16.5     0     1
#>  2  21       6  160    110  3.9   2.88  17.0     0     1
#>  3  22.8     4  108     93  3.85  2.32  18.6     1     1
#>  4  21.4     6  258    110  3.08  3.22  19.4     1     0
#>  5  18.7     8  360    175  3.15  3.44  17.0     0     0
```

```
#>  6  18.1     6  225    105  2.76  3.46  20.2     1     0
#>  7  14.3     8  360    245  3.21  3.57  15.8     0     0
#>  8  24.4     4  147.    62  3.69  3.19  20       1     0
#>  9  22.8     4  141.    95  3.92  3.15  22.9     1     0
#> 10  19.2     6  168.   123  3.92  3.44  18.3     1     0
#> # i 22 more rows
#> # i 2 more variables: gear <dbl>, carb <dbl>
```

Part II : Data Cleaning and Transformation

Data cleaning is a fundamental step in the data analysis process, aimed at improving data quality and ensuring its appropriateness for specific analytical tasks. The process involves identifying and rectifying errors or inconsistencies in data to enhance its accuracy, completeness, and reliability.

Key aspects of data cleaning include:

- **Removing Duplicates**: This involves detecting and eliminating duplicate records that could skew analysis results.
- **Handling Missing Dat**a: Missing values can be dealt with by imputing data (filling in missing values using statistical methods or domain knowledge), or in some cases, deleting rows or columns with too many missing values.
- **Correcting Errors**: This involves identifying outliers or incorrect entries (due to data entry errors, measurement errors, etc.) and correcting them based on context or predefined rules.
- **Standardizing Formats**: Ensuring that data across different sources or fields conforms to a consistent format, such as converting all dates to the same format, standardizing text entries (capitalization, removing leading/trailing spaces), or ensuring consistent measurement units.
- **Filtering Irrelevant Information**: Removing data that is not relevant to the specific analysis task to focus on more significant data.
- **Validating Accuracy**: Checking data against known standards or validation rules to ensure it correctly represents the real-world constructs it is supposed to reflect.
- **Consolidating Data Sources**: Combining data from multiple sources and ensuring that the combined dataset is coherent and correctly integrated.

The aim of data cleaning is not only to correct errors but also to bring structure and order to the data, facilitating more effective and accurate analysis. By cleaning data, analysts can ensure that their insights and conclusions are based on reliable and valid data, which is crucial for making informed decisions.

The Policy data set

- PolicyData.csv available in the course material
- Data stored in a .csv file.
- Individual records separated by a semicolon.

```
policy_data <- read.csv(file = './Data/PolicyData.csv', sep = ';')
```

Exercise 1

Use the skills you obtained in the first R workshop and Part 1

1. Inspect the top rows of the data set.
2. How many observations does the data set contain?
3. Calculate the total exposure (exposition) in each region (type_territoire).

The Gapminder package

- Describes the evolution of a number of population characteristics (GDP, life expectancy, …) over time.

```
#install.packages("gapminder")
library(gapminder)
```

Exercise 2

Use the skills obtained in Part I:

1. Inspect the top rows of the data.
2. Select the data for countries in Asia.
3. Which type of variable is `country`?

Revisit factor()

What is a factor variable ?

- Representation for categorical data.
- Predefined list of outcomes (levels).
- Protecting data quality.

Example , sex a categorical value with two possible outcomes, `m` and `f`

```
sex <- factor(c('m', 'f', 'm', 'f'),
              levels = c('m', 'f'))
sex
#> [1] m f m f
#> Levels: m f
```

- The `factor` command creates a new factor variable. The first input is the categorical variable.
- `levels` specifies the possible outcomes of the variable.

Assigning an unrecognized level to a factor variable results in a warning

```
sex[1] <- 'male'
#> Warning in `[<-.factor`(`*tmp*`, 1, value = "male"):
#> invalid factor level, NA generated
```

This protects the quality of the data

```
sex
#> [1] <NA> f    m    f
#> Levels: m f
```

The value NA is assigned to the invalid observation.

levels()

levels print the allowed outcomes for a factor variable

```
levels(sex)
#> [1] "m" "f"
```

Assigning a vector to levels() renames the allowed outcomes.

```
levels(sex) <- c('male', 'female')
sex
#> [1] <NA>   female male   female
#> Levels: male female
```

Exercise 4

The variable country in the gapminder data set is a factor variable.

1. What are the possible levels for country in the subset asia.
2. Is this the result you expected?

To add a level

```
levels(sex) <- c(levels(sex), 'x')
```

cut()

```
gapminder
#> # A tibble: 1,704 x 6
#>    country     continent  year lifeExp      pop gdpPercap
```

```
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # i 1,694 more rows
```

```
head(cut(gapminder$pop,
    breaks = c(0, 10^7, 5*10^7, 10^8, Inf)))
#> [1] (0,1e+07]     (0,1e+07]     (1e+07,5e+07] (1e+07,5e+07]
#> [5] (1e+07,5e+07] (1e+07,5e+07]
#> 4 Levels: (0,1e+07] (1e+07,5e+07] ... (1e+08,Inf]
```

```
gapminder$pop_category = cut(gapminder$pop,
                             breaks = c(0, 10^7, 5*10^7, 10^8, Inf),
                             labels = c("<= 10M", "10M-50M", "50M-100M", "> 100M"))
```

```
gapminder
#> # A tibble: 1,704 x 7
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
#> # i 1,694 more rows
#> # i 1 more variable: pop_category <fct>
```

Exercise 5

Bin the life expectancy in 2007 in a factor variable.

1. Select the observations for year 2007.

2. Bin the life expectancy in four bins of roughly equal size (hint: quantile).
3. How many observations are there in each bin?

Handling missing data

Some history

The practice of imputing missing values has evolved significantly over the years as statisticians and data scientists have sought to deal with the unavoidable problem of incomplete data. The history of imputation reflects broader trends in statistical methods and computational capabilities, as well as growing awareness of the impacts of different imputation strategies on the integrity of statistical analysis.

Missing Data Mechanisms

Rubin (1976) classified missing data into 3 categories: - Missing Completely at Random (MCAR) - Missing at Random (MAR) - Not Missing at Random (NMAR), also called Missing Not at Random (MNAR) - Aka the most confusing statistical terms ever invented

Early Approaches and Simple Imputation

Early approaches to handling missing data were often quite simple, including methods like listwise deletion (removing any record with a missing value) and pairwise deletion (excluding missing values on a case-by-case basis for each analysis). These methods, while straightforward, can lead to biased results and reduced statistical power if the missingness is not completely random.

Simple imputation techniques, such as filling in missing values with the mean, median, or mode of a variable, were developed as a way to retain as much data as possible. These methods are easy to understand and implement, which contributed to their widespread use, especially in the era before advanced computational methods became widely accessible.

Limitations of Mean and Median Imputation

Imputing missing values with the mean or median is intuitive and can be effective in certain contexts, but these methods have significant limitations:

Bias in Estimation: Mean and median imputation do not account for the inherent uncertainty associated with missing data. They can lead to an underestimation of variances and covariances because they artificially reduce the variability of the imputed variable.

Distortion of Data Distribution: These methods can distort the original distribution of data, especially if the missingness is not random (Missing Not

at Random - MNAR) or if the proportion of missing data is high. This distortion can affect subsequent analyses, such as regression models, by providing misleading results.

Ignores Relationships Between Variables: Mean and median imputation treat each variable in isolation, ignoring the potential relationships between variables. This can be particularly problematic in multivariate datasets where variables may be correlated.

Modern Imputation Techniques

As awareness of the limitations of simple imputation methods grew, researchers developed more sophisticated techniques designed to address these shortcomings:

Multiple Imputation: Developed in the late 20th century, multiple imputation involves creating several imputed datasets by drawing from a distribution that reflects the uncertainty around the true values of missing data. These datasets are then analyzed separately, and the results are combined to produce estimates that account for the uncertainty due to missingness. This method addresses the issue of underestimating variability and provides more reliable statistical inferences.

Model-Based Imputation: Techniques like Expectation-Maximization (EM) algorithms and imputation using random forests or other machine learning models take into account the relationships between variables in a dataset. These methods can more accurately reflect the complex structures in data and produce imputations that preserve statistical relationships.

Conclusion

The evolution of imputation methods from simple mean or median filling to sophisticated model-based and multiple imputation techniques reflects a broader shift in statistical practice. This shift is characterized by increased computational power, more complex datasets, and a deeper understanding of the impact of missing data on statistical inference. While mean and median imputation can still be useful in specific, well-considered circumstances, modern techniques offer more robust and principled approaches to handling missing data.

Missing Values in R

Missing values are denoted by NA or NaN for q undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

0.1.1 Difference Between NA and NaN in R

In R, `NA` and `NaN` represent two different kinds of missing or undefined values, but they are used in distinct contexts:

0.1.1.1 NA (Not Available)

- `NA` stands for **Not Available**.
- It is used to represent **missing or undefined data**, typically in cases where data is expected but not present.
- `NA` can be used in any logical or statistical operation, but unless handled specifically, operations involving `NA` will generally result in `NA`.
- `NA` has a flexible context and can be used with **any data type** in R, such as numeric, character, or logical.
- You can test for `NA` using the `is.na()` function.

0.1.1.2 NaN (Not a Number)

- `NaN` stands for **Not a Number**.
- It is a special value used to represent **undefined or unrepresentable numerical results**, such as the result of `0/0`.
- `NaN` is a specific type of `NA` but specifically for numeric calculations that result in undefined or indeterminate values.
- Operations that result in `NaN` are typically those that are mathematically indeterminate or outside the domain of mathematical functions (e.g., square root of a negative number in the realm of real numbers).
- You can test for `NaN` using the `is.nan()` function. Note that `is.na()` also returns `TRUE` for `NaN` values, reflecting their status as a kind of missing value, but `is.nan()` does not return `TRUE` for all `NA` values.

Key Differences

- **Context of Use**: `NA` is used more broadly for missing data across all data types, while `NaN` is specific to numerical operations that do not produce a defined, real number.
- **Nature of Undefinedness**: `NA` indicates the absence of data, whereas `NaN` indicates that a calculation has failed to produce a meaningful result.

In summary, the use of `NA` vs. `NaN` helps distinguish between data that is missing (`NA`) and numerical operations that result in undefined or unrepresentable values (`NaN`).

```
coffee_data <- data.frame(
  Age = c(25, 32, NA, 45, 22, 33, NA, 28),
  Gender = c("Female", "Male", "Male", "Female", "Female", "Male", "Female", NA),
  Cups_Per_Day = c(1, 3, 2, NA, 2, 3, 1, 2)
)
coffee_data
```

```
#>   Age Gender Cups_Per_Day
#> 1  25 Female            1
#> 2  32   Male            3
#> 3  NA   Male            2
#> 4  45 Female           NA
#> 5  22 Female            2
#> 6  33   Male            3
#> 7  NA Female            1
#> 8  28   <NA>            2
```

Identifying Missing Values

You can use the `is.na()` function to check for missing values. To count them in a specific column:

```
sum(is.na(coffee_data$Age))
#> [1] 2
```

Removing NA Values

A common task in data analysis is removing missing values (NAs).

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
print(bad)
#> [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

We can remove them by

```
x[!bad]
#> [1] 1 2 4 5
```

A faster way ,

```
x[!is.na(x)]
#> [1] 1 2 4 5
```

In a Data frame

Also, using our coffee example,

```
coffee_data_clean <- na.omit(coffee_data)
coffee_data_clean
#>   Age Gender Cups_Per_Day
#> 1  25 Female            1
#> 2  32   Male            3
#> 5  22 Female            2
#> 6  33   Male            3
```

To remove rows with missing values in a specific column:

```
coffee_data_clean2 <- coffee_data[!is.na(coffee_data$Age), ]
coffee_data_clean2
#>   Age Gender Cups_Per_Day
#> 1  25 Female            1
#> 2  32   Male            3
#> 4  45 Female           NA
#> 5  22 Female            2
#> 6  33   Male            3
#> 8  28   <NA>            2
row.names(coffee_data_clean2) <- NULL
coffee_data_clean2
#>   Age Gender Cups_Per_Day
#> 1  25 Female            1
#> 2  32   Male            3
#> 3  45 Female           NA
#> 4  22 Female            2
#> 5  33   Male            3
#> 6  28   <NA>            2
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", NA, "d", NA, "f")
good <- complete.cases(x, y)
good
#> [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
x[good]
#> [1] 1 2 4 5
y[good]
#> [1] "a" "b" "d" "f"
```

You can use complete.cases on data frames too.

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

```
good <- complete.cases(airquality)
head(airquality[good, ])
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 7    23     299  8.6   65     5   7
#> 8    19      99 13.8   59     5   8
```

```
sd(airquality$Ozone)
#> [1] NA
sd(airquality$Ozone, na.rm = TRUE)
#> [1] 32.98788
```

Imputing Missing Values

Replacing missing values with a specific value, like the mean or median:

```
coffee_data2<-coffee_data

coffee_data2$Age[is.na(coffee_data$Age)] <- mean(coffee_data2$Age, na.rm = TRUE)
coffee_data2
#>        Age Gender Cups_Per_Day
#> 1 25.00000 Female            1
#> 2 32.00000   Male            3
#> 3 30.83333   Male            2
#> 4 45.00000 Female           NA
#> 5 22.00000 Female            2
#> 6 33.00000   Male            3
#> 7 30.83333 Female            1
#> 8 28.00000   <NA>            2
```

```
# Assuming 'median' is the mode of the column
median(coffee_data$Age, na.rm = TRUE)
#> [1] 30
coffee_data2$Age[is.na(coffee_data$Age)] <- median(coffee_data$Age, na.rm = TRUE)
coffee_data2
#>   Age Gender Cups_Per_Day
#> 1  25 Female            1
#> 2  32   Male            3
#> 3  30   Male            2
#> 4  45 Female           NA
#> 5  22 Female            2
#> 6  33   Male            3
#> 7  30 Female            1
#> 8  28   <NA>            2
```

Using Packages for Advanced Imputation

```
# install.packages("mice")
library(mice)
#>
#> Attaching package: 'mice'
#> The following object is masked from 'package:stats':
#>
#>     filter
#> The following objects are masked from 'package:base':
#>
#>     cbind, rbind
# Display the first few rows of the airquality dataset
head(airquality)
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6

# Perform multiple imputation
imputed_data <- mice(airquality, m=5, method='pmm', seed = 123)
#>
#>  iter imp variable
#>   1   1  Ozone  Solar.R
#>   1   2  Ozone  Solar.R
#>   1   3  Ozone  Solar.R
#>   1   4  Ozone  Solar.R
#>   1   5  Ozone  Solar.R
#>   2   1  Ozone  Solar.R
#>   2   2  Ozone  Solar.R
#>   2   3  Ozone  Solar.R
#>   2   4  Ozone  Solar.R
#>   2   5  Ozone  Solar.R
#>   3   1  Ozone  Solar.R
#>   3   2  Ozone  Solar.R
#>   3   3  Ozone  Solar.R
#>   3   4  Ozone  Solar.R
#>   3   5  Ozone  Solar.R
#>   4   1  Ozone  Solar.R
#>   4   2  Ozone  Solar.R
#>   4   3  Ozone  Solar.R
#>   4   4  Ozone  Solar.R
```

```
#>   4   5  Ozone  Solar.R
#>   5   1  Ozone  Solar.R
#>   5   2  Ozone  Solar.R
#>   5   3  Ozone  Solar.R
#>   5   4  Ozone  Solar.R
#>   5   5  Ozone  Solar.R

# Extract the first completed dataset
completed_data <- complete(imputed_data, 1)

# Display the first few rows of the completed data
head(completed_data)
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    18     150 14.3   56     5   5
#> 6    28      48 14.9   66     5   6
```

Exercise 1: Explore Missingness

Dataset: ChickWeight

Task: Determine if the ChickWeight dataset contains any missing values. Print a message stating whether the dataset has missing values or not.

Hint Use the any() function combined with is.na() applied to the dataset.

Exercise 2: Calculate Summary Statistics Before Handling NA

```
data(mtcars)
mean_mpg <- mean(mtcars$mpg)
mean_mpg
#> [1] 20.09062
sd_mpg <- sd(mtcars$mpg)
sd_mpg
#> [1] 6.026948
```

Dataset: mtcars

Task: The mtcars dataset is almost complete but let's pretend some values are missing in the mpg (miles per gallon) column. First, artificially introduce

missing values into the mpg column (e.g., set the first three values of mpg to NA). Then, calculate and print the mean and standard deviation of mpg without removing or imputing the missing values.

Hint: Modify the mtcars$mpg directly to introduce NAs. Use mean() and sd() functions with na.rm = FALSE to calculate statistics without handling NA.

Exercise 3: Impute Missing Values with Column Median

Dataset: mtcars with modified mpg

Task: First Calculate the mean and standard deviation handling the missing values.

Then,Impute the artificially introduced missing values in the mpg column with the column's median (excluding the missing values). Print the first 6 rows of the modified mtcars dataset.

Now, calculate the mean and standard deviation with the imputed values.

Hint: First, calculate the median of mpg excluding NAs. Then, use indexing to replace NAs with this median.

Exercise 4: Identifying Complete Rows

Dataset: airquality

Task: Before any analysis, you want to ensure that only complete cases are used. Create a new dataset from airquality that includes only the rows without any missing values. Print the number of rows in the original versus the cleaned dataset.

Hint Use complete.cases() on the dataset and then subset it.

Exercise 5: Advanced Imputation on a Subset

Dataset: mtcars

Task: Create a subset of mtcars containing only the mpg, hp (horsepower), and wt (weight) columns. Introduce missing values in hp and wt columns (e.g., set first two values of each to NA). Perform multiple imputation using the mice package on this subset with 3 imputations, and extract the third completed dataset. Print the first 6 rows of this completed dataset.

Hint: Subset mtcars first, then modify to add NAs. Use mice() for imputation and complete() to extract the desired imputed dataset.

Intermediate R II

In this session , we will do some more advanced data visualization ,and statistical analysis.

If time permits, we will cover Dates and Times

Part I: Advanced Data Visualization

The aim of the `ggplot2` package is to create elegant data visualizations using the grammar of graphics.

Here are the basic steps:

- begin a plot with the function `ggplot()` creating a coordinate system that you can add layers to

-the first argument of `ggplot()` is the dataset to use in the graph

We will use the mpg dataset from `ggplot2`

```
library(ggplot2)
head(mpg)
#> # A tibble: 6 x 11
#>   manufacturer model displ  year   cyl trans     drv     cty
#>   <chr>        <chr> <dbl> <int> <int> <chr>     <chr> <int>
#> 1 audi         a4      1.8  1999     4 auto(l5)  f        18
#> 2 audi         a4      1.8  1999     4 manual(m~ f        21
#> 3 audi         a4      2    2008     4 manual(m~ f        20
#> 4 audi         a4      2    2008     4 auto(av)  f        21
#> 5 audi         a4      2.8  1999     6 auto(l5)  f        16
#> 6 audi         a4      2.8  1999     6 manual(m~ f        18
#> # i 3 more variables: hwy <int>, fl <chr>, class <chr>
```

Run the following code,

What do you obtain ?

```
ggplot(data = mpg)
```

```
ggplot(mpg)
```


You create an empty graph.

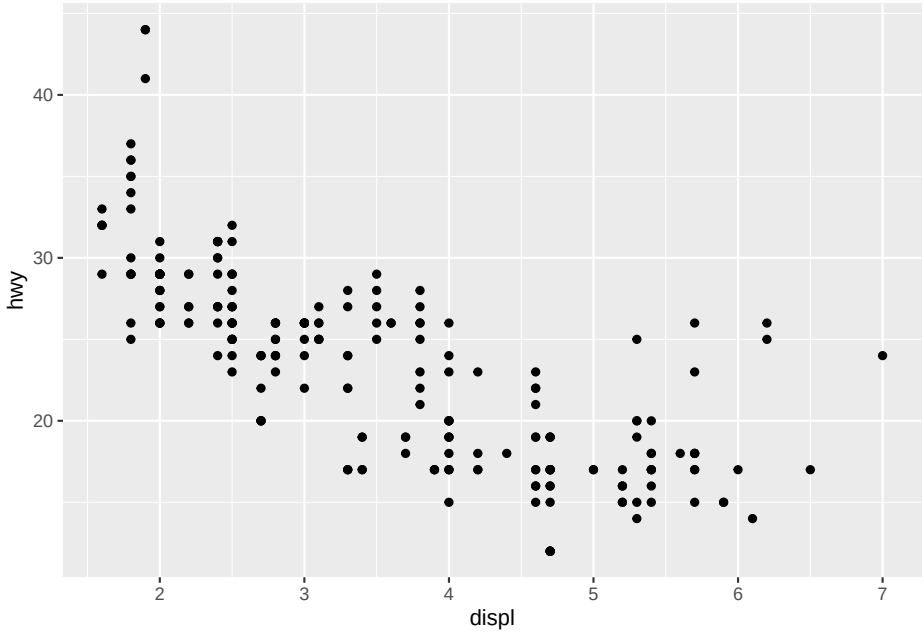
You complete the graph by adding one or more layers to ggplot()

for example: - geom_point() adds a layer of points to your plot, which creates

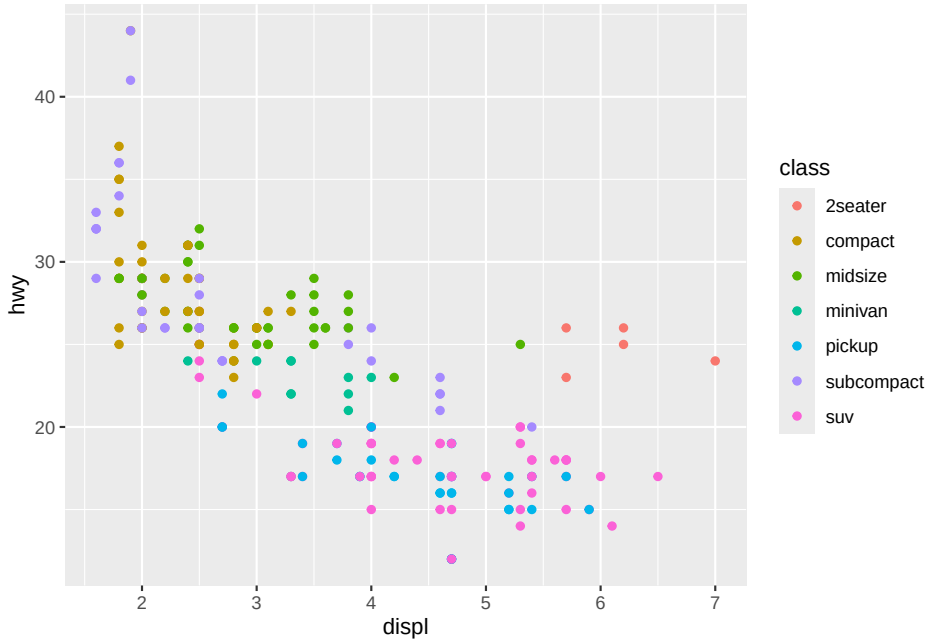
a scatterplot - geom_smooth() adds a smooth line - geom_bar a bar plot.

Each geom function in ggplot2 takes a mapping argument: - how variables in your dataset are mapped to visual properties - always paired with `aes()` and the and arguments of `aes()` specify which variables to map to the and axes.

```
library(ggplot2)
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



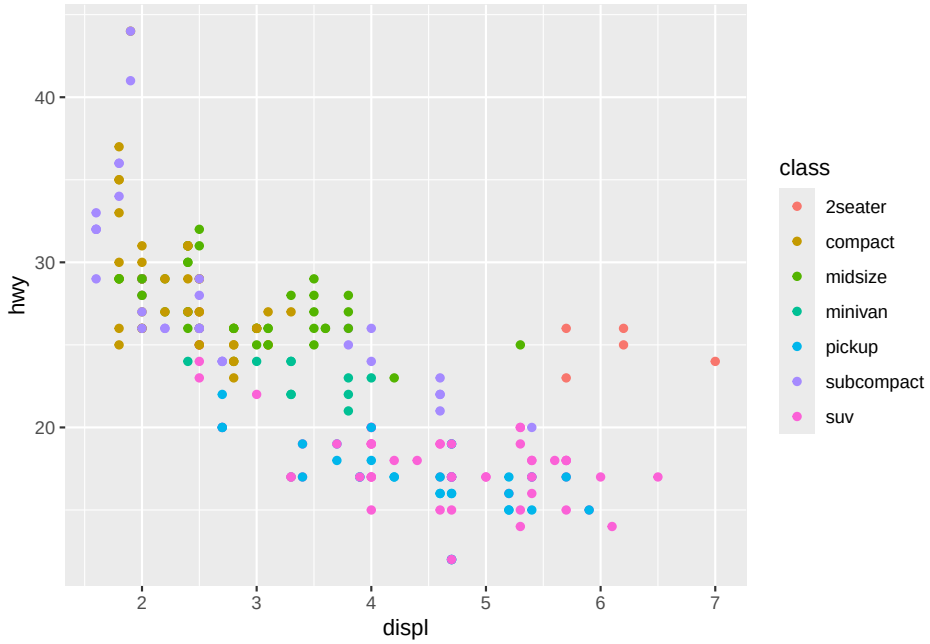

```
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, color = class))
```



Compare the following set of instructions:

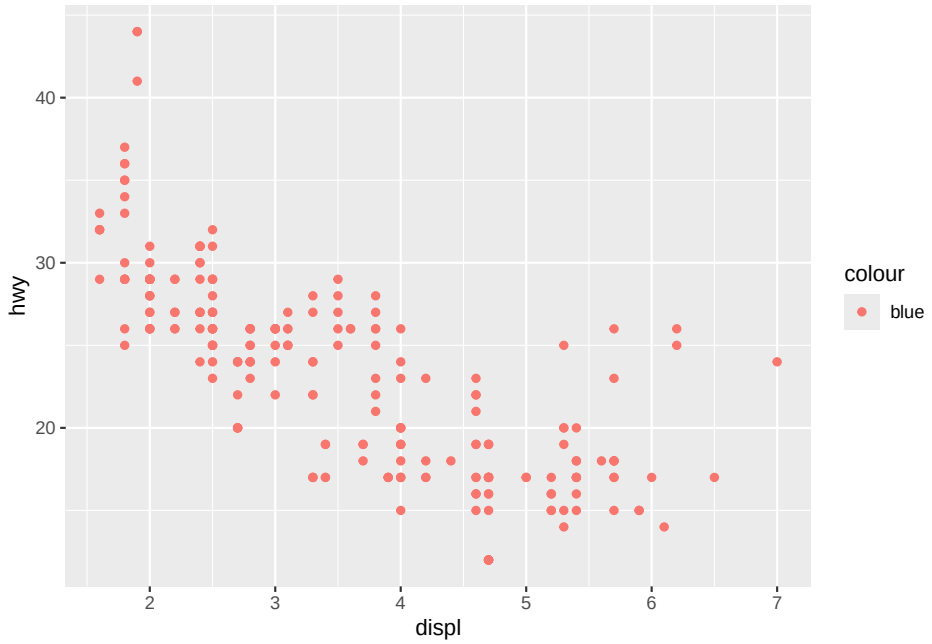
- inside of aesthetics

```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy, color = class))
```



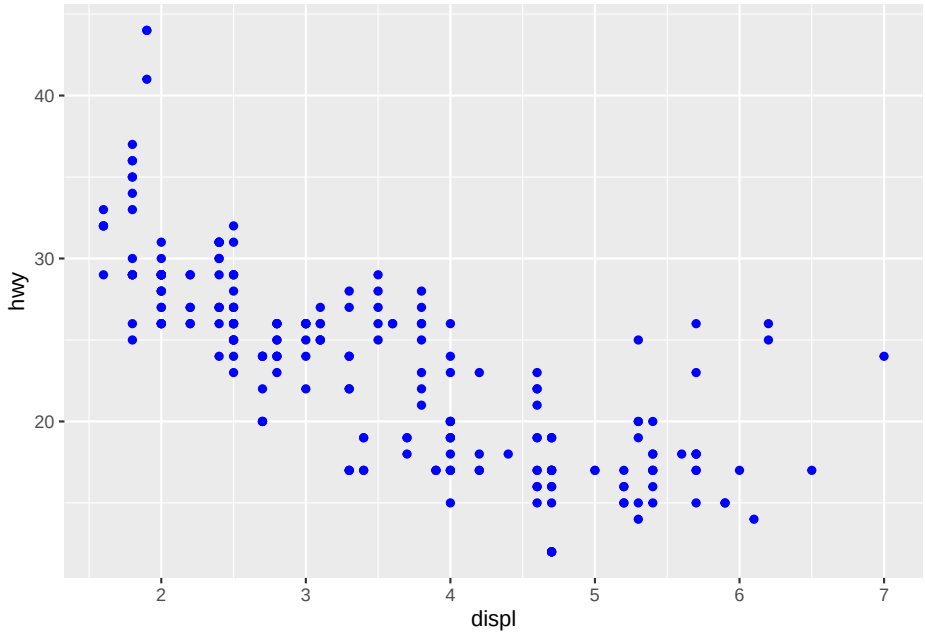
- inside of aesthetics, not mapped to a variable

```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy, color = "blue"))
```



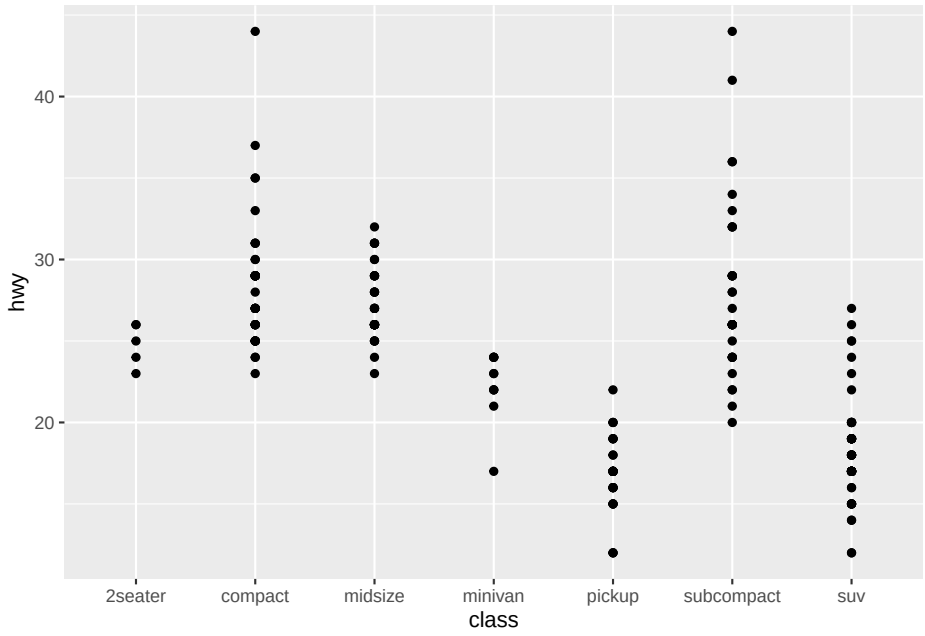
- outside of aesthetics

```
ggplot(mpg) + geom_point(aes(x = displ, y = hwy), color = "blue")
```



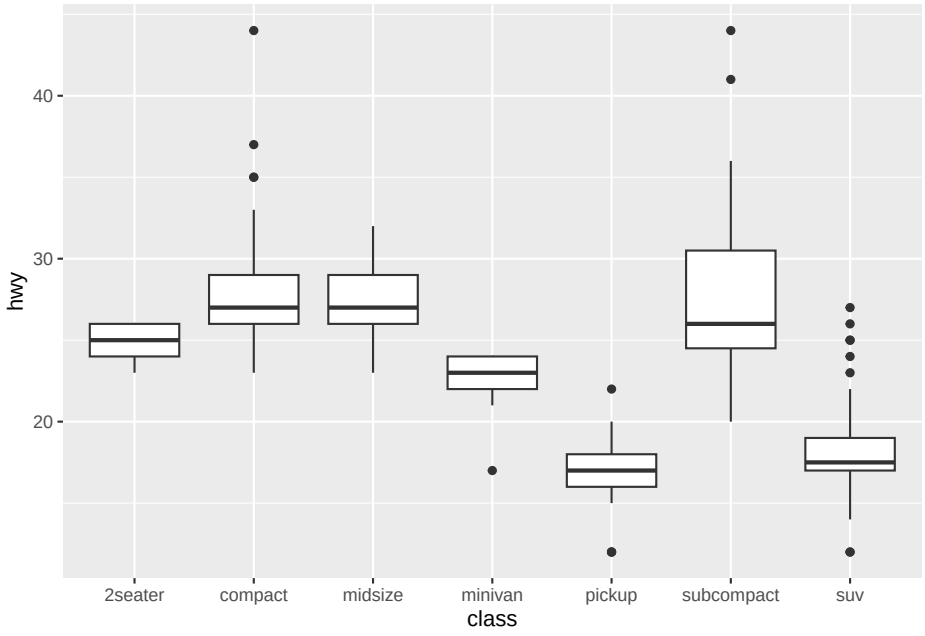
- Scatterplot

```
ggplot(mpg) +
  geom_point(mapping = aes(x = class, y = hwy))
```



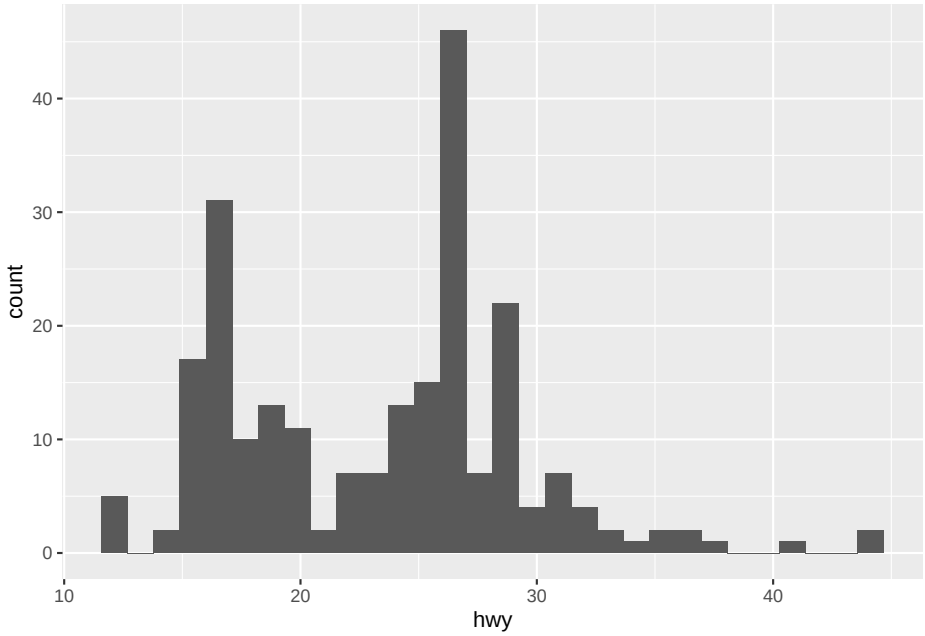
- boxplot

```
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = class, y = hwy))
```

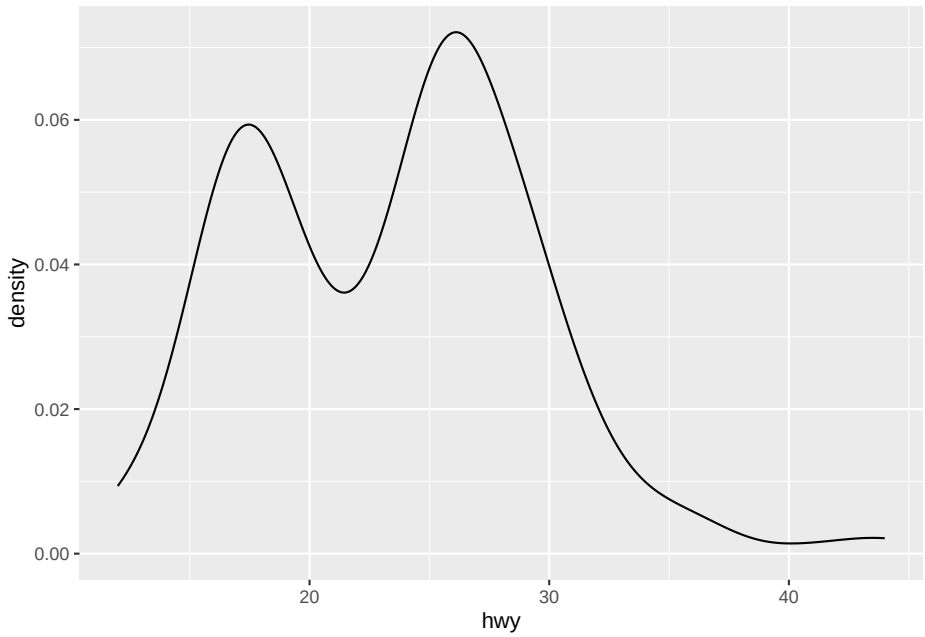


- histogram

```
ggplot(data = mpg) +
  geom_histogram(mapping = aes(x = hwy))
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```

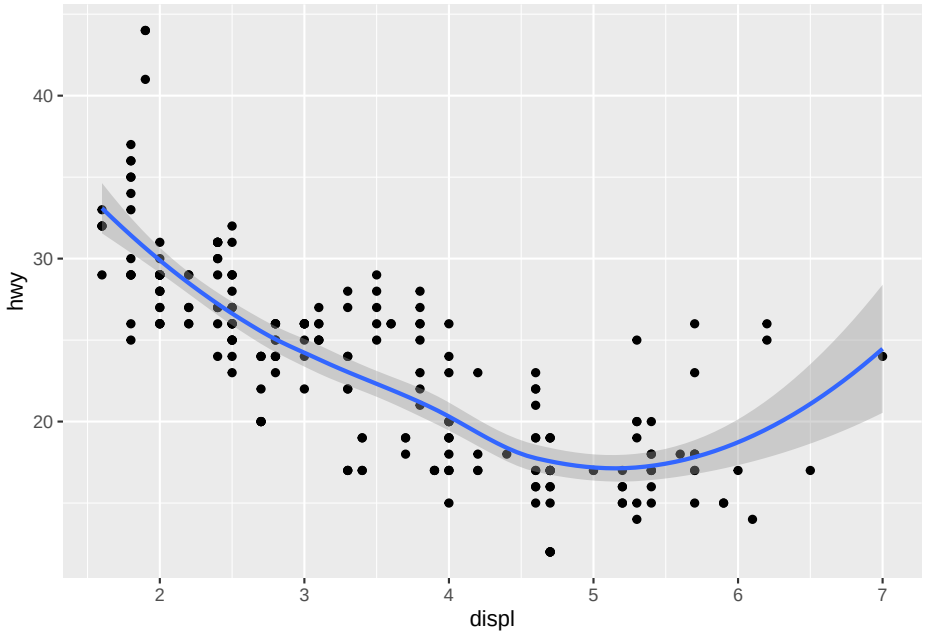



```
ggplot(data = mpg) +
  geom_density(mapping = aes(x = hwy))
```



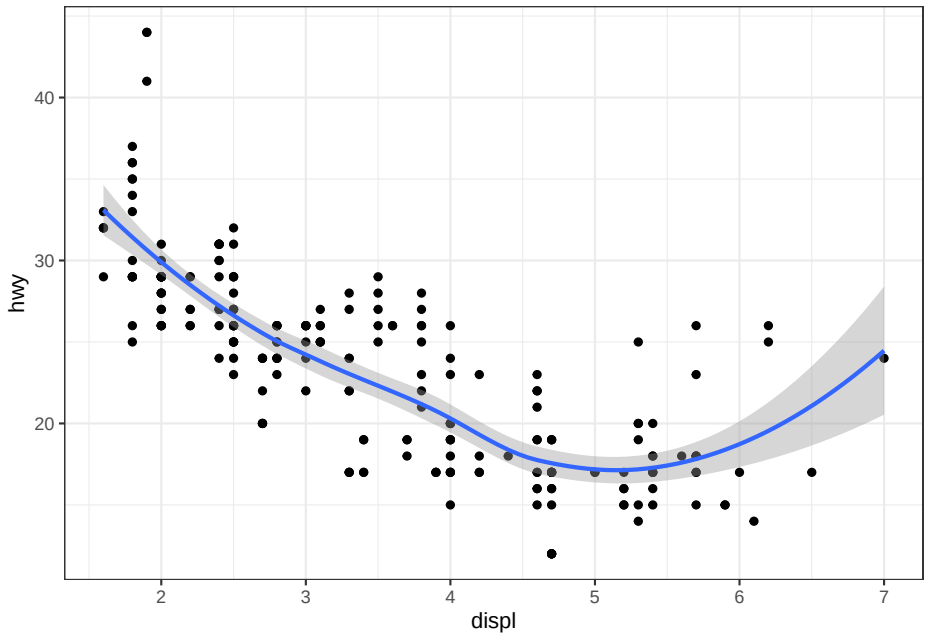
Now you will add multiple geoms to the same plot. Predict what the following code does:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
#> `geom_smooth()` using method = 'loess' and formula = 'y ~
#> x'
```



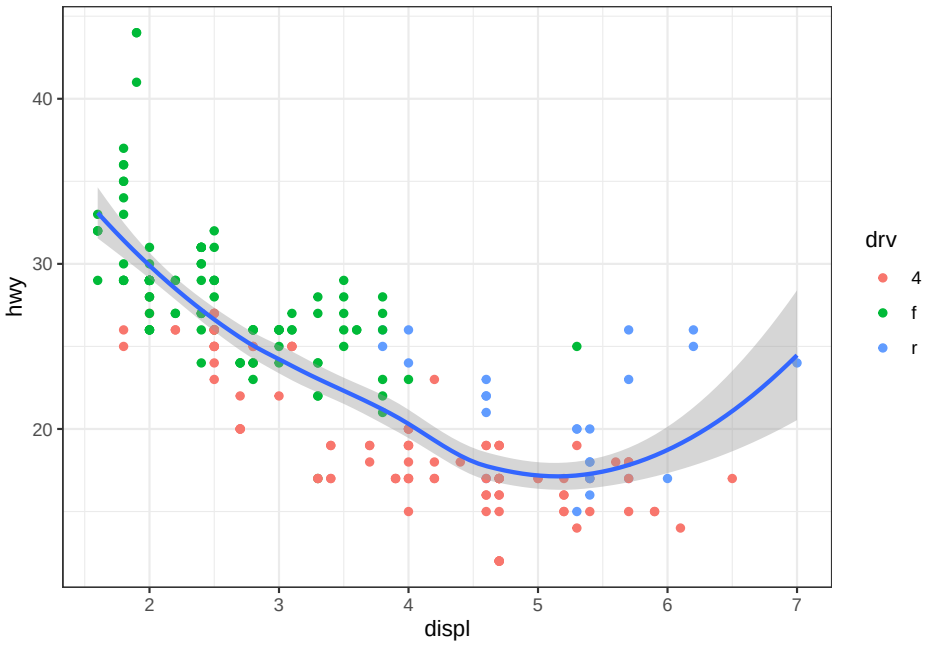
Mappings and data can be specified global (in `ggplot()`) or local.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth() + theme_bw()        # adjust theme
#> `geom_smooth()` using method = 'loess' and formula = 'y ~
#> x'
```

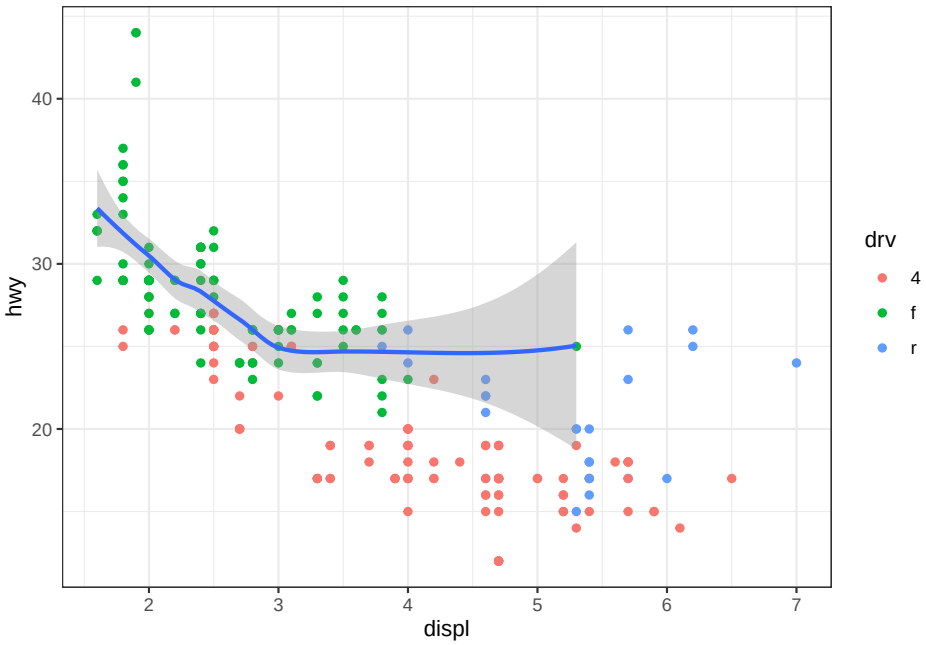


local.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = drv)) +
  geom_smooth() + theme_bw()
#> `geom_smooth()` using method = 'loess' and formula = 'y ~
#> x'
```




```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = drv)) +
  geom_smooth(data = filter(mpg, drv == "f")) + theme_bw()
#> `geom_smooth()` using method = 'loess' and formula = 'y ~
#> x'
```



Exercise 1

Use the Danish fire insurance losses. Plot the arrival of losses over time.

1. Use type= "l" for a line plot, label the and -axis, and give the plot a title using main.
2. Do the same with instructions from ggplot2. Use geom_line() to create the line plot.

Exercise 2

1. Use the data set car_price.csv available in the documentation. Import the data in R.

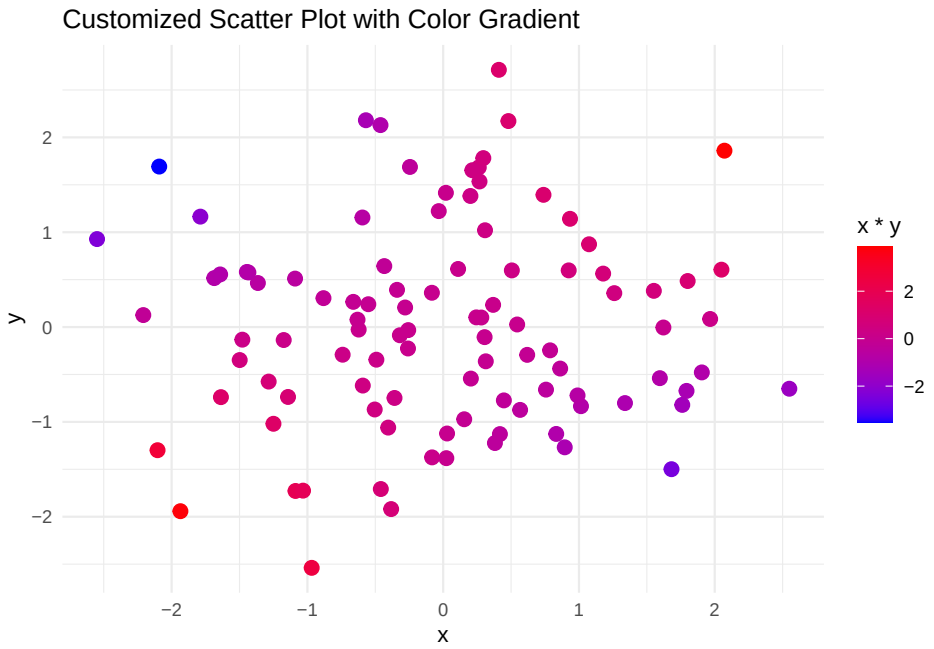
2. Explore the data.

3. Make a scatterplot of price versus income, use basic plotting instructions and use ggplot2.

4. Add a smooth line to each of the plots (using lines to add a line to an existing plot and lowess to do scatterplot smoothing and using geom_smooth in the ggplot2 grammar).

Creating customized plots with ggplot2

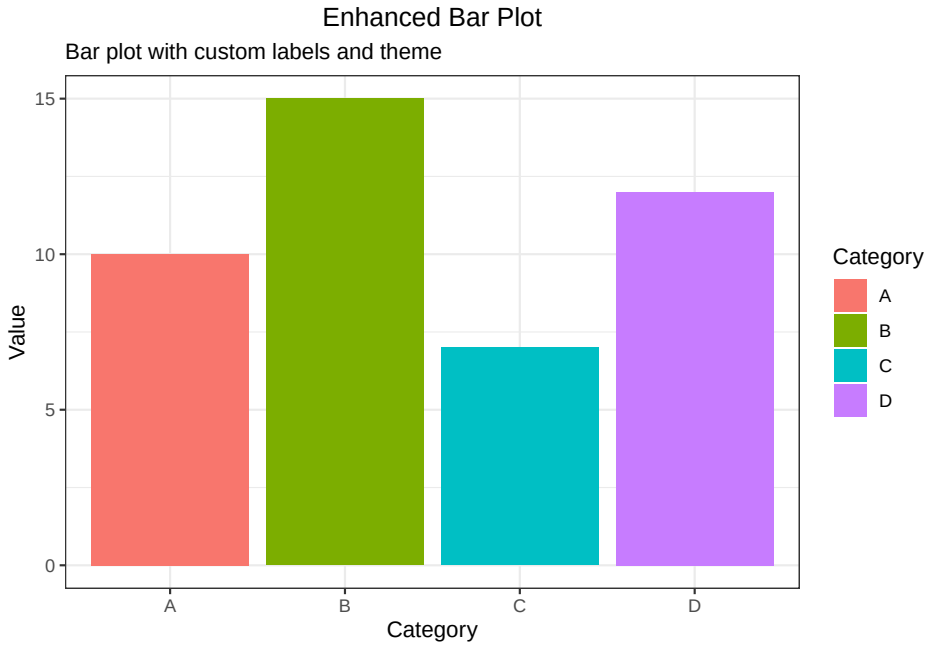
```r
# Load ggplot2 package
library(ggplot2)

# Example: Customized scatter plot with ggplot2
data <- data.frame(x = rnorm(100), y = rnorm(100))
ggplot(data, aes(x = x, y = y)) +
  geom_point(aes(color = x*y), size = 3) +
  scale_color_gradient(low = "blue", high = "red") +
  ggtitle("Customized Scatter Plot with Color Gradient") +
  theme_minimal()
```



Adding titles, labels, and themes to plots

```r
# Example: Enhanced bar plot with titles, labels, and a custom theme
data <- data.frame(
  category = c("A", "B", "C", "D"),
  value = c(10, 15, 7, 12)
)
ggplot(data, aes(x = category, y = value, fill = category)) +
  geom_bar(stat = "identity") +
  labs(title = "Enhanced Bar Plot",
```

```
      subtitle = "Bar plot with custom labels and theme",
      x = "Category",
      y = "Value",
      fill = "Category") +
theme_bw() +
theme(plot.title = element_text(hjust = 0.5))
```



Part II: Statistical Analysis

Introduction to hypothesis testing and statistical tests

Hypothesis testing is a statistical method used to make inferences or draw conclusions about a population based on sample data. It starts with a null hypothesis (H0) that assumes no effect or no difference, and an alternative hypothesis (H1) that contradicts the null hypothesis.

The process involves: 1. Defining the null and alternative hypotheses.

2. Selecting a significance level (alpha, typically 0.05).
3. Calculating a test statistic based on the sample data.
4. Determining the p-value, which is the probability of observing the test statistic or something more extreme under the null hypothesis.
5. Comparing the p-value with the significance level to decide whether to reject the null hypothesis.

Statistical tests vary based on the type of data and the research question. Common tests include t-tests (for means), chi-squared tests (for categorical data), ANOVA (for comparing means across multiple groups), and regression analysis (for relationships between variables).

0.2 Comparing Variances

a. F test to compare variances (Parametric)

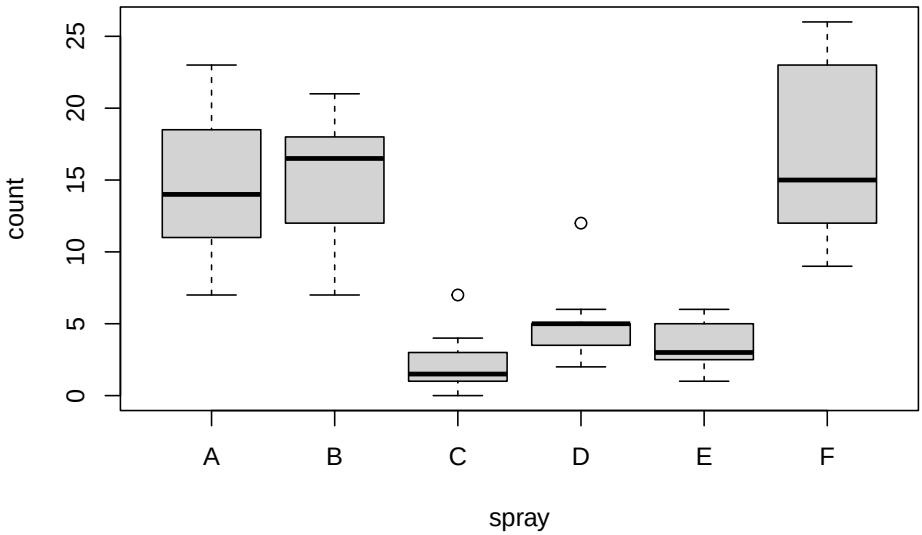
```
x <- rnorm(50, mean = 0, sd = 2)
y <- rnorm(30, mean = 1, sd = 1)
var.test(x, y)
#>
#>  F test to compare two variances
#>
#> data:  x and y
```

```
#> F = 4.2381, num df = 49, denom df = 29, p-value =
#> 8.72e-05
#> alternative hypothesis: true ratio of variances is not equal to 1
#> 95 percent confidence interval:
#>  2.129306 7.973580
#> sample estimates:
#> ratio of variances
#>           4.238073
```

b. Barlett test: Testing homogeneity (Parametric)

Performs Bartlett's test of the null that the variances in each of the groups (samples) are the same.

```
require(graphics)

plot(count ~ spray, data = InsectSprays)
```




```
bartlett.test(InsectSprays$count, InsectSprays$spray)
#>
#>  Bartlett test of homogeneity of variances
#>
#> data:  InsectSprays$count and InsectSprays$spray
#> Bartlett's K-squared = 25.96, df = 5, p-value =
#> 9.085e-05
```

c. Fligner-Killeen Test of Homogeneity of Variances (Non-parametric)

```
fligner.test(InsectSprays$count, InsectSprays$spray)
#>
```

```
#>  Fligner-Killeen test of homogeneity of variances
#>
#> data:  InsectSprays$count and InsectSprays$spray
#> Fligner-Killeen:med chi-squared = 14.483, df = 5,
#> p-value = 0.01282
```

d. Mood Two-Sample Test of Scale (Non-Parametric)

```
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
mood.test(ramsay, jung.parekh)
#>
#>  Mood two-sample test of scale
#>
#> data:  ramsay and jung.parekh
#> Z = 1.0371, p-value = 0.2997
#> alternative hypothesis: two.sided
```

e. Ansari-Bradley Test (Non-parametric)

```
ramsay <- c(111, 107, 100, 99, 102, 106, 109, 108, 104, 99,
            101, 96, 97, 102, 107, 113, 116, 113, 110, 98)
jung.parekh <- c(107, 108, 106, 98, 105, 103, 110, 105, 104,
            100, 96, 108, 103, 104, 114, 114, 113, 108, 106, 99)
ansari.test(ramsay, jung.parekh)
#> Warning in ansari.test.default(ramsay, jung.parekh): cannot
#> compute exact p-value with ties
#>
#>  Ansari-Bradley test
#>
#> data:  ramsay and jung.parekh
#> AB = 185.5, p-value = 0.1815
#> alternative hypothesis: true ratio of scales is not equal to 1
```

Testing two normal distributions

```
ansari.test(rnorm(100), rnorm(100, 0, 2), conf.int = TRUE)
#>
#>  Ansari-Bradley test
#>
#> data:  rnorm(100) and rnorm(100, 0, 2)
#> AB = 5892, p-value = 3.875e-05
#> alternative hypothesis: true ratio of scales is not equal to 1
#> 95 percent confidence interval:
#>  0.4153001 0.7153555
```

```
#> sample estimates:
#> ratio of scales
#>       0.5394925
```

Performing Tests

1. Tests for Comparing Means

a. One-Sample t-Test

We'll test if the average miles per gallon (mpg) in the mtcars dataset is significantly different from 20 mpg.

```
t.test(mtcars$mpg, mu = 20)
#>
#>  One Sample t-test
#>
#> data:  mtcars$mpg
#> t = 0.08506, df = 31, p-value = 0.9328
#> alternative hypothesis: true mean is not equal to 20
#> 95 percent confidence interval:
#>  17.91768 22.26357
#> sample estimates:
#> mean of x
#>  20.09062
```

The results of this one-sample t-test suggest that the average mpg for the cars in the `mtcars` dataset is not significantly different from 20 mpg, as the p-value is far above the typical alpha level of 0.05 used to determine statistical significance. The data supports the null hypothesis that the true mean is 20 mpg, within the confidence interval provided.

b. Independent Two-Sample t-Test

We'll compare the means of mpg between cars with automatic (am = 0) and manual (am = 1) transmissions.

```
auto_mpg <- mtcars$mpg[mtcars$am == 0]
manual_mpg <- mtcars$mpg[mtcars$am == 1]
t.test(auto_mpg, manual_mpg, var.equal = TRUE)
#>
#>  Two Sample t-test
#>
#> data:  auto_mpg and manual_mpg
#> t = -4.1061, df = 30, p-value = 0.000285
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  -10.84837  -3.64151
```

```
#> sample estimates:
#> mean of x mean of y
#>  17.14737  24.39231
```

The results of this two-sample t-test indicate that the average mpg for cars with automatic transmissions significantly differs from those with manual transmissions, as the p-value (0.000285) is well below the alpha level of 0.05 typically used for determining statistical significance. The data strongly support the alternative hypothesis that there is a true difference in means, with manual transmission cars averaging higher mpg (24.39 mpg) compared to automatics (17.15 mpg), as reflected within the confidence interval provided.

c. Paired t-Test

Let's use sleep data

```
# Load the dataset
data(sleep)

# Perform the paired t-test comparing the effects of two drugs
t_test_result <- t.test(extra ~ group, data = sleep, paired = TRUE)

# Print the results
print(t_test_result)
#>
#>  Paired t-test
#>
#> data:  extra by group
#> t = -4.0621, df = 9, p-value = 0.002833
#> alternative hypothesis: true mean difference is not equal to 0
#> 95 percent confidence interval:
#>  -2.4598858 -0.7001142
#> sample estimates:
#> mean difference
#>           -1.58
```

The results of this paired t-test suggest that there is a statistically significant difference in the `extra` sleep effects between the two treatment groups, as the p-value (0.002833) is well below the typical alpha level of 0.05 used for determining statistical significance. The data strongly support the alternative hypothesis that the true mean difference in sleep effects is not equal to zero, with an average mean difference of -1.58 hours. This difference indicates that one treatment group experienced a greater increase in sleep duration compared to the other, as confirmed by the confidence interval ranging from -2.46 to -0.70 hours.

d. One-Way ANOVA

Test if there are differences in mpg across different levels of the number of

cylinders (cyl).

```
anova_model <- aov(mpg ~ factor(cyl), data = mtcars)
summary(anova_model)
#>             Df Sum Sq Mean Sq F value   Pr(>F)
#> factor(cyl)  2  824.8   412.4    39.7 4.98e-09 ***
#> Residuals   29  301.3    10.4
#> ---
#> Signif. codes:
#> 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA analysis clearly shows that the number of cylinders in a vehicle significantly affects its fuel efficiency, with different cylinder groups exhibiting notably different mpg. This finding is robust, with very strong statistical significance, suggesting that engine size, as indicated by the number of cylinders, is a key factor influencing a car's fuel consumption. This information can be vital for both consumers seeking fuel-efficient vehicles and manufacturers aiming to improve vehicle designs.

e. Repeated Measures ANOVA

```
# Load the CO2 dataset from the datasets package
data(CO2)

# Check the structure of the data
str(CO2)
#> Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame':   84 obs
#>  $ Plant    : Ord.factor w/ 12 levels "Qn1"<"Qn2"<"Qn3"<..: 1 1 1 1 1 1 1 2 2 2 ...
#>  $ Type     : Factor w/ 2 levels "Quebec","Mississippi": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ Treatment: Factor w/ 2 levels "nonchilled","chilled": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ conc     : num  95 175 250 350 500 675 1000 95 175 250 ...
#>  $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7 13.6 27.3 37.1 ...
#>  - attr(*, "formula")=Class 'formula'  language uptake ~ conc | Plant
#>   .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
#>  - attr(*, "outer")=Class 'formula'  language ~Treatment * Type
#>   .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
#>  - attr(*, "labels")=List of 2
#>   ..$ x: chr "Ambient carbon dioxide concentration"
#>   ..$ y: chr "CO2 uptake rate"
#>  - attr(*, "units")=List of 2
#>   ..$ x: chr "(uL/L)"
#>   ..$ y: chr "(umol/m^2 s)"

# Load necessary package for analysis
#install.packages("nlme")
library(nlme)  # for linear mixed-effects models
```

```r
# Fit a repeated measures model
# Treat 'Plant' as a random effect to account for measurements from the same plant
model <- lme(uptake ~ Type * Treatment, random = ~ 1 | Plant, data = CO2)

# Summary of the model
summary(model)
#> Linear mixed-effects model fit by REML
#>   Data: CO2
#>        AIC      BIC    logLik
#>   584.0375 598.3297 -286.0188
#>
#> Random effects:
#>  Formula: ~1 | Plant
#>          (Intercept) Residual
#> StdDev: 0.0004510923 8.005933
#>
#> Fixed effects:  uptake ~ Type * Treatment
#>                                    Value Std.Error DF
#> (Intercept)                     35.33333  1.747038 72
#> TypeMississippi                 -9.38095  2.470685  8
#> Treatmentchilled                -3.58095  2.470685  8
#> TypeMississippi:Treatmentchilled -6.55714  3.494076  8
#>                                   t-value p-value
#> (Intercept)                     20.224710  0.0000
#> TypeMississippi                 -3.796904  0.0053
#> Treatmentchilled                -1.449377  0.1853
#> TypeMississippi:Treatmentchilled -1.876646  0.0974
#>  Correlation:
#>                                 (Intr) TypMss Trtmnt
#> TypeMississippi                 -0.707
#> Treatmentchilled                -0.707  0.500
#> TypeMississippi:Treatmentchilled  0.500 -0.707 -0.707
#>
#> Standardized Within-Group Residuals:
#>        Min         Q1        Med         Q3        Max
#> -2.8044677 -0.4526405  0.2706326  0.7210426  1.3299660
#>
#> Number of Observations: 84
#> Number of Groups: 12

# Anova table for the model
anova(model)
#>                numDF denDF  F-value p-value
#> (Intercept)        1    72 970.5351  <.0001
#> Type               1     8  52.5086  0.0001
```

```
#> Treatment          1     8  15.4164  0.0044
#> Type:Treatment     1     8   3.5218  0.0974
```

The model provides strong evidence that both the type of plant and whether it is chilled significantly affect CO2 uptake, independently. The lack of a significant interaction suggests that the effect of chilling does not differ between types in the way that might have been expected.

2. Tests for Comparing Medians

a. Mann-Whitney U Test

Comparing mpg between cars with 4 and 6 cylinders.

```
mpg_4 <- mtcars$mpg[mtcars$cyl == 4]
mpg_6 <- mtcars$mpg[mtcars$cyl == 6]
wilcox.test(mpg_4, mpg_6)
#> Warning in wilcox.test.default(mpg_4, mpg_6): cannot
#> compute exact p-value with ties
#>
#>  Wilcoxon rank sum test with continuity correction
#>
#> data:  mpg_4 and mpg_6
#> W = 76.5, p-value = 0.0006658
#> alternative hypothesis: true location shift is not equal to 0
```

The result of the Wilcoxon rank sum test strongly suggests that the median mpg values for cars with 4 cylinders differ significantly from those with 6 cylinders in the mtcars dataset. Given the very low p-value, it is likely that 4-cylinder cars either achieve higher or lower mpg compared to 6-cylinder cars, depending on the direction of the rank sums (not specified here but typically inferred from the data setup). This finding is crucial for automotive studies focusing on fuel efficiency based on engine size, providing evidence that engine size (as represented by cylinder count) may impact fuel economy.

b. Wilcoxon Signed-Rank Test

Again, a hypothetical example for paired data.

```
# Extracting the groups

group1 <- sleep$extra[sleep$group == 1]
group2 <- sleep$extra[sleep$group == 2]

# Wilcoxon Signed-Rank Test
wilcox_test_results <- wilcox.test(group1, group2, paired = TRUE)
#> Warning in wilcox.test.default(group1, group2, paired =
#> TRUE): cannot compute exact p-value with ties
#> Warning in wilcox.test.default(group1, group2, paired =
```

```
#> TRUE): cannot compute exact p-value with zeroes

# Print the results
print(wilcox_test_results)
#>
#>  Wilcoxon signed rank test with continuity correction
#>
#> data:  group1 and group2
#> V = 0, p-value = 0.009091
#> alternative hypothesis: true location shift is not equal to 0
```

The results of the Wilcoxon signed-rank test suggest a statistically significant difference between the two groups tested, with a p-value indicating strong evidence against the null hypothesis of no difference. This significant finding implies that the treatment or condition represented by these two groups had a different impact on the variable measured (extra sleep hours), with the direction of effect (which group had more sleep) needing further description from the data setup. This analysis is particularly useful in clinical or psychological studies where the normality assumption may not hold, and robust, non-parametric methods are required.

```
x <- c(1.83,  0.50,  1.62,  2.48, 1.68, 1.88, 1.55, 3.06, 1.30)
y <- c(0.878, 0.647, 0.598, 2.05, 1.06, 1.29, 1.06, 3.14, 1.29)
wilcox.test(x, y, paired = TRUE, alternative = "greater")
#>
#>  Wilcoxon signed rank exact test
#>
#> data:  x and y
#> V = 40, p-value = 0.01953
#> alternative hypothesis: true location shift is greater than 0
```

c. Kruskal-Wallis Test

Comparing mpg across different cylinder groups.

```
kruskal.test(mpg ~ factor(cyl), data = mtcars)
#>
#>  Kruskal-Wallis rank sum test
#>
#> data:  mpg by factor(cyl)
#> Kruskal-Wallis chi-squared = 25.746, df = 2, p-value
#> = 2.566e-06
```

The Kruskal-Wallis test conclusively shows that the number of cylinders is a significant factor in determining a car's miles per gallon, with the differences in mpg across groups being statistically significant. This insight can inform decisions related to car manufacturing and consumer choice, particularly in

contexts where fuel efficiency is a critical concern.

d. Friedman Test

```
RoundingTimes <-
matrix(c(5.40, 5.50, 5.55,
         5.85, 5.70, 5.75,
         5.20, 5.60, 5.50,
         5.55, 5.50, 5.40,
         5.90, 5.85, 5.70,
         5.45, 5.55, 5.60,
         5.40, 5.40, 5.35,
         5.45, 5.50, 5.35,
         5.25, 5.15, 5.00,
         5.85, 5.80, 5.70,
         5.25, 5.20, 5.10,
         5.65, 5.55, 5.45,
         5.60, 5.35, 5.45,
         5.05, 5.00, 4.95,
         5.50, 5.50, 5.40,
         5.45, 5.55, 5.50,
         5.55, 5.55, 5.35,
         5.45, 5.50, 5.55,
         5.50, 5.45, 5.25,
         5.65, 5.60, 5.40,
         5.70, 5.65, 5.55,
         6.30, 6.30, 6.25),
       nrow = 22,
       byrow = TRUE,
       dimnames = list(1 : 22,
                       c("Round Out", "Narrow Angle", "Wide Angle")))
RoundingTimes
#>    Round Out Narrow Angle Wide Angle
#> 1       5.40         5.50       5.55
#> 2       5.85         5.70       5.75
#> 3       5.20         5.60       5.50
#> 4       5.55         5.50       5.40
#> 5       5.90         5.85       5.70
#> 6       5.45         5.55       5.60
#> 7       5.40         5.40       5.35
#> 8       5.45         5.50       5.35
#> 9       5.25         5.15       5.00
#> 10      5.85         5.80       5.70
#> 11      5.25         5.20       5.10
#> 12      5.65         5.55       5.45
#> 13      5.60         5.35       5.45
```

```
#> 14      5.05          5.00       4.95
#> 15      5.50          5.50       5.40
#> 16      5.45          5.55       5.50
#> 17      5.55          5.55       5.35
#> 18      5.45          5.50       5.55
#> 19      5.50          5.45       5.25
#> 20      5.65          5.60       5.40
#> 21      5.70          5.65       5.55
#> 22      6.30          6.30       6.25
friedman.test(RoundingTimes)
#>
#>  Friedman rank sum test
#>
#> data:  RoundingTimes
#> Friedman chi-squared = 11.143, df = 2, p-value =
#> 0.003805
```

The significant Friedman test result suggests that the conditions or treatments applied in the RoundingTimes study have differing effects, which are statistically notable. This finding may lead to further investigation into which specific treatments differ from each other and how these differences might be exploited or managed in practical applications, such as clinical, psychological, or educational settings where such treatments or interventions are used.

3. Tests for Proportions

a. Chi-Square Test of Independence

Testing if transmission type (am) is independent of engine cylinders (cyl).

```
table_data <- table(mtcars$am, mtcars$cyl)
chisq.test(table_data)
#> Warning in chisq.test(table_data): Chi-squared
#> approximation may be incorrect
#>
#>  Pearson's Chi-squared test
#>
#> data:  table_data
#> X-squared = 8.7407, df = 2, p-value = 0.01265
```

The results of the Pearson's Chi-squared test suggest a significant relationship between the type of transmission and the number of cylinders in the vehicles. This finding implies that certain transmission types might be more or less common in vehicles with different numbers of cylinders, potentially reflecting design preferences, performance characteristics, or market trends specific to certain types of vehicles. This insight could be valuable for automotive manufacturers and marketers who are targeting specific segments of the car market.

b. Fisher's Exact Test

Let's create a hypothetical dataset that is suitable for this test

```
#
# Data: Drug success (Yes, No) by Treatment group (Drug, Placebo)
drug_data <- matrix(c(4, 1, 1, 3), ncol = 2, byrow = TRUE,
                    dimnames = list(c("Drug", "Placebo"),
                                    c("Success", "Failure")))

drug_data
#>         Success Failure
#> Drug          4       1
#> Placebo       1       3
# Perform Fisher's Exact Test
fisher_results <- fisher.test(drug_data)

# Print the results
print(fisher_results)
#>
#>  Fisher's Exact Test for Count Data
#>
#> data:  drug_data
#> p-value = 0.2063
#> alternative hypothesis: true odds ratio is not equal to 1
#> 95 percent confidence interval:
#>    0.3071304 776.3482393
#> sample estimates:
#> odds ratio
#>   8.355086
```

The results suggest that while there might be a difference in the odds of the event occurring between the two groups, the data do not provide strong enough evidence to assert that there is a statistically significant association between the groups under study. The wide confidence interval for the odds ratio further underscores the need for cautious interpretation of the odds ratio estimate. More data or additional studies might be required to clarify the nature of the relationship between these groups.

c. One-Proportion Z-Test

Testing if the proportion of cars with more than 4 cylinders is different from 50%.

```
prop.test(sum(mtcars$cyl > 4), nrow(mtcars), p = 0.5)
#>
#>  1-sample proportions test with continuity correction
#>
```

```
#> data:  sum(mtcars$cyl > 4) out of nrow(mtcars), null probability 0.5
#> X-squared = 2.5312, df = 1, p-value = 0.1116
#> alternative hypothesis: true p is not equal to 0.5
#> 95 percent confidence interval:
#>  0.4677478 0.8082695
#> sample estimates:
#>       p
#> 0.65625
```

The results suggest that the proportion of cars with more than 4 cylinders in the mtcars dataset does not significantly differ from the hypothesized 50%. The p-value indicates that the observed difference could reasonably occur by chance under the null hypothesis. The confidence interval includes the null value (0.5), further supporting this conclusion. This finding implies that there may not be a strong bias towards cars with more than 4 cylinders in the mtcars dataset, although the observed proportion leans slightly towards a higher number of cylinders. More data or a larger sample might provide clearer insights or more definitive evidence regarding the distribution of cylinder numbers in cars.

d. Two-Proportion Z-Test

Comparing proportion of manual vs automatic cars that are 6-cylinder.

```
manual_six <- sum(mtcars$cyl == 6 & mtcars$am == 1)
auto_six <- sum(mtcars$cyl == 6 & mtcars$am == 0)
prop.test(c(manual_six, auto_six), c(sum(mtcars$am == 1), sum(mtcars$am == 0)))
#> Warning in prop.test(c(manual_six, auto_six),
#> c(sum(mtcars$am == 1), sum(mtcars$am == : Chi-squared
#> approximation may be incorrect
#>
#>  2-sample test for equality of proportions with
#>  continuity correction
#>
#> data:  c(manual_six, auto_six) out of c(sum(mtcars$am == 1), sum(mtcars$am == 0))
#> X-squared = 2.8616e-32, df = 1, p-value = 1
#> alternative hypothesis: two.sided
#> 95 percent confidence interval:
#>  -0.2933577  0.3338435
#> sample estimates:
#>    prop 1    prop 2
#> 0.2307692 0.2105263
```

Given the peculiar results, especially the p-value and chi-squared statistic, it would be prudent to double-check the input data and consider whether the test assumptions are met or if a different statistical approach might be more appropriate. If the data inputs are correct and the assumptions met, the findings would suggest that transmission type does not significantly influence whether a

car has 6 cylinders in the mtcars dataset. This lack of difference could be important for automotive studies examining the relationship between transmission type and engine size, though the unusual statistical outputs warrant a careful review of the data and method.

```
smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)
#>
#>  4-sample test for equality of proportions without
#>  continuity correction
#>
#> data:  smokers out of patients
#> X-squared = 12.6, df = 3, p-value = 0.005585
#> alternative hypothesis: two.sided
#> sample estimates:
#>    prop 1    prop 2    prop 3    prop 4
#> 0.9651163 0.9677419 0.9485294 0.8536585
```

4. Correlation Tests

a. Pearson Correlation Coefficient

Correlation between mpg and wt (weight).

```
cor.test(mtcars$mpg, mtcars$wt, method = "pearson")
#>
#>  Pearson's product-moment correlation
#>
#> data:  mtcars$mpg and mtcars$wt
#> t = -9.559, df = 30, p-value = 1.294e-10
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  -0.9338264 -0.7440872
#> sample estimates:
#>        cor
#> -0.8676594
```

The findings from the Pearson correlation test provide clear evidence that an increase in car weight is associated with a decrease in miles per gallon in the mtcars dataset. This relationship is both strong and statistically significant, with nearly no chance of occurring due to random variation in the sample. Such insights are crucial for automotive design and consumer choice, particularly in discussions around fuel efficiency and vehicle performance optimization.

b. Spearman's Rank Correlation

Correlation between mpg and hp (horsepower).

```
cor.test(mtcars$mpg, mtcars$hp, method = "spearman")
#> Warning in cor.test.default(mtcars$mpg, mtcars$hp, method =
#> "spearman"): Cannot compute exact p-value with ties
#>
#>  Spearman's rank correlation rho
#>
#> data:  mtcars$mpg and mtcars$hp
#> S = 10337, p-value = 5.086e-12
#> alternative hypothesis: true rho is not equal to 0
#> sample estimates:
#>       rho
#> -0.8946646
```

The test results suggest that cars with higher horsepower tend to have lower fuel efficiency, as measured by miles per gallon, in the mtcars dataset. This finding could be useful for automotive manufacturers and buyers who prioritize fuel efficiency. The high degree of correlation provides robust evidence that increasing horsepower in vehicle design typically comes at the expense of fuel economy. This relationship is an important consideration for both engineering and marketing strategies in the automotive industry.

c. Kendall's Tau

Correlation between mpg and disp (displacement).

```
cor.test(mtcars$mpg, mtcars$disp, method = "kendall")
#> Warning in cor.test.default(mtcars$mpg, mtcars$disp, method
#> = "kendall"): Cannot compute exact p-value with ties
#>
#>  Kendall's rank correlation tau
#>
#> data:  mtcars$mpg and mtcars$disp
#> z = -6.1083, p-value = 1.007e-09
#> alternative hypothesis: true tau is not equal to 0
#> sample estimates:
#>       tau
#> -0.7681311
```

The significant results from Kendall's test confirm that increases in engine displacement are associated with decreases in fuel efficiency across the cars sampled in the mtcars dataset. This finding is crucial for understanding how engine size affects fuel economy and can guide both consumer choices and manufacturer designs, especially in contexts where fuel efficiency is a priority. This correlation is an essential consideration in automotive design, influencing decisions about engine specifications in relation to fuel economy objectives.

Exercises Hypothesis Testing

Exercise 1

Test if the average wind speed in the airquality dataset is significantly different from 10 mph.

Exercise 2

Independent Two-Sample t-Test: PlantGrowth Dataset

Compare the means of weight between two groups of plants: ctrl and trt1.

Exercise 3

0.2.1 Paired t-Test

Use the following data to perform a T-test to check if the score after is greater than before.

The data is about 20 students testing before and after studying .

```
before <- c(12.2, 14.6, 13.4, 11.2, 12.7, 10.4, 15.8, 13.9, 9.5, 14.2)
after <- c(13.5, 15.2, 13.6, 12.8, 13.7, 11.3, 16.5, 13.4, 8.7, 14.6)
```

```
data <- data.frame(subject = rep(c(1:10), 2),
                   time = rep(c("before", "after"), each = 10),
                   score = c(before, after))
```

We reject the null hypothesis.

Exercise 4

One-Way ANOVA: ChickWeight Dataset

Test if there are differences in weight across different feed types.

Exercise 5

Repeated Measures ANOVA: Orthodont Dataset

Load Orthodont dataset from the nlme package and then fit a repeated measures model

0.3 Tests for Comparing Medians

Exercise 6

0.3.1 Mann-Whitney U Test: InsectSprays Dataset

Compare the effectiveness of two insect sprays.

Exercise 7

Wilcoxon Signed-Rank Test: Airquality Dataset

Compare Ozone levels from the first half to the second half of the dataset.

Exercise 8

###Kruskal-Wallis Test: ChickWeight Dataset {-}

Compare the weights across different diets using Kruskal-Wallis test.

0.4 Tests for Proportions

Exercise 9

0.4.1 Chi-Square Test of Independence: HairEyeColor Dataset

Test if hair color is independent of eye color.

Exercise 10

One-Proportion Z-Test: Using Airquality Dataset

Suppose you want to analyze the proportion of observations where chick weights exceed 250.

Correlation Tests

Exercise 11

0.4.2 Pearson Correlation Coefficient: Using USJudgeRatings Dataset

Correlation between lawyer Judicial integrity and their judicial Diligence

Exercise 12

0.4.3 Spearman's Rank Correlation: Using USJudgeRatings Dataset

Correlation between lawyers' rating of integrity and their number of contacts with judge

Exercise 13

Kendall's Tau: Using USJudgeRatings Dataset

Correlation between preparation for trial and their diligence

(Optional) Part V: Working with Dates and Times (20 minutes)

Handling date and time data in R

R provides the Date class for dates and the POSIXct and POSIXlt classes for times.

```
# Converting a string to a Date object
date_example <- as.Date("2021-01-01")
print(date_example)
#> [1] "2021-01-01"

# Converting a string to a POSIXct datetime object
datetime_example <- as.POSIXct("2021-01-01 10:00:00", tz = "GMT")
print(datetime_example)
#> [1] "2021-01-01 10:00:00 GMT"
```

Common date and time functions

```
# Extracting parts of a date
year <- format(date_example, "%Y")
month <- format(date_example, "%m")
day <- format(date_example, "%d")
print(paste("Year:", year, "- Month:", month, "- Day:", day))
#> [1] "Year: 2021 - Month: 01 - Day: 01"

# Working with time intervals
start_time <- as.POSIXct("2021-01-01 08:00:00", tz = "GMT")
end_time <- as.POSIXct("2021-01-01 10:00:00", tz = "GMT")
```

```r
time_diff <- difftime(end_time, start_time, units = "hours")
print(paste("Difference in hours:", time_diff))
#> [1] "Difference in hours: 2"

# Loading a dataset with date and time data for exercises
# Using the 'airquality' dataset from the 'datasets' package
data(airquality)
airquality$Date <- as.Date(with(airquality, paste(1973, Month, Day, sep = "-")))
print(head(airquality))
#>   Ozone Solar.R Wind Temp Month Day       Date
#> 1    41     190  7.4   67     5   1 1973-05-01
#> 2    36     118  8.0   72     5   2 1973-05-02
#> 3    12     149 12.6   74     5   3 1973-05-03
#> 4    18     313 11.5   62     5   4 1973-05-04
#> 5    NA      NA 14.3   56     5   5 1973-05-05
#> 6    28      NA 14.9   66     5   6 1973-05-06
```

Exercises:

```r
data(airquality)

airquality$Date <- as.Date(with(airquality, paste(1973, Month, Day, sep = "-")))
print(head(airquality))
#>   Ozone Solar.R Wind Temp Month Day       Date
#> 1    41     190  7.4   67     5   1 1973-05-01
#> 2    36     118  8.0   72     5   2 1973-05-02
#> 3    12     149 12.6   74     5   3 1973-05-03
#> 4    18     313 11.5   62     5   4 1973-05-04
#> 5    NA      NA 14.3   56     5   5 1973-05-05
#> 6    28      NA 14.9   66     5   6 1973-05-06
```

1. Convert the 'Date' column in the 'airquality' dataset to a week day and create a new column 'WeekDay'.

```r
airquality$WeekDay <- weekdays(airquality$Date)
print(head(airquality))
#>   Ozone Solar.R Wind Temp Month Day       Date   WeekDay
#> 1    41     190  7.4   67     5   1 1973-05-01   Tuesday
#> 2    36     118  8.0   72     5   2 1973-05-02 Wednesday
#> 3    12     149 12.6   74     5   3 1973-05-03  Thursday
#> 4    18     313 11.5   62     5   4 1973-05-04    Friday
#> 5    NA      NA 14.3   56     5   5 1973-05-05  Saturday
#> 6    28      NA 14.9   66     5   6 1973-05-06    Sunday
```

2. Calculate the number of days between the first and last measurements in the 'airquality' dataset.

```
date_diff <- difftime(max(airquality$Date), min(airquality$Date), units = "days")
print(paste("Days between first and last measurement:", date_diff))
#> [1] "Days between first and last measurement: 152"
```

0.5 Working with dates and times

0.6 Create and format dates

To create a Date object from a simple character string in R, you can use the as.Date() function. The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 13 January, 1982):

`%Y`: 4-digit year (1982) `%y`: 2-digit year (82) `%m`: 2-digit month (01) `%d`: 2-digit day of the month (13) `%A`: weekday (Wednesday) `%a`: abbreviated weekday (Wed) `%B`: month (January) `%b`: abbreviated month (Jan)

For more information and a full list use `?strptime`

0.7 as.Date()

```
as.Date('2019-06-05',format = '%Y-%m-%d')
#> [1] "2019-06-05"
```

Dates are often stored as integers.

Convert integers to dates by speciying the origin (Day 0).

For example: SAS stores dates at the number of days elapsed since 1 Jan 1960.

```
as.Date(21705, origin = '1960-01-01')
#> [1] "2019-06-05"
```

Exercise 1

Work with the policy_data data set. 1. Convert the start date (debut_pol) and end date (fin_pol) into R Date objects.

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
```

```
#>     intersect, setdiff, setequal, union
policy_data <- read.csv(file = 'Data/PolicyData.csv', sep = ';')
policy_data$start <- as.Date(policy_data$debut_pol, '%d/%m/%Y')
policy_data$end <- as.Date(policy_data$fin_pol, '%d/%m/%Y')
head(policy_data %>% select(c('debut_pol', 'start')))
#>    debut_pol      start
#> 1 14/09/1995 1995-09-14
#> 2 25/04/1996 1996-04-25
#> 3  1/03/1995 1995-03-01
#> 4  1/03/1996 1996-03-01
#> 5 15/01/1997 1997-01-15
#> 6  1/02/1997 1997-02-01
```

format()

```
today <- as.Date('2019-06-05',
                 format = '%Y-%m-%d')
format(today, '%A %d %B %Y')
#> [1] "Wednesday 05 June 2019"
```

Calculate the duration of a contract.

```
policy_duration =
  policy_data$end - policy_data$start
```

You can add and subtract integers from dates.

```
tomorrow = today + 1
print(tomorrow)
#> [1] "2019-06-06"
```

Lubridate

0.8 Access date components

```
# install.packages("lubridate")
library(lubridate)
#>
#> Attaching package: 'lubridate'
#> The following objects are masked from 'package:base':
#>
#>     date, intersect, setdiff, union
```

```
year(today)
#> [1] 2019
```

Other components are: month(), day(), quarter(), ...

0.9 Advanced math

```
today + months(3)
#> [1] "2019-09-05"
```

Other periods are: years() and days().

```
floor_date(today, unit = "month")
#> [1] "2019-06-01"
```

floor_date rounds down to the nearest unit.

In the example convert daily into monthly data.

seq()

```
seq(from = as.Date('2019-01-01'),
    to = as.Date('2019-12-31'),
    by = '1 month')
#>  [1] "2019-01-01" "2019-02-01" "2019-03-01" "2019-04-01"
#>  [5] "2019-05-01" "2019-06-01" "2019-07-01" "2019-08-01"
#>  [9] "2019-09-01" "2019-10-01" "2019-11-01" "2019-12-01"
```

Exercise 2

Visualize the exposure contribution by start month of the contract in the policy_data data set. 1. Add a covariate start_month to the data set. 2. Group the data by start_month. 3. Calculate the exposure within each group. 4. Plot the data.

Part I: Advanced Data Manipulation with dplyr (30 minutes)

Grouping and summarizing data

```
# Loading the dplyr package
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

# Using the 'mtcars' dataset
data(mtcars)

# Example: Grouping by 'cyl' (number of cylinders) and calculating mean mpg (miles per gallon)
grouped_data <- mtcars %>%
  group_by(cyl) %>%
  summarize(mean_mpg = mean(mpg))
print(grouped_data)
#> # A tibble: 3 x 2
#>     cyl mean_mpg
#>   <dbl>    <dbl>
#> 1     4     26.7
#> 2     6     19.7
#> 3     8     15.1
```

Exercise:

1. Group the 'mtcars' dataset by 'gear' and calculate the average horsepower ('hp') for each gear group.

Joining and merging datasets

```
# Creating a sample dataset to join with 'mtcars'
car_names <- data.frame(model = rownames(mtcars), car_type = rep(c("Type A", "Type B",

# Converting row names of 'mtcars' to a column
mtcars$model <- rownames(mtcars)

# Example: Joining 'mtcars' and 'car_names'
joined_data <- left_join(mtcars, car_names, by = "model")
print(head(joined_data))
#>    mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
#> 4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> 5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
#> 6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
#>               model car_type
#> 1         Mazda RX4   Type A
#> 2     Mazda RX4 Wag   Type B
#> 3        Datsun 710   Type C
#> 4    Hornet 4 Drive   Type A
#> 5 Hornet Sportabout   Type B
#> 6           Valiant   Type C
```

Exercise:

2. Create a new dataframe with a subset of columns from 'iris' and merge it with the original 'iris' dataset based on a common column.

Part II: Text Data Processing (30 minutes)

Manipulating and analyzing text data using regular expressions

```
# --- Part II: Text Data Processing (30 minutes) ---

# Load necessary libraries
library(stringr)

## Manipulating and analyzing text data using regular expressions

# Example: Extracting email addresses from a string
text <- "Contact us at support@example.com or feedback@example.net"
emails <- str_extract_all(text, "[[:alnum:]_.]+@[[:alnum:]]+\\.[[:alpha:]]{2,}")
print(emails)
#> [[1]]
#> [1] "support@example.com"  "feedback@example.net"
```

Exercise:

1. Write a regular expression to find all the words starting with 'b' in a given text.

Text mining basics

```
# Load the 'tm' package for text mining
library(tm)
#> Loading required package: NLP

# Example: Basic text mining with a simple corpus
```

```
docs <- Corpus(VectorSource(c("Text mining is awesome", "R is a versatile tool for text
dtm <- DocumentTermMatrix(docs)
inspect(dtm)
#> <<DocumentTermMatrix (documents: 2, terms: 7)>>
#> Non-/sparse entries: 8/6
#> Sparsity           : 43%
#> Maximal term length: 9
#> Weighting          : term frequency (tf)
#> Sample             :
#>     Terms
#> Docs analysis awesome for mining text tool versatile
#>    1        0       1   0      1    1    0         0
#>    2        1       0   1      0    1    1         1
```

Exercise:

2. Create a corpus from your own text data and compute its term frequency-inverse document frequency (tf-idf) matrix.

Part III: Building Predictive Models (30 minutes)

Introduction to machine learning in R

Brief overview of machine learning: Machine learning in R involves using statistical techniques to enable computers to improve at tasks with experience. It encompasses a variety of techniques for classification, regression, clustering, and more.

```
# Load necessary libraries
#install.packages("caret")
library(caret)
#> Loading required package: ggplot2
#> Loading required package: lattice


# Example: Splitting a dataset into training and testing sets
data(iris)
set.seed(123) # Setting seed for reproducibility
trainingIndex <- createDataPartition(iris$Species, p = 0.8, list = FALSE)
trainingData <- iris[trainingIndex, ]
testingData <- iris[-trainingIndex, ]
```

Exercise:

1. Load a different dataset and partition it into training and testing sets.

Creating predictive models with caret

```
# Example: Building a predictive model for the iris dataset
```

```r
model <- train(Species ~ ., data = trainingData, method = "rpart")
print(model)
#> CART
#>
#> 120 samples
#>   4 predictor
#>   3 classes: 'setosa', 'versicolor', 'virginica'
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 120, 120, 120, 120, 120, 120, ...
#> Resampling results across tuning parameters:
#>
#>   cp    Accuracy   Kappa
#>   0.00  0.9398492  0.9086993
#>   0.45  0.7426390  0.6253355
#>   0.50  0.5557896  0.3665192
#>
#> Accuracy was used to select the optimal model using
#>  the largest value.
#> The final value used for the model was cp = 0.

# Predicting using the model
predictions <- predict(model, testingData)
confusionMatrix(predictions, testingData$Species)
#> Confusion Matrix and Statistics
#>
#>             Reference
#> Prediction   setosa versicolor virginica
#>   setosa         10          0         0
#>   versicolor      0         10         2
#>   virginica       0          0         8
#>
#> Overall Statistics
#>
#>                Accuracy : 0.9333
#>                  95% CI : (0.7793, 0.9918)
#>     No Information Rate : 0.3333
#>     P-Value [Acc > NIR] : 8.747e-12
#>
#>                   Kappa : 0.9
#>
#>  Mcnemar's Test P-Value : NA
#>
#> Statistics by Class:
```

```
#>
#>                      Class: setosa Class: versicolor
#> Sensitivity                 1.0000            1.0000
#> Specificity                 1.0000            0.9000
#> Pos Pred Value              1.0000            0.8333
#> Neg Pred Value              1.0000            1.0000
#> Prevalence                  0.3333            0.3333
#> Detection Rate              0.3333            0.3333
#> Detection Prevalence        0.3333            0.4000
#> Balanced Accuracy           1.0000            0.9500
#>                      Class: virginica
#> Sensitivity                    0.8000
#> Specificity                    1.0000
#> Pos Pred Value                 1.0000
#> Neg Pred Value                 0.9091
#> Prevalence                     0.3333
#> Detection Rate                 0.2667
#> Detection Prevalence           0.2667
#> Balanced Accuracy              0.9000
```

Exercise: 2. Build a predictive model for another dataset and evaluate its performance.

```
<!--chapter:end:13-Advanced-R-Part3.Rmd-->

# Part IV: Interactive Dashboards with Shiny (30 minutes) {-}
## Introduction to Shiny for building web-based data dashboards {-}

Shiny is an R package that makes it easy to build interactive web applications (apps) straight fr


```r
# Load the Shiny package
#install.packages("shiny")
library(shiny)
```

The basic structure of a Shiny app involves two main parts:

1. A user interface (UI) script, which controls the layout and appearance of the app.
2. A server script, which contains the instructions to build and rebuild the app based on user input.
```

Creating a simple Shiny app

UI Component: The UI has a sliderInput for selecting the mpg range and a tableOutput to display the filtered data.

Server Logic: The reactive function creates a reactive subset of mtcars based on the selected mpg range. The renderTable function then renders this filtered data as a table in the main panel.

Running the App: As with any Shiny app, shinyApp(ui = ui, server = server) runs the app.

```
# Example: A simple Shiny app for displaying a plot

# Define UI
ui <- fluidPage(
  titlePanel("Simple Shiny App"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("num", "Number of bins:",
                  min = 1, max = 50, value = 30)
    ),
    mainPanel(
       plotOutput("distPlot")
    )
  )
)

# Define server logic
server <- function(input, output) {
  output$distPlot <- renderPlot({
    x <- faithful$eruptions
    bins <- seq(min(x), max(x), length.out = input$num + 1)
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

```
# Define UI
ui <- fluidPage(
  titlePanel("Data Filtering App"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("mpgRange", "Miles per Gallon (mpg):",
```

```
                  min = min(mtcars$mpg), max = max(mtcars$mpg),
                  value = c(min(mtcars$mpg), max(mtcars$mpg))
      )
    ),
    mainPanel(
      tableOutput("filteredData")
    )
  )
)

# Define server logic
server <- function(input, output) {
  filteredData <- reactive({
    mtcars[mtcars$mpg >= input$mpgRange[1] & mtcars$mpg <= input$mpgRange[2], ]
  })

  output$filteredData <- renderTable({
    filteredData()
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Exercise:

1. Modify the example Shiny app to include a dataset of your choice and create a different type of plot.
2. Add additional input options, like checkboxes or dropdown menus, to manipulate the plot.

(Optional) Part V: Version Control and Collaboration (10 minutes)

Using Git and GitHub for version control and collaboration in R projects

Git is a distributed version control system that helps track changes in source code during software development. GitHub is a cloud-based hosting service that lets you manage Git repositories.

Integrating Git with R: 1. Install Git and set up a GitHub account. 2. Configure Git with your username and email. - Use Git Bash or the terminal: git config –global user.name “Your Name” git config –global user.email “your.email@example.com”

3. Initialize a Git repository in an R project:
    - In RStudio, start a new project and select the option to create a Git repository.
4. Basic Git commands:
    - git init: Initialize a new Git repository.
    - git status: Check the status of changes.
    - git add: Add files to the staging area.
    - git commit: Commit changes to the repository.
    - git push: Push changes to a remote repository like GitHub.
    - git pull: Pull updates from a remote repository.
5. Collaborating with GitHub:
    - Fork and clone repositories.
    - Create branches for features or fixes.
    - Use pull requests for code reviews.
    - Merge changes to the main branch.

Exercise:

1. Create a new R project with a Git repository.
2. Make changes in your project, commit them, and push them to a GitHub repository.
3. Collaborate with a colleague or friend by having them fork your repository and submit a pull request.

Bibliography

Katrien Antonio. *Introduction to R*. 2020. URL https://katrienantonio.github.io/intro-R-book/.

Introduction to R. Intro to R Authors, 2024. URL https://intro2r.com/.

Roger D. Peng. *R Programming for Data Science*. 2024. URL https://bookdown.org/rdpeng/rprogdatascience/.

R Core Team. *R Data Import/Export*. R Foundation for Statistical Computing, Vienna, Austria, 2024a. URL https://cran.r-project.org/doc/manuals/r-release/R-data.html.

R Core Team. *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2024b. URL https://cran.r-project.org/doc/manuals/r-release/R-exts.html.

R Core Team. *An Introduction to R*. R Foundation for Statistical Computing, Vienna, Austria, 2024c. URL https://cran.r-project.org/doc/manuals/r-release/R-intro.html.

R Core Team. *R Language Definition*. R Foundation for Statistical Computing, Vienna, Austria, 2024d. URL https://cran.r-project.org/doc/manuals/r-release/R-lang.html.

R Workshops. University of Nebraska-Lincoln, Department of Statistics, 2024. URL https://unl-statistics.github.io/R-workshops/.

Hadley Wickham and Garrett Grolemund. *R for Data Science*. 2017. URL https://r4ds.had.co.nz/.