2.1 Getting Help

To view the help documentation for a function, use either the `?` operator or the `help()` function:

```
?setwd     # Opens help for setwd()
help("lm")  # Alternative way to get help for lm()
```

2.2 Libraries

Libraries in R provide additional functions and features. For example:

- **Syntax:** `library(packageName)`
- **Example:**

```
library(ggplot2)  # Loads the ggplot2 package for data visua
```

2.3 Reading in Data

There are many ways to read data into R. For example, to read data from a CSV file:

```
data <- read.csv("data/yourfile.csv")
```

Other packages such as `readr`, `readxl`, and `openxlsx` offer additional methods and options for reading various data formats.

2.11 Working with File Paths

R provides several functions to work with file paths, allowing you to build, explore, and manipulate directories and files in a platform-independent way. Here are some common functions along with examples and explanations:

2.11.1 Constructing File Paths with `file.path()`

- Purpose:
  `file.path()` constructs file paths by combining directory and file names. It automatically uses the correct file separator for the operating system.
- Example:

```
# Combine folder, subfolder, and file name
full_path <- file.path("data", "raw", "myfile.txt")
print(full_path)
```

This will produce a path like `"data/raw/myfile.txt"` on Unix-like systems or `"data\\raw\\myfile.txt"` on Windows.

2.11.2 Listing Files with `list.files()`

- Purpose:
  `list.files()` returns a character vector of the file names in a specified directory. It can also be used to filter files by pattern.
- Example:

```
# List all files in the "data" directory
files <- list.files(path = "data")
print(files)
```

```
# List only CSV files in the "data" directory
csv_files <- list.files(path = "data", pattern = "\\.csv$")
print(csv_files)
```

This function is useful for dynamically accessing the contents of a directory.

2.11.2.1 Listing Directories with `list.dirs()`

- Purpose:
  `list.dirs()` provides a recursive list of directories (subdirectories included) within a specified path.
- Example:

```
# List all directories within the "data" folder, including s
directories <- list.dirs(path = "data")
print(directories)
```

This function helps you understand the structure of your file system under a particular directory.

2.11.2.2 Creating Directories with `dir.create()`

- Purpose:
  `dir.create()` is used to create new directories. When creating nested directories (sub-directories that do not yet exist), you must set the `recursive` argument to `TRUE`.
- Example:

```
# Create a nested directory structure "data/raw"
dir.create(path = file.path("data", "raw"), recursive = TRUE
```

Setting `recursive = TRUE` ensures that if the parent directory `"data"` does not exist, it will be created along with `"raw"`.

2.11.2.3 Reading and Writing Text Files with `readLines()` and `writeLines()`

- `readLines()`:
  - Purpose:
    Reads text files line by line into a character vector, which is useful for processing or analyzing text.
  - Example:

```
# Read the content of a text file into a vector of lines
lines <- readLines("data/raw/myfile.txt")
print(lines)
```

- `writeLines()`:
  - Purpose:
    Writes a character vector to a text file, with each element being written as a separate line.
  - Example:

```
# Create a vector of text lines
output_lines <- c("This is the first line.", "This is th

# Write the vector to a file called "output.txt"
writeLines(output_lines, con = "data/output/output.txt")
```

These functions are ideal for simple text processing, such as reading logs, configuration files, or exporting simple reports.

2.4 Assigning Values to Objects

In R, you assign values to objects using `<-` (preferred) or `=`:

```
x <- 1              # Assigns the value 1 to x
msg <- "Hello, world!"  # Assigns a character string to msg
print(x)            # Prints the value of x
```

To check an object's type, use the `class()` function:

```
class(x)   # Returns "numeric"
class(msg) # Returns "character"
```

2.5 Basic Computations

R supports basic arithmetic following standard mathematical precedence:

```
2 + 3 * 5      # Multiplication happens before addition
log(10)        # Natural logarithm of 10
4^2            # Exponentiation: 4 squared
sqrt(16)       # Square root of 16
abs(-7)        # Absolute value of -7
```

For integer division and modulus operations:

```
15 %/% 4  # Integer division: returns 3
15 %% 4   # Modulus: returns 3
```

2.6 Working with Vectors

Vectors store multiple values of the same type and are a fundamental data structure in R:

```
x <- c(1, 3, 2, 10, 5)  # Creates a numeric vector
y <- 1:5                # Creates a sequence from 1 to 5
```

Mathematical operations can be applied directly to vectors:

```
y + 2   # Adds 2 to each element
2 * y   # Multiplies each element by 2
x + y   # Adds corresponding elements of x and y
x * y   # Multiplies corresponding elements of x and y
```

```
x^y     # Raises each element of x to the power given in y
```

To extract specific elements from a vector:

```
x[2]      # Returns the second element
x[3:5]    # Returns elements from index 3 to 5
x[-2]     # Returns all elements except the second one
x[x > 3]  # Returns elements greater than 3
```

8 Working with dataframes

For an accompanying video, see `df.mp4`.

8.1 Motivation

While base R is powerful, its syntax can be verbose and inconsistent for everyday data manipulation. The tidyverse offers a suite of packages that work seamlessly together, providing a coherent and intuitive framework for your workflow.

Instead of installing individual packages like **dplyr** or **tidyr** separately, the tidyverse metapackage installs the core packages and recommended dependencies all at once:

```
if (!requireNamespace("tidyverse", quietly = TRUE)) {
  install.packages("tidyverse")
}
library(tidyverse)
```

Printing the entire dataframe (e.g., simply typing `example_df`) would display all rows, which is both inconvenient and time-consuming. Instead, we typically use the `head()` function to preview just the first few rows:

```
head(example_df)
```

8.2.1 Enhancing Dataframe Display with `tibble`

The `tibble` package (a core part of the tidyverse) offers a more concise and informative display. When you print a tibble, it only shows the first 10 rows and includes data type information for each column. This is particularly useful because it lets you quickly verify, for example, whether `timepoint` is numeric or character—information that can be obscured in the default dataframe printout.

8.2.1.1 Installation and Conversion

First, ensure that the `tibble` package is installed and loaded:

```
if (!requireNamespace("tibble", quietly = TRUE)) {
  install.packages("tibble")
}
library(tibble)
```

2.7 Working with Character Vectors

Character vectors store text data:

```
colours <- c("green", "blue", "orange", "yellow", "red")
colours[2]  # Returns "blue"
colours[5]  # Returns "red"
```

Note that character vectors do not support arithmetic operations.

2.8 Matrices

Matrices are two-dimensional arrays where all elements are of the same type:

```
x <- c(1, 3, 2, 10, 5)
y <- 1:5

m1 <- cbind(x, y)  # Column-binding creates a matrix
m2 <- rbind(x, y)  # Row-binding creates a matrix
```

Matrix operations include:

```
t(m1)       # Transposes the matrix
m1 + m2     # Adds corresponding elements
m1 %*% m2   # Matrix multiplication
solve(m1)   # Computes the inverse of a square matrix (if possil
```

2.9 Lists

Lists can store elements of different types, making them flexible for various data:

```
my_list <- list(num = 1, vec = c(1, 2, 3), mat = matrix(1:6, nr
```

Access elements of a list using the `$` operator:

```
my_list$num  # Retrieves the numeric element
```

```
my_list$vec  # Retrieves the vector
my_list$mat  # Retrieves the matrix
```

2.10 Data Frames

Data frames store tabular data, similar to spreadsheets:

```
df <- data.frame(name = c("Alice", "Bob"), age = c(25, 30))
```

Extract data from a data frame:

```
df$name           # Extracts the "name" column
df[1, ]           # Returns the first row
df[df$age > 25, ] # Filters rows where age is greater than 25
```

2.12 Writing Functions

Functions in R allow you to encapsulate operations and reuse code. Define a function using `function()`:

```
increment <- function(x) {
  x + 1  # Adds 1 to the input value and returns it
}

increment(3)  # Returns 4
```

Note that functions always return the last value computed:

```
increment <- function(x) {
  x + 2
  x + 1
}

increment(3)  # Returns 4
```

Functions can also take multiple arguments:

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

add_numbers(2, 5)  # Returns 7
```

A common example is a function to calculate the hypotenuse of a right triangle (you will use this daily):

```
hypotenuse <- function(a, b) {
```

```
  sqrt(a^2 + b^2)  # Uses the Pythagorean theorem
}

hypotenuse(3, 4)  # Returns 5
```

2.13 Basic Statistical Functions

R includes built-in functions for summary statistics:

```
mean(df$age)     # Computes the mean
sd(df$age)       # Computes the standard deviation
summary(df)      # Provides a summary of the data frame
```

To create frequency tables:

```
table(df$name)  # Counts occurrences of each name
```

Convert your dataframe to a tibble using `as_tibble()`:

```
# Check the class before conversion
class(example_df)
```

```
[1] "data.frame"
```

```
# Convert to tibble
example_df <- as_tibble(example_df)

# Check the class after conversion
class(example_df)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Now, simply typing `example_df` will display a neat summary of your data:

```
example_df
```

can also create new dataframes directly as tibbles:

```
new_tbl <- tibble(x = 1:5, y = rnorm(5))
new_tbl
```

A final, very nice feature of `tibbles` is that selecting one column using `df[, "column"]` will return a tibble, rather than a vector. This is typically what we would expect:

```
example_df[, "id"]
```

8.2.2 Exploring Data Columns with `view_cols`

Whilst `tibble`s provide a more informative display, they still only show the first few rows of each column.

Another useful tool is the `view_cols` function from the `UtilsDataRSV` package. This function displays unique entries for each column—always showing any missing values (NAs)—so you can quickly identify anomalies such as typos or unexpected values.

To install the `UtilsDataRSV` package, use the following code:

```
if (!requireNamespace("remotes", quietly = TRUE)) {
  install.packages("remotes")
}
remotes::install_github("SATVILab/UtilsDataRSV")
```

Once installed, you can apply `view_cols` to your tibble to inspect the unique values in each column:

```
UtilsDataRSV::view_cols(example_df)
```

You can also display the dataframe on its side, using `dplyr::glimpse()`:

```
dplyr::glimpse(example_df)
```

8.4 Working with rows and columns

8.4.1.1 Rows

The rows functions select rows (`filter`), order rows (`arrange`), and select distinct rows (`distinct`).

- **filter()**: Keeps rows that meet specified conditions.

```
flights |> filter(month == 1)
```

arrange(): Reorders rows based on column values.

```
flights |> arrange(dep_delay)
```

distinct(): Returns unique rows or unique combinations of specified columns.

```
flights |> distinct(origin, dest)
```

8.4.1.2 Columns

The columns functions create new columns (`mutate`), select columns (`select`), rename columns (`rename`), and relocate columns (`relocate`).

- **select()**: Chooses and reorders specific columns.

```
flights |> select(year, month, day, dep_delay)
```

  - Selecting a range:

```
flights |> select(year:day)
```

  - Selecting all columns except some:

```
flights |> select(!year:arr_delay)
```

mutate(): Creates new columns derived from existing ones.

```
flights |>
  mutate(speed = distance / air_time * 60) |>
  select(distance, air_time, speed)
```

rename(): Changes column names without removing other columns.

```
flights |> rename(YEAR = year)
```

- **relocate()**: Moves columns to a new position within the data frame.

```
flights |> relocate(carrier, flight)
```

Moving the columns before a specified column:

```
flights |> relocate(carrier, flight, .before = year)
```

8.4.1.3 Groups & Summaries

The groups and summaries functions group data (`group_by`) and summarise data within groups (`summarise`).

- **group_by()**: Splits the data into groups based on one or more columns. The grouping is not visible, and does not create multiple dataframes. See `summarise` below for how to use the groups.

```
flights |> group_by(month)
```

summarise(): Computes summary statistics for each group.

```
flights |> group_by(month) |> summarise(avg_delay = mean(dep
```

8.5 Changing the shape of data

pivot_longer():

Converts data from wide to long format by gathering multiple columns into key-value pairs (results in fewer columns, more rows).

```
billboard |>
  pivot_longer(
    cols = starts_with("wk"), # columns to pivot (display a
    names_to = "week",  # new column for the column names
    values_to = "rank", # new column for the values
    values_drop_na = TRUE # drop rows with NA values
  )
```

pivot_wider():

Transforms long data to wide format by spreading key-value pairs across columns (results in more columns, fewer rows).

```
cms_patient_experience |>
  pivot_wider(
    id_cols = c("org_pac_id", "org_nm"), # columns to keep
    names_from = measure_cd, # column to spread (unique entr
    values_from = prf_rate # column to use for values (value
  )
```

8.6 Joining dataframes

Mutating joins add new columns from one data frame to another based on matching key values. They share a common interface:

- **left_join()**: Keeps all rows from the left table and adds matching columns from the right table.

```
# Add full airline names to flights2 data
flights2 |> left_join(airlines)
```

inner_join(): Keeps only rows with matching keys in both tables.

```
# Only keep rows where both x and y have a matching key
df1 |> inner_join(df2)
```

Specifying Keys:

By default, join functions match on columns with the same name. Use `join_by()` to specify different keys:

```
flights2 |> left_join(planes, join_by(tailnum))
```

semi_join(): Keeps rows in `x` that have at least one match in `y` (does not add columns from `y`).

```
# Keep only origin airports that appear in flights2
airports |> semi_join(flights2, join_by(faa == origin))
```

anti_join(): Keeps rows in `x` that have no match in `y`.

```
# Find tail numbers in flights2 that are missing from planes
flights2 |> anti_join(planes, join_by(tailnum)) |> distinct(
```

10 ggplot2

- **Initiating a Plot**:
  Use `ggplot(data, aes(x, y))` to start a plot. This sets up the canvas by specifying the data and how variables are mapped to the x and y axes. Note that without specifying any `geom`s (see below), you will not see any content:

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g
```

Mapping Variables to Visual Properties:
Inside `aes()`, map variables to aesthetics such as:

  - **Color**: Differentiates groups (e.g., `aes(color = species)`).
  - **Shape**: Uses different symbols for groups (e.g., `aes(shape = species)`).
  - **Size**: Can reflect magnitude differences.

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g
  geom_point(aes(color = species, shape = species))
```

Different Plot Types:
Add layers to your plot using geoms. Common examples include:

  - **Scatterplots**: `geom_point()`
  - **Smooth Lines**: `geom_smooth()` (e.g., for adding a best-fit line)
  - **Bar Charts**: `geom_bar()` or `geom_col()`
  - **Histograms and Density Plots**: `geom_histogram()`, `geom_density()`

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g
  geom_point(aes(color = species, shape = species)) +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

Adding Layers:
Combine multiple geoms to add detail to your plot. Layers are added using the `+` operator. (must be at end of line) (can't be on new line)

Custom Labels and Themes:
Enhance your plot with titles, axis labels, and legends using `labs()` and adjust appearance with theme functions.

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g
  geom_point(aes(color = species, shape = species)) +
  geom_smooth(method = "lm") +
  labs(
    title = "Body mass and flipper length",
    subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo
    x = "Flipper length (mm)",
    y = "Body mass (g)",
    color = "Species",
    shape = "Species"
  ) +
  scale_color_colorblind()
```

Splitting Data into Panels:
Use faceting (with `facet_wrap()` or `facet_grid()`) to create subplots based on a categorical variable, making it easier to compare groups.

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g
  geom_point(aes(color = species, shape = species)) +
  facet_wrap(~ island)
```

Exporting Graphics:
Use `ggsave()` to save the most recent plot to a file {#sec-day4-prac}(e.g., PNG, PDF).

```
ggsave("penguin-plot.png")
```

11 Tables

```
```{r}
#| results: asis
# Create a summary table of key statistics by species
penguin_summary <- penguins |>
  group_by(species) |>
  summarise(
    `Flipper length` = round(mean(flipper_length_mm, na.rm = TRU
    `Body Mass` = round(mean(body_mass_g, na.rm = TRUE), 2),
    Count = n(),
    .groups = "drop"
  ) |>
  rename(Species = species)

# Display the table using knitr::kable
kable(
  penguin_summary,
  caption = "Summary statistics for penguin easurements. Values
)
```
```

Values are rounded to 2 decimal places. Abbreviations: FL = Flipper Length (mm), BM = Body Mass (g)."

10.4 Combining Plots with cowplot::plot_grid

When you want to display multiple plots side by side (or in a grid), the **cowplot** package provides the convenient function `plot_grid()`. This function not only aligns axes but also lets you add labels (e.g., "A", "B", ...) so that each subplot is clearly identified. This is especially useful in academic work where figures need to be self-contained and cross-referenced.

Here's an example of combining two ggplot figures:

```
# Install and load cowplot if not already installed
if (!requireNamespace("cowplot", quietly = TRUE)) {
  install.packages("cowplot")
}
```

```
library(cowplot)

# Create two example plots
library(ggplot2)
library(palmerpenguins)
library(ggthemes)

p1 <- ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_
  geom_point(aes(color = species, shape = species)) +
  geom_smooth(method = "lm") +
  labs(title = "Scatterplot: Flipper Length vs Body Mass") +
  scale_color_colorblind() +
  theme_cowplot()  # using cowplot theme for consistent font si

p2 <- ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_m
  geom_point(aes(color = species)) +
  geom_smooth(method = "loess") +
  labs(title = "Scatterplot: Bill Length vs Bill Depth") +
  scale_color_colorblind() +
  theme_cowplot()

# Combine the two plots into one figure with labels
combined_plot <- plot_grid(
  p1, p2,
  labels = c("A", "B"),    # Add subplot labels
  label_size = 16,         # Adjust label size if needed
  align = "hv",            # Align both horizontally and vertica
  ncol = 1
)               # Arrange side by side
```

```
# Print the combined plot
combined_plot
```

10.6 General Guidelines for Plots

There are many guidelines for plots, but these are the main focuses:

- **Plot is readable**: text large enough, colours and shapes distinguishable.
- **Plot is captioned and cross-referenced**: plot caption goes below the plot, and the plot is cross-referenced in the text. It should be cross referenced *before* it appears.
- **Plot is self-contained:** the plot should be understandable without needing to refer to the main text. Achieve this by:
    - Including a caption with:
        - An informative title
        - Additional sentences to describe any important details
        - Closing definitions of abbreviations or symbols used in the plot.
    - Labelling axes and legends clearly
- **Neatly formatted text**:
    - Headings should have a consistent sentence/title case
    - All text should be free of punctuation and programming syntax/formatting (e.g. `Participant_ID`).

8.3 Pipe

- Essentially, the pipe is a rewrite of code, from

```
f(x,y)
```

to

```
x |> f(y)
```

In other words, the object to the left of the pipe (`x`) becomes the first argument to the function on the right (`f`) and `y` becomes its second parameter.

As as silly example, this:

```
test_vec <- 1:5
mean(test_vec[test_vec > 3], trim = 0.5)
```

```
[1] 4.5
```

is equivalent to this:

```
test_vec[test_vec > 3] |>
  mean(trim = 0.5)
```

```
[1] 4.5
```

In terms of `f(x,y)`, `f` is `mean`, `x` is `test_vec[test_vec > 3]` and `y` is `trim = 0.5`.

11 Tables

When it comes to tables in academic writing or reports, clear and consistent formatting is essential.

11.1 General Guidelines

- **Table Styling:**
    - **Avoid Vertical Lines:** Most good style guides recommend that tables do not include vertical lines. Horizontal lines should typically only appear above and below the header row and at the bottom of the table.
    - **Minimal Horizontal Lines:** Use only as many horizontal lines as necessary to separate the header from the data.
    - **Rounding and Significant Figures:** Present numerical data with a limited, preferrably consistent number of decimal/significant places (usually 2 or 3) to enhance readability. Use a functions like or `round()` or `signif()`.
    - **Headings**: Headings should consistently be either title or sentence case, and should not contain punctuation or programming syntax/formatting (e.g. `Participant_ID`). Headings can be bold or not, but should be consistent throughout the document.
- **Captions and Cross-Referencing:**

  Every table should have a concise caption that includes a title and a descriptive sentence. Abbreviations or symbols used within the table must be defined in the caption. In Quarto, assign a label (e.g., `tab-summary`) so that you can cross-reference the table in your text with `@tab-summary`. *Note:* All tables should be referenced in your text before they appear.
- **Self-contained:**:

  Tables should be self-contained in the sense that the reader can understand the table without refering to the main text. This is achieved by including a caption that has an informative detail, uses sentences to describe any important details, and closes with abbreviations or symbols used in the table.