Data Wrangling

```r
library(tidyverse)
```

The Tidyverse

The Tidyverse is a collection of R packages designed for data science that share a common philosophy, grammar, and data structures. It makes working with data in R more consistent and readable.

Core Tidyverse Packages

Package	Purpose
`ggplot2`	Data visualisation
`dplyr`	Data manipulation (filtering, summarising, transforming)
`tidyr`	Tidying data (reshaping, pivoting)
`purrr`	Functional programming / advanced iteration
`readr`	Reading rectangular data files (CSV, TSV, etc.)
`tibble`	Modern version of `data.frame` with better printing and subsetting
`forcats`	Working with factors
`stringr`	Working with character strings

`dplyr` Basics

dplyr aims to provide a function for each basic verb of data manipulation. These verbs can be organised into three categories based on the component of the dataset that they work with:

Rows:

- filter() chooses rows based on column values.
- slice() chooses rows based on location.
- arrange() changes the order of the rows.

Columns:

- select() changes whether or not a column is included.
- rename() changes the name of columns.
- mutate() changes the values of columns and creates new columns.
- relocate() changes the order of the columns.

Groups of rows:

- summarise() collapses a group into a single row.
- The pipe

All of the dplyr functions take a data frame (or tibble) as the first argument.

At the most basic level, you can only alter a tidy data frame in five useful ways: you can reorder the rows (arrange()), pick observations and variables of interest (filter() and select()), add new variables that are functions of existing variables (mutate()), or collapse many values to a summary (summarise()).

Others

- Pipeline operator %>%
- group_by()
- Family of `join` operations

Example

```r
name <- c("A", "B", "C", "D", "E", "F")
rcomputing <- c(45,40,90,94,80,65)
eda <- c(63,66,75,83,80,59)
sl <- c(59,56,91,92,86,67)
marks <- data.frame(name,rcomputing,eda,sl)
head(marks, 3)
```

```
##   name rcomputing eda sl
## 1    A         45  63 59
## 2    B         40  66 56
## 3    C         90  75 91
```

select() function: Select columns

The `select()` function is used to choose specific columns from a data frame (or tibble). This is useful when you want to focus on certain variables or drop the ones you don't need.

```r
# Select specific columns by name
select(marks, name, eda)

# Equivalent using the pipe
marks %>% select(name, eda)
```

```r
# Selecting by position
select(marks, 1)          # select col 1
select(marks, 1:3)        # select cols 1 through 3
select(marks, c(1, 3))    # select cols 1 and 3
select(marks, -1)         # select all cols except column 1
```

```r
marks %>% select(name:eda)    # select all cols between name & eda
marks %>% select(!(name:eda)) # select all cols except those between name & eda
```

- Column names act as positions in `select()`.
  - For example, `select(marks, name)` is equivalent to `select(marks, 1)` if name is the first column.
- Because of this, variables from the surrounding environment are not automatically used inside `select()`.

```r
rprogramming <- 5
select(marks, rprogramming)  # looks for a col called 'rprogramming'
select(marks, 5)             # error if column 5 doesn't exist
```

Contextual variables in helpers

This restriction only applies to bare names (e.g., name, eda, name:eda). When using selection helpers (like starts_with(), ends_with(), etc.), external variables can be used.

```r
specify <- "ing"
select(marks, ends_with(specify))  # selects all columns ending with "ing"
```

You can also select columns based on specific criteria with:

- `starts_with()` - select cols that start with a character string
- `ends_with()` - select cols that end with a character string
- `contains()` - select cols that contain a character string
- `matches()` - select cols that match a regular expression
- `one_of()` - select cols names that are from a group of names

```r
select(marks, starts_with("n"))
marks %>% select(ends_with("ing"))
```

filter() function: Filter rows

- The `filter()` function is used to extract a subset of rows from a data frame (or tibble).
- You provide conditions, and only the rows where those conditions evaluate to TRUE are returned.

```r
# Select rows where rcomputing > 50
filter(marks, rcomputing > 50)

# Multiple conditions (rows where both are TRUE)
marks %>% filter(rcomputing > 50, eda > 75)
```

This is roughly equivalent to base R subsetting:

```r
marks[marks$rcomputing > 50 & marks$eda > 75, ]
```

Equality and logical expressions

```r
filter(marks, eda == 80) # rows where eda is exactly 80
filter(marks, rcomputing > 50 & eda > 60) # multiple logical conditions
```

Using between()

The `between()` helper checks if a value falls within a closed interval (inclusive).

```r
filter(marks, between(rcomputing, 50, 80))
```

This is the same as:

```r
filter(marks, rcomputing >= 50 & rcomputing <= 80)
```

arrange() function: Arrange rows

The arrange() function reorders the rows of a data frame (or tibble) according to one or more variables.

```r
# Arrange rows by eda (ascending, the default)
arrange(marks, eda)

# Equivalent with pipes
marks %>% arrange(eda, sl)
```

- By default, sorting is ascending.
- When multiple columns are given, subsequent columns are used to break ties.
  - In the example above, rows with the same eda value are ordered by sl.

Use desc() to sort in descending order:

```r
marks %>% arrange(desc(eda))
```

slice() function: Choose rows by position

The `slice()` family of functions selects rows based on their integer positions, rather than conditions on values (as with filter()). This makes it useful for sampling, keeping top/bottom rows, or indexing directly.

```r
marks %>% slice(3:5)    # select rows at positions 3 to 5
```

First or last rows:

```r
marks %>% slice_head(n = 2)  # first 2 rows
marks %>% slice_tail(n = 2)  # last 2 rows
```

Random rows:

```r
marks %>% slice_sample(n = 3)        # randomly select 3 rows
marks %>% slice_sample(prop = 0.7)  # randomly select 70% of rows
```

- Use replace = TRUE for bootstrap sampling.
- Add the weight_by argument to perform weighted sampling.

Highest or lowest values:

- Use slice_min() and slice_max() to select rows with the smallest or largest values of a variable.

- Remove missing values first if necessary.

```
marks %>%
  filter(!is.na(eda)) %>%
  slice_max(eda, n = 3)    # top 3 rows by eda
```

rename() function: Change column names

The rename() function changes the names of existing columns.

The syntax is:

```
rename(.data, new_name = old_name)
```

```
# Rename column 'sl' to 'supervised'
rename(marks, supervised = sl)
```

The new name goes on the left, the old name on the right.

Use `rename_with()` when you want to apply a function to rename multiple columns at once (e.g. converting all names to lowercase).

```
marks %>% rename_with(tolower)  # all names to lowercase
marks %>% rename_with(~ gsub(" ", "_", .x))  # replace spaces with underscores
```

You can rename columns selected with `starts_with()`, `ends_with()`, `matches()`, etc.

```
marks %>% rename_with(toupper, starts_with("r"))
# all columns starting with "r" converted to UPPERCASE
```

If the new names are stored in a variable, use setNames() inside rename_with().

```
new_names <- c("student_name", "exploratory", "supervised")
marks %>% rename_with(~ new_names)
```

mutate() function: Add or transform columns

The `mutate()` function creates new variables or transforms existing ones in a data frame (or tibble).

- Unlike `select()`, which works with column names/positions, `mutate()` works with column vectors (the actual values).
- Often used for feature engineering: standardizing, creating flags, applying mathematical/logical transformations.
- Order aware: later mutations can use columns created earlier in the same call.
- Supports logical conditions, boolean operators, and helper arguments like .keep.

1. Create new variables

```
mutate(marks, st.eda = (eda - mean(eda)) / sd(eda))  # standardize eda
```

2. Conditional variables

```
marks <- mutate(marks, pass.rcomp = ifelse(rcomputing < 50, "fail", "pass"))
```

3. Chain multiple mutations

```
marks %>%
  mutate(pass.rcomp = ifelse(rcomputing < 50, "fail", "pass"),
         log_eda = log(eda))
```

4. Keep only new variables

Use `.keep = "none"` to return just the new column(s):

```
mutate(marks, pass.rcomp = ifelse(rcomputing < 50, "fail", "pass"),
       .keep = "none")
```

5. Use existing mutations immediately

```r
marks %>%
  mutate(total = rcomputing + eda,
         avg   = total / 2)    # 'avg' uses 'total' created above
```

6. Logical and boolean transformations

```r
marks %>%
  mutate(high_score = eda > 75 & rcomputing > 60)
```

7. Programmatic renaming with across()

Combine `mutate()` with `across()` for applying transformations to multiple columns at once:

```r
marks %>%
  mutate(across(c(eda, rcomputing), log, .names = "log_{.col}"))
```

relocate() function: Change column order

The relocate() function reorders columns in a data frame (or tibble).

```r
relocate(.data, ..., .before = NULL, .after = NULL)
```

- .data: the data frame or tibble.
- …: columns to move (by name, range, or helper functions).
- .before: move the selected columns before a given column.
- .after: move the selected columns after a given column.

```r
marks %>% relocate(rcomputing:eda, .before = name)
```

Move to front or back

```r
marks %>% relocate(eda)  # move 'eda' to the front
marks %>% relocate(eda, .after = last_col())  # move 'eda' to the end
```

Reorder multiple columns

```r
marks %>% relocate(c(name, sl), .before = rcomputing)
```

Use helper functions

```r
marks %>% relocate(starts_with("r"), .after = name)  # move all 'r*' cols after 'name'
```

summarise() function: Summarise values

The `summarise()` function collapses a data frame into a single row or grouped summaries by computing summary statistics such as mean, sum, min, max, count, etc.

```r
marks %>% summarise(eda_mean = mean(eda, na.rm = TRUE))
```

```
##   eda_mean
## 1       71
```

Summarise all numeric columns

`summarise_all()` applies a summary function to every column.

```r
marks %>%
  select_if(is.numeric) %>%             # keep numeric columns
  summarise_all(~ sum(., na.rm = TRUE)) # sum each column, ignoring NA
```

- ~ defines a formula.
- . represents the current column's data.

Using across() (modern approach)

`summarise_all()` is superseded. The recommended way is to use `across()` inside summarise().

```r
marks %>%
  summarise(across(where(is.numeric), ~ sum(., na.rm = TRUE)))
```

Multiple summaries at once

```r
marks %>%
  summarise(across(where(is.numeric),
                   list(mean = ~ mean(., na.rm = TRUE),
                        sd   = ~ sd(., na.rm = TRUE))))
```

Grouped summaries

Combine with `group_by()` to summarise by category:

```r
marks %>%
  group_by(pass.rcomp) %>%
  summarise(eda_mean = mean(eda, na.rm = TRUE),
            count = n())
```

The pipeline operator %>%

The pipeline operator (`%>%`) from magrittr (and used heavily in dplyr) allows you to chain multiple operations in a clear, readable sequence.

- It passes the output of one function as the first argument of the next function.
- This avoids nested function calls like: `third(second(first(x)))`
- Instead, we write: `x %>% first() %>% second() %>% third()`

```r
marks %>%
  mutate(year = 2020) %>%
  filter(eda > 60)
```

Multi-step workflows

```r
marks %>%
  filter(eda > 60) %>%
  mutate(pass = ifelse(rcomputing > 50, "pass", "fail")) %>%
  arrange(desc(eda)) %>%
  summarise(avg_eda = mean(eda, na.rm = TRUE))
```

group_by() function: Group data by a variable

The `group_by()` function is used to create groups within a data frame, so that subsequent operations (e.g. `summarise()`, `mutate()`, `filter()`) are applied within each group rather than across the entire dataset.

- group_by() does not change the data immediately — it just defines the grouping structure.
- Use `ungroup()` to remove the grouping after operations are complete.

```r
marks %>%
  group_by(pass.rcomp) %>%
  summarise(eda_mean = mean(eda, na.rm = TRUE)) # separate mean for each pass.rcomp value
```

Inspect groups:

```r
marks %>% distinct(pass.rcomp)  # see the unique group values
```

Grouped mutation:

```r
marks %>%
  group_by(pass.rcomp) %>%
  mutate(max_eda = max(eda, na.rm = TRUE)) # col with max eda in each group
```

Ungroup after operations:

```r
marks %>%
  group_by(pass.rcomp) %>%
```

```
  summarise(eda_mean = mean(eda, na.rm = TRUE)) %>%
  ungroup() # ensures further operations are applied to the entire dataset
```

Multiple grouping variables:

```
marks %>%
  group_by(pass.rcomp, year) %>%
  summarise(avg_eda = mean(eda, na.rm = TRUE))
```

count() function: Count occurrences

The `count()` function counts the number of occurrences of each unique value in a column, or a combination of columns.

By default, it returns:

1. A column with the unique values from the specified variable(s)
2. A column n showing the frequency of each value

```
marks %>% count(pass.rcomp)
# Counts how many rows belong to each value of pass.rcomp
```

Count multiple columns:

```
marks %>% count(pass.rcomp, year)
# Counts occurrences for combinations of values across multiple columns
```

Sort counts:

```
marks %>% count(pass.rcomp, sort = TRUE)
# Orders results by n in descending order (most frequent first)
```

Weighted counts:

```
marks %>% count(pass.rcomp, wt = eda)
# Instead of counting rows, sums the eda values within each group
```

Equivalent `group_by()` + `summarise()`:

```
marks %>%
  group_by(pass.rcomp) %>%
  summarise(n = n())
# Shows that count() is a shortcut for grouped summarisation.
```

Family of join operations

The `dplyr` package provides a set of join functions to merge two data frames (or tibbles) based on one or more key columns.

Syntax:

```
joined_data <- join_type(df1, df2, by = "key_column")
```

- `by` specifies the column(s) used to match rows between the two tibbles.
- If the column names are identical in both tables, `by` can be omitted.
- If the columns have different names, use a named vector:

```
by = c("df1_col" = "df2_col")
```

```
df1 <- tibble(id = c(1, 2, 3), name = c("Alice", "Bob", "Charlie"))
df2 <- tibble(id = c(2, 3, 4), score = c(90, 85, 88))

inner_join(df1, df2, by = "id")
left_join(df1, df2, by = "id")
right_join(df1, df2, by = "id")
full_join(df1, df2, by = "id")
semi_join(df1, df2, by = "id")
anti_join(df1, df2, by = "id")
```

Joining on multiple columns:

```
full_join(df1, df2, by = c("col1", "col2", "col3"))
```

- Combines rows where all specified columns match.
- Non-matching rows are kept with NA in columns that don't exist in the other table.

Joining columns with different names:

```
full_join(df1, df2, by = c("col1" = "colX", "col2" = "colY"))
# Maps col1 in df1 to colX in df2, and col2 in df1 to colY in df2
```

Note:

- Always check column names before joining.
- Use `distinct()` if necessary to remove duplicates before joining.
- Use `select()` to include only the relevant key columns if the datasets are wide

Ranking Functions in dplyr

Ranking functions assign relative positions to numeric values, with different strategies for handling ties. These are useful for ranking performance scores, sales, or any numeric variable.

1. min_rank()

- Assigns ranks to values.
- Tied values receive the same rank.
- The smallest possible rank is assigned to tied values, and subsequent ranks are skipped.

```
x <- c(10, 20, 20, 30)
min_rank(x)
# [1] 1 2 2 4
```

2. dense_rank()

- Similar to min_rank().
- Tied values receive the same rank, but no ranks are skipped.

```
dense_rank(x)
# [1] 1 2 2 3
```

3. row_number()

- Assigns unique ranks to each value, even when tied.

```
row_number(x)
# [1] 1 2 3 4
```

4. percent_rank()

- Computes the relative percentile rank of each value, ranging from 0 to 1.
- 0 corresponds to the smallest value, 1 to the largest.

```
percent_rank(x)
# [1] 0.0 0.3333 0.3333 1.0
```

5. ntile()

- Divides data into n quantiles (e.g. quartiles, deciles).
- Assigns each value a rank corresponding to its quantile.

```
ntile(x, 2)  # Divide into 2 quantiles
# [1] 1 1 2 2
```

Function	Tie Handling	Output Type	Notes
`min_rank()`	Smallest rank for ties, skips next	Integer rank	Skips numbers after ties
`dense_rank()`	Same rank for ties, no skipping	Integer rank	Consecutive ranks
`row_number()`	Unique rank for each value	Integer rank	Tied values get arbitrary order
`percent_rank()`	Same rank for ties	Numeric 0–1	Relative percentile
`ntile(n)`	Groups into n quantiles	Integer rank 1–n	Useful for categories

NYCFlights13 Dataset

On-time data for all flights that departed NYC (i.e. JFK, LGA or EWR) in 2013.

- year, month, day Date of departure.
- dep_time, arr_time: Actual departure and arrival times (format HHMM or HMM), local tz.
- sched_dep_time, sched_arr_time: Scheduled departure and arrival times (format HHMM or HMM), local tz.
- dep_delay, arr_delay: Departure and arrival delays, in minutes. Negative times represent early departures/arrivals.
- carrier: Two letter carrier abbreviation. See airlines to get name.
- flight: Flight number.
- tailnum: Plane tail number. See planes for additional metadata.
- origin, dest: Origin and destination. See airports for additional metadat
- air_time: Amount of time spent in the air, in minutes.
- distance: Distance between airports, in miles.
- hour, minute: Time of scheduled departure broken into hour and minutes.
- time_hour: Scheduled date and hour of the flight as a POSIXct date. Along with origin, can be used to join flights data to weather data.

```
library(nycflights13)
head(flights, 3)
```

```
## # A tibble: 3 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

1. Find all flights that

a) Had an arrival delay of two or more hours

```
filter(flights, arr_delay >= 120)
```

b) Flew to Houston (IAH or HOU)

```
filter(flights, dest == "IAH" | dest == "HOU")
```

c) Were operated by United, American, or Delta

```
filter(flights, carrier %in% c("UA", "AA", "DL"))
# OR
filter(flights, carrier == "UA" | carrier == "AA" | carrier == "DL")
```

d) Departed in summer (July, August, and September)

```
filter(flights, month %in% c(7, 8, 9))
```

e) Arrived more than two hours late, but didn’t leave late

```
filter(flights, arr_delay > 120 & dep_delay == 0)
```

f) Were delayed by at least an hour, but made up over 30 minutes in flight

```
filter(flights, dep_delay >= 60 & arr_delay <= 30)
```

g) Departed between midnight and 6am (inclusive)

```
filter(flights, dep_time >= 0 & dep_time <= 600)
```

2. Use between() to simplify the code needed to answer the previous questions.

```
filter(flights, between(dep_time, 0, 600))
filter(flights, between(month, 7, 9))
```

3. How many flights have a missing dep_time? What other variables are missing? What might these rows represent?

```
nrow(filter(flights, is.na(dep_time)))

flights %>%
  summarise_all(~ sum(is.na(.)))
```

4. Sort the flights according to day, month and year

```
arrange(flights, day, month, year)
```

5. Sort the flights using the arrival time in a descending order.

```
arrange(flights, desc(arr_time))
```

6. How could you use arrange() to sort all missing values to the start?

```
arrange(flights, desc(is.na(arr_time)))
```

7. Sort flights to find the most delayed flights. Find the flights that left earliest.

```
arrange(flights, desc(dep_delay))  # most delayed flights at the top
arrange(flights, dep_delay)  # flights that left the earliest (neg delay times)
```

8. Sort flights to find the fastest (highest speed) flights.

```
arrange(flights, air_time/distance)
```

9. Which flights travelled the farthest? Which travelled the shortest?

```
head(arrange(flights, desc(distance)), 3)
head(arrange(flights, distance), 3)
```

10. Select all columns in the flights dataframe between year and day (inclusive).

```
select(flights, year:day)
```

11. Select all columns except those from year to day (inclusive).

```
select(flights, (!year:day))
```

12. Rename the tail_num variable in flights dataframe with tailnum.

```
rename(flights, tailnum = tail_num)
```

13. Using the pipeline operator do the following:

a) Select all columns in the flights dataframe between year and day (inclusive).

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

b) Select all columns that ends with delay and time.

```
flights %>%
  select(ends_with("delay"), ends_with("time"))
```

c) Select the distance and air_time variables.

```
flights %>%
  select(distance, air_time)
```

d) Create a gain/loss travel time for each flight.

```
flights %>%
  mutate(gain_loss = ifelse((arr_time - sched_arr_time) > 0, "gain",
                     ifelse((arr_time - sched_arr_time) < 0, "loss", "no gain/loss")))
```

e) What is the speed of the flight.

```
flights %>%
  mutate(speed = distance / (air_time / 60))  # convert air_time to hours
```

14. Convert dep_time and sched_dep_time to minutes since midnight.

- The `dep_time` and `sched_dep_time` columns are convenient to look at (format HHMM), but they are not continuous numeric values, which makes calculations tricky.
- We can convert them to minutes since midnight using integer division (`%/%`) and modulus (`%%`).

```
flights %>%
  mutate(
    dep_time_minutes = (dep_time %/% 100) * 60 + (dep_time %% 100),
    sched_dep_time_minutes = (sched_dep_time %/% 100) * 60 + (sched_dep_time %% 100))
```

- `dep_time %/% 100` → extracts the hour part (integer division by 100)
- `dep_time %% 100` → extracts the minute part (remainder after division by 100)
- Multiply the hour part by 60 and add the minutes to get total minutes since midnight

Example of `%/%` and `%%`

```
123 %/% 10  # 12 - quotient (tens)
123 %% 10  # 3 - remainder (ones)
```

- `%/%` → integer division (quotient)
- `%%` → modulus (remainder)

15. Compare air_time with arr_time - dep_time. What do you expect to see? What do you see? What do you need to do to fix it?

```
flights %>%
  mutate(actual_air_time = arr_time - dep_time) %>%
  select(actual_air_time, air_time)
```

- dep_time and arr_time are HHMM numbers, not minutes since midnight.
- Subtracting HHMM numbers does not account for hours and minutes correctly.
- Flights that cross midnight or are delayed make the subtraction even more misleading.

How to fix it:

1. Convert dep_time and arr_time to minutes since midnight
2. If arr_time_minutes < dep_time_minutes (flight crosses midnight), add 24*60 to arr_time_minutes before subtraction:

```
flights %>%
  mutate(
    dep_time_minutes = (dep_time %/% 100) * 60 + (dep_time %% 100),
    arr_time_minutes = (arr_time %/% 100) * 60 + (arr_time %% 100),
    actual_air_time = ifelse(arr_time_minutes < dep_time_minutes,
                             arr_time_minutes + 1440 - dep_time_minutes,
                             arr_time_minutes - dep_time_minutes)
  ) %>%
  select(actual_air_time, air_time)
```

Tidying Data: `pivot_longer()` & `pivot_wider()`

The `tidyr` package provides functions to reshape data between long and wide formats.

1. `pivot_longer()`

- Converts data from **wide format** (many columns) into **long format** (fewer columns, more rows).

- Useful when you have multiple columns representing the same type of measurement.

```
pivot_longer(
  data,
  cols,          # columns to pivot
  names_to,      # name of the new key column
  values_to)     # name of the new value column
```

```r
df <- tibble(id = 1:2, math = c(90, 80), english = c(85, 75))
df
```

```
## # A tibble: 2 x 3
##      id  math english
##   <int> <dbl>   <dbl>
## 1     1    90      85
## 2     2    80      75
```

```r
df_long <- df %>%
  pivot_longer(cols = math:english, names_to = "subject", values_to = "score")
df_long
```

```
## # A tibble: 4 x 3
##      id subject score
##   <int> <chr>   <dbl>
## 1     1 math       90
## 2     1 english    85
## 3     2 math       80
## 4     2 english    75
```

2. `pivot_wider()`

- Converts data from long format into wide format.
- Useful when you want separate columns for different values of a variable.

```r
pivot_wider(
  data,
  names_from,   # column whose values become new column names
  values_from)   # column containing values to fill
```

```r
df_wide <- df_long %>%
  pivot_wider(
    names_from = subject,
    values_from = score)
df_wide
```

```
## # A tibble: 2 x 3
##      id  math english
##   <int> <dbl>   <dbl>
## 1     1    90      85
## 2     2    80      75
```