Data Organization Made Easy with dplyr

SCSRU Workshop

Statistical Consulting and Survey Research Unit
University of Waterloo

2026-01-20

After data collection

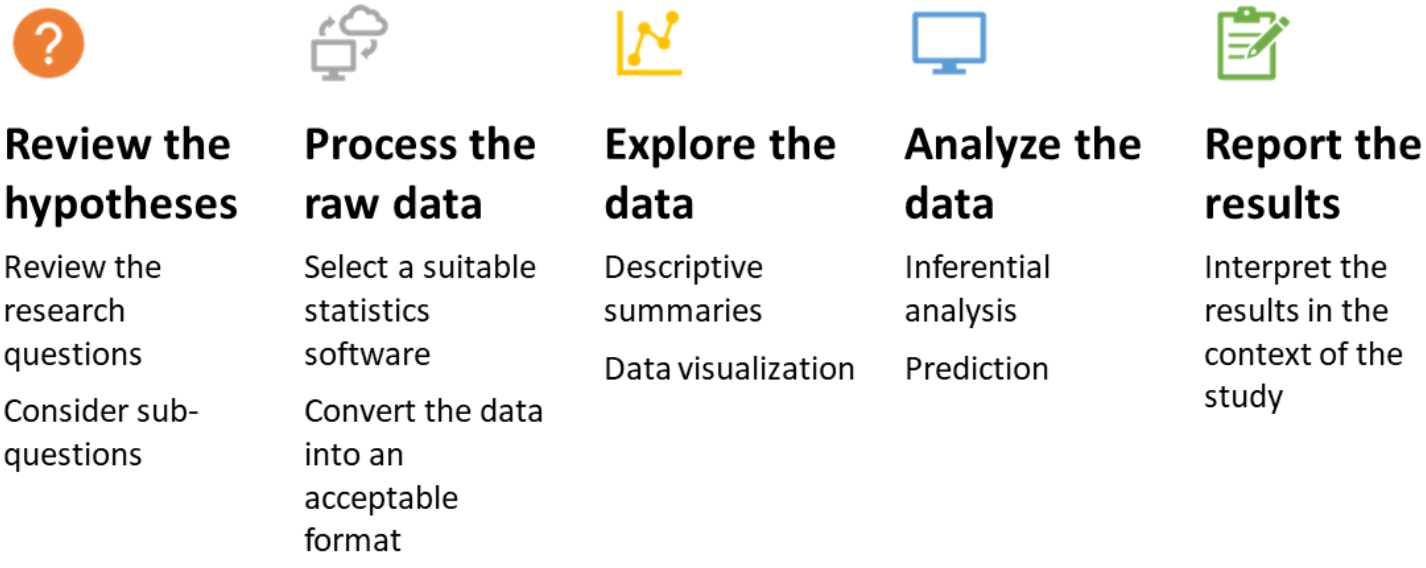

Figure 1: Recommended process

1. The dplyr package

The dplyr is part of the tidyverse packages which is commonly use for data manipulation. It has a lot of functions that are useful for exploring, transforming, and aggregating data.

We will learn more about this library in this workshop. If you have not installed this library, please do so now.

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

We will also load the library into the environment.

The pipeline

Commonly in R, we use operators -> or = to assign values to variables. When we load any `tidyverse` packages, a powerful operator, pipe %>%, is also loaded. It takes what's on its left and *pipes* it to what's on its right. When we have complex code, which means there are multiple functions and we have to nest the parentheses together, the pipe operator will make the code easier to read and understand. For example, suppose we have a vector

```
x <- c(0.239, 0.060, 0.984, 0.788, 0.595)
```

We want to compute the logarithm of `x`, return suitably lagged and iterated differences, compute the exponential function and round the result. Usually we will write

```
round(exp(diff(log(x))), 1)
```

```
## [1]  0.3 16.4  0.8  0.8
```

The pipeline - cont.

With pipe operator %>%, we can rewrite the code as follows:

```
x %>%
  log() %>%
  diff() %>%
  exp() %>%
  round(1)
```

```
## [1]  0.3 16.4  0.8  0.8
```

Does this seem more clear than the previous code?

The pipeline - cont.

- When functions require only one argument, `x %>% f` is equivalent to `f(x)`.
- When multiple arguments are required, the default behavior of %>% is to place `x` as the first argument, i.e., `x %>% f(y)` is equivalent to `f(x,y)`. For example,

```
x <- 10.36849
x %>% round(2)
```

```
## [1] 10.37
```

- If you want to put `x` at another position than the first, you can use dot (`.`) as placeholder, e.g., `x %>% f(y, .)` is equivalent to `f(y, x)`.

1.1 The data set

Throughout this workshop, we will be looking at the data set *Marvel Wikia Data*, which includes the information of Marvel comic characters such as name, eye color, hair color, gender, number of appearances in comic books, etc.

The data set was sent out to you a few days ago. Please set your working directory to where you saved the data set and load the data set into your R environment.

```
# Import data here
df <- read.csv("marvel-wikia-data.csv")
```

Or you can also use the `read_csv()` function from `Tidyverse` package:

```
# install.packages("tidyverse")
library(tidyverse)
df <- read_csv("marvel-wikia-data.csv")
```

Data Background

The *Marvel Wikia Data* comes from Marvel Wikia website, which include the following variables:

Variable	Definition
page_id	The unique identifier for that characters page within the wikia
name	The name of the character
urlslug	The unique url within the wikia that takes you to the character
ID	The identity status of the character (Secret Identity, Public identity)
ALIGN	If the character is Good, Bad or Neutral
EYE	Eye color of the character
HAIR	Hair color of the character
SEX	Sex of the character (e.g. Male, Female, etc.)
GSM	If the character is a gender or sexual minority (e.g. Homosexual characters, bisexual characters)
ALIVE	If the character is alive or deceased
APPEARANCES	The number of appearances of the character in comic books
FIRST APPEARANCE	The month and year of the character's first appearance in a comic book, if available
YEAR	The year of the character's first appearance in a comic book, if available

1.2 The verbs

In this workshop, we will focus on the following common and useful functions:

- `select()`
- `filter()`
- `arrange()`
- `mutate()`
- `summarize()`
- `group_by()`
- `count()`

We will also cover a few advanced topics for those interested:

- Various types of `joins` to merge related datasets
- Window functions, which are similar to `summarize()` but serve different purposes
- Pivoting data between long and wide formats

A glimpse of the data

Before manipulating the data set, it is common to take a look at the data set to get an overall understanding of what we are dealing with. With the `dplyr` library, there is the `glimpse()` function. The function allows us to view the first few values from each variable, along with the data type.

```
glimpse(df)
```

```
## Rows: 16,376
## Columns: 13
## $ page_id             <dbl> 1678, 7139, 64786, 1868, 2460, ~
## $ name                <chr> "Spider-Man (Peter Parker)", "C~
## $ urlslug             <chr> "\\/Spider-Man_(Peter_Parker)",~
## $ ID                  <chr> "Secret Identity", "Public Iden~
## $ ALIGN               <chr> "Good Characters", "Good Charac~
## $ EYE                 <chr> "Hazel Eyes", "Blue Eyes", "Blu~
## $ HAIR                <chr> "Brown Hair", "White Hair", "Bl~
## $ SEX                 <chr> "Male Characters", "Male Charac~
## $ GSM                 <chr> NA, NA, NA, NA, NA, NA, NA, NA,~
## $ ALIVE               <chr> "Living Characters", "Living Ch~
## $ APPEARANCES         <dbl> 4043, 3360, 3061, 2961, 2258, 2~
## $ `FIRST APPEARANCE`  <chr> "Aug-62", "Mar-41", "Oct-74", "~
## $ Year                <dbl> 1962, 1941, 1974, 1963, 1950, 1~
```

2. The `select()` verb

- In many applications, the data sets contain more variables than we need.
- Since we do not need all of the variables, it is recommended to extract the required variable.
- To do so, we can use the `select()` verb.

Example 2.1

Suppose we only need to the name of all the characters

```
df %>% select(name)
```

```
## # A tibble: 16,376 x 1
##    name
##    <chr>
##  1 "Spider-Man (Peter Parker)"
##  2 "Captain America (Steven Rogers)"
##  3 "Wolverine (James \\\"Logan\\\" Howlett)"
##  4 "Iron Man (Anthony \\\"Tony\\\" Stark)"
##  5 "Thor (Thor Odinson)"
##  6 "Benjamin Grimm (Earth-616)"
##  7 "Reed Richards (Earth-616)"
##  8 "Hulk (Robert Bruce Banner)"
##  9 "Scott Summers (Earth-616)"
## 10 "Jonathan Storm (Earth-616)"
## # i 16,366 more rows
```

Helpful operators

When you need to select multiple variables, the following operators will help you.

- `:` for selecting a range of consecutive variables.
- `!` for taking the complement of a set of variables.
- `&` and `|` for selecting the intersection or the union of two sets of variables.
- `c()` for combining selections.

Example 2.2

Suppose we want to select three variables: EYE, HAIR, and SEX. Since they are consecutive in the dataset, we can use ":"

```
df %>% select(EYE:SEX)
```

```
## # A tibble: 16,376 x 3
##    EYE         HAIR        SEX
##    <chr>       <chr>       <chr>
##  1 Hazel Eyes Brown Hair Male Characters
##  2 Blue Eyes  White Hair Male Characters
##  3 Blue Eyes  Black Hair Male Characters
##  4 Blue Eyes  Black Hair Male Characters
##  5 Blue Eyes  Blond Hair Male Characters
##  6 Blue Eyes  No Hair    Male Characters
##  7 Brown Eyes Brown Hair Male Characters
##  8 Brown Eyes Brown Hair Male Characters
##  9 Brown Eyes Brown Hair Male Characters
## 10 Blue Eyes  Blond Hair Male Characters
## # i 16,366 more rows
```

Example 2.3

We can also use `select()` verb to remove variables from the data frame. Since `page_id` is the unique identifier for the characters page within the wikia, we do not need this variable for now. Let's remove it from our data frame.

```
df_trim <- df %>% select(!page_id)
glimpse(df_trim)
```

```
## Rows: 16,376
## Columns: 12
## $ name                <chr> "Spider-Man (Peter Parker)", "C~
## $ urlslug             <chr> "\\/Spider-Man_(Peter_Parker)",~
## $ ID                  <chr> "Secret Identity", "Public Iden~
## $ ALIGN               <chr> "Good Characters", "Good Charac~
## $ EYE                 <chr> "Hazel Eyes", "Blue Eyes", "Blu~
## $ HAIR                <chr> "Brown Hair", "White Hair", "Bl~
## $ SEX                 <chr> "Male Characters", "Male Charac~
## $ GSM                 <chr> NA, NA, NA, NA, NA, NA, NA, NA,~
## $ ALIVE               <chr> "Living Characters", "Living Ch~
## $ APPEARANCES         <dbl> 4043, 3360, 3061, 2961, 2258, 2~
## $ `FIRST APPEARANCE`  <chr> "Aug-62", "Mar-41", "Oct-74", "~
## $ Year                <dbl> 1962, 1941, 1974, 1963, 1950, 1~
```

Practice 2.1

Now we have a new trimmed data frame `df_trim`, but there are still some variables we do not need. Please use `select()` verb to build a new data frame `df_new` from `df_trim` with variables `name`, `ID`, `ALIGN`, `EYE`, `HAIR`, `SEX`, `GSM`, `ALIVE`, `APPEARANCES` and `Year`. You can either remove the rest variables or keep the selected variables.

3. The `filter()` verb

- After we selected the variables we are interested in, we do not always keep all the rows. For example, we may want to remove the rows with missing values.
- Similar to the `select()` verb, it is recommended to extract the required rows.
- To do so, we can use the `filter()` verb.

Example 3.1

Let's find which character has a gold hair.

```
df_new %>% filter(HAIR == "Gold Hair")
```

```
## # A tibble: 8 x 10
##   name              ID    ALIGN EYE   HAIR  SEX   GSM   ALIVE
##   <chr>             <chr> <chr> <chr> <chr> <chr> <chr> <chr>
## 1 Adam Warlock (E~ No D~ Good~ Whit~ Gold~ Male~ <NA>  Livi~
## 2 Blitziana (Eart~ Secr~ Good~ Whit~ Gold~ Fema~ <NA>  Livi~
## 3 Gilpetperdon (E~ Secr~ Bad ~ Gold~ Gold~ Male~ <NA>  Livi~
## 4 Mordecai Midas ~ Secr~ Bad ~ Gold~ Gold~ Male~ <NA>  Livi~
## 5 Aeroika (Earth-~ <NA>  <NA>  <NA>  Gold~ Male~ <NA>  Livi~
## 6 Blaze (Dog) (Ea~ <NA>  Good~ Brow~ Gold~ Male~ <NA>  Livi~
## 7 Jaard (Earth-61~ <NA>  Neut~ Gold~ Gold~ Male~ <NA>  Dece~
## 8 Will Power (Now~ Secr~ Good~ Gold~ Gold~ Male~ <NA>  Livi~
## # i 2 more variables: APPEARANCES <dbl>, Year <dbl>
```

Practice 3.1

Can you find who has “Compound Eyes” in this data frame?

Practice 3.2

I want to know the information about the characters with yellow eyeballs. Could you show the information of the characters with "Yellow Eyeballs" but remove `ID` and `GSM`? Also, we can save this data frame as `df_yelloweye`.

4. The `arrange()` verb

- Sometimes we want to order the rows of a data frame by the values of a particular variable, especially when we have some numerical variables.
- We can use the `arrange()` verb to order the rows in either ascending order or descending order.

Example 4.1

For our `df_yelloweye` data frame, let's order it by the year of the character's first appearance in ascending order.

```
df_yelloweye %>% arrange(by = Year)
```

```
## # A tibble: 6 x 8
##   name       ALIGN EYE   HAIR  SEX   ALIVE APPEARANCES  Year
##   <chr>      <chr> <chr> <chr> <chr> <chr>       <dbl> <dbl>
## 1 Lupo (Ear~ Bad ~ Yell~ No H~ Male~ Livi~          22  1969
## 2 Razor Cut~ <NA>  Yell~ Blac~ Male~ Livi~           4  1987
## 3 Gator (Ea~ Good~ Yell~ No H~ Male~ Dece~          18  1997
## 4 Lupa (Sav~ Bad ~ Yell~ Blac~ Fema~ Livi~           5  2001
## 5 Nurotox (~ Good~ Yell~ No H~ Male~ Livi~           1  2007
## 6 Rana Phil~ Bad ~ Yell~ Gree~ Fema~ Dece~           4  2009
```

Example 4.2

Now let's order it by the number of appearances of the character in descending order. To make it descending, we need to use `desc` function.

```
df_yelloweye %>% arrange(by = desc(APPEARANCES))
```

```
## # A tibble: 6 x 8
##   name       ALIGN EYE   HAIR  SEX   ALIVE APPEARANCES  Year
##   <chr>      <chr> <chr> <chr> <chr> <chr>       <dbl> <dbl>
## 1 Lupo (Ear~ Bad ~ Yell~ No H~ Male~ Livi~          22  1969
## 2 Gator (Ea~ Good~ Yell~ No H~ Male~ Dece~          18  1997
## 3 Lupa (Sav~ Bad ~ Yell~ Blac~ Fema~ Livi~           5  2001
## 4 Razor Cut~ <NA>  Yell~ Blac~ Male~ Livi~           4  1987
## 5 Rana Phil~ Bad ~ Yell~ Gree~ Fema~ Dece~           4  2009
## 6 Nurotox (~ Good~ Yell~ No H~ Male~ Livi~           1  2007
```

Example 4.3

We can also focus on the living characters by combining the `filter()` verb.

```
df_yelloweye %>%
  filter(ALIVE == "Living Characters") %>%
  arrange(by = desc(APPEARANCES))
```

```
## # A tibble: 4 x 8
##   name        ALIGN EYE   HAIR  SEX   ALIVE APPEARANCES  Year
##   <chr>       <chr> <chr> <chr> <chr> <chr>       <dbl> <dbl>
## 1 Lupo (Ear~ Bad ~ Yell~ No H~ Male~ Livi~           22  1969
## 2 Lupa (Sav~ Bad ~ Yell~ Blac~ Fema~ Livi~            5  2001
## 3 Razor Cut~ <NA>  Yell~ Blac~ Male~ Livi~            4  1987
## 4 Nurotox (~ Good~ Yell~ No H~ Male~ Livi~            1  2007
```

Practice 4.1

Try to show the `name`, `EYE`, `SEX`, `ALIVE` of the characters in `df_yelloweye` who have "No Hair" and order them by `Year` in ascending order.

5. The `mutate()` verb

- Sometimes we want to modify existing variables or create new variables, such as changing the type of variables or creating a new variable to save the total of two variables.
- We can use the `mutate()` verb to modify or create variables.

Example 5.1

In `df_yelloweye`, I want to use number 0 for `Female Characters` and 1 for `Male Characters` for the SEX variable to make it simpler.

```
df_yelloweye %>%
  mutate(SEX = ifelse(SEX=="Female Characters", 0, 1))
```

```
## # A tibble: 6 x 8
##   name        ALIGN EYE   HAIR    SEX ALIVE APPEARANCES  Year
##   <chr>       <chr> <chr> <chr> <dbl> <chr>       <dbl> <dbl>
## 1 Lupo (Ear~ Bad ~ Yell~ No H~     1 Livi~          22  1969
## 2 Gator (Ea~ Good~ Yell~ No H~     1 Dece~          18  1997
## 3 Lupa (Sav~ Bad ~ Yell~ Blac~     0 Livi~           5  2001
## 4 Razor Cut~ <NA>  Yell~ Blac~     1 Livi~           4  1987
## 5 Rana Phil~ Bad ~ Yell~ Gree~     0 Dece~           4  2009
## 6 Nurotox (~ Good~ Yell~ No H~     1 Livi~           1  2007
```

Example 5.2

I want to see the ratio `APPEARANCES / Year` and save it in a new variable named `Ratio`.

```
df_yelloweye %>%
  mutate(Ratio = round(APPEARANCES / Year, 3))
```

```
## # A tibble: 6 x 9
##   name                                ALIGN        EYE   HAIR  SEX   ALIVE APPEARANCES  Year Ratio
##   <chr>                               <chr>        <chr> <chr> <chr> <chr>       <dbl> <dbl> <dbl>
## 1 Lupo (Earth-616)                    Bad Charact~ Yell~ No H~ Male~ Livi~          22  1969 0.011
## 2 Gator (Earth-616)                   Good Charac~ Yell~ No H~ Male~ Dece~          18  1997 0.009
## 3 Lupa (Savage Land Mutate) (Earth-616) Bad Charact~ Yell~ Blac~ Fema~ Livi~          5  2001 0.002
## 4 Razor Cut (Earth-616)               <NA>         Yell~ Blac~ Male~ Livi~          4  1987 0.002
## 5 Rana Philips (Earth-616)            Bad Charact~ Yell~ Gree~ Fema~ Dece~          4  2009 0.002
## 6 Nurotox (Earth-616)                 Good Charac~ Yell~ No H~ Male~ Livi~          1  2007 0
```

Practice 5.1

Can you change `Living Characters` to 1 and `Deceased Characters` to 0 for the `ALIVE` variable in `df_yelloweye`?

6. The `summarize()` verb

- We usually need to calculate some summary statistics of the data when we conduct a data analysis. For example, we may want mean or median of a variable.
- We can use the `summarize()` verb in these cases.

The difference between `mutate()` and `summarize()`

- Remember `mutate()`? Seems like we can do the same thing with `mutate()`, right?
- However, `mutate()` returns the same number of rows in a data frame, while `summarize()` returns just one row or one row for each group (if you summarize with groups which we will mention later).

Example 6.1

Let's find the minimum and maximum `Year` in `df_yelloweye`.

```
df_yelloweye %>%
  summarize(min_year = min(Year),
            max_year = max(Year))
```

```
## # A tibble: 1 x 2
##   min_year max_year
##      <dbl>    <dbl>
## 1     1969     2009
```

Practice 6.1

What is the mean of `APPEARANCES`?

Revisiting `mutate()`

Recall Example 5.1, we used `mutate()` to change `SEX` variable from character to numeric.

```
df_yelloweye %>%
  mutate(SEX = ifelse(SEX=="Female Characters", 0, 1))
```

```
## # A tibble: 6 x 8
##   name        ALIGN EYE   HAIR    SEX ALIVE APPEARANCES  Year
##   <chr>       <chr> <chr> <chr> <dbl> <chr>        <dbl> <dbl>
## 1 Lupo (Ear~ Bad ~ Yell~ No H~     1 Livi~           22  1969
## 2 Gator (Ea~ Good~ Yell~ No H~     1 Dece~           18  1997
## 3 Lupa (Sav~ Bad ~ Yell~ Blac~     0 Livi~            5  2001
## 4 Razor Cut~ <NA>  Yell~ Blac~     1 Livi~            4  1987
## 5 Rana Phil~ Bad ~ Yell~ Gree~     0 Dece~            4  2009
## 6 Nurotox (~ Good~ Yell~ No H~     1 Livi~            1  2007
```

But this overwrites the original `SEX` variable, and we can no longer access its original values of `Female Characters` and `Male Characters`.

Revisiting `mutate()`

It is good practice to keep the original `SEX` variable and create a new numeric variable `SEX_num`. In the example below, we create a new variable `SEX_num` and use it to compute the proportion of male characters by `ALIVE` status in `df_yelloweye`:

```
df_yelloweye %>%
  mutate(SEX_num = ifelse(SEX=="Female Characters", 0, 1)) %>%
  select(SEX_num, ALIVE) %>%
  group_by(ALIVE) %>%
  summarize(prop_male = mean(SEX_num))
```

```
## # A tibble: 2 x 2
##   ALIVE                prop_male
##   <chr>                    <dbl>
## 1 Deceased Characters      0.5
## 2 Living Characters        0.75
```

Revisiting `mutate()`

A more general form of this is using `case_when()` function inside `mutate()`. For example, we can recode `SEX` variable as follows:

```
df_yelloweye %>%
  mutate(
    SEX_num = case_when(
      SEX == "Female Characters" ~ 0,
      SEX == "Male Characters" ~ 1,
      TRUE ~ 0 # Default value is 0
    )
  ) %>%
  select(name, ALIGN, EYE, HAIR, SEX, SEX_num)
```

```
## # A tibble: 6 x 6
##   name                        ALIGN EYE   HAIR  SEX   SEX_num
##   <chr>                       <chr> <chr> <chr> <chr>   <dbl>
## 1 Lupo (Earth-616)            Bad ~ Yell~ No H~ Male~       1
## 2 Gator (Earth-616)           Good~ Yell~ No H~ Male~       1
## 3 Lupa (Savage Land Mutate)~ Bad ~ Yell~ Blac~ Fema~       0
## 4 Razor Cut (Earth-616)       <NA>  Yell~ Blac~ Male~       1
## 5 Rana Philips (Earth-616)    Bad ~ Yell~ Gree~ Fema~       0
## 6 Nurotox (Earth-616)         Good~ Yell~ No H~ Male~       1
```

It is much easier to use `case_when()` to handle multiple conditions than using nested `ifelse()` statements.

7. The `group_by()` verb

- A common paring with `summarize()` is the `group_by()` verb. It splits the data frame into a number of smaller groups.
- We use `group_by()` to tell `summarize()` which subgroups to apply the calculations on.

Example 7.1

Let's find the minimum and maximum `Year` for each category of `HAIR`.

```
df_yelloweye %>%
  group_by(HAIR) %>%
  summarize(min_year = min(Year),
            max_year = max(Year))
```

```
## # A tibble: 3 x 3
##   HAIR       min_year max_year
##   <chr>         <dbl>    <dbl>
## 1 Black Hair     1987     2001
## 2 Green Hair     2009     2009
## 3 No Hair        1969     2007
```

Practice 7.2

Can you find the mean `APPEARANCES` for each gender?

Example 7.2

Furthermore, `group_by()` accepts multiple grouping variables. Say we want to find the total `APPEARANCES` across `SEX` and `HAIR`.

```
df_yelloweye_summarized <- df_yelloweye %>%
  group_by(SEX, HAIR) %>%
  summarize(appearance_per_group = sum(APPEARANCES))
df_yelloweye_summarized
```

```
## # A tibble: 4 x 3
## # Groups:   SEX [2]
##   SEX               HAIR       appearance_per_group
##   <chr>             <chr>                     <dbl>
## 1 Female Characters Black Hair                    5
## 2 Female Characters Green Hair                    4
## 3 Male Characters   Black Hair                    4
## 4 Male Characters   No Hair                      41
```

Example 7.3

Now in `df_yelloweye.summarized`, I want a new column to show the total `APPEARANCES` by `Sex`.

```
df_yelloweye_summarized %>%
  group_by(SEX) %>%
  mutate(appearance_per_sex = sum(appearance_per_group))
```

```
## # A tibble: 4 x 4
## # Groups:   SEX [2]
##   SEX               HAIR       appearance_per_group appearance_per_sex
##   <chr>             <chr>                     <dbl>              <dbl>
## 1 Female Characters Black Hair                    5                  9
## 2 Female Characters Green Hair                    4                  9
## 3 Male Characters   Black Hair                    4                 45
## 4 Male Characters   No Hair                      41                 45
```

8. The `count()` verb

- When we want to count the unique values of one or more variables, we can use the `count()` verb.
- It is roughly equivalent to a combination of `group_by()` and `summarize(n=n())`.

Example 8.1

We want to know the number of characters for each gender with different hair colors:

```
df_yelloweye %>%
  count(SEX, HAIR)
```

```
## # A tibble: 4 x 3
##   SEX                 HAIR           n
##   <chr>               <chr>      <int>
## 1 Female Characters Black Hair      1
## 2 Female Characters Green Hair      1
## 3 Male Characters   Black Hair      1
## 4 Male Characters   No Hair         3
```

Example 8.2

If you want to use `group_by()` and `summarize()`, you can get the same results by

```
df_yelloweye %>%
  group_by(SEX, HAIR) %>%
  summarize(n = n())
```

```
## # A tibble: 4 x 3
## # Groups:   SEX [2]
##   SEX                HAIR           n
##   <chr>              <chr>      <int>
## 1 Female Characters Black Hair     1
## 2 Female Characters Green Hair     1
## 3 Male Characters   Black Hair     1
## 4 Male Characters   No Hair        3
```

9.1 Joins

- `Joins` combine two tables based on one or more common key columns.
- They allow you to bring together related pieces of information stored in separate tables.
- Common use cases:
  - Adding extra attributes to a dataset
  - Merging tables to create a single, unified view for comprehensive analysis
  - Identifying unmatched or missing records

9.2 Types of Joins

- `inner_join(x, y, by = "key_column")`: Returns all rows from `x` where there are matching values in `y`, and all columns from both `x` and `y`. Non-matching rows are dropped. `"key_column"` specifies the common column(s) to join on.
  - Useful when you only need matching records between two tables.
- `left_join(x, y, by = "key_column")`: Returns all rows from `x` and all columns from both `x` and `y`. If there's a match in `y` on `"key_column"`, the corresponding data is included; otherwise, NA values are added for the unmatched rows from `y`.
  - Ideal when you have a primary dataset and want to enrich it with extra information.
- `full_join(x, y, by = "key_column")`: Returns all rows and all columns from both `x` and `y`. If a row doesn't have a match in the other data frame, NA values are added for the missing columns.

Example 9.1

Suppose we have another dataset `ratings` that contains the average ratings of the Marvel contents for each year between 2001 and 2010.

```
ratings = data.frame(Year = 2001:2010,
                     avg_rating = c(7.8, 8.0, 8.2, 8.5, 8.7, 8.9, 9.0, 9.2, 9.3, 9.5))
```

We can inner join `df_yelloweye` and `ratings` based on the `Year` column, to get the average ratings of Marvel contents for each character's first appearance year between 2001 and 2010.

```
df_yelloweye %>%
  inner_join(ratings, by = c("Year"))
```

```
## # A tibble: 3 x 9
##   name                          ALIGN EYE   HAIR  SEX   ALIVE APPEARANCES  Year avg_rating
##   <chr>                         <chr> <chr> <chr> <chr> <chr>       <dbl> <dbl>      <dbl>
## 1 Lupa (Savage Land Mutate) (E~ Bad ~ Yell~ Blac~ Fema~ Livi~           5  2001        7.8
## 2 Rana Philips (Earth-616)      Bad ~ Yell~ Gree~ Fema~ Dece~           4  2009        9.3
## 3 Nurotox (Earth-616)           Good~ Yell~ No H~ Male~ Livi~           1  2007        9
```

Practice 9.1

Suppose that the average rating of Marvel contents outside of 2001-2010 is 8.0. Use `left_join()` to assign a rating to every character in `df_yelloweye`.

10.1 Window Functions

- `Window functions` perform calculations within groups without collapsing the data.
- Unlike `summarize()`, they preserve the original number of rows.
- This makes them ideal for comparing individual records to their peers.

10.2 Types of Window Functions

- `row_number()`: Assigns a unique sequential integer to rows within a group, ordered by specified columns.
- `min_rank()`: Assigns ranks to rows within a group, with ties receiving the same rank and gaps in ranking for subsequent values.
- `lead()` and `lag()`: Accesses values from subsequent or previous rows within a group, useful for calculating differences over time.
- `cumsum()`: Computes the cumulative sum of a numeric column within a group. We also have `cummean()`, `cummin()`, and `cummax()` for cumulative mean, minimum, and maximum respectively.

Example 10.1

Let's rank characters in `df_yelloweye` by their `APPEARANCES` within each `HAIR` color category.

```
df_yelloweye %>%
  group_by(HAIR) %>%
  mutate(appearance_rank = row_number(desc(APPEARANCES))) %>%
  arrange(HAIR, appearance_rank) %>%
  select(name, HAIR, APPEARANCES, appearance_rank)
```

```
## # A tibble: 6 x 4
## # Groups:   HAIR [3]
##   name                                  HAIR       APPEARANCES appearance_rank
##   <chr>                                 <chr>            <dbl>            <int>
## 1 Lupa (Savage Land Mutate) (Earth-616) Black Hair           5                1
## 2 Razor Cut (Earth-616)                 Black Hair           4                2
## 3 Rana Philips (Earth-616)              Green Hair           4                1
## 4 Lupo (Earth-616)                      No Hair             22                1
## 5 Gator (Earth-616)                     No Hair             18                2
## 6 Nurotox (Earth-616)                   No Hair              1                3
```

Practice 10.1

Suppose we want to compute the cumulative appearances of characters over the years within each `SEX` category in `df_yelloweye`. Can you do this using `cumsum()`?

11.1 Pivoting

- Pivoting functions help reshape data between wide and long formats.
- Long format data: Each row is a single observation for a specific variable, with values repeated for the same individual/entity.
- Wide format data: Each row is a single observation, with variables in separate columns.

Long Format

Name	Subject	Score
Alice	Math	90
Alice	Science	85
Bob	Math	78
Bob	Science	82
Carol	Math	85
Carol	Science	89

Wide Format

Name	Math	Science
Alice	90	85
Bob	78	82
Carol	85	89

11.2 When to Use Each Format

- Long format
  - Good for database storage, processing (`dplyr`) and visualization (`ggplot2`)
  - Analysis functions with repeated measures (`lmer`)
- Wide format
  - Easy for human interpretation and reporting data summaries
  - Analysis functions that assume one feature per column (`lm`, `cor`)

11.3 Pivot Functions

- `pivot_longer(df, cols, names_to, values_to)`: Given a wide-format `df`, this function reshapes specified `cols` into a longer format. The names of the original columns are stored as values in a new column named `names_to`, and their corresponding values are stored in another new column named `values_to`.
- `pivot_wider(df, names_from, values_from)`: Given a long-format `df`, this function reshapes it into a wider format. The unique values from the `names_from` column become new column names, and their corresponding values from the `values_from` column populate these new columns.

Example 11.1

Let's pivot long-format data to wide-format.

```
options(width = 90)
long_df = df_trim %>%
  select(ALIGN, SEX) %>%
  filter((SEX == "Female Characters" | SEX == "Male Characters") & !is.na(ALIGN)) %>%
  group_by(ALIGN, SEX) %>%
  summarize(Count = n())
long_df
```

```
## # A tibble: 6 x 3
## # Groups:   ALIGN [3]
##   ALIGN             SEX               Count
##   <chr>             <chr>             <int>
## 1 Bad Characters    Female Characters   976
## 2 Bad Characters    Male Characters    5338
## 3 Good Characters   Female Characters  1537
## 4 Good Characters   Male Characters    2966
## 5 Neutral Characters Female Characters  640
## 6 Neutral Characters Male Characters   1440
```

```
long_df %>%
  pivot_wider(names_from = SEX, values_from = Count)
```

```
## # A tibble: 3 x 3
## # Groups:   ALIGN [3]
##   ALIGN             `Female Characters` `Male Characters`
##   <chr>                           <int>             <int>
## 1 Bad Characters                    976              5338
## 2 Good Characters                  1537              2966
## 3 Neutral Characters                640              1440
```

12. Final Practice

Now, it's time to review everything we learnt from this workshop! Here are some final tasks:

- Let's see if we can find some famous characters! Build a dataset from the original dataset `df` with the characters who has more than 1000 appearances in the comic books and whose first appearance is before the year of 1976.
- Of course, we only need important variables in our dataset, including `name`, `ALIGN`, `EYE`, `HAIR`, `SEX`, `ALIVE`, `APPEARANCES` and `Year`, and order them by `Year` in ascending order.
- Build a new variable `Age`, which shows how many years the character has been created. Hint: You can use this year (2024) and the value of `Year` to calculate it.
- Show the average `Age` for each gender.
- Find the number of characters in each `EYE` category.

Beyond this workshop

There are many more functions in the `dplyr` package that we did not cover in this workshop. You can check the official documentation for more information. There are many `dplyr` cheat sheets available online. A good example can be found here.

For those who are interested to learn more, the SCSRU hosts statistics seminars and workshops focusing on topics commonly encountered by researchers on campus. Please check our website for future events.

Thank you!

The Statistical Consulting and Survey Research Unit (SCSRU) is the unit through which the Department of Statistics and Actuarial Science provides statistical advice to those working on research problems.