DSA2101

Essential Data Analytics Tools: Data Visualization

Yuting Huang

Week 5 Introduction to `tidyverse`

What is data manipulation/data wrangling?

“Data janitor work”

It is extremely rare that the data you obtain will be in precisely the right format for the analysis that you wish to do. Very often, we need to do some or all of the following:

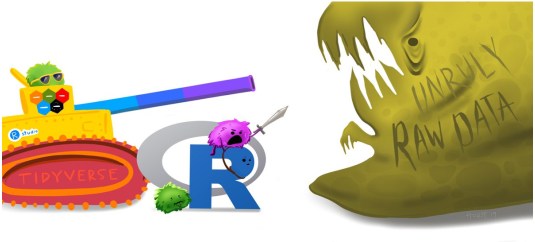
- Select only a subset of rows and/or columns
- Create new variables or summaries
- Rename the variables
- Re-order the data
- Re-shape the data
- …

What is data manipulation/data wrangling?

`dplyr` is a grammar of data manipulation, providing a set of functions that help us solve the most common data manipulation challenges.

1. **Data transformation** Week 5
   - `filter()`, `select()`, `mutate()`, `arrange()`, `summarize()`
   - `group_by()` and `%>%`
2. **Tidy data** Week 6
   - `gather()`, `spread()`, `separate()`, `unite()`
3. **Relational data** Week 7

Pre-requisites



The easiest way to get `dplyr` is to install the `tidyverse` package (https://www.tidyverse.org/packages/).

- We will use the `starwars` data set from the package.

```r
# install.packages("tidyverse")
library(tidyverse)
# Load data set
data(starwars)
```

Artwork by Allison Horst.

Starwars data set

- `glimpse()` is similar to `str()` from base `R`. It allows us to see as much data as possible.

```
glimpse(starwars)
```

```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180,
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 7
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown",
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light",
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femi
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan",
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Hu
## $ films      <list> <"A New Hope", "The Empire Strikes Back", "Return of the
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "I
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1
```

Starwars data set

```
head(starwars)
```

```
## # A tibble: 6 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   ge
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <c
## 1 Luke Sky~    172    77 blond      fair       blue            19   male  ma
## 2 C-3PO        167    75 <NA>       gold       yellow         112   none  ma
## 3 R2-D2         96    32 <NA>       white, bl~ red             33   none  ma
## 4 Darth Va~    202   136 none       white      yellow          41.9 male  ma
## 5 Leia Org~    150    49 brown      light      brown           19   fema~ fe
## 6 Owen Lars    178   120 brown, gr~ light      blue            52   male  ma
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
class(starwars)
```

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

Tibbles

The output shows that it is in fact not a data frame, it is a **tibble**.

- Base `R` functions import data as data frames.
- `tidyverse` functions import data as tibbles.
    - A modern version of data frame, typically useful for large data sets.
    - Easy to view the numbers of rows, columns, and variable types.

```r
data.frame(name = c("Sarah", "Ana", "Jone"),
           age = c(19, 21, 28),
           city = c(NA, "Singapore", "New York"))
```

```
##    name age      city
## 1 Sarah  19      <NA>
## 2   Ana  21 Singapore
## 3  Jone  28  New York
```

```r
tibble(name = c("Sarah", "Ana", "Jone"),
       age = c(19, 21, 28),
       city = c(NA, "Singapore", "New York"))
```

```
## # A tibble: 3 x 3
##   name    age city
##   <chr> <dbl> <chr>
## 1 Sarah    19 <NA>
## 2 Ana      21 Singapore
## 3 Jone     28 New York
```

Key functions

The following five functions, and the combinations of them, will allow you to accomplish the vast majority of data cleaning tasks.

- `filter()`: select observations (rows) by the value in their columns.
- `select()`: select variables (columns) by their names.
- `mutate()`: create new variables.
- `arrange()`: reorder the rows by ascending or descending order.
- `summarize()` or `summarise()`: collapse many values down to a single value.

In conjunction with `group_by()`, which splits a data set by values in a variable, these functions help us deal with common data manipulation challenges.

Applying these functions

Each of these functions is called in an identical manner.

- The first argument is the data frame.
- The subsequent arguments describe what to do with the data frame, using variable names *without* quotes.
- The output is a new data frame; the original data frame is not modified.

These operations can be chained using the **pipe operator `%>%`**.

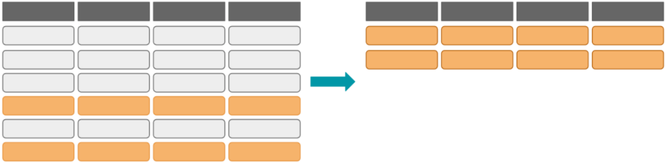
Let’s get started!

Artwork by Allison Horst.

The `filter()` function

The `filter()` function subsets the **rows** in a data frame by testing against a conditional statement.

The output will be a data set with fewer rows than the original data.



The filter() function

```
filter(starwars, sex == "female")
```

```
## # A tibble: 16 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   ge
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <c
##  1 Leia Or~    150  49   brown      light      brown             19 fema~ fe
##  2 Beru Wh~    165  75   brown      light      blue              47 fema~ fe
##  3 Mon Mot~    150  NA   auburn     fair       blue              48 fema~ fe
##  4 Padmé A~    185  45   brown      light      brown             46 fema~ fe
##  5 Shmi Sk~    163  NA   black      fair       brown             72 fema~ fe
##  6 Ayla Se~    178  55   none       blue       hazel             48 fema~ fe
##  7 Adi Gal~    184  50   none       dark       blue              NA fema~ fe
##  8 Luminar~    170  56.2 black      yellow     blue              58 fema~ fe
##  9 Barriss~    166  50   black      yellow     blue              40 fema~ fe
## 10 Dormé       165  NA   brown      light      brown             NA fema~ fe
## 11 Zam Wes~    168  55   blonde     fair, gre~ yellow            NA fema~ fe
## 12 Taun We     213  NA   none       grey       black             NA fema~ fe
## 13 Jocasta~    167  NA   white      fair       blue              NA fema~ fe
## 14 Shaak Ti    178  57   none       red, blue~ black             NA fema~ fe
## 15 Rey          NA  NA   brown      light      hazel             NA fema~ fe
## 16 Captain~     NA  NA   none       none       unknown           NA fema~ fe
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

The `filter()` function

```r
filter(starwars, sex == "female")
```

`filter()` allows us to keep rows based on the value of the entries.

- The first argument is the name of the data frame.
- The second and subsequent arguments are conditions that must be true to keep the row.
- When we run `filter()`, the function executes the filtering operation and print the result.
- It does not modify the original data frame. We need to assign the output to a new object in order to save it.

Logical operators

- We can combine conditions with & or , to indicate **and** (check for both conditions).
- … with | to indicate **or** (check for either condition).
- To filter all light-skin female or light-skin male,

```
filter(starwars,
       skin_color == "light" & (sex == "female" | sex == "male"))
```

```
## # A tibble: 10 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   ge
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <c
##  1 Leia Or~    150    49 brown      light      brown             19 fema~ fe
##  2 Owen La~    178   120 brown, gr~ light      blue              52 male  ma
##  3 Beru Wh~    165    75 brown      light      blue              47 fema~ fe
##  4 Biggs D~    183    84 black      light      brown             24 male  ma
##  5 Lobot       175    79 none       light      blue              37 male  ma
##  6 Padmé A~    185    45 brown      light      brown             46 fema~ fe
##  7 Dormé       165    NA brown      light      brown             NA fema~ fe
##  8 Raymus ~    188    79 brown      light      brown             NA male  ma
##  9 Rey          NA    NA brown      light      hazel             NA fema~ fe
## 10 Poe Dam~     NA    NA brown      light      brown             NA male  ma
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Logical operators

The `%in%` operator matches conditions provided in a vector constructed with `c()`.

- A useful shortcut when we are combining `|` and `==`.

```r
filter(starwars,
       skin_color == "light", sex %in% c("female", "male"))
```

- The code reads:

  Give me the observations in ***`starwars`*** *with "light" skin color, whose sex is either "female" or "male".*

Compared to base R functions

- Base R uses the **bracket method** to select rows that satisfy certain conditions.

```r
# Base R
starwars[which(starwars$skin_color == "light" &
                 (starwars$sex == "female" | starwars$sex == "male")), ]
```

- You can see the advantage of the dplyr syntax:

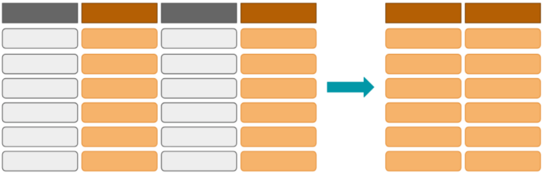
```r
# dplyr method 1
filter(starwars,
       skin_color == "light" & (sex == "female" | sex == "male"))

# dplyr method 2
filter(starwars,
       skin_color == "light", sex %in% c("female", "male"))
```

The `select()` function

The `select()` function returns a subset of **columns**.

The output will be a data set with fewer columns than the original data.



The select() function

It is not uncommon to get data sets with hundreds (or even thousands) or variables.

- When we want to zoom in on a particular set of variables, we can use the `select()` command.
- To select columns by name:

```r
select(starwars, hair_color, birth_year)
```

- Compared to the base `R` bracket method

```r
starwars[ , c("hair_color", "birth_year")]
```

The `select()` function

Compared to the base `R` method, the `dplyr` verbs are much more flexible.

- To select columns located between `hair_color` and `eye_color`.

```
select(starwars, hair_color:eye_color)
```

- To select columns *except* those located between `hair_color` and `eye_color`.

```
select(starwars, -(hair_color:eye_color))
```

Helper functions

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")` matches column names that begin with “abc”.
- `ends_with("xyz")` matches column names that end with “xyz”.
- `contains("ijk")` matches column names that contain “ijk”.
- `num_range("x", 1:3)` matches columns `x1`, `x2`, and `x3`.

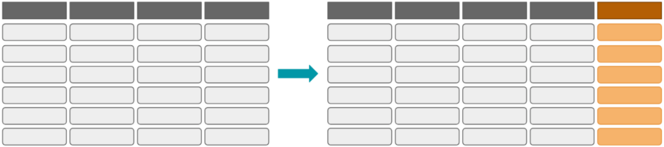
For example, to select all columns that end with **color**.

```
select(starwars, ends_with("color"))
```

The `mutate()` function

The `mutate()` function **adds new columns** of data, thus "mutating" the dimensions of the original data set.

The output will be a data frame with more columns than the original data.



- By default, `mutate()` adds the new columns to the **right hand side** of the data set.

The `mutate()` function

- Let us first create a new data frame, `df`, with fewer columns so we can see the manipulation results more easily.

```
df1 <- select(starwars, name, height, mass, species)
head(df1)
```

```
## # A tibble: 6 x 4
##   name           height  mass species
##   <chr>           <int> <dbl> <chr>
## 1 Luke Skywalker    172    77 Human
## 2 C-3PO             167    75 Droid
## 3 R2-D2              96    32 Droid
## 4 Darth Vader       202   136 Human
## 5 Leia Organa       150    49 Human
## 6 Owen Lars         178   120 Human
```

The mutate() function

- The following code creates two new columns.

```r
df2 <- mutate(df1,
              height_m = height/100,
              BMI = mass/(height_m^2))
head(df2, 3)
```

```
## # A tibble: 3 x 6
##   name           height  mass species height_m   BMI
##   <chr>           <int> <dbl> <chr>      <dbl> <dbl>
## 1 Luke Skywalker    172    77 Human       1.72  26.0
## 2 C-3PO             167    75 Droid       1.67  26.9
## 3 R2-D2              96    32 Droid       0.96  34.7
```

- What is the corresponding command in base R?

The mutate() function

- By default, `mutate()` adds new columns to the right of the data set.
- We can use the `.before` argument to add the variables to the left of `name`.

```r
df2 <- mutate(df1,
              height_m = height/100,
              BMI = mass/(height_m^2),
              .before = name)
head(df2, 3)
```

```
## # A tibble: 3 x 6
##   height_m   BMI name           height  mass species
##      <dbl> <dbl> <chr>           <int> <dbl> <chr>
## 1     1.72  26.0 Luke Skywalker    172    77 Human
## 2     1.67  26.9 C-3PO             167    75 Droid
## 3     0.96  34.7 R2-D2              96    32 Droid
```

- We can also add the new variables after `name` by `.after = name`.

Variant functions to `mutate()`

There are a number of variant functions to `mutate()`:

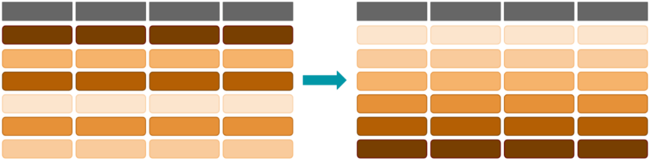
- `mutate_if()` first requires a function that returns a boolean to select columns. If the condition is met, the mutate function will be applied on those variables.
- `mutate_at()` requires selection of a set of columns via the `vars()` argument. The mutate function will be applied on the selected columns.
- `mutate_all()` applies the mutate function across all columns.

For example, to convert all variables of type character to lowercase:

```r
mutate_if(df1, is.character, tolower)
```

The `arrange()` function

The `arrange()` function **changes the order of observations** in a data frame.



- It takes a data frame and a set of column names to order by.
- If you provide more than one column name, each additional column will be used to break ties in the values of preceding ones.

The `arrange()` function

- By default, the function arranges observations in ascending order of the provided variable.

```r
arrange(df1, mass)
```

```
## # A tibble: 87 x 4
##    name                  height  mass species
##    <chr>                  <int> <dbl> <chr>
##  1 Ratts Tyerel              79    15 Aleena
##  2 Yoda                      66    17 Yoda's species
##  3 Wicket Systri Warrick     88    20 Ewok
##  4 R2-D2                     96    32 Droid
##  5 R5-D4                     97    32 Droid
##  6 Sebulba                  112    40 Dug
##  7 Padmé Amidala            185    45 Human
##  8 Dud Bolt                  94    45 Vulptereen
##  9 Wat Tambor               193    48 Skakoan
## 10 Sly Moore                178    48 <NA>
## # i 77 more rows
```

The arrange() function

- To arrange a column in descending order, use the `desc()` operator inside of `arrange()`.

```r
arrange(df1, desc(mass))
```

```
## # A tibble: 87 x 4
##    name                  height  mass species
##    <chr>                  <int> <dbl> <chr>
##  1 Jabba Desilijic Tiure    175  1358 Hutt
##  2 Grievous                 216   159 Kaleesh
##  3 IG-88                    200   140 Droid
##  4 Darth Vader              202   136 Human
##  5 Tarfful                  234   136 Wookiee
##  6 Owen Lars                178   120 Human
##  7 Bossk                    190   113 Trandoshan
##  8 Chewbacca                228   112 Wookiee
##  9 Jek Tono Porkins         180   110 <NA>
## 10 Dexter Jettster          198   102 Besalisk
## # i 77 more rows
```

Compared to base R functions

- To arrange the data set in ascending order of `mass`,

```
df1[order(df1$mass), ]
```

- To do so in descending order of `mass`,

```
df1[order(-df1$mass), ]
```

The `arrange()` function

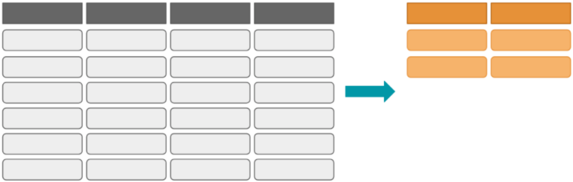
- To arrange the data first by mass (in ascending order), then by height (in descending order):

```r
arrange(df1, mass, desc(height))
```

```
## # A tibble: 87 x 4
##    name                  height  mass species
##    <chr>                  <int> <dbl> <chr>
##  1 Ratts Tyerel              79    15 Aleena
##  2 Yoda                      66    17 Yoda's species
##  3 Wicket Systri Warrick     88    20 Ewok
##  4 R5-D4                     97    32 Droid
##  5 R2-D2                     96    32 Droid
##  6 Sebulba                  112    40 Dug
##  7 Padmé Amidala            185    45 Human
##  8 Dud Bolt                  94    45 Vulptereen
##  9 Wat Tambor               193    48 Skakoan
## 10 Sly Moore                178    48 <NA>
## # i 77 more rows
```

The `summarize()` function

The `summarize()`, or `summarise()`, function creates individual summary statistics from large data sets.



The `summarize()` function

- To compute the average height for Star Wars characters:

```r
summarize(starwars, height = mean(height, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   height
##    <dbl>
## 1   175.
```

- `na.rm = TRUE` removes `NA` values from the calculation.
- The output data set collapses to a 1×1 tibble, containing the mean heights of all Star Wars characters.
- This function is not useful on its own. However, when paired with `group_by()`, we can change the unit of analysis from the entire data set to individual groups.

The `group_by()` function

The `group_by()` operator changes the unit of analysis from the complete data set to individual **groups**.

- It has no effect on the `select()` function.
- The `filter()` and `mutate()` functions work within the group.
- The `arrange()` function ignores groupings by default. We can turn it on by `.by_group = TRUE`.
- When paired with `summarize()`, we can compute summary statistics for individual groups.

Example

- Let us first use a simple data frame to understand the concepts.

```
df3 <- tibble(Name = c("a", "b", "c", "c", "b"),
              x = c(1, 9, 4, 15, NA))
df3
```

```
## # A tibble: 5 x 2
##   Name      x
##   <chr> <dbl>
## 1 a         1
## 2 b         9
## 3 c         4
## 4 c        15
## 5 b        NA
```

Create a group

- Create a group using the character values in `Name`.

```
df4 <- group_by(df3, Name)
df4
```

```
## # A tibble: 5 x 2
## # Groups:   Name [3]
##   Name      x
##   <chr> <dbl>
## 1 a         1
## 2 b         9
## 3 c         4
## 4 c        15
## 5 b        NA
```

summarize() by group

- Compute group mean and name the new variable as `x_mean`.

```
summarize(df4, x_mean = mean(x, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   Name  x_mean
##   <chr>  <dbl>
## 1 a        1
## 2 b        9
## 3 c        9.5
```

`filter()` by group

- Within each group, keep rows with value larger than or equal to the group mean.

```r
filter(df4, x >= mean(x))
```

```
## # A tibble: 2 x 2
## # Groups:   Name [2]
##   Name      x
##   <chr> <dbl>
## 1 a         1
## 2 c        15
```

- Notice that group `b` is excluded from the result.

mutate() by group

- Add a column that calculates the **cumulative sum** within each group.

```
# Replace NAs with 0
mutate(df4, sum_x = cumsum(replace_na(x, 0)))
```

```
## # A tibble: 5 x 3
## # Groups:   Name [3]
##   Name      x sum_x
##   <chr> <dbl> <dbl>
## 1 a         1     1
## 2 b         9     9
## 3 c         4     4
## 4 c        15    19
## 5 b        NA     9
```

`arrange()` by group

- Sort data within each group.

```r
arrange(df4, x, .by_group = TRUE)
```

```
## # A tibble: 5 x 2
## # Groups:   Name [3]
##   Name      x
##   <chr> <dbl>
## 1 a         1
## 2 b         9
## 3 b        NA
## 4 c         4
## 5 c        15
```

- By default, `arrange()` ignores grouping.
- We can turn on `.by_group = TRUE` to sort the data within each pre-defined group.
- `NA`s are sorted to the end of each group.

ungroup() after each group_by()

- It is a good habit to use **ungroup()** at the end of a series of grouped operations.
- Otherwise the groupings will be carried in downstream analysis, which is not always desirable.

```
df5 <- ungroup(df4)
df5
```

```
## # A tibble: 5 x 2
##   Name      x
##   <chr> <dbl>
## 1 a         1
## 2 b         9
## 3 c         4
## 4 c        15
## 5 b        NA
```

The pipe operator %>%

- Notice what we do in the following code.

```
starwars_by_sex <- group_by(starwars, sex)
summarize(starwars_by_sex, mean_mass = mean(mass, na.rm = TRUE))
```

- Introduce a grouping in the data and then apply the mean function to the `mass` column within each sex group.
- We can revise the code using the pipe operator `%>%`
- Essentially, we "pipe" `starwars` into `group_by()`, and then the output from `group_by()` into `summarize()`.

```
starwars %>% group_by(sex) %>%
  summarize(mean_mass = mean(mass, na.rm = TRUE))
```

The pipe operator %>%

- The pipe takes the operation on its left, and passes it along to the function on its right.
- To further remove the missing values (NA) in the sex variable:

```
starwars %>%
  filter(!is.na(sex)) %>%
  group_by(sex) %>%
  summarize(mean_mass = mean(mass, na.rm = TRUE))
```

```
## # A tibble: 4 x 2
##   sex            mean_mass
##   <chr>              <dbl>
## 1 female              54.7
## 2 hermaphroditic    1358
## 3 male                80.2
## 4 none                69.8
```

Revisit the IMDA data set

- Recall in last week, we used `dplyr` verbs to prepare the data before plotting the IMDA data.

```r
young <- filter(media_data, age == "20-29",
                year == 2015) %>%
  mutate(pct = as.numeric(ever_used)) %>%
  arrange(desc(pct))
```

- Now you should be able to understand the commands.

Useful `summary()` functions

Here are some useful summary functions that come with `dplyr`:

- Measures of center: `mean()`, `median()`
- Measures of spread: `sd()`, `var()`, `IQR()`
- Measures of range: `min()`, `quantile()`, `max()`
- Measures of positions: `first(x)`, `nth(x, 2)`, `last(x)`
- Measures of count: `n()`, `n_distinct()`.

Useful summary() functions

```
# Find the shortest character
starwars %>%
  summarize(shortest = first(name, order_by = height))
```

```
## # A tibble: 1 x 1
##   shortest
##   <chr>
## 1 Yoda
```

```
# Count the number of characters by gender
starwars %>%
  group_by(gender) %>%
  summarize(n = n())
```

```
## # A tibble: 3 x 2
##   gender        n
##   <chr>     <int>
## 1 feminine     17
## 2 masculine    66
## 3 <NA>          4
```

Variants of `summarize()`

When we need to perform the same function(s) to a set of columns, we can use variants of `summarize()`.

- `summarize_all()` applies the functions to all columns.
- `summarize_at()` applies the functions to selected columns.
- `summarize_if()` applies the functions to columns that satisfy a certain condition.

Try out the following commands.

```r
starwars %>% summarize_at(vars(height:mass), mean, na.rm = TRUE)
starwars %>% summarize_if(is.numeric, mean, na.rm = TRUE)
```

Other useful functions

- `distinct()` finds all unique rows in a data set.

```r
# Remove duplicated rows, if any
starwars %>% distinct()

# Find all unique gender and hair_color pairs
starwars %>% distinct(gender, hair_color)

# Keep all columns and show the first row of distinct values
starwars %>% distinct(gender, hair_color, .keep_all = TRUE)
```

Other useful functions

- The `count()` function counts the number of occurrences.

```r
# Count occurrences of unique gender
starwars %>% count(gender)

# Count occurrences of unique gender and hair_color pairs
starwars %>% count(gender, hair_color)

# ... and sort in descending order of occurrences
starwars %>% count(gender, hair_color, sort = TRUE)
```

- The `rename()` function helps s to rename variables.

```r
# Rename "name" as "character_name"
starwars %>% rename(character_name = name)
```

Other useful functions

The `slice_` functions allow us to extract specific rows within each group.

- `slice_head(n = 1)` takes the first row from each group.
- `slice_tail(n = 1)` takes the last row from each group.
- `slice_min(x, n = 1)` takes the row with the smallest value of column `x`.
- `slice_max(x, n = 1)` takes the row with the largest value of column `x`.
- `slice_sample(n = 1)` takes one random row from each group.

```r
starwars %>%
  group_by(gender) %>%
  slice_max(mass, n = 1) %>%
  relocate(gender, mass)
```

Dealing with missing or duplicated value

- There are missing values in the `starwars` tibble.

```
# Remove all missing values
starwars %>% na.omit()

# Remove missing values in "gender"
starwars %>% filter(!is.na(gender))
```

Dealing with missing value

- Sometimes missing value is not coded as `NA`.
- Missing values can be represented by a value (e.g., `999`), a string (e.g., `none`, `unknown`), or just an empty cell.
- The following code reads `none` into `NA` across all columns:

```r
starwars %>% mutate_if(is.character, na_if, "none")
```

- After converting the value to `NA`, we can remove missing values with `na.omit()`.

```r
starwars %>% mutate_if(is.character, na_if, "none") %>% na.omit()
```

Case study: New York flights data

- The `nycflights13::flights` data contain all 336,776 flights that departed from New York City in 2013.
- You can read its documentation in `?flights`.
- We will use it to practice our `dplyr` skills.

flights data

```
# install.packages("nycflights13")
library(nycflights13)
data(flights)
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_ti
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <in
##   1  2013     1     1      517            515         2      830            8
##   2  2013     1     1      533            529         4      850            8
##   3  2013     1     1      542            540         2      923            8
##   4  2013     1     1      544            545        -1     1004           10
##   5  2013     1     1      554            600        -6      812            8
##   6  2013     1     1      554            558        -4      740            7
##   7  2013     1     1      555            600        -5      913            8
##   8  2013     1     1      557            600        -3      709            7
##   9  2013     1     1      557            600        -3      838            8
##  10  2013     1     1      558            600        -2      753            7
## # i 336,766 more rows
## # 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>
```

Subset observations by their values

`filter()` allows us to subset observations based on their values.

- Select all flights on February 7.

```
flights %>% filter(month == 2, day == 7)
```

Exercises: Find all flights that

- were delayed (on arrival or departure) by more than two hours.
- departed in summer (July, August, and September).
- had a missing `dep_time`.

Subset columns by their names

`select()` allows us to zoom in on a useful subset of variables based on their names.

- Select columns by name.

```
# 1
flights %>% select(dep_time, dep_delay, arr_time, arr_delay)
# 2
flights %>% select(starts_with("dep_"), starts_with("arr_"))
```

Exercise: What do the following commands do? Try them out.

```
# 3
flights %>% select_if(is.numeric)
# 4
df6 <- flights %>%
  filter(origin == "JFK") %>%
  select(year:day, dest, ends_with("delay"), distance, dep_time)
```

Extract hours and minutes from departure time

`mutate()` allows us to add new variables at the end of the data set.

- Let's work with the `df6` tibble we just created. Compute `hour` and `minute` from `dep_time`:

```
df6 %>%
  mutate(hour = dep_time %/% 100,
         minute = dep_time %% 100) %>% head(3)
```

```
## # A tibble: 3 x 10
##    year month   day dest  dep_delay arr_delay distance dep_time  hour minute
##   <int> <int> <int> <chr>     <dbl>     <dbl>    <dbl>    <int> <dbl>  <dbl>
## 1  2013     1     1 MIA           2        33     1089      542     5     42
## 2  2013     1     1 BQN          -1       -18     1576      544     5     44
## 3  2013     1     1 MCO          -3        -8      944      557     5     57
```

Summary on flight status

- Bucket flight status into three categories: `late`, `on time`, and `cancelled`. Then summarize the occurrences of flight status.

```r
flights %>%
  mutate(arr_status = ifelse(is.na(arr_delay), "cancelled",
                             ifelse(arr_delay <= 0, "on time", "late"))) %>%
  count(arr_status)
```

```
## # A tibble: 3 x 2
##   arr_status      n
##   <chr>       <int>
## 1 cancelled    9430
## 2 late       133004
## 3 on time    194342
```

Bucket flight status

- There is a much better option for categorization with more than two categories. Here is how it works:

```
# More convenient option for more than two categories
flights %>%
  mutate(arr_status =
           case_when(is.na(arr_delay) ~ "cancelled",
                     arr_delay <= 0   ~ "on time",
                     arr_delay > 0    ~ "late")) %>%
  count(arr_status)
```

- Not only is this much clearer code, it is more robust since it does not depend on the order we list the conditions.
- If we don't want to specify the last remaining condition explicitly, we can also enter `TRUE` for this condition.

Monthly mean departure delay

`arrange()` changes the order of the rows based on a set of column names.

- Order monthly mean departure delay in descending order:

```
df6 %>% group_by(month) %>%
  summarize(mean_dep_delay = mean(dep_delay, na.rm = TRUE)) %>%
  arrange(desc(mean_dep_delay))
```

Exercises:

- Find the five most delayed flights originated from JFK in 2013.
- Find the fastest (highest average speed) flights.

Destinations by the number of flights

The previous code pairs `summarize()` with `group_by()` to collapse a data frame into individual groups, before computing the summaries.

- Display destination airports with more than 5000 flights originated from JFK in 2013.

```
flights %>% filter(origin == "JFK") %>%
  count(dest) %>%
  filter(n > 5000)
```

```
## # A tibble: 4 x 2
##   dest      n
##   <chr> <int>
## 1 BOS    5898
## 2 LAX   11262
## 3 MCO    5464
## 4 SFO    8204
```

Summary

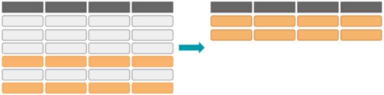
In this week we learn the key dplyr functions that allow you solve the vast majority of your data manipulation challenges:

- Subset observations by their values: filter()
- Subset variables by their names: select()
- Create new variables based on existing variables: mutate()
- Reorder the rows: arrange()
- Collapse many values down to a single summary: summarize()

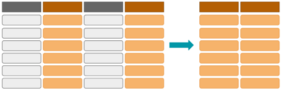
These functions can be used in conjunction with group_by(), which changes the scope of each function from operating on the entire data set to operating on it within groups.

Summary

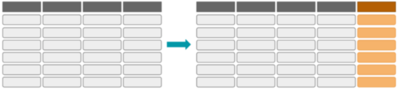
filter()



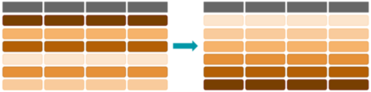
select()



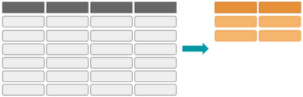
mutate()



arrange()



summarize()



More on `tidyverse`

- `ticyverse` is a suite of `R` packages designed for data science.
- Core packages include `dplyr`, `readr`, `stringr`, `forcats`, `purrr`, `ggplot2`.
- Learn more about it at https://www.tidyverse.org/packages/.