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()`, and `summarize()`
   - `group_by()` and `%>%`
2. **Tidy data** Week 6
   - `gather()`, `spread()`, `separate()`, and `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/).

Let us first install and load the package.

```r
# install.packages("tidyverse")
library(tidyverse)
```

Artwork by Allison Horst

Starwars data set

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

```
head(starwars)
```

```
## # A tibble: 6 x 14
##   name         height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex   gender hom
##   <chr>         <int> <dbl> <chr>   <chr>   <chr>     <dbl> <chr> <chr>  <ch
## 1 Luke Skywal~    172    77 blond   fair    blue       19   male  mascu~ Tat
## 2 C-3PO           167    75 <NA>    gold    yellow    112   none  mascu~ Tat
## 3 R2-D2            96    32 <NA>    white,~ red        33   none  mascu~ Nab
## 4 Darth Vader     202   136 none    white   yellow     41.9 male  mascu~ Tat
## 5 Leia Organa     150    49 brown   light   brown      19   fema~ femin~ Ald
## 6 Owen Lars       178   120 brown,~ light   blue       52   male  mascu~ Tat
## # ... with 4 more variables: species <chr>, films <list>, vehicles <list>,
## #   starships <list>, and abbreviated variable names 1: hair_color,
## #   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

```
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.

```
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
```

```
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.

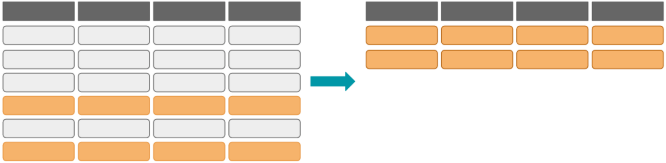
These operations can be chained using the **pipe operator `%>%`**.

Let's get started!

The `filter()` function

The `filter()` function subsets the rows in a data set 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", skin_color == "light")
```

```
## # A tibble: 6 x 14
##   name         height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex   gender hom
##   <chr>         <int> <dbl> <chr>   <chr>   <chr>     <dbl> <chr> <chr>  <ch
## 1 Leia Organa     150    49 brown   light   brown        19 fema~ femin~ Ald
## 2 Beru Whites~    165    75 brown   light   blue         47 fema~ femin~ Tat
## 3 Cordé           157    NA brown   light   brown        NA fema~ femin~ Nab
## 4 Dormé           165    NA brown   light   brown        NA fema~ femin~ Nab
## 5 Rey              NA    NA brown   light   hazel        NA fema~ femin~ <NA
## 6 Padmé Amida~    165    45 brown   light   brown        46 fema~ femin~ Nab
## # ... with 4 more variables: species <chr>, films <list>, vehicles <list>,
## #   starships <list>, and abbreviated variable names 1: hair_color,
## #   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

The `filter()` function

The output contains all observations that satisfy the stated conditions.

- When conditions are separated with a comma (,), `filter()` combines them using the **AND** operator.
- It does not modify the original data set. We need to assign the output to an object in order to save it.

Logical operators in `filter()` function

If we want to use other operators, such as the **OR** operation, we will have to manually specify them. To filter all light-skin female or light-skin male,

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

Logical operators in `filter()` function

```
## # A tibble: 11 x 14
##    name         height  mass hair_~1 skin_~2 eye_c~3 birth~4 sex   gender hom
##    <chr>         <int> <dbl> <chr>   <chr>   <chr>     <dbl> <chr> <chr>  <ch
##  1 Leia Organa    150    49 brown   light   brown        19 fema~ femin~ Ald
##  2 Owen Lars      178   120 brown,~ light   blue         52 male  mascu~ Tat
##  3 Beru White~    165    75 brown   light   blue         47 fema~ femin~ Tat
##  4 Biggs Dark~    183    84 black   light   brown        24 male  mascu~ Tat
##  5 Lobot          175    79 none    light   blue         37 male  mascu~ Bes
##  6 Cordé          157    NA brown   light   brown        NA fema~ femin~ Nab
##  7 Dormé          165    NA brown   light   brown        NA fema~ femin~ Nab
##  8 Raymus Ant~    188    79 brown   light   brown        NA male  mascu~ Ald
##  9 Rey             NA    NA brown   light   hazel        NA fema~ femin~ <NA
## 10 Poe Dameron     NA    NA brown   light   brown        NA male  mascu~ <NA
## 11 Padmé Amid~    165    45 brown   light   brown        46 fema~ femin~ Nab
## # ... with 4 more variables: species <chr>, films <list>, vehicles <list>,
## #   starships <list>, and abbreviated variable names 1: hair_color,
## #   2: skin_color, 3: eye_color, 4: birth_year, 5: homeworld
```

Logical operators in `filter()` function

The `%in%` operator matches conditions provided in a vector constructed using the combine function `c()`.

- Simplifies the previous command:

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

- The code reads: Give me the observations in the `starwars` tibble 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.

- If we do the same task in base R, the command will be:

```
starwars[starwars$skin_color == "light" &
           (starwars$sex == "female" | starwars$sex == "male"), ]
```

Compared to base R functions

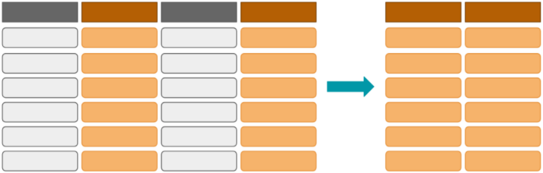
You can see the advantage of the `tidyverse` syntax:

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

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

# tidyverse 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

When we want to zoom in on a particular set of variables, we can use the select() command.

- To select columns by name

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

- Compared to the base R bracket method

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

The select() function

Compared to the base R method, the dplyr language is much easier to read and 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))
```

Functions to assist `select()`

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”.
- `matches(".a.")` matches columns whose names match the provided expression.

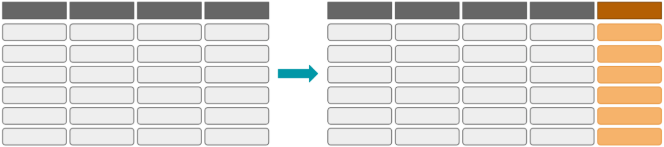
For example, to select all columns that end with **color**.

```
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 set with more columns than the original data.



- `mutate()` always adds the new columns to the **end** of the data set.

The `mutate()` function

Let us first create a new data set with fewer columns so that we can see the manipulation results more easily.

```r
starwars_small = select(starwars, name, height, mass, species)
head(starwars_small, n = 3)
```

```
## # A tibble: 3 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  
```

The `mutate()` function

The following code creates two new columns to the end of the data set

```
data = mutate(starwars_small,
              height_m = height/100, BMI = mass/(height_m^2))
head(data, n = 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 syntax that does the same task?

Variant functions to `mutate()`

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

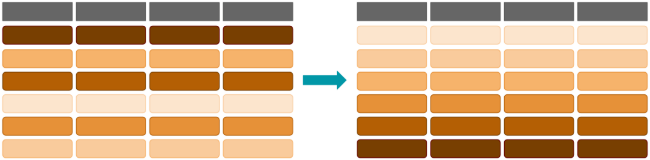
- `mutate_if()` first requires a function that returns a boolean to select columns. If the condition is met, the mutate function will be applied on those variables.
- `mutate_at()` requires selection of a set of columns via the `var()` argument. The mutate function will be applied on the selected columns.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

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

```
mutate_if(starwars_small, is.character, tolower)
```

The `arrange()` function

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



- 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 data in ascending order.

```
arrange(starwars_small, mass)
```

```
## # A tibble: 87 x 4
##    name                  height  mass species
##    <chr>                  <int> <dbl> <chr>
##  1 Ratts Tyerell             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 Dud Bolt                  94    45 Vulptereen
##  8 Padmé Amidala            165    45 Human
##  9 Wat Tambor               193    48 Skakoan
## 10 Sly Moore                178    48 <NA>
## # ... with 77 more rows
```

The `arrange()` function

To arrange a column in descending order, use the `desc` operator.

```
arrange(starwars_small, 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 Human
## 10 Dexter Jettster          198   102 Besalisk
## # ... with 77 more rows
```

Compared to base R functions

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

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

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

```
starwars_small[order(rev(starwars_small$mass)), ]
```

The `arrange()` function

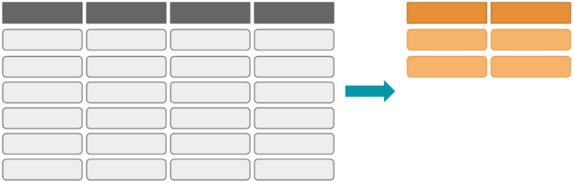
- To arrange the data first by mass (in ascending order), then by height (in descending order):

```
arrange(starwars_small, mass, desc(height))
```

```
## # A tibble: 87 x 4
##    name                  height  mass species
##    <chr>                  <int> <dbl> <chr>
##  1 Ratts Tyerell             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            165    45 Human
##  8 Dud Bolt                  94    45 Vulptereen
##  9 Wat Tambor               193    48 Skakoan
## 10 Sly Moore                178    48 <NA>
## # ... with 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   174.
```

- `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()` helper

This 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 specifying `.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.

```
data2 = tibble(Name = c("a", "b", "c", "c", "b"),
               x = c(1, 9, 4, 15, NA))
```

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

```
data3 = group_by(data2, Name)
data3
```

```
## # 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
```

`filter()` by group

- Filter out group(s) with value smaller or equal to the sample median.

```
filter(data3, x <= median(x))
```

```
## # A tibble: 2 x 2
## # Groups:   Name [2]
##   Name      x
##   <chr> <dbl>
## 1 a         1
## 2 c         4
```

- Group `b` is missing from the result. Why?

mutate() by group

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

```
# Replace NAs with 0
mutate(data3, 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.

```
arrange(data3, 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 need to turn on `.by_group = TRUE` in order to sort the data within each pre-defined group.
- `NA`s are sorted to the end of each group.

`summarize()` by group

- Compute the mean of `x` within each group and name the new variable as `x_mean`.

```
summarize(data3, 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
```

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.

```
data4 = ungroup(data3)
data4
```

```
## # 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 did on the previous slide.

- Introduce a grouping in the data set and then apply the mean function to the height variable within each gender group.

We can revise the code using the pipe operator %>%

- Essentially, we “pipe” the output from group_by() into summarize().

Example

- To calculate the average mass for each sex group:

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

- The following is an equivalent command using the %>% operator:

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

```
## # A tibble: 5 x 2
##   sex            mean_height
##   <chr>                <dbl>
## 1 female                54.7
## 2 hermaphroditic      1358
## 3 male                  81.0
## 4 none                  69.8
## 5 <NA>                  48
```

Example

- To further remove the NAs in the grouping variable:

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

```
## # A tibble: 4 x 2
##   sex            mean_height
##   <chr>                <dbl>
## 1 female                54.7
## 2 hermaphroditic      1358
## 3 male                  81.0
## 4 none                  69.8
```

Revisit the IMDA data set

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

```
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 `tidyverse`:

- Measures of center: `mean()` and `median()`
- Measures of spread: `sd()`, `var()`, and `IQR()`
- Measures of rank: `min()`, `quantile()`, and `max()`
- Measures of positions for vector x: `first(x)`, `nth(x, 2)`, and `last(x)`

Useful summary functions

Example:

- The shortest and tallest Star Wars characters in the data set.

```
starwars %>%
  summarize(shortest = first(name, order_by = height),
            tallest = last(name, order_by = height))
```

```
## # A tibble: 1 x 2
##   shortest tallest
##   <chr>    <chr>
## 1 Yoda     Captain Phasma
```

Variants of `mutate()`

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

- `mutate_all()` applies the functions to all columns.
- `mutate_at()` applies the functions to selected columns.
- `mutate_if()` applies the functions to columns that satisfy a certain condition.

Similar variants exist for `summarize()`.

Variants of `mutate()`

```
set.seed(101)
data3$y = rnorm(5)
data3
```

```
## # A tibble: 5 x 3
## # Groups:   Name [3]
##   Name      x      y
##   <chr> <dbl>  <dbl>
## 1 a         1 -0.326
## 2 b         9  0.552
## 3 c         4 -0.675
## 4 c        15  0.214
## 5 b        NA  0.311
```

What do these commands do? Try them out:

```
data3 %>% mutate_all(abs)
data3 %>% mutate_at(vars(x, y), abs)
data3 %>% mutate_if(is.double, round, digits = 2)
```

Dealing with missing or duplicated value

Create another simple data frame:

```r
data5 = tibble(group = c("a", "b", "b", "c", "c", "c"),
                   x = c(1, 9, NA, 15, 15, 999))
```

- Use `na.omit()` to remove rows with missing values.
- Use `distinct()` to remove duplicated rows.
- What do these commands do? Try them out:

```r
data5 %>% na.omit()
data5 %>% distinct()
data5 %>% na.omit() %>% distinct()
```

Dealing with missing value

Sometimes the missing value is not coded as `NA`.

- Missing values can be represented by a value (e.g., `999`), a string (e.g., `unknown`), or just an empty cell.

The following code reads `999` into `NA` across all columns:

```
data5 %>% mutate_all(na_if, 999)
```

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

Exercises: 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 data manipulation skills.

flights data

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

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 car
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <ch
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Subset observations by their values

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

- Select all flights on January 1, 2013.

```
filter(flights, month == 1, day == 1)
```

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.

- (Many possible ways to) select columns by name.

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

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

```
# 3
select_if(flights, is.numeric)
# 4
flights_small = 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 flights_small tibble we just created. Compute hour and minute from dep_time:

```
flights_small %>% select(year:day, dest, dep_time) %>%
  mutate(hour = dep_time %/% 100,
         minute = dep_time %% 100) %>% head(3)
```

```
## # A tibble: 3 x 7
##    year month   day dest  dep_time  hour minute
##   <int> <int> <int> <chr>    <int> <dbl>  <dbl>
## 1  2013     1     1 MIA        542     5     42
## 2  2013     1     1 BQN        544     5     44
## 3  2013     1     1 MCO        557     5     57
```

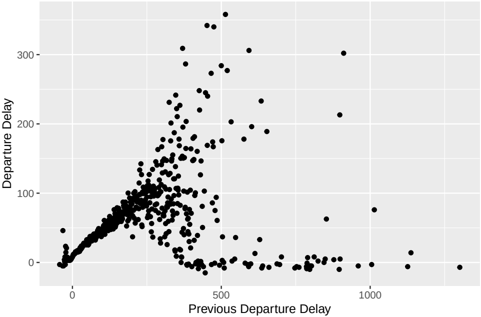
Relationship between flight delays

Delays are typically temporally correlated.

- Even after the problem that caused the initial delay has been resolved, later flights would still be delayed to allow earlier flights to leave.
- The following code uses the offset functions, `lag()` and `lead()`, to explore how the delay of a flight is related to the delay of the immediately preceding flight.

```r
lagged_delays = flights %>%
  arrange(origin, month, day, dep_time) %>%
  group_by(origin) %>%
  # remove cancelled flights
  filter(!is.na(dep_delay)) %>%
  # departure delay of the preceding flight from the same airport
  mutate(lag_dep_delay = lag(dep_delay)) %>%
  select(origin, month, day, dep_delay, lag_dep_delay)
```

```
lagged_delays %>%
  # remove the first flight of a day
  filter(!is.na(lag_dep_delay)) %>%
  # compute mean departure delay for each lagged delay value
  group_by(lag_dep_delay) %>%
  summarize(dep_delay_mean = mean(dep_delay)) %>%
  # create a scatter plot
  ggplot(aes(x = lag_dep_delay, y = dep_delay_mean)) +
  geom_point() +
  labs(y = "Departure Delay", x = "Previous Departure Delay")
```



Note: We will introduce `ggplot()` after the midterm exam.

Bucket flight status

Bucket flight status into three categories: `late`, `on time`, and `cancelled`.

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

```
## # A tibble: 3 x 2
##   arr_status  count
##   <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:

```r
# 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")) %>%
  group_by(arr_status) %>%
  summarize(count = n())
```

- 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:

```
flights_small %>% 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_small %>% group_by(dest) %>%
  summarize(count = n()) %>%
  filter(count > 5000)
```

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

Additional questions

Use the `nycflights13::flights` data to answer the following questions:

1. Which destination receives the most flights in December?
2. Which carrier has the worst delays?
3. If you are flying to Baltimore (BWI) from New York, what time of day should you fly if you want to avoid delays as much as possible?
4. How would you disentangle the effects of bad airports vs. bad carriers?

Try to solve these questions by yourself first. We will discuss them in the next lecture.

Summary

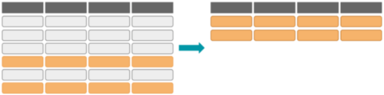
In this week we learn the key `tidyverse` functions that allow you solve the vast majority of your data manipulation challenges:

- 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()`

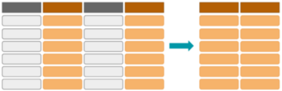
These functions can be used in conjunction with `group_by()`, which changes the scope of each function from operating on the entire data set to operating on it within groups.

Summary

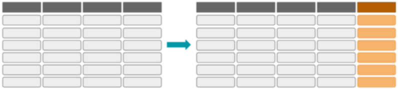
filter()



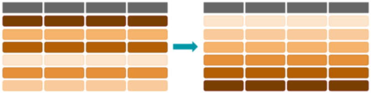
select()



mutate()



arrange()



summarize()