Code Sheet for DSCI 101: Fundamentals of Modern Data Science with R

Kejin Wu
Department of Mathematics and Statistics, Loyola University Chicago
kwu8@luc.edu

2025-11-29

Abstract

The materials in this code sheet is collected from the lecture slides of DSCI 101 I will keep updating this code sheet to enrich it. If you see any typo, please contact me. Thank you! Credit is given to Dr. Widad Abdalla Mukhaimer at LUC. The course materials are mainly based on his lecture design.

Contents

Creating Functions

```
calories_women <- function(weight, height, age){
  calories <- (10 * weight) + (6.25 * height) - (5 * age) - 161
  return(calories)
}

calories_women(weight = 65, height = 170, age = 35)
```

```
## [1] 1376.5
```

```
calories_women(65, 170, 35)
```

```
## [1] 1376.5
```

Creating functions with default values

```
calories_women_fixed <- function(weight, height, age = 30){
  calories <- (10 * weight) + (6.25 * height) - (5 * age) - 161
  return(calories)
}

calories_women_fixed(weight = 65, height = 170)
```

```
## [1] 1401.5
```

```
calories_women_fixed(weight = 65, height = 170, age = 35)
```

```
## [1] 1376.5
```

Data Reading and Selecting

Read data from csv file

```
Illustration_Data <- read_csv("PUT YOUR DIRECTORY HERE")
```

```
library(tidyverse)
library(Lahman)
library(babynames)
library(mdsr)
library(palmerpenguins)
Illustration_Data <- read_csv("Illustration_Data.csv")
Illustration_Data
```

Read data from built-in packages

You can use `?dataset_name` to check the descirption of the dataset.

```
library(palmerpenguins)
data("penguins")
?penguins
```

slice: Display the first few rows

```
slice(penguins, 1:3)
```

```
## # A tibble: 3 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
## 1 Adelie  Torgersen           39.1          18.7               181        3750
## 2 Adelie  Torgersen           39.5          17.4               186        3800
## 3 Adelie  Torgersen           40.3          18                 195        3250
## # i 2 more variables: sex <fct>, year <int>
```

glimpse: Get summary of the variables

```
glimpse(penguins)
```

```
## Rows: 344
## Columns: 8
## $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
## $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
## $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
## $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
## $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
## $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
## $ sex               <fct> male, female, female, NA, female, male, female, male~
## $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

str: Display the structure of the data

```
str(penguins)
```

```
## tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
##  $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
##  $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
##  $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
##  $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
##  $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
##  $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

table(): use it on specific columns.Shows how many times each observation appeared

```
table(penguins$sex)
```

```
##
## female   male
##    165    168
```

```
table(penguins$sex, useNA = "ifany")
```

```
##
## female   male   <NA>
##    165    168     11
```

colnames(): shows you the names of the columnns in the dataframe.

```
colnames(penguins)
```

```
## [1] "species"           "island"            "bill_length_mm"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"
```

Select columns "Age" and "total_Income"

```
example <- Illustration_Data %>%
  select(Age, total_Income)
```

Select columns that start with "var"

```
example <- Illustration_Data %>%
  select(starts_with("var"))
```

Select columns that contains "total" in their names

```
example <- Illustration_Data %>%
  select(contains("total"))
```

Exclude columns "honesty" and "zipcode"

```
example <- Illustration_Data %>%
  select(-honesty, -zipcode)
```

Exclude some specific columns

```
example <- Illustration_Data %>%
  select(c(1,3,5,6))
```

Data Filtering

`filter` uses Boolean logic.

Simple Conditions (one boolean expression)

Examples: Greater than (`>`), less than (`<`), or equal to (`==`)

```
example <- Illustration_Data %>%
  filter(Age > 30)

example <- Illustration_Data %>%
  filter(honesty == "agree")
```

Multiple Conditions -

You can combine conditions using logical operators like `&` (AND) and `|` (OR).

```r
example <- Illustration_Data %>%
  filter(Age > 30 & total_Income < 50000)

example <- Illustration_Data %>%
  filter(Age > 20 & total_Income < 30000 & zipcode == 60073)

example <- Illustration_Data %>%
  filter(Age > 30 | total_Income > 20000)
```

Exclusion

To exclude certain rows, you can use the `!=` operator (not equal to).

```r
example <- Illustration_Data %>%
  filter(zipcode != 60111)
```

Filter rows based on vector of conditions

The `%in%` operator is useful for filtering rows with values in a specified vector.

```r
example <- Illustration_Data %>%
  filter(var_1 %in% c("bananas", "grapes"))
```

Data Mutating

mutate(): Creates a new column typically based on computations related to other columns in the dataset.

```r
example <- Illustration_Data %>%
  mutate(income_per_month = total_Income / 12)
```

mutate with ifelse

We can use other functions (like `ifelse`) within mutate to help us make a new variable.

```r
example <- Illustration_Data %>%
  mutate(status = ifelse(Age > 30, "Older", "Younger"))
```

mutate case_when()

```r
example <- Illustration_Data %>%
  mutate(group = case_when(
    Age < 30 ~ "Young",
    Age >= 30 & Age <= 40 ~ "Middle-Aged",
    Age > 40 ~ "Old"
  ))
```

If you have multiple condition you can use the `case_when()` function and list out your possible options.

mutate cut()

```r
SAT_2010 <- SAT_2010 %>%
  mutate(ptr = ifelse(pupil_teacher_ratio >= 15, "ptr - high","ptr - low"),
         sal = ifelse(salary >= 52000, "sal - high","sal - low"),
         SAT_rate = cut(
```

```
          sat_pct,
          breaks = c(0, 30, 60, 100),
          labels = c("low", "medium", "high")
        ))
```

Change variable type

You can use the mutate function to change the variable type.

```
Illustration_Data_new <- Illustration_Data %>%
  mutate(total_Income = as.character(total_Income))
```

Data Summarizing

Using `group_by()` and `summarise()` together allows you to efficiently compute summary statistics, aggregations, or any other computations of data based on different groups defined by one or more variables.

group_by() and summarise()

1. group_by(): is used to group a data frame by one or more variables. This creates a "grouped" data frame where subsequent operations are performed within each group separately. This works best with categorical variables or factor variables.Using group_by() in it's own doesn't change the "look" of the data.
2. summarise(): is used to compute summary statistics or other values for each group. It condenses the grouped data into a single row per group, summarizing the specified variables.

```
example <- Illustration_Data %>%
  group_by(var_1) %>%
  summarise(mean_income = mean(total_Income),
            median_income = median(total_Income))
```

Count observations for each group

```
example <- Illustration_Data %>%
  group_by(honesty) %>%
  summarise(Num_Pro = n())
```

Data Counting and Arranging

arrange()

```
example<-Illustration_Data %>%
  arrange(cat_total)
```

Descending arrange()

```
example<-Illustration_Data %>%
  arrange(desc(cat_total))
```

Arrange data in alphabetical order by name.

```
example <-Illustration_Data %>%
  arrange(Name)
```

```
example <- Illustration_Data %>%
  arrange(honesty, cat_total)
```

Joining Dataset

In the tidyverse, joining two datasets together is a way to combine data from different sources based on common variables. The dplyr package within the tidyverse provides functions to perform these joins.

```
Illustration_Data_1 <- read_csv("Illustration_Data_1.csv")
Illustration_Data_2 <- read_csv("Illustration_Data_2.csv")
Illustration_Data_3 <- read_csv("Illustration_Data_3.csv")
Illustration_Data_1
Illustration_Data_2
Illustration_Data_3
```

Inner join

An inner join combines rows from two datasets where there's a match between the specified variables. Rows with no matching values are excluded. Inner joins return results if the keys are matched in BOTH tables.

```
example_1 <- Illustration_Data_1 %>%
  inner_join(Illustration_Data_2, c("Name" = "First_Name"))
example_1
```

```
## # A tibble: 4 x 5
##   Name       Age num_kids Last_Name    Gender
##   <chr>    <dbl>    <dbl> <chr>        <chr>
## 1 Val         18        1 Chmerkovskiy Male
## 2 Derek       25        0 Hough        Male
## 3 Whitney     30        2 Carson       Female
## 4 Daniella    45        1 Karagach     Female
```

```
example_2 <- Illustration_Data_1 %>%
  inner_join(Illustration_Data_3, c("Name" = "Name"))
example_2
```

```
## # A tibble: 4 x 5
##   Name    Age num_kids Last_Name    Car
##   <chr> <dbl>    <dbl> <chr>        <chr>
## 1 Val      18        1 Chmerkovskiy Mercedes
## 2 Val      18        1 Chmerkovskiy Tesla
## 3 Val      18        1 Chmerkovskiy Audi
## 4 Derek    25        0 Hough        Ferrari
```

Left join

A left join includes all rows from the left dataset and the matching rows from the right dataset. If there's no match, the columns from the right dataset will be filled with NA. Here the rows of the first table are always returned, regardless of whether there is a match in the second table.

A left join is equivalent to `inner_join` when the rows in left dataset can be matched all according to the chosen comman variable from the right dataset.

```r
example_1 <- Illustration_Data_1 %>%
  left_join(Illustration_Data_2, by = c("Name" = "First_Name"))
example_1
```

```
## # A tibble: 4 x 5
##   Name       Age num_kids Last_Name    Gender
##   <chr>    <dbl>    <dbl> <chr>        <chr>
## 1 Val         18        1 Chmerkovskiy Male
## 2 Derek       25        0 Hough        Male
## 3 Whitney     30        2 Carson       Female
## 4 Daniella    45        1 Karagach     Female
```

```r
example_2 <- Illustration_Data_2 %>%
  left_join(Illustration_Data_1, by = c("First_Name" = "Name"))
example_2
```

```
## # A tibble: 7 x 5
##   First_Name Last_Name    Gender   Age num_kids
##   <chr>      <chr>        <chr>  <dbl>    <dbl>
## 1 Val        Chmerkovskiy Male      18        1
## 2 Derek      Hough        Male      25        0
## 3 Whitney    Carson       Female    30        2
## 4 Sasha      Farber       Male      NA       NA
## 5 Daniella   Karagach     Female    45        1
## 6 Lindsay    Arnold       Female    NA       NA
## 7 Mark       Ballas       Male      NA       NA
```

```r
example_3 <- Illustration_Data_3 %>%
  left_join(Illustration_Data_1, by = c("Name" = "Name"))
example_3
```

```
## # A tibble: 6 x 5
##   Name    Last_Name    Car        Age num_kids
##   <chr>   <chr>        <chr>    <dbl>    <dbl>
## 1 Val     Chmerkovskiy Mercedes    18        1
## 2 Val     Chmerkovskiy Tesla       18        1
## 3 Val     Chmerkovskiy Audi        18        1
## 4 Derek   Hough        Ferrari     25        0
## 5 Lindsay Arnold       Tesla       NA       NA
## 6 Mark    Ballas       BMW         NA       NA
```

Count NA values

We can count the NA values by using the `summarize` function

```r
example_6 <- example_2 %>%
  summarize(num_people = n(),
            num_na = sum(is.na(Age)),
            num_not_na = sum(!is.na(Age)))
example_6
```

```
## # A tibble: 1 x 3
##   num_people num_na num_not_na
##        <int>  <int>      <int>
## 1          7      3          4
```

Full join

A full join includes all rows from both datasets. Columns from the dataset with missing values will be filled with NA where there's no match.

```
example_1 <- Illustration_Data_1 %>%
  full_join(Illustration_Data_2, by = c("Name" = "First_Name"))
example_1
```

```
## # A tibble: 7 x 5
##   Name       Age num_kids Last_Name    Gender
##   <chr>    <dbl>    <dbl> <chr>        <chr>
## 1 Val         18        1 Chmerkovskiy Male
## 2 Derek       25        0 Hough        Male
## 3 Whitney     30        2 Carson       Female
## 4 Daniella    45        1 Karagach     Female
## 5 Sasha       NA       NA Farber       Male
## 6 Lindsay     NA       NA Arnold       Female
## 7 Mark        NA       NA Ballas       Male
```

```
example_2 <- Illustration_Data_1 %>%
  full_join(Illustration_Data_3, by = c("Name" = "Name"))
example_2
```

```
## # A tibble: 8 x 5
##   Name       Age num_kids Last_Name    Car
##   <chr>    <dbl>    <dbl> <chr>        <chr>
## 1 Val         18        1 Chmerkovskiy Mercedes
## 2 Val         18        1 Chmerkovskiy Tesla
## 3 Val         18        1 Chmerkovskiy Audi
## 4 Derek       25        0 Hough        Ferrari
## 5 Whitney     30        2 <NA>         <NA>
## 6 Daniella    45        1 <NA>         <NA>
## 7 Lindsay     NA       NA Arnold       Tesla
## 8 Mark        NA       NA Ballas       BMW
```

Joining dataset with multiple columns

Join by multiple columns: The `by` argument specifies the multiple columns that should be used for matching. This is useful when the only **one** column is not sufficient to match the rows between two datasets.

Tidy Data

Data often comes in various formats, and its structure might not be ideal for the task at hand. Pivoting helps you reorganize your data to a format that makes it easier to analyze, visualize, or model.

pivot_longer()

`pivot_longer()` is used to convert data from a wide format (with multiple columns) into a long format (fewer columns). It's particularly useful when you have variables spread across different columns and you want to stack them into a single column, often with corresponding values.

NOTE: All columns not listed in cols will stay as-is (they're treated as ID variables).

```
data_wide <- data.frame(
  Country = c("USA", "Canada"),
  Y_2018 = c(250, 180),
```

```
  Y_2019 = c(260, 190),
  Y_2020 = c(270, 200)
)
data_wide
```

```
##   Country Y_2018 Y_2019 Y_2020
## 1     USA    250    260    270
## 2  Canada    180    190    200
```

```
wide_to_long <- data_wide %>%
  pivot_longer(cols = -Country, names_to = "Year", values_to = "Value")

wide_to_long
```

```
## # A tibble: 6 x 3
##   Country Year   Value
##   <chr>   <chr>  <dbl>
## 1 USA     Y_2018   250
## 2 USA     Y_2019   260
## 3 USA     Y_2020   270
## 4 Canada  Y_2018   180
## 5 Canada  Y_2019   190
## 6 Canada  Y_2020   200
```

pivot_longer()

`pivot_wider()` is used to convert data from a long format to a wide format. It's useful when you want to take distinct values from a column and spread them across new columns.

```
data_long <- data.frame(
  Country = c("USA", "Canada"),
  Year = rep(c("2018", "2019", "2020"), each = 2),
  Value = c(250, 260, 270, 180, 190, 200)
)
data_long
```

```
##   Country Year Value
## 1     USA 2018   250
## 2  Canada 2018   260
## 3     USA 2019   270
## 4  Canada 2019   180
## 5     USA 2020   190
## 6  Canada 2020   200
```

```
long_to_wide <- data_long %>%
  pivot_wider(names_from = Year, values_from = Value)
long_to_wide
```

```
## # A tibble: 2 x 4
##   Country `2018` `2019` `2020`
##   <chr>    <dbl>  <dbl>  <dbl>
## 1 USA        250    270    190
## 2 Canada     260    180    200
```

Combination with other tidyverse functions

Using flights dataset (from nycflights13 package), create a dataset where each airline is a row and each column is a month (1-12) to see which is the busiest months for each airline. In the "core" of this table you should have the number of flights per carrier for each of the months.

```
library(nycflights13)
question_1<-flights %>%
  select(carrier, month) %>%
  group_by(carrier, month) %>%
  summarize(Number_of_flights = n()) %>%
  pivot_wider(names_from = month, values_from = Number_of_flights)
```

```
## `summarise()` has grouped output by 'carrier'. You can override using the
## `.groups` argument.
```

ggplot

Fundamental components of ggplot2:

1. Data: The dataset you're working with.
2. Aesthetics Mapping (aes): How data variables map to plot aesthetics, like position, color, shape, etc.
3. Geometric Objects (geom): The visual elements to represent the data (points, lines, bars, etc.).
4. Facets (facet_wrap or facet_grid): Splitting data into subplots based on a variable.
5. Theme: Controlling the overall appearance of the plot.
6.

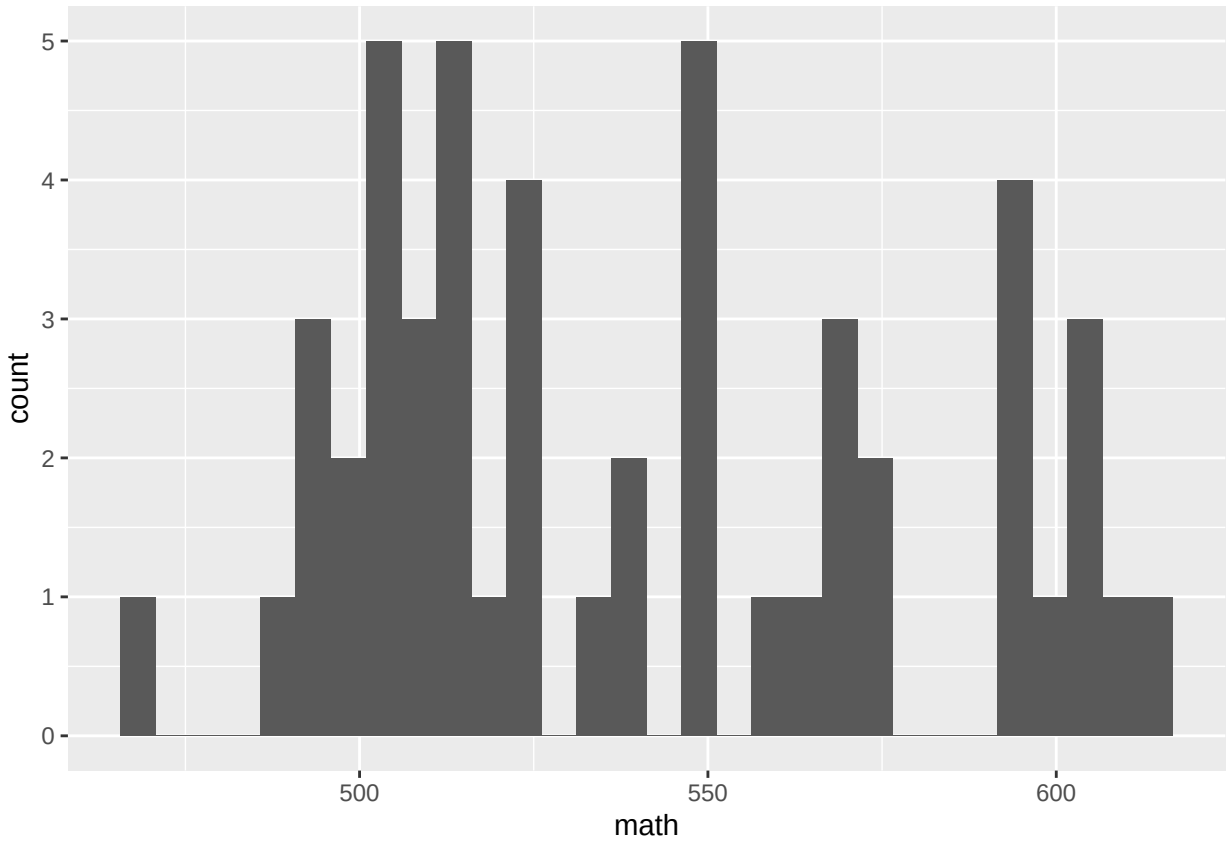
We can think of every command as a layer like the pipe for `dplyr`

```
SAT_2010 <- SAT_2010 %>%
  mutate(salary_level = case_when(
    salary < 52000 ~ "Low",
    salary >= 52000 & salary < 60000 ~ "Medium",
    salary >= 60000 ~ "High"
  ))
```

Histograms

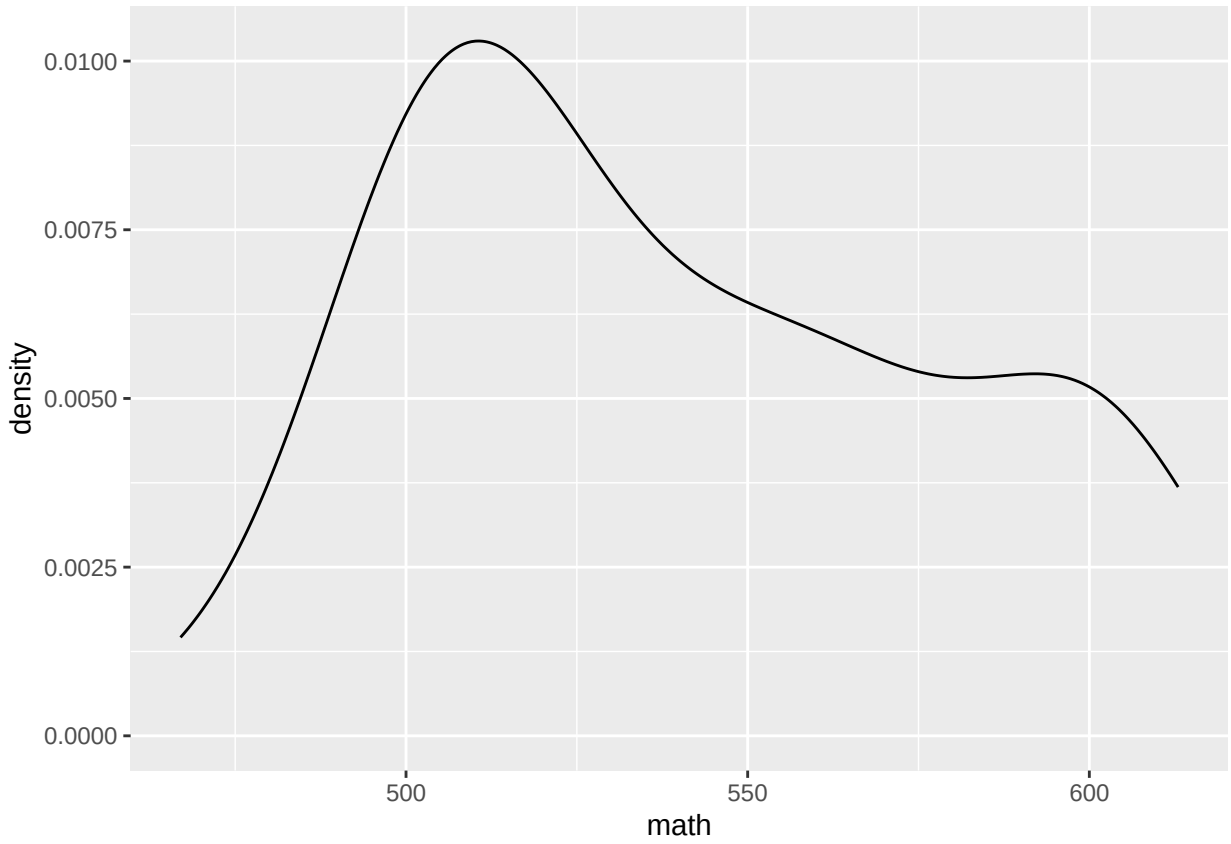
```
fig <- SAT_2010 %>%
  ggplot(aes(x = math)) +
  geom_histogram()
fig
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



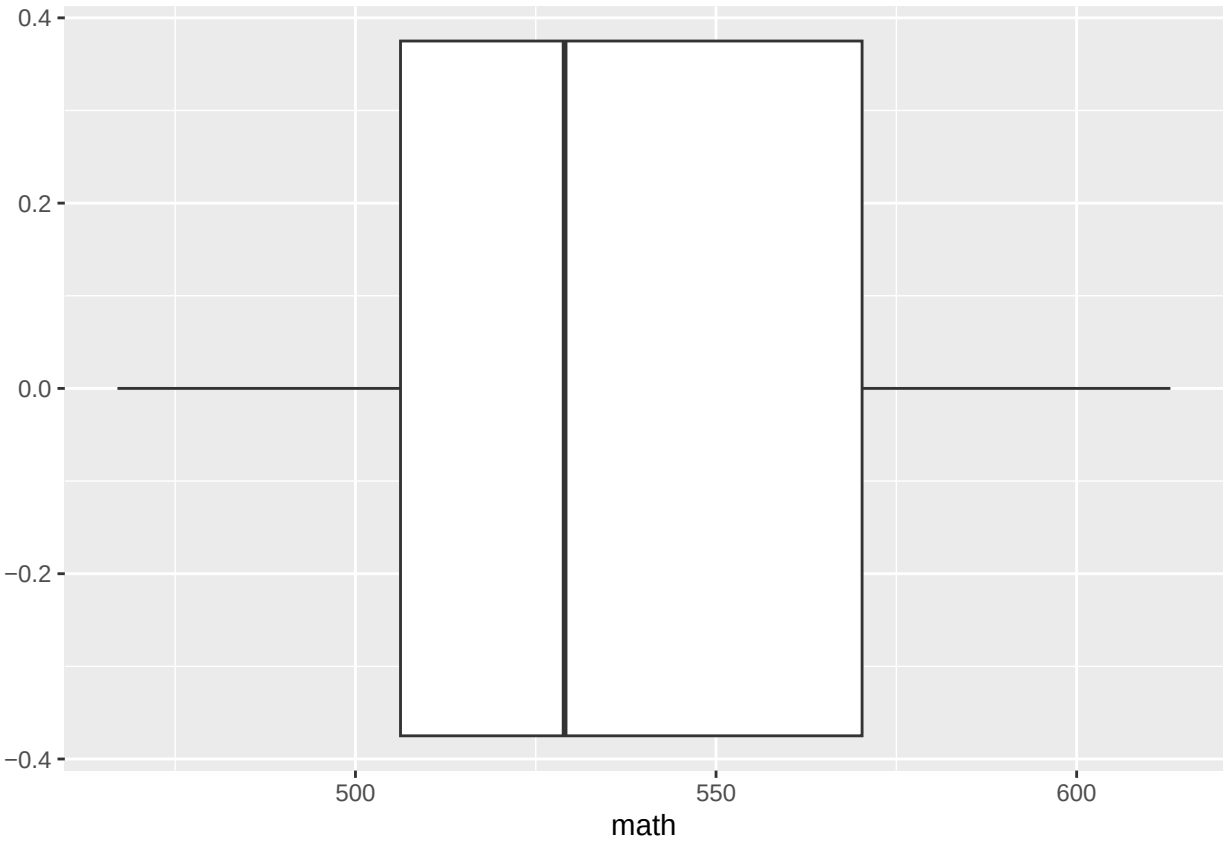
Density Plots

```
fig <- SAT_2010 %>%
  ggplot(aes(x = math)) +
  geom_density()
fig
```



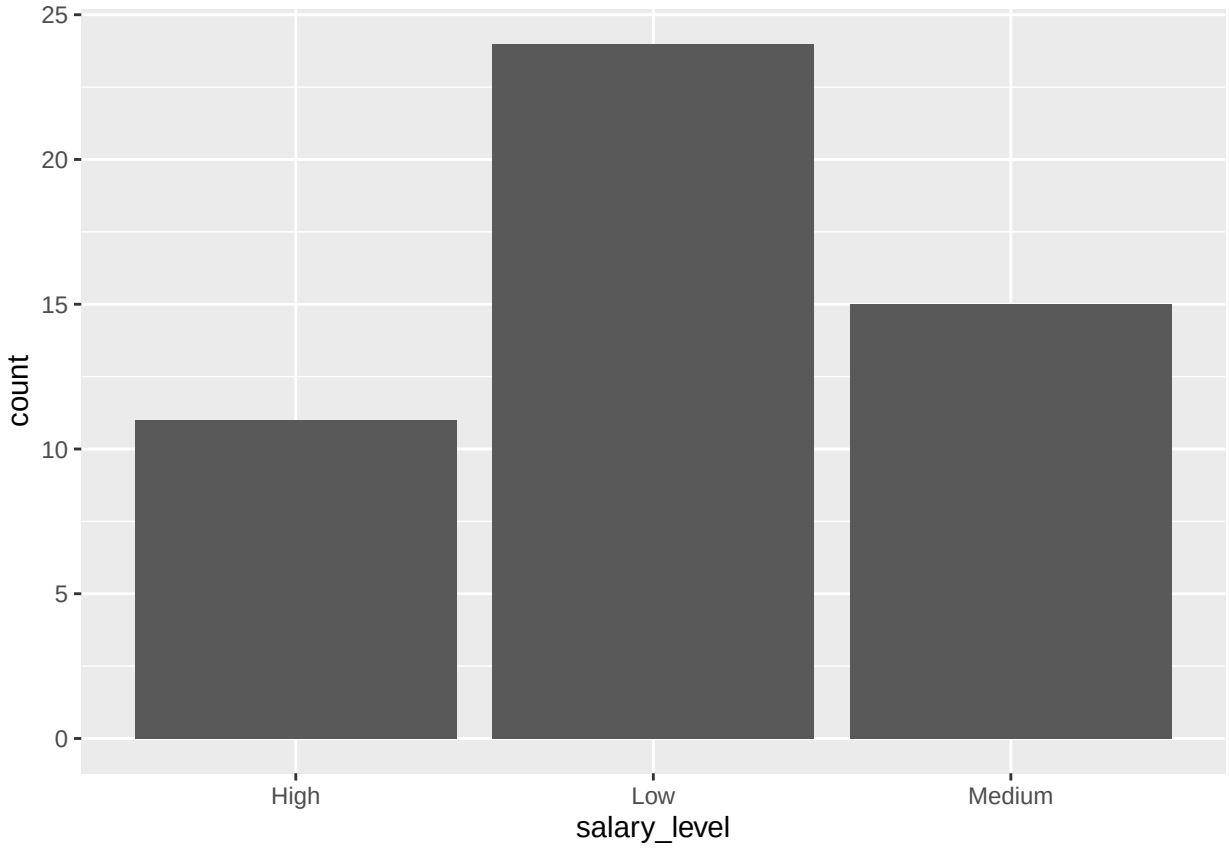
Boxplot

```
fig <- SAT_2010 %>%
  ggplot(aes(x = math)) +
  geom_boxplot()
fig
```



Barplot

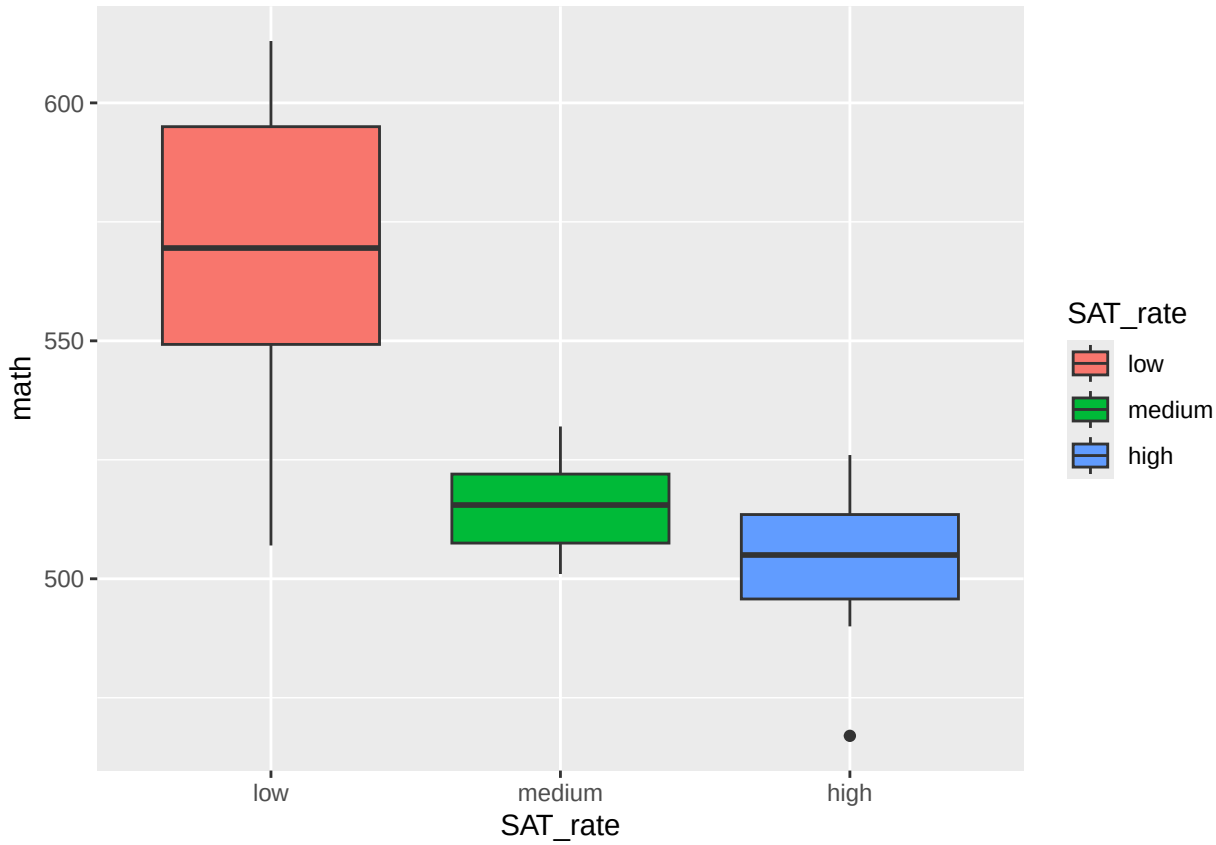
```
fig <- SAT_2010 %>%
  ggplot(aes(x = salary_level)) +
  geom_bar()
fig
```



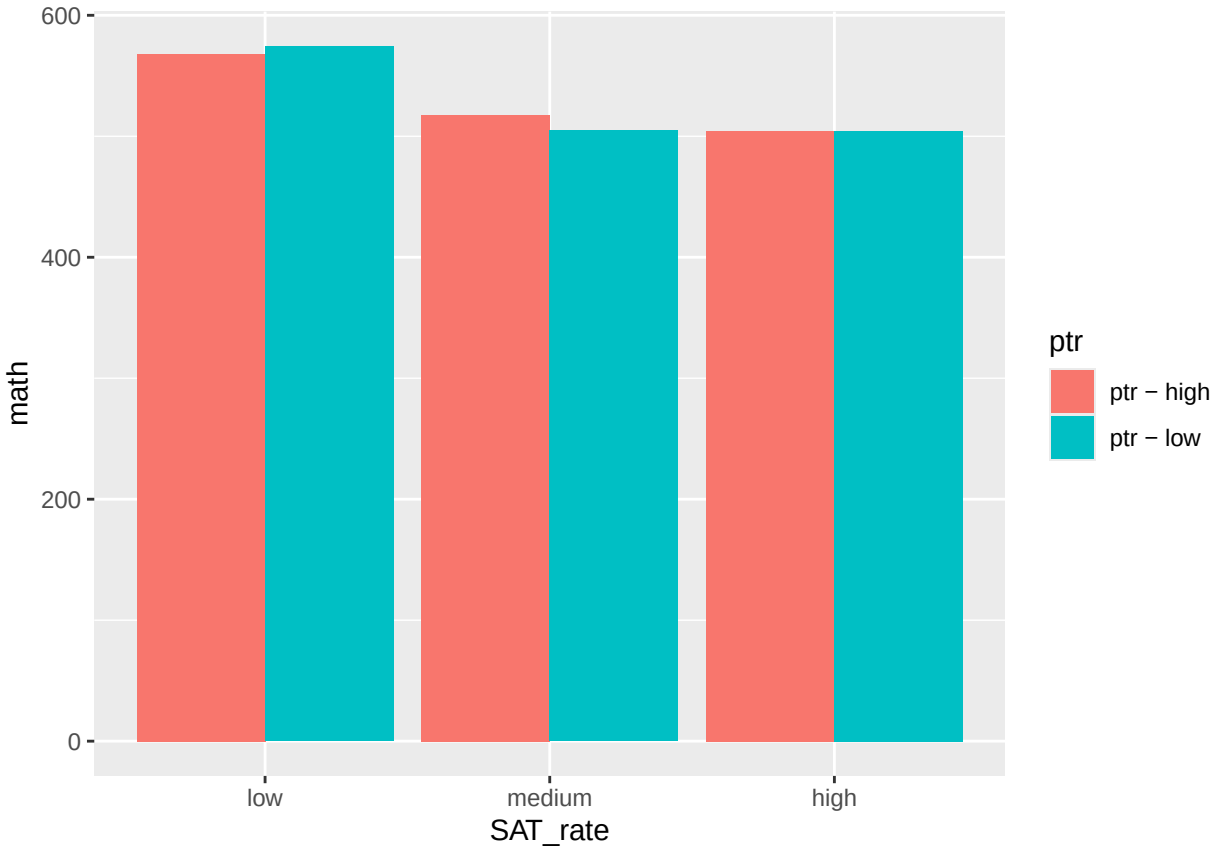
Plot with color defined by other variables

```
SAT_2010 <- SAT_2010 %>%
  mutate(ptr = ifelse(pupil_teacher_ratio >= 15, "ptr - high","ptr - low"),
         sal = ifelse(salary >= 52000, "sal - high","sal - low"),
         SAT_rate = cut(
           sat_pct,
           breaks = c(0, 30, 60, 100),
           labels = c("low", "medium", "high")
         ))

fig <- SAT_2010 %>%
  ggplot(aes(x = SAT_rate, y = math, fill = SAT_rate)) +
  geom_boxplot()
fig
```

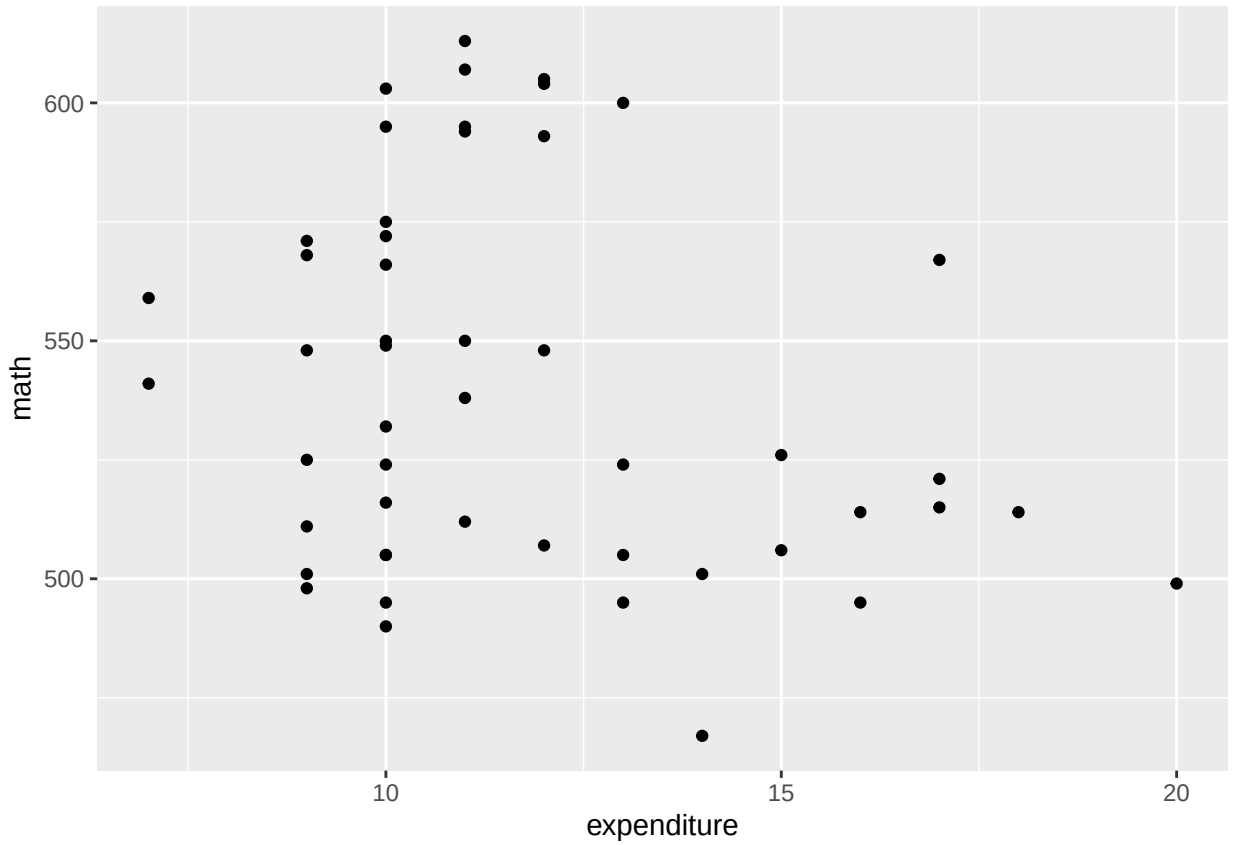



```
fig <- SAT_2010 %>%
  ggplot(aes(x = SAT_rate, y = math, fill = ptr)) +
  stat_summary(fun = mean, geom = "bar",position = "dodge")
fig
```



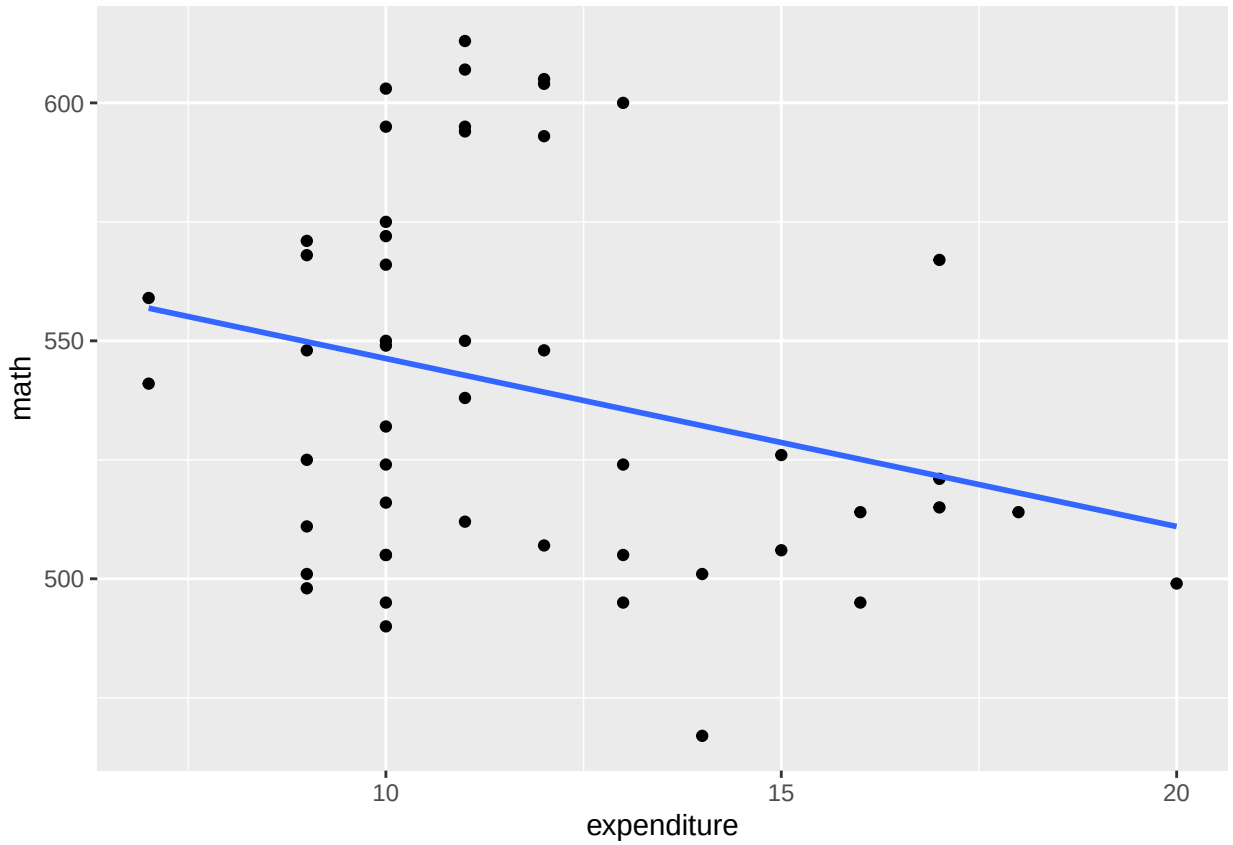
Scatter Plots with different shapes and/or colors

```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math)) +
  geom_point()
fig
```



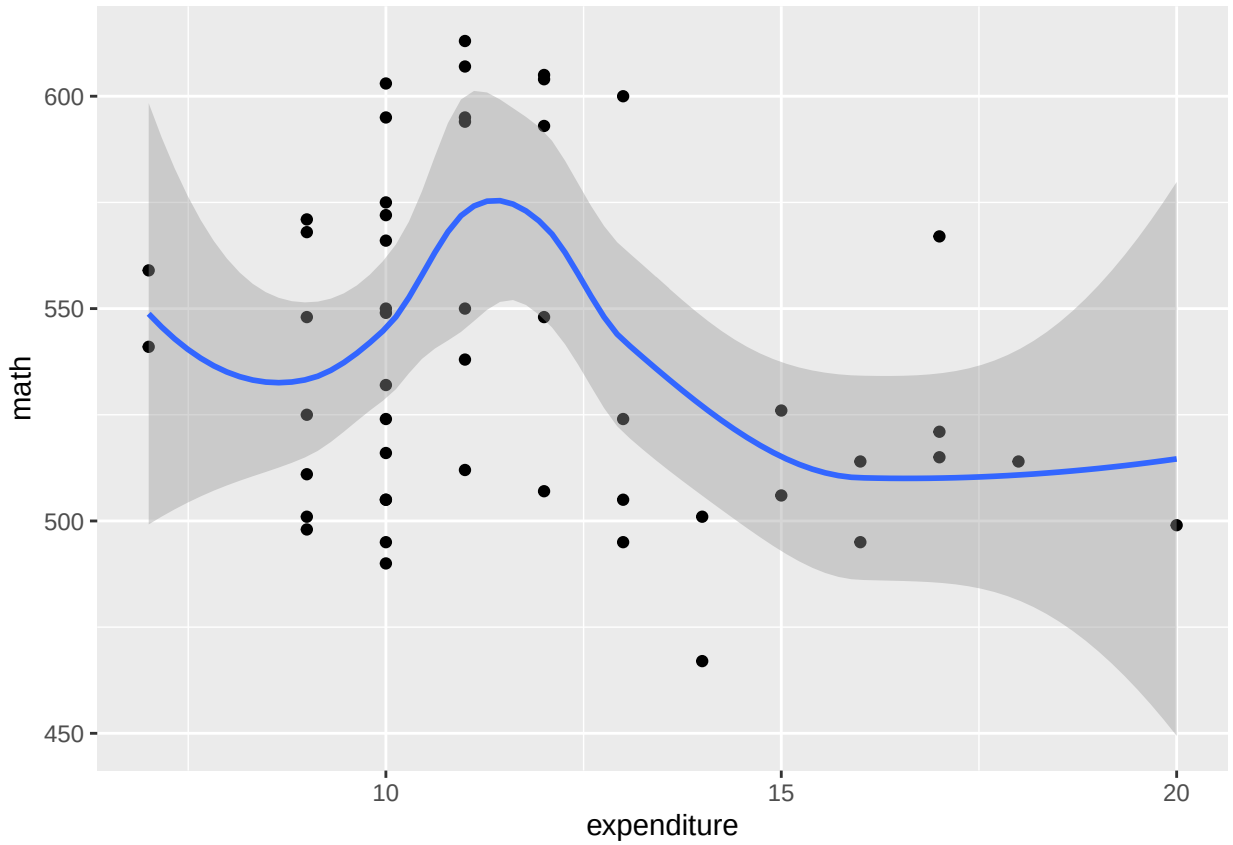

```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
fig
```

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

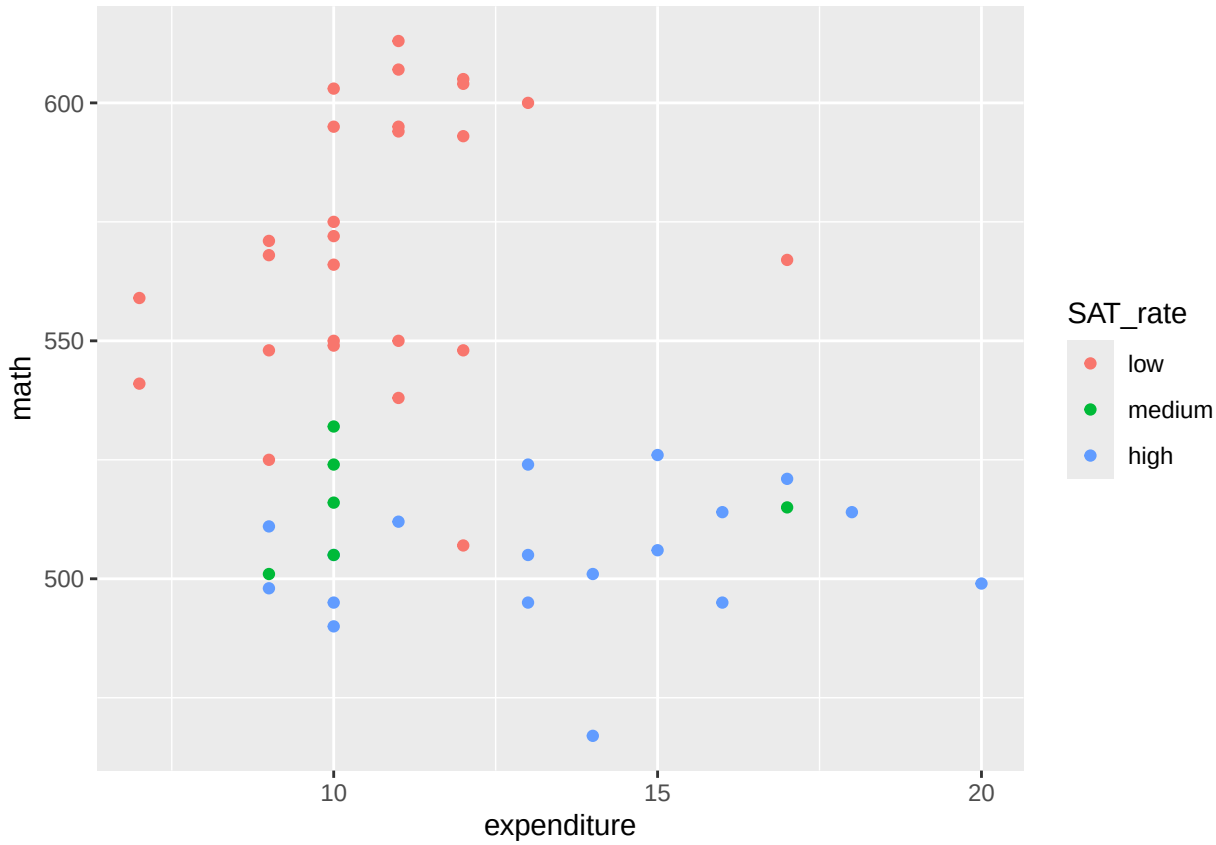



```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math)) +
  geom_point() +
  geom_smooth(method = "loess", se = TRUE)
fig
```

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

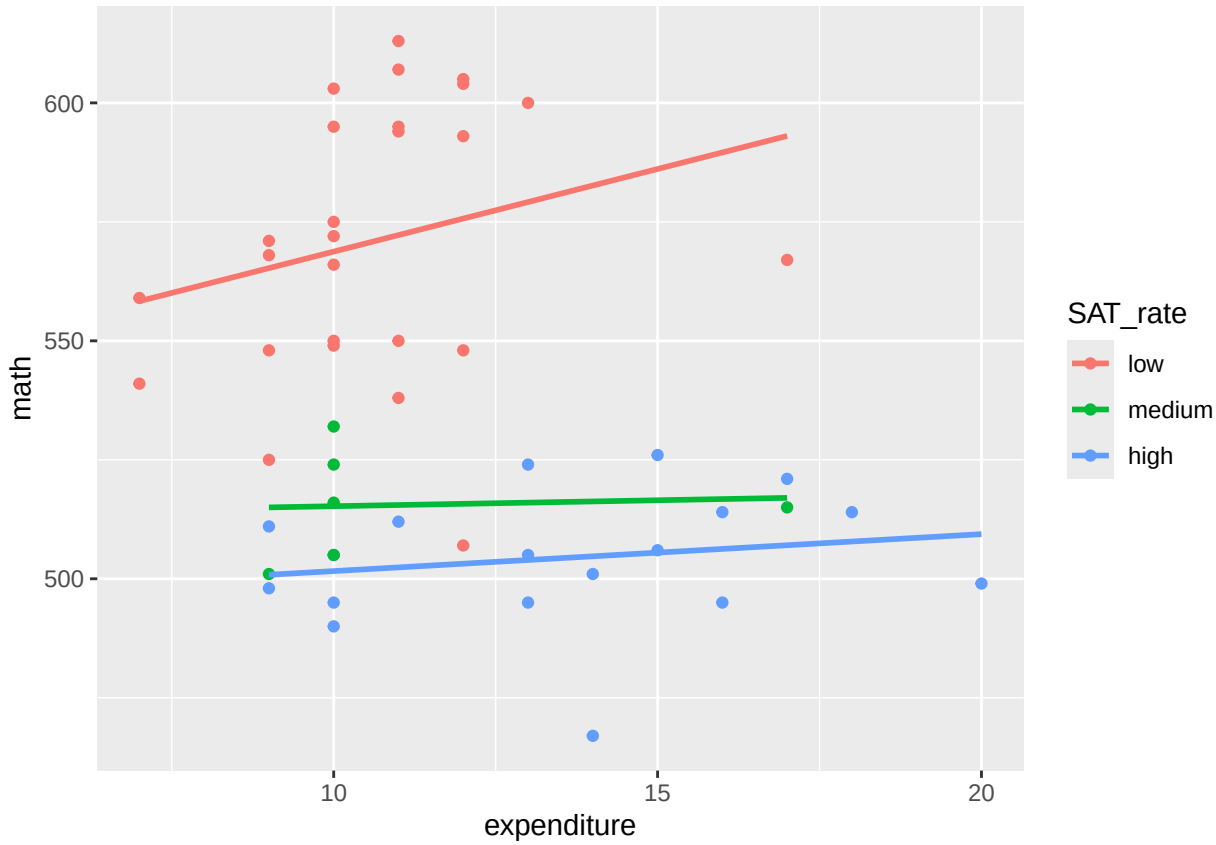



```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math, color = SAT_rate)) +
  geom_point()
fig
```

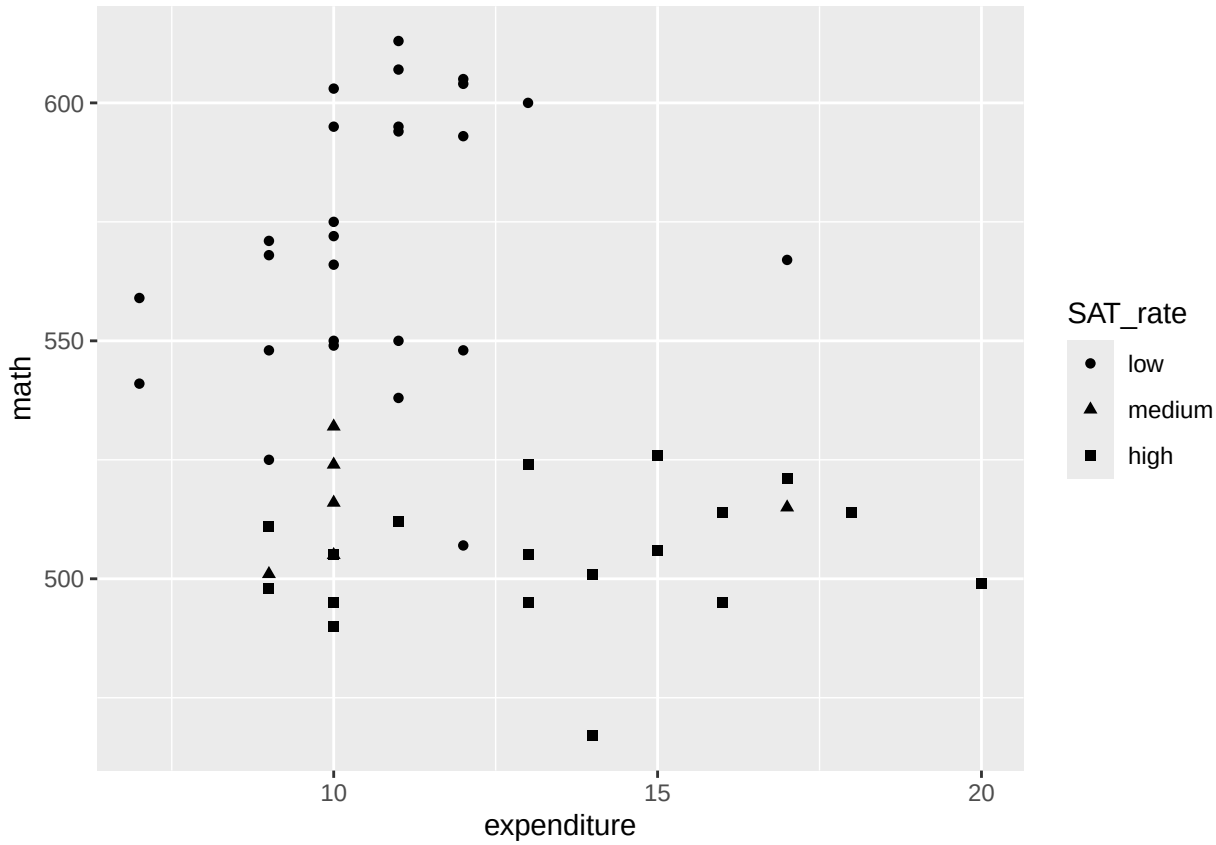



```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math, color = SAT_rate)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
fig
```

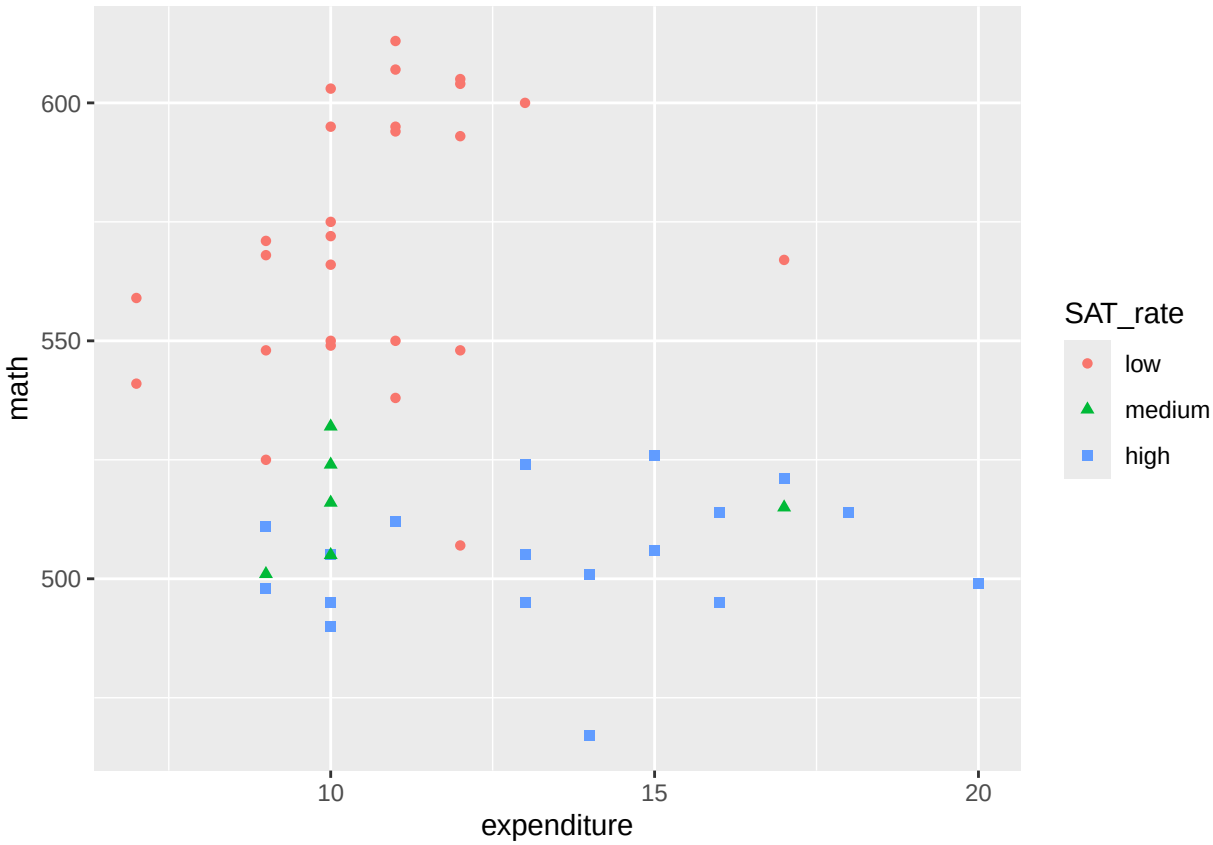
```
## `geom_smooth()` using formula = 'y ~ x'
```




```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math, shape = SAT_rate)) +
  geom_point()
fig
```




```
fig_14 <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math, color = SAT_rate, shape = SAT_rate)) +
  geom_point()
fig_14
```



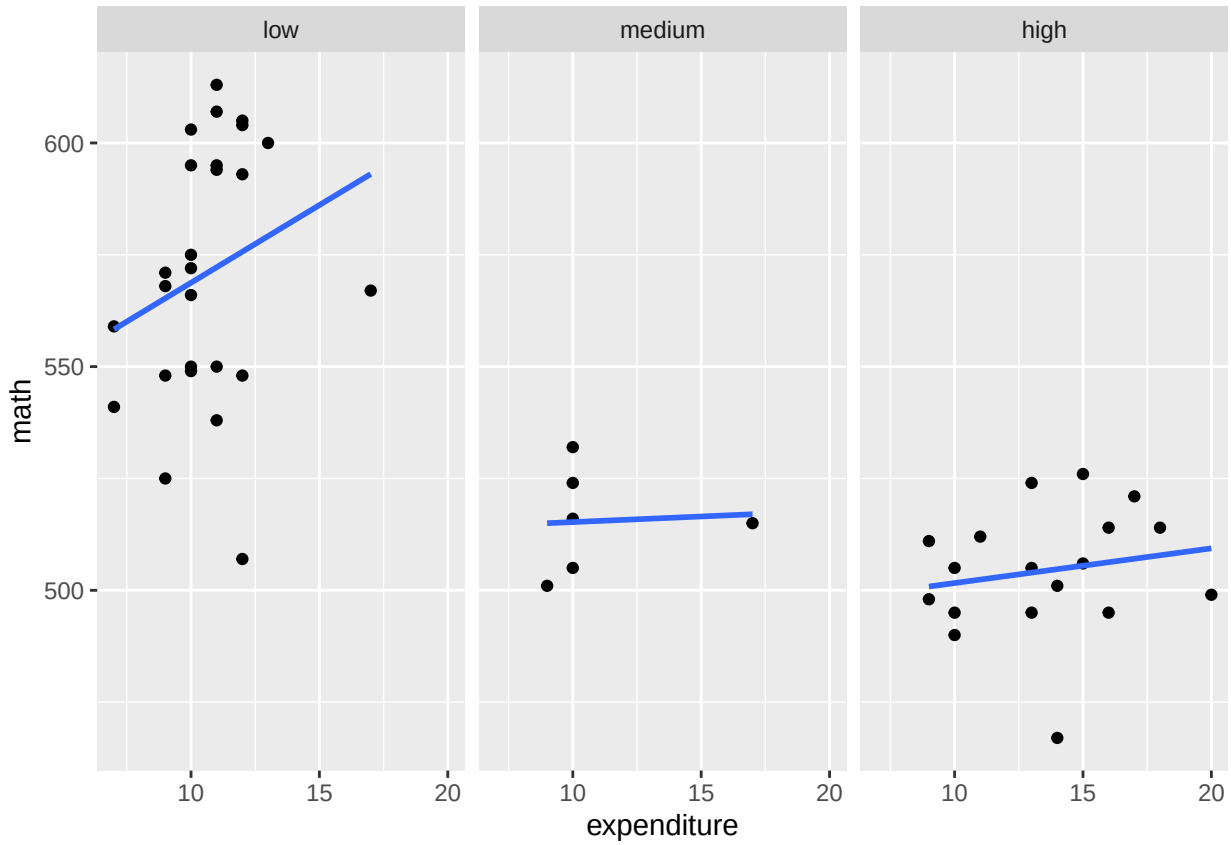
Faceting

This is similar to using shape or color for a categorical variables but puts them on separate plots.

Facet with 1 variable:

```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  facet_wrap(~SAT_rate)
fig
```

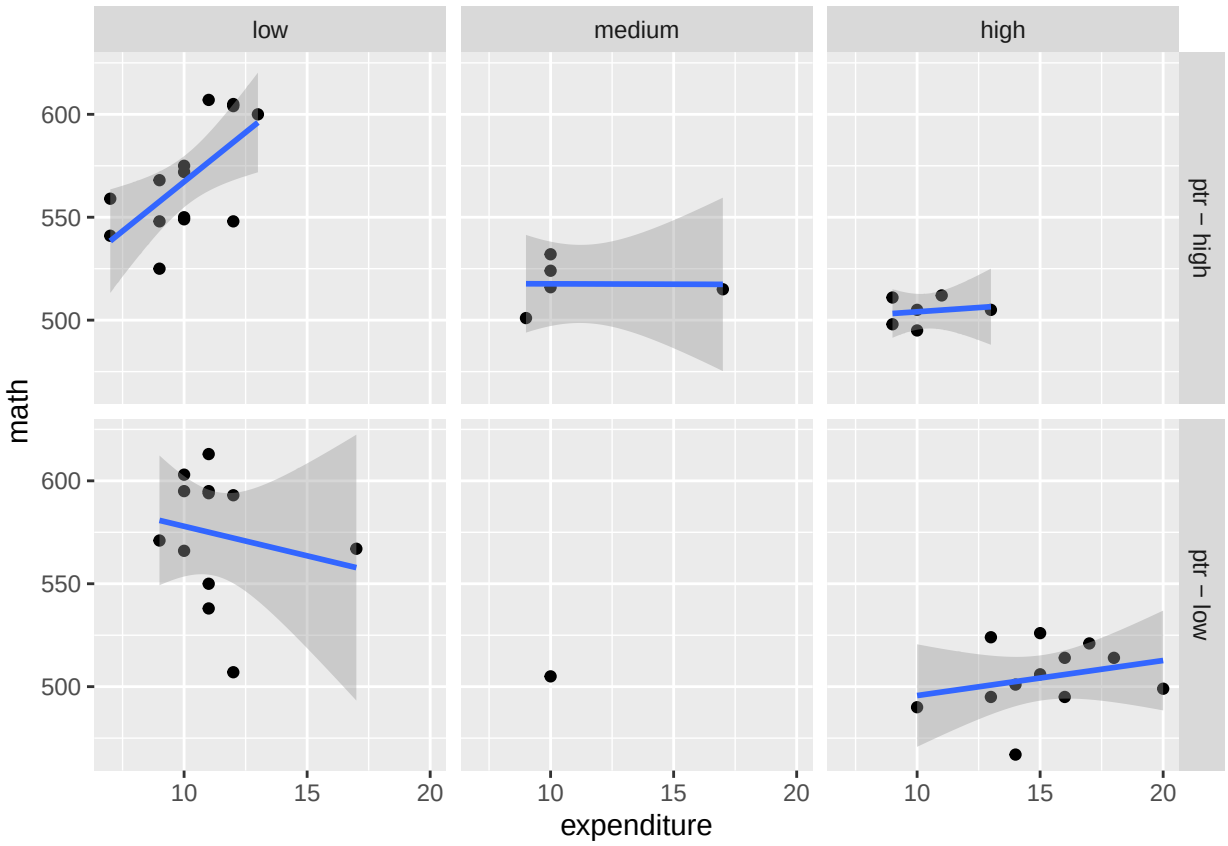
```
## `geom_smooth()` using formula = 'y ~ x'
```



Facet with 2 variables:

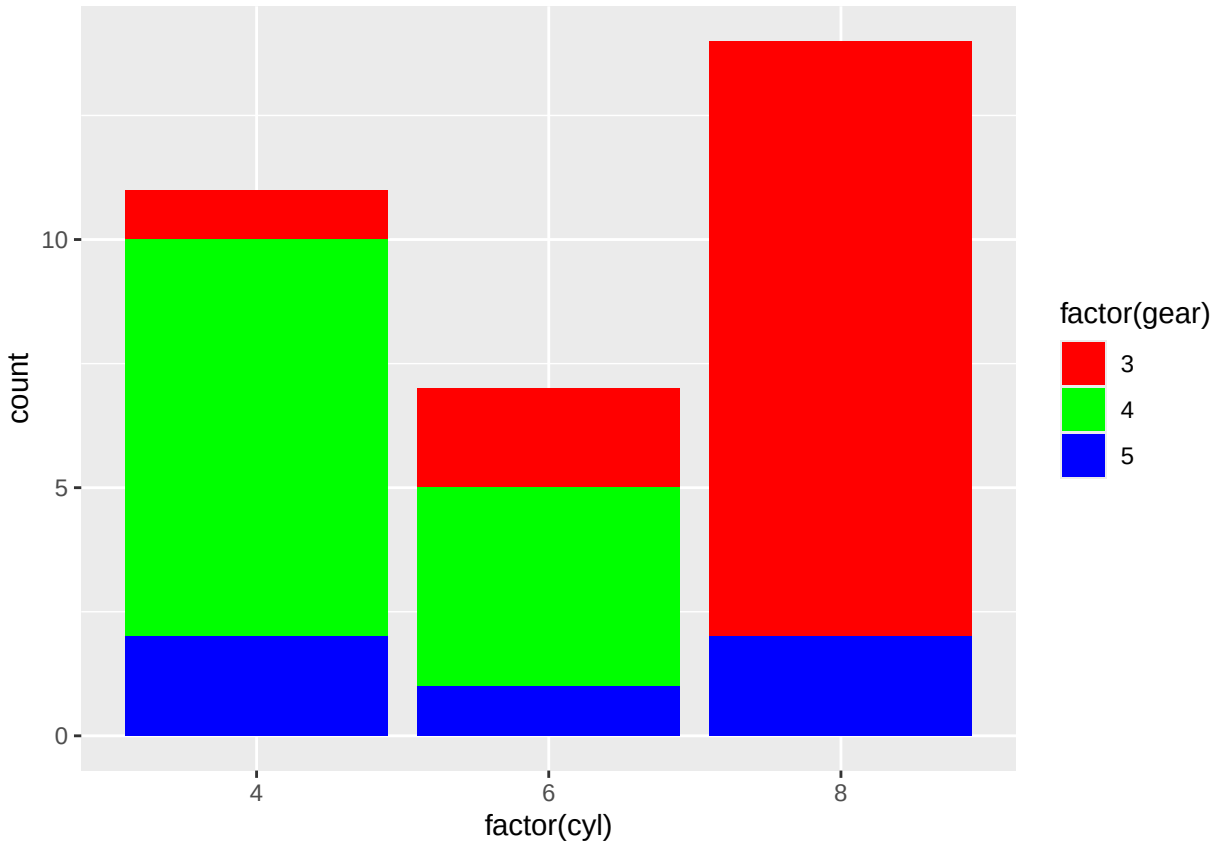
```
fig <- SAT_2010 %>%
  ggplot(aes(x = expenditure, y = math)) +
  geom_point() +
  geom_smooth(method = "lm", se = TRUE) +
  facet_grid(ptr ~ SAT_rate)
fig
```

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



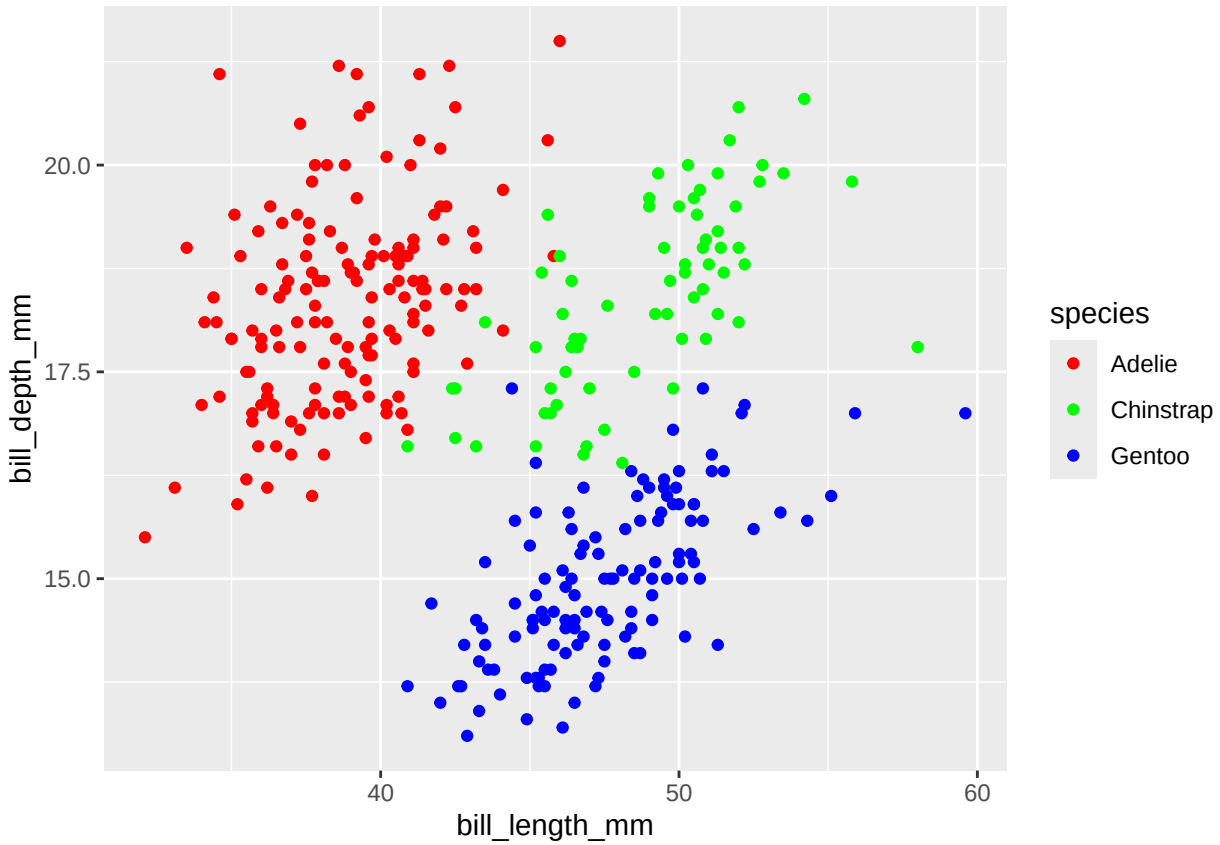
Multivariate displys

```
fig <- mtcars %>%
  ggplot(aes(x = factor(cyl), fill = factor(gear)))+
  geom_bar()+
  scale_fill_manual(values = c("red", "green", "blue"))
fig
```




```
fig <- penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm, color = species)) +
  geom_point() +
  scale_color_manual(values = c("red", "green", "blue"))
fig
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



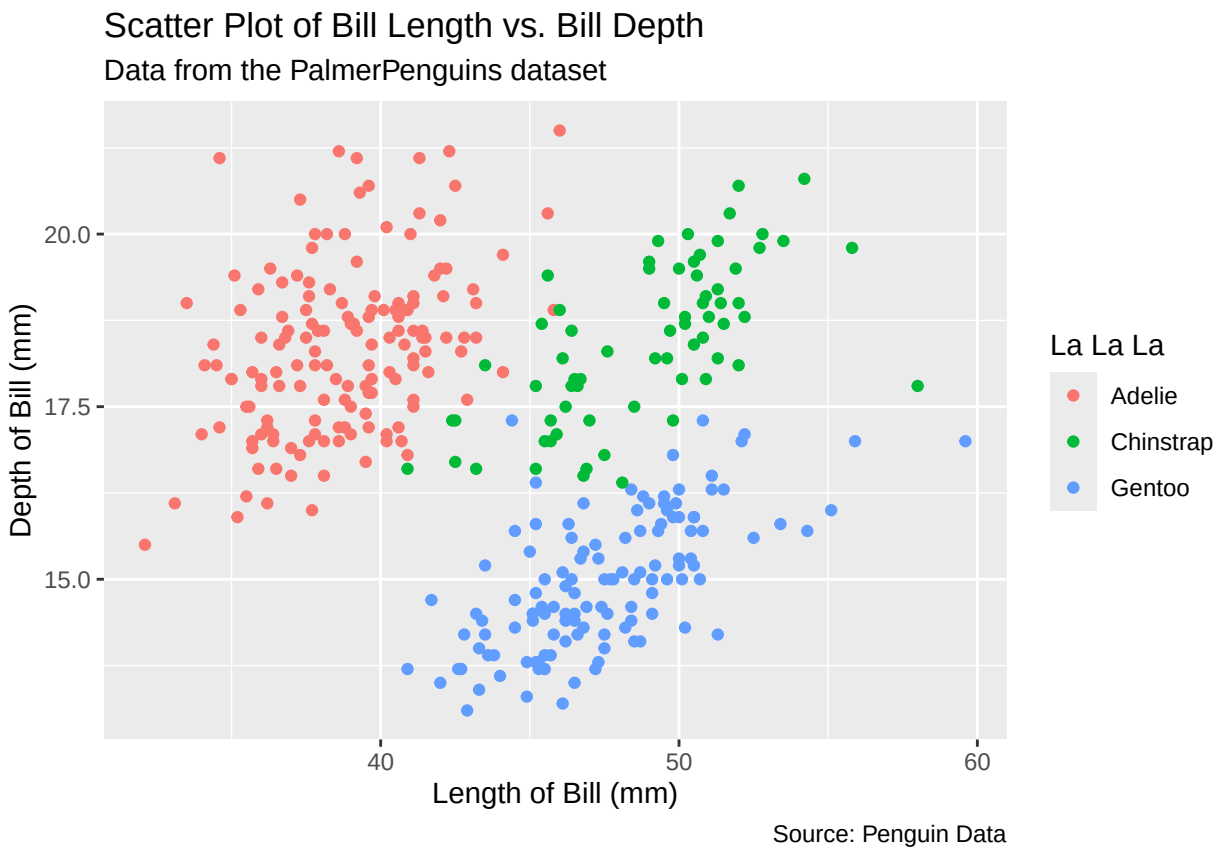
There are difference between `scale_fill_manual` and `scale_color_manual`.

Customizing Labels with labs()

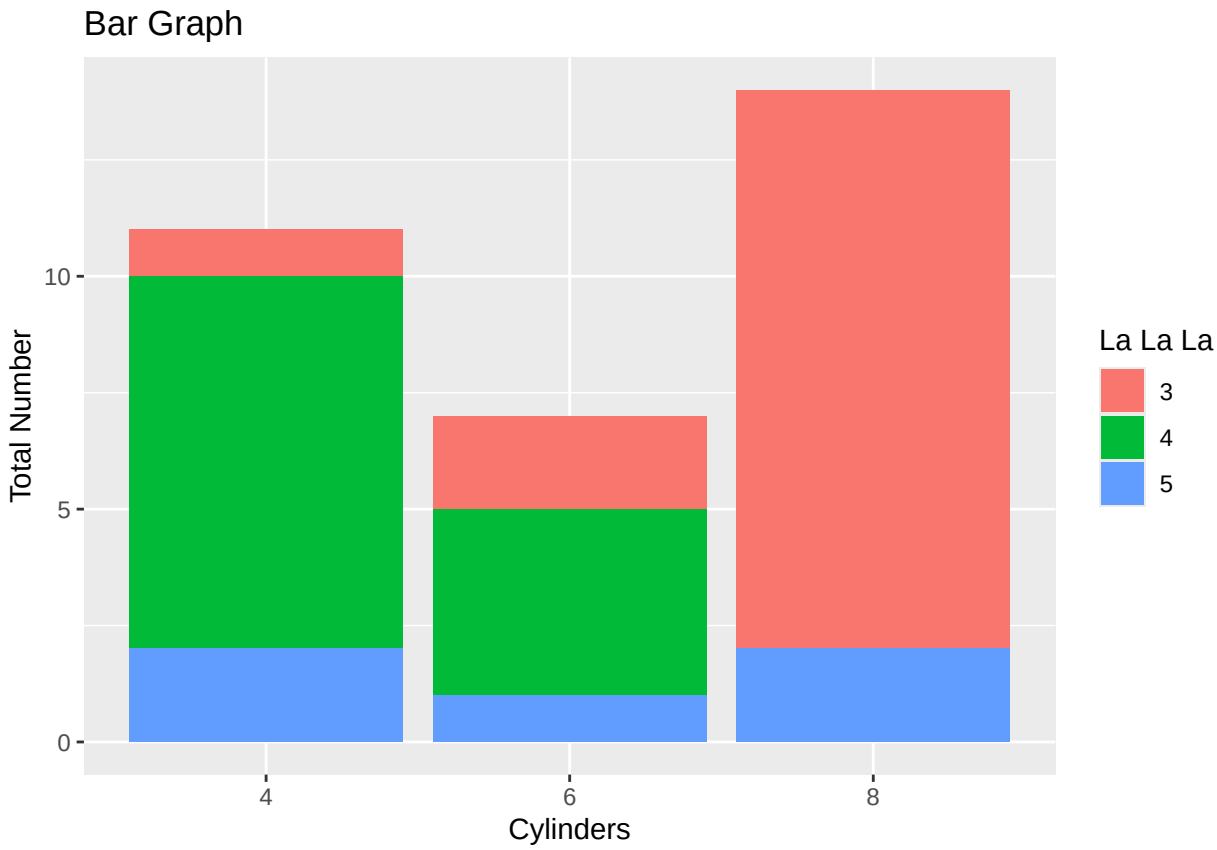
The `labs()` function in ggplot2 allows you to customize plot labels, including titles, subtitles, captions, and axis labels. You can use `labs()` to change the text displayed in various parts of your plot.

```
fig <- penguins %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm, color = species)) +
  geom_point() +
  labs(
    title = "Scatter Plot of Bill Length vs. Bill Depth",
    x = "Length of Bill (mm)",
    y = "Depth of Bill (mm)",
    subtitle = "Data from the PalmerPenguins dataset",
    caption = "Source: Penguin Data",
    color = "La La La"
  )
fig
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale range
## (`geom_point()`).
```




```
fig <- mtcars %>%
  ggplot(aes(x = factor(cyl), fill = factor(gear)))+
  geom_bar() +
  labs(
    title = "Bar Graph",
    x = "Cylinders",
    y = "Total Number",
    fill = "La La La"
  )
fig
```



Strings

str_detect()

Detects Patterns - checks if a string contains a specific pattern. Output: TRUE/FALSE.

```
my_string <- "The quick brown fox jumps over the lazy dog."
str_detect(my_string, "fox")  # Output: TRUE
```

```
## [1] TRUE
```

We can combine this function with mutate function

```
illustration_data <- data.frame(
  Example = c("the quick brown fox jumps over the lazy dog.",
              "hello, world!",
              "fox and cat")
)
example_2 <- illustration_data %>%
  mutate(fox_check = str_detect(Example, "fox"))
example_2
```

```
##                                     Example fox_check
## 1 the quick brown fox jumps over the lazy dog.      TRUE
## 2                                hello, world!     FALSE
## 3                                  fox and cat      TRUE
```

str_length()

Finds the length (the number of characters) in a string.

```
my_string <- "Hello, World!"
str_length(my_string) # Output: 13
```

```
## [1] 13
```

str_trim():

Trims white space - removes leading and trailing white space from a string.

```
my_string <- "  Trim me!   "
str_trim(my_string) # Output: "Trim me!"
```

```
## [1] "Trim me!"
```

str_squish():

Removes extra white space withing a string, as well as the beginning, and the end.

```
my_string <- "   Hello   world    from    R!    "
str_squish(my_string)  # Output: "Hello world from R!"
```

```
## [1] "Hello world from R!"
```

str_sub()

Extracts substrings from a string:

```
my_string <- "Extract this part"
str_sub(my_string, start = 9, end = 13)  # Output: "this "
```

```
## [1] "this "
```

str_replace()

Replaces a pattern with another string:

```
my_string <- "The quick brown fox jumps over the lazy dog."
str_replace(my_string, "fox", "cat")  # Output: "The quick brown cat jumps over the lazy dog."
```

```
## [1] "The quick brown cat jumps over the lazy dog."
```

str_to_lower() and str_to_upper()

Changes the case to all lower or upper case of a string.

```
my_string <- "Hello, World!"
str_to_lower(my_string)  # Output: "hello, world!"
```

```
## [1] "hello, world!"
```

```
str_to_upper(my_string)  # Output: "HELLO, WORLD!"
```

```
## [1] "HELLO, WORLD!"
```

str_c()

Combining Strings - concatenates strings together.

```r
first_name <- "John"
last_name <- "Doe"
str_c(first_name, "-", last_name) # Output: "John-Doe"
```

```
## [1] "John-Doe"
```

str_split()

Splits a string into a character vector using a specified delimiter.

```r
my_string <- "apple,banana,cherry"
str_split(my_string, ",")  # Output:  "apple"  "banana"  "cherry"
```

```
## [[1]]
## [1] "apple"  "banana" "cherry"
```

str_extract()

Extracts the first occurrence of a pattern from a string.

```r
my_string <- "Email me at john@example.com or call at 555-123-4567."
str_extract(my_string, "\\d{3}-\\d{3}-\\d{4}")  # Output: "555-123-4567"
```

```
## [1] "555-123-4567"
```

Dates

We rely on the lubridate package to do manipulations with date type variables. lubridate is a popular R package that makes it easy to work. with date-times. It provides a wide range of functions andmethods to parse, manipulate, and format date-time objects.

ymd()

`ymd()` is a function used to create a date object from year, month, and day components.

```r
my_date <- "2023-09-15"
class(my_date)
```

```
## [1] "character"
```

```r
my_date <- ymd(my_date)
class(my_date)
```

```
## [1] "Date"
```

mdy():

`mdy()` is a function used to create a date object in a month-day-year format.

```r
my_date2 <- "12-14-2023"
my_date2 <- mdy(my_date2)
my_date2
```

```
## [1] "2023-12-14"
```

ymd_hms():

`ymd_hms()` creates a date-time object with hours, minutes, and seconds.

```r
my_datetime <- "2023-09-15 14:30:45"
my_datetime <- ymd_hms(my_datetime)
```

Extratc data components

```r
my_date <- ymd("2024-10-30")
year(my_date)  # Extract year
```

```
## [1] 2024
```

```r
month(my_date)  # Extract month
```

```
## [1] 10
```

```r
day(my_date)  # Extract day
```

```
## [1] 30
```

```r
wday(my_date)  # Extract day of the week (Sunday: 1, Monday: 2, etc.)
```

```
## [1] 4
```

Example with a Dataframe

```r
dwts_dates <- data.frame(
  Name = c("Derek", "Mark", "Lindsay"),
  Birth = c("1985-05-17", "1986-05-24", "1994-01-11"),
  Wedding = c("08-26-2023", "11-25-2016", "06-18-2015"))

dwts_dates <- dwts_dates %>%
  mutate(Birth = ymd(Birth),
         Wedding = mdy(Wedding))

dwts_info <- dwts_dates %>%
  mutate(Year = year(Birth),
         Day = day(Wedding),
         Month = month(Wedding),
         Day_Week = wday(Wedding))
```

Adding Days

```r
my_date <- ymd("2024-10-30")
new_date <- my_date + days(7)  # Add 7 days to my_date
new_date
```

```
## [1] "2024-11-06"
```

Finding Time Difference

```r
my_date <- ymd("2024-10-30")
my_date2 <- ymd("2024-11-06")
diff <- my_date2 - my_date
diff
```

```
## Time difference of 7 days
```

Or, we can rely on `difftime()`

```r
time1 <- ymd_hms("2023-09-15 08:30:00")
time2 <- ymd_hms("2023-09-15 12:45:30")
difftime(time2, time1, units = "secs")
```

```
## Time difference of 15330 secs
```

```r
as.numeric(difftime(time2, time1, units = "secs"))
```

```
## [1] 15330
```

Formatting Dates

```r
my_date <- ymd("2024-10-30")

format(my_date, format = "%A") # %A - The day of the week (i.e, Friday)
```

```
## [1] "Wednesday"
```

```r
format(my_date, format = "%a") # %a - The day of the week truncated (i.e, Fri)
```

```
## [1] "Wed"
```

```r
format(my_date, format = "%B") # %B - The full name of the month (i.e September)
```

```
## [1] "October"
```

```r
format(my_date, format = "%b") # %b - The name of the month truncated (i.e., Sep)
```

```
## [1] "Oct"
```

```r
format(my_date, format = "%D") # %D - The date in month/day/year format.
```

```
## [1] "10/30/24"
```

```r
format(my_date, format = "%d") # %d - The number of the day
```

```
## [1] "30"
```

```r
format(my_date, format = "%Y") # %Y - The year with four digits
```

```
## [1] "2024"
```

```r
format(my_date, format = "%y") # %y - The year with two digits
```

```
## [1] "24"
```

Suppose we want it to be "Friday, September 15, 2023"

```r
format(my_date, format = "%A, %B %d, %Y")
```

```
## [1] "Wednesday, October 30, 2024"
```

Map Function

`Map` Functions: allow you to apply a function to each element of a list, vector, or dataframe. Map functions allow you to operate on entire vectors or data frames in a single step, without using explicit loops.

map_dbl(), map_chr(), map_dfr()

```r
illustration_data <- Teams %>%
  select(25:30) # Selecting columns 25 to 30 because they are numeric columns


map(illustration_data, median)
```

```
## $HBP
## [1] NA
##
## $SF
## [1] NA
##
## $RA
## [1] 690
##
## $ER
## [1] 597
##
## $ERA
## [1] 3.86
##
## $CG
## [1] 39
```

```r
unlist(map(illustration_data, median))
```

```
##    HBP     SF     RA     ER    ERA     CG
##     NA     NA 690.00 597.00   3.86  39.00
```

```r
example_1 <- map_dbl(illustration_data, median) # Returns it directly as a numerical vector and not a l
example_1
```

```
##    HBP     SF     RA     ER    ERA     CG
##     NA     NA 690.00 597.00   3.86  39.00
```

```r
example_2 <- map_chr(illustration_data, median) # Returns it as a character vector
```

```
## Warning: Automatic coercion from integer to character was deprecated in purrr 1.0.0.
## i Please use an explicit call to `as.character()` within `map_chr()` instead.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
example_2
```

```
##        HBP         SF         RA         ER        ERA         CG
##         NA         NA      "690"      "597" "3.860000"       "39"
```

```r
example_3 <- map_dfr(illustration_data, median) # Returns it as a dataframe
example_3
```

```
## # A tibble: 1 x 6
##     HBP    SF    RA    ER   ERA    CG
##   <int> <int> <int> <int> <dbl> <int>
## 1    NA    NA   690   597  3.86    39
```

Map Functions when you have to use additional arguments

```
map_dbl(illustration_data, median, na.rm = TRUE)
```

```
##    HBP     SF     RA     ER    ERA     CG
##  45.00  44.00 690.00 597.00   3.86  39.00
```

Map with our own functions

```
illustration_data_2 <- Teams %>%
  filter(franchID == "ANA") %>%
  group_by(teamID, name) %>%
  summarize(began = first(yearID), ended = last(yearID)) %>%
  arrange(began)
```

```
## `summarise()` has grouped output by 'teamID'. You can override using the
## `.groups` argument.
```

```
top5 <- function(team_name, data) {
  output <- data %>%
    filter(name == team_name) %>%
    select(teamID, yearID, W, L, name) %>%
    arrange(desc(W)) %>%
    head(n = 5)
  return(output)
}
map_dfr(illustration_data_2$name, top5, Teams)
```

```
##    teamID yearID   W  L                          name
## 1     LAA   1962  86 76            Los Angeles Angels
## 2     LAA   1964  82 80            Los Angeles Angels
## 3     LAA   1961  70 91            Los Angeles Angels
## 4     LAA   1963  70 91            Los Angeles Angels
## 5     CAL   1982  93 69             California Angels
## 6     CAL   1986  92 70             California Angels
## 7     CAL   1989  91 71             California Angels
## 8     CAL   1985  90 72             California Angels
## 9     CAL   1979  88 74             California Angels
## 10    ANA   2002  99 63                Anaheim Angels
## 11    ANA   2004  92 70                Anaheim Angels
## 12    ANA   1998  85 77                Anaheim Angels
## 13    ANA   1997  84 78                Anaheim Angels
## 14    ANA   2000  82 80                Anaheim Angels
## 15    LAA   2008 100 62 Los Angeles Angels of Anaheim
## 16    LAA   2014  98 64 Los Angeles Angels of Anaheim
## 17    LAA   2009  97 65 Los Angeles Angels of Anaheim
## 18    LAA   2005  95 67 Los Angeles Angels of Anaheim
## 19    LAA   2007  94 68 Los Angeles Angels of Anaheim
```

Sampling Distribution and Bootstrap

Sampling distribution

1. Population vs Sample: Population is EVERYTHING. Sample is a subset of the population.

2. What is a "statistic"? A statistic is a numerical value or measure that summarizes some aspect of a sample. (i.e., mean, median, sample standard deviation... etc.)
3. Sampling Distribution: it's a distribution of a sample statistic based on all possible simple random samples of the same size from the same population.
4. Standard Error: it's the standard deviation of a sampling distribution.

Bootstrap

The bootstrap is a resampling technique used to estimate standard errors and confidence intervals for sample statistics. It's particularly useful when you don't know the underlying data distribution.

A bootstrap sample is a sample of the same size of the original sample and sampling is done with replacement.

We can perform a simple bootstrap inference on the standard error of the sample mean in several steps:

Step 1: Start with a sample from population.

```
# We take normal_pop as our population
normal_pop <- rnorm(1000000, mean = 100, sd = 15)
initial_sample <- sample(normal_pop, 100)

# We will pretend that we only have this sample at hand.
```

Step 2: Create a function that draws one bootstrap sample from a given sample and computes a statistic on the bootstrap sample. In this example the statistic is the mean.

```
boot_mean <- function(i, y){
  boot_sample <- sample(y, length(y), replace = TRUE)
  value <- mean(boot_sample)
  return(value)
}
```

Step 3: Using the function from Step 2, draw 10000 bootstrap samples and compute the mean on each.

```
all_values <- map_dbl(1:10000, boot_mean, y = initial_sample)
```

Step 4: Compute the standard deviation of the 10000 bootstrap sample means from Step 3. That is the estimated standard error.

```
sd(all_values)
```

```
## [1] 1.464087
```

When the sample size and the number of bootstrap sample (10000) used in Step 3 converge to ∞, the standard error of a statistics can be estimated by bootstrap pretty well.