Intro to ggplot, dplyr, and pivoting

Table of contents

Much of this class will center on manipulating and visualizing data. This section gets into that.

To run much of the code in this walkthrough, you will need to load the tidyverse libraries (technically we will primarily use `ggplot`, `dplyr`, and `tidyr`, with a tiny bit of `tibble`):

```r
# tidyverse packages
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
```

```
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

Throughout, I will use the `mtcars` dataset, which is a built-in R data frame (meaning it's always available to you within any R installation; run `print(mtcars)` on your console to see what I mean) that contains automobile data extracted from the 1974 *Motor Trend* magazine. The dataset has 32 rows (each representing a different car model, with the name of that model being the row names of the data frame), and 11 columns (variables). These variables include:

- **mpg** (miles per gallon): numeric data on the fuel efficiency
- **cyl** (Number of cylinders): typically 4, 6, or 8
- **hp** (horsepower): numeric data on the engine power
- **wt** (Weight in 1000s of lbs): numeric data on the car's weight
- **disp** (Displacement in cubic inches): engine size

Why this data? Because it's an R education classic.

An introduction to ggplot

The `ggplot2` package is part of the tidyverse collection. It implements the **Grammar of Graphics**, which allows you to build plots layer by layer. The general structure is:

```r
ggplot(data = <DATA>,
       mapping = aes(x = <X-VARIABLE>,
                     y = <Y-VARIABLE>)) +
  <GEOM_FUNCTION>()
```

where things in `<...>` are general placeholders that you would have to edit to get working code. The general features are:

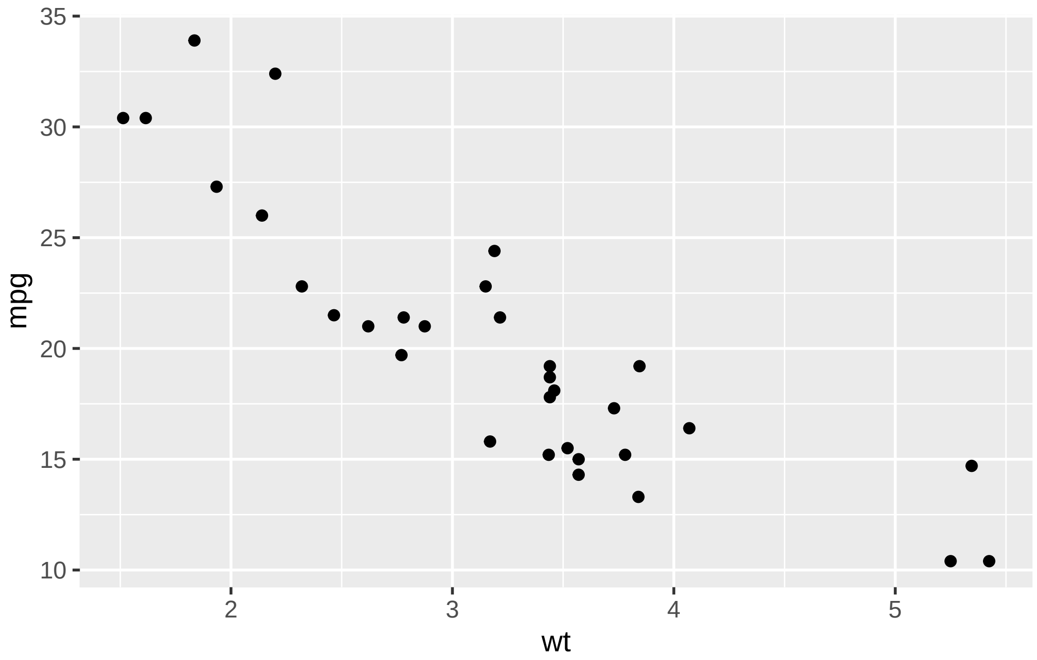
- `ggplot(data = ...)`: specifies the dataset.
- `aes()`: Which columns map to the x-axis, y-axis, color, size, etc. of the plot
- `geom_*()`: The geometry or type of plot (points, bars, lines, tiles, lines, tiles, etc.)

Making scatter plots with `geom_point()`

Scatter plots are used to visualize the relationship between two numeric variables.

Example Plot `mpg` (miles per gallon) against `wt` (weight of the car) from the `mtcars` dataset:

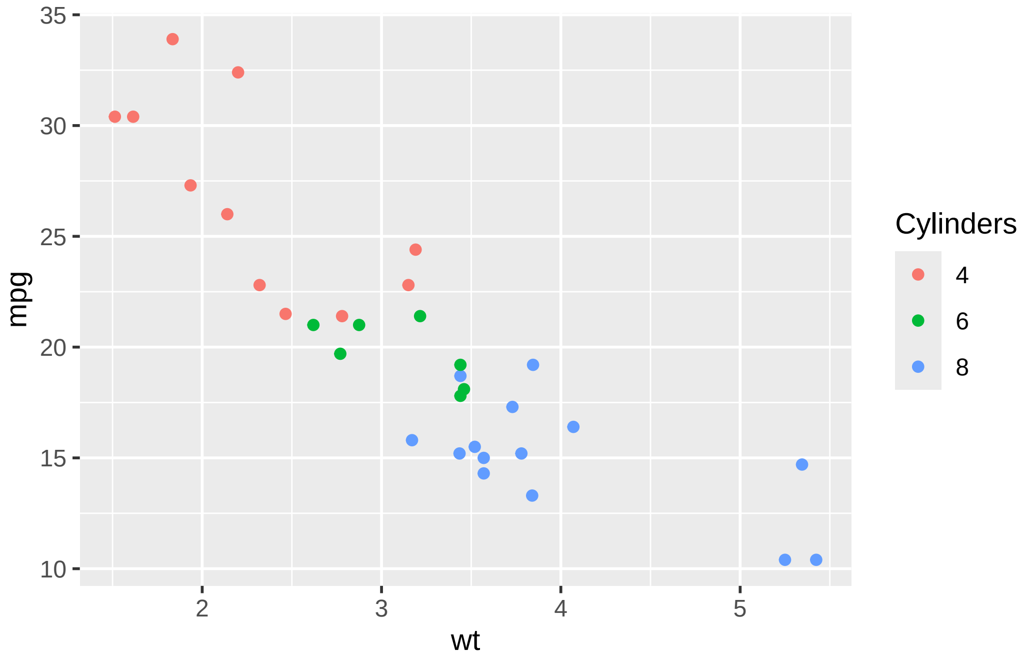
```r
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```



- We are mapping `wt` on the x-axis and `mpg` on the y-axis
- `geom_point()` draws the scatter plot by making each data points x and y a point (circular by default).

You can add more aesthetics, for example coloring the points by the number of cylinders:

```r
ggplot(data = mtcars,
       aes(x = wt, y = mpg, color = factor(cyl))) +
  geom_point() +
  labs(color = "Cylinders")
```

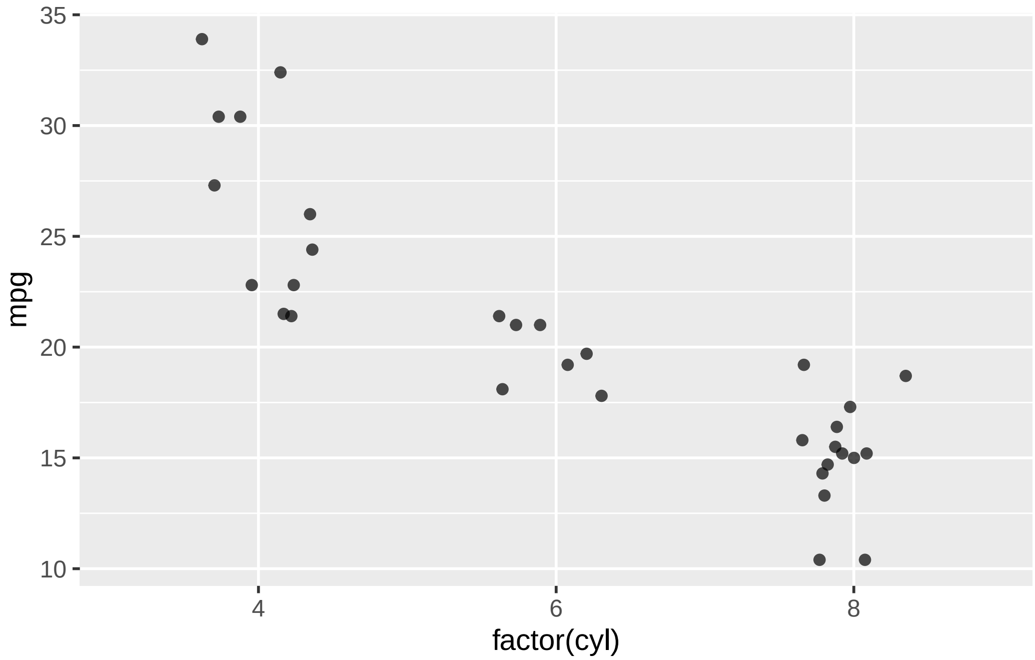


Note that `cyl` is numeric in `mtcars`, so we can convert it to a factor (`factor(cyl)`) when using it as a categorical variable (a variable with a finite set of values).

Jittered scatter plots with `geom_jitter()`

A jitter plot helps when data points overlap (i.e., have identical x or y values). It adds a small random noise to the position of each point, preventing them from lying exactly on top of each other

```
ggplot(
  data = mtcars, aes(x = factor(cyl),
                     y = mpg)
) +
  geom_jitter(
    width = 0.2,
    height = 0,
    alpha = 0.7
  )
```

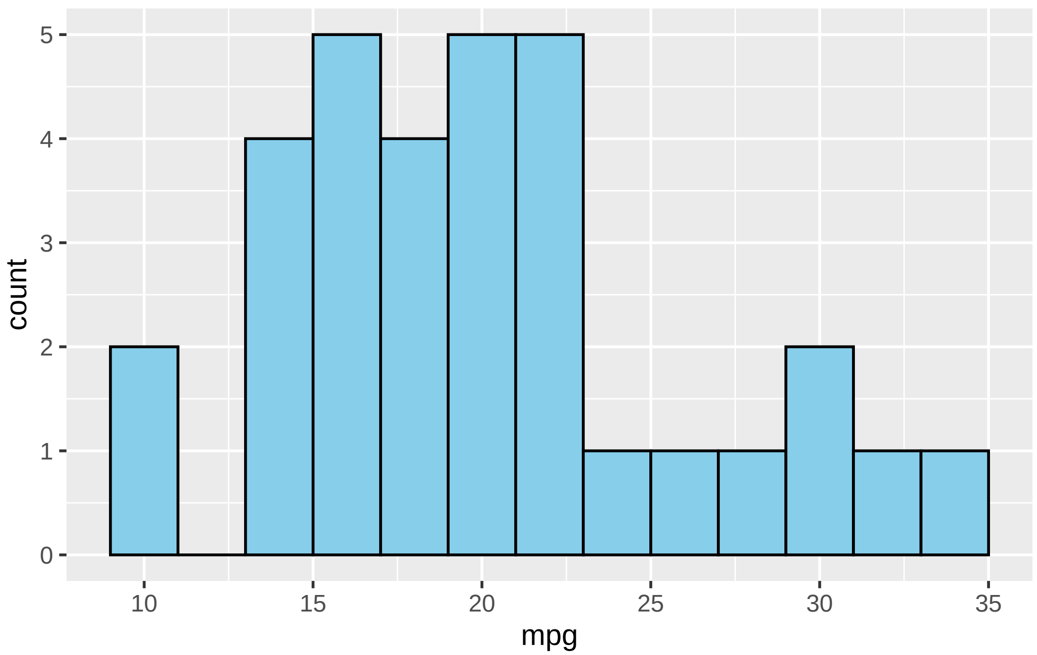


- Here, x is `factor(cyl)`, turning cylinders into discrete categories.
- We add jitter on the x-axis (using `width = 0.2`), and none on the y-axis (`height = 0`).
- `alpha = 0.7` makes the points slightly transparent to give you a sense of point density.

Histograms with `geom_histogram()`

A histogram is used to visualize the distribution of a single numeric variable. Let's look at the distribution of miles per gallon (`mpg`):

```
ggplot(data = mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 2,
                 fill = "skyblue",
                 color = "black")
```



- `binwidth = 2` sets the width of the histogram bins.
- `fill` sets the color of the bars, `color` sets the outline color.

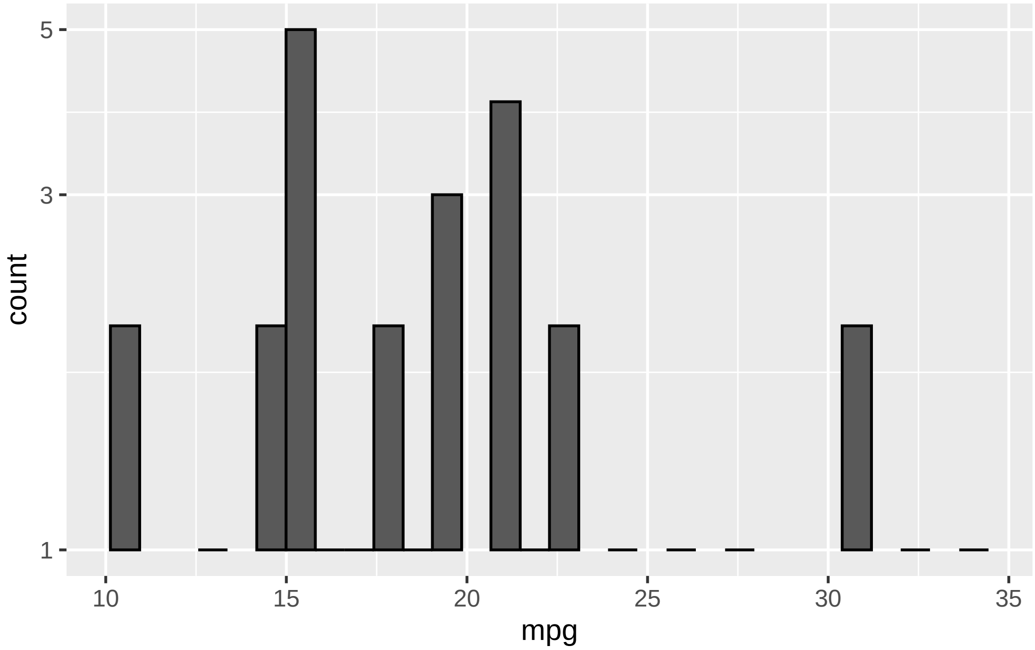
Somtimes, data can have a large range of frequencies, making it helpful to display the counts on a log scale:

```
ggplot(data = mtcars, aes(x = mpg)) +
  geom_histogram(color = 'black') +
  scale_y_log10()
```

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

```
Warning in scale_y_log10(): log-10 transformation introduced infinite values.
```

```
Warning: Removed 12 rows containing missing values or values outside the scale range
(`geom_bar()`).
```



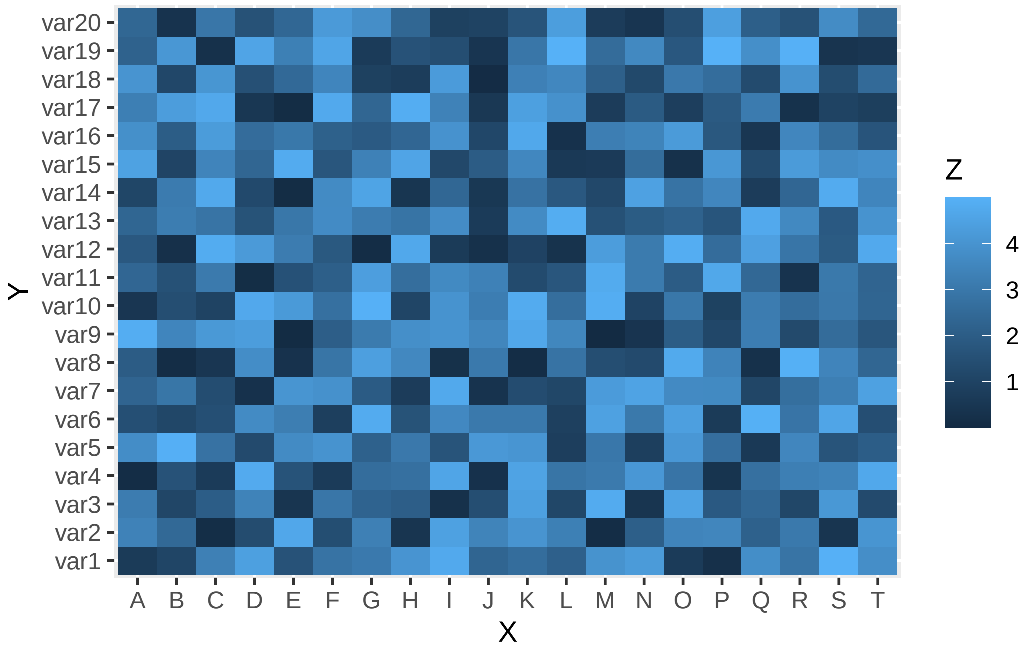
Heatmaps with `geom_tile()`

A heatmap is useful for visualizing a matrix of values or the relationship between two categorical variables, colored by a numeric value. One common example is a correlation matrix among numeric variables.

A somewhat silly example of this is below; all you need to know is that `data` ends up being a data frame with columns `X`, `Y`, and `Z`, where `X` and `Y` are categorical variables (i.e., they take on a finite set of values) and `Z` is a continuous numeric variable:

```r
# Dummy data
x <- LETTERS[1:20]
y <- paste0("var", seq(1,20))
data <- expand.grid(X=x, Y=y)
data$Z <- runif(400, 0, 5)

# Heatmap
ggplot(data, aes(X, Y, fill= Z)) +
  geom_tile()
```



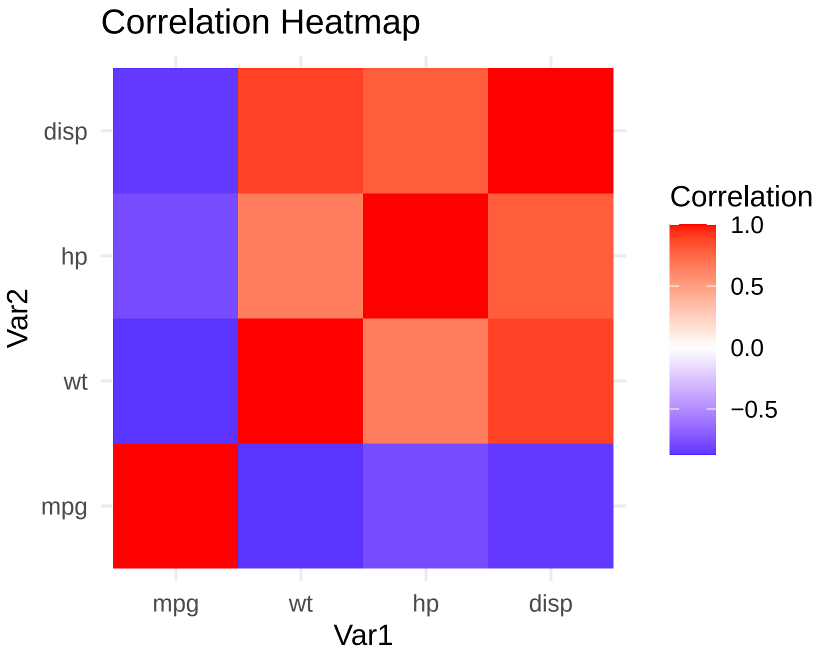
- `geom_tile()` draws the heatmap squares, one for each X and Y combo. `fill = Z` colors the tiles by the Z-value.

For a more hardcore example, let's create a correlation matrix among select columns in `mtcars` (e.g., `mpg`, `wt`, `hp`, `disp`) and plot it as a heatmap:

```
# Compute correlations
cor_mat <- cor(mtcars[, c("mpg", "wt", "hp", "disp")])

# Convert to a long format dataframe for plotting
cor_df <- as.data.frame(as.table(cor_mat))
colnames(cor_df) <- c("Var1", "Var2", "Correlation")

# Plot with geom_tile()
ggplot(data = cor_df, aes(x = Var1, y = Var2, fill = Correlation)) +
  geom_tile() +
  scale_fill_gradient2(low = "blue", mid = "white", high = "red", midpoint = 0) +
  theme_minimal() +
  coord_fixed() +
  labs(title = "Correlation Heatmap")
```



- `cor()` calculates the correlation matrix, this has rows and columns "mpg", "wt", "hp", and "disp", with entries being the correlation between of these variables and all of the other varialbes.
- `as.table()` and then `as.data.frame()` converts the matrix into a long format with columns for what was the rowname of the correlation matrix (renamed to "Var1") and what was the colname of the correlation matrix (renamed to "Var2"), as well as the correlation value between "Var1" and "Var2".
- `scale_fill_gradient2()` helps us visualize positive vs. negative correlations using a diverging color scale.
- `coord_fixed()` ensures each tile is square

ggplot themes

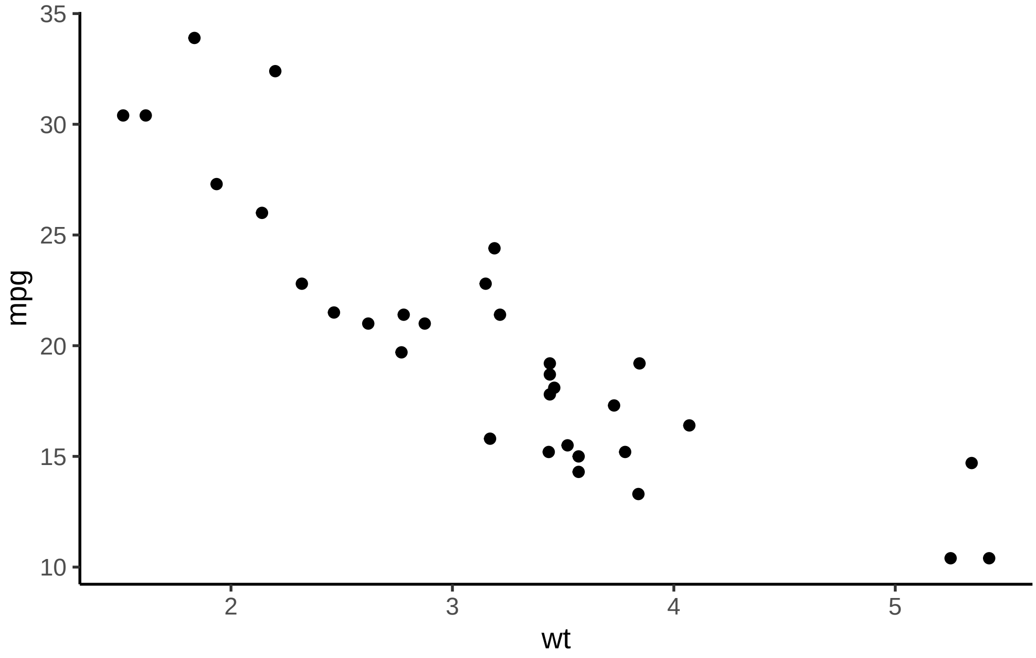
So far, we have discussed the basics of how to plot data with ggplot. One thing that might stand out about this plot is various aspects of its aesthetics:

- The gray checkerboard background
- The font sizes
- Text on axes and color legends
- etc.

To change these aspects, we can use the concept of "themes" in ggplot2. This is done through the `theme()` function.

Starting with the simple scatter plot we made earlier, we can update it's look with a number of built-in themes that ggplot provides. For example, I am a fan of the "classic" theme:

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  theme_classic()
```



Other popular themes include:

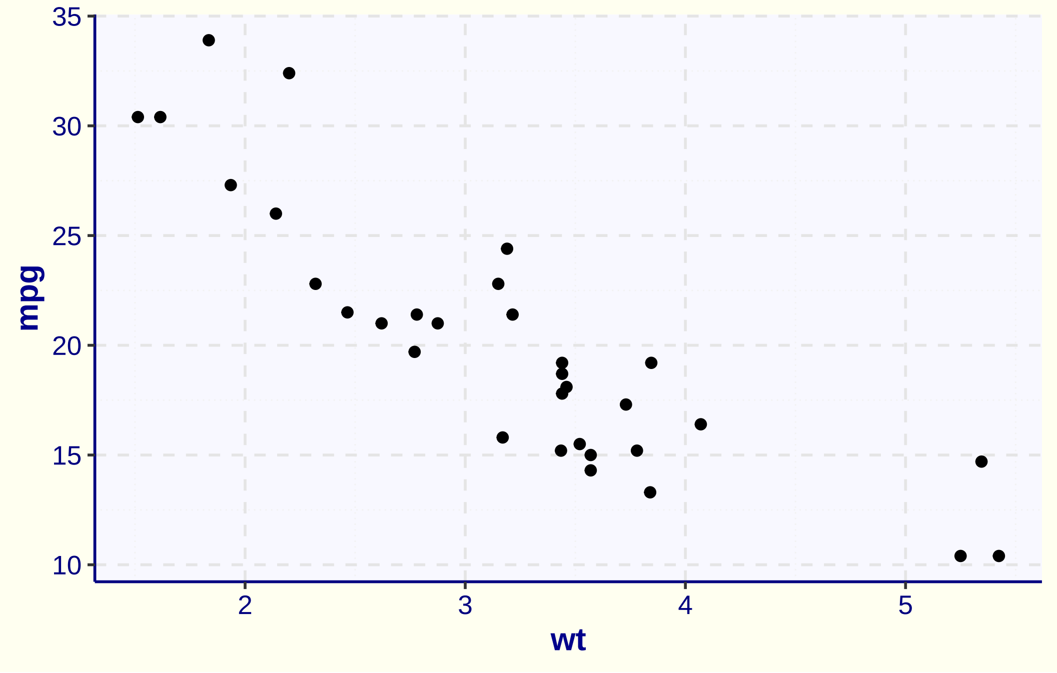
- theme_minimal()
- theme_dark()
- theme_void()

You can also customize every aspect of your plot with the use of `theme()`:

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  theme(
      # Plot background
  plot.background = element_rect(fill = "ivory", color = NA),

  # Panel (plot area) settings
  panel.background = element_rect(fill = "ghostwhite"),
  panel.grid.major = element_line(color = "gray90", linetype = "dashed"),
  panel.grid.minor = element_line(color = "gray95", linetype = "dotted"),

  # Axis customization
  axis.title = element_text(face = "bold", color = "darkblue", size = 12),
  axis.text = element_text(color = "navy", size = 10),
  axis.line = element_line(color = "navy", linewidth = 0.5),
  )
```



Check out the documentation (e.g., by running `?ggplot2::theme` or going to this link) to learn more. There is also a nice article on the topic of customizing themes here.

An introduction to dplyr and tidyr

dplyr and tidyr are both part of the tidyverse collection of packages:

- dplyr: Focuses on data manipulation and transformation. It provides a set of "verbs" that correspond to common data manipulation tasks (e.g., filter, select, mutate, summarise, arrange). These functions are often used together in a pipeline (with the `%>%` operator) to create clean, readable code that closely expresses the steps of your data processing.
- tidyr: Specializes in reshaping data between wide and long (the latter called "tidy") formats. In a tidy dataset, each variable is its own column, each observation is its own

row, and each value is in its own cell. By using functions like `pivot_longer()` and `pivot_wider()`, tidyr helps you reorganize your data so that it's consistent with these tidy data principles, facilitating analysis and visualization later on.

Combining commands with %>%

Throughout this tutorial, I will use the so-called "magrittr pipe" (`%>%`). This allows you to pass the result of one function as the first argument of the next function. It can make your code a lot cleaner and easier to read. A simple example of the pipe is:

```r
myvect <- c(1, 2, 3)

# Sum the elements of the vector
sum(myvect)
```

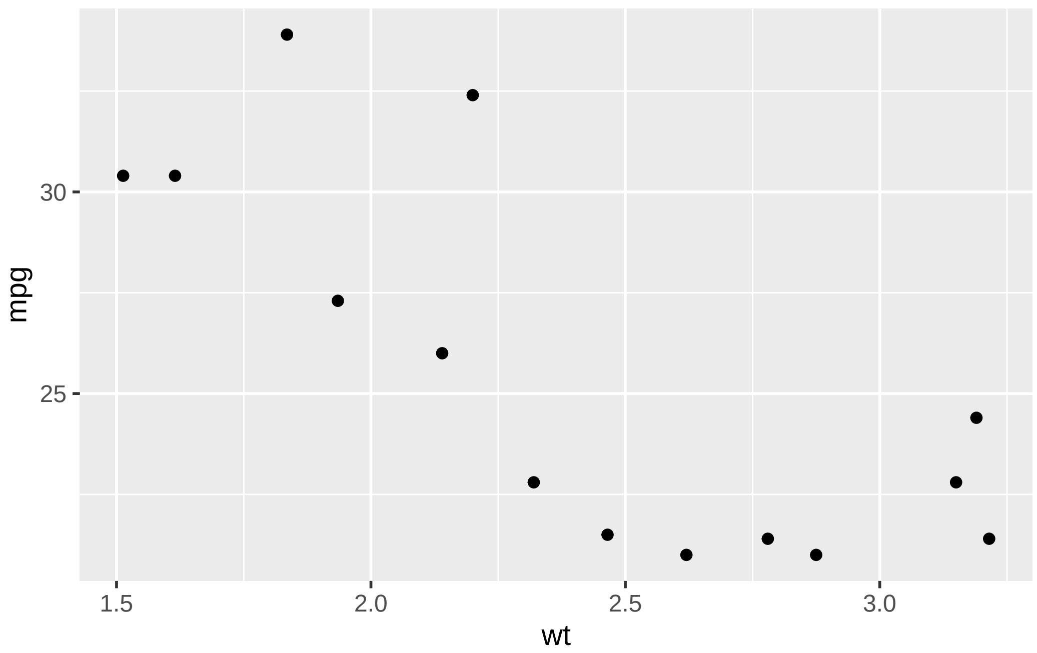
```
[1] 6
```

```r
# Same thing, but with a pipe
myvect %>% sum()
```

```
[1] 6
```

These are two ways of doing the same thing. Either you can provide `myvect` to the function `sum()` as normal, or you can pipe it in; these are equivalent. The real power of `%>%` comes from how it allows you to stitch together multiple operations:

```r
mtcars %>%
  subset(mpg > 20) %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point()
```



- First, we pipe our data into `subset()`, which is a base R function that takes a data frame as input and returns only the rows that match a certain condition (`mpg > 20` in this case).
- We are then piping the output of `subset()` into `ggplot()`, which is the equivalent of writing `ggplot(data = subset(mtcars, mpg > 20), aes(x = wt, y = mpg)) + ...`

Technically, newer versions of R (version 4.1 and later) have a base R version of the pipe, known as the native pipe (`|>`). They work pretty similarly, but I am an old hat and thus like to stick with the trusty magrittr pipe. You should feel free to use either though.

Selecting columns with `dplyr::select()`

`select()` can be used to choose (or exclude) specific columns in a data frame. The simplest usage of `select()` is to specify the columns you want to keep by name:

```
mtcars %>%
  dplyr::select(mpg, cyl, hp) %>%
  head()
```

```
                   mpg cyl  hp
Mazda RX4         21.0   6 110
Mazda RX4 Wag     21.0   6 110
```

```
Datsun 710        22.8   4  93
Hornet 4 Drive    21.4   6 110
Hornet Sportabout 18.7   8 175
Valiant           18.1   6 105
```

This returns a data frame with just those three columns. You can also exclude columns by using a `-` sign:

```
mtcars %>%
  dplyr::select(-hp) %>%
  head()
```

```
                   mpg cyl disp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 2.76 3.460 20.22  1  0    3    1
```

This returns all columns except `hp`.

If you want to get fancy, sometimes you don't know the column names beforehand – maybe they come from user input to a function. In those cases, you can store column names in a charater variable and use `!!` (the "bang-bang" operator) to unquote them (convert them from strings to as if you were typing them in yourself without the `" "`):

```
cols_to_select <- c("mpg", "wt")

mtcars %>%
  select(!!cols_to_select) %>%
  head()
```

```
                   mpg    wt
Mazda RX4         21.0 2.620
Mazda RX4 Wag     21.0 2.875
Datsun 710        22.8 2.320
Hornet 4 Drive    21.4 3.215
Hornet Sportabout 18.7 3.440
Valiant           18.1 3.460
```

Adding columns to a data frame with `dplyr::mutate()`

`mutate()` allows you to add new columns (variables) or modify existing columns.

For instance, suppose you want to create a new column named `mpg_level`, categorizing `mpg` into "high" or "low" mileage based on whether a car gets more than 20 mpg:

```
mtcars_new <- mtcars %>%
  mutate(mpg_level = ifelse(mpg > 20, "high", "low"))
```

- We take the `mtcars` data frame and pipe (`%>%`) it to `mutate()`. This means we are passing `mtcars` as the first argument of `mutate`, which needs to be a data frame.
- `ifelse()` assigns a value of "high" to rows for which `mpg > 20`, and "low" otherwise.
- The resulting new column is called `mpg_level`, and it is only present in the new data frame `mtcars_new`

Grouping and summarizing with `dplyr::summarise()` and `dplyr::group_by()`

When working with data grouped by categories, you can compute summary statistics per group.

Example: find the average miles per gallon (`mpg`) for each number of cylinders:

```
mtcars %>%
  group_by(cyl) %>%
  summarise(mean_mpg = mean(mpg), n = n())
```

```
# A tibble: 3 x 3
    cyl mean_mpg     n
  <dbl>    <dbl> <int>
1     4     26.7    11
2     6     19.7     7
3     8     15.1    14
```

- `group_by(cyl)` splits the data by the cyl variable. Whatever happens next will be done on each group separately.
- `summarise()` calculates summary statistics. Here, we calculate the average value (`mean()`) of the `mpg` column, and the number of data points (count of rows, `n()`).

Filtering data frames with `dplyr::filter()`

Use `filter()` to only keep rows that pass a certain set of conditions. For example, to select cars with more than 20 mpg:

```r
mtcars %>%
  filter(mpg > 20)
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

You can also combine filtering with grouping and only keep entire groups that pass a certain filter, for example by using `any()` and `all()`. For example, to keep only the groups of cars with the same cylinder that have at least one member of the group with more than 150 horsepower:

```r
mtcars %>%
  group_by(cyl) %>%
  filter(any(hp > 150))
```

```
# A tibble: 21 x 11
# Groups:   cyl [2]
     mpg   cyl  disp    hp  drat    wt  qsec     vs   am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
 3  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
 4  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
 5  18.1      6  225    105  2.76  3.46  20.2     1     0     3     1
 6  14.3      8  360    245  3.21  3.57  15.8     0     0     3     4
 7  19.2      6  168.   123  3.92  3.44  18.3     1     0     4     4
 8  17.8      6  168.   123  3.92  3.44  18.9     1     0     4     4
 9  16.4      8  276.   180  3.07  4.07  17.4     0     0     3     3
10  17.3      8  276.   180  3.07  3.73  17.6     0     0     3     3
# i 11 more rows
```

- `any(hp > 150)` is evaluated within each `cyl` group.
- If any car in that group has `hp > 150`, all rows of that group are kept.

By contrast, if you wanted to keep only the groups where all cars exceeded 150 horsepower:

```r
mtcars %>%
  group_by(cyl) %>%
  filter(all(hp > 150))
```

```
# A tibble: 0 x 11
# Groups:   cyl [0]
# i 11 variables: mpg <dbl>, cyl <dbl>, disp <dbl>, hp <dbl>, drat <dbl>,
#   wt <dbl>, qsec <dbl>, vs <dbl>, am <dbl>, gear <dbl>, carb <dbl>
```

- `all(hp > 150)` means every car in the group must have `hp` greater than 150.

Pivoting data longer and wider with `tidyr`

The `tidyr` package provides convenient functions to reshape data:

- `pivot_longer()`: Makes wide data longer, gathering columns into key-value pairs.
- `pivot_wider()`: Spreads long (tidy) data into wider format, creating new columns.

Let's create a simplifed data frame with a few columns:

```r
car_data <- mtcars %>%
  dplyr::select(mpg, cyl, hp, wt) %>%
  rownames_to_column(var = "car_name")

head(car_data)
```

```
           car_name  mpg cyl  hp    wt
1         Mazda RX4 21.0   6 110 2.620
2     Mazda RX4 Wag 21.0   6 110 2.875
```

```
3        Datsun 710 22.8   4  93 2.320
4    Hornet 4 Drive 21.4   6 110 3.215
5 Hornet Sportabout 18.7   8 175 3.440
6           Valiant 18.1   6 105 3.460
```

- `row_names_to_column` converts the row names of the data frame into a column. This is generally good practice. It is part of the `tibble` package, included in the `tidyverse`.
- `head()` prints the first 6 rows of a data frame to give you an easy to parse look at its contents

Suppose we want to pivot this data so that `mpg`, `hp`, and `wt` become rows under a single "measurement" column, with their values in another column:

```r
car_data_long <- car_data %>%
  pivot_longer(
    cols = c(mpg, hp, wt),
    names_to = "measurement",
    values_to = "value"
  )

head(car_data_long)
```

```
# A tibble: 6 x 4
  car_name        cyl measurement  value
  <chr>         <dbl> <chr>        <dbl>
1 Mazda RX4         6 mpg          21   
2 Mazda RX4         6 hp          110   
3 Mazda RX4         6 wt            2.62
4 Mazda RX4 Wag     6 mpg          21   
5 Mazda RX4 Wag     6 hp          110   
6 Mazda RX4 Wag     6 wt            2.88
```

- `cols` denotes the set of columns you want to "pivot"
- `names_to` is the name of the new columns that will store the names of the columns in `cols`. We are "pivoting longer", so these three columns in `cols` will no longer exist, with their content spread throughout the data frame. This new column ("measurement") will track which rows correspond to information originally contained in these columns
- `values_to` is the name of the new column that will store the values of the original columns.

We can go back to the wide format by "pivoting wider":

```r
car_data_wide <- car_data_long %>%
  pivot_wider(
    names_from = measurement,
    values_from = value
  )

head(car_data_wide)
```

```
# A tibble: 6 x 5
  car_name            cyl   mpg    hp    wt
  <chr>             <dbl> <dbl> <dbl> <dbl>
1 Mazda RX4             6  21     110  2.62
2 Mazda RX4 Wag         6  21     110  2.88
3 Datsun 710            4  22.8    93  2.32
4 Hornet 4 Drive        6  21.4   110  3.22
5 Hornet Sportabout     8  18.7   175  3.44
6 Valiant               6  18.1   105  3.46
```

- `names_from` specifies a column from which new column names will be derived.
- `values_from` will specify which column to get values that will be put int the new columns.