Data Carpentry Tutorial

Ella Crotty

Contents

Tutorial used

```r
library(tidyverse)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.1

## Warning: package 'tidyr' was built under R version 4.3.1

## Warning: package 'readr' was built under R version 4.3.1

## Warning: package 'dplyr' was built under R version 4.3.1

## Warning: package 'stringr' was built under R version 4.3.1

## Warning: package 'lubridate' was built under R version 4.3.1

## -- 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 1.9.3     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 become error
```

```
library(lubridate)
library(patchwork)
```

```
## Warning: package 'patchwork' was built under R version 4.3.1
```

```
library(nycflights13)
```

Markdown

- Don't put a space between `{r` and the chunk title
  - Also, chunks in a document must have unique titles!
- `[text] (link)` but with no space between `](`
- See header of this document for my favorite settings

Table!

Right	Left	Default	Center
12	12	12	12

dplyr

```
# Find unique rows
flights |> # another pipe operator. sure.
  distinct() # only unique rows
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # 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>
```

```r
# Find all unique origin and destination pairs
flights |>
  distinct(origin, dest, .keep_all = TRUE) # .keep_all returns full row of first
  ↪  occurence of each distinct origin/destination pair
```

```
## # A tibble: 224 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # i 214 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>
```

```r
flights |>
  count(origin, dest, sort = TRUE)
```

```
## # A tibble: 224 x 3
##    origin dest      n
##    <chr>  <chr> <int>
##  1 JFK    LAX   11262
##  2 LGA    ATL   10263
##  3 LGA    ORD    8857
##  4 JFK    SFO    8204
##  5 LGA    CLT    6168
##  6 EWR    ORD    6100
##  7 JFK    BOS    5898
##  8 LGA    MIA    5781
##  9 JFK    MCO    5464
## 10 EWR    BOS    5327
## # i 214 more rows
```

```r
flights |>
  relocate(year:dep_time, .after = time_hour) # Move year through dep_time after
  ↪  time_hour
```

```
## # A tibble: 336,776 x 19
##    sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier flight
##             <int>     <dbl>    <int>          <int>     <dbl> <chr>    <int>
##  1            515         2      830            819        11 UA        1545
##  2            529         4      850            830        20 UA        1714
##  3            540         2      923            850        33 AA        1141
##  4            545        -1     1004           1022       -18 B6         725
##  5            600        -6      812            837       -25 DL         461
##  6            558        -4      740            728        12 UA        1696
##  7            600        -5      913            854        19 B6          507
##  8            600        -3      709            723       -14 EV         5708
##  9            600        -3      838            846        -8 B6           79
## 10            600        -2      753            745         8 AA          301
## # i 336,766 more rows
## # i 12 more variables: tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>, year <int>,
## #   month <int>, day <int>, dep_time <int>
```

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 2 # Put it before current column 2
  )
```

```
## # A tibble: 336,776 x 21
##     year  gain speed month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <dbl> <dbl> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013    -9  370.     1     1      517            515         2      830
##  2  2013   -16  374.     1     1      533            529         4      850
##  3  2013   -31  408.     1     1      542            540         2      923
##  4  2013    17  517.     1     1      544            545        -1     1004
##  5  2013    19  394.     1     1      554            600        -6      812
##  6  2013   -16  288.     1     1      554            558        -4      740
##  7  2013   -24  404.     1     1      555            600        -5      913
##  8  2013    11  259.     1     1      557            600        -3      709
##  9  2013     5  405.     1     1      557            600        -3      838
## 10  2013   -10  319.     1     1      558            600        -2      753
## # i 336,766 more rows
## # i 12 more variables: sched_arr_time <int>, 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>
```

Slice

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

Formatting

```
# Reformat:
flights|>filter(carrier=="UA",dest%in%c("IAH","HOU"),sched_dep_time>
0900,sched_arr_time<2000)|>group_by(flight)|>summarize(delay=mean(
arr_delay,na.rm=TRUE),cancelled=sum(is.na(arr_delay)),n=n())|>filter(n>10)
```

```
## # A tibble: 74 x 4
##    flight delay cancelled     n
##     <int> <dbl>     <int> <int>
##  1     53  12.5         2    18
##  2    112  14.1         0    14
##  3    205 -1.71         0    14
##  4    235 -5.36         0    14
##  5    255 -9.47         0    15
##  6    268 38.6          1    15
##  7    292  6.57         0    21
##  8    318 10.7          1    20
##  9    337 20.1          2    21
## 10    370 17.5          0    11
## # i 64 more rows
```

```
# Nicer, same output
flights |>
  filter(
    carrier == "UA",dest %in% c("IAH","HOU"),
    sched_dep_time > 0900,
    sched_arr_time < 2000
    ) |>
  group_by(flight) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    cancelled = sum(is.na(arr_delay)),
    n = n()
    ) |>
  filter(n > 10)
```

```
## # A tibble: 74 x 4
##    flight delay cancelled     n
##     <int> <dbl>     <int> <int>
##  1     53 12.5          2    18
##  2    112 14.1          0    14
##  3    205 -1.71         0    14
##  4    235 -5.36         0    14
##  5    255 -9.47         0    15
##  6    268 38.6          1    15
##  7    292  6.57         0    21
##  8    318 10.7          1    20
##  9    337 20.1          2    21
## 10    370 17.5          0    11
## # i 64 more rows
```

Tidy Data * Each variable is a column; each column is a variable. * Each observation is a row; each row is an observation. * Each value is a cell; each cell is a single value.

See the DataCarpentry tutorial for more info on tidying data

```
# Tidying when there's a shitton of data stored in your column names
head(who2) # Lookit all that shit. sp is diagnosis, m is Male, 014 is 0-14 years old.
↪  Mess.
```

```
## # A tibble: 6 x 58
##   country      year sp_m_014 sp_m_1524 sp_m_2534 sp_m_3544 sp_m_4554 sp_m_5564
##   <chr>       <dbl>    <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 Afghanistan  1980       NA        NA        NA        NA        NA        NA
## 2 Afghanistan  1981       NA        NA        NA        NA        NA        NA
## 3 Afghanistan  1982       NA        NA        NA        NA        NA        NA
## 4 Afghanistan  1983       NA        NA        NA        NA        NA        NA
## 5 Afghanistan  1984       NA        NA        NA        NA        NA        NA
## 6 Afghanistan  1985       NA        NA        NA        NA        NA        NA
## # i 50 more variables: sp_m_65 <dbl>, sp_f_014 <dbl>, sp_f_1524 <dbl>,
## #   sp_f_2534 <dbl>, sp_f_3544 <dbl>, sp_f_4554 <dbl>, sp_f_5564 <dbl>,
## #   sp_f_65 <dbl>, sn_m_014 <dbl>, sn_m_1524 <dbl>, sn_m_2534 <dbl>,
## #   sn_m_3544 <dbl>, sn_m_4554 <dbl>, sn_m_5564 <dbl>, sn_m_65 <dbl>,
## #   sn_f_014 <dbl>, sn_f_1524 <dbl>, sn_f_2534 <dbl>, sn_f_3544 <dbl>,
## #   sn_f_4554 <dbl>, sn_f_5564 <dbl>, sn_f_65 <dbl>, ep_m_014 <dbl>,
## #   ep_m_1524 <dbl>, ep_m_2534 <dbl>, ep_m_3544 <dbl>, ep_m_4554 <dbl>, ...
```

```
who2 |> # WHO data that comes with tidyverse
  pivot_longer(
    cols = !(country:year), # Ignore country and year because they're already recorded
    names_to = c("diagnosis", "gender", "age"), # Slap the column names into columns
    names_sep = "_",
    values_to = "count" # Name the column that the variables will go to
  )
```

```
## # A tibble: 405,440 x 6
##    country      year diagnosis gender age   count
##    <chr>       <dbl> <chr>     <chr>  <chr> <dbl>
##  1 Afghanistan  1980 sp        m      014      NA
##  2 Afghanistan  1980 sp        m      1524     NA
##  3 Afghanistan  1980 sp        m      2534     NA
##  4 Afghanistan  1980 sp        m      3544     NA
##  5 Afghanistan  1980 sp        m      4554     NA
##  6 Afghanistan  1980 sp        m      5564     NA
##  7 Afghanistan  1980 sp        m      65       NA
##  8 Afghanistan  1980 sp        f      014      NA
##  9 Afghanistan  1980 sp        f      1524     NA
## 10 Afghanistan  1980 sp        f      2534     NA
## # i 405,430 more rows
```

Pivot_wider

```
tibble(
  x = c(1, 2, 5),
  y = c("h", "m", "g"),
  z = c(0.08, 0.83, 0.60)
)
```

```
## # A tibble: 3 x 3
##       x y         z
##   <dbl> <chr> <dbl>
## 1     1 h      0.08
## 2     2 m      0.83
## 3     5 g      0.6
```

```
# Just an example of making a dataframe by column

df <- tribble(
  ~id, ~measurement, ~value,
  "A",        "bp1",    100,
  "B",        "bp1",    140,
  "B",         "bp2",     115,
  "A",         "bp2",     120,
  "A",         "bp3",     105
)
# tribble = transposed tibble, for when you want to write by rows

head(df)
```

```
## # A tibble: 5 x 3
##   id    measurement value
##   <chr> <chr>       <dbl>
## 1 A     bp1           100
## 2 B     bp1           140
## 3 B     bp2           115
## 4 A     bp2           120
## 5 A     bp3           105
```

```
# Each ID is spread over several rows!

df |>
  pivot_wider(
    names_from = measurement,
    values_from = value
  )
```

```
## # A tibble: 2 x 4
##   id      bp1   bp2   bp3
##   <chr> <dbl> <dbl> <dbl>
## 1 A       100   120   105
## 2 B       140   115    NA
```

Clean Imports

```
students <- read_csv("https://pos.it/r4ds-students-csv")
```

```
## Rows: 6 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): Full Name, favourite.food, mealPlan, AGE
## dbl (1): Student ID
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
head(students)
```

```
## # A tibble: 6 x 5
##   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
##          <dbl> <chr>            <chr>              <chr>               <chr>
## 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
## 2            2 Barclay Lynn     French fries       Lunch only          5
## 3            3 Jayendra Lyne    N/A                Breakfast and lunch 7
## 4            4 Leon Rossini     Anchovies          Lunch only          <NA>
## 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
## 6           6 Güvenç Attila    Ice cream          Lunch only          6
```

```
head(students |> janitor::clean_names())
```

```
## # A tibble: 6 x 5
##   student_id full_name        favourite_food     meal_plan           age
##        <dbl> <chr>            <chr>              <chr>               <chr>
## 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
## 2          2 Barclay Lynn     French fries       Lunch only          5
## 3          3 Jayendra Lyne    N/A                Breakfast and lunch 7
## 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
## 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
## 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

An example of how to read in weird NA values `students <- read_csv("data/students.csv", na = c("N/A", ""))`

An example of how to read in multiple files and stack them `sales_files <- c("data/01-sales.csv", "data/02-sales.csv", "data/03-sales.csv") read_csv(sales_files, id = "file")`

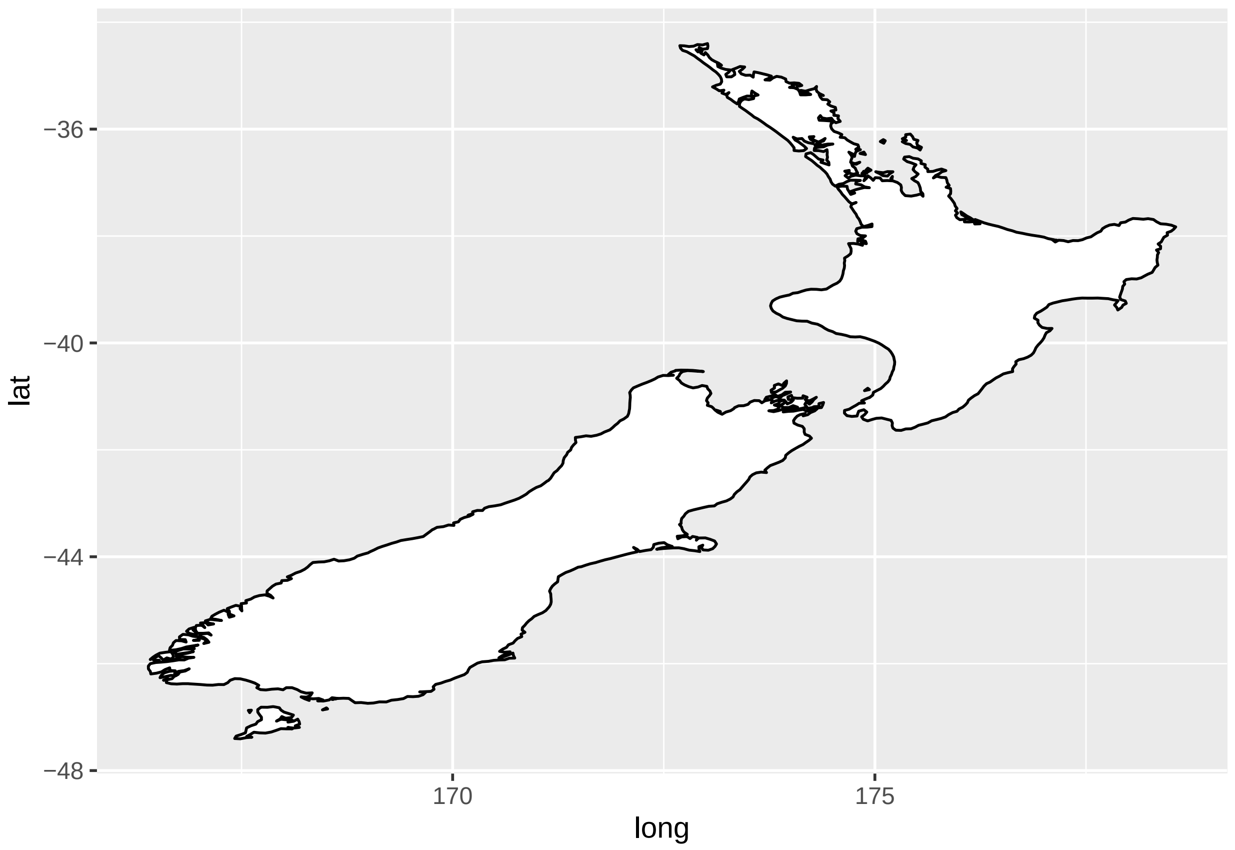
The `id` argument adds a new column called `file` to the resulting data frame that identifies the file the data come from.

Plotting

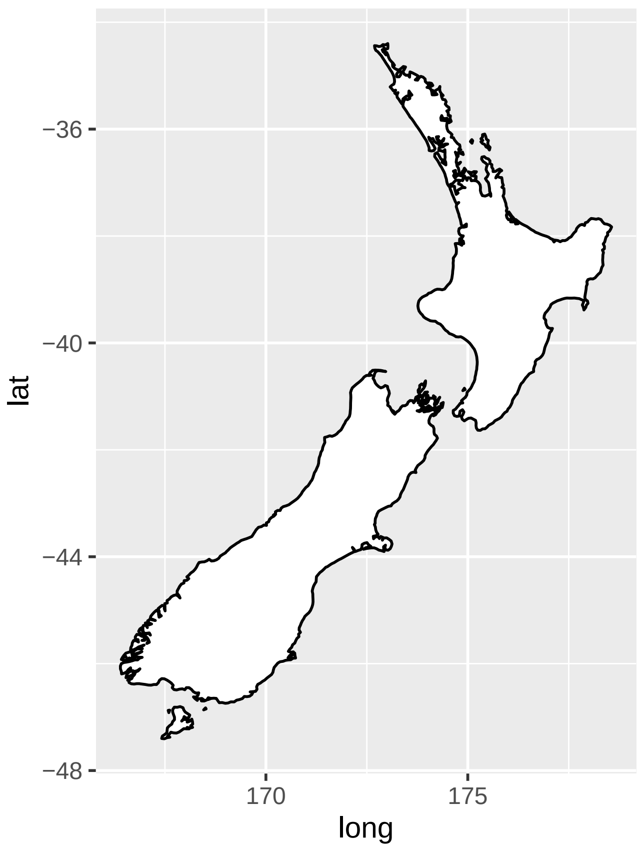
Coordinate Shenanigans

Coordinate Mapping ggplot Mapping

```
nz <- map_data("nz")

ggplot(nz, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "white", color = "black")
```



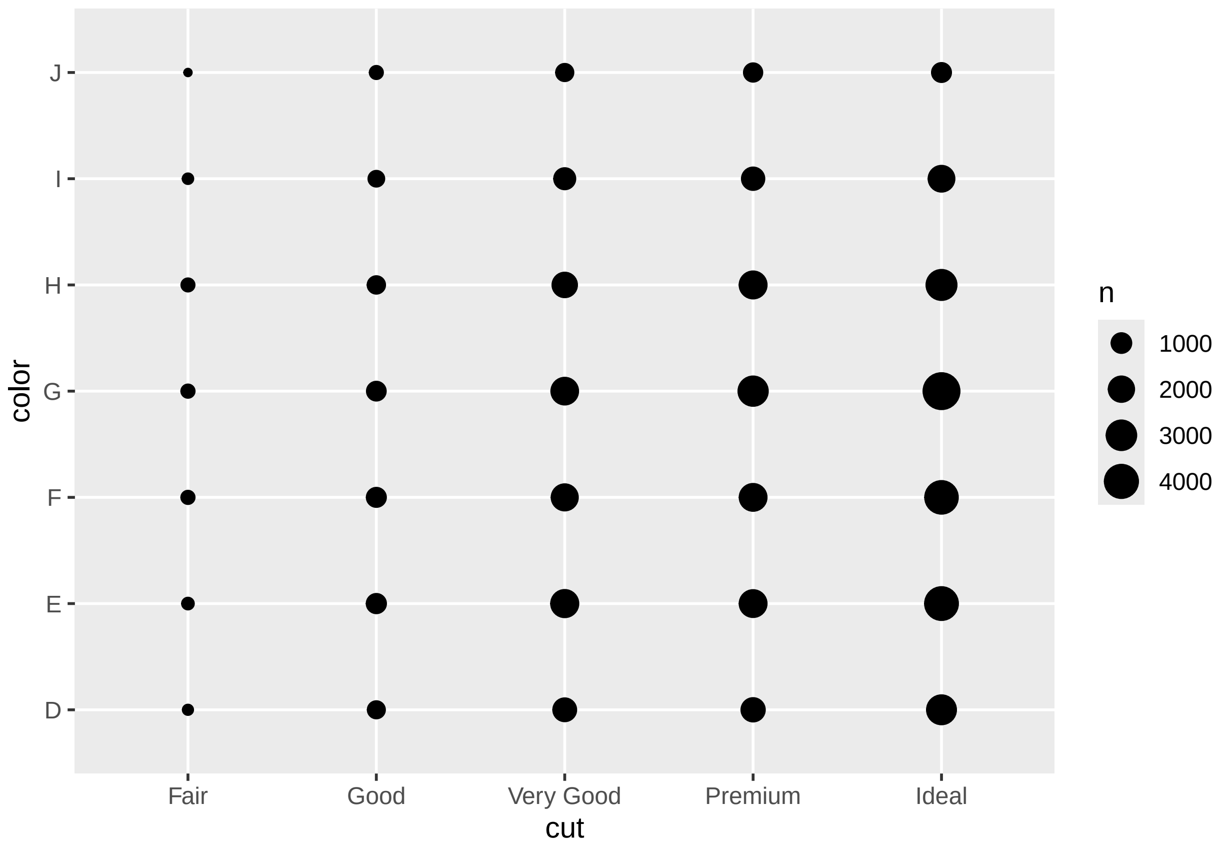
```
ggplot(nz, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "white", color = "black") +
  coord_quickmap()
```



## Exploratory Data Analysis * Remember to include the number of data points * Calculate summary statistics for the entire dataset, and subgroups as relevant.

To visualize the covariation between categorical variables, you'll need to count the number of observations for each combination of levels of these categorical variables. One way to do that is to rely on the built-in geom_count():

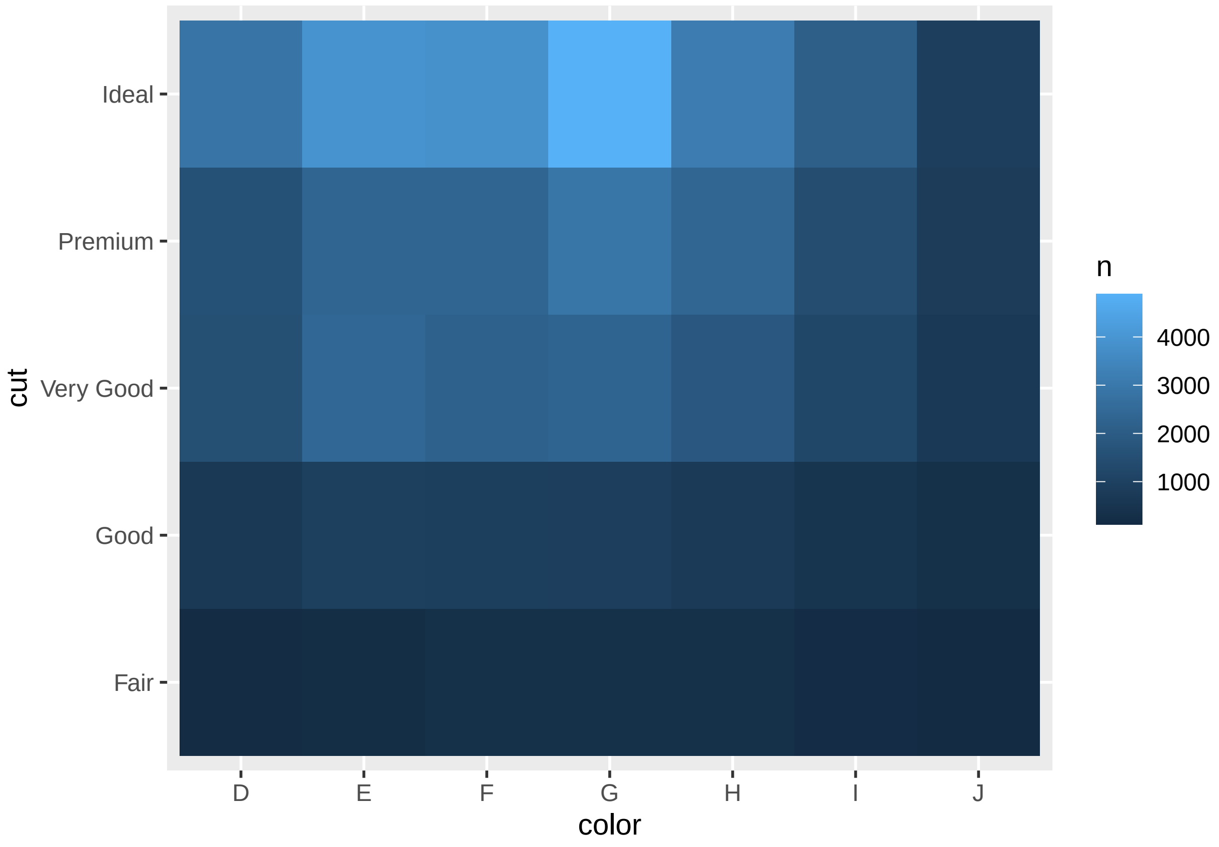
```
ggplot(diamonds, aes(x = cut, y = color)) +
  geom_count()
```



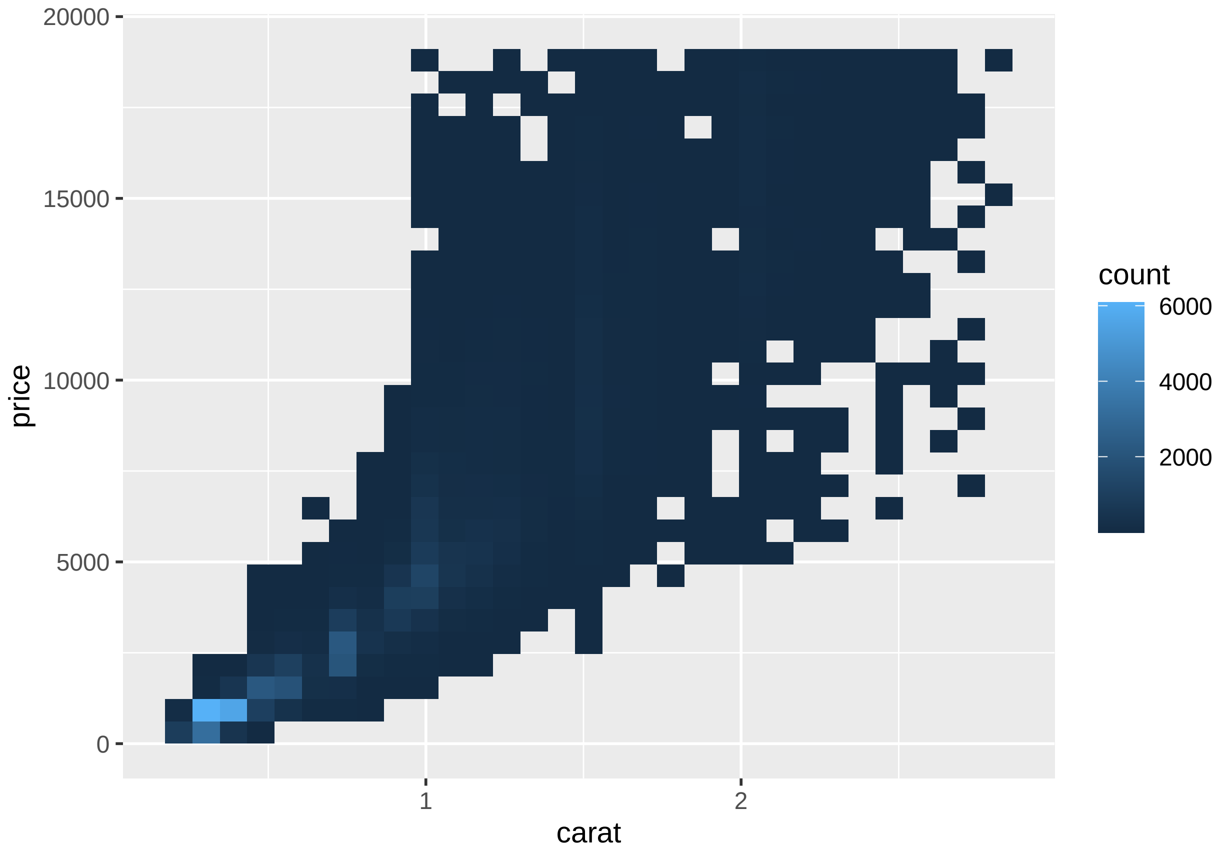
```r
diamonds |>
  count(color, cut)
```

```
## # A tibble: 35 x 3
##    color cut           n
##    <ord> <ord>     <int>
##  1 D     Fair        163
##  2 D     Good        662
##  3 D     Very Good  1513
##  4 D     Premium    1603
##  5 D     Ideal      2834
##  6 E     Fair        224
##  7 E     Good        933
##  8 E     Very Good  2400
##  9 E     Premium    2337
## 10 E     Ideal      3903
## # i 25 more rows
```

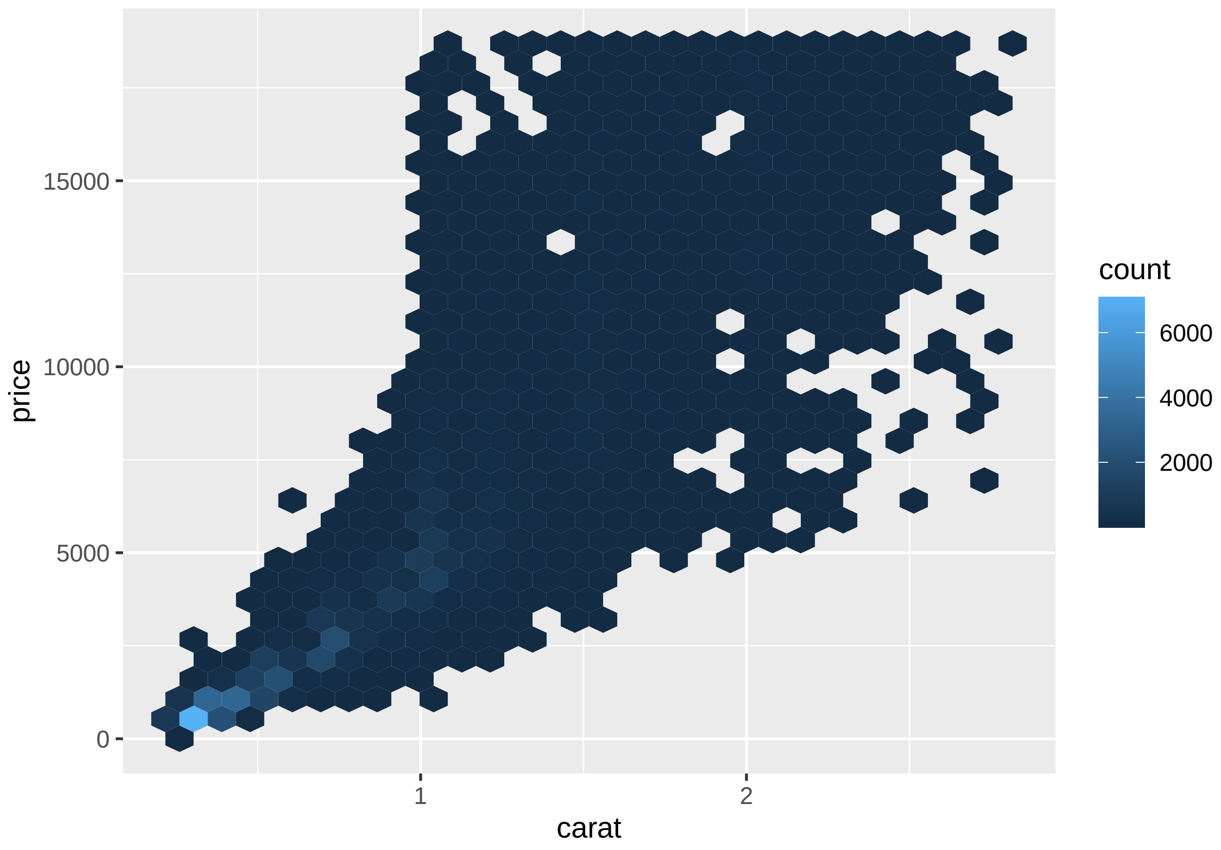
```r
diamonds |>
  count(color, cut) |>
  ggplot(aes(x = color, y = cut)) +
  geom_tile(aes(fill = n))
```



```r
smaller <- diamonds |>
  filter(carat < 3)
# Similar to hexbin, but squares
ggplot(smaller, aes(x = carat, y = price)) +
  geom_bin2d()
```



```r
library(hexbin)
ggplot(smaller, aes(x = carat, y = price)) +
  geom_hex()
```



Communication

```r
# Useful communication libraries
library(tidyverse)
library(scales)
```

```
## Warning: package 'scales' was built under R version 4.3.1
```

```
##
## Attaching package: 'scales'
```

```
## The following object is masked from 'package:purrr':
##
##     discard
```

```
## The following object is masked from 'package:readr':
##
##     col_factor
```

```r
library(ggrepel)
```

```
## Warning: package 'ggrepel' was built under R version 4.3.1
```

```r
library(patchwork)
```

labs(

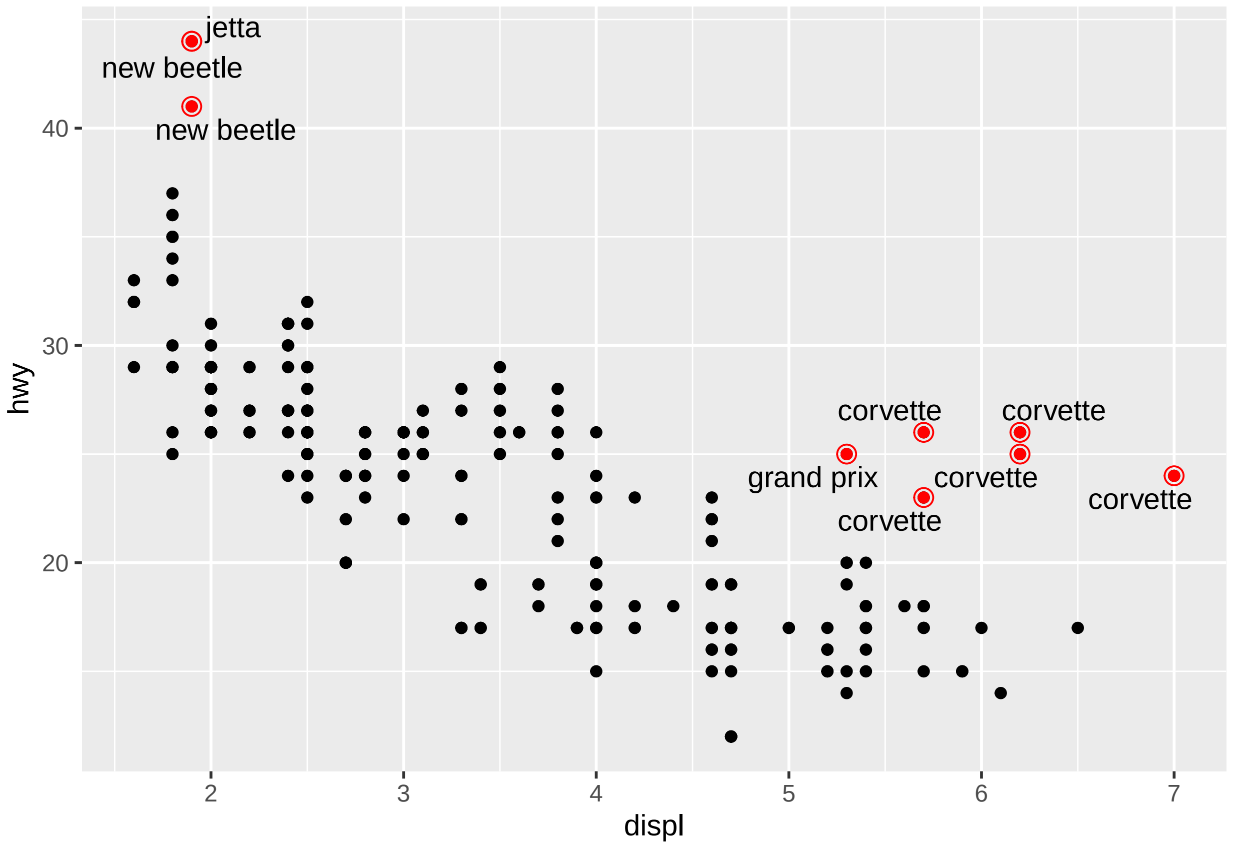
x = "X",

y = "Y",

```
color = "Legend Title",

title = "Plot Title",

subtitle = "Plot Subtitle",

caption = "Caption (defaults to lower right)"

)
```

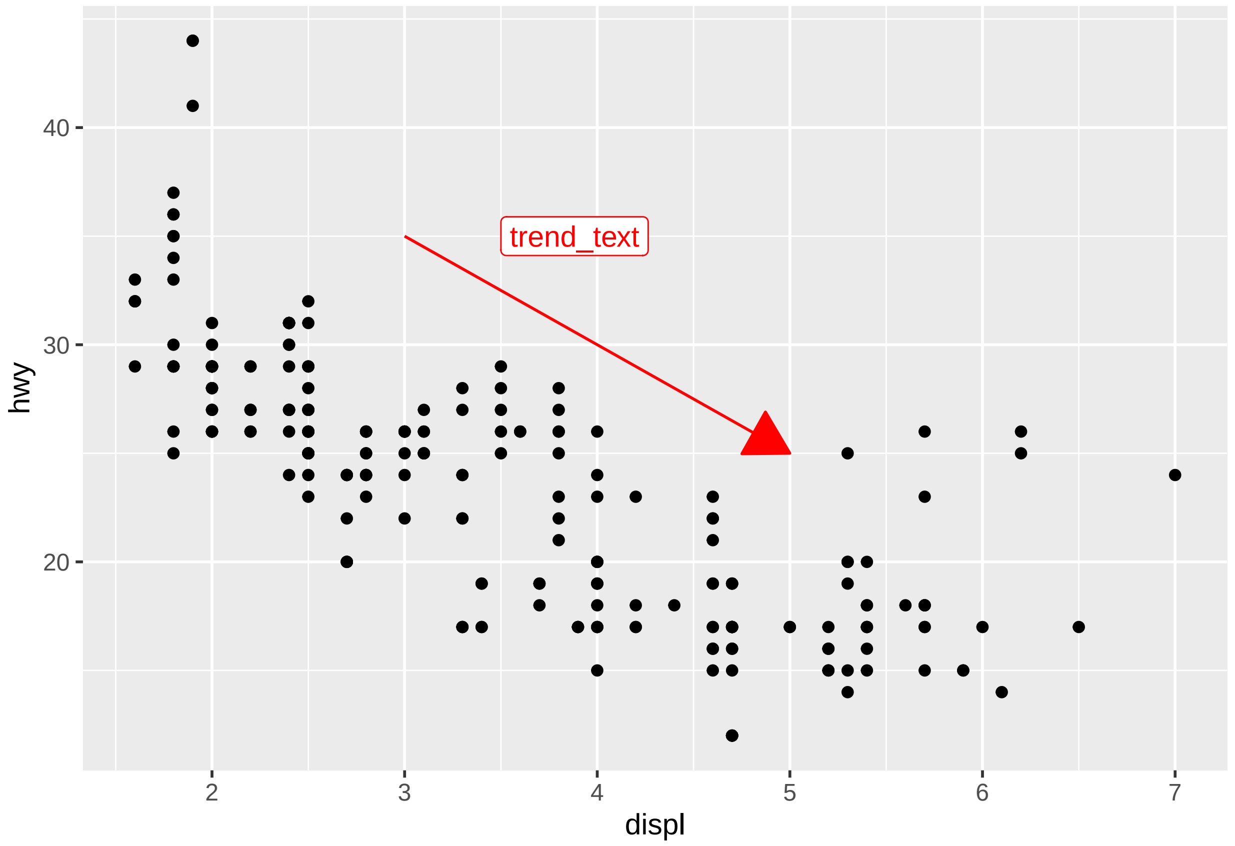
Labels can include equations - just use `quote(text)` and see `?plotmath` for options. x[i] is x subscript i.

```
potential_outliers <- mpg |>
  filter(hwy > 40 | (hwy > 20 & displ > 5))

ggplot(mpg, aes(x = displ, y = hwy)) + # Plot main data
  geom_point() +
  geom_text_repel(data = potential_outliers, aes(label = model)) + # Adds text and
  ↪  automatically tries not to overlap
  geom_point(data = potential_outliers, color = "red") + # Plot outliers again in red
  geom_point(
    data = potential_outliers,
    color = "red", size = 3, shape = "circle open"
  )
```

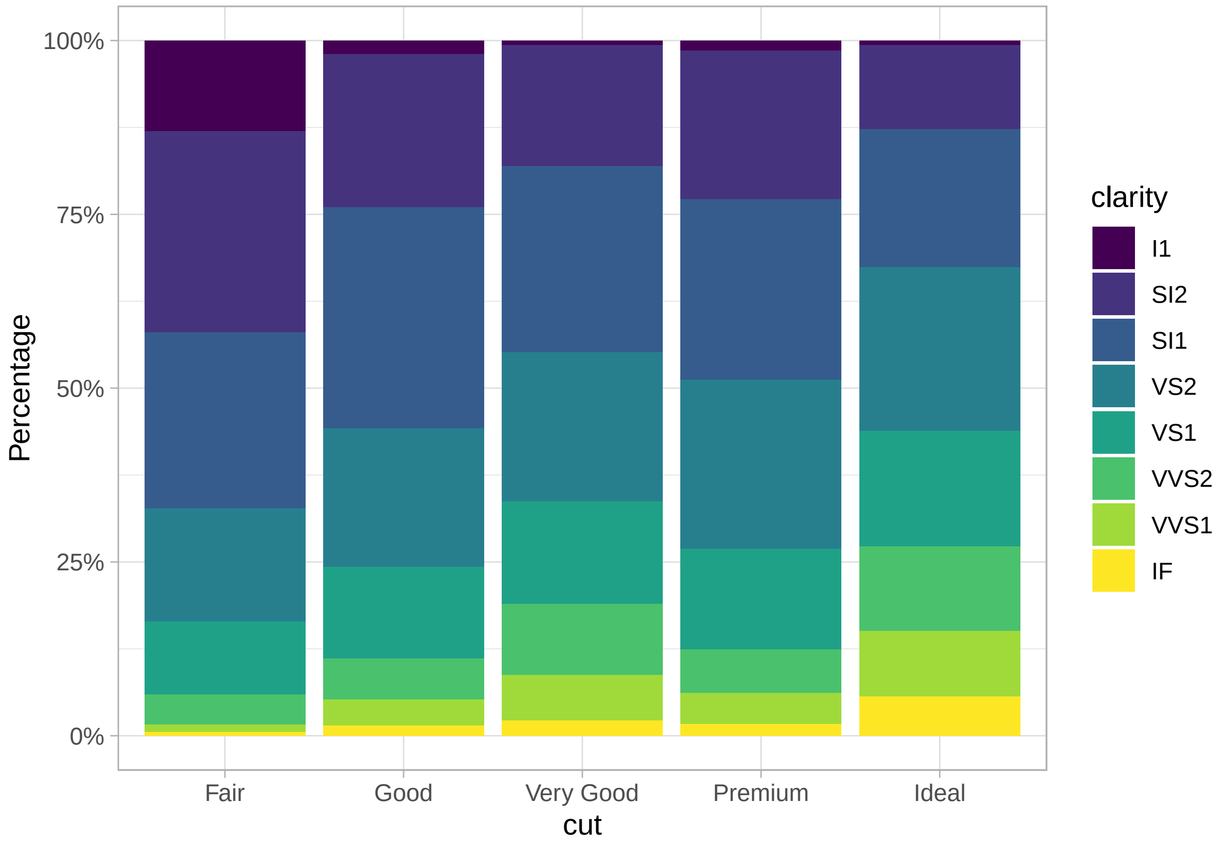


```
# Adding an arrow
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  annotate(
    geom = "label", x = 3.5, y = 35,
```

```r
    label = "trend_text",
    hjust = "left", color = "red"
  ) +
  annotate(
    geom = "segment",
    x = 3, y = 35, xend = 5, yend = 25, color = "red",
    arrow = arrow(type = "closed")
  )
```

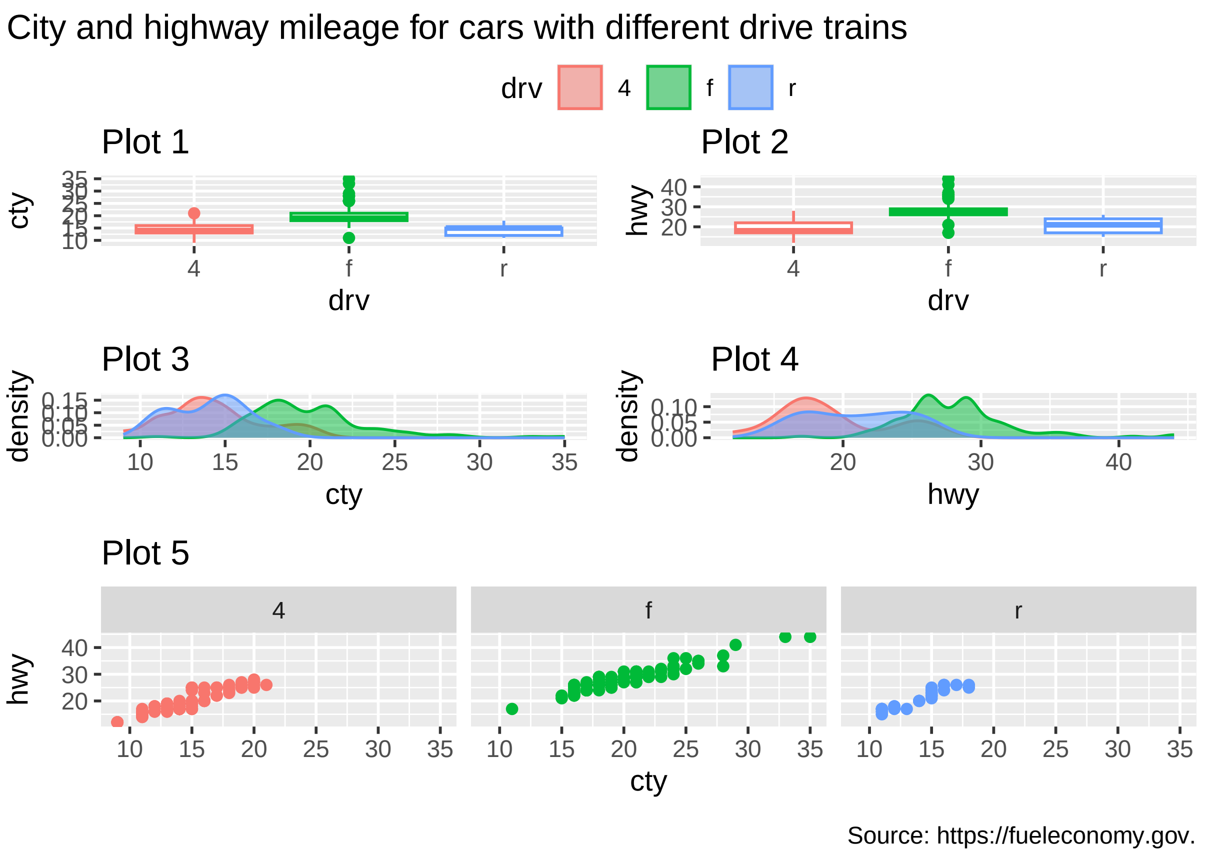


```r
# Adding percentages
ggplot(diamonds, aes(x = cut, fill = clarity)) +
  geom_bar(position = "fill") +
  scale_y_continuous(name = "Percentage", labels = label_percent()) +
  theme_light()
```



## Patchwork Tips

```r
# Make some fairly random plots, all with legends hidden
p1 <- ggplot(mpg, aes(x = drv, y = cty, color = drv)) +
  geom_boxplot(show.legend = FALSE) + # Hide legend
  labs(title = "Plot 1")

p2 <- ggplot(mpg, aes(x = drv, y = hwy, color = drv)) +
  geom_boxplot(show.legend = FALSE) +
  labs(title = "Plot 2")

p3 <- ggplot(mpg, aes(x = cty, color = drv, fill = drv)) +
  geom_density(alpha = 0.5) +
  labs(title = "Plot 3")

p4 <- ggplot(mpg, aes(x = hwy, color = drv, fill = drv)) +
  geom_density(alpha = 0.5) +
  labs(title = "Plot 4")

p5 <- ggplot(mpg, aes(x = cty, y = hwy, color = drv)) +
  geom_point(show.legend = FALSE) +
  facet_wrap(~drv) +
  labs(title = "Plot 5")

(guide_area() / (p1 + p2) / (p3 + p4) / p5) + # Open area over the plots
  plot_annotation(
    title = "City and highway mileage for cars with different drive trains",
    caption = "Source: https://fueleconomy.gov."
  ) +
  plot_layout(
    guides = "collect", # Gather all the legends
    heights = c(1, 3, 2, 4) # Customize the heights of the various components of our
    ↪  patchwork - the guide has a height of 1, the box plots 3, density plots 2, and
    ↪  the faceted scatterplot 4
    ) & # Note & instead of + operator
  theme(legend.position = "top") # Place the collected legend
```



Data Transformations

```
# Keep the underlying comparison column when using (mutate) - this is useful for
↪  troubleshooting
flights |>
  mutate(
    daytime = dep_time > 600 & dep_time < 2000,
    approx_ontime = abs(arr_delay) < 20,
    .keep = "used"
  )
```

```
## # A tibble: 336,776 x 4
##    dep_time arr_delay daytime approx_ontime
##       <int>     <dbl> <lgl>   <lgl>
##  1      517        11 FALSE   TRUE
##  2      533        20 FALSE   FALSE
##   3      542        33 FALSE   FALSE
##   4      544       -18 FALSE   TRUE
##   5      554       -25 FALSE   FALSE
##   6      554        12 FALSE   TRUE
##   7      555        19 FALSE   TRUE
##   8      557       -14 FALSE   TRUE
##   9      557        -8 FALSE   TRUE
##  10      558         8 FALSE   TRUE
## # i 336,766 more rows
```

Logical

```
x <- c(1 / 49 * 49, sqrt(2) ^ 2) # Close to 1 and 2, so get rounded when displayed
x
```

```
## [1] 1 2
```

```
print(x, digits = 16)
```

```
## [1] 0.9999999999999999 2.0000000000000004
```

```
x == c(1, 2)
```

```
## [1] FALSE FALSE
```

```
near(x, c(1, 2)) # Asks if equal + rounds
```

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

```
NA == NA # This is why is.na() is a useful function - we don't know if unknowns are equal
↪  to each other
```

```
## [1] NA
```

As well as `&` and `|`, R also has `&&` and `||`. **Don't use them in dplyr functions!** These are called short-circuiting operators and only ever return a single `TRUE` or `FALSE`. They're important for programming, not data science.

Boolean vectors can be added together using logical operators - see the rLogic.png image in this folder for more.

```
# INCORRECT
flights |>
  filter(month == 11 | 12)
```

```
## # A tibble: 336,776 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##     <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##   1  2013     1     1      517            515         2      830            819
##   2  2013     1     1      533            529         4      850            830
##   3  2013     1     1      542            540         2      923            850
##   4  2013     1     1      544            545        -1     1004           1022
##   5  2013     1     1      554            600        -6      812            837
##   6  2013     1     1      554            558        -4      740            728
##   7  2013     1     1      555            600        -5      913            854
##   8  2013     1     1      557            600        -3      709            723
##   9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # 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>
```

```
# CORRECT
flights |>
   filter(month == 11 | month == 12)
```

```
## # A tibble: 55,403 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013    11     1        5           2359         6      352            345
##  2  2013    11     1       35           2250       105      123           2356
##  3  2013    11     1      455            500        -5      641            651
##  4  2013    11     1      539            545        -6      856            827
##  5  2013    11     1      542            545        -3      831            855
##  6  2013    11     1      549            600       -11      912            923
##  7  2013    11     1      550            600       -10      705            659
##  8  2013    11     1      554            600        -6      659            701
##  9  2013    11     1      554            600        -6      826            827
## 10  2013    11     1      554            600        -6      749            751
## # i 55,393 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>
```

```
flights |>
  filter(month %in% c(11, 12))
```

```
## # A tibble: 55,403 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013    11     1        5           2359         6      352            345
##  2  2013    11     1       35           2250       105      123           2356
##  3  2013    11     1      455            500        -5      641            651
##  4  2013    11     1      539            545        -6      856            827
##  5  2013    11     1      542            545        -3      831            855
##  6  2013    11     1      549            600       -11      912            923
##  7  2013    11     1      550            600       -10      705            659
##  8  2013    11     1      554            600        -6      659            701
##  9  2013    11     1      554            600        -6      826            827
## 10  2013    11     1      554            600        -6      749            751
## # i 55,393 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>
```

```
NA %in% NA # Is true, can be used
```

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

- `any(x)` is the equivalent of `|`; it'll return `TRUE` if there are any `TRUE`'s in `x`. `all(x)` is equivalent of `&`; it'll return `TRUE` only if all values of `x` are `TRUE`'s. Like all summary functions, they'll return `NA` if there are any missing values present, and as usual you can make the missing values go away with `na.rm =`

TRUE.

- When you use a logical vector in a numeric context, TRUE becomes 1 and FALSE becomes 0. sum(x) gives the number of TRUEs and mean(x) gives the proportion of TRUEs (because mean() is just sum() divided by length().
- if_else(condition, if_true_output, if_false_output, NA_val) - used to transform vectors/data columns
- If you are nesting if_else() statements, it's probably time to use case_when(condition ~ output). When condition (a logical vector) is true, output will be used. If multiple conditions match, the first will be used. The parameter .default = "__" sets the output if none of the conditions match.
- if_else() and case_when() need compatible types
  - Numeric and logical vectors are compatible, as we discussed in Section 12.4.2.
  - Strings and factors (Chapter 16) are compatible, because you can think of a factor as a string with a restricted set of values.
  - Dates and date-times, which we'll discuss in Chapter 17, are compatible because you can think of a date as a special case of date-time.
  - NA, which is technically a logical vector, is compatible with everything because every vector has some way of representing a missing value.
- abs() gives absolute value, this may be useful for logical comparisons

Numerical

- parse_double() can turn "1.6" into 1.6
- parse_number() can turn "$1.60" into 1.6
- n_distinct(x) counts the number of distinct (unique) values of one or more variables.
- When you do arithmetic, R handles mismatched lengths by recycling, or repeating, the short vector. This is cool and normal for c(1,2,3)*5, but probably not what you're looking for in c(1,2,3)/c(1,2). Same with comparisons - make sure not to use == when you mean %in%!

```
# pmin() and pmax() return the extreme value in each ROW
df <- tribble(
  ~x, ~y,
  1,  3,
  5,  2,
  7, NA,
)

df |>
  mutate(
    min = pmin(x, y, na.rm = TRUE),
    max = pmax(x, y, na.rm = TRUE)
  )
```

```
## # A tibble: 3 x 4
##       x     y   min   max
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     3     1     3
## 2     5     2     2     5
## 3     7    NA     7     7
```

```
# vs normal min/max
df |>
  mutate(
    min = min(x, y, na.rm = TRUE),
    max = max(x, y, na.rm = TRUE)
  )
```

```
## # A tibble: 3 x 4
##       x     y   min   max
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     3     1     7
## 2     5     2     1     7
## 3     7    NA     1     7
```

- Modular arithmetic: In R, `%/%` does integer division and `%%` computes the remainder. One super cursed use for this is to separate milatary time into hours and minutes.

```
flights |>
  mutate(
    hour = sched_dep_time %/% 100,
    minute = sched_dep_time %% 100,
    .keep = "used"
  )
```

```
## # A tibble: 336,776 x 3
##    sched_dep_time  hour minute
##             <int> <dbl>  <dbl>
##  1            515     5     15
##  2            529     5     29
##  3            540     5     40
##  4            545     5     45
##  5            600     6      0
##  6            558     5     58
##  7            600     6      0
##  8            600     6      0
##  9            600     6      0
## 10            600     6      0
## # i 336,766 more rows
```

- In R, you have a choice of three logarithms: `log()` (the natural log, base e), `log2()` (base 2), and `log10()` (base 10).
- `round(123.456, 2)` rounds 123.456 to 2 digits.
- `round(x)` rounds to the nearest integer. 0.5s are rounded to the nearest **even** integer.
- `floor(x)` rounds down
- `ceiling(x)` rounds up

```
x <- 123.456
# Round to nearest 0.25
round(x / 0.25) * 0.25
```

```
## [1] 123.5
```

```
# Cut
x <- c(1, 2, 5, 10, 15, 20)
cut(x, breaks = c(0, 5, 10, 15, 20))
```

```
## [1] (0,5]   (0,5]   (0,5]   (5,10]  (10,15] (15,20]
## Levels: (0,5] (5,10] (10,15] (15,20]
```

```r
cut(x,
  breaks = c(0, 5, 10, 15, 20),
  labels = c("sm", "md", "lg", "xl")
)
```

```
## [1] sm sm sm md lg xl
## Levels: sm md lg xl
```

- Base R provides `cumsum()`, `cumprod()`, `cummin()`, `cummax()` for running, or cumulative, sums, products, mins and maxes. dplyr provides `cummean()` for cumulative means.
- `dplyr` can rank numbers in order
  - If `min_rank()` doesn't do what you need, look at the variants `dplyr::row_number()`, `dplyr::dense_rank()`, `dplyr::percent_rank()`, and `dplyr::cume_dist()`. See the documentation for details.

```r
df_rank <- tibble(x = c(1, 3, 3, 4, 7, NA))
df_rank %>%
  mutate(ranked = min_rank(x),
         rev_ranked = min_rank(desc(x)))
```

```
## # A tibble: 6 x 3
##       x ranked rev_ranked
##   <dbl>  <int>      <int>
## 1     1      1          5
## 2     3      2          3
## 3     3      2          3
## 4     4      4          2
## 5     7      5          1
## 6    NA     NA         NA
```

```r
# Note the double 2 and no 3 for dealing with a tie!
```

```r
# Dividing data into similar-sized chunks by new column value
df <- tibble(id = 1:10)

df |>
  mutate(
    row0 = row_number() - 1,
    three_groups = row0 %% 3,
    three_in_each_group = row0 %/% 3
  )
```

```
## # A tibble: 10 x 4
##       id  row0 three_groups three_in_each_group
##    <int> <dbl>        <dbl>               <dbl>
##  1     1     0            0                   0
##  2     2     1            1                   0
##  3     3     2            2                   0
##  4     4     3            0                   1
##  5     5     4            1                   1
##  6     6     5            2                   1
##  7     7     6            0                   2
##  8     8     7            1                   2
##  9     9     8            2                   2
## 10    10     9            0                   3
```

For even fancier slicing, check out this section

```
x <- c(2, 5, 11, 11, 19, 35)
lag(x)
```

```
## [1] NA  2  5 11 11 19
```

```
#> [1] NA  2  5 11 11 19
lag(x, n = 2) # Lag by more
```

```
## [1] NA NA  2  5 11 11
```

```
#> [1] NA NA  2  5 11 11
lead(x)
```

```
## [1]  5 11 11 19 35 NA
```

```
#> [1]  5 11 11 19 35 NA
```

- `x - lag(x)` gives you the difference between the current and previous value.

```
x - lag(x)
```

```
## [1] NA  3  6  0  8 16
```

```
#> [1] NA  3  6  0  8 16
```

- `x == lag(x)` tells you when the current value changes.

```
x == lag(x)
```

```
## [1]    NA FALSE FALSE  TRUE FALSE FALSE
```

```
#> [1]    NA FALSE FALSE  TRUE FALSE FALSE
```

We can `group_by()` and then find a specific ranking

```
flights |>
  group_by(year, month, day) |>
  summarize(
    first_dep = first(dep_time, na_rm = TRUE), # na_rm is a dplyr thing
    fifth_dep = nth(dep_time, 5, na_rm = TRUE),
    last_dep = last(dep_time, na_rm = TRUE)
  )
```

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

```
## # A tibble: 365 x 6
## # Groups:   year, month [12]
##     year month   day first_dep fifth_dep last_dep
##    <int> <int> <int>     <int>     <int>    <int>
##  1  2013     1     1       517       554     2356
##  2  2013     1     2        42       535     2354
##  3  2013     1     3        32       520     2349
##  4  2013     1     4        25       531     2358
##  5  2013     1     5        14       534     2357
##  6  2013     1     6        16       555     2355
##  7  2013     1     7        49       536     2359
##  8  2013     1     8       454       544     2351
##  9  2013     1     9         2       524     2252
## 10  2013     1    10         3       530     2320
## # i 355 more rows
```

Some other uses of these numeric functions: * x / sum(x) calculates the proportion of a total. * (x - mean(x)) / sd(x) computes a Z-score (standardized to mean 0 and sd 1). * (x - min(x)) / (max(x) - min(x)) standardizes to range [0, 1]. * x / first(x) computes an index based on the first observation.

Strings

```r
library(babynames) # Get a bunch of strings
# stringr is part of tidyverse, all the functions start with str_
```

- \ in front of special characters like quotes lets you put them in a string
- If you are using too many backslashes and shit gets unreadable, use raw strings. A raw string usually starts with `r"( and finishes with )"`. But if your string contains `)"` you can instead use `r"[]"` or `r"{}"`, and if that's still not enough, you can insert any number of dashes to make the opening and closing pairs unique, e.g., `r"--()--"`, `r"---()---"`, etc. Raw strings are flexible enough to handle any text.
- As well as `\"`, `\'`, and `\\`, there are a handful of other special characters that may come in handy. The most common are `\n`, a new line, and `\t`, tab.

```r
# Special characters!
# ?Quotes # List of special characters
# Unicode supported, but not for knitting with LaTeX LOL
# x <- c("one\ntwo", "one\ttwo", "\u00b5", "\U0001f604")
x
```

```
## [1]  2  5 11 11 19 35
```

```r
str_view(x)
```

```
## [1] | 2
## [2] | 5
## [3] | 11
## [4] | 11
## [5] | 19
## [6] | 35
```

```r
str_c("Hello ", c("John", "Susan")) # Similar to paste() but plays nicer with tidyverse
```

```
## [1] "Hello John"  "Hello Susan"
```

```r
df <- tibble(name = c("Flora", "David", "Terra", NA))
df |> mutate(greeting = str_c("Hi ", name, "!"))
```

```
## # A tibble: 4 x 2
##   name  greeting
##   <chr> <chr>
## 1 Flora Hi Flora!
## 2 David Hi David!
## 3 Terra Hi Terra!
## 4 <NA>  <NA>
```

```r
# Deal with missing values
df |>
  mutate(
    greeting1 = str_c("Hi ", coalesce(name, "you"), "!"),
    greeting2 = coalesce(str_c("Hi ", name, "!"), "Hi!")
  )
```

```
## # A tibble: 4 x 3
##   name  greeting1 greeting2
##   <chr> <chr>     <chr>
## 1 Flora Hi Flora! Hi Flora!
## 2 David Hi David! Hi David!
## 3 Terra Hi Terra! Hi Terra!
## 4 <NA>  Hi you!   Hi!
```

```r
# Use str_glue for more variable combinations, this interprets {} as outside the quotes
df |> mutate(greeting = str_glue("Hi {name}!"))
```

```
## # A tibble: 4 x 2
##   name  greeting
##   <chr> <glue>
## 1 Flora Hi Flora!
## 2 David Hi David!
## 3 Terra Hi Terra!
## 4 <NA>  Hi NA!
```

```r
# All into one string - this jives better with summarize()
str_flatten(c("x", "y", "z"))
```

```
## [1] "xyz"
```

```r
#> [1] "xyz"
str_flatten(c("x", "y", "z"), ", ")
```

```
## [1] "x, y, z"
```

```r
#> [1] "x, y, z"
str_flatten(c("x", "y", "z"), ", ", last = ", and ")
```

```
## [1] "x, y, and z"
```

```r
#> [1] "x, y, and z"
```

```r
# Separate into rows by delimiter character
df1 <- tibble(x = c("a,b,c", "d,e", "f"))
df1 |>
  separate_longer_delim(x, delim = ",")
```

```
## # A tibble: 6 x 1
##   x
##   <chr>
## 1 a
## 2 b
## 3 c
## 4 d
## 5 e
## 6 f
```

```
# Make each character a row
df2 <- tibble(x = c("1211", "131", "21"))
df2 |>
  separate_longer_position(x, width = 1)
```

```
## # A tibble: 9 x 1
##   x
##   <chr>
## 1 1
## 2 2
## 3 1
## 4 1
## 5 1
## 6 3
## 7 1
## 8 2
## 9 1
```

```
# Similar two functions but with columns

# Separate into columns by delimiter character
df3 <- tibble(x = c("a10.1.2022", "b10.2.2011", "e15.1.2015"))
df3 |>
  separate_wider_delim(
    x,
    delim = ".",
    names = c("code", "edition", "year") # Gotta name the columns
  )
```

```
## # A tibble: 3 x 3
##   code  edition year
##   <chr> <chr>   <chr>
## 1 a10   1       2022
## 2 b10   2       2011
## 3 e15   1       2015
```

```
df3 |>
  separate_wider_delim(
    x,
    delim = ".",
    names = c("code", NA, "year") # Ignore middle chunk
  )
```

```
## # A tibble: 3 x 2
##   code  year
##   <chr> <chr>
## 1 a10   2022
## 2 b10   2011
## 3 e15   2015
```

```
# Separate defined-size chunks of characters into named columns
df4 <- tibble(x = c("202215TX", "202122LA", "202325CA"))
df4 |>
  separate_wider_position(
    x,
    widths = c(year = 4, age = 2, state = 2)
  )
```

```
## # A tibble: 3 x 3
##   year  age   state
##   <chr> <chr> <chr>
## 1 2022  15    TX
## 2 2021  22    LA
## 3 2023  25    CA
```

```
df <- tibble(y = c("1-1-1", "1-1-2", "1-3", "1-3-2", "1"))

# Troubleshooting Problem Rows
df |>
  separate_wider_delim(
    y,
    delim = "-",
    names = c("x", "y", "z"),
    too_few = "align_start" # Or "align_end"
    # For too_many, you can either "drop" or "merge"
  )
```

```
## # A tibble: 5 x 3
##   x     y     z
##   <chr> <chr> <chr>
## 1 1     1     1
## 2 1     1     2
## 3 1     3     <NA>
## 4 1     3     2
## 5 1     <NA>  <NA>
```

```
str_length("fhifrh") # Counts characters
```

```
## [1] 6
```

```
str_length("c c")
```

```
## [1] 3
```

`str_sub(string, start, end)`

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3) # First three characters
```

```
## [1] "App" "Ban" "Pea"
```

```
str_sub(x, -3, -1) # Last three characters
```

```
## [1] "ple" "ana" "ear"
```

Regular Expressions

- `str_view(fruit, "berry")` will return all rows that contain the string "berry"
  - '.' is a wildcard
  - ? makes a pattern optional (i.e. it matches 0 or 1 times)
  - $*$ lets a pattern repeat (i.e. it matches at least once)

  - `*` lets a pattern be optional or repeat (i.e. it matches any number of times, including 0).
  - Character classes are defined by `[]` and let you match a set of characters, e.g., `[abcd]` matches "a", "b", "c", or "d". You can also invert the match by starting with ^: `[^abcd]` matches anything **except** "a", "b", "c", or "d".
  - `str_view(fruit, "aa|ee|ii|oo|uu")` finds any of these patterns

```r
# ab? matches an "a", optionally followed by a "b".
str_view(c("a", "ab", "abb"), "ab?")
```

```
## [1] | <a>
## [2] | <ab>
## [3] | <ab>b
```

```r
#> [1] | <a>
#> [2] | <ab>
#> [3] | <ab>b

# ab+ matches an "a", followed by at least one "b".
str_view(c("a", "ab", "abb"), "ab+")
```

```
## [2] | <ab>
## [3] | <abb>
```

```r
#> [2] | <ab>
#> [3] | <abb>

# ab* matches an "a", followed by any number of "b"s.
str_view(c("a", "ab", "abb"), "ab*")
```

```
## [1] | <a>
## [2] | <ab>
## [3] | <abb>
```

```r
#> [1] | <a>
#> [2] | <ab>
#> [3] | <abb>
```

```r
# Return a logical vector that is true if anything in brackets is matched
str_detect(c("a", "b", "c"), "[aeiou]")
```

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

```r
# Count how many matches in a string
x <- c("apple", "banana", "pear")
str_count(x, "p") # You can set ignore_case = TRUE
```

```
## [1] 2 0 1
```

```r
# It won't overlap!
str_count("abababa", "aba")
```

```
## [1] 2
```

```r
#> [1] 2

# str_replace() replaces the first match, and as the name suggests, str_replace_all()
↪  replaces all matches.
```

```r
x <- c("apple", "pear", "banana")
str_replace_all(x, "[aeiou]", "-")
```

```
## [1] "-ppl-"  "p--r"   "b-n-n-"
```

```r
str_replace(x, "[aeiou]", "-")
```

```
## [1] "-pple"  "p-ar"   "b-nana"
```

```r
str_remove_all(x, "[aeiou]")
```

```
## [1] "ppl" "pr"  "bnn"
```

```r
str_remove(x, "[aeiou]")
```

```
## [1] "pple"  "par"   "bnana"
```

Extract data out of one column into one or more new columns with `separate_wider_regex()`

```r
df <- tribble(
  ~str,
  "<Sheryl>-F_34",
  "<Kisha>-F_45",
  "<Brandon>-N_33",
  "<Sharon>-F_38",
  "<Penny>-F_58",
  "<Justin>-M_41",
  "<Patricia>-F_84",
)

df |> # Input dataframe
  separate_wider_regex(
    str, # Input column
    patterns = c(
      "<", # Separator, not named so it will disappear into the void
      name = "[A-Za-z]+", # Any letters + more letters
      ">-",
      gender = ".", # Any single letter
      "_",
      age = "[0-9]+" # Any number incl. more digits
    )
  )
```

```
## # A tibble: 7 x 3
##   name     gender age
##   <chr>    <chr>  <chr>
## 1 Sheryl   F      34
## 2 Kisha    F      45
## 3 Brandon  N      33
## 4 Sharon   F      38
## 5 Penny    F      58
## 6 Justin   M      41
## 7 Patricia F      84
```

Some exceptions and tricks:

- To match `"."`, use `"\\."`

- To match "\", use "\\\\"
- You can also use raw strings like `str_view(x, r"{\\}")`
- By default, regular expressions will match any part of a string. If you want to match at the start or end you need to anchor the regular expression using ^ (e.g. `^a`) to match the start or `$` (e.g. `a$`) to match the end
  - You can also force-match a string like `^a$` to avoid strings that include your pattern
- You can also match the boundary between words (i.e. the start or end of a word) with .
- Matching classes:
  - defines a range, e.g., [a-z] matches any lower case letter and [0-9] matches any number.
  - \ escapes special characters, so `[\^\-\]]` matches `^`, `-`, or].
  - `\d` matches any digit;
  - `\D` matches anything that isn't a digit.
  - `\s` matches any whitespace (e.g., space, tab, newline);
  - `\S` matches anything that isn't whitespace.
  - `\w` matches any "word" character, i.e. letters and numbers;
  - `\W` matches any "non-word" character.
  - Add a `+` at the end if you want to allow several in a row (like `\d+` for multi-digit)

```
x <- c("summary(x)", "summarize(df)", "rowsum(x)", "sum(x)")
str_view(x, "sum")
```

```
## [1] | <sum>mary(x)
## [2] | <sum>marize(df)
## [3] | row<sum>(x)
## [4] | <sum>(x)
```

```
#> [1] | <sum>mary(x)
#> [2] | <sum>marize(df)
#> [3] | row<sum>(x)
#> [4] | <sum>(x)
str_view(x, "\\bsum\\b")
```

```
## [4] | <sum>(x)
```

```
#> [4] | <sum>(x)
```

Regular expressions have their own precedence rules: quantifiers have high precedence and alternation has low precedence which means that ab+ is equivalent to `a(b+)`, and `^a|b$` is equivalent to `(^a)|(b$)`. Just like with algebra, you can use parentheses to override the usual order. But unlike algebra you're unlikely to remember the precedence rules for regexes, so feel free to use parentheses liberally.

- `dotall = TRUE` lets `.` match everything, including `\n`:
- `multiline = TRUE` makes^and$match the start and end of each line rather than the start and end of the complete string

```
# Comments = T allows comments in the middle
phone <- regex(
  r"(
    \(?     # optional opening parens
    (\d{3}) # area code
    [)\-]?  # optional closing parens or dash
    \ ?     # optional space
    (\d{3}) # another three numbers
    [\ -]?  # optional space or dash
    (\d{4}) # four more numbers
  )",
  comments = TRUE
)

str_extract(c("514-791-8141", "(123) 456 7890", "123456"), phone)
```

```
## [1] "514-791-8141"   "(123) 456 7890" NA
```

Factors

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
x1 <- c("Dec", "Apr", "Jan", "Mar")
y1 <- factor(x1, levels = month_levels) # No defined levels = alphabetical order
sort(y1) # It has an order now!
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

```
levels(y1)
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

```
# Import as factor (remember that we defined month_levels above!)
csv <- "
month,value
Jan,12
Feb,56
Mar,12"

# df <- read_csv(csv, col_types = cols(month = col_factor(month_levels))) # THIS CODE
↪  WON'T RUN **************
# df$month
```

- `count(factor)` will return a frequency table of factor levels
- To reorder a factor: `fct_reorder(.f = factor, .x = by)`
  - .x: numeric vector to reorder by such as another column in a dataframe
  - `fct_relevel(factor, "Column to move to top", "Another column to move to top")`
  - `fct_reorder2(.f, .x, .y)` reorders the factor `.f` by the `.y` values associated with the largest `.x` values. This makes the plot of `x` vs `y` colored by `f` easier to read because the colors of the line at the far right of the plot will line up with the legend.
  - `fct_infreq()` or `fct_rev(fct_infreq())` sorts by increasing or decreasing frequency
- To rename levels of a factor: `df |> mutate(factor = fct_recode(factor, "OLD NAME"    = "NEW NAME", "OLD NAME 2" = "NEW NAME 2"))`
  - Unmentioned levels will stay the same
  - You can assign multiple old names to the same new name
    * To do that faster: `df |> mutate(factor = fct_collapse(factor, "NEW NAME 1" = c("OLD NAME 1", "OLD NAME 2"), "NEW NAME 2" = c("OLD NAME 3", "OLD NAME 4"),)`
- To mash low-frequency levels together:
  - `fct_lump_lowfreq()` is a simple starting point that progressively lumps the smallest groups categories into "Other", always keeping "Other" as the smallest category.
  - `fct_lump_n(factor, n = 10)` will make 10 categories, and the 10-n smallest categories will all

  be lumped into "Other."
  - Read the documentation to learn about `fct_lump_min()` and `fct_lump_prop()` which are useful in other cases.
- Ordered factors have an equal distance between levels, they behave pretty much the same except for color or fill (defaults to viridis) or linear models. Create one with `ordered(c("a", "b", "c"))`

Dates and Times

- lubridate is now in tidyverse, yay
- Classes include date, time (no native R class, hms package has one), and date-time (POSIXct in R, `<dttm>` in tibbles)
  - If you don't really need time, dates are much easier to work with than date-times!
  - `today()` returns today's date, `now()` returns the current date-time
- Making date-times
  - Import CSV, readr should automatically detect ISO8601 (`yyyy-mm-dd hh:mm:ss` OR `yyyy-mm-ddThh:mm:ss`)
    * For other formats, use `col_types` + `col_date()` or `col_datetime()` with a date-time format

Date Formats Understood by Readr

Type	Code	Meaning
Year	%Y	4 digit year
	%y	2 digit year
Month	%m	Number
	%b	Abbreviated (ex. Feb)
	%B	Full name
Day	%d	1-2 digits
	%e	2 digits
Time	%H	24 hour hour
	%I	12 hour hour
	%p	am or pm
	%M	Minutes
	%S	Seconds
	%OS	Seconds with decimal component
	%Z	Time zone name
	%z	Offset from UTC
Other	%.	Skip one non-digit, (ex. :)
	%*	Skip any number of non-digits

```r
csv <- "date
  01/02/15"

read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))
```

```
## # A tibble: 1 x 1
##   date
##   <date>
## 1 2015-01-02
```

```r
#> 1 2015-01-02
read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))
```

```
## # A tibble: 1 x 1
##   date
##   <date>
## 1 2015-02-01
```

```
#> 1 2015-02-01
read_csv(csv, col_types = cols(date = col_date("%y/%m/%d")))
```

```
## # A tibble: 1 x 1
##   date
##   <date>
## 1 2001-02-15
```

```
#> 1 2001-02-15

# Parsing strings for dates
ymd("2017-01-31")
```

```
## [1] "2017-01-31"
```

```
#> [1] "2017-01-31"
mdy("January 31st, 2017")
```

```
## [1] "2017-01-31"
```

```
#> [1] "2017-01-31"
dmy("31-Jan-2017")
```

```
## [1] "2017-01-31"
```

```
#> [1] "2017-01-31"

# Parsing strings for date-times
ymd_hms("2017-01-31 20:11:59")
```

```
## [1] "2017-01-31 20:11:59 UTC"
```

```
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
```

```
## [1] "2017-01-31 08:01:00 UTC"
```

```
#> [1] "2017-01-31 08:01:00 UTC"
ymd("2017-01-31", tz = "UTC") # Force to time without _hm
```

```
## [1] "2017-01-31 UTC"
```

```
#> [1] "2017-01-31 UTC"

# Sticking date-time components together
date_split <- flights |>
  select(year, month, day, hour, minute)
date_split %>%
    mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
## # A tibble: 336,776 x 6
##      year month   day  hour minute departure
##     <int> <int> <int> <dbl>  <dbl> <dttm>
##   1  2013     1     1     5     15 2013-01-01 05:15:00
##   2  2013     1     1     5     29 2013-01-01 05:29:00
##  3  2013     1     1     5     40 2013-01-01 05:40:00
##  4  2013     1     1     5     45 2013-01-01 05:45:00
##  5  2013     1     1     6      0 2013-01-01 06:00:00
##  6  2013     1     1     5     58 2013-01-01 05:58:00
##  7  2013     1     1     6      0 2013-01-01 06:00:00
##  8  2013     1     1     6      0 2013-01-01 06:00:00
##  9  2013     1     1     6      0 2013-01-01 06:00:00
## 10  2013     1     1     6      0 2013-01-01 06:00:00
## # i 336,766 more rows
```

```
# Pulling date-time components apart
datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime)
```

```
## [1] 2026
```

```
#> [1] 2026
month(datetime)
```

```
## [1] 7
```

```
#> [1] 7
mday(datetime)
```

```
## [1] 8
```

```
#> [1] 8

yday(datetime) # Day of the year
```

```
## [1] 189
```

```
#> [1] 189
wday(datetime) # Day of the week
```

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

```
#> [1] 4

month(datetime, label = TRUE) # As name, defaults to abbreviated
```

```
## [1] Jul
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

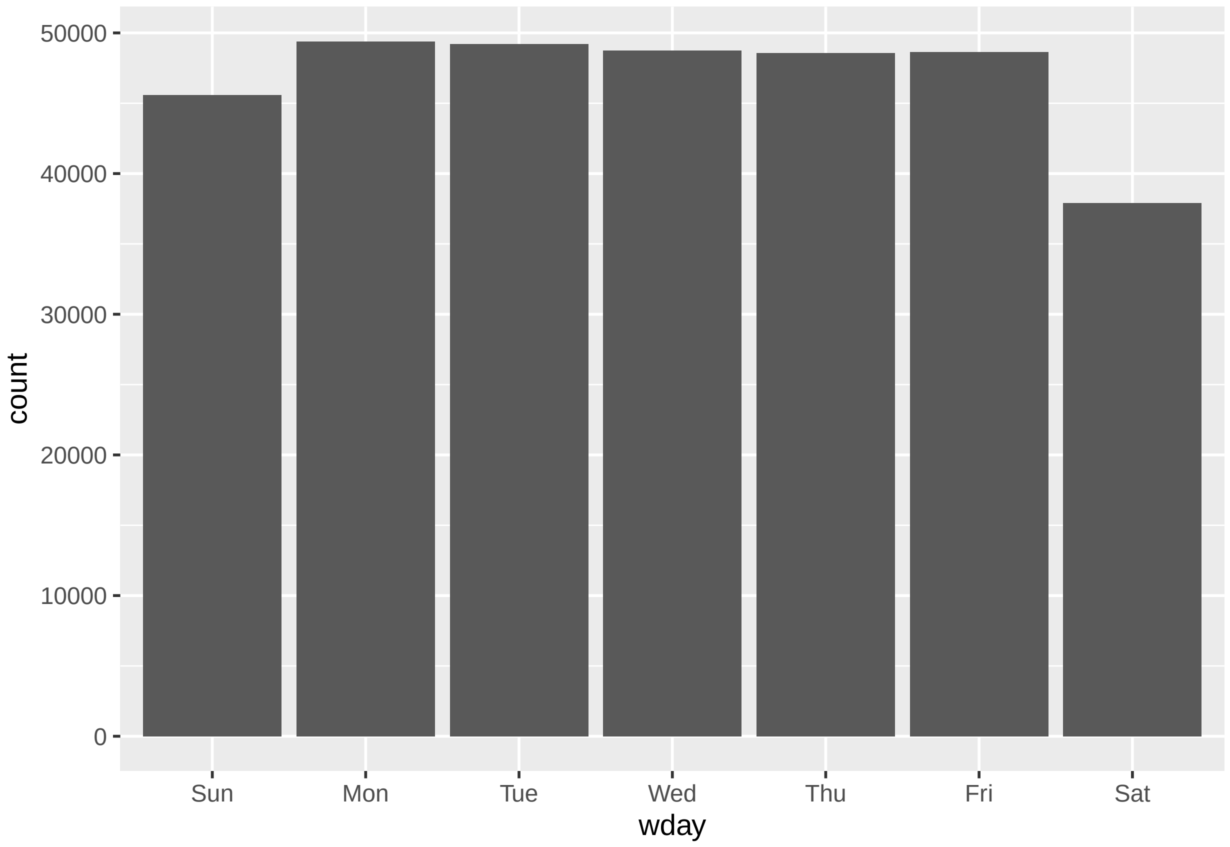
```
#> [1] Jul
wday(datetime, label = TRUE, abbr = FALSE)
```

```
## [1] Wednesday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

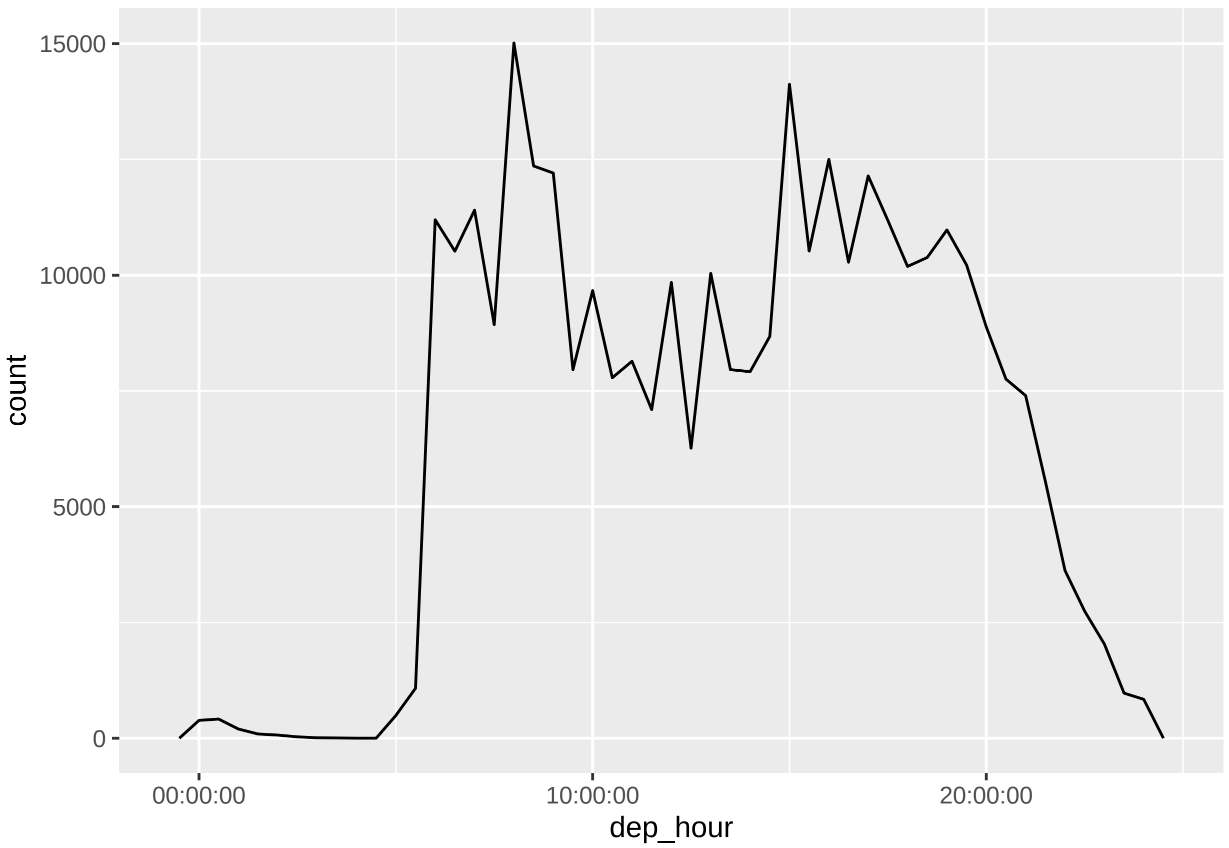
```
#> [1] Wednesday
```

- `as_date()` and `as_datetime()` can switch between the two
- Date-times may come as numeric offsets from 1970-01-01 (Unix Epoch)
  - If the offset is in seconds (like HUGE number) use `as_datetime()`
  - If the offset is in days use `as_date()`

```r
# Setup
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights |>
  filter(!is.na(dep_time), !is.na(arr_time)) |>
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
    sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
  ) |>
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt |>
  mutate(wday = wday(dep_time, label = TRUE)) |>
  ggplot(aes(x = wday)) +
  geom_bar()
```



```r
# Subtracting times = a difftime, which can be converted to an hms object
flights_dt |>
  mutate(dep_hour = hms::as_hms(dep_time - floor_date(dep_time, "day"))) |>
  # floor_date(dep_time, "day") = first instant of day, so subtracting it from dep_time
  ↪  gets you the time of day it departed
  # round_date(), floor_date(), and ceiling_date() round datetimes
  # "day" is the unit - we could also round to the nearest second, minute, 2 minutes,
  ↪  etc.
  # round goes closest specified unit (halfway rounds up), floor rounds down, ceiling
  ↪  rounds up
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)
```



```
(datetime <- ymd_hms("2026-07-08 12:34:56"))
```

```
## [1] "2026-07-08 12:34:56 UTC"
```

```
#> [1] "2026-07-08 12:34:56 UTC"

year(datetime) <- 2030 # Precision fixing
datetime
```

```
## [1] "2030-07-08 12:34:56 UTC"
```

```
#> [1] "2030-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
```

```
## [1] "2030-01-08 12:34:56 UTC"
```

```
#> [1] "2030-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1 # Move one timezone
datetime
```

```
## [1] "2030-01-08 13:34:56 UTC"
```

```
#> [1] "2030-01-08 13:34:56 UTC"

update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
```

```
## [1] "2030-02-02 02:34:56 UTC"
```

```
#> [1] "2030-02-02 02:34:56 UTC"

# If values are too big they'll roll over
update(ymd("2023-02-01"), mday = 30)
```

```
## [1] "2023-03-02"
```

```
#> [1] "2023-03-02"
update(ymd("2023-02-01"), hour = 400)
```

```
## [1] "2023-02-17 16:00:00 UTC"
```

```
#> [1] "2023-02-17 16:00:00 UTC"
```

Time Spans

- Duration: Number of seconds
- Period: Human unit of time passing (ex. 3 weeks)
- Interval: Start and endpoint
- Pick the simplest data structure that solves your problem

```
h_age <- today() - ymd("1979-10-14")
# This will return a difftime object, which is kinda annoying
as.duration(h_age)
```

```
## [1] "1408665600s (~44.64 years)"
```

```
# This will return a duration, which is easier to work with

# Duration constructors
dseconds(15)
```

```
## [1] "15s"
```

```
#> [1] "15s"
dminutes(10)
```

```
## [1] "600s (~10 minutes)"
```

```
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
```

```
## [1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
```

```
## [1] "0s"                 "86400s (~1 days)"  "172800s (~2 days)"
## [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

```
#> [1] "0s"                 "86400s (~1 days)"  "172800s (~2 days)"
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
```

```
## [1] "1814400s (~3 weeks)"
```

```
#> [1] "1814400s (~3 weeks)"
dyears(1)
```

```
## [1] "31557600s (~1 years)"
```

```
#> [1] "31557600s (~1 years)"
```

Duration Math

- You can add and multiply durations
- You can add and subtract durations and days
  - Remember that a duration is a number of seconds, so daylight savings, etc. may give weird results

Periods

- Periods don't have a fixed length in seconds, so they work more intuitively
- Periods can be added and subtracted with days
- Periods can be added to each other and multiplied
- Periods can be added to dates

```
hours(c(12, 24))
```

```
## [1] "12H 0M 0S" "24H 0M 0S"
```

```
days(7)
```

```
## [1] "7d 0H 0M 0S"
```

```
print("***Months***")
```

```
## [1] "***Months***"
```

```
months(1:6)
```

```
## [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
## [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

```
print("***Math***")
```

```
## [1] "***Math***"
```

```
10 * (months(6) + days(1))
```

```
## [1] "60m 10d 0H 0M 0S"
```

```
days(50) + hours(25) + minutes(2)
```

```
## [1] "50d 25H 2M 0S"
```

```
ymd("2024-01-01") + dyears(1)
```

```
## [1] "2024-12-31 06:00:00 UTC"
```

Intervals

- Create an interval by writing `start %--% end`

```
y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

c(y2023, y2024)
```

```
## [1] 2023-01-01 UTC--2024-01-01 UTC 2024-01-01 UTC--2025-01-01 UTC
```

```
y2023 / days(1)
```

```
## [1] 365
```

```
y2024 / days(1) # Leap year!
```

```
## [1] 366
```

Timezones are a hot mess. Be afraid of them.

- R uses the IANA time zone names, usually continent/city or ocean/city, ex. "America/New_York"
  - IANA does this in order to avoid two countries naming a timezone the same thing, and avoid having to deal with country names changing since cities and continents tend to stay pretty consistent.
  - This needs to cover historical times, so there are a LOT of timezones including ones that are merged in modern day
- `Sys.timezone()` tells you what timezone R thinks it is in
- `OlsonNames()` gives a list of all the timezone names
- lubridate uses UTC, which is roughly equivalent to GMT but does not have daylight savings

```
# Setup
x1 <- ymd_hms("2024-06-01 12:00:00", tz = "America/New_York")
x2 <- ymd_hms("2024-06-01 18:00:00", tz = "Europe/Copenhagen")
x3 <- ymd_hms("2024-06-02 04:00:00", tz = "Pacific/Auckland")
x4 <- c(x1, x2, x3)

x4a <- with_tz(x4, tzone = "Australia/Lord_Howe") # Same instant in time, displays
↪  different time and timezone

x4b <- force_tz(x4, tzone = "Australia/Lord_Howe") # Different instant in time, this is
↪  for when an instant has been labeled with the wrong timezone but it is displaying the
↪  correct time
```

Missing values

- Missing values may be stored as `NA` or implicit
- `NA` values in hand-entered data often mean that the value from the previous row is carried forward
  - Fill this in with `fill(column)` from tidyr
- `NA` values may represent one specific value in a dataset, like 0. In this case, use `dplyr::coalesce(x, 0)`
- Numbers may represent missing values * some older software uses 99 or -999 to represent missing values
  - Handle on import: `read_csv(path, na = "99")`
  - Handle later: `na_if(x, -99)`
- `NaN` = "Not a number", generally behaves like `NA`
  - Produced by 0/0 and other evil math
- Pivoting can make implicit missing values into `NA`
  - You can also give `complete()` the combinations of rows and columns that should exist, which will make implicit missing values explicit

  - To find rows that are in df x but not in df y, use `anti_join(x,y)`, which can tell use missing values in y if they are supposed to have matching values
- Empty group: Factor level with no actual observations
  - We can include empty levels with `count or group_by(x, .drop = F)` or + `scale_x_discrete(drop = F)`

Joins

- Before joining, make sure the variable you're joining by is a unique identifier and you are not missing an observation (`NA`)
- dplyr join functions (see cheat sheet):
  - `left_join()` (mutating)
  - `inner_join()` (mutating)
  - `right_join()` (mutating)
  - `full_join()` (mutating)
  - `semi_join()` (filtering)
  - `anti_join()` (filtering)
- dplyr joins will default to using all variables with matching names as the key
  - If you don't want that, `_join(x, join_by(keyname))` or `_join(x, join_by(keyname == keyname))`
  - Remaining variables with the same names will be given `.x` and `.y` suffixes
  - You can also `join_by(keyname <= keyname)` in extremely specific cases
  - `cross_join()` gets you every permutation
  - `join_by(closest(x <= y))` matches the smallest y <= x, which **can be useful when combining two tables with dates that don't have the exact same intervals**
  - Overlap joins include `between()`, `within()`, and `overlaps()`, which can be useful for making sure intervals don't overlap

Imports

- xml2 package can import XML data
- readxl package can load Excel
  - Non-core tidyverse, so need `library(readxl)` and maybe `library(writexl)`
  - `read_xls()`, `read_xlsx()`, and `read_excel()` read in Excel files. `read_excel()` guesses whether the format is `xls` or `xlsx`
  - First argument is the filepath, then `col_names = c(), skip = 1` if you want to change the variable names, `na = c("", "N/A", etc)` if you need to specify what `NA` looks like, and `col_types = c("skip", "guess", "logical", "numeric", "date", "text" or "list" for each variable)`. "list" will store each item as a vector length one with its own type.
  - `col_type` of `"numeric"` will turn any issues into NA values
  - Defaults to first sheet, but you can use `read_excel(path, sheet = "Sheet Name")`
    - Inspect with `excel_sheets(path)`
  - You can import only a certain range with `read_excel(path, range = A4:F15)` * this example would skip the first three rows
  - Excel has fun and funky data types
- googlesheets3 can import Google SHeets
  - `read_sheet(URL or file ID)`
  - Can supply `col_names`, `skip`, `na`, `range`, `sheet` and `col_types = "dccc"` just like Excel - note that the column types are coded as single letters like d = double and c = character
  - `write_sheet()` also exists
- Data can also be imported directly from databases using an SQL query
  - `library(DBI)` executes SQL
  - `library(dbplyr)` translates dplyr to SQL
  - Connect to database

    - Load data
    - Check data
    - SQL is syntax for querying databases, selecting which data you are interested in. Most important: `SELECT variable, variable, variable` and `FROM table`
    - More on SQL
- The arrow package is Apache Arrow for R, which can get data from the parquet format, often used for big data
    - Very fast and can handle huge datasets
    - Arrow Instructions
    - Parquet is very efficient, but generally unreadable to humans
- A lot of web data is hierarchical (tree-like) and that's got a package too
- Web scraping textbook link

Code

Functions

- Once you've got three copies of the same code, you should really be writing a function

```
name <- function(argument1, argument2 = default) {
  body (repeating code, calls arguments like variables)
}
```

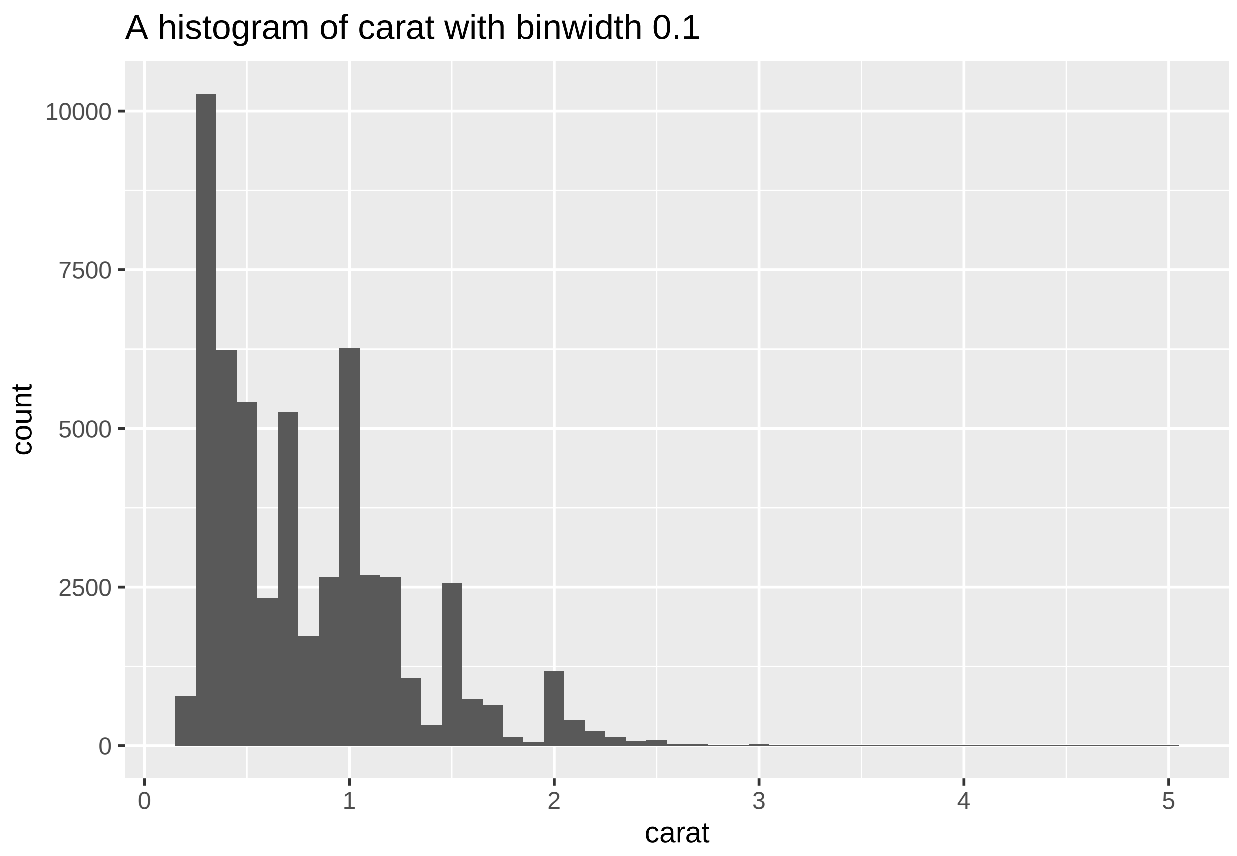
- Test a new function with a few simple inputs
- Putting tidyverse code in a function can cause some problems
    - To look for columns in a dataframe, use brackets! This is called embracing.

  ```
  function(df, col1) {
  df %>%
    summarize(mean({{ col1 }}))
  }
  ```
- A similar problem arises with ggplot

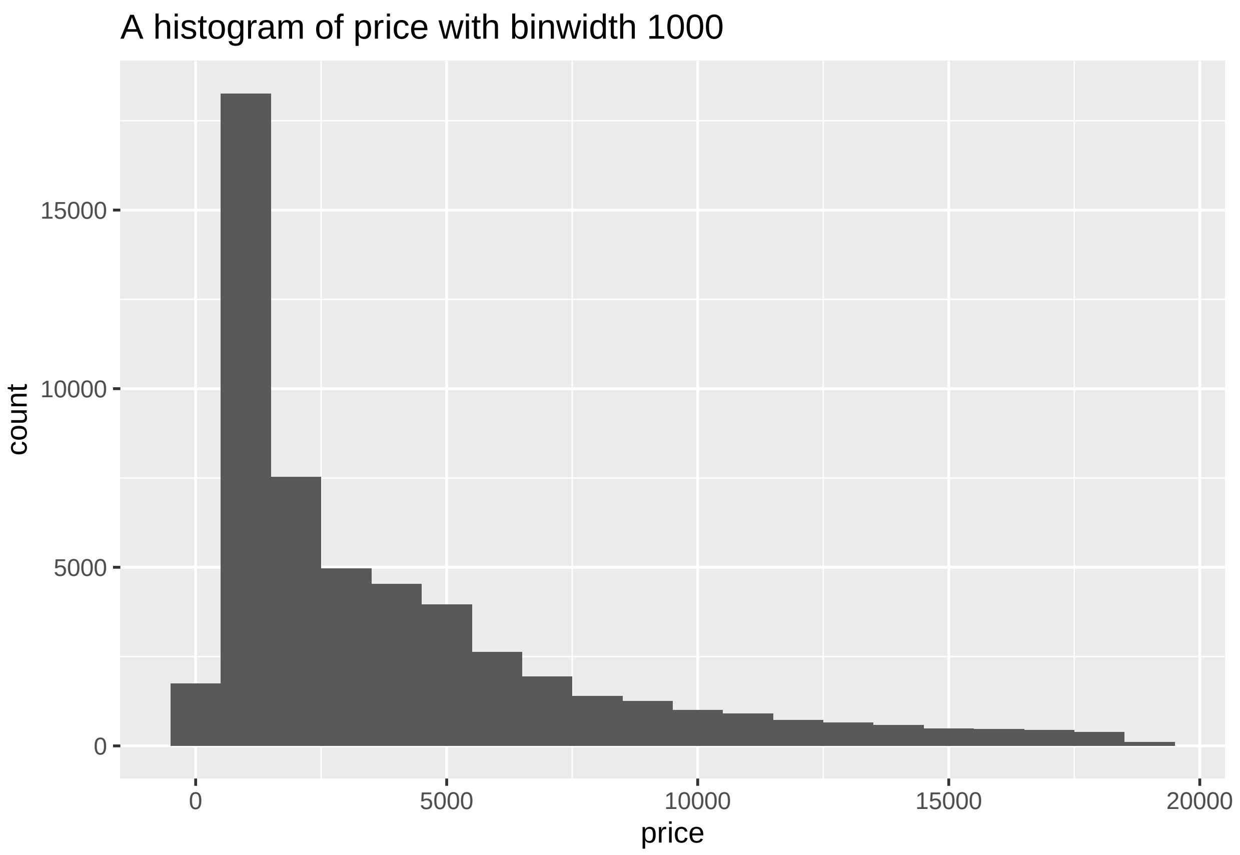
```
histogram <- function(df, var, binwidth = NULL) {
  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth)
}

histogram(data) +
  theme_bw()
```

- To label within a ggplot function, you can also embrace!

```
histogram <- function(df, var, binwidth) {
  label <- rlang::englue("A histogram of {{var}} with binwidth {binwidth}")

  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth) +
    labs(title = label)
}

diamonds |> histogram(carat, 0.1)
```



```
diamonds |> histogram(price, 1000)
```



Iteration

- The purrr package from tidyverse is useful for programming and iteration

```
# To compute the median of every column
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
# Cringe
df |> summarize(
  n = n(),
  a = median(a),
  b = median(b),
  c = median(c),
  d = median(d),
)
```

```
## # A tibble: 1 x 5
##       n      a       b      c      d
##   <int>  <dbl>   <dbl>  <dbl>  <dbl>
## 1    10 -0.250 -0.0136 -0.554 -0.155
```

```
# Good! Shorter, same output
df |> summarize(
  n = n(),
  across(a:d, median), # across(columns, function, .names) or
  # (columns, list(output1 = function1, output2 = function2))
)
```

```
## # A tibble: 1 x 5
##       n      a       b      c      d
##   <int>  <dbl>   <dbl>  <dbl>  <dbl>
## 1    10 -0.250 -0.0136 -0.554 -0.155
```

- `across()` can also be used with filtering, such as ‘across(where(is.Date), list(Year = year, Month = month))

Importing A Shitton Of Files

1. `paths <- list.files(path, pattern, full.names)`

- `path` is the directory of interest
- `pattern` is usually something like `[.]xlsx$` or `[.]csv$`
- `full.names` is true or false, determines whether the directory name should be included in the output. You should default to `TRUE`.

2. `files <- map(paths, read_excel)`

- `purrr::map()` is like `across()` but for each element in a vector

3. `list_rbind(files)`

- Alternate option: `paths %>% Step 2 %>% Step 3`

4. If that doesn’t work, try inspecting the files for matching types and then `rbind()`

Exporting A Shitton Of Files

```
by_clarity <- diamonds |>
  group_nest(clarity)
# This gives a new tibble with one clarity column and one data column, which contains a
↪  listed
by_clarity$data[[1]]
```

```
## # A tibble: 741 x 9
##    carat cut       color depth table price     x     y     z
##    <dbl> <ord>     <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.32 Premium   E      60.9    58   345  4.38  4.42  2.68
##  2  1.17 Very Good J      60.2    61  2774  6.83  6.9   4.13
##  3  1.01 Premium   F      61.8    60  2781  6.39  6.36  3.94
##  4  1.01 Fair      E      64.5    58  2788  6.29  6.21  4.03
##  5  0.96 Ideal     F      60.7    55  2801  6.37  6.41  3.88
##  6  1.04 Premium   G      62.2    58  2801  6.46  6.41  4
##  7  1    Fair      G      66.4    59  2808  6.16  6.09  4.07
##  8  1.2  Fair      F      64.6    56  2809  6.73  6.66  4.33
##  9  0.43 Very Good E      58.4    62   555  4.94  5     2.9
## 10  1.02 Premium   G      60.3    58  2815  6.55  6.5   3.94
## # i 731 more rows
```

```
by_clarity <- by_clarity |> # Add an output name by clarity
  mutate(path = str_glue("diamonds-{clarity}.csv"))
```

```
walk2(by_clarity$data, by_clarity$path, write_csv) # map2() varies the first and second
↪  arguments, walk2() is similar but is used when we doin't care about the output as
↪  much
# This is like write_csv(by_clarity$data[[every]], by_clarity$path[[every]])
```

Saving plots

```
# Make plot function
carat_histogram <- function(df) {
  ggplot(df, aes(x = carat)) + geom_histogram(binwidth = 0.1)
}

# Create a list of many plots and their filepaths
by_clarity <- by_clarity |>
  mutate(
    plot = map(data, carat_histogram),
    path = str_glue("clarity-{clarity}.png")
  )

# Save them all with walk2() and ggsave()
walk2(
  by_clarity$path,
  by_clarity$plot,
  \(path, plot) ggsave(path, plot, width = 6, height = 6)
)
# This is like ggsave(by_clarity$path[[all]], by_clarity$plot[[all]], width = 6, height =
↪  6)
```

Link to more functionals information

Base R

- Tidyverse is not the only solution
- `[` is used to extract components from vectors and dataframes
  - `vector[i]` returns an item in a vector. *`i` can be a number,
    - a vector of positive integers (including repeats),
    - a vector of negative integers (drops the elements at specific positions, like "give me the vector but not items 3 and 5"),
    - a logical vector (keeps everything corresponding to TRUE, useful if you get a logical vector from a comparison function),
    - a character vector if the vector is named (ex. `x <* c(abc = 1, def = 2, xyz = 5)`, `x[c("xyz", "def")]`),
    - `x[]` returns `x`. This can be useful for subsetting 2D stuff.
  - `df[rows, cols]` subsets a dataframe, or `df[rows, ]` to get those entire rows
    - Subsetting a dataframe this way returns a vector if you only ask for one column (`drop = F` returns a one-column dataframe though), and returns a dataframe if it selects more than one.
    - Subsetting a tibble returns a tibble
- `$` and `[[]]` are used to select single elements
  - These can access columns, `$` is specialized to access by name
  - These can also be used to make new columns, ex. `tb$z <* tb$x + tb$y`
  - `pull()` takes a variable name or position and returns the column
- Tibbles are different from data frames: They are more strict, requiring you to match a variable name exactly

- `[]` vs `[[]]`
    - `list[i]` extracts a list (even if it is length 1)
    - `list[[i]]` extracts an item from a list
    - `list$i` also extracts an item
    - `df["col"]` returns a one-column data frame
    - `df[["col"]]` returns a vector
- `apply()` applies a function over each element of a matrix or array
- `lapply()` applies a function to every element in a list. For less advanced operations, this can be used interchangeabley with `purrr::map()`
- `sapply()` is similar to `lapply()` but tries to simplify the result - not recommended for programming

For loops

```
for (element in vector) {
 # do something in here
 print(element * 2)
}
```

- To get output, make an empty list with the names you want

```
# Make a list of paths
paths <- dir("data/gapminder", pattern = "\\.xlsx$", full.names = TRUE)
# This is what we want to do:
files <- map(paths, readxl::read_excel)
# Make an empty list to put the output in:
files <- vector("list", length(paths))
# Add things to that empty list
for (i in seq_along(paths)) { # seq_along() generates an indes for each element of paths
  files[[i]] <- readxl::read_excel(paths[[i]])
}
# Put all the tibbles in the list into one tibble
do.call(rbind, files)
```

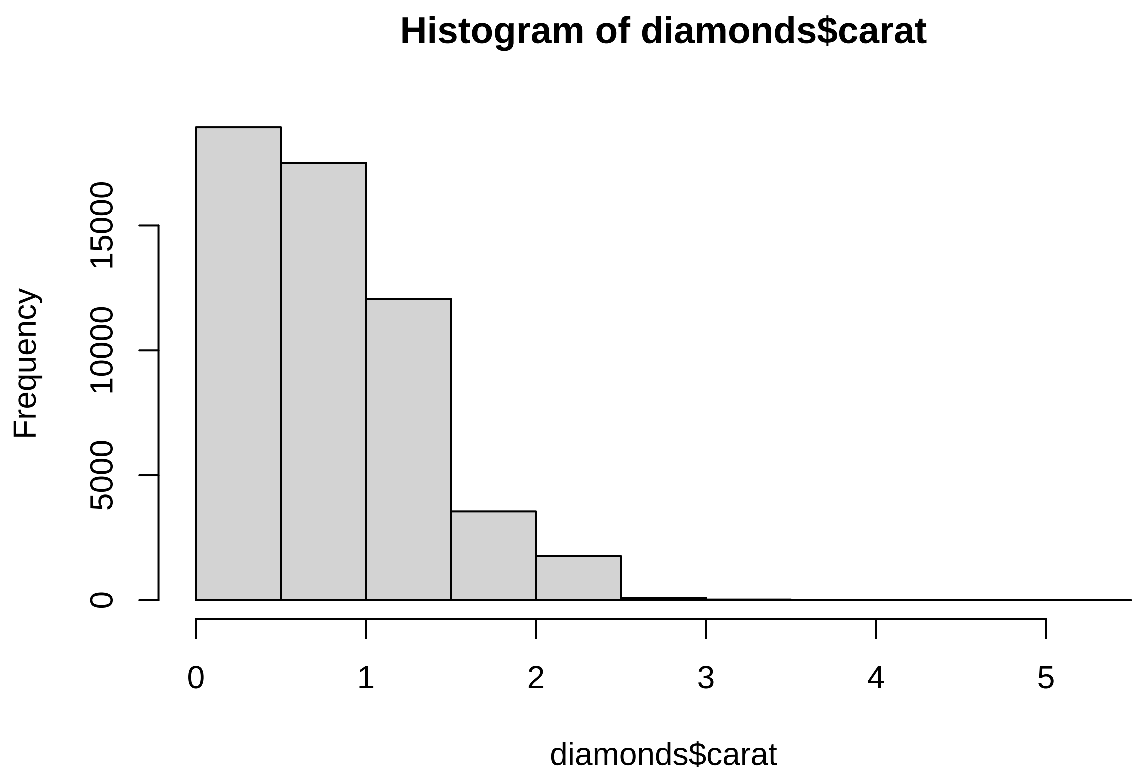
```
## NULL
```

```
# An alternative, building the data frame piece-by-piece:
out <- NULL
for (path in paths) {
  out <- rbind(out, readxl::read_excel(path))
} # Note that this can be really slow!
```

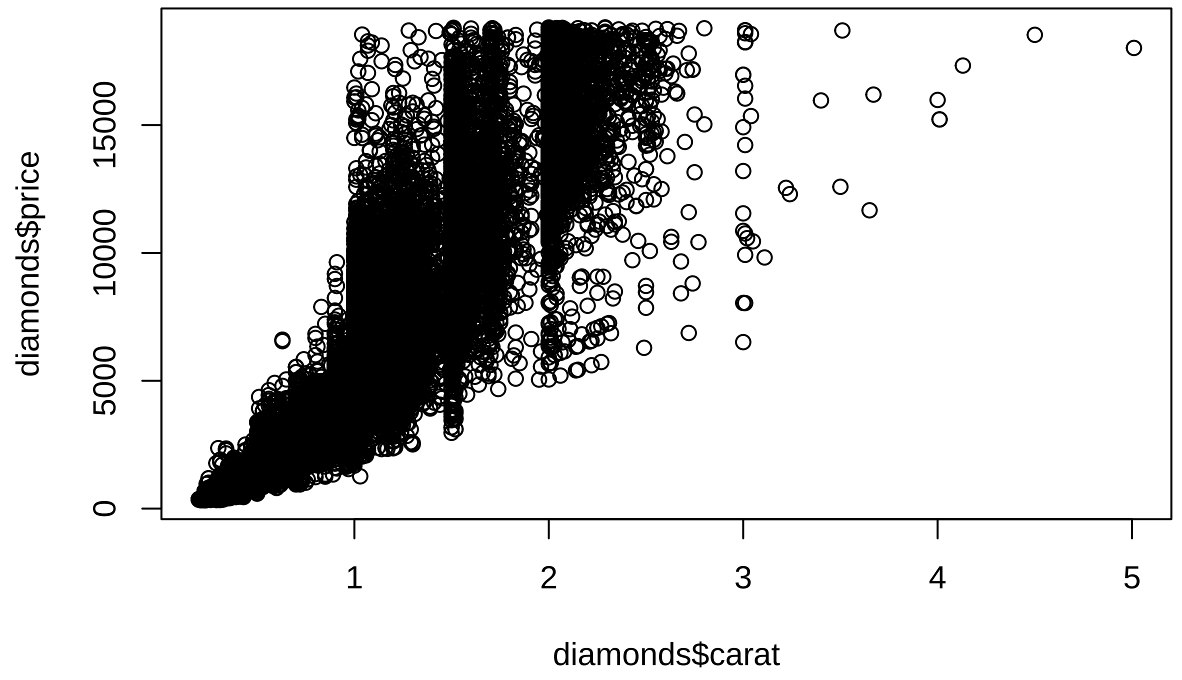
Plots

- Base R plots can be useful for quick exploratory analysis

```
hist(diamonds$carat)
```



```
plot(diamonds$carat, diamonds$price)
```



- Final note: Quarto is a code for integrating prose, code, and results. It is useful for communication and can produce dashboards, websites, and books.
  - Someone's working on a Reed thesis template in Quarto
  - Quarto is a command line interface tool
  - Link to Quarto documentation
  - Quarto is similar to RMarkdown including packages from RMarkdown, and it supports Python

  and Julia.
  - Quarto documents (`.qmd`) can be run and edited in RStudio (source editor or visual editor)