DATA 100,Week 2 (B)

Comments on typeof a vector

```
x <- c(1L, 3L)
typeof(x)
#> [1] "integer"
```

Situation becomes subtler when types are mixed together.

- ✓ If two or more atomic types are in the same vector, the type of the vector is the type that needs the most bits to represent.

- ✓ In short, boolean < integer < double < character

DATA 100,Week 2 (B)

boolean < integer < double < character

```
# boolean and integer mix
x <- c(TRUE, 2L)
typeof(x)
#> [1] "integer"

# integer and double mix
x <- c(1L, 3)
typeof(x)
#> [1] "double"
```

DATA 100,Week 2 (B)

boolean < integer < double < character

```
# double and character mix
x <- c(3, "string")
typeof(x)
#> [1] "character"

# more types mix
x <- c(TRUE, 1L, 3, "a")
typeof(x)
#> [1] "character
```

DATA 100,Week 2 (B)

When the construction involves non-atomic types, it basically follows the same rule, and turns the whole thing into a list.

```
x <- c(TRUE, 1L, 3, "a", sin)
typeof(x)
#> [1] "list"
```

DATA 100,Week 2 (B)

Continue with ggplot2 basics (1e: Chapter 3 and 2e: Chapter 10)

Look at diamonds **data**

library(tidyverse)

glimpse(diamonds)

```
#> Rows: 53,940
#> Columns: 10
#> $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23,~
#> $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very Good, ~
#> $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, J, J,~
#> $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI1, VS~
#> $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, 59.4,~
#> $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62,~
#> $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340,~
#> $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, 4.00,~
#> $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, 4.05,~
#> $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, 2.39,~
```

DATA 100,Week 2 (B)

It shows

- the number of variables as Columns: 10, and
- the number of observations as Rows:53,940.
- Also, the types of variables are shown after the names of each.

Use ?diamonds in Console to see more info.

Aside: What is in a diamond?

Chemically, a diamond is formed by carbon atoms arranged in a particularly tight fashion.

DATA 100,Week 2 (B)

Note: Visualization turns data into pictures.

• geom_point: does not do much behind the scene

• geom_smooth: does quite a bit behind the scene.

Most geom_ functions do some statistics behind the scene and lead to understanding of some aspects of the dataset.

Generally should try a few of them for a fuller picture.

DATA 100,Week 2 (B)

Use coord_flip

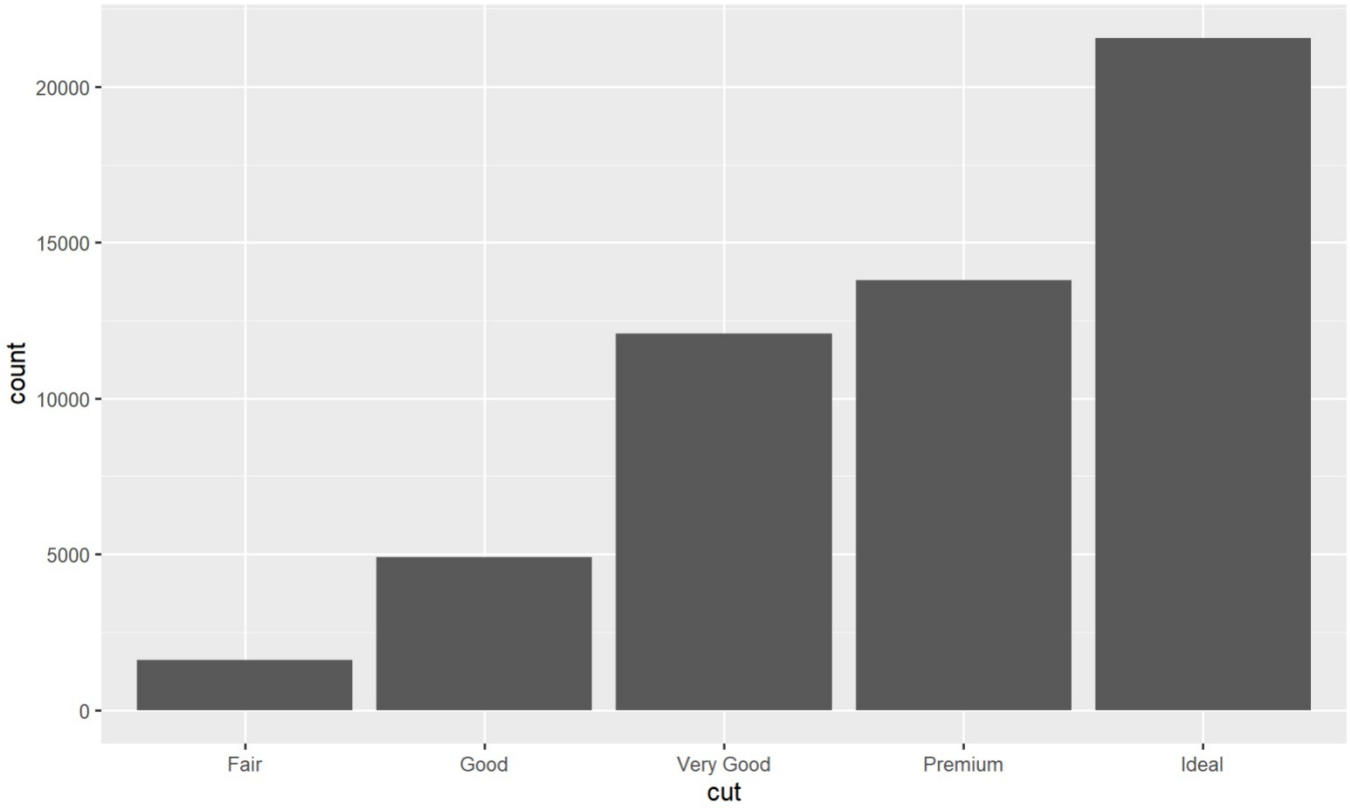
The function coord_flip() turns the plot sideways *flip*ping the x and y directions, so that

- x is vertical, and y is horizontal

DATA 100,Week 2 (B)

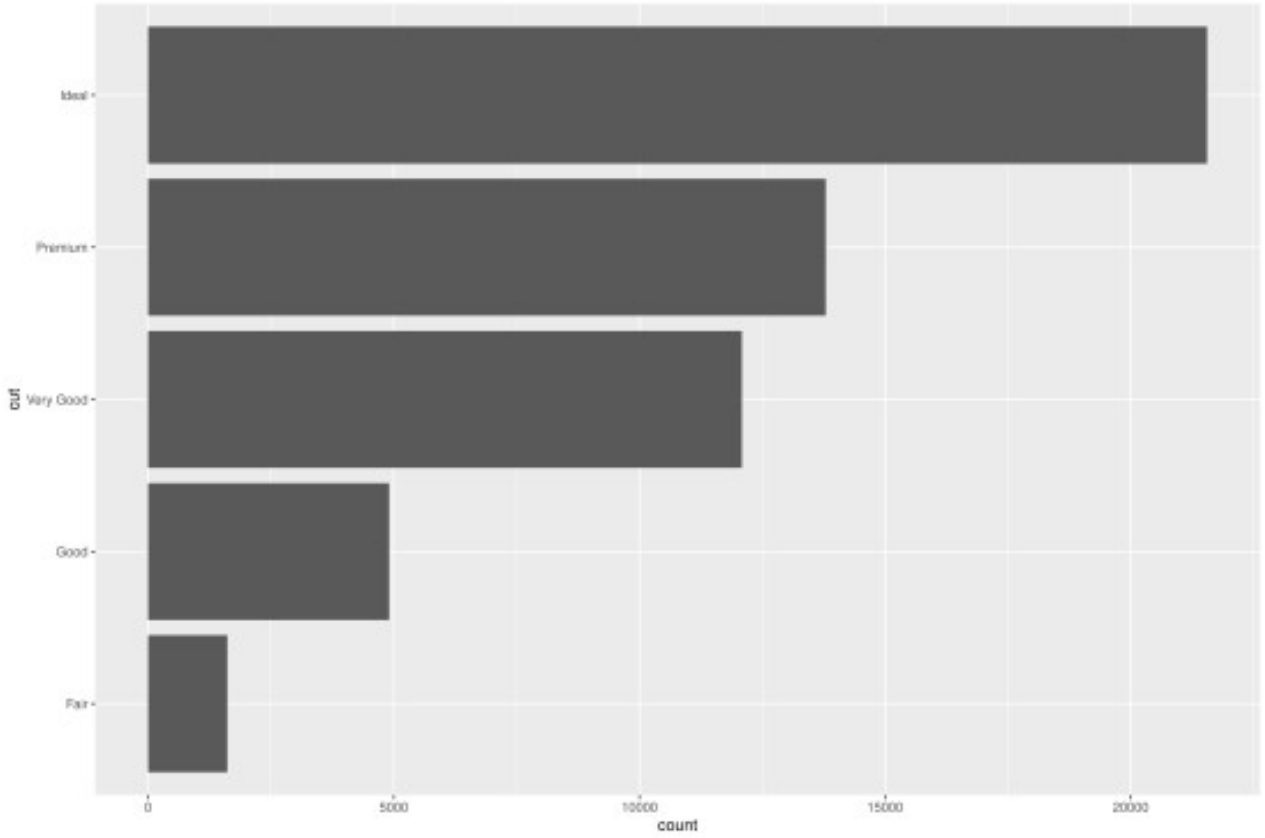
Bar diagram

```
ggplot(data = diamonds, mapping = aes(x = cut)) + geom_bar()
```



DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = cut)) + geom_bar() + coord_flip()
```



DATA 100,Week 2 (B)

Observations:

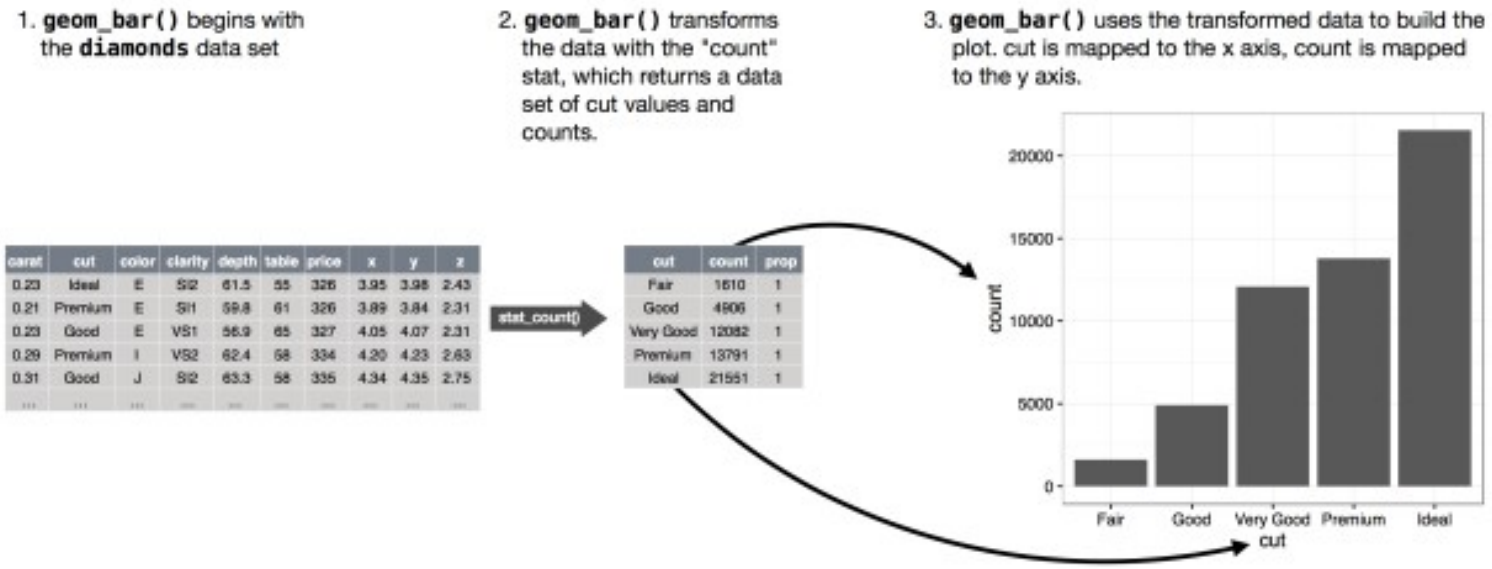
The count variable on the horizontal axis is *not* one of the variables in the original diamonds dataframe. It is an aggregate statistic

• that counts the number of observations with each of the cut value

When the function geom_bar() is called, it does the computation (counting) and forms a new dataframe, with count as a variable

• then the new dataframe is plotted as a bar diagram

DATA 100,Week 2 (B)



Usage: Bar diagram is most natural for categorical variables, which have discrete values.

DATA 100,Week 2 (B)

geom_ **and** stat_

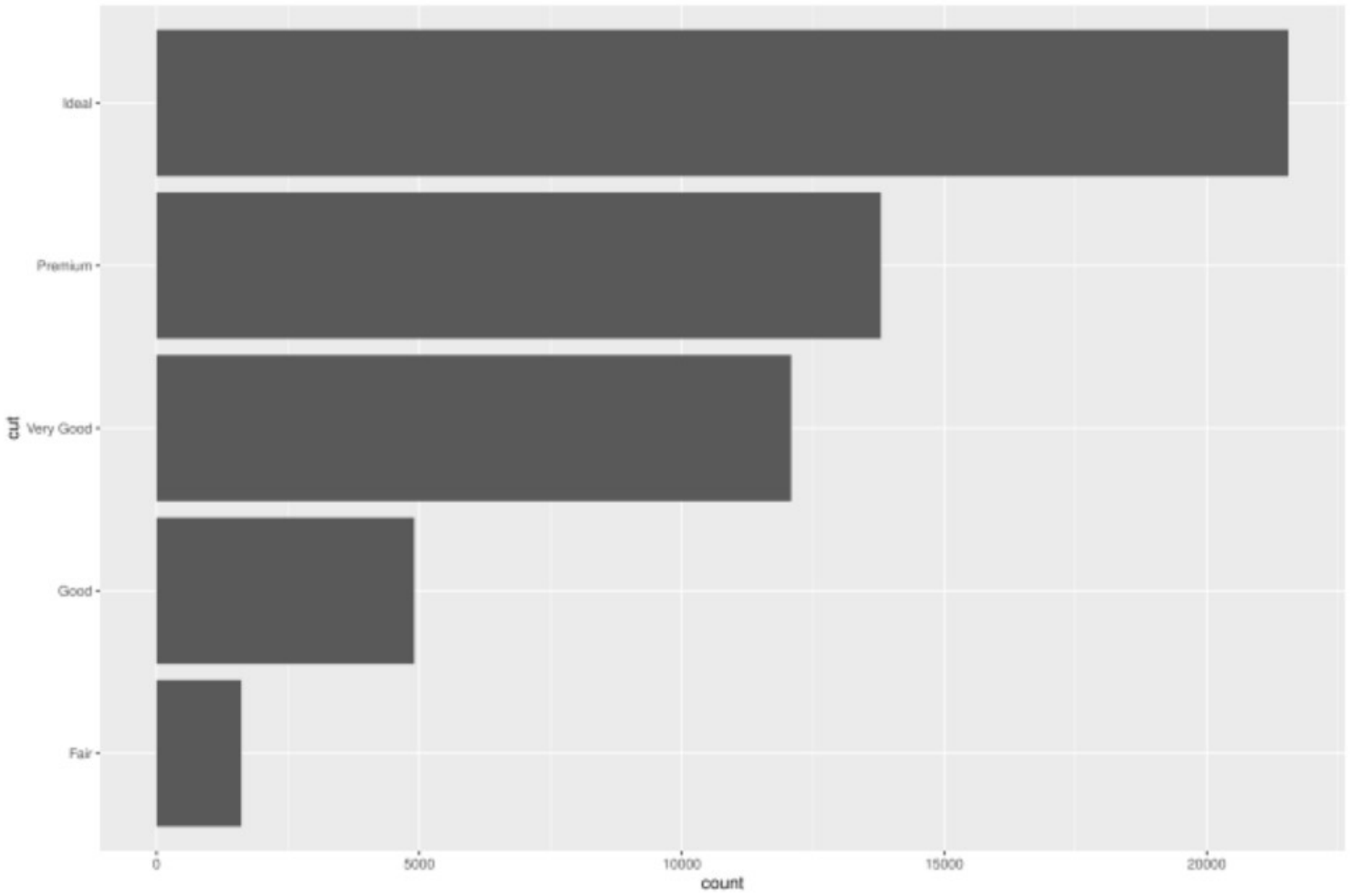
Each geom_ function has a default stat_ function and *vice versa*. The pair generally can be used interchangeably.

For instance

- geom_bar default to stat_count, as the code block below shows

```
ggplot(data = diamonds, mapping = aes(x = cut)) +
stat_count() + coord_flip()
```

DATA 100,Week 2 (B)



DATA 100,Week 2 (B)

There are two types of bar charts: geom_bar() and geom_col().

Can also specify a *non*-default stat_ function in a geom_ if have to

- The following does the transformation to get a dataframe with count as a variable
- … basically, done the aggregate statistics already
- then plot the same bar diagram using the processed data by_cut
- Notice the stat = "identity" in geom_bar
  - … meaning that the height of the bars in the diagram **is** the values in the variable (n) mapped to y
  - Do ?geom_bar in Console to see the default stat for it
- Can also use geom_col which is geom_bar with stat = "identity"

DATA 100,Week 2 (B)

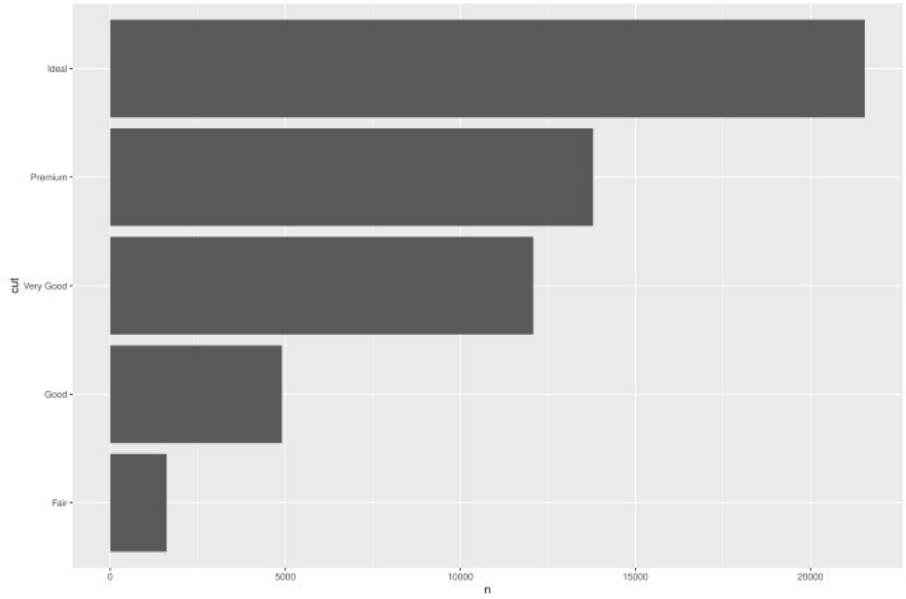
```
by_cut <- diamonds |>
group_by(cut) |>
count()

by_cut

#> # A tibble: 5 x 2
#> # Groups: cut [5]
#> cut n
#> <ord> <int>
#> 1 Fair 1610
#> 2 Good 4906
#> 3 Very Good 12082
#> 4 Premium 13791
#> 5 Ideal 21551
#by_cut
```

DATA 100,Week 2 (B)

```
ggplot(data = by_cut, mapping = aes(x = cut, y = n)) +
#geom_bar(stat = "identity") +
 geom_col() + coord_flip()
```



DATA 100,Week 2 (B)

Proportion in bar diagram

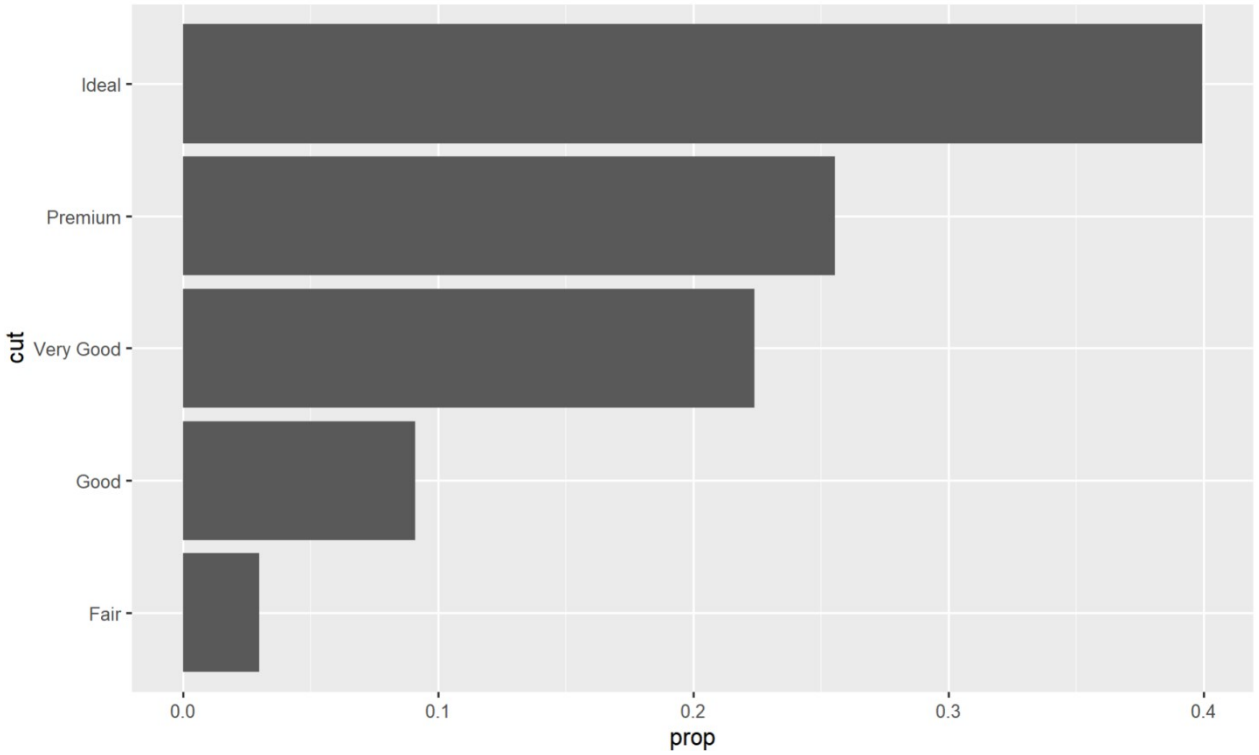
This is an example of using after_stat() function in geom_ functions, to gain access to the **computed** variables that **are not** present in the original dataset.

The parameter contained in the after_stat() function is evaluated **after stat transformation**.

Use after_stat(prop) for y to show the *proportion* of diamonds with certain cut:

DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = cut)) + geom_bar(mapping = aes(y = after_stat(prop), group = 3)) + coord_flip()
```



DATA 100,Week 2 (B)

More aesthetics details

color v.s. fill:

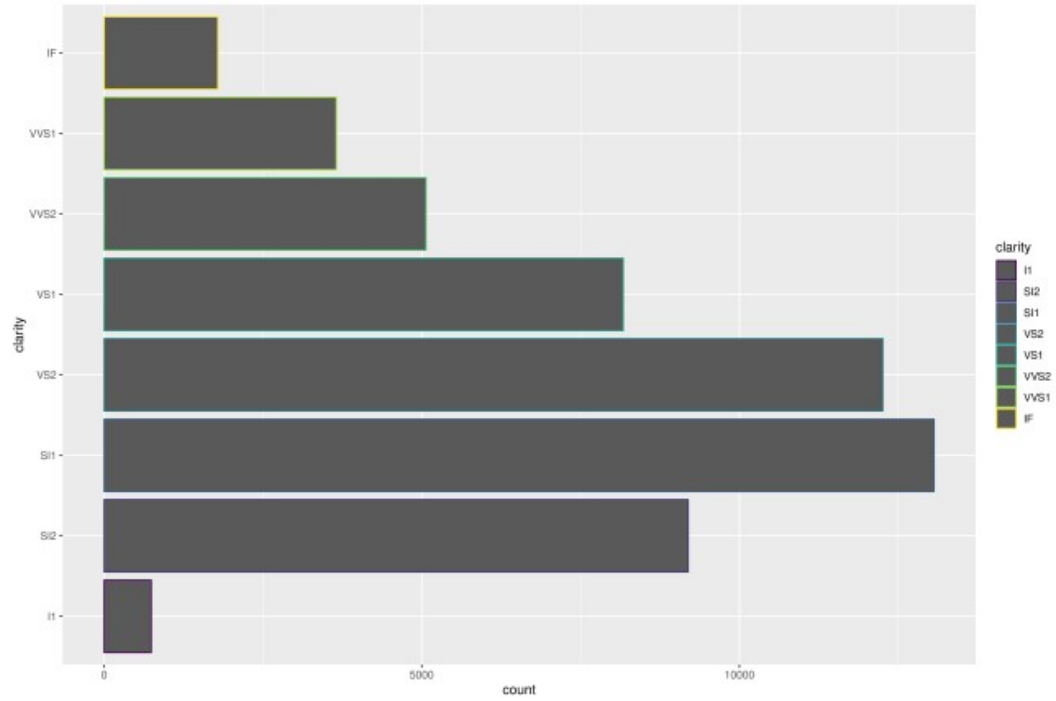
- color draws the borders in color

- fill gives color to the whole bars

They can also be used outside aes to change every bar, which overrides the corresponding item in aes

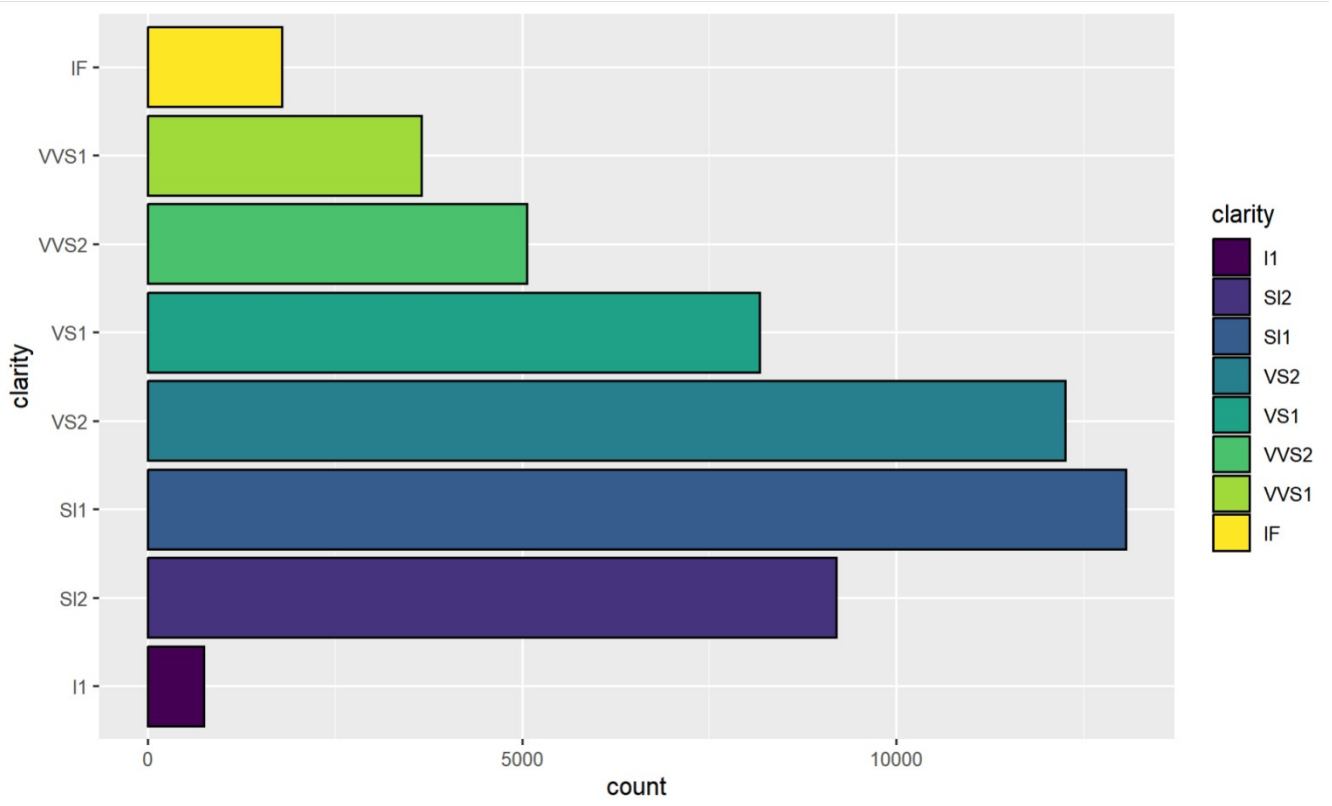
DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = clarity)) + geom_bar(mapping = aes(color = clarity)) + coord_flip()
```



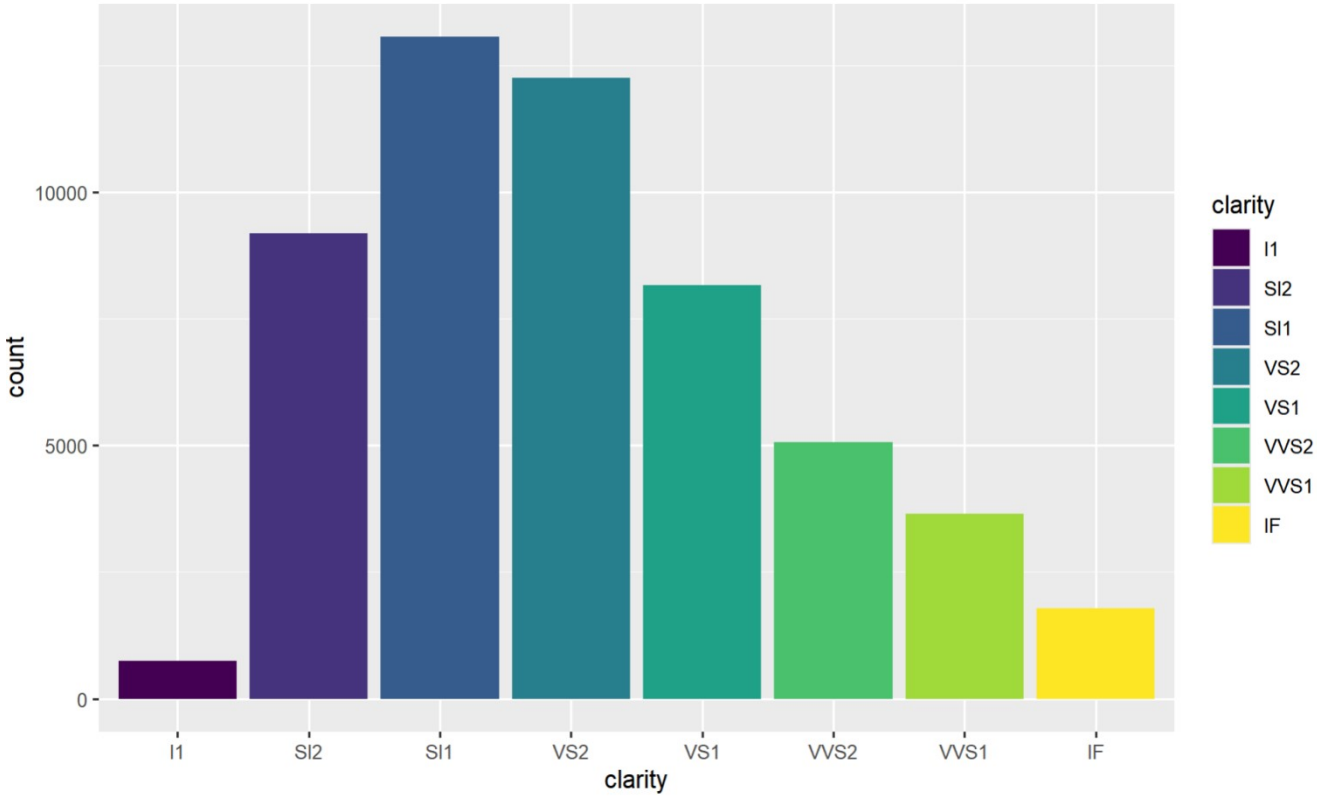
DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = clarity)) +
 #geom_bar(mapping = aes(fill = clarity)) +
 geom_bar(mapping = aes(fill = clarity), color = "black") + coord_flip()
```



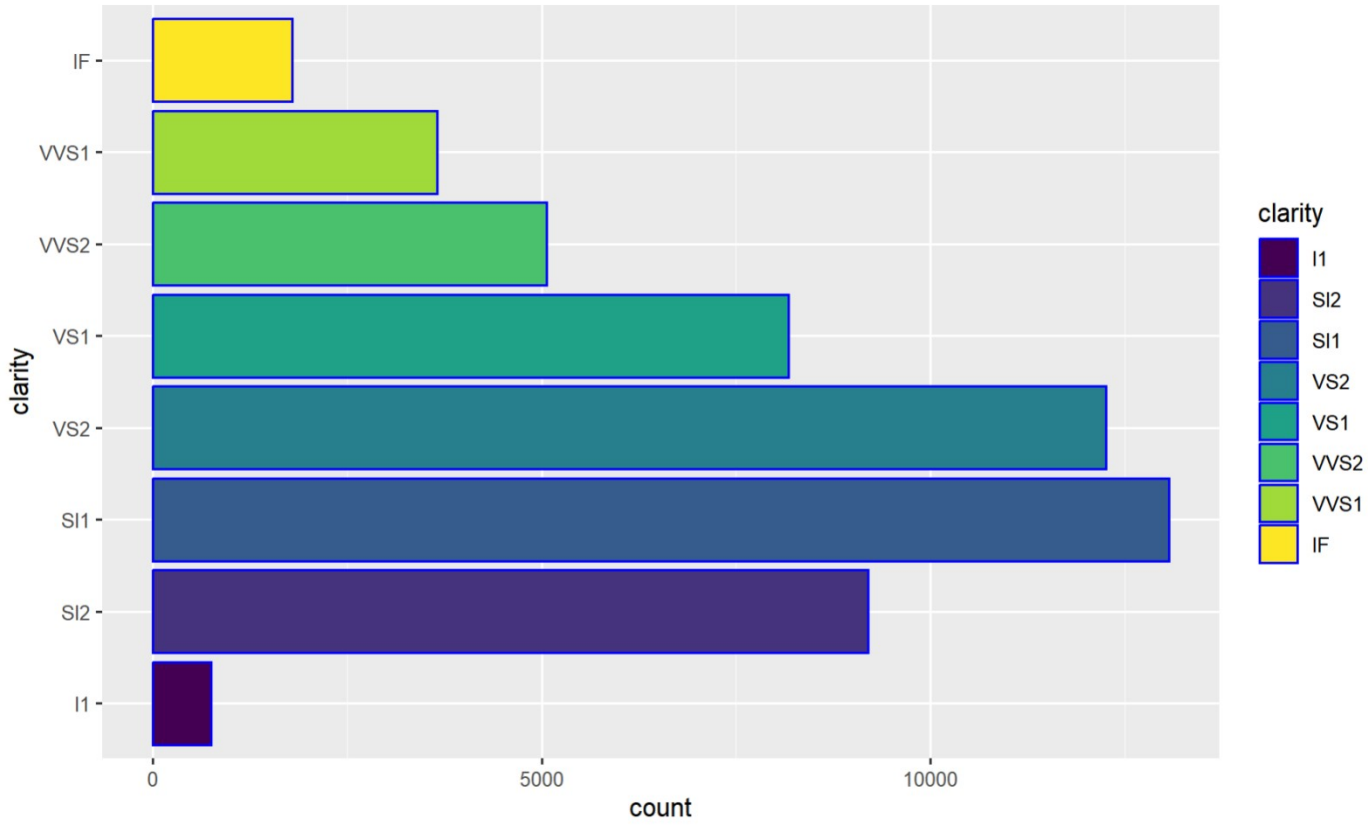
DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = clarity)) + geom_bar(mapping = aes(fill = clarity))
```



DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = clarity)) +
#geom_bar(mapping = aes(fill = clarity)) +
geom_bar(mapping = aes(fill = clarity), color = "blue") + coord_flip()
```

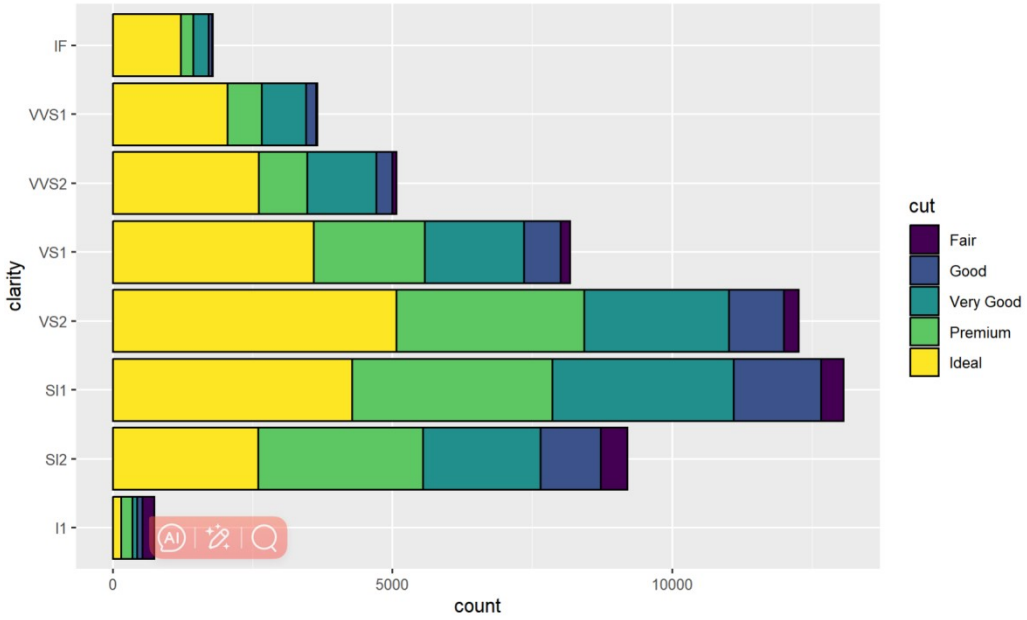


DATA 100,Week 2 (B)

Combine different variables in one bar plot

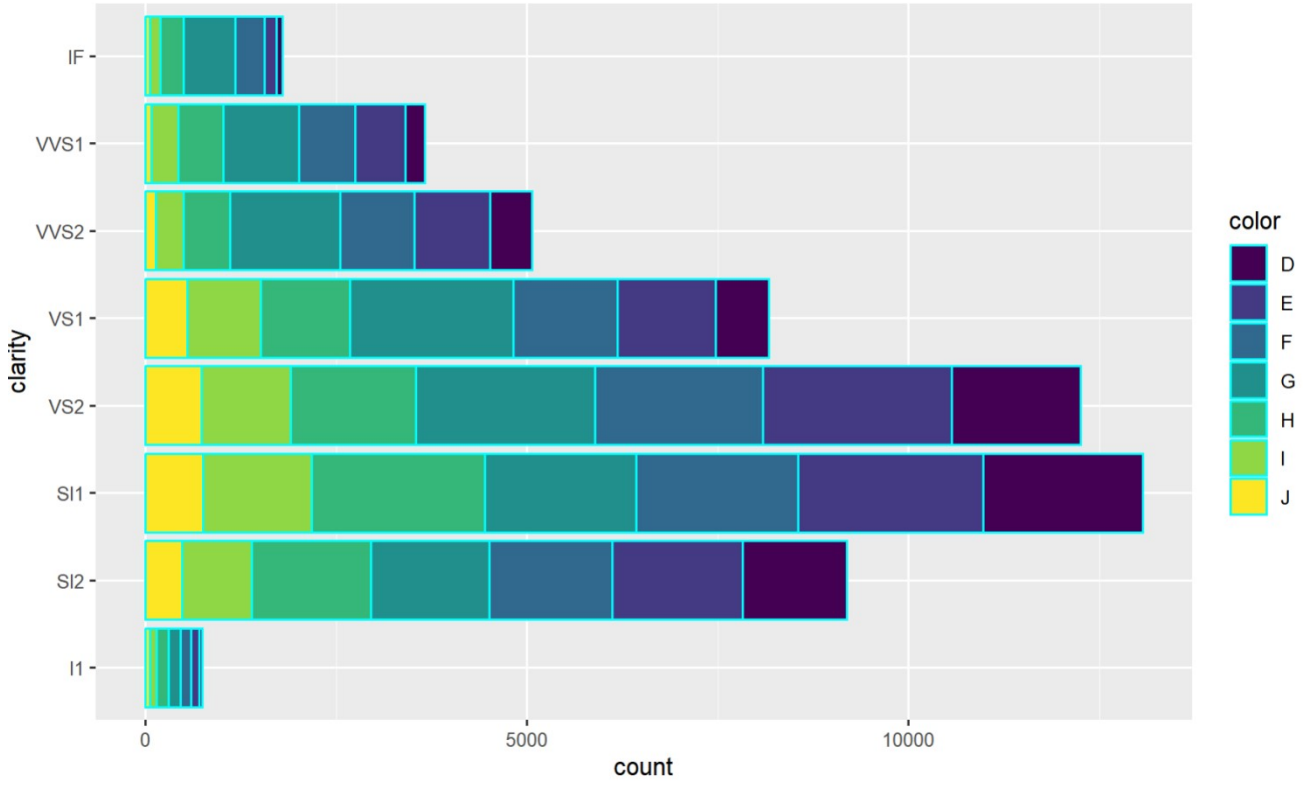
Default: stack different values of the second variable, here is cut

```
ggplot(data = diamonds, mapping = aes(x = clarity)) + geom_bar(mapping = aes(fill = cut), color = "black") + coord_flip()
```



DATA 100,Week 2 (B)

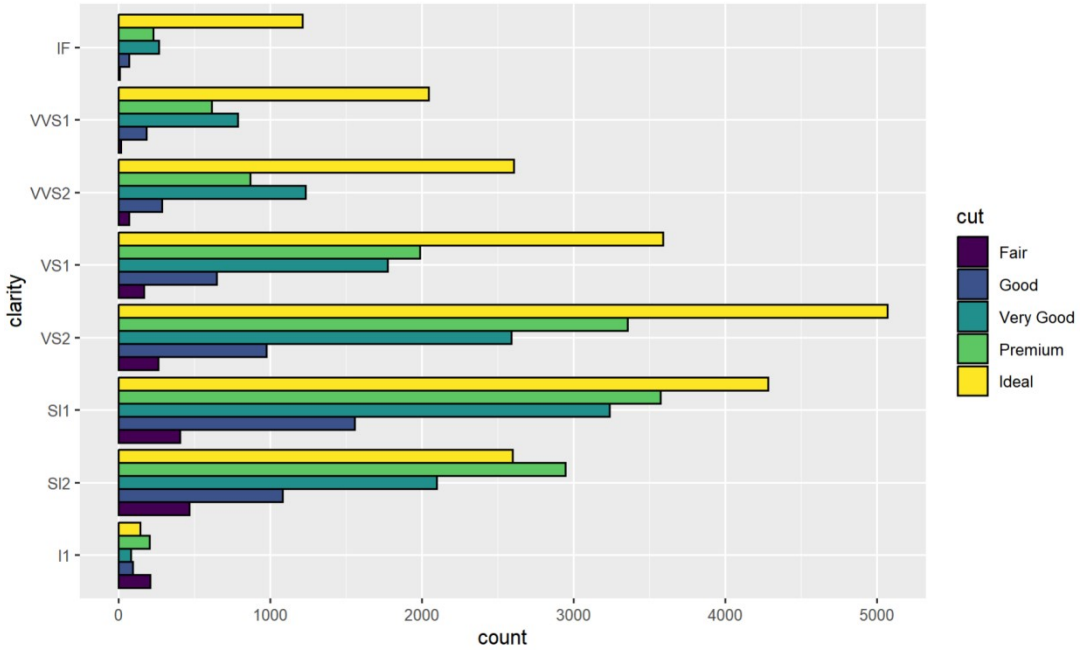
```
ggplot(data = diamonds, mapping = aes(x = clarity)) + geom_bar(mapping = aes(fill = color), color = "Cyan") + coord_flip()
```



DATA 100,Week 2 (B)

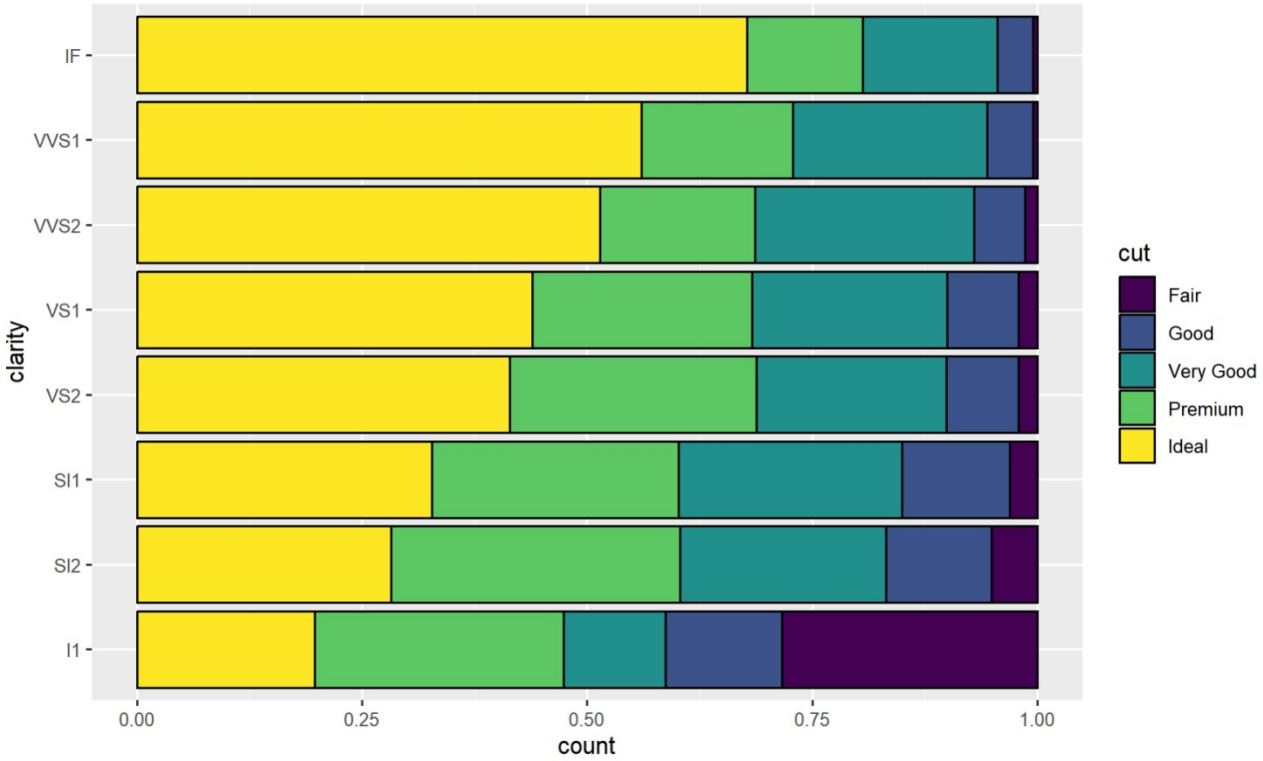
Display side-by-side

```
ggplot(data = diamonds, mapping = aes(x = clarity)) +
geom_bar(mapping = aes(fill = cut), color = "black", position = 'dodge') +
coord_flip()
```



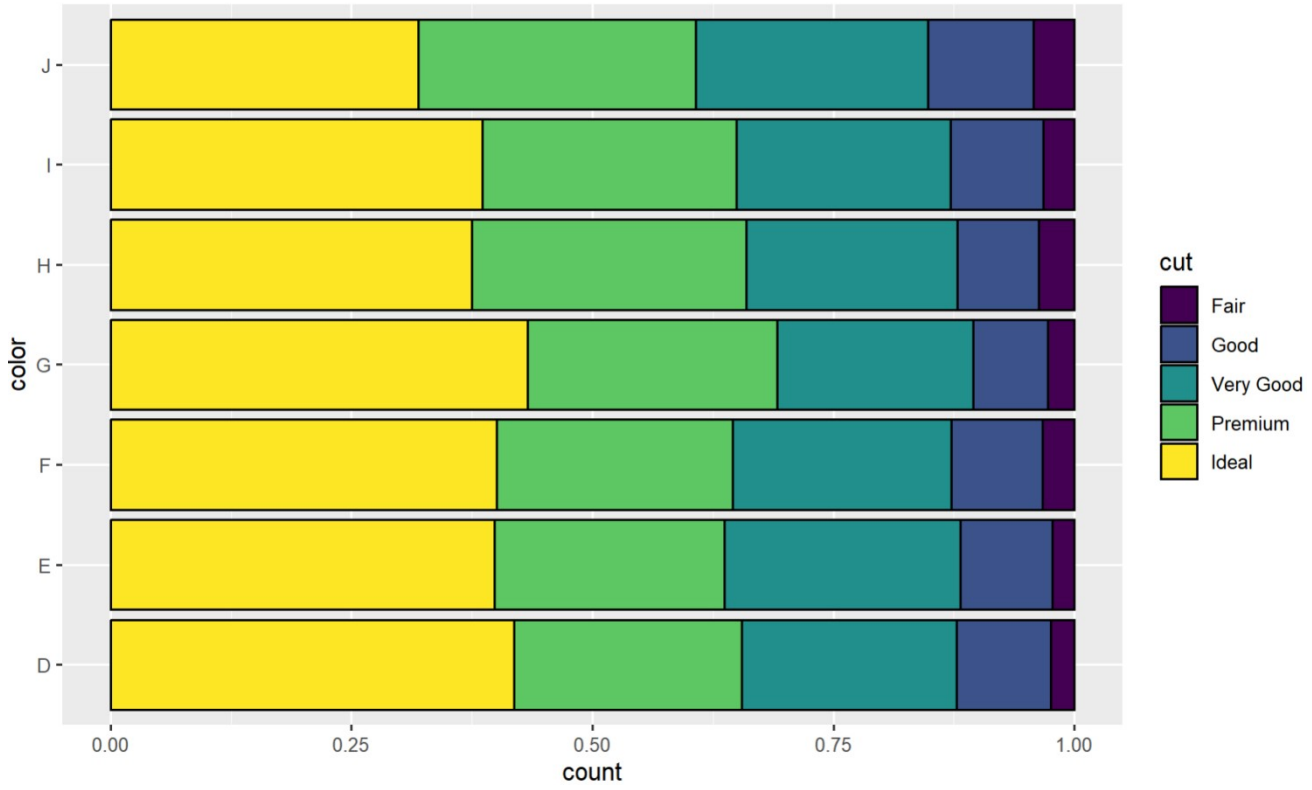
DATA 100,Week 2 (B)

```
ggplot(data = diamonds, mapping = aes(x = clarity)) +
geom_bar(mapping = aes(fill = cut), color = "black", position = "fill") +
coord_flip()
```



DATA 100,Week 2 (B)

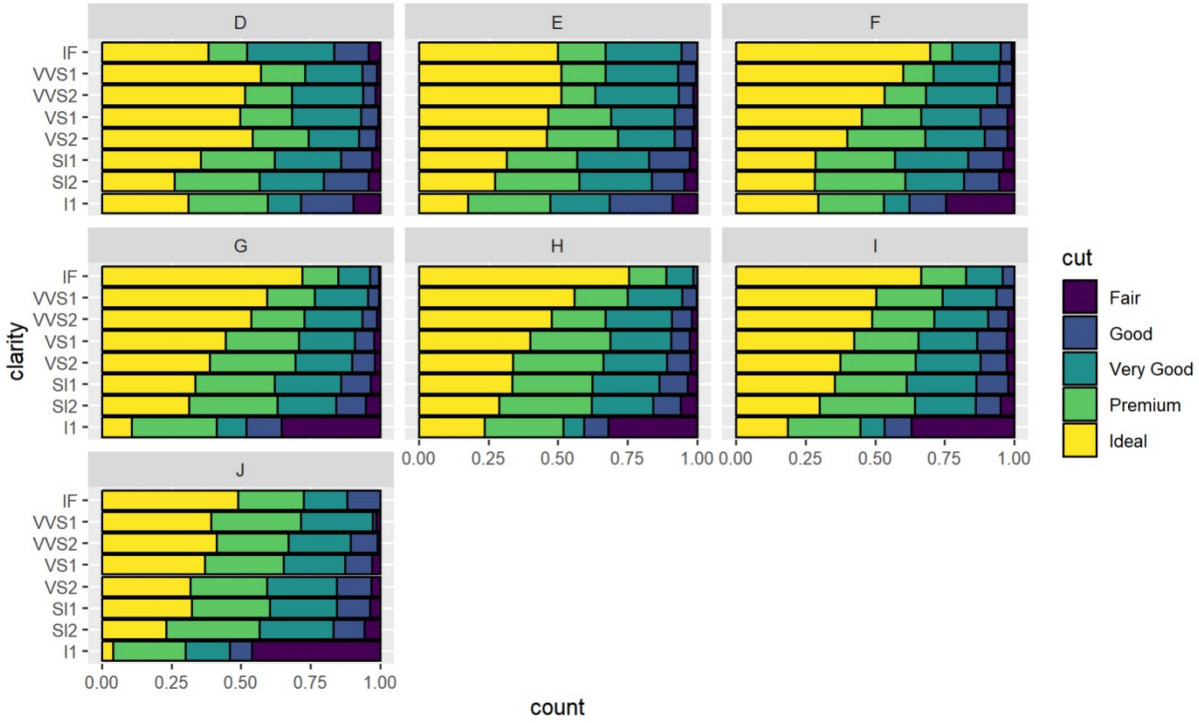
```
ggplot(data = diamonds, mapping = aes(x = color)) +
geom_bar(mapping = aes(fill = cut), color = "black", position = "fill") +
coord_flip()
```



DATA 100,Week 2 (B)

Combine with facet_

```
ggplot(data = diamonds, mapping = aes(x = clarity)) +
geom_bar(mapping = aes(fill = cut), color = "black", position = "fill") +
coord_flip() + facet_wrap(~ color)
```



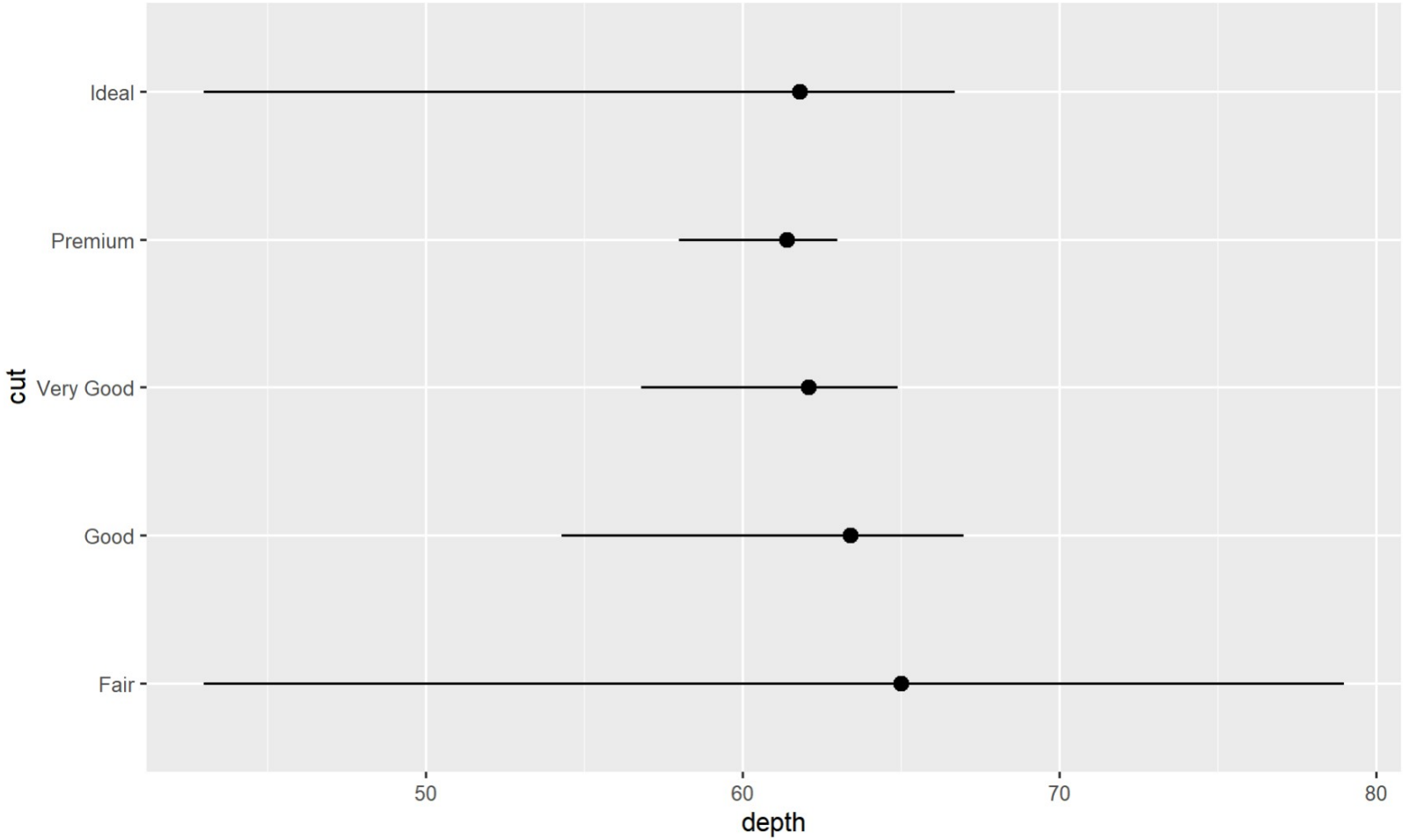
DATA 100,Week 2 (B)

Complete layered structure for ggplot

- ggplot(data = DATA) +
  – GEOM_FUNCTION(mapping = aes(MAPPINGS), stat = STAT, position = POSITION) +
  – COORDINATE_FUNCTION +
  – FACET_FUNCTION

```
ggplot(diamonds) + stat_summary(
aes(x = cut, y = depth), fun.min = min,
fun.max = max,
fun = median ) + coord_flip()
```

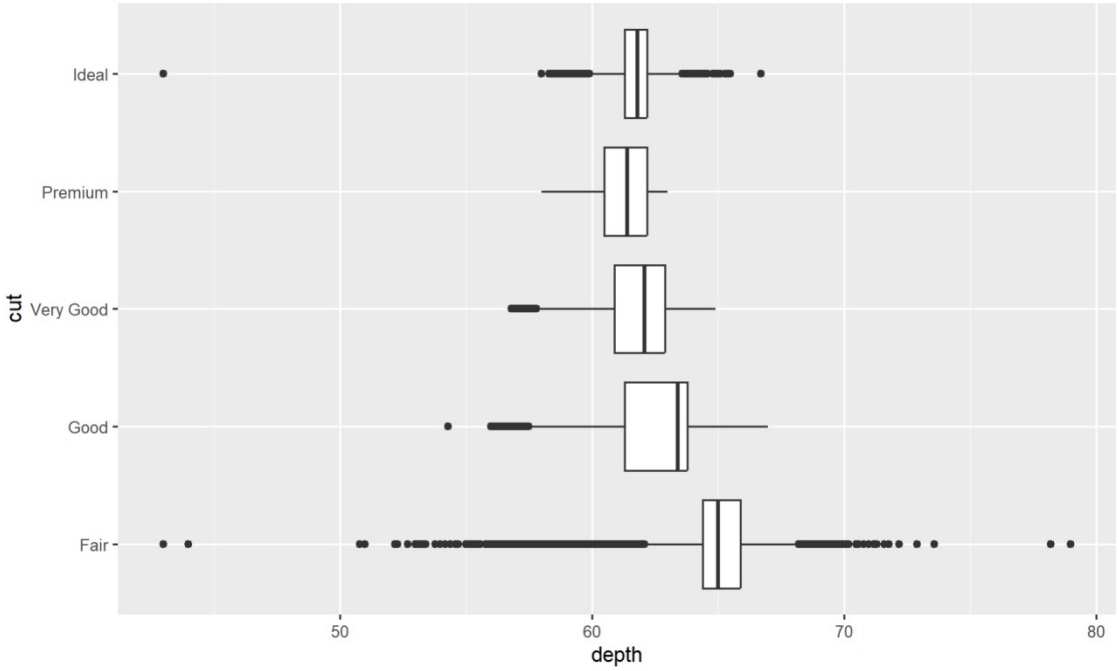
DATA 100,Week 2 (B)



DATA 100,Week 2 (B)

The full boxplot containing more information is given below. The simplicity of the previous plot sometimes could be a strength.

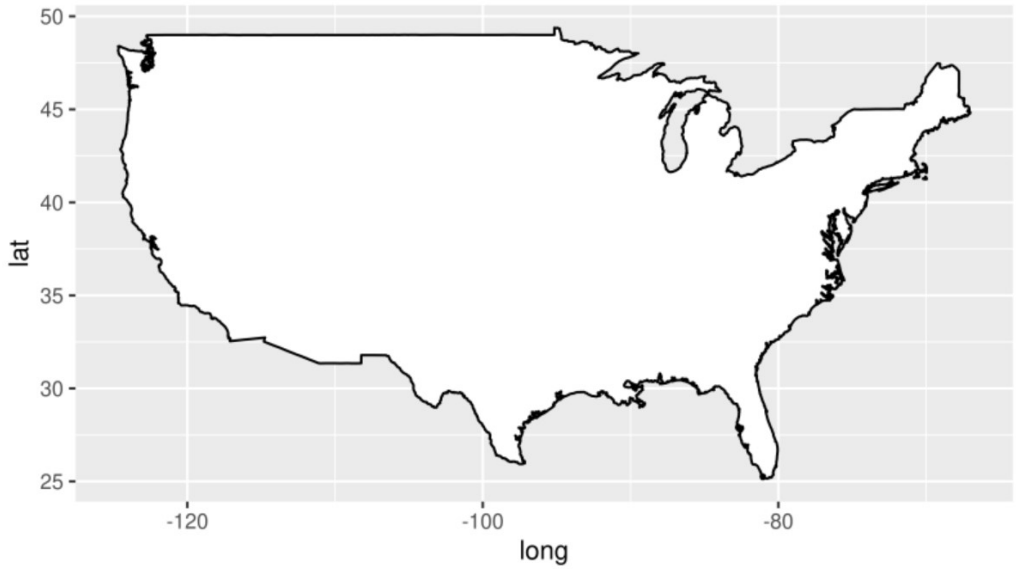
```
ggplot(data = diamonds) +
geom_boxplot(mapping = aes(x = cut, y = depth)) + coord_flip()
```



DATA 100,Week 2 (B)

Use ?map_data() to see how to use map_data

```
usa <- map_data("usa")
ggplot(usa, aes(long, lat, group = group)) + geom_polygon(fill = "white", color = "black") + coord_quickmap()
```



DATA 100,Week 2 (B)

Example

By default, **geom_bar()** will simply count the occurrences of each unique value for the x variable and use bars to display the counts.

```
#create data frame
df <- data.frame(team=rep(c('A', 'B', 'C'), each=4),
                 points=c(3, 5, 5, 6, 5, 7, 7, 8, 9, 9, 9, 8))

#view data frame

df
```

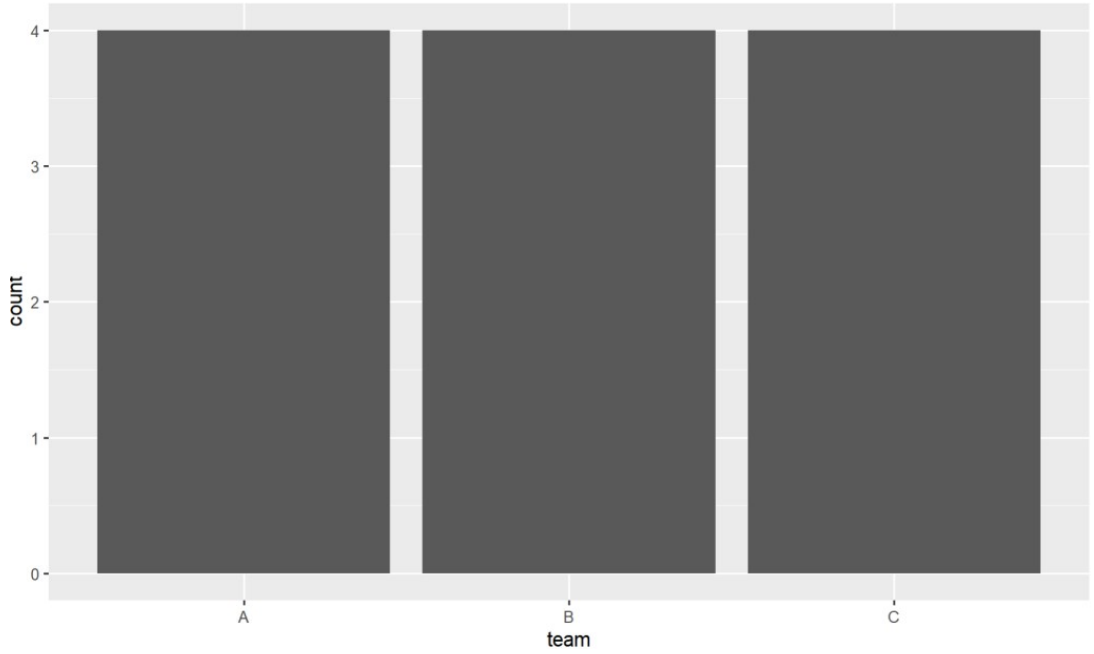
DATA 100,Week 2 (B)

	team	points
1	A	3
2	A	5
3	A	5
4	A	6
5	B	5
6	B	7
7	B	7
8	B	8
9	C	9
10	C	9
11	C	9
12	C	8

DATA 100,Week 2 (B)

library(ggplot2)

#create bar chart to visualize occurrence of each unique value in team column
ggplot(df, aes(team)) + geom_bar()



DATA 100,Week 2 (B)

The x-axis displays the unique values in the team column and the y-axis displays the number of times each unique value occurred. Since each unique value occurred 4 times, the height of each bar is 4 in the plot.

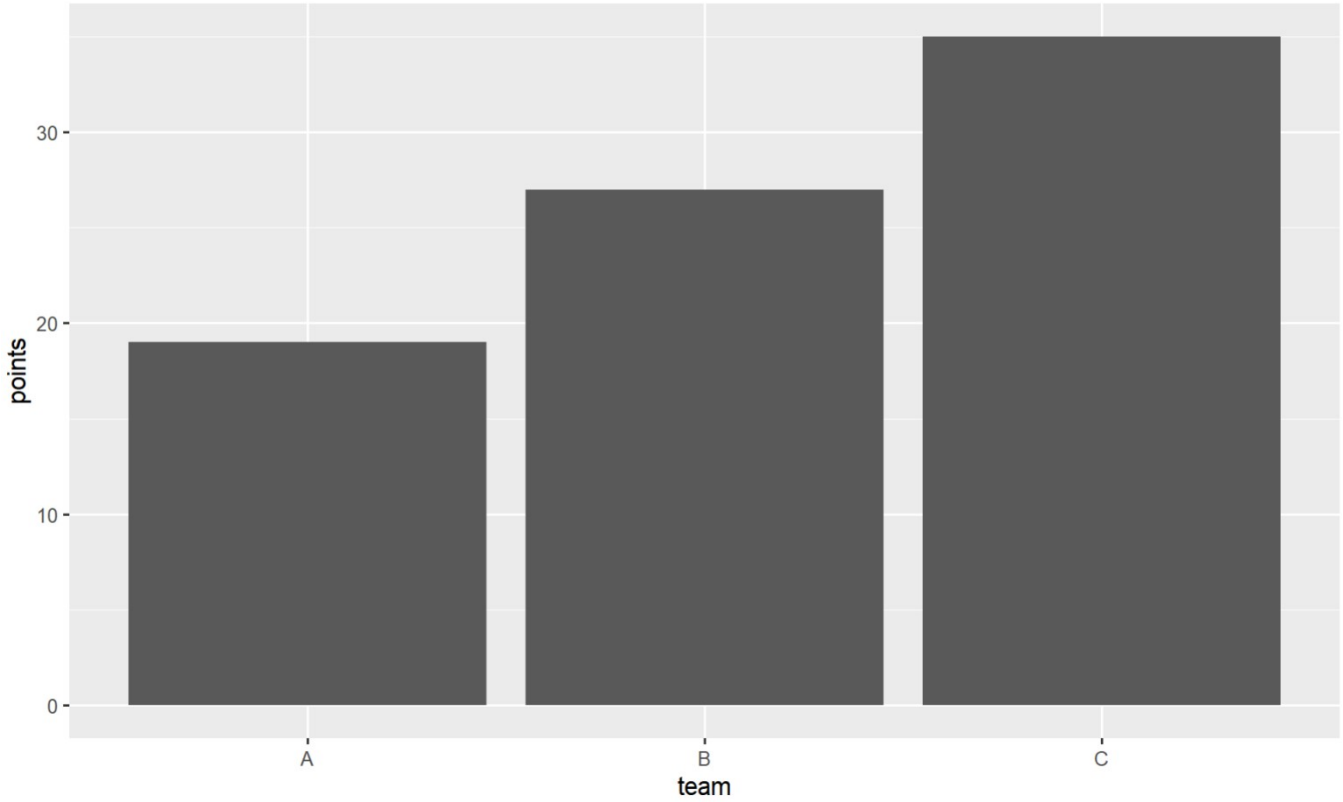
Using geom_bar(stat="identity")

The following code shows how to use the geom_bar() function with the stat="identity" argument to create a bar chart that displays the sum of values in the points column, grouped by team:

```
library(ggplot2)

#create bar chart to visualize sum of points, grouped by team
ggplot(df, aes(team, points)) +
  geom_bar(stat="identity")
```

DATA 100,Week 2 (B)



DATA 100,Week 2 (B)

The x-axis displays the unique values in the team column and the y-axis displays the sum of the values in the points column for each team.

For example:

The sum of points for team A is 19.

The sum of points for team B is 27.

The sum of points for team C is 35.

By using stat="identity" in the geom_bar() function, we're able to display the sum of values for a particular variable in our data frame instead of counts.

Note: For stat="identity" to work properly, you must provide both an x variable and a y variable in the aes() argument.

DATA 100,Week 2 (B)

dplyr basics

- ✓ dplyr transforms data so that more details can be made explicit. It also makes data easier to work with.

- ✓ A dplyr function takes a dataframe, and after processing, output an **updated dataframe**

1. The first argument is always a **dataframe**
2. The subsequent arguments indicates what is been done
3. Output another dataframe, while the input dataframe is **NOT modified**

DATA 100,Week 2 (B)

✓ Designed such that they can be *chained* together one after another
— using pipe |>.

✓ The primary functions (or verbs) are discussed here:

• For rows:
– filter: concerning observations / rows
– arrange: reorder rows based on values of some variables
– distinct: taking only rows that are distinct (in values of some collection of variables)

• For columns:
– mutate: change/add variables involved
– select: choose/discard variables
– rename: change the names of variables
– relocate: reorder variables

DATA 100,Week 2 (B)

• Groups:
– summarize: do some stats in backstage
– group_by: like facetting, useful in getting group-wise information

Dataset to use

All flights departing from New York city in 2013, from US Bureau of Transportation Statistics.

```
# already installed
library(nycflights13)
# ---- The next command takes a (relatively) long time ---- # view(flights)
# ---- The CAPITALIZED version works only in RStudio and is faster ---- #
View(flights)
# ---- glimpse works everywhere but not as pretty ----
glimpse(flights)
```

The flights data consists of a collection of *time series* data, labeled by time and dates.

DATA 100,Week 2 (B)

Data types

The most up-to-date list of data types used in tidyverse can be found online or via

```
vignette("types")
```

In particular, the column header fctr is now fct, which is abbreviation of factor, representing a categorical type

- more about factor later

Manipulating data is based on elementary operations on the different data types

DATA 100,Week 2 (B)

Logical operations

• *and*: is represented by &

• *or*: is represented by |

• *not*: is represented by !

They combine simple conditions to form complicated criteria, as those you could see in *advanced* search features on some sites:

• Books with subject on biology, not in French, by an author whose last name is either Weber or Schwarz

DATA 100,Week 2 (B)

– (type == "book") & ("biology" %in% subject) & (language != "French") &
((last_name == "Weber")
| (last_name == "Schwarz"))

Comparisons

• ==, != : coinciding (equals to), not coinciding (not equals to)

• <, > : less than, more than

• <=, >= : less than or equals to (no more than), more than or equals to (no less than)

DATA 100,Week 2 (B)

Belonging

The operator %in% returns TRUE if the thing on its left side is an element of the thing on its right side. Hence

```
3 %in% -1:9 # returns TRUE
#> [1] TRUE
3 %in% 4:10 # returns FALSE
#> [1] FALSE
```

Recall that : Creates a sequence of numbers as a vector

```
-1:9
#> [1] -1 0 1 2 3 4 5 6 7 8 9
# same as
# c(-1:9), or c(-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

DATA 100,Week 2 (B)

```
# +, -, *, / as usual
# %/% and %% deal with modular arithmetic
# Note that 3250 = 60 * 54 + 10
3250 %/% 60 # integral quotient
#> [1] 54
3250 %% 60 # remainder
#> [1] 10

# Log and exp: useful in putting data to different scale
log10(1e7) # log base 10
#> [1] 7
log2(64) # log base 2
#> [1] 6
log(100) # log base e -- natural logarithm #> [1] 4.60517
exp(1) # base of natural logarithm #> [1] 2.718282
# and other mathematical functions
```

DATA 100,Week 2 (B)

Other interesting values related to numeric computation

```
3/0 # infinity
 #> [1] Inf

sqrt(-1) # Not a Number
 #> [1] NaN

Warning message:
#> Warning in sqrt(-1): NaNs produced
```

DATA 100,Week 2 (B)

Generally numbers are represented as approximate values inside a computer. Sometimes exact equality might fail, somewhat unexpectedly.

```
1.1 * 1 == 1.1
#> [1] TRUE
1/25 * 25 == 1
#> [1] TRUE
sqrt(2)^2 == 2 #> [1] FALSE
1/49 * 49 == 1 #> [1] FALSE
```

DATA 100,Week 2 (B)

Approximation can be checked using the near function (provided by dplyr). It kind of make sense, because almost nothing in real life can be exactly measured anyway

```
near(sqrt(2)^2, 2)
#> [1] TRUE

near(1/49*49, 1)
#> [1] TRUE
```

DATA 100,Week 2 (B)

Missing value NA

Many reasons for allowing missing values in a dataframe

• Mistakes happen

– When there are many variables under consideration, missing a few in an observation is better than throwing away the whole observation

• Certain columns may not make sense for a particular row

DATA 100,Week 2 (B)

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

NA | TRUE
#> [1] TRUE

! NA
#> [1] NA

NA + 1
#> [1] NA

NA * 0
```

DATA 100,Week 2 (B)

```
#> [1] NA


NA == NA # This output in fact does make sense
 #> [1] NA

NA^0 # This output does not quite make mathematical sense
#> [1] 1
```

DATA 100,Week 2 (B)

Use is.na() to determine if a value is NA

```
is.na(NA)
```

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