
Dayna's cat


Lucy's pupper Teddy

Summarizing data with `dplyr`

Data Science for Biologists

Spring 2021, Dr. Spielman

First, we look at the data

```
diamonds
```

```
## # A tibble: 500 x 5
##    carat cut       color price     x
##    <dbl> <ord>     <ord> <int> <dbl>
##  1  0.23 Very Good E       402  4.01
##  2  0.72 Ideal     F      2879  5.76
##  3  1.11 Ideal     F      7823  6.62
##  4  1.01 Very Good D      4338  6.43
##  5  0.3  Ideal     G       684  4.31
##  6  0.23 Very Good E       485  4.02
##  7  1.51 Premium   H      9762  7.54
##  8  0.34 Premium   I       765  4.49
##  9  0.59 Ideal     E      1607  5.36
## 10  1.41 Ideal     G     11009  7.22
## # … with 490 more rows
```

`price`: Price in US dollars

`carat`: Weight of the diamond

`cut`: Quality of the diamond

`color`

`x`: width of diamond in mm

Functions ("verbs") for wrangling datasets

- Subsetting rows. `filter()`
  - Ex: Work with Premium diamonds.
  - Ex: Work with only diamonds above a certain carat.
- Removing duplicate rows `distinct()`

- Creating new columns `mutate()`
- Rearranging, removing, or keeping only certain columns `select()`
- Renaming columns `rename()`

- Arrange the data based on a column `arrange()`
  - Ex: Arrange in order of price

- Summarizing data `summarize()`
  - Ex: Calculating the mean price
  - Ex: Calculating the mean price for *each* diamond quality category

Summary statistics: From many values to *one value*

```
# Get 10 random numbers from 1-1000
x <- sample(1:1000, 10)
length(x)
## [1] 10
x
##  [1] 337 349 862 946 705  17 220 465 399 304
```

```
mean(x)
## [1] 460.4
median(x)
## [1] 374
min(x)
## [1] 17
max(x)
## [1] 946
sd(x)
## [1] 291.9673
sum(x)
## [1] 4604
```

These functions do *not summarize*

> The operation is performed separately on each value in `x`

```
x
##  [1] 337 349 862 946 705  17 220 465 399 304
```

```
sqrt(x)
##  [1] 18.357560 18.681542 29.359837 30.757113 26.551836  4.123106
##  [8] 21.563859 19.974984 17.435596

log(x, 10) # log in base 10!
##  [1] 2.527630 2.542825 2.935507 2.975891 2.848189 1.230449 2.3424
##  [9] 2.600973 2.482874

exp(x) # e^x
##  [1] 2.276357e+146 3.704880e+151          Inf          Inf 1.50
##  [6]  2.415495e+07  3.505791e+95 8.849813e+201 1.920871e+173 1.06

abs(x) # absolute value. kinda boring here, all numbers are positive
##  [1] 337 349 862 946 705  17 220 465 399 304
```

The `dplyr` function `summarize()`

```
diamonds %>%
  summarize(mean_price = mean(price))
```

```
## # A tibble: 1 x 1
##   mean_price
##        <dbl>
## 1      4124.
```

The `dplyr` function `summarize()`

> It is NOT AT ALL the same as `mutate()`!!

```
diamonds %>%
  summarize(mean_price = mean(price))
```

```
## # A tibble: 1 x 1
##   mean_price
##        <dbl>
## 1      4124.
```

```
diamonds %>%
  mutate(mean_price = mean(price))
```

```
## # A tibble: 500 x 6
##    carat cut       color price     x mean_price
##    <dbl> <ord>     <ord> <int> <dbl>      <dbl>
##  1  0.23 Very Good E       402  4.01      4124.
##  2  0.72 Ideal     F      2879  5.76      4124.
##  3  1.11 Ideal     F      7823  6.62      4124.
##  4  1.01 Very Good D      4338  6.43      4124.
##  5  0.3  Ideal     G       684  4.31      4124.
##  6  0.23 Very Good E       485  4.02      4124.
##  7  1.51 Premium   H      9762  7.54      4124.
```

Don't forget to name the column!

> Otherwise, it's going to be *really annoying* to work with

```
diamonds %>%
  # Works, but ends up gnarly
  summarize(mean(price))
```

```
## # A tibble: 1 x 1
##   `mean(price)`
##           <dbl>
## 1         4124.
```

When column names contain characters like parentheses, spaces, or other weirdness, you're gonna have a bad time.

More comparison with `mutate()`

> `sqrt()` is not a summary function!

```
diamonds %>%
  mutate(price_sqrt = sqrt(price)) %>%
  select(price_sqrt, everything())
## # A tibble: 500 x 6
##    price_sqrt carat cut       color price     x
##         <dbl> <dbl> <ord>     <ord> <int> <dbl>
##  1       20.0  0.23 Very Good E       402  4.01
##  2       53.7  0.72 Ideal     F      2879  5.76
##  3       88.4  1.11 Ideal     F      7823  6.62
##  4       65.9  1.01 Very Good D      4338  6.43
##  5       26.2  0.3  Ideal     G       684  4.31
##  6       22.0  0.23 Very Good E       485  4.02
##  7       98.8  1.51 Premium   H      9762  7.54
##  8       27.7  0.34 Premium   I       765  4.49
##  9       40.1  0.59 Ideal     E      1607  5.36
## 10      105.   1.41 Ideal     G     11009  7.22
## # … with 490 more rows
```

More comparison with `mutate()`

> `sqrt()` is not a summary function!

```
diamonds %>%
  summarize(price_sqrt = sqrt(price))
## # A tibble: 500 x 1
##    price_sqrt
##         <dbl>
##  1       20.0
##  2       53.7
##  3       88.4
##  4       65.9
##  5       26.2
##  6       22.0
##  7       98.8
##  8       27.7
##  9       40.1
## 10      105.
## # … with 490 more rows
```

The lesson: `summarize()` will get rid of any columns that are not used in the calculation.

Working with grouped data

```
head(diamonds, 3) # show first three rows only, to fit on slides
## # A tibble: 3 x 5
##   carat cut       color price     x
##   <dbl> <ord>     <ord> <int> <dbl>
## 1  0.23 Very Good E       402  4.01
## 2  0.72 Ideal     F      2879  5.76
## 3  1.11 Ideal     F      7823  6.62
```

```
diamonds %>%
  group_by(cut)
## # A tibble: 500 x 5
## # Groups:   cut [5]
##    carat cut       color price     x
##    <dbl> <ord>     <ord> <int> <dbl>
##  1  0.23 Very Good E       402  4.01
##  2  0.72 Ideal     F      2879  5.76
##  3  1.11 Ideal     F      7823  6.62
##  4  1.01 Very Good D      4338  6.43
##  5  0.3  Ideal     G       684  4.31
##  6  0.23 Very Good E       485  4.02
##  7  1.51 Premium   H      9762  7.54
##  8  0.34 Premium   I       765  4.49
```

Working with grouped data

`cut` is the *grouping variable*

```
diamonds %>%
  group_by(cut) %>%
  # Perform calculation for each group separately
  summarize(mean_price = mean(price))
## # A tibble: 5 x 2
##   cut       mean_price
## * <ord>          <dbl>
## 1 Fair           4950.
## 2 Good           3933.
## 3 Very Good      4237.
## 4 Premium        4466.
## 5 Ideal          3854.
```

Notice how `cut` is retained in the output - the grouping variable was used in the summary calculation.

Working with grouped data

Both `cut` and `color` are the *grouping variables*

```
diamonds %>%
  group_by(cut, color) %>%
  # Perform calculation for each group separately
  summarize(mean_price = mean(price))
## `summarise()` has grouped output by 'cut'. You can override using
## # A tibble: 33 x 3
## # Groups:   cut [5]
##    cut   color mean_price
##    <ord> <ord>      <dbl>
##  1 Fair  D          6056.
##  2 Fair  G          5850
##  3 Fair  H          3883.
##  4 Fair  I          2129
##  5 Fair  J          5859
##  6 Good  D          4818.
##  7 Good  E          3691.
##  8 Good  F          3275.
##  9 Good  G          4121.
## 10 Good  H          5024.
```

How would we plot this data?

```
diamonds %>%
  group_by(cut) %>%
  summarize(mean_price = mean(price)) -> diamond_mean_prices

diamond_mean_prices
## # A tibble: 5 x 2
##   cut       mean_price
## * <ord>          <dbl>
## 1 Fair           4950.
## 2 Good           3933.
## 3 Very Good      4237.
## 4 Premium        4466.
## 5 Ideal          3854.
```

A barplot is a good choice:

We want `cut` on the X-axis and `mean_price` on the Y-axis. Each bar's height should be the mean price of the diamond cut.

```
ggplot(diamond_mean_prices,
       aes(x = cut,
           y = mean_price)) +
  geom_bar()
```

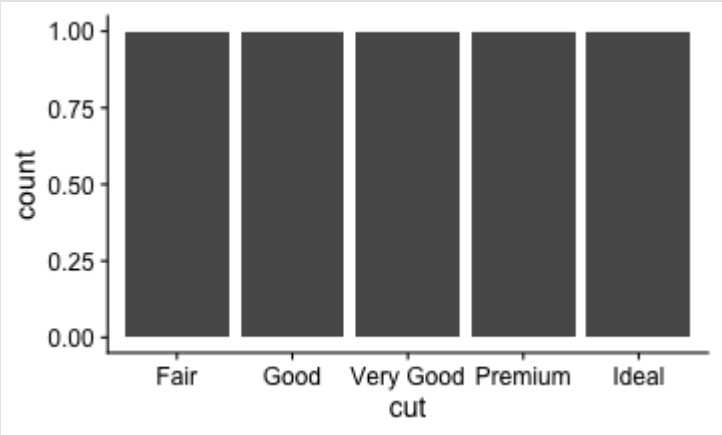
```
## Error: stat_count() can only have an x or y aesthetic.
```

UH-OH! This code gave an error!

Let's try again

`geom_bar()` **automatically determines the Y-axis as the COUNT**. So, we shouldn't provide a y-axis when using this geom.

```
ggplot(diamond_mean_prices,
       aes(x = cut)) +
  geom_bar()
```



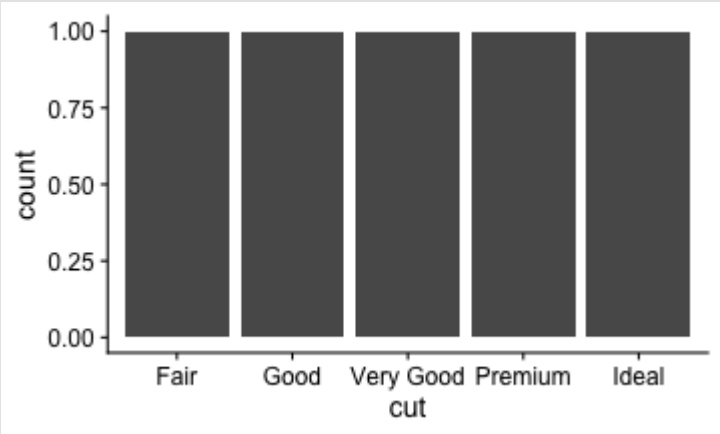
No more error, but this code isn't doing what *we want* it to do. The computer only knows what you tell it - not what you *want to tell it.*

What happened?

The only way to figure out what happened is to *look at the data*

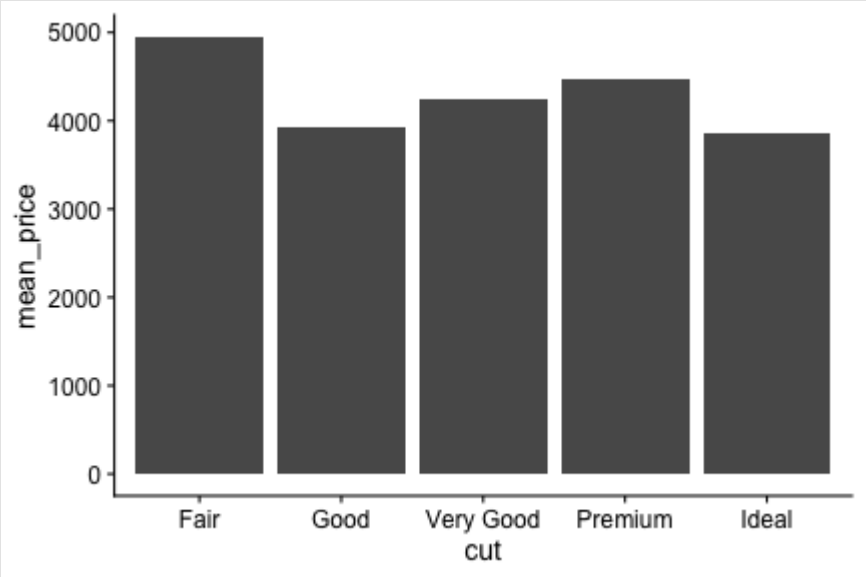
```
diamond_mean_prices
```

```
## # A tibble: 5 x 2
##   cut       mean_price
## * <ord>          <dbl>
## 1 Fair           4950.
## 2 Good           3933.
## 3 Very Good      4237.
## 4 Premium        4466.
## 5 Ideal          3854.
```



Introducing `geom_col()`

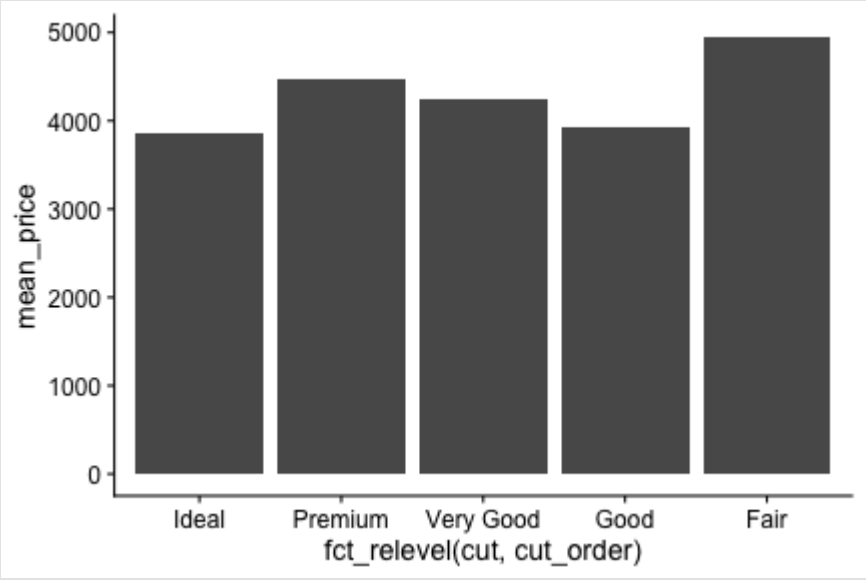
```
ggplot(diamond_mean_prices,
       aes(x = cut,
           y = mean_price)) +
  geom_col()
```



Use `geom_col()` when you want to *literally plot a VARIABLE* (not a count!) on the y-axis

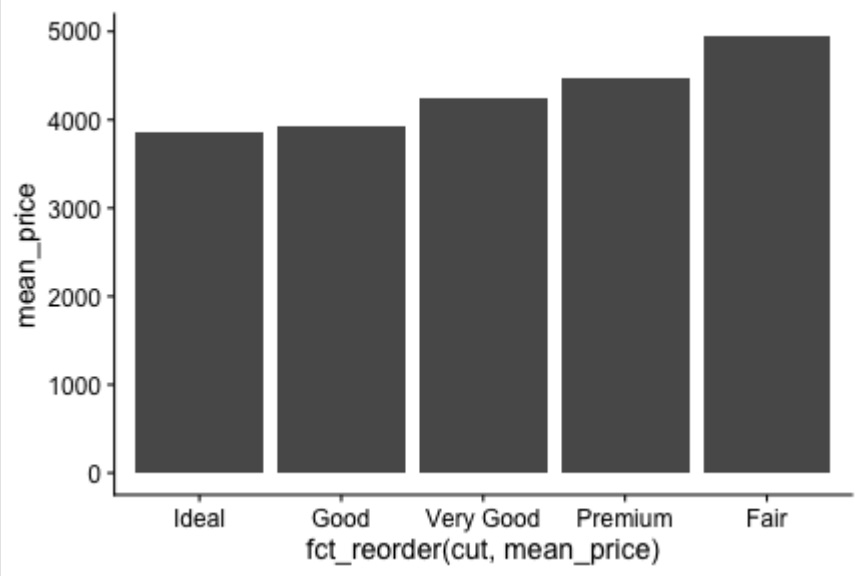
Do we remember `fct_relevel()` ???

```
cut_order <- c("Ideal", "Premium", "Very Good", "Good", "Fair")
ggplot(diamond_mean_prices,
       aes(x = fct_relevel(cut, cut_order),
           y = mean_price)) +
  geom_col()
```



Introducting `fct_reorder()`

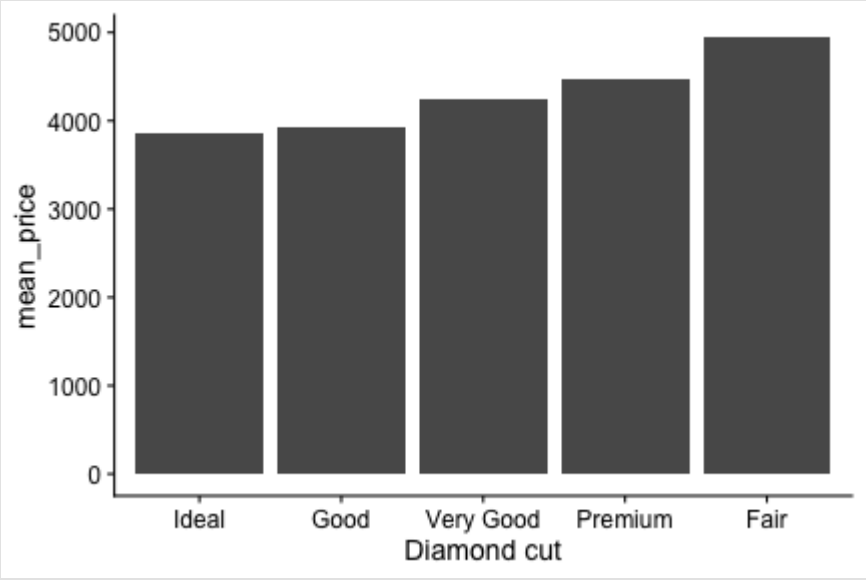
```
ggplot(diamond_mean_prices,
       aes(x = fct_reorder(cut, mean_price),
           y = mean_price)) +
  geom_col()
```



Automatically reorder a factor *based on values of another variable*. Magic.

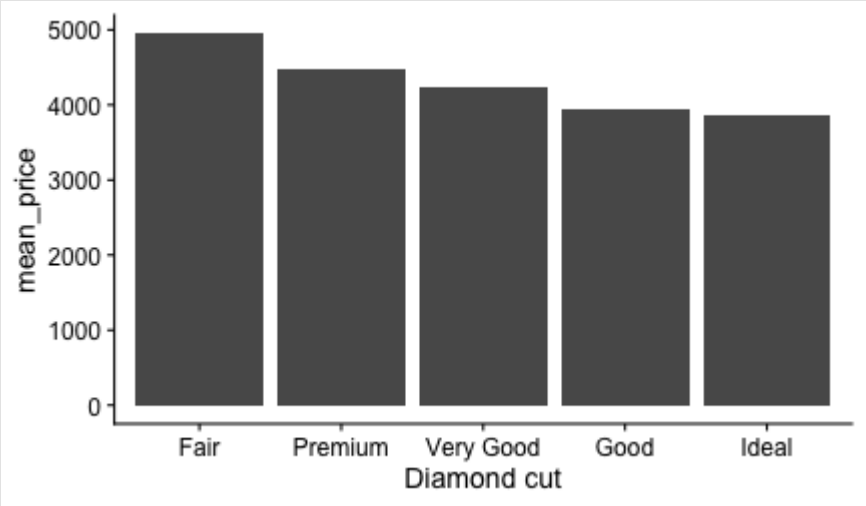
Introducting `fct_reorder()`

```
ggplot(diamond_mean_prices,
       aes(x = fct_reorder(cut, mean_price),
           y = mean_price)) +
  geom_col() +
  labs(x = "Diamond cut")
```



Introducting `fct_reorder()`

```
ggplot(diamond_mean_prices,
       aes(x = fct_reorder(cut, mean_price, .desc=TRUE),
           y = mean_price)) +
  geom_col() +
  labs(x = "Diamond cut")
```



Use the `fct_reorder()` argument `.desc=TRUE` for descending order

Helpful `dplyr` functions for summarizing and/or grouping

- `ungroup()`: Remove groupings from data. No arguments needed
  - If grouping is not the last step in your pipeline, ALWAYS ALWAYS **ALWAYS**
- `tally()`: Count all observations in a group
- `n()`: Acts as a *variable* indicating the number of observations in a group
- `count()`: Super handy shortcut for `group_by()` %>% `tally()` %>% `ungroup()`

Don't ever forget to `ungroup()`

This is absolutely critical. Build the habit now, even if it sometimes seems silly.

```
diamonds %>%
  group_by(cut, color) %>%
  summarize(median_x = median(x)) %>%
  ungroup()
## # A tibble: 33 x 3
##    cut   color median_x
##    <ord> <ord>    <dbl>
##  1 Fair  D         6.27
##  2 Fair  G         6.20
##  3 Fair  H         6.35
##  4 Fair  I         6.05
##  5 Fair  J         7.43
##  6 Good  D         5.16
##  7 Good  E         5.47
##  8 Good  F         5.67
##  9 Good  G         6.28
## 10 Good  H         6.38
```

Ungrouping matters

```
diamonds_tiny
```

```
## # A tibble: 4 x 3
##   expensive price carat
##   <chr>     <int> <dbl>
## 1 no          844  0.3
## 2 no          863  0.41
## 3 yes       10291  1.5
## 4 yes       11379  2.03
```

Ungrouping matters

```
diamonds_tiny %>%
  group_by(expensive) %>%
  mutate(mean_price = mean(price)) %>%
  mutate(mean_carat = mean(carat))
## # A tibble: 4 x 5
## # Groups:   expensive [2]
##   expensive price carat mean_price mean_carat
##   <chr>     <int> <dbl>      <dbl>      <dbl>
## 1 no          844  0.3        854.      0.355
## 2 no          863  0.41       854.      0.355
## 3 yes       10291  1.5      10835       1.76
## 4 yes       11379  2.03     10835       1.76
```

Ungrouping matters

```
diamonds_tiny %>%
  group_by(expensive) %>%
  mutate(mean_price = mean(price)) %>%
  ungroup() %>%
  mutate(mean_carat = mean(carat))
## # A tibble: 4 x 5
##   expensive price carat mean_price mean_carat
##   <chr>     <int> <dbl>      <dbl>      <dbl>
## 1 no          844  0.3        854.       1.06
## 2 no          863  0.41       854.       1.06
## 3 yes       10291  1.5      10835        1.06
## 4 yes       11379  2.03     10835        1.06
```

Neither of those versions is "right" or "wrong"

Instead, it depends which calculation you actually wanted to calculate!

Count observations (rows) with `tally()`

```
# No grouping - counts all rows
diamonds %>%
  tally()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   500
```

```
diamonds %>%
  group_by(cut) %>%
  tally() %>%
  ungroup()
```

```
## # A tibble: 5 x 2
##   cut           n
## * <ord>     <int>
## 1 Fair         10
## 2 Good         52
## 3 Very Good   108
## 4 Premium     128
## 5 Ideal       202
```

```
# Change `n` with argument "name
diamonds %>%
  group_by(cut) %>%
  tally(name = "count_cut") %>%
  ungroup()
```

```
## # A tibble: 5 x 2
##   cut       count_cut
## * <ord>         <int>
## 1 Fair             10
## 2 Good             52
## 3 Very Good       108
## 4 Premium         128
## 5 Ideal           202
```

`tally()` returns a tibble with a column (named `n` by default) representing the number of rows

Magically count with `count()`

```
diamonds %>%
  group_by(cut) %>%
  tally() %>%
  ungroup()
```

```
## # A tibble: 5 x 2
##   cut           n
## * <ord>     <int>
## 1 Fair         10
## 2 Good         52
## 3 Very Good   108
## 4 Premium     128
## 5 Ideal       202
```

`count()` is a handy shortcut that groups, tallies, and ungroups for you!

```
diamonds %>%
  count(cut)
```

```
## # A tibble: 5 x 2
##   cut           n
## * <ord>     <int>
## 1 Fair         10
## 2 Good         52
## 3 Very Good   108
## 4 Premium     128
## 5 Ideal       202
```

```
diamonds %>%
  count(cut, color)
```

```
## # A tibble: 33 x 3
##    cut   color     n
##    <ord> <ord> <int>
##  1 Fair  D         3
##  2 Fair  G         2
##  3 Fair  H         3
##  4 Fair  I         1
##  5 Fair  J         1
##  6 Good  D         8
```

`count()` can also take the `name` argument

```
diamonds %>%
  count(cut, color)
```

```
## # A tibble: 33 x 3
##    cut   color     n
##    <ord> <ord> <int>
##  1 Fair  D         3
##  2 Fair  G         2
##  3 Fair  H         3
##  4 Fair  I         1
##  5 Fair  J         1
##  6 Good  D         8
##  7 Good  E        12
##  8 Good  F         9
##  9 Good  G         9
## 10 Good  H         6
## # … with 23 more rows
```

```
diamonds %>%
  count(cut, color,
    name = "count_cut_color")
```

```
## # A tibble: 33 x 3
##    cut   color count_cut_color
##    <ord> <ord>           <int>
##  1 Fair  D                   3
##  2 Fair  G                   2
##  3 Fair  H                   3
##  4 Fair  I                   1
##  5 Fair  J                   1
##  6 Good  D                   8
##  7 Good  E                  12
##  8 Good  F                   9
##  9 Good  G                   9
## 10 Good  H                   6
## # … with 23 more rows
```

The `n()` function

> A special function you can use *with* other `dplyr` verbs

```
diamonds %>%
  mutate(how_many_rows = n()) %>
  select(how_many_rows)
```

```
## # A tibble: 500 x 1
##    how_many_rows
##            <int>
##  1           500
##  2           500
##  3           500
##  4           500
##  5           500
##  6           500
##  7           500
##  8           500
##  9           500
## 10           500
## # … with 490 more rows
```

```
diamonds %>%
  group_by(color) %>%
  mutate(how_many_rows = n()) %>
  ungroup() %>%
  select(color, how_many_rows)
```

```
## # A tibble: 500 x 2
##    color how_many_rows
##    <ord>         <int>
##  1 E                88
##  2 F                76
##  3 F                76
##  4 D                71
##  5 G               113
##  6 E                88
##  7 H                76
##  8 I                47
##  9 E                88
## 10 G               113
## # … with 490 more rows
```