

CH1-5

2024-03-09

1: The penguins data frame

You can see all variables and the first few observations of each variable by using `glimpse()`.

```
penguins <- palmerpenguins::penguins

glimpse(penguins)

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

2: Creating a ggplot

The **mapping** argument of the `ggplot()` function defines how variables in your dataset are mapped to visual properties (*aesthetics*) of your plot. The **mapping** argument is always defined in the `aes()` function, and the **x** and **y** arguments of `aes()` specify which variables to map to the x and y axes.

geom: The geometrical object that a plot uses to represent data. These geometric objects are made available in ggplot2 with functions that start with `geom_`.

People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms (`geom_bar()`), line charts use line geoms (`geom_line()`), boxplots use boxplot geoms (`geom_boxplot()`), scatterplots use point geoms (`geom_point()`), and so on.

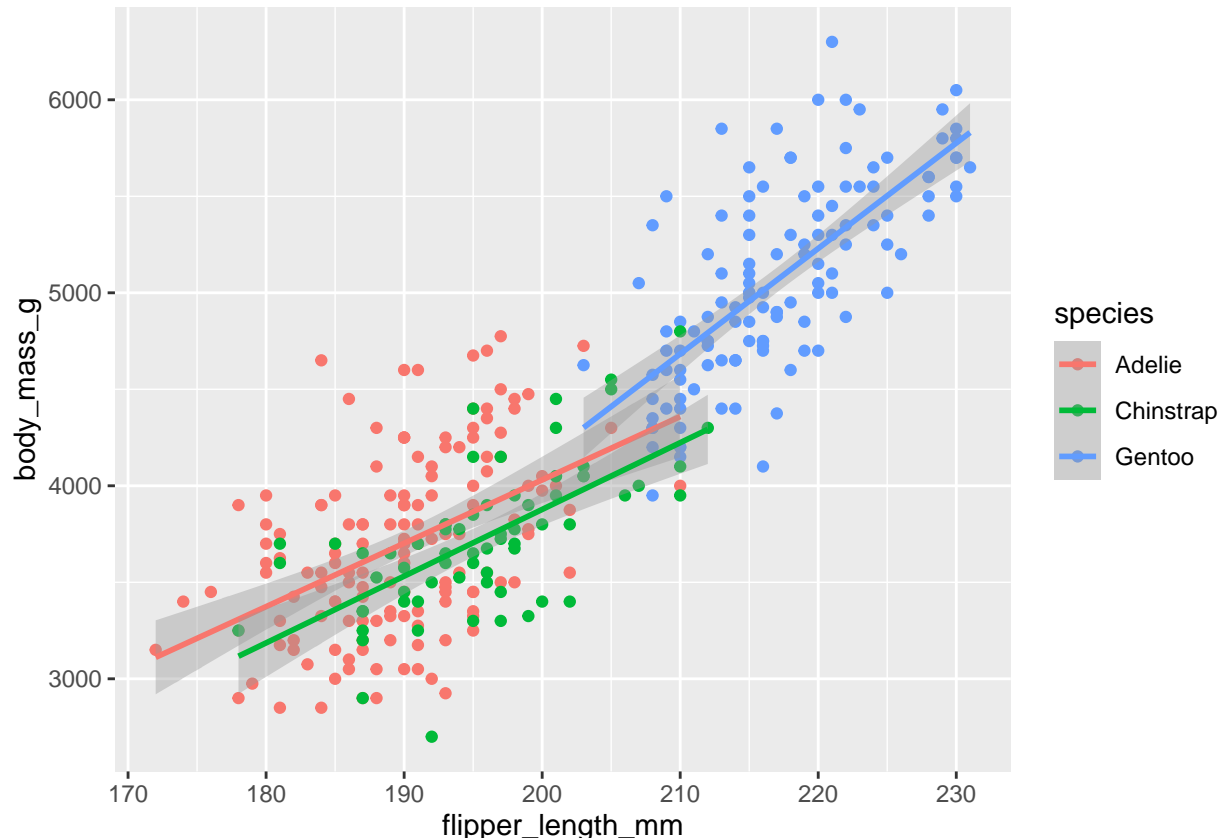
The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot.

When a categorical variable is mapped to an aesthetic, ggplot2 will automatically assign a unique value of the aesthetic (here a unique color) to each unique level of the variable (each of the three species), a process known as *scaling*. ggplot2 will also add a legend that explains which values correspond to which levels.

Now let's add one more layer: a smooth curve displaying the relationship between body mass and flipper length. Since this is a new geometric object representing our data, we will add a new geom as a layer on top of our point geom: `geom_smooth()`. And we will specify that we want to draw the line of best fit based on a linear model with `method = "lm"`.

```
ggplot(penguins,
       mapping = aes(x = flipper_length_mm,
                     y = body_mass_g,
                     color = species)) +
  geom_point() +
  geom_smooth(method = "lm")
```

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



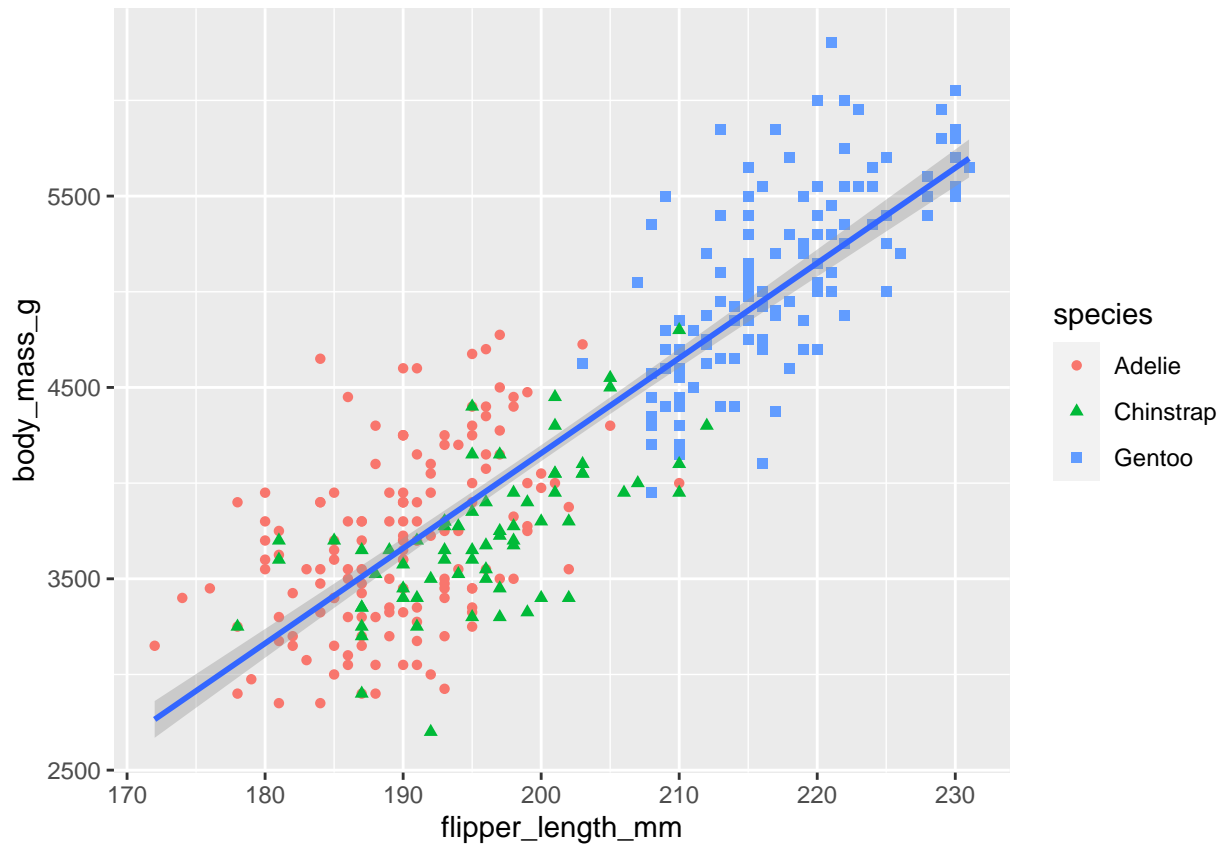
When aesthetic mappings are defined in `ggplot()`, at the global level, they're passed down to each of the subsequent geom layers of the plot. However, each geom function in `ggplot2` can also take a `mapping` argument, which allows for aesthetic mappings at the local level that are added to those inherited from the global level.

Since we want points to be colored based on species but don't want the lines to be separated out for them, we should specify `color = species` for `geom_point()` only.

It's generally not a good idea to represent information using only colors on a plot, as people perceive colors differently due to color blindness or other color vision differences. Therefore, in addition to color, we can also map `species` to the `shape` aesthetic.

```
ggplot(penguins,
       mapping = aes(x = flipper_length_mm,
                     y = body_mass_g)) +
  geom_point(mapping = aes(color = species, shape = species)) +
  geom_smooth(method = "lm")
```

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



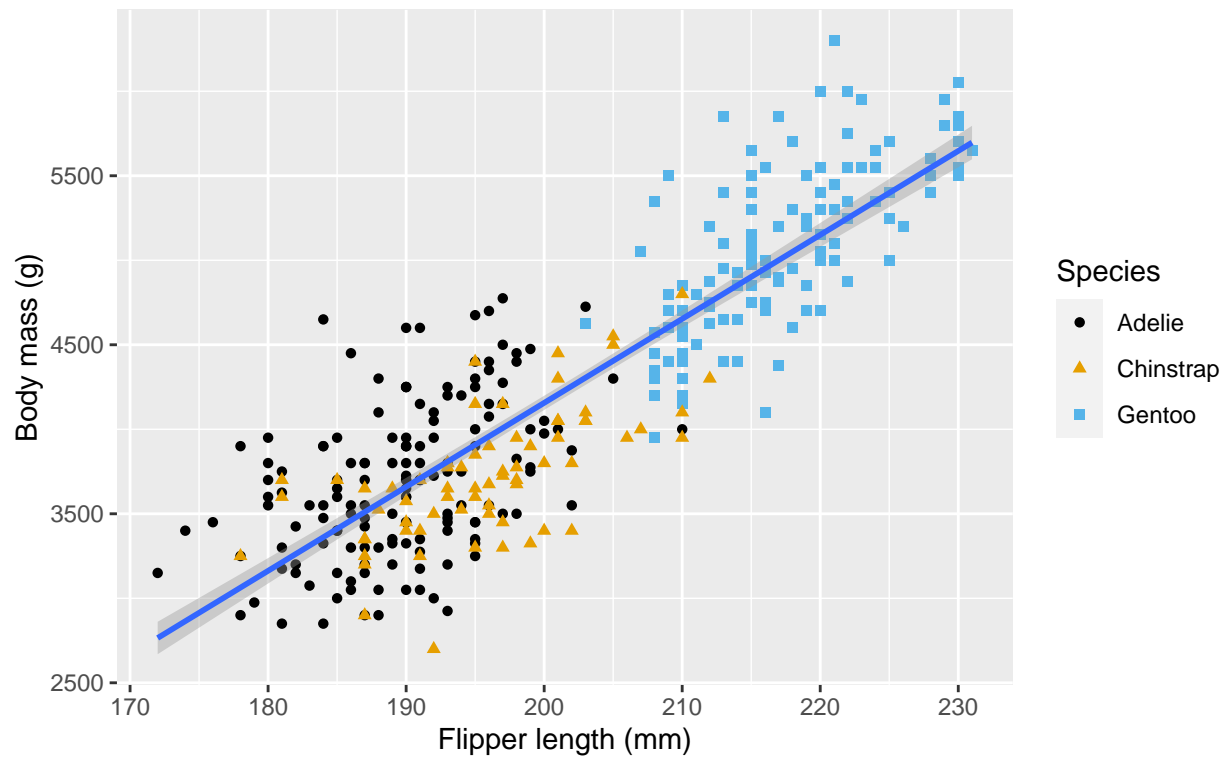
We can improve the labels of our plot using the `labs()` function in a new layer. Some of the arguments to `labs()` might be self explanatory: `title` adds a `title` and `subtitle` adds a `subtitle` to the plot. Other arguments match the aesthetic mappings, `x` is the x-axis label, `y` is the y-axis label, and `color` and `shape` define the label for the legend. In addition, we can improve the color palette to be colorblind safe with the `scale_color_colorblind()` function from the `ggthemes` package.

```
ggplot(penguins,
  mapping = aes(x = flipper_length_mm,
    y = body_mass_g)) +
  geom_point(mapping = aes(color = species, shape = species)) +
  geom_smooth(method = "lm") +
  labs(title = "Body mass and flipper length",
    subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo Penguins",
    x = "Flipper length (mm)",
    y = "Body mass (g)",
    color = "Species",
    shape = "Species") +
  scale_color_colorblind()
```

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

Body mass and flipper length

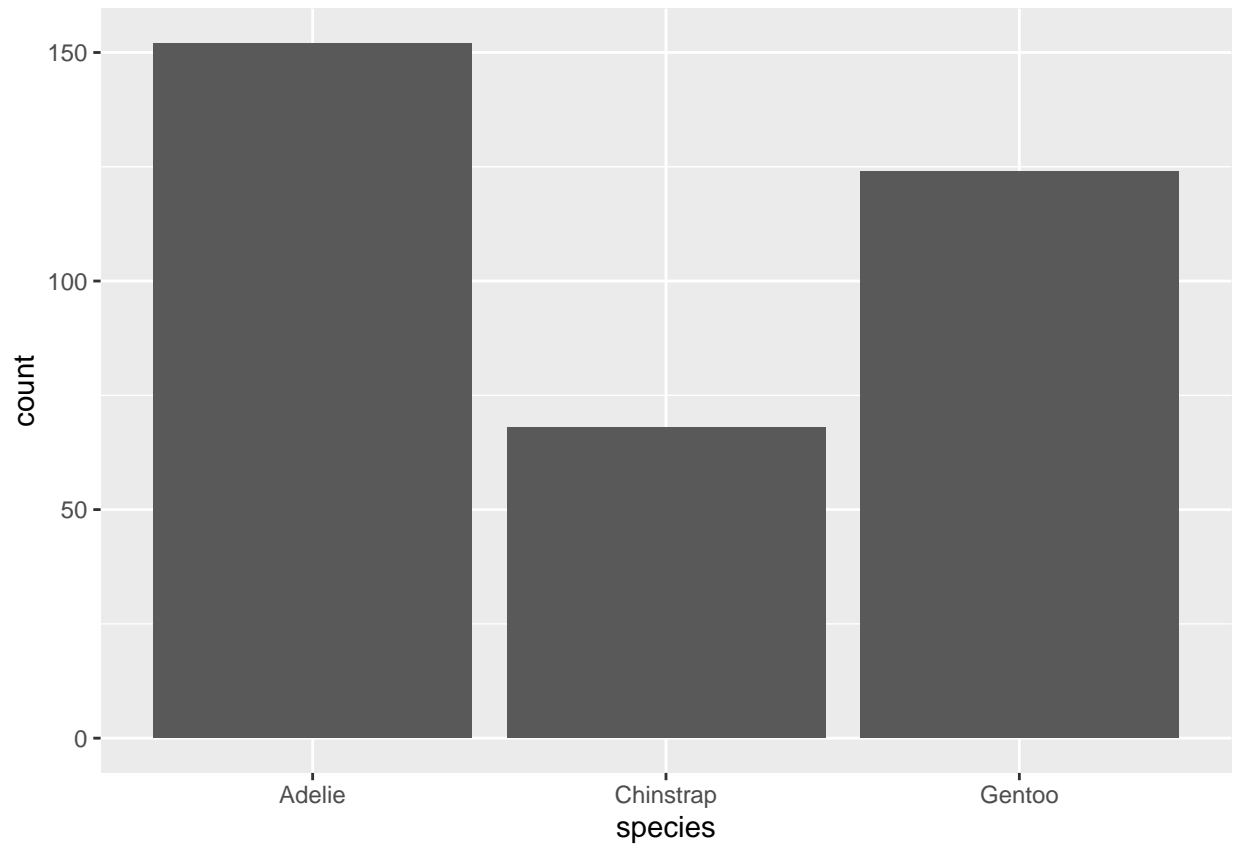
Dimensions for Adelie, Chinstrap, and Gentoo Penguins



3. Visualizing distributions

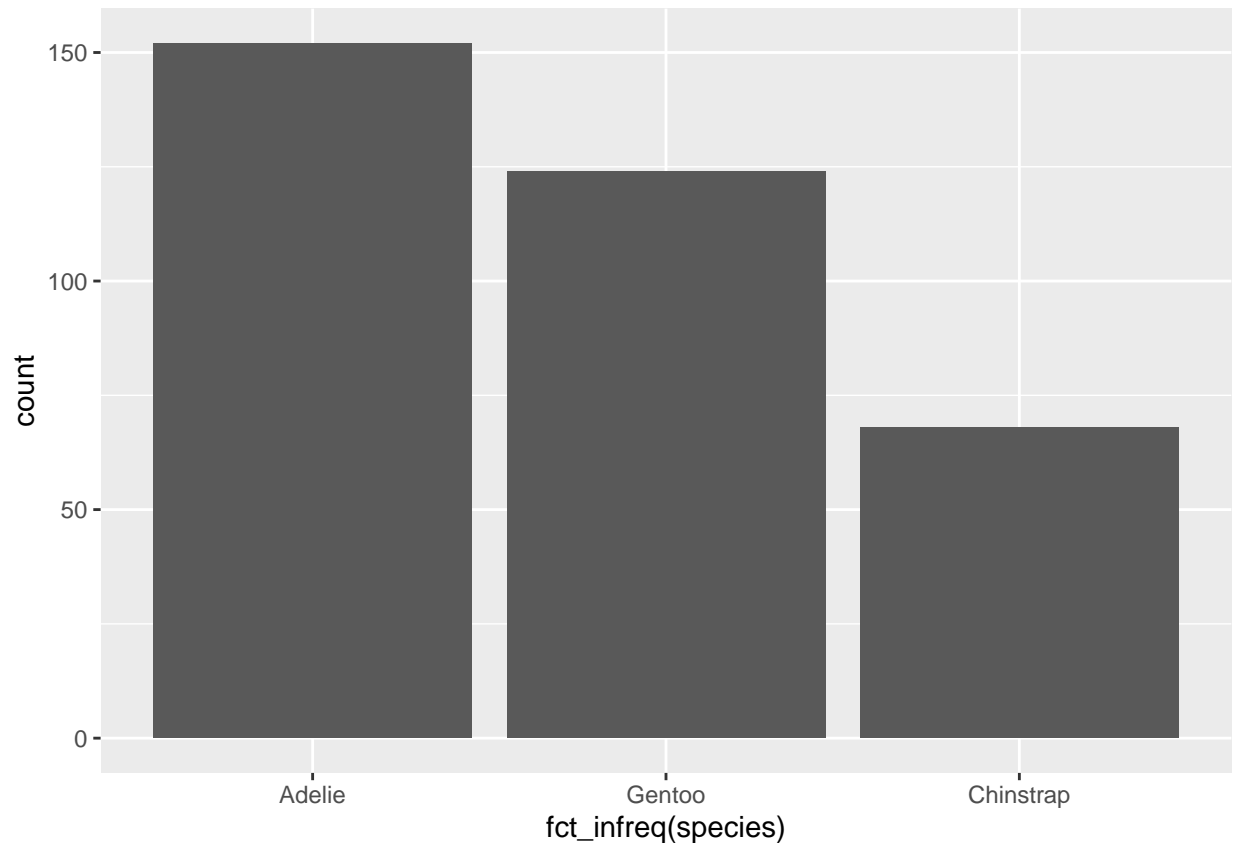
A variable is *categorical* if it can only take one of a small set of values. To examine the distribution of a categorical variable, you can use a bar chart. The height of the bars displays how many observations occurred with each x value.

```
ggplot(penguins,  
  aes(x = species)) +  
  geom_bar()
```



In bar plots of categorical variables with non-ordered levels, like the penguin species above, it's often preferable to reorder the bars based on their frequencies. Doing so requires transforming the variable to a factor (how R handles categorical data) and then reordering the levels of that factor. So, you can use `fct_infreq()`.

```
ggplot(penguins,  
  aes(x = fct_infreq(species))) +  
  geom_bar()
```

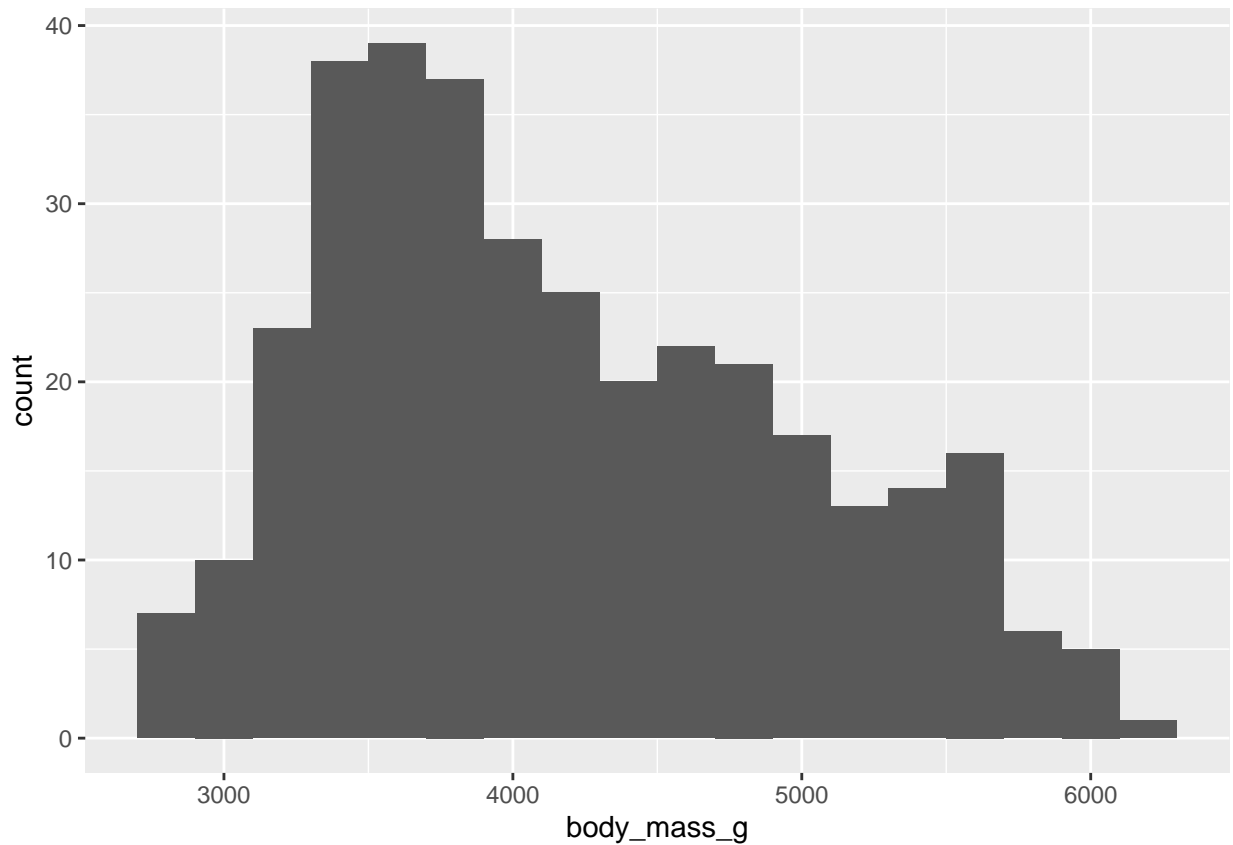


A variable is *numerical* (or quantitative) if it can take on a wide range of numerical values, and it is sensible to add, subtract, or take averages with those values. Numerical variables can be continuous or discrete.

One commonly used visualization for distributions of continuous variables is a *histogram*. A histogram divides the x-axis into equally spaced bins and then uses the height of a bar to display the number of observations that fall in each bin.

You can set the width of the intervals in a histogram with the `binwidth` argument, which is measured in the units of the x variable.

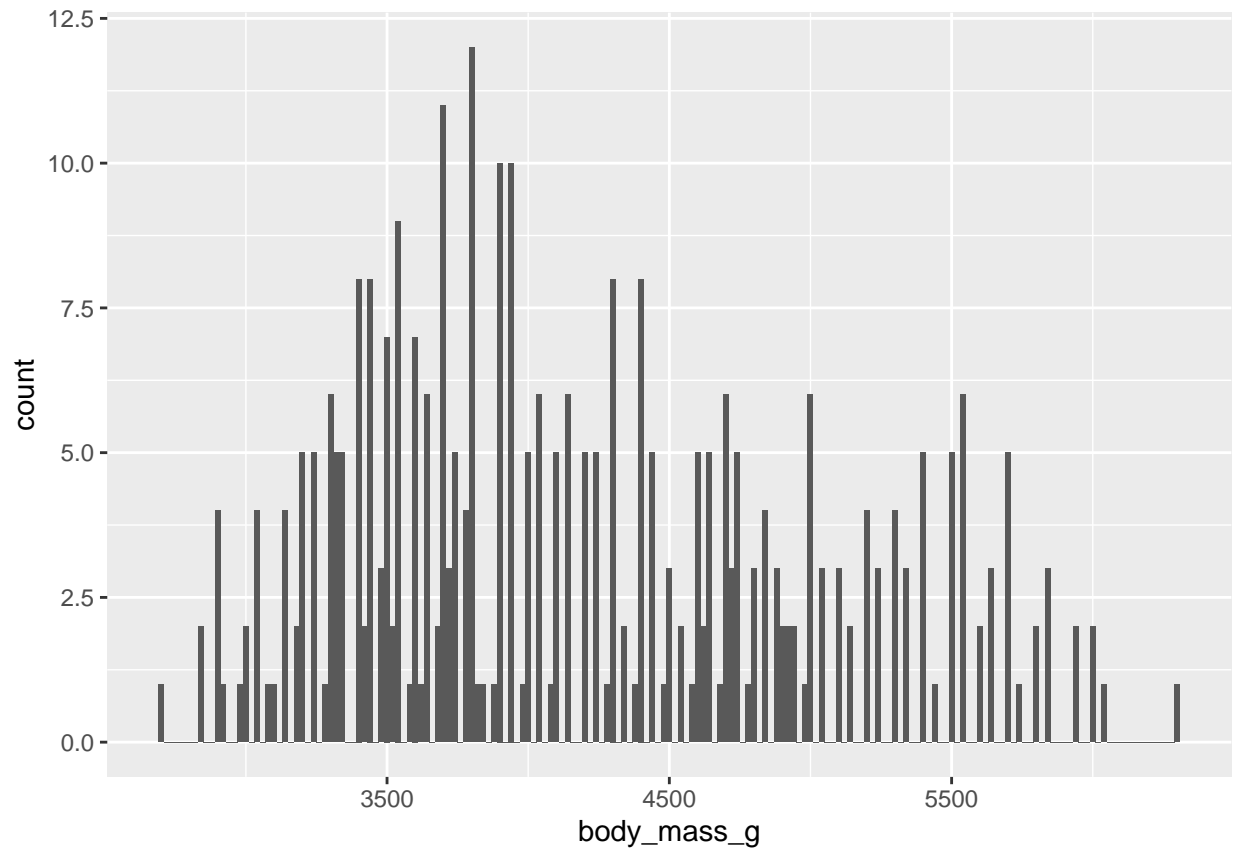
```
ggplot(penguins,  
  aes(x = body_mass_g)) +  
  geom_histogram(binwidth = 200)
```



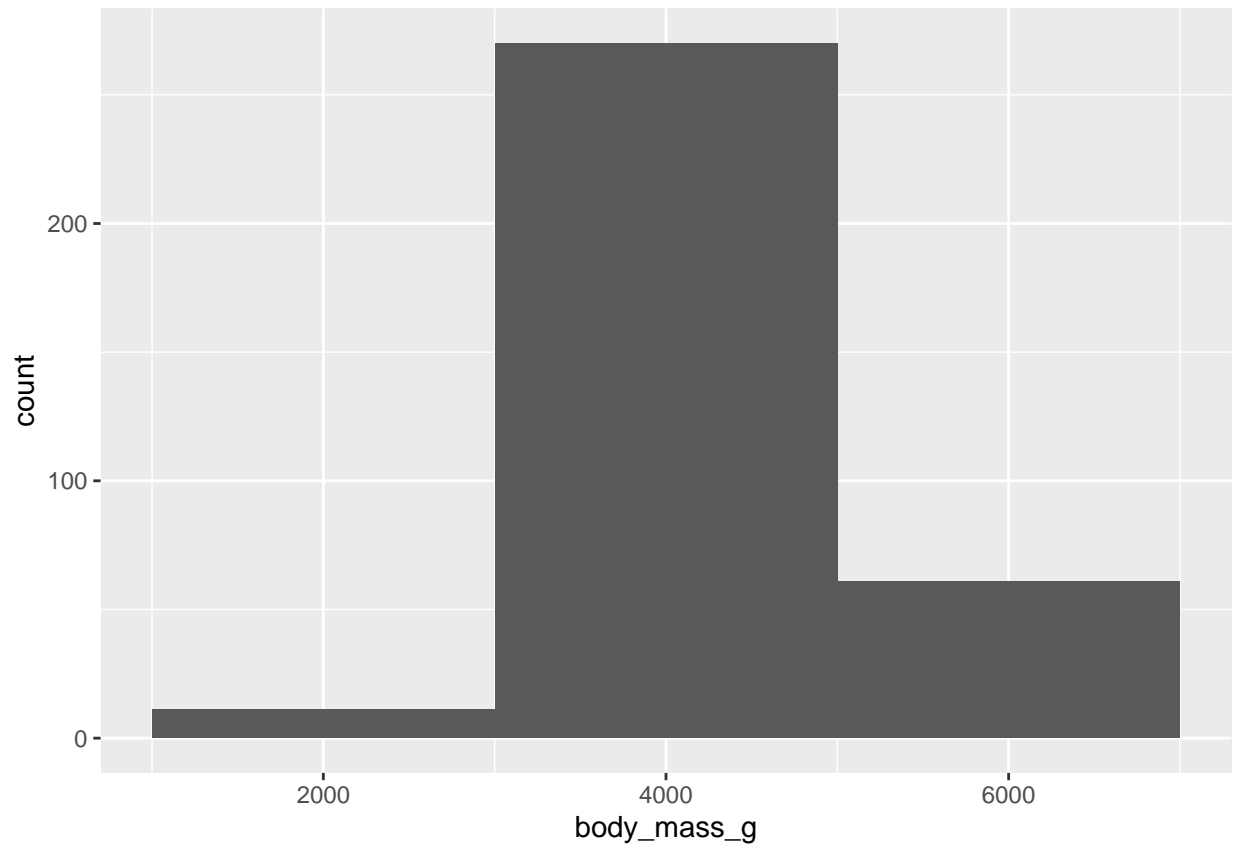
You should always explore a variety of binwidths when working with histograms, as different binwidths can reveal different patterns.

In the plots below a binwidth of 20 is too narrow, resulting in too many bars, making it difficult to determine the shape of the distribution. Similarly, a binwidth of 2,000 is too high, resulting in all data being binned into only three bars, and also making it difficult to determine the shape of the distribution.

```
ggplot(penguins,  
  aes(x = body_mass_g)) +  
  geom_histogram(binwidth = 20)
```

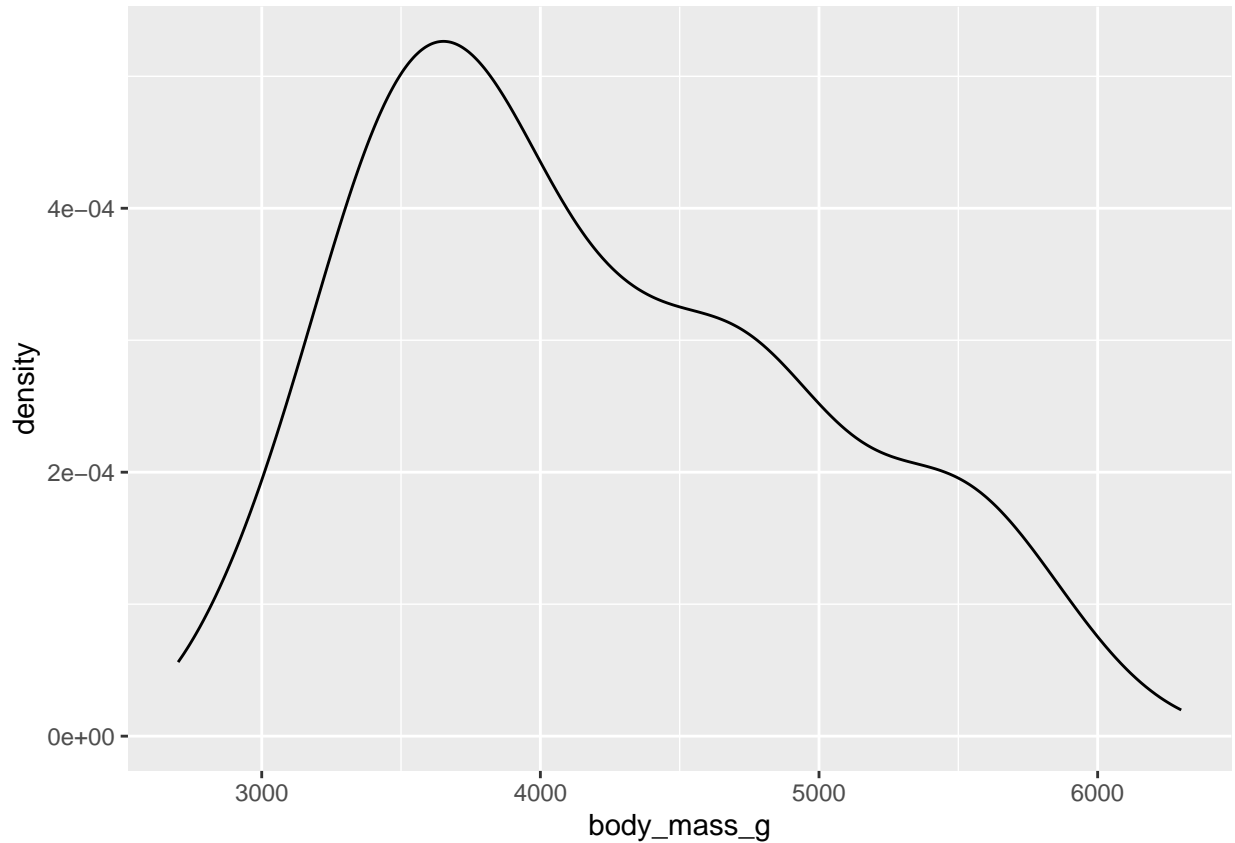


```
ggplot(penguins,  
  aes(x = body_mass_g)) +  
  geom_histogram(binwidth = 2000)
```

A *density* plot is a smoothed-out version of a histogram and a practical alternative, particularly for continuous data that comes from an underlying smooth distribution.

```
ggplot(penguins,  
  aes(x = body_mass_g)) +  
  geom_density()
```



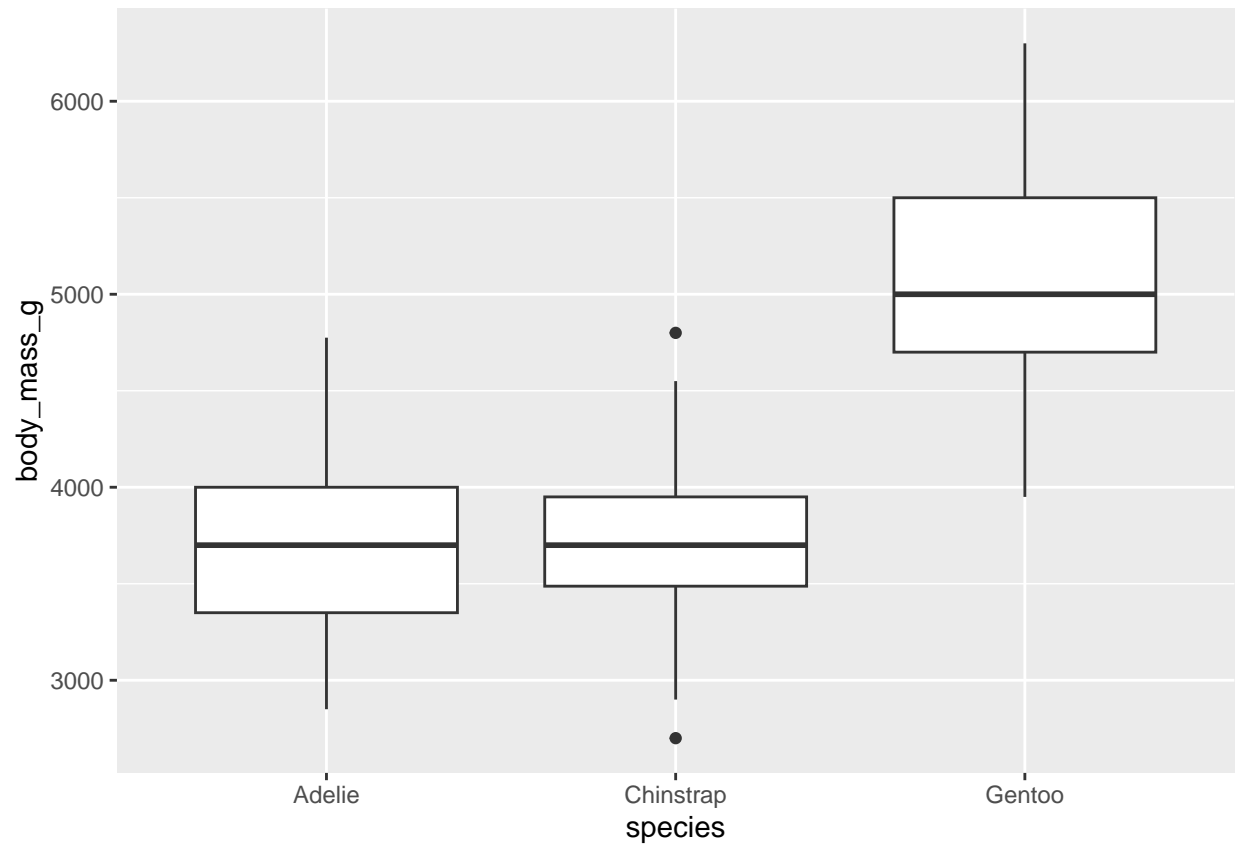
4. Visualizing relationships

To visualize the relationship between a *numerical* and a *categorical* variable we can use side-by-side box plots.

A *boxplot* is a type of visual shorthand for measures of position (percentiles) that describe a distribution. It is also useful for identifying potential outliers.

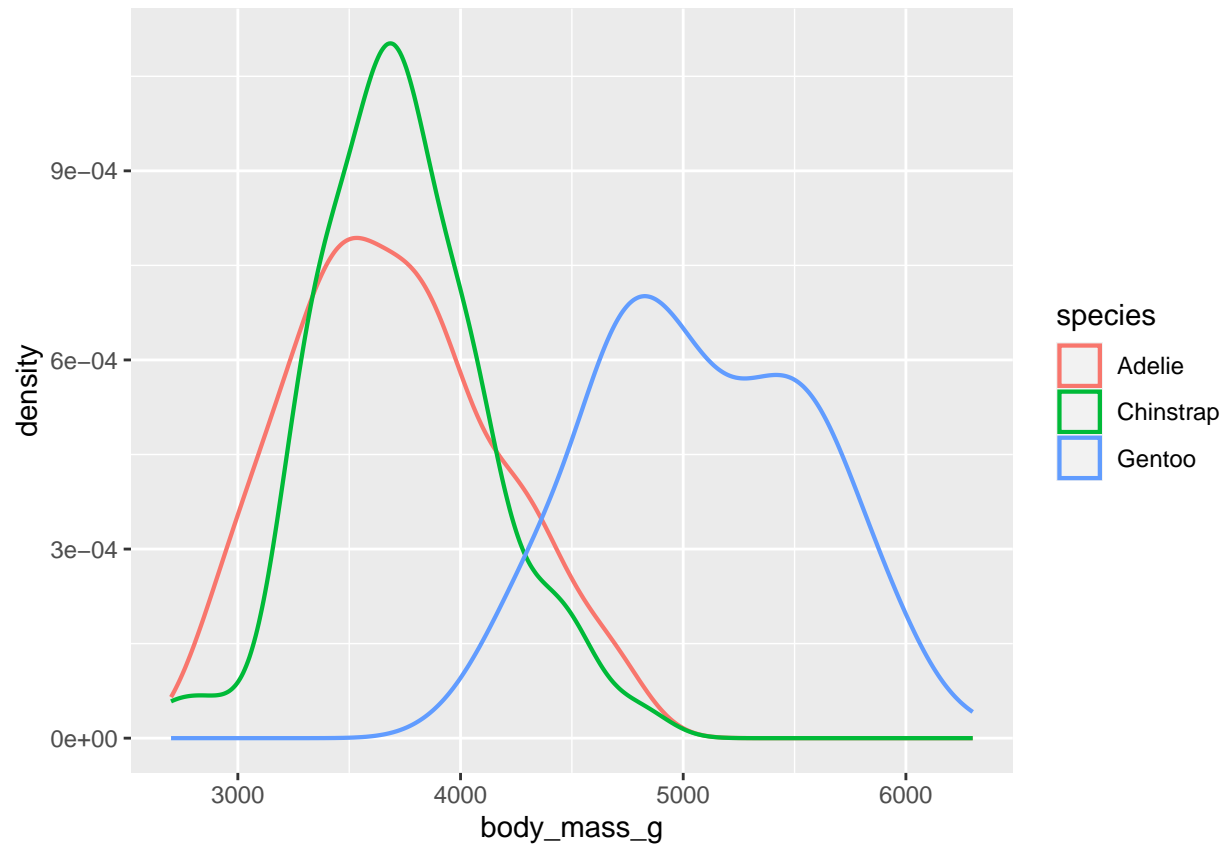
- A box that indicates the range of the middle half of the data, a distance known as the interquartile range (IQR). The 25th, 75th, and the median lines give you a sense of the spread of the distribution and whether or not the distribution is symmetric about the median or skewed to one side.
- A line (or whisker) that extends from each end of the box and goes to the farthest non-outlier point in the distribution.

```
ggplot(penguins,  
  aes(x = species,  
      y = body_mass_g)) +  
  geom_boxplot()
```



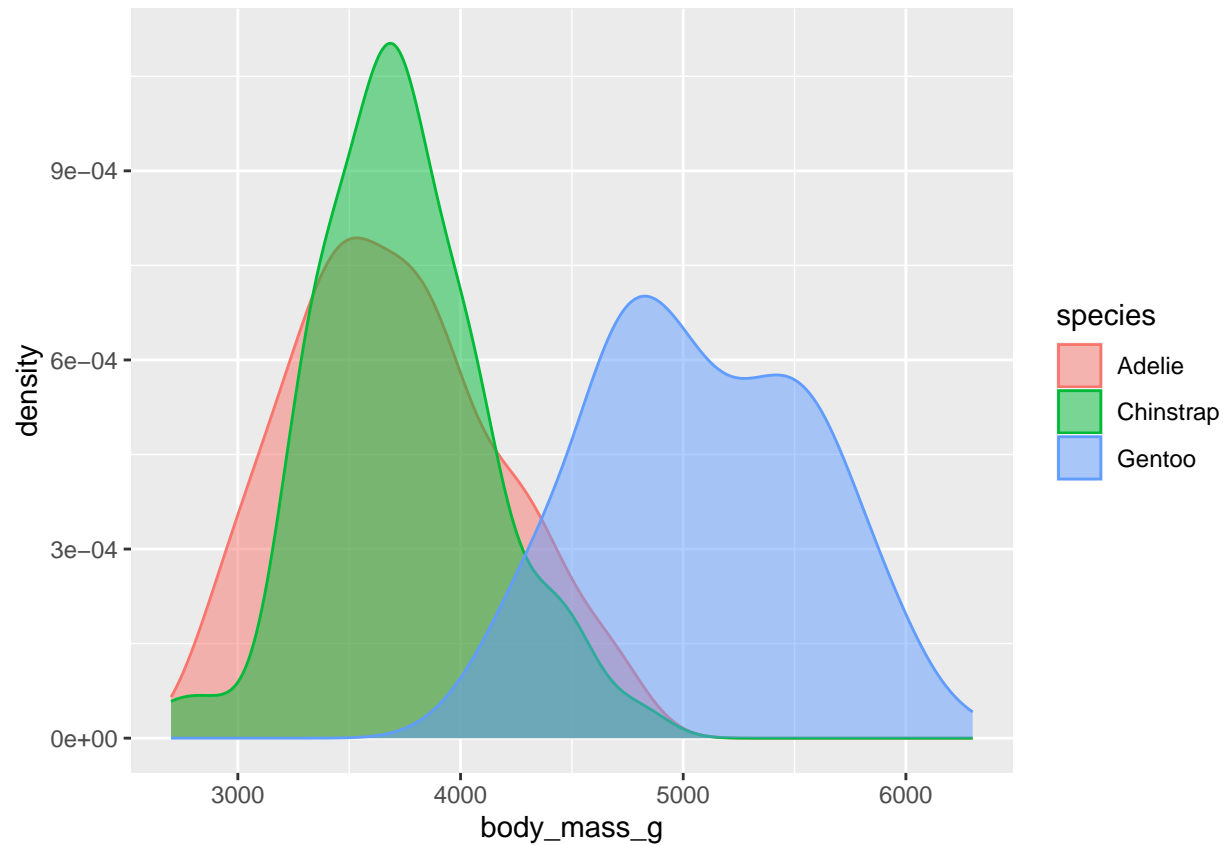
Alternatively, we can make density plots with `geom_density()`. You can customize the thickness of the lines using the `linewidth` argument in order to make them stand out a bit more against the background.

```
ggplot(penguins,  
  aes(x = body_mass_g,  
      color = species)) +  
  geom_density(linewidth = 0.75)
```



Additionally, we can map species to both `color` and `fill` aesthetics and use the `alpha` aesthetic to add transparency to the filled density curves. This aesthetic takes values between 0 (completely transparent) and 1 (completely opaque).

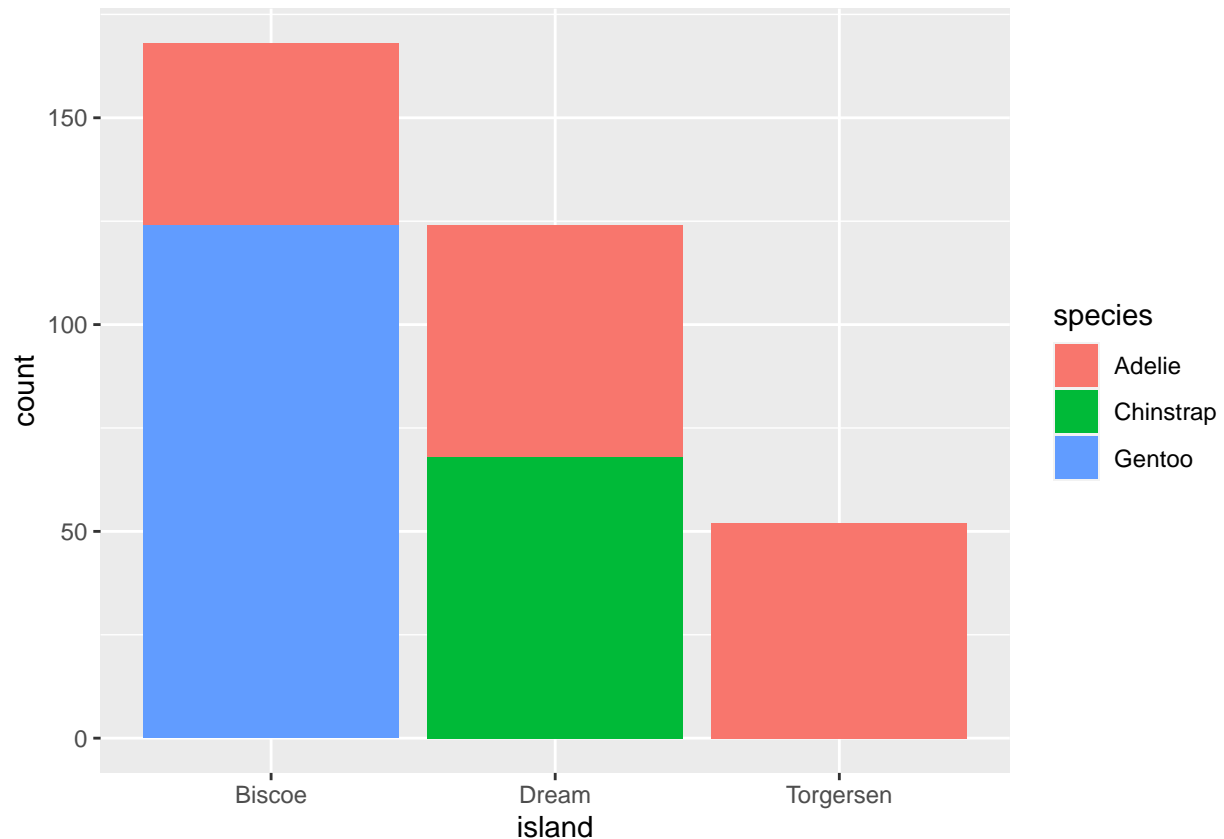
```
ggplot(penguins,  
  aes(x = body_mass_g,  
    color = species,  
    fill = species)) +  
  geom_density(alpha = 0.5)
```



We can use **stacked bar plots** to visualize the relationship between two categorical variables.

The first plot shows the frequencies of each species of penguins on each island. The plot of frequencies shows that there are equal numbers of Adelies on each island. But we don't have a good sense of the percentage balance within each island.

```
ggplot(penguins,
  aes(x = island,
    fill = species)) +
  geom_bar()
```

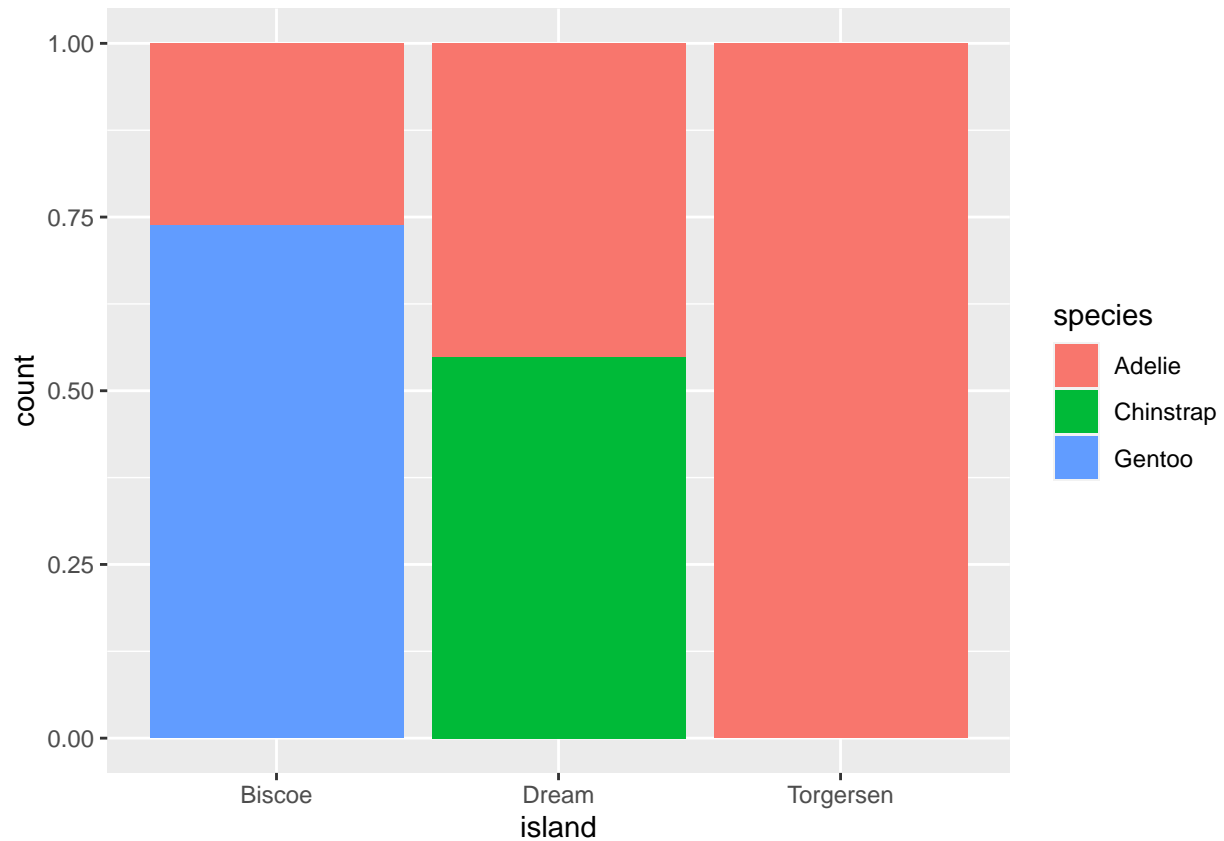


The second plot, a relative frequency plot created by setting `position = "fill"` in the geom, is more useful for comparing species distributions across islands since it's not affected by the unequal numbers of penguins across the islands.

Using this plot we can see that Gentoo penguins all live on Biscoe island and make up roughly 75% of the penguins on that island, Chinstrap all live on Dream island and make up roughly 50% of the penguins on that island, and Adelie live on all three islands and make up all of the penguins on Torgersen.

In creating these bar charts, we map the variable that will be separated into bars to the `x` aesthetic, and the variable that will change the colors inside the bars to the `fill` aesthetic.

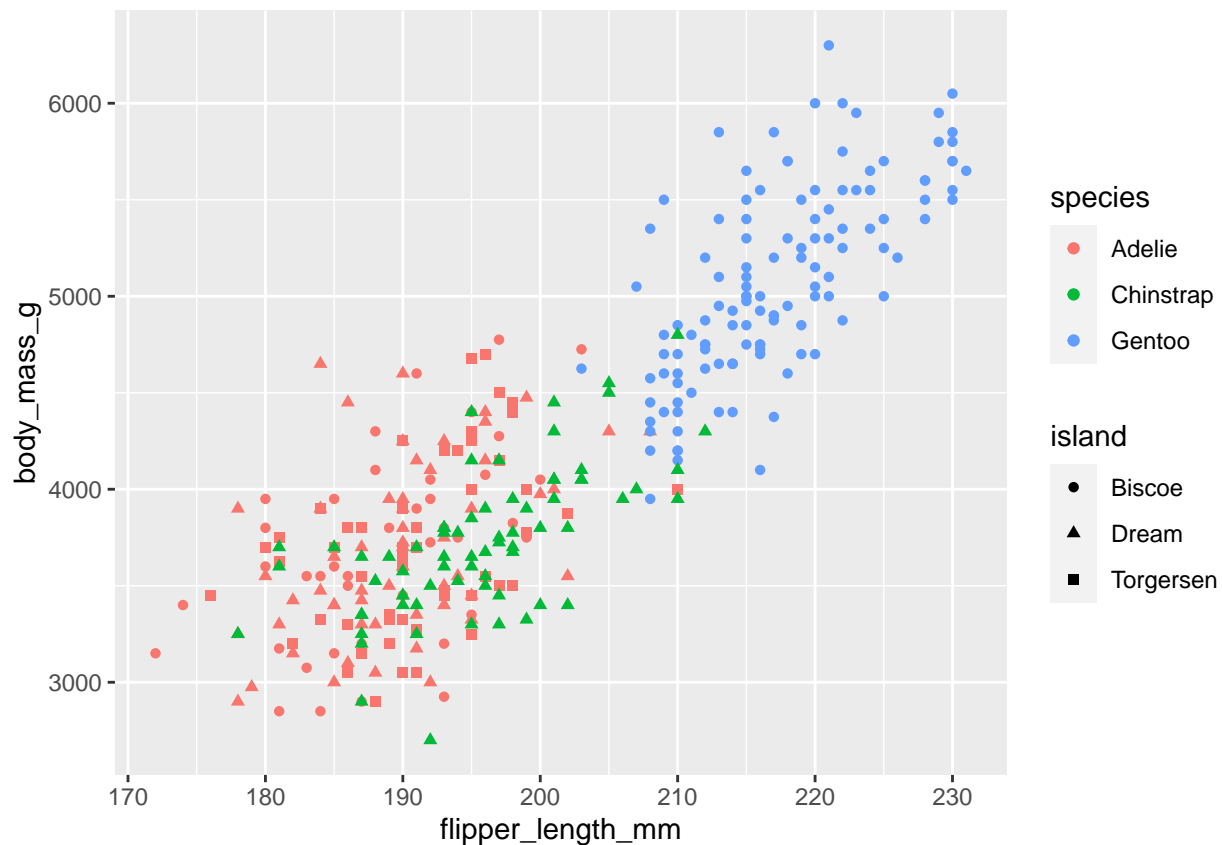
```
ggplot(penguins,
       aes(x = island,
           fill = species)) +
  geom_bar(position = "fill")
```



5. Three or more variables

We can incorporate more variables into a plot by mapping them to additional aesthetics. For example, in the following scatterplot the colors of points represent species and the shapes of points represent islands.

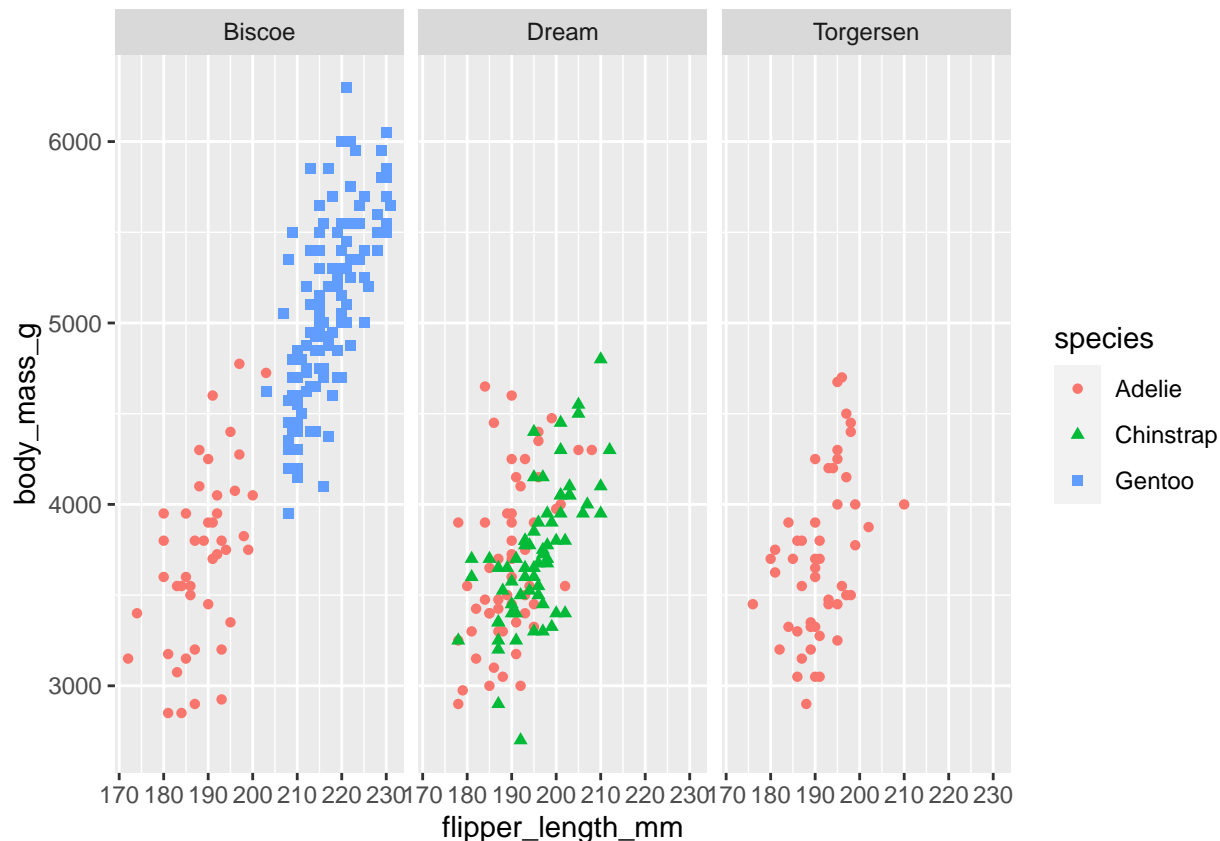
```
ggplot(penguins,  
  aes(x = flipper_length_mm,  
      y = body_mass_g)) +  
  geom_point(aes(color = species,  
                 shape = island))
```



However adding too many aesthetic mappings to a plot makes it cluttered and difficult to make sense of. Another way, which is particularly useful for categorical variables, is to split your plot into *facets*, subplots that each display one subset of the data.

To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` is a formula, which you create with `~` followed by a variable name. The variable that you pass to `facet_wrap()` should be *categorical*.

```
ggplot(penguins,
  aes(x = flipper_length_mm,
      y = body_mass_g)) +
  geom_point(aes(color = species, shape = species)) +
  facet_wrap(~island)
```

6. Data Transformation

The primary dplyr verbs (functions) have in common:

1. The first argument is always a data frame.
2. The subsequent arguments typically describe which columns to operate on, using the variable names (without quotes).
3. The output is always a new data frame.

Because each verb does one thing well, solving complex problems will usually require combining multiple verbs, and we'll do so with the pipe, `|>`.

The pipe takes the thing on its left and passes it along to the function on its right so that `x |> f(y)` is equivalent to `f(x, y)`, and `x |> f(y) |> g(z)` is equivalent to `g(f(x, y), z)`. The easiest way to pronounce the pipe is "then".

```
flights |>
  filter(dest == "IAH") |>
  group_by(year, month, day) |>
  summarize(
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )
```

```
## # A tibble: 365 x 4
```

```
## # Groups:   year, month [12]
##   year month   day arr_delay
##   <int> <int> <int>     <dbl>
## 1  2013     1     1      17.8
## 2  2013     1     2       7
## 3  2013     1     3      18.3
## 4  2013     1     4      -3.2
## 5  2013     1     5      20.2
## 6  2013     1     6       9.28
## 7  2013     1     7      -7.74
## 8  2013     1     8       7.79
## 9  2013     1     9      18.1
## 10 2013     1    10       6.68
## # i 355 more rows
```

dplyr's verbs are organized into four groups based on what they operate on: *rows*, *columns*, *groups*, or *tables*.

6.1. Rows

The most important verbs that operate on rows of a dataset are `filter()`, which changes which rows are present without changing their order, and `arrange()`, which changes the order of the rows without changing which are present. `distinct()` which finds rows with unique values but unlike `arrange()` and `filter()` it can also optionally modify the columns.

6.1.1 filter()

`filter()` allows you to keep rows based on the values of the columns. The first argument is the data frame. The second and subsequent arguments are the conditions that must be true to keep the row.

```
flights |>
  filter(dep_delay > 120)
```

```
## # A tibble: 9,723 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     848           1835         853       1001           1950
## 2  2013     1     1     957           733         144       1056           853
## 3  2013     1     1    1114           900         134      1447           1222
## 4  2013     1     1    1540          1338         122      2020           1825
## 5  2013     1     1    1815          1325         290      2120           1542
## 6  2013     1     1    1842          1422         260      1958           1535
## 7  2013     1     1    1856          1645         131      2212           2005
## 8  2013     1     1    1934          1725         129      2126           1855
## 9  2013     1     1    1938          1703         155      2109           1823
## 10 2013     1     1    1942          1705         157      2124           1830
## # i 9,713 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 <dtm>
```

As well as `>` (greater than), you can use `>=` (greater than or equal to), `<` (less than), `<=` (less than or equal to), `==` (equal to), and `!=` (not equal to). You can also combine conditions with `&` or `,` to indicate “and” (check for both conditions) or with `|` to indicate “or” (check for either condition):

```
# Flights that departed on January 1
flights |>
  filter(month == 1 & day == 1)
```

```
## # A tibble: 842 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 832 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 <dtm>
```

```
# Flights that departed in January or February
flights |>
  filter(month == 1 | month == 2)
```

```
## # A tibble: 51,955 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 51,945 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 <dtm>
```

There's a useful shortcut when you're combining `|` and `==`: `%in%`. It keeps rows where the variable equals one of the values on the right.

```
# A shorter way to select flights that departed in January or February
flights |>
  filter(month %in% c(1, 2))
```

```
## # A tibble: 51,955 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 51,945 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 <dtm>
```

6.1.2 arrange()

`arrange()` changes the order of the rows based on the value of the columns. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns. For example, the following code sorts by the departure time, which is spread over four columns. We get the earliest years first, then within a year the earliest months, etc.

```
flights |>
  arrange(year, month, day, dep_time)
```

```
## # 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 <dtm>
```

You can use `desc()` on a column inside of `arrange()` to re-order the data frame based on that column in descending (big-to-small) order.

```
flights |>
  arrange(desc(dep_delay))
```

```
## # 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     9     641             900      1301    1242         1530
## 2  2013     6    15    1432            1935      1137    1607         2120
## 3  2013     1    10    1121            1635      1126    1239         1810
## 4  2013     9    20    1139            1845      1014    1457         2210
## 5  2013     7    22     845            1600      1005    1044         1815
## 6  2013     4    10    1100            1900       960    1342         2211
## 7  2013     3    17    2321             810       911     135         1020
## 8  2013     6    27     959            1900       899    1236         2226
## 9  2013     7    22    2257             759       898     121         1026
## 10 2013    12     5     756            1700       896    1058         2020
## # 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 <dtm>
```

6.1.3 distinct()

`distinct()` finds all the unique rows in a dataset, so in a technical sense, it primarily operates on the rows. Most of the time, however, you'll want the distinct combination of some variables, so you can also optionally supply column names:

```
# Remove duplicate rows, if any
flights |>
  distinct()
```

```
## # 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 <dtm>
```

```
# Find all unique origin and destination pairs
flights |>
  distinct(origin, dest)
```

```
## # A tibble: 224 x 2
##   origin dest
```

```
##      <chr> <chr>
## 1 EWR    IAH
## 2 LGA    IAH
## 3 JFK    MIA
## 4 JFK    BQN
## 5 LGA    ATL
## 6 EWR    ORD
## 7 EWR    FLL
## 8 LGA    IAD
## 9 JFK    MCO
## 10 LGA   ORD
## # i 214 more rows
```

Alternatively, if you want to keep other columns when filtering for unique rows, you can use the `.keep_all = TRUE` option.

```
flights |>
  distinct(origin, dest, .keep_all = TRUE)
```

```
## # 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 <dtm>
```

It's not a coincidence that all of these distinct flights are on January 1: `distinct()` will find the first occurrence of a unique row in the dataset and discard the rest.

If you want to find the number of occurrences instead, you're better off swapping `distinct()` for `count()`, and with the `sort = TRUE` argument you can arrange them in *descending* order of number of occurrences.

```
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
```

6.2 Columns

There are four important verbs that affect the columns without changing the rows: `mutate()` creates new columns that are derived from the existing columns, `select()` changes which columns are present, `rename()` changes the names of the columns, and `relocate()` changes the positions of the columns.

6.2.1 mutate()

The job of `mutate()` is to add new columns that are calculated from the existing columns.

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
  )
```

```
## # A tibble: 336,776 x 21
##   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 13 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 <dtm>, gain <dbl>, speed <dbl>
```

By default, `mutate()` adds new columns on the right hand side of your dataset, which makes it difficult to see what's happening here. We can use the `.before` argument to instead add the variables to the left hand side.

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
```

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

The `.` is a sign that `.before` is an argument to the function, not the name of a third new variable we are creating. You can also use `.after` to add after a variable, and in both `.before` and `.after` you can use the variable name instead of a position.

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .after = day
  )
```

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

Alternatively, you can control which variables are kept with the `.keep` argument. A particularly useful argument is "used" which specifies that we only keep the columns that were involved or created in the `mutate()` step. For example, the following output will contain only the variables `dep_delay`, `arr_delay`, `air_time`, `gain`, `hours`, and `gain_per_hour`.


```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours,
    .keep = "used"
  )
```

```
## # A tibble: 336,776 x 6
##   dep_delay arr_delay air_time  gain hours gain_per_hour
##   <dbl>    <dbl>    <dbl> <dbl> <dbl>    <dbl>
## 1         2        11      227    -9  3.78      -2.38
## 2         4        20      227   -16  3.78      -4.23
## 3         2        33      160   -31  2.67     -11.6
## 4        -1       -18      183    17  3.05       5.57
## 5        -6       -25      116    19  1.93       9.83
## 6        -4        12      150   -16  2.5       -6.4
## 7        -5        19      158   -24  2.63      -9.11
## 8        -3       -14        53    11  0.883     12.5
## 9        -3        -8      140     5  2.33       2.14
## 10       -2         8      138   -10  2.3       -4.35
## # i 336,766 more rows
```

6.2.2 select()

It's not uncommon to get datasets with hundreds or even thousands of variables. In this situation, the first challenge is often just focusing on the variables you're interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

```
# Select columns by name:
flights |>
  select(year, month, day)
```

```
## # A tibble: 336,776 x 3
##   year month  day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
## 7  2013     1     1
## 8  2013     1     1
## 9  2013     1     1
## 10 2013     1     1
## # i 336,766 more rows
```

```
# Select all columns between year and day (inclusive):
flights |>
  select(year:day)
```

```
## # A tibble: 336,776 x 3
##   year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
## 7  2013     1     1
## 8  2013     1     1
## 9  2013     1     1
## 10 2013     1     1
## # i 336,766 more rows
```

```
# Select all columns except those from year to day (inclusive):
flights |>
  select(!year:day)
```

```
## # A tibble: 336,776 x 16
##   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##   <int>         <int>      <dbl>   <int>         <int>      <dbl> <chr>
## 1     517           515         2     830           819        11 UA
## 2     533           529         4     850           830        20 UA
## 3     542           540         2     923           850        33 AA
## 4     544           545        -1    1004          1022       -18 B6
## 5     554           600        -6     812           837       -25 DL
## 6     554           558        -4     740           728        12 UA
## 7     555           600        -5     913           854        19 B6
## 8     557           600        -3     709           723       -14 EV
## 9     557           600        -3     838           846         -8 B6
## 10    558           600        -2     753           745         8 AA
## # i 336,766 more rows
## # i 9 more variables: flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

```
# Select all columns that are characters
flights |>
  select(where(is.character))
```

```
## # A tibble: 336,776 x 4
##   carrier tailnum origin dest
##   <chr>    <chr>   <chr> <chr>
## 1 UA      N14228  EWR   IAH
## 2 UA      N24211  LGA   IAH
## 3 AA      N619AA  JFK   MIA
## 4 B6      N804JB  JFK   BQN
## 5 DL      N668DN  LGA   ATL
## 6 UA      N39463  EWR   ORD
## 7 B6      N516JB  EWR   FLL
## 8 EV      N829AS  LGA   IAD
## 9 B6      N593JB  JFK   MCO
## 10 AA     N3ALAA  LGA   ORD
## # i 336,766 more rows
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")`: matches names that begin with “abc”.
- `ends_with("xyz")`: matches names that end with “xyz”.
- `contains("ijk")`: matches names that contain “ijk”.
- `num_range("x", 1:3)`: matches x1, x2 and x3.

You can rename variables as you `select()` them by using `=`. The new name appears on the left hand side of the `=`, and the old variable appears on the right hand side.

```
flights |>
  select(tail_num = tailnum)
```

```
## # A tibble: 336,776 x 1
##   tail_num
##   <chr>
## 1 N14228
## 2 N24211
## 3 N619AA
## 4 N804JB
## 5 N668DN
## 6 N39463
## 7 N516JB
## 8 N829AS
## 9 N593JB
## 10 N3ALAA
## # i 336,766 more rows
```

6.2.3 `rename()`

If you want to keep all the existing variables and just want to rename a few, you can use `rename()` instead of `select()`:

```
flights |>
  rename(tail_num = tailnum)
```

```
## # 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>,
## #   tail_num <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

6.2.4 relocate()

Use `relocate()` to move variables around. You might want to collect related variables together or move important variables to the front. By default `relocate()` moves variables to the front:

```
flights |>
  relocate(time_hour, air_time)
```

```
## # A tibble: 336,776 x 19
##   time_hour          air_time year month   day dep_time sched_dep_time
##   <dtm>              <dbl> <int> <int> <int>   <int>         <int>
## 1 2013-01-01 05:00:00      227  2013     1     1     517           515
## 2 2013-01-01 05:00:00      227  2013     1     1     533           529
## 3 2013-01-01 05:00:00      160  2013     1     1     542           540
## 4 2013-01-01 05:00:00      183  2013     1     1     544           545
## 5 2013-01-01 06:00:00      116  2013     1     1     554           600
## 6 2013-01-01 05:00:00      150  2013     1     1     554           558
## 7 2013-01-01 06:00:00      158  2013     1     1     555           600
## 8 2013-01-01 06:00:00       53  2013     1     1     557           600
## 9 2013-01-01 06:00:00      140  2013     1     1     557           600
## 10 2013-01-01 06:00:00     138  2013     1     1     558           600
## # i 336,766 more rows
## # i 12 more variables: dep_delay <dbl>, arr_time <int>, sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
## #   dest <chr>, distance <dbl>, hour <dbl>, minute <dbl>
```

You can also specify where to put them using the `.before` and `.after` arguments, just like in `mutate()`:

```
flights |>
  relocate(year: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 <dtm>, year <int>,
## #   month <int>, day <int>, dep_time <int>
```

```
flights |>
  relocate(starts_with("arr"), .before = dep_time)
```

```
## # A tibble: 336,776 x 19
##   year month   day arr_time arr_delay dep_time sched_dep_time dep_delay
##   <int> <int> <int>   <int>     <dbl>   <int>         <int>         <dbl>
## 1  2013     1     1     830         11     517           515             2
## 2  2013     1     1     850         20     533           529             4
## 3  2013     1     1     923         33     542           540             2
## 4  2013     1     1    1004        -18     544           545            -1
## 5  2013     1     1     812        -25     554           600            -6
## 6  2013     1     1     740         12     554           558            -4
## 7  2013     1     1     913         19     555           600            -5
## 8  2013     1     1     709        -14     557           600            -3
## 9  2013     1     1     838         -8     557           600            -3
## 10 2013     1     1     753          8     558           600            -2
## # i 336,766 more rows
## # i 11 more variables: sched_arr_time <int>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

6.3 The Pipe

We've shown you simple examples of the pipe above, but its real power arises when you start to combine multiple verbs. For example, imagine that you wanted to find the fastest flights to Houston's IAH airport: you need to combine `filter()`, `mutate()`, `select()`, and `arrange()`:

```
flights |>
  filter(dest == "IAH") |>
  mutate(speed = distance / air_time * 60) |>
  select(year:day, dep_time, carrier, flight, speed) |>
  arrange(desc(speed))
```

```
## # A tibble: 7,198 x 7
##   year month   day dep_time carrier flight speed
##   <int> <int> <int>   <int> <chr>   <int> <dbl>
## 1  2013     7     9     707 UA       226  522.
## 2  2013     8    27    1850 UA      1128  521.
## 3  2013     8    28     902 UA      1711  519.
## 4  2013     8    28    2122 UA      1022  519.
## 5  2013     6    11    1628 UA      1178  515.
## 6  2013     8    27    1017 UA       333  515.
## 7  2013     8    27    1205 UA      1421  515.
## 8  2013     8    27    1758 UA       302  515.
## 9  2013     9    27     521 UA       252  515.
## 10 2013     8    28     625 UA       559  515.
## # i 7,188 more rows
```

6.4 Groups

dplyr gets even more powerful when you add in the ability to work with groups. In this section, we'll focus on the most important functions: `group_by()`, `summarize()`, and the slice family of functions.

6.4.1 group_by

Use `group_by()` to divide your dataset into groups meaningful for your analysis:

```
flights |>
  group_by(month)

## # A tibble: 336,776 x 19
## # Groups:   month [12]
##   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 <dtm>
```

`group_by()` doesn't change the data but, if you look closely at the output, you'll notice that the output indicates that it is "grouped by" month (**Groups: month [12]**). This means subsequent operations will now work "by month". `group_by()` adds this grouped feature (referred to as class) to the data frame, which changes the behavior of the subsequent verbs applied to the data.

6.4.2 summarize()

The most important grouped operation is a summary, which, if being used to calculate a single summary statistic, reduces the data frame to have a single row for each group. In dplyr, this operation is performed by `summarize()`, as shown by the following example, which computes the average departure delay by month.

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE)
  )

## # A tibble: 12 x 2
##   month avg_delay
##   <int>   <dbl>
## 1     1    10.0
## 2     2    10.8
## 3     3    13.2
## 4     4    13.9
## 5     5    13.0
## 6     6    20.8
```

```
## 7      7      21.7
## 8      8      12.6
## 9      9       6.72
## 10     10       6.24
## 11     11       5.44
## 12     12      16.6
```

You can create any number of summaries in a single call to `summarize()`. One very useful summary is `n()`, which returns the number of rows in each group.

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  )
```

```
## # A tibble: 12 x 3
##   month avg_delay     n
##   <int>   <dbl> <int>
## 1     1    10.0 27004
## 2     2    10.8 24951
## 3     3    13.2 28834
## 4     4    13.9 28330
## 5     5    13.0 28796
## 6     6    20.8 28243
## 7     7    21.7 29425
## 8     8    12.6 29327
## 9     9     6.72 27574
## 10    10     6.24 28889
## 11    11     5.44 27268
## 12    12    16.6 28135
```

6.4.3 The `slice_` functions

There are five handy functions that allow you extract specific rows within each group:

- `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.

You can vary `n` to select more than one row, or instead of `n =`, you can use `prop = 0.1` to select (e.g.) 10% of the rows in each group. For example, the following code finds the flights that are most delayed upon arrival at each destination:

```
flights |>
  group_by(dest) |>
  slice_max(arr_delay, n=1) |>
  relocate(dest)
```

```
## # A tibble: 108 x 19
## # Groups:   dest [105]
##   dest   year month   day dep_time sched_dep_time dep_delay arr_time
##   <chr> <int> <int> <int>   <int>         <int>      <dbl>   <int>
## 1 ABQ    2013     7    22    2145           2007         98     132
## 2 ACK    2013     7    23    1139           800         219    1250
## 3 ALB    2013     1    25     123           2000        323     229
## 4 ANC    2013     8    17    1740          1625         75    2042
## 5 ATL    2013     7    22    2257           759        898     121
## 6 AUS    2013     7    10    2056          1505        351    2347
## 7 AVL    2013     8    13    1156           832        204    1417
## 8 BDL    2013     2    21    1728          1316        252    1839
## 9 BGR    2013    12     1    1504          1056        248    1628
## 10 BHM   2013     4    10     25           1900        325     136
## # i 98 more rows
## # i 11 more variables: sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

Note that there are 105 destinations but we get 108 rows here. What's up? `slice_min()` and `slice_max()` keep tied values so `n = 1` means give us all rows with the highest value. If you want exactly one row per group you can set `with_ties = FALSE`.

6.4.4 Grouping by multiple variables

You can create groups using more than one variable. For example, we could make a group for each date.

```
daily <- flights |>
  group_by(year, month, day)
daily
```

```
## # A tibble: 336,776 x 19
## # Groups:   year, month, day [365]
##   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 <dtm>
```

When you summarize a tibble grouped by more than one variable, each summary peels off the last group. In hindsight, this wasn't a great way to make this function work, but it's difficult to change without breaking

existing code. To make it obvious what's happening, dplyr displays a message that tells you how you can change this behavior:

```
daily_flights <- daily |>  
  summarize(n = n())
```

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

If you're happy with this behavior, you can explicitly request it in order to suppress the message:

```
daily_flights <- daily |>  
  summarize(  
    n = n(),  
    .groups = "drop_last"  
  )
```