Class 5

Data visualization with ggplot2

AUTHOR
Barry Grant

PUBLISHED
October 10, 2022

1. Overview

The ability to make clear and compelling data visualizations is a vital skill for scientists. The difference between good and bad figures can be the difference between a highly influential or an obscure paper, a grant or contract won or lost, a job interview gone well or poorly. In short, If you want to be successful in any technical field you will need to master these skills!

The goals for this hands-on session are for you to:

- Understand major scientific plot types and when you might want to use them.
- Learn the fundamentals of the ggplot2 package and what it can do for you.
- Get experience of working with RStudio projects and Quarto documents to organize your work.

2. Background

One of the biggest attractions of the R programming language is the ability to have complete programmatic control over the plotting of complex graphs and figures. R offers a ridiculously large set of tools and packages for data visualization. The core R language already provides a rich set of plotting functions and plot types. These plotting functions require users to specify how to plot each element on the canvas step by step. These "**base R plots**" offer complete control over virtually every pixel. However, they can be fiddly and time consuming to get just the way you want. By contrast, the **ggplot2** package allows the specification of all plots through set of common *plotting layers* that minimally includes:

- Specifying the **input data** (in the form of a data.frame),
- How the data maps to **aesthetic features** of the plot (the x and y axis, color, plotting character, line type, etc.), and
- The **geometric layers** used in the plot (such as points, bars, lines etc.)

Data visualization with ggplot always involves these steps . Once you have mastered this sequence of steps we will layer on additional customizations and you will see that beautiful and sophisticated plots come within your reach very quickly (Figure 1). Let's get cracking!

Figure 1: By combining *geometric layers* (**geoms**) for your *aesthetic mappings* (**aes**) of your data you can make beautiful visualizations

Side note:

If you have not already, please watch this weeks intro videos, which cover why we want to visualize data graphically, what makes an effective figure, and an introduction to ggplot. You may also wish to (re) visit our previous introduction to R and RStudio as well as our video on major R data structures, data types, and using functions.

- **Q1.** For which phases is data visualization important in our scientific workflows?

- **Q2.** True or False? The ggplot2 package comes already installed with R?

Discussion (click to expand)

3. Getting Organized

Creating a Project

We will begin by getting organized. This entails you opening up RStudio and creating a new RStudio *Project*, then creating a new R *script* for storing your work and notes for this session.

> **Side-note**: If you are alrady fimilar with RMarkdown format documents feel free to use one of these rather than an RScript. If you have not yet heard of these, don't worry we will be building towards these in our next class.

- Begin by opening RStudio and creating a new **Project** `File > New Project > New Directory > New Project` make sure you are working in the folder (a.k.a. directory!) where you want to keep all your work for this class organized. For example, for me this is a directory on my Desktop with the class name (see animated figure below Figure 2). We will create our project as a *subdirectory* called `class05` in this location.

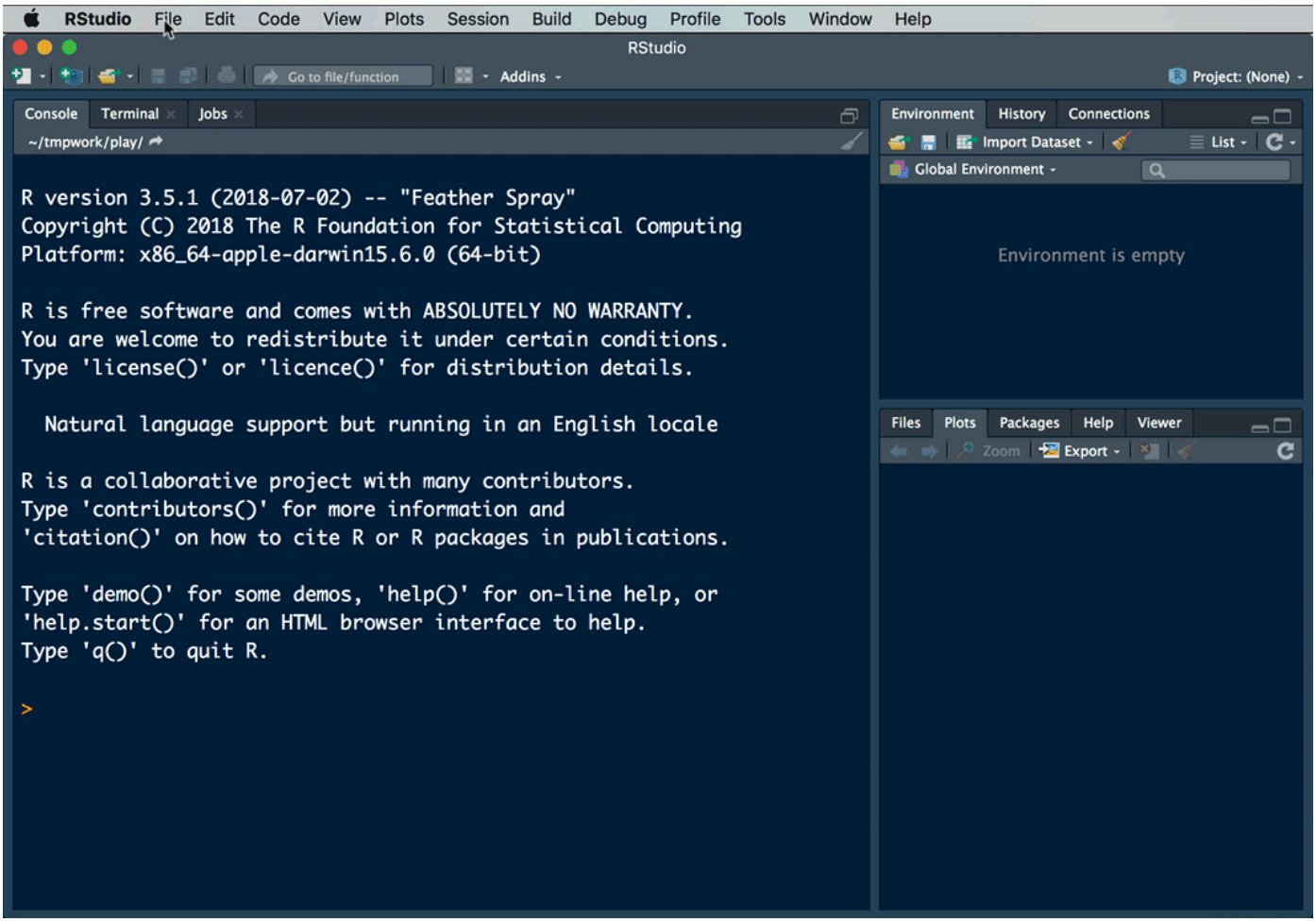


Figure 2: Use *File > New Project > New Directory > New Project* to create a new sub-folder for storing your class work for today.

> **Side-note**: The key step here is to name your project after this class session (i.e. "class05") and make sure it is a sub-directory of where ever you are organizing all your work for this course.

4. Using Quarto

Create a Quarto Document

Open a new **Quarto Document**: `File > New File > Quarto Document`. Give the tile **Class 05: Data Visualization with GGPLOT** your name and click **off** the `Use Visual Markdown Editor` option (see Figure 3). Finally, save the resulting file as save as `class05.qmd` and click the Render button.

> **Note:** If your RStudio does not have the option to create a Quarto Document then you need to update to a more recent version of RStudio. Quarto only came out at the end of Summer 2020 and you will thus need a recent release of RStudio (i.e. v2022.07 or later).

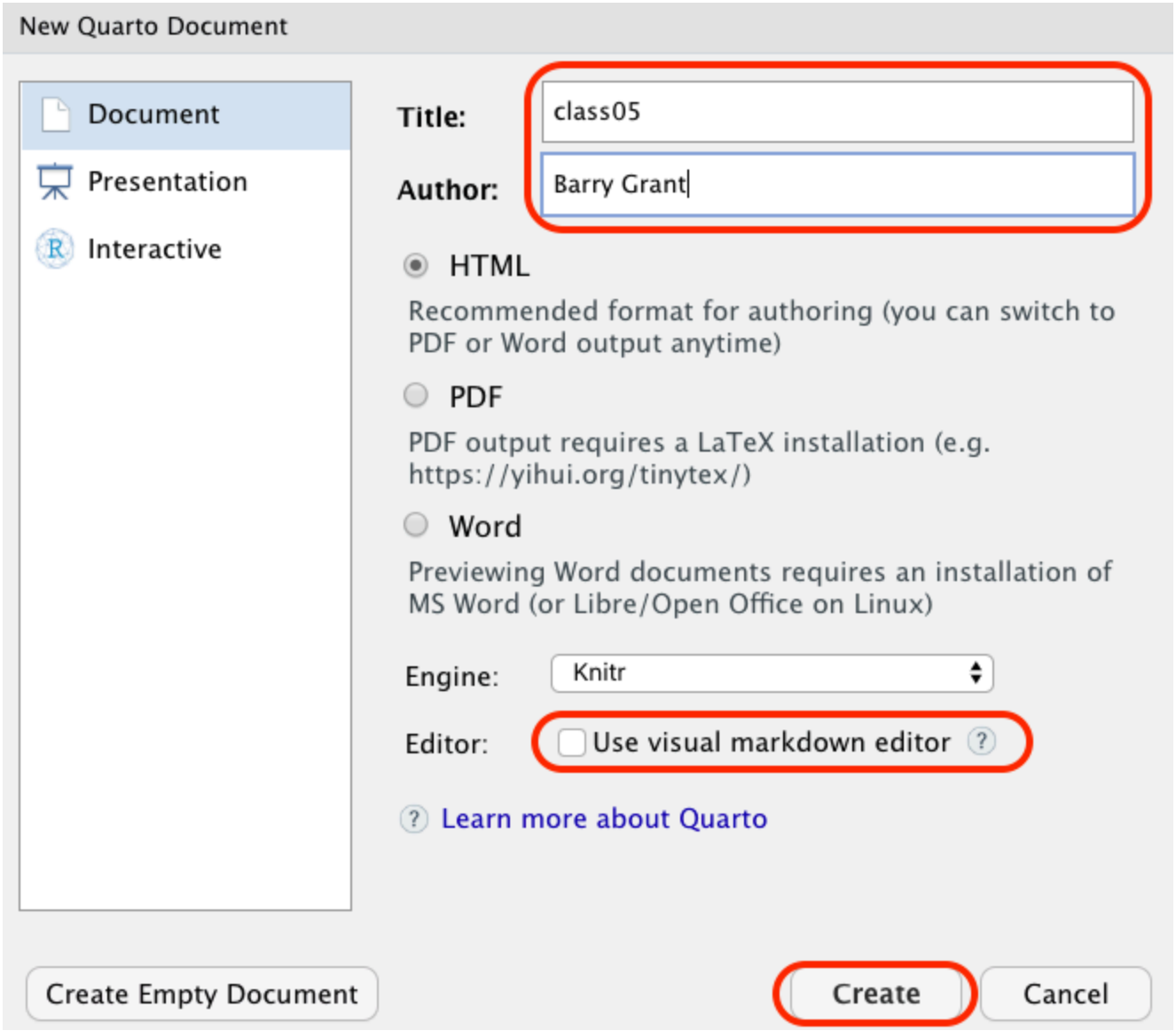


Figure 3: Open a new Quarto document with the *File > New File > Quarto Document* option. If you don't have this option you should update RStudio or use an R Markdown document for today.

What is Quarto?

Recall that last day we worked with a simple **R script** where we focused on writing R code alone. If we wanted to add notes to this R script (basically a text file) we needed to put a hash # comment character at the start of the line so the R brain would not try to execute our narrative

text and comments as R code. Also if we wanted to see the results of our code (e.g. a graph or calculation result) we had to "Run" the entire script.

Quarto documents are much more powerful. Quarto is a complete scientific publishing system. At it's most basic Quarto documents allow us to combine formatted narrative text (think bold, italic, headings, colored text etc.), external figures, (including images, animations, videos etc.) and, most importantly, code form different languages (in so called "code chunks") along with the output of your code all in one document (that is also plain text and thus works well with version control systems).

Quarto is brand new and builds on the older Rmarkdown format. If you are used to Rmarkdown or Jupyter/iPython notebooks then Quarto will feel familiar. If you are not **stop here for an introduction to Quarto from Barry!** If you are reading this outside of class you can learn more about Quarto here https://quarto.org. I also encourage you to toggle between the "Source" and "Visual" tabs as well as you rendered output to get a feel for how text formatting and code chunks work.

> **Key-Point:** The big advantage for us at this stage is that we will have a record of our work and associated notes in a robust transferable format that will enable us to reproduce and automate our analysis and produce a professional looking report for collaborators and GradeScope (more on this later...).

Fundamentals of Quarto

Stop for a hands-on demo from Barry of:

- Rendering quarto documents to different formats,
- Inserting and running code chunks,
- Formating text,
- Visual and Source editing modes,
- What to do if you don't have quarto.

5. Common Plot Types

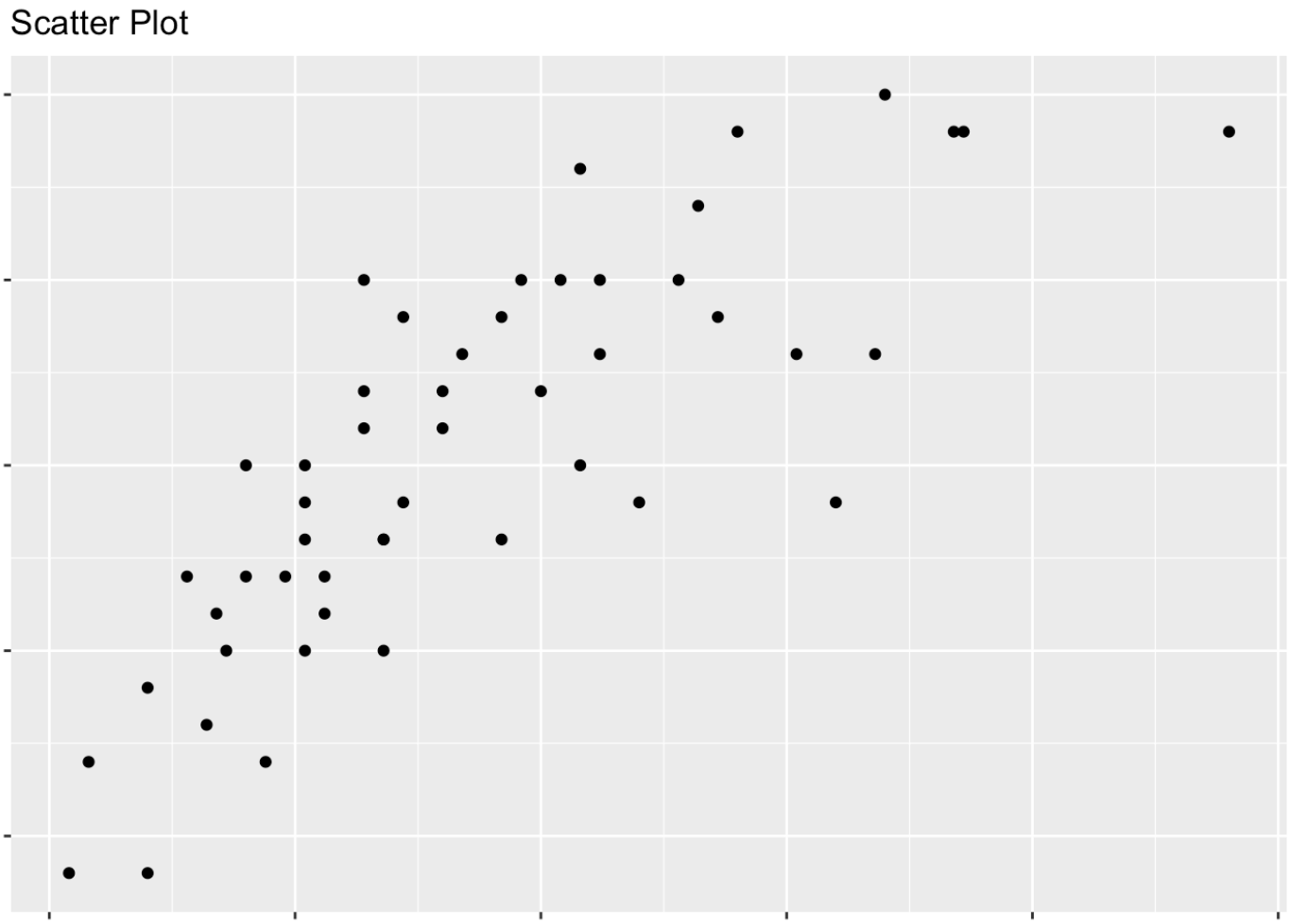
There are many plot types available that can help you understand different features and relationships in your data.

During the exploratory data analysis phase we typically want to detect the most obvious patterns by looking at each variable in isolation or by detecting relationships of variables against

others. The used plot type is also determined by the data type of the input variables like **continuous numeric** or **discrete categorical**.

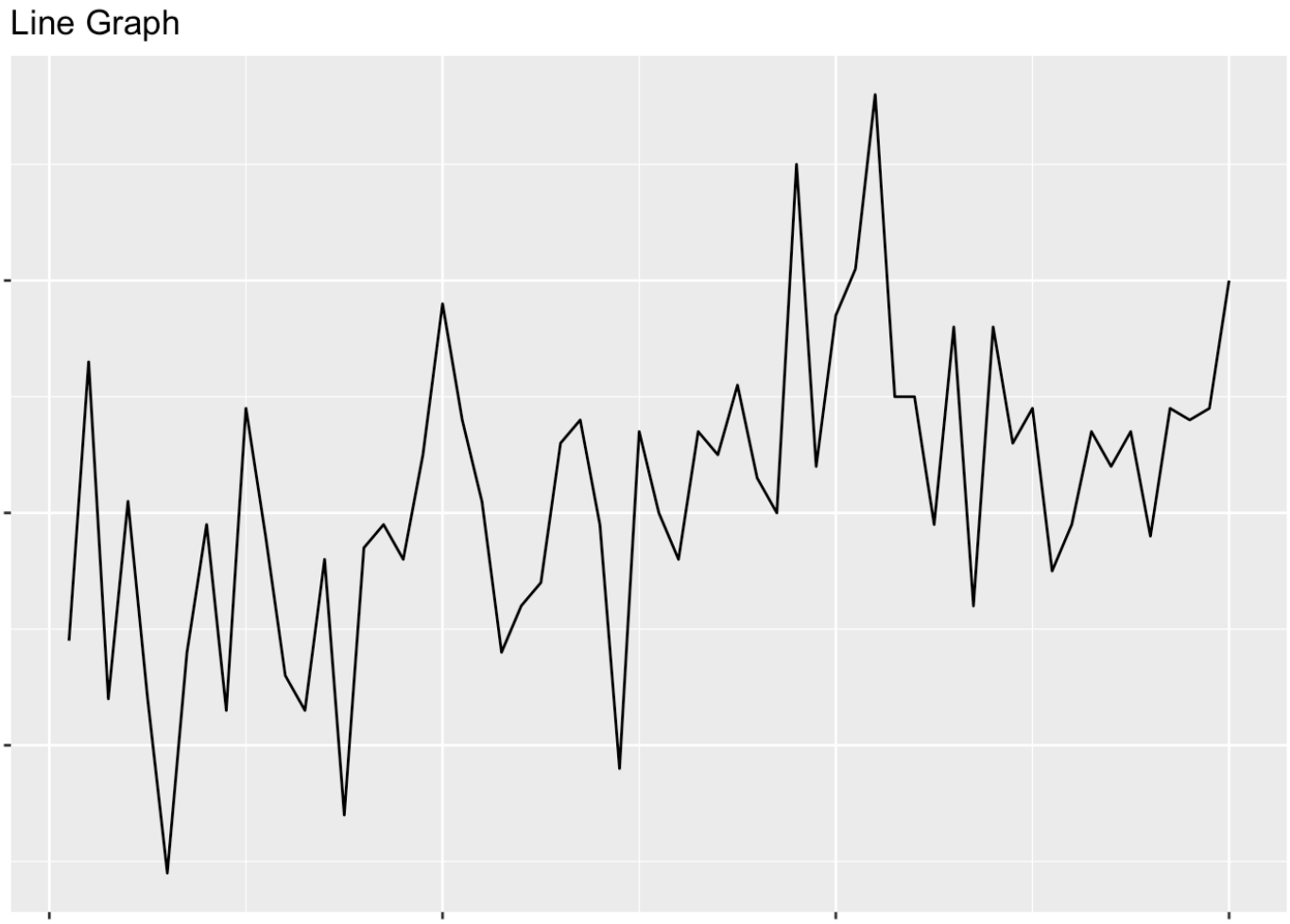
Scatter Plots

Scatter plots are used to visualize the relationship between *two numeric variables*. The position of each point represents the value of the variables on the x and y-axis.



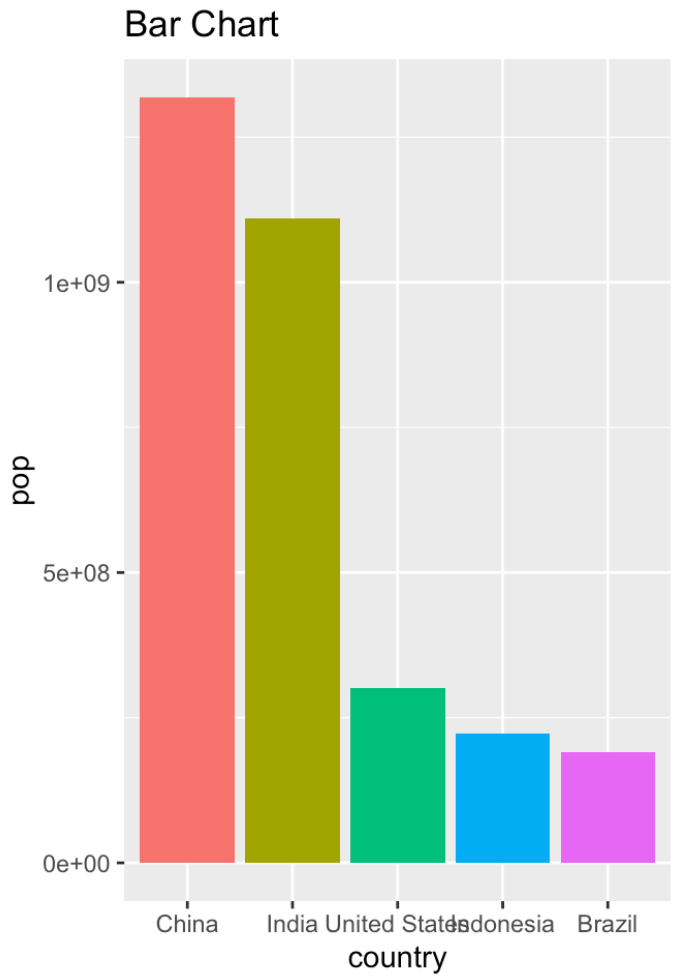
Line Graphs

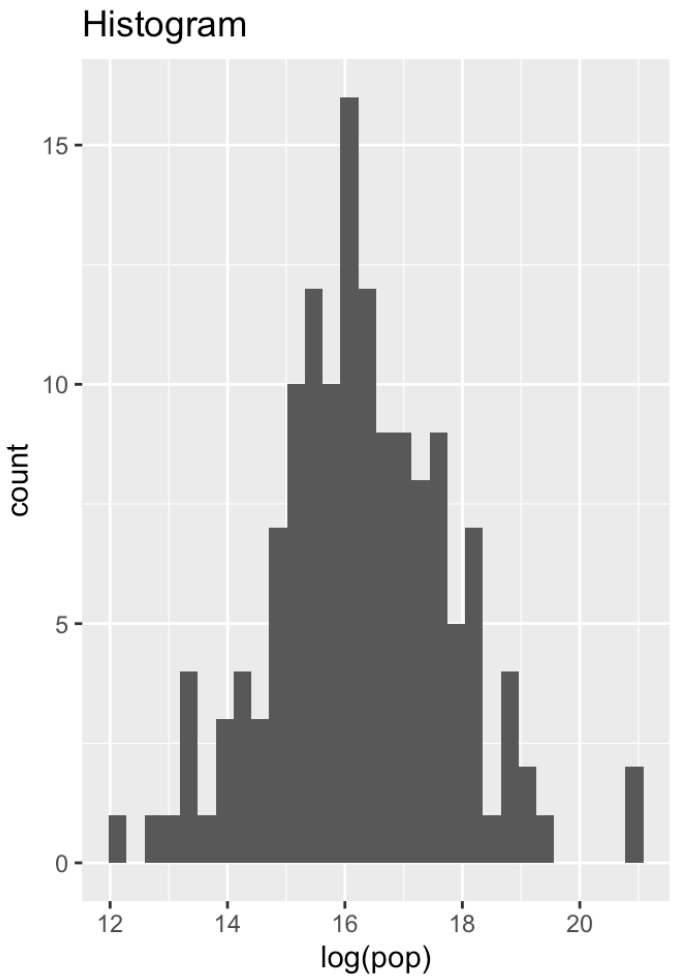
Line graphs are used to visualize the trajectory of one numeric variable against another which are connected through lines. They are well suited if values change continuously - like temperature over time.



Bar Charts and Histograms

Bar charts visualize numeric values grouped by categories. Each category is represented by one bar with a height defined by each numeric value. Histograms are specific bar charts that aim to summarize the number of occurrences of numeric values over a set of value ranges (a.k.a. *bins*). They are typically used to examine the distribution of numeric values.

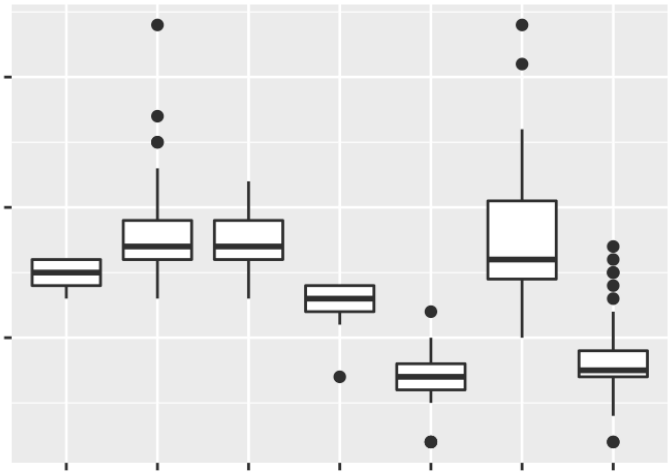
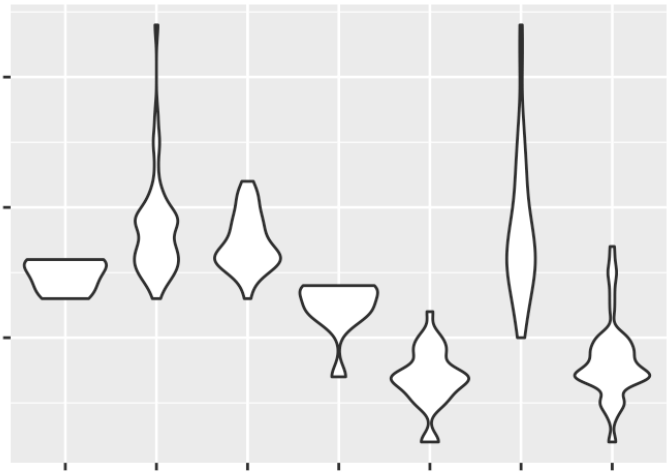
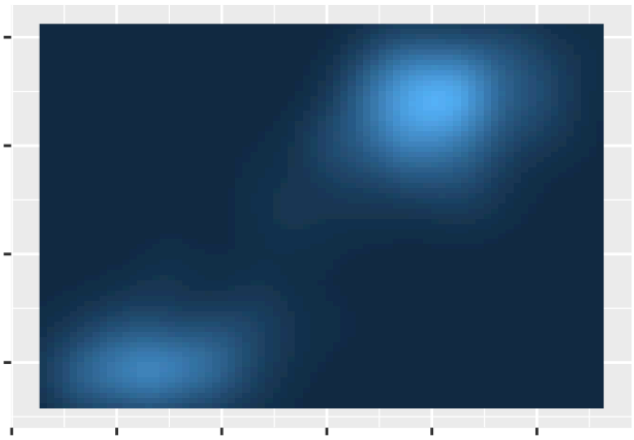
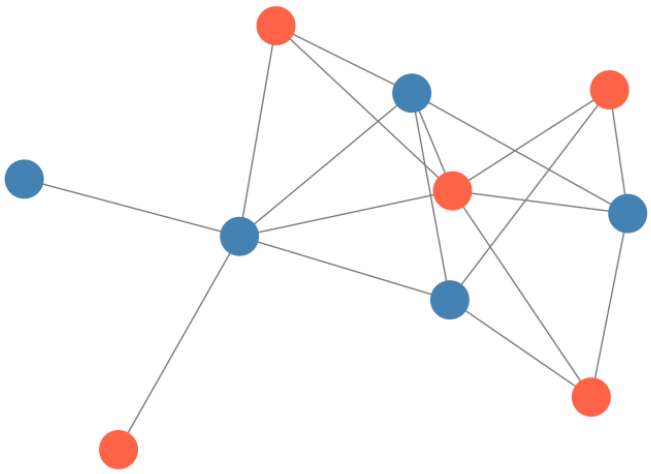




Others

Other frequently used plot types in the biosciences include:

- **Box plots**: Show summary distributional information of numeric values grouped in categories as boxes. Great to quickly compare multiple distributions.
- **Violin plots**: Same as box plots but show distributions as violins.
- **Heat Maps**: Show interactions of variables - typically correlations - as rastered image highlighting areas of high interaction.
- **Network Graphs**: Show connections (as lines - a.k.a. *edges* or vertices) between nodes (typically shown as spheres).
- **Density plots**: A popular alternative to histograms.
- **Dendrograms**: Also known as cluster trees display the results of a clustering analysis (we will have a separate class on these).

- **Q.** Which plot types are typically NOT used to compare distributions of numeric variables?

- **Q.** Which statement about data visualization with ggplot2 is incorrect?

More Details

There are multiple plot types that can be used to examine and compare numeric data distributions. We will examine only a handful of the most common types in this session.

For a detailed discussion and compassion of these approaches see Chapters 7-9 of Claus Wilke's excellent online book Fundamentals of Data Visualization.

Due to the importance of data visualization R offers a very large set of tools and packages in this area. Indeed, the core R language itself provides a rich set of plotting functions and plot types. Personally, I love "base R plots" but I have been using them for many, many years and acknowledge that they are not the easiest for newcomers to pick up quickly. The ggplot2 package is a popular alternative but is only one of many. Particularly noteworthy alternatives include interactive packages like plotly, rgl, and various javascript based R graphics packages.

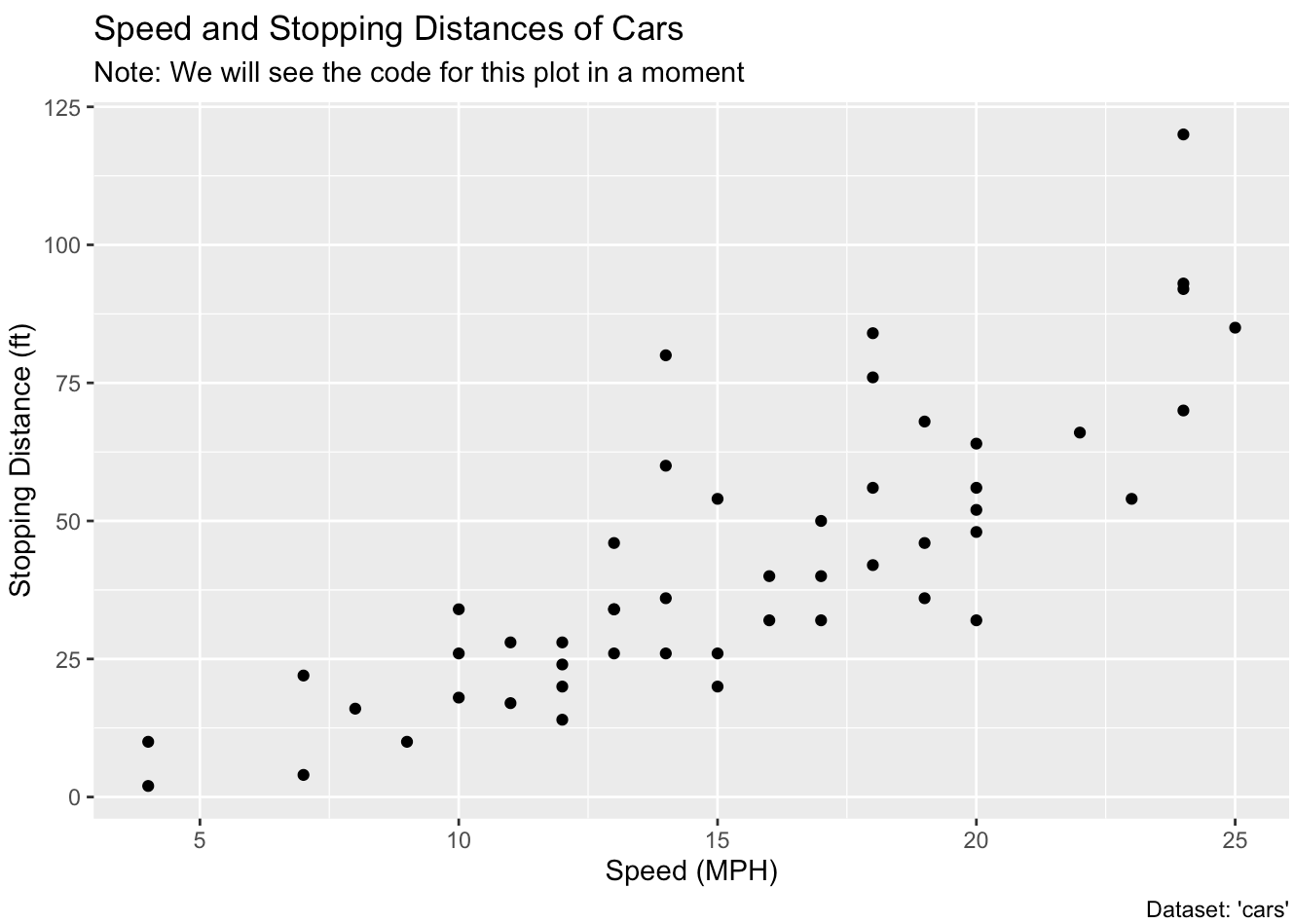
6. Creating Scatter Plots

In this section we will focus on:

- Defining a dataset for your plot using the main `ggplot()` function.
- Specifying how your data maps to plot aesthetics with the `aes()` function.
- Adding geometric layers using the `geom_point()` function.
- Combining the above function calls with `+` operator to make your plot.

Introduction to scatter plots

Scatter plots use points to visualize the relationship between two numeric variables. The position of each point represents the value of the variables on the x- and y-axis. Let's see an example of a scatter plot to understand the relationship between the speed and the stopping distance of cars:

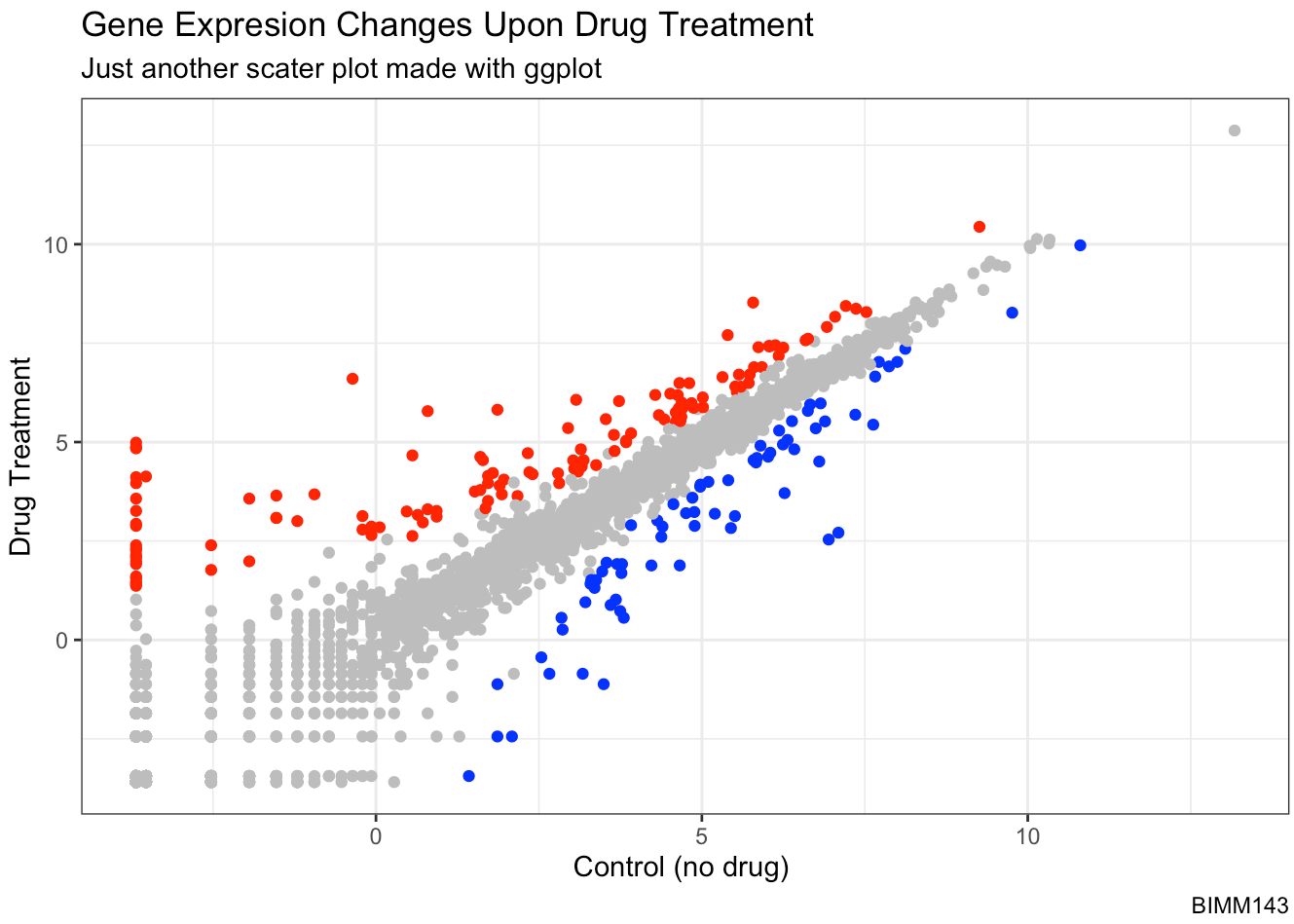


Each point in this plot represents a car. Each car starts to break at a speed given on the bottom x-axis and travels the distance shown on the side y-axis until full stop. If we take a look at all

points in the plot, we can clearly see that it takes faster cars a longer distance until they are completely stopped.

> Scatter plots like this one allow us to visualize the relationship between two numeric variables (in this case speed and stopping distance). The core of this plot is built with only three short lines of code that involve calling three ggplot2 functions as we will see in a moment (`ggplot()`, `aes()` and `geom_point()`).

We will use these same three functions to produce all sorts of scatter plots like the following plot that shows gene expression changes upon treating a particular cell line with a new anit-viral drug:



In this plot points represent individual genes. The red points are for genes that are up-regulated when treated with the drug (i.e. have more expression) and blue points are for down-regulated genes. Gray points are for genes with a non-significant difference in expression values when we look across replicate experiments. We will learn much more about these types of datasets in an upcoming class.

Our focus for this session is to learn how to produce a plot like this. The steps (and R functions) used are the same as those for the previous plot (namely `ggplot()`, `aes()`, and `geom_point()`). Are next sections will focus on each of these in turn. Once we understand these core functions we will be able to make small changes and additions to make all sorts of cool plots.

Specifing a dataset with `ggplot()`

To create plots with ggplot2 you first need to load the package using `library(ggplot2)`.

After the package has been loaded specify the dataset to be used as an argument of the main `ggplot()` function. For example, we specify a plot using the in-built `cars` dataset. You should type (not paste) this code into your R console to see the effect:

```
library(ggplot2)

ggplot(cars)
```

N.B. Note that this command does not plot anything but a blank gray canvas yet. The `ggplot()` function alone just defines the dataset for the plot and creates an empty base on top of which we will add additional layers to build up our plot.

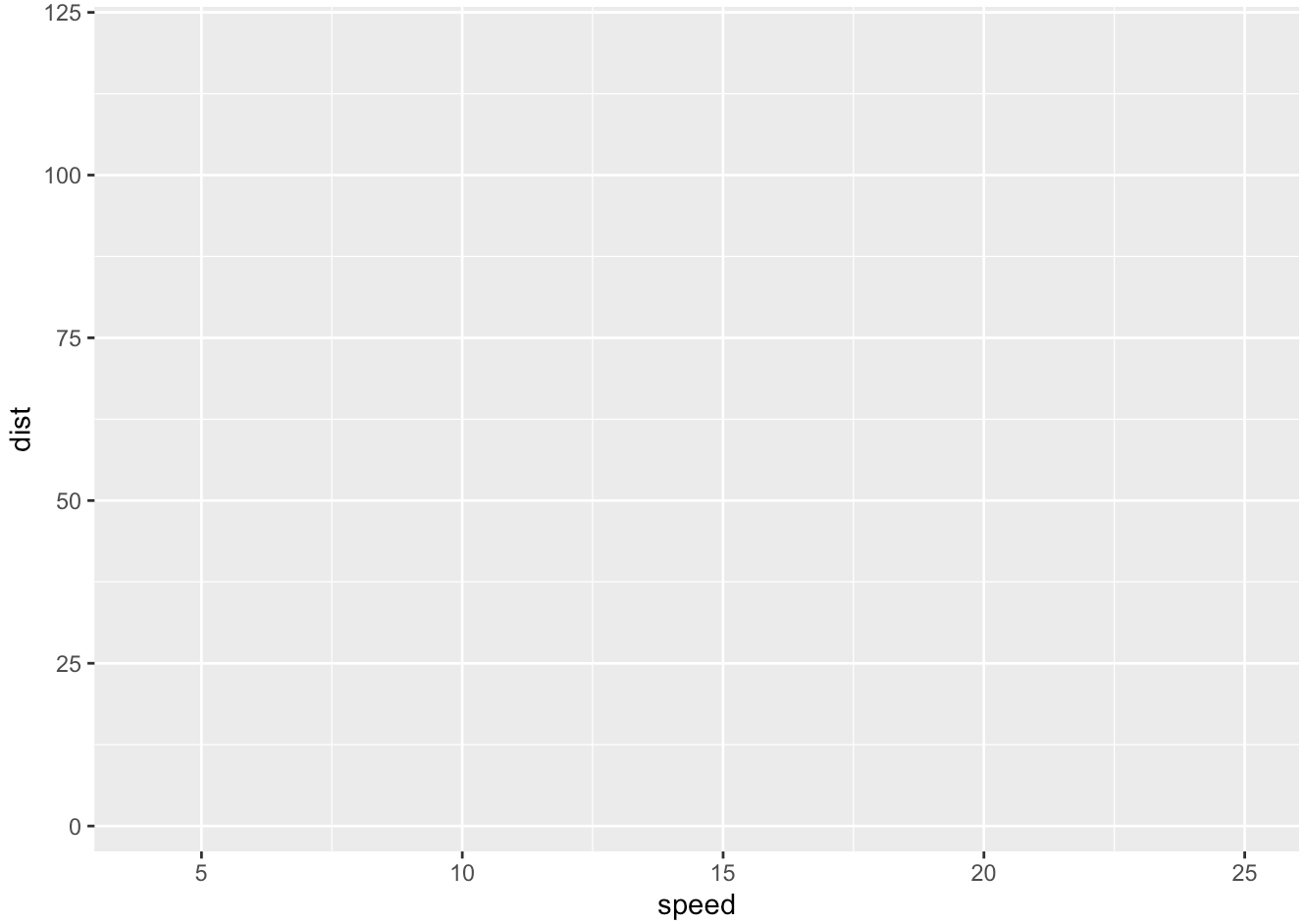
> **Side-note**: If you get an error message at this stage then you likely need to install the ggplot2 package first with `install.packages("ggplot2")`.

Specifing aesthetic mappings with `aes()`

The ggplot2 package uses the concept of aesthetics, which map variables (i.e. columns) from your dataset to the visual features of the plot. The most common aesthetics include `x` and `y` that determine the x- and y-axis coordinates of the points to plot (we will see others further below). The aesthetics are mapped with a call to the `aes()` function.

We will use the columns labeled `speed` and `distance` from the `cars` dataset to set the `x` and `y` aesthetics of our plot. Critically, we combine our call to the `aes()` function with our previous specification of the input dataset with the `ggplot(cars)` function call from above.

```
ggplot(cars) +
  aes(x=speed, y=dist)
```

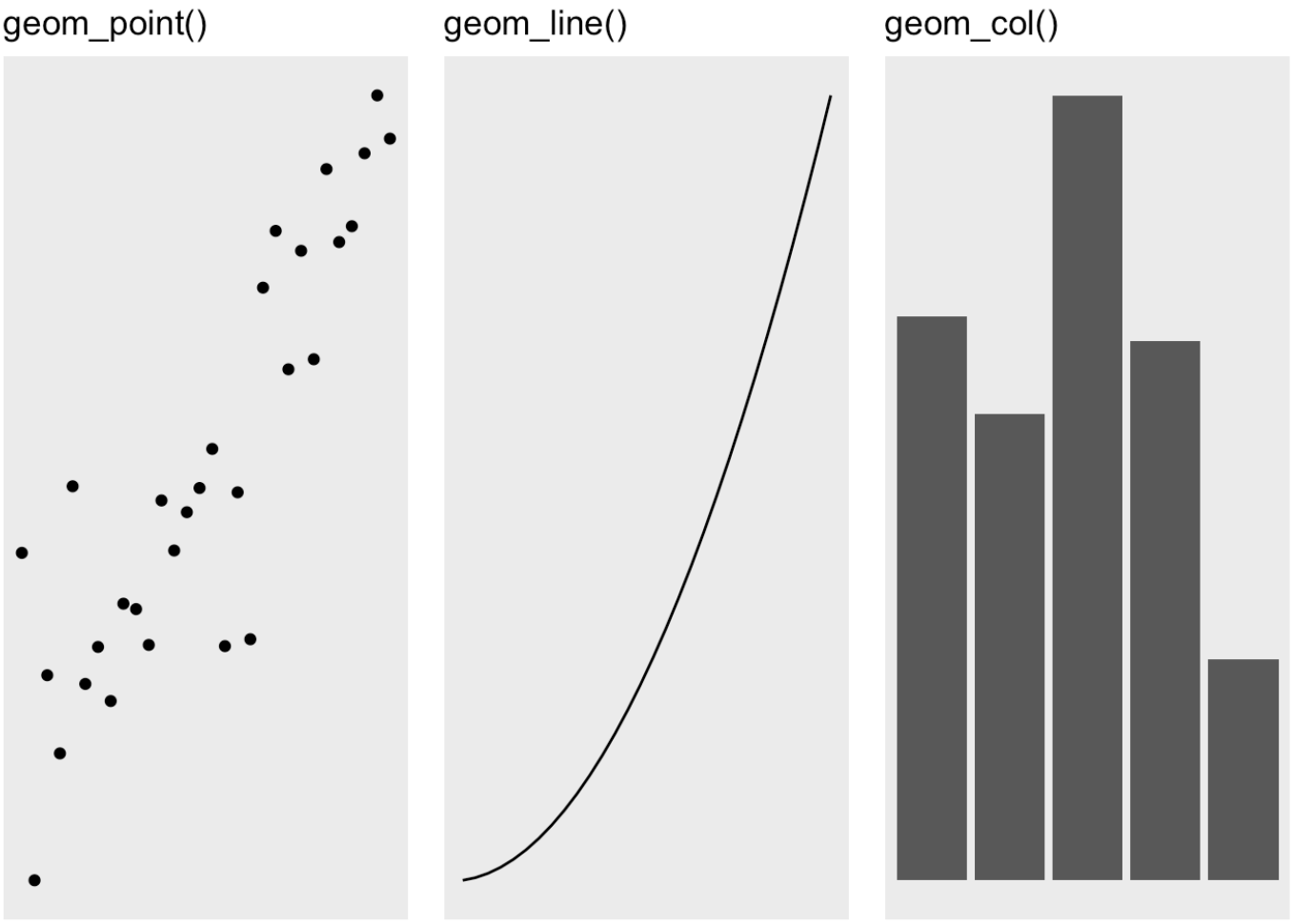


N.B. Note how we combine the `ggplot(cars)` function call and the `aes(x=speed, y=dist)` function calls with the `+` plus sign. We will keep adding more layers to our plot in this same way as we will see in a moment.

As we can see above we don't have any points in our plot yet. Adding them is the job of the `geom_point()` function discussed next.

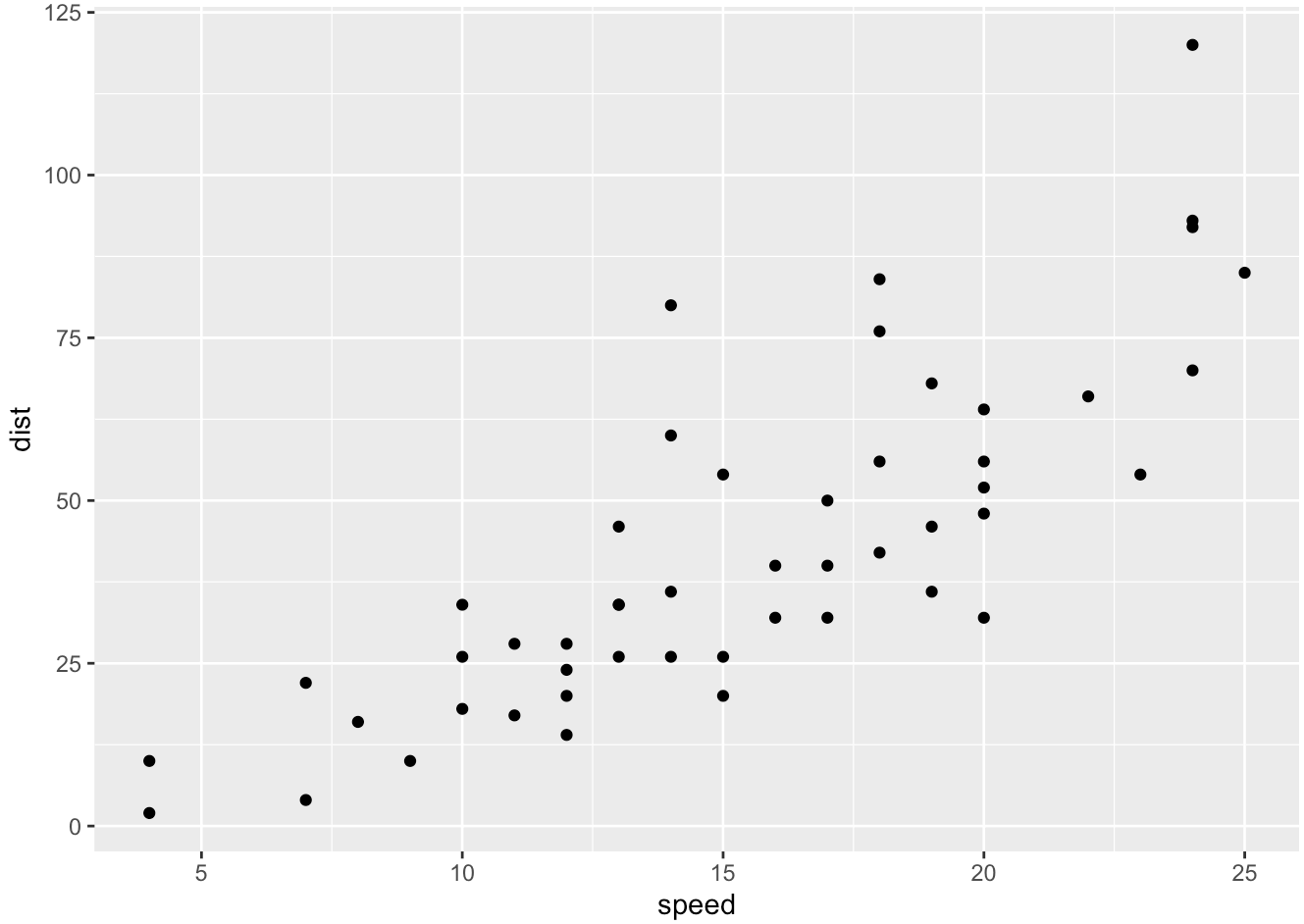
Specifing a geom layer with `geom_point()`

Next we need to add one of ggplot's geometric layers (or **geoms**) to define how we want to visualize our dataset. At this early stage we can thing of geoms as defining the plot type we actually want to produce. For example, `geom_line()` produces a line plot, `geom_bar()` produces a bar plot, `geom_boxplot()` a box plot (more on this later).



Here, we use `geom_point()` to add - you guessed it - points! Let's take our code snippet from above and add a `geom_point()` function call:

```
ggplot(cars) +
  aes(x=speed, y=dist) +
  geom_point()
```



N.B. Note again that we use the `+` operator to connect the `ggplot(cars)` with `aes(x=speed, y=dist)` and now the `geom_point()` lines. Through the linking ggplot knows that the mapped speed and dist variables are taken from the cars dataset. finally the `geom_point()` line instructs ggplot to plot the mapped variables as points.

> **Side-note**: The required steps to create a scatter plot with ggplot can be summarized as follows:
>
> - Load the package ggplot2 using `library(ggplot2)`.
> - Specify the dataset to be plotted using `ggplot()`.
> - Use the `+` operator to add layers to the plot.
> - Map variables from the dataset to aesthetic properties through the `aes()` function.
> - Add a geometric layer with to define the shapes to be plotted. In the case of scatter plots, use `geom_point()`.

- **Q.** Which geometric layer should be used to create scatter plots in ggplot2?
- **Q.** In your own RStudio can you add a trend line layer to help show the relationship between the plot variables with the `geom_smooth()` function?

Solution

- **Q.** Argue with `geom_smooth()` to add a straight line from a linear model without the shaded standard error region?

Solution

- **Q.** Can you finish this plot by adding various label annotations with the `labs()` function and changing the plot look to a more conservative "black & white" theme by adding the `theme_bw()` function:

Solution

Adding more plot aesthetics through `aes()`

In their most basic form scatter plots can only visualize datasets in two dimensions through the `x` and `y` aesthetics passed to the `geom_point()` layer. However, most data sets have more than two variables and thus might require additional plotting dimensions. Ggplot makes it very easy to map additional variables to different plotting aesthetics like *size*, *transparency alpha* and *color*.

Here we will cover how to:

- Adjust the point size of a scatter plot using the `size` parameter.
- Change the point color of a scatter plot using the `color` parameter.
- Set a parameter `alpha` to change the transparency of all points.

We will also try to stress an important point that is often confusing for newcomers to ggplot: How to differentiate between *aesthetic mappings* (plot features you want mapped to variables in your data) and *constant parameters* (specifications of plot features you want to remain the same or otherwise come from elsewhere - i.e. not from your data).

Let's turn for a moment to more relevant example data set. The code below reads the results of a differential expression analysis where a new anti-viral drug is being tested.

```
url <- "https://bioboot.github.io/bimm143_S20/class-material/up_down_expression.t
genes <- read.delim(url)
head(genes)
```

```
        Gene Condition1 Condition2      State
1      A4GNT -3.6808610 -3.4401355 unchanging
2       AAAS  4.5479580  4.3864126 unchanging
3      AASDH  3.7190695  3.4787276 unchanging
```

```
4       AATF  5.0784720  5.0151916 unchanging
5       AATK  0.4711421  0.5598642 unchanging
6 AB015752.4 -3.6808610 -3.5921390 unchanging
```

- **Q.** Use the `nrow()` function to find out how many genes are in this dataset. What is your answer? ☐
- **Q.** Use the `colnames()` function and the `ncol()` function on the `genes` data frame to find out what the column names are (we will need these later) and how many columns there are. How many columns did you find? ☐
- **Q.** Use the `table()` function on the `State` column of this data.frame to find out how many 'up' regulated genes there are. What is your answer? ☐
- **Q.** Using your values above and 2 significant figures. What fraction of total genes is up-regulated in this dataset? ☐

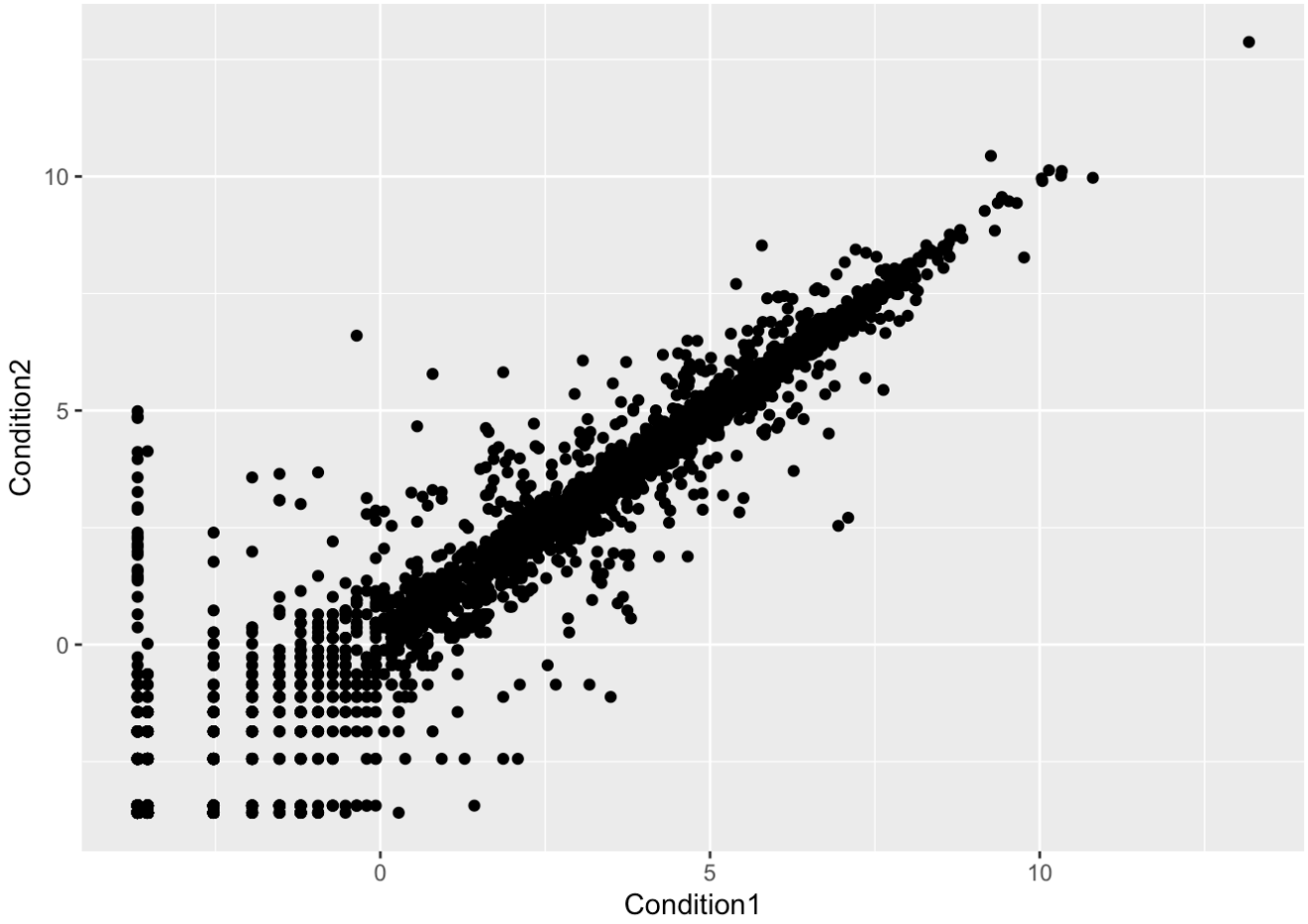
More Details

We can make a first basic scatter plot of this dataset by following the same recipe we have already seen, namely:

- Pass the `genes` data.frame as input to the `ggplot()` function.
- Then use the `aes()` function to set the `x` and `y` aesthetic mappings to the `Condition1` and `Condition2` columns.
- Finally add a `geom_point()` layer to add points to the plot.
- Don't forget to add layers step-wise with the `+` operator at the end of each line.

- **Q.** Complete the code below to produce the following plot

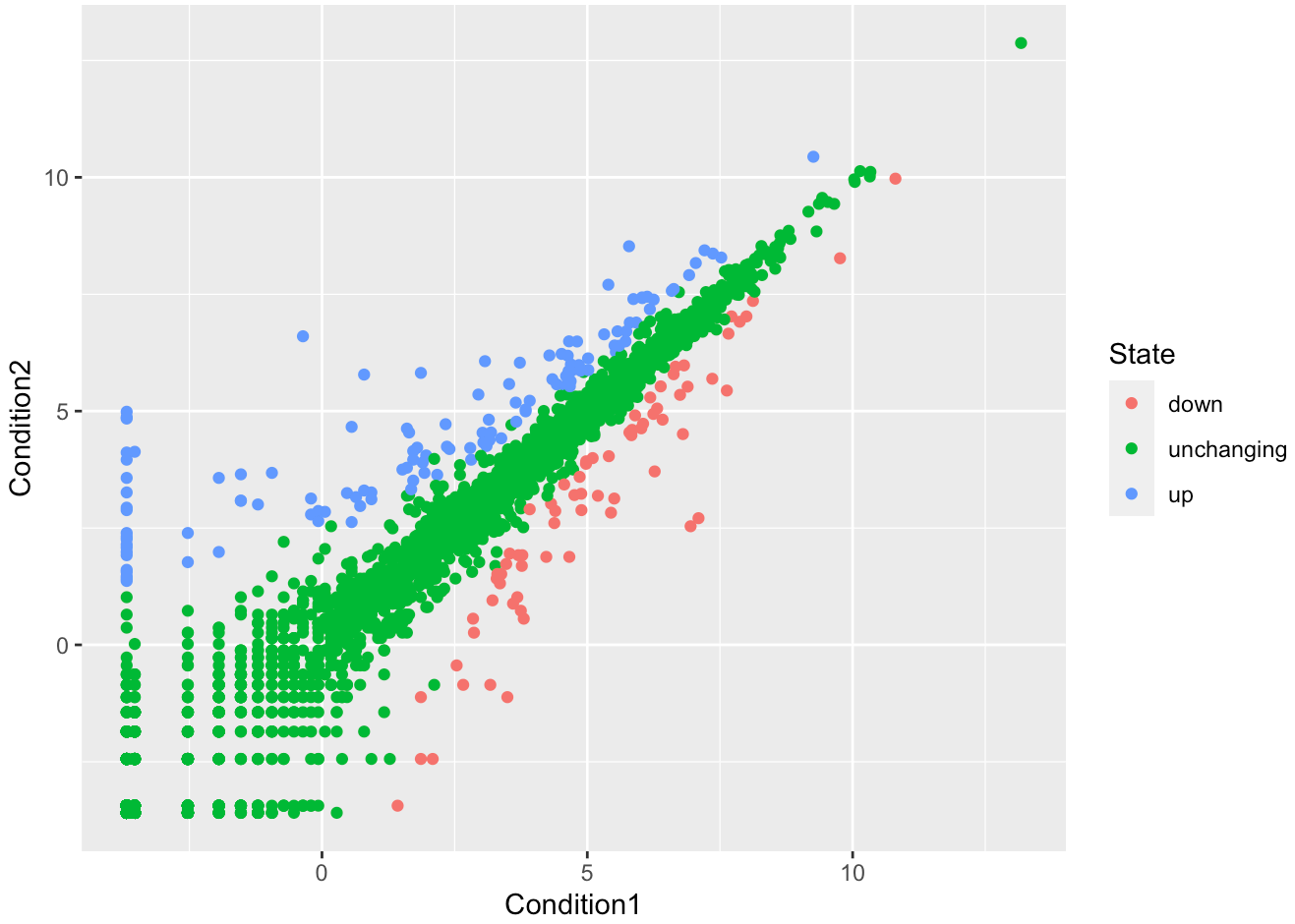
```
ggplot(___) +
    aes(x=Condition1, y=___) _
    _____
```



Solution

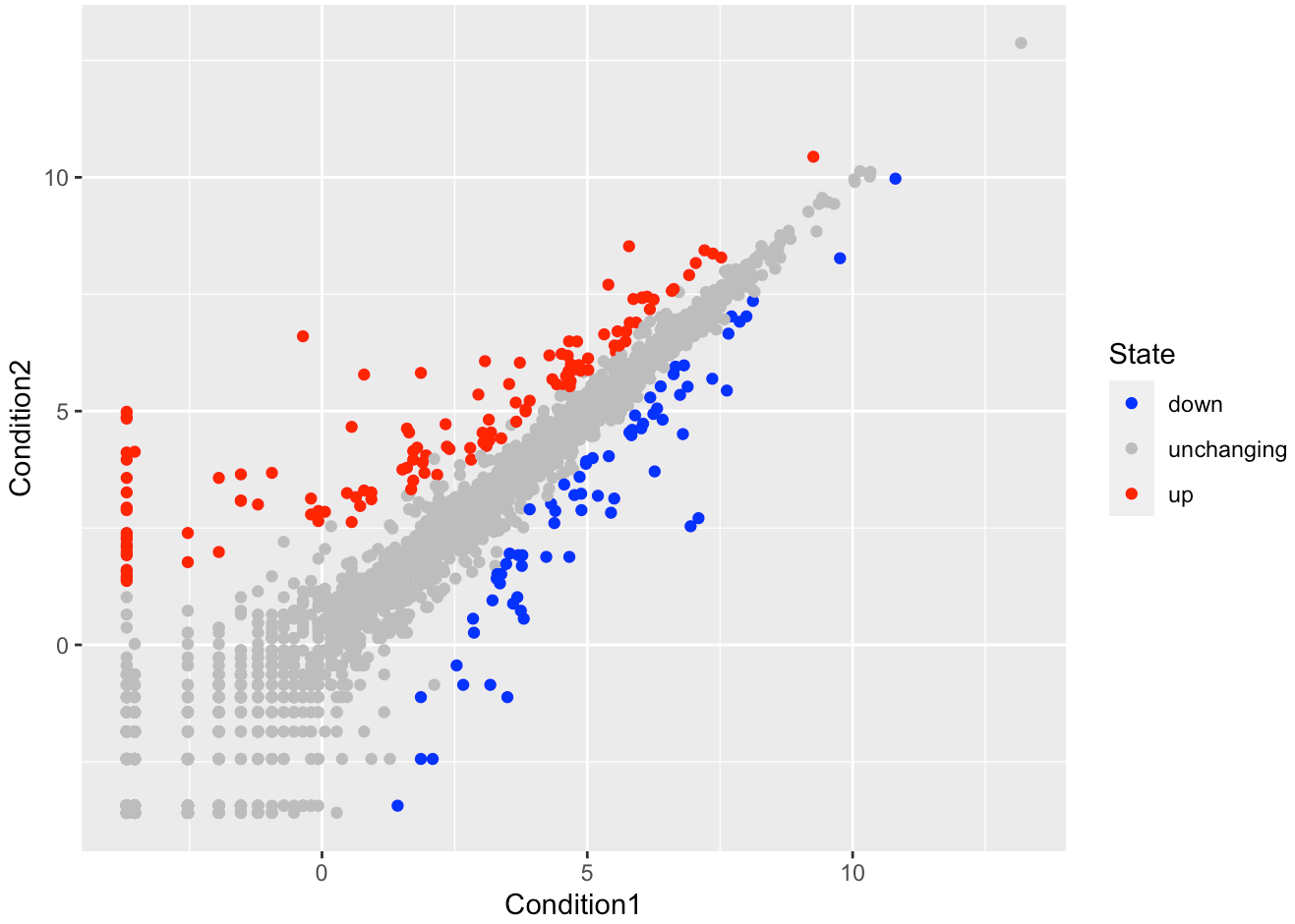
There is extra information in this dataset, namely the `State` column, which tells us whether the difference in expression values between conditions is statistically significant. Let's map this column to point color:

```
p <- ggplot(genes) +
    aes(x=Condition1, y=Condition2, col=State) +
    geom_point()
p
```

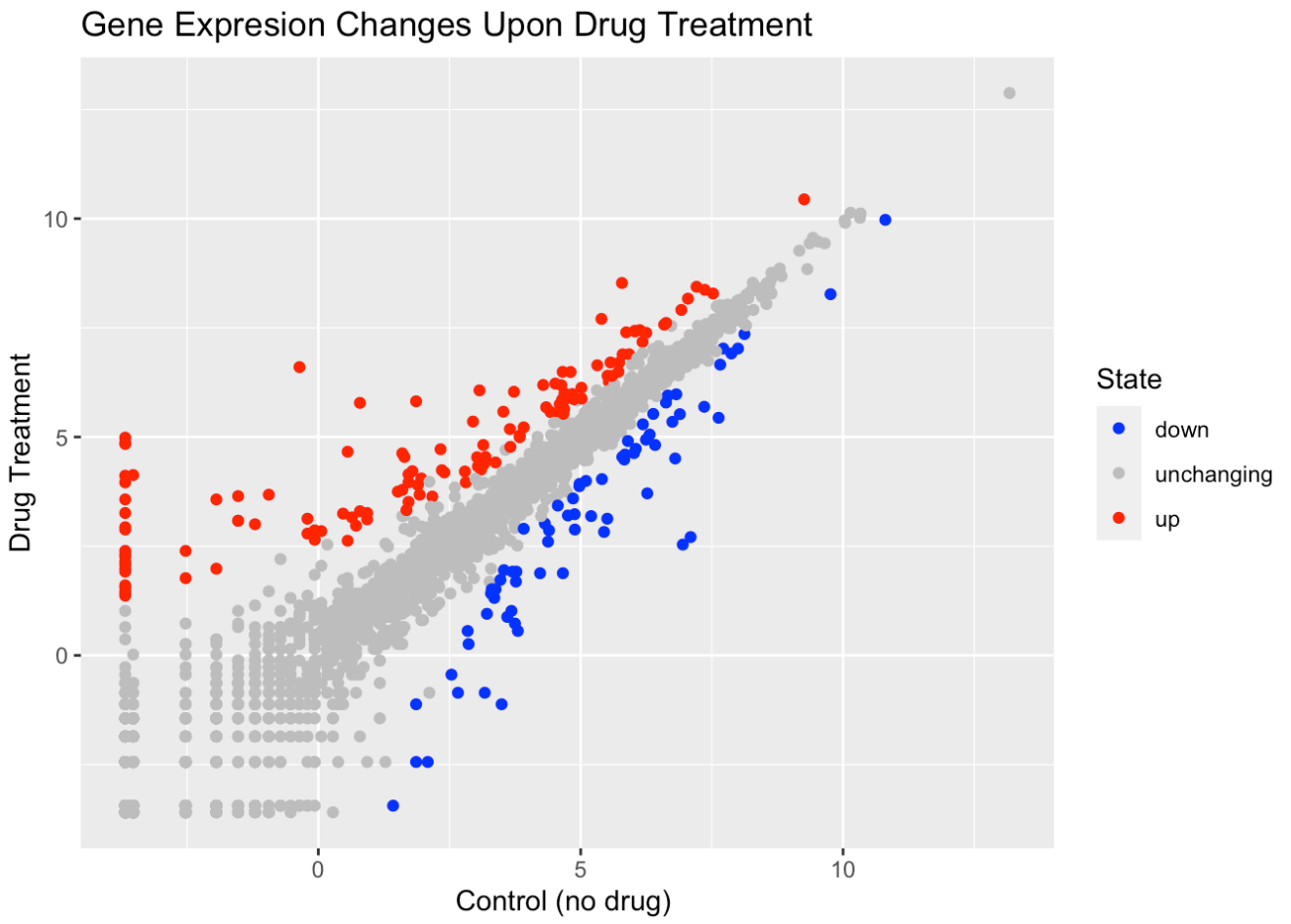


I am not a big fan of these default colors so let's change them up by adding another layer to explicitly specifcy our color scale. Note how we saved our previous plot as the object `p` and can use it now to add more layers:

```
p + scale_colour_manual( values=c("blue","gray","red") )
```



- **Q.** Nice, now add some plot annotations to the `p` object with the `labs()` function so your plot looks like the following:



Solution

7. Going Further

The following sections are considered **optional extensions** for motivated students.

The **gapminder** dataset contains economic and demographic data about various countries since 1952. This dataset features in your DataCamp course for this week and you can find out more about the portion we are using here.

The data itself is available as either a tab-delimited file online, or via the `gapmider` package. You can use whichever method you are more comfortable with to obtain the dataset. I show both below:

```
install.packages("gapminder")
library(gapminder)
```

Or read the TSV file from online:

```
# File location online
url <- "https://raw.githubusercontent.com/jennybc/gapminder/master/inst/extdata/g

gapminder <- read.delim(url)
```

This dataset covers many years and many countries. Before we make some plots we will use some **dplyr** code to focus in on a single year. You can install the **dplyr** package with the command `install.packages("dplyr")`. We will learn more about **dplyr** in next weeks class. For now, feel free to just copy and paste the following line that takes `gapmider` data frame and filters to contain only the rows with a `year` value of 2007.

```
# install.packages("dplyr")  ## un-comment to install if needed
library(dplyr)

gapminder_2007 <- gapminder %>% filter(year==2007)
```

Let's consider the `gapminder_2007` dataset which contains the variables GDP per capita `gdpPercap` and life expectancy `lifeExp` for 142 countries in the year 2007

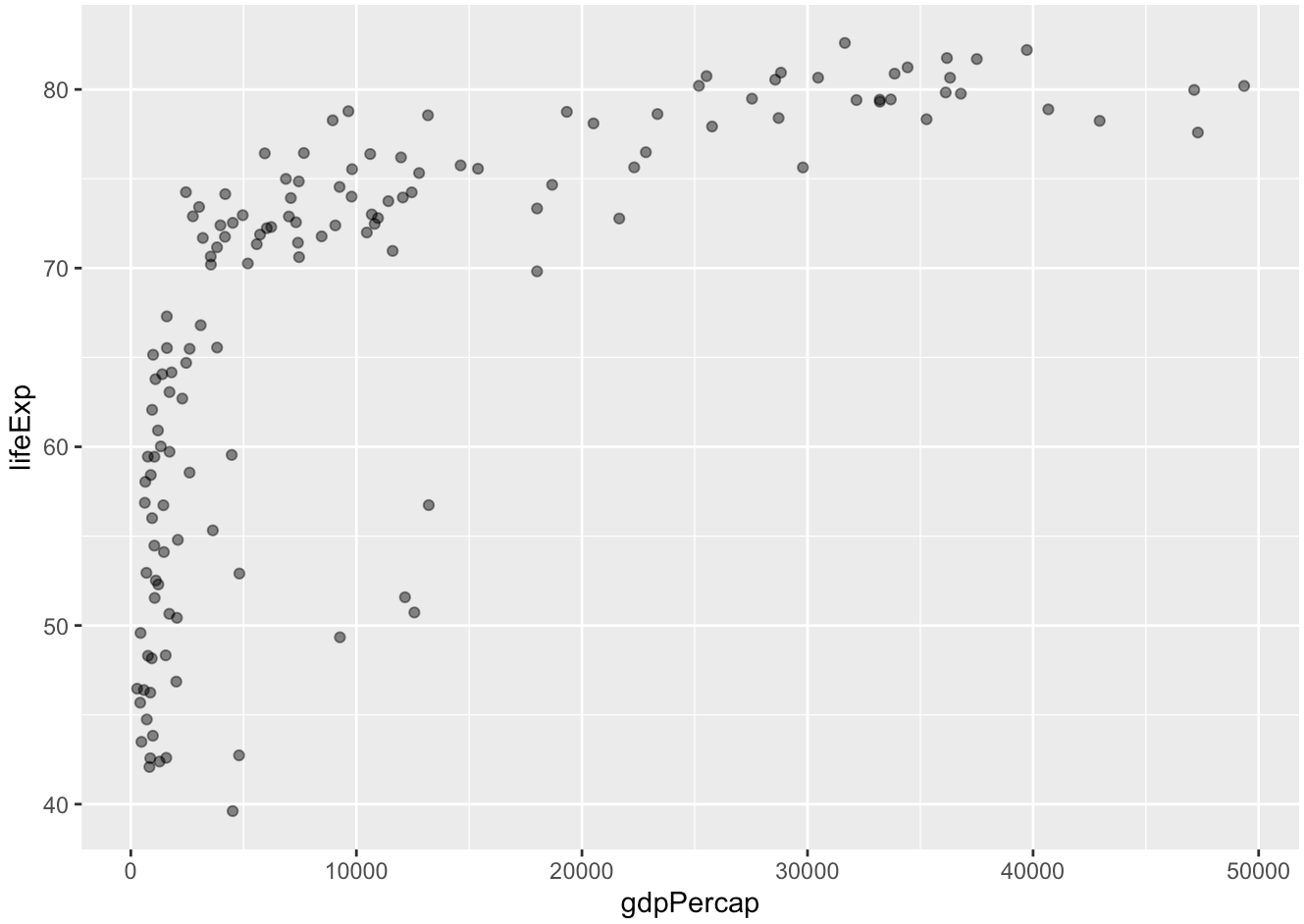
- **Q.** Complete the code below to produce a first basic scater plot of this `gapminder_2007` dataset:

```
ggplot(gapminder_2007) +
  aes(x=___, y=___) +
  ___
```

Solution

There are quite a few points that are nearly on top of each other in the above plot. One useful approach here is to add an `alpha=0.4` argument to your `geom_point()` call to make the points slightly transparent. This will help us see things a little more clearly later on.

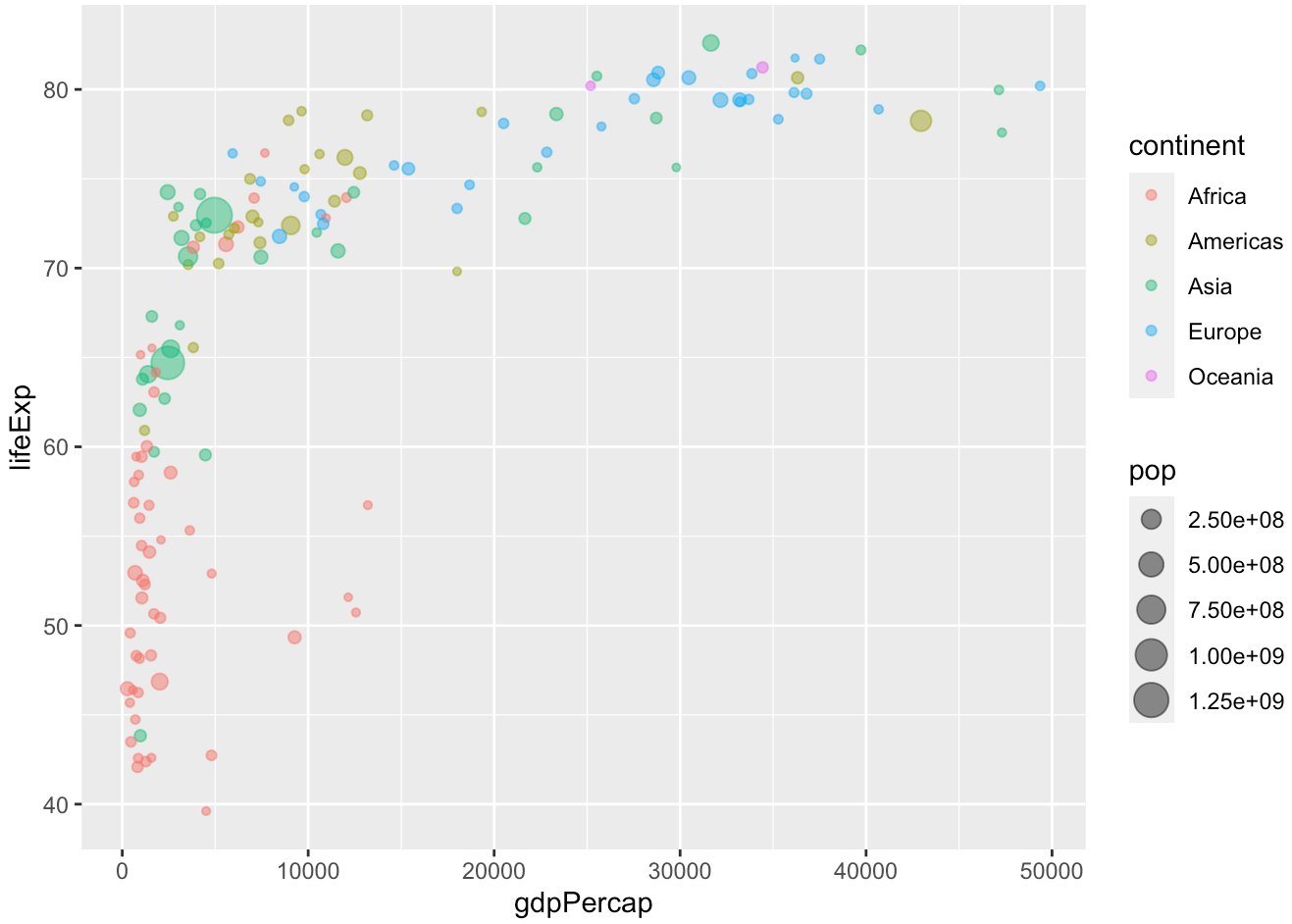
```
ggplot(gapminder_2007) +
  aes(x=gdpPercap, y=lifeExp) +
  geom_point(alpha=0.5)
```



Adding more varables to `aes()`

By mapping the `continent` variable to the point `color` aesthetic and the population `pop` (in millions) through the point `size` argument to `aes()` we can obtain a much richer plot that now includes 4 different variables from the data set:

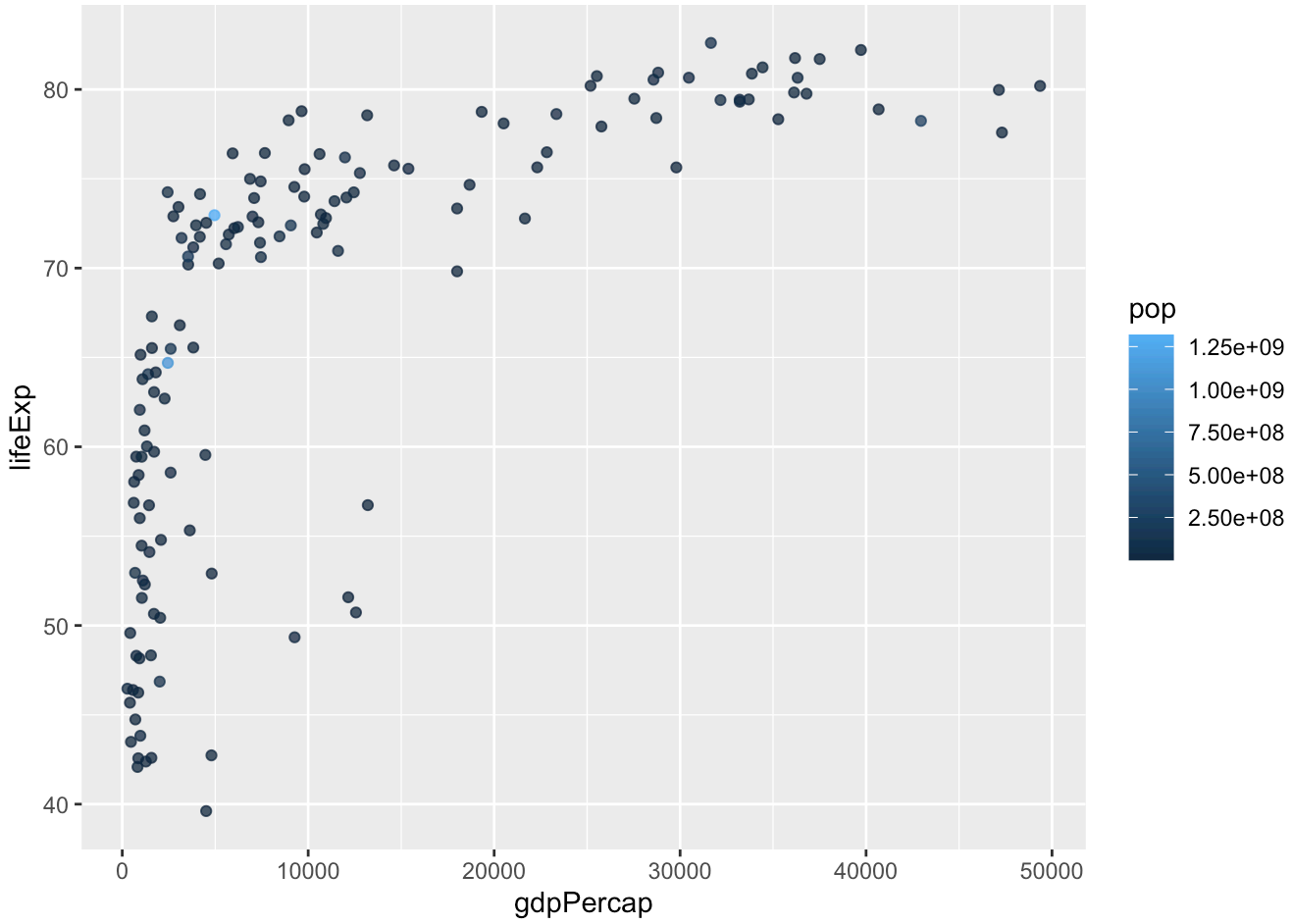
```
ggplot(gapminder_2007) +
  aes(x=gdpPercap, y=lifeExp, color=continent, size=pop) +
  geom_point(alpha=0.5)
```



N.B. Here each new aesthetic we add results in a new dimension to our scatter plot. We see that in the resulting plot each point is colored differently based on the continent of each country. ggplot uses the coloring scheme based on the categorical data type of the variable continent.

By contrast, let's see how the plot looks like if we color the points by the numeric variable population pop:

```
ggplot(gapminder_2007) +
  aes(x = gdpPercap, y = lifeExp, color = pop) +
  geom_point(alpha=0.8)
```

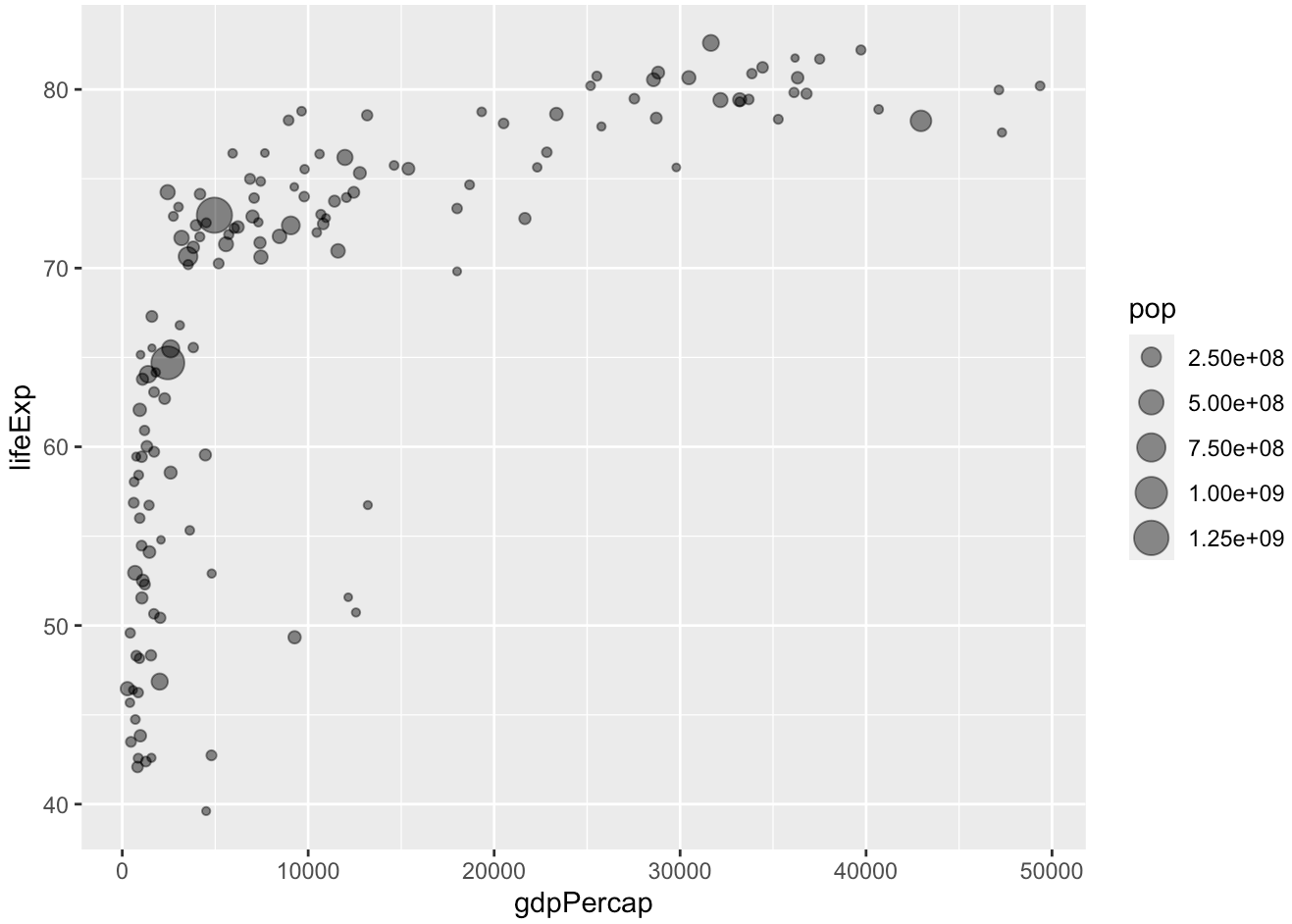


The scale immediately changes to continuous as can be seen in the legend and the light-blue points are now the countries with the highest population number (China and India).

Adjusting point size

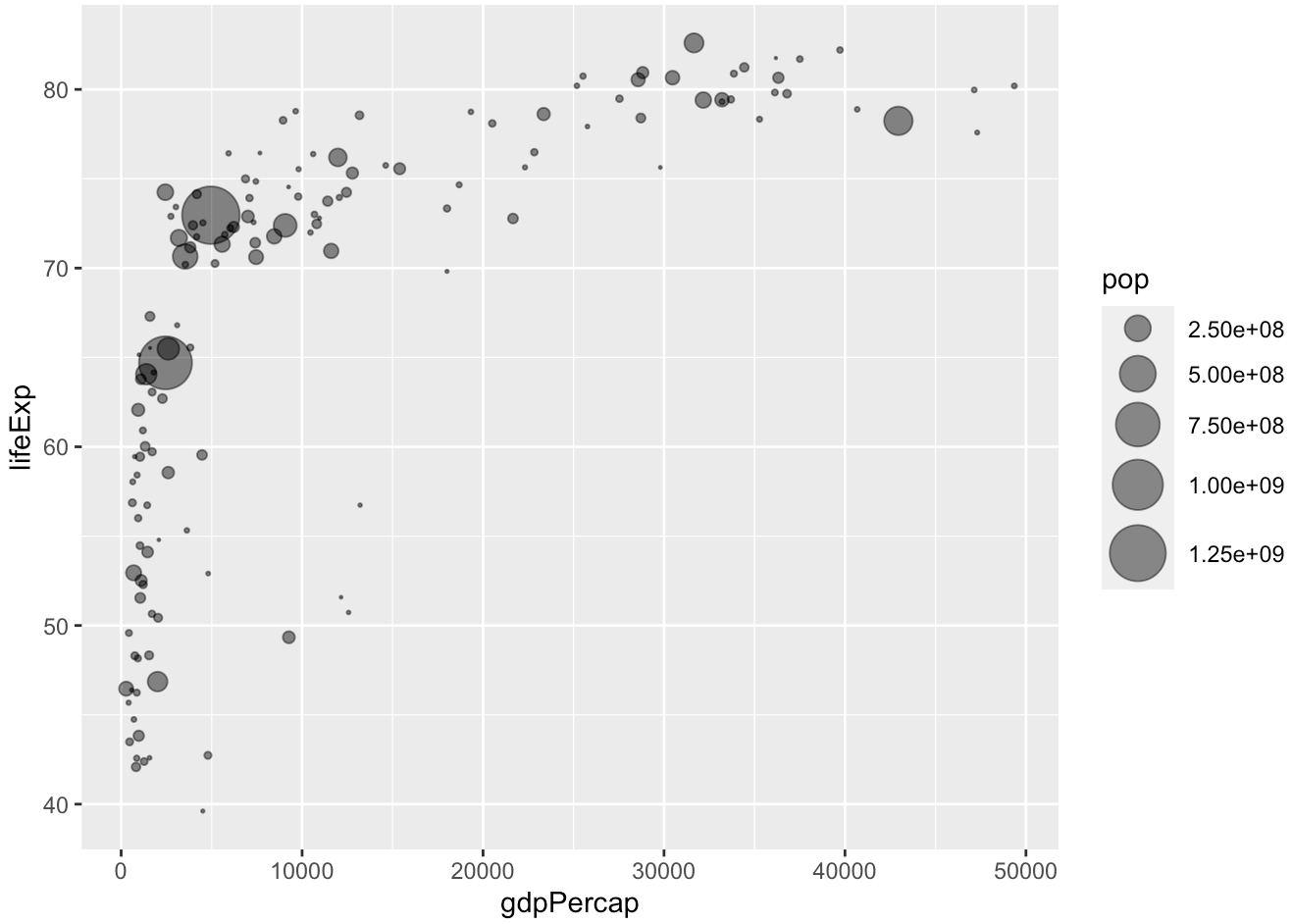
For the gapminder_2007 dataset we can plot the GDP per capita (`x=gdpPercap`) vs. the life expectancy (`y=lifeExp`) and set the point `size` based on the population (`size=pop`) of each country we can use:

```
ggplot(gapminder_2007) +
  aes(x = gdpPercap, y = lifeExp, size = pop) +
  geom_point(alpha=0.5)
```



However, if you look closely we see that the point sizes in the plot above do not clearly reflect the population differences in each country. If we compare the point size representing a population of 250 million people with the one displaying 750 million, we can see, that their sizes are not proportional. Instead, the point sizes are binned by default. To reflect the actual population differences by the point size we can use the `scale_size_area()` function instead. The scaling information can be added like any other ggplot object with the `+` operator:

```
ggplot(gapminder_2007) +
  geom_point(aes(x = gdpPercap, y = lifeExp,
                 size = pop), alpha=0.5) +
  scale_size_area(max_size = 10)
```



Note that we have adjusted the point's max_size which results in bigger point sizes.

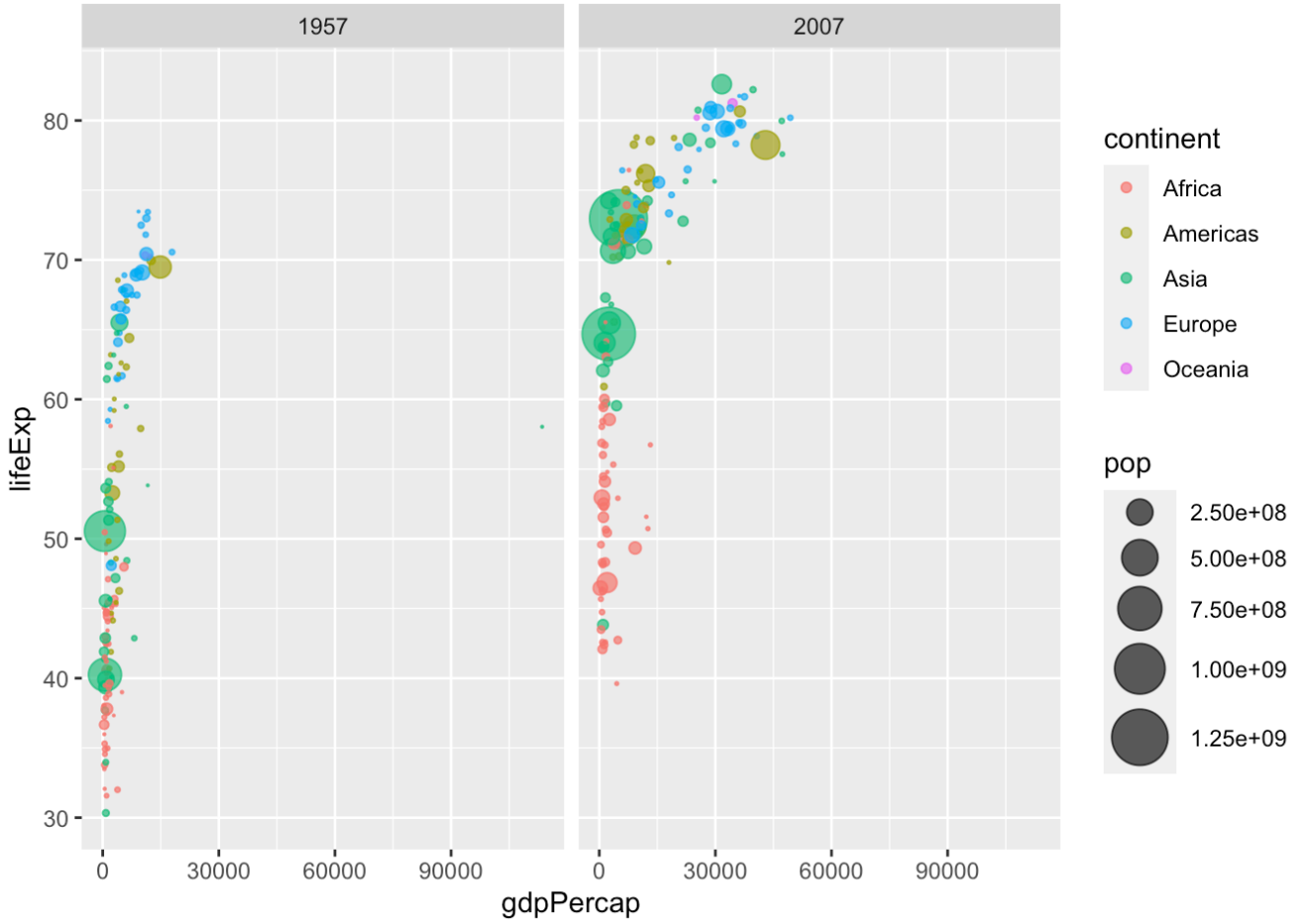
-**Q.** Can you adapt the code you have learned thus far to reproduce our gapminder scatter plot for the year 1957? What do you notice about this plot is it easy to compare with the one for 2007?

Steps to produce your 1957 plot should include:

- Use dplyr to `filter` the `gapmider` dataset to include only the year 1957 (check above for how we did this for 2007).
- Save your result as `gapminder_1957`.
- Use the `ggplot()` function and specify the `gapminder_1957` dataset as input
- Add a `geom_point()` layer to the plot and create a scatter plot showing the GDP per capita `gdpPercap` on the x-axis and the life expectancy `lifeExp` on the y-axis
- Use the `color` aesthetic to indicate each `continent` by a different color
- Use the `size` aesthetic to adjust the point size by the population pop
- Use `scale_size_area()` so that the point sizes reflect the actual population differences and set the `max_size` of each point to 15 -Set the opacity/transparency of each point to 70% using the `alpha=0.7` parameter

Solution

Q. Do the same steps above but include 1957 and 2007 in your input dataset for `ggplot()`. You should now include the layer `facet_wrap(~year)` to produce the following plot:



Solution

8. OPTIONAL: Bar Charts

In this section we will cover:

- How to create bar charts using `geom_col()`.
- How to fill bars with color using the `fill` aesthetic.
- Order your bars by the trend you are most interested in.
- Flip (or rotate) your plots to help with presentation clarity.
- Use alternative plot types when bar charts become too crowded.

Introduction to bar charts

Bar charts visualize numeric values grouped by categories. Each category is represented by one bar with a height defined by each numeric value.

Bar charts are well suited to compare values among different groups e.g. number of votes by parties, number of people in different countries or GDP per capita in different countries. Bar charts are a bit spacious and work best if the number of groups to compare is rather small.

Below you can find an example showing the number of people (in millions) in the five biggest countries by population in 2007:

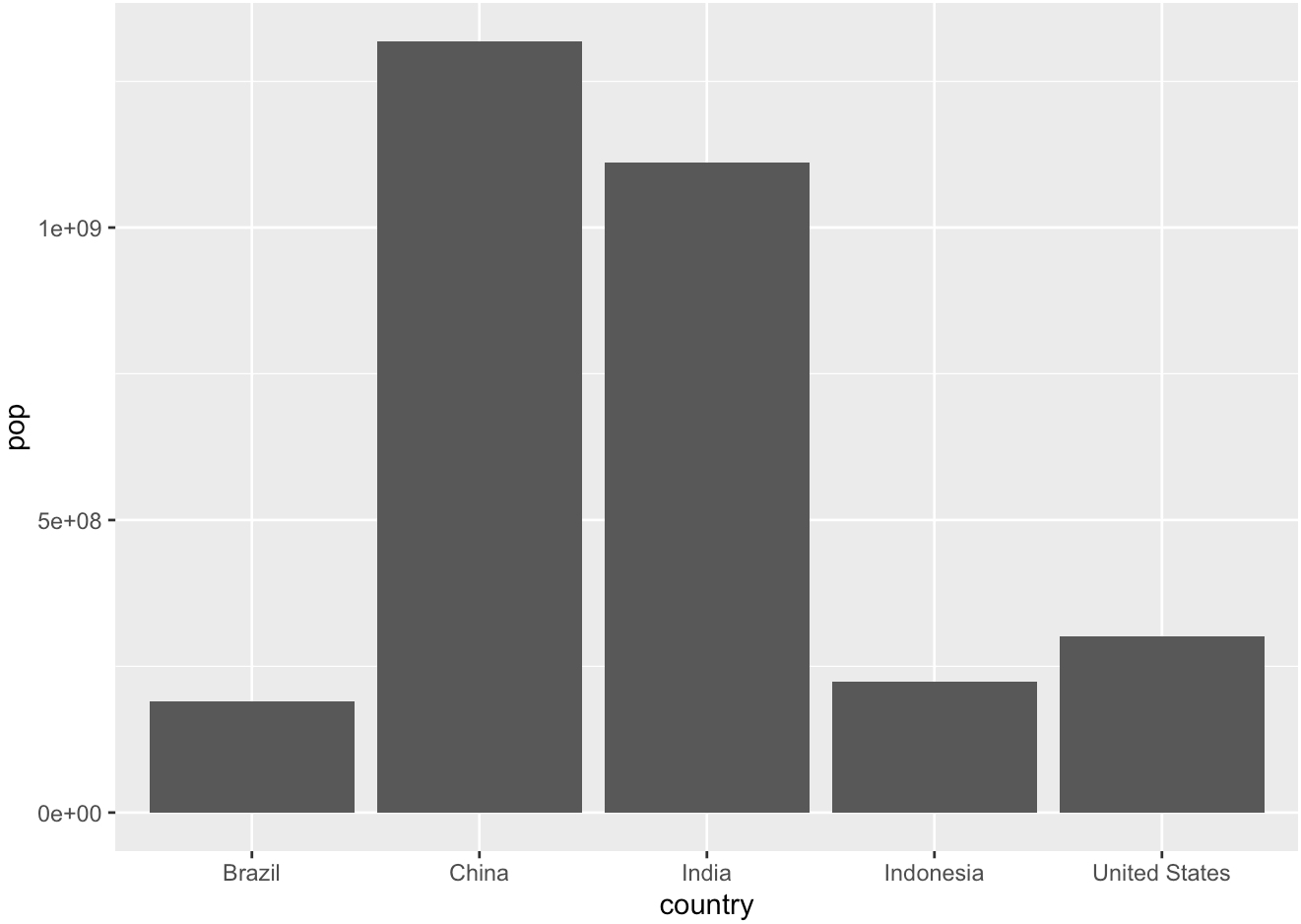
```
gapminder_top5 <- gapminder %>%
  filter(year==2007) %>%
  arrange(desc(pop)) %>%
  top_n(5, pop)

gapminder_top5
```

```
        country continent year lifeExp        pop gdpPercap
1         China      Asia 2007  72.961 1318683096  4959.115
2         India      Asia 2007  64.698 1110396331  2452.210
3 United States  Americas 2007  78.242  301139947 42951.653
4     Indonesia      Asia 2007  70.650  223547000  3540.652
5        Brazil  Americas 2007  72.390  190010647  9065.801
```

Creating a simple bar chart

In ggplot2, bar charts are created using the `geom_col()` geometric layer. The `geom_col()` layer requires the x aesthetic mapping which defines the different bars to be plotted. The height of each bar is defined by the variable specified in the y aesthetic mapping. Both mappings, x and y are required for `geom_col()`. Let's create our first bar chart with the `gapminder_top5` dataset. It contains population (in millions) and life expectancy data for the biggest countries by population in 2007.

```
ggplot(gapminder_top5) +
  geom_col(aes(x = country, y = pop))
```



We see that the resulting bars are sorted by the country names in alphabetical order by default.

Q Create a bar chart showing the life expectancy of the five biggest countries by population in 2007.

Tips

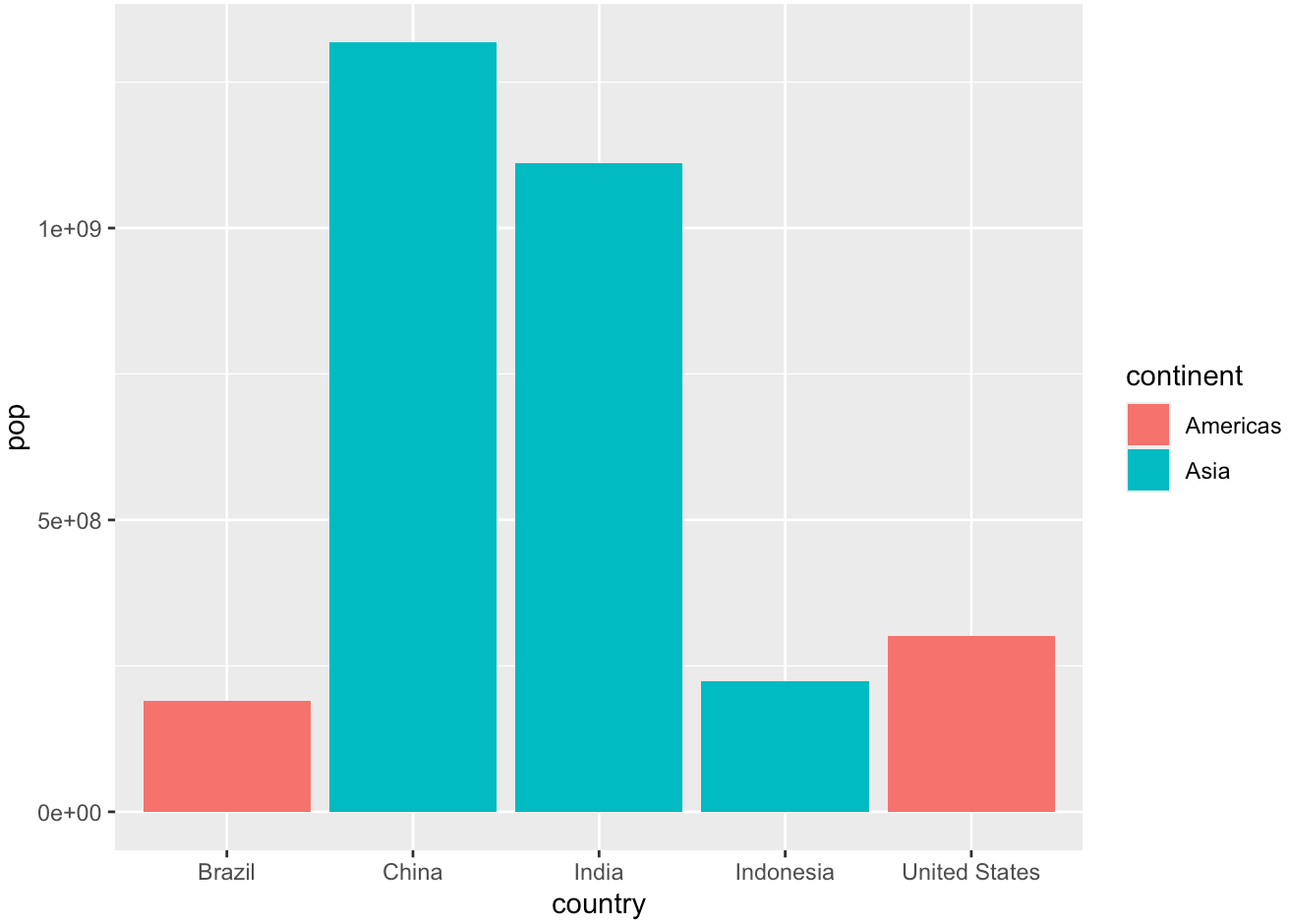
Filling bars with color

Like other geoms `geom_col()` allows users to map additional dataset variables to the color attribute of the bar. The `fill` aesthetic can be used to fill the entire bars with color. A usual confusion is the `color` aesthetic which specifies the line color of each bar's border instead of the `fill` color.

Based on the `gapminder_top5` dataset we plot the population (in millions) of the biggest countries and use the continent variable to color each bar:

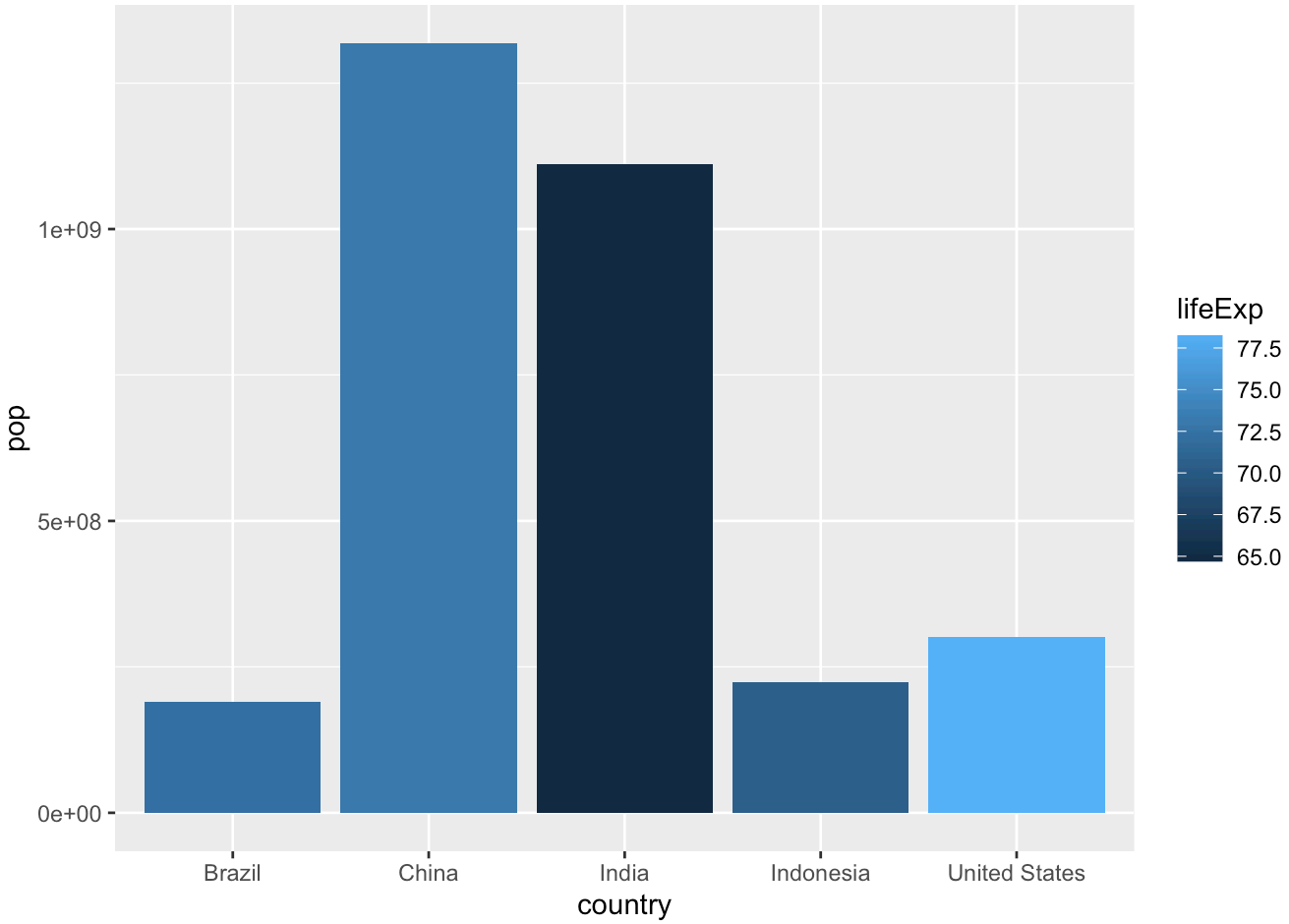
```
ggplot(gapminder_top5) +
```

```
  geom_col(aes(x = country, y = pop, fill = continent))
```



Since the `continent` variable is a *categorical variable* the bars have a clear color scheme for each continent. Let's see what happens if we use a `numeric` variable like life expectancy `lifeExp` instead:

```
ggplot(gapminder_top5) +
  geom_col(aes(x = country, y = pop, fill = lifeExp))
```

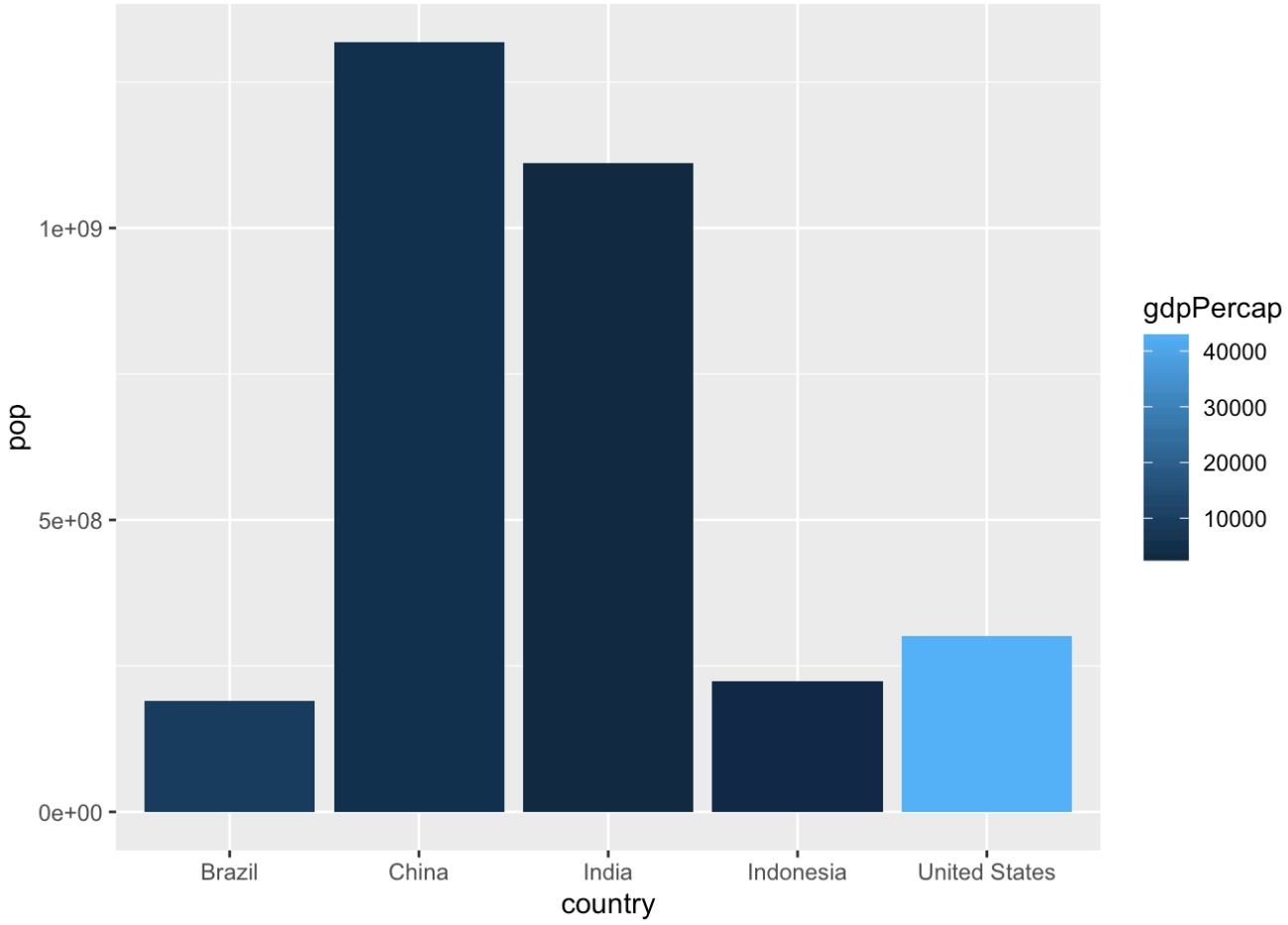


The bar colors have now changed according the **continuous** legend on the right. We see that also `numeric` variables can be used to `fill` bars.

Q. Plot population size by country. Create a bar chart showing the population (in millions) of the five biggest countries by population in 2007.

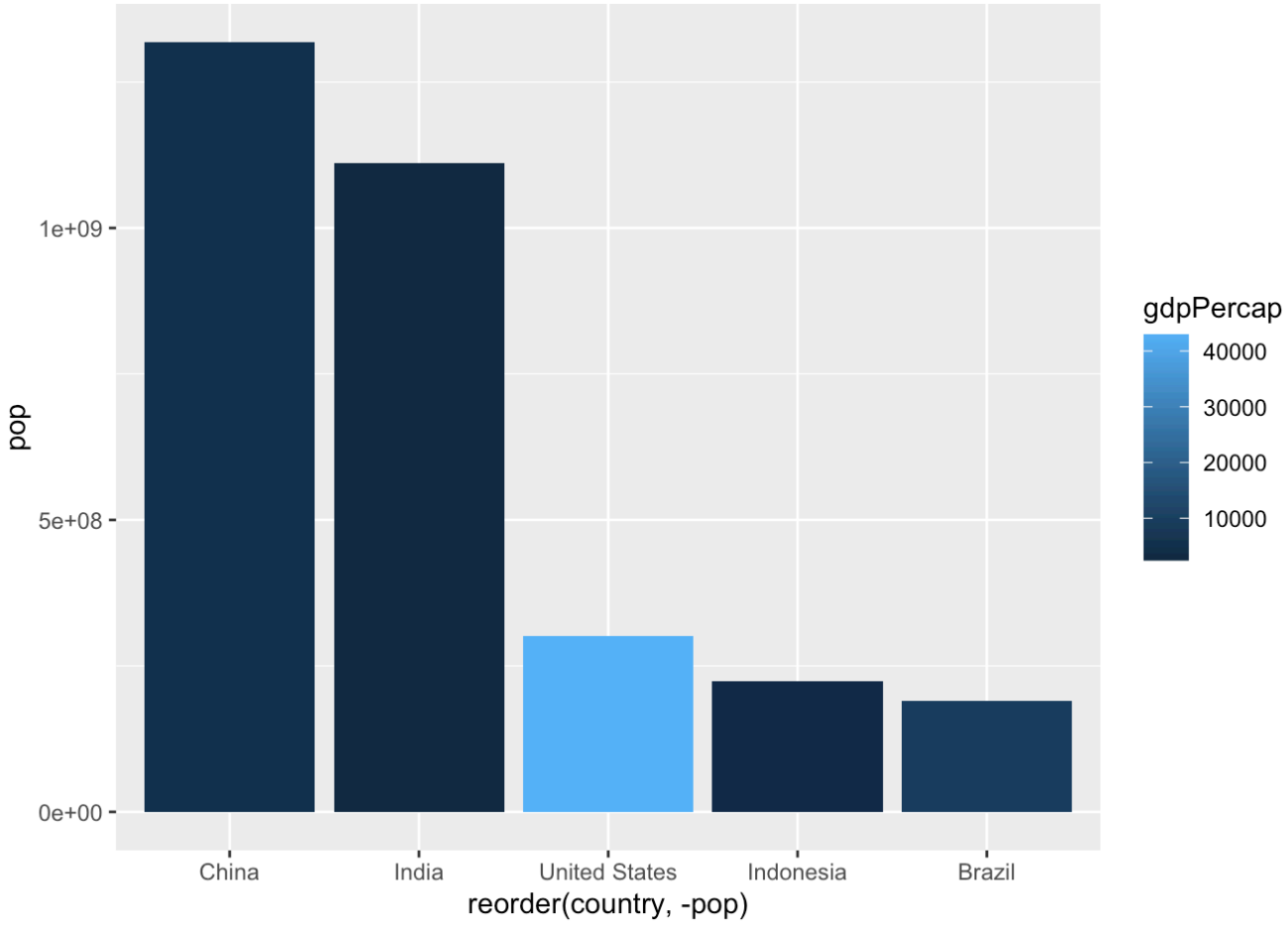
- Use the `ggplot()` function and specify the `gapminder_top5` dataset as input
- Add a `geom_col()` layer to the plot
- Plot one bar for each `country` (`x` aesthetic)
- Use population `pop` as bar height (`y` aesthetic)
- Use the GDP per capita `gdpPercap` as `fill` aesthetic

```
ggplot(gapminder_top5) +
  aes(x=country, y=pop, fill=gdpPercap) +
  geom_col()
```



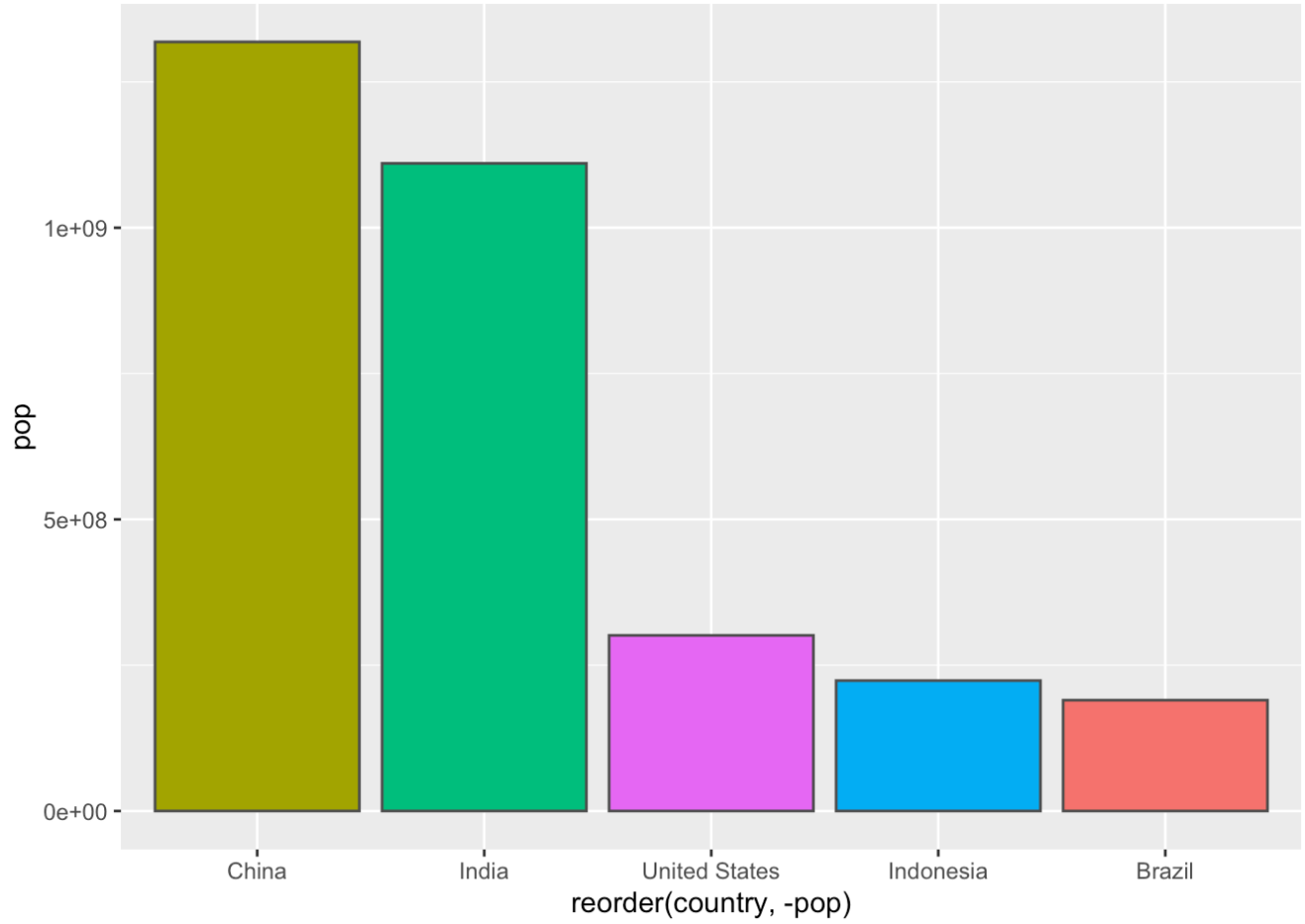
And change the order of the bars

```
ggplot(gapminder_top5) +
  aes(x=reorder(country, -pop), y=pop, fill=gdpPercap) +
  geom_col()
```



and just fill by country

```
ggplot(gapminder_top5) +
  aes(x=reorder(country, -pop), y=pop, fill=country) +
  geom_col(col="gray30") +
  guides(fill="none")
```



Flipping bar charts

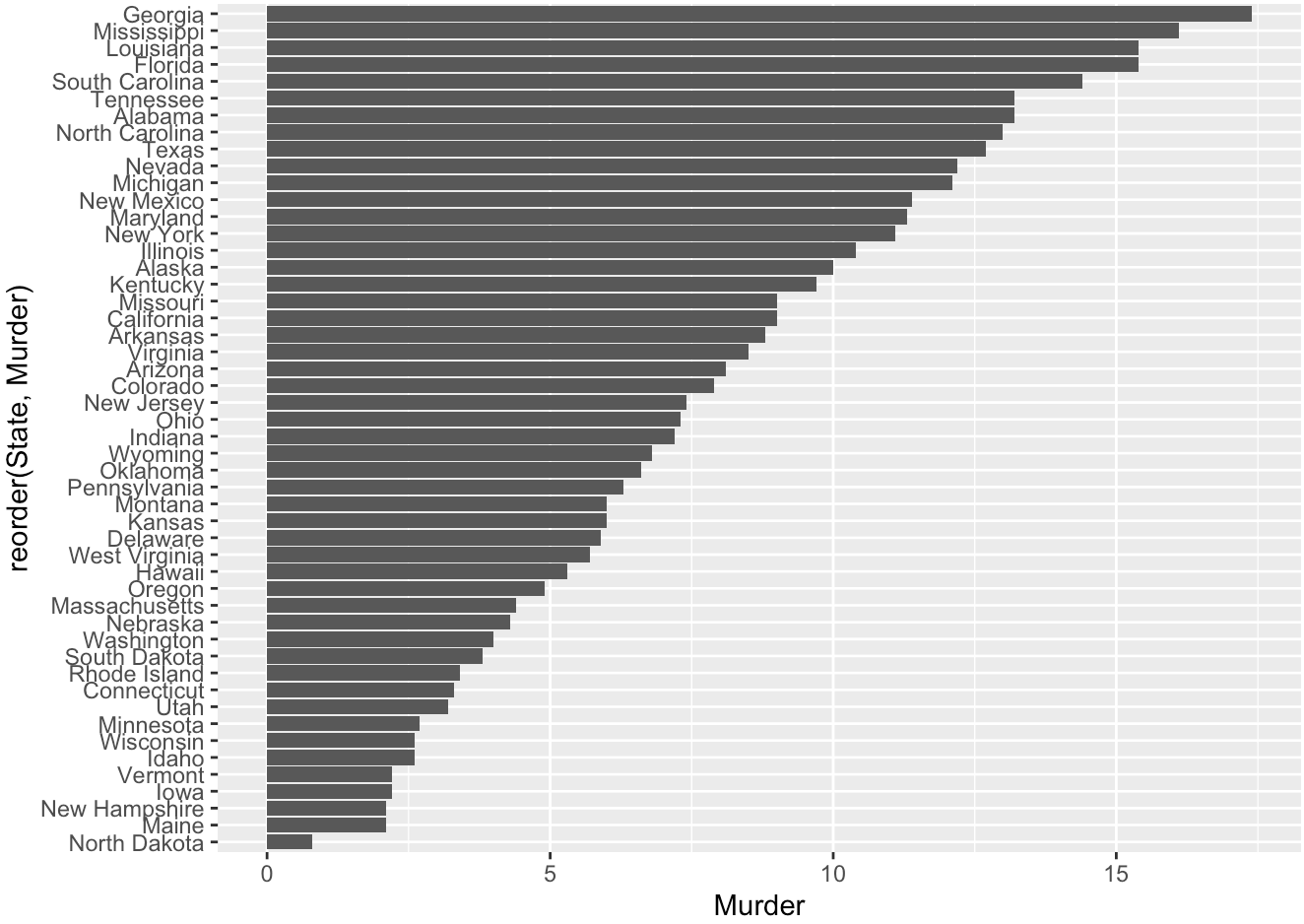
In some circumstances it might be useful to rotate (or "flip") your plots to enable a more clear visualization. For this we can use the `coord_flip()` function. Lets look at an example considering arrest data in US states. This is another inbult dataset called `USArrests`.

```
head(USArrests)
```

```
           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
```

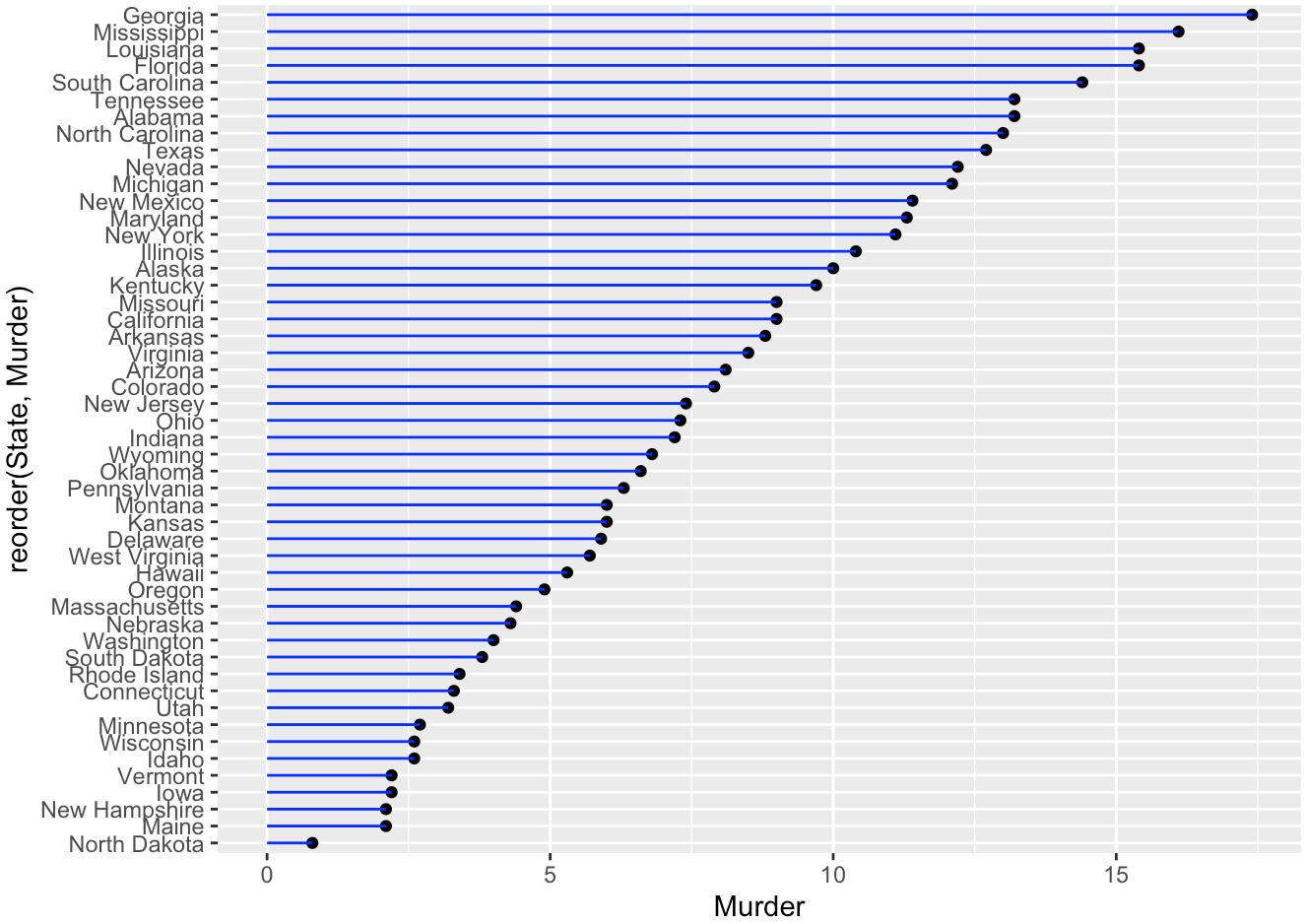
```
USArrests$State <- rownames(USArrests)
ggplot(USArrests) +
  aes(x=reorder(State,Murder), y=Murder) +
```

```
  geom_col() +
  coord_flip()
```



Hmm... this is too crowded for an effective display in small format. Let's try an alternative custom visualization by combining `geom_point()` and `geom_segment()`:

```
ggplot(USArrests) +
  aes(x=reorder(State,Murder), y=Murder) +
  geom_point() +
  geom_segment(aes(x=State,
                   xend=State,
                   y=0,
                   yend=Murder), color="blue") +
  coord_flip()
```



We will be exploring and producing different plot types in this way throughout the rest of our course going forward. **Happy ggplotting!**

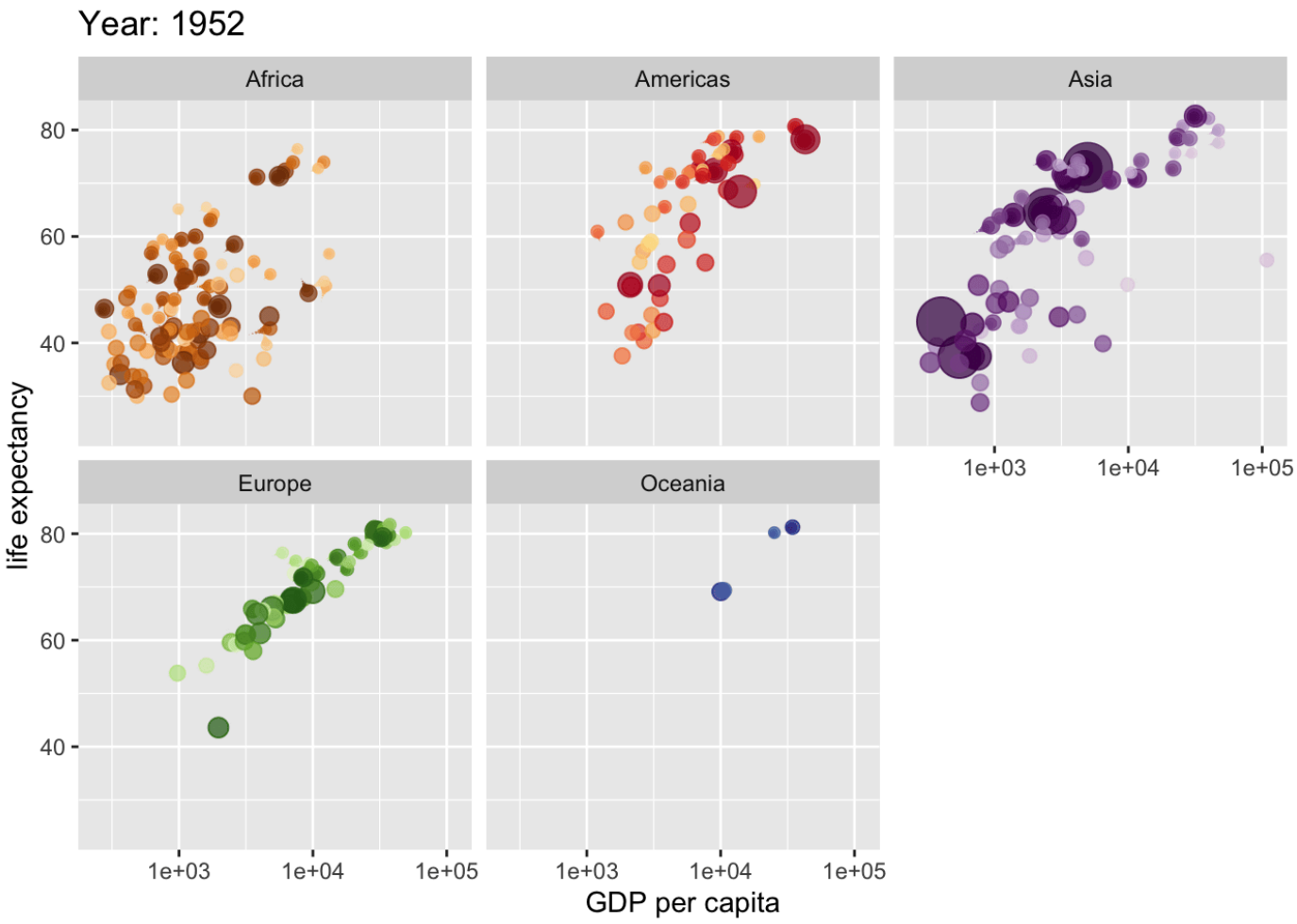
9. Extensions: Animation

There are lots of excellent extension packages for ggplot that you can find out about here:

Lets play with one for making animated plots called **gganimate**. You will first need to install this package, and it's buddy **gifski** for making animated gifs, (i.e. In your console typle: `install.packages("gifski")` and `install.packages("gganimate")`)

```
library(gapminder)
library(gganimate)

# Setup nice regular ggplot of the gapminder data
ggplot(gapminder, aes(gdpPercap, lifeExp, size = pop, colour = country)) +
  geom_point(alpha = 0.7, show.legend = FALSE) +
  scale_colour_manual(values = country_colors) +
  scale_size(range = c(2, 12)) +
  scale_x_log10() +
```

```
  # Facet by continent
  facet_wrap(~continent) +
  # Here comes the gganimate specific bits
  labs(title = 'Year: {frame_time}', x = 'GDP per capita', y = 'life expectancy') +
  transition_time(year) +
  shadow_wake(wake_length = 0.1, alpha = FALSE)
```



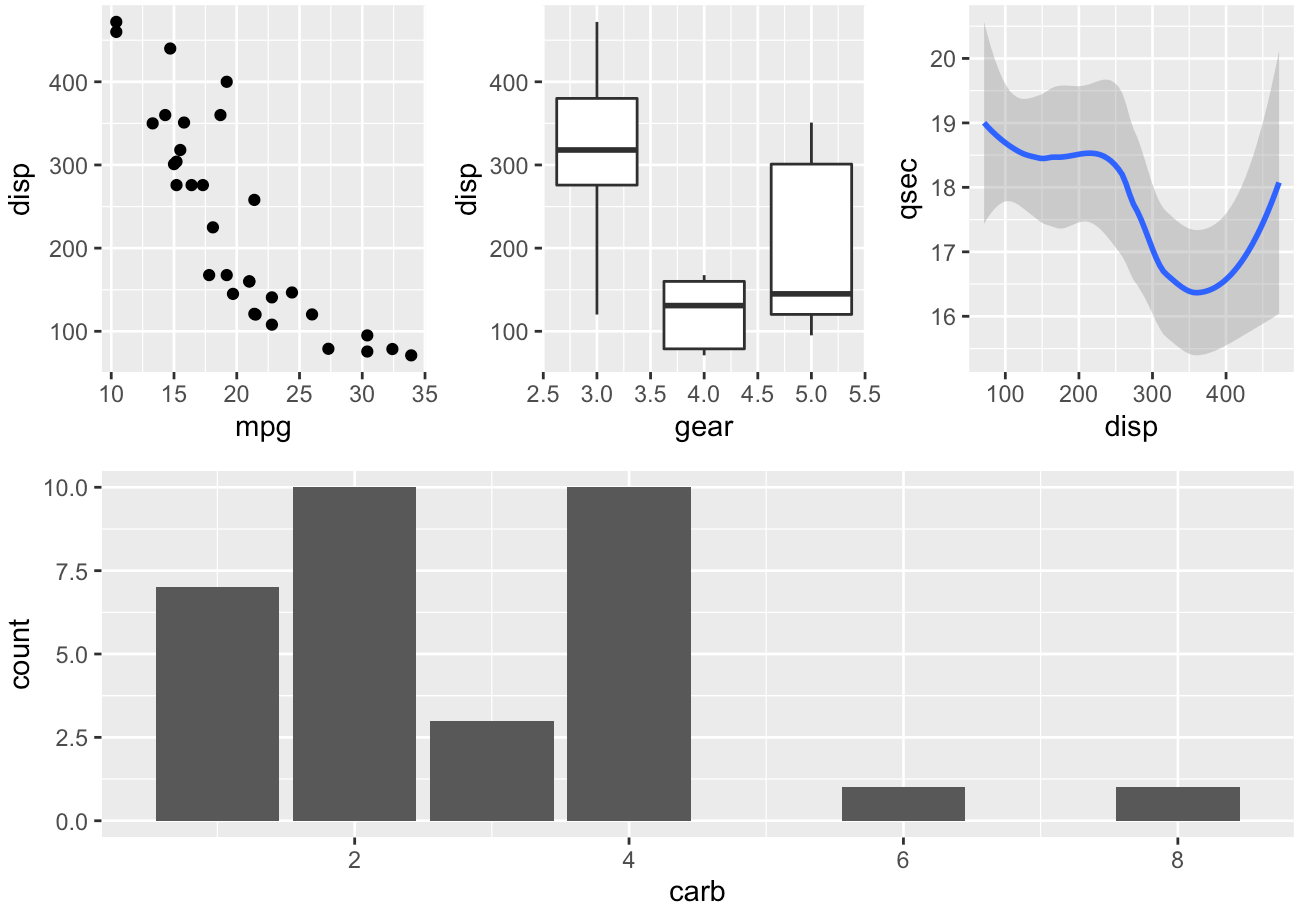
You can lean more about each of these gganimate functions on the package vignette.

10. Combining plots

Another excellent package extending ggplot is **patchwork** that is useful for combining plots to make an all-in-one multi-panel figure. For example:

```
library(patchwork)

# Setup some example plots
p1 <- ggplot(mtcars) + geom_point(aes(mpg, disp))
p2 <- ggplot(mtcars) + geom_boxplot(aes(gear, disp, group = gear))
p3 <- ggplot(mtcars) + geom_smooth(aes(disp, qsec))
```

```
p4 <- ggplot(mtcars) + geom_bar(aes(carb))

# Use patchwork to combine them here:
(p1 | p2 | p3) /
      p4
```



Again you can learn more from the package vignette.

About this document

Here we use the `sessionInfo()` function to report on our R systems setup at the time of document execution.

```
sessionInfo()
```

```
R version 4.1.2 (2021-11-01)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Big Sur 10.16

Matrix products: default
```

```
BLAS:
/Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.0.dylib
LAPACK:
/Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] patchwork_1.1.2 gganimate_1.0.8 gapminder_0.3.0 dplyr_1.0.10
[5] ggplot2_3.3.6   labsheet_0.1.2

loaded via a namespace (and not attached):
 [1] progress_1.2.2    tidyselect_1.1.2  xfun_0.33         purrr_0.3.4
 [5] splines_4.1.2     lattice_0.20-45   colorspace_2.0-3  vctrs_0.4.2
 [9] generics_0.1.3    htmltools_0.5.3   yaml_2.3.5        mgcv_1.8-40
[13] utf8_1.2.2        rlang_1.0.6       pillar_1.8.1      glue_1.6.2
[17] withr_2.5.0       DBI_1.1.3         tweenr_2.0.2      lifecycle_1.0.3
[21] stringr_1.4.1     munsell_0.5.0     gtable_0.3.1      htmlwidgets_1.5.4
[25] evaluate_0.17     labeling_0.4.2    knitr_1.40        fastmap_1.1.0
[29] gifski_1.6.6-1    fansi_1.0.3       scales_1.2.1      jsonlite_1.8.2
[33] farver_2.1.1      hms_1.1.2         digest_0.6.29     stringi_1.7.8
[37] grid_4.1.2        cli_3.4.1         tools_4.1.2       magrittr_2.0.3
[41] tibble_3.1.8      crayon_1.5.2      pkgconfig_2.0.3   ellipsis_0.3.2
[45] Matrix_1.5-1      prettyunits_1.1.1 assertthat_0.2.1  rmarkdown_2.16
[49] rstudioapi_0.14   R6_2.5.1          nlme_3.1-159      compiler_4.1.2
```