Data Visualization in R with ggplot2

Kara Woo

This document contains the full lesson text for each video in the course.

1 Introduction

1.1 Introduction

Welcome to this course on data visualization in R with ggplot2. I'm Kara Woo, and over the coming videos I'll show you how to create effective and attractive data visualizations using the R package ggplot2.

This course is for anyone who wants to learn the principles of good data visualization and how to apply them in R using ggplot2. To get the most out of this course, you should have some basic familiarity with R. You should know how to install and load packages, read in data, and perform simple data manipulations. You don't need any knowledge of ggplot2 or any other plotting systems in R.

To teach ggplot2, I'll show code and plots onscreen using RStudio. All of the code that I show in the videos is available in the working files. For most of the coding lessons I've also included some exercises and answer keys to help you get more practice using ggplot2.

By the end of the course, you'll understand what makes an effective visualization and you'll be able to make informed choices about data presentation. You'll also be able to create a variety of types of visualizations using ggplot2 and customize them to meet your needs.

Lastly, you'll have a framework for thinking systematically about visualization that will help you move on to tackle more complex visualizations, interactive visualizations, and other visualization tools.

Last updated on July 13, 2016

1.2 About the author

Hi! I'm Kara Woo, and I'm excited to introduce you to data visualization in ggplot2. I'm currently a graduate student in information science at the University of Washington where I mainly focus on data management and visualization. I have been using R and ggplot2 since 2012, and I have experience teaching R through Software Carpentry and Data Carpentry, which are two great organizations that teach programming and data skills to researchers.

You can fine me online at karawoo.com and on Twitter at @kara_woo.

In my free time I co-curate a blog called Accidental aRt, where we collect visualizations gone beautifully wrong – that is, those that didn't turn out as intended, but produced something interesting or pretty nonetheless. You can find us at accidental-art.tumblr.com and on Twitter at @accidental___aRt with two underscores. Check us out and see all of the beautiful mistakes people have made with visualization, or submit your own accidental art.

2 Data visualization principles

2.1 What is a data visualization and why do we visualize data?

Charts and graphs are everywhere, but what is their purpose? In this video, we'll talk about what constitutes a data visualization, as well as the different reasons one might create a visualization.

There are many definitions of data visualization, but for our purposes, a data visualization consists of:

1. Mapping variables to visual encodings (like position along an axis)
2. Geometric elements to represent data (like points)
3. Reference elements (like axes)

A data visualization presents data visually in a **systematic way**.

So why do we want to create visualizations? I'm going to focus on two main reasons, and the implications these have on the types of visualizations you'll create.

Broadly speaking, people create visualizations as part of exploratory data analysis or to convey a story about their data to an audience.

When you're working with data, especially a new dataset, exploratory data analysis is an early step that helps guide the course of your work. Exploratory data analysis can allow you to see patterns and develop new questions about your topic. Using

visualization as part of your exploratory data analysis helps you see trends in your data, allows you to spot unexpected patterns or anomalies, and can inspire new questions or ideas much more easily than if you viewed the raw numbers or statistical summaries alone.

Visualization lets you quickly see the distribution of your data, as in this histogram.

Visualization also helps you spot outliers more quickly. It's much easier to see the point that stands out in this visualization than by viewing the raw numbers in the table.

When using visualization as part of exploratory data analysis, you'll want to focus on making many visualizations to answer different questions and to view your data from many different angles. The more visualizations you make in this stage, the more likely you are to spot something interesting that warrants further study.

That said, it is still important to let questions drive your exploratory data analysis and not fish haphazardly for patterns that might be statistically significant, but functionally meaningless.

The other main reason to visualize data is to convey information to others, or to tell a story about the data. You may be presenting your findings to your colleagues, your boss, or the readers of your blog, website, or research paper.

Unlike in exploratory data analysis, the goal here is not necessarily to show many views of the data, but rather to present a clear story. You'll want to put a lot more thought into what you want the viewers to learn from the visualization, and how to present that information clearly and effectively, with minimal distractions. Fortunately, in the rest of this course we'll be learning how to do just that.

2.2 Types of variables and encodings

Data visualization involves mapping variables in your data to different visual encodings, such as color or position along an axis. How you choose an encoding should depend on the type of data you're presenting. In this video, we'll discuss the three main types of variables and some common visual encodings.

The three main types of variables are nominal, ordinal, and quantitative.

You might know nominal variables as categorical – they are discrete categories with no hierarchical ranking between them, such as types of flowers, country names, or binary yes/no responses.

Ordinal variables are ordered categories, like low, medium, and high, or grades on a letter grade scale. You can rank ordinal categories; for instance, an A grade is higher

than a B. But you cannot calculate differences or do other kinds of computation. That is, you can't subtract a grade of B from a grade of C and get an A.

Quantitative variables are generally measured on a numeric scale. Examples include chemical concentrations, temperature, or revenue generated in a particular period. With these types of variables, you can typically calculate differences, or sums, or other perform various other types of computations.

Visual encodings refer to how data is represented visually. Here are some examples:

- Position is the first example, and it is one of the most effective ways of conveying information. Here we can see two points, one of which is higher. We can assume that the value of point B is greater than the value of point A because it is closer to twenty on the y scale. Position is also being used on the x axis to differentiate points A and B.
- Length. Length is often used to encode data, for example in bar charts.
- Area can be used to scale the size of chart elements with data.
- Angle is being used in this pie chart to represent the different proportions that make up the whole
- Color is another very common visual encoding, and it has several different aspects that can be used to encode data: saturation, shown on top; lightness in the middle; and hue on the bottom.

These are only a few visual encodings that are particularly common, but there can be many more, including shape, texture, orientation, volume, and more.

To see an example of these encodings in action, let's look at this chart. This is an early data visualization by William Playfair and is one of the first examples of a line chart. The chart shows imports and exports between England and Denmark and Norway from 1700 to 1780. Here the year is encoded as position along the x axis and the monetary value of the imports and exports is encoded as position along the y axis. Color is used to encode both the type of line (that is, whether it represents imports or exports) and whether the net balance of imports against exports favors England or not. Finally, the extent of the balance against or in favor of England is mapped to the vertical length between the two lines.

Thinking of data visualizations in terms of visual encodings can help you make solid choices about how to present your own data. I'd encourage you the next time you see a data visualization in the wild, see if you can break it down into its various encodings and think about why the creators might have chosen certain encodings over others.

2.3 Choosing effective visual encodings

In this video, we're going to talk about how to choose effective visual encodings for your data.

Researchers in the field of graphical perception have examined how well humans can read and understand various kinds of graphics. One of the most foundational works in this field is Cleveland and McGill's "Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods" paper, which is really the place to start if you want to read more research on how humans perceive plots.

When choosing encodings, there are two main considerations you'll need to make. First, the encoding you choose needs to match your data type, and we'll talk more about what that means in a moment. Second, within the space of encodings that match the data type, usually you'll want to choose those that help people read the data most easily.

When I say that the encoding you choose should match your data type, for nominal data that means that the encoding should be something that easily represents discrete, unordered categories. Position can be used to separate bars in a bar chart. Hue and shape also make sense for nominal data. Area, length, and volume do not match nominal data, because they scale in magnitude in a way that would imply an order to the categories according to size.

Ordinal data requires encodings that have discrete, ordered categories. Position can be used if the categories are placed in order along an axis or if separate charts for each category are arranged sequentially. Lightness and saturation also make sense, but hue does not because it is not naturally ordered.

For quantitative data, you usually want an encoding that can represent continuous ordered values. Position works here too if you use a continuous scale. Length is another encoding that makes sense for quantitative data.

You probably noticed that position came up as a reasonable encoding for all three kinds of data. Position is one of the most effective visual encodings there is – most people are very good at perceiving differences in position on a plane. It is also one of the only encodings that can easily convey data that is ordered or unordered and discrete or continuous.

When it comes to choosing encodings that help people easily read the data, your choices will depend a little bit more on your goals for the visualization.

There is some research on what encodings people are able to interpret most precisely. Again, position is best across the board. For nominal data, hue and texture are among the more effective encodings. For ordinal data, lightness and saturation are

good choices. And for quantitative data, length, angle, and slope are effective.

Generally speaking, people are better at estimating lengths than areas, and are worst at estimating volumes. If you want your viewers to be able to precisely interpret different values from your visualization, you are better off using length than area or volume.

However, extremely precise interpretation is not always the most important goal of every visualization. The suggestions I just made about the most perceptually effective encodings are not hard and fast rules, and in fact new research often refutes what we previously believed to be true about the effectiveness of various encodings. When choosing encodings for your data, the most important thing is to think carefully about what you want to convey and choose the encodings that will support this goal without misleading or deceiving the viewer.

2.4 Color

In this video we're going to focus on the use of color in visualization. Color can be a really powerful component of visualization, but if used improperly it can obscure information or mislead viewers, so I'm going to cover some important considerations when using color in visualization.

There are three types of color scales that get used in visualizations: sequential, diverging, and qualitative.

Sequential scales usually use a single hue of varying lightness or saturation. Lightness is generally perceived as ordered, so this makes sense for ordinal or quantitative data. It's generally better to map higher values to the darker colors in the scale.

Diverging color scales are useful when your data has a meaningful midpoint, like if the data includes positive and negative values. The positive and negative sides should have different, fairly saturated hues, while the midpoint should be a more neutral gray or white.

Finally, qualitative color scales with discrete hues should be used for nominal data. Hues aren't naturally ordered, so they make sense for categorical values.

Here are some additional tips for using color in visualization.

The first is to try to limit the number of color steps, particularly in qualitative color palettes. The more colors you have, the more likely those colors will be similar and the harder it is to quickly differentiate between colors. Generally it's best to stick to fewer than 9 colors if you can.

You should avoid continuous rainbow color scales for quantitative data. These can be misleading in a number of ways. For one, hues aren't naturally ordered, and

people tend to segment colors into classes, which isn't appropriate for continuous data. Hues in a rainbow color scale also have different lightnesses, which emphasize certain values over others. And finally, rainbow color scales artificially emphasize boundaries between certain colors.

Take this rainbow scale as an example. The two sets of lines are equally far apart, yet there appears to be a much greater difference in hues between the lines on the right than between the lines on the left. If you were viewing a heat map with this color scale, it would appear that values that match those between the right set of lines are more different and transition much more steeply than values that fall between the left set of lines, when fact this is not the case. This has actually been the source of errors and misinterpretations in published studies that used a rainbow color map.

Color blindness is quite common and affects up to 8% of men and 0.5% of women. You should be aware of this when choosing color palettes for your visualizations. Here I've put the color palettes I showed earlier through a color blindness simulator, so if you do not have deuteranopia you can see how the palettes appear to someone who does have this common form of color blindness.

Colorbrewer2.org is a great resource for color palettes, and you can use it to search for palettes that are colorblind safe.

3 Introduction to visualization in ggplot2

3.1 Getting set up with ggplot2

In this video, I'll talk a little bit about what makes ggplot2 an excellent choice for visualizing data and take you through the setup needed to get started with it.

One of the big advantages of creating data visualizations in R or another scripting language is reproducibility. There is no need to remember which buttons you clicked on to create a visualization, because the code is right there. This also means that you can tinker with the code to create variations on a plot without having to do everything over from scratch.

There are several plotting systems available for R: base graphics (which comes built in to R), lattice, and ggplot2 are the main ones for static graphics. Many people happily use base graphics and lattice, and it is possible to create excellent visualizations in any of these systems. I personally find ggplot2 the most intuitive and easiest to learn of all of these options. The grammar behind ggplot2 makes for a consistent framework for developing your visualizations. There are a number of ggplot2 extensions that can help you make more specialized plots. And finally, there

is a large community of ggplot2 users, which makes finding help easy.

This course will cover the basics of ggplot2, but if you want to do additional reading I highly recommend the following resources:

- The ggplot2 book by Hadley Wickham
- the R Graphics Cookbook by Winston Chang
- The online ggplot2 documentation

Throughout the course, I will be showing code samples and plots using RStudio. RStudio is a great development environment for R, and I highly recommend it if you're just getting started. However, there's no requirement that you use RStudio if you prefer a different text editor.

This screen shows the layout that you'll be seeing for all of our code work. The top left panel is my script window where I edit the R file that contains my code. The bottom left is the console, where the code gets executed. In the top right we see the working environment, which is currently empty. Once we create some objects those will show up here. In the bottom right is the space where plots will appear once we start making them.

This course requires three packages: ggplot2, dplyr, and gapminder. ggplot2 contains the functions we'll use to create plots. dplyr has some great functions for easily working with data, and we'll use it to subset or aggregate our data. gapminder is a data package that contains data on life expectancy, population, and GDP for many countries of the world. We'll use this data in a lot of our example plots. The data originally comes from http://www.gapminder.org/data/.

I'm recording this course using R version 3.3.0 and ggplot2 version 2.1.0, as well as dplyr version 0.4.3 and gapminder version 0.2.0. If you're using different versions of this software, it's possible that there could be subtle differences in how the code works compared to what I show, but there probably won't be any major problems.

To install these three packages at once, run `install.packages` with a vector of the package names.

```
install.packages(c("ggplot2", "dplyr", "gapminder"))
```

3.2 ggplot2's grammar of graphics

This video will explain the grammar of graphics philosophy that underlies ggplot2. Understanding this grammar will help you create effective visualizations, because you'll understand the tools that you can leverage to create powerful custom graphics.

The grammar of graphics was a concept articulated by Leland Wilkinson to describe the fundamental features that make up statistical graphics.

Hadley Wickham built on the grammar of graphics to develop the layered grammar of graphics of ggplot2. This grammar describes how to build plots from the data up.

Rather than focusing on chart type (bar charts, scatter plots, line charts), the layered grammar of graphics allows you to build your own graphics using visual elements that you specify, regardless of whether they fit narrowly into a particular chart type. This means you can create your own new visualization types from combinations of various elements.

I'm going to walk you through the elements of ggplot2's grammar of graphics. We'll come back to these concepts in later videos when we start building our graphics.

The most fundamental component of a data visualization is the data. ggplot2 expects data to be stored in a data frame, and for the variables you'll be visualizing to represent columns in your data, as in these few rows of data showing gas mileage, weight, and cylinder count for several types of cars

The next component of the grammar of graphics are the aesthetics. Aesthetics map data to visual variables, such as color or position on the x or y axes.

Geometric objects, or "geoms", represent the actual items you see on the plot. Geoms can be points, lines, shapes, etc.

These three elements: data, aesthetics, and geoms, are typically the bare minimum you'll need to specify to create a plot with ggplot2. However, there are other elements of the layered grammar of graphics that you can use to create more customized graphics.

Statistical transformations, or "stats" transform your data in various ways, usually to provide a summary of some kind. Examples of how these can be used are to create a histogram to show the distribution of your data, or fit a smoother to your data as in this example. Stats typically work behind the scenes unless you choose to override the default stats.

Facets allow you to break up your data into subsets to visualize as what's called small multiples, like in this chart which shows the data on gas mileage split by the number of cylinders in the car's engine.

Scales control how data values are mapped to the aesthetics you've chosen for your plot. This goes on behind the scenes if you use the default values, but you can also override the defaults, for instance if you want to use a custom color scale or log-transform an axis..

Finally, the coordinate system or "coord" controls how data coordinates are mapped to the visual plane of the graphic. We won't go into much detail with coordinates in this video series, but it is possible to use this element of the grammar of graphics to specify coordinates besides the standard Cartesian coordinate system, such as polar coordinates or map projections.

To summarize, the elements of ggplot2's layered grammar of graphics are data, aesthetics, geometric objects, statistical transformations, facets, scales, and coordinate systems. Typically, you will at least specify the data, aesthetics, and geoms at a minimum.

This concludes our introduction to ggplot2's grammar of graphics.

3.3 Tidy data and data manipulation

This video is going to cover tidy data and a few commands for manipulating data that we'll use throughout the course.

Hadley Wickham described the idea of tidy data in a 2014 paper called "Tidy Data". The concept basically boils down to the idea that data should have variables in columns and observations in rows. A third point is that each observational unit should be its own table, meaning that if you have different types of data that are collected at different levels of specificity, these would go in separate tables to reduce duplication of data.

Here's an example of untidy data. This is data on population of a few countries for a couple years. In an untidy form, you might have rows for each country and columns for each year of population data. This violates the first two principles of tidy data, because year is a variable that is not in a column and rows represent multiple observations of population in different years.

In a tidy format, country, year, and population are each variables with their own columns. This structure makes it easier for programs to access all of the variables, and it is the format that ggplot2 will expect.

How to go from messy to tidy data is outside the scope of this course, but I highly recommend the dplyr and tidyr packages for working with data.

I am, though, going to show a few functions from the dplyr package that we'll be using for simple data manipulations in this course.

Here in the script window I am going to load dplyr and the data package gapminder. When I highlight these lines and hit Command-Enter on Mac, or Control-Enter on Windows, the code is sent to the console and evaluated.

```
library("dplyr")
library("gapminder")
```

By loading these packages we now have access to the gapminder dataset, which has information on life expectancy, population, and gross domestic product for many countries over time. Viewing `head(gapminder)` shows us the first few rows of the data, and we can see that we have columns for country, continent, year, life expectancy, population, and gross domestic product.

Sometimes we'll want to subset the data to only a particular year. The dplyr function `filter` lets us subset rows of a dataset. To subset data for the year 2007, we call `filter` on the gapminder data, and then keep only those rows where the year column is equal to 2007.

```
filter(gapminder, year == 2007)
```

When I evaluate this, we can see that it spits out the first few lines of the data in the console. I can also assign this filtered dataset to a new variable, let's call it `gap_07`.

```
gap_07 <- filter(gapminder, year == 2007)
```

If I wanted to filter data based on several possible values, I can use the `%in%` operator. This will let me do things like keep all the data where the country column has a value of "France" or "Turkey".

```
filter(gapminder, country %in% c("France", "Turkey"))
```

It's also possible to filter based on multiple conditions. To get the data from 2007, excluding data from the continent Oceania, I use an ampersand to add multiple conditions.

```
filter(gapminder, year == 2007 & continent != "Oceania")
```

Aggregating data is another common task that we'll use a little bit in the course. With dplyr, we can aggregate data using the `group_by` and `summarize` functions. A handy thing about dplyr is that it provides access to a pipe operator that lets you chain together statements such that the output of one function gets sent to the next. Let's start with our 2007 data.

```
filter(gapminder, year == 2007)
```

Instead of putting gapminder inside the call to `filter`, I can pipe the gapminder data into the filter function.

```
gapminder %>%
  filter(year == 2007)
```

This is the same as calling `filter` with `gapminder` as the first argument because whatever is on the left side of the `%>%` symbol gets used as the first argument of the function following the symbol.

Let's say I want to find the median population by continent for 2007. I can then add another pipe to send the 2007 data to the `group_by` function, where I'll group it by continent.

```
gapminder %>%
  filter(year == 2007) %>%
  group_by(continent)
```

Then, I pipe that to `summarize`. I'll use the `median` function to calculate the median population for each of the groups, and put that in a new column called `medpop`.

```
gapminder %>%
  filter(year == 2007) %>%
  group_by(continent) %>%
  summarize(medpop = median(pop))
```

The end result is a table of median population by continent, which we've created through this nice, linear chain of function calls. There are many other useful dplyr functions, but these are the ones that will come up in the rest of this course.

3.4 Basic plots

In this video, we're going to make our first plots using ggplot2. The code that I'm going to show is included in your Working Files.

The first thing to do is load the packages we're going to use: ggplot2, gapminder (which contains the data we're going to use), and dplyr (for manipulating data). In RStudio I can run code by highlighting it and hitting Command-Enter on Mac or Control-Enter on Windows.

```
library("ggplot2")
library("gapminder")
library("dplyr")
```

To simplify things, I'm creating a small dataset consisting of just the data from the year 2007. Using dplyr's `filter` function, I filter the data to keep only the data where the "year" column has a value of 2007.

```
gap_07 <- filter(gapminder, year == 2007)
```

Let's view the first few lines of `gap_07`. We can see there are six columns: the country name, continent, year, average life expectancy of the country in that year, its population, and its GDP.

```
head(gap_07)
```

To create a plot with ggplot2, we call the `ggplot` function on the data we want to plot. The second argument will be a call to the `aes` function, which specifies which variables, or columns in our data, we want to map to which visual encodings. In this example we're going to make a scatterplot of life expectancy vs. per capita GDP.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp))
```

This line of code creates a plot object, but if we run it we see that no points appear, it's just an empty plot. That is because we have not specified which geometric elements or "geoms" to use to represent the data on the plot. Let's add points now

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

Note the plus symbol at the end of the first line. We'll use this plus symbol to add additional layers or elements to our plots. When we run this whole section of code, we now get a scatterplot as a result.

Let's try another example using a different geom, `geom_bar`. `geom_bar` is an interesting case, because by default it will create bar charts that show the counts of observations for various variables in the data. So if we wanted to show how many observation there were in each continent in 2007, which in this data corresponds to the number of countries because we have one row per country, we would do so like this:

```
ggplot(gap_07, aes(x = continent)) +
  geom_bar()
```

Note we do NOT specify a y aesthetic here. `geom_bar` takes care of that for us.

So that's good, but what if we wanted to create other kinds of bar charts, for example a bar chart showing the median population by continent in 2007? We can do that with `geom_bar` too.

First we're going to get the data we want by calculating the median population of each continent using dplyr's `group_by` and `summarize` functions. Taking the 2007 data, we group by continent and summarize to find the median for each continent. The `%>%` symbol indicates that we're creating a chain where the `gap_07` data gets passed to `group_by`, which groups the data by continent, and then to `summarize` which calculates the median population for each continent.

```
gap_07_med <- gap_07 %>%
  group_by(continent) %>%
  summarize(pop = median(pop))
```

With this data we'll now create our chart, and to do so we need to make our first use of `stat`. In the previous bar chart, `stat` was working behind the scenes to generate the counts of data that appeared in the chart. This counting behavior is the default for `geom_bar`, but we want to override it to show the median population values we calculated, so we're going to specify `stat = "identity"` in our call to `geom_bar`, and now we do specify a y aesthetic so ggplot knows what to plot instead of counts.

```
ggplot(gap_07_med, aes(x = continent, y = pop)) +
  geom_bar(stat = "identity")
```

We're going to make one more chart in this video. Let's look at how the GDP of Iceland has changed over time. First we're going to subset the gapminder data to just include Iceland.

```
gap_iceland <- filter(gapminder, country == "Iceland")
```

Then we'll start building the plot. This time I'm going to save the plot as an object called `p`.

```
p <- ggplot(gap_iceland, aes(x = year, y = gdpPercap))
```

By saving the first line of code as an object, we now can easily try out different versions of the plot. If we want to see the GDP values as points, we do:

```
p + geom_point()
```

Or as a line:

```
p + geom_line()
```

Because ggplot2 focuses on the idea of building charts layer by layer, we can also add both points and lines:

```
p + geom_point() + geom_line()
```

If we want to save a plot as a file we'll use the `ggsave` function. By default this will save the last plot that was created, but it's generally a better idea to be explicit about which plot you want to save. We can do this one of two ways. First, we can add `ggsave` with a plus symbol to our chain of commands that creates a plot:

```
p + geom_point() + geom_line() +
  ggsave("iceland_gdp.png", height = 6, width = 8)
```

We specify the name and extension of the file, and we can specify the dimensions of the image as well. Several options are available for file types, including pdf, jpeg, svg, and others.

We can also create an object containing our whole plot and then call `ggsave` separately like so:

```
p <- ggplot(gap_iceland, aes(x = year, y = gdpPercap)) +
  geom_point() +
  geom_line()

ggsave("iceland_gdp.png", plot = p, height = 6, width = 8)
```

And that's it! Now we have made some basic plots and saved them.

3.5 Viewing data distributions

In this video we're going to look at a few ways to view data distributions and summary information. These kinds of visualizations are an important part of exploratory data analysis, since they can reveal patterns or anomalies in your data, and you can use them to help determine whether certain statistical assumptions are correct.

We're going to start with a histogram, which shows the distribution of numeric data. I've gone ahead and run the first few lines of code to load the packages and subset the data. To create a histogram of GDP using the `gap_07` data we've been working with, we call `ggplot` with an x aesthetic for the variable of interest and then add `geom_histogram`.

```
ggplot(gap_07, aes(x = gdpPercap)) +
  geom_histogram()
```

When we run this code, we get a message from ggplot:

```
"`stat_bin()` using `bins = 30`. Pick better value with `binwidth`."
```

By default, ggplot will separate the data into thirty bins and display the number of observations that fall into each bin. Sometimes, this might not be the right level of resolution for our data, so we can choose either the number of bins or the width of the bins.

Let's try setting the number of bins to 15. This goes in an argument to `geom_histogram`:

```
ggplot(gap_07, aes(x = gdpPercap)) +
  geom_histogram(bins = 15)
```

Or we can set the bin width. This creates bins based on the values in the data, so here we're choosing to create bins in increments of 5000.

```
ggplot(gap_07, aes(x = gdpPercap)) +
  geom_histogram(binwidth = 5000)
```

A different way of viewing the distribution is with `geom_density`, which will create a smoothed density estimate.

```
ggplot(gap_07, aes(x = gdpPercap)) +
  geom_density()
```

Next we're going to look at some summary information in the form of a box plot. A box plot will show us the range of our data, the quartiles, and the median.

Let's look at life expectancy by continent. Our x aesthetic will be continent, and our y aesthetic will be life expectancy. Then we use `geom_boxplot` as our geom.

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_boxplot()
```

If you haven't used box plots much before, the bottom and top of the boxes represent the first and third quartiles, which means that fifty percent of the data values fall within the box. The line through the box is the median, and the whiskers on either end of the box extend to the highest and lowest values that are within one and a half times the interquartile range of the data. Points beyond this are considered outliers and are shown as separate points

The box plot gives a helpful summary, but you may want to see the raw values in addition. To do this, we can add points on top of the box plot in a separate layer. One option is to use `geom_point`

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  geom_point()
```

However, having all the points in a line makes it hard to see, as the points can easily obscure each other. Instead let's use `geom_jitter`, which offsets the points from each other.

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  geom_jitter()
```

Now we have a box plot overlaid with the values, so we can see both the raw data and a summary simultaneously.

Note that the order of the layers matters with ggplot. If we wanted to put the box plot on top of the data, we'd switch the order of `geom_boxplot` and `geom_jitter`.

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_jitter() +
  geom_boxplot()
```

But that defeats the purpose of showing the data values, so let's switch it back.

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
```

```
  geom_jitter()
```

There are some arguments you can make to `geom_jitter` to customize how the points appear. To keep them within a more narrow band, set the width to a value less than one:

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  geom_jitter(width = 0.5)
```

You can also reduce the opacity of individual points by specifying an alpha value less than one, which is helpful if you have a large number of points.

```
ggplot(gap_07, aes(x = continent, y = lifeExp)) +
  geom_boxplot() +
  geom_jitter(width = 0.5, alpha = 0.2)
```

Now we've gone through a few ways of viewing the distribution of your data and summary information about your data.

4 Adding complexity to visualizations

4.1 Additional visual encodings: color, shape, and size

We've made some very basic visualizations; now we're going to add additional elements to convey more information and make more interesting visualizations.

So far, we haven't used color at all. Let's do that now and take our first scatterplot and color the points by continent. In ggplot2, this is as easy as adding an additional aesthetic, the `color` aesthetic

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point()
```

We can also map the continent to point shape.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, shape = continent)) +
  geom_point()
```

Or shape and color simultaneously, though this is a little redundant.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, shape = continent,
                   color = continent)) +
  geom_point()
```

Let's take it back to just coloring the points by continent, and then scale the size of the points by population.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent,
                   size = pop)) +
  geom_point()
```

Next let's try showing trends in population over time with a line chart where each line represents a continent. First I'm going to group the data by continent and year and use summarize to find the mean population for each continent in each year.

```
gap_pop <- gapminder %>%
  group_by(continent, year) %>%
  summarize(pop = mean(pop))
```

Next I'll call `ggplot` on this data with year as the x aesthetic and population as the y, and add `geom_line`.

```
ggplot(gap_pop, aes(x = year, y = pop)) +
  geom_line()
```

Something is clearly wrong here. `ggplot` doesn't know that each continent is supposed to be its own line, so it tries to connect all the data points. This is a common mistake, and we can use the `group` aesthetic to group the data by continent.

```
ggplot(gap_pop, aes(x = year, y = pop, group = continent)) +
  geom_line()
```

This isn't very useful though if we can't tell which continent is which. Using the `color` aesthetic will let us color the lines by continent, and it actually eliminates the need to use `group`.

```
ggplot(gap_pop, aes(x = year, y = pop, color = continent)) +
  geom_line()
```

These are some ways to incorporate color and other visual encodings into your plots with ggplot2. If you want some more practice, there are practice exercises in your Working Files.

4.2 Small multiples

This short video is going to cover how to use the small multiples visualization technique in ggplot2. In small multiples you create a series of similar charts that appear together in a grid, allowing you to easily compare the charts. This helps avoid creating single charts that are excessively cluttered.

In ggplot2 we use facetting to achieve this effect. There are two facetting functions, `facet_grid` and `facet_wrap`, and I'll go over both, starting with `facet_grid`.

We're going to create a set of scatter plots of GDP and life expectancy by year and continent. First I'll call ggplot on the entire gapminder dataset, which we have through the gapminder package. I specify GDP and life expectancy as the x and y aesthetics. Next, I add `geom_point`. So far, this code is the same as we'd use to create a single scatter plot. Finally, I add `facet_grid` to perform the facetting. The two variables that I'll use to separate the data into rows and columns, respectively are the arguments to `facet_grid`.

```
ggplot(gapminder, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  facet_grid(continent ~ year)
```

I know that's a little hard to see in the corner of my screen, so here it is close up.

If you want to have your plots all in one row or column, you can do this with `facet_grid` as well. Let's view a line chart of life expectancy over time separated by continent. First I group by continent and year and summarize to find the average life expectancy, then I plot them and use `facet_grid` to separate the plots by continent.

```
gap_life <- gapminder %>%
  group_by(continent, year) %>%
  summarize(lifeExp = mean(lifeExp))
```

Next I add `geom_line`.

```
ggplot(gap_life(aes(x = year, y = lifeExp))) +
  geom_line()
```

Finally I add `facet_grid`.

```
ggplot(gap_life, aes(x = year, y = lifeExp)) +
  geom_line() +
  facet_grid(continent ~ .)
```

When I'm not specifying a variable by which to split the charts into columns, I use a period as a placeholder.

Viewing the charts this way makes the individual plots very stretched out, so let's flip things and put the charts into a single row.

```
ggplot(gap_life, aes(x = year, y = lifeExp)) +
  geom_line() +
  facet_grid(. ~ continent)
```

Finally, `facet_wrap` will wrap a one dimensional sequence of panels into two dimensions, which can help use screen space more efficiently when you are facetting by a

single variable.

```
ggplot(gap_life, aes(x = year, y = lifeExp)) +
  geom_line() +
  facet_wrap(~ continent)
```

Note that there is no period on either side of the tilde with `facet_wrap`.

Now our five plots have been arranged to make good use of the screen space while keeping the aspect ratios of the individual plots a bit more reasonable.

And that's how you create small multiples with ggplot2!

4.3 Smoothing functions

In this video, I'm going to demonstrate how to add smoothers to scatter plots to help show patterns and trends in the data quickly.

Let's demonstrate this by adding a smoother to our scatterplot of 2007 life expectancy and GDP. Here I've included the code for our scatterplot, with GDP and life expectancy as our x and y aesthetics and `geom_point` as our sole geom. To add the smoother, we add `geom_smooth`.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  geom_smooth()
```

This takes our data and uses local regression to create the smoothed curve. By default, it also shows the 95% confidence interval.

You can learn more about this method for fitting the curve by typing `?loess` into the console.

We're not going to get into too much detail on the underlying method, but it is the default smoothing method for plots with fewer than 1000 observations. For plots with more data points, ggplot2 defaults to using a generalized additive model, which comes with the mgcv package.

One thing that is fairly easy to do with the loess method is to control how wiggly the line is. We do this with the `span` argument, which can take values between 0 and 1.

Smaller values indicate more wiggliness…

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  geom_smooth(span = 0.2)
```

...and larger ones indicate less.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  geom_smooth(span = 0.9)
```

You can remove the confidence intervals by setting the `se` argument to FALSE.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

Or change the level of the confidence interval with the `level` argument.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  geom_smooth(level = 0.90)
```

Finally, another useful smoothing method is `lm`, which fits a linear model.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  geom_smooth(method = "lm")
```

As a final example, let's put together several of the techniques we've learned recently and facet this scatterplot by continent, add a smoother to each plot, and color that smoother by continent as well.

Let's start with our regular scatterplot.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

Then we'll facet by continent.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  facet_wrap(~ continent)
```

Then we'll add a smoother to each facet.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  facet_wrap(~ continent) +
  geom_smooth()
```

Oceania has too few countries for the smoothing to work, but we can see it worked for the other continents.

Now, to color the smoother by continent, we could add a color aesthetic to our call to `ggplot`.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  facet_wrap(~ continent) +
  geom_smooth()
```

However, this colors the points too. If we don't want that, we can actually specify aesthetics for a particular geom only, so in this case we would add a call to the `aes` function within `geom_smooth`.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  facet_wrap(~ continent) +
  geom_smooth(aes(color = continent))
```

This is a useful thing to remember. When you map variables to aesthetics in your initial call to the ggplot function, those mappings apply to the *entire* plot. When you map variables to aesthetics within a particular geom, the mappings apply only to that geom.

5 Customizing your plots

5.1 Axis scales

This video is going to cover how to customize axis scales, and in particular how to apply transformations to scales. The scales of axes are important as they control how viewers orient themselves to the plots.

Let's return to our scatter plot of life expectancy and GDP for 2007, which should look pretty familiar by now.

```
gap_07 <- filter(gapminder, year == 2007)
```

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```

A lot of the points in this plot are squished along the left side of the plot.

You might recall from our discussion of the grammar of graphics that scales are what control how data gets mapped to aesthetics.

Our plot is using a continuous scale by default. We're going to focus on the x axis, and you can see it does not change the plot at all when we explicitly add a

continuous x scale with the `scale_x_continuous` function.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_continuous()
```

To get a clearer view of the points on the plot, we can log-transform the x axis using the `trans` argument to `scale_x_continuous`.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_continuous(trans = "log10")
```

This transforms the x axis so that instead of incrementing values linearly, each value on the axis is ten times larger than the previous value. ggplot2 offers a number of different transformations that you can learn about by reading the help file for `scale_continuous`.

Some of the common transformations come with shortcuts, so for a log-transformed axis we can actually write `scale_x_log10()` and save a few keystrokes.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_log10()
```

When a few large values dominate our plots in a way that obscures the many smaller values, using log-transformed scales can help show all of the individual points more clearly.

However, in this example there is a tradeoff because the immediate visual impression that the plot shows is no longer accurate. By viewing the plot with a log transformed x axis scale, the trend of the relationship between GDP and life expectancy appears linear.

A viewer must read and interpret the scale of the x axis in order to understand that *it* is not linear, and then attempt to imagine how the plot would appear with a linear scale in order to understand the shape of the trend.

Showing the data with a log transformed axis is not a bad choice if your goal is to make the points easy to see, but you should be aware of the ways that this presentation hinders the kinds of inferences that could otherwise be made from the plot.

The one other useful axis scale customization this video will cover is setting axis limits. In our scatter plot, the y axis begins around 40. It's often a good idea to start axes at zero to give an accurate perspective on the values. I would say it is not such a huge deal in this case, but nonetheless let's set our y axis to go from 0 to

95 with the limits argument to `scale_y_continuous`, which takes a vector of the minimum and maximum values we want to use.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point() +
  scale_x_log10() +
  scale_y_continuous(limits = c(0, 95))
```

Now you know how to perform some common manipulations of axis scales.

5.2 Alternate color scales

In this video we're going to talk about how to customize the color scales of your plots in ggplot2. ggplot2 comes with a very nice default color scale, but sometimes you'll want to choose your own color scale.

Let's return to our scatter plot of 2007 life expectancy vs. GDP to illustrate the various color scales.

We'll use the same x and y aesthetics as we've been using, and color the points by continent by adding the color aesthetic to our call to ggplot.

Then we'll add `geom_point` and `scale_x_log10` to help us see the points a little better.

```
gap_07 <- filter(gapminder, year == 2007)

ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10()
```

Since the hue of the points is an aesthetic attribute that data values are mapped to, it's controlled by a scale, and we can customize that scale if we want to.

Colorbrewer2.org is a useful website for selecting color palettes. ggplot2 actually comes with a function that lets you access the Color Brewer palettes, so to customize our color palette we can add `scale_color_brewer` and use the `palette` argument to specify which palette to use.

I'm going to use a palette called "Dark2".

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  scale_color_brewer(palette = "Dark2")
```

There are also ways to manually set a color palette. For this we use the function `scale_color_manual` and pass a vector of the colors we want to use as the `values` argument.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  scale_color_manual(values = c("#FF0000", "#00A08A", "#F2AD00",
                                "#F98400", "#5BBCD6"))
```

These colors actually come from a package of palettes based on Wes Anderson movies. For this particular palette, another way we could use it is to install the package wesanderson. I'm going to do this up at the top of my script file just to keep all the packages together in the file.

```
install.packages("wesanderson")
```

Then load it.

```
library("wesanderson")
```

And then the package provides the values through the function `wes_palette`.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  scale_color_manual(values = wes_palette("Darjeeling"))
```

There are a number of packages that provide various color palettes for you to use, or you can always specify your own values with `scale_color_manual`.

Finally, if we wanted to customize the color of these points without using them to encode data values, we can do that too. Let's say that we don't want to color the points by continent, but we do want to make all the points red instead of the default black.

`geom_point` takes an argument `color` that allows us to set the color of the points. We can set it to "red".

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(color = "red") +
  scale_x_log10()
```

Or "darkblue".

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(color = "darkblue") +
  scale_x_log10()
```

Or a number of other colors. We can also set the size with the `size` argument

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp)) +
  geom_point(color = "darkblue", size = 3) +
  scale_x_log10()
```

Here we are *setting* values rather than mapping data to them. When we are mapping data to an aesthetic like color, we specify the column of data in a call to `aes` and customize the aesthetic by modifying the scale. If we want to *set* a value like color without mapping it to data, this is done through additional arguments to the geom.

5.3 Themes

The plots we've been making so far look pretty nice, but what if we want to polish them up for publication? We might want to do things like add titles, modify axis labels, change the appearance of the plot background, etc.

ggplot2's theme system lets you make changes to the non-data aspects of your plot.

There are four main components to the theme system:

- Elements are the things you can control, like plot title and legend position. There are about 40 elements total that can be customized.
- Element functions describe the element's properties.
- The theme() function lets you customize theme elements.
- Finally, there are complete themes that are sets of customized elements that have been pre-designed and are available for you to use.

On the left here is a plot with no customizations, and on the right is a plot where I have used a different pre-made theme and customized various additional theme elements.

The pre-made themes are what we're going to focus on for the rest of this video, since you can go a long way with them alone. In the next video we'll go over customizing individual elements.

ggplot2 comes with some built in themes, so I'll show how they look on our 2007 GDP vs. life expectancy plot with a log transformed x axis.

The default theme in ggplot2 is `theme_gray`, which we can also explicitly specify if we want to.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_gray()
```

"Gray" can also be spelled with an "e".

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_grey()
```

There are seven other themes built in to ggplot2, and you can see a list of all of with `?ggtheme`. Here are a few examples.

`theme_bw` uses a white background with light gray grid lines instead of the standard gray background.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_bw()
```

`theme_dark` shows the points on a dark background.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_dark()
```

And `theme_void` provides a plot that is completely blank aside from the data elements. This is useful if you want to start from scratch and add only the elements you want.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_void()
```

Beyond the built-in themes, there is also a package called ggthemes that you can install to get access to additional themes. I'm going to load this package at the top of my script.

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

And then I can use many other themes, including a minimal theme inspired by Edward Tufte…

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_tufte()
```

...a solarized theme with its own color palette...

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_solarized() +
  scale_colour_solarized("blue")
```

...and a theme imitating plots from the base R plotting functions that come built in with R.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point() +
  scale_x_log10() +
  theme_base()
```

You can view information about the ggthemes package, including all the available themes, at https://github.com/jrnold/ggthemes.

There are no practice exercises for this lesson, but you can spend some time exploring the available ggplot2 themes.

5.4 Customizing titles, axis labels, and legends

This video is going to cover how to customize particular elements of ggplot visualizations. Since there are over forty theme elements that you can customize we won't go through every single one, but you can read a description of all of the theme elements at http://docs.ggplot2.org/current/theme.html

Let's start once again with our 2007 GDP vs. life expectancy plot, with points colored by continent and a log transformed x axis. I'm also going to increase the point size slightly by setting the `size` argument of `geom_point` to 2.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10()
```

Next I'm going to use a different built-in theme for this plot, `theme_light`.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
```

```
  scale_x_log10() +
  theme_light()
```

To get this plot all nice and polished, I want to move the legend inside the area of the plot and remove the boxes around the legend key elements. The axis text and titles should also be a bit bigger, and I want to add a title to the overall plot and more complete axis labels.

To start customizing some of these components of the plot, I'm going to start a call to the theme function. Inside this is where I'll put all the theme elements I'm going to customize. To customize the legend position, we use the `legend.position` theme element. We can place the legend on a different side of the plot, like at the bottom

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = "bottom")
```

We can remove the legend entirely by setting `legend.position` to "none".

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = "none")
```

Or use a vector of coordinates to place it in a particular location.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = c(0.1, 0.85))
```

Here it's worth noting that this position is going to appear differently depending on how we view the plot. In the RStudio plot window the legend appears to overlap the edge of the plot, but if I save the plot to a file with different dimensions, it will be in a more convenient place.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = c(0.1, 0.85)) +
  ggsave("life_exp_gdp_2007_custom.png", width = 7, height = 7)
```

Sometimes it'll just take some tinkering to get the location of things like legends exactly how you want them.

Now, to remove the boxes around the legend elements, we'll use an element function. These functions are often used to modify properties of theme elements. We're going to use the `element_blank` function to remove the legend key background, thus removing the boxes around the legend items.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = c(0.1, 0.85),
        legend.key = element_blank()) +
  ggsave("life_exp_gdp_2007_custom.png", width = 7, height = 7)
```

Next I'm going to make the axis text and axis title sizes a little bigger using the `element_text` function. The theme element `axis.text` refers to the tick labels along the axes. If we wanted to customize only the x or only the y axis text, we would use `axis.text.x` or `axis.text.y`. `axis.title` refers to the labels of the axes.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = c(0.1, 0.85),
        legend.key = element_blank(),
        axis.text = element_text(size = 12),
        axis.title = element_text(size = 14)) +
  ggsave("life_exp_gdp_2007_custom.png", width = 7, height = 7)
```

To customize the content of the labels, I use the `labs` function. I'm going to write out more complete x and y axis labels.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = c(0.1, 0.85),
        legend.key = element_blank(),
        axis.text = element_text(size = 12),
        axis.title = element_text(size = 14)) +
  labs(x = "Per capita GDP",
       y = "Life Expectancy") +
```

```
  ggsave("life_exp_gdp_2007_custom.png", width = 7, height = 7)
```

I'm also going to add a title to the plot, and I'm going to capitalize the legend label. I use the word "color" here to customize the legend label because color is the aesthetic that is being represented in the legend.

```
ggplot(gap_07, aes(x = gdpPercap, y = lifeExp, color = continent)) +
  geom_point(size = 2) +
  scale_x_log10() +
  theme_light() +
  theme(legend.position = c(0.1, 0.85),
        legend.key = element_blank(),
        axis.text = element_text(size = 12),
        axis.title = element_text(size = 14)) +
  labs(x = "Per capita GDP",
       y = "Life Expectancy",
       title = "2007 Life Expectancy and GDP",
       color = "Continent") +
  ggsave("life_exp_gdp_2007_custom.png", width = 7, height = 7)
```

The end result is a very polished looking plot!

6 Conclusion

6.1 Conclusion

In this course, we've covered principles of effective visualizations and how to create visualizations in ggplot2. We've learned about types of variables and what encodings work for each type, as well as important considerations of perceptual effectiveness and the appropriate use of color. ggplot2's grammar of graphics uses the idea of encodings by allowing you to map data variables to aesthetics. We discussed the fundamental aspects of the grammar of graphics – data, aesthetics, geoms, stats, facets, scales, and coordinate systems. Then we used ggplot2's grammar of graphics to create visualizations which we customized and polished with ggplot2's theme system.

There are plenty of details of ggplot2 that we did not cover, but after this course you should be well equipped to explore additional geoms, coordinate systems, etc.

As of version 2, ggplot2 features a mechanism for writing extensions. The extension mechanism lets you write your own ggplot2 components like stats and geoms. People have used this extension mechanism to create network graphs, interactive plots, animations, emoji geoms, and more.

The extension mechanism is a more advanced topic, and you can read more about it at `http://docs.ggplot2.org/dev/vignettes/extending-ggplot2.html`. You can also see extensions in action at `https://www.ggplot2-exts.org/`.

I hope you've enjoyed the topics we've covered in this course and are excited to go on and create more visualizations with ggplot2. Thank you for watching!

7 Photograph sources

All photographs come from unsplash.com

- Many windows - Vladimir Kudinov
- Many books - Patrick Tomasso
- Book with glasses - Darius Sankowski
- Question mark - Evan Dennis
- Flower - Blair Connelly
- Flower - Bill Williams
- Flower - H K
- Flower - Pamela Nhlengethwa
- Stack of rocks - Deniz Altindas
- Ruler - Dawid Malecki
- Alphabet - Amador Loureiro
- Macaroons - Tatiana Lapina
- Hands in the air - William White
- Chart on screen - Negative Space
- Card catalog - Sanwal Deen
- Fishing - Alan Bishop
- Eye - Sean Brown
- Dew on leaf - kazuend
- Road - Aleksandr Kozlovskii