# Hertie School/SCRIPTS Data Science Workshop Series

Session 3: Data Visualization with `ggplot2`

*Therese Anders*[*]
*Allison Koh*[†]

*March 6, 2020*

## Introduction

The goal of this workshop series is to create a space for faculty and researchers at the Hertie School and SCRIPTS to efficiently catch up with recent developments in data science topics and tools. In the spring semester, we will focus on the basics of data manipulation and visualization in R, text as data, the gathering of information from online sources, and principles of machine learning.

## Basic Principles of `ggplot2`

In this first part of the workshop, we will go over basic principles of `ggplot2`. We will work with data from the `gapminder` package. First, install `gapminder` and get an overview over the data. The dataset contains information on life expectancy, GDP per capita, and population by country from 1952 to 2007 in increments of 5 years. Let's use the help function to get an overview of the data.

```
# install.packages("gapminder")
library(gapminder)
?gapminder # getting an overview
```

Start by making a copy of the original data in a data frame called `df`. Then use the `str()` function to get an overview over the variable types in the data frame. The dataframe has 1704 observations and 6 variables.

```
df <- gapminder
str(df)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 222
##  $ gdpPercap: num  779 821 853 836 740 ...
```

### `ggplot2` package

`ggplot2` was developed by Hadley Wickham based on Leland Wilkinson's "grammar of graphics" principles. According to the "grammar of graphics," you can create each graph from the following components: "a data set, a set of geoms–visual marks that represent data points, and a coordinate system" (Wilkinson 2012). You can access the data visualization with `ggplot2` cheat sheet here.

For most applications, the code to produce a graph in `ggplot2` is roughly structured as follows:

[*]Instructor, Hertie School/SCRIPTS, anders@hertie-school.org.
[†]Teaching Assistant, Hertie School, kohallison3@gmail.com.

```
ggplot(data = , aes(x = , y = , color = , linetype = )) +

geom() +

[other graphical parameters, e.g. title, color schemes, background]
```

- `ggplot()`: Function to initiate a graph in `ggplot2`.
- `data`: Specifies the data frame from which the plot is produced.
- `aes()`: Specifies aesthetic mappings that describe how variables are mapped to the visual properties of the graph. The minimum value that needs to be specified (for univariate data visualization) is the `x` parameter, where `x` specifies the variable to be plotted on the x-axis. Analogously, the `y` parameter specifies the variable to be plotted on the y-axis. Other examples include the `color` parameter, which specifies the variable to be onto different colors, or the `linetype` parameter, which specifies the variable to be mapped onto different line types in case of line graphs.
- `geom()`: Specifies the type of plot to use. There are many different geoms ("geometric objects") to be specified with the `geom()` layer. Some of the most common ones include `geom_point()` for scatterplots, `geom_line()` for line graphs, `geom_boxplot()` for Boxplots, `geom_bar()` for bar plots for discrete data, and `geom_histogram()` for continuous data.

For an overview of the most important functions and geoms available through `ggplot2`, see the `ggplot2` cheat sheet.

`ggplot2` is part of the `tidyverse` collection of `R` packages. You can load the entire collection by downloading the `tidyverse` package and loading it using the `library(tidyverse)` command, but for this workshop we will be downloading and calling each package separately.

```r
# install.packages("ggplot2")
library(ggplot2)
```

## Showing data distributions

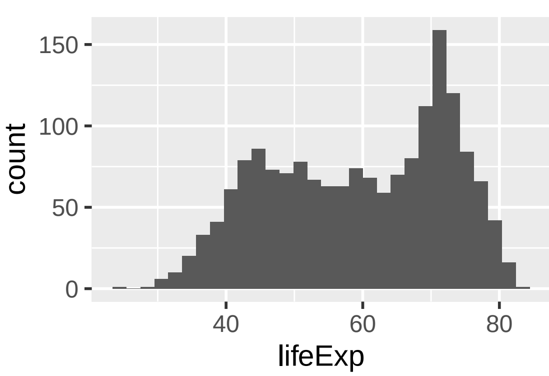
### Histograms

Histograms graph the distribution of continuous variables. In this first example, we graph the distribution of the life expectancy variable (i.e. `lifeExp`).

```r
summary(df$lifeExp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   23.60   48.20   60.71   59.47   70.85   82.60
```

```r
ggplot(df,
       aes(x = lifeExp)) +
  geom_histogram()
```



**Question 1** Can you make sense of this graph? What is plotted on the x-axis? What is plotted on the y-axis? What specifies the width of each bar? What specifies the height of each bar?

*A histogram plots the distribution of a variable. The x-axis specifies the values of the variable. The y-axis specifies the number of observations for each value (or group of values) of the variable. The width of the bar specifies which values of the variable are grouped into one bin. The height of the bar specifies the number of observations in each bin.*

**Question 2** Which conclusions do you draw from the histogram above about the distribution of life expectancy in the world?

*The distribution is not normal (i.e. not a bell curve). It is bimodal with a skew to the left. There is a cluster of country-year observations that has a lower life expectancy (approximately 45-60 years), and a cluster of countries with much higher life expectancies (approx 70 years).*

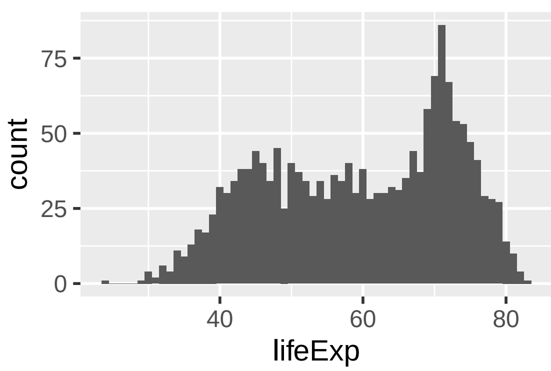
### Adjusting the number of bins

The default number of bins is 30, which means that the entire range of the variable (here 23.60 to 82.60) is split into 30 equally spaced bins. We can change the number of bins manually. Below, we specify 60 bins to approximate a binwidth of 1 year, taking into account the range of the variable `lifeExp`.

```
min(df$lifeExp) - max(df$lifeExp) # 60 years
```

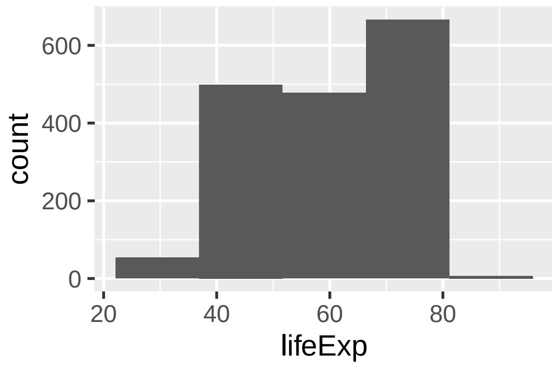
```
## [1] -59.004
```

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_histogram(bins = 60)
```




What if we specified just 5 bins?

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_histogram(bins = 5)
```

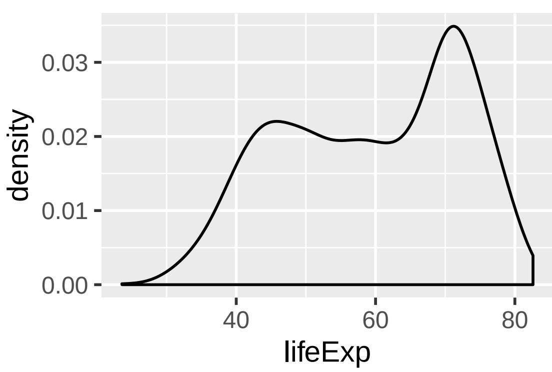



## Density plots

We saw that the shape of the distribution is highly influenced by how many bins we specify. If we specify too few bins, we run the risk of masking a lot of variation within the bins. If we specify too many bins, we trade parsimony for detail–which might make it harder to draw conclusions about the overall distribution of the variable of interest from the graph.

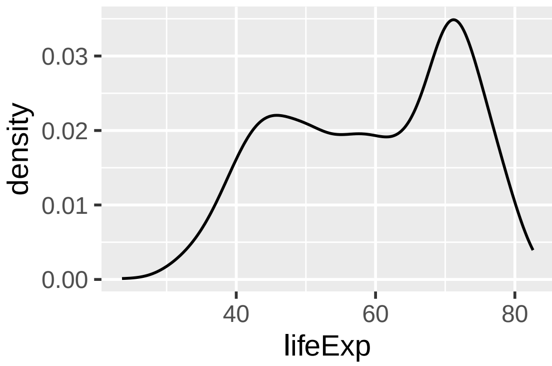
Density plots are continuous alternatives to histograms that do not rely on bins. We will cover details about the mechanics behind density plots and their estimation here. Just know that we can interpret the height of the density curve in a similar way that we interpreted the height of the bars in a histogram: The higher the curve, the more observations we have at that specific value of the variable of interest. In this first example, we use the `geom_density()` function to create the density plot.

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_density()
```




If you do not want the density graph to be plotted as a closed polygon, you can instead use the `geom_line()` geometric object function with the `stat = "density"` parameter.

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density")
```

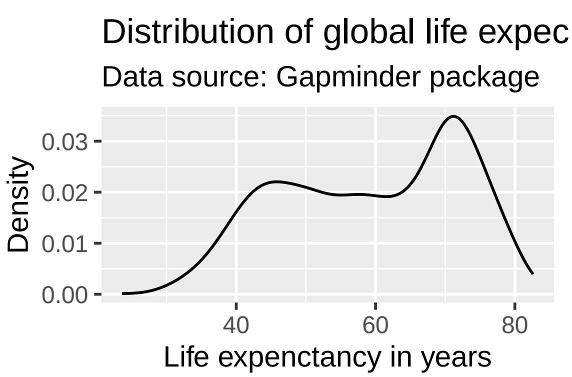



### Controlling the appearance of graphs

The default graphs we have produced so far are not (yet) ready for publication. In particular, they lack informative labels. In addition, we might want to change the appearance of the graph in terms of size, color, linetype, etc.

### Adding title, subtitle, and axes titles

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expenctancy in years",
       y = "Density")
```

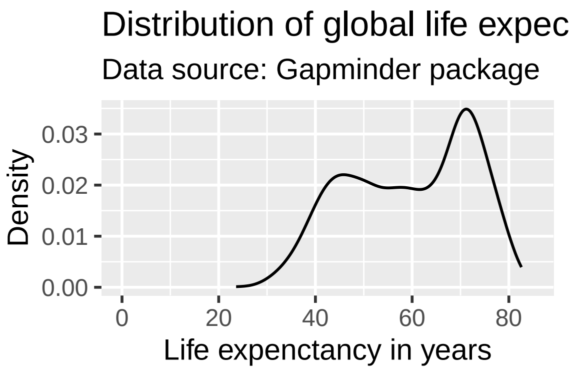



### Adjusting the range of the axes

By default, `ggplot()` adjusted the x-axis to start not at zero but at approximately 23 to reduce the amount of empty space in the plot. We can manually adjust the range of the axes using the `coord_cartesian()` parameter.

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
```

```
       x = "Life expectancy in years",
       y = "Density") +
  coord_cartesian(xlim = c(0, 85))
```

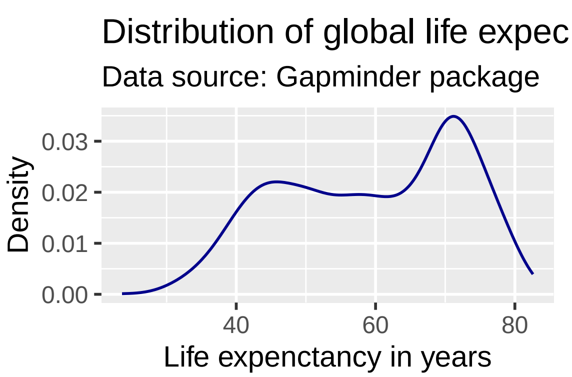



**Caution!!** You will sometimes see the command `scale_y_continuous(limits = c(0, 85))` instead of `coord_cartesian(ylim = c(0, 85))`. Note that these are not the same. `coord_cartesian()` only adjusts the range of the axes (it "zooms" in and out), while `scale_y_continuous(limits = c())` subsets the data. For density plots, this does not make a difference. But there are other examples where it alters the actual shape of the graph, rather than just the part of the graph that is visible.

### Changing the color

Any changes to the appearance of the curve itself are made within the argument that specifies the geometric object to be plotted, here `geom_line()`. `R` knows many colors by name; for a great overview see http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf.

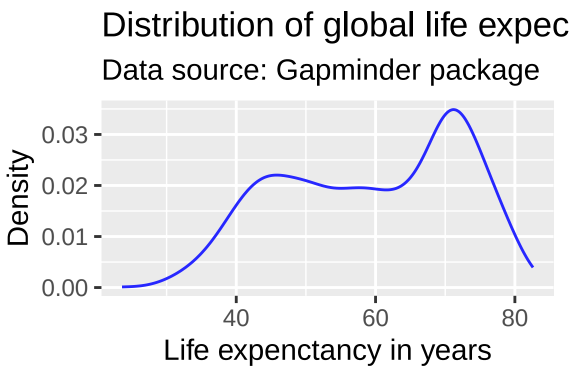
```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density",
            color = "darkblue") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density")
```




We can also use hexadecimal or RGB (red, green, blue) strings to specify colors. There are plenty of online tools to pick colors and extract hexadecimal or RBG strings. One of my favorites is http://www.colorhexa.com. This online tool allows you to specify a color name, hexadecimal, or RGB string, and returns information on color schemes, complementary colors, as well as alternative shades, tints, and tones. It also offers a color blindness simulator.

Suppose, I like the general tone of the darkblue color above, but am worried that it is a bit too dark for my plot. I enter the color "darkblue" into the search field at http://www.colorhexa.com and look for a brighter alternative. Suppose I really like the color displayed in the second tile from the left on the tints scale. I can extract this color's hexadecimal value of `#2727ff` by hovering over the tile of that color.

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density",
            color = "#2727ff") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density")
```

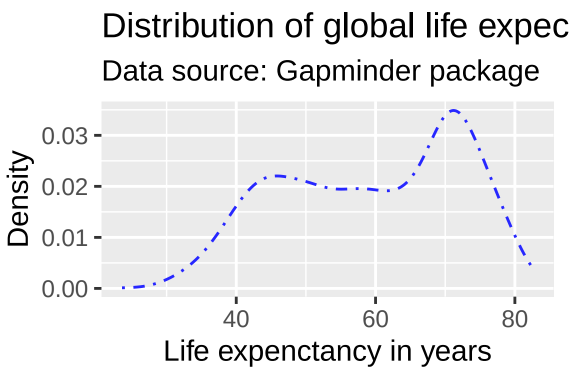



We will talk more about color schemes later in the workshop.

## Changing the line type

We can adjust the type of the line via the `linetype` parameter within `geom_line()`. For an overview of line types see http://sape.inf.usi.ch/quick-reference/ggplot2/linetype.

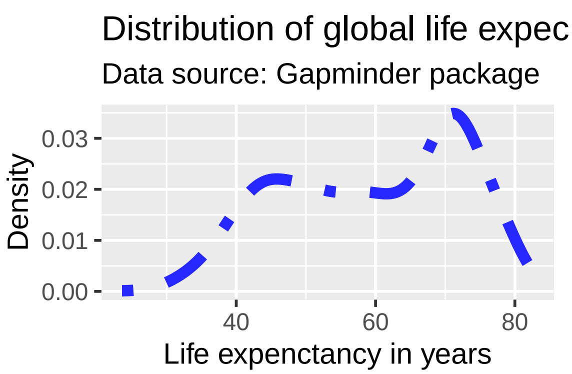
```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density",
            color = "#2727ff",
            linetype = "dotdash") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density")
```



### Changing width and opacity of the line (not shown in class)

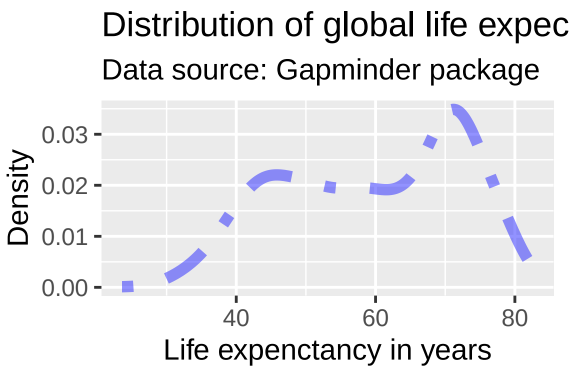
Changing the width of the line We can adjust the *width* of the line via the `size` parameter within `geom_line()`. Note that the `size` parameter is universal in the way that it controls line width in line plots and point size in scatter plots.

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density",
            color = "#2727ff",
            linetype = "dotdash",
            size = 2) +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expenctancy in years",
       y = "Density")
```




We can adjust the *opacity* of the line via the `alpha` parameter within any geometric object. The `alpha` parameter ranges between zero and one. Adjusting the opacity of the geometric objects is especially important when plotting multiple lines (or objects) in the same graph to reduce overplotting.

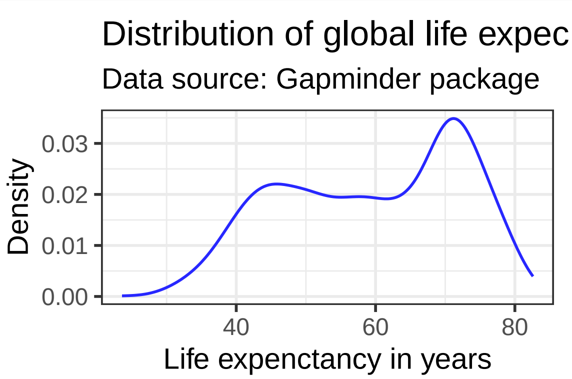
```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density",
            color = "#2727ff",
            linetype = "dotdash",
            size = 2,
            alpha = 0.5) +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expenctancy in years",
       y = "Density")
```




### Themes

We can alter the appearance of any element in the plot. Below, we change the pre-specified `theme` that `ggplot2` uses to determine the appearance of the plot. Popular options are `theme_bw()` or `theme_minimal()`. For a full list of themes, see https://ggplot2.tidyverse.org/reference/ggtheme.html. We can change all parameters manual using the `theme()` function.

```
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density",
            color = "#2727ff") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expenctancy in years",
       y = "Density") +
  theme_bw()
```




## Graphing distributions across groups

### Using different colors

Sometimes, we want to compare distributions across different groups in our data set. Suppose, we wanted to assess the distribution of the life expectancy on different continents. We can use the `table()` function to get an overview over the groups in our data.

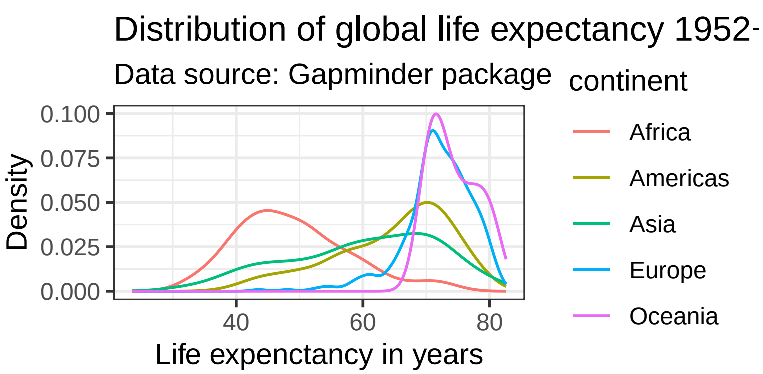
```
table(df$continent)
```

```
##
##   Africa Americas     Asia   Europe  Oceania
```

```
##       624       300       396       360        24
```

We pass a separate color to the distribution of the `lifeExp` for each continent by specifying the `color` parameter within the aesthetics. Remember, to remove the `color` parameter from the `geom_line()` function. The ability to pass a second variable to the graph with just one aesthetic (here: color) is where the true power of `ggplot2` for data visualization lies.

```
ggplot(df,
       aes(x = lifeExp,
           color = continent)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expenctancy in years",
       y = "Density") +
  theme_bw()
```

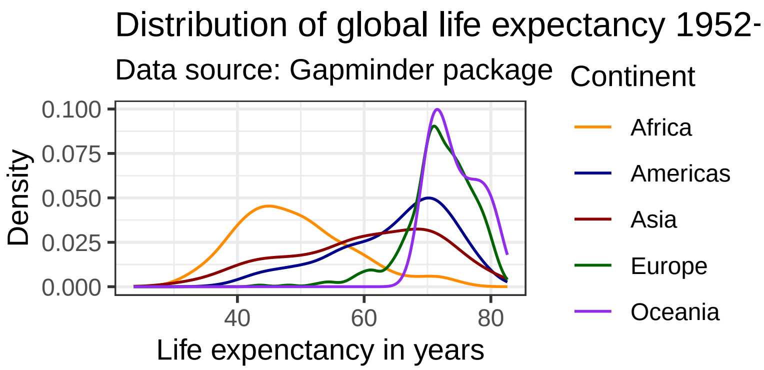



**Question 3** What is the difference between specifying the `color` parameter outside the `aes()` argument versus within the `aes()` argument?

*If the color parameter is specified outside the* `aes()` *argument, one color is passed all geometric objects of the same type. If the color parameter is specified within the* `aes()` *argument, different colors are passed to each value of the variable that is passed to the* `color` *parameter. A separate geometric object will be plotted for value–each in a different color.*

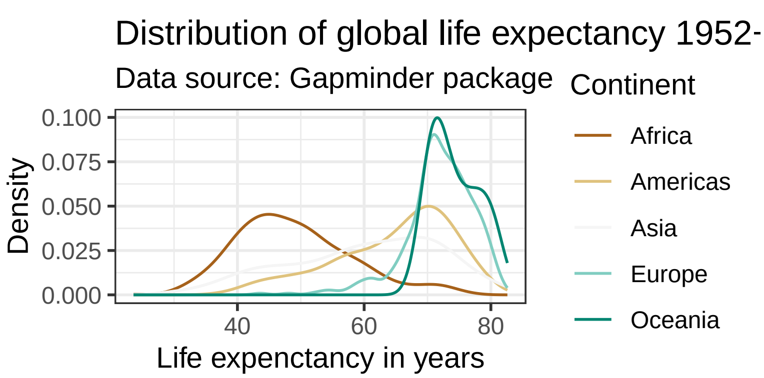
We can adjust the colors used in the plot in a variety of ways. Below, we first use the `scale_color_manual()` function. This will change the colors in both the plot and the legend, based on our manual specification. Within the `scale_color_manual()` argument, we can also specify a name and labels for the legend.

```
ggplot(df,
       aes(x = lifeExp,
           color = continent)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expenctancy in years",
       y = "Density") +
  theme_bw() +
  scale_color_manual(values = c("Africa" = "darkorange",
                                "Americas" = "darkblue",
                                "Europe" = "darkgreen",
                                "Asia" = "darkred",
                                "Oceania" = "purple2"),
                     name = "Continent")
```




There are a ton of resources and packages with pre-defined color schemes. The most popular is www.colorbrewer2.org. You can either pick the desired colors manually, or use the `scale_color_brewer()` function in `ggplot2()`.

```
ggplot(df,
       aes(x = lifeExp,
           color = continent)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density") +
  theme_bw() +
  scale_color_brewer(palette = "BrBG",
                     name = "Continent")
```



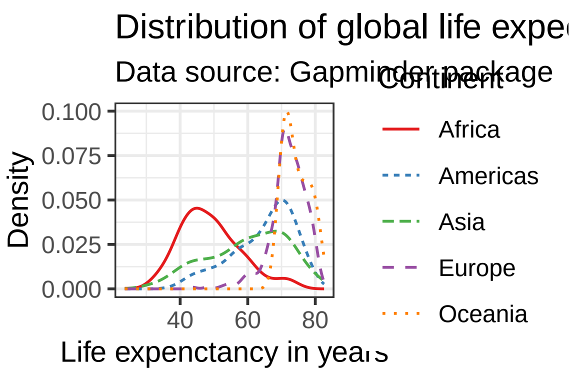

Check out the list of color palettes compiled by Emil Hvitfeldt. There is even a LaCroix inspired color scheme available using the package `LaCroixColoR`! Another popular option are the color schemes from the `viridis` package due to their desirable properties with respect to colorblindness and printability.

### Using different linetypes

Many academic journals will only accept graphs on a gray scale. This means that color will not be enough to differentiate five lines. We can use different line types instead by specifying the `linetype` parameter within the `aes()` argument. This also makes the graph more color blind friendly. Notice below that in order to combine the legends for the `lintype` and `color` aesthetics, we need to pass the same name within the `scale` function.

```
ggplot(df,
       aes(x = lifeExp,
```

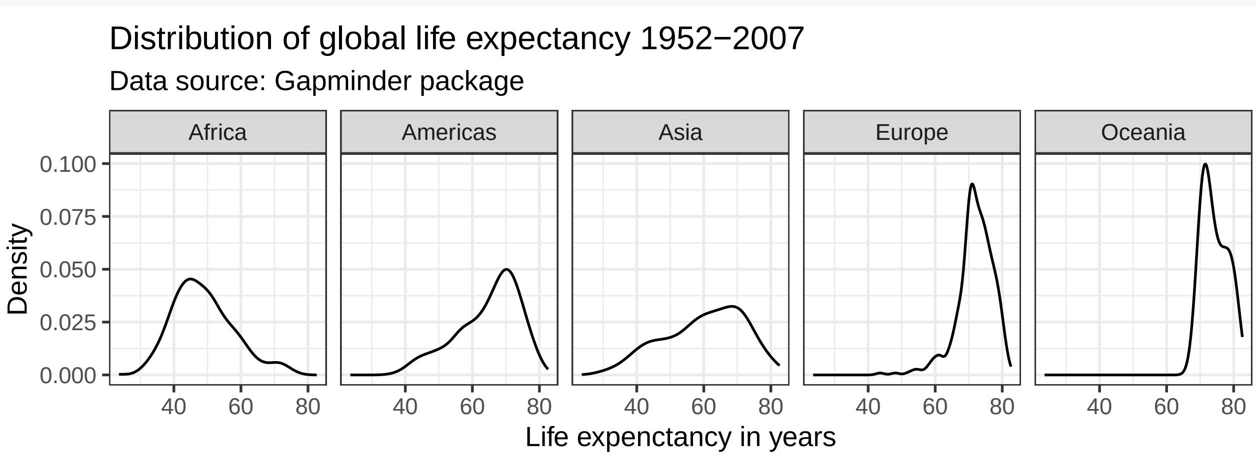
```r
           color = continent,
           linetype = continent)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density") +
  theme_bw() +
  scale_color_brewer(palette = "Set1",
                     name = "Continent") +
  scale_linetype_discrete(name = "Continent")
```




## Faceting

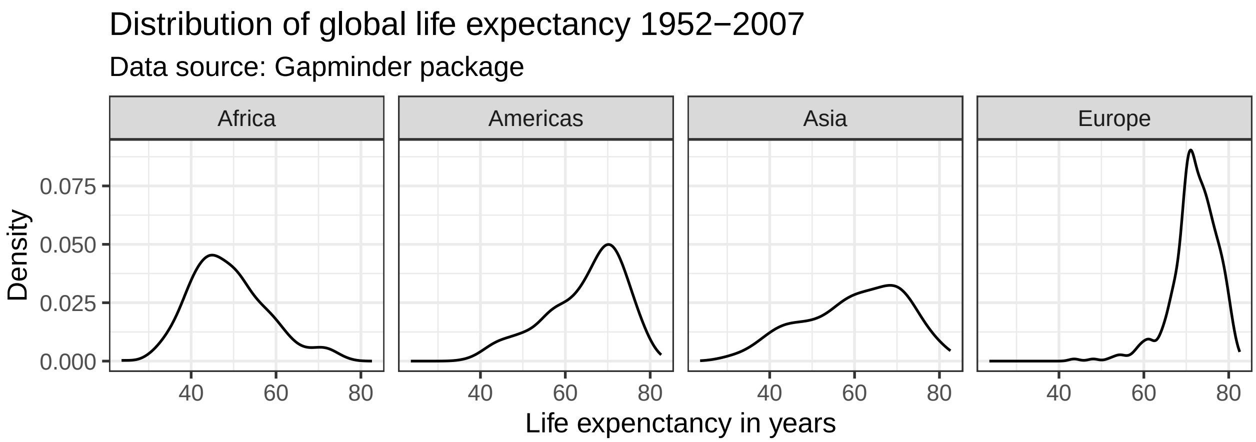
Another option to graph different groups is to use faceting. This means to plot each value of the variable upon which we facet in a different panel within the same plot. Here, we will use the `facet_wrap()` function. We could also use the `facet_grid()` which allows faceting across more than one variable.

```r
ggplot(df,
       aes(x = lifeExp)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density") +
  theme_bw() +
  facet_wrap(~ continent, nrow = 1)
```



Suppose, we wanted to exclude the plot for Oceania, since it is only comprised of Australia and New Zealand. We can either create a new subsample data frame, or use the `subset()` command directly within `ggplot()`.

```
ggplot(subset(df, continent != "Oceania"),
       aes(x = lifeExp)) +
  geom_line(stat = "density") +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density") +
  theme_bw() +
  facet_wrap(~ continent, nrow = 1)
```



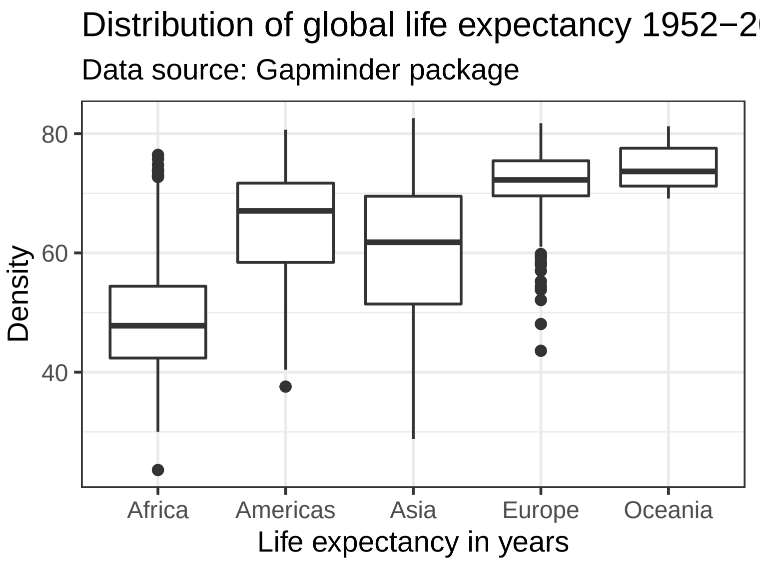

## Boxplots

Another way to show the distribution of variables across groups are boxplots. Boxplots graph different properties of a distribution:

- The borders of the box denote the 25th and 75th percentile.
- The line within the box denotes the median.
- The position of the whiskers (vertical lines) denote the first quartile value minus 1.5 times the interquartile range and the third quartile value plus 1.5 times the interquartile range. We will not go into details here.
- Dots denote outliers (values that lie outside the whiskers), if applicable.

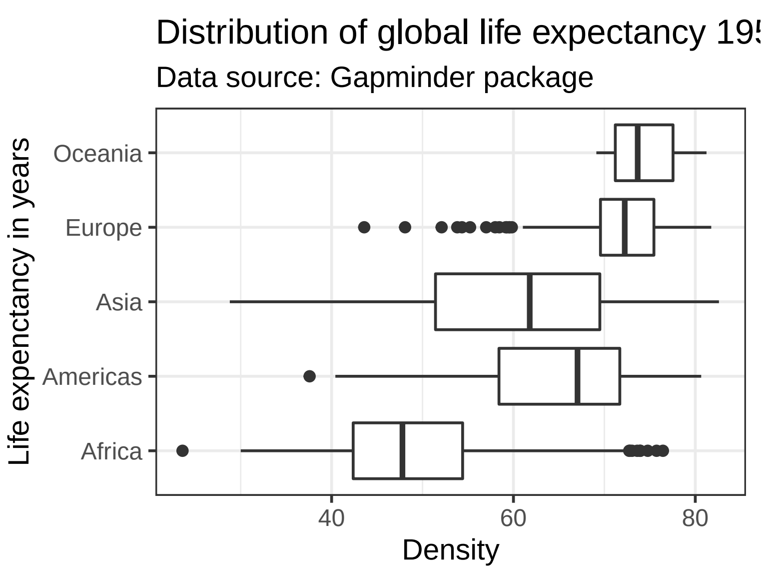
In `ggplot2` we can graph boxplots across multiple variables using the `geom_boxplot()` geometric object. Here, the continuous variable (i.e. `lifeExp`) should be specified as the `y` variable, and the categorical variable (i.e. `continent`) as the `x` variable.

```
ggplot(subset(df),
       aes(x = continent,
           y = lifeExp)) +
  geom_boxplot() +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density") +
  theme_bw()
```




We can flip the axes by using the `coord_flip()` command.

```
ggplot(subset(df),
       aes(x = continent,
           y = lifeExp)) +
  geom_boxplot() +
  labs(title = "Distribution of global life expectancy 1952-2007",
       subtitle = "Data source: Gapminder package",
       x = "Life expectancy in years",
       y = "Density") +
  theme_bw() +
  coord_flip()
```




## Saving plots

We can output your plots to many different format using the `ggsave()` function, including but not limited to `.pdf`, `.jpeg`, `.bmp`, `.tiff`, or `.eps`. Here, we output the graph as a Portable Network Graphics (.png) file. We can specify the size of the output graph as well as the resolution in dots per inch (dpi). If no graph is specified, `ggsave()` will save the last graph that was executed. For us, this is the boxplot in horizontal orientation. If we no not specify the complete file path, the plot will be saved to your working directory.

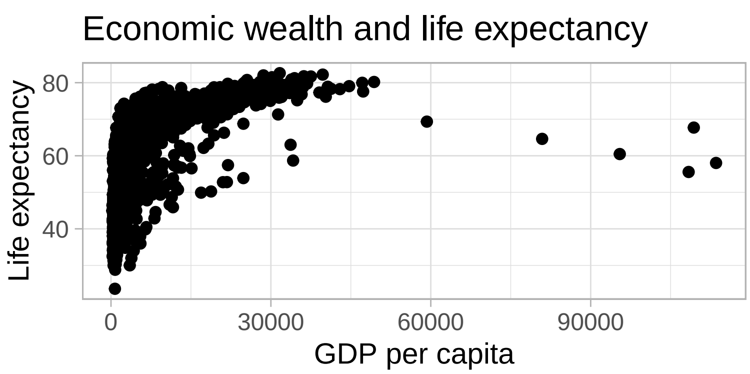
```
ggsave("boxplot_lifeexp_continent.png", width = 6, height = 3, dpi = 400)
```

## Showing relationships in data

### Scatter plots

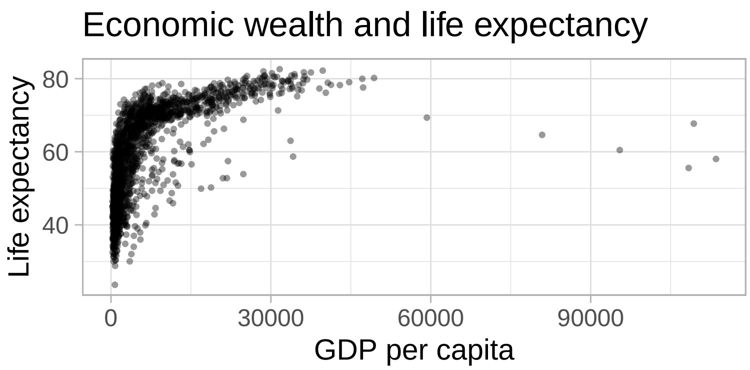
In their basic form, scatter plots are used to display values of two variables on a Cartesian coordinate system. Below, we inspect the relationship between GDP per capita and life expectancy.

```
ggplot(df,
       aes(x = gdpPercap,
           y = lifeExp)) +
  geom_point() +
  labs(title = "Economic wealth and life expectancy",
       x = "GDP per capita",
       y = "Life expectancy") +
  theme_light()
```




The plot above shows a large amount of clustering (and overplotting) on the left side of the plot, while the right side of the plot is sparsely populated with data. This makes it hard to gauge the relationship between the two variables. Below, we make a number of adjustments to the graph to better display the relationship.

```
ggplot(df,
       aes(x = gdpPercap,
           y = lifeExp)) +
  geom_point(alpha = 0.4,
             size = 0.5) +
  labs(title = "Economic wealth and life expectancy",
       x = "GDP per capita",
       y = "Life expectancy") +
  theme_light()
```

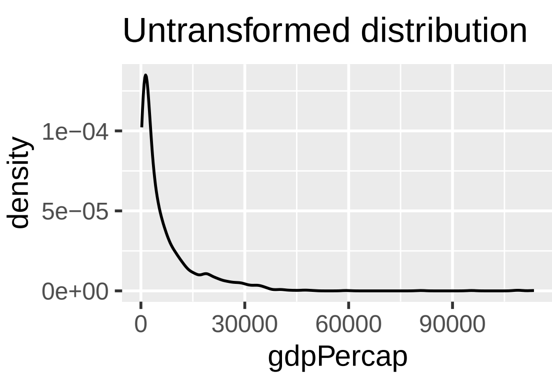



### Scaling the data

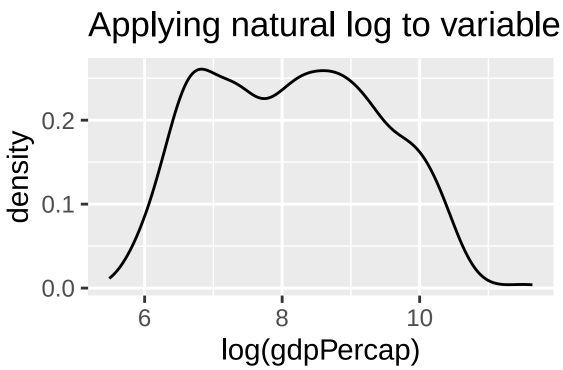
One reason why the plot above is hard to read is rooted in the shape of the distribution of the GDP per capita variable. GDP per capita has a strong right skew. We can correct for this skew and transform the variable to have a more "normal" distribution by taking the natural logarithm. There are multiple ways to do this.

1. Create a new variable [not shown below]
2. Take the natural logarithm within the `aes()` statement when specifying the variable to be displayed.
3. Using (scales)[https://ggplot2.tidyverse.org/reference/scale_continuous.html] to transform the display. Note that the data is transformed before properties such as the range of the axis are determined.

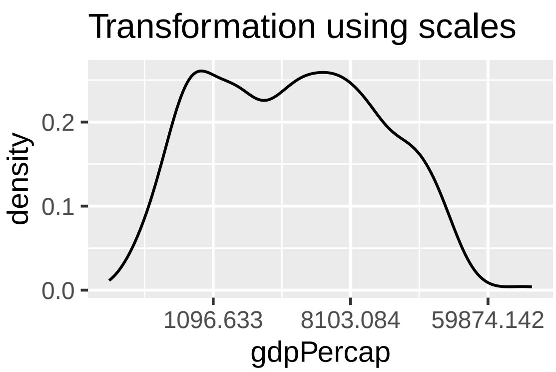
```
ggplot(df,
       aes(x = gdpPercap)) +
  geom_line(stat = "density") +
  labs(title = "Untransformed distribution")
```




```
ggplot(df,
       aes(x = log(gdpPercap))) +
  geom_line(stat = "density") +
  labs(title = "Applying natural log to variable directly")
```




```
ggplot(df,
       aes(x = gdpPercap)) +
  geom_line(stat = "density") +
  labs(title = "Transformation using scales") +
  scale_x_continuous(trans = "log")
```

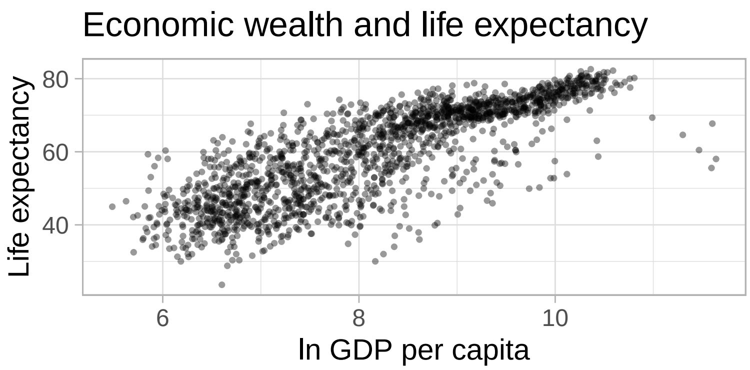



**Question 4** Can you explain the differences between the plot applying the natural log to the variable within the `aes()` function versus using `scale_x_continuous()`.

*Transforming the variable using the natural logarithm within `aes()` causes the x-axis to be displayed in log values. Using `scale_x_continuous()`, the data is transformed in the same way, however, the x-axis is displayed in the original, non-logged version.*

We can use the same principle in bivariate (or multivariate) displays of data. `scale` transformations are extremely helpful, especially when transforming color scales. However, below, I use the transformation on the variable and reflect it in the axis label clarify that it is the relationship between life expectancy and the natural log of GDP per capita that has a strong positive relationship.

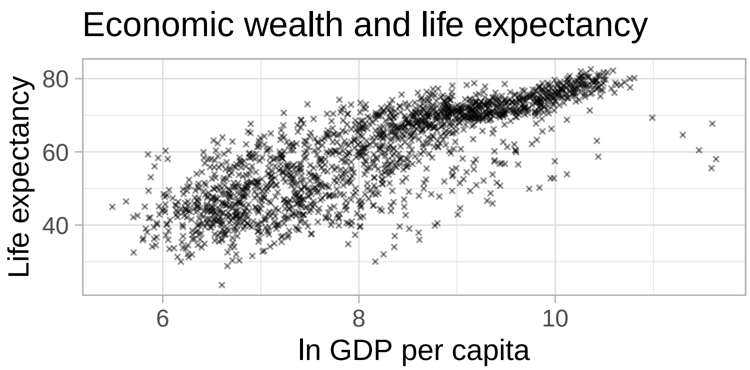
```
ggplot(df,
       aes(x = log(gdpPercap),
           y = lifeExp)) +
  geom_point(alpha = 0.4,
             size = 0.5) +
  labs(title = "Economic wealth and life expectancy",
       x = "ln GDP per capita",
       y = "Life expectancy") +
  theme_light()
```




### Shape

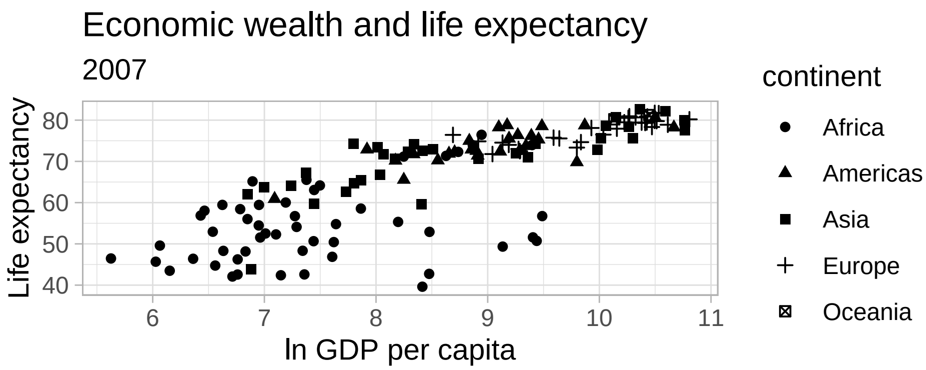
We can adjust the default symbol used by `ggplot2` to display the points. The parameter is called `shape`.

```
ggplot(df,
       aes(x = log(gdpPercap),
           y = lifeExp)) +
  geom_point(alpha = 0.4,
             size = 0.5,
             shape = 4) +
  labs(title = "Economic wealth and life expectancy",
       x = "ln GDP per capita",
       y = "Life expectancy") +
  theme_light()
```




We can also have groups of data displayed using different point shapes. Below, we group by continent. We subset the data to just the year 2007 to de-clutter the plot.

```
ggplot(subset(df, year == 2007),
       aes(x = log(gdpPercap),
           y = lifeExp,
           shape = continent)) +
  geom_point() +
  labs(title = "Economic wealth and life expectancy",
       subtitle = "2007",
       x = "ln GDP per capita",
       y = "Life expectancy") +
  theme_light()
```



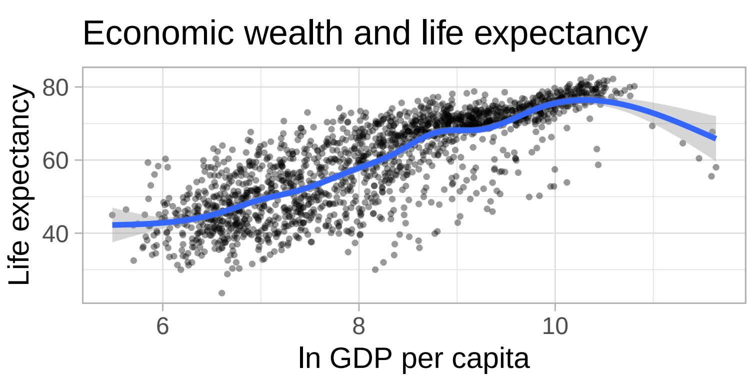

### Adding trend lines

The plot above illustrates a strong positive relationship between GDP per capita and life expectancy. We can highlight the direction and strength of the relationship by adding a trend line using the `geom_smooth()` aesthetic.

The default smoothing method is `loess` for less than 1,000 observations and `gam` (Generalized Additive Models) for observations greater or equal to 1,000. `ggplot2` informs us which smoothing method was used via a message. By default, a 95% confidence interval is added to the trend line. It shows that the negative relationship at higher values of GDP per capita has a much lower precision than the positive relationship we observe for the majority of the observations.

```
ggplot(df,
       aes(x = log(gdpPercap),
           y = lifeExp)) +
  geom_point(alpha = 0.4,
             size = 0.5) +
  labs(title = "Economic wealth and life expectancy",
       x = "ln GDP per capita",
       y = "Life expectancy") +
  theme_light() +
  geom_smooth()
```

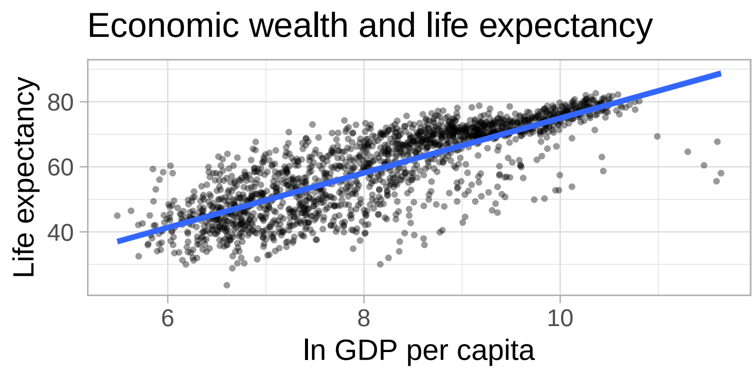
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```




Alternatively, we can add a linear regression trend line to the data.

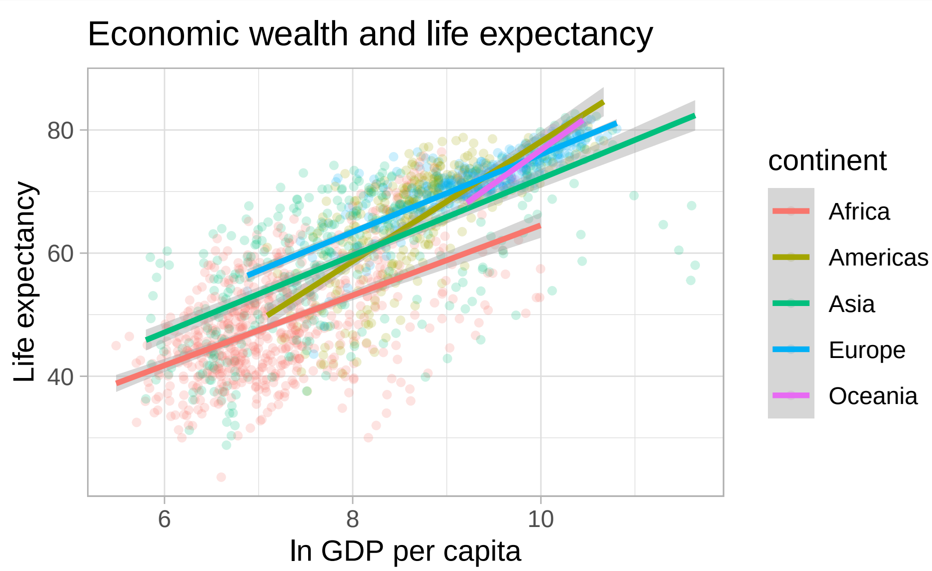
```
ggplot(df,
       aes(x = log(gdpPercap),
           y = lifeExp)) +
  geom_point(alpha = 0.4,
```

```
             size = 0.5) +
  labs(title = "Economic wealth and life expectancy",
       x = "ln GDP per capita",
       y = "Life expectancy") +
  theme_light() +
  geom_smooth(method = "lm")
```




Finally, we can display separate trendlines for groups of data. For example, suppose we wanted to know how the relationship between GDP per capita and life expectancy varies by continent. We can pass the grouping variable to the `color` (and/or `linetype`) parameter within the `aes()` function. Below, I further reduce the opacity of the points to avoid overplotting. Note that the color grouping is passed to both the `geom_point()` and the `geom_smooth()` aesthetic.

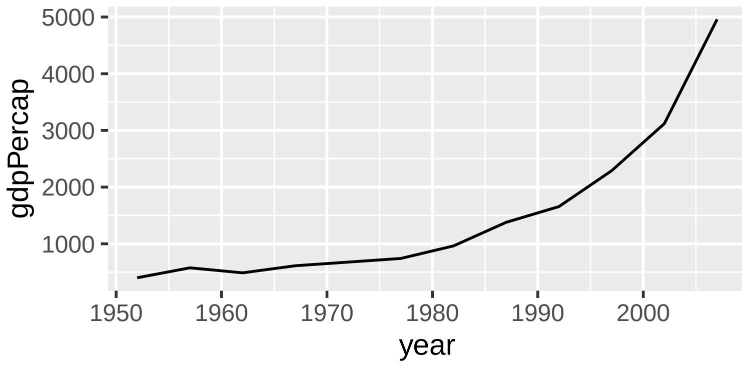
```
ggplot(df,
       aes(x = log(gdpPercap),
           y = lifeExp,
           color = continent)) +
  geom_point(alpha = 0.2,
             size = 1) +
  labs(title = "Economic wealth and life expectancy",
       x = "ln GDP per capita",
       y = "Life expectancy") +
  theme_light() +
  geom_smooth(method = "lm")
```



### Line plots

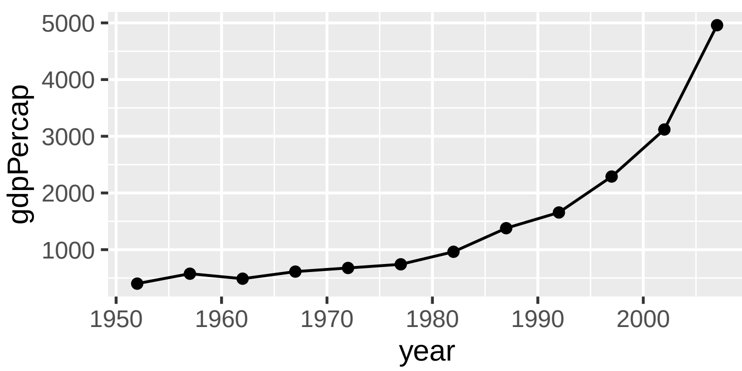
Line plots are particularly useful for time series data. Below, we will graph the GDP per capita development of China from 1952 to 2007. We select the data for China by using the `subset()` function on the original data frame.

```
ggplot(subset(df, country == "China"),
       aes(x = year,
           y = gdpPercap)) +
  geom_line()
```

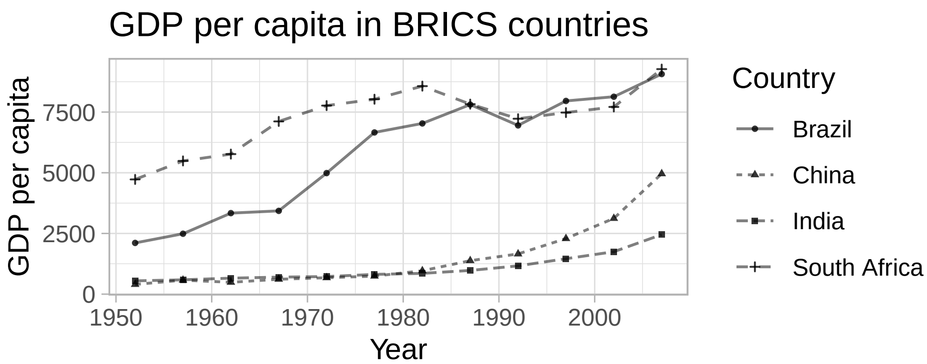


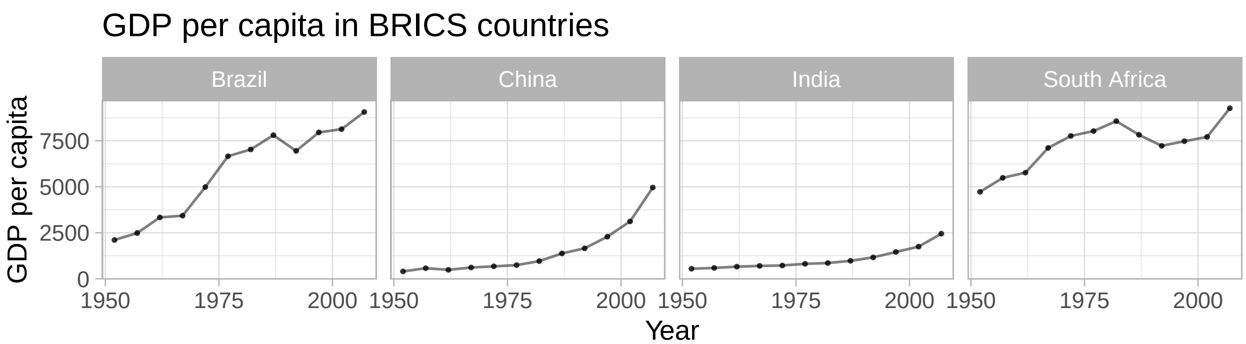

We can add points to the line to highlight which observations are available in the underlying data.

```
ggplot(subset(df, country == "China"),
       aes(x = year,
           y = gdpPercap)) +
  geom_line() +
  geom_point()
```




**Practice 2** Create a plot to compare the GDP per capita development of the BRICS countries (Brazil, Russia, India, China, South Africa). Unfortunately, Russia (or the Soviet Union) is not part of the `gapminder` data, so we cannot display it in the plot. Please create a publication-ready graph that can be printed using grayscale.




# Data Wrangling for Visualization

*Why data wrangling in a visualization workshop?* In practice, data visualization is only the last part in a long stream of data gathering, cleaning, wrangling, and analysis.

`ggplot2` is the most powerful when we have "tidy" data. There are three rules for tidy data, based on Hadley Wickham's R for Data Science.

1. "Each variable must have its own column."
2. "Each observation must have its own row."
3. "Each value must have its own cell."

If the data are in a tidy format, we can pass separate variables to separate geometric objects (`geoms`) and create layered displays of multiple variables. Thus an important component of creating interesting data visualizations is to get the data to be in the right format. We will also learn a number of new data visualization tools as part of the data wrangling section, including

- Bar charts
- Error bars on plots

RStudio offers a great Data wrangling cheat sheet you should take a look at.

## Introduction to `dplyr`

`dplyr` does not accept tables or vectors, just data frames (similar to `ggplot2`)! `dplyr` uses a strategy called "Split - Apply - Combine". Some of the key functions include:

- `select()`: Subset columns.
- `filter()`: Subset rows.
- `arrange()`: Reorders rows.
- `mutate()`: Add columns to existing data.

- `summarise()`: Summarizing data set.
- joins: Combine two data frames together

First, lets dowload the package and call it using the `library()` function.

```
# install.packages("dplyr")
library(dplyr)
```

Today, we will be working with a data set from the `hflights` package. The data set contains all flights from the Houston IAH and HOU airports in 2011. Install the package `hflights`, load it into the library, extract the data frame into a new object called `raw` and inspect the data frame.

**NOTE:** The `::` operator specifies that we want to use the *object* `hflights` from the *package* `hflights`. In the case below, this explicit programming is not necessary. However, it is useful when functions or objects are contained in multiple packages to avoid confusion. A classic example is the `select()` function that is contained in a number of packages besides `dplyr`.

```
# install.packages("hflights")
library(hflights)
raw <- hflights::hflights
str(raw)
```

```
## 'data.frame':    227496 obs. of  21 variables:
##  $ Year             : int  2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 ...
##  $ Month            : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ DayofMonth       : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ DayOfWeek        : int  6 7 1 2 3 4 5 6 7 1 ...
##  $ DepTime          : int  1400 1401 1352 1403 1405 1359 1359 1355 1443 1443 ...
##  $ ArrTime          : int  1500 1501 1502 1513 1507 1503 1509 1454 1554 1553 ...
##  $ UniqueCarrier    : chr  "AA" "AA" "AA" "AA" ...
##  $ FlightNum        : int  428 428 428 428 428 428 428 428 428 428 ...
##  $ TailNum          : chr  "N576AA" "N557AA" "N541AA" "N403AA" ...
##  $ ActualElapsedTime: int  60 60 70 70 62 64 70 59 71 70 ...
##  $ AirTime          : int  40 45 48 39 44 45 43 40 41 45 ...
##  $ ArrDelay         : int  -10 -9 -8 3 -3 -7 -1 -16 44 43 ...
##  $ DepDelay         : int  0 1 -8 3 5 -1 -1 -5 43 43 ...
##  $ Origin           : chr  "IAH" "IAH" "IAH" "IAH" ...
##  $ Dest             : chr  "DFW" "DFW" "DFW" "DFW" ...
##  $ Distance         : int  224 224 224 224 224 224 224 224 224 224 ...
##  $ TaxiIn           : int  7 6 5 9 9 6 12 7 8 6 ...
##  $ TaxiOut          : int  13 9 17 22 9 13 15 12 22 19 ...
##  $ Cancelled        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ CancellationCode : chr  "" "" "" "" ...
##  $ Diverted         : int  0 0 0 0 0 0 0 0 0 0 ...
```

## Using `select()` and introducing the Piping Operator `%>%`

Using the so-called **piping operator** will make the `R` code faster and more legible, because we are not saving every output in a separate data frame, but passing it on to a new function. First, let's use only a subsample of variables in the data frame, specifically the year of the flight, the airline, as well as the origin airport, the destination, and the distance between the airports.

Notice a couple of things in the code below:

- We can assign the output to a new data set.
- We use the piping operator to connect commands and create a single flow of operations.
- We can use the select function to rename variables.

- Instead of typing each variable, we can select sequences of variables.
- Note that the `everything()` command inside `select()` will select all variables.

```
data <-  raw %>%
  dplyr::select(Month,
                DayOfWeek,
                DepTime,
                ArrTime,
                ArrDelay,
                TailNum,
                Airline = UniqueCarrier, #Renaming the variable
                Time = ActualElapsedTime, #Renaming the variable
                Origin:Cancelled) #Selecting a number of columns.
names(data)
```

```
##  [1] "Month"     "DayOfWeek" "DepTime"   "ArrTime"   "ArrDelay"
##  [6] "TailNum"   "Airline"   "Time"      "Origin"    "Dest"
## [11] "Distance"  "TaxiIn"    "TaxiOut"   "Cancelled"
```

Suppose, we didn't really want to select the `Cancelled` variable. We can use `select()` to drop variables.

```
data <- data %>%
  dplyr::select(-Cancelled)
```

## Introducting `filter()`

There are a number of key operations when manipulating observations (rows).

- `x < y`
- `x <= y`
- `x == y`
- `x != y`
- `x >= y`
- `x > y`
- `x %in% c(a,b,c)` is `TRUE` if `x` is in the vector `c(a, b, c)`.

Suppose, we wanted to filter all the flights that have their destination in the greater Los Angeles area, specifically Los Angeles (LAX), Ontario (ONT), and John Wayne (SNA) airports. Note that based on the `hflights` dataset, there are no flights from the Houston area to Bob Hope (BUR) or Long Beach (LGB) airports.

```
airports <- c("LAX", "ONT", "SNA")

la_flights <- data %>%
  filter(Dest %in% airports)
```

**Caution**: The following command does not return the flights to LAX or ONT!

```
head(la_flights)
```

```
##   Month DayOfWeek DepTime ArrTime ArrDelay TailNum Airline Time Origin
## 1     1         1    1916    2103        2  N76522      CO  227    IAH
## 2     1         1     747     936        5  N67134      CO  229    IAH
## 3     1         1    1433    1629       14  N73283      CO  236    IAH
## 4     1         1    1750    1921        6  N34282      CO  211    IAH
## 5     1         1     917    1120       15  N76515      CO  243    IAH
## 6     1         1    1550    1736        8  N76502      CO  226    IAH
```

```
##   Dest Distance TaxiIn TaxiOut
## 1  LAX     1379      8      20
## 2  LAX     1379     11      17
## 3  LAX     1379     10      27
## 4  ONT     1334      5      17
## 5  SNA     1347      6      35
## 6  LAX     1379     13      15
```

```
la_flights_alt <- data %>%
  filter(Dest == c("LAX", "ONT"))
head(la_flights_alt)
```

```
##   Month DayOfWeek DepTime ArrTime ArrDelay TailNum Airline Time Origin
## 1     1         1    1916    2103        2  N76522      CO  227    IAH
## 2     1         1    1433    1629       14  N73283      CO  236    IAH
## 3     1         1    2107    2247        7  N73270      CO  220    IAH
## 4     1         1     920    1116        5  N77867      CO  236    IAH
## 5     1         1    1325    1538       32  N26210      CO  253    IAH
## 6     1         1    1749    1938        6  N73860      CO  229    IAH
##   Dest Distance TaxiIn TaxiOut
## 1  LAX     1379      8      20
## 2  LAX     1379     10      27
## 3  LAX     1379      7      12
## 4  LAX     1379      8      33
## 5  LAX     1379     11      30
## 6  LAX     1379     15      14
```

Why? We are basically returning all values for which the following is `TRUE` (using the correct output of the `la_flights` data frame:

`Dest[1] == LAX`

`Dest[2] == ONT`

`Dest[3] == LAX`

`Dest[4] == ONT ...`

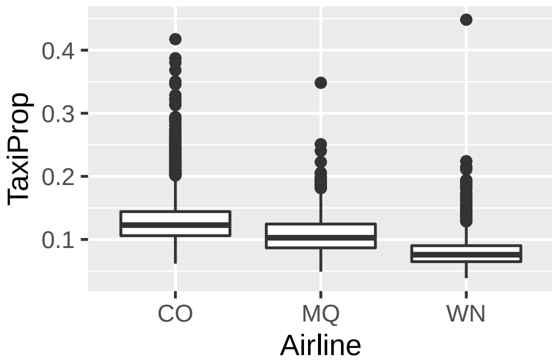
## Introducting `mutate()`

Currently, we have two taxi time variables in our data set: `TaxiIn` and `TaxiOut`. I care about total taxi time, and want to add the two together. Also, people hate sitting in planes while it is not in the air. To see how much time is spent taxiing versus flying, we create a variable which measures the proportion of taxi time of total time of flight.

```
la_flights <- data %>%
  filter(Dest %in% airports) %>%
  mutate(TaxiTotal = TaxiIn + TaxiOut,
         TaxiProp = TaxiTotal/Time)
```

We can the graph the average proportion of taxi time per airline.

```
library(ggplot2)
ggplot(la_flights,
       aes(x = Airline,
           y = TaxiProp)) +
  geom_boxplot()
```

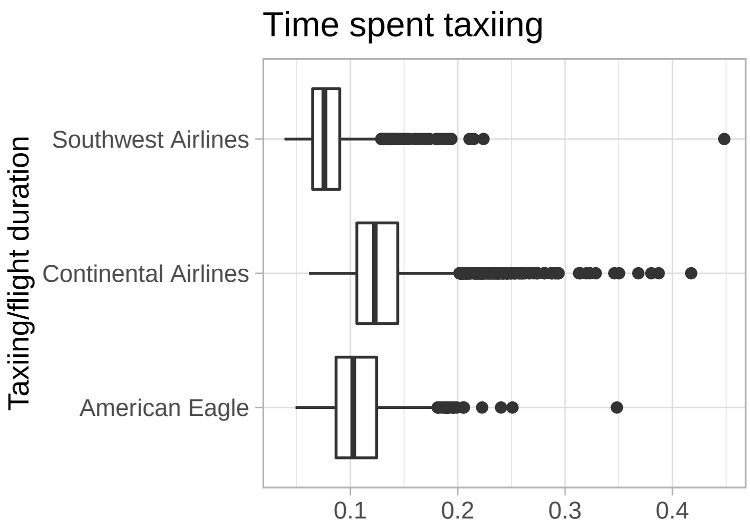



There is only three airlines flying to LA out of Houston. Lets create a new variable with the airline name using the `case_when()` function to make the graph more informative.

```
table(la_flights$Airline)
```

```
##
##   CO   MQ   WN
## 6471  810 1396
```

```
la_flights <- data %>%
  filter(Dest %in% airports) %>%
  mutate(TaxiTotal = TaxiIn + TaxiOut,
         TaxiProp = TaxiTotal/Time,
         AirlineName = case_when(
           Airline == "CO" ~ "Continental Airlines",
           Airline == "MQ" ~ "American Eagle",
           Airline == "WN" ~ "Southwest Airlines"
         ))

ggplot(la_flights,
       aes(x = AirlineName,
           y = TaxiProp)) +
  geom_boxplot() +
  coord_flip() +
  labs(title = "Time spent taxiing",
       x = "Taxiing/flight duration",
       y = "") +
  theme_light()
```




### Introducting `summarise()` and `arrange()`

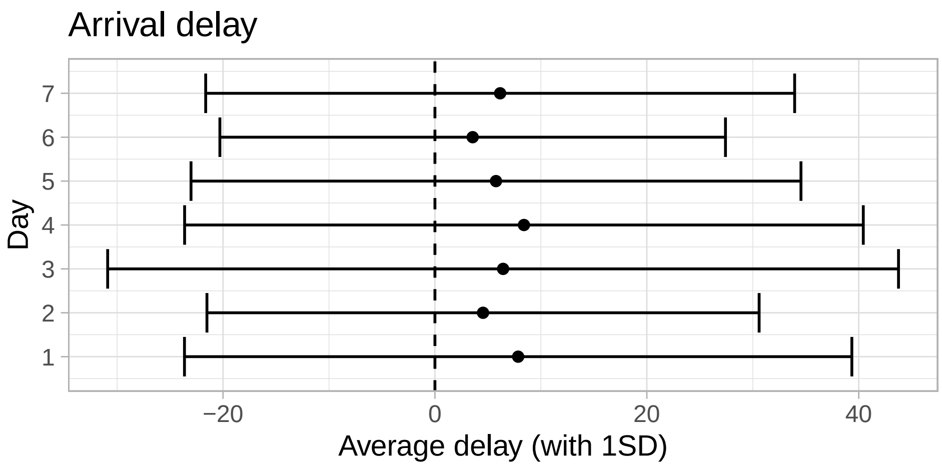
One of the most powerful `dplyr` features is the `summarise()` function, especially in combination with `group_by()`.

First, in a simple example, lets compute the average delay from Houston to Los Angeles by each day of the week. Note that the arrival delay variable is given in minutes. Also, I want to know what standard deviation of the delay is for each day of the weak. Note, that because there are missing values, we need to tell `R` what to do with them.

```
la_flights_delay <- la_flights %>%
  group_by(DayOfWeek) %>%
  summarise(av_delay = mean(ArrDelay, na.rm = T),
            sd_delay = sd(ArrDelay, na.rm = T))
```

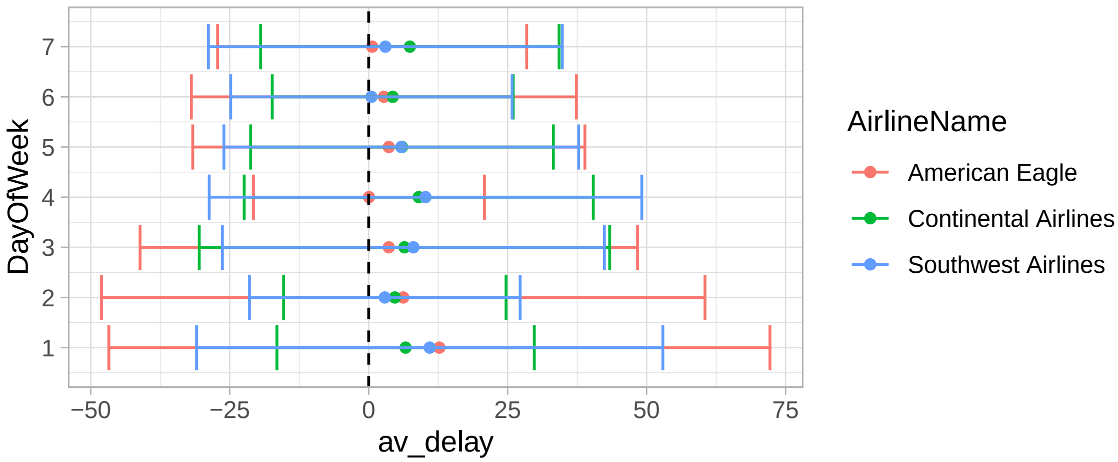
We can use error bars to show the standard deviation of the delay time for each day of the weak. I add a line to denote no delay using the `geom_hline()` geometric object.

```
ggplot(la_flights_delay,
       aes(x = DayOfWeek,
           y = av_delay,
           ymin = av_delay - sd_delay,
           ymax = av_delay + sd_delay)) +
  geom_point() +
  geom_errorbar() +
  geom_hline(yintercept = 0,
             linetype = "dashed") +

  # Making the graph prettier
  scale_x_continuous(breaks = seq(1,7)) +
  theme_light() +
  labs(y = "Average delay (with 1SD)",
       x = "Day",
       title = "Arrival delay") +
  coord_flip()
```



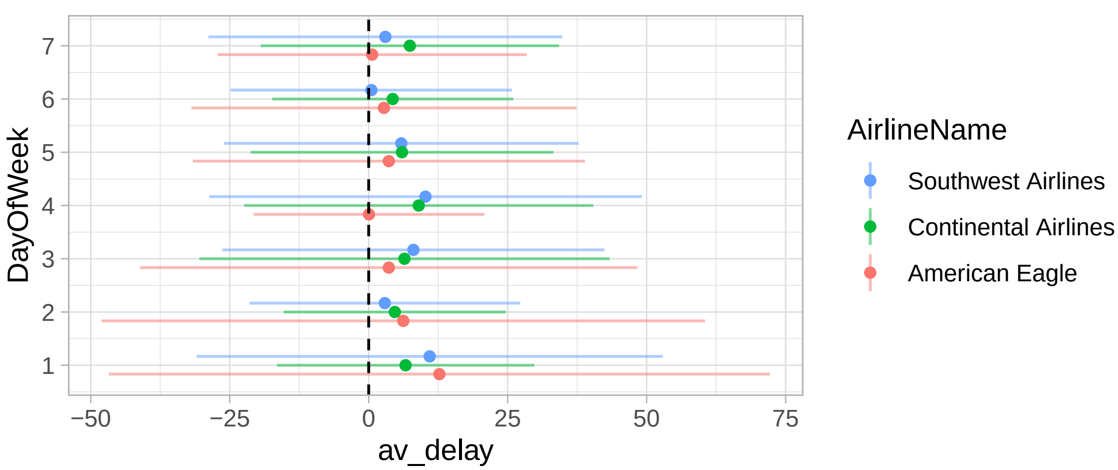

Suppose, I wanted to know whether some airlines have on average shorter arrival delays than others. We can add the airline to the `group_by()` function to compute the mean and standard deviation of arrival delay per day and airline.

```
la_flights_delay_airline <- la_flights %>%
  group_by(DayOfWeek, AirlineName) %>%
  summarise(av_delay = mean(ArrDelay, na.rm = T),
            sd_delay = sd(ArrDelay, na.rm = T))

# Plotting it
ggplot(la_flights_delay_airline,
       aes(x = DayOfWeek,
           y = av_delay,
           ymin = av_delay - sd_delay,
           ymax = av_delay + sd_delay,
           color = AirlineName)) +
  geom_point() +
  geom_errorbar() +
  geom_hline(yintercept = 0,
             linetype = "dashed") +

  # Making graph prettier
  theme_light() +
  coord_flip() +
  scale_x_continuous(breaks = seq(1,7))
```




To de-clutter the graph, below, I use the `geom_linerange()` geometric object rather than `geom_errorbar()`. I can use the `position = dodge` command within the `geom_point()` and `geom_linerange()` geometric object to display the values for each airline next to each other, instead on top of each other. Note that I could have used `position = dodge` with `geom_errorbar()` as well; the functionality is essentially the same.

```
ggplot(la_flights_delay_airline,
       aes(x = DayOfWeek,
           y = av_delay,
           ymin = av_delay - sd_delay,
           ymax = av_delay + sd_delay,
           color = AirlineName)) +
  geom_point(position = position_dodge(width = 0.5)) +
  geom_linerange(position = position_dodge(width = 0.5),
                 alpha = 0.5) +
  geom_hline(yintercept = 0,
             linetype = "dashed") +

  # Making graph prettier
  theme_light() +
  coord_flip() +
  scale_x_continuous(breaks = seq(1,7)) +

  # Matching order of legend and graph
  guides(color = guide_legend(reverse = T))
```



## Joins

`dplyr` has powerful tools to merge data frames together. Because we want to focus on data visualization here, I will not go over all possible joints in depth. Please see the Data Wrangling Cheat Sheet and the dplyr documentation for more details.

Suppose, we have two data frames: `x` and `y`. The basic syntax for data merging with `dplyr` is the following:

`output <- join(A, B, by = "variable")`

We will focus on the following three join functions:

- `left_join()`: Join only those rows from `y` that appear in `x`, retaining all data in `x`. Here, `x` is the "master."
- `right_join()`: Join only those rows from `x` that appear in `y`, retaining all data in `y`. Here, `y` is the "master."
- `full_join()`: Join data from `x` and `y` upon retaining all rows and values. This is the maximum join possible. Neither `x` nor `y` is the "master."

For demonstration purposes, lets create a new data frame that contains the name of the city for each of the Greater Los Angeles Area airports.

```
loc_airport <- data.frame(code = c("LAX", "ONT", "SNA", "BUR"),
                          location = c("Los Angeles", "Ontario", "Santa Ana", "Burbank"))
loc_airport
```

```
##   code    location
## 1  LAX Los Angeles
## 2  ONT     Ontario
## 3  SNA   Santa Ana
## 4  BUR     Burbank
```

First, we treat the `la_flights` data frame as the master and join it with the data frame containing the airport locations using `left_join()`. If the variable names in both data frames were the same, `dplyr` would automatically join the correct columns. Here, we manually match the column names.

```
la_flights_new <- left_join(la_flights, loc_airport,
                            by = c("Dest" = "code"))
```

```
## Warning: Column `Dest`/`code` joining character vector and factor, coercing
## into character vector
```

```
table(la_flights_new$Dest)
```

```
##
##  LAX  ONT  SNA
## 6064  952 1661
```

Second, lets create a similar result using `right_join()`. Again, `la_flights` is the master data frame.

```
la_flights_new2 <- right_join(loc_airport, la_flights,
                              by = c("code" = "Dest"))
```

```
## Warning: Column `code`/`Dest` joining factor and character vector, coercing
## into character vector
```

```
table(la_flights_new2$code)
```

```
##
##  LAX  ONT  SNA
## 6064  952 1661
```

Finally, for demonstration, we create a third data frame using `full_join()`. Because all observations are retained, this join creates one observation with empty values for the Burbank value in `loc_airport`. For most applications, this would be an undesirable outcome. However, below, we use the fact that all possible values are retained to set up the data for visualization.

```
la_flights_new3 <- full_join(la_flights, loc_airport,
                             by = c("Dest" = "code"))
```

```
## Warning: Column `Dest`/`code` joining character vector and factor, coercing
## into character vector
```

```
table(la_flights_new3$Dest)
```

```
##
##  BUR  LAX  ONT  SNA
##    1 6064  952 1661
```

```
la_flights_new3[la_flights_new3$Dest == "BUR",]
```

```
##      Month DayOfWeek DepTime ArrTime ArrDelay TailNum Airline Time Origin
## 8678    NA        NA      NA      NA       NA    <NA>    <NA>   NA   <NA>
##      Dest Distance TaxiIn TaxiOut TaxiTotal TaxiProp AirlineName location
## 8678  BUR       NA     NA      NA        NA       NA        <NA>  Burbank
```

## Heatmaps

For this example, we will go back to our original `data` tibble that contains the complete set of flight data for the Houston airports in 2011. Suppose we wanted to know, what are the busiest times at each of the two Houston airports, George Bush Intercontinental/Houston Airport (IAH) and William P. Hobby Airport (HOU). We create a new summary data frame that counts the number of departures per hour and day for each of the airports. We display these data using heatmaps.

To do so, we need to create a new variable that codes the hour of departure, using information from the `DepTime` variable. There are more advanced workflows available using the `stringr` and/or `lubridate` packages (both are part of the `tidyverse`). However, because we want to focus on data visualization, I simply divide the departure time by 100 and then use the `floor()` function to extract the hour of departure.

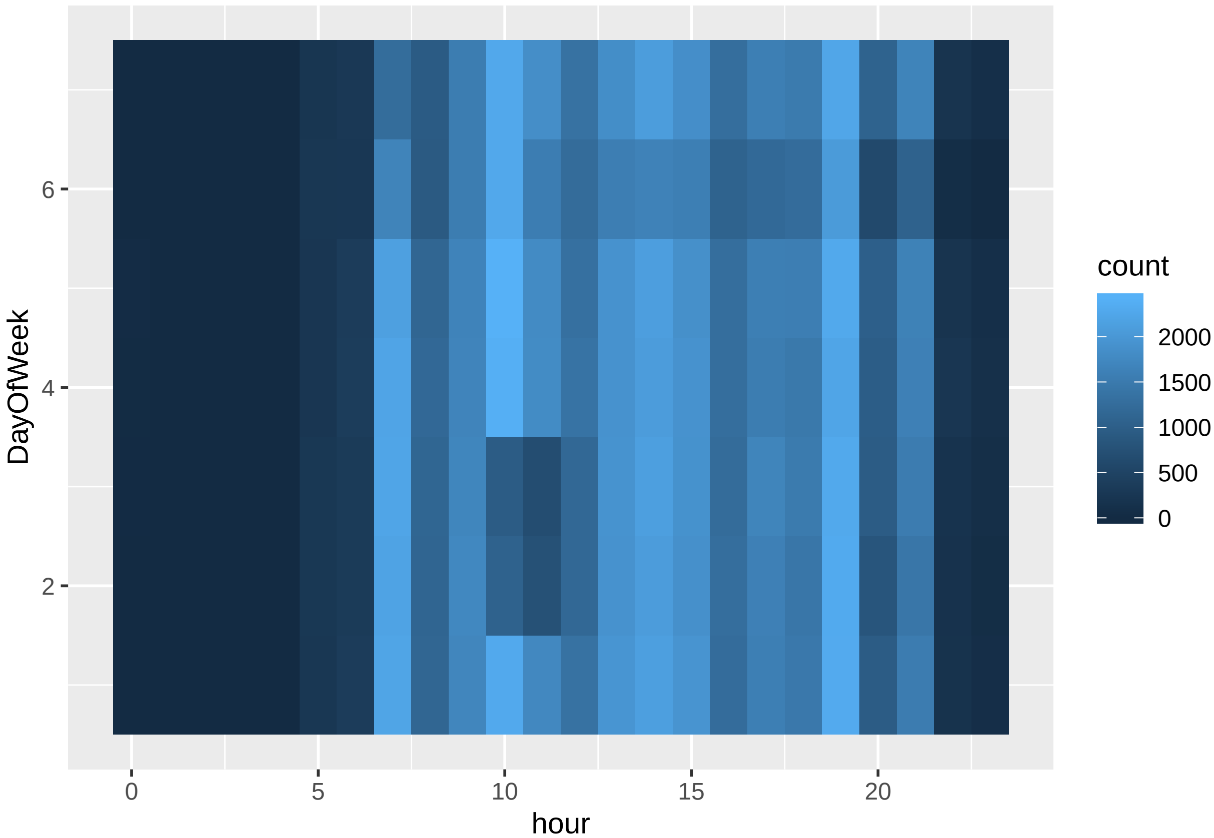
For this data frame, we also have to impute missing data. The data frame does not include observations for hours during which there are no flights, so we can assume that that these observations are not actually missing, but that there are no flights during these time slots.

Therefore, we create a data frame with all possible combinations of the variable values for day of the week and hour using `expand.grid()`, and use the `full_join()` function to create a new data frame (**not shown in class**). Similar to the application above, this procedure will result in missing values. We again use the `replace` function to re-code these missing values to zero.

```
# loading data frame
departures <- readr::read_csv("data/hflights_impute.csv")
```

```
## Parsed with column specification:
## cols(
##   Origin = col_character(),
##   DayOfWeek = col_double(),
##   hour = col_double(),
##   count = col_double()
## )
```

```r
# visualizing data on flights from HOU
ggplot(departures,
       aes(x = hour,
           y = DayOfWeek,
           fill = count)) +
  geom_tile()
```




To improve on this graph, we can add some of the other elements offered by the `ggplot2` package.

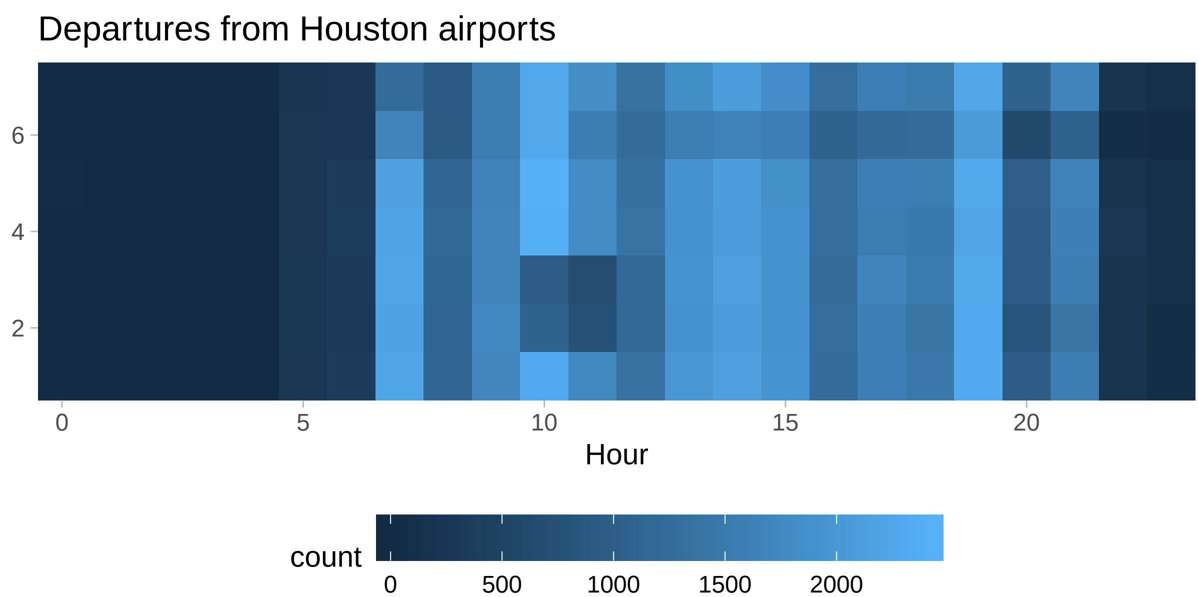
```r
# loading data frame
departures <- readr::read_csv("data/hflights_impute.csv")
```

```
## Parsed with column specification:
## cols(
##   Origin = col_character(),
##   DayOfWeek = col_double(),
##   hour = col_double(),
##   count = col_double()
## )
```

```r
# visualizing data on flights from HOU
ggplot(departures,
       aes(x = hour,
           y = DayOfWeek,
           fill = count)) +
  geom_tile() +
    coord_flip() +
  labs(x = "Hour",
```

```
      y = "",
      title = "Departures from Houston airports") +

  # Changing appearance of the plot
  theme_light() +
  theme(panel.grid = element_blank(),
        legend.position = "bottom",
        legend.key.width = unit(1.5, "cm"),
        panel.border=element_blank()) +

  # Making the space equal by fixing aspect ratio
  # Reducing space
  coord_fixed(expand = c(0,0))
```

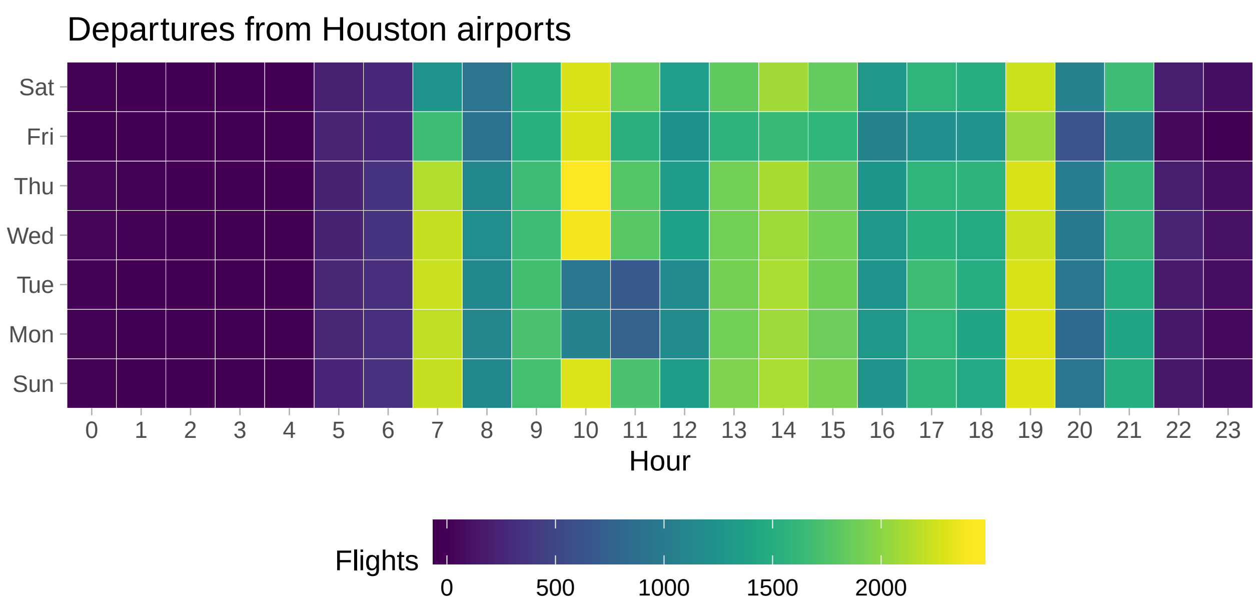
```
## Coordinate system already present. Adding new coordinate system, which will replace the existing one

## Warning in if (expand) expand_default(scale) else c(0, 0): the condition
## has length > 1 and only the first element will be used

## Warning in if (expand) expand_default(scale) else c(0, 0): the condition
## has length > 1 and only the first element will be used
```




Changing the color scheme would also improve the appearance of this plot. Below, we use color scales from the `viridis` package.

```
# install.packages("viridis")
library(viridis)


# visualizing it
ggplot(departures,
       aes(x = hour,
           y = DayOfWeek,
           fill = count)) +

  geom_tile(color = "white") +
```

```r
  scale_fill_viridis(name = "Flights") +
  scale_x_continuous(breaks = seq(0,23)) +
  scale_y_continuous(breaks = seq(1,7),
                     labels = c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")) +

  coord_flip() +
  labs(x = "Hour",
       y = "",
       title = "Departures from Houston airports") +

  # Changing appearance of the plot
  theme_light() +
  theme(panel.grid = element_blank(),
        legend.position = "bottom",
        legend.key.width = unit(1.5, "cm"),
        panel.border=element_blank()) +

  # Making the space equal by fixing aspect ratio
  # Reducing space
  coord_fixed(expand = c(0,0))
```




```r
table(departures$DayOfWeek)
```

```
##
##  1  2  3  4  5  6  7
## 48 48 48 48 48 48 48
```

## Primer on `tidyr`

Often, data does not come in the format that we need for data merging, data visualization, statistical analysis, or vectorized programming. In general, we want data in the following format:

1. Each variable forms a column.

2. Each observation forms a row[1].
3. For panel data, the unit (e.g. country) and time (e.g. year) identifier form columns.

The `tidyr` package offers two main functions for data reshaping:

- `pivot_longer()`: Shaping data from wide to long.
- `pivot_wider()`: Shaping data from long to wide.

### Wide versus long data

For **wide** data formats, each unit's responses are in a single row. For example:

| Country | Area | Pop1990 | Pop1991 |
|---|---|---|---|
| A | 300 | 56 | 58 |
| B | 150 | 40 | 45 |

For **long** data formats, each row denotes the observation of a unit at a given point in time. For example:

| Country | Year | Area | Pop |
|---|---|---|---|
| A | 1990 | 300 | 56 |
| A | 1991 | 300 | 58 |
| B | 1990 | 150 | 40 |
| B | 1991 | 150 | 45 |

### `pivot_longer()`

We use the `pivot_longer()` function to reshape data from wide to long. In general, the syntax of the data is as follows:

`new_df <- pivot_longer(old_df, columns to transform, names_to = "name", values_to = "value")`[2]

Below, we use the `murder_2016_prelim` data set from the `fivethirtyeight` package. The data contains number of murders in 79 U.S. cities. The dataset contains a column `murder_2015` and a column `murder_2016`. For tidy data, we want one observation per row and one variable per column. The data is untidy because the two columns confuse the variables `murder` and `year`.

Below, we use `pivot_longer()` to tidy the data. For illustration we drop the variable change to show how to re-create it.

```
# install.packages("tidyr")
library(tidyr)

# install.packages("fivethirtyeight")
library(fivethirtyeight)
murder <- fivethirtyeight::murder_2016_prelim
head(murder)
```

[1] Hadley Wickham (2014, "Tidy Data" in *Journal of Statistical Analysis*) adds another condition, namely that "Each type of observational unit forms a table." We will not go into this here, but I can highly recommend you read Wickham's piece if you want to dive deeper into tidying data.

[2] If we use the function in a pipe, we do not need to specify the `old_df` parameter. `tidyr` automatically knows to use the lastest version of the data frame in the pipe.

```
## # A tibble: 6 x 7
##   city    state   murders_2015 murders_2016 change source          as_of
##   <chr>   <chr>          <int>        <int>  <int> <chr>           <date>
## 1 Chica~ Illin~           378          536    158 https://portal~ 2016-10-02
## 2 Orlan~ Flori~            19           73     54 OPD             2016-09-22
## 3 Memph~ Tenne~           114          158     44 MPD             2016-09-11
## 4 Phoen~ Arizo~            72          111     39 PPD             2016-08-31
## 5 Las V~ Nevada            90          125     35 http://www.lvm~ 2016-09-28
## 6 San A~ Texas             78          111     33 SAPD            2016-09-26
```

```
# using gather to re-shape
murder_tidy <- murder %>%
  dplyr::select(-change) %>%
  pivot_longer(murders_2015:murders_2016, names_to = "murders_year", values_to = "n_murder")

head(murder_tidy)
```

```
## # A tibble: 6 x 6
##   city    state   source                       as_of      murders_year n_murder
##   <chr>   <chr>   <chr>                        <date>     <chr>           <int>
## 1 Chica~ Illin~ https://portal.chicagopol~ 2016-10-02 murders_2015      378
## 2 Chica~ Illin~ https://portal.chicagopol~ 2016-10-02 murders_2016      536
## 3 Orlan~ Flori~ OPD                        2016-09-22 murders_2015       19
## 4 Orlan~ Flori~ OPD                        2016-09-22 murders_2016       73
## 5 Memph~ Tenne~ MPD                        2016-09-11 murders_2015      114
## 6 Memph~ Tenne~ MPD                        2016-09-11 murders_2016      158
```

We can use the `separate()` function from the `tidyr` package to turn the column `murders_year` into two separate columns and then drop the `murders` column.

```
murder_tidier <- murder %>%
  dplyr::select(-change) %>%
  pivot_longer(murders_2015:murders_2016,names_to="murders_year",values_to="n_murder") %>%
  separate(murders_year,c("murders", "year"), sep = "_") %>%
  dplyr::select(-murders)
head(murder_tidier)
```

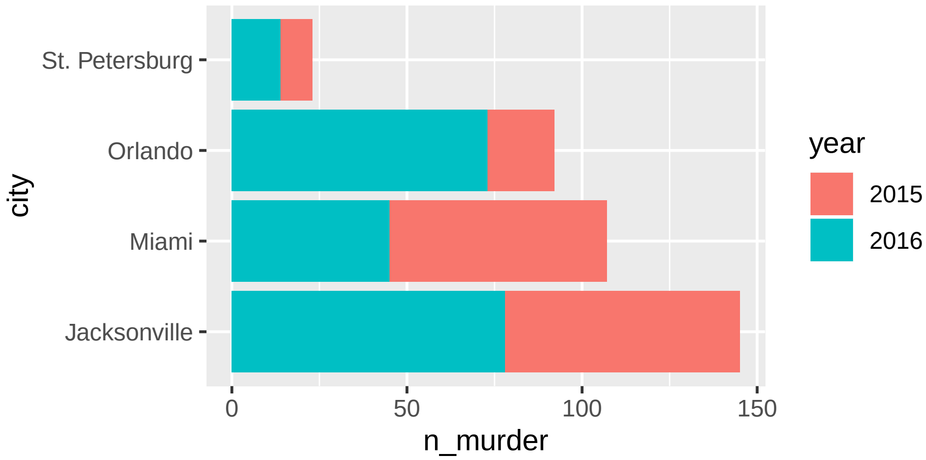
```
## # A tibble: 6 x 6
##   city    state    source                              as_of      year  n_murder
##   <chr>   <chr>    <chr>                               <date>     <chr>    <int>
## 1 Chica~ Illino~ https://portal.chicagopolice.or~ 2016-10-02 2015       378
## 2 Chica~ Illino~ https://portal.chicagopolice.or~ 2016-10-02 2016       536
## 3 Orlan~ Florida OPD                               2016-09-22 2015        19
## 4 Orlan~ Florida OPD                               2016-09-22 2016        73
## 5 Memph~ Tennes~ MPD                               2016-09-11 2015       114
## 6 Memph~ Tennes~ MPD                               2016-09-11 2016       158
```

### Dataviz: Barplots

Suppose we wanted to know what was the city in Florida with the overall highest number of murders. Now that the data is tidy, we can create a grouped bar plot, showing the 2014 and 2015 values with different fill colors.

```
ggplot(subset(murder_tidier, state == "Florida"),
       aes(x = city,
           y = n_murder,
```

```
         fill = year)) +
  geom_bar(stat = "identity") +
  coord_flip()
```

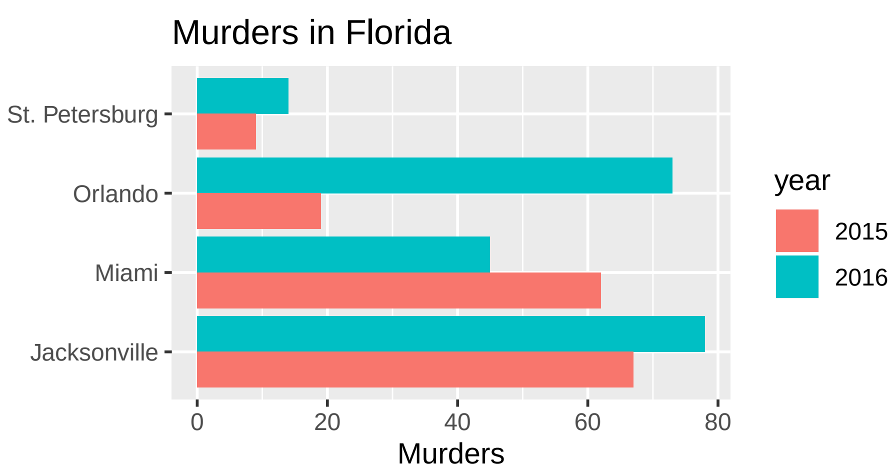



**Question** Is the plot above showing what we want? How would you improve it?

*The plot above is confusing! It is adding together the murders for 2014 and 2015, and differences are hard to gauge*

We can use `position = "dodge" within the`geom_bar()‘ statement place the bars for 2014 and 2015 next to each other and group them by city.

```
ggplot(subset(murder_tidier, state == "Florida"),
       aes(x = city,
           y = n_murder,
           fill = year)) +
  geom_bar(stat = "identity", position = "dodge") +
  coord_flip() +
  labs(x = "",
       y = "Murders",
       title = "Murders in Florida")
```



### Creating the first difference

Below, we create a variable that captures the first difference of murders between 2014 and 2015 for each city using the `lag()` function. in combination with `group_by()`. Make sure your data is ordered in the right way using `arrange()` before taking the lagged value $t-1$ and subtracting it from the value at $t$.

Note, that we need to change the `year` variable from character to numeric to make the code work.

```
str(murder_tidier)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    158 obs. of  6 variables:
##  $ city    : chr  "Chicago" "Chicago" "Orlando" "Orlando" ...
##  $ state   : chr  "Illinois" "Illinois" "Florida" "Florida" ...
##  $ source  : chr  "https://portal.chicagopolice.org/portal/page/portal/ClearPath/News/Crime%20Statis
##  $ as_of   : Date, format: "2016-10-02" "2016-10-02" ...
##  $ year    : chr  "2015" "2016" "2015" "2016" ...
##  $ n_murder: int  378 536 19 73 114 158 72 111 90 125 ...
```

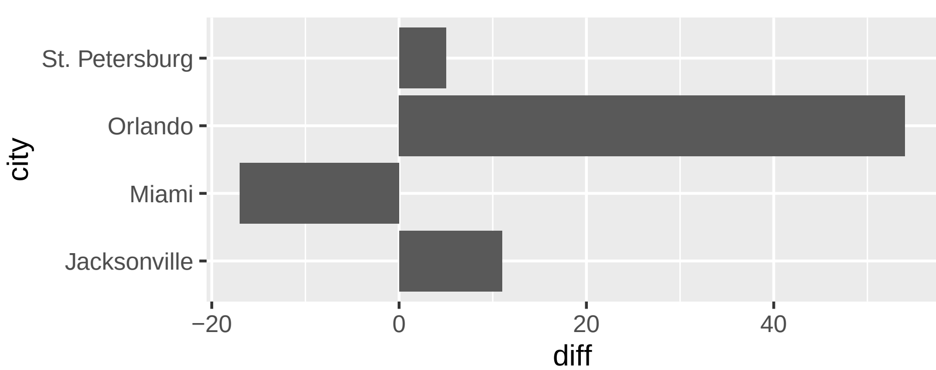
```
murder_change <- murder_tidier %>%
  mutate(year = as.numeric(year)) %>%
  group_by(city) %>%
  arrange(year) %>%

  # Creating variable for first difference
  mutate(diff = n_murder - lag(n_murder),

         # Creating indicator for negative differences
         diff_neg = ifelse(diff < 0, 1, 0))
```

We can visualize this difference for cities in Florida using a bar plot.

```
summary(murder_change$diff)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
## -21.000  -3.000   2.000   5.911   9.000 158.000      79
```

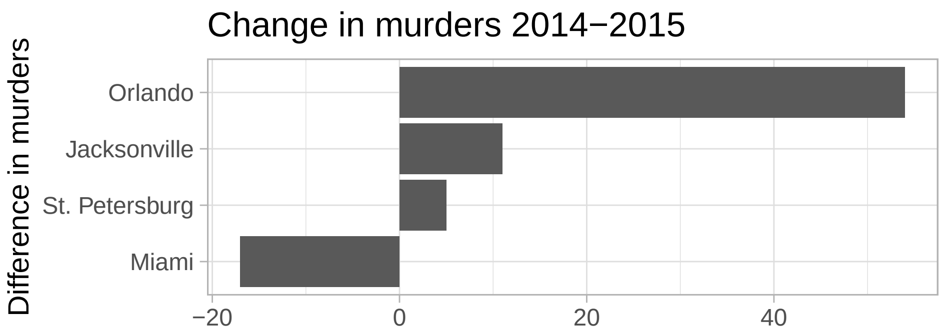
```
ggplot(subset(murder_change, !is.na(diff) & state == "Florida"),
       aes(x = city,
           y = diff)) +
  geom_bar(stat='identity') +
  coord_flip()
```




The plot can be improved by ordering the bars based on the difference in murder rate.

```
ggplot(subset(murder_change, !is.na(diff) & state == "Florida"),
       aes(x = reorder(city, diff),
```

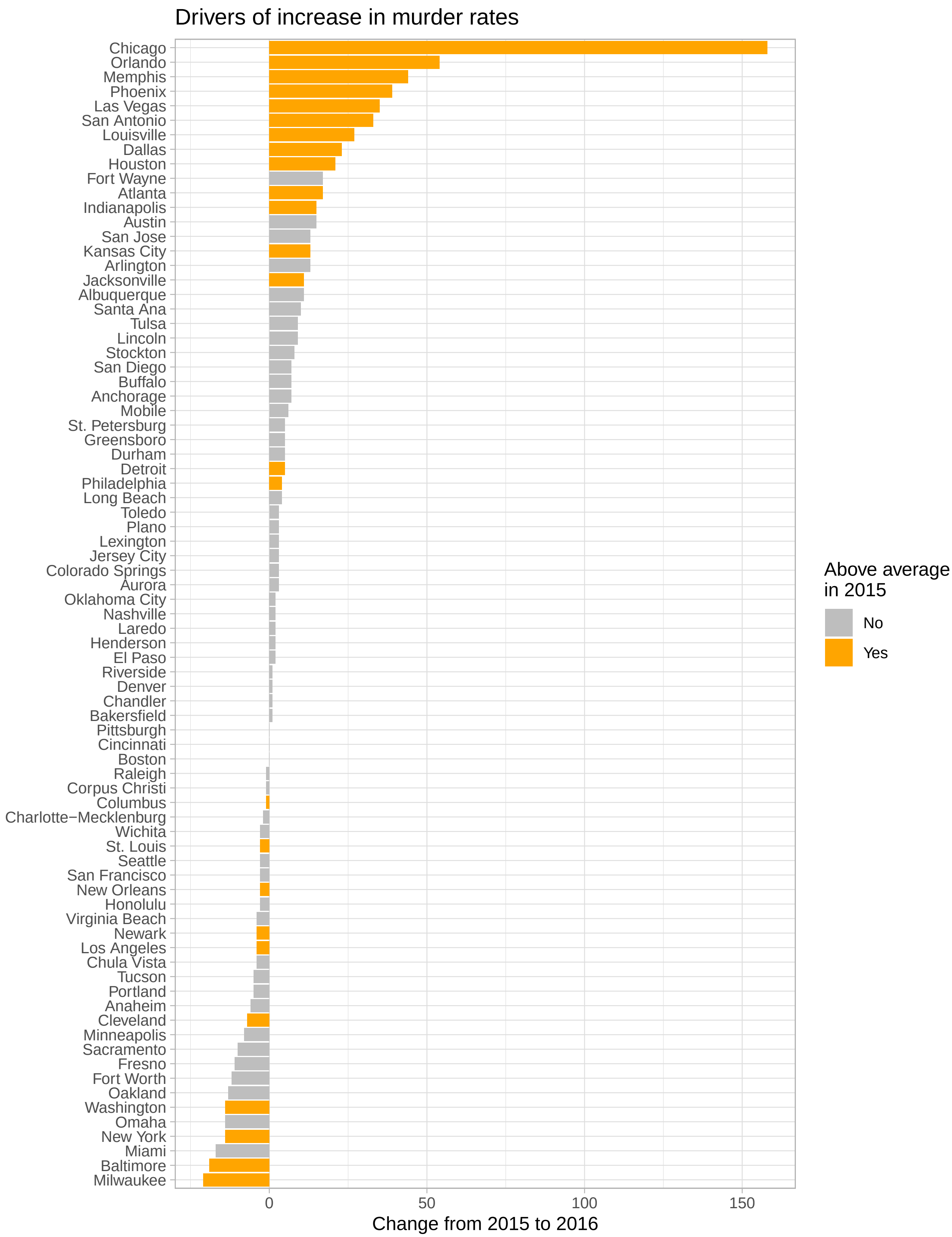
```
           y = diff)) +
  geom_bar(stat='identity') +
  coord_flip() +
  theme_light() +
  labs(x = "Difference in murders",
       y = "",
       title = "Change in murders 2014-2015")
```




Suppose, I wanted to know whether murders appear to increase in cities that already had a large number of murders (i.e. above average in 2014), or whether it is "low murder rate" cities experiencing more homicides. We could plot the change in the number of murders for all cities in the data frame. This will create a very large bar plot that is hard to read without appropriately grouping the data.

Below, create a new data frame from `murder_change`, called `murder_change_av`, that adds a dummy variable coded 1 for observations that had above average murder rates in 2014 (taking into account only the year 2014), and 0 otherwise. Note that in the code below, we are not taking the population size of the cities into account, and plot just the absolute values.

```
murder_change_av <- murder_change %>%
  ungroup() %>%
  mutate(aboveav2015 = ifelse(n_murder >= mean(n_murder[year == 2015]), 1, 0)) %>%
  ungroup()

ggplot(subset(murder_change_av, !is.na(diff)),
       aes(x = reorder(city, diff),
           y = diff,
           fill = factor(aboveav2015))) +
  geom_bar(stat = 'identity') +
  coord_flip() +
  theme(axis.text.x = element_text(size = 1),
        legend.position = "none") +
  theme_light() +
  labs(title = "Drivers of increase in murder rates",
       y = "Change from 2015 to 2016",
       x = "") +
  scale_fill_manual(name = "Above average\nin 2015",
                    values = c("0" = "grey",
                               "1" = "orange"),
                    labels = c("No", "Yes"))
```



Drivers of increase in murder rates
Chicago
Orlando
Memphis
Phoenix
Las Vegas
San Antonio
Louisville
Dallas
Houston
Fort Wayne
Atlanta
Indianapolis
Austin
San Jose
Kansas City
Arlington
Jacksonville
Albuquerque
Santa Ana
Tulsa
Lincoln
Stockton
San Diego
Buffalo
Anchorage
Mobile
St. Petersburg
Greensboro
Durham
Detroit
Philadelphia
Long Beach
Toledo
Plano
Lexington
Jersey City
Colorado Springs
Aurora
Oklahoma City
Nashville
Laredo
Henderson
El Paso
Riverside
Denver
Chandler
Bakersfield
Pittsburgh
Cincinnati
Boston
Raleigh
Corpus Christi
Columbus
Charlotte−Mecklenburg
Wichita
St. Louis
Seattle
San Francisco
New Orleans
Honolulu
Virginia Beach
Newark
Los Angeles
Chula Vista
Tucson
Portland
Anaheim
Cleveland
Minneapolis
Sacramento
Fresno
Fort Worth
Oakland
Washington
Omaha
New York
Miami
Baltimore
Milwaukee
0
50
100
150
Change from 2015 to 2016
Above average in 2015
No
Yes

### pivot_wider()

Suppose we wanted to revert our operation (or generall shape data from a long to a wide format), we can use `tidyr`'s `pivot_wider()` function. The syntax is similar to `pivot_longer()`.

`new_df <- pivot_wider(old_df, names_from = key, values_from = value)`,

where `key` refers to the colum which contains the values that are to be converted to column names and `value` specifies the column that contains the values which is to be stored in the newly created columns.

Below, we semi-revert the creation of a single column containing the variable types and a single column containing our variable values.

```
murders_untidy <- murder_change %>%
  dplyr::select(-diff) %>%
  pivot_wider(names_from = year,
              values_from = n_murder)
head(murders_untidy)
```

```
## # A tibble: 6 x 7
## # Groups:   city [6]
##   city    state  source                   as_of      diff_neg `2015` `2016`
##   <chr>   <chr>  <chr>                    <date>        <dbl>  <int>  <int>
## 1 Chicago Illin~ https://portal.chicagop~ 2016-10-02       NA    378     NA
## 2 Orlando Flori~ OPD                      2016-09-22       NA     19     NA
## 3 Memphis Tenne~ MPD                      2016-09-11       NA    114     NA
## 4 Phoenix Arizo~ PPD                      2016-08-31       NA     72     NA
## 5 Las Ve~ Nevada http://www.lvmpd.com/Se~ 2016-09-28       NA     90     NA
## 6 San An~ Texas  SAPD                     2016-09-26       NA     78     NA
```

# Visualizing regression results

In data collection for the 2000 American National Election Studies survey, interviewers assigned an ordinal rating to each respondent's "general level of information" about politics and public affairs. We will be working with an adapted version of the `politicalInformation` dataset from the `pscl` ("Political Science Computational Laboratory") package to visualize regression results.

```
dat <- readr::read_csv("data/polinfo.csv")
```

```
## Parsed with column specification:
## cols(
##   y = col_character(),
##   collegeDegree = col_double(),
##   female = col_double(),
##   age = col_double(),
##   homeOwn = col_character(),
##   govt = col_character(),
##   length = col_double(),
##   id = col_double(),
##   aboveav = col_double()
## )
```

Let's run an OLS regression to identify the determinants of interviewer ratings on "general level of information" about politics and public affairs (**not shown in class; results stored in the RStudio Cloud environment**)

```r
# mod1 <- lm(y ~ collegeDegree + female + length + age,
           # data = dat)
# summary(mod1)
```

## Coefficient plot

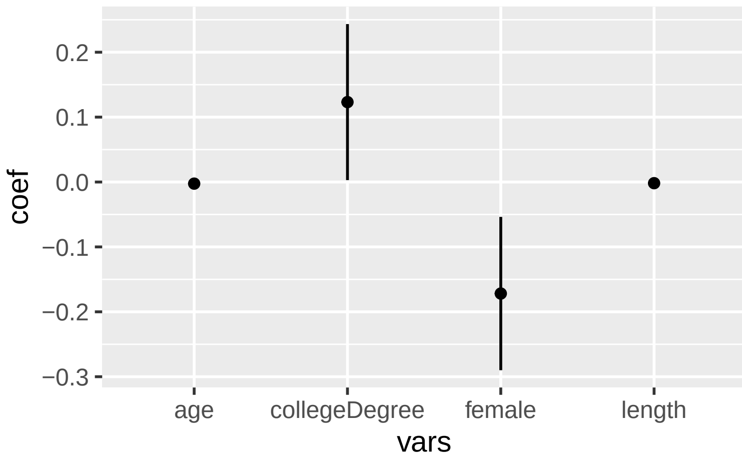
There are a number of packages that offer off-the-shelf solutions to plotting coefficient plots for regression outcomes. In this workshop, we will create a coefficient plot manually. This will allow you to create coefficient plots for models that are not supported by existing packages.

Below, we extract properties of interest from the `mod1` object (**not shown in class; data frame stored in RStudio Cloud environment**).

```r
# str(summary(mod1)$coefficients)
# dimnames(summary(mod1)$coefficients)
#
# # Note that dimnames() returns a list object, not a vector
# df_mod1 <- data.frame(vars = dimnames(summary(mod1)$coefficients)[[1]],
#                       coef = summary(mod1)$coefficients[,1],
#                       se = summary(mod1)$coefficients[,2]) %>%
#
#   # Computing CIs
#   mutate(cilo_95 = coef - 1.96*se,
#          cihi_95 = coef + 1.96*se,
#          cilo_99 = coef - 2.56*se,
#          cihi_99 = coef + 2.56*se) %>%
#
#   # remove intercept for coef plot
#   filter(vars != "(Intercept)")
```

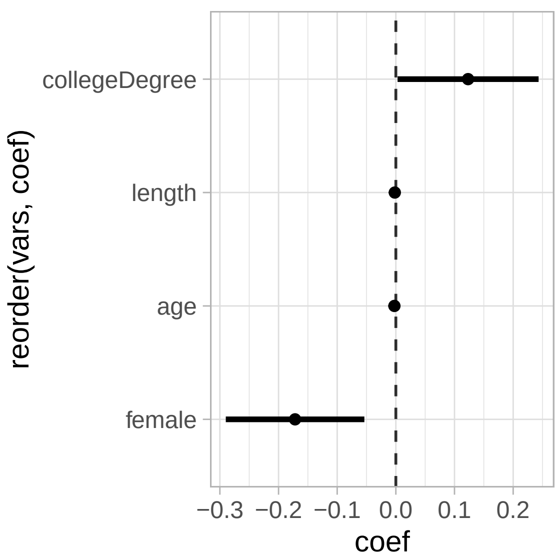
We can graph the coefficient plot using the `geom_point()` aesthetic for the coefficient and the `geom_linerange()` aesthetic for the 95% confidence intervals.

```r
# load data
df_mod1 <- readr::read_csv("data/polinfo_mod1results.csv")

# visualize it
ggplot(df_mod1,
       aes(x = vars,
           y = coef)) +
  geom_point() +
  geom_linerange(aes(ymin = cilo_95, ymax = cihi_95))
```




We can flip the axes and order the coefficients based on their size to clean up the plot. We also add a line at zero to illustrate which coefficients are statistically significantly different from zero. Note that I add the zero line before the `geom_point()` aesthetic so it is in the background.

```
ggplot(df_mod1,
       aes(x = reorder(vars, coef),
           y = coef)) +
  geom_hline(yintercept = 0, alpha = 0.8, linetype = "dashed") +
  geom_point() +
  geom_linerange(aes(ymin = cilo_95, ymax = cihi_95),
                 size = 1) +
  coord_flip() +
  theme_light()
```




## Plotting predicted probabilities

Suppose, we estimated a logit model of the probability of being classified as above average on political knowledge, incorporating a quadratic term for age. We can add these estimates to the data frame of regression results and plot them on the same coefficient plot to compare the results.

(**Not shown in class**) Below, I estimate a new model, `mod2` that includes the squared `age` variable. Note that I add an indicator for the model number `modnum` that we will later use to visually distinguish the results from both models.

```
dat <- dat %>%
  mutate(age2 = age^2)
mod2 <- glm(aboveav ~ collegeDegree + female + length + age + age2,
           data = dat,
           family = binomial(link = "logit"))
summary(mod2)
```

```
##
## Call:
## glm(formula = aboveav ~ collegeDegree + female + length + age +
##     age2, family = binomial(link = "logit"), data = dat)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1271  -0.9197  -0.5919   0.9878   2.2663
##
## Coefficients:
##                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)   -3.7832857  0.4694704  -8.059 7.72e-16 ***
## collegeDegree  1.5032597  0.1089735  13.795  < 2e-16 ***
## female        -0.7360677  0.1065410  -6.909 4.89e-12 ***
## length         0.0107356  0.0023225   4.622 3.79e-06 ***
## age            0.0984565  0.0184042   5.350 8.81e-08 ***
## age2          -0.0008445  0.0001761  -4.797 1.61e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2445.9  on 1789  degrees of freedom
## Residual deviance: 2099.9  on 1784  degrees of freedom
##   (10 observations deleted due to missingness)
## AIC: 2111.9
##
## Number of Fisher Scoring iterations: 4
```

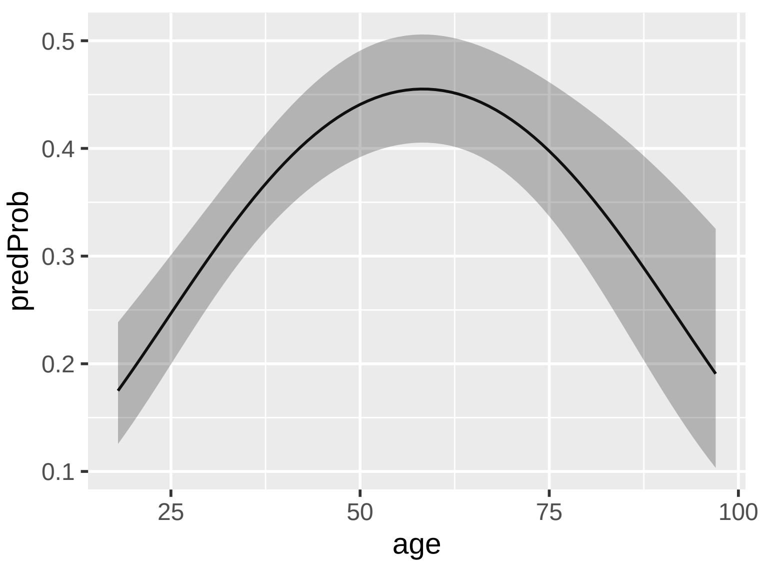
```
# Extracting the estimates
df_mod2 <- data.frame(vars = dimnames(summary(mod2)$coefficients)[[1]],
                      coef = summary(mod2)$coefficients[,1],
                      se = summary(mod2)$coefficients[,2]) %>%

  # Computing CIs
  mutate(cilo_95 = coef - 1.96*se,
         cihi_95 = coef + 1.96*se,
         cilo_99 = coef - 2.56*se,
         cihi_99 = coef + 2.56*se)
```

```
# Male, no college degree
scen_male <- expand.grid(collegeDegree = 0,
                         female = 0,
                         length = mean(dat$length, na.rm = T),
                         age = seq(min(dat$age, na.rm = T), max(dat$age, na.rm = T), 1)) %>%
```

```r
  mutate(age2 = age^2)

# Below, get estimates on link scale, transform to predicted probabilities
# See https://stats.idre.ucla.edu/r/dae/logit-regression/
df_male <- cbind(scen_male,
                 predict(mod2, newdata = scen_male, type = "link", se = TRUE)) %>%
  mutate(predProb = plogis(fit),
         cilo = plogis(fit - (1.96 * se.fit)),
         cihi = plogis(fit + (1.96 * se.fit)))

# write csv of results to visualize
# readr::write_csv(df_male,"data/polinfo_mod2results.csv")
```

Below, we use `geom_ribbon()` to graph the confidence interval around the age estimate. Note that we could use `geom_linerange()` instead.

```r
# load data
df_male <- readr::read_csv("data/polinfo_mod2results.csv")

# visualize it
ggplot(df_male,
       aes(x = age,
           y = predProb)) +
  geom_line() +
  geom_ribbon(aes(ymin = cilo, ymax = cihi),
              alpha = 0.3)
```

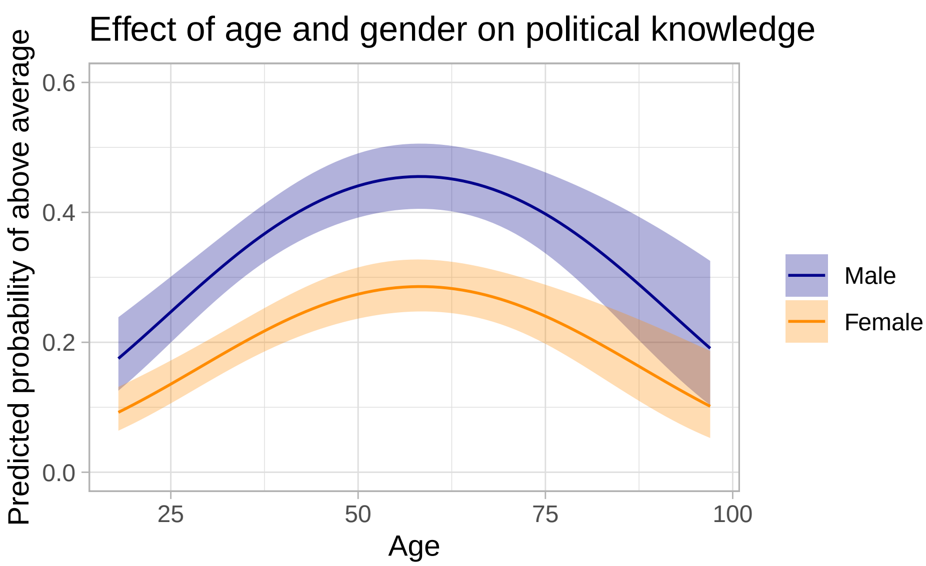



**Advanced exercise (not shown in class)** Suppose we wanted to compare the effect of age on recorded political knowledge for male and female respondents.

Please compute the predicted probability of being classified as having above average political knowledge for female respondents, combine the two data frames into one, and graph the results for both male and female respondents in the same plot. Try to re-create the graph below as closely as possible.

```r
scen_female <- expand.grid(collegeDegree = 0,
                           female = 1,
                           length = mean(dat$length, na.rm = T),
                           age = seq(min(dat$age, na.rm = T), max(dat$age, na.rm = T), 1)) %>%
```

```r
  mutate(age2 = age^2)

df_both <- cbind(scen_female,
                 predict(mod2, newdata = scen_female, type = "link", se = TRUE)) %>%
  mutate(predProb = plogis(fit),
         cilo = plogis(fit - (1.96 * se.fit)),
         cihi = plogis(fit + (1.96 * se.fit))) %>%
  bind_rows(df_male)

#plotting effect for male and female respondents
ggplot(df_both,
       aes(x = age,
           y = predProb,
           color = factor(female),
           fill = factor(female))) +
  geom_line() +
  geom_ribbon(aes(ymin = cilo, ymax = cihi),
              alpha = 0.3,
              color = NA) +

  # adjusting the appearance of the plot
  scale_color_manual(values = c("0" = "darkblue",
                                "1" = "darkorange"),
                     name = "",
                     labels = c("Male", "Female")) +
  scale_fill_manual(values = c("0" = "darkblue",
                               "1" = "darkorange"),
                     name = "",
                     labels = c("Male", "Female")) +
  theme_light() +
  labs(x = "Age",
       y = "Predicted probability of above average",
       title = "Effect of age and gender on political knowledge") +
  coord_cartesian(ylim = c(0,0.6))
```



## Sources

Wilkinson, L., 2012. The grammar of graphics. In *Handbook of Computational Statistics* (pp. 375-414). Springer, Berlin, Heidelberg.