Data Visualization (Healy, 2020) Notes

Austin Moss

2/9/2021

Data Visualization Notes

This is a starter RMarkdown template to accompany *Data Visualization* (Princeton University Press, 2019). You can use it to take notes, write your code, and produce a good-looking, reproducible document that records the work you have done. At the very top of the file is a section of *metadata*, or information about what the file is and what it does. The metadata is delimited by three dashes at the start and another three at the end. You should change the title, author, and date to the values that suit you. Keep the `output` line as it is for now, however. Each line in the metadata has a structure. First the *key* ("title", "author", etc), then a colon, and then the *value* associated with the key.

This is an RMarkdown File

Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. A *code chunk* is a specially delimited section of the file. You can add one by moving the cursor to a blank line choosing Code > Insert Chunk from the RStudio menu. When you do, an empty chunk will appear:

Code chunks are delimited by three backticks (found to the left of the 1 key on US and UK keyboards) at the start and end. The opening backticks also have a pair of braces and the letter `r`, to indicate what language the chunk is written in. You write your code inside the code chunks. Write your notes and other material around them, as here.

Before you Begin

To install the tidyverse, make sure you have an Internet connection. Then *manually* run the code in the chunk below. If you knit the document if will be skipped. We do this because you only need to install these packages once, not every time you run this file. Either knit the chunk using the little green "play" arrow to the right of the chunk area, or copy and paste the text into the console window.

```
## This code will not be evaluated automatically.
## (Notice the eval = FALSE declaration in the options section of the
## code chunk)

my_packages <- c("tidyverse", "broom", "coefplot", "cowplot",
                 "gapminder", "GGally", "ggrepel", "ggridges", "gridExtra",
                 "here", "interplot", "margins", "maps", "mapproj",
                 "mapdata", "MASS", "quantreg", "rlang", "scales",
                 "survey", "srvyr", "viridis", "viridisLite", "devtools")
```

```
install.packages(my_packages, repos = "http://cran.rstudio.com")
```

Set Up Your Project and Load Libraries

To begin we must load some libraries we will be using. If we do not load them, R will not be able to find the functions contained in these libraries. The tidyverse includes ggplot and other tools. We also load the socviz and gapminder libraries.

Notice that here, the braces at the start of the code chunk have some additional options set in them. There is the language, `r`, as before. This is required. Then there is the word `setup`, which is a label for your code chunk. Labels are useful to briefly say what the chunk does. Label names must be unique (no two chunks in the same document can have the same label) and cannot contain spaces. Then, after the comma, an option is set: `include=FALSE`. This tells R to run this code but not to include the output in the final document.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

The remainder of this document contains the chapter headings for the book, and an empty code chunk in each section to get you started. Try knitting this document now by clicking the "Knit" button in the RStudio toolbar, or choosing File > Knit Document from the RStudio menu.

Look at Data

Get Started

Everything in R is an *object* that has a *name*, including variables (like `x` or `y`), datasets (like `mydata`), and functions (like `mean()`). The code directly below creates an object named `mynumbers` that is a vector of numbers.

```
mynumbers <- c(1, 2, 3, 1, 3, 5, 25)
yournumbers <- c(5, 31, 71, 1, 3, 21, 6)
```

STOPPED AT PAGE 41 ON 2021-01-28 BEGINNING AT PAGE 41 ON 2021-02-09

Use the summary function to summarize the vector of numbers named 'mynumbers'.

```
mysummary <- summary(mynumbers)

mysummary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   1.500   3.000   5.714   4.000  25.000
```

If you are not sure what an object is, ask for its class

```
class(mynumbers)
```

```
## [1] "numeric"
```

```
class(mysummary)
```

```
## [1] "summaryDefault" "table"
```

```
class(summary)
```

```
## [1] "function"
```

R can store datasets in many different ways, but the most popular (and most often my use case) is the data frame. A data frame has columns representing variables and rows representing observations.

The $ operator allows us to reference a specific variable from a dataframe.

```
titanic
```

```
##       fate    sex    n percent
## 1 perished   male 1364    62.0
## 2 perished female  126     5.7
## 3 survived   male  367    16.7
## 4 survived female  344    15.6
```

```
class(titanic)
```

```
## [1] "data.frame"
```

```
titanic$percent
```

```
## [1] 62.0  5.7 16.7 15.6
```

The tidyverse libraries make extensive use of *tibbles*, which are very similar to dataframes.(Although the data.frame output above looks the same as the tibble output... Maybe an update to R? Anyways, should probably use Tibbles.)

```
titanictb <- as_tibble(titanic)
titanictb
```

```
## # A tibble: 4 x 4
##   fate     sex        n percent
##   <fct>    <fct>  <dbl>   <dbl>
## 1 perished male    1364    62
## 2 perished female   126     5.7
## 3 survived male     367    16.7
## 4 survived female   344    15.6
```

#Input data

The `read_csv()' function in the`readr' package is used to input CSV files. Other packages that can input STATA, SAS, and other software datasets directly can be found in the 'haven' package.

STOPPED AT PAGE 50 ON 2021-02-09

```
url <- "https://cdn.rawgit.com/kjhealy/viz-organdata/master/organdonation.csv"

organs <- read_csv(file = url)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   .default = col_double(),
##   country = col_character(),
##   world = col_character(),
##   opt = col_character(),
##   consent.law = col_character(),
##   consent.practice = col_character(),
##   consistent = col_character(),
##   ccode = col_character()
## )
## i Use `spec()` for the full column specifications.
```

Make a Plot

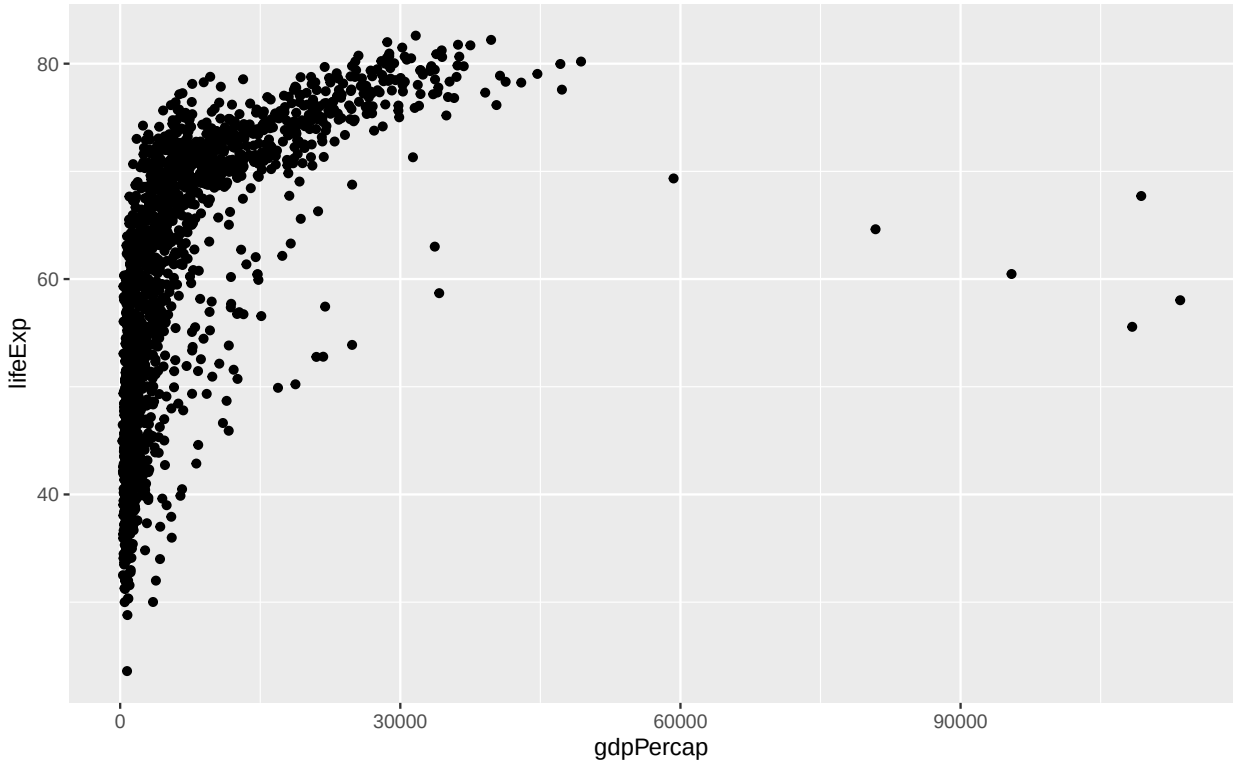
View the Gapminder data to see what we are working with.

```
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

Create a basic scatter plot of the Gapminder data.

```
p <- ggplot(data=gapminder,
            mapping=aes(x=gdpPercap, y=lifeExp))
p + geom_point()
```



###Create another scatterplot using Gapminder data. Will elaborate on this in more detail.

```
#First, tell ggplot the data we are using.
p <- ggplot(data=gapminder)

#Second, tell ggplot which variables in data should be represented by which visual elements in the plot
p <- ggplot(data=gapminder,
            mapping=aes(x=gdpPercap,
                        y=lifeExp))
```

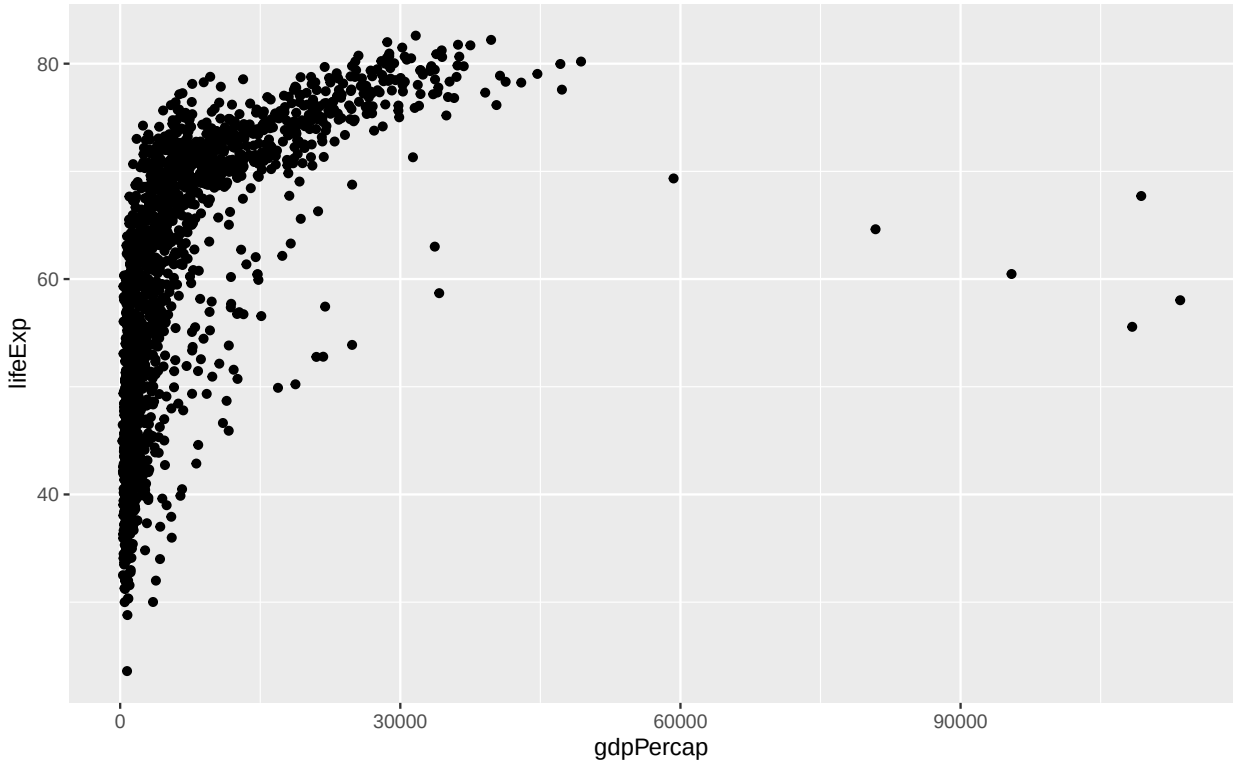
So far, we have given the ggplot function two arguments: `data` and `mapping`.

The `mapping` argument is itself a FUNCTION (i.e., the `mapping` function is an argument to the `ggplot` function). The `mapping = aes(...)` argument links variables to things you will see on the plot. `x` and `y` values are obvious. Other aesthetic mappings include color, shape, size, line type, etc. **IMPORTANT** A mapping does not say what particular color, shape, line type, etc. will be on the plot. It simply says which *variables* in the data will be *represented* by these aesthetics (e.g., % of fake news is represented by color).

At this point, we have created the `p` object using the `ggplot()` function and given it some basic information (i.e., `data` and `mapping`). `ggplot()` has also created `p` using a lot of other information as defaults. To see how much default information ask for `str(p)`.

Before a plot can be made, we must tell `ggplot()` what type of plot to draw. This is called adding a *layer* to the plot. This simply means picking a `geom_` function. The `geom_point()` function creates a scatterplot.

```
#Create a scatterplot
p + geom_point()
```



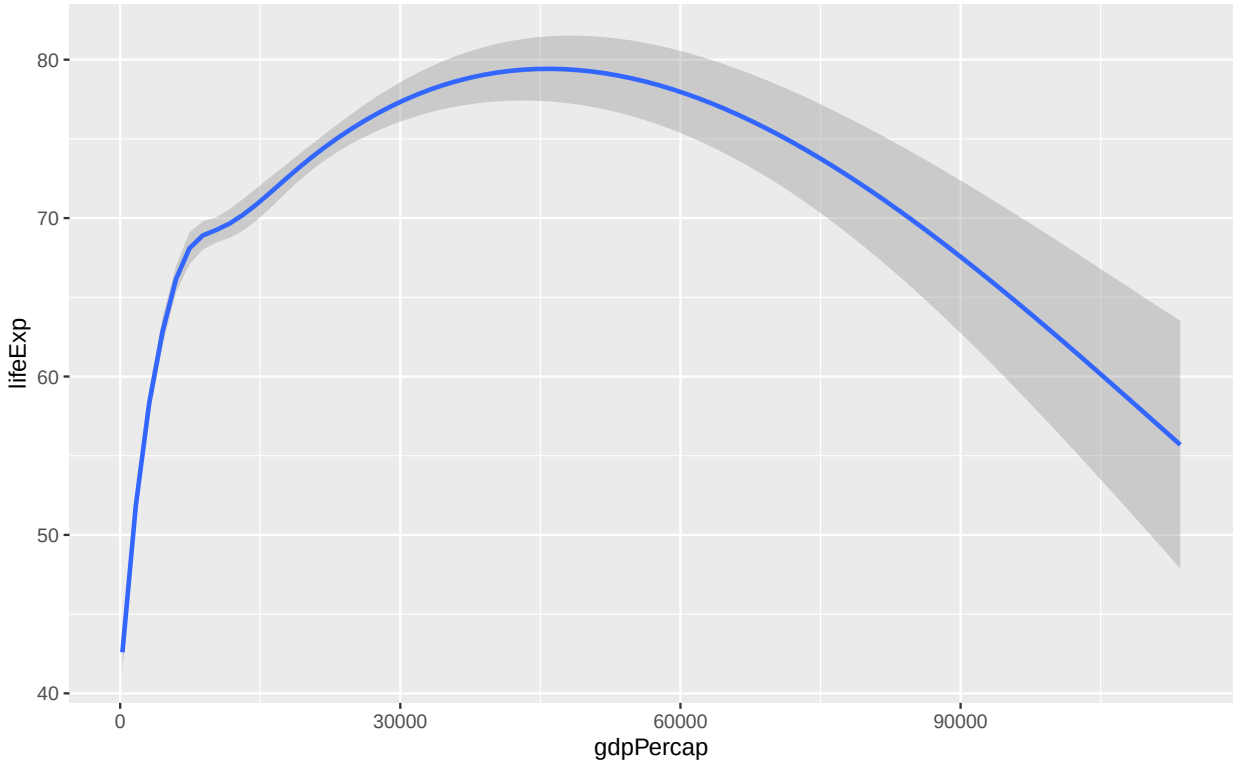
The basic recipe to create a plot with ggplot is:

1. Tell the ggplot() function what our data is.
2. Tell ggplot() what relationships we want to see. For convenience we will put the results of the first two steps in an object called p.
3. Tell ggplot how we want to see the relationships in our data.
4. Layer on geoms as needed, by adding them to the p object one at a time.
5. Use some additional functions to adjust scales, labels, tick marks, titles. We'll learn more about some of these functions shortly.

Let's try adding some additional layers to our scatterplot

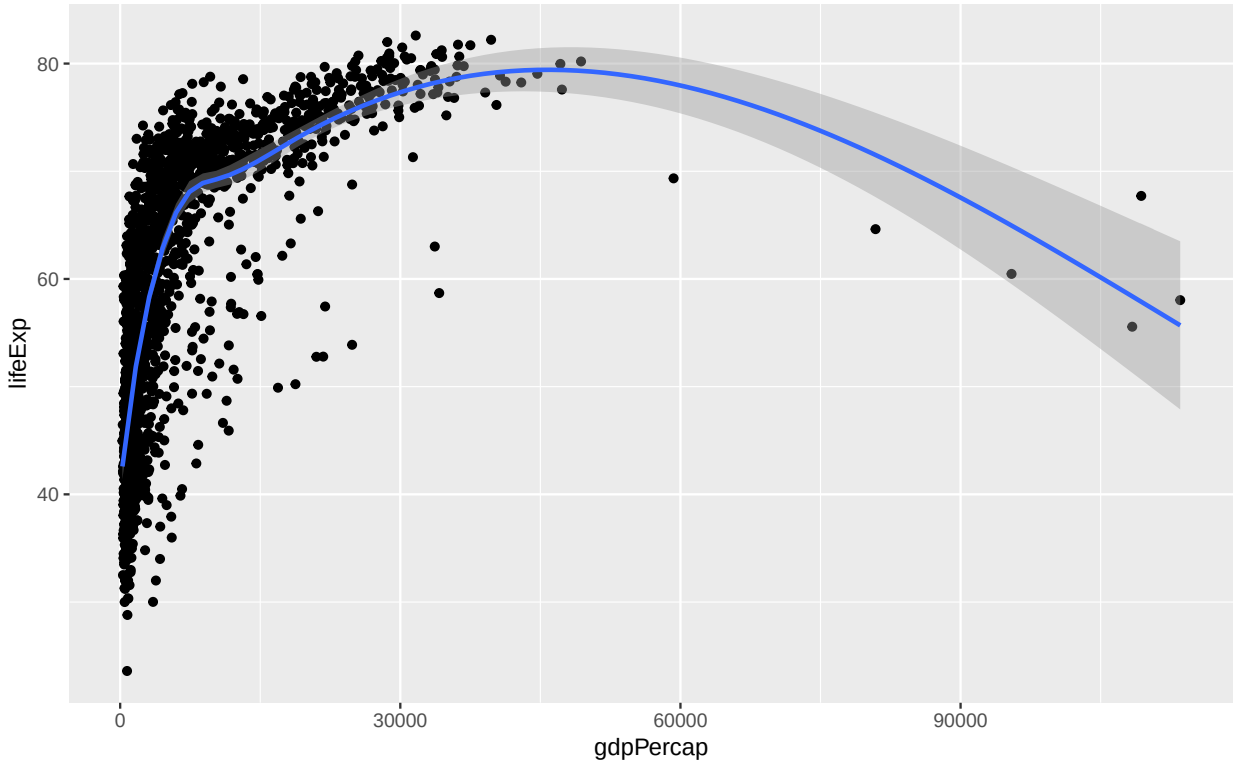
```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y=lifeExp))
#Create a smoothed line with standard error shading
p + geom_smooth()
```

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



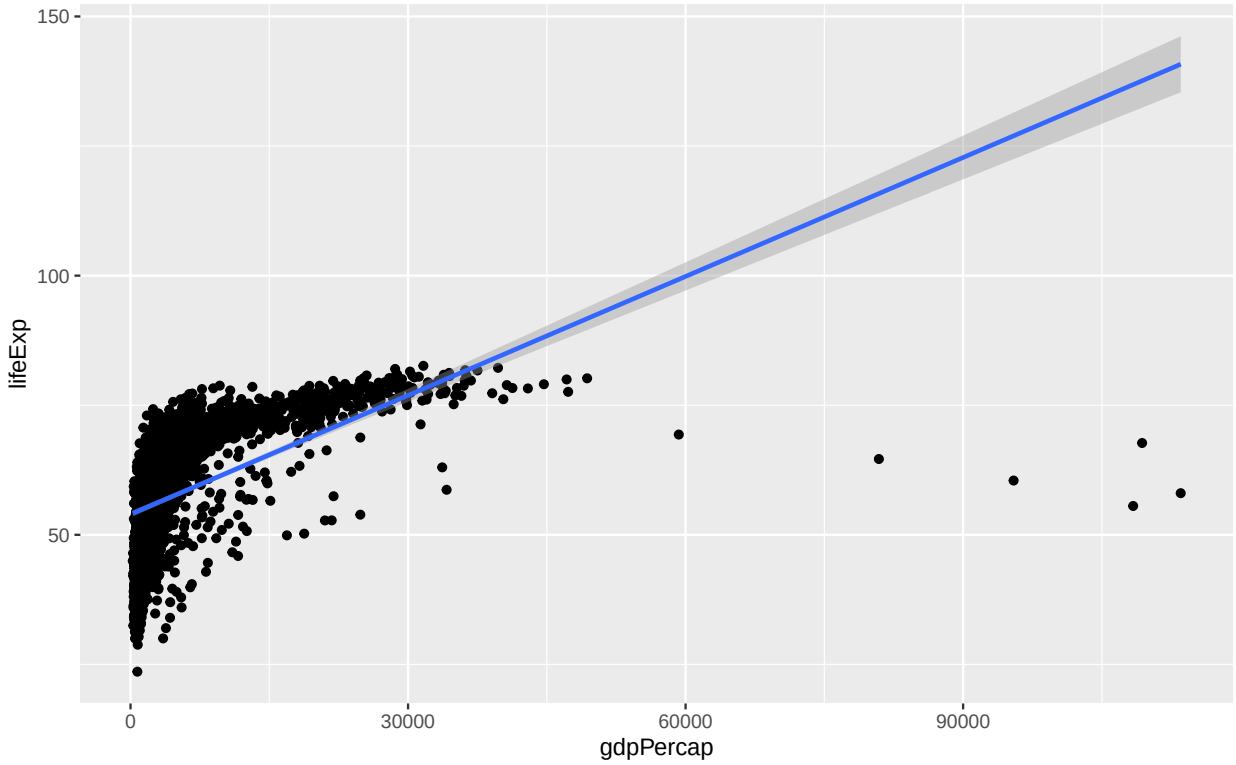

```
#Smoothed line PLUS scatterplot
p + geom_point() + geom_smooth()
```

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



```
#Smoothed line fit using a linear model PLUS scatterplot
p + geom_point() + geom_smooth(method = "lm")
```

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

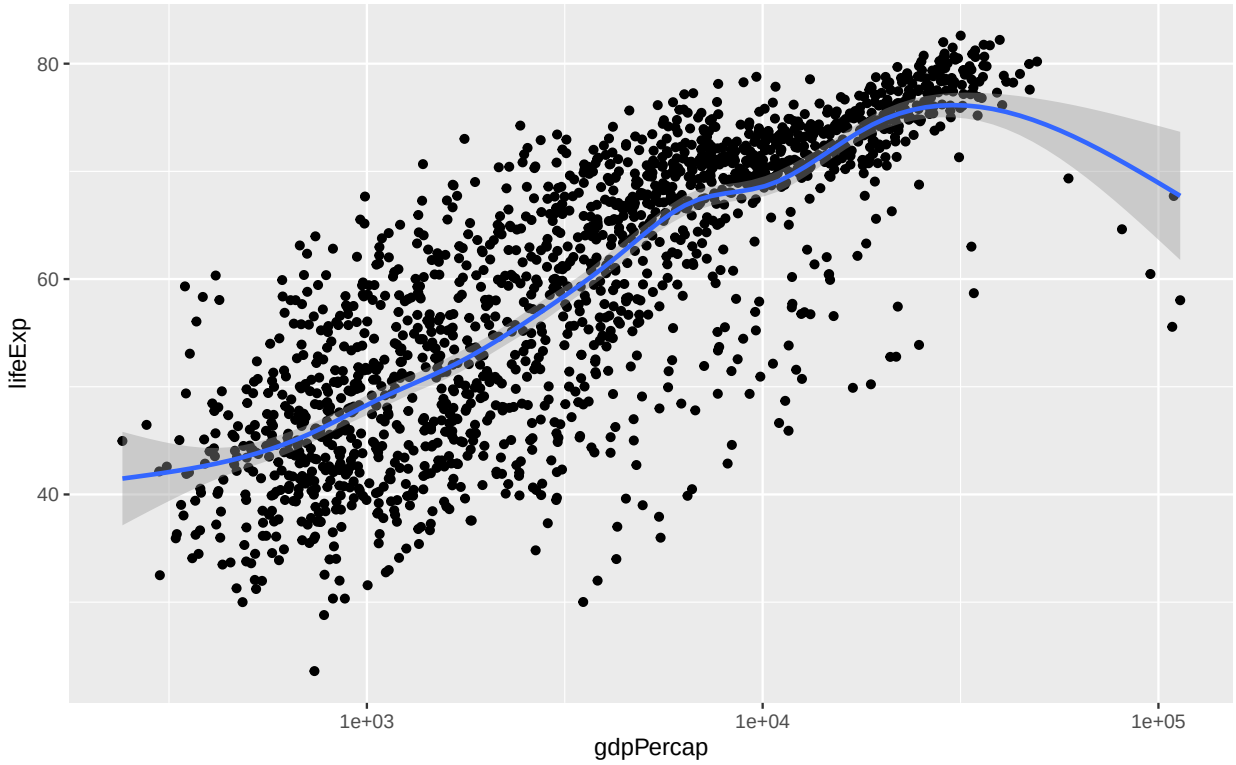


Notice that `geom_point()` and `geom_smooth` inherited the information from object `p` as default. We can give geoms separate instructions that they will follow instead.

In our Gapminder data, the data is quite bunched to the left side. The plot may look better if we transformed the x-axis from a linear scale to a log scale. Additionally, test out what happens if we switch the order of the `geom_` functions... It seems that the plot is created in order of the functions specified, so functions later in the list are "on top" of earlier functions.

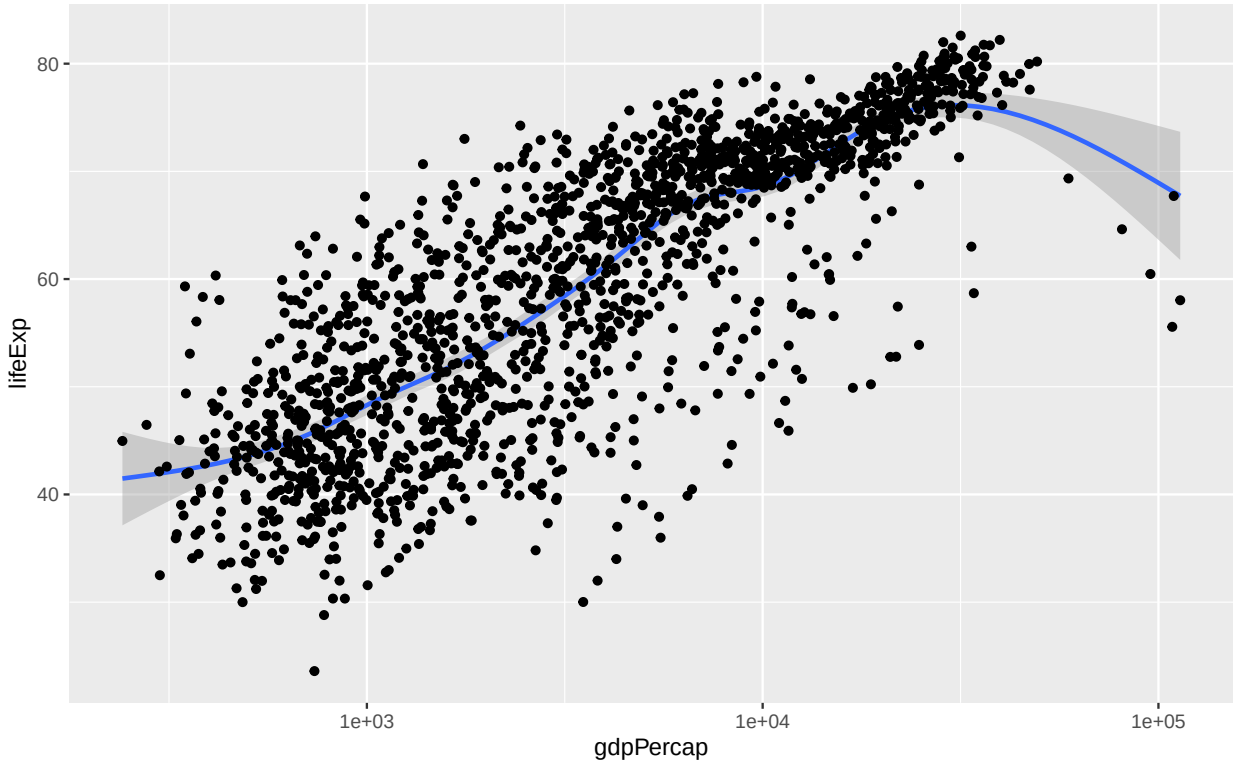
```
p + geom_point() +
    geom_smooth(method = "gam") +
    scale_x_log10()
```

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




```
p + geom_smooth(method = "gam") +
    geom_point() +
    scale_x_log10()
```

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

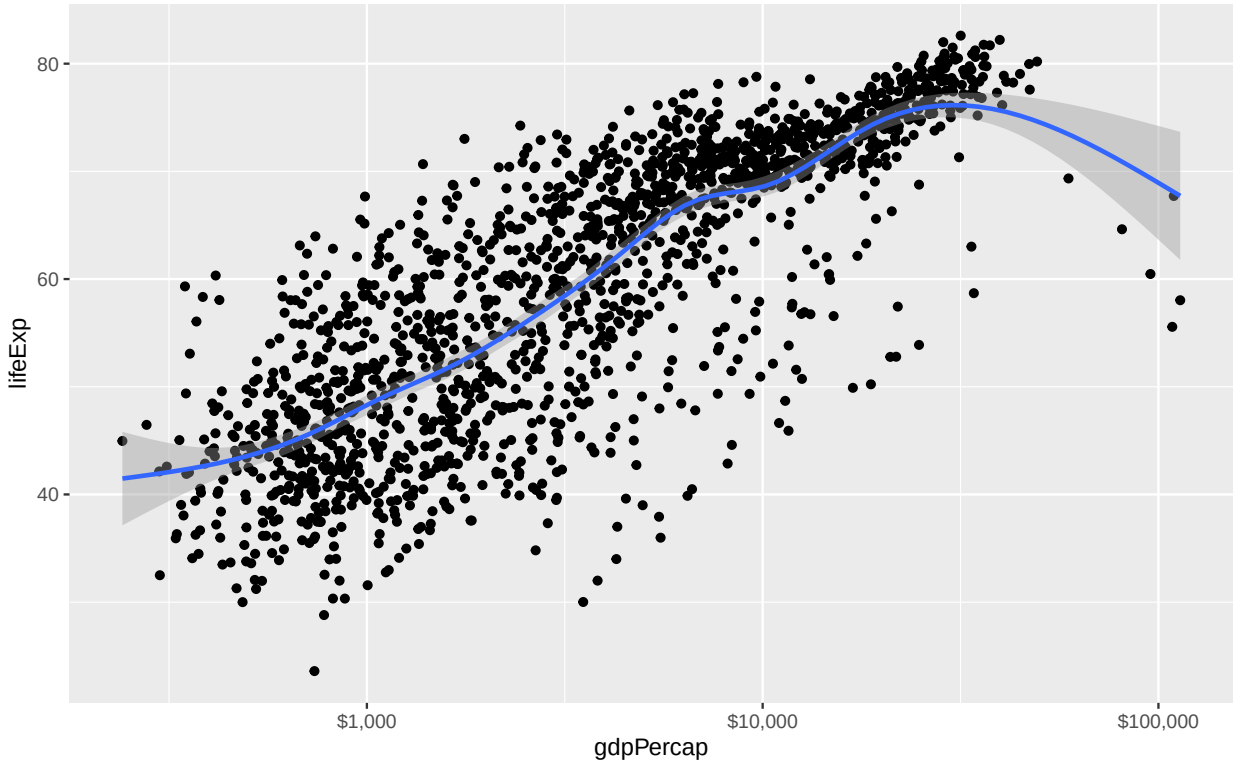


Let's clean up our scatterplot

We likely want to polish our plot with nicer axis labels, a title, and different x-axis notation (from scientific to dollars). Let's only worry about the x-axis notation for now.

```
p + geom_point() +
    geom_smooth(method = "gam") +
    scale_x_log10(labels = scales::dollar)
```

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



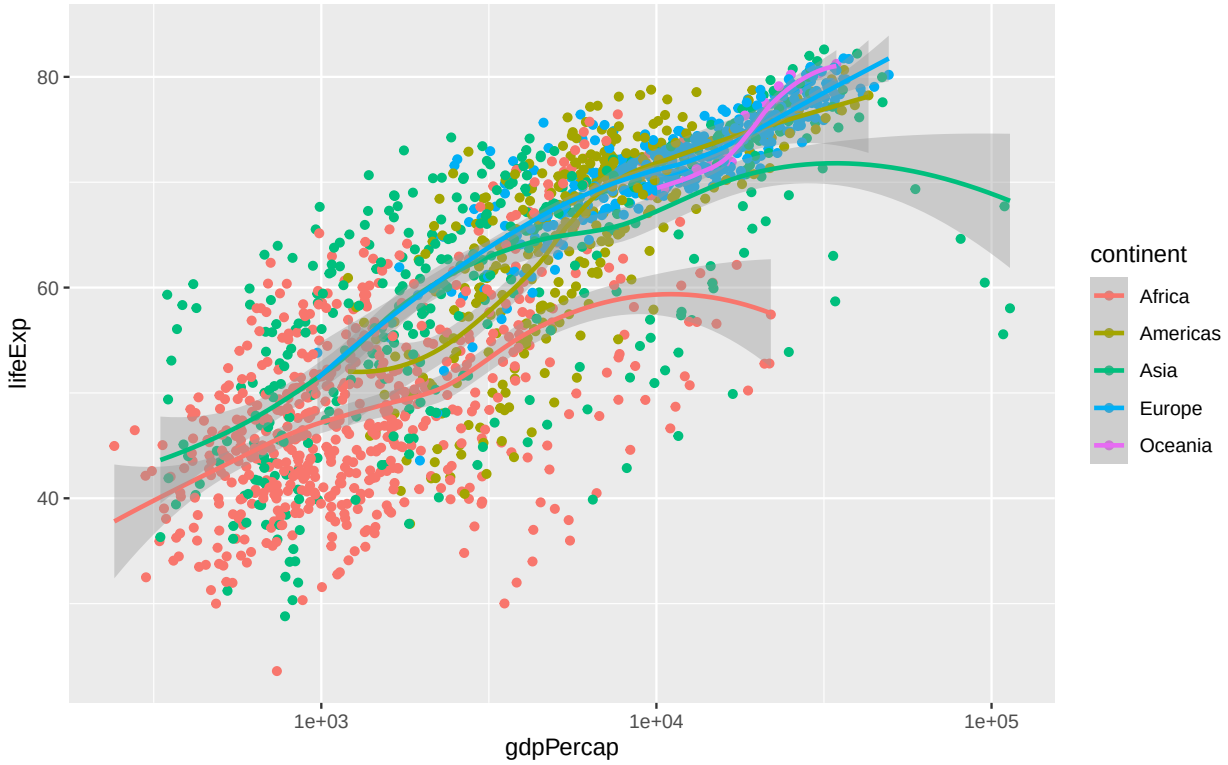
Notice that the scale of the x-axis is changed by passing an argument to the `scale_` functions.The `scales` library contains useful pre-made formatting functions. If a library is not already loaded then a specific function can be grabbed from that library using the syntax `thelibrary::thefunction`. Otherwise, load the library using `library(scales)`.

Mapping vs setting aesthetics

Let's map the variable `continent` to be represented by color.

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp,
                          color = continent))
#Create a plot that represents each continent in our data with a different color
p + geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10()
```

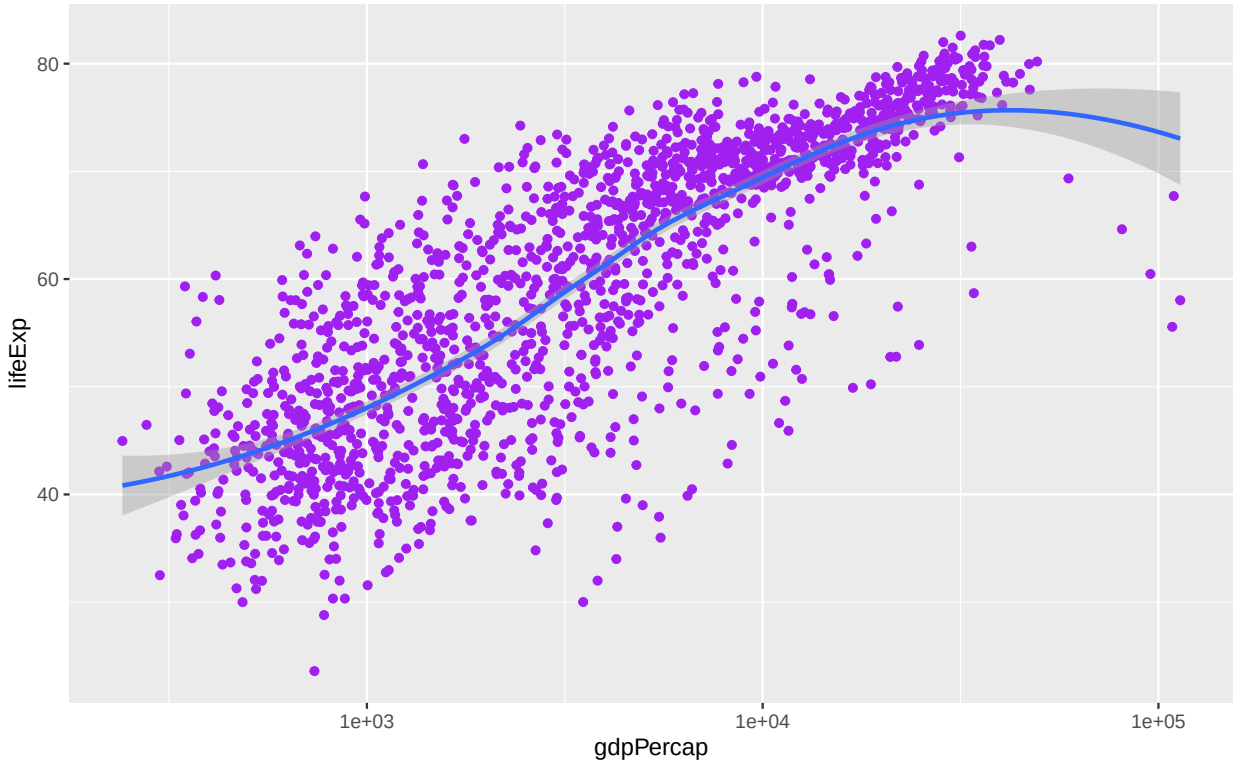
```
## `geom_smooth()` using formula 'y ~ x'
```



If we want to *set* a property (i.e., change the scatterpoints to be a different color – say, purple), we do this within the `geom_` function.

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp))

#Create our original scatterplot and smoothed line using the color purple
p + geom_point(color = "purple") +
    geom_smooth(method = "loess") +
    scale_x_log10()
```

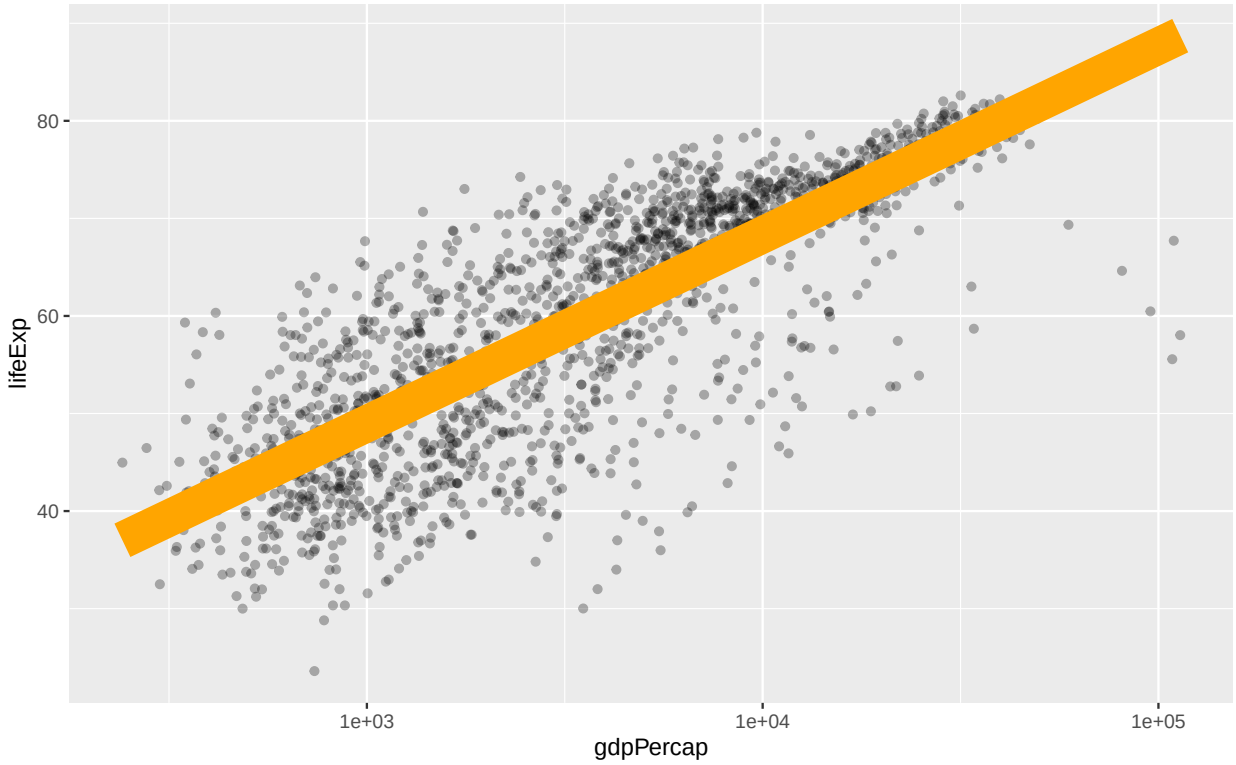
```
## `geom_smooth()` using formula 'y ~ x'
```



Show some additional adjustments that can be made to plots by giving the `geom_` functions various arguments. *alpha* changes the transparency of the objects. *se* turns off the standard errors shading. *size* changes the size of the line.

```
p + geom_point(alpha = 0.3) +
    geom_smooth(color = "orange",
                se = FALSE,
                size = 8,
                method = "lm") +
    scale_x_log10()
```

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

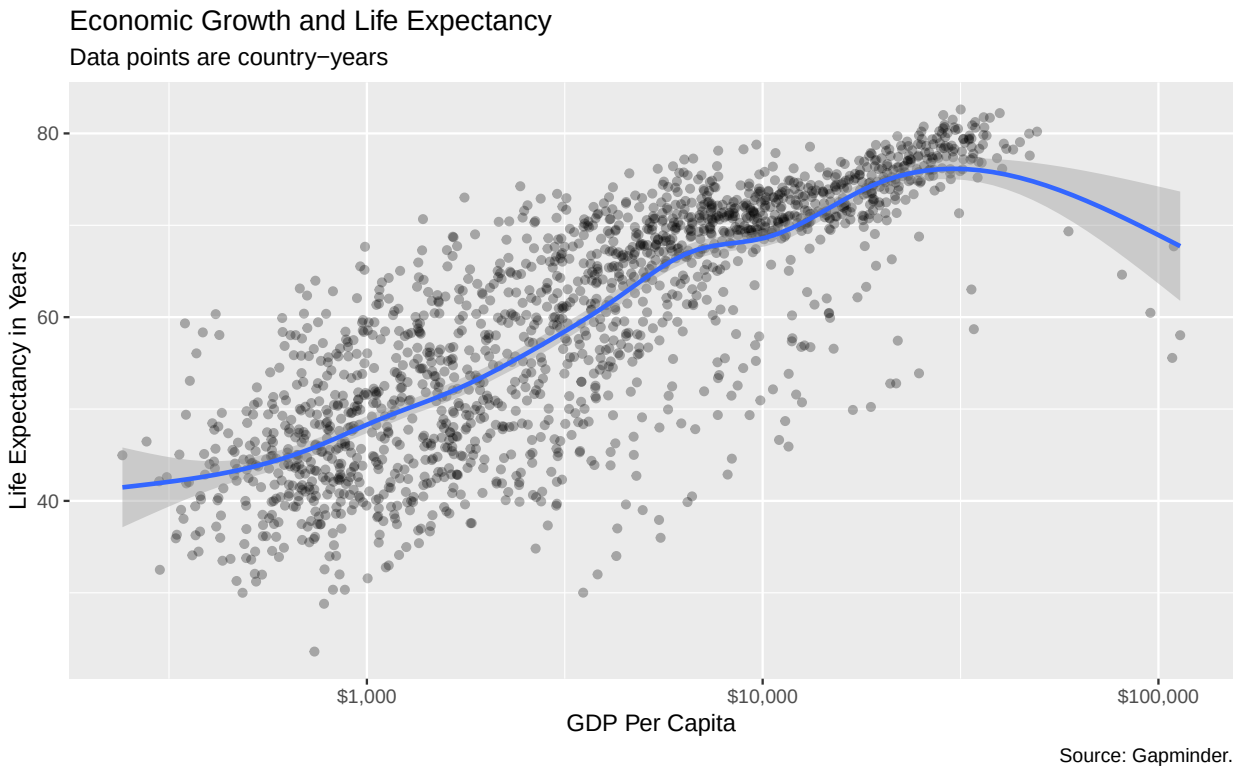


Let's add some labels

Let's continue to polish our scatterplot by adding appropriate labels.

```
p + geom_point(alpha = 0.3) +
    geom_smooth(method = "gam") +
    scale_x_log10(labels = scales::dollar) +
    labs(x = "GDP Per Capita",
         y = "Life Expectancy in Years",
         title = "Economic Growth and Life Expectancy",
         subtitle = "Data points are country-years",
         caption = "Source: Gapminder.")
```

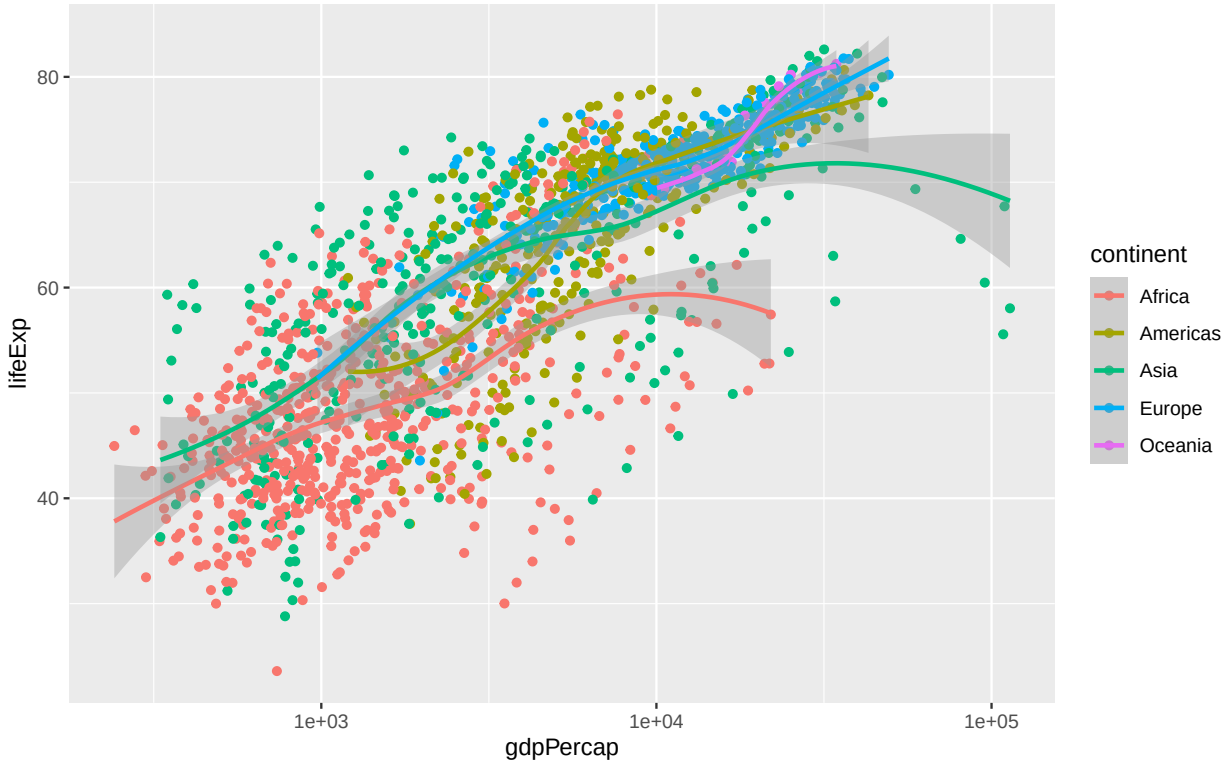
```
## `geom_smooth()` using formula 'y ~ s(x, bs = "cs")'
```



Let's go back to mapping the continent variable to color.

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp,
                          color = continent))
#Create a plot that represents each continent in our data with a different color
p + geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10()
```

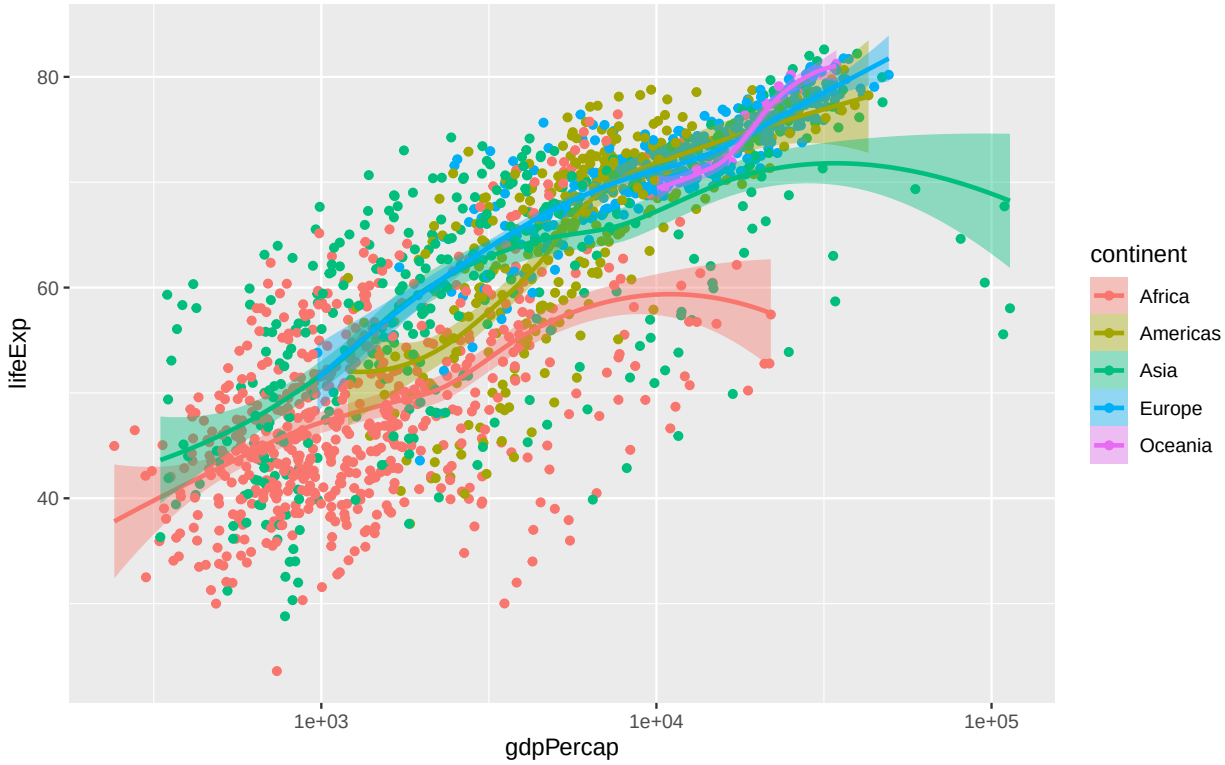
```
## `geom_smooth()` using formula 'y ~ x'
```



We should shade the standard error ribbon for each line the same color as the line. The color of the standard error ribbon is controlled by the `fill` aesthetic. **whereas** the `color` aesthetic affects the appearance of lines and points, `fill` affects appearance of the filled areas of bars, polygons, and, in this case, the interior of the smoother's standard error ribbon.

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp,
                          color = continent,
                          fill = continent))

#Create a plot that represents each continent in our data with a different color
p + geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10()
```

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

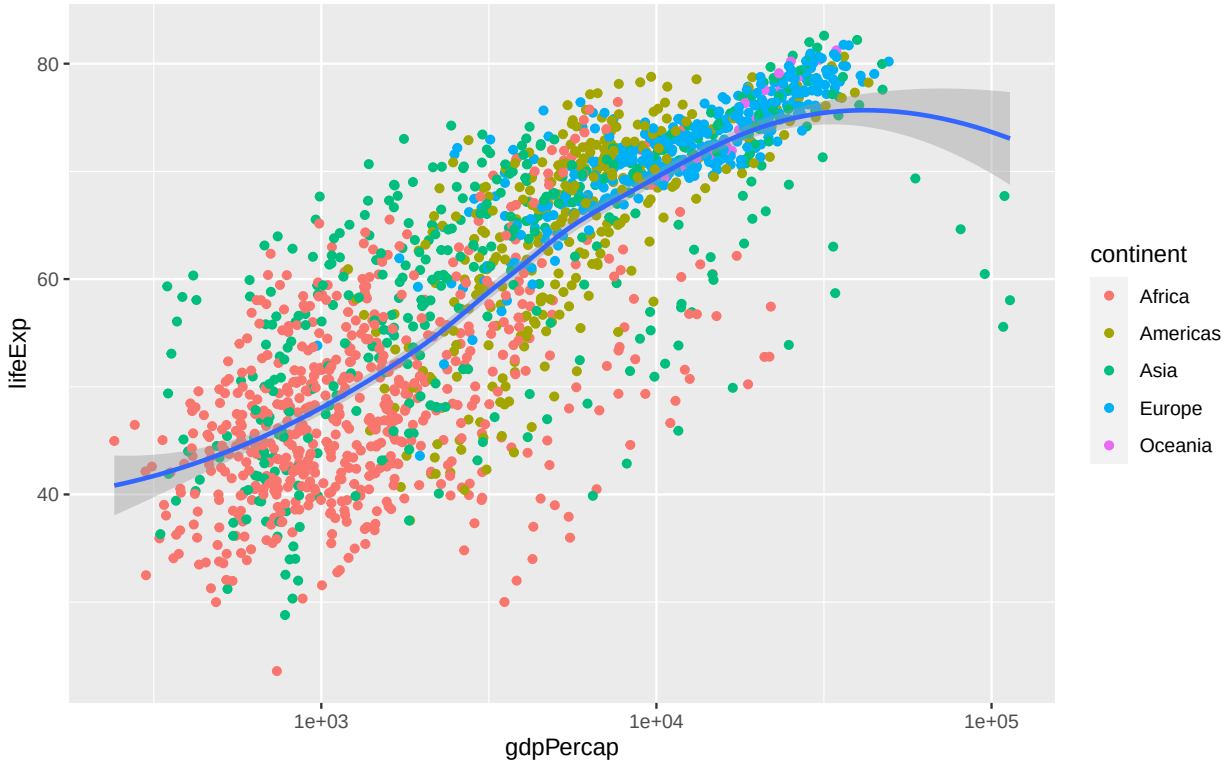


Let's say that we think the above plot is too busy with five fitted lines. Instead, we want only one fitted line but to still shade each data point according to its continent.

Remember, by default, geoms inherit their mappings from the `ggplot()` function. However, we can change this by mapping the aesthetics we want only to the `geom_` functions that we want them to apply to. We use the same `mapping = aes(...)` syntax that is in the `ggplot()` function, but now we write it directly in the `geom_` function.

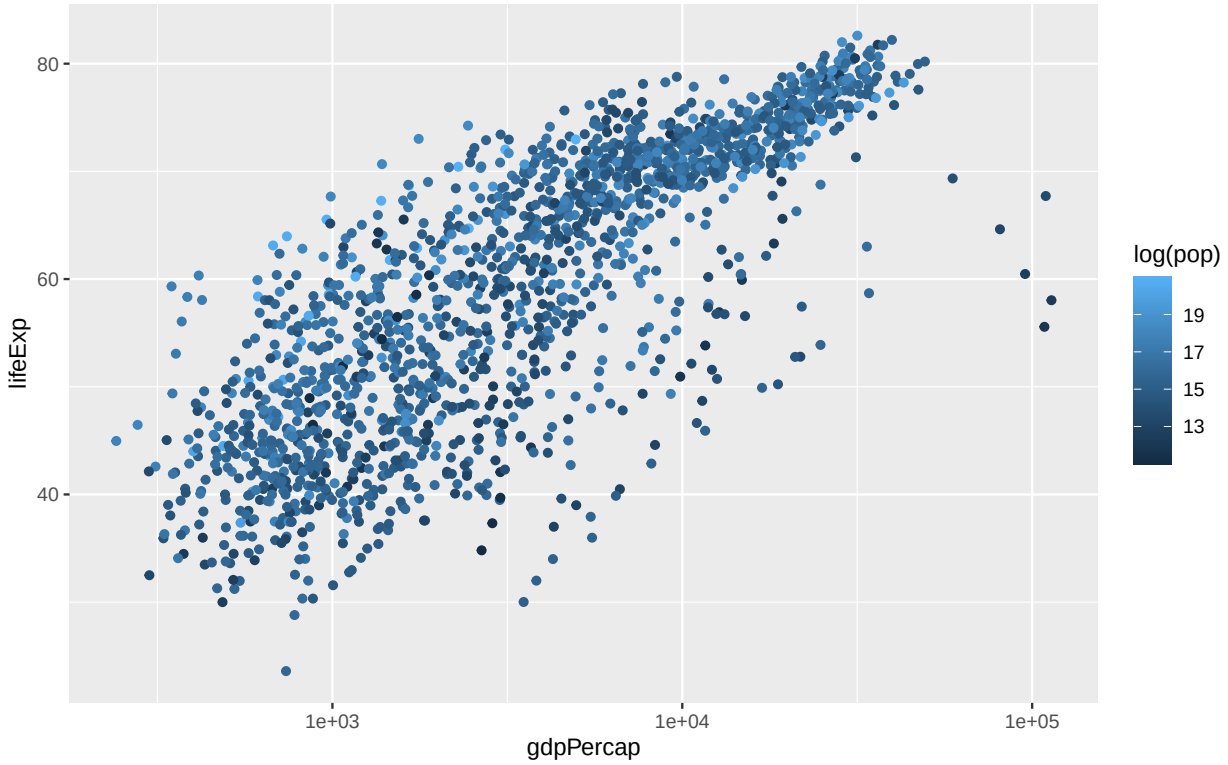
```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp))

#Create a plot--directly map continent to color within the geom_point() function, so it only applies to
p + geom_point(mapping = aes(color = continent)) +
    geom_smooth(method = "loess") +
    scale_x_log10()
```

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



We can also map continuous variables to color. It will present the data as a gradient of the color and provide a discretized scale for interpretation. Additionally, notice that we can transform variables directly within the `aes(...)` statement.

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp))
p + geom_point(mapping = aes(color = log(pop))) +
    scale_x_log10()
```

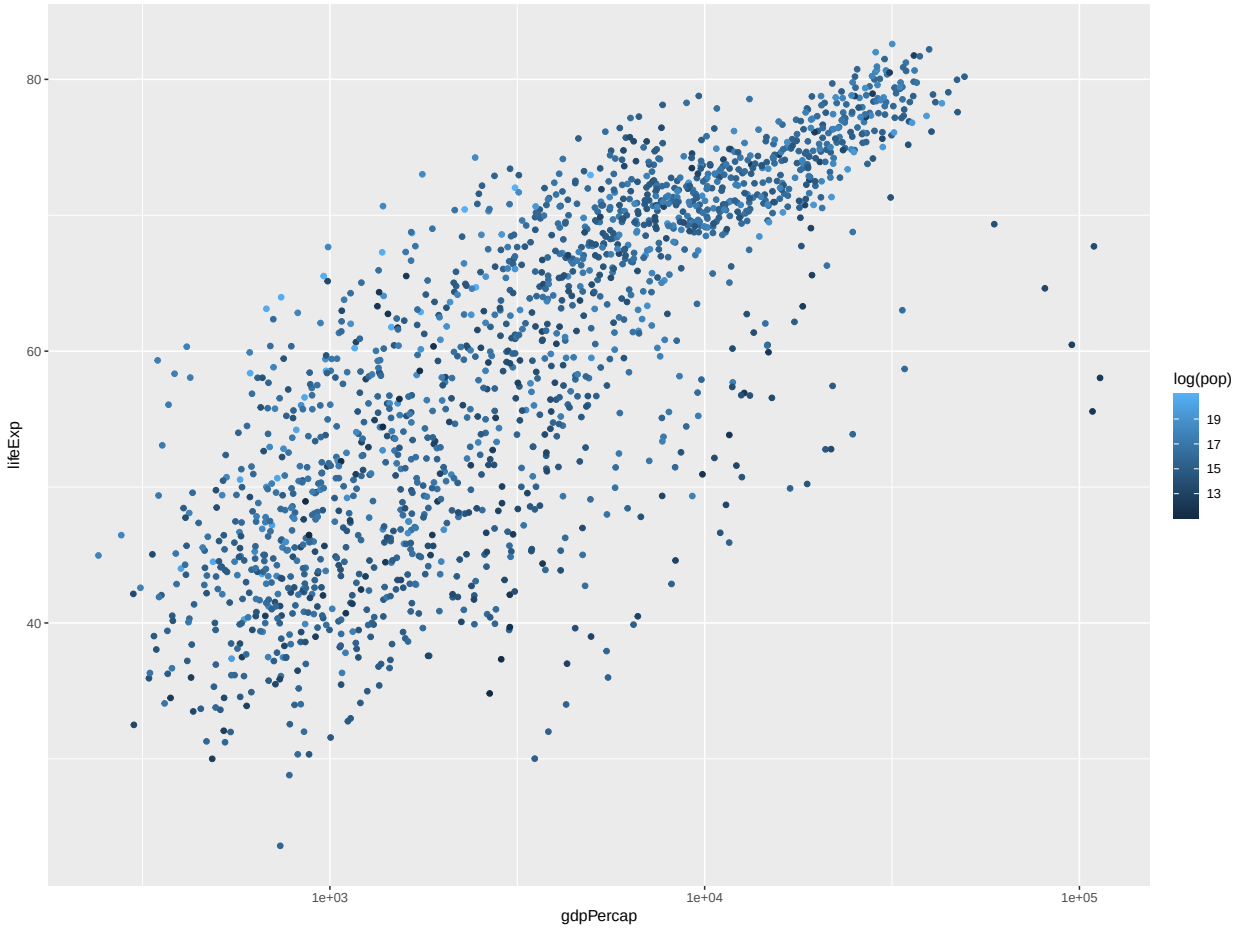


Customizing plot export options

To change the size of your plots, you can either change the default settings for the entire `.Rmd` document by setting an option in your first code chunk. The syntax tells R to make 8X5 figures: `knitr::opts_chunk$set(fig.width=8, fig.height=5)`

More practically, we will want to customize the size of specific plots. This can be done by adding options to the `R` code chunk as follows:

```
p + geom_point(mapping = aes(color = log(pop))) +
    scale_x_log10()
```



To save a plot, the most convenient method is the `ggsave()` function. `ggsave()` will save the most recently displayed figure. Syntax: `ggsave(filename = "my_figure.png")`. Formats other than `.png` are available as well, most notably `.pdf`.

Instead of outputting our plots when we call our *layers*, we can assign a plot to an object just like any other thing in R. We can then save this plot at any point by giving the `plot` argument to the `ggsave()` function. For example:

```
p1 <- ggplot(data = gapminder,
             mapping = aes(x = gdpPercap,
                           y = lifeExp))

p1_out <- p1 + geom_point(mapping = aes(color = log(pop))) +
    scale_x_log10()

ggsave("my_figure.pdf", plot = p1_out)
```

```
## Saving 8 x 5 in image
```

STOPPED AT PAGE 72 ON 2021-03-17

Show the Right Numbers

Graph Tables, Make Labels, Add Notes

Work with Models

Draw Maps

Refine your Plots