Data Visualization, Exploration and Analysis
Practical Notes for Making Plots and Doing Regression Analysis in R

Kamarul Imran Musa
Wan Nor Arifin

2019-02-17

Contents

1 Data Visualization

1.1 Introduction to visualization

Data visualization is viewed by many disciplines as a modern equivalent of visual communication. It involves the creation and study of the visual representation of data.

Data visualization requires "information that has been abstracted in some schematic form, including attributes or variables for the units of information".

References on data visualization:

1. Link 1 https://en.m.wikipedia.org/wiki/Data_visualization
2. Link 2 https://en.m.wikipedia.org/wiki/Michael_Friendly

1.1.1 History of data visualization

1983 book The Visual Display of Quantitative Information, Edward Tufte defines **graphical displays** and principles for effective graphical display

The book defines "Excellence in statistical graphics consists of complex ideas communicated with clarity, precision and efficiency."

1.1.2 Processes and Objectives of visualization

Visualization is the process of representing data graphically and interacting with these representations. The objective is to gain insight into the data.

Reference: http://researcher.watson.ibm.com/researcher/view_group.php?id=143

1.2 What makes good graphics

You may require these to make good graphics:

1. Data
2. Substance rather than about methodology, graphic design, the technology of graphic production or something else
3. No distortion to what the data has to say
4. Presence of many numbers in a small space
5. Coherence for large data sets
6. Encourage the eye to compare different pieces of data
7. Reveal the data at several levels of detail, from a broad overview to the fine structure
8. Serve a reasonably clear purpose: description, exploration, tabulation or decoration
9. Be closely integrated with the statistical and verbal descriptions of a data set.

1.3 Graphics packages in R

There are many **graphics packages** in R. Some packages are aimed to perform general tasks related with graphs. Some provide specific graphics for certain analyses.

The popular general graphics packages in R are:

1. **graphics** : a base R package
2. **ggplot2** : a user-contributed package by Hadley Wickham

3. **lattice** : a user-contributed package

Except for **graphics** package (a a base R package), other packages need to downloaded and installed into your R library.

Examples of other more specific packages - to run graphics for certain analyses - are:

1. **survminer::ggsurvlot**
2. **sjPlot**

For this course, we will focus on using the **ggplot2** package.

1.4 Introduction to ggplot2 package

- **ggplot2** is an elegant, easy and versatile general graphics package in R.
- it implements the **grammar of graphics** concept
- the advantage of this concept is that, it fasten the process of learning graphics
- it also facilitates the process of creating complex graphics

To work with **ggplot2**, remember

- start with: `ggplot()`
- which data: `data = X`
- which variables: `aes(x = , y = )`
- which graph: `geom_histogram()`, `geom_points()`

The official website for ggplot2 is here http://ggplot2.org/.

ggplot2 is a plotting system for R, based on the grammar of graphics, which tries to take the good parts of base and lattice graphics and none of the bad parts. It takes care of many of the fiddly details that make plotting a hassle (like drawing legends) as well as providing a powerful model of graphics that makes it easy to produce complex multi-layered graphics.

1.5 Preparation

1.5.1 Set a new project or set the working directory

It is always recommended that to start working on data analysis in RStudio, you create first a new project.

Go to File, then click New Project.

You can create a new R project based on existing directory. This method is useful because an RStudio project keep your data, your analysis, and outputs in a clean dedicated folder or sets of folders.

If you do not want to create a new project, then make sure you are inside the correct directory (the working directory). The working directory is a folder where you store.

Type `getwd()` in your Console to display your working directory. Inside your working directory, you should see and keep

1. dataset or datasets
2. outputs - plots
3. codes (R scripts `.R`, R markdown files `.Rmd`)

1.5.2 Questions to ask before making graphs

You must ask yourselves these:

1. Which variable or variables do I want to plot?
2. What is (or are) the type of that variable?

- Are they factor (categorical) variables ?
- Are they numerical variables?

3. Am I going to plot

- a single variable?
- two variables together?
- three variables together?

1.5.3 Read data

The common data formats include

1. comma separated files (`.csv`)
2. MS Excel file (`.xlsx`)
3. SPSS file (`.sav`)
4. Stata file (`.dta`)
5. SAS file

Packages that read these data include **haven** package

1. SAS: `read_sas()` reads .sas7bdat + .sas7bcat files and read_xpt() reads SAS transport files (version 5 and version 8). write_sas() writes .sas7bdat files.
2. SPSS: `read_sav()` reads .sav files and read_por() reads the older .por files. write_sav() writes .sav files.
3. Stata: `read_dta()` reads .dta files (up to version 15). write_dta() writes .dta files (versions 8-15).

Data from databases are less common but are getting more important and more common. Some examples of databases

1. MySQL
2. SQLite
3. Postgresql
4. Mariadb

1.5.4 Load the library

ggplot2 is one of the core member of **tidyverse** package (https://www.tidyverse.org/).

Once we load the **tidyverse** package, we will also have access to

1. help pages
2. functions
3. datasets

```
library(tidyverse)
```

```
## -- Attaching packages --------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.1.0     v purrr   0.2.5
## v tibble  1.4.2     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.3.1     v forcats 0.3.0
```

```
## -- Conflicts ------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
```

```
## x dplyr::lag()    masks stats::lag()
```

If you run the code and you see *there is no package called tidyverse* then you need to install the **tidyverse** package.

to do that type `install.package("tidyverse")`, then run again `library(tidyverse)`.

1.5.5 Open dataset

For now, we will use the built-in dataset in the **gapminder** package.

You can read more about *gapminder* from https://www.gapminder.org/

The website contains many useful datasets and show wonderful graphics. It is made popular by Dr Hans Rosling.

Load the package,

```
library(gapminder)
```

call the data into R and browse the data the top of the data

```
head(gapminder)
```

```
## # A tibble: 6 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.
```

We can list the variables and look at the type of the variables in the dataset

```
glimpse(gapminder)
```

```
## Observations: 1,704
## Variables: 6
## $ country   <fct> Afghanistan, Afghanistan, Afghanistan, Afghanistan, ...
## $ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia...
## $ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992...
## $ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.8...
## $ pop       <int> 8425333, 9240934, 10267083, 11537966, 13079460, 1488...
## $ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 78...
```

The data have

1. 6 variables
2. 1704 observations
3. There are 2 factor variables, 2 integer variables and 2 numeric variables

We can examine the basic statistics of the datasets by using `summary()`. It will list

1. frequencies
2. min, 1st quartile, median, mean, 3rd quartile and max

```
summary(gapminder)
```

```
##         country        continent        year         lifeExp
##  Afghanistan:  12   Africa  :624   Min.   :1952   Min.   :23.60
##  Albania    :  12   Americas:300   1st Qu.:1966   1st Qu.:48.20
##  Algeria    :  12   Asia    :396   Median :1980   Median :60.71
##  Angola     :  12   Europe  :360   Mean   :1980   Mean   :59.47
##  Argentina  :  12   Oceania : 24   3rd Qu.:1993   3rd Qu.:70.85
##  Australia  :  12                  Max.   :2007   Max.   :82.60
##  (Other)    :1632
##       pop              gdpPercap
##  Min.   :6.001e+04   Min.   :   241.2
##  1st Qu.:2.794e+06   1st Qu.:  1202.1
##  Median :7.024e+06   Median :  3531.8
##  Mean   :2.960e+07   Mean   :  7215.3
##  3rd Qu.:1.959e+07   3rd Qu.:  9325.5
##  Max.   :1.319e+09   Max.   :113523.1
##
```

To know more about the the package, we can use ?

```
?gapminder
```

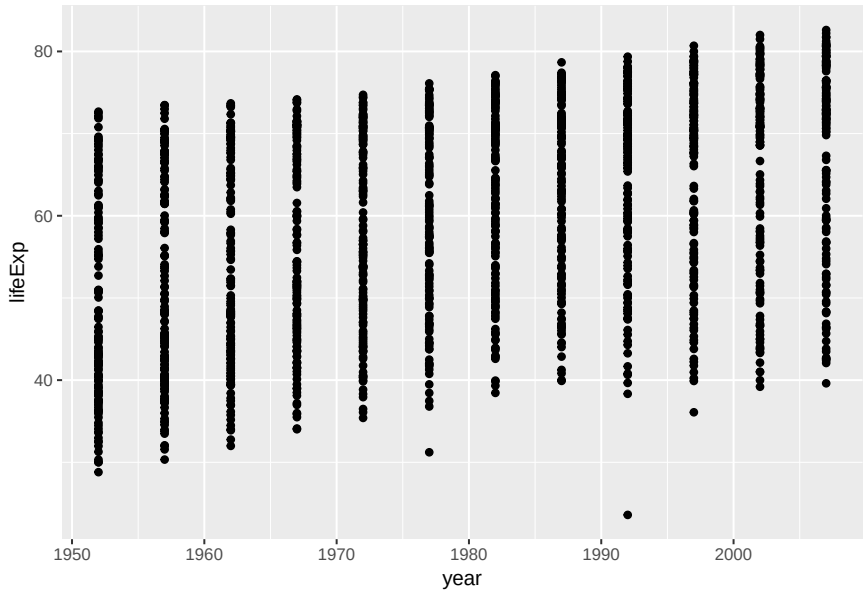
```
## starting httpd help server ... done
```

1.6 Basic plot

We can start create a basic plot

- data = gapminder
- variables = year, lifeExp
- graph = scatterplot

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp))
```



The plot shows:

1. the relationship between year and life expectancy.

2. as year advances, the life expectancy increases.

the `ggplot()` tells R to plot what variables from what data. And `geom_point()` tells R to make a scatter plot.

1.7 Adding another variable

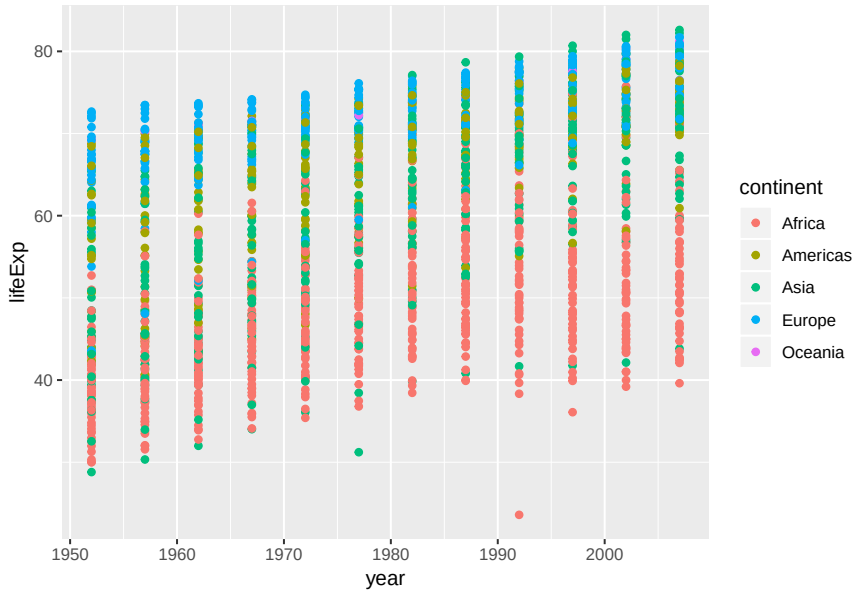
You realize that we plotted 2 variables based on `aes()`. We can add the third variable to make a more complicated plot.

For example:

1. data = gapminder
2. variables = year, life expectancy, continent

Objective: to plot the relationship between year and life expectancy based on continent.

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp, colour = continent))
```

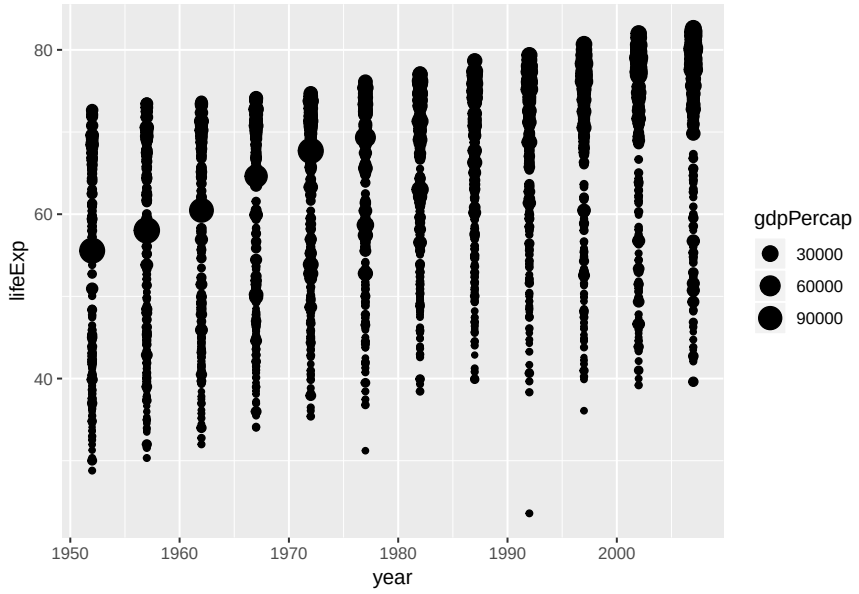


What can you see from the scatterplot.

1. Europe countries have high life expectancy
2. Africa countries have lower life expectancy
3. One Asia country looks like an outlier (very low life expectancy)
4. One Africa country looks like an outlier (very low life expectancy)

Now, we will replace the 3rd variable with GDP (variable gdpPercap) and make the plot correlates with the size of GDP.

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp, size = gdpPercap))
```



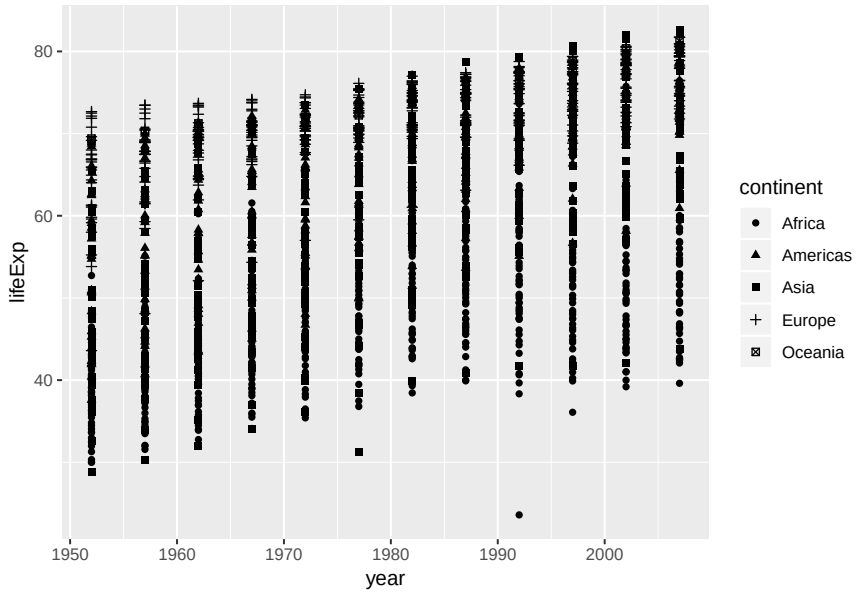
ggplot2 will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as scaling.

ggplot2 will also add a legend that explains which levels correspond to which values.

The plot suggets that higher GDP countries have longer life expectancy.

Instead of using colour, we can use shape especially in instances where there is no facility to print out colour plots

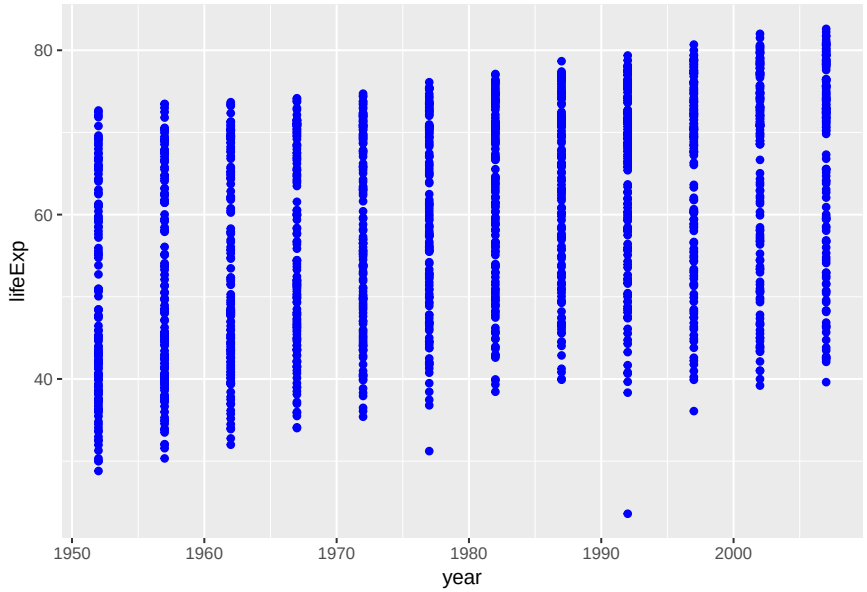
```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp, shape = continent))
```



But, see what will happen if you set the colour and shape like below but outside the aes parentheses.

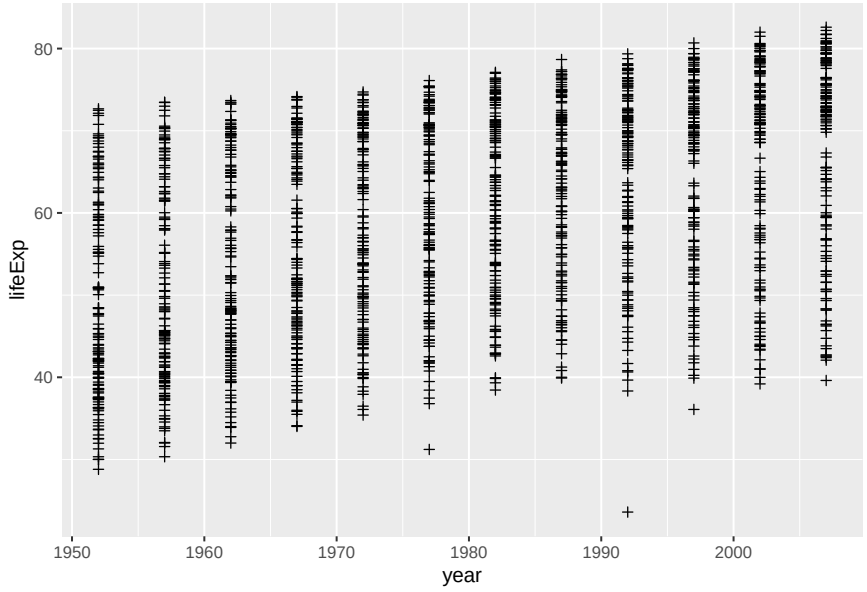
colour as blue

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp), colour = 'blue')
```



shape as plus

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp), shape = 3)
```



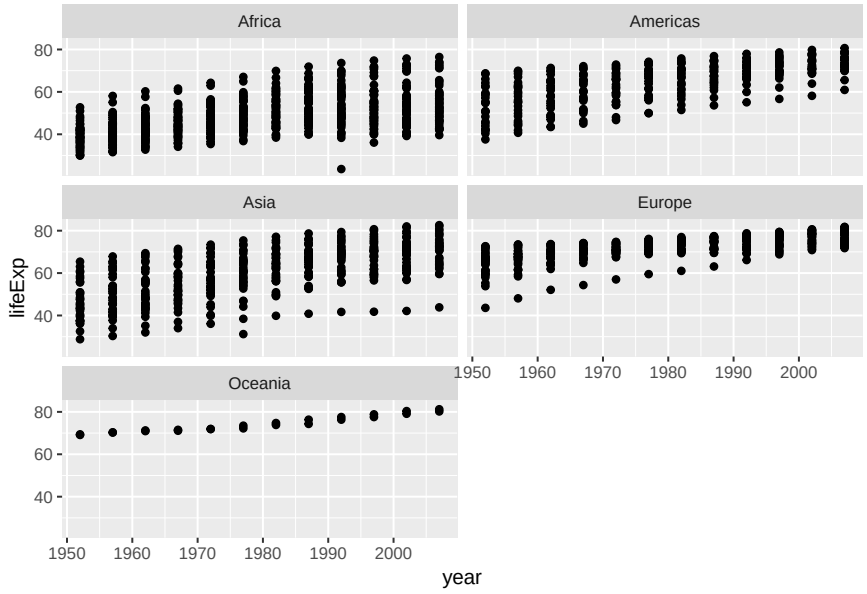
You can type ?*pch* to see the number that correspond to the shape

1.8 Making subplots

We can split our plots based on a factor variable and make subplots using the `facet()`.

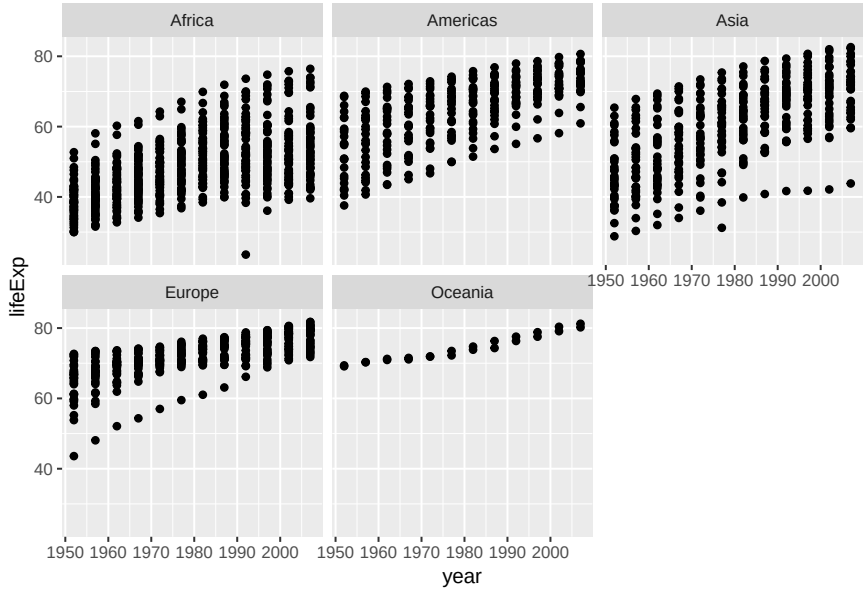
For example, if we want to make subplots based on continents, you can run these codes

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp)) +
  facet_wrap(~ continent, nrow = 3)
```



and change the nrow

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = year, y = lifeExp)) +
  facet_wrap(~ continent, nrow = 2)
```

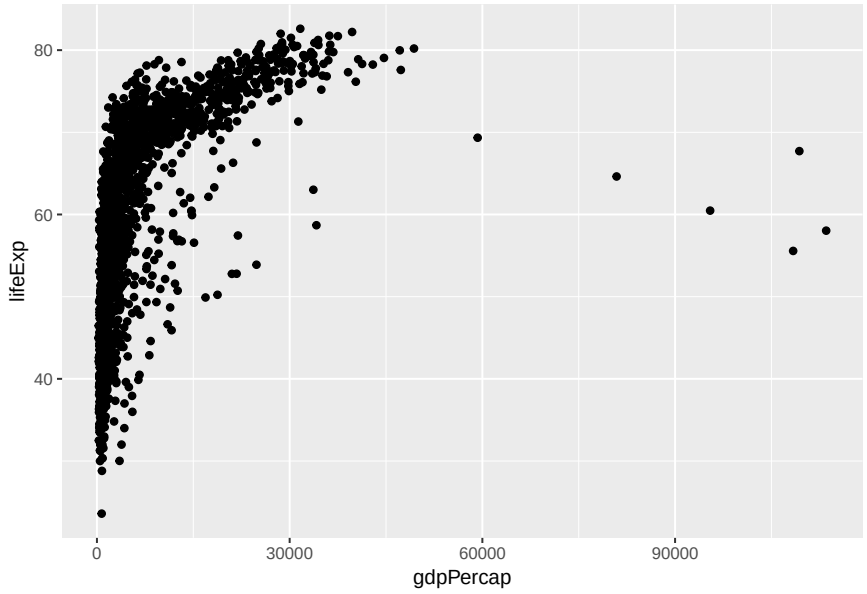


1.9 Overlaying plots

Each `geom_X()` in ggplot2 indicates different visual objects.

Scatterplot

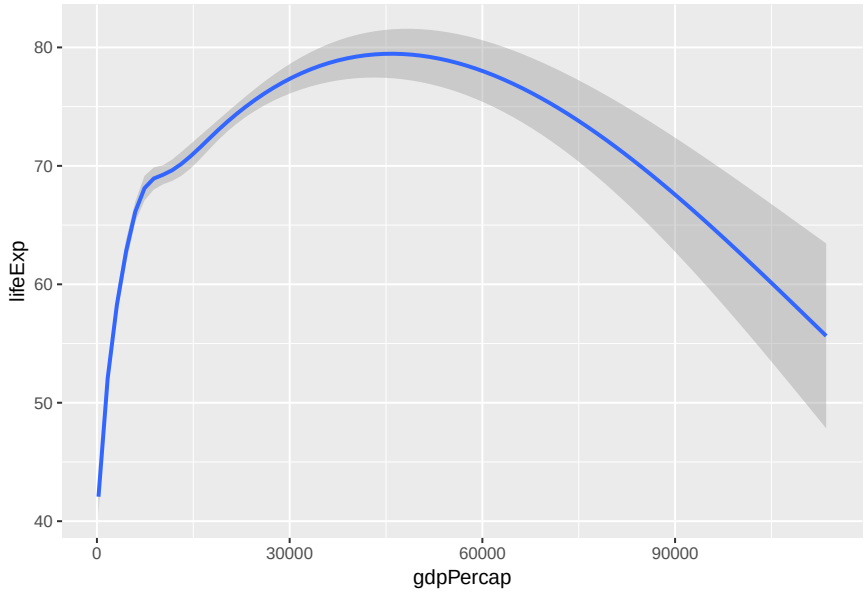
```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = gdpPercap, y = lifeExp))
```



Smooth line

```
ggplot(data = gapminder) +
  geom_smooth(mapping = aes(x = gdpPercap, y = lifeExp))
```

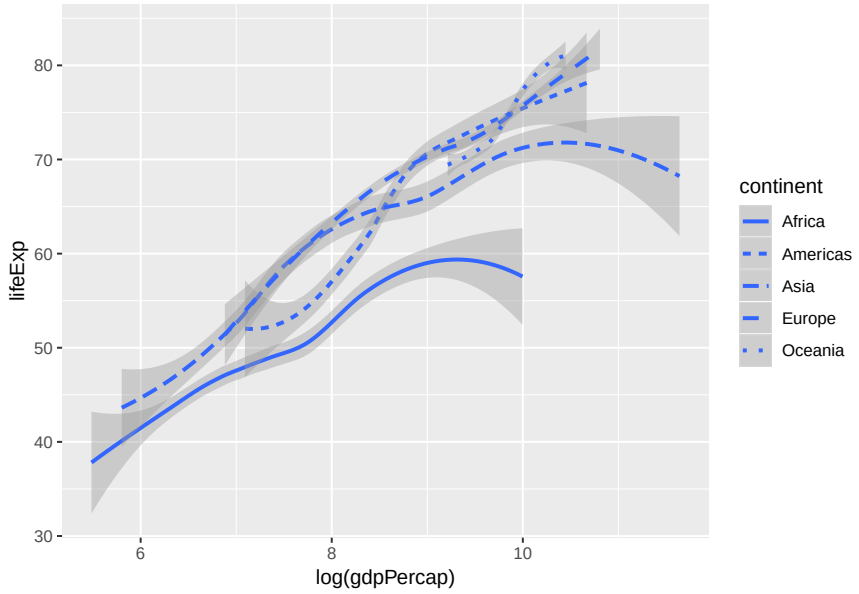
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



And we can regenerate the smooth plot based on continent using the `linetype()`. We use `log(gdpPercap)` to reduce the skewness of the data.

```
ggplot(data = gapminder) +
  geom_smooth(mapping = aes(x = log(gdpPercap), y = lifeExp, linetype = continent))
```

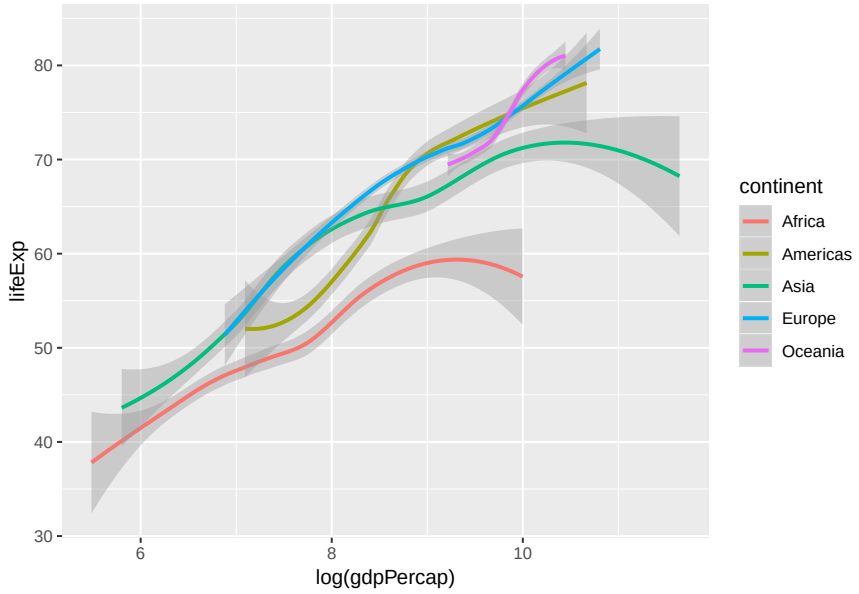
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Another plot but using colour

```
ggplot(data = gapminder) +
  geom_smooth(mapping = aes(x = log(gdpPercap), y = lifeExp, colour = continent))
```

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

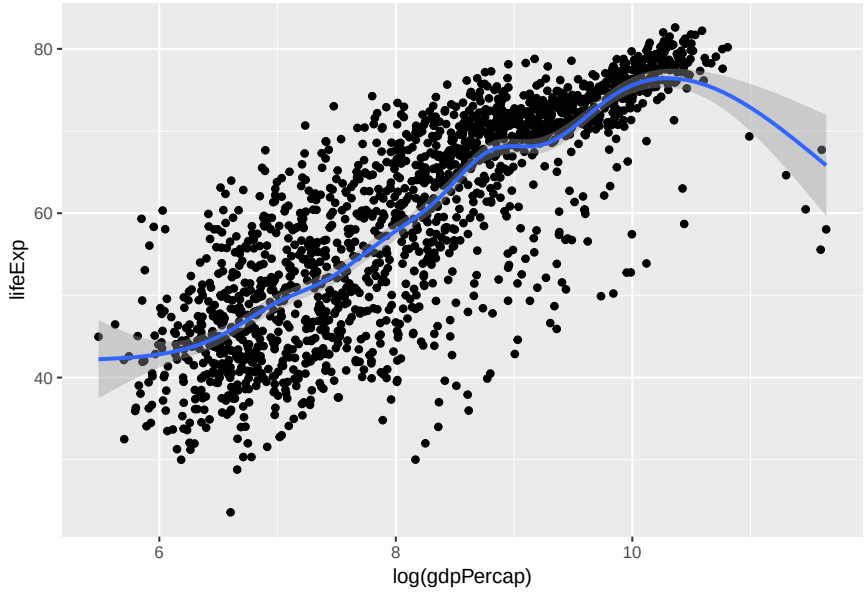


1.10 Combining geom

We can combine more than one geoms to overlay plots. The trick is to use multiple geoms in a single line of R code

```
ggplot(data = gapminder) +
  geom_point(mapping = aes(x = log(gdpPercap), y = lifeExp)) +
  geom_smooth(mapping = aes(x = log(gdpPercap), y = lifeExp))
```

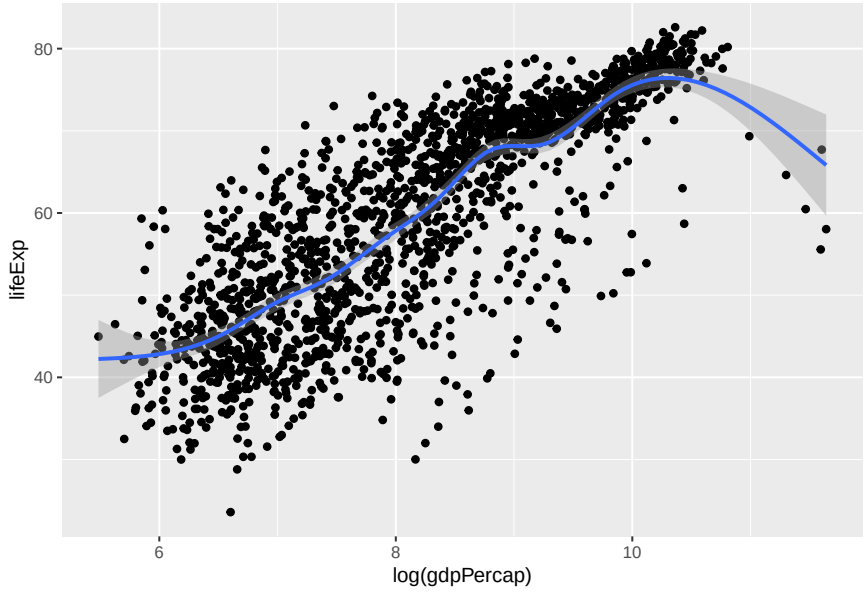
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



The codes above show duplication or repetition. To avoid this, we can pass the mapping to `ggplot()`.

```
ggplot(data = gapminder, mapping = aes(x = log(gdpPercap), y = lifeExp)) +
  geom_point() +
  geom_smooth()
```

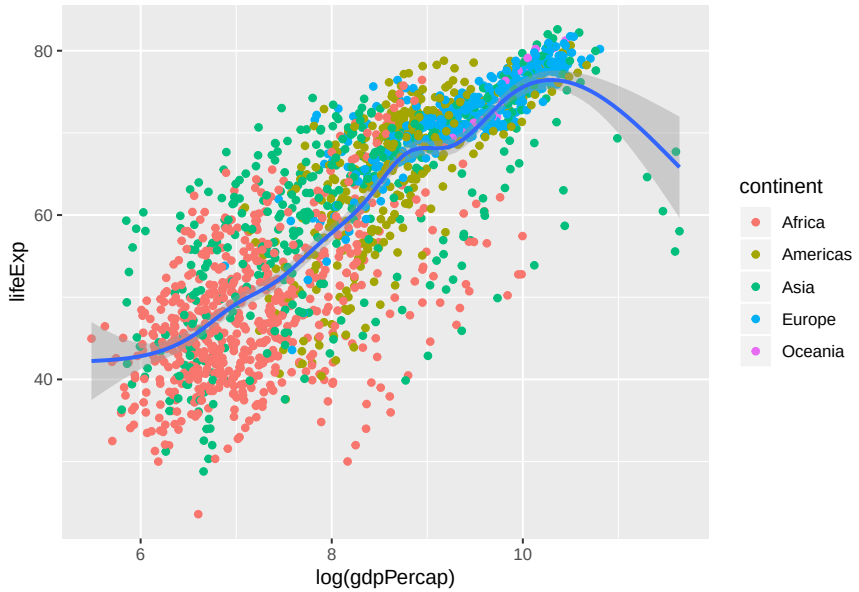
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



And we can expand this to make scatterplot shows different colour for continent

```
ggplot(data = gapminder, mapping = aes(x = log(gdpPercap), y = lifeExp)) +
  geom_point(mapping = aes(colour = continent)) +
  geom_smooth()
```

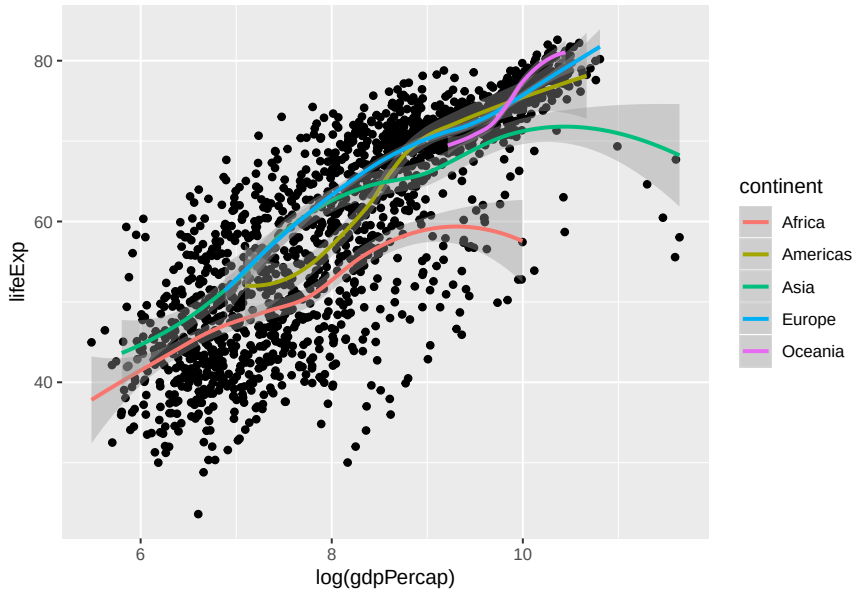
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Or expand this to make the smooth plot shows different colour for continent

```
ggplot(data = gapminder, mapping = aes(x = log(gdpPercap), y = lifeExp)) +
  geom_point() +
  geom_smooth(mapping = aes(colour = continent))
```

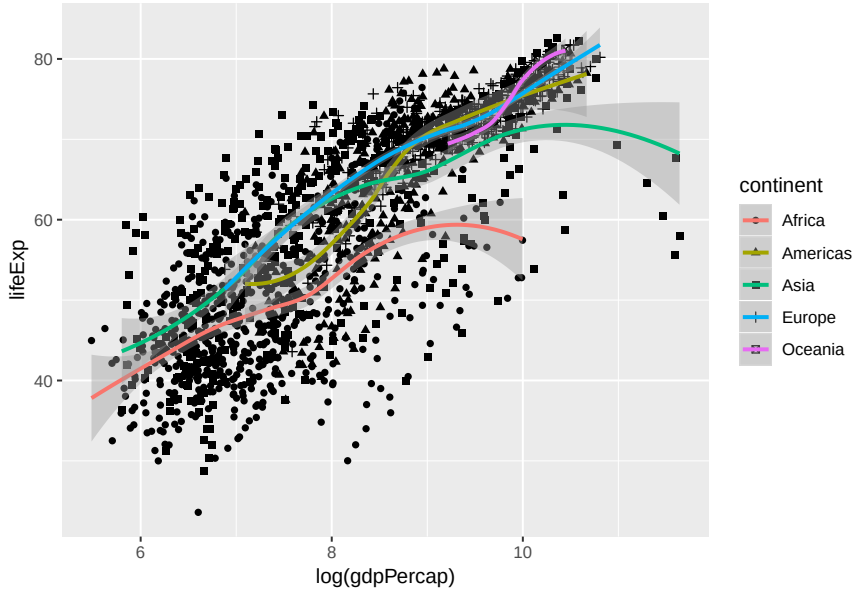
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Or both the scatterplot and the smoothplot

```
ggplot(data = gapminder, mapping = aes(x = log(gdpPercap), y = lifeExp)) +
  geom_point(mapping = aes(shape = continent)) +
  geom_smooth(mapping = aes(colour = continent))
```

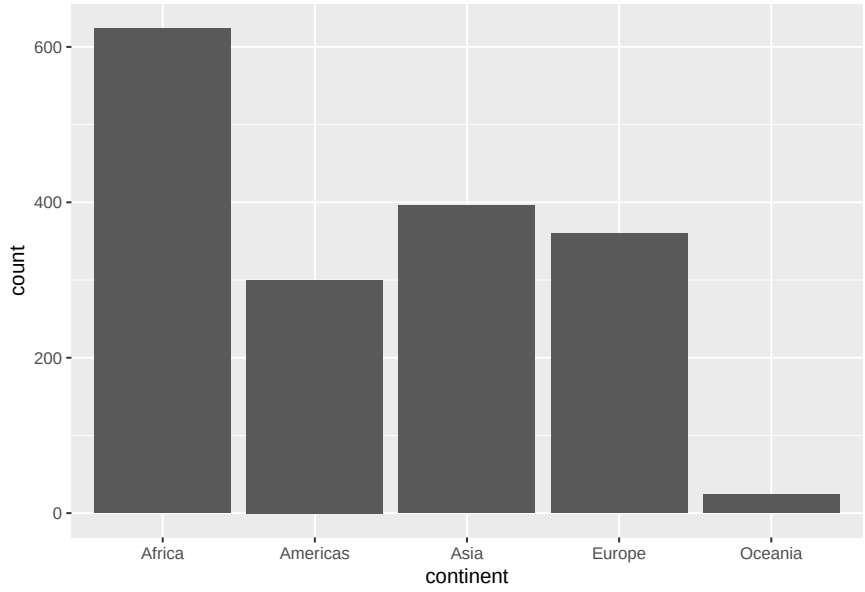
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



1.11 Statistical transformation

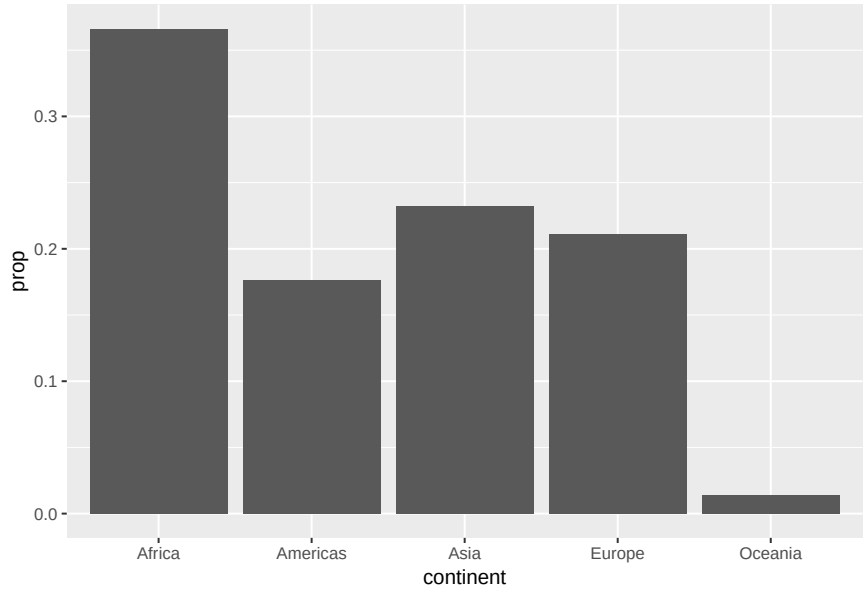
Let us create a bar chart, with y axis as the frequency.

```
ggplot(data = gapminder) +
  geom_bar(mapping = aes(x = continent))
```



If we want the y-axis to show proportion, we can use these codes

```
ggplot(data = gapminder) +
  geom_bar(mapping = aes(x = continent, y = ..prop..,
                         group = 1))
```



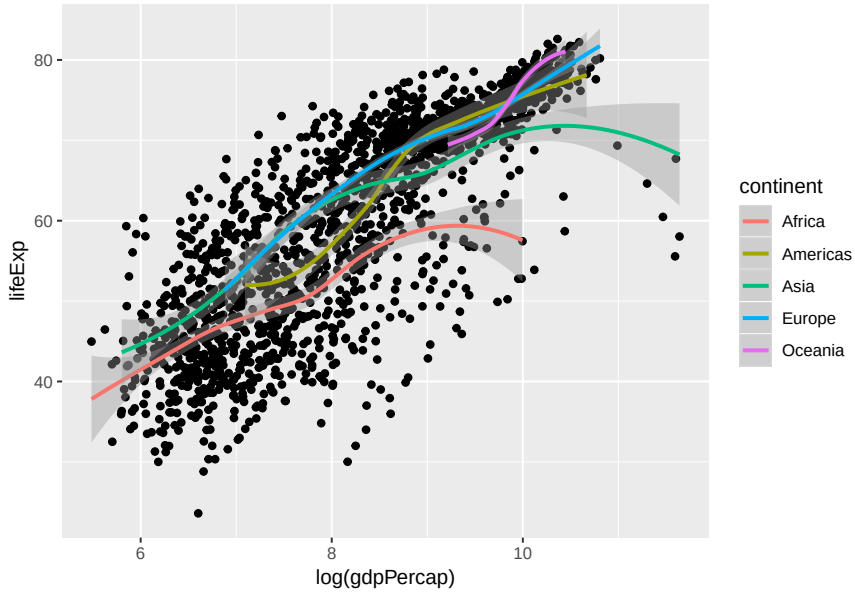
1.12 Customizing title

We can customize many aspects of the plot using ggplot package.

For example, from gapminder dataset, we choose GDP and log it (to reduce skewness) and life expectancy, and make a scatterplot. We named the plot as `my_pop`

```
mypop <- ggplot(data = gapminder, mapping = aes(x = log(gdpPercap), y = lifeExp)) +
  geom_point() +
  geom_smooth(mapping = aes(colour = continent))
mypop
```

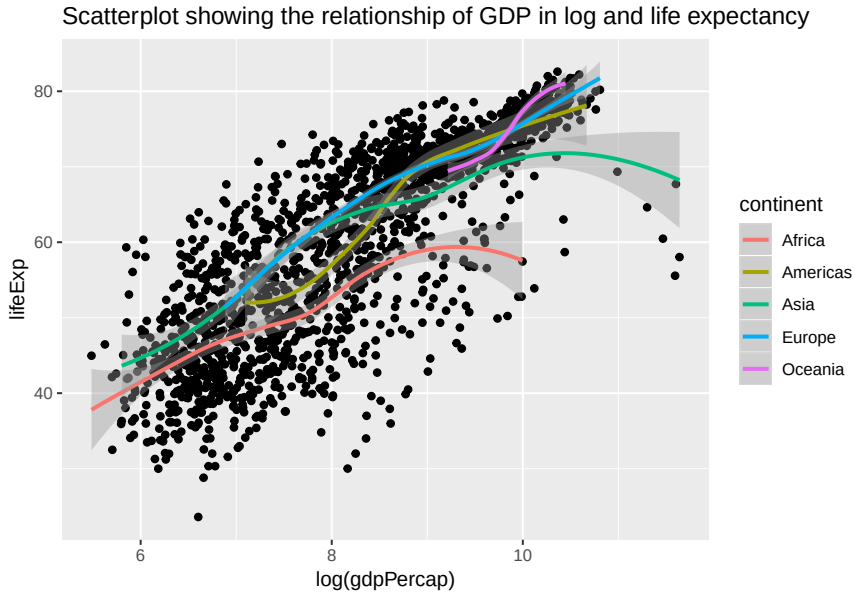
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



You will notice that there is no title in the plot. A title can be added to the plot.

```
mypop + ggtitle("Scatterplot showing the relationship of GDP in log and life expectancy")
```

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



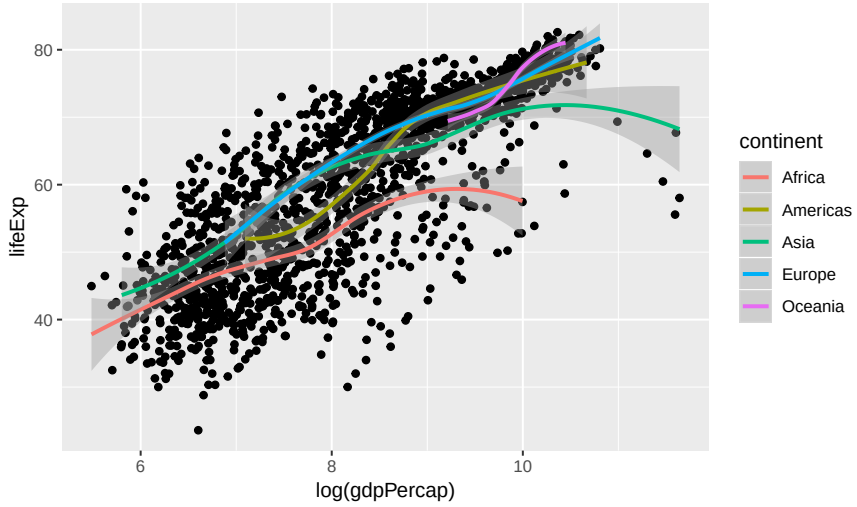
Title in multiple lines by adding `\n`

```
mypop + ggtitle("Scatterplot showing the relationship of GDP in log and life expectancy:
                \nData from Gapminder")
```

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

Scatterplot showing the relationship of GDP in log and life expectancy:

Data from Gapminder



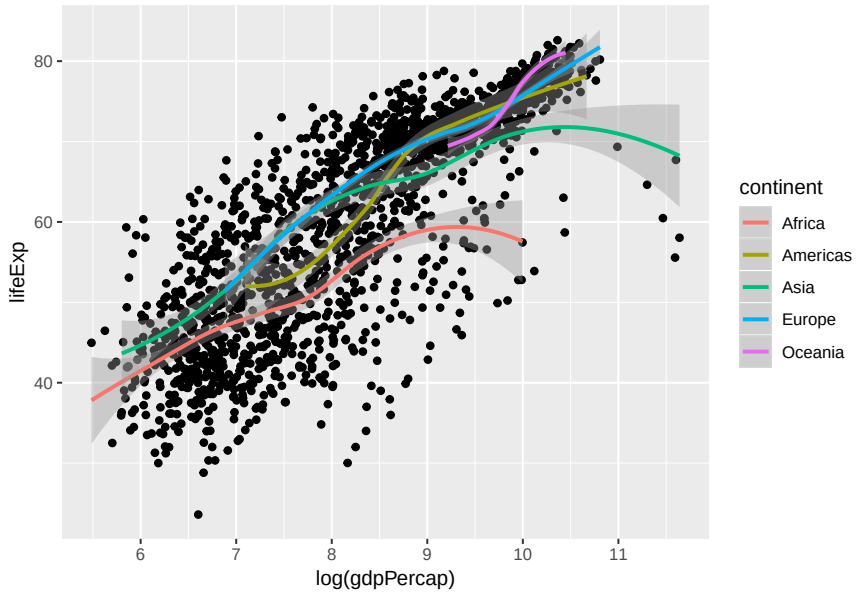
1.13 Adjusting axes

We can specify the tick marks

1. min = 0
2. max = 12
3. interval = 1

```
mypop + scale_x_continuous(breaks = seq(0,12,1))
```

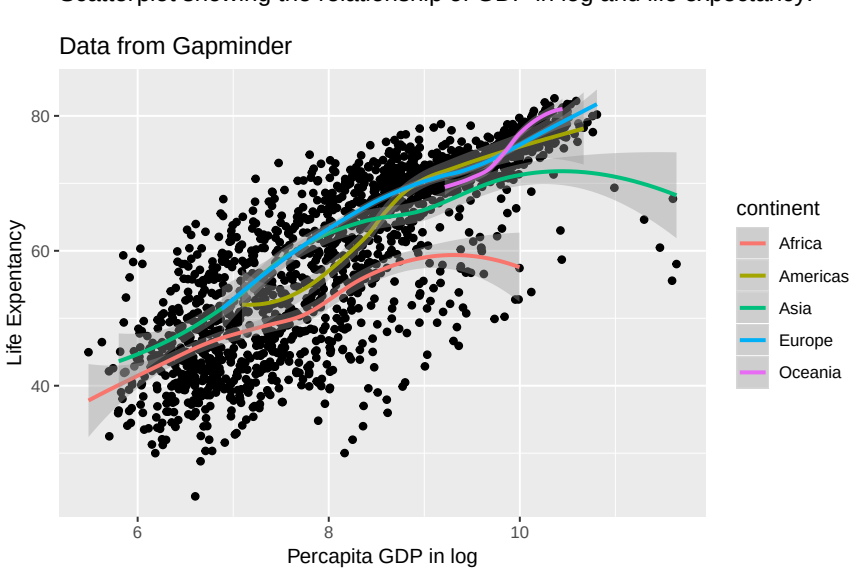
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



And we can label the x-axis and y-axis

```
mypop + ggtitle("Scatterplot showing the relationship of GDP in log and life expectancy:
                \nData from Gapminder") + ylab("Life Expentancy") + xlab("Percapita GDP in log")
```

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



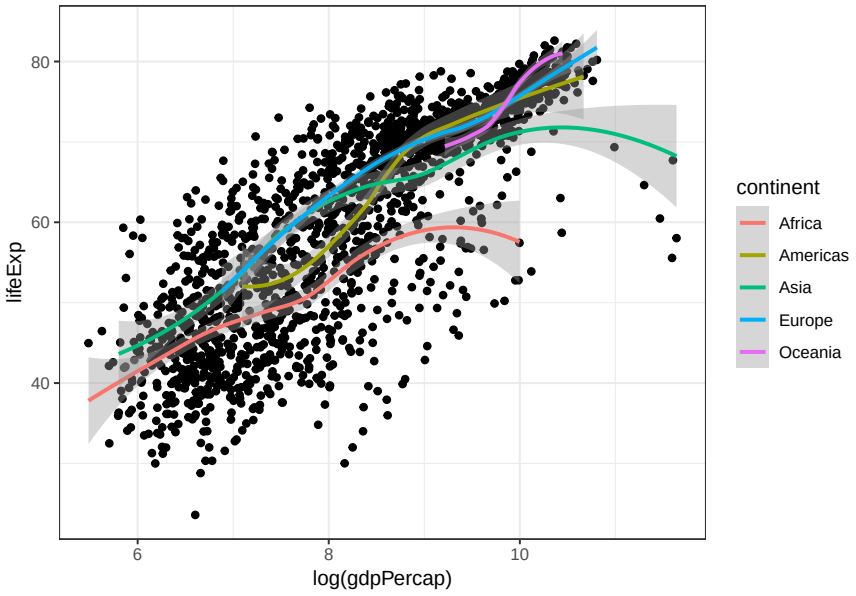
1.14 Choosing theme

The default is gray theme or `theme_gray()`

The black and white theme

```
mypop + theme_bw()
```

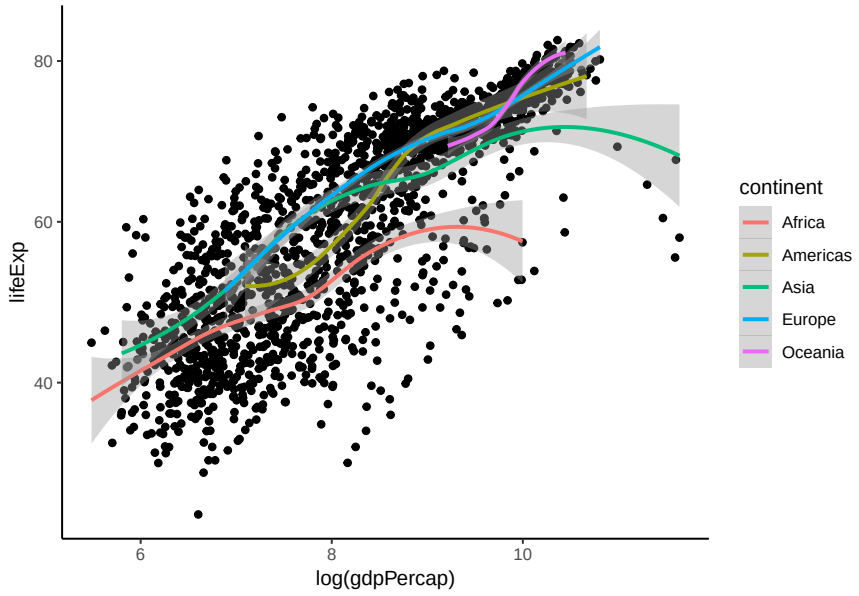
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Classic theme

```
mypop + theme_classic()
```

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



1.15 Saving plot

The preferred format for saving file is PDF.

1.16 Saving plot using ggplot2

We can save the last plot (plot on the screen) to a file.

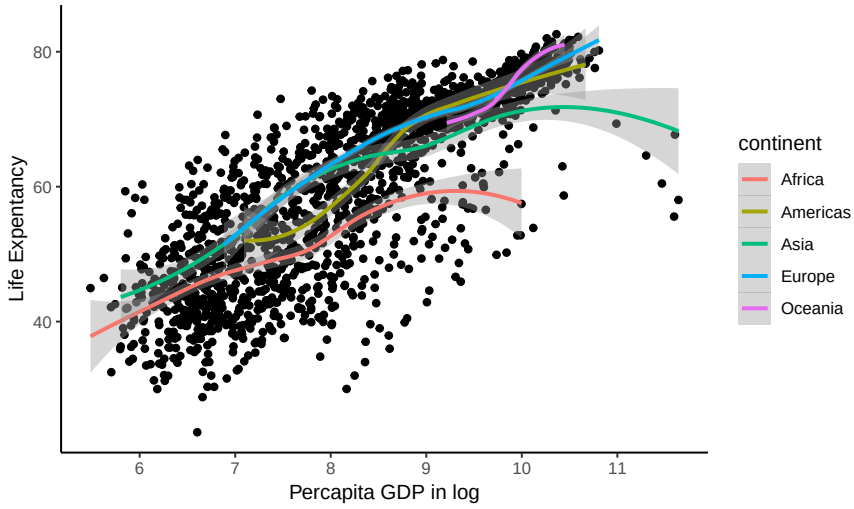
For example, let us create a more complete plot with added title, x label and y label and a classic theme

```
mypop + ggtitle("Scatterplot showing the relationship of GDP in log and life expectancy:
                \nData from Gapminder") + ylab("Life Expentancy") + xlab("Percapita GDP in log") +
  scale_x_continuous(breaks = seq(0,12,1)) +
  theme_classic()
```

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

Scatterplot showing the relationship of GDP in log and life expectancy:

Data from Gapminder



And we want to save the plot (now on the screen) to these formats:

1. pdf format
2. png format
3. jpg format

```
ggsave("my_pdf_plot.pdf")
```

```
## Saving 6.5 x 4.5 in image
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
ggsave("my_png_plot.png")
```

```
## Saving 6.5 x 4.5 in image
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
ggsave("my_jpg_plot.jpg")
```

```
## Saving 6.5 x 4.5 in image
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

We can customize the

1. width = 10 cm
2. height = 6 cm
3. dpi = 100

```
ggsave("my_pdf_plot2.pdf", width = 10, height = 6, units = "cm")
```

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

```
ggsave("my_png_plot2.png", width = 10, height = 6, units = "cm")
```

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

```
ggsave("my_jpg_plot2.jpg", width = 10, height = 6, units = "cm")
```

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

2 Data transformation

2.1 Definition of data transformation

Data transformation is also known as Data Munging or Data Wrangling. It is loosely the process of manually converting or mapping data from one "raw" form into another format. The process allows for more convenient consumption of the data. In doing so, we will be using semi-automated tools in RStudio.

For more information, please refer https://community.modeanalytics.com/sql/tutorial/data-wrangling-with-sql/

2.2 Data transformation with dplyr package

2.2.1 dplyr package

dplyr is a package grouped inside **tidyverse** collection of packages. **dplyr** package is a very useful package to munge or wrangle or to tranform your data. It is a grammar of data manipulation. It provides a consistent set of verbs that help you solve the most common data manipulation challenges

For more information, please read https://github.com/tidyverse/dplyr

2.3 Common procedures for doing data transformation

When we begin to work with data, common procedures include transforming variables in the dataset.

The common procedures that data analyst does include:

1. reducing the size of dataset by selecting certain variables (or columns)
2. generating new variable from existing variables
3. sorting observation of a variable
4. grouping observations based on certain criteria
5. reducing variable to groups to in order to estimate summary statistic

2.4 Some dplyr functions

For the procedures listed above, the corresponding **dplyr** functions are

1. `dplyr::select()` - to select a number of variables from a dataframe
2. `dplyr::mutate()` - to generate a new variable from existing variables
3. `dplyr::arrange()` - to sort observation of a variable
4. `dplyr::filter()` - to group observations that fulfil certain criteria
5. `dplyr::group_by()` and `dplyr::summarize()` - to reduce variable to groups in order to provide summary statistic

2.5 Create a new project or set your working directory

It is very important to ensure you know where your working directory is.

To do so, the best practice is *is to create a new project everytime you want to start new analysis with R.* To do so, create a new project by `File -> New Project`.

If you do not start with a new project, you still need to know **Where is my working directory?**.

So, I will emphasize again, every time you want to start processing your data, please make sure:

1. to use R project to work with your data or analysis
2. if you are not using R project, make sure you are inside the correct working directory. Type `getwd()` to display the active **working directory**. And to set a working directory use `setwd()`.
3. once you are know where your working directory is, you can start read or import data into your working directory. Remember, there are a number of packages you can use to read the data into R. It depends on the format of your data.

For example, we know that data format can be in:

1. SPSS (`.sav`) format,
2. Stata (`.dta`) format,
3. SAS format,
4. MS Excel (`.xlsx`) format
5. Comma-separated-values `.csv` format.
6. other formats

Three packages - **haven**, **readr** and **foreign** packages - are very useful to read or import your data into R memory.

1. **readr** provides a fast and friendly way to read rectangular data (like csv, tsv, and fwf).
2. **readxl** reads .xls and .xlsx sheets.
3. **haven** reads SPSS, Stata, and SAS data.

2.6 *starwars* data

To make life easier and to facilitate reproducibility, we will use examples available from the public domains.

We will produce and reproduce the outputs demonstrated on **tidyverse** website (https://github.com/tidyverse/dplyr).

One of the useful datasets is `starwars` dataset. The `starwars` data comes together with **dplyr** package. This original source of data is from SWAPI, the Star Wars API accessible at http://swapi.co/.

The `starwars` data is class of `tibble`. The data have:

- 87 rows (observations)
- 13 columns (variables)

Now, let us:

1. load the **tidyverse** package
2. examine the column names (variable names)

Loading **tidyverse** packages will load **dplyr** automatically. If you want to load only **dplyr**, just type `library(dplyr)`.

```
library(dplyr)
```

Take a peek at the `starwars` data

```
glimpse(starwars)
```

```
## Observations: 87
## Variables: 13
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", ...
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188...
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 8...
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "b...
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "l...
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue",...
```

```
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0...
## $ gender     <chr> "male", NA, NA, "male", "female", "male", "female",...
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alder...
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human...
## $ films      <list> [<"Revenge of the Sith", "Return of the Jedi", "Th...
## $ vehicles   <list> [<"Snowspeeder", "Imperial Speeder Bike">, <>, <>,...
## $ starships  <list> [<"X-wing", "Imperial shuttle">, <>, <>, "TIE Adva...
```

Next, we examine the first 10 observations of the data. There are 77 more rows NOT SHOWN. You can also see the types of the variables:

1. `chr` (character),
2. `int` (integer),
3. `dbl` (double)

```
starwars
```

```
## # A tibble: 87 x 13
##    name  height  mass hair_color skin_color eye_color birth_year gender
##    <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>
##  1 Luke~    172    77 blond      fair       blue            19   male
##  2 C-3PO    167    75 <NA>       gold       yellow         112   <NA>
##  3 R2-D2     96    32 <NA>       white, bl~ red             33   <NA>
##  4 Dart~    202   136 none       white      yellow          41.9 male
##  5 Leia~    150    49 brown      light      brown           19   female
##  6 Owen~    178   120 brown, gr~ light      blue            52   male
##  7 Beru~    165    75 brown      light      blue            47   female
##  8 R5-D4     97    32 <NA>       white, red red             NA   <NA>
##  9 Bigg~    183    84 black      light      brown           24   male
## 10 Obi-~    182    77 auburn, w~ fair       blue-gray       57   male
## # ... with 77 more rows, and 5 more variables: homeworld <chr>,
## #   species <chr>, films <list>, vehicles <list>, starships <list>
```

2.7 `dplyr::select()` , `dplyr::mutate()` and `dplyr::rename()`

2.7.1 `dplyr::select()`

When you work with large datasets with many columns, sometimes it is easier to select only the necessary columns to reduce the size of dataset.

This is possible by creating a smaller dataset (less variables). Then you can work on at the initial part of data analysis with this smaller dataset. This will greatly help data exploration.

To create smaller datasets, select some of the columns (variables) in the dataset.

In `starwars` data, we have 13 variables. From this dataset, let us generate a new dataset named as `mysw` with only these 4 variables ,

1. name
2. height
3. mass
4. gender

```
mysw <- starwars %>% select(name, gender, height, mass)
mysw
```

```
## # A tibble: 87 x 4
##    name               gender height  mass
```

```
##    <chr>              <chr>   <int> <dbl>
##  1 Luke Skywalker     male      172    77
##  2 C-3PO              <NA>      167    75
##  3 R2-D2              <NA>       96    32
##  4 Darth Vader        male      202   136
##  5 Leia Organa        female    150    49
##  6 Owen Lars          male      178   120
##  7 Beru Whitesun lars female    165    75
##  8 R5-D4              <NA>       97    32
##  9 Biggs Darklighter  male      183    84
## 10 Obi-Wan Kenobi     male      182    77
## # ... with 77 more rows
```

The new dataset `mysw` is now created. You can see it in the `Environment` pane.

2.7.2 `dplyr::mutate()`

With `dplyr::mutate()`, you can generate new variable.

For example, in the dataset `mysw`, we want to create a new variable named as `bmi`. This variable equals mass in kg divided by squared height (in meter)

$$bmi = \frac{kg}{m^2}$$

```
mysw <- mysw %>% mutate(bmi = mass/(height/100)^2)
mysw
```

```
## # A tibble: 87 x 5
##    name               gender height  mass   bmi
##    <chr>              <chr>   <int> <dbl> <dbl>
##  1 Luke Skywalker     male      172    77  26.0
##  2 C-3PO              <NA>      167    75  26.9
##  3 R2-D2              <NA>       96    32  34.7
##  4 Darth Vader        male      202   136  33.3
##  5 Leia Organa        female    150    49  21.8
##  6 Owen Lars          male      178   120  37.9
##  7 Beru Whitesun lars female    165    75  27.5
##  8 R5-D4              <NA>       97    32  34.0
##  9 Biggs Darklighter  male      183    84  25.1
## 10 Obi-Wan Kenobi     male      182    77  23.2
## # ... with 77 more rows
```

Now, your dataset `mysw` has 5 columns (variables). The last variable is `bmi`

2.7.3 `dplyr::rename()`

Now,

1. create a new variable *bmi2* which equals bmi^2.
2. rename *bmi2* to *bmisq*

```
mysw <- mysw %>% mutate(bmi2 = bmi^2)
mysw
```

```
## # A tibble: 87 x 6
##    name               gender height  mass   bmi  bmi2
##    <chr>              <chr>   <int> <dbl> <dbl> <dbl>
##  1 Luke Skywalker     male      172    77  26.0  677.
##  2 C-3PO              <NA>      167    75  26.9  723.
##  3 R2-D2              <NA>       96    32  34.7 1206.
##  4 Darth Vader        male      202   136  33.3 1111.
##  5 Leia Organa        female    150    49  21.8  474.
##  6 Owen Lars          male      178   120  37.9 1434.
##  7 Beru Whitesun lars female    165    75  27.5  759.
##  8 R5-D4              <NA>       97    32  34.0 1157.
##  9 Biggs Darklighter  male      183    84  25.1  629.
## 10 Obi-Wan Kenobi     male      182    77  23.2  540.
## # ... with 77 more rows
```

```
mysw <- mysw %>% rename(bmisq = bmi2)
mysw
```

```
## # A tibble: 87 x 6
##    name               gender height  mass   bmi bmisq
##    <chr>              <chr>   <int> <dbl> <dbl> <dbl>
##  1 Luke Skywalker     male      172    77  26.0  677.
##  2 C-3PO              <NA>      167    75  26.9  723.
##  3 R2-D2              <NA>       96    32  34.7 1206.
##  4 Darth Vader        male      202   136  33.3 1111.
##  5 Leia Organa        female    150    49  21.8  474.
##  6 Owen Lars          male      178   120  37.9 1434.
##  7 Beru Whitesun lars female    165    75  27.5  759.
##  8 R5-D4              <NA>       97    32  34.0 1157.
##  9 Biggs Darklighter  male      183    84  25.1  629.
## 10 Obi-Wan Kenobi     male      182    77  23.2  540.
## # ... with 77 more rows
```

2.8 dplyr::arrange() and dplyr::filter()

2.8.1 dplyr::arrange()

We can sort data in ascending or descending order.

To do that, we will use `dplyr::arrange()`. It will sort the observation based on the values of the specified variable.

For dataset `mysw`, let us sort the `bmi` from the biggest bmi (descending).

```
mysw <- mysw %>% arrange(desc(bmi))
mysw
```

```
## # A tibble: 87 x 6
##    name                  gender        height  mass   bmi   bmisq
##    <chr>                 <chr>          <int> <dbl> <dbl>   <dbl>
##  1 Jabba Desilijic Tiure hermaphrodite    175  1358 443.  196629.
##  2 Dud Bolt              male              94    45  50.9   2594.
##  3 Yoda                  male              66    17  39.0   1523.
##  4 Owen Lars             male             178   120  37.9   1434.
##  5 IG-88                 none             200   140  35     1225
```

```
##  6 R2-D2                <NA>             96    32  34.7   1206.
##  7 Grievous             male            216   159  34.1   1161.
##  8 R5-D4                <NA>             97    32  34.0   1157.
##  9 Jek Tono Porkins     male            180   110  34.0   1153.
## 10 Darth Vader          male            202   136  33.3   1111.
## # ... with 77 more rows
```

Now, we will replace the dataset `mysw` with data that contain the `bmi` values from the lowest to the biggest bmi (ascending).

```
mysw <- mysw %>% arrange(bmi)
mysw
```

```
## # A tibble: 87 x 6
##    name          gender height  mass   bmi bmisq
##    <chr>         <chr>   <int> <dbl> <dbl> <dbl>
##  1 Wat Tambor    male      193    48  12.9  166.
##  2 Adi Gallia    female    184    50  14.8  218.
##  3 Sly Moore     female    178    48  15.1  230.
##  4 Roos Tarpals  male      224    82  16.3  267.
##  5 Padmé Amidala female    165    45  16.5  273.
##  6 Lama Su       male      229    88  16.8  282.
##  7 Jar Jar Binks male      196    66  17.2  295.
##  8 Ayla Secura   female    178    55  17.4  301.
##  9 Shaak Ti      female    178    57  18.0  324.
## 10 Barriss Offee female    166    50  18.1  329.
## # ... with 77 more rows
```

2.8.2 `dplyr::filter()`

To select observations based on certain criteria, we use the `dplyr::filter()` function.

Here, we will create a new dataset (which we will name as `mysw_m_40`) that contains observations with these criteria:

- gender is male AND
- bmi at or above 40

```
mysw_m_40 <- mysw %>% filter(gender == 'male', bmi >= 40)
mysw_m_40
```

```
## # A tibble: 1 x 6
##   name     gender height  mass   bmi bmisq
##   <chr>    <chr>   <int> <dbl> <dbl> <dbl>
## 1 Dud Bolt male       94    45  50.9 2594.
```

Next, we will create a new dataset (named as `mysw_ht_45`) that contain

- `height` above 200 OR `BMI` above 45, AND
- does not include `NA` (which is missing value) observation for `bmi`

```
mysw_ht_45 <- mysw %>% filter(height >200 | bmi >45, bmi != 'NA')
mysw_ht_45
```

```
## # A tibble: 9 x 6
##   name                gender        height  mass   bmi   bmisq
##   <chr>               <chr>          <int> <dbl> <dbl>   <dbl>
## 1 Roos Tarpals        male             224    82  16.3    267.
```

```
## 2 Lama Su               male             229    88  16.8    282.
## 3 Tion Medon            male             206    80  18.9    355.
## 4 Chewbacca             male             228   112  21.5    464.
## 5 Tarfful               male             234   136  24.8    617.
## 6 Darth Vader           male             202   136  33.3   1111.
## 7 Grievous              male             216   159  34.1   1161.
## 8 Dud Bolt              male              94    45  50.9   2594.
## 9 Jabba Desilijic Tiure hermaphrodite    175  1358 443.  196629.
```

2.9 dplyr::group_by() and dplyr::summarize

2.9.1 dplyr::group_by()

The group_by function will prepare the data for group analysis.

For example,

1. to get summary values for mean bmi, mean ht and mean mass
2. for male, female, hermaphrodite and none (gender variable)

```
mysw_g <- mysw %>% group_by(gender)
mysw_g
```

```
## # A tibble: 87 x 6
## # Groups:   gender [5]
##    name          gender height  mass   bmi bmisq
##    <chr>         <chr>   <int> <dbl> <dbl> <dbl>
##  1 Wat Tambor    male      193    48  12.9  166.
##  2 Adi Gallia    female    184    50  14.8  218.
##  3 Sly Moore     female    178    48  15.1  230.
##  4 Roos Tarpals  male      224    82  16.3  267.
##  5 Padmé Amidala female    165    45  16.5  273.
##  6 Lama Su       male      229    88  16.8  282.
##  7 Jar Jar Binks male      196    66  17.2  295.
##  8 Ayla Secura   female    178    55  17.4  301.
##  9 Shaak Ti      female    178    57  18.0  324.
## 10 Barriss Offee female    166    50  18.1  329.
## # ... with 77 more rows
```

2.9.2 dplyr::summarize()

Now that we have a group data named mysw_g, now, we would summarize our data using the mean and standard deviation (SD).

```
mysw_g %>% summarise(meanbmi = mean(bmi, na.rm = TRUE),
                     meanht  = mean(height, na.rm = TRUE),
                     meanmass = mean(mass, na.rm = TRUE),
                     sdbmi = sd(bmi, na.rm = TRUE),
                     sdht = sd(height, na.rm = TRUE),
                     sdmass = sd(mass, na.rm = TRUE))
```

```
## # A tibble: 5 x 7
##   gender       meanbmi meanht meanmass  sdbmi  sdht sdmass
##   <chr>          <dbl>  <dbl>    <dbl>  <dbl> <dbl>  <dbl>
## 1 female          18.8   165.     54.0   3.71  23.0   8.37
```

```
## 2 hermaphrodite   443.    175    1358   NaN    NaN   NaN
## 3 male             25.7   179.     81.0   6.49  35.4  28.2
## 4 none             35     200     140   NaN    NaN   NaN
## 5 <NA>             31.9   120      46.3   4.33  40.7  24.8
```

To calculate the frequencies

- with one variable

```
freq_species <- starwars %>% count(species, sort = TRUE)
freq_species
```

```
## # A tibble: 38 x 2
##    species      n
##    <chr>    <int>
##  1 Human       35
##  2 Droid        5
##  3 <NA>         5
##  4 Gungan       3
##  5 Kaminoan     2
##  6 Mirialan     2
##  7 Twi'lek      2
##  8 Wookiee      2
##  9 Zabrak       2
## 10 Aleena       1
## # ... with 28 more rows
```

- with two variables

```
freq_species_home <- starwars %>% count(species, homeworld, sort = TRUE)
freq_species_home
```

```
## # A tibble: 58 x 3
##    species  homeworld     n
##    <chr>    <chr>     <int>
##  1 Human    Tatooine      8
##  2 Human    Naboo         5
##  3 Human    <NA>          5
##  4 Gungan   Naboo         3
##  5 Human    Alderaan      3
##  6 Droid    Tatooine      2
##  7 Droid    <NA>          2
##  8 Human    Corellia      2
##  9 Human    Coruscant     2
## 10 Kaminoan Kamino        2
## # ... with 48 more rows
```

2.10 More complicated dplyr verbs

To be more efficent, use multiple **dplyr** functions in one line of R code

```
starwars %>% filter(gender == "male", height > 100, mass > 100) %>%
  select(height, mass, species) %>%
  group_by(species) %>%
  summarize(mean_ht = mean(height, na.rm = TRUE),
```

```
            mean_mass = mean(mass, na.rm = TRUE),
            freq = n())
```

```
## # A tibble: 5 x 4
##   species    mean_ht mean_mass  freq
##   <chr>        <dbl>     <dbl> <int>
## 1 Besalisk      198        102     1
## 2 Human         187.       122     3
## 3 Kaleesh       216        159     1
## 4 Trandoshan    190        113     1
## 5 Wookiee       231        124     2
```

2.11 Data transformation for categorical variables

2.11.1 forcats package

Data transformation for categorical variables (factor variables) can be facilitated using the **forcats** package.

2.11.2 Create a dataset

Let us create create a dataset to demonstrate **forcats** package. The dataset will contain

1. a vector column named as **sex1** , values = 0,1
2. a vector column named as **race1** , values = 1,2,3,4
3. a tibble dataframe (dataset) named as **data_f**

```
sex1 <- rbinom(n = 100, size = 1, prob = 0.5)
str(sex1)
```

```
##  int [1:100] 1 1 1 0 1 1 1 0 1 1 ...
```

```
race1 <- rep(seq(1:4), 25)
str(race1)
```

```
##  int [1:100] 1 2 3 4 1 2 3 4 1 2 ...
```

```
data_f <- tibble(sex1, race1)
head(data_f)
```

```
## # A tibble: 6 x 2
##    sex1 race1
##   <int> <int>
## 1     1     1
## 2     1     2
## 3     1     3
## 4     0     4
## 5     1     1
## 6     1     2
```

Now let us see the structure of the dataset. You should see that they are all in the integer (numerical) format

```
str(data_f)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    100 obs. of  2 variables:
##  $ sex1 : int  1 1 1 0 1 1 1 0 1 1 ...
##  $ race1: int  1 2 3 4 1 2 3 4 1 2 ...
```

2.11.3 Conversion from numeric to factor variables

Now, we will convert the integer (numerical) variable to a factor (categorical) variable.

For example, we will generate a new factor (categorical) variable named as `male` from variable `sex1` (which is an integer variable). We will label `male`as *No* or *Yes*.

We then generate a new factor (categorical) variable named as `race2` from `race1` (which is an integer variable) and label as *Mal, Chi, Ind, Others*

```
data_f$male <- factor(data_f$sex1, labels = c('No', 'Yes'))
data_f$race2 <- factor(data_f$race1, labels = c('Mal', 'Chi', 'Ind', 'Others'))
str(data_f)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    100 obs. of  4 variables:
##  $ sex1 : int  1 1 1 0 1 1 1 0 1 1 ...
##  $ race1: int  1 2 3 4 1 2 3 4 1 2 ...
##  $ male : Factor w/ 2 levels "No","Yes": 2 2 2 1 2 2 2 1 2 2 ...
##  $ race2: Factor w/ 4 levels "Mal","Chi","Ind",..: 1 2 3 4 1 2 3 4 1 2 ...
```

```
head(data_f) ; tail(data_f)
```

```
## # A tibble: 6 x 4
##    sex1 race1 male  race2
##   <int> <int> <fct> <fct>
## 1     1     1 Yes   Mal
## 2     1     2 Yes   Chi
## 3     1     3 Yes   Ind
## 4     0     4 No    Others
## 5     1     1 Yes   Mal
## 6     1     2 Yes   Chi
```

```
## # A tibble: 6 x 4
##    sex1 race1 male  race2
##   <int> <int> <fct> <fct>
## 1     1     3 Yes   Ind
## 2     1     4 Yes   Others
## 3     0     1 No    Mal
## 4     0     2 No    Chi
## 5     1     3 Yes   Ind
## 6     0     4 No    Others
```

2.11.4 `forcats::fct_recode()`

Recode old levels to new levels

Our objectives:

1. For variable **male**, change from `No` vs `Yes` to `Fem` and `Male`
2. Create a new variable **malay** from variable **race2** and label `Chi` to `Non-Malay`, `Ind` to `Non-Malay` and `Others` to `Non-Malay`. But we keep `Mal` as it is

We will use **forcats** packages for that. Below we show two ways of recoding the variables.

```
library(forcats)
data_f$male2 <- data_f$male %>% fct_recode('Fem' = 'No', 'Male' = 'Yes')
data_f <- data_f %>% mutate(malay = fct_recode(race2,
                                               'Non-Malay' = 'Chi',
```

```
                                        'Non-Malay' = 'Ind',
                                        'Non-Malay' = 'Others'))
head(data_f) ; tail(data_f)
```

```
## # A tibble: 6 x 6
##    sex1 race1 male  race2  male2 malay
##   <int> <int> <fct> <fct>  <fct> <fct>
## 1     1     1 Yes   Mal    Male  Mal
## 2     1     2 Yes   Chi    Male  Non-Malay
## 3     1     3 Yes   Ind    Male  Non-Malay
## 4     0     4 No    Others Fem   Non-Malay
## 5     1     1 Yes   Mal    Male  Mal
## 6     1     2 Yes   Chi    Male  Non-Malay
```

```
## # A tibble: 6 x 6
##    sex1 race1 male  race2  male2 malay
##   <int> <int> <fct> <fct>  <fct> <fct>
## 1     1     3 Yes   Ind    Male  Non-Malay
## 2     1     4 Yes   Others Male  Non-Malay
## 3     0     1 No    Mal    Fem   Mal
## 4     0     2 No    Chi    Fem   Non-Malay
## 5     1     3 Yes   Ind    Male  Non-Malay
## 6     0     4 No    Others Fem   Non-Malay
```

2.12 Summary

dplyr package is a very useful package that encourages users to use proper verb when manipulating variables (columns) and observations (rows).

We have learned to use 5 functions but there are more functions available. Other useful functions include:

1. `dplyr::distinct()`
2. `dplyr::transmutate()`
3. `dplyr::sample_n()` and `dplyr::sample_frac()`

Also note that, package **dplyr** is very useful when it is combined with another function that is `group_by`

If you working with database, you can use **dbplyr** which has been developed to perform very effectively with databases.

For categorical variables, you can use **forcats** package.

2.13 Self-practice

If you have completed the tutorial above, you may:

1. Read your own data (hints: **haven**, **foreign**) or you can download data from https://www.kaggle.com/datasets . Maybe can try this dataset https://www.kaggle.com/blastchar/telco-customer-churn
2. Create a smaller dataset by selecting some variable (hints: `dplyr::select()`)
3. Creating a dataset with some selection (hints: `dplyr::filter()`)
4. Generate a new variable (hints`dplyr::mutate()`)
5. Creata an object using pipe and combining `dplyr::select()`, `dplyr::filter()` and `dplyr::mutate()` in one single line of R code
6. Summarise the mean, standard deviation and median for numerical variables `dplyr::group_by()` and `dplyr::summarize()`
7. Calculare the number of observations for categorical variables (hints: `dplyr::count()`)

8. Recode a categorical variable (hints: `forcats::fct_recode()`)

2.14 References

1. dplyr vignettes here https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html
2. forcats examples here http://r4ds.had.co.nz/factors.html
3. reading data into R https://garthtarr.github.io/meatR/rio.html