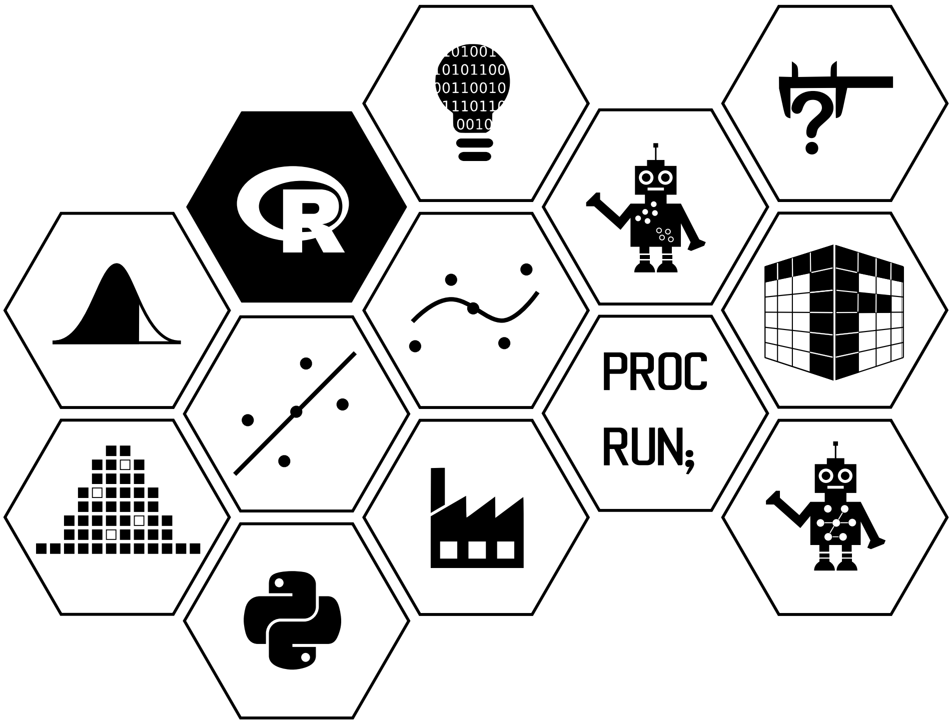
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

Week 6:

Advanced R Graphics using ggplot2



Overview

An overview of ggplot2

https://youtu.be/aiTVeohI8Ec

Duration: 14m0s

The package ggplot2 provides an abstract and declarative environment for creating graphics.

The graphics system built into R is already quite powerful and flexible, but creating sophisticated graphics can be time-consuming and many steps that could be performed automatically, like adding a legend, have to be performed manually. Code producing more complex visualisations tends be "procedural": rather than describing how the visualisation should look like, the code describes the detailed control flow of how the plot is constructed.

`ggplot2` aims like the other packages in the tidyverse, and also like languages such as SQL, to be declarative: your code should just describe what the plot should look like, and not how it is being put together in a detailed step-by-step manner.

`ggplot2` has become by far the post popular R package for graphics, with many extension packages being available. `ggplot2` has been ported to other languages and environments, such as Python or Julia.

R Graph Gallery

http://www.r-graph-gallery.com/portfolio/ggplot2-package/

The R Graph Gallery has a section entirely dedicated to `ggplot2`.

ggplot terms

This section gives an overview of key terms in the `ggplot2` world. `ggplot2` is based on the philosophy of a "layered grammar of graphics": plots in `ggplot2` are made up of at least one layer of geometric objects.

Tidy data

http://vita.had.co.nz/papers/layered-grammar.pdf

Wickham, H. (2010). A Layered Grammar of Graphics. Journal of Computational and Graphical Statistics. Volume 19, Number 1.

This paper explains some of the philosophy behind `ggplot2` (as seen in Week 4).

Geometric objects A geometric object (or `geom_<type>(...)` in `ggplot2` commands) controls what type of plot a layer contains. The are many different geometric objects. The most important ones are (see the HTML version of the notes for some additional aesthetic features for each geometry):

Geometry name	Description	Basic R equivalent
`geom_point`	Points (scatter plot)	`plot` / `points`
`geom_line`	Lines (drawn left to right)	`lines` (after ordering)
`geom_path`	Lines (drawn in original order)	`lines`
`geom_abline`	Line (one line)	`abline`
`geom_hline`	Horizontal line	`abline`
`geom_vline`	Vertical line	`abline`
`geom_text`	Text	`text`

Geometry name	Description	Basic R equivalent
`geom_label`	Text (styled as label)	`text`
`geom_rect`	Rectangle	`rect`
`geom_polygon`	Polygon	`polygon`
`geom_ribbon`	Ribbon (for confidence bands)	-
`geom_bar`	Bar plot	`barplot`
`geom_boxplot`	Boxplot	`boxplot`
`geom_histogram`	Histogram	`hist`
`geom_raster`/ `geom_tile`	Image plot	`image`
`geom_contour`	Contour lines	`contour`

There is a cheat sheet providing a detailed overview of the different geometries and data.

Aesthetics An aesthetic (or `aes(...)` in `ggplot2` commands) controls which variables are mapped to which properties of the geometric objects (like x-coordinates, y-coordinates, colours, etc.). The aesthetics available depend on the geometric object. Aesthetics commonly available are:

Aesthetic	Description
`x`	x-coordinate
`y`	y-coordinate
`color` or `colour`	Colour (outline)
`fill`	Fill colour
`alpha`	Transparency (transparent $0 \leq \alpha \leq 1$ opaque)
`linetype`	Line type ("`lty`")
`symbol`	Plotting symbol ("`pch`")
`size`	Size of plotting symbol / font or line thickness

The help file for each geometry lists the available aesthetics.

Data Visualisation Cheat Sheet

https://github.com/rstudio/cheatsheets/blob/main/data-visualization-2.1.pdf

Rstudio have put together a very handy and compact cheat sheet for `ggplot2`.

Background reading: Chapter 3 of R for Data Science

http://r4ds.had.co.nz/data-visualisation.html

Chapter 3 of *R for Data Science* gives an introduction to data visualisation using `ggplot2`.

Background reading: Chapter 28 of R for Data Science

http://r4ds.had.co.nz/graphics-for-communication.html

Chapter 28 of *R for Data Science* focuses on the presentation of visual information (with a focus on `ggplot2`).

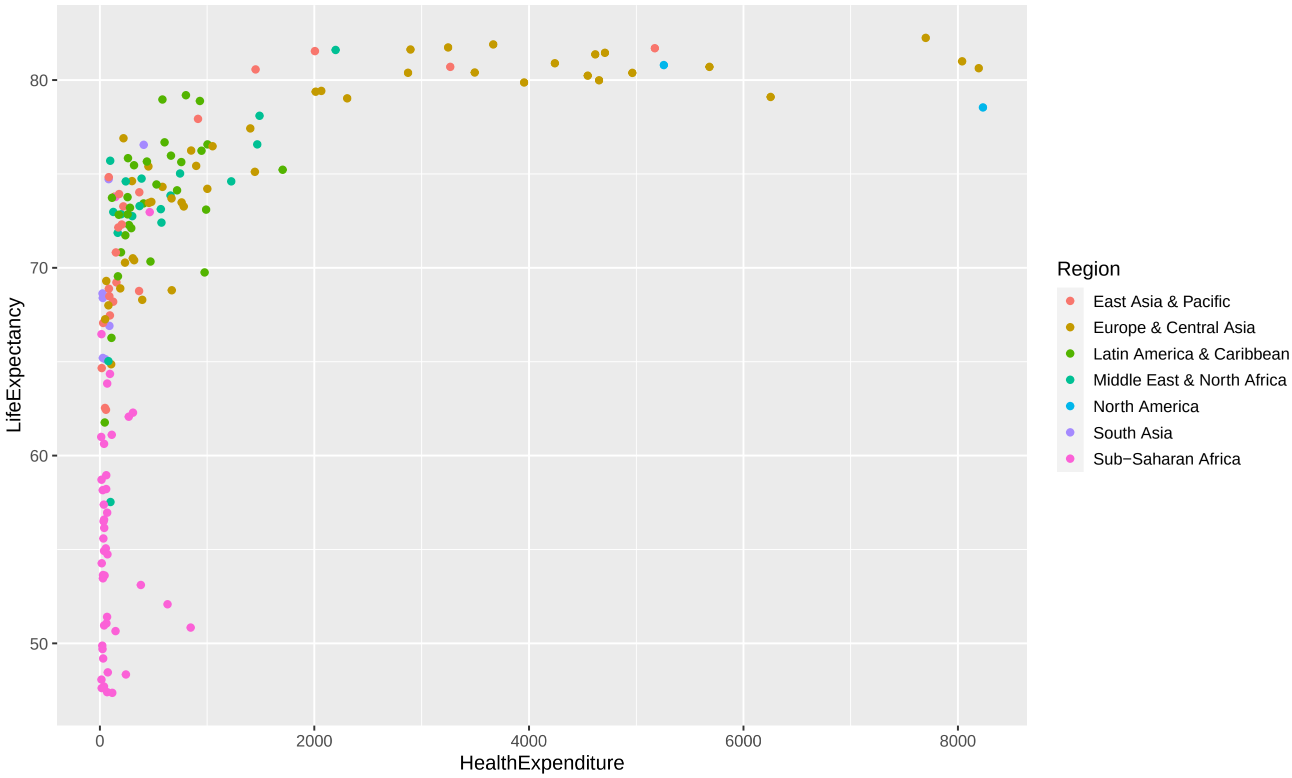
Quick plots

The function `qplot` (or `quickplot`) provides a compact interface for simple `ggplot2` graphics. Its syntax is meant to resemble the syntax of `plot` in basic R.

The basic syntax of `qplot` is `qplot(x, y, data=data, geom=geom, ...)`. It plots `y` against `x` (taken from `data`) using the geometry `geom`. `geom` is specified as a string and without `geom_` (for example `geom="line"` instead of `geom_line`). `qplot` also accepts the optional arguments `log`, `main`, `sub`, `xlab`, `ylab`, `xlim` and `ylim`, which have similar effects as the arguments of that name have for `plot` in standard R graphics.

We can re-create the plot of life expectancy against health expenditure from last week using

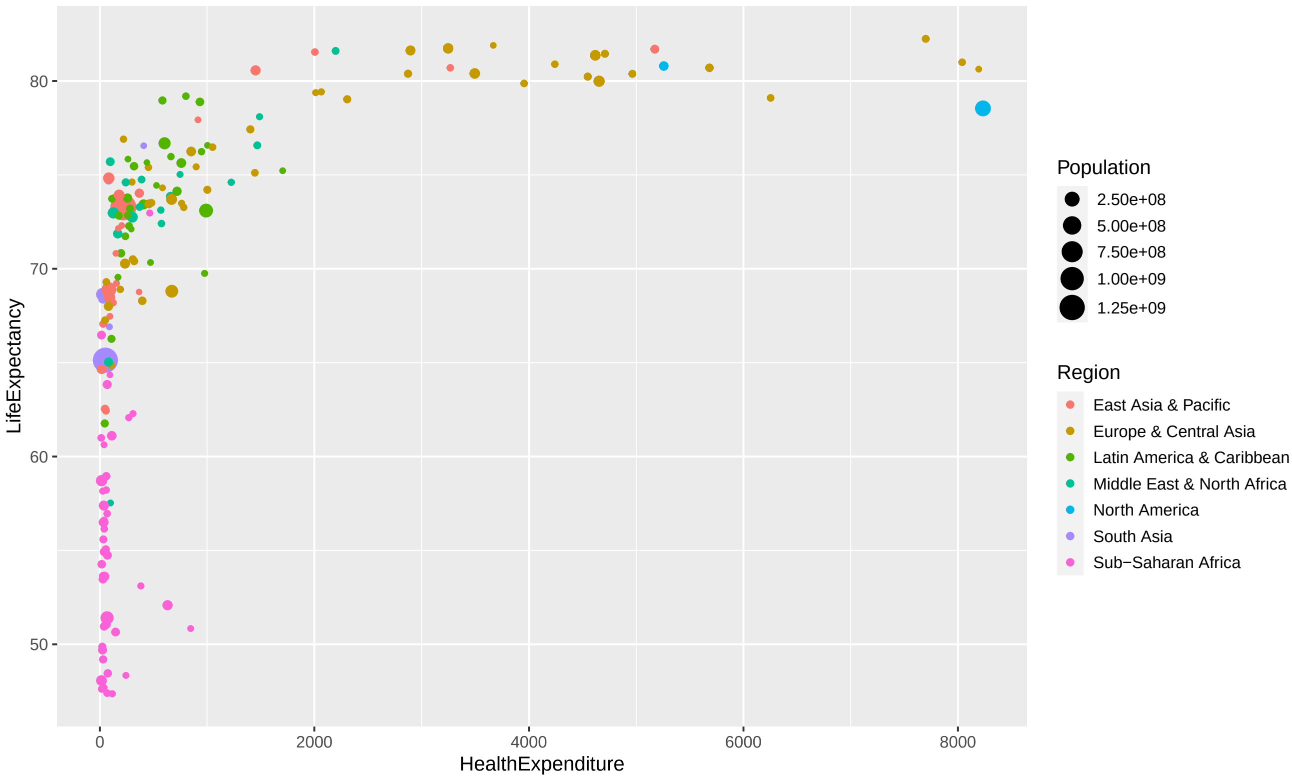
```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%205/w5.RData"))
qplot(HealthExpenditure, LifeExpectancy,
      data=health, colour=Region)
```



We can already see a major advantage of using `ggplot2`: we don't need to `unclass` Region and `ggplot2` has already drawn a legend for us.

`ggplot2` also allows for a graphical parameter `size`, which controls the size of the plotting symbol (on a square-root scale, so that the area of the plotting symbol is on a linear scale.)

```
qplot(HealthExpenditure, LifeExpectancy,
      data=health, colour=Region, size=Population)
```



Task 1.

In this task we will use the data set `diamonds` from `ggplot2`. Create a scatter plot of carat against price, using different colours to denote the different colour and different plotting symbols to denote the different cuts.

Using the more general ggplot interface

A typical ggplot call

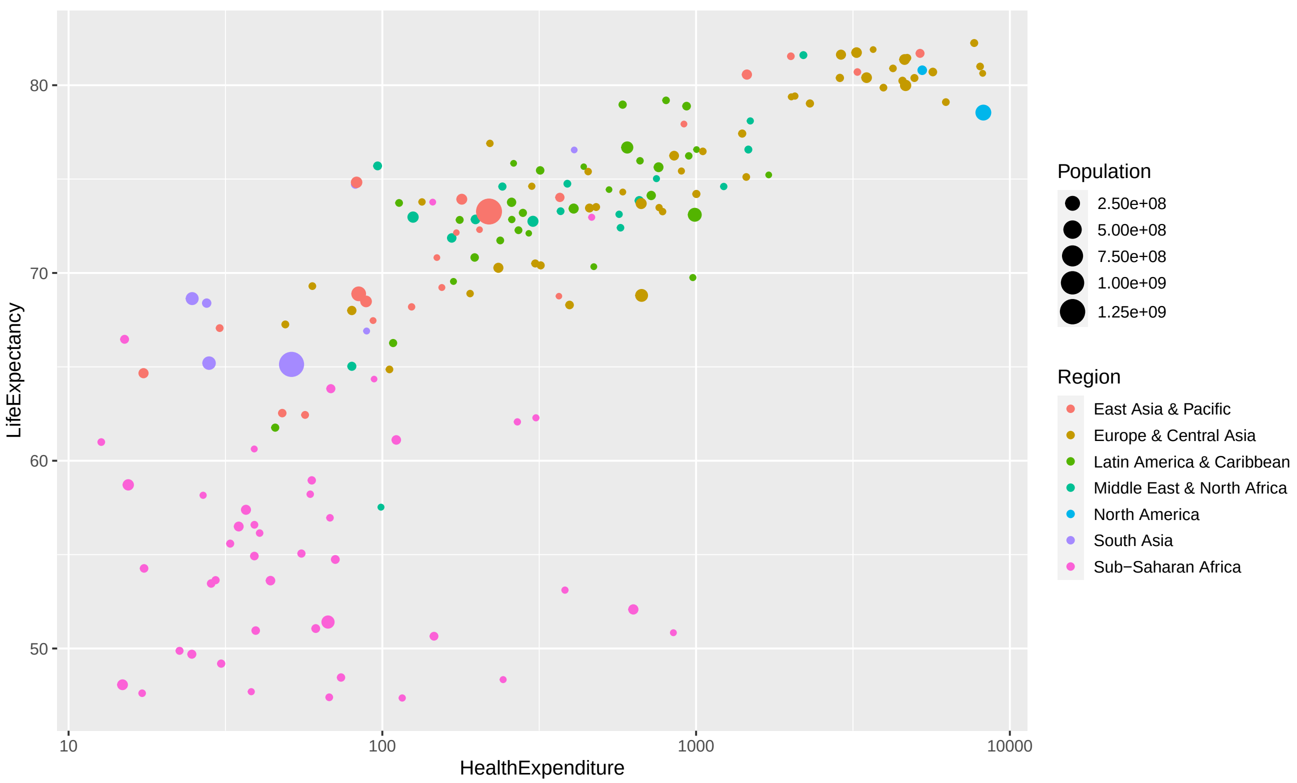
A plotting command for `ggplot` consists of a sequence of function calls added together using the standard sum operator +:

```
ggplot(data=...) +                  # Specify data source
    aes(...) +                      # Generic aesthetics applying to all layers
    geom_<type>(aes(...), ...)  +   # Geometry for one layer with layers-specific aesthetics
    geom_<type>(aes(...), ...)  +
    ...                             # Further arguments for fine-tuning (themes, scales, facets, ...)
```

`geom_<type>` objects do not necessarily have to use the same data as specified in the call to `ggplot`. If the optional argument `data` is specified, then the data source provided is used for this layer.

We can recreate the plot we have just drawn using `ggplot` instead of `qplot`.

```
ggplot(data=health) +
  aes(x=HealthExpenditure,  y=LifeExpectancy) +
  geom_point(aes(colour=Region, size=Population)) +
  scale_x_log10()
```

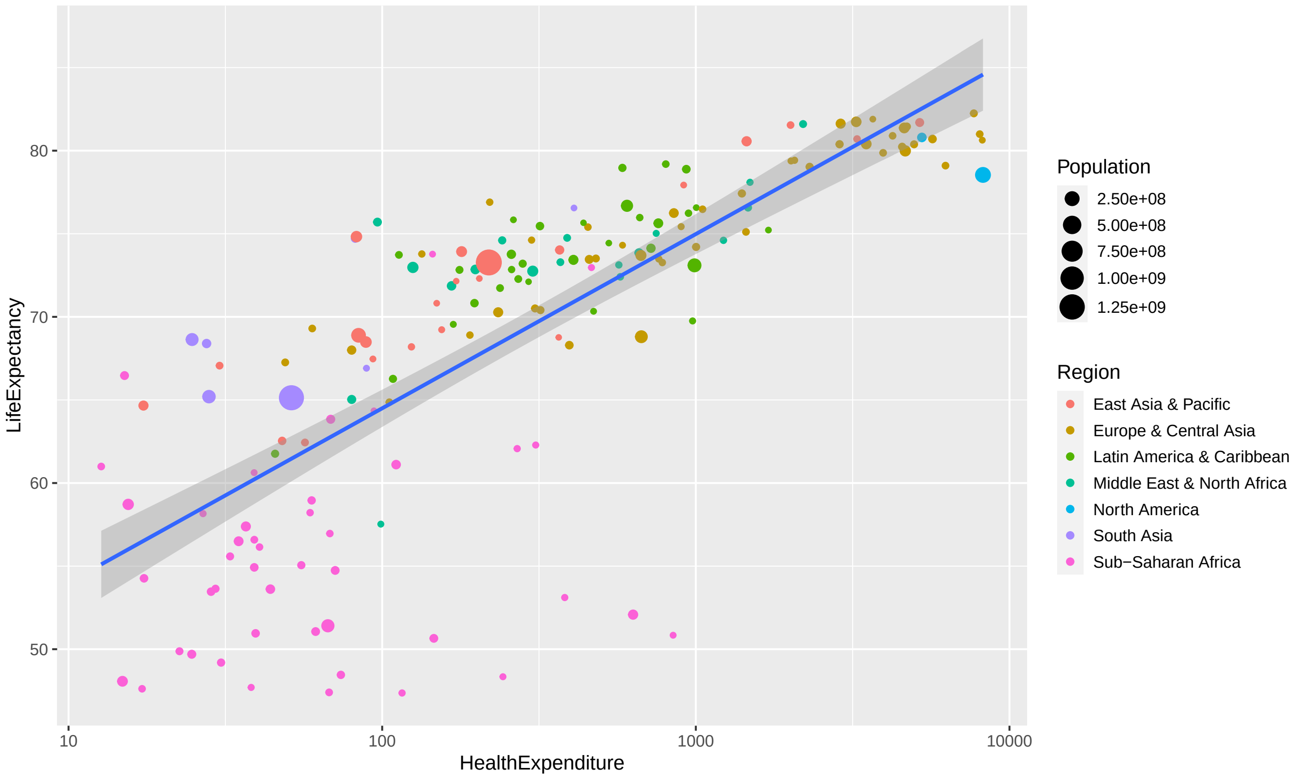


Adding additional layers

Additional layers can simply be added to the plot. For example, we can add an overall regression line with confidence bands (you will learn more about regression lines in the Predictive Modelling course) using

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy) +
    geom_point(aes(colour=Region, size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10()
```

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



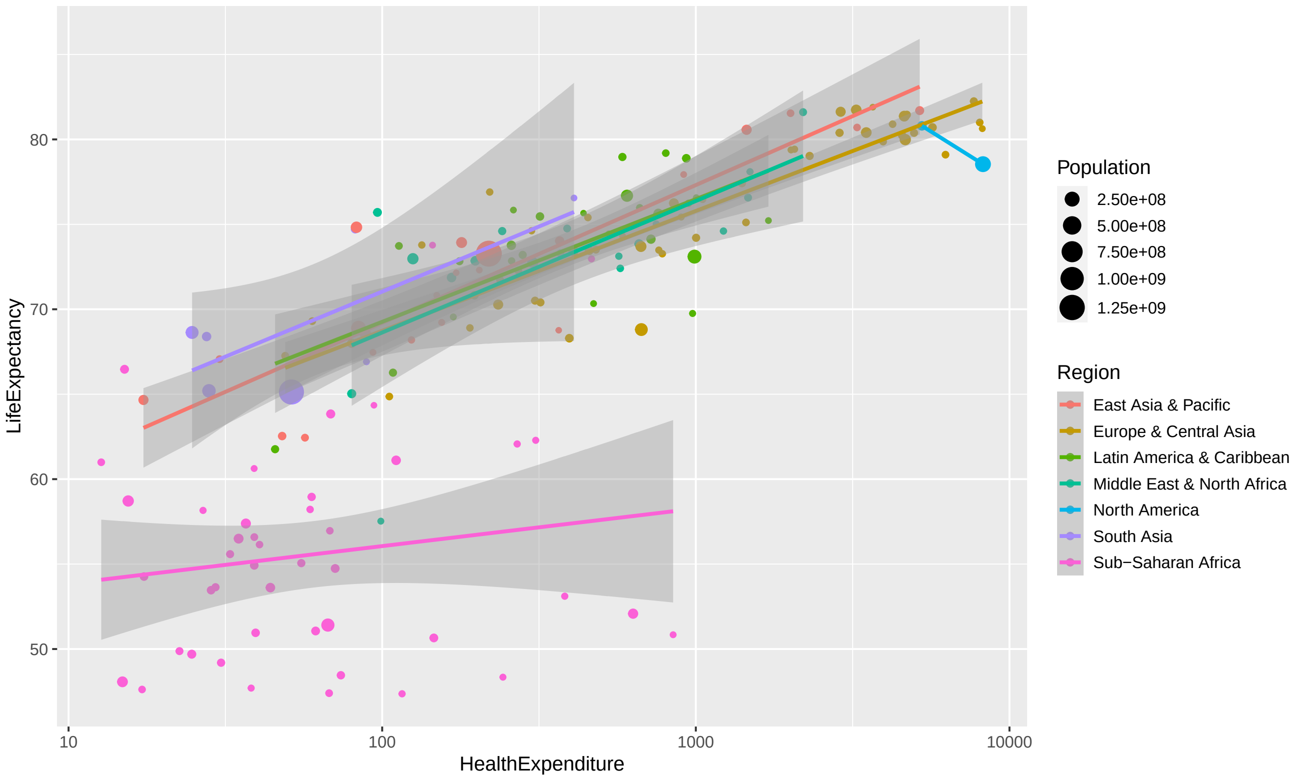
If we want to add a different regression line for each country we have to make sure that a `group` or `colour` aesthetic is passed to `geom_smooth`. We could pass `aes(colour=Region)` to `geom_smooth`. Alternatively, we can move `colour=Region` from the aesthetics specific to `geom_point` to the generic aesthetics, so that `colour=Region` now applies to both `geom_point` and `geom_smooth`.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10()
```

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

## Warning in qt((1 - level)/2, df): NaNs produced

## Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning -Inf
```

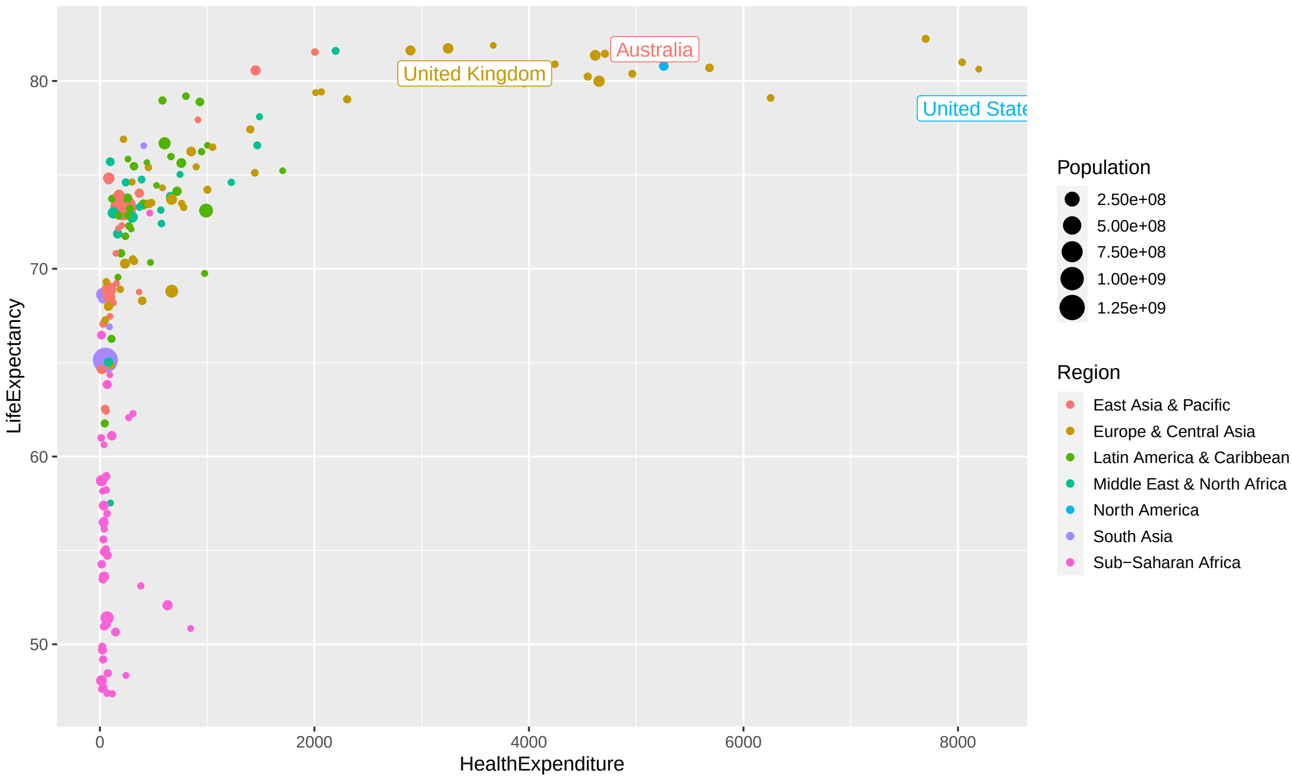


The warning comes from the fact that there are only two North American countries, so we can fit a line through them with no error, which means we cannot draw confidence bands.

The plot looks slightly messy, we will use `facet_wrap` later on to split it into separate panels.

Suppose we want to annotate the observations belonging to Australia, the UK, the US.

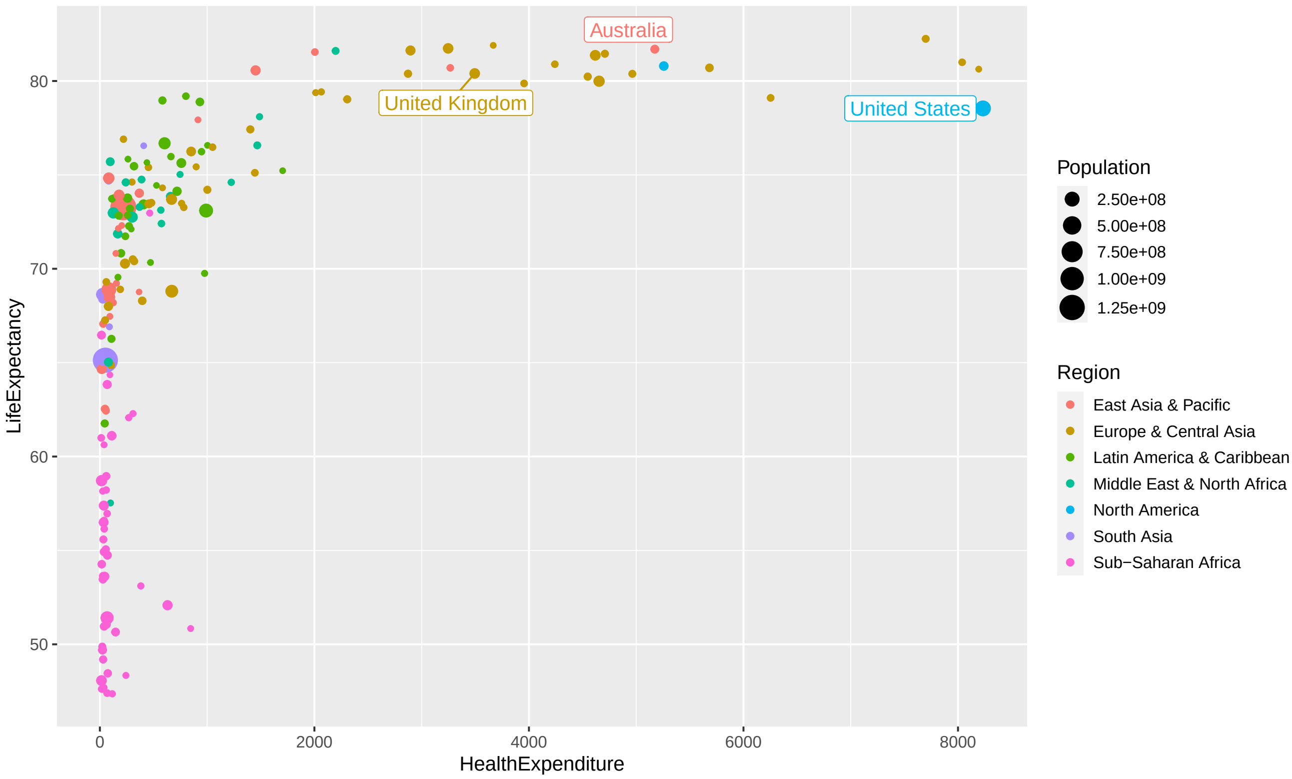
```
health2 <- health %>%
    filter(Country %in% c("Australia", "United Kingdom", "United States"))
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_label(data=health2,
               aes(x=HealthExpenditure,  y=LifeExpectancy, label=Country),
               show.legend=FALSE)
```



The labels however cover the observations and might not be fully visible. This can be avoided by using the function `geom_label_repel` from ggrepel.

```
health <- health %>%
    mutate(CountryLabel=ifelse(Country%in%c("Australia", "United Kingdom", "United States"),
                               as.character(Country),""))

library(ggrepel)
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_label_repel(aes(label=CountryLabel), show.legend=FALSE)
```



This time, we have used a different approach. Rather than subsetting the data and creating a separate data frame only containing the data for the three countries, we have created a new column in the data frame `health`, which is blank except for the three countries. This is required because `ggrepel` layers are only aware of data drawn in their own layer: this way we can avoid the labels covering observations we have not labelled.

Explicit drawing

The standard R plotting functions draw a plot as soon as the plot function is invoked.

Plotting commands in `ggplot2` (including `qplot`) return objects (otherwise the + notation would not work) and only draw the plot when their `print` or `plot` methods are invoked. In the console this is the case when they are used without an assignment.

```
a <- ggplot(data=health) +              # Does not draw anything
       aes(x=HealthExpenditure, y=LifeExpectancy) +
       geom_point()

b <- a + scale_x_log10()                # Does not draw anything either

a                                       # Now the plot stored in a gets drawn
print(a)                                # Draw a again (explicit invocation)

b                                       # Now the plot stored in b gets drawn
```

Inside loops and functions the `print` or `plot` methods need to be invoked explicitly by using the methods `print` or `plot`.

Task 2.

Just like in Week 5, consider two vectors `x` and `y` created using

```
n <- 1e3
x <- runif(n, 0, 2*pi)              # x is random uniform from (0,2*pi)
# x <- sort(x)                      # Sorting of x _not_ needed for ggplot
```

```r
y <- sin(x)                              # Set y to the sine of x
y.noisy <- y + .25 * rnorm(n)            # Create noisy version of y
```

Use `ggplot2` to create a scatterplot of `y.noisy` against `x`, which also shows the noise-free sine curve in `y`.

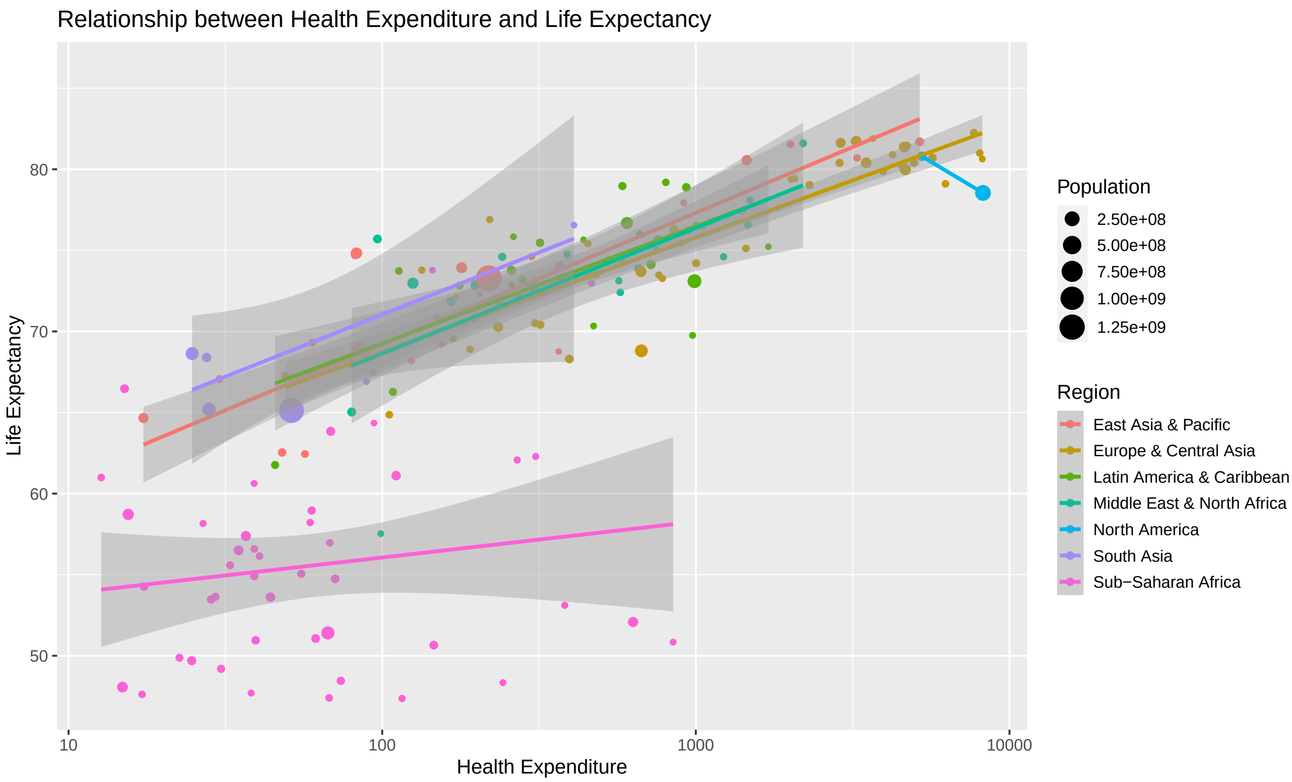
Modifying plots

Labels and titles

We can set the plot title using `ggtitle(title)` and the axis labels using `xlab(label)` and `ylab(label)`.

```
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10() +
    ggtitle("Relationship between Health Expenditure and Life Expectancy") +
    xlab("Health Expenditure") +
    ylab("Life Expectancy")
```

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



Changing the text shown in legends (like in our case the names of the regions) is more complicated. It is almost always easier to simply change the levels of the categorical variable in the dataset itself before invoking `ggplot2` commands.

Scales

Aesthetics control *which* variables are mapped to *which* property of the geometric object. However, aesthetics do not specify *how* this mapping is performed. This is where scales come into play. Scales control *how* any value from the variable is translated into a property of a geometric object: scales control for example how a variable is translated into coordinates (say through a log transform) or into colours (say though a discrete colour palette).

`ggplot2` automatically chooses (what it thinks is) a suitable scale. This is usually reasonable, but on occasions it might be necessary to override this.

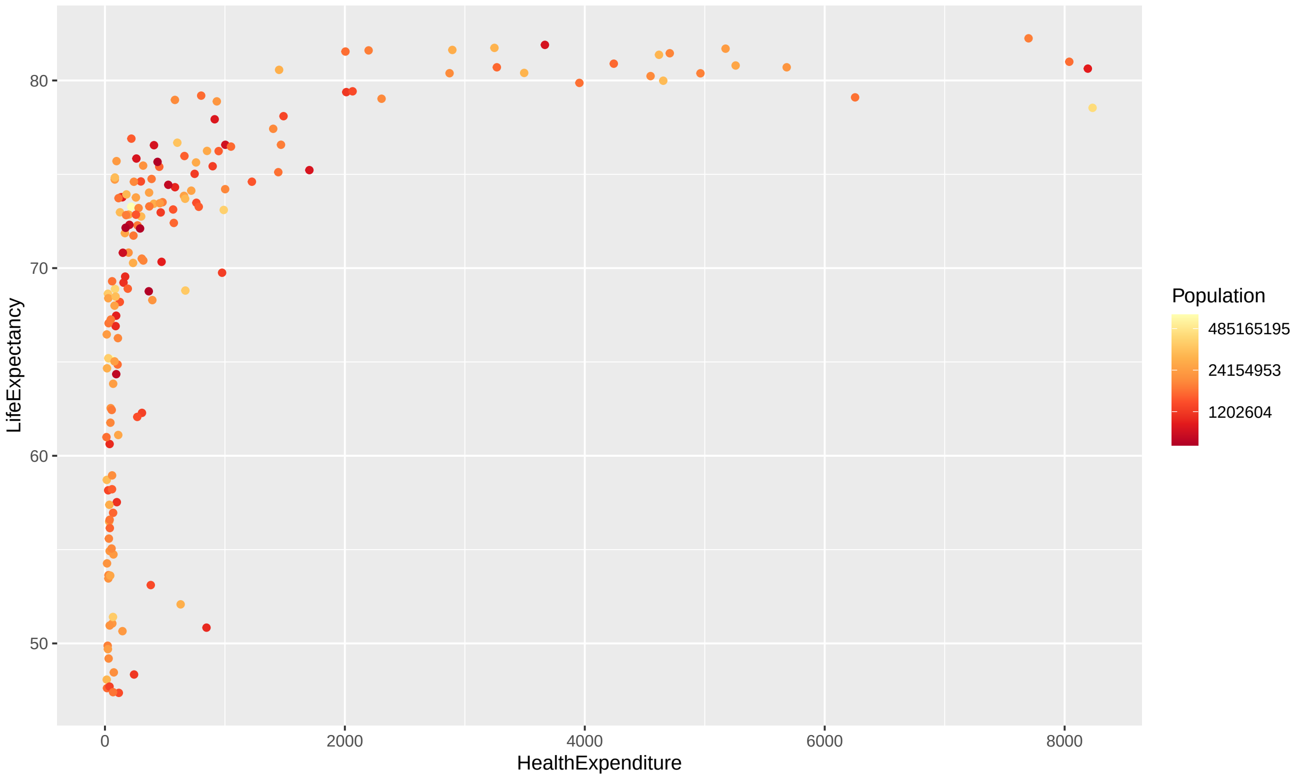
There is a family of scale functions for each aesthetic. The template for the function name for scales is `scale_<aesthetic>_<type>`.

Scales for continuous data We have already seen that we can log-transform the axes using `scale_x_log10` and `scale_x_log10`. The more general functions for coordinate transforms are `scale_<x or y>_continous(...)`. We can. amongst others, set the axis label (argument `name`, the ticks and tickmarks (arguments `breaks` and `labels`) the limits (argument `limit`) and the transform to be used (argument `trans`).

The axes might use scientific notation (e.g. "4e5"). If you want to avoid using scientific notation and use fixed notation, change the `scipen` option in R, which controls when scientific notation is used (for example run `options(scipen=1e3)`).

There are functions for mapping continuous data to other aesthetics, too. For example, `scale_colour_gradient` converts a continuous variable to a colour using a gradient of colours. The arguments `low` and `high` specify the colours used at the two ends. `scale_colour_gradient2` allows for also specifying a mid-point colour (argument `mid`). `scale_colour_gradientn` is the most general function it allows specifying a vector of colours and corresponding vector of colours. The function `scale_colour_distiller` uses the colour brewer available at ColorBrewer and allows for constructing colours scales which are photocopier-safe and/or work for colour-blind readers.

```
a <- ggplot(data=health) +
       aes(x=HealthExpenditure, y=LifeExpectancy) +
       geom_point(aes(colour=Population)) +
       scale_colour_distiller(palette="YlOrRd" , trans="log")
a
```



We have used `trans="log"` to use the log-transformed values of the population sizes (due to its skewness). The values given in the legend seem slightly odd choices: this is due to the log-transform (they are roughly $\exp(14)$, $\exp(17)$ and $\exp(20)$, so "nice" numbers on the log scale).

We have stored the plot in a variable `a` so that we can redraw it later on with different themes.

Scales for discrete data There are also various scaling functions for discrete data, such as `scale_colour_brewer`.

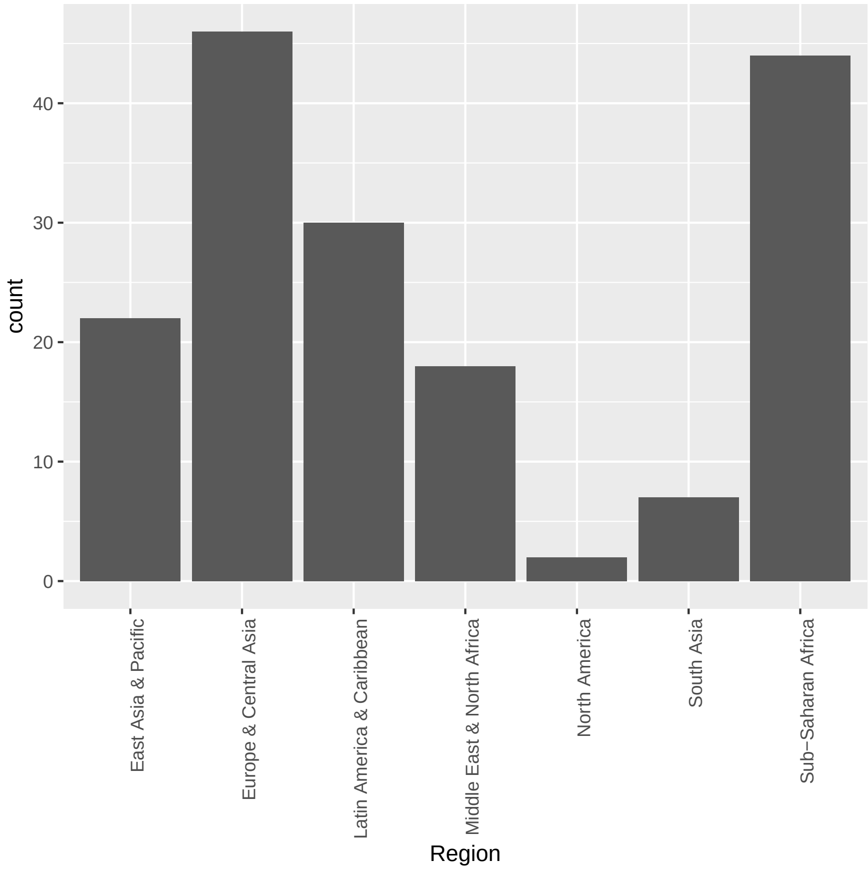
Note that there are separate scales for colour (outline colour – example: `scale_colour_brewer`) and fill (fill colour – example: `scale_fill_brewer`).

Statistics

Sometimes data has to be aggregated before it can be used in a plot. For example, when creating a bar plot illustrating the distribution of a categorical variable we have to count how many observations there are in each category. This will then determine the height of the bars. ggplot2 automatically chooses (what it thinks is) a suitable statistic.

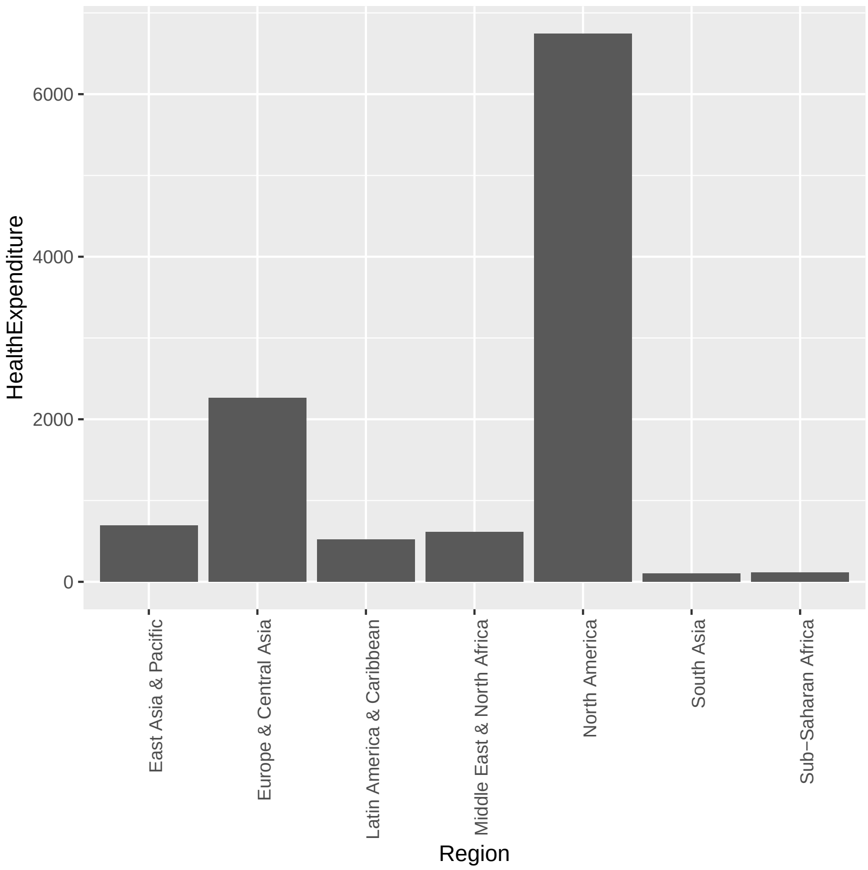
For example, when we draw a bar plot using geom_bar, it uses by default the statistic count, which first produces a tally. We don't need to worry about this, ggplot2 does all the work for us.

```
ggplot(data=health) +
    geom_bar(aes(x=Region)) +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) # Rotate x axis labels
```



Suppose we now want to a draw bar chart visualising the mean health expenditure in each region. Now we don't want ggplot2 to produce a tally of how often which value occurs, we want it to simply draw the bars to the heights specified in the data. Because we now want no aggregation, we have to use the statistic identity.

```
library(dplyr)
HESummary <- health %>%                                       # Get avg health exp
                 group_by(Region) %>%
                 summarise(HealthExpenditure=mean(HealthExpenditure))
ggplot(data=HESummary) +
    geom_bar(aes(x=Region, y=HealthExpenditure), stat="identity") +
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) # Rotate x axis labels
```

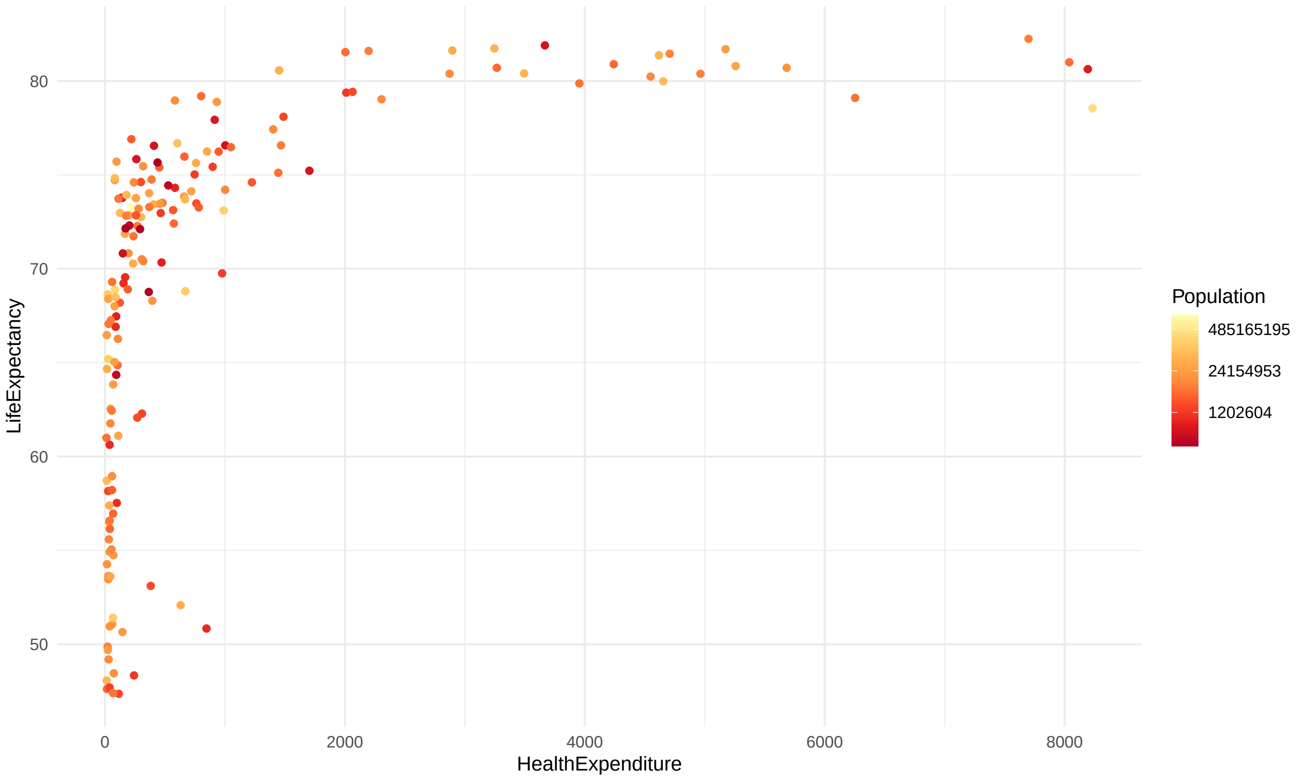


Theming

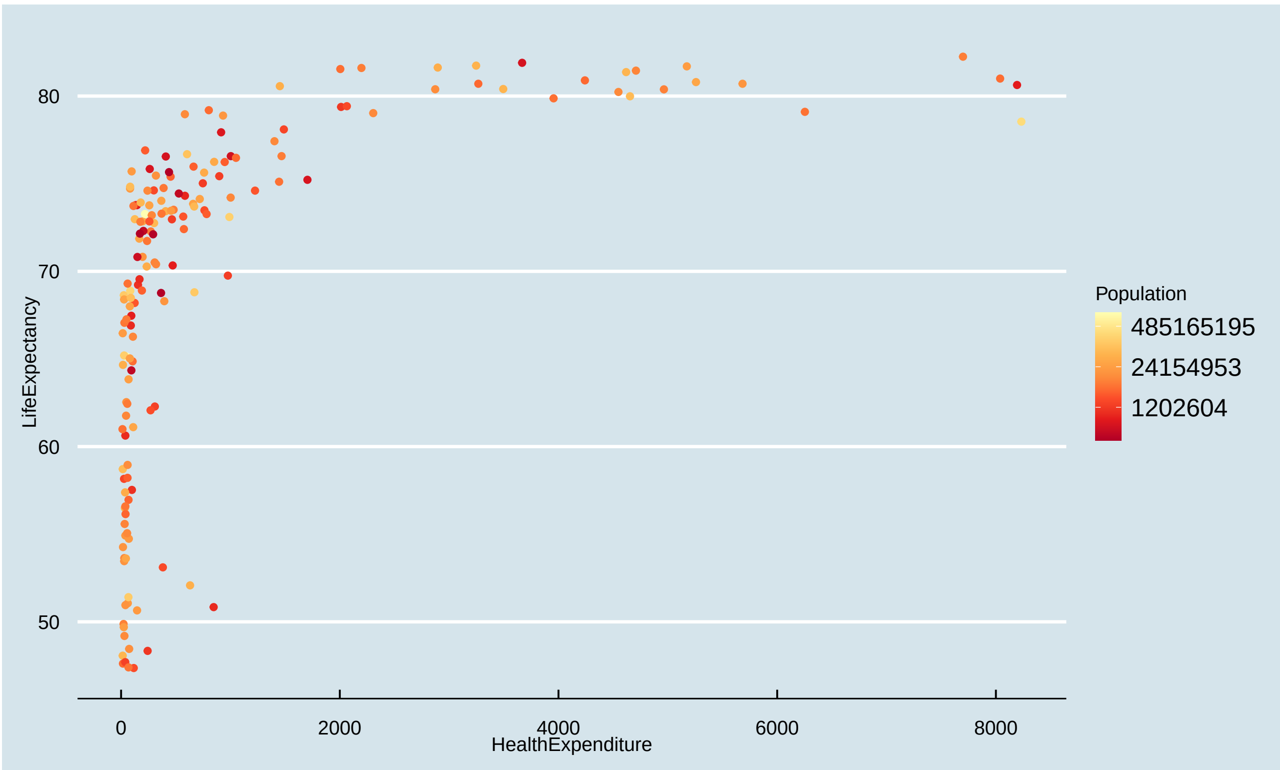
Themes can be used to customise how `ggplot2` graphics look like. We have already used `theme` to change how the horizontal axis is typeset.

`ggplot2` has several themes built-in. The default theme is `theme_gray`. Other themes available are `theme_bw` (monochrome), `theme_light`, `theme_lindedraw` and `theme_minimal`. Further themes are available in extension packages such ggthemes.

```
a + theme_minimal()
```




```
library(ggthemes)
a + theme_economist() + theme(legend.position="right")
```



Arranging plots (faceting)

The function `facet_grid(rvar~cvar)` creates separate plots based on the values `rvar` (rows) and `cvar` (columns) takes. The function `facet_wrap(~var1+var2)` arranges the plots in several rows and columns without rigidly associating one variable with rows and one with columns. Continuous variables need to be discretised (for example using `cut`) before they can be used for defining facets.

```r
ggplot(data=health) +
    aes(x=HealthExpenditure,  y=LifeExpectancy, colour=Region) +
    geom_point(aes(size=Population)) +
    geom_smooth(method="lm")  +
    scale_x_log10() +
    facet_wrap(~Region)
```

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

## Warning in qt((1 - level)/2, df): NaNs produced

## Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning -Inf
```

Arranging plots in more general ways (like in `par(mfrow=c(...))` or `layout`) is not directly possible with `ggplot2`. The package gridExtra however provides a function `grid.arrange`, which allows for arranging `ggplot2` plots side by side.

Classical plot customisation functions are not compatible with ggplot2

`ggplot2` plots are not compatible with the functions used to customise or arrange basic R plots, such as `par` or `layout`.

Examples

In this section we return to some of the examples from last week and reproduce them in `ggplot2`.

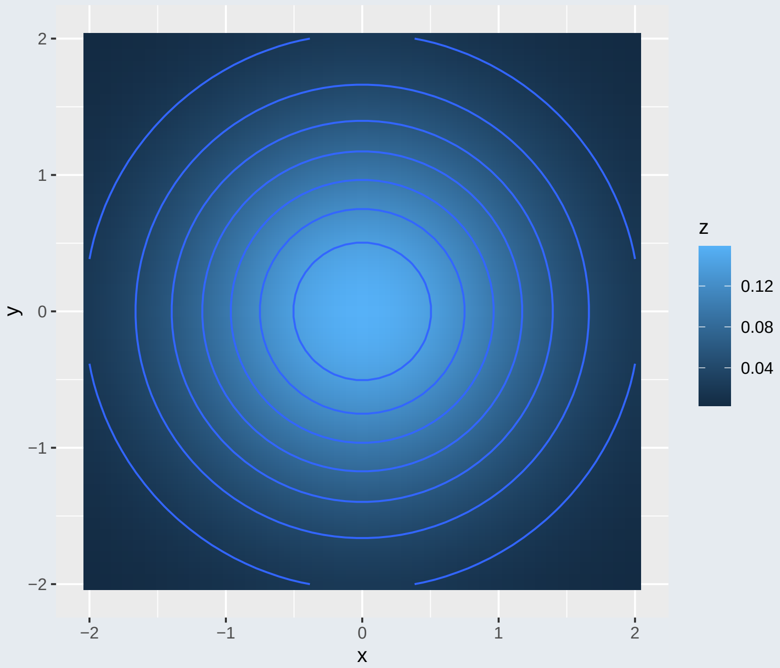
Example 1 (The bivariate Gaussian density).

We start by creating a data frame with three columns, `x`, `y` and `z`, which holds the value of the bivariate Gaussian density.

```
x <- seq(-2, 2, len=50)
y <- seq(-2, 2, len=50)
data <- expand.grid(x=x, y=y) %>%
    mutate(z=dnorm(x)*dnorm(y))
```

In contrast to the classical plotting functions `ggplot2` needs the input data in "long", rather than "wide" format, so there is no need to call `pivot_wider` as we did last week.

```
# Note - coord_fixed is used her to make sure
# plot uses equal scales, so circles are
# actually circles, and not ellipsoids
ggplot(data=data) +
    aes(x=x, y=y) +
    geom_raster(aes(fill=z), interpolate=TRUE) +
    geom_contour(aes(z=z)) +
    coord_fixed()
```

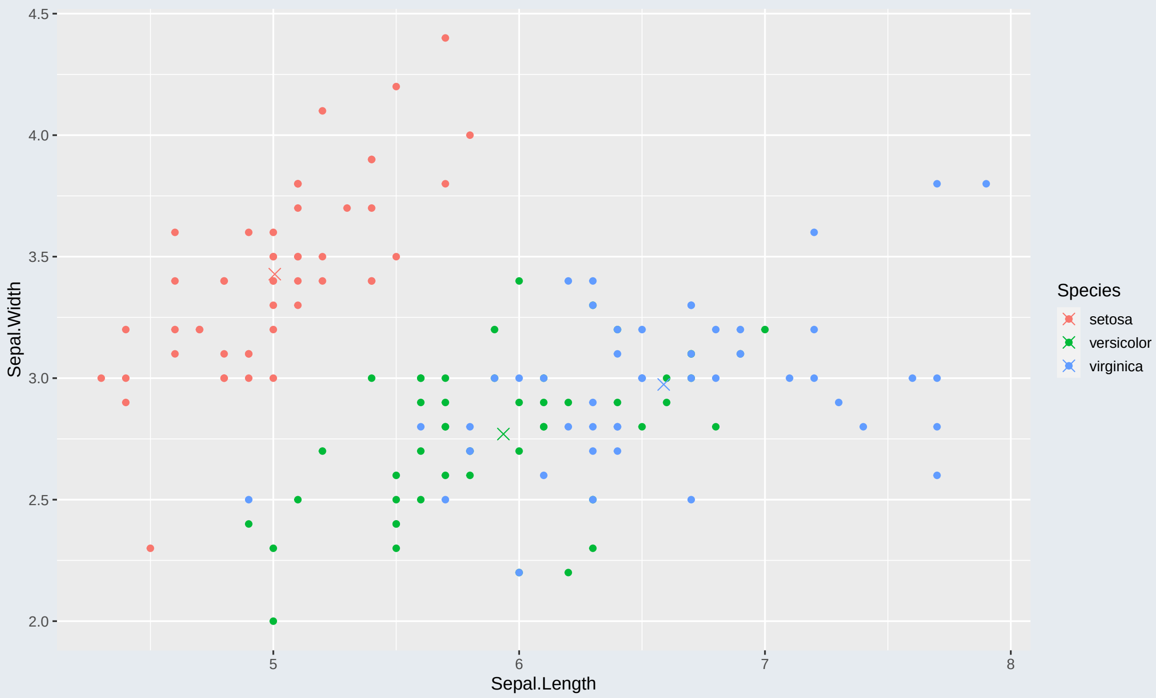


Example 2 (Fisher's iris data).

In this example we look at again the sepal length and width from Fisher's famous iris data.

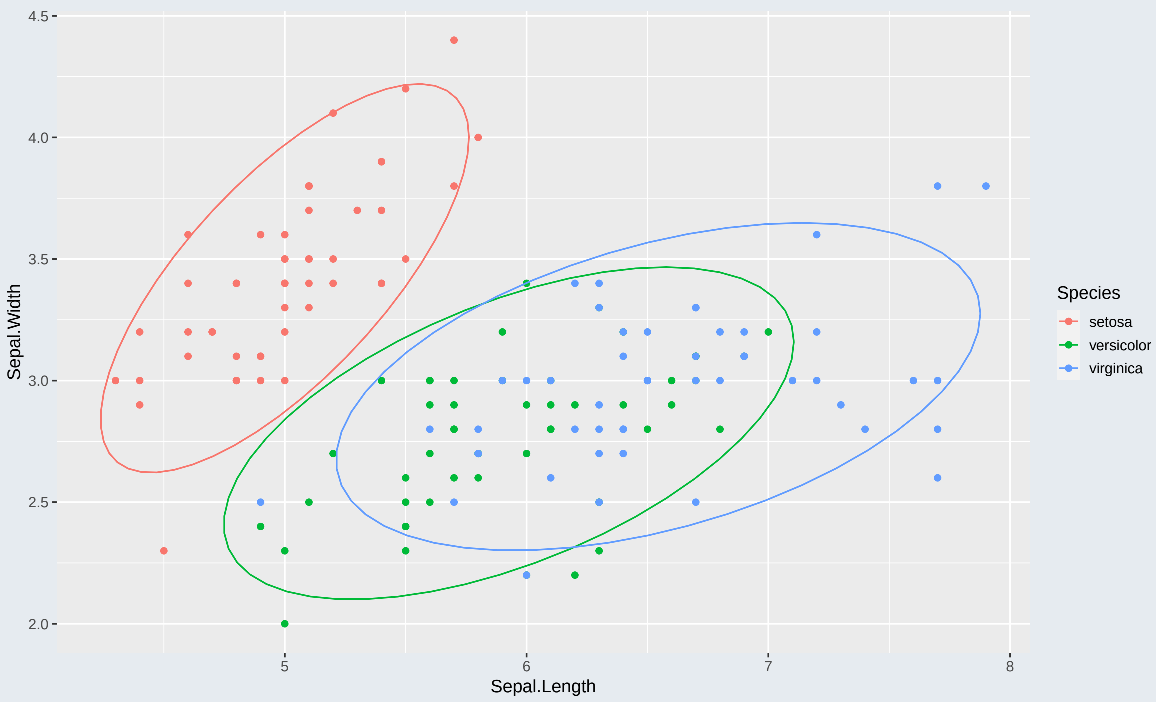
```
species.means <- iris %>%              # Get species means for centroid
                     group_by(Species) %>%
                     summarise_all(mean)
```

```
ggplot(data=iris) +
    geom_point(aes(x=Sepal.Length, y=Sepal.Width, colour=Species)) +
    geom_point(data=species.means,
               aes(x=Sepal.Length, y=Sepal.Width, colour=Species), size=3, shape=4)
```



In this example we could also use the function `stat_ellipse`, which draws confidence ellipsoids around data.

```
ggplot(data=iris) +
    aes(x=Sepal.Length, y=Sepal.Width, colour=Species) +
    geom_point() +
    stat_ellipse()
```

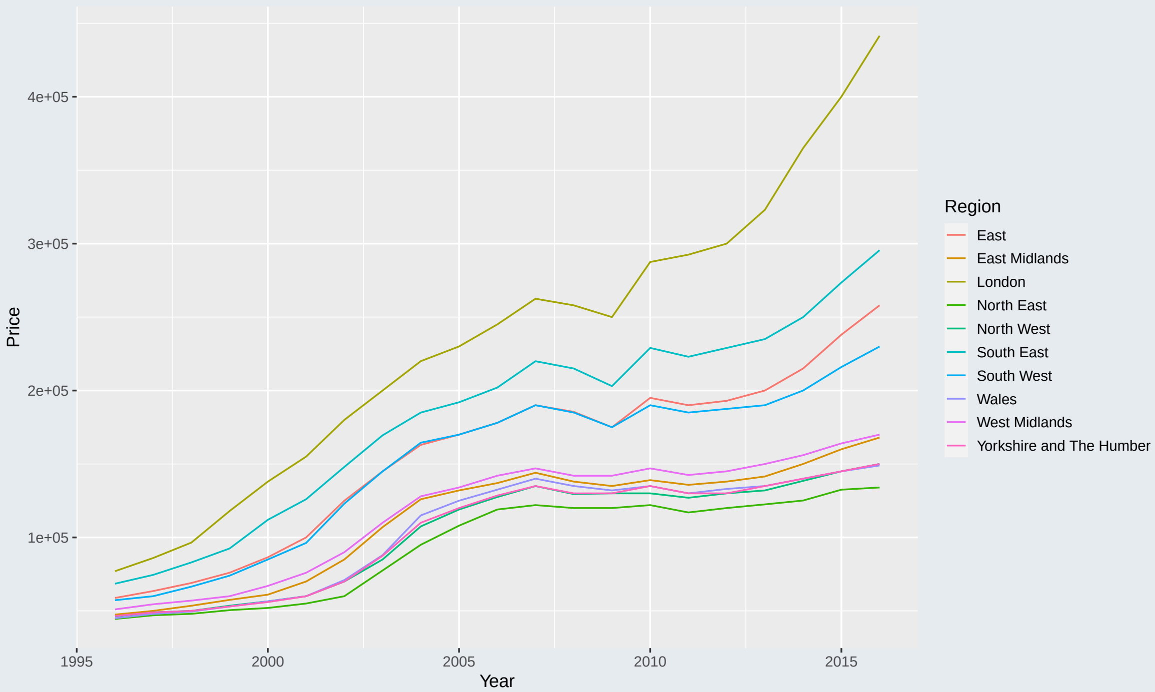


Example 3 (House prices in the UK).

In this example we will use the house price data from last week.

The data is in "wide" format. To be able to use the data in `ggplot` we first need to translate it into "long" format using the function `pivot_longer` from `tidyr`.

```
library(tidyr)
hp <- hp %>%pivot_longer(cols=2:11,names_to="Region",values_to="Price")
ggplot(data=hp) +
    geom_line(aes(x=Year, y=Price, colour=Region))
```



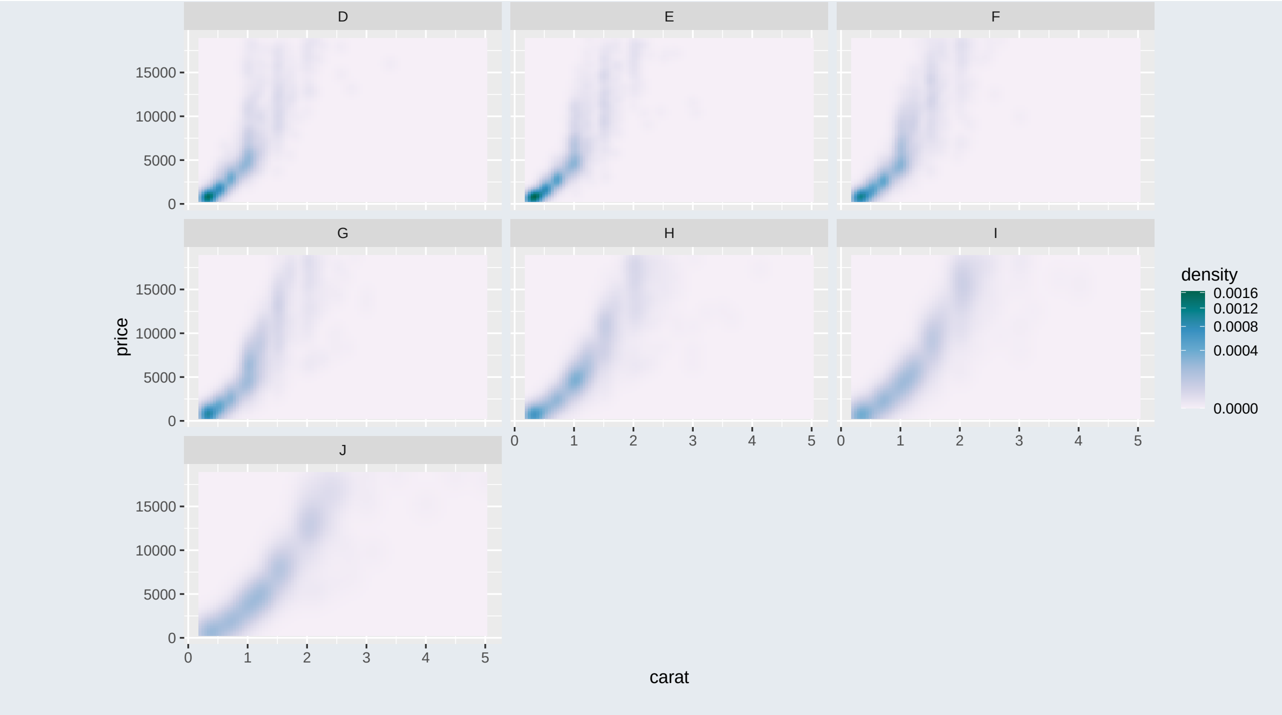
We could have also used `qplot` in this case.

```
qplot(Year, Price, data=hp, geom="line",colour=Region)
```

Example 4 (Diamonds data (revisited)).

There are a large number of observations in the `diamond` data from task 1, making the plot difficult to read as we cannot see how many observations were plotted on top of each other. It might be better to plot the density of the data, rather than individual observations.

```
ggplot(data=diamonds) +
    aes(x=carat, y=price) +
    stat_density_2d(geom = "raster", aes(fill = ..density..), contour = FALSE) +
    scale_fill_distiller(palette="PuBuGn", direction=1, trans="sqrt") +
    facet_wrap(~color)
```

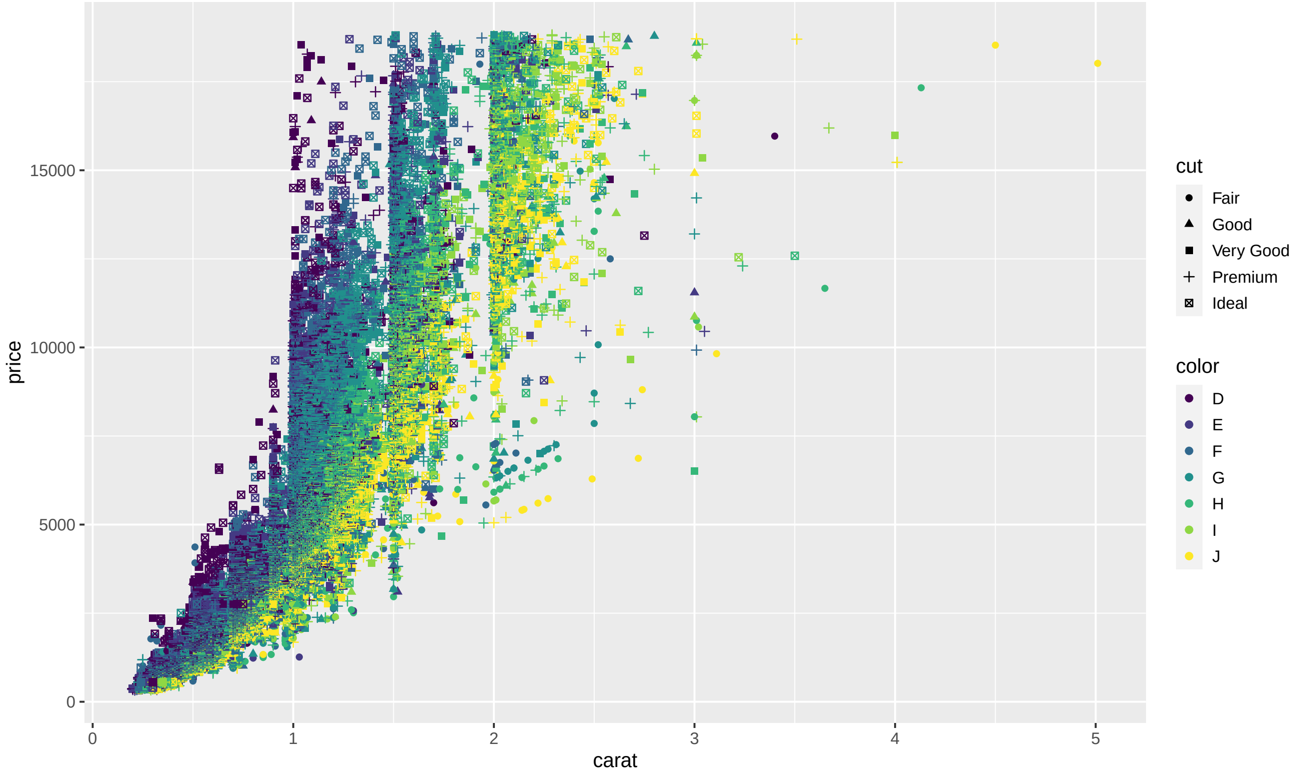

D
E
F
G
H
I
J
price
15000
10000
5000
0
carat
0
1
2
3
4
5
density
0.0016
0.0012
0.0008
0.0004
0.0000

Answers to tasks

Answer to Task 1. We can use the following R code.

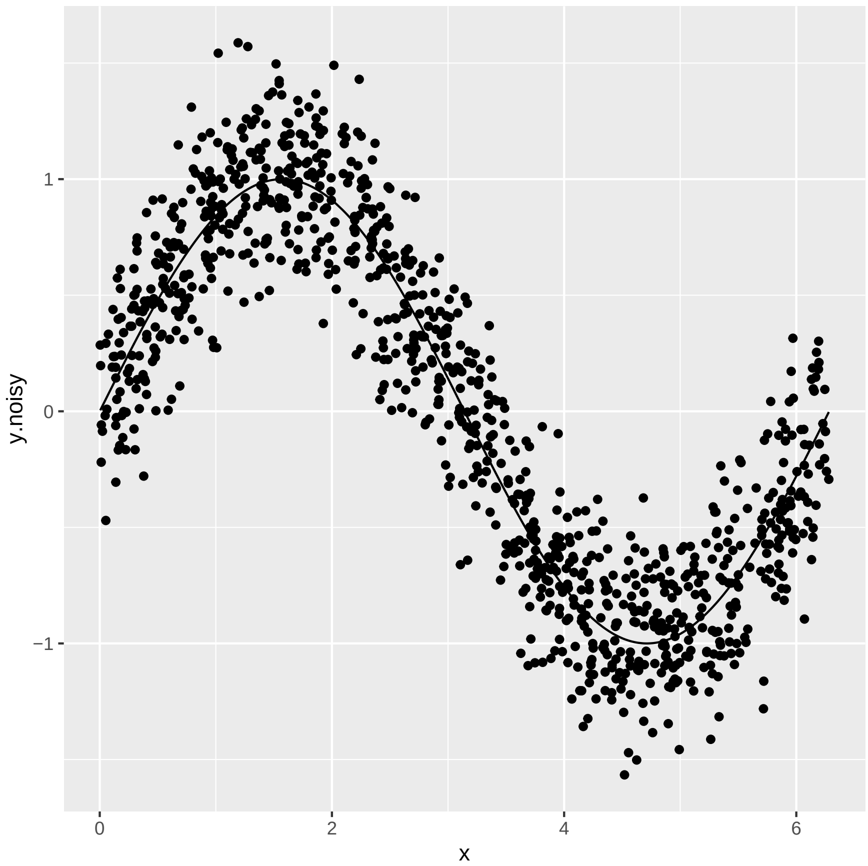
```
qplot(carat, price, data=diamonds, colour=color, shape=cut)
```

```
## Warning: Using shapes for an ordinal variable is not advised
```



Answer to Task 2. We can use the following R code:

```
ggplot() +                             # No need to use data=... as x, y and y.noisy
                                       # are variables in the workspace and not columns
                                       # in a dataset
    geom_point(aes(x, y.noisy)) +
    geom_line(aes(x, y))
```



It does not matter whether `geom_point` or `geom_line` comes first. `ggplot2` adapts the axes so that all objects drawn fit (and not just the first one as is the case when using standard R plotting functions `plot` and `points`).