```r
# note to self: use the regexp ```\{r.*\}\s([\s\S]+?)``` to extract only the r-code
# from the complete rmd tutorial file

# first we need to install the ggplot and some supporting libraries
# (skip this step if the library is already loaded)
install.packages("ggplot2")

# we will require the ggplot2 package for our graphics
# note: there are some additional useful packages such as plyr,
#  reshape2 and scales which you may find useful
require("ggplot2")

# prices of 50'000 sparkly round cut diamonds
head(diamonds)

# motor trend car road tests
head(mtcars)

# vapor pressure of mercury at certain temperatures
head(pressure)

# histogram with qplot
qplot(clarity, data=diamonds, fill=cut, geom="bar")

# histogram with ggplot -> same output
ggplot(diamonds, aes(clarity, fill=cut)) + geom_bar()

# quickly create a scatterplot of our data
qplot(wt, mpg, data=mtcars)

# data can be transformed with functions
qplot(log(wt), mpg-10, data=mtcars)

# plots can be further refined by using additional parameters
# note: we are mapping the variable «qsec» to a color
qplot(wt, mpg, data=mtcars, color=qsec)

# color and colour will work for most cases
qplot(wt, mpg, data=mtcars, color=qsec)
qplot(wt, mpg, data=mtcars, colour=qsec)

# note: in this example ggplot is trying to map the size of a point
#  to a scale of [10] (which is probably not as intended)
qplot(wt, mpg, data=mtcars, color=qsec, size=10)

# use the I() function «as is» to set aesthetics instead of mapping
qplot(wt, mpg, data=mtcars, color=qsec, size=I(10))
# side note: it is possible to use alpha-blending for overlapping elements
qplot(wt, mpg, data=mtcars, color=qsec, size=I(10), alpha=qsec)

# note: alpha-opacity is set between 0 (transparent) and 1 (opaque)
qplot(wt, mpg, data=mtcars, color=qsec, size=I(10), alpha=I(0.5))

# we take a closer look at the variable cyl from the dataset mtcars
# note: the variable is stored as a continuous number not as a factor
head(mtcars)
summary(mtcars$cyl)
table(mtcars$cyl)

# regular mapping will be displayed on a continuous scale
qplot(wt, mpg, data=mtcars, color=cyl)

# factored variables will be displayed with a discrete scale
qplot(wt, mpg, data=mtcars, color=factor(cyl))
```

```r
# ggplot will try to guess the «correct» plot for your data
qplot(wt, mpg, data=mtcars)
qplot(factor(cyl), data=mtcars)

# a specific type of plot can be set with the attribute geom=«type»
qplot(wt, mpg, data=mtcars, geom="point")
qplot(wt, mpg, data=mtcars, geom="line")

# plot-types can be combined
qplot(wt, mpg, data=mtcars, geom=c("line", "point"))

# note: problem if only size of points should be increased
qplot(wt, mpg, data=mtcars, geom=c("line", "point"), size=I(2))

# pro-tipp: resort to ggplot syntax (more on that later)
qplot(wt, mpg, data=mtcars) + geom_line() + geom_point(size=4)

# a plot can be flipped by 90°
# note: coord_flip() will rotate the plot after calculation of
#  any summary statistics (i.e. smoothers or alike)
qplot(factor(cyl), data=mtcars)
qplot(factor(cyl), data=mtcars) + coord_flip()

# difference between fill/color bars
qplot(factor(cyl), data=mtcars, fill=factor(cyl))
qplot(factor(cyl), data=mtcars, color=factor(cyl))

# use different position properties for bars (stacked, dodged, fill, identity)
head(diamonds)
qplot(clarity, data=diamonds, geom="bar", fill=cut, position="stack")
qplot(clarity, data=diamonds, geom="bar", fill=cut, position="dodge")
qplot(clarity, data=diamonds, geom="bar", fill=cut, position="fill")
qplot(clarity, data=diamonds, geom="bar", fill=cut, position="identity")

# we are going to use some pressure data
head(pressure)

# nothing happens if we only define our data
ggplot(pressure)

# but we can quickly add a representation
# note: the aes() function is used for variable mapping
ggplot(pressure) + geom_point(aes(x=temperature, y=pressure))

# as x and y are used so often, we can leave it of
# note: for later maintenance it is usually better to specify it
ggplot(pressure) + geom_point(aes(temperature, pressure))

# note: you can access the previously created plot with «last_plot()»
last_plot()

# specify a value allocation outside of the aes() function
# if an aestetic should be set to a specific value
ggplot(pressure) + geom_point(aes(temperature, pressure), size=4)

# aesthetics can also be defined separately
ggplot(pressure) + aes(temperature, pressure) + geom_point(size=4)

# create some normal distributed test data
tmp <- data.frame(x=rnorm(4000), y=rnorm(4000))
p.myplot <- ggplot(tmp, aes(x,y))

# default plotting
p.myplot + geom_point(color="red")
```

```r
# plotting using hollow circles
p.myplot + geom_point(shape=1, color="red")

# plotting using pixels
p.myplot + geom_point(shape=".", color="red")

# plotting using alpha transparency
# note: requires the scales package (included with ggplot2)
p.myplot + geom_point(color=scales::alpha("red", 1/2))
p.myplot + geom_point(color=scales::alpha("red", 1/6))

# ggplot will actually return an object that can be modified
# note: the object can also be saved for later use with save()
#   saving a plot or layer definitions will also include the plot data
p.myplot <- ggplot(pressure)

# summary information about the plot
summary(p.myplot)

# adding some additional layers
p.myplot <- p.myplot + aes(temperature, pressure) + geom_point(size=4)
summary(p.myplot)

# the plot can be printed by just calling the object or using print()
p.myplot
print(p.myplot)

# the underlying data is saved within the ggplot-object. modifications of
# the data will not alter the plot if the plot-code is not rerun.
# there is however a special syntax to run the plot with updated data
pressure2 <- data.frame(
    "temperature"=pressure$temperature, "pressure"=log(pressure$pressure))

# print the plot with updated data
p.myplot %+% pressure2

# a plot can be exported using ggsave
# note: the respective rendering device needs to be installed
ggsave(file="testplot.pdf", plot=p.myplot, width=10, height=5)
ggsave(file="testplot.svg", plot=p.myplot, width=10, height=5)
ggsave(file="testplot.png", plot=p.myplot, dpi=72, width=10, height=5)

# let's define a base plot and aesthetic-mapping
p.myplot <- ggplot(pressure) + aes(x=temperature, y=pressure)

# using multiple layers
p.myplot +
    geom_point(color="purple3", size=6) +
    geom_line(color="steelblue2", size=2)

# the order of the layers does mather
# (each new layer is drawn on top of the previous)
p.myplot +
    geom_line(color="steelblue", size=2) +
    geom_point(color="purple3", size=6)

# aesthetics defined in the base layer will be used for all layers
# note: setting attributes to a value will not apply it to other layers
ggplot(pressure, aes(x=temperature, y=pressure), color="red") +
    geom_line(size=4, alpha=0.3) +
    geom_point(size=4)


# the actual arguments to map variables is mapping=«aes()» and
# geom_params=«list()» to set variables respectively
ggplot(pressure) +
```

```r
    geom_point(
        mapping=aes(x=temperature, y=pressure, color=factor(temperature)),
        geom_params=list(size=4, shape=18)
    )

# it is possible to mix qplot and ggplot
qplot(temperature, pressure, data=pressure, geom="line", lty=I("dashed")) +
    geom_point(size=4)

# there is also a different syntax with layer()
ggplot(pressure, aes(temperature, pressure)) +
    layer(geom="line", mapping=aes(color=temperature), size=4) +
    layer(geom="point", size=4, color="purple3")

# let's use some additional data in the plot
# note: the scales of the different datasets are unified
ggplot(pressure) +
    aes(x=temperature, y=pressure) +
    layer(geom="line", mapping=aes(color=temperature), size=4) +
    geom_point(data=mtcars, aes(hp, disp), color="purple3", size=3)

# setup our plot with default scales
p.myplot <- ggplot(pressure) +
    aes(temperature, pressure, color=factor(temperature)) +
    geom_point(size=4)

# scales can be limited to a certain range
p.myplot
p.myplot + scale_x_continuous("Temperature", limits=c(200, 400))

# scales that are used as axes will take the name as axis label
p.myplot +
    scale_color_discrete(name="Temperature \nin C°") +
    scale_y_continuous(name="Air pressure at sea level")

# legends can also be removed (if not important to understand the plot)
p.myplot + scale_color_discrete(guide="none")

# setup a different plot
p.myplot <- ggplot(diamonds, aes(cut, fill=color)) + geom_bar()
p.myplot

# the axis can be renamed using two different methods
p.myplot + xlab("Diamond Cut")
p.myplot + scale_x_discrete(name="Diamond Cut Description")
p.myplot + scale_y_continuous(name="Number of Diamonds")

# names of legends can also be set
p.myplot + scale_fill_discrete(name="Diamond Color")

# using some custom colors
# note: brewer colors were created for good readable maps and often provide
#  a good alternative to the standard colors. to see all available brewer
#  palettes use «RColorBrewer::display.brewer.all()»
p.myplot + scale_fill_grey()
p.myplot + scale_fill_hue()
p.myplot + scale_fill_brewer()
p.myplot + scale_fill_brewer(type="seq", palette="3")
p.myplot + scale_fill_brewer(palette="Paired")

# using a custom color palette with specified order
# note: color values should be specified as hex or color names
p.myplot + aes(fill=cut) + scale_fill_manual(
    values = c("#7fc6bc","#083642","#b1df01",
                        "#cdef9c","#466b5d", "#744db5", "#ccb2e8"))

```

```r
# using predefinded colors for specific values
# note: values that are not present in the data will not be shown
p.myplot + aes(fill=cut) + scale_fill_manual(
    values = c("Fair"="#083642", "Good"="#466b5d",
                         "Very Good"="#7fc6bc","Premium"="#cdef9c",
                         "Ideal"="#b1df01", "Not specified"="#ffffff"))

# removing values from the legend and custom labelling of values
# note: you must specify colors for all existing values
p.myplot + scale_fill_manual(
    name="Colors",
    values = c("D"="#083642", "E"="#466b5d", "F"="#7fc6bc","G"="#cdef9c",
                         "H"="#b1df01", "I"="#ababab", "J"="#ececec"),
    breaks = c("D", "E", "F"),
    labels = c("E"="Dark Green", "D"="Esmerald", "F"="Wood"))

# legends can also be styled using guides
# note: guides can be defined once and be easily applied to multiple plots
p.mylegend <- guide_legend(
        title="Color of the \nDiamond",
        title.position="top",
        direction="horizontal",
        label.position="top",
        label.hjust=0.5,
        label.vjust=0.5,
        ncol=2,
        byrow=TRUE,
    )

# apply some styling to the legend
p.myplot + guides(fill = p.mylegend)
p.myplot + scale_fill_discrete(guide=p.mylegend)

# handling problems with alpha transparency
p.myplot + aes(alpha=color)

# remove the alpha transparency for the legend
p.myplot + aes(alpha=color) +
    guides(fill = guide_legend( override.aes=list(alpha=1) ))

# limiting scales will remove all points that are outside of the scale
# note: be careful, this is not the same as just focusing on a graph region
p.myplot + scale_y_continuous(limits=c(0,15000))

# to focus on a specific region, the coord_cartesian() function
#  should be used with the specified limits
p.myplot + coord_cartesian(ylim=c(0,15000))

# histograms will use stat_bin to calculate number of items per bin
ggplot(mtcars) + aes(qsec) + geom_histogram(binwidth=0.5)
ggplot(mtcars) + aes(qsec) + geom_histogram(binwidth=1)

# define a base plot to illustrate smoothed lines
p.myplot <- ggplot(mtcars) + aes(x=disp, y=mpg) + geom_point(size=4)
p.myplot

# draw a smooth line (local regression function) through the points
# note: the default smoothing function is loess
p.myplot + geom_line(stat="smooth")

# using the smooth geom with standard deviation
p.myplot + geom_smooth()

# fit the regression closer to the data with span=«0-1»
p.myplot + geom_smooth(span=0.4)
p.myplot + geom_smooth(span=1)
```

```r

# turning off the confidence interval
# note: the attribute level can be used to set ci-level
p.myplot + geom_smooth(se=FALSE)

# using a different method for smoothing (i.e. linear modelling)
p.myplot + geom_smooth(method="lm")

# using a cutom formular for fitting
library(splines)
p.myplot + geom_smooth(method="lm", formula = y ~ ns(x,5))

# be careful when flippling a plot
# note: details on transformations on the following slide
p.myplot + geom_smooth()
p.myplot + geom_smooth() + coord_flip()
p.myplot + aes(x=mpg, y=disp) + geom_smooth()

# define a base plot to illustrate transformation
p.myplot <- ggplot(mtcars) + aes(x=disp, y=mpg) + geom_point(size=4) +
    geom_smooth(method="lm", se=FALSE)

# take a look at linear regression plot
p.myplot

# apply a logarhithmic transformation
p.myplot + scale_x_continuous(trans="log", name="log(disp)")

# apply a log-transformation on the y-axis, add a linear regression and
# transform the display of the scale back with exponentation
p.myplot + scale_x_continuous(trans="log") +
    coord_trans(x="exp") +
    xlab("exp(log(disp)) = disp")

# adjust the y-scale breaks to match our original non transformed plot
p.myplot + scale_x_continuous(trans="log", breaks=seq(100,400,100)) +
    coord_trans(x="exp") +
    xlab("exp(log(disp)) = disp")

# split data to create frequency polygon for each subgroup
qplot(clarity, data=diamonds, geom="bar", fill=cut, position="dodge")
qplot(clarity, data=diamonds, geom="freqpoly", group=cut, color=cut, position="identity")

# split the data by a variable and calculate a regression for each group
ggplot(mtcars, aes(x=disp, y=mpg, color=factor(am))) + geom_point(size=4) +
    geom_smooth(aes(group=factor(am)), method="lm", se=FALSE, lty="dashed")

# use facets to split the data
p.myplot <- ggplot(mtcars) +
    aes(x=disp, y=mpg, color=factor(am)) +
    geom_point(size=4) +
    geom_smooth(method="lm", se=FALSE, lty="dashed")

# facet_wrap will wrap the specified panels
p.myplot + facet_wrap(~ am, nrow=1)
p.myplot + facet_wrap(~ am, ncol=1)

# per default the scales of the different panels will match
# it is however possible to use adaptive panes
# note: more options can be found in the documentation
p.myplot + facet_wrap(~ am, nrow=1, scales="free")

# facet_grid can be used to split by two variables
p.myplot + facet_grid(cyl ~ am)

```

```r
# it is even possible to add margin calculations
p.myplot + facet_grid(cyl ~ am, margins=TRUE)

# setup plot to illustrate annotations
p.myplot = ggplot(mtcars, aes(x = wt, y = mpg))

# plot without annotations
p.myplot + geom_point(size=4, color="purple3")

# a plot with some simple annotations
p.myplot +
    annotate("rect",
                     fill="lightsteelblue", alpha=0.4,
                     xmin=3, xmax=4, ymin=12, ymax=20.5) +
    annotate("segment",
                     size = 1, color="steelblue",
                     arrow = grid::arrow(length=grid::unit(1, "char")),
                     x=4.73, y=30.5, xend=3.8, yend=21) +
    annotate("text", label="A custom region",
                     x=4.32, y=31.2, hjust=0, vjust=0, color="steelblue", size=8) +
    geom_point(size=4, color="purple3")

# we create a function, that will calculate the coordinates for stripes that
# are contained to the given rect coordinates
# note: this involves some trigonometry and is outside
#   the scoope of this tutorial
stripesInRect <- function(angle=45, distance=0.5, xmin=0, xmax=10, ymin=0, ymax=10) {
    # this function will calculate a data.frame of vectors for a
    # stripped background in a rectangular area

    # convert angle from degree to radians
    radians <- (pi / 180) * angle

    # calculate the tangens
    tangens <- tan(radians)

    # calculate height und width of the clippling box
    height <- ymax - ymin
    width <- xmax - xmin

    # calculate the horizontal distance of the lines
    horizontalDistance = distance / tangens

    # calculate the difference of start-y to end-y for full width
    verticalDifference <- tangens * width

    # steps for the height and width
    stepsHeight = seq(from = ymin, to = ymax, by = distance)
    stepsWidth = seq(from = xmin, to = xmax, by = horizontalDistance)

    # initialize a data frame of coordinates
    # note: distance is used for distance of lines when cutting
    #   through the side of the box
    # note: we have to remove the first step from the widthsteps
    #  to avoid a duplicated start line
    data <- data.frame(
        "x1" = c(rep(xmin, times = length(stepsHeight)), stepsWidth[-1]),
        "y1" = c(stepsHeight, rep(ymin, length(stepsWidth))[-1] ))

    # define a function to calculate the endpoints
    calculateEndpoint <- function(x1, y1) {

        # calculate the maximal available width for the x range
        availableWidthRange <- xmax - x1

        if (availableWidthRange >= width) {
```

```
            # calculation of lines that start from the left side

            # calculate the maximal available height for the y range
            availableHeightRange <- ymax - y1

            # we are done if the vertical-side fits into the rect
            if (availableHeightRange >= verticalDifference) {
                return(c(
                    "x2" = xmax,
                    "y2" = y1 + verticalDifference))
            }

            # otherwise we have to adapt to the available height
            horizontalDifference <- availableHeightRange / tangens

            return(c(
                "x2" = x1 + horizontalDifference,
                "y2" = ymax))

        } else {
            # calculation of lines that start from the bottom side

            # calculate the vertical difference
            verticalDifference <- availableWidthRange * tangens

            return(c(
                "x2" = xmax,
                "y2" = y1 + verticalDifference
                ))

        }

    }

    # calculate the endpoints
    endpoints <- mapply(calculateEndpoint, data$x1, data$y1)

    # extract the endpoint coordinates
    data$x2 <- endpoints[1,]
    data$y2 <- endpoints[2,]

    return(data)
}


# calculate the pattern coordinates for our plot
pattern <- stripesInRect(angle=80, distance=0.25,
                                                 xmin=3, xmax=4, ymin=12, ymax=20.5)

# create the plot with a striped background for the annotation
# note: annotation aesthetics are not mapped but will be processed as vectors
p.myplot +
    annotate("segment",
                     size = 0.5, color="deeppink", alpha=0.25,
                     x=pattern$x1, y = pattern$y1,
                     xend = pattern$x2, yend = pattern$y2) +
    annotate("segment",
                     size = 1, color="deeppink",
                     arrow = grid::arrow(length=grid::unit(1, "char")),
                     x=4.73, y=30.5, xend=3.8, yend=21) +
    annotate("text", label="A custom region",
                     x=4.32, y=31.2, hjust=0, vjust=0, color="deeppink", size=8) +
    geom_point(size=4, color="purple3")

# define our plot to illustrate theming
p.myplot <- ggplot(pressure) +
```

```r
    aes(temperature, pressure, color=factor(temperature)) +
    geom_point(size=4)

# plotting using the default theme
p.myplot

# plotting using a black & white theme
# note: the theme does not change the aesthetics controlled by data
p.myplot + theme_bw()

# modifiying specific elements of a theme
# note: more options can be found in the documentation
#  theme modifications may require some understanding of the grid-package
p.myplot + theme(
    legend.position="top",
    legend.margin=grid::unit(1, "cm"))

# legends usually need some further specific adjustments
p.myplot + theme(
    legend.position="bottom",
    legend.margin=grid::unit(1, "cm")) +
guides(
     color=guide_legend("Temperature", nrow=2,
                                          title.position="top", byrow=TRUE))

# use combination of geoms and specific stat for bin calculation
# note: values from stat-calculations can be accessed via ..«parameter»..
ggplot(mtcars) + aes(x=factor(gear)) +
    layer(
        stat = "bin",
        geom = "linerange",
        geom_params = list(ymin=0, size=0.5, color="blue"),
        mapping = aes(ymax=..count..)) +
    layer(
        stat = "bin",
        geom = "point",
        geom_params = list(size=3, color="blue")) +
    layer(
        stat = "bin",
        geom = "text",
        geom_params = list(vjust=-0.8, color="blue"),
        mapping = aes(label=..count..)) +
    coord_flip() + theme_bw()

# we can also define the configuration in a custom function
latticebars <- function(color = "blue") {
    layer1 <- layer(
        geom = "linerange", stat = "bin",
        mapping = aes(ymax=..count..),
        geom_params = list(ymin=0, size=0.5, color=color))

    layer2 <- layer(
        geom = "point", stat = "bin",
        geom_params = list(size=3, color=color))

    layer3 <- layer(
         geom = "text", stat = "bin",
        mapping = aes(label=..count..),
        geom_params = list(vjust=-0.8, color=color))

    # note: ggplot2 elements can also be combined by creating
    #  a list of the separate components. +-symbol might
    #  throw an error if used inside of a function
    return(list(layer1,layer2,layer3, coord_flip(), theme_bw()))
}

```

```r
# create a lattice like barplot with default color
ggplot(mtcars) + aes(x=factor(gear)) +
    latticebars()

# easily change the color of the plot
ggplot(mtcars) + aes(x=factor(gear)) +
    latticebars("red") +
    xlab("Type of Gear\n") + ylab("\nNumber of Items")

# we are going to need the grid package
require("grid")

# convenience function to create multi-plot setup (nrow, ncol)
vp.setup <- function(x,y){
    # create a new layout with grid
    grid.newpage()

    # define viewports and assign it to grid layout
    pushViewport(viewport(layout = grid.layout(x,y)))
}

# convenience function to easily access layout (row, col)
vp.layout <- function(x,y){
    viewport(layout.pos.row=x, layout.pos.col=y)
}

# define three plots to be displayed together
p.a <- qplot(mpg, wt, data=mtcars, geom="point") + theme_bw()
p.b <- qplot(mpg, wt, data=mtcars, geom="bar", stat="identity")
p.c <- qplot(mpg, wt, data=mtcars, geom="step")

# setup amulti plot layout with grid (2x2 fields)
vp.setup(2,2)

# plot all graphics into our layout
print(p.a, vp=vp.layout(1, 1:2))
print(p.b, vp=vp.layout(2, 1))
print(p.c, vp=vp.layout(2, 2))
```