Introduction to R, session 3

Indrek Seppo

Nov 3, 2020

What have we learned until now?

How to create objects:

```
my_precious_object <- "ring"
```

How to remove objects

```
rm(my_precious_object)
```

How to deal with functions, they look like this:

```
function_name(arg1 = 1, arg2 = "somethingelse")
```

Remember, to get some help regarding the function you will need to use a question mark:

```
‘?‘(read.csv)
```

So let’s read in the `piaac` dataset:

```
piaac <- read.csv("http://www.ut.ee/~iseppo/piaacENG.csv")
```

Get a quick overview with commands like `summary()`, `head()` or take a look at it with `View()`.

This is a data frame. To access specific columns or variables in it, use `$`: find the average hourly income:

```
mean(piaac$earnhr, na.rm = TRUE)
```

```
## [1] 5.855744
```

Or – lets create a new variable in this dataset:

```
piaac$logincome <- log(piaac$Income)
```

There are a lot of functions in base-R, but most of them are in different libraries. To install them, you can write:

```
install.package("somelibrary")
```

or you can go to Packages pane on the down right in Rstudio and click Install. You need to install it once, update it at times, but you need to read in a package every time you reopen your RStudio.

```r
library(tidyverse)
```

We will now use the function `filter()` from tidyverse (from package called `dplyr` to be excact) to create subsets of the dataset. It is pretty much equivalent to `subset()` from base R, but has some additional controls which would not let us make some stupid mistakes in it.

For subsetting we need to know some logic in R. The main things would be:

```r
< #is smaller than
> #is bigger than
<= #smaller or equal
>= #bigger or equal
== #is equal to
!= #not equal to
%in%c("uhhuu", "ahhaa") # one of "uhhuu" or "ahhaa"
!is.na() #only the values that are not available
```

So all the men in the dataset:

```r
onlymen <- filter(piaac, gender == "Male")
```

Only the ones who'se health is not NA:

```r
hashealth <- filter(piaac, !is.na(Health))
```

Or those, whose health is either "Excellent" or "Very good" (btw – use `unique()` to find out what kind of values are present in the data)

```r
healthypeople <- filter(piaac, Health %in% c("Excellent", "Very good"))
```

The main points from ggplot

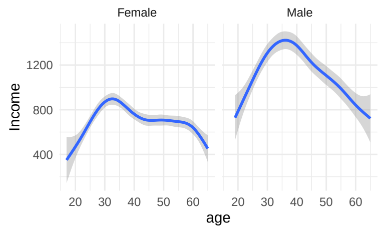
There are three important concepts: geometrics (**geom_*()**), aesthetics** aes() **and facets** (facet_wrap() **and** facet_grid()**) – if you manage these, you can do everything you need to understand your data. Everything else is for presenting your data and you will look this up anyway.

A typical plot in the ggplot language looks something like this:

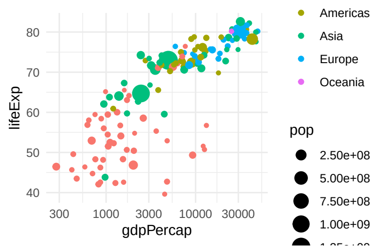
```r
ggplot(data=mydataset, aes(x=firstvariable, y=secondvariable))+
  geom_something(color=groupingvariable)+
  facet_wrap(~othergroupingvariable)
```

Your turn:

- Using the piaac dataset we read in previously:
- Create an age-earning profile for Estonia with the new piaac data, using the `Income` as an earning indicator. Age-earning profile shows the average earnings for every age, use the smoother here (`geom_smooth()`) to get the smoothed profile. Think for a moment – you only need to set what is connected to x, what to y. And then add `geom_smooth()`.
- Then facet it by gender. How did we add facets?

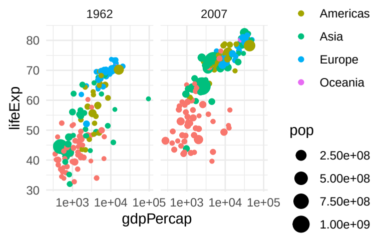


2) Lets recreate one classic visualisation. Install a package called `gapminder` and load it in. You can now use a dataset called gapminder (try it: `summary(gapminder)`).



- take a subset of this data leaving only year 2007 in (you can use `dplyr::fiter()` or the base `subset()` function).
- create a graph where `gdpPercap` is mapped to x, `lifeExp` is mapped to y and add points as a geom.
- map the size of the points to variable `pop`
- map the color of the points to variable `continent`
- add three more lines: `scale_x_log10()` – this will change the x-scale to be logarithmic; `scale_size_area()` – this will map the population to the area of the point, and `theme_minimal()`

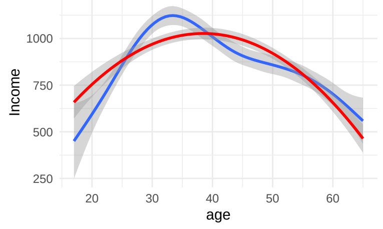
2) Now recreate the previous graph with years 1962 and 2007 (so make a new subset of the data – the year can now be either 1962 or 2007). Use previous graph as a starting point and add faceting to it using `year` as a faceting variable.



As a side-note – `geom_smooth()` can be really helpful for analysing if a particular parametric modelling choice is a good idea or where would it err the most. The relationship between age and earning is usually modelled with a quadratic function. We can add another smoother and specify that it would find the best quadratic function it can (identical to running a regression with a quadratic term) by adding:

```
geom_smooth(method = "lm", formula = y ~ poly(x, 2), color = "red")
```

This term – `poly(x,2)` is one of the ways to add polynomials when creating regressions with polynomials in R (you can also add x and x^2, they would be highly correlated though).

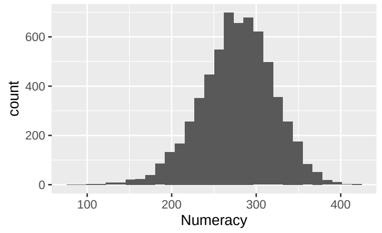


You see that (unless we are overfitting with our loess-curve) the quadratic function would overestimate the wages of youth in Estonia and understimate the 30-40 year olds.

Some additional geoms and concepts

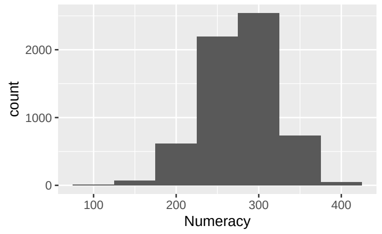
Some of the graphs need only one variable. If we want to show the distribution of some variable, we can use the histogram **geom_histogram()**:

```
library(ggplot2)
piaac <- read.csv("piaacENG.csv")
piaac$logincome <- log(piaac$earnhr)
ggplot(data = piaac, aes(x = Numeracy)) +
  geom_histogram()
```



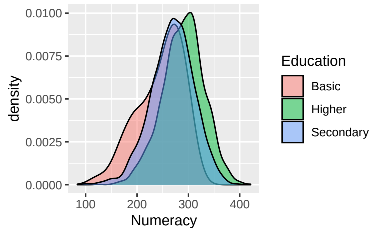
It creates 30 bins by default, we can change it with a **binwidth**-parameter:

```
ggplot(data = piaac, aes(x = Numeracy)) +
  geom_histogram(binwidth=50)
```



The same type of information can be given with density function – **geom_density()**, we can again group data by color (both fill and color can be used):

```
ggplot(data = piaac, aes(x = Numeracy))  +
  geom_density(aes(fill=Education), alpha=0.5)
```



We are seeing a usual problem here – the levels of education are in the alphabetical order. Lets change them to be in a logical order. You can do this with factor variables:

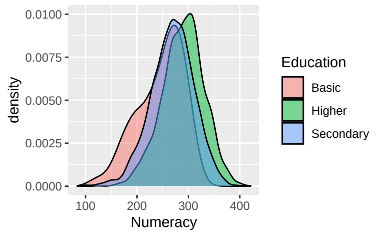
```
levels(piaac$Education)
```

```
## NULL
```

```
library(forcats)
piaac$Education <- fct_relevel(piaac$Education, levels = c("Low", "Medium",
    "High"))
```

```
## Warning: Outer names are only allowed for unnamed scalar atomic inputs
```

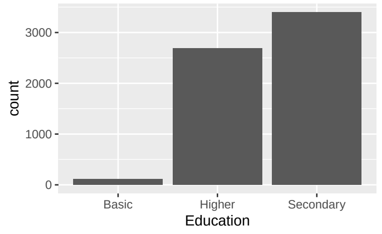
```
## Warning: Unknown levels in ‘f‘: Low, Medium, High
```



```
ggplot(data = piaac, aes(x = Numeracy))  +
  geom_density(aes(fill=Education), alpha=0.5)
```

If we have a discrete variable, we can present it with **geom_bar()**. It will, by default, count the nr of occurrencies:

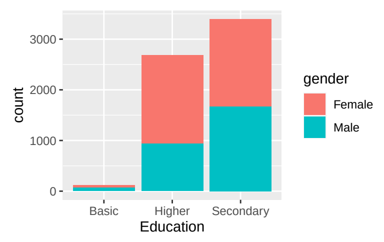
```
ggplot(data = piaac, aes(x = Education)) +
  geom_bar()
```



Position: stack, dodge, fill, jitter, jitterdodge

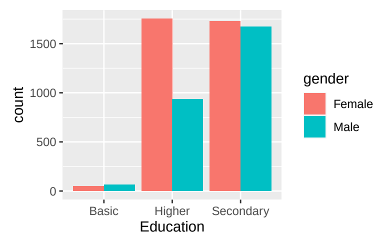
What happens if we group data in the bar chart?

```
ggplot(data = piaac, aes(x = Education)) +
  geom_bar(aes(fill=gender))
```



It will stack the data! This is not what we usually want, we would like the bars to be next to each other. And we can do it with the position parameter:

```
ggplot(data = piaac, aes(x = Education)) +
  geom_bar(aes(fill=gender), position="dodge")
```



Or we can use a more complicated way, but this would give us greater control:

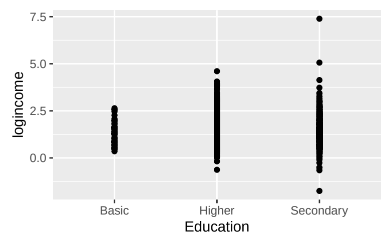
```
ggplot(data = piaac, aes(x = Education)) +
  geom_bar(aes(fill=gender), position=position_dodge(width=0.5))
```

There is also **position_stack()** available (this is what it does by default). You can also try **position_fill()** with bar charts.

What we use a lot in practice is **position_jitter()** - this helps us to avoid some overplotting.

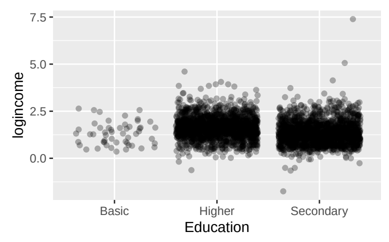
Lets try to visualize the logincome by education level:

```
ggplot(data = piaac, aes(x = Education, y=logincome)) +
  geom_point()
```



You see a lot of overplotting, it would be much better to shake things up a bit:

```
ggplot(data = piaac, aes(x = Education, y=logincome)) +
  geom_point(position="jitter", alpha=0.3)
```

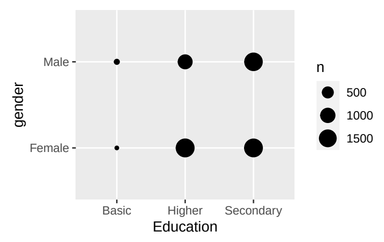


In fact, there is a convenience geom – geom_jitter() that does just this:

```
ggplot(data = piaac, aes(x = Education, y=logincome)) +
  geom_jitter(alpha=0.3)
```

If we have two nominal variables, we can use **geom_count()**:

```
ggplot(data = piaac, aes(x = Education, y=gender)) +
  geom_count()
```



Scales

x ja y

Everything that is connected with aesthetic, is connected with a scale - x, y, size, color etc. All the scales have some common features – e.g **breaks** and **labels**. Lets try it:

```
ggplot(data = piaac, aes(x = Education, y=logincome)) +
  geom_jitter(alpha=0.3)+
  theme_minimal()+
  scale_y_continuous(breaks=c(0,2,3), labels=c("small", "average", "big"))
```

R will automatically add some space in the beginning and end of the **x** and **y** scale. We can control it with a parameter **expand=c(0.1, 0.1)**, with the first number showing how many percantages and the other number how much in the nominal terms is added.

NB! There is one particular thing you should never use: you can set limits in **scale_x_continuous()**, but do not do it ever. It will just cut all the data above and below the limit and the smoothers etc will then be meaningless. Use **coord_cartesian()** to zoom into where you want to zoom in.

We were dealing with the continuos scales up until now, but there are others – **scale_x_discrete()** for discrete x-axis, **scale_x_date()** and **scale_x_datetime()**, to show the dates with some convenience() **date_breaks="1 months"**, will show all the months etc).

Colorscales and legends

GGplot adds the colors automatically, but we can tell it what kind of colors to use. There are two different scales again - for continuous and discrete data. You can see all the available scales bywriting `scale_color` (või `scale_fill`) into the console and hit the tabulator.

One very usable color scale is **scale_color_brewer()** (if your data is discrete). Take a look at the colorbrewer2 website: `http://colorbrewer2.org/`. You can choose the pallettes that are color-blind safe etc.

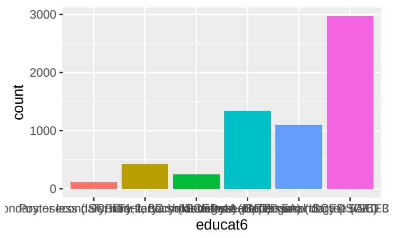
```
ggplot(data = piaac, aes(x = educat6))+
  geom_bar(aes(fill=educat6))+
  scale_color_brewer("Education level", palette="Pastel1")
```

Note, that *color brewer* will usually give you up to 8 different colors

The place to go if you have any questions about the legends is: `http://www.cookbook-r.com/Graphs/Legends_(ggplot2)/`.

You will usually have the legend automagically, the legend can be removed with the function `guides()`:

```
ggplot(data = piaac, aes(x = educat6))+
  geom_bar(aes(fill=educat6))+
  scale_color_brewer(palette="Pastel1")+
 guides(fill=FALSE)
```



Manual scales

We will sometimes want to predefine which colors to use. E.g. if we want to use the same colors in the subsequent graphs.

We will need named vectors for this:

```
ggplot(data = piaac, aes(x = Education))+
  geom_bar(aes(fill=Education))+
  scale_fill_manual(values=c("Low"="red",
                             "Medium"="blue",
                             "High"="orange"))
```

You can use names, you can use RGB-codes etc. And it is exactly the same with other manual scales – if you want to predefine point shapes, you would use `scale_shape_manual()`

Your turn:

We downloaded the gdp per capita PPP data before by:

```
gdp.pc <- wb(indicator = "NY.GDP.PCAP.PP.KD", POSIXct = TRUE)
```

1) Select three "countries" – the world, Estonia and some other country (e.g. your native country) from the data (using `subset()` or `filter()`)[1].

2) create a graph where x-axis is the date (date_ct), y-axis is the value, and countries are grouped by color

3) change the color scheme so, that one of the countries is orange, another one blue and the third one is green.

4) create a second graph where each country has its own facet.

[1] You can check the names of countries in this dataset using `unique(gdp.pc$country)`, note that they are not in alphabetical order – the aggregates are in the beginning and then the countries come in alphabetical order.

Statistics in ggplot

There is one more concept in grammar of graphics, called statistics. We do not need to specify these often, as each geom has its default statistics and it usually just works, but there are couple of cases were we need them.

Lets look at the help file of geom_bar(). It will tell us that the default statistics used is count (stat="count"). What this means is that by default it will count all the cases as we saw previously:

```
ggplot(piaac, aes(x = Education)) + geom_bar()
```

But what if we do not want to count the ocurrences, but have the following data precomputed for us:

```
averages <- data.frame(edlevel = c("High", "Medium", "Low"), nrOfPeople = c(10,
    20, 30))
averages
```

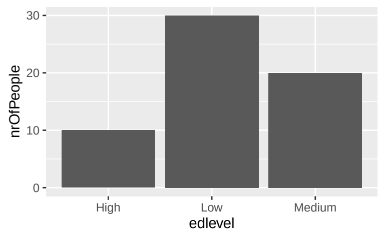
```
##   edlevel nrOfPeople
## 1    High         10
## 2  Medium         20
## 3     Low         30
```

Imagine we want the bars to represent the number. The way we used geom_bar() previously wont work:

```
ggplot(averages, aes(x=edlevel))+
  geom_bar()
```

What we need is to change the stat argument like this:

```
ggplot(averages, aes(x=edlevel, y=nrOfPeople))+
  geom_bar(stat="identity")
```



Or take geom_boxplot as an example – by default it uses stat="boxplot" – finds the medians and quartiles by itself from the data. But if you happen to have agregated data, you can use your own values.

Lets load a dataset called wagebirthyear:

```
earnings <- read.csv("http://www.ut.ee/~iseppo/wagebirthyear.csv")
```

Take a look at this data. It includes percentiles of monthly salaries earned in 2011 by birth year. Lets first create a variable age into this dataset:

```
earnings$age <- earnings$year - earnings$birthyear
```

Lets see if you manage to make the following graph. The grey area shows 25.-th to 75.-th percentile for each age group. So 25% of the people are below the grey area (for their age), 25% are above it and 50% are inside the area. The dataset used is `earnings`, x is connected to `age` variable in the dataset, the grey area is done by `geom_ribbon()` which requires `x`, `ymin` and `ymax` to be set as aesthetics. I have also told it with additional parameters that `fill="grey"` and `alpha=0.3`. Then there are two lines – average and median. For the average I have set the color to be blue, for the median I have set the color to be orange. At the very end I have added theme_minimal().

Lists

List is a data structure that we will meet quite often. Data frame consists of vectors of the same length, list can include whatever.

For example[2]:

[2] Note that we give names to list elements. R will not do it automatically, as it does for data frames.

```
name <- "Masha"
surname <- "Mishka"
length <- 1.65
peripherals = data.frame(nr.of.fingers = c(5, 5), nr.of.toes = c(5, 5))
mashadata <- list(first.name = name, surname = surname, length = length,
    peripherals = peripherals)
class(mashadata)
```

```
## [1] "list"
```

```
mashadata  #let us look into it:
```

```
## $first.name
## [1] "Masha"
##
## $surname
## [1] "Mishka"
##
## $length
## [1] 1.65
##
## $peripherals
##   nr.of.fingers nr.of.toes
## 1             5          5
## 2             5          5
```

Lists can contain lists that can contain lists etc.

To address a single element of list we can use the $-sign:

```
mashadata$first.name
```

```
## [1] "Masha"
```

Or double straight brackets (the name of the element has to be quoted then):

```
mashadata[["first.name"]]
```

```
## [1] "Masha"
```

You can in fact use single straight brackets, but they will return a list, not the initial data type (the data.frame, which we included in the list), which you will want to get in practice.

```
class(mashadata["peripherals"])
```

```
## [1] "list"
```

```
class(mashadata[["peripherals"]])
```

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

If we want to access the vector **nr.of.fingers** from the data frame **peripherals** from the list **mashadata**, then we can do it in the following way:

```
mashadata[["peripherals"]]$nr.of.fingers
```

```
## [1] 5 5
```

but not like this:

```
mashadata["peripherals"]$nr.of.fingers
```

```
## NULL
```

There is one thing to note when dealing with lists. Data frame will not let you include two columns with the same name. List would not care less:

```
list2 <- list(a = 3, b = 4, a = 5)
list2$a  #Not even a warning!
```

```
## [1] 3
```

The reason we pay so much attention to lists, is that a lot of the results R will give you, are in the form of lists. Take t.test() – t test compares two samples and asks if it is statistically viable that they come from the same population, that the differences could be just thanks to sampling error. If we only give it one vector as an input, it will tell us whether the mean is statistically different from 0.

Lets do a t-test for **Literacy**-variable (measuring literacy levels) in our **piaac** dataset[3]:

[3] Note that we are using the piaac data incorrectly here! In reality there are ten different plausible values given for literacy levels, as there is considerable uncertainty involved. Do do it correctly you should use all the information – this is possible with a package called **svyPVpack**. Neither do we use the weights here.

```
t.test(piaac$Literacy)
```

```
##
##  One Sample t-test
##
## data:  piaac$Literacy
## t = 511.9, df = 6203, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
```

```
##  278.3766 280.5169
## sample estimates:
## mean of x
##  279.4468
```

It will give us 95% confidence intervals for the population. Meaning – considering this sample, believing that this is a random sample, we would believe that the correct average for the whole population should be inside this interval with quite a high certainty. The average for the sample is 275,64, we believe, that the average for the whole population should fall somewhere between 275. . . 277.

`t.test()` shows us the results, but if we save it to a data object, we will find out, that it actually produces a list. Lets do it:

```
result <- t.test(piaac$Literacy)
class(result)
```

```
## [1] "htest"
```

```
typeof(result)
```

```
## [1] "list"
```

```
str(result)
```

```
## List of 10
##  $ statistic  : Named num 512
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 6203
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 0
##  $ conf.int   : num [1:2] 278 281
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num 279
##   ..- attr(*, "names")= chr "mean of x"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "mean"
##  $ stderr     : num 0.546
##  $ alternative: chr "two.sided"
##  $ method     : chr "One Sample t-test"
##  $ data.name  : chr "piaac$Literacy"
##  - attr(*, "class")= chr "htest"
```

Your turn:

- confidence intervals are inside this list as a vector called **conf.int**. Can you access it?

1) create a variable called **lower**, and assign it the lower value of conf.int (the first value in this vector)
2) create a variable called **upper**, and assign it the upper value of conf.int (the second value in this vector)

Functions

Writing a custom function or command in R is extremely simple.
Take a look at the example:

```
timesthree <- function(x) {
    # we name the function and say that its input will be named x
    result <- x * 3  #we do some stuff
    return(result)  #we will return the answer
}
```

You now need to make R aware of the new function - just select it and click Run (or cntrl+enter).

Lets find out how it works:

```
timesthree(4)

## [1] 12

timesthree(c(3, 4, 5))

## [1]  9 12 15
```

Remember, the functions could have default values:

```
multiplydivide <- function(x, multiplier = 3, divisor = 1) {
    result <- x * multiplier/divisor  #use the input
    return(result)  #return the result
}
```

If we will not give any values for **multiplier** or **divisor**, then it will use the default values. But we can change them:

```
multiplydivide(x = 4)

## [1] 12

multiplydivide(x = 4, multiplier = 1, divisor = 2)

## [1] 2
```

Minimal about conditional statements

You can write conditional statements in R in the following way

```
if (condition) {
    whattodo
} else {
    dosomethingelse  #not required
}
```

For example[4]:

[4] You do not need to use the square brackets, if the `whattodo` is a single line. For more lines you need the brackets.

```
nameOfTheBear <- "George"  #let us have a bear named George

if (nameOfTheBear == "George") friendOfABear <- "Tom"
if (nameOfTheBear == "Bill") friendOfABear <- "Mary"

friendOfABear  #who is the friend of the bear?

## [1] "Tom"
```

Your turn:

1) create a function called **subtract**, which takes as an input two data objects (for examplem *a* and *b*) and subtracts one from the other.

2) create a function called **findlower**, which takes in a vector, does a t-test and returns the lower bound.

3) create a function called **findupper**, which takes in a vector, does a t-test and returns the upper bound.

4) **Only for people who are too quick:** Rewrite the previous function, that found the upper bound so, that you can give it additional parameter specifing which bound to return (by default "lower"), and if it is "lower", then it returns the lower bound, if it is "upper", it returns the upper bound.