Introduction to R, session 2

Indrek Seppo and Nicolas Reigl

‘Feb 10, 2018

From last time

We learned some of the most basic things last time:

a) how to assign values to data objects:

```
mydata <- 5
```

b) how to delete data objects:

```
rm(mydata)
```

c) we learned that vectors can created with `c()`:

```
myvector <- c(2, 3, 4)
```

Vectors can consist of one type of data – be it numeric, logical, dates, strings (character type in R) or something else.

d) we learned that the most usual form of data is data frame, which consists of vectors. To access particular variables (or vectors) in the data frame, we will need to use `$` sign:

```
mean(mydataframe$myvariable)
```

we can add variables or columns to the data frame by:

```
mydataframe$newvariable <- something
```

and we can delete variables by assigning them NULL:

```
mydataframe$stupidvariable <- NULL
```

e) we learned that there are functions in R. This in turn means that we can do even more with R than just basic algebra or creating vectors and data frames! We can even find a mean of a variable:

```
mean(mydataframe$somevariable)
```

Functions usually need some arguments or parameters. Some of these arguments have default values, so you don’t have to change them, but you may if you so wish. The arguments are separated by commas:

```
functionname(arg1 = something, arg2 = somethingelse)
```

To see, what arguments a function can take, look at the help:

```
'?'(functionname)
```

f) We looked at how to get an initial overview of our data:

```
summary(mydataframe)
str(mydataframe)
head(mydataframe)
tail(mydataframe)
names(mydataframe)
```

g) We looked at how to install and load additional packages. You need to install them once[1], update them every now and then, but you need to load them every time you have restarted R:

[1] Actually after every major R update – every couple of years.

```
library(somepackage)
```

Your turn[2]:

[2] YOUR TURN

- Read in the data `piaacest.csv` from from `http://www.ut.ee/~iseppo/piaacest.csv` (it is different than the one we used last time!). Note that you need to put the quotation marks around the filename/URL when passing it to `read.csv()`. Write it into an object named `piaac`.
- Find the mean and median hourly income (the variable name is `earnhr`) in the data, ignoring the NA-s. For median you can use the function `median()`
- Create a new variable in the data frame called logincome from this hourly income variable using the `log()` function[3].
- What is the correlation between numeracy score (`pvnum1`) and literacy score `pvlit1`) in the data[4]? As there are missing values, you will need to tell it to *only use complete observations ("complete.obs")*, `na.rm=TRUE` does not work here. You will find the parameter for this in the help system.

[3] NB! `log()` will take a natural logarithm in R.

[4] Note that I am not using this data correctly here – in fact PIAAC has a number of plausible values given for these scores as it does not measure the scores precisely. If you want to use this data for scientific purposes you will need to do it slightly differently to get the correct standard errors etc.

Getting data in

We have used the `read.csv()` function from base R (and `read.csv2()`, which just has some different default values). There are a number of additional parameters there, that may be useful:

- `na.strings=c(NA, ".", " ", "")` – you can specify how are the not-available values encoded in your dataset. NA means that there is nothing, but sometimes "." or a space is used[5]. With this parameter you can let R know of it, and it saves you from some additional data-cleaning.

[5] Sometimes you have to specify that an empty string (`""`) is NA as well.

- `strip.white=FALSE` – change this to TRUE to get rid of the whitespace before and after the names of the variables (this is rather common if you export smth from Excel).
- you will use `fileEncoding=` parameter a lot in your work.

Interfaces to online databases

You can access many online databases directly from R. For Eurostat just install and load the **eurostat** package. You just need to know the name of the table you are interested in:

```
library(eurostat)
eurostatgdp <- get_eurostat("tec00114")
head(eurostatgdp, n = 3)
```

or (to get the full labels)

```
eurostatgdp <- get_eurostat("tec00114", type = "label")
head(eurostatgdp, n = 1)
```

For OECD datasets, install and load the **OECD** package[6].

[6] The server seemed to be down when I tested it yesterday. You can find additional info about the package at https://github.com/expersso/OECD

```
library(OECD)
search_dataset("FDI")
FDIdata <- get_dataset("FDI_FLOW_CTRY")
head(FDIdata)
```

For World Bank data, use package called **wbstats**.

```
library(wbstats)
wbsearch("GDP per capita, PPP")
gdp.pc <- wb(indicator = "NY.GDP.PCAP.PP.KD", POSIXct = TRUE)
```

or **WDI** (as in World Development Indicators) – https://github.com/vincentarelbundock/WDI.

For IMF data check out the **imfr** package (for usage check https://github.com/christophergandrud/imfr).

There are many more datasources covered – finance data (stocks etc), there are packages that allow you to access data from statistical offices for many countries etc.

Accessing specific values in the vector and the data frame

There are couple of ways to access specific values in the datasets. One way is with square brackets.

Lets create a quick demo-dataset:

```
mydata <- data.frame(animal = c("elephant", "bear", "cat"), length = c(5,
    3, 2), weight = c(10, 5, 1))
```

The first value of a vector mydata$length:

```
mydata$length[1]
```

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

Btw – we can overwrite it too:

```
mydata$length[1] <- 5
mydata
```

```
##     animal length weight
## 1 elephant      5     10
## 2     bear      3      5
## 3      cat      2      1
```

The length in the first row is now 5!

The second and the third value

```
mydata$length[c(2, 3)]
```

```
## [1] 3 2
```

Note that we are using a vector to tell R that we want two values. This is most common in R and the reason we deal with the vectors so much – if we want to tell R to use multiple values for the same parameter, we will always use a vector of values.

All the values except the second one (using a negative index)

```
mydata$length[-2]
```

```
## [1] 5 2
```

To access some specific values inside a data frame, we will again use the square brackets. Now it needs two arguments – first the nr of the row, then the nr of the column:

```
mydata[1, 2]
```

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

If we leave one parameter blank, we will get entire row or column:

```
mydata[1, ]  #entire first row
```

```
##     animal length weight
## 1 elephant      5     10
```

```
mydata[, 3]  #entire third column
```

```
## [1] 10  5  1
```

We can again use vectors:

```
mydata[c(2, 3), ]  #second and third row
```

```
##   animal length weight
## 2   bear      3      5
## 3    cat      2      1
```

But it is rather rare, that we access the data by position. We usually use conditions – we want to select all men, or everyone over 45 etc. There is a second way of subsetting data that helps us here: selecting data by logical vector:

```
mydata$animal[c(TRUE, FALSE, TRUE)]
```

```
## [1] elephant cat
## Levels: bear cat elephant
```

We selected first and third value by this. If only there was an easy way to create those logical vectors...

Conditional statements and logic in R

Lets take a "is bigger than" condition as an example. The operator is >, and it gives us a logical value of TRUE or FALSE as an answer:

```
3 > 4
```

```
## [1] FALSE
```

Remember that in R pretty much all the functions are "vectorized", so if we apply them to a data vector it will test every single item one by one and return a logical vector:

```
mydata$length > 2
```

```
## [1]  TRUE  TRUE FALSE
```

The first value is higher than 2, the second value is higher than 2 and the third value is not higher than 2. Combine this with our previous knowledge, and we can select all the names of the animals, whose length is higher than 2:

```
mydata$animal[mydata$length > 2]
```

```
## [1] elephant bear
## Levels: bear cat elephant
```

Or we can select all the entire rows, where length is > 2:

```
mydata[mydata$length > 2, ]
```

```
##     animal length weight
## 1 elephant      5     10
## 2     bear      3      5
```

In a moment we will see an easier way to do all this. I showed you this way for two reasons:

1) a lot of code you encounter in the wild is using this logic, and when you get it, it is a very quick way to do stuff
2) this is the quickest way to change values of some variables that meet some conditions. For example:

```
mydata$length[mydata$animal == "elephant"] <- 100
```

I am here changing the length in all the rows, where the variable `animal` has a value of "elephant".

The basic comparison operators are the following:

Comparison

```
x <- 3
z <- 4  #lets create x and z
x > 3  #is bigger than
x >= 3  #bigger or equal
z < 4  #smaller than
z <= 4  #smaller or equal
x == z  #equality
x != z  #not equal
is.na(x)  # is it NA
```

Logic

```
x > 2 | z > 5  #logical OR
x > 2 & z > 5  #logical AND
```

You will encounter operators '&&' ja '||' in the wild, but be aware that these compare only the first element of vectors! Do not use them unless you know what you are doing.

Another frequently used operator is `%in%`, asking if value is part of a set:

```
4 %in% c(2, 3, 4, 5)
```

```
## [1] TRUE
```

```
myanimals <- c("chicken", "duck", "bear")
"frog" %in% (myanimals)
```

```
## [1] FALSE
```

We will use it frequently – *e.g.* if we have data that has a lot of age groups in it, and we want to filter out people who are either in age group "50-59", "60-69" or "70 and older".

Filtering data

There are easier and harder ways to filter data in R. An easy way is to use `dplyr::filter()` (this means a `filter()` function provided by a package called `dplyr`), we will take a look at it later.

The base-R way is either using `subset()` or the straight brackets with logical conditions. `subset()` needs a dataframe to be filtered and the condition according to which you want to filter it.

We'll use the piaac dataset that was loaded previously.

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

Take a quick look at it using `summary()` or `str()`. So there is a variable called `edlevel3`. Lets look inside:

```
summary(piaac$edlevel3)
```

```
##   High    Low Medium   NA's
##   2733   1409   3445     45
```

Now lets filter out only the people who have medium level of education:

```
highschool <- subset(piaac, edlevel3 == "Medium")
```

`summary(highschool)` will show us if everything went alright.

These conditions can be added to each other. Lets find people who have medium level of education and who are living with their spouse. We need to use logical AND here:

```
highschool2 <- subset(piaac, edlevel3 == "Medium" & livingwithspouse ==
    "Yes")
```

And we can now do statistical analysis for this particular group:

```
mean(highschool2$pvnum1, na.rm = TRUE)
```

```
## [1] 268.1483
```

Or lets find everyone whose education level is either High or Medium:

```
sample3 <- subset(piaac, edlevel3 %in% c("High", "Medium"))

summary(sample3)
```

Removing NA-s

Lets remove everyone whose edlevel3 is NA from the dataset. We now have to use the `is.na()` function and actually the negation of it:

```
piaac <- subset(piaac, !is.na(edlevel3))
```

Lets read in the gdp data again and create the difference between first and latest estimate:

```
gdp <- read.csv("./gdpestimate.csv")
gdp$difference <- gdp$firstEstimate - gdp$latestEstimate
```

Your turn:

- Find what was the average difference of between the first and latest estimate of the GDP between years 2012 and 2014
  - create a subset of this data where year is either 2012, 2013 or 2014 (you can do it in a number of ways – by using %in%, or two conditions (year>2011 and year<2015 bind together by a logical AND))
  - find the mean of variable difference for this subset
- Find what is the average hourly income in the piaac data set for those who live with their spouse (variable names **livingwithspouse**).
- Find the average hourly income for those, whose age (**AGE_R**) is higher than 34 and whose education level (**edlevel3**) is Medium.

The hadleyverse (aka tidyverse)

We will now enter the hadleyverse or tidyverse – a slightly different way of working in R. These are the packages created by Hadley Wickham, a superstar in the R world[7]. We will, of course use other packages as well, but you can do 90% of your stuff inside tidyverse.

We will start with ggplot2 – implementation of **grammar of graphics**. An old idea, and it has actually been tried before[8], but for some reason never got traction before the ggplot package. It is now being implemented everywhere (including Python). We will then continue with **tidyr** – this helps us to convert data from long format to wide format, and **dplyr** – which helps us with all the other data manipulation tasks.

[7] You will find that the modern python data packages are copying a lot of ideas from him – and vice versa, so the logic you learn here will be applicable in other places

[8] The creator of grammar of graphics is Leland Wilkinson, a top statistician, who published a book by the same name in 1999. He is also the guy who designed the graphics system of SPSS and recently worked some time for Tableu.

ggplot2

Data visualisation has two means: 1) it is one of the best ways to understand your data, it is thus part of analysis process itself; 2) it is one of the best ways to communicate your results.

R has a number of graphics systems. There are graphing functions in the base R (**plot()** function), there is a package called **lattice** and many packages that can do some very specific types of graphs. We will only look at ggplot, as

this is the most used one, it has actually been one of the reasons why many people have turned to R.

The grammar of graphics started from a question – what is the absolute minimum you need to know to produce a graph? How to describe a graph with minimum nr of words? Look at the following graph:

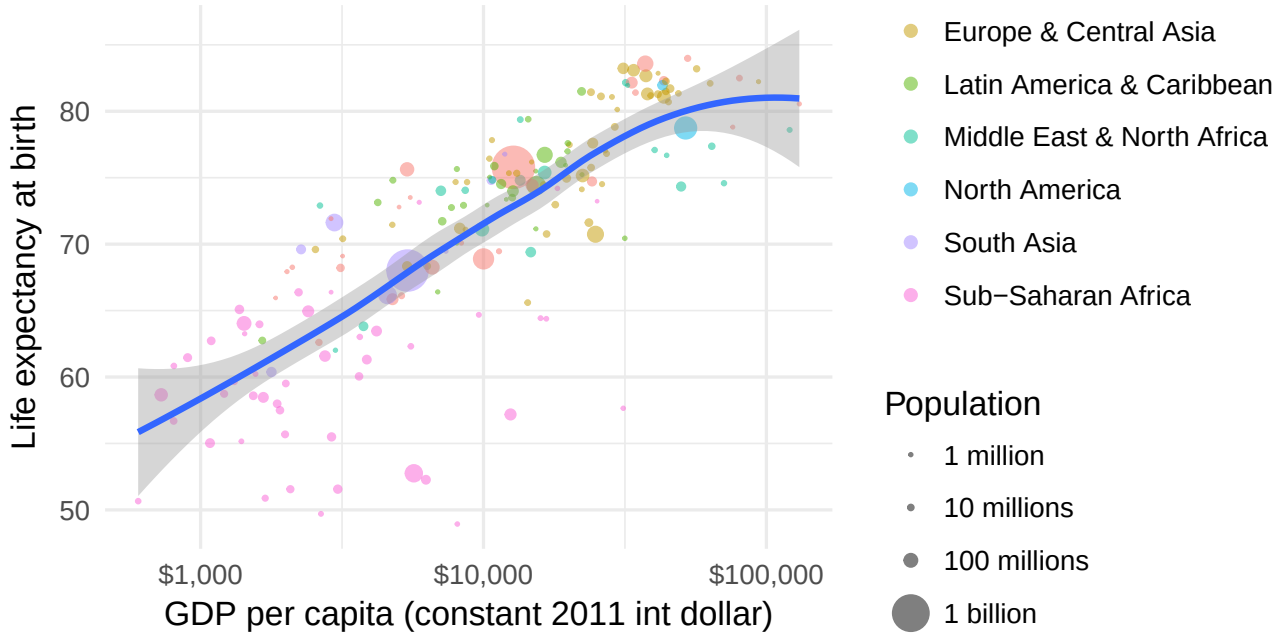


Figure 1: An example of ggplot

This is this graph in the ggplot language:

```
ggplot(data=gdplife, aes(x='GDP per capita (constant 2011 int dollar)', y='Life expectancy at birth'))+
  geom_point(aes(size=Population, color=region), alpha=0.5)+
  geom_smooth()+
  scale_x_continuous(trans="log10", labels = dollar_format())+
  scale_size_area(breaks=c(1000000, 10000000, 100000000, 1000000000),
                  labels=c("1 million", "10 millions", "100 millions", "1 billion"))+
  theme_minimal()
```

We need to know:

1) what do we want to show? Points, lines, smoothers, boxplots etc. In grammar of graphics these are called geometrics, **geom** in short. In this graph we have included two types of geometrics – first points (**geom_point()**), then a smoother (**geom_smooth()**).

2) what kind of data should the geometrics use. If we want to show a point, we will need to give at least x and y coordinates. But we can also tell it to tie some data to the color of the point. Or size. These kinds of connections

are calles aesthetics, or **aes** in short. Here we tell it to take the x-coordinate from a variable called `GDP per capita (constant 2011 int dollar)` and the y coordinate from the variable called `Life expectancy at birth`. We also tell it to connect the size of the point to a variable called `Population` and the color of the point to a variable called `region`.

3) would we like to add any statistics – averages, quantiles etc (or what kind of statistics should the geometrics use). These are called **stats** in ggplot. We do not explicitly add them often, they come by default with the geoms.

4) How are the scales defined – should the x-scale be linear or logarithmic? Here we tell it to be logarithmic. Should the population define the radius or the area of the point? Here it is an area. What should be included in the legend.

5) Whether we want to slice the graph to facets (not done here)

6) Which kind of coordinate system should we use – **coord** in ggplot (e.g **coord_cartesian()** – the default, but there are also **coord_polar()** and others available).

7) How should it all look like – **theme** in ggplot, we use minimalistic theme – **theme_minimal()** here.

First example

Lets go through the graphing process from the start to the end through one example.

Lets look at how the GDP per capita for the entire world changed during our epic financial crisis.

First download the gdp per capita PPP data from the World bank:

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

If we take a look at the data, we will see that there are a lot of countries included. We will need to filter out the numbers for the whole world. The problem with R is that it would not show us anything smart with character type variables. So `summary(gdp.pc$country)` will not help us in any way. There are two options. One is to use the function `unique()`, which only shows us the unique levels of a variable:

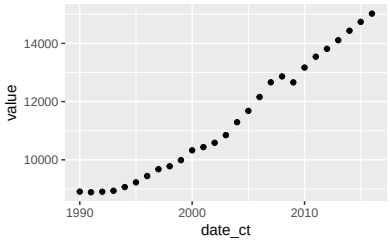
```
unique(gdp.pc$country)
```

Ok, there is a country called “World” in the dataset. Lets create a new dataset with only the numbers for the World in it:

```
gdp.pc.world <- subset(gdp.pc, country == "World")
```

and create our first graph:

```
library(ggplot2)
ggplot(data=gdp.pc.world)+
  geom_point(aes(x=date_ct, y=value))
```



What did we just do? We told ggplot to use `gdp.pc.world` dataframe as a data source. Then we asked it to add a layer of points, where x is mapped to `date_ct` variable in this dataframe and y to `value` variable. Ggplot created everything else for us. Note that if you want to map variables from the data frame to aesthetics, you will **always need to do it inside `aes()`**.

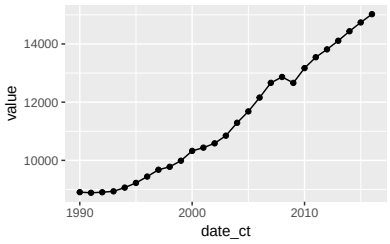
We can add another geoms – lets add a line connecting the dots:

```
ggplot(data=gdp.pc.world)+
  geom_point(aes(x=date_ct, y=value))+
  geom_line(aes(x=date_ct, y=value))
```

Creating a ggplot means adding layers to the plot. Note that there is a +- sign after the first lines. Try to remove the + and see what happens (you will forget it a lot).

We see that both of the geoms use the same data for x and y. Everything that is written in the initial `ggplot()`-call, will be inherited by later lines, so we can rewrite it:

```
ggplot(data=gdp.pc.world, aes(x=date_ct, y=value))+
  geom_point()+
  geom_line()
```



To save it, we will need to first write the graph into an R object:

```
p.gdp <- ggplot(data=gdp.pc.world, aes(x=date_ct, y=value))+
  geom_point()+
  geom_line()
```

We can then save ggplot objects with the function `ggsave()`:

```
ggsave(p.gdp, filename = "worldgdp.png", height = 4, width = 4, scale = 2)
```

ggsave() wants to know the name of the file. it will choose the filtype automatically form it. It can do jpg, png, pdf, on Windows it can create wmf (can be useful when exporting stuff to MS Office). It is a good practice to give it the height and width arguments (they are in inches), you will frequently need the scale-argument, try to play with it. There are a some other parameters like **dpi=** to set the dots per inch (it is by default 300 - good enough for printing).

Your turn:

We use the piaac data that we have downloaded previously.

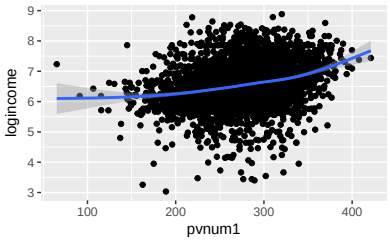


Figure 2: Expected result

- Create a ggplot graph where you use this data, map `x` to numeracy score (variable `pvnum1`), y to logincome and add two geoms – first `geom_point()` and then `geom_smooth()`. You should end up with something like the figure in the margin.

Parameters and their inheritance

Everything we write in the first ggplot() function call is inherited by the next levels – so these will be used by default, but we can change them if we would like to.

```
ggplot(data=piaac, aes(x=pvnum1, y=logincome))+
  geom_smooth()+
  geom_smooth(aes(x=PVLIT1), color="red")
```

I told it to add a second layer, which shows the relationship between literacy and logincome (and I also told ggplot to color it red).

We could use data from a different dataframe, just adding `data=somethingelse` to a layer.

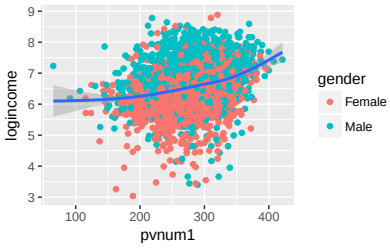
Lets look again at the graph you created, which had both points and a smoother on it. We can use the normal R help system to see, what parameters can be changed and how.

```
?geom_point
```

In the **Aesthetics** section it tells us that **geom_point()** can do with the following aesthetics: x (required), y (required), alpha, colour, fill, shape, size. You can find the same info from RStudio ggplot cheatsheet (just check Help -> cheatsheets!). Another place for help and examples is the documentation on the ggplot2 page: http://docs.ggplot2.org/current/

Every aesthetic can be mapped to some variable. We can map gender to color:

```
ggplot(data = piaac, aes(x = pvnum1, y = logincome))+
  geom_point(aes(color=gender))+
  geom_smooth()
```



It will automagically create a legend! We can additonally tell it that the point shape is connected to whether the person has children or not (it would not help here much though):

```
ggplot(data = piaac, aes(x = pvnum1, y = logincome))+
  geom_point(aes(color=gender, shape=children))+
  geom_smooth()
```

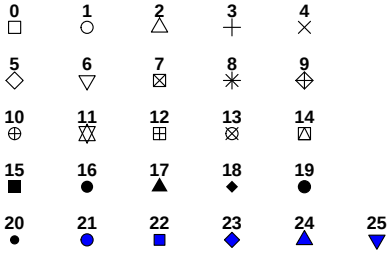


Figure 3: Point shapes

We can set these same parameters outside the **aes()** call to set the values manually (they will then not be connected to the data). We can thus choose the color by giving it the color code, we can choose the shape of the point.

Lets change the previous graph so that the point color is red, point shape is 21 and alpha is 0.3, and lets do it all outside the aes()-parameters:

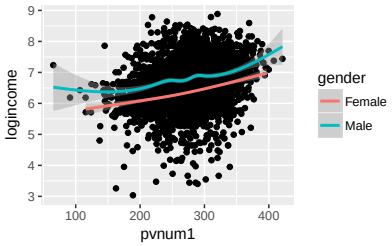
```
ggplot(data = piaac, aes(x = pvnum1, y = logincome)) +
  geom_point(color="red", shape=21, alpha=0.3)+
  geom_smooth()
```

Try to map a variable to an aesthetic outside the aes() – see what happens. It will notice you that *object 'x' not found*. This usually means, that we have forgotten to quote something, in the ggplot it means that we have forgotten to put the variable in the aes()-call.

Groups

Some of the parameters group the data. E.g – lets map the gender variable to color for the **geom_smooth()**

```
ggplot(data = piaac, aes(x = pvnum1, y = logincome))+
  geom_point()+
  geom_smooth(aes(color=gender))
```



We will now see two smoothers – one for men and another for women.

geom_smooth() is a typical way to study the relations in the data, see if it is close enough to linear relationship so one could use the linear modelling tools (like linear regression) etc. Grouping by some other variable helps us understand how the relationship differs between groups.

It will overfit, if the dataset is small. In case we are more or less certain that we are dealing with linear relationship, we can actually tell it to show us the best linear function (adding a parameter **method=lm**).

Your turn:

1) We have the gdpEstimates dataset (`gdp`). Convert the variable date to be Date-type first, if you do not have it already (`gdp$date <- as.Date(gdp$date)`), create a ggplot graph using this data where on the x-axis would be date and y-axis would be latestEstimate, use line as the geom (geom_line()).
2) Add a second layer where y would be connected to firstEstimate and color would be red.

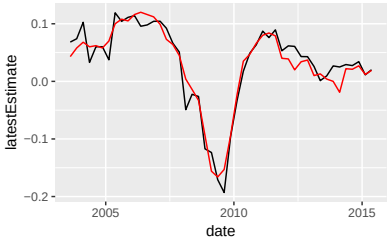


Figure 4: Expected outcome

Facets

One of the most powerful features of ggplot is faceting – slicing the data and producing graphs for or the slices.

Lets look at the correlation of income and numeracy level by education level.

We can do it in the following way, but it gets kind of messy:

```
ggplot(data = piaac, aes(x = pvnum1, y = earnmth, color=edlevel3))
  geom_point()+
  geom_smooth()
```

But we can also create separate graphs by adding one line of code

```
ggplot(data = piaac, aes(x = pvnum1, y = earnmth)) +
  geom_point()+
  geom_smooth(aes(color="gender"))+
  facet_wrap(~edlevel3)
```

There are two ways to facet in R - one is `facet_wrap()`, another is `facet_grid()`

```
ggplot(data = piaac, aes(x = pvnum1, y = earnmth)) +
  geom_point()+
  geom_smooth(aes(color=children), method="lm")+
  facet_grid(gender~edlevel3)
```

Some important parameters for facets are – nrow for facet_wrap() helps you to define nr of rows, `scales="free"` allows facets to show different parts of the coordinate space.

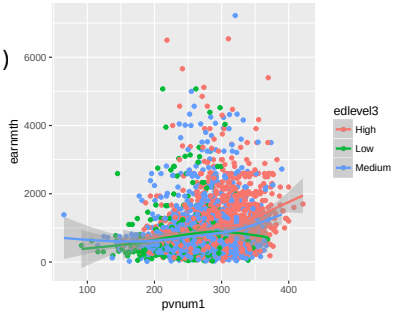
Your turn:

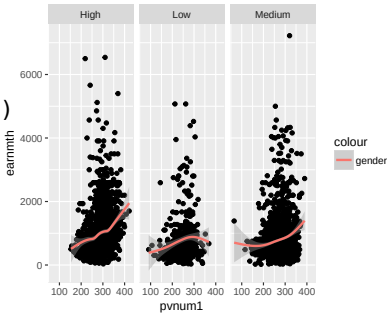
1) Create a graph from piaac where x-axis is mapped to `gender`, y is mapped to logincome and use a `geom_boxplot()` – a convenient way to get a quick overview of a data where one of the variables is categorical. It shows us the median, the 25.-th and 75.-th percentile (top and bottom of the box), so 25% of the observations are above the box, 25% below the box and 50% inside it.

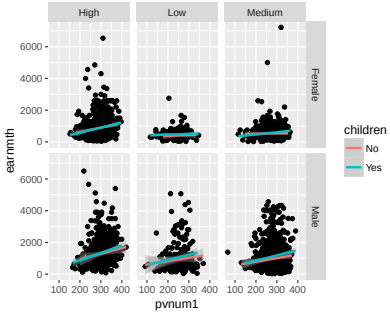
2) Group the data by education level, mapping the variable edlevel3 to color. Do not forget to do it in **aes()**-call!

3) Boxplot shows the outliers as points. Sometimes they disturb us (e.g. if we have the data points visible anyway). We can remove them with parameter **outlier.shape=NA**. Remove them (this parameter can not be inside the aes()-call, but it has to be inside geom_boxplot()-call. Just separate it with a comma.

4) An alternative to the boxplot is a violindiagram, which shows the distribution of data – the wider is the violin at some points, the more are there observations. Create a violin diagram(**geom_violin()**), **but switching the variables** – map education to x-scale, and group by gender.







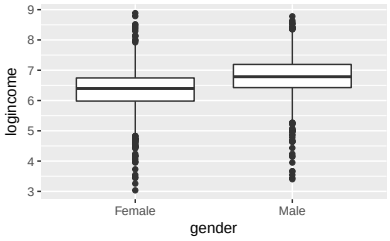


Figure 5: Expected result for the first exercise

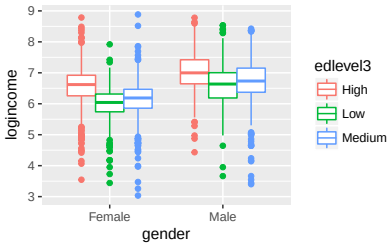


Figure 6: Expected result for the second exercise

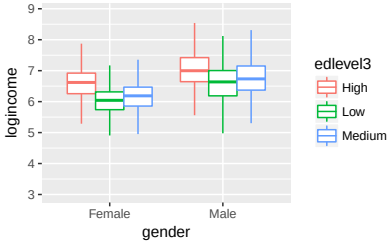


Figure 7: Expected result of the third exercise

5) Add one additional dimension by facets – facet by the variable children using `facet_wrap()`

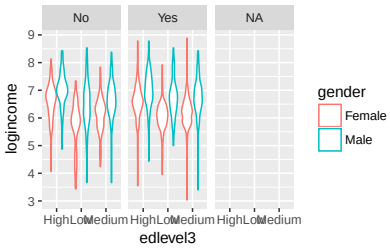


Figure 9: Expected result for the fourth exercise

With this you have managed the main logic of ggplot – how to map variables to graphical parameters, how to add additional parameters etc. Most of the rest is just how to make the graphs publication-ready. We will take a look at this next week.

Theming

You can add predefined themes to the plot:

```
ggplot(data = piaac, aes(x = edlevel3, y=logincome)) +
  geom_jitter(alpha=0.3)+
  theme_minimal()
```

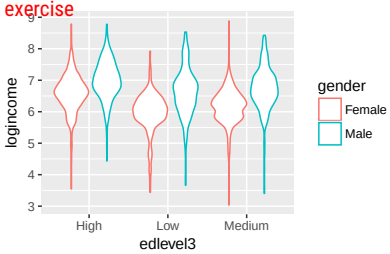


Figure 8: Expected result for the fourth exercise

Ggplot comes with `theme_light()`, `theme_classic()`, `theme_bw()`, `theme_minimal()`, `theme_dark()`, `theme_grey()`, but you can create your own themes and some are available in the packages ggthemes.

Some additional info for ggplot graphs: R Cookbook:
http://www.cookbook-r.com/Graphs/

Some additional themes:
https://cran.r-project.org/web/packages/ggthemes/vignettes/ggthemes.html

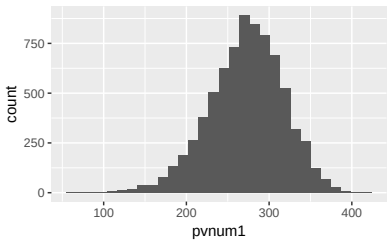
Ggplots theme system: http://docs.ggplot2.org/current/theme.html

Currently the following book is available for free on the internet: http://serialmentor.com/dataviz/

Some additional geoms

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("piaacest.csv")
piaac$logincome <- log(piaac$earnhr)
ggplot(data = piaac, aes(x = pvnum1)) +
  geom_histogram()
```

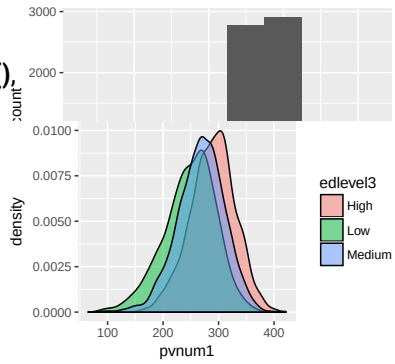


It creates 30 bins by default, we can change it with a **binwidth**-parameter:

```
ggplot(data = piaac, aes(x = pvnum1)) +
  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 = pvnum1))  +
  geom_density(aes(fill=edlevel3), alpha=0.5)
```



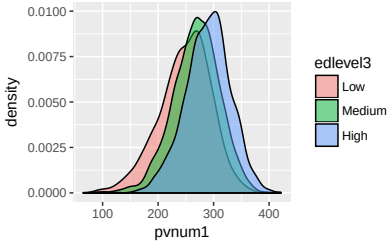
We are seeing a usual problem here – the levels of education are in the alphabetical order. Lets change them to be in a logical order. You can do this with factor variables:

```
levels(piaac$edlevel3)
```

```
## [1] "High"   "Low"    "Medium"
```

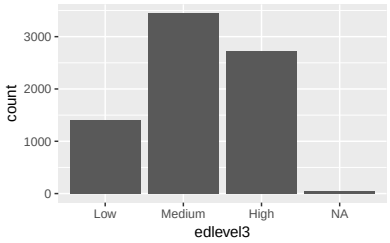
```
library(forcats)
piaac$edlevel3 <- fct_relevel(piaac$edlevel3, levels = c("Low", "Medium",
    "High"))
```

```
ggplot(data = piaac, aes(x = pvnum1))  +
  geom_density(aes(fill=edlevel3), 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:

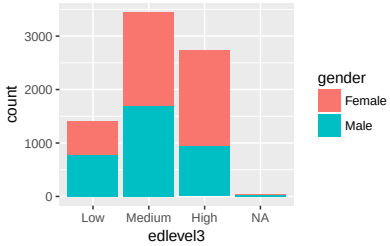
```
ggplot(data = piaac, aes(x = edlevel3)) +
  geom_bar()
```



Position: stack, dodge, fill, jitter, jitterdodge

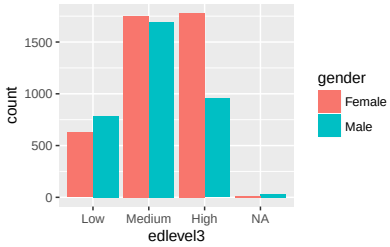
What happens if we group data in the bar chart?

```
ggplot(data = piaac, aes(x = edlevel3)) +
  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 = edlevel3)) +
  geom_bar(aes(fill=gender), position="dodge")
```



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

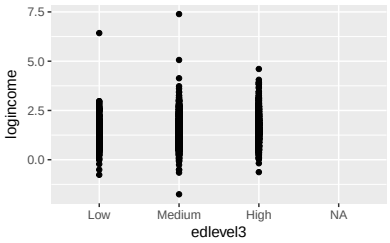
```
ggplot(data = piaac, aes(x = edlevel3)) +
  geom_bar(aes(fill=gender), position=position_dodge(width=0.5))
```

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.

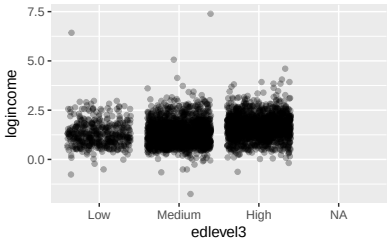
Lets try to visualize the logincome by education level:

```
ggplot(data = piaac, aes(x = edlevel3, 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 = edlevel3, y=logincome)) +
  geom_point(position="jitter", alpha=0.3)
```

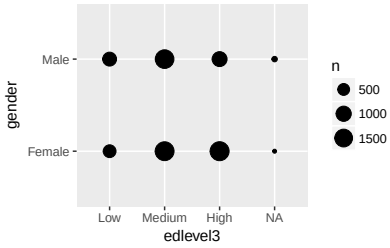


In fact, there is a convenience geom – geom_jitter() that does just this:

```
ggplot(data = piaac, aes(x = edlevel3, y=logincome)) +
  geom_jitter(alpha=0.3)
```

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

```
ggplot(data = piaac, aes(x = edlevel3, y=gender)) +
  geom_count()
```



Additional material – plotly

ggplot creates *publication-ready* graphs, but these are static. But you can turn them interactive with plotly:

```
install.packages("plotly")
library(plotly)
ggplotly(p.gdp)
```

You can save it from the Viewer pane *Export -> save as web page*, or open it from plotly web page. More info at https://plot.ly/ggplot2/.

Additional material: reading data from different formats

For xls-files you can use package called **readxl**. You can define the range of cells from which you want to read the data in, the sheets etc. Take a look at the help file.

If you have data in SPSS, Stata or SAS format, use functions provided by **haven**:

- `read_por()` and `read_sav()` for **SPSS** files (you can write spss files with `write_sav()`)

- `read_dta()` for **Stata** files (`write_dta()`writes Stata files)
- `read_sas()`if you ever encounter a **SAS**-file.

Then there is a package called **foreign**, that supports some other formats (octave, arff, systat etc).

If you are going to work both in Python and R, you might want to use a new package called **feather**[9] – this is a new package, but it helps you to save and load data frames in both R and Python to quickly interchange data between these two.

[9] You can check here for additional information: https://blog.rstudio.org/2016/03/29/feather/.