R Intro for Intro Stats Course

Bodo Winter

So you're reading this presumably because you're taking my intro stats course. This is a little repetition, and some new stuff, on R, the statistical programming environment that we're using throughout the course.

The course will then re-introduce you to R anyway – but it doesn't hurt to get a little headstart!

Why R?

First, I want to justify my choice of using R rather than another software environment (SPSS, SAS, HLM, MPlus). While some of these software tools might – at first sight – seem like they would be much easier than learning an actual programming language, you'll save yourself **A LOT** of time if you switch to R as soon as possible. The reason has to do with the following: When you work with any kind of data, you'll spend most of your time not actually doing sophisticated statistical analysis. Instead, you'll mostly do what people call "data cleaning" or "preprocessing". That is, you'll do a lot of sorting, naming, reordering, finding double entries, finding missing entries, restructuring your table – all that kind of stuff that happens before you do any fancy stats. Now, when it comes to preprocessing, all the other software packages suck. Their scripting capabilities are limited or cumbersome and so if you'd work with these tools, you'd have to do the preprocessing anyway using a different software tool. So why not have it all in one?

If you're linguist, you need R even more. That is because R gives you powerful text processing and corpus analysis tools – combined with all the statistical analysis capabilities that it has anyway!

There's a host of other reasons to choose R over anything else: It's free. It's cross-platform. It has no known bugs. And it has excellent implementations for the models that we'll use in this course. Finally, R has a great community with lots of reference materials and tutorials available. Plus, it looks like R is soon going to be the most-used statistical software (if it isn't already).

Now, one disclaimer: Some participants in the stats workshop might not actually want to use R or do the statistical analyses themselves – and that's o.k.! However, I still think that you should give this tutorial a try because you'll be able to follow the applications better, and because your mind will expand as the result of learning just a tiny bit of programming.

So, let's do it!

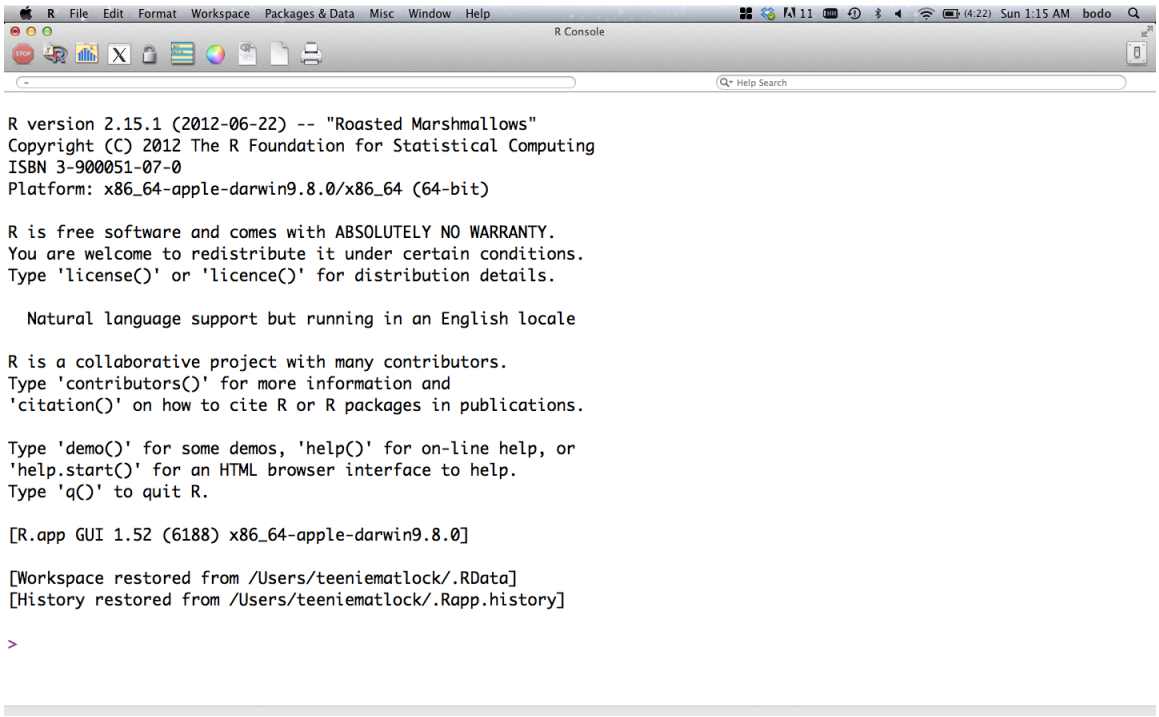
Installing R

The first step is to install R. Go to http://www.r-project.org/. Then, go to the section "CRAN" (short for "Comprehensive R Archive Network"). Choose a server in your country. Then download R for Mac, Windows or Linux (whatever your platform is). Once downloaded, double click on the executable, go through all the installation steps and open R – here we are! (Sorry the pun.)

IMPORTANT: You have to actually follow these steps… otherwise it won't stick!

R as a calculator

Once you opened R, you should see something like this:



The thing in the bottom left (">") is called the prompt – it's R telling you that it's ready for input. Now, R is basically just an overblown calculator. So let's type in something that we might wanna calculate (and press ENTER afterwards):

```
2+2
[1] 4
```

In this tutorial, blue is what I typed in. So, after every blue line, press ENTER. Black represents R's output. Everything that's blue you should type for yourself… and

everything that's black should match what you have on your screen. Ignore the `[1]` for now – it'll become relevant later.

Playtime

As your first exercise, play around for 5 minutes or so with different mathematical operators. There's "+", "-", "*" (multiplication) and "/" (division). Put in different numbers to get familiar with these. Once you've done that, check out "^" (power). So "2^3" would be number two to the power of number three (two three times multiplied by itself).

Spaces and brackets

R ignores spaces. So…

```
2+2
[1] 4
```

… and …

```
2 +   2
[1] 4
```

… lead to the same output. Keep this in mind! Sometimes, adding spaces might help making your code more readable.

Another thing that's important for readability but also for making sure R is interpreting your input correctly is bracketing. Remember brackets in mathematics? They determine the order in which different operations are supposed to be executed. So, for example, compare the following two inputs and outputs:

```
(2+3)/4
[1] 1.25

2+(3/4)
[1] 2.75
```

Makes sense?

A bit more math

Now, what we did above was basic arithmetic. In math that means to apply different operations (addition, subtraction etc.) to numbers. In R that means to apply different *functions* (addition, subtraction etc.) to *objects* (which in this case are numbers). So addition is a function in R, and so is subtraction, multiplication, division, power. They are all functions.

In R, these basic mathematical functions are, however, relatively unusual in their appearance. Most functions in R look like this:

```
sqrt(4)
[1] 2

log(4)
[1] 1.386294

exp(4)
[1] 54.59815
```

The first one is the square root function. The second is the logarithmic function. The third is the exponent function. For now, just recognize that this is a different way of writing functions in R: First you write the command (usually a bunch of letters, some kind of abbreviation) and then in brackets the *argument(s)* that you feed to the function. In this case, we had three different functions and always the same argument (=4).

Note that this is the more usual way of writing functions in R.

Variables

R being a programming language has the advantage that you can store numbers into variables. Remember variables from school mathematics?

Type in:

```
mynumbers = 4
```

Strangely, you didn't see an output this time. R didn't print out anything for you. That is because you "saved" the value "4" in the variable named "mynumbers". Or, in other words, you assigned the value "4" to the variable "mynumbers".

So let's unpack these ideas. "mynumbers" is just a name. Think of this as a variable in mathematics, like "x=3" where then people would go on talking about "x" but know that this variable "contains" the number 3. And just like in mathematics, you could use any name for a number that you want (in principle, although there are

conventions). Say, “x”, “y”, “theta”, “pi”. These names are basically arbitrary, chosen by humans for their convenience.

Variable names in R can be almost anything. Letters are allowed, small and capital. Numbers are allowed too, but not at the beginning of a variable name. Not all special characters are allowed, but I use a lot of dots and underscores in my variable names (which works).

To create variables you just put the name (whatever name you chose) on the left, and then an equal sign and the value to store. This is how we did it above. An alternative way of doing this is the following:

```
mynumbers <- 4
```

This makes more explicit the idea that you somehow store “4” in the variable “mynumbers”. Now, to look “into” the variable, just type its name and press ENTER.

```
mynumbers
[1] 4
```

You should see “4”, the value that you stored in there. So, by just re-typing the name, you can look up what’s in a variable.

By creating a variable, you essentially create an object in the R environment. You can look at that which objects you have available in your environment by typing in the following:

```
ls()
[1] "mynumbers"
```

As you can see, you currently only have one object defined – and that’s the “mynumbers” that we specified above.

You can also use the variable name directly in further calculations, for example:

```
sqrt(mynumbers)
[1] 2

mynumbers*2
[1] 8

mynumbers+0.4
[1] 4.4

4/mynumbers
[1] 1
```

```
mynumbers^3
[1] 64
```

Makes sense? We can even combine the a variable with itself:

```
mynumbers*mynumbers
[1] 16

mynumbers/mynumbers
[1] 1
```

And we can "override" the variable with a new version of itself:

```
mynumbers = mynumbers+2
```

If you type in mynumbers now, you will see the number "6" instead of "4". That is because with the above command, you took the variable "mynumbers", added "2" to it and then you reassigned this new value ("6") again to the variable "mynumbers", thus overriding its initial value.

Vectors

Statistics would be pretty boring if we always only worked with single numbers like in our examples above. Statistics is all about data and data usually means having many numbers. So how do we deal with this in R? Let's save a bunch of numbers. Type in the following:

```
morenumbers = c(1,2,3,4)
```

Again, now to see what's "in" this variable, type in its name…

```
morenumbers
[1] 1 2 3 4
```

…and you should see the numbers that you just stored in the variable. An alternative way of saving the numbers 1 to 4 in a variable is:

```
morenumbers = 1:4
```

The colon stands for "all consecutive integers between the first number and the second". So the colon command allows you to specify longer sequences. If you want to, try just typing in "`7:13`", "`-4:32`" etc.

With our "morenumbers" variable, we can also do simple calculations. For example:

```
morenumbers + 1
[1] 2 3 4 5

2 + morenumbers
[1] 3 4 5 6
```

So applying these simple arithmetic operations to a variable that contains multiple numbers repeats the operation for each element in the variable. There's other operations that work on the whole set of numbers, so-called summary functions. Type in the following and see what each one does:

```
mean(morenumbers)
[1] 2.5

median(morenumbers)
[1] 2.5

sd(morenumbers)
[1] 1.290994

min(morenumbers)
[1] 1

max(morenumbers)
[1] 4

range(morenumbers)
[1] 1 4

sum(morenumbers)
[1] 10
```

Isn't it beautiful how self-explanatory these function names are? The first one gives you the mean (the central location of all the numbers), the second the median (the midpoint of all the numbers), the third the standard deviation (a measure of spread for the numbers), the minimum number in your variable, the maximum number, the range (min and max together) and the sum of all numbers.

What we did before when we executed the command "mynumbers = c(1,2,3,4)" or alternatively, "mynumbers = 1:4", was to create a *vector*. "Vector" sounds very mathematical, but it's simply a string of elements (such as numbers). Vectors are essential to computing with R. In R, everything's somehow a vector or rooted in vectors. But remember, it's not a fancy concept: It's just a string of numbers. Above, we created a vector that "contained" the numbers one to four. Used in this way, the word "variable" and the word "vector" are interchangeable.

Now that we know about vectors, we can begin to make sense of that "[1]" in front of every output line above. Type in the following:

```
7:40
[1]  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
[17] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
[33] 39 40
```

The numbers in brackets might vary. This depends on your screen resolution and the font size in your R window. If you don't see any other bracketed numbers, you probably have a really high screen resolution. If that's the case, just create an even longer vector, like 7:100, so you can see at least one line break.

In the output above, the brackets in front of each line indicate that this is the first, the seventeenth and the thirty-third *position* of the vector. So, vectors have positions. The very first element in the above vector (the value "7") is at the first position of the vector. The value "8" is at the second position. The value "23" is at the seventeenth position. The value "24" at the eighteenth. And so on.

So, conceptually, you need to separate a vector's position from the value that is stored at that position.

All the [1]'s that we saw in the many examples above simply indicated that we dealt with vectors that only had one element (at the first position).

Indexing vectors

With this new knowledge, we can do something new and fancy. We can retrieve information from vectors at specific positions. So let's make ourselves a new vector:

```
positionexample = 2:5
```

So, I gave this vector the stupid name "positionexample". And I assigned the numbers two to five to this vector. Printing the vector gives:

```
positionexample
[1] 2 3 4 5
```

Say, you wanted to work with just the number at the second position (the number three in this case). What you can do is this:

```
positionexample[2]
[1] 3
```

Those square brackets are *indexing* a specific position of the vector, namely the second position. The output is a vector that only contains one element, the number 3. Why? Because that's the value that's "stored" at the second position.

You can also retrieve multiple elements of a vector. Try:

```
positionexample[1:2]
[1] 2 3

positionexample[c(1,4)]
[1] 2 5
```

So in the first example, you indexed the vector "positionexample" with a sequence of numbers going from one to two… and that gives you the elements at the first and the second position. In the second example, you used the concatenate operator to concatenate the numbers 1 and 4 with each other. Using this concatenated object for indexing positions gives you the first and the fourth number. Try the following:

```
positionexample[c(1,1,4)]
[1] 2 2 5
```

With this command, you concatenated the value "1" two times to each other, and then the number "4". When you use this to index, you get two times the element of the first position and then of the fourth position. Now try:

```
positionexample[4:1]
[1] 5 4 3 2
```

Within the brackets, you created a vector that is a sequence of consecutive integers from four to one (counting downwards). Using this to index the "positionexample" vector/variable gives you the fourth position, followed by the third, the second and the first. So it basically reverses the order.

The key insight that you should take home from the preceding examples is that the thing within the brackets (that does the indexing for you) is actually a vector itself. So you use a vector to index positions of another vector. The following example drives this point home:

```
positionvector = c(3,2,1,4)
positionexample[positionvector]
[1] 4 3 2 5
```

Here's a good point to stop for a minute and think. Look at the preceding example. See how you assigned the numbers three, two, one and four to a new vector called "positionvector". Then you used this vector to index positions in the "positionexample" vector that we've been working with. The vector called

"positionvector" is thus a name, an object, a variable, that represents the numbers that you entered. You use the numbers that are stored in this vector to *index*, or retrieve, specific positions of "positionexample".

This insight is key. Ponder it: You use vectors to index vectors. Much of the flexibility of R stems from this idea.

Vector classes

Vectors come in different types. So far we've worked with numbers (in fact, only with integers). But data comes in many different forms. For example, we might have the names of the participants in an experiment.

```
mynames = c("Herbert","Detlef","Dieter")
mynames
[1] "Herbert" "Detlef"  "Dieter"
```

We used concatenate again to create a vector. Crucially, we used characters … and we bracketed each character string with quotation marks. The output preserves these quotation marks. Quotation marks are R's signal that you're dealing with character vectors.

So, we're now beginning to understand that the vector concept in R is in fact quite general. We now know that vectors can be vectors of numbers and vectors of character strings. Here's a way to check the class of a vector:

```
class(mynames)
[1] "character"


class(positionexample)
[1] "integer"
```

So, the "`class()`" function tells you the *class* of a vector.

As we move from vectors of numbers to vectors of character strings, the logic of indexing positions doesn't change:

```
mynames[1]
[1] "Herbert"

mynames[2]
[1] "Detlef"
```

```
mynames[3]
[1] "Dieter"

mynames[3:1]
[1] "Dieter"  "Detlef"  "Herbert"
```

But, of course, you can't do any arithmetic with character strings:

```
mynames+1
Error in mynames + 1 : non-numeric argument to binary
operator
```

This doesn't make any sense and rightly so, R prevents you from doing it and gives you an error message. After all, what's the outcome of "Herbert + 1"???

Logical vectors

O.k., so far, we know "numeric" or "integer" vectors, and we know "character" vectors. There's two more classes that you should know (in fact, there's many more… but most of the others are a bit more esoteric and won't become relevant in this course). You should know about "logical" vectors and "factor" vectors. Let's work on these in that order!

```
2==2
[1] TRUE

2==3
[1] FALSE
```

The "==" sign means "equal to" in R. Do you see why they decided to use two equal signs instead of one? The "=" is already used for the assignment operator – that's what we used before to assign values to variables. Because it's already used in that way, the equal sign needs to be something different, hence "==".

You can think of "2==2" as asking the question "Is 2 equal to 2?". In this case, the answer is obviously TRUE. And "2==3" is obviously false.

By the way, "!=" is the converse of "==". Putting the exclamation mark in front of the equal sign means "not equal to". So try the above with "!=". Try a bunch of other numbers.

We can save the results of a logical operation in a vector:

```
mylogicalvector = 2==2
mylogicalvector
[1] TRUE
```

This vector has class "logical":

```
class(mylogicalvector)
[1] "logical"
```

You can also use logical operators on vectors with multiple elements. Try the following commands:

```
positionexample == 3
[1] FALSE  TRUE FALSE FALSE

positionexample != 3
[1]  TRUE FALSE  TRUE  TRUE

positionexample < 3
[1]  TRUE FALSE FALSE FALSE

positionexample <= 3
[1]  TRUE  TRUE FALSE FALSE

positionexample > 3
[1] FALSE FALSE  TRUE  TRUE

positionexample >= 3
[1] FALSE  TRUE  TRUE  TRUE
```

I've introduced some new logical operators that should be self-explanatory. Can you figure out what each one does by comparing what's in the vector to the output?

So you might think: What's all this logical vector stuff useful for? Well, logical vectors can be used for indexing. Before, we were only able to get the values at specific positions… but we had to know the position. Now, we can retrieve things by value rather than just by position. Here's an example of this:

```
positionexample[positionexample == 3]
[1] 3
```

The part within the brackets creates a logical vector of TRUE's and FALSE's. This logical vector is then used to index positions within the main vector. All this is doing is giving you the value for which "`positionexample==3`" is TRUE … which is 3. Note that by using this approach, you didn't have to know the position within the vector in order to retrieve the element. This becomes important when you're

dealing with very long vectors where it becomes difficult to keep track of the exact position of individual elements of a vector.

Now, to see the power of using logical vectors to index numerical vectors, take all of the other logical operators that we used in the above example ("!=",">","<=","<","<=") and combine them with the above statement. Here, I'll give you the first one…

```
positionexample[positionexample != 3]
[1] 2 4 5
```

… now you go on and try all the others!

Factor vectors

Let's learn the last important vector class: factor vectors. At first sight, factors are nasty. A lot of errors happen when people work with factor vectors without knowing that they're working with factor vectors. Among all the vector classes, factors have the most pitfalls. And conceptually, they are the least straightforward. But they are extremely important for all the stats that we work on later. Let's make us a factor vector:

```
mynames = as.factor(mynames)
```

Remember our "mynames" vector from above? If you don't have this any more, you have to re-create it by re-entering those commands. And then use "`as.factor()`" just as in the example above. This function converts vectors into factor vectors. Look at the output:

```
mynames
[1] Herbert Detlef  Dieter
Levels: Detlef Dieter Herbert
```

Note a few things. In comparison to the character vector, the quotation marks have disappeared. And now, there's an extra line which says "levels". These are the unique elements of the vector. Note also that the levels are listed in alphabetical order.

To see the point, let's extend our vector. Say, there were two people in our experiment called Herbert (you don't have to replicate the line break in the example below):

```
mynames = as.factor(c("Herbert","Detlef",
                "Dieter","Herbert"))
```

Note that we combined two commands. First, we concatenated the character strings for our names, making them into a character vector. Wrapped around that is the "`as.factor()`" function which makes these into a factor. So we just did the same as above but two steps in one… and we have one Herbert more.

If you print the object, you should see this:

```
mynames
[1] Herbert Detlef  Dieter  Herbert
Levels: Detlef Dieter Herbert
```

The line that is started by `[1]` replicates the vector as we specified it, in that order. The levels below show the unique levels, so Herbert is mentioned only one time, because two people share the same name. If you're familiar with the linguistic/philosophical distinction between type and token, the following explanation might help: The elements in the vector are the tokens, the levels are the types.

With respect to indexing, factors work just like number or character vectors. For example:

```
mynames[mynames=="Herbert"]
[1] Herbert Herbert
Levels: Detlef Dieter Herbert
```

Changing already existing vectors

We can use our understanding of indexing to also change already existing vectors of all classes. Let's work with numbers first:

```
positionexample[2] = -4
```

Can you think what this command did? Think of it step by step. The brackets single out the second position. Then, you assign the number "negative four" to that position. So now, your vector should look like this:

```
positionexample
[1]  2 -4  4  5
```

You can also re-assign multiple values:

```
positionexample[1:2] = 0
```

This command sets the first and second position to zero.

You can change factor vectors, too:

```
mynames[4] = "Detlef"
mynames
[1] Herbert Detlef  Dieter  Detlef
Levels: Detlef Dieter Herbert
```

But factor vectors are a bit more tricky. You can only change existing values to values for which there are already levels. That's why I changed Herbert to Detlef in the example above, because Detlef already has a level in this specific factor vector. Let's do one more change:

```
mynames[3] = "Detlef"
mynames
[1] Herbert Detlef  Detlef  Detlef
Levels: Detlef Dieter Herbert
```

Note how even though there's no more Dieter in the actual vector, the levels remain the same. (In other words: The type information is still represented, even though there's no more actual token.) Think of the levels as something that's solidified. The levels itself remain in place (you can actually change them, but we'll worry about that later when we do some of the applications in the course).

If you assign to an existing factor vector something that's not already a level… you get a warning message.

```
mynames[3] = "Hans"
Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "Hans") :
  invalid factor level, NAs generated
```

Let's see what happened to our vector:

```
mynames
[1] Herbert Detlef  <NA>    Detlef
Levels: Detlef Dieter Herbert
```

R did not recognize the character string "Hans" among its already existing levels. So it could not interpret this new character string in the context of the "solidified" levels and simply gave the third position an NA, which stands for "not applicable" and represents a missing value.

One simple way to get around all of this is to transform factor vectors to character vectors whenever you want to make changes. This will automatically update the levels of the vector. The function for making a vector into a character vector is "`as.character()`".

Oh, and just to complete things… to convert a vector into a numeric vector, you can use "`as.numeric()`".

Playtime

Try making your variable "mynames" (which now looks a bit ugly) into a character vector and then back to a factor using "`as.factor()`". Make some changes while your vector is of class character. Always check what happens to the levels when you make it into a factor. Also play around with "`class()`".

Now, do factors finally seem really annoying to you? In a way they are… but they are essential for working with all of the more advanced statistical functions that we'll employ later.

Dataframes

So far, we've only worked with one-dimensional vectors. That is, we worked with simple strings or series of numbers. Let's make ourselves a table, so something that's two-dimensional!!

```
mydataframe = data.frame(
     c("Hans","Willy","Frank"),
     c(4,3,6)
     )
```

O.k., this looks a bit more complicated. Let's unpack this. You pick a name, in our case "mydataframe". This is your variable, which you can see by noticing that it's to the left of the equal sign. Then you assign something to this variable. This something is a data frame (basically a spreadsheet, or a big table). The function is called "`data.frame()`". Within the function, I've made the line breaks in such a way that you can see the internal structure of this command (you can have all in one line if you want to). And there are two arguments. First, `c("Hans",…)` (a list of names, concatenated into a character vector). Then, "`c(4,3,6)`", a list of numbers, made into a numeric vector. So you basically "feed" the function two vectors. If you type in the following, you'll see what the above command did:

```
mydataframe
  c..Hans....Willy....Frank.. c.4..3..6.
1                        Hans          4
2                       Willy          3
3                       Frank          6
```

This is a table, or, in R's language it's a data frame (again, basically, a spreadsheet). Each of the two vectors that were the arguments in the function above are now

columns of this new data frame. But the column names are really ugly. That's because you didn't give R any column names, so it just "thought up" something that's made up of the elements of the vectors. Let's change the column names:

```
colnames(mydataframe) = c("participants","ratings")
```

The above command retrieves the column names and reassigns two new names to it. (If you want to understand what "`colnames()`" does, try "`colnames(mydataframe)`" without assigning anything to it.) Now, let's look at the data frame again:

```
mydataframe
  participants ratings
1         Hans       4
2        Willy       3
3        Frank       6
```

This looks much better, yes? There's one column called "participants" and one called "ratings" (maybe these are grammaticality ratings in a linguistic experiment). And there's three rows that are named 1, 2 and 3.

Indexing data frames

We can index data frames just as before, by position. However, because a data frame is essentially a two-dimensional object (one dimension being the rows, another dimension being the columns), your indexing needs to be two-dimensional as well. Try these commands:

```
mydataframe[1,1]
[1] Hans
Levels: Frank Hans Willy

mydataframe[2,1]
[1] Willy
Levels: Frank Hans Willy

mydataframe[1,2]
[1] 4
```

Do you get it? In case you didn't: The first element in the square brackets is always the row number. The second element is the column number. So the first command above indexed the first row and first column (retrieving "Hans"), the second command indexed the second row of the first column (retrieving "Willy"). The third command indexed the first row and the second column (retrieving "4").

If you want a specific row, but all columns, leave the column index blank:

```
mydataframe[3,]
  participants ratings
3        Frank       6
```

This gives you the third row but all columns. Or, you can retrieve one specific column and all rows:

```
mydataframe[,2]
[1] 4 3 6
```

Or, all columns and all rows:

```
mydataframe[,]
  participants ratings
1         Hans       4
2        Willy       3
3        Frank       6
```

Which is a bit redundant, as simply typing "`mydataframe`" gives you the same.

Finally, let's talk about one other way of indexing data frames, and that is, by column name. Remember how above, we named the first column "participants" and the second column "ratings"? We can use this information for retrieving specific columns:

```
mydataframe$participants
[1] Hans  Willy Frank
Levels: Frank Hans Willy
```

And, just as before, you can use the elements that you retrieved for further computations as in the example below, where we take the mean of all ratings:

```
mean(mydataframe$ratings)
[1] 4.333333
```

Or even, index something of the thing we just indexed:

```
mydataframe$ratings[3]
[1] 6
```

Loading in data

O.k., so far so good. All of this has been rather toy-ish. When you do stats, of course, you're going to be working with real data. So let's work with an actual data set. Download the following file:

http://www.bodowinter.com/tutorial/politeness_data.csv

So, how do we load in data so we can use it in R?

First, you could download the data from within R, using the following command (again, no need to replicate line breaks):

```
download.file(
     "http://www.bodowinter.com/tutorial/politeness_da
     ta.csv",
     destfile="politeness_data.csv"
     )
```

This will have put the data file right into the current working directory of R. You can then simply use this command, where I chose to name the data frame object "politeness_data":

```
politeness_data = read.csv("politeness_data.csv")
```

This will have created a new data frame. Check by typing in its name:

```
politeness_data
```

Now, usually, you would probably not download a file immediately from the web, but you would have stored it somewhere on your computer. You would also simply use the command "`read.csv()`" (for .csv files) or "`read.table()`" (for .txt files). However, the usage of these commands is contingent on having the file in the current working directory. To find out where R thinks your working directory is right now, type in the following:

```
getwd()
[1] "/Users/bodo"
```

So, for me, my current working directory is simply "/Users/bodo". Here, platform differences will emerge. You will see something like the statement above only if you use Mac or Linux. If you use a Windows PC, the statement will go "C:\…"

To change the working directory, you can use the command "`setwd()`". If you don't want to avoid a lot of typing, you can copy and paste the folder that you're interested in from the "properties" tab of your folder (usually retrieved by right click). Alternatively, you can specify the full directory and file name in

"`read.csv()`" or "`read.table()`". For example, as in the following command (which only works on my computer because only I have those folders):

```
setwd("/Users/bodo/Desktop/research/politeness/politeness_production_study_acoustic_correlates/final_analyses/")
```

So, it's good knowing the folder structure of your computer very well!!

Typing in that are necessary for loading in data or setting your working directory takes a lot of time and is an error-prone process. Therefore, for the sake of this course, I recommend saving all data files in one folder and saving a "`setwd()`" command that directs to that folder. Then, you don't have to re-type this again for all the exercise.

You might wonder: Why have I only shown you how to load in .csv or .txt files… what about Excel files (.xls or .xlsx)? Well, my answer is: You shouldn't use Excel files. It's easiest to work with .csv, as Excel can open .csv files quickly, but you can also more easily load them into R.

Big Playtime

Now that you have a data frame object called "`politeness_data`" (or however you named it), try the following commands and see what they give you "`head(politeness_data)`", "`tail(politeness_data)`", "`summary(politeness_data)`" and "`str(politeness_data)`". These are all different ways of displaying information (or parts of it) from the data frame.

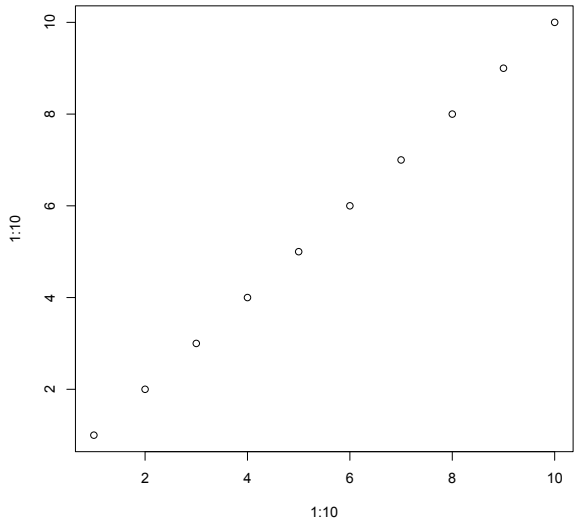
Then, check your knowledge of R commands and indexing above. Play a bit around with different indexes. Try to index specific row/column combinations. Try to index columns by name.

Plotting

A final thing we have to cover is visualization. R has beautiful graphics, but we're only going to work with the basics here. Type in:

```
plot(1:10,1:10)
```

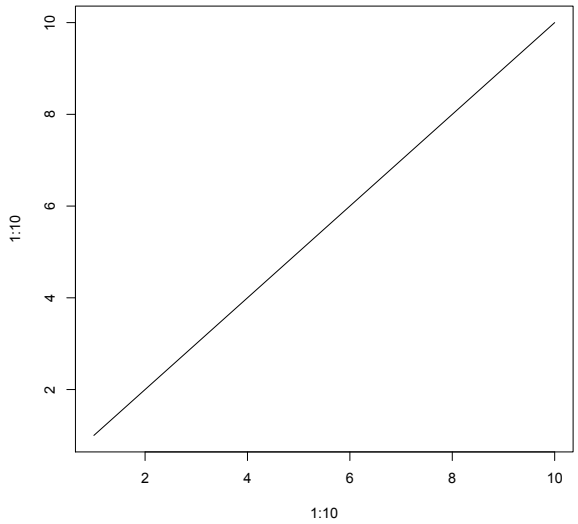
You should see this:



What is it? Well, an ugly plot of the numbers one to ten plotted against the numbers one to ten. That's what the two things separated by the comma in the above command mean. The first set is the x-coordinates, the second the y-coordinates.

You can make this into a line by adding an argument, as I did in the following command:

```
plot(1:10,1:10,type="l")
```



Try what happens if you specify "`b`" instead of "`l`". Can you guess what each abbreviation means by looking at the plot? What happens if you specify "`n`"?

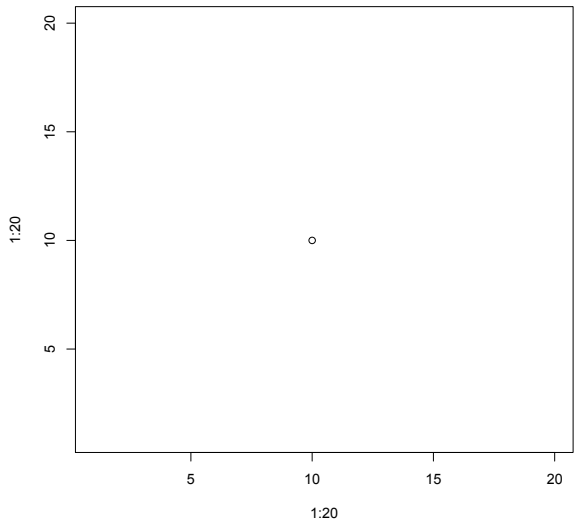
Extended Playtime Exercise

This playtime will be a bit tricky and requires more exploration. I want you to draw a smiley – it doesn't have to be a very beautiful one. Can you do that? Start by specifying an empty plot. Think of this as your canvas. You can do this using the following:

```
plot(1:20,1:20,type="n")
```

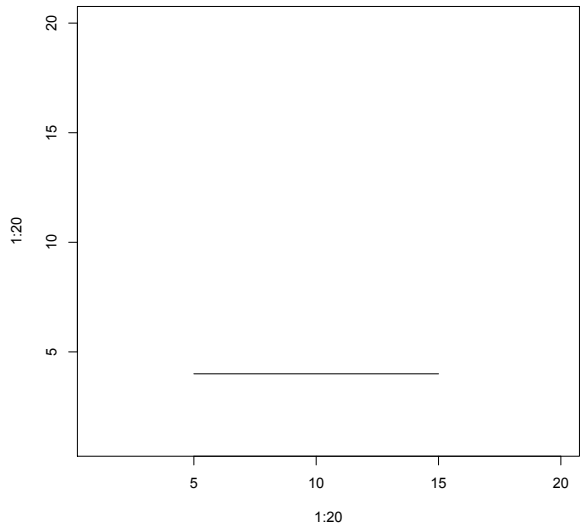
I increased the x- and y-coordinates to go from 1 to 20. This will give you more numbers to play with. You can plot individual points by using the command "`points()`". This command takes two arguments (separated by commas). The first one is the x-coordinate, the second one the y-coordinate. Note that you have to have the plot window opened to be able to plot points into it. So you have to use "`plot()`" first, then "`points()`". Below, I plotted a point at the coordinate x=10 and y=10 (right in the middle of the plot):

```
plot(1:20,1:20,type="n")
points(10,10)
```



You can plot lines by using the "`segments()`" command, as I did below:

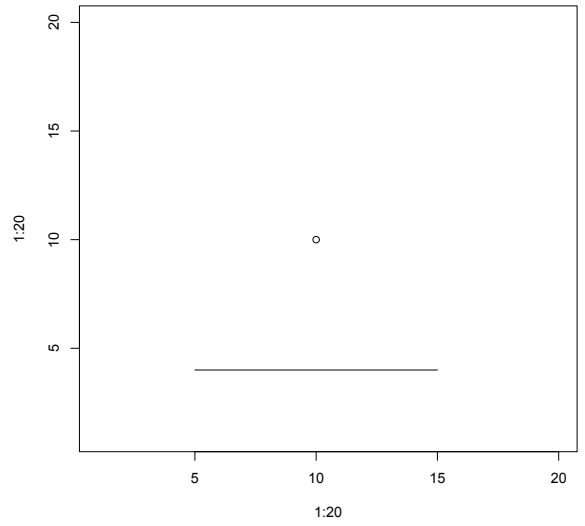
```
plot(1:20,1:20,type="n")
segments(5,4,15,4)
```



The "`segments()`" command takes on four different numerical values. In the above example, the first x-coordinate is at 5. The first y-coordinate at 4. The second x-coordinate is 15. And the second y-coordinate 4. Think of the first *x,y* coordinates as the origin of the line and the second *x,y* coordinates as the end point. The order of the arguments has to be in that order (first x-coordinate, first-y, second-x, second-y).

In the commands below, I drew a point and a line segment combined – the start of my personal smiley (you can do better than that!):

```
plot(1:20,1:20,type="n")
points(10,10)
segments(5,4,15,4)
```



You can be creative with points and line segments. Go for it!

It will become useful to you store all your code in a text file or a text editor and then copy-and-paste it into R… because the plot-construction is sequential (first you need to create a plot, then the segments and points) it'll be difficult to revise your plot without having all the code somewhere outside the R window.

Oh, and just in case: One thing that I haven't explained to you yet is that if you're in R, you can use the up or down arrows to cycle through the history of used commands.

Seeking help

Now, this was just a very very preliminary intro to R. You will very soon encounter limitations and/or problems. I cannot prepare you for everything in such a short time.

Part of knowing how to program is knowing how to seek help. The first thing that you might try is the "`?`" function. For example:

```
?mean
?sum
?head
```

This gives you the help page for each function. It's always worth copying-and-pasting the examples from the bottom of these help pages into R to see how the respective functions work.

Then, you can always google. This should be your next step if you don't find the solution using the "`?`" function. For error and warning messages, you will almost certainly find a relevant solution by just copy-pasting the message from R into the google search bar.

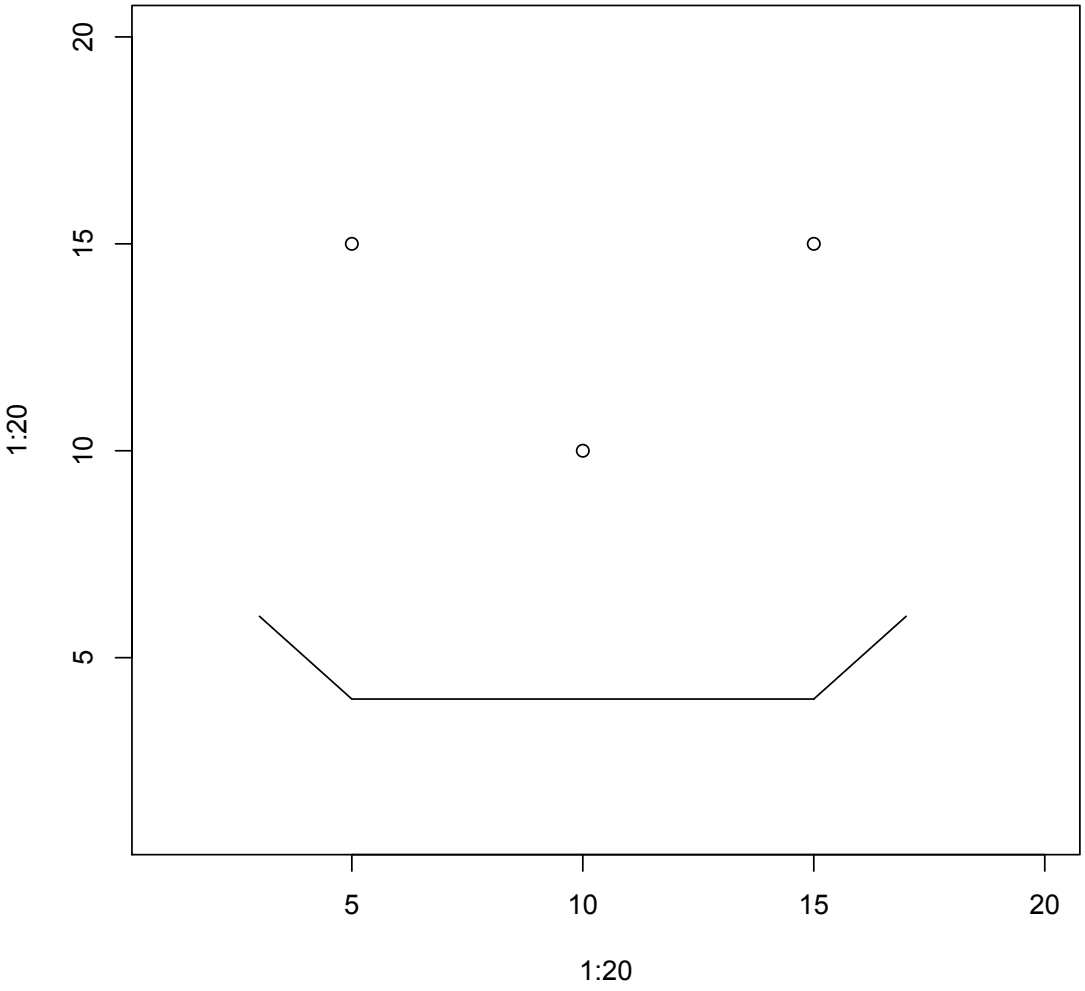
Finally, you can contact people (in forums or via email). However, when you do so, make sure that you clearly formulate your problem, provide enough information, and simplify your problem to the essentials. There's an art to asking for help when it comes to programming. We will discuss some of that in this course.

R Workflow

Finally, here's a few personal tips for better "R workflow". When you're working on larger projects, you shouldn't use the interactive command line much. Try to get yourself into using a text editor, from where you work on your scripts. These scripts contain all the commands that you want to use *in the order in which you want to use them*. To execute the commands, you can simply copy and paste them into R from the text editor. There are some specialized R editors. Mac has an in-built script editor for (that's how I work on my scripts). And many people like RStudio (I don't, however). For Windows, a supplementary tool such as Tinn-R is recommended.

In your scripts, you can add comments by using "`#`". Everything behind that pound sign is ignored (R doesn't "interpret" it). Try to use comments to structure your scripts and remind yourself of what you're doing. Every programmer has made the experience where they're looking back at some code or some analysis that they've done a few months or even years ago and it took them ages to figure out what's going on. Comments can help yourself when you revisit old projects later on. They also make your code more understandable to outsiders.

You should develop some other useful habits, for example, keeping variable names short and clear. There's a trade-off between clarity and length. Try to be as short as possible (thus requiring less typing) while still being transparent. Then, I, personally, like to re-use the same variable names for different data analyses. In a lot of my projects, I only have to work with one data frame… and so I developed a habit of calling that data frame "`xdata`" for every project that I work on. This saves thinking time and typing time.


1:20
20
15
10
5
5
10
15
20
1:20