Tutorial One: Variables

Numeric variables and vectors

Most analyses and code will deal with numbers and these are introduced here.

Simple Math

You could use R as a simple calculator by typing in the exact formula's and checking the results yb hand. For example, if we have some starting position A, and two other positions B and C, and we want to see if B or C is closer to A, we can just use the pythagorean theorem twice, and compare the results. Let's say A is at (2, 3), B is at (3, 0) and C is at (4, 5).

First we calculate the distance between A and B:

```
sqrt((3-2)^2 + (0-3)^2)
```

If you press the green 'play' button at the top right of the chunk of code, you should see that the distance between A and B is 3.162278...

Then we calculate the distance between A and C, complete this yourself:

```
sqrt(( - )^2 + ( - )^2)
```

You should see a distance of around 2.828, so that C is closer to A than B is to A. Of course, for just two points this is easy and you probably shouldn't bother writing code. But let's say you have a list with 100 cities and towns, and want to see all their distances to the capitol, it becomes worth your time to automate this, and then usually, you would use variables.

What Are Variables?

A short and incomplete answer is: variables are "containers" for data. They resemble the A's, B's, X's and Y's from high school math, in that they have some form of name, that represents a value. The names can be 'A' or 'X', but in code it is better to use descriptive labels, like 'slope' or 'age'. Such labels can refer to a single number, but also to vectors or matrices or other types of variables that can hold more than a single value. There are formal definitions, but for a practical understanding it is better to just try out how they work yourself, which is what you will do here.

Numeric variables

Let's put the starting positions coordinates in the variables startX and startY, complete the code yourself:

```
startX <- 2
startY <-
```

You see the `<-` operator, called 'gets', is used to assign a value to a label, two times. You can check what the values are by typing the variable's names at the command line again:

```
startX
startY
```

If you did not complete the previous chunk of code, the above chunk will generate an error, because startY has no value yet. Go back and fill in the value to complete the rest of this notebook.

Numeric vectors

The start position coordinates are two single numbers. For the points B and C you will use a vector of X coordinates and a vector of Y coordinates. You create a vector with the `c()` function. I've already created the vector of X coordinates below, and you should complete the vector of Y coordinates:

```
posX <- c(3, 4)
posY <- c(0,  )
```

This gives two vectors, each with two numbers in them, in the order that they were put in the vector:

```
posX
posY
```

Vectors are a good way to keep many numbers organized in a specific order.

Vectorized calculations

Vectors can also be used in calculations, where the operation is applied to all elements of the vector.

Try adding a number to the `posY` vector:

```
posY +
```

Try multiplying `posX` with a number:

```
posX *
```

Try squaring `posY`:

```
posY
```

You can even use two vectors in one calculation. See what happens when you add `posY` ro `posX`:

```
posX +
```

Or multiplying the two vectors:

```
posX * posY
```

So apart from storing data in an ordered fashion, another advantage of vectors is that you can do vectorized calculation. For example, you can calculate all the distances from towns to the capital at once. Using the same pythogorean theorem, complete this chunk of code:

```
sqrt((posX-startX)^2 + ( - )^2)
```

Again, with only two positions, we can easily see which distance is shorter, but if we want to look at our list of 100 cities and towns, we can store the distances in another variable, like this (complete this code):

```
dist <- sqrt(  + (posY-startY)^2)
```

And we can look at the list of distances by typing `dist`:

```
dist
```

And see which distance is shorter.

Here is an example with longer vectors, that are created by random sampling from a set of possible coordinates (we'll pretend to live in Wyoming, which is not a country, but it is almost rectangular).

```
posX100 sample(x=c(0,1,3,4,5,6,7,8,9,10,11,12,13,14,15), size=100, replace=TRUE)
posY100 <- sample(x=c(0,1,2,4,5,6,7,8,9,10,11,12,13), size=100, replace=TRUE)
```

With this random sampling, some cities might end up on the exact same spot, but to prevent distances to the capitol to be 0, the coordinates (2,3) are left out of the lists (complete the first line).

Let's look at the X coordinates:

```
posX100
```

Notice that at the start of each line, there is a number in square brackets. This is the index of the first thing that follows on that line. All the other numbers are the actual content of the `posX100` numeric vector.

We can check how many items there are in the vector with the `length()` function:

```
length(posX100)
```

There are other ways to make vectors, but using `c()` is very common. Apart from putting in the numbers or other data yourself, there are some ways that you can fill a vector automatically:

```
c(1,2,3,4,5)
c(1:5)
```

Using `seq()` you can do some more advanced things:

```
seq(from=1, to=5, by=1)
```

Use the above code to generate a vector with all even numbers between 1 and 11, which would be the same as whatyou get from c(2,4,6,8,10):

```
seq(   )
```

You can do much more with this, but for now it's good to know `seq()` exists.

Vector indexing

While vectors are very useful, sometimes you need to look at a single number, or perhaps a subset of the vector. That's when you need indexing or subsetting. If we only want to see the third number in this vectors, we use the index 3 with square brackets to do so:

```
posX100[3]
```

If we want to see the fifth to the tenth number, we can also use indexing with square brackets:

```
posX100[5:10]
```

Indexing is a great tool that will be used later on. Let's practice it a bit more. First we will get all the even indexes from 2 to 10:

```
posX100[c(2,4,6,8,10)]
```

Or the combination of the third number and the fifth to tenth, fix the error in this code:

```
posX100( c(3,5:10) )
```

Calculations with vectors

Now let's calculate all of these cities distances to the capital:

```
dist100 <- sqrt((posX100-startY)^2 + (posY100-startY)^2)
dist100
```

So in 1 line of code, you can calculate 100 distances! Or even more... But now it is not so easy to see the shortest distance to the capital anymore. Of course there is a solution. R can give you the minimum or maximum from a vector, or both, or a summary of the numbers in the vector:

```
min(dist100)
max(dist100)
range(dist100)
summary(dist100)
```

Notice that there are indices even when there is only one number to show.

So you can see what the smallest (or largest) distance was, but which town or city is closest to the capital? There's a trick for that too:

```
which.min(dist100)
```

And `which.max()` works similarly.

Check if this has the same minimum distance as reported before, by completing this code:

```
dist100[which.min(  )]
```

Is that right?

Floating point numbers

By default, R will create numeric variables that can hold floating point doubles. These can be very large or very small and they be fractions, like π:

```
pi
```

But they are not infinitely precise, so that sometimes you get rounding errors:

```
round((.575*100),0)
round(57.5,0)
```

Which also means that the number π that R, or any other programming language, provides is a good approximation but not the exact value. It's also another reason to always check if your calculations make sense.

Integer numbers

If you only want round numbers, you can use `as.integer()` to create an integer number from any other number:

```
as.integer(pi)
```

In accordance with the Indiana Pi Bill. But if you then do some calculations with the numbers, the output will again be numeric, even when it could be an integer:

```
a <- as.integer(pi)
class(a)
class(a + 3)
```

However, if you have very large datasets that can be effectively represented by integers, it could still make sense to use them, as they take up a little less computer memory, and some computations can be done a little faster with them.

Character variables and lists

Not all data consists of numbers, or just vectors of numbers. Here we will briefly introduce character variables and how they can be used as keys in lists. But most data we deal with will be numeric, and in data frame format. Data frames are the topic of the next tutorial.

Character variables

Sometimes data does not consist of numbers, but text. In that case, character variables can represent the data in R:

```
my_text <- "hello world!"
my_text
```

Numbers can also be converted to character variables:

```
as.character(pi)
```

When you `paste()` character variables, the two bits of text are simply concatenated:

```
paste(my_text, as.character(pi))
```

By default, `paste()` adds a space between the character variables, but this can be changed. If you don't want anything in between the pieces of text, you can use `paste0()`. Both function will convert any input that is not already a character variable to one – if possible:

```
paste0(my_text, pi)
```

There are some operations you can do on character variables. Since they can be seen as a vector of single characters, you can use a kind of indexing on them. Correct this code chunk so it outputs the word 'hello':

```
substr(my_text, ,10)
```

Similarly you can find certain letters:

```
grepRaw('l', my_text)
grepRaw('l', my_text, all=TRUE)
```

Change the code to find the location of the whole word 'world':

```
grepRaw('', my_text)
```

You can split a character variable into several character variables. The split will occur at specified characters. Split `my_text` into the words 'hello' and 'world!' by changing this chunk of code:

```
strsplit(my_text, split='l')
```

Another example of an operation on a string is to capitalize all letters in the string:

```
toupper(my_text)
```

But I don't recommend screaming.

You can also put character variables in a vector:

```
ducklings <- c('Huey', 'Dewey', 'Louie')
```

Many more operations can be done on text data as well, but in this introduction, we'll only mention one more, and that is how to construct strings with `sprintf()`. This function is very useful in reading in data, generating messages for user feedback or putting labels on figures, even when working with mostly numeric data. It will be used like that in future tutorials.

Sprintf

Sprintf **prints** **f**ormatted **s**trings. That means you can assemble a character variable, or "string"" from pre-specifed kinds of pieces. There are three kinds of pieces you will practice with here: 1) other strings, 2) integers, 3) floats.

Without further arguments, `sprintf()` simply returns the string you give it:

```
sprintf('This is a string...')
```

The most straightforward thing to add is another string. The places where something gets added are marked with a percentage sign, plus at least one character that indicates what kind of thing you want to insert at that point. To add a character variable or string, you need `%s` with an s for string.

```
sprintf('Obligatory greeting: %s', my_text)
```

This can be done for multiple strings as well, that is, `sprintf()` is also vectorized:

```
sprintf('Duckling: %s',ducklings)
```

Let' ssay we want to number the ducklings as well, so we'll first need to create a vector with the numbers of the three ducklings:

```
duckling_numbers <- c(1:length(ducklings))
```

You can use that in the sprintf command as well. These numbers are floating point numbers, but you want it as an integer. You can tell sprintf to represent the numbers as integers with `%d` with a d for digit.

```
sprintf('Duckling %d: %s',duckling_numbers,ducklings)
```

You can assign the output to a new variable, which will be a vector with three character variables. Correct and complete the following chunk of code:

```
duckling_vector <- sprintf('Duckling %d: %s',duckling_numbers,ducklings)
class( )
len(duckling_vector)
```

Using these two options for sprintf you can already automatically generate most file names from a project that you would like to analyze. For example, if you number your participants and each of them does two conditions that get saved into seperate files with specified filenames, you can use %s to insert the condition names and %d for the participant numbers.

In some figures you would like to show some descriptive numbers, for example the average age of your participants. Let's another bit of code already calculated that for you to be 22.8333333333... (they are all university students), but here we'll just assign it manually:

```
mean_age <- 22.833333333333333333
```

All those three's should not end up on your figure though, so you decided to limit it to two decimals. You can now use `sprintf()` with the `f%` which stand for fixed point in this case, as you can more or less fix where the decimal point ends up:

```
sprintf('mean age: %.2f', mean_age)
```

Matrices

There is one more variable type that you should know about: matrices. They are the matrices from matrix algebra, so they are kind of rectangular tables that hold numbers (not strictly true in R). They can be created with the `matrix()` function:

```
my_matrix <- matrix(data=c(1:9),nrow=3)
my_matrix
```

By giving it a vector of 9 numbers and specifying that there are 3 rows, the function figures out that there should also be 3 columns in order to fit all the numbers. You can see that the matrix is filled by column, that is: the first column is filled first, and then the second column, and so on. You can change all this behavior:

```
matrix(data=c(1:9), nrow=2, ncol=6, byrow=TRUE)
```

This code complains, but it *does* create a matrix of the specified shape, filling in the first row first, then moving on to the second row. When it runs out of numbers it just starts over again at the beginning of the input vector.

We can go back to the first matrix, and use indexing to retrieve part of the content of the matrix:

```
my_matrix[3,3]
```

You can see that we now need two indices, one for the row, and one for the column. We can also retrieve an entire row at once, by not specifying the column index:

```
my_matrix[2, ]
```

Change this code chunk to retrieve only the second column:

```
my_matrix[,]
```

We can even get a sub-matrix from a matrix, by specifying a range of numbers for both the row and column indices:

```
my_matrix[2:3, 2:3]
```

You can do simple math with matrices, just like with vectors:

```
my_matrix + 2
```

Most importantly, matrices can be used to do matrix algebra, which will sooner or later pop up in your analyses when you grow more advanced in using R. Here you won't go deeper into this topic. However, the row-by-column indexing will come in useful when dealing with data frames, the topic of the next tutorial.